ECE590 Lab #2

Construct, Train, and Optimize Neural Network Models

Keru Wu, NetID: kw297

1 LeNet-5

(a) Memory consumption (number of parameters):

Conv,6: (5x5+1)x6=156. Output: 28x28x6

Pool: 0 Output: 14x14x6

Conv,16: (6x5x5+1)x16=2416 Output: 10x10x16

Pool:0 Output: 5x5x16

FC,120: (5x5x16+1)x120 =48120

FC, 84: (120+1)*84 = 10164

FC, 10: (84+1)*10 = 850

All : 156+2416+48120+10164+850=61706

Multiply-Accumulates

Conv,6: 5x5x28x28x6=117600

Pool: 0

Conv,16: 6x5x5x10x10x16=240000

Pool: 0

FC, 120: 5x5x16x120=48000

FC, 84: 120x84=10080

FC, 10: 84x10 = 840

All : 117600+240000+48000+10080+840=416520

2 Constucting DNN

(a) Please see the Jupyter Lab file, we also added batch normalization and Xavier normal initialization.

(b) We used normalization and batch shuffiling for preprocessing. Later on, we tried random crop and random flip for data augmentation.

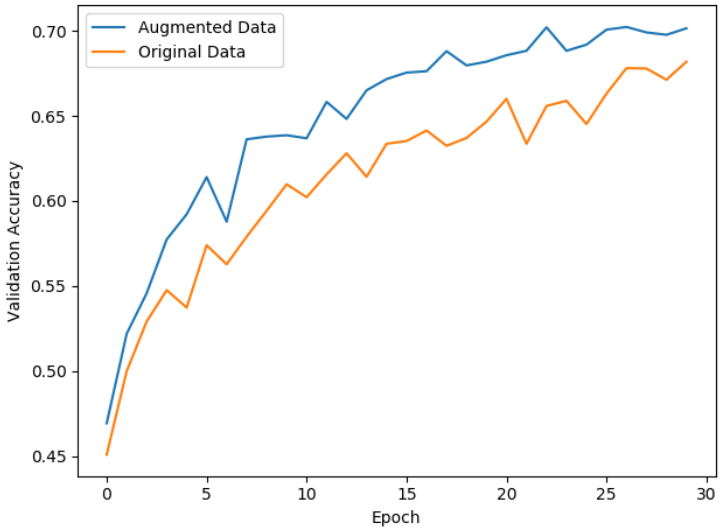
(c) Originally, we used SGD with original learning rate 0.01 and momentum 0.9. We note that the accuracy on training set is much higher than that of validation set, so we increase the regularization term to 5e-4.

3 Training DNN

(a) Please see the Jupyter Lab file. It's interesting to find out that without Xavier normal, the sanity check will be 2.3 (log(10)). However, when we add Xavier normal initilization, the original loss will be larger. (near 3)
The final best validation accuracy is 64.90%.

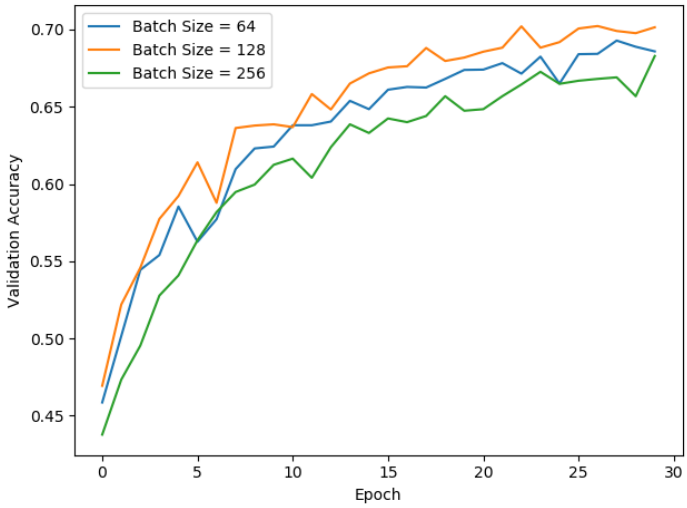
(b) We use transforms.RandomHorizontalFlip() and transforms.RandomCrop(32, padding=4) for data augmentation (only in train set). The final validation (epoch = 30) accuracy rises to 66.6%. What we can find is that the training accuracy is close to (or even lower than) the validation accuracy, while without data augmentation the training accuracy is generally much larger than the validation accuracy.

(c) After we apply batch normalization after the conv-maxpool and fully connected layers. The validation accuracy rises to 69.40% finally. Then we increaset the learning rate from 0.01 to 0.015, we get validation accuracy 70.74%. From the plot we can see that data augmentation lifts up the final accuracy. To some extent, the training process in both cases don't go smoothly, in that our optimization methods (e.g. SGD) doesn't guarantee the increase of accuracy between all epochs. But generally speaking, the accuracy will gradually increase and finally the model will get stuck in some local maximum.

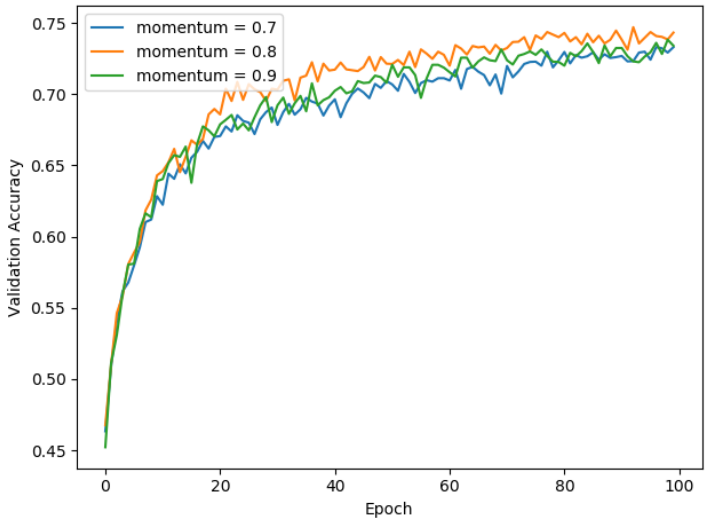


4 Hyperparameter Optimization

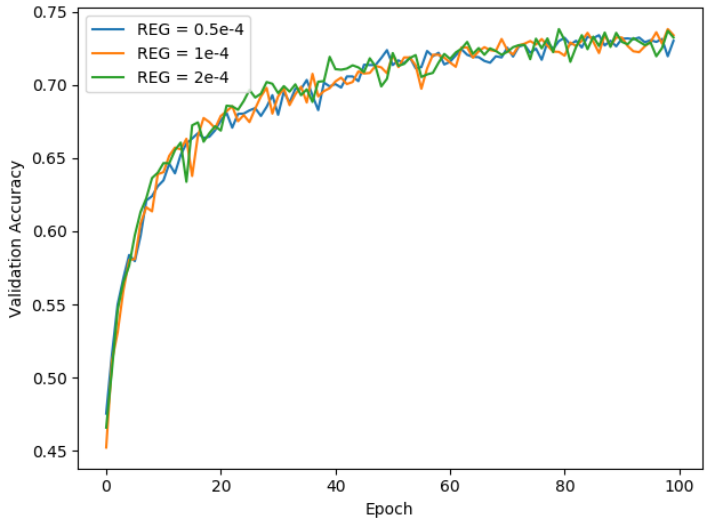
(a) We considered batch size = 64, 128, 256, momentum = 0.7, 0.8, 0.9, REG = 0.5e-4, 1e-4, 2e-4, initial learning rate = 0.015, 0.2, 0.4 and later extend epochs to 100. The validation accuracy plots are as follows. **(Note: we adopted the best parameters from the former parameter experiment to carry out the next experiment)**



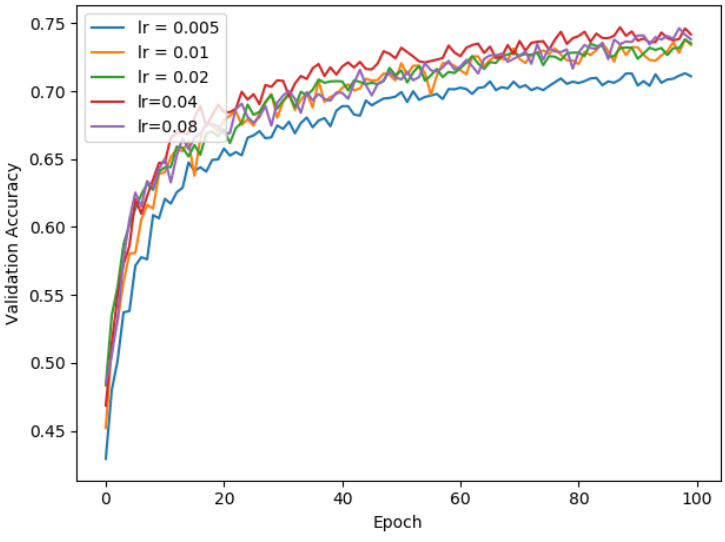
For batch size, we find out that the original batch size = 128 has the best performance. Batch size = 64 may not give correct direction for the whole data, while batch size = 256 the optimization is not stochastic enough.



For SGD momentum, we considered 0.7, 0.8, 0.9 respectively. As is shown above, momentum = 0.8 has the best performance.



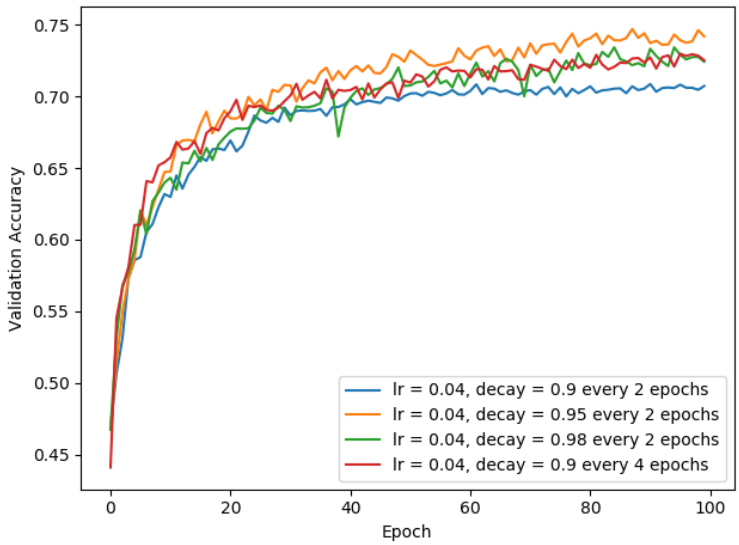
For regularization term, we considered 0.5e-4, 1e-4, 2e-4 respectively. They do not have a large impact on the final model, but the oscillation of training process is related to this term. As we can see, in our experiment, REG = 1e-4 has relatively large oscillation.



For learning rate, we considered 0.005, 0.01, 0.02, 0.04, 0.08. As we can see, larger learning rate may lead to large oscillation, while smaller learning rate may lead to slow convergence (or bad local maxima).

So far, the best hyperparameters are: batch size = 128, REG = 1e-4, momentum = 0.8, learning rate = 0.04.

(b) We set lr = 0.04, and change the decay parameter as well as number of epochs before the decay of learning rate. Results are as follows:



It's interesting to note that learning rate and decay policy are critical in final performance. In our experiment, the best learning rate=0.02 and we decay lr to 0.95 for every 2 epochs. When decay = 0.9, the oscillation of accuracy is the smallest, but the performance becomes the worst as well. Validation accuracy grows fast at first and them becomes stable later. The decay parameter has large influence on both the final accuracy and the learning curve.

(c) Final setting: LeNet-5

(1) Batch normalization after Conv-ReLU and FC-ReLU
(2) Training data augmented by random crop and flip
(3) Hpyerparameter settings: batch size = 128, REG = 1e-4, momentum = 0.8, learning rate = 0.04, decay = 0.95 (every 2 epochs)

Findings (i.e. what I tried and found):
(1) Batch normalization and data augmentation can improve performance greatly (at least in this prediction problem)
(2) Batch size need to be appropriate (not too large nor small)
(3) Including REG term is essential, but performance is not that sensitive to its value (to some extent).
(4) SGD generally performs better than Adam, but it's slower.
(5) Learning policy is critical during training. We may need to try different learning policy for several times to get the best result.

Final performance for LeNet-5: Validation Accuracy ≈ 75%

5 Creating your own DNN for CIFAR-10

In order to use the GPU server, I still use Jupyter notebook for programming. Note that the code for training, validation and testing is almost the same as previous sections. But my DNN structure is different as follows:

```
MyDNN(

(conv1): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv1_bn): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(conv2): Conv2d(32, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv2_bn): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(pool12): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
(dropout1): Dropout(p=0.2, inplace=False)

(conv3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv3_bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(conv4): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
(conv4_bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(pool34): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
(dropout2): Dropout(p=0.2, inplace=False)

(conv5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
(conv5_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(conv6): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1,
1))
```

```
(conv6_bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(pool56): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1,
ceil_mode=False)
(dropout3): Dropout(p=0.2, inplace=False)

(flat): flatten()

(fc1): Linear(in_features=2048, out_features=256, bias=True)
(fc1_bn): BatchNorm1d(256, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(dropout4): Dropout(p=0.5, inplace=False)

(fc2): Linear(in_features=256, out_features=64, bias=True)
(fc2_bn): BatchNorm1d(64, eps=1e-05, momentum=0.1, affine=True,
track_running_stats=True)
(dropout5): Dropout(p=0.5, inplace=False)

(fc3): Linear(in_features=64, out_features=10, bias=True)
)
```

Specifically, it has 6 convolutional layers with 32, 32, 64, 64, 128, 128 filters. The filter size is 3x3 with padding = 1, which maintains the shape 32x32. In addition, we use batch normalization after each layer, and add pooling layer with dropout (20%) after each 2 convolutional layers.

After the convolutional layers, we add two full connected layers with 256 and 64 neurons seperately. We also adopt batch normalization and dropout (50%) layers.

The accuracy on validation set is around 91%.
The accuracy on Kaggle is 91.2%