Uppsala University
Department of Information Technology
Division of Systems and Control
NW 2018-02
Last rev. February 21, 2018 by NW

Statistical Machine Learning

Instruction to the laboratory work

Image classification with Deep Learning

Preparation:
Read Section 2 carefully
Do all preparatory exercises in Section 3

Reading instructions:
Lecture notes (Lindsten, et. al): Chapter 4
This lab-pm: Chapter 1-3.

Name		Assistant's comments
Program	Year of reg.	
Date		
Passed prep. ex.	Sign	
Passed lab.	Sign	

Contents

1 Introduction

Deep Learning is a sub-field within machine leaning that models high level abstractions of *high-dimensional* and *complex* data. One example of such complex and high-dimensional data is images. Consider a classification model that takes an image as input and the class that this image belongs to (for example "cat", "dog" or "bike") as output. Assume further that the image has $100 \times 100 = 10\,000$ pixels where each pixel in the image is considered as one input variable. This makes the problem very high-dimensional (even for such a small image). Further, the relation between these pixel values and the actual class is far from obvious, which makes the data very complex.

Deep neural networks, or just *deep larning*, have in the last 5 to 10 years proven to be very successful to model such complex and high-dimensional data sets. In this lab we will look at a smaller image classification task. We will learn a neural network to read hand-written digits with up to 98% prediction accuracy. In the very end of the lab we will load a pre-trained network that has been trained on a much larger dataset with 1 000 different classes.

This laboratory work is based on Lecture 8 and 9 together with Chapter 4 in the lecture notes. Therefore, it is advisable to have the material from those lectures fresh in mind before starting this laboratory work.

The goal of this laboratory work is to:

- Learn how to build and train a neural network
- Learn how to improve the neural network model and its training.
- Application 1: Learn how to classify hand-written digits using neural network.
- Application 2: See how a state-of-the-art deep neural network performs at classifying real world images
- Get a glimpse of a state-of-the-art software library (Tensorflow) for deep learning.

Throughout the lab we will us a software library called *Tensorflow*, which can be accessed via R. This library is introduced in Section 2. Section 3 contains the preparatory exercises, and Section 4 contains the exercises that you do during the 4h lab session.

Important: Read Chapter 2 and try to answer the preparatory exercises *before* the lab.

2 Tensorflow for R

Tensorflow[1] is an open source software library for machine learning. It was originally developed by the Google Brain team and is today used by many companies as well as academic research groups. Tensorflow can be used for any type of computation but it is tailored especially for deep learning and neural networks.

Tensorflow is natively written in Python and C++ and well documented API:s exist for these languages. There is also an interface[2] for R by which the full Python API can be accessed from R. This is the interface that we will use throughout this laboratory work.

2.1 Installation

Tensorflow for R is already installed in the computer rooms (Linux-system) where the laboratory session is scheduled. You can either choose to use these computers during the lab or to bring your own computer.

If you choose to use your own computer, you need to have Tensorflow for R properly installed *before* the lab. The lab supervisors will not be able to assist you with the installation process during the lab. To get Tensorflow up and running in R, the installation procedure is similar to the installation of other R-packages that we have used throughout the course. Basically, the following lines have to be executed in the R console

```
install.packages("tensorflow")
library(tensorflow)
install_tensorflow()
```

See also `https://tensorflow.rstudio.com/tensorflow/` for more details on the installation procedure.

2.2 Building a graph

A Tensorflow program is normally divided into two parts. The first part assembles the operations into a so-called *computational graph*, and the second part executes these operations.

Assume we want to add two constant numbers $z = x + y$ where $x = 1$ and $y = 2$. First, we define the constants using `tf$constant`

[1]`www.tensorflow.org`
[2]`https://tensorflow.rstudio.com/`

```
# Load the tensorflow package
library(tensorflow)

# Create a constant representing x=2
x <- tf$constant(value = 2)

# Create a constant representing y=1
y <- tf$constant(value = 1)

# Add the two numbers
z <- x + y
```

The operations have now been defined but no operation has yet been performed. To actually do the summation, we need to launch the graph in a *session*.

2.3 Launching the graph

To execute the operations above we need to start a session, in which the operations are performed. In this session we can choose to evaluate any node in the graph. Here, we choose to evaluate `z`, which is the sum of `x` and `y`.

```
# Create session
sess <- tf$Session()

# Compute
result <- sess$run(z)
print(result)
```

This gives the following output.

```
[1] 3
```

We can also choose to evaluate other nodes in the graph, or even multiple of them. For example, the following command

```
result <- sess$run(c(x,y,z))
print(result)
```

gives a list with the numbers 2, 1 and 3

```
[[1]]
[1] 2

[[2]]
[1] 1
```

```
[[3]]
[1] 3
```

After we are done, the session should be closed to release resources.

```
# Close the Session when we are done.
sess$close()
```

2.4 Placeholder and feeds

The graph above is only able to add the numbers 2 and 1. Usually, we would like to construct the graph as a function, which can take any input. This we can do with *placeholders*.

The following code first constructs the function $f(x, y) = x + y$.

```
# Create a constant op representing x=2
x <- tf$placeholder(dtype = tf$float32)

# Create a constant op representing y=1
y <- tf$placeholder(dtype = tf$float32)

# Add the two numbers
z <- x + y
```

We need to specify the type of the placeholder, which in this case is `tf$float32`.

The following code will then evaluate the function with $x = 2$ and $y = 1$ giving $f(2, 1) = 2 + 1 = 3$.

```
# Create session
sess <- tf$Session()

# Feed the desired data points into the graph
result <- sess$run(z, feed_dict = dict(x = 2, y = 1))
print(result)
sess$close()
```

Note that to evaluate z the placeholders must be fed with values (otherwise an error will be generated). This is done by passing these values to the `feed_dict` argument while running an operation that is dependent on these placeholders. Later, we will use `feed_dict` to feed test and training data to our the algorithm.

2.5 Variables

Similar to constants, variables hold a value but can update during runtime of the program. Later, when we train a model, variables will be used to hold and update parameters in the model.

Assume we want to write a program that assigns $x := x + 1$ in a for loop. Since x changes value, it will be introduced as a variable. All variables require an initial value, in our case we use $x = 0$ as initial value.

Here, the operation that assigns $x + 1$ to x is called `update`. This operation is executed multiple times in a for-loop and the output is printed each time.

```
# Create a Variable and, that will be initialized to the
   scalar value 0. it will be of type float32
x <- tf$Variable(initial_value = 0.0)

# Create an constant equal to 1
y <- tf$constant(value = 1.0)
# ... and add theis value to the variable x
new_value <- x + y

# Assign the sum to x. Call this operation 'update'
update <- tf$assign(x, new_value)

# Variables must be initialized while executing the
   graph by running the corresponding operation.
# We first have to add this initialization operation to
   the graph.
init_op <- tf$global_variables_initializer()

# Create session
sess <- tf$Session()

# Run the initialization operation to initialize the
   variable x (with 0)
sess$run(init_op)

# Print the initial value of 'x'
print(sess$run(x))

# Run the operation that updates 'x' and print 'x'.
for (i in 1:3) {
sess$run(update)
print(sess$run(x))
}

sess$close()
```

The code produces the following output

```
[1] 1
[1] 2
[1] 3
```

2.6 Linear regression example

Now we are ready to do something (slightly) more interesting. We will do linear regression using Tensorflow. We consider the model

$$Y = \beta_0 + X\beta_1 + \epsilon, \tag{1}$$

where ϵ represents the noise term.

First, we generate training data $\{x_i, y_i\}_{i=1}^n$ with $n = 100$ data point from the model (1), with the true parameters $\beta = [\beta_0,\ \beta_1]^\mathsf{T} = [0.3,\ 0.1]^\mathsf{T}$ and $\epsilon \sim \mathcal{N}(0, 0.01^2)$. We sample the inputs x_i uniformly from the interval $[0, 1]$. We plot the data in the similar manner as we have done previously in the course.

```
# Linear regression
n = 100
xt <- matrix(runif(n, min=0, max=1))
beta0 = c(0.3,0.1)
yt <- matrix(beta0[1] + beta0[2]*xt + 0.01*rnorm(n,0,1))

# Plot the data
plot(xt,yt) # Plot data
abline(beta0) # Plot true model
legend(x="topleft", legend = c("Data","True model"), col =
    c("black", "black"), pch = c(1,NA), lty = c(NA,1),
    lwd=c(1,1))
```

We use (1) as our regression model where we want estimate the parameters $\beta = [\beta_0,\ \beta_1]^\mathsf{T}$ via linear regression based on the training data $\{x_i, y_i\}_{i=1}^n$. With this regression model, the training data statistics can be written as

$$\mathbf{y} = \mathbf{X}\beta + \boldsymbol{\epsilon}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} 1 & x_1 \\ \vdots & \vdots \\ 1 & x_n \end{bmatrix}, \quad \beta = \begin{bmatrix} \beta_0 \\ \beta_1 \end{bmatrix}$$

where $\boldsymbol{\epsilon}$ is a vector with the noise realizations.

Now, load the library and define one placeholder `X` for the input data $\mathbf{X}$ and one placeholder `Y` for the output data $\mathbf{y}$. Since `X` and `Y` are matrices, we need to tell Tensorflow their sizes. We know that `X` has two columns and `Y` has one column.

The number of rows corresponds to the number of data points. This dimension can be determined at runtime when data is fed into the graph. Now, we specify this dimension as `NULL`, meaning it will be determined when the graph is executed.

```
# Load the tensorflow library
library(tensorflow)

# Create a placeholder, Y, of size NULL x 2 and of type tf$float32
X <- tf$placeholder(dtype = tf$float32, shape = shape(NULL, 2))

# Create a placeholder, Y, of size None x 1 and of type tf$float32
Y <- tf$placeholder(dtype = tf$float32, shape = shape(NULL, 1))
```

We also need to create a variable containing β, which will be updated iteratively during the learning process. We initialize the variable with a constant matrix $\beta = [1 \;\; 1]^\mathsf{T}$

```
# Create the variable betahat
beta <- tf$Variable(initial_value = tf$constant(value =
    1.0, shape = shape(2,1)))
```

We proceed by constructing the mean-squared-error cost function

$$\arg\min_{\beta} J(\beta), \;\text{ where } J(\beta) = \frac{1}{n}\sum_{i=1}^{n}(\beta_0 + \beta_1 x_i \beta - y_i)^2 = \frac{1}{n}\|\mathbf{X}\beta - \mathbf{y}\|^2. \quad (2)$$

First, we multiply the input placeholder X with the variable β using `tf$matmul` for matrix multiplication. We then subtract the data to get the error, take the square, and finally take the mean, all according to (2).

```
# Now we can multiply our input placeholder X by beta
Z <- tf$matmul(X, beta)

# Construct a loss function by comparing the predicted mean
    with the placholder for the measurement
mean_squared_error <- tf$reduce_mean((Z - Y)^2)
```

We perform the optimization with gradient descent. This means that during optimization we update the parameter β by going in the opposite direction of the gradient of the cost function with a certain step length γ. This means we compute $\beta := \beta - \gamma\nabla J(\beta)$. In Tensorflow, this operation is implemented as

```
# Finally define the optimization routine to
gamma <- 0.1 # The learning rate
train_step <-
    tf$train$GradientDescentOptimizer(gamma)$minimize(mean_squared_error)
```

Now we are done with the computational graph. So far no computation has been done, we have only told the algorithm what to do. To launch the graph and execute these commands we create a session and run the object `train_step` multiple times in a for-loop. Afterwards, the predicted output is computed.

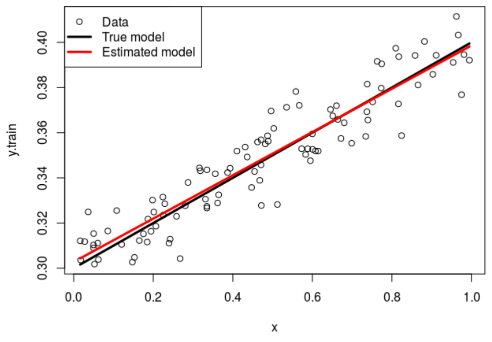
```
  # Create session and initialize variables
  sess <- tf$Session()
  sess$run(tf$global_variables_initializer())

  # Build the X matrix to be used later in the regression
  Xt <- cbind(matrix(1,nrow=n,ncol=1),xt)

  # Update the parameter beta in a loop
  for (i in 1:1000) {
  # Run 'train_setp
  sess$run(train_step, feed_dict = dict(X = Xt, Y = yt))
  }
  # Predict the output before close the session
  betahat <- sess$run(beta,feed_dict = dict(X = Xt))
  sess$close()
```

We can now plot the result using the usual plot commands in R.

```
# Plot the estimated model
abline(betahat,col = "red", lwd=3)
# ...and add a legend
legend(x="topleft", legend = c("Data","True model",
    "Estimated model"), col = c("black", "black", "red"),
    pch = c(1,NA, NA), lty = c(NA,1, 1), lwd = c(1,3,3))
```



Note that we have performed linear regression by iteratively minimizing the square error cost function, instead of using the closed form solution given by the normal equations. This numerical approach can be used also with other models and cost functions to find the parameters, also in problems where no closed form solution is available. In this lab, we will replace the linear model with a neural network and the mean squared error cost function with the cross entropy cost function.

