

Homework 1 Part 1

An Introduction to Neural Networks

11-785: INTRODUCTION TO DEEP LEARNING (FALL, 2023)

Release Date: **September 6, 2023, 11:59 P.M., E.S.T.**

Early Submission Bonus Date: **September 14, 2023, 11:59 P.M., E.S.T.**

Final Due Date: **September 23, 2023, 11:59 P.M., E.S.T.**

VERSION: 2.0.0

Start Here

- **Collaboration policy:**

- You are expected to comply with the [University Policy on Academic Integrity and Plagiarism](#).
- You are allowed to help your friends debug
- You are allowed to look at your friends code
- You are allowed to copy math equations from any source that are not in code form
- You are not allowed to type code for your friend
- You are not allowed to look at your friends code while typing your solution
- You are not allowed to copy and paste solutions off the internet
- You are not allowed to import pre-built or pre-trained models
- You can share ideas but not code, you must submit your own code. All code submitted will be compared with all code submitted this semester and in previous semesters using [MOSS](#).

We encourage you to meet regularly with your study group to discuss and work on your homework. You will not only learn more, you will also be more efficient that way. However, as noted above, the actual code used to obtain the final submission must be entirely your own.

- **Directions:**

- You are required to do this assignment in the Python (version 3) programming language. Do not use any auto-differentiation toolboxes (PyTorch, TensorFlow, Keras, etc.) - you are only permitted and recommended to vectorize your computation using the Numpy library.
- We recommend that you look at all the problems before trying to solve the first one. However, we recommend that you complete the problems in order, as the difficulty increases, and questions often rely on the completion of previous questions.

- **Early submission bonus:**

- If you complete this assignment successfully and achieve full marks on Autolab before , you will receive **5** point bonus for this assignment.

Homework Objectives

In this homework, you will learn how to implement and train an entire MLP from scratch, on your own. You will learn

- to write code for all the components that comprise a simple MLP;
- to *chain* these components up to actually compose a complete MLP of any depth;
- to implement *losses* to train the network parameters;
- how to backpropagate the derivatives of those losses through the network, to compute loss derivatives with respect to all network parameters;
- how to incorporate those derivatives into stochastic gradient descent (SGD) to update network parameters;
- how to implement at least one common regularization method, namely batch normalization, to improve training.

This homework comes with an optional, separately posted bonus part, in which you will also learn to implement other optimizers including ADAM, and another key regularization technique, dropout.

Checklist

Here is a checklist page that you can use to keep track of your progress as you go through the write-up and implement the corresponding sections in your starter notebook. As you complete each function in the notebook, you can check the corresponding boxes aligned with each section.

1. Getting Started
 - Download code handout and extract the file
 - Install required python libraries
 - Read the whole assignment write-up for an overview
 - Watch the 10-min video [Backpropagation](#) by 3B1B to understand the `backward` methods ¹.
2. Complete the Components of a Multilayer Perceptron Model
 - Revisit lecture 2 about linear classifier, activation function, and perceptron
 - Complete the linear layer class
 - Complete the 4 activation functions
3. Complete 3 Multilayer Perceptron Models using Components Built
 - Revisit lecture 2 about MLP
 - Write a MLP model with 0 hidden layers
 - Write a MLP model with 1 hidden layers
 - Write a MLP model with 4 layers
4. Implement the Criterion Functions to evaluate a machine learning model
 - Revisit lecture 3 about Loss
 - Implement Mean Squared Error (MSE) Loss for regression models
 - Implement Cross-Entropy Loss for classification models
5. Implement an Optimizer to train a machine learning model
 - Revisit lecture 6 about momentum, and lecture 7 about SGD
 - Implement SGD optimizer
6. Implement a Regularization method: Batch Normalization
 - Revisit lecture 8 about Batch normalization
 - Translate the element-wise equations to matrix equations
 - Write the code based on the matrix equations you wrote
7. Hand-in
 - Set all flags to `True` in `hw1p1_autograder_flags.py`
 - Make sure you pass all textcases in the local autograder
 - Make the `handin.tar` file and submit to autolab

¹Lecture 5 will cover backpropagation in more details

Contents

1	Introduction to MyTorch series	6
2	Setup and Submission	6
3	Notation	9
4	The Big Picture	10
5	Neural Network Layers [15 Points]	11
5.1	Linear Layer [mytorch.nn.Linear]	11
5.1.1	Linear Layer Forward Equation	12
5.1.2	Linear Layer Backward Equation	12
6	Activation Functions [10 points]	14
6.1	Sigmoid [mytorch.nn.Sigmoid]	16
6.1.1	Sigmoid Forward Equation	16
6.1.2	Sigmoid Backward Equation	16
6.2	Tanh [mytorch.nn.Tanh]	16
6.2.1	Tanh Forward Equation	16
6.2.2	Tanh Backward Equation	16
6.3	ReLU [mytorch.nn.ReLU]	17
6.3.1	ReLU Forward Equation	17
6.3.2	ReLU Backward Equation	17
6.4	GELU [mytorch.nn.GELU]	18
6.4.1	GELU Forward Equation	18
6.4.2	GELU Backward Equation	18
6.5	Softmax [mytorch.nn.Softmax]	18
6.5.1	Softmax Forward Equation	18
6.5.2	Softmax Backward Equation	19
7	Neural Network Models [35 points]	20
7.1	MLP (Hidden Layers = 0) [mytorch.models.MLP0] [10 points]	21
7.1.1	MLP Forward Pseudocode (Hidden Layers = 0)	21
7.1.2	MLP Backward Pseudocode (Hidden Layers = 0)	21
7.2	MLP (Hidden Layers = 1) [mytorch.models.MLP1] [10 points]	22
7.2.1	MLP Forward Method Description (Hidden Layers = 1)	22
7.2.2	MLP Backward Method Descriptions (Hidden Layers = 1)	23
7.3	MLP (Hidden Layers = 4) [mytorch.models.MLP4] [15 points]	24
7.3.1	MLP Forward Equations (Hidden Layers = 4)	25
7.3.2	MLP Backward Equations (Hidden Layers = 4)	25
8	Criterion - Loss Functions [10 points]	26
8.1	MSE Loss [mytorch.nn.MSELoss]	27
8.1.1	MSE Loss Forward Equation	27
8.1.2	MSE Loss Backward Equation	27
8.2	Cross-Entropy Loss [mytorch.nn.CrossEntropyLoss]	28
8.2.1	Cross-Entropy Loss Forward Equation	28
8.2.2	Cross-Entropy Loss Backward Equation	28
9	Optimizers [10 points]	29
9.1	Stochastic Gradient Descent (SGD) [mytorch.optim.SGD]	29
9.1.1	SGD Equation (Without Momentum)	30
9.1.2	SGD Equations (With Momentum)	30

10 Regularization [20 points]	31
10.1 Batch Normalization [<code>mytorch.nn.BatchNorm1d</code>]	31
10.1.1 Batch Normalization Forward Training Equations (When <code>eval = False</code>)	33
10.1.2 Batch Normalization Forward Inference Equations (When <code>eval = True</code>)	35
10.1.3 Batch Normalization Backward Equations	35
11 Appendix	37
11.1 Anaconda Installation and Setup Instructions	37

1 Introduction to MyTorch series

In this series of homework assignments, you will implement your own deep-learning library from scratch. Inspired by PyTorch, your library – *MyTorch* – will be used to create everything from multilayer perceptrons (MLP), convolutional neural networks (CNN), to recurrent neural networks with gated recurrent units (GRU), and long-short term memory (LSTM) structures. This is an ambitious undertaking, and we are here to help you through the entire process. At the end of this work, you will understand forward propagation, loss calculation, backward propagation, and gradient descent.

The culmination of all of the Homework Part 1's will be your own custom deep learning library *MyTorch*[©], along with detailed examples. It is structured similarly to popular deep library learning libraries like PyTorch and TensorFlow, and you can easily import and reuse code modules for subsequent homework.

In this assignment, we will start by creating the core components of multilayer perceptrons: linear layers, activation functions, and batch normalization. Then, you will implement loss functions and stochastic gradient decent optimizer in MyTorch. The auto-grader tests will compare the outputs of your MyTorch methods and class attributes with a reference PyTorch solution. We have made the necessary components of these classes and class functions as explicit as possible. Your job is to understand how all the components are related and implement the mathematics into code.

In looking at the mathematics, you will be coding the equations needed to build a simple Neural Network Layer. This includes forward and backward propagation for the activations, loss functions, linear layers, and batch normalization. If you have challenges going from math to code, consider the shapes involved and do what you can to make the operations possible.

Welcome, and we are grateful to be with you on this journey!

2 Setup and Submission

- **Extract** the downloaded handout *hw1p1_handout.tar* by running the following command in the same directory²

```
tar -xvf hw1p1_handout.tar
```

This will create a directory called HW1P1 with the following file structure.

```
HW1P1
├── mytorch
│   ├── nn
│   │   ├── linear.py
│   │   ├── activation.py
│   │   ├── loss.py
│   │   └── batchnorm.py
│   ├── optim
│   │   └── sgd.py
│   └── models
│       └── mlp.py
├── hw1p1_autograder_flags.py
├── hw1p1_autograder.py
└── requirements.txt
```

Apart from the above major files, there are files named `__init__.py` that don't have to be edited. You may also see other files like `.DS_STORE` or `__pycache__` folders that you should not be concerned about.

²The handout might have an extension like `handout.tar.112`. In such a case, you will first have to rename the downloaded file as `handout.tar` by removing the `.112` extension and then `untar` the file.

- **Install Anaconda and Setup Environment**

Please refer to the [Appendix](#) section for detailed instructions on setting up Anaconda environment on different operating systems (Windows, macOS, and Linux).

- **Follow the writeup and edit code files**

The sections of the writeup are ordered to help you build the MyTorch library incrementally. Each section has a corresponding Python file that contains code for classes implementing the theory in that section. For instance the section on activation functions corresponds to `activation.py`, the neural network models section to `mlp.py`, etc. You need to edit these files according to the writeup.

