Course: Deep Learning **(28.11.2021)**

Submission Assignment #2

Coordinator: Jakub Tomczak *Name:* Bartłomiej Boczek, *Netid:* bbk205

1 Part 1: Working out backward functions

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.*

- Backward for X:
  $X_{ij}^{\nabla} = \frac{\partial l}{\partial X_{ij}} = \sum_{kl} S_{kl}^{\nabla} \frac{\partial S_{kl}}{\partial X_{ij}} = \sum_{kl} S_{kl}^{\nabla} \frac{\partial [\frac{X}{Y}]_{kl}}{\partial X_{ij}} = S_{ij}^{\nabla} \frac{1}{Y_{ij}}$
  thus,
  $X^{\nabla} = S^{\nabla} \frac{1}{Y}$
- Backward for Y:
  $Y_{ij}^{\nabla} = \frac{\partial l}{\partial Y_{ij}} = \sum_{kl} S_{kl}^{\nabla} \frac{\partial S_{kl}}{\partial Y_{ij}} = \sum_{kl} S_{kl}^{\nabla} \frac{\partial [\frac{X}{Y}]_{kl}}{\partial Y_{ij}} = -S_{ij}^{\nabla} \frac{X_{ij}}{Y_{ij}^2}$
  thus,
  $Y^{\nabla} = -S^{\nabla} \frac{X}{Y^2}$

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.*

$X_i^{\nabla} = \sum_i Y_i^{\nabla} \frac{\partial Y_i}{\partial X_i} == \sum_i Y_i^{\nabla} \frac{\partial F(X_i)}{\partial X_i} == Y_i^{\nabla} \frac{\partial F(X_i)}{\partial X_i} == Y_i^{\nabla} f^{'}(X_i)$
thus,
$X^{\nabla} = Y^{\nabla} f^{'}(X)$
The results shows, that backward of F is the element-wise application of $f^{'}$ applied to the elements of X, multiplied by the gradient of the loss with respect to the outputs.

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.*

The layer outputs can be computed with the linear combination of weight matrix W and the input matrix X.

- Backward for W:
  $W_{ij}^{\nabla} = \frac{\partial l}{\partial W_{ij}} = \sum_{kl} k_{kl}^{\nabla} \frac{\partial k_{kl}}{\partial W_{ij}} = \sum_{kl} k_{kl}^{\nabla} \frac{\partial [WX]_{kl}}{\partial W_{ij}} = k_{ij}^{\nabla} X_{ij}$
- Backward for X:
  $X_{ij}^{\nabla} = \frac{\partial l}{\partial X_{ij}} = \sum_{kl} k_{kl}^{\nabla} \frac{\partial k_{kl}}{\partial X_{ij}} = \sum_{kl} k_{kl}^{\nabla} \frac{\partial [WX]_{kl}}{\partial X_{ij}} = k_{ij}^{\nabla} W_{ij}$
  thus,
  $W^{\nabla} = k^{\nabla} X, X^{\nabla} = k^{\nabla} W,$

Question 4 *Let f(x) = Y be a function that takes a vector x, and returns the matrix Y consisting of 16 columns that are all equal to x. Work out the backward of f. (This may seem like a contrived example, but it's actually an instance of broadcasting).*

The output of this function is the matrix shape(X) x 16. The value of columns - 16 is constant, thus gradient will be the same for all of the columns and only for x.
$x_i^{\nabla} = \sum_{kl} Y_{kl}^{\nabla} \frac{\partial Y_{kl}}{\partial x_i} = \sum_l Y_{il}^{\nabla} \frac{\partial x_{il}}{\partial x_i} = \sum_l Y_i l^{\nabla}$

2 Part 2: Backpropagation

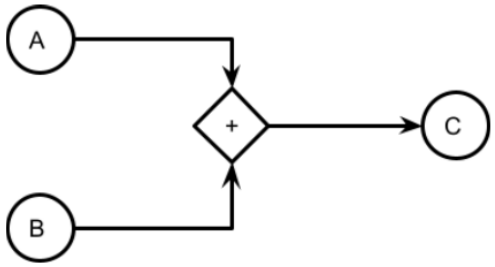

Figure 1: Computation graph for the operation $C \leftarrow A + B$

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))
Recreate the computation graph above: create two tensor nodes a and b containing numpy arrays of the same size, and sum them (using the + operator) to create a TensorNode c. Answer the following questions (in words, tell us what these class members mean, don't just copy/paste their values).

The graph from figure 1 is implemented using vugrad library by the following code:

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

1. *What does c.value contain?*

   It contains a numpy array with the result of the summation of 2 matrices $a + b$

2. *What does c.source refer to?*

   It contains the reference of the OpNode i.e. the operational node used to create this TensorNode. In this case c.source is the OpNode with operation of type vugrad.core.Add

3. *What does c.source.inputs[0].value refer to?*

   In this case it refers to TensorNode a, so the first input of the OpNode, so the first ingredient of the addition defined with the computational graph.

4. *What does a.grad refer to? What is its current value?*

   It refert to the value of the gradient of TensorNode a. Before .backward is run on this node, it is set to the matrix of zeros, and filled with the gradient during .backward() - so during backpropagation run. Current value is square matrix of size $2x2$ with zeros.

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?* The operation of OpNNode object is defined by the Op abstract class, which can be subclassed to define certain operations. Those operations have to implement **forward** and **backward** methods, as defined in Op interface.

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?*

   It is defined in core.py:324

```
    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}).'
324         return a + b
```

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?*

   It's done because the computational graph is eagerly executed, it's build on the fly. The graph only defines the flow of the computations, but not their results. OpNode is connected to the output nodes in core.py: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?

It happens in core.py: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.*

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.*

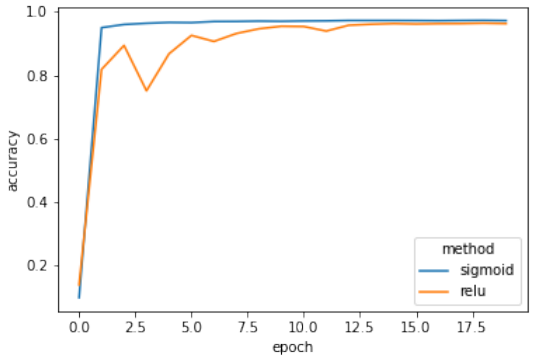

Figure 2: Comparsion of 2 MLP trainings with different nonlinearities - ReLU and Sigmoid - trained on MNIST data

ReLU nonlinearity was defined using vg.ops.Ops interface. It implements both forward and backward methods. Also functional API for relu was added. At the last MLP with relu nonlinearity was defined.

```
class ReLU(vg.ops.Op):
```

```
    """
    Op for element-wise application of ReLU function
    """

    @staticmethod
    def forward(context, input):
        #print(input.shape)
        relux = input * (input > 0)
        context['relux'] = relux
        return relux

    @staticmethod
    def backward(context, goutput):
        relux = context['relux']
        drelux = np.greater(relux, 0).astype(int)
        return drelux*goutput

def relu(x):
    """ Wrap the sigmoid op in a funciton (just for symmetry with the softmax). """
    return ReLU.do_forward(x)

class MLP_ReLU(MLP):
    def forward(self, input):
        assert len(input.size()) == 2
        hidden = self.layer1(input)
        hidden = relu(hidden)
        output = self.layer2(hidden)
        output = vg.logsoftmax(output)
        return output
```

In the figure 2 we can see the comparison of 2 trainings of MLP classifier in the mnist dataset. Accuracy on the validation set is presented. Besides different nonlinearity - ReLU vs Sigmoid - all other parameters was the same, i.e learning rate = 0.0001, hidden size = 4 * input size, 1 hidden layer and Glorot initialization of weights, 20 epochs and batch size of 128.

From the figure, we can see, that validation accuracy of training on MNIST dataset with ReLU is less stable among epochs than using sigmoid. Both models achieved similar accuracy at the end, but sigmoid model is slightly better: 96.36 % (ReLU) compared to 97.26 % (Sigmoid).

Question 10* *Change the network architecture (and other aspects of the model) and show how the training behavior changes. Here are some ideas (but feel free to try something else).*

In this question we are checking whether increasing the number of hidden neurons is helpful on the MNIST dataset. The number of hidden neurons used is hidden size = 8 * input size. Both Sigmoid and ReLU activations were checked.

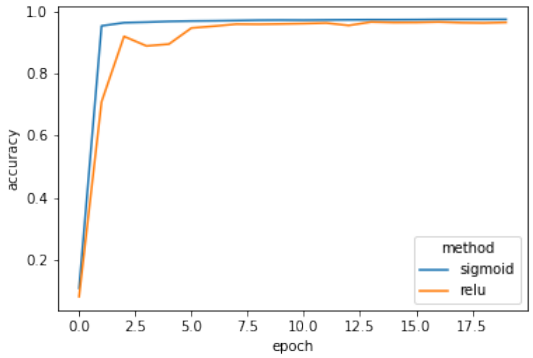

Figure 3: Comparsion of 2 MLP trainings with different nonlinearities - ReLU and Sigmoid - trained on MNIST data. hidden size = 8 * input size

Results of the training progress are showed in Figure 3. They are not very different when compared to using smaller hidden size. They are only slightly better, for sigmoid accuracy was 97.44% and for ReLU 96.6%.

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.*

3 Problem statement

- The problem in this assignment is an image classification task on CIFAR10 dataset.
- The objective is to classify input images to one of the classes.

4 Methodology

- Dataset has the classes: 'airplane', 'automobile', 'bird', 'cat', 'deer', 'dog', 'frog', 'horse', 'ship', 'truck'. The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of 32x32 pixels in size.
- Dataset is split by authors into 2 sets: train (60k samples) and test (10k samples). In this experiments train set is split into train set (50k samples) and validation set (10k samples) and all experiments are evaluated on the validation set. Final results will be given on the testset.
- Accuracy is used as a performance metric - in hyperparameter tuning it is calculated on the validation set, final result is the accuracy on the testset.
- 2 layer Convolutional Neural Network is used as a classifier. First conv layer is max-polled. Filters are of size 3x6x5 and 6x16x5 respectively. Outputs of the second filter are concatenated into fully-connected layer. Than one hidden and one output layers are used.

5 Experiments

- In experiments we try to find best performing set of hyperparameters for given neural netowrk architecture.
- Parameters that were examined are batch size, learning rate, optimized and criterion.
- Model is evaluated on the validation set every 2000 updates (every time 2000 batches are processed).
- Batch sizes of 1,2,3,4,8 and 16 was checked for lr=0.001 and sgd optimizer.
- Learning rates of 0.001, 0.01, 0.005, and 0.05 was checked for batch size =16 and sgd optimizer.
- All experiments was trained for 10 epochs.
- All the experiments was implemted using Pytorch and trained on 1 Nvidia GeForce RTX 2060 CUDA gpu.

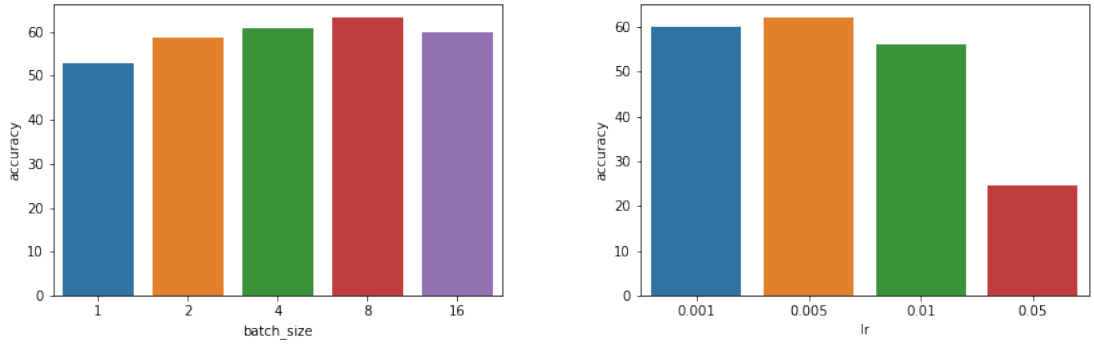

Figure 4: Batch size and learning rate for different experiments

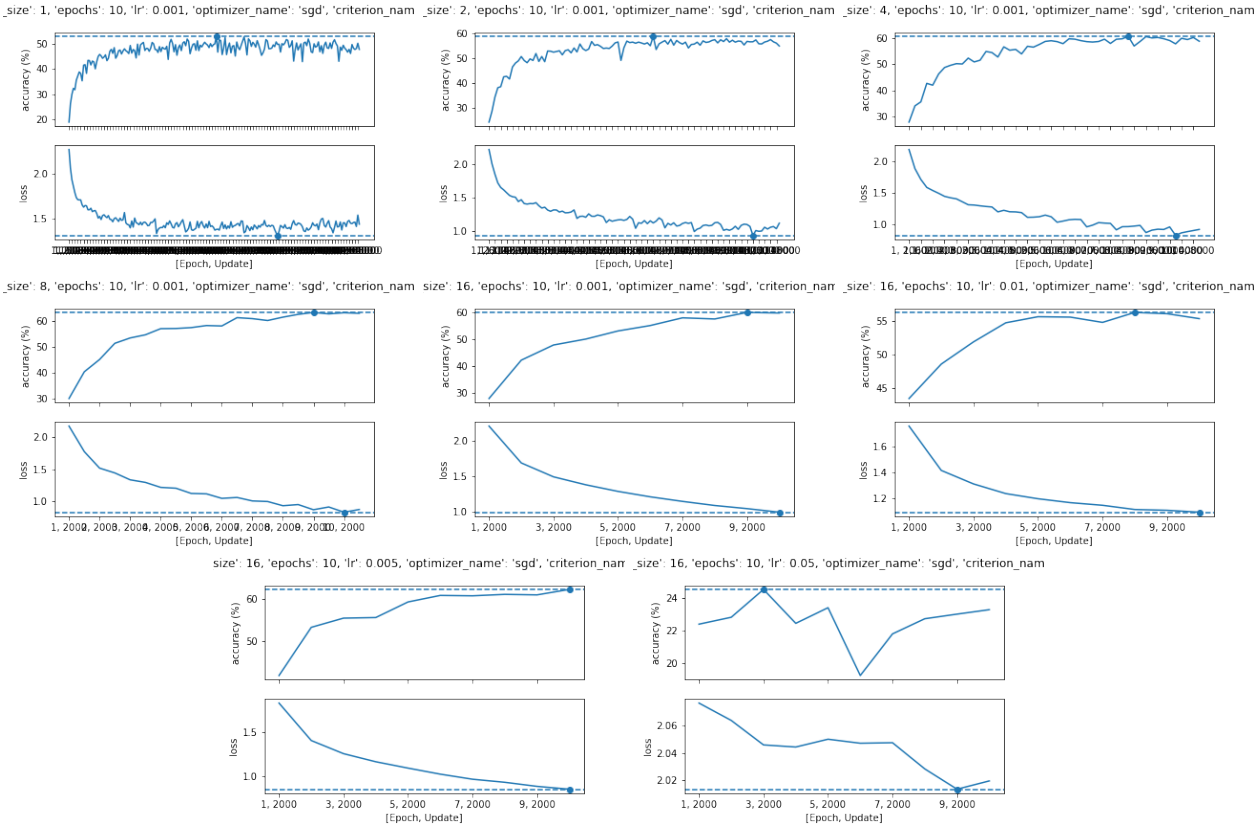

Figure 5: Learning curves, rows 1,2 - different batch sizes, row3 - different learning rates

6 Results

From the hyperparamters search presented in the figure 4 we can see that batch size of 8 and learning rate = 0.005 are best. Also from the learning curves presented in the figure 5 we can see, that for batch size = 8 best validation accuracy was achieved just before 9th epoch. In overview best eperiment top out o around 60% accuracy. Final testset evaluation was done with batch-size = 0, learning rate 0.005 and the model was trained for 9 epochs.

Here are the results on the testset:

Test accuracy of the network on the 10000 test images: 56 %
Accuracy for class plane is: 57.8 %
Accuracy for class car is: 71.2 %
Accuracy for class bird is: 39.8 %
Accuracy for class cat is: 39.0 %
Accuracy for class deer is: 58.5 %
Accuracy for class dog is: 30.5 %
Accuracy for class frog is: 76.5 %
Accuracy for class horse is: 54.4 %
Accuracy for class ship is: 70.5 %
Accuracy for class truck is: 70.1 %

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.*

7 Experiments

- SGD, RMSProp and Adam optimizers was checked with batch size = 16 and learning rate = 0.001

8 Results

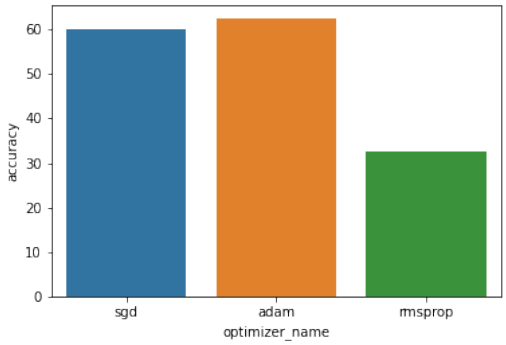

Figure 6: Results for different optimizers

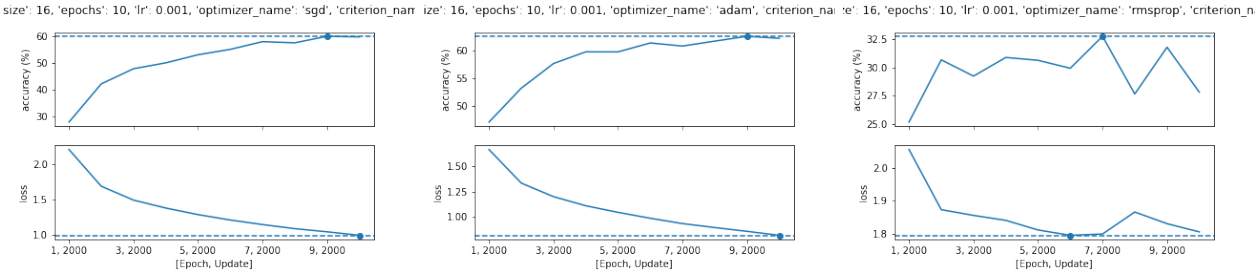

Figure 7: Learning curves for different optimizers, from left SGD, ADAM, RMSPROP

From the results presented in figure 6 we can see that SGD and Adam achieved very similar performance and rmsprop is much worse. It might be due to we are using default parameters for all the algorithms, it might be that rmsprop would be better after some parameters tuning. Because model optimized with Adam performed best on the validation set, final model will be trained with best hyperparameters from Question 11 with the addition of Adam Optimizer.

At the end I have decided to reduce learning rate to 0.001, because for 0.005 and Adam model achieved only 40 % accuracy. Below results are presented for 0.001 learning rate:

Test accuracy of the network on the 10000 test images: 59 %
Accuracy for class plane is: 72.2 %
Accuracy for class car is: 58.4 %
Accuracy for class bird is: 44.9 %
Accuracy for class cat is: 59.4 %
Accuracy for class deer is: 52.8 %
Accuracy for class dog is: 40.9 %
Accuracy for class frog is: 70.3 %
Accuracy for class horse is: 63.2 %
Accuracy for class ship is: 65.0 %
Accuracy for class truck is: 64.7 %

References
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.