Course: Deep Learning **(14-11-2021)**

Submission Assignment 1

Coordinator: Jakub Tomczak *Name:* Venkat Mohit Sornapudi, *Netid:* 2721697

1 Question answers

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.*

We have: $f(X,Y) = X/Y$
So,

$$X_{ij}^{\nabla} = \frac{\partial l}{\partial X_{ij}} = \sum_{kl} \frac{\partial l}{\partial f_{kl}} \frac{\partial f_{kl}}{\partial X_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{\partial f_{kl}}{\partial X_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{\partial (X/Y)_{kl}}{\partial X_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{\partial (X_{kl}/Y_{kl})}{\partial X_{ij}} = \sum_{kl} f_{kl}^{\nabla} \left(\frac{1}{Y_{kl}}\right) \frac{\partial X_{kl}}{\partial X_{ij}}$$

$\frac{\partial X_{kl}}{\partial X_{ij}}$ is equal to 1 if and only if $k = i$ and $l = j$, else it is zero. Hence, the above expression can be simplified to:

$$X_{ij}^{\nabla} = \frac{f_{ij}^{\nabla}}{Y_{ij}} \Rightarrow X^{\nabla} = \frac{f^{\nabla}}{Y}$$

Similarly, we can workout the gradient of loss with respect to (w.r.t) Y as follows:

$$Y_{ij}^{\nabla} = \frac{\partial l}{\partial Y_{ij}} = \sum_{kl} \frac{\partial l}{\partial f_{kl}} \frac{\partial f_{kl}}{\partial Y_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{\partial f_{kl}}{\partial Y_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{\partial (X/Y)_{kl}}{\partial Y_{ij}} = \sum_{kl} f_{kl}^{\nabla} \frac{\partial (X_{kl}/Y_{kl})}{\partial Y_{ij}} = \sum_{kl} f_{kl}^{\nabla} X_{kl} \frac{\partial (1/Y_{kl})}{\partial Y_{ij}}$$

$\frac{\partial (1/Y_{kl})}{\partial Y_{ij}}$ is equal to $(-\frac{1}{Y_{ij}^2})$ if and only if $k = i$ and $l = j$, else it is zero. Hence, the above expression can be simplified to:

$$Y_{ij}^{\nabla} = f_{ij}^{\nabla} X_{ij} (-\frac{1}{Y_{ij}^2}) \Rightarrow Y^{\nabla} = -\frac{f^{\nabla} \otimes X}{Y^2}$$

Question 2 *Let f be a scalar-to-scalar function $f : \mathrm{R} \to \mathrm{R}$. 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.*

We have: $F(X) = \{f(X_{ij})\}$ i.e. matrix obtained after element-wise application of function f on matrix X.

To be shown: $X^{\nabla} = F^{\nabla} \otimes f'$

$$X_{ij}^{\nabla} = \frac{\partial l}{\partial X_{ij}} = \sum_{kl} \frac{\partial l}{\partial F_{kl}} \frac{\partial F_{kl}}{\partial X_{ij}} = \sum_{kl} F_{kl}^{\nabla} \frac{\partial (F(X))_{kl}}{\partial X_{ij}}$$

Using the definition of $F(X)$, we get:

$$X_{ij}^{\nabla} = \sum_{kl} F_{kl}^{\nabla} \frac{\partial f(X_{kl})}{\partial X_{ij}} = \sum_{kl} F_{kl}^{\nabla} f'_{kl} \frac{\partial X_{kl}}{\partial X_{ij}}$$

$\frac{\partial X_{kl}}{\partial X_{ij}}$ is equal to 1 if and only if $k = i$ and $l = j$, else it is zero. Hence, the above expression can be simplified to:

$$X_{ij}^{\nabla} = F_{ij}^{\nabla} f'_{ij} \Rightarrow X^{\nabla} = F^{\nabla} \otimes f'$$

**Note that, in the above derived tensor form gradient expressions, the division is done element-wise and $\otimes$ stands for element-wise multiplication.

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 non-linearity, 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.*

Assuming o represents the output layer, we have: $o_{ki} = \sum_{z=1}^{f} x_{kz} w_{iz}$

$$w_{ij}^{\nabla} = \frac{\partial l}{\partial w_{ij}} = \sum_{k=1}^{n} \frac{\partial l}{\partial o_{ki}} \frac{\partial o_{ki}}{\partial w_{ij}}$$

Using the definition of o, we get:

$$w_{ij}^{\nabla} = \sum_{k=1}^{n} o_{ki}^{\nabla} \frac{\partial (\sum_{z=1}^{f} x_{kz} w_{iz})}{\partial w_{ij}} = \sum_{k=1}^{n} o_{ki}^{\nabla} \left(\sum_{z=1}^{f} x_{kz} \frac{\partial w_{iz}}{\partial w_{ij}} \right)$$

$\frac{\partial w_{iz}}{\partial w_{ij}}$ is equal to 1 if and only if $z = j$, else it is zero. Hence, the above expression can be simplified to:

$$w_{ij}^{\nabla} = \sum_{k=1}^{n} o_{ki}^{\nabla} x_{kj} \Rightarrow w^{\nabla} = o^{\nabla^T} x$$

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.*

$$x_i^{\nabla} = \frac{\partial l}{\partial x_i} = \sum_{kl} \frac{\partial l}{\partial Y_{kl}} \frac{\partial Y_{kl}}{\partial x_i} = \sum_{kl} Y_{kl}^{\nabla} \frac{\partial (f(x))_{kl}}{\partial x_i}$$

Using $f(x) = \{z_{ij}\}$ where $z_{ij} = x_i$, we get:

$$x_i^{\nabla} = \sum_{kl} Y_{kl}^{\nabla} \frac{\partial x_k}{\partial x_i}$$

$\frac{\partial x_k}{\partial x_i}$ is equal to 1 if and only if $k = i$, else it is zero. Hence, the above expression can be simplified to:

$$x_i^{\nabla} = \sum_{l} Y_{il}^{\nabla} = 16 Y_{i0}^{\nabla}$$

In other words, if gy is the gradient of loss w.r.t Y (in numpy) we can express backward of f as gy.sum(axis=1).

Question 5

5.1 *What does c.value contain?*

c.value contains numpy array obtained from adding tensor a (numpy array) and tensor b (numpy array).

5.2 *What does c.source refer to?*

It refers to the OpNode that produced c.

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

It refers to first input to the OpNode that produced c, which is a in this case.

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

It refers to the gradient of a scalar (generally loss) w.r.t a. Its value is an array of zeros of the same shape as a. This is because a backward propagation is not yet done over a. But, in the first place, to run backward propagation, it needs a scalar node at the end of the graph over which the gradient can be calculated. Here, there is no such a scalar node in the graph created.

Question 6

6.1 *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?*

OpNode is a particular instance of an op applied to some inputs. So, class op defines the operation.

6.2 *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?*

The actual addition is performed while returning value in froward function of Add class (at line 324 of core.py file) as shown below:

```
317 class Add(Op):
318     """
319     Op for element-wise matrix addition.
320     """
321     @staticmethod
322     def forward(context, a, b):
323         assert a.shape == b.shape, f'Arrays not the same sizes ({a.shape} {b.shape}).'
324         return a + b
325
```

6.3.1 *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?*

The OpNode's output is initialized as None. It is updated when the forward pass of the network. It is derived from the outputs of adjacent Tensornodes. This offers the flexibility of using the OpNode according to any neural network created.

6.3.2 *In which line is the OpNode connected to the output nodes?*

This happens at line 249 of the do_forward function of Op class as shown below:

```
245
246         opnode = OpNode(cls, context, inputs)
247
248         outputs = [TensorNode(value=output, source=opnode) for output in outputs_raw]
249         opnode.outputs = outputs
250
```

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?*

This happens at line 159 of the backward function of OpNode class as shown below:

```
154
155         # extract the gradients over the outputs (these have been computed already)
156         goutputs_raw = [output.grad for output in self.outputs]
157
158         # compute the gradients over the inputs
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.*

Derivation to prove the backward computation of Expand operation:

Let's consider x and Y to be a vector (singleton dimension) and the resultant matrix obtained by expanding the vector respectively. Let's also consider f to be the expanding function. So, $f(x) = Y$.

If we expand over dimension 1 i.e. the vector x repeated in all columns of Y:

$$x_i^{\nabla} = \frac{\partial l}{\partial x_i} = \sum_{kl} \frac{\partial l}{\partial Y_{kl}} \frac{\partial Y_{kl}}{\partial x_i} = \sum_{kl} Y_{kl}^{\nabla} \frac{\partial (f(x))_{kl}}{\partial x_i}$$

Using $f(x) = \{z_{ij}\}$ where $z_{ij} = x_i$, we get:

$$x_i^{\nabla} = \sum_{kl} Y_{kl}^{\nabla} \frac{\partial x_k}{\partial x_i}$$

$\frac{\partial x_k}{\partial x_i}$ is equal to 1 if and only if $k = i$, else it is zero. Hence, the above expression can be simplified to:

$$x_i^{\nabla} = \sum_{l} Y_{il}^{\nabla}$$

In other words, if gy is the gradient of loss w.r.t Y (in numpy) we can express backward of f as gy.sum(axis=1).

If we expand over dimension 0 i.e. the vector x repeated in all rows of Y:

$$x_j^{\nabla} = \frac{\partial l}{\partial x_j} = \sum_{kl} \frac{\partial l}{\partial Y_{kl}} \frac{\partial Y_{kl}}{\partial x_j} = \sum_{kl} Y_{kl}^{\nabla} \frac{\partial (f(x))_{kl}}{\partial x_j}$$

Using $f(x) = \{z_{ij}\}$ where $z_{ij} = x_j$, we get:

$$x_j^{\nabla} = \sum_{kl} Y_{kl}^{\nabla} \frac{\partial x_l}{\partial x_j}$$

$\frac{\partial x_l}{\partial x_j}$ is equal to 1 if and only if $l = j$, else it is zero. Hence, the above expression can be simplified to:

$$x_i^{\nabla} = \sum_{k} Y_{kj}^{\nabla}$$

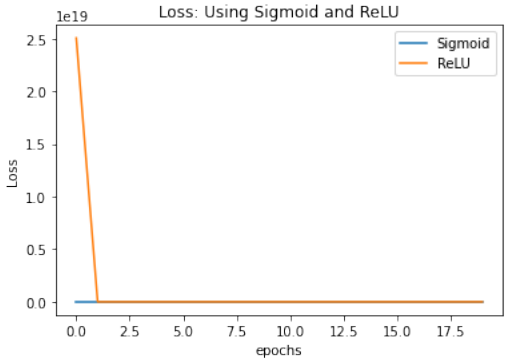
In other words, if gy is the gradient of loss w.r.t Y (in numpy) we can express backward of f as gy.sum(axis=0).

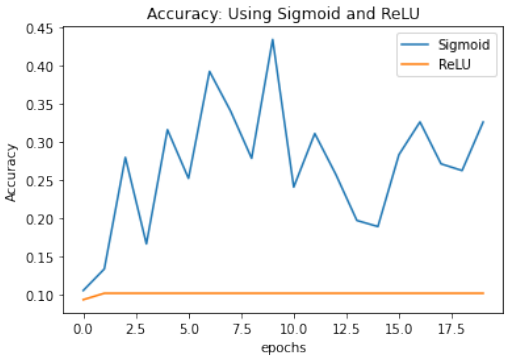
Question 9 *The current network uses a Sigmoid non-linearity on the hidden layer. Create an Op for a ReLU non-linearity (details in the last part of the lecture). Retrain the network. Compare the validation accuracy of the Sigmoid and the ReLU versions.*

By training and validating using Sigmoid and ReLU activation functions on MNIST dataset, the obtained results (at last epoch) are as follows:

- **Accuracy using Sigmoid:** 0.3256
- **Accuracy using ReLU:** 0.1014
- **Loss using Sigmoid:** 6111456
- **Loss using ReLU:** 126668

To compare the results using the two functions better, the following graphs are plotted:





Clearly, we can observe that the results are better when sigmoid function was used.

Question 10 *Change the network architecture (and other aspects of the model) and show how the training behavior changes.*

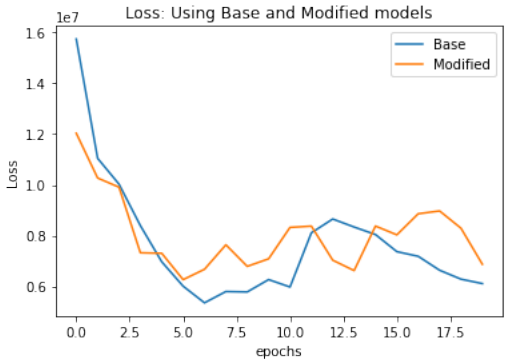
The following changes have been made to the base code:

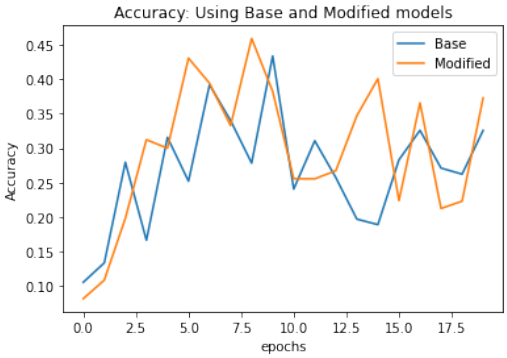
- **Widened hidden layer:** 4*number of features → 6*number of features
- **Changed initialization:** Xavier/Glorot initialization → He initialization

The results (at last epoch) after training and validating on MNIST dataset are as follows:

- **Accuracy using base code:** 0.3256
- **Accuracy using modified code:** 0.3728
- **Loss using base code:** 6111456
- **Loss using modified code:** 6873601

To compare the results using the two implementations better, the following graphs are plotted:





Question 11 *Install pytorch using the installation instructions on its main page. 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.*

The model has been trained using Cross entropy loss function, SGD optimizer (momentum=0.9) and batch size of 30 on CIFAR10 dataset. The varied hyperparameters for this experiment are:

- **learning rate:** [0.1, 0.01]
- **number of epochs:** [1, 3]

Structure of network (in same order):

- **Conv2d** (3, 6, 5)
- **MaxPool2d** (2, 2)
- **Conv2d** (6, 16, 5)
- **Linear** (16 * 5 * 5, 120)
- **Linear** (120, 84)
- **Linear** (84, 10)

Method: Various combinations of the above learning rates and number of epochs are taken. For each combination of hyperparameters the model has been trained and validated. The best combination is chosen by comparing the resultant validation accuracies.

Thus, the best combination of hyperparameters are found to be (learning rate= 0.1, number of epochs= 3).

Table 1: Question 12 — The resultant class-wise accuracies of the network on the 10000 test

Class	**Accuracy**
plane	0.63
car	0.69
bird	0.42
cat	0.21
deer	0.49
dog	0.67
frog	0.69
horse	0.62
ship	0.68
truck	0.53

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.*

The model has been trained using Multi-Margin loss function and Adam optimizer on CIFAR10 dataset. The chosen hyperparameters for this experiment are:

- **batch_size:** 4
- **learning rate:** 0.001
- **number of epochs:** 10

The structure of the network used is same as in Question 11.

The resultant accuracy of the network on the 10000 test images is 0.56. The following is the loss curve obtained during training: