2023 Fall NYCU-EE Deep Learning – Lab01

Name: Kuan-Wei Chen 陳冠瑋 **ID: 310510221 / Institute of Electronics /Sep. 29, 2023**

***Abstract*- In this lab, I built a simple three-layer CNN neural network from scratch, with forward and backward propagation. It includes a convolution layer, max pooling layer, and fully-connected layer. I chose softmax for activation and used cross-entropy for loss function. To improve accuracy, I incorporated a step learning rate scheduler for stable training. Additionally, I applied K-fold cross-validation and shuffling to counter overfitting.**

I. Introduction

The objective of this lab is to manually implement both forward and backward propagation for a neural network without relying on AI frameworks or related packages (i.e. PyTorch, TensorFlow, Keras, etc.). We will work with the Fashion MNIST dataset, which comprises 60,000 training images and 10,000 test images. These images are 28x28 pixels in size, and the dataset includes 10 distinct classes.

II. Methodology

A. *Network Architecture*

In this lab, I utilized a simple CNN model with a single 2D convolution layer, 2D max pooling, and a fully-connected layer, as shown in Fig. 1.

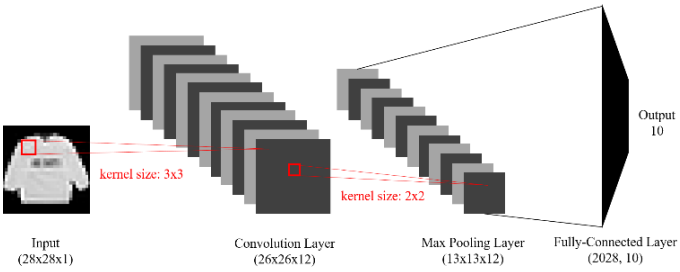

Fig 1. Simple CNN Model

For the convolution operation, I defined the input size (*input_size*), number of filters (*n_filters*), and kernel size (*k_size*) as parameters. In practice, due to training on a CPU where time is a significant constraint, I set the input size to 28, the number of filters to 12, and the kernel size to 3 for constructing my convolution layer. For simplicity, I set the parameter of stride to 1, padding to 0, and batch size to 1. I also employed a method to generate iterate regions, allowing iteration through each receptive field for direct convolution with the kernel. In backpropagation, it's can be seen as the reverse propagation of the fully-connected layer, involving the computation of the output gradients and the update of filter weights.

In the MaxPooling operation, I set the default kernel size to 2 and create iteration regions to generate 2x2 regions for each iteration, selecting the maximum value among the pixels. During backpropagation, it calculates the gradient. This process involves taking an output gradient tensor (*output_grad*) received from the layer after the pooling layer. Initially, I initialize a gradient tensor (*dL_dinput*) with the same shape as the input. Then, I iterate through the previously visited 2x2 regions, identifying the maximum value in each region. Subsequently, I multiply the corresponding output gradient by 1 (as max pooling only transmits the gradient of the maximum value) and store the result in the corresponding position of the input gradient tensor. Finally, I return the computed input gradient.

In the last Fully Connected layer, with an input size of 13x13x20 and an output of 10 classes, the softmax activation function is applied. During backpropagation, gradients for each output node are computed sequentially using derivation formulas, and then weights and biases are updated accordingly.

B. *Activation Function*

In the fully-connected layer, I use Softmax as the activation function. The formula is as follows and the implementation of Python code as in Fig. 2:

$$softmax(x_i) = \frac{\exp(x_i)}{\sum_j \exp(x_j)} \quad (1)$$

```python
# e^totals
t_exp = np.exp(self.last_total)

# Sum of all e^totals
S = np.sum(t_exp)
```

Fig 2. Softmax function (Python)

C. *Loss Function*

I opted for the cross-entropy loss function, commonly used with Softmax for effective multi-class classification training. The calculation formula is as follows, and the implementation of Python code as in Fig. 3:

$$Loss = -\sum_{i}^{N} y_i \cdot log\hat{y}_i \qquad (2)$$

```python
epsilon = 10e-10
loss = -np.sum(label * np.log(out + epsilon))
```

Fig 3. Cross-entropy Loss (Python)

III. Optimization Method

A. *Learning Rate Scheduler*

I implemented a step learning rate scheduler that reduces the learning rate by a factor of 2 every four epochs, illustrated in the Fig. 4. This helps achieve better convergence in the later stages of training. Python code implementation is shown in Fig. 5.

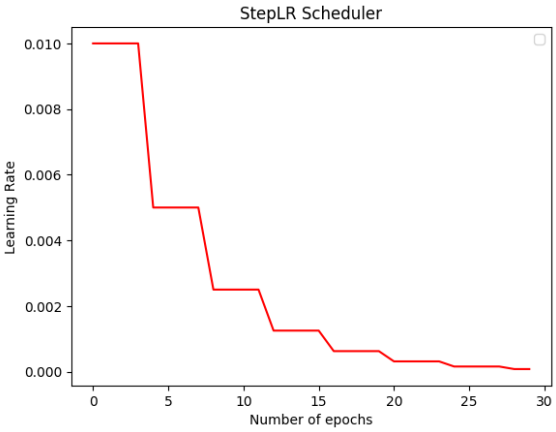

Fig 4. StepLR Scheduler

```python
lr_scheduler = [Learning_rate]*EPOCH

exp = 1
current_lr = Learning_rate
for idx, lr in enumerate(lr_scheduler):
    if idx != 0:
        if idx % scheduler_step == 0:
            lr_scheduler[idx] = lr * 0.5**exp
            exp += 1
            current_lr = lr_scheduler[idx]
        else:
            lr_scheduler[idx] = current_lr
```

Fig 5. Learning Rate Scheduler (Python)

B. *K-Fold Cross Validation & Shuffling*

The K-fold cross validation assesses the model performance effectively by dividing the dataset into K subsets, using K-1 subset for training and one for validation in each iteration, repeating this process K times with different test subsets. It provides a stable performance estimate by averaging results across iterations and helps evaluate model generalization while reducing data splitting randomness. In practice, I used K-fold cross validation with K set to number of epochs, which is 30, generating K subsets. Each epoch involved using a different subset for validation, as shown in Fig. 6.

Shuffling disrupts the sequential order of data, prevent model bias from the sample sequence. This enhances feature and pattern learning, improving overall performance generalization by reducing reliance on data order.

```python
k = EPOCH
k_fold_train_data, k_fold_train_label, k_fold_train_label_onehot = [], [], []
k_fold_val_data, k_fold_val_label, k_fold_val_label_onehot = [], [], []

fold_len = int(len(train_data) / k)

print("Generate: ")
for i in range(k):
    print("fold ", i)
    start_idx = i * fold_len
    end_idx = (i + 1) * fold_len

    # training fold
    k_fold_train_data.append(np.vstack((train_data[:start_idx],train_data[end_idx:])))
    k_fold_train_label.append(np.hstack((train_label[:start_idx],train_label[end_idx:])))
    k_fold_train_label_onehot.append(np.vstack((train_label_onehot[:start_idx],train_label_onehot[end_idx:])))

    # validation fold
    k_fold_val_data.append(train_data[start_idx:end_idx])
    k_fold_val_label.append(train_label[start_idx:end_idx])
    k_fold_val_label_onehot.append(train_label_onehot[start_idx:end_idx])
```

Fig 6. K-fold Cross Validation

C. Experimental Result

Fig 7. and Fig 8. represent the training curves of my model. The model was trained for a total of 30 epochs and employed techniques such as *Learning Rate Scheduler*, *K-Fold cross-validation (K=30)*, and *Shuffling* to enhance performance. Table1 shows a comparison of all the experiments I conducted with the performance metrics in comparison to the best and median performance on the leaderboard.

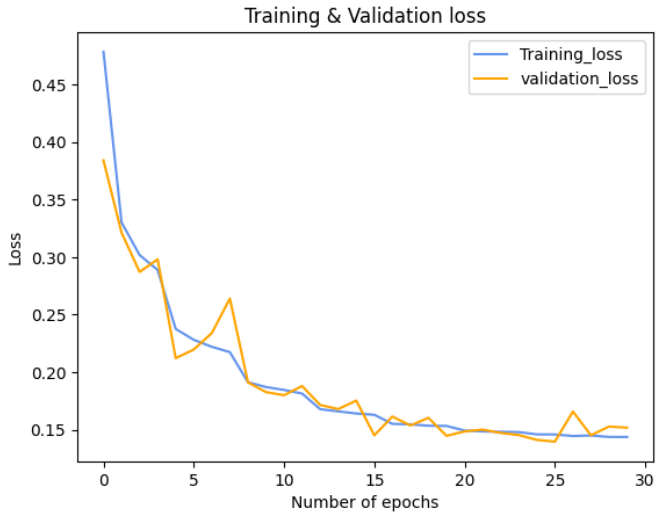

Fig 7. Training and Validation Loss

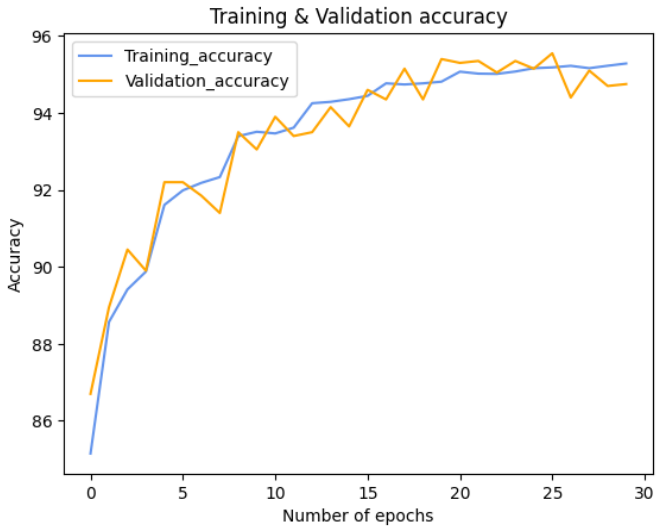

Fig 8. Training and Validation Accuracy

Table1. Model Performance

Method	**Top1 Acc.**
Baseline [FC (13 x 13 x 12, 10)] (15 epochs)	88.83%
Baseline [FC (13 x 13 x 12, 10)] (15 epochs, LR Scheduler)	89.31%
Baseline [FC (13 x 13 x 12, 10)] (15 epochs, LR Scheduler, K-Fold)	89.79%
Fin-tune [FC (13 x 13 x 16, 10)] (20 epochs, LR Scheduler, K-Fold & Shuff.)	90.00%
Fine-tune [FC (13 x 13 x 20, 10)] (30 epochs, LR Scheduler, K-Fold & Shuff.)	90.17%
Best Code	91.94%
Median	89.18%

D. Conclusion

In this lab, I created a three-layer CNN neural network from scratch, including forward and backward propagation. It had a convolutional layer, max-pooling layer, and fully-connected layer, using softmax activation and cross-entropy loss. I improved training stability with a step learning rate scheduler and combated overfitting through K-fold cross-validation and data shuffling. As a result, the model achieved an top-1 accuracy of 91.94%.