Deep Learning 2020 **(Due: 18 nov 2020)**

Submission Assignment #2

Instructor: Jakub Tomczak *Name:* Mick IJzer, *Netid:* 2552611

For automatic differentiation the backwards of functions and operations are necessary. The backward of a function computes the gradient of the loss with respect to the inputs, given the gradient of the loss with respect to the output. In general when computing the backward the chain rule is used (see equation 0.1), where $\frac{\delta loss}{\delta f}$ is the gradient of the loss with respect to the output (i.e. the downstream gradient) and where $\frac{\delta f}{\delta X}$ is the gradient of the output with respect to the input.

$$\frac{\delta loss}{\delta X} = \frac{\delta loss}{\delta f} * \frac{\delta f}{\delta X} \tag{0.1}$$

1 Question 1

To clarify the process of finding the backward of an element wise division $f(X,Y) = \frac{X}{Y}$, first the backward of a single element of $f(X,Y)$ is studied. For notation purposes let $G = \frac{\delta loss}{\delta f}$ (from equation 0.1). The resulting backward with respect to X_{ij} and Y_{ij} is given by equations 1.1 and 1.2. These two functions can easily be generalized to the complete X and Y matrices, since $f(X,Y)$ is an element wise operator. The complete backward function with respect to X and Y are $\frac{G}{Y}$ and $\frac{G*X}{Y^2}$ respectively.

$$\frac{\delta loss}{\delta X_{ij}} = G_{ij} * (\frac{\delta f}{\delta X})_{ij} = G_{ij} * \frac{\delta \frac{X_{ij}}{Y_{IJ}}}{\delta X_{ij}} = G_{ij} * \frac{1}{Y_{ij}} \tag{1.1}$$

$$\frac{\delta loss}{\delta Y_{ij}} = G_{ij} * (\frac{\delta f}{\delta Y})_{ij} = G_{ij} * \frac{\delta \frac{X_{ij}}{Y_{IJ}}}{\delta Y_{ij}} = G_{ij} * \frac{X_{ij}}{Y_{ij}^2} \tag{1.2}$$

2 Question 2

Equation 0.1 will also be used for this question. However, $f(X)$ is a function that is element wise applied to a vector X (e.g. X^2 or $log(X)$). Again for notation purposes let G be the downstream gradient given by $G = \frac{\delta loss}{\delta f}$. The backward for a single unit in X_i is given in equation 2.1. The backward with respect to X can therefore be generalized to $G * f'(X)$, which is the element-wise application of f' applied to the elements of X (i.e. $f'(X)$, element wise multiplied by the downstream gradient (i.e. G).

$$\frac{\delta loss}{\delta X_i} = G_i * \frac{\delta f(X_i)}{\delta X_i} = G_i * f'(X_i) \tag{2.1}$$

3 Question 3

Let X be an n-by-f batch of n inputs with f features each and let W be the f-by-o weight matrix, with f features and o outputs. Then the layer outputs are computed by the matrix multiplication $f(X,W) = XW$, where $f(X,W)$ is a n-by-o matrix. The backward of matrix multiplication with respect to W will first be computed by looking at a single unit of W (see equation 3.1). Again, for notation purposes let G be the downstream gradient of $f(X,W)$ given by $G = \frac{\delta loss}{\delta f}$. When this is generalized to the complete matrix W, the result becomes X^TG an f-by-o matrix. The backward with respect to X is similar and shown in equation 3.2, which results in a n-by-f matrix with gradients.

$$\frac{\delta loss}{\delta W_{ij}} = G_{nj} * (\frac{\delta X_n W}{\delta W})_{ij} = G_{nj} * \frac{\delta \sum_{k=1}^{f} X_{ni}W_{ik}}{\delta W_{ij}} = X^T_{ni}G_{nj} \tag{3.1}$$

$$\frac{\delta loss}{\delta X_{ni}} = G_{nj} * (\frac{\delta XW}{\delta X})_{ni} = G_{nj} * \frac{\delta \sum_{k=1}^{f} X_{ni} W_{kj}}{\delta X_{ni}} = G_{nj} W_{ij}^T \tag{3.2}$$

4 Question 4

Let X be a vector of size m, and I be a vector of 16 ones (i.e. [1 1 ... 1 1]). Then $f(X) = X \otimes I$ is the outer product of X and I and results in a matrix consisting of 16 columns that are all equal to X. Again, for notation purposes let G be the downstream gradient given by $G = \frac{\delta loss}{\delta f}$. The backward for a single entry i of X is given in equation 4.1. For the complete vector X this can be generalized to the sum of the rows of G.

$$\frac{\delta loss}{\delta X_i} = \sum_j G_{ij} * \frac{\delta I_j X_i}{\delta X_i} = \sum_j G_{ij} * 1 \tag{4.1}$$

5 Question 5

Two TensorNodes, A and B, were created, both with shape (4, 2). Next, C was defined as $C = A + B$, which reflects an element wise summation of the two TensorNodes. The value property of C (i.e. C.value) contains the result of the element wise summation of A and B. C.source refers to the operation node that was created when defining C and C.source.inputs[0].value refers to the values of the first input, which in this case are the values of A. Lastly the gradient of A is stored in A.grad, which contains only zeros. This makes sense since the gradients are not yet computed, since no backward function has been called yet.

6 Question 6

The specific operation that defines an OpNode is an instance of the class Op, which on its own doesn't mean anything. A class for specific operations (a subclass of the class Op) is needed to actually do something with the OpNode. Some example subclasses are the classes Add, Multiply, and MatrixMultiply. In these subclasses the necessary forward and backward are defined.

The computation graph given in question 4 uses the Add class as the specific operation for the OpNode. Within the code the actual element wise summation of the inputs is done in line 287 (i.e. the forward function of the Add class).

So, after the TensorNodes are created an operation is called, the "do_forward"-function of the respective subclass of Op (e.g. Add at line 89) is also called. First, this function computes the outputs (line 204). Then it creates an OpNode instance, with the operation, context and inputs (line 209). It then creates new TensorNodes for each output and assigns these outputs to the "outputs" property of the OpNode instance (line 212). The "Outputs" property is not immediately set for the OpNode instance, because the output TensorNodes need an OpNode as source to be initialized.

7 Question 7

To start the backpropagation a TensorNode which contains the loss value (as a single scalar value) is needed. The "backward" function of this TensorNode can then be called to backpropagate the loss through the entire network. This is done in the lines 74 through 82. The first two if statements make sure that the TensorNode is the start of the backpropagation and contains a single scalar value. If those two requirements are satisfied the grads of the TensorNode are set to 1 (they were 0). Next, the backward function is called in line 82, which allows the loss to be propagated through the network. It is important that the grads of the loss TensorNode are set to 1, otherwise all grads will be multiplied with 0 which leads to all updates being 0.

8 Question 8

The non element wise operation that was chosen to check its implementation in the VuGrad framework was the Normalize function. The Normalize function normalizes the row-values of a matrix. Let X be a m-by-n matrix. The function of this operation for a single element of X is given in equation 8.1. For the computation of the backward see equation 8.3, again with G being a matrix containing the downstream gradients with respect to the output of $f(X)$. Also, let $\sum = \sum_{k=1}^{n} X_{ik}$. Note that the coefficient rule is used. When generalizing equation 8.3 to the complete matrix X one must be careful, because of the matrices G and X. Equation **??** shows the backward with respect to X, note that $G * X$ is the dot product of the two matrices. The computed backward alligns with the backward that was implemented in the VuGrad framework.

$$f(X_{ij}) = \frac{X_{ij}}{\sum_{k=1}^{n} X_{ik}} \tag{8.1}$$

$$\frac{\delta loss}{\delta X_{ij}} = G_{ij} * \frac{\delta \frac{X_{ij}}{\sum}}{\delta X_{ij}} = G_{ij} * \frac{(\sum * 1) - (X_{ij} * 1)}{(\sum)^2} = G_{ij} * (\frac{1}{\sum} - \frac{X_{ij}}{(\sum)^2}) \tag{8.2}$$

$$\frac{\delta loss}{\delta X} = G * \frac{\delta \frac{X}{\sum}}{\delta X} = G * \frac{(\sum * 1) - (X * 1)}{(\sum)^2} = \frac{G}{\sum} - \frac{G * X}{(\sum)^2} \tag{8.3}$$

9 Question 9

The VuGrad framework only has sigmoid activation implemented. However, more commonly used in hidden layers is ReLu activation. So, ReLu activation operation was implemented in the framework. The forward of the ReLu operation is simply 0 if the input is negative, and linear otherwise. Since the operation is linear for positive input, the backward is also simply 0 if the forward was negative and 1 if the forward was positive.

Next, the performance of two similar networks, with the only difference being the activation function, was evaluated. Figure 1 shows the accuracy on the validation set of the 'sigmoid' and 'ReLu' networks during training. The training was repeated 5 times to combat random effects of the initialisation of the weights. The standard deviation for the repetitions is shown in a filled area around the mean. In general, both networks have near perfect performance at the end of training. However, the 'ReLu' network seems to consistently reach the optimal performance faster than the 'sigmoid' network. This is probably due to the gradients being larger, which basically increases the learning speed.

10 Question 10

After completing the Pytorch 60-minute blitz a classifier for the CIFAR10 dataset has been created. To optimize the convolutional network hyperparameter tuning is needed. In the following paragraphs the effect of changing different hyperparameters is described. The basic convolutional network has two convolutional layers, maxpooling for both, and three fully connected layers with ReLu activation. The learning rate is 0.001, batch size of 4, 5 epochs, and an SGD optimizer. The accuracy of this model on the validation set was 62%. So the performance after hyperparameter tuning will be compared to this as a baseline. It is hard to draw a conclusion based on manual hyperparameter tuning, since the different parameters interact with eachother. For example, where increasing the batch size might lead to a lower performance, the performance might be boosted when the batch size is increased while simultaneously decreasing the learning rate.

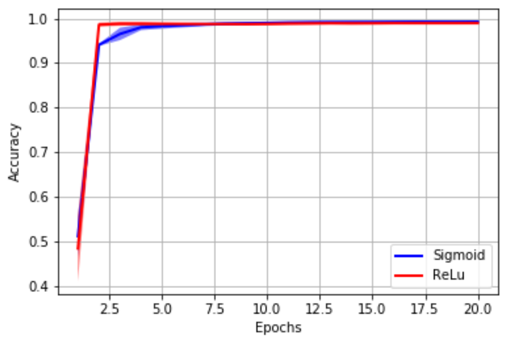

Figure 1: Validation Accuracy for Sigmoid and ReLu

Learning Rate A higher learning rate of 0.005 and a lower learning rate of 0.0005 were used to check the change in performance of the network. The accuracy of the model with learning rate of 0.0005 was similar to the baseline (61%). However, the model with the higher learning rate performed a lot worse (41%). Obviously this doesn't necessarily imply that the higher learning rate is worse, since the performance could be better depending on the settings for other hyperparameters.

Activation Function The activation function in the fully connected layer was changed from ReLu to Tanh and Sigmoid. The resulting accuracy of the model was similar (59%, 57% respectively). Therefore, the activation function doesn't seem to have a significant influence on the performance given the model architecture and other hyperparameters.

Batch Size For the batch size parameter several values were tried, namely 32, 8, and 2. The accuracy on the validation set was 54%, 60%, and 63% respectively. This means that given the other parameters, the batch size should be relatively small (i.e. between 2 and 8) for optimal performance. If a larger batch size would be used, the learning rate should probably be lower.

Convolutional Architecture Lastly, the architecture of the network was changed to study its effect on the performance. First, the number of nodes in the second hidden layer was increased from 120 to 275, which didn't result in an improvement (59%). Next, the kernel size of the two convolutional layers were increased from 6 and 16 to 20 and 25. This did increase the accuracy of the model on the validation set to 68%. The next logical step would be to add another convolutional layer to the model so the kernel sizes would in order be 15, 20, and 25. The performance was still similar to the previous network (67%). Lastly, two dropout layers of 20% were introduced to the network, which slightly decreased the performance to 65%.