

National University of Singapore
College of Design & Engineering - ECE

EE4400 Data Engineering and Deep Learning
Tutorial 2 - RNN Solutions

Q1. As a result of using the backpropagation algorithm with the sigmoid activation function, gradient values can become very small and vanish. On the other hand, it is also possible for some values to become very large, leading to the exploding gradients problem. Proper weight initialisation, use of ReLU activation function and batch normalization can help to overcome these issues.

Q2. Batch normalization accelerates learning and better performance can be seen even with fewer hidden nodes per layer. See attached Jupyter notebook with results. The non-trainable parameters are from the batch normalization layers, i.e. the mean and standard deviation terms are moving averages and are not trained by back-propagation.

Q3. Recurrent neural networks (RNNs) have internal states that are computed and passed forward in time to the next time step. The internal state is a function of the current input and past internal state, hence it is a function of all the past inputs.

Q4. The sequence-to-sequence type should be used for time series prediction. The output at each time step should be a value or sequence of values which are shifted by one or more steps into the future.

Q5. Batch normalization (BN) involves capturing the statistics of inputs in mini-batches, normalising the inputs, and applying a scale factor and offset to them. This works well in feedforward network training.

However, BN does not work well with RNN because the operation changes the characteristics of the sequential data presented in an RNN. The repeated application of the scale factor and offset in BN during the recurrent procedure may also lead to exploding or vanishing gradients.

Layer normalization can work well in an RNN as it captures statistics across inputs at a single time step.

Q6. Refer to the Jupyter notebook with results and explanations.

EE4400 - Tutorial 2, Question 2 (Tut1Q3 with Batch Normalization)

This question is on the Multi-Layer Perceptron (MLP) and using it to do classification. The aim is to find the best number of hidden nodes in the 3 hidden layers, assuming the same number of hidden nodes in each hidden layer. Cross-validation needs to be done on the training set. The MLP classifier with the best network size is then used for testing.

We shall use the Tensorflow Keras package to implement the MLP classifier. Obtain the data set "from sklearn.datasets import load_iris". Import the necessary packages.

```
## load data from scikit
import numpy as np
import pandas as pd
print("pandas version: {}".format(pd.__version__))
import sklearn
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn import metrics

from tensorflow import keras
from keras.models import Sequential
from keras.layers import Dense, Activation, BatchNormalization
from keras.optimizers import SGD

pandas version: 1.1.5
```

(a) Load the data and split the database into two sets: 80% of samples for training, and 20% of samples for testing.

```
## load data
iris_dataset = load_iris()
## split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(iris_dataset['data'],
                                                    iris_dataset['target'],
                                                    test_size=0.20,
                                                    random_state=0)
```

Change y_train and y_test to categorical values (required for classification).

```
y_train = keras.utils.to_categorical(y_train, num_classes = 3)
y_test = keras.utils.to_categorical(y_test, num_classes = 3)
```

(b) Perform a 5-fold Cross-validation using only the training set to determine the best 3-layer MLP classifier with hidden_layer_sizes=(Nhidd,Nhidd,Nhidd) for Nhidd in range(1,11)[^] for prediction. In other words, partition the training set into two sets, 4/5 for training and 1/5 for validation; and repeat this process until each of the 1/5 has been validated. ^ The assumption of hidden_layer_sizes=(Nhidd,Nhidd,Nhidd) is to reduce the search space in this exercise. In field applications, the search needs to consider different sizes for each hidden layer.

```
def MLP_model(Nhidd):
    model = Sequential()
    model.add(Dense(Nhidd, activation='relu', input_shape=(4,)))
    model.add(BatchNormalization())
    model.add(Dense(Nhidd, activation='relu'))
    model.add(BatchNormalization())
    model.add(Dense(Nhidd, activation='relu'))
    model.add(BatchNormalization())
    ...
```

```

model.add(Dense(3, activation='softmax'))
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])
return model

```



```

acc_train_array = []
acc_valid_array = []
for Nhidd in range(1,11):
    acc_train_array_fold = []
    acc_valid_array_fold = []
    ## Random permutation of data
    Idx = np.random.RandomState(seed=8).permutation(len(y_train))
    ## Tuning: perform 5-fold cross-validation on the training set to determine the best network size
    clf = MLP_model(Nhidd)
    for k in range(0,5):
        N = np.around((k+1)*len(y_train)/5)
        N = N.astype(int)
        Xvalid = X_train[Idx[N-24:N]] # validation features
        Yvalid = y_train[Idx[N-24:N]] # validation targets
        Idxtrn = np.setdiff1d(Idx, Idx[N-24:N])
        Xtrain = X_train[Idxtrn] # training features in tuning loop
        Ytrain = y_train[Idxtrn] # training targets in tuning loop
        ## MLP Classification with same size for each hidden-layer (specified in question)
        clf.fit(Xtrain, Ytrain, epochs = 100, verbose = 0)
        ## trained output
        y_est_p = clf.predict(Xtrain)
        Ytrain_class = np.argmax(Ytrain, axis=1)
        y_est_p_class = np.argmax(y_est_p, axis=1)
        acc_train_array_fold += [metrics.accuracy_score(y_est_p_class,Ytrain_class)]
        ## validation output
        yt_est_p = clf.predict(Xvalid)
        Yvalid_class = np.argmax(Yvalid, axis=1)
        yt_est_p_class = np.argmax(yt_est_p, axis=1)
        acc_valid_array_fold += [metrics.accuracy_score(yt_est_p_class,Yvalid_class)]
    acc_train_array += [np.mean(acc_train_array_fold)]
    acc_valid_array += [np.mean(acc_valid_array_fold)]
clf.summary()

```

dense_29 (Dense)	(None, 8)	72
batch_normalization_22 (Batch Normalization)	(None, 8)	32
dense_30 (Dense)	(None, 8)	72
batch_normalization_23 (Batch Normalization)	(None, 8)	32
dense_31 (Dense)	(None, 3)	27
<hr/>		
Total params:	307	
Trainable params:	259	
Non-trainable params:	48	

Model: "sequential_8"

Layer (type)	Output Shape	Param #
dense_32 (Dense)	(None, 9)	45
batch_normalization_24 (Batch Normalization)	(None, 9)	36
dense_33 (Dense)	(None, 9)	90
batch_normalization_25 (Batch Normalization)	(None, 9)	36
dense_34 (Dense)	(None, 9)	90
batch_normalization_26 (Batch Normalization)	(None, 9)	36
dense_35 (Dense)	(None, 3)	30
<hr/>		

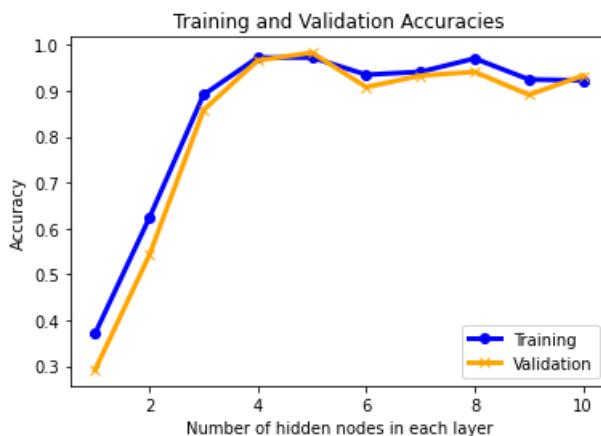
```
Total params: 363
Trainable params: 309
Non-trainable params: 54
```

```
Model: "sequential_9"
```

Layer (type)	Output Shape	Param #
<hr/>		
dense_36 (Dense)	(None, 10)	50
<hr/>		
batch_normalization_27 (Batch Normalization)	(None, 10)	40
<hr/>		
dense_37 (Dense)	(None, 10)	110
<hr/>		
batch_normalization_28 (Batch Normalization)	(None, 10)	40
<hr/>		
dense_38 (Dense)	(None, 10)	110
<hr/>		
batch_normalization_29 (Batch Normalization)	(None, 10)	40
<hr/>		
dense_39 (Dense)	(None, 3)	33
<hr/>		
Total params: 423		
Trainable params: 363		
Non-trainable params: 60		

(c) Provide a plot of the average 5-fold training and validation accuracies over the different network sizes, i.e. different number of nodes in the hidden layer. Determine the hidden layer size Nhidd that gives the best validation accuracy for the training set.

```
## plotting
import matplotlib.pyplot as plt
hiddensize = [x for x in range(1,11)]
plt.plot(hiddensize, acc_train_array, color='blue', marker='o', linewidth=3, label='Training')
plt.plot(hiddensize, acc_valid_array, color='orange', marker='x', linewidth=3, label='Validation')
plt.xlabel('Number of hidden nodes in each layer')
plt.ylabel('Accuracy')
plt.title('Training and Validation Accuracies')
plt.legend()
plt.show()
## find the best hidden layer size that gives the best validation accuracy using only the training set
Nhidd = np.argmax(acc_valid_array, axis=0)+1
print('best hidden layer size =', Nhidd, 'based on 5-fold cross-validation on training set')
```



best hidden layer size = 5 based on 5-fold cross-validation on training set

```
print(acc_train_array)
print(acc_valid_array)
```

```
[0.3708333333333333, 0.6229166666666667, 0.8916666666666668, 0.9729166666666667, 0.9729166666666667, 0.93541
[0.2916666666666663, 0.5416666666666666, 0.8583333333333334, 0.9666666666666668, 0.9833333333333332, 0.9083
```

(d) Using the best hidden layer size Nhidd in the MLP classifier with hidden_layer_sizes=(Nhidd,Nhidd,Nhidd), evaluate the performance of the MLP by computing the prediction accuracy based on the 20% of samples for testing in part (a).

```
## perform evaluation
clf = MLP_model(Nhidden)
history=clf.fit(X_train, y_train, epochs = 100, batch_size = 32, verbose = 0)
## trained output
y_test_predict = clf.predict(X_test)
y_test_class = np.argmax(y_test, axis=1)
y_test_predict_class = np.argmax(y_test_predict, axis=1)
test_accuracy = metrics.accuracy_score(y_test_predict_class,y_test_class)
print('test accuracy =', test_accuracy)
```

test accuracy = 0.9666666666666667

```
plt.plot(history.history["loss"])
```

