Convolutional neural networks

Third lecture

Lecturer: A. Taylor

Version of the slides: February 17, 2026.

Thanks Erwan Scornet (original slides) and Pontus Giselsson

Today's playgrounds:

- Neural nets:
  - http://playground.tensorflow.org.
- Filters/kernels:
  - https://setosa.io.
- Convnets:
  - https://cs.stanford.edu/people/karpathy/,
  - https://poloclub.github.io,
  - https://dataflowr.github.io.

A bit of history

Old techniques:

- backpropagation (Rumelhart et al. 1986),
- convolutional neural networks (CNNs) (LeCun, Boser, et al. 1989).

A bit of history

Old techniques:

- backpropagation (Rumelhart et al. 1986),
- convolutional neural networks (CNNs) (LeCun, Boser, et al. 1989).

Renewed interest: new insights for training deep neural networks

- greedy (incremental) pre-training for deep learning (Hinton et al. 2006),
- large enough datasets $\rightarrow$ no pre-training needed (L. Deng et al. 2009).

A bit of history

Old techniques:

- backpropagation (Rumelhart et al. 1986),
- convolutional neural networks (CNNs) (LeCun, Boser, et al. 1989).

Renewed interest: new insights for training deep neural networks

- greedy (incremental) pre-training for deep learning (Hinton et al. 2006),
- large enough datasets $\rightarrow$ no pre-training needed (L. Deng et al. 2009).

Technical reasons for growing interest:

- larger datasets,
- more computing power,
- minor algorithmic changes:
  - MSE replaced by cross-entropy (for classification),
  - ReLU instead of sigmoid (Fukushima 1975).

Classical neural networks for images?

No: ⚠ need to exploit data structure

- spatial organization of pixels not taken into account,
- non-robust to, e.g., image translations/shifts.

Classical neural networks for images?

No: ⚠ need to exploit data structure

- spatial organization of pixels not taken into account,
- non-robust to, e.g., image translations/shifts.

Ideas:

- apply local operations to nearby pixels (use spatial nature of images),
- repeat the operation over whole image ($\rightarrow$ shift-invariant outputs).
- Ideally:
  - construct low-level features in first layers,
  - use them to construct higher and higher-level features.

Classical neural networks for images?

No: ⚠ need to exploit data structure

- spatial organization of pixels not taken into account,
- non-robust to, e.g., image translations/shifts.

Ideas:

- apply local operations to nearby pixels (use spatial nature of images),
- repeat the operation over whole image ($\rightarrow$ shift-invariant outputs).
- Ideally:
  - construct low-level features in first layers,
  - use them to construct higher and higher-level features.

High-level intuition: through vision, we are able to

- detect oriented edges, end-points, corners (low-level features),
- combine them to detect more complex geometrical forms (high-level features).

Outline

Outline

Convolutional neural networks (CNNs)

- Neural networks that rely on **convolutions** instead of fully connected layers.
- CNNs typically involve 3 operations:
  - **activation**,
  - **convolution**, and
  - **pooling**.
- in other words: we use **parameters sharing**, through convolutions
  - *local* operation,
  - repeated accross the whole image,

  instead of **fully connected** layers (*linear* layers).

Convolution

Convolution

Convolution operator in neural networks relies on filters (or kernels) K:

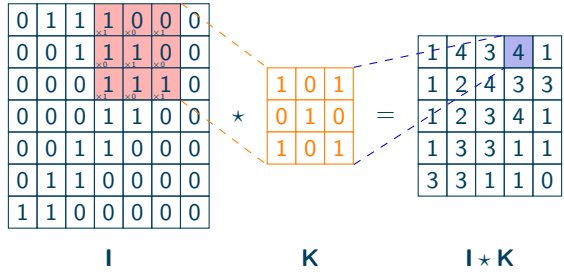
$$O(i,j) = (I \star K)(i,j) = \sum_k \sum_l I(i+k, j+l) K(k,l),$$

where sums over k and l correspond to sums over lines and columns of K, and

- O is output image,
- I is the input,
- K is the kernel (to be **learned**) $\rightarrow$ replaces weights W in a fully connected layer.

Convolution

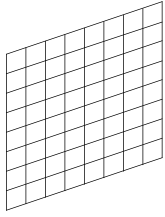
- Size of input is 7×7,
- size of filter is 3×3,
- size of output is 5×5:



$$1 \times 1 + 0 \times 0 + 0 \times 1 + 1 \times 0 + 1 \times 1 + 0 \times 0 + 1 \times 1 + 1 \times 0 + 1 \times 1 = 4$$

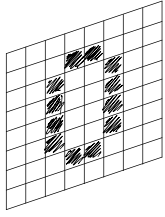
Convolution on black and white images

- Size of input image is $8 \times 8 \times 1$ (height, width, depth),



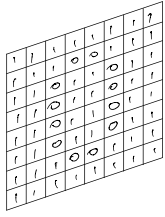
Convolution on black and white images

- Size of input image is $8 \times 8 \times 1$ (height, width, depth),



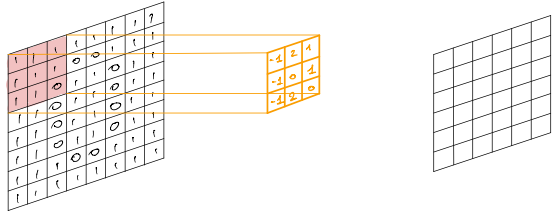
Convolution on black and white images

- Size of input image is $8 \times 8 \times 1$ (height, width, depth),



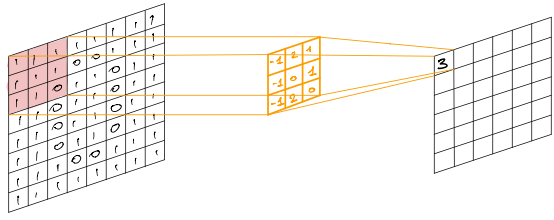
Convolution on black and white images

- Size of input image is $8 \times 8 \times 1$ (height, width, depth),
- size of the kernel is $3 \times 3 \times 1$:



Convolution on black and white images

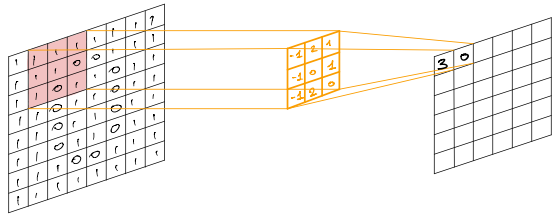
- Size of input image is $8 \times 8 \times 1$ (height, width, depth),
- size of the kernel is $3 \times 3 \times 1$:



$$1 \times (-1) + 1 \times 2 + 1 \times 1 + 1 \times (-1) + 1 \times 0 + 1 \times 1 + 1 \times (-1) + 1 \times 2 + 0 \times 0 = 3$$

Convolution on black and white images

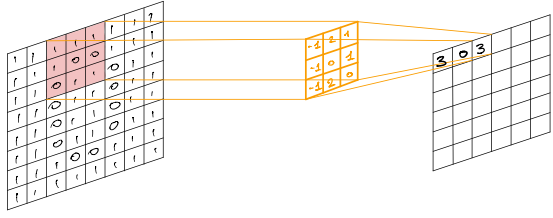
- Size of input image is $8 \times 8 \times 1$ (height, width, depth),
- size of the kernel is $3 \times 3 \times 1$:



$$1 \times (-1) + 1 \times 2 + 1 \times 1 + 1 \times (-1) + 1 \times 0 + 0 \times 1 + 1 \times (-1) + 0 \times 2 + 1 \times 0 = 0$$

Convolution on black and white images

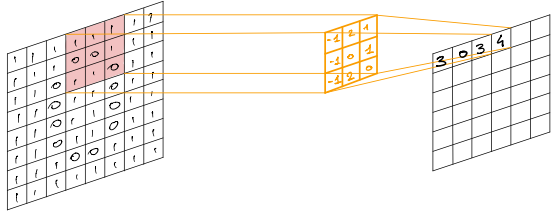
- Size of input image is $8 \times 8 \times 1$ (height, width, depth),
- size of the kernel is $3 \times 3 \times 1$:



$$1 \times (-1) + 1 \times 2 + 1 \times 1 + 1 \times (-1) + 0 \times 0 + 0 \times 1 + 0 \times (-1) + 1 \times 2 + 1 \times 0 = 3$$

Convolution on black and white images

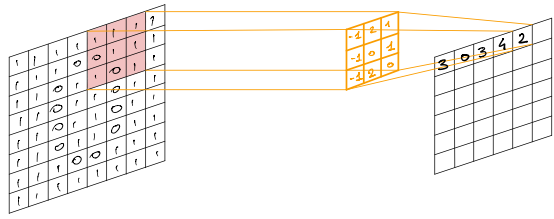
- Size of input image is $8 \times 8 \times 1$ (height, width, depth),
- size of the kernel is $3 \times 3 \times 1$:



$$1 \times (-1) + 1 \times 2 + 1 \times 1 + 0 \times (-1) + 0 \times 0 + 1 \times 1 + 1 \times (-1) + 1 \times 2 + 0 \times 0 = 4$$

Convolution on black and white images

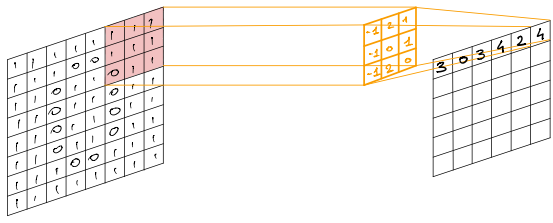
- Size of input image is $8 \times 8 \times 1$ (height, width, depth),
- size of the kernel is $3 \times 3 \times 1$:



$$1 \times (-1) + 1 \times 2 + 1 \times 1 + 0 \times (-1) + 1 \times 0 + 1 \times 1 + 1 \times (-1) + 0 \times 2 + 1 \times 0 = 2$$

Convolution on black and white images

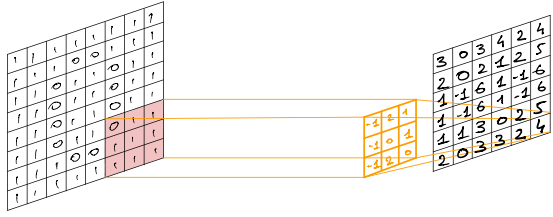
- Size of input image is $8 \times 8 \times 1$ (height, width, depth),
- size of the kernel is $3 \times 3 \times 1$:



$$1 \times (-1) + 1 \times 2 + 1 \times 1 + 1 \times (-1) + 1 \times 0 + 1 \times 1 + 0 \times (-1) + 1 \times 2 + 1 \times 0 = 4$$

Convolution on black and white images

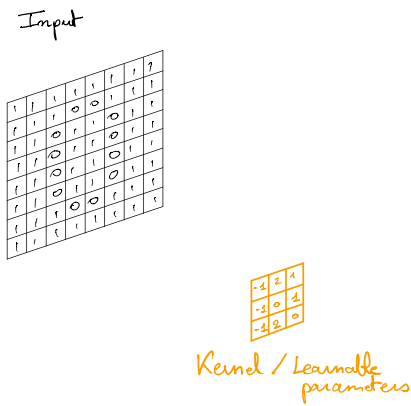
- Size of input image is $8 \times 8 \times 1$ (height, width, depth),
- size of the kernel is $3 \times 3 \times 1$:



$$0 \times (-1) + 1 \times 2 + 1 \times 1 + 1 \times (-1) + 1 \times 0 + 1 \times 1 + 1 \times (-1) + 1 \times 2 + 1 \times 0 = 4$$

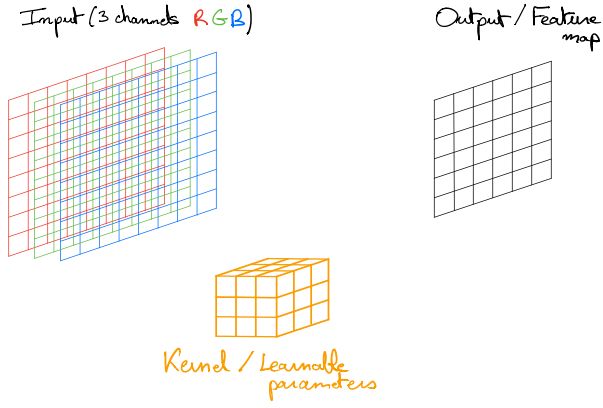
Convolution on black and white images

- Size of input image is $8 \times 8 \times 1$ (height, width, depth),
- size of the kernel is $3 \times 3 \times 1$:



Convolution - RGB

- Size of input image is $8 \times 8 \times 3$ (height, width, depth),
- size of the kernel is $3 \times 3 \times 3$:



Warning: every filter is small spatially (width and height), but extends through the full depth of input volume.

Filters

- Play with filters: `https://setosa.io/ev/image-kernels/`,
  - play with the filters: effects? sums of the weights? etc.
- example on MNIST (this afternoon).

Convolution

Convolution

Convolution operator in neural networks relies on filters (or kernels) K:

$$O(i,j) = (I \star K)(i,j) = \sum_k \sum_l \sum_c I(i+k, j+l, c) K(k,l,c),$$

where sums over k and l correspond to sums over lines and columns of K, and c corresponds to channels.

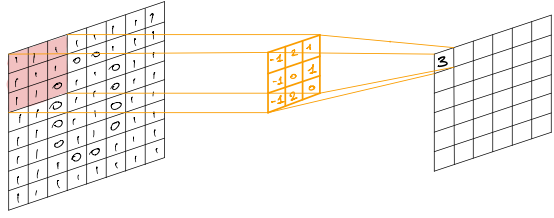
Main hyperparameters of convolutional layers (1/5)

Four hyperparameters control size of output volumes:

- **size of the kernel**: typically 3×3, 5×5,
- **depth of output volume**: number of filters/activation maps/feature maps,
- **stride**: of how many pixels do shift the filter horizontally and vertically,
- **size of zero-padding**: number of zeros we add to the borders of the image.

Main hyperparameters of convolutional layers (2/5)

Size of the kernel: typically 3×3, 5×5.



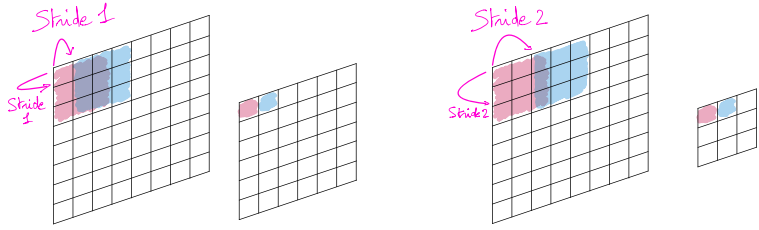
Main hyperparameters of convolutional layers (3/5)

Depth of output volumes: number of filters/activation maps/feature maps.

Main hyperparameters of convolutional layers (4/5)

Stride:

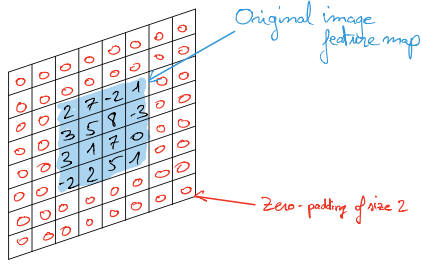
- how many pixels do we move the filter horizontally and vertically.
- Generally, stride is set to 1 (rarely to 2, and even more rarely larger).



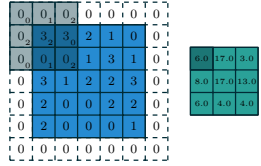
Main hyperparameters of convolutional layers (5/5)

Size of zero-padding:

- the number of zeros we add to the borders of the image.
- Typically used to obtain input and output images of same sizes.



Zero-padding



Source: ["A guide to convolution arithmetic for deep learning", Dumoulin and Visin 2016].

How to choose zero-padding?

Let

- I the height/width of the input,
- O the height/width of the output,
- P the size of the zero-padding,
- F the height/width of the filter (or spatial extent),
- S the stride.

Relationship between these quantities?

How to choose zero-padding?

Let

- I the height/width of the input,
- O the height/width of the output,
- P the size of the zero-padding,
- F the height/width of the filter (or spatial extent),
- S the stride.

Relationship between these quantities?

$$O = \left\lfloor \frac{I + 2P - F}{S} \right\rfloor + 1$$

How to choose zero-padding?

Let

- I the height/width of the input,
- O the height/width of the output,
- P the size of the zero-padding,
- F the height/width of the filter (or spatial extent),
- S the stride.

Relationship between these quantities?

$$O = \left\lfloor \frac{I + 2P - F}{S} \right\rfloor + 1$$

How do we choose zero-padding to obtain same input and output size?

Other settings (more rarely used)

Non-zero padding:

- reflection padding: example with $(1, 2)$-reflection:

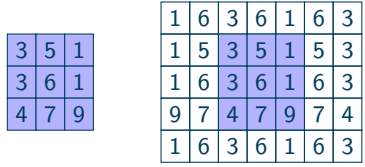
3	5	1
3	6	1
4	7	9

1	6	3	6	1	6	3
1	5	3	5	1	5	3
1	6	3	6	1	6	3
9	7	4	7	9	7	4
1	6	3	6	1	6	3

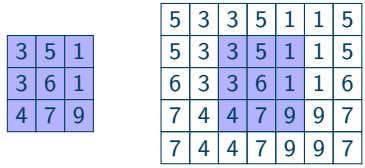
Other settings (more rarely used)

Non-zero padding:

- reflection padding: example with $(1, 2)$-reflection:



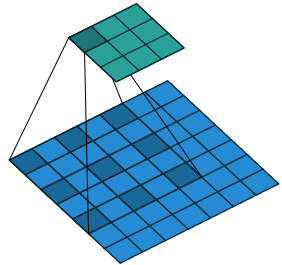
- replication (or symmetric) padding: example with $(1, 2)$-replication



Other settings (more rarely used)

Dilatation (D): *inflate* kernel by inserting holes between the kernel elements. The *dilation rate* indicates how much the kernel is widened.

$$O = \left\lfloor \frac{I + 2P - F - (F-1)(D-1)}{S} \right\rfloor + 1.$$



Source: ["A guide to convolution arithmetic for deep learning", Dumoulin and Visin 2016].

Why convolution?

- Same transformation applied to all parts of the image
  $\rightarrow$ accounts for spatial dependence between pixels and translation invariances.

Why convolution?

- Same transformation applied to all parts of the image
  $\rightarrow$ accounts for spatial dependence between pixels and translation invariances.
- Input image: millions of pixel values. But we want to detect small meaningful features (e.g., edges) using only few hundred of pixels.

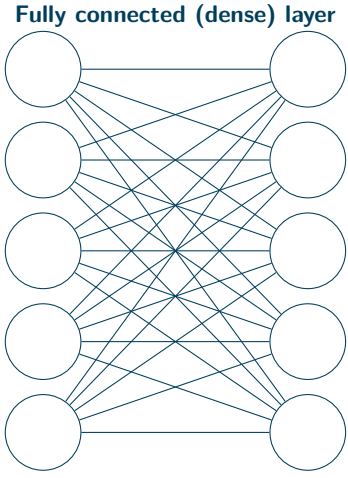
Why convolution?

- Same transformation applied to all parts of the image
  $\rightarrow$ accounts for spatial dependence between pixels and translation invariances.
- Input image: millions of pixel values. But we want to detect small meaningful features (e.g., edges) using only few hundred of pixels.
- When using matrix products: all input and output units are connected.
  $\rightarrow$ Convolutions connect only output neurons with several pixels of input image.

Why convolution?

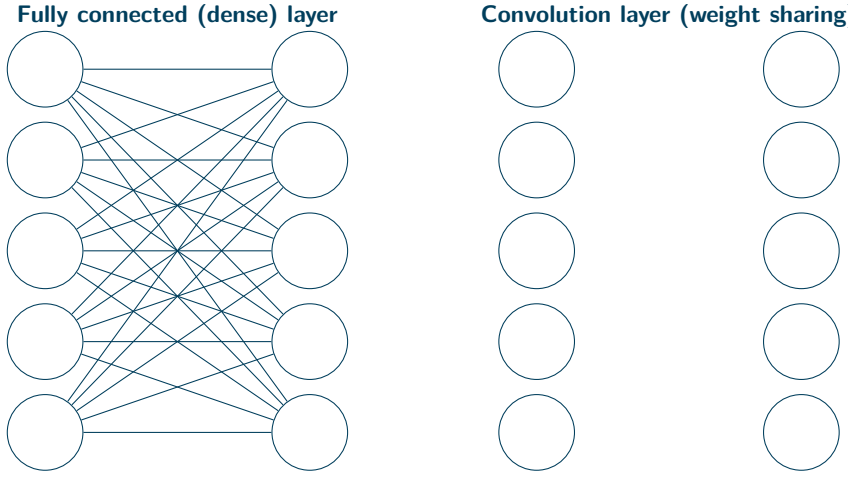
- Same transformation applied to all parts of the image
  - $\rightarrow$ accounts for spatial dependence between pixels and translation invariances.
- Input image: millions of pixel values. But we want to detect small meaningful features (e.g., edges) using only few hundred of pixels.
- When using matrix products: all input and output units are connected.
  - $\rightarrow$ Convolutions connect only output neurons with several pixels of input image.
- Convolution corresponds to **weight sharing**. It requires less parameters
  - $\rightarrow$ regularization,
  - $\rightarrow$ less memory-intensive,
  - $\rightarrow$ more statistically efficient,
  - $\rightarrow$ computationally faster.

Sparse connections / weight sharing (1D image)



- Left: all outputs connected to all inputs. Connectivity is **dense** (dense layer).
- Right: convolution with kernel (width 3). Connectivity is **sparse**.

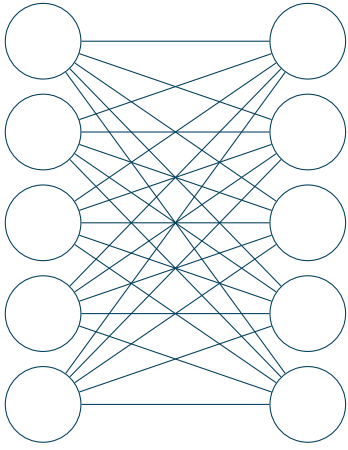
Sparse connections / weight sharing (1D image)



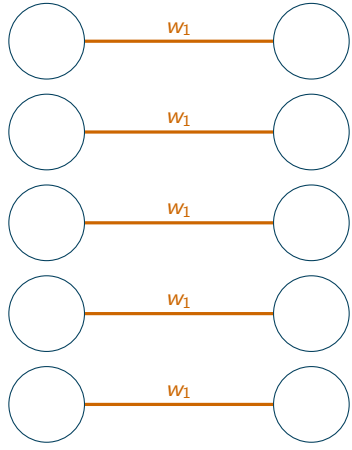
- Left: all outputs connected to all inputs. Connectivity is **dense** (dense layer).
- Right: convolution with kernel (width 3). Connectivity is **sparse**.

Sparse connections / weight sharing (1D image)

Fully connected (dense) layer



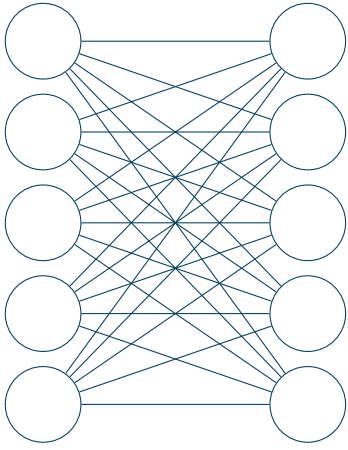
Convolution layer (weight sharing)



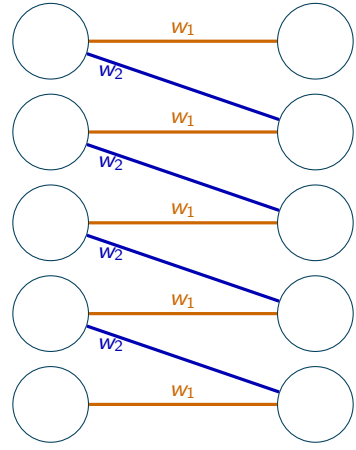
- Left: all outputs connected to all inputs. Connectivity is **dense** (dense layer).
- Right: convolution with kernel (width 3). Connectivity is **sparse**.

Sparse connections / weight sharing (1D image)

Fully connected (dense) layer



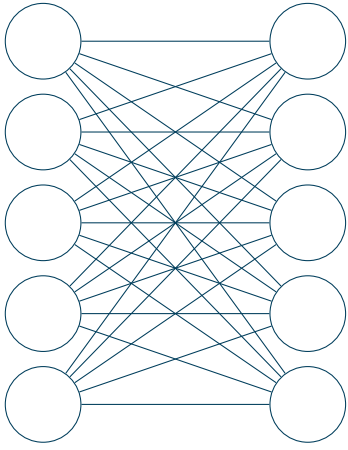
Convolution layer (weight sharing)



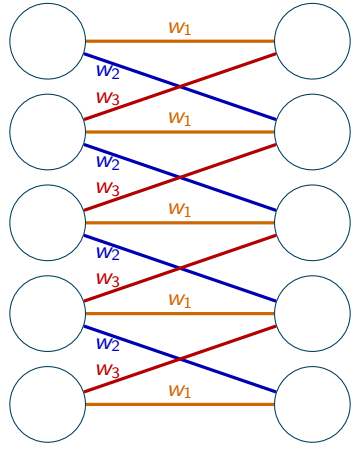
- Left: all outputs connected to all inputs. Connectivity is **dense** (dense layer).
- Right: convolution with kernel (width 3). Connectivity is **sparse**.

Sparse connections / weight sharing (1D image)

Fully connected (dense) layer

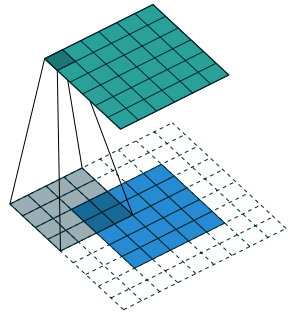


Convolution layer (weight sharing)



- Left: all outputs connected to all inputs. Connectivity is **dense** (dense layer).
- Right: convolution with kernel (width 3). Connectivity is **sparse**.

Hyperparameters here?



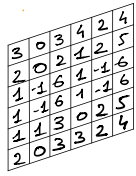
Source: ["A guide to convolution arithmetic for deep learning", Dumoulin and Visin 2016].

Outline

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

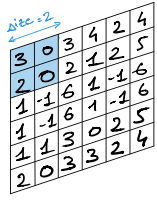
- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

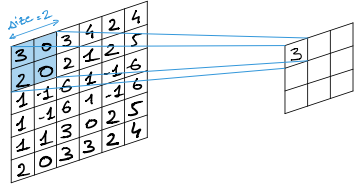
- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

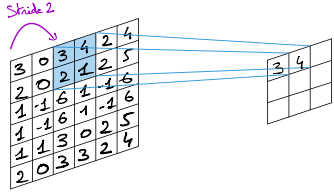
- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

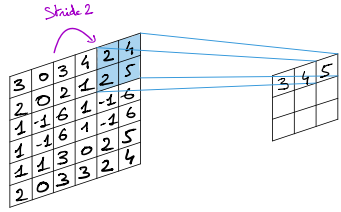
- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

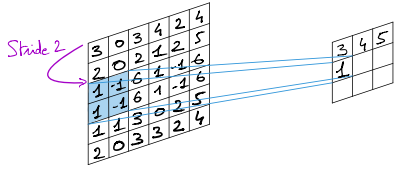
- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

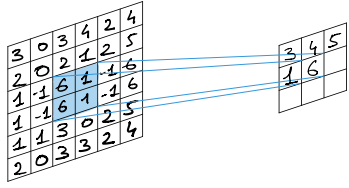
- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

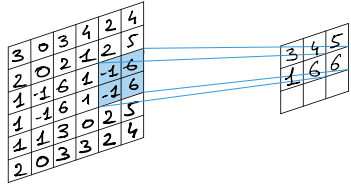
- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

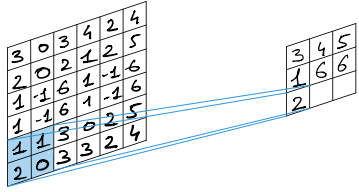
- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

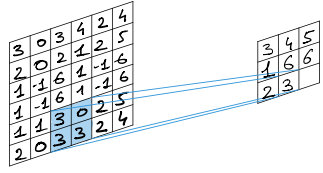
- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

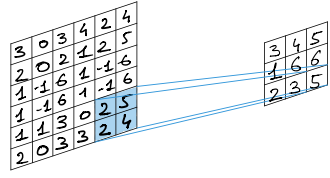
- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

A pooling layer:

- operates independently on every depth slice of input
- resizes the width/height.



Parameters:

- stride $S = 2$
- spatial extend $F = 2$.

Usually, $S = F = 2$. Less frequent: $F = 3, S = 2$ (overlapping pooling).

Pooling

- Pooling layer:
  - each output pixel is summary statistic of neighboring input pixels.
- Most widely used is the max aggregation, referred to as **max-pooling**.

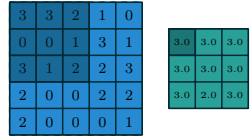
Pooling

- Pooling layer:
  - each output pixel is summary statistic of neighboring input pixels.
- Most widely used is the max aggregation, referred to as **max-pooling**.
- Pooling helps representations to become (approximately) invariant to small translations of input.
  $\rightarrow$ if small translation is applied to input: output is almost unchanged.

Pooling

- Pooling layer:
  - each output pixel is summary statistic of neighboring input pixels.
- Most widely used is the max aggregation, referred to as **max-pooling**.
- Pooling helps representations to become (approximately) invariant to small translations of input.
  $\rightarrow$ if small translation is applied to input: output is almost unchanged.
- Very useful if we care more about presence/absence of some feature than its position.
  $\rightarrow$ Example: for face detection, presence/absence of eyes more important their positions.

Pooling

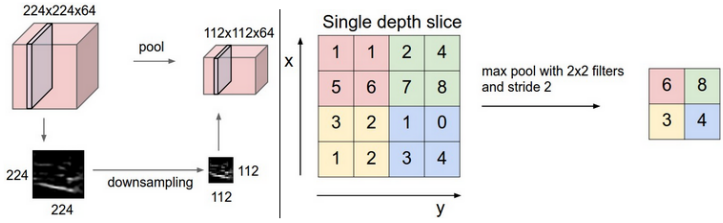


Source: ["A guide to convolution arithmetic for deep learning", Dumoulin and Visin 2016].

Pooling

Classical pooling:

- average-pooling (often used historically)
- max-pooling: often better in practice.

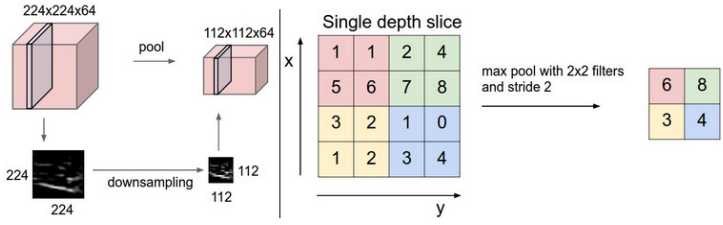


https://compsci682.github.io/notes/convolutional-networks/

Pooling

Classical pooling:

- average-pooling (often used historically)
- max-pooling: often better in practice.



https://compsci682.github.io/notes/convolutional-networks/

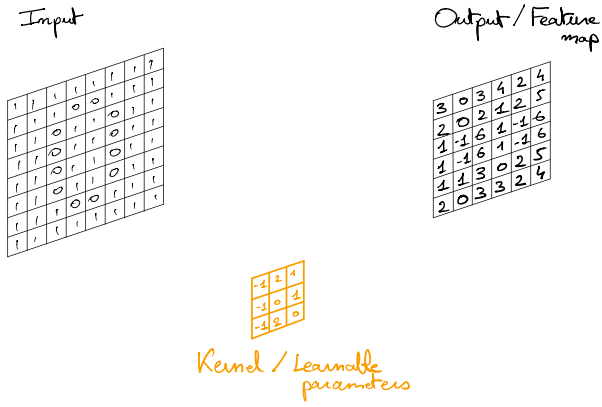
Getting rid of pooling? Pooling is not always popular/liked: can we get away without it:

- example: ["Striving for simplicity: The all convolutional net", Springenberg et al. 2014],
- architecture: only consists convolutional layers,
- reduced representation sizes using larger strides once in a while,
- discarding pooling layers has gained popularity (e.g., for generative models),

$\rightarrow$ future architectures might get rid of pooling layers.

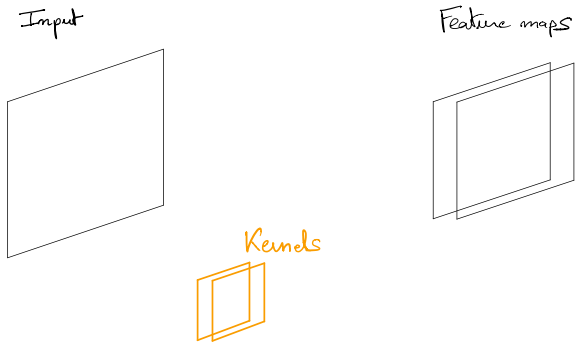
Possible architecture for a CNN

Consider grayscale image. Each kernel of first layer produces one feature map.



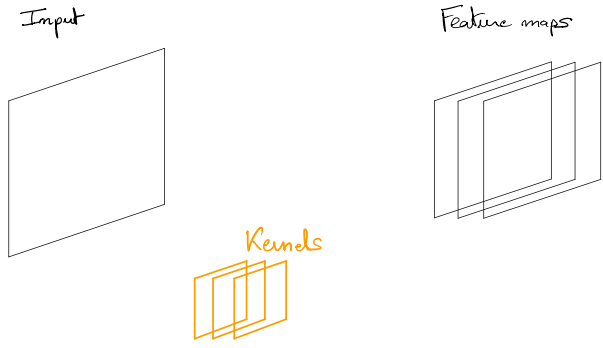
Possible architecture for a CNN

Consider grayscale image. Each kernel of first layer produces one feature map.



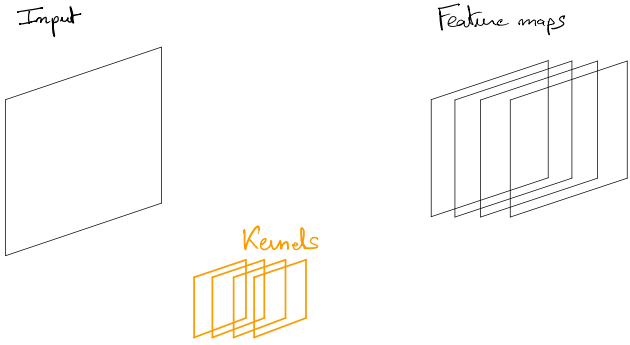
Possible architecture for a CNN

Consider grayscale image. Each kernel of first layer produces one feature map.



