FINAL PROJECT REPORT

Implement a handwritten digit recognition classifier using CNN

Zonghao Li zl2613
Wenting Yu wy2294
Guyu Zhang gz2263

May. 6th

Supervised by
Professor Xiaofu He
TA Sai Pavan Kumar Unnam

Contents

I. How CNN Works:

1. An image is converted into arrays where each cell represents a value.
2. Then filters are applied to find features of an image. Features may include edges.
3. Then the image size is reduced by the pooling layer (max pooling is used in our model).
4. After the pooling layer, the reduced image is passed through an activation function like RELU (Rectified Linear Units), which is an activation function that decides the final value of a neuron.
5. Then we apply a dropout layer to randomly remove some neurons to prevent overfitting.
6. Finally, a dense layer is applied to connect all neurons where each neuron gives a probability of image accuracy.

II. Parameters/Hyperparameters of CNN

Model.add and Sequential code: layers are added piecewise through the 'Sequential' object

1. Conv2D(32, (3, 3), input_shape=(28,28,1)):
    a. Number of filters: 32
    b. Size of our filters: (3,3)
    c. input_shape(28,28,1): the input shape of our image data
    d. We have four Conv2D layers and each has 32, 32, 64, 128 filters respectively with (3, 3) filter size
2. BatchNormalization(axis=-1):
    a. our data format is "channels_last", so set axis = -1
    (if use "channels_first", then set axis = 1)
3. Activation('relu'):
    a. Choose ReLu as our activation function
4. MaxPooling2D(pool_size=(2,2)):
    a. Our pool size is (2,2) which means we downscale both vertical and horizontal dimension.
5. Dense: A dense layer is a layer where each neuron is connected to each neuron in the next layer that could accurately classify the data.
    a. Dense(512): output shape 512
    b. Dense(10): output shape is 10 which equals the number of classes
6. Dropout(0.25): randomly disable 25% of neurons in our model
7. Other parameters:
    a. epochs:30 epochs times

b. batch_size: size of 128
c. num_class: 10 number of classes
d. img_rows/img_cols: both 28 since the shape

III. Code Logic:

1. Import all libraries
2. Set random seed
3. Set our epoch as 30, batch size as 128, and the number of classes as 10 for labeling our y_train and y_test later
4. Download the MNIST dataset provided by Keras use the code minist.load_data( ). The shape of x_train is (60000, 28, 28), and each image has a 28*28 resolution. The shape of x_test is (10000, 28, 28)
5. The input shape CNN accepted in the Tensorflow is a (batch, height, width, channels) format. Thus, we reshape our data as this format through the if else code. Then the shape of our x_train becomes (60000, 28, 28, 1), and the shape of x_test become (10000, 28, 28, 1). 1 channel here means all our images are in grayscale.
6. (x_train/255, y_train/255) rescales the image data so that each pixel is within the interval [0,1] instead of [0, 255]
7. Then we use previously assigned number of classes (num_class=10) to label each of our y_train and y_test data point to convert class vectors to binary class matrices. For example y_train[i]=[0,0,0,0,0,0,0,1,0,0] represents that the data observation i has a label 8. After this steps, we finished the data preparation part and could move to create CNN.
8. CNN steps
    a. Convolution: Conv2D
        i. Create a convolution kernel that is convolved with the layer input to produce a tensor of outputs. For this part of the code, we have 32, 32, 64, 128 number of filters respectively and (3, 3) as the size of our filter for each. One more thing needs to mention is that Conv2D only needs to specify the input shape when used for the first time. The input shape here is (28,28,1).
    b. Activation:
        i. This is the second layer. We use ReLU (rectified linear unit) as our activation function. This function changes all the negative values in matrix as 0 and keeps remaining same as previous.
    c. Pooling: MaxPooling

        i. This is the third layer and is used to down-sample the input in order to reduce overfitting. Our pool size is (2, 2) which means we downscale both vertical and horizontal dimension.
    d. Normalization: BatchNormalization
        i. scaled the matrix after each convolution
    e. Flatten all convolutional layers to enable them as an input for Dense layer we use later. Before flattening all layers, we add more layers through repeated previous steps a-d.
    f. Dense (512): means fully connected layer
        i. A dense layer is a layer where each neuron is connected to each neuron in the next layer that could accurately classify the data. (The neuron of each layer will define the output shape (the shape of the tensor that is produced by the layer and that will be the input of the next layer)).
        ii. The output space dimension is 512
    g. Dropout:
        i. A regularization technique that reduces the complexity of model in order to avoid overfitting. It forces the model to deactivate certain neurons in a layer with a certain probability.
        ii. In our model, we randomly disable 25% of neurons.
    h. Dense (10) layer:
        i. The dense layer of ten neutrons and we need this layer to be the number of classes we want to predict as this is the output layer.
    i. Softmax Activation layer: this layer assigned a probability to each class and the class with maximum probability is our model final output.
8. Compile the model by code model.compile
    a. Categorical_crossentropy is used to calculating the error rate between the predicted value and the true value.
    b. Use Adadelta (adaptive learning rate method) optimizer with its default value.
9. Training and validation by code model.fit
    a. Treat ⅛ of training data as the validation set
    b. Fit the model on ⅞ of training data and validate the model on the validation set
10. Testing
    a. Test the result on the testing data to obtain the test accuracy
    b. Final model accuracy: 99.55%

IV. Start Model

We choose to implement the model from the tutorial (https://github.com/keras-team/keras/blob/master/examples/mnist_cnn.py). The model is a sequential model with 8 layers, and it uses cross-entropy as loss function and Adadelta as the optimizer. The test accuracy rate for this initial model is 99.22% without any modification. But since this model uses test data for validation when training the model, the test accuracy is not valid. We will explain this part later in the extra credit part.

V. Final Model with intermediate stages

1. We first changed the validation method from using test dataset into a split of training data set. This lowers the final test accuracy from 99.22% to 99.08%.
2. After searching through plenty of online tutorials about Keras and TensorFlow, we learned that adding a BatchNormalization after a layer would help to reduce training time and give a better estimation. It would normalize the matrix to make sure that each dimension remains the same after transformation.
3. After that, we added several additional convolutional layers and also a dense layer into our model. Since each layer could add its own level of non-linearity, multiple layers could extract different features from the data and add levels of abstraction. We expect adding more layers could help to make the model more representative and accurate.
4. Finally, we increased the epochs from 12 to 30 to make sure our model was fully trained and extracted distinguishable and important features from the handwritten data. After all the changes and modification, the test accuracy for our final model reached 99.55%.

VI. Difficulties

1. Unlike other machine learning techniques, Neural Network is essentially a black box thus difficult to understand how the model works. We could train the model with various of combinations of different layers and activation functions and the only performance evaluation for each model was based on testing error. Since there exists no determined research or conclusion on how to find the best combination, we have to arbitrarily try different models by adding layers or changing the number of filers to find the the model that gives the best testing accuracy for this project.
2. Training a neuron network model requires high-performance hardware. The waiting time for training each model is significantly long, which limits the number of different models

we could try to find the best one. Also, adding more layers to the model could increase the waiting time significantly. As a result, the number of layers we could add to this project is limited, which might be a reason for our limited improvement of test accuracy.

VII. Further Improvement

1. Due to the limitation of time and hardware, we could not build highly advanced models. For further studies, we could try to build more complex models by adding more layers to the model and increasing the epoch size. This would require a longer time to train the model, but it may give a better performance.
2. Instead of "Adadelta", we could also try other optimizers like "rmsprop" or "Adam" that could possibly give us a better performance. Tuning parameters of the layers like the size of convolutional kernel may also make the model more accurate for this handwritten data set.

VIII. Extra Credits Parts

Part 1. Improve the model further (test accuracy >=99.5%)

Final model Accuracy: 99.55%. For details, please check our ipython file.

Part 2. Fix the bug for the Keras MNIST sample code

- **Bug:** model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs,verbose=1, validation_data=(x_test, y_test))
- **Reason:** The sample code uses test data as validation set when training the model. Since the information from test data has already been used when training, this would artificially make the test accuracy rate higher than real accuracy rate. In order to make the test accuracy rate valid, we changed "validation_data=(x_test, y_test)" into "validation_split=1/5." as shown below.
- **Correction:** model.fit(x_train, y_train,batch_size=batch_size,epochs=epochs, verbose=1, validation_split=1/5.)