Introduction to neural networks

Second lecture

Lecturer: A. Taylor

Version of the slides: February 17, 2026.

Thanks Erwan Scornet (original slides) and Pontus Giselsson

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Reminder: supervised learning

Problem setup:

- input measurement $\mathbf{X} \in \mathcal{X}$,
- output measurement $Y \in \mathcal{Y}$,
- $(\mathbf{X}, Y) \sim \mathcal{D}$ with $\mathcal{D}$ unknown,
- training data : $\mathcal{D}_n = \{(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_n, Y_n)\}$ (i.i.d. $\sim \mathbb{P}$).

Reminder: supervised learning

Problem setup:

- input measurement $\mathbf{X} \in \mathcal{X}$,
- output measurement $Y \in \mathcal{Y}$,
- $(\mathbf{X}, Y) \sim \mathcal{D}$ with $\mathcal{D}$ unknown,
- training data : $\mathcal{D}_n = \{(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_n, Y_n)\}$ (i.i.d. $\sim \mathbb{P}$).

Examples:

- $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \{-1, 1\}$ (classification, sometimes $Y \in \{0, 1\}$),
- or $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \mathbb{R}$ (regression).

Reminder: supervised learning

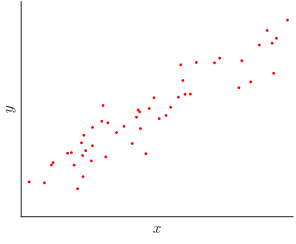
Problem setup:

- input measurement $\mathbf{X} \in \mathcal{X}$,
- output measurement $Y \in \mathcal{Y}$,
- $(\mathbf{X}, Y) \sim \mathcal{D}$ with $\mathcal{D}$ unknown,
- training data : $\mathcal{D}_n = \{(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_n, Y_n)\}$ (i.i.d. $\sim \mathbb{P}$).

Examples:

- $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \{-1, 1\}$ (classification, sometimes $Y \in \{0, 1\}$),
- or $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \mathbb{R}$ (regression).

Target: construct predictor $f_\theta : \mathcal{X} \to \mathcal{Y}$ parametrized by θ.



Regression

Reminder: supervised learning

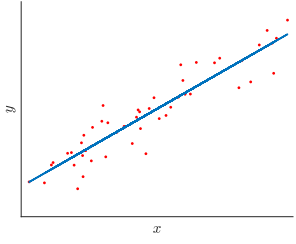
Problem setup:

- input measurement $\mathbf{X} \in \mathcal{X}$,
- output measurement $Y \in \mathcal{Y}$,
- $(\mathbf{X}, Y) \sim \mathcal{D}$ with $\mathcal{D}$ unknown,
- training data : $\mathcal{D}_n = \{(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_n, Y_n)\}$ (i.i.d. $\sim \mathbb{P}$).

Examples:

- $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \{-1, 1\}$ (classification, sometimes $Y \in \{0, 1\}$),
- or $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \mathbb{R}$ (regression).

Target: construct predictor $f_\theta : \mathcal{X} \to \mathcal{Y}$ parametrized by θ.



Regression

Reminder: supervised learning

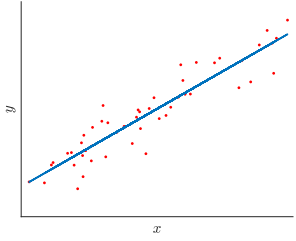
Problem setup:

- input measurement $\mathbf{X} \in \mathcal{X}$,
- output measurement $Y \in \mathcal{Y}$,
- $(\mathbf{X}, Y) \sim \mathcal{D}$ with $\mathcal{D}$ unknown,
- training data : $\mathcal{D}_n = \{(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_n, Y_n)\}$ (i.i.d. $\sim \mathbb{P}$).

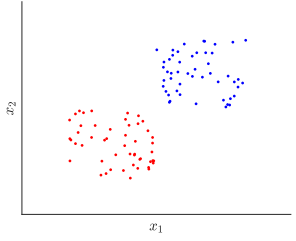
Examples:

- $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \{-1, 1\}$ (classification, sometimes $Y \in \{0, 1\}$),
- or $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \mathbb{R}$ (regression).

Target: construct predictor $f_\theta : \mathcal{X} \to \mathcal{Y}$ parametrized by θ.



Regression



Classification

Reminder: supervised learning

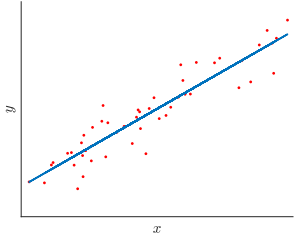
Problem setup:

- input measurement $\mathbf{X} \in \mathcal{X}$,
- output measurement $Y \in \mathcal{Y}$,
- $(\mathbf{X}, Y) \sim \mathcal{D}$ with $\mathcal{D}$ unknown,
- training data : $\mathcal{D}_n = \{(\mathbf{X}_1, Y_1), \ldots, (\mathbf{X}_n, Y_n)\}$ (i.i.d. $\sim \mathbb{P}$).

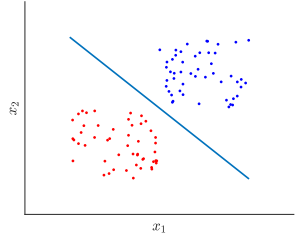
Examples:

- $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \{-1, 1\}$ (classification, sometimes $Y \in \{0, 1\}$),
- or $\mathbf{X} \in \mathbb{R}^d$ and $Y \in \mathbb{R}$ (regression).

Target: construct predictor $f_\theta : \mathcal{X} \to \mathcal{Y}$ parametrized by θ.



Regression



Classification

Reminder: empirical risk minimization

- The prediction is given by $f_\theta(\mathbf{x})$.

Reminder: empirical risk minimization

- The prediction is given by $f_\theta(\mathbf{x})$.
- Empirical risk minimization:

$$\operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i)) \equiv \operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \ell_i(\theta)$$

Reminder: empirical risk minimization

- The prediction is given by $f_\theta(\mathbf{x})$.
- Empirical risk minimization:

$$\operatorname*{argmin}_\theta \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i)) \equiv \operatorname*{argmin}_\theta \frac{1}{n}\sum_{i=1}^{n} \ell_i(\theta)$$

  - for now: assume $\ell_i(\cdot)$ is differentiable with nonzero gradient almost everywhere,
  - (so 0–1 loss function is prohibited).

Reminder: empirical risk minimization

- The prediction is given by $f_\theta(\mathbf{x})$.
- Empirical risk minimization:

$$\operatorname*{argmin}_\theta \frac{1}{n}\sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i)) \equiv \operatorname*{argmin}_\theta \frac{1}{n}\sum_{i=1}^{n} \ell_i(\theta)$$

  - for now: assume $\ell_i(\cdot)$ is differentiable with nonzero gradient almost everywhere,
  - (so 0–1 loss function is prohibited).
- Apply stochastic gradient descent with step-size $\eta_t > 0$

Stochastic gradient descent

While $|\theta_t - \theta_{t-1}| \geqslant \varepsilon$ do

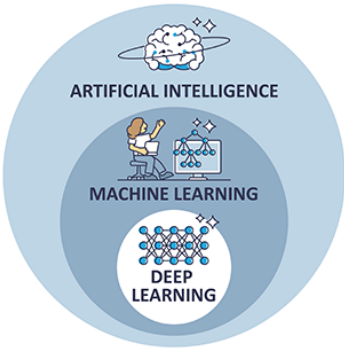
- Sample $I_t \subset \{1, \ldots, n\}$
-

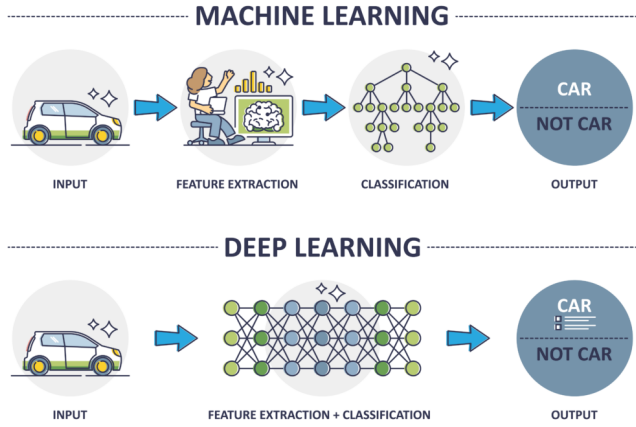
$$\theta_{t+1} = \theta_t - \eta_t \left(\frac{1}{|I_t|} \sum_{i \in I_t} \nabla_\theta \ell_i(\theta_t) \right)$$

Reminder: deep learning vs. traditional learning

Deel learning is subfield of machine learning:

- unstructured data,
- automated feature extraction.





Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Linear regression

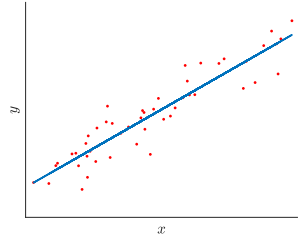
Data: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$. For simplicity, denote $\tilde{\mathbf{x}}_i = (\mathbf{x}_i, 1)$.

Linear regression

Data: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$. For simplicity, denote $\tilde{\mathbf{x}}_i = (\mathbf{x}_i, 1)$. $\rightarrow$ Find model: $y \approx f_\theta(\mathbf{x}) = \langle \theta, \tilde{\mathbf{x}} \rangle$:

Linear regression

Data: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$. For simplicity, denote $\tilde{\mathbf{x}}_i = (\mathbf{x}_i, 1)$. $\rightarrow$ Find model: $y \approx f_\theta(\mathbf{x}) = \langle \theta, \tilde{\mathbf{x}} \rangle$:



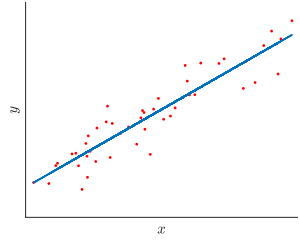
Example: linear least-squares

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i))$$

with $\ell(y_i, f_\theta(\mathbf{x}_i)) = |\langle \theta, \tilde{\mathbf{x}}_i \rangle - y_i|^2$.

Linear regression

Data: $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$. For simplicity, denote $\tilde{\mathbf{x}}_i = (\mathbf{x}_i, 1)$. $\rightarrow$ Find model: $y \approx f_\theta(\mathbf{x}) = \langle \theta, \tilde{\mathbf{x}} \rangle$:



Example: linear least-squares

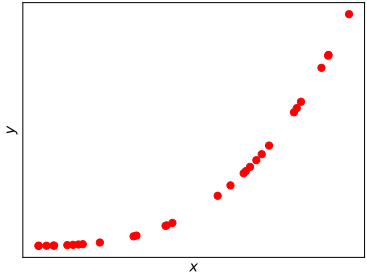
$$\underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i))$$

with $\ell(y_i, f_\theta(\mathbf{x}_i)) = |\langle \theta, \tilde{\mathbf{x}}_i \rangle - y_i|^2$. Shorthand (matrix) notation:

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{n} \|\mathbf{X}\theta - Y\|_2^2 \quad \text{with} \quad \mathbf{X} = \begin{pmatrix} x_1 & 1 \\ x_2 & 1 \\ \vdots & \vdots \\ x_n & 1 \end{pmatrix}, Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

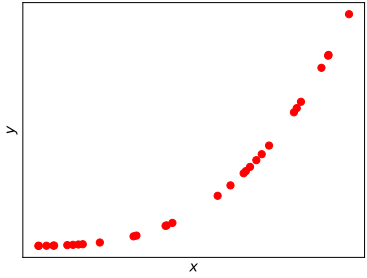
Regression: example

What if actual data is nonlinear? For instance, generate data according to: $y = w_1 x + w_3 x^3 + b$.

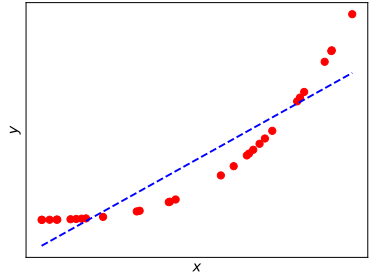


Regression: example

What if actual data is nonlinear? For instance, generate data according to: $y = w_1 x + w_3 x^3 + b$.

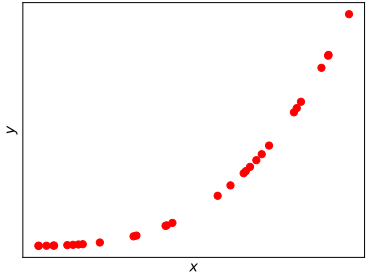


linear
regression
$\longrightarrow$

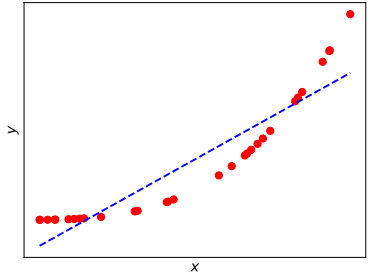


Regression: example

What if actual data is nonlinear? For instance, generate data according to: $y = w_1 x + w_3 x^3 + b$.



linear
regression
$\longrightarrow$



What to do?

- add features?
- how to solve the resulting regression problem?

Example: polynomial regression

- Model: $f_\theta(x) = w_1 x + w_2 x^2 + \ldots + w_d x^d + b$.
- Parameters: $\theta = (w_1, w_2, \ldots, w_d, b)$.

Example: polynomial regression

- Model: $f_\theta(x) = w_1 x + w_2 x^2 + \ldots + w_d x^d + b$.
- Parameters: $\theta = (w_1, w_2, \ldots, w_d, b)$.

- Fact: polynomial regression is linear regression in higher-dimensional space:

$$\operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i))$$

with $\ell(y_i, f_\theta(\mathbf{x}_i)) = |b + w_1 x_i + w_2 x_i^2 + \ldots + w_d x_i^d - y_i|^2$.

Example: polynomial regression

- Model: $f_\theta(x) = w_1 x + w_2 x^2 + \ldots + w_d x^d + b$.
- Parameters: $\theta = (w_1, w_2, \ldots, w_d, b)$.

- Fact: polynomial regression is linear regression in higher-dimensional space:

$$\operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i))$$

with $\ell(y_i, f_\theta(\mathbf{x}_i)) = |b + w_1 x_i + w_2 x_i^2 + \ldots + w_d x_i^d - y_i|^2$.

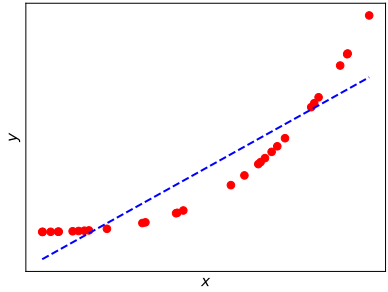
Shorthand (matrix) notation:

$$\operatorname*{argmin}_\theta \frac{1}{n} \|\mathbf{X}\theta - Y\|_2^2 \quad \text{with} \quad \mathbf{X} = \begin{pmatrix} x_1 & x_1^2 & \ldots & x_1^d & 1 \\ x_2 & x_2^2 & \ldots & x_2^d & 1 \\ \vdots & \vdots & & \vdots & \vdots \\ x_n & x_n^2 & \ldots & x_n^d & 1 \end{pmatrix}, Y = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}.$$

Regression: example

What if actual data is nonlinear? For instance, generate data according to: $y = w_1 x + w_3 x^3 + b$.

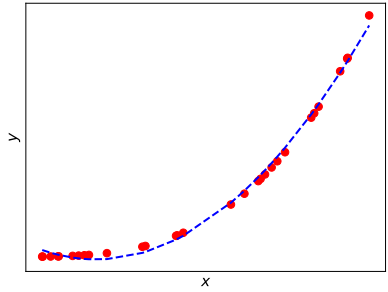
Linear regression:



Regression: example

What if actual data is nonlinear? For instance, generate data according to: $y = w_1 x + w_3 x^3 + b$.

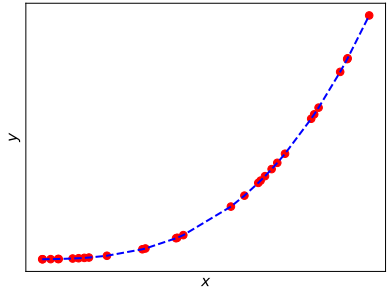
Quadratic regression:



Regression: example

What if actual data is nonlinear? For instance, generate data according to: $y = w_1 x + w_3 x^3 + b$.

Cubic regression:

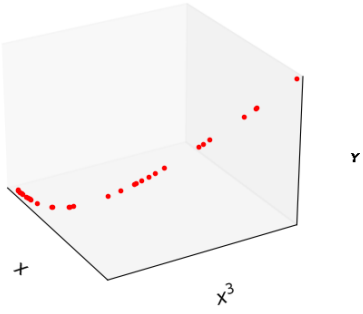


Regression with more features: interpretation

How to enrich those models? add (or change) features!

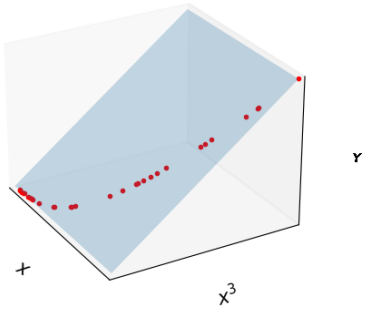
Regression with more features: interpretation

How to enrich those models? add (or change) features!



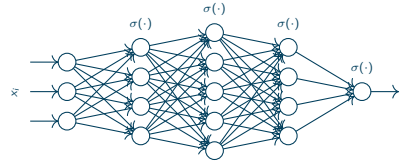
Regression with more features: interpretation

How to enrich those models? add (or change) features!

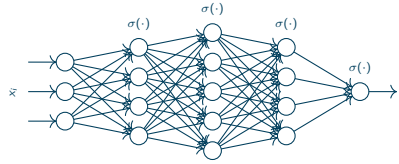


Interpretation: perform linear regression in larger-dimensional space.

Relationship with neural networks?



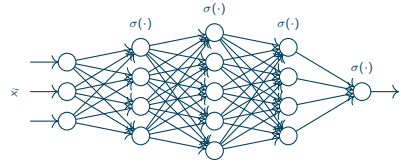
Relationship with neural networks?



Interpretation: for neural network with N layers:

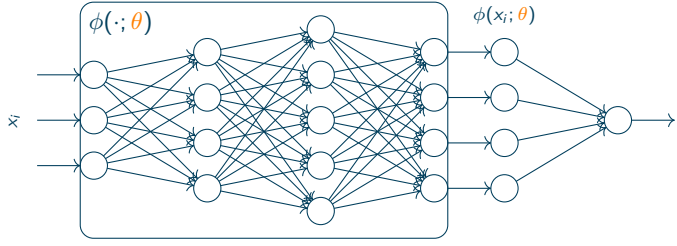
- first $N-1$ layers perform "feature extraction",
- last layer perform linear regression/classification.

Relationship with neural networks?



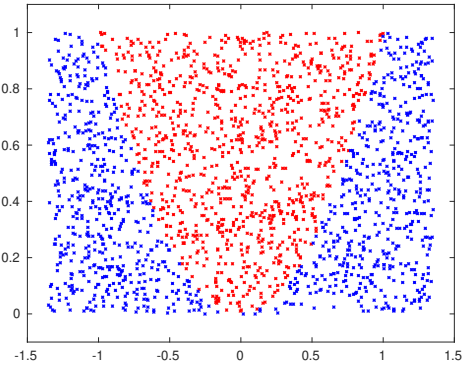
Interpretation: for neural network with N layers:

- first $N-1$ layers perform "feature extraction",
- last layer perform linear regression/classification.



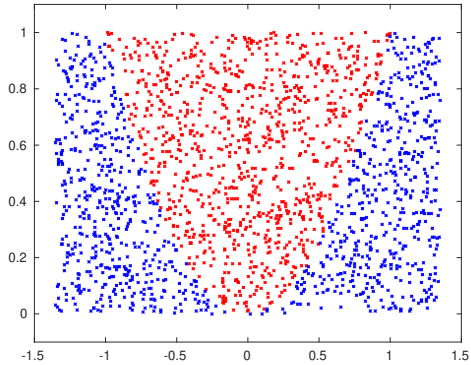
Classification

Non-linear classification using the same idea?



Classification

Non-linear classification using the same idea?



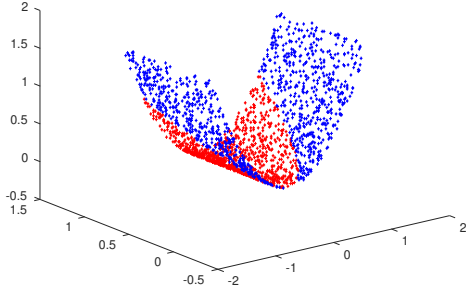
it looks like a parabolla $\rightarrow$ add feature x^2.

Classification

Non-linear classification using the same idea? it looks like a parabolla $\rightarrow$ add feature x^2.

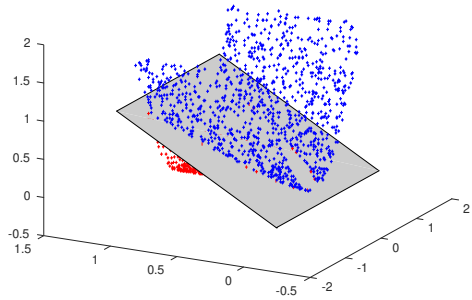
Classification

Non-linear classification using the same idea? it looks like a parabolla $\rightarrow$ add feature x^2.



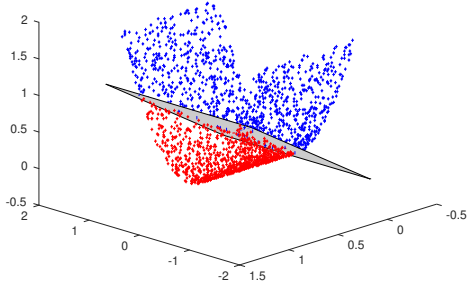
Classification

Non-linear classification using the same idea? it looks like a parabolla $\rightarrow$ add feature x^2.



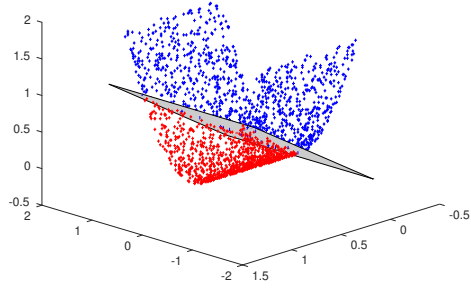
Classification

Non-linear classification using the same idea? it looks like a parabolla $\rightarrow$ add feature x^2.



Classification

Non-linear classification using the same idea? it looks like a parabolla $\rightarrow$ add feature x^2.



$\rightarrow$ lift & perform linear separation!

Reminder: training problem

- The prediction of the network is given by $f_\theta(\mathbf{x})$ $\rightarrow \theta$ contains all weights $W^{(k)}$ and biases $b^{(k)}$.

Reminder: training problem

- The prediction of the network is given by $f_\theta(\mathbf{x})$ $\quad \rightarrow \theta$ contains all weights $W^{(k)}$ and biases $b^{(k)}$.
- Empirical risk minimization:

$$\underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i)) \equiv \underset{\theta}{\operatorname{argmin}} \frac{1}{n} \sum_{i=1}^{n} \ell_i(\theta)$$

Reminder: training problem

- The prediction of the network is given by $f_\theta(\mathbf{x})$ $\quad \rightarrow \theta$ contains all weights $W^{(k)}$ and biases $b^{(k)}$.
- Empirical risk minimization:

$$\operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i)) \equiv \operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \ell_i(\theta)$$

  - for now: assume $\ell_i(\cdot)$ is differentiable with nonzero gradient almost everywhere,
  - (so 0–1 loss function is prohibited).

Reminder: training problem

- The prediction of the network is given by $f_\theta(\mathbf{x})$ $\quad \rightarrow \theta$ contains all weights $W^{(k)}$ and biases $b^{(k)}$.
- Empirical risk minimization:

$$\operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i)) \equiv \operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \ell_i(\theta)$$

  - for now: assume $\ell_i(\cdot)$ is differentiable with nonzero gradient almost everywhere,
  - (so 0–1 loss function is prohibited).
- Apply stochastic gradient descent with step-size $\eta_t > 0$.

Stochastic gradient descent

While $|\theta_t - \theta_{t-1}| \geqslant \varepsilon$ do

- Sample $I_t \subset \{1, \ldots, n\}$
-

$$\theta_{t+1} = \theta_t - \eta_t \left(\frac{1}{|I_t|} \sum_{i \in I_t} \nabla_\theta \ell_i(\theta_t) \right)$$

Reminder: training problem

- The prediction of the network is given by $f_\theta(\mathbf{x})$ $\quad \rightarrow \theta$ contains all weights $W^{(k)}$ and biases $b^{(k)}$.
- Empirical risk minimization:

$$\operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \ell(y_i, f_\theta(\mathbf{x}_i)) \equiv \operatorname*{argmin}_\theta \frac{1}{n} \sum_{i=1}^{n} \ell_i(\theta)$$

  - for now: assume $\ell_i(\cdot)$ is differentiable with nonzero gradient almost everywhere,
  - (so 0–1 loss function is prohibited).
- Apply stochastic gradient descent with step-size $\eta_t > 0$.

Stochastic gradient descent

While $|\theta_t - \theta_{t-1}| \geqslant \varepsilon$ do

- Sample $I_t \subset \{1, \ldots, n\}$
-

$$\theta_{t+1} = \theta_t - \eta_t \left(\frac{1}{|I_t|} \sum_{i \in I_t} \nabla_\theta \ell_i(\theta_t) \right)$$

Need to compute $\nabla_\theta \ell_i$ efficiently $\rightarrow$ backpropagation!

Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

How to compute $\nabla_\theta \ell_i$ efficiently?

A clever gradient descent implementation

- Popularized by Rumelhart, McClelland, Hinton,[a]
- can be traced back to Werbos,[b]
- a (smart) use of chain rule for derivatives.

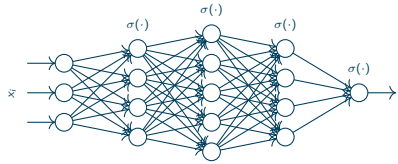
[a] Rumelhart et al. 1986; see also *Nobel prizes in Physics, 2024*.
[b] Werbos 1974

$\rightarrow$ key ingredient to make neural networks (NN) work!

$\rightarrow$ at the core of deep learning!

Backpropagation

Apply same ideas to gradients of neural networks.