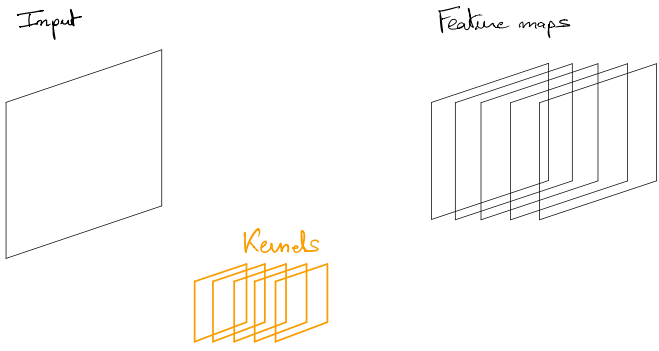
Possible architecture for a CNN

Consider grayscale image. Each kernel of first layer produces one feature map.



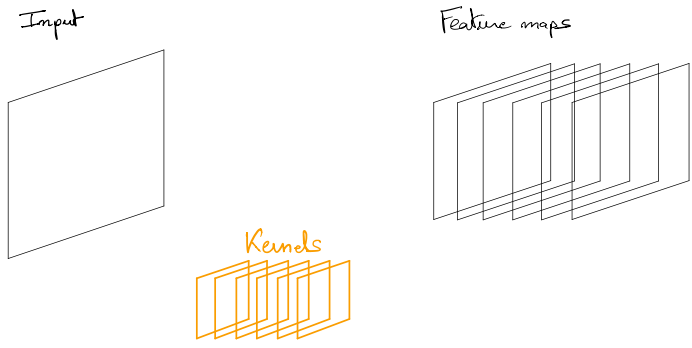
Possible architecture for a CNN

Consider grayscale image. Each kernel of first layer produces one feature map.



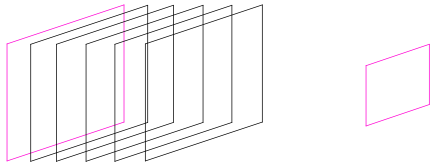
Possible architecture for a CNN

Consider grayscale image. Each kernel of first layer produces one feature map.



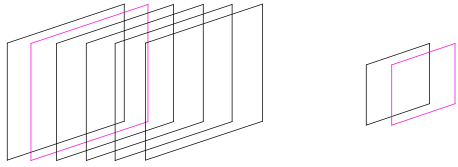
Possible architecture for a CNN

Pooling layer operates on each feature map separately.



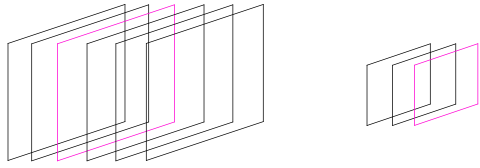
Possible architecture for a CNN

Pooling layer operates on each feature map separately.



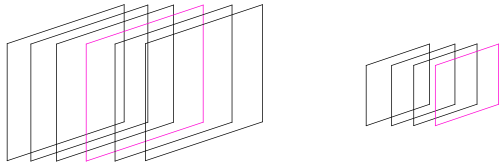
Possible architecture for a CNN

Pooling layer operates on each feature map separately.



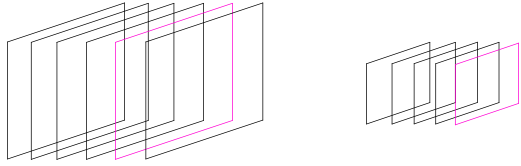
Possible architecture for a CNN

Pooling layer operates on each feature map separately.



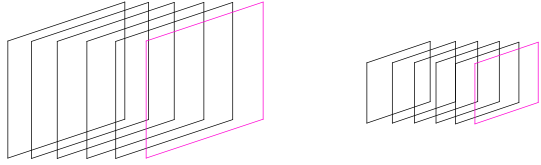
Possible architecture for a CNN

Pooling layer operates on each feature map separately.

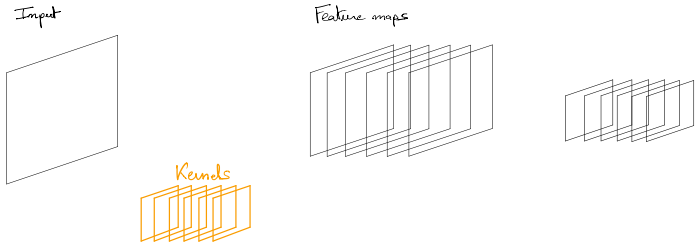


Possible architecture for a CNN

Pooling layer operates on each feature map separately.

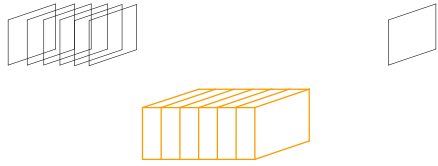


Possible architecture for a CNN



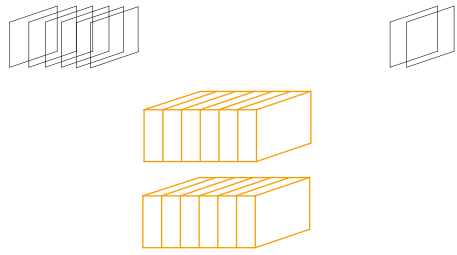
Possible architecture for a CNN

Convolutional layer operates on feature maps output by pooling layer. Each kernel is volume whose depth equals depth of input volume.



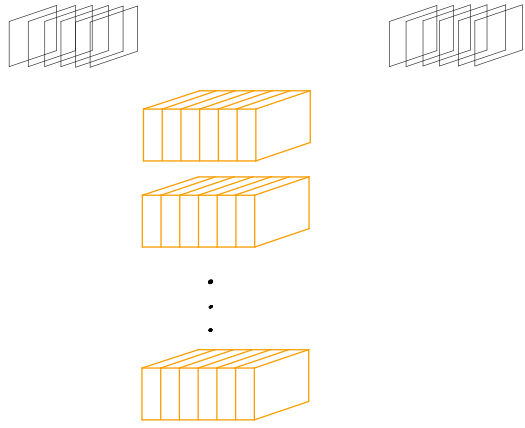
Possible architecture for a CNN

Convolutional layer operates on feature maps output by pooling layer. Each kernel is volume whose depth equals depth of input volume.

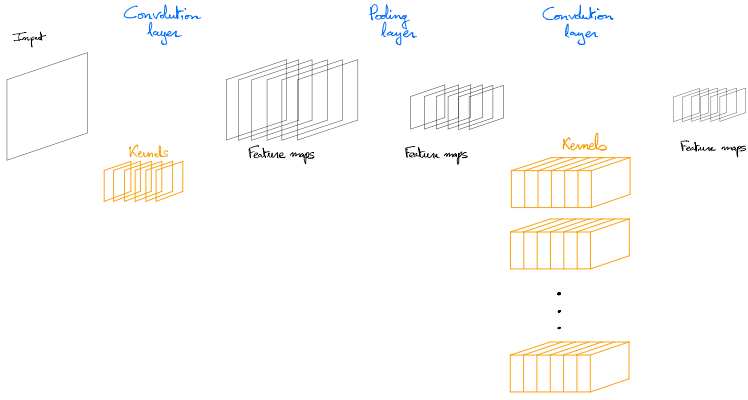


Possible architecture for a CNN

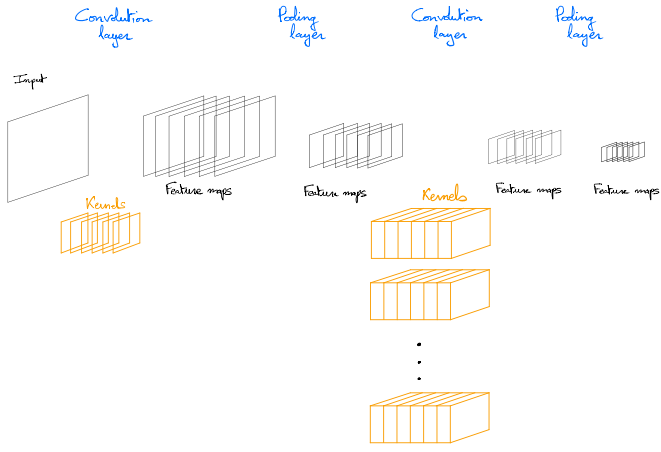
Convolutional layer operates on feature maps output by pooling layer. Each kernel is volume whose depth equals depth of input volume.



Possible architecture for a CNN

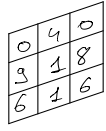


Possible architecture for a CNN



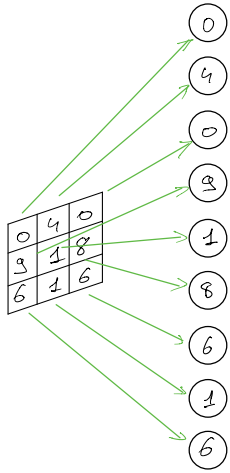
Possible architecture for a CNN

At end of network: flattened feature maps and fully connected layers.



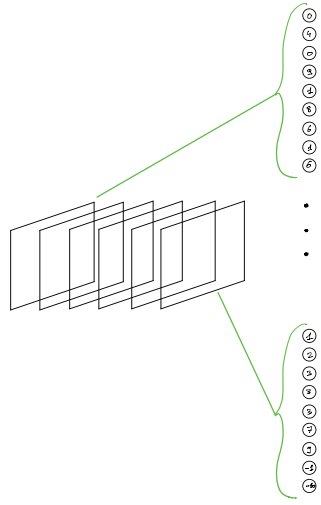
Possible architecture for a CNN

At end of network: flattened feature maps and fully connected layers.



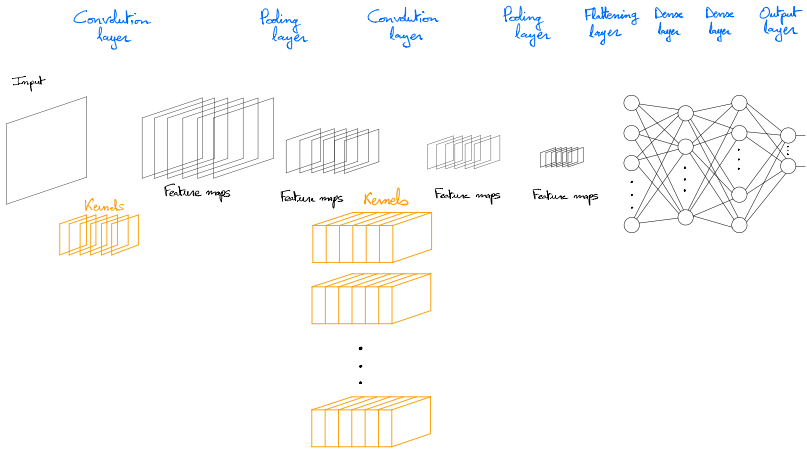
Possible architecture for a CNN

At end of network: flattened feature maps and fully connected layers.



Possible architecture for a CNN

Full architecture is summarized below.



Outline

Data processing

Normalized data.
For each channel R, G, B, center and normalize data:

- compute pixel mean value over all images in whole data set. Subtract this value to each channel of each image.
- Normalize pixel values (either between 0 and 1 or wrt standard deviation).

$\rightarrow$ no relative information lost between images.

Data augmentation.

1. Sampling ["Imagenet large scale visual recognition challenge", Russakovsky et al. 2015]
2. Translation/shifting ["Deep convolutional neural networks and data augmentation for environmental sound classification", Salamon and Bello 2017]
3. Horizontal reflection/mirroring ["Mirror, mirror on the wall, tell me, is the error small?", Yang and Patras 2015]
4. Rotating ["Holistically-nested edge detection", Xie and Tu 2015]
5. Various photometric transformations ["Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture", Eigen and Fergus 2015]

Prediction. At test time:

- patches extracted from new images together with some reflection/translation/...
- Predictions for each artificial image are aggregated for final prediction.

Adding noise — data augmentation and regularization

- Add noise to input

  ["Training with noise is equivalent to Tikhonov regularization", Bishop 1995]

  ["Adding noise to the input of a model trained with a regularized objective", Rifai et al. 2011]

  ["Explaining and harnessing adversarial examples", Goodfellow et al. 2014]

- Add noise to weights

  ["An analysis of noise in recurrent neural networks: convergence and generalization", Jim et al. 1996]

  ["Practical variational inference for neural networks", Graves 2011]

- Add noise to output

  ["Randomizing outputs to increase prediction accuracy", Breiman 2000]

- Select best data transformations (computationally expensive, many re-trainings).

  ["Transformation pursuit for image classification", Paulin et al. 2014]

Outline

Backpropagation

Recall ideas:

- chain rule,
- *clean* formulas for traditional fully-connected layers.
- What do we have to change for CNNs?
  - → weight sharing makes it a bit different.
    ... modifications are (morally) minor,
    ... but requires a bit of thoughts for implementations (that you have already met!).

Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider 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 (ter): effect of weight sharing?

Consider simple unidimensional network:

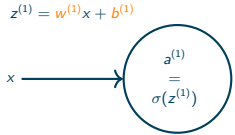
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).

x

Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

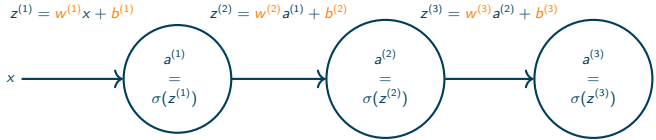
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

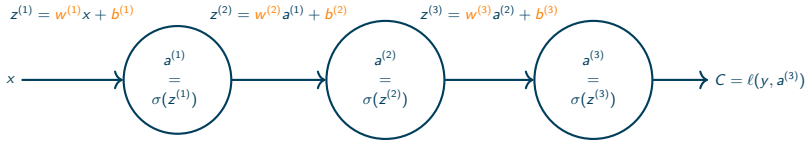
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

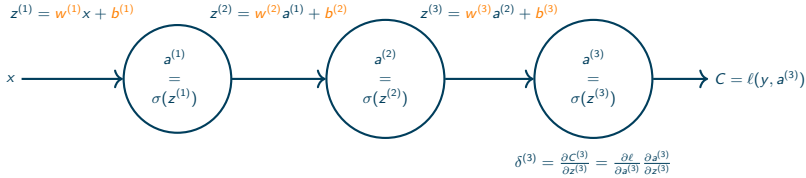
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

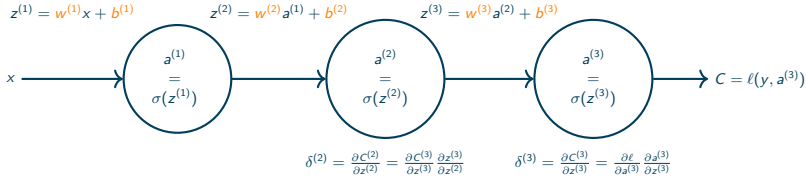
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

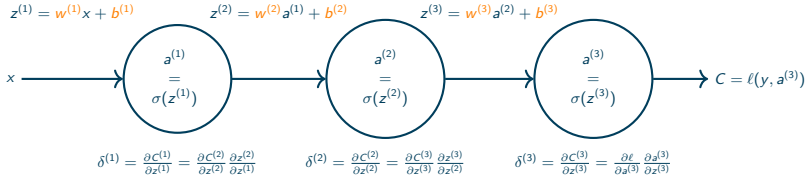
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

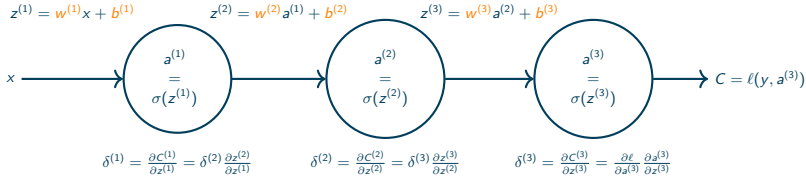
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

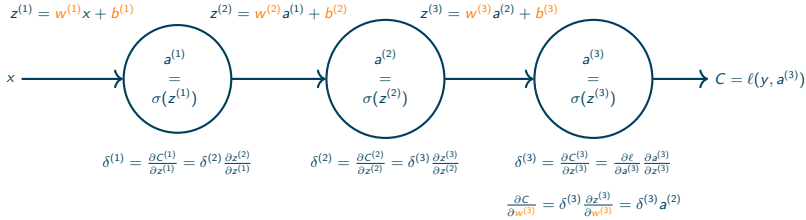
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

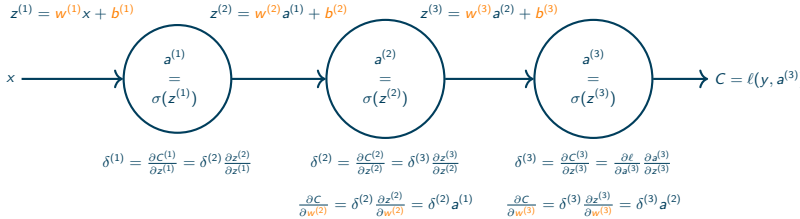
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

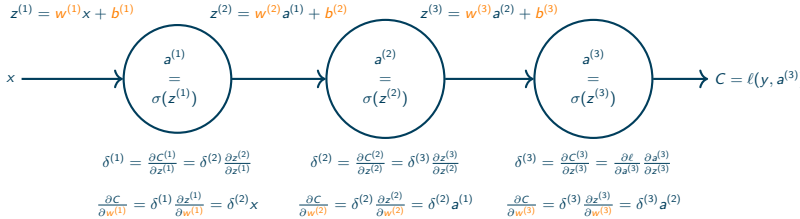
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

