Course: Deep Learning **(28.11.2021)**

Submission Assignment #2

Coordinator: Jakub Tomczak *Name:* Josip Grguric, *Netid:* jgc800

1 Text based Questions (1, 2, 3, 4, 8)

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.***

First we define what f is.

$$f(X,Y) = \begin{bmatrix} \frac{x_{1,1}}{y_{1,1}} & \dots & \frac{x_{1,n}}{y_{1,n}} \\ \dots & \dots & \dots \\ \frac{x_{m,1}}{y_{m,1}} & \dots & \frac{x_{m,n}}{y_{m,n}} \end{bmatrix}$$

The derivative of X is:

$$\begin{aligned} \frac{df(x_{1,1}, y_{1,1})}{dx_{1,1}} &= \frac{d}{dx_{1,1}}[x_{1,1} \times y_{1,1}^{-1}] \\ &= \frac{1}{y_{1,1}} \\ \frac{df(X,Y)}{dX} &= \begin{bmatrix} \frac{1}{y_{1,1}} & \dots & \frac{1}{y_{1,n}} \\ \dots & \dots & \dots \\ \frac{1}{y_{m,1}} & \dots & \frac{1}{y_{m,n}} \end{bmatrix} \end{aligned}$$

The derivative of Y is:

$$\begin{aligned} \frac{df(x_{1,1}, y_{1,1})}{dy_{1,1}} &= \frac{d}{dy_{1,1}}[\frac{x_{1,1}}{y_{1,1}}] \\ &= \frac{y_{1,1} \times dx_{1,1} - x_{1,1} \times dy_{1,1}}{y_{1,1}^2} \\ \frac{df(X,Y)}{dY} &= \begin{bmatrix} \frac{-x_{1,1}}{y_{1,1}^2} & \dots & \frac{-x_{1,n}}{y_{1,n}^2} \\ \dots & \dots & \dots \\ \frac{-x_{m,1}}{y_{m,1}^2} & \dots & \frac{-x_{m,n}}{y_{m,n}^2} \end{bmatrix} \end{aligned}$$

Question 2 ***Let f be a scalar-to-scalar function*** $f : \mathcal{R} \to \mathcal{R}$. ***Let*** $F(X)$ ***be a tensor-to-tensor function that applies f element-wise. Show what whatever f is, the backward of F is the element-wise application of f' applied to the elements of X, multiplied by the gradient of the loss w.r.t. the outputs.***

From the description of $F(X)$, we can conclude that $F(X) = \{f_1(x_1), \dots, f_n(x_n)\}$. When calculating the derivative of F w.r.t. f_i we get the following:

$$\frac{dF}{df_i} = \frac{d}{df_i}[f_i'(x_i)]$$

Given F(X), an element-wise application of f(x), this gives us the conclusion that $dF(X) = \{f_1'(x_1), \dots, f_n'(x_n)\}$

When implemented in backwards propagation, this is then multiplied by the previous gradients ∇Y, thus giving us the following:

$$\begin{aligned} \nabla Y \times F(X) &= \{\nabla Y \times f_1'(x_1), \dots, \nabla Y \times f_n'(x_n)\} \\ &= \nabla F(X) \end{aligned}$$

Question 3 —

Question 4 ***Let $f(x) = Y$ be a function that takes vector x and returns the matrix Y consisting of 16 columns that are all equal to x. Work out the backward of f***

Let us establish the following: $x = [x_1, \ldots, x_n]$ As well as a vector of ones with the lengh of 16 $\mathbb{1} = [1_1, \ldots, 1_{16}]$.

Based on those assumptions, our $f(x) = Y$ is now equal to $f(x) = x^T \mathbb{1}$

Finally, we work out the backward of f:

$$\frac{df(x)}{dx} = \mathbb{1}$$
$$\frac{df(x)}{d\mathbb{1}} = x^T$$

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.***

The op that I have decided to prove was the Normalize op. The normalization forward and the given backward are:

$$\lambda = \frac{X}{\sum_{j=1}^{n} x_j}$$
$$\nabla\lambda = (\frac{\nabla}{\sum_{j=1}^{n} x_j}) - \sum_{i=1}^{m}(\frac{\nabla \cdot x_i}{(\sum_{j=1}^{n} x_j)^2})$$

