Deep Learning 2020 **(Due: [November 18, 2020])**

Submission Assignment #[2]

Instructor: Jakub Tomczak *Name:* [Qinghe Gao], *Netid:* [qgo500]

1 Problem statement

In previous project, we implemented neural network from scratch and did a basic classification on simple dataset. However, python already provided many frameworks and it is quite handy to build complicated neural network. Thus, in this report we build AlexNet to train CIFAR 10 dataset. To get better classification, we tune hyperparameters such as learning rate, batch size, number of epochs and so on. Finally, we do final prediction on test dataset. **To note that, the first part is report of question 10 and the second part is for question 1-9. And for question 9 we solve like a report but we write like a question.**

2 Methodology

2.1 AlexNet

AlexNet is famous and classical deep convolutional neural network and it also has Epoch-making meaning for deep learning area. Figure 1 shows the main structure of AlexNet. Basically three types of layers are used in this network: convolutional layer, Pooling layer and fully connected layer. AlexNet has three distinguish features which helps model achieve excellent performance on image classification.

- **ReLU**. Using ReLU instead of Sigmoid/tanh. Since ReLU is linear and the derivative are always 1 or 0, the amount of calculation is greatly reduced, and the convergence speed will be much faster than Sigmoid/tanh.
- **Dropout or other methods**. These methods are mainly used to aviod overfitting. For example, In the neural network, Dropout is achieved by modifying the structure of the neural network itself. The main idea is randomly dropping out some neurons and these neuron does not participate in forward and backward propagation. In the next iteration, some neurons are randomly deleted again (set to 0) until the end of training.
- **multiple GPU.**

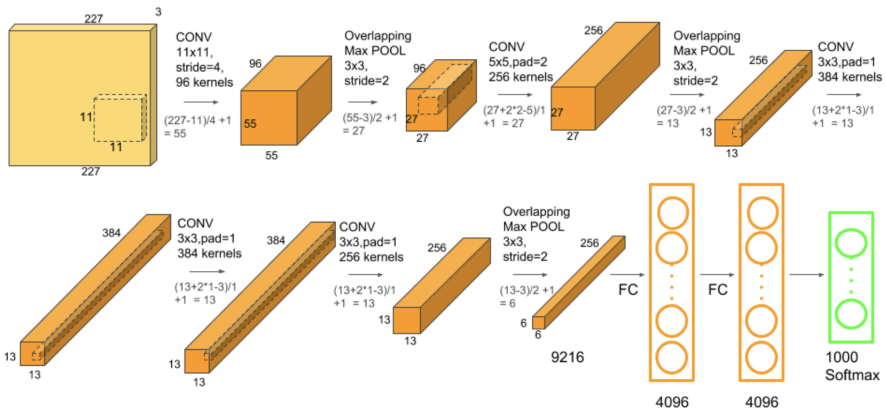

Figure 1: Structure of Alexnet, adapted from (Ale)

2.2 Batch normalization

Batch normalization is a method to make neural network faster and more stable by re-scaling the input. And it also can help the activation to capture more vital information. Furthermore, it can also increase the classification effect. One explanation is that this method is similar to Dropout to prevent overfitting, so it can be achieved without Dropout Considerable effect; In addition, the parameter adjustment process is much simpler, the initialization requirements are not so high, and a large learning rate can be used. (Bat)

2.3 Adam optimizer

Adam optimizer is one of gradient descent method, which is quite popular now in deep learning. It uses the first-order moment estimation and the second-order moment estimation of the gradient to dynamically adjust the learning rate of each parameter. The main advantage of Adam is that after bias correction, each iteration of the learning rate has a certain range, making the parameters relatively stable. (Ada)

3 Experiments

3.1 Dataset

In this project, we use CIFAR-10 dataset. This dataset consists of 60000 colour images in 10 classes and each image has the shape of 3*32*32. There are 50000 training images and 10000 test images. Figure 2 shows ten images of ten classes.

Figure 2: CIFAR-10 dataset. 10 images of each class.

3.2 Basic structure

The basic structure of this neural network:

Layer 1:[Conv2d, ReLU, MaxPool2d] →
Layer 2:[Conv2d, ReLU, MaxPool2d] →
Layer 3:[Conv2d, ReLU] →
Layer 4:[Conv2d, ReLU] →
Layer 5:[Conv2d, ReLU, MaxPool2d] →
Layer 6:[Drop, Fully-connected, ReLu] →
Layer 7:[Drop, Fully-connected, ReLu]→ **Layer 8:**[Fully-connected]→output.
Since there are so many parameters in each layer, we put detailed information the code snippet part.

3.3 Tuning parameters

Since this is quite large network and we only have limited computational power, we can only tune limited parameters. Thus, table 1 shows our logistics in this process. In every step we use previous best parameters to train current model. For example, in the first step, we use default parameters: Epoch:10, Learning rate: 0.001, Batch Size: 100, Optimizer: Adam, and Batch normalization: No/Yes. Thus, if we get better accuracy without batch normalization in step 1, in step 2 we will not use batch normalization to train the Optimizer.

Step	Parameters	Value	Step	Parameters	Value
1	Batch Norm	Yes/None	2	Optimizer	Adam/SGD
3	Batch size	[10, 100, 500]	4	Learning rate	[0.001,0.0001,0.00001]

Table 1: Step of tuning parameter

4 Results and discussion

We use **Alex.py** to do the model implementation. Furthermore, we save all the training and test accuracy as **.npy** files in result folder. And using **assignment2.ipython** to do visualization.

4.1 Batch normalization

Figure 3 shows the results of batch normalization. The results quite counter intuition because the model without batch normalization has better accuracy for test dataset. The possible explanation is that in the model we also use Dropout method. And we mentioned before that batch normalization is quite similar to Dropout method. Thus, if we use both method simultaneously, it may hinder the learning process of model. Thus, we decide to abandon batch normalization.

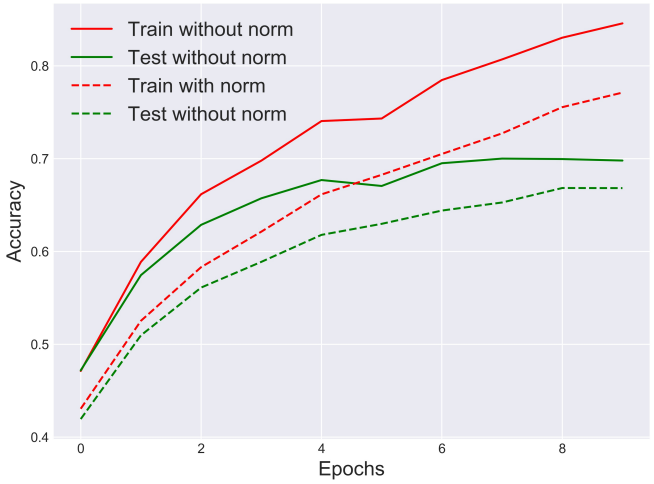

Figure 3: Train and validation accuracy with or without batch normalization. Solid lines are the results without batch normalization and dashed lines are the results with batch normalization

4.2 Optimizer

Figure 4 shows the results with Adam or SGD optimizer. It is clear to see that the results with Adam optimizer have way too better performance than SGD optimizer, which is quite intuitive since we have many instances in training dataset. And using one instance or batch to do gradient descent is quite random for large training dataset within 10 epochs. Thus, we should use Adam optimizer.

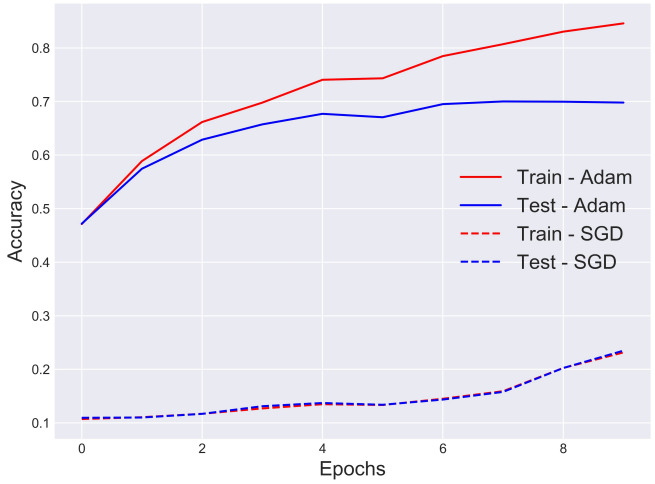

Figure 4: Train and validation accuracy with Adam or SGD optimizer. Solid lines are the results with Adam optimizer and dashed lines are the results with SGD optimizer.

4.3 Learning rate

Furthermore, we explore how different learning rates influence the accuracy. Figure 5 show small learning rate is difficult to converge within 10 epochs. Thus, we can see 0.00001 learning rate has both smaller training and test accuracy than large learning rate. However, 0.001 learning rate has similar performance as 0.0001 learning rate. Because both test accuracy are all nearly 70% after 9 epochs. 0.001 learning rate is chose to tune batch size because it has decent accuracy and can converge faster.

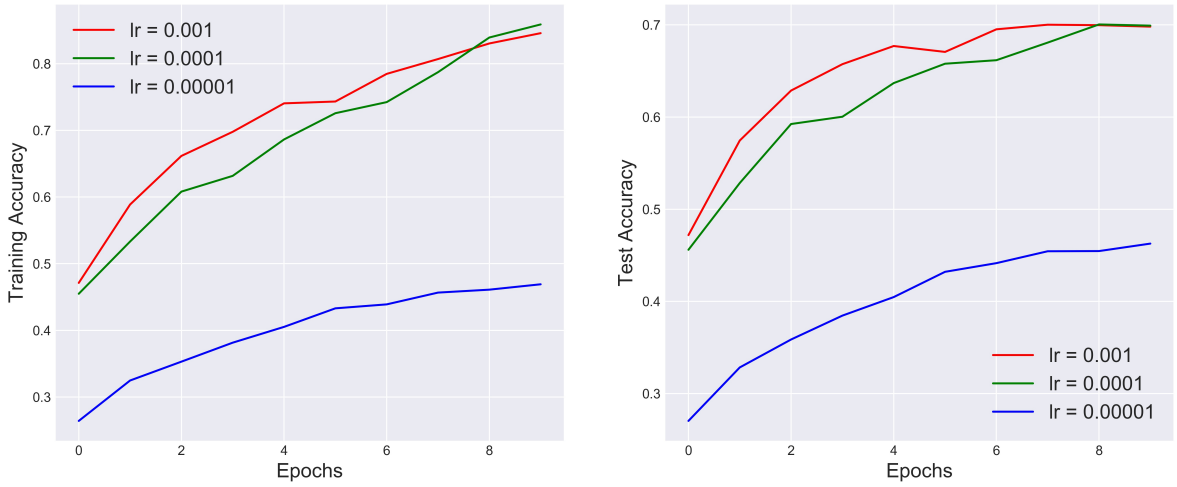

Figure 5: Train and validation accuracy with different learning rates. Learning rates are 0.001,0.0001 and 0.00001. The left plot is training accuracy and right plot is test accuracy.

4.4 Batch Size

Finally, we tune batch size and figure 6 shows the results. Within 10 epochs small batch size can not converge and the accuracy of test dataset are smaller than large batch size. Besides, 100 and 500 batch size can both achieve decent test accuracy(70%) and has overfitting after 8 epochs because accuracy of test dataset drops. Thus, we decide to choose 100 batch size.

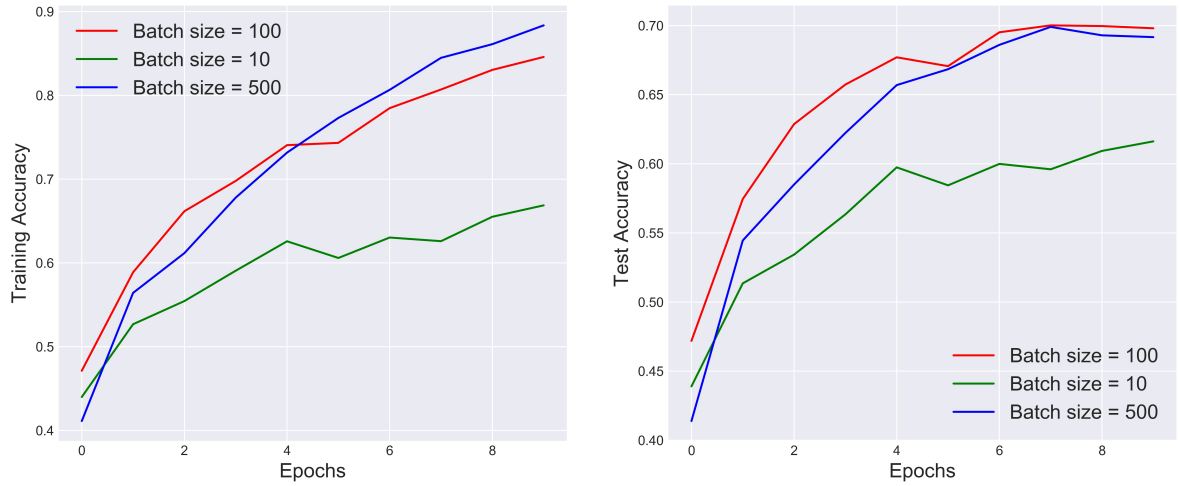

Figure 6: Train and validation accuracy with different batch size. Batch size are 10, 100 and 500. The left plot is training accuracy and right plot is test accuracy.

4.5 Final result

We also decided to try different number of epoch. However, according to the previous experiments models are already overfitting within 10 epochs. Thus, final parameters are Epoch:10, Learning rate: 0.001, Batch Size: 100, Optimizer: Adam, and Batch normalization: No. And the best accuracy is 70%.

5 Question Part

Question 1: $\bigotimes$ Means element-wise multiplication.

$$
\begin{aligned}
C = f(X,Y) &= \frac{X}{Y} \\
X_{ij}^{\triangledown} &= \sum_{kl} \frac{\partial l}{\partial C_{kl}} \frac{\partial C_{kl}}{\partial X_{ij}} = \sum_{kl} C_{kl}^{\triangledown} \frac{\partial C_{kl}}{\partial X_{ij}} \\
&= \sum_{kl} C_{kl}^{\triangledown} \frac{\partial [\frac{X}{Y}]_{kl}}{\partial X_{ij}} = \sum_{kl} C_{kl}^{\triangledown} \frac{1}{Y_{kl}} \frac{\partial X_{kl}}{\partial X_{ij}} \\
&= C_{ij}^{\triangledown} \frac{1}{Y_{ij}} \\
\text{Thus, } X^{\triangledown} &= C^{\triangledown} \bigotimes \frac{1}{Y} \\
\text{Same for,} Y^{\triangledown} &= -C^{\triangledown} \bigotimes \frac{X}{Y^2}
\end{aligned}
\tag{5.1}
$$

Question 2:

$$
\begin{aligned}
C = F(X) &= f(X) \\
X_{ij}^{\triangledown} &= \sum_{kl} \frac{\partial l}{\partial C_{kl}} \frac{\partial C_{kl}}{\partial X_{ij}} = \sum_{kl} C_{kl}^{\triangledown} \frac{\partial C_{kl}}{\partial X_{ij}} \\
&= \sum_{kl} C_{kl}^{\triangledown} \frac{\partial [f(X)]_{kl}}{\partial X_{ij}} \\
&= C_{ij}^{\triangledown} f^{'}(X)_{ij} \\
\text{Thus, } X^{\triangledown} &= C^{\triangledown} \bigotimes f^{'}(X)
\end{aligned}
\tag{5.2}
$$

Question 3: Matrix multiplication. X is (n,f), W is (f,m) and C (n,m)

$$
\begin{aligned}
C &= f(X) = XW \\
X^{\triangledown}_{nf} &= \sum_{nm} \frac{\partial l}{\partial C_{nm}} \frac{\partial C_{nm}}{\partial X_{nf}} = \sum_{nm} C^{\triangledown}_{nm} \frac{\partial C_{nm}}{\partial X_{nf}} \\
&= \sum_{nm} C^{\triangledown}_{nm} \frac{\partial [XW]_{nm}}{\partial X_{ij}} = \sum_{nm} C^{\triangledown}_{nm} \frac{\partial \sum_i X_{ni} W im}{\partial X_{nf}} \\
&= \sum_{nmi} C^{\triangledown}_{nm} \frac{\partial X_{ni} W_{im}}{\partial X_{nf}} = \sum_{nm} C^{\triangledown}_{nm} \frac{\partial X_{nf} W_{fm}}{\partial X_{nf}} \\
&= \sum_{nm} C^{\triangledown}_{nm} W^T_{fm} \\
\text{Thus, } X^{\triangledown} &= C^{\triangledown} W^T
\end{aligned}
\tag{5.3}
$$

Question 4: In this case j =16

$$
\begin{aligned}
C &= f(X) = X \\
X^{\triangledown}_{ij} &= \sum_{kl} \frac{\partial l}{\partial C_{kl}} \frac{\partial C_{kl}}{\partial X_{ij}} = \sum_{kl} C^{\triangledown}_{kl} \frac{\partial C_{kl}}{\partial X_{ij}} \\
&= \sum_{kl} C^{\triangledown}_{kl} \frac{\partial [X]_{kl}}{\partial X_{ij}} = \sum_{kl} C^{\triangledown}_{kl} \frac{\partial X_{kl}}{\partial X_{ij}} \\
&= C^{\triangledown}_{ij} \frac{\partial X_{ij}}{\partial X_{ij}} \\
\text{Thus, } X^{\triangledown} &= C^{\triangledown}
\end{aligned}
\tag{5.4}
$$

Question 5:

- c.value contains exact results of a+b(element-wise). Here is one example: [[-4.59966335, 3.74994202],[-1.61047585, 3.07620961]].
- c.source refers to operation node. In this case it means addition in diamond shape node.
- c.source.inputs[0].value refer to the exact value of the first input. In this case, it means value of a.
- a.grad refers to $\frac{\partial l}{\partial c}\frac{\partial c}{\partial a}$. The current value is [[0., 0.],[0., 0.]]

Question 6:

- In OpNode is self.op and In class Op is cls.
- outputs_raw = cls.forward(context, *inputs_raw, **kwargs)
- Because we already calculate the output which is outputs_raw and we can just link the result to self.output later, which is opnode.outputs = outputs

Question 7: if self.source is not None: self.source.backward()

Question 8: Sum operation is not element-wise. Because it needs to sum all over the possible value. To implement this for forward we need to save the shape of input then just sum it all to return a scalar value. For backward, equation 8 shows our result. And The final result is $C^{\triangledown}\mathbf{1}_{ij}$ means the backward is a matrix with shape of i by j which is the same as shape of X. And all the element in that matrix are $C^{\triangledown}$. This result is exact the same as implementation of the code.

$$
\begin{aligned}
C &= F(X) = \sum_{i,j} X_{ij} \\
X^{\triangledown}_{ij} &= \frac{\partial l}{\partial C} \frac{\partial C}{\partial X_{ij}} = C^{\triangledown} \frac{\partial C}{\partial X_{ij}} \\
&= C^{\triangledown} \frac{\partial \sum_{i,j} X_{ij}}{\partial X_{ij}} \\
&= C^{\triangledown} \\
\text{Thus, } X^{\triangledown} &= C^{\triangledown} \mathbf{1}_{ij}
\end{aligned}
\tag{5.5}
$$

Question 9: After careful exploration of code, we finally can discuss how the structure influences the neural network. Besides, we use synthetic dataset, 128 batchszie, and 20 epochs. And the basic structure is input → linear → non-linear activation → linear →softmax.

To do all the experiments, there are several changes in the code. Firstly, we change original **ops.py** file and add **ReLu** function in it. And the implementation of ReLu shows in code snippet. Besides, like sigmoid function we also add relu in **function.py**. And to run different setting, we implement our own module and train files which are **my_modules.py**(in vugard folder) and **my_train.py**(in experiment folder). Furthermore, we save all the training and test accuracy as **.npy** files in result folder. And using **assignment2.ipython** to do visualization.

Firstly, we want to discuss how the activation function influence the performance of model and we use Sigmoid and ReLu in this part. In order to evaluate the validation accuracy we want to fix all the parameters. And we find out that the learning rates significantly influence ReLu. Because large learning rate tends to make weight parameters negative, which means most of input of ReLu are negative and the output of ReLu becomes zero. In this way, we cannot calculate log function. To avoid this situation we use comparatively small learning rate which is 0.01.

Figure 7 shows the validation accuracy of Sigmoid and ReLu. It is clear to see that Sigmoid converges faster than ReLu and after 10 epochs accuracy becomes the same. Thus, we can conclude that there is no significant different influence on performance. But we want to discuss some parts which can not be showed by validation accuracy. ReLu can reduce the possibility of vanished gradient, which is brought by Sigmoid. And it is more computationally efficient to compute output or gradient. The disadvantage of ReLu is dying ReLu, which we discussed before that if too many output get below zero then most of the units(neurons) in network with Relu will simply output zero, in other words, die and thereby prohibiting learning.

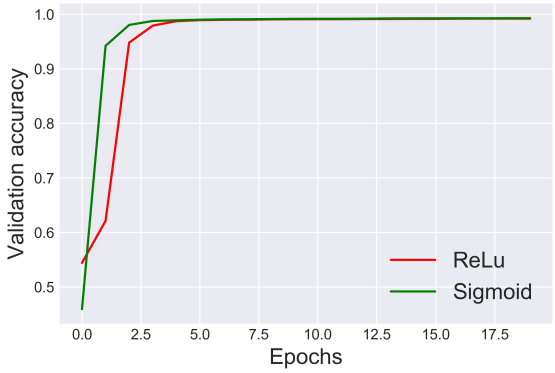

Figure 7: Validation accuracy of different activation function.

Furthermore, we want to elaborate the structure of model: input → linear → ReLu → linear → Sigmoid → Linear → softmax. And the implementation of new model shows in code snippet. Figure 8 shows validation accuracy of new model and old model(sigmoid). We can see that the new model show better accuracy than original model. Adding new layer indeed can get better performance.

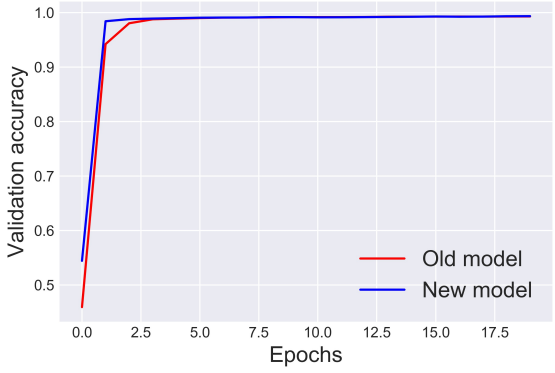

Figure 8: Validation accuracy of new model.

Additionally, we discuss different weight initialization. Since we use ReLu activation in the new model and

then we try He Initialization to compare with original Glorot initialization. He initialization can be implemented like this: **w = np.random.randn(output_size, input_size) /np.sqrt(output_size/ 2)**. Figure 9 shows that weight initialization has even better validation accuracy when compared with new model. And the best accuracy is even 99.64%. Thus weight initialization indeed has big influence on performance.

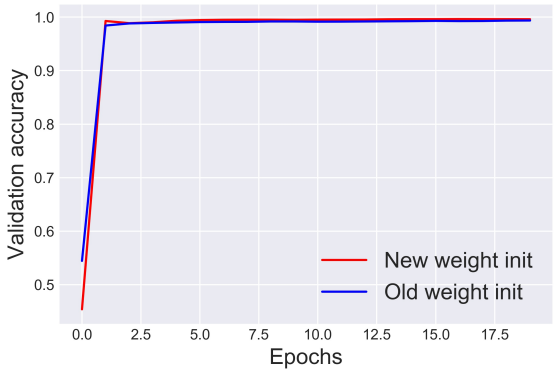

Figure 9: Validation accuracy of new weight initialization.

6 A code snippet

```
class ReLu(Op):
    """
    Op for element-wise application
    of ReLu function
    """
    @staticmethod
    def forward(context, input):
        relu = np.maximum(0,input)
        context['relu'] = relu

        # store the sigmoid of x
        for the backward pass

        return relu

    @staticmethod
    def backward(context, goutput):
        relu = context['relu']

        # retrieve the sigmoid of x
        dZ = np.array(goutput,
        copy=True)

        dZ[relu <= 0] = 0
        return dZ
```

```
### New model
def forward(self, input):

    assert len(input.size()) == 2

    # first layer
    hidden = self.layer1(input)

    # non-linearity
    hidden = relu(hidden)

    # second layer
    hidden = self.layer2(hidden)

    hidden = sigmoid(hidden)

    output = self.layer3(hidden)
    output = softmax(output)

    return output
```

```
class AlexNet(nn.Module):
    def __init__(self, num_classes=10):
        super(AlexNet, self).__init__()

        self.models = nn.Sequential(
            ##layer 1
            nn.Conv2d(in_channels=3,out_channels=64,kernel_size = 3,
            stride =2 ,padding =1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),

            ##layer 2
            nn.Conv2d(in_channels = 64, out_channels = 192,kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2),

            ##layer 3
            nn.Conv2d(in_channels=192, out_channels=384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            ##layer 4
            nn.Conv2d(in_channels=384, out_channels=256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),

            ##layer 5
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2)
        )
        self.classifier = nn.Sequential(
             ##layer 6
            nn.Dropout(),
            nn.Linear(256 * 2 * 2, 4096),
            nn.ReLU(inplace=True),

            ##layer 7
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),

            ##layer 8
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.models(x)
        x = x.view(x.size(0), 256 * 2 * 2)
        x = self.classifier(x)
        return x
```

References

Adam. https://blog.csdn.net/brucewong0516/article/details/78838124?utm_medium=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param&depth_1-utm_source=distribute.pc_relevant.none-task-blog-BlogCommendFromMachineLearnPai2-1.channel_param.
Alexnet. https://neurohive.io/en/popular-networks/alexnet-imagenet-classification-with-deep-convolutional-neural-networks/.
Batchnorm. https://www.cnblogs.com/guoyaohua/p/8724433.html.