Machine Learning

Lecture 8: Deep Learning II

Prof. Dr. Stephan Günnemann

Data Analytics and Machine Learning Group
Technische Universität München

Winter term 2020/2021

Roadmap

Structured data

Training deep neural networks

Deep learning frameworks

Modern architectures & tricks

Section 1

Structured data

Adding prior knowledge is always a good idea, instead of learning everything from scratch

Different layers

Our goal is to learn / generalize better

So far we've seen only fully-connected feed-forward layers and our (deep) NNs were obtained by stacking them.

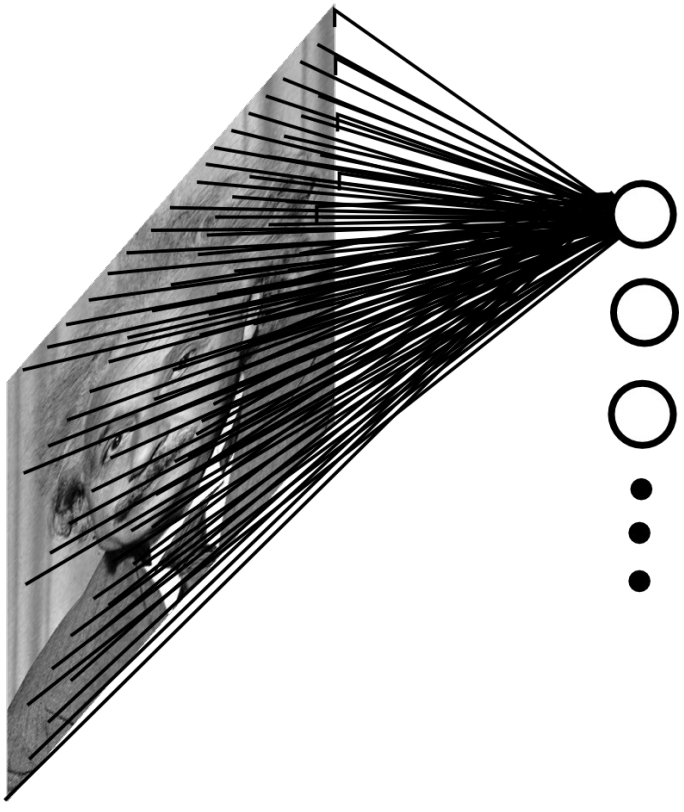
There are many more types of layers for specific tasks/data:

- Convolution layer (typically used for images)
- Recurrent layer (typically used for sequences, discussed in-depth in our MLGS lecture)
- Graph convolutional layers (covered in our MLGS lecture)
- etc.

These layers leverage the data's known structure and provide an inductive bias.

Think of them as building blocks (i.e. lego pieces) that you can compose.

Neural networks for images



from https://cs.nyu.edu/~fergus/tutorials/deep_learning_cvpr12/

Suppose we have an image with 100×100 pixels and want to process it with a neural network with a single hidden layer with 1,000 units.

In a feed forward neural network this results in 10 millon parameters (weights)!

too many dof

We can solve this problem by using the convolution operation to build neural networks. This exploits the high local correlation of pixel values in natural images.

For example, 1,000 (learnable) 5x5 convolutional filters only require 25,000 parameters.

* weighted version of pixel goes into neuron

CNN: Convolution

Continuous convolution is defined as

$$(x * k)(t) = \int_{-\infty}^{\infty} x(\tau)k(t-\tau)\,\mathrm{d}\tau.$$

"sliding window"

This can be intuitively described as a weighted average of the input signal x using the weights (or kernel, filter) k at each point in time t.

CNNs use the discrete variant i.e input size of an image

$$(x * k)(t) = \sum_{\tau=-\infty}^{\infty} x(\tau)k(t-\tau).$$

CNN: Convolution

CNNs for images are based on a 2D convolution

$$(x * k)(i, j) = \sum_l \sum_m x(l, m) k(i - l, j - m)$$

what's actually done

However, when talking about convolution in CNNs we usually mean *cross-correlation*.

This is what is usually implemented in libraries. It is similar to convolution, but the summation indices are swapped:

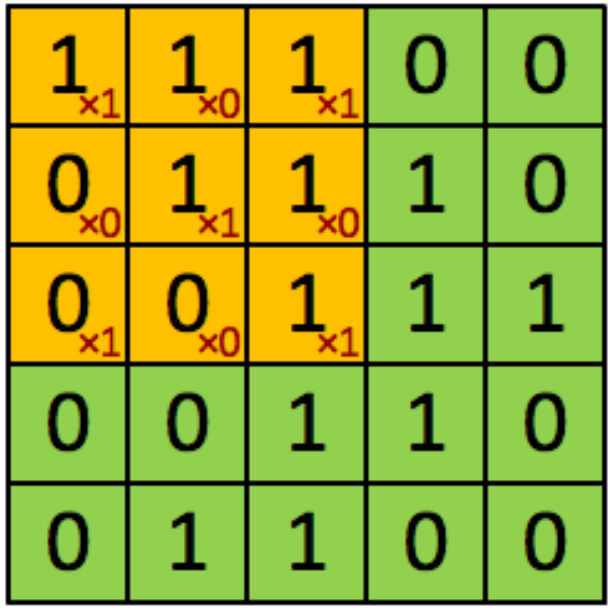
$$\hat{x}(i, j) = \sum_{l=1}^{L} \sum_{m=1}^{M} x(i + l, j + m) k(l, m)$$

* Convolutions usually act on multiple channels. Filters therefore have $L \times M \times C_{\text{in}} \times C_{\text{out}}$ parameters.

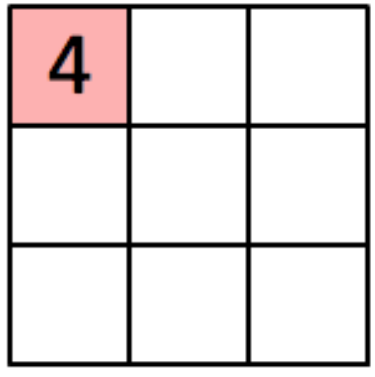
Weight Sharing

9 weights are used instead of 5x5 in FC

learnable parameters is reduced



Image



Convolved Feature

* Weights of the sliding window (kernel) are shared for every patch.

from `http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution`

* Usually multiple filters are applied. Depending on the filter, certain features are extracted

from
http://cs.nyu.edu/~fergus/tutorials/deep_learning_cvpr12/

CNN: Padding

One undesirable effect of applying filters is reducing the output size

Inputs are finite. What happens at the boundary?

We can either reduce the output size (by not applying the filter at the boundary) or pad the input (e.g. with zeros, constant values, or by reflecting or repeating the image).

Padding schemes:

- VALID: Do not use padding, reduce size in output to $D_{l+1} = (D_l - K) + 1$, where D_l is the input size along a dimension and K is the kernel width (width of its "receptive field").
- SAME (half) padding: Add padding so that the input size is preserved, i.e. add $P = \lfloor K/2 \rfloor$ values on each side.
- FULL: Add $K - 1$ values on each side, increasing the output size.

CNN: Strides

A stride S is the distance between positions the kernel is applied.

This changes the output size to

$$D_{l+1} = \lfloor \frac{D_l + 2P - K}{S} \rfloor + 1$$

Strides $S > 1$ are a way of *downsampling* the signal. disjoining the sliding window

They can be used to handle inputs that differ in size.

* Multiple overlaps provide redundancy, which is not helping in extracting features.

