Handwritten Digit Pairs Classification

ECE4200 MiniProject Report

Yiyan Peng (yp449)

1 Introduction

After the Naïve Bayes, softmax regression, and kernel SVM were implemented, these machine learning algorithms were found to not able t finish the multiclass classification really well. So, the neural networks were considered for more complex problems. Here, I got the highest test accuracy by using a Convolutional Neural Network (CNN) with or without the Residual Neural Network (ResNet) elements in the PyTorch framework.

2 Data Cleaning & Preprocessing

2.1 Training & Validation Dataset Split

In order to see the fit performance of the model, ***sklearn.model_selection.train_test_split*** was used to split the "training" data into ***training and validation sets***, with the ratio of 8:2. The ***random state*** was set fixed so that the model can be trained or validated on the data sets in the same orders but with shuffle. In this way, it is convenient to estimate the model performance on the unknown test set.

2.2 Dataset Normalization

By checking the dataset elements and the corresponding visualization, it is easy to find that the range of the pixel values is [0, 255). Here, the dataset was divided by 255 to do the ***normalization***, which can help CNN to train more stably, and maintain the adaptability and stability of the ***ReLU*** activation function.

2.3 Tensor Conversion & Customized TensorDataset and DataLoader

In PyTorch, we have to convert the data into ***tensor*** data type to use the CUDA and GPU's acceleration. Also, the ***customized tensordataset and dataloader*** were defined to help PyTorch to grasp and train the data. As for the training dataloader, ***shuffle*** was applied to stablize the training process. And for the validation dataloader, the data did not need to be shuffled because we only need to use the validation set to validate the performance. In the ***pure-CNN*** model, both two dataloaders used the same ***batchsize*** 32. In the ***CNN-ResNet*** model, the ***batchsize*** was set to 125. (If the size is too big, we can train the model faster by utilizing more capability of the GPU, but we cannot so much frequently update our model. If the size is too small, we can update model more frequently but much more easily to take in the data noise.)

3 Model Architecture

3.1 Naive Bayes

Apart from using the above data preprocessing methods, the ***sklearn.preprocessing.Binarizer*** (threshold=0) was used to convert the data into 0 if the data is smaller than 0, or into 1 if the data is bigger than 0, which is beneficial for classification. Here, the ***BernoulliNB*** was chosen to be the model because all the data right now is independent and binary. Then, the model was trained by using the sklearn's ***fit*** and the testset predictions were gotten from the ***predict*** function. After trials, the Naive Bayes algorithm could get 61.97% training accuracy, 61.35% validation accuracy, and 63.2% test accuracy.

3.2 Softmax Regression

Softmax regression here is the same as the multiclass logistic regression. After the data got the preprocessing, ***sklearn.linear_model.LogisticRegression*** was used to train the model. As for the *logisticRegression* function, we should specify the parameters. we should choose "multi_class" as "multinomial" because we are dealing with 15 class classification tasks. The optimizer "solver" should be set as "lbfgs" which is in the family of quasi-Newton methods. It approximates the Broyden–Fletcher–Goldfarb–Shanno (BFGS) algorithm, which is a robust and efficient algorithm for solving non-linear optimization problems. The maximum iteration was set to 1000 and the random state was 20 which was to guarantee random training but training the same order dataset every time. After trials, the softmax regression could get 79.23% training accuracy, 76.13% validation accuracy, and 76.98% test accuracy.

3.3 Kernel SVM

Due to non-linearity, it is better to choose the Radial Basis Function (RBF) as the kernel for our SVM. After the data preprocessing, the ***sklearn.svm.SVC*** was applied as the SVM classifier, with the parameters "kernel='rbf', gamma='scale', C=1, random_state=20". The gamma parameter was set to default "scale" ($gamma = \frac{1}{\text{n\_features} \times X.var()}$) and the regularization parameter "C" was set to default 1, which were all good for our model considering the dataset. Then, from the fit and predict functions from sklearn, the kernel SVM algorithm could get 90.20% training accuracy, 84.90% validation accuracy, and 83.39% test accuracy.

3.4 Pure-CNN

At the beginning, a few layers were tested, which was found could not achieve good performance. Then, more layers with more filters were designed to let the network learn the deep features of the dataset images. At last, the pure-CNN structure is shown as follows.

