Machine Learning

Lecture 9: Deep Learning

Prof. Dr. Stephan Günnemann

Data Mining and Analytics
Technische Universität München

17.12.2018

Beyond classification...

NNs for regression

Remember that our data set consists of targets $\boldsymbol{y} = \{y_1, y_2, \ldots, y_N\}$ and corresponding input vectors $\boldsymbol{X} = \{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N\}$.

We measure random variable y as

$$y = f(\boldsymbol{x}, \boldsymbol{W}) + \epsilon \quad [\epsilon\text{: Gaussian, zero mean}]$$

Then the log likelihood is

$$\ln p(\boldsymbol{y} \mid \boldsymbol{X}, \boldsymbol{W}) \propto -\frac{1}{2} \sum_{i=1}^{N} \big(y_i - f(\boldsymbol{x}_i, \boldsymbol{W})\big)^2$$

We call the negative log likelihood the loss $\mathcal{L}(\boldsymbol{W})$ aka $E(\boldsymbol{W})$.

Note: Previously we had sigmoid activation function in the output layer (to obtain probabilities). Now use a linear activation function (i.e. identity or no activation function).

Unsupervised deep learning

So far we have only considered neural networks for supervised learning. Unsupervised deep learning is also a very active field of research, which models such as

- Autoencoder (covered later in this course)
- Variational autoencoder (covered later in this course)
- Generative adversarial networks (GAN; covered in our MMDS lecture)
- Unsupervised representation learning (e.g. word or node embeddings; covered in our MMDS lecture)

Beyond simple architectures...

Different layers

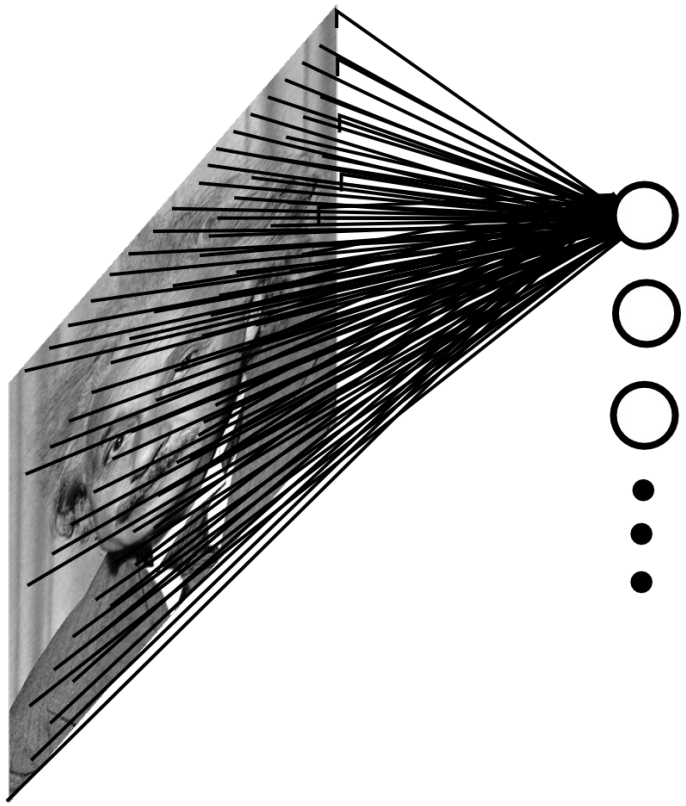
So far we've seen only fully-connected feed-forward layers and our (deep) NNs were obtained by stacking them.

There are many more layers useful for specific tasks/data:

- Convolution layer (typically used for images)
- Recurrent layer (typically used for sequences)
- Batch-norm layer
- Graph convolutional layers (covered in our MMDS lecture in spring)
- etc.

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/

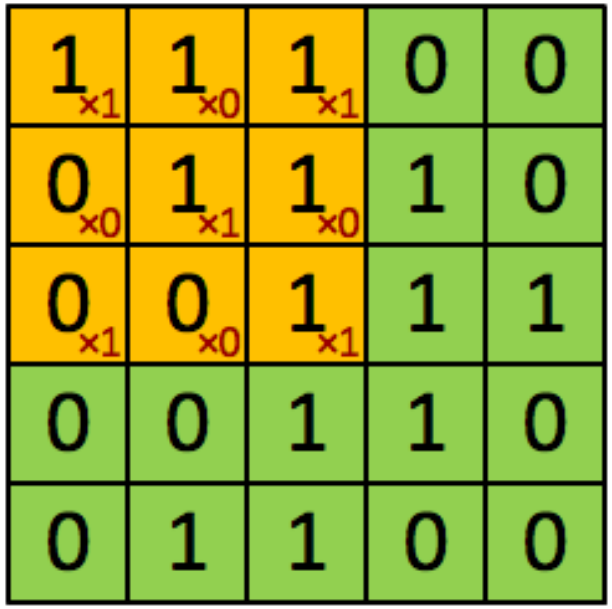
Suppose we have an image with 100×100 pixels and want to process it with a neural network with a single hidden layer with 1,000 units.

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

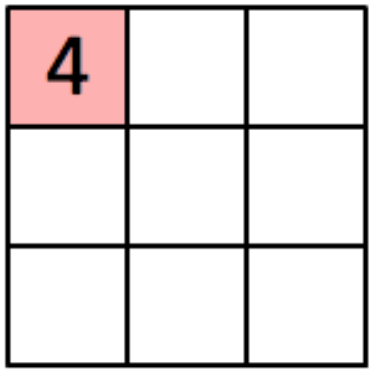
We can solve this problem by using the convolution operation to build neural networks. This exploits the high local correlation of pixel values in natural images.

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

CNN: Convolution



Image

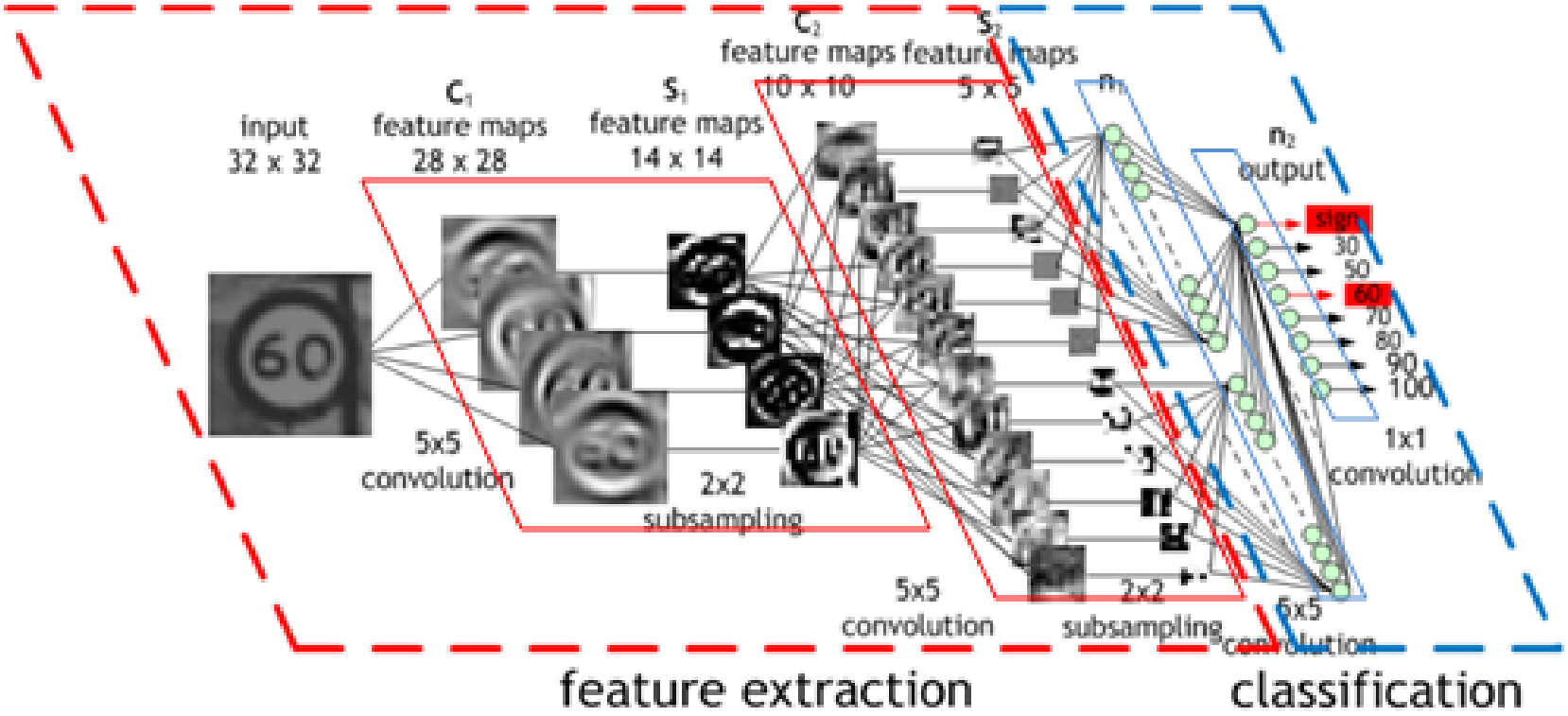


Convolved Feature

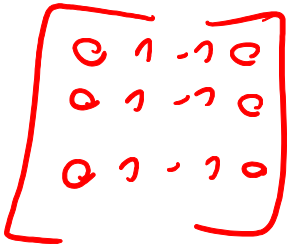
Weights of the sliding window are shared for every patch.

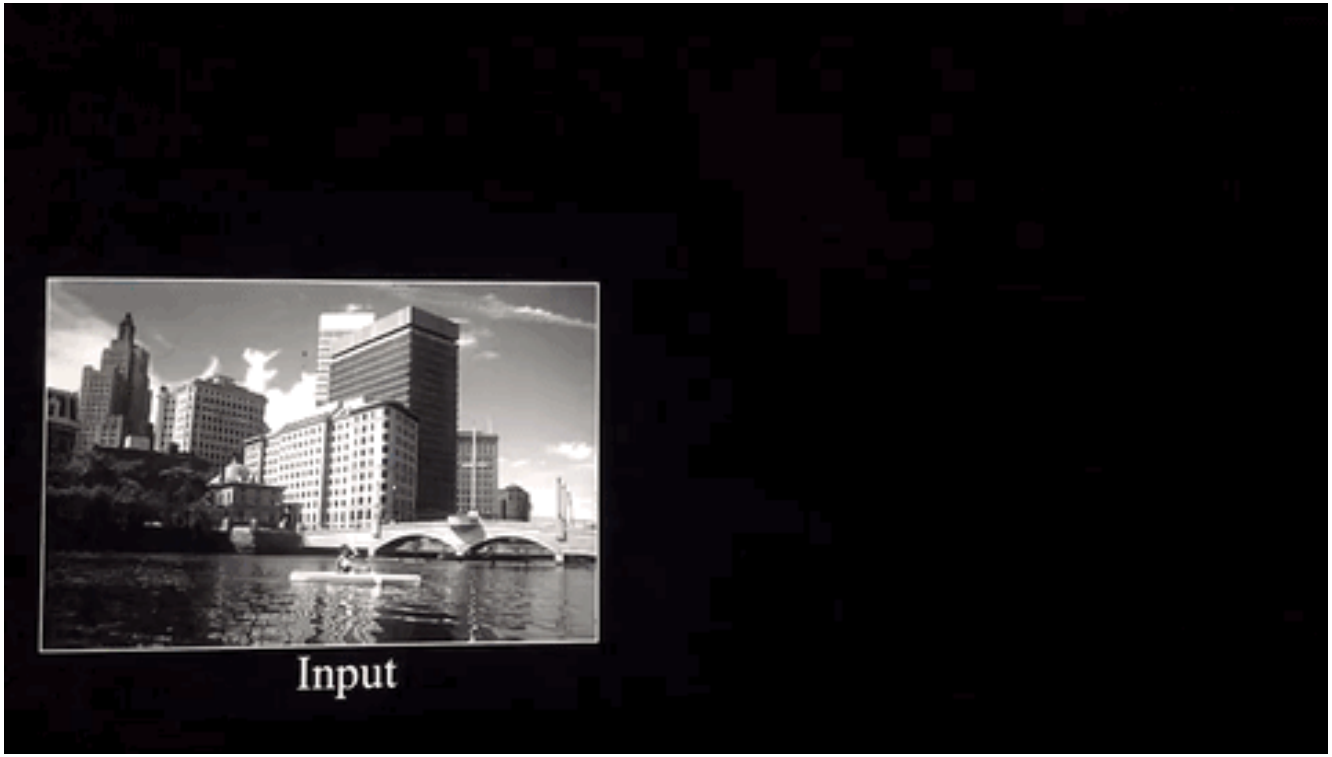
from `http://deeplearning.stanford.edu/wiki/index.php/Feature_extraction_using_convolution`

CNN: Convolutional Neural Network



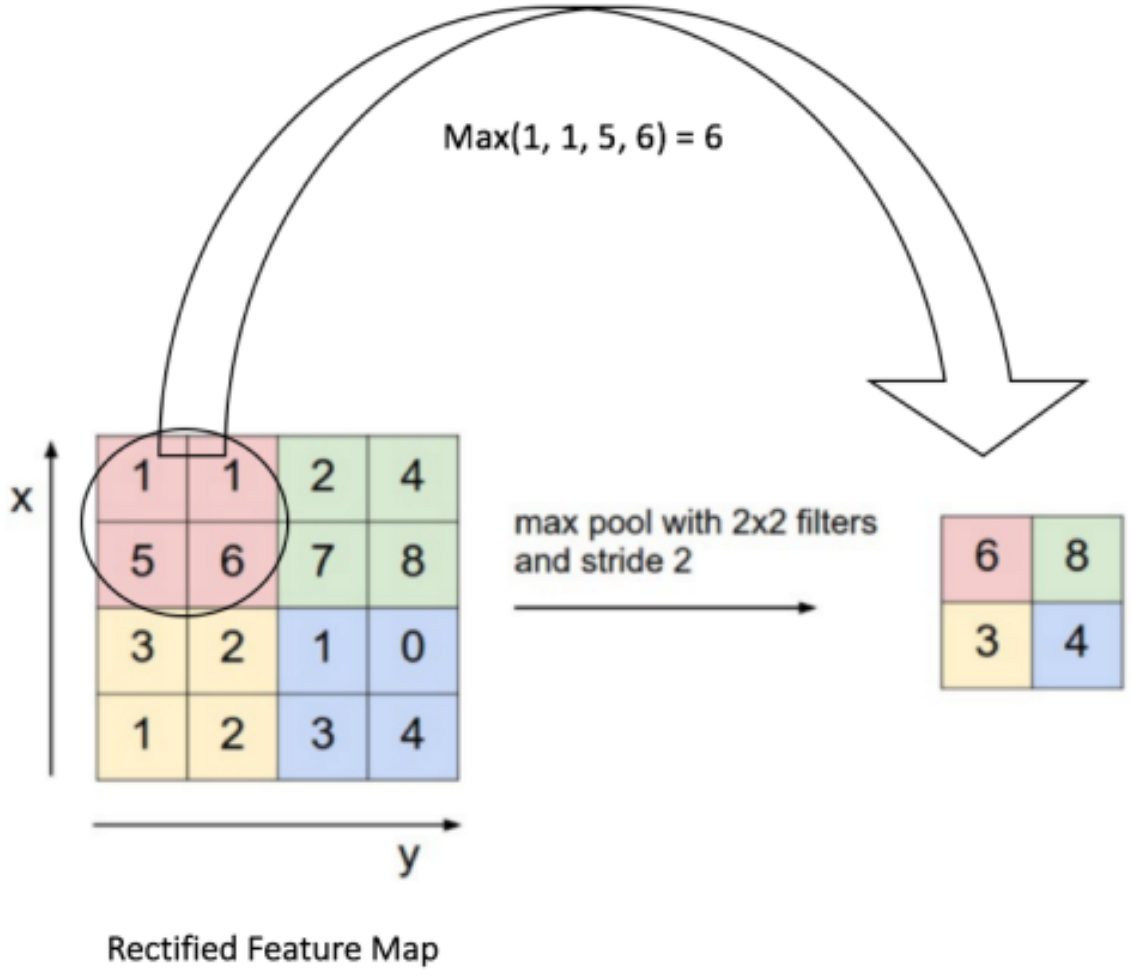
CNN: Convolution



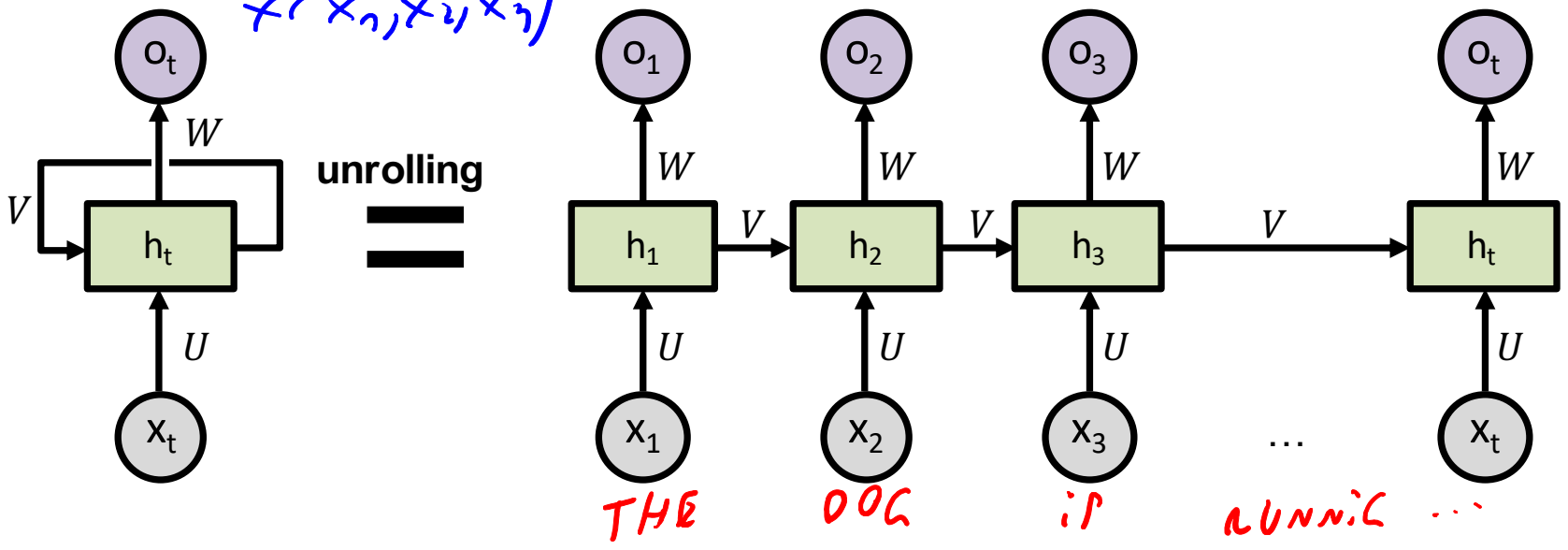


from
http://cs.nyu.edu/~fergus/tutorials/deep_learning_cvpr12/

CNN: Maxpooling



RNN: Recurrent Neural Networks



$o_t = \sigma_o(Wh_t + b_o)$
$h_t = \sigma_h(Ux_t + Vh_{t-1} + b_h)$
σ_o, σ_h: activation functions
o_t: output at time t

h_t: hidden state at time t
x_t: input at time t
U, V, W: weight matrices
b_o, b_h: bias terms

Source: http://colah.github.io

RNN: Recurrent Neural Networks

RNN = FFNN rolled out in time

Can be trained with backpropagation-through-time:

- input a finite sequence $\{\boldsymbol{x}(t)\}$, $t = 1, 2, \ldots T$ step by step;
- weights are shared across every time step;
- back-propagate the error $f(\boldsymbol{x}(t)) - \boldsymbol{y}(t)$ timestep by timestep, starting at T down to 1, and sum the residuals;
- adapt the weights using the residuals computed above.

RNNs are known to suffer from the **vanishing/exploding gradient problem**. One popular model that aims to alleviate this issue is the **Long short-term memory network (LSTM)**. This is convered in our Mining Massive Datasets (MMDS) lecture in the summer term.

Weight sharing

More generally, both RNNs and CNNs (and other specialized NNs) can be seen as feed forward neural networks where weights are shared among certain nodes.

This can bring several advantages:

- We need fewer learnable parameters for the network (lower risk of overfitting)
- It introduces an *inductive bias*, i.e. 'forces' the model to exploit the characteristics of the data (e.g., translation invariance and high local correlation in images) and allows the model to prioritize one solution (or interpretation) over another.

Component	**Entities**	**Relations**	**Rel. inductive bias**	**Invariance**
Fully connected	Units	All-to-all	Weak	-
Convolutional	Grid elements	Local	Locality	Spatial translation
Recurrent	Timesteps	Sequential	Sequentiality	Time translation
Graph network	Nodes	Edges	Arbitrary	Node, edge permutations

Source: `https://arxiv.org/pdf/1806.01261.pdf`

Training deep neural networks

Weights initialization

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.

