DeepDive: Deep Learning Library from scratch

Hills Road Sixth Form College

Minhal Valiya Peedikakkal

Computer Science NEA

September 2023

1 Analysis

1.1 Identification of the problem

The increasing complexity of deep learning libraries has presented significant challenges for new AI researchers and enthusiasts seeking to enhance these libraries for their own use. One major factor contributing to this difficulty is the use of performance optimization by library developers, which can lead to significant portions of the code being written in lower-level languages like C++. PyTorch, the most widely-used deep learning library, is a prime example of this, with 43% of its code base consisting of C++. As a result, researchers with expertise in higher-level programming languages like Python may find it challenging to contribute and customize these libraries. To address this issue, I will aim to develop a deep learning library in python using 100% python code with a focus on simplicity over complexity. This way it is easier for AI researchers and enthusiasts to understand and develop this library.

1.2 Clients

My primary client is Alex Davicenko, a programmer who is interested in learning deep learning. He is aware of the theory behind neural networks but is less proficient in its practical side. Although my primary client is not an AI researcher, he is another type of user that this project is aimed at: beginners. I interviewed my client to gain insights before constructing my project requirements

Interview with Client

ME: How do you think a deep learning library implemented fully in Python could help you and other new users?

ALEX DAVICENKO: A Python-only library would allow me to inspect all of the code and understand how everything fits together. Even if it sacrifices some performance, I think a simpler, more transparent design is more important for learning and experimentation. I could also easily modify and extend it since I'm proficient in Python. This would really accelerate my ability to try new ideas with deep learning.

ME: What challenges have you faced when trying to use existing deep learning libraries like PyTorch?

ALEX DAVICENKO: I find existing libraries like PyTorch very complex and difficult to modify or extend. There is a lot of low-level C++ code that I don't understand. As a Python programmer without a background in C++, I struggle to contribute new features or optimizations.

ME: Any extra features which you would specifically like to see in the library?

ALEX DAVICENKO: Examples of features I would like to see is a smart and intuitive method of constructing networks using layers. I would also like to easily add custom layers and operators (only using python).

Client Hardware Specification

Knowing the client's hardware is useful for determining the appropriate types and sizes of models that can serve as examples for library usage.

- GPU
  - Model: GeForce RTX 3060
  - VRAM: 12 GB GDDR6
  - CUDA Cores: 3,584
- CPU
  - Model: AMD Ryzen 7 5700X
  - 8 cores 16 threads
- Memory
  - RAM: 16 GB

1.3 Existing similar software - PyTorch

PyTorch is the most popular and regarded as the best deep learning library (as of 2023), so I'll use it as a benchmark to showcase my project's advantages.

Installation

Installing PyTorch involves downloading a single package along with common dependencies, which is approximately 2.0GB in size. This can be a long process for new developers, especially considering the numerous dependencies required for GPU support. In contrast, my project aims to utilize lightweight packages like NumPy and CuPy (optional for GPU support), ensuring a simpler installation experience.

Usage

One of the important aspects of the library is its ease-of-use. Using PyTorch and observing its features has given me inspiration and ideas to how I should implement my library.

- **Automatic Differentiation:** The library must provide automatic differentiation capabilities. PyTorch's 'autograd' module provides this, enabling users to compute gradients easily.

```
    import torch

    # Define the input tensors with requires_grad=True to enable gradient
        computation
    a = torch.tensor([2.0], requires_grad=True)
    b = torch.tensor([3.0], requires_grad=True)

    # Define the function y = x^2 + 3xz + z^2
    y = a.pow(2) + 3 * a * b + b.pow(2)

    # Compute the gradients of y with respect to x and z using PyTorch's
        autodiff
    y.backward(torch.ones_like(y))

    # Print the gradients of y with respect to x and z
    print(a.grad)  # Output: tensor([7.])
    print(b.grad)  # Output: tensor([10.])
```

- **Neural Networks:** Implementing common neural network functionality such as layers would be beneficial for building deep learning models. I will make an interface similar to PyTorch as shown below to construct neural networks. I should also make it easy for users to add new layers to the library.

```
    import torch
    import torch.nn as nn

    # Define a simple neural network with one hidden layer
    class Net(nn.Module):
        def __init__(self):
            super(Net, self).__init__()
            self.fc1 = nn.Linear(2, 3)
            self.fc2 = nn.Linear(3, 1)

        def forward(self, x):
            x = torch.relu(self.fc1(x))
            x = self.fc2(x)
            return x

    # Define the input tensor
    x = torch.tensor([[0.0, 0.0], [0.0, 1.0], [1.0, 0.0], [1.0, 1.0]])

    # Define the neural network
    net = Net()

    # input x into the net
    y = net(x)
```

- **Data Utilities:** Providing utilities for data preprocessing, augmentation, and handling datasets can simplify the data pipeline.

```
    import data_utils
    p
    dataset = data_utils.Dataset(data, labels)
    dataloader = data_utils.DataLoader(dataset, batch_size=32, shuffle=True)

    for inputs, targets in dataloader:
        # Training code here
```

These are just a few examples of features that could be implemented in your library. By incorporating such functionality, users will find it easier to work with your library and build sophisticated machine learning models.

1.4 Algorithms Research

This section contains the algorithms I have researched and plan to use in my solution

Automatic Differentiation

Automatic differentiation (in short autodiff) is a technique used to numerically compute the exact gradient of a function. It exploits the fact that every computation, no matter how complex, executes a combination of elementary arithmetic operations e.g. addition and multiplication and elementary functions e.g. *sin, cos, exp, log* etc. The chain rule is applied to these operations allowing the gradient of complex function to be broken down and calculated independently and automatically

$$\begin{tikzcd} x \arrow[r] & a \arrow[r] & b \arrow[r] & y \end{tikzcd}$$

The graph above shows the data flow from the inputs x to the outputs y via variables a and b. Using the chain rule from calculus, we are able to calculate the gradient of y with respect to x

$$\frac{dy}{dx} = \frac{dy}{db} \cdot \frac{db}{da} \cdot \frac{da}{dx}$$

Automatic Differentiation has forward and reverse modes. The difference in both these methods are the order in which they are multiplied. In reverse mode autodiff, the flow of the gradients is reverse to the flow of the data [2]

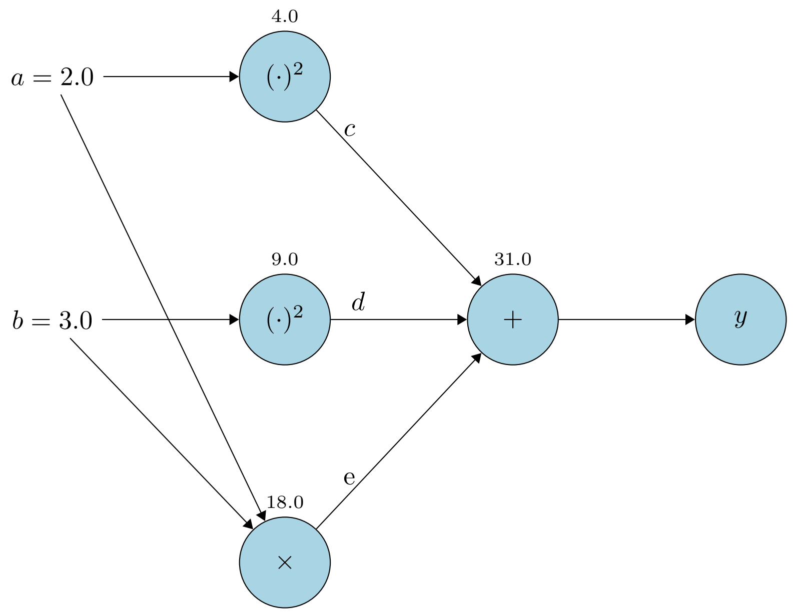
$$\frac{dy}{dx} = \left(\frac{dy}{db} \cdot \frac{db}{da}\right) \cdot \frac{da}{dx} \quad \textbf{Reverse mode}$$

$$\frac{dy}{dx} = \frac{dy}{db} \cdot \left(\frac{db}{da} \cdot \frac{da}{dx}\right) \quad \textbf{Forward mode}$$

