Machine Learning Using Python

Basics of Python for Deep Learning - Notebook Summary

Basic use of Python to implement machine learning models.

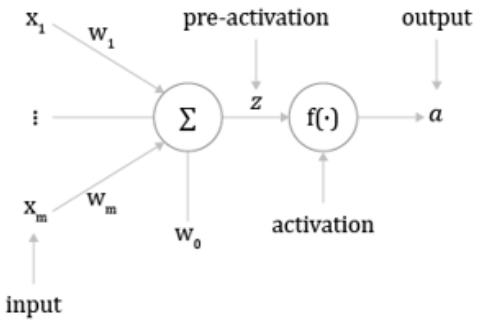
Scope: This document is a summary of the "Module 2: Basics of Python for Deep Learning" notebook from the course "Deep Learning: Mastering Neural Networks" by MIT xPRO. This summary offers a quick overview of the main aspects included in the notebook.

Table of Contents

1. Single Neuron Models

Single neuron models consist of a single neuron, or node, that sums weighted multiplications of the features of an input sample, adds a bias term, and passes that sum through an activation function.

The following formula and diagram provide the formal mathematical notation for a single neuron model:



$$z = \sum_{j=1}^{m} x_j w_j + w_0$$

$$z = xw^t + w_0$$

$$a = f(z)$$

Single neuron models can be automated using the Python programming language. These Python scripts can execute machine learning methods such as **regression** and **classification**.

Remember:

- Regression models are used to predict a real value $\hat{y}^{(i)} = f\left(x^{(i)}\right)$ for some datapoint i (i indicates which of the datapoints is being considered).
- The classification models considered here implement a binary classifier, also known as logistic regression, to predict a target variable which is binary in nature. Logistic regressions contain input-output data $(x^{(i)},y^{(i)})$ pairs that are used to output labels to one of only two classes (values between 0 and 1). Therefore, in classification models, the activation function is a sigmoid function.

2. Single Neuron Regression Model in Python

The following code script implements the necessary setup for a single neuron regression model in Python. This code uses the linear function *f(z) = z* as the activation function.

```
# First we need to import packages that will be used for visualization.
# You can ignore this for now as we will explain further in a future notebook
import numpy as np
import math, random
import matplotlib.pyplot as plt
from mpl_toolkits import mplot3d

# Define activation function
def single_neuron_regression_model(w, w_0, x):
    # Perform the dot or scalar product on the input x and the learned weights w
    z = 0
    for feature, weight in zip(x, w):
        z += feature * weight
    z += w_0  #Add the bias term

    # Apply the activation function, and return
    a = linear(z)
    return a
```

! Important: The single_neuron_regression_model function is scripted for input data x that can have any number of features. Therefore, the model can use a dataset with an x vector of any dimensions, as long as the weights w is of equal length.

Here, zip(a,b) will generate tuples of corresponding elements in a and b. For example, list(zip([1,2,3],[10,20,30])) will return [(1,10),(2,20),(3,30)].

Next, we assign values to the variables of single_neuron_regression_model(w, w_0, x) to test the model. The following example uses a two dimensional x data point.

```
# Test model output for a single 2D datapoint:
x = [1, 2]
w = [5, 3]
w_0 = -8

y = single_neuron_regression_model(w, w_0, x)
print("input", x, "=> output", y)

# Results from print:
```

2.a. Training a Regression Model – Gradient Descent

Any regression model must be trained to provide accurate predictions.

The gradient-descent method helps to determine the weights w and w_0 that minimize the cost function of the model.

A reduction in the cost function means a smaller deviation from predicted values $\hat{y}^{(i)}$ and correct values $y^{(i)}$.

The following formula gives the formal mathematical definition of the cost function J, where L is the per-data point loss:

$$J = \sum_{i}^{n} L(\hat{y}^{(i)}, y^{(i)})$$

Gradient-descent training procedure:

The model performs training loops, also known as epochs, for a specified number of iterations through the dataset.

