جامعة محمد بن زايد للذكاء الاصطناعي
MOHAMED BIN ZAYED UNIVERSITY OF ARTIFICIAL INTELLIGENCE

AI702: Deep Learning
Spring 2023

Homework-1
An Introduction to Neural Networks

`Release Date: Jan 30, 2023`
`Due Date: Feb 20, 2023 (23:59 GST)`

`Version: 1.0.0`

- **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.
  - 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 submitted code will be compared against 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 the 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 through all of the problems before attempting the first problem. However we do recommend you complete the problems in order, as the difficulty increases, and questions often rely on the completion of previous questions.

Homework Objectives

If you complete this homework successfully, you would ideally have learned:

- How to write code to implement an MLP from scratch
  - How to implement linear layers
  - How to implement various activation functions
  - How to implement batch norm
  - How to chain these up to compose an MLP of any size
- Your code will be able to perform forward inference [1] through the MLP
- How to write code to implement *training* of your MLP
  - How to perform a forward pass through your network
  - How to implement Mean Squared Error Loss and Cross-Entropy Loss functions
  - How to compute loss derivatives for the network parameters (including weights, biases and batch norm parameters)
  - How to implement backpropagation through the linear and activation layers
  - How to implement the Stochastic Gradient Descent (SGD) optimizer

Homework Preparation Acknowledgement and Credit:
Prof. Bhiksha Raj (CMU, MBZUAI)
Dr. Muhammad Haris (MBZUAI)

[1] Machine learning inference is the process of running data points into a machine learning model to calculate an output such as a single numerical score.

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
2. Complete the Components of a Multilayer Perceptron Model *train_model()*
   - [ ] Complete the linear layer class
   - [ ] Complete the 3 activation functions
3. Complete 3 Multilayer Perceptron Models using Components Built *evaluate_model()*
   - [ ] 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
   - [ ] 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
   - [ ] Implement SGD optimizer
6. Implement a Regularization method: 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

Contents

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 these work, you will understand forward propagation, loss calculation, backward propagation, and gradient descent.

The culmination of all of the Homework-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 modules of code for your subsequent homeworks.

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 autograder 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 `ai702_spring2023_hw1.tar` by running the following command in the same directory

  ```
  tar -xvf ai702_spring2023_hw1.tar
  ```

  This will create a directory called `ai702_spring2023_hw1` with the following file structure:

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

- **Install** Python3, NumPy and PyTorch in order to run the local autograder on your machine:

  ```
  pip3 install -r requirements.txt
  ```

- **Autograde** your code by
  - Step 1 (**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 2: Running local autograder by: Confirm that you are in the top level directory and execute the following in terminal:

    ```
    python hw1p1_autograder.py
    ```

- **Hand-in** your code by running the following command from the top level directory, then **SUBMIT** the created *handin.tar* file to autolab[2]:

  ```
  sh create_tarball.sh
  ```

- **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[3]. Libraries like `PyTorch, TensorFlow, Keras` are not allowed.
  - Add, move, or remove any files or change any filenames.
- **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. Make sure you get full points on the local autograder for any section, before submitting it to autolab.

[2]If you are a Windows user, run "sh.exe create_tarball.sh" in the terminal

[3]We 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`. Remove other libraries when making the submission.

3 Notation

Numpy Tips:

- Use `A * B` for element-wise multiplication $A \odot B$.
- Use `A @ B` for m**AT**rix 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
$A \cdot B$	Matrix multiplication of A and B
$A \oslash B$	Element-wise division of A and B

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, $\frac{e^x - e^{-x}}{e^x + e^{-x}}$
$\max\limits_{x \in \mathbb{X}} f(x)$	The operator $\max\limits_{x \in \mathbb{X}} f(x)$ returns the highest value $f(x)$ for all elements in the set $\mathbb{X}$
$\operatorname*{argmax}\limits_{x \in \mathbb{X}} f(x)$	The operator $\operatorname*{argmax}\limits_{x \in \mathbb{X}} f(x)$ returns the element of the set $\mathbb{X}$ that maximizes $f(x)$
$\sigma(x)$	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 $i = 1, ..., K$

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 scaler, 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**[4] depending on the task.

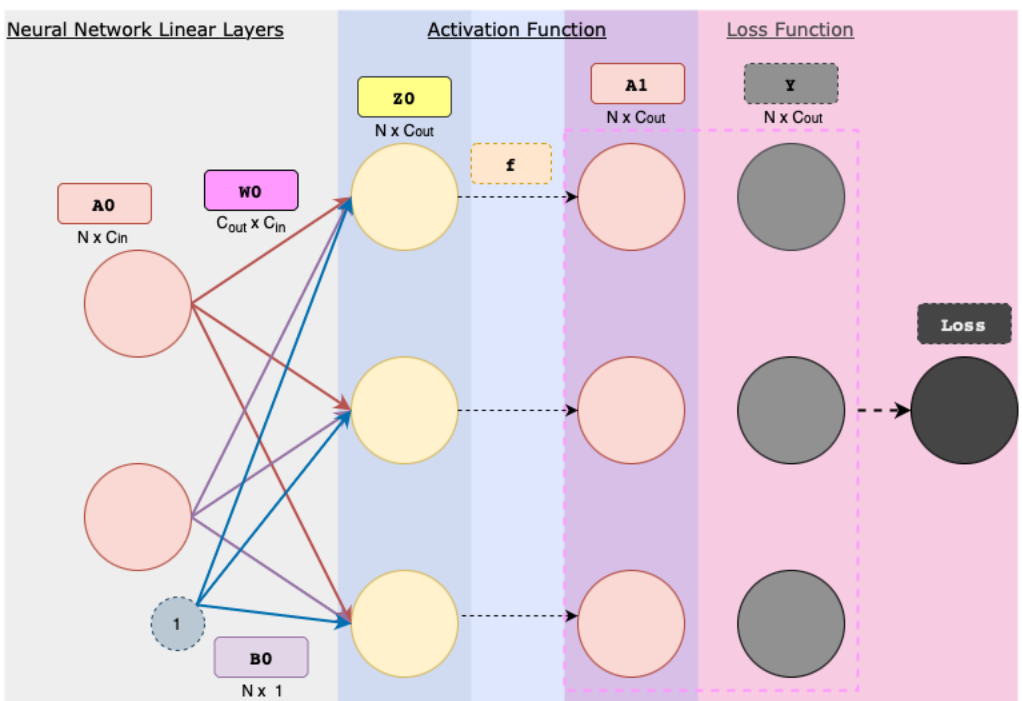


Figure A: End to end topology

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

[4]The 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 to see 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.
  - Variables stored during backward-propogation 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 $\mathbf{A}$ of shape $N \times C_{in}$ (representing N samples where each sample has C_{in} features), and computes output $\mathbf{Z}$ 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):
        dZdA      = # TODO
        dZdW      = # TODO
        dZdb      = # TODO
        dLdA      = # TODO
        dLdW      = # TODO
        dLdb      = # TODO
        self.dLdW = dLdW / self.N
        self.dLdb = dLdb / self.N

        return dLdA
```

Table 1: Linear Layer Components

Code Name	Math	Type	Shape	Meaning
`N`	N	scalar	-	batch size
`in_features`	C_{in}	scalar	-	number of input features
`out_features`	C_{out}	scalar	-	number of output features
`A`	A	matrix	$N \times C_{in}$	batch of N inputs each represented by C_{in} features
`Z`	Z	matrix	$N \times C_{out}$	batch of N outputs each represented by C_{out} features
`W`	W	matrix	$C_{out} \times C_{in}$	weight parameters
`b`	b	matrix	$C_{out} \times 1$	bias parameters
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_{out}$	how changes in outputs affect loss
`dZdA`	$\partial Z/\partial A$	matrix	$C_{in} \times C_{out}$	how changes in inputs affect outputs
`dZdW`	$\partial Z/\partial W$	matrix	$N \times C_{in}$	how changes in weights affect outputs
`dZdb`	$\partial Z/\partial b$	matrix	$N \times 1$	how changes in bias affect outputs
`dLdA`	$\partial L/\partial A$	matrix	$N \times C_{in}$	how changes in inputs affect loss
`dLdW`	$\partial L/\partial W$	matrix	$C_{out} \times C_{in}$	how changes in weights affect loss
`dLdb`	$\partial L/\partial b$	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 broadcasting[5] the bias.

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

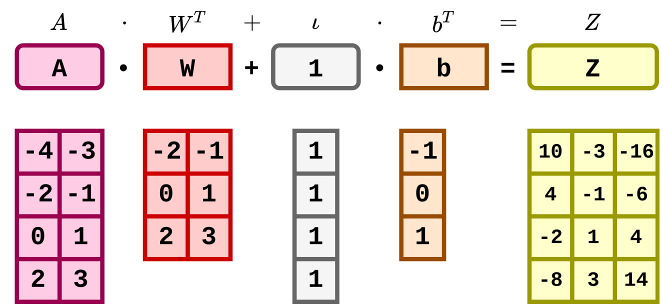


Figure B: Linear Layer Forward Example

5.1.2 Linear Layer Backward Equation

To implement backward propagation, we use the following rules:

