PATTERN RECOGNITION AND MACHINE LEARNING

Slide Set 6: Neural Networks and Deep Learning

February 2017

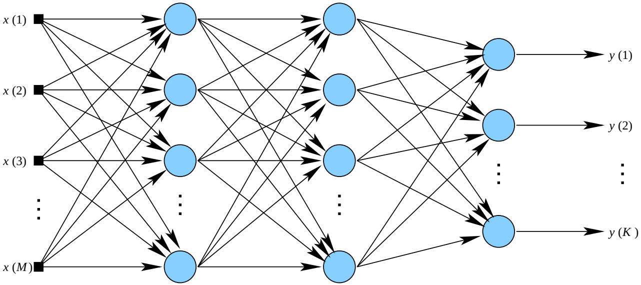
Heikki Huttunen

heikki.huttunen@tut.fi

Department of Signal Processing
Tampere University of Technology

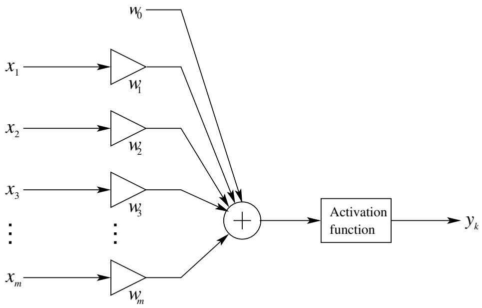
Traditional Neural Networks

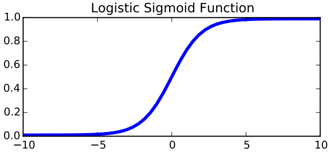
- Neural networks have been studied for decades.
- Traditional networks were *fully connected* (also called *dense*) networks consisting of typically 1-3 layers.
- Input dimensions were typically in the order of few hundred from a few dozen categories.
- Today, input may be 10k...100k variables from 1000 classes and network may have over 1000 layers.

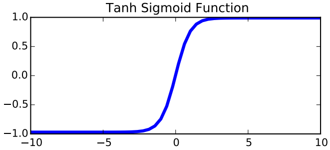


Traditional Neural Networks

- The neuron of a vanilla network is illustrated below.
- In essence, the neuron is a dot product between the inputs $\mathbf{x} = (1, x_1, \ldots, x_n)$ and weights $\mathbf{w} = (w_0, w_1, \ldots, w_n)$ followed by a nonlinearity, most often *logsig* or *tanh*.



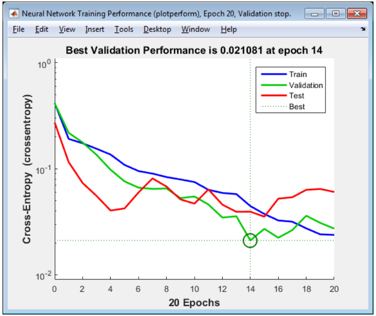


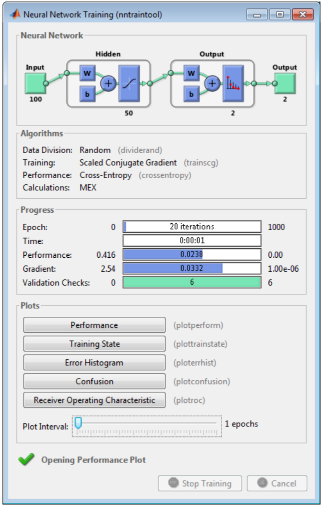


- In other words: this is *logistic regression* model, and the full net is just a stack of logreg models.

Training the Net

- Earlier, there was a lot of emphasis on training algorithms: *conjugate gradient, Levenberg-Marquardt, etc.*
- Today, people mostly use *stochastic gradient descent*.



