Report of program3

1. Experimental Setting

For this programming assignment, we are constructing a classifier for the CIFAR10 dataset. The data contains the 10 digits: from 0 to 9. The RGB images are 32×32 and each pixel is an interger between 0 and 255. The dataset is split into 50000 training images and 10000 test images, each with the labels in a test file.

Our task is to train a convolutional neural network (CNN) using the stochastic gradient descent method with mini-batch. This NN has one input layer with the shape as $(32 \times 32 \times 3)$ and the output is a softmax layer with 10 nodes output 0-9.

This cnn takes in images after some preprocessing. The preprocessing includes: coverting the unit8s data to float32 data, scaling from [0,255] to [0,1], substracing the mean. The input image should have float32 pixel that are scaled from [-.5, .5]. This preprocessing can prevent the explosiion of the value of the pixels during the training and enhance the training performance.

We are also instructed not to do any modification to the labels.

We used Tensorflow to code the experiment. See the attached file: tensorcode.py.

2. Convolutional Neural Network

2.1 Network Structure

The goal of this CNN is to learn a mapping function which takes an input vector of features $I[m]$, and assigns it to one of K classes $y[m] = k \in \{1, \ldots, K\}$. For the classification problem, we can use a probabilistic approach. In this assignment we used a discrimant method. The NN outputs a probability distribution over all possible value, $p(y = k|I)$, conditioned on the input I.

The input layer of the CNN layer with the shape as $32 \times 32 \times 3$, two convolution layers with 32 size 5×5 filters with two 2×2 max_pooling layers, follwed by another convolutional layer with 64 3×3 filters. After convolution, the nodes are flatten to a vector feed into a fully connect neural network with $3 \times 3 \times 64$ nodes. The output is a softmax layer with 10 nodes output 0-9. A clear description of the structure can be described by the following diagram (Program3Spring20b, page 4):

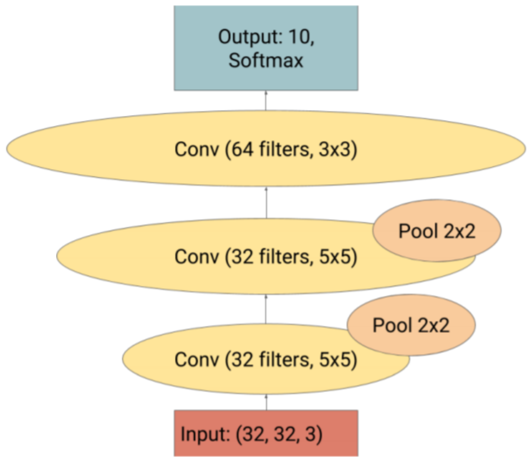


Figure 1: The model that we will be using for this exercise.

2.2 Notation

- The input is denoted as I, which is a $32 \times 32 \times 3$ matrix
- The partially-connected filters of the first conv layers is denoted W^{c1}; second W^{c2}, and the third W^{c3}. The biases are W_o^{c1} for conv1, W_o^{c2} for conv2,W_o^{c1} for conv3. The outputs of the convolution layer are denoted respectively as C_1, C_2, and C_3.
- We are instructed to not use activation function in this assignment, so the output of each convolution layer is fed to the maxpooling layer. The ouput of the maxpool1 is denoted as P_1; maxpool2, P_2.
- Flatten layer output $\vec{P}$ which is a vector.
- The weight of the fully connect layer is denoted as W, bias W_0. The output is Z
- The output layer is a softmax function denoted as $sigma_M$ output $\hat{y}$, which is a probability distribution over the $K = 10$ classes.

2.3 Convolution layer

In the convolution neural net, the first thing is the pass the data through several convolutional layers. The benefit of doing this comes from two aspects.

- The first one is the ensure the training of the network is computationally feasible by having input sharing the weights. In this toy experiment, the image is 32 by 32. If we directly feed the image into a fully connected network with 32 nodes, the numbers of the weights between these two layers is 32^3. Now we have the input stimuli shared these 32 5-by-5 filters, the number of the weights is reduced to 25×32 about 40 times less than the fully connected layer. When processing the real-world photo, which has more than 1000 by 1000 pixels, we can get more computational benefits from using convolutional filters.
- Another benefit is that we can obtain more semantic meaningful features for the image when convoluting the images. The way human process the image is not through pixels but some meaningful features. Given a picture, what comes to the human mind is not 1,2,3,4 pixels value, but some abstract elements, like triangles, dogs..., in the image. We can try to have the filters approximate these semantic features by convolving them on the image. The idea of stacking up several convolutional layers is also understood as approximating how humans process their visual input. Humans can generate some high-level features based on the low-level features, and the stacked convolutional layers are doing something similar. Although when we print the filters, we may usually find them interpretable. But the empirical observations tell that CNN filters learn better semantic features, at least better than fully connected NN (Gonzalez-Garcia et al., 2018)

The mathematically expression of the first convolution is, here I follow the index rule of python, start from 0, so it is slightly different from the class note:

$$C_1(r,c,n_1)=\sum_{i=0}^{4}\sum_{j=0}^{4}\sum_{k=0}^{2} I(r+i,c+j,k)*W^{c1}(i,j,k,n_1)+w_0^{c1}(n_1),\forall r,c=0,\ldots,$$
$$(32-5)/1+1-1,\forall n_1=0,\ldots,32-1$$

The output convolved image C_1 is a 28×28 matrix, so this operation will repeat $28\times 28\times 32$ times because there are 32 filters. For each operation, because it is multiple and sum, we can understand it as something similar to the dot product.

The filters extract the features through a very simple idea, the dot product of two similar distributions is larger than that of two distinct distributions (I guess this is why we need to do normalization of the image). If the convolved value is that, this means in this area, a certain feature or something similar exists, otherwise it does not exist.

The output convolved image can be understood as a map telling if the features existed in this area, except this map does not output a categorical value but a regression value.

The first max-pooling layers are implemented on this map, aiming at compressing the image because usually, the value of those pixels that adjacent to each other are cohensive. We only need to do a summary of them, choosing the large value. The mathematical equation of the max-pooling is:

$$P_1(r,c,n_1)=\max_{0<i<2,0<j<2} C_1(r+i,c+j,n_1),\forall r,c=0,\ldots,28/2-1,\forall n_1=0,..,31$$

Because, we only output one value output a 2-by-2 area, the width and the height of the image is shrunk to half of the input.

The idea of conv2 and conv3 is similar to conv1. I simplify the following description. They are equation are:

- The conv2 output a 32 10-by-10 images ((14-5)+1).

$$C_2(r,c,n_2)=\sum_{i=0}^{4}\sum_{j=0}^{4}\sum_{k=0}^{31} C_1(r+i,c+j,k)*W^{c2}(i,j,k,n_2)+w_0^{c2}(n_2),\forall r,c=0,\ldots,$$
$$9,\forall n=0,\ldots,31$$

- The max-pooling2 output a 32 5-by-5 images (10/2).

$$P_2(r,c,n_2)=\max_{0<i<2,0<j<2} C_2(r+i,c+j,n_2),\forall r,c=0,\ldots,4-1,\forall n_2=0,..,31$$

- The conv3 output 64 3-by-3 images(5-3+1):

$$C_3(r,c,n_3)=\sum_{i=0}^{2}\sum_{j=0}^{2}\sum_{k=0}^{63} C_2(r+i,c+j,k)*W^{c3}(i,j,k,n_3)+w_0^{c3}(n_3),\forall r,c=0,\ldots,$$
$$2,\forall n=0,\ldots,63$$

The only thing I want to add is that the latter two convolution layer takes the output from the previous layers, just like doing reasoing.

2.4 Flatten and fully connected layer

After convolution, the images are flatten to a vector and feed to a fully connected layer with a softmax output.

$$Z = (W^\top \vec{p}) + W_0$$
$$\hat{y} = \sigma_M(Z) = \frac{\exp(Z[k])}{\sum_k{}' \exp(Z[k'])} \forall k = 0, . . , 9$$

This is the regression and categorical regression. In fact, regression is the basis for almost all perdiction algorithm. Convolution is just a feature extraction part, the prediction part should always be done using regression (Categorical regression).

2.5 Loss function:

Let the training data $D = \{I[m], y[m]\}, m = 1, 2, \ldots, M$. Θ indicates all the parameters in the CNN.

And $\tilde{y}$ as the one hot version of y. The CNN can be seen as a function $\hat{y} = p_\Theta(y = k|I)$, so the loss function is the negative log likelihood.

$$\begin{aligned}\mathcal{L}(D; \Theta) &= -\frac{1}{M}\sum_{m=1}^{M} l(\tilde{y}[m], \hat{y}[m]) \\ &= -\frac{1}{M}\sum_{m=1}^{M} \tilde{y}[m]^\top \hat{y}[m]\end{aligned}$$

However, when coding we don't use this loss function. Since $\tilde{y}$ is a one-hot distribution, which means only 1 value is 1, the rest of them are zero. In my tensorflow code, we are instructed to use a fancy tensorflow loss function which can be described as:

$$\mathcal{L}(D; \Theta) = -\frac{1}{M}\sum_{m=1}^{M} \log P_\Theta(y = y[m]|X[m])$$

3. Hyperparameters

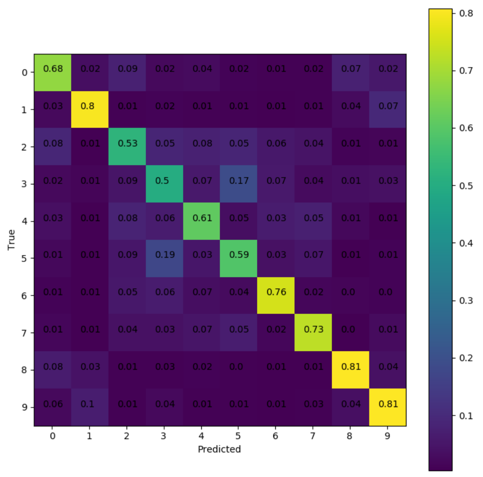
In this assignment, the only hyper-parameter is the learning rate. The learning rate descides how much the weights change from the previous iterations. If the learning rate is too small, the training will be very slow; if the learning rate is too large, the training can be acelerated but we may not be able to each the local minima. I try the learning rate = 0.001, 0.00075, 0.00065, 0.0005, 0.00025. The reason why I choose this if because I choose the same batch_size with the instruction document, so I can make comparsion with the plot in the instruction. I tried .00025 first and I found it learned to slow. I then try .001, and found though it learned fast, the best performance it can reach is about .64. I tried the rest of three, they are pretty close to each other. I finally decided with lr = .00065, which returns about 68.8% accuracy.

I think maybe something is missing in my model.

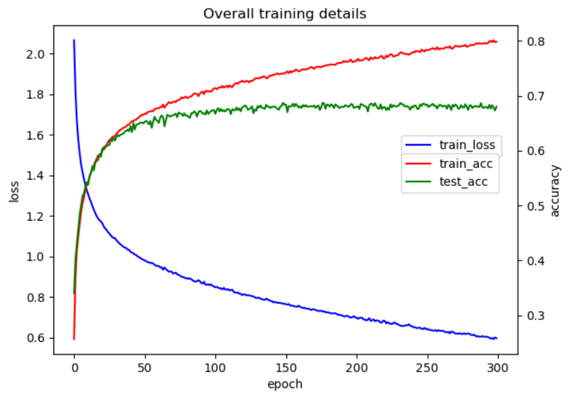
4. Experiment results

4.1 confusion matrix of the final model

We can see the prediction accuracy. The performance of the algorithm is not stable over the prediction over different classes. Class 2-4 is apparently very hard problem, and my CNN performance slightly better than guessing by chance. For the class three, it is even lower than by chance.



4.2 Overall training loss and error



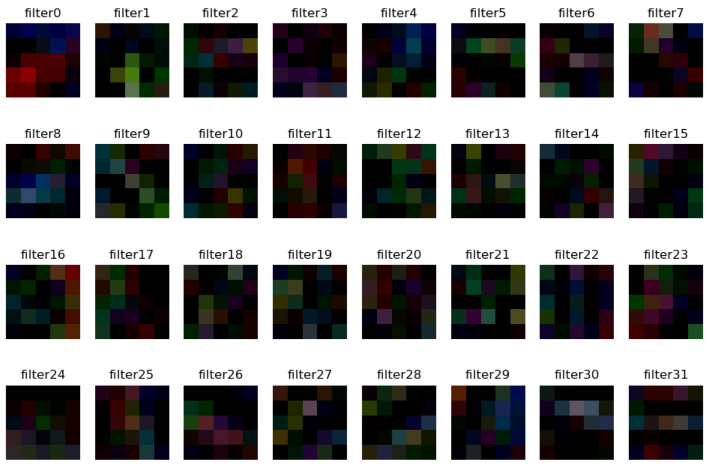
My final model cannot reach the 70% accuracy, might because something is missing in my model. The good news is the tendency of the curves is similar to the instruction documents, which means my model is not that bad.

Here I also used a batch size of 2500, and I plotted the loss and accuracy over the epochs. Due to selecting the same batch size, I can adjust my learning rate by comparing it with the plot in the instruction document. However, I tried a lot of learning rates but failed to reach the performance of the benchmark model.

Also, when I used the format_check file from LMS, I got a very different accuracy from my expectations, about 58.5%. When I add preprocessing to that file, the performance of my model then met what I expect. I guess maybe my model relays too much on the preprocessing.

4.3 plot the filter.

I normalized the filters using: filter / (filter.max() - filter.min()). The filters looks better.



```
In [ ]:
```