For any linear equation of the kind $Z = A \cdot X + c$, the derivative of Z with respect to A is X. The derivative of Z with respect to X is A^T. (We will explain the rationale behind this in class). Also, the derivative with respect to a transpose is the transpose of the derivative, so the derivative of Z with respect to X is A^T but the derivative of Z with respect to X^T is A.

Applying this logic to the linear forward equation $Z = A \cdot W^T + \iota \cdot b^T$, fill in the blanks below:

$$\frac{\partial Z}{\partial A} = ? \qquad \frac{\partial Z}{\partial W} = ? \qquad \frac{\partial Z}{\partial b} = ? \tag{2}$$

[5]Read numpy documentation if you have never seen the word broadcasting before. We will refer to this term frequently in future homeowrk.

Then, 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:

$$\frac{\partial L}{\partial A} = \left(\frac{\partial L}{\partial Z}\right) \cdot \left(\frac{\partial Z}{\partial A}\right)^T \qquad \in \mathbb{R}^{N \times C_{in}} \tag{3}$$

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial Z}\right)^T \cdot \left(\frac{\partial Z}{\partial W}\right) \qquad \in \mathbb{R}^{C_{out} \times C_{in}} \tag{4}$$

$$\frac{\partial L}{\partial b} = \left(\frac{\partial L}{\partial Z}\right)^T \cdot \left(\frac{\partial Z}{\partial b}\right) \qquad \in \mathbb{R}^{C_{out} \times 1} \tag{5}$$

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.

Popular choices of activation functions are **Sigmoid**, as well as **ReLU** and **Tanh**, as shown in Table 2:

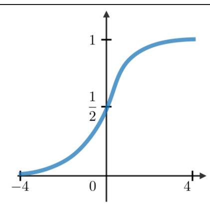
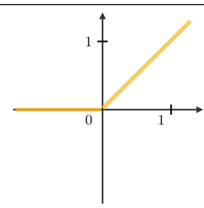
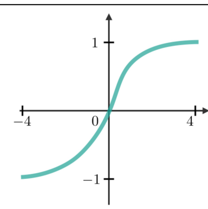
Sigmoid	ReLU	Tanh
$\frac{1}{1+e^{-z}}$	$\max(0, z)$	$\frac{e^z - e^{-z}}{e^z + e^{-z}}$
		

Table 2: Equation and graph of activation functions

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 each element of Z to compute output $\mathbf{A}$ of shape $N \times C$.
  - `backward`: backward method calculates and returns `dAdZ`, how changes in pre-activation features Z affect post-activation values A. It is used to enable downstream computation, as seen in subsequent sections.

Please consider the following class structure:

```
class Activation:

    def forward(self, Z):

        self.A = # TODO

        return self.A

    def backward(self):

        dAdZ = # TODO
```

```
        return dAdZ
```

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
`dAdZ`	$\partial A/\partial Z$	matrix	$N \times C$	how changes in pre-activation features affect post-activation values

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, $A_{l+1} = f_l(Z_l)$.

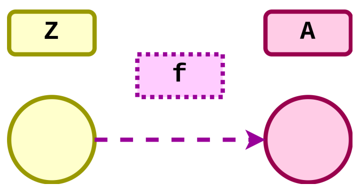


Figure C: Activation Function Topology

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 $\mathbf{A}$.

$$A = \texttt{Sigmoid.forward}(Z) \tag{6}$$
$$= \varsigma(Z) \tag{7}$$
$$= \frac{1}{1+e^{-Z}} \tag{8}$$

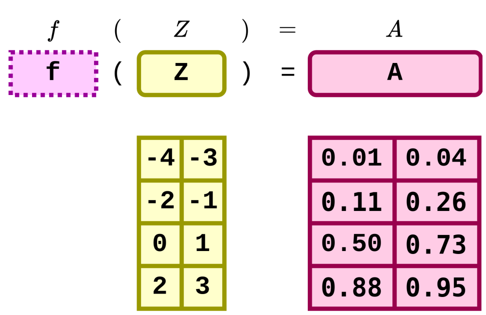


Figure D: Sigmoid Activation Forward Example

6.1.2 Sigmoid Backward Equation

Backward propagation helps us understand how changes in pre-activation features **Z** affect post-activation values **A**.

$$\frac{dA}{dZ} = \texttt{sigmoid.backward}() \tag{9}$$
$$= \varsigma(Z) - \varsigma^2(Z) \tag{10}$$
$$= A - A \odot A \tag{11}$$

6.2 Tanh [`mytorch.nn.Tanh`]

6.2.1 Tanh Forward Equation

$$A = \texttt{Tanh.forward}(Z) \tag{12}$$
$$= \tanh(Z) \tag{13}$$
$$= \frac{e^Z - e^{-Z}}{e^Z + e^{-Z}} \tag{14}$$

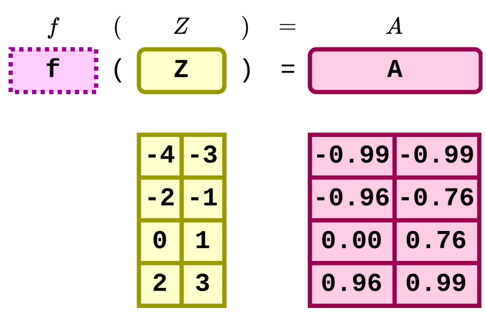


Figure E: Tanh Activation Forward Example

6.2.2 Tanh Backward Equation

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

$$\frac{dA}{dZ} = \texttt{tanh.backward}() \tag{15}$$
$$= \_?\_ \tag{16}$$

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

6.3 ReLU [`mytorch.nn.ReLU`]

6.3.1 ReLU Forward Equation

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

$$A = \texttt{relu.forward}(Z) \tag{17}$$
$$= \_?\_ \tag{18}$$

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

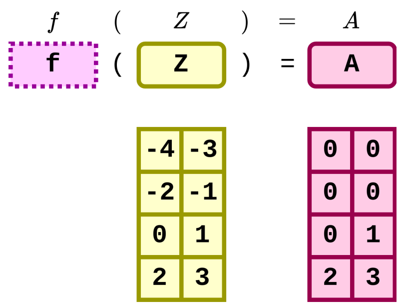


Figure F: ReLU Activation Forward Example

6.3.2 ReLU Backward Equation

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

$$\frac{dA}{dZ} = \texttt{relu.backward}() \tag{19}$$

$$= \begin{cases} \_?\_, & A > 0 \\ \_?\_, & A \leq 0 \end{cases} \tag{20}$$

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

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 layers
  - **f**: a list storing activation functions after each linear layer.
- Class methods:
  - `forward`: forward method takes input data $\mathbf{A_0}$ and applies the linear transformation `self.layers[i].forward` and activation function `self.f[i].forward` for $i = 0, ..., l-1$[6] where l is the 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.f[i].backward` and `self.layers[i].backward` for $i = l-1, ..., 0$. It does not return anything.

Please consider the following class structure:

