315-Hw-3

March 9, 2024

Homework 3: In this homework, you will train a two layer neural network using gradient descent. However, instead of manually computing the gradients, you will use the autodiff provided by Tensorflow package.

[1]:
```
import numpy as np
import tensorflow as tf
```

1 Two-layer Neural Network

A two layer neural network contains an input layer, a hidden layer, and an output layer. The number of the input nodes in the diagram below is determined the dimension of our features. We are free to choose the dimension of the hidden layer. As for the final output layer, the number of nodes is determined by the type of the problem. For instance, for a regression problem, we will only have one node and the output value corresponds to our prediction of the target. As for classification, we will first output a vector that has same number of dimension as the number of classes in our classification dataset. Then, we will apply the softmax transformation on the vector to transform real-valued predictions to the class probabilities.

Mathematically, this model can be written as

$$f(x) = \sigma(x^\top W_1 + b_1)W_2 + b_2.$$

Note that if $x \in \mathbb{R}^{d\times 1}$, then $W_1 \in \mathbb{R}^{d\times H}$ and $W_2 \in \mathbb{R}^{H\times O}$, where O is the output dimension. The dimension of b_1 and b_2 is self-evident.

Given an input x, the model first transforms it using the weight matrix W_1 and subsequently shifts the output by the bias term b_1. The function $\sigma(.)$ is called the activation function that introdues non-linearity in the model. For the purpose of this homework, we will use the so called relu-activation function that is defined as $\sigma(t) = \max\{t, 0\}$. Note that that $x^\top W_1 + b_1$ generally gives us a vector, so we have to apply the relu activation to each element of the vector. Following the activation, the vector $h(x) = \sigma(x^\top W_1 + b_1)$ is defines the hidden layer. Finally, we apply the linear trasformation $ h(x) W_1 + b_2$ on the hidden layer.

This representation of the network is very convenient if you instead want to do matrix operations on your data. Suppose X be your data matrix where i^{th} row of X contains $x_i^\top$, then the output of the network on the entire dataset can be written as

$$f(X) = \sigma(XW_1 + b_1)W_2 + b_2.$$

2 Regression

In homework 2, you trained a linear regression model on california housing dataset. In this homework, you will train a two-layer neural network on this dataset using gradient descent.

[15]:
```
# loading the dataset
from sklearn.datasets import fetch_california_housing
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler

california_housing = fetch_california_housing( return_X_y=True, as_frame=True)
X = california_housing[0]
y = california_housing[1]
X_train_unscaled, X_test_unscaled, y_train, y_test = train_test_split(X, y,␣
 ↪test_size=0.3, random_state=42)
sc=StandardScaler()
X_train=sc.fit_transform(X_train_unscaled)
X_test = sc.transform(X_test_unscaled)
```

[16]:
```
# converting numpy arrays to tensors

X_train = tf.convert_to_tensor(X_train, dtype = tf.float32)
X_test = tf.convert_to_tensor(X_test, dtype = tf.float32)
y_train = tf.convert_to_tensor(y_train, dtype = tf.float32)
y_test= tf.convert_to_tensor(y_test, dtype = tf.float32)
```

[17]:
```
print(X_train)
print(y_train)
```

```
tf.Tensor(
[[ 0.13350628  0.50935745  0.18106018 … -0.01082519 -0.8056819
   0.78093404]
 [-0.53221804 -0.6798731  -0.42262954 … -0.08931585 -1.3394727
   1.2452699 ]
 [ 0.1709897  -0.36274496  0.07312834 … -0.04480037 -0.49664515
  -0.27755183]
 …
 [-0.49478713  0.5886395  -0.59156984 …  0.01720102 -0.75885814
   0.60119116]
 [ 0.967171   -1.0762833   0.39014888 …  0.00482125  0.903385
  -1.186252  ]
 [-0.6832017   1.8571521  -0.82965606 … -0.0816717   0.99235016
  -1.4159235 ]], shape=(14448, 8), dtype=float32)
tf.Tensor([1.938 1.697 2.598 … 2.221 2.835 3.25 ], shape=(14448,),
dtype=float32)
```

Question 1: Fill in the input dimension and output dimension of the two layer neural network for regression on this dataset. (5 pts)

For the purpose of this homework, we will just choose hidden layer with dimension double that of input dimension. However, going forward choosing the hidden dimension appropriately would be an important part of deep learning.

