Advanced course in machine learning
582744
Lecture 11

Arto Klami

The rest of the course

Still two lectures: Deep learning continues on Thursday, and the next Tuesday lecture is re-cap and instructions for preparing for the exam

The 7th set of exercises w ould be due May 10th, a day before the exam... I will also release some extra problems for compensating missed exercises

The decision made during the lecture: The deadline will be postponed by a week

Deep learning

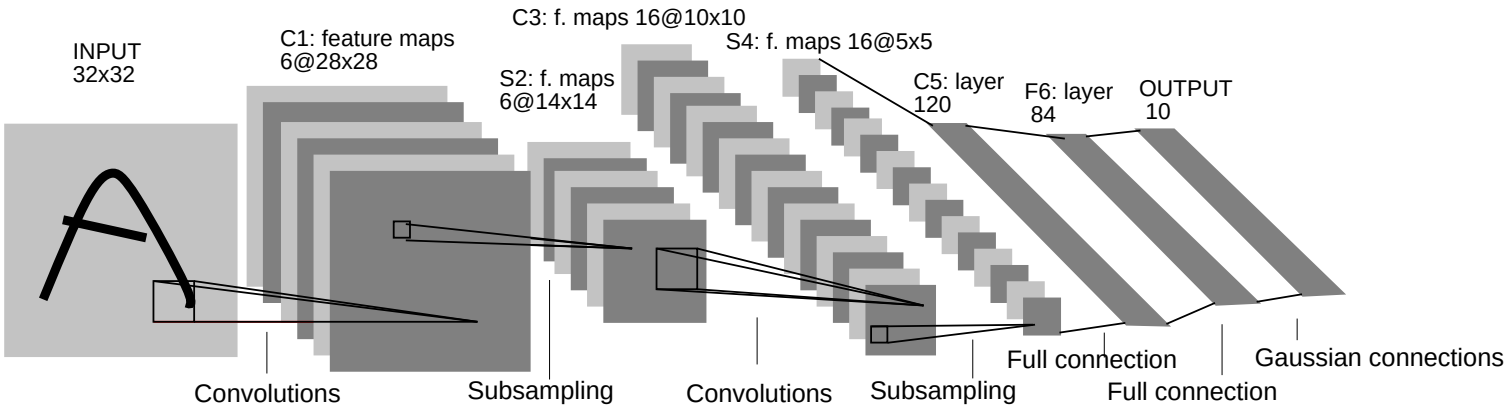
Deep learning refers to any machine learning solution that is "deep", but typically it is understood as neural networks that have high number of layers

The intuitive reasoning is roughly that deep networks extract progressively higher-level *distributed representations*, so that the final supervised problem is easier to solve

Important for applications where the raw data features are not very informative (pixel values vs words)

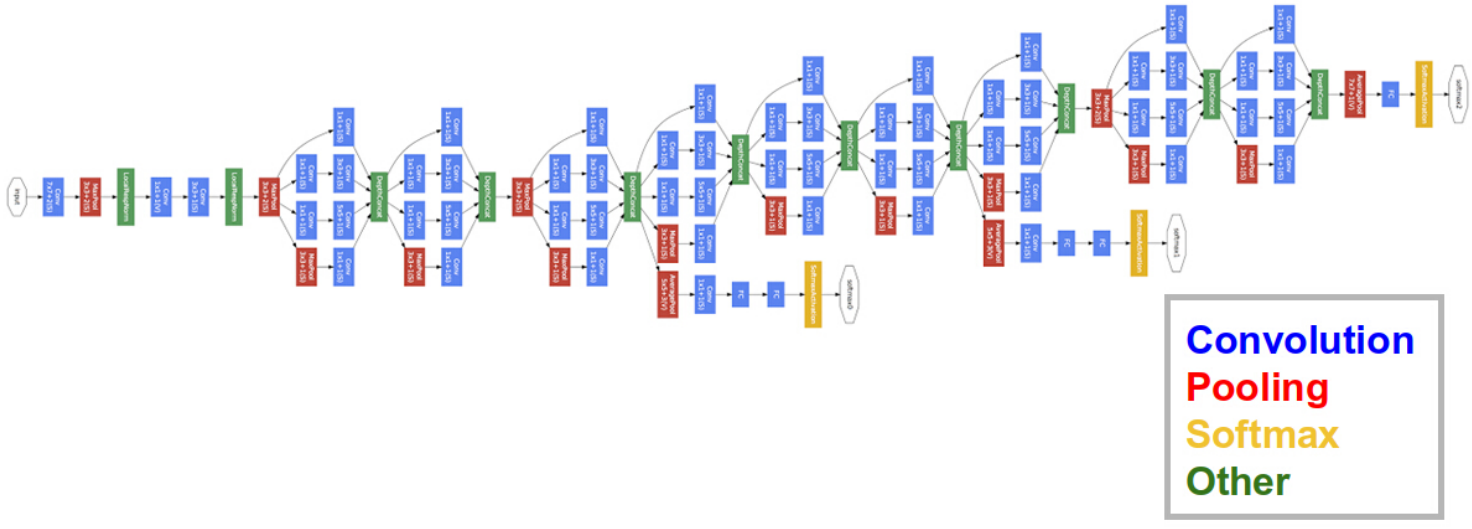
Shallow models sometimes used to describe all other models