The proof for a specific x_1 (knowing that x_j contains x_1):

$$\lambda = \frac{x_1}{\sum_{j=1}^{n} x_j}$$
$$\frac{d\lambda}{dx_j} = \frac{d}{dx_j}[\frac{x_1}{\sum_{j=1}^{n} x_j}]$$
$$= \frac{\sum_{j=1}^{n} x_j \cdot dx_1 - x_1 \cdot d\sum_{j=1}^{n} x_j}{(\sum_{j=1}^{n} x_j)^2}$$
$$= \frac{\sum_{j=1}^{n} x_j - x_1}{(\sum_{j=1}^{n} x_j)^2}$$
$$= \frac{1}{\sum_{j=1}^{n} x_j}(\frac{\sum_{j=1}^{n} x_j}{\sum_{j=1}^{n} x_j} - \frac{x_1}{\sum_{j=1}^{n} x_j})$$
$$= \frac{1}{\sum_{j=1}^{n} x_j} - \frac{x_1}{(\sum_{j=1}^{n} x_j)^2}$$

Given that during normalization, we dont use x_1 but the entirety of X, this is then applied to, and proves the given formula of the normalize op:

$$\nabla\lambda = \nabla \cdot ((\frac{1}{\sum_{j=1}^{n} x_j}) - \sum_{i=1}^{m}(\frac{x_i}{(\sum_{j=1}^{n} x_j)^2}))$$
$$\nabla\lambda = (\frac{\nabla}{\sum_{j=1}^{n} x_j}) - \sum_{i=1}^{m}(\frac{\nabla \cdot x_i}{(\sum_{j=1}^{n} x_j)^2})$$

2 Programming based questions (5, 6, 7, 9, 10, 11, 12)

Question 5 These questions are related to the structure of tensor nodes.

1) What does c.value contain? c.value contains an ndarray of shape (2,2)

2) What does c.source refer to? c.source refers to the operation (i.e. add, subtract, multiply) that created the c node. It will be None if the TensorNode was created manually.

3) What is c.source.inputs[0].value? In short it contains the values of tensor node 'a'. The first part is c.source.inputs, which contains a tuple of Tensor Nodes upon which we have applied a specific operation. In our case, it is an tuple of Tensor nodes [a, b]. When we add [0] to our c.source.inputs, we get the first element of our list which in our case is the tensor node 'a'. Lastly we do '.value' which returns the matrix of the tensor node 'a'.

4) What does a.grad refer to? What is its current value? Grad refers to the calculated gradient over the loss. In the comments of the code it is defined as a "tensor with the same dimensions as the value." Inspecting the variable, it is an numpy ndarray initiated with zeros with the same shape as the values of tensor node 'a', thus the value in our case is ((0,0)(0,0))

Question 6 More questions about tensor nodes + op nodes.

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? It is unclear to me what is meant by this, however during addition in the example of the creation of object 'c' the values of 'a' and 'b' have to be Arrays.

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? Line 324 (figure 1, code snippet 3.1) of core.py, under 'class Add(Op)", where we first make an assert that a and b values have the same shape, then we return their addition. This is then returned to line 243 of core.py under the name *outputs_raw* within class Op.

Figure 1: Addition of two tensor nodes

```
317  class Add(Op):
318      """
319      Op for element-wise matrix addition.
320      """
321      @staticmethod
322      def forward(context, a, b):  context: {}  a: [[ 0.16283072 -0.23330974], [-0.1309418
323          assert a.shape == b.shape, f'Arrays not the same sizes ({a.shape} {b.shape}).'
324          return a + b
325
```

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? The outputs value of an OpNode is a pointer to a Tensor Node object. To create a Tensor Node, we must instantiate it with some value (and optionally, a source and name). Because we have no values to give to our Tensor Node object, we cannot create it, ergo we instantiate OpNode.outputs as None.

The outputs are created on line 248 (figure 2, code snippet 3.2) within the method *do_forward* inside of class Op. The Tensor Node object is instantiated with the previously calculated output value and opnode set as source.