```
class Model:

    def __init__(self):

        self.layers = # TODO
        self.f      = # TODO

    def forward(self, A):

        l = len(self.layers)
        for i in range(l):
            Z = # TODO
            A = # TODO

        return A

    def backward(self, dLdA):

        l = len(self.layers)
        for i in reversed(range(l)):
            dAdZ = # TODO
            dLdZ = # TODO
            dLdA = # TODO
```

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

[6] python 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 functions[7].

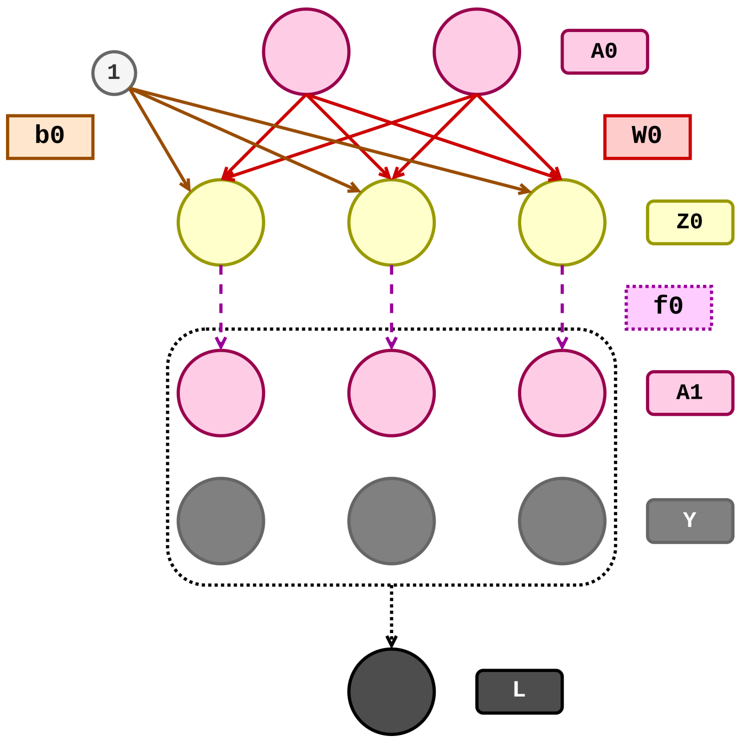


Figure G: MLP 0 Example Topology (Hidden Layers = 0)

7.1.1 MLP Forward Pseudocode (Hidden Layers = 0)

$$Z_0 = \texttt{layer0.forward}(A_0) \qquad \in \mathbb{R}^{N\times C_1} \tag{21}$$

$$A_1 = \texttt{f0.forward}(Z_0) \qquad \in \mathbb{R}^{N\times C_1} \tag{22}$$

7.1.2 MLP Backward Pseudocode (Hidden Layers = 0)

$$\frac{\partial A_1}{\partial Z_0} = \texttt{f0.backward()} \tag{23}$$

$$\frac{\partial L}{\partial Z_0} = \frac{\partial L}{\partial A_1} \odot \frac{\partial A_1}{\partial Z_0} \qquad \in \mathbb{R}^{N\times C_1} \tag{24}$$

$$\frac{\partial L}{\partial A_0} = \texttt{layer0.backward}(\frac{\partial L}{\partial Z_0}) \qquad \in \mathbb{R}^{N\times C_0} \tag{25}$$

$$\tag{26}$$

[7] Refer 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.

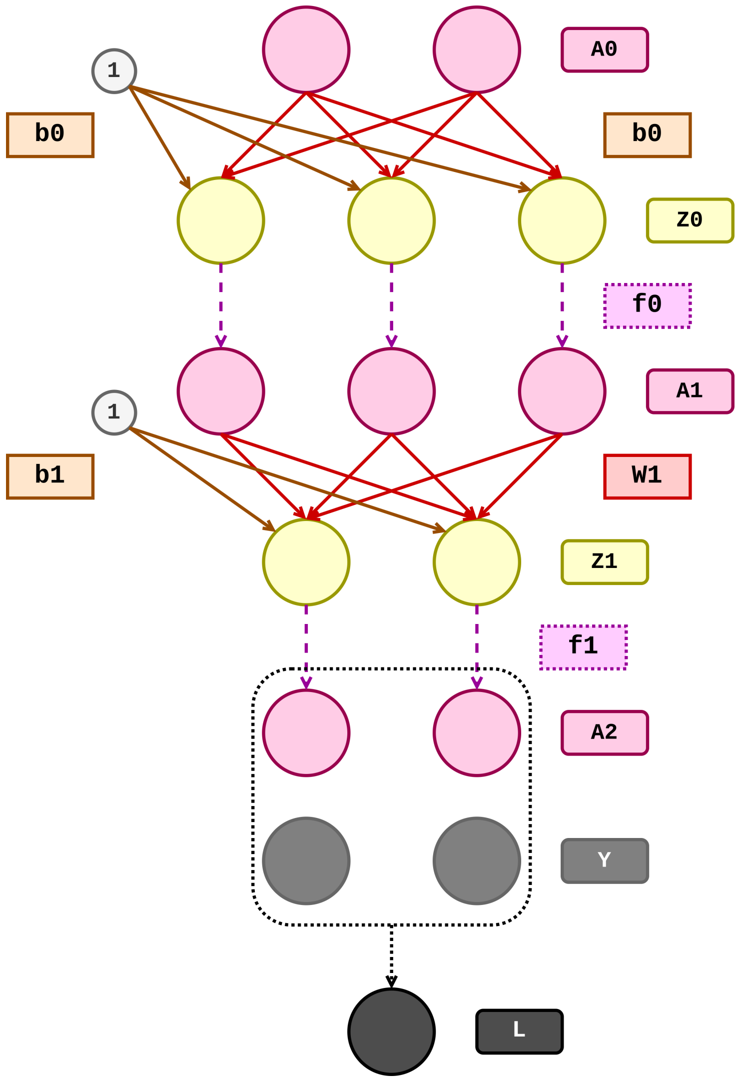


Figure H: MLP 1 Example Topology (Hidden Layers = 1)

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 one more layer. Hence, we won't provide you the pseudocode, but only a high level description with reference to Fig H:

- `forward` method takes input data $\mathbf{A_0}$ and applies the linear transformation `self.layers[0].forward` to get $\mathbf{Z_0}$.
- It then applies activation function `self.f0.forward` on $\mathbf{Z_0}$ to compute layer output $\mathbf{A_1}$.

- $\mathbf{A_1}$ is passed to the next linear layer, and we apply `self.layers[1].forward` to obtain $\mathbf{Z_1}$.
- Finally, we apply activation function `self.f1.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 $dLdA2$, 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.f[i].backward` and `self.layers[i].backward` for $i = 1, 0$.

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.

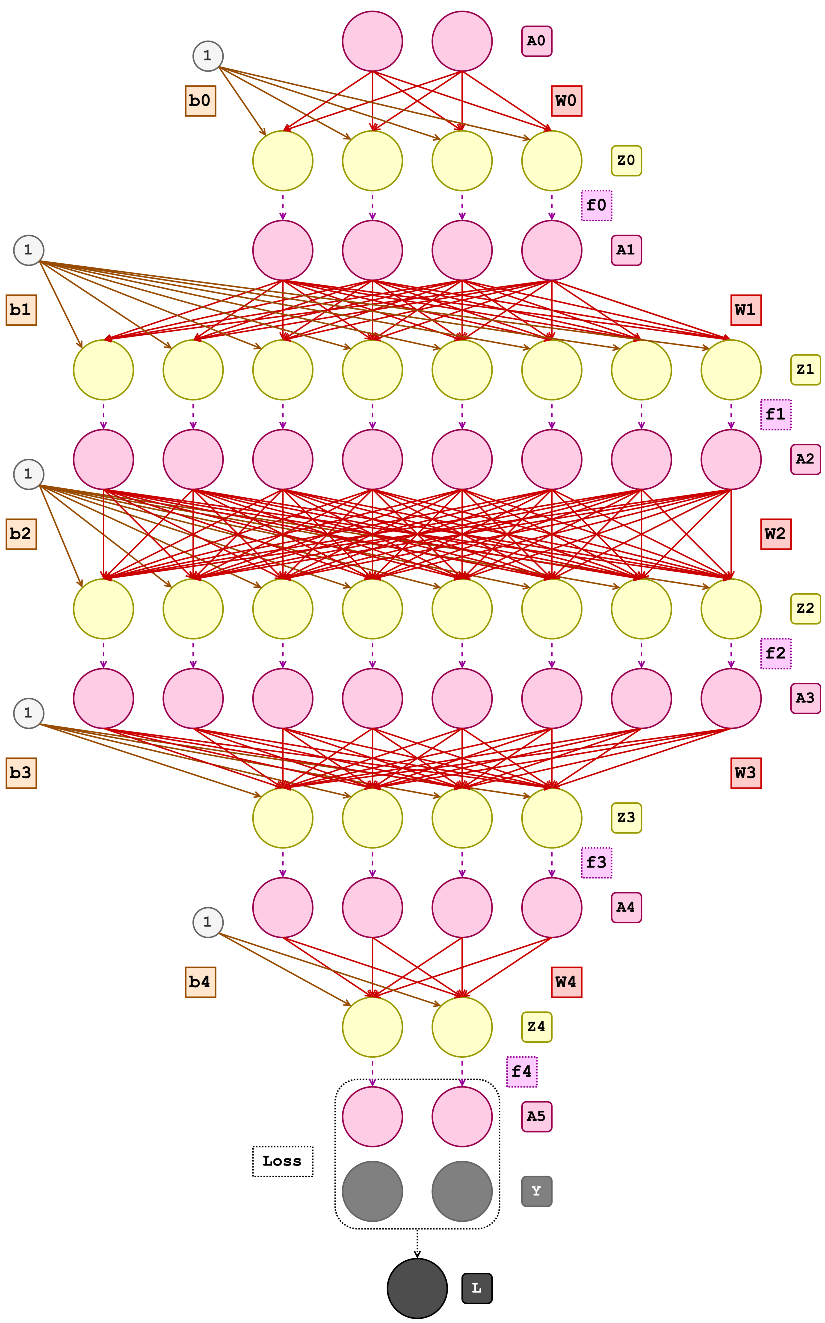


Figure I: MLP 4 Example Topology (Hidden Layers = 4)

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?

$$Z_i = A_i \cdot W_i + \iota \cdot b_i \qquad \in \mathbb{R}^{N \times C_{i+1}} \tag{27}$$

$$A_{i+1} = f_i(Z_i) \qquad \in \mathbb{R}^{N \times C_{i+1}} \tag{28}$$

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 calculation of which equation?

$$\frac{\partial A_{i+1}}{\partial Z_i} = \frac{\partial}{\partial Z_i} f_i(Z_i) \qquad \in \mathbb{R}^{N \times C_{i+1}} \tag{29}$$

$$\frac{\partial L}{\partial Z_i} = \frac{\partial L}{\partial A_{i+1}} \odot \frac{\partial A_{i+1}}{\partial Z_i} \qquad \in \mathbb{R}^{N \times C_{i+1}} \tag{30}$$

$$\frac{\partial L}{\partial A_i} = \frac{\partial L}{\partial Z_i} \cdot \left(\frac{\partial Z_i}{\partial A_i}\right)^T \qquad \in \mathbb{R}^{N \times C_i} \tag{31}$$

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      = A.shape[0]
        C      = A.shape[1]
        # 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.

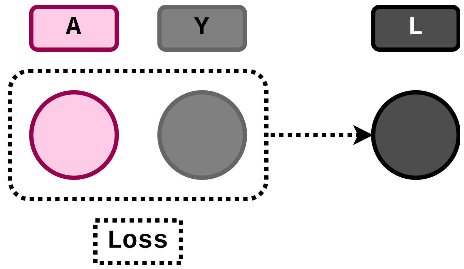


Figure J: Loss Function Topology

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

MSE stands for Mean Squred 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**:

$$SE(A, Y) = (A - Y) \odot (A - Y) \tag{32}$$

Then we calculate the sum of the squared error **SSE**, where $\iota_{\mathbf{N}}, \iota_{\mathbf{C}}$ are column vectors of size N and C which contain all 1s:

$$SSE(A, Y) = \iota_N^T \cdot SE(A, Y) \cdot \iota_C \tag{33}$$

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

$$MSELoss(A, Y) = \frac{SSE(A, Y)}{2 \cdot N \cdot C} \tag{34}$$

8.1.2 MSE Loss Backward Equation

$$\texttt{MSELoss.backward}() = \frac{A - Y}{N \cdot C} \tag{35}$$

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

Cross-entropy loss if one of the most commonly used loss function for probability-based classification problems.

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.
$\iota_\mathbf{N}, \iota_\mathbf{C}$ are column vectors of size N and C which contain all 1s. [8]

$$\texttt{softmax}(A) = \sigma(A) \tag{36}$$

$$= \frac{\exp(A)}{\sum_{j=1}^{C} \exp(A_{ij})} \tag{37}$$

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 $\mathbf{H}(\mathbf{A}, \mathbf{Y})$ of the distribution $\mathbf{A_i}$ relative to the target distribution $\mathbf{Y_i}$ for $i = 1, ..., N$:

$$\texttt{crossentropy} = H(A, Y) \tag{38}$$

$$= (-Y \odot \log(\sigma(A))) \cdot \iota_C \tag{39}$$

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-entrpy loss as that is the default for `PyTorch` as well:

$$\texttt{sum\_crossentropy\_loss} := \iota_N^T \cdot H(A, Y) \tag{40}$$

$$= SCE(A, Y) \tag{41}$$

$$\texttt{mean\_crossentropy\_loss} := \frac{SCE(A, Y)}{N} \tag{42}$$

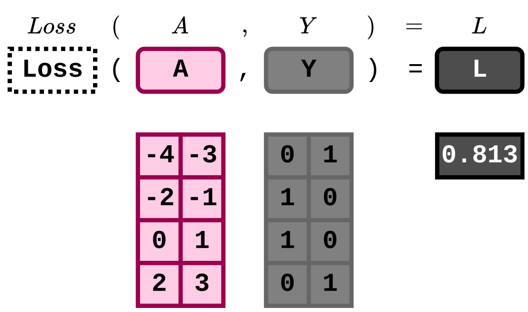


Figure K: Cross Entropy Loss Example Mapping

8.2.2 Cross-Entropy Loss Backward Equation

$$\texttt{xent.backward}() = \sigma(A) - Y \tag{43}$$

[8]The matrix division in Equation 37 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 [`mytorch.optim.SGD`] [10 points]

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. To do this, we perform stochastic gradient descent or SGD. There are many optimization methods to choose from, but SGD is used here because it is popular and straightforward to implement.

In this section, 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 none zero, update velocities `v_W` and `v_b`, which are changes in the gradient to get to the global minima. Momentum is a method that helps accelerate SGD by incorporating velocity from the previous update to reduce oscillations. 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 SGD Equation (Without Momentum)

$$W := W - \lambda \frac{\partial L}{\partial W} \tag{44}$$

$$b := b - \lambda \frac{\partial L}{\partial b} \tag{45}$$

9.2 SGD Equations (With Momentum)

$$v_W := \mu v_W + \frac{\partial L}{\partial W} \tag{46}$$

$$v_b := \mu v_b + \frac{\partial L}{\partial b} \tag{47}$$

$$W := W - \lambda v_W \tag{48}$$

$$b := b - \lambda v_b \tag{49}$$

10 Regularization [20 points]

10.1 Batch Normalization [mytorch.nn.BatchNorm1d]

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 1 and the elements of $Var[Z]$ are set to 0.
    - 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
```

