NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

EE4414: Week 1

By: Zhi-Wei Tan

Course coordinator: Prof Yap

Learning objective

- Familiarity with PyTorch:
  - Reading documentation
  - Common parameters
  - Construct a vector and matrix
  - Perform the following operations:
    - Manipulate elements
    - Stack and concatenate
    - Transpose an N-dimensional matrix
    - Element-wise multiplication and broadcasting
    - Matrix multiplication

Introduction to PyTorch

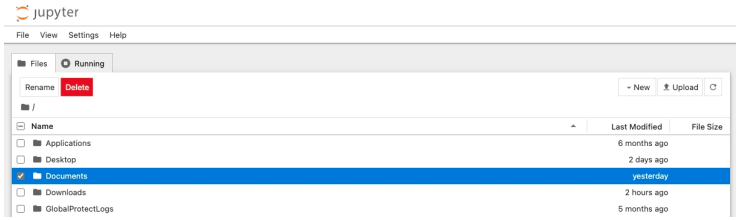
- Widely used in academia and research industry (OpenAI, Microsoft, etc)
- Code can easily run on a GPU without much modification
- Performs backpropagation for you! (will come back to that in a later part)
- Comes with pre-trained models
- Other similar work includes Tensorflow

Setting up Jupyter notebook

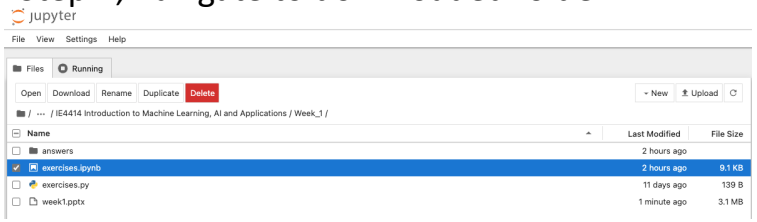
1. Download week 1 from the Blackboard; extract if needed, and it should contain a slide and an exercises.ipynb file
2. In search, type "**cmd**" without the quotes to open a command prompt.
3. Type "**jupyter notebook**" without the quotes in the command prompt, and it should open a browser
4. Navigate to the **downloaded folder** in Step 1 and open the **exercises.ipynb**
5. Run the first cell
6. If successful, you should not see any red text
7. Once completed, smile and help others.

At any time, raise your hand if you have trouble.

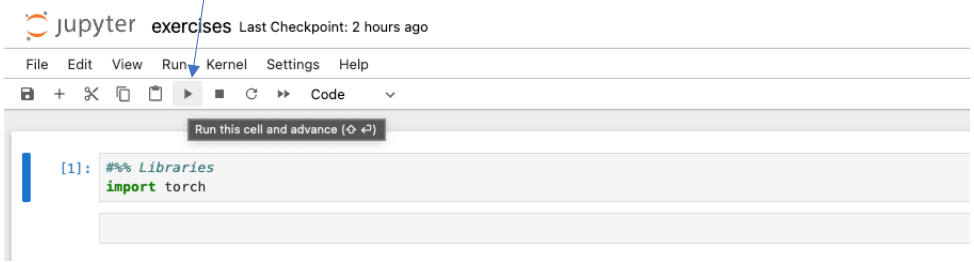
After step 3, A browser pops up with Jupyter notebook