Description:

$$f_\theta(\mathbf{x}) := \sigma(W^{(L)}\sigma(W^{(L-1)}\sigma(\cdots(W^{(2)}\sigma(W^{(1)}\mathbf{x}+b^{(1)})+b^{(2)})\cdots)+b^{(L-1)})+b^{(L)})$$

where θ contains all $W^{(k)}$ and $b^{(k)}$ $(k=1,\ldots,L)$. Convenient notation:

$$\mathbf{a}^{(k)} = \sigma\big(\underbrace{W^{(k)}\mathbf{a}^{(k-1)}+b^{(k)}}_{:=\mathbf{z}^{(k)}}\big) \quad \text{or } \mathbf{z}^{(k)} = W^{(k)}\underbrace{\sigma\left(\mathbf{z}^{(k-1)}\right)}_{:=\mathbf{a}^{(k-1)}}+b^{(k)}.$$

Backpropagation for a traditional NN

Neural network with L layers, with vector output and loss

$$C(\theta) \triangleq \ell(y, \sigma(W^{(L)}\sigma(W^{(L-1)}\sigma(\cdots(W^{(2)}\sigma(W^{(1)}\mathbf{x} + b^{(1)}) + b^{(2)})\cdots) + b^{(L-1)}) + b^{(L)})).$$

For convenience, introduce a few intermediary notation:

Backpropagation for a traditional NN

Neural network with L layers, with vector output and loss

$$C(\theta) \triangleq \ell(y, \sigma(W^{(L)}\sigma(W^{(L-1)}\sigma(\cdots(W^{(2)}\sigma(W^{(1)}\mathbf{x} + b^{(1)}) + b^{(2)})\cdots) + b^{(L-1)}) + b^{(L)})).$$

For convenience, introduce a few intermediary notation:

- truncated parameters $\theta^{(k)}$ that contain $W^{(k+1)}, \ldots, W^{(L)}$ and $b^{(k+1)}, \ldots, b^{(L)}$ (so $\theta^{(L)}$ is empty and $\theta^{(0)}$ contains all parameters).

Backpropagation for a traditional NN

Neural network with L layers, with vector output and loss

$$C(\theta) \triangleq \ell(y, \sigma(W^{(L)}\sigma(W^{(L-1)}\sigma(\cdots(W^{(2)}\sigma(W^{(1)}\mathbf{x} + b^{(1)}) + b^{(2)})\cdots) + b^{(L-1)}) + b^{(L)})).$$

For convenience, introduce a few intermediary notation:

- truncated parameters $\theta^{(k)}$ that contain $W^{(k+1)}, \ldots, W^{(L)}$ and $b^{(k+1)}, \ldots, b^{(L)}$ (so $\theta^{(L)}$ is empty and $\theta^{(0)}$ contains all parameters).
- truncated loss

$$C^{(k)}(\theta^{(k)}, z^{(k)}) \triangleq \ell(y, \sigma(W^{(L)}\sigma(\cdots(W^{(k+1)}\sigma(z^{(k)}) + b^{(k+1)})\cdots) + b^{(L)}),$$

with $C^{(L)}(\theta^{(L)}, z^{(L)}) \triangleq \ell(y, \sigma(z^{(L)}))$. Important note: in practice,

$$C^{(L)}(\theta^{(L)}, z^{(L)}) = C^{(L-1)}(\theta^{(L-1)}, z^{(L-1)}) = \ldots = C^{(0)}(\theta^{(0)}, z^{(0)}).$$

Backpropagation for a traditional NN

Neural network with L layers, with vector output and loss

$$C(\theta) \triangleq \ell(y, \sigma(W^{(L)}\sigma(W^{(L-1)}\sigma(\cdots(W^{(2)}\sigma(W^{(1)}\mathbf{x} + b^{(1)}) + b^{(2)})\cdots) + b^{(L-1)}) + b^{(L)})).$$

For convenience, introduce a few intermediary notation:

- truncated parameters $\theta^{(k)}$ that contain $W^{(k+1)}, \ldots, W^{(L)}$ and $b^{(k+1)}, \ldots, b^{(L)}$ (so $\theta^{(L)}$ is empty and $\theta^{(0)}$ contains all parameters).
- truncated loss

$$C^{(k)}(\theta^{(k)}, z^{(k)}) \triangleq \ell(y, \sigma(W^{(L)}\sigma(\cdots(W^{(k+1)}\sigma(z^{(k)}) + b^{(k+1)})\cdots) + b^{(L)}),$$

  with $C^{(L)}(\theta^{(L)}, z^{(L)}) \triangleq \ell(y, \sigma(z^{(L)}))$. Important note: in practice,

$$C^{(L)}(\theta^{(L)}, z^{(L)}) = C^{(L-1)}(\theta^{(L-1)}, z^{(L-1)}) = \ldots = C^{(0)}(\theta^{(0)}, z^{(0)}).$$

- We will intensively use $\delta_j^{(k)} \triangleq \frac{\partial C^{(k)}}{\partial \mathbf{z}_j^{(k)}}(\theta^{(k)}, z^{(k)})$ for our computations.

Backpropagation for a traditional NN

Neural network with L layers, with vector output and loss

$$C(\theta) \triangleq \ell(y, \sigma(W^{(L)}\sigma(W^{(L-1)}\sigma(\cdots(W^{(2)}\sigma(W^{(1)}\mathbf{x} + b^{(1)}) + b^{(2)})\cdots) + b^{(L-1)}) + b^{(L)})).$$

For convenience, introduce a few intermediary notation:

- truncated parameters $\theta^{(k)}$ that contain $W^{(k+1)}, \ldots, W^{(L)}$ and $b^{(k+1)}, \ldots, b^{(L)}$ (so $\theta^{(L)}$ is empty and $\theta^{(0)}$ contains all parameters).
- truncated loss

$$C^{(k)}(\theta^{(k)}, z^{(k)}) \triangleq \ell(y, \sigma(W^{(L)}\sigma(\cdots(W^{(k+1)}\sigma(z^{(k)}) + b^{(k+1)})\cdots) + b^{(L)}),$$

  with $C^{(L)}(\theta^{(L)}, z^{(L)}) \triangleq \ell(y, \sigma(z^{(L)}))$. Important note: in practice,

$$C^{(L)}(\theta^{(L)}, z^{(L)}) = C^{(L-1)}(\theta^{(L-1)}, z^{(L-1)}) = \ldots = C^{(0)}(\theta^{(0)}, z^{(0)}).$$

- We will intensively use $\delta_j^{(k)} \triangleq \frac{\partial C^{(k)}}{\partial \mathbf{z}_j^{(k)}}(\theta^{(k)}, z^{(k)})$ for our computations.
- Additional Notes:
  - $z^{(L)}, \ldots, z^{(k)}$ (and $a^{(L)}, \ldots, a^{(k)}$) do not depend on $\theta^{(k)}$ ("causality").

Backpropagation for a traditional NN

Neural network with L layers, with vector output and loss

$$C(\theta) \triangleq \ell(y, \sigma(W^{(L)}\sigma(W^{(L-1)}\sigma(\cdots(W^{(2)}\sigma(W^{(1)}\mathbf{x} + b^{(1)}) + b^{(2)})\cdots) + b^{(L-1)}) + b^{(L)})).$$

For convenience, introduce a few intermediary notation:

- truncated parameters $\theta^{(k)}$ that contain $W^{(k+1)}, \ldots, W^{(L)}$ and $b^{(k+1)}, \ldots, b^{(L)}$ (so $\theta^{(L)}$ is empty and $\theta^{(0)}$ contains all parameters).
- truncated loss

$$C^{(k)}(\theta^{(k)}, z^{(k)}) \triangleq \ell(y, \sigma(W^{(L)}\sigma(\cdots(W^{(k+1)}\sigma(z^{(k)}) + b^{(k+1)})\cdots) + b^{(L)}),$$

  with $C^{(L)}(\theta^{(L)}, z^{(L)}) \triangleq \ell(y, \sigma(z^{(L)}))$. Important note: in practice,

$$C^{(L)}(\theta^{(L)}, z^{(L)}) = C^{(L-1)}(\theta^{(L-1)}, z^{(L-1)}) = \ldots = C^{(0)}(\theta^{(0)}, z^{(0)}).$$

- We will intensively use $\delta_j^{(k)} \triangleq \frac{\partial C^{(k)}}{\partial \mathbf{z}_j^{(k)}}(\theta^{(k)}, z^{(k)})$ for our computations.
- Additional Notes:
  - $z^{(L)}, \ldots, z^{(k)}$ (and $a^{(L)}, \ldots, a^{(k)}$) do not depend on $\theta^{(k)}$ ("causality").
  - consequence: $\frac{\partial C}{\partial \theta^{(k)}}(\theta) = \frac{\partial C^{(k)}}{\partial \theta^{(k)}}(\theta^{(k)}, z^{(k)})$.

Unrolling derivatives via chain rule

Consider a simple unidimensional network:

- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).

Unrolling derivatives via chain rule

Consider a simple unidimensional network:

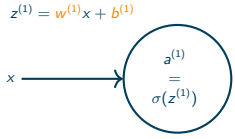
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).

x

Unrolling derivatives via chain rule

Consider a simple unidimensional network:

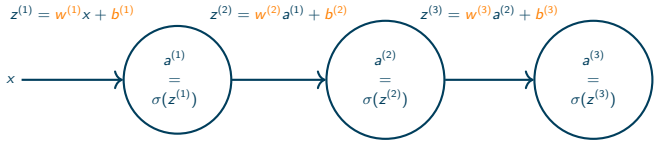
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

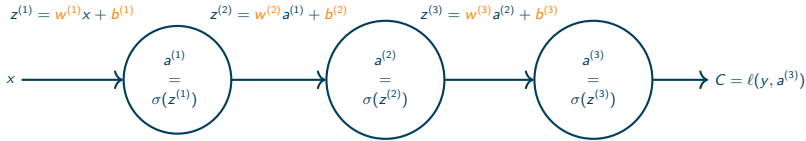
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

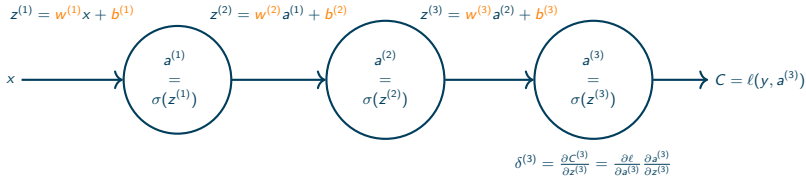
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

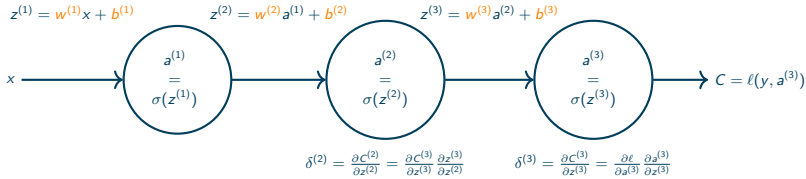
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \dots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \dots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

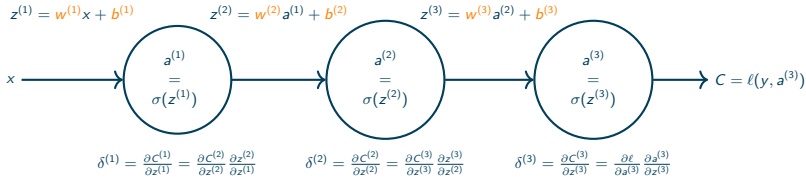
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \dots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \dots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

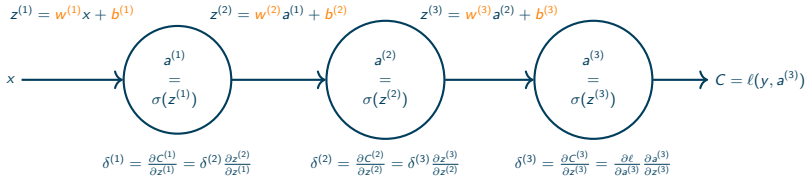
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \dots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \dots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

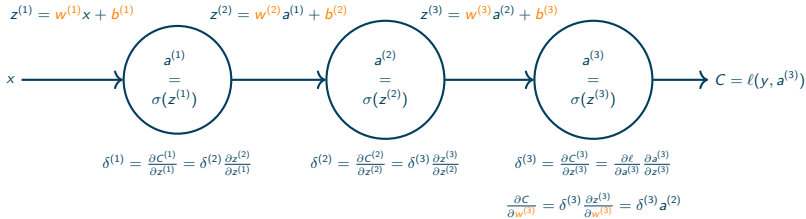
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \dots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \dots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

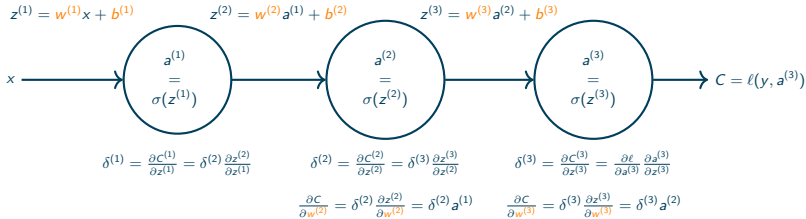
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

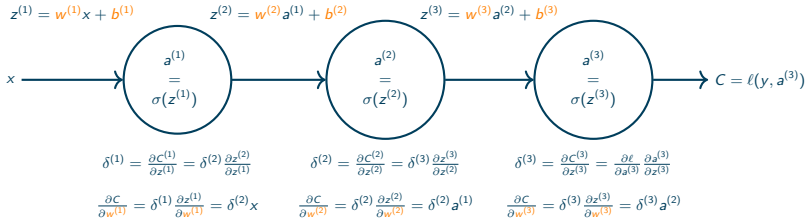
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \dots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \dots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

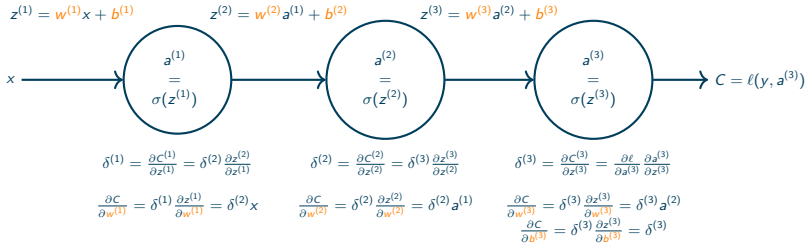
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \dots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \dots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

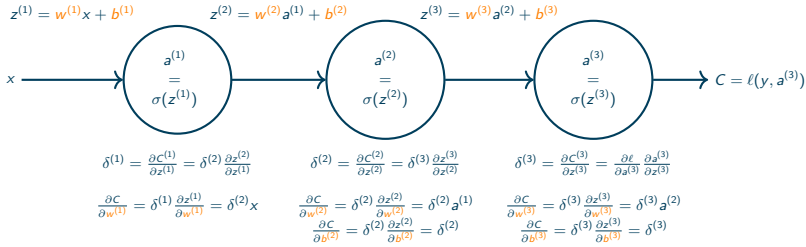
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

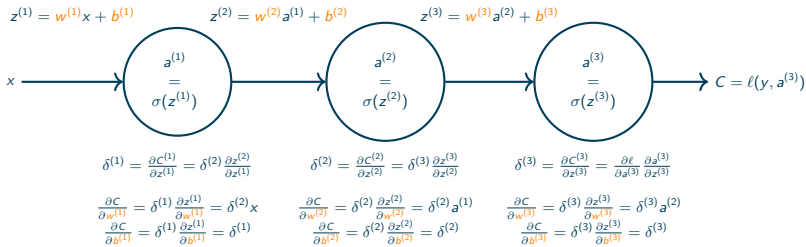
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

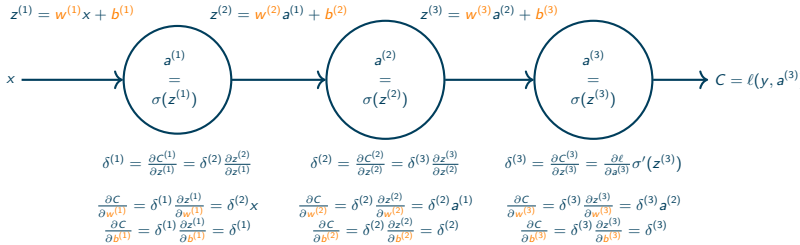
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \dots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \dots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

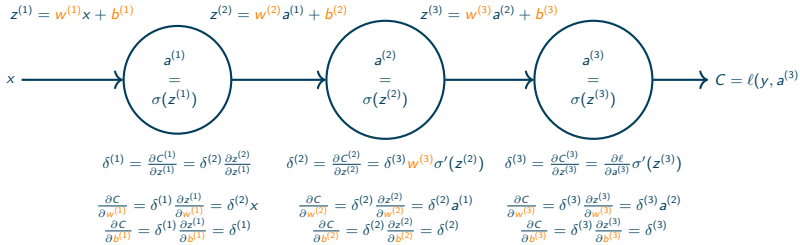
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \dots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \dots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

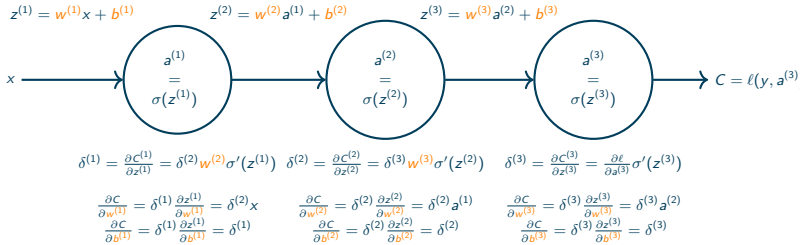
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \dots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \dots, L$ (all layers).



Unrolling derivatives via chain rule

Consider a simple unidimensional network:

- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Backpropagation equations for traditional neural network

Denote by $\odot$ the element-wise (a.k.a., Hadamard) product and $\ell(y, a)$ the loss

Initialize $\delta^{(L)} = \sigma'(z^{(L)}) \odot \frac{\partial \ell}{\partial a}(y, a^{(L)})$.

Fundamental equations of backpropagation (for traditional neural network):

- derivatives with respect to input values at layer k

$$\delta^{(k)} = ((W^{(k+1)})^T \delta^{(k+1)}) \odot \sigma'(z^{(k)}) \tag{1}$$

(those quantities are not of direct interest, but allows for efficient computations),

Backpropagation equations for traditional neural network

Denote by $\odot$ the element-wise (a.k.a., Hadamard) product and $\ell(y, a)$ the loss

Initialize $\delta^{(L)} = \sigma'(z^{(L)}) \odot \frac{\partial \ell}{\partial a}(y, a^{(L)})$.

Fundamental equations of backpropagation (for traditional neural network):

- derivatives with respect to input values at layer k

$$\delta^{(k)} = ((W^{(k+1)})^T \delta^{(k+1)}) \odot \sigma'(z^{(k)}) \tag{1}$$

  (those quantities are not of direct interest, but allows for efficient computations),

- derivatives with respect to parameters

$$\begin{aligned} \frac{\partial C}{\partial b_j^{(k)}}(\theta) &= \delta_j^{(k)} \\ \frac{\partial C}{\partial w_{i,j}^{(k)}}(\theta) &= a_j^{(k-1)} \delta_i^{(k)}, \end{aligned} \tag{2}$$

Reminder: backpropagation

Convenient notation:

$$\mathbf{z}^{(k)} = W^{(k)} \underbrace{\sigma\left(\mathbf{z}^{(k-1)}\right)}_{:=\mathbf{a}^{(k-1)}} + b^{(k)}.$$

(recall that $\mathbf{z}^{(i)}$ is input for layer i).

Backpropagation:

Reminder: backpropagation

Convenient notation:

$$\mathbf{z}^{(k)} = W^{(k)} \underbrace{\sigma\left(\mathbf{z}^{(k-1)}\right)}_{:=\mathbf{a}^{(k-1)}} + b^{(k)}.$$

(recall that $\mathbf{z}^{(i)}$ is input for layer i).

Backpropagation:

- apply chain rule to compute (efficiently) gradient of the loss,
- one full forward and backward pass necessary per datapoint,
- so: full gradient requires n forward and n backward passes,
- so: usually only "stochastic gradient" are computed.

Reminder: backpropagation

Convenient notation:

$$\mathbf{z}^{(k)} = W^{(k)} \underbrace{\sigma\left(\mathbf{z}^{(k-1)}\right)}_{:=\mathbf{a}^{(k-1)}} + b^{(k)}.$$

(recall that $\mathbf{z}^{(i)}$ is input for layer i).

Backpropagation:

- apply chain rule to compute (efficiently) gradient of the loss,
- one full forward and backward pass necessary per datapoint,
- so: full gradient requires n forward and n backward passes,
- so: usually only "stochastic gradient" are computed.

Generally: even when things are not differentiable, do as if they were. ["Conservative set valued fields, automatic differentiation, stochastic gradient methods and deep learning", Bolte and Pauwels 2021]

Training a traditional neural network

Let

$$\delta_j^{(k)} = \frac{\partial C^{(k)}}{\partial z_j^{(k)}},$$

where $z_j^{(k)}$ is the input of neuron j at layer k.

Neural network training

(a) Initialize randomly the weights and biases in the network.

(b) For all training samples $(x_i)_{i \in B}$ in the batch B,

Training a traditional neural network

Let

$$\delta_j^{(k)} = \frac{\partial C^{(k)}}{\partial z_j^{(k)}},$$

where $z_j^{(k)}$ is the input of neuron j at layer k.

Neural network training

(a) Initialize randomly the weights and biases in the network.

(b) For all training samples $(x_i)_{i \in B}$ in the batch B,

1. **Feedforward:** send all samples of the batch through the network and store the values of activation functions and their derivatives (at each neuron).

Training a traditional neural network

Let

$$\delta_j^{(k)} = \frac{\partial C^{(k)}}{\partial z_j^{(k)}},$$

where $z_j^{(k)}$ is the input of neuron j at layer k.

Neural network training

(a) Initialize randomly the weights and biases in the network.

(b) For all training samples $(x_i)_{i \in B}$ in the batch B,

1. **Feedforward:** send all samples of the batch through the network and store the values of activation functions and their derivatives (at each neuron).
2. **Output loss:** compute the neural network loss average on all samples of the batch.

Training a traditional neural network

Let

$$\delta_j^{(k)} = \frac{\partial C^{(k)}}{\partial z_j^{(k)}},$$

where $z_j^{(k)}$ is the input of neuron j at layer k.

Neural network training

(a) Initialize randomly the weights and biases in the network.

(b) For all training samples $(x_i)_{i \in B}$ in the batch B,

1. **Feedforward:** send all samples of the batch through the network and store the values of activation functions and their derivatives (at each neuron).
2. **Output loss:** compute the neural network loss average on all samples of the batch.
3. **Backpropagation (BP):** compute recursively the vectors $\delta^{(k)}$ starting from $k = L$ to $k = 1$ with (1). Compute gradients wrt parameters with (2).

Training a traditional neural network

Let

$$\delta_j^{(k)} = \frac{\partial C^{(k)}}{\partial z_j^{(k)}},$$

where $z_j^{(k)}$ is the input of neuron j at layer k.

Neural network training

(a) Initialize randomly the weights and biases in the network.

(b) For all training samples $(x_i)_{i \in B}$ in the batch B,

1. **Feedforward:** send all samples of the batch through the network and store the values of activation functions and their derivatives (at each neuron).
2. **Output loss:** compute the neural network loss average on all samples of the batch.
3. **Backpropagation (BP):** compute recursively the vectors $\delta^{(k)}$ starting from $k = L$ to $k = 1$ with (1). Compute gradients wrt parameters with (2).
4. **Optimization:** update the weights and biases using a gradient-based optimization procedure, using the gradient previously computed.

Training a traditional neural network

Let

$$\delta_j^{(k)} = \frac{\partial C^{(k)}}{\partial z_j^{(k)}},$$

where $z_j^{(k)}$ is the input of neuron j at layer k.

Neural network training

(a) Initialize randomly the weights and biases in the network.

(b) For all training samples $(x_i)_{i \in B}$ in the batch B,

1. **Feedforward:** send all samples of the batch through the network and store the values of activation functions and their derivatives (at each neuron).
2. **Output loss:** compute the neural network loss average on all samples of the batch.
3. **Backpropagation (BP):** compute recursively the vectors $\delta^{(k)}$ starting from $k = L$ to $k = 1$ with (1). Compute gradients wrt parameters with (2).
4. **Optimization:** update the weights and biases using a gradient-based optimization procedure, using the gradient previously computed.

(c) Repeat step (b) until some convergence criterion is reached.

Targets for today: understand key ingredients & pipeline!

What does this do?

```
import torch
import math

# Create Tensors to hold input and outputs.
x = torch.linspace(-math.pi, math.pi, 2000)
y = torch.sin(x)

# Prepare the input tensor (x, x^2, x^3).
p = torch.tensor([1, 2, 3])
xx = x.unsqueeze(-1).pow(p)

# Use the nn package to define our architecture
model = torch.nn.Sequential(
    torch.nn.Linear(3, 3),
    torch.nn.LeakyReLU(),
    torch.nn.Linear(3, 1),
    torch.nn.Flatten(0, 1)
)

# use the nn package to define our loss
loss_fn = torch.nn.MSELoss(reduction='mean')
```

Targets for today: understand the key ingredients & pipeline!

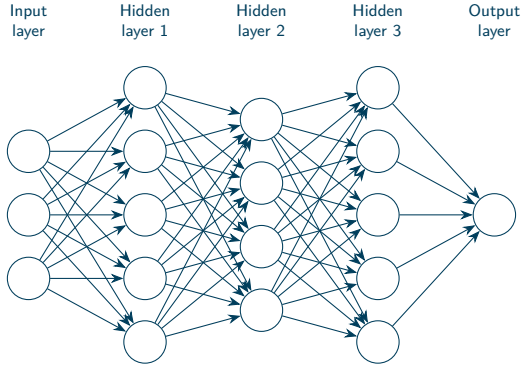
What does this do?

```
# use the optim package to define the algorithm
learning_rate = 1e-3
optimizer = torch.optim.RMSprop(model.parameters(), lr=learning_rate)

for t in range(2000):

    # Forward pass: compute predicted y by passing x to the model.
    y_pred = model(xx)

    # Computes the loss
    loss = loss_fn(y_pred, y)

    # Zero out accumulated gradients
    optimizer.zero_grad()

    # Backward pass --> computes gradients
    loss.backward()

    # One step of the optimizer
    optimizer.step()
```

Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

What to set in a neural network?



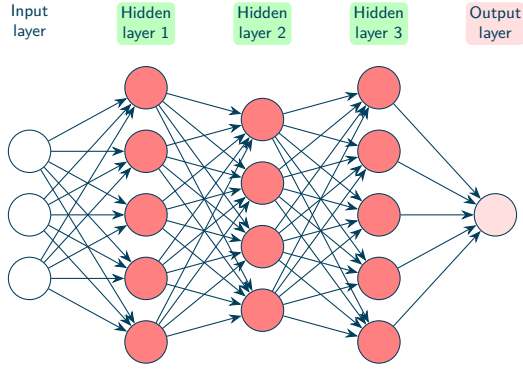
Network structure:

- Number of layers / neurons per layer.
- Activation functions.
- Output unit.
- Specific layers (dropout, batch normalization, etc.).

Optimization:

- Optimization algorithm.
- Weights/biases initialization.
- Loss function.

What to set in a neural network?



Network structure:

- Number of layers / neurons per layer
- Activation functions
- Output unit
- Specific layers (dropout, batch normalization, etc.)

Optimization:

- Optimization algorithm
- Weights/biases initialization
- Loss function

Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Number of hidden layers/neurons

Number of hidden layers/neurons

- **No particular rules** for choosing number of layers or number of neurons per layer.

