Neural network architectures

Arto Klami

2 April, 2025

Where are we?

- We now how to solve all kinds of problems with supervised DL
- Quite a bit more complex in practice: See e.g. `https://karpathy.github.io/2019/04/25/recipe/` for things you need to pay attention to
- ...but we haven't talked about the network architecture at all
- Today we discuss (on rather high level)
  1. Convolutional and recurrent architectures
  2. Construction of larger (convolutional) networks
  3. Residual connections

NNs with fully connected layers

- The basic architecture of $\mathbf{g}(\boldsymbol{x}) = \psi_K(\boldsymbol{W}_K(\psi_{K-1}(\boldsymbol{W}_{K-1} \dots \psi_1(\boldsymbol{W}_1 \boldsymbol{x})))$ is universally capable, but really naive
- Between two layers of M units we have M^2 parameters, so large networks result in really high number of parameters and we e.g. quickly run out of memory
- Fully connected layer from 25 megapixel RGB image to a hidden layer of similar size would have more than $5e^{15}$ parameters

NNs with fully connected layers

- Fully connected NNs solve simple prediction tasks (low input and output dimensionality, not much known about data) well
- So do quite a few other models as well: Random forests, gradient boosting, ...
- NNs likely not the best models for simple low-dimensional problems
- ...but not terribly bad either, and you get to use the same tools you need to master for more complex problems
- Fully connected feedforward NNs almost never used to solve problems of practical interest, but we routinely use them as parts in bigger models

Parameter sharing and sparse connectivity

- If we want lots of neurons (=large network) but less weights, we need
  - *Sparse connectivity*: Neurons only connect to a small subset of previous layer nodes, $\boldsymbol{W}_{ij} = 0$ for (many) pairs of i and j
  - *Parameter sharing*: Not every weight is unique, but instead we use the same weight for multiple connections, $\boldsymbol{W}_{ij} = \boldsymbol{W}_{kl}$
- Could be achieved by regularization that encourages sparse $\boldsymbol{W}$ and that pools weights towards each other, but in practice we would rather design the architecture to satisfy this be construction
- Most commonly used in context of *convolutional neural networks* (today) and *transformers* (next lecture)

See: Math exercise

Convolution

- Instead of arbitrary vectors, we consider our inputs to have order structure
- In 1D: x_{i-1}, x_i, x_{i+1} are related because they are consecutive in index
- We want to perform local computation in equivariant manner: If we translate the input (shift some indexes right) then the output shifts in the same way, formally $f(t(x)) = t(f(x))$ for some translation operation $t(\cdot)$
- Achieve by computing the cross-correlation (mistakenly called convolution in DL literature) of the input and some filter:

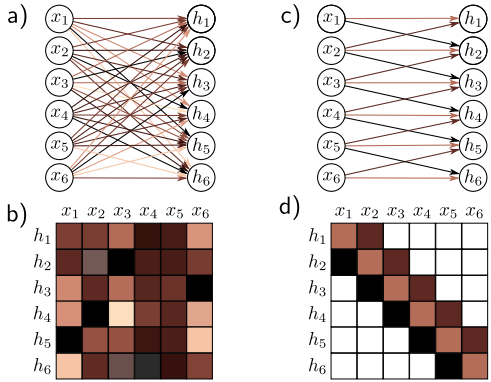
$$z_i = w_1 x_{i-1} + w_2 x_i + w_3 x i + 1$$

  Parameterized by $\mathbf{w} = [w_1, w_2, w_3]$
- The exact same weights used for all z_i, and again we need to push z_i through an activation function

Related concept of invariance: The result does not change after a transformation (e.g. rotation of an image), $f(r(x)) = f(x)$

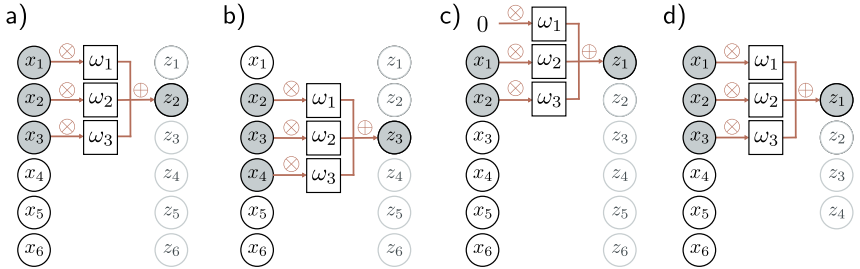
Convolution

Convolutional networks combine sparse connectivity and parameter sharing. Here goes from 36 to 3 parameters.