[29]:
```
# replace ______ with your code
input_dim = X_train.shape[1]
hidden_dim = 2 * input_dim
output_dim = 1
```

Question 2: Define a tensorflow variables for weights W1, b1, W2, and b2. Then, initialize both biases b1 and b2 to be 0 vectors and initialize W1 and W2 by picking values uniformly at random from the interval [0,1]. (5 pts)

[56]:
```
# replace ______ with your code
W1 = tf.Variable(tf.random.uniform(shape=(input_dim, hidden_dim), minval=0,␣
 ↪maxval=1))
b1 = tf.Variable(tf.zeros(shape=(hidden_dim)))
W2 = tf.Variable(tf.random.uniform(shape=(hidden_dim, output_dim), minval=0,␣
 ↪maxval=1))
b2 = tf.Variable(tf.zeros(shape=(output_dim)))
```

[57]:
```
print(W1.shape, b1.shape, W2.shape, b2.shape)
```

```
(8, 16) (16,) (16, 1) (1,)
```

Question 3: Complete the model function below to define a two layer neural network. Here, inputs is a matrix of shape $n \times d$, where i^{th} row of the inputs matrix contains $x_i^\top$. (10 pts)

Hint: Use tf.nn.relu() function for relu activation.

[58]:
```
# Replace "________" with your code
def model(inputs):
  hidden = tf.nn.relu(tf.matmul(inputs, W1) + b1) # hidden layer with␣
 ↪relu-activation
  output = tf.matmul(hidden, W2) + b2  # fully-connected output linear layer
  return output
```

Question 4: Write a function below that computes mean-squared error given the predictions and targets. You should use tensorflow operations like tf.square and tf.reduce_mean for autodiff to work. Note that you cannot use inbuilt tensorflow MSE function. (10 pts)

[59]:
```
# Replace "________" with your code
def mse(predictions, targets):
  squared_error = tf.square(predictions - targets)
  mean_squared_error = tf.reduce_mean(squared_error)
  return mean_squared_error
```

Question 5: Complete the function below that takes in features and targets and trains your two layer neural network using gradient descent. Please use autodiff by Gradient taping. (10 pts)

[60]:
```
# Replace "________" with your code
learning_rate = 0.01

def train(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = model(inputs)   # predict
        loss = mse(predictions, targets)  # loss

    # tape gradients of the 4
    gradients = tape.gradient(loss, [W1, b1, W2, b2])

    # update weights and biases
    W1.assign_sub(learning_rate * gradients[0])
    b1.assign_sub(learning_rate * gradients[1])
    W2.assign_sub(learning_rate * gradients[2])
    b2.assign_sub(learning_rate * gradients[3])
```

Question 6: Complete the routine below that divides the randomly shuffled data into multiple mini-batches of size 1000 and use the train function above to run gradient descent on those minibatches. Within each step, you should make a complete pass through the dataset. (10 pts)

[61]:
```
B = 1000
n = X_train.shape[0]
k = int(n/B)
for step in range(100):
    X_train = tf.random.shuffle(X_train, seed = step)   #separately shuffling␣
 ↪X_train and y_train is reasonable here because of the same seed
    y_train = tf.random.shuffle(y_train, seed = step)
  ␣
 ↪#############################################################################
    #                               Write your code here                        ␣
 ↪#
  ␣
 ↪#############################################################################
    for i in range(k):
      # update batch
      start = i * B
      end = start + B
      X_batch = X_train[start:end , ]
      y_batch = y_train[start:end]

      # train
      train(X_batch, y_batch)

    # compute batch loss
    loss = mse(model(X_train), y_train)
    if (step+1) % 10 == 0:
```

```
        print(f"Loss at step {step}: {loss:.4f}")
```

```
Loss at step 9: 1.4517
Loss at step 19: 1.3864
Loss at step 29: 1.3674
Loss at step 39: 1.3579
Loss at step 49: 1.3524
Loss at step 59: 1.3491
Loss at step 69: 1.3470
Loss at step 79: 1.3456
Loss at step 89: 1.3446
Loss at step 99: 1.3439
```

Question 7: Get predictions on your test data set and report the MSE loss on the test data. (5 pts)

[62]:
```
predictions_test = model(X_test)
test_loss = mse(predictions_test, y_test)
print(f"MSE Loss on the test data: {test_loss:.4f}")
```

```
MSE Loss on the test data: 1.3166
```

3 Multiclass Classification

Next, you will train the two layer neural network to do multiclass classification on digits dataset. Digits data is similar to MNIST but has even smaller pixel.

[63]:
```
from sklearn.datasets import load_digits
X,y_int = load_digits(return_X_y=True)
```

In HW2, you manually created an one hot encoding of the multiclass targets. We can use a function from keras, which is a high level deep learning API thats works pretty well with tensorflow.

[64]:
```
from keras.utils import to_categorical
y_one_hot = to_categorical(y_int, num_classes=10)
```

[65]:
```
# training-testing split and appropriate rescaling
X_train_unscaled, X_test_unscaled, y_train, y_test = train_test_split(X,␣
 ↪y_one_hot, test_size=0.3, random_state=42)
sc=StandardScaler()
X_train=sc.fit_transform(X_train_unscaled)
X_test = sc.transform(X_test_unscaled)
```

[66]:
```
#convert to tensors
X_train = tf.convert_to_tensor(X_train, dtype = tf.float32)
X_test = tf.convert_to_tensor(X_test, dtype = tf.float32)
y_train = tf.convert_to_tensor(y_train, dtype = tf.float32)
y_test= tf.convert_to_tensor(y_test, dtype = tf.float32)
```

Question 8: Fill in the input dimension and output dimension of the two layer neural network appropriate for this dataset. (5 pts)

[69]:
```
# Replace "________" with your code
input_dim = X_train.shape[1]
hidden_dim = X_train.shape[1] * 2
output_dim = y_train.shape[1]
```

Question 9: Define tensorflow variables for weights W1, b1, W2, and b2. Then, initialize both biases b1 and b2 to be 0 and initialize W1 and W2 by picking values uniformly at random from the interval [0, 0.1]. (5 pts)

[70]:
```
# Replace "________" with your code
W1 = tf.Variable(tf.random.uniform(shape=(input_dim, hidden_dim), minval=0,␣
 ↪maxval=0.1))
b1 = tf.Variable(tf.zeros(shape=(hidden_dim)))
W2 = tf.Variable(tf.random.uniform(shape=(hidden_dim, output_dim), minval=0,␣
 ↪maxval=0.1))
b2 = tf.Variable(tf.zeros(shape=(output_dim)))
```

Question 10: Complete the classifier function below to define a two layer neural network that outputs class probabilities for each class. Here, inputs is a matrix of shape $n \times d$, where i^{th} row of the inputs matrix contains $x_i^\top$. (10 pts)

You may use tf.softmax.nn() function to compute softmax class probabilities.

[72]:
```
# Replace "________" with your code
def classifier(inputs):
  hidden = tf.nn.relu(tf.matmul(inputs, W1) + b1) # hidden layer with␣
 ↪relu-activation
  linear = tf.matmul(hidden, W2) + b2  # fully-connected linear layer
  softmax = tf.nn.softmax(linear)  # softmax layer
  return softmax
```

Question 11: Complete the function below to compute cross entropy loss given one hot encoding of targets and predictions with class probabilities for each prediction. Write the function such that autodiff will work, and note that you cannot use inbuilt tensorflow cross entropy function. (10 pts)

[80]:
```
# Replace "________" with your code
def cross_entropy(predictions, targets):
  cross_entropy = -1 * tf.reduce_sum(targets * tf.math.log(predictions), axis=1)
  mean_cross_entropy = tf.reduce_mean(cross_entropy)
  return mean_cross_entropy
```

Question 12: Complete the function below to train the neural network classifier using gradient descent. (5 pts)

[81]:
```
# Replace "________" with your code
learning_rate = 0.1
```

```
def train(inputs, targets):
    with tf.GradientTape() as tape:
        predictions = classifier(inputs)   # get predictions
        loss = cross_entropy(predictions, targets)   # compute loss
    gradients = tape.gradient(loss, [W1, b1, W2, b2])   # tape the gradients of␣
 ↪both weights and biases

    # update all weights and biases using the gradients computed above
    W1.assign_sub(learning_rate * gradients[0])
    b1.assign_sub(learning_rate * gradients[1])
    W2.assign_sub(learning_rate * gradients[2])
    b2.assign_sub(learning_rate * gradients[3])
```

Train the model using 500 GD iterations.

[82]:
```
for step in range(500):
    train(X_train, y_train)
    loss = cross_entropy(classifier(X_train), y_train)
    if (step +1)%10==0:
      print(f"Loss at step {step}: {loss:.4f}")
```

```
Loss at step 9: 2.1877
Loss at step 19: 2.0467
Loss at step 29: 1.8591
Loss at step 39: 1.6296
Loss at step 49: 1.3826
Loss at step 59: 1.1453
Loss at step 69: 0.9360
Loss at step 79: 0.7646
Loss at step 89: 0.6323
Loss at step 99: 0.5333
Loss at step 109: 0.4591
Loss at step 119: 0.4022
Loss at step 129: 0.3577
Loss at step 139: 0.3218
Loss at step 149: 0.2924
Loss at step 159: 0.2680
Loss at step 169: 0.2474
Loss at step 179: 0.2297
Loss at step 189: 0.2144
Loss at step 199: 0.2010
Loss at step 209: 0.1893
Loss at step 219: 0.1789
Loss at step 229: 0.1696
Loss at step 239: 0.1613
Loss at step 249: 0.1537
Loss at step 259: 0.1467
Loss at step 269: 0.1403
```

```
Loss at step 279: 0.1345
Loss at step 289: 0.1291
Loss at step 299: 0.1241
Loss at step 309: 0.1195
Loss at step 319: 0.1151
Loss at step 329: 0.1111
Loss at step 339: 0.1073
Loss at step 349: 0.1037
Loss at step 359: 0.1003
Loss at step 369: 0.0971
Loss at step 379: 0.0941
Loss at step 389: 0.0913
Loss at step 399: 0.0886
Loss at step 409: 0.0860
Loss at step 419: 0.0836
Loss at step 429: 0.0813
Loss at step 439: 0.0791
Loss at step 449: 0.0770
Loss at step 459: 0.0750
Loss at step 469: 0.0730
Loss at step 479: 0.0712
Loss at step 489: 0.0694
Loss at step 499: 0.0676
```

Suppose, given the vector of class probabilities, you output the label with the highest class probability as your label. As an evaluation of our model, we want to compute the number of mistakes that the model makes. For instance, if y is the true label and $\hat{y}$ is the prediction of the model, we will evaluate our model on this point with 0-1 loss

$$\mathbb{1}(\hat{y} \neq y) = \begin{cases} 1 & \hat{y} \neq y \\ 0 & \hat{y} = y \end{cases}$$

Over n points, we will compute the mean 0-1 loss,

$$\frac{1}{n} \sum_{i=1}^{n} \mathbb{1}(\hat{y}_i \neq y_i).$$

Question 13: Compute the mean 0-1 loss of your classifier on the test data. (10 pts)

[88]:
```
def get_labelled(predictions):
  max_indices = tf.math.argmax(predictions, axis=1)
  output_tensor = tf.one_hot(max_indices, depth=output_dim)
  return output_tensor

def zero_one_loss(prediction, real):
  # is prediction on axis1 all equal to real?
  row_equal = tf.reduce_all(tf.equal(prediction, real), axis=1)
  # if equal take 0, otherwise 1
  loss = 1 - tf.cast(row_equal, tf.int32)
```

```
  average_loss = tf.reduce_mean(tf.cast(loss, tf.float32))
  return average_loss

zero_one_loss_val = zero_one_loss(get_labelled(classifier(X_test)), y_test)
print(f"0-1 Loss: {zero_one_loss_val.numpy()}")
```

```
0-1 Loss: 0.031481482088565826
```