1. Introduction to Deep Learning

Holzinger Philipp

1. What is TensorFlow?
2. Building a Graph
3. Basic Training and Inference
    1. Neural Network Fundamentals
    2. The MNIST Dataset
    3. Using Fully Connected Layers
4. Convolutional Neural Networks (CNN)
    1. CNN Motivation
    2. Convolution Basics
    3. Using Convolutional Layers
    4. LeNet

1.1 What is TensorFlow?

“TensorFlow™ is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices. Originally developed by researchers and engineers from the Google Brain team within Google’s AI organization, it comes with strong support for machine learning and deep learning and the flexible numerical computation core is used across many other scientific domains.”

- https://www.tensorflow.org/

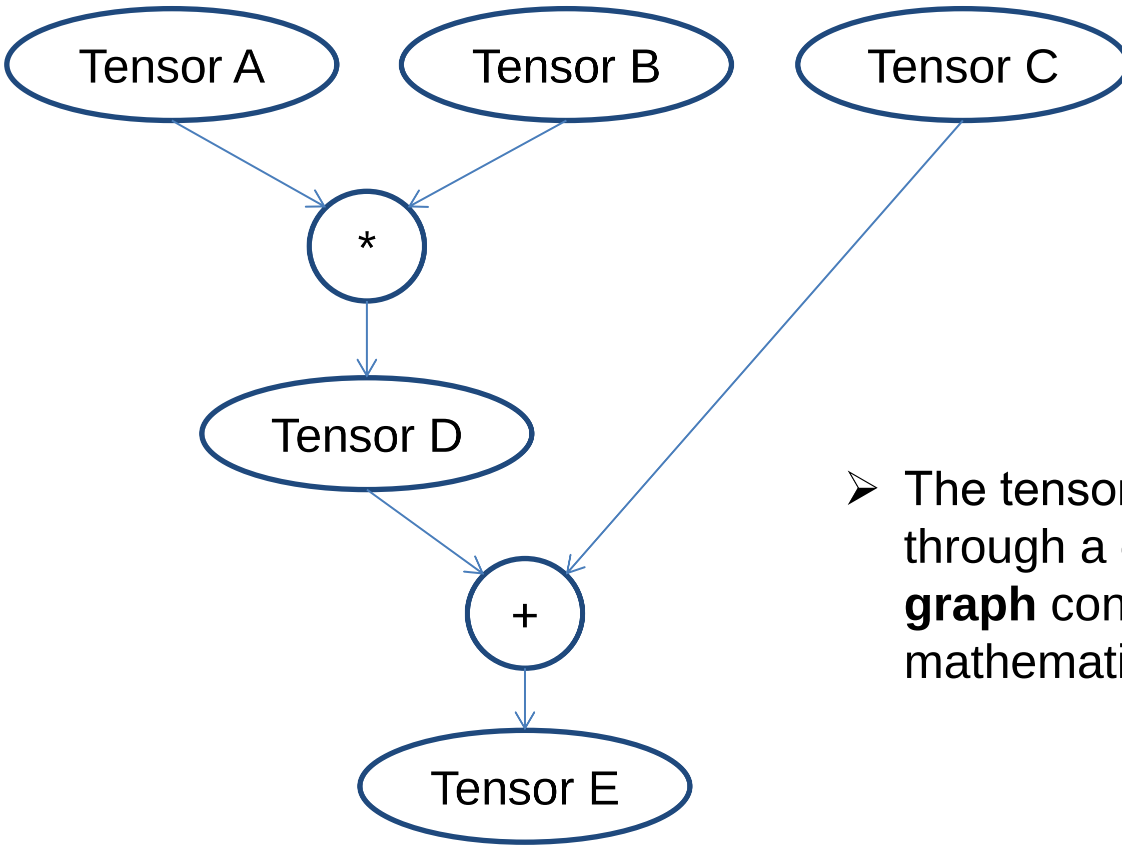
What is a tensor?

A **tensor** is an arbitrarily complex geometric object that maps in a (multi-)linear manner geometric […] tensors to a resulting tensor. Thereby, vectors and scalars themselves […] are considered as the simplest tensors.

- https://en.wikipedia.org/wiki/Tensor

a	$\begin{pmatrix} a \\ b \\ c \end{pmatrix}$	$\begin{pmatrix} a & b & c \\ d & e & f \\ g & h & i \end{pmatrix}$
Scalar Order 0 tensor	Vector Order 1 tensor	Matrix Order 2 tensor

What is flowing?



- The tensors are "flowing" through a **computational graph** containing a series of mathematical operations

1.2 Building a Graph

Note:

- Python 3 is used as programming language
- To better understand the fundamental principals of deep neural networks we use the internal low-level TensorFlow 1.x API in this presentation which is closer to the actual C++ code
  - In later versions these functions have been moved to the *tf.compat.v1* namespace (we omit this to shorten the code)
- In practice always use the high-level APIs like *"Keras"* to build and train deep learning models, which also became the default method since TensorFlow 2.x
  - An introduction to Keras can be found here: https://keras.io/guides/functional_api/

Differences between imperative languages and the TensorFlow concept:

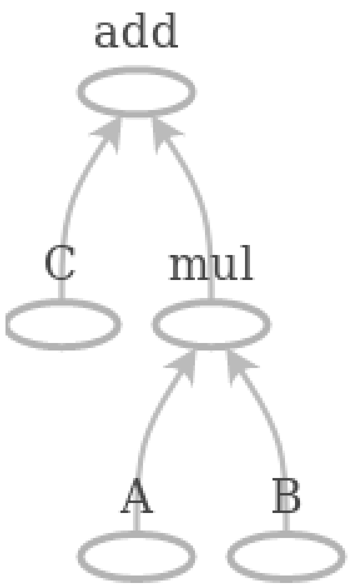
- Execution in TensorFlow is graph based
- The TensorFlow graph represents a series of operations to perform
- Typical TensorFlow operations like add, multiply or matmul modify the current graph and do NOT directly perform the operation

Therefore the TensorFlow workflow looks like this:

1. Build a TensorFlow graph with all needed operations
2. Feed the desired input values into the graph
3. Run the graph to get the result

```
import tensorflow as tf

# these are tensors of a graph
a = tf.placeholder(tf.float32, shape=(None), name='A')
b = tf.placeholder(tf.float32, shape=(None), name='B')
c = tf.placeholder(tf.float32, shape=(None), name='C')
d = tf.multiply(a,b, name='mul')
e = tf.add(c,d, name='add')

# open execution session
sess = tf.Session()

# this feeds tensors to the placeholders, runs the graph
# and stores the result (= last tensor)
result = sess.run(e, feed_dict={a: [1.2, 2.4], b: [1.0, 0.5],
c: [0.1, 0.2]})

print("the type of result is {}".format(type(result)))
print("the result value of the graph is {}".format(result))
```

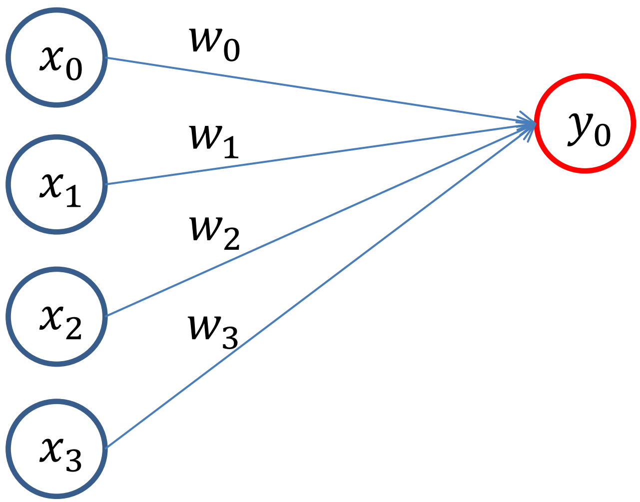
TensorBoard graph:



1.3 Basic Training and Inference

1.3.1 Neural Network Fundamentals

When does a single artificial neuron *y* fire?

