An improved scheme of convolutional neural network based on CIFAR-10

Jingcheng Li

November 30, 2023

Abstract

Convolutional Neural Networks(CNN), as an essential research object in the field of machine learning, have great potential room for improvement. In this project, I will explore the performance reinforcement of neural networks using the classical datasets CIFAR-10 and compare the test results of different schemes like global average pooling and label smoothing horizontally. Ultimately, this project will provide informative suggestions on the direction of enhancement in training efficiency and testing accuracy for CNN.

1 Introduction

Nowadays, machine learning has received more and more attention. This field has been involved in many industries such as industrial manufacturing, education and research. Convolutional neural network(CNN) plays an important role as an essential idea and tool in machine learning. The most basic CNN structure consists of input layer, convolution layer, pooling layer, fully connected layer and output layer. It completes the specified classification tasks by receiving image data. However, as more and more tasks require CNN, the network needs to be improved to meet different needs. Therefore, this project will focus on improving the network architecture itself and the optimizer. At the same time, I will test the improved network with the help of the CIFAR-10 datasets, aiming to improve the efficiency of network training while ensuring the accuracy of target classification.

2 More explanations on convolution kernel and loss functions

2.1 Convolution kernel

Since the direct use of full connection between the input layer and the hidden layer for the input image would result in too many parameters, the convolution kernel was created to reduce the computation [1]. The essence of convolutional kernel is to extract local features from input images by imitating convolutional neurons, thus greatly reducing the number of parameters while maintaining the original image features.

Specifically, a convolution kernel has three properties: kernel size, stride, and padding. The size of the kernel determines the receptive field of the convolution kernel on the orginal input. A larger convolution kernel means that more image information can be withdrawn, whereas the parameter size and the calculation amount will increase. The stride determines the accuracy of extraction. If the step size is too large, many pixels of the input image will be missed, which will make the image extracted by features less accurate. The padding is related to the case that the original input image is not proportional to the size of the convolution kernel, and this problem is solved by boundary filling. In practice, multiple similar convolution kernels are set to specific receptive fields and locate at different parts of the image. The outputs of such set of kernels form the feature map [2].

2.2 Loss function

The role of the loss function is to measure the difference between the predicted result and the target value. Researchers can reduce the value of the loss function by assessing the difference between these two values and adjusting the parameters of the model using optimization algorithms. The loss

functions commonly used in convolutional neural networks are Mean Square Error(MSE) function and Cross-entropy function [3]. The fundamental form of MSE function is:

$$E = \frac{1}{2}\sum_{k}(y_k - t_k)$$

It is obviously that this function focuses on calculating the difference between predicted value 'y' and the target value 't'. Meanwhile, the basic form of Cross-entropy function is as follows:

$$E = -\frac{1}{2}\sum_{k} t_k \log y_k$$

In this function, 't' represents the target value for each category. Another essential parameter in the loss function is 'y', and it represents the predicted value of each input data. For the part within the log, it has gone through the traditional normalization process using the activate function like softmax function and sigmoid function.

3 The output of original CNN

The original CNN has three convolutional layers and three pooling layers. In each convolution layer, the convolution kernel of size 2×2 scans the input images of size $32 \times 32 \times 3$ with one step size for each time, and extracts the specified number of feature channels. The pooling Windows of the three pooling layers are also set to a size of 2×2 to halve the number of features. After alternatively going through the convolution layers and the pooling layers, the probability distribution of each image about the ten types in the dataset is obtained through three fully connected layers.

The training process of CNN model on CIFAR-10 is presented below:

```
Epoch: 1        Training Loss: 1.843251         Validation Loss: 0.460536
Validation loss decreased (inf --> 0.460536).  Saving model ...
Epoch: 2        Training Loss: 1.842029         Validation Loss: 0.460370
Validation loss decreased (0.460536 --> 0.460370).  Saving model ...
Epoch: 3        Training Loss: 1.841430         Validation Loss: 0.460267
Validation loss decreased (0.460370 --> 0.460267).  Saving model ...
Epoch: 4        Training Loss: 1.840988         Validation Loss: 0.460169
Validation loss decreased (0.460267 --> 0.460169).  Saving model ...
Epoch: 5        Training Loss: 1.840481         Validation Loss: 0.460026
Validation loss decreased (0.460169 --> 0.460026).  Saving model ...
Epoch: 6        Training Loss: 1.839751         Validation Loss: 0.459779
Validation loss decreased (0.460026 --> 0.459779).  Saving model ...
Epoch: 7        Training Loss: 1.838382         Validation Loss: 0.459313
Validation loss decreased (0.459779 --> 0.459313).  Saving model ...
Epoch: 8        Training Loss: 1.835760         Validation Loss: 0.458355
Validation loss decreased (0.459313 --> 0.458355).  Saving model ...
Epoch: 9        Training Loss: 1.829476         Validation Loss: 0.455818
Validation loss decreased (0.458355 --> 0.455818).  Saving model ...
Epoch: 10       Training Loss: 1.809479         Validation Loss: 0.446600
Validation loss decreased (0.455818 --> 0.446600).  Saving model ...
```

Figure 1: The early stage of training

In the early stage of training, the model converges rapidly, and the value of the loss function decreases after almost every iteration. But after two hundred iterations, the model is no longer convergent. As shown in the picture below:

```
Epoch: 290      Training Loss: 0.513072         Validation Loss: 0.189926
Epoch: 291      Training Loss: 0.510231         Validation Loss: 0.189327
Epoch: 292      Training Loss: 0.505542         Validation Loss: 0.193254
Epoch: 293      Training Loss: 0.507851         Validation Loss: 0.190110
Epoch: 294      Training Loss: 0.509143         Validation Loss: 0.191498
Epoch: 295      Training Loss: 0.507213         Validation Loss: 0.190394
Epoch: 296      Training Loss: 0.508902         Validation Loss: 0.189338
Epoch: 297      Training Loss: 0.510122         Validation Loss: 0.192081
Epoch: 298      Training Loss: 0.508823         Validation Loss: 0.187666
Epoch: 299      Training Loss: 0.503886         Validation Loss: 0.191578
Epoch: 300      Training Loss: 0.506955         Validation Loss: 0.191786
```

Figure 2: The training after two hundred iterations

After three hundred iterations of training, I calculate how the model behaves on the test set. The final overall accuracy is 67%:

```
Test Loss: 0.928306

Test Accuracy of airplane: 70% (709/1000)
Test Accuracy of automobile: 75% (756/1000)
Test Accuracy of  bird: 66% (666/1000)
Test Accuracy of   cat: 47% (478/1000)
Test Accuracy of  deer: 60% (603/1000)
Test Accuracy of   dog: 54% (546/1000)
Test Accuracy of  frog: 79% (795/1000)
Test Accuracy of horse: 72% (723/1000)
Test Accuracy of  ship: 76% (760/1000)
Test Accuracy of truck: 75% (759/1000)

Test Accuracy (Overall): 67% (6795/10000)
```

Figure 3: The result on test set for original CNN model

Before I output the test results, I use the softmax function to calculate the confidence of each picture relating to all ten categories. Since the 'torch.max' function has already sifted out the most likely categories for each sample, confident value can be eventually found based on this maximum prediction value. The code changes are shown as follows:

```
# calculate the confidence value
confidence = F.softmax(output, dim = -1)
```

```
# plot the images in the batch, along with predicted and true labels
fig = plt.figure(figsize=(25, 4))
for idx in np.arange(10):
    ax = fig.add_subplot(2, 10, idx+1, xticks=[], yticks=[])
    imshow(images[idx] if not train_on_gpu else images[idx].cpu())
    ax.set_title("{} ({}) {:.2f}".format(class_names[preds[idx]], class_names[labels[idx]], confidence[idx, preds[idx]].item()),
                 color=("green" if preds[idx]==labels[idx].item() else "red"))
```

Figure 4: The code calculating the confidence value

After calculating the corresponding confidence of each test image, I randomly select a batch of test images for display. The results are as follows:

Figure 5: The images in one batch of test set

4 Several methods to improve the network

4.1 Remove one convolutional layer and corresponding pooling layer

The most intuitive way to speed up training by simplifying the model is to remove too many hidden layers. Therefore, I modified the original convolutional neural network to have only two convolutional layers and two pooling layers. This action directly reduces the number of parameters in the model, thus helping the model to speed up training efficiency. In addition, I added a variable of storage time into the code and calculated that the total training time of the modified model for 30 epochs is 592.23s. This method is easy and convenient because it does not involve the modification of network structure, optimizer and loss function.

4.2 Use Global Average Pooling

Since the original network uses three fully connected layers, I use a global average pooling operation instead of a fully connected layer to calculate the average of each feature map of the input. The advantage of the global pooling layer is that it effectively reduces the number of parameters and reduces the risk of over-fitting while preserving the spatial information of the feature map [4]. The mathematical expression of global averaging pooling is:

$$GlobalAveragePooling(i,j,k) = \frac{1}{height \times width} \sum_{m=0}^{height-1} \sum_{n=0}^{width-1} Input(i,j,m,n)$$

where 'i' represents the index in each batch, 'j' is the index of the feature plots, and 'k' is the index of the height and width of the feature plot. The figure below shows the modified code:

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        # Input image size is 32*32, output size of conv2d is: (input_size-kernal_size+2*padding)/stride +1
        self.conv1 = nn.Conv2d(3, 6, 2 , padding=1, stride= 1) # the third param means 2*2 conv_kernal
        self.pool1 = nn.MaxPool2d(2, 2) # 2*2 pooling window
        self.conv2 = nn.Conv2d(6, 16, 2, padding=1, stride= 1)
        self.pool2 = nn.MaxPool2d(2, 2)
        self.conv3 = nn.Conv2d(16, 32, 2, stride= 1)
        self.pool3 = nn.MaxPool2d(2, 2)
        self.global_pool = nn.AdaptiveAvgPool2d(output_size=(4, 4)) # global average pool replaces linear connection
#         self.fc1 = nn.Linear(32* 3**2, 120)
#         self.fc2 = nn.Linear(120, 84)
#         self.fc3 = nn.Linear(84, 10)
        self.fc = nn.Linear(512, 10) # output layer

    def forward(self, x):
        # add sequence of convolutional and max pooling layers
        x = self.pool1( F.relu(self.conv1(x)))
        x= self.pool2( F.relu(self.conv2(x)))
        x= self.pool3(F.relu(self.conv3(x)))
        x= self.global_pool(x)
        # Flatten it
        x= torch.flatten(x, 1)
        # x = x.view(x.size(0), -1)
#         x= F.relu(self.fc1(x))
#         x= F.relu(self.fc2(x))
#         x= self.fc3(x)
        x = self.fc(x)
        return x

# create a complete CNN
model = Net()
print(model)

# move tensors to GPU if CUDA is available
if train_on_gpu:
    model.cuda()
```

Figure 6: The code for global average pooling

As presented from the code, the three fully connected layers are replaced by a global pooling layer, and finally output the probability of ten categories through a linear connection layer. After 590.59s training, I put the results of the test as follows:

4.3 Use Label Smoothing with Kullback-Leibler Divergence as loss function

Label smoothing is a method to solve the problem of overconfidence in the prediction of uniquely thermally-encoded labels in traditional classification tasks by adjusting some contents of labels to be non-zero and introducing noise. This approach can improve the generalization performance of the model and make the model more robust. In addition, it can reduce the sensitivity of the model to abnormal labels in the training data, helping the model to better generalize to previously unseen data [5].

Traditionally, the most common loss function used by convolutional neural networks is the cross-entropy loss function mentioned above. But to better align the idea of label smoothing, I replaced the loss function with the Kullback-Leibler(KL) divergence function. The KL divergence function, also known as the relative entropy function, is often used to measure the difference between different probability distributions. For two discrete probability distributions P and Q, the formula for KL is as follows:

$$D_{KL}(P||Q) = \sum_i P(i) log(\frac{P(i)}{Q(i)})$$

The definition of Label Smoothing class and the results of such loss function into the test set are presented below:

```
Test Loss: 1.745049

Test Accuracy of airplane: 40% (409/1000)
Test Accuracy of automobile: 49% (490/1000)
Test Accuracy of  bird: 15% (158/1000)
Test Accuracy of   cat: 16% (162/1000)
Test Accuracy of  deer: 27% (271/1000)
Test Accuracy of   dog: 33% (333/1000)
Test Accuracy of  frog: 51% (513/1000)
Test Accuracy of horse: 26% (265/1000)
Test Accuracy of  ship: 47% (475/1000)
Test Accuracy of truck: 38% (387/1000)

Test Accuracy (Overall): 34% (3463/10000)
```

Figure 7: The result for global average pooling

```python
import torch.optim as optim
import time


# Define label smoothing loss class
class LabelSmoothLoss(nn.Module):
    def __init__(self, epsilon=0.1, reduction='mean', num_classes=10):
        super(LabelSmoothLoss, self).__init__()
        self.epsilon = epsilon
        self.reduction = reduction
        self.num_classes = num_classes

    def forward(self, output, target):
        code = F.one_hot(target, num_classes=self.num_classes).float() # generate one-hot code
        smooth_target = (1.0 - self.epsilon) * code + self.epsilon / self.num_classes

        loss = F.kl_div(F.log_softmax(output, dim=1), smooth_target, reduction='none')
        # deal with the loss
        if self.reduction == 'mean':
            return loss.mean()
        elif self.reduction == 'sum':
            return loss.sum()
        else:
            return loss

# specify loss function
criterion = LabelSmoothLoss(epsilon=0.1, reduction = 'sum')
# criterion = nn.CrossEntropyLoss()

# specify optimizer
optimizer = optim.SGD(model.parameters(), lr= 0.0001, momentum= 0.9)
```

Figure 8: The code definition for Label Smoothing

4.4 Use Adaptive Moment Estimation as optimizer

Adaptive Moment Estimation(Adam) optimization algorithm optimizes traditional optimization algorithms such as stochastic gradient descent by combining first-order and second-order estimation matrix. It is able to adjust the learning rate in an adaptive way, so that the model converges to a better solution faster [6]. The steps of the Adam optimization algorithm are:

1. **Initialize Parameters:**
    - Initialize model parameters θ.
    - Initialize first moment variable m to zeros.
    - Initialize second moment variable v to zeros.
    - Initialize time step $t = 0$.
2. **Compute Gradients:**

```
Test Loss: 7.441264

Test Accuracy of airplane: 68% (688/1000)
Test Accuracy of automobile: 73% (734/1000)
Test Accuracy of  bird: 51% (519/1000)
Test Accuracy of   cat: 52% (521/1000)
Test Accuracy of  deer: 62% (624/1000)
Test Accuracy of   dog: 50% (504/1000)
Test Accuracy of  frog: 77% (770/1000)
Test Accuracy of horse: 75% (752/1000)
Test Accuracy of  ship: 79% (795/1000)
Test Accuracy of truck: 81% (819/1000)

Test Accuracy (Overall): 67% (6726/10000)
```

Figure 9: The result for Label Smoothing with KL

- Compute the gradient of the objective function $J(\theta)$ with respect to parameters θ.

3. **Update First Moment Estimate:**
   - Update the biased first moment estimate m using exponential decay (β_1 is the first moment decay rate):
   $$m_i = \beta_1 \cdot m_i + (1 - \beta_1) \cdot \nabla_{\theta_i} J(\theta)$$
4. **Update Second Moment Estimate:**
   - Update the biased second raw moment estimate v using exponential decay (β_2 is the second moment decay rate):
   $$v_i = \beta_2 \cdot v_i + (1 - \beta_2) \cdot (\nabla_{\theta_i} J(\theta))^2$$
5. **Correct Bias:**
   - Compute bias-corrected first moment estimate $\hat{m}$ (β_1^t updates with each iteration):
   $$\hat{m}_i = \frac{m_i}{1 - \beta_1^t}$$
   - Compute bias-corrected second moment estimate $\hat{v}$ (β_2^t updates with each iteration):
   $$\hat{v}_i = \frac{v_i}{1 - \beta_2^t}$$
6. **Update Parameters:**
   - Update parameters θ using the bias-corrected moment estimates (α is the learning rate, ϵ is a non-zero constant):
   $$\theta_i = \theta_i - \frac{\alpha}{\sqrt{\hat{v}_i} + \epsilon} \cdot \hat{m}_i$$
   where the
7. **Increment Time Step:**
   - Increment the time step: $t = t + 1$.

The rewritten code is shown below:

```python
# specify optimizer
# optimizer = optim.SGD(model.parameters(), lr= 0.0001, momentum= 0.9)
optimizer = optim.Adam(model.parameters(), lr=0.0001)
```

Figure 10: The code for Adam

The testing result is:

```
Test Loss: 1.165956

Test Accuracy of airplane: 62% (623/1000)
Test Accuracy of automobile: 66% (668/1000)
Test Accuracy of  bird: 43% (435/1000)
Test Accuracy of   cat: 56% (560/1000)
Test Accuracy of  deer: 49% (492/1000)
Test Accuracy of   dog: 43% (437/1000)
Test Accuracy of  frog: 68% (686/1000)
Test Accuracy of horse: 61% (618/1000)
Test Accuracy of  ship: 73% (731/1000)
Test Accuracy of truck: 58% (589/1000)

Test Accuracy (Overall): 58% (5839/10000)
```

Figure 11: The test result for Adam

4.5 Conclusion of four methods

In general, the purpose of the first two of these four methods is to reduce the number of parameters in the model and speed up the convergence of the model. The last two methods emphasize the optimization of the model itself, and strive to make the original network meet higher classification requirements. The following table shows a cross-cutting comparison of training timeliness and accuracy between the four methods and the original network:

Comparison for different conditions in 30 epochs					
	Original CNN	Delete one convolution layer and pooling layer	Using Global average pooling	Using Label Smoothing with Kullback-Leibler(KL) Divergence as loss function	Using Adaptive Moment Estimation(Adam) as optimizer
Training time	593.95s	592.23s	590.59s	609.02s	616.85s
Testing accuracy	67%	55%	34%	67%	58%

Figure 12: The comparison between four methods

5 Further improvements and Conclusion

I can not deny that there is some loss of accuracy because the models in some methods do not converge completely due to the epoch is set to 30 instead of 300 and the learning rate and other parameters have not been more adjusted. Moreover, there is no significant difference in training time due to the large amount of processing time is occupied by CPU. As can be seen in the figure below:

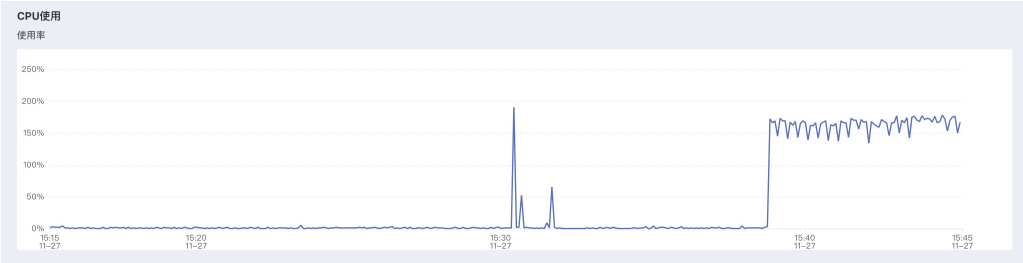

Figure 13: CPU monitoring during training process

This makes the optimization for GPU behavior less effective. If the larger batch size is modified to allow more time to be spent on training the model on the GPU, the experimental results will be more pronounced. Based on the drawbacks, I conducted further experiments. By fully training the four improved models proposed above, I got their final accuracy on the test set as follows:

	Different models' accuracy on test set (fully trained)			
Model name	Delete one convolutional layer and one pooling layer	Using Global average pooling	Using Label Smoothing with Kullback-Leibler(KL) Divergence as loss function	Using Adaptive Moment Estimation(Adam) as optimizer
Accuracy	60%	61%	71%	68%

Figure 14: Comparisons on different methods after fully training

All in all, the above four methods can effectively improve the training speed or the accuracy of the convolutional neural network for classification problems. I believe that these methods will soon be widely used in different classification tasks to meet different requirements.

References

[1] S. Albawi, T. A. Mohammed, and S. Al-Zawi, "Understanding of a convolutional neural network," in *2017 International Conference on Engineering and Technology (ICET)*, pp. 1–6, 2017.

[2] Y. Lecun and Y. Bengio, "Convolutional networks for images, speech, and time-series," *Handbook of Brain Theory Neural Networks*, 1995.

[3] Q. Zhu, P. Zhang, Z. Wang, and X. Ye, "A new loss function for cnn classifier based on predefined evenly-distributed class centroids," *IEEE Access*, vol. 8, pp. 10888–10895, 2020.

[4] M. Lin, Q. Chen, and S. Yan, "Network in network," 2014.

[5] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," 2015.

[6] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *Computer Science*, 2014.