In neural networks, the input dimensionality is commonly much higher than that of the labels. As a result, reverse mode autodiff is computationally cheaper than forward mode for the application of neural networks [2]. Using the code snippet that shows the usage of PyTorch's autodiff, the process which automatic differentiation takes to find the gradients

$$a = 2.0$$

$$b = 3.0$$
$$c = a^2$$
$$d = b^2$$
$$e = 3ab$$
$$y = c + e + d$$

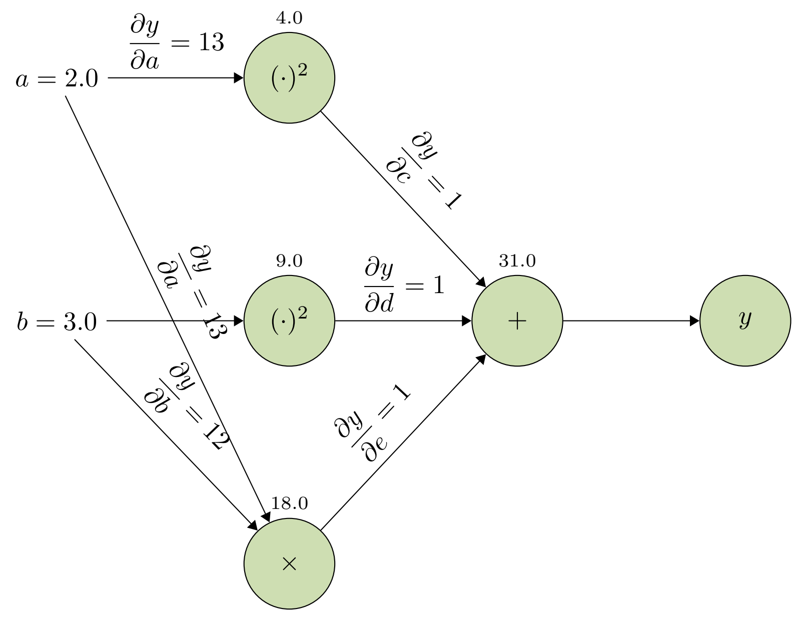


In reverse mode automatic differentiation, the forward pass involves evaluating the function and recording all intermediate values and dependencies without computing any derivatives. This creates a computational graph with nodes for each variable connected by edges representing dependencies, as depicted above.

After the forward pass completes, the reverse pass begins by calculating the "Adjoint" values. The Adjoint represents the partial derivative of the final output with respect to that intermediate variable. It is computed recursively starting from the output node and working backwards through the graph.

$$\bar{v}_i = \frac{\partial f}{\partial v_i} = \sum_{j:\text{ child of } i} \bar{v}_j \frac{\partial v_j}{\partial v_i} \ \textbf{Adjoint}$$

Specifically, for each node, the Adjoint is calculated by taking the sum of the Adjoints of all the node's children multiplied by the derivative of the child with respect to the current node. This effectively chains together the partial derivatives using the chain rule of calculus. So the Adjoint values propagate backwards from the output through the intermediate variables.



$a = 2.0$
$b = 3.0$
$\frac{\partial y}{\partial a} = 13$
$\frac{\partial y}{\partial a} = 13$
$\frac{\partial y}{\partial b} = 12$
$\frac{\partial y}{\partial c} = 1$
$\frac{\partial y}{\partial d} = 1$
$\frac{\partial y}{\partial e} = 1$
4.0
$(\cdot)^2$
9.0
$(\cdot)^2$
18.0
$\times$
31.0
$+$
y

Once the Adjoints are computed for every node, they represent the partial derivatives of the final output with respect to each intermediate variable. This gradient information can then be used for things like optimizing parameters using gradient descent. The reverse mode approach is efficient for cases where the number of inputs is large compared to the number of outputs. Reverse mode autodiff builds up a computational graph through the forward pass, then traverses it backwards to calculate gradients using the Adjoint, avoiding redundant derivative calculations. This allows efficient computation of gradients through long sequences of mathematical operations. This is how the library will calculate the gradient of the parameters of the network (with respect to loss etc) efficiently.

Neural Network Optimization Algorithms

Implementing both SGD and Adam optimization algorithms provides a solid foundation with 2 main benefits. SGD's simplicity with minimal hyperparameters accelerates initial testing during development. Its low memory requirements also allows scaling up network sizes. Meanwhile, Adam offers nice properties like fast initial convergence on some problems. Having both available gives users flexibility to choose the right optimizer for their needs and enables research into the tradeoffs. SGD serves as a simple, default option, while Adam provides an alternative for when its better convergence speed is needed.

Stochastic Gradient Descent

Stochastic gradient descent (SGD) is a basic neural network optimization algorithm used to minimize an objective function. It addresses the computational inefficiency of traditional gradient descent methods when dealing with large datasets hence making it useful for training neural networks. SGD processes training data in small batches instead of the entire dataset at once (batch training). Instead of using the entire dataset for each iteration, a small random batch is selected to calculate the gradient and update the model parameters. This random selection from the dataset introduces randomness into the optimization process, hence the term "stochastic" in stochastic gradient descent. SGD can converge faster than batch training because it performs updates more frequently. The learning rate is a hyperparameter which specifies the size of the step taken by gradient descent. The steps taken by the SGD algorithm [4]:

1. Randomly shuffle the data set of size m
2. Select a learning rate α
3. Generate initial random parameter values (using Xavier initialization) θ as the starting point
4. Compute the gradient vector with respect to a random single training example or small batch $x^j, y^j = \nabla_\theta J(\theta; x^j; y^j)$
5. Update all parameters θ, compute $\theta_{i+1} = \theta_i - \alpha \times \nabla_\theta J(\theta; x^j; y^j)$
6. Repeat Step 4 and 5 until a local minimum is reached

Adaptive Moment Estimation (ADAM)

The Adam optimizer is a combination of the best properties of the Momentum (with SGD) and RMSProp algorithms. Momentum helps accelerate gradients in the right direction and dampens oscillations. RMSprop divides the learning rate by a running average of the magnitudes of recent gradients. Adam is relatively easy to configure where the default configuration parameters do well on most problems. The Adam optimizer calculates a running average of the gradients and the squared gradients for each parameter in the model. It then uses these averages to calculate the update for each parameter during training. It is an adaptive learning rate algorithm designed to improve training speeds in deep neural networks and reach convergence quickly. The steps taken by the Adam Algorithm [1]:

1. Select hyperparameters: learning rate α, 1st moment decay β_1, 2nd moment decay β_2, and small constant ϵ
2. Calculate the gradient of the loss function:
$$\frac{\partial \mathcal{J}}{\partial W^{[l]}}$$
3. Update exponential moving average of past gradients (1st moment estimate):
$$v_{dW^{[l]}} = \beta_1 v_{dW^{[l]}} + (1 - \beta_1)\frac{\partial \mathcal{J}}{\partial W^{[l]}}$$
4. Update exponential moving average of squared past gradients (2nd moment estimate):
$$s_{dW^{[l]}} = \beta_2 s_{dW^{[l]}} + (1 - \beta_2)(\frac{\partial \mathcal{J}}{\partial W^{[l]}})^2$$
5. Bias correct the 1st moment exponential moving average:
$$v^{\text{corrected}}_{dW^{[l]}} = \frac{v_{dW^{[l]}}}{1 - (\beta_1)^t}$$
6. Bias correct the 2nd moment exponential moving average:
$$s^{\text{corrected}}_{dW^{[l]}} = \frac{s_{dW^{[l]}}}{1 - (\beta_2)^t}$$
7. Update the weights using the bias corrected 1st and 2nd moment estimates:
$$W^{[l]} = W^{[l]} - \alpha \frac{v^{\text{corrected}}_{dW^{[l]}}}{\sqrt{s^{\text{corrected}}_{dW^{[l]}}} + \varepsilon}$$

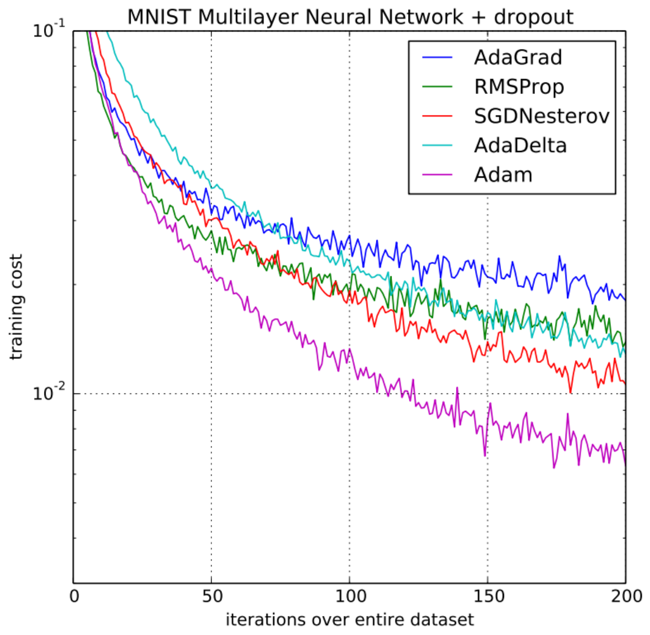


Figure 1: Adam performance compared to other optimizers [3]

2-Dimensional Convolutions

Interview Analysis

Regularization

Regularization is a technique used to reduce overfitting in neural networks. Overfitting occurs when a model fits the training data too well but fails to generalize to new data. This happens when the network learns the noise in the training data in addition to the underlying patterns. Regularization penalizes complex models, thereby favoring simpler models that don't overfit. This improves the generalization of the network. Some common regularization techniques are:

- **L1 regularization**: Adds a penalty equal to the absolute value of the weights to the loss function. This forces sparse solutions by shrinking some weights to exactly 0.

  $$\text{Loss} = \text{Loss} + \lambda \sum_{i=1}^{n} |w_i|$$

  Where λ controls the regularization strength.

- **L2 regularization**: Adds a penalty equal to the square of the weights to the loss function. This causes the weights to prefer smaller non-zero values rather than exact zeros like L1. Also known as weight decay.

  $$\text{Loss} = \text{Loss} + \lambda \sum_{i=1}^{n} w_i^2$$

Xavier Initialization

Xavier initialization is a method for setting the starting values for the weights in a neural network. The goal is to prevent vanishing or exploding gradients during training. It works by making sure the signal passing through each layer stays at a similar level. Here's how:

1. The initial weights $\mathbf{W}$ are randomly drawn from a distribution with zero mean.
2. The variance of the distribution is set to $\frac{1}{n_{\text{in}}}$.
3. n_{in} is the number of inputs going into that layer.

For example, say a layer has 300 input units and 100 output units. To initialize the weights $\mathbf{W}$ for that layer:

1. Set the mean of the distribution to 0.
2. Set the variance to $\frac{1}{300}$. Since there are 300 inputs, $n_{\text{in}} = 300$.
3. Randomly draw values for $\mathbf{W}$ from this distribution.

This results in the output activations having about the same variance as the inputs.

So Xavier initialization carefully scales the initial random weights based on the number of inputs. This helps keep the signals and gradients flowing properly during training

Normalization

Normalization rescales the activations to speed up training. Two common normalization techniques are:

- **Batch Normalization**: Normalizes each batch independently during training by subtracting the batch mean and dividing by the batch standard deviation.
- **Layer Normalization**: Normalizes across all hidden units for each training example, rather than across batches. Applied across layers.

1.5 Objectives of the Deep Learning Library

Challenge objectives are written in *italics*

1. **Automatic Differentiation**
   - 1.1 Tensor object can store n-dimensional values for parameters and gradients
   - 1.2 Develop gradient functions for basic operations such as addition, subtraction, and matrix multiplication, as well as for common Deep Learning operations like logsoftmax, sigmoid, and ReLU.
   - 1.3 Implement a backward function that computes the gradient for each intermediate variable in the computational graph. This function should be recursive to simplify understanding and usage.
   - 1.4 Automatic Differentiation Testing
     - 1.4.1 Implement unit tests to validate correctness of automatic differentiation gradients.
     - 1.4.2 Compare gradients computed by automatic differentiation with gradients from PyTorch for a set of basic operations.
     - 1.4.3 Ensure absolute difference between automatic differentiation and PyTorch gradients is small (below threshold).
2. **Easy-to-use Neural Network Functionality**
   - 2.1 Create a NeuralNet class which can be used to construct neural networks from individual layers and operations
   - 2.2 Create a class for Layers which will be used to define a type of layer in the NeuralNet class. Implement common neural network layers (e.g. Linear Layers, 2D Convolutions Layers and Normalization layers)
     - 2.2.1 Fully-connected layer (PyTorch Linear Layer)
     - 2.2.2 2D Convolution

2.2.3 Normalization Layers

2.3 Activation Functions

2.3.1 Include activation functions e.g. ReLU, sigmoid, tanh, softmax etc.

2.4 Optimizers for neural network training

2.4.1 Stochastic Gradient Descent

2.4.2 *Adaptive Moment Estimation (Adam)*

2.4.3 Layers should be initialized with random weights

2.4.3.1 Xavier Initialization

2.4.4 Implement Regularization Techniques

2.4.4.1 L1 Regularization

2.4.4.2 L2 Regularization

2.4.4.3 *Batch Normalisation*

2.4.4.4 *Layer Normalisation*

2.4.5 Model saving to disk and loading from files. Use an appropriate method to export model weights.

2.4.6 *GPU support for neural network training*

2.5 **Data Utilities**

2.5.1 *Ability to easily load large datasets into the Tensor format using external libraries for downloading the dataset (e.g. HF datasets?)*

2.5.2 *A dataloader which can split data into batches for training*

2.5.3 *Datasets can be shuffled*

1.6 Prototyping and Modelling

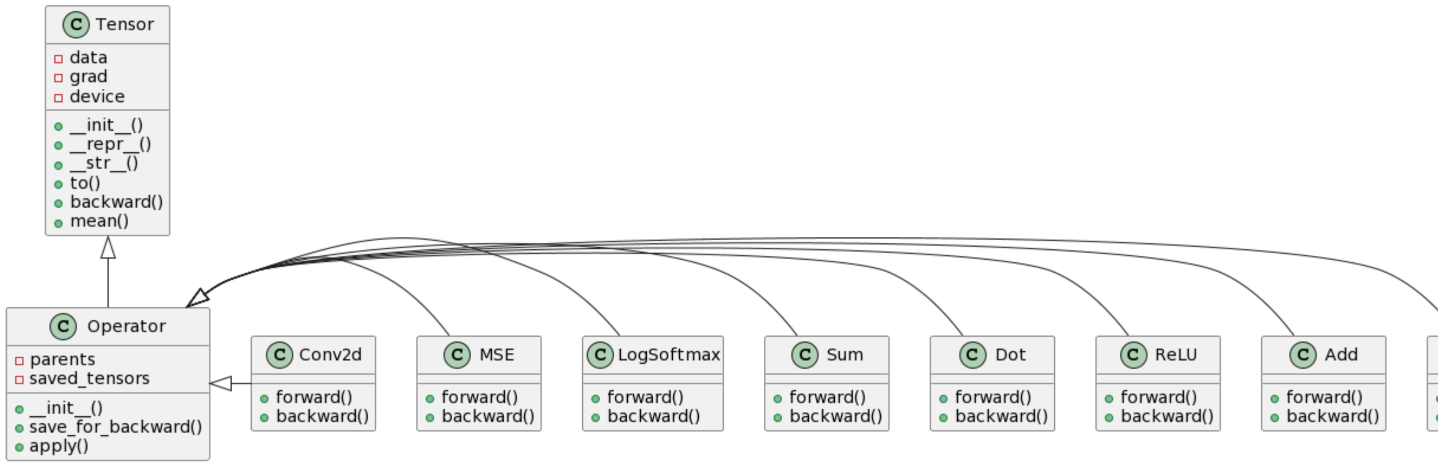


Figure 2: Tensor Class and subclasses

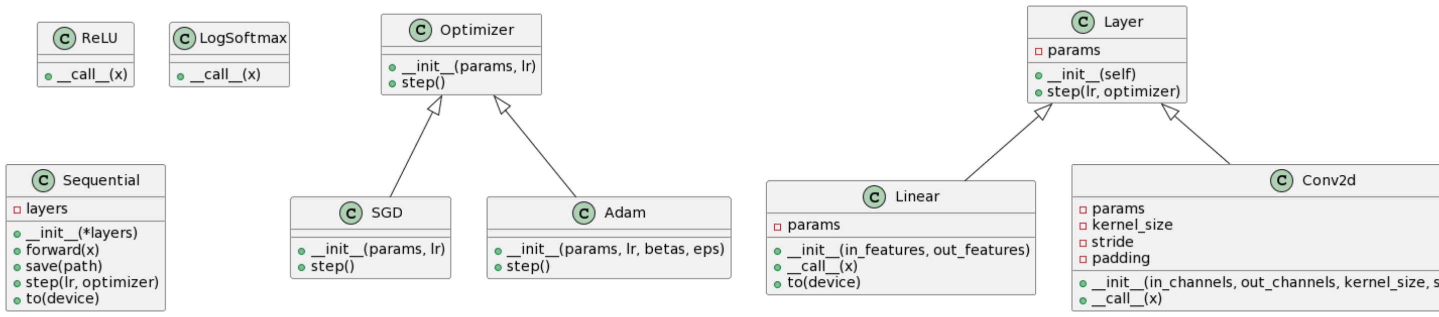


Figure 3: Neural Network classes

Bibliography

[1] Ajagekar, A. Adam - cornell university computational optimization open textbook - optimization wiki.

[2] Deisenroth, M. P., Faisal, A. A., and Ong, C. S. *Mathematics for Machine Learning.* Cambridge University Press, 2020.

[3] Kingma, D. P., and Ba, J. Adam: A method for stochastic optimization, 12 2014.

[4] Price, J., Wong, A., Yuan, T., Mathews, J., and Olorunniwo, T. Stochastic gradient descent - cornell university computational optimization open textbook - optimization wiki.