Course: Deep Learning *Coordinator:* Jakub Tomczak

Submission Assignment 2

Name: Kleio Fragkedaki, *Student ID:* 2729842

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 The backward for X is calculated as follows:

$$X_{ij}^{\nabla} = \frac{dLoss}{dX_{ij}} = \sum_{kl} \frac{dLoss}{df_{kl}} \frac{df_{kl}}{dX_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{d(\frac{X}{Y})_{kl}}{dX_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{d(\frac{X_{kl}}{Y_{kl}})}{dX_{ij}} = f_{ij}^{\nabla} \frac{1}{Y_{ij}}$$

Keep in mind that the division of the function f(X, Y) is element-wise, thus $(\frac{X}{Y})_{kl} = \frac{X_{kl}}{Y_{kl}}$ and that all elements will be zero apart from when $k = i$ and $l = j$.
Therefore, we end up with $X^{\nabla} = f^{\nabla} \frac{1}{Y}$, which is calculated element-wise.

On the other hand, the backward for Y is calculated as follows:

$$Y_{ij}^{\nabla} = \frac{dLoss}{dY_{ij}} = \sum_{kl} \frac{dLoss}{df_{kl}} \frac{df_{kl}}{dY_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{d(\frac{X}{Y})_{kl}}{dY_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{d(\frac{X_{kl}}{Y_{kl}})}{dY_{ij}} = f_{ij}^{\nabla}(-\frac{X_{ij}}{Y_{ij}^2}) = -f_{ij}^{\nabla} \frac{X_{ij}}{Y_{ij}^2}$$

A similar concept holds in this case as well. We end up with $X^{\nabla} = -f^{\nabla} \frac{X}{Y^2}$, which is calculated element-wise.

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.

Answer If $F(X) = f(Xijk)$, then the backward of F can be calculated as follows:

$$X_{ijk}^{\nabla} = \frac{dLoss}{dX_{ijk}} = \sum_{abc} \frac{dLoss}{dF_{abc}} \frac{dF_{abc}}{dX_{ijk}} = \sum_{abc} F_{abc}^{\nabla} \frac{d(f(X_{abc}))}{dX_{ijk}} = F_{ijk}^{\nabla} f'(X_{ijk})$$

Thus, 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. As such, $X^{\nabla} = F^{\nabla} \otimes f'$

Question 3

Let matrix **W** *be the weights of an MLP layer with f input nodes and m output nodes, with no bias and no nonlinearity, and let* **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* **W** *and* **X**.

Answer Let **W** be an m-by-f matrix, **X** an f-by-n matrix and **K** the output of the network, where the matrix operation is $K = WX$.

The backward for this operation wrt. **W** is:

$$W_{ij}^{\nabla} = \frac{dLoss}{dW_{ij}} = \sum_{m,n} \frac{dLoss}{dK_{mn}} \frac{dK_{mn}}{dW_{ij}} = \sum_{m,n,f} K_{mn}^{\nabla} \frac{d(W_{mf} X_{fn})}{dW_{ij}} = \sum_{n} (K_{in}^{\nabla} X_{jn})$$

Note that all the elements will be zero apart from when $m = i$ and $f = j$. To jump to the matrix multiplication, the above equation is translated to the following $W^{\nabla} = K^{\nabla} X^T$.

The backward for this operation wrt. $\mathbf{X}$ is:

$$X_{ij}^{\nabla} = \frac{dLoss}{dX_{ij}} = \sum_{m,n} \frac{dLoss}{dK_{mn}} \frac{dK_{mn}}{dX_{ij}} = \sum_{m,n,f} K_{mn}^{\nabla} \frac{d(W_{mf} X_{fn})}{dX_{ij}} = \sum_{m} (K_{mj}^{\nabla} W_{mi})$$

Note that all the elements will be zero, apart from when $n = j$ and $f = i$. To jump to the matrix multiplication, the above equation is translated to the following $X^{\nabla} = W^T K^{\nabla}$.

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 of f, where $f(x) = \{y_{mn}\}$ and $y_{mn} = x_m$, can be calculated as follows.

$$x_i^{\nabla} = \frac{dLoss}{dx_i} = \sum_{m,n} \frac{dLoss}{dY_{mn}} \frac{dY_{mn}}{dx_i} = \sum_{m,n} \frac{dLoss}{dY_{mn}} \frac{dx_m}{dx_i} = \sum_{m,n} Y_{mn}^{\nabla} \frac{dx_m}{dx_i} = \sum_{n=0}^{15} Y_{in}^{\nabla}$$

Thus, $x^{\nabla} = \sum_{n=0}^{15} Y_n^{\nabla}$

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 is a matrix (of the same shape as the inputs) that contains the values of the element-wise summation of tensor nodes a and b.
2. c.source refers to the location of the operation node that produced c.
3. c.source.inputs[0].value refers to the value of the first input node of the operation that produced c. In our case, this equals to a.value.
4. a.grad refers to the derivative of the loss with respect to a, and its current value is a zero matrix of the same shape as a.

```
a = vg.TensorNode(np.random.randn(2, 2))
b = vg.TensorNode(np.random.randn(2, 2))
c = a + b
```

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. An *OpNode* is a particular instance of an operation applied to some inputs. The operation is defined by the class *Op*.

2. In line 324 of code, the actual addition is performed. In more detail, the summation is defined in class *Add*, while the relevant code is performed in *forward* method.

```
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. The output node(s) is set to None at first, since the value of the output is calculated based on the operation and the input TensorNodes. The value of the output is computed inside the *forward* method of the particular operation, which gives the flexibility to reuse the *OpNode* with different operations.

   In line 249 of code, the *OpNode* is connected to the output node(s). In particular, this connection is done inside the *do_forward* method of the class *Op*.

```
237         # extract the raw input values
238         inputs_raw = [input.value for input in inputs]
239 
240         # compute the raw output values
241         outputs_raw = cls.forward(context, *inputs_raw, **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
```

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 In line 97 of code, the backpropagation of loss starts, and this leads to the backward function of the relevant operation indicated in line 159. In this line, the actual computation is done.

```
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)
160 
```

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 Ops *Normalize* is the operation which normalizes a matrix along the rows. In forward function, a matrix x is passed, and after summing along the rows (axis $= 1$) and keeping the dimensions, the normalization of x is performed by simply dividing the summed dimensions.

As regards the backward, it is computed as follows, using the derivation rule of $(\frac{f}{g})' = \frac{f'*g - f*g'}{g^2}$.

$$x_{ij}^{\nabla} = \frac{dLoss}{dx_{ij}} = \sum_{kl} \frac{dLoss}{dY_{kl}} \frac{dY_{kl}}{dx_{ij}} = \sum_{kl} \frac{dLoss}{dY_{kl}} \frac{d(\frac{x_{kl}}{\sum_l x_{il}})}{dx_{ij}} = \sum_{kl} Y_{kl}^{\nabla} \frac{d(\frac{x_{kl}}{\sum_l x_{il}})}{dx_{ij}}$$

From the previous equation, we obtain that $k = i$ and $l = j$ when deriving over x_{ij}.

$$\sum_{kl} Y_{kl}^{\nabla} \frac{((x_{kl})' * \sum_l x_{il} - x_{kl} * (\sum_l x_{il})')}{(\sum_l x_{il})^2} = (\frac{Y_{ij}^{\nabla} * 1}{\sum_l x_{il}} - \sum_l \frac{Y_{il}^{\nabla} * x_{il}}{(\sum_l x_{il})^2})$$

Given that the gradient of the loss wrt to the outputs (Y^{∇}) is the parameter *go* and the parameter *sumd* is the sum over rows of x, the equation equals to $x^{\nabla} = (\frac{go}{sumd} - \sum_l \frac{go*x}{(sumd)^2})$, where $\sum_l$ denotes the $.sum(axis = 1)$ - the sum over rows.

```
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 In Figure 1, the validation accuracy is displayed for two different versions of the network, one with two different activation functions. One of those versions (blue line) refers to the Sigmoid activation function, while the other (red line) to the ReLU activation function. Based on the plot, Sigmoid activation seems to overcome the relu one, but their difference is not significant. This is probably because of the weight initialization used, since Xavier (Glorot) initialization is not quite as optimal for ReLU functions. The relevant code for the ReLU implementation can be found in Appendix A.

