

Nabla ∇ : A Simple Neural Network Library

Adrian Rodriguez (AXR190042)
CS 6375 002
Code: <https://github.com/axr2718/nabla>

1 Introduction

Modern deep learning frameworks are what powers current machine learning (ML). Libraries such as PyTorch, TensorFlow, and JAX are what make it possible to engineer, deploy, and research current artificial intelligence (AI) systems. Although these libraries are very powerful, they abstract away the true mechanisms underlying all of ML. Users do not need to know anything about how the models, layers, activations, and optimizers are implemented or work under the hood, and do not need to know the mathematics underlying such processes.

In this report, I introduce Nabla ∇ , a simple neural network library that implements a Multi-layer Perceptron (MLP) from scratch, including the forward and backward passes, activations, the calculus of backpropagation, different tasks, losses, and optimizers. The library is validated through a series of tests that show the implementation is correct. Finally, I create many different MLPs with varying activations and optimizers to experiment with, train them on different datasets on different tasks, such as classification or regression, and show the results.

Nabla ∇ provides a simple and clean implementation from scratch for neural networks that is easy to plug-and-play and understand. The library is also compatible with Numpy, Scikit Learn, and PyTorch, allowing us to use their datasets and tensors/arrays. I show that this library can teach anyone how the main deep learning libraries work.

2 Implementation

At its core, a neural network implementation is a computation graph, where each layer, activation, and the loss can be viewed as the nodes of a computation graph. The forward pass simply follows from the input nodes to the loss and the backwards pass computes the gradients going in reverse of this graph using the chain rule from calculus, and then the parameters of each layer are updated with these gradients. See Figure 1 for an example of a computation graph describing the forward and backward pass.

For the implementation, I create a base class called `Module` that acts as the nodes of the computation graph. Each layer, activation, and loss can reference this `Module` and implement their own logic. Since I am only implemented an MLP, we only require the Linear layer as the main layer. For the layer, I implemented a few different activation functions to test which are better. For different tasks, there is also different losses. To experiment further, I also implemented multiple optimizers and the Dropout layer.

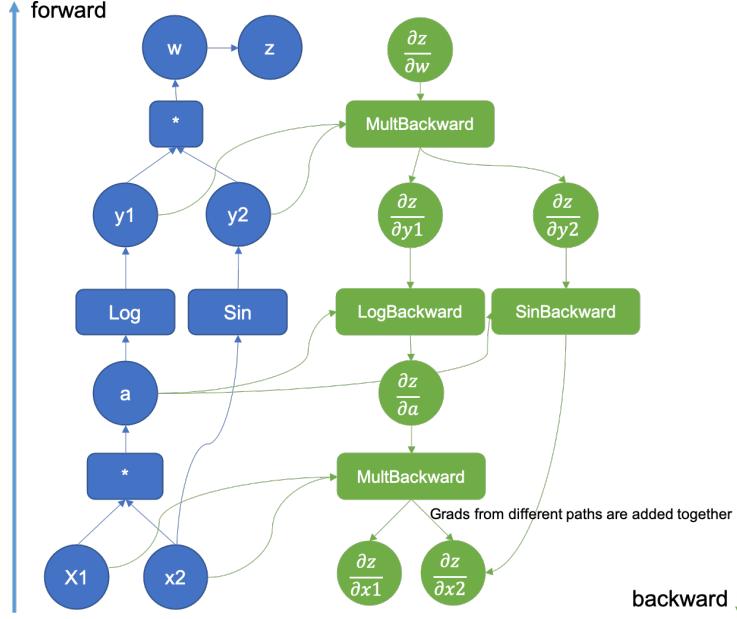


Figure 1: Computation graph

All of the code was implemented using Python and the NumPy library.

2.1 Linear Layer

The main component of an MLP is the Linear layer, which is a layer that takes as input the output of the previous layer, computes a weighted sum with the layer's weight and bias, then adds a non-linearity. For Linear layers, $l = 1, \dots, L$, the forward pass of each Linear layer is as follows

$$a^{[l]} = g^{[l]}(w^{[l]T} a^{[l-1]} + b^{[l]}) = g^{[l]}(z^{[l]}) \quad (1)$$

where $a^{[l]}$, $g^{[l]}$, $w^{[l]}$, $b^{[l]}$ are the output, activation, weight, and bias of layer l , respectively, and $a^{[l-1]}$ is the input of the layer, which is the output of layer $l - 1$. We let $a^{[0]} = x$. Due to the computation graph formulation of the library, the activation also has its own forward and backward pass, but this will be covered in section 2.2.

The weights and bias of the Linear layer are what will be optimized during training, so we need to formulate the backward pass for each of these layers. The backward pass for the Linear layers computes the gradient of the loss with respect to the layer's weight, bias, and also for the activation to send downstream. We assume that we have the gradient of the loss with respect the layer's activation function, $\frac{\partial L}{\partial a^{[l]}} \frac{\partial a^{[l]}}{\partial z^{[l]}} = \frac{\partial L}{\partial a^{[l]}} \odot g'^{[l]}$. Using the chain rule, we derive the gradients for $w^{[l]}$ and $b^{[l]}$, and for $a^{[l-1]}$ to send downstream.

$$\frac{\partial L}{\partial w^{[l]}} = \frac{\partial L}{\partial a^{[l]}} \frac{\partial a^{[l]}}{\partial z^{[l]}} \frac{\partial z^{[l]}}{\partial w^{[l]}} = \left(\frac{\partial L}{\partial a^{[l]}} \odot g'^{[l]} \right) a^{[l-1]T} \quad (2)$$

$$\frac{\partial L}{\partial b^{[l]}} = \frac{\partial L}{\partial a^{[l]}} \odot g'^{[l]} \quad (3)$$

$$\frac{\partial L}{\partial a^{[l-1]}} = w^{[l]T} \left(\frac{\partial L}{\partial a^{[l]}} \odot g'^{[l]} \right) \quad (4)$$

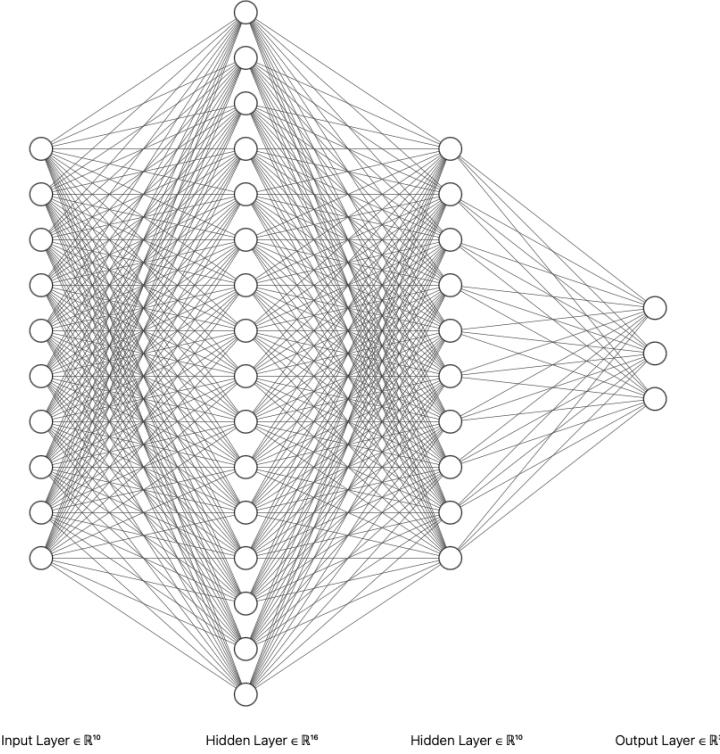


Figure 2: An MLP with 3 Linear layers, each with its own activation.

In code, for the Linear layer, we simply take $\frac{\partial L}{\partial a^{[l]}} \frac{\partial a^{[l]}}{\partial z^{[l]}}$ that is given by the activation function's backward pass. It is not computed by the Linear layer itself, since the Linear layer represents $z^{[l]}$ and the activation function, $a^{[l]}$, has its own forward and backward pass.

An MLP with 3 Linear layers, each with an activation, is shown in Figure 2. Note that we do not count the input layer as a layer in our framework. The layers that are not the input or output layers are known as hidden layers.

2.2 Activation Functions

I implemented the Rectified Linear Unit (ReLU), Sigmoid Linear Unit (SiLU), Sigmoid, and Tanh activation functions to experiment with each of them and see which work better for a regression and classification task. All of these activations are widely used and popular for neural networks.

