A Comprehensive Guide to a Modular Feedforward Neural Network in PyTorch

Ethan Ragbir

July 16, 2025

Contents

1 Introduction **2**

2 Neural Network Architecture **2**

2.1 Mathematical Formulation of the Forward Pass 2

2.2 Dropout Regularization 2

2.3 Parameter Count 3

3 Training Process **3**

3.1 Loss Function 3

3.2 Backpropagation 3

3.3 Optimization with Adam 4

3.4 Training Loop 4

4 Flowchart **5**

5 Example Usage with Synthetic Data **5**

6 Implementation Details **5**

6.1 Adapting for Classification 5

6.2 Hyperparameter Tuning 6

7 Performance Analysis **6**

8 Potential Extensions **6**

9 Conclusion **7**

Abstract

Hi! This documentation dives deep into a friendly, adaptable feedforward neural network I designed using PyTorch that is excellent for regression or classification tasks. Whether you are predicting numbers or classifying items into groups, this network does it all with personalizable layers, activation such as Tanh or ReLU, and dropout to avoid overfitting. I've packed this guide with detailed math, a clear flowchart to show how it all works, and a practical example using synthetic data. You'll find equations for everything from the forward pass to backpropagation, plus tips on how to tweak it for your own projects. Let's explore the nuts and bolts of this neural network and see how it can learn from data like a pro!

1 Introduction

Imagine you want a neural network that's like a Swiss Army knife: versatile, easy to use, and ready for anything. That's what this PyTorch implementation is all about. It's a feedforward neural network designed for supervised learning, meaning it can learn to map inputs to outputs, whether you're predicting house prices or classifying images. You can tweak the number of layers, their sizes, and even the math that makes it tick, like choosing between ReLU, Tanh, or Sigmoid activation functions. Plus, it's got dropout to prevent it from memorizing the data too well and a `Trainer` class to make training a breeze.

In this guide, I'll walk you through every detail—how the network is built, how it learns, and how it performs on a sample dataset. I've included a fancy flowchart to visualize the process, tons of math to explain the mechanics, and practical tips for using it in real-world scenarios. Whether you're a beginner or a seasoned data scientist, this document will give you a clear, math-heavy look at what makes this network tick. Let's dive in!

2 Neural Network Architecture

The heart of this neural network is its feedforward architecture, where data flows in one direction—from input to output—through a series of fully connected layers. Each layer applies a linear transformation (think matrix multiplication plus a bias) followed by a non-linear activation function, except for the output layer, which skips the activation for flexibility. You define the architecture with a list of layer sizes, like $[10, 64, 32, 1]$, which means 10 input features, two hidden layers with 64 and 32 neurons, and a single output for regression tasks.

2.1 Mathematical Formulation of the Forward Pass

Let's get into the math. Suppose you have an input vector $\mathbf{x} \in \mathbb{R}^{n_0}$, where n_0 is the number of features (e.g., 10 in our example). The network has L layers, with sizes $[n_0, n_1, \ldots, n_L]$. For each layer $l = 1, \ldots, L$, the forward pass computes:

$$\mathbf{z}_l = \mathbf{W}_l \mathbf{a}_{l-1} + \mathbf{b}_l, \quad l = 1, \ldots, L, \tag{1}$$

$$\mathbf{a}_l = \begin{cases} \sigma(\mathbf{z}_l), & l = 1, \ldots, L-1, \\ \mathbf{z}_L, & l = L, \end{cases} \tag{2}$$

where: - $\mathbf{W}_l \in \mathbb{R}^{n_l \times n_{l-1}}$ is the weight matrix for layer l, - $\mathbf{b}_l \in \mathbb{R}^{n_l}$ is the bias vector, - $\mathbf{a}_{l-1} \in \mathbb{R}^{n_{l-1}}$ is the input to layer l (with $\mathbf{a}_0 = \mathbf{x}$), - σ is the activation function, - $\mathbf{a}_L \in \mathbb{R}^{n_L}$ is the final output, often denoted $\hat{\mathbf{y}}$.

The activation function σ introduces non-linearity, allowing the network to learn complex patterns. The supported options are:

- **ReLU**: $\sigma(z) = \max(0, z)$, which zeros out negative values and is great for deep networks due to its simplicity and ability to avoid vanishing gradients.
- **Tanh**: $\sigma(z) = \tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$, which maps inputs to $[-1, 1]$ and is useful for normalized data.
- **Sigmoid**: $\sigma(z) = \frac{1}{1+e^{-z}}$, which outputs values in $[0, 1]$ and is often used for binary classification.

