

mlp_d

October 16, 2019

1 IST597:- Multi-layer Perceptron

#Week 8 tutorial Building your first MLP with dropout in eager Author :- aam35

```
In [2]: # -*- coding: utf-8 -*-
        """
        Author:-aam35
        """

        import numpy as np
        import os
        import sys
        import tensorflow as tf
        from tensorflow.examples.tutorials.mnist import input_data
        import time
        tf.enable_eager_execution()
        tf.executing_eagerly()

        # random seed to get the consistent result
        tf.random.set_random_seed(1234)

        data = input_data.read_data_sets("data/MNIST_data/", one_hot=True)

        minibatch_size = 32
        learning_rate = 0.01

        Extracting data/MNIST_data/train-images-idx3-ubyte.gz
        Extracting data/MNIST_data/train-labels-idx1-ubyte.gz
        Extracting data/MNIST_data/t10k-images-idx3-ubyte.gz
        Extracting data/MNIST_data/t10k-labels-idx1-ubyte.gz

In [0]: ## model 2
        size_input = 784 # MNIST data input (img shape: 28*28)
        size_hidden = 256
        size_output = 10 # MNIST total classes (0-9 digits)
```

```

# Define class to build mlp model
class MLP_2(object):
    def __init__(self, size_input, size_hidden, size_output, device=None):
        """
        size_input: int, size of input layer
        size_hidden: int, size of hidden layer
        size_output: int, size of output layer
        device: str or None, either 'cpu' or 'gpu' or None. If None, the device to be used
        """
        self.size_input, self.size_hidden, self.size_output, self.device = \
            size_input, size_hidden, size_output, device

        # Initialize weights between input layer and hidden layer
        self.W1 = tf.Variable(tf.random_normal([self.size_input, self.size_hidden], stddev=0.1))
        # Initialize biases for hidden layer
        self.b1 = tf.Variable(tf.zeros([1, self.size_hidden]), name = "b_1")
        # Initialize weights between hidden layer and output layer
        self.W2 = tf.Variable(tf.random_normal([self.size_hidden, self.size_output], stddev=0.1))
        # Initialize biases for output layer
        self.b2 = tf.Variable(tf.random_normal([1, self.size_output]), name="b_2")

        # Define variables to be updated during backpropagation
        self.variables = [self.W1, self.b1, self.W2, self.b2]

    # prediction
    def forward(self, X):
        """
        forward pass
        X: Tensor, inputs
        """
        if self.device is not None:
            with tf.device('gpu:0' if self.device=='gpu' else 'cpu'):
                self.y = self.compute_output(X)
        else:
            self.y = self.compute_output(X)

        return self.y

    ## loss function
    def loss(self, y_pred, y_true):
        """
        y_pred - Tensor of shape (batch_size, size_output)
        y_true - Tensor of shape (batch_size, size_output)
        """

```

```

    """
    y_true_tf = tf.cast(tf.reshape(y_true, (-1, self.size_output)), dtype=tf.float32)
    y_pred_tf = tf.cast(y_pred, dtype=tf.float32)
    return tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=y_pred_tf, labels=y_true_tf))

def backward(self, X_train, y_train):
    """
    backward pass
    """
    # optimizer
    # Test with SGD, Adam, RMSProp
    optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
    #optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
    with tf.GradientTape() as tape:
        predicted = self.forward(X_train)
        current_loss = self.loss(predicted, y_train)
    grads = tape.gradient(current_loss, self.variables)
    optimizer.apply_gradients(zip(grads, self.variables),
                              global_step=tf.train.get_or_create_global_step())

def compute_output(self, X):
    """
    Custom method to obtain output tensor during forward pass
    """
    # Cast X to float32
    X_tf = tf.cast(X, dtype=tf.float32)
    #Remember to normalize your dataset before moving forward
    # Compute values in hidden layer
    what = tf.matmul(X_tf, self.W1) + self.b1
    hhat = tf.nn.relu(what)
    hhat_tilda = tf.compat.v1.nn.dropout(hhat, rate=0.2)
    # Compute output
    output = tf.matmul(hhat_tilda, self.W2) + self.b2
    #Now consider two things , First look at inbuilt loss functions if they work with
    #Second add tf.Softmax(output) and then return this variable
    #print(output)
    return (output)
    #return output

```

```

In [0]: def accuracy_function(yhat, true_y):
        correct_prediction = tf.equal(tf.argmax(yhat, 1), tf.argmax(true_y, 1))
        accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
        return accuracy

```

```

In [7]: # Initialize model using GPU
        mlp_on_cpu = MLP_2(size_input, size_hidden, size_output, device='cpu')

```

```

num_epochs = 8

time_start = time.time()
num_train = 55000

for epoch in range(num_epochs):
    train_ds = tf.data.Dataset.from_tensor_slices((data.train.images, data.train.labels))\
        .shuffle(buffer_size=1000)\
        .batch(batch_size=minibatch_size)
    loss_total = tf.Variable(0, dtype=tf.float32)
    for inputs, outputs in train_ds:
        preds = mlp_on_cpu.forward(inputs)
        loss_total = loss_total + mlp_on_cpu.loss(preds, outputs)
        mlp_on_cpu.backward(inputs, outputs)
    print('Number of Epoch = {} - loss:= {:.4f}'.format(epoch + 1, loss_total.numpy()))
    preds = mlp_on_cpu.compute_output(data.train.images)
    accuracy_train = accuracy_function(preds, data.train.labels)
    accuracy_train = accuracy_train * 100
    print ("Training Accuracy = {}".format(accuracy_train.numpy()))

    preds_val = mlp_on_cpu.compute_output(data.validation.images)
    accuracy_val = accuracy_function(preds_val, data.validation.labels)
    accuracy_val = accuracy_val * 100
    print ("Validation Accuracy = {}".format(accuracy_val.numpy()))

    # test accuracy
    preds_test = mlp_on_cpu.compute_output(data.test.images)
    accuracy_test = accuracy_function(preds_test, data.test.labels)
    # To keep sizes compatible with model
    accuracy_test = accuracy_test * 100
    print ("Test Accuracy = {}".format(accuracy_test.numpy()))

time_taken = time.time() - time_start
print('\nTotal time taken (in seconds): {:.2f}'.format(time_taken))
#For per epoch_time = Total_Time / Number_of_epochs

```

```

Number of Epoch = 1 - loss:= 0.0208
Training Accuracy = 87.4618148803711
Validation Accuracy = 87.94000244140625
Number of Epoch = 2 - loss:= 0.0118
Training Accuracy = 89.81090545654297
Validation Accuracy = 90.81999969482422
Number of Epoch = 3 - loss:= 0.0100
Training Accuracy = 91.29090881347656

```

Validation Accuracy = 91.72000122070312
Number of Epoch = 4 - loss:= 0.0089
Training Accuracy = 91.97090911865234
Validation Accuracy = 92.91999816894531
Number of Epoch = 5 - loss:= 0.0081
Training Accuracy = 92.81818389892578
Validation Accuracy = 93.05999755859375
Number of Epoch = 6 - loss:= 0.0075
Training Accuracy = 93.28545379638672
Validation Accuracy = 93.62000274658203
Number of Epoch = 7 - loss:= 0.0070
Training Accuracy = 93.76000213623047
Validation Accuracy = 94.27999877929688
Number of Epoch = 8 - loss:= 0.0066
Training Accuracy = 94.06181335449219
Validation Accuracy = 94.77999877929688
Test Accuracy = 94.29000091552734

Total time taken (in seconds): 200.06

2 Things to do

- Change dropout rate
- Is dropout always beneficial? If not when do you think it is beneficial[Explore this practically or theoretically, Publish your findings]
- Effect of dropout on deeper model vs shallow model.