Open in Colab

T81-558: Applications of Deep Neural Networks

Module 4: Training for Tabular Data

- Instructor: Jeff Heaton, McKelvey School of Engineering, Washington University in St. Louis
- For more information visit the class website.

Module 4 Material

- Part 4.1: Encoding a Feature Vector for Keras Deep Learning [Video] [Notebook]
- Part 4.2: Keras Multiclass Classification for Deep Neural Networks with ROC and AUC [Video] [Notebook]
- Part 4.3: Keras Regression for Deep Neural Networks with RMSE [Video] [Notebook]
- **Part 4.4: Backpropagation, Nesterov Momentum, and ADAM Neural Network Training** [Video] [Notebook]
- Part 4.5: Neural Network RMSE and Log Loss Error Calculation from Scratch [Video] [Notebook]

Google CoLab Instructions

The following code ensures that Google CoLab is running the correct version of TensorFlow.

In [1]:
```
try:
    %tensorflow_version 2.x
    COLAB = True
    print("Note: using Google CoLab")
except:
    print("Note: not using Google CoLab")
    COLAB = False
```

```
Note: not using Google CoLab
```

Part 4.4: Training Neural Networks

Backpropagation [Cite:rumelhart1986learning] is one of the most common methods for training a neural network. Rumelhart, Hinton, & Williams introduced backpropagation, and it remains popular today. Programmers frequently train deep neural networks with backpropagation because it scales really well when run on graphical processing units (GPUs). To understand this algorithm for neural networks, we must examine how to train it as well as how it processes a pattern.

Researchers have extended classic backpropagation and modified to give rise to many different training algorithms. This section will discuss the most commonly used training algorithms for neural networks. We begin with classic backpropagation and end the chapter with stochastic gradient descent (SGD).

Backpropagation is the primary means of determining a neural network's weights during training. Backpropagation works by calculating a weight change amount (v_t) for every weight(θ, theta) in the neural network. This value is subtracted from every weight by the following equation:

$$\theta_t = \theta_{t-1} - v_t$$

We repeat this process for every iteration(t). The training algorithm determines how we calculate the weight change. Classic backpropagation calculates a gradient (∇, nabla) for every weight in the neural network for the neural network's error function (J). We scale the gradient by a learning rate (η, eta).

$$v_t = \eta \nabla_{\theta_{t-1}} J(\theta_{t-1})$$

The learning rate is an important concept for backpropagation training. Setting the learning rate can be complex:

- Too low a learning rate will usually converge to a reasonable solution; however, the process will be prolonged.
- Too high of a learning rate will either fail outright or converge to a higher error than a better learning rate.

Common values for learning rate are: 0.1, 0.01, 0.001, etc.

Backpropagation is a gradient descent type, and many texts will use these two terms interchangeably. Gradient descent refers to calculating a gradient on each weight in the neural network for each training element. Because the neural network will not output the expected value for a training element, the gradient of each weight will indicate how to modify each weight to achieve the expected output. If the neural network did output exactly what was expected, the gradient for each weight would be 0, indicating that no change to the weight is necessary.

The gradient is the derivative of the error function at the weight's current value. The error function measures the distance of the neural network's output from the

expected output. We can use gradient descent, a process in which each weight's gradient value can reach even lower values of the error function.

The gradient is the partial derivative of each weight in the neural network concerning the error function. Each weight has a gradient that is the slope of the error function. Weight is a connection between two neurons. Calculating the gradient of the error function allows the training method to determine whether it should increase or decrease the weight. In turn, this determination will decrease the error of the neural network. The error is the difference between the expected output and actual output of the neural network. Many different training methods called propagation-training algorithms utilize gradients. In all of them, the sign of the gradient tells the neural network the following information:

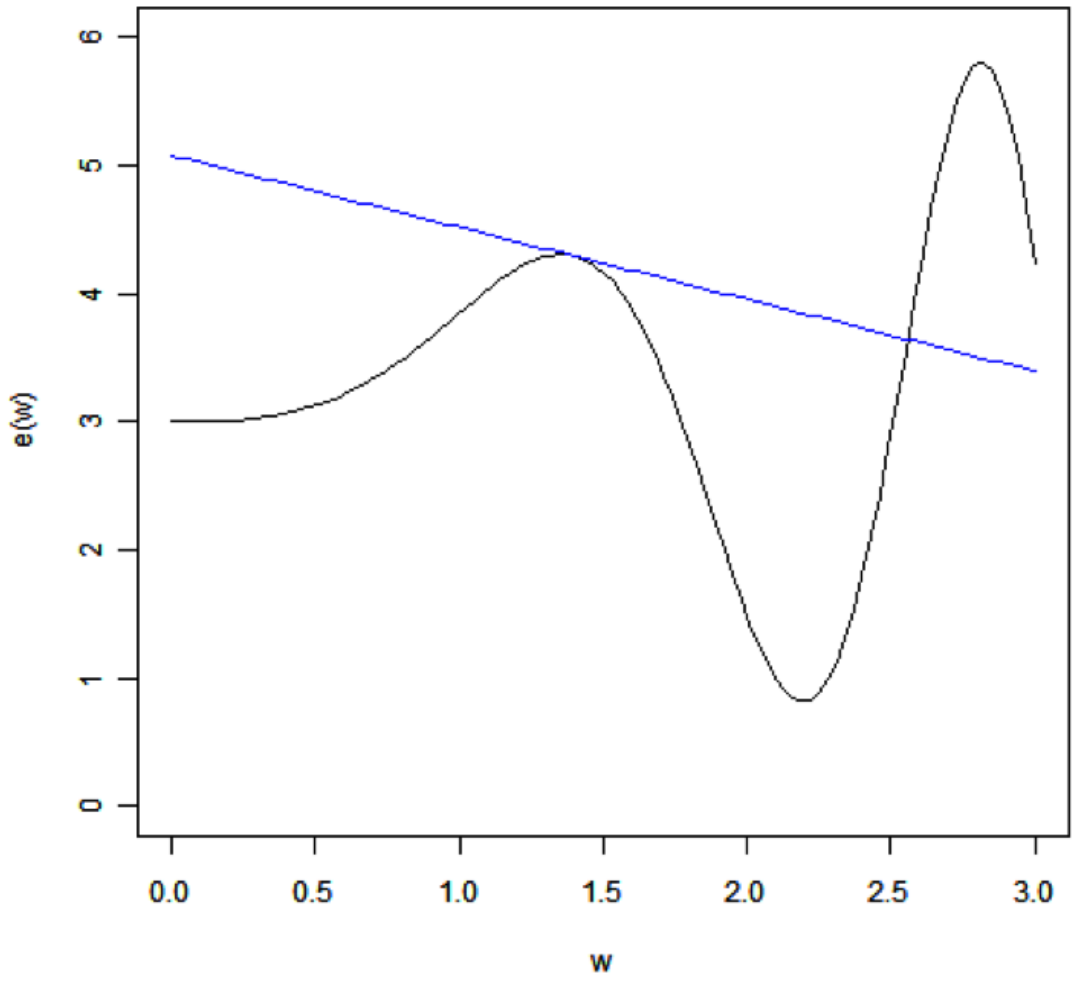
- Zero gradient - The weight does not contribute to the neural network's error.
- Negative gradient - The algorithm should increase the weight to lower error.
- Positive gradient - The algorithm should decrease the weight to lower error.

Because many algorithms depend on gradient calculation, we will begin with an analysis of this process. First of all, let's examine the gradient. Essentially, training is a search for the set of weights that will cause the neural network to have the lowest error for a training set. If we had infinite computation resources, we would try every possible combination of weights to determine the one that provided the lowest error during the training.

Because we do not have unlimited computing resources, we have to use some shortcuts to prevent the need to examine every possible weight combination. These training methods utilize clever techniques to avoid performing a brute-force search of all weight values. This type of exhaustive search would be impossible because even small networks have an infinite number of weight combinations.

Consider a chart that shows the error of a neural network for each possible weight. Figure 4.DRV is a graph that demonstrates the error for a single weight:

Figure 4.DRV: Derivative



Looking at this chart, you can easily see that the optimal weight is where the line has the lowest y-value. The problem is that we see only the error for the current value of the weight; we do not see the entire graph because that process would require an exhaustive search. However, we can determine the slope of the error curve at a particular weight. In the above chart, we see the slope of the error curve at 1.5. The straight line barely touches the error curve at 1.5 gives the slope. In this case, the slope, or gradient, is -0.5622. The negative slope indicates that an increase in the weight will lower the error. The gradient is the instantaneous slope of the error function at the specified weight. The derivative of the error curve at that point gives the gradient. This line tells us the steepness of the error function at the given weight.

Derivatives are one of the most fundamental concepts in calculus. For this book, you need to understand that a derivative provides the slope of a function at a specific point. A training technique and this slope can give you the information to

adjust the weight for a lower error. Using our working definition of the gradient, we will show how to calculate it.

Momentum Backpropagation

Momentum adds another term to the calculation of v_t:

$$v_t = \eta \nabla_{\theta_{t-1}} J(\theta_{t-1}) + \lambda v_{t-1}$$

Like the learning rate, momentum adds another training parameter that scales the effect of momentum. Momentum backpropagation has two training parameters: learning rate (η, eta) and momentum (λ, lambda). Momentum adds the scaled value of the previous weight change amount (v_{t-1}) to the current weight change amount(v_t).

This technique has the effect of adding additional force behind the direction a weight is moving. Figure 4.MTM shows how this might allow the weight to escape local minima.

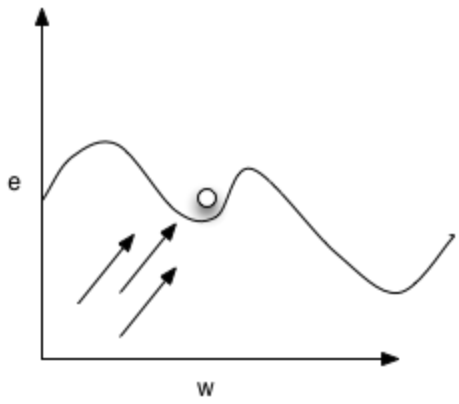


Figure 4.MTM: Momentum

A typical value for momentum is 0.9.

Batch and Online Backpropagation

How often should the weights of a neural network be updated? We can calculate gradients for a training set element. These gradients can also be summed together into batches, and the weights updated once per batch.

- **Online Training** - Update the weights based on gradients calculated from a single training set element.
- **Batch Training** - Update the weights based on the sum of the gradients over all training set elements.
- **Batch Size** - Update the weights based on the sum of some batch size of training set elements.

- **Mini-Batch Training** - The same as batch size, but with minimal batch size. Mini-batches are very popular, often in the 32-64 element range.

Because the batch size is smaller than the full training set size, it may take several batches to make it completely through the training set.

- **Step/Iteration** - The number of processed batches.
- **Epoch** - The number of times the algorithm processed the complete training set.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is currently one of the most popular neural network training algorithms. It works very similarly to Batch/Mini-Batch training, except that the batches are made up of a random set of training elements.

This technique leads to a very irregular convergence in error during training, as shown in Figure 4.SGD.

Figure 4.SGD: SGD Error

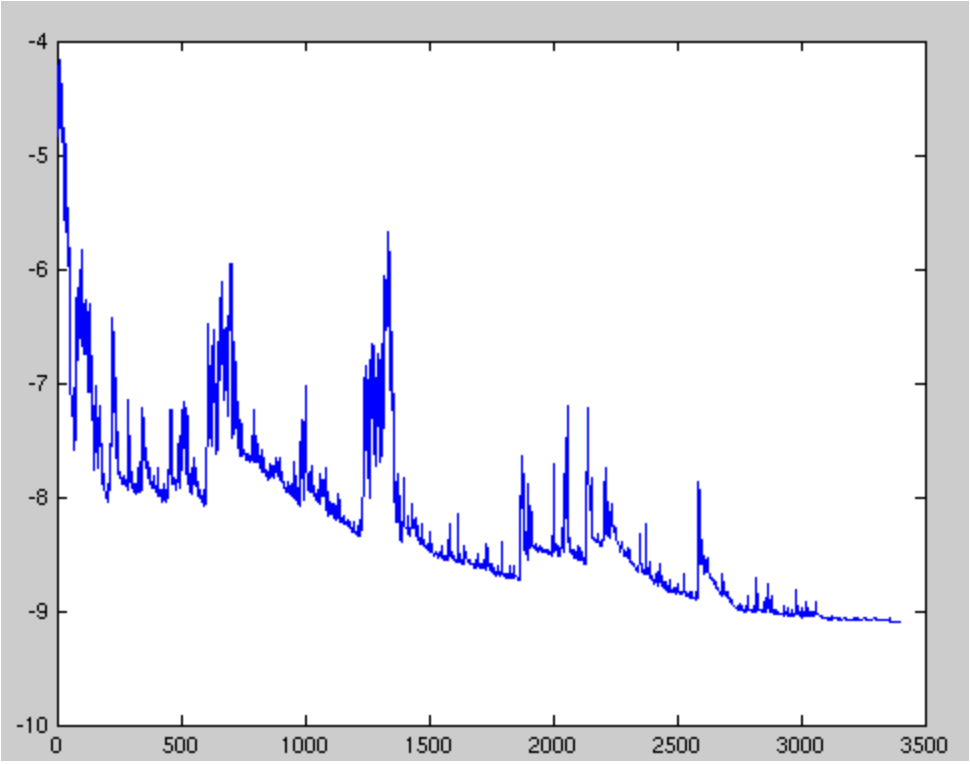


Image from Wikipedia

Because the neural network is trained on a random sample of the complete training set each time, the error does not make a smooth transition downward. However, the error usually does go down.

Advantages to SGD include:

- Computationally efficient. Each training step can be relatively fast, even with a huge training set.
- Decreases overfitting by focusing on only a portion of the training set each step.

Other Techniques

One problem with simple backpropagation training algorithms is that they are susceptible to learning rate and momentum. This technique is difficult because:

- Learning rate must be adjusted to a small enough level to train an accurate neural network.
- Momentum must be large enough to overcome local minima yet small enough not to destabilize the training.
- A single learning rate/momentum is often not good enough for the entire training process. It is often helpful to automatically decrease the learning rate as the training progresses.
- All weights share a single learning rate/momentum.

Other training techniques:

- **Resilient Propagation** - Use only the magnitude of the gradient and allow each neuron to learn at its rate. There is no need for learning rate/momentum; however, it only works in full batch mode.
- **Nesterov accelerated gradient** - Helps mitigate the risk of choosing a bad mini-batch.
- **Adagrad** - Allows an automatically decaying per-weight learning rate and momentum concept.
- **Adadelta** - Extension of Adagrad that seeks to reduce its aggressive, monotonically decreasing learning rate.
- **Non-Gradient Methods** - Non-gradient methods can *sometimes* be useful, though rarely outperform gradient-based backpropagation methods. These include: simulated annealing, genetic algorithms, particle swarm optimization, Nelder Mead, and many more.

ADAM Update

ADAM is the first training algorithm you should try. It is very effective. Kingma and Ba (2014) introduced the Adam update rule that derives its name from the adaptive moment estimates. [Cite:kingma2014adam] Adam estimates the first (mean) and second (variance) moments to determine the weight corrections. Adam begins with an exponentially decaying average of past gradients (m):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

This average accomplishes a similar goal as classic momentum update; however, its value is calculated automatically based on the current gradient (g_t). The update rule then calculates the second moment (v_t):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

The values m_t and v_t are estimates of the gradients' first moment (the mean) and the second moment (the uncentered variance). However, they will be strongly biased towards zero in the initial training cycles. The first moment's bias is corrected as follows.

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

Similarly, the second moment is also corrected:

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

These bias-corrected first and second moment estimates are applied to the ultimate Adam update rule, as follows:

$$\theta_t = \theta_{t-1} - \frac{\alpha \cdot \hat{m}_t}{\sqrt{\hat{v}_t} + \eta} \hat{m}_t$$

Adam is very tolerant to initial learning rate (\alpha) and other training parameters. Kingma and Ba (2014) propose default values of 0.9 for β_1, 0.999 for β_2, and 10-8 for η.

Methods Compared

The following image shows how each of these algorithms train. It is animated, so it is not displayed in the printed book, but can be accessed from here: https://bit.ly/3kykkbn.

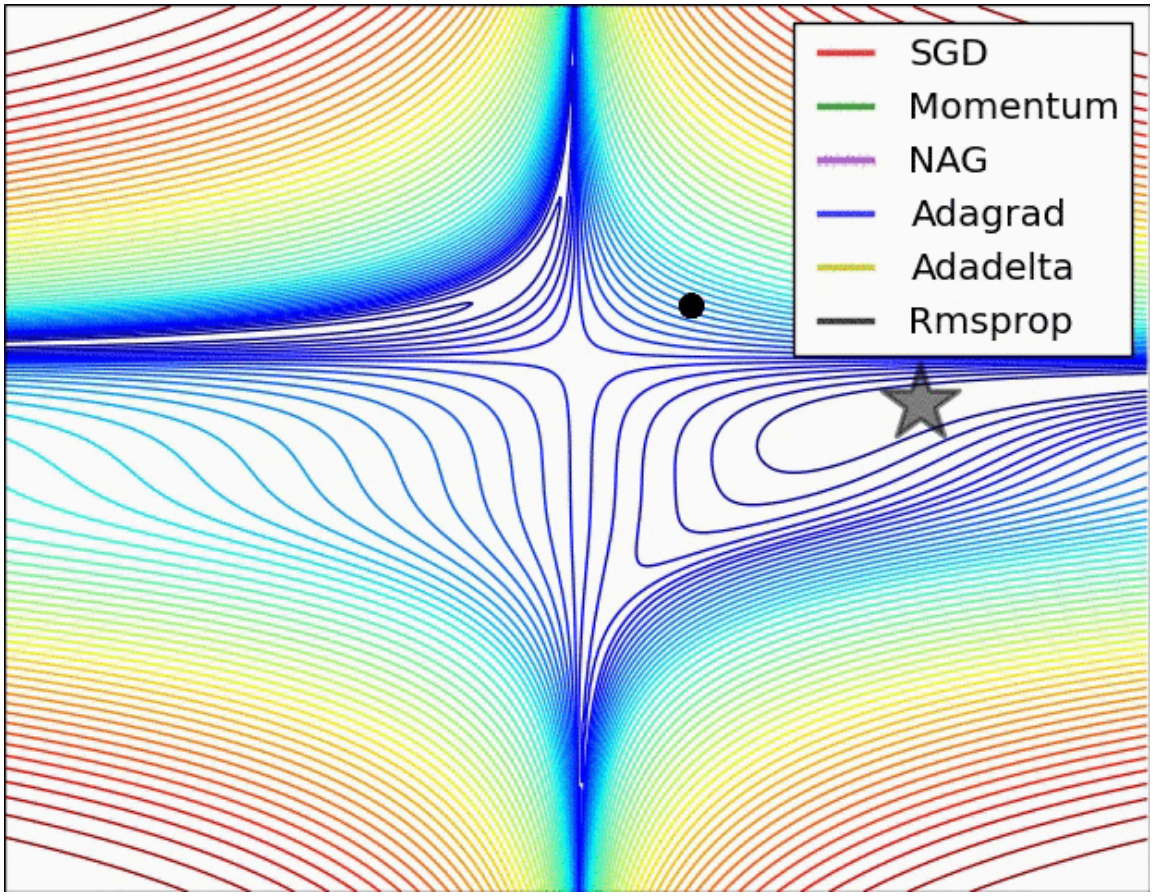


Image credits: Alec Radford

Specifying the Update Rule in Keras

TensorFlow allows the update rule to be set to one of:

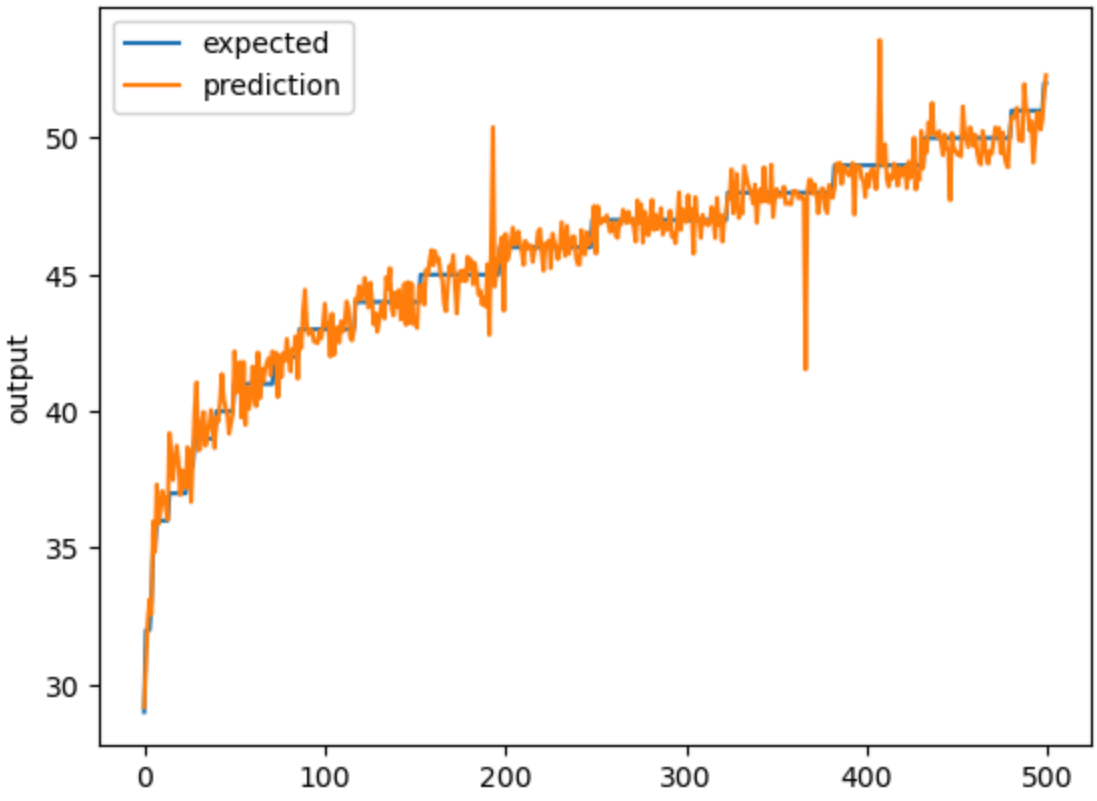
- Adagrad
- **Adam**
- Ftrl
- Momentum
- RMSProp
- **SGD**

In [2]:
```
%matplotlib inline

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.callbacks import EarlyStopping
from scipy.stats import zscore
from sklearn.model_selection import train_test_split
import pandas as pd
import matplotlib.pyplot as plt


# Regression chart.
def chart_regression(pred, y, sort=True):
```

```
    t = pd.DataFrame({'pred': pred, 'y': y.flatten()})
    if sort:
        t.sort_values(by=['y'], inplace=True)
    plt.plot(t['y'].tolist(), label='expected')
    plt.plot(t['pred'].tolist(), label='prediction')
    plt.ylabel('output')
    plt.legend()
    plt.show()

# Read the data set
df = pd.read_csv(
    "https://data.heatonresearch.com/data/t81-558/jh-simple-dataset.csv",
    na_values=['NA','?'])

# Generate dummies for job
df = pd.concat([df,pd.get_dummies(df['job'],prefix="job")],axis=1)
df.drop('job', axis=1, inplace=True)

# Generate dummies for area
df = pd.concat([df,pd.get_dummies(df['area'],prefix="area")],axis=1)
df.drop('area', axis=1, inplace=True)

# Generate dummies for product
df = pd.concat([df,pd.get_dummies(df['product'],prefix="product")],axis=1)
df.drop('product', axis=1, inplace=True)

# Missing values for income
med = df['income'].median()
df['income'] = df['income'].fillna(med)

# Standardize ranges
df['income'] = zscore(df['income'])
df['aspect'] = zscore(df['aspect'])
df['save_rate'] = zscore(df['save_rate'])
df['subscriptions'] = zscore(df['subscriptions'])

# Convert to numpy - Classification
x_columns = df.columns.drop('age').drop('id')
x = df[x_columns].values
y = df['age'].values

# Create train/test
x_train, x_test, y_train, y_test = train_test_split(
    x, y, test_size=0.25, random_state=42)

# Build the neural network
model = Sequential()
model.add(Dense(25, input_dim=x.shape[1], activation='relu')) # Hidden 1
model.add(Dense(10, activation='relu')) # Hidden 2
model.add(Dense(1)) # Output
model.compile(loss='mean_squared_error', optimizer='adam') # Modify here
monitor = EarlyStopping(monitor='val_loss', min_delta=1e-3, patience=5,
                        verbose=1, mode='auto', restore_best_weights=True)
model.fit(x_train,y_train,validation_data=(x_test,y_test),
          callbacks=[monitor],verbose=0,epochs=1000)
```

```
# Plot the chart
pred = model.predict(x_test)
chart_regression(pred.flatten(),y_test)
```

```
2024-02-14 00:07:14.691344: I tensorflow/core/platform/cpu_feature_guard.cc:
193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Lib
rary (oneDNN) to use the following CPU instructions in performance-critical
operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2024-02-14 00:07:16.219137: I tensorflow/core/platform/cpu_feature_guard.cc:
193] This TensorFlow binary is optimized with oneAPI Deep Neural Network Lib
rary (oneDNN) to use the following CPU instructions in performance-critical
operations:  SSE4.1 SSE4.2 AVX AVX2 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2024-02-14 00:07:16.221467: I tensorflow/core/common_runtime/process_util.c
c:146] Creating new thread pool with default inter op setting: 2. Tune using
inter_op_parallelism_threads for best performance.
Restoring model weights from the end of the best epoch: 155.
Epoch 160: early stopping
16/16 [==============================] - 0s 2ms/step
```



In [ ]: