FIT3181: Deep Learning (2023)

CE/Lecturers: **Prof Dinh Phung** [dinh.phung@monash.edu (mailto:dinh.phung@monash.edu)] | **Prof Jianfei Cai** [jainfei.cai@monash.edu (mailto:jainfei.cai@monash.edu)] | **Dr Toan Do** [toan.do@monash.edu (mailto:toan.do@monash.edu)] | **Dr Lim-Chern Hong** [lim.chernhong@monash.edu (mailto:lim.chernhong@monash.edu)]

Tutors: **Dr Binh Nguyen** [binh.nguyen1@monash.edu (mailto:binh.nguyen1@monash.edu)] | **Mr Tony Bui** [tuan.bui@monash.edu (mailto:tuan.bui@monash.edu)] | **Ms Vy Vo** [v.vo@monash.edu (mailto:v.vo@monash.edu)]

Faculty of Information Technology, Monash University, Australia

Lab 02a: Feed-forward Neural Nets with TensorFlow 2.x

This tutorial demonstrates how to implement a Feedforward Neural Network using keras TF 2.x.

As you can see later, the implementation is much simpler compared to TF 1.x.

Feedforward Neural Network ***** (highly important)

We will consider a fairly realistic deep NNs with *three* layers plus the *output* layer. Its architecture will be specified as: $16 \rightarrow 10(ReLU) \rightarrow 20(ReLU) \rightarrow 15(ReLu) \rightarrow 26$. This means:

- Input size is 16
- First layer has 10 hidden units with ReLU activation function
- Second layer has 20 hidden units with 20 ReLU activiation function
- Third layer has 15 hidden units with 15 ReLU activiation function
- And output layer is logit layer with 26 hidden units

This network, for example, can take the `letter` dataset with an input of 16 features and target label of 26 classes (A-Z). **Our objective in this tutorial is to implement this specific network in `TensorFlow 2.x`.**

III. Implementation with TensorFlow 2.x ***** (highly important)

Let's recall our task and what constitutes a standard machine learning pipeline.

We need to implement the aforementioned network with the architecture of $16 \rightarrow 10(ReLU) \rightarrow 20(ReLU) \rightarrow 15(ReLu) \rightarrow 26$. We again experiment with using the `letter` dataset, which can be found at [the LIBSVM website](https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/multiclass.html#letter).

The objective is to identify each of a large number of black-and-white rectangular pixel displays as one of the 26 capital letters in the English alphabet. The character images were based on 20 different fonts and each letter within these 20 fonts was randomly distorted to produce a file of 20,000 unique stimuli. Each stimulus

The standard process of implementing a deep learning model is described as follows:

Step 1: Data processing

- Load the dataset and split into train, valid, and test sets.

Step 2: Construction phase

- Define the NN model and construct the corresponding computational graph.
- Define the loss function and the relevant measures of performance of interest (accuracy, F1, and AUC).

Step 3: Execution and evaluation phase

- Train the model using mini-batches from the train set by minimizing the loss function with an optimizer.
- Predict on the test set and access its performance.

Step 1: Data Processing

We repeat the data processing steps from the previous tutorial. We first use `sklearn` to load the dataset.

In [ ]:

```
import os
import numpy as np
from sklearn.datasets import load_svmlight_file
```

In [ ]:

```
data_file_name= "letter_scale.libsvm"
data_file = os.path.abspath(os.path.join("./Data", data_file_name))
X_data, y_data = load_svmlight_file(data_file)
X_data= X_data.toarray()
y_data= y_data.reshape(y_data.shape[0],-1)
print("X data shape: {}".format(X_data.shape))
print("y data shape: {}".format(y_data.shape))
print("# classes: {}".format(len(np.unique(y_data))))
print(np.unique(y_data))
```

The data has 26 classes, encoded by numbers from 1 to 26. In case labels are in categorical form, you might need to encode them in the form of numerical labels. For example, we want to turn $y\_data = [" \ cat \ ", " \ dog \ ", " \ cat \ ", " \ lion \ ", " \ dog \ "]$ to $y\_data = [0, 1, 0, 2, 1]$.

To do this, in the following segment of code, we use the object `le` as an instance of the class `preprocessing.LabelEncoder()` which supports us to transform catefgorial labels in `y_data` to numerical vector.

In [ ]:

```
from sklearn import preprocessing
le = preprocessing.LabelEncoder()
le.fit(y_data.ravel())
y_data = le.transform(y_data)
y_data = y_data.ravel()
print(y_data[:])
print(np.unique(y_data))
```

Note that labels are now encoded by integers starting from.