Batch normalization normalizes the input data by standardizing it to have zero mean and unit variance. To help mitigate the issues of vanishing or exploding gradients, leading to faster convergence during training, every convolutional layer has a ***batchnorm*** layer behind it. The convolutional layers all used the 3×3 kernel and applied the padding in size of 1, which can generate the outputs in the same size of the inputs. The structure of the ***Conv2d*** layers with filters is: 64 → 128 → 256 → 512 → 512 → 256 → 128 → 32. To reduce dimensionality and extract the main features, the ***MaxPool*** layer was applied behind the third and fifth conv layers. Also, a ***dropout*** layer was applied after the fifth conv layer to reduce the overfit. At last, the linear fully connected layers were used to output the results.

Note. The training loss curve image and the model structure are contained in the submission folder.

3.5 CNN-ResNet (Highest Test Accuracy)

After the pure-CNN model was trained, it seemed that it was difficult to optimize the loss when the epoch reached a certain number. To help the model optimize faster and avoid the performance degradation when the training gets deeper and avoid the vanishing gradients while training, the ***Residual Network elements*** were introduced to the pure-CNN.

Every residual block first consisted of two convolutional layers. As for the ***residual connection***, the input (*i.e.* residual part) was added to the output of the second batch normalization if no ***downsampling*** was needed. If downsampling was required, it was applied to match the dimensions. Then, process input through these layers and add the residual connection before the final ReLU activation.

With the residual blocks, the CNN-ResNet structure was designed as follows.

Conv2d(filter=64) → *Batchnorm&Relu* → *ResidualBlock1(filter=64)* → *ResidualBlock2(filter=128)* → *ResidualBlock3(filter=256)* → *ResidualBlock4(filter=512)*→ *AveragePooling(pooling region=*3×3*)* → *Linear Fully-Connected layer(output=15).*

From the training loss and the final prediction accuracy, the CNN-ResNet model performed better than the pure-CNN model.

Note. The final output corresponds the probablity of the 15 predicted classes. The training loss curve image and the model structure are contained in the submission folder.

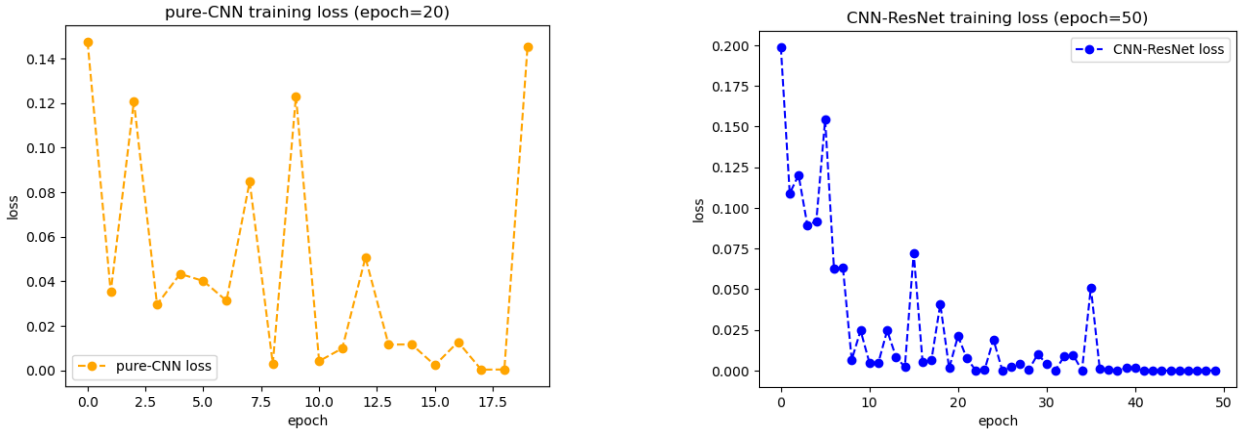
4 Loss Function & Optimizer & Training

Considering this is a multi-class (class=15) classification problem, the ***CrossEntropyLoss*** was chosen as the criterion/loss function from PyTorch. There is the softmax calculation inside the loss itself.

For both the two models, the ***Adam*** was used as the optimizer. But the pure-CNN's learning rate was 0.0005 and the CNN-ResNet was 0.0008. (After trials, Adam can generally perform better than RMSprop and other optimizers.)

Because CNN can be difficult to optimize the loss within quite few epochs, the pure-CNN model was only trained 20 epochs. The 96.94% validation accuracy and 96.879% test accuracy on Kaggle were gotten, which were really good. Later when more epochs were used, the accuracy would not increase, but even dropped instead. As for the CNN-ResNet model, 50 epochs were trained and got 97.44% validation accuracy and 97.536% test accuracy on Kaggle.

The pure-CNN & CNN-ResNet training loss curves:



(a) Pure-CNN Training Loss Curve

(b) CNN-ResNet Training Loss Curve

Figure 1: Comparison of Training Loss Curves