The model examines the input and output pairs in each epoch:

1. Estimation of the loss L between the predicted values $\hat{y}^{(i)}$ and correct values $y^{(i)}$
2. Calculation of the gradient of the loss L with respect to each weight
3. Update of the weights based on the gradient and the learning rate

In other words, the gradient-descent algorithm enables the model to adapt its weights to find the lowest point in the cost function. That is, the smallest error.

The following script executes the gradient-descent training procedure with a squared error loss function.

```
# Define the training function with square error (SE) loss
def train_model_SE_loss(model_function, w, w_0,

                        input_data, output_data,
                        learning_rate, num_epochs):
  for epoch in range(num_epochs):
    total_loss = 0 #Keep track of total loss across the data set
    for x, y in zip(input_data, output_data):
      y_predicted = model_function(w, w_0, x)
      error = y_predicted - y
      total_loss += (error**2)/2

      # Update bias coefficient using gradient w.r.t w_0
      w_0 -= learning_rate * error * 1

      # Update other model coefficients using gradient w.r.t that
       coefficient
      for j, x_j in enumerate(x):
        w[j] -= learning_rate * error * x_j

    # Every few epochs, report on progress:
    report_every = max(1, num_epochs // 10)
    if epoch % report_every == 0:
```

In addition to the training of weights, the following function evaluates how well the trained regression model performs by calculating its accuracy.

```
# Define function to estimate the accuracy of the trained model
def evaluate_regression_accuracy(model_function, w, w_0, input_data, output_data):
    total_loss = 0
    n = len(input_data)
    for x, y in zip(input_data, output_data):
        y_predicted = model_function(w, w_0, x)
        error = y_predicted - y
        total_loss += (error**2)/2
    accuracy = total_loss / n
    print("Our model has mean square error of", accuracy)
```

2.b. Single Neuron Regression Example

The following example of a single neuron uses functions previously created to train the regression model with a simple set of input data.

```
# Here we have a simple 1D set of input data: a list of “x” points.
# Each x point is a list of length 1 with a corresponding “y” response.
X_1D = [[1], [-2], [3], [4.5], [0], [-4], [-1], [4], [-1]]
Y_1D = [4,  3,  6,  8,  2,  -3,  -2,  7,  2.5]

# In this example we set the initial weights to zero.
# The learning rate is relatively small.
w_0 = 0
w = [0]
learning_rate = 0.01
epochs = 11

# Calls the train_model_SE_loss function for training data.
w, w_0 = train_model_SE_loss(single_neuron_regression_model, w, w_0,
                             X_1D, Y_1D,
                             learning_rate, epochs)

# Print results with trained weights.
print("\nFinal weights:")
print(w, w_0)

# Evaulate the accuracy of final model
```

The result of executing the training of the single neuron regression example is shown below:

```
# Results from print:
epoch 0 has total loss 75.15118164194563
epoch 1 has total loss 40.109157956509094
epoch 2 has total loss 29.801523004849642
epoch 3 has total loss 25.47970121810249
epoch 4 has total loss 22.84563043777799
epoch 5 has total loss 20.86922066083035
epoch 6 has total loss 19.269845776759738
epoch 7 has total loss 17.945486824052068
epoch 8 has total loss 16.841512143299816
epoch 9 has total loss 15.91945696552539
epoch 10 has total loss 15.148892782711584
```

3. Single Neuron Classification Model in Python

In a classification model, the activation function must be changed to a sigmoid function to limit the activation output to values between 0 and 1.

The following script applies changes to the previous single_neuron_regression_model function of the regression model to define a new single_neuron_classification_model that uses a sigmoid activation function.

```
# Define sigmoid function for a classification model
def single_neuron_classification_model(w, w_0, x):
    # Perform the dot or scalar product on the input x and the learned weights w
    z = 0
    for feature, weight in zip(x, w):
        z += feature * weight
    z += w_0  #Add the bias term

    # Apply the activation function, and return
    a = sigmoid(z) #New activation function
    return a

# Sigmoid activation function; squashes a real value z to between 0 and 1
def sigmoid(z):
    non_zero_tolerance = 1e-8 # Add this to divisions to ensure we don't divide by 0
```

3.a. Training a Classification Model – Gradient Descent

Classification models use a different loss function for training weights with gradient-descent optimization. The loss function in classification problems is the negative log-likelihood (NLL) loss.

The following formula is the formal mathematical notation for the loss function L_{NLL}.

$$L_{NLL} = y\ log(\hat{y}) + (1 - y)\ log(1 - \hat{y})$$

The following script applies changes to the previous train_model_SE_loss function of the regression model to define a new function train_model_NLL_loss for the classification model.

```
# Training process using negative log likelihood (NLL) loss for classification
def train_model_NLL_loss(model_function, w, w_0,
                         input_data, output_data,
                         learning_rate, num_epochs):
    non_zero_tolerance = 1e-8 # Add this to log calculations to ensure we don't
                              # take the log of 0
    for epoch in range(num_epochs):
        total_loss = 0 # Keep track of total loss across the data set

        for x, y in zip(input_data, output_data):
            y_predicted = model_function(w, w_0, x)
            nll_loss = -(y * math.log(y_predicted + non_zero_tolerance) +
                         (1-y) * math.log(1-y_predicted + non_zero_tolerance))
            total_loss += nll_loss

        # Update bias coefficent using gradient w.r.t w_0
            w_0 -= learning_rate * (y_predicted - y)

        # Update other model coefficients using gradient w.r.t that coefficient
            for j, x_j in enumerate(x):
                w[j] -= learning_rate * (y_predicted - y) * x_j

        report_every = max(1, num_epochs // 10)
        if epoch % report_every == 0: # Every few epochs, report on progress
```

In addition to the training of weights, the following function evaluates how well the trained classification model performs by calculating its accuracy, but not customized for the classification task.

```
# We will use this function to evaluate how well our trained classifier perfoms
def evaluate_classification_accuracy(model_function, w, w_0, input_data, labels):

    # Count the number of correctly classified samples given a set of weights
    correct = 0
    n = len(input_data)
    for x, y in zip(input_data, labels):
        y_predicted = model_function(w, w_0, x)
        label_predicted = 1 if y_predicted > 0.5 else 0
        if label_predicted == y:
            correct += 1
        else:
            print("Misclassify", x, y, "with activation", y_predicted)
    accuracy = correct / n
    print("Our model predicted", correct, "out of", n,
```

3.b. Single Neuron Classification Example

The following example of a single neuron uses functions previously created to train the classification model with a new set of input data.

```
# Here we have a dataset of 2D linearly separable datapoints
input_data = [[1, 1], [1, 5], [-2, 3], [3, -4], [4.5, 2], [0, 1], [-4, -4],
          [-1, 2], [4, -7], [-1, 8]]
# And their corresponding labels
labels = [1, 1, 0, 1, 1, 0, 0, 0, 1, 0]

# Initialize weights to some small non-zero values, and train
# for more epochs.
w_0 = 0.1
w = [0.1, 0.1]
learning_rate = 0.01
epochs = 101

w, w_0 = train_model_NLL_loss(single_neuron_classification_model, w, w_0,
                    input_data, labels,
                    learning_rate, epochs)
print("\nFinal weights:")
print(w, w_0)

# Evaluate the accuracy of the final model.
```

The result of executing the training of the single neuron classification example is shown below:

```
# Results from print:
epoch 0 has total loss 6.223923692215003
epoch 10 has total loss 2.9287154378367712
epoch 20 has total loss 2.3584449039881488
epoch 30 has total loss 2.0636076120006903
epoch 40 has total loss 1.8696299804962355
epoch 50 has total loss 1.7267167165811537
epoch 60 has total loss 1.6144108218404705
epoch 70 has total loss 1.5224295144979199
epoch 80 has total loss 1.4448950765018669
epoch 90 has total loss 1.3781364940480267
epoch 100 has total loss 1.319708591557062

Final weights:
```