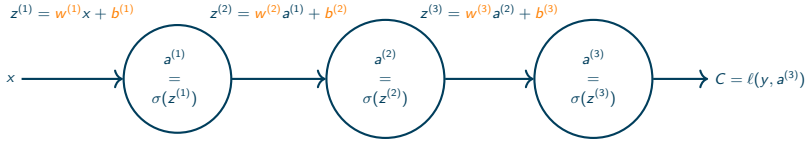
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

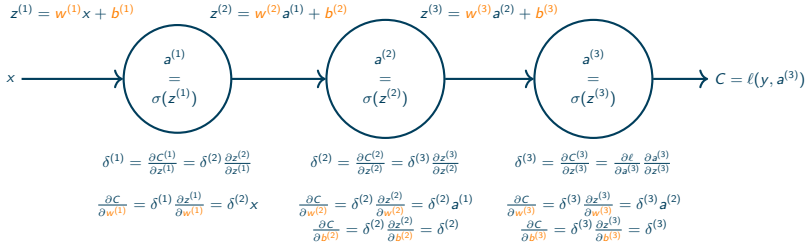
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

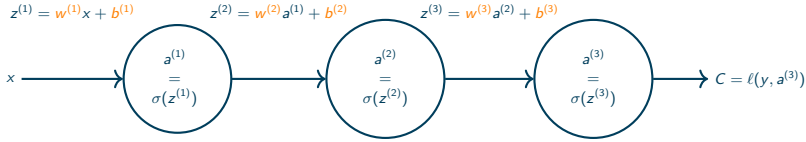
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider simple unidimensional network:

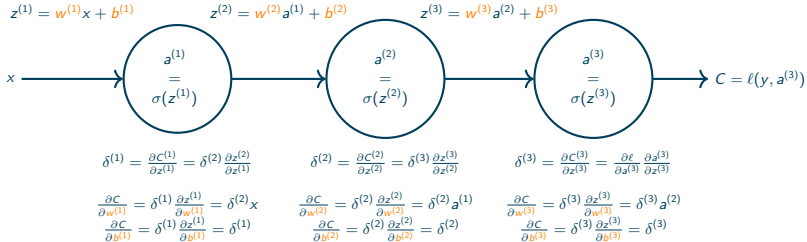
- dataset $x_i \in \mathbb{R}$ and $y_i \in \{-1, 1\}$ for $i = 1, \ldots, n$ (all datapoints),
- parameters $w^{(k)} \in \mathbb{R}$, $b^{(k)} \in \mathbb{R}$ for $k = 1, \ldots, L$ (all layers).



Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider 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).



What if we force $w^{(1)} = w^{(2)} = w^{(3)} \triangleq w$ and $b^{(1)} = b^{(2)} = b^{(3)} \triangleq b$?

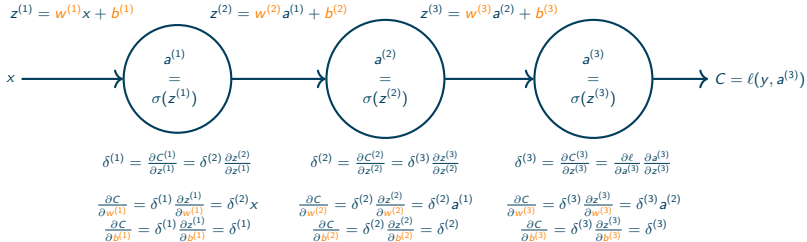
$$\frac{\partial C}{\partial w} =$$

$$\frac{\partial C}{\partial b} =$$

Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider 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).



What if we force $w^{(1)} = w^{(2)} = w^{(3)} \triangleq w$ and $b^{(1)} = b^{(2)} = b^{(3)} \triangleq b$?

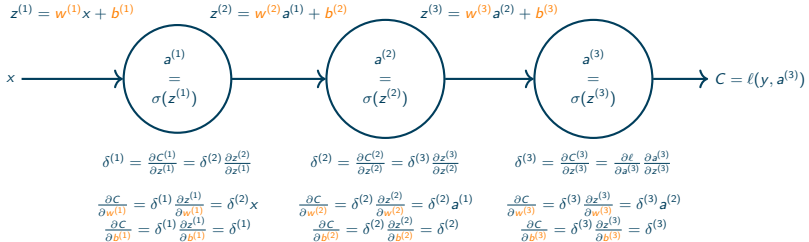
$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial w^{(1)}} + \frac{\partial C}{\partial w^{(2)}} + \frac{\partial C}{\partial w^{(3)}},$$

$$\frac{\partial C}{\partial b} =$$

Unrolling derivatives via chain rule (ter): effect of weight sharing?

Consider 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).



What if we force $w^{(1)} = w^{(2)} = w^{(3)} \triangleq w$ and $b^{(1)} = b^{(2)} = b^{(3)} \triangleq b$?

$$\frac{\partial C}{\partial w} = \frac{\partial C}{\partial w^{(1)}} + \frac{\partial C}{\partial w^{(2)}} + \frac{\partial C}{\partial w^{(3)}},$$
$$\frac{\partial C}{\partial b} = \frac{\partial C}{\partial b^{(1)}} + \frac{\partial C}{\partial b^{(2)}} + \frac{\partial C}{\partial b^{(3)}}.$$

In Pytorch: zero grad?

```python
for e in range(epochs):
    running_loss = 0
    running_test_loss=0

    for images, labels in trainloader:
        # Flatten MNIST images into a 784 long vector
        images = images.view(images.shape[0], -1)

        # Training pass
        optimizer.zero_grad() # What is this line for ?

        output = model(images)
        loss = criterion(output, labels)

        # now: backpropagate and perform one optimization step
        loss.backward() # Computes the gradients
        optimizer.step() # Perform an iteration of the optimization algorithm (SGD?) step.

        running_loss += loss.item()
```

In Pytorch: no grad?

```python
for e in range(epochs):
    running_loss = 0
    running_test_loss=0

    for images, labels in trainloader:
        # Flatten MNIST images into a 784 long vector
        images = images.view(images.shape[0], -1)

        # Training pass
        optimizer.zero_grad() # What is this line for ?

        output = model(images)
        loss = criterion(output, labels)

        # now: backpropagate and perform one optimization step
        loss.backward() # Computes the gradients
        optimizer.step() # Perform an iteration of the optimization algorithm (SGD?) step.

        running_loss += loss.item()

    #Compute validation loss
    with torch.no_grad():
        for images, labels in testloader:
            # Flatten MNIST images into a 784 long vector
            images = images.view(images.shape[0], -1)

            # Complete those lines:
            outputs = model(images)
            test_loss = criterion(outputs, labels)
            running_test_loss += test_loss.item()

            predictions = torch.argmax(outputs, 1)
            correct_test = predictions.eq(labels).sum().item()

            test_accuracy = 100 * correct_test / len(predictions)
```

Outline

Example

Let's exercise with MNIST (recall: 28×28 pictures of digits 0–9):

- previous practical session (learning): MNIST with dense layers,
- alternative #1: learn simple filters for MNIST,
- alternative #2: MNIST with base filters (not learned)? As an exercise.
- alternative #3: MNIST with base filters using pre-trained models? As an exercise.

How many parameters? What performance?

Example

Let's exercise with MNIST (recall: 28×28 pictures of digits 0–9):

- previous practical session (learning): MNIST with dense layers,
- alternative #1: learn simple filters for MNIST,
- alternative #2: MNIST with base filters (not learned)? As an exercise.
- alternative #3: MNIST with base filters using pre-trained models? As an exercise.

How many parameters? What performance?

Pytorch: what does this do?

```
class myLogisticNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden1 = nn.Linear(784, 10)
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.hidden1(x)
        x = self.logsoftmax(x)
        return x
```

Network 0 (myLogisticNet)

- Network structure:

```
$ model0 = myLogisticNet()
$ summary(model0, (1,1,784))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1          [-1, 1, 1, 10]           7,850
       LogSoftmax-2          [-1, 1, 1, 10]               0
================================================================
Total params: 7,850
Trainable params: 7,850
Non-trainable params: 0
----------------------------------------------------------------
```

- Train with SGD (learning rate 0.01 with 10 epochs)
  - train loss: 0.236
  - test loss: 0.273
  - test accuracy: 92.53

Pytorch: what does this do?

```
class myReLUNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.hidden1 = nn.Linear(784, 128)
        self.hidden2 = nn.Linear(128, 64)
        self.output = nn.Linear(64, 10)
        self.activation1 = nn.ReLU()
        self.activation2 = nn.ReLU()
        self.logsoftmax = nn.LogSoftmax(dim=1)


    def forward(self, x):
        x = self.hidden1(x)
        x = self.activation1(x)
        x = self.hidden2(x)
        x = self.activation2(x)
        x = self.output(x)
        x = self.logsoftmax(x)
        return x
```

Network 1 (myReLUNet)

- Network structure:

```
$ model1 = myReLUNet()
$ summary(model1, (1,1,784))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Linear-1          [-1, 1, 1, 128]         100,480
              ReLU-2          [-1, 1, 1, 128]               0
            Linear-3           [-1, 1, 1, 64]           8,256
              ReLU-4           [-1, 1, 1, 64]               0
            Linear-5           [-1, 1, 1, 10]             650
       LogSoftmax-6           [-1, 1, 1, 10]               0
================================================================
Total params: 109,386
Trainable params: 109,386
Non-trainable params: 0
----------------------------------------------------------------
```

- Train with SGD (learning rate 0.01 with 10 epochs)
  - train loss: 0.114
  - test loss: 0.12
  - test accuracy: 96.46

Pytorch: what does this do?

- Network structure (see documentation).

```
class mySimpleConvNet(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, 3, 1, 1)
        self.conv2 = nn.Conv2d(32, 64, 3, 1)
        self.max1 = nn.MaxPool2d(2)
        self.max2 = nn.MaxPool2d(2)
        self.fc1 = nn.Linear(2304, 10)
        self.logsoftmax = nn.LogSoftmax(dim=1)

    def forward(self, x):
        x = self.conv1(x)
        x = self.max1(x)
        x = self.conv2(x)
        x = self.max2(x)
        x = torch.flatten(x, 1)
        x = self.fc1(x)
        x = self.logsoftmax(x)
        return x
```

Network 2 (mySimpleConvNet)

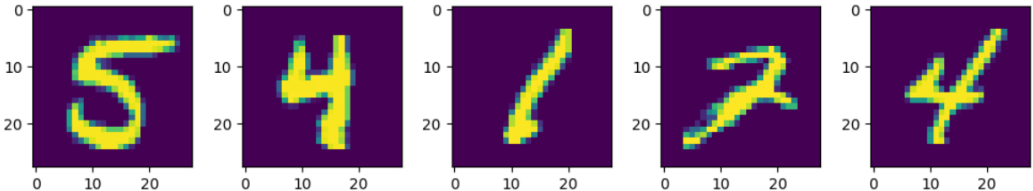
- Network structure:

```
$ model2 = mySimpleConvNet()
$ summary(model2, (1,28,28))
```

```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 32, 28, 28]             320
         MaxPool2d-2          [-1, 32, 14, 14]               0
            Conv2d-3          [-1, 64, 12, 12]          18,496
         MaxPool2d-4            [-1, 64, 6, 6]               0
            Linear-5                 [-1, 10]          23,050
       LogSoftmax-6                 [-1, 10]               0
================================================================
Total params: 41,866
Trainable params: 41,866
Non-trainable params: 0
----------------------------------------------------------------
```

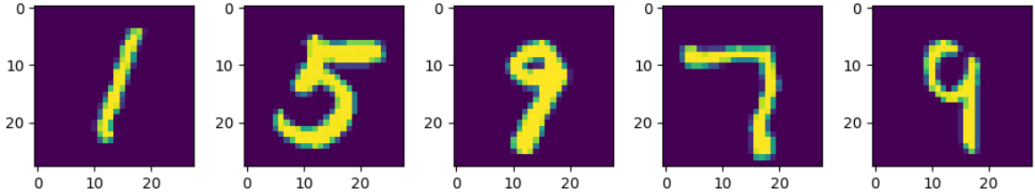
- Train with SGD (learning rate 0.01 with 10 epochs)
  - train loss: 0.044,
  - test loss: 0.043,
  - test accuracy: 98.68.

My network 0 vs. network 1 vs. my network 2



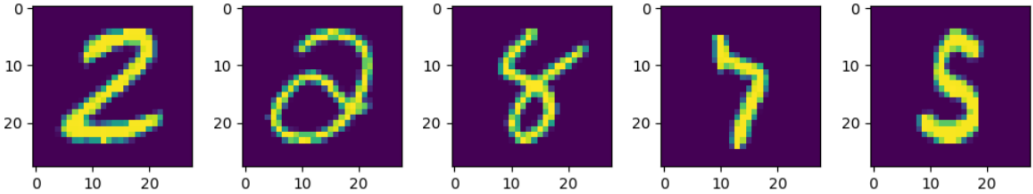
- True labels: $[5, 4, 1, 2, 4]$,
- predicted (myLogisticNet): $[5, 4, 1, 2, 4]$,
- predicted (myReLUNet): $[5, 4, 1, 7, 4]$,
- predicted (mySimpleConvNet): $[5, 4, 1, 7, 4]$.

My network 0 vs. network 1 vs. my network 2



- True labels: $[1, 5, 9, 7, 9]$,
- predicted (myLogisticNet): $[1, 5, 9, 7, 4]$,
- predicted (myReLUNet): $[1, 5, 9, 7, 4]$,
- predicted (mySimpleConvNet): $[1, 5, 9, 7, 9]$.

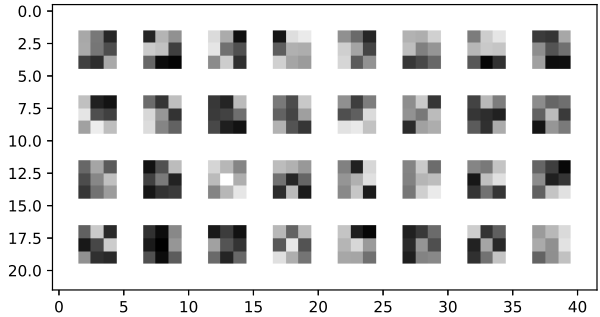
My network 0 vs. network 1 vs. my network 2



- True labels: $[2, 2, 8, 7, 5]$,
- predicted (myLogisticNet): $[2, 3, 8, 7, 8]$,
- predicted (myReLUNet): $[2, 3, 8, 7, 5]$,
- predicted (mySimpleConvNet): $[2, 2, 8, 7, 5]$.

Vizualizing CNNs

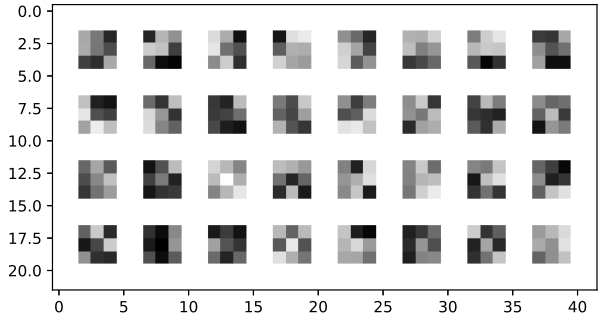
- Visualize filters?



... mostly for first layers (not too many filters, somewhat low dimensions; but already difficult to visualize).

Vizualizing CNNs

- Visualize filters?

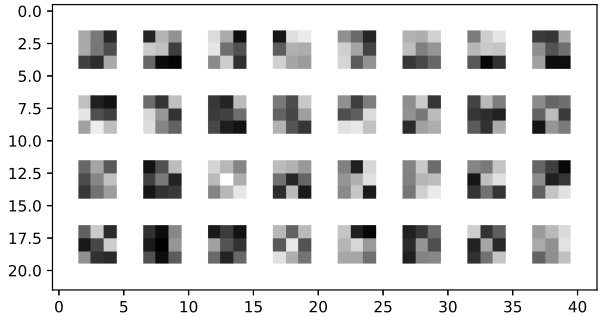


... mostly for first layers (not too many filters, somewhat low dimensions; but already difficult to visualize).

- Visualize feature maps? Examples:
  - filters and feature maps,
  - example on CIFAR-10: `https://poloclub.github.io/cnn-explainer/`,
  - deconvnets, and guided backpropagation.

Vizualizing CNNs

- Visualize filters?



... mostly for first layers (not too many filters, somewhat low dimensions; but already difficult to visualize).

- Visualize feature maps? Examples:
  - filters and feature maps,
  - example on CIFAR-10: `https://poloclub.github.io/cnn-explainer/`,
  - deconvnets, and guided backpropagation.

... overall, many techniques: they are important for improving & debugging networks.

Poll!

How many parameters (weights, biases) are there?

- good sanity check for understanding (but also check docs)!

https://app.wooclap.com/CGYRYZ?from=instruction-slide

Outline

Famous architectures

Deep learning is **experimental**. Efficient learning involves:

- experiments (architectures, hyperparameters, data preprocessing, etc.)