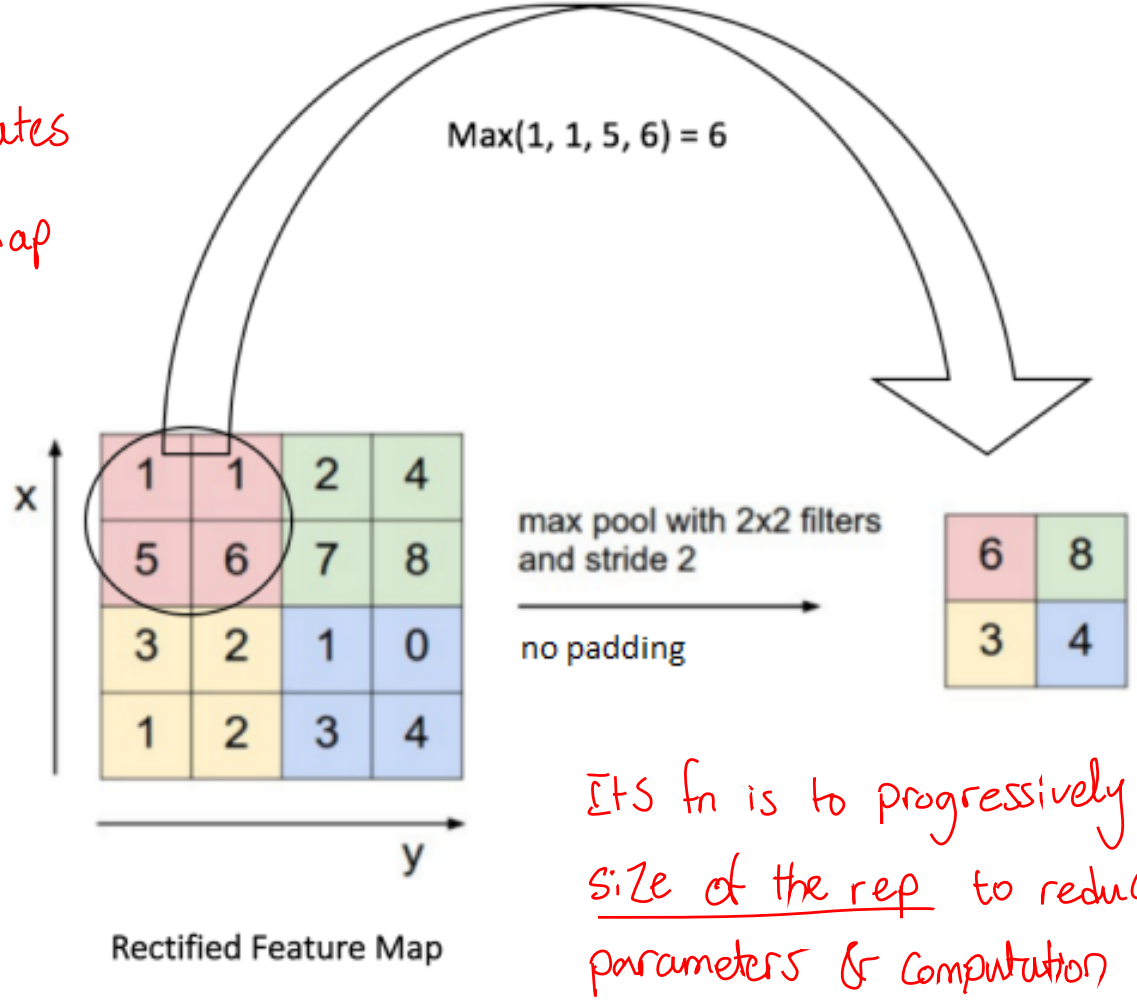
CNN: Pooling

Downsampling + Aggregation

Calculate summary statistics in sliding window. Introduces invariance to small movements.

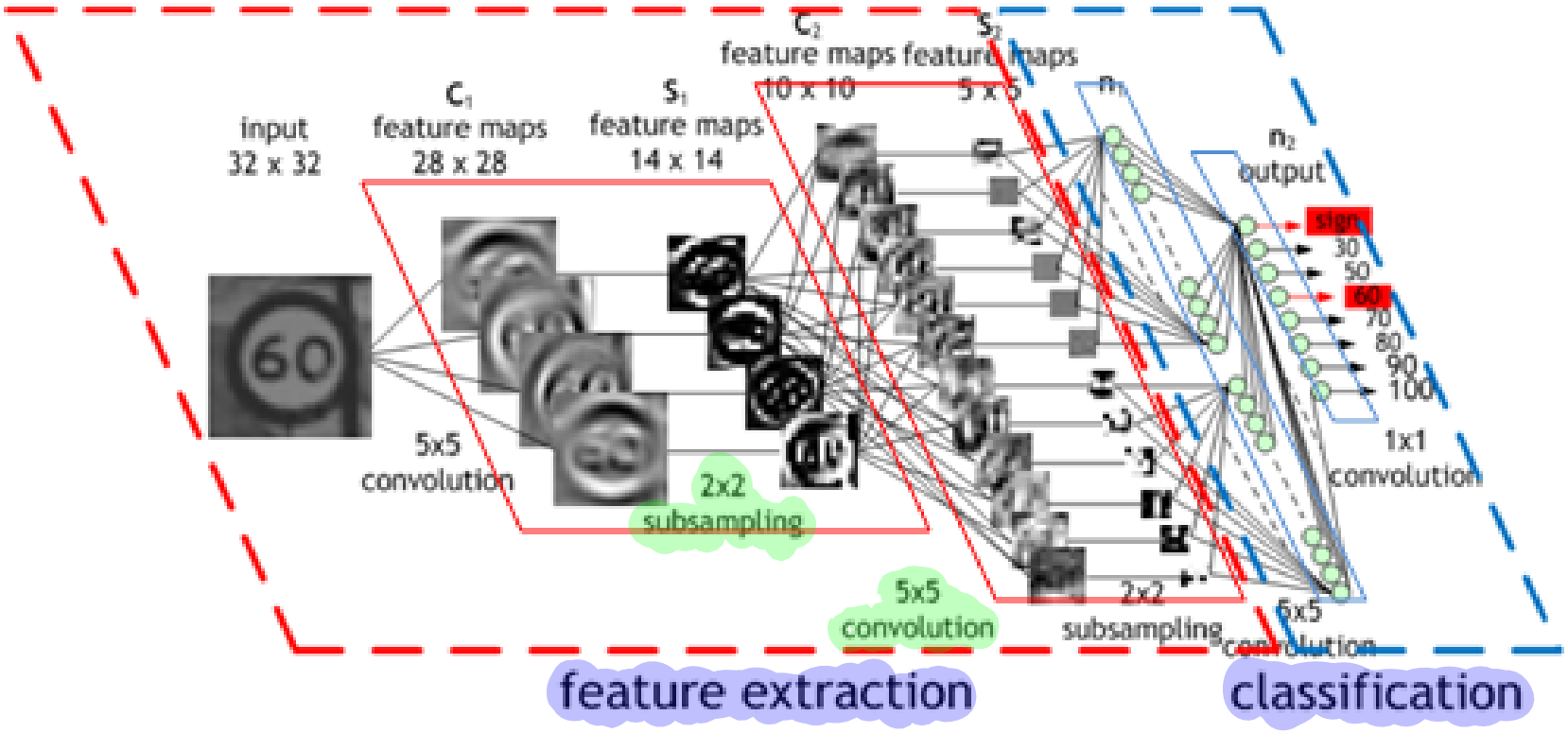
Variants: Max pooling, mean pooling, L_p norm pooling

Pooling layer operates on each feature map independently.