Note: In the following sections, we are providing you with element-wise equations instead of matrix equations. As a deep learning ninja, please don't use `for loops` to implement them – that will be extremely slow!
Your task is first to come up with a matrix equation for each element-wise equation we provide, then implement them as code. If you ask TAs for help in this section, we will ask you to provide your matrix equations.

10.1.1 Batch Normalization Forward Training Equations (When `eval = False`)

First, we calculate the mini-batch mean μ and variance σ^2 of the current batch of data Z. μ_j and σ_j^2 represents the mean and variance of the jth feature. Z_{ij} refers to the element at the ith row and jth column of Z and represents the value of the jth feature in ith sample in the batch.

Table 6: Activation Function Components

Code Name	Math	Type	Shape	Meaning
`N`	N	scalar	-	batch size
`num_features`	C	scalar	-	number of features (same for input and output)
`alpha`	α	scalar	-	the coefficient used for running_M and running_V computations
`eps`	ϵ	scalar	-	a value added to the denominator for numerical stability.
`Z`	Z	matrix	$N \times C$	data input to the BN layer
`NZ`	$\hat{Z}$	matrix	$N \times C$	normalized input data
`BZ`	$\tilde{Z}$	matrix	$N \times C$	data output from the BN layer
`M`	μ	matrix	$1 \times C$	Mini-batch per feature mean
`V`	σ^2	matrix	$1 \times C$	Mini-batch per feature variance
`running_M`	$E[Z]$	matrix	$1 \times C$	Running average of per feature mean
`running_V`	$Var[Z]$	matrix	$1 \times C$	Running average of per feature variance
`BW`	γ	matrix	$1 \times C$	Scaling parameters
`Bb`	β	matrix	$1 \times C$	Shifting parameters
`dLdBW`	$\partial L/\partial\gamma$	matrix	$1 \times C$	how changes in γ affect loss
`dLdBb`	$\partial L/\partial\beta$	matrix	$1 \times C$	how changes in β affect loss
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C$	how changes in inputs affect loss
`dLdNZ`	$\partial L/\partial\hat{Z}$	matrix	$N \times C$	how changes in $\hat{Z}$ affect loss
`dLdBZ`	$\partial L/\partial\tilde{Z}$	matrix	$N \times C$	how changes in $\tilde{Z}$ affect loss
`dLdV`	$\partial L/\partial(\sigma^2)$	matrix	$1 \times C$	how changes in (σ^2) affect loss
`dLdM`	$\partial L/\partial\mu$	matrix	$1 \times C$	how changes in μ affect loss

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

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} Z_{ij} \qquad j = 1, ..., C \tag{50}$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N} (Z_{ij} - \mu_j)^2 \qquad j = 1, ..., C \tag{51}$$

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 .

$$\hat{Z}_i = \frac{Z_i - \mu}{\sqrt{\sigma^2 + \epsilon}} \qquad i = 1, ..., N \tag{52}$$

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

$$\tilde{Z}_i = \gamma \odot \hat{Z}_i + \beta \qquad i = 1, ..., N \tag{53}$$

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

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.

$$E[Z] = \alpha * E[Z] + (1 - \alpha) * \mu \qquad \in \mathbb{R}^{1\times C} \tag{54}$$

$$Var[Z] = \alpha * Var[Z] + (1 - \alpha) * \sigma^2 \qquad \in \mathbb{R}^{1\times C} \tag{55}$$

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 $\hat{Z}$.

$$\hat{Z}_i = \frac{Z_i - E[Z]}{\sqrt{Var[Z] + \epsilon}} \qquad i = 1, ..., N \tag{56}$$

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

$$\tilde{Z}_i = \gamma \odot \hat{Z}_i + \beta \qquad i = 1, ..., N \tag{57}$$

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.

$$\left(\frac{\partial L}{\partial \beta}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \beta}\right)_{ij} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}}\right)_{ij} \qquad j = 1, ..., C \tag{58}$$

$$\left(\frac{\partial L}{\partial \gamma}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \gamma}\right)_{ij} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \odot \hat{Z}\right)_{ij} \qquad j = 1, ..., C \tag{59}$$

$$\frac{\partial L}{\partial \hat{Z}} = \frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \hat{Z}} = \frac{\partial L}{\partial \tilde{Z}} \odot \gamma \tag{60}$$

$$\left(\frac{\partial L}{\partial \sigma^2}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \hat{Z}} \frac{\partial \hat{Z}}{\partial \sigma^2}\right)_{ij} \qquad j = 1, ..., C \tag{61}$$

$$= -\frac{1}{2} \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \hat{Z}} \odot (Z - \mu) \odot (\sigma^2 + \epsilon)^{-\frac{3}{2}}\right)_{ij} \tag{62}$$

$$\frac{\partial \hat{Z}_i}{\partial \mu} = \frac{\partial}{\partial \mu} \left[(Z_i - \mu)(\sigma^2 + \epsilon)^{-\frac{1}{2}}\right] \qquad i = 1, ..., N \tag{63}$$

$$= -(\sigma^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2}(Z_i - \mu) \odot \left(\sigma^2 + \epsilon\right)^{-\frac{3}{2}} \left(-\frac{2}{N} \sum_{i=1}^{N} (Z_i - \mu)\right) \tag{64}$$

$$\frac{\partial L}{\partial \mu} = \sum_{i=1}^{N} \frac{\partial L}{\partial \hat{Z}_i} \frac{\partial \hat{Z}_i}{\partial \mu} \tag{65}$$

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.

$$\frac{\partial L}{\partial Z_i} = \frac{\partial L}{\partial \hat{Z}_i} \frac{\partial \hat{Z}}{\partial Z_i} = \frac{\partial L}{\partial \hat{Z}_i} \left[(\sigma^2 + \epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L}{\partial \sigma^2} \left[\frac{2}{N}(Z_i - \mu)\right] + \frac{1}{N} \frac{\partial L}{\partial \mu} \tag{66}$$