Tampere University

PATTERN RECOGNITION AND MACHINE LEARNING

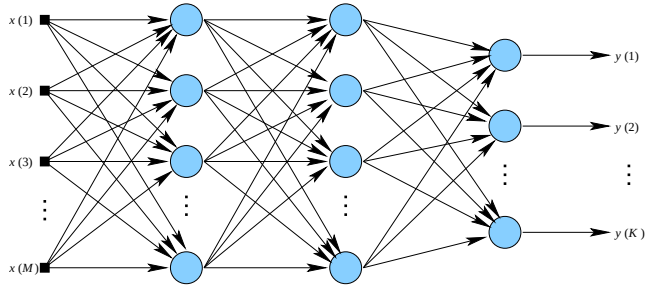
Slide Set 5: Neural Networks and Deep Learning

November 2019

Heikki Huttunen
heikki.huttunen@tuni.fi
Signal Processing
Tampere University

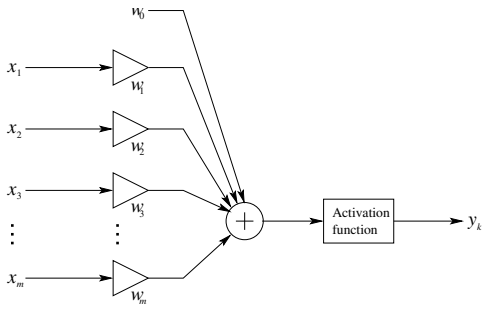
Traditional Neural Networks

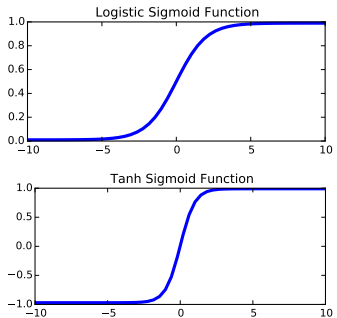
- Neural networks have been studied for decades.
- Traditional networks were *fully connected* (also called *dense*) networks consisting of typically 1-3 layers.
- Input dimensions were typically in the order of few hundred from a few dozen categories.
- Today, input may be 10k...100k variables from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

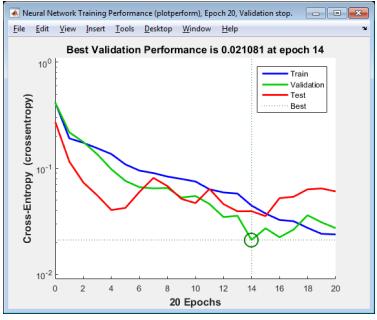


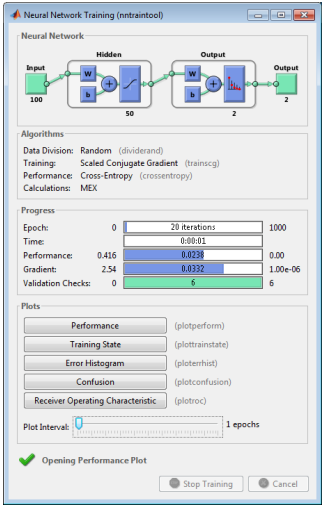


- In other words: this is *logistic regression* model, and the full net is just a stack of logreg models.

Training the Net

- Earlier, there was a lot of emphasis on training algorithms: *conjugate gradient, Levenberg-Marquardt, etc.*
- Today, the optimizers are less mathematical: *stochastic gradient descent*, *RMSProp*, *Adam*.



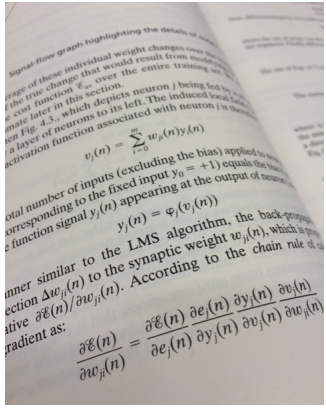


Backpropagation

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

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

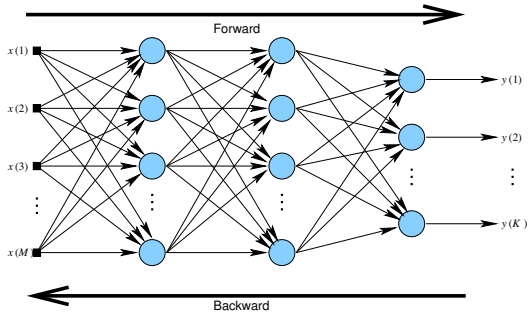
- In other words, the j^{th} weight of the i^{th} node steps towards the negative gradient with step size $\eta > 0$.
- In the 1990's the network structure was rather fixed, and the formulae would be derived by hand.
- Today, the same principle applies, but the exact form is computed symbolically.



Backpropagation in Haykin: Neural networks, 1999.

Forward and Backward

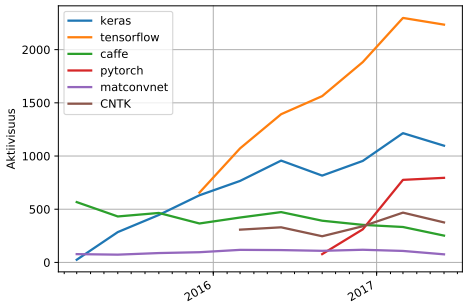
- Training has two passes: *forward pass* and *backward pass*.
- The forward pass feeds one (or more) samples to the net.
- The backward pass computes the (mean) error and propagates the gradients back adjusting the weights one at a time
- When all samples are shown to the net, one *epoch* has passed.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

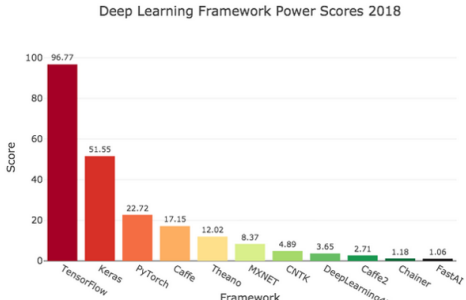


Neural Network Software

- **TensorFlow**: Google deep learning engine. Open sourced in Nov 2015.
  - Supported by **Keras**, which is integrated to TF since TF version 2.0.
- **MS Cognitive TK (CNTK)**: Microsoft deep learning engine.
  - Supported by **Keras**
- **mxnet**: Scalable deep learning (e.g. Android). First in multi-gpu. Many contributors.
  - Supported by **Keras** (in beta).
- **PlaidML**: OpenCL backend. Supported by **Keras**.
  - Supported by **Keras**
- **Torch**: Library implemented in Lua language (Facebook). Python interface via pyTorch.
- All but PlaidML use Nvidia **cuDNN** middle layer.

Popularity of deep learning platforms





Credits: Jeff Hale / TowardsDataScience

Train a 2-layer Network with Keras

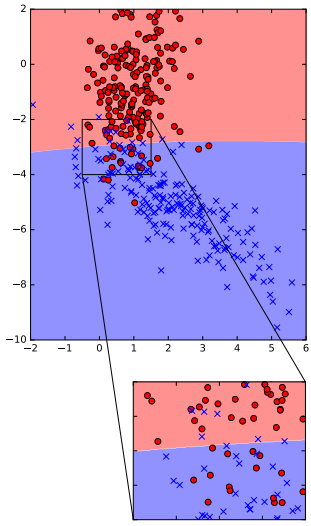
```
# Training code:
import tensorflow as tf

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

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

# The code is compiled to CUDA or C++
clf.compile(loss='mean_squared_error', optimizer='sgd')
clf.fit(X, y, epochs = 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

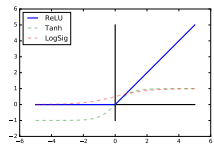
- The neural network research was rather silent after the rapid expansion in the 1990's.
- The hot topic of 2000's were, *e.g.,* the SVM and *big data*.
- However, at the end of the decade, neural networks started to gain popularity again: A group at Univ. Toronto led by Prof. Geoffrey Hinton studied 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.
- 10 years ago, it was discovered that both problems could be 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]: $\text{ReLU}(x) = \max(0, x)$.
  - 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."

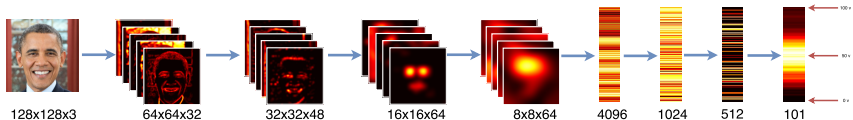
Convolutional Layers

- 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

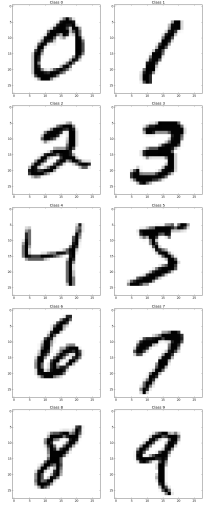
- The typical structure of a convolutional network repeats the following elements:
  convolution $\Rightarrow$ **nonlinearity** $\Rightarrow$ **subsampling**

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 *Rectified Linear Unit*.
   - $\mathrm{ReLU}(x) = \max(0, x)$.
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: 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

from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation, Flatten,
       Conv2D, MaxPooling2D
from tensorflow.keras.utils import to_categorical
import numpy as np
import os

# 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()

# Keras assumes 4D input, but MNIST is lacking color channel.
# -> Add a dummy dimension at the end.

X_train = X_train[..., np.newaxis] / 255.0
X_test  = X_test[..., np.newaxis] / 255.0

# Output has to be one-hot-encoded
y_train = to_categorical(y_train)
y_test  = to_categorical(y_test)

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(Conv2D(num_featmaps, (w, h),
          input_shape=(28, 28, 1),
          activation = 'relu'))

# Layer 2:
model.add(Conv2D(num_featmaps, (w, h), activation = 'relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

# 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'))

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

# Compile and train
model.compile(loss='categorical_crossentropy',
              optimizer='adam',
              metrics = ['accuracy'])

model.fit(X_train, y_train, epochs = 10,
          validation_data = (X_test, y_test))
```

Convolutional Network: Training Log

- The code runs for about a minute on a GPU.
- On a CPU, this would take about 30 minutes (1 epoch ≈ 200 s)

```
Using gpu device 0: Tesla P100
Training ...
Train on 60000 samples, validate on 10000 samples
Epoch 1/10
60000/60000 [==============================] – 10s 159us/sample – loss: 0.1043 – accuracy: 0.9683 – val_loss: 0.0367 – val_accuracy: 0.9883
Epoch 2/10
60000/60000 [==============================] – 7s 118us/sample – loss: 0.0371 – accuracy: 0.9883 – val_loss: 0.0391 – val_accuracy: 0.9871
Epoch 3/10
60000/60000 [==============================] – 7s 119us/sample – loss: 0.0251 – accuracy: 0.9918 – val_loss: 0.0264 – val_accuracy: 0.9920
Epoch 4/10
60000/60000 [==============================] – 7s 119us/sample – loss: 0.0175 – accuracy: 0.9944 – val_loss: 0.0281 – val_accuracy: 0.9916
Epoch 5/10
60000/60000 [==============================] – 7s 117us/sample – loss: 0.0132 – accuracy: 0.9957 – val_loss: 0.0354 – val_accuracy: 0.9899
Epoch 6/10
60000/60000 [==============================] – 7s 119us/sample – loss: 0.0101 – accuracy: 0.9969 – val_loss: 0.0362 – val_accuracy: 0.9895
Epoch 7/10
60000/60000 [==============================] – 7s 118us/sample – loss: 0.0083 – accuracy: 0.9973 – val_loss: 0.0504 – val_accuracy: 0.9896
Epoch 8/10
60000/60000 [==============================] – 7s 117us/sample – loss: 0.0070 – accuracy: 0.9977 – val_loss: 0.0442 – val_accuracy: 0.9905
Epoch 9/10
60000/60000 [==============================] – 7s 118us/sample – loss: 0.0065 – accuracy: 0.9977 – val_loss: 0.0408 – val_accuracy: 0.9907
Epoch 10/10
60000/60000 [==============================] – 7s 117us/sample – loss: 0.0070 – accuracy: 0.9976 – val_loss: 0.0444 – val_accuracy: 0.9906
Training took 1.2 minutes.
```

Save and Load the Net

- The network can be saved and loaded to disk in a straightforward manner:
  - **Saving:**

```
model.save("my_net.h5")
```

  - **Loading:**

```
from keras.models import load_model
load_model("my_net.h5")
```

- Network is saved in HDF5 format. HDF5 is a serialization format similar to `.mat` or `.pkl` although a lot more efficient.
- Use HDF5 for data storage with `h5py`.

```
# Save np.array X to h5 file:
import h5py
with h5py.File("my_data.h5", "w") as h5:
   h5["X"] = X
```

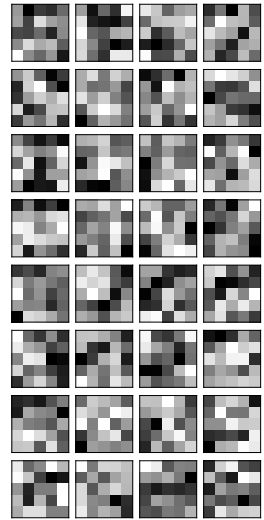
```
# Load np.array X from h5 file:
import h5py
with h5py.File("my_data.h5", "r") as h5:
   X = np.array(h5["X"])

# Note: Don't cast to numpy unless necessary.
# Data can be accessed from h5 directly.
```

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:

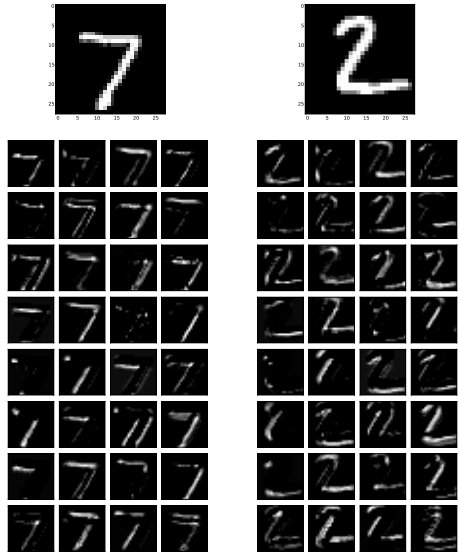
```
# Zeroth layer weights:
>>> model.layers[0].get_weights()[0].shape
(32, 1, 5, 5)
# First layer weights:
>>> model.layers[1].get_weights()[0].shape
(32, 32, 5, 5)
```

- The dense layer is the 5th (conv $\rightarrow$ conv $\rightarrow$ maxpool $\rightarrow$ dropout $\rightarrow$ flatten $\rightarrow$ 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.