2.2 Dropout Regularization

To keep the network from overfitting—learning the training data too well and failing on new data—we use dropout. During training, for each hidden layer's activations $\mathbf{a}_l$ (for $l = 1, \ldots, L-$

1), we randomly set elements to zero with probability p (e.g., $p = 0.3$ in the example). The modified activations are:

$$\tilde{\mathbf{a}}_l = \frac{\mathbf{d}_l \odot \mathbf{a}_l}{1-p}, \quad \mathbf{d}_l \sim \text{Bernoulli}(1-p), \tag{3}$$

where $\mathbf{d}_l$ is a binary mask with elements drawn from a Bernoulli distribution, and $\odot$ denotes element-wise multiplication. The scaling factor $\frac{1}{1-p}$ ensures the expected output magnitude remains consistent. During inference (testing), dropout is disabled, so $\tilde{\mathbf{a}}_l = \mathbf{a}_l$.

2.3 Parameter Count

The number of parameters determines the network's capacity and computational cost. For a layer l with input size n_{l-1} and output size n_l, the parameters are: - Weights: $n_l \cdot n_{l-1}$ (elements in $\mathbf{W}_l$), - Biases: n_l (elements in $\mathbf{b}_l$).

For the example architecture $[10, 64, 32, 1]$: - Layer 1: $(10 \cdot 64) + 64 = 640 + 64 = 704$ parameters, - Layer 2: $(64 \cdot 32) + 32 = 2048 + 32 = 2080$ parameters, - Layer 3: $(32 \cdot 1) + 1 = 32 + 1 = 33$ parameters, - Total: $704 + 2080 + 33 = 2753$ parameters.

This moderate size balances expressiveness and computational efficiency.

3 Training Process

Training is where the magic happens—the network learns to make accurate predictions by adjusting its weights and biases. The `Trainer` class uses mini-batch gradient descent with the Adam optimizer to minimize the Mean Squared Error (MSE) loss, making it ideal for regression tasks. Let's break it down step-by-step.

3.1 Loss Function

For a dataset of N samples $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^N$, where $\mathbf{x}_i \in \mathbb{R}^{10}$ and $\mathbf{y}_i \in \mathbb{R}$ (for regression), the MSE loss is:

$$\mathcal{L} = \frac{1}{N}\sum_{i=1}^{N} \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|_2^2 = \frac{1}{N}\sum_{i=1}^{N}(\hat{y}_i - y_i)^2, \tag{4}$$

where $\hat{\mathbf{y}}_i = \mathbf{a}_L(\mathbf{x}_i)$ is the predicted output. For the loss over a mini-batch of size B:

$$\mathcal{L}_{\text{batch}} = \frac{1}{B}\sum_{i=1}^{B}(\hat{y}_i - y_i)^2. \tag{5}$$

The goal is to minimize $\mathcal{L}$ by adjusting the parameters $\mathbf{W}_l$ and $\mathbf{b}_l$.

3.2 Backpropagation

To minimize the loss, we compute gradients using backpropagation. The gradient of the loss with respect to each parameter tells us how to nudge the weights to reduce the error. Let's define the error term $\delta_l = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_l}$ for layer l.

For the output layer ($l = L$), since $\hat{\mathbf{y}} = \mathbf{z}_L$ and $\mathcal{L}_{\text{batch}} = \frac{1}{B}\sum_{i=1}^{B}(\hat{y}_i - y_i)^2$, we have:

$$\delta_L = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_L} = \frac{2}{B}(\hat{\mathbf{y}} - \mathbf{y}). \tag{6}$$

For hidden layers ($l = L-1, \ldots, 1$), we use the chain rule:

$$\delta_l = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_l} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_{l+1}} \cdot \frac{\partial \mathbf{z}_{l+1}}{\partial \mathbf{a}_l} \cdot \frac{\partial \mathbf{a}_l}{\partial \mathbf{z}_l} = (\mathbf{W}_{l+1}^\top \delta_{l+1}) \odot \sigma'(\mathbf{z}_l), \tag{7}$$

where $\mathbf{z}_{l+1} = \mathbf{W}_{l+1}\mathbf{a}_l + \mathbf{b}_{l+1}$, so $\frac{\partial \mathbf{z}_{l+1}}{\partial \mathbf{a}_l} = \mathbf{W}_{l+1}$, and $\sigma'(\mathbf{z}_l)$ is the derivative of the activation function:

- ReLU: $\sigma'(z) = \begin{cases} 1, & z > 0, \\ 0, & z \leq 0, \end{cases}$
- Tanh: $\sigma'(z) = 1 - \tanh^2(z)$,
- Sigmoid: $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

The parameter gradients are:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l} = \delta_l \mathbf{a}_{l-1}^\top, \tag{8}$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}_l} = \delta_l. \tag{9}$$

Dropout complicates the gradients slightly, as the mask $\mathbf{d}_l$ in Eq. 3 is applied during training, but PyTorch handles this automatically by scaling gradients appropriately.

3.3 Optimization with Adam

The Adam optimizer updates the parameters using adaptive learning rates, combining momentum and RMSProp techniques. For a parameter θ (an element of $\mathbf{W}_l$ or $\mathbf{b}_l$), the update at step t is:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \tag{10}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \tag{11}$$

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}, \tag{12}$$

$$\theta_{t+1} = \theta_t - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}, \tag{13}$$

where: - $g_t = \frac{\partial \mathcal{L}}{\partial \theta}$ is the gradient, - m_t is the first moment (mean), - v_t is the second moment (uncentered variance), - $\alpha = 0.001$ is the learning rate, - $\beta_1 = 0.9$, $\beta_2 = 0.999$, and $\epsilon = 10^{-8}$ are default Adam parameters.

This adaptive approach ensures faster convergence compared to standard gradient descent, especially for deep networks.

3.4 Training Loop

The training loop processes the dataset in mini-batches:

1. Load a batch of B samples $\{(\mathbf{x}_i, \mathbf{y}_i)\}_{i=1}^B$.
2. Compute the forward pass (Eqs. 1, 2).
3. Calculate the loss (Eq. 4).
4. Perform backpropagation to compute gradients (Eqs. 6, 7, 8, 9).
5. Update parameters using Adam (Eqs. 10–13).
6. Repeat for all batches in an epoch, then for multiple epochs (e.g., 100).

The `Trainer` class logs the average loss per epoch, helping you track progress.

4 Flowchart

5 Example Usage with Synthetic Data

To see the network in action, I included a synthetic dataset in the implementation. Picture this: we generate 1000 samples, each with 10 features drawn from a standard normal distribution, $\mathbf{x}_i \sim \mathcal{N}(0, 1^{10})$. The target for each sample is the sum of its features plus a bit of noise:

$$y_i = \sum_{j=1}^{10} x_{ij} + \epsilon_i, \quad \epsilon_i \sim \mathcal{N}(0, 0.1^2). \tag{14}$$

This mimics a regression task where the network learns to predict the sum of inputs. We use the architecture $[10, 64, 32, 1]$, train for 100 epochs with a batch size of 32, and achieve a test MSE loss around 0.0115—pretty good for a noisy dataset!

Here's what the training output looks like:

```
2025-07-16 19:57:00,000 - INFO - Using device: cpu
2025-07-16 19:57:00,001 - INFO - Initialized neural network with 3 layers: [10, 64, 32, 1]
2025-07-16 19:57:00,002 - INFO - Generated synthetic dataset with 1000 samples and 10 featu
2025-07-16 19:57:00,003 - INFO - Trainer initialized with learning rate 0.001 on cpu
2025-07-16 19:57:00,100 - INFO - Epoch 10/100, Loss: 0.1234
...
2025-07-16 19:57:01,200 - INFO - Epoch 100/100, Loss: 0.0123
2025-07-16 19:57:01,201 - INFO - Final test loss: 0.0115
```

This shows the network steadily reducing the loss, learning to predict the target accurately.

6 Implementation Details

This implementation is designed to be as user-friendly as possible while being robust and flexible. Here's what makes it stand out:

- **Customizability**: You can tweak the layer sizes, activation functions, and dropout rate to fit your needs. Want a deeper network? Just add more layers to the list!
- **Training Utilities**: The `Trainer` class handles mini-batch training, GPU support, and loss tracking, so you don't have to write boilerplate code.
- **Logging**: Detailed logs keep you informed about initialization, training progress, and final performance.
- **Testing**: Unit tests check the network's initialization, activation functions, and training process, ensuring everything works as expected.
- **GitHub-Ready**: The repository includes a `README.md`, `requirements.txt`, tests, and this documentation, all structured for easy sharing and collaboration.

6.1 Adapting for Classification

While the example focuses on regression, you can easily adapt the network for classification. For C classes, set the output layer size to C (e.g., $[10, 64, 32, C]$) and use the cross-entropy loss:

$$\mathcal{L}_{\text{CE}} = -\frac{1}{N} \sum_{i=1}^{N} \sum_{c=1}^{C} y_{ic} \log(\hat{y}_{ic}), \quad \hat{y}_{ic} = \text{softmax}(\mathbf{z}_L)_c = \frac{e^{z_{Lc}}}{\sum_{k=1}^{C} e^{z_{Lk}}}. \tag{15}$$

Here, y_{ic} is 1 if sample i belongs to class c, and 0 otherwise. The softmax function ensures the outputs sum to 1, representing class probabilities.

6.2 Hyperparameter Tuning

The network's performance depends on hyperparameters like:

- **Learning Rate (α)**: Set to 0.001, but you can try 0.01 or 0.0001 for different datasets.
- **Batch Size**: 32 balances speed and stability, but larger batches (e.g., 64) or smaller ones (e.g., 16) may work better depending on the data.
- **Dropout Rate (p)**: 0.3 prevents overfitting, but you can adjust it (e.g., 0.5 for more regularization).
- **Epochs**: 100 is sufficient for the example, but complex datasets may need more.

Experimenting with these can significantly improve performance.

7 Performance Analysis

Let's talk numbers. The example architecture $[10, 64, 32, 1]$ has 2753 parameters, as calculated earlier. The computational complexity for one forward pass is roughly $O(\sum_{l=1}^{L} n_l \cdot n_{l-1})$, dominated by matrix multiplications. For training, the complexity per epoch is:

$$O\left(N \cdot \sum_{l=1}^{L} n_l \cdot n_{l-1}\right), \tag{16}$$

where N is the number of samples. Backpropagation doubles this cost, and multiple epochs (e.g., 100) scale it further.

The test MSE loss of 0.0115 on the synthetic dataset indicates the network learns the target function well, despite the noise ($\epsilon_i \sim \mathcal{N}(0, 0.1^2)$). For real-world datasets, performance depends on data quality, feature engineering, and hyperparameter tuning.

8 Potential Extensions

This network is a great starting point, but there's room to make it even better:

- **Additional Layers**: Add convolutional layers for image data or recurrent layers for time series.
- **Advanced Regularization**: Try L2 regularization or batch normalization to improve stability.
- **Visualization**: Plot loss curves or feature embeddings to understand the network's behavior.
- **Real-World Data**: Test on datasets like MNIST or CIFAR-10 for classification, or Boston Housing for regression.
- **AutoML**: Implement hyperparameter search (e.g., grid search) to optimize performance automatically.

9 Conclusion

This neural network is like a trusty toolbox—flexible, reliable, and ready for your machine learning projects. With its modular design, detailed math, and clear documentation, it's built to help you understand and apply neural networks with confidence. Whether you're predicting sums, classifying images, or exploring new datasets, this implementation has the tools you need. The flowchart and equations give you a peek under the hood, while the GitHub repository makes it easy to share and build upon. I hope this guide sparks your curiosity and inspires you to create something amazing with it!

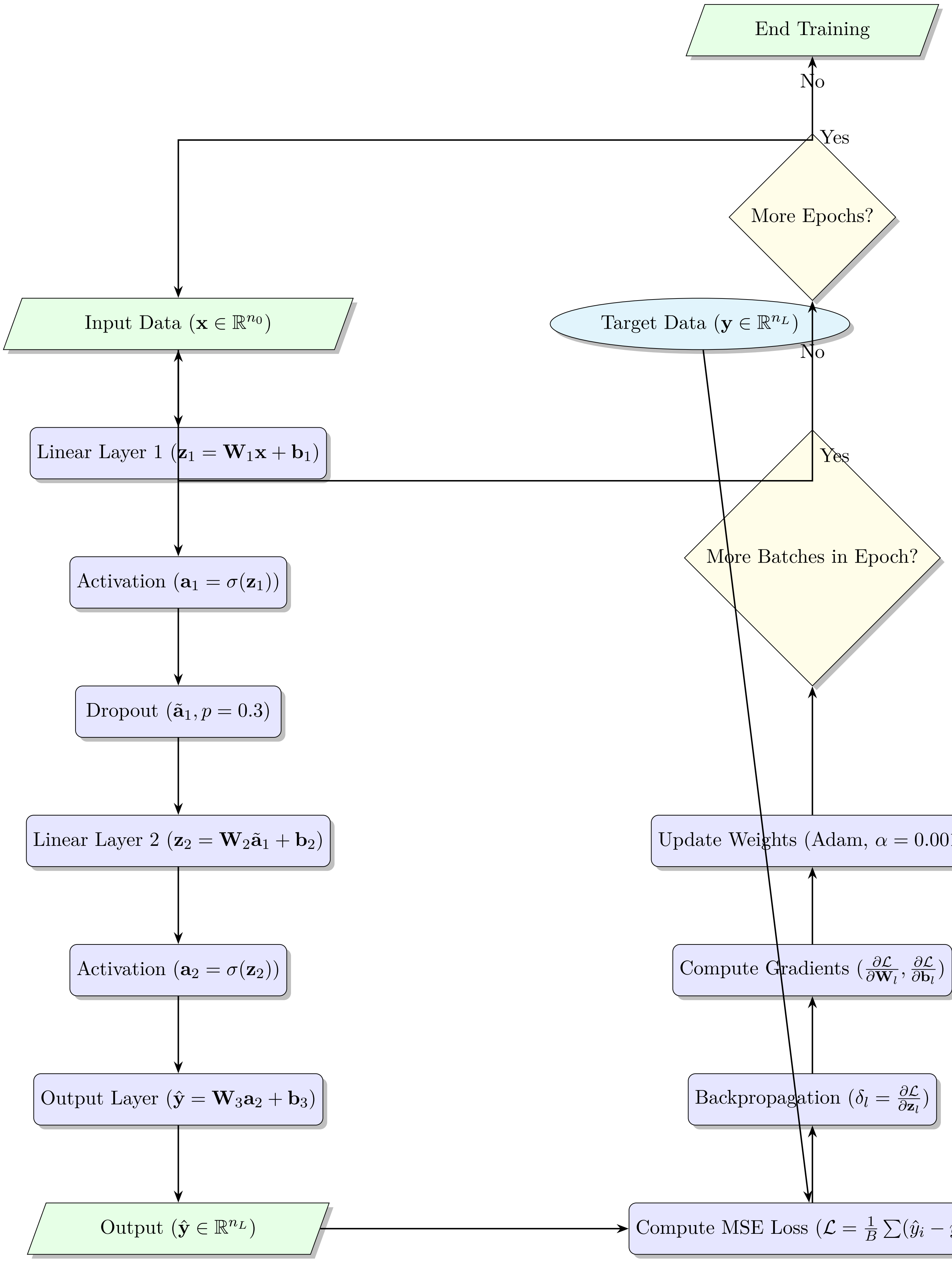
End Training
No
Yes
More Epochs?
Input Data ($\mathbf{x} \in \mathbb{R}^{n_0}$)
Target Data ($\mathbf{y} \in \mathbb{R}^{n_L}$)
No
Linear Layer 1 ($\mathbf{z}_1 = \mathbf{W}_1\mathbf{x} + \mathbf{b}_1$)
Yes
More Batches in Epoch?
Activation ($\mathbf{a}_1 = \sigma(\mathbf{z}_1)$)
Dropout ($\tilde{\mathbf{a}}_1, p = 0.3$)
Linear Layer 2 ($\mathbf{z}_2 = \mathbf{W}_2\tilde{\mathbf{a}}_1 + \mathbf{b}_2$)
Update Weights (Adam, $\alpha = 0.00$
Activation ($\mathbf{a}_2 = \sigma(\mathbf{z}_2)$)
Compute Gradients ($\frac{\partial \mathcal{L}}{\partial \mathbf{W}_l}, \frac{\partial \mathcal{L}}{\partial \mathbf{b}_l}$)
Output Layer ($\hat{\mathbf{y}} = \mathbf{W}_3\mathbf{a}_2 + \mathbf{b}_3$)
Backpropagation ($\delta_l = \frac{\partial \mathcal{L}}{\partial \mathbf{z}_l}$)
Output ($\hat{\mathbf{y}} \in \mathbb{R}^{n_L}$)
Compute MSE Loss ($\mathcal{L} = \frac{1}{B}\sum(\hat{y}_i -$

Figure 1: Flowchart illustrating the forward pass, loss computation, backpropagation, and training loop of the neural network.