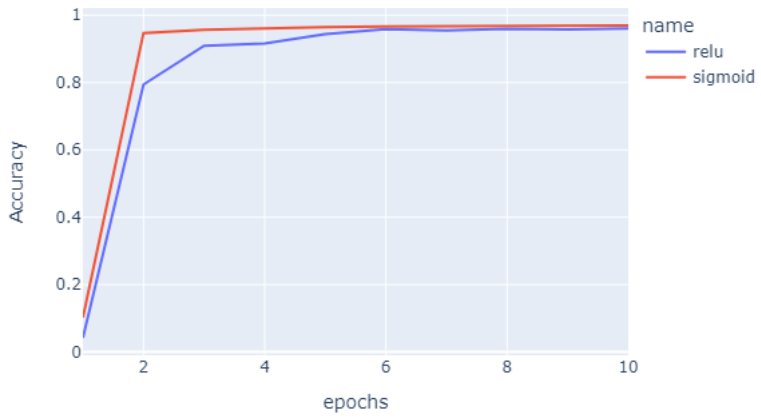

Figure 1: Validation accuracy for two versions of MLP with learning rate 0.0001.

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).

Answer In this section, the network architecture was changed in two different ways and the training behavior is reported.

Widening the network by adding more nodes in the hidden layer To widen the network, I added more nodes in the hidden layer by changing the multiplier parameter, *hidden_mult*. The parameter indicates how many times bigger the hidden layer is than the input layer, and four different values were tested, as can be seen in Figure 2. The left image depicts that the loss is lower when the hidden layer is wider, while there is not an observable difference when it comes to the validation accuracy (plot on the right).

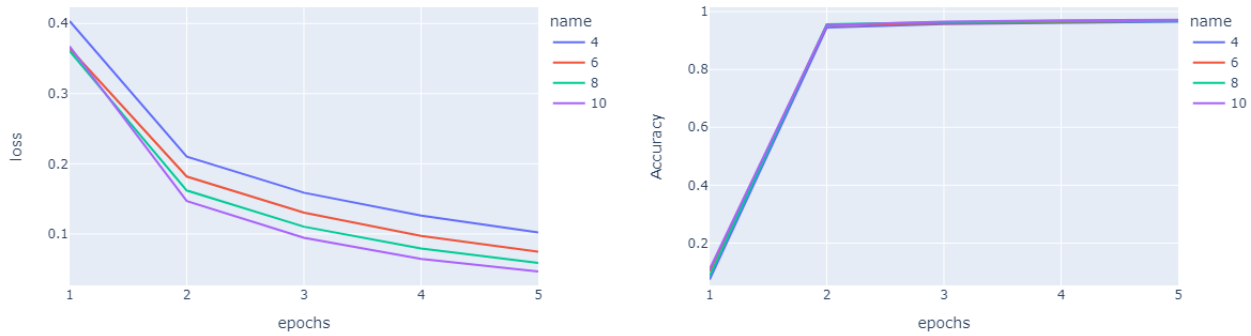

Figure 2: Training loss and Validation accuracy for different sizes of hidden nodes (with a learning rate of 0.0001, batch size 128, and Sigmoid activation function). The number indicates how many times the hidden nodes are bigger than the input layer.

Weight Initialization Different initialization methods were analyzed as well. In Figure 3, the training loss and validation accuracy are reported for five different initialization methods [1], namely the Normal Xavier (Glorot), the Uniform Xavier (Glorot), the Normal He, the Uniform He, and zero initialization of the weights. The bias has been initialized with zero in all cases. Based on plots, the training loss is lowest when initializing the weights either with zero or using the Normal Xavier (Glorot) initialization, while Uniform Xavier (Glorot) follows. The Normal He and the Uniform do not seem to be a good choice along with Sigmoid activation function. Nevertheless, when it comes to accuracy, the results do not seem to differ significantly. The code related to this change can be found in Appendix B.

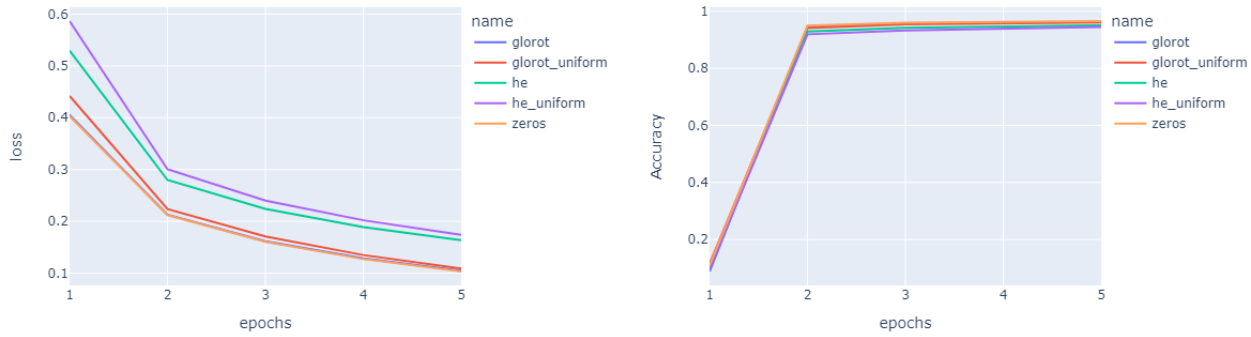

Figure 3: Training loss and Validation accuracy for different initialization of weights (with a learning rate of 0.0001, batch size 128, and Sigmoid activation function).

[1] https://prateekvishnu.medium.com/xavier-and-he-normal-he-et-al-initialization-8e3d7a087528

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 In this section, two different hyperparameters are analyzed, learning rate and batch size.

Learning Rate The hyperparameter learning rate was tested by using different values, namely a value of 0.1, 0.01, 0.001, and 0.0001. As shown in Figure 4, the worst training accuracy and loss is when the learning rate equals 0.1, while the highest accuracy and the lowest loss is when the learning rate is 0.001. Taking into account the test accuracy shown in Table 1, we conclude to the same outcome that the learning rate of 0.001 yields the best results.

Figure 4: Training loss and accuracy for different learning rates. The network was trained with batch size 4, momentum 0.9, and for 5 epochs.

Table 1: The accuracy of the network on the 10000 test images after 5 epochs. LR stands for learning rate.

Accuracy	**LR 0.1**	**LR 0.01**	**LR 0.001**	**LR 0.0001**
Class of plane	0.0 %	1.4 %	52.6 %	57.1 %
Class of car	0.0 %	15.7 %	78.5 %	66.5 %
Class of bird	100.0 %	1.0 %	39.2 %	33.3 %
Class of cat	0.0 %	28.5 %	40.6 %	29.1 %
Class of deer	0.0 %	0.4 %	45.0 %	39.4 %
Class of dog	0.0 %	14.7 %	58.0 %	43.7 %
Class of frog	0.0 %	53.2 %	73.2 %	57.2 %
Class of horse	0.0 %	45.8 %	60.5 %	66.6 %
Class of ship	0.0 %	68.4 %	84.0 %	53.2 %
Class of truck	0.0 %	20.9 %	59.5 %	56.3 %
Total	10 %	25 %	59 %	50 %

Batch Sizes The different batch size values tested are 4, 50, 100, and 150. In Figure 5, the training loss and accuracy indicate that the lower the batch size, the higher the accuracy. This makes sense, since the lower the batch size (closer to the Batch Gradient Descent), the more instances in a batch, and thus the model is more certain about the direction of the update. In Table 2, the test accuracy of the batch size 4 is the highest, while the accuracy is dropping when batch size increases.

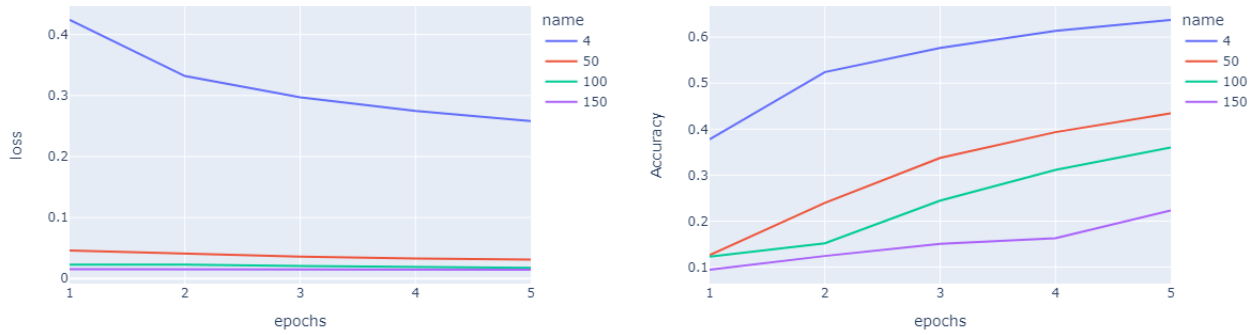

Figure 5: Training loss and accuracy for different batch sizes. The network was trained with a learning rate of 0.001, momentum of 0.9, and for 5 epochs.

Table 2: The accuracy of the network on the 10000 test images after 5 epochs.

Accuracy	**BatchSize 4**	**BatchSize 50**	**BatchSize 100**	**BatchSize 150**
Class of plane	71.1 %	44.7 %	47.7 %	32.5 %
Class of car	79.8 %	66.2 %	39.9 %	19.4 %
Class of bird	36.5 %	15.6 %	14.1 %	3.5 %
Class of cat	56.1 %	22.8 %	7.7 %	12.1 %
Class of deer	45.0 %	35.2 %	37.8 %	1.0 %
Class of dog	52.7 %	40.9 %	34.8 %	12.9 %
Class of frog	74.8 %	53.5 %	59.6 %	41.9 %
Class of horse	62.0 %	63.0 %	50.7 %	44.3 %
Class of ship	74.8 %	53.2 %	46.0 %	35.7 %
Class of truck	54.8 %	54.1 %	43.6 %	54.4 %
Total	60 %	44 %	38 %	25 %

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.

Answer In this section, three different optimizers are examined, namely SGD, Adam, and RMSprop. In Figure 6, the training loss and accuracy are reported. Based on the results, the SGD starts with the highest loss, but quickly drops, reaching the same loss and the same accuracy as the Adam optimizer. RMSprop optimizer, on the other hand, starts with the lowest loss (close to Adam optimizer) but ends with the highest loss and the lowest accuracy compared to the other two optimizers. Taking into account the test accuracies shown in Table 3, RMSProp performs the worst compared to the other optimizers, while the SGD and Adam optimizers have the same total accuracy and approximately the same accuracy for each class.

Figure 6: Training loss and accuracy for activation functions. The network was trained with a learning rate of 0.001, batch size of 4, momentum of 0.9, and for 5 epochs.

Table 3: The accuracy of the network on the 10000 test images after 5 epochs.

Accuracy	**SGD**	**Adam**	**RMSprop**
Class of plane	70.0 %	58.8 %	70.5 %
Class of car	79.2 %	80.6 %	79.4 %
Class of bird	51.7 %	48.8 %	27.5 %
Class of cat	34.4 %	46.3 %	41.7 %
Class of deer	52.5 %	54.4 %	51.0 %
Class of dog	38.8 %	46.7 %	33.8 %
Class of frog	72.0 %	71.6 %	59.3 %
Class of horse	64.6 %	66.1 %	61.3 %
Class of ship	74.6 %	68.8 %	50.9 %
Class of truck	69.5 %	62.3 %	73.4 %
Total	60 %	60 %	54 %

Appendix

A ReLU Implementation Code

```
def relu(x):
    """
    Wrap the relu op in a funciton (just for symmetry with the softmax).

    :param x:
    :return:
    """
    return ReLU.do_forward(x)

class ReLU(Op):
    """
    Op for element-wise application of relu function
    """

    @staticmethod
    def forward(context, input):
        relux = np.maximum(0, input)
        context['relux'] = relux # store the relu of x for the backward pass
```

```
        return relux

    @staticmethod
    def backward(context, goutput):
        relux = context['relux'] # retrieve the relu of x
        derivative = relux
        derivative[relux > 0] = 1 # if x > 0 then derivative equals to 1 else equals to 0
        return goutput * derivative
```

B Weight Initialization

```
class Linear(Module):
    """
    A linear operation. Applies a matrix transformation and a vector translation.
    """

    def __init__(self, input_size, output_size, init='glorot'):
        super().__init__()

        w = np.zeros(output_size, input_size)

        # weights of the matrix transformation
        if init == 'he':
            he_std = np.sqrt(2.0 / (input_size)) # scalar for He normal init
            w = np.random.randn(output_size, input_size) * he_std
        elif init == 'he_uniform':
            he_uni_std = np.sqrt(6.0 / (input_size)) # scalar for He uniform init
            w = np.random.randn(output_size, input_size) * he_uni_std
        elif init == 'glorot_uniform':
            glorot_uni_std = np.sqrt(6.0 / (input_size + output_size)) # scalar for Glorot uniform init
            w = np.random.randn(output_size, input_size) * glorot_uni_std
        elif init == 'glorot':
            glorot_std = np.sqrt(2.0 / (input_size + output_size)) # scalar for Glorot normal init
            w = np.random.randn(output_size, input_size) * glorot_std

        self.w = TensorNode(w)

        # weights of the bias (the translation)
        b = np.zeros((1, output_size))
        self.b = TensorNode(b)
        # -- We initialize the biases to zero for simplicity. This is a common approach, but with ReLU units it's
        #    sometimes best to add a little noise to avoid dead neurons.
```