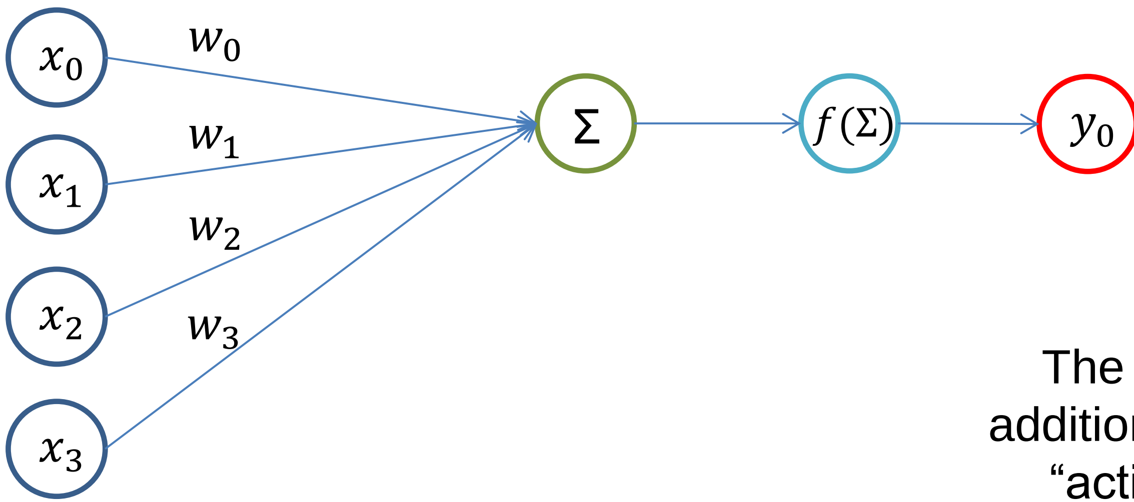


$$\begin{pmatrix} w_0 & w_1 & w_2 & w_3 \end{pmatrix} \cdot \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = y_0$$

- $\sum_{i=0}^{N-1} w_i \cdot x_i = y$
- $w \cdot x = y$ (scalar product)

x = input
w = weights
y = output

When does a single artificial neuron y fire?

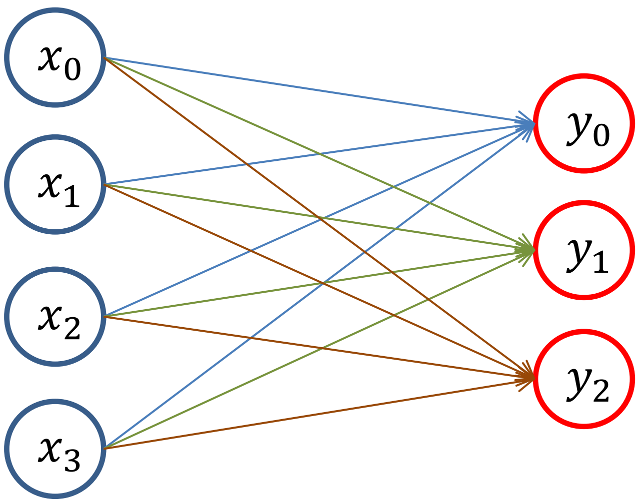


The value of y can be additionally modified by an "activation function" f

x = input
w = weights
y = output
$f(\Sigma)$ = activation function

- $f(\sum_{i=0}^{N-1} w_i \cdot x_i) = y$
- $f(w \cdot x) = y$

What is the activation of multiple artificial neurons?



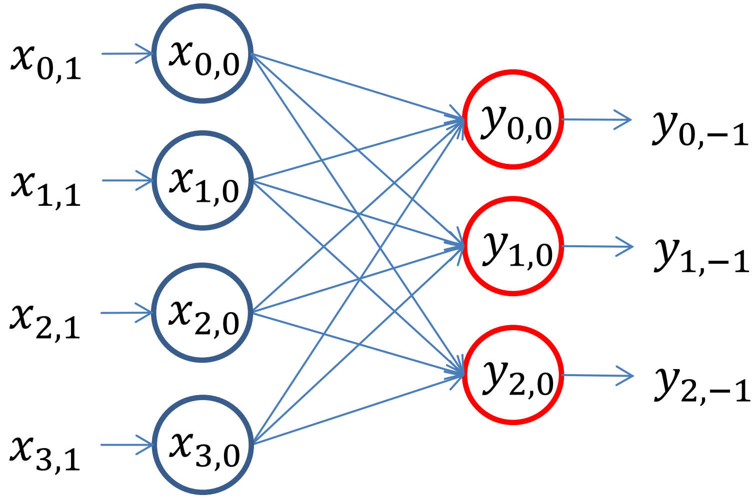
$$\begin{pmatrix} w_{0,0} & w_{0,1} & w_{0,2} & w_{0,3} \\ w_{1,0} & w_{1,1} & w_{1,2} & w_{1,3} \\ w_{2,0} & w_{2,1} & w_{2,2} & w_{2,3} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}$$

- $\sum_{i=0}^{N-1} w_{m,i} \cdot x_i = y_m$
- $Wx = y$ (matrix-vector product)

x = input
w = weights
y = output

What is the activation of successive impulses?

next input | current input | current output | previous output



$$\begin{pmatrix} w_{0,0} & \cdots & w_{0,3} \\ \vdots & \ddots & \vdots \\ w_{2,0} & \cdots & w_{2,3} \end{pmatrix} \begin{pmatrix} x_{0,0} & x_{0,1} \\ \vdots & \vdots \\ x_{3,0} & x_{3,1} \end{pmatrix} = \begin{pmatrix} y_{0,0} & y_{0,1} \\ y_{1,0} & y_{1,1} \\ y_{2,0} & y_{2,1} \end{pmatrix}$$

- $\sum_{i=0}^{N} w_{m,i} \cdot x_{i,b} = y_{m,b}$
- $WX = Y$ (matrix-matrix product)

x = input
w = weights
y = output

Final matrices which describe a “fully connected layer”:

$$\begin{pmatrix} w_{0,0} & \cdots & w_{0,N-1} \\ \vdots & \ddots & \vdots \\ w_{M-1,0} & \cdots & w_{M-1,N-1} \end{pmatrix} \begin{pmatrix} x_{0,0} & \cdots & x_{0,B-1} \\ \vdots & \ddots & \vdots \\ x_{N-1,0} & \cdots & x_{N-1,B-1} \end{pmatrix} = \begin{pmatrix} y_{0,0} & \cdots & y_{0,B-1} \\ \vdots & \ddots & \vdots \\ y_{M-1,0} & \cdots & y_{M-1,B-1} \end{pmatrix}$$

x = input

w = weight

y = output

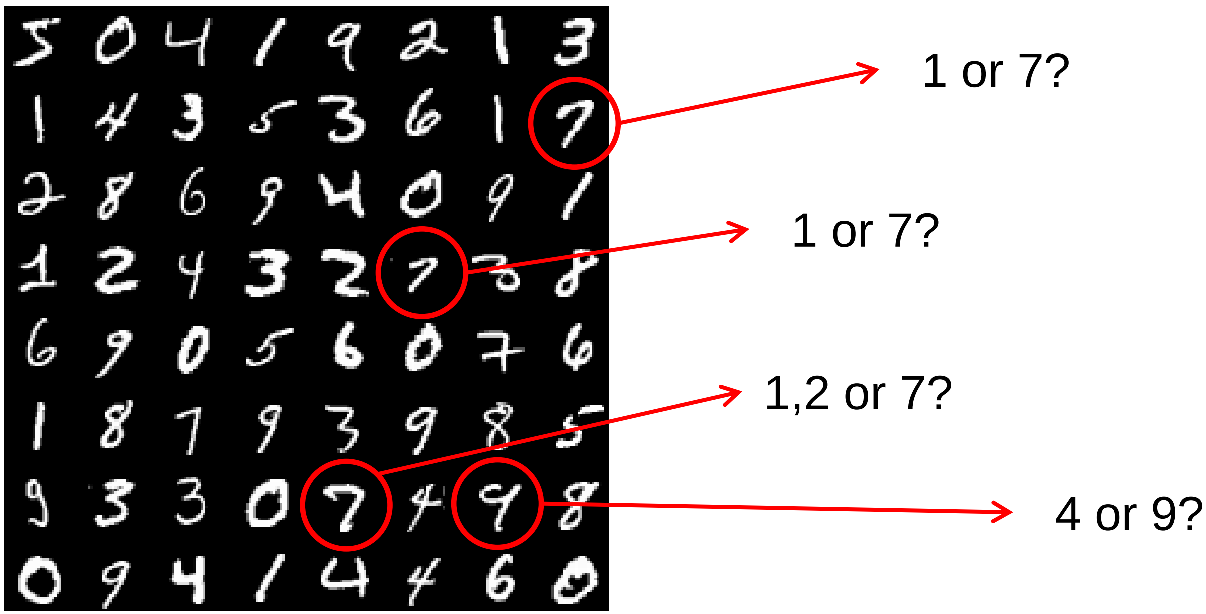
N = number of inputs

M = number of outputs

B = batch size

MNIST (Modified National Institute of Standards and Technology) **dataset** is a collection of handwritten digits, commonly used for training neural networks. It contains 60,000 training images and 10,000 testing images

