HomeWork 2: Optimization & Regularization

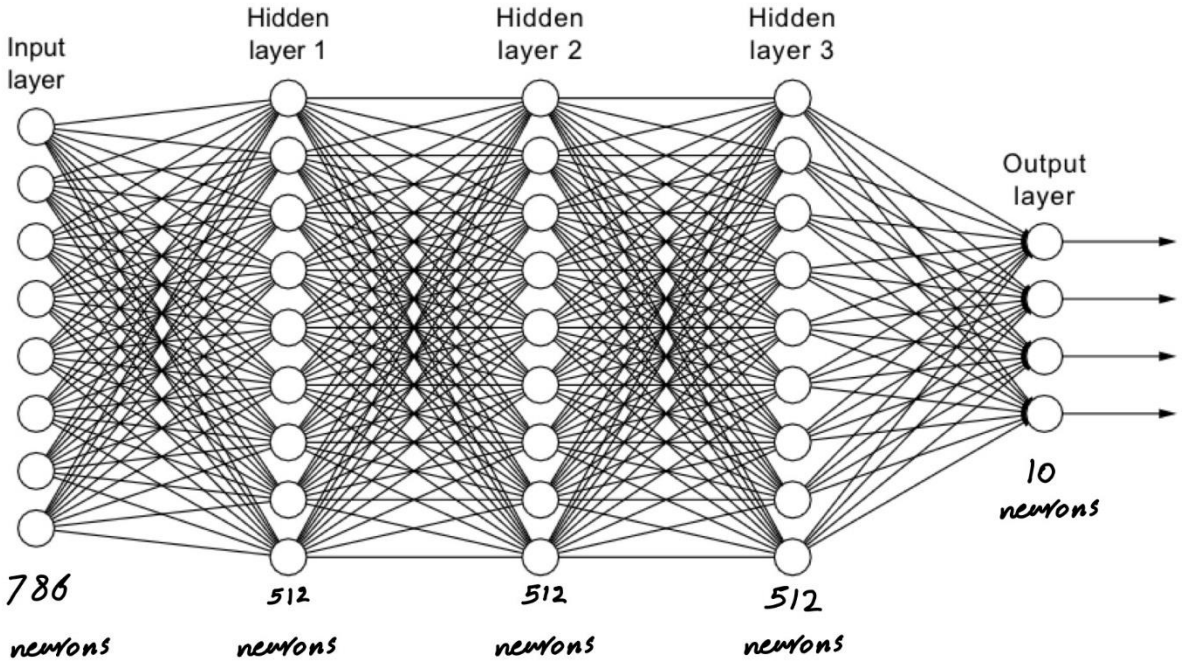
Introduction

Model hyperparameters are the properties that governs the entire training process. They include variables which determines the network structure (for example, Number of Hidden layers) and the variables which determine how the network is trained (for example, Learning Rate). These parameters are important because they control behavior of the training algorithm, having important impact on performance of the model under training.

Choosing appropriate hyperparameters plays a key role in the success of neural network architectures, given the impact on the learned model. For instance, if the learning rate is too low, the model will miss the important patterns in the data; conversely, if it is high, it may have collisions. In this assignment the aim is to compare between two hyperparameter-selection methods, grid search and randomized search. the testing environment is a 3-layer neural network which will use MNIST Handwritten Digit Classification.

The Architecture of Network

It generally a good practice to start with a basic neural network and build on it as it grows, therefore we'll continue to use the basic 5 layer neural network in HomeWork 1. The aim in this assignment is to continue improving the accuracy of our neural network by focusing on obtaining the best parameters using two hyperparameter tuning algorithms. Moreover, the machine hardware and the time value limit constraints us with the scale of the network to test on.



Experiment Setup

The neural network is using the Keras interface, and it consist of 5 layers (1 input, 3 hidden, 1 output) with 512 neurons for each hidden layer. The MNIST handwritten data is first flattened to 784 input neurons and then fed to the hidden layers, the final output layer consists of 10 neurons each correspond to its associated digit. The parameters that will be optimized are batch size, optimizer, drop out, and activation function.

The code will start by loading the data into x_train, y_train, x_test, y_test and proceed to flatten them and limit their value between 0 and 1, corresponding the pixel level (0 is black 1 is white). The model defined is then pass to KerasClassifer which is a wrapper that will allow us to pass the model into sci-learn search algorithm. The search pool parameters are defined as seen in the screenshot below.

```
batch_size = [80, 120, 160, 180, 220]
optimizer = ["SGD", "adam", "RMSprop", "Adadelta"]
dropout = [0.0, 0.1, 0.2, 0.3]
activation = ['relu', 'sigmoid', 'tanh', 'hard_sigmoid']

search_pool = dict(batch_size=batch_size, optimizer=optimizer, dropout=dropout
, activation =activation )
```

For performance testing, the data re-splitted again using train_test_split method from Keras. The test size chosen is 33% and 67% for training. The best parameters are then passed to the model for evaluation run as shown in the code below.

```
print("Best: %f using %s" % (grid_result.best_score_, grid_result.best_params_
))
(x_train, y_train), (x_test, y_test) = mnist.load_data()

x = np.concatenate((x_train, x_test))
y = np.concatenate((y_train, y_test))
x_train, x_test, Y_train, Y_test = train_test_split(x, y, test_size=0.33, rand
om_state=1)
Y_train= to_categorical(Y_train, 10)
Y_test= to_categorical(Y_test, 10)
```

Finally, after re-splitting the data the best parameters are then fed into the same model and evaluated using matplot library as seen below.

```
params = grid_result.best_params_


model = create_model(optimizer=params["optimizer"],activation=params["activati
on"], dropout=params["dropout"])




history = model.fit(x_train,
                    Y_train,
                    batch_size=params["batch_size"],
                    epochs=40,
                    verbose = 2,
                    validation_data=(x_test, Y_test)
                    )
model.summary()
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()


plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

The Grid Search

Grid search defines a search space as a grid of hyperparameter values and evaluate every position in the grid. It is great for spot-checking combinations that are known to perform well generally.

In order to implement grid search we must first import sci-learn library that will provide us with GridSearchCV method and KerasClassifer that we will need to pass the model. The grid search parameters used include some common choices for similar networks, with their values selected based on the size of data (60,000) taken into consideration. The grid search is defined as following:

Batch_size = [80, 120, 160, 180, 220]
Optimizer = [‘SGD’, ‘adam’, ‘RMSprop’, ‘Adadelta’]
Dropout = [0.0, 0.1, 0.2, 0.3]
Activation= [‘relu’, ‘sigmoid’, ‘tanh’, ‘hard_sigmoid’]

```
def create_model(optimizer="adam", activation="relu", dropout=0.2):

    model=keras.Sequential()

    model.add(Dense(512, input_shape=(784,), activation=activation))
    model.add(Dropout(dropout))
    model.add(Dense(512, activation=activation))
    model.add(Dropout(dropout))
    model.add(Dense(512, activation=activation))
    model.add(Dropout(dropout))


    model.add(Dense(10, activation='softmax'))



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


        )
    return model
model = KerasClassifier(build_fn=create_model)
#define the grid search parameter
batch_size = [80, 120, 160, 180, 220]
optimizer = ["SGD", "adam", "RMSprop", "Adadelta"]
dropout = [0.0, 0.1, 0.2, 0.3]
activation = ['relu', 'sigmoid', 'tanh', 'hard_sigmoid']
#creating a Dictionary
param_grid = dict(batch_size=batch_size, optimizer=optimizer, dropout=dropout, activation =activation
)

grid = GridSearchCV(estimator=model, param_grid=param_grid, cv= 10, return_train_score=True, n_jobs=-
1)
grid_result = grid.fit(x_train, Y_train)
```

The Randomized Search

Similar to grid search, RandomizedSearchCV method requires importing sci-learn library. However, randomized search defines a search space as a bounded domain of hyperparameter values and randomly sample points in that domain. This means that it is great for discovery of hyperparameter combinations that are not preemptively set in the search grid.

In this model we need to define a search pool that the method will randomly select values from to determine the best combination, therefore the same optimization parameters are used. Only difference is changing the method to RandomizedSearchCV as seen in the screenshot below.

```
def create_model(optimizer="adam", activation="relu", dropout=0.2):

    model=keras.Sequential()

    model.add(Dense(512, input_shape=(784,), activation=activation))
    model.add(Dropout(dropout))
    model.add(Dense(512, activation=activation))
    model.add(Dropout(dropout))
    model.add(Dense(512, activation=activation))
    model.add(Dropout(dropout))


    model.add(Dense(10, activation='softmax'))



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


        )
    return model
model = KerasClassifier(build_fn=create_model)
#define the grid search parameter
batch_size = [80, 120, 160, 180, 220]
optimizer = ["SGD", "adam", "RMSprop", "Adadelta"]
dropout = [0.0, 0.1, 0.2, 0.3]
activation = ['relu', 'sigmoid', 'tanh', 'hard_sigmoid']

search_pool = dict(batch_size=batch_size, optimizer=optimizer, dropout=dropout, activation =activation

grid = RandomizedSearchCV(model,search_pool, return_train_score=True, n_jobs=-1)
grid_result = grid.fit(x_train, Y_train)
```

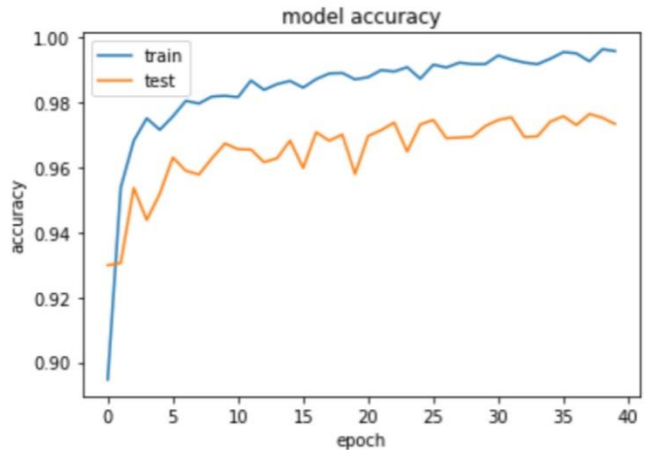
Results & Discussion

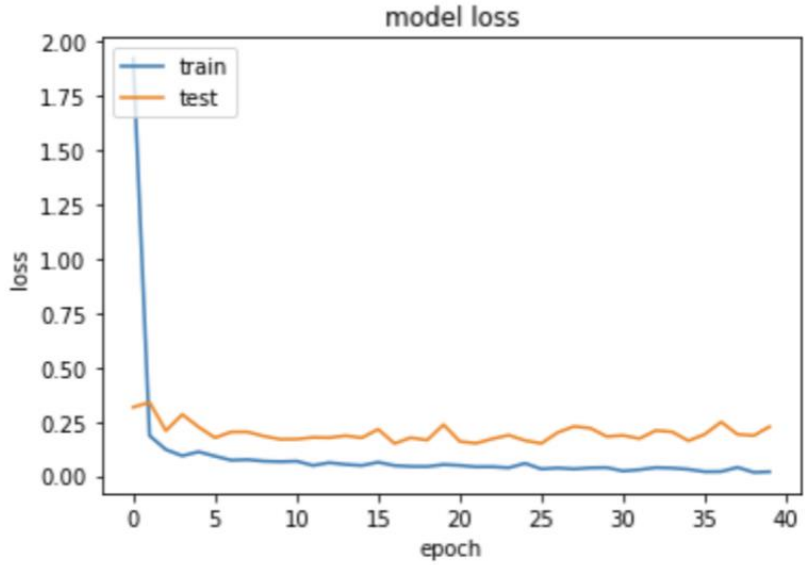
Grid search have taken about 9 hours to complete with the best parameters being:

```
Best: 0.963967 using {'activation': 'relu', 'batch_size': 120, 'dropout': 0.
0, 'optimizer': 'adam'}
```

testing these parameters on the same model gives us the following measurement of accuracy and loss.

```
Model: "sequential_6"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_24 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_18 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_25 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_19 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_26 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_20 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_27 (Dense)             (None, 10)                5130
=================================================================
Total params: 932,362
Trainable params: 932,362
Non-trainable params: 0
_________________________________________________________________
```



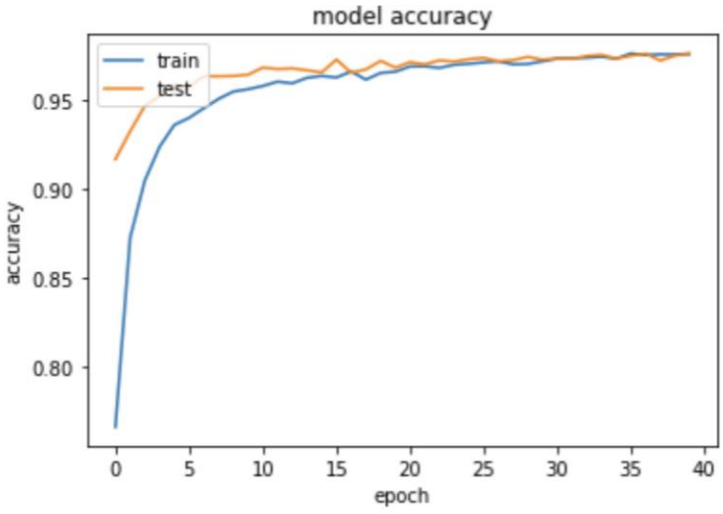


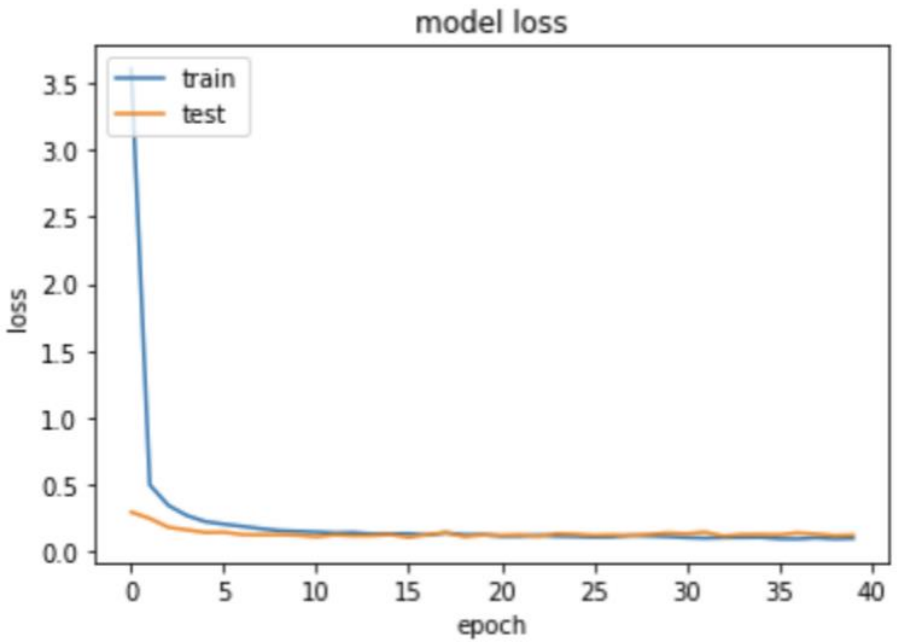
On the other hand, randomized search gives us the following results:

```
Best: 0.958583 using {'optimizer': 'adam', 'dropout': 0.3, 'batch_size': 160,
'activation': 'relu'}
```

Running these parameters on validation model gives us the following measurement of accuracy and loss.

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense_24 (Dense)             (None, 512)               401920
_________________________________________________________________
dropout_18 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_25 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_19 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_26 (Dense)             (None, 512)               262656
_________________________________________________________________
dropout_20 (Dropout)         (None, 512)               0
_________________________________________________________________
dense_27 (Dense)             (None, 10)                5130
=================================================================
Total params: 932,362
Trainable params: 932,362
Non-trainable params: 0
_________________________________________________________________
```





Running both methods side by side the first thing to note is how much faster randomized search is. Taking about 40 minute while grid search took about 9 hours. The reason is that, by definition, grid search tests every combination in the search grid on the train model and then compares the scores and gives the best performing hyperparameters. By contrast, randomized search will not attempt to test out all combination, but rather, a fixed number of hyperparameter settings is sampled from specified probability distributions. While it's possible that RandomizedSearchCV will not find as accurate of a result as GridSearchCV, it surprisingly picks the best result more often than not and in a fraction of the time it takes GridSearchCV would have taken.

This variance in performance is reflected in the output of the both methods, was able to achieve 95.8 accuracy in fraction of the time it took for GridSearchCV to achieve 96.4 accuracy. Testing the parameters further on the same model with re-splitted data showed similar accuracy on the test data (97% for GridSearchCV parameters and 96% for RandomizedSearchCV).

In term of loss, running RandomizedSearchCV parameters showed a clear underfitting (the small gab between training data and test data). underfitting can be easily addressed by increasing the capacity of the network. However, for the sake of comparison same parameters were used on both models.

```
params = grid_result.best_params_


model = create_model(optimizer=params["optimizer"],activation=params["activati
on"], dropout=params["dropout"])




history = model.fit(x_train,
                    Y_train,
                    batch_size=params["batch_size"],
                    epochs=40,
                    verbose = 2,
                    validation_data=(x_test, Y_test)
                    )
model.summary()
plt.plot(history.history['accuracy'])
plt.plot(history.history['val_accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()


plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper left')
plt.show()
```

Conclusion

In conclusion, hyperparameter tuning is one of the most important aspects in neural network configuration due to its impact on the accuracy. Grid and randomized search algorithms are two methods that can be used to find the optimal hyperparameters configuration. In this report, the two methods are compared using the same model, same set of hyperparameters, and the same validation model. Randomized search was found to be considerably faster in terms of time, yet its set of parameters achieved accuracy results that slightly behind grid search since the ladder test out every combination of hyperparameters. Running both outputs in a validation model with re-splitted data to extract accuracy and loss plots. In general, Grid search outperforms Randomized search at the cost of time, with that in mind it can be inefficient to use grid search on large data sets and instead use Randomized search which provide close results in a fraction of the time.