3 Preparation exercises

3.1 Softmax and cross-entropy

In Lecture 3 the logistic regression model was introduced for problems with two classes. The class-1 probability $q(X;\beta) = \Pr(Y = 1|X)$ was modeled as

$$q(X;\beta) = f(Z), \quad \text{where} \quad Z = \beta_0 + \sum_{j=1}^{p} X_j\beta_j, \quad \text{and} \quad f(Z) = \frac{e^Z}{1+e^Z}. \tag{3}$$

Question 3.1: *Assume that we have estimated the parameters in (3) $\widehat{\beta}_0 \dots \widehat{\beta}_p$ using logistic regression from a set of training data. After training, we compute $z_\star = \widehat{\beta}_0 + \sum_{j=1}^{p} x_{*j}\widehat{\beta}_j = 1.0$ for a certain test input $x_\star = [x_{*1}, \dots x_{*p}]^\mathsf{T}$. What are the probabilities according to the logistic regression model that the corresponding output Y belongs to either of the two classes 0 or 1, i.e., what is $\Pr(Y = 0|X = x_\star)$ and $\Pr(Y = 1|X = x_\star)$?*

Answer:

In this lab we will consider classification problems with $K > 2$ classes. For each input X we define K different *class probabilities* $q_1(X;\beta), \dots, q_K(X;\beta)$, which are the probabilities that X belongs to either of the K classes, i.e. $q_k(X;\beta) = \Pr(Y = k|X)$.

We also extend the logistic function in (3) to a function that has these K class probabilities as outputs. We use the *softmax function* mapping $Z = [Z_1, \dots, Z_K]^\mathsf{T}$ onto K probabilities, i.e. $\mathbb{R}^K \mapsto [0,1]^K$. The softmax function is defined as

$$\text{softmax}(Z) = \frac{1}{\sum_{l=1}^{K} e^{Z_l}} \begin{bmatrix} e^{Z_1} & \dots & e^{Z_K} \end{bmatrix}^\mathsf{T}. \tag{4}$$

Now we are ready to extend logistic regression model (3) to multiple classes $K > 2$. This extension is defined as

$$q_k(X;\beta) = \Pr(Y = k|X) = [\text{softmax}(Z)]_k, \quad \text{where} \quad Z_k = \beta_{0k} + \sum_{j=1}^{p} X_j\beta_{jk} \tag{5}$$

and where $[\text{softmax}(Z)]_k = \frac{e^{Z_k}}{\sum_{l=1}^{K} e^{Z_l}}$ is the kth output from the softmax function. In contrast to logistic regression (3), we now have a set of parameters $\beta_{0k}, \ldots, \beta_{pk}$ for each class k.

Question 3.2: *Consider a problem with three classes* $\{1, 2, 3\}$ *where we have estimated all parameters* $\widehat{\beta}_{kj}$ *from a training data set using the softmax model in* (5). *For a certain test input* $X = x_\star$ *we compute* $Z = z_\star = [z_{*1},\ z_{*2},\ z_{*3}]^\mathsf{T}$ *where* $z_{*1} = 0$, $z_{*2} = -1$, *and* $z_{*3} = 1$. *What are the class probabilities for the three different classes, i.e. what are* $q_1(x_\star; \widehat{\beta}) = Pr(Y = 1|X = x_\star)$, $q_2(x_\star; \widehat{\beta}) = Pr(Y = 2|X = x_\star)$, *and* $q_3(x_\star; \widehat{\beta}) = Pr(Y = 3|X = x_\star)$*? Which class has the highest probability?*

Answer:

Suppose we want to learn the parameters β based on a training data set $\{x_i, y_i\}_{i=1}^{n}$. Instead of letting the output y_i of a data point i be an integer in $\{1, \ldots, K\}$, we represent the kth class with a vector $y_i = [y_{i1}\ \ldots\ y_{iK}]^\mathsf{T}$, where $y_{ij} = 1$ if $j = k$, and $y_{ik} = 0$ otherwise. This is also known as the *one-hot encoding*. For example if we have the three classes $\{1, 2, 3\}$ we encode class $k = 1$ as $y_i = [1\ 0\ 0]^\mathsf{T}$, $k = 2$ as $y_i = [0\ 1\ 0]^\mathsf{T}$, and $k = 3$ as $y_i = [0\ 0\ 1]^\mathsf{T}$.

As a measure of fit between true output $y_i = [y_{1i}, \ldots,\ y_{iK}]^\mathsf{T}$ and the class probabilities $q_1(x_i; \beta), \ldots, q_K(x_i; \beta)$ we use the *cross-entropy loss function*

$$L(x_i, y_i, \beta) = -\sum_{k=1}^{K} y_{ik} \log(q_k(x_i; \beta)) \tag{6}$$

Question 3.3: *Consider the class probabilities* $q_1(x_\star; \widehat{\beta}), \ldots, q_3(x_\star; \widehat{\beta})$ *you got from the previous exercise. Compute the cross-entropy loss* $L(x_\star, y_\star, \widehat{\beta})$ *between these probabilities and* $y_\star^{(1)} = [1\ 0\ 0]^\mathsf{T}$, $y_\star^{(2)} = [0\ 1\ 0]^\mathsf{T}$, *and* $y_\star(3) = [0\ 0\ 1]^\mathsf{T}$, *respectively. Which one has the lowest cross-entropy?*

Answer:

3.2 Dense neural network

Consider a classification problem where the input consists of $p = 144$ input variables $X = [X_1, \dots X_p]^\mathsf{T}$ and the output belongs to four classes $Y \in \{1, \dots, 4\}$. We want to model the class probabilities $q_k(X;\theta) = \Pr(Y = k|X)$ with a neural network with two dense layers.

$$H = \sigma\left(W^{(1)\mathsf{T}}X + b^{(1)\mathsf{T}}\right), \tag{7a}$$

$$Z = W^{(2)\mathsf{T}}H + b^{(2)\mathsf{T}}, \tag{7b}$$

$$\begin{aligned} q_1(X;\theta) &= [\mathrm{softmax}(Z)]_1, \\ &\vdots \\ q_K(X;\theta) &= [\mathrm{softmax}(Z)]_K. \end{aligned} \tag{7c}$$

The hidden layer $H = [H_1, \dots H_M]^\mathsf{T}$ has $M = 30$ hidden units. The weight matrices $W^{(1)}$ and $W^{(2)}$ in each of the two dense layers has the size (input units $\times$ output units) and each offset vector $b^{(1)}$ and $b^{(2)}$ has the size (1 $\times$ output units).

Question 3.4: *What are the sizes of the two weight matrices $W^{(1)}$, $W^{(2)}$ and the two offset vectors $b^{(1)}$, $b^{(2)}$ in the network above? How many parameters does the network have in total?*

Answer:

3.3 Convolutional neural network

Consider the same classification problem as in Section 3.2 but where the 144 input units represent 12×12 grayscale pixels in an image. We want to model this with a convolutional neural network (CNN).

In a CNN, the hidden units in each layer are organized in a *tensor*[3] of order 3 with the size (rows $\times$ columns $\times$ channels). Each grayscale image in our setting has the size (12 $\times$ 12 $\times$ 1). Hence, each image has only one channel since each grayscale pixel can be represented with one scalar value corresponding to the brightness of that pixel.[4]

[3] A tensor is a generalization of a matrix to more than two different dimensions. This is also what has given Tensorflow its name since the library is used to calculate and operate on tensors.

[4] For a color image we would have three channels representing the three RGB colors red, green and blue. Color images are not considered further here.

The design of a CNN is that in each layer, the units from the previous hidden layer are convolved with a patch of weights, a so-called *kernel*. We can have multiple kernels (with different parameters) operate on the same input in parallel. Each kernel then produces a new output channel for the next hidden layer. Each kernel has the size (kernel rows $\times$ kernel columns $\times$ input channels) and if we stack all parameters in all kernels in one weight tensor W, that tensor has size (kernel rows $\times$ kernel columns $\times$ input channels $\times$ output channels). Read more about CNNs in Section 4.3 in the Lecture notes. Especially Figure 4.11 in the Lecture notes might help you to answer the questions below.

Consider a CNN with two convolutional layers parameterized with $W^{(1)}, b^{(1)}$, $W^{(2)}, b^{(2)}$ and two dense layers parameterized with $W^{(3)}, b^{(3)}, W^{(4)}, b^{(4)}$.

The first convolutional layer consists of 4 kernels of size (kernel rows $\times$ kernels columns) $= (5 \times 5)$ and we use the stride [1,1] with zero-padding, i.e. the kernel is moving by one step (both row- and column-wise) during the convolution such that the first hidden layer has the same number of rows and columns as the image. As previously stated, each grayscale image in has the size (12 $\times$ 12 $\times$ 1).

Question 3.5: *What are the sizes of the weight tensor $W^{(1)}$, offset vector $b^{(1)}$ and the first hidden layer $H^{(1)}$?*

Answer:

In the following convolutional layer we use 8 kernels of size 3×3 and the stride [2,2] , i.e. the kernel is moving by two steps (both row- and column-wise) during the convolution such that the second hidden layer has half as many rows and columns as the previous first hidden layer.

Question 3.6: *What is the size of the weight tensor $W^{(2)}$, offset vector $b^{(2)}$, and the second hidden layer $H^{(2)}$?*

Answer:

After the two convolutional layers we implement two dense layers with an intermediate hidden layer of 60 hidden units before we end with a softmax function to produce the predicted class probabilities.

Question 3.7: *What are the sizes of the weight tensors and offset vectors* $W^{(3)}$, $b^{(3)}$ *and* $W^{(4)}$, $b^{(4)}$ *belonging to the two dense layers?*

Answer:

4 Laboratory exercises

This section contains the laboratory exercises to be executed during the laboratory session. The main lab exercise is to build and train a neural network to classify hand-written digits as presented in Section 4.1. The lab ends in Section 4.2 with an evaluation of a much bigger state-of-the-art network that has been trained on over a million images.

4.1 Classification of hand-written digits

In this part of the lab we will learn how to use a neural network to classify images.[5] We will consider the so called MNIST data set[6], which is one of the most well studied data sets within machine learning and image processing.

The dataset consist of 60 000 training data points and 10 000 test data points. Each data point consist of a grayscale image with 28 $\times$ 28 pixels of a handwritten digit. The digit has been size-normalized and centered within a fixed-sized image. Each image is also labeled with the digit (0,1,...,8, or 9) it is depicting. In Figure 1 a batch of 100 data points from this data set is displayed.

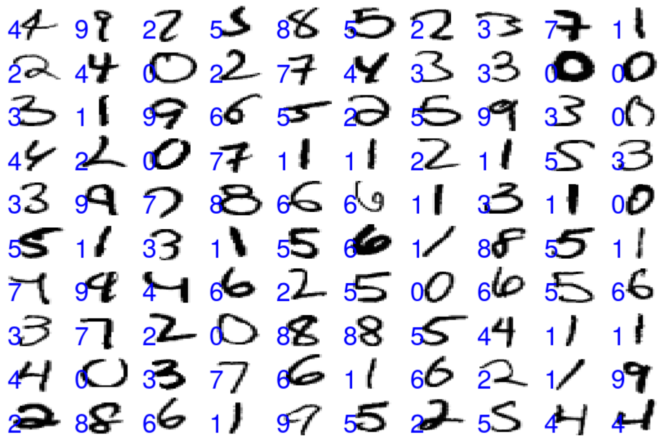


Figure 1: Some samples form the data we will use in the entire Section 4.1. The input is the pixels values of an image (black and white), and the output is the label of the digit it represents (blue).

In this classification task we consider the image as our input $X = [X_1, \dots X_p]^\mathsf{T}$. Each input variable X_i corresponds to a pixel in the image. In total we have $p = 28 \times 28 = 784$ input variables.

[5]This MNIST lab has been quite inspired by a similar one in the crash-course *Learn TensorFlow and deep learning, without a Ph.D.* available at `https://cloud.google.com/blog/big-data/2017/01/learn-tensorflow-and-deep-learning-without-a-phd`

[6]`https://en.wikipedia.org/wiki/MNIST_database`

The value of each X_j represents the color of that pixel. The color-value is within the interval [0,1], where $X_j = 0$ corresponds to a black pixel and $X_j = 1$ to a white pixel. Anything between 0 and 1 is a gray pixel with corresponding intensity.

We have in total 10 classes representing the 10 digits. We will use the so called one-hot encoding for the indicator output described in Section 3. This means that the $y = [1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]^\mathsf{T}$ represents the digit "0", $y = [0\ 1\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ 0]^\mathsf{T}$ represents the digit "1", and so forth. Consequently, the output y has 10 dimensions.

Based on a set of training data $\{x_i, y_i\}_{i=1}^n$ with images and labels, the problem is to find a good model for the class probabilities

$$\text{Pr}(Y = k|X), \qquad k = 1, \dots, 10, \tag{8}$$

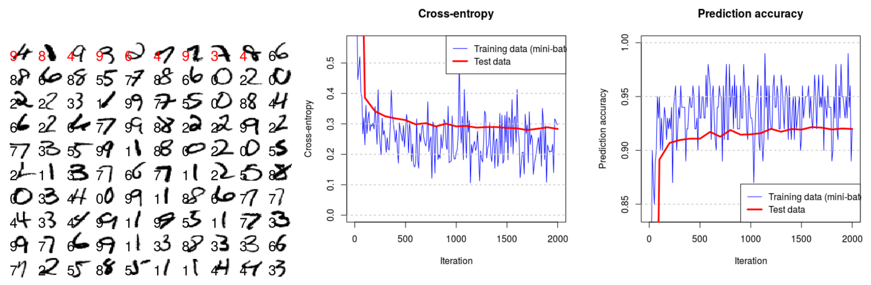
i.e. the probabilities that an unseen image X belongs to each of the 10 classes.

4.1.1 Preparation

Download the zip-file `DLlab_code.zip` from the course homepage `http://www.it.uu.se/edu/course/homepage/sml/lab`, save it on your computer and unzip it. Launch RStudio and open `mnist_onelayer.R`.

4.1.2 Run the code sample

Task 4.1 Run the code as it is. After a while when the training is done, three figures appears, see Figure 2. ○



(a) (b) (c)

Figure 2: Three figures generated by the code. Figure 2a: Prediction performance on 100 randomly selected test points. Figure 2b: Cost function (cross-entropy) on test/training data. Figure 2c: Prediction accuracy on test/training data.

In Figure 2a, 100 randomly selected test images (out of the total 10 000 test images) are displayed together with their predicted label. For those images that have been incorrectly classified, the label is colored red.

To train the network we minimize a cost function that tells how bad we are at predicting the training data correctly. The cost function for this problem is the cross-entropy (more about that in Section 4.1.4). Figure 2b displays the cost function on test and training data with the iteration number on the x-axis.

In Figure 2c the prediction accuracy on test and training data[7] is displayed. The prediction accuracy on test data is the performance measure that we are mostly interested in. The cross-entropy and prediction accuracy on both training data and test data as displayed on the figures are also printed in the terminal during training.

Question 4.1: *What classification accuracy do you get on the test data? What about the digits that are missclassified, do you see any pattern?*

Answer:

[7]The prediction accuracy on training data is evaluated on only 100 randomly selected training samples used during training (called mini-batch). That is why the training accuracy is so “noisy”. More about mini-batch and the training procedure in Section 4.1.4.

4.1.3 Understand the model

The sample code is an implementation of a one-layer neural network with softmax transformation of the output. For a certain class k and data point i with $x_i = [x_{i1}, \ \ldots, \ x_{ip}]^\mathsf{T}$ the model of the class probabilities $\Pr(Y = k | X = x_i)$ is

$$q_k(x_i; \theta) = \frac{e^{z_{ik}}}{\sum_{l=1}^{K} e^{z_{il}}} \quad \text{where} \quad z_{ik} = \beta_{0k} + \sum_{j=1}^{p} x_{ij}\beta_{jk}, \tag{9}$$

where θ is a vector with all the parameters β_{jk} in the model. With n training data points $\{x_i, y_i\}_{i=1}^{n}$ and $k = 1, \ldots, K$ classes we can write the model in matrix notation

$$\mathbf{Q} = \text{softmax}(\mathbf{X}W + b), \tag{10}$$

where

$$\mathbf{Q} = \begin{bmatrix} q_1^\mathsf{T} \\ \vdots \\ q_n^\mathsf{T} \end{bmatrix}, \quad \mathbf{X} = \begin{bmatrix} x_1^\mathsf{T} \\ \vdots \\ x_n^\mathsf{T} \end{bmatrix}, \quad W = \begin{bmatrix} \beta_{11} & \ldots & \beta_{1K} \\ \vdots & & \vdots \\ \beta_{p1} & \ldots & \beta_{pK} \end{bmatrix}, \quad b = \begin{bmatrix} \beta_{01} & \ldots & \beta_{0K} \end{bmatrix},$$

with $q_i^\mathsf{T} = [q_1(x_i; \theta), \ldots, q_K(x_i; \theta)]$, and where W and b are the *weight matrix* and the *offset vector*, respectively. The offset vector b is added to all n rows in (10). Also the softmax function is applied row by row as

$$\text{softmax}(z_i^\mathsf{T}) = \frac{1}{\sum_{l=1}^{K} e^{z_{il}}} \begin{bmatrix} e^{z_{i1}} & \ldots & e^{z_{iK}} \end{bmatrix}, \quad \text{where} \quad z_i^\mathsf{T} = x_i^\mathsf{T} W + b. \tag{11}$$

Remember, the ith row in $\mathbf{X}$ (i.e. x_i^T) contains the 784 pixel values of image i and the ith row in $\mathbf{Q}$ (i.e. q_i^T) contains the 10 probabilities that the ith image belongs to each of the 10 classes. The weight matrix W and the offset vector b contains all the parameters of the model.

Task 4.2 Make sure you understand the model above. Read the code down to line 30 and make sure that you can map the model presented above to what is implemented. If something is unclear, ask the lab supervisors! ○

Question 4.2: *How many parameters does this model have?*

Answer:

4.1.4 Understand the training

To find good parameters, we need to train the model. This requires a cost function. The cost function describes the "distance" between the outputs $y_i = [y_{i1} \; \dots \; y_{iK}]^\mathsf{T}$ and the predicted class probabilities $q_1(X;\theta), \dots, q_K(X;\theta)$. For classification we typically use the cross-entropy cost function

$$J(\theta) = -\frac{1}{n}\sum_{i=1}^{n}\sum_{k=1}^{K} y_{ik}\log(q_k(x_i;\theta)), \tag{12}$$

where $\theta = \{W, b\}$ are the parameters. Remember, if y_i encodes the hand-written digit "3", all ten elements in y_i will be zero except the forth element, which will be one. Each $q_k(x_i;\theta)$ is a value between 0 and 1 which corresponds to the probability that the image i belongs to class k.

The training of the network is done by minimizing the cost function (12) in a loop. In each iteration, we compute the gradient $\nabla_\theta J(\theta)$ of the cost function with respect to the parameters and update the parameters by going a small step in the opposite direction of the gradient

$$\theta_{t+1} := \theta_t - \gamma\nabla_\theta J(\theta). \tag{13}$$

This is called gradient descent and γ is the learning rate. In each gradient step we do not use all $n = 60\,000$ training data to compute the cost function (12) and its gradient. Instead we use 100 randomly selected training data points in each iteration. We call such a group of training data a *mini-batch*.

Next iteration we randomly selected 100 new training data points (of the ones which have not yet been selected) and compute the gradient based on this group of data. We continue until all training data points have been used. One such sweep through the training data is called an *epoch*. After one epoch is completed, we start the process over again. This training procedure is called *stochastic gradient descent*.

Task 4.3 Make sure you understand the training procedure described above. Read through the code down to line 68. Map the training procedure described above to what is written in the code. Ask the lab supervisors if anything is unclear! ○

Question 4.3: *How many iterations does it take until we have seen all training data points, i.e., how many iterations are included in each epoch?*

Answer:

4.1.5 Train a two-layer neural network

We will now add a second layer to the network. We get the model

$$\mathbf{H} = \sigma(\mathbf{X}W_1 + b_1), \tag{14a}$$

$$\mathbf{Q} = \text{softmax}(\mathbf{H}W_2 + b_2), \tag{14b}$$

where $\mathbf{H}$ is an intermediate layer of hidden units. For this we need two weight matrices W_1, W_2 and two offset vectors b_1, b_2.

```
M <- 200L # Number of hidden units
W1 <- tf$Variable(tf$truncated_normal(shape(784L, M),stddev = 0.1))
b1 <- tf$Variable(tf$zeros(shape(M)))
W2 <- tf$Variable(tf$truncated_normal(shape(M, 10L),stddev = 0.1))
b2 <- tf$Variable(tf$zeros(shape(10L)))
```

We keep the softmax as activation function on the last layer, but uses the logistic function also called the *sigmoid function*[8] after the first layer. The sigmoid function is defined as $\sigma(x) = 1/(1 + e^{-x})$. The model will then be

```
H <- tf$nn$sigmoid(tf$matmul(X, W1) + b1)
Q <- tf$nn$softmax(tf$matmul(H, W2) + b2)
```

When we grow the network deeper, it is important to initialize the weights randomly. If not, we easily get stuck in local minima during the minimization. In the lines above, each weight is initialized by a value drawn from a truncated normal distribution with standard deviation 0.1, which work fine.

Task 4.4 Save `mnist_onelayer.R` as a new file `mnist_twolayers.R`. Add one more layer with 200 hidden units. ○

Question 4.4: *What classification accuracy on test data do you get? Try some other numbers of hidden units. How does it affect the performance? What happens if you initialize the weights with zeros as we did for the single layer neural network?*

Answer:

[8]This is the same function as the logistic function (3). However, here it appears in a different context, hence the different name.

4.1.6 Go even deeper!

Now we can continue to go even deeper!

Task 4.5 Save `mnist_twolayers.R` as a new file `mnist_fivelayers.R`. Continue to add even more layers. Add in total five layers with 200, 100, 60, and 30 hidden units between each layer. ◦

Question 4.5: *What classification accuracy on test data do you get?*

Answer:

When you go deeper there are some things to be aware of

- The sigmoid activation function causes issues in deep networks. It squashes all inputs into [0,1] and when doing so repeatedly in multiple layers the gradients with respect to the parameters in the deepest layers might get close to zero. Instead, you might consider using the *Rectified Linear Unit* (ReLU) activation function $\sigma(x) = \max(0, x)$. To use ReLU, simply replace all `tf$nn$sigmoid` in the code with `tf$nn$relu`.
- Initialize all weights randomly! If you have not yet done so, do that now! For the offset vectors, when using ReLU:s, the best practice is to initialize them to a small positive values such that it operates in the non-negative range of ReLU. We can initialize all offset parameters with 0.1 as `b5 <- tf$Variable(tf$ones(shape(10L))/10)`.
- In problems where we have a lot of parameters (how many do we have now?) we also have many so-called "saddle points". The gradient descent algorithm tends to get stuck at such saddle points. A better optimizer for dealing with this problem is the Adam-optimizer. To train with that routine, simply replace `tf$train$GradientDescentOptimiser` with `tf$train$AdamOptimizer`. When swapping optimization routine also change the learning rate `gamma` from `0.5` to `0.005`.

Task 4.6 Improve the training of the network by using the tips above. Does this improve the prediction accuracy? ◦

Task 4.7 Is the model done training after 2 000 iterations or do you think you can get an even better classification accuracy if you train longer? Try to train for 10 000 iterations. ◦

By training this deeper model longer, you might suddenly get a very poor performance due to numerical issues. Look on the next page why this happens and how to solve it.

4.1.7 Train longer, be aware of numerical issues!

If you train the network longer, some elements in q_i start getting very close to zero (meaning that the network is very certain that image i does not belong to that class). This is a problem in (12) since $\log(0)$ will then output `NaN` (not-a-number), so also the whole cross-entropy, and the network will not be able to train anything.

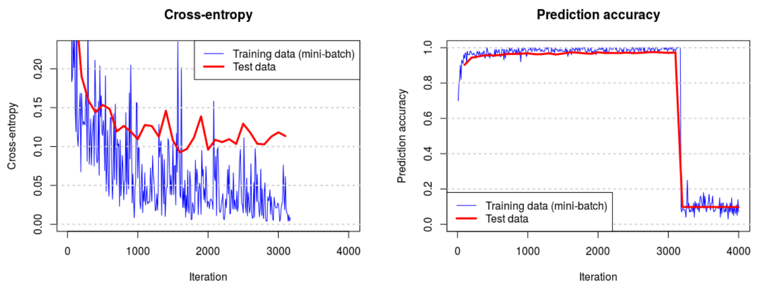


Figure 3: If some elements in q_i get very close to zero, the logarithm function outputs not-a-number and the performance crashes.

To circumvent this numerical issue, the softmax (5) and the cross-entropy (12) can be computed in one step. Since the softmax (5) is an exponential (with normalization) no logarithm of a small number is needed since $\log(e^t) = t$ even if, say, $t = -100$ and $e^{-100} \approx 0$.

In Tensorflow, softmax and cross-entropy can be computed in one step with `tf$nn$softmax_cross_entropy_with_logits`. First we have to rewrite the line `Q <- tf$nn$softmax(tf$matmul(H4, W5) + b5)` into two steps

```
Z <- tf$matmul(H4, W5) + b5
Q <- tf$nn$softmax(Z)
```

and `Z` (called logits) goes into the combined softmax-cross-entropy function

```
cross_entropy <-
tf$nn$softmax_cross_entropy_with_logits(logits = Z, labels = Y)
cross_entropy <- tf$reduce_mean(cross_entropy)
```

Task 4.8 Implement the suggested change above to make the code more robust. Try again to train for 10 000 iterations. ○

Question 4.6: *What classification accuracy on test data do you get?*

Answer:

4.1.8 Use convolutional neural networks

All models that we considered so far started with putting all the 28×28 pixels in each image into a long vector with 784 elements. This partially destroys the spatial information present in the images. In contrast, a convolutional neural network (CNN) exploits this information, which enables us to achieve an even better classification performance. With this adjustment you should be able to reach approximately 98.5% prediction accuracy!

We will use a CNN with three convolutional layers and two final dense layers. The settings for the three convolutional layers are given in Table 1 and a graphical illustration of the whole network in Figure 4.

Table 1: Architecture of the three convolutional layers.

	Layer 1	**Layer 2**	**Layer 3**
Number of kernels/output channels	4	8	12
Kernel rows and columns	(5×5)	(5×5)	(4×4)
Stride	[1,1]	[2,2]	[2,2]

In a convolutional layer, the weight tensor W has the size (kernel rows $\times$ kernel columns $\times$ input channels $\times$ output channels). The weight tensor and offset vector for the first convolutional layer are implemented as

```
M1 <- 4L
W1 <- tf$Variable(initial_value =
   tf$truncated_normal(shape(5L,5L,1L, M1),stddev = 0.1))
b1 <- tf$Variable(initial_value = tf$ones(shape(M1))/10)
```

and the corresponding model implementation for that convolutional layer as

```
stride <- 1
H1 <- tf$nn$relu(tf$nn$conv2d(Xim, W1, strides =
   shape(1L, stride, stride, 1L), padding='SAME') + b1)
```

Note that we use the non-vectorized version of the images stored in `Xim`. Before we used `X` where the input dimensions were reshaped into a vector.

Do not bother about the two `1L` on first and fourth place in the stride syntax. They are not important here. Furthermore, `padding='SAME'` means that we use zero-padding such that we get equally many columns and rows for `H1` as we have for `Xim` (if we use stride = 1).

Task 4.9 Save `mnist_fivelayer.R` as a new file `mnist_CNN.R`. Replace the first three dense layers in the previous code with three convolutional layers using the settings in Table 1 for each of these three layers. ○

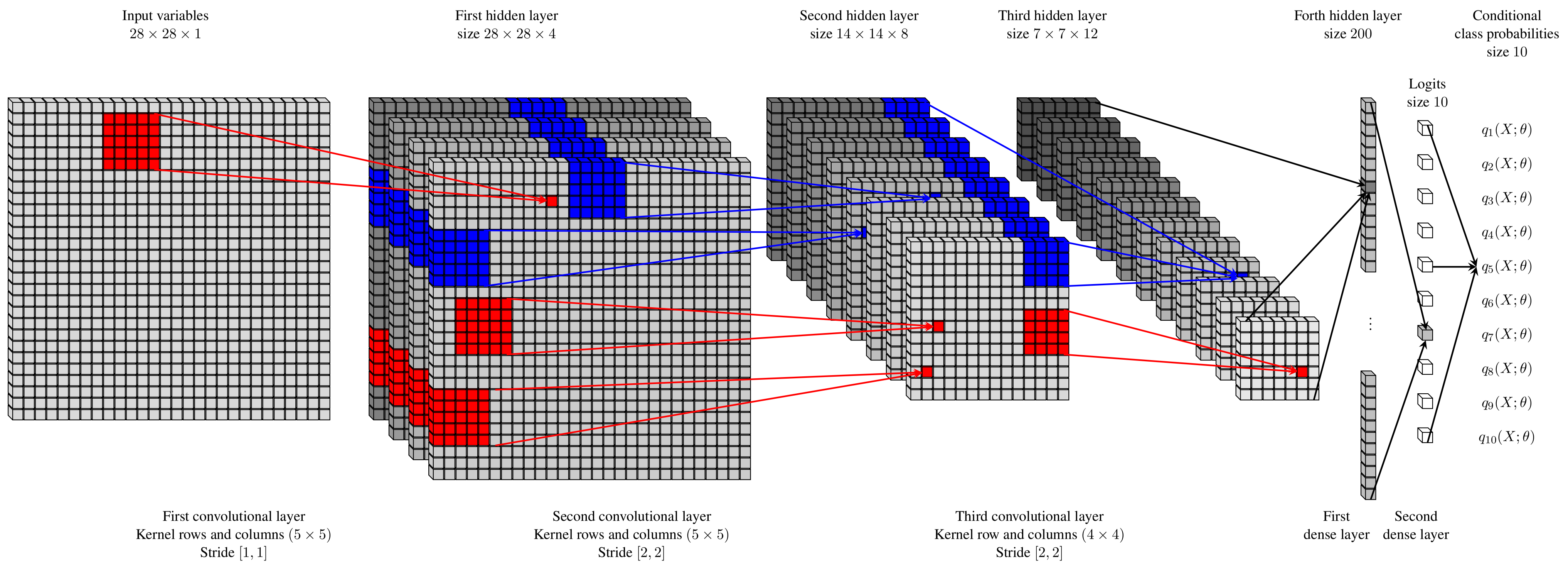


Figure 4: The architecture of the convolutional neural network in the lab. It consists of three convolutional layers and two dense layers. The leftmost square is the input image where each cube represents a pixel. The following three stacks are the first three hidden layers produced by the three convolutional layers organized in rows, columns and channels, where each cube represents a hidden unit. A certain hidden unit in one of these hidden layers depends on a patch of rows and columns and all channels from the previous layer, here illustrated as colored regions. Each hidden unit is multiplied with parameters in a kernel. The same kernel is used for all hidden units in a certain channel. A different kernel is used to produce a different channels, here illustrated with the different colors red and blue. The filter size is 5×5, 5×5 and 4×4 for the three convolutional layers. The network ends with two dense layers with an intermediate hidden layer and a final softmax function to produce class probabilities as output.

After the three convolutional layers, we use two dense layers. Before we can apply the first dense layer, all hidden units produced by the third convolutional layer needs to be flattened into a long vector. This can be done with the command

```
H3flat <- tf$reshape(H3, shape=shape(-1L,M3))
```

where the tensor H3 holds the hidden units after the third convolutional layer and has the size (batch size $\times$ rows $\times$ columns $\times$ channels). The command reshapes H3 into a tensor H3flat of size (batch size $\times M_3$). The -1L means that the size of the first dimension representing the batch size is preserved. The vectorized H3flat then the input to the first dense layer.

Question 4.7: *How many hidden units are there in total after the third convolutional layer, i.e. what is* M3 *supposed to be in the code above? Hint: We have applied a stride of 2 twice were each of them reduced the number of rows and columns in the original image by a factor of 2.*

Answer:

Finally, the dense layer after the final convolutional layer should have 200 hidden units.

Task 4.10 Find the value of M3 and add the reshaping command according to what is stated above. Update the forth and fifth layer according to the instructions above. Train for at least 4 000 iterations. Note that the training of the CNN takes a bit longer, be patient! ○

Question 4.8: *What prediction accuracy do you achieve?*

Answer:

The next two pages contains tasks that are not mandatory (but, of course, interesting!). The mandatory part of the lab continues in Section 4.2 on page 27.

4.1.9 Improve learning rate* (extras, not mandatory)

In the end of the training, the learning rate $\gamma = 0.005$ is really too fast. Neither the prediction accuracy nor the cross-entropy on test data really converges and we will not get down to the best minimum of the cost function. You would therefore prefer having a lot smaller γ, but with a very small γ the training takes too long. One solution is to start learning fast (to get approximately close to the minimum) and then slow down. Consider adjusting the learning rate γ as

$$\gamma_t = \gamma_{\text{min}} + (\gamma_{\text{max}} - \gamma_{\text{min}})e^{-\frac{t}{2000}}, \tag{15}$$

where $\gamma_{\text{max}} = 0.003$, $\gamma_{\text{min}} = 0.0001$, and t being the iteration number. This means that we start with a learning rate of $\gamma_{\text{max}} = 0.003$ and approaches $\gamma_{\text{min}} = 0.0001$ as $t \to \infty$.

To implement this you need to implement γ_t as a placeholder and in each loop feed the new learning rate to γ_t.

Task (optional) 4.11 Improve the training by adjusting the learning rate according to the formula above or any other formula that you come up with! Train for at least 6 000 iterations. You should now be able to push the performance above 99% prediction accuracy on test data! ○

Question 4.9: *Do you manage to get* 99% *prediction accuracy on training data?*

Answer:

Question 4.10: *At this point, the network can start overfitting on training data. How can you see that? What could you do to avoid this overfitting from happening?*

Answer:

4.1.10 Regularize with dropout* (extras, not mandatory)

When we extend our network and start seeing signs of over-fitting we know that there is room for improvement! One way to avoid the over-fitting is to reduce the size of the network. However, in practice a better strategy is to extend it a bit more and add regularization to the network. In Lecture 9 we talk about a popular regularization method for neural networks called *dropout*.

In dropout we at each training iteration remove a random selection of hidden units together with its incoming and outgoing links. We then train the remaining part of the network as if the removed units were not present. At each training iteration a new random selection of hidden units are removed. During test time all hidden units are used but their outputs are scaled with the probability that they were present during training. Read more about it in Section 4.4.4 in the Lecture notes.

Here we choose to add dropout to the last hidden layer, since this will affect the weights in the first dense layer where most of the weight in our network reside. This is implemented as

```
H4dropout <- tf$nn$dropout(H4, keep_prob = pkeep)
```

where `pkeep` is the probability that a hidden unit in that hidden layer is kept. During training we keep 75% of the hidden units and during testing we want keep all 100% of the hidden units. Therefore, we need to implement `pkeep` as a placeholder and in each training loop feed the value `pkeep = 0.75` and during testing feed the value `pkeep = 1.00`.

Task (optional) 4.12 Regularize the network by implementing dropout according to the description above. Train for 6 000 iterations. ○

Question 4.11: *What classification performance do you achieve? Does it seems like you are doing less overfitting?*

Answer:

Task (optional) 4.13 Play around with number of layers, channels, stride, kernels size, dropout, learning rate etc. to achieve an even better classification performance. For example extending network to 6,12,24 kernels with the sizes (6×6), (5×5), (4×4) in the three conv layers might give an even better performance. If you overfit again, fight back with some more dropout. ○

4.2 Real world image classification

We have implemented and trained five layer CNN. We also learned how to deal with several practical issues that appear when training a deeper network. Working with the MNIST data set, we went from $\approx 92\%$ prediction accuracy on test data up to $\approx 99.2\%$. This is quite close to the world record on 99.77%! See the full "leaderboard" on `http://yann.lecun.com/exdb/mnist/`.

The MNIST data set is a fairly small data set in a deep learning context. In this final lab exercise, we will use a network that has been trained on 1.2 million images provided by ImageNet[9] used in the *Large Scale Visual Recognition Challenge* 2012-2014, see Figure 5 for a few training data examples. Each image is labeled (by hand!) with presence or absence of one of 1000 object categories[10]. Simonyan and Zisserman (2014) provided the winning contribution of the 2014 competition called VGG16. See Table 2 for a comparison between the model used previously in this lab trained on MNIST, and the VGG16 model trained on the ImageNet data.

Figure 5: 100 images from ImageNet

Table 2: Comparison between the MNIST classification problem in lab with the VGG16 network trained on ImageNet.

	This lab	**VGG16**
Dataset	MNIST	ImageNet (only a subset)
Training data size	60 000	1 200 000
Test data size	10 000	150 000
Nr of pixels	$28 \times 28 = 784$	$224 \times 224 = 50'176$
Nr of image channels (colors)	1 (only gray-scale)	3 (RGB images)
Nr of class labels	10	1 000
Nr of layers	5	16
Nr of parameters	122 260	138 000 000
Training time	8 min[11]	2-3 weeks[12]

To train such a network to get good weights, a lot of time and computational resources are required. However, when that network already has been trained it

[9] `http://www.image-net.org/`

[10] Class categories: `http://image-net.org/challenges/LSVRC/2014/browse-synsets`

[11] With 10 000 iterations on a laptop

[12] On a system equipped with four NVIDIA Titan Black GPUs

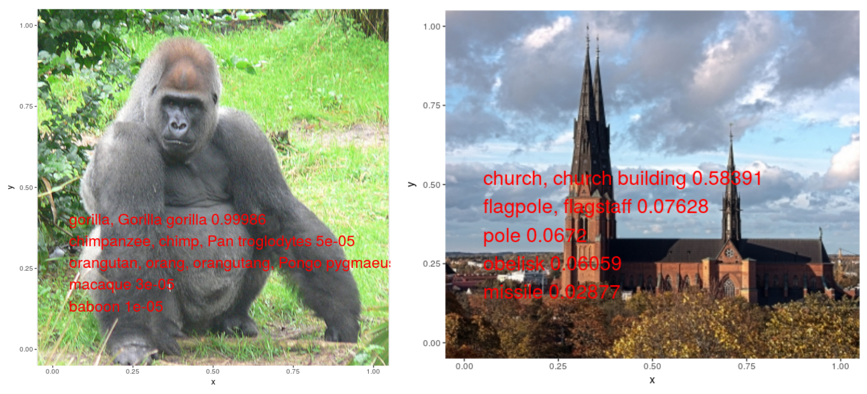

gorilla, Gorilla gorilla 0.99986
chimpanzee, chimp, Pan troglodytes 5e-05
orangutan, orang, orangutang, Pongo pygmaeus
macaque 3e-05
baboon 1e-05
church, church building 0.58391
flagpole, flagstaff 0.07628
pole 0.0672
obelisk 0.06059
missile 0.02877
x
y
0.00
0.25
0.50
0.75
1.00

Figure 6: A few results from the VGG16 classifier trained on data from ImageNet.

is fast to predict the class of a new unseen image since no additional training is needed. Here, we will predict the class for images of our choice using the pre-trained network VGG16 presented above.

Task 4.14 Open `VGG16_classification.R`[13] and in RStudio select Session -> Set working directory -> To source file location. Look at the code, you don't have to understand the details. Note that the code will load precomputed weights `vgg_16.ckpt` instead of training them on a training data set.[14] Run it and look at the output. ○

Task 4.15 Download some images (jpeg-format) from internet and change the line where the image is loaded accordingly. How well does it perform on your choice of images? Figure 6 displays some results that we got. ○

References

Simonyan, K. and Zisserman, A. (2014). Very Deep Convolutional Networks for Large-Scale Image Recognition. Technical report, arXiv:1409.1556.

[13]The R-code is an adjusted version of `http://randomthoughtsonr.blogspot.se/2016/11/image-classification-in-r-using-trained.html`

[14]If you are on your own computer, the file `vgg_16_2016_08_28.tar.gz` (500 MB) from here `http://download.tensorflow.org/models/vgg_16_2016_08_28.tar.gz`. Unpack the file to get the file `vgg_16.ckpt`. This file includes the weights of the pre-trained network. Change the path on line 60 in `VGG16_classification.R` to the global path where you have stored `vgg_16.ckpt`.