2.2.1 ReLU

ReLU is used often due to how easy it is to compute and has worked great empirically. The function is defined as

$$\text{ReLU}(z^{[l]}) = \max(0, z^{[l]}) \quad (5)$$

This function has values 0 on the negative side of the graph and it grows linearly on the positive side. The derivative of this function, even though it's not defined at 0, is

$$\text{ReLU}'(z^{[l]}) \begin{cases} 1, & z^{[l]} > 0, \\ 0, & z^{[l]} \leq 0. \end{cases} \quad (6)$$

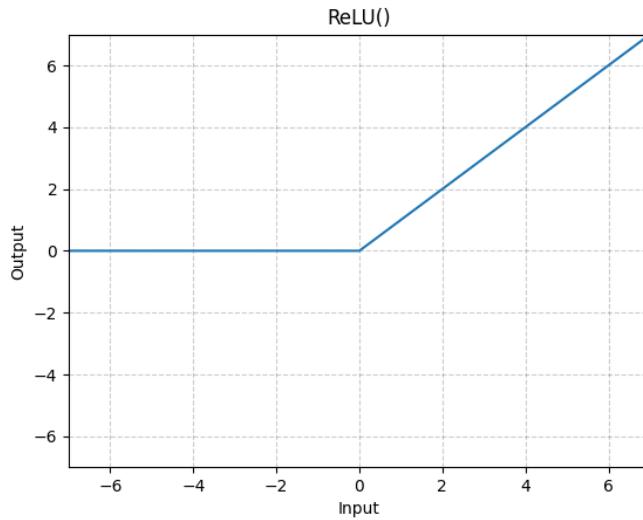


Figure 3: The ReLU function

2.2.2 SiLU

The SiLU function is typically a replacement to ReLU because it is a smoothed version and differentiable everywhere, and it avoids the dead neuron problem, where negative values have 0 gradient and are thus not updated during optimization. The function is defined as

$$\text{SiLU}(z^{[l]}) = z^{[l]} \sigma(z^{[l]}) \quad (7)$$

where σ is the Sigmoid function. The derivative of SiLU is

$$\text{SiLU}'(z^{[l]}) = \sigma(z^{[l]}) + \sigma(z^{[l]})z^{[l]}(1 - \sigma(z^{[l]})) \quad (8)$$

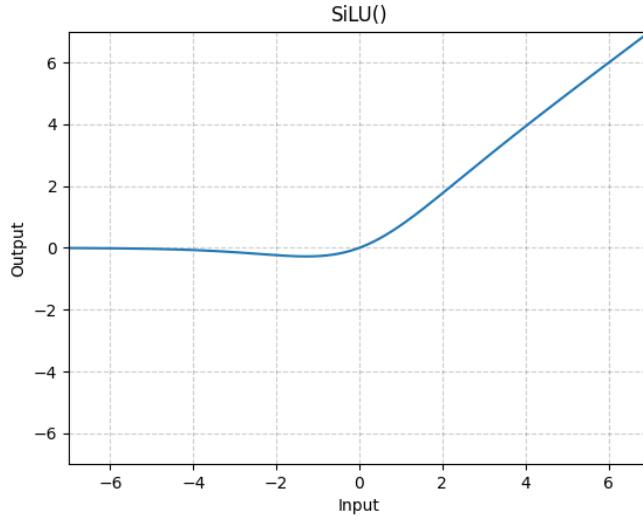


Figure 4: The SiLU function

2.2.3 Sigmoid

The Sigmoid function has been used historically since it squishes the output values to be between 0 and 1, which was inspired by how neurons in the brain work in an on or off fashion. The function is defined as

$$\sigma(z^{[l]}) = \frac{1}{1 + e^{-z^{[l]}}} \quad (9)$$

and its derivative is

$$\sigma'(z^{[l]}) = \sigma(z^{[l]})(1 - \sigma(z^{[l]})) \quad (10)$$

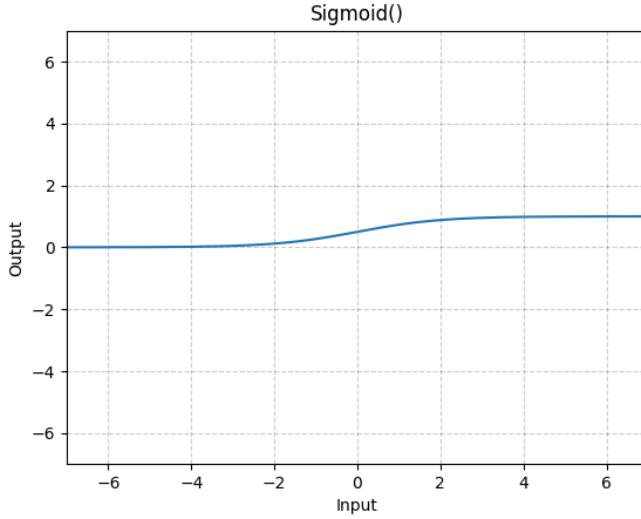


Figure 5: The Sigmoid function

2.2.4 Tanh

Just like SiLU is a replacement for ReLU in some cases, Tanh is also a replacement for Sigmoid for similar reasons. When Sigmoid is a large negative or positive number, the gradient becomes very small, and leads to slow learning. The tanh has better gradient flow and can avoid this issue and is defined as

$$\text{Tanh}(z^{[l]}) = \frac{e^{z^{[l]}} - e^{-z^{[l]}}}{e^{z^{[l]}} + e^{-z^{[l]}}}. \quad (11)$$

and its derivative is

$$\text{Tanh}'(z^{[l]}) = 1 - \tanh^2(z^{[l]}). \quad (12)$$

The reason for choosing these specific activation functions, other than the fact that they are widely used and popular, is to compare them and see if my results match what current literature says. I will compare ReLU vs. SiLU and Sigmoid vs. Tanh, along with comparing the other combinations, to see if learning is better with SiLU and Tanh in comparison to what was used previously.

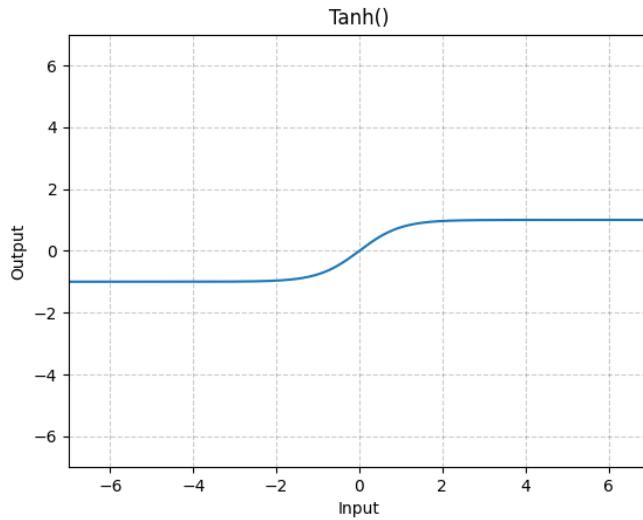


Figure 6: The Tanh function

Although I defined the forward pass for the activation functions, and derived their derivatives, I did not provide the backward pass. We let the derivative of the activation functions be denoted as $\frac{\partial a^{[l]}}{\partial z^{[l]}} = g'(z^{[l]})$, where g is any of the activation functions. When we get $\frac{\partial L}{\partial a^{[l]}}$ from the backward pass, we compute $\frac{\partial L}{\partial a^{[l]}} \frac{\partial a^{[l]}}{\partial z^{[l]}}$ with the current activation function and then pass this down to the Linear layer. This term appears in equations (2), (3), and (4), which are used by the Linear layer to compute its own gradients.

2.3 Loss Functions

In order to evaluate our tasks, regression and classification, we need a loss function for each. For regression, I chose the Mean Square Error (MSE) loss and for classification, I chose the Cross Entropy loss. We let N represent the number of training samples in the entire training dataset to simplify the math. However, this is rarely done in practice, especially if the datasets are large, and mini batches are used instead.

For both tasks, we take the loss of every sample and average the loss before doing the backward pass. Another simplification I make is that even though the loss function is dependent on $a^{[L]}$, w , b , and y , we only choose to show the w and b arguments since that is what we will use the evaluate the loss.

2.3.1 Mean Square Error Loss

Given the output $a^{[L](i)}$ and true label $y^{(i)}$, $y^{(i)} \in \mathbb{R}$, the MSE loss is given by

$$L(w, b) = \frac{1}{N} \sum_{i=1}^N (a^{[L](i)} - y^{(i)})^2 \quad (13)$$

and the derivative with respect to $a^{[L](i)}$ is

$$\frac{\partial L}{\partial a^{[L](i)}} = \frac{2}{N} (a^{[L](i)} - y^{(i)}) \quad (14)$$

2.3.2 Cross Entropy Loss

For multiclass classification problems, the network outputs a vector $a^{[L](i)} \in \mathbb{R}^C$ for each sample i , where C is the number of classes. We convert these logits into probabilities using the softmax function

$$\text{softmax}(a^{[L](i)})_j = \frac{e^{a_j^{[L](i)}}}{\sum_{k=1}^C e^{a_k^{[L](i)}}} \quad (15)$$

Let $p_j^{(i)}$ denote the resulting probability for class j . Given the true class label $y^{(i)} \in \{1, \dots, C\}$, the multiclass cross entropy loss is

$$L(w, b) = -\frac{1}{N} \sum_{i=1}^N \log p_{y^{(i)}}^{(i)} \quad (16)$$

The gradient of the loss with respect to the logits $a^{[L](i)}$ is

$$\frac{\partial L}{\partial a^{[L](i)}} = \frac{1}{N} \left(p^{(i)} - \text{one_hot}(y^{(i)}) \right) \quad (17)$$

where $\text{one_hot}(y^{(i)})$ is the one-hot encoded vector for the correct class.

The most important part of the loss function, other than that it is the last step of the forward pass, is that it kicks off the entire chain reaction of the backward pass. Given $\frac{\partial L}{\partial a^{[L]}}$, it passes this to the previous activation, which computes $\frac{\partial L}{\partial a^{[L]}} \frac{\partial a^{[L]}}{\partial z^{[L]}}$, which passes it to the previous Linear layer to compute the gradients for w , b , and $a^{[l-1]}$. This process continues backwards until all parameters have been updated.

2.4 Optimizers

I implement two of the most widely used optimizers in the field: Stochastic Gradient Descent (SGD) and Adaptive Moment Estimation (Adam). We compare these in the experiments and see which works better for our models and datasets.

2.4.1 Stochastic Gradient Descent

SGD is a very simple variant of Gradient Descent (GD) where instead of updating the parameters using the entire dataset, we use randomly drawn mini batches of the dataset, which is where the term "stochastic" comes from. However, the optimizer is not just limited to mini batches and can use the entire batch as well. The update rule for SGD is

$$w_{t+1} = w_t - \alpha \frac{\partial L}{\partial w_t} \quad (18)$$

where $\alpha \in \mathbb{R}$ is the step size, also called the learning rate, t is the iteration, and $\frac{\partial L}{\partial w_t}$ is the term we get through backpropagation. One of the downsides of using SGD with mini batches is that it can make finding the minima of the loss function be a noisy and unstable process. For this library, we ignore the momentum term and just use it in Adam.

2.4.2 Adam

Adam is an optimizer based on Gradient Descent with momentum that maintains exponentially decaying moving averages of both the gradients and the squared gradients. These moving averages smooth the update direction, allowing the optimizer to make faster progress near the minima of the loss function. Adam combines the advantages of two earlier methods: Momentum and RMSProp. Given the gradient at iteration t , denoted g_t , Adam updates the first and second moment estimates as

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad (19)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \quad (20)$$

where m_t estimates the mean of the gradients and v_t estimates the variance. To correct for initialization bias, Adam computes the bias-corrected estimates

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t} \quad (21)$$

The parameter update rule for Adam is

$$w_{t+1} = w_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon} \quad (22)$$

where α is the learning rate, β_1 and β_2 control the decay rates of the moving averages, and ϵ is a small constant added for numerical stability. In practice, Adam tends to converge faster than vanilla Stochastic Gradient Descent and requires less hyperparameter tuning. See figure 7.

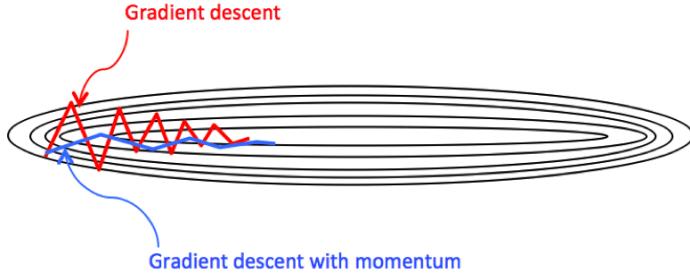


Figure 7: Gradient Descent with mini batches vs Gradient Descent with momentum

3 Experiments

To experiment with the library, we choose the tasks of regression and multi-class classification. For regression I use the California Housing dataset, which contains 20,640 samples, each with 8 features, and the labels are a single real number. For classification I use the CIFAR-10 dataset, which contains 10 classes, 50,000 training images, and 10,000 testing images. As mentioned in the introduction, we can simply download and use these datasets from Scikit Learn and PyTorch, respectively, since we can easily transform these datasets into numpy arrays, which is what ∇ is built on. This showcases the compatibility of ∇ with current AI/ML libraries in Python.

The main experiment will be comparing ReLU vs. SiLU in the regression task and Sigmoid vs. Tanh in the classification task. Historically, ReLU and SiLU are used for regression or

continuous-valued functions because it keeps the numbers continuous. SiLU seems to have been taking off in previous years and replaced ReLU. We experiment on our MLPs to see if SiLU does indeed beat ReLU. The same thing happened with Sigmoid and Tanh. Both of these activation functions function in an "on" or "off" state and squish the values between either 0 and 1, or -1 and 1, respectively. Sigmoid has an issue that big values lead to small gradients, and thus lead to slow learning, which is why Tanh replaced it. We experiment to see whether Sigmoid or Tanh is better.

On top of experimenting with these activation functions, we also experiment with Dropout layers. Dropout layers are widely used and randomly drop units in the Linear layer during training. This acts as a form of regularization and improves performance. We evaluate MLPs with Dropout to see if this is the case for this experiment.

Lastly, we experiment with SGD and Adam to see which optimizer performs better in these tasks.

For all of the experiments, we keep the same seed and hyperparameters to keep the comparisons as fair as possible. We first compare the activation functions, then choose the better one, then we compare the optimizers and choose the better one, and then we compare the best model for that task with its dropout variant. Each training loop only trains on 1,000 epochs, with the same hyperparameters for everything else.

3.1 Regression Task

The main metrics I computed for regression were Mean Square Error (MSE), Root MSE (RMSE), and the R squared. The MLP had two hidden layers both with hidden size of 16.

I compared ReLU vs. SiLU at the beginning to see which of the two had better metrics. Both activation functions performed very similar, with ReLU slightly winning out. Although ReLU won, the differences were negligible, but I still moved on to test SGD vs. Adam with the ReLU MLP. This is where the most important revelation appeared. Adam performed way better in all metrics compared to SGD, showing that the momentum and smoothing of Adam makes training more efficient. Lastly, we compared the Adam MLP with Dropout layers and noticed that the performance degraded using Dropout.

I hypothesize that the negligible difference between ReLU and SiLU, and the worse performance of Dropout layers, is due to the fact that the dataset is small and simple. I summarize the results in the table below and show graphs the loss curve and parity plot of the MLP with Adam. The loss graph validates and shows that the ∇ library is implemented correctly and trains neural networks. The parity plot shows that the model is indeed learning good representations of the dataset.

Metric	ReLU w/ SGD	SiLU w/ SGD	ReLU w/ Adam	ReLU w/ Adam & Dropout
MSE ↓	0.766	0.789	0.469	0.693
RMSE ↓	0.875	0.888	0.685	0.832
R ² ↑	0.4156	0.397	0.641	0.470

Table 1: Regression performance across different models on California Housing dataset.

