Course: Deep Learning **(28-11-2021)**

Submission Assignment #2

Coordinator: Jakub Tomczak *Name:* Tiddo Loos*, Netid:* tls430

Question 1

The gradient of the loss with respect to the output is defined as $\frac{\delta \text{ loss}}{\delta f}$ and the gradient of the output with respect to the input is defined as $\frac{\delta f}{\delta X}$. Given these two gradients the backward of X and the backward of Y are derived in equation 1 and 2 respectively.

$$\frac{\delta \text{ loss}}{\delta X_{ij}} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{ij} * \frac{\delta f}{\delta X}_{ij} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{ij} * \frac{\delta \frac{X_{ij}}{Y_{IJ}}}{\delta X_{ij}} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{ij} * \frac{1}{Y_{ij}} \tag{0.1}$$

$$\frac{\delta loss}{\delta Y_{ij}} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{ij} * \frac{\delta f}{\delta Y}_{ij} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{ij} * \frac{\delta \frac{X_{ij}}{Y_{IJ}}}{\delta Y_{ij}} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{ij} * \frac{X_{ij}}{Y_{ij}^2} \tag{0.2}$$

Question 2

The function f is an element-wise applied function. Therefore, the backward of F is the element-wise application of f' applied to the elements of X, multiplied (element-wise) by the gradient of the loss with respect to the outputs. The derivation shows that the backward of X_i is the gradient of the loss with respect to the output times the derivative of the function f applied on X_i:

$$\nabla X = \frac{\delta loss}{\delta X_i} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_i * \frac{\delta f\left(X_i\right)}{\delta X_i} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_i * f'\left(X_i\right) \tag{0.3}$$

Question 3

X is a n-by-f in put. The neurons in the layers are calculated by $X * W$ (in this example there is no bias), where weight W is an f-by-o matrix (features, outputs). The multiplication of X and W results in a matrix which has an n-by-o shape. First the backward of this operation with respect to W and then with respect to X is derived.

$$\frac{\delta loss}{\delta W_{ij}} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{nj} * \left(\frac{\delta X_n W}{\delta W}\right)_{ij} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{nj} * \frac{\delta \sum_{k=1}^{f} X_{ni} W_{ik}}{\delta W_{ij}} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{nj} * X_{ni}^T \tag{0.4}$$

$$\frac{\delta loss}{\delta X_{ni}} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{nj} * \left(\frac{\delta X W}{\delta X}\right)_{ni} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{nj} * \frac{\delta \sum_{k=1}^{f} X_{ni} W_{kj}}{\delta X_{ni}} = \left(\frac{\delta \text{ loss}}{\delta f}\right)_{nj} * W_{ij}^T \tag{0.5}$$

Question 4

For this exercise we take $f(X) = Y$ where the function returns Y, a matrix with 16 columns that are equal to X. We take V as a vector containing all 1's and X is a vector of size m. The outer product of X and V results in a matrix of 16 columns with the values of X.

$$\frac{\delta \text{ loss}}{\delta X_i} = \sum_j \left(\frac{\delta \text{ loss}}{\delta f}\right)_{ij} * \frac{\delta V_j X_i}{\delta X_i} \tag{0.6}$$

Now we fill in the the 16 columns for j:

$$\sum \left(\frac{\delta \text{ loss}}{\delta f}\right)_{ij} * V_{16\times 1}^T = \sum \left(\frac{\delta \text{ loss}}{\delta f}\right)_{ij} * 1_{16\times 1}^T = \sum \left(\frac{\delta \text{ loss}}{\delta f}\right)_{ij} * 1 \tag{0.7}$$

Question 5

1) *c.value* contains a 2 by 2 matrix of the values of *c*. Taking the sum of *a* and *b* yields the matrix *c*. Both *a* and *b* are 2 by 2 matrices initiated with random values from a standard normal distribution. 2) The command *c.source* refers to the source object, which consists of the operation node (which contains the two inputs *a* and *b*). 3) $c.source.inputs[0].value$ refers to the matrix of *a*. Therefore when running the command *a.value* the same 2 by 2 matrix (and its values) is presented as $c.source.inputs[0].value$. When printing $c.source.inputs[1].value$ it shows the values of matrix *b*. 4) The $self.grad$ variable in the class $TensorNode$ contains a matrix (in this case 2 by 2) which is updated when the loss is computed. At this point all values are set to 0 logically. the $self.grad$ variable can be used for back propagation.

Question 6

1) In the *Op* class the *OpNode* is created giving it a context and and the inputs. But, the object that defines the operation, depends on the operation type. different classes are programmed for different mathematical operations. 2) In the class *Add* the addition occurs with $return\ a + b$ in line 324. 3) The *OpNode* is created in the *Op* class first. Then the outputs are calculated, depending on the operation. After that, the outputs are assigned to the outputs variable in the *OpNode* class with $opnode = outputs$ in line 249. The OpNode is created before the output node is created. Therefore, the $self.outputs$ is set to $None$ when initialising the $OpNode$ class.

Question 7

The call for making the computations is made in line 159:

```
# compute the gradients over the inputs
ginputs_raw = self.op.backward(self.context, *goutputs_raw)
```

The computations for the backward propagation happen in lines 317, 330, 343, and 363 in the operation classes in core.py (Add, Multiply, Sub and MatrixMultiply). In this "staticmethod" the command $return\ np.matmul(go, b.T)$, $np.matmul(a.T, go)$ is responsible for the matrix computation of the backward propagation (line 374). But, as said earlier, the call to for using these operations is made in line 159.

Question 8

Using the derivation from question 4 the expand function can be reviewed. The expand operation class in ops.py calculates the backward with $return\ goutput.sum(axis = dim,\ keepdims = True)$. Looking at the derived backward in question 4, this is the right way to do it as the local derivative of the expand function is 1 and therefore taking the sum over the gradient is sufficient.

Question 9

For this exercise the ReLu and Sigmoid activation function are compared in terms of the the classification accuracy of the network on the MNIST validation set. Note that in the upcoming questions that when the MNIST dataset is used, the data has been normalised by dividing the inputs by 255 (max value present in the dataset). the ReLU function is implemented for the forward and back propagation in *op.py*, *functions.py* and *mlp.py* as follows:

```
#added to ops.py
class Relu(Op):
    @staticmethod
    def forward(context, input):
        relux =  input * (input > 0)
        context['relux'] = relux # store the ReLu of input for the backward pass
        return relux

    @staticmethod
    def backward(context, goutput):
        relux = context['relux'] # retrieve the ReLu of goutput
        return 1.0 * goutput * (relux > 0)
```

```
#added to function.py
import Relu
def relu(x):
    return Relu.do_forward(x)

#added to mlp.py
hidden = vg.relu(hidden)
```

After training of 20 epochs, the training with the Sigmoid and the ReLU activation yield a 94.8% and 93.3% respectively. For both training methods the accuracy on the validation set it calculated at the start of each epoch. The results are presented in Figure 1. What also can be seen in Figure 1 is that the network with the Sigmoid activation function converges to the max accuracy score sooner than the network with the ReLU activation function.

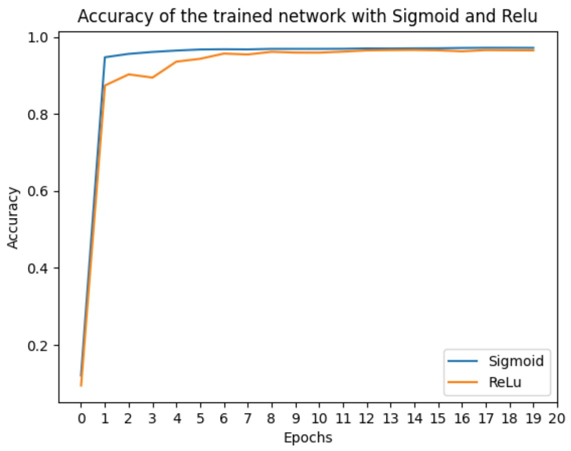

Figure 1: Accuracy on the validation data with ReLU and Sigmoid activation function

Question 10

Hidden Layer Size

For this exercise network structures are investigated. In the first experiment the hidden layer size was researched by varying the hidden layer multiplication parameter. This parameter indicates the size of the multiplication of the input features, resulting in the size of the hidden layer. It was chosen to investigate the hidden layer multiplication parameter with the values 2, 4 and 6. to clarify, using a hidden layer multiplication of 6 means: $features * 6 = hiddenlayer\ size$. The results can be reviewed in Figure 2. In this experiment the Sigmoid was used as activation function. The resulting accuracy's after 20 epochs are 94.2%, 94.0% and 93.6% for hidden_mult is set to 2, 4 and 6 respectively. Setting *hidden_mult* to 2 results in the best performance of the network. However, the results with the network setting $hidden\_mult = 2$ does not differ significantly from the other two samples ($p > 0.05$). When performing the test multiple times, the results for the three values of the hidden multiplayer parameter are close to each other and they outperform each other randomly with little difference between them. Therefore, in the next experiment the *hidden_mult* parameter is kept to its default value of 4.

Hidden Layers

Secondly the amount of layers in the neural network is examined. Note, that for this experiment the hidden_mult parameter was set to 4 according to the results of the previous experiment and the Sigmoid was used as activation function. In Figure 3 the results are shown during training of two neural networks: one neural network with 1 layer and the one neural network with 2 layers. The results show that the 1-layer neural network outperforms the 2-layer neural network in terms of accuracy on the MNIST validation set.

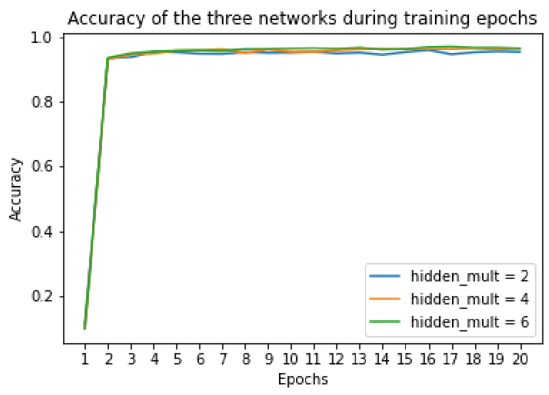

Figure 2: Accuracy during training varying the hidden layer size

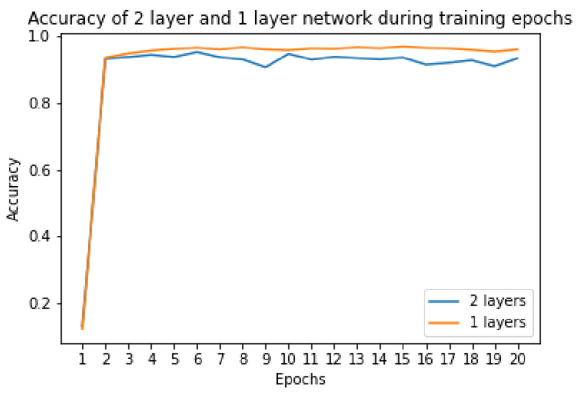

Figure 3: Accuracy during training with 1- and 2-layer network

Question11

After the pytorch neural network has been implemented, with the the batch size to be used in batch gradient descent is investigated. This experiment is conducted with the CIFAR10 dataset. First the batch sizes 100, 200 and 300 were tested. The results showed that the accuracy increased when using smaller batches. Therefore it was decided to also test a batch size of 50 and 25. The results can be reviewed in Figure 4. When setting the batch size 50, the highest accuracy of 63.3% was yielded after 46 epochs. After examining the batch size it was decided to investigate the learning rate. Different learning rates have been tested, starting at 0.0005. The accuracy's on the validation data were calculated in each epoch. In Figure 5 the results are presented and the highest accuracy of 65.1% resulted form a learning rate of 0.003 after 20 epochs.

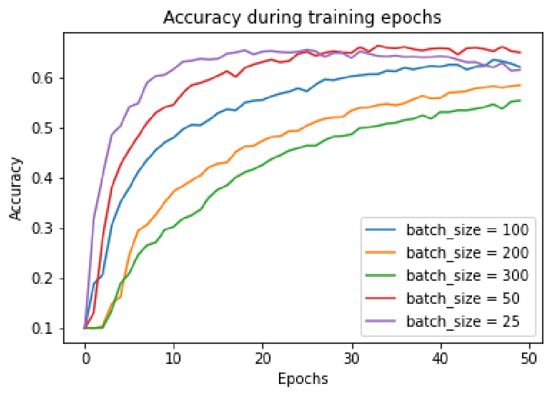

Figure 4: Accuracy during training varying the batch size

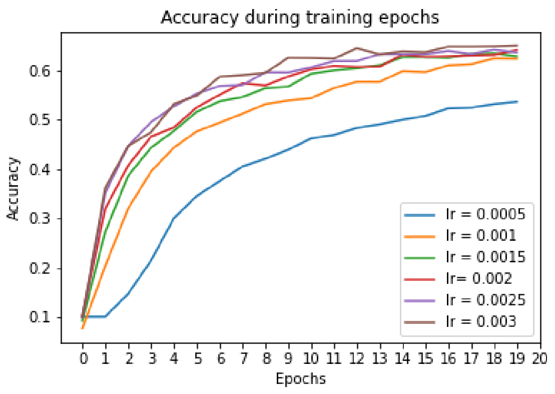

Figure 5: Accuracy during training with different learning rates

Question12

For this exercise is was decided to compare the Adam optimiser with Stochastic Gradient Descent (SGD). The batch size was set to 50, the learning rate to 0.003 and the epochs to 50. The accuracy on the validation set was measured at the start of each epoch. The best accuracy's of Adam optimisation and stochastic gradient descent were 63.2% and 62.7% after 14 and 50 training epochs respectively (Figure 6 on the next page). The Adam optimiser seems to converge earlier to a max accuracy and then it decays slowly. The SGD seems still to improve in the last training epoch. However, to cap the training time, the epochs were set to 50. based on these results the Adam optimiser was used in a next experiment to get a higher accuracy in reasonable training time. The batch size was set to a value of 4 and the Adam optimiser was used. The epochs are set to 50 again to see if there will be a decay in the accuracy after 20 training epochs. The results are presented in Figures 7 and 8. The highest accuracy is reached after 6 epochs resulting in a 62.3% accuracy. What stands out is that the cross entropy loss is dropping but the accuracy on the validation set is not increasing or even decreasing.

This is an indication for over fitting. Therefore, the model should generalise more in stead of overfitting the training data. A last test is carried out with Adam optimiser. The batch size is set to 50, the learning rate to 0.0005 (to delay convergence) and the amount of epochs will be set to 20. The results of this experiment yielded a max accuracy of 64.7% in epoch 17.

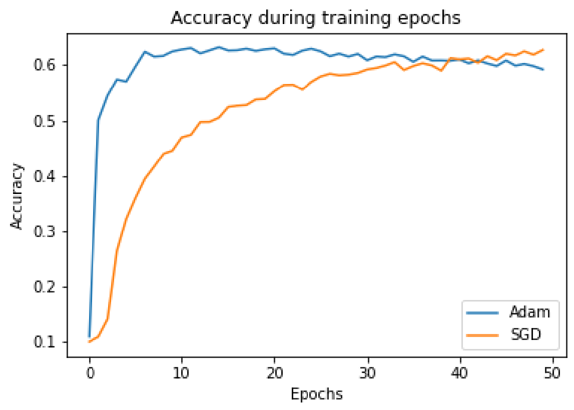

Figure 6: Accuracy on validation set during training with Adam and SGD

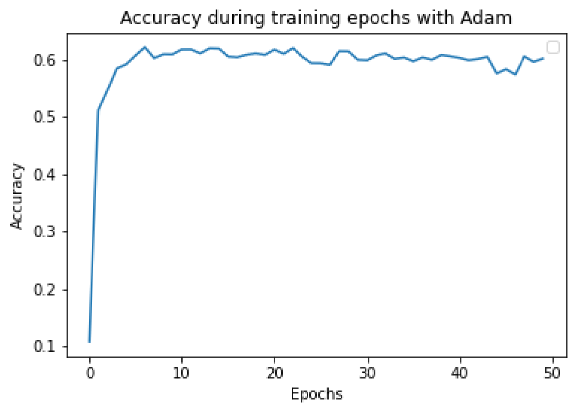

Figure 7: Accuracy during training on the validation set with Adam optimiser

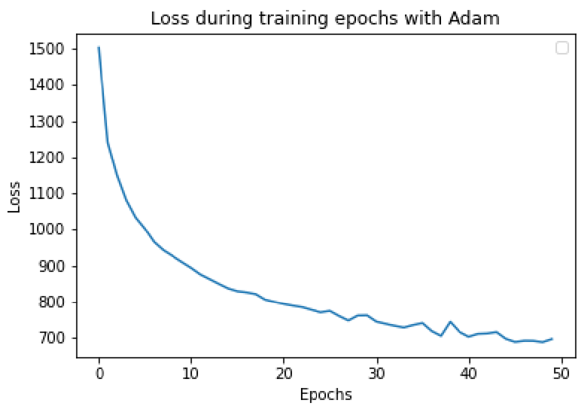

Figure 8: Average CE loss during training epochs with Adam optimiser