Deep learning



LeNet5 (LeCun et al. 1998)

Deep learning



GoogLeNet

Image recognition

Instead of fully connected MLPs we typically want to encode knowledge about the problem domain into the network

One standard example is image recognition, where we know the input vector is actually a two-dimensional array where neighboring pixels are related

The standard model for this is *convolutional neural network*, which means any network that uses convolutions as part of the model

Convolution

Convolution is the integral of the product of two function where one is reversed and shifted:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau$$

Discrete convolution with $g(\cdot)$ having finite support is simply

$$(f * g)[t] = \sum_{m=-t}^{m} f[t - m] g[m]$$

which computes for all $f[t]$ a weighted average of its surrounding

Two-dimensional extension applies a local filter and computes its output for each pixel

Convolutional neural networks

The first layer of a neural network computes convolution over the image

In other terms: Each unit looks at a local reseptive field and averages the inputs using the weights corresponding to some filter

The weights for each unit are identical, so they correspond to applying the same filter for each area of the image

Translation invariance: We can detect the same pattern irrespective of where it is (though it is recognized by a different unit)

The layer naturally has several such filters; their number is called the *depth* of the convolutional layer

Convolutional neural networks

Pooling or subsampling: Reduce the dimensionality of the convolutional layer by

- Average pooling: Compute the average of 2x2 or 4x4 neighborhoods of the convolutional layer
- Max pooling: Take the max of the same neighborhood – this if often used in practice

Reduces the number of parameters, provides further translation invariance

The combination of convolution and subsampling motivated by simple and complex cells in visual cortex

Convolutional neural networks

The model can have several nested combinations of convolution and pooling, possibly having some ReLU layers in between as well

Also one-dimensional convolutions are meaningfull: Weighted averages of text or other strings

Practical trick for optimization: Generate additional training images by distorting the original ones; helps making the solution invariant to small distortions

Generating a deep learning model to classify images would be considerably harder if not taking the structure into account (permutation-invariant models have worse accuracy)

Deep learning in image recognition

You can play around with the interactive demo at `http://cs.stanford.edu/people/karpathy/convnetjs/demo/cifar10.html`

CIFAR-100 image classification benchmark: Top deep learning methods around 75% accuracy and majority of the top results are post-2015. The best other method at 61% – similar story for all image benchmarks

During the past few years the top solutions often had 10-30 layers, but e.g. the ImageNet 2015 winner went up to 150

Training deep MLPs

LeNet5 already from 1998 – why did the deep learning hype start only some years ago?

Training deep networks is difficult:

- The "vanishing gradient problem": The derivatives wrt to the lower layers often tend to zero
- Only the last layers capture meaningful representations: One hidden layer is enough for universal approximation so the networks have local optima for which the lower levels need not do anything
- High number of parameters means slow computation
- A lot of data is needed to gain advantage of a more complex model

Training deep MLPs

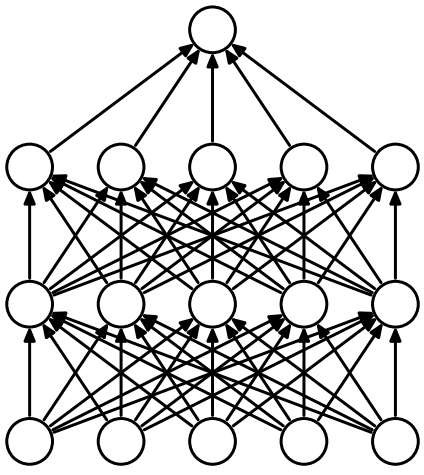
So what has changed?