Step 4, navigate to downloaded folder



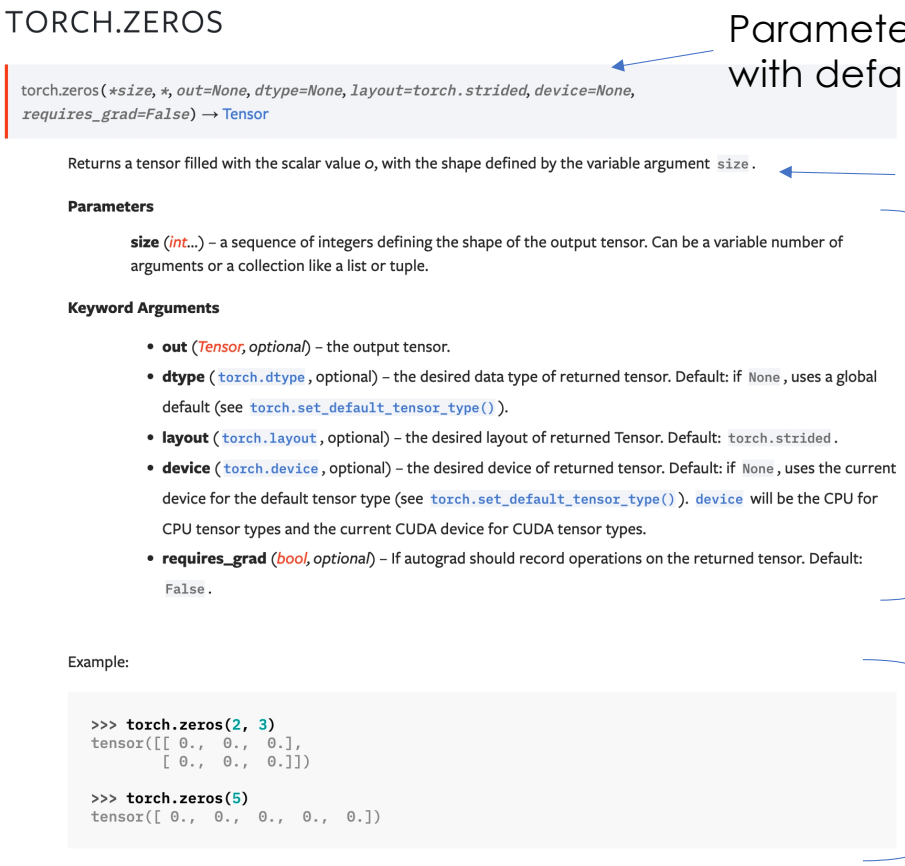
Step. 5



First exercise: Creating a vector and matrix

- In PyTorch, creating a vector (a 1-D matrix) and matrix is simple.
- You will create a **column zero** vector with a **shape** of **3 by 1**, **real data type** of **64-bit floating point** in the **CPU**, and this is a data point in a dataset.
- To do so, we look at the documentation of PyTorch and look for the command **torch.zeros**

Intepreting the documentation



TORCH.ZEROS

torch.zeros(*size, *, out=None, dtype=None, layout=torch.strided, device=None, requires_grad=False) → Tensor

Returns a tensor filled with the scalar value 0, with the shape defined by the variable argument `size`.

Parameters

size (*int...*) – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Keyword Arguments

- **out** (*Tensor, optional*) – the output tensor.
- **dtype** (`torch.dtype`, optional) – the desired data type of returned tensor. Default: if `None`, uses a global default (see `torch.set_default_tensor_type()`).
- **layout** (`torch.layout`, optional) – the desired layout of returned Tensor. Default: `torch.strided`.
- **device** (`torch.device`, optional) – the desired device of returned tensor. Default: if `None`, uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.
- **requires_grad** (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Example:

```
>>> torch.zeros(2, 3)
tensor([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])

>>> torch.zeros(5)
tensor([ 0.,  0.,  0.,  0.,  0.])
```

Parameters (those with = comes with default values)

Brief description of its functionality

Further details of the parameters

Quick way to understand its functionality via examples

Tidbits: You can write and generate your own Documentation using libraries like Sphinx!

Explaining important parameters: size

size (*int*...) – a sequence of integers defining the shape of the output tensor. Can be a variable number of arguments or a collection like a list or tuple.

Used to define the shape of the output

Three ways to define it:

- Sequence: …
- List: […]
- Tuples: (…)

Examples:

- Sequence: torch.zeros(10, 10, 3)
- List: torch.zeros([10, 10, 3])
- Tuple: torch.zeros((10, 10, 3))

Explaining important parameters: data type

dtype (`torch.dtype` , optional) – the desired data type of returned tensor. Default: if `None` , uses a global default (see `torch.set_default_tensor_type()`).

Used to define the **data type** (short form is dtype)

Takes in type of torch.dtype and is optional, and it defaults to a global default

Some commonly used data types (click torch.dtype in the documentation to see full list) :

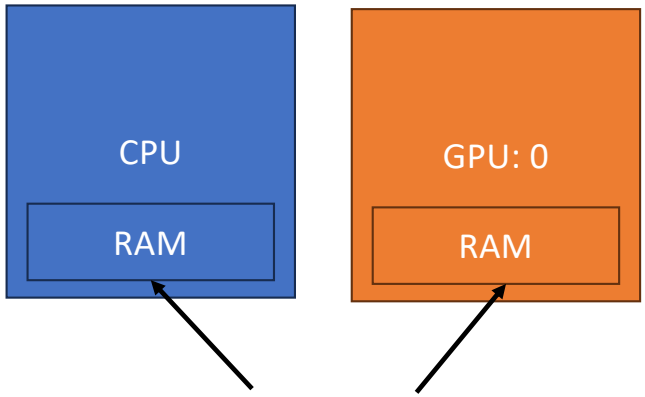
- torch.float — 32-bit floating point
- torch.double — _____
- torch.cfloat — _____
- _____ — 16-bit floating point

Explaining important parameters: device

device (`torch.device` , optional) – the desired device of returned tensor. Default: if `None` , uses the current device for the default tensor type (see `torch.set_default_tensor_type()`). `device` will be the CPU for CPU tensor types and the current CUDA device for CUDA tensor types.

Tidbits: CUDA is a C++ programming language for GPU

- Sets which devices the data is kept
- Allows one to easily set between CPU and GPU
- Generally, transferring data between the CPU's RAM and the GPU takes time and can be time-consuming for large amounts of data!
- GPU generally processes data faster than CPU but with a smaller RAM size
- Some commonly used inputs:
  - torch.cuda.device(0) — RAM of GPU with index 0
  - torch.device("cpu") — RAM of CPU
  - torch.cuda.current_device() — RAM of GPU in current device



Explaining important parameters: requires_grad

requires_grad (*bool, optional*) – If autograd should record operations on the returned tensor. Default: `False`.

Used to define if gradient should be recorded for this parameter

Generally, this is set to **True** for learnable and **False** for non-learnable parameters.

Examples of learnable parameters:
- Weights and biases of algorithms

Examples of non-learnable parameters:
- Datasets
- Fixed hyperparameters (i.e., learning rate)

We will discuss this further in Week 2, so no worries . ☺

Ex. 1.1: Creating a zero vector

- Create a **column zero** vector with a **shape** of **3 by 1**, **real** data type of **64-bit floating point** in the **CPU**, and this is a **data point** in a **dataset**

Ex 1.1: Answer

a = torch.zeros((3,1), dtype=torch.double, device=torch.device("cpu"), requires_grad=False)

a = torch.zeros((3,1), dtype=torch.float64, device="cpu", requires_grad=False)

…

.

Academic writing

- In general, a column vector is defined using a bold-faced small letter via

$$\mathbf{a} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}$$

- This vector was created in ex. 1
- We will use this notation henceforth

Ex. 1.2: Creating a column vector

- Create the following column vector using the function **torch.tensor**:

$$\mathbf{a} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix}$$

- For **32-bit floating-point**
- Has a shape of **3 by 1**
- Print out its shape to check!

Answer

- a = torch.tensor([1,3,5], dtype=torch.float)
- a = torch.tensor([[1],[3],[5]], dtype=torch.float) # preferred

- Elements in the outer-most bracket are in the first dimension. Subsequently, the inner bracket is the higher dimension.

- For the second answer, there are **three** elements in the **outer**-most bracket (first dimension) and **one** element in the **inner** bracket (second dimension). Hence, the shape of **3** by **1**

Ex. 1.3: Creating a row vector

- Create the following **row** vector using the function torch.tensor:

$$\mathbf{b} = \begin{bmatrix} 1, 3, 5 \end{bmatrix}$$

- For **32-bit floating-point**
- The row vector has a shape of **1 by 3**

Answer and explanation

- b = torch.tensor([[1,3,5]], dtype=torch.float)

- There is **one** element in the outer-most bracket (first dimension) and **three** elements in the inner bracket (second dimension). Hence, the shape of **1** by **3.**

- What is the shape of torch.tensor([[1,2],[3,4],[5,6]])?

Ex. 1.4: Creating a matrix

- Create the following matrix **A** using torch.tensor

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \in \mathbb{R}^{3\times 4}$$

- Remember for torch.tensor: Elements in the outer-most bracket are in the first dimension. Subsequently, the inner bracket is the higher dimension.

Answer

- A = torch.tensor([[1,2,3,4],[5,6,7,8],[9,10,11,12]])

Ex. 1.5: Creating a 3-D matrix

- Create the following 3-D matrix **A** using torch.tensor

$$\underline{\mathbf{A}} = \left[\begin{bmatrix} 1 & 2 \\ 5 & 6 \\ 9 & 10 \end{bmatrix} \begin{bmatrix} 3 & 4 \\ 7 & 8 \\ 11 & 12 \end{bmatrix} \right] \in \mathbb{R}^{2\times 3\times 2}$$

Answer

- A_ = torch.tensor([[[1,2],[5,6],[9,10]],[[3,4],[7,8],[11,12]]])

Other ways to create vectors and matrices

- Similar to torch.zeros:
  - torch.ones — All ones
  - torch.arange — (e.g., 1,2,3,4…)
  - torch.eye — Identity
  - torch.rand — Random elements with values between 0 and 1

Ex. 2: Operations

- In this section, we will cover the following operations that are commonly used:
  1. Indexing and slicing
  2. Stack and concatenate
  3. Transpose
  4. Element-wise multiplication
  5. Matrix multiplication
  6. Moving from CPU to GPU
  7. Convert float variables to double

Ex. 2.1.1: Manipulating elements

- Update the vector $\mathbf{z}$ via indexing:

$$\mathbf{z} = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \qquad \mathbf{z} = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

- Indexing in Pytorch uses the [] brackets
- Pytorch employs zero-indexing:
  - First-element is in z[0], second in z[1], and so on and so forth

Ex. 2.1.2: Extracting an element from a matrix

- Update matrix **Z** via indexing:

$$\mathbf{Z} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \qquad \mathbf{Z} = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

- To obtain a single element from the matrix, we employ Z[i,j], where i and j are indices corresponding to the row and column, respectively.

Ex 2.1.2: Answer

- Z[0,2]=1

$$\mathbf{Z} = \begin{matrix} \\ 0 \\ 1 \\ \end{matrix} \begin{matrix} \begin{matrix} 0 & 1 & \mathbf{2} & 3 \end{matrix} \\ \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{matrix}$$

Ex. 2.1.3: Extracting an element from a 3D matrix

- Update $\underline{\mathbf{Z}}$:

$$\underline{\mathbf{Z}} = \left[\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}\right] \in \mathbb{R}^{2\times 3\times 4} \qquad \underline{\mathbf{Z}} = \left[\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}\right]$$

- To obtain a single element from the 3D matrix, we employ $\underline{\mathbf{Z}}$[i,j,k], where i is the index corresponding to the 2D matrix with indices j and k corresponding to its row and column, respectively.

$\mathbf{Z} =$

Ex 2.1.3: Answer

- Z_[1,1,2]

$$\underline{\mathbf{Z}} = \begin{matrix} \\ \\ \\ \end{matrix} \begin{matrix} 0 & 1 & \mathbf{2} & 3 \end{matrix} \\ \left[\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right] \begin{matrix} 0 \\ \mathbf{1} \end{matrix}$$

Ex. 2.1.4: Slicing I

- Not only single elements, vectors, or matrices of various dimensions can also be extracted.

- For e.g. calling Z_[1, :, :] (equivalent to Z_[1]) in Ex. 2.1.3 gives a matrix of shape **3** by **4**

$$\underline{\mathbf{Z}} = \left[\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right] \in \mathbb{R}^{2\times 3\times 4}$$

- Now obtain the vector in the red box from Z_

Answer and review

$$\underline{\mathbf{Z}} = \left[\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right] \in \mathbb{R}^{2\times 3\times 4}$$

- Ans:
- Z_[1, :, 2]
- Z_[1,:, 2:3]
- Here, ':' means slicing for everything in that dimension.
- Note that this is equivalent to Z_[1, 0:3, 2]
- Here, 0:**3** means slicing for index 0, 1, and **2**
- This is important for the next exercise!

Ex. 2.1.5: Slicing II

- Obtain the matrix in the red box from $\underline{\mathbf{Z}}$

$$\underline{\mathbf{Z}} = \left[\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right] \in \mathbb{R}^{2 \times 3 \times 4}$$

Answer and review

- Ans: Z_[1, :, 1:3]

$$\underline{\mathbf{Z}} = \begin{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{2\times 3\times 4}$$

0 1 2

- Here, 1:3 means slicing for index 1 and 2

Ex. 2.1.6: Slicing III

- Obtain the matrix in the red box from $\underline{\mathbf{Z}}$

$$\underline{\mathbf{Z}} = \left[\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right] \in \mathbb{R}^{2 \times 3 \times 4}$$

Answer and review

- Ans: Z_[1, :, 1:]

$$\underline{\mathbf{Z}} = \left[\begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \right] \in \mathbb{R}^{2\times 3\times 4}$$

0 1 2 3

- Here, '1:' means slicing from index 1 to the maximum possible index

- Can you infer the shape after slicing?

Ex. 2.2.1 Stack vectors to matrices

- Employ the torch.stack function on the following vectors:

$$\mathbf{a} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \in \mathbb{R}^{3\times 1} \qquad \mathbf{b} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} \in \mathbb{R}^{3\times 1}$$

Answer and review

- torch.stack([a, b], dim=0)

- Note: The two vectors must be of the same shape!
- What is the shape of the output?
- How can stack on a different dimension?
- How many vectors can I stack?
- Can vectors with different data type stack?

