Neural Networks without Keras

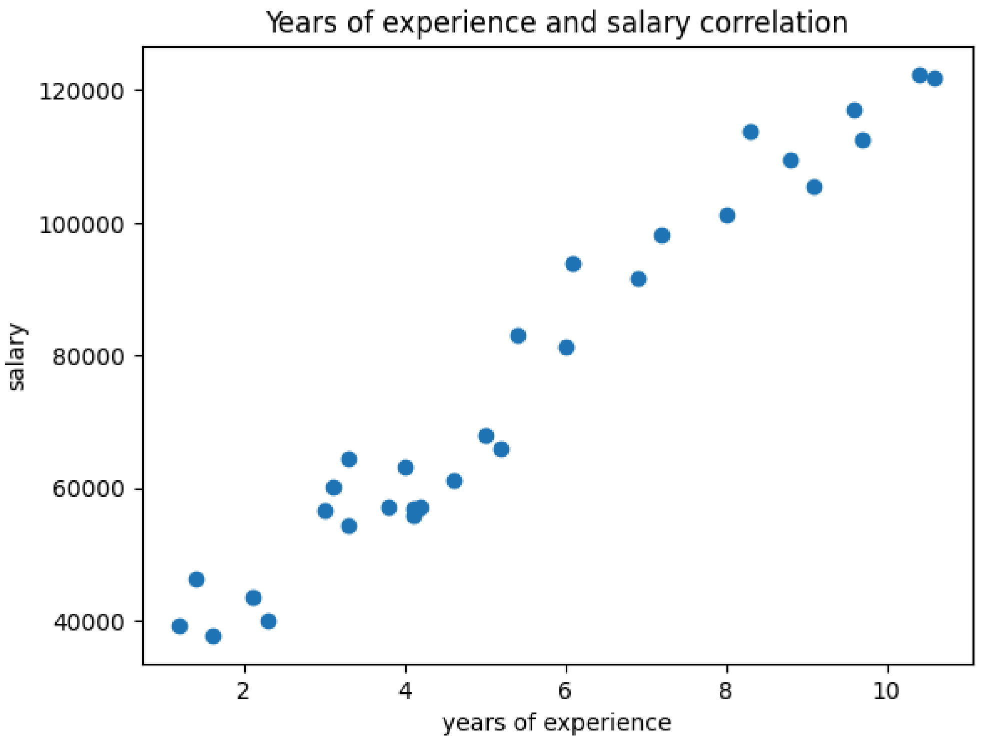
In this project, I experimented with various neural network implementations using PyTorch to get a better understanding of how neural networks learn and how predictions are made. The tasks involved building models for linear regression, non-linear regression, regression with the Boston housing dataset, and classification with the MNIST dataset. My goal was to observe how learning rate, gradient descent, and the loss function interact during training. Additionally, I wanted to compare my experiences with PyTorch to other frameworks like TensorFlow, which I've used before.

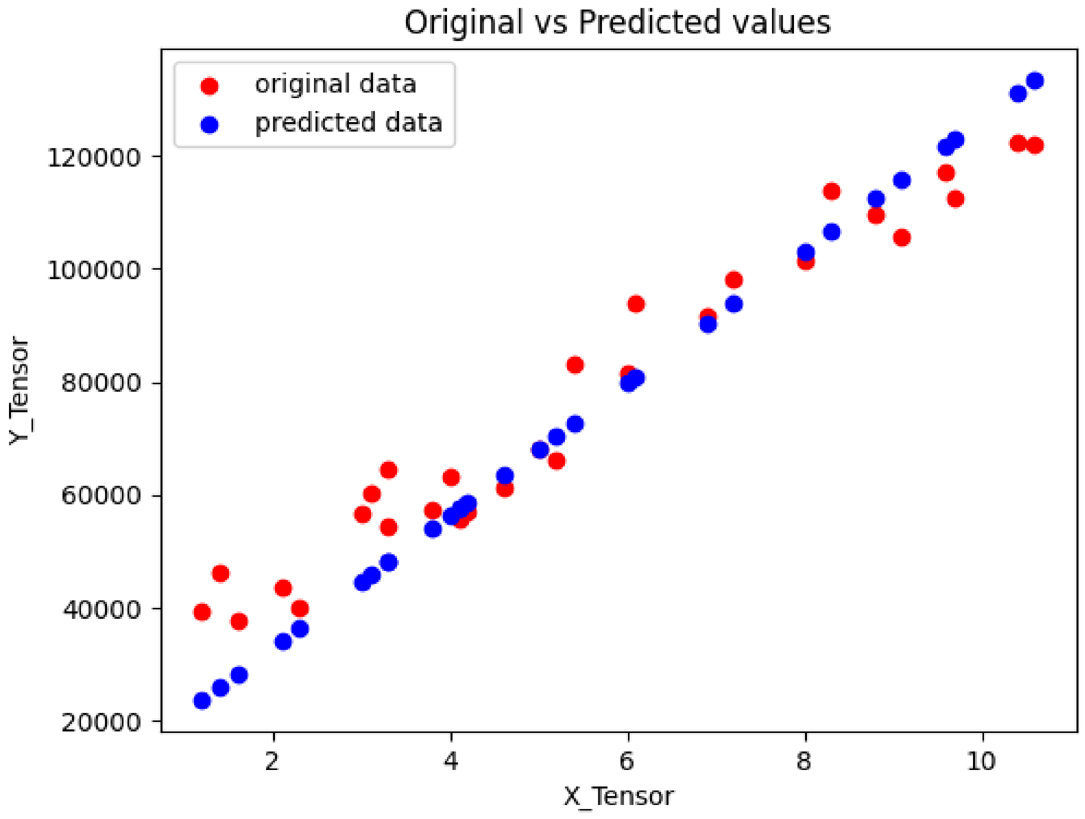
PyTorchLinearRegression.ipynb

Linear Regression with PyTorch. Your first step towards deep learning | by Arun Purakkatt | Analytics Vidhya

In this notebook, I built a simple neural network to perform linear regression. The model had only one input and one output neuron without any hidden layers. The dataset was generated with a linear relationship, making it straightforward to train and test. The network learned by adjusting weights and biases through gradient descent. The loss function used was Mean Squared Error (MSE), and I observed that the learning rate greatly influenced how fast the model converged.

Findings: Higher learning rates caused the loss to fluctuate wildly, while very low learning rates made the training process slow. The optimal learning rate achieved quick convergence within a few epochs.


Years of experience and salary correlation
salary
years of experience
120000
100000
80000
60000
40000
2
4
6
8
10


Original vs Predicted values
original data
predicted data
Y_Tensor
X_Tensor
120000
100000
80000
60000
40000
20000
2
4
6
8
10

PyTorchNonLinearRegression.ipynb

This notebook aimed to tackle a non-linear regression problem. The model was more complex, having one hidden layer with ReLU activation to handle the non-linear patterns in the data.

By using a hidden layer, the network could learn non-linear relationships by adjusting the weights and biases over multiple epochs. The optimizer used was Adam, which performed better than plain gradient descent.

Findings: The model managed to capture the non-linear patterns reasonably well, but the performance depended heavily on the network architecture. Overfitting was noticeable when too many neurons were added to the hidden layer.

PyTorchBoston.ipynb

Boston-Housing-Dataset PyTorch

This notebook focused on regression with the **Boston housing dataset**. I used a neural network with several hidden layers to predict housing prices based on 13 different features.

The model learned by minimizing the MSE loss function, and I experimented with different learning rates. The Adam optimizer showed better results compared to standard SGD.

Training was successful, but the model required careful tuning of the learning rate to avoid overfitting. Also, having too many layers resulted in higher training accuracy but poor generalization on the test dataset.

MINSTCoreAPI.ipynb

Multilayer perceptrons for digit recognition with Core APIs | TensorFlow Core

The final notebook dealt with the **MNIST dataset** to perform image classification. This network was a simple feed-forward neural network with two hidden layers, using ReLU activation for non-linearity and Softmax activation for classification.

The network processed the pixel values of images and learned to distinguish digits by minimizing the Cross-Entropy loss. The learning rate played a crucial role in how fast the network converged to a good solution.

The model achieved high accuracy (>90%) even with a simple architecture. However, training for too long resulted in overfitting, as seen by the increasing training accuracy and declining validation accuracy.

How do neural networks learn and make predictions?

Through these experiments, I learned that neural networks learn by adjusting their weights and biases based on the gradients calculated during backpropagation. The loss function measures how far off the predictions are from the true values, and gradient descent helps the model move closer to the optimal weights. In all the experiments, I saw how quickly the network could converge to a good solution if the learning rate was set correctly.

Relationship Between Learning Rate, Gradient Descent, and Loss Function

The learning rate is one of the most critical hyperparameters in training neural networks. It determines how much the model's weights are adjusted at each step of gradient descent. If the learning rate is too high, the network may overshoot optimal values, causing unstable training. If it's too low, training becomes inefficient or gets stuck in local minima.

The loss function provides feedback about the network's performance. During training, the network minimizes the loss function by adjusting weights and biases through gradient descent or other optimization techniques like Adam. I noticed that Adam performed better across most tasks due to its ability to adapt the learning rate throughout training.

Comparison with TensorFlow/Keras

Here are my observations based on my previous experience:

Ease of use: PyTorch feels simpler to work with because it's more like regular Python code. It's easier to try out new ideas or make changes to my models because the code is straightforward and not too complicated.

Speed: TensorFlow and Keras are usually faster when working with large datasets because it's designed to work efficiently with big tasks, using all available computing power.

Flexibility: PyTorch gives me more control over how the model works and makes it easier to see what's going on during training. This makes debugging (fixing mistakes) simpler.

Resources and tutorials: Both PyTorch and TensorFlow/Keras have good documentation and lots of helpful guides. However TensorFlow/Keras has more ready-to-use models and examples, which can be handy if I want to use something that's already been made.

Conclusion

Through these experiments, I got a much better idea of how neural networks actually learn and how changing different settings affects how well they work. Building models from scratch helped me see how important things like learning rates, architecture design, and training techniques are for making good predictions. I also learned how to tell when a model is overfitting or not learning well enough.

Comparing my PyTorch experiments to my previous work with TensorFlow/Keras, I realized PyTorch is much easier to use when I want to try out new ideas and fix problems. It's more hands-on and makes it simpler to see what's happening during training. Overall, I feel like I understand how neural networks work much better now.