MNIST (Modified National Institute of Standards and Technology) **dataset** is a collection of handwritten digits, commonly used for training neural networks. It contains 60,000 training images and 10,000 testing images



Some numbers are even hard for humans to classify…

Using a fully connected layer to build the first graph for an MNIST image:

- All MNIST Images have a resolution of 28×28 pixels
  - Input layer has $28 \cdot 28 = 784$ nodes
  - $N = 784$
- They represent a number between 0 and 9
  - Output layer has 10 nodes
  - $M = 10$
- First try connect all of them with one layer
  - $784 \cdot 10 = 7840$ float weights
  - $7840 \cdot 4 = 31360$ bytes of storage

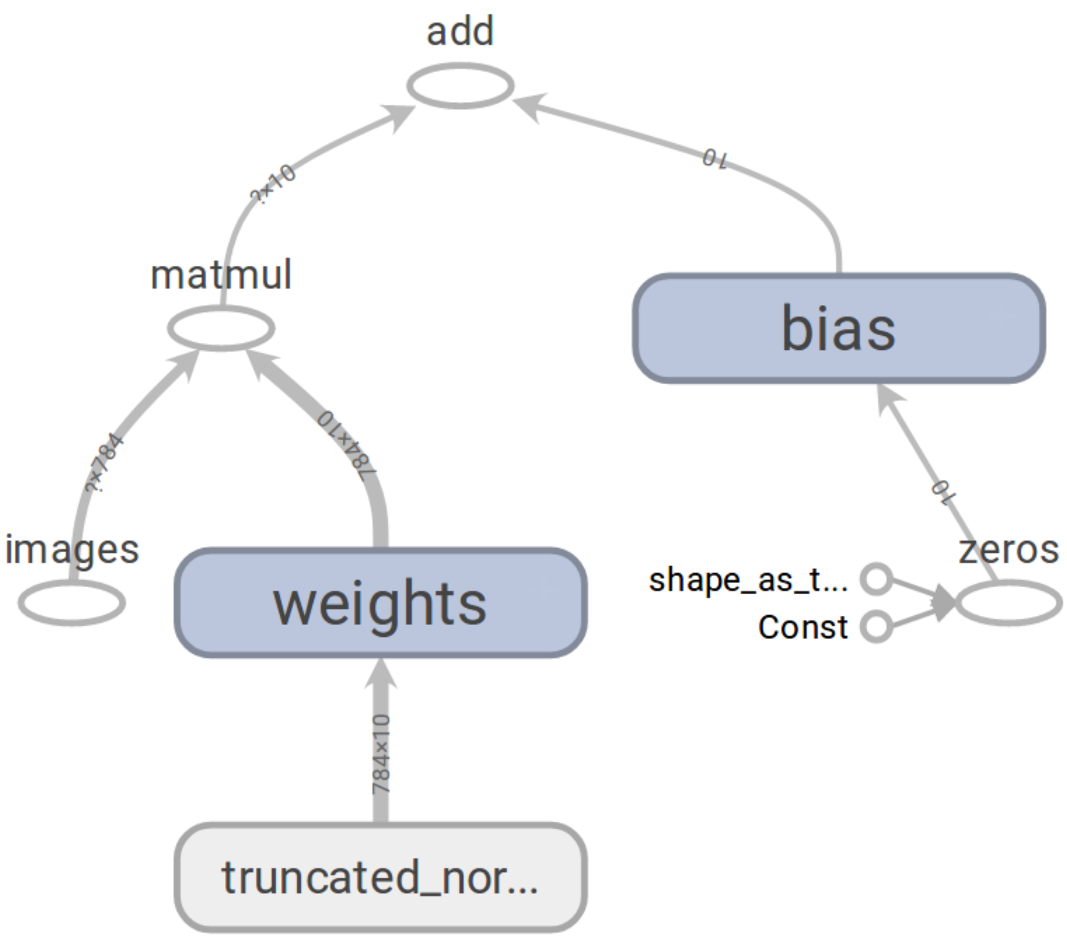
Import the MNIST data and iterate through it:

```
import tensorflow as tf
from official.mnist import dataset

# download dataset and store reference in variable
ds_download_dir = "mnist_dataset"
test_ds = dataset.test(ds_download_dir) # different for every data set

# split data in batches
batch_size = 128
test_ds = test_ds.batch(batch_size)

# define iterator over batches of data
data_iterator =
tf.data.Iterator.from_structure(tf.data.get_output_types(test_ds),tf.data.get_output_shapes(test_ds))

# define graph operation which initializes the iterator with the dataset
test_init_op = data_iterator.make_initializer(test_ds)

# define graph operation which gets the next batch of the iterator over the dataset
next_data_batch = data_iterator.get_next()
```

Building the evaluation graph for an MNIST image:

```
# define initialization parameters
mu = 0
sigma = 0.1

# build evaluation graph with one fully connected layer
images = tf.placeholder(tf.float32, shape=(None,784),name='images')
weights = tf.Variable(tf.random.truncated_normal(shape=(784,10), mean=mu, stddev=sigma))
bias = tf.Variable(tf.zeros(10), name='bias')

input_layer = images
output_layer = tf.add(tf.matmul(input_layer, weights, name='matmul'), bias, name='add')
```

The described graph visualized with TensorBoard:



Building the graph to calculate the accuracy of a batch:

```
# feed the correct labels into the net
labels = tf.placeholder(tf.int32, (None), name='labels')

# highest value is the guess of the network
network_prediction = tf.argmax(output_layer,axis=1,output_type=tf.int32)

# return 0.0 if net prediction is wrong, 1.0 if true
is_correct_prediction = tf.equal(network_prediction, labels)

# percentage of correct predictions is the mean of the batch
accuracy_op = tf.reduce_mean(tf.cast(is_correct_prediction, tf.float32))
```

Running the graphs:

```
# initialize all variables before evaluating the graph
sess = tf.Session()
sess.run(tf.global_variables_initializer())

# initialize the batch iterator
sess.run(test_init_op)

# loop over all batches and calculate predictions and accuracy
while True:
    try:
        data_batch = sess.run(next_data_batch)
        image_batch = data_batch[0]
        label_batch = data_batch[1]
        logits = sess.run(output_layer, feed_dict={images:image_batch})
        accuracy = sess.run(accuracy_op, feed_dict={images:image_batch, labels:label_batch})
    except tf.errors.OutOfRangeError:
        break
```

Accuracy of the graph in this state:

- accuracy: 0.104
- That means 10.4% of all guesses are correct
- 10 numbers must be distinguished (chance of $\frac{1}{10} = 10\%$ to randomly guess correctly)
- Network not much better than random values!
- The weights need to be trained first!

Defining the loss:

```
# define the loss graph
onehot_labels = tf.one_hot(labels, 10)
loss = tf.losses.mean_squared_error(labels=onehot_labels, predictions=output_layer)
```

Loss is the measuring unit of how “wrong” the results really are, e.g.:

Mean Squared Error (MSE):

$$MSE = \frac{1}{M} \sum_{i=0}^{M-1} (c_i - p_i)^2$$

- Useful for regression problems

Cross entropy (CE):

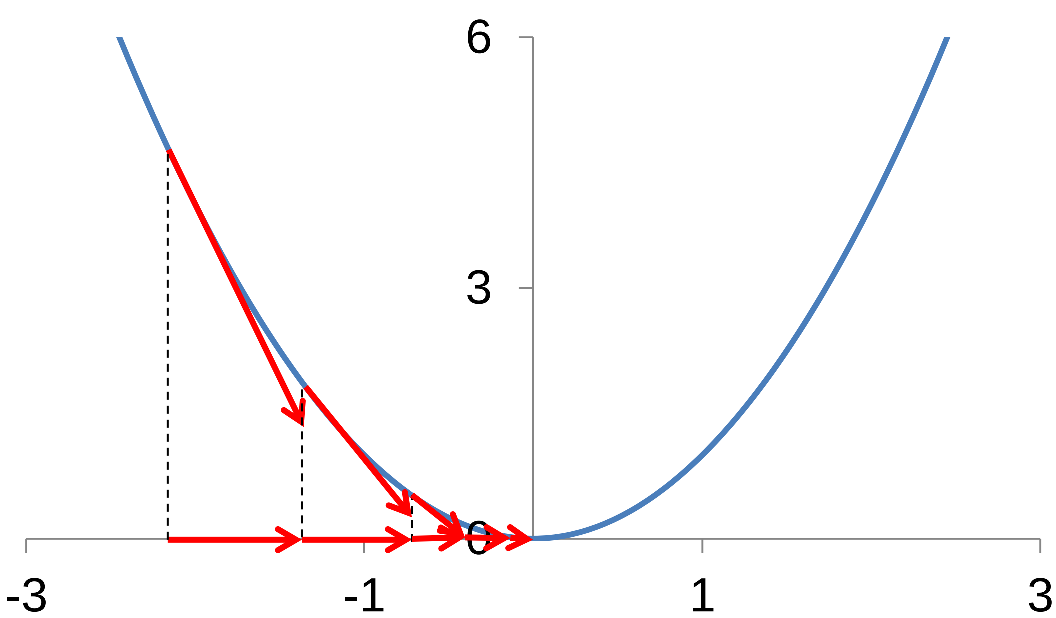
$$CE = -\sum_{i=0}^{M-1} c_i \cdot \log(p_i)$$

- Useful for classification problems

c_i = correct value (label) p_i = prediction of the network

Minimize the loss:

```
# define training parameters
learning_rate = 0.01

# define the training graph
optimizer = tf.train.GradientDescentOptimizer(learning_rate)
train_op = optimizer.minimize(loss)
```



Update of the weights:

$$W_{t+1} = W_t - \delta E_n X^\top$$

δ = learning rate

E_n = error tensor

Running the graphs (again):

```
# download and prepare training data as before
train_ds = dataset.train(ds_download_dir) # different for every data set
train_ds = train_ds.shuffle(60000).batch(batch_size)
train_init_op = data_iterator.make_initializer(train_ds)

# train the weights by looping repeatedly over all the data (and shuffling in between)
for i in range(epochs):
    sess.run(train_init_op)
    while True:
        try:
            data_batch = sess.run(next_data_batch)
            image_batch = data_batch[0]
            label_batch = data_batch[1]
            sess.run(train_op, feed_dict={images:image_batch, labels:label_batch})
        except tf.errors.OutOfRangeError:
            break

    # every epoch calculate predictions, accuracy and loss as before to check progress
    <…>
```

Progress during training (MSE loss):

Epoch 0 done: accuracy 0.369, loss 0.173
Epoch 1 done: accuracy 0.509, loss 0.126
Epoch 2 done: accuracy 0.581, loss 0.106
Epoch 3 done: accuracy 0.627, loss 0.095
Epoch 4 done: accuracy 0.658, loss 0.087
Epoch 5 done: accuracy 0.680, loss 0.082
Epoch 6 done: accuracy 0.701, loss 0.078
Epoch 7 done: accuracy 0.715, loss 0.074
Epoch 8 done: accuracy 0.726, loss 0.072
Epoch 9 done: accuracy 0.735, loss 0.069
Epoch 10 done: accuracy 0.746, loss 0.068
Epoch 11 done: accuracy 0.752, loss 0.066
⋮
Epoch 96 done: accuracy 0.850, loss 0.042
Epoch 97 done: accuracy 0.852, loss 0.042
Epoch 98 done: accuracy 0.851, loss 0.042
Epoch 99 done: accuracy 0.852, loss 0.042

Accuracy of the graph now:

- Much better, 85.2% of all predictions are correct!

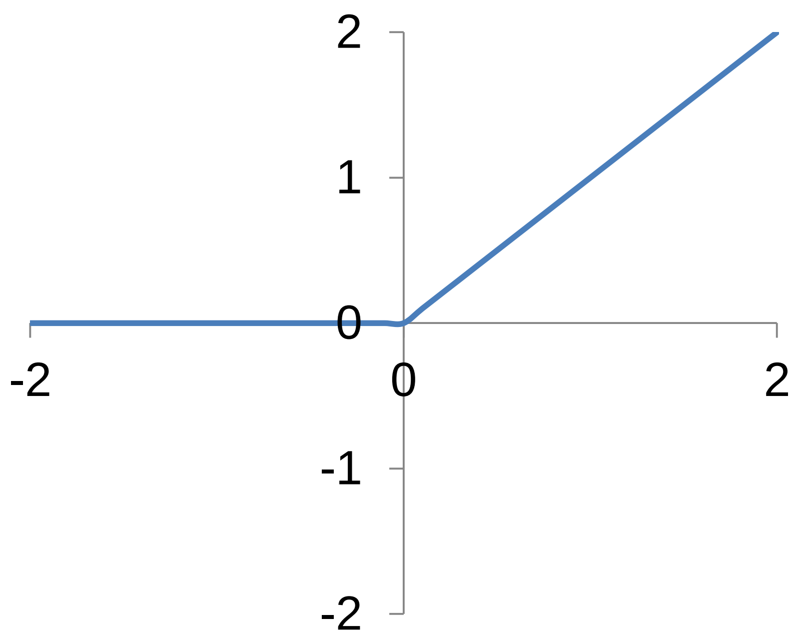
How to further improve the accuracy?

- Use a second fully connected layer
- However: $Y = W_2(W_1X) = (W_2W_1)X = \mathrm{W}_{2,1}\mathrm{X}$
- Two matrix multiplications in series (two layers) can be expressed as a single one (= one layer)
- Use a suitable activation function $f(\Sigma)$ between the layers!

Examples of activation functions:

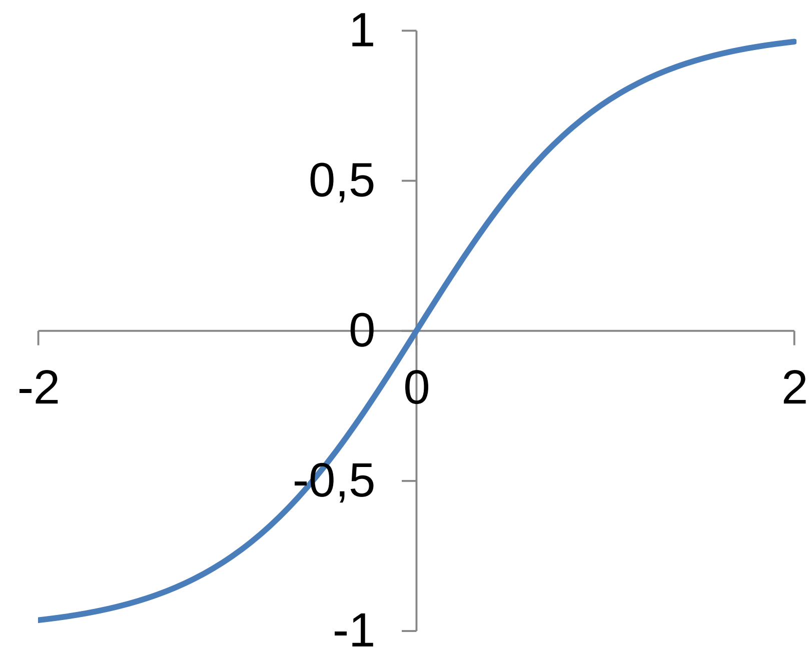
Rectified linear unit (ReLU):

$$f(x) = \begin{cases} 0, & when\ x < 0 \\ x, & else \end{cases}$$



Hyperbolic tangent (tanh):

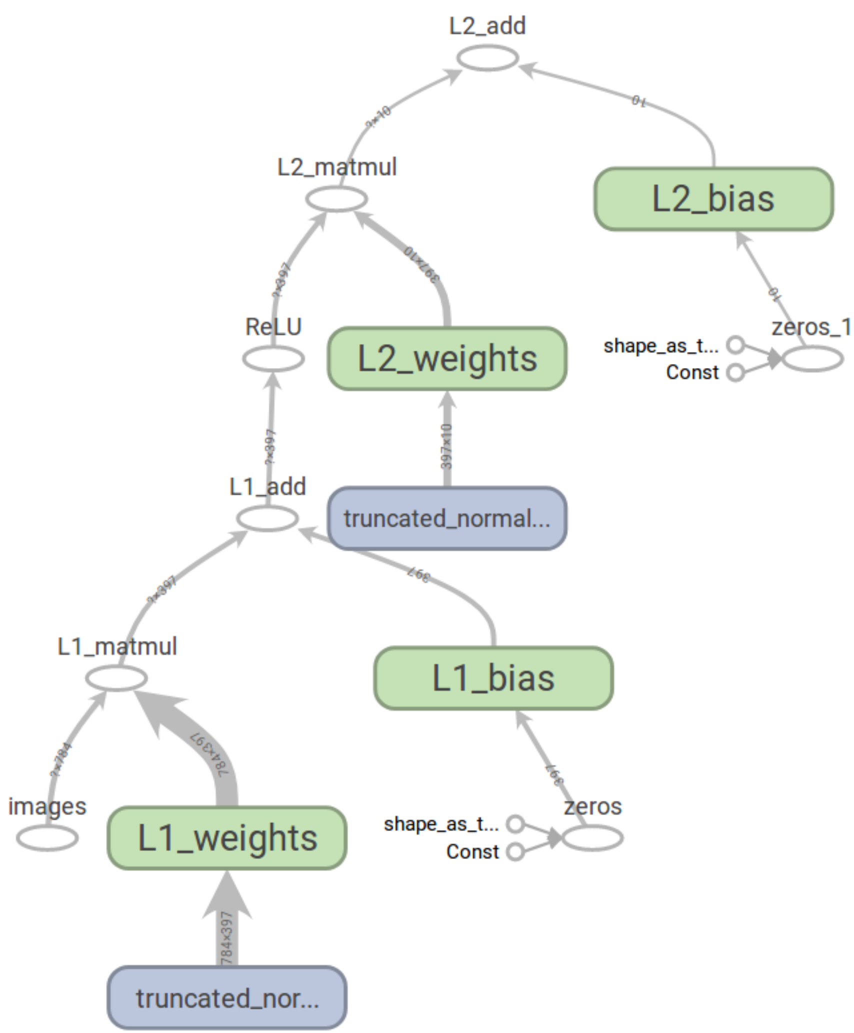
$$f(x) = \tanh(x)$$



Building the evaluation graph for an MNIST image with two layers:

```
# define size of hidden layer
hl_size = 397

# build evaluation graph with two fully connected layers
images = tf.placeholder(tf.float32, shape=(None,784),name='images')
l1_weights = tf.Variable(tf.random.truncated_normal(shape=(784,hl_size), mean=mu, stddev=sigma))
l2_weights = tf.Variable(tf.random.truncated_normal(shape=(hl_size,10), mean=mu, stddev=sigma))
l1_bias = tf.Variable(tf.zeros(hl_size), name='L1_bias')
l2_bias = tf.Variable(tf.zeros(10), name='L2_bias')

input_layer = images
hidden_layer = tf.add(tf.matmul(input_layer, l1_weights, name='L1_matmul'), l1_bias, name='L1_add')
hidden_layer = tf.nn.relu(hidden_layer, name='ReLU')
output_layer = tf.add(tf.matmul(hidden_layer, l2_weights, name='L2_matmul'), l2_bias, name='L2_add')
```

The described graph visualized with TensorBoard:



Progress during training (MSE loss):

Epoch 0 done: accuracy 0.502, loss 0.146
Epoch 1 done: accuracy 0.608, loss 0.110
Epoch 2 done: accuracy 0.671, loss 0.093
Epoch 3 done: accuracy 0.713, loss 0.083
Epoch 4 done: accuracy 0.743, loss 0.075
Epoch 5 done: accuracy 0.764, loss 0.069
Epoch 6 done: accuracy 0.781, loss 0.065
Epoch 7 done: accuracy 0.795, loss 0.061
Epoch 8 done: accuracy 0.810, loss 0.058
Epoch 9 done: accuracy 0.820, loss 0.055
Epoch 10 done: accuracy 0.829, loss 0.053
Epoch 11 done: accuracy 0.838, loss 0.051
⋮
Epoch 96 done: accuracy 0.930, loss 0.023
Epoch 97 done: accuracy 0.931, loss 0.022
Epoch 98 done: accuracy 0.931, loss 0.022
Epoch 99 done: accuracy 0.931, loss 0.022

Progress during training (softmax + CE loss):

Epoch 0 done: accuracy 0.865, loss 0.530
Epoch 1 done: accuracy 0.893, loss 0.403
Epoch 2 done: accuracy 0.903, loss 0.355
Epoch 3 done: accuracy 0.908, loss 0.326
Epoch 4 done: accuracy 0.915, loss 0.306
Epoch 5 done: accuracy 0.917, loss 0.289
Epoch 6 done: accuracy 0.922, loss 0.278
Epoch 7 done: accuracy 0.924, loss 0.268
Epoch 8 done: accuracy 0.927, loss 0.258
Epoch 9 done: accuracy 0.930, loss 0.250
Epoch 10 done: accuracy 0.932, loss 0.244
Epoch 11 done: accuracy 0.934, loss 0.236
⋮
Epoch 96 done: accuracy 0.972, loss 0.098
Epoch 97 done: accuracy 0.972, loss 0.096
Epoch 98 done: accuracy 0.972, loss 0.097
Epoch 99 done: accuracy 0.972, loss 0.096

Higher accuracy after first epoch
➢ Faster training

Slightly better accuracy after last epoch
➢ Better results

1.4 Convolutional Neural Networks (CNN)

A feature is often only in a subset of the image

How relevant are the pixels in these areas to detect the face?

A feature is often only in a subset of the image

How relevant are the pixels in these areas to detect the face?

How relevant are the pixels in this area to detect the face?

- Pixels which are close together have more common information than pixels far away

Using this assumption to improve the weight matrix:

Only weights which correspond to pixels in the proximity are non zero

$$\begin{pmatrix} w_0 \cdots w_i & 0 \cdots 0 & w_j \cdots w_k & \cdots & 0 \cdots 0 & 0 \cdots 0 & 0 \cdots 0 \\ w_l \cdots w_m & 0 \cdots 0 & w_n \cdots w_o & \cdots & 0 \cdots 0 & 0 \cdots 0 & 0 \cdots 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 \cdots 0 & 0 \cdots 0 & 0 \cdots 0 & \cdots & w_p \cdots w_q & 0 \cdots 0 & w_r \cdots w_s \end{pmatrix} \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_{M-1} \end{pmatrix}$$

Using the many 0 constants to improve memory footprint and runtime:

- Sparse matrix multiplication
- Convolutions

Factors to consider:

- How to choose the size of the neighborhood?
- How many regions are good (= size of M)?

Definition Convolution:

Integral over the product of a function with (another) reversed and shifted function:

$f, g : \mathbb{R} \rightarrow \mathbb{R}$ (continuous case):

$$(f * g)(n) = \int_{-\infty}^{\infty} f(k) \cdot g(n-k)\, dk$$

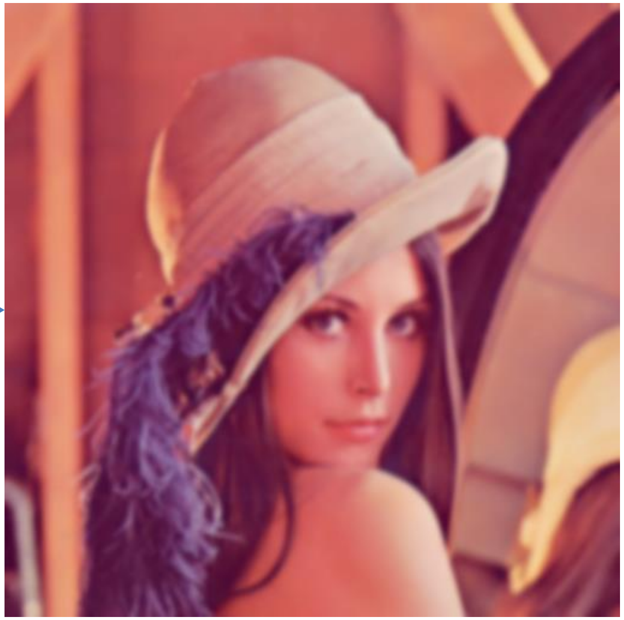
$f, g : \mathbb{Z} \rightarrow \mathbb{Z}$ (discrete case):

$$(f * g)[n] = \sum_{k \epsilon \mathbb{Z}} f[k] \cdot g[n-k]$$

Image convolved with Gaussian filter to blur edges and reduce noise:

$$\frac{1}{16} \cdot \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix}$$

10 X



- Low pass filter

Image convolved with Sobel filter to highlight edges:

$$\begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

1 X

- High pass filter

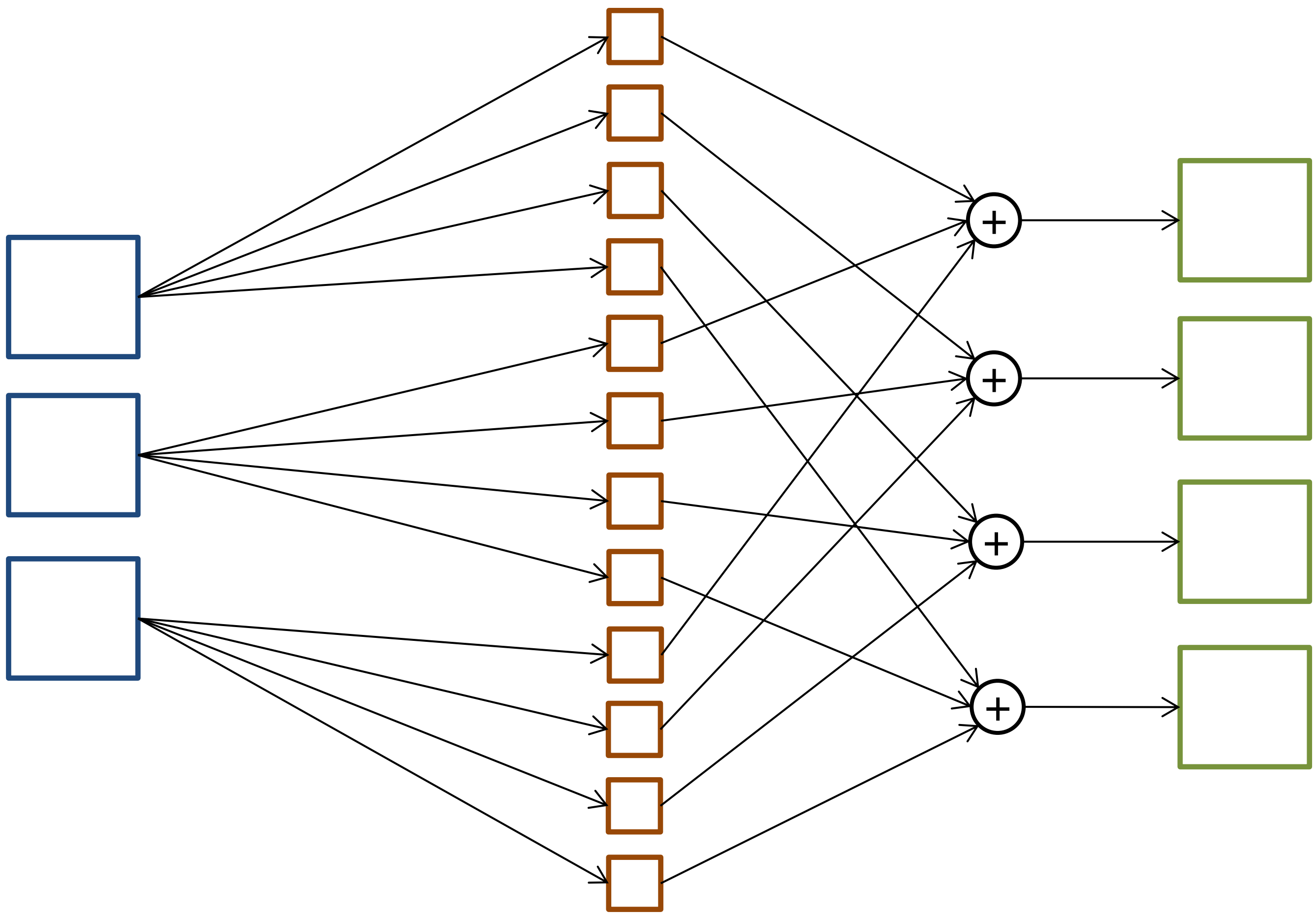
Weight matrix for convolutions (e.g. 3×3):

$$\begin{pmatrix} w_4 & w_5 & 0 & 0 & 0 \cdots 0 & w_7 & w_8 & 0 & 0 & \cdots & 0 & 0 & 0 \\ w_3 & w_4 & w_5 & 0 & 0 \cdots 0 & w_6 & w_7 & w_8 & 0 & \cdots & 0 & 0 & 0 \\ 0 & w_3 & w_4 & w_5 & 0 \cdots 0 & 0 & w_6 & w_7 & w_8 & \cdots & 0 & 0 & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 \cdots 0 & 0 & 0 & 0 & 0 & \cdots & w_3 & w_4 & w_5 \\ 0 & 0 & 0 & 0 & 0 \cdots 0 & 0 & 0 & 0 & 0 & \cdots & 0 & w_3 & w_4 \end{pmatrix} \begin{pmatrix} x_0 \\ \vdots \\ x_{N-1} \end{pmatrix} = \begin{pmatrix} y_0 \\ \vdots \\ y_{M-1} \end{pmatrix}$$

- Use the same $\mathrm{filter\_height} \times \mathrm{filter\_width}$ weights as a sliding window for the whole image
- Convolutions are still matrix multiplications
- Convolutions can be more efficiently implemented by stencil operations

Convolutions in Tensorflow:

N Input Channels: | N×M Masks: | M Output Channels:



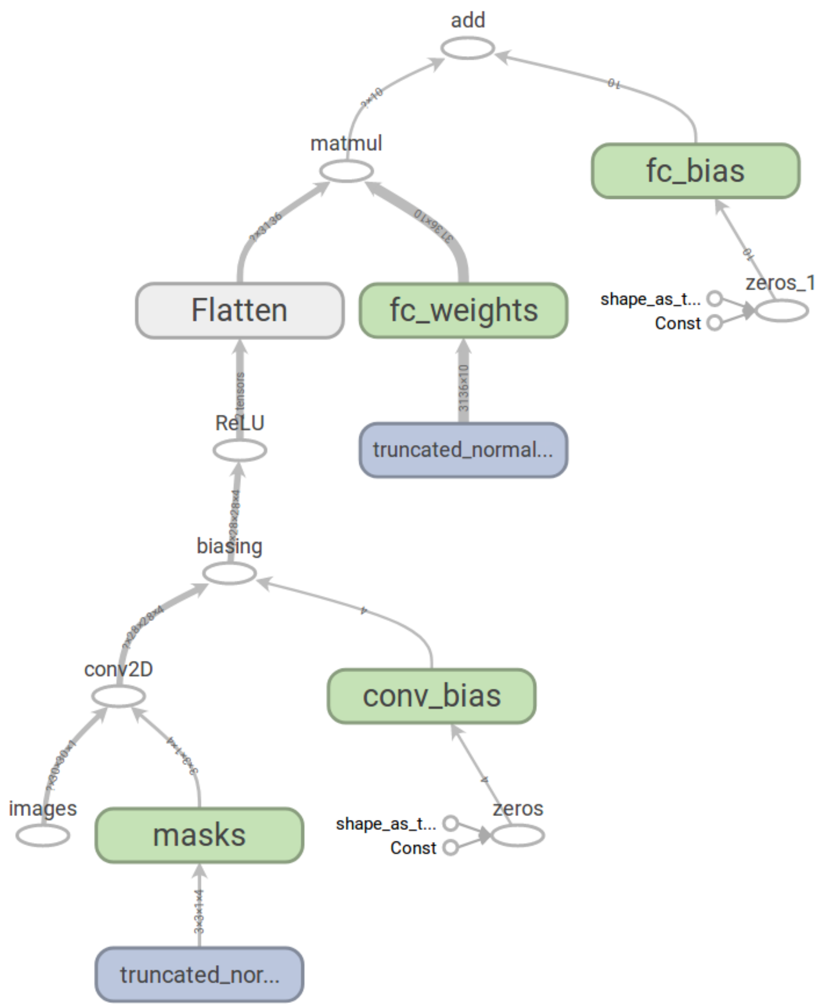
Preparing the MNIST dataset (border handling):

```
# reshape datasets to 28 x 28 x 1 pixels (height x width x color channels)
train_ds = train_ds.map(lambda image, label: (tf.reshape(image,[28,28,1]),label))
test_ds = test_ds.map(lambda image, label: (tf.reshape(image,[28,28,1]),label))

# pad images with 1 row/column of pixels on each side for 3 x 3 filter (border handling)
train_ds = train_ds.map(lambda image, label: (tf.pad(image,[[1,1],[1,1],[0,0]],'CONSTANT'),label))
test_ds = test_ds.map(lambda image, label: (tf.pad(image,[[1,1],[1,1],[0,0]],'CONSTANT'),label))

# cache the modified data in memory
train_ds = train_ds.cache()
test_ds = test_ds.cache()

# shuffling and dividing in batches as before
train_ds = train_ds.shuffle(60000).batch(batch_size)
test_ds = test_ds.batch(batch_size)
```

Building the graph with convolution and fully connected layer:

```
# build graph with one convolutional layer (with 4 masks) and one fully connected layer
images = tf.placeholder(tf.float32, shape=(None,30,30,1),name='images')
masks = tf.Variable(tf.random.truncated_normal(shape=(3,3,1,4), mean=mu, stddev=sigma))
fc_weights = tf.Variable(tf.random.truncated_normal(shape=(28*28*4,10), mean=mu, stddev=sigma))
conv_bias = tf.Variable(tf.zeros(4), name="conv_bias")
fc_bias = tf.Variable(tf.zeros(10), name='fc_bias')

input_layer = images
convolution = tf.nn.conv2d(input_layer, masks, strides=[1,1,1,1], padding='VALID', name='conv2D')
convolution = tf.add(convolution, conv_bias, name='biasing')
convolution = tf.nn.relu(convolution, name='ReLU')
hidden_layer = tf.contrib.layers.flatten(convolution)
output_layer = tf.add(tf.matmul(hidden_layer, fc_weights, name='matmul'), fc_bias, name='add')
```

The described graph visualized with TensorBoard:



Progress during training (softmax + CE loss):

Epoch 0 done: accuracy 0.835, loss 0.552
Epoch 1 done: accuracy 0.890, loss 0.371
Epoch 2 done: accuracy 0.903, loss 0.337
Epoch 3 done: accuracy 0.906, loss 0.317
Epoch 4 done: accuracy 0.908, loss 0.308
Epoch 5 done: accuracy 0.913, loss 0.302
Epoch 6 done: accuracy 0.915, loss 0.296
Epoch 7 done: accuracy 0.916, loss 0.290
Epoch 8 done: accuracy 0.918, loss 0.288
Epoch 9 done: accuracy 0.919, loss 0.281
Epoch 10 done: accuracy 0.921, loss 0.281
Epoch 11 done: accuracy 0.922, loss 0.274
⋮
Epoch 96 done: accuracy 0.971, loss 0.095
Epoch 97 done: accuracy 0.971, loss 0.092
Epoch 98 done: accuracy 0.971, loss 0.092
Epoch 99 done: accuracy 0.972, loss 0.090

Accuracy of the graph:

- 97.2% of all predictions are correct
- Same accuracy as the example with two fully connected layers

Why convolutions?

- Computations and memory for two fully connected layers:
  - Multiplications: 28*28*397 + 397*10 = 315,218
  - Additions: 28*28*397 + 397*10 = 315,218
  - Memory: (28*28*397 + 397*10)*4 = 1.26 MB

- Computations and memory for convolution and fully connected layer:
  - Multiplications: 28*28*3*3*4 + 28*28*4*10 = 59,584
  - Additions: 28*28*3*3*4 + 28*28*4*10 = 59,584
  - Memory: (3*3*4 + 28*28*4*10)*4 = 125.58 KB

- 81% less multiplications/additions and 90% less memory!

Result of a convolutional layer are M images

- Can be quite large and therefore increases computations
- Are all resulting pixels necessary?
- Pooling as tradeoff between accuracy and computational time

4	6	0	8
4	2	9	3
1	3	3	2
7	1	2	5

Max Pooling:

6	9
7	5

Avg Pooling:

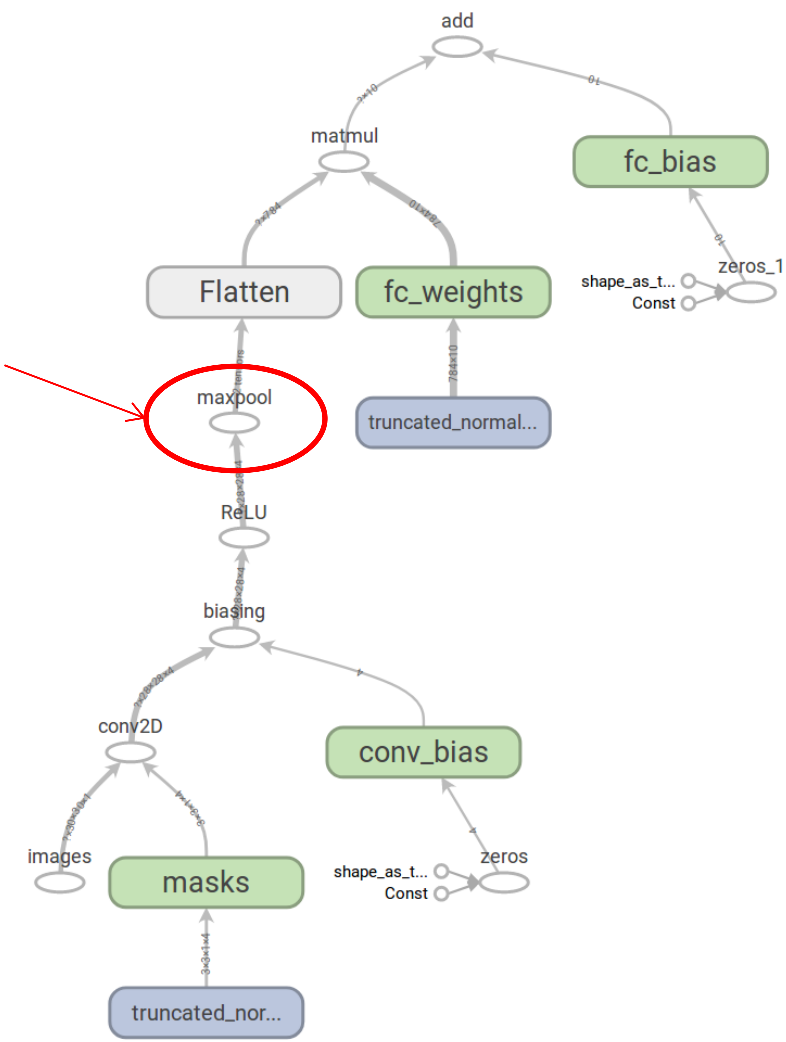
4	5
3	3

Adding maximum pooling to the graph:

```
# build graph with one convolutional layer (with 4 masks), pooling and one fully connected layer
images = tf.placeholder(tf.float32, shape=(None,30,30,1),name='images')
masks = tf.Variable(tf.random.truncated_normal(shape=(3,3,1,4), mean=mu, stddev=sigma))
fc_weights = tf.Variable(tf.radnom.truncated_normal(shape=(14*14*4,10), mean=mu, stddev=sigma))
conv_bias = tf.Variable(tf.zeros(4), name="conv_bias")
fc_bias = tf.Variable(tf.zeros(10), name='fc_bias')

input_layer = images
convolution = tf.nn.conv2d(input_layer, masks, strides=[1,1,1,1], padding='VALID', name='conv2D')
convolution = tf.add(convolution, conv_bias, name='biasing')
convolution = tf.nn.relu(convolution, name='ReLU')
pooling = tf.nn.max_pool2d(convolution, ksize=[1,2,2,1], strides=[1,2,2,1], padding='VALID')
hidden_layer = tf.contrib.layers.flatten(pooling)
output_layer = tf.add(tf.matmul(hidden_layer, fc_weights, name='matmul'), fc_bias, name='add')
```

The described graph visualized with TensorBoard:

new maximum pooling layer



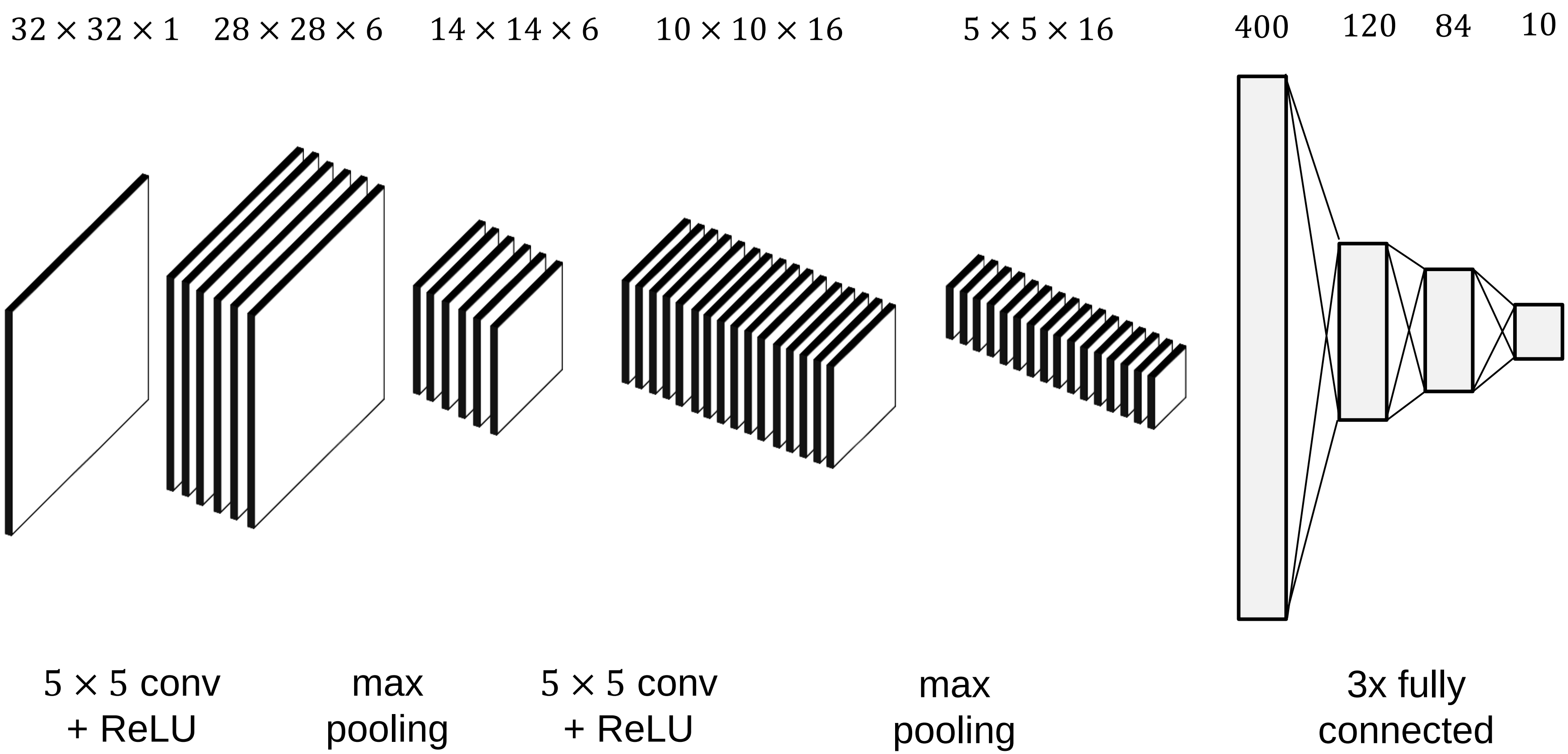
Progress during training (softmax + CE loss):

Epoch 0 done: accuracy 0.827, loss 0.579
Epoch 1 done: accuracy 0.883, loss 0.398
Epoch 2 done: accuracy 0.901, loss 0.352
Epoch 3 done: accuracy 0.908, loss 0.329
Epoch 4 done: accuracy 0.910, loss 0.317
Epoch 5 done: accuracy 0.912, loss 0.308
Epoch 6 done: accuracy 0.914, loss 0.301
Epoch 7 done: accuracy 0.914, loss 0.300
Epoch 8 done: accuracy 0.915, loss 0.294
Epoch 9 done: accuracy 0.918, loss 0.290
Epoch 10 done: accuracy 0.916, loss 0.291
Epoch 11 done: accuracy 0.918, loss 0.285
⋮
Epoch 96 done: accuracy 0.967, loss 0.106
Epoch 97 done: accuracy 0.967, loss 0.106
Epoch 98 done: accuracy 0.967, loss 0.107
Epoch 99 done: accuracy 0.968, loss 0.106

Compared to graph without max pooling:

- -0.4% points accuracy
- 39.5% less multiplications/additions
- 75% less memory for weights

Yann LeCun’s proposed CNN for the MNIST dataset:



- http://yann.lecun.com/exdb/lenet/

Progress during training (softmax + CE loss):

Epoch 0 done: accuracy 0.962, loss 0.124
Epoch 1 done: accuracy 0.974, loss 0.080
Epoch 2 done: accuracy 0.984, loss 0.050
Epoch 3 done: accuracy 0.984, loss 0.051
Epoch 4 done: accuracy 0.989, loss 0.033
Epoch 5 done: accuracy 0.987, loss 0.046
Epoch 6 done: accuracy 0.988, loss 0.039
Epoch 7 done: accuracy 0.989, loss 0.035
Epoch 8 done: accuracy 0.988, loss 0.038
Epoch 9 done: accuracy 0.990, loss 0.032
Epoch 10 done: accuracy 0.990, loss 0.036
Epoch 11 done: accuracy 0.989, loss 0.041
⋮
Epoch 96 done: accuracy 0.991, loss 0.073
Epoch 97 done: accuracy 0.991, loss 0.073
Epoch 98 done: accuracy 0.991, loss 0.074
Epoch 99 done: accuracy 0.991, loss 0.074