Course: Deep Learning *Coordinator:* Jakub Tomczak

Submission Assignment 1

Name: Max Faber, *Student ID:* 2676067

Please provide (concise) answers to the questions below. If you don't know an answer, please leave it blank. If necessary, please provide a (relevant) code snippet. If relevant, please remember to support your claims with data/figures.

Question 1

Work out the local derivatives of both, in scalar terms. Show the derivation. Assume that the target class is given as an integer value.

Answer

$$\frac{\partial l}{\partial y_i} = \frac{\partial(-\log y_c)}{\partial y_i} = -\frac{1}{y_c} = -\frac{1}{y_i} \qquad \text{(iff. } c = i\text{)}$$
$$= 0 \qquad \text{(iff. } c \neq i\text{)}$$

Eq. 1: Derivative of the cross-entropy loss function w.r.t. y_i

To get the derivative of the softmax function, the quotient rule is applied:

$$\begin{aligned}\frac{\partial y_i}{\partial o_i} &= \frac{\partial(\frac{e^{o_i}}{\sum_j e^{o_j}})}{\partial o_i} \\ &= \frac{e^{o_i}\sum_j e^{o_j} - e^{o_j}e^{o_i}}{(\sum_j e^{o_j})^2} \\ &= \frac{e^{o_i}(\sum_j e^{o_j} - e^{o_j})}{(\sum_j e^{o_j})^2} \\ &= \frac{e^{o_i}}{\sum_j e^{o_j}} \cdot \frac{\sum_j e^{o_j} - e^{o_j}}{\sum_j e^{o_j}} \\ &= \frac{e^{o_i}}{\sum_j e^{o_j}} \cdot \frac{\sum_j e^{o_j}}{\sum_j e^{o_j}} - \frac{e^{o_j}}{\sum_j e^{o_j}} \\ &= y_i(1 - y_j) \qquad \text{(iff. } i = j\text{)}\end{aligned}$$

$$\begin{aligned}\frac{\partial y_i}{\partial o_j} &= \frac{\partial(\frac{e^{o_i}}{\sum_j e^{o_j}})}{\partial o_j} \\ &= \frac{e^{o_i}\cdot 0 - e^{o_j}e^{o_i}}{(\sum_j e^{o_j})^2} \\ &= \frac{-e^o_j}{\sum_j e^{o_j}} \cdot \frac{e^{o_i}}{\sum_j e^{o_j}} \\ &= -y_j y_i \qquad \text{(iff. } i \neq j\text{)}\end{aligned}$$

Eq. 2: Derivative of softmax activation function w.r.t. o_j

Question 2

Work out the derivative $\frac{\partial l}{\partial o_i}$. Why is this not strictly necessary for a neural network, if we already have the two derivatives we worked out above?

Answer

$$\frac{\partial l}{\partial o} = \frac{\partial l}{\partial y} \cdot \frac{\partial y}{\partial o}$$
$$\frac{\partial l}{\partial o_i} = \frac{\partial(-\log y_c)}{\partial o_i} = -\frac{1}{y_i} \cdot \frac{\partial y_i}{\partial o_i} = -\frac{1}{y_i} \cdot y_i(1 - y_j) = 1 - y_j \qquad \text{(iff. } i = j\text{)}$$
$$\frac{\partial l}{\partial o_i} = \frac{\partial(-\log y_c)}{\partial o_i} = -\frac{1}{y_i} \cdot \frac{\partial y_i}{\partial o_i} = -\frac{1}{y_i} \cdot -y_j y_i = -y_j \qquad \text{(iff. } i \neq j\text{)}$$

Eq. 3: Derivative of the cross-entropy loss function w.r.t. o_i

The derivation of the cross-entropy loss function w.r.t. o_i is not strictly necessary for a neural network because, the product of the intermediate derivations $\frac{\partial l}{\partial y_i}$ and $\frac{y_i}{o_j}$ will lead to the same result.

Question 3

*Implement the network drawn in the image below, including the weights. Perform one forward pass, up to the loss on the target value, and one backward pass. Show the relevant code in your report. Report the derivatives on all weights (including biases). Do not use anything more than plain Python and the **math** package.*

```python
# Calculate results of the hidden layer (k, h)
for j in range(layer_sizes['n_hidden']):
    for i in range(layer_sizes['n_inputs']):
        k[j] += params['w'][i][j] * x[i]
    k[j] += params['b'][j]
    h[j] = sigmoid(k[j])
# Calculate the results of the output layer (o, y)
for j in range(layer_sizes['n_outputs']):
    for i in range(layer_sizes['n_hidden']):
        o[j] += h[i] * params['v'][i][j]
    o[j] += params['c'][j]
y = softmax(O=o)
```

Listing 1: Forward pass

```python
# Calculate gradients of the output layer (dl_dy, dy_do, dy_dc)
dl_dy[y] = -1. / cache['y'][y]
for j in range(layer_sizes['n_outputs']):
    for i in range(layer_sizes['n_outputs']):
        if i == j:
            dy_do[j] += dl_dy[i] * cache['y'][j] * (1. - cache['y'][i])
        else:
            dy_do[j] += dl_dy[i] * -cache['y'][j] * cache['y'][i]
    do_dc[j] = dy_do[j]
# Calculate gradients of the hidden layer (do_dv, do_dh)
for j in range(layer_sizes['n_outputs']):
    for i in range(layer_sizes['n_hidden']):
        do_dv[i][j] = dy_do[j] * cache['h'][i]
        do_dh[i] += dy_do[j] * params['v'][i][j]
# Calculate gradients of the input layer (dh_dk, dk_dw, dk_db)
for j in range(layer_sizes['n_hidden']):
    dh_dk[j] = do_dh[j] * cache['h'][j] * (1. - cache['h'][j])
    for i in range(layer_sizes['n_inputs']):
        dk_dw[i][j] = dh_dk[j] * x[i]
    dk_db[j] = dh_dk[j]
```

Listing 2: Backward pass

Running the listed forward and backward pass with the provided weights, biases, input and output results in the following derivatives:

Local derivative	**Values**
$\partial o/\partial v$	[[-0.4404, 0.4404], [-0.4404, 0.4404], [-0.4404, 0.4404]]
$\partial o/\partial c$	[-0.5, 0.5]
$\partial k/\partial w$	[[0.0, 0.0, 0.0], [0.0, 0.0, 0.0]]
$\partial k/\partial b$	[0.0, 0.0, 0.0]

Table 1: Derivatives after forward & backward pass

Question 4

Implement a training loop for your network and show that the training loss drops as training progresses.

Answer In figure 1 we see that after running the network on the synth training dataset for 100 epochs with a learning rate of 0.01, there's a little fluctuation during the first few epochs. Afterwards, the loss drops very fast which indicates that the network is picking up the necessary patterns in the data. After approximately 15 epochs we see that the network slows down converging and ultimately reaches a loss of around 0.05.

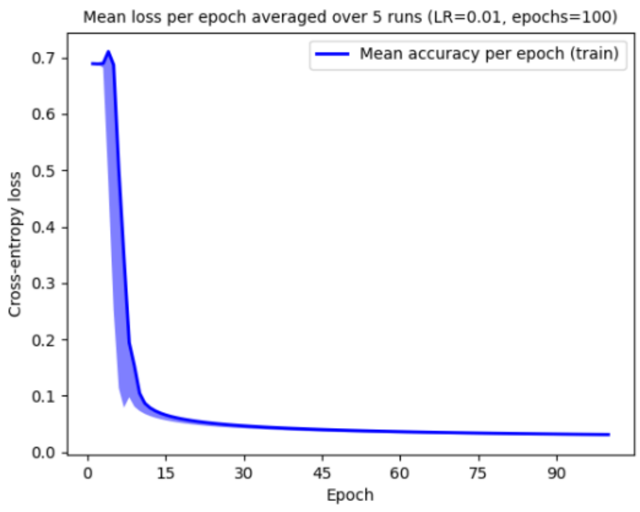


Figure 1: Mean training loss after 100 epochs on the synth dataset

Question 5

Implement a neural network for the MNIST data. Use two linear layers as before, with a hidden layer size of 300, a sigmoid activation, and a softmax activation over the output layer, which has size 10.

```
# Forward pass of first layer
k = np.matmul(X, params['w']) + params['b']
h = sigmoid(k)
# Forward pass of second layer
o = np.matmul(h, params['v']) + params['c']
y = softmax(o)
```

Listing 3: 2D version of the forward pass

```
# Convert the target class to a one-hot encoding
Y_one_hot = np.zeros(layer_sizes['n_outputs'])
Y_one_hot[Y] = 1
# Backward pass of second layer
dy_do = cache['y'] - Y_one_hot
dy_dc = np.copy(dy_do)
do_dv = np.outer(cache['h'], dy_do)
do_dh = np.matmul(params['v'], dy_do)
# Backward pass of first layer
dh_dk = do_dh * cache['h'] * (1. - cache['h'])
dk_dw = np.outer(X, dh_dk)
dk_db = np.copy(dh_dk)
```

Listing 4: 2D version of the backward pass

Please refer to the submitted code for a more in depth view of the implementation.

Question 6

Work out the vectorized version of a batched forward and backward. That is, work out how you can feed your network a batch of images in a single 3-tensor, and still perform each multiplication by a weight matrix, the addition of a bias vector, computation of the loss, etc. in a single `numpy` call.

```
# Forward pass of first layer
k = np.matmul(X, params['w']) + params['b']
h = sigmoid(k)
# Forward pass of second layer
o = np.matmul(h, params['v']) + params['c']
y = softmax(o)
```

Listing 5: 3D version of the forward pass

```
# Backward pass of second layer
dy_do = cache['y'] - Y
dy_dc = np.copy(dy_do)
do_dv = np.einsum('ij,ik->ijk', cache['h'], dy_do)
```

```
do_dh = np.matmul(dy_do, params['v'].T)
# Backward pass of first layer
dh_dk = do_dh * cache['h'] * (1. - cache['h'])
dk_dw = np.einsum('ij,ik->ijk', X, dh_dk)
dk_db = np.copy(dh_dk)
```

Listing 6: 3D version of the backward pass

The 2D and 3D versions of the forward- and backward pass result in tensor shapes illustrated in tables 2 and 3:

Local derivative	**2D tensor shape**	**3D tensor shape**
k	(300)	(32, 300)
h	(300)	(32, 300)
o	(10)	(32, 10)
y	(10)	(32, 10)

Table 2: Tensor shapes for the 2D and 3D variant of the forward pass

Local derivative	**2D tensor shape**	**3D tensor shape**
$\partial y/\partial o$	(10)	(32, 10)
$\partial y/\partial c$	(10)	(32, 10)
$\partial o/\partial v$	(300, 10)	(32, 300, 10)
$\partial o/\partial h$	(300)	(32, 300)
$\partial h/\partial k$	(300)	(32, 300)
$\partial k/\partial w$	(784, 300)	(32, 784, 300)
$\partial k/\partial b$	(300)	(32, 300)

Table 3: Tensor shapes for the 2D and 3D variant of the backward pass

Question 7

Train the network on MNIST and plot the loss of each batch or instance against the timestep. This is called a learning curve or a loss curve. You can achieve this easily enough with a library like `matplotlib` in a `jupyter` notebook, or you can install a specialized tool like tensorboard. We'll leave that up to you.

Answer In figure 2, the mean loss for every batch is illustrated in a scatter plot after training the MNIST dataset for 5 epochs with a learning rate of 0.01 and a batch size of 32. Here we see that the network converges fast at the start of the training process and slower as the training progresses, which is what a normal learning curve looks like.

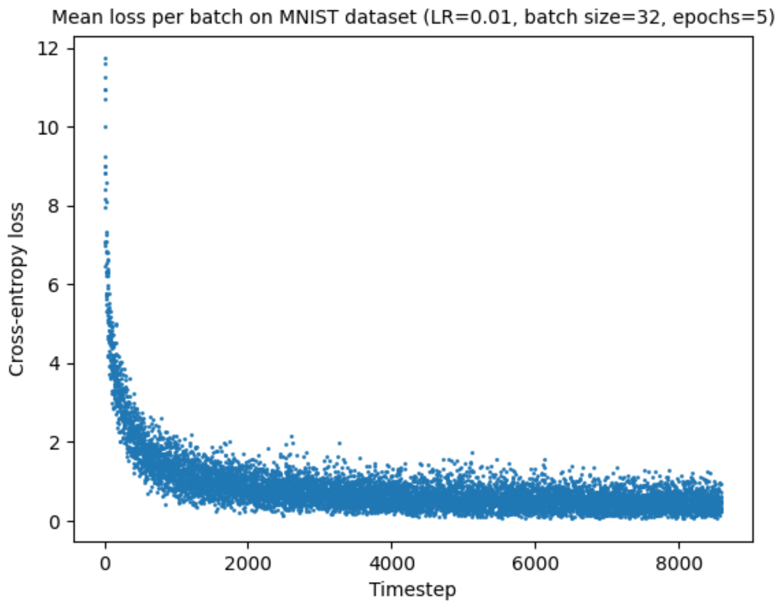
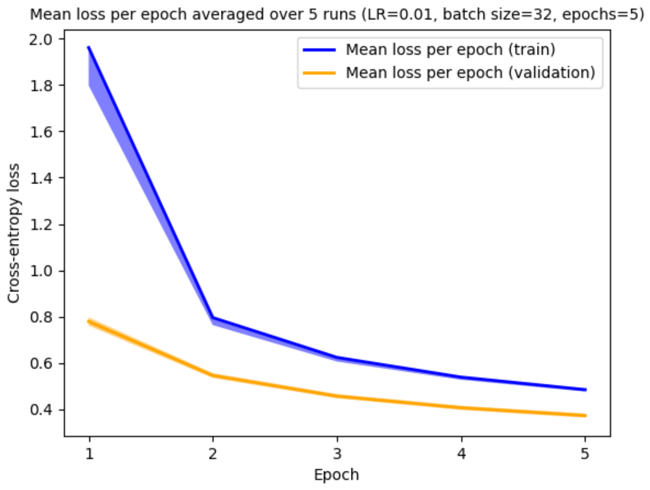
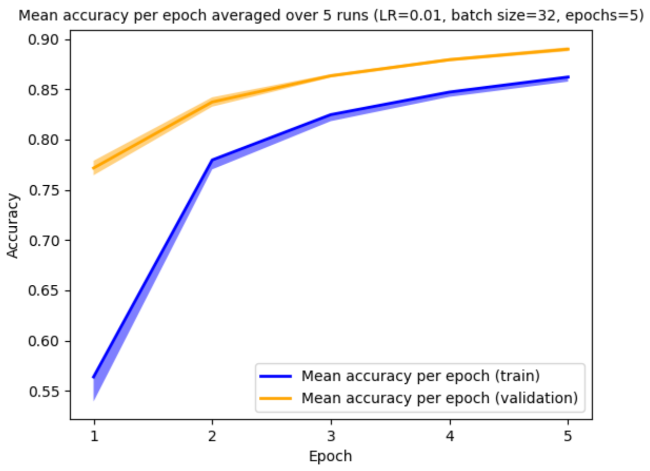


Figure 2: Mean loss per batch (mini-batch Gradient Descent)

Experiments: Figure 3a and 3b display the mean loss and mean accuracy per epoch averaged over 5 consecutive runs respectively:



(a) Mean loss per epoch over 5 runs (mini-batch Gradient Descent)



(b) Mean accuracy per epoch over 5 runs (mini-batch Gradient Descent)

Figure 3: A figure with two subfigures

At first glance it may be called remarkable that the validation loss is lower compared to the training loss and the validation accuracy is higher compared to the training accuracy. However, this can be explained by the fact that the network converges really fast and the validation loss and accuracy are both evaluated after all the training data has been fed into the network during an epoch. The validation set therefore takes advantage of the weights that are already updated during the training part of the epoch. Also, the mean values of the training set are heavily influenced by the fast converging network (especially in the first epoch).

The variance of the plots is rather small, indicating that the results are very consistent. Lastly, the loss of the training set as well as the the validation set seems to decrease during the entire training process, indicating that there's no sign of overfitting. The same applies to the (rising) accuracy.

	Learning rate			
Learning rate	0.001	0.003	0.01	0.03
Mean/std. cross-entropy loss	$0.2524 \pm 5.867 \cdot 10^{-3}$	$0.1871 \pm 6.726 \cdot 10^{-3}$	$0.1398 \pm 6.714 \cdot 10^{-3}$	$0.1287 \pm 6.031 \cdot 10^{-3}$
Mean/std. accuracy	$0.9266 \pm 2.849 \cdot 10^{-3}$	$0.9459 \pm 1.741 \cdot 10^{-3}$	$0.9598 \pm 2.621 \cdot 10^{-3}$	$0.9667 \pm 2.341 \cdot 10^{-3}$

Table 4: Mean/standard deviation of the cross-entropy loss and accuracy on the validation set after 5 epochs using Stochastic Gradient Descent

Looking at the results above, in which we compare different learning rates for a variant of the network using Stochastic Gradient Descent. We again obtain results with a small variance, indicating similar results for every run. Besides, we see that the best results are obtained with a learning rate of 0.03 using Stochastic Gradient Descent (batch size = 1). Therefore, the model is trained again with these hyperparameters. This time, with the full training data and canonical test set. On this canonical test set, a final loss of 0.124 and accuracy of 0.971 is obtained. These results are very similar compared to the ones on the non-final train/validation split, indicating that the model truly classifies the target classes well and the model is not overfitted.

References
Indicate papers/books you used for the assignment. References are unlimited. I suggest to use `bibtex` and add sources to `literature.bib`. An example citation would be Eiben et al. (2003) for the running text or otherwise (Eiben et al., 2003).

References

Eiben, A. E., Smith, J. E., et al. (2003). *Introduction to evolutionary computing*, volume 53. Springer.

Appendix

The TAs may look at what you put here, **but they're not obliged to**. This is a good place for, for instance, extra code snippets, additional plots, hyperparameter details, etc.