A3

Sean Pavlak

Mar 30 2018

1 Exercise 1

Configure your package and work through a simple demo using a network with one hidden layer and fully connected weights. This will be used as a baseline and should be as simple as possible. Construct a larger neural network and train it on the same dataset. Contrast the performance with the baseline network.

For this exercise I set out to iteratively improve the Convolutional Neural Network, CNN, shown in the demo. Before I could work on that I first needed to design a way to implement a CNN using keras efficiently so that I could easily change one thing at a time and determine if the resulting CNN was an improvement.

Once this was implemented I was able to simply add or remove layers in order to optimize my CNN as much as possible.

In this exercise I utilized the Keras library. Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano. It was developed with a focus on enabling fast experimentation. My approach was to take this library and condense it down into a single executable function and make small incremental changes to my layers in order to develop the fastest and most accurate CNN possible.

The dataset shown in the demo, the MNIST dataset, is the same dataset I chose to train my CNNs. This dataset is a large database of handwritten digits containing 60,000 training images and 10,000 testing images.

1.1 Code Description

I was able to create an iterative approach by defining a single function, fit_score_mnist_data, and passing in the feature layers, classification

layers, the number of iterations or epochs, a loss function, and an optimization function.

```python
def fit_score_mnist_data(feature_layers, classification_layers,
↪ epochs, loss, optimizer):
    (x_train, y_train), (x_test, y_test) = load_mnist_data()

    model = Sequential(feature_layers + classification_layers)

    model.compile(
    loss=loss,
    optimizer=optimizer,
    metrics=['accuracy'])

    model.fit(
    x_train, y_train,
    epochs=epochs,
    batch_size=batch_size,
    verbose=1,
    validation_data=(x_test, y_test))

    score = model.evaluate(x_test, y_test, verbose=0)
    print('Test loss:', score[0])
    print('Test accuracy:', score[1])

    return score, model
```

The function works by first loading in the MNIST dataset by utilizing the other helper function I wrote, load_mnist_data. This function returns the training and test data. I then pass my layers into the Sequential function defined by the Keras library, which is how I define my model.

Keras then requires the model to be compiled prior to fitting it with the training data. This requires a loss and optimizer function, which are arguments to the fit_score_mnist_data function as they are one of the variables we test.

```python
def load_mnist_data():
    # the data, split between train and test sets
    (x_train, y_train), (x_test, y_test) = mnist.load_data()

    if K.image_data_format() == 'channels_first':
        x_train = x_train.reshape(x_train.shape[0], 1, img_rows,
         ↪  img_cols)
        x_test = x_test.reshape(x_test.shape[0], 1, img_rows,
         ↪  img_cols)
        input_shape = (1, img_rows, img_cols)
    else:
        x_train = x_train.reshape(x_train.shape[0], img_rows,
         ↪  img_cols, 1)
        x_test = x_test.reshape(x_test.shape[0], img_rows,
         ↪  img_cols, 1)
        input_shape = (img_rows, img_cols, 1)

    x_train = x_train.astype('float32')/255
    x_test = x_test.astype('float32')/255

    print('x_train shape:', x_train.shape)
    print(x_train.shape[0], 'train samples')
    print(x_test.shape[0], 'test samples')

    # convert class vectors to binary class matrices
    y_train = keras.utils.to_categorical(y_train, num_classes)
    y_test = keras.utils.to_categorical(y_test, num_classes)

    return (x_train, y_train), (x_test, y_test)
```

Once these were defined, training our CNNs were as easy as defining our feature and classification layers and picking a loss and optimization function. Once those are defined and passed into the fit_score_mnist_data function will also evaluate the trained model and return a score. This score and the time elapsed in training is what I am using to determine the efficiency of the function.

The first thing I needed to do was reproduce the CNN shown in the demo. This was as simple as inputting the used layers into arrays and defining the loss function and optimization function to match the functions used in the demo. Then running the fit_score_mnist_data function.

```
feature_layers = [
    Conv2D(32, kernel_size=(3, 3), activation='relu',
     ↪  input_shape=(28, 28, 1)),
    MaxPooling2D(pool_size=(2, 2)),
    Flatten()
]

classification_layers = [
    Dense(num_classes, activation='softmax')
]

loss = keras.losses.categorical_crossentropy
optimizer = keras.optimizers.Adadelta()
epochs = 1

score = fit_score_mnist_data(feature_layers,
↪  classification_layers, epochs, loss, optimizer)
```

This function was able to produce some decent results, shown in the results section. However, there is always room to improve.

In order to improve the CNN I needed to see what worked and what did not experimentally. Right off the bat I added a second convolution filter and implemented a second max pooling function as well. Along with implementing a couple dropout functions to avoid over fitting. Regarding the classification layer I simply added a second dense layer and implemented another dropout function.

Just like the demo I kept the cross entropy loss function and the adadelta optimization function. This also only needed a single epoch to train the model efficiently.

```
feature_layers = [
    Conv2D(32, kernel_size=(3, 3), activation='relu',
    ↪  input_shape=(28, 28, 1)),
    Conv2D(64, (3, 3), activation='relu'),
    MaxPooling2D(pool_size=(2, 2)),
    Dropout(0.25),
    MaxPooling2D(pool_size=(2,2)),
    Dropout(.25),
    Flatten()
]

classification_layers = [
    Dense(128, activation='relu'),
    Dropout(0.5),
    Dense(num_classes, activation='softmax')
]

```

```
loss = keras.losses.categorical_crossentropy
optimizer = keras.optimizers.Adadelta()
epochs = 1

score = fit_score_mnist_data(feature_layers,
↪  classification_layers, epochs, loss, optimizer)
```

This is where I decided to start experimenting. I decided to test the effect of the optimization function. Previously I had been using the adadelta optimization function, which is a per-dimension learning rate method for gradient descent. This method dynamically adapts over time using only first order information and has minimal computational overhead. The method requires no manual tuning of a learning rate and appears robust to noisy gradient information, different model architecture choices, various data modalities and selection of hyperparameters. However, I decided to see how it compared to stochastic gradient descent, SGD.

SGD is simply an optimizer which includes support for momentum, learning rate decay, and Nesterov momentum.

```
feature_layers = [
    Conv2D(32, (3, 3), padding='same', input_shape=(28, 28, 1)),
    Activation('relu'),
    Conv2D(32, (3, 3), padding='same'),
    Activation('relu'),
    MaxPooling2D(pool_size=(2,2)),
    Dropout(.25),
    Flatten()
]

classification_layers = [
    Dense(128),
    Activation('relu'),
    Dropout(.50),
    Dense(num_classes),
    Activation('softmax')
]

loss=keras.losses.categorical_crossentropy
optimizer=SGD(clipnorm=10000, clipvalue=10000)
epochs = 1

```

```
score = fit_score_mnist_data(feature_layers,
    classification_layers, epochs, loss, optimizer)
```

I then tested the effectiveness of the activation function. Previously we used the rectified linear unit, relu, activation function, which is a smooth approximation to the softplus function.

$$f(x) = \log(1 + \exp x) \tag{1}$$

For this iteration I implimented a new feature layer, average pooling, with the intent of it improving results due to being an ideal layer for training on spatial data. I also changed our activation function from relu to a sigmoid activation function. Which is generally applied to train a model over many epochs, starting with poor results then gradually approaching an asymptotic climax over time.

$$S(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{e^x + 1} \tag{2}$$

This asymptotic approach to training requires a large degree of iterations, or epochs. So I arbitrarily selected 30 in order to guarantee that the model was trained by the end of the cycle.

```
feature_layers = [
    Conv2D(32, (3, 3), padding='same', input_shape=(28, 28, 1)),
    Activation('sigmoid'),
    Conv2D(32, (3, 3), padding='same'),
    Activation('sigmoid'),
    AveragePooling2D(pool_size=(2,2)),
    Dropout(.25),
    Flatten()
]

classification_layers = [
    Dense(128),
    Activation('sigmoid'),
    Dropout(.50),
    Dense(num_classes),
    Activation('softmax')
]

loss=keras.losses.categorical_crossentropy
optimizer=SGD(clipnorm=10000, clipvalue=10000, lr=0.1,
↪  momentum=0.9)
epochs = 30

score = fit_score_mnist_data(feature_layers,
↪  classification_layers, epochs, loss, optimizer)
```

The last change that I performed was to change the loss function that we have been using throughout the exercise, categorical cross entropy. This loss function is the number of bits we'll need if we encode symbols from one set using the wrong tool of another set. I changed this to use a mean square error loss function.

```
feature_layers = [
    Conv2D(32, (3, 3), padding='same', input_shape=(28, 28, 1)),
    Activation('sigmoid'),
    Conv2D(32, (3, 3), padding='same'),
    Activation('sigmoid'),
    AveragePooling2D(pool_size=(2,2)),
    Dropout(.25),
    Flatten()
]

classification_layers = [
    Dense(128),
    Activation('sigmoid'),
    Dropout(.50),
    Dense(num_classes),
    Activation('softmax')
]

loss='mean_squared_error'
optimizer=SGD(clipnorm=10000, clipvalue=10000, lr=0.1,
↪  momentum=0.9)
epochs = 30

score = fit_score_mnist_data(feature_layers,
↪  classification_layers, epochs, loss, optimizer)
```

1.2 Results

Demo Results:

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Train on 60000 samples, validate on 10000 samples
Epoch 1/1
60000/60000 [==============================] - 100s 2ms/step -
 ↪  loss: 0.2136 - acc: 0.9352 - val_loss: 0.0510 - val_acc:
 ↪  0.9836
Test loss: 0.0510165721905
Test accuracy: 0.9836
```

This model took about 100 s to train and was able to achieve an accuracy score of about 98.4%. This is a decent baseline for model efficiency, but can definitely be improved upon.

This result differed significantly from the result shown in the demo; however, this demo used 3 epochs and validated the data differently. In the demo the score to beat was 85.8%. Which was improved upon simply by performing the fit with a different set of validation data.

Improved Results:

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Train on 60000 samples, validate on 10000 samples
Epoch 1/1
60000/60000 [==============================] - 11s 182us/step -
 ↪  loss: 0.2159 - acc: 0.9370 - val_loss: 0.0966 - val_acc:
 ↪  0.9733
Test loss: 0.096591734235
Test accuracy: 0.9733
```

This improved model took about 11 s to train and was able to achieve a slightly smaller accuracy score of about 97.3%. While I did not achieve as high an accuracy score as the demo in this iteration, I was able to reduce the training time by a tenth. Which means that this model has an efficiency of about 8.9% per second. Granting it the most efficient model tested.

SGD Results:

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
```

```
Train on 60000 samples, validate on 10000 samples
Epoch 1/1
60000/60000 [==============================] - 93s 2ms/step -
 ↪  loss: 0.6740 - acc: 0.7819 - val_loss: 0.2118 - val_acc:
 ↪  0.9380
Test loss: 0.211838701645
Test accuracy: 0.938
```

After implementing a SGD optimization function, this model took about 93 s to train and was able to achieve a significantly smaller accuracy score of about 93.8%. Granting it a significantly smaller efficiency value than both previous models.

Sigmoid Activation and Average Pooling Results:

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 85s 1ms/step -
 ↪  loss: 2.3561 - acc: 0.1016 - val_loss: 2.3036 - val_acc:
 ↪  0.1032
Epoch 2/30
60000/60000 [==============================] - 85s 1ms/step -
 ↪  loss: 2.3084 - acc: 0.1035 - val_loss: 2.3246 - val_acc:
 ↪  0.1135
Epoch 3/30
60000/60000 [==============================] - 75s 1ms/step -
 ↪  loss: 2.3088 - acc: 0.1038 - val_loss: 2.3075 - val_acc:
 ↪  0.1135

...

Epoch 29/30
60000/60000 [==============================] - 81s 1ms/step -
 ↪  loss: 0.0807 - acc: 0.9745 - val_loss: 0.0422 - val_acc:
 ↪  0.9869
Epoch 30/30
60000/60000 [==============================] - 86s 1ms/step -
 ↪  loss: 0.0764 - acc: 0.9759 - val_loss: 0.0382 - val_acc:
 ↪  0.9880
Test loss: 0.0382246035573
Test accuracy: 0.988
```

After implementing a sigmoid activation and average pooling layers, this model took about 2,400 s to train and was able to achieve

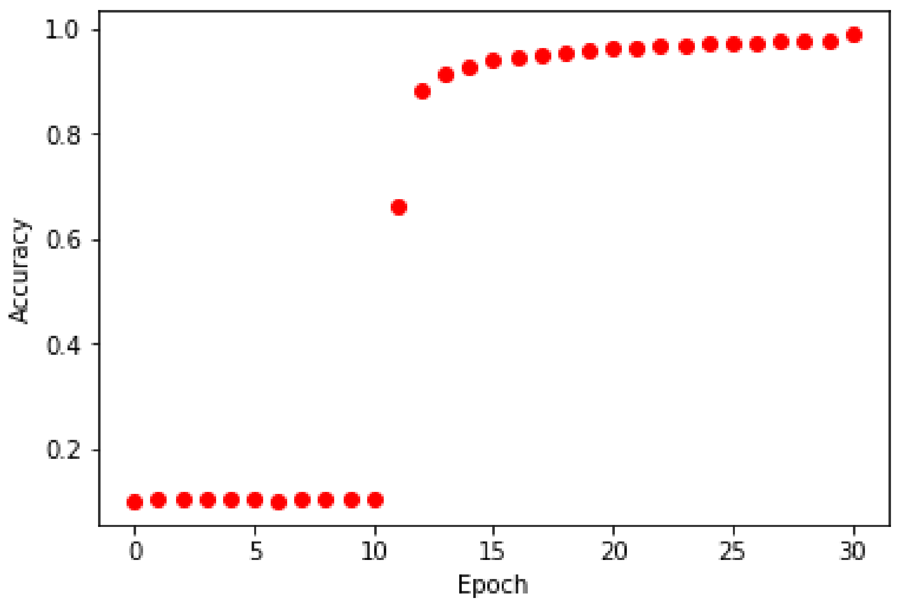

Figure 1: As shown in the figure above it appears that the model's accuracy reaches its asymptotic maximum around 26 epochs.

an accuracy score of about 98.8%. Also granting it a significantly smaller efficiency value, but increasing the total accuracy by about 0.5%.

It is also important to note that this model's accuracy spiked at 13 epochs and reached their approximate maximum around 26 epochs. These values would directly effect how we measure the efficiency of this model.

MSE Loss Results:

```
x_train shape: (60000, 28, 28, 1)
60000 train samples
10000 test samples
Train on 60000 samples, validate on 10000 samples
Epoch 1/30
60000/60000 [==============================] - 84s 1ms/step -
↪  loss: 0.0901 - acc: 0.1068 - val_loss: 0.0900 - val_acc:
↪  0.1135
Epoch 2/30
```

```
60000/60000 [==============================] - 85s 1ms/step -
↪  loss: 0.0900 - acc: 0.1120 - val_loss: 0.0900 - val_acc:
↪  0.1135
Epoch 3/30
60000/60000 [==============================] - 84s 1ms/step -
↪  loss: 0.0900 - acc: 0.1124 - val_loss: 0.0900 - val_acc:
↪  0.1135

...

Epoch 29/30
60000/60000 [==============================] - 81s 1ms/step -
↪  loss: 0.0807 - acc: 0.8918 - val_loss: 0.0422 - val_acc:
↪  0.9869
Epoch 30/30
60000/60000 [==============================] - 86s 1ms/step -
↪  loss: 0.0764 - acc: 0.9050 - val_loss: 0.0382 - val_acc:
↪  0.9880
Test loss: 0.0382246035573
Test accuracy: 0.9351
```

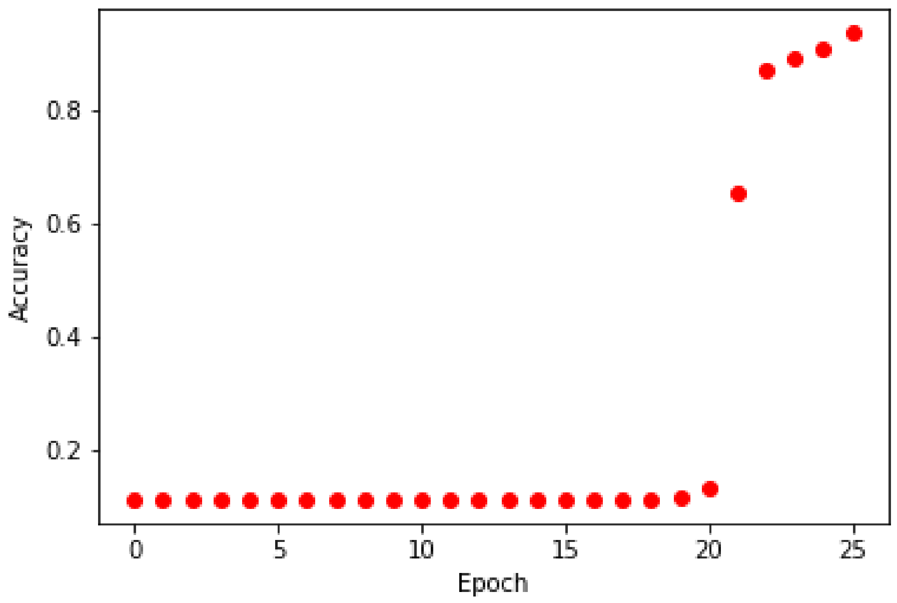

Figure 2: As shown in the figure above it appears that the model's accuracy reaches its asymptotic maximum around 25 epochs.

After implementing a mean square error loss function, this model

also took about 2,400 s to train and was able to achieve an accuracy score of about 93.5%. Also granting it a significantly smaller efficiency value, but unlike the other sigmoid model this model's efficiency could not be overlooked by achieving a high accuracy.

It is also important to note that this model's accuracy spiked around 25 epochs where it met its approximate maximum. These values would directly effect how we measure the efficiency of this model, while still presenting a low efficiency this effect is still necessary to consider.

After all of this it appears that the two ideal choices are to either take the improved demo model, with a fast training time and high accuracy, or take the first sigmoid model as it was the only one to achieve an accuracy of 98.8%.

In my opinion I would take the efficient model as it performed quickly and did not sacrifice accuracy to a significant degree.