We then split the dataset into the train, validation, and test sets.

In [ ]:

```
from sklearn.model_selection import train_test_split

def train_valid_test_split(data, target, train_size, test_size):
    valid_size = 1 - (train_size + test_size)
    X1, X_test, y1, y_test = train_test_split(data, target, test_size = test_size,
    X_train, X_valid, y_train, y_valid = train_test_split(X1, y1, test_size = floa
    return X_train, X_valid, X_test, y_train, y_valid, y_test
```

We now use the function defined above to prepare our data for training, validating and testing.

In [ ]:

```
X_train, X_valid, X_test, y_train, y_valid, y_test = train_valid_test_split(X_data
                                                                            train_
                                                                            test_s
y_train = y_train.reshape(-1)
y_test = y_test.reshape(-1)
y_valid = y_valid.reshape(-1)
print(X_train.shape, X_valid.shape, X_test.shape)
print(y_train.shape, y_valid.shape, y_test.shape)
print("lables: {}".format(np.unique(y_train)))
```

In [ ]:

```
train_size = int(X_train.shape[0])
n_features = int(X_train.shape[1])
n_classes = len(np.unique(y_train))
```

Step 2: Construction phase

Let's now build a feedforward neural network with the architecture:
$16 \rightarrow 10(ReLU) \rightarrow 20(ReLU) \rightarrow 15(ReLu) \rightarrow 26$ in TensorFlow 2.x.

In [ ]:

```
import tensorflow as tf
from tensorflow.keras.layers import Dense, Flatten
from tensorflow.keras.models import Sequential
```

In [ ]:

```
tf.random.set_seed(1234)
```

`Sequential` API is the easiest way to create Keras models. It allows us to specify a neural network **sequentially** by adding individual layers on top of one another. A sequential model is a plain stack of layers, which can be simply done by calling the `add()` method.

As shown below, we will add 3 intermediate Dense layers with $ReLu$ activation function. The last one is the output layer with $Softmax$ activation that returns a probability output tensor of size equal to the number of classes to be predicted.

In [ ]:

```
dnn_model = Sequential()
dnn_model.add(Dense(units=10,  input_shape=(16,), activation='relu'))
dnn_model.add(Dense(units=20, activation='relu'))
dnn_model.add(Dense(units=15, activation='relu'))
dnn_model.add(Dense(units=n_classes, activation='softmax'))
```

To finish the model construction, we call the `build()` method and call `summary()` to view the model setup.

In [ ]:

```
dnn_model.build()
dnn_model.summary()
```

You can further inspect the model components such as layers, their names, types as well as the parameters at each layer.

In [ ]:

```
dnn_model.layers # returns a list of model layers
```

In [ ]:

```
hidden1 = dnn_model.layers[0]
hidden1
print(hidden1.name) # returns the name of the first layer
```

In [ ]:

```
weights, biases = hidden1.get_weights()
weights.shape, biases.shape # returns the dimensions of the weight matrix and bias
```

Step 3: Execution and Evaluation Phase

Next, we use the `compile()` method to specify the training configurations, most importantly the optimizer, loss function and evaluation metrics. The model is now ready for training!

In [ ]:

```
dnn_model.compile(optimizer='adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy'])
```

Training and Evaluating

To train the model, we call the `fit()` method while inputting the training and validation data. The training progress will be printed out by default in your terminal. We now show you how to inspect the training output.

Visualizing Training Progress

In this example, we demonstrate two approaches to visualize training progress, using a History object and using TensorBoard.

Using History object The history object is the output of `fit()` method, which contains the training parameters `history.params` , the list of epochs `history.epoch` , and most importantly a dictionary `history.history` containing the loss and extra metrics evaluated at the end of each epoch on the training set and on the validation set (if any). The training need to complete before we can visualize using the history output.

Using TensorBoard To visualize with TensorBoard we first need to create a `tensorboard callback` method with specific log directory. We then pass the callback method to `model.fit()` method. Unlike the previous method, the callback method writes log data to the log file on-the-fly. Therefore, by opening Tensorboard on a separate browser, we can train a model and parallelly visualize the training progress.

In [ ]:

```
from tensorflow import keras
logdir = "tf_logs/example01"

# Init a tensorboard_callback
tensorboard_callback = keras.callbacks.TensorBoard(log_dir=logdir)

# Call the fit method, passing the tensorboard_callback
history = dnn_model.fit(x=X_train, y=y_train, batch_size=32,
                        epochs=20,
                        validation_data=(X_valid, y_valid),
                       callbacks=[tensorboard_callback])
```

We now can evaluate the trained model on the testing set or any subset.

In [ ]:

```
dnn_model.evaluate(X_test, y_test)  #return loss and accuracy
```

In [ ]:

```
X_new = np.reshape(X_test[10, :], (1,-1))
y_prob = dnn_model.predict(X_new)
y_prob.round(2)
```

In [ ]:

```
y_pred = np.argmax(dnn_model.predict(X_new), axis=-1)
if y_pred[0]==y_test[0]:
    print("Correct predeiction !")
else:
    print("Incorrect prediction !")
```

Visualizing Model Performance and Loss Objective Function

There are four keys in the history dictionary: `loss` and `val_loss` measure the loss on the training set and the validation set, respectively, while `accuracy` and `val_accuracy` measure the accuracy on the training set and the validation set.

The following figure visualize all four metrics with two y-axes, losses (blue lines, in descending) and accuracies (red lines, in asending)

In [ ]:

```
import pandas as pd
import matplotlib.pyplot as plt


his = history.history
fig = plt.figure(figsize=(8, 5))
ax = fig.add_subplot(111)
ln1 = ax.plot(his['loss'], 'b--',label='loss')
ln2 = ax.plot(his['val_loss'], 'b-',label='val_loss')
ax.set_ylabel('loss', color='blue')
ax.tick_params(axis='y', colors="blue")

ax2 = ax.twinx()
ln3 = ax2.plot(his['accuracy'], 'r--',label='accuracy')
ln4 = ax2.plot(his['val_accuracy'], 'r-',label='val_accuracy')
ax2.set_ylabel('accuracy', color='red')
ax2.tick_params(axis='y', colors="red")


lns = ln1 + ln2 + ln3 + ln4
labels = [l.get_label() for l in lns]
ax.legend(lns, labels)
plt.grid(True)
plt.show()
```

To visualize using Tensorboard on the same jupyter notebook, we first need to load the TensorBoard extension, then call the tensorboard with log file directory.

In [ ]:

```
# Load the TensorBoard notebook extension.
# %load_ext tensorboard
%tensorboard --logdir tf_logs/
```

Playing around with different optimizers

In the following code, we will experiment with different optimizers to find which yields the best predictive performance (evaluate on the validation set). It can easily be done by passing the names of specific optimizers when compiling model.

In [ ]:

```
optimizer_names = ["Nadam", "Adam", "Adadelta", "Adagrad", "RMSprop", "SGD"]
optimizer_list = [keras.optimizers.Nadam(learning_rate=0.001), keras.optimizers.Ad
                  keras.optimizers.Adagrad(learning_rate=0.001), keras.optimizers.
best_acc = 0
best_i = -1
for i in range(len(optimizer_list)):
    print("*Evaluating with {}\n".format(str(optimizer_names[i])))
    dnn_model.compile(optimizer=optimizer_list[i], loss='sparse_categorical_crosse
    dnn_model.fit(x=X_train, y=y_train, batch_size=32, epochs=30, validation_data=
    acc = dnn_model.evaluate(X_valid, y_valid)[1]
    print("The valid accuracy is {}\n".format(acc))
    if acc > best_acc:
        best_acc = acc
        best_i = i
print("The best valid accuracy is {} with {}".format(best_acc, optimizer_names[bes
```

Fine-tuning the learning rate

Learning rate plays an important role when training a deep learning model. In the following code, we will run a simple greedy search to find a good learning rate.

In [ ]:

```
lr = [1e-2, 5e-3, 1e-3, 1e-4, 1e-5]

best_acc = 0
best_i = -1
for i in range(len(lr)):
    print("*Evaluating with learning rate = {}\n".format(str(lr[i])))
    dnn_model.compile(optimizer=keras.optimizers.Adam(learning_rate=lr[i]), loss='
    dnn_model.fit(x=X_train, y=y_train, batch_size=32, epochs=30, validation_data=
    acc = dnn_model.evaluate(X_valid, y_valid)[1]
    print("The valid accuracy is {}\n".format(acc))
    if acc > best_acc:
        best_acc = acc
        best_i = i
print("The best valid accuracy is {} with learning rate {}".format(best_acc, lr[be
```

Save and Load Models

There are different ways to save TensorFlow models depending on the API you are using. Since we are using the Keras model in this tutorial, saving and loading it is quite simple. It can be done by calling `model.save()` and `load_model()` methods. When calling `model.save()`, the entire model will be saved including:

- The architecture, or configuration, which specifies what layers the model contain, and how they're connected.
- A set of weights values (the "state of the model").
- An optimizer (defined by compiling the model).
- A set of losses and metrics (defined by compiling the model or calling add_loss() or add_metric()).

In [ ]:

```
# Saving the entire model to a directory
dnn_model.save('models/')

# Loading the model back
from tensorflow import keras
loaded_model = keras.models.load_model('models/')

# Checking the loaded model
print(dnn_model.predict(X_new) == loaded_model.predict(X_new))
```

Save model during training

One major disadvantage of the above saving method is that we cannot save the model during training but only when the training is finished. Therefore, when the training is stopped or interrupted, we have to train the model again. To save model during training process, we can use the `ModelCheckpoint` callback that allows you to continually save the model both during and at the end of training.

Some important arguments of the `ModelCheckpoint` callback:

- `filepath` : checkpoin directory
- `save_weights_only` : if True, then only the model's weights will be saved (i.e., equivalent to `model.save_weights(filepath)`); otherwise the full model is saved (i.e., equivalent to `model.save(filepath)` which saves: model weight, model architecture, optimizer, etc.)
- `save_best_only` : if True, it only saves when the model is considered the "best" and the latest best model according to the quantity monitored will not be overwritten. The "best" model is evaluated based on "mode" and "monitor". For example, if `monitor=val_accuracy` it means that validation accuracy is used to monitor the best checkpoint, and `mode` should be set to `max`. If `monitor=val_loss` it means that validation loss is used instead, and `mode` in this case should be `min`.

More details can be found in this documentation: https://www.tensorflow.org/tutorials/keras/save_and_load (https://www.tensorflow.org/tutorials/keras/save_and_load)

In [ ]:

```
# Create a tf.keras.callbacks.ModelCheckpoint callback that saves weights only dur
checkpoint_path = "models/cp.ckpt"

# Create a callback that saves the model's weights
cp_callback = tf.keras.callbacks.ModelCheckpoint(filepath=checkpoint_path,
                                                 save_weights_only=True,
                                                 verbose=1)

# Train the model with the new callback
dnn_model.fit(x=X_train, y=y_train, batch_size=32,
                         epochs=20,
                         validation_data=(X_valid, y_valid),
                        callbacks=[tensorboard_callback, # Callback for writing log
                                   cp_callback]) # Callback for saving model
```

Save model during training

If we have only saved the model weight, we need to recreate a same architecture before loading the model weight.

In [ ]:

```
# Because we alreary created a model, therefore, we just need to load the weight
# dnn_model = create_model() # run this step if you have not defined the model in
dnn_model.load_weights(checkpoint_path)
```

If we saved the entire model (by set `save_weights_only=False`), then the pretrained model can be reloaded by `load_model` method

IV. Two Approaches to Build Up Models with TensorFlow 2.x *** (relatively important)

There are two approaches to build up a model with tensorflow 2.x, a simple method using **Sequential API** and a more flexible method using **Functional API**.

Approach 1: Using `Sequential API`

In [ ]:

```
dnn_model = Sequential()
dnn_model.add(Dense(units=10,  input_shape=(16,), activation='relu'))
dnn_model.add(Dense(units=20, activation='relu'))
dnn_model.add(Dense(units=15, activation='relu'))
dnn_model.add(Dense(units=26, activation='softmax'))
dnn_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metric
```

Approach 2: Using `Functional API`

In [ ]:

```
X = tf.keras.layers.Input(shape=(16,)) #declare input layer
h = Dense(units=10, activation= 'relu')(X)
h = Dense(units=20, activation= 'relu')(h)
h = Dense(units=15, activation= 'relu')(h)
h = Dense(units=26, activation= 'softmax')(h)
dnn_model = tf.keras.Model(inputs= X, outputs=h)
dnn_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metric
```

We can also declare a class inherited from `tf.keras.Model`

In [ ]:

```
class MyDNN(tf.keras.Model):
    def __init__(self, n_classes= 26):
        super(MyDNN, self).__init__()
        self.n_classes = n_classes
        self.dense1 = tf.keras.layers.Dense(units=10, activation= 'relu')
        self.dense2 = tf.keras.layers.Dense(units=20, activation= 'relu')
        self.dense3 = tf.keras.layers.Dense(units=15, activation= 'relu')
        self.dense4 = tf.keras.layers.Dense(units=self.n_classes, activation= 'sof

    def call(self,X): #X is the input, method call specifies how to compute the ou
        h = self.dense1(X)
        h = self.dense2(h)
        h = self.dense3(h)
        h = self.dense4(h)
        return h
dnn_model = MyDNN(n_classes= 26)
dnn_model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metric
```

V. Other approaches to Train a Model with TensorFlow 2.x *** (relatively important)

There are two main approaches to training a model with Tensorflow 2.x. The simplest method is the `fit` method as we did before. This method automatically helps us process data when training (e.g., split an entire dataset into multiple mini-batches), applies callback methods such as saving model or writing TensorBoard and monitors validation performance.

However, some projects require more intervention in the training process (for example, doing data augmentation in self-supervised learning or training a generative model that we will learn later in this unit). In this case, you should be able to manually construct the training pipeline.

In Tensorflow 2.X, we can do this with `train_on_batch` method. Basically, you will need to

- manually split entire dataset into mini-batches and applied data augmentaion (if any)
- feed training data to `train_on_batch` method. It returns a training loss (which is pre-defined when compiling model) and an updated model.

The following code presents a simple example (without any data-augmentation).

In [ ]:

```
n_epochs =20
batch_size = 64
for epoch in range(n_epochs):
    for idx_start in range(0, X_train.shape[0], batch_size):
        idx_end = min(X_train.shape[0], idx_start + batch_size)
        X_batch, y_batch = X_train[idx_start:idx_end], y_train[idx_start:idx_end]
        train_loss_batch = dnn_model.train_on_batch(X_batch, y_batch)  #return the

    train_loss, train_acc = dnn_model.evaluate(x= X_train, y= y_train, batch_size=
    valid_loss, valid_acc = dnn_model.evaluate(x= X_valid, y= y_valid, batch_size=
    print('Epoch {}: train acc={:.4f}, train loss={:.4f} | valid acc={:.4f}, valid
```

Additional Exercises

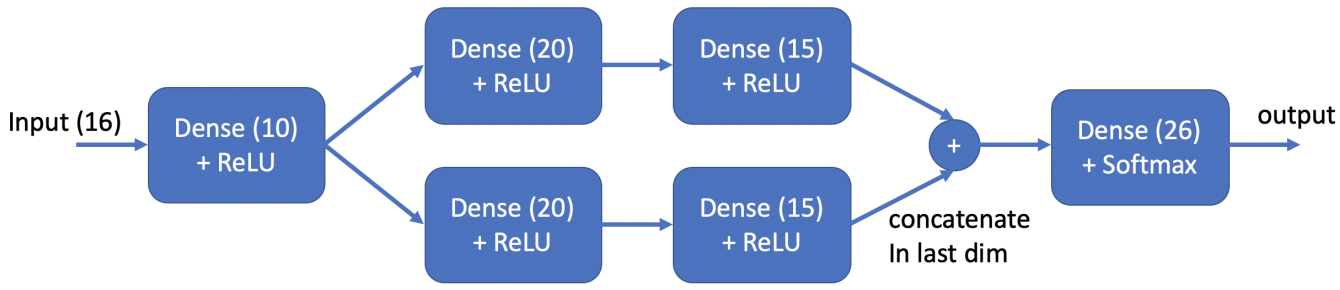
Exercise 1: Write your own code to save a trained model to the hard disk and restore this model, then use the restored model to output the prediction result on the test set.

Exercise 2: Insert new code to the above code to enable outputting to TensorBoard the values of `training loss`, `training accuracy`, `valid loss`, and `valid accuracy` at the end of epochs. You can refer to the code here (https://www.tensorflow.org/tensorboard/get_started).

Exercise 3: Write code to do regression on the dataset `cadata` which can be downloaded here (https://www.csie.ntu.edu.tw/~cjlin/libsvmtools/datasets/regression.html). Note that for a regression problem, you need to use the `L2` loss instead of the `cross-entropy` loss as in a classification problem.

Exercise 4: Using the problem in this tutorial, however, using a much deeper network (i.e., $16 \rightarrow 100(ReLU) \rightarrow 200(ReLU) \rightarrow 200(ReLU) \rightarrow 100(ReLu) \rightarrow 26$). Applying callback methods to save the model on training and writing a TensorBoard. Visualize `training loss`, `training accuracy`, `valid loss`, and `valid accuracy`. Provide observation and explanation of any issue if happen (hint, overfitting issue).

Exercise 5: Build up a more complex feedforward neural network with `Functional API` method as shown in figure below. The network splits into two branches and then merges in the last layer. The concatenate operation is in the last dimenstion (for example, two arrays [10,15], [10,15] will be concatenated to an array [10,30].



THE END