Another thing to note is the file structure. The `mytorch` folder contains code for individual components like linear layers, optimizer, losses, etc. These components are independent of each other. The `models` folder on the other hand has code for an entire neural network that uses some of these independent components. You can follow a similar structure if you try to write the entire code from scratch.

Lastly, testing the code is not performed by these files. There is a separate `hw1p1_autograder.py` to help you do that. It runs some local tests as a preliminary check of code correctness. Instructions to use it are given below.

- **Autograde your code by**

- Step 1: Open your preferred IDE or code editor, locate and open the desired `.py` file, make the necessary edits, and save the changes.
- Step 2: (**IMPORTANT**): Setting the flags in `hw1p1_autograder_flags.py` to `True` to test any individual component on your local autograder. For example, if you only implement the sigmoid activation functions, set `DEBUG_AND_GRADE_SIGMOID_flag = True` and everything else to `False`.
- Step 3: Running local autograder by: Confirm that you are in the top level director and execute the following in anaconda prompt or terminal:

```
python hw1p1_autograder.py
```

Please keep in mind that the local autograder only has a few tests as a preliminary check. The entire suite of tests is run on Autolab after you hand-in your code as described below.

- **Hand-in** your code by running the following command from the top level directory, then **SUBMIT** the created `handin.tar` file to autolab³:

```
tar -cvf handin.tar models mytorch
```

- **DO**

- Make sure you understand the concept of each function, we don't want you to "translate" math equations to codes without understanding them.
- Go through the examples we provide to have a better visualization of the matrix calculations. If you ask TAs for help, we will ask you to explain the example to us before giving you more hints.

- **DO NOT**

- Import other external libraries other than `numpy` in your submission, as extra packages that do not exist in autolab will cause submission failures.⁴ Libraries like `PyTorch`, `TensorFlow`, `Keras` are not allowed.
- Add, move, or remove any files or change any filenames.

³If you are a Windows user, navigate to HW1P1 directory and run "create_tarball.sh" in the terminal.

⁴We are not intending to make the `numpy` restriction arbitrarily prohibitive. You can use `os`, `sys`, `matplotlib`, and other functions needed to get familiar with your environment and what is going on. However, AutoLab expects only `numpy`, `math` and `scipy`. Remove other libraries when making the submission.

- **Scoring:**

The homework comprises several sections. You get points for each section. Within any individual section, however, you are expected to pass all tests within the section to get the score for it. Sections do not have partial credit.

The local autograder provided to you has is very detailed. You will be able to isolate and verify individual components of the sections on it. This can help you identify any issues or bugs in your code that need to be addressed. Make sure you get full points on the local autograder for any section, before submitting it to autolab.

3 Notation

****Numpy Tips:**

- Use $A * B$ for element-wise multiplication $A \odot B$.
- Use $A @ B$ for matrix multiplication $A \cdot B$.
- Use A / B for element-wise division $A \oslash B$.

Linear Algebra Operations

A^T	Transpose of A
$A \odot B$	Element-wise (Hadamard) Product of A and B (i.e. every element of A is multiplied by the corresponding element of B. A and B must have identical size and shape)
$A \cdot B$	Matrix multiplication of A and B
$A \oslash B$	Element-wise division of A by B (i.e. every element of A is divided by the corresponding element of B. A and B must have identical size and shape)

Set Theory

\mathbb{S}	A set
\mathbb{R}	The set of real numbers
$\mathbb{R}^{N \times C}$	The set of $N \times C$ matrices containing real numbers

Functions and Operations

$f : \mathbb{A} \rightarrow \mathbb{B}$	The function f with domain \mathbb{A} and range \mathbb{B}
$\log(x)$	Natural logarithm of x
$\varsigma(x)$	Sigmoid, $\frac{1}{(1 + \exp^{-x})}$
$\tanh(x)$	Hyperbolic tangent, $\frac{e^x - e^{-x}}{e^x + e^{-x}}$
$\max_{\mathbb{S}} f$	The operator $\max_{a \in \mathbb{S}} f(a)$ returns the highest value $f(a)$ for all elements in the set \mathbb{S}
$\arg \max_{\mathbb{S}} f$	The operator $\arg \max_{a \in \mathbb{S}} f(a)$ returns the element of the set \mathbb{S} that maximizes $f(a)$
$\sigma(x)$	Softmax function, $\sigma : \mathbb{R}^K \rightarrow (0, 1)^K$ and $\sigma(x)_i = \frac{e^{x_i}}{\sum_{j=1}^K e^{x_j}}$ for $i = 1, \dots, K$

Calculus

$\frac{dy}{dx}$	Derivative of scalar y with respect to scalar x
$\frac{\partial y}{\partial x}$	Partial derivative of scalar y with respect to scalar x
$\frac{\partial f(Z)}{\partial Z}$	Jacobian matrix $\mathbf{J} \in \mathbb{R}^{N \times M}$ of $f : \mathbb{R}^M \rightarrow \mathbb{R}^N$

4 The Big Picture

We can think of a neural network (NN) as a mathematical function which takes an input data x and computes an output y :

$$y = f_{NN}(\mathbf{x})$$

For example, a model trained to identify spam emails takes in an email as input data x , and output 0 or 1 indicating whether the email is spam.

The function f_{NN} has a particular form: it's a *nested function*. In lecture, we learnt the concepts of network **layers**. So, for a 3-layer neural network that returns a scalar, f_{NN} looks like this:

$$y = f_{NN}(\mathbf{x}) = f_3(\mathbf{f}_2(\mathbf{f}_1(\mathbf{x})))$$

In the above equation, \mathbf{f}_1 and \mathbf{f}_2 are vector functions of the following form:

$$f_l(z) = g_l(W_l \cdot z + b_l)$$

where l is called the layer index. The function \mathbf{g}_l is called an **activation function** (e.g. **ReLU**, **Sigmoid**). The parameters \mathbf{W}_l (weight matrix) and \mathbf{b}_l (bias vector) for each layer are learnt using **gradient descent** by optimizing a particular **loss function**⁵ depending on the task.

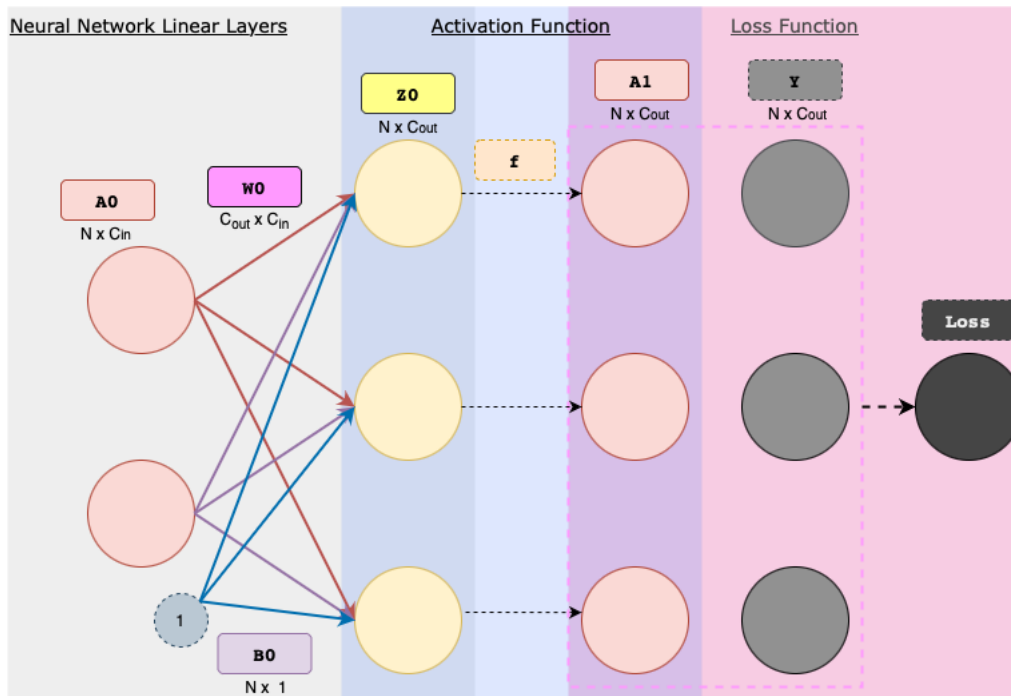


Figure A: End-to-End Topology.

In this assignment, we will create one architecture of neural networks called **multilayer perceptron (MLP)**. Refer to Figure A.

⁵The terms cost function and loss function are analogous.

5 Neural Network Layers [15 Points]

5.1 Linear Layer [mytorch.nn.Linear]

Linear layers, also known as **fully-connected layers**, connect every input neuron to every output neuron and are commonly used in neural networks. Refer to Figure A for the visual representation of a linear layer. In this section, your task is to implement the Linear class in file `linear.py`:

- Class attributes:
 - Learnable model parameters weight W , bias b .
 - Variables stored during forward-propagation to compute derivatives during back-propagation: layer input A , batch size N ⁶.
 - Variables stored during backward-propagation to train model parameters $dLdW$, $dLdb$.
- Class methods:
 - `__init__`: Two parameters define a linear layer: `in_feature` (C_{in}) and `out_feature` (C_{out}). Zero initialize weight W and bias b based on the inputs. Refer to Table 5.1 to see how the shapes of W and b are related to the inputs.
 - `forward`: forward method takes in a batch of data A of shape $N \times C_{in}$ (representing N samples where each sample has C_{in} features), and computes output Z of shape $N \times C_{out}$ – each data sample is now represented by C_{out} features.
 - `backward`: backward method takes in input $dLdZ$, how changes in its output Z affect loss L . It calculates and stores $dLdW$, $dLdb$ – **how changes in the layer weights and bias affect loss**, which are used to improve the model. It returns $dLdA$, how changes in the layer inputs affect loss to enable downstream computation.

Please consider the following class structure:

```
class Linear:

    def __init__(self, in_features, out_features):
        self.W = # TODO
        self.b = # TODO

    def forward(self, A):
        self.A = # TODO
        self.N = # TODO: store the batch size
        Z = # TODO

        return Z

    def backward(self, dLdZ):
        dLdA = # TODO
        dLdW = # TODO
        dLdb = # TODO
        self.dLdW = dLdW
        self.dLdb = dLdb

        return dLdA
```

⁶**Important:** We will introduce the concept of "batch" in lecture 7, for now, think of batch size as number of input samples

Table 1: Linear Layer Components

Code Name	Math	Type	Shape	Meaning
<code>N</code>	N	scalar	-	batch size
<code>in_features</code>	C_{in}	scalar	-	number of input features
<code>out_features</code>	C_{out}	scalar	-	number of output features
<code>A</code>	A	matrix	$N \times C_{in}$	batch of N inputs each represented by C_{in} features
<code>Z</code>	Z	matrix	$N \times C_{out}$	batch of N outputs each represented by C_{out} features
<code>W</code>	W	matrix	$C_{out} \times C_{in}$	weight parameters
<code>b</code>	b	matrix	$C_{out} \times 1$	bias parameters
<code>dLdZ</code>	$\partial L / \partial Z$	matrix	$N \times C_{out}$	how changes in outputs affect loss
<code>dLdA</code>	$\partial L / \partial A$	matrix	$N \times C_{in}$	how changes in inputs affect loss
<code>dLdW</code>	$\partial L / \partial W$	matrix	$C_{out} \times C_{in}$	how changes in weights affect loss
<code>dLdb</code>	$\partial L / \partial b$	matrix	$C_{out} \times 1$	how changes in bias affect loss

5.1.1 Linear Layer Forward Equation

During forward propagation, we apply a linear transformation to the incoming data **A** to obtain output data **Z** using a **weight matrix W** and a **bias vector b**. ι_N is a column vector of size N which contain all 1s, and is used for broadcasting⁷ the bias.

$$Z = A \cdot W^T + \iota_N \cdot b^T \in \mathbb{R}^{N \times C_{out}} \quad (1)$$

$$\begin{array}{c}
 \begin{array}{c} A \\ \text{A} \end{array} \cdot \begin{array}{c} W^T \\ \text{W} \end{array} + \begin{array}{c} \iota \\ \text{1} \end{array} \cdot \begin{array}{c} b^T \\ \text{b} \end{array} = \begin{array}{c} Z \\ \text{Z} \end{array} \\
 \begin{array}{|c|c|} \hline -4 & -3 \\ \hline -2 & -1 \\ \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} \cdot \begin{array}{|c|c|} \hline -2 & -1 \\ \hline 0 & 1 \\ \hline 2 & 3 \\ \hline \end{array} + \begin{array}{|c|} \hline 1 \\ \hline 1 \\ \hline 1 \\ \hline 1 \\ \hline \end{array} \cdot \begin{array}{|c|} \hline -1 \\ \hline 0 \\ \hline 1 \\ \hline \end{array} = \begin{array}{|c|c|c|} \hline 10 & -3 & -16 \\ \hline 4 & -1 & -6 \\ \hline -2 & 1 & 4 \\ \hline -8 & 3 & 14 \\ \hline \end{array}
 \end{array}$$

Figure B: Linear Layer Forward Example

5.1.2 Linear Layer Backward Equation

As mentioned earlier, the objective of backward propagation is to calculate the derivative of the loss with respect to the weight matrix, bias, and input to the linear layer, i.e., `dLdW`, `dLdb`, and `dLdA` respectively.

Given $\partial L / \partial Z$ as an input to the backward function, we can apply chain rule to obtain how changes in A , W , b affect loss L :

$$\frac{\partial L}{\partial A} = \left(\frac{\partial L}{\partial Z} \right) \cdot \left(\frac{\partial Z}{\partial A} \right)^T \in \mathbb{R}^{N \times C_{in}} \quad (2)$$

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial Z} \right)^T \cdot \left(\frac{\partial Z}{\partial W} \right) \in \mathbb{R}^{C_{out} \times C_{in}} \quad (3)$$

$$\frac{\partial L}{\partial b} = \left(\frac{\partial L}{\partial Z} \right)^T \cdot \left(\frac{\partial Z}{\partial b} \right) \in \mathbb{R}^{C_{out} \times 1} \quad (4)$$

⁷Read [numpy documentation](#) if you have never seen the word broadcasting before. We will refer to this term frequently in future homework.

In the above equations, \mathbf{dZdA} , \mathbf{dZdW} , and \mathbf{dZdb} represent how the input, weights matrix, and bias respectively affect the output of the linear layer.

Now, \mathbf{Z} , \mathbf{A} , and \mathbf{W} are all two-dimensional matrices (see Table 1 above). \mathbf{dZdA} would have derivative terms corresponding to each term of \mathbf{Z} with respect to each term of \mathbf{A} , and hence would be a 4-dimensional tensor. Similarly, \mathbf{dZdW} would be 4-dimensional and \mathbf{dZdb} would be 3-dimensional (since \mathbf{b} is 1-dimensional). These high-dimensional matrices would be sparse (many terms would be 0) as only some pairs of terms have a dependence. So, to make things simpler and avoid dealing with high-dimensional intermediate tensors, the derivative equations given above are simplified to the below form:

$$\frac{\partial L}{\partial A} = \left(\frac{\partial L}{\partial Z} \right) \cdot W \quad \in \mathbb{R}^{N \times C_{in}} \quad (5)$$

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial Z} \right)^T \cdot A \quad \in \mathbb{R}^{C_{out} \times C_{in}} \quad (6)$$

$$\frac{\partial L}{\partial b} = \left(\frac{\partial L}{\partial Z} \right)^T \cdot \iota_N \quad \in \mathbb{R}^{C_{out} \times 1} \quad (7)$$

6 Activation Functions [10 points]

Congratulations for finishing the first section! Here, we will introduce to you a few popular **activation functions** and how to implement them!

As a machine learning engineer, you can theoretically choose any **differentiable function** as the activation function. The primary purpose of having nonlinear components in the neural network (f_{NN}) is to allow it to **approximate nonlinear functions**. Without activation functions, f_{NN} will always be linear, no matter how deep it is. The reason is that $A \cdot W + b$ is a linear function, and a linear function of a linear function is also linear.

Activation functions can either take scalar or vector arguments. Scalar activations apply a function to a single number. Thus, when they are applied to a vector, they operate element-wise. This one-to-one dependence between the input and output makes calculating derivatives easier. Popular choices of scalar activation functions are **Sigmoid**, **ReLU**, **Tanh**, and **GELU**, as shown in Table 2. More details about these functions are provided in their respective subsections.

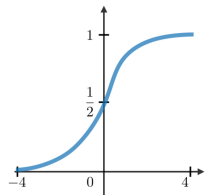
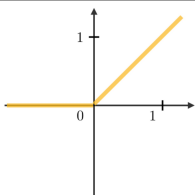
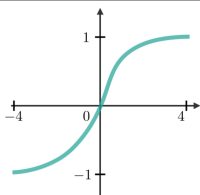
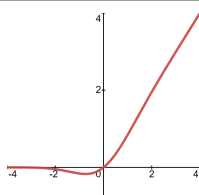
Sigmoid	ReLU	Tanh	GELU
$\frac{1}{1+e^{-z}}$	$\max(0, z)$	$\frac{e^z - e^{-z}}{e^z + e^{-z}}$	$z\Phi(z)$
			

Table 2: Equation and graph of activation functions

In the case of vector activations, however, each output element depends on each of the input elements. This makes calculating derivatives tricky. A popular vector activation is the **Softmax** which you will be implementing in addition to the scalar activations mentioned above.

In this section, your task is to implement the Activation class in file `activation.py`:

- Class attributes:
 - Activation functions have no trainable parameters.
 - Variables stored during forward-propagation to compute derivatives during back-propagation: layer output A .
- Class methods:
 - **forward**: forward method takes in a batch of data \mathbf{Z} of shape $N \times C$ (representing N samples where each sample has C features), and applies the activation function to \mathbf{Z} to compute output \mathbf{A} of shape $N \times C$.
 - **backward**: backward method takes in $dLdA$, a measure of how the post-activations (output) affect the loss. Using this and the derivative of the activation function itself, the method calculates and returns $dLdZ$, how changes in pre-activation features (input) Z affect the loss L . In the case of scalar activations, $dLdZ$ is computed as:

$$dLdZ = dLdA \odot \frac{\partial A}{\partial Z} \quad (8)$$

Here, $\frac{\partial A}{\partial Z}$ is the element wise derivative of A with respect to the corresponding element of Z . In other words, for one input of size $1 \times C$, it represents the diagonal of the Jacobian matrix in a

vector of size $1 \times C$ (recall from the lecture that the Jacobian of a scalar activation function is a diagonal matrix). For a batch of size N , the size of $\frac{\partial A}{\partial Z}$ is $N \times C$. $\frac{\partial A}{\partial Z}$ is calculated differently for different scalar activation functions as you'll see in the respective subsections.

The Jacobian of a vector activation function is not a diagonal matrix. For each input vector $Z^{(i)}$ ($1 \times C$) and corresponding output vector $A^{(i)}$ (also $1 \times C$) in the batch, you would calculate the Jacobian matrix $\mathbf{J}^{(i)}$ separately. This would be of size $C \times C$. Then, $dLdZ^{(i)}$ is given by:

$$dLdZ^{(i)} = dLdA^{(i)} \cdot \mathbf{J}^{(i)} \quad (9)$$

After calculating each of the $1 \times C$ $dLdZ^{(i)}$ vectors, you can stack them vertically to get the final $N \times C$ $dLdZ$ matrix to return.

Please consider the following class structure for the scalar activations:

```
class Activation:

    def forward(self, Z):
        self.A = # TODO

        return self.A

    def backward(self, dLdA):
        dAdZ = # TODO
        dLdZ = # TODO

        return dLdZ
```

Table 3: Activation Function Components

Code Name	Math	Type	Shape	Meaning
N	N	scalar	-	batch size
C	C	scalar	-	number of features
Z	Z	matrix	$N \times C$	batch of N inputs each represented by C features
A	A	matrix	$N \times C$	batch of N outputs each represented by C features
dLdA	$\partial L / \partial A$	matrix	$N \times C$	how changes in post-activation features affect loss
dLdZ	$\partial L / \partial Z$	matrix	$N \times C$	how changes in pre-activation features affect loss

The activation function topology is visualized in Figure C, revisit Figure A to see where it is in the bigger picture.

Note: By convention in this class, Z is the output of a linear layer, and A is the input of a linear layer. Here, Z is the output from the previous linear layer and A is the input to the next linear layer, i.e. let f_l be the activation function of layer l , $A_{l+1} = f_l(Z_l)$.

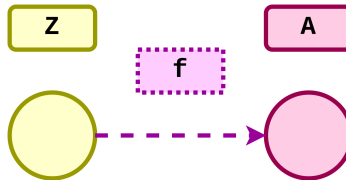


Figure C: Activation Function Topology

6.1 Sigmoid [mytorch.nn.Sigmoid]

6.1.1 Sigmoid Forward Equation

During forward propagation, pre-activation features \mathbf{Z} are passed to the activation function **Sigmoid** to calculate their post-activation values \mathbf{A} .

$$A = \text{sigmoid.forward}(Z) \quad (10)$$

$$= \varsigma(Z) \quad (11)$$

$$= \frac{1}{1 + e^{-Z}} \quad (12)$$

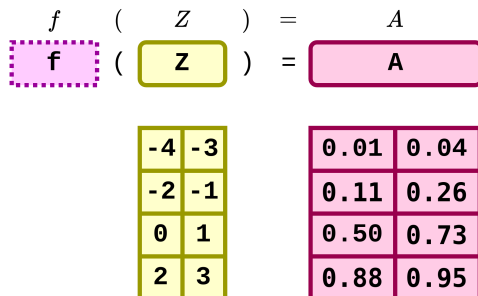


Figure D: Sigmoid Activation Forward Example

6.1.2 Sigmoid Backward Equation

Backward propagation helps us understand how changes in pre-activation features \mathbf{Z} affect the loss, given how changes in post-activation values \mathbf{A} affect the loss.

$$\frac{dL}{dZ} = \text{sigmoid.backward}(dLdA) \quad (13)$$

$$= dLdA \odot \frac{\partial A}{\partial Z} \quad (14)$$

$$= dLdA \odot (\varsigma(Z) - \varsigma^2(Z)) \quad (15)$$

$$= dLdA \odot (A - A \odot A) \quad (16)$$

6.2 Tanh [mytorch.nn.Tanh]

6.2.1 Tanh Forward Equation

$$A = \text{Tanh.forward}(Z) \quad (17)$$

$$= \tanh(Z) \quad (18)$$

$$= \frac{e^Z - e^{-Z}}{e^Z + e^{-Z}} \quad (19)$$

6.2.2 Tanh Backward Equation

Fill in the blank in the equation below. Represent the final result in terms of A and $dLdA$, similar to Sigmoid backward equation in the previous section.

$$\frac{dL}{dZ} = \text{tanh.backward}(dLdA) \quad (20)$$

$$= _? _ \quad (21)$$

Hint: $\tanh'(x) = 1 - \tanh^2(x)$.

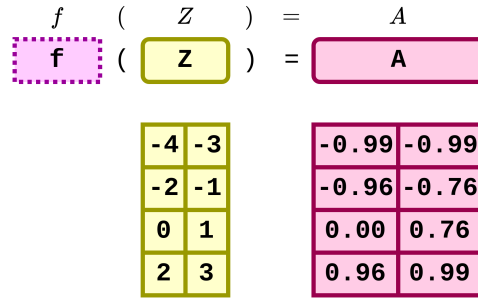


Figure E: Tanh Activation Forward Example

6.3 ReLU [mytorch.nn.ReLU]

6.3.1 ReLU Forward Equation

Recall the equation of ReLU and fill in the blank below:

$$A = \text{relu.forward}(Z) \quad (22)$$

$$= _? _ \quad (23)$$

Hint: You might find the graph of ReLU in Table 2 helpful.

Hint: For coding, search and read the docs on `np.amax`, `np.maximum`.

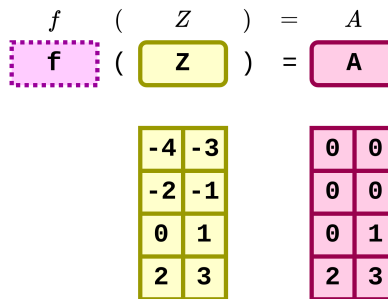


Figure F: ReLU Activation Forward Example

6.3.2 ReLU Backward Equation

Complete the piece-wise function for `relu.backward`:

$$\frac{dL}{dZ} = \text{relu.backward}(dLdA) \quad (24)$$

$$= \begin{cases} _? _, & A > 0 \\ _? _, & A \leq 0 \end{cases} \quad (25)$$

Hint: For coding, search and read the docs on `np.where`.

6.4 GELU [mytorch.nn.GELU]

6.4.1 GELU Forward Equation

The GELU (Gaussian Error Linear Unit) activation function is defined in terms of the cumulative distribution function of the standard Gaussian distribution $\Phi(Z) = \mathbb{P}(X \leq Z)$ where $X \sim \mathcal{N}(0, 1)$:

$$A = \text{gelu.forward}(Z) \quad (26)$$

$$= Z\Phi(Z) \quad (27)$$

$$= Z \int_{-\infty}^Z \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right) dx \quad (28)$$

$$= \frac{1}{2} Z \left[1 + \text{erf}\left(\frac{Z}{\sqrt{2}}\right) \right] \quad (29)$$

Here, *erf* refers to the **error function** which is frequently seen in probability and statistics. It can also take complex arguments but will take real ones here. **Hint:** Search the docs of the **math** and **scipy** libraries for help with implementation.

6.4.2 GELU Backward Equation

For the `gelu.backward` part where we calculate $\frac{\partial A}{\partial Z}$, the GELU equation given above needs to be differentiated with respect to Z :

$$\frac{dA}{dZ} = \frac{d}{dZ} Z\Phi(Z) \quad (30)$$

$$= \Phi(Z) + Z\Phi'(Z) \quad (31)$$

$$= \Phi(Z) + Z\mathbb{P}(X = Z) \quad (32)$$

$$= \frac{1}{2} \left[1 + \text{erf}\left(\frac{Z}{\sqrt{2}}\right) \right] + \frac{Z}{\sqrt{2\pi}} \exp\left(-\frac{Z^2}{2}\right) \quad (33)$$

This gives us the final expression to implement the **backward** function:

$$\frac{\partial L}{\partial Z} = \text{gelu.backward}(\text{dLdA}) \quad (34)$$

$$= \text{dLdA} \odot \frac{\partial A}{\partial Z} \quad (35)$$

$$= \text{dLdA} \odot \left[\frac{1}{2} \left(1 + \text{erf}\left(\frac{Z}{\sqrt{2}}\right) \right) + \frac{Z}{\sqrt{2\pi}} \exp\left(-\frac{Z^2}{2}\right) \right] \quad (36)$$

6.5 Softmax [mytorch.nn.Softmax]

The Softmax activation function is a vector activation function that is mostly applied at the end of neural network to convert a vector or raw outputs to a probability distribution in which the output elements sum up to 1. However, it can also be applied in the middle of a neural network like other activation functions discussed before this.

6.5.1 Softmax Forward Equation

Given a C -dimensional input vector Z , whose m -th element is denoted by z_m , `softmax.forward(Z)` will give a vector A whose m -th element a_m is given by:

$$a_m = \frac{\exp(z_m)}{\sum_{k=1}^C \exp(z_k)} \quad (37)$$

Here Z was a single vector. Similar calculations can be done for batch of N vectors.

6.5.2 Softmax Backward Equation

As discussed in the description of the `backward` method for vector activations earlier in the section, the first step in backpropagating the derivatives is to calculate the Jacobian for each vector in the batch. Let's take the example of an input vector Z (a row of the input data matrix) and corresponding output vector A (a row of the output matrix calculated by `softmax.forward`). The Jacobian \mathbf{J} is a $C \times C$ matrix. Its element at the m -th row and n -th column is given by:

$$\mathbf{J}_{mn} = \begin{cases} a_m(1 - a_m) & \text{if } m = n \\ -a_m a_n & \text{if } m \neq n \end{cases} \quad (38)$$

where a_m refers to the m -th element of the vector A .

Now the derivative of the loss with respect to this input vector, i.e., $dLdZ$ is $1 \times C$ vector and is calculated as:

$$dLdZ = dLdA \cdot \mathbf{J} \quad (39)$$

Similar derivative calculation can be done for all the N vectors in the batch and the resulting vectors can be stacked up vertically to give the final $N \times C$ derivatives matrix.

Some code hints for Softmax are given in the handout to help you with implementation.

7 Neural Network Models [35 points]

In this section, you will bring together the different components you have made so far – linear layers and activation functions – and create your own `Model Class` in file `models/mlp.py`!

- Class attributes:
 - **layers**: a list storing all linear and activation layers in the correct order.
- Class methods:
 - **forward**: forward method takes input data \mathbf{A}_0 and applies transformations corresponding to the layers (linear and activation) sequentially as `self.layers[i].forward` for $i = 0, \dots, l - 1$ ⁸ where l is the total number of layers, to compute output \mathbf{A}_l .
 - **backward**: backward method takes in $dLdA_l$, how changes in loss L affect model output A_l , and performs back-propagation from the last layer to the first layer by calling `self.layers[i].backward` for $i = l - 1, \dots, 0$. It does not return anything. Note that activation and linear layers don't need to be treated differently as both take in the derivative of the loss with respect to the layer's output and give back the derivative of the loss with respect to the layer's input.

Please consider the following class structure:

```
class Model:

    def __init__(self):
        self.layers = # TODO

    def forward(self, A):
        l = len(self.layers)
        for i in range(l):
            A = # TODO - keep modifying A by passing it through a layer

        return A

    def backward(self, dLdA):
        l = len(self.layers)
        for i in reversed(range(l)):
            dLdA = # TODO - keep modifying dLdA by passing it backwards through a layer

        return dLdA
```

Note that the A mentioned in the for loop in the **forward** pseudo code above is written so to maintain the same name of the variable containing the current output. In case of linear layers, it is the same as the output that was written as Z in the linear layer section. The case with $dLdA$ mentioned in the **backward** pseudo code is similar. In the case of activation functions, it will be the same as what was mentioned as $dLdZ$ in the activation functions section after the current $dLdA$ is passed through the activation layer's **backward** function.

We will start by building a shallow network with 0 hidden layer in subsection 7.1, and then a slightly deeper network with 1 hidden layer in subsection 7.2. Finally, we will build a deep neural network with 4 hidden layers in subsection 7.3. Note: all models have one additional layer for the output mapping, i.e. the total number of layers l for a model with 1 hidden layer is actually 2.

We do not provide a reference table here. Using what you have learned so far, we encourage you to make a reference table yourself. Though it takes time, it will aid the debugging process and help make clear your

⁸python lists are 0-indexed

understanding of the relevant components. If you ask for help, we will likely ask to see the reference table you have created before attempting to diagnose your issue.

7.1 MLP (Hidden Layers = 0) [mytorch.models.MLP0] [10 points]

In this subsection, your task is to implement the forward and backward attribute functions of the MLP0 class.

The MLP0 topology is visualized in Figure G. The network is displayed vertically to fit on the page. To facilitate understanding, you can try labelling the graph to show which parts are linear layers and which parts are activation functions⁹. The `layers` class attribute will contain a linear layer (`layer0`) followed by the activation layer `f0`.

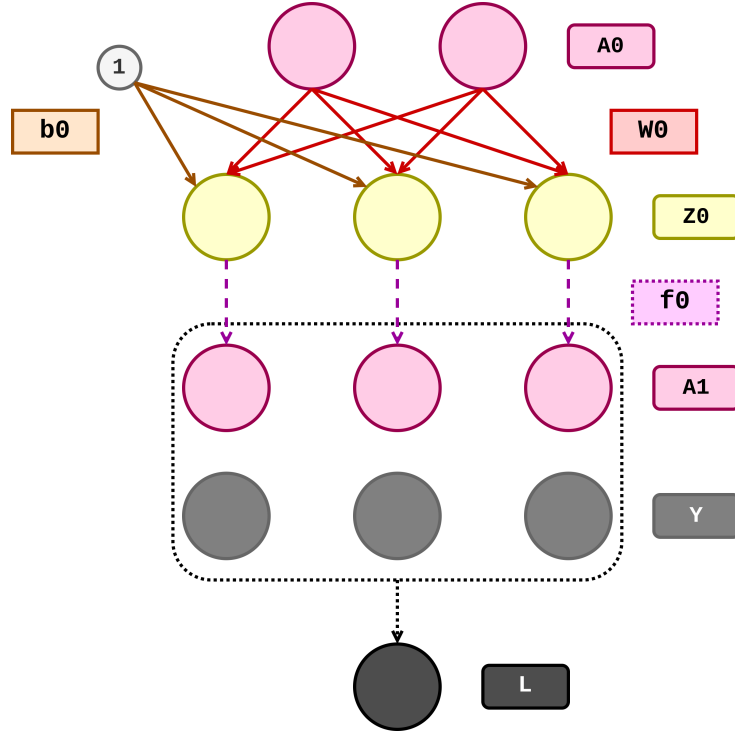


Figure G: MLP 0 Example Topology (Hidden Layers = 0)

7.1.1 MLP Forward Pseudocode (Hidden Layers = 0)

$$Z_0 = \text{layer0.forward}(A_0) \quad \in \mathbb{R}^{N \times C_1} \quad (40)$$

$$A_1 = \text{f0.forward}(Z_0) \quad \in \mathbb{R}^{N \times C_1} \quad (41)$$

7.1.2 MLP Backward Pseudocode (Hidden Layers = 0)

$$\frac{\partial L}{\partial Z_0} = \text{f0.backward} \left(\frac{\partial L}{\partial A_1} \right) \quad \in \mathbb{R}^{N \times C_1} \quad (42)$$

$$\frac{\partial L}{\partial A_0} = \text{layer0.backward} \left(\frac{\partial L}{\partial Z_0} \right) \quad \in \mathbb{R}^{N \times C_0} \quad (43)$$

$$(44)$$

⁹Refer to Fig A for solution

7.2 MLP (Hidden Layers = 1) [mytorch.models.MLP1] [10 points]

In this section, your task is to implement the forward and backward attribute functions of the MLP1 class.

The MLP1 topology is visualized in Figure H. You must use the diagram to deduce what the model specification is for the linear layers. To facilitate understanding, you should try labelling the graph to show which parts correspond to which linear layers and activation functions.

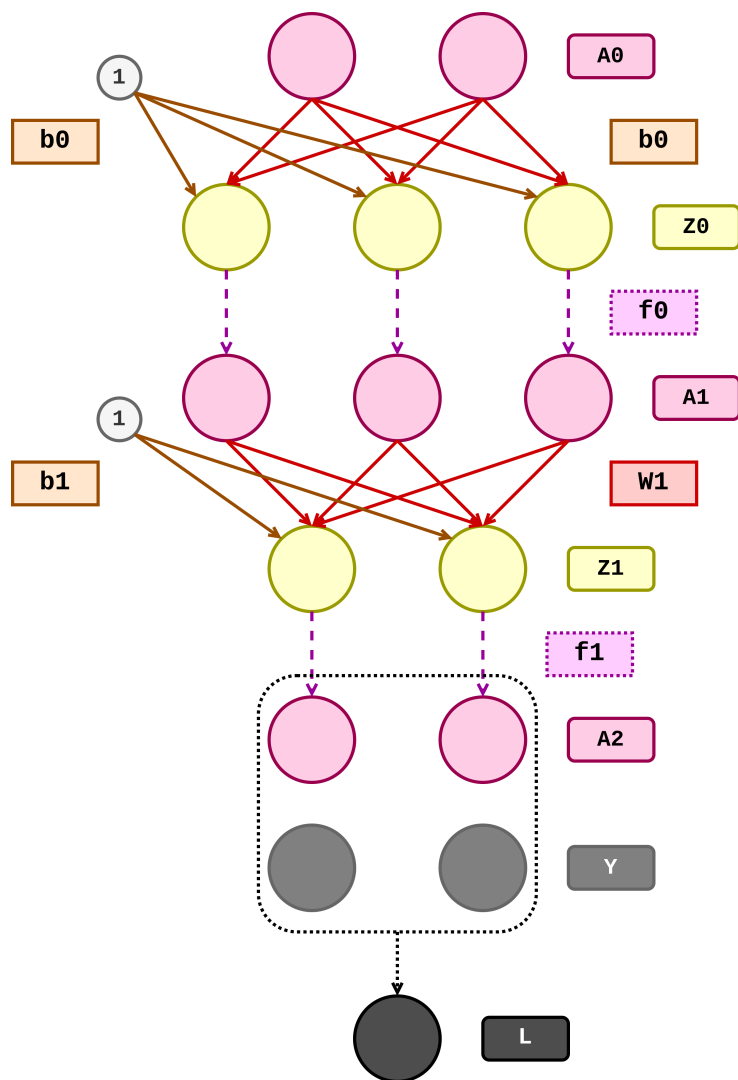


Figure H: MLP 1 Example Topology (Hidden Layers = 1)

7.2.1 MLP Forward Method Description (Hidden Layers = 1)

The code for `MLP1.forward()` is highly similar to `MLP0.forward()`, you are doing the same thing, except for two more layers (one linear and the corresponding activation). Hence, we won't provide you the pseudocode, but only a high level description with reference to Fig H:

- `forward` method takes input data A_0 and applies the linear transformation `self.layers[0].forward` to get Z_0 .
- It then applies activation function `self.layers[1].forward` on Z_0 to compute layer output A_1 .

- \mathbf{A}_1 is passed to the next linear layer, and we apply `self.layers[2].forward` to obtain \mathbf{Z}_1 .
- Finally, we apply activation function `self.layers[3].forward` on \mathbf{Z}_1 to compute model output \mathbf{A}_2 .

7.2.2 MLP Backward Method Descriptions (Hidden Layers = 1)

`backward`: backward method takes in $dLdA_2$, how changes in loss L affect model output A_2 , and performs back-propagation from the last layer to the first layer by calling `self.layers[i].backward` for $i = 3, 2, 1, 0$.

7.3 MLP (Hidden Layers = 4) [mytorch.models.MLP4] [15 points]

In this section, your task is to initialize the `MLP4` class and implement the forward and backward attribute functions.

The MLP4 topology is visualized in Figure I. You must use the diagram to deduce what the model specification is for the linear layers. To facilitate understanding, you can try labelling the graph to show which parts correspond to which linear layers and activation functions.

