Applying Regression in TensorFlow 2: Methods and Techniques

Analysis of linear, polynomial, regularized and locally weighted regression in TensorFlow 2

Hernández Pérez Erick Fernando

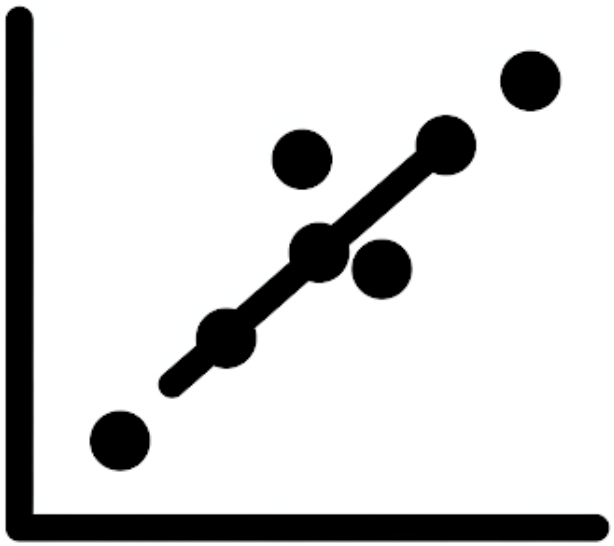

Figure 1: Oversimplified icon of Linear Regression

Data analysis and predictive modeling are fundamental to the study of patterns in a variety of disciplines, from economics and engineering to artificial intelligence. A common goal in these fields is to understand the relationship between variables and how certain factors can influence others.

In this context, fitting functions to data sets requires techniques that balance fidelity to observations with generalizability. Methods such as interpolation, smoothing, and error minimization make it possible to find functional representations that capture the essence of a phenomenon without overfitting to specific data.

But how can regression be defined? Many authors converge on a shared fundamental concept: it is a method used to predict a continuous numerical outcome based on one or more input features or variables (Fenner, 2020[1]; Harrington, 2012[2]; Mattmann, 2020[3]; Raschka et al., 2022[4]).

A brief review of the mathematical theory behind regression

First, we begin with the mathematical foundation that allows us to compute the optimal weight values by minimizing the error through differentiation. Then, we will discuss how to achieve this through training, iterating over the dataset to progressively improve our weight approximation.

Let us suppose that our data is represented in the matrix X, and our weight vector for the regression is denoted by w. Each x_i value corresponds to a y_i value, together forming the dataset upon which we wish to perform the regression.

Thus, we define our regression function as follows:

$$\hat{y}_i = x_1^T w$$

Since y_i and x_i are fixed and do not change, our error becomes a function of the weight vector w only. Therefore, we define our error as:

$$e(w) = \sum_{i=1}^{m} (y_i - x_i^T w)^2$$

To minimize the error, we seek to find a minimum in this function. We achieve this by differentiating with respect to w and setting the result equal to zero.

$$\begin{aligned} e'(w) &= \frac{d}{dw} \sum_{i=1}^{m} (y_i - x_i^T w)^2 \\ &= \sum_{i=1}^{m} \frac{d}{dw} (y_i - x_i^T w)^2 \\ &= \sum_{i=1}^{m} 2(y_i - x_i^T w)(-x_i^T) \\ &= -2 \sum_{i=1}^{m} (y_i - x_i^T w)(x_i^T) \\ &= -2X^T(y - Xw) \end{aligned}$$

And equating to zero and solving for w, we have:

$$X^T y - X^T X w = 0 \rightarrow (X^T X)^{-1} X^T y = w^*$$

Where w^* is the best weight vector for the given data.

Iterative Weight Adjustment through Gradient Descent

After establishing the foundation of regression, we now shift our focus to an iterative approach to adjust the weights by using the gradient descent algorithm. This method is particularly useful when we do not have a closed-form solution for the optimal weights, or when dealing with large datasets where solving the normal equations might be computationally expensive.

In gradient descent, the key idea is to update the weight vector iteratively by moving in the direction of the negative gradient of the error function. The weights are adjusted in small steps determined by the learning rate η, and the process is repeated over multiple epochs until the weights converge to a value that minimizes the error.

The update rule for gradient descent can be expressed as follows:

$$w_{\text{new}} = w_{\text{old}} - \eta \nabla_w e(w)$$

Where:

- w_{new} is the updated weight vector.
- w_{old} is the current weight vector.
- η is the learning rate, a small positive scalar that determines the step size.
- $\nabla_w e(w)$ is the gradient of the error function with respect to the weight vector w.

In the case of linear regression, we can express the error function as:

$$e(w) = \sum_{i=1}^{m} (y_i - x_i^T w)^2$$

The gradient of the error function with respect to w is:

$$\nabla_w e(w) = -2X^T(y - Xw)$$

Thus, the weight update rule becomes:

$$w_{\text{new}} = w_{\text{old}} + 2\eta X^T(y - Xw_{\text{old}})$$

This update rule is applied iteratively for a specified number of epochs, or until the weights converge to a minimum error.

Linear Regression

Linear Regression with Gradient Descent

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

learning_rate = 0.01
training_epochs = 100

x_train = np.linspace(-1, 1, 101)
y_train = 2 * x_train + np.random.randn(*x_train.shape) * 0.33

w = tf.Variable(0.0, name='weights')

for epoch in range(training_epochs):
    for (x, y) in zip(x_train, y_train):
        with tf.GradientTape() as tape:
            y_pred = tf.multiply(x, w)
            loss = tf.square(y - y_pred)
        grad = tape.gradient(loss, [w])
        w.assign_sub(learning_rate * grad[0])

w_val = w.numpy()

plt.scatter(x_train, y_train)
y_learned = x_train * w_val
plt.plot(x_train, y_learned, 'r')
plt.show()
```

The following code implements a linear regression using TensorFlow, applying the gradient descent algorithm to iteratively tune the model weights. First, training data is generated, where x_{train} are uniformly distributed values in the interval $[-1, 1]$, and y_{train} follows a linear relationship with added random noise to simulate a more realistic dataset. The initial weight w is set to 0, and during each training epoch, the model performs a prediction calculation y_{pred} using the current value of w and the input x.

The error or loss is calculated as the mean square error between the model predictions and the actual values of y, i.e. $\text{loss} = (y - y_{\text{pred}})^2$. The gradient of the loss with respect to w is then computed using TensorFlow's 'GradientTape' tool. With the gradient computed, the value of w is updated following the gradient descent update rule: $w_{\text{new}} = w_{\text{old}} - \eta \cdot \nabla_w \text{loss}$, where η is the learning rate.

In the resulting visualization, the original data points are displayed alongside the fitted regression line, demonstrating how the model learns from the data.

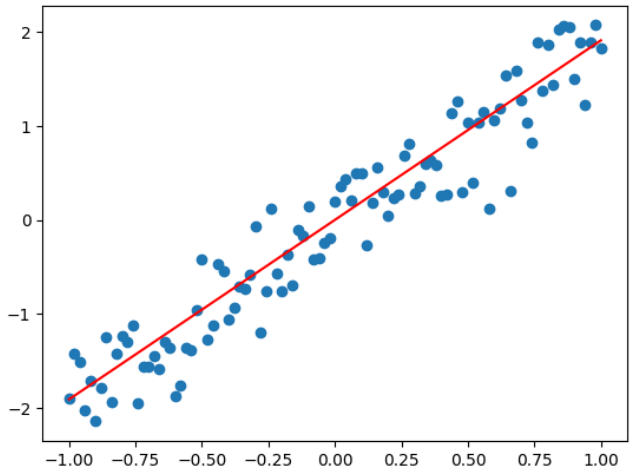

Figure 2: Linear Regression

Locally weighted linear regression

> One problem with linear regression is that it tends to underfit the data. It gives us the lowest mean-squared error for unbiased estimators. With the model underfit, we aren't getting the best predictions. There are a number of ways to reduce this mean squared error by adding some bias into our estimator. (Harrington, 2012)[2]

Locally Weighted Linear Regression (LWLR) fits a linear model to each query point using local weights based on distance. Unlike standard linear regression, which assumes a single global relationship between variables, LWLR assigns more weight to data points close to the query, allowing for more flexible, nonlinear relationships to be captured in the data.

We have three major steps:

1. For each query point x_{query}, a set of weights for the training points are computed, using a Gaussian kernel or similar function.
$$W(i,i) = \exp\left(\frac{|x^{(i)} - x|}{-2k^2}\right)$$
2. The weighted normal equation is solved:
$$\theta = (X^T W X)^{-1} X^T W y$$
3. The fitted model is used to make the prediction on x_{query}

The τ (bandwidth or smoothing) parameter controls how local the fit is:

- A small τ assigns significant weights only to very nearby points, capturing more local variation.
- A large τ behaves more like a standard linear regression, considering a more global relationship in the data.

This method is useful when the relationship between X and Y is complex and non-linear, but it has the disadvantage of being computationally expensive, as it requires a separate fit for each new prediction [2].

Locally Weighted Linear Regression

```
import tensorflow as tf
import matplotlib.pyplot as plt

# Generate synthetic data
def generate_data(num_points=100):
    X = tf.linspace(-3.0, 3.0, num_points)
```

```
    y = tf.sin(3 * X)**2 + 0.5 * X + tf.random.normal(shape=(num_points,), mean=0.0,
    ↪  stddev=0.2)
    return tf.reshape(X, (-1, 1)), tf.reshape(y, (-1, 1))

# Compute weights using a Gaussian kernel
def get_weights(X, x_query, tau):
    distances = tf.square(X - x_query)
    weights = tf.exp(-distances / (2 * tau**2))
    return tf.linalg.diag(tf.squeeze(weights))

# LWLR implementation in TensorFlow 2
def locally_weighted_regression(X, y, x_query, tau):
    m = tf.shape(X)[0]
    X_bias = tf.concat([tf.ones((m, 1)), X], axis=1)
    W = get_weights(X, x_query, tau)

    # Solve weighted normal equation:  = (X^T W X)^(-1) X^T W y
    XTWX = tf.matmul(tf.matmul(tf.transpose(X_bias), W), X_bias)
    XTWy = tf.matmul(tf.matmul(tf.transpose(X_bias), W), y)
    theta = tf.linalg.solve(XTWX, XTWy)

    # Ensure that x_query has the correct shape
    x_query_bias = tf.concat([tf.ones((1, 1)), tf.reshape(x_query, (1, 1))], axis=1)
    y_pred = tf.matmul(x_query_bias, theta)

    return tf.squeeze(y_pred)

# Generate data
X_tensor, y_tensor = generate_data(100)
tau = 0.1  # Smoothing parameter

# Predict using LWLR
x_test = tf.reshape(tf.linspace(-3.0, 3.0, 100), (-1, 1))
y_pred = tf.map_fn(lambda x_q: locally_weighted_regression(X_tensor, y_tensor, x_q, tau),
↪  x_test)

# Plot results
plt.scatter(X_tensor.numpy(), y_tensor.numpy(), label="Data", color="blue", alpha=0.5)
plt.plot(x_test.numpy(), y_pred.numpy(), label="LWLR", color="red", linewidth=2)
plt.title("Locally Weighted Linear Regression in TensorFlow 2")
plt.legend()
plt.show()
```

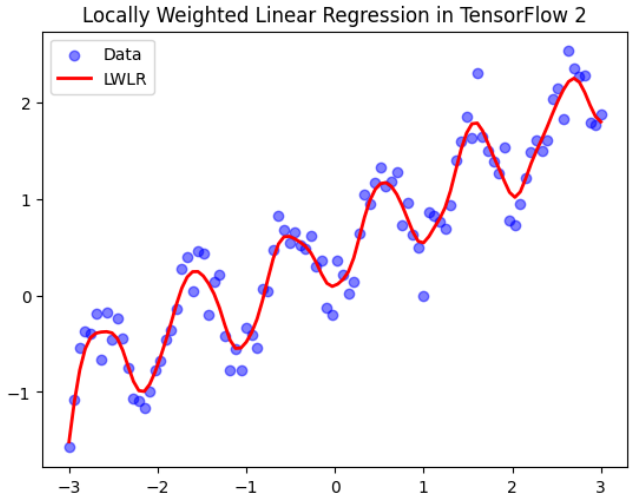

Figure 3: Locally Weighted Linear Regression

Polynomial Regression

Polynomial Regression

```
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

learning_rate = 0.01
training_epochs = 100

trX = np.linspace(-1, 1, 101)

num_coeffs = 6
trY_coeffs = [0,1,0,-5,0,4]
trY = 0
for i in range(num_coeffs):
    trY += trY_coeffs[i] * np.power(trX, i)

trY += np.random.randn(*trX.shape) * 0.2  # Noise

def model(X, w):
    terms = []
    for i in range(num_coeffs):
        term = tf.multiply(w[i], tf.pow(X, i))
        terms.append(term)
    return tf.add_n(terms)

w = tf.Variable([0.] * num_coeffs, name='parameters', dtype=tf.float64)

for epoch in range(training_epochs):
    for (x, y) in zip(trX, trY):
        x = tf.constant(x, dtype=tf.float64)
        y = tf.constant(y, dtype=tf.float64)
        with tf.GradientTape() as tape:
            y_pred = model(x, w)
```

```
            loss = tf.square(y - y_pred)
        grad = tape.gradient(loss, [w])
        w.assign_sub(learning_rate * grad[0])

w_val = w.numpy()

y_learned = model(trX, w).numpy()

plt.scatter(trX, trY, label='Data')
plt.plot(trX, y_learned, 'r', label='Learned Model')
plt.title('Polynomial Model')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.show()
```

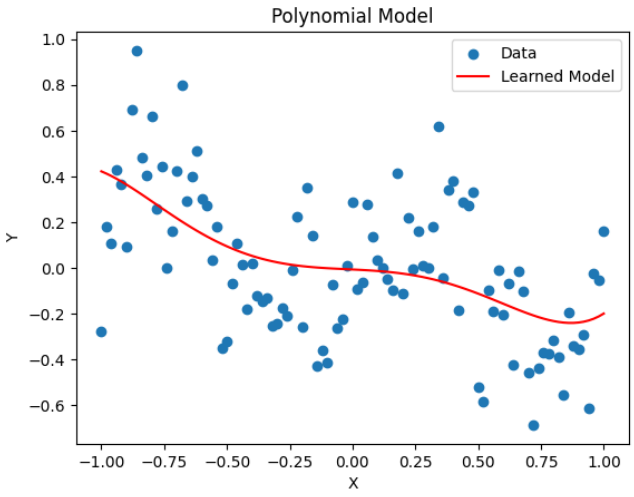

Figure 4: Polynomial Regression

Regularization in Regression

In regression models, regularization techniques help prevent overfitting by adding a penalty term to the loss function. This discourages overly complex models and ensures better generalization to unseen data. The two most common types of regularization are Ridge regression (also known as Tikhonov regularization) and Lasso regression.

Ridge Regression (L2 Regularization)

Ridge regression introduces an L_2 penalty term to the standard least squares objective function. The cost function is modified as follows:

$$J(w) = \sum_{i=1}^{m} (y_i - x_i^T w)^2 + \lambda \|w\|_2^2$$

where $\lambda \geq 0$ is the regularization parameter that controls the strength of the penalty. Larger values of λ shrink the weights, making the model less complex.

To find the optimal weights w^*, we set the derivative to zero and solve:

$$(X^T X + \lambda I) w^* = X^T y$$

where I is the identity matrix. The closed-form solution is:

$$w^* = (X^T X + \lambda I)^{-1} X^T y$$

Unlike standard least squares regression, Ridge regression ensures numerical stability, especially when $X^T X$ is nearly singular.

Lasso Regression (L1 Regularization)

Lasso regression introduces an L_1 penalty term, which can shrink some coefficients exactly to zero, leading to feature selection. The cost function is given by:

$$J(w) = \sum_{i=1}^{m} (y_i - x_i^T w)^2 + \lambda \|w\|_1$$

Unlike Ridge regression, the L_1 norm enforces sparsity, meaning that some features can be completely ignored if they are deemed unimportant.

The optimization problem for Lasso does not have a closed-form solution like Ridge regression, but it can be solved using iterative methods such as coordinate descent or gradient-based techniques.

L2 Regularization

```
from sklearn.model_selection import train_test_split
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

learning_rate = 0.01
training_epochs = 100
reg_lambda = 0.5
x_data = np.linspace(-1, 1, 100, dtype=np.float32)
num_coeffs = 9
y_data_params = np.zeros(num_coeffs, dtype=np.float32)
y_data_params[2] = 1.0
y_data = np.zeros_like(x_data, dtype=np.float32)

for i in range(num_coeffs):
    y_data += y_data_params[i] * np.power(x_data, i)
y_data += np.random.randn(*x_data.shape).astype(np.float32) * 0.3

X_train, X_test, y_train, y_test = train_test_split(
    x_data, y_data, test_size=0.3, random_state=42)

def model(X, w):
    terms = []
    for i in range(num_coeffs):
        exponent = tf.cast(i, X.dtype)
        term = tf.multiply(w[i], tf.pow(X, exponent))
        terms.append(term)
    return tf.add_n(terms)

w = tf.Variable(tf.zeros([num_coeffs], dtype=tf.float32), name="parameters")

optimizer = tf.optimizers.SGD(learning_rate=learning_rate)

X_train = tf.constant(X_train, dtype=tf.float32)
y_train = tf.constant(y_train, dtype=tf.float32)
```

```python
for epoch in range(training_epochs):
    for (x, y) in zip(X_train, y_train):
        x = tf.reshape(x, shape=[1])
        y = tf.reshape(y, shape=[1])
        with tf.GradientTape() as tape:
            y_pred = model(x, w)
            loss = tf.square(y - y_pred) + reg_lambda * tf.reduce_sum(tf.square(w)) #
            ↪  <-L2   L1 ->[+ reg_lambda * tf.reduce_sum(tf.abs(w))]
        gradients = tape.gradient(loss, [w])
        optimizer.apply_gradients(zip(gradients, [w]))

w_val = w.numpy()

y_pred_train = model(X_train, w)
loss_final = tf.reduce_mean(tf.square(y_train - y_pred_train)) + reg_lambda *
↪  tf.reduce_sum(tf.square(w))
loss_value = loss_final.numpy()
print(f"Costo final: {loss_value}")

sorted_indices = np.argsort(X_train.numpy())
X_train_sorted = X_train.numpy()[sorted_indices]
y_train_sorted = y_train.numpy()[sorted_indices]
y_learned_sorted = y_pred_train.numpy()[sorted_indices]

plt.figure(figsize=(10, 6))
plt.scatter(X_train_sorted, y_train_sorted, label='Dataset')
plt.plot(X_train_sorted, y_learned_sorted, 'r', label='Learned Model')
plt.title(f'Lambda: {reg_lambda}, Final Cost: {loss_value:.4f}')
plt.xlabel('X')
plt.ylabel('Y')
plt.legend()
plt.grid(True)

plt.show()
```

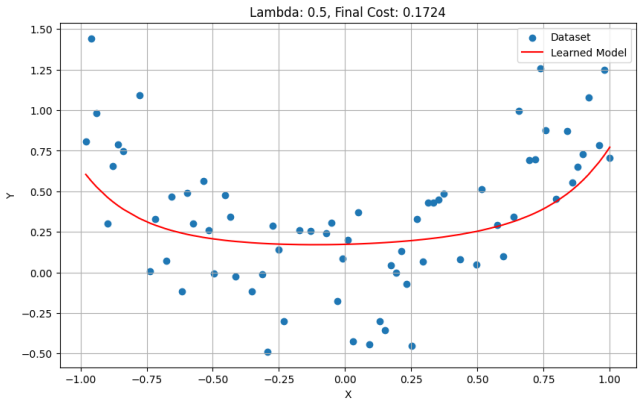

Figure 5: L2 Regularization

References

[1] Fenner, M. (2020). *Machine Learning with Python for Everyone*. Addison-Wesley.

[2] Harrington, P. (2012). *Machine Learning in Action*. Manning.

[3] Mattmann, C. (2020). *Machine Learning with TensorFlow*. Manning.

[4] Raschka, S., Liu, Y. & Mirjalili, V. (2022). *Machine Learning with PyTorch and Scikit-Learn*. Packt.