George Mason University

Learning From Data Fall 2024

Computer Exercise #8 Part 1

Assigned: November 20, 2024
Due Date: December 05, 2024

In Part 1 of this exercise you will be experimenting with convolutional neural networks to classify objects in the MNIST dataset (as a warm-up). In Part 2, soon to be issued, you will use what you have learned about CNNs to classify images in an image database that you will be given. A test set will be withheld and your classifier will be submitted and tested on this dataset. You code **must work** on images read into your notebook, and what you have done **must be** fully documented.

The **Questions** in this exercise are for you to think about and answer in order to understand what you are doing, and do not need to be included in your writeup.

Note: It is easy to find code for designing CNNs on a variety of datasets, and some examples were given in class. It is expected that you will not copy code from other sources and put them into your assignment. You should design your own and perform experiments. This is the **only way** to gain a better understanding of CNNs - how they work, issues in training, trade-offs in design architectures and so on. You will be expected to be able to look at code and see what it is doing, debug and find errors, and understand how to design and build a CNN. The goal is for you to learn something, and not to score the maximum number of points on this exercise or checking the box that you did this assignment.

Computer Exercise 8.1 (Warm-up):

This part is a warm-up exercise to get comfortable with Keras and, more importantly, to begin to understand the nuances in designing a CNN. You will be looking at a number of different CNN architectures to classify digits in the MNIST dataset along with some fully connected neural networks and comparing the performance of each. The steps involved in building an image classifier, using either a fully-connected neural network or a CNN are:

(i) **Common Imports**: Loading the libraries necessary to design, train and run the CNN.

(ii) **Loading the Dataset**

(iii) **Preprocessing**: The dataset needs to be put into the correct format for Keras, and to help in training, proper scaling of the data should be done.

(iv) **Define the Model**: Here is where the architecture of the neural network is defined - how many layers, what type of layers (fully connected or convolutional), and the activation functions in each layer (hidden and output). In the case of CNNs, other decisions would include the number of feature maps, the type of pooling, and so on.

(v) **Compiling the Model**: Once the model is specified, it must be compiled so that it can be trained. Here it will be necessary to specify how the SGD search is done, the loss function that is to be minimized, and the metrics to be tracked during training.

(vi) **Training**: Training data is used to find the parameters of the model. What needs to be decided for training are the number of epochs, the batch size, and the data set that is used for validation.

(vii) **Model Evaluation and Prediction**

1. Common Imports

Begin by loading the following libraries:

```
from tensorflow import keras
import numpy as np
from keras import Input as Input
from keras.datasets import mnist
from keras.layers import Dense, Flatten
from keras.layers import Conv2D, MaxPooling2D
from keras.models import Sequential
from keras.utils import np_utils
import matplotlib
import matplotlib.pyplot as plt
```

Descriptions of the keras modules will be discussed later.

2. Loading and Preparing the MNIST Dataset

The next step is to load the MNIST dataset, which comes as part of the Keras library. This may be done as follows:

```
(x_train, y_train), (x_test, y_test) = mnist.load_data()
```

Next, it is necessary to reshape the datasets `x_train` and `x_test` into a 4D tensor, which is the format that keras expects to see. The first is the number of images (60,000 for `x_train` and 10,000 for `x_test`), and the next two define the shape of each image, which for the MNIST data set is 28×28. The last is the number of channels. Since the MNIST dataset consists of greyscale images, then there is only one channel. For RGB color images, the number of channels would be three. The instructions to perform this reshaping are as follows:

```
#Image dimensions
img_x, img_y = 28, 28
x_train = x_train.reshape(x_train.shape[0], img_x, img_y, 1)
x_test = x_test.reshape(x_test.shape[0], img_x, img_y, 1)
```

The next step is to convert the image pixel values to the right type (32-bit floating point numbers) and normalize the data, converting them from $[0, 255]$ to numbers in the interval $[0, 1]$. **This makes the network easier to train**.

```
# convert the data to the right type
x_train = x_train.astype('float32')
x_test = x_test.astype('float32')
x_train /= 255
x_test /= 255
```

Question: What do you get when you type `y_train[0]`?

Since you will be using the softmax output activation function and a cross-entropy loss function, it is necessary to *one-hot-encode* the target variable. This means that a binary column vector will be created and assigned to each category.

```
num_classes=10
y_train = np_utils.to_categorical(y_train, num_classes)
y_test = np_utils.to_categorical(y_test, num_classes)
```

Questions:

(a) Now what do you get when you type `y_train[0]`?

(b) How has the encoding of the target values changed, and why is this change necessary?

3. Building a Model

Every Keras model is built using either the `sequential` class, which allows for the design of a linear stack of layers, or the functional `model` class, which is more customizable. Since our network will be a stack of layers, it is convenient to use the simpler sequential model.

The first step is to instantiate a sequential model, and then follow this by adding layers to the model with the `add(...)` function.

```
#create model
model = Sequential()
#add model layers
model.add( ...)
model.add( ...)
...
model.add( ...)
```

The types of layers that may be added include (see Keras documentation):

Core layers	`Dense, Activation, Dropout, Flatten, Input+`, and `Reshape`
Convolution Layers:	`Conv1d()` and `Conv2d()`
Pooling Layers:	`MaxPooling1d`, `MaxPooling2d` `AveragePooling1d`, and `AveragePooling2d`
Normalization Layers:	`BatchNormalization`

A simple example is given below

```
model = Sequential()
model.add(Flatten())
model.add(Dense(32,))
model.add(Dense(10))
```

When a sequential model is instantiated without an input shape, it isn't *built* - it has no weights and calling a method such as `model.weights` will result in an error. The weights are created when the model first sees some input data, or if the input size is defined. One way to do this is to pass an input object to the model, so that it knows its input shape, or an input layer may be added that defines the input shape:

```
model = Sequential()
model.add(Flatten())
model.add(keras.Input(shape=(28,28,)))
model.add(Dense(32,))
model.add(Dense(10))
```

At this point, the network is built and a summary may be displayed with

```
model.summary()
```

Question: What is the structure of this network? Be as specific as you can, e.g. number of layers, number of neurons in each layer, output size, activation functions, etc.

A more complicated model is given in the following example.

```
num_classes=10
model = Sequential()
model.add(Input(shape=(28,28,1)))
model.add(Conv2D(32, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(1000, activation='relu'))
model.add(Dense(num_classes, activation='softmax'))
```

Questions

(a) What is the structure of this network? Be as specific as you can, e.g. number of layers, number of neurons in each layer, output size, activation functions, etc.

(b) How many parameters need to be learned?

(c) How would you modify this code so that the outputs of first max-pooling layer are 14×14, which is half the size of the input image?

Once the model has been defined structurally, it must be compiled, which is the next step in your design.

4. Compiling the Model

Before the model can be trained, it must be compiled and the process for training must be configured. Three key factors that should be specified for the compilation step are:

(a) *The optimizer*: Options include `SGD`, `Adagrad`, and `Adam` among others.

(b) *The loss function.* For the MNIST dataset, since there are 10 classes and the softmax function is used in the output layer, then the cross-entropy loss is the one that should be used. Note that Keras distinguishes between `binary_crossentropy` (two classes) and `categorical_crossentropy` (more than two classes). See the Keras documentation for other losses.

(c) *Metrics.* A metric is a function that is used to judge the performance of the model. Available metrics include `accuracy` and `TopKCategoricalAccuracy`.

Here is what a compilation might look like:

```
model.compile(loss=keras.losses.categorical_crossentropy, optimizer='Adam',
              metrics=['accuracy'])
```

Question: There is nothing that prevents us from using a cost function such as the least squares error with the softmax activation function. Why is the cross-entropy loss function preferred?

5. Training the Model

Once a model has been compiled, it may be trained. Training a model in Keras is done by calling the `model.fit()` function and specifying the parameters on how training is to be done. There are many parameters, but some of the key ones are:

(a) *The training set.* In your set-up this will be `x_train` and `y_train`.

(b) *The number of epochs.* The number of passes through the entire dataset.

(c) *The batch size.* The number of data samples that are sent through the network before the gradient is estimated.

(d) *The validation data.* This is used during training to measure the network's performance against data it has not seen.

Here is an example:

```
hist = model.fit(x_train, y_train, batch_size=128, epochs=20,
                 verbose=1, validation_data=(x_test,y_test), callbacks=None)
```

You should look at other options for the validation data. For example, you may use `validation_split` if you want to save your test set for final evaluation (recall the data contamination problem).

Note that the `fit` function returns a `History` object. The `history` attribute is a record of the training loss values and metric values at successive epochs, as well as validation loss values and validation metric values.

The argument `verbose` is an integer, 0, 1, or 2 that sets the verbosity mode, where

```
0 = silent
1 = progress bar
2 = one line per epoch
```

With `verbose=1` something similar to the following will be printed as the training progresses:

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Train on 60000 samples, validate on 10000 samples
Epoch 1/40
60000/60000 [==============================] - 24s 406us/step - loss: 0.2109 - accuracy: 0.9378 - val_loss:
0.0488 - val_accuracy: 0.9854
Epoch 2/40
60000/60000 [==============================] - 24s 405us/step - loss: 0.0490 - accuracy: 0.9856 - val_loss:
0.0325 - val_accuracy: 0.9901
Epoch 3/40
60000/60000 [==============================] - 24s 403us/step - loss: 0.0346 - accuracy: 0.9888 - val_loss:
0.0246 - val_accuracy: 0.9915
Epoch 4/40
60000/60000 [==============================] - 25s 415us/step - loss: 0.0243 - accuracy: 0.9924 - val_loss:
0.0232 - val_accuracy: 0.9921
Epoch 5/40
60000/60000 [==============================] - 23s 390us/step - loss: 0.0176 - accuracy: 0.9946 - val_loss:
0.0239 - val_accuracy: 0.9922
Epoch 6/40
```

```
60000/60000 [==============================] - 25s 409us/step - loss: 0.0143 - accuracy: 0.9956 - val_loss:
0.0244 - val_accuracy: 0.9914
Epoch 7/40
60000/60000 [==============================] - 25s 414us/step - loss: 0.0111 - accuracy: 0.9965 - val_loss:
0.0327 - val_accuracy: 0.9898
Epoch 8/40
60000/60000 [==============================] - 24s 407us/step - loss: 0.0105 - accuracy: 0.9965 - val_loss:
0.0291 - val_accuracy: 0.9912
Epoch 9/40
60000/60000 [==============================] - 24s 400us/step - loss: 0.0085 - accuracy: 0.9973 - val_loss:
0.0257 - val_accuracy: 0.9925
Epoch 10/40
60000/60000 [==============================] - 24s 398us/step - loss: 0.0059 - accuracy: 0.9982 - val_loss:
0.0404 - val_accuracy: 0.9904
```

Note: To simplify experimenting with your model, it is recommended to put the training hyperparameters at the top of your file so that they will be easy to modify. These include the following:

```
batch_size = 256
num_classes = 10
epochs = 40
```

6. Using the Model to Make Predictions and Visualizing the Modeling History

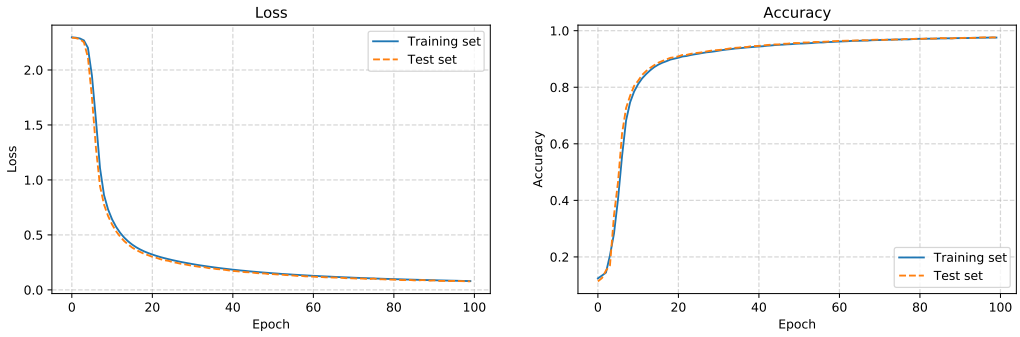
Once the model has been trained, it is possible to predict the classes for new samples,

```
#predict first 4 images in the test set
model.predict(X_test[:4])
```

or visualize the model history by plotting the accuracy or loss on the training and validation sets over the training epochs. For example, the following will create a plot of the training and test accuracy (some of the formatting commands are not included).

```
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
```

A similar plot may be made of the training and test loss. An example is shown in the figure below.



7. Handwritten Digit Recognition Using a Multilayer Perceptron

Having learned how to use Keras to build multilayer perceptrons and convnets, we now look at handwritten digit recognition using MLPs.

Assignment

A simple classifier that may be used for handwritten character recognition is **logistic regression**. In order to see what improvements can be made using a neural network, the first thing to do is see how well logistic regression works.

(a) Design a logistic regression classifier using Keras, and train it on the MNIST training set.

(b) Find the performance of the classifier on the test set.

(c) How many parameters needed to be learned?

Now we turn to the design of an MLP to recognize handwritten digits. As before, the MNIST data set will be used for training. The following exercise is open-ended and unscripted, and you should experiment freely. However, an outline of how you should proceed is given below.

Assignment

(a) Build a **one-hidden-layer** and a **two-hidden-layer** MLP, and train it on the MNIST training set and evaluate its performance on the test set.

(b) Experiment with different numbers of neurons in the hidden layers, and see what kind of performance you are able to get.

(c) Compare your classifiers to the one using logistic regression.

(d) How many neurons are needed in a one hidden layer neural network to get an accuracy of 99%? Or is it not possible to get one that performs this well?

(e) With a two hidden layer network, how small can you make it and still get an accuracy of 99%? How many parameters in this network compared to the one you designed using only one hidden layer?

Assignment

Since there are ten classes to recognize in the handwritten digit classification problem, suppose that you design a symmetric MLP with 10 hidden layers, each with 10 neurons. You program this network as follows:

```
model = Sequential()
model.add(keras.Input(shape=(28,28,)))
model.add(Dense(10,activation='relu'))
model.add(Dense(10,activation='relu'))
model.add(Dense(10,activation='relu'))
model.add(Dense(10,activation='relu'))
model.add(Dense(10,activation='relu'))
model.add(Dense(10,activation='relu'))
model.add(Dense(10,activation='relu'))
model.add(Dense(10,activation='relu'))
model.add(Dense(10,activation='relu'))
model.add(Dense(10,activation='relu'))
```

However, when you compile this MLP you find that there is an error. Fix this error, and any

others that you may find, so that it implements a 10 hidden layer MLP with 10 neurons in each layer.

(a) What is the accuracy of your classifier? Does it perform as well as you would expect? How many parameters does this network need to learn?

(b) Your network most likely does not perform as well as you would like, so increase the number of neurons in each layer from 10 to 32. If you get any errors when you compile this network, find the error and explain what is wrong. If this error has any effect on your design in part (i), go back and fix it, and train that network again.

(c) How well does this network perform, and how many parameters need to be learned?

8. Handwritten Digit Recognition Using a Convolutional Neural Network

Now we will look at CNNs to see what performance improvements are possible. Again, here the exercise is open-ended and unscripted. However, an outline of how you should proceed is given below.

Assignment

Instead of a multilayer perception, consider using a convolutional neural network for your digit recognition system.

Using a number of different CNN structures, and experimenting with options such as dropout, normalization, and one or more dense layers, determine the most efficient structure for achieving a recognizer with recognition rates close to 99%. Keep in mind that you would like a structure as simple as possible (in terms of the number of layers and the number of parameters to learn) that achieves your design goals. The deeper the network, the longer it will take to perform a recognition task. The larger the number of parameters, the higher the chances of overfitting, the more difficult it may be to train, and the more computational requirements will be needed for classification.

Questions

(1) What is the effect of the batch size on training time and on the accuracy of the model?

(2) What is a dense layer? Are one or more dense layers necessary?

(3) What is batch normalization and is it important? Does it have any effect in your training algorithm?

(4) What is dropout, and is it necessary for your designs?

(5) Compare the number of parameters used to the number you had for the fully-connected networks.