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

If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot.

- We generally want smaller incoming weights when the fan-in is big, so initialize the weights to be proportional to $\sqrt{(\text{fan-in})}$.

Regularization

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

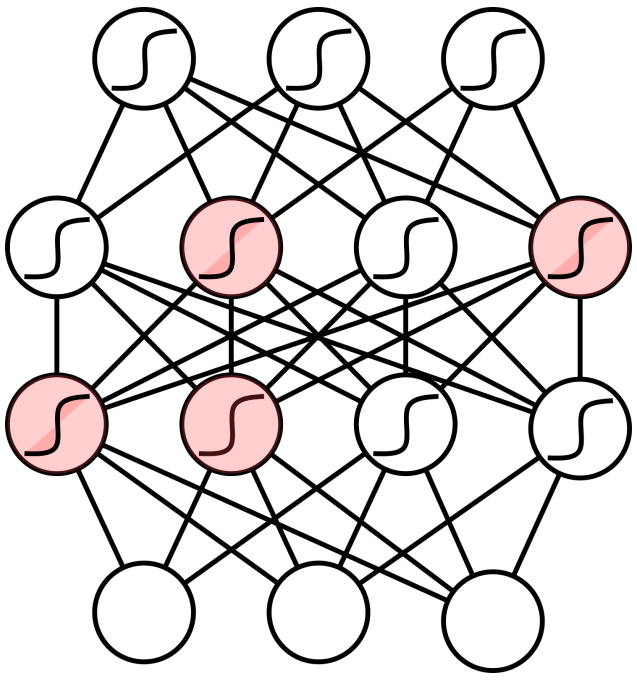
Typically, we use the familiar L_2 parameter norm penalty. Sometimes we also use L_1 norm to e.g. promote sparsity.

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 (is a form of regularization)

Dropout

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



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

Hyperparameter optimization

Getting the last 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, tanh, rectifier, ...)
- learning method (adam, adadelta, rprop, ...)
- learning rate
- data preprocessing
- ...

We often start finding these by "playing around" with some reasonable estimates. In the end, one often uses hyperparameter optimization to find a good set. Random search or Bayesian Optimisation are both viable candidates.

Side note: gradient-based hyperparameter optimization

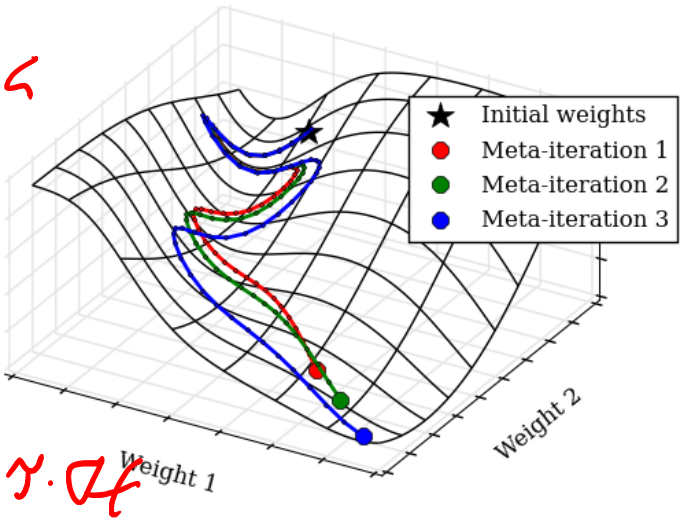
META-LEARNING

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)!

$\theta^{t+1} \leftarrow \theta^t - \gamma \cdot \nabla$



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.

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

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
- Caffe2
- ...

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`

Static vs dynamic computation graphs

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

Typically we first *define* the computational graph to later *execute* it with actual data.
This means that the framework can optimize the computations e.g. on the GPU. However it means that we cannot change the graph at runtime.

One example are e.g. RNNs. With a static computation graph a RNN gets explicitly unrolled for a specified number of time steps. This means that we cannot use the RNN to process sequences of varying time.

In general, deep learning frameworks are moving towards dynamic computation graphs in favor of static graphs since they are more natural to work with.

Tips and tricks

- Use only differentiable operations
  - Nondifferentiablle operations are e.g. $\arg\max$, sampling from a distribution
  - More on sampling in neural networks later in this course and in our course Mining Massive Datasets (MMDS)
- 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.