Its fn is to progressively reduce the spatial size of the rep to reduce the amount of parameters & computation in the net

CNN: Convolutional Neural Network



Architectures for other types of structured data

Sequential data (e.g., text, time series)

- Recurrent neural networks (RNN)
- Transformers

Graph data (e.g., social networks, molecules)

- Graph neural networks (GNN)

More about these topics in IN2323 in the Summer semester!

Section 2

Training deep neural networks

Training deep neural networks

Training a neural network means optimizing the loss with SGD, Adam, AMSgrad, or some other optimizer.

From which point do we start optimizing? Weight initialization can be crucial for successful training.

Non convex optimization is more complicated

There are 2 essential issues with naïve weight initialization:

1. Weight symmetry
2. Weight scale

Weight symmetry

If two hidden units have exactly the same bias and exactly the same incoming and outgoing weights, they will always get exactly the same gradient.

i.e they become redundant

- So they can never learn to extract different features.
- We break symmetry by initializing the weights to have small random values.

Weight scale 2

If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause learning to overshoot (take a huge update step).

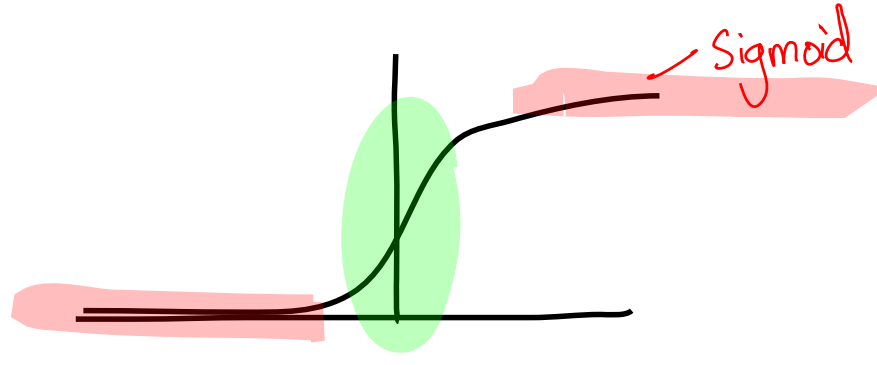
If it has a large fan-out, the same thing can happen during the backward pass.

Wrong weight scales (mean and variance) can therefore lead to vanishing or exploding gradients.