Famous architectures

Deep learning is **experimental**. Efficient learning involves:

- experiments (architectures, hyperparameters, data preprocessing, etc.)
- read works / feedbacks by others,

Famous architectures

Deep learning is **experimental**. Efficient learning involves:

- experiments (architectures, hyperparameters, data preprocessing, etc.)
- read works / feedbacks by others,
- use simple baselines,

Famous architectures

Deep learning is **experimental**. Efficient learning involves:

- experiments (architectures, hyperparameters, data preprocessing, etc.)
- read works / feedbacks by others,
- use simple baselines,
- don't hesitate to use pre-trained models,
  → many known architecture have parameters available online.
  examples: AlexNet, VGG16 (see practical session). More info:
  `https://pytorch.org/vision/stable/models.html`

Famous architectures

Deep learning is **experimental**. Efficient learning involves:

- experiments (architectures, hyperparameters, data preprocessing, etc.)
- read works / feedbacks by others,
- use simple baselines,
- don't hesitate to use pre-trained models,
  → many known architecture have parameters available online.
  examples: AlexNet, VGG16 (see practical session). More info:
  `https://pytorch.org/vision/stable/models.html`
- read documentations!

Outline

Common datasets

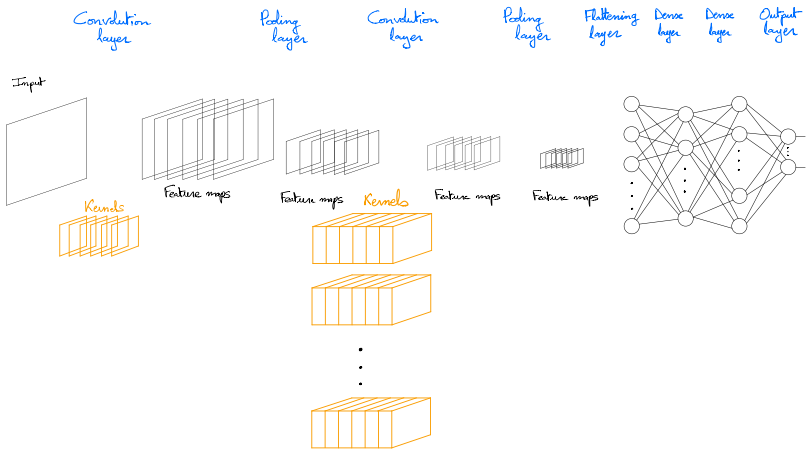
- MNIST
  - 28×28 images (black & white), 10 classes.
  - Training set: 60000 images. Test set: 10000 images.
- SVHN (`http://ufldl.stanford.edu/housenumbers/`)
  - 32×32 images (RGB), 10 classes.
  - Training set: 73257 images (+531131 easier ones). Test set: 26032.
- CIFAR-10 (`https://www.cs.toronto.edu/~kriz/cifar.html`)
  - 32×32 images (RGB), 10 classes.
  - Training set: 50000 images. Test set: 10000 images.
  - 6000 images per class.
- CIFAR-100 (`https://www.cs.toronto.edu/~kriz/cifar.html`)
  - 32×32 images (RGB), 100 classes.
  - Training set: 50000 images. Test set: 10000 images.
  - 600 images per class.
  - 20 "superclasses" (coarse labels).
- ImageNet (`https://www.image-net.org/download.php`, description)
  - Most highly-used subset is *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) 2012-2017.
  - Images of various sizes (RGB), 1000 classes.
  - Training set: $\sim$ 1.200.000 images, validation+test set: $\sim$ 150.000 images.

Outline

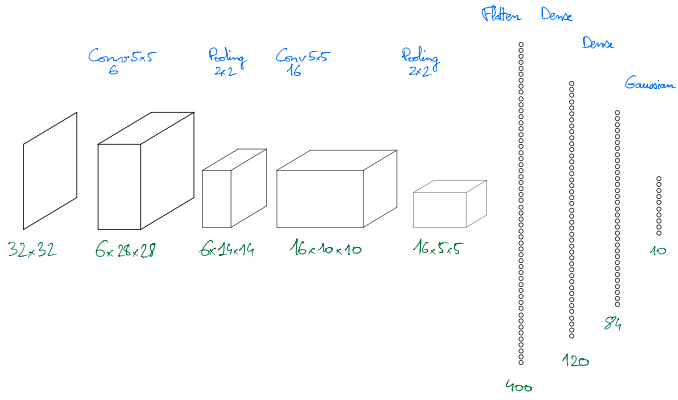
LeNet

["Generalization and network design strategies", LeCun et al. 1989]

["Gradient-based learning applied to document recognition", LeCun, Bottou, et al. 1998]



LeNet



First layer (C1): convolutional layer with

- kernel size $= 5 \times 5$ and a bias,
- stride $= 1$ (overlapping contiguous receptive fields),
- zero-padding $= 0$,
- output: 6 different feature maps (5×5 kernels with activation function σ).

Second layer (S2): subsampling/pooling layer with

- type of pooling: averaging,
- kernel size $= 2 \times 2$,
- stride $= 2$ (non-overlapping receptive fields),
- zero-padding $= 0$,
- output: one feature map per input feature map resulting from the operation

$$\sigma((2 \times 2 \text{ averaging})w + b),$$

with w and b (non-standard) learned parameters for the pooling layer.

Second layer (S2): subsampling/pooling layer with

- type of pooling: averaging,
- kernel size $= 2 \times 2$,
- stride $= 2$ (non-overlapping receptive fields),
- zero-padding $= 0$,
- output: one feature map per input feature map resulting from the operation

$$\sigma((2 \times 2 \text{ averaging})w + b),$$

with w and b (non-standard) learned parameters for the pooling layer.

Third-layer (C3): convolutional layer

- ⚠ this layer operates on several feature maps whereas layer (C1) operates on the input image (depth $= 1$).
- Each feature map is connected to specific input feature maps in order to
  - reduce the number of connections,
  - break symmetries between different layers of the network.

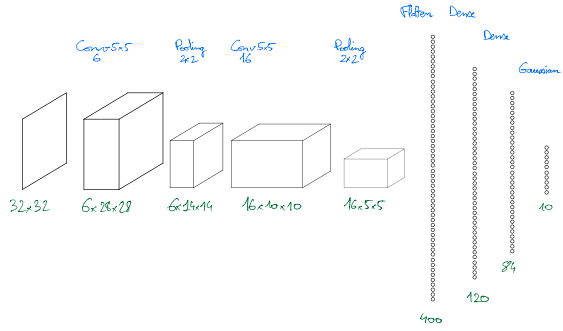
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	X				X	X	X			X	X	X	X		X	X
1	X	X				X	X	X			X	X	X	X		X
2	X	X	X				X	X	X			X		X	X	X
3		X	X	X			X	X	X	X			X		X	X
4			X	X	X			X	X	X	X		X	X		X
5				X	X	X			X	X	X	X		X	X	X

Remaining layers

- Layer S4: Pooling layer as before,
- layer F5: fully-connected layer with 120 outputs
- layer F6: fully-connected layer with 84 outputs,
- output: “Gaussian” layer (see later).

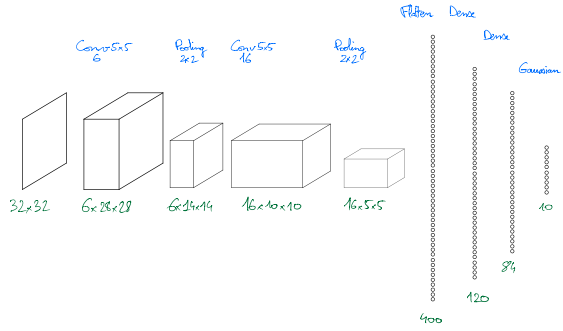
Remaining layers

- Layer S4: Pooling layer as before,
- layer F5: fully-connected layer with 120 outputs
- layer F6: fully-connected layer with 84 outputs,
- output: “Gaussian” layer (see later).



Remaining layers

- Layer S4: Pooling layer as before,
- layer F5: fully-connected layer with 120 outputs
- layer F6: fully-connected layer with 84 outputs,
- output: “Gaussian” layer (see later).



Classical **bi-pyramidal structure**:

- number of feature maps increases, while
- spatial resolutions decrease.

Output layer

Radial basis function (RBF) units. Neuron j ($j = 0, 1 \ldots, 9$) of output layer computes

$$\|z - w_j\|_2^2 = \sum_{i=1}^{84} (x_i - w_{j,i})^2,$$

with $z \in \mathbb{R}^{84}$ produced by layer F6 and $w_j = (w_{j,1}, \ldots, w_{j,84})$ weight vector for neuron j.

Output layer

Radial basis function (RBF) units. Neuron j ($j = 0, 1 \ldots, 9$) of output layer computes

$$\|z - w_j\|_2^2 = \sum_{i=1}^{84} (x_i - w_{j,i})^2,$$

with $z \in \mathbb{R}^{84}$ produced by layer F6 and $w_j = (w_{j,1}, \ldots, w_{j,84})$ weight vector for neuron j.

Gaussian connections. Assuming vectors from layer F6 are Gaussian, neuron j outputs negative log likelihood of a Gaussian distribution with mean w_j and covariance matrix I.

Output layer

Radial basis function (RBF) units. Neuron j ($j = 0, 1 \ldots, 9$) of output layer computes

$$\|z - w_j\|_2^2 = \sum_{i=1}^{84} (x_i - w_{j,i})^2,$$

with $z \in \mathbb{R}^{84}$ produced by layer F6 and $w_j = (w_{j,1}, \ldots, w_{j,84})$ weight vector for neuron j.

Gaussian connections. Assuming vectors from layer F6 are Gaussian, neuron j outputs negative log likelihood of a Gaussian distribution with mean w_j and covariance matrix I.

$\rightarrow$ each neuron outputs squared distance between its parameter vector and input.

Output layer

Radial basis function (RBF) units. Neuron j ($j = 0, 1 \ldots, 9$) of output layer computes

$$\|z - w_j\|_2^2 = \sum_{i=1}^{84} (x_i - w_{j,i})^2,$$

with $z \in \mathbb{R}^{84}$ produced by layer F6 and $w_j = (w_{j,1}, \ldots, w_{j,84})$ weight vector for neuron j.

Gaussian connections. Assuming vectors from layer F6 are Gaussian, neuron j outputs negative log likelihood of a Gaussian distribution with mean w_j and covariance matrix I.

$\rightarrow$ each neuron outputs squared distance between its parameter vector and input.

Question. How to choose $w_j \in \{-1, 1\}^{84}$?

Output layer and activation function

For $w_j \in \{-1, 1\}^{84}$, one could pick:

- pick stylized version of the image of number j (of size $7 \times 12 = 84$)
- pick other references (e.g., nicely "equidistant" to each other).

Those targets w_j are parameters of the network.

Output layer and activation function

For $w_j \in \{-1, 1\}^{84}$, one could pick:

- pick stylized version of the image of number j (of size $7 \times 12 = 84$)
- pick other references (e.g., nicely "equidistant" to each other).

Those targets w_j are parameters of the network.

Why not one-hot encoding? (LeCun, Bottou, et al. 1998) states that it does not work with more than few dozens classes (requires output units to be saturated most of the time; challenging with sigmoids, see below).

Output layer and activation function

For $w_j \in \{-1, 1\}^{84}$, one could pick:

- pick stylized version of the image of number j (of size $7 \times 12 = 84$)
- pick other references (e.g., nicely "equidistant" to each other).

Those targets w_j are parameters of the network.

Why not one-hot encoding? (LeCun, Bottou, et al. 1998) states that it does not work with more than few dozens classes (requires output units to be saturated most of the time; challenging with sigmoids, see below).

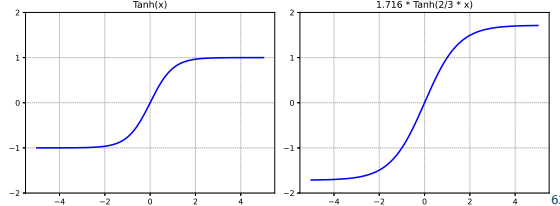
Activation function

$$\sigma(x) = A \tanh(\alpha x),$$

where $A = 1.7159$, $\alpha = 2/3$.

$\rightarrow$ **Prevent saturation** since neurons outputs belong to $\{-1, 1\}$ (normalized images)

- $\sigma(1) = 1$
- $\sigma(-1) = -1$



Optimization criterion

Let $[f_\theta(x)]_j = \|z - w_j\|_2^2$ be output of jth neuron (from output layer) and z be output of layer F6.

$\rightarrow$ loss incurred by sample (x_i, y_i) is defined as

$$\ell_i(\theta) = [f_\theta(x_i)]_{y_i} + \log\left(e^{-C} + \sum_{j=0}^{9} e^{-[f_\theta(x_i)]_j}\right),$$

where $C > 0$ is a parameter (to make sure argument of log is away from 0).

Second term is small when w_j is far away from output of layer F6.

Optimization criterion

Let $[f_\theta(x)]_j = \|z - w_j\|_2^2$ be output of jth neuron (from output layer) and z be output of layer F6.

$\rightarrow$ loss incurred by sample (x_i, y_i) is defined as

$$\ell_i(\theta) = [f_\theta(x_i)]_{y_i} + \log\left(e^{-C} + \sum_{j=0}^{9} e^{-[f_\theta(x_i)]_j}\right),$$

where $C > 0$ is a parameter (to make sure argument of log is away from 0).

Second term is small when w_j is far away from output of layer F6.

Equivalently:

$$\ell_i(\theta) = -\log\left(\frac{e^{-[f_\theta(x_i)]_{y_i}}}{e^{-C} + \sum_{j=0}^{9} e^{-[f_\theta(x_i)]_j}}\right),$$

$\rightarrow$ very close to **negative log likelihood with a softmax** output layer.

Optimization procedure

Related to stochastic gradient descent:

$$\theta_{t+1} = \theta_t - \eta(\mu I + H)^{-1}\nabla \ell_i(\theta_t)$$

where:

- ℓ_i is the loss of a single observation,
- η is the initial learning rate,
- μ a hand-picked constant, and
- $H \approx \mathrm{Diag}(\nabla^2 \ell_i(\theta_t))$ is diagonal matrix containing diagonal of Hessian of ℓ_i.

Optimization procedure

Related to stochastic gradient descent:

$$\theta_{t+1} = \theta_t - \eta(\mu I + H)^{-1}\nabla\ell_i(\theta_t)$$

where:

- ℓ_i is the loss of a single observation,
- η is the initial learning rate,
- μ a hand-picked constant, and
- $H \approx \mathrm{Diag}(\nabla^2\ell_i(\theta_t))$ is diagonal matrix containing diagonal of Hessian of ℓ_i.

Expression of $\mathrm{Diag}(\nabla^2\ell_i(\theta_t))$ quite complicated since each parameter in θ appears in different connections.

Optimization procedure

Related to stochastic gradient descent:

$$\theta_{t+1} = \theta_t - \eta(\mu I + H)^{-1}\nabla\ell_i(\theta_t)$$

where:

- ℓ_i is the loss of a single observation,
- η is the initial learning rate,
- μ a hand-picked constant, and
- $H \approx \mathrm{Diag}(\nabla^2\ell_i(\theta_t))$ is diagonal matrix containing diagonal of Hessian of ℓ_i.

Expression of $\mathrm{Diag}(\nabla^2\ell_i(\theta_t))$ quite complicated since each parameter in θ appears in different connections.

Optimization procedure

Related to stochastic gradient descent:

$$\theta_{t+1} = \theta_t - \eta(\mu I + H)^{-1}\nabla \ell_i(\theta_t)$$

where:

- ℓ_i is the loss of a single observation,
- η is the initial learning rate,
- μ a hand-picked constant, and
- $H \approx \mathrm{Diag}(\nabla^2 \ell_i(\theta_t))$ is diagonal matrix containing diagonal of Hessian of ℓ_i.

Expression of $\mathrm{Diag}(\nabla^2 \ell_i(\theta_t))$ quite complicated since each parameter in θ appears in different connections.

Approximation of each diagonal term performed at beginning of each epoch, using first 500 observations (data set has 60000 observations).

Parameters

Weight initialization: $\mathcal{U}([-2.4/F_i, 2.4/F_i])$, with F_i number of inputs of the unit.

Parameters

Weight initialization: $\mathcal{U}([-2.4/F_i, 2.4/F_i])$, with F_i number of inputs of the unit.

SGD (preconditioned)

$$\theta_{t+1} = \theta_t - \eta(\mu I + H)^{-1}\nabla \ell_i(\theta_t)$$

with $\mu = 0.02$.

Optimization lasts 20 epochs:

- $\eta = 0.0005$ for the first two epochs,
- $\eta = 0.0002$ for the next three epochs,
- $\eta = 0.0001$ for the next three epochs,
- $\eta = 0.00005$ for the next four epochs,
- $\eta = 0.00001$ for the remaining epochs.

Results

The 82 misclassified patterns by LeNet5. Below each image is displayed the correct answer (left) and the prediction (right). Errors are mostly caused by genuinely ambiguous patterns, or by under-represented writing styles in the training set.

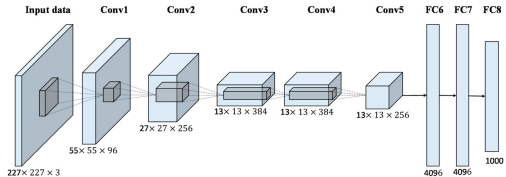
Outline

AlexNet

["Imagenet classification with deep convolutional neural networks", Krizhevsky et al. 2012]

Main ingredients:

- convolutions (e.g., first layer: 11×11 filters with $s = 4$)
- activations: ReLU (instead of tanh),
- local response normalization (step towards batch normalization),
- overlapping pooling (3×3 with stride $S = 2$ for reducing overfitting),
- dropout, data augmentation.



ReLU activations

From Krizhevsky et al. 2012, CNNs with ReLU activations enjoy much faster training timings compared to networks with tanh activations.

ReLU activations

From Krizhevsky et al. 2012, CNNs with ReLU activations enjoy much faster training timings compared to networks with tanh activations.

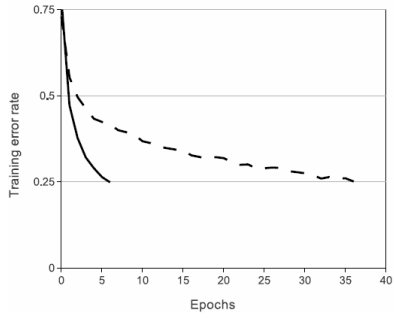


Figure: Four-layer convolutional neural network with ReLU (solid line) reaches a 25% training error rate on CIFAR-10 six times faster than equivalent network with tanh (dashed line). Learning rates for each network chosen independently to make training as fast as possible.

Main differences with LeNet

- Much larger (1000 more parameters),
- Trained on ImageNet (more data),
- GPU training (parallelism),
- Activation: ReLU instead of tanh,
- Loss: softmax + cross-entropy instead of Gaussian classifier,
- dropout + regularization + data augmentation,
- LRN normalization (historical step toward BatchNorm).

Local response normalization (LRN)/ brightness normalization

Let $a^i_{x,y}$ be the activity of a neuron from kernel i applied to position (x, y) and followed by a ReLU, and $b^i_{x,y}$ be the corresponding renormalized activity given by:

$$b^i_{x,y} = a^i_{x,y} \left(C + \alpha \sum_{j=\max(0, i-q/2)}^{\min(Q-1, i+q/2)} (a^j_{x,y})^2 \right)^{-\beta},$$

(avg over q adjacent feature maps at same spatial position, and Q total number of feature maps in layer).

Local response normalization (LRN)/ brightness normalization

Let $a^i_{x,y}$ be the activity of a neuron from kernel i applied to position (x, y) and followed by a ReLU, and $b^i_{x,y}$ be the corresponding renormalized activity given by:

$$b^i_{x,y} = a^i_{x,y} \left(C + \alpha \sum_{j=\max(0, i-q/2)}^{\min(Q-1, i+q/2)} (a^j_{x,y})^2 \right)^{-\beta},$$

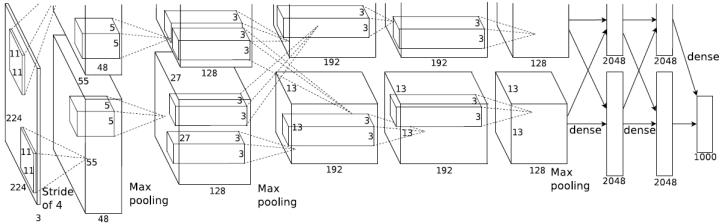
(avg over q adjacent feature maps at same spatial position, and Q total number of feature maps in layer).

Parameters (determined with validation set): $C = 2, q = 5, \alpha = 10^{-4}, \beta = 0.75$.

This renormalization creates competition between feature maps.

From ["What is the best multi-stage architecture for object recognition?", Jarrett et al. 2009]: similar normalization procedure where mean activity is substracted (local contrast normalization).

Overall architecture



- architecture split across two GPUs (with little communication with each other).
  - ReLU right after all convolutional layers and fully connected layers.
  - LRN after ReLU in the first and second convolutional layer.
  - Max-pooling after first, second and fifth convolutional layers.

Optimization

Initialization:

- Weights: $\mathcal{N}(0, 0.0001)$
- Biases of second, fourth and fifth convolutional layers and biases of fully connected layers set to 1 (seems to accelerate early stages of learning, prevent dead ReLUs). Other biases set to 0.

Optimization

Initialization:

- Weights: $\mathcal{N}(0, 0.0001)$
- Biases of second, fourth and fifth convolutional layers and biases of fully connected layers set to 1 (seems to accelerate early stages of learning, prevent dead ReLUs). Other biases set to 0.

SGD with momentum

$$v_{t+1} = 0.9\, v_t - 0.0005\, \eta\, \theta_t - \frac{\eta}{B} \sum_{i \in \mathcal{B}} \nabla \ell_i(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1},$$

with batch size $|\mathcal{B}| = B = 128$.

Second term in first update equation: ℓ_2-regularization with parameter $\lambda = 0.0005$ (weight decay).

Optimization

Initialization:

- Weights: $\mathcal{N}(0, 0.0001)$
- Biases of second, fourth and fifth convolutional layers and biases of fully connected layers set to 1 (seems to accelerate early stages of learning, prevent dead ReLUs). Other biases set to 0.

SGD with momentum

$$v_{t+1} = 0.9\, v_t - 0.0005\, \eta\, \theta_t - \frac{\eta}{B} \sum_{i \in \mathcal{B}} \nabla \ell_i(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1},$$

with batch size $|\mathcal{B}| = B = 128$.

Second term in first update equation: ℓ_2-regularization with parameter $\lambda = 0.0005$ (weight decay).

Same learning rate for all layers with following heuristic:

- initialization: $\eta = 0.01$
- divide η by 10 when validation error stops improving.
- 90 epochs on 1.2 million images: 6 days.

Numerical results

Model	Top-1 (val)	Top-5 (val)	Top-5 (test)
SIFT + FVs[7]	–	–	26.2%
1CNN	40.7%	18.2%	–
5CNNs	38.1%	16.4%	16.4%
1CNN*	39.0%	16.6%	–
7CNNs*	36.7%	15.4%	15.3%

- Error rates:
  - Top-1: is top 1 score predicting correct label?
  - Top-5: is correct label within top 5 scores?
- First line: second runner-up.
- Second and third lines: results by averaging over 1 or 5 CNN described before.
- *Last two lines correspond to networks with extra convolutional layer after last pooling layer. It has been trained on ImageNet Fall 2011 and "fine-tuned" on ImageNet 2012.

Numerical results

Model	Top-1 (val)	Top-5 (val)	Top-5 (test)
SIFT + FVs[7]	–	–	26.2%
1CNN	40.7%	18.2%	–
5CNNs	38.1%	16.4%	16.4%
1CNN*	39.0%	16.6%	–
7CNNs*	36.7%	15.4%	15.3%

- Error rates:
  - Top-1: is top 1 score predicting correct label?
  - Top-5: is correct label within top 5 scores?
- First line: second runner-up.
- Second and third lines: results by averaging over 1 or 5 CNN described before.
- *Last two lines correspond to networks with extra convolutional layer after last pooling layer. It has been trained on ImageNet Fall 2011 and "fine-tuned" on ImageNet 2012.

AlexNet has very similar architecture to LeNet, but

- deeper, bigger, ($\sim 60k$ vs. $60000k$ parameters),
- ReLUs,
- stacked convolutional layers without intermediate pooling layers.

Results

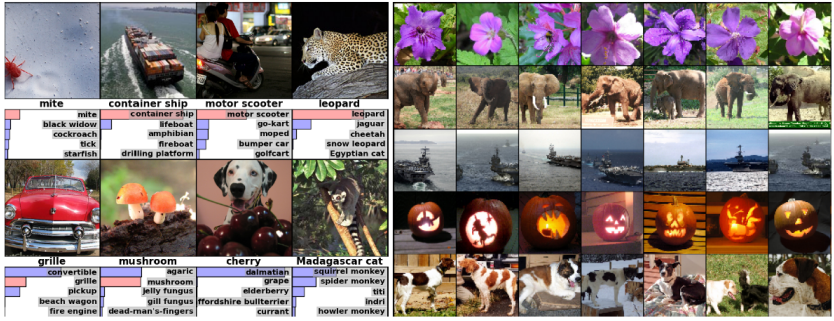
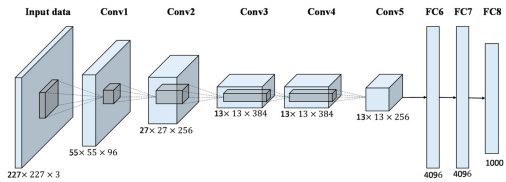


Figure 4: **(Left)** Eight ILSVRC-2010 test images and the five labels considered most probable by our model. The correct label is written under each image, and the probability assigned to the correct label is also shown with a red bar (if it happens to be in the top 5). **(Right)** Five ILSVRC-2010 test images in the first column. The remaining columns show the six training images that produce feature vectors in the last hidden layer with the smallest Euclidean distance from the feature vector for the test image.

Outline

ZFNet improvements upon AlexNet

["Visualizing and understanding convolutional networks", Zeiler and Fergus 2014]



Aim: interpret **different feature maps** (what are they after?) to obtain better tuning of network architecture.

In ZFNet: feature maps not divided across GPUs. Connections between layers are **less sparse** than in AlexNet.

Deconvnet

Find pixels maximizing activation of given feature map.

Deconvnet

Find pixels maximizing activation of given feature map.

How? Invert network.

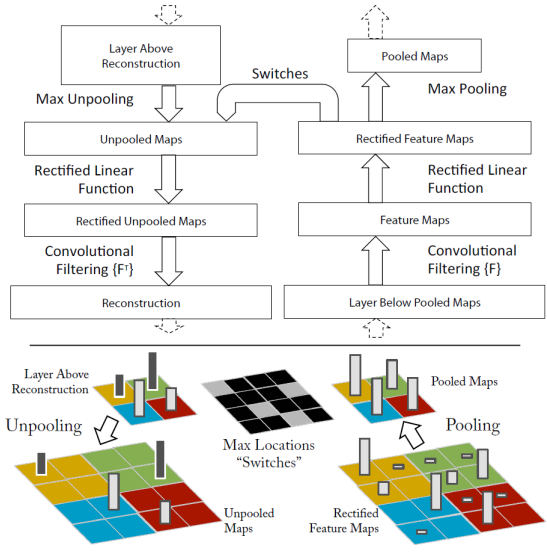
Deconvnet

Find pixels maximizing activation of given feature map.

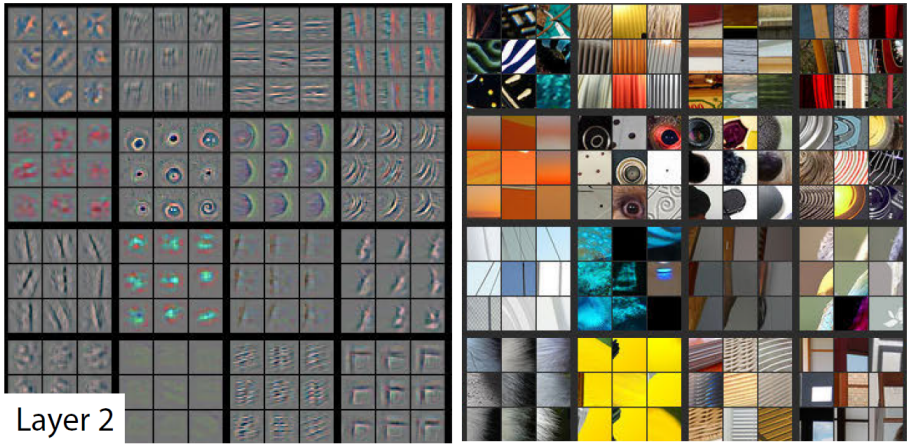
How? Invert network.

More precisely:

- pick a layer,
- pick a feature map,
- run network (forward) on a validation set,
- choose an image maximizing activation of feature map,
- "backpropagate" this activation to obtain a stylized image in original pixel space.

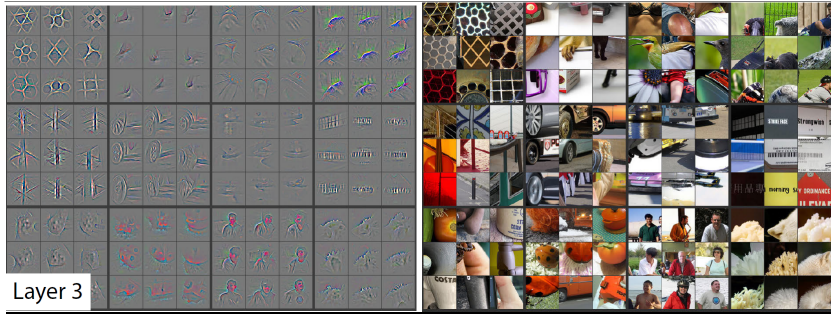


Results

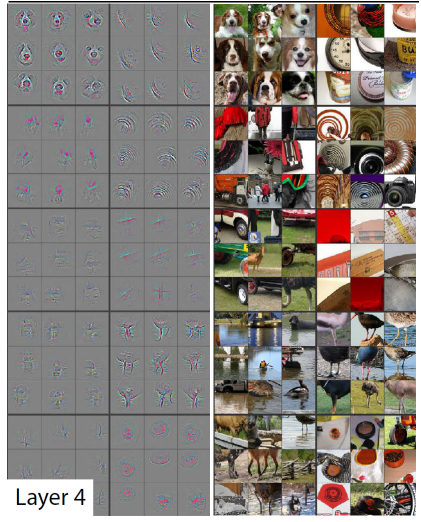


Top 9 activations in a random subset of feature maps across validation data, projected down to pixel space using previous deconvolutional network approach.

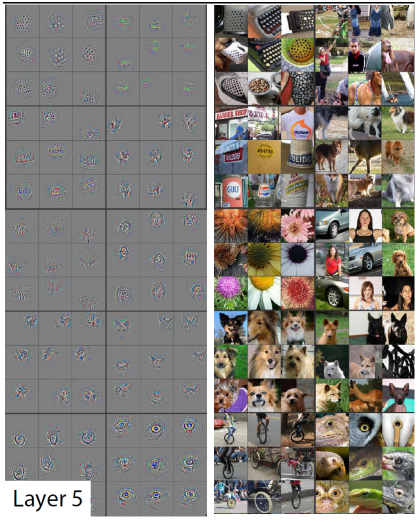
Results



Results



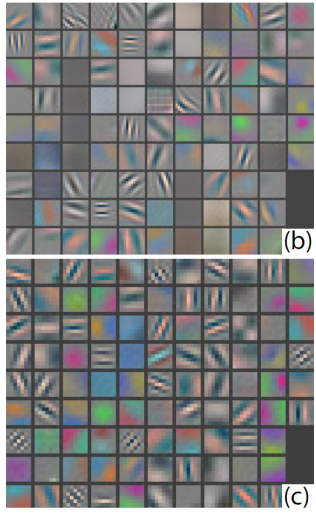
Results



Remarks

- strong groups within each feature map,
- greater invariance at higher layers
- exaggeration of discriminative parts of image, e.g. eyes and noses of dogs (layer 4, row 1, cols 1).

Visualization of previous modifications

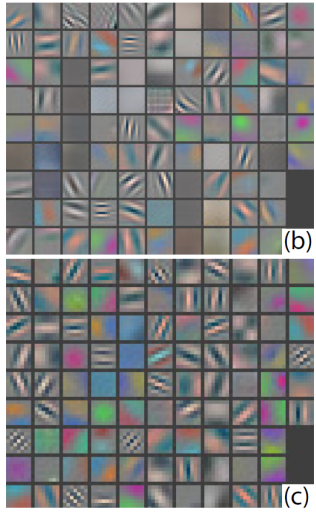


(b): 1st layer features from AlexNet Krizhevsky et al. 2012.

(c): 1st layer features of ZFNet.

Source: ["Visualizing and understanding convolutional networks", Zeiler and Fergus 2014].

Visualization of previous modifications



(b): 1st layer features from AlexNet Krizhevsky et al. 2012.

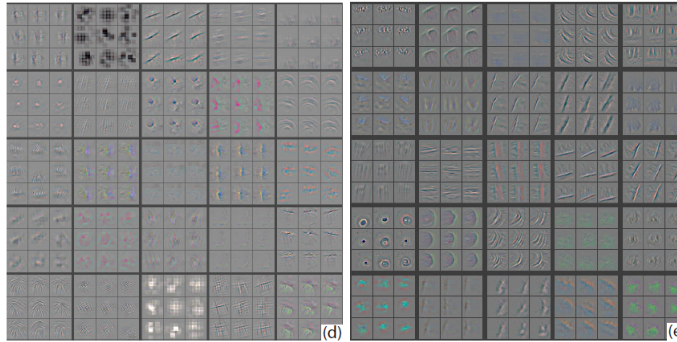
(c): 1st layer features of ZFNet.

Differences: smaller stride (2 vs 4) and filter size (7x7 vs 11x11)

Results in **more distinctive features** and **fewer dead features**.

Source: ["Visualizing and understanding convolutional networks", Zeiler and Fergus 2014].

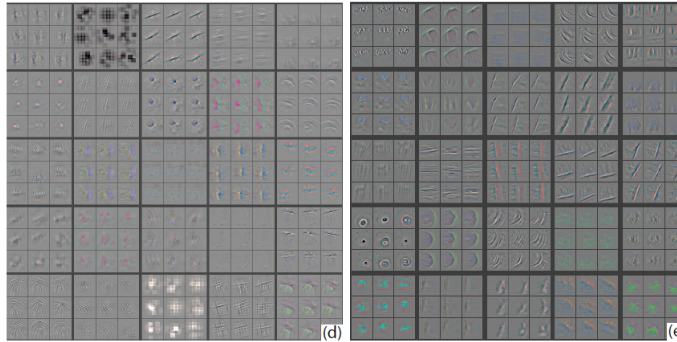
Visualization of previous modifications



Source: ["Visualizing and understanding convolutional networks", Zeiler and Fergus 2014].

(d): Visualizations of 2nd layer from Krizhevsky et al. 2012; (e): Visualizations of 2nd layer of ZFNet.

Visualization of previous modifications



Source: ["Visualizing and understanding convolutional networks", Zeiler and Fergus 2014].

(d): Visualizations of 2nd layer from Krizhevsky et al. 2012; (e): Visualizations of 2nd layer of ZFNet.

Feature maps in (e) are **cleaner**, with **no aliasing artefacts** visible in (d).

Conclusion for AlexNet

- First layer filters mix high and low frequency information. Little coverage of mid freq.
  $\rightarrow$ Reduced first layer filter size from 11×11 to 7×7.
- Aliasing artifacts are present in second layer because of large stride of 4 used in first convolutional layer.
  $\rightarrow$ change stride from 4 to 2.

With these modifications:

- winner of *ImageNet Large Scale Visual Recognition Challenge* (ILSVRC) 2013
- improvement upon AlexNet by
  - expanding size of middle convolutional layers,
  - smaller stride and filter sizes on first layer.

A few more notes:

- Visualization details (incl. implementation): `https://hackmd.io/@machine-learning/ByaTE80BI`.
- Many other visualization techniques! Examples:
  `https://github.com/utkuozbulak/pytorch-cnn-visualizations/tree/master`

ZFNet final structure

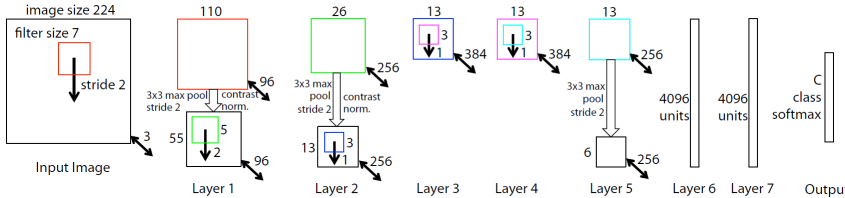


Figure 3. Architecture of our 8 layer convnet model. A 224 by 224 crop of an image (with 3 color planes) is presented as the input. This is convolved with 96 different 1st layer filters (red), each of size 7 by 7, using a stride of 2 in both x and y. The resulting feature maps are then: (i) passed through a rectified linear function (not shown), (ii) pooled (max within 3x3 regions, using stride 2) and (iii) contrast normalized across feature maps to give 96 different 55 by 55 element feature maps. Similar operations are repeated in layers 2,3,4,5. The last two layers are fully connected, taking features from the top convolutional layer as input in vector form ($6 \cdot 6 \cdot 256 = 9216$ dimensions). The final layer is a C-way softmax function, C being the number of classes. All filters and feature maps are square in shape.

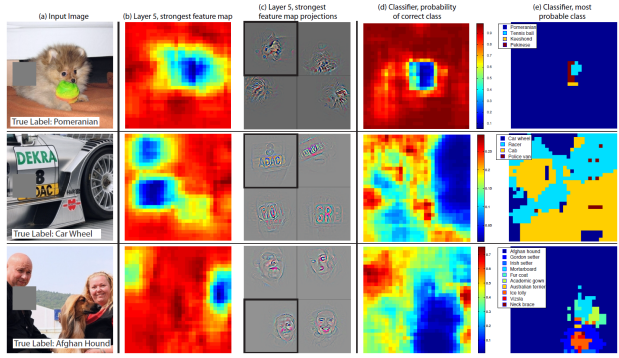
Source: ["Visualizing and understanding convolutional networks", Zeiler and Fergus 2014].

Results in classification

Error %	Val Top-1	Val Top-5	Test Top-5
(Gunji et al., 2012)	–	–	26.2
(Krizhevsky et al., 2012), 1 convnet	40.7	18.2	––
(Krizhevsky et al., 2012), 5 convnets	38.1	16.4	16.4
(Krizhevsky et al., 2012)*, 1 convnets	39.0	16.6	––
(Krizhevsky et al., 2012)*, 7 convnets	36.7	15.4	15.3
Our replication of (Krizhevsky et al., 2012), 1 convnet	40.5	18.1	––
1 convnet as per Fig. 3	38.4	16.5	––
5 convnets as per Fig. 3 - (a)	36.7	15.3	15.3
1 convnet as per Fig. 3 but with layers 3, 4, 5: 512,1024,512 maps - (b)	37.5	16.0	16.1
6 convnets, (a) & (b) combined	**36.0**	**14.7**	**14.8**

Source: ["Visualizing and understanding convolutional networks", Zeiler and Fergus 2014].

Occlusion



Source: ["Visualizing and understanding convolutional networks", Zeiler and Fergus 2014].

Three test examples where we systematically cover up different portions of scene with a gray square (1st column) and see how top (layer 5) feature maps ((b) & (c)) and classifier output ((d) & (e)) changes.

(b): for each position of gray scale, we record total activation in one layer feature map (the one with strongest response in unoccluded image).

(c): a visualization of this feature map projected down into input image (black square), along with visualizations of this map from other images. First row example shows strongest feature to be dog's face. When this is covered-up, activity in feature map decreases (blue area in (b)).

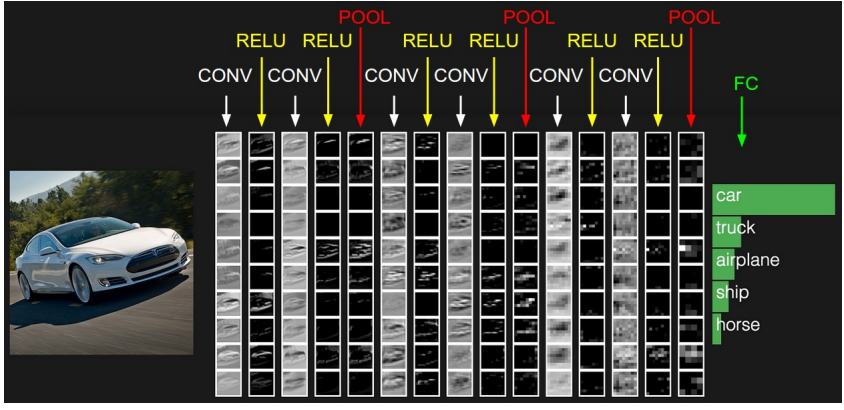
(d): a map of correct class probability, as a function of position of gray square. E.g. when dog's face is obscured, probability for pomeranian drops significantly.

(e): most probable label as a function of occluder position.

Outline

Tiny VGGnet

["Very deep convolutional networks for large-scale image recognition", Simonyan and Zisserman 2014]



Network features

Convolutional layers:

- Small receptive field (spatial extent): 3×3
  smallest kernel capturing local spatial interactions while allowing deeper compositions that efficiently increase the effective receptive field.
- stride of 1
- spatial resolution preserved after convolution.

Max-pooling layers:

- 2×2 kernels,
- stride of 2.

All hidden layers use ReLU activations.

LRN layers do not improve performance.

Network features

Stacking 3 convolutional layers with receptive fields 3×3, corresponds to single convolutional layer with receptive field 7×7 ("connects the same inputs"). Interesting?

1. Stack of 3 convolutional layers of size 3×3, number of parameters: $3 \times 3 \times 3 = 27$.
2. One standard convolutional layer of size 7×7: results in 49 parameters.

(to be multiplied by depth/number of channels)

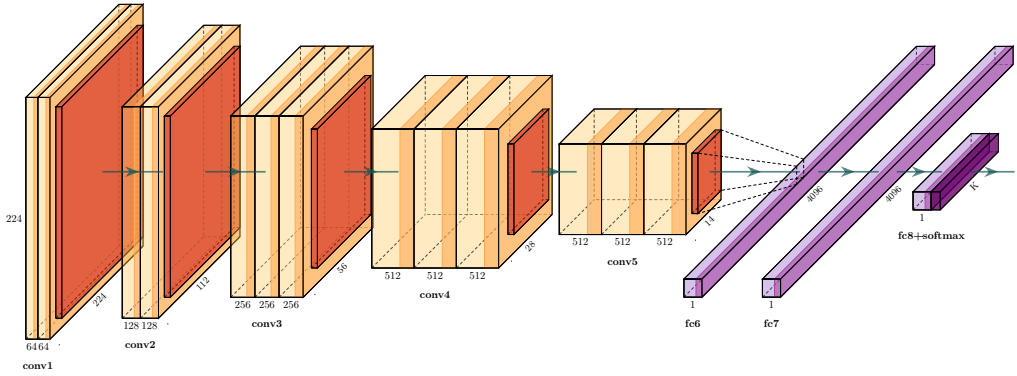
$\rightarrow$ VGGNet simplifies AlexNet's architecture (more systematic layers).

VGGNets

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv(receptive field size)-(number of channels)". The ReLU activation function is not shown for brevity.

VGGNet-16



Source: https://github.com/HarisIqbal88/PlotNeuralNet/tree/master

Hyperparameters

Initialization:

- Network A: $\mathcal{N}(0, 0.01)$ for weights and 0 for biases.
- For other networks: first four conv layers and last three fully connected layers initialized using network A. Remaining layers initialized randomly.

Hyperparameters

Initialization:

- Network A: $\mathcal{N}(0, 0.01)$ for weights and 0 for biases.
- For other networks: first four conv layers and last three fully connected layers initialized using network A. Remaining layers initialized randomly.

SGD with momentum

$$v_{t+1} = 0.9\, v_t - 0.0005\, \eta\, \theta_t - \frac{\eta}{B} \sum_{i \in \mathcal{B}} \nabla \ell_i(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1},$$

with batch size $B = 128$.

Hyperparameters

Initialization:

- Network A: $\mathcal{N}(0, 0.01)$ for weights and 0 for biases.
- For other networks: first four conv layers and last three fully connected layers initialized using network A. Remaining layers initialized randomly.

SGD with momentum

$$v_{t+1} = 0.9\, v_t - 0.0005\, \eta\, \theta_t - \frac{\eta}{B} \sum_{i \in \mathcal{B}} \nabla \ell_i(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1},$$

with batch size $B = 128$.

Same learning rate accross all layers with following heuristic:

- initialization: $\eta = 0.01$
- divide η by 10 when validation error stop improving (done three times here).
- 74 epochs.
- ℓ_2 penalty with hyperparameter 5.10^{-4}
- Dropout regularization for first two fully connected layers (probability $p = 0.5$)

Results

Method	top-1 val. error (%)	top-5 val. error (%)	top-5 test error (%)
VGG (2 nets, multi-crop & dense eval.)	23.7	6.8	6.8
VGG (1 net, multi-crop & dense eval.)	24.4	7.1	7.0
VGG (ILSVRC submission, 7 nets, dense eval.)	24.7	7.5	7.3
GoogLeNet (Szegedy et al., 2014) (1 net)	—		7.9
GoogLeNet (Szegedy et al., 2014)(7 nets)	—		6.7
MSRA (He et al, 2014)(11 nets)	—	—	8.1
MSRA (He et al., 2014)(1 net)	27.9	9.1	9.1
Clarifai (Russakovsky et al., 2014) (multiple nets)	—	—	11.7
Clarifai (Russakovsky et al., 2014)(1 net)	—	—	12.5
Zeiler & Fergus (Zeiler & Fergus, 2013) (6 nets)	36.0	14.7	14.8
Zeiler & Fergus (Zeiler & Fergus, 2013)(1 net)	37.5	16.0	16.1
OverFeat (Sermanet et al, 2014) (7 nets)	34.0	13.2	13.6
OverFeat (Sermanet et al, 2014) (1 net)	35.7	14.2	—
Krizhevsky et al. (Krizhevsky et al., 2012)(5 nets)	38.1	16.4	16.4
Krizhevsky et al. (Krizhevsky et al., 2012) (1 net)	40.7	18.2	—

Source: ["Very deep convolutional networks for large-scale image recognition", Simonyan and Zisserman 2014].

Downsides of VGGNet: more expensive to evaluate, requires a lot more memory and parameters (140M).
Most parameters are in first fully connected layer. It was since found that such FC layers can be removed with no performance downgrade $\rightarrow$ significant reduction of number of necessary parameters.

Outline

GoogLeNet

["Going deeper with convolutions", Szegedy et al. 2015]

Target. Increase depth and width of state-of-the-art CNNs while keeping **small number of parameters**:

- to approximate more complicated functions
- while being robust to overfitting, and computationally more tractable.

GoogLeNet

["Going deeper with convolutions", Szegedy et al. 2015]

Target. Increase depth and width of state-of-the-art CNNs while keeping **small number of parameters**:

- to approximate more complicated functions
- while being robust to overfitting, and computationally more tractable.

How? Use 1×1 convolution layers to reduce number of parameters. Then, apply filters of different sizes 3×3, 5×5 and 3×3 max pooling (on each feature maps).

GoogLeNet

["Going deeper with convolutions", Szegedy et al. 2015]

Target. Increase depth and width of state-of-the-art CNNs while keeping **small number of parameters**:

- to approximate more complicated functions
- while being robust to overfitting, and computationally more tractable.

How? Use 1×1 convolution layers to reduce number of parameters. Then, apply filters of different sizes 3×3, 5×5 and 3×3 max pooling (on each feature maps).

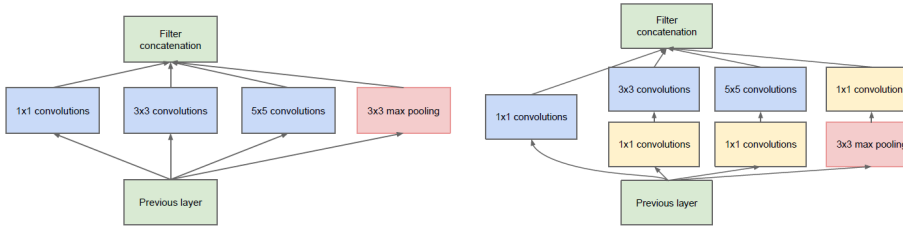
Details.

- All convolution layers use ReLU activations.
- Same spatial resolution for each feature map.

GoogLeNet - inception module

Same spatial resolution for each feature map.

Use 1×1 convolution layers to reduce number of parameters. Then, apply filters of different sizes 3×3, 5×5 and 3×3 max pooling (on each feature maps).



(a) Inception module, naïve version (b) Inception module with dimension reductions

Figure 2: Inception module

Source: ["Going deeper with convolutions", Szegedy et al. 2015].

GoogLeNet - inception module

type	patch size/ stride	output size	depth	#1×1	#3×3 reduce	#3×3	#5×5 reduce	#5×5	pool proj	params	ops
convolution	7×7/2	112×112×64	1							2.7K	34M
max pool	3×3/2	56×56×64	0								
convolution	3×3/1	56×56×192	2		64	192				112K	360M
max pool	3×3/2	28×28×192	0								
inception (3a)		28×28×256	2	64	96	128	16	32	32	159K	128M
inception (3b)		28×28×480	2	128	128	192	32	96	64	380K	304M
max pool	3×3/2	14×14×480	0								
inception (4a)		14×14×512	2	192	96	208	16	48	64	364K	73M
inception (4b)		14×14×512	2	160	112	224	24	64	64	437K	88M
inception (4c)		14×14×512	2	128	128	256	24	64	64	463K	100M
inception (4d)		14×14×528	2	112	144	288	32	64	64	580K	119M
inception (4e)		14×14×832	2	256	160	320	32	128	128	840K	170M
max pool	3×3/2	7×7×832	0								
inception (5a)		7×7×832	2	256	160	320	32	128	128	1072K	54M
inception (5b)		7×7×1024	2	384	192	384	48	128	128	1388K	71M
avg pool	7×7/1	1×1×1024	0								
dropout (40%)		1×1×1024	0								
linear		1×1×1000	1							1000K	1M
softmax		1×1×1000	0								

Table 1: GoogLeNet incarnation of the Inception architecture

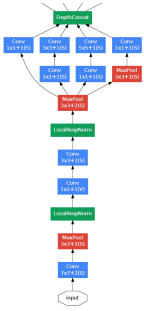
Source: ["Going deeper with convolutions", Szegedy et al. 2015].

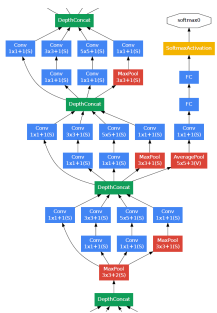
"3x3 reduce" and "5x5 reduce" stands for number of 1x1 filters in reduction layer used before 3x3 and 5x5 convolutions.

Pool proj column shows number of 1x1 filters in projection layer after built-in max-pooling.

Structure of GoogLeNet

Structure of GoogLeNet





A few of GoogLeNet's tricks

Additional auxiliary classifiers used to backpropagate gradients.

During training: loss of auxiliary classifiers is weighted by 0.3 and added to total loss of network. Auxiliary networks are removed for inference.

A few of GoogLeNet's tricks

Additional auxiliary classifiers used to backpropagate gradients.

During training: loss of auxiliary classifiers is weighted by 0.3 and added to total loss of network. Auxiliary networks are removed for inference.

Auxiliary network put after $(4a)$ and $(4d)$:

- average pooling layer 5×5, stride of 3,
- 1×1 convolution with 128 filters, with ReLU.
- fully connected layer with 1024 neurons and ReLU,
- dropout layer with a dropout ratio of 70%,
- linear layer with softmax loss, predicting same 1000 classes as main classifier.

Parameters

Initialization:

- weights drawn from $\mathcal{N}(0, 1)$ and biases set to 0.

["Deep learning via Hessian-free optimization.", Martens 2010]

[1] ["On the importance of initialization and momentum in deep learning", Sutskever et al. 2013]

Parameters

Initialization:

- weights drawn from $\mathcal{N}(0, 1)$ and biases set to 0.

["Deep learning via Hessian-free optimization.", Martens 2010]

SGD with momentum

$$v_{t+1} = \mu_t v_t - \frac{\eta}{B} \sum_{i \in \mathcal{B}} \nabla \ell_i(\theta_t + \mu_t v_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1},$$

with $B = 200$, $\mu_t = \min(1 - 2^{-1-\log_2(\lfloor t/250 \rfloor + 1)}, \mu_{max})$, $\mu_{max} \in \{0, 0.9, 0.99, 0.995, 0.999\}$.

[1] ["On the importance of initialization and momentum in deep learning", Sutskever et al. 2013]

Parameters

Initialization:

- weights drawn from $\mathcal{N}(0, 1)$ and biases set to 0.

["Deep learning via Hessian-free optimization.", Martens 2010]

SGD with momentum

$$v_{t+1} = \mu_t v_t - \frac{\eta}{B} \sum_{i \in \mathcal{B}} \nabla \ell_i(\theta_t + \mu_t v_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1},$$

with $B = 200$, $\mu_t = \min(1 - 2^{-1-\log_2(\lfloor t/250 \rfloor + 1)}, \mu_{max})$, $\mu_{max} \in \{0, 0.9, 0.99, 0.995, 0.999\}$.

Same learning rate accross layers with heuristic:[1]

- initialization: $\eta = 0.01$,
- multiply η by 0.96 every 8 epochs,
- training lasts 125 epochs.

[1] ["On the importance of initialization and momentum in deep learning", Sutskever et al. 2013]

Results

- Polyak averaging used to create final model for inference.
- 7 different versions of GoogleNet were trained and aggregated for predictions.

Team	Year	Place	Error (top-5)	Uses external data
SuperVision	2012	1st	16.4%	no
SuperVision	2012	1st	15.3%	Imagenet 22k
Clarifai	2013	1st	11.7%	no
Clarifai	2013	1st	11.2%	Imagenet 22k
MSRA	2014	3rd	7.35%	no
VGG	2014	2nd	7.32%	no
GoogLeNet	2014	1st	6.67%	no

Table 2: Classification performance

Number of models	Number of Crops	Cost	Top-5 error	compared to base
1	1	1	10.07%	base
1	10	10	9.15%	-0.92%
1	144	144	7.89%	-2.18%
7	1	7	8.09%	-1.98%
7	10	70	7.62%	-2.45%
7	144	1008	6.67%	-3.45%

Main contribution: “inception module” dramatically reducing number of parameters in network (4M, AlexNet has 60M).

Outline

ResNet (2016)

["Deep residual learning for image recognition", He et al. 2016]

Statement: optimization can be hard for some deep networks.

Proposed fix: ease optimization via simple paths in network $\rightarrow$ "skip connections" / "shortcut connections".

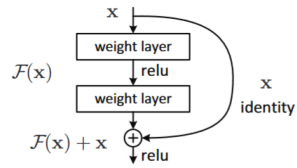


Figure 2. Residual learning: a building block.

$\rightarrow$ no extra parameters, no additional computational burden.

Literature on shortcut connections

Early practice in MLP: add linear layer between inputs and outputs.

[*Pattern recognition and neural networks*, Ripley 2007]

Literature on shortcut connections

Early practice in MLP: add linear layer between inputs and outputs.

[*Pattern recognition and neural networks*, Ripley 2007]

Few intermediate classifiers can also be added to ease optimization:

- ["Going deeper with convolutions", Szegedy et al. 2015],
- ["Deeply-supervised nets", Lee et al. 2015].

Literature on shortcut connections

Early practice in MLP: add linear layer between inputs and outputs.

[*Pattern recognition and neural networks*, Ripley 2007]

Few intermediate classifiers can also be added to ease optimization:

- ["Going deeper with convolutions", Szegedy et al. 2015],
- ["Deeply-supervised nets", Lee et al. 2015].

Highway networks: shortcut connections with gating functions. Gates are data dependent and have parameters.

- ["Highway networks", Rupesh Kumar Srivastava et al. 2015],
- ["Training very deep networks", Rupesh K Srivastava et al. 2015].

Residual Networks: General Idea

Motivation: very deep networks become harder to train (degradation problem: deeper $\neq$ better training error).

Key idea: learn a residual correction instead of a full mapping

$$\mathbf{y} = f(\mathbf{x}, \mathbf{W}_i) + \mathbf{x},$$

where:

- $\mathbf{x}$: input of a block,
- $f(\mathbf{x}, \mathbf{W}_i)$: residual function (stack of layers),
- shortcut connection adds the identity mapping.

Residual Networks: General Idea

Motivation: very deep networks become harder to train (degradation problem: deeper $\neq$ better training error).

Key idea: learn a residual correction instead of a full mapping

$$\mathbf{y} = f(\mathbf{x}, \mathbf{W}_i) + \mathbf{x},$$

where:

- $\mathbf{x}$: input of a block,
- $f(\mathbf{x}, \mathbf{W}_i)$: residual function (stack of layers),
- shortcut connection adds the identity mapping.

Architecture design (inherited from VGG):

- same number of filters while feature map size is unchanged,
- when spatial resolution is halved $\Rightarrow$ number of filters doubled.

Residual Networks: General Idea

Motivation: very deep networks become harder to train (degradation problem: deeper $\neq$ better training error).

Key idea: learn a residual correction instead of a full mapping

$$\mathbf{y} = f(\mathbf{x}, \mathbf{W}_i) + \mathbf{x},$$

where:

- $\mathbf{x}$: input of a block,
- $f(\mathbf{x}, \mathbf{W}_i)$: residual function (stack of layers),
- shortcut connection adds the identity mapping.

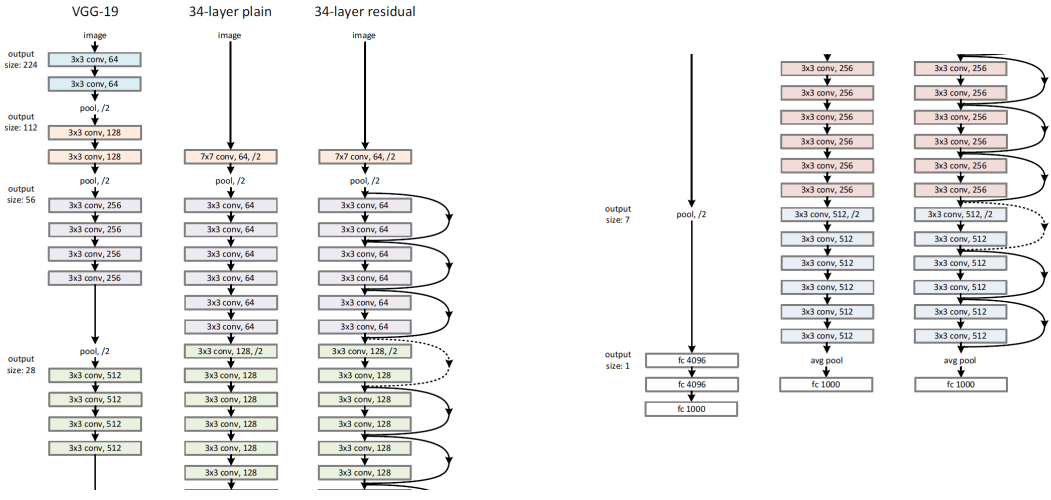
Architecture design (inherited from VGG):

- same number of filters while feature map size is unchanged,
- when spatial resolution is halved $\Rightarrow$ number of filters doubled.

Dimension mismatch between x and y, use tricks (padding, projections, etc.) to match dimensions.

Structure of ResNet

Structure of ResNet



Parameters

Initialization, as in He et al. 2015: weights are drawn from $\mathcal{N}(0, 2/n_L)$ (n_L is number of neurons in previous layer); biases are set to 0.

Parameters

Initialization, as in He et al. 2015: weights are drawn from $\mathcal{N}(0, 2/n_L)$ (n_L is number of neurons in previous layer); biases are set to 0.

SGD with momentum

$$v_{t+1} = 0.9\, v_t - 0.0001\, \eta\, \theta_t - \frac{\eta}{B} \sum_{i \in \mathcal{B}} \nabla \ell_i(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$

with batch size $B = 256$.

Parameters

Initialization, as in He et al. 2015: weights are drawn from $\mathcal{N}(0, 2/n_L)$ (n_L is number of neurons in previous layer); biases are set to 0.

SGD with momentum

$$v_{t+1} = 0.9\, v_t - 0.0001\, \eta\, \theta_t - \frac{\eta}{B} \sum_{i \in \mathcal{B}} \nabla \ell_i(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$

with batch size $B = 256$.

Same learning rate accross layers with heuristic:

- initialization: $\eta = 0.1$,
- divide η by 10 when validation error stop improving,
- 120 epochs.

Parameters

Initialization, as in He et al. 2015: weights are drawn from $\mathcal{N}(0, 2/n_L)$ (n_L is number of neurons in previous layer); biases are set to 0.

SGD with momentum

$$v_{t+1} = 0.9\, v_t - 0.0001\, \eta\, \theta_t - \frac{\eta}{B} \sum_{i \in \mathcal{B}} \nabla \ell_i(\theta_t)$$

$$\theta_{t+1} = \theta_t + v_{t+1}$$

with batch size $B = 256$.

Same learning rate accross layers with heuristic:

- initialization: $\eta = 0.1$,
- divide η by 10 when validation error stop improving,
- 120 epochs.

Miscellaneous:

- batch normalization after each convolution and before activation,
- no dropout.

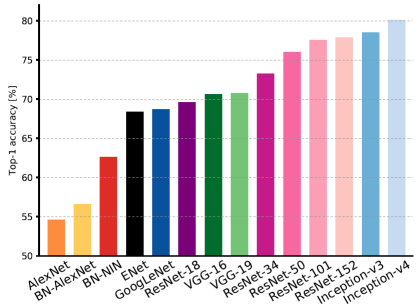
Results

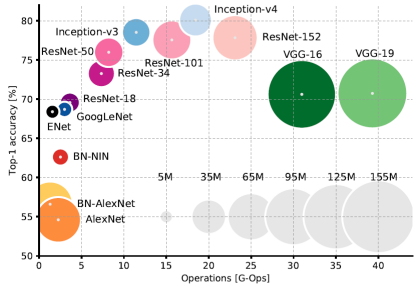
method	top-5 err. (**test**)
VGG [41] (ILSVRC'14)	7.32
GoogLeNet [44] (ILSVRC'14)	6.66
VGG [41] (v5)	6.8
PReLU-net [13]	4.94
BN-inception [16]	4.82
ResNet (ILSVRC'15)	**3.57**

- Winner of ImageNet Large Scale Visual Recognition Challenge (ILSVRC) 2015,
- special skip connections and heavy use of batch normalization,
- no fully-connected layers (beyond output layer) at end of network.

Outline

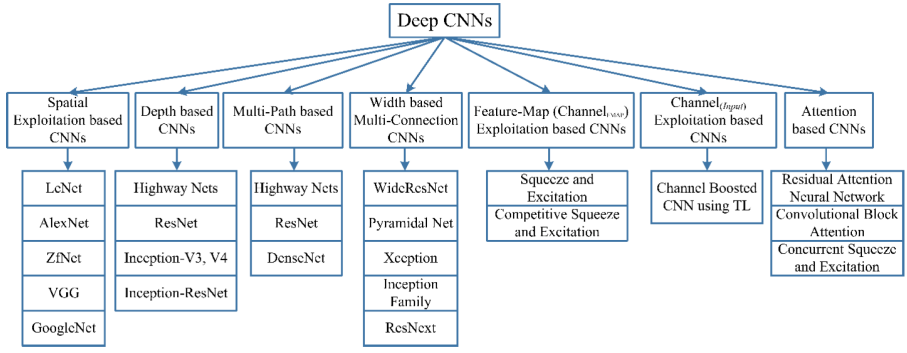
Number of parameters and performance





Source: ["An analysis of deep neural network models for practical applications", Canziani et al. 2016] (check it here).

CNN Taxonomy



See detailed review ["A survey of the recent architectures of deep convolutional neural networks", Khan et al. 2020].

Outline

Poll!

https://app.wooclap.com/CGYRYZ?from=instruction-slide

End words

Main ingredients:

- Weight sharing, backprop with weight sharing,
- filters (kernels): structured weight sharing for images,
- pooling layers (summary statistics),
- classical architectures for CNNs.

A few playgrounds:

- Neural nets: http://playground.tensorflow.org.
- Filters/kernels: https://setosa.io.
- Visualize CNNs: https://poloclub.github.io.

References

[Bis95] Chris M Bishop. "Training with noise is equivalent to Tikhonov regularization". In: *Neural computation* 7.1 (1995), pp. 108–116.

[Bre00] Leo Breiman. "Randomizing outputs to increase prediction accuracy". In: *Machine Learning* 40.3 (2000), pp. 229–242.

[CPC16] Alfredo Canziani, Adam Paszke, and Eugenio Culurciello. "An analysis of deep neural network models for practical applications". In: *arXiv preprint arXiv:1605.07678* (2016).

[DV16] Vincent Dumoulin and Francesco Visin. "A guide to convolution arithmetic for deep learning". In: *ArXiv e-prints* (2016). eprint: 1603.07285.

[DYH09] Li Deng, Dong Yu, and G Hinton. "Deep learning for speech recognition and related applications". In: *NIPS workshop*. 2009.

[EF15] David Eigen and Rob Fergus. "Predicting depth, surface normals and semantic labels with a common multi-scale convolutional architecture". In: *Proceedings of the IEEE International Conference on Computer Vision*. 2015, pp. 2650–2658.

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

References

[Gra11] Alex Graves. "Practical variational inference for neural networks". In: *Advances in Neural Information Processing Systems*. 2011, pp. 2348–2356.

[GSS14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. "Explaining and harnessing adversarial examples". In: *arXiv preprint arXiv:1412.6572* (2014).

[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.

[He+16] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[HOT06] Geoffrey E Hinton, Simon Osindero, and Yee-Whye Teh. "A fast learning algorithm for deep belief nets". In: *Neural computation* 18.7 (2006), pp. 1527–1554.

[JGH96] Kam-Chuen Jim, C Lee Giles, and Bill G Horne. "An analysis of noise in recurrent neural networks: convergence and generalization". In: *IEEE Transactions on neural networks* 7.6 (1996), pp. 1424–1438.

References

[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.

[Kha+20] Asifullah Khan et al. "A survey of the recent architectures of deep convolutional neural networks". In: *Artificial Intelligence Review* 53.8 (2020), pp. 5455–5516.

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "Imagenet classification with deep convolutional neural networks". In: *Advances in neural information processing systems*. 2012, pp. 1097–1105.

[LeC+89a] Yann LeCun et al. "Generalization and network design strategies". In: *Connectionism in perspective* (1989), pp. 143–155.

[LeC+89b] Yann LeCun, Bernhard Boser, et al. "Backpropagation applied to handwritten zip code recognition". In: *Neural computation* 1.4 (1989), pp. 541–551.

[LeC+98] Yann LeCun, Léon Bottou, et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324.

References

[Lee+15] Chen-Yu Lee et al. “Deeply-supervised nets”. In: *Artificial Intelligence and Statistics*. 2015, pp. 562–570.

[Mar10] James Martens. “Deep learning via Hessian-free optimization.”. In: *ICML*. Vol. 27. 2010, pp. 735–742.

[Pau+14] Mattis Paulin et al. “Transformation pursuit for image classification”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2014, pp. 3646–3653.

[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.

[Rif+11] Salah Rifai et al. “Adding noise to the input of a model trained with a regularized objective”. In: *arXiv preprint arXiv:1104.3250* (2011).

[Rip07] Brian D Ripley. *Pattern recognition and neural networks*. Cambridge university press, 2007.

[Rus+15] Olga Russakovsky et al. “Imagenet large scale visual recognition challenge”. In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252.

References

[SB17] Justin Salamon and Juan Pablo Bello. "Deep convolutional neural networks and data augmentation for environmental sound classification". In: *IEEE Signal Processing Letters* 24.3 (2017), pp. 279–283.

[SGS15a] Rupesh K Srivastava, Klaus Greff, and Jürgen Schmidhuber. "Training very deep networks". In: *Advances in neural information processing systems*. 2015, pp. 2377–2385.

[SGS15b] Rupesh Kumar Srivastava, Klaus Greff, and Jürgen Schmidhuber. "Highway networks". In: *arXiv preprint arXiv:1505.00387* (2015).

[Spr+14] Jost Tobias Springenberg et al. "Striving for simplicity: The all convolutional net". In: *arXiv preprint arXiv:1412.6806* (2014).

[Sut+13] Ilya Sutskever et al. "On the importance of initialization and momentum in deep learning". In: *International conference on machine learning*. 2013.

[SZ14] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014).

[Sze+15] Christian Szegedy et al. "Going deeper with convolutions". In: Cvpr. 2015.

References

[XT15] Saining Xie and Zhuowen Tu. “Holistically-nested edge detection”. In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1395–1403.

[YP15] Heng Yang and Ioannis Patras. “Mirror, mirror on the wall, tell me, is the error small?” In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2015, pp. 4685–4693.

[ZF14] Matthew D Zeiler and Rob Fergus. “Visualizing and understanding convolutional networks”. In: *European conference on computer vision*. Springer. 2014, pp. 818–833.