mlp_2

October 16, 2019

1 IST597:- Multi-layer Perceptron

#Week 8 tutorial Building your first MLP 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]:
```
Simple MLP
```

In [0]:
```
## model 1
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(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 us
        """
        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],stdde
        # Initialize biases for hidden layer
        self.b1 = tf.Variable(tf.zeros([1, self.size_hidden]), name = "b1")
        # Initialize weights between hidden layer and output layer
        self.W2 = tf.Variable(tf.random_normal([self.size_hidden, self.size_output],stdd
        # Initialize biases for output layer
        self.b2 = tf.Variable(tf.random_normal([1, self.size_output]),name="b2")



        # 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_t


    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)
        # Compute output
        output = tf.matmul(hhat, self.W2) + self.b2
        #Now consider two things , First look at inbuild loss functions if they work wit
        #Second add tf.Softmax(output) and then return this variable
        #print(output)
        return (output)
        #return output
```

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)
```

```python
# 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 us
        """
        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],stdde
        # 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],stdd
        # 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_t


    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 inbuild loss functions if they work wit
        #Second add tf.Softmax(output) and then return this variable
        #print(output)
        return (output)
        #return output
```

MLP with regularizer

```
In [0]: ## model 3
        size_input = 784 # MNIST data input (img shape: 28*28)
        size_hidden = 256
        size_output = 10 # MNIST total classes (0-9 digits)
        beta = 0.01
```

```
# Define class to build mlp model
class MLP_3(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 us
        """
        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],stdde
        # Initialize biases for hidden layer
        self.b1 = tf.Variable(tf.zeros([1, self.size_hidden]), name = "b_13")
        # Initialize weights between hidden layer and output layer
        self.W2 = tf.Variable(tf.random_normal([self.size_hidden, self.size_output],stdd
        # Initialize biases for output layer
        self.b2 = tf.Variable(tf.random_normal([1, self.size_output]),name="b_23")



        # 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)
        loss = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits_v2(logits=y_pred_t
        regularizers = tf.nn.l2_loss(self.W1) + tf.nn.l2_loss(self.W2)
        Reg_loss = tf.reduce_mean(loss + beta * regularizers)
        return Reg_loss


    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)
        # Compute output
        output = tf.matmul(hhat, self.W2) + self.b2
        #Now consider two things , First look at inbuild loss functions if they work wit
        #Second add tf.Softmax(output) and then return this variable
        #print(output)
        return (output)
        #return output
```

create accuracy function which takes prediction and targets as input

```
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 [22]:
```
# Initialize model using GPU
mlp_on_cpu = MLP(size_input, size_hidden, size_output, device='gpu')

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.la
           .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:= 0.0193
Training Accuracy = 89.207275390625
Validation Accuracy = 89.5
Number of Epoch = 2 - loss:= 0.0105
Training Accuracy = 91.04545593261719
Validation Accuracy = 91.23999786376953
```

```
Number of Epoch = 3 - loss:= 0.0090
Training Accuracy = 92.18909454345703
Validation Accuracy = 92.47999572753906
Number of Epoch = 4 - loss:= 0.0080
Training Accuracy = 92.90545654296875
Validation Accuracy = 93.13999938964844
Number of Epoch = 5 - loss:= 0.0073
Training Accuracy = 93.53636169433594
Validation Accuracy = 93.68000030517578
Number of Epoch = 6 - loss:= 0.0067
Training Accuracy = 94.18000030517578
Validation Accuracy = 94.27999877929688
Number of Epoch = 7 - loss:= 0.0062
Training Accuracy = 94.54181671142578
Validation Accuracy = 94.54000091552734
Number of Epoch = 8 - loss:= 0.0058
Training Accuracy = 94.96726989746094
Validation Accuracy = 94.94000244140625
Test Accuracy = 94.66000366210938

Total time taken (in seconds): 136.07
```

For one layer feed-forward we can easily reach in high 90's # Things to try * Increase the depth or layers of this model and analyze the change in convergence * Change the optimizer , play with learning rate , try various activation functions. * You can also use various initialization method[We have used random_init, play with xavier or orthogonal] * Tweak the network, for better understanding how hyper-parameters plays a vital role in achieving state of the art.