Regression with neural networks in Python

Mark Gluzman, Cornell University

September 29, 2019

Please see a seperate document on how to install Python packages: tensorFlow and Keras.

In a regression problem, we aim to predict some values based on available information. For example, we want to estimate an expected total discounted profit given current inventory level in the homework inventory problem. Regression usually involves independent variables (or features) x, dependent variables (or outputs) y, and unknown parameters θ that we want to "learn". A regression model relates dependent variables y to features x via a function specified by parameters θ:

$$y = f_\theta(x).$$

In this note we show how to fit regression data through a neural network model in Python. Generally, we handle a regression problem through following steps:

1. Generating dataset
2. Building a model
3. Preprocessing data
4. Learning parameters
5. Evaluating the model

Note that that steps 2-5 are general for any regression problem with the neural network model. The corresponding Python code (distributed separately) can be directly used in the homework problem.

1 Generating dataset

First, we need to get data. We will have three independent variables $x = [x_1, x_2, x_3]$ and one depended variable y.

We generate N data points of the form $\Big(X^i = [X^i_1, X^i_2, X^i_3], Y^i\Big)_{i=1}^N$, where X^i have been randomly generated and

$$Y^i = \frac{1}{3}\left(X^i_1 + 2X^i_2 + \sin(X^i_3)\right) - 5, \qquad \text{for } i = 1, ..., N.$$

We use the following Python function to get dataset with N data points:

```
def CreateDataset(N):
    # dataset is regenerated based on 3 features x = [x1, x2, x3] and
    # one output y = [x1 + 2*x2 + sim(x3)]/3 -5
    lin = np.arange(N)
    x1 = lin / 10. + np.random.uniform(-2, 2, N)
    x2 = lin / 30. + np.random.uniform(-4, 4, N)
    x3 = lin / 50. + np.random.uniform(-3, 3, N)
    y = (x1 + 2*x2 + np.sin(x3))/3. - 5.  # y = [x1 + 2*x2 + sim(x3)]/3 -5

    return np.vstack([x1, x2, x3]).T, y[:, np.newaxis]
```

Using dataset (X, Y) our goal will be to recover the relationship between features x and output y via a neural network $y = f_\theta(x)$.

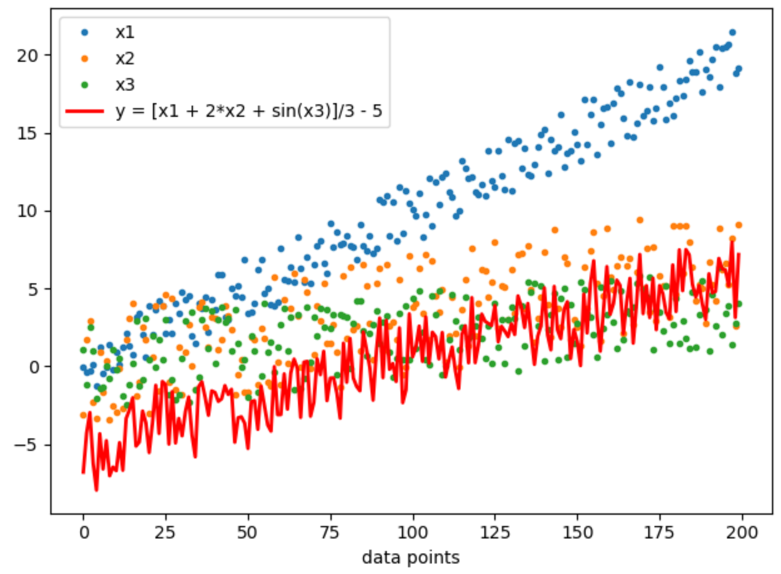

Figure 1: Example of a dataset. Red-line is y output, and the remaining dots are x features.

2 Building a model

We choose a neural network as the model for function f_θ. Building the neural network requires configuring its architecture, then compiling the model. The following discussion is intended to be an illustration of building a neural network. It is hoped that through this project, students are motivated to take a full course on this subject.

The basic building block of a neural network is the layer. A neural network model consists of an input layer, several hidden layers, and an output layer. These hidden layers allow one to extract more complicated representations from the data at the cost of increasing number of parameters that have to be learned. Our network will consist of a sequence of two hidden 'tf.keras.layers.Dense' layers, meaning neural layers are densely connected (or fully connected). See Figure 2. Both hidden layers have 5 nodes (or neurons). The

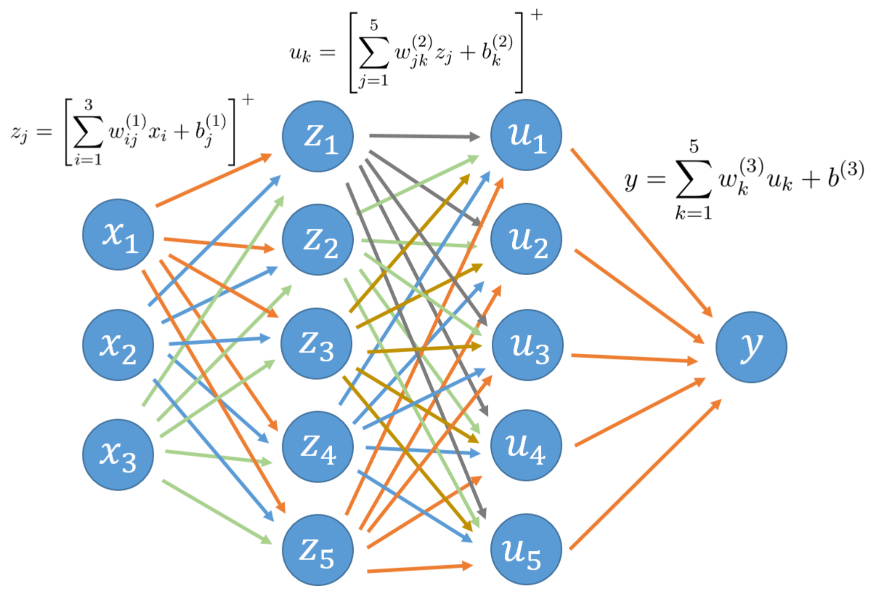
$u_k = \left[\sum_{j=1}^{5} w_{jk}^{(2)} z_j + b_k^{(2)}\right]^+$
$z_j = \left[\sum_{i=1}^{3} w_{ij}^{(1)} x_i + b_j^{(1)}\right]^+$
$y = \sum_{k=1}^{5} w_k^{(3)} u_k + b^{(3)}$
x_1
x_2
x_3
z_1
z_2
z_3
z_4
z_5
u_1
u_2
u_3
u_4
u_5
y

Figure 2: The neural network architecture.

nodes in the first hidden lay are labeled as Z_1 through Z_5 in Figure 2. The nodes in the second hidden lay are labeled as U_1 through U_5. The number of nodes in a layer is known as the *width* of the layer. The hidden layers often do not have the same width.

Each node receives input from nodes of a preceding layer. Thus each node in the first hidden layer receives inputs from all nodes in the input layer, each node in the second hidden layer receives inputs from all nodes in the first hidden layer, each (the) node in the output layer receives inputs from all nodes in the second hidden layer. Therefore, each node in a layer (excluding the input layer) and each input node in the preceding layer forms a connection, on which an associated weight (unknown parameter) is assigned. For example, $w_{jk}^{(2)}$ is the weight for connection from node k in hidden layer 2 to the input node j in hidden layer 1. Each node in a hidden layer applies the rectifier ('relu') activation function $[x]^+ = \max(0, x)$ to the weighted sum of its inputs and a bias vector – another unknown parameter. For example, the output of the jth node in the first hidden layer of our neural network model is:

$$z_j = \left[\sum_{i=1}^{3} w_{ij}^{(1)} x_i + b_j^{(1)}\right]^+, \qquad j = 1, ..., 5.$$

The output of the kth node in the second hidden layer of the neural network model is equal to:

$$u_k = \left[\sum_{j=1}^{5} w_{jk}^{(2)} z_j + b_k^{(2)} \right]^{+}, \qquad k = 1, ..., 5.$$

In the last output layer a simple linear activation function is applied:

$$y = \sum_{k=1}^{5} w_k^{(3)} u_k + b^{(3)}.$$

Let us summarize all parameters that have to be learned:

- In the first hidden layer we have 20 parameters : $w^{(1)} = \left(w_{ij}^{(1)} \right)_{i=1,j=1}^{3\times5}$ and $b^{(1)} = \left(b_j^{(1)} \right)_{j=1}^{5}$.
- In the second hidden layer – 30 parameters: $w^{(2)} = \left(w_{jk}^{(2)} \right)_{j=1,k=1}^{5\times5}$ and $b^{(2)} = \left(b_k^{(2)} \right)_{k=1}^{5}$.
- In the output layer – 6 parameters: $w^{(3)} = \left(w_k^{(3)} \right)_{k=1}^{5}$ and $b^{(3)} \in \mathbb{R}$.

As we see the neural network model f_θ we have chosen has a complicated structure and depends on 56 parameters $\theta = \left[w^{(1)}, w^{(2)}, w^{(3)}, b^{(1)}, b^{(2)}, b^{(3)} \right]$. In general, the number of hidden layers and the width of each layers are hyper-parameters that need to be decided by experiments.

After we have specified the architecture of the model, we need to compile it. We are required to set:

- loss function, the measure of how accurate the model is during training. In this example we want to minimize mean squared error: $\min_\theta \frac{1}{N} \sum_{i=1}^{N} (Y^i - f_\theta(X^i))^2$
- optimizer, the learning algorithm to find an θ in the previous optimization problem.

The following Python code is responsible for building the model:

```
class MyNeuralNetwork:
    def __init__(self, input_dimension):
        # define the keras model
        self.model = Sequential()
        self.model.add(Dense(5, input_dim=input_dimension,
                                activation='relu')) # 1st hidden layer
        self.model.add(Dense(5, activation='relu')) # 2nd hidden layer
        self.model.add(Dense(1, activation='linear')) # output layer
        # compile the keras model
        self.model.compile(loss='mean_squared_error', optimizer='adam')
```

3 Preprocessing data

We will use 80% of our dataset for learning (training). The rest data is a test set, which is not used for training. The test set allows one to check that the model has been learned successfully and we can expect the model to predict the output with high accuracy for inputs that have never been seen (not in the training set).

The following code is an example how one can divide available dataset into training and test sets.

```
    ########  divide data on training set and test set; here  80% of data
        is used for training and 20% for testing
    idx_train = np.random.choice(np.arange(len(x)), int(N * 0.8), replace=
        False)  # indexes included in training set
    idx_test = np.ones((N,), bool)
    idx_test[idx_train] = False  # indexes included in the test set
    x_train = x[idx_train]
    x_test = x[idx_test]
    y_train = y[idx_train]
    y_test = y[idx_test]
    ########################################
```

The numerical range of data can be wide and it is good practice to normalize data. Although the optimization algorithms to find θ might still converge without data normalization, without such a normalization may make the training more difficult.

The following Python class is created to compute means and variances of each feature and output:

```
class Scaler:
    # save mean and variance of x, y sets
    def __init__(self, x, y):
        self.x_mean = np.mean(x, axis=0)
        self.y_mean = np.mean(y)
        self.x_std = np.std(x, axis=0)
        self.y_std = np.std(y)


    def get_x(self):
        # return saved mean and variance of x
        return self.x_std, self.x_mean

    def get_y(self):
        # return saved mean and variance of y
        return self.y_std, self.y_mean
```

Here is an example how this Python class is used to normalize data:

```
    ####### normilize data #######################
    normalizer = Scaler(x_train, y_train)
    std_x, mean_x = normalizer.get_x()
    x_train_norm = (x_train - mean_x) / std_x
    x_test_norm = (x_test - mean_x) / std_x
    std_y, mean_y = normalizer.get_y()
    y_train_norm = (y_train - mean_y) / std_y
    ###########################################
```

4 Learning parameters

To start training, we call the 'model.fit' method in the Python code. The parameters will be learned based on the normalized data (x_train_norm, y_train_norm).

```
neural_network.model.fit(x_train_norm, y_train_norm, epochs=100,
    batch_size=8)
```

Usually the dataset is large and it can be computationally expensive to pass all the data to the optimization algorithm at once. To overcome this issue we need to divide the data into smaller pieces (batches). Before the beginning of a new epoch the entire dataset is randomly reshuffled and divided into batches, where a batch size specified by 'batch_size' in the code above. Then each batch is passed to the learning algorithm and the parameters of the neural networks are updated at the end of every such step. Therefore, after one epoch an entire dataset is passed through the neural network, but only once. In practice, many epochs are required to optimize the model. We use 100 epochs in the example.

5 Evaluating the model

Now it is time to use the test set to see how well our model can predict the output. Since the model has been trained on normalized data, we have to pass normalized data into 'neural_network.model.predict' method and scale the output of the neural network back to the original range.

```
y_from_nn_norm = neural_network.model.predict(x_test_norm)  # predict
    values for x_test states
y_from_nn = y_from_nn_norm * std_y + mean_y  # transform the results into
    original scaling
```

We compute the mean squared error between true values of the output 'y_test' from the test set and values obtained from the neural network 'y_from_nn':

```
mse = mean_squared_error(y_test, y_from_nn) # compute mean squared error
print('Mean squared error: ', mse)
```

We plot true ‘y_test’ and predicted ‘y_from_nn’ values to see visually that the accuracy is very high:

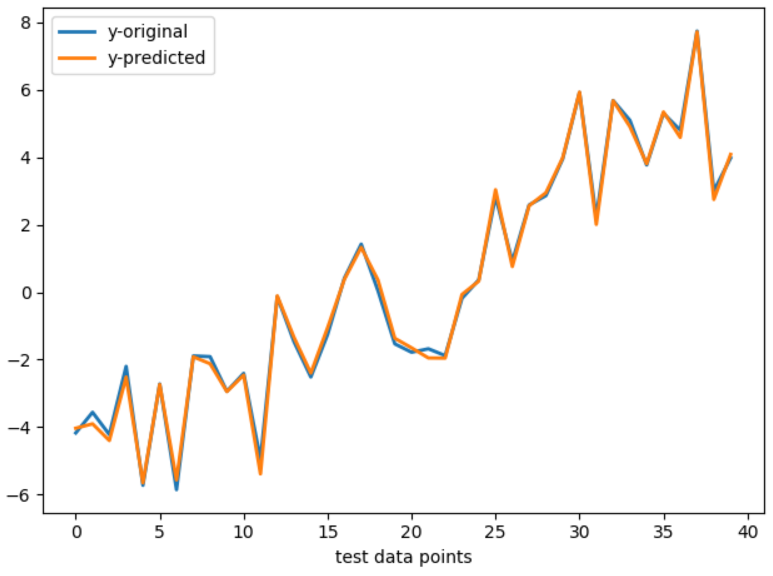

Figure 3: Comparison between original y values to predicted values.