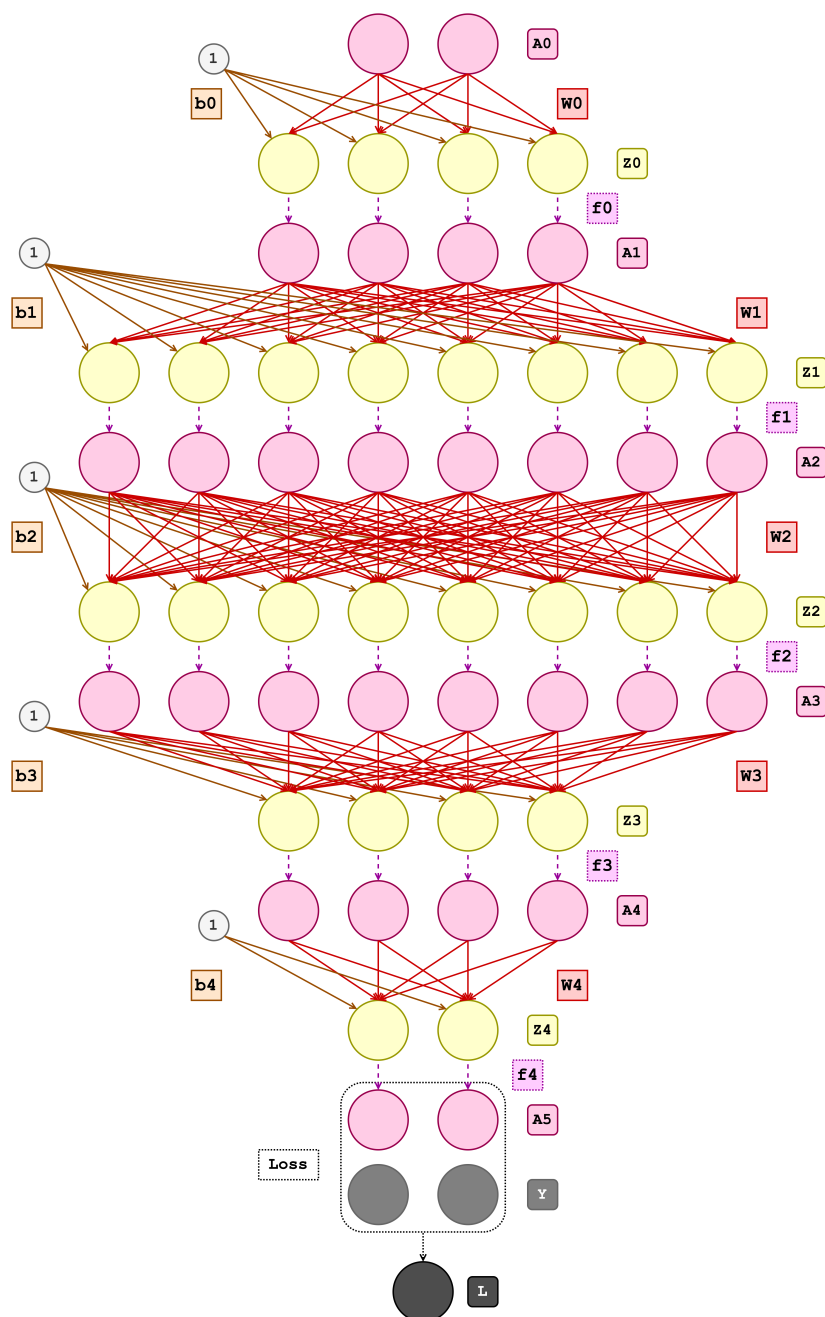


Figure I: MLP 4 Example Topology (Hidden Layers = 4)

7.3.1 MLP Forward Equations (Hidden Layers = 4)

Given the math equations, can you figure out which class methods of `Linear class` and `Activation class` perform the calculation of which equation?

$$Z_i = A_i \cdot W_i^T + \iota_N \cdot b_i^T \quad \in \mathbb{R}^{N \times C_{i+1}} \quad (45)$$

$$A_{i+1} = f_i(Z_i) \quad \in \mathbb{R}^{N \times C_{i+1}} \quad (46)$$

7.3.2 MLP Backward Equations (Hidden Layers = 4)

Given the math equations, can you figure out which class methods of `Linear class` and `Activation class` perform the calculations?

$$\frac{\partial A_{i+1}}{\partial Z_i} = \frac{\partial}{\partial Z_i} f_i(Z_i) \quad \in \mathbb{R}^{N \times C_{i+1}} \quad (47)$$

$$\frac{\partial L}{\partial Z_i} = \frac{\partial L}{\partial A_{i+1}} \odot \frac{\partial A_{i+1}}{\partial Z_i} \quad \in \mathbb{R}^{N \times C_{i+1}} \quad (48)$$

$$\frac{\partial L}{\partial A_i} = \frac{\partial L}{\partial Z_i} \cdot \left(\frac{\partial Z_i}{\partial A_i} \right)^T \quad \in \mathbb{R}^{N \times C_i} \quad (49)$$

8 Criterion - Loss Functions [10 points]

Much as you did for activation functions you will now program some simple loss functions. Different loss functions may become useful depending on the type of neural network and type of data you are using. Here we will program Mean Squared Error Loss **MSE** and **Cross Entropy Loss**. It is important to know how these are calculated, and how they will be used to update your network. As before we will provide the formulas, and know that each of these functions can be done in less than 10 lines of code, so if your code begins to get more complex than that you may be overthinking the problem.

In this section, your task is to implement the forward and backward attribute functions of the `Loss` class in file `loss.py`:

- Class attributes:
 - Stores model prediction **A** to compute back-propagation.
 - Stores desired output **Y** stored to compute back-propagation.
- Class methods:
 - **forward**: forward method takes in model prediction **A** and desired output **Y** of the same shape to calculate and return a loss value **L**. The loss value is a **scalar quantity** used to quantify the mismatch between the network output and the desired output.
 - **backward**: backward method calculates and returns **dLdA**, how changes in model outputs **A** affect loss **L**. It is used to enable downstream computation, as seen in previous sections.

Please consider the following class structure:

```
class Loss:

    def forward(self, A, Y):
        self.A = A
        self.Y = Y
        self.    # TODO (store additional attributes as needed)
        N        = # TODO,  this is the first dimension of A and Y
        C        = # TODO,  this is the second dimension of A and Y
        # TODO

        return L

    def backward(self):
        dLdA = # TODO

        return dLdA
```

Table 4: Loss Function Components

Code Name	Math	Type	Shape	Meaning
N	N	scalar	-	batch size
C	C	scalar	-	number of classes
A	A	matrix	$N \times C$	model outputs
Y	Y	matrix	$N \times C$	ground-truth values
L	L	scalar	-	loss value
dLdA	$\partial L / \partial A$	matrix	$N \times C$	how changes in model outputs affect loss

The loss function topology is visualized in Figure J, whose reference persists throughout this document.

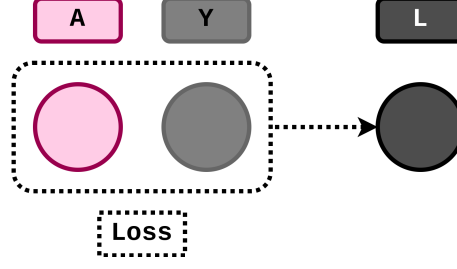


Figure J: Loss Function Topology

8.1 MSE Loss [`mytorch.nn.MSELoss`]

MSE stands for Mean Squared Error, and is often used to quantify the prediction error for regression problems. Regression is a problem of predicting a real-valued label given an unlabeled example. Estimating house price based on features such as area, location, the number of bedrooms and so on is a classic regression problem.

8.1.1 MSE Loss Forward Equation

We first calculate the squared error **SE** between the model outputs **A** and the ground-truth values **Y**:

$$SE(A, Y) = (A - Y) \odot (A - Y) \quad (50)$$

Then we calculate the sum of the squared error **SSE**, where ι_N, ι_C are column vectors of size N and C which contain all 1s:

$$SSE(A, Y) = \iota_N^T \cdot SE(A, Y) \cdot \iota_C \quad (51)$$

Here, we are calculating the sum of all elements of the $N \times C$ matrix $SE(A, Y)$. The first pre multiplication with ι_N^T sums across rows. Then, the post multiplication of this product with ι_C sums the row sums across columns to give the final sum as a single number.

Lastly, we calculate the per-component Mean Squared Error **MSE** loss:

$$MSELoss(A, Y) = \frac{SSE(A, Y)}{N \cdot C} \quad (52)$$

8.1.2 MSE Loss Backward Equation

$$\text{MSELoss.backward}() = 2 \cdot \frac{A - Y}{N \cdot C} \quad (53)$$

8.2 Cross-Entropy Loss [mytorch.nn.CrossEntropyLoss]

Cross-entropy loss is one of the most commonly used loss function for probability-based classification problems. In this course, most of the part 2 homework problems involve classification problems, hence you will use this loss function very often.

8.2.1 Cross-Entropy Loss Forward Equation

Firstly, we use softmax function to transform the raw model outputs A into a probability distribution consisting of C classes proportional to the exponentials of the input numbers.

ι_N, ι_C are column vectors of size N and C which contain all 1s. ¹⁰

$$\text{softmax}(A) = \sigma(A) \quad (54)$$

$$= \frac{\exp(A)}{\sum_{j=1}^C \exp(A_{ij})} \quad (55)$$

Now, each row of A represents the model's prediction of the probability distribution while each row of Y represents target distribution of an input in the batch.

Then, we calculate the cross-entropy $H(A, Y)$ of the distribution A_i relative to the target distribution Y_i for $i = 1, \dots, N$:

$$\text{crossentropy} = H(A, Y) \quad (56)$$

$$= (-Y \odot \log(\sigma(A))) \cdot \iota_C \quad (57)$$

Remember that the output of a loss function is a scalar, but now we have a column matrix of size N . To transform it into a scalar, we can either use the sum or mean of all cross-entropy.

Here, we choose to use the mean cross-entropy as the cross-entropy loss as that is the default for PyTorch as well:

$$\text{sum_crossentropy_loss} := \iota_N^T \cdot H(A, Y) \quad (58)$$

$$= SCE(A, Y) \quad (59)$$

$$\text{mean_crossentropy_loss} := \frac{SCE(A, Y)}{N} \quad (60)$$

$$\begin{array}{l} \text{Loss} (A, Y) = L \\ \boxed{\text{Loss}} (A, Y) = L \end{array}$$

-4	-3	0	1	0.813
-2	-1	1	0	
0	1	1	0	
2	3	0	1	

Figure K: Cross Entropy Loss Example

8.2.2 Cross-Entropy Loss Backward Equation

$$\text{xent.backward}() = \frac{\sigma(A) - Y}{N} \quad (61)$$

¹⁰The matrix division in Equation 55 is element-wise (the formal symbol for the element-wise division operator of two matrices is \oslash , but we use the simpler A over B notation here).

9 Optimizers [10 points]

In deep learning, optimizers are used to adjust the parameters for a model. The purpose of an optimizer is to adjust model weights to maximize a loss function.

To recap, we built our own MLP models in Section 7 using `linear class` we built in Section 5 and `activation classes` we built in Section 6 and have seen how to do forward propagation, and backward propagation for the core components used in neural networks. Forward propagation is used for estimation, and backward propagation informs us on how changes in parameters affect loss. And in Section 8, we coded some loss functions, which are criterion we use to evaluate the quality of our model's estimates. The last step is to improve our model using the information we learned on how changes in parameters affect loss.

9.1 Stochastic Gradient Descent (SGD) [`mytorch.optim.SGD`]

In this section, we are going to implement **Minibatch stochastic gradient descent** with momentum, which we will refer to as **SGD** in this homework. Minibatch SGD is a version of SGD algorithm that speeds up the computation by approximating the gradient using smaller batches of the training data, and Momentum is a method that helps accelerate SGD by incorporating velocity from the previous update to reduce oscillations. The `sgd` function in PyTorch library is actually an implementation of Minibatch stochastic gradient descent with momentum.

Your task is to implement the step attribute function of the SGD class in file `sgd.py`:

- Class attributes:
 - `l`: list of model layers
 - `L`: number of model layers
 - `lr`: learning rate, tunable hyperparameter scaling the size of an update.
 - `mu`: momentum rate μ , tunable hyperparameter controlling how much the previous updates affect the direction of current update. $\mu = 0$ means no momentum.
 - `v_W`: list of weight velocity for each layer
 - `v_b`: list of bias velocity for each layer
- Class methods:
 - `step`: Updates `W` and `b` of each of the model layers:
 - * Because parameter gradients tell us which direction makes the model worse, we move opposite the direction of the gradient to update parameters.
 - * When momentum is non-zero, update velocities `v_W` and `v_b`, which are changes in the gradient to get to the global minima. The velocity of the previous update is scaled by hyperparameter μ , refer to lecture slides for more details.

Please consider the following class structure:

```
class SGD:

    def __init__(self, model, lr=0.1, momentum=0):
        self.l = model.layers
        self.L = len(model.layers)
        self.lr = lr
        self.mu = momentum
        self.v_W = [np.zeros(self.l[i].W.shape) for i in range(self.L)]
        self.v_b = [np.zeros(self.l[i].b.shape) for i in range(self.L)]
```

```

def step(self):
    for i in range(self.L):
        if self.mu == 0:
            self.l[i].W = # TODO
            self.l[i].b = # TODO

        else:
            self.v_W[i] = # TODO
            self.v_b[i] = # TODO
            self.l[i].W = # TODO
            self.l[i].b = # TODO

```

Table 5: SGD Optimizer Components

Code Name	Math	Type	Shape	Meaning
model	-	object	-	model with layers attribute
l	-	object	-	layers attribute selected from the model
L	L	scalar	-	number of layers in the model
lr	λ	scalar	-	learning rate hyperparameter to scale affect of new gradients
momentum	μ	scalar	-	momentum hyperparameter to scale affect of prior gradients
v_W	-	list	L	list of velocity weight parameters, one for each layer
v_b	-	list	L	list of velocity bias parameters, one for each layer
v_W[i]	v_{W_i}	matrix	$C_{i+1} \times C_i$	velocity for layer i weight
v_b[i]	v_{b_i}	matrix	$C_{i+1} \times 1$	velocity for layer i bias
l[i].W	W_i	matrix	$C_{i+1} \times C_i$	weight parameter for a layer
l[i].b	b_i	matrix	$C_{i+1} \times 1$	bias parameter for a layer

9.1.1 SGD Equation (Without Momentum)

$$W := W - \lambda \frac{\partial L}{\partial W} \quad (62)$$

$$b := b - \lambda \frac{\partial L}{\partial b} \quad (63)$$

9.1.2 SGD Equations (With Momentum)

$$v_W := \mu v_W + \frac{\partial L}{\partial W} \quad (64)$$

$$v_b := \mu v_b + \frac{\partial L}{\partial b} \quad (65)$$

$$W := W - \lambda v_W \quad (66)$$

$$b := b - \lambda v_b \quad (67)$$

10 Regularization [20 points]

Regularization is a set of techniques that can prevent overfitting in neural networks and thus improve the accuracy of a Deep Learning model when facing completely new data from the problem domain.

10.1 Batch Normalization [mytorch.nn.BatchNorm1d]

Z-score normalization is the procedure during which the feature values are rescaled so that they have the properties of a **normal distribution**. Let μ be the mean (the average value of the feature, averaged over all examples in the dataset) and σ be the standard deviation from the mean.

Standard scores (or z-scores) of features are calculated as follows:

$$\hat{x} = \frac{x - \mu}{\sigma} \quad (68)$$

Batch normalization is a method used to make training of artificial neural networks faster and more stable through normalization of the layers' inputs by re-centering and re-scaling. It comes from the paper [Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift](#), we encourage you to read the paper for a better understanding. You can find pseudocode and explanation in the paper if you are stuck!

In this section, your task is to implement the forward and backward attribute functions of the BatchNorm1d class in file `batchnorm.py`.

- Class attributes:
 - **alpha**: a hyperparameter used for the running mean and running var computation.
 - **eps**: a value added to the denominator for numerical stability.
 - **BW**: learnable parameter of a BN (batch norm) layer to scale features.
 - **Bb**: learnable parameter of a BN (batch norm) layer to shift features.
 - **dLdBW**: how changes in γ affect loss
 - **dLdBb**: how changes in β affect loss
 - **running_M**: learnable parameter, the estimated mean of the training data
 - **running_V**: learnable parameter, the estimated variance of the training data
- Class methods:
 - **forward**: It takes in a batch of data Z computes the batch normalized data \hat{Z} , and returns the scaled and shifted data \tilde{Z} . In addition:
 - * During training, **forward** calculates the mean and standard-deviation of each feature over the mini-batches and uses them to update the **running_M** $E[Z]$ and **running_V** $Var[Z]$, which are learnable parameter vectors trained during forward propagation. By default, the elements of $E[Z]$ are set to 0 and the elements of $Var[Z]$ are set to 1.
 - * During inference, the learnt mean **running_M** $E[Z]$ and variance **running_V** $Var[Z]$ over the entire training dataset are used to normalize Z .
 - **backward**: takes input $dLdBZ$, how changes in BN layer output affects loss, computes and stores the necessary gradients $dLdBW$, $dLdBb$ to train learnable parameters BW and Bb . Returns $dLdZ$, how the changes in BN layer input Z affect loss L for downstream computation.

Please consider the following class structure:

```

class BatchNorm1d:

    def __init__(self, num_features, alpha=0.9):

        self.alpha      = alpha
        self.eps         = 1e-8

        self.BW          = np.ones((1, num_features))
        self.Bb          = np.zeros((1, num_features))
        self.dLdBW       = np.zeros((1, num_features))
        self.dLdBb       = np.zeros((1, num_features))

        self.running_M   = np.zeros((1, num_features))
        self.running_V   = np.ones((1, num_features))

    def forward(self, Z, eval=False):
        """
        The eval parameter is to indicate whether we are in the
        training phase of the problem or the inference phase.
        So see what values you need to recompute when eval is False.
        """
        if eval==False:
            # training mode
            self.Z          = Z
            self.N          = None # TODO

            self.M          = None # TODO
            self.V          = None # TODO
            self.NZ         = None # TODO
            self.BZ         = None # TODO

            self.running_M  = None # TODO
            self.running_V  = None # TODO
        else:
            # inference mode
            self.NZ         = None # TODO
            self.BZ         = None # TODO

        return self.BZ

    def backward(self, dLdBZ):

        self.dLdBW        = None # TODO
        self.dLdBb        = None # TODO

        dLdNZ             = None # TODO
        dLdV               = None # TODO
        dLdM               = None # TODO

        dLdZ              = None # TODO

        return dLdZ

```


Hint: check the documentation for `np.sum` and apply it along the right axis.

$$\mu_j = \frac{1}{N} \sum_{i=1}^N Z_{ij} \quad j = 1, \dots, C \quad (69)$$

$$\sigma_j^2 = \frac{1}{N} \sum_{i=1}^N (Z_{ij} - \mu_j)^2 \quad j = 1, \dots, C \quad (70)$$

Using the mean and variance, we normalize the input Z to get the normalized data \hat{Z} . Note: we add ϵ in denominator for numerical stability and to prevent division by 0 error .

$$\hat{Z}_i = \frac{Z_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \quad i = 1, \dots, N \quad (71)$$

Here, we give you an example for the above equation to facilitate understanding:

$$\left(\begin{matrix} Z_0 & - & \iota & \cdot & \mu_0 \end{matrix} \right) \div \left(\begin{matrix} \iota & \cdot & \sqrt{\sigma_0^2 + \epsilon} \end{matrix} \right) = \hat{Z}_0$$

$$\left(\begin{matrix} \text{Z0} & - & 1 & \cdot & \text{M0} \end{matrix} \right) \div \left(\begin{matrix} 1 & \cdot & \text{V0}^{-1/2} \end{matrix} \right) = \text{NZ0}$$

Figure M: Batchnorm Forward Equation 1 Example

Scale the normalized data by γ and shift it by β :

$$\tilde{Z}_i = \gamma \odot \hat{Z}_i + \beta \quad i = 1, \dots, N \quad (72)$$

Hint: In your matrix equation, first broadcast γ and β to make them have the same shape $N \times C$ as \hat{Z} .

$$\hat{Z}_0 \times \left(\begin{matrix} \iota & \cdot & \gamma_0 \end{matrix} \right) + \begin{matrix} \iota & \cdot & \beta_0 \end{matrix} = \tilde{Z}_0$$

$$\begin{matrix} \text{NZ0} \end{matrix} \times \left(\begin{matrix} 1 & \cdot & \text{BW0} \end{matrix} \right) + \begin{matrix} 1 & \cdot & \text{Bb0} \end{matrix} = \text{BZ0}$$

Figure N: Batchnorm Forward Equation 2 Example

During training (and only during training), your forward method should be maintaining a **running average** of the mini-batch mean and variance. These running averages should be used during inference. Hyperparameter α is used to compute weighted running averages.

$$E[Z] = \alpha * E[Z] + (1 - \alpha) * \mu \quad \in \mathbb{R}^{1 \times C} \quad (73)$$

$$Var[Z] = \alpha * Var[Z] + (1 - \alpha) * \sigma^2 \quad \in \mathbb{R}^{1 \times C} \quad (74)$$

10.1.2 Batch Normalization Forward Inference Equations (When `eval = True`)

Once the network has been trained, we use the population statistics $E[Z]$ and $Var[Z]$ to calculate the normalized data \hat{Z} .

$$\hat{Z}_i = \frac{Z_i - E[Z]}{\sqrt{Var[Z] + \epsilon}} \quad i = 1, \dots, N \quad (75)$$

Scale the normalized data by γ and shift it by β :

$$\tilde{Z}_i = \gamma \odot \hat{Z}_i + \beta \quad i = 1, \dots, N \quad (76)$$

10.1.3 Batch Normalization Backward Equations

We can now derive the analytic partial derivatives of the BatchNorm transformation. Let L be the training loss over the batch and $\frac{\partial L}{\partial \tilde{Z}}$ the derivative of the loss with respect to the output of the BatchNorm transformation for Z .

$$\left(\frac{\partial L}{\partial \beta} \right)_j = \sum_{i=1}^N \left(\frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \beta} \right)_{ij} = \sum_{i=1}^N \left(\frac{\partial L}{\partial \tilde{Z}} \right)_{ij} \quad j = 1, \dots, C \quad (77)$$

$$\left(\frac{\partial L}{\partial \gamma} \right)_j = \sum_{i=1}^N \left(\frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \gamma} \right)_{ij} = \sum_{i=1}^N \left(\frac{\partial L}{\partial \tilde{Z}} \odot \hat{Z} \right)_{ij} \quad j = 1, \dots, C \quad (78)$$

$$\frac{\partial L}{\partial \hat{Z}} = \frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \hat{Z}} = \frac{\partial L}{\partial \tilde{Z}} \odot \gamma \quad (79)$$

$$\left(\frac{\partial L}{\partial \sigma^2} \right)_j = \sum_{i=1}^N \left(\frac{\partial L}{\partial \hat{Z}} \frac{\partial \hat{Z}}{\partial \sigma^2} \right)_{ij} \quad j = 1, \dots, C \quad (80)$$

$$= -\frac{1}{2} \sum_{i=1}^N \left(\frac{\partial L}{\partial \hat{Z}} \odot (Z - \mu) \odot (\sigma^2 + \epsilon)^{-\frac{3}{2}} \right)_{ij} \quad (81)$$

$$\frac{\partial \hat{Z}_i}{\partial \mu} = \frac{\partial}{\partial \mu} \left[(Z_i - \mu)(\sigma^2 + \epsilon)^{-\frac{1}{2}} \right] \quad i = 1, \dots, N \quad (82)$$

$$= -(\sigma^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2} (Z_i - \mu) \odot (\sigma^2 + \epsilon)^{-\frac{3}{2}} \left(-\frac{2}{N} \sum_{i=1}^N (Z_i - \mu) \right) \quad (83)$$

$$\frac{\partial L}{\partial \mu} = \sum_{i=1}^N \frac{\partial L}{\partial \hat{Z}_i} \frac{\partial \hat{Z}_i}{\partial \mu} \quad (84)$$

Now for the grand finale, let's compute $\frac{\partial L}{\partial Z}$. For clarity, we present the derivation for $\frac{\partial L}{\partial Z_i}$ for one data sample Z_i .

$$\frac{\partial L}{\partial Z_i} = \frac{\partial L}{\partial \hat{Z}_i} \frac{\partial \hat{Z}}{\partial Z_i} = \frac{\partial L}{\partial \hat{Z}_i} \left[(\sigma^2 + \epsilon)^{-\frac{1}{2}} \right] + \frac{\partial L}{\partial \sigma^2} \left[\frac{2}{N} (Z_i - \mu) \right] + \frac{1}{N} \frac{\partial L}{\partial \mu} \quad (85)$$

In figure O, we present you with the illustration of batchnorm in a 0-hidden layer MLP model. Since the variables are color coded, it should be very clear each variable is used in which equations, which will help you apply chain rule and understand where the backward equations come from.

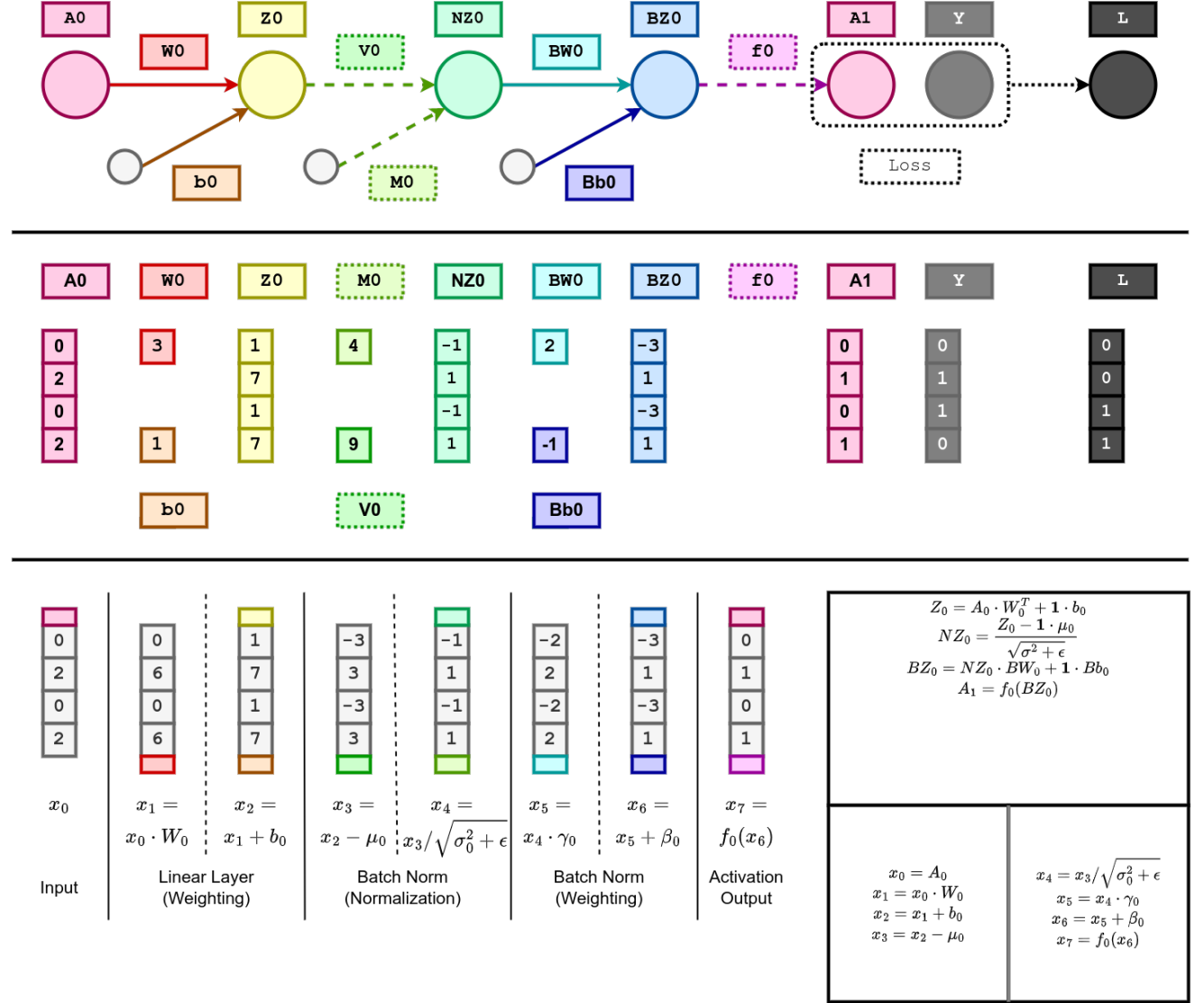


Figure O: Batchnorm Example (big picture)

11 Appendix

11.1 Anaconda Installation and Setup Instructions

1. Download the Anaconda installer specific to your operating system ¹¹:

- [Windows Installer](#)
- [macOS Installer](#)
- [Linux Installer](#)

2. Once the installation is complete, open the Anaconda Prompt or the terminal in Linux and macOS.

- Windows: Click **Start**, Search for Anaconda Prompt, and click to open.
- macOS: Open the Terminal application. You can find it by going to “**Applications**” < “**Utilities**” < “**Terminal**”.
- Linux: Open the **Dash** by clicking the Ubuntu icon, then type “**Terminal**”.

3. In the Anaconda Prompt, use the `cd` command to navigate to the directory where you have the “HW1P1” directory. For example:

```
cd /path/to/HW1P1
```

4. Create a new Anaconda environment named “idl23” with Python version 3.8 by running the following command:

```
conda create -n idl23 python=3.8
```

You may be prompted to answer “y” for a couple of prompts. Respond accordingly.

5. Activate the newly created “idl23” environment using the following command:

```
conda activate idl23
```

6. Install the required packages listed in the “**requirements.txt**” file:

```
pip install -r requirements.txt
```

¹¹If you are using a non-linux system and have issue installing the exact same version of the packages, it is fine to install a slightly newer/older version that is compatible with your OS. In case your codes give you full mark on your local machine and raises an issue on autolab, read autolab’s feedback to figure out which functions are not supported by autolab and replace them.