TensorFlow: neural networks lab

Gianluca Corrado and Andrea Passerini

gianluca.corrado@unitn.it
passerini@disi.unitn.it

Machine Learning

TensorFlow

- TensorFlow is a Python package
- Numerical computation using data flow graphs
- Developed (by Google) for the purpose of machine learning and deep neural networks research

Installation and Documentation

- `https://www.tensorflow.org/`

Outline

- MNIST dataset
  - https://www.tensorflow.org/versions/master/tutorials/mnist/beginners/index.html#the-mnist-data
- MNIST for ML Beginners
  - https://www.tensorflow.org/versions/master/tutorials/mnist/beginners/index.html
- Deep MNIST for Experts
  - https://www.tensorflow.org/versions/master/tutorials/mnist/pros/index.html

On Ubuntu 12.04

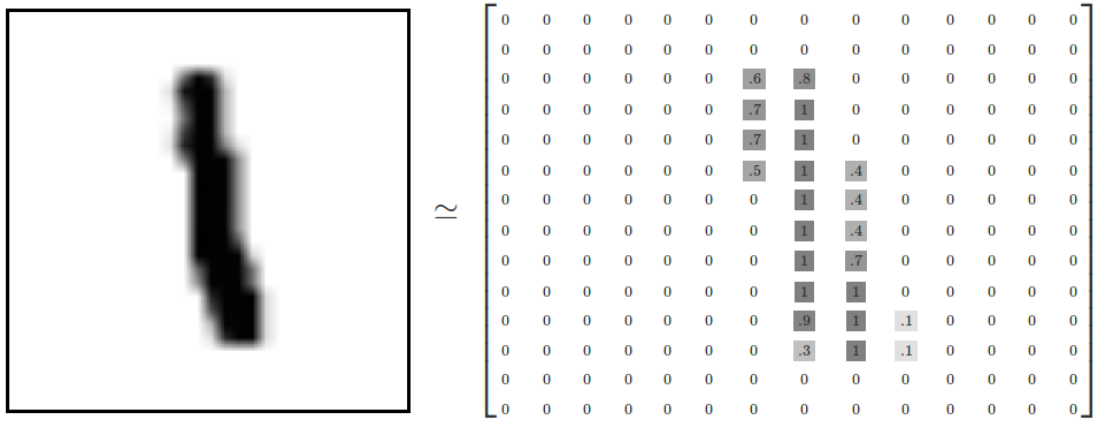
LD_LIBRARY_PATH

Run the script `run_me_on_ubuntu1204.sh`

MNIST dataset

- Dataset of handwritten digits

- Each image $28 \times 28 = 784$ pixels



- Train: 60k test: 10k

Importing MNIST

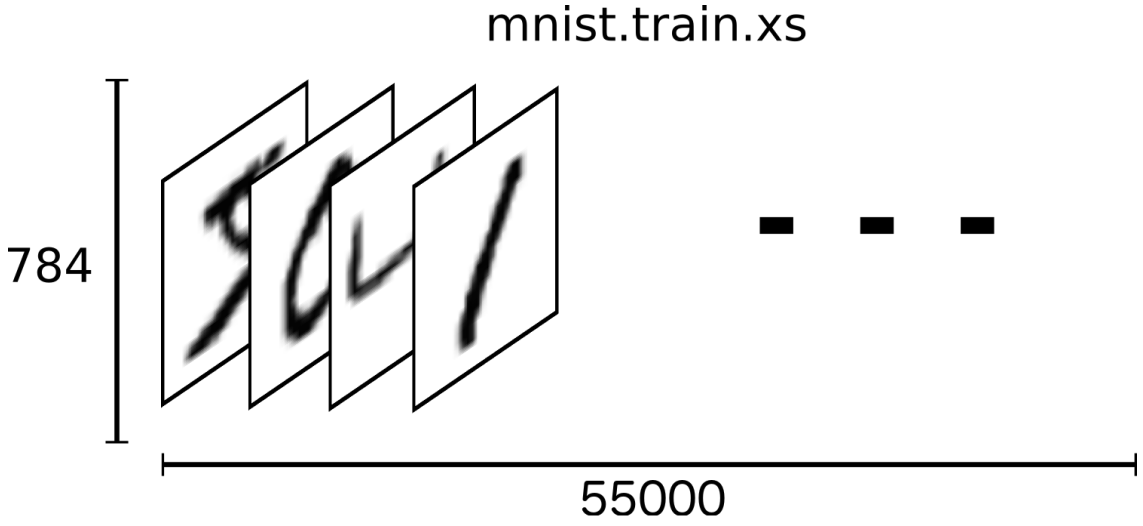
- Use the given `input_data.py` script

```
import input_data
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

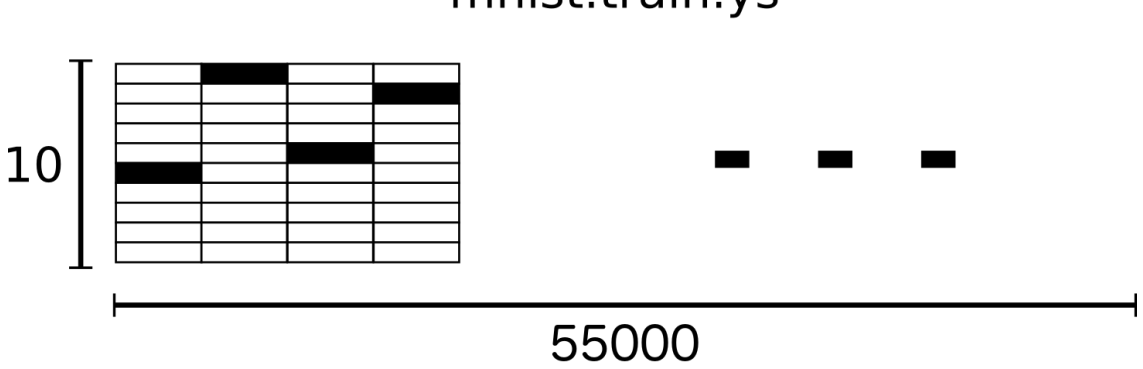
- Train: 55k, validation: 5k, test: 10k
- `mnist` is a Python `Class` (has attributes, and methods)
  - `mnist.train.images`
  - `mnist.train.labels`
  - `mnist.train.next_batch(n)`
  - `...`

Data representation

- The $28 \times 28 = 784$ pixels are represented as a vector



- The 10 classes are represented with the one-hot encoding



Softmax regressions

- Look at an image and give probabilities for it being each digit
- Evidence that an image is a particular class i

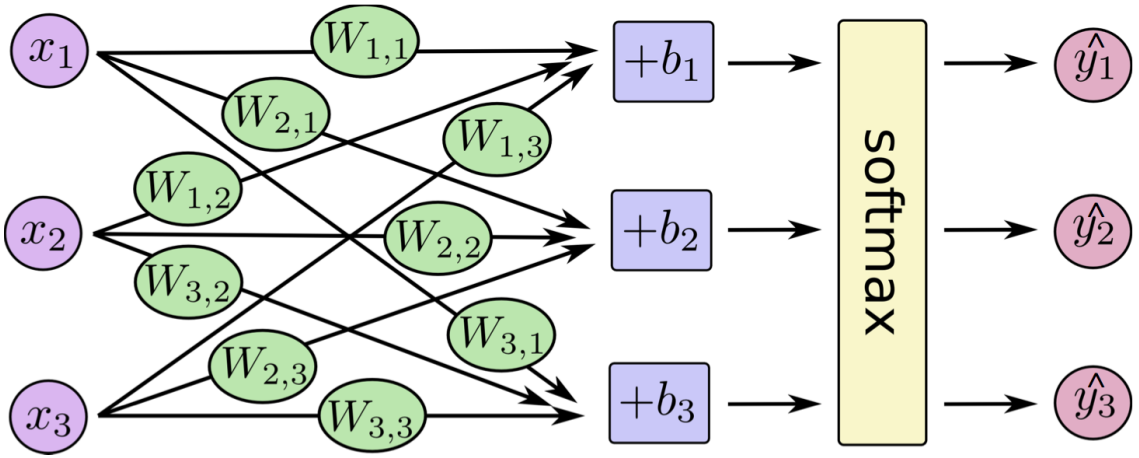
$$evidence_i = \sum_j W_{ij} x_j + b_i \qquad \forall i$$

- Softmax to shape the evidence as a probability distribution over i cases

$$softmax_i = \frac{exp(evidence_i)}{\sum_j exp(evidence_j)} \qquad \forall i$$

Softmax regressions

- Schematic view



- Vectorized version

$$\begin{bmatrix} \hat{y_1} \\ \hat{y_2} \\ \hat{y_3} \end{bmatrix} = \text{softmax}\left(\begin{bmatrix} W_{1,1} & W_{1,2} & W_{1,3} \\ W_{2,1} & W_{2,2} & W_{2,3} \\ W_{3,1} & W_{3,2} & W_{3,3} \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \\ b_3 \end{bmatrix} \right)$$

- Compactly

$$\hat{y} = softmax(Wx + b)$$

Implementation: model

- Import TensorFlow

