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 02b: Stochastic Gradient Descent and Optimization

This tutorial aims to help you understand the idea of gradients and optimization in machine learning/deep learning algorithms using TensorFlow. The tutorial consists of two parts:

I. Visualization of the gradient of the training of a logistic regression model on a synthetic dataset

II. Demonstration the training procedure of a DNN model on the MNIST dataset

References and additional reading and resources

- An overview of gradient descent optimization algorithms (link (https://ruder.io/optimizing-gradient-descent/)).
- Logistic Regression (link (https://towardsdatascience.com/logistic-regression-detailed-overview-46c4da4303bc)).
- TensorFlow Keras model (link (https://www.tensorflow.org/api_docs/python/tf/keras/Model)).

I. Visualize the gradient of the training of a logistic regression model on a synthetic dataset ***** (highly important)

I.1 Introduction of logistic regression

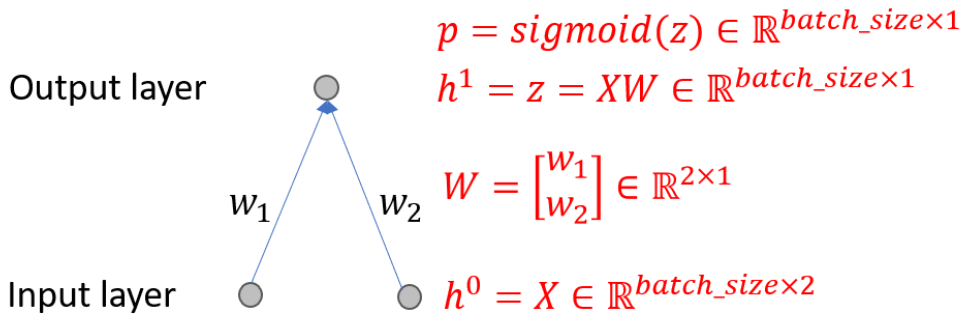
Logistic regression can be viewed as a simple neural network for binary classification with only two layers: input and output.

- The input layer takes inputs as one feature vector (data point, data instance, or data example), mini-batches of feature vectors, or entire dataset of feature vectors.
- The output layer has one neuron and was applied with the activation function, i.e., sigmoid to obtain the predicted probability of a given data instance to be in the positive class (i.e., the class 1).

To help you understand how forward/backward propagation works in training deep neural networks, we here provide a visualization of these propagation process for doing SGD (Stochastic Gradient Descend) updates to solve the optimization problem behind logistic regression. Progressing through this tutorial, you will engage in an end-to-end process of training a deep network, which includes feeding a mini-batch to a network, doing forward propagation, defining the loss function, and then performing backward propagation for computing the gradients for updating the model in an SGD manner.

For the sake of comprehensibility, we present the technical details for the case of a synthetic dataset with two features (i.e., data point $x \in \mathbb{R}^2$), meaning that the input layer has two neurons. However, the implementation can be generalized to a general case with many features (i.e., data point $x \in \mathbb{R}^d$).

The architecture of logistic regression is shown in the following figure.



The model parameters of our simple logistic regression include $W = \begin{bmatrix} w_1 \\ w_2 \end{bmatrix} \in \mathbb{R}^{2 \times 1}$. Here note that we do not use the bias to simplify the model for the visualizing purpose in the following steps. Let us denote the training set as $D = \{(x^1, y^1), (x^2, y^2) \ldots, (x^N, y^N)\}$ where data point and feature vector

Forward propagation

The computational process of logistic regression for a data point $x \in \mathbb{R}^{1 \times 2}$ with the label $y \in \{0, 1\}$ is as follows:

- We feed $x \in \mathbb{R}^{1 \times 2}$ to the logistic regression network.
- We compute: $z = xW \in \mathbb{R}^{1 \times 1}$.
- We apply sigmoid over z: $p = sigmoid(z) \in \mathbb{R}^{1 \times 1}$.

p represents the probability that x is classified as positive class or equivalently $p = p(y = 1 \mid x) = sigmoid(z) = \frac{1}{1+e^{-z}} = \frac{1}{1+e^{-xW}}$. The loss incurred is as follows:

- $l(x, y; W) = -y * log(p) - (1 - y) * log(1 - p)$. Your task is to explain why it is.

The loss incurred by the entire dataset is as follows:

- $l(W, D) = \frac{1}{N} \sum_{i=1}^{N} l(x^i, y^i; W)$

To efficiently solve the above optimization problem, we apply SGD in which at each iteration we feed a mini-batch X to the network and then rely on backward propagation to compute the gradient of the loss function w.r.t. to W for the current mini-batch.

The computational process of logistic regression for a mini-batch $X \in \mathbb{R}^{batch\_size \times 2}$ with labels $y \in \mathbb{R}^{batch\_size \times 1}$ is as follows:

- We feed a mini-batch $X = \begin{bmatrix} b_1^1 & b_2^1 \\ b_1^2 & b_2^2 \\ \dots & \dots \\ b_1^{batch\_size} & b_2^{batch\_size} \end{bmatrix} \in \mathbb{R}^{batch\_size \times 2}$ of 2D feature vectors (data points) to our logistic regression network.
- We compute $z = XW \in \mathbb{R}^{batch\_size \times 1}$.
- We apply sigmoid over z: $p = sigmoid(z) \in \mathbb{R}^{batch\_size \times 1}$.

The loss incurred by the current mini-batch is as follows:

- $l(W, X) = \frac{1}{batch\_size} \sum_{i=1}^{batch\_size} l(b^i, y^i; W) = -\frac{1}{batch\_size} \sum_{i=1}^{batch\_size} \left[y^i \log p^i + (1 - y^i) \log(1 - p^i) \right]$

Backward propagation

We now compute the gradient of the loss incurred by a mini-batch $l(W, X)$ w.r.t. the model parameter W. This can be done conveniently via backward propagation with some matrix multiplications. The mathematical tool for this derivation is the chain rule based on the computational process of the forward propagation.

- $z = XW$ --> $p = sigmoid(z)$ --> $l = -mean(y * log(p) + (1 - y) * log(1 - p))$ where $*$ is element-wise product between two vectors.

We derive as follows:

- $\frac{\partial l}{\partial W} = \frac{1}{batch\_size} \sum_{i=1}^{batch\_size} \frac{\partial l(b_i, y_i; W)}{\partial W} = \frac{1}{batch\_size} \sum_{i=1}^{batch\_size} \frac{\partial l(b_i, y_i; W)}{\partial p^i} \frac{\partial p^i}{\partial z^i} \frac{\partial z^i}{\partial W} = \frac{1}{batch\_size} \sum_{i=1}^{batch\_size}$

To facilitate the computation, we can rewrite the above derivative in the form of matrix multiplication as follows:

- $\frac{\partial l}{\partial W} = mean\left(X^T \left[\left(\frac{-y}{p} + \frac{1-y}{1-p} \right) * p * (1 - p) \right] \right) \in \mathbb{R}^{2 \times 1}$.

SGD update

We update the model based on the gradient of the mini-batch w.r.t. W as:

- $W = W - \eta * \frac{\partial l}{\partial W}$ where $\eta > 0$ is the learning rate.

I.2 Implementation of logistic regression

First, we create a simple synthetic dataset for logistic regressions

In [ ]:

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt
import copy
```

We would like to generate a few data points on a two-dimensional plane and assume that the optimal coefficient of the logistic regression is known to us.

In [ ]:

```
W_opt = np.array([[1],[2]], dtype=np.float64) # the optimal W
print(W_opt)
```

We generate the training set of 2D data points.

In [ ]:

```
# Randomly generate 500 synthetic data points
N = 500
delta = 2
X1 = (np.random.rand(N) -0.5)*2*delta
X2 = (np.random.rand(N) -0.5)*2*delta
X_train = np.array([x for x in zip(X1,X2)])
print("X_train shape: {}".format(X_train.shape))
```

We assign labels (i.e., 0 or 1) for the data points based on whether they stay on what side of the optimal hyperplane (W_opt).

In [ ]:

```
# Generate the labels for those data points
y_value = X_train.dot(W_opt) + np.random.normal(0.0, 0.2)
y_train = np.ones([N,1])
y_train[np.where(y_value <0)]= 0
num_pos = len(np.argwhere(y_train==1)[:,0])
num_neg = len(np.argwhere(y_train==0)[:,0])
print("Number of positive labels: {}, number of negative labels: {}".format(num_po
```

We now visualize the data points with labels and the optimal decision boundary.

In [ ]:

```
# The following code visualizes the hyperplane, i.e., the decision boundary, which
def visualize_hyperplane(X= None, y= None, W=None):
    X_pos = X[np.argwhere(y ==1)[:,0]]
    X_neg = X[np.argwhere(y ==0)[:,0]]
    plt.scatter(X_pos[:,0], X_pos[:,1], label='1',  c='green', marker='+')
    plt.scatter(X_neg[:,0], X_neg[:,1], label='0', c='blue', marker='_')
    plt.legend(loc='upper right')
    f = lambda x1: -W[0,0]*x1/W[1,0] #Plot the decision boundary
    X1 = np.linspace(-2,2,500)
    plt.plot(X1, f(X1),  'r--')
    plt.show()
```

In [ ]:

```
visualize_hyperplane(X_train, y_train, W_opt)
```

Now we define the logistic regression model

In [ ]:

```
# Define the sigmoid function
def sigmoid(z):
    return 1.0/ (1.0 + np.exp(-z))
```

Below is the code for **forward propagation** w.r.t. the mini-batch X or the set of data points X.

In [ ]:

```
# Define the logistic regression model and the corresponding loss, which is simila
def forward(X= None, y=None, W= None, eps= 1E-10):
    z = X.dot(W)
    p = sigmoid(z)
    losses = [-np.log(p[i]+eps) if y[i]==1 else -np.log(1- p[i]+eps) for i in rang
    loss = np.mean(losses)
    return (z,p,loss)
```

In [ ]:

```
W = np.array([[-1],[1]])
X, y = X_train, y_train
eps = 1e-10
z = X.dot(W)
p = sigmoid(z)
# losses_1 = [-np.log(p[i]+eps) if y[i]==1 else -np.log(1- p[i]+eps) for i in rang
# losses_2 = np.where(y==1, -np.log(p+eps), -np.log(1- p+eps))
# np.allclose(losses_1, losses_2)
```

Below is the code for **backward propagation** in which we compute the gradient of the loss of X w.r.t. W.

In [ ]:

```
# Compute the gradient of the loss by the chain rule, which is similar to backprop
def grad(X= None, y= None, z= None, p=None, eps= 1E-10):
    batch_size = len(y)
    grad_p = [-1.0/(p[i] + eps) if y[i]==1 else 1.0/(1-p[i]+ eps) for i in range(b
    grad_p = np.array(grad_p).reshape([batch_size,1])
    grad_z = grad_p*p*(1-p)
    grad_W = X.transpose().dot(grad_z)*(1.0/batch_size)  # refer to the formula to
    return grad_W
```

Here we train our logistic regression model on the synthetic dataset with gradient descent

We consider three schemes:

- (i) **Gradient descent** where we use the entire dataset (i.e., X is the entire dataset) to compute the gradient.
- (ii) **SGD** where we use mini-batches to compute gradient for which the mini-batches are uniformly sampled from the training set at each iteration.
- (iii) **SGD** where we use mini-batches to compute gradient, but we first shuffle the training set, split the training set into many equal mini-batches with the same number of data points (i.e., $batch\_size$). In each iteration we then feed a mini-batch to the network. Although this mini-batch generating strategy is a bit different from the theory of SGD, it always obtains comparable performance compared to (ii), hence widely being used in training deep learning models.

In [ ]:

```
from IPython.display import clear_output
from pylab import rcParams
rcParams['figure.figsize'] = 10, 5

# Visualize the hpyterplane and the training loss
def visualize_hyperplane_loss(X= None, y= None, W= None, losses= None):
    X_pos = X[np.where(y ==1)[0]]
    X_neg = X[np.where(y ==0)[0]]
    clear_output(wait=True)

    plt.subplot(1,2,1)
    plt.scatter(X_pos[:,0], X_pos[:,1], label='1',  c='green', marker='+')
    plt.scatter(X_neg[:,0], X_neg[:,1], label='0', c='blue', marker='_')
    plt.legend(loc='upper right')

    f = lambda x1: -W[0,0]*x1/W[1,0] # Plot the decision boundary

    X1 = np.linspace(-2,2,500)
    plt.plot(X1, f(X1),  'r--')
    plt.subplot(1,2,2)
    plt.plot(losses, label = 'Epoch loss', marker='o')

    if len(losses):
        txt="Epoch %.2d W: [[%.3f], [%.3f]]; Loss: %.3f" % (len(losses), W[0,0], W
        plt.figtext(0.5, 0.01, txt, wrap=True, horizontalalignment='center', fonts

    plt.show()
```

Below is the code to implement **(i) Gradient Descent** optimization.

In [ ]:

```
W = np.array([[-1],[1]])
lst_W1 = [copy.copy(W)]
epochs = 30
eta = 0.5
plt.ion()
losses = []
for epoch in range(epochs):
    X = X_train
    y= y_train
    z, p, loss= forward(X,y, W)
    losses.append(loss)
    # Compute gradient
    grad_W = grad(X,y,z,p)
    # Update coefficients with gradient descent
    W = W - eta*grad_W
    lst_W1.append(copy.copy(W))

    # We update the coefficients through all the data points
    visualize_hyperplane_loss(X_train, y_train, W, losses)
    plt.pause(0.5)
plt.ioff()
```

Below is the code to implement **(ii) SGD**.

In [ ]:

```
plt.ion()
W = np.array([[-1],[1]])
lst_W2 = [copy.copy(W)]
epochs =30
eta = 0.5
batch_size = 16
iter_per_epoch = int(N/batch_size)
epoch_losses= []
for epoch in range(epochs):
    losses =[]
    # We update the coefficients in each random batch
    for i in range(iter_per_epoch):
        # randomly sample a batch of data points
        idxs= np.random.choice(np.arange(N), batch_size, replace= False)
        X= X_train[idxs]
        y = y_train[idxs]
        z, p, batch_loss= forward(X, y,W)

        losses.append(batch_loss)
        grad_W = grad(X,y,z,p)
        W = W - eta*grad_W

    lst_W2.append(copy.copy(W))
    epoch_losses.append(np.mean(losses))
    visualize_hyperplane_loss(X_train, y_train, W, epoch_losses)
    plt.pause(0.5)
plt.ioff()
```

Below is the code to implement the third strategy **(iii) SGD.**

In [ ]:

```
plt.ion()
W = np.array([[-1],[1]])
lst_W3 = [copy.copy(W)]
epochs =30
eta = 0.5
batch_size = 16
iter_per_epoch = int(N/batch_size)
epoch_losses =[]

p = np.random.permutation(len(X_train))
X_train_shuff = X_train[p]
y_train_shuff = y_train[p]

for epoch in range(epochs):
    losses = []
    # We update the coefficients in each sequential batch
    for idx_start in range(0, N, batch_size):
        # We go through the batches sequencially, i.e., non-randomly
        idx_end = min(N, idx_start + batch_size)
        X= X_train_shuff[idx_start:idx_end]
        y= y_train_shuff[idx_start:idx_end]
        z, p, batch_loss= forward(X, y, W)

        losses.append(batch_loss)
        grad_W = grad(X, y, z,p)
        W = W - eta*grad_W

    lst_W3.append(copy.copy(W))
    epoch_losses.append(np.mean(losses))
    visualize_hyperplane_loss(X_train_shuff, y_train_shuff, W, epoch_losses)
    plt.pause(0.5)
plt.ioff()
```

We now visualize the trajectories of the models for three strategies (i), (ii), and (iii).

In [ ]:

```
from pylab import rcParams
rcParams['figure.figsize'] = 15, 10
def visualize_W(lst_W1= None, lst_W2= None, lst_W3= None, k=1, W_opt= None):
    clear_output(wait=True)
    plt.plot([lst_W1[i][0] for i in range(k)],[lst_W1[i][1] for i in range(k)], "-
    plt.plot([lst_W2[i][0] for i in range(k)],[lst_W2[i][1] for i in range(k)], "-
    plt.plot([lst_W3[i][0] for i in range(k)],[lst_W3[i][1] for i in range(k)], "-

    # plot some optimal values W*
    x = W_opt[0, 0] * np.linspace(0, 4, 10)
    y = W_opt[1, 0] * x
    plt.plot(x, y, "*", markersize= 10, label='W*', color='red')

    plt.legend(loc="upper left")
    plt.show()
```

In [ ]:

```
plt.ion()
for i in range(len(lst_W1)):
    visualize_W(lst_W1, lst_W2, lst_W3, i+1, W_opt)
    plt.pause(0.1)
plt.ioff()
```

Conclusion

From the above experiments, we have the following conclusions:

- In this example, the performances of the 3 approaches are comparable with the same learning rate.
- In terms of convergence, although GD has the better direction to downhill towards the global minima, its loss value is a bit higher than SGD (i) and (ii), which suggests a higher learning rate for GD.
- During the training, SGD (i) and (ii) only compute the gradient on minibatches. This approximation makes noise, therefore helpful for escaping the saddle points/local minima/local maxima faster. However, a disadvantage of SGD is its oscillation, especially when setting with a high learning rate, which would take more time to converge. This motivates the researchers to develop better algorithms, namely, SGD with momentum, and later RMSprop, and Adam.

Exercise 1: Try with smaller learning rates and different initial W to compare the models obtained by three strategies in (i), (ii), and (iii). Report your observations.

Exercise 2: Implement SGD with momentum (refer here (https://ruder.io/optimizing-gradient-descent/)) and compare with the standard SGD in terms of the convergence rate to the optimal solution.

II. Implement a DNN model on the MNIST dataset with SGD **** (important)

This part shows you how to build up flexible deep learning models with a composite loss: $loss1 + \alpha \times loss2 + \beta \times loss3$ where $\alpha > 0$ and $\beta > 0$ are two trade-off parameters. You will later learn see the benefits of shuffling a dataset, splitting a dataset into subsets, and batching a dataset.

In this example, we will build a DNN model for a classification problem. The dataset is the handwritten digits which has a training set of 60,000 examples, and a test set of 10,000 examples. Each image is an black-white image with resolution 28x28 pixels. The dataset splits to 10 classes (digit 0 to 9). The DNN model consits of 3 dense layers: $784 \rightarrow 20(ReLU) \rightarrow 20(ReLU) \rightarrow 10(softmax)$.

The objective loss is composited of two losses $L = L_1 + \alpha \times L_2$ where

- L_1 is the standard cross entropy loss: $L_1 = -\sum_{i=1}^{10} y_i \log p_i$
- L_2 is the entropy loss which helps to encourage a more confident prediction (by minimizing the prediction entropy): $L_2 = -\sum_{i=1}^{10} p_i \log p_i$

II.1 We first load the MNIST dataset in TensorFlow

In [ ]:

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

In [ ]:

```
mnist = tf.keras.datasets.mnist
(X_train_full_img, y_train_full), (X_test_img, y_test) = mnist.load_data()
```

In [ ]:

```
num_train = X_train_full_img.shape[0]
num_test = X_test_img.shape[0]
X_train_full = X_train_full_img.reshape(num_train,-1)/255.0
X_test = X_test_img.reshape(num_test, -1)/255.0
print(X_train_full.shape, y_train_full.shape)
print(X_test.shape, y_test.shape)
```

II.2 We build a deep neural network with several fully-connected layers

In [ ]:

```
class DNN:
    def __init__(self, n_classes= 10, optimizer= tf.keras.optimizers.SGD(learning_
                 batch_size= 32, epochs= 20, alpha=1.0, beta=1.0):
        self.n_classes= 10
        self.batch_size= batch_size
        self.epochs = epochs
        self.optimizer = optimizer
        self.alpha = alpha # hyper-parameter corresponding to entropy loss
        self.beta = beta # hyper-parameter for the max-margin loss

        # create a tensorflow dataset and shuffle
        self.train_full_set = tf.data.Dataset.from_tensor_slices((X_train_full, y_
        # take train and valid sets from full dataset
        self.train_set = self.train_full_set.take(50000)
        self.valid_set = self.train_full_set.skip(50000).take(10000)
        # batching train and valid sets
        self.train_set= self.train_set.batch(self.batch_size).prefetch(1)
        self.valid_set= self.valid_set.batch(self.batch_size).prefetch(1)
        tf.keras.backend.set_floatx('float64')

    def build(self):
        self.model= Sequential([Dense(20, activation='relu'), Dense(20, activation
                                Dense(self.n_classes, activation= 'softmax')])

    def compute_loss(self, X, y): # X is data batch, y is label batch
        pred_probs = self.model(X)
        l1 =  tf.keras.losses.sparse_categorical_crossentropy(y, pred_probs) # Cro
        l2 = tf.reduce_sum(- pred_probs * tf.math.log(pred_probs), axis=-1) # Pred

        assert (l1.shape == l2.shape)
        return l1 + self.alpha * l2

    def compute_grads(self, X, y):
        with tf.GradientTape() as g:  # use gradient tape to compute gradients
            loss= self.compute_loss(X,y)
        grads= g.gradient(loss, self.model.trainable_variables) # compute gradient
        return grads

    def train_one_batch(self, X, y): # train in one batch
        grads= self.compute_grads(X,y)
        # the gradients will be applied according to optimizer for example SGD, Ad
        self.optimizer.apply_gradients(zip(grads, self.model.trainable_variables))

    def evaluate(self, tf_dataset= None):
        dataset_loss = tf.keras.metrics.Mean()
        dataset_accuracy = tf.keras.metrics.SparseCategoricalAccuracy()
        for X,y in tf_dataset:
            loss = self.compute_loss(X,y)
            dataset_loss.update_state(loss)
            dataset_accuracy.update_state(y, self.model(X, training= False))
        return dataset_loss.result(), dataset_accuracy.result()


    def train(self):
        for epoch in range(self.epochs):
            for X,y  in self.train_set: # use batch_index if you want to display s
                self.train_one_batch(X,y)
            train_loss, train_acc = self.evaluate(self.train_set)
            valid_loss, valid_acc = self.evaluate(self.valid_set)
```

```
            print('Epoch {}: train acc={:.4f}, train loss={:.4f} | valid acc={:.4f
```

II.3 Next we train the DNN on MNIST using SGD

In [ ]:

```
opt = tf.keras.optimizers.SGD(learning_rate=0.001)
dnn = DNN(optimizer=opt, epochs=10, batch_size=64)
dnn.build()
```

In [ ]:

```
dnn.train()
```

Exercise 3: Extend the above class `DNN` to allow evaluating on the test set.

Exercise 4: Develop a feed-forward neural network to work with the MNIST dataset according to the following requirements:

- The architecture is $Input \rightarrow 20(ReLU) \rightarrow 40(ReLU) \rightarrow 20(ReLU) \rightarrow Output(softmax)$.
- Extend the code to allow doing grid search for tunining the learning rate in the list $[0.1, 0.01, 0.001]$ and optimizer in the list $[tf.keras.optimizers.Adam(), tf.keras.optimizers.RMSprop(), tf.keras.optimizers.SGD(n$ and save the best model to the hard disk (link (https://www.tensorflow.org/guide/keras/save_and_serialize)). Plot the training progress of the best model (e.g., training and valid accuracies, training and valid losses).
- Load the best model and evaluate on the test set.

THE END