Exercise 04 — Solution Reference

I2DL — Introduction to Deep Learning

exercise_code/networks/classifier.py

```
################################################################################
# TODO:                                                                        #
# Implement the forward pass and return the output of the model. Note  #
# that you need to implement the function self.sigmoid() for that.     #
# Also, save in self.cache an array of all the relevant variables that #
# you will need to use in the backward() function. E.g.: (X, ...)      #
#                                                                              #
# Hint:                                                                        #
#       The function is made up of TWO functions: Affine and sigmoid.  #
#       The sigmoid is applied to the result of the affine step.       #
################################################################################

y = X.dot(self.W)
z = self.sigmoid(y)

# Save the samples for the backward pass
self.cache = (X, z)


################################################################################
#                             END OF YOUR CODE                                 #
################################################################################
```

```
################################################################################
# TODO:                                                                        #
# Implement the backward pass. Return the gradient w.r.t W --> dW.     #
# Make sure you've stored ALL needed variables in self.cache. (!!!!)   #
#                                                                              #
# Hint 1: It is recommended to follow the TUM article (Section 3) on   #
# calculating the chain-rule, while dealing with matrix notations:     #
# https://bit.ly/tum-article                                                   #
#                                                                              #
# Hint 2: Remember that the derivative of sigmoid(x) is independent of #
# x, and could be calculated with the result from the forward pass.    #
#                                                                              #
# Hint 3: The argument "dout" stands for the upstream gradeint to this #
# layer.                                                                       #
################################################################################

# We calculate the derivatives in order, like in the chain rule.
# Let us denote y = XW + b, z = sigmoid(y)

X, z = self.cache
```

```
# 1) dl/dy = dL/dz * dz / dy. According to stanford's trick, with "dout" standing for dl / dz:
dz_dy = z * (1 - z)
dl_dy = dout * dz_dy  # Now, this is the upstream derivative for step 2.

# 2) dl/dw = dl/dy * dy/dw. According to stanford's trick:
dW = X.T.dot(dl_dy)


##########################################################################
#                             END OF YOUR CODE                           #
##########################################################################
```

```
##########################################################################
# TODO:                                                                  #
# Implement the sigmoid function over the input x. Return "out".        #
# Note: The sigmoid() function operates element-wise.                    #
##########################################################################

out = 1 / (1 + np.exp(-x))

##########################################################################
#                             END OF YOUR CODE                           #
##########################################################################
```

exercise_code/networks/loss.py

```
##########################################################################
# TODO:                                                                  #
# Implement the forward pass and return the output of the BCE loss      #
# for each instance in the batch.                                        #
#                                                                        #
##########################################################################

result = - (y_truth * np.log(y_out) + (1 - y_truth) * np.log(1 - y_out))

##########################################################################
#                             END OF YOUR CODE                           #
##########################################################################
```

```
##########################################################################
# TODO:                                                                  #
# Implement the backward pass. Return the gradient w.r.t to the input   #
# to the loss function, y_out.                                           #
#                                                                        #
# Hint:                                                                  #
#   Don't forget to divide by N, which is the number of samples in      #
#   the batch. It is crucial for the magnitude of the gradient.         #
##########################################################################
```

```
gradient = (-(y_truth / y_out) + (1 - y_truth) / (1 - y_out)) / len(y_truth)

###########################################################################
#                           END OF YOUR CODE                              #
###########################################################################
```

exercise_code/networks/optimizer.py

```
###########################################################################
# TODO:                                                                   #
# Implement the gradient descent step over the weight, using the          #
# learning rate.                                                          #
###########################################################################

weight -= self.lr * dw

###########################################################################
#                           END OF YOUR CODE                              #
###########################################################################
```

exercise_code/solver.py

```
###########################################################################
#   TODO: ^^I^^I^^I^^I^^I^^I^^I^^I^^I^^I^^I^^I           #
#^^IPerform the optimizer step, on higher level of abstraction.         #
#   Simply call the relevant functions of your model and the loss       #
#   function, according to the deep-learning pipline. Then, use         #
#   the optimizer variable to perform the step.                         #
#                                                                       #
#   Hint 1: What inputs each step requires? How do we obtain them?      #
#                                                                       #
#   Hint 2: Don't forget the order of operations: forward, loss,        #
#   backward.                                                           #
###########################################################################

model_forward = model.forward(X_train)
loss = loss_func(model_forward, y_train)
loss_grad = loss_func.backward(model_forward, y_train)

grad = model.backward(loss_grad)
opt.step(grad)
###########################################################################
#                           END OF YOUR CODE                              #
###########################################################################
```