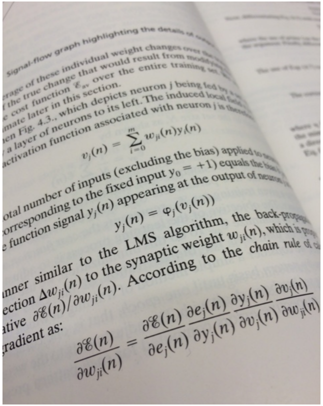


Backpropagation

- The network is trained by adjusting the weights according to the partial derivatives

$$w_{ij} \leftarrow w_{ij} - \eta \frac{\partial \mathcal{E}}{\partial w_{ij}}$$

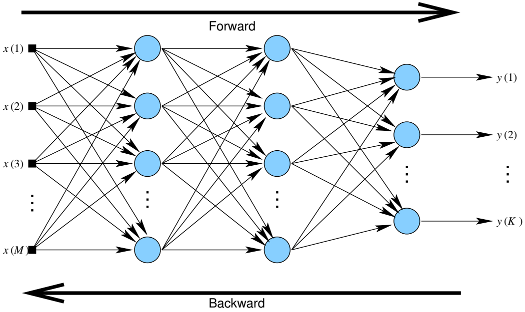
- In other words, the j^{th} weight of the i^{th} node steps towards the negative gradient with step size $\eta > 0$.
- In the 1990's the network structure was rather fixed, and the formulae would be derived by hand.
- Today, the same principle applies, but the exact form is computed symbolically.



Backpropagation in Haykin: Neural networks, 1999.

Forward and Backward

- Training has two passes: *forward pass* and *backward pass*.
- The forward pass feeds one (or more) samples to the net.
- The backward pass computes the (mean) error and propagates the gradients back adjusting the weights one at a time
- When all samples are shown to the net, one *epoch* has passed. Typically the network runs for thousands of epochs.

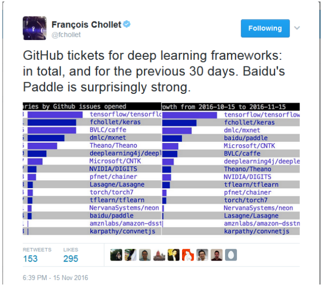


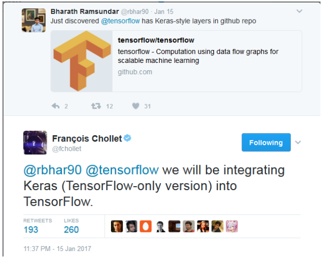
Neural Network Software

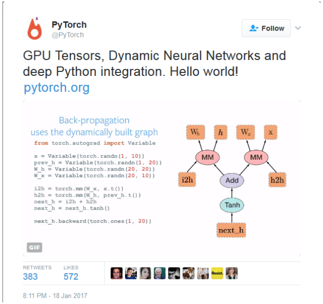
- Several packages exist:
  - **Matlab NN Toolbox**: Obsolete.
  - **Caffe**: C++ / CUDA with Python and Matlab interfaces
  - **Theano**: Python based CUDA engine; several front ends available: e.g., **Keras** and Lasagne.
  - **Torch**: Library implemented in Lua language (Facebook). Also pyTorch interface exists since Jan 2017.
  - **TensorFlow**: Google deep learning engine. Open sourced in Nov 2015. Supported by **Keras**, which will be the default interface.
  - Others: VELES (Samsung), Minerva,...
  - Most use Nvidia **cuDNN** middle layer.
- The important ones:
  - **Caffe** is very fast and default in image recognition. Good Python and Matlab interface.
  - **Torch** has a large user base and lot of momentum from Facebook.
  - **Keras** is very flexible and readable full-python interface to Theano and Tensorflow. **This is our choice for this course.**
  - **http://keras.io/** and **https://github.com/fchollet/keras**

Current Trends

- Python has become the language of machine learning.







Train a 2-layer Network with Keras

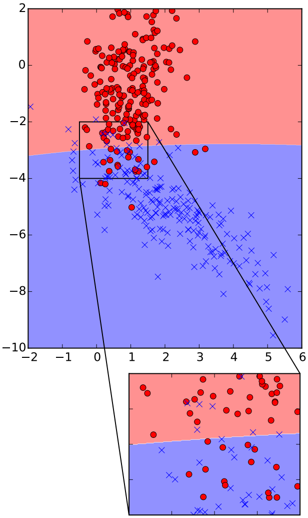
```
# Training code:
from keras.models import Sequential
from keras.layers.core import Dense, Activation

# First we initialize the model. "Sequential" means there are no loops.
clf = Sequential()

# Add layers one at the time. Each with 100 nodes.
clf.add(Dense(100, input_dim=2, activation = 'sigmoid'))
clf.add(Dense(100, activation = 'sigmoid'))
clf.add(Dense(1, activation = 'sigmoid'))

# The code is compiled to CUDA or C++
clf.compile(loss='mean_squared_error', optimizer='sgd')
clf.fit(X, y, nb_epoch=20, batch_size=16) # takes a few seconds
```

```
# Testing code:
# Probabilities
>>> clf.predict(np.array([[1, -2], [-3, -5]]))
array([[ 0.50781795],
       [ 0.48059484]])
# Classes
>>> clf.predict(np.array([[1, -2], [-3, -5]])) > 0.5
array([[ True],
       [False]], dtype=bool)
```



Deep Learning

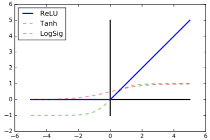
- The neural network research was rather silent after the rapid expansion in the 1990's.
- The hot topic of 2000's were, *e.g.*, the SVM and *big data*.
- However, at the end of the decade, neural networks started to gain popularity again: A group at Univ. Toronto led by Prof. Geoffrey Hinton studied unconventionally **deep** networks using *unsupervised* pretraining.
- He discovered that training of large networks was indeed possible with an unsupervised pretraining step that initializes the network weights in a layerwise manner.
- Another key factor to the success was the rapidly increased computational power brought by recent Graphics Processing Units (GPU's).

Unsupervised Pretraining

- There were two key problems why network depth did not increase beyond 2-3 layers:
  1. The error has huge **local minima areas** when the net becomes deep: Training gets stuck at one of them.
  2. The **gradient vanishes** at the bottom layers: The logistic activation function tends to decrease the gradient magnitude at each layer; eventually the gradient at the bottom layer is very small and they will not train at all.
- The former problem was corrected by **unsupervised** pretraining:
  - Train layered models that learned to *represent* the data (no class labels, no classification, just try to learn to reproduce the data).
  - Initialize the network with the weights of the unsupervised model and train in a supervised setting.
  - Common tools: *restricted Boltzmann machine* (RBM), *deep belief network* (DBN), *autoencoders*, etc.

Back to Supervised Training

- After the excitement of deep networks was triggered, the study of fully supervised approaches started as well (purely supervised training is more familiar, well explored and less scary angle of approach).
- A few key discoveries avoid the need for pretraining:
  - New activation functions that better preserve the gradient over layers; most importantly the Rectified Linear Unit[a]: $\mathrm{ReLU}(x) = \max(0, x)$.
  - Novel weight initialization techniques; *e.g.,* Glorot initialization (aka. Xavier initialization) adjusts the initial weight magnitudes layerwise[b].
  - Dropout regularization; avoid overfitting by injecting noise to the network[c]. Individual neurons are shut down at random in the training phase.



[a] Glorot, Bordes, and Bengio. "Deep sparse rectifier neural networks."

[b] Glorot and Bengio. "Understanding the difficulty of training deep feedforward neural networks."

[c] Srivastava, Hinton, Krizhevsky, Sutskever and Salakhutdinov. "Dropout: A simple way to prevent neural networks from overfitting."

Convolutional Layers

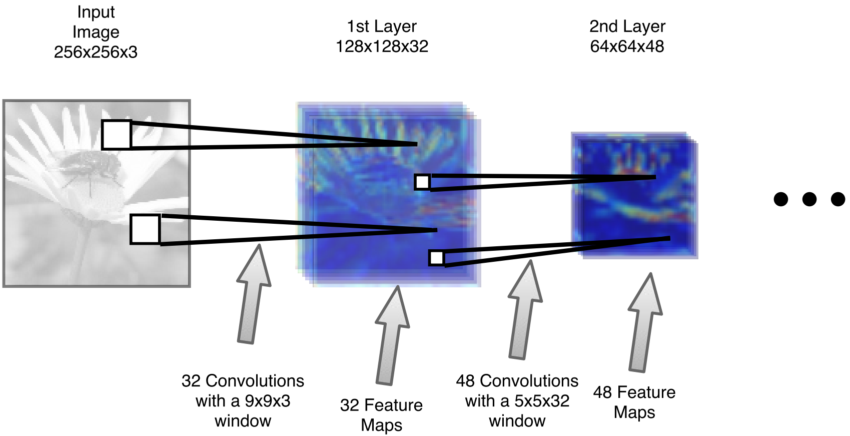
- In addition to the novel techniques for training, also new network architectures have been adopted.
- Most important of them is *convolutional layer*, which preserves also the topology of the input.
- Convolutional network was proposed already in 1989 but had a rather marginal role as long as image size was small (*e.g.,* 1990’s MNIST dataset of size 28×28 as compared to current ImageNet benchmark of size 256×256).

Convolutional Network

- The typical structure of a convolutional network repeats the following elements:
  convolution ⇒ **ReLU** ⇒ **subsampling**

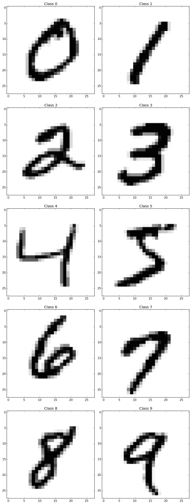
  1. **Convolution** filters the input with a number of convolutional kernels. In the first layer these can be, *e.g.*, $9 \times 9 \times 3$; *i.e.*, they see the local window from all RGB layers.
     - The results are called **feature maps**, and there are typically a few dozen of those.
  2. **ReLU** passes the feature maps through a pixelwise ReLU.
     - In numpy: `y = numpy.maximum(x, 0)`.
  3. **Subsampling** shrinks the input dimensions by an integer factor.
     - Originally this was done by averaging each 2×2 block.
     - Nowadays, **maxpooling** is more common (take max of each 2×2 block).
     - Subsampling reduces the data size and improves spatial invariance.

Convolutional Network



Convolutional Network: Example

- Let's train a convnet with the famous MNIST dataset.
- MNIST consists of 60000 training and 10000 test images representing handwritten numbers from US mail.
- Each image is 28×28 pixels and there are 10 categories.
- Generally considered an easy problem: Logistic regression gives over 90% accuracy and convnet can reach (almost) 100%.
- However, 10 years ago, the state of the art error was still over 1%.



Convolutional Network: Example

```
# Training code (modified from mnist_cnn.py at Keras examples)
from keras.datasets import mnist
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation,
      Flatten
from keras.layers.convolutional import Convolution2D,
      MaxPooling2D

# We use the handwritten digit database "MNIST".
# 60000 training and 10000 test images of
# size 28x28
(X_train, y_train), (X_test, y_test) = mnist.load_data()

num_featmaps = 32   # This many filters per layer
num_classes = 10    # Digits 0,1,...,9
num_epochs = 50     # Show all samples 50 times
w, h = 5, 5         # Conv window size
```

```
model = Sequential()

# Layer 1: needs input_shape as well.
model.add(Convolution2D(num_featmaps, w, h,
          input_shape=(1, 28, 28),
          activation = 'relu'))

# Layer 2:
model.add(Convolution2D(num_featmaps, w, h, activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

# Layer 3: dense layer with 128 nodes
# Flatten() vectorizes the data:
# 32x10x10 -> 3200
# (10x10 instead of 14x14 due to border effect)
model.add(Flatten())
model.add(Dense(128, activation = 'relu'))
model.add(Dropout(0.5))

# Layer 4: Last layer producing 10 outputs.
model.add(Dense(num_classes, activation='softmax'))

# Compile and train
model.compile(loss='categorical_crossentropy', optimizer='adadelta
      ')
model.fit(X_train, Y_train, nb_epoch=100)
```

Convolutional Network: Training Log

- The code runs for about 5-10 minutes on a GPU.
- On a CPU, this would take 1-2 hours (1 epoch ≈ 500 s)

```
Using gpu device 0: Tesla K40m
Using Theano backend.
Compiling model...
Model compilation took 0.1 minutes.
Training...
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============] - 31s - loss: 0.2193 - acc: 0.9322 - val_loss: 0.0519 - val_acc: 0.9835
Epoch 2/10
60000/60000 [==============] - 31s - loss: 0.0807 - acc: 0.9758 - val_loss: 0.0398 - val_acc: 0.9863
Epoch 3/10
60000/60000 [==============] - 31s - loss: 0.0581 - acc: 0.9825 - val_loss: 0.0322 - val_acc: 0.9898
Epoch 4/10
60000/60000 [==============] - 31s - loss: 0.0500 - acc: 0.9851 - val_loss: 0.0276 - val_acc: 0.9913
Epoch 5/10
60000/60000 [==============] - 31s - loss: 0.0430 - acc: 0.9872 - val_loss: 0.0287 - val_acc: 0.9906
Epoch 6/10
60000/60000 [==============] - 31s - loss: 0.0387 - acc: 0.9882 - val_loss: 0.0246 - val_acc: 0.9922
Epoch 7/10
60000/60000 [==============] - 31s - loss: 0.0352 - acc: 0.9897 - val_loss: 0.0270 - val_acc: 0.9913
Epoch 8/10
60000/60000 [==============] - 31s - loss: 0.0324 - acc: 0.9902 - val_loss: 0.0223 - val_acc: 0.9928
Epoch 9/10
60000/60000 [==============] - 31s - loss: 0.0294 - acc: 0.9907 - val_loss: 0.0221 - val_acc: 0.9926
Epoch 10/10
60000/60000 [==============] - 31s - loss: 0.0252 - acc: 0.9922 - val_loss: 0.0271 - val_acc: 0.9916
Training (10 epochs) took 5.8 minutes.
```

Save and Load the Net

- The network can be saved to disk in two parts:
  - Network topology as JSON or YAML: `model.to_json()` or `model.to_yaml()`. The resulting string can be written to disk using `.write()` of a file object.
  - Coefficients are saved in HDF5 format using `model.save_weights()`. HDF5 is a serialization format similar to `.mat` or `.pkl`
- Alternatively, the net can be pickled, although this is not recommended.
- Read back to memory using `model_from_json` and `load_weights`


```yaml
class_mode: categorical
layers:
- W_constraint: null
  W_regularizer: null
  activation: relu
  activity_regularizer: null
  b_constraint: null
  b_regularizer: null
  border_mode: valid
  dim_ordering: th
  init: glorot_uniform
  input_shape: !!python/tuple [1, 28, 28]
  name: Convolution2D
  nb_col: 5
  nb_filter: 32
  nb_row: 5
  subsample: &id001 !!python/tuple [1, 1]
- W_constraint: null
  W_regularizer: null
  activation: relu
  activity_regularizer: null
  b_constraint: null
  b_regularizer: null
  border_mode: valid
  dim_ordering: th
  init: glorot_uniform
  name: Convolution2D
  nb_col: 5
  nb_filter: 32
  nb_row: 5
  subsample: *id001
- border_mode: valid
  dim_ordering: th
  name: MaxPooling2D
  pool_size: &id002 !!python/tuple [2, 2]
  strides: *id002
- {name: Dropout, p: 0.25}
- {name: Flatten}
```


Part of network definition in YAML

Network Structure

- It is possible to look into the filters on the convolutional layers.

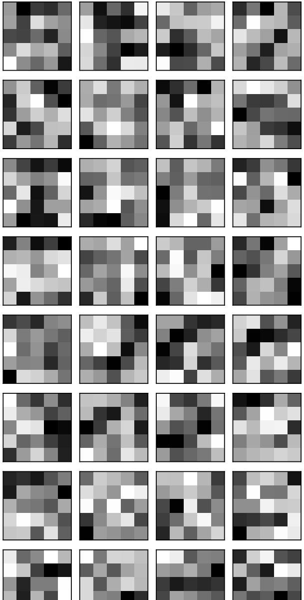
```
# First layer weights (shown on the right):
weights = model.layers[0].get_weights()[0]
```

- The second layer is difficult to visualize, because the input is 32-dimensional:

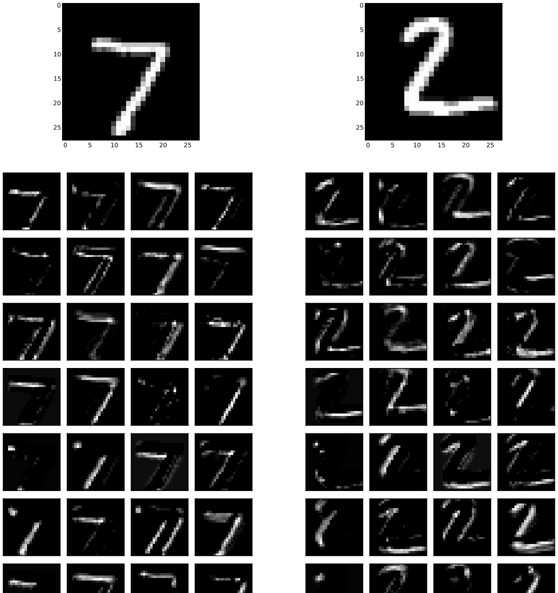
```
# Zeroth layer weights:
>>> model.layers[0].get_weights()[0].shape
(32, 1, 5, 5)
# First layer weights:
>>> model.layers[1].get_weights()[0].shape
(32, 32, 5, 5)
```

- The dense layer is the 5th (conv → conv → maxpool → dropout → flatten → dense).

```
# Fifth layer weights map 3200 inputs to 128 outputs.
# This is actually a matrix multiplication.
>>> model.layers[5].get_weights()[0].shape
(3200, 128)
```



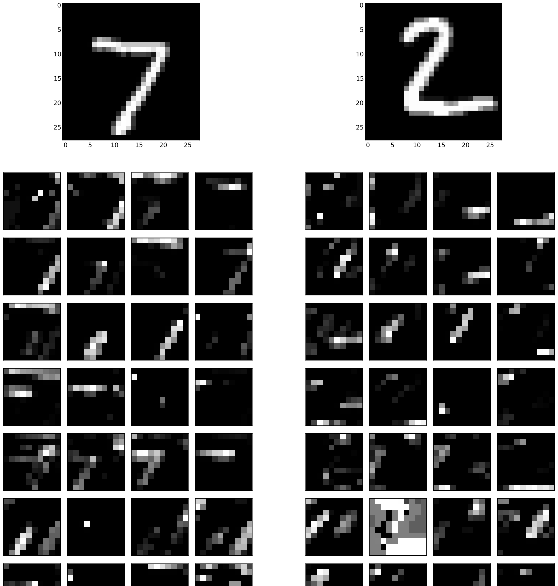
Network Activations



- The layer outputs are usually more interesting than the filters.
- These can be visualized as well.
- For details, see Keras FAQ.

Second Layer Activations

- On the next layer, the figures are downsampled to 12x12.
- This provides *spatial invariance*: The same activation results although the input would be slightly displaced.



DEEP LEARNING HIGHLIGHTS OF 2015-2016

2015 was Full of Breakthroughs:
Let’s See Some of Them