Course: Deep Learning *Coordinator:* Jakub Tomczak

Submission Assignment 2

Name: Simone Colombo, *Student ID:* sco550

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

Let f(X, Y) = X / Y for two matrices X and Y (where the division is element-wise). Derive the backward for X and for Y. Show the derivation.

Answer If the matrices X and Y are defined as $X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}$ and $Y = \begin{bmatrix} y_{1,1} & y_{1,2} & y_{1,3} \\ y_{2,1} & y_{2,2} & y_{2,3} \\ y_{3,1} & y_{3,2} & y_{3,3} \end{bmatrix}$,then

$$f(X,Y) = \frac{X}{Y} = \begin{bmatrix} \frac{x_{1,1}}{y_{1,1}} & \frac{x_{1,2}}{y_{1,2}} & \frac{x_{1,3}}{y_{1,3}} \\ \frac{x_{2,1}}{y_{2,1}} & \frac{x_{2,2}}{y_{2,2}} & \frac{x_{2,3}}{y_{2,3}} \\ \frac{x_{3,1}}{y_{3,1}} & \frac{x_{3,2}}{y_{3,2}} & \frac{x_{3,3}}{y_{3,3}} \end{bmatrix} \tag{0.1}$$

The backward function takes in the derivative of $f(X,Y)$, $f(X,Y)^{\nabla}$, as input and gives the derivative of the input of f, X_{∇} and Y_{∇}, as output. Hence, the backward for X would be $f : f(X,Y)^{\nabla} \to X^{\nabla}$, while, the backward for Y would be $f : f(X,Y)^{\nabla} \to Y^{\nabla}$. In more details for X

$$\mathrm{X}^{\nabla}_{i,j} = \frac{\partial l}{\partial X_{i,j}} = \frac{\partial l}{\partial f(X,Y)} \frac{\partial f(X,Y)}{\partial X_{i,j}} = \sum_z \sum_k \frac{\partial l}{\partial f(X,Y)_{z,k}} \frac{\partial f(X,Y)_{z,k}}{\partial X_{i,j}} = \begin{cases} f(X,Y)^{\nabla}_{i,j} \frac{\partial f(X,Y)_{i,j}}{\partial X_{i,j}} & \text{if i = z and j = k} \\ 0 & \text{otherwise} \end{cases} \tag{0.2}$$

meanwhile, for Y,

$$\mathrm{Y}^{\nabla}_{i,j} = \frac{\partial l}{\partial Y_{i,j}} = \frac{\partial l}{\partial f(X,Y)} \frac{\partial f(X,Y)}{\partial Y_{i,j}} = \sum_z \sum_k \frac{\partial l}{\partial f(X,Y)_{z,k}} \frac{\partial f(X,Y)_{z,k}}{\partial Y_{i,j}} = \begin{cases} f(X,Y)^{\nabla}_{i,j} \frac{\partial f(X,Y)_{i,j}}{\partial Y_{i,j}} & \text{if i = z and j = k} \\ 0 & \text{otherwise} \end{cases} \tag{0.3}$$

The final results are obtained by applying the Kronecker Delta notation [1]. In more details, the values of $\sum_z \sum_k \frac{\partial l}{\partial f(X,Y)_{z,k}} \frac{\partial f(X,Y)_{z,k}}{\partial Y_{i,j}}$ are all zeros besides when, $i = z$ and $j = k$. Furthermore, the summation can be removed since it would be equal to sum over a bunch 0s and only one 1 (where the indices coincide). Hence, $X^{\nabla}_{i,j} = f(X,Y)^{\nabla}_{i,j} \frac{1}{Y_{i,j}} = f(X,Y)^{\nabla}_{i,j} (J \oslash Y_{i,j})$, while, $Y^{\nabla}_{i,j} = f(X,Y)^{\nabla}_{i,j} \frac{-X_{i,j}}{Y^2_{i,j}} = f(X,Y)^{\nabla}_{i,j} (-X_{i,j} \oslash Y^2_{i,j})$ (where J is the all-ones matrix and $\oslash$ is the Hadamard division [2] for element-wise division between matrices). In conclusion, by vectorizing the result, the following outcomes are obtained

$$\mathrm{X}^{\nabla} = \begin{pmatrix} \ddots & \vdots & \\ \dots & f(X,Y)^{\nabla}_{i,j}(J \oslash Y^2_{i,j}) & \dots \\ & \vdots & \ddots \end{pmatrix} = f(X,Y)^{\nabla}(J \oslash Y^2)^T \text{ and } Y^{\nabla} = \begin{pmatrix} \ddots & \vdots & \\ \dots & f(X,Y)^{\nabla}_{i,j}(-X_{i,j} \oslash Y^2_{i,j}) & \dots \\ & \vdots & \ddots \end{pmatrix} = f(X,Y)^{\nabla}((-X \oslash Y^2)^T \tag{0.4}$$

Question 2

Let f be a scalar-to-scalar function $f : \mathbb{R} \to \mathbb{R}$. Let F(X) be a tensor-to-tensor function that applies f element-wise (For a concrete example think of the sigmoid function from the lectures). Show that whatever f is, 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.

[1] https://en.wikipedia.org/wiki/Kronecker_delta

[2] https://en.wikipedia.org/wiki/Hadamard_product_(matrices)

Answer The backward of $F(X)$ can be expressed as F: $F(X)^{\nabla} \rightarrow X^{\nabla}$ by the definition of backward function. Thus, X^{∇} can be computed as follows

$$\mathrm{X}_{i,j}^{\nabla} = \frac{\partial l}{\partial X_{i,j}} = \frac{\partial l}{\partial F(X)} \frac{\partial F(X)}{\partial X_{i,j}} = \sum_z \sum_k \frac{\partial l}{\partial F(X)_{z,k}} \frac{\partial F(X)_{z,k}}{\partial X_{i,j}} = \begin{cases} F(X)_{i,j}^{\nabla} \frac{\partial F(X_{i,j})}{\partial X_{i,j}} & \text{if i = z and j = k} \\ 0 & \text{otherwise} \end{cases} \tag{0.5}$$

The final results are obtained by applying the Kronecker Delta notation 1. In more details, the values of $\sum_z \sum_k \frac{\partial l}{\partial F(X)_{z,k}} \frac{\partial F(X)_{z,k}}{\partial X_{i,j}}$ are all zeros besides when, $i = z$ and $j = k$. Furthermore, the summation can be removed since it would be equal to sum over a bunch 0s and only one 1 (where the indices coincide). Note that, $\frac{\partial F(X)_{i,j}}{\partial X_{i,j}} = F(X)'_{i,j}$. Hence, by vectorizing the results **??**

$$X^{\nabla} = \begin{pmatrix} \ddots & \vdots & \\ \dots & F(X)_{i,j}^{\nabla} \frac{\partial F(X_{i,j})}{\partial X_{i,j}} & \dots \\ & \vdots & \ddots \end{pmatrix} = f(X,Y)^{\nabla} F(X)' \tag{0.6}$$

Question 3

Let matrix $\mathbf{W}$ *be the weights of an MLP layer with f input nodes and m output nodes, with no bias and no nonlinearity, and let* $\mathbf{X}$ *be an n-by-f batch of n inputs with f features each. Which matrix operation computes the layer outputs? Work out the backward for this operation, providing gradients for both* $\mathbf{W}$ *and* $\mathbf{X}$*.*

Answer The matrix operation to compute the layer outputs is dot product of $\mathbf{W}\boldsymbol{X}$. Let $\boldsymbol{Z} = \mathbf{W}\boldsymbol{X}$, then, the backward function for X can be expressed as $F : Z^{\nabla} \rightarrow X^{\nabla}$. Hence,

$$X^{\nabla} = \frac{\partial \mathbf{W}\boldsymbol{X}}{\partial \boldsymbol{X}} \tag{0.7}$$

Hence, the derivative of the dot product in 0.7 can be solved by applying the formula 0.8

$$\frac{\delta \boldsymbol{ab}}{\delta x} = \frac{\delta \boldsymbol{a}}{\delta x} \cdot \boldsymbol{b} + \frac{\delta \boldsymbol{b}}{\delta x} \cdot \boldsymbol{a} \tag{0.8}$$

In more details, the gradient for X is

$$\mathrm{X}_{i,j}^{\nabla} = \frac{\partial \mathbf{W}\boldsymbol{X}}{\partial X_{i,j}} = \frac{\partial \mathbf{W}}{\partial X_{i,j}} \boldsymbol{X} + \frac{\partial \boldsymbol{X}}{\partial X_{i,j}} \mathbf{W} = \sum_k \sum_z \frac{\partial W_{k,z}}{X_{i,j}} X_{k,z} + \sum_m \sum_n \frac{\partial X_{m,n}}{\partial X_{i,j}} W_{m,n} = \begin{cases} W_{i,j} & \text{if } n = i \text{ and } m = j \\ 0 & \text{otherwise} \end{cases} \tag{0.9}$$

Meanwhile, the gradient for W is

$$\mathrm{W}_{i,j}^{\nabla} = \frac{\partial \mathbf{W}\boldsymbol{X}}{\partial X_{i,j}} = \frac{\partial \mathbf{W}}{\partial W_{i,j}} \boldsymbol{X} + \frac{\partial \boldsymbol{X}}{\partial W_{i,j}} \mathbf{W} = \sum_k \sum_z \frac{\partial W_{k,z}}{W_{i,j}} X_{k,z} + \sum_m \sum_n \frac{\partial X_{m,n}}{\partial W_{i,j}} W_{m,n} = \begin{cases} X_{i,j} & \text{if } n = i \text{ and } m = j \\ 0 & \text{otherwise} \end{cases} \tag{0.10}$$

Question 4

Let $f(\mathbf{x}) = \mathbf{Y}$ *be a function that takes a vector* $\mathbf{x}$*, and returns the matrix* $\mathbf{Y}$ *consisting of 16 columns that are all equal to* $\mathbf{x}$*. Work out the backward of f. (This may seem like a contrived example, but it's actually an instance of broadcasting).*

Answer The backward function for X can be expressed as $F : \mathbf{x}^{\nabla} \rightarrow \mathbf{Y}^{\nabla}$. In more details, the backward can be further expressed as in Eq. 0.11

$$x_i^{\nabla} = \frac{\partial l}{\partial x_i} = \sum_z^{16} \sum_k \partial Y_{z,k}^{\nabla} \frac{\partial Y_{k,z}}{\partial x_i} = \dots \tag{0.11}$$

Then, the derivative of a matrix with regard of a vector is $C_{k,z,i} = \frac{\partial Y_{k,z}}{\partial x_i} = 1$ if $k = j$, regardless of i, otherwise, it is zero. That is because the non-zero values are positioned in the diagonal. Furthermore, by applying the Kronecker deltas notation Eq. 0.12 is obtained

$$\sum_z^{16} \sum_k \partial Y_{z,k}^{\nabla} \frac{\partial Y_{k,z}}{\partial x_i} * \delta_{k,i} = \begin{cases} \sum_z^{16} \sum_k \partial Y_{z,k}^{\nabla} \frac{\partial Y_{k,z}}{\partial x_i} * 1 \text{ if k = i} \\ \sum_z^{16} \sum_k \partial Y_{z,k}^{\nabla} \frac{\partial Y_{k,z}}{\partial x_i} * 0 \text{ otherwise} \end{cases} \tag{0.12}$$

Therefore, by simplifying the sum over z in 0.12

$$x_i^{\nabla} = \sum_k \partial Y_n^{\nabla} \tag{0.13}$$

Question 5

Open an ipython session or a jupyter notebook in the same directory as the README.md file, and import the library with import vugrad as vg. Also do import numpy as np.

Create a TensorNode with x = vg.TensorNode(np.random.randn(2, 2))

Answer the following questions (in words, tell us what these class members mean, don't just copy/paste their values).

1) What does c.value contain?
2) What does c.source refer to?
3) What does c.source.inputs[0].value refer to?
4) What does a.grad refer to? What is its current value?

Answer

1)`c.value` containts the raw value the node holds.
2)`c.source` contains the exadecimal memory address of the operation node that produced c
3)`c.source.inputs[0].value` refers to the first TensorNode value given to the OpNode (e.g. in this case the value of a)
4)`a.grad` refers to the gradient of the TensorNode a when the loss is computed.

Question 6

You will find the implementation of TensorNode and OpNode in the file vugrad/core.py. Read the code and answer the following questions

1) An OpNode is defined by its inputs, its outputs and the specific operation it represents (i.e. summation, multiplication). What kind of object defines this operation?
2) In the computation graph of question 5, we ultimately added one numpy array to another (albeit wrapped in a lot of other code). In which line of code is the actual addition performed?
3) When an OpNode is created, its inputs are immediately set, together with a reference to the op that is being computed. The pointer to the output node(s) is left None at first. Why is this? In which line is the OpNode connected to the output nodes?

Answer

1) The operation is defined as a subclass of the abstract class Op.
2) The addition is performed at Line 324 in the class `AddOp` 1

Source Code 1: AddOp function

```
317 class Add(Op):
318     """
319     Op for element-wise matrix addition.
320     """
321     @staticmethod
322     def forward(context, a, b):
323         assert a.shape == b.shape, f'Arrays not the same sizes ({a.shape} {b.shape}).'
324         return a + b
```

3) This happens because the operations on the inputs node is not called yet, hence, the outputs are initially unknown, so the OpNode is also not connected to the output nodes. When the forward pass is performed the operation is also computed and the OpNode is connected to the outputs (see Line 249).

Source Code 2: `do_forward` function in class Op

```
240     @classmethod
241     def do_forward(cls, *inputs, **kwargs):
242     ....
243         if not type(outputs_raw) == tuple:
244             outputs_raw = (outputs_raw, )
245 
246         opnode = OpNode(cls, context, inputs)
247 
248         outputs = [TensorNode(value=output, source=opnode) for output in outputs_raw]
249         opnode.outputs = outputs
250     ....
251     return ...
```

Question 7

When we have a complete computation graph, resulting in a TensorNode called loss, containing a single scalar value, we start backpropagation by calling
`loss.backward()`
Ultimately, this leads to the backward() functions of the relevant Ops being called, which do the actual computation. In which line of the code does this happen?

Answer The backward of the Ops is called at line 159 in the `backward()` function. In more details, relevant code is 3

Source Code 3: `badward` function in class OpNode

```
145     def backward(self):
146         """
147         Walk backwards down the graph to compute the gradients.
148         Note that this should be a breadth-first walk.
149         When we get to a particular op, we need to be sure that all its
150         outputs have already been visited.
151         To this end, we only move down to the children of a node once we are sure
152         that it has been visited from all its parents.
153         """
154 
155         # extract the gradients over the outputs (these have been computed already)
156         goutputs_raw = [output.grad for output in self.outputs]
157 
158         # compute the gradients over the inputs
159         ginputs_raw = self.op.backward(self.context, *goutputs_raw)
```

Question 8

core.py contains the three main Ops, with some more provided in ops.py. Choose one of the ops Normalize, Expand, Select, Squeeze or Unsqueeze, and show that the implementation is correct. That is, for the given forward, derive the backward, and show that it matches what is implemented.

Answer The function chosen to verify its implementation is the `Normalize` function, $F(X)$. The backward of $F(X)$ can be expressed as F: $F(X)^{\nabla} \rightarrow X^{\nabla}$ by the definition of backward function. Thus, X^{∇} can be computed as follows 0.14

$$X^{\nabla}_{i,j} = \frac{\partial l}{\partial X_{i,j}} = \frac{\partial l}{\partial F(X)} \frac{\partial F(X)}{\partial X_{i,j}} = \sum_z \sum_k \frac{\partial l}{\partial F(X)_{z,k}} \frac{\partial F(X)_{z,k}}{\partial X_{i,j}} = \begin{cases} F(X)^{\nabla}_{i,j} \frac{\partial F(X_{i,j})}{\partial X_{i,j}} & \text{if i = z and j = k} \\ 0 & \text{otherwise} \end{cases} = \ldots \quad (0.14)$$

The, result in 0.14 are obtained by using the Kronecker deltas notation1. Then, by substiting $F(X)$ with the `Normalize` forward function and by applying the quotient rule, is obtained

$$\ldots = \begin{cases} F(X)^{\nabla}_{i,j} \frac{\partial \frac{X_{i,j}}{\sum_j (X_i)_j})}{\partial X_{i,j}} & \text{if i = z and j = k} \\ 0 & \text{otherwise} \end{cases} = \begin{cases} F(X)^{\nabla}_{i,j} \frac{X'_{ij} \sum_j (X_i)_j - X_{ij} \sum_j (X_i)'_j}{\sum_j (X_i)^2_j} & \text{if i = z and j = k} \\ 0 & \text{otherwise} \end{cases} = \ldots \quad (0.15)$$

Hence, by solving the quotient rule 0.16 is obtained, and

$$= \begin{cases} F(X)^{\nabla}_{i,j} \frac{X'_{ij} \sum_j (X_i)_j - X_{ij} \sum_j (X_i)'_j}{\sum_j (X_i)^2_j} & \text{if i = z and j = k} \\ 0 & \text{otherwise} \end{cases} = \begin{cases} F(X)^{\nabla}_{i,j} \frac{\sum_j (X_i)_j}{\sum_j (X_i)^2_j} - \frac{X_{ij}}{\sum_j (X_i)^2_j} & \text{if i = z and j = k} \\ 0 & \text{otherwise} \end{cases} = \ldots \quad (0.16)$$

by simplifying 0.16, 0.17 is obtained

$$= \begin{cases} F(X)^{\nabla}_{i,j} \frac{1}{\sum_j (X_i)_j} - \frac{X_{ij}}{\sum_j (X_i)^2_j} & \text{if i = z and j = k} \\ 0 & \text{otherwise} \end{cases} = \begin{cases} \frac{F(X)^{\nabla}_{i,j}}{\sum_j (X_i)_j} - \frac{F(X)^{\nabla}_{i,j} X_{ij}}{(\sum_j (X_i)_j)^2} & \text{if i = z and j = k} \\ 0 & \text{otherwise} \end{cases} = \quad (0.17)$$

Hence, it is shown that $X^{\nabla}_{ij} = \frac{F(X)^{\nabla}_{i,j}}{\sum_j (X_i)_j} - \frac{F(X)^{\nabla}_{i,j} X_{ij}}{(\sum_j (X_i)_j)^2}$, which is the same result for the `Normalize` backward function 5.

Source Code 4: Normalize

```
127 class Normalize(Op):
128     """
129     Op that normalizes a matrix along the rows
130     """
131     @staticmethod
132     def forward(context, x):
133 
134         sumd = x.sum(axis=1, keepdims=True)
135 
136         context['x'], context['sumd'] = x, sumd
137 
138         return x / sumd
139 
140     @staticmethod
141     def backward(context, go):
142 
143         x, sumd = context['x'], context['sumd']
144 
145         return (go / sumd) - ((go * x)/(sumd * sumd)).sum(axis=1, keepdims=True)
```

Question 9

The current network uses a Sigmoid nonlinearity on the hidden layer. Create an Op for a ReLU nonlinearity (details in the last part of the lecture). Retrain the network. Compare the validation accuracy of the Sigmoid and the ReLU versions.

Answer The ReLU function is implemented as shown in

Source Code 5: ReLU activation function

```
173 class ReLU(Op):
174     """
175     Op for element-wise application of ReLU function
176     """
177 
178     @staticmethod
179     def forward(context, input):
180 
181         relx = np.maximum(0,input)
182         context['relx'] = relx # store the ReLU of x for the backward pass
183         return relx
184 
185     @staticmethod
186     def backward(context, goutput):
```

```
187         relx = context['relx'] # retrieve the relu of x
188         return goutput * (relx > 0)
```

In Figures 1 and 2 the train loss and validation accuracy for with the sigmoid and the relu activation functions are analyzed over 20 epochs respectively. The batch size is 128 and the learning rate is 0.00001.

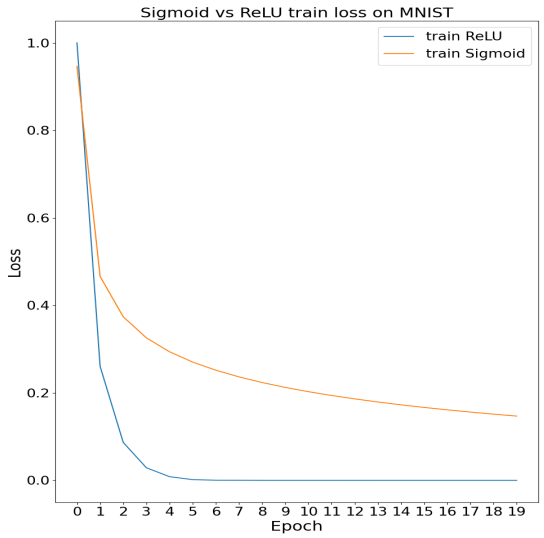

Figure 1: Train loss over epochs Mini-batch GD

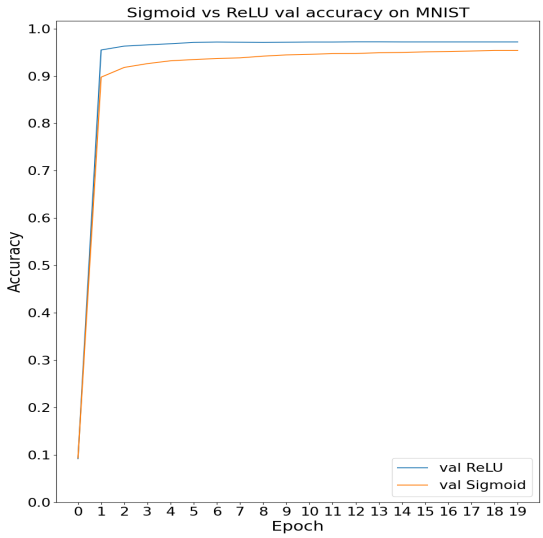

Figure 2: Val. accuracy over epochs Mini-batch GD

It is possible to observe that the relu function converges much faster towards the minimum and also reaches a higher validation accuracy than the sigmoid. That might be because the gradient of the sigmoid becomes increasingly small as the absolute value of x increases, while, the constant gradient of ReLUs results in faster learning.

Question 10

Change the network architecture (and other aspects of the model) and show how the training behavior changes for the MNIST data. Here are some ideas (but feel free to try something else). Try adding more layers to the MLP, or widening the network (more nodes in the hidden layer).
- Add a momentum term to the gradient descent. This is discussed in lecture 4.
- Try adding a residual connection between layers. These are also discussed in lecture 4.
- It is often said that good initialization is the key to neural network performance. What happens if you replace the Glorot initialization (used in the Linear module) by something else? If you initialize to all 0s or sample from a standard normal distribution, do you see a drop in performance?
- If your computer is too slow to do this quickly enough on the MNIST data, you can also work with the synthetic data, but it may be too simple to show the benefit of things like residual connections. In that case, just report what you find.

Answer The effect of the hidden layer sizes, the momentum and the weight initializations are tested on the MNIST dataset separately. The learning rate is set to 0.0001, the sigmoid activation function is used for non-linearity and the network is trained and tested for 20 epochs with batch size 128. Plots 3 and 4 shows that the wider the network the better the generalization of the network. Thus, from 3 it is possible to see that the loss goes down as the hidden layer size is x ($x1$,$x2$, $x4$, $x8$, $x16$ and $x32$) times bigger than the input layer, while, from plot 3 the validation accuracy increases. Interestingly enough, training wider NNs took much more time than for narrower ones for a slight increase in validation accuracy. Hence, the trade-off between validation accuracy and training time is also to be considered.

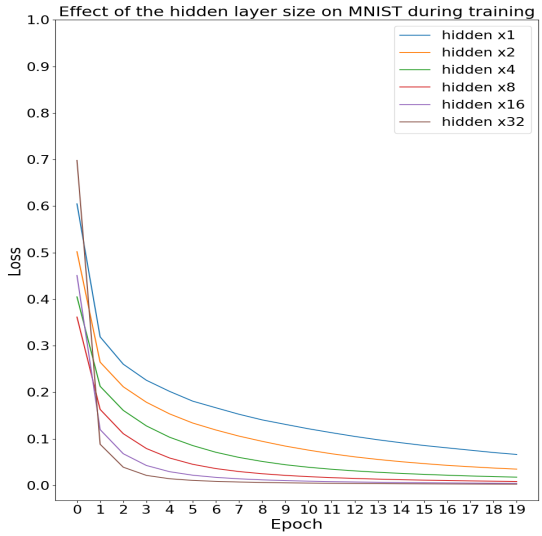

Figure 3: Train loss over epochs Mini-batch GD

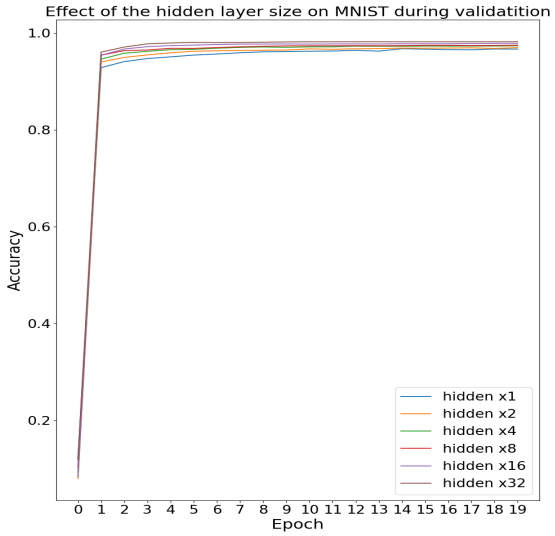

Figure 4: Val. accuracy over epochs Mini-batch GD

Plots 5 and 6 show the effect of the GD with momentum (β) over training and validation respectively. From 5 it is possible to notice that the momentum increases the training loss if smaller than 0.5. On the other side, a bigger momentum than 0.9 leads to increase the loss. From plot 6 it is possible to observe a similar behaviour where the validation accuracy decreases if the momentum is bigger than 0.9, otherwise, it improves. Overall, a momentum greater than 0.9 blows up and always go in the same direction, or jumps around the local minimum. Note that the momentum threshold values are approximations, hence, finer thresholds might be found with more experiments.