Figure 2: Creation of output tensor node

```
248          outputs = [TensorNode(value=output, source=opnode) for output in outputs_raw]
249          opnode.outputs = outputs
```

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?*** In core.py, class OpNode, under the method backward, on line 159 (figure 3, code snipet 3.3), we call the op class to do our backward calculations for us and then return the values to *ginputs_raw*. In this line, the calculation we do depends on the operation that was

performed during the forward pass. **The actual backward calculations are done in the static method "backward" of the classes: MatrixMultiply, Multiply, Sub, Add, etc.**

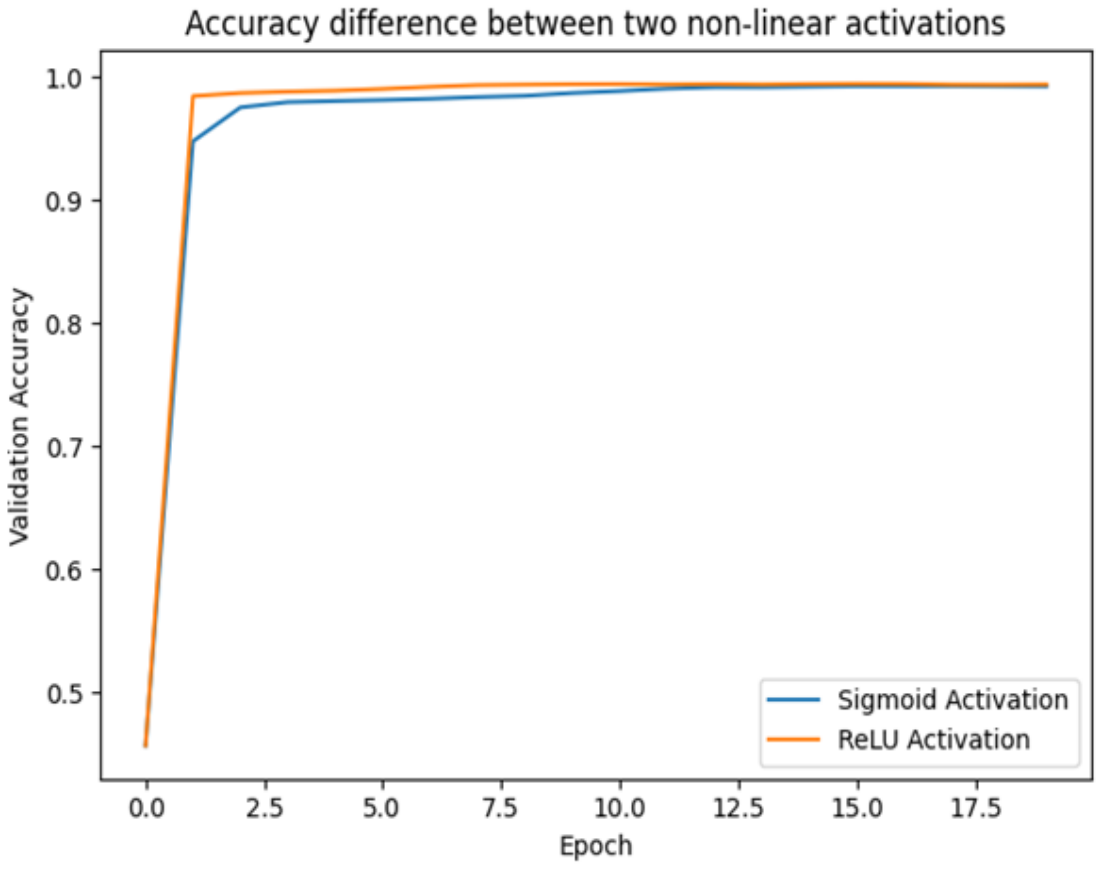
Figure 3: Calculation of gradient

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

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.*** I have implemented the ReLU non linearity, and it can be seen in the code given with this file (it is implemented in a simillar fashion as sigmoid function).

The implementation gave the resulting graphs (figure 4 and figure 5):

Figure 4: Synth dataset



As conclusion, there does not seem to be a significant difference between the Sigmoid and ReLU activations for the two given datasets. However, it should be noted that the comparisons might not accurate due to the difference in default learning rates as well as the two different datasets.

Question 10 ***Change the network architecture (and other aspects of the model) and show how the training behavior changes.***

For this experiment, I have opted to focus on the impact of hidden layer size on accuracy as well as the impact of learning rate on aforementioned hidden layers. For this experiment I've set the dataset to Synth and ran two experiments, with 3 different layer sizes (4 times input size, 2 times input size, and 0.9 times input size) and two different learning rates (0.01, 0.0001). This has given the following results (figure 6 and figure 7):

Figure 5: MNIST dataset

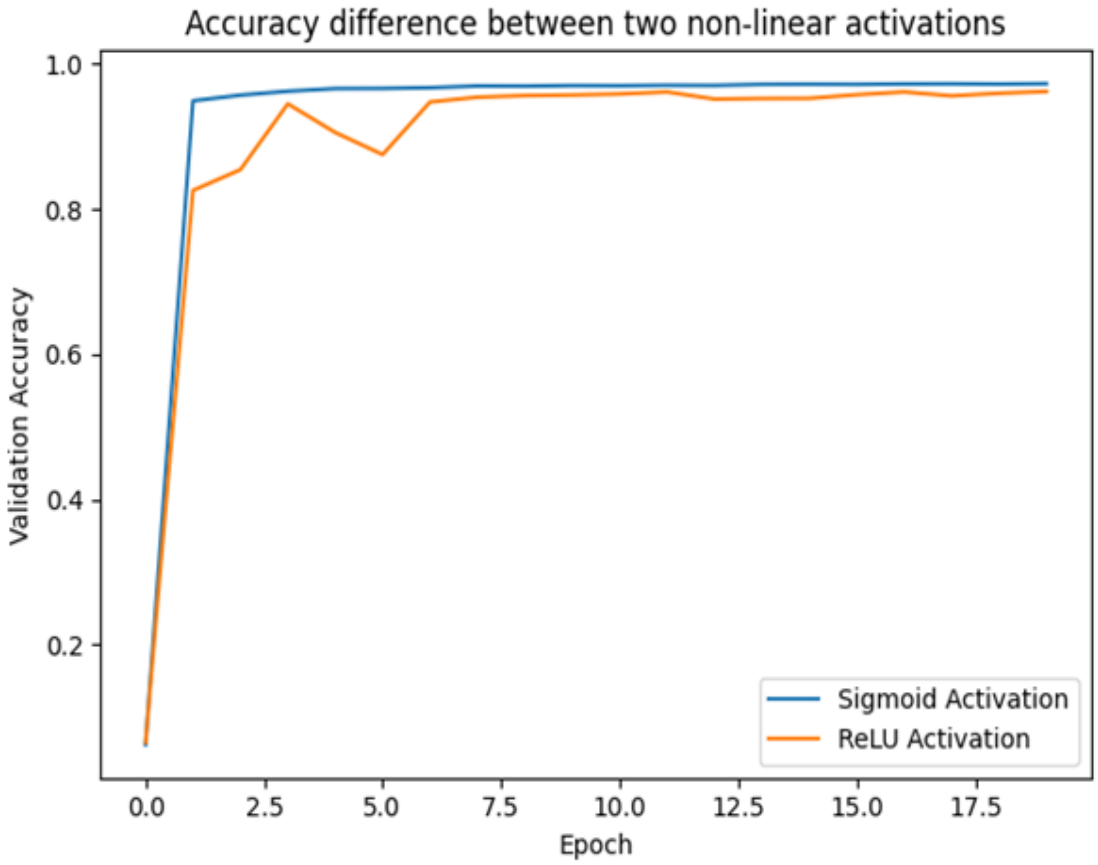


Figure 6: Learning rate 0.01

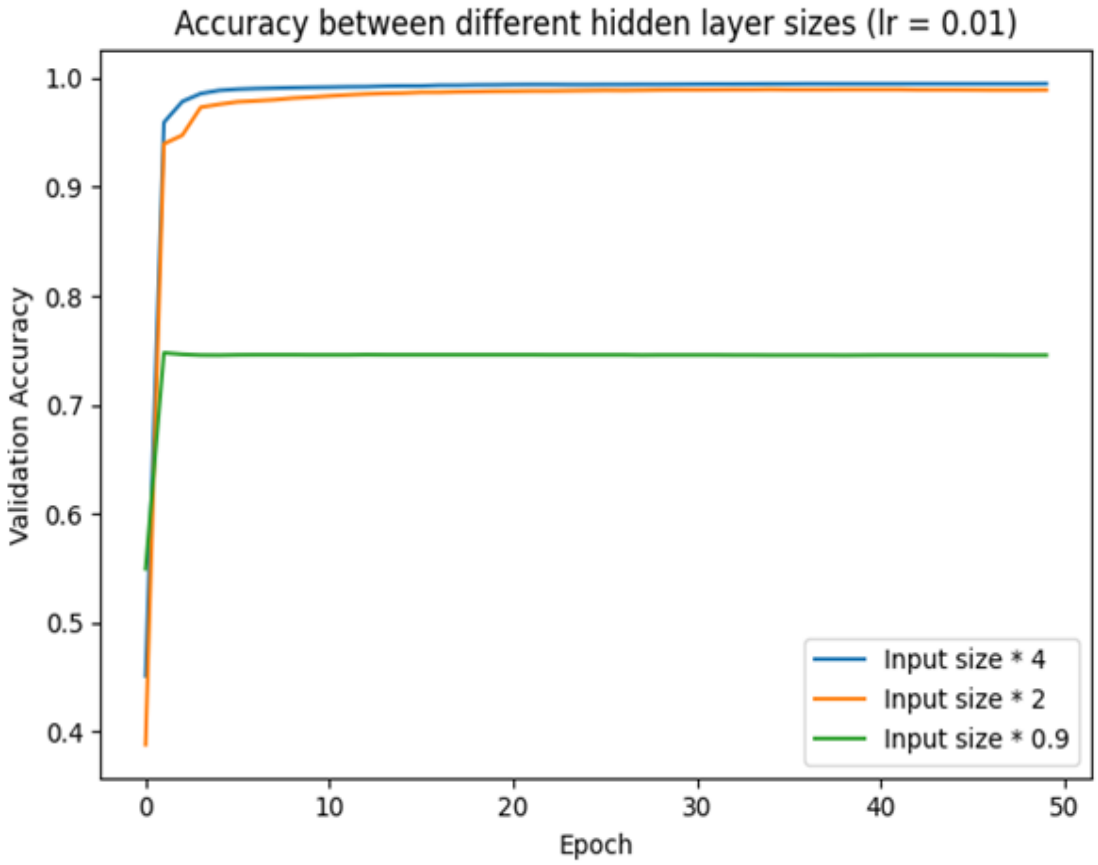
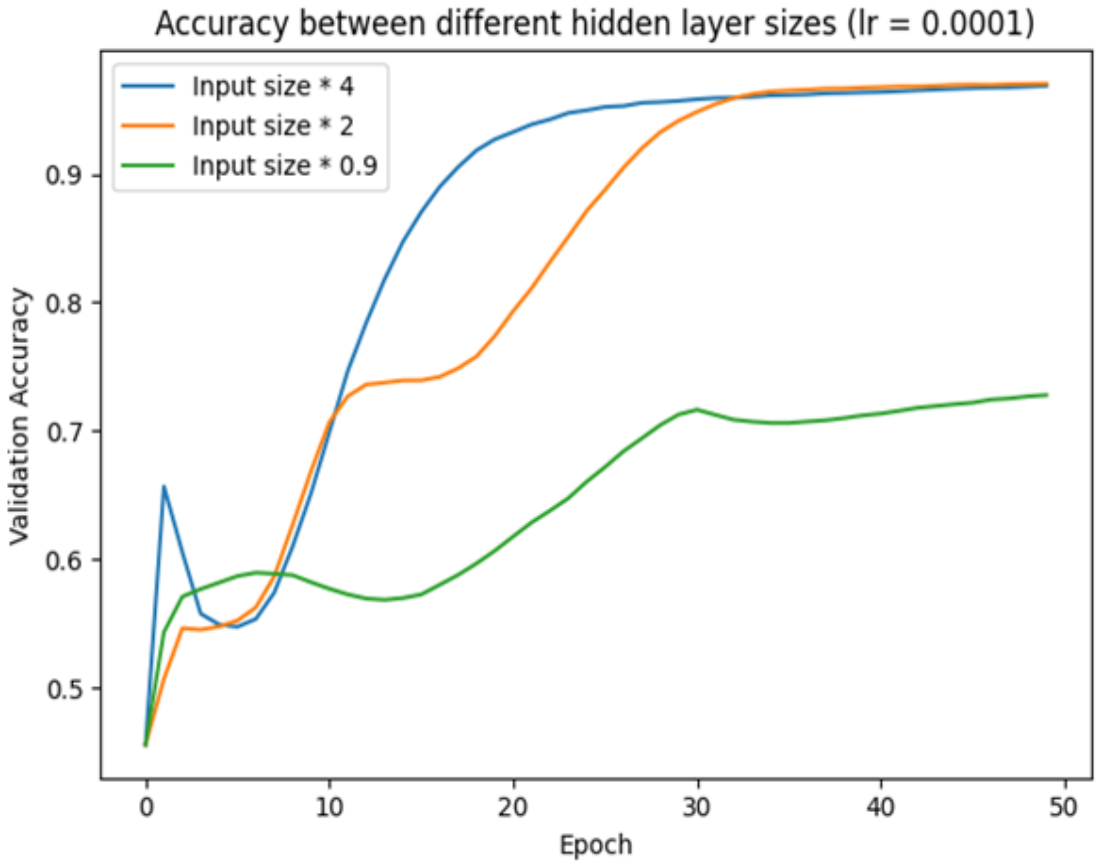


Figure 7: Learning rate 0.0001



Given the figures, we can conclude that the learning rate did impact initial accuracy, but eventually two of them they have reached the global optimum. The interesting result here is with hidden layer size smaller than the input layer. In both cases, this has resulted in an eventual stagnation. Meaning that the hidden layer was unable to leave a local optima.

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***

For both question 11 and 12 I have opted to create a table with 4 different learning rates and 4 different epochs.

For question 11, the maximum accuracy that I have managed to achieve was 63.24% (figure 8).

Figure 8: Learning rate / Epoch Table for Q11

	Epochs				
Learning Rate		**2**	**5**	**10**	**20**
	0.01	29.42	28.28	10.0	24.26
	0.001	55.58	61.91	62.54	59.13
	0.0001	27.65	47.61	57.57	63.24
	0.00001	10.57	13.53	27.22	37.78

Question 12 ***Change some other aspects of the training and report the results. For instance, the package torch.optim contains other optimizers than SGD 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***

For question 12, just as 11 I have created a table (figure 9). I have opted to change 3 things. We have switched from SGD to ASGD gradient descent, we have also added a batch normalization layer as well as a 4th fully connected layer at the end.

Even though we did manage to increase our maximum accuracy to 63.91%, it is not a massive difference and there would need to be a lot more tests to see what works and what doesn't.

Figure 9: Learning rate / Epoch Table for Q11

	Epochs				
Learning Rate		**2**	**5**	**10**	**20**
	0.01	52.63	62.10	61.45	61.72
	0.001	36.66	52.76	61.54	63.91
	0.0001	10.05	14.92	26.54	45.42
	0.00001	10.02	9.53	9.94	10.70

3 Code snippets

3.1 Addition operation

Line of interest is $returna + b$

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

    @staticmethod
    def backward(context, go):
        return go, go
```

3.2 Output Node Definition

Line of interest is $outputs = [TensorNode(value = output, source = opnode) for output in outputs\_raw]$

```
    @classmethod
    def do_forward(cls, *inputs, **kwargs):
        context = {}

        assert all([type(input) == TensorNode for input in inputs]), 'All inputs ...'

        for input in inputs:
            input.numparents += 1

        inputs_raw = [input.value for input in inputs]

        outputs_raw = cls.forward(context, *inputs_raw, **kwargs)

        if not type(outputs_raw) == tuple:
            outputs_raw = (outputs_raw, )

        opnode = OpNode(cls, context, inputs)

        outputs = [TensorNode(value=output, source=opnode) for output in outputs_raw]
        opnode.outputs = outputs

        if len(outputs) == 1:
            return outputs[0]
        return outputs
```

3.3 Gradient calculation

Line of interest: $ginputs\_raw = self.op.backward(self.context, *goutputs\_raw)$

```
    def backward(self):
        goutputs_raw = [output.grad for output in self.outputs]

        ginputs_raw = self.op.backward(self.context, *goutputs_raw)

        if not type(ginputs_raw) == tuple:
            ginputs_raw = (ginputs_raw,)
```

```
        for node, grad in zip(self.inputs, ginputs_raw):

            assert node.grad.shape == grad.shape, f'node shape is {node.size()} but grad shape is {grad

            node.grad += grad

        self.visits += 1

        if self.visits == len(self.outputs):
            for node in self.inputs:
                node.backward(start=False)
```