large weights

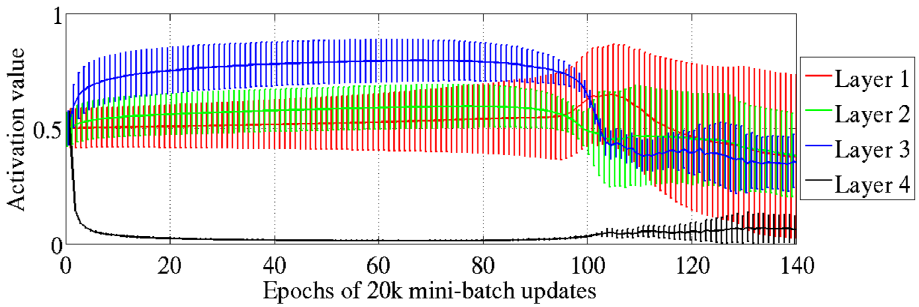
$\alpha(w^T x) \uparrow$

Vanishing grad

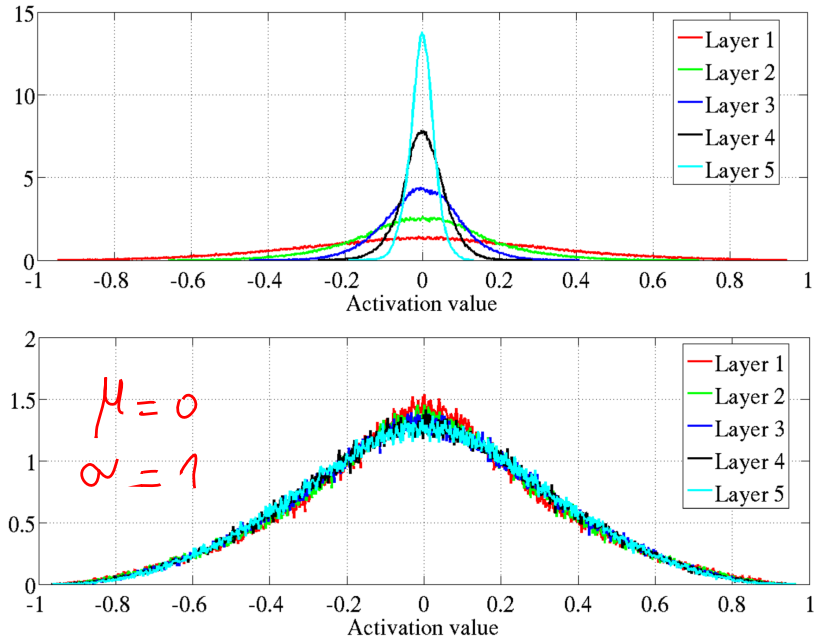


Xavier Glorot initialization

Q: How can we preserve the mean and variance of an incoming i.i.d. signal with zero mean in both forward and backward pass?
A: By controlling the *statistical moments* of the weight distribution.



Mean and standard deviation of activation.
Note saturation of layer 4.



Activation histogram. Top: Unnormalized. Bottom: Glorot init.

from Glorot, Bengio. Understanding the difficulty of training deep feedforward neural networks. 2010

layers still have the same shape as input signal

We can preserve the signal's properties by using weight matrices with zero mean and variance of

$$\mathrm{Var}(\boldsymbol{W}) = \frac{2}{\text{fan-in} + \text{fan-out}}.$$

This is Xavier (or Glorot) initialization. An example weight distribution fulfilling this is

$$\boldsymbol{W} \sim \mathrm{Uniform}\left(-\sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}}, \sqrt{\frac{6}{\text{fan-in} + \text{fan-out}}}\right)$$

no activations

Note that the output layer for regression tasks is usually best initialized taking into account the target scale.

i.e output

check notes

Vanishing and exploding gradients

Deep networks can suffer from the vanishing / exploding gradient problem

Toy example: We multiply the input t times by a matrix $\boldsymbol{W}$

$$\boldsymbol{W} = \boldsymbol{V}\,\mathrm{diag}(\boldsymbol{D})\,\boldsymbol{V}^{-1} \quad \Rightarrow \quad \boldsymbol{W}^t = \boldsymbol{V}\big(\mathrm{diag}(\boldsymbol{D})\big)^t\boldsymbol{V}^{-1}$$

The gradient w.r.t. the elements of $\boldsymbol{D}$ will likely vanish or explode:

- if $\boldsymbol{D}_{ii} < 1.0$, then $\boldsymbol{D}_{ii}^t$ will be near zero, and gradients will become small — the network will take a long time to converge
- if $\boldsymbol{D}_{ii} > 1.0$, then $\boldsymbol{D}_{ii}^t$ will explode and the computations become unstable

Gradients also vanish when using saturating activations (e.g.,sigmoid)

Solutions:

- Change the architecture (e.g., use batch norm, change activations)
- Gradient clipping

check notes

Regularization

Recall that models with high capacity (like NNs) are prone to overfitting. We need regularization to prevent this.

Typically, we use the familiar L_2 parameter norm penalty (or weight decay, which is mostly equivalent). Sometimes we also use L_1 norm to e.g. promote sparsity.

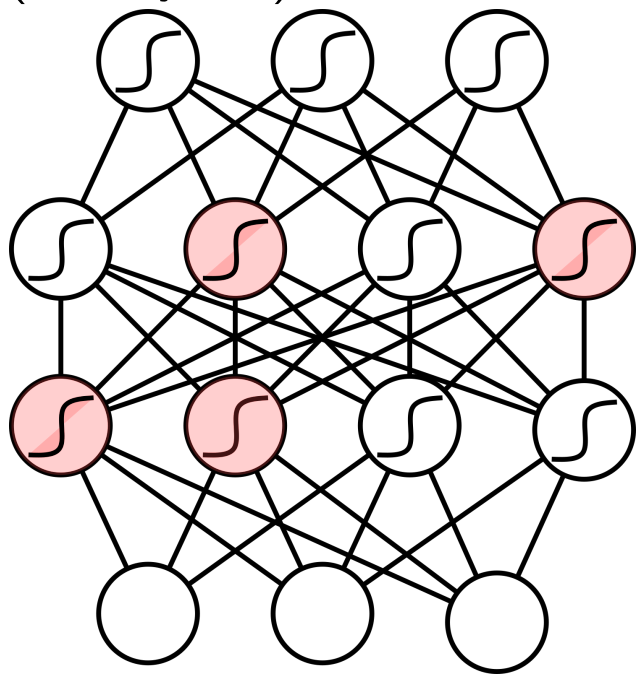
We can combine it with other regularization methods:

- Dataset augmentation: e.g. rotate/translate/skew/change lighting of images
- Injecting noise
- Parameter tying and sharing
- Dropout

Dropout

At every update step, grad is not calc for the actual full architecture but for a subset of this architecture

Let's look at a neural network with two hidden layers.
Each time a training sample is learned, we randomly put to 0 each hidden unit with probability p (usually 0.5).



* We are therefore randomly sampling from 2^H different architectures, but these share the same weights.

No fixed architecture to train on

Hyperparameter optimization

To squeeze every bit of performance out of your NN, you need to tune:

- number of hidden layers (1, 2, 3, . . .)
- number of hidden units (50, 100, 200, . . .)
- type of activation function (sigmoid, ReLU, Swish, . . .)
- optimizer (SGD, Adam, ADADELTA, Rprop, . . .)
- learning rate schedule (warmup, decay, cyclic)
- data preprocessing/augmentation
- . . .

We often start by finding these by "playing around" with some reasonable estimates. A better way of finding a good set is by using hyperparameter optimization. Random search or Bayesian Optimisation are both viable candidates.

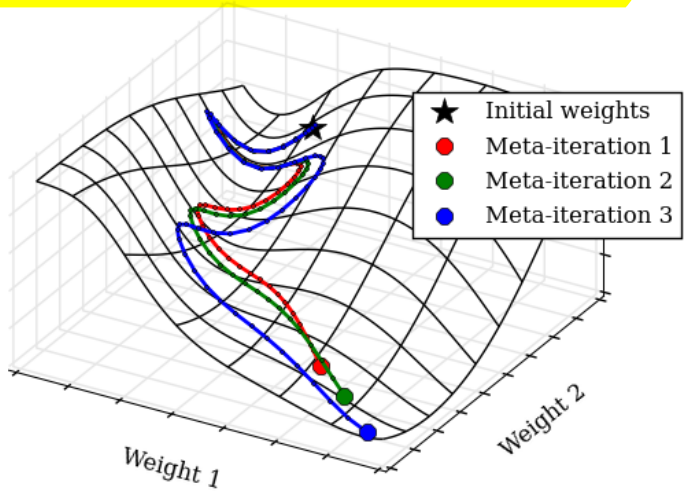
* Lec_6 I2DL

Side note: gradient-based hyperparameter optimization

Instead of "searching" for good hyperparameters, can't we...

...learn them, e.g., with gradient descent?

It turns out we actually can (for continuous hyperparameters)!



from `arxiv.org/pdf/1502.03492.pdf`

We can backpropagate through the training procedure to compute the gradient of the final loss w.r.t. hyperparameters, e.g. the learning rate. We can then perform a 'meta update' on the hyperparameter and repeat.

Some researchers even use this technique to learn the initial weights to enable a model to adapt to different tasks with few training examples ("few-shot learning").

For large neural networks this is however extremely expensive since we have to store all parameters at each training iteration.

Section 3

Deep learning frameworks

Deep learning frameworks

Programming your own NN in Python is quite simple, but not efficient.
For efficient code, use one of many open source libraries, e.g.

- TensorFlow
- PyTorch
- MXNet
- ...

You will then find a number of implementations based on such libraries.

Static vs. dynamic computation graphs

Deep learning frameworks build a computation graph that defines in which order the operations are performed.

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))
```

from https://medium.com/intuitionmachine/pytorch-dynamic-computational-graphs-and-modular-deep-learning-7e7f89f18d1

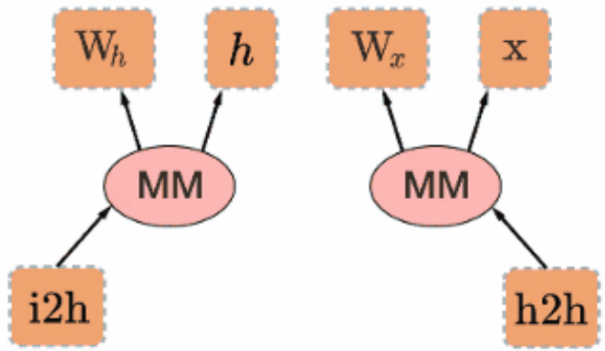
Deep learning frameworks build a computation graph that defines in which order the operations are performed.

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

h2h = torch.mm(W_x, x.t())
i2h = torch.mm(W_h, prev_h.t())
```



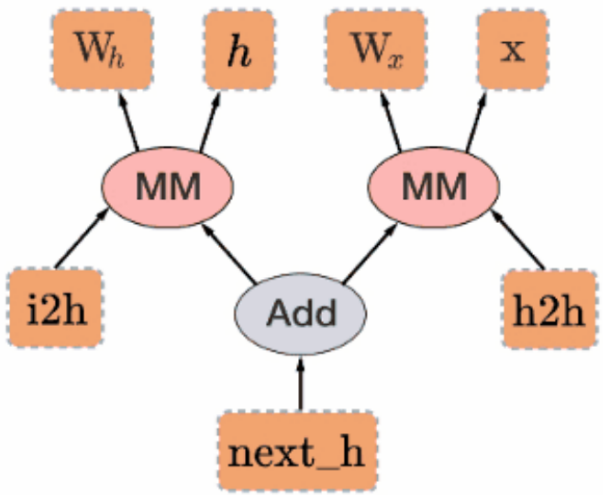
Deep learning frameworks build a computation graph that defines in which order the operations are performed.

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

h2h = torch.mm(W_x, x.t())
i2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
```



Deep learning frameworks build a computation graph that defines in which order the operations are performed.

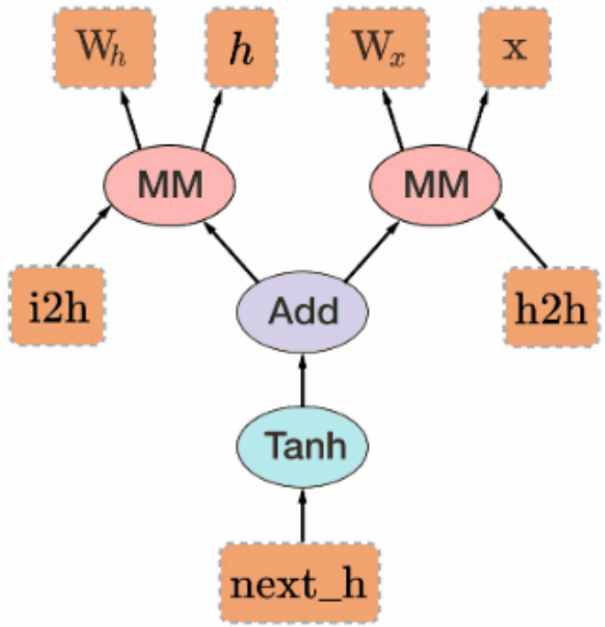
architecture is constructed while code is executed

A graph is created on the fly

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

h2h = torch.mm(W_x, x.t())
i2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()
```



Deep learning frameworks build a computation graph that defines in which order the operations are performed.

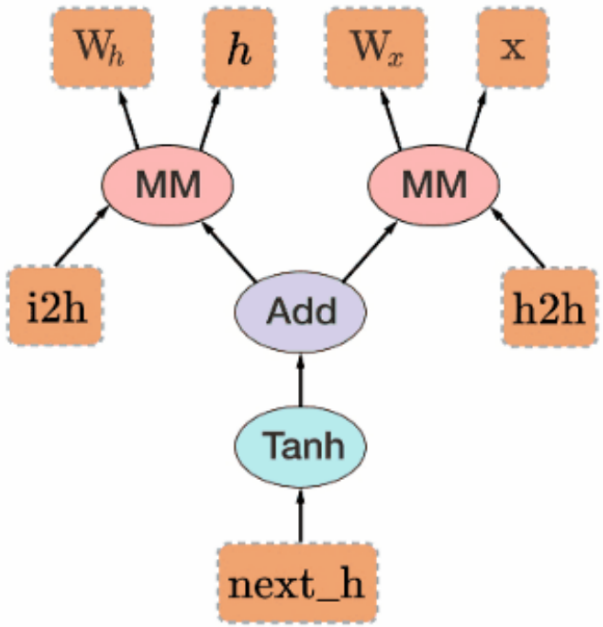
Back-propagation
uses the dynamically built graph

```
from torch.autograd import Variable

x = Variable(torch.randn(1, 10))
prev_h = Variable(torch.randn(1, 20))
W_h = Variable(torch.randn(20, 20))
W_x = Variable(torch.randn(20, 10))

h2h = torch.mm(W_x, x.t())
i2h = torch.mm(W_h, prev_h.t())
next_h = i2h + h2h
next_h = next_h.tanh()

next_h.backward(torch.ones(1, 20))
```



Deep learning frameworks build a computation graph, primarily for calculating gradients.

In the static variant we first *define* the computational graph to later *execute* it with actual data ("Define-and-Run").
In the dynamic variant the graph is *defined by executing* the desired operations ("Define-by-Run").

easier to debug

Compile on the fly

Static computation graphs can be optimized by the framework, similar to a compiler optimizing source code. However, we cannot change a static graph at runtime.

One example are RNNs. With a static computation graph an RNN gets explicitly unrolled for a specified number of time steps. This means that it cannot process sequences of varying time/length.

Such architectures don't have static graphs

The major frameworks have now moved to dynamic computation graphs, since they are more natural to work with. Static graphs can then be generated via JIT compilation of annotated functions.

Section 4

Modern architectures & tricks

Batch normalization

Widely used method for improving the training of deep neural networks.

Goal: Stabilize the distribution of each layer's activations.

Ideal: Whiten each activation independently. Too expensive for each training step, use minibatch statistics instead (or a moving average):

$$\hat{x} = \frac{x - \mathbb{E}_{\mathcal{B}}[x]}{\sqrt{\mathrm{Var}_{\mathcal{B}}[x] + \epsilon}}$$

Layer might not be informative as before

To maintain the model's representational power we also introduce a scale parameter γ and an offset β. These are learned via backpropagation.

$$y = \gamma \hat{x} + \beta$$

This is done after each layer (before the non-linearity). x is the batch norm layer's input, y its output

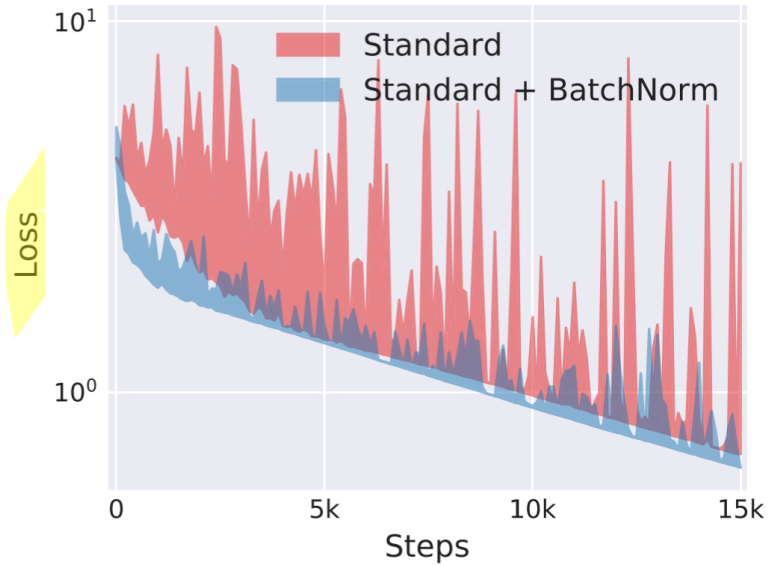
y is later fed into activation

Check notes

loss fluctuates alot ;

because of SGD

Why does it work?



Loss landscape and gradient change per optimization step. From: Santurkar et al. How Does Batch Normalization Help Optimization? 2018

* Batch normalization smoothens the loss landscape.

This causes more predictive and stable gradients, which helps training.

(Note: The original explanation based on internal covariate shift turned out to be wrong.)

Skip connections

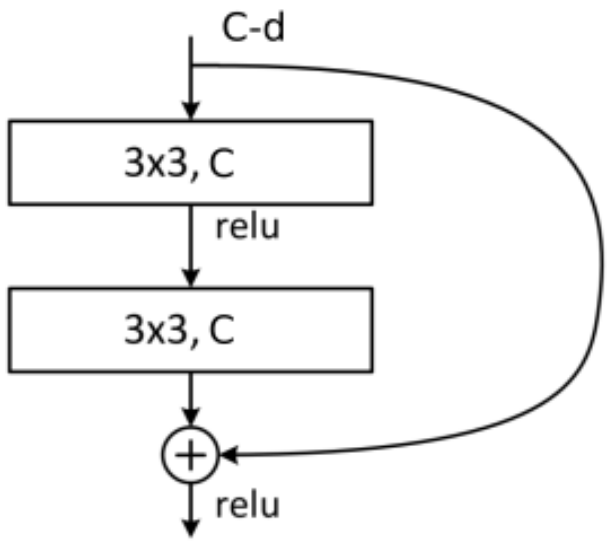
Improve information flow in deep neural networks.

learnable

Highway networks: Add layer input to its output, weighted by gate T:

$$\boldsymbol{y} = f(\boldsymbol{x}, \boldsymbol{W})T(\boldsymbol{x}, \boldsymbol{W}_T) + \boldsymbol{x}(1 - T(\boldsymbol{x}, \boldsymbol{W}_T))$$

Residual connections: Add previous layers' output to a subsequent layer, skipping multiple (usually 2) layers. Usually no gating or other transformation.



Residual connection. From He et al. Deep Residual Learning for Image Recognition. 2016

Tips and tricks

- Use only differentiable operations. Nondifferentiable operations are e.g. $\arg\max$, sampling from a distribution (see our MLGS lecture)
- Always try to overfit your model to a single training batch or sample to make sure it is correctly 'wired'.
- Start with small models and gradually add complexity while monitoring how the performance improves.
- Be aware of the properties of activation functions, e.g. no sigmoid output when doing regression.
- Monitor the training procedure and use early stopping.
- See also: A Recipe for Training Neural Networks
  `https://karpathy.github.io/2019/04/25/recipe/`

Use as much prior knowledge as possible

Summary

- CNNs: Convolution, padding, strides, pooling
- Xavier Glorot initialization, regularization (dropout), hyperparameter optimization
- deep learning frameworks; static & dynamic computation graphs
- batch normalization, skip connections, tips & tricks

Reading material

- Goodfellow, Deep Learning: chapters 7, 8, 9, 11
- Karpathy, A Recipe for Training Neural Networks
  `https://karpathy.github.io/2019/04/25/recipe/`