```
import tensorflow as tf
```

- Define the placeholders

```
# Values that we will input during the computation
x = tf.placeholder("float", shape=[None, 784])
y = tf.placeholder("float", shape=[None, 10])
```

- Define the variables

```
# Variables of the model (part of the TensorFlow computational graph)
W = tf.Variable(tf.zeros([784, 10]))
b = tf.Variable(tf.zeros([10]))
# Variables initialization
init = tf.initialize_all_variables()
```

- Define the softmax layer

```
# Softmax regression (output layer)
y_hat = tf.nn.softmax(tf.matmul(x, W) + b)
```

Implementation: optimization

- Define the cost (cross-entropy): $H_y(\hat{y}) = -\sum y \log \hat{y}$

```
# Cross-entropy
cross_entropy = -tf.reduce_sum(y*tf.log(y_hat))
```

- Define the training algorithm

```
# SGD for minimizing the cross-entropy (learning rate = 0.01)
train_step = tf.train.GradientDescentOptimizer(0.01).minimize(cross_entropy)
```

- Start a new session (for now we have not computed anything)

```
# If ipython, start a new InteractiveSession, it prevents garbage collection.
# all the objects will be inspectable in ipython.
sess = tf.InteractiveSession()
# Otherwise, start a new Session with sess = tf.Session().
```

- Initialize the variables

```
# Initialize the variables
sess.run(tf.initialize_all_variables())
```

- Train the model

```
# Train the model
for n in range(1000):
    batch_xs, batch_ys = mnist.train.next_batch(100)
    sess.run(train_step, feed_dict={x: batch_xs, y: batch_ys})
```

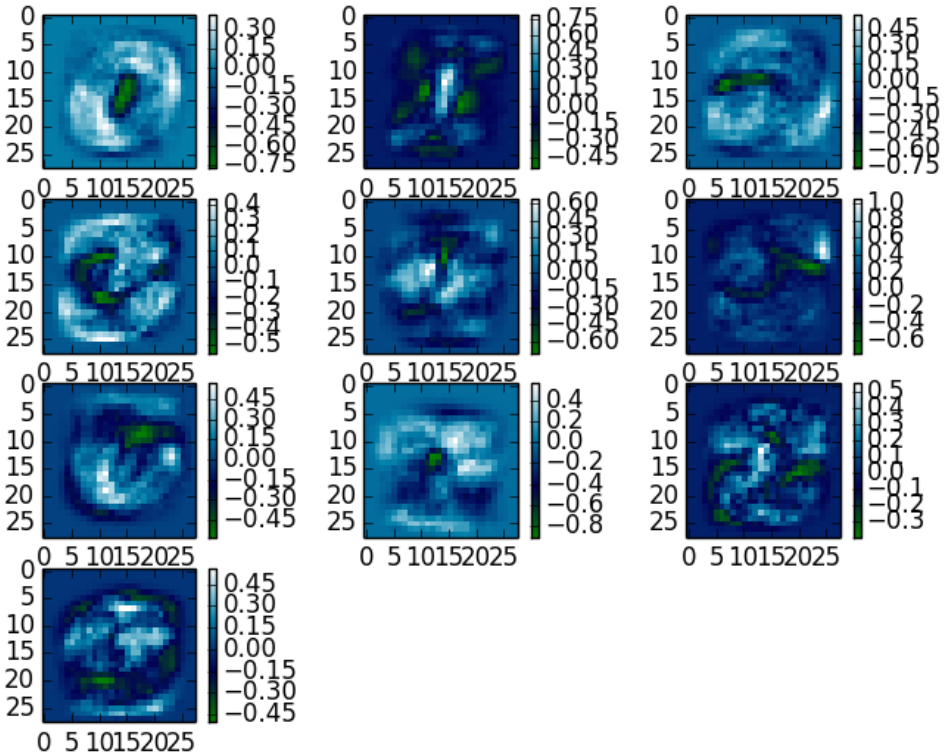
Implementation: evaluation

- Evaluate accuracy (it should be around 0.91)