- Faster hardware, especially GPU computation: Neural networks consists of huge number of simple calculations, which is exactly what GPUs were designed to do – now we just operate on units instead of pixels of a visual rendering
- Good large data sets: ImageNet, CIFAR, Google internal data, etc.
- Better optimization methods (in fact, largely SGD), initialization and regularization (dropout), batch-normalization
- Good software platforms, automatic differentiation
- Layer-wise generative pre-training, long-term connections

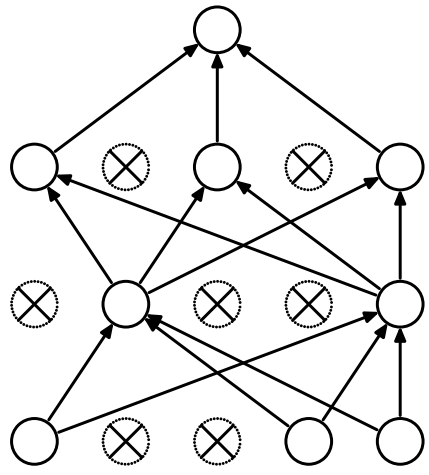
Dropout

Unconventional but efficient regularization technique:

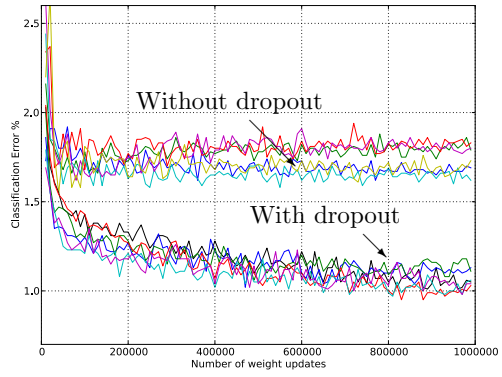
- While learning the model, we randomly exclude some subset of units (with percentage $p \approx 0.5$ for each unit) for processing each training sample
- At test time we use all units



(a) Standard Neural Net



(b) After applying dropout.



Picture from Srivastava et al. JMLR 2014

Dropout

Various motivations:

- It makes the network an ensemble: We learn 2^n networks of n nodes
- Each unit has to learn to work with any randomly chosen subset of other units
- Avoids co-adaptation
- Increases sparsity
- Additional noise for the hidden units
- Marginalizing the noise out for simple networks shows relationship to classical regularization: For linear regression it matches ridge regression

Batch normalization

Imagine a linear network of single unit per layer: $y = w_1 w_2 w_3 w_4 x$

We can arbitrarily scale w_1 and w_2 as long as their product remains the same – we do not want to spend precious computation trying to do this

Batch normalization replaces the outputs $\mathbf{H}$ of a hidden layer by

$$\frac{\mathbf{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}},$$

where the mean and variance as estimated based on the current mini-batch used for SGD

This transformation is encoded into the network, so that we backpropagate through it – the gradients cannot propose any changes that would only re-scale or translate the outputs

Computational graphs

TensorFlow, Theano and Torch provide another layer of computational abtraction, representing neural networks as *computational graphs*

Programming a deep learning model consists of specifying the computational nodes and their reletionships

Learning procedures of computing a gradient or updating the weights are simply another set of computational instructions

- Automatic differentiation
- State-of-the-art optimization algorithms
- Parallel computation, GPU support
- Existing implementations for typical models

Note: The same tools work for many other ML models too!

