ID2223 – Lab 2
Deep Learning with TensorFlow

Introduction

Goals

This lab has the following goals:

- Learn how to setup and run a computational graph in Tensorflow
- Implement a single-layer as well as a multi-layer Neural Network in Tensorflow
- Combine different activation functions to increase the accuracy
- Tackle overfitting using regularization
- Further improve the performance by using Convolutional Layers

Requirements

For this lab assignment in *Tensorflow*, we will use the Tensorflow Python API. The version of Python you have installed on your machine must be at least 2.7. There are different options for installing Tensorflow but we recommend to use the Docker based installation as it is less invasive (less things to install locally) and easier to make sure it works on every platform. For the docker based installation you will need to install Docker Engine first. The Docker image for Tensorflow contains Jupyter IPython which is an interactive environment Python. (Note: if you do not have a laptop, we will make an instance of Hopsworks available to you that supports the python interpreter with Tensorflow).

MNIST Dataset

For this lab we will use a dataset called *MNIST*. The MNIST dataset (Mixed National Institute of Standards and Technology database) is a database of handwritten digits that is commonly used for evaluating image classification algorithms. You can read more about the dataset in Yann LeCun's MNIST page or Chris Olah's visualizations of MNIST.

Tensorflow

The main building blocks of a Tensorflow program are: *Tensors*, a *Computational Graph*, *Variables*, *Constants*, *PlaceHolders*, *Fetches* and *Feeds*. These building blocks are well presented in the Tensorflow's Documentation. Please read up on them and make sure that you understand how a typical Tensorflow program is written. Specifically, for Section 1 of this lab look at Tensorflow Tutorial and Convolutional Netowrk is useful for Section 6.

Useful Python Pointers

In this section, we give you some practical python pointers that are helpful for this lab. First, Python Tutorial is a good baseline to learn the language of Python (e.g., have a quick look into section 4 and 5). NumPy is the fundamental package for scientific computing with Python. NumPy Arrays are the backbone of Tensors in Tensorflow (read about ndarray and broadcasting).

Run Tensorflow with Docker

Running the default starting command:

```
docker run -it -p 8888:8888 gcr.io/tensorflow/tensorflow
```

will not save your notebooks and you might lose work. You should use named containers in order to save your work. For this, the first time you start the docker image, name it *tensorflow-id2223* by running:

```
docker run -p 8888:8888 --name tensorflow-id2223 -it gcr.io/tensorflow/↙
    ↳ tensorflow
```

And the next times you want to continue work on your docker image run:

```
docker start -ai tensorflow-id2223
```

Code Template

The code that is used in all of the subsequent sections, contains a common sequence of steps. These common steps are presented in Listing 1. According to the requirements of each task, you should only complete the template code (numbered steps in the template) in order to complete the implementation. The learning and visualization steps are also given but not in this document, you can find them in our github repo.

Listing 1: "Code Template"

```
# all tensorflow api is accessible through this
import tensorflow as tf
# to visualize the resutls
import matplotlib.pyplot as plt
# 70k mnist dataset that comes with the tensorflow container
from tensorflow.examples.tutorials.mnist import input_data

tf.set_random_seed(0)
```

```
# load data, 60K trainset and 10K testset
mnist = input_data.read_data_sets("MNIST_data/", one_hot=True)

# 1. Define Variables and Placeholders

# 2. Define the model

# 3. Define the loss function

# 4. Define the accuracy

# 5. Define an optimizer

# initialize
init = tf.initialize_all_variables()
sess = tf.Session()
sess.run(init)

# 6. Train and test the model, store the accuracy and loss per iteration

# 7. Plot and visualise the accuracy and loss
```

1 One-Layer Softmax Regression

In this task, you are going to design a neural network with 1 layer of 10 softmax neurons. If you are new to python and Tensorflow, it might be useful to look at the beginner part of the Tensorflow Tutorial.

You should build your model based on the softmax regression formula:

$$Y = softmax(X * W + b)$$

We will use cross entropy as the loss (error) for evaluating the model.

$$H(Y', Y) = -\sum Y_i' \cdot \log Y_i$$

Due to the fact that you train in batches, you will compute the mean of the cross entropy over the trained batches.

Finally you should compute the correctness of your prediction for each image as a 0/1 result and then compute the accuracy as the mean over the predictions.

When you compute accuracy and loss you run the whole training/testing dataset through your current model and this is quite expensive. Doing so every iteration would make it impractical, so you will compute it every 100 rounds, which we can call epochs. Plot accuracy and loss for each epoch to see how your model evolves during the training iterations

The images from the MNIST dataset are defined as 3 dimensional tensors : *(image_height, image_width, input_channel)* with the parameters:

- image shape 28 X 28
- input channels 1 - gray-scale images.

The softmax function that you use for each of the neurons takes as input a vector of values. Since our images are defined as 3 dimensional tensors you will need to reshape it into a vector with *image_height * image_width * input_channel* elements.

Listing 2: "One-layer softmax regression"

```
# 1. Define Variables and Placeholders
#the first dimension (None) will index the images in the mini-batch
X = tf.placeholder(tf.float32, [None, ?, ?, ?])
Y_ = ?                                  # correct answers(labels)
W = tf.Variable(tf.zeros([784, 10]))    # weights W[784, 10] 784=28*28
b = ?                                   # biases b[10]
XX = tf.reshape(X, [-1, 784])           # flatten the images into a single line of pixels

# 2. Define the model - compute predicitions
Y = tf.nn.softmax(?)

# 3. Define the loss function
cross_entropy = ?

# 4. Define the accuracy
accuracy = ?

# 5. Train with an Optimizer
train_step = tf.train.GradientDescentOptimizer(<learning_rate>).minimize(<loss_function>)
```

Task 1

As per listing 2 perform the programing tasks:

- Define the variables and placeholders
- Define a model
- Define the loss
- Define the accuracy
- Train with the GradientDescentOptimizer and a learning rate of 0.5
- Train with the AdamOptimizer (a slightly better optimizer) and a learning rate of 0.005

What loss and accuracy do you get when training over 10.000 iterations?

2 Feedforward Neural Network with Back-Propagation

Now we are going to improve the accuracy by adding 4 more layers into our Neural Network. In a multi-layer, feedforward Neural Network, each layer instead of doing a weighted sum of all pixels, it does a weighted sum of the output of its previous layer. Design your layers such that you will have 200, 100, 60, 30 and 10 neurons for each layer respectively.

You can choose a different activation function for each hidden layer. For this task we want you to investigate two different combinations of activation function: I) *sigmoid* and *softmax*, and II) $ReLU$ and *softmax*. We always apply softmax in the last layer (the output layer) and the other activation functions in the hidden layers.

Now complete the code in Listing 3, use the 2 different settings for the activation function and learn your model. In the end, compare the convergence rate by looking at the accuracy and loss plot.

Listing 3: "5-layers with a separate activations"

```
# 1. Define Variables and Placeholders

X = tf.placeholder(tf.float32, [None, ?, ?, ?]) #the first dimension (None) will index the images
Y_ = ?                                  # correct answers

# Weights initialised with small random values between -0.2 and +0.2
W1 = tf.Variable(tf.truncated_normal([784, ?], stddev=0.1)) # 784 = 28 * 28
B1 = tf.Variable(tf.zeros([?]))
W2 = ?
B2 = ?
W3 = ?
B3 = ?
W4 = ?
B4 = ?
W5 = ?
B5 = ?

# 2. Define the model

XX = ?
Y1 = tf.nn.?(tf.matmul(XX, W1) + B1)
Y2 = ?
Y3 = ?
Y4 = ?
Ylogits = ?
Y = tf.nn.?(Ylogits)

# 3. Define the loss function

cross_entropy = tf.nn.?(Ylogits, Y_) # calculate cross-entropy with logits
cross_entropy = tf.reduce_mean(?)*?
```

Task 2

Do the following tasks in your program:

1. initialize all of the weights and biases
2. redefine your model by connecting the output of each layer to the input of the next layer
3. calculate the Logits (scores)
4. use a specific cross-entropy function to calculate loss using logits

Finally, analyse your results and try to answer the following questions:

Questions

1. What is the maximum accuracy that you can get in each setting for running your model with 10000 iterations?
2. Is there a big difference between the convergence rate of the sigmoid and the ReLU ? If yes, what is the reason for the difference?
3. What is the reason that we use the softmax in our output layer?
4. By zooming into the second half of the epochs in accuracy and loss plot, do you see any strange behaviour? What is the reason and how you can overcome them? (e.g., look at fluctuations or sudden loss increase after a period of decreasing loss).

3 Regularization and Tuning

In the result of the model in the previous section, you probably can see the sign of overfitting or a high learning rate. In this section, we want you to try the techniques such as *dropout* and *learning rate decay* to see whether you can improve the accuracy of your model. You are, of course, free to add other regularization techniques (such as L2 regularization).

Task 3

The programming tasks involve adding to your neural network:

1. learning rate decay
2. regularization, such as dropout
3. both

An example is given in listing 4 on how to tell Tensorflow to apply dropout to layer1. It defines Y1d and uses this as input for Y2, instead of the previous Y1. When you run it, the model will expect to receive a value for the pkeep placeholder. You will need to provide a value for it in the `sess.run`

When implementing learning rate decay, you need to tell the optimizer to use a placeholder for the learning rate as it will change during the iterations and it will be provided at runtime. You will thus need to provide a new learning rate value for the placeholder every time you call the `sess.run`. Alternatively you can define a variable for the learning rate using the `tf.train.exponential_decay`

Listing 4: "Dropout and learning rate decay"

```
# 1. Define Variables and Placeholders

#learning rate placeholder
lr = ?
# placeholder for probability of keeping a node during dropout = 1.0 at test time (no dropout) and 0.75 at ↙
      ↳ training time
pkeep = ?

# 2. Define the model
Y1 = ?
Y1d = tf.nn.dropout(Y1, pkeep)

Y2 = tf.nn.relu(tf.matmul(?, W2) + B2)
Y2d = ?

Y3 = ?
Y3d = ?

Y4 = ?
Y4d = ?

Ylogits = ?
Y = ?

# 3. In the training step - provide the appropriate pkeep
# 4. In the training step, if you used a placeholder - adjust learning rate - according to exponential decay rate
def training_step(...):
   ...
   sess.run(?)
   ...
```

Task 3

The analysis tasks involve:

- Explain during grading the motivation behind learning rate decay.
- Explain during grading why dropout can be an effective regularization technique.
- For each of the programming tasks plot accuracy and loss, and analyze whether your additions influence the accuracy/loss and if yes, in what way.

4 Convolutional Neural Network

In this section you will change your neural network to use convolutional layers. In the previous tasks, we have reshaped our input from a matrix to a vector from the very beginning, however, this means that we lose precious information that could be including in the learning task. By getting rid of the matrix in the beginning we lose locality information. We lose the fact that the digits have lines and curves in them. The convolutional layers that we will add to the network will let us extract and compose higher level features from this locality information.

For this task you should read this part of the tensorflow tutorial in order to see how to setup a convolutional layer.

It is important to understand the shape of the input/output of your convolutional layers and what changes the shape of the input to that of the output. The weight variables from your convolutional layers should follow 4-dimensional tensors of *(patch_height, patch_width, input_channels, output_channels)*. This tensor defines the weights structure of one unit in the layer. The patch is the area of the image that the unit will try to look at. The input to the whole layer is a 4 dimensional tensor of *(batch_size, image_height, image_width, input_channels)*. The stride defines how to move the patch over the layer's input data in order to compute a unit's input data. Thus, the stride's structure is also 4 dimensional: *(batch_step, height_step, width_step, channel_step)*. Keep in mind that if the stride has any element bigger than 1 in its structure it will reduce that particular dimension in the output of the layer.

Next we provide a full layer parameter description.

Conv layer 1

The input data is (batch_size, image_height, image_width, input_channels) = (100,28,28,1)

- stride of (1,1,1,1)
- patch of 5X5
- input depth/channels of 1
- output depth/channels of 4

Conv layer 2

- stride of (1,2,2,1)
- patch of 5X5
- input depth/channels of 4 - previous output
- output depth/channels of 8

Conv layer 3

- stride of (1,2,2,1)
- patch of 4X4
- input depth/channels of 8 - previous output
- output depth/channels of 12

Densely connected layer

The output from the third layer is a four dimensional tensor of *(batch, height, width, depth)*. The densely connected layer is a relu layer like the ones defined in previous tasks and takes as input two dimension tensor of *(batch, values)*. You will need to reshape the tensor from having *height X width X depth* in matrix form to having a vector of *height * width * depth* elements.

- input structure - determine it from the previous layer output
- output structure: vector of 200 elements

Readout layer

The readout layer is the same softmax layer from the previous tasks.

- input structure: a vector of 200 elements
- output structure: vector of 10 elements

Task 4

Your programming part of this task includes:

1. Setup the network layer with 3 conv layers, 1 relu layer and 1 softmax layer with a GradientDescentOptimizer.
2. Change the optimizer to the AdamOptimizer.
3. Add a learning decay to the network.

4. Add regularization through dropout.

As a note - for programming subtask 1, all the changes you need to make are within variable declaration/initialization and model setup and as an example you have listing 5.

Your analysis part of this task includes:

- Define the output structure of the convolutional layers based on the given stride.
- For each of the programming subtasks 2-4 point out the changes that happen to the accuracy and error and explain why your modifications caused those changes.

Listing 5: "Convolutional network abstractions"

```
#define weight variable for a convolutional layer
W_I = tf.Variable(tf.truncated_normal([<patch_height>, <patch_width>, <input_channels>, <output_channels>], ↙
      ↳ stddev=0.1))
#define convolutional layer in model
Y_I = tf.nn.relu(tf.nn.conv2d(<input>, <weights>, strides=[<batch_step>, <height_step>, <width_step>, ↙
      ↳ <channel_step>], padding='SAME') + <bias>)
```

5 Convolutional Network for CIFAR-10 (Bonus task: 25% extra)

In this task you will run the convolutional network that you have developed in the previous task on a different dataset called CIFAR-10, available on www.hops.site.

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. You can read more about the dataset here and on the original kaggle site. You can import the CIFAR-10 dataset into one of your projects on hops.site by selecting it from the available public datasets. From your project, you can then download the dataset to your docker container.

This task is straight forward, apply the previously defined neural network on this new dataset. You will need to do some small changes in order to account for the differences in this dataset, like image size:32X32, and different depth/channels due to color. In CIFAR-10, as in MNIST, there are 10 object classes (with 6000 images per class). Also, while in your previous task you had the utility to split your training set into batches, you do not have it now.