Homework 1 Part 1

An Introduction to Neural Networks

11-785: Introduction to Deep Learning (Spring 2022)

OUT: **January 23, 2022**
Early Submission Bonus Deadline: **February 2, 2022, 11:59 PM, Eastern Time**
DUE: **February 17, 2022, 11:59 PM, Eastern Time**

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 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.

- **Early submission bonus deadline:**
  - If you complete this assignment successfully and achieve full marks on Autolab before **February 2, 2022, 11:59 PM, Eastern Time**, you will receive **5** point bonus for this assignment.

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 activations
  - 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 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 implement backpropagation through the linear and activation layers
  - How to compute loss derivatives for the network parameters (including weights, biases and batch norm parameters)
  - How to implement the Stochastic Gradient Descent (SGD) optimizer

Contents

1 Introduction

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, convolutional neural networks, 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.

In this assignment, we will start by creating the core components of multilayer perceptrons: linear layers, activations, loss functions, and batch normalization. You will implement these classes 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. You job is to specifically implement the mathematics into code, and understand how all the components are related. You are required to use `numpy` and no other python library.

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`. Libraries like `PyTorch`, `TensorFlow`, and `Keras` are not allowed.

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 doing what you can to make the operations possible.

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

2 Installation

The culmination of all of the Homework Part 1's will be your own custom deep learning library, along with detailed examples, which we are calling *MyTorch*©. 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.

- **Install** python 3, numpy, ipython. In order to run the provided autograder, you need to install the following libraries in python.

```
pip3 install numpy==1.18.5
pip3 install ipython==7.16.1
pip3 install notebook
```

  Please check your installation versions with the list below.

```
#   Name        Version
    anaconda    2020.07
    conda       4.8.3
```

```
ipython     7.16.1
python      3.8.3
numpy       1.18.5
notebook
```

3 Setup and Submission

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

  ```
  tar -xvf handout.tar
  ```

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

```
handin
└── autograde
    ├── hw1p1_autograder.ipynb
    ├── hw1p1_autograder.py
    └── mytorch
        ├── __init__.py
        ├── models
        │   ├── __init__.py
        │   └── hw1.py
        ├── nn
        │   ├── __init__.py
        │   └── modules
        │       ├── __init__.py
        │       ├── activation.py
        │       ├── batchnorm.py
        │       ├── linear.py
        │       └── loss.py
        └── optim
            ├── __init__.py
            └── sgd.py
```

Figure A: File Structure Tree, `handin`

- **Autograde** your code by

  Option 1: Executing in Terminal

  Please confirm that you are in the `./autograde` directory and execute the following in terminal:

  ```
  python hw1p1_autograder.py
  ```

  Option 2: Executing in Jupyter Notebook

  Please confirm that you are in the `./autograde` directory and launch jupyter notebook:

  ```
  jupyter notebook
  ```

  Then open `hw1p1_autograder.ipynb` and run the notebook.

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

  ```
  tar -cvf handin.tar handin
  ```

- **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. Libraries like `PyTorch, TensorFlow, Keras` are not allowed.
  - Add, move, or remove any files or change any filenames.

4 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. This is by design – you are *required* to verify all parts of your homework on the local autograder before you submit it.

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.

To test any individual component on your local autograder, set the "DEBUG_AND_GRADE" flag for it to true. E.g. to test your implementation of RELU set "DEBUG_AND_GRADE_RELU = True".

5 Activation Functions [5 points]

In artificial neural networks, the activation function of a node defines the output of that node given an input or set of inputs. A standard integrated circuit can be seen as a digital network of activation functions that can be "ON" (1) or "OFF" (0), depending on input. This is similar to the linear perceptron in neural networks. However, only nonlinear activation functions allow such networks to compute nontrivial problems using only a small number of nodes, and such activation functions are called nonlinearities.[1] An extensive list of activation functions including many new promising alternate activation functions that have been shown to perform better than popular activation functions on benchmark problems is presented in.[2][3]

(Source: https://en.wikipedia.org/wiki/Activation_function)

During forward propagation, pre-activation features are passed to the activation function to calculate their post-activation values. As a non-linearity, activation functions allow for more complex relationships to be estimated. Activation functions do not have parameters. Backward propagation helps us understand how changes in pre-activation features affect post-activation values. The values calculated during backward propagation are used to enable downstream computation, as seen in subsequent sections.

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

As you can see, the Activation class has forward and backward attribute functions. In forward, we calculate `A` and store its value for use in backward. The attribute function `forward` includes multiple components.

- As an argument, forward expects input `Z`.
- As an attribute, forward stores variable `A`.
- As an output, forward returns variable `A`.

In backward, we calculate the gradient changes needed for optimization. The attribute function `backward` includes multiple components.

- As arguments, backward expects no inputs.
- As attributes, backward stores no variables.
- As an output, backward returns variable `dAdZ`.

To facilitate understanding, we have organized a table describing all relevant variables.

The activation function topology is visualized in Figure A, whose reference persists throughout this document.

The following subsections provide: the equation needed for forward computation (Section 4.X.1), an example forward computation (Section 4.X.2), and equations needed for backward computation (Section 4.X.3).

Table 1: Activation Function Components

Code Name	Math	Type	Shape	Meaning
`N`	N	scalar	-	number of observations
`C`	C	scalar	-	number of features
`Z`	Z	matrix	$N \times C$	pre-activation features
`A`	A	matrix	$N \times C$	post-activation values
`Ones`	ι	matrix	$C \times 1$	data constant for features
`dAdZ`	$\partial A / \partial Z$	matrix	$N \times C$	how changes in pre-activation features affect post-activation values

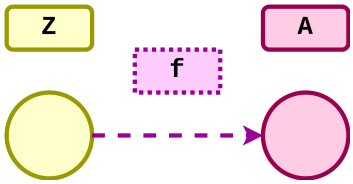

Figure B: Activation Function Topology

5.1 Sigmoid [`mytorch.nn.Sigmoid`]

5.1.1 Sigmoid Forward Equation

$$\texttt{Sigmoid}(Z) = \texttt{Sigmoid.forward}(Z) \tag{1}$$
$$= \varsigma(Z) \tag{2}$$
$$= \frac{1}{1+e^{-Z}} \tag{3}$$

5.1.2 Sigmoid Forward Example

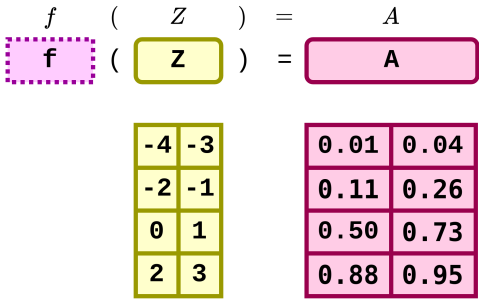

Figure C: Sigmoid Activation Forward Example

5.1.3 Sigmoid Backward Equation

$$\texttt{sigmoid.backward}() = \varsigma(Z) - \varsigma^2(Z) \tag{4}$$
$$= A - A^2 \tag{5}$$

5.2 Tanh [`mytorch.nn.Tanh`]

5.2.1 Tanh Forward Equation

$$\texttt{tanh}(Z) = \texttt{Tanh.forward}(Z) \tag{6}$$
$$= \tanh(Z) \tag{7}$$
$$= \frac{\sinh(Z)}{\cosh(Z)} \tag{8}$$

5.2.2 Tanh Forward Example

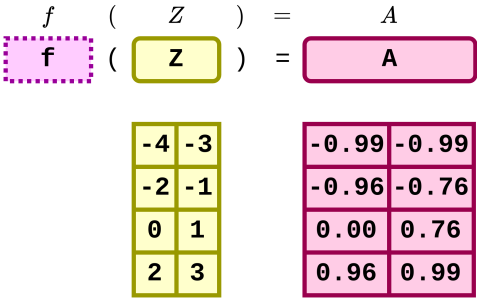

Figure D: Tanh Activation Forward Example

5.2.3 Tanh Backward Equation

$$\texttt{tanh.backward}() = 1 - \tanh^2(Z) \tag{9}$$
$$= 1 - A^2 \tag{10}$$

5.3 ReLU [`mytorch.nn.ReLU`]

5.3.1 ReLU Forward Equation

$$\texttt{relu}(Z) = \texttt{relu.forward}(Z) \tag{11}$$
$$= \max(Z, 0) \tag{12}$$

5.3.2 ReLU Forward Example

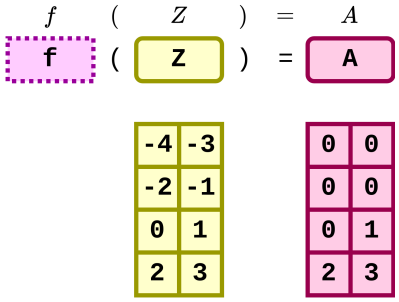

Figure E: ReLU Activation Forward Example

5.3.3 ReLU Backward Equation

$$\texttt{relu.backward}() = \begin{cases} 1, & \max(Z, 0) > 0 \\ 0, & \max(Z, 0) \leq 0 \end{cases} \tag{13}$$

$$= \begin{cases} 1, & A > 0 \\ 0, & A \leq 0 \end{cases} \tag{14}$$

Hint: Search and read the docs on `np.max`, `np.maximum`, and `np.where`.

6 Loss Functions [5 points]

In statistics, typically a loss function is used for parameter estimation, and the event in question is some function of the difference between estimated and true values for an instance of data. The concept, as old as Laplace, was reintroduced in statistics by Abraham Wald in the middle of the 20th century.[2] In the context of economics, for example, this is usually economic cost or regret. In classification, it is the penalty for an incorrect classification of an example. In actuarial science, it is used in an insurance context to model benefits paid over premiums, particularly since the works of Harald Cramér in the 1920s.[3] In optimal control, the loss is the penalty for failing to achieve a desired value. In financial risk management, the function is mapped to a monetary loss.

(Source: https://en.wikipedia.org/wiki/Loss_function)

During forward propagation, an input and target are passed to the loss function to calculate a loss value. The input of the loss function is typically a model output, while the target is the ground truth values we are estimating. The loss value is a scalar quantity used to evaluate the quality of the model predictions. To improve a model, we need to know how changes in model outputs affect loss. The values calculated during backward propagation are used to enable downstream computation, as seen in subsequent sections.

In this section, your task is to implement the forward and backward attribute functions of the Loss class. 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]
        error  = # TODO
        L      = np.sum(error) / self.N

        return L

    def backward(self):

        dLdA = # TODO

        return dLdA
```

As you can see, the Loss class has forward and backward attribute functions. In forward, we calculate `L` and store values need for backward. The attribute function `forward` include:

- As an argument, forward expects input `A`, and `Y`.
- As attributes, forward stores variable `A`, `Y`, and additional attributes as needed.
- As an output, forward returns variable `L`.

In backward, we calculate multiple gradient changes and store values needed for optimization. The attribute function `backward` includes:

- As an argument, backward expects no inputs.
- As attributes, backward stores no variables.
- As an output, backward returns variable `dLdA`.

To facilitate understanding, we have organized a table describing all relevant variables.

Table 2: Loss Function Components

Code Name	Math	Type	Shape	Meaning
`N`	N	scalar	-	number of observations
`C`	C	scalar	-	number of features
`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
`*`	$\odot$	op	-	elementwise multiply (aka Hadamard Product)
`@`	$\cdot$	op	-	dot product

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

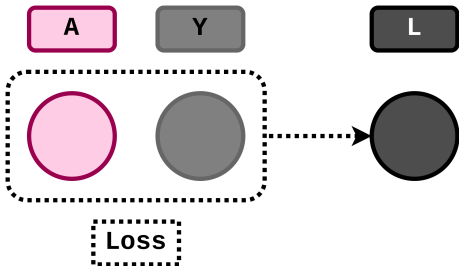

Figure F: Loss Function Topology

The following subsections provide: the equation needed for forward computation (Section 5.X.1), an example forward computation (Section 5.X.2), and equations needed for backward computation (Section 5.X.3).

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

6.1.1 MSE Loss Forward Equation

We first calculate the squared error between the model outputs and the ground-truth values. Then we sum the squared error and calculate the per-component MSE loss.

$$\texttt{Ones}_C := \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{C \times 1} \tag{15}$$

$$= \iota_C \tag{16}$$

$$\texttt{Ones}_N := \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{N \times 1} \tag{17}$$

$$= \iota_N \tag{18}$$

$$SE = (A - Y) \odot (A - Y) \tag{19}$$

$$SSE = \iota_N^T \cdot SE(A, Y) \cdot \iota_C \tag{20}$$

$$MSE = \frac{SSE(A, Y)}{N \cdot C} \tag{21}$$

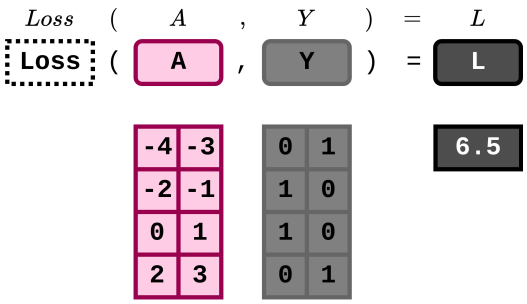

Figure G: MSE Loss Example Mapping

6.1.2 MSE Loss Forward Example

6.1.3 MSE Loss Backward Equation

$$\texttt{MSELoss.backward}() = A - Y \tag{22}$$

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

6.2.1 Cross-Entropy Loss Forward Equation

$$\texttt{Ones}_C := \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{C \times 1} \tag{23}$$

$$= \iota_C \tag{24}$$

$$\texttt{Ones}_N := \begin{bmatrix} 1 \\ \vdots \\ 1 \end{bmatrix}_{N \times 1} \tag{25}$$

$$= \iota_N \tag{26}$$

$$\texttt{softmax} := \frac{\exp(A)}{\exp(A) \cdot \iota_C} \tag{27}$$

$$= \sigma(A) \tag{28}$$

$$\texttt{crossentropy} := -Y \odot \log(\sigma(A)) \tag{29}$$

$$= H(A, Y) \tag{30}$$

$$\texttt{sum\_crossentropy} := \iota_N^T \cdot H(A, Y) \cdot \iota_C \tag{31}$$

$$= SCE(A, Y) \tag{32}$$

$$\texttt{loss} := \frac{SCE(A, Y)}{N} \tag{33}$$

6.2.2 Cross-Entropy Loss Forward Example

6.2.3 Cross-Entropy Loss Backward Equation

$$\texttt{xent.backward}() = \sigma(A) - Y \tag{34}$$

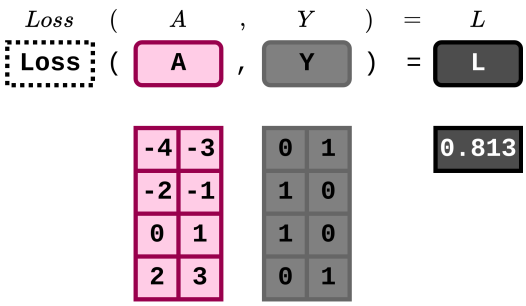

Figure H: Cross Entropy Loss Example Mapping

7 Neural Network Layers [15 points]

7.1 Linear Layer [mytorch.nn.Linear]

Linear layers use linear predictor functions to model relationships whose unknown parameters are estimated from the data. Most commonly, the conditional mean of the output features given the values of the input features is assumed to be an affine function of those values. Linear layers focus on the joint probability distribution of all these variables.

In linear regression, the relationships are modeled using linear predictor functions whose unknown model parameters are estimated from the data. Most commonly, the conditional mean of the response given the values of the explanatory variables (or predictors) is assumed to be an affine function of those values. Multivariate regression focuses on the joint probability distribution of all of these variables. (Source: https://en.wikipedia.org/wiki/Linear_regression)

During forward propagation, input features are passed to the linear layer to calculate its output features. These values are used to optimize the loss function. To improve the linear layer, we need to know how changes in its output affect loss. This allows us to do backward propagation and calculate how changes in the layer weights or bias affect loss. To enable downstream computation, backward propagation returns how changes in the layer inputs affect loss.

In this section, your task is to implement the forward and backward attribute functions of the Linear class. Please consider the following class structure.

```
class Linear:

    def __init__(self, in_features, out_features):

        self.W    = np.zeros((out_features, in_features))
        self.b    = np.zeros((out_features, 1))
        self.dLdW = np.zeros((out_features, in_features))
        self.dLdb = np.zeros((out_features, 1))

    def forward(self, A):

        self.A    = A
        self.N    = A.shape[0]
        self.Ones = np.ones((self.N,1), dtype="f")
        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
```

As you can see, the Linear class has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__`

includes:

- As arguments, Linear will be specified using `in_feature` and `out_feature`.
- As attributes, Linear will be initialized with `W`, `dLdW`, `b`, and `dLdb`.

In forward, we calculate `Z` and store values needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A`.
- As an attribute, forward stores variables `A`, `N`, and `Ones`.
- As an output, forward returns variable `Z`.

In backward, we calculate multiple gradient changes and store values needed for optimization. The attribute function `backward` includes:

- As an argument, backward expects input `dLdZ`.
- As attributes, backward stores variables `dLdW` and `dLdb`.
- As an output, backward returns variable `dLdA`.

To facilitate understanding, we have organized a table describing all relevant variables.

Table 3: Linear Layer Components

Code Name	**Math**	**Type**	**Shape**	**Meaning**
`N`	N	scalar	-	number of observations
`in_features`	C_0	scalar	-	number of input features
`out_features`	C_1	scalar	-	number of output features
`A`	A	matrix	$N \times C_0$	data input to be weighted
`Z`	Z	matrix	$N \times C_1$	data output from the layer
`Ones`	ι	matrix	$N \times 1$	data constant for bias
`W`	W	matrix	$C_1 \times C_0$	weight parameters
`b`	b	matrix	$C_1 \times 1$	bias parameters
`dLdZ`	$\partial L/\partial Z$	matrix	$N \times C_1$	how changes in outputs affect loss
`dZdA`	$\partial Z/\partial A$	matrix	$C_0 \times C_1$	how changes in inputs affect outputs
`dZdW`	$\partial Z/\partial W$	matrix	$N \times C_0$	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	$N \times C_0$	how changes in inputs affect loss
`dLdW`	$\partial L/\partial W$	matrix	$C_1 \times C_0$	how changes in weights affect loss
`dLdb`	$\partial L/\partial b$	matrix	$C_1 \times 1$	how changes in bias affect loss

The linear layer topology is visualized in Figure I, whose reference persists throughout this document.

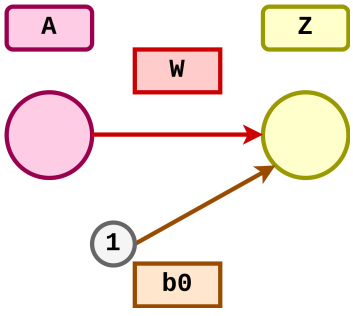

Figure I: Linear Layer Topology

7.1.1 Linear Layer Forward Equation

$$Z = A \cdot W^T + \iota \cdot b^T \qquad \in \mathbb{R}^{N \times C_1} \tag{35}$$

7.1.2 Linear Layer Forward Example

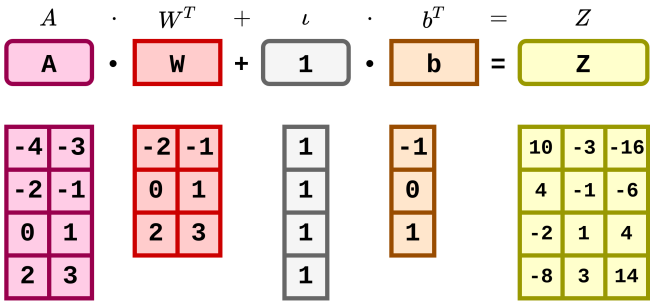

Figure J: Linear Layer Example Mapping

7.1.3 Linear Layer Backward Equations

$$\frac{\partial L}{\partial A} = \left(\frac{\partial L}{\partial Z}\right) \cdot \left(\frac{\partial Z}{\partial A}\right)^T \qquad \in \mathbb{R}^{N \times C_0} \tag{36}$$

$$\frac{\partial L}{\partial W} = \left(\frac{\partial L}{\partial Z}\right)^T \cdot \left(\frac{\partial Z}{\partial W}\right) \qquad \in \mathbb{R}^{C_1 \times C_0} \tag{37}$$

$$\frac{\partial L}{\partial b} = \left(\frac{\partial L}{\partial Z}\right)^T \cdot \left(\frac{\partial Z}{\partial b}\right) \qquad \in \mathbb{R}^{C_1 \times 1} \tag{38}$$

8 Optimizers [`mytorch.optim.SGD`] [10 points]

Stochastic gradient descent (often abbreviated SGD) is an iterative method for optimizing an objective function with suitable smoothness properties (e.g. differentiable or subdifferentiable). It can be regarded as a stochastic approximation of gradient descent optimization, since it replaces the actual gradient (calculated from the entire data set) by an estimate thereof (calculated from a randomly selected subset of the data). Especially in high-dimensional optimization problems this reduces the computational burden, achieving faster iterations in trade for a lower convergence rate.

(Source: https://en.wikipedia.org/wiki/Stochastic_gradient_descent)

To recap, we have seen how to do forward propagation, loss calculation, and backward propagation for the core classes used in neural networks. Forward propagation is used for estimation, loss calculation tells us valuate the quality of our estimates, and backward propagation informs us on how changes in parameters affect loss.

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. Because parameter gradients tell us which direction makes the model worse, we move opposite the direction of the gradient to update parameters. The learning rate, lr, is a hyperparemeter scaling the updates. Momentum incorporates information from previous updates, scaled by hyperparameter μ. For reference, $\mu = 0$ means no momentum.

In this section, your task is to implement the step attribute function of the SGD class. 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

        return None
```

As you can see, SGD has initialization and a step attribute function. Immediately once the class is instan-

tiated, the code in `__init__` is run. The initialization phase using `__init__` includes:

- As arguments, SGD will be specified using `model`, `lr`, and `momentum`.
- As attributes, SGD stores the variables `l`, `lr`, and `momentum`.
- Also as attributes, SGD initializes lists of variables `v_W` and `v_b`.

In step, we update `W` and `b` of each of the model layers, and store values needed for momentum. The attribute function `step` includes:

- As arguments, step expects no inputs.
- As attributes, step stores variables `v_W[i]` and `v_b[i]` at each layer i.
- Also as attributes, step updates variables `l[i].W` and `l[i].b`.
- As an output, forward returns no values.

To facilitate understanding, we have organized a table describing all relevant variables.

Table 4: 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 momentum weight parameters, one for each layer
`v_b`	-	list	L	list of momentum bias parameters, one for each layer
`v_W[i]`	v_{W_i}	matrix	$C_{i+1} \times C_i$	weight parameter for layer i momentum
`v_b[i]`	v_{b_i}	matrix	$C_{i+1} \times 1$	bias parameter for layer i momentum
`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

8.1 SGD Equation (Without Momentum)

$$W := W - \lambda \frac{\partial L}{\partial W} \tag{39}$$

$$b := b - \lambda \frac{\partial L}{\partial b} \tag{40}$$

8.2 SGD Equations (With Momentum)

$$v_W := \mu v_W + \frac{\partial L}{\partial W} \tag{41}$$

$$v_b := \mu v_b + \frac{\partial L}{\partial b} \tag{42}$$

$$W := W - \lambda v_W \tag{43}$$

$$b := b - \lambda v_b \tag{44}$$

8.3 Testing SGD Equations

Testing SGD is a little different than in other sections. We must create a pseudo model, with all the attributes we expect to use. This example shows what a typical model initialization would look like. In later sections, you will implement forward and backward, but you do not need to implement these to test SGD.

```
class PseudoModel:

    def __init__(self):

        self.layers = [ mytorch.nn.Linear(3,2) ]
        self.f      = [ mytorch.nn.ReLU() ]

    def forward(self, A):

        return NotImplemented

    def backward(self):

        return NotImplemented

# Create Example Model
pseudo_model = PseudoModel()

pseudo_model.layers[0].W    = np.ones((3,2))
pseudo_model.layers[0].dLdW = np.ones((3,2))/10
pseudo_model.layers[0].b    = np.ones((3,1))
pseudo_model.layers[0].dLdb = np.ones((3,1))/10

print("W\n\n", pseudo_model.layers[0].W)
print("W\n\n", pseudo_model.layers[0].b)

# Test Example Models
optimizer = SGD(pseudo_model, lr=1)
optimizer.step()

print("W\n\n", pseudo_model.layers[0].W)
print("W\n\n", pseudo_model.layers[0].b)
```

9 Neural Network Models [45 points]

An MLP consists of at least three layers of nodes: an input layer, a hidden layer and an output layer. Except for the input nodes, each node is a neuron that uses a nonlinear activation function. MLP utilizes a supervised learning technique called backpropagation for training.[2][3] Its multiple layers and non-linear activation distinguish MLP from a linear perceptron. It can distinguish data that is not linearly separable.

(Source: https://en.wikipedia.org/wiki/Multilayer_perceptron)

9.1 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. Please consider the following class structure.

```
class MLP0:

    def __init__(self):

        self.layers = [ mytorch.nn.Linear(2, 3) ]
        self.f      = [ mytorch.nn.ReLU() ]

    def forward(self, A0):

        Z0 = #TODO
        A1 = #TODO

        return A1

    def backward(self, dLdA1):

        dA1dZ0 = # TODO
        dLdZ0  = # TODO
        dLdA0  = # TODO

        return None
```

As you can see, MLP0 has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes:

- As arguments, MLP0 expects no inputs.
- As attributes, MLP0 stores lists of objects `layers` and `f`.
- In layers, the Linear class arguments will be specified using `in_feature` and `out_feature`.
  - Linear Layer 0 has `in_feature` = 2 and `out_feature` = 3.
- Also in layers, Linear attributes are automatically defined for `W`, `dLdW`, `b`, and `dLdb`.
- In activations, the ReLU class has no arguments

In forward, we calculate `A1`. Our layer and activation objects automatically store variables needed for backward. The attribute function `forward` includes:

- As an argument, forward expects input `A0`.
- No new attributes are defined in forward.
- As an output, forward returns variable `A1`.

In backward, we calculate multiple gradient changes. Our layer and activation objects also automatically store values needed for optimization. The attribute function `backward` includes:

- As an argument, backward expects input `dLdA1`.
- No new attributes are defined in backward.
- As an output, backward returns no values.

The MLP0 topology is visualized in Figure J. The network is displayed vertically to fit on the page.

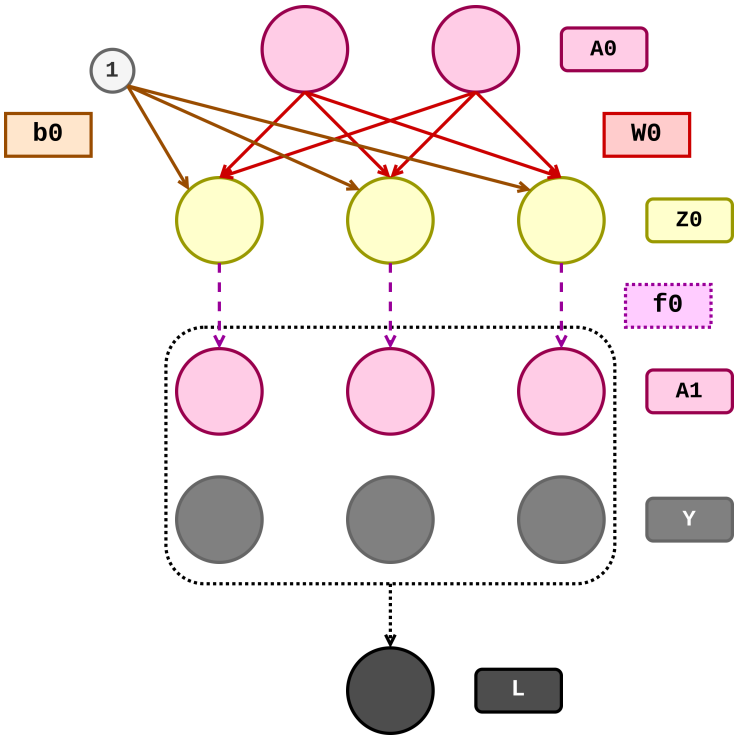

Figure K: MLP 0 Example Topology (Hidden Layers = 0)

9.1.1 MLP Forward Equations (Hidden Layers = 0)

$$Z_0 = A_0 W_0^T + \iota b_0^T \qquad \in \mathbb{R}^{N \times C_1} \tag{45}$$

$$A_1 = f_0(Z_0) \qquad \in \mathbb{R}^{N \times C_1} \tag{46}$$

9.1.2 MLP Backward Equations (Hidden Layers = 0)

$$\frac{\partial A_1}{\partial Z_0} = \frac{\partial}{\partial Z_0} f_0(Z_0) \qquad \in \mathbb{R}^{N \times C_1} \tag{47}$$

$$\frac{\partial L}{\partial Z_0} = \frac{\partial L}{\partial A_1} \odot \frac{\partial A_1}{\partial Z_0} \qquad \in \mathbb{R}^{N \times C_1} \tag{48}$$

$$\frac{\partial L}{\partial A_0} = \frac{\partial L}{\partial Z_0} \cdot \left(\frac{\partial Z_0}{\partial A_0} \right)^T \qquad \in \mathbb{R}^{N \times C_0} \tag{49}$$

$$\frac{\partial L}{\partial W_0} = \left(\frac{\partial L}{\partial Z_0} \right)^T \cdot \left(\frac{\partial Z_0}{\partial W_0} \right) \qquad \in \mathbb{R}^{C_1 \times C_0} \tag{50}$$

$$\frac{\partial L}{\partial b_0} = \left(\frac{\partial L}{\partial Z_0} \right)^T \cdot \left(\frac{\partial Z_0}{\partial b_0} \right) \qquad \in \mathbb{R}^{C_1 \times 1} \tag{51}$$

$$\tag{52}$$

9.2 MLP (Hidden Layers = 1) [mytorch.models.MLP1] [15 points]

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

```
class MLP1:

    def __init__(self):

        self.layers = # TODO
        self.f      = [ mytorch.nn.ReLU(),
                        mytorch.nn.ReLU() ]

    def forward(self, A0):

        Z0 = # TODO
        A1 = # TODO
        Z1 = # TODO
        A2 = # TODO

        return A2

    def backward(self, dLdA2):

        dA2dZ1 = # TODO
        dLdZ1  = # TODO
        dLdA1  = # TODO
        dA1dZ0 = # TODO
        dLdZ0  = # TODO
        dLdA0  = # TODO

        return None
```

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

The MLP1 topology is visualized in Figure K. The network is displayed vertically to fit on the page. You must use the diagram to deduce what the model specification is for the linear layers.

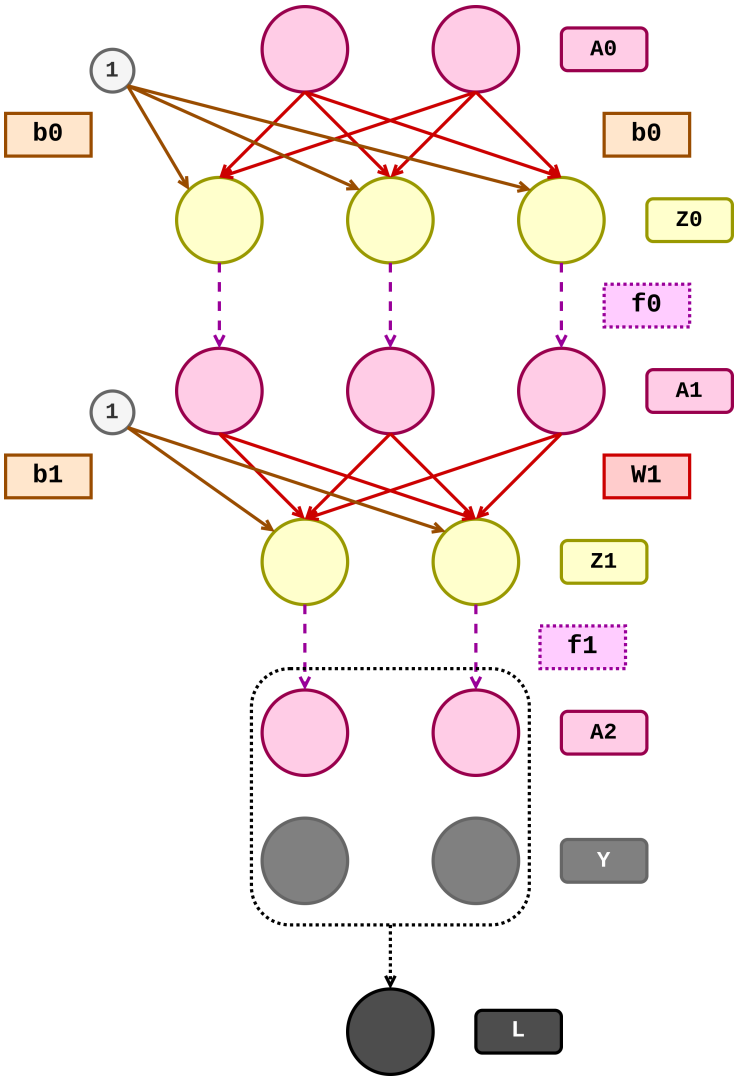

Figure L: MLP 1 Example Topology (Hidden Layers = 1)

9.2.1 MLP Forward Equations (Hidden Layers = 1)

$$Z_0 = A_0 W_0^T + \iota b_0^T \qquad \in \mathbb{R}^{N \times C_1} \tag{53}$$
$$A_1 = f_0(Z_0) \qquad \in \mathbb{R}^{N \times C_1} \tag{54}$$
$$Z_1 = A_1 W_1^T + \iota b_1^T \qquad \in \mathbb{R}^{N \times C_2} \tag{55}$$
$$A_2 = f_1(Z_1) \qquad \in \mathbb{R}^{N \times C_2} \tag{56}$$

9.2.2 MLP Backward Equations (Hidden Layers = 1)

$$\frac{\partial A_2}{\partial Z_1} = \frac{\partial}{\partial Z_1} f_1(Z_1) \qquad \in \mathbb{R}^{N \times C_2} \tag{57}$$
$$\frac{\partial L}{\partial Z_1} = \frac{\partial L}{\partial A_2} \odot \frac{\partial A_2}{\partial Z_1} \qquad \in \mathbb{R}^{N \times C_2} \tag{58}$$
$$\frac{\partial L}{\partial A_1} = \frac{\partial L}{\partial Z_1} \cdot \left(\frac{\partial Z_1}{\partial A_1} \right)^T \qquad \in \mathbb{R}^{N \times C_1} \tag{59}$$
$$\frac{\partial L}{\partial W_1} = \left(\frac{\partial L}{\partial Z_1} \right)^T \cdot \left(\frac{\partial Z_1}{\partial W_1} \right) \qquad \in \mathbb{R}^{C_2 \times C_1} \tag{60}$$
$$\frac{\partial L}{\partial b_1} = \left(\frac{\partial L}{\partial Z_1} \right)^T \cdot \left(\frac{\partial Z_1}{\partial b_1} \right) \qquad \in \mathbb{R}^{C_1 \times 1} \tag{61}$$

$$\frac{\partial A_1}{\partial Z_0} = \frac{\partial}{\partial Z_0} f_0(Z_0) \qquad \in \mathbb{R}^{N \times C_1} \tag{62}$$
$$\frac{\partial L}{\partial Z_0} = \frac{\partial L}{\partial A_1} \odot \frac{\partial A_1}{\partial Z_0} \qquad \in \mathbb{R}^{N \times C_1} \tag{63}$$
$$\frac{\partial L}{\partial A_0} = \frac{\partial L}{\partial Z_0} \cdot \left(\frac{\partial Z_0}{\partial A_0} \right)^T \qquad \in \mathbb{R}^{N \times C_0} \tag{64}$$
$$\frac{\partial L}{\partial W_0} = \left(\frac{\partial L}{\partial Z_0} \right)^T \cdot \left(\frac{\partial Z_0}{\partial W_0} \right) \qquad \in \mathbb{R}^{C_1 \times C_0} \tag{65}$$
$$\frac{\partial L}{\partial b_0} = \left(\frac{\partial L}{\partial Z_0} \right)^T \cdot \left(\frac{\partial Z_0}{\partial b_0} \right) \qquad \in \mathbb{R}^{C_1 \times 1} \tag{66}$$
$$\tag{67}$$

9.3 MLP (Hidden Layers = 4) [mytorch.models.MLP4] [20 points]

In this section, your task is to implement the forward and backward attribute functions of the MLP4 class. Please consider the following class structure.

```
class MLP4:

    def __init__(self):

        self.layers = # TODO
        self.f      = [ mytorch.nn.ReLU(),
                        mytorch.nn.ReLU(),
                        mytorch.nn.ReLU(),
                        mytorch.nn.ReLU(),
                        mytorch.nn.ReLU() ]

    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

        return None
```

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

The MLP4 topology is visualized in Figure L. The network is displayed vertically to fit on the page. You must use the diagram to deduce what the model specification is for the linear layers.

9.3.1 MLP Forward Equations (Hidden Layers = 4)

$$Z_i = A_i W_i + \iota b_i \qquad \in \mathbb{R}^{N \times C_{i+1}} \tag{68}$$

$$A_{i+1} = f_i(Z_i) \qquad \in \mathbb{R}^{N \times C_{i+1}} \tag{69}$$

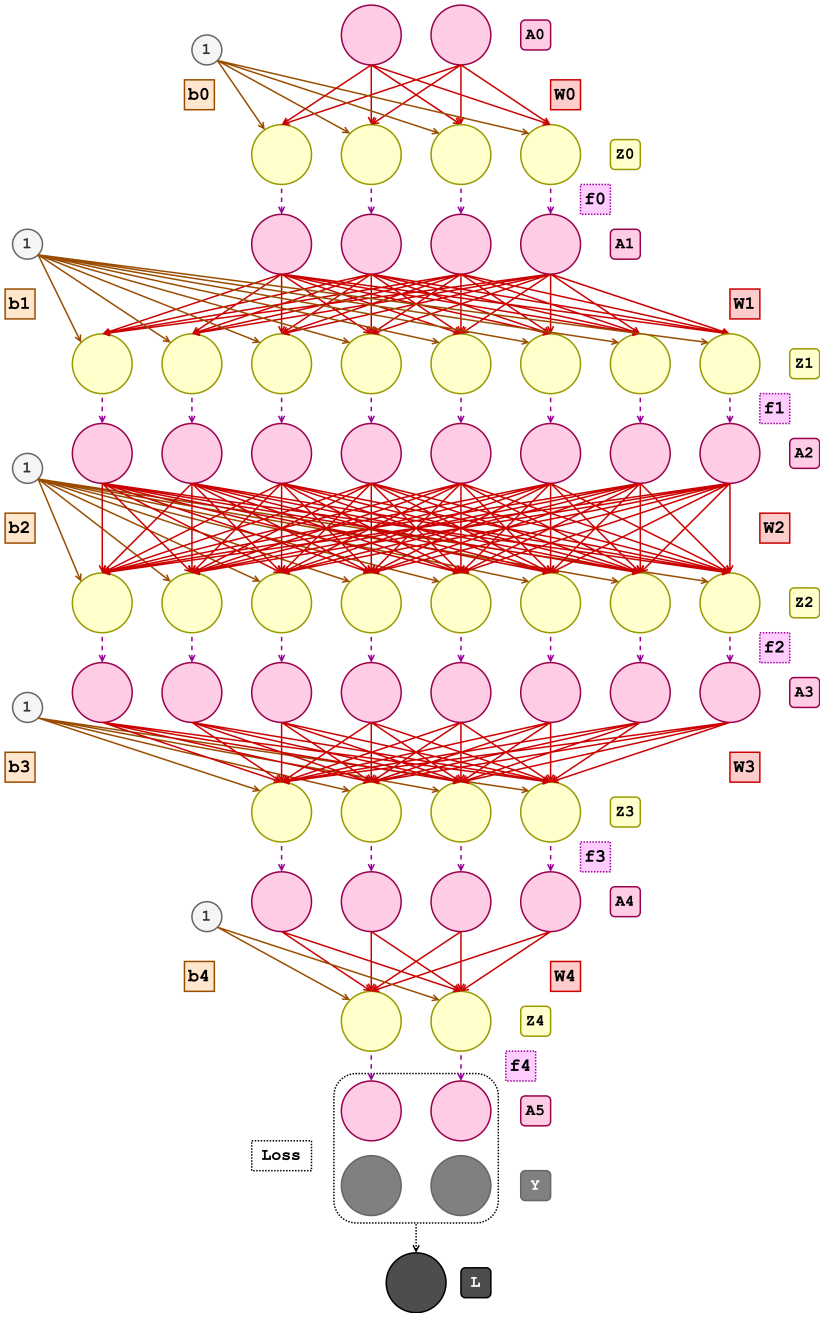

Figure M: MLP 4 Example Topology (Hidden Layers = 4)

9.3.2 MLP Backward Equations (Hidden Layers = 4)

$$\frac{\partial A_{i+1}}{\partial Z_i} = \frac{\partial}{\partial Z_i} f_i(Z_i) \qquad \in \mathbb{R}^{N \times C_{i+1}} \tag{70}$$

$$\frac{\partial L}{\partial Z_i} = \frac{\partial L}{\partial A_{i+1}} \odot \frac{\partial A_{i+1}}{\partial Z_i} \qquad \in \mathbb{R}^{N \times C_{i+1}} \tag{71}$$

$$\frac{\partial L}{\partial A_i} = \frac{\partial L}{\partial Z_i} \cdot \left(\frac{\partial Z_i}{\partial A_i} \right)^T \qquad \in \mathbb{R}^{N \times C_i} \tag{72}$$

$$\frac{\partial L}{\partial W_i} = \left(\frac{\partial L}{\partial Z_i} \right)^T \cdot \left(\frac{\partial Z_i}{\partial W_i} \right) \qquad \in \mathbb{R}^{C_{i+1} \times C_i} \tag{73}$$

$$\frac{\partial L}{\partial b_i} = \left(\frac{\partial L}{\partial Z_i} \right)^T \cdot \left(\frac{\partial Z_i}{\partial b_i} \right) \qquad \in \mathbb{R}^{C_{i+1} \times 1} \tag{74}$$

$$\tag{75}$$

10 Regularization [20 points]

10.1 Batch Normalization [mytorch.nn.BatchNorm1d]

In this section, your task is to implement the forward and backward attribute functions of the BatchNorm1d class.

The **appendix** has very detailed information necessary to complete the forward and backward functions of batch norm. Please consider the following class structure.

```
class BatchNorm1d:

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

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

        self.Z         = None
        self.NZ        = None
        self.BZ        = None

        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.M         = np.zeros((1, num_features))
        self.V         = np.ones((1, num_features))

        # inference parameters
        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 are we in the inference phase.
        So see what values you need to recompute when eval is True.
        """

        if eval:
            # TODO
            return # TODO

        self.Z         = Z
        self.N         = # TODO

        self.M         = np.mean # TODO
        self.V         = np.var  # TODO
        self.NZ        = # TODO
        self.BZ        = # TODO

        self.running_M = # TODO
        self.running_V = # TODO
```

```
        return self.BZ

    def backward(self, dLdBZ):

        self.dLdBW  = # TODO
        self.dLdBb  = # TODO

        dLdNZ       = # TODO
        dLdV        = # TODO
        dLdM        = # TODO

        dLdZ        = # TODO

        return  dLdZ
```

As you can see, BatchNorm has initialization, forward, and backward attribute functions. Immediately once the class is instantiated, the code in `__init__` is run. The initialization phase using `__init__` includes:

- As arguments, BatchNorm expects `num_features` and `alpha`.
- As attributes, BatchNorm stores `alpha, eps, Z, NZ, BZ, BW, dLdBW, Bb, dLdBb, M, V, running_M, running_V`.

In forward, we differentiate the training phase and the inference phase by the variable `eval`. We calculate all necessary variables and keep track of the running mean and variance (useful for inference). The attribute function `forward` includes:

- As arguments, forward expects input `Z` and `eval`.
- As attributes, forward stores `Z, N, M, V, NZ, BZ, running_M, running_V`.
- As an output, forward returns `BZ`. Pay attention to the difference in calculation between the training and inference phase.

In backward, we calculate multiple gradients needed for optimization. The attribute function `backward` includes:

- As an argument, backward expects input `dLdBZ`.
- As attributes, backward stores `dLdBW, dLdBb`.
- As an output, backward returns `dLdZ`.

To facilitate understanding, we have organized a table describing all relevant variables.

Table 5: Activation Function Components

Code Name	Math	Type	Shape	Meaning
`N`	N	scalar	-	number of observations
`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	-	the noise added to the variance
`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	$1 \times C$	Per feature mean
`V`	σ^2	matrix	$1 \times C$	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`	W	matrix	$1 \times C$	Weight parameters
`Bb`	b	matrix	$1 \times C$	Bias parameters
`dLdBW`	$\partial L/\partial W$	matrix	$1 \times C$	how changes in weights affect loss
`dLdBb`	$\partial L/\partial b$	matrix	$1 \times C$	how changes in bias 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

10.1.1 Batch Normalization Forward Equations

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} Z_{ij} \qquad j = 1, ..., N \tag{76}$$

$$(\sigma_j^2) = \frac{1}{N}\sum_{i=1}^{N}(Z_{ij} - \mu_j)^2 \qquad j = 1, ..., N \tag{77}$$

$$\hat{Z} = \frac{Z - \mu}{\sqrt{\sigma^2 + \epsilon}} \qquad \in \mathbb{R}^{N \times C} \tag{78}$$

$$\tilde{Z} = W\hat{Z} + b \qquad \in \mathbb{R}^{N \times C} \tag{79}$$

μ_j and σ_j^2 refer to the jth element of μ and σ^2. Z_{ij} refers to the element at the ith row and jth column of Z.

10.1.2 Batch Normalization Inference Equations

$$E[Z] = \alpha * E[Z] + (1 - \alpha) * \mu \tag{80}$$

$$Var[Z] = \alpha * Var[Z] + (1 - \alpha) * \sigma^2 \tag{81}$$

During training (and only during training), your forward method should be maintaining a running average of the mean and variance. These running averages should be used during inference. (Check the **Inference** section of appendix under Batch Norm for why we use them). You will need to manually pass the `eval` value when using batch norm in your network.

10.1.3 Batch Normalization Backward Equations

$$\frac{\partial L}{\partial \hat{Z}} = \frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \hat{Z}} = \frac{\partial L}{\partial \tilde{Z}} \odot W \tag{82}$$

$$\left(\frac{\partial L}{\partial b}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \beta}\right)_{ij} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}}\right)_{ij} \qquad j = 1, ..., N \tag{83}$$

$$\left(\frac{\partial L}{\partial W}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial W}\right)_{ij} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \odot \hat{Z}\right)_{ij} \qquad j = 1, ..., N \tag{84}$$

$$(\frac{\partial L}{\partial \sigma^2})_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \hat{Z}} \frac{\partial \hat{Z}}{\partial \sigma^2}\right)_{ij} \qquad j = 1, ..., N \tag{85}$$

All other useful derivatives with derivations for batch norm can be found in the Appendix.

Appendix

A Batch Normalization

Batch Normalization (commonly referred to as "BatchNorm") is a wildly successful and simple technique for accelerating training and learning better neural network representations. The general motivation of BatchNorm is the non-stationarity of unit activity during training that requires downstream units to adapt to a non-stationary input distribution. This co-adaptation problem, which the paper authors refer to as *internal covariate shift*, significantly slows learning.

Just as it is common to whiten training data (standardize and de-correlate the covariates), we can invoke the abstraction of a neural network as a hierarchical set of feature filters and consider the activity of each layer to be the covariates for the subsequent layer and consider whitening the activity over all training examples after each update. Whitening layer activity across the training data for each parameter update is computationally infeasible, so instead we make some (large) assumptions that end up working well anyway. The main assumption we make is that the activity of a given unit is independent of the activity of all other units in a given layer. That is, for a layer l with m units, individual unit activities (consider each a random variable) $\mathbf{x} = \{\mathbf{x}^{(1)}, \ldots, \mathbf{x}^{(m)}\}$ are independent of each other – $\{\mathbf{x}^{(1)} \perp \ldots \mathbf{x}^{(k)} \ldots \perp \mathbf{x}^{(m)}\}$.

Under this independence assumption, the covariates are not correlated and therefore we only need to normalize the individual unit activities. Since it is not practical in neural network optimization to perform updates with a full-batch gradient, we typically use an approximation of the "true" full-batch gradient over a subset of the training data. We make the same approximation in our normalization by approximating the mean and variance of the unit activities over this same subset of the training data.

For a training set $\mathcal{X}$ with n examples, we partition it into n/m batches $\mathcal{B}$ of size N. Consider μ to be the mean and σ^2 the variance of a unit's activity over one batch of the training data (size N). For an arbitrary unit k, we compute the batch statistics μ and σ^2 and normalize as follows (σ^2 is added with $\epsilon = 1e-8$ such that we do not divide by zero):

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} Z_{ij} \qquad j = 1, ..., N \tag{86}$$

$$(\sigma_j^2) = \frac{1}{N}\sum_{i=1}^{N}(Z_{ij} - \mu_j)^2 \qquad j = 1, ..., N \tag{87}$$

$$\hat{Z} = \frac{Z - \mu}{\sqrt{\sigma^2 + \epsilon}} \qquad \in \mathbb{R}^{N \times C} \tag{88}$$

$$\tag{89}$$

where μ_j and σ_j^2 refer to the jth element of μ and σ^2. Z_{ij} refers to the element at the ith row and jth column of Z.

A significant issue posed by simply normalizing individual unit activity across batches is that it limits the set of possible network representations. A way around this is to introduce a set of learnable parameters for each unit that ensure the BatchNorm transformation can be learned to perform an identity transformation. To do so, these per-unit learnable parameters W and b for the kth unit, rescale and reshift the normalized unit activity. Thus the output of the BatchNorm transformation for the data samples, $\tilde{Z}$ is given as follows,

$$\tilde{Z} \leftarrow W\hat{Z} + b \tag{90}$$

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.

$$\frac{\partial L}{\partial \hat{Z}} = \frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \hat{Z}} = \frac{\partial L}{\partial \tilde{Z}} \odot W \tag{91}$$

$$\left(\frac{\partial L}{\partial b}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \beta}\right)_{ij} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}}\right)_{ij} \qquad j = 1, ..., N \tag{92}$$

$$\left(\frac{\partial L}{\partial W}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial W}\right)_{ij} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \odot \hat{Z}\right)_{ij} \qquad j = 1, ..., N \tag{93}$$

$$\left(\frac{\partial L}{\partial \sigma^2}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \hat{Z}} \frac{\partial \hat{Z}}{\partial \sigma^2}\right)_{ij} \tag{94}$$

$$= \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \hat{Z}} \frac{\partial}{\partial \sigma^2} \left[(Z - \mu)(\sigma^2 + \epsilon)^{-\frac{1}{2}}\right]\right)_{ij} \tag{95}$$

$$= -\frac{1}{2} \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \hat{Z}} \odot (Z - \mu) \odot (\sigma^2 + \epsilon)^{-\frac{3}{2}}\right)_{ij} \qquad j = 1, ..., N \tag{96}$$

$$\frac{\partial L}{\partial \mu} = \sum_{i=1}^{N} \frac{\partial L}{\partial \hat{Z}_i} \frac{\partial \hat{Z}_i}{\partial \mu} \tag{97}$$

where $\hat{Z}_i$ refers to the ith sample in $\hat{Z}$.

Solve for $\frac{\partial \hat{Z}_i}{\partial \mu}$

$$\frac{\partial \hat{Z}_i}{\partial \mu} = \frac{\partial}{\partial \mu}\left[(Z_i - \mu)(\sigma^2 + \epsilon)^{-\frac{1}{2}}\right] \tag{98}$$

$$= -(\sigma^2 + \epsilon)^{-\frac{1}{2}} + (Z_i - \mu) \odot \frac{\partial}{\partial \mu}\left[(\sigma^2 + \epsilon)^{-\frac{1}{2}}\right] \tag{99}$$

$$= -(\sigma^2 + \epsilon)^{-\frac{1}{2}} + (Z_i - \mu) \odot \frac{\partial}{\partial \mu}\left[\left(\frac{1}{N}\sum_{i=1}^{N} (Z_i - \mu)^2 + \epsilon\right)^{-\frac{1}{2}}\right] \tag{100}$$

$$= -(\sigma^2 + \epsilon)^{-\frac{1}{2}} \tag{101}$$

$$- \frac{1}{2}(Z_i - \mu) \odot \left[\left(\frac{1}{N}\sum_{i=1}^{N} (Z_i - \mu)^2 + \epsilon\right)^{-\frac{3}{2}} \frac{\partial}{\partial \mu}\left(\frac{1}{N}\sum_{i=1}^{N} (Z_i - \mu)^2\right)\right]$$

$$= -(\sigma^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2}(Z_i - \mu) \odot \left(\sigma^2 + \epsilon\right)^{-\frac{3}{2}} \left(-\frac{2}{N}\sum_{i=1}^{N} (Z_i - \mu)\right) \tag{102}$$

Now sub this expression for $\frac{\partial \hat{Z}_i}{\partial \mu}$ into $\frac{\partial L}{\partial \mu} = \sum_{i=1}^{N} \frac{\partial L}{\partial \hat{Z}_i} \frac{\partial \hat{Z}_i}{\partial \mu}$

$$\frac{\partial L}{\partial \mu} = -\sum_{i=1}^{N} \frac{\partial L}{\partial \hat{Z}_i}(\sigma^2 + \epsilon)^{-\frac{1}{2}} - \frac{1}{2}\sum_{i=1}^{N} \frac{\partial L}{\partial \hat{Z}_i} \odot (Z_i - \mu) \odot \left(\sigma^2 + \epsilon\right)^{-\frac{3}{2}} \left(-\frac{2}{N}\sum_{i=1}^{N} (Z_i - \mu)\right) \tag{103}$$

Notice that part of the expression in the second term is just $\frac{\partial L}{\partial \sigma^2}$

$$\begin{aligned}\frac{\partial L}{\partial \mu} &= -\sum_{i=1}^{N} \frac{\partial L}{\partial \hat{Z}_i}(\sigma^2+\epsilon)^{-\frac{1}{2}} + \frac{\partial L}{\partial \sigma^2}\left(-\frac{2}{N}\sum_{i=1}^{N}(Z_i-\mu)\right) \\ &= -\sum_{i=1}^{N} \frac{\partial L}{\partial \hat{Z}_i}(\sigma^2+\epsilon)^{-\frac{1}{2}} - \frac{2}{N}\frac{\partial L}{\partial \sigma^2}\sum_{i=1}^{N}(Z_i-\mu)\end{aligned}$$

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

$$\frac{\partial L}{\partial Z_i} = \frac{\partial L}{\partial \hat{Z}_i}\frac{\partial \hat{Z}}{\partial Z_i} \tag{104}$$

$$= \frac{\partial L}{\partial \hat{Z}_i}\left[\frac{\partial \hat{Z}_i}{\partial Z_i} + \frac{\partial \hat{Z}_i}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial Z_i} + \frac{\partial \hat{Z}}{\partial \mu}\frac{\partial \mu}{\partial Z_i}\right] \tag{105}$$

$$= \frac{\partial L}{\partial \hat{Z}_i}\frac{\partial \hat{Z}_i}{\partial Z_i} + \frac{\partial L}{\partial \hat{Z}_i}\frac{\partial \hat{Z}_i}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial Z_i} + \frac{\partial L}{\partial \hat{Z}_i}\frac{\partial \hat{Z}_i}{\partial \mu}\frac{\partial \mu}{\partial Z_i} \tag{106}$$

$$= \frac{\partial L}{\partial \hat{Z}_i}\frac{\partial \hat{Z}_i}{\partial Z_i} + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial Z_i} + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial Z_i} \tag{107}$$

$$= \frac{\partial L}{\partial \hat{Z}_i}\left[\frac{\partial}{\partial Z_i}\left((Z_i-\mu)(\sigma^2+\epsilon)^{-\frac{1}{2}}\right)\right] + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial Z_i} + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial Z_i} \tag{108}$$

$$= \frac{\partial L}{\partial \hat{Z}_i}\left[(\sigma^2+\epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L}{\partial \sigma^2}\frac{\partial \sigma^2}{\partial Z_i} + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial Z_i} \tag{109}$$

$$= \frac{\partial L}{\partial \hat{Z}_i}\left[(\sigma^2+\epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L}{\partial \sigma^2}\left[\frac{\partial}{\partial Z_i}\left(\frac{1}{N}\sum_{j=1}^{N}(Z_j-\mu)^2\right)\right] + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial Z_i} \tag{110}$$

$$= \frac{\partial L}{\partial \hat{Z}_i}\left[(\sigma^2+\epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L}{\partial \sigma^2}\left[\frac{2}{N}(Z_i-\mu)\right] + \frac{\partial L}{\partial \mu}\frac{\partial \mu}{\partial Z_i} \tag{111}$$

$$= \frac{\partial L}{\partial \hat{Z}_i}\left[(\sigma^2+\epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L}{\partial \sigma^2}\left[\frac{2}{N}(Z_i-\mu)\right] + \frac{\partial L}{\partial \mu}\left[\frac{\partial}{\partial Z_i}\left(\frac{1}{N}\sum_{j=1}^{N}Z_j\right)\right] \tag{112}$$

$$= \frac{\partial L}{\partial \hat{Z}_i}\left[(\sigma^2+\epsilon)^{-\frac{1}{2}}\right] + \frac{\partial L}{\partial \sigma^2}\left[\frac{2}{N}(Z_i-\mu)\right] + \frac{\partial L}{\partial \mu}\left[\frac{1}{N}\right] \tag{113}$$

In summary, we have derived the following quantities required in the forward and backward computation for BatchNorm:

Forward

$$\mu_j = \frac{1}{N}\sum_{i=1}^{N} Z_{ij} \qquad j = 1, ..., N \tag{114}$$

$$\sigma_j^2 = \frac{1}{N}\sum_{i=1}^{N}(Z_{ij}-\mu_j)^2 \qquad j = 1, ..., N \tag{115}$$

$$\hat{Z} = \frac{Z-\mu}{\sqrt{\sigma^2+\epsilon}} \qquad \in \mathbb{R}^{N\times C} \tag{116}$$

$$\tilde{Z} = W\hat{Z} + b \qquad \in \mathbb{R}^{N\times C} \tag{117}$$

μ_j and σ_j^2 refer to the jth element of μ and σ^2. Z_{ij} refers to the element at the ith row and jth column of Z.

Backward

$$\frac{\partial L}{\partial \hat{Z}} = \frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \hat{Z}} = \frac{\partial L}{\partial \tilde{Z}} \odot W \tag{118}$$

$$\left(\frac{\partial L}{\partial b}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial \beta}\right)_{ij} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}}\right)_{ij} \qquad j = 1, ..., N \tag{119}$$

$$\left(\frac{\partial L}{\partial W}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \frac{\partial \tilde{Z}}{\partial W}\right)_{ij} = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \tilde{Z}} \odot \hat{Z}\right)_{ij} \qquad j = 1, ..., N \tag{120}$$

$$\left(\frac{\partial L}{\partial \sigma^2}\right)_j = \sum_{i=1}^{N} \left(\frac{\partial L}{\partial \hat{Z}} \frac{\partial \hat{Z}}{\partial \sigma^2}\right)_{ij} \qquad j = 1, ..., N \tag{121}$$

Inference

$$E[Z] = \alpha * E[Z] + (1 - \alpha) * \mu \tag{122}$$

$$Var[Z] = \alpha * Var[Z] + (1 - \alpha) * \sigma^2 \tag{123}$$

We cannot calculate the mean and variance during inference, hence we need to maintain an estimate of the mean and variance to use when calculating the norm of Z ($\hat{Z}$) at test time. You need to calculate the running average at training time, because you really want to find an estimate for the overall covariate shifts over the entire data. Running averages give you an estimate of the overall covariate shifts. At test time you typically have only one test instance, so if you use the test data itself to compute means and variances, you'll wipe the data out (mean will be itself, var will be inf). Thus, you use the global values (obtained as running averages) from the training data at test time. The running mean is defined as $E[Z]$ and the running variance is defined as $Var[Z]$ above.