Source: Prince (2023) CC-BY-NC-ND

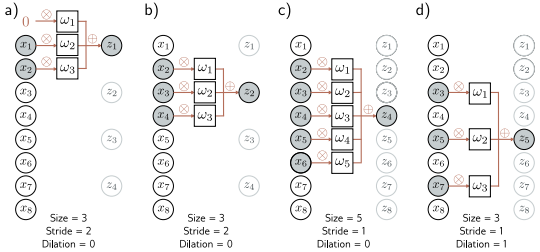
Convolution



Padding helps in keeping layers in same size

Convolution

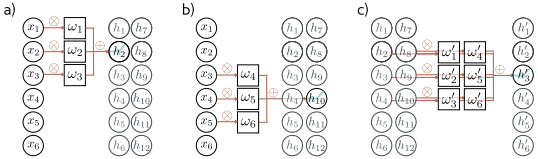
- Behavior of a convolutional filter controlled by *width* (or size), *stride*, and *dilation*
- Width influences the number of parameters, stride changes the output size
- Dilation can be interpreted as using wider but sparse convolutions



Source: Prince (2023) CC-BY-NC-ND

Convolution

- Generalizes directly for 2D (the most common case) and higher order inputs
- A single convolutional filter retains the shape of the input: If the input is $W \times H$ image then the result of the convolution is also $W \times H$ set of neurons (assuming suitable padding)
- Typically we use several filters for the same input, stacking their outputs as *channels* of the hidden layer, analogous to how we can have different color channels in the input
- The result is then $W \times H \times D_o$ tensor



Source: Prince (2023) CC-BY-NC-ND

Convolution

- For $W \times H$ input and D_o filters we get the hidden layer as $W \times H \times D_o$ tensor
- If the input already has multiple channels then the convolution kernel is actually 3D, even though we only convolve over two of them: each filter is size $W \times W \times D_i$. The result size remains the same
- Between two multi-channel layers we might still have quite a bit of parameters: For 2D filters of width 5 and 128 channels in input and output we need 400k parameters, corresponding full connectivitity between layers of 640 neurons
- 1×1 convolution (a filter of $1 \times 1 \times D_i$) can be used to change the channel resolution; each input dimension is processed independently, but we can increase or decrease the channel depth

See: Prince, Chapter 10

Convolutional layer

- A collection of filters defines a *convolutional layer*, characterized by the number of channels, the filter widths, and the stride/dilation/padding choices
- Equivalent to the linear part of a fully-connected layer, so the output still goes through some non-linear activation function
- From the perspective of a DL library and automatic differentiation there are no new challenges, even though manually deriving backpropagation for a convolutional layer would require very careful bookkeeping
- All libraries have relatively clean syntax for defining convolutional layers, but you need to check the syntax

CNNs

- Any NN that uses a convolutional layer is a convolutional neural network (CNN)
- In practice, CNNs typically share also other architectural choices
- Typically we process large inputs (images, sequences) by progressively reducing the layer sizes, by downsampling:
  - Stride: Automatically reduces the size by a factor of stride
  - Pooling: Explicitly reduce the size by computing either average or maximum of the outputs in a local neighborhood
- In most cases a convolutional layer is followed by such pooling layer
- The result after each layer still retains the original spatial structure (sequence, image)
- For classification tasks (and similar) the final representation is 'flattened' to a vector and a fully connected layer (or a few) is used for creating the final output

See: Prince, Section 10.4

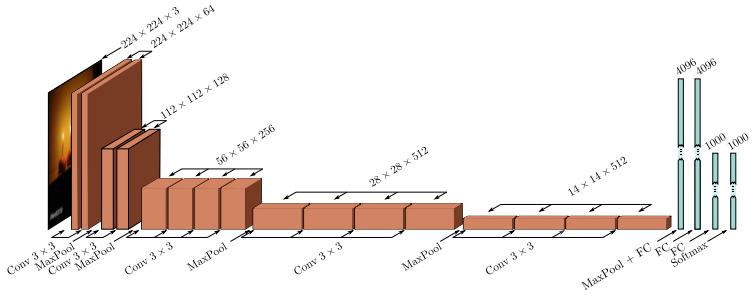
CNNs for image classification

Chapter 8 of Dive into deep learning (`hhttps://d2l.ai/chapter_convolutional-modern/index.html`) gives a good overview of how CNNs for image classification evolved in just a few years

- 2012: *AlexNet*, scaled up version of standard CNNs from 1998
- 2014: *VGG*, blocks of consecutive small convolutions instead of large filters, repeated use of the same blocks
- 2015: *GoogLeNet*, multiple parallel processing branches (the inception block), processing images at different scales in parallel
- 2016: *ResNet*, residual connections as a simpler form of multiple branches. Perhaps the go-to architecture today.
- 2017: *DenseNet*, concatenation instead of addition in residual connections

...and soon after this the field changed to vision transformers as the leading solution for very large data sets, so the flood of new CNN architectures slowed down

CNNs for image classification

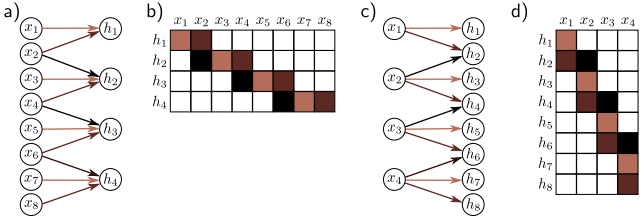


The VGG image classification model from 2014 with block structure. Image scale decreases and the number of channels increases.

Source: Prince (2023) CC-BY-NC-ND

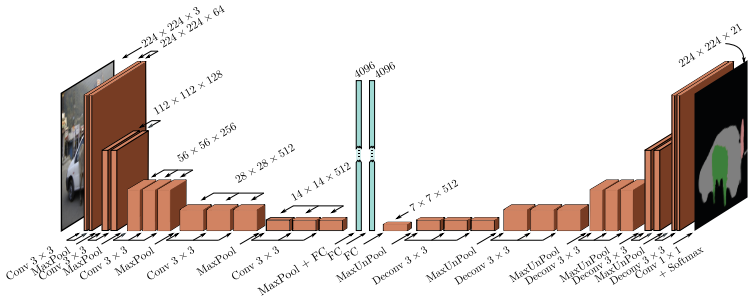
Transposed convolutions and upsampling

- For models where the output is also large (e.g. segmentation) we need to go back from the small representation
- This requires *upsampling*, either explicitly to counter the max/average pooling or by reverting the convolution operation itself
- Explicit upsampling: Repeat the values, do interpolation, ...
- *transposed convolution* (often called *deconvolution*, which is incorrect): Each input connects to a few local outputs
- Can even reuse the same weights that were used for convolution earlier



Source: Prince (2023) CC-BY-NC-ND

CNNs for image segmentation



Note: Left side is (slightly smaller version of) VGG!

Source: Prince (2023) CC-BY-NC-ND

Residual connections

- A *residual connection*

$$\boldsymbol{y}_k = \boldsymbol{y}_{k-1} + \psi(\boldsymbol{W}_k \boldsymbol{y}_{k-1})$$

  is the simplest form of multi-branch networks
- That is, the output is a sum of identity function and a standard layer (or a small network)

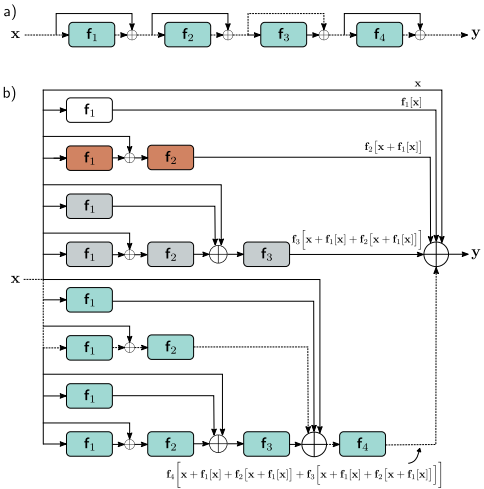
See: Prince, Chapter 11

Residual connections

Motivation:

- Learning identity functions is somewhat difficult for neural networks, so this ensures we can more reliably learn easy functions
- Modelling the *residual* of some fitted function may be easier than modelling the function itself
- Helps propagating gradients through layers; at least some information always goes through the identity path
- Empirically appear to result in smoother loss surfaces around the minima (implicit regularization?)
- In practice allows use of extremely deep networks: 1000 layers is quite fine

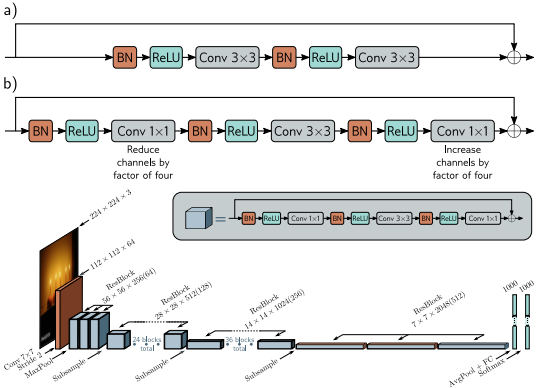
Residual connections



Going deeper: ResNet

- *ResNet* is a popular CNN model that uses residual connections
- Constructed of *residual blocks* where the identity path is combined with a small network of a few layers
- With *bottleneck blocks* scales up to hundreds of layers, and ResNet-200 (with 200 layers, from 2016) was one of the first networks to reach roughly human performance in ImageNet classification
- The *U-Net* architecture uses residual connections in image-to-image tasks

ResNet

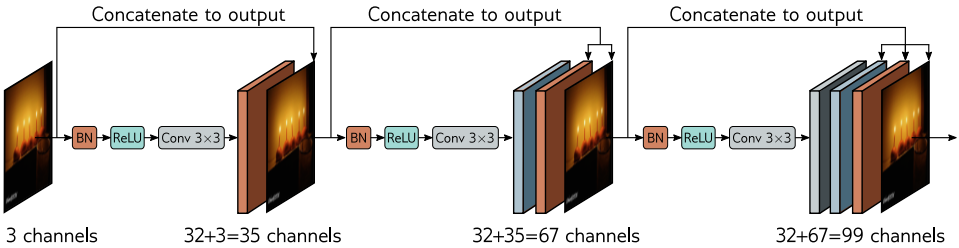


Source: Prince (2023) CC-BY-NC-ND

Going deeper: DenseNet

- *DenseNet* replaces the addition in $\boldsymbol{y}_k = \boldsymbol{y}_{k-1} + \psi(\boldsymbol{W}_k \boldsymbol{y}_{k-1})$ with concatenation $\boldsymbol{y}_k = [\boldsymbol{y}_{k-1}, \psi(\boldsymbol{W}_k \boldsymbol{y}_{k-1})]$
- Sounds like a really minor change but this is more flexible: We could add them back in the next layer, but can also do other computation
- Residual connection retains the size of the representation, but the concatenation means the representation size grows all the time. Need to bring back down at some point.
- A block is formed by a few layers of this

DenseNet



Source: Prince (2023) CC-BY-NC-ND

Recurrent NNs

- Both fully connected and convolutional networks are examples of *feedforward networks*
- Information flows only in one direction; each layer processes information from the previous layer(s), and the multi-branch components (residual connections etc) do not change this fundamental property
- Any neural network architecture that uses as inputs information from a latter layer is called *recurrent* (RNN)
- We still feed inputs one by one, so the values for those layers correspond to what was computed for the previous input

See: Anything else but Prince

Recurrent NNs

- The simplest recurrent network

$$\boldsymbol{h}_t = \psi_h(\boldsymbol{W}_x \boldsymbol{x}_t + \boldsymbol{W}_h \boldsymbol{h}_{t-1})$$
$$\boldsymbol{y} = \psi_y(\boldsymbol{W}_y \boldsymbol{h}_t)$$

- If we initialize $\boldsymbol{h}_0 = 0$ we get

$$\boldsymbol{h}_1 = \psi_h(\boldsymbol{W}_x \boldsymbol{x}_1)$$
$$\boldsymbol{h}_2 = \psi_h(\boldsymbol{W}_x \boldsymbol{x}_2 + \boldsymbol{W}_h \boldsymbol{h}_1) = \psi_h(\boldsymbol{W}_x \boldsymbol{x}_2 + \boldsymbol{W}_h \psi_h(\boldsymbol{W}_x \boldsymbol{x}_1))$$
$$\boldsymbol{h}_3 = \psi_h(\boldsymbol{W}_x \boldsymbol{x}_3 + \boldsymbol{W}_h \boldsymbol{h}_2) = \psi_h(\boldsymbol{W}_x \boldsymbol{x}_3 + \boldsymbol{W}_h \psi_h(\boldsymbol{W}_x \boldsymbol{x}_2) + \boldsymbol{W}_h \psi_h(\boldsymbol{W}_h \psi_h(\boldsymbol{W}_x \boldsymbol{x}_1)))$$

- Already a single hidden layer results in 'deep' computation, but using the same weights for each layer

Recurrent NNs

- Formally recurrence makes the network more powerful, giving it (a rudimentary) memory
- However, for finite-length input streams we can represent the exact same computation with a feedforward network that reads the inputs at different levels: In the previous example, $\boldsymbol{x}_3$ goes through one hidden layer, $\boldsymbol{x}_2$ through two, and $\boldsymbol{x}_1$ through three
- From this perspective, a RNN is still effectively a feedforward network with strong parameter sharing
- Rather than thinking of RNNs are more powerful models, we mostly use RNNs as convenient models for processing sequential data
- RNNs were the default choice for sequential data still recently, but appear to be largely overtaken by transformers (next lecture)

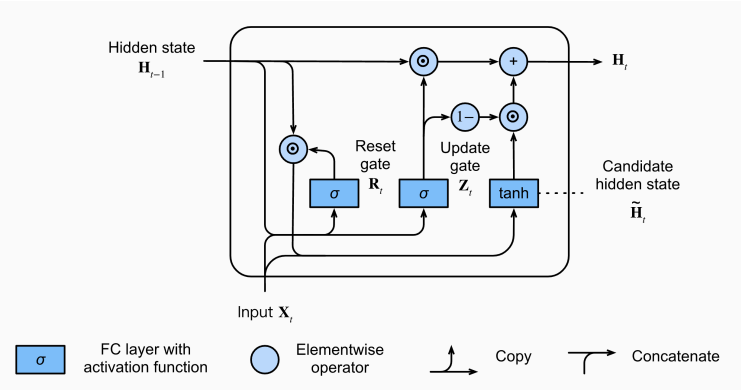
Backpropagation for RNNs

- Despite the recurrence, we can still use standard automatic differentiation for computing the gradients, called *backpropagation through time*
- However, for long sequences this is costly and numerically unstable
  - Information infinitely far in the past still formally has some effect
  - Exploding and vanishing gradients are a bigger problem than for deep NNs in general because we repeatedly use the same weights
- In practice solved by truncation: We simply do not go further than some T steps in history
- See `https://d2l.ai/chapter_recurrent-neural-networks/bptt.html` for a lot of details if interested

Towards longer sequences

- Standard RNNs are rather difficult to get working outside toy problems
- Since we repeatedly use the same weight matrix, we either forget the history fast (if eigenvalues are below 1) or the gradients explode (if eigenvalues are above 1)
- LSTM and GRU are two typical architectures designed to specifically address this
- The basic idea is to make it easy for the network to
  - Forget the history if needed, only using the latest input
  - Let the historical information pass through almost unmodified, to better remember old things
- The basic idea is close to residual connections, but the terminology is different because the RNN-specific solutions were invented (decades) earlier

GRU as practical example



Reset gate: If close to zero, we forget the history and just use latest input
Update gate: Linear combination of previous hidden state and the candidate state

How to design networks?

- Basic concepts
  - Parameter sharing and sparse connections
  - Residual connections (and gates in RNNs)
- Only skimmed the surface of the alternative layers and constructs here, but you should be able to read through scientific papers and tutorials
- Modular construction: Not just 'a convolutional layer' but 'a bottleneck residual block', or even complete architectures 'VGG as first part of segmentation model'
- The design of both individual layers and their combinations can be thought of as hyperparameters, but we cannot realistically use black-box optimization routines to cover all choices
- Some attempts of making the design a bit more systematic: `https://d2l.ai/chapter_convolutional-modern/cnn-design.html`
- New architecture designs typically build on theoretically valid intuitions, but their details are results of extremely heavy experimentation