

## Submission Assignment 2

Name: Ángel A. Gil Álamo, Student ID: agi239

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** In order for the element-wise operation to make sense,  $X$  and  $Y$  must be of the same dimensions, namely  $X \in \mathbb{R}^{n \times m}$ ,  $Y \in \mathbb{R}^{n \times m}$ . Then, in  $f(X, Y) = \frac{X}{Y} = S$ , the resulting  $S$  also belongs to  $\mathbb{R}^{n \times m}$ . Figure 1 shows the computation graph if we consider  $S$  as the output and finally a loss represented by the scalar  $l$ .

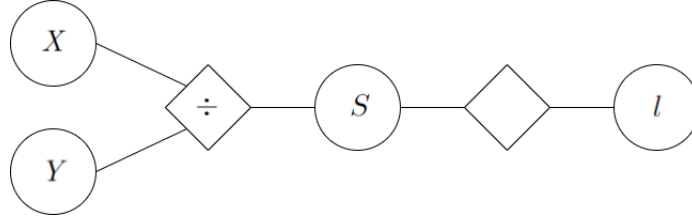


Figure 1: computation graph of exercise 1

The derivatives for  $X$  and  $Y$  are obtained as follows:

$$\begin{aligned}
 \bullet \quad X_{ij}^\nabla &= \sum_{kl} \frac{\partial l}{\partial S_{kl}} \frac{\partial S_{kl}}{\partial X_{ij}} = \sum_{kl} S_{kl}^\nabla \frac{\partial S_{kl}}{\partial X_{ij}} = \sum_{kl} S_{kl}^\nabla \frac{\partial [X/Y]_{kl}}{\partial X_{ij}} = \sum_{kl} S_{kl}^\nabla \frac{\partial X_{kl}/Y_{kl}}{\partial X_{ij}} = S_{ij}^\nabla \frac{\partial X_{ij}/Y_{ij}}{\partial X_{ij}} = \boxed{S_{ij}^\nabla \frac{1}{Y_{ij}}} \\
 \bullet \quad Y_{ij}^\nabla &= \sum_{kl} \frac{\partial l}{\partial S_{kl}} \frac{\partial S_{kl}}{\partial Y_{ij}} = \sum_{kl} S_{kl}^\nabla \frac{\partial S_{kl}}{\partial Y_{ij}} = \sum_{kl} S_{kl}^\nabla \frac{\partial [X/Y]_{kl}}{\partial Y_{ij}} = \sum_{kl} S_{kl}^\nabla \frac{\partial X_{kl}/Y_{kl}}{\partial Y_{ij}} = S_{ij}^\nabla \frac{\partial X_{ij}/Y_{ij}}{\partial Y_{ij}} = \boxed{-S_{ij}^\nabla \frac{X_{ij}}{Y_{ij}^2}}
 \end{aligned}$$

That is the expression for each element of  $X$  and  $Y$ . In tensor notation, we would have

$$\boxed{X^\nabla = S^\nabla \frac{1}{Y}} \quad \text{and} \quad \boxed{Y^\nabla = -S^\nabla \frac{X}{Y^2}}$$

## Question 2

Let  $f$  be a scalar-to-scalar function  $f: \mathbb{R} \rightarrow \mathbb{R}$ . 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** This is a generalization of the previous question. Again, let  $f(X) = S$ , Figure 2 shows the computation graph for this case, where the gradient is obtained as follows:

$$X_{ij}^\nabla = \sum_{kl} \frac{\partial l}{\partial S_{kl}} \frac{\partial S_{kl}}{\partial X_{ij}} = \sum_{kl} S_{kl}^\nabla \frac{\partial S_{kl}}{\partial X_{ij}} = \sum_{kl} S_{kl}^\nabla \frac{\partial f(X_{kl})}{\partial X_{ij}} = S_{ij}^\nabla \frac{\partial f(X_{ij})}{\partial X_{ij}} = \boxed{S_{ij}^\nabla f'(X_{ij})}$$

From where the gradient would be written as  $\boxed{X^\nabla = S^\nabla f'(X)}$

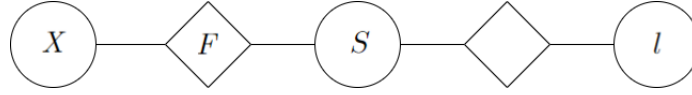


Figure 2: computation graph of exercise 2

### 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**  $\mathbf{X} \in \mathbb{R}^{n \times f}$  and, since there are  $f$  input nodes and  $m$  output nodes,  $\mathbf{W} \in \mathbb{R}^{f \times m}$ . The output  $S$  would be obtained as  $S = \mathbf{X}\mathbf{W}$  where  $S \in \mathbb{R}^{n \times m}$ , (i.e. a  $n$ -by- $m$  batch of  $n$  outputs with  $m$  features each). Nevertheless, I have seen that in lecture 2 the notation for  $\mathbf{W}$  is inverted, being *first index*  $\equiv$  *column* and *second index*  $\equiv$  *row*. Following this, then  $\mathbf{W} \in \mathbb{R}^{m \times f}$ . Although this notation is contrary to what I believe is the standard for matrices, I feel like it is important to mention that, this way,  $S$  would be obtained as  $S = \mathbf{X}\mathbf{W}^T$ . Figure 3 shows the computation graph for this question and Figure 4, my interpretation of  $\mathbf{W}$ , which is the one for which I will work out the gradients. Note that I skip some steps and that for  $S = \mathbf{X}\mathbf{W}^T$  the process would be analogous.

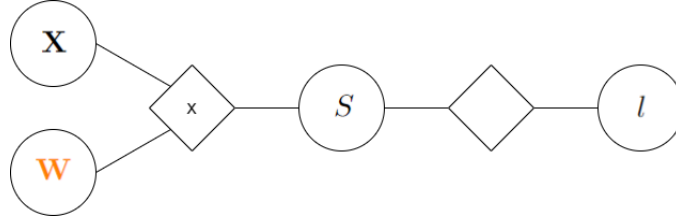
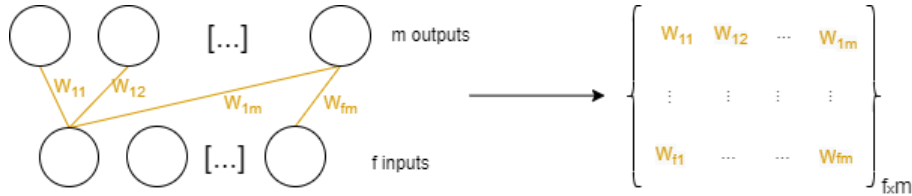


Figure 3: computation graph of exercise 3

Figure 4: My interpretation for the dimensions of  $\mathbf{W}$ 

$$\begin{aligned}
 \bullet \quad \mathbf{X}_{ij}^\nabla &= \sum_{kl} \frac{\partial l}{\partial S_{kl}} \frac{\partial S_{kl}}{\partial \mathbf{X}_{ij}} = \sum_{kl} S_{kl}^\nabla \frac{\partial [\mathbf{X}\mathbf{W}]_{kl}}{\partial \mathbf{X}_{ij}} = S_{ij}^\nabla \frac{\partial \mathbf{X}_{ij} \mathbf{W}_{ij}}{\partial \mathbf{X}_{ij}} = \boxed{S_{ij}^\nabla \mathbf{W}_{ij}} \\
 \bullet \quad \mathbf{W}_{ij}^\nabla &= \sum_{kl} \frac{\partial l}{\partial S_{kl}} \frac{\partial S_{kl}}{\partial \mathbf{W}_{ij}} = \sum_{kl} S_{kl}^\nabla \frac{\partial [\mathbf{X}\mathbf{W}]_{kl}}{\partial \mathbf{W}_{ij}} = S_{ij}^\nabla \frac{\partial \mathbf{X}_{ij} \mathbf{W}_{ij}}{\partial \mathbf{W}_{ij}} = \boxed{S_{ij}^\nabla \mathbf{X}_{ij}}
 \end{aligned}$$

Finally,

$$\boxed{\mathbf{X}^\nabla = S^\nabla \mathbf{W}} \quad \text{and} \quad \boxed{\mathbf{W}^\nabla = S^\nabla \mathbf{X}}$$

