ECE472 – Project 3

Jonathan Lam

September 22, 2020

Notes

- The keras library and its builtin utilities were used for this project. kerastuner was used for automatic tuning. (It has a very appealing logging output.) As expected, the final output takes quite a while to iterate over the search space, but consistently produces the desired results (classification accuracy $> 95.5\%$).
- The MNIST dataset files were downloaded from `http://yann.lecun.com/exdb/mnist/` and parsed with ordinary file operations.
- The train dataset (60,000 images) was randomly split into 50,000 train/10,000 validation.
- The neural network comprises some hidden layers and a final softmax (logit) layer. The `tf.keras.losses.SparseCategoricalCrossEntropy` loss was used to evaluate the function and calculate the gradient. The `tf.keras.optimizers.Adam` optimizer was used. Each hidden layer included a dense linear layer with L^2 regularization, batch normalization, a leaky ReLU activation function, and dropout (following the activation function).
- Tunable parameters include:
  - Number of hidden layers
  - Width (of each hidden layer)
  - L^2 penalization coefficient (for each hidden layer)
  - Adam optimizer learning rate

Source Code

The final tuned trained model generated by this code produces 97.60% accuracy on the test dataset. (This can be seen at the bottom of the code output in the following section.)

```
### Jonathan Lam
### Prof. Curro
### ECE 472 Deep Learning
### 9 / 20 / 20
### Project 3

### MNIST data from http://yann.lecun.com/exdb/mnist/
```

```
import numpy as np
import tensorflow as tf
import kerastuner as kt

# 32-bit big-endian byte buffer to int
def be32_to_int(buf):
    return np.ndarray(shape=(1,), dtype='>i4', buffer=buf)[0]

# see URL of MNIST data source for data layout
def read_mnist_file(filename, is_labels):
    with open(filename, 'rb') as mnist_file:
        # read header info
        be32_to_int(mnist_file.read(4))     # read and discard magic number
        num_imgs = be32_to_int(mnist_file.read(4))
        num_rows = None if is_labels else be32_to_int(mnist_file.read(4))
        num_cols = None if is_labels else be32_to_int(mnist_file.read(4))

        # read raw image data
        data = np.frombuffer(mnist_file.read(), dtype=np.uint8)
        return num_imgs, num_rows, num_cols, data

# read train/validation dataset files
num_train_file_imgs, num_rows, num_cols, train_file_imgs = \
    read_mnist_file('train-images-idx3-ubyte', False)
_, _, _, train_file_lbls = read_mnist_file('train-labels-idx1-ubyte', True)

# read test dataset files
num_test, _, _, test_imgs = read_mnist_file('t10k-images-idx3-ubyte', False)
_, _, _, test_lbls = read_mnist_file('t10k-labels-idx1-ubyte', True)

# P := # features (pixels) = num_rows x num_cols
# N := num_imgs
# reshape features to NxP, scale to [0, 1)
train_file_imgs = train_file_imgs.reshape(-1, num_rows * num_cols) / 255.
test_imgs = test_imgs.reshape(-1, num_rows * num_cols) / 255.
# reshape labels to Nx1
train_file_lbls = train_file_lbls.reshape(-1, 1)
test_lbls = test_lbls.reshape(-1, 1)

# split train/validation dataset into train and validation datasets
indices = np.arange(num_train_file_imgs)
np.random.shuffle(indices)
cutoff = 50000
train_imgs, val_imgs, train_lbls, val_lbls = \
    train_file_imgs[indices[:cutoff],:], \
    train_file_imgs[indices[cutoff:],:], \
    train_file_lbls[indices[:cutoff],:], \
    train_file_lbls[indices[cutoff:],:]

# tunable model builder
def build_model(hp):
    model = tf.keras.models.Sequential()
    model.add(tf.keras.Input(shape=(num_rows * num_cols,)))

    # tunable hidden layers
    for i in range(hp.Int('layers', 3, 5)):
        model.add(tf.keras.layers.Dense(
```

```
            units=hp.Choice('units' + str(i), [64, 128, 256]),
            kernel_regularizer=tf.keras.regularizers.L1L2(
                l2=hp.Float('l2_' + str(i), 1e-8, 1e-4, sampling='log'))))
        model.add(tf.keras.layers.BatchNormalization())
        model.add(tf.keras.layers.LeakyReLU())
        model.add(tf.keras.layers.Dropout(
            rate=hp.Float('dropout_' + str(i), 0.1, 0.5, step=0.2)))

    # final layer: calculate logits
    model.add(tf.keras.layers.Dense(
        units=10,
        kernel_regularizer=tf.keras.regularizers.L1L2(
            l2=hp.Float('l2_final', 1e-8, 1e-4, sampling='log'))))

    # set up model loss and optimizer
    model.compile(
        optimizer=tf.keras.optimizers.Adam(
            learning_rate=hp.Float('learning_rate', 1e-4, 1e-1, sampling='log')),
        loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
        metrics=['accuracy'])

    return model

# create kerastuner hyperband tuner
tuner = kt.Hyperband(build_model, objective='val_acc', max_epochs=10)

# print search space summary
tuner.search_space_summary()

# search for best hyperparameters, evaluate on validation set
tuner.search(train_imgs, train_lbls, validation_data=(val_imgs, val_lbls))

# print best result
tuner.results_summary()

# train model with best hyperparameters
best_hps = tuner.get_best_hyperparameters(num_trials=1)[0]
model = tuner.hypermodel.build(best_hps)
model.fit(train_imgs, train_lbls, epochs=10)

# evaluate model on test dataset
print('Evaluating on test dataset:')
model.evaluate(test_imgs, test_lbls, verbose=2)
```

Sample Output

All but the final kerastuner trial is omitted for brevity. This is a "sample" because the validation and training sets were randomly partitioned from the official MNIST training data file.

```
/usr/bin/python3.7 /home/jon/Documents/ece472/proj3/proj3.py
WARNING:tensorflow:From /usr/local/lib/python3.7/dist-packages/tensorflow/python/ops/
    ↪ init_ops.py:1251: calling VarianceScaling.__init__ (from tensorflow.python.ops.
    ↪ init_ops) with dtype is deprecated and will be removed in a future version.
Instructions for updating:
Call initializer instance with the dtype argument instead of passing it to the
    ↪ constructor
```

```
[Search space summary]
 |-Default search space size: 12
 > layers (Int)
 |-default: None
 |-max_value: 5
 |-min_value: 3
 |-sampling: None
 |-step: 1
 > units0 (Choice)
 |-default: 64
 |-ordered: True
 |-values: [64, 128, 256]
 > l2_0 (Float)
 |-default: 1e-08
 |-max_value: 0.0001
 |-min_value: 1e-08
 |-sampling: log
 |-step: None
 > dropout_0 (Float)
 |-default: 0.5
 |-max_value: 0.5
 |-min_value: 0.1
 |-sampling: None
 |-step: 0.2
 > units1 (Choice)
 |-default: 64
 |-ordered: True
 |-values: [64, 128, 256]
 > l2_1 (Float)
 |-default: 1e-08
 |-max_value: 0.0001
 |-min_value: 1e-08
 |-sampling: log
 |-step: None
 > dropout_1 (Float)
 |-default: 0.5
 |-max_value: 0.5
 |-min_value: 0.1
 |-sampling: None
 |-step: 0.2
 > units2 (Choice)
 |-default: 64
 |-ordered: True
 |-values: [64, 128, 256]
 > l2_2 (Float)
 |-default: 1e-08
 |-max_value: 0.0001
 |-min_value: 1e-08
 |-sampling: log
 |-step: None
 > dropout_2 (Float)
 |-default: 0.5
 |-max_value: 0.5
 |-min_value: 0.1
 |-sampling: None
 |-step: 0.2
 > l2_final (Float)
 |-default: 1e-08
```

```
 |-max_value: 0.0001
 |-min_value: 1e-08
 |-sampling: log
 |-step: None
 > learning_rate (Float)
 |-default: 0.0001
 |-max_value: 0.1
 |-min_value: 0.0001
 |-sampling: log
 |-step: None

... keras tuning output truncated ...

[Trial summary]
 |-Trial ID: 901d97905dbd0f468748596d357a3183
 |-Score: 0.9519000053405762
 |-Best step: 0
 > Hyperparameters:
 |-dropout_0: 0.30000000000000004
 |-dropout_1: 0.30000000000000004
 |-dropout_2: 0.30000000000000004
 |-dropout_3: 0.30000000000000004
 |-dropout_4: 0.30000000000000004
 |-l2_0: 8.353674406062614e-06
 |-l2_1: 1.1390330286140142e-05
 |-l2_2: 9.532844285293838e-08
 |-l2_3: 1.2916292421935892e-08
 |-l2_4: 1.6675587436184615e-05
 |-l2_final: 8.072807530434067e-07
 |-layers: 5
 |-learning_rate: 0.00044957516806020043
 |-tuner/bracket: 1
 |-tuner/epochs: 4
 |-tuner/initial_epoch: 0
 |-tuner/round: 0
 |-units0: 128
 |-units1: 128
 |-units2: 256
 |-units3: 128
 |-units4: 128
Train on 50000 samples, validate on 10000 samples
Epoch 1/10
50000/50000 [==============================] - 10s 202us/sample - loss: 0.6770 - acc:
    ↪ 0.8044 - val_loss: 0.2488 - val_acc: 0.9309
Epoch 2/10
50000/50000 [==============================] - 10s 194us/sample - loss: 0.3694 - acc:
    ↪ 0.8967 - val_loss: 0.1911 - val_acc: 0.9472
Epoch 3/10
50000/50000 [==============================] - 10s 194us/sample - loss: 0.2932 - acc:
    ↪ 0.9178 - val_loss: 0.1636 - val_acc: 0.9555
Epoch 4/10
50000/50000 [==============================] - 10s 194us/sample - loss: 0.2540 - acc:
    ↪ 0.9301 - val_loss: 0.1448 - val_acc: 0.9600
Epoch 5/10
50000/50000 [==============================] - 10s 193us/sample - loss: 0.2217 - acc:
    ↪ 0.9382 - val_loss: 0.1321 - val_acc: 0.9637
Epoch 6/10
50000/50000 [==============================] - 10s 194us/sample - loss: 0.1974 - acc:
```

```
    ↪ 0.9454 - val_loss: 0.1189 - val_acc: 0.9667
Epoch 7/10
50000/50000 [==============================] - 10s 194us/sample - loss: 0.1862 - acc:
    ↪ 0.9482 - val_loss: 0.1188 - val_acc: 0.9658
Epoch 8/10
50000/50000 [==============================] - 10s 194us/sample - loss: 0.1718 - acc:
    ↪ 0.9520 - val_loss: 0.1143 - val_acc: 0.9673
Epoch 9/10
50000/50000 [==============================] - 10s 194us/sample - loss: 0.1621 - acc:
    ↪ 0.9550 - val_loss: 0.1053 - val_acc: 0.9711
Epoch 10/10
50000/50000 [==============================] - 10s 194us/sample - loss: 0.1507 - acc:
    ↪ 0.9578 - val_loss: 0.1051 - val_acc: 0.9705
Evaluating on test dataset:
10000/10000 - 1s - loss: 0.0904 - acc: 0.9760
```

80% classification accuracy with fewer parameters

Attempt 1

A single softmax layer without any regularization (the rest of the program and the model are unchanged) produced 92.65% accuracy. The number of weights is

$$\text{num\_weights} = \text{size } W_{dense} + \text{size } b_{dense} = (28 \times 28) \times 10 + 10 = 7850$$

```
model = tf.keras.models.Sequential()
model.add(tf.keras.Input(shape=(num_rows * num_cols,)))
model.add(tf.keras.layers.Dense(units=10))
```

I was able to check the number of weights using

```
np.array([w.size for layer in model.layers for w in layer.get_weights()]).sum())
```

Attempt 2

By performing an average pooling (5x5 pools with stride 5), I was able to maintain a 82.6% accuracy and greatly reduce the number of weights.

$$\text{num\_weights} = \text{size } W_{dense} + \text{size } b_{dense} = (5 \times 5) \times 10 + 10 = 260$$

```
# reshape inputs to allow for pooling correctly
train_imgs = train_imgs.reshape(-1, num_rows, num_cols, 1) / 255.
test_imgs = test_imgs.reshape(-1, num_rows, num_cols, 1) / 255.

# ...

model.add(tf.keras.layers.AveragePooling2D(pool_size=(5,5)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=10))
```

Attempt 3

By adding a convolutional layer, I was able to increase the pool sizes and further decrease the number of weights:

$$\text{num\_weights} = \text{filters} \times (\text{filter\_size}+1) + \text{size } W_{dense} + \text{size } b_{dense} = 2 \times (5 \times 5 + 1) + 8 \times 10 + 10 = 142$$

This achieved an accuracy of 85.25%.

```
# reshape inputs to allow for pooling correctly
train_imgs = train_imgs.reshape(-1, num_rows, num_cols, 1) / 255.
test_imgs = test_imgs.reshape(-1, num_rows, num_cols, 1) / 255.

# ...

model.add(tf.keras.layers.Conv2D(filters=2, kernel_size=(5,5)))
model.add(tf.keras.layers.MaxPool2D(pool_size=(9,9)))
model.add(tf.keras.layers.Flatten())
model.add(tf.keras.layers.Dense(units=10,
          kernel_regularizer=tf.keras.regularizers.L1L2(l2=0.0001)))
```

Other attempts

To try to reduce the number of features, I tried downsizing and center-cropping the images. I also tried multiple small convolutional layers. Neither of these methods really got the number of weights to decrease much more than in Attempt 3.

Unattempted attempts

If I had more time to spare on this homework, I think the goal would be to use some complex hardcoded filters that would make each of the numbers linearly spaced after a convolution and filtering layer, so that a regression can be performed to classify the data (since this requires much fewer parameters in the final step than a softmax). At this point however, the point might be moot because the filters would not be learned (although we could still learn the weights for the final dense layer).