Number of hidden layers/neurons

- **No particular rules** for choosing number of layers or number of neurons per layer.
- **Read research papers** related to the target task, test proposed architectures.

Number of hidden layers/neurons

- **No particular rules** for choosing number of layers or number of neurons per layer.
- **Read research papers** related to the target task, test proposed architectures.
- **Play with the architecture** and see how it influences performance.

Number of hidden layers/neurons

- **No particular rules** for choosing number of layers or number of neurons per layer.
- **Read research papers** related to the target task, test proposed architectures.
- **Play with the architecture** and see how it influences performance.

⚠ : many rules of thumb not supported by evidence (neither practical, nor theoretical).

Number of hidden layers/neurons

- **No particular rules** for choosing number of layers or number of neurons per layer.
- **Read research papers** related to the target task, test proposed architectures.
- **Play with the architecture** and see how it influences performance.

⚠ : many rules of thumb not supported by evidence (neither practical, nor theoretical).

- Use **data-driven strategies:**
  - **Network pruning** following the procedure training/pruning/training/pruning/...
    ["What is the state of neural network pruning?", Blalock et al. 2020]
  - More complex **evolutionary algorithms**
    ["AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data", Egele et al. 2020]

Number of hidden layers/neurons

- **No particular rules** for choosing number of layers or number of neurons per layer.
- **Read research papers** related to the target task, test proposed architectures.
- **Play with the architecture** and see how it influences performance.

⚠ : many rules of thumb not supported by evidence (neither practical, nor theoretical).

- Use **data-driven strategies:**
  - **Network pruning** following the procedure training/pruning/training/pruning/...
    ["What is the state of neural network pruning?", Blalock et al. 2020]
  - More complex **evolutionary algorithms**
    ["AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data", Egele et al. 2020]
- Experience is the key!
- Don't hesitate to test `http://playground.tensorflow.org`.

Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Step $(0-1)$ activation function

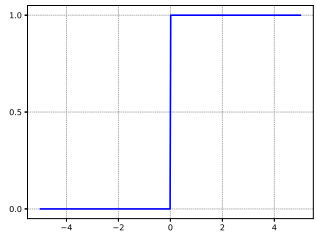


Figure: Step $(0-1)$ activation function.

Comments:

- Any issue?

Step $(0-1)$ activation function

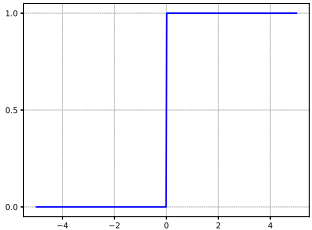


Figure: Step $(0-1)$ activation function.

Comments:

- Any issue?
  - Zero gradient almost everywhere.

Step $(0-1)$ activation function

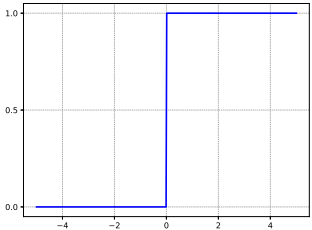


Figure: Step $(0-1)$ activation function.

Comments:

- Any issue?
  - Zero gradient almost everywhere.
  - How to optimize?

Step $(0 - 1)$ activation function

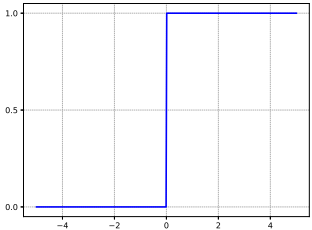


Figure: Step $(0 - 1)$ activation function.

Comments:

- Any issue?
  - Zero gradient almost everywhere.
  - How to optimize?

  $\rightarrow$ Should be avoided at all cost.

Sigmoid activation function

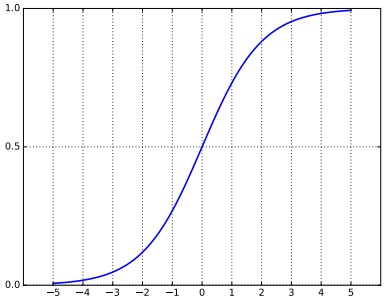


Figure: Sigmoid activation function σ

$$\sigma : x \mapsto \frac{\exp(x)}{1 + \exp(x)}$$

$$\sigma' : x \mapsto \frac{\exp(x)}{(1 + \exp(x))^2}$$

Comments:

- Saturation (horizontal asymptotes):
  - gradient close to zero in these areas ($\pm\infty$),
  - rescaling inputs of each layer can help to avoid these areas.
- Sigmoid is not a zero-centered function.
- Computing $\exp(x)$ is a bit costly.

Hyperbolic tangent

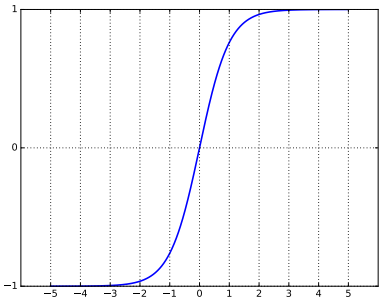


Figure: Hyperbolic tangent (tanh)

$$\tanh : x \mapsto \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$

$$\tanh' : x \mapsto 4\frac{\exp(2x)}{(1 + \exp(2x))^2}$$

Comments:

- Function tanh is zero-centered
  - no need to rescale data.
- Saturation (horizontal asymptotes):
  - gradient close to zero in these areas ($\pm\infty$),
  - rescaling inputs of each layer can help to avoid these areas.
- Computing $\exp(x)$ is a bit costly.

Note that $\tanh(x) = 2\sigma(2x) - 1$.

Rectified linear unit (ReLU)

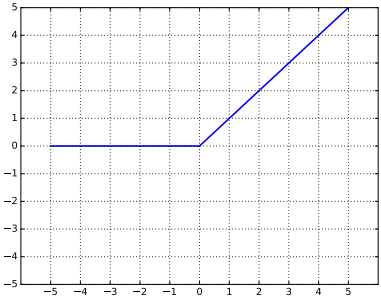


Figure: Rectified linear unit (ReLU)

$$ReLU : x \mapsto \max(0, x)$$

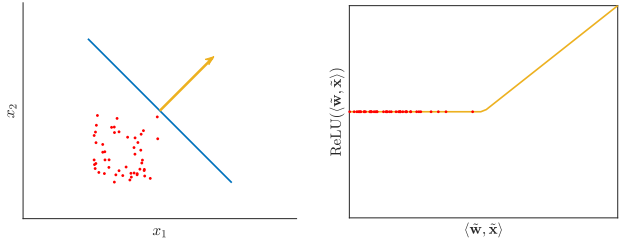
Comments:

- No saturation as $x \to +\infty$,
- but saturation (and zero) when $x \leqslant 0$.
- Computationally efficient.
- Empirically: faster convergence than with sigmoid/tanh.
- Note: nonsmooth!

Dead neurons

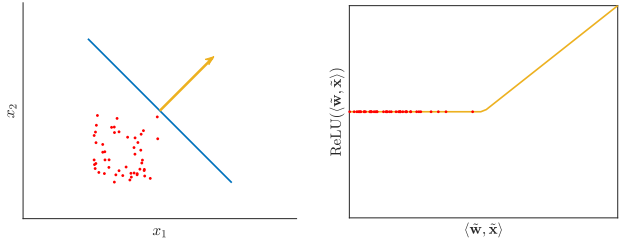
Dead neurons

Dead ReLU:

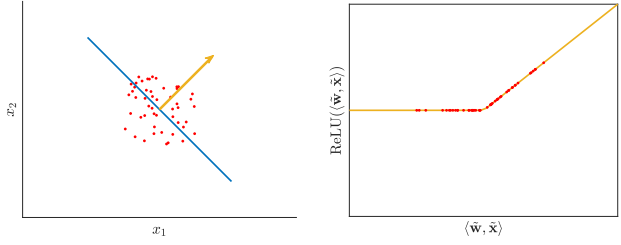


Dead neurons

Dead ReLU:



Active ReLU:



More on ReLU

ReLU in neural networks can be traced back to ["Cognitron: A self-organizing multilayered neural network"; "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition", Fukushima 1975; Fukushima and Miyake 1982].

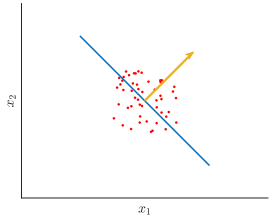


Figure: Good parameter init.–ReLU is active.

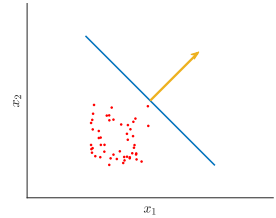


Figure: Bad parameter init.–ReLU outputs zero.

More on ReLU

ReLU in neural networks can be traced back to ["Cognitron: A self-organizing multilayered neural network"; "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition", Fukushima 1975; Fukushima and Miyake 1982].

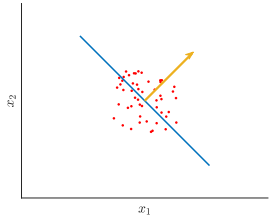


Figure: Good parameter init.–ReLU is active.

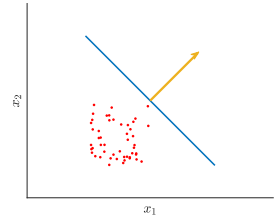


Figure: Bad parameter init.–ReLU outputs zero.

ReLU output can be zero but positive initial bias can help.

More on ReLU

ReLU in neural networks can be traced back to ["Cognitron: A self-organizing multilayered neural network"; "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition", Fukushima 1975; Fukushima and Miyake 1982].

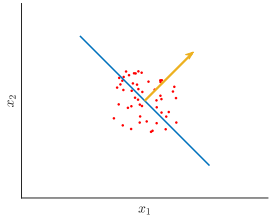


Figure: Good parameter init.–ReLU is active.

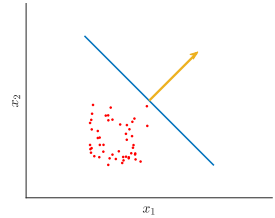


Figure: Bad parameter init.–ReLU outputs zero.

ReLU output can be zero but positive initial bias can help.

Motivation ["Deep sparse rectifier neural networks", Glorot, Bordes, et al. 2011]:

- Most of the time, neurons are inactive.
- when they activate, their activation is proportional to their input.

Parametric ReLU

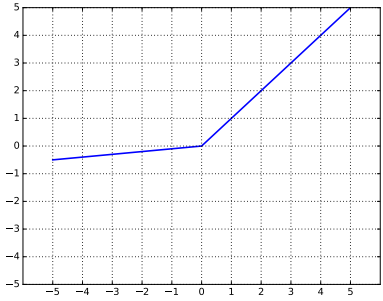


Figure: Parametric ReLU

Comments:

- Leaky ReLU picks $\alpha = 0.1$.

  ["Rectifier nonlinearities improve neural network acoustic models", Maas et al. 2013]

- Absolute value rectification:

  $$\alpha = -1.$$

  ["What is the best multi-stage architecture for object recognition?", Jarrett et al. 2009]

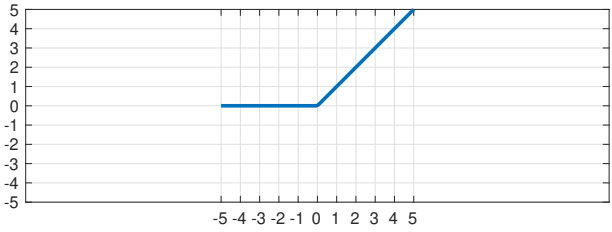
- Parametric ReLU: α is trained

  $\rightarrow$ activation function is learned.

  ["Empirical evaluation of rectified activations in convolutional network", Xu et al. 2015]

$$\text{Parametric ReLU} : x \mapsto \max(\alpha x, x)$$

Concatenated rectified linear unit (CReLU)



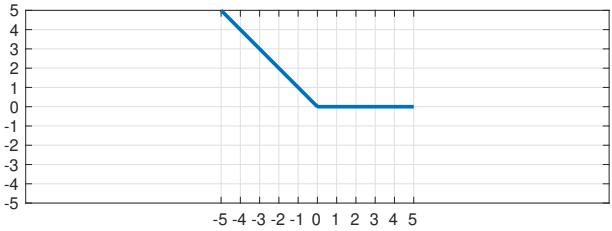


Figure: Concatenated rectified linear unit (CReLU)

Comments:

- it has two outputs!
- Another way to fight dead neurons.
- See ["Understanding and improving convolutional neural networks via concatenated rectified linear units", Shang et al. 2016].

Exponential linear unit (ELU)

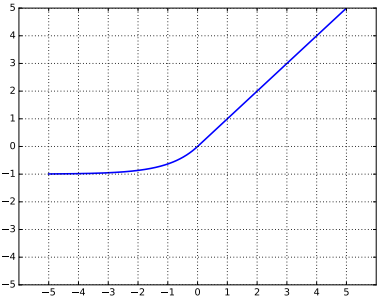


Figure: Exponential linear unit (ELU)

Comments:

- α often set to 1.0.
- Close to ReLU but differentiable everywhere.
- Robust to noise.

  ["Fast and accurate deep network learning by exponential linear units (elus)", Clevert et al. 2015]

$$ELU : x \mapsto \begin{cases} x & \text{if } x \geqslant 0 \\ \alpha(\exp(x) - 1) & \text{otherwise} \end{cases}$$

Maxout

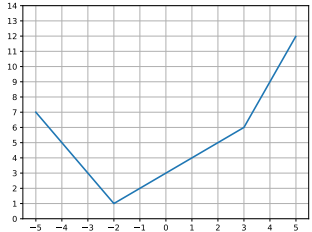


Figure: Maxout activation ($k = 3$ pieces).

Maxout

$$x \mapsto \max(W_1 x + b_1, \dots, W_k x + b_k)$$

Comments:

- Learn piecewise linear functions (k pieces).
- Linear regime with k pieces: no saturation.
- Increases number of parameters

  "full maxout" network has k times more parameters than conventional activations (e.g., compared to ReLU+fully connected layer).

  ["Maxout networks", Goodfellow, Warde-Farley, et al. 2013] ["Deep maxout neural networks for speech recognition", Cai et al. 2013]
- Resist to catastrophic forgetting.

  ["An empirical investigation of catastrophic forgetting in gradient-based neural networks", Goodfellow, Mirza, et al. 2013]

Swish

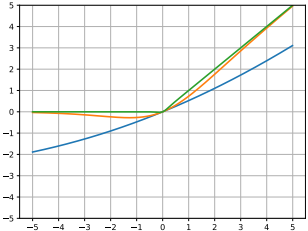


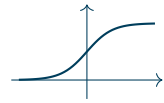
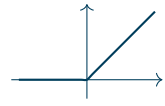
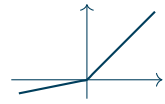
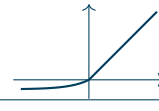
Figure: Swish for $\beta = 0.1, 1, 10$

$$\textit{Swish} : x \mapsto x \frac{\exp(\beta x)}{1 + \exp(\beta x)}$$

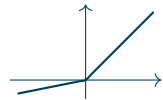
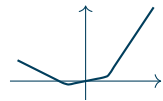
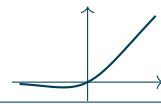
Comments:

- Swish interpolates between the linear function and ReLU.
- ["Searching for activation functions", Ramachandran et al. 2017]
- Non-monotonic function which seems to be an important feature.

Summary – examples of activation functions (1/2)

Name	$\sigma(u)$	Graph
Sigmoid	$\frac{1}{1+e^{-u}}$	
Tanh	$\frac{e^{u}-e^{-u}}{e^{-u}+e^{u}}$	
ReLU	$\max(u, 0)$	
LeakyReLU	$\max(\alpha u, u)$	
ELU	$\begin{cases} u & \text{if } u \geqslant 0 \\ \alpha(e^{u}-1) & \text{else} \end{cases}$	

Summary – examples of activation functions (2/2)

Name	$\sigma(u)$	Graph
Parametric ReLU	$\max(\alpha u, u)$	
Maxout	$\max(W_1 u + b_1, \ldots, W_k u + b_k)$	
Swish	$u \frac{e^{\beta u}}{1+e^{\beta u}}$	

Activation functions: conclusions

- Use ReLU (or Swish).

Activation functions: conclusions

- Use ReLU (or Swish).
- Test Leaky ReLU, maxout, ELU.

Activation functions: conclusions

- Use ReLU (or Swish).
- Test Leaky ReLU, maxout, ELU.
- Try out Tanh, but do not expect too much.

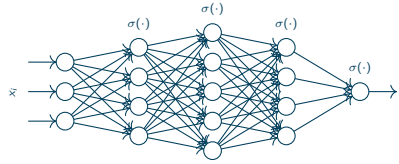
Activation functions: conclusions

- Use ReLU (or Swish).
- Test Leaky ReLU, maxout, ELU.
- Try out Tanh, but do not expect too much.
- Generally avoid sigmoid.

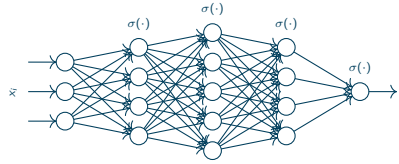
Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Output units



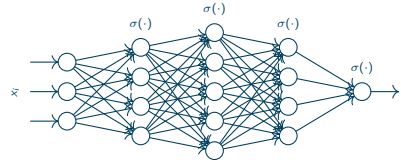
Output units



Interpretation: neural network with N layers:

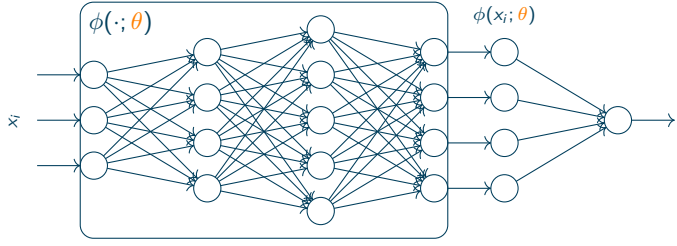
- first $N-1$ layers perform "feature extraction",
- last layer performs linear regression/classification $\rightarrow$ **what output units**?

Output units



Interpretation: neural network with N layers:

- first $N-1$ layers perform "feature extraction",
- last layer performs linear regression/classification $\rightarrow$ **what output units**?



Output units

Denote output of the $(L-1)$th layer by $\mathbf{z}^{(L-1)}$. How to choose Lth layer for predicting $\widehat{\mathbf{y}}$?

Output units

Denote output of the $(L-1)$th layer by $\mathbf{z}^{(L-1)}$. How to choose Lth layer for predicting $\widehat{\mathbf{y}}$?

- **Linear output unit** (if output is continuous):

$$\widehat{\mathbf{y}} = W^{(L)}\mathbf{z}^{(L-1)} + b^{(L)}$$

$\rightarrow$ Linear regression based on new variables (features) $\mathbf{z}^{(L-1)}$.

Output units

Denote output of the $(L-1)$th layer by $\mathbf{z}^{(L-1)}$. How to choose Lth layer for predicting $\widehat{\mathbf{y}}$?

- **Linear output unit** (if output is continuous):

$$\widehat{\mathbf{y}} = W^{(L)}\mathbf{z}^{(L-1)} + b^{(L)}$$

  $\rightarrow$ Linear regression based on new variables (features) $\mathbf{z}^{(L-1)}$.

- **Sigmoid output unit**, used to predict within two categories (e.g., $\{0, 1\}$) outputs:

$$\mathbb{P}(\hat{y} = 1 \,|\, \mathbf{z}^{(L-1)}) = \sigma(\mathbf{z}^{(L-1)}),$$

  where $\sigma(t) = e^t/(1+e^t)$.

  $\rightarrow$ Fully-connected layer + sigmoid output $\Rightarrow$ logistic regression based on new variables (features) $\mathbf{z}^{(L-1)}$.

Output units

Denote output of the $(L-1)$th layer by $\mathbf{z}^{(L-1)}$. How to choose Lth layer for predicting $\widehat{\mathbf{y}}$?

- **Linear output unit** (if output is continuous):

$$\widehat{\mathbf{y}} = W^{(L)}\mathbf{z}^{(L-1)} + b^{(L)}$$

  $\rightarrow$ Linear regression based on new variables (features) $\mathbf{z}^{(L-1)}$.

- **Sigmoid output unit**, used to predict within two categories (e.g., $\{0, 1\}$) outputs:

$$\mathbb{P}(\hat{y} = 1 \,|\, \mathbf{z}^{(L-1)}) = \sigma(\mathbf{z}^{(L-1)}),$$

  where $\sigma(t) = e^t/(1+e^t)$.

  $\rightarrow$ Fully-connected layer + sigmoid output $\Rightarrow$ logistic regression based on new variables (features) $\mathbf{z}^{(L-1)}$.

- **Softmax output unit**, used to predict $\{1, \ldots, K\}$:

$$\mathbb{P}(\hat{y} = i \,|\, \mathbf{z}^{(L-1)}) = \left[\mathrm{softmax}(\mathbf{z}^{(L-1)})\right]_i = \frac{e^{\mathbf{z}_i^{(L-1)}}}{\sum_{k=1}^{K} e^{\mathbf{z}_k^{(L-1)}}}.$$

  $\rightarrow$ Fully-connected layer + softmax output $\Rightarrow$ multinomial logistic regression based on new variables (features) $\mathbf{z}^{(L-1)}$.

Multiclass regression via softmax + cross-entropy

Softmax, used with cross-entropy:

$$\begin{aligned} -\log(\mathbb{P}(Y = y|\mathbf{z}^{(L-1)})) &= -\log\left[\mathrm{softmax}(\mathbf{z}^{(L-1)})\right]_y \\ &= -\mathbf{z}_y^{(L-1)} + \log\left(\sum_j e^{\mathbf{z}_j^{(L-1)}}\right) \\ &\approx \max_j \mathbf{z}_j^{(L-1)} - \mathbf{z}_y^{(L-1)}, \end{aligned}$$

this is also called **LogSumExp** (LSE) or **RealSoftMax**.

Multiclass regression via softmax + cross-entropy

Softmax, used with cross-entropy:

$$\begin{aligned} -\log(\mathbb{P}(Y = y|\mathbf{z}^{(L-1)})) &= -\log\left[\mathrm{softmax}(\mathbf{z}^{(L-1)})\right]_y \\ &= -\mathbf{z}_y^{(L-1)} + \log\left(\sum_j e^{\mathbf{z}_j^{(L-1)}}\right) \\ &\approx \max_j \mathbf{z}_j^{(L-1)} - \mathbf{z}_y^{(L-1)}, \end{aligned}$$

this is also called **LogSumExp** (LSE) or **RealSoftMax**.

Approximately no contribution to the cost when $\left[\mathrm{softmax}(\mathbf{z}^{(L-1)})\right]_{\hat{y}}$ is maximal.

Multiclass regression via softmax + cross-entropy

Softmax, used with cross-entropy:

$$\begin{aligned} -\log(\mathbb{P}(Y = y|\mathbf{z}^{(L-1)})) &= -\log\left[\mathrm{softmax}(\mathbf{z}^{(L-1)})\right]_y \\ &= -\mathbf{z}_y^{(L-1)} + \log\left(\sum_j e^{\mathbf{z}_j^{(L-1)}}\right) \\ &\approx \max_j \mathbf{z}_j^{(L-1)} - \mathbf{z}_y^{(L-1)}, \end{aligned}$$

this is also called **LogSumExp** (LSE) or **RealSoftMax**.

Approximately no contribution to the cost when $\left[\mathrm{softmax}(\mathbf{z}^{(L-1)})\right]_{\hat{y}}$ is maximal.

Lateral inhibition: (`https://en.wikipedia.org/wiki/Lateral_inhibition`)

(biology bonus) believed to exist between nearby neurons in the cortex. When the difference between the max and the other is large, winner takes all: one neuron is set to 1 and the others go to zero.

Multiclass logistic regression—via maximum likelihood

Generalization of logistic regression for multiclass outputs: for all $1 \leqslant y \leqslant K$, the model is

$$\mathbb{P}(Y = y \,|\, X = x) = \frac{e^{\langle \tilde{w}_y \,;\, x \rangle}}{\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x \rangle}},$$

for some $\tilde{w}_1, \ldots, \tilde{w}_K \in \mathbb{R}^d$.

Multiclass logistic regression—via maximum likelihood

Generalization of logistic regression for multiclass outputs: for all $1 \leqslant y \leqslant K$, the model is

$$\mathbb{P}(Y = y \mid X = x) = \frac{e^{\langle \tilde{w}_y \,;\, x \rangle}}{\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x \rangle}},$$

for some $\tilde{w}_1, \ldots, \tilde{w}_K \in \mathbb{R}^d$.

Likelihood (which we aim to maximize) similar to binary logistic regression:

$$\Pi_{i=1}^{n} \mathbb{P}(Y = y_i \mid X = x_i) = \Pi_{i=1}^{n} \frac{e^{\langle \tilde{w}_{y_i} \,;\, x_i \rangle}}{\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x_i \rangle}}$$

Multiclass logistic regression—via maximum likelihood

Generalization of logistic regression for multiclass outputs: for all $1 \leqslant y \leqslant K$, the model is

$$\mathbb{P}(Y = y \,|\, X = x) = \frac{e^{\langle \tilde{w}_y \,;\, x \rangle}}{\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x \rangle}},$$

for some $\tilde{w}_1, \ldots, \tilde{w}_K \in \mathbb{R}^d$.

Likelihood (which we aim to maximize) similar to binary logistic regression:

$$\Pi_{i=1}^{n} \mathbb{P}(Y = y_i \,|\, X = x_i) = \Pi_{i=1}^{n} \frac{e^{\langle \tilde{w}_{y_i} \,;\, x_i \rangle}}{\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x_i \rangle}}$$

and negative log-likelihood (to be minimized)

$$\begin{aligned} -\sum_{i=1}^{n} \log\left(\mathbb{P}(Y = y_i \,|\, X = x_i)\right) &= -\sum_{i=1}^{n} \log\left(\frac{e^{\langle \tilde{w}_{y_i} \,;\, x_i \rangle}}{\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x_i \rangle}}\right) \\ &= \sum_{i=1}^{n} \log\left(\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x_i \rangle}\right) - \langle \tilde{w}_{y_i} \,;\, x_i \rangle. \end{aligned}$$

Multiclass logistic regression—via maximum likelihood

Generalization of logistic regression for multiclass outputs: for all $1 \leqslant y \leqslant K$, the model is

$$\mathbb{P}(Y = y \,|\, X = x) = \frac{e^{\langle \tilde{w}_y \,;\, x \rangle}}{\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x \rangle}},$$

for some $\tilde{w}_1, \ldots, \tilde{w}_K \in \mathbb{R}^d$.

Likelihood (which we aim to maximize) similar to binary logistic regression:

$$\Pi_{i=1}^{n} \mathbb{P}(Y = y_i \,|\, X = x_i) = \Pi_{i=1}^{n} \frac{e^{\langle \tilde{w}_{y_i} \,;\, x_i \rangle}}{\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x_i \rangle}}$$

and negative log-likelihood (to be minimized)

$$\begin{aligned} -\sum_{i=1}^{n} \log\left(\mathbb{P}(Y = y_i \,|\, X = x_i)\right) &= -\sum_{i=1}^{n} \log\left(\frac{e^{\langle \tilde{w}_{y_i} \,;\, x_i \rangle}}{\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x_i \rangle}}\right) \\ &= \sum_{i=1}^{n} \log\left(\sum_{k=1}^{K} e^{\langle \tilde{w}_k \,;\, x_i \rangle}\right) - \langle \tilde{w}_{y_i} \,;\, x_i \rangle. \end{aligned}$$

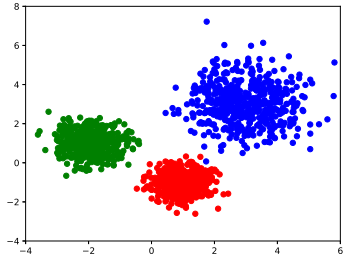
Set $K = 2$. Do we recover logistic regression? How?

Multiclass regression

Random generation of three classes ($x \in \mathbb{R}^2$), see first lab session:

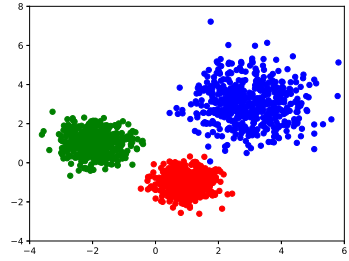
Multiclass regression

Random generation of three classes ($x \in \mathbb{R}^2$), see first lab session:



Multiclass regression

Random generation of three classes ($x \in \mathbb{R}^2$), see first lab session:

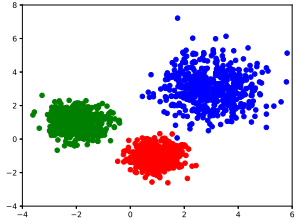


$\rightarrow$ you should be able to code & train model for this (after labs 3 & 4)!

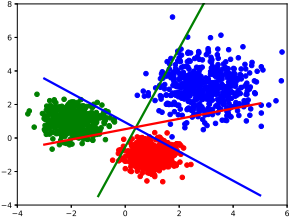
```
import torch

model = torch.nn.Sequential(
            torch.nn.Linear(2,3),
            torch.nn.Softmax(dim=1)
)
loss_fn = torch.nn.CrossEntropyLoss(reduction='mean')
```

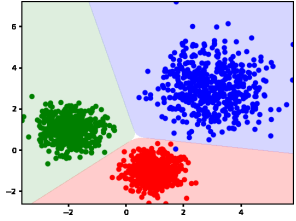
Multiclass regression

Random generation of three classes ($x \in \mathbb{R}^2$), see first lab sessions:



Outputs:





Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Cost functions

- Mean squared error (MSE):

$$\frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f_\theta(\mathbf{X}_i)) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - f_\theta(\mathbf{X}_i))^2.$$

Cost functions

- Mean squared error (MSE):

$$\frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f_\theta(\mathbf{X}_i)) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - f_\theta(\mathbf{X}_i))^2.$$

- Mean absolute error (MAE):

$$\frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f_\theta(\mathbf{X}_i)) = \frac{1}{n}\sum_{i=1}^{n}|Y_i - f_\theta(\mathbf{X}_i)|.$$

Cost functions

- Mean squared error (MSE):

$$\frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f_\theta(\mathbf{X}_i)) = \frac{1}{n}\sum_{i=1}^{n}(Y_i - f_\theta(\mathbf{X}_i))^2.$$

- Mean absolute error (MAE):

$$\frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f_\theta(\mathbf{X}_i)) = \frac{1}{n}\sum_{i=1}^{n}|Y_i - f_\theta(\mathbf{X}_i)|.$$

- 0-1 error:

$$\frac{1}{n}\sum_{i=1}^{n}\ell(Y_i, f_\theta(\mathbf{X}_i)) = \frac{1}{n}\sum_{i=1}^{n}\mathbb{1}_{Y_i \neq f_\theta(\mathbf{X}_i)}.$$

Cost functions

- Cross entropy (or negative log-likelihood):

$$\ell(y_i, f_\theta(\mathbf{x}_i)) = -\log\left([f_\theta(\mathbf{x}_i)]_{y_i}\right)$$

Cross-entropy:

Cost functions

- Cross entropy (or negative log-likelihood):

$$\ell(y_i, f_\theta(\mathbf{x}_i)) = -\log\left([f_\theta(\mathbf{x}_i)]_{y_i}\right)$$

Cross-entropy:

- Very popular!

Cost functions

- Cross entropy (or negative log-likelihood):

$$\ell(y_i, f_\theta(\mathbf{x}_i)) = -\log\left([f_\theta(\mathbf{x}_i)]_{y_i}\right)$$

Cross-entropy:

- Very popular!
- Interpretation: comparison of probability distribution (related to KL divergence).

Cost functions

- Cross entropy (or negative log-likelihood):

$$\ell(y_i, f_\theta(\mathbf{x}_i)) = -\log\left([f_\theta(\mathbf{x}_i)]_{y_i}\right)$$

Cross-entropy:

- Very popular!
- Interpretation: comparison of probability distribution (related to KL divergence).
- Helps to prevent saturation phenomenon compared to MSE. E.g., for $y \in \{-1, 1\}$

$$-\log(\mathbb{P}(Y = y|\mathbf{X} = \mathbf{x})) = -\log(\sigma(y(W^{(L)}\mathbf{z}^{(L)} + b^{(L)})),$$

with

$$\sigma(t) = \frac{e^t}{1 + e^t}$$

When $y(W^{(L)}\mathbf{z}^{(L)} + b^{(L)}) \ll -1$, we have $\sigma(t) \approx e^{-t}$

$\rightarrow$ $-\log(\mathbb{P}(Y = y_i|X))$ is approximately linear in W and b,

$\rightarrow$ gradient is simple to compute; gradient descent easy to implement!

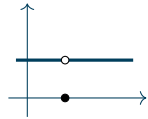
MSE should not be used with softmax output units!

["Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition", Bridle 1990]

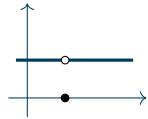
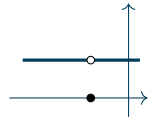
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		

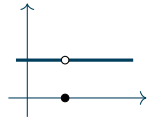
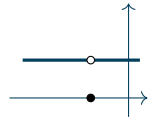
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		

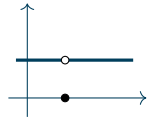
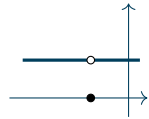
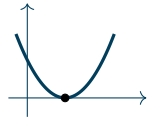
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		

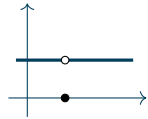
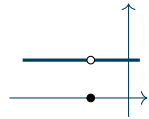
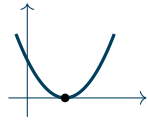
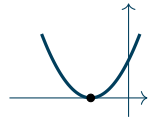
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		
quadratic loss	$\ell(y, y_p) = (y_p - y)^2$		

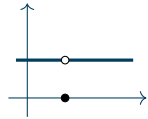
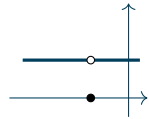
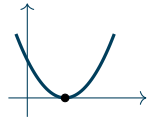
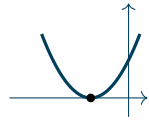
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		
quadratic loss	$\ell(y, y_p) = (y_p - y)^2$		

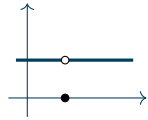
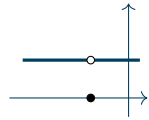
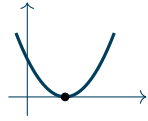
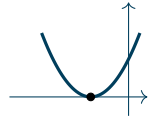
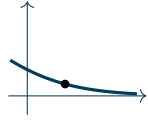
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		
quadratic loss	$\ell(y, y_p) = (y_p - y)^2$		

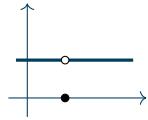
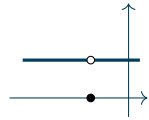
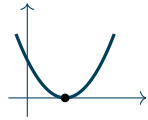
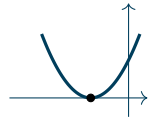
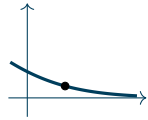
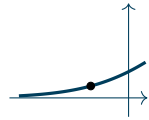
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		
quadratic loss	$\ell(y, y_p) = (y_p - y)^2$		
logistic loss	$\ell(y, y_p) = \log\left(1 + \exp(-y_p y)\right)$		

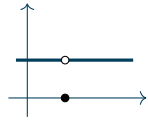
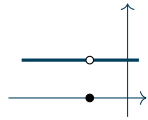
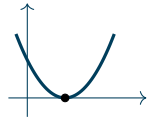
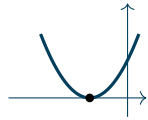
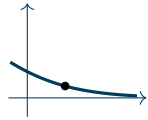
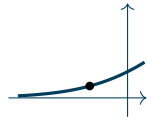
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		
quadratic loss	$\ell(y, y_p) = (y_p - y)^2$		
logistic loss	$\ell(y, y_p) = \log\left(1 + \exp(-y_p y)\right)$		

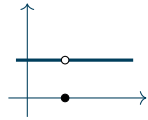
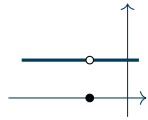
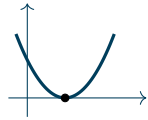
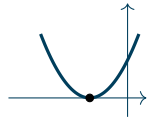
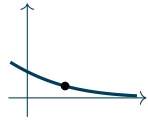
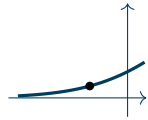
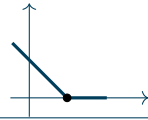
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		
quadratic loss	$\ell(y, y_p) = (y_p - y)^2$		
logistic loss	$\ell(y, y_p) = \log\left(1 + \exp(-y_p y)\right)$		

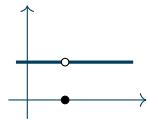
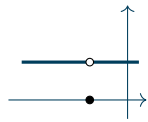
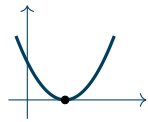
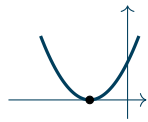
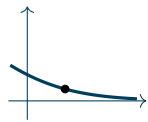
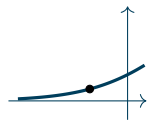
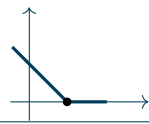
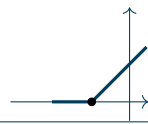
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		
quadratic loss	$\ell(y, y_p) = (y_p - y)^2$		
logistic loss	$\ell(y, y_p) = \log\left(1 + \exp(-y_p y)\right)$		
hinge loss	$\ell(y, y_p) = \max\{0, 1 - y_p y\}$		

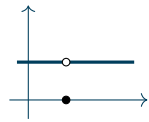
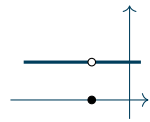
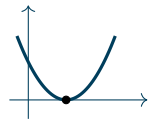
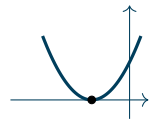
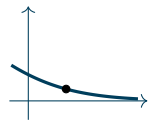
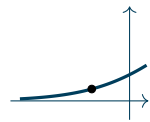
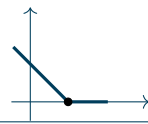
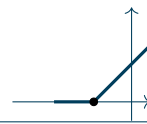
Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		
quadratic loss	$\ell(y, y_p) = (y_p - y)^2$		
logistic loss	$\ell(y, y_p) = \log\left(1 + \exp(-y_p y)\right)$		
hinge loss	$\ell(y, y_p) = \max\{0, 1 - y_p y\}$		

Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		
quadratic loss	$\ell(y, y_p) = (y_p - y)^2$		
logistic loss	$\ell(y, y_p) = \log(1 + \exp(-y_p y))$		
hinge loss	$\ell(y, y_p) = \max\{0, 1 - y_p y\}$		

Losses for classification (1/2)

Name	$\ell(y, y_p)$	Graph $\ell(1, y_p)$	Graph $\ell(-1, y_p)$
$0-1$ loss	$\ell(y, y_p) = \begin{cases} 0 & \text{if } y_p = y \\ 1 & \text{if } y_p \neq y \end{cases}$		
quadratic loss	$\ell(y, y_p) = (y_p - y)^2$		
logistic loss	$\ell(y, y_p) = \log(1 + \exp(-y_p y))$		
hinge loss	$\ell(y, y_p) = \max\{0, 1 - y_p y\}$		

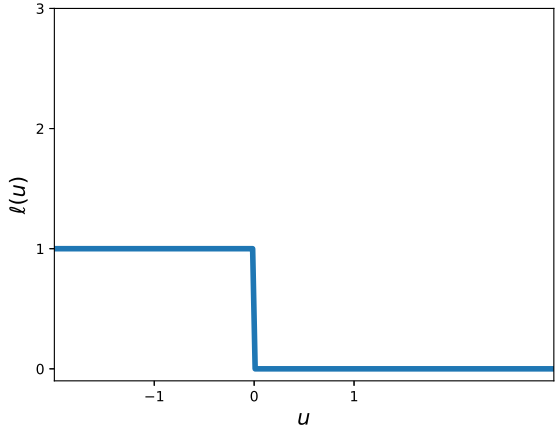
Notice symmetry $\ell(y, y_p) = \ell(1, y_p y) \Rightarrow$ common notation is $\ell(y_p y)$ (one argument only).

Losses for classification (2/2)

Notation $y \in \{-1, 1\}$, cost of misclassification (using $u = y_p y$)

Losses for classification (2/2)

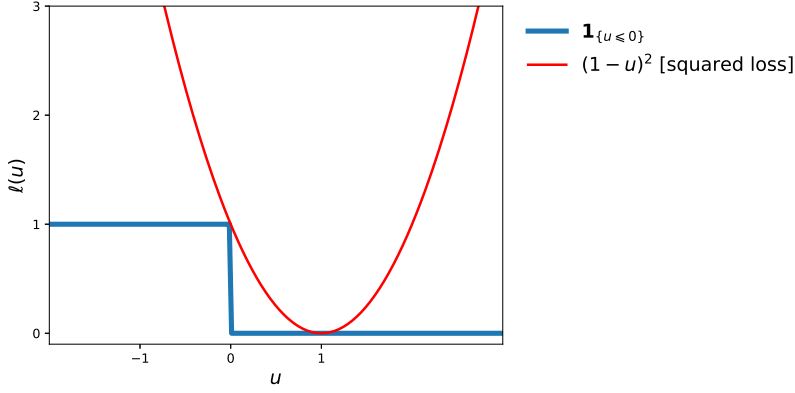
Notation $y \in \{-1, 1\}$, cost of misclassification (using $u = y_p y$)



$\mathbf{1}_{\{u \leqslant 0\}}$

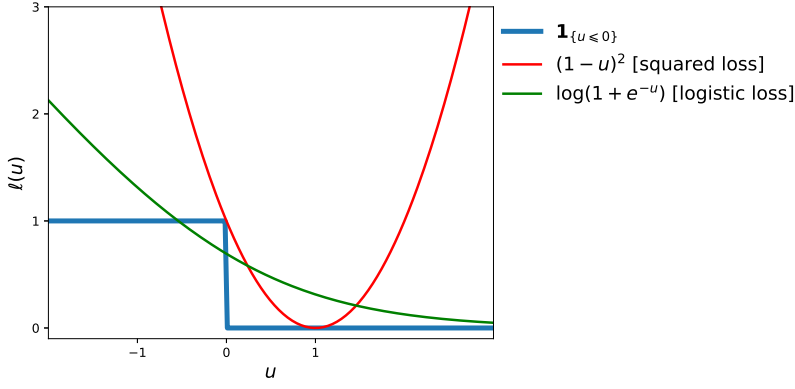
Losses for classification (2/2)

Notation $y \in \{-1, 1\}$, cost of misclassification (using $u = y_p y$)



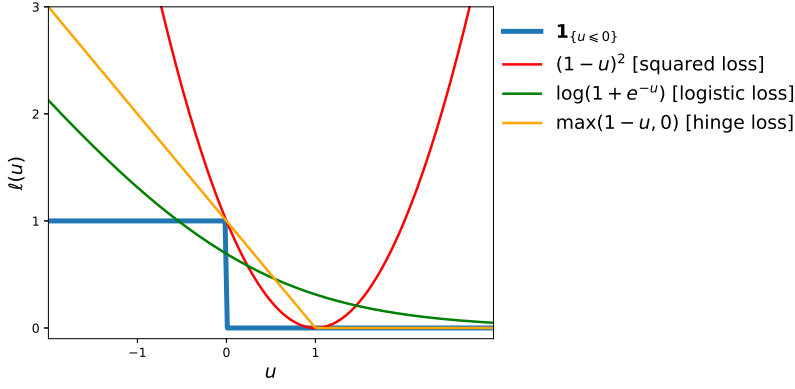
Losses for classification (2/2)

Notation $y \in \{-1, 1\}$, cost of misclassification (using $u = y_p y$)



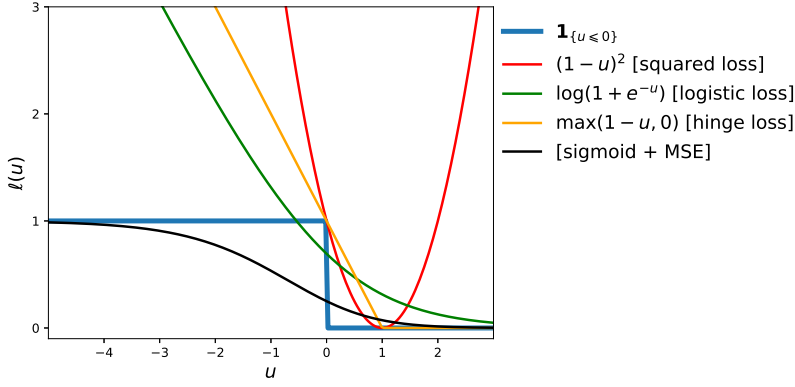
Losses for classification (2/2)

Notation $y \in \{-1, 1\}$, cost of misclassification (using $u = y_p y$)



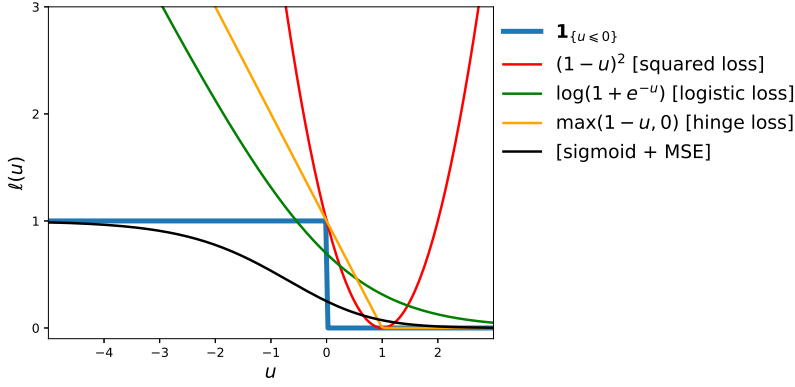
Losses for classification (2/2)

Notation $y \in \{-1, 1\}$, cost of misclassification (using $u = y_p y$)



Losses for classification (2/2)

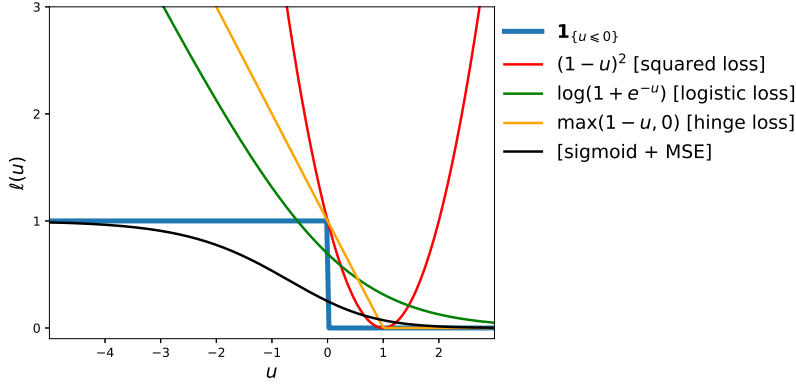
Notation $y \in \{-1, 1\}$, cost of misclassification (using $u = y_p y$)



⚠ MSE should not be used with softmax output units!

Losses for classification (2/2)

Notation $y \in \{-1, 1\}$, cost of misclassification (using $u = y_p y$)



⚠ MSE should not be used with softmax output units!
For regression, usually MSE, sometimes MAE.

What did we see so far?

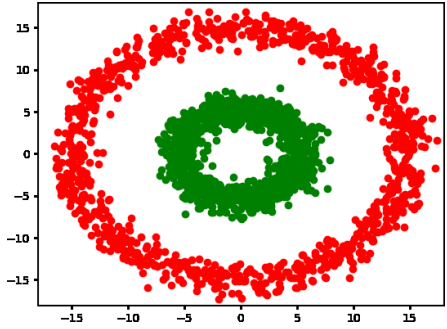
https://app.wooclap.com/YUKENL?from=instruction-slidelink to wooclap

What would you try here (activations, output units,loss)?

How to classify this?

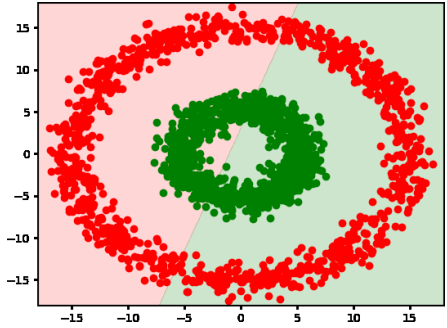
What would you try here (activations, output units,loss)?

How to classify this?



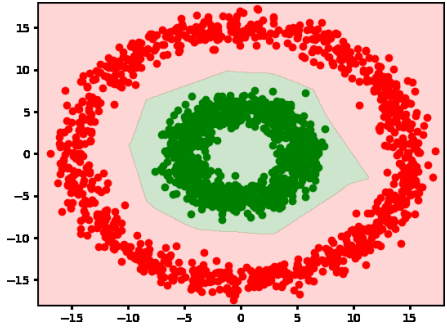
What would you try here (activations, output units, loss)?

Linear separation?



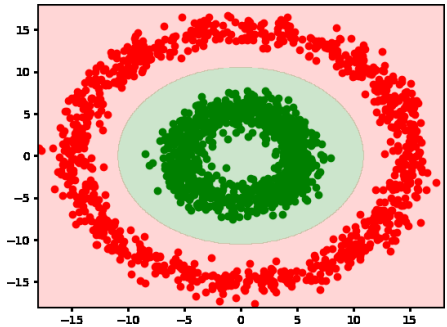
What would you try here (activations, output units, loss)?

Neural net? (here: 2×10 linear layer, ReLU, 10×2 linear layer, softmax, cross-entropy).



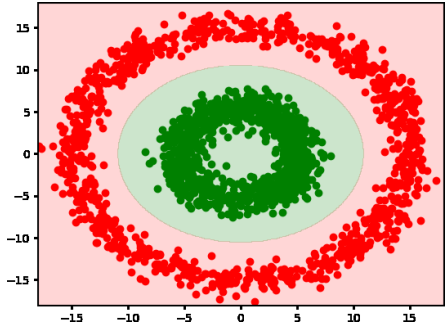
What would you try here (activations, output units, loss)?

Manually add x_1^2, x_2^2 + classical logistic regression:



What would you try here (activations, output units, loss)?

Manually add x_1^2, x_2^2 + classical logistic regression:



To go further, see also classical machine learning technique $\longrightarrow$ kernel methods (we will not cover it)!

Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Weight initialization

First idea: Set all weights and bias to the same value (zero?).

Weight initialization

First idea: Set all weights and bias to the same value (zero?).

Problems?

Weight initialization

First idea: Set all weights and bias to the same value (zero?).

Problems?

- network is symmetric: in each layer, each neuron express the same output,

Weight initialization

First idea: Set all weights and bias to the same value (zero?).

Problems?

- network is symmetric: in each layer, each neuron express the same output,
- network is symmetric, each layer can be reduced to single neuron,

Weight initialization

First idea: Set all weights and bias to the same value (zero?).

Problems?

- network is symmetric: in each layer, each neuron express the same output,
- network is symmetric, each layer can be reduced to single neuron,
- even worse if initialization is zero (dead neurons, etc.)

Weight initialization

First idea: Set all weights and bias to the same value (zero?).

Problems?

- network is symmetric: in each layer, each neuron express the same output,
- network is symmetric, each layer can be reduced to single neuron,
- even worse if initialization is zero (dead neurons, etc.)

$\rightarrow$ optimization/training is difficult

Weight initialization

First idea: Set all weights and bias to the same value (zero?).

Problems?

- network is symmetric: in each layer, each neuron express the same output,
- network is symmetric, each layer can be reduced to single neuron,
- even worse if initialization is zero (dead neurons, etc.)

$\rightarrow$ optimization/training is difficult

$\rightarrow$ proper weight initialization: random.

Weight initialization

First idea: Set all weights and bias to the same value (zero?).

Problems?

- network is symmetric: in each layer, each neuron express the same output,
- network is symmetric, each layer can be reduced to single neuron,
- even worse if initialization is zero (dead neurons, etc.)

$\rightarrow$ optimization/training is difficult

$\rightarrow$ proper weight initialization: random.

Further: no prior insight on signs of the weights

Weight initialization

First idea: Set all weights and bias to the same value (zero?).

Problems?

- network is symmetric: in each layer, each neuron express the same output,
- network is symmetric, each layer can be reduced to single neuron,
- even worse if initialization is zero (dead neurons, etc.)

$\rightarrow$ optimization/training is difficult

$\rightarrow$ proper weight initialization: random.

Further: no prior insight on signs of the weights

$\rightarrow$ make sense to have symmetric distributions around zero.

Small or big weights?

Consider initial weight distributions of type $\mathcal{N}(0, \sigma^2)$.

Small or big weights?

Consider initial weight distributions of type $\mathcal{N}(0, \sigma^2)$.

1. If the variance of the weights is too small ($\sigma^2 \ll 1$):
   - weights are approximately zero: there is no activation at all,

   $\rightarrow$ σ too small means difficult/impossible training.

Small or big weights?

Consider initial weight distributions of type $\mathcal{N}(0, \sigma^2)$.

1. If the variance of the weights is too small ($\sigma^2 \ll 1$):
   - weights are approximately zero: there is no activation at all,

   $\rightarrow$ σ too small means difficult/impossible training.

2. If the variance of the weights is too large ($\sigma^2 \gg 1$):
   - linear combinations very large: increased saturation phenomenon,

   $\rightarrow$ network might be difficult/impossible to train!

   $\rightarrow$ Similar to non-normalized data.

Small or big weights?

Consider initial weight distributions of type $\mathcal{N}(0, \sigma^2)$.

1. If the variance of the weights is too small ($\sigma^2 \ll 1$):
   - weights are approximately zero: there is no activation at all,

   $\rightarrow$ σ too small means difficult/impossible training.

2. If the variance of the weights is too large ($\sigma^2 \gg 1$):
   - linear combinations very large: increased saturation phenomenon,

   $\rightarrow$ network might be difficult/impossible to train!

   $\rightarrow$ Similar to non-normalized data.

3. In any case, **no need to tune the bias**: they can be initially set to zero.

Small or big weights?

Consider initial weight distributions of type $\mathcal{N}(0, \sigma^2)$.

1. If the variance of the weights is too small ($\sigma^2 \ll 1$):
   - weights are approximately zero: there is no activation at all,

   $\rightarrow$ σ too small means difficult/impossible training.

2. If the variance of the weights is too large ($\sigma^2 \gg 1$):
   - linear combinations very large: increased saturation phenomenon,

   $\rightarrow$ network might be difficult/impossible to train!

   $\rightarrow$ Similar to non-normalized data.

3. In any case, **no need to tune the bias**: they can be initially set to zero.
   If using ReLU's, positive bias might be encouraged.

Alternate initializations

Idea: variance of the input should be the same as the variance of the output.

Let w_j be any weight between layer j and layer $j+1$.

Alternate initializations

Idea: variance of the input should be the same as the variance of the output.

Let w_j be any weight between layer j and layer $j+1$.

1. **He et al. initialization**

   ["Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", He et al. 2015]

   Initialize bias to zero and weights randomly using

   $$W^{(j)} \sim \mathcal{N}\left(0, \frac{2}{n_j}\right),$$

   where n_j is the size of layer j. Intuition: keeps balanced variance input/output in case of ReLU layers.

Alternate initializations

Idea: variance of the input should be the same as the variance of the output.

Let w_j be any weight between layer j and layer $j+1$.

1. **He et al. initialization**

   ["Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", He et al. 2015]

   Initialize bias to zero and weights randomly using

   $$W^{(j)} \sim \mathcal{N}\left(0, \frac{2}{n_j}\right),$$

   where n_j is the size of layer j. Intuition: keeps balanced variance input/output in case of ReLU layers.

2. **Xavier initialization**

   ["Understanding the difficulty of training deep feedforward neural networks", Glorot and Bengio 2010]

   Initialize bias to zero and weights randomly using

   $$W^{(j)} \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right],$$

   where n_j is the size of layer j

   Intuition: also try to enforce expected gradient sizes to stay balanced accross layers.

Alternate initializations

Idea: variance of the input should be the same as the variance of the output.

Let w_j be any weight between layer j and layer $j+1$.

1. **He et al. initialization**

   ["Delving deep into rectifiers: Surpassing human-level performance on imagenet classification", He et al. 2015]

   Initialize bias to zero and weights randomly using

   $$W^{(j)} \sim \mathcal{N}\left(0, \frac{2}{n_j}\right),$$

   where n_j is the size of layer j. Intuition: keeps balanced variance input/output in case of ReLU layers.

2. **Xavier initialization**

   ["Understanding the difficulty of training deep feedforward neural networks", Glorot and Bengio 2010]

   Initialize bias to zero and weights randomly using

   $$W^{(j)} \sim \mathcal{U}\left[-\frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}, \frac{\sqrt{6}}{\sqrt{n_j + n_{j+1}}}\right],$$

   where n_j is the size of layer j

   Intuition: also try to enforce expected gradient sizes to stay balanced accross layers.

$\rightarrow$ check what is done in classical packages (e.g., Pytorch)!

Bonus: ["All you need is a good init", Mishkin and Matas 2015]

Exercise

Consider:

- neural network with two hidden layers,
- respectively with n_1 and n_2 neurons,
- equipped with linear activation functions,
- choice of weights: i.i.d. centered ($\mathbb{E}[W_{ij}^{(k)}] = 0$, $\mathbb{V}[W_{ij}^{(k)}] = \sigma_k^2$), zero bias.

Exercise

Consider:

- neural network with two hidden layers,
- respectively with n_1 and n_2 neurons,
- equipped with linear activation functions,
- choice of weights: i.i.d. centered ($\mathbb{E}[W_{ij}^{(k)}] = 0$, $\mathbb{V}[W_{ij}^{(k)}] = \sigma_k^2$), zero bias.

Questions:

1. Find simple condition on weights for output variance of hidden units to stay constant accross layers.

Exercise

Consider:

- neural network with two hidden layers,
- respectively with n_1 and n_2 neurons,
- equipped with linear activation functions,
- choice of weights: i.i.d. centered ($\mathbb{E}[W_{ij}^{(k)}] = 0$, $\mathbb{V}[W_{ij}^{(k)}] = \sigma_k^2$), zero bias.

Questions:

1. Find simple condition on weights for output variance of hidden units to stay constant accross layers.

2. Within the same network, find simple condition on weights for gradients to stay constant across layers.

Exercise

Consider:

- neural network with two hidden layers,
- respectively with n_1 and n_2 neurons,
- equipped with linear activation functions,
- choice of weights: i.i.d. centered ($\mathbb{E}[W_{ij}^{(k)}] = 0$, $\mathbb{V}[W_{ij}^{(k)}] = \sigma_k^2$), zero bias.

Questions:

1. Find simple condition on weights for output variance of hidden units to stay constant accross layers.
2. Within the same network, find simple condition on weights for gradients to stay constant across layers.
3. Based on previous answers: propose simple way to initialize weights.

(see ["Understanding the difficulty of training deep feedforward neural networks", Glorot and Bengio 2010])

Solution

4. Consider the neuron i of the second hidden layer. The output of this neuron is

$$z_i^{(2)} = \sum_{j=1}^{n_2} W_{ij}^{(2)} z_j^{(1)}$$

$$= \sum_{j=1}^{n_2} W_{ij}^{(2)} \Big(\sum_{k=1}^{n_1} W_{jk}^{(1)} x_k \Big).$$

At fixed x, since the weights are centered, we have

$$\mathbb{V}\Big[z_i^{(2)}\Big] = \mathbb{V}\Big[\sum_{j=1}^{n_2}\sum_{k=1}^{n_1} W_{ij}^{(2)} W_{jk}^{(1)} x_k\Big] = \sum_{j=1}^{n_2}\sum_{k=1}^{n_1} x_k^2 \mathbb{V}\Big[W_{ij}^{(2)} W_{jk}^{(1)}\Big] = \sum_{j=1}^{n_2} \sigma_2^2 \sum_{k=1}^{n_1} x_k^2 \sigma_1^2,$$

$$\mathbb{V}\Big[z_i^{(1)}\Big] = \mathbb{V}\Big[\sum_{k=1}^{n_1} W_{ik}^{(1)} x_k\Big] = \sum_{k=1}^{n_1} x_k \sigma_1^2,$$

where σ_1^2 (resp. σ_2^2) is the shared variance of all weights from layer 1 (resp. layer 2). Since we want that $\mathbb{V}\Big[z_i^{(2)}\Big] = \mathbb{V}\Big[z_i^{(1)}\Big]$, it is sufficient that

$$n_2 \sigma_2^2 = 1.$$

Solution

2. Consider a neural network where the cost function is given, for one training example, by

$$C = \frac{1}{2}\Big(y - \sum_{j=1}^{n_d} W_j z_j^{(d)}\Big)^2,$$

which implies

$$\frac{\partial C}{\partial z_j^{(d)}} = -W_j\Big(y - \sum_{i=1}^{n_d} W_i z_i^{(d)}\Big).$$

Hence,

$$\mathbb{E}\Big[\frac{\partial C}{\partial z_j^{(d)}}\Big] = \mathbb{E}\Big[{W_j}^2 z_j^{(d)}\Big] = \sigma_{d+1}^2 \mathbb{E}\Big[z_j^{(d)}\Big].$$

Besides, according to the chain rule, we have

$$\frac{\partial C}{\partial z_k^{(d-1)}} = \sum_{j=1}^{n_d} \frac{\partial C}{\partial z_j^{(d)}} \frac{\partial z_j^{(d)}}{\partial z_k^{(d-1)}} = \sum_{j=1}^{n_d} \frac{\partial C}{\partial z_j^{(d)}} W_{jk}^{(d)}.$$

Solution

More precisely,

$$\mathbb{E}\Big[\frac{\partial C}{\partial z_j^{(d)}} W_{jk}^{(d)}\Big] = \mathbb{E}\Big[-W_j y W_{jk}^{(d)} + W_j W_{jk}^{(d)} \sum_{i=1}^{n_d} W_i \Big(\sum_{\ell=1}^{n_{d-1}} W_{i\ell}^{(d)} z_\ell^{(d-1)} + b_i^{(d)}\Big)\Big]$$

$$= \mathbb{E}\Big[W_j{}^2 W_{jk}^{(d)} \Big(\sum_{\ell=1}^{n_{d-1}} W_{j\ell}^{(d)} z_\ell^{(d-1)} + b_j^{(d)}\Big)\Big]$$

$$= \mathbb{E}\Big[W_j{}^2 (W_{jk}^{(d)})^2 z_k^{(d-1)}\Big]$$

$$= \sigma_{d+1}^2 \sigma_d^2 \mathbb{E}\Big[z_k^{(d-1)}\Big].$$

Using the chain rule (see previous slide), we get

$$\mathbb{E}\Big[\frac{\partial C}{\partial z_k^{(d-1)}}\Big] = \mathbb{E}\Big[\frac{\partial C}{\partial z_j^{(d)}}\Big]$$

$$n_d \sigma_{d+1}^2 \sigma_d^2 \mathbb{E}\Big[z_\ell^{(d-1)}\Big] = \sigma_{d+1}^2 \mathbb{E}\Big[z_j^{(d)}\Big].$$

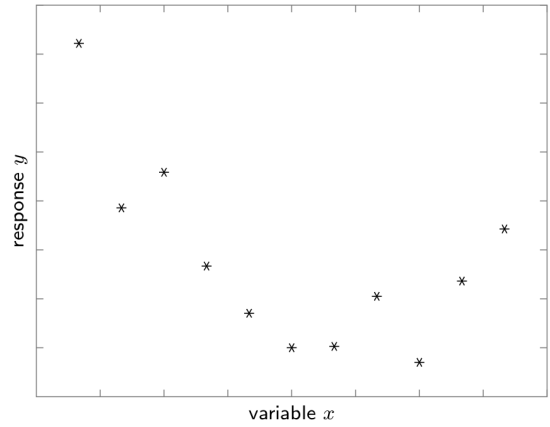
$\Rightarrow$ sufficient condition: $n_d \sigma_d^2 \sim 1$ (assuming activations of consecutive layers are similar).

Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

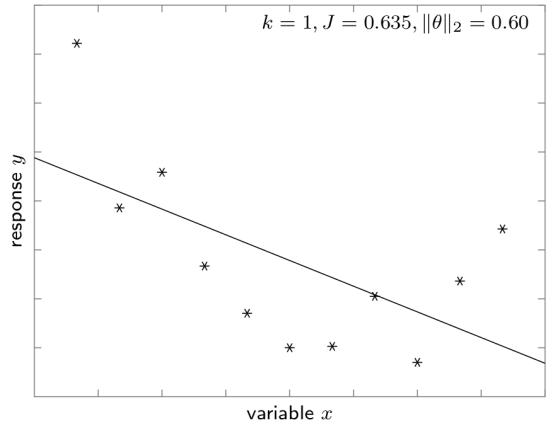
Polynomial regression

- kth degree polynomial regression problem (J is cost)



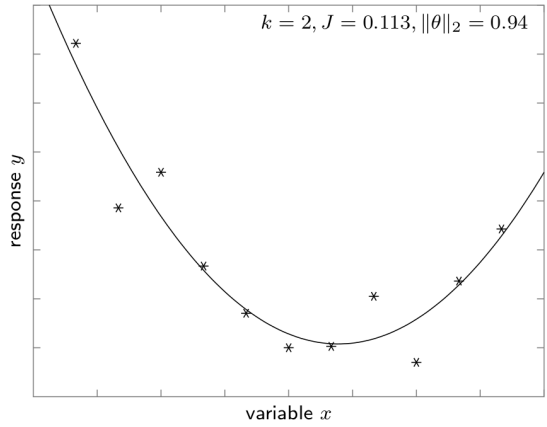
Polynomial regression

- kth degree polynomial regression problem (J is cost)



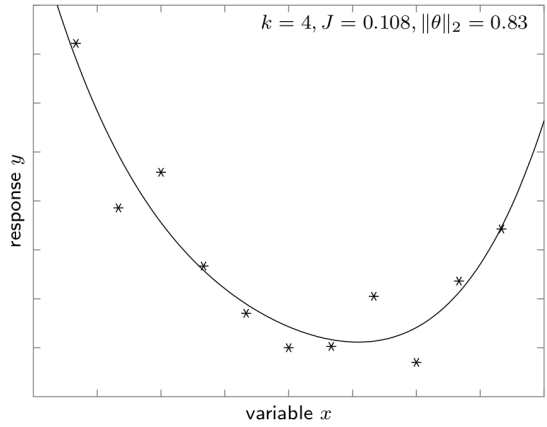
Polynomial regression

- kth degree polynomial regression problem (J is cost)



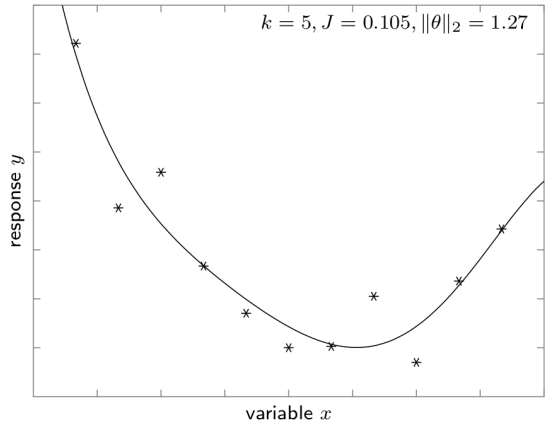
Polynomial regression

- kth degree polynomial regression problem (J is cost)



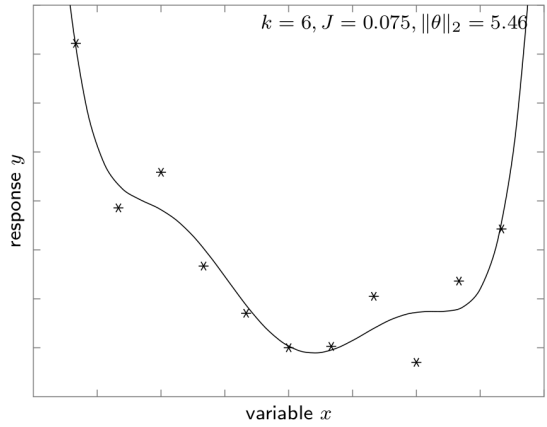
Polynomial regression

- kth degree polynomial regression problem (J is cost)



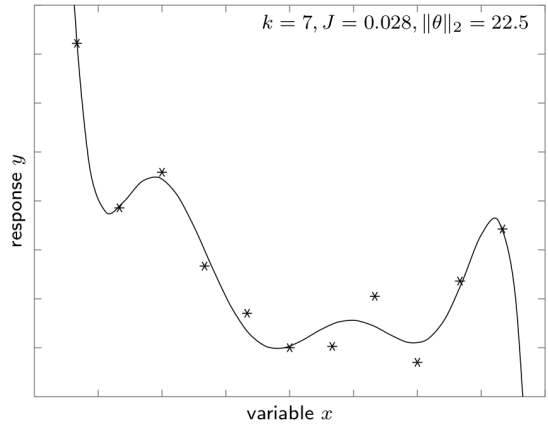
Polynomial regression

- kth degree polynomial regression problem (J is cost)



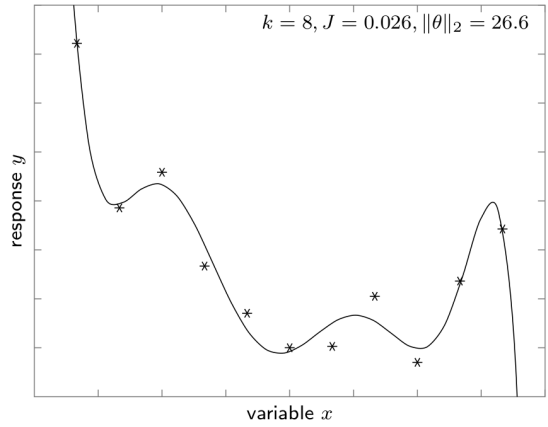
Polynomial regression

- kth degree polynomial regression problem (J is cost)



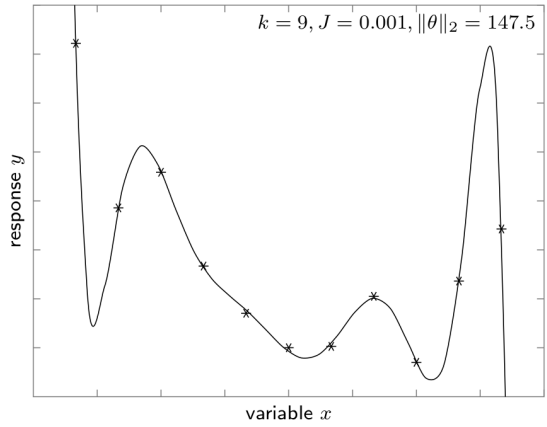
Polynomial regression

- kth degree polynomial regression problem (J is cost)



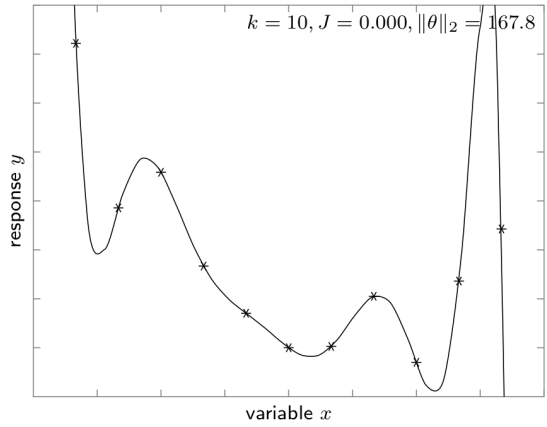
Polynomial regression

- kth degree polynomial regression problem (J is cost)



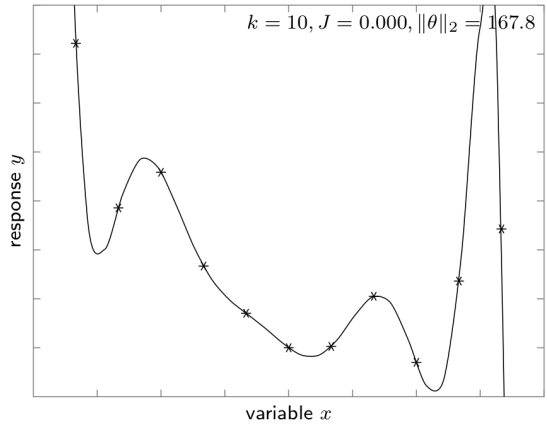
Polynomial regression

- kth degree polynomial regression problem (J is cost)



Polynomial regression

- kth degree polynomial regression problem (J is cost)



Overfitting and bias-variance decomposition

Overfitting and bias-variance decomposition

- Dataset $\{(x_i, y_i)\}$ with $y_i = f(x_i) + \epsilon_i$ with ϵ_i's i.i.d.

Overfitting and bias-variance decomposition

- Dataset $\{(x_i, y_i)\}$ with $y_i = f(x_i) + \epsilon_i$ with ϵ_i's i.i.d.
- estimate $f_{\hat{\theta}} \approx f$ based on data (so $f_{\hat{\theta}}$ is a random variable),

Overfitting and bias-variance decomposition

- Dataset $\{(x_i, y_i)\}$ with $y_i = f(x_i) + \epsilon_i$ with ϵ_i's i.i.d.
- estimate $f_{\hat{\theta}} \approx f$ based on data (so $f_{\hat{\theta}}$ is a random variable),
- decompose MSE:

$$\mathbb{E}[(f_{\hat{\theta}}(x) - f(x))^2] = \underbrace{(\mathbb{E}[f_{\hat{\theta}}(x)] - f(x))^2}_{\text{Bias}} + \underbrace{\mathbb{E}\left[(f_{\hat{\theta}}(x) - \mathbb{E}[f_{\hat{\theta}}(x)])^2\right]}_{\text{Variance}},$$

Overfitting and bias-variance decomposition

- Dataset $\{(x_i, y_i)\}$ with $y_i = f(x_i) + \epsilon_i$ with ϵ_i's i.i.d.
- estimate $f_{\hat{\theta}} \approx f$ based on data (so $f_{\hat{\theta}}$ is a random variable),
- decompose MSE:

$$\mathbb{E}[(f_{\hat{\theta}}(x) - f(x))^2] = \underbrace{(\mathbb{E}[f_{\hat{\theta}}(x)] - f(x))^2}_{\text{Bias}} + \underbrace{\mathbb{E}\left[(f_{\hat{\theta}}(x) - \mathbb{E}[f_{\hat{\theta}}(x)])^2\right]}_{\text{Variance}},$$

- add variance term for $\mathbb{E}[(f_{\hat{\theta}}(x) - y)^2] = \mathbb{E}[(f_{\hat{\theta}}(x) - f(x))^2] + \mathbb{E}(\epsilon^2)$ (with $\mathbb{E}[\epsilon] = 0$),

Overfitting and bias-variance decomposition

- Dataset $\{(x_i, y_i)\}$ with $y_i = f(x_i) + \epsilon_i$ with ϵ_i's i.i.d.
- estimate $f_{\hat{\theta}} \approx f$ based on data (so $f_{\hat{\theta}}$ is a random variable),
- decompose MSE:

$$\mathbb{E}[(f_{\hat{\theta}}(x) - f(x))^2] = \underbrace{(\mathbb{E}[f_{\hat{\theta}}(x)] - f(x))^2}_{\text{Bias}} + \underbrace{\mathbb{E}\left[(f_{\hat{\theta}}(x) - \mathbb{E}[f_{\hat{\theta}}(x)])^2\right]}_{\text{Variance}},$$

- add variance term for $\mathbb{E}[(f_{\hat{\theta}}(x) - y)^2] = \mathbb{E}[(f_{\hat{\theta}}(x) - f(x))^2] + \mathbb{E}(\epsilon^2)$ (with $\mathbb{E}[\epsilon] = 0$),
- note: expectation is over training dataset (i.e., averaged over many estimations of $\hat{\theta}$),

Overfitting and bias-variance decomposition

- Dataset $\{(x_i, y_i)\}$ with $y_i = f(x_i) + \epsilon_i$ with ϵ_i's i.i.d.
- estimate $f_{\hat{\theta}} \approx f$ based on data (so $f_{\hat{\theta}}$ is a random variable),
- decompose MSE:

$$\mathbb{E}[(f_{\hat{\theta}}(x) - f(x))^2] = \underbrace{(\mathbb{E}[f_{\hat{\theta}}(x)] - f(x))^2}_{\text{Bias}} + \underbrace{\mathbb{E}\left[(f_{\hat{\theta}}(x) - \mathbb{E}[f_{\hat{\theta}}(x)])^2\right]}_{\text{Variance}},$$

- add variance term for $\mathbb{E}[(f_{\hat{\theta}}(x) - y)^2] = \mathbb{E}[(f_{\hat{\theta}}(x) - f(x))^2] + \mathbb{E}(\epsilon^2)$ (with $\mathbb{E}[\epsilon] = 0$),
- note: expectation is over training dataset (i.e., averaged over many estimations of $\hat{\theta}$),
- (there is a more general statement beyond MSE by looking at the risk $\mathcal{R}(f_{\hat{\theta}})$).

Overfitting and bias-variance decomposition

- Dataset $\{(x_i, y_i)\}$ with $y_i = f(x_i) + \epsilon_i$ with ϵ_i's i.i.d.
- estimate $f_{\hat{\theta}} \approx f$ based on data (so $f_{\hat{\theta}}$ is a random variable),
- decompose MSE:

$$\mathbb{E}[(f_{\hat{\theta}}(x) - f(x))^2] = \underbrace{(\mathbb{E}[f_{\hat{\theta}}(x)] - f(x))^2}_{\text{Bias}} + \underbrace{\mathbb{E}\left[(f_{\hat{\theta}}(x) - \mathbb{E}[f_{\hat{\theta}}(x)])^2\right]}_{\text{Variance}},$$

- add variance term for $\mathbb{E}[(f_{\hat{\theta}}(x) - y)^2] = \mathbb{E}[(f_{\hat{\theta}}(x) - f(x))^2] + \mathbb{E}(\epsilon^2)$ (with $\mathbb{E}[\epsilon] = 0$),
- note: expectation is over training dataset (i.e., averaged over many estimations of $\hat{\theta}$),
- (there is a more general statement beyond MSE by looking at the risk $\mathcal{R}(f_{\hat{\theta}})$).

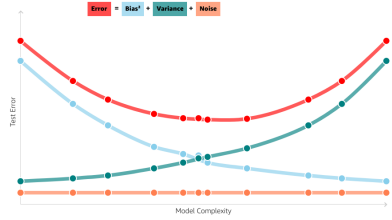


Image: https://mlu-explain.github.io/bias-variance/

Overfitting

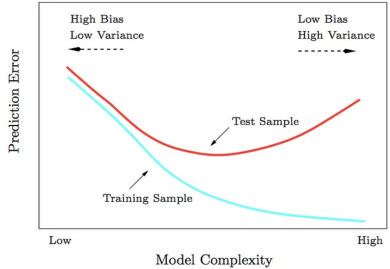


Image: https://towardsdatascience.com

Bias-variance decomposition: dataset (x_i, y_i) with $y_i = f(x_i) + \epsilon_i$.
Fight **overfitting** by imposing some constraints over the parameter space.

Reducing variance and **increasing bias**.

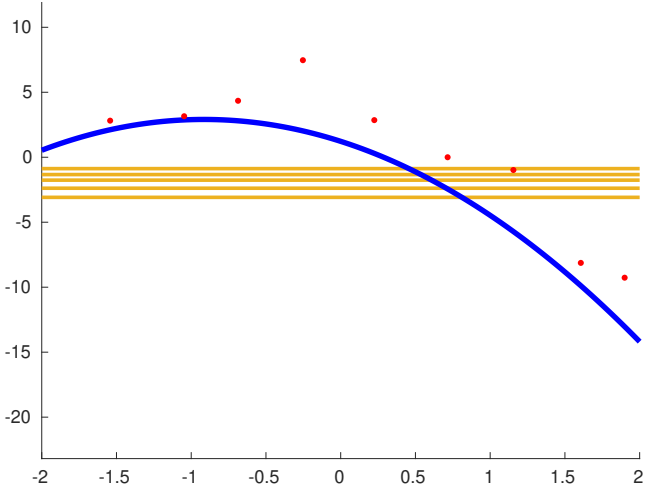
This is called the classical **bias-variane tradeoff**. Is that the end of the story?

Polynomial regression: bias-variance?

- acquisition process: $y_i = f(x_i) + \epsilon_i$ (f is quadratic here)
- 0th degree polynomial regression problem
- repeat many times the estimation process

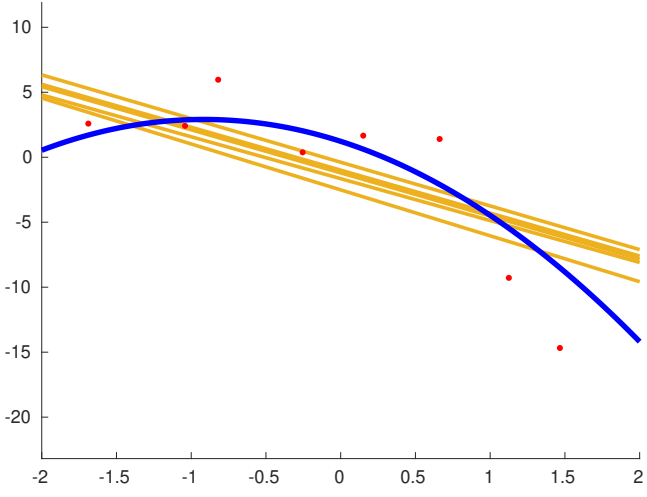
Polynomial regression: bias-variance?

- acquisition process: $y_i = f(x_i) + \epsilon_i$ (f is quadratic here)
- 0th degree polynomial regression problem
- repeat many times the estimation process



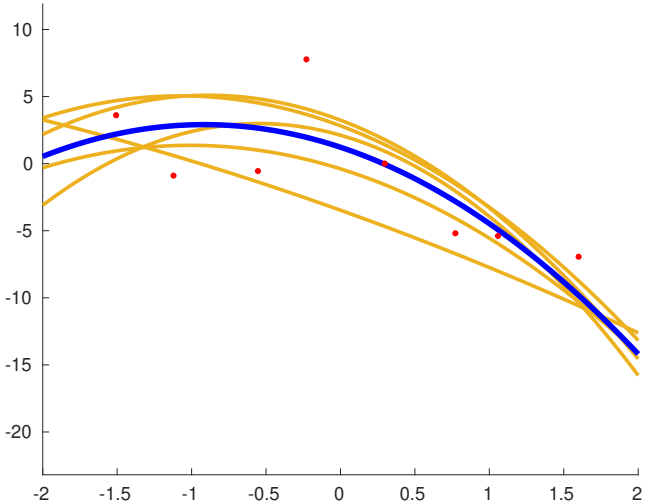
Polynomial regression: bias-variance?

- acquisition process: $y_i = f(x_i) + \epsilon_i$ (f is quadratic here)
- 1st degree polynomial regression problem
- repeat many times the estimation process



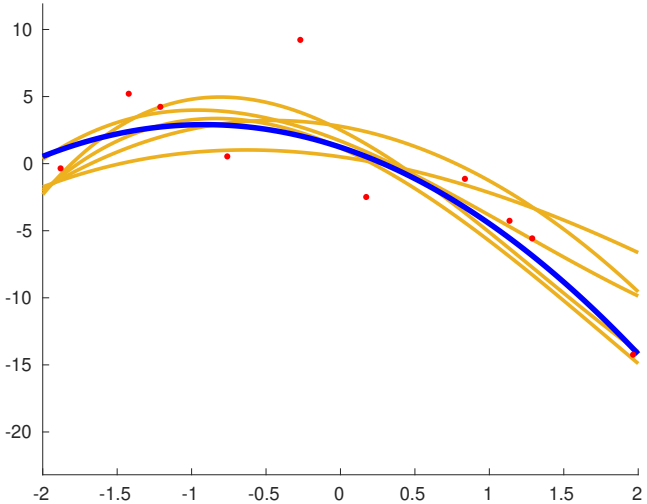
Polynomial regression: bias-variance?

- acquisition process: $y_i = f(x_i) + \epsilon_i$ (f is quadratic here)
- 2nd degree polynomial regression problem
- repeat many times the estimation process



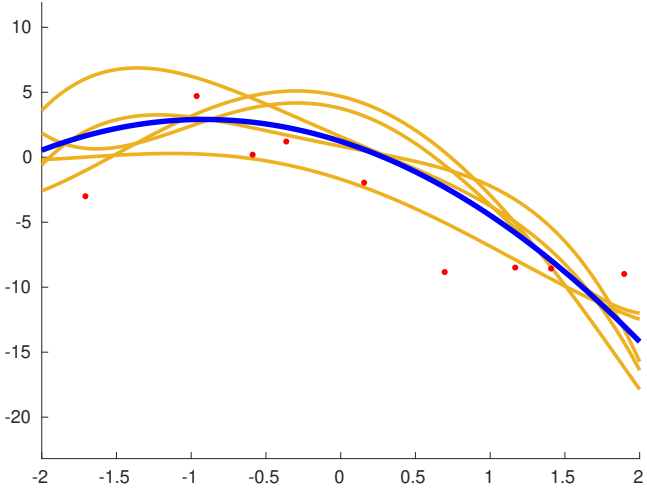
Polynomial regression: bias-variance?

- acquisition process: $y_i = f(x_i) + \epsilon_i$ (f is quadratic here)
- 3rd degree polynomial regression problem
- repeat many times the estimation process



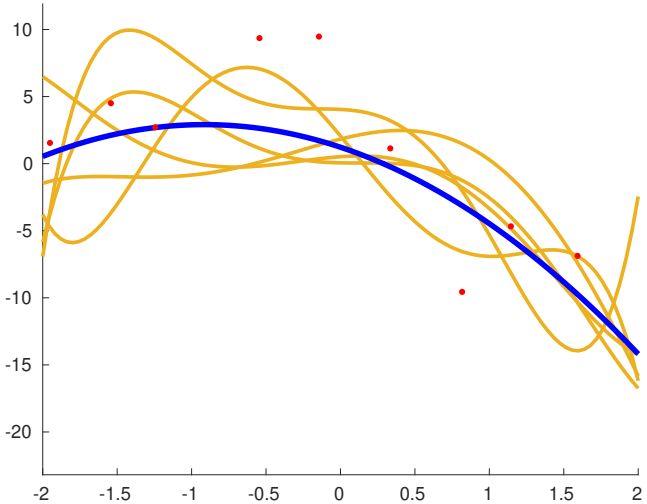
Polynomial regression: bias-variance?

- acquisition process: $y_i = f(x_i) + \epsilon_i$ (f is quadratic here)
- 4th degree polynomial regression problem
- repeat many times the estimation process



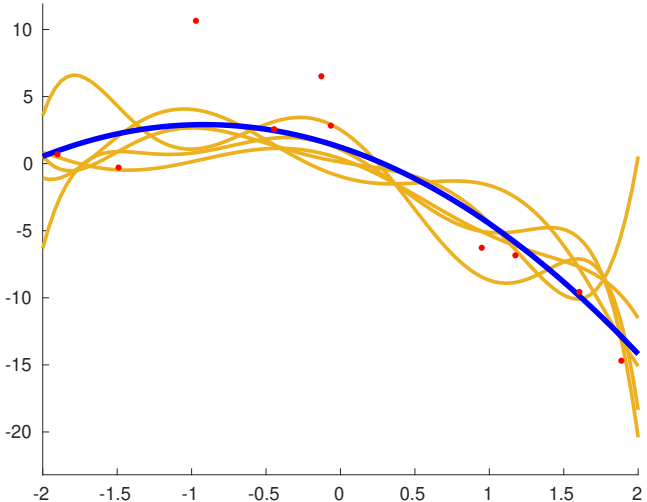
Polynomial regression: bias-variance?

- acquisition process: $y_i = f(x_i) + \epsilon_i$ (f is quadratic here)
- 5th degree polynomial regression problem
- repeat many times the estimation process

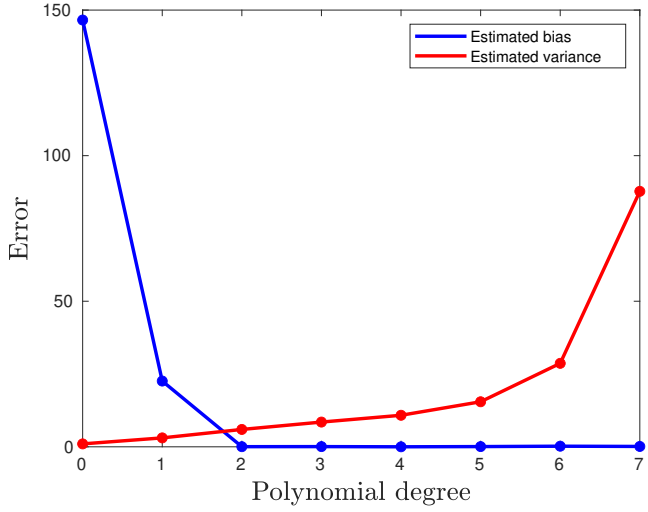


Polynomial regression: bias-variance?

- acquisition process: $y_i = f(x_i) + \epsilon_i$ (f is quadratic here)
- 6th degree polynomial regression problem
- repeat many times the estimation process



Polynomial regression: bias-variance



Overfitting

Many different techniques to **fight overfitting**.

Overfitting

Many different techniques to **fight overfitting**.

- **Validation procedures**

Overfitting

Many different techniques to **fight overfitting**.

- **Validation procedures**
- **Penalization (ℓ_1 or ℓ_2)**
  Replacing the cost function $\mathcal{L}$ by $\tilde{\mathcal{L}}(\theta, X, y) = \mathcal{L}(\theta, X, y) + \text{pen}(\theta)$.

Overfitting

Many different techniques to **fight overfitting**.

- **Validation procedures**
- **Penalization (ℓ_1 or ℓ_2)**
  Replacing the cost function $\mathcal{L}$ by $\tilde{\mathcal{L}}(\theta, X, y) = \mathcal{L}(\theta, X, y) + \text{pen}(\theta)$.
- **Soft weight sharing** (see lecture on CNNs)
  Reduce the parameter space artificially by imposing explicit constraints.

Overfitting

Many different techniques to **fight overfitting**.

- **Validation procedures**
- **Penalization (ℓ_1 or ℓ_2)**
  Replacing the cost function $\mathcal{L}$ by $\tilde{\mathcal{L}}(\theta, X, y) = \mathcal{L}(\theta, X, y) + \text{pen}(\theta)$.
- **Soft weight sharing** (see lecture on CNNs)
  Reduce the parameter space artificially by imposing explicit constraints.
- **Dropout**
  Randomly kill some neurons during optimization and predict with the full network.

Overfitting

Many different techniques to **fight overfitting**.

- **Validation procedures**
- **Penalization (ℓ_1 or ℓ_2)**
  Replacing the cost function $\mathcal{L}$ by $\tilde{\mathcal{L}}(\theta, X, y) = \mathcal{L}(\theta, X, y) + \text{pen}(\theta)$.
- **Soft weight sharing** (see lecture on CNNs)
  Reduce the parameter space artificially by imposing explicit constraints.
- **Dropout**
  Randomly kill some neurons during optimization and predict with the full network.
- **Batch normalization**
  Renormalize a layer inside a batch, so that the network does not overfit on this particular batch.

Overfitting

Many different techniques to **fight overfitting**.

- **Validation procedures**
- **Penalization (ℓ_1 or ℓ_2)**
  Replacing the cost function $\mathcal{L}$ by $\tilde{\mathcal{L}}(\theta, X, y) = \mathcal{L}(\theta, X, y) + \text{pen}(\theta)$.
- **Soft weight sharing** (see lecture on CNNs)
  Reduce the parameter space artificially by imposing explicit constraints.
- **Dropout**
  Randomly kill some neurons during optimization and predict with the full network.
- **Batch normalization**
  Renormalize a layer inside a batch, so that the network does not overfit on this particular batch.
- **Early stopping**
  Stop the gradient descent procedure when the error on the validation set increases.

Outline

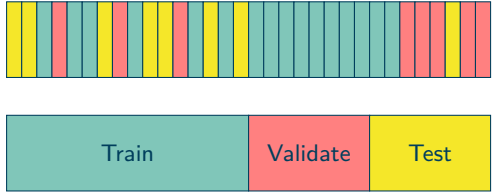
1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Validation

We search $\widehat{f} \in \mathcal{F}$ from the training data $\mathcal{D}_n$, gence: risk to overperform on $\mathcal{D}_n$.

$\rightarrow$ in practice: need to validate!

$\rightarrow$ assign (randomly) data to training, validation, or test sets.



Training set:

- solve training problem.

Validation set:

- estimate generalization performance of trained model(s),
- use this for assessing quality of training procedure and model selection.

Test set:

- final assessment on how chosen model generalizes to unseen data,
- *not* for model selection.

In deep learning: often no validation set (only validation and test sets).

k-fold cross validation

- Divide training/validate set into k data chunks.

k-fold cross validation

- Divide training/validate set into k data chunks.
- Train k models with $k-1$ chunks, use kth chunk for validation.

k-fold cross validation

- Divide training/validate set into k data chunks.
- Train k models with $k-1$ chunks, use kth chunk for validation.
- Loop
  1. Set hyperparameters and train all k models.
  2. Evaluate generalization score on its validation data.
  3. Sum scores to get model performance.

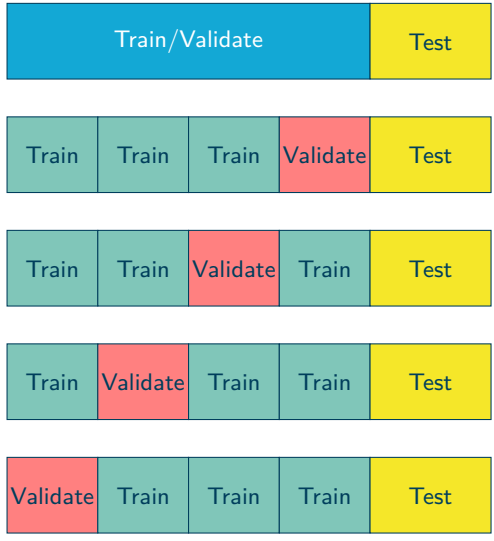
k-fold cross validation

- Divide training/validate set into k data chunks.
- Train k models with $k-1$ chunks, use kth chunk for validation.
- Loop
  1. Set hyperparameters and train all k models.
  2. Evaluate generalization score on its validation data.
  3. Sum scores to get model performance.
- Select final model hyperparameters based on best score.

k-fold cross validation

- Divide training/validate set into k data chunks.
- Train k models with $k-1$ chunks, use kth chunk for validation.
- Loop
  1. Set hyperparameters and train all k models.
  2. Evaluate generalization score on its validation data.
  3. Sum scores to get model performance.
- Select final model hyperparameters based on best score.
- Estimate generalization performance via test set.

4-fold cross validation – graphical representation



Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Penalization

Constrain the optimization problem

$$\min_{\theta} \mathcal{L}(\theta, X, y), \quad \text{s.t. } \text{pen}(\theta) \leqslant cste.$$

Penalization

Constrain the optimization problem

$$\min_{\theta} \mathcal{L}(\theta, X, y), \quad \text{s.t. } \text{pen}(\theta) \leqslant \textit{cste}.$$

Using Lagrangian formulation, this problem is equivalent to:

$$\min_{\theta} \mathcal{L}(\theta, X, y) + \lambda \, \text{pen}(\theta),$$

where

- $\mathcal{L}(\theta, X, y)$ is the loss function (data-driven term),
- pen is a function that increases when θ becomes more *complex* (penalty term),
- $\lambda \geqslant 0$ is a constant standing for the strength of the penalty term.

Penalization

Constrain the optimization problem

$$\min_{\theta} \mathcal{L}(\theta, X, y), \quad \text{s.t. } \text{pen}(\theta) \leqslant cste.$$

Using Lagrangian formulation, this problem is equivalent to:

$$\min_{\theta} \mathcal{L}(\theta, X, y) + \lambda \, \text{pen}(\theta),$$

where

- $\mathcal{L}(\theta, X, y)$ is the loss function (data-driven term),
- pen is a function that increases when θ becomes more *complex* (penalty term),
- $\lambda \geqslant 0$ is a constant standing for the strength of the penalty term.

For neural networks, pen$(\cdot)$ only penalizes weights (not biases).

Penalization: examples

$$\min_{\theta} \mathcal{L}(\theta, X, y) + \text{pen}(\theta),$$

Penalization: examples

$$\min_{\theta} \mathcal{L}(\theta, X, y) + \text{pen}(\theta),$$

- **Ridge**

$$\text{pen}(\theta) = \lambda \|\theta\|_2^2$$

["Ridge regression: Biased estimation for nonorthogonal problems", Hoerl and Kennard 1970], see also ["Lecture notes on ridge regression", Wieringen 2015]

Penalization: examples

$$\min_{\theta} \mathcal{L}(\theta, X, y) + \text{pen}(\theta),$$

- **Ridge**

$$\text{pen}(\theta) = \lambda \|\theta\|_2^2$$

["Ridge regression: Biased estimation for nonorthogonal problems", Hoerl and Kennard 1970], see also ["Lecture notes on ridge regression", Wieringen 2015]

- **LASSO**

$$\text{pen}(\theta) = \lambda \|\theta\|_1$$

["Regression shrinkage and selection via the lasso", Tibshirani 1996].

Penalization: examples

$$\min_{\theta} \mathcal{L}(\theta, X, y) + \text{pen}(\theta),$$

- **Ridge**

$$\text{pen}(\theta) = \lambda \|\theta\|_2^2$$

["Ridge regression: Biased estimation for nonorthogonal problems", Hoerl and Kennard 1970], see also ["Lecture notes on ridge regression", Wieringen 2015]

- **LASSO**

$$\text{pen}(\theta) = \lambda \|\theta\|_1$$

["Regression shrinkage and selection via the lasso", Tibshirani 1996].

- **Elastic Net**

$$\text{pen}(\theta) = \lambda \|\theta\|_2^2 + \mu \|\theta\|_1$$

["Regularization and variable selection via the elastic net", Zou and Hastie 2005]

Simple case: linear regression

Linear regression

The estimate of linear regression $\hat{\theta}$ is given by

$$\hat{\theta} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2,$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$.

Simple case: linear regression

Linear regression

The estimate of linear regression $\hat{\theta}$ is given by

$$\hat{\theta} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2,$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$.

Solution:

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T Y.$$

Simple case: linear regression

Linear regression

The estimate of linear regression $\hat{\theta}$ is given by

$$\hat{\theta} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2,$$

where $\mathbf{X} \in \mathbb{R}^{n \times d}$.

Solution:

$$\hat{\theta} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T Y.$$

Why?

Penalized linear regression

The estimate of linear regression $\hat{\theta}_{\lambda,q}$ is given by

$$\hat{\theta}_{\lambda,q} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_q^q.$$

Penalized regression

Penalized linear regression

The estimate of linear regression $\hat{\theta}_{\lambda,q}$ is given by

$$\hat{\theta}_{\lambda,q} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_q^q.$$

- $q = 2$: Ridge linear regression,

Penalized regression

Penalized linear regression

The estimate of linear regression $\hat{\theta}_{\lambda,q}$ is given by

$$\hat{\theta}_{\lambda,q} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_q^q.$$

- $q = 2$: Ridge linear regression,
- $q = 1$: LASSO.

Ridge regression, $q = 2$

Ridge linear regression

The ridge estimate $\hat{\theta}_{\lambda,2}$ is given by

$$\hat{\theta}_{\lambda,2} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_2^2.$$

Ridge regression, $q = 2$

Ridge linear regression

The ridge estimate $\hat{\theta}_{\lambda,2}$ is given by

$$\hat{\theta}_{\lambda,2} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_2^2.$$

Solution:

$$\hat{\theta}_{\lambda,2} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T Y.$$

Ridge regression, $q = 2$

Ridge linear regression

The ridge estimate $\hat{\theta}_{\lambda,2}$ is given by

$$\hat{\theta}_{\lambda,2} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_2^2.$$

Solution:

$$\hat{\theta}_{\lambda,2} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T Y.$$

Assume $Y = X\theta + \varepsilon$ with $\mathbb{E}[\varepsilon] = 0$ and $\mathbb{E}[\varepsilon\varepsilon^T] = \sigma^2 I$:

- Ridge estimate has bias $\mathbb{E}[\hat{\theta}_{\lambda,2}] - \theta = -\lambda(\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\theta$
- and variance $\mathbb{V}[\hat{\theta}_{\lambda,2}] = \sigma^2(\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T\mathbf{X}(\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}$.

Ridge regression, $q = 2$

Ridge linear regression

The ridge estimate $\hat{\theta}_{\lambda,2}$ is given by

$$\hat{\theta}_{\lambda,2} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_2^2.$$

Solution:

$$\hat{\theta}_{\lambda,2} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T Y.$$

Assume $Y = X\theta + \varepsilon$ with $\mathbb{E}[\varepsilon] = 0$ and $\mathbb{E}[\varepsilon\varepsilon^T] = \sigma^2 I$:

- Ridge estimate has bias $\mathbb{E}[\hat{\theta}_{\lambda,2}] - \theta = -\lambda(\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\theta$
- and variance $\mathbb{V}[\hat{\theta}_{\lambda,2}] = \sigma^2(\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T\mathbf{X}(\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}$.

Note that $\mathbb{V}[\hat{\theta}_{\lambda,2}] \preccurlyeq \mathbb{V}[\hat{\theta}_{0,2}]$ (variance is reduced).

Ridge regression, $q = 2$

Ridge linear regression

The ridge estimate $\hat{\theta}_{\lambda,2}$ is given by

$$\hat{\theta}_{\lambda,2} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_2^2.$$

Solution:

$$\hat{\theta}_{\lambda,2} = (\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T Y.$$

Assume $Y = X\theta + \varepsilon$ with $\mathbb{E}[\varepsilon] = 0$ and $\mathbb{E}[\varepsilon\varepsilon^T] = \sigma^2 I$:

- Ridge estimate has bias $\mathbb{E}[\hat{\theta}_{\lambda,2}] - \theta = -\lambda(\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\theta$
- and variance $\mathbb{V}[\hat{\theta}_{\lambda,2}] = \sigma^2(\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}\mathbf{X}^T\mathbf{X}(\mathbf{X}^T\mathbf{X} + \lambda I)^{-1}$.

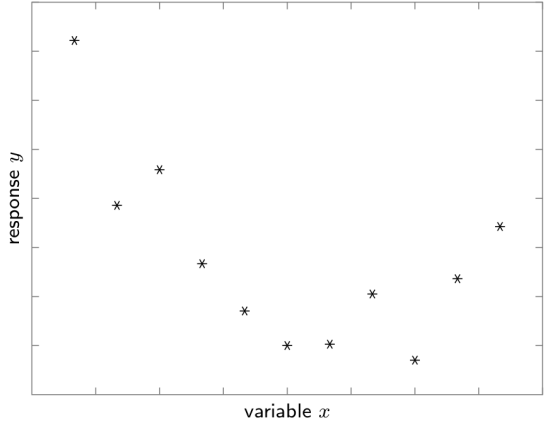
Note that $\mathbb{V}[\hat{\theta}_{\lambda,2}] \preccurlyeq \mathbb{V}[\hat{\theta}_{0,2}]$ (variance is reduced).

Particular case: for orthonormal design ($\mathbf{X}^T\mathbf{X} = I$), we reach

$$\hat{\theta}_{\lambda,2} = \frac{\hat{\theta}_{0,2}}{1+\lambda} = \frac{1}{1+\lambda}\mathbf{X}^T Y.$$

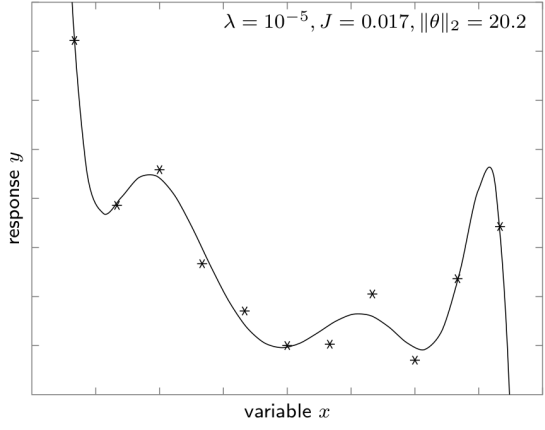
Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



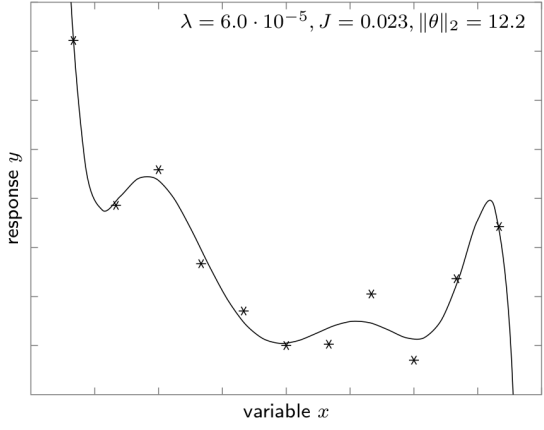
Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



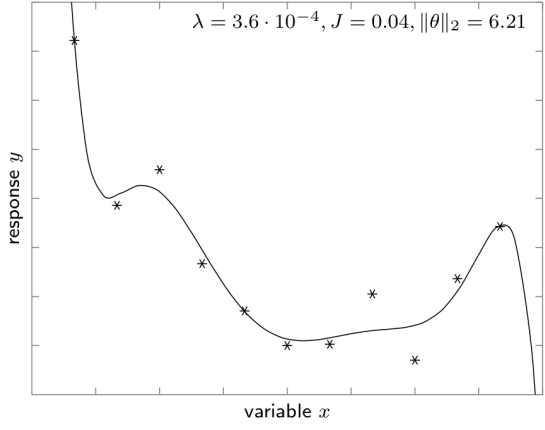
Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



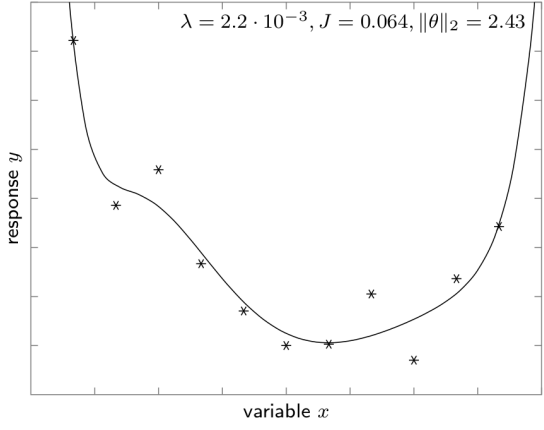
Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



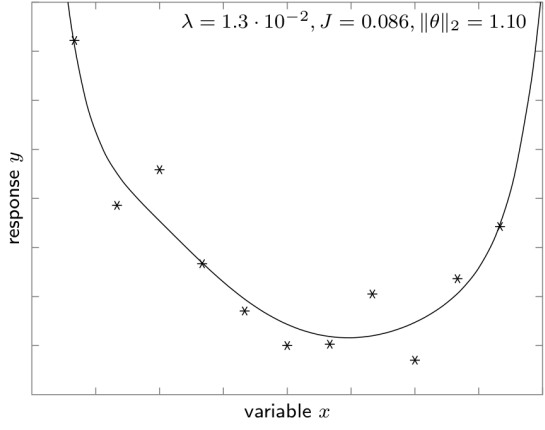
Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



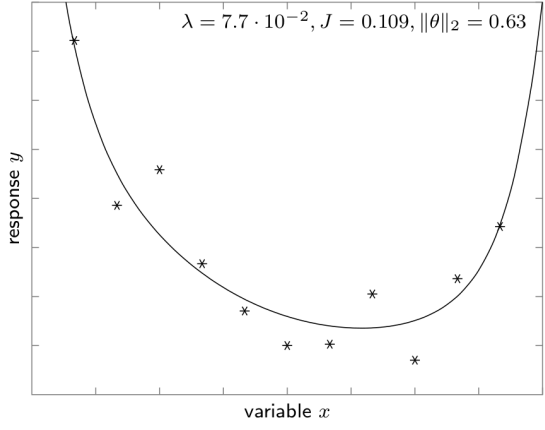
Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



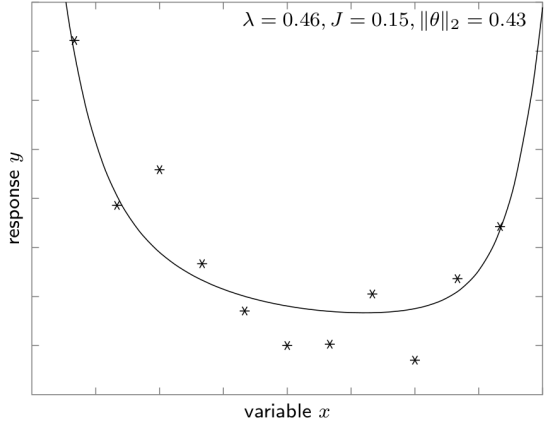
Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



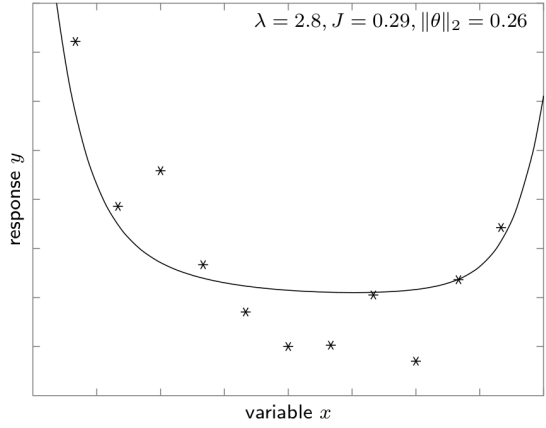
Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



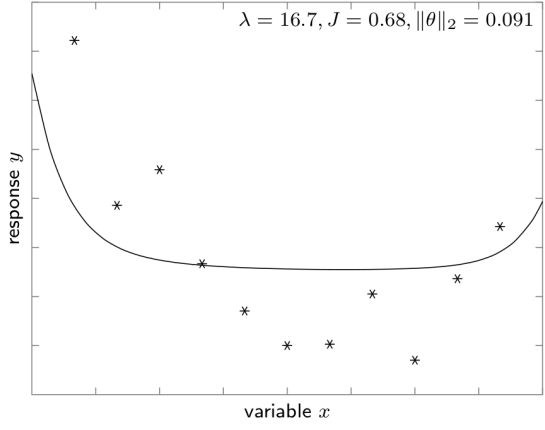
Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



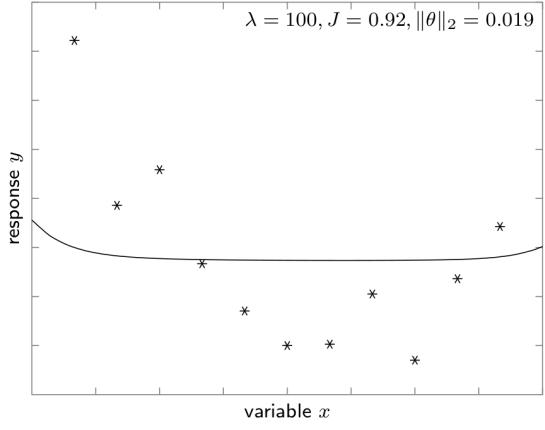
Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



Ridge regression

- 10th degree polynomial regression problem (J is cost),
- quadratic penalization $\lambda\|\theta\|_2^2$.



Sparsity

Sparsity

Other desirable properties for $\hat{\theta}$?

Sparsity

Other desirable properties for $\hat{\theta}$?

If $\hat{\theta}_j = 0$, then feature j has no impact on the prediction: $\hat{y} = \text{sign}(\langle \hat{\theta}, x \rangle + \hat{b})$.

Sparsity

Other desirable properties for $\hat{\theta}$?

If $\hat{\theta}_j = 0$, then feature j has no impact on the prediction: $\hat{y} = \text{sign}(\langle \hat{\theta}, x \rangle + \hat{b})$.

Nice if $\hat{\theta}$ contains **many zeros**, particularly if d is large (many features).

- It means that only **few** features are statistically relevant,
- It means that only **few** features are used to predict labels.

$\rightarrow$ model is more robust,

$\rightarrow$ model is cheaper/faster to compute (at prediction/test time).

Sparsity

Other desirable properties for $\hat{\theta}$?

If $\hat{\theta}_j = 0$, then feature j has no impact on the prediction: $\hat{y} = \text{sign}(\langle \hat{\theta}, x \rangle + \hat{b})$.

Nice if $\hat{\theta}$ contains **many zeros**, particularly if d is large (many features).

- It means that only **few** features are statistically relevant,
- It means that only **few** features are used to predict labels.

$\rightarrow$ model is more robust,

$\rightarrow$ model is cheaper/faster to compute (at prediction/test time).

Leads to a simpler model, with "reduced" dimension

Sparsity

Other desirable properties for $\hat{\theta}$?

If $\hat{\theta}_j = 0$, then feature j has no impact on the prediction: $\hat{y} = \text{sign}(\langle \hat{\theta}, x \rangle + \hat{b})$.

Nice if $\hat{\theta}$ contains **many zeros**, particularly if d is large (many features).

- It means that only **few** features are statistically relevant,
- It means that only **few** features are used to predict labels.

$\rightarrow$ model is more robust,

$\rightarrow$ model is cheaper/faster to compute (at prediction/test time).

Leads to a simpler model, with "reduced" dimension

How do we search for **sparse** θ?

Sparsity

Ideal target

$$\hat{\theta}_{\lambda,0} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_0.$$

with

$$\|\theta\|_0 = \#\{j \in \{1, \dots, d\} : \theta_j \neq 0\}.$$

Sparsity

Ideal target

$$\hat{\theta}_{\lambda,0} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_0.$$

with

$$\|\theta\|_0 = \#\{j \in \{1, \dots, d\} : \theta_j \neq 0\}.$$

Solving this requires exploring **all** possible supports of θ. Too long! (NP-hard)

Sparsity

Ideal target

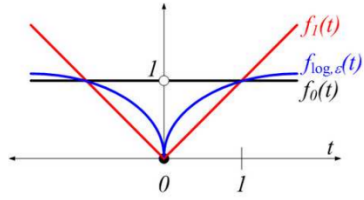
$$\hat{\theta}_{\lambda,0} \in \operatorname*{argmin}_{\theta \in \mathbb{R}^d} \|\mathbf{X}\theta - Y\|_2^2 + \lambda \|\theta\|_0.$$

with

$$\|\theta\|_0 = \#\{j \in \{1, \dots, d\} : \theta_j \neq 0\}.$$

Solving this requires exploring **all** possible supports of θ. Too long! (NP-hard)

Convex proxy for $\|\cdot\|_0$: the ℓ_1-**norm** $\|\theta\|_1 = \sum_{j=1}^d |\theta_j|$.



LASSO

LASSO linear regression

The LASSO (Least Absolute Selection and Shrinkage Operator) estimate of linear regression $\hat{\theta}_{\lambda,1}$ is given by

$$\hat{\theta}_{\lambda,1} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda\|\theta\|_1.$$

LASSO

LASSO linear regression

The LASSO (Least Absolute Selection and Shrinkage Operator) estimate of linear regression $\hat{\theta}_{\lambda,1}$ is given by

$$\hat{\theta}_{\lambda,1} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda\|\theta\|_1.$$

Solution: No closed-form in general case.

LASSO

LASSO linear regression

The LASSO (Least Absolute Selection and Shrinkage Operator) estimate of linear regression $\hat{\theta}_{\lambda,1}$ is given by

$$\hat{\theta}_{\lambda,1} \in \underset{\theta \in \mathbb{R}^d}{\operatorname{argmin}} \|\mathbf{X}\theta - Y\|_2^2 + \lambda\|\theta\|_1.$$

Solution: No closed-form in general case.

If $\mathbf{X}_{:,j}$'s are orthonormal ($\mathbf{X}^T\mathbf{X} = I$) then

$$\hat{\theta}_{\lambda,1,j} = (\mathbf{X}_{:,j})^T Y \left(1 - \frac{\lambda}{2|(\mathbf{X}_{:,j})^T Y|}\right)_+,$$

with $(x)_+ = \max(0, x)$.

In very specific case of orthogonal design, ℓ_1 penalization implies sparse vector if the parameter λ is large enough.

Sparsity - a picture

Why ℓ_2 (ridge) does not induce sparsity?

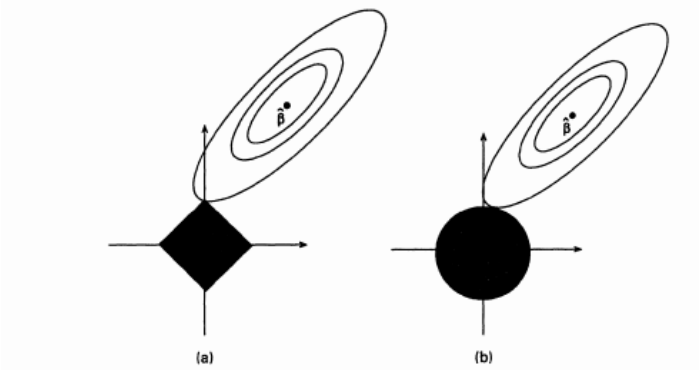


Fig. 2. Estimation picture for (a) the lasso and (b) ridge regression

Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Dropout

Dropout refers to dropping out units (hidden and visible) in a neural network:
temporary removal from network (along with all incoming and outgoing connections).

Each unit is independently dropped with probability

- $p = 0.5$ for hidden units
- $p \in [0, 0.5]$ for input units, usually $p = 0.2$.

["Improving neural networks by preventing co-adaptation of feature detectors", Hinton et al. 2012]

Dropout — at network level (train time)

Classical training loop with dropout:

Dropout — at network level (train time)

Classical training loop with dropout:

- Pick random samples

Dropout — at network level (train time)

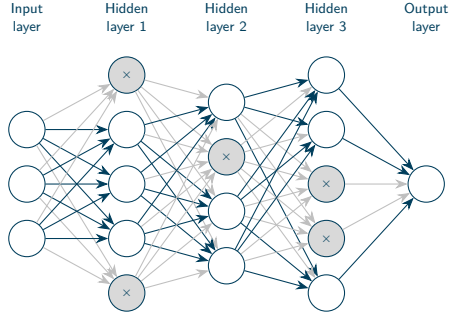
Classical training loop with dropout:

- Pick random samples
- Mask random neurons

Dropout — at network level (train time)

Classical training loop with dropout:

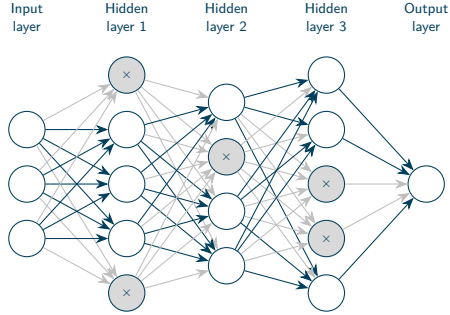
- Pick random samples
- Mask random neurons



Dropout — at network level (train time)

Classical training loop with dropout:

- Pick random samples
- Mask random neurons

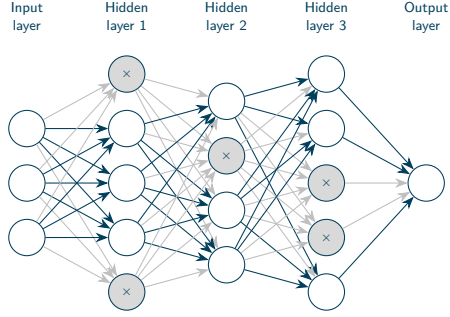


- Forward pass

Dropout — at network level (train time)

Classical training loop with dropout:

- Pick random samples
- Mask random neurons

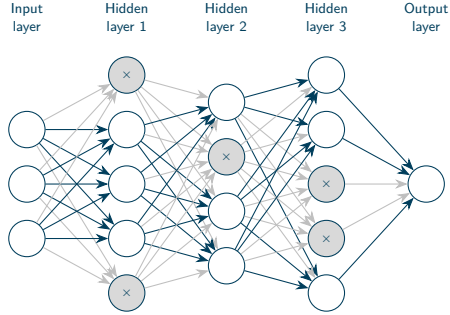


- Forward pass
- Backward pass

Dropout — at network level (train time)

Classical training loop with dropout:

- Pick random samples
- Mask random neurons

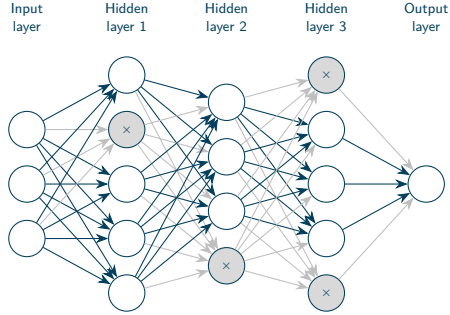


- Forward pass
- Backward pass
- Update weights (via optimization algorithm)

Dropout — at network level (train time)

Classical training loop with dropout:

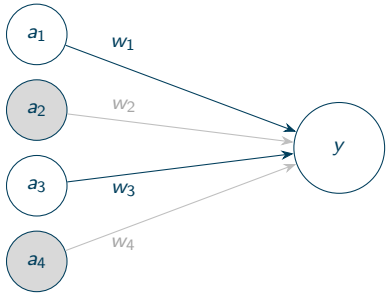
- Pick random samples
- Mask random neurons



- Forward pass
- Backward pass
- Update weights (via optimization algorithm)

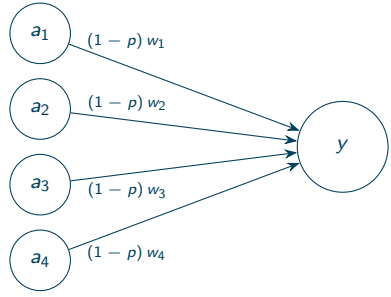
Dropout — at neuron level

Train time (dropout active)



$$y = w_1 a_1 + \cancel{w_2 a_2} + w_3 a_3 + \cancel{w_4 a_4}$$

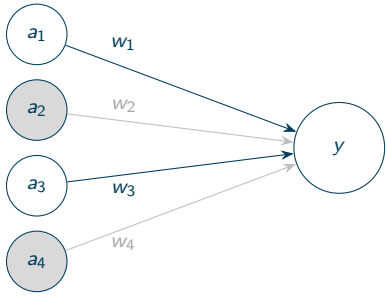
Test time (scaled weights)



$$y = (1-p)(w_1 a_1 + w_2 a_2 + w_3 a_3 + w_4 a_4)$$

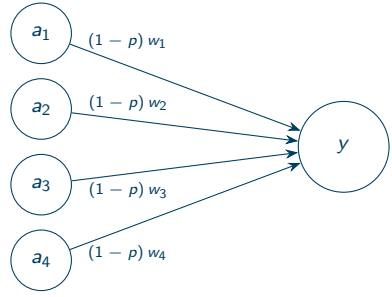
Dropout — at neuron level

Train time (dropout active)



$$y = w_1 a_1 + \cancel{w_2 a_2} + w_3 a_3 + \cancel{w_4 a_4}$$

Test time (scaled weights)



$$y = (1-p)(w_1 a_1 + w_2 a_2 + w_3 a_3 + w_4 a_4)$$

Why? Output keeps the same average magnitude.

Dropout algorithm

Training step. While *not convergence*

Dropout algorithm

Training step. While *not convergence*

1. Inside one epoch, for each mini-batch of size m,

Dropout algorithm

Training step. While *not convergence*

1. Inside one epoch, for each mini-batch of size m,
    1. Sample m different mask. A mask consists in one Bernoulli per node of the network (inner and entry nodes but not output nodes). These Bernoulli variables are *i.i.d.*. Usually
        - ⋆ the probability of selecting an hidden node is 0.5
        - ⋆ the probability of selecting an input node is 0.8

Dropout algorithm

Training step. While *not convergence*

1. Inside one epoch, for each mini-batch of size m,
   1. Sample m different mask. A mask consists in one Bernoulli per node of the network (inner and entry nodes but not output nodes). These Bernoulli variables are *i.i.d.*. Usually
      - the probability of selecting an hidden node is 0.5
      - the probability of selecting an input node is 0.8
   2. For each one of the m observation in the mini-batch,
      - Do a forward pass on the masked network
      - Compute backpropagation in the masked network
      - Compute the average gradient

Dropout algorithm

Training step. While *not convergence*

1. Inside one epoch, for each mini-batch of size m,
   1. Sample m different mask. A mask consists in one Bernoulli per node of the network (inner and entry nodes but not output nodes). These Bernoulli variables are *i.i.d.*. Usually
      - ★ the probability of selecting an hidden node is 0.5
      - ★ the probability of selecting an input node is 0.8
   2. For each one of the m observation in the mini-batch,
      - ★ Do a forward pass on the masked network
      - ★ Compute backpropagation in the masked network
      - ★ Compute the average gradient
   3. Update the parameter according to the usual formula.

Dropout algorithm

Training step. While *not convergence*

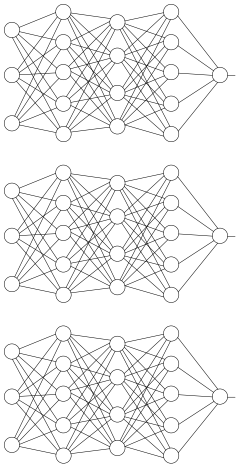
1. Inside one epoch, for each mini-batch of size m,
   1. Sample m different mask. A mask consists in one Bernoulli per node of the network (inner and entry nodes but not output nodes). These Bernoulli variables are *i.i.d.*. Usually
      - ★ the probability of selecting an hidden node is 0.5
      - ★ the probability of selecting an input node is 0.8
   2. For each one of the m observation in the mini-batch,
      - ★ Do a forward pass on the masked network
      - ★ Compute backpropagation in the masked network
      - ★ Compute the average gradient
   3. Update the parameter according to the usual formula.

Prediction step.
Use all neurons in the network with weights given by the previous optimization procedure, times probability of being selected (0.5 for inner nodes, 0.8 for input nodes).

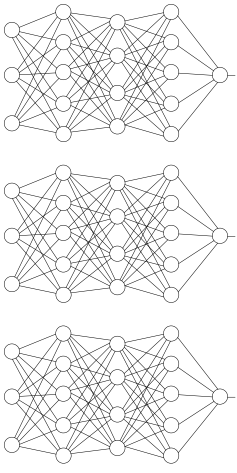
Another approach to dropout – ensemble methods

Average many different networks.

Different could mean:



Another approach to dropout – ensemble methods

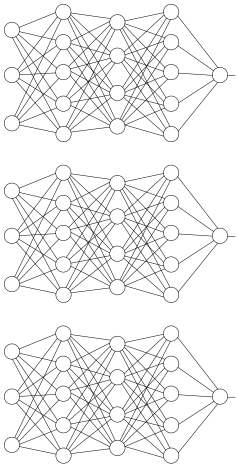


Average many different networks.

Different could mean:

- randomize data set on which we train each network (via subsampling).
  Problem: not enough data to obtain good performance...

Another approach to dropout – ensemble methods



Average many different networks.

Different could mean:

- randomize data set on which we train each network (via subsampling).
  Problem: not enough data to obtain good performance...
- build different network architectures and train each large network separately on the whole training set.
  Problem: computationally prohibitive at training time and test time!

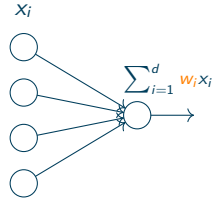
["Fast dropout training", Wang and Manning 2013]

["Dropout: A simple way to prevent neural networks from overfitting", Srivastava et al. 2014]

Drop weights instead of the whole neurons: ["Regularization of neural networks using dropconnect", Wan et al. 2013]

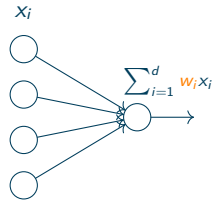
Interpretation via linear units

One-layer linear neural network with mean squared loss:



Interpretation via linear units

One-layer linear neural network with mean squared loss:



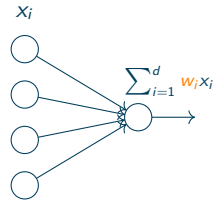
- Dropout network loss ($\delta_i \sim \mathrm{Ber}(q_i)$, q_i proba of i to **be selected**) with weights w_i

$$\mathcal{L}_{\text{dropout}} = \frac{1}{2}\left(y - \sum_{i=1}^{d} \delta_i w_i x_i\right)^2 .$$

Gradient wrt w_i is $\frac{\partial \mathcal{L}_{\text{dropout}}}{\partial w_i} = -y\delta_i x_i + w_i \delta_i^2 x_i^2 + \sum_{j=1, j\neq i}^{d} w_j \delta_i \delta_j x_i x_j$.

Interpretation via linear units

One-layer linear neural network with mean squared loss:



- Dropout network loss ($\delta_i \sim \text{Ber}(q_i)$, q_i proba of i to **be selected**) with weights w_i

$$\mathcal{L}_{\text{dropout}} = \frac{1}{2}\left(y - \sum_{i=1}^{d} \delta_i w_i x_i\right)^2.$$

  Gradient wrt w_i is $\frac{\partial \mathcal{L}_{\text{dropout}}}{\partial w_i} = -y\delta_i x_i + w_i \delta_i^2 x_i^2 + \sum_{j=1, j\neq i}^{d} w_j \delta_i \delta_j x_i x_j$.

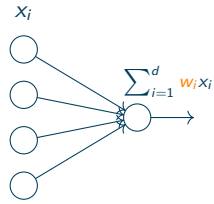
- Regular network loss with weights $w_i' = w_i q_i$

$$\mathcal{L}_{\text{regular}} = \frac{1}{2}\left(y - \sum_{i=1}^{d} w_i q_i x_i\right)^2.$$

  Gradient wrt w_i is $\frac{\partial \mathcal{L}_{\text{regular}}}{\partial w_i} = -yq_i x_i + w_i q_i^2 x_i^2 + \sum_{j=1, j\neq i}^{d} w_j q_i q_j x_i x_j$.

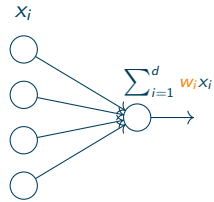
Interpretation via linear units

One-layer linear neural network with mean squared loss:



Interpretation via linear units

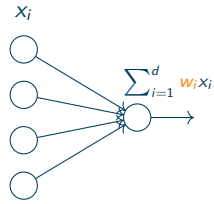
One-layer linear neural network with mean squared loss:



- Dropout net.: $\mathbb{E}\left[\frac{\partial \mathcal{L}_{\text{dropout}}}{\partial w_i}\right] = -yq_i x_i + w_i q_i^2 x_i^2 + q_i(1-q_i) w_i x_i^2 + \sum_{j=1, j\neq i}^{d} w_j q_i q_j x_i x_j.$

Interpretation via linear units

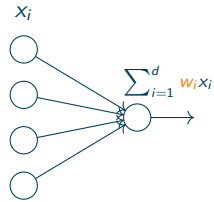
One-layer linear neural network with mean squared loss:



- Dropout net.: $\mathbb{E}\left[\frac{\partial \mathcal{L}_{\text{dropout}}}{\partial w_i}\right] = -y q_i x_i + w_i q_i^2 x_i^2 + q_i(1-q_i) w_i x_i^2 + \sum_{j=1, j\neq i}^{d} w_j q_i q_j x_i x_j$.
- Regular net.: $\frac{\partial \mathcal{L}_{\text{regular}}}{\partial w_i} = -y q_i x_i + w_i q_i^2 x_i^2 + \sum_{j=1, j\neq i}^{d} w_j q_i q_j x_i x_j$.

Interpretation via linear units

One-layer linear neural network with mean squared loss:

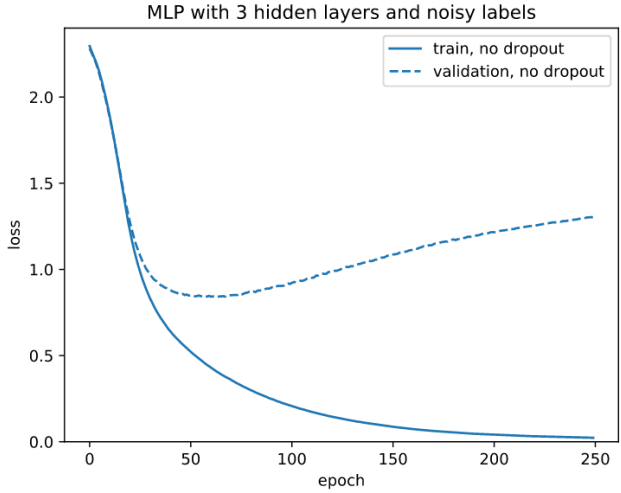


- Dropout net.: $\mathbb{E}\left[\frac{\partial \mathcal{L}_{\text{dropout}}}{\partial w_i}\right] = -yq_i x_i + w_i q_i^2 x_i^2 + q_i(1-q_i) w_i x_i^2 + \sum_{j=1, j\neq i}^{d} w_j q_i q_j x_i x_j$.
- Regular net.: $\frac{\partial \mathcal{L}_{\text{regular}}}{\partial w_i} = -yq_i x_i + w_i q_i^2 x_i^2 + \sum_{j=1, j\neq i}^{d} w_j q_i q_j x_i x_j$.

So: expectated gradient for dropout net. is gradient for regular network plus regularization

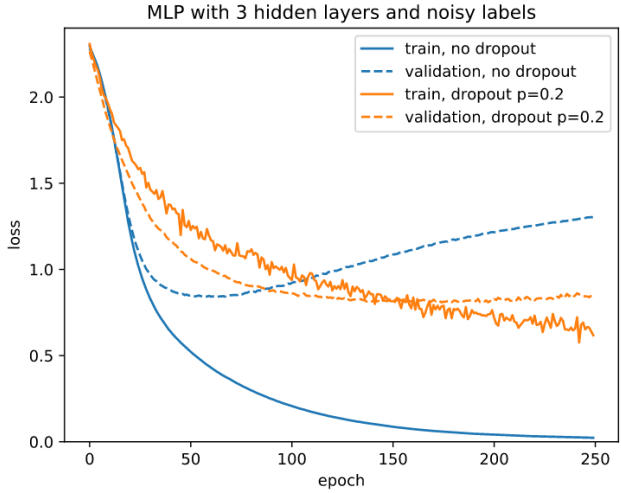
$$\mathcal{L}_{\text{regular+dropout regularization}} = \frac{1}{2}\left(y - \sum_{i=1}^{d} w_i q_i x_i\right)^2 + \frac{1}{2}\sum_{i=1}^{d} q_i(1-q_i) x_i^2 {w_i}^2.$$

Dropout as regularization



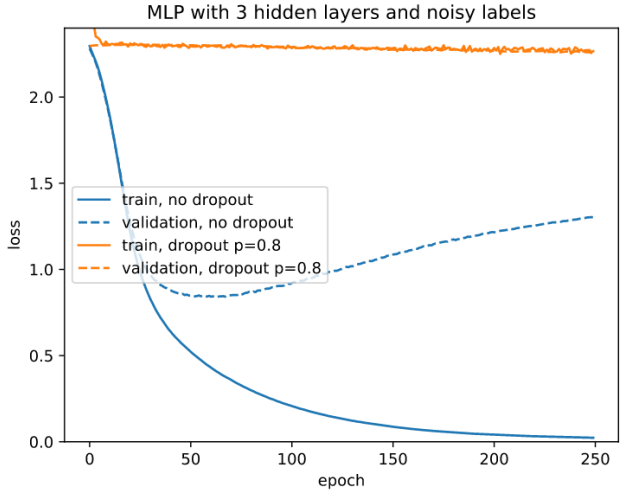
Source: `https://dataflowr.github.io/website/`.