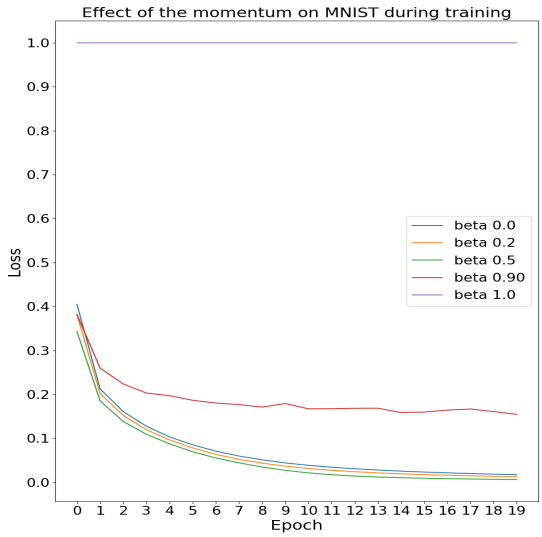

Figure 5: Train loss over epochs Mini-batch GD

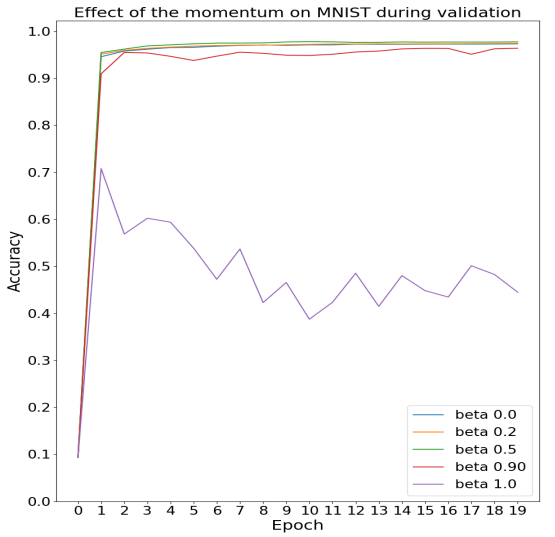

Figure 6: Val. accuracy over epochs Mini-batch GD

Lastly, Plots 7 and 8 show the effect of the weight initialization on the MNIST dataset during training and validation respectively. The Glorot, Zero and Normal weight initializations are compared. For both training and validation, it is possible to notice that the how the weights are initialize really matters for the network performance. Plot 7 shows that the Glorot initialization converges the fastest to the minimum, the Normal initialization converges to the minimum at a slower pace, and the zero initialization does not converge (since the derivatives remain the same).

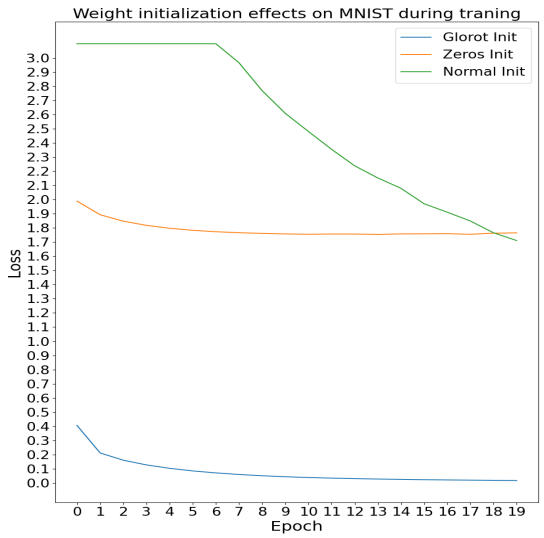

Figure 7: Train loss over epochs Mini-batch GD

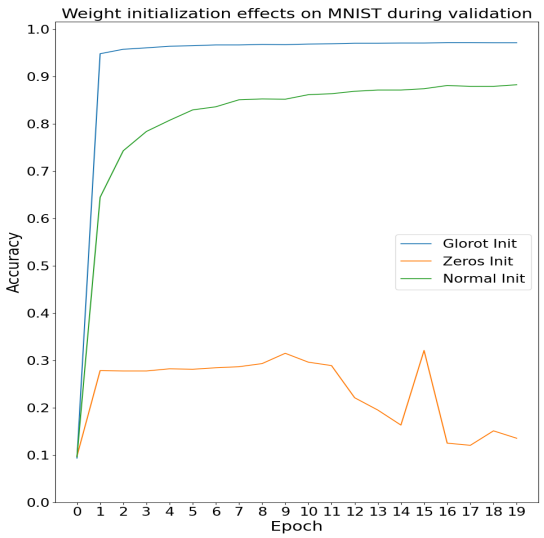

Figure 8: Val. accuracy over epochs Mini-batch GD

Question 11

Install pytorch using the installation instructions on its main page. Follow the Pytorch 60-minute blitz. When you've built a classifier, play around with the hyperparameters (learning rate, batch size, nr of epochs, etc) and see what the best accuracies are that you can achieve. Report your hyperparameters and your results.

Answer The network used for the classification on the MNIST data is a Convolution Neural Network (CNN) with 2 convolutional layers and 1 input channel 6. The input is fed into the network as images of size $[28 \times 28]$

Source Code 6: CNNs for classification on MNIST

```
class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 4 * 4, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

To find the best set of hyperparameters for the CNN the learning rate, the batch size, the number of epochs and the kernel size are tested. Plots 9 and 10 show the effect of the learning rate on training and validation respectively. Learning rate of 0.1, 0.01, 0.001, 0.0001 and 0.00001 are tested. From 9 it is possible to see that as the learning rate decreases the convergence rate decreases as well. Hence, the smaller the learning rate the longer the network takes to converge. On the other side, with a relatively high value for the learning rate (e.g. 0.1 or 0.01) the network seems to converge to the local minimum faster. Similar results are shown in the validation plot 10 where the learning rate of 0.01 gives the highest validation accuracy.

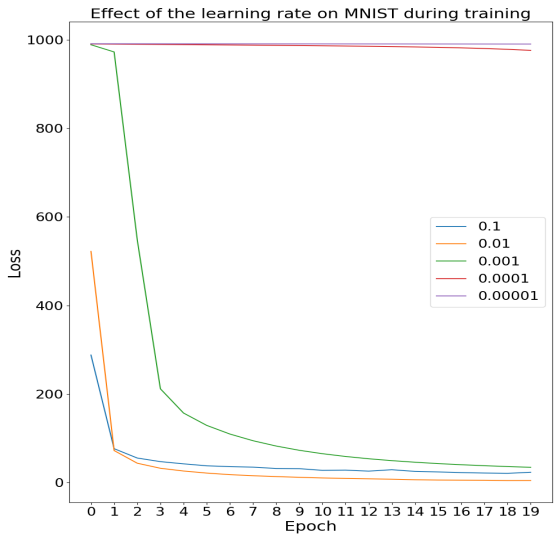

Figure 9: Train loss over epochs Mini-batch GD

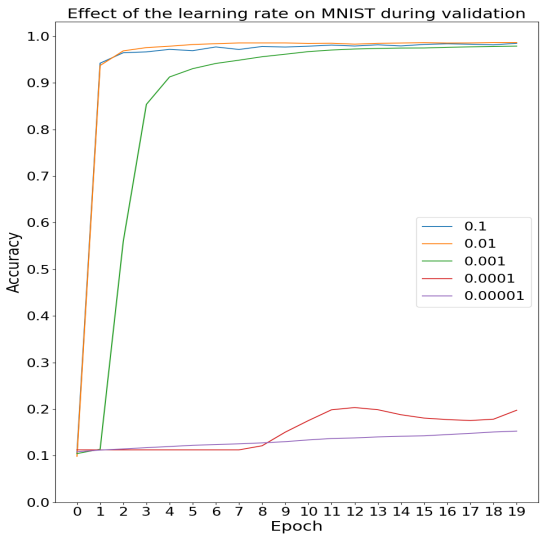

Figure 10: Val. accuracy over epochs Mini-batch GD

Successively, with learning rate set to 0.01 the batch size is tuned. Batch sizes of 16, 32, 64, 128 and 256 are tested. As it is shown in plot 11 the bigger the batch size the faster the network converges to the minimum. Similar behavior is reflected for the validation accuracy 12.

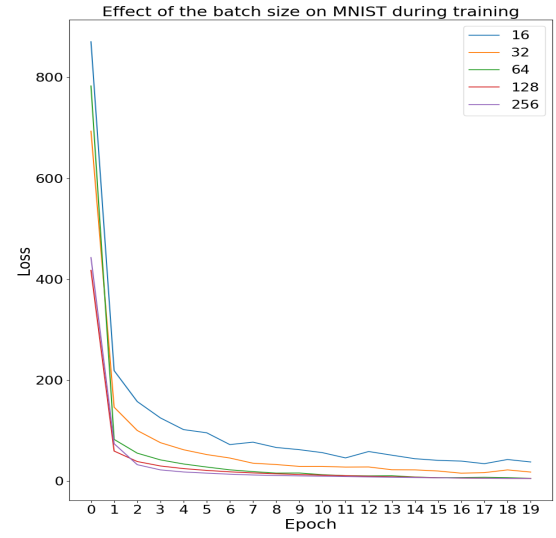

Figure 11: Train loss over epochs Mini-batch GD

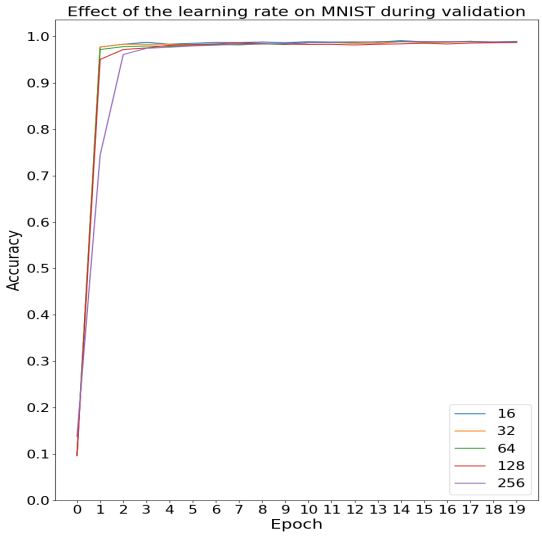

Figure 12: Val. accuracy over epochs Mini-batch GD

Lastly, the effect of the number of epochs is analyzed. The results in table 1 shows that by increasing the number of epochs there is an improvement in both the training loss and the validation accuracy. However, as the number of epochs increases the improvement in training loss and validation accuracy decreases. Hence, it is possible to see a bigger improvement when increasing the epochs from 5 to 30 rather than from 50 to 100 epochs. The final, chosen amount of epochs, based on the aforementioned epoch-improvement tradeoff, is 50.

Epochs	5	10	20	30	50	70	100
Train loss	16.25	10.00	4.62	2.45	0.57	0.06	0.02
Val. accuracy	0.9812	0.9830	0.9834	0.9870	0.9880	0.9884	0.9886

Table 1: Final train loss and val. accuracy with different #epochs

Question 12

Change some other aspects of the training and report the results. For instance, the package torch.optim contains other optimizers than Mini-batch GD which you could try. There are also other loss functions. You could even look into some tricks for improving the network architecture, like batch normalization or residual connections. We haven't discussed these in the lectures yet, but there are plenty of resources available online. Don't worry too much about getting a positive result, just report what you tried and what you found.

Answer As further testing 3 different types of optimizers, SGD, Adam and RMSProp, are compared in Plots 13 and 14 respectively. As it is shown in 13 the optimizer that converges the faster towards the local minimum is Adam, followed by SGD and lastly by RMSProp (which get stuck in a local minimum rather early). Similar, behaviour is observed during validation 14.

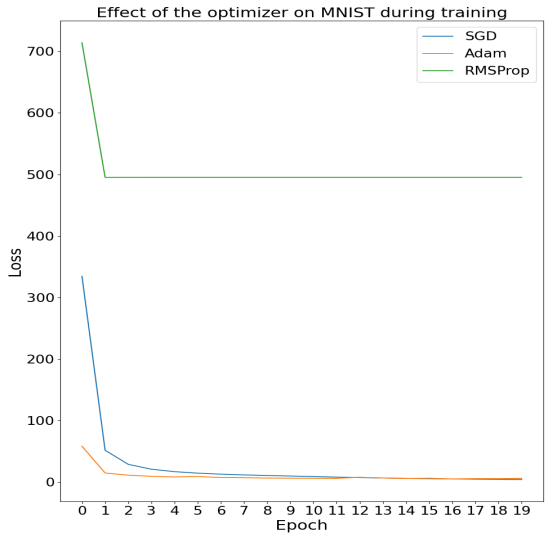

Figure 13: Train loss over epochs Mini-batch GD

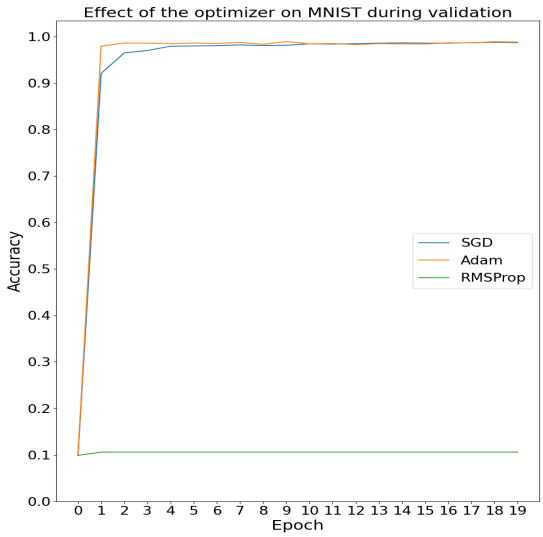

Figure 14: Val. accuracy over epochs Mini-batch GD

Hence, the final model uses the Adam optimizer with learning rate 0.01, batch size 256, Glorot parameter initialization and it is trained over 50 epochs. The final result can be seen in plots 15 and 16 for train loss and test accuracy (train loss = 0.002 and test accuracy= 0.9912 at epoch 50).

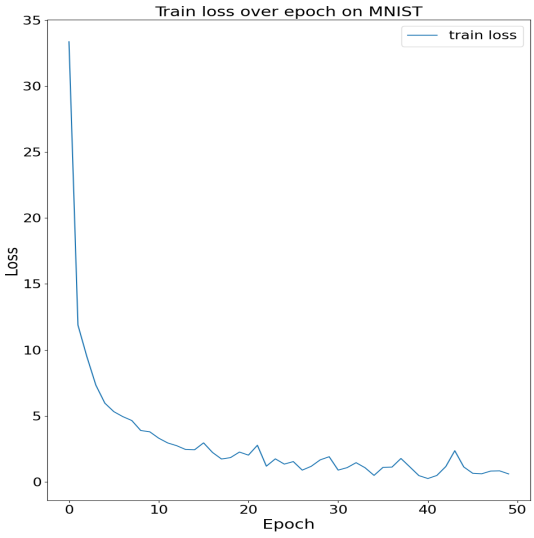

Figure 15: Train loss over epochs Mini-batch GD

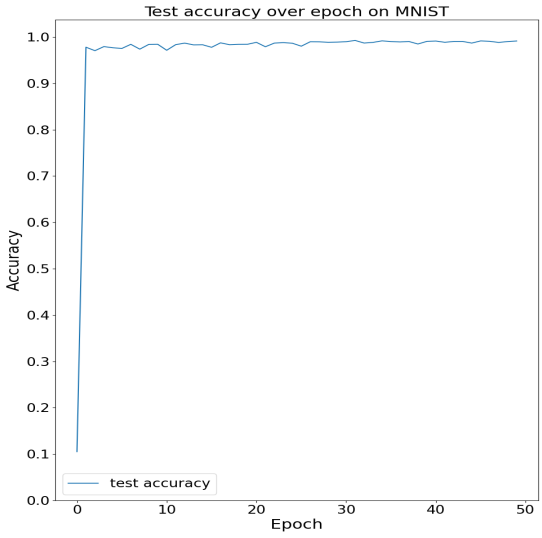

Figure 16: Test accuracy over epochs Mini-batch GD

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.