### 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).

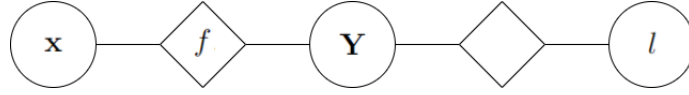


Figure 5: Computation graph of exercise 4

**Answer** Figure 5 shows the computation graph. In this case,  $\mathbf{x} \in \mathbb{R}^n$ ,  $\mathbf{Y} \in \mathbb{R}^{n \times 16}$ . With this in mind, let's work out the backward of  $f$

$$\mathbf{x}_i^\nabla = \sum_{kl} \frac{\partial l}{\partial \mathbf{Y}_{kl}} \frac{\partial \mathbf{Y}_{kl}}{\partial \mathbf{x}_i} = \sum_{kl} \mathbf{Y}_{kl}^\nabla \frac{\partial \mathbf{Y}_{kl}}{\partial \mathbf{x}_i} = \sum_{kl} \mathbf{Y}_{kl}^\nabla \frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_i} = \sum_k \frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_i} \sum_l \mathbf{Y}_{kl}^\nabla = \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_i} \sum_l \mathbf{Y}_{il}^\nabla = \sum_l \mathbf{Y}_{il}^\nabla$$

If we translate this expression to tensor notation (abusing the notation, in fact), we have  $\mathbf{x}^\nabla = \sum_{l=1}^{16} \mathbf{Y}_l^\nabla$

### 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** Figure 6 shows the basic script from which the computation graph of this question is recreated, together with the random values obtained for  $a$  and  $b$  the resulting sum  $c$ .

<pre>import vugrad as vg import numpy as np  # Eager execution a = vg.TensorNode(np.random.randn(2, 2)) b = vg.TensorNode(np.random.randn(2, 2))  c = a + b</pre>	<pre>a.value: [[-0.51321263  0.25623137]  [-0.37337302 -0.88394819]] b.value: [[ 0.45537562 -1.17750527]  [ 0.74176327  0.57617082]] c.value: [[-0.05783701 -0.9212739 ]  [ 0.36839026 -0.30777738]]</pre>
---	--

Figure 6: Left, script that produces the computation graph. Right, the random values obtained for  $a$  and  $b$ 

- 1) The attribute `c.value` contains the values of the element-wise sum of  $a + b$ , as we can see in Figure 6.
- 2) The attribute `c.source` refers (i.e. points) to the source node (i.e. `OpNode`) that produced  $c$  (i.e. its parent node). In this case, we obtain a pointer to the operation node of the sum of  $a$  and  $b$ . Figure 7 shows this, and also the results for the remaining parts of Question 5.
- 3) The attribute `c.source.inputs[0].value` refers to the values of the first node that took part into creating  $c$ , in this case, it contains the values of  $a$ .
- 4) The attribute `a.grad` refers to the values of the gradient of  $a$  (i.e.  $a^\nabla$ ). Right now it only contains zeros, since no backward computation has been performed yet.

<pre>b.source.inputs[0].value: [[-0.51321263  0.25623137]  [-0.37337302 -0.88394819]]</pre>	<pre>c.source: &lt;vugrad.core.OpNode object at 0x00000218A6CFD5D0&gt;</pre>	<pre>a.grad: [[0. 0.]  [0. 0.]]</pre>
---	--	---------------------------------------

Figure 7: Left, center, right respectively: output for sections 2), 3) and 4)

## 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) Figure 8 shows the output for the attribute `c.source.op` from the previous question. `vugrad` implements operation objects starting from an abstract class `Op`. This abstract class lists the methods (such as `backward`) that the concrete classes (such as `Add`) have to implement. For the example above, the attribute `c.source.op` is an `Add` object and virtually an `Op` object.

```
>>> c.source.op
<class 'vugrad.core.Add'>
```

Figure 8: the `OpNode.op` attribute is an `Add` object for the example of Q5

2) As we can see in Figure 9, the `__add__` method of `TensorNode` calls the `Add` class in line 108 (highlighted by a red box in the left caption). This class is defined starting at line 317 and the actual addition is not performed until line 324 (highlighted by a red box in the right caption).

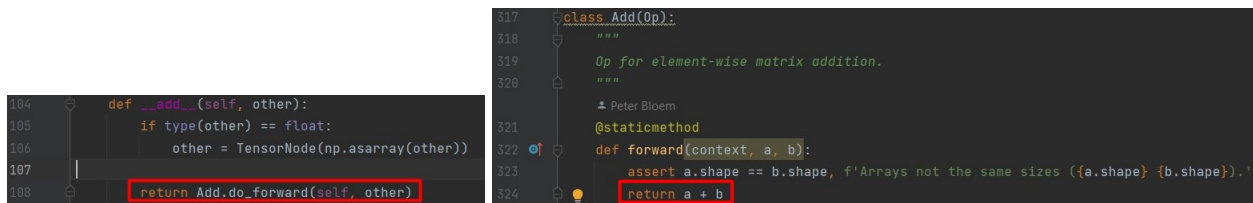


Figure 9: Left, `TensorNode` calls 'Add'. Right, the addition is ultimately performed at line 324

3) Figure 10 does a great job in illustrating why the pointer to the output node(s) is left `None` at first: because that(those) node(s) still has(have) to be created - *I will refer now to "them" as if there where multiple `TensorNodes` to be defined*. Once the `TensorNodes` are defined, specifying the `OpNode` that computed their values, the output node is linked with them. The outputs for the `OpNode` are set in line 249 (squared in red).

```
246 opnode = OpNode(cls, context, inputs)
247
248 outputs = [TensorNode(value=output, source=opnode) for output in outputs_raw]
249 opnode.outputs = outputs
```

Figure 10: Lines where `Op` initializes the `OpNode` and `TensorNode(s)` resulting from an operation

## 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** Figure 11 shows the `backward` method's code (of the class `TensorNode`). As highlighted in this picture, the call to the `Ops` that takes care of the computation is done at line 97.

```

80 def backward(self, start=True):
81     """
82     Start (or continue) the backpropagation from this node. This will fail if the node holds
83     scalar value.
84     """
85     if start:
86         if self.value.squeeze().shape != ():
87             raise Exception('backward() can only start from a scalar node.')
88
89         self.grad = np.ones_like(self.value)
90         # -- the gradient of the loss node is 1, with the same shape as the loss node itself
91
92         self.visits += 1
93
94         # If we've been visited by all parents, move down the tree
95         if self.visits == self.numparents or start:
96             if self.source is not None:
97                 self.source.backward()
98             else:
99                 assert self.visits < self.numparents, f'{self.numparents} {self.visits} {self.name}'

```

Figure 11: *TensorNode*'s backward function

## 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** Figure 12 shows both the forward and the backward of the *Expand Op*. The forward expands singleton dimensions of the given tensor (*input*) by repeating it a number of times (*repeats*) along a given dimension (*dim*). With this notation in mind, letting  $\mathbf{x}_i$  represent each of the scalars contained along the axis of the singleton dimension  $\text{input}[\text{dim}]$  (see Figure 13 for clarification about this abuse of notation) and  $\mathbf{Y}$  the output, Figure 14 represents the computation graph. As a final remark, the abuse of notation reduces the dimensions to  $i$  and  $j$ , where  $i$  represents the reduction of the shape of the tensor (namely  $\alpha \cdot \beta \cdot \gamma \dots$  to a single set, and  $j$  represents the expansion over the singleton dimension's axis. Now, we are ready to work out the gradient and check if the backward matches

$$\bullet \mathbf{x}_i^\nabla = \sum_{j=1}^{\text{repeats}} \mathbf{Y}_{ij}^\nabla \frac{\partial \mathbf{Y}_{ij}}{\partial \mathbf{x}_i} = \sum_{j=1}^{\text{repeats}} \mathbf{Y}_{ij}^\nabla \frac{\partial \mathbf{x}_i}{\partial \mathbf{x}_i} = \sum_{j=1}^{\text{repeats}} \mathbf{Y}_{ij}^\nabla$$

Then,  $\mathbf{x}^\nabla = \sum_{j=1}^{\text{repeats}} \mathbf{Y}_j^\nabla$  Assuming that  $\text{goutput} \equiv \mathbf{Y}_j^\nabla$ , then this is exactly what the backward part of the *Expand Op* is doing.

```

def forward(context, input, *, dim, repeats):
    assert input.shape[dim] == 1, 'Can only expand singleton dimensions'

    context['dim'] = dim

    return np.repeat(input, repeats, axis=dim)

# Peter Bloem
@staticmethod
def backward(context, goutput):
    dim = context['dim']

    return goutput.sum(axis=dim, keepdims=True)

```

Figure 12: Forward and backward of the *Expand Op*

## 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.

$\begin{bmatrix} [-0.5], [0.2] \\ [0.3], [-0.8] \end{bmatrix} = \text{input (shape } 2, 2, 1)$   
 $x_1 \dots x_n \rightarrow n = \alpha \cdot \beta \cdot \gamma \dots \text{ where } \alpha \neq \beta \neq \gamma$   
 $\begin{bmatrix} [-0.5], [0.2], [0.3], [-0.8] \\ [-0.5], [0.2], [0.3], [-0.8] \\ [-0.5], [0.2], [0.3], [-0.8] \end{bmatrix} = \text{input expanded (shape } 2, 2, 3)$   
 $\text{repeats} = 3, \text{axis} = 2$

Figure 13: Abuse of notation for the gradient of the Expand Op

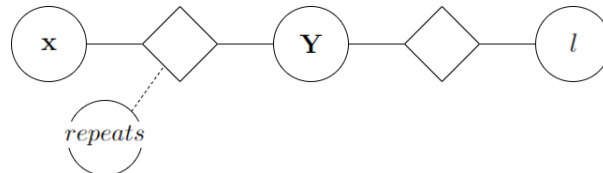


Figure 14: Computation graph of the Expand Op

**Answer** Figure 15 shows the code used for the implementation of the ReLU Op. Figure 16 shows the mean and standard deviation for the accuracy of the validation set over 3 runs for the neural network trained with each of the activation functions during 20 epochs. Although the accuracy grows in a similar trend for both activation functions, *sigmoid* comes out on top of *ReLU* consistently. The average accuracy for the last epoch is 96.65% for *ReLU* and 97.43% for *Sigmoid*, being the maximum accuracy 96.86% and 97.67% respectively.

```

class Relu(Op):
    """
    Op for element-wise application of relu function
    """
    @staticmethod
    def forward(context, input):
        relu = np.maximum(0, input)
        context['relu'] = relu # store the relu of x for the backward pass
        return relu

    @staticmethod
    def backward(context, goutput):
        relu = context['relu'] # retrieve the relu of x
        relu[relu <= 0] = 0
        relu[relu > 0] = 1
        return goutput * relu
  
```

Figure 15: Implementation of the ReLU Op

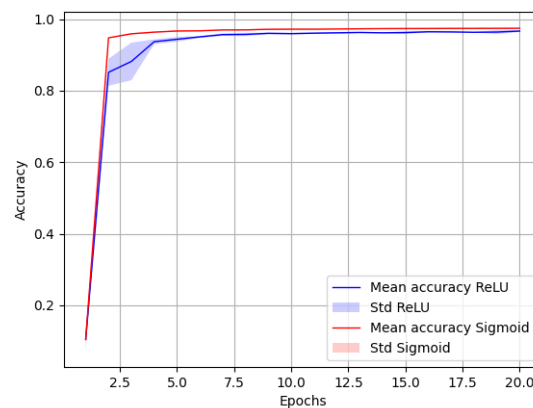


Figure 16: Mean and std of accuracy for ReLU (blue) and Sigmoid (red) activation functions



## 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** Figure 17 show the addition of a hidden layer of the same dimensions of the one already present, and the implementation of residual connection between layers. All of this is included in `experiments/A2Q10_3layers`. Figures 18, 19 and 20 respectively show the comparison between the original model's performance, the one with an additional hidden layer and the one with residual connection between layers for the last 5 epochs. There is a small improvement in the performance of the model using the additional layer, although the computational cost is considerable. As for the residual connection between layers, the performance based on the last five epochs is slightly worse than that of the original network.

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

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

    @staticmethod
    def backward(context, goutput):
        relu = context['relu'] # retrieve the relu of x
        relu[relu<0] = 0
        relu[relu>0] = 1
        return goutput + relu
```

Figure 17: Implementation of an additional layer and residual connection

```
epoch 015
    accuracy: 0.9708
    running loss: 0.02315
epoch 016
    accuracy: 0.9708
    running loss: 0.02135
epoch 017
    accuracy: 0.9712
    running loss: 0.01984
epoch 018
    accuracy: 0.9716
    running loss: 0.0185
epoch 019
    accuracy: 0.972
    running loss: 0.01732
```

Figure 18: Base algorithm

```
epoch 015
    accuracy: 0.9772
    running loss: 0.02363
epoch 016
    accuracy: 0.9776
    running loss: 0.02179
epoch 017
    accuracy: 0.9776
    running loss: 0.02022
epoch 018
    accuracy: 0.9778
    running loss: 0.01883
epoch 019
    accuracy: 0.9774
    running loss: 0.01762
```

Figure 19: Additional layer

```
epoch 015
    accuracy: 0.964
    running loss: 0.08815
epoch 016
    accuracy: 0.9646
    running loss: 0.08226
epoch 017
    accuracy: 0.9642
    running loss: 0.07699
epoch 018
    accuracy: 0.9656
    running loss: 0.07217
epoch 019
    accuracy: 0.9662
    running loss: 0.06771
```

Figure 20: Residual connection

Figure 21 shows the addition of momentum, which is implemented in `experiments/A2Q10_momentum`. Figure 22 shows the results using SGD with momentum, which does not show any performance improvement over the base model. Figures 23 and 24 show, respectively, the training of the base model initializing all weights to zero and from a standard normal distribution. The former performs very poorly, with a really low accuracy

and no real improvement in decreasing the loss, which stress the importance of initialization. The latter does not perform too well, but it is closer to the results of the original Glorot's initialization.

```

273 momentum = []
274 for param in mlp.parameters():
275     momentum.append(np.zeros(param.size()))
276 gamma = 0.9
277 i=0
278 # SGD
279 for param in mlp.parameters():
280     momentum[i] = gamma*momentum[i] + param.grad
281     param.value -= args.lr * momentum[i]
282     i+=1

```

Figure 21: Implementation of momentum

```

epoch 015
    accuracy:0.9722
    running loss: 0.02325
epoch 016
    accuracy:0.9726
    running loss: 0.02146
epoch 017
    accuracy:0.9726
    running loss: 0.0199
epoch 018
    accuracy:0.9728
    running loss: 0.01852
epoch 019
    accuracy:0.9732
    running loss: 0.01731

```

Figure 22: Base with momentum

```

epoch 015
    accuracy:0.3214
    running loss: 1.758
epoch 016
    accuracy:0.1251
    running loss: 1.759
epoch 017
    accuracy:0.1206
    running loss: 1.755
epoch 018
    accuracy:0.1510
    running loss: 1.762
epoch 019
    accuracy:0.1355
    running loss: 1.755

```

Figure 23: Weights init as 0s

```

epoch 015
    accuracy:0.8838
    running loss: 1.812
epoch 016
    accuracy:0.8853
    running loss: 1.742
epoch 017
    accuracy:0.8896
    running loss: 1.674
epoch 018
    accuracy:0.8883
    running loss: 1.6002
epoch 019
    accuracy:0.8879
    running loss: 1.559

```

Figure 24: Normal init weights

## 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** Mainly because neither have I stored the run loss over the epochs, nor the validation accuracy over the tests; I will not show plots for the different hyperparameters sets that I have used. During the experiments, I have played with the parameters *batch size*, *epochs* and *learning rate*, setting the rest to their default values.

Table 1 shows the results of the first experiment, where the learning rate (LR) was set to its default value, 0.001. In this table we can see the lost obtained in the last epoch and the validation set's accuracy for each of the set of hyperparameters. Since a bigger batch size sped up the process substantially, I tried to create a balance between number of epochs and batch size, having a similar *budget*, taking as baseline the first row of the table, which corresponds to the default settings used in the tutorial. The criteria consist on an increment on the number of epochs proportional to the batch size. Although the relation in CPU time is not linear, it came out close during the testing. The results made me think that a batch size not too big and not too small might be preferable, but I could not come yet to a conclusion for the decision of a final hyperparameter set to test for a larger number of epochs.

For further insight, I chose to test the same sets of batch size and n° of epochs with LR's of 0.01, 0.0001 and 0.03, but the loss with  $LR = 0.0001$  was dropping too slowly, being superior to 2.0 after 10 epochs for all cases tested. For  $LR = 0.01$  the loss dropped much faster, but after the first couple epochs the progress was shaky and slower than that of the default LR. Table 2 shows the results for  $LR = 0.03$ , where in contrast to the previous experiment, the largest batch size tested (32) came ahead. Since the disparate number of epochs



Table 1: First experiment, checking behaviour with similar budget (LR = 0.001)

Batch Size	Number of epochs	Final epoch's loss	Accuracy
4	2	1.278	53%
12	6	1.003	60%
20	10	0.991	62%
24	12	0.936	63%
32	16	0.986	61%

might invalidate a direct comparison, I decided to train the most promising sets for the same number of epochs in a third experiment.

Table 2: Second experiment, checking behaviour with similar budget (LR = 0.003)

Batch Size	Number of epochs	Final epoch's loss	Accuracy
12	6	0.875	62%
20	10	0.812	63%
24	12	0.783	63%
<b>32</b>	<b>16</b>	<b>0.681</b>	<b>64%</b>

Table 3 shows the results of the last experiment, where the batches of sizes {24, 32} were tested for {30, 50} epochs with a learning rate of 0.003, resulting in 4 runs. As a side-note, the results for learning rate 0.001 were similar, so I decided not to include them. In any case, the resulting accuracy was unexpectedly worse in this experiment than the obtained in the previous one, while the loss improved significantly. The only reasonable cause that I can think of is that it happens due to an overfitting of the weights of model for the training set.

After all, the hyperparameter set that produced the best outcome was obtained in the second experiment. It is {32, 16, 0.003} for batch size, number of epochs and learning rate respectively, obtaining an accuracy of 64%.

Table 3: Third experiment, training the most promising sets for longer

Batch Size	Number of epochs	Final epoch's loss	Accuracy
24	30	0.457	61%
24	50	0.446	59%
32	30	0.553	62%
32	50	0.309	61%

## 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** For this question, I went back to the default settings (2 epochs, LR=0.001, batch size=4). After exploring a few of the optimizers of `torch.optim`, I ended up testing Adagrad, Adam and RMSprop, and comparing them with the baseline SGD. The highest overall accuracy, 56%, was achieved by RMSprop. Figure 25 shows the accuracy obtained by each of the optimizers across the different categories. Here, we can see that RMSprop performed well overall, being the worse at identifying dogs, but not last in any other category. Adagrad (51% overall accuracy), on the other hand, performed considerably worse than the rest in most of the categories. SGD (53%) performed fairly well compared to the rest, coming out close with Adam (53%). For further visualization, figure 26 shows the loss curves over the 2 epochs.

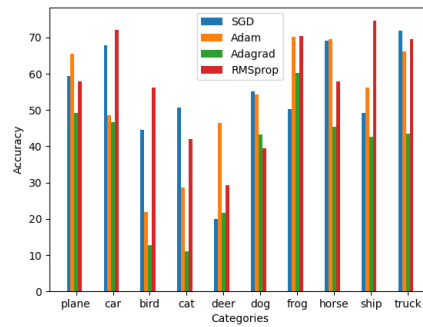


Figure 25: Final accuracy for each of the optimizers

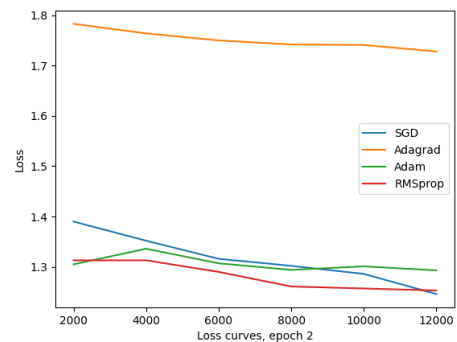
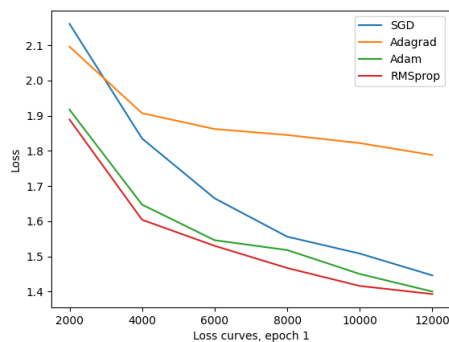


Figure 26: Loss curves over the 2 epochs

### 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.