TensorFlow example

```
x = tf.placeholder(tf.float32, [None, 784])
y_ = tf.placeholder(tf.float32, [None, 10])

W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
y = tf.nn.softmax(tf.matmul(x, W) + b)

cross_entropy = -tf.reduce_sum(y_*tf.log(y))

train_step = tf.train.GradientDescentOptimizer(0.01)
  .minimize(cross_entropy)
```

TensorFlow example

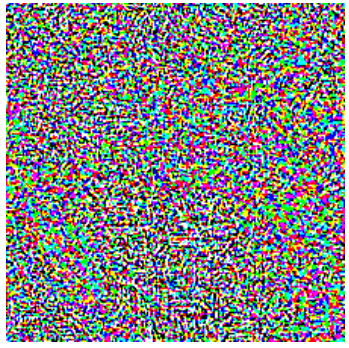
```
init = tf.initialize_all_variables()

sess = tf.Session()
sess.run(init)

for i in range(1000):
  batch_xs, batch_ys = mnist.train.next_batch(100)
  sess.run(train_step, feed_dict={x: batch_xs, y_: batch_ys})
```

Modern practices for deep feedforward nets

- Use enough data and GPU computation (via TensorFlow or a similar tool)
- Rectified linear units the default choice for hidden layers
- Convolutions + max pooling for images and other spatial data
- The structure: Pretty much still trial and error; need not even be a chain of layers (skip connections); parameter sharing
- Backpropagation with SGD and momentum, adaptive step-length
- Dropout regularization, probably combined with l_2 or max-norm regularization
- Use batch normalization

Adversarial training

Szegdy et al. (2014) found out that deep learning models for image classification can be fooled by constructing adversarial inputs

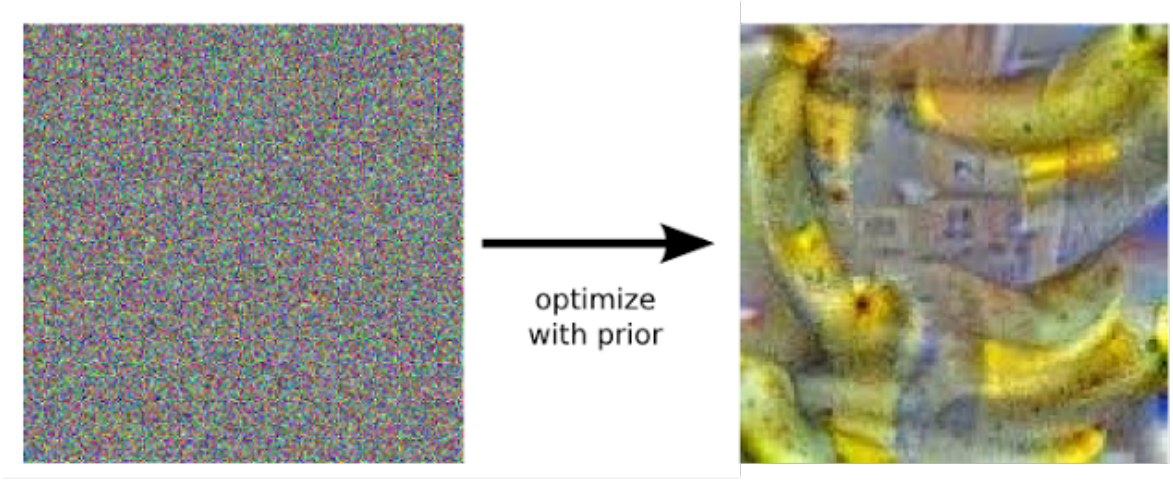



If the scale of the perturbation that looks like random noise is small enough compared to the scale of the original pixel values, humans would not notice a difference, but flexible enough classifier will change its decision

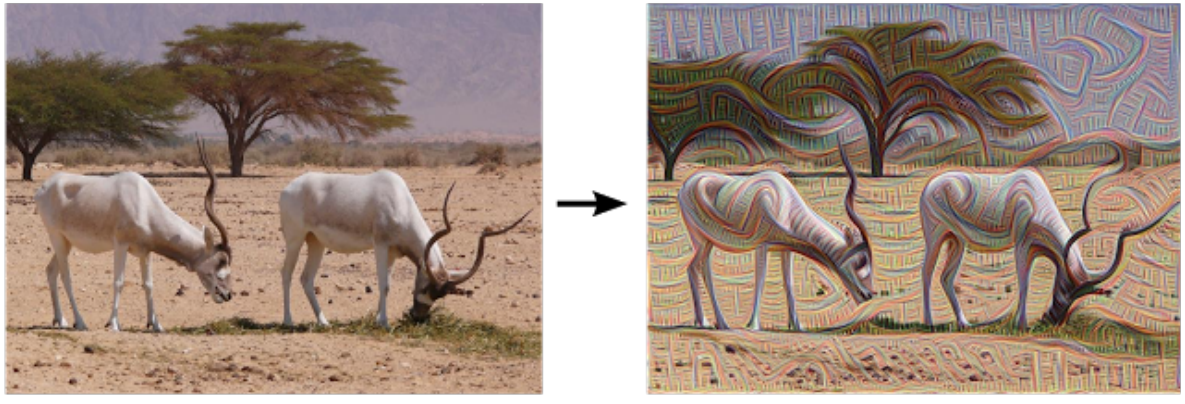
Find input $\mathbf{x}'$ so that it visually looks exactly like $\mathbf{x}$ but has different class prediction – we can use gradient-based optimization wrt to $\mathbf{x}$ instead of the weights to force the output to any class

Inceptionism

The same trick can be used to make deep networks "dream": Use noise as input for the network and then modify the input until it corresponds to a given class label



Images from



http://googleresearch.blogspot.fi/2015/06/inceptionism-going-deeper-into-neural.html

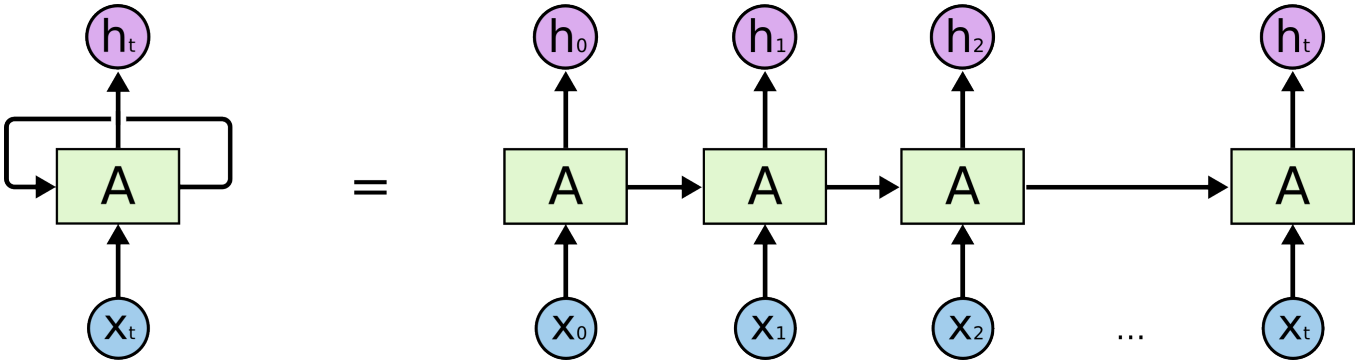
Recurrent neural networks

Until now we only considered feedforward links in the network, which correspond to directed acyclic graphs (DAG)

Any network with cycles is called *recurrent neural network* (RNN)

RNNs have memory – they somehow represent the previous training samples implicitly in the activations – and are good for sequential data

Can be thought of as infinitely deep feedforward network, unrolled as copies of itself



Picture from `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

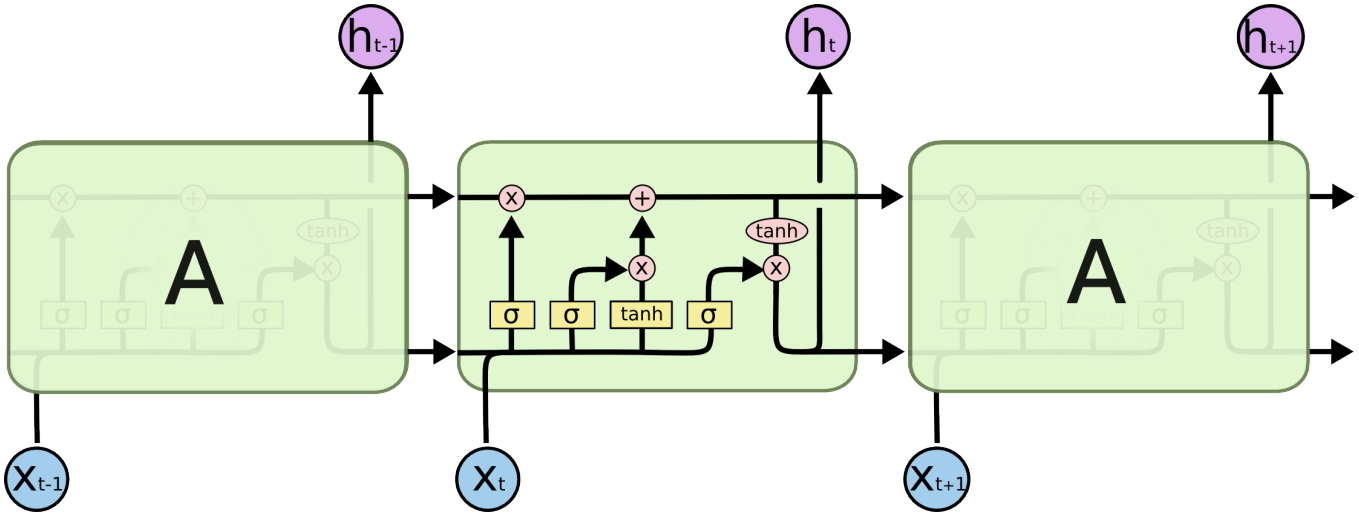
Recurrent neural networks

Training by *backpropagation through time*, which is simply backpropagation for the unrolled network

Makes all the problems related to backpropagation much more severe – especially vulnerable to effects of the inputs for outputs far in the future

Long short term memory (LSTM)

LSTM is one practical RNN model which explicitly remembers the past values



- The *forgetting gate* determines how much of the previous cell state is forgotten by multiplying with $[0, 1]$
- The *input gate* determines which values we should modify, and a separate tanh-based layer indicates the new values
- The new cell state is obtained by summing these up, and the output is then another non-linear function of the state and input

Picture from `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

Unsupervised deep learning

The previous examples all considered supervised learning, but neural networks and deep learning match well also unsupervised tasks

Some interesting connections between unsupervised and supervised too:

- Unsupervised techniques can be used for initialization or *pre-training* of supervised models
- Models intended for unsupervised use are sometimes trained in supervised fasion – if we learnt features that are good at classifying images then perhaps they are a good representation for the imaages

Autoencoders

Remember the PCA formulation as lower-dimensional projection that has low *re-construction error*

It is a special case of more general concept of *autoencoder*, a network that learns how to best re-construct the inputs

The information needs to flow through a bottleneck of some sort – otherwise we could just copy the inputs as such

For PCA the bottleneck if the rank constraint (number of dimensions retained) – if it matches the dimensionality then we indeed reach zero re-construction error

Autoencoders

Any neural network trained using the inputs as outputs is an autoencoder

Often the first part of the network that compresses the inputs somehow is called *encoder* and the second part is *decoder*, whereas the narrowest part of the representation is the *code* – this matches the compression terminology

Denoising autoencoders

A neat trick is to train autoencoders so that we feed in noisy versions of the vectors but still use the originals as outputs

The network learns to de-noise the noisy inputs and often solves the original autoencoding task more accurately as well

With denoising autoencoders we can actually use codes that are larger than the input

Sequence-to-sequence autoencoders

What if we use recurrent networks as encoders and decoders?

The input and output are sequences of arbitrary length (running text etc) but the representation is of fixed dimensionality

State-of-the-art machine translation is done largely this way, though the output is in another language

Layer-wise pretraining

One starting point for the deep learning hype was introduction of the concept ...

If we cannot train a deep model directly, perhaps training each layer alone would help

Stacked RBMs, pre-training for supervised models

Generative training followed by supervised fine-tuning

Neat idea, but not really used nowadays – the supervised training algorithm are good enough to work out of the box for good enough initialization

Supervised representation learning

Supervised deep networks often learn interesting internal representations for the hidden layers

We can use those as "unsupervised" feature extraction for other tasks

Overfeat is a pre-trained image classifier; if you need vectorial representations for images – for any purpose – you can just pick the last hidden layer of that network

Even better representations obtained if we use richer supervision; deep learning is especially suitable for multi-task learning