Homework 1 – Basic Feedforward Network

Göktürk Kağan Dede

February 25, 2025

https://colab.research.google.com/drive/1JGtisGu4EOcq-CG$_jxnmaw57X4Hr4eWq?usp=sharing$

Abstract

In this homework, we aim to learn a simple linear function

$$y = 5x + 10000 + \text{Uniform}(0, 100)$$

using a feedforward neural network. Our dataset is generated randomly from this distribution, and we use different neural network architectures and hyperparameters in PyTorch. Finally, we compare the neural network performance with the `scikit-learn LinearRegression` baseline. The best neural network achieved a mean squared error (MSE) of about 800–1200 on the test set, which is in a reasonable range compared to the linear baseline whose learned slope was about 4.973 and intercept about 10051.73.

1 Introduction and Problem Statement

The main goal is to predict the function $y = 5x + 10000 + \text{Uniform}(0, 100)$ by training a basic feedforward neural network. We generate two datasets:

- **Training set**: 200 random samples
- **Test set**: 50 random samples

Each x is drawn uniformly from the range $[0, 100]$, and the seed is set to 42 for reproducibility. Hence, we have control over the data generation process. Because this task is essentially linear, it provides a clear baseline for comparing standard linear regression results with neural network approaches.

2 Method

2.1 Network Architecture

We built a feedforward neural network in PyTorch with the following components:

- **Input Layer**: One input feature (x).
- **Hidden Layers**: Configurable number of hidden layers (1 or 2), each with a configurable hidden dimension (25, 50, or 100 neurons). After each linear layer, we apply a ReLU activation.
- **Output Layer**: A single neuron providing the predicted y value.

In code, we encapsulated this in a `Net` class, which uses:

```
self.input  = nn.Sequential(nn.Linear(1, hidden_dim), nn.ReLU())
self.hiddens= nn.ModuleList(
    [nn.Sequential(nn.Linear(hidden_dim, hidden_dim), nn.ReLU())
     for _ in range(num_hidden_layers)]
)
self.output = nn.Linear(hidden_dim, 1)
```

2.2 Training Setup

We experimented with various hyperparameters:

- **Number of hidden layers**: 1 or 2
- **Hidden dimension**: 25, 50, or 100
- **Optimizers**:
  - Stochastic Gradient Descent (SGD) with and without momentum=0.9
  - Adam
- **Learning rates**: $\{10^{-2}, 10^{-3}, 10^{-4}, 10^{-5}\}$

We used `MSELoss` as the criterion and up to 200,000 epochs (or an early stop if the training loss fell below a threshold, or if we detected divergence).

2.3 Hyperparameter Tuning Approach

We tested combinations of these hyperparameters in a single script. We observed that certain combinations (e.g. large learning rate combined with high momentum) caused loss to diverge to `NaN`. When we noticed a set causing repeated divergence, we skipped further training or used a reduced learning rate. In some cases, models would train but converge more slowly.

3 Results

Below is a *sample* of our experimental results in a tabular format (not all combinations shown, for brevity). Each row shows the final training and test MSE. Note that some runs encountered `NaN` due to exploding gradients:

c

# Layers	Hidden Dim	Optimizer	LR	Momentum	Train MSE	Test MSE	Notes
1	25	SGD	0.01	0.0	NaN	NaN	Diverged quick
1	25	SGD	0.001	0.9	22254.60	27102.72	Converged, high
1	25	Adam	0.001	N/A	853.58	1019.07	Best among 1-laye
2	25	Adam	0.01	N/A	774.14	1029.44	Also converged
1	50	SGD	0.001	0.0	NaN	NaN	Diverged or stu
1	25	Adam	1e-5	N/A	2.23e7	2.35e7	Very slow LR, subo

Table 1: Sample table of key experiments. Some configurations diverged (NaN). Others converged but gave relatively large MSE.

Scikit-learn Linear Regression Baseline We compared our best neural networks to a baseline:

```
Learned slope(s):  4.973285, Learned intercept:  10051.73
```

For this dataset, the linear model is essentially capturing the slope near 5 and intercept near 10000, plus random noise. A perfect fit is impossible because of the Uniform(0,100) term.

4 Discussion

4.1 What Worked Best

Our best results came from using **Adam** with a moderate learning rate (e.g. 10^{-3}). This configuration typically converged to an MSE around 800–1200 on the test set. Though this is still larger than we might hope for purely linear data, it is near the baseline performance once we account for the noise range (0–100).

4.2 Comparison to Linear Regression

The scikit-learn LinearRegression learned a slope of 4.97 and intercept of about 10051.73. That approach perfectly fits linear data except for the unavoidable noise. Some neural network configurations (especially larger learning rates with SGD) diverged. Others converged but with higher final MSE. This is not surprising since a simple linear function is *exactly* what the linear regressor is built for.

4.3 Training Difficulties

We encountered several difficulties:

- **Divergence/NaNs:** High learning rates combined with momentum often caused exploding gradients.

- **Slow Convergence:** Very low learning rates (10^{-5}) sometimes converged but got stuck at suboptimal solutions, yielding large MSE.
- **Overfitting:** Not particularly observed here, since the function is linear and the dataset is relatively small. The main issue was not overfitting but rather poor convergence or divergence.

4.4 Preference for Layer Counts

Because the underlying data is linear, adding more layers did not significantly improve performance in most cases. A single hidden layer was generally sufficient, though Adam with two layers sometimes achieved similar performance. For purely linear data, a deeper network does not help much and can complicate training.

4.5 Summary

In conclusion, the best setting for this linear problem was typically Adam with a moderate learning rate (around 10^{-3}). This setting yields a final MSE of roughly 800–1200, close to the linear regression baseline once the noise is considered. Other settings with large learning rates or high momentum tended to diverge or plateau at higher MSE values. The results confirm that for a near-linear function, simpler optimizers or directly using a linear model can be more efficient, though the exercise helped illustrate how feedforward networks can be trained and tuned.