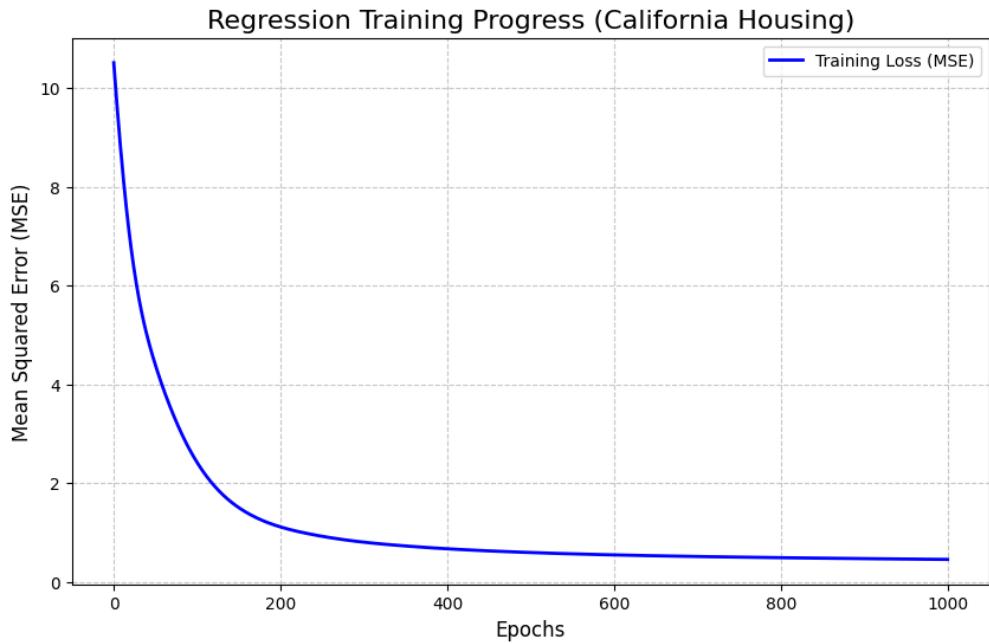


Figure 8: Loss graph of MLP with ReLU and Adam.

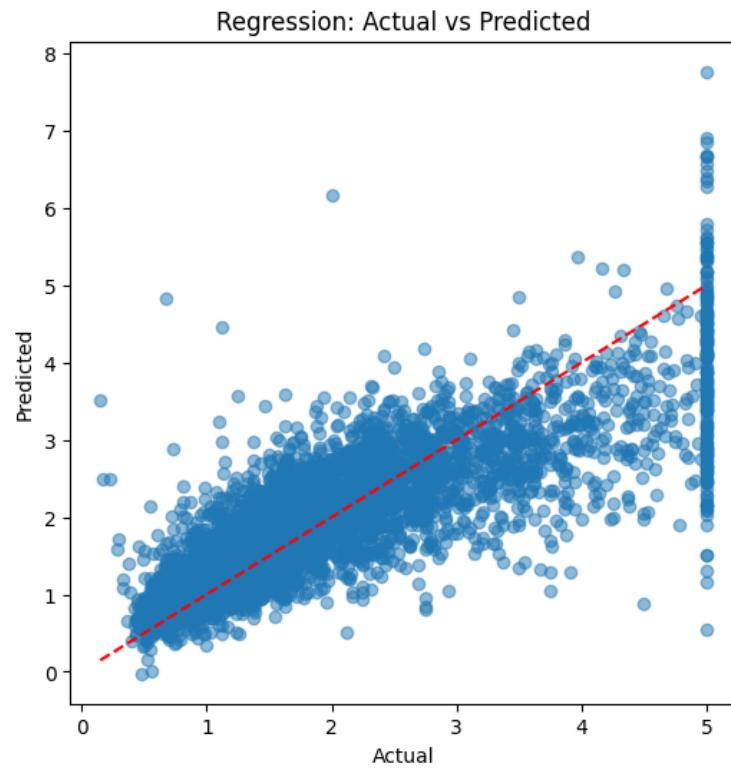


Figure 9: Actual vs. Predicted Parity Plot

3.2 Classification Task

For classification, I used the accuracy, precision, recall and F1 metrics to compare the models. Due to the CIFAR-10 dataset having an input size of 3,072, I increased the hidden size of the MLP in all hidden layers to 100.

First I compared Sigmoid vs. Tanh. Unlike the regression task, where ReLU and SiLU were comparable, in this case, Tanh performed vastly better than Sigmoid across all metrics, winning by over 10% on all metrics. This validates the use of the Tanh activation function over the Sigmoid activation function that is seen in literature. Not surprisingly, Adam beat out SGD once again in all metrics by at least 6%, showing the superiority of Adam in these experiments. And for the last experiment, adding Dropout to the Tanh MLP with Adam increased the metrics compared to all models by at least 9%, validating the use of dropout as a regularization method to increase performance and generalization.