```
# Evaluate the prediction
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_hat,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
print sess.run(accuracy, feed_dict={x: mnist.test.images, y: mnist.test.labels})
```

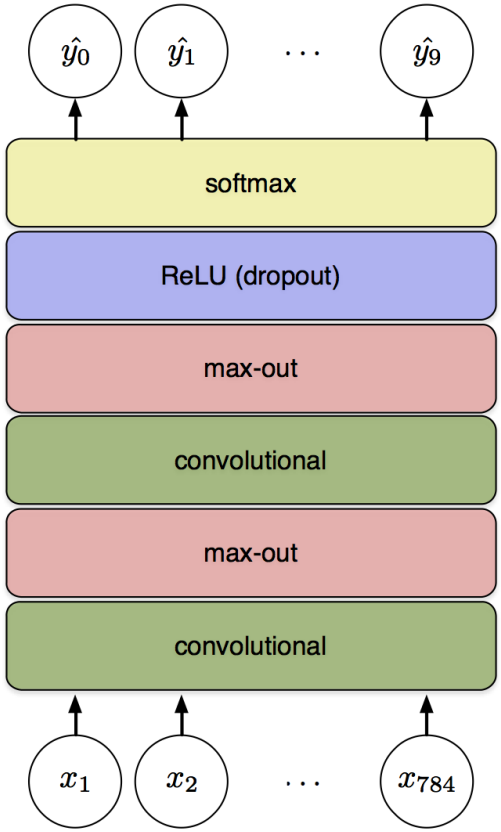
- Plot the model weights (`plotter.py`)

```
import plotter
plotter.plot_mnist_weights(W.eval())
```



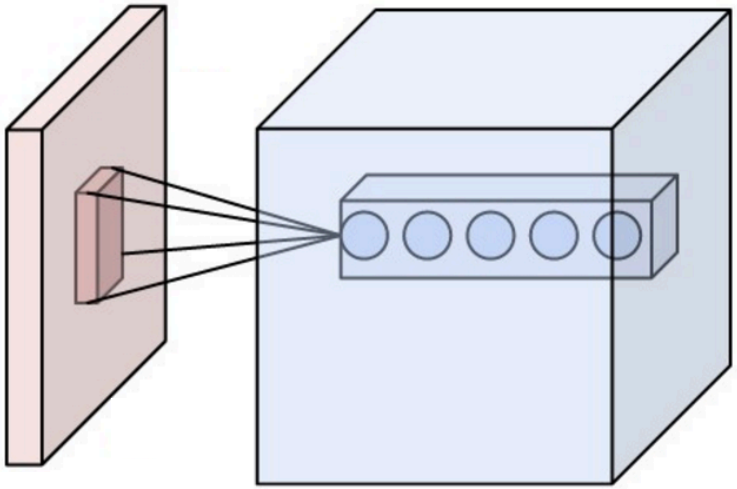
Deep architechtures

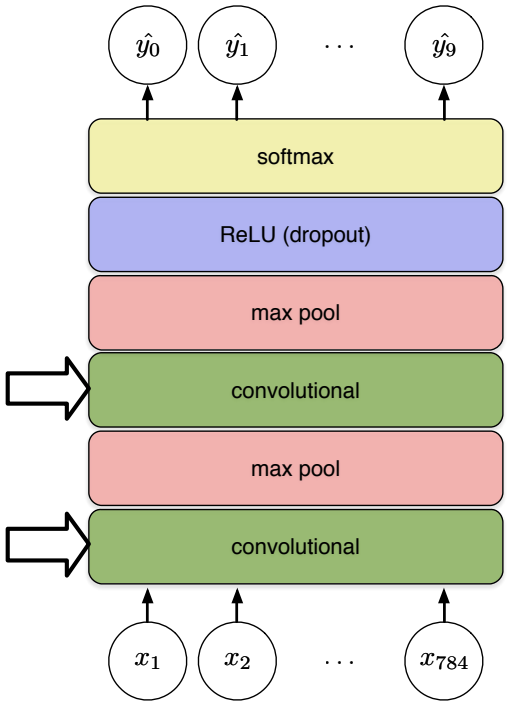
- 0.91 accuracy on MNIST is NOT good!
- State of the art performance is 0.9979
- Let's refine our model
  - 2 convolutional layers
  - alternated with 2 max pool layers
  - ReLU layer (with dropout)
  - Softmax regressions
- Accuracy target: 0.99



Convolutional layer

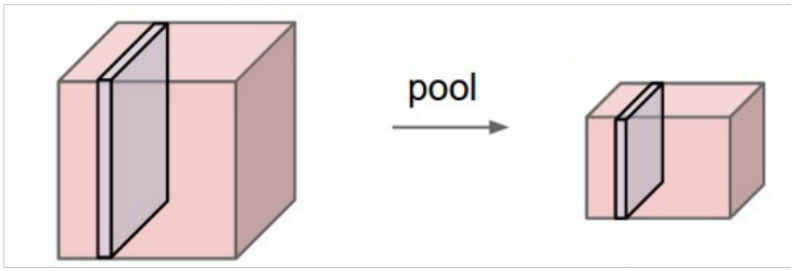
- Broadly used in image classification
- Local connettivity (width, height, depth)
- Spatial arrangement (stride, padding)
- Parameter sharing





Max pooling layer

- Commonly used after convolutional layer(s)
- Reduce spatial size
- Avoid overfitting
- Max pooling 2×2 is very common

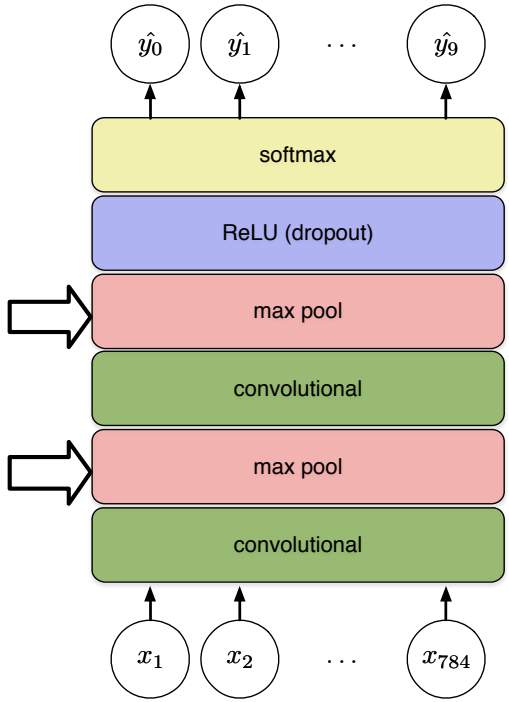


Single depth slice

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

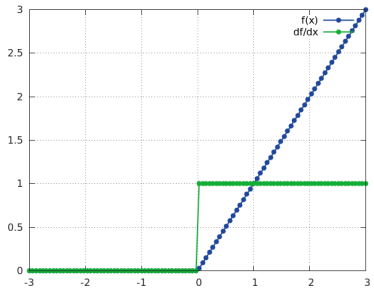
max pool with 2x2 filters and stride 2

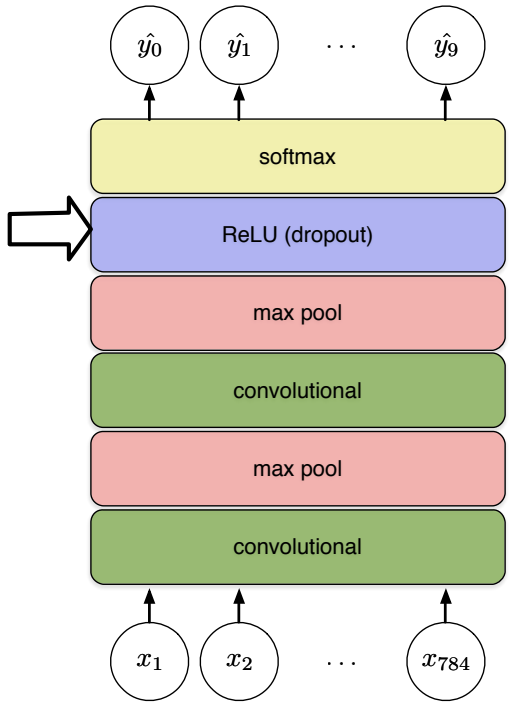
6	8
3	4



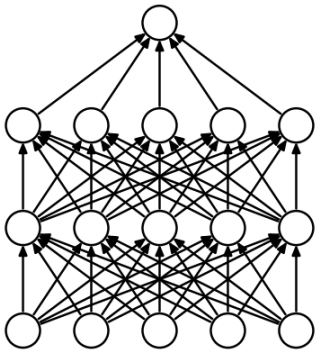
ReLU (and Dropout)

- Fully connected layer
- Rectified Linear Unit activation

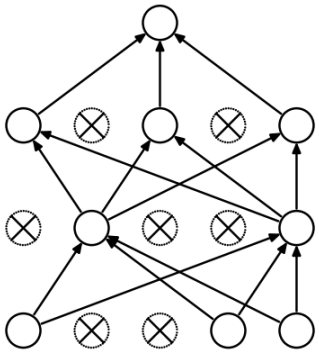




- Dropout randomly excludes neurons to avoid overfitting



(a) Standard Neural Net



(b) After applying dropout.

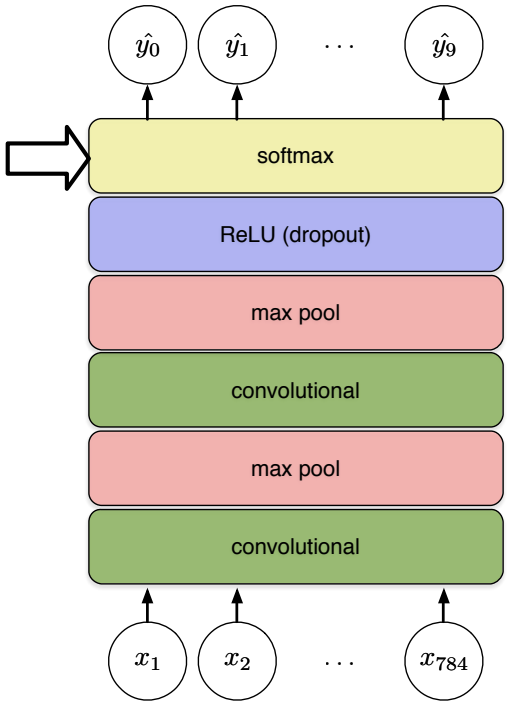
Softmax layer

- Look at a feature configuration (coming from the layers below) and give probabilities for it being each digit
- Evidence that a feature configuration is a particular class i

$$evidence_i = \sum_j W_{ij} x_j + b_i \qquad \forall i$$

- Softmax to shape the evidence as a probability distribution over i cases

$$softmax_i = \frac{exp(evidence_i)}{\sum_j exp(evidence_j)} \qquad \forall i$$



Implementation: preparation

- Imports

