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 02c: Feedforward Neural Nets with TensorFlow 1.x

The purpose of this tutorial is to demonstrate an end-to-end process of building and training a Feedforward Neural Network with TensorFlow. In this tutorial, you will learn:

- What constitutes a deep learning pipeline.
- How to implement a feedforward neural net for a multi-class classfication problem using TF 1.x.

Feedforward Neural Network

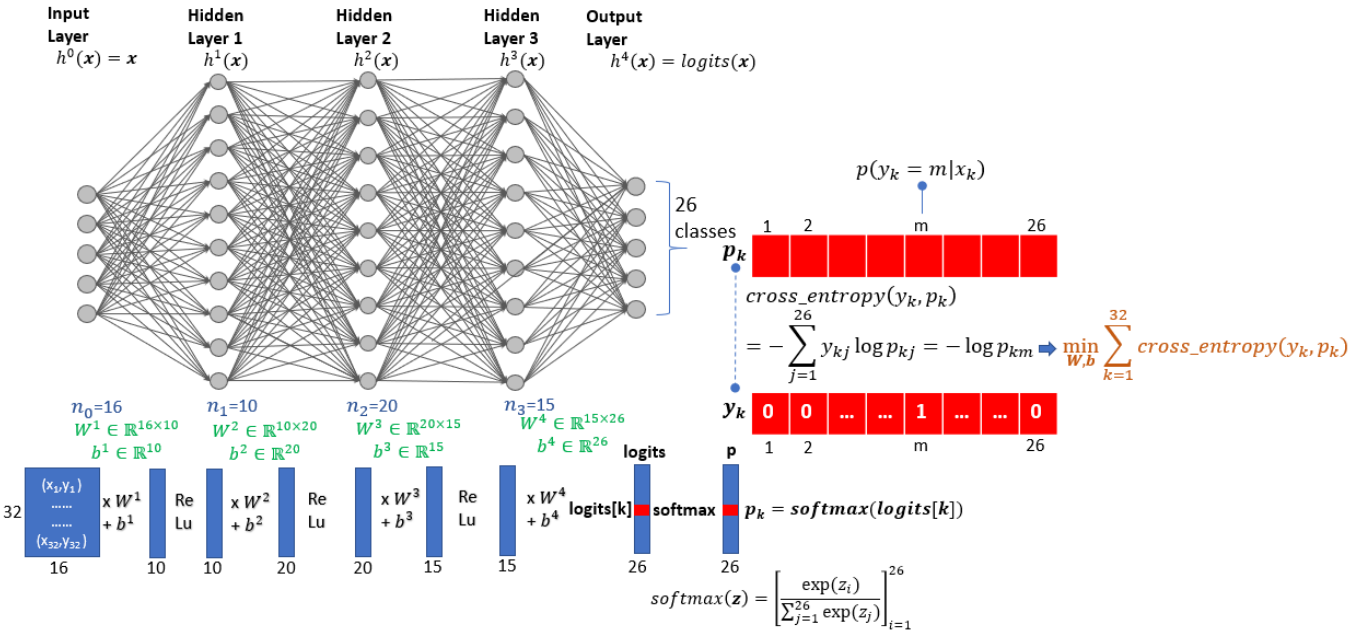
In this tutorial, we will consider a fairly realistic deep NNs with *three* layers plus the *output* layer. Its architecture will be specified as: $16 \rightarrow 10(ReLU) \rightarrow 20(ReLU) \rightarrow 15(ReLu) \rightarrow 26$. This means:

- Input size is 16
- First layer has 10 hidden units with ReLU activation functions
- Second layer has 20 hidden units with 20 ReLU activiation functions
- Third layer has 15 hidden units with 15 ReLU activiation functions
- 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 1.x`.**

I. Specifying the Neural Network Architecture

We can visualize this network as in the figure below. Please note that for readability, the number of hidden



The above figure shows the pipeline of the entire process of feeding a mini-batch of size 32 into the network. It is Note that using***mini-batch*** is a common way to train deep NNs in practice.

Let us denote the mini-batch by $X_b = \{(x_1, y_1), \ldots, (x_{32}, y_{32})\}$. The mini-batch can be stored using a $2D$ tensor with the shape $(32, 16)$. Assume that in this network, we use the activation function $ReLu$ where $ReLu(t) = \max\{0, t\}$. The computation in the forward propagation step is as follows:

- Input X_b with mini-batch size of 32
- $h_1 = ReLu(X_b \times W^1 + b^1) \in \mathbb{R}^{32\times10}$.
- $h_2 = ReLu(h_1 \times W^2 + b^2 \in \mathbb{R}^{32\times20}$.
- $h_3 = ReLu(h_2 \times W^3 + b^3 \in \mathbb{R}^{32\times15}$.
- $logits = h_3 \times W^4 + b^4 \in \mathbb{R}^{32\times26}$
- $p = softmax(logits) \in \mathbb{R}^{32\times26}$

where we note that the activation function is perfomed element-wise and the softmax function is used to transform a vector of scalars to a discrete distribution as:

$$softmax(z) = \left[\frac{\exp(z_i)}{\sum_{j=1}^{26}\exp(z_j)}\right]_{i=1}^{26}$$

The k-th row p_k of the matrix p can represent the probability distribution to classify the data point x_k to the classes $1, 2, \ldots, 26$. In particular, we have:

$$p_{km} = p(y_k = m \mid x_k) \text{ for } m = 1, 2, \ldots, 26$$

Exercise 1 : Explain why the dimension for h_1 is 32×10? Similarly, please work out the dimension for $h2, h3, logits$ and p.

Specifying the Loss Function

Essiential to training a deep NN is the concept of the **loss function**. This function will tell us how good the

For classification task, a common approach is to use the **cross-entropy** loss function. Given a data-label instance (x_k, y_k) where feature $x_k \in \mathbb{R}^{16}$ and the label $y_k \in \{1, 2, \ldots, 26\}$ is a numeric label (for example if x_k is in the class 2, then $y_k = 2$ and its one-hot vector $1_{y_k} = [0, 1, 0, \ldots, 0]$). The cross-entropy between the classification distribution p_k returned from the NN and true label distribution y_k is defined as:

$$cross\_entropy(1_{y_k}, p_k) = -\sum_{j=1}^{26} y_{kj} \log p_{kj} = -\log p_{k,y_k}.$$

This loss basically enforces the model to predict the label as close as the true label by minimizing $cross\_entropy(1_{y_k}, p_k)$.

The above loss function was applied for each instance. For the entire current mini-batch, our loss function becomes:

$$\min \sum_{k=1}^{32} cross\_entropy(1_{y_k}, p_k)$$

Exercise 2: : **In the corss-entropy equation above, y_k is the class for x_k, explain why the end result is $-\log p_{k,y_k}$.**

Exercise 3: : **Let $p = [0.1, 0.3, 0.6]$ and $q = [0.0, 0.5, 0.5]$ be two discrete distributions, what is the $cross\_entropy(q, p)$?**

II. Implementation with TensorFlow 1.x **** (important)

We now implement the aforementioned network with the architecture of $16 \rightarrow 10(ReLU) \rightarrow 20(ReLU) \rightarrow 15(ReLu) \rightarrow 26$ in Tensorflow. We will 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)](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 was converted into 16 primitive numerical attributes (statistical moments and edge counts) which were then scaled to fit into a range of integer values from 0 through 15.

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 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("./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))
```

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

In [ ]:

```
from sklearn.model_selection import train_test_split
from sklearn import preprocessing
import tensorflow.compat.v1 as tf
tf.disable_v2_behavior()

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
```

Next, we would like to encode the label in the form of numeric vector. 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 transforms the catefgorial labels in `y_data` into a numerical vector.

In [ ]:

```
le = preprocessing.LabelEncoder()
le.fit(y_data)
y_data = le.transform(y_data)
print(y_data[:20])
```

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
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)))
```

We store some information about the training set for later uses.

In [ ]:

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

Stochastic Gradient Descent (SGD) is a commonly used optimizer in real-world implementation of deep learning models. Input to this algorithm is a sequence of **mini-batch** of data drawn from the training dataset.

Step 2: Construction Phase

We build up a feedforward neural network with the architecture:
$16 \rightarrow 10(ReLU) \rightarrow 20(ReLU) \rightarrow 15(ReLu) \rightarrow 26$ in TensorFlow.

In [ ]:

```
n_in = n_features    # dimension of input
n1 = 10              # number of hidden units at the first layer
n2 = 20              # number of hidden units at the second layer
n3 = 15              # number of hidden units at the third layer
n_out = n_classes    # number of classification classes
```

The function `dense_layer` represents a fully connected layer in a deep learning network. This takes W, b and input as inputs and returns $\sigma(W \times input + b)$ where the activation function σ is specified by the parameter `act` .

`TensorFlow` provides many options for `activation functions` such as `tf.nn.relu` , `tf.nn.sigmoid` , `tf.nn.tanh` . You can also define your own activation function.

In [ ]:

```
def dense_layer(inputs, output_size, act=None, name="hidden-layer"):
    with tf.name_scope(name):
        input_size = int(inputs.get_shape()[1])
        W_init = tf.random.normal([input_size, output_size], mean=0, stddev= 0.1,
        b_init = tf.random.normal([output_size], mean=0, stddev= 0.1, dtype= tf.fl
        W = tf.Variable(W_init, name= "W")
        b = tf.Variable(b_init, name="b")
        Wxb = tf.matmul(inputs, W) + b
        if act is None:
            return Wxb
        else:
            return act(Wxb)
```

We now construct the computational graph. But before that, remember to reset the default graph.

In [ ]:

```
tf.reset_default_graph()
with tf.name_scope("network"):
    X = tf.placeholder(shape=[None, n_in], dtype= tf.float32)
    y = tf.placeholder(shape=[None], dtype= tf.int32)
    h1 = dense_layer(X, n1, act= tf.nn.relu, name= "layer1")
    h2 = dense_layer(h1, n2, act= tf.nn.relu, name= "layer2")
    h3 = dense_layer(h2, n3, act= tf.nn.relu, name= "layer3")
    logits = dense_layer(h3, n_out, name="logits")
```

We compute the cross-entropy loss. In TensorFlow, you can use two of the following functions for evaluating the cross-entropy loss:

- `tf.nn.sparse_softmax_cross_entropy_with_logits` : if the labels `y_train` is in the categorial format (e.g., `y_train=[0,1,0,1,1,2]`).
- `tf.nn.softmax_cross_entropy_with_logits` : if the labels `y_train` is in the one-hot format (e.g., `y_train=[[1,0,0], [0,1,0], [1,0,0], [0,0,1]]` .

We also need to specify an optimizer to minimize the loss. Here, we are using the Adam optimizer for this optimization.

In [ ]:

```
with tf.name_scope('train'):
    xentropy = tf.nn.sparse_softmax_cross_entropy_with_logits(labels=y,
                                                              logits=logits,
                                                              name='xentropy')
    loss = tf.reduce_mean(xentropy, name="loss")
    tf.summary.scalar("loss", loss)    # summarize the loss
    optimizer = tf.train.AdamOptimizer(learning_rate=0.001)
    train_op = optimizer.minimize(loss)
```

In the above code, we run to add the loss value the summary.

To compute the prediction accuracy of our model, you can use the function `in_top_k()` (https://www.tensorflow.org/api_docs/python/tf/nn/in_top_k) with `k = 1` . This returns a 1D tensor full of boolean values, so we need to cast these booleans to floats and then compute the average. This will give us the network's overall accuracy.

The lines `tf.summary.scalar("loss", loss)` and `tf.summary.scalar("accuracy", accuracy)` are used to add the loss and accuracy values respectively to the summary.

In [ ]:

```
with tf.name_scope('evaluation'):
    correct = tf.nn.in_top_k(logits, y, k = 1)
    accuracy = tf.reduce_mean(tf.cast(correct, tf.float32))
    tf.summary.scalar("accuracy", accuracy)  #summarize the accuracy
```

We now define two FileWriters to write the summary to two log folders. In this way, we can plot the train, valid losses (or accuracies) on the same graph. Note that you can use this trick whenever you wish to display multiple plots on the same graph.

In [ ]:

```
if(not os.path.exists("./logs/train")):
    os.makedirs("./logs/train")

if(not os.path.exists("./logs/val")):
    os.makedirs("./logs/val")

merged = tf.summary.merge_all()
train_writer = tf.summary.FileWriter("./logs/train")
valid_writer = tf.summary.FileWriter("./logs/val")
```

Step 3: Execution and Evaluation Phase

In the `execution phase`, we need to create a `TensorFlow session`, then initialize `all variables` in the graph, execute `train_op`, and query the values of necessary nodes (e.g., `loss` and `accuracy`). The corresponding methods for each step are given as follows:

- Initialize all variables
  - `init = tf.global_variables_initializer()` and `sess.run(init)`.
- Execute `train_op` when feeding mini-batches to the network
  - `sess.run([train_op], feed_dict={X:x_batch, y:y_batch})`
- Query the values of necessary nodes
  - `val_loss, val_accuracy = sess.run([loss, accuracy], feed_dict={X:X_valid, y:y_valid})`

Note that as a rule of machine learning, we **must not** use the `test set` during the training phase and **only use** this set for evaluating the predictive performance of a trained model.

Putting everything altogether, the following snippet demonstrates the training process of our feedforward neural network.

The second last line `test_accuracy = sess.run(accuracy, feed_dict={X:X_test, y:y_test})` is to output the predictive performance on the test set.

In [ ]:

```
import math
batch_size = 32
history = []  #used to store train, valid accuracies and losses for showing later
num_epoch = 100
iter_per_epoch= math.ceil(float(train_size)/batch_size)  #number of iterations per

with tf.Session() as sess:
    init = tf.global_variables_initializer()
    sess.run(init)
    for epoch in range(num_epoch):
        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_e
            sess.run([train_op], feed_dict={X:X_batch, y:y_batch})
        #compute accuracies and losses at the end of each epoch
        train_summary, train_loss, train_accuracy= sess.run([merged,loss, accuracy
        train_writer.add_summary(train_summary, epoch +1)
        train_writer.flush()

        valid_summary,val_loss, val_accuracy= sess.run([merged,loss, accuracy], fe
        valid_writer.add_summary(valid_summary, epoch +1)
        valid_writer.flush()
        print("Epoch {}: valid loss={:.4f}, valid acc={:.4f}".format(epoch+1, val_
        print("########: train loss={:.4f}, train acc={:.4f}".format(train_loss, t
        hist_item={"train_loss": train_loss, "train_acc": train_accuracy,
                   "val_loss":val_loss, "val_acc": val_accuracy}
        history.append(hist_item)
    print("-----------------------------------------------\n")
    test_accuracy = sess.run(accuracy, feed_dict={X:X_test, y:y_test})
    print("Test accuracy: {:.4f}".format(test_accuracy))
```

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: Write code to add the plots of `test accuracy` and `loss` to the above line charts with your color of interest.

Exercise 3: 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/guide/summaries_and_tensorboard).

Exercise 4: 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.

THE END