I hypothesize that a big reason there was huge differences between this task and regression was because of the nature of the data. CIFAR-10 is a high-dimensional and much bigger dataset and I operated the models in pixel space for the input. This meant that the huge weighted sums caused Sigmoid to learn slowly, while Tanh avoided that problem. Adam won because in this very high dimensional space, Adam is able to smooth out and use momentum to have a better time finding the minima. And lastly, dropout worked in this cause because of how big the network was. Dropping out units randomly helped the model with classification of only 10 classes. Below I show a table of the results and plot the loss graph of Tanh MLP with Adam and Dropout.

Metric	Sigmoid w/ SGD	Tanh w/ SGD	Tanh w/ Adam	Tanh w/ Adam & Dropout
Accuracy ↑	20.80	31.67	37.85	47.38
Precision ↑	18.51	30.61	38.15	47.19
Recall ↑	20.80	31.67	37.85	47.38
F1 ↑	17.75	30.45	37.95	47.18

Table 2: Classification performance across different models on CIFAR-10.

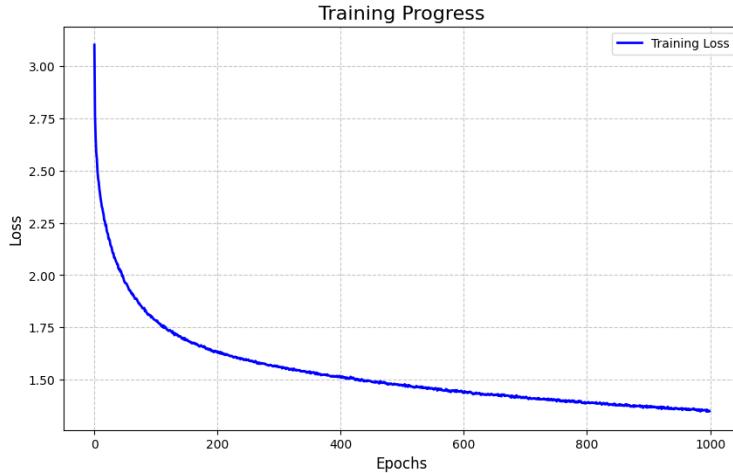


Figure 10: Loss graph of MLP with Tanh, Adam, and Dropout

4 Conclusion

In this report, I introduced Nabla ∇ , a simple neural network library implemented from scratch in Python using the Numpy library. Nabla ∇ supports the Linear layer, a variety of activation and loss functions, and two widely used optimizers, SGD and Adam. It is also compatible with other AI/ML libraries. On top of the code, this report also includes the derivations and explanations for the calculus and linear algebra that is under the hood of MLP neural networks. The code shows everything implemented while this report shows the math behind it. I validated the library by running experiments across different models in regression and classification tasks. In the experiments, I showed that there are differences between the activation functions and optimizers given the datasets. Namely, in smaller datasets, ReLU and SiLU performed similar. In a bigger dataset, Tanh beat Sigmoid by a wide margin. In both tasks, Adam beat SGD by a wide margin as well. Dropout was shown to be useful in a larger dataset and larger model.

5 References

References

- [1] PyTorch Team. *Computational Graphs Constructed in PyTorch*.
<https://pytorch.org/blog/computational-graphs-constructed-in-pytorch/>
- [2] Alex Lenail. *NN-SVG: Publication-ready Neural Network Architecture Diagrams*.
<https://alexlenail.me/NN-SVG/>
- [3] PyTorch Documentation. *torch.nn.ReLU*.
<https://pytorch.org/docs/stable/generated/torch.nn.ReLU.html>
- [4] PyTorch Documentation. *torch.nn.SiLU*.
<https://pytorch.org/docs/stable/generated/torch.nn.SiLU.html>
- [5] PyTorch Documentation. *torch.nn.Sigmoid*.
<https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html>
- [6] PyTorch Documentation. *torch.nn.Tanh*.
<https://pytorch.org/docs/stable/generated/torch.nn.Tanh.html>
- [7] KDnuggets. *Tuning the Adam Optimizer Parameters in PyTorch*. <https://www.kdnuggets.com/2022/12/tuning-adam-optimizer-parameters-pytorch.html>