```
import input_data
import tensorflow as tf
```

- Load data

```
# MNIST dataset
mnist = input_data.read_data_sets('MNIST_data', one_hot=True)
```

- Placeholders and data reshaping

```
# Values that we will input during the computation
x = tf.placeholder("float", shape=[None, 784])
y = tf.placeholder("float", shape=[None, 10])

# reshape vectors of size 784, to squares of size 28x28
x_image = tf.reshape(x, [-1,28,28,1])
```

NOTE

Reshaping is needed for convolution and max pooling

Implementation: weights initialization

- Define functions to initialize variables of the model

```
"""
Initializing variables to zero, when the activation of
a layer is made of ReLUs will yield a null gradient. This generates
dead neurons -> no learning!
More precisely a ReLU is not differentiable in 0, but it
is differentiable in any epsilon bubble defined around 0.
"""
# init for a weight variable
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

# init for a bias variable
def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

Implementation: convolution and pooling

- Define functions (to keep the code cleaner)

```
# define functions for the convolution and the max pooling.
"""
input: A Tensor. Must be one of the following types: float32, float64.
filter: A Tensor. Must have the same type as input.
strides: A list of ints. 1-D of length 4. The stride of the sliding window for each dimension of input.
padding: A string from: "SAME", "VALID". The type of padding algorithm to use. Usually, this yields
an output that is smaller than the input. The pad argument allows you to simplicitly pad
the input with zeros.
"""
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1],padding='SAME')
"""
value: A 4-D Tensor with shape [batch, height, width, channels] and type float32, float64, qint8, quint8, qint3
ksize: A list of ints that has length >= 4. The size of the window for each dimension of the input tensor.
strides: A list of ints that has length >= 4. The stride of the sliding window for each dimension of the input
tensor.
padding: A string, either 'VALID' or 'SAME'. The padding algorithm. In this case 'SAME' shrinks the dimension
according to ksize and strides.
"""
def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],strides=[1, 2, 2, 1],padding='SAME')
```

Implementation: model (convolution and max pooling)

- 1st layer: convolutional with max pooling

```
"""1st layer: convolutional layer with max pooling"""
#[width, height, depth, output_size]
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])

h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)
```

- 2nd layer: convolutional with max pooling

```
"""2nd layer: convolutional layer with max pooling"""
#[width, height, depth, output_size]
W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])

h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)
```

Shrinking

- Applying 2×2 max pooling we are shrinking the image
- After 2 layers we moved from 28×28 to 7×7
- For each point we have 64 features

Implementation: model (ReLu, dropout and softmax)

- 3rd layer: ReLU

```
"""3rd layer: fully connected layer"""
# [input_size, output_size]
W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])
# flattening the output of the previous layer
h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)
```

Reshape

We are switching back to fully connected layers, we want to reshape the input as a flat vector.

- 3rd layer: add dropout

```
"""Add dropout"""
# using a placeholder for keep_prob will allow to turn off the dropout during testing
keep_prob = tf.placeholder("float")
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)
```

- 4th layer: softmax (output)

```
"""4th layer: fully connected layer"""
# [input_size, output_size]
W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_hat = tf.nn.softmax(tf.matmul(h_fc1_drop, W_fc2) + b_fc2)
```

Implementation: optimization and evaluation

- Define the cost (cross-entropy): $H_y(\hat{y}) = -\sum y \log \hat{y}$

```
# Cross-entropy
cross_entropy = -tf.reduce_sum(y*tf.log(y_hat))
```

- Define the training algorithm

```
# Minimization of the cross-entropy (learning rate = 1e-4) with adaptive gradients.
train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
```

- Start a new session (for now we have not computed anything)

```
# If ipython, start a new InteractiveSession, it prevents garbage collection.
# all the objects will be inspectable in ipython.
sess = tf.InteractiveSession()
# Otherwise, start a new Session with sess = tf.Session().
```

- Initialize the variables

```
# Initialize the variables
sess.run(tf.initialize_all_variables())
```

- Define the accuracy before training (for monitoring)

```
# Define accuracy
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_hat,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, "float"))
```

Implementation: optimization and evaluation

- Train the model (may take a while)

```
for n in range(20000):
    batch = mnist.train.next_batch(50)
    if n % 100 == 0:
        train_accuracy = accuracy.eval(feed_dict={x:batch[0], y: batch[1], keep_prob: 1.0})
        print "step %d, training accuracy %g" % (n,train_accuracy)
    sess.run(train_step, feed_dict={x: batch[0], y: batch[1], keep_prob: 0.5})
```

- Evaluate the accuracy (it should be around 0.99)

```
# Evaluate the prediction
print "test accuracy %g" % accuracy.eval(feed_dict={x: mnist.test.images, y: mnist.test.labels, keep_prob: 1.0})
```

Assignment

The third ML assignment is to compare the performance of the deep convolutional network when removing layers.
For this assignment you need to adapt the code of the complete deep architecture. By removing one layer at the time, and keeping all the others, you can evaluate the change in performance of the neural network in classifing the MNIST dataset.

Note

- The only coding required is to modify the shape and/or size of the input vectors of the layers. The output of each layer has to remain the same.
- The report has to contain a short introduction on the methodologies used in the deep architecture showed during the lab (convolution, max pooling, ReLU, dropout, softmax).

Assignment

Steps

1. Remove the 1st layer: convolutional and max pooling
2. Train and test the network
3. Remove the 2nd layer: convolutional and max pooling
4. Train and test the network
5. Remove the 3rd layer: ReLU with dropout
6. Train and test the network

Computation

Training a model on a quad-core CPU takes 30-45 mins. You may want to use the computers in the lab.

Assignment

- After completing the assignment submit it via email
- Send an email to gianluca.corrado@unitn.it (cc: passerini@disi.unitn.it)
- Subject: tensorflowSubmit2016
- Attachment: id_name_surname.zip containing:
  - the Python code
    - ★ model_no1.py (model without the 1st layer)
    - ★ model_no2.py (model without the 2nd layer)
    - ★ model_no3.py (model without the 3rd layer)
  - the report (PDF format)

NOTE

- No group work
- This assignment is mandatory in order to enroll to the oral exam

References

- https://www.tensorflow.org/
- http://cs231n.github.io/convolutional-networks/
- https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf