Course: Deep Learning *Coordinator:* Jakub Tomczak

Submission Assignment 2

Name: Max Emanuel Feucht, *Student ID:* 2742061

Introduction

Training Deep neural networks requires the adaption of the network's paramters in a manner that the objective function or the loss of the network is minimzed. A commonly used tool to attain this is Gradient Descent, whereby the network's parameters are updated in the opposite direction as the gradient of the loss with respect to the parameters. How the gradient is computed for the single network parameters highly depends on the architecture of the network and the operations used in the network. Deriving the gradients for the single parameters by hand for a new neural network, as done in the first Assignment, would thus be highly infeasible. Automatic differentiation provides an approach to compute the gradient for parameters step-by-step, operation-wise. Given a particular forward operation, such as the sigmoid function, the backward operation can be worked out by hand. Then, given the the gradient of the loss with respect to the outputs of that operation, the gradient of the loss with respect to the inputs of that operation can be computed. This approach allows to stack different operations in a modular manner, passing the gradients from the top of the network to the bottom layers, until all necessary gradients are computed. In the report at hand, automatic differentiation is being explored by deriving backward functions by hand and subsequently applied using the mini differentiation library *vugrad*. Finally, we use the acquired knowledge to train an image detection network using the *pytorch* package.

In the following section, we derive the backward for a given forward by hand. In doing so, we show how we can compute the gradient in a modular manner, when only given the forward functions.

Backward for element-wise division (Question 1)

For the function $f(X, Y) = X/Y$, whereby "/" denotes element-wise division, we need to work out the backward for X and Y separately. To make notation easier and more understandable, I write X/Y as Z. The gradient of the loss w.r.t. the output of the forward function is $\nabla(X/Y)$, which can thus be written as ∇Z.

To find the gradient of the loss w.r.t. X (and Y, respectively), we first try to find a derivative of the loss w.r.t. a single element of X, x_{ijk} (and y_{ijk}, respectively).

Derivative of the loss w.r.t. X:

$$\nabla x_{ijk} = \sum_{abc} \nabla z_{abc} \frac{\partial z_{abc}}{\partial x_{ijk}} = \sum_{abc} \nabla z_{abc} \frac{\partial x_{abc} y_{abc}^{-1}}{\partial x_{ijk}}, \text{ whereby } \frac{\partial x_{abc} y_{abc}^{-1}}{\partial x_{ijk}} = \begin{cases} \frac{\partial x_{ijk} y_{ijk}^{-1}}{\partial x_{ijk}}, & \text{iff } a, b, c = i, j, k \\ 0 & \text{otherwise.} \end{cases},$$

with a, b, c denoting the element's positions in a three-dimensional tensors X and Y, and i, j, k denoting the position of the element x in X with respect to which we compute the derivative of the loss for.

This gives: $\nabla x_{ijk} = \nabla z_{abc} \frac{\partial x_{ijk} y_{ijk}^{-1}}{\partial x_{ijk}}$, whereby $\frac{\partial x_{ijk} y_{ijk}^{-1}}{\partial x_{ijk}} = y_{ijk}^{-1}$, so $\nabla x_{ijk} = \nabla z_{abc} y_{ijk}^{-1}$ for a single element x_{ijk}. Extending this to the whole matrix X, we require element-wise multiplication between ∇z_{abc} and y_{ijk}^{-1}.

Thus, $\nabla X = \nabla Z / Y$, so the element-wise division of ∇Z by Y. (Without Z: $\nabla X = \nabla(X/Y)/Y$)

Derivative of the loss w.r.t. Y:

$$\nabla y_{ijk} = \sum_{abc} \nabla z_{abc} \frac{\partial x_{abc} y_{abc}^{-1}}{\partial y_{ijk}}, \text{ whereby } \frac{\partial x_{abc} y_{abc}^{-1}}{\partial y_{ijk}} = \begin{cases} \frac{\partial x_{ijk} y_{ijk}^{-1}}{\partial y_{ijk}}, & \text{iff } a, b, c = i, j, k \\ 0 & \text{otherwise.} \end{cases},$$

with a, b, c again denoting the element's positions in a three-dimensional tensors X and Y, and i, j, k denoting the position of the element x in X with respect to which we compute the derivative of the loss for.

This gives: $\nabla y_{ijk} = \nabla z_{ijk} \frac{\partial x_{ijk} y_{ijk}^{-1}}{\partial y_{ijk}}$, whereby $\frac{\partial x_{ijk} y_{ijk}^{-1}}{\partial x_{ijk}} = -x_{ijk} y_{ijk}^{-2} = -\frac{x_{ijk}}{y_{ijk}^2}$, so $\nabla y_{ijk} = -\nabla z_{ijk} \frac{x_{ijk}}{y_{ijk}^2}$ for a single element y_{ijk}. Extending this to the whole matrix X, we require element-wise multiplication between ∇z_{ijk} and $-\frac{x_{ijk}}{y_{ijk}^2}$.

Thus, $\nabla Y = -\nabla Z \odot \ (X/Y^2) = -\nabla(X/Y) \odot \ (X/Y^2)$, with $\odot$ representing element-wise multiplication.

Backward for element-wise scalar-to-scalar functions (Question 2)

The input to a tensor-to-tensor function $F(X)$ (that applies the scalar-to-scalar function $f(x)$ to each element of the input) is the tensor X, and the output can be denoted as Y. Given the gradient of the input ∇Y, the backward of the formula $F(X)$ aims to obtain ∇X, or, differently expressed, $\frac{\partial l}{\partial X}$, which is equal to $\frac{\partial l}{\partial Y}\frac{\partial Y}{\partial X}$. As the function $F(X)$ applies $f(x)$ to each element of its input X, each element of Y can be written as $y_{ijk} = f(x_{ijk})$. When we work out the gradient ∇X for one single element x_{ijk} of X, we can write $\frac{\partial l}{\partial Y}\frac{\partial Y}{\partial x_{ijk}} = \sum_{abc} \frac{\partial l}{\partial y_{abc}} \frac{\partial y_{abc}}{\partial x_{ijk}} = \sum_{abc} \frac{\partial l}{\partial y_{abc}} \frac{\partial f(x_{abc})}{\partial x_{ijk}}$.

It is important to note that $\frac{\partial l}{\partial y_{abc}} \frac{\partial f(x_{abc})}{\partial x_{ijk}} = \begin{cases} \frac{\partial l}{\partial y_{ijk}} f'(x_{ijk}) & \text{iff } a,b,c = i,j,k \\ 0 & \text{otherwise} \end{cases}$, which leads to:

$$\frac{\partial l}{\partial x_{ijk}} = \sum_{abc} \frac{\partial l}{\partial y_{abc}} \frac{\partial y_{abc}}{\partial x_{ijk}} = \frac{\partial l}{\partial y_{ijk}} f'(x_{ijk})$$

Extending this to each element of the tensor X, we obtain $\nabla X = \nabla Y \odot F'(X)$, whereby $\odot$ denotes element-wise multiplication and $F'(X)$ denotes a tensor-to-tensor function that applies the scalar-to-scalar function $f'(x)$ (i.e., the derivative of the scalar-to-scalar function $f(x)$) to each element of its input.

Backward for linear layers (Question 3)

The matrix operation needed to compute the layer output from input matrix X and weight matrix W is matrix multiplication between X (dimensions $n * f$) and W (dimensions $f * m$), resulting in an output matrix Y of dimensions $n * m$. Thus: $F(X) = XW = Y$.

For a single element in Y, y_{ab}, we compute it's value by taking the dot-product of a column of X and and a row of W: $y_{nm} = \sum_f x_{nf} w_{fm}$. With this knowledge we can derive the backward:

Derivative of the loss w.r.t. X:

First, we work out the backward function in terms of single matrix elements:

$$\nabla x_{nf} = \sum_{ab} \nabla y_{ab} \frac{\partial y_{ab}}{\partial x_{nf}} = \sum_{ab} \nabla y_{ab} \frac{\partial \sum_f x_{af} w_{fb}}{\partial x_{nf}}$$

$$\text{whereby } \frac{\partial \sum_f x_{af} w_{fb}}{\partial x_{nf}} = \begin{cases} w_{fb} & \text{iff } a = n \\ 0 & \text{otherwise} \end{cases}$$

Thus,

$$\nabla x_{nf} = \sum_b \nabla y_{nb} w_{fb} = \nabla y_n w_f^T$$

Extending this to the whole matrix X, we get:

$$\nabla X = \nabla Y W^T$$

Derivative of the loss w.r.t. W:

First in terms of single matrix elements:

$$\nabla w_{fm} = \sum_{ab} \nabla y_{ab} \frac{\partial y_{ab}}{\partial w_{fm}} = \sum_{ab} \nabla y_{ab} \frac{\partial \sum_f x_{af} w_{fb}}{\partial f_{fm}}$$

$$\text{whereby } \frac{\partial \sum_f x_{af} w_{fb}}{\partial x_{fm}} = \begin{cases} x_{af} & \text{iff } b = m \\ 0 & \text{otherwise} \end{cases}$$

Thus,

$$\nabla w_{fm} = \sum_a \nabla y_{am} x_{af} = x_f^T \nabla y_m$$

Extending this to the whole matrix X, we get:
$\nabla W = X^T \nabla Y$

Backward for expanding a vector to a matrix (column-wise) (Question 4)

First in terms of single matrix elements:

$$\nabla x_n = \sum_m \nabla y_{nm} \frac{\partial y_{nm}}{\partial x_n},$$

whereby $y_{nm} = x_n$, as x_n was duplicated over all $m(= 16 in our example)$ columns.
Thus,

$$\nabla x_n = \sum_m \nabla y_{nm} \frac{\partial x_n}{\partial x_n} = \sum_m \nabla y_{nm}$$

and in matrix form:

$$\nabla x = \sum_m \nabla Y_m,$$

so the sum over the $m(= 16)$ columns of ∇Y.

VUGRAD implementation

In this section, we explore how automatic differentiation can be applied in practice using the differentiation mini-library *vugrad*.

Inspection of Tensor- and OpNodes (Question 5)

We first create two TensorNodes a and b, and add them to generate a third TensorNode c.

```
import numpy as np
import vugrad as vg

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

c = vg.Add.do_forward(a, b)
```

Then, we inspect what values the single Nodes store:

c.value refers to the raw value of the TensorNode c, which is a numpy array containing the output of the addition of TensorNode a and TensorNode b.

c.source refers to the source of TensorNode c, which is the OpNode by which it was created. Importantly, source doesn't refer to the Operation (Op), but the OpNode in the Computation Graph.

c.source.inputs[0] refers to the first input to the addition operation, i.e., TensorNode a, as indicated by the 0 index. An index of 1 would give the second input to the Addition, i.e., TensorNode b.

a.grad refers to a tensor (np.array) currently filled with zeros, with a shape equal to the shape of the raw values (*a.value*). In this tensor the gradient of TensorNode a will be stored. For multiple backward passes, the gradients for every backward pass will be added to that tensor, practically executing the Multivariate Chain Rule.

Understanding TensorNodes and OpNodes (Question 6)

The following questions test the understanding of how TensorNodes and OpNodes are implemented:

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?

Answer: The operation is defined by an object of the Class Op, which represents the actual operation to be performed.

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?

Answer: The actual addition is performed in line 324 "return a + b" in the static function *forward* the Add class, which is inheriting from the Op class. The static function *forward* is called within the $do\_forward$ function of the Op class.

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: An *OpNode* is created every time a $do\_forward$ function of an *Op* class is called. As the *OpNode* does not refer to a specific Operation, we don't know of what shape the Output will be. As every output (just like every input) of an operation must be a *TensorNode* that needs some source to keep track of by which *OpNode* it was created, we first create a *TensorNode* for each raw Op output with a pointer to the *OpNode* as the source. Only then, after building the *TensorNode* for the outputs, we can assign the output (the newly instantiated *TensorNode*) to the *OpNode*. Thus, the *OpNode* has to be instantiated before the output is defined and brought in the right format, and is thus instantiated as None first. This order is also evident in the code within the $do_forward$ function in the *Op* class:

The *OpNode* is connected to the outputs in line 249 (line 3 in the code displayed below), just after a *TensorNode* was instantiated for every Operation output (line 2 below), with a previously instantiated new *OpNode* (line 1 below) as the source.

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

How gradients are propagated through the network (Question 7)

The following paragraph aims to explain how the whole backpropagation algorithm is triggered in vugrad, starting from the TensorNode that contains the loss, to the bottom-most TensorNodes denoting the weights in the first layer.

If either the TensorNode for which the backward() function is called is the last Node in the graph (the TensorNode then only contains a scalar value, which is the loss) or the backward function has been called for each of the parent Nodes (kept track of by self.visits), the backward function of that TensorNodes source (the OpNode by which it was generated) gets called. In Code (Lines 95 - 98):

```
if self.visits == self.numparents or start:
            if self.source is not None:
                self.source.backward()
```

The *backward* function in the OpNode class then triggers the backward function of the associated Op, that computes the gradient for each input TensorNode to the OpNode given the gradients of the output TensorNodes (which is a tensor filled with ones for the last TensorNode). In Code (lines 155 - 159):

```
# extract the gradients over the outputs (these have been computed already)
goutputs_raw = [output.grad for output in self.outputs]

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

The *backward*() function of the relevant Ops gets called in the last line of the code snippet shown above (line 159).

Proof of concept: the Normalize function in *vugrad* (Question 8)

I chose to investigate whether the Normalize function is implemented correctly in *vugrad*, that normalizes the input over row-wise: $f(x_{ij}) = y_{ij} = \frac{x_{ij}}{\sum_j x_{ij}}$

First in terms of single matrix elements:

$$\nabla x_{ij} = \sum_{ab} \nabla y_{ab} \frac{\partial y_{ab}}{\partial x_{ij}} = \sum_{ab} \nabla y_{ab} \frac{\partial}{\partial x_{ij}} \frac{x_{ab}}{\sum_j x_{ab}}$$

$$\text{whereby } \frac{\partial}{\partial x_{ij}} \frac{x_{ab}}{\sum_j x_{ab}} = \begin{cases} 0 & \text{if } a, b \neq i, j \\ -\frac{x_{ij}}{(\sum_j x_{ij})^2} & \text{if } a = i, b \neq j \\ -\frac{\sum_j x_{ij} - x_{ij}}{(\sum_j x_{ij})^2} & \text{if } a, b = i, j \end{cases}$$

Thus,

$$\nabla x_{ij} = -\sum_{b \neq j} \nabla y_{ib} \frac{x_{ij}}{(\sum_j x_{ij})^2} + \nabla y_{ij} \frac{\sum_j x_{ij} - x_{ij}}{(\sum_j x_{ij})^2}$$

$$= -\sum_{b \neq j} \nabla y_{ib} \frac{x_{ij}}{(\sum_j x_{ij})^2} - \nabla y_{ij} \frac{x_{ij}}{(\sum_j x_{ij})^2} + \nabla y_{ij} \frac{1}{(\sum_j x_{ij})^2}$$

$$= -\sum_{j} \nabla y_{ij} \frac{x_{ij}}{(\sum_j x_{ij})^2} + \nabla y_{ij} \frac{1}{\sum_j x_{ij}}$$

and in matrix form:

$$\nabla X = \nabla Y / \sum_j X - \sum_j \nabla Y \odot X / (\sum_j X)^2,$$

where "/" denotes element-wise division and $\odot$ denots element-wise multiplication. It is important to note that the notation $\sum_j X$ denotes a matrix of the same size as X, but where the elements in each are identical and identical to the row-wise sums in X (the same applies to $\sum_j \nabla Y \odot X / (\sum_j X)^2$, where the elements in each row are identical and identical to the row-wise sums in $\nabla Y \odot X / (\sum_j X)^2$). The formula derived above is implemented 1:1 in *vugrad*:

```
(go / sumd) - ((go * x)/(sumd * sumd)).sum(axis=1, keepdims=True)
```

whereby **go** corresponds to ∇Y, **sumd** corresponds to $\sum_j X$ and **x** corresponds to X. The first term $\nabla Y / \sum_j X$ thus corresponds to **(go / sumd)** and the second term $\sum_j \nabla Y \odot X / (\sum_j X)^2$ corresponds to **((go * x)/(sumd * sumd)).sum(axis=1, keepdims=True)**.

Thus, the derived backward of the Normalize function and the implementation in *vugrad* are identical.

Extending *vugrad* by the ReLU activation function (Question 9)

To test how different activation functions influence the learning behavior of a network, we compared a simple neural network using once the default Sigmoid activation function, and once a self-built Op that implements the ReLU, or rectified linear unit, which applies the function $relu(x) = max(x, 0)$ to the elements of a given input x.

For the comparison of the Sigmoid and the ReLU activation function, I chose to classify the MNIST dataset with the same architecture as provided in the train_mlp.py file. The only difference was the activation function used, which I defined as:

```
class ReLU(Op):
    """
    ReLU activation function
    """

    @staticmethod
    def forward(context, input):
        context['relu_input'] = input
        return np.where(input > 0., input,0.)

    @staticmethod
    def backward(context, goutput):
        input = context['relu_input']
        return goutput * np.where(input > 0.,1.,0.)
```

Using the sigmoid activation function, the model attained a validation accuracy of **0.969** with a loss of **0.0444** after 10 epochs. In contrast, the validation accuracy using the ReLU was slightly lower with **0.9552** and a loss of **0.1758**.

Experimenting with network architecture in *vugrad* (Question 10)

To get an understanding of the impact of different network architectures on the classification accuracy obtained, I tested the effect of only an additional linear layer with a Sigmoid activation function and the effect of an additional residual connection between the first and the third hidden layer. The residual layer was implemented like this, within the definition of MLP module, as in the train_mlp.py file:

```
hidden_1 = self.layer1(input)

# non-linearity
hidden_1 = vg.Sigmoid.do_forward(hidden_1)
# -- We've called a utility function here, to mimin how this is usually done in pytorch. We could also
#    hidden = Sigmoid.do_forward(hidden)

# second layer
hidden_2 = self.layer2(hidden_1)

#non-linearity and adding residual connection
if self.residual:
    hidden_2 = vg.Sigmoid.do_forward(hidden_2) + hidden_1
```

```
else:
    hidden_2 = vg.Sigmoid.do_forward(hidden_2)

# third layer
output = self.layer3(hidden_2)
```

I trained both networks on the MNIST set for 10 epochs and compared their accuracies:

The residual network attained an accuracy of **0.97** with a loss of **0.03857** after the 10th epoch, while the network with only an additional linear layer showed a slightly lower accuracy **0.9698** and higher loss **0.3881** after 10 epochs. Thus, adding a third linear layer increased performance compared to a two-layer network (see Question 9), and adding a residual connection between the first and the third linear layer further improves performance slightly.

Building and improving an image classifier with *pytorch* (Question 11)

After having grasped how automatic differentiation in packages like *vugrad* or *pytorch* works, I implemented a classifier for the CIFAR-10 dataset, following the model architecture presented in the 60-minute blitz tutorial from *pytorch*. In an attempt to maximize the classification accuracy on unseen data, I tested different values for the learning rate (0.0005, 0.001, 0.01) and the momentum in SGD (0.9, 0.95) when training the model for 10 epochs and with the otherwise same architecture. A learning rate of 0.001 and a momentum of 0.9 proved to provide the best classification accuracy on both the training data (0.74688) and the test data (0.6327). Figure 1 depicts how the loss developed for the different parameter combinations.

As evident, the parameter combination that yielded the highest loss on the training and validation set also exhibits the lowest loss over training steps (green line).

Experimenting with network architectures (Question 12)

In an attempt to understand the impact of different network architecture choices, I augmented the network provided in the *pytorch* tutorial (2 convolutional layers and 3 densely connected layers) by adding a third convolutional layer. Addtionally, I altered several model parameters, as outlined below:

Until now, we only investigated models optimized with stochastic gradient descent. However, the Adam optimizer presents a highly promising alternative (Kingma and Ba (2014)), that was shown to outperform conventional SGD optimizers in many cases (Keskar and Socher (2017)). Thus, in order to investigate the impact of using the Adam over the SGD optimizer, I trained a new model using both the Adam and SGD separately.

Additionally, dropout provides a means of regularizing a neural network, such that it does not overfit on the training data, which results in subsequent low classification performance on new and unseen data. In

Dropout, several elements of an input tensor are zeroed, such that the network cannot use the information at these elements (or neurons) for classification (Srivastava et al. (2014)). Which elements are zeroed is decided randomly, but the proportion of which elements are zeroed can be defined by the user. To further also investigate the impact of dropout, the new model is tested with different proportions of neurons zeroed, once 10% and once 30%, between the first and second dense layer (after the convolutional layers) of the network.

Lastly, I also wanted to investigate the impact of adding a residual connection over convolutional layers. Thus, the model is also tested with a residual connection between the second convolutional layer and the first densely connected layer, and once without.

Figure 2 shows the results of the new model, tested with the different setups as outlined above, trained for 10 epochs.

As evident, the model with a residual connection and a dropout of 10%, optimized by the SGD algorithm yielded the highest classification performance of 0.7477 for the training set and 0.632 for the test set. Thus, the model slightly improved the classification performance compared to the best tuned basic model (see previous section). The models optimized by the Adam algorithm exhibited a faster decrease in loss, but ultimately did not attain the loss and accuracy as with SGD optimized models.

Interestingly, adding Batch Normalization Layers between the fully connected layers lead to a strong decrease in training accuracy, resulting in an almost equal training (0.55002) and validation accuracy (0.5216). However, due to the considerably lower accuracy level, testing with batch normalization was not continued.

Conclusions

In the report at hand, the principle and the mathematical intuition behind automatic differentiation was explained, along with practical implementation using the mini-library *vugrad* and the commonly used *pytorch* package. In both experiments assessing the model performance when (1) classifying the MNIST dataset using the *vugrad* package and (2) classfying the CIFAR-10 image dataset using *pytorch*, adding residual layers slightly improved the performance of deep neural networks. Thus, transmitting information through models over multiple pathways seems to provide advantages over linear information transmission.

References

Keskar, N. S. and Socher, R. (2017). Improving generalization performance by switching from adam to sgd. *arXiv preprint arXiv:1712.07628*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.