Dropout as regularization



Source: `https://dataflowr.github.io/website/`.

Dropout as regularization



Source: `https://dataflowr.github.io/website/`.

Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Batch normalization

The network converges faster if its input are scaled (mean, variance) and decorrelated.

["Efficient backprop", LeCun et al. 1998]

Hard to decorrelate variables: requiring to compute covariance matrix.

["Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015]

Batch normalization

The network converges faster if its input are scaled (mean, variance) and decorrelated.

["Efficient backprop", LeCun et al. 1998]

Hard to decorrelate variables: requiring to compute covariance matrix.

["Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015]

Ideas:

- allows higher learning rates,
- reduces strong dependence on initialization,
- relates to regularization (slightly reduces the need for dropout).

Algorithm

See ["Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015]

1. For every neuron $k = 1, \ldots, H$ in the layer, which outputs $a_i^{(k)}$ for the ith observation ($i = 1, \ldots, m$) of batch B,
   1. $\mu_B^{(k)} = \frac{1}{m} \sum_{i=1}^{m} a_i^{(k)}$
   2. $\sigma_{B,k}^2 = \frac{1}{m} \sum_{i=1}^{m} (a_i^{(k)} - \mu_B^{(k)})^2$
   3. $\hat{a}_i^{(k)} = \frac{a_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_{B,k}^2 + \varepsilon}}$
   4. $z_i^{(k)} = \gamma^{(k)} \hat{a}_i^{(k)} + \beta^{(k)} \equiv BN_{\gamma^{(k)}, \beta^{(k)}}(a_i^{(k)})$

Algorithm

See ["Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015]

1. For every neuron $k = 1, \dots, H$ in the layer, which outputs $a_i^{(k)}$ for the ith observation ($i = 1, \dots, m$) of batch B,
   1. $\mu_B^{(k)} = \frac{1}{m} \sum_{i=1}^{m} a_i^{(k)}$
   2. $\sigma_{B,k}^2 = \frac{1}{m} \sum_{i=1}^{m} (a_i^{(k)} - \mu_B^{(k)})^2$
   3. $\hat{a}_i^{(k)} = \frac{a_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_{B,k}^2 + \varepsilon}}$
   4. $z_i^{(k)} = \gamma^{(k)} \hat{a}_i^{(k)} + \beta^{(k)} \equiv BN_{\gamma^{(k)}, \beta^{(k)}}(a_i^{(k)})$
2. $z_i^{(k)}$ is fed to the next layer and the procedure iterates.

Algorithm

See ["Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015]

1. For every neuron $k = 1, \dots, H$ in the layer, which outputs $a_i^{(k)}$ for the ith observation ($i = 1, \dots, m$) of batch B,
   1. $\mu_B^{(k)} = \frac{1}{m} \sum_{i=1}^{m} a_i^{(k)}$
   2. $\sigma_{B,k}^2 = \frac{1}{m} \sum_{i=1}^{m} (a_i^{(k)} - \mu_B^{(k)})^2$
   3. $\hat{a}_i^{(k)} = \frac{a_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_{B,k}^2 + \varepsilon}}$
   4. $z_i^{(k)} = \gamma^{(k)} \hat{a}_i^{(k)} + \beta^{(k)} \equiv BN_{\gamma^{(k)}, \beta^{(k)}}(a_i^{(k)})$
2. $z_i^{(k)}$ is fed to the next layer and the procedure iterates.
3. Backpropagation also include $(\gamma^{(k)}, \beta^{(k)})$ (init. $\gamma^{(k)} = 1, \beta^{(k)} = 0$)).

Algorithm

See ["Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015]

1. For every neuron $k = 1, \dots, H$ in the layer, which outputs $a_i^{(k)}$ for the ith observation ($i = 1, \dots, m$) of batch B,
   1. $\mu_B^{(k)} = \frac{1}{m} \sum_{i=1}^{m} a_i^{(k)}$
   2. $\sigma_{B,k}^2 = \frac{1}{m} \sum_{i=1}^{m} (a_i^{(k)} - \mu_B^{(k)})^2$
   3. $\hat{a}_i^{(k)} = \frac{a_i^{(k)} - \mu_B^{(k)}}{\sqrt{\sigma_{B,k}^2 + \varepsilon}}$
   4. $z_i^{(k)} = \gamma^{(k)} \hat{a}_i^{(k)} + \beta^{(k)} \equiv BN_{\gamma^{(k)}, \beta^{(k)}}(a_i^{(k)})$
2. $z_i^{(k)}$ is fed to the next layer and the procedure iterates.
3. Backpropagation also include $(\gamma^{(k)}, \beta^{(k)})$ (init. $\gamma^{(k)} = 1, \beta^{(k)} = 0$)).
4. For inference, compute the average over many training batches $\mathcal{B}$ of size m:
$$\mathbb{E}_{\mathcal{B}}[a^{(k)}] = \mathbb{E}_{\mathcal{B}}[\mu_{\mathcal{B}}^{(k)}] \quad \text{and} \quad \mathbb{V}_{\mathcal{B}}[a^{(k)}] = \frac{m}{m-1} \mathbb{E}_{\mathcal{B}}[\sigma_{B,k}^2].$$
5. For inference, replace every function $a^{(k)} \mapsto BN_{\gamma^{(k)}, \beta^{(k)}}(a^{(k)})$ in the network by
$$a^{(k)} \mapsto \gamma^{(k)} \left(\frac{a^{(k)} - \mathbb{E}_{\mathcal{B}}[a^{(k)}]}{\sqrt{\mathbb{V}_{\mathcal{B}}[a^{(k)}] + \varepsilon}} \right) + \beta^{(k)}.$$

Batchnorm

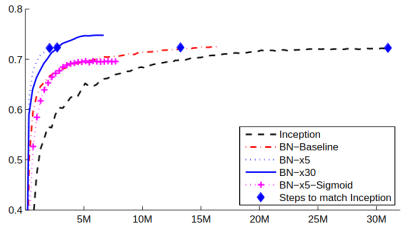


Figure 2: *Single crop validation accuracy of Inception and its batch-normalized variants, vs. the number of training steps.*

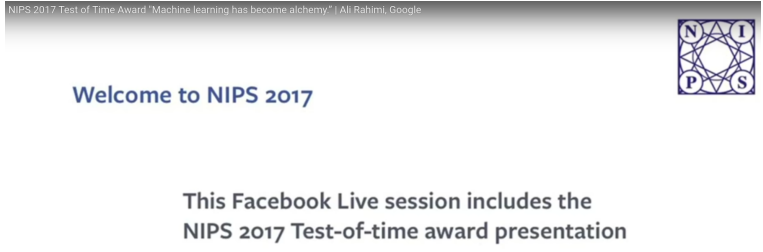
Model	Steps to 72.2%	Max accuracy
Inception	$31.0 \cdot 10^6$	72.2%
BN-Baseline	$13.3 \cdot 10^6$	72.7%
BN-x5	$2.1 \cdot 10^6$	73.0%
BN-x30	$2.7 \cdot 10^6$	74.8%
BN-x5-Sigmoid		69.8%

Figure 3: *For Inception and the batch-normalized variants, the number of training steps required to reach the maximum accuracy of Inception (72.2%), and the maximum accuracy achieved by the network.*

Source: ["Batch normalization: Accelerating deep network training by reducing internal covariate shift", Ioffe and Szegedy 2015].

- larger learning rates,
- reduces necessity for other regularization techniques (ℓ_2, dropout, ...).
- Nowadays: layer-norm ["Layer normalization", Ba et al. 2016].

NIPS 2017 Test of Time Award
"Machine learning has become alchemy."
by Ali Rahimi.



Layer normalization

LayerNorm

CLASS torch.nn.LayerNorm(*normalized_shape*, *eps=1e-05*, *elementwise_affine=True*, *bias=True*, *device=None*, *dtype=None*) [SOURCE]

Applies Layer Normalization over a mini-batch of inputs.

This layer implements the operation as described in the paper Layer Normalization

$$y = \frac{x - \mathrm{E}[x]}{\sqrt{\mathrm{Var}[x] + \epsilon}} * \gamma + \beta$$

The mean and standard-deviation are calculated over the last *D* dimensions, where *D* is the dimension of `normalized_shape`. For example, if `normalized_shape` is `(3, 5)` (a 2-dimensional shape), the mean and standard-deviation are computed over the last 2 dimensions of the input (i.e. `input.mean((-2, -1))`). γ and β are learnable affine transform parameters of `normalized_shape` if `elementwise_affine` is `True`. The variance is calculated via the biased estimator, equivalent to *torch.var(input, unbiased=False)*.

• NOTE

Unlike Batch Normalization and Instance Normalization, which applies scalar scale and bias for each entire channel/plane with the `affine` option, Layer Normalization applies per-element scale and bias with `elementwise_affine`.

In short: similar to batchnorm, but no normalization accross batch:

$$\mu^{(k)} = \frac{1}{H}\sum_{i=1}^{H} a_i^{(k)}, \quad \sigma^{(k)} = \sqrt{\frac{1}{H}\sum_{i=1}^{H}(a_i^{(k)} - \mu^{(k)})^2}$$

for layer k (which has $i = 1, \ldots, H$ outputs).

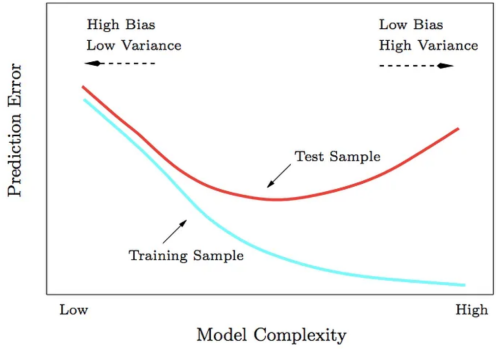
Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

Early stopping

Idea:

- Store the parameter values that lead to the **lowest error on the validation set**
- **Return these values** rather than the latest ones.



Source: https://towardsdatascience.com

Early stopping: aim for the minimum of the validation/test curve.

Early stopping algorithm

Parameters:

- patience p of the algorithm: number of times to observe no improvement on the validation set error before giving up;
- the number of steps n between evaluations.

Early stopping algorithm

Parameters:

- patience p of the algorithm: number of times to observe no improvement on the validation set error before giving up;
- the number of steps n between evaluations.

1. Start with initial random values θ_0.
2. Let $\theta_\star = \theta_0$, $\mathrm{Err}_\star = \infty$, $j = 0$, $i = 0$.
3. While $j < p$
   1. Update θ by running the training algorithm for n steps
   2. $i = i + n$
   3. Compute the error $\mathrm{Err}(\theta)$ on the validation set
   4. If $\mathrm{Err}(\theta) < \mathrm{Err}_\star$
      - $\theta_\star = \theta$
      - $\mathrm{Err}_\star = \mathrm{Err}(\theta)$
      - $j = 0$

      else $j = j + 1$.
4. Return $\theta_\star$ and the overall number of steps $i_\star = i - np$.

How to leverage on early stopping?

First idea: use early stopping to **determine the best number of iterations $i_\star$** and **train on the whole data set for $i_\star$ iterations**.

Let $X^{(train)}, y^{(train)}$ be the training set.

- Split $X^{(train)}, y^{(train)}$ into $X^{(subtrain)}, y^{(subtrain)}$ and $X^{(valid)}, y^{(valid)}$.
- Run early stopping algorithm starting from random θ using $X^{(subtrain)}, y^{(subtrain)}$ for training data and $X^{(valid)}, y^{(valid)}$ for validation data. This returns $i_\star$ the optimal number of steps.
- Set θ to random values again.
- Train on $X^{(train)}, y^{(train)}$ for $i_\star$ steps.

How to leverage on early stopping?

Second idea: use early stopping on subset of training dataset to determine

- best parameters and
- the **training error at the best number of iterations**.

Starts $\theta^\star$, **train on full dataset** until training error matches previous early stopping error.

How to leverage on early stopping?

Second idea: use early stopping on subset of training dataset to determine

- best parameters and
- the **training error at the best number of iterations**.

Starts $\theta^\star$, **train on full dataset** until training error matches previous early stopping error.

Let $X^{(train)}, y^{(train)}$ be the training set.

- Split $X^{(train)}, y^{(train)}$ into $X^{(subtrain)}, y^{(subtrain)}$ and $X^{(valid)}, y^{(valid)}$.
- Run early stopping algorithm starting from random θ using $X^{(subtrain)}, y^{(subtrain)}$ for training data and $X^{(valid)}, y^{(valid)}$ for validation data. This returns the optimal parameters $\theta^\star$.
- Set $\varepsilon = \mathcal{L}(\theta^\star, X^{(subtrain)}, y^{(subtrain)})$.
- While $\mathcal{L}(\theta, X^{(valid)}, y^{(valid)}) > \varepsilon$, continue training on $X^{(train)}, y^{(train)}$.

To go further

- Early stopping is a very old idea
  - ["Three topics in ill-posed problems", Wahba 1987]
  - ["A formal comparison of methods proposed for the numerical solution of first kind integral equations", Anderssen and Prenter 1981]
  - ["Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping", Caruana et al. 2001]
- But also an active area of research
  - ["Adaboost is consistent", Bartlett and Traskin 2007]
  - ["Boosting algorithms as gradient descent", Mason et al. 2000]
  - ["On early stopping in gradient descent learning", Yao et al. 2007]
  - ["Boosting with early stopping: Convergence and consistency", Zhang, Yu, et al. 2005]
  - ["Early stopping for kernel boosting algorithms: A general analysis with localized complexities", Wei et al. 2017]

More on reducing overfitting

- Soft-weight sharing:
  ["Simplifying neural networks by soft weight-sharing", Nowlan and Hinton 1992]
- Model averaging:
  Average over: random initialization, random selection of minibatches, hyperparameters, or outcomes of nondeterministic neural networks.
- Boosting neural networks by incrementally adding neural networks to the ensemble
  ["Training methods for adaptive boosting of neural networks", Schwenk and Bengio 1998]
- Boosting has also been applied interpreting an individual neural network as an ensemble, incrementally adding hidden units to the networks
  ["Convex neural networks", Bengio et al. 2006]

Outline

1. Learning & neural network basics
   - Features
   - Backpropagation
2. Hyperparameters
   - How to choose the number of hidden layers/neurons?
   - Activation functions
   - Output units
   - Loss functions
   - Weight initialization
3. Generalization & regularization
   - Validation
   - Penalization
   - Dropout
   - Batch normalization
   - Early stopping
4. All in all

What did we see so far?

https://app.wooclap.com/YUKENL?from=instruction-slidelink to wooclap

Pipeline for neural networks

Pipeline for neural networks

- Step 1: preprocess data (substract mean, divide by standard deviation). More complex if data are images.

Pipeline for neural networks

- Step 1: preprocess data (substract mean, divide by standard deviation). More complex if data are images.
- Step 2: choose architecture (number of layers, number of nodes per layer).

Pipeline for neural networks

- Step 1: preprocess data (substract mean, divide by standard deviation). More complex if data are images.
- Step 2: choose architecture (number of layers, number of nodes per layer).
- Step 3:
  1. First, run network and see if loss is reasonable (compare with dumb classifier: uniform for classification, mean for regression)
  2. Add some regularization and check that the error on the training set increases.
  3. On small portion of data, make sure you can overfit when turning down the regularization.
  4. Find best learning rate
     1. Error does not change too much $\rightarrow$ learning rate too small.
     2. Error explodes, NaN $\rightarrow$ learning rate too large.
     3. Find a rough range, e.g., $[10^{-5}, 10^{-3}]$.

Playing with neural network:

http://playground.tensorflow.org/

To go further

Kernel methods, kernel perceptron was already introduced by

[“Theoretical foundations of the potential function method in pattern recognition learning”, Aizerman 1964].

To go further

Kernel methods, kernel perceptron was already introduced by

["Theoretical foundations of the potential function method in pattern recognition learning", Aizerman 1964].

General idea (work for all methods using only dot product): replace dot product by more complex kernel function.

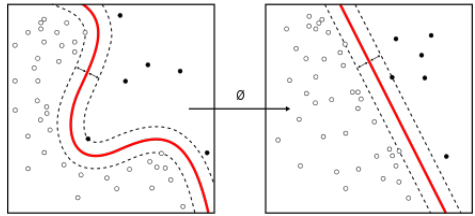
To go further

Kernel methods, kernel perceptron was already introduced by

["Theoretical foundations of the potential function method in pattern recognition learning", Aizerman 1964].

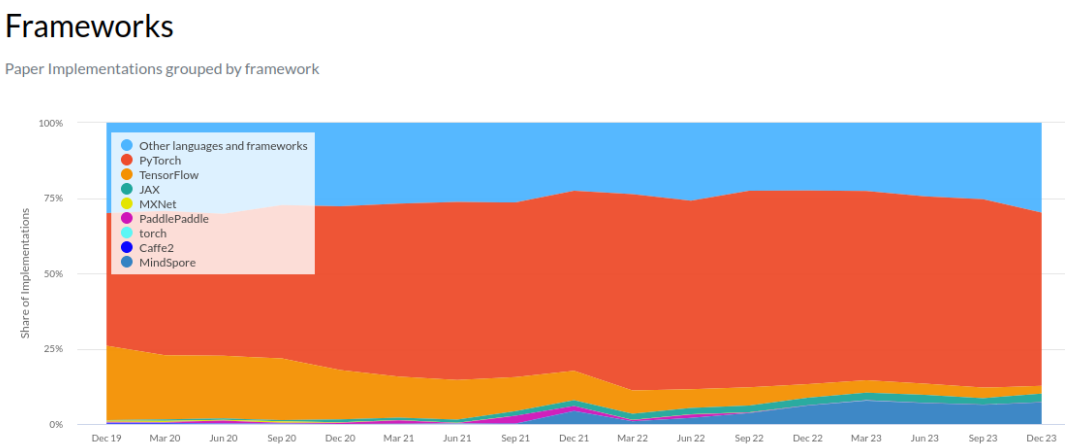
General idea (work for all methods using only dot product): replace dot product by more complex kernel function.

Linear separation in original space becomes linear separation in a more complex space, i.e., non linear separation in original space.



Software

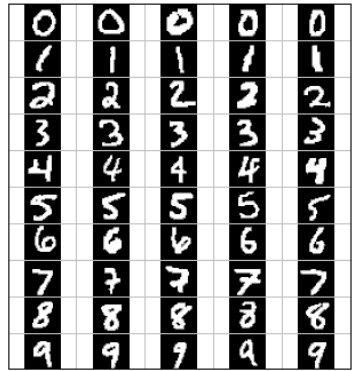
Today's lab: Pytorch!



Source: `https://paperswithcode.com/trends`.

This afternoon

- play & design base neural nets and objects (backprop, optimizers).
- Get familiar with software $\rightarrow$ mostly PyTorch.
- Play with MNIST



- Data: annotated database of images (each image is represented by a vector of $28 \times 28 = 784$ pixel intensities),
- input: image,
- output: corresponding label (a number in 0–9).

[Aiz64] Mark A Aizerman. "Theoretical foundations of the potential function method in pattern recognition learning". In: *Automation and remote control* 25 (1964), pp. 821–837.

[AP81] RS Anderssen and PM Prenter. "A formal comparison of methods proposed for the numerical solution of first kind integral equations". In: *The ANZIAM Journal* 22.4 (1981), pp. 488–500.

[Ben+06] Yoshua Bengio et al. "Convex neural networks". In: *Advances in neural information processing systems*. 2006, pp. 123–130.

[BKH16] Jimmy Lei Ba, Jamie Ryan Kiros, and Geoffrey E Hinton. "Layer normalization". In: *arXiv preprint arXiv:1607.06450* (2016).

[Bla+20] Davis Blalock et al. "What is the state of neural network pruning?" In: *arXiv preprint arXiv:2003.03033* (2020).

[BP21] Jérôme Bolte and Edouard Pauwels. "Conservative set valued fields, automatic differentiation, stochastic gradient methods and deep learning". In: *Mathematical Programming* 188 (2021), pp. 19–51.

[Bri90] John S Bridle. "Probabilistic interpretation of feedforward classification network outputs, with relationships to statistical pattern recognition". In: *Neurocomputing*. Springer, 1990, pp. 227–236.

[BT07] Peter L Bartlett and Mikhail Traskin. "Adaboost is consistent". In: *Journal of Machine Learning Research* 8.Oct (2007), pp. 2347–2368.

[CLG01] Rich Caruana, Steve Lawrence, and C Lee Giles. "Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping". In: *Advances in neural information processing systems*. 2001, pp. 402–408.

[CSL13] Meng Cai, Yongzhe Shi, and Jia Liu. "Deep maxout neural networks for speech recognition". In: *Automatic Speech Recognition and Understanding (ASRU), 2013 IEEE Workshop on*. IEEE. 2013, pp. 291–296.

[CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. "Fast and accurate deep network learning by exponential linear units (elus)". In: *arXiv preprint arXiv:1511.07289* (2015).

[Ege+20] Romain Egele et al. "AgEBO-Tabular: Joint Neural Architecture and Hyperparameter Search with Autotuned Data-Parallel Training for Tabular Data". In: *arXiv preprint arXiv:2010.16358* (2020).

[FM82] Kunihiko Fukushima and Sei Miyake. "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition". In: *Competition and cooperation in neural nets*. Springer, 1982, pp. 267–285.

[Fuk75] Kunihiko Fukushima. "Cognitron: A self-organizing multilayered neural network". In: *Biological cybernetics* 20.3-4 (1975), pp. 121–136.

[GB10] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, pp. 249–256.

[GBB11] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep sparse rectifier neural networks". In: *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*. 2011, pp. 315–323.

[Goo+13a] Ian J Goodfellow, Mehdi Mirza, et al. "An empirical investigation of catastrophic forgetting in gradient-based neural networks". In: *arXiv preprint arXiv:1312.6211* (2013).

[Goo+13b] Ian J Goodfellow, David Warde-Farley, et al. "Maxout networks". In: *arXiv preprint arXiv:1302.4389* (2013).

[He+15] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[Hin+12] Geoffrey E Hinton et al. "Improving neural networks by preventing co-adaptation of feature detectors". In: *arXiv preprint arXiv:1207.0580* (2012).

[HK70] Arthur E Hoerl and Robert W Kennard. "Ridge regression: Biased estimation for nonorthogonal problems". In: *Technometrics* 12.1 (1970), pp. 55–67.

[IS15] Sergey Ioffe and Christian Szegedy. "Batch normalization: Accelerating deep network training by reducing internal covariate shift". In: *arXiv preprint arXiv:1502.03167* (2015).

[JKL+09] Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. "What is the best multi-stage architecture for object recognition?" In: *Computer Vision, 2009 IEEE 12th International Conference on*. IEEE. 2009, pp. 2146–2153.

[LeC+98] Yann LeCun et al. "Efficient backprop". In: *Neural networks: Tricks of the trade*. Springer, 1998, pp. 9–50.

[Mas+00] Llew Mason et al. "Boosting algorithms as gradient descent". In: *Advances in neural information processing systems*. 2000, pp. 512–518.

[MHN13] Andrew L Maas, Awni Y Hannun, and Andrew Y Ng. "Rectifier nonlinearities improve neural network acoustic models". In: *Proc. icml*. Vol. 30. 1. 2013, p. 3.

[MM15] Dmytro Mishkin and Jiri Matas. "All you need is a good init". In: *arXiv preprint arXiv:1511.06422* (2015).

[NH92] Steven J Nowlan and Geoffrey E Hinton. "Simplifying neural networks by soft weight-sharing". In: *Neural computation* 4.4 (1992), pp. 473–493.

[RHW86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. "Learning representations by back-propagating errors". In: *nature* 323.6088 (1986), pp. 533–536.

[RZL17] Prajit Ramachandran, Barret Zoph, and Quoc V Le. "Searching for activation functions". In: *arXiv preprint arXiv:1710.05941* (2017).

[SB98] Holger Schwenk and Yoshua Bengio. "Training methods for adaptive boosting of neural networks". In: *Advances in neural information processing systems*. 1998, pp. 647–653.

[Sha+16] Wenling Shang et al. "Understanding and improving convolutional neural networks via concatenated rectified linear units". In: *International conference on machine learning*. 2016.

[Sri+14] Nitish Srivastava et al. "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[Tib96] Robert Tibshirani. "Regression shrinkage and selection via the lasso". In: *Journal of the Royal Statistical Society. Series B (Methodological)* (1996), pp. 267–288.

[Wah87] Grace Wahba. "Three topics in ill-posed problems". In: *Inverse and ill-posed problems*. Elsevier, 1987, pp. 37–51.

[Wan+13] Li Wan et al. "Regularization of neural networks using dropconnect". In: *International conference on machine learning*. 2013, pp. 1058–1066.

[Wer74] Paul Werbos. "Beyond regression: New tools for prediction and analysis in the behavioral sciences". PhD thesis. Harvard University, 1974.

[Wie15] Wessel N van Wieringen. "Lecture notes on ridge regression". In: *arXiv preprint arXiv:1509.09169* (2015).

[WM13] Sida Wang and Christopher Manning. “Fast dropout training”. In: *International conference on machine learning*. 2013.

[WYW17] Yuting Wei, Fanny Yang, and Martin J Wainwright. “Early stopping for kernel boosting algorithms: A general analysis with localized complexities”. In: *Advances in Neural Information Processing Systems*. 2017, pp. 6067–6077.

[Xu+15] Bing Xu et al. “Empirical evaluation of rectified activations in convolutional network”. In: *arXiv preprint arXiv:1505.00853* (2015).

[YRC07] Yuan Yao, Lorenzo Rosasco, and Andrea Caponnetto. “On early stopping in gradient descent learning”. In: *Constructive Approximation* 26.2 (2007), pp. 289–315.

[ZH05] Hui Zou and Trevor Hastie. “Regularization and variable selection via the elastic net”. In: *Journal of the Royal Statistical Society: Series B (Statistical Methodology)* 67.2 (2005), pp. 301–320.

[ZY+05] Tong Zhang, Bin Yu, et al. “Boosting with early stopping: Convergence and consistency”. In: *The Annals of Statistics* 33.4 (2005), pp. 1538–1579.