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

Submission Assignment #2

Instructor: Jakub Tomczak *Name:* Asif Anwar, *Netid:* aar216

The following section layout is a suggestion for the structure of the report. If it doesn't quite fit, you can use whatever structure does work.

1 Question answers

Question 1 *Derivation of $\frac{X}{Y}$ where X and Y are two metrics in respect to X and Y are shown below.*

- $\frac{\delta}{\delta X}\delta(\frac{X}{Y}) = \sum_{kl} \frac{\delta(\frac{X_{kl}}{Y_{kl}})}{\delta X_{ij}} = \frac{1}{Y_{ij}}$
- $\frac{\delta}{\delta Y}\delta(\frac{X}{Y}) = \sum_{kl} \frac{\delta(\frac{X_{kl}}{Y_{kl}})}{\delta Y_{ij}} = -\frac{X_{ij}}{Y_{ij}^2}$

The backward of X and Y for element wise division will be as derived above $\frac{1}{Y_ij}$, $\frac{X_ij}{Y_{ij}^2}$

Note: *Considering backward function requires the comparison with loss and question 2 has the similar requirements, the full backward inrespect to loss is shown in question 2.*

Question 2 *Let's consider the function* f *has been applied to the given computational graph and* l *is the loss of the function. And its given that derivative of* f *if* f'. *If we calculate the derivatives we get below:*

- $X_{ij}^{\nabla} = \frac{\delta l}{\delta X_{ij}} = \sum \frac{\delta l}{\delta f} \frac{\delta f}{X_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{\delta(\frac{X_{kl}}{Y_{kl}})}{\delta X_{ij}} = f_{ij}^{\nabla} \sum_{kl} \frac{\delta(\frac{X_{kl}}{Y_{kl}})}{\delta X_{ij}} = f_{ij}^{\nabla} \frac{1}{Y_{ij}}$
- $Y_{ij}^{\nabla} = \frac{\delta l}{\delta Y_{ij}} = \sum \frac{\delta l}{\delta f} \frac{\delta f}{Y_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{\delta(\frac{X_{kl}}{Y_{kl}})}{\delta Y_{ij}} = f_{ij}^{\nabla} \sum_{kl} \frac{\delta(\frac{X_{kl}}{Y_{kl}})}{\delta Y_{ij}} = f_{ij}^{\nabla} \frac{X_{ij}}{Y_{ij}^2}$

From above equations we can see 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 *Given, **m**atrix W is the weights and matrix X is the batch input where n is number of instance and f is number features. For this we can do matrix dot product or matrix multiplication to get the* output *of the node. The derivation of this multiplication operation is shown below.*
Forward $W \cdot X^T$
Backward:

- $X^{\delta} = \frac{\delta l}{\delta X} = \sum \frac{\delta l}{\delta X_{nf}} = \sum \frac{\delta l}{\delta S_{nf}} \frac{\delta S_{nf}}{X_{nf}} = S^{\nabla} \sum_{kl} \frac{W.X}{X_{nf}} = S^{\nabla} \cdot W_{nf}$
- $W^{\delta} = \frac{\delta l}{\delta W} = \sum \frac{\delta l}{\delta W_{ij}} = S^{\nabla} \sum_{kl} \frac{W.X}{W_{nf}} = S^{\nabla} \cdot X_{nf}$

Question 4 *Given f(x) = Y where Y is 16 column x. S^{∇} is the derivative from function of the output.*

$$x^{\nabla} = S^{\nabla} \cdot \frac{\delta Y}{\delta x} = S^{\nabla} \frac{\delta[x_1, ..x_16]}{\delta x} = S^{\nabla}[1, ...1] = [S_1^{\nabla}, .., S_{16}^{\nabla}]$$

Question 5 *The outputs of TensorNode c are given below*

1. **c.value** contains the element wise sum of TensorNode a and b.
2. **c.source** refers to core.OpNode object at given address at memory.
3. **c.source.inputs[0].value** refers to the input value of first element 'a'.
4. **a.grad** refers to the gradient of a. The current value of which is [[0., 0.],[0.,0.]]

Question 6 *Explaining core.py*

1. OpNode operation is defines be 'op' objects
2. The addition happens inside the class Add(Op) at line number 287 in core.py file.
3. The OpNode output set to none because this will be calculated during forward function. The output of OpNode is assigned in the do_forward() function.

Question 7 *The backward function is called inside TensorNode backward() function which leads to OpNode backward() function. finally its been calcualsted in line 128, then based given Op then its get calculated in the respective Ops function*

Questin 8 *from the ops.py Normalize class is selected to prove that given method is workable for both forward and backward pass.*

```
class Normalize(Op):
    @staticmethod
    def forward(context, x):
        sumd = x.sum(axis=1, keepdims=True)
        context['x'], context['sumd'] = x, sumd
        return x / sumd
    @staticmethod
    def backward(context, go):
        x, sumd = context['x'], context['sumd']
        return (go / sumd) - ((go * x)/(sumd * sumd)).sum(axis=1, keepdims=True)
```

Forward *From the forward pass we get* $Y_{ij} = \frac{\sum x_i}{Y_{ij}}$

Backward *From below equation we see that derivative for backward function also matches. hence we can say that the implementation processes correct for operation classes*

$$x_{ij}^{\nabla} = \frac{\delta l}{\delta x_{ij}} = \sum_{kl} Y_{kl} \frac{\delta Y_{kl}}{\delta xij} = \frac{Y_{ij}^{\nabla}}{\sum x_i} - \sum \frac{Y_{ij}^{\nabla}}{\sum x_{ij}} (\sum_i x_i)^2$$

Questin 9 *Based on the given instructions The network was updated with relu function and validation compression was done. A detailed experiment was done on this section, which can be found in Next section of the report.*

Questin 10 *The classifier network has been created as per the tutorial. Beside changing the parameter, the new optimizer* ***adam*** *was introduced which gave a significant improvement of the network. Below are the different outcome of the classifier.*

a. *We can see that MLP performs mediocre in validation set for GSD optimizer55% overall performance*

b. *The overall performance doesn't change with adam optimizer with current scenario. In different run the performance change. In below data we can see the overall performance 52% which is worst than GSD*

c. *making the learning rate to 0.1 we can see that MLP only identify 10% of the validation, which means the network did not learn anything.*

d. *After running the network for 5 epochs we get the best result. 59% overall performance was achieved*

2 Problem statement

The impact on the performance of the network based on activation function change and adding new layers

- Change in performance with relu
- Change in performance with adding layer
- Change in performance with learning rate momentum

```
Accuracy of plane : 57 %
Accuracy of   car : 78 %
Accuracy of  bird : 51 %
Accuracy of   cat : 43 %
Accuracy of  deer : 60 %
Accuracy of   dog : 17 %
Accuracy of  frog : 63 %
Accuracy of horse : 56 %
Accuracy of  ship : 68 %
Accuracy of truck : 57 %
```

(a) SGD optimizer

```
Accuracy of plane : 49 %
Accuracy of   car : 53 %
Accuracy of  bird : 49 %
Accuracy of   cat : 35 %
Accuracy of  deer : 40 %
Accuracy of   dog : 34 %
Accuracy of  frog : 55 %
Accuracy of horse : 68 %
Accuracy of  ship : 75 %
Accuracy of truck : 67 %
```

(b) adam optimizer

```
Accuracy of plane :  0 %
Accuracy of   car :  0 %
Accuracy of  bird :  0 %
Accuracy of   cat :  0 %
Accuracy of  deer : 100 %
Accuracy of   dog :  0 %
Accuracy of  frog :  0 %
Accuracy of horse :  0 %
Accuracy of  ship :  0 %
Accuracy of truck :  0 %
```

(c) adam, lr=0.1

```
Accuracy of plane : 66 %
Accuracy of   car : 66 %
Accuracy of  bird : 39 %
Accuracy of   cat : 25 %
Accuracy of  deer : 52 %
Accuracy of   dog : 61 %
Accuracy of  frog : 68 %
Accuracy of horse : 67 %
Accuracy of  ship : 67 %
Accuracy of truck : 79 %
```

(d) adam, 5 epochs

Figure 1: Performance of MLP build using pytorch

3 Methodology

Overview: *To accomplish the target problem solution, different training mechanism was used. Initially the training will was done with 60000 synthetic dataset. with this training data the network was trained in different stages by changing different parameters and observe the performance impact on the network.*

Strategy *The training script was build based on the given train_mlp.py file. There are total four network was created to perform the testing. Initially the network was design with 2 layers and sigmoid as activation function of the hidden layer. Then new network with relu function was created. for this new network class MLP2 was created inside modules.py file and new class Relu was created in the ops.py file. Then for further testing a new MLP class was created with relu activation and three layers. Last but not least a simplified version of learning rate momentum was introduced by corelating the change with epochs*

- Running the two layer neural network using relu activation function for the hidden layer.
- Change the activation function of hidden layer to relu function from sigmoid activation.
- Adding new layer on top of rule layer
- Implementing momentum

4 Experiments

2 Layer Sigmoid *60000 intense of synthetic dataset was loaded and fed in the network. Total 20 epochs where used for the training where batch size was kept to 128 and the learning rate was 0.001*

2 Layer Relu *The next training was done on same dataset and parameters. However the new network of 2 layers was build where relu activation function was used to verify the performance.*

3 Layer Relu *Similar to above experiment all the parameters were kept same. However a new MLP was build with three layers where relu activation was used for hidden layer.*

Momentum *To experiment the performance of the network over momentum two layer network with relu activation was used. for this the learning rate was modified with the change of epochs*

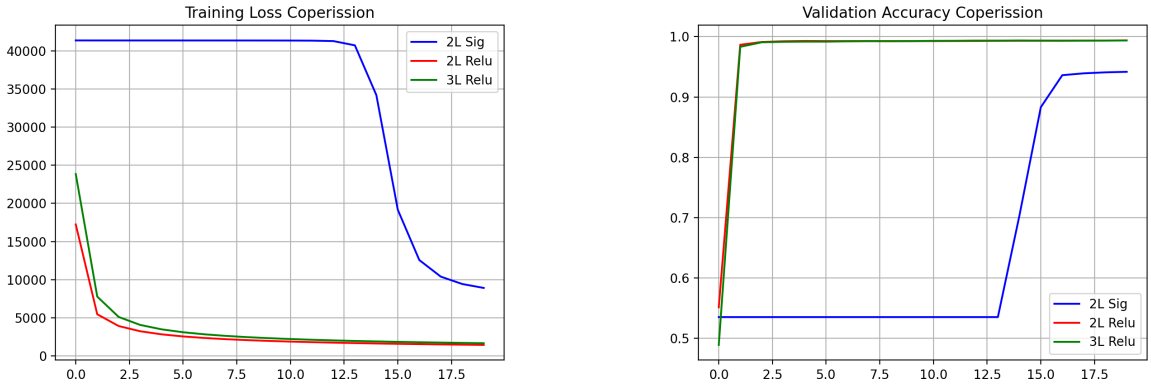
5 Results and discussion

2 Layer Sigmoid *2 layers MLP with Sigmoid function has been considered here as the benchmark. From the figure 1 we can notice that for long time there was no improvement in training loss or accuracy, however after 13 epochs we can the the rapid improvement in both training loss and validation accuracy. This implies that this network may fall in to local optimum and may take time to recover. Overall performance of the system is worst than relu function.*

2 Layer Relu *Adding the relu activation function function in the hidden layer improved both training loss and validation accuracy. we can observe the validation accuracy quickly reaches close to 98 percent*

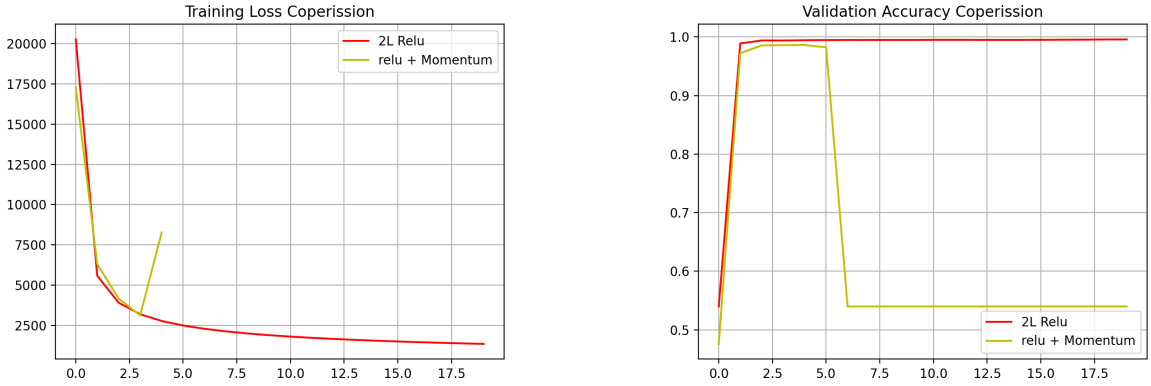
3 Layer Relu *Adding a new layer did not significantly improved the performance of the network from two layer relu activation. From the figure 1 we can see that it perform similar to 2 layer relu network. Considering the provided data is not significantly complex, we can conclude that 3 layers network is not required.*

Momentum *The performance of the momentum with respect to relu network is shown in the figure two. The simpler version of momentum change does not create any positive changes in performance. on the other hand we can see that network initially perform good similar to two layer relu then performance drastically drops.*



(a) Loss per epoch (b) Accuracy epochs

Figure 2: Loss and Accuracy over 20 epochs



(a) Loss per epoch (b) Accuracy epochs

Figure 3: Loss and Accuracy for Momentum

6 A code snippet

Relu Actiovation

```
class Relu(Op):
    @staticmethod
    def forward(context, input):
        relux = np.maximum(0, input)
        context['input'] = input
        return relux
    @staticmethod
    def backward(context, goutput):
        g_in = goutput.copy()
        g_in[context['input'] <=0 ] = 0
        return g_in
```

3 Layer MLP

```
class MLP3L(Module):
    def __init__(self, input_size, output_size, hidden_mult=2):
        super().__init__()
        hidden_size = hidden_mult * input_size
        self.layer1 = Linear(input_size, hidden_size)
        self.layer2 = Linear(hidden_size, hidden_size)
        self.layer3 = Linear(hidden_size, output_size)

    def forward(self, input):
        assert len(input.size()) == 2
        x = self.layer1(input)
        x = relu(x)
        x = self.layer2(x)
        x = relu(x)
        x = self.layer3(x)
        x = softmax(x)
        return x

    def parameters(self):
        return self.layer1.parameters() + self.layer2.parameters()
```

(Eiben et al., 2003).

References

Eiben, A. E., Smith, J. E., et al. (2003). *Introduction to evolutionary computing*, volume 53. Springer.

Appendix

NA