Ex. 2.2.2 Concatenate vectors

- Employ the torch.cat function on the following vectors into a **6** by **1**:

$$\mathbf{a} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \in \mathbb{R}^{3\times 1} \qquad \mathbf{b} = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} \in \mathbb{R}^{3\times 1}$$

Answer

- torch.cat([a, b], dim=0)
- What is the main difference between torch.stack and torch.cat?

Ex. 2.3.1: Transpose

- The difference between vectors **a** and **b** is the shape
- Employ the torch.transpose function on **a** to obtain vector **b**

$$\mathbf{a} = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \in \mathbb{R}^{3\times 1} \qquad \mathbf{b} = \begin{bmatrix} 1, 3, 5 \end{bmatrix} \in \mathbb{R}^{1\times 3}$$

Ex. 2.3.1 and Ex. 2.3.2

torch.transpose(a, 0,1)

a.T

a.t()

torch.transpose(A, 0,1)

A.T

Ex. 2.3.2: Transpose II

- Transpose the matrix **A**

$$\mathbf{A} = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \in \mathbb{R}^{3\times 4} \qquad \mathbf{B} = \mathbf{A}^{\mathsf{T}} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix} \in \mathbb{R}^{4\times 3}$$

Ex. 2.3.3: Transpose III

- Transpose $\underline{\mathbf{A}}$ on its second and third dimension

$$\underline{\mathbf{A}} = \left[\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}\right] \in \mathbb{R}^{2\times 3\times 4}$$

Ex. 2.3.3

- torch.transpose(A_, 1,2)

$$\underline{\mathbf{A}} = \left[\begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix}\right] \in \mathbb{R}^{2\times 3\times 4}$$

Ex. 2.4.1: Element-wise multiplication

- Perform element-wise multiplication on the following vectors:

$$\mathbf{a} = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} \in \mathbb{R}^{3\times 1} \qquad \mathbf{b} = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} \in \mathbb{R}^{3\times 1}$$

$$\mathbf{c} = \mathbf{a} \cdot \mathbf{b} = \_\_\_\_\_ \in \mathbb{R}^{-\times-}$$

- Also called the Hadamard product and pointwise multiplication
- To perform element-wise multiplication, use '*' or torch.multiply (Note: this is not matrix multiplication)

Answers and reviews

- torch.multiply(a, b) or a * b

$$\mathbf{c} = \mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} 4 \\ 9 \\ 25 \end{bmatrix} \in \mathbb{R}^{3\times 1}$$

- For this ex., the output is of the same shape as its inputs
- Can different shapes perform multiplication?

Ex. 2.4.2: Element-wise multiplication

- Perform element-wise multiplication on the following matrices:

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix} \in \mathbb{R}^{2\times 3} \qquad \mathbf{B} = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix} \in \mathbb{R}^{2\times 3}$$

$$\mathbf{C} = \mathbf{A} \cdot \mathbf{B} = \underline{\qquad\qquad} \in \mathbb{R}^{-\times -}$$

Ex. 2.4.3: Concept of broadcasting

- Perform element-wise multiplication on the following vectors:

$$\mathbf{a} = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} \in \mathbb{R}^{3\times 1} \qquad \mathbf{a}^{\mathsf{T}} = \begin{bmatrix} 2 & 3 & 5 \end{bmatrix} \in \mathbb{R}^{1\times 3}$$

$$\mathbf{a} \cdot \mathbf{a}^{\mathsf{T}} = \_\_\_\_\_ \in \mathbb{R}^{-\times-}$$

Ex. 2.4.3: Answers and reviews

- result = a * a.T

$$\mathbf{a} \cdot \mathbf{a}^{\mathsf{T}} = \begin{bmatrix} 4 & 6 & 10 \\ 6 & 9 & 15 \\ 10 & 15 & 25 \end{bmatrix} \in \mathbb{R}^{3\times 3}$$

- Why is the result a matrix?

Broadcasting explained

$$\mathbf{a} = \begin{bmatrix} 2 \\ 3 \\ 5 \end{bmatrix} \in \mathbb{R}^{3\times 1} \qquad \mathbf{a}^{\mathsf{T}} = \begin{bmatrix} 2 & 3 & 5 \end{bmatrix} \in \mathbb{R}^{1\times 3}$$

Because 1
and the other is 3

$$\begin{bmatrix} 2 & 2 & 2 \\ 3 & 3 & 3 \\ 5 & 5 & 5 \end{bmatrix} \cdot \begin{bmatrix} 2 & 3 & 5 \\ 2 & 3 & 5 \\ 2 & 3 & 5 \end{bmatrix} = \begin{bmatrix} 4 & 6 & 10 \\ 6 & 9 & 15 \\ 10 & 15 & 25 \end{bmatrix}$$

Ex. 2.4.4: Concept of broadcasting II

- Compute the following element-wise multiplication operation without code (you can, but why not give it a try):

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix} \in \mathbb{R}^{3\times 2} \qquad \mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \in \mathbb{R}^{3\times 1}$$

$$\mathbf{A} \cdot \mathbf{a} = \_\_\_\_\_ \cdot \_\_\_\_\_$$

$$= \_\_\_\_\_ \in \mathbb{R}^{\_\times\_}$$

Ex. 2.4.4: Concept of broadcasting II

- Compute the following element-wise multiplication operation without code (you can, but why not give it a try):

$$\mathbf{A} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix} \in \mathbb{R}^{3\times 2} \qquad \mathbf{a} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \in \mathbb{R}^{3\times 1}$$

$$\mathbf{A} \cdot \mathbf{a} = \begin{bmatrix} 2 & 3 \\ 4 & 5 \\ 6 & 7 \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 2 & 2 \\ 3 & 3 \end{bmatrix}$$

$$= \begin{bmatrix} 2 & 3 \\ 8 & 10 \\ 18 & 21 \end{bmatrix} \in \mathbb{R}^{3\times 2}$$

Ex. 2.5.1: Matrix multiplication

- Perform matrix multiplication using torch.matmul and torch.transpose on the following matrices:

$$\mathbf{C} = \mathbf{A}\mathbf{B}^{\mathsf{T}}$$

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix} \in \mathbb{R}^{2\times 3} \qquad \mathbf{B} = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix} \in \mathbb{R}^{2\times 3}$$

- What is the shape of **C**?
- Is it possible to perform $\mathbf{AB}$?
- What about $\mathbf{C} = \mathbf{A}^{\mathsf{T}}\mathbf{B}$?

Ex. 2.5.1: Answers and reviews

C=torch.matmul(A, B.transpose(0,1))

C = A @ B.transpose(0,1)

$$\mathbf{A} = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix} \in \mathbb{R}^{2\times 3} \qquad \mathbf{B} = \begin{bmatrix} 2 & 3 & 4 \\ 5 & 6 & 7 \end{bmatrix} \in \mathbb{R}^{2\times 3}$$

The ways to infer the shape **C** is:

- Infer the shape of $\mathbf{B}^{\mathsf{T}} \in \mathbb{R}^{3\times 2}$
- Identify the reduction dimension $\mathbf{A} \in \mathbb{R}^{2\times 3} \quad \mathbf{B}^{\mathsf{T}} \in \mathbb{R}^{3\times 2}$
- Obtain the remaining dimension i.e., 2 by 2

Ex. 2.5.2: Matrix multiplication with broadcasting (optional)

- Perform matrix multiplication on the following matrices:

$$\underline{\mathbf{A}} = \begin{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{2\times3\times4} \qquad \mathbf{B} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix} \in \mathbb{R}^{4\times3}$$

$$\underline{\mathbf{C}} = \underline{\mathbf{A}}\mathbf{B} \in \mathbb{R}^{-\times-\times-}$$

Ex. 2.5.2: Answers and reviews

- Ans: torch.matmul(A_, B)

$$\underline{\mathbf{A}} = \begin{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{2\times 3\times 4} \qquad \mathbf{B} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix} \in \mathbb{R}^{4\times 3}$$

$$\underline{\mathbf{C}} = \underline{\mathbf{A}}\mathbf{B} \in \mathbb{R}^{2\times 3\times 3}$$

- Why the above is allowed despite the matrices having a different number of dimensions?

Ex. 2.5.2: Answers and reviews II

- Ans: Broadcasting

$$\underline{\mathbf{A}} = \begin{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \begin{bmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \end{bmatrix} \end{bmatrix} \in \mathbb{R}^{2\times 3\times 4}$$

$$\mathbf{B} = \begin{bmatrix} 1 & 5 & 9 \\ 2 & 6 & 10 \\ 3 & 7 & 11 \\ 4 & 8 & 12 \end{bmatrix} \in \mathbb{R}^{4\times 3}$$

- Pytorch focuses on the **last N dimensions** for the operation and broadcasts the earlier dimensions if needed
- For matrix multiply, **N=2**, so it checks if the last two dimensions of each matrix can perform the multiplication.
- If so, it adds dimension with a length of one to **B** to have the same number of dimensions as **A** i.e., $\underline{\mathbf{B}} \in \mathbb{R}^{1\times 4\times 3}$.
- After that, it performs broadcasting on **B** to match the first dimension of **A** via
- $$\underline{\mathbf{B}} = \begin{bmatrix} \mathbf{B} & \mathbf{B} \end{bmatrix} \in \mathbb{R}^{2\times 4\times 3}$$
- Thereafter, the matrix multiplication is performed.
- Another way to interpret this is:
  - Each of the two 2D matrices in **A** performs a matrix multiplication with **B** and the results of each are stacked in **C**.

NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

EE4414: Week 2

By: Zhi-Wei Tan

Course coordinator: Prof Yap

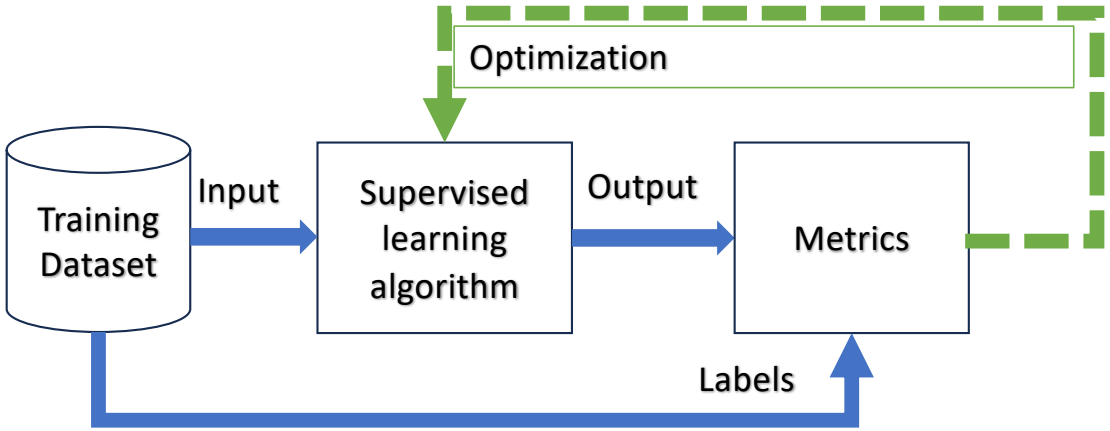
Learning objective

- Brief introduction to supervised learning
- Familiarity with PyTorch functionality relating to supervised learning and neural networks
  - Creating a linear model
  - Compute its gradient
  - Optimizing its weights
- Exercise on calculating its gradient

Supervised learning

- Supervised learning requires a dataset with **labels**, or **true** values
- Generally, it involves the following:
  - An algorithm with learnable parameters
  - A metric that decides how well the algorithm is performing
    - E.g. error, closeness
  - An optimizer to adjust/change the learnable parameter based on the metric
    - E.g. gradient descent
- Other learning methods include:
  - Unsupervised learning
  - Reinforcement learning

The training cycle for supervised learning

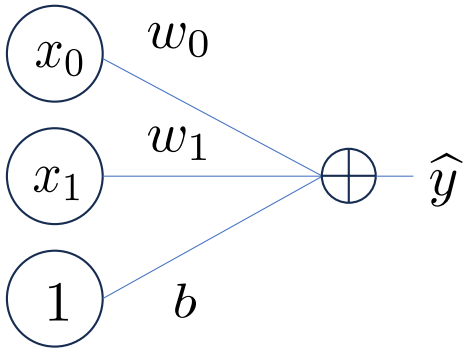


Ch. 1: Linear approaches

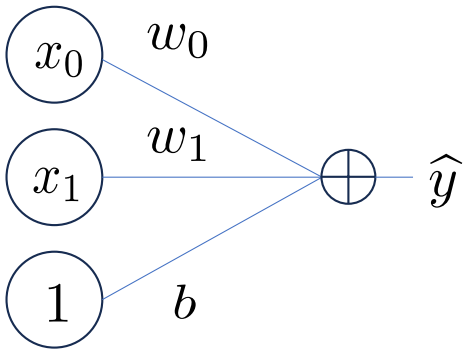
- One of the most fundamental operations in neural networks is called linear transformation or linear mapping.

- This transformation can be interpreted in multiple ways:
  - Projection, Rotation, Scaling, …
  - Decision boundary: SVM
  - Weightage of features: McCulloch–Pitts (MCP) neuron

$$\widehat{y} = w_0 x_0 + w_1 x_1 + b$$



Ex. 1.1: Construct the linear model



$$\begin{aligned}\widehat{y} &= w_0 x_0 + w_1 x_1 + b \\ &= \mathbf{w}^\mathsf{T}\mathbf{x} + b\end{aligned}$$

Use torch.tensor to create the weights $\mathbf{w} \in \mathbb{R}^{2\times 1}$ and bias b using torch.tensor given $w_0 = 0.4$, $w_1 = 0.1$, and $b = 1$. Similarly, create the input features $\mathbf{x} \in \mathbb{R}^{2\times 1}$ given $x_0 = 0.5$ and $x_1 = 0.4$. Thereafter, obtain $\widehat{y}$.

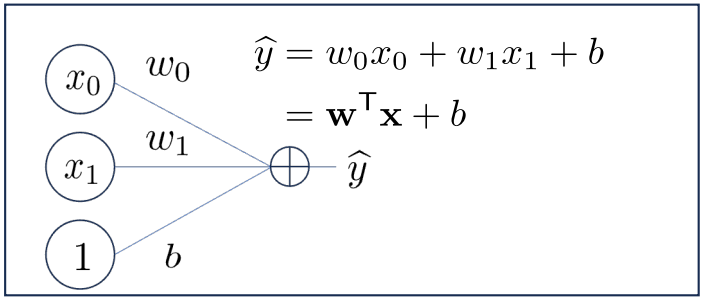
Ans: 1.1

```
w = torch.tensor([[0.4], [0.1]], requires_grad=True)
b = torch.tensor(1.0, requires_grad=True)
x = torch.tensor([[0.5], [0.4]], requires_grad=False)

y_hat = w.T @ x + b
```

```
tensor([[1.2400]], grad_fn=<AddBackward0>)
```

Ex 1.2: Using torch.nn.Module

Supervised algorithm



Encapsulates the approach using the class torch.nn.Module. There are two required functions in the class:

- Initialize parameters such as weights and biases (__init__)
- Declare the operations of the module and return its output: (forward)

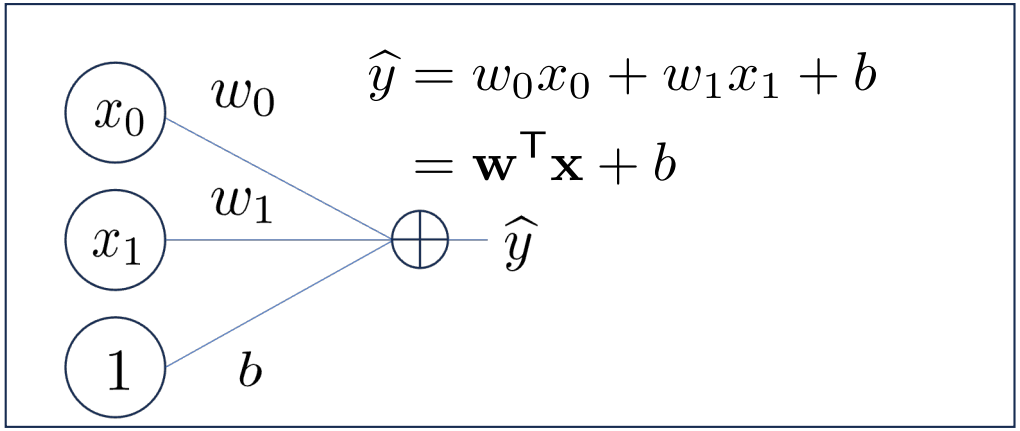
For the weight and biases, use torch.nn.Parameter to wrap them in the __init__. Check that the model's output is the same as the previous exercise.

Ans: 1.2

```
class Linear(torch.nn.Module):
    def __init__(self):
        super().__init__()
        w = torch.tensor([[0.4], [0.1]])
        b = torch.tensor(1.0)
        self.w = torch.nn.Parameter(w, requires_grad=True)
        self.b = torch.nn.Parameter(b , requires_grad=True)

    def forward(self, x):

        return self.w.T @ x + self.b

model = Linear()
y_hat = model(x) # also can call model.forward()
```

Ex. 1.3: Performance computation

Supervised algorithm



Suppose the input features $\mathbf{x} \in \mathbb{R}^{2\times 1}$ with $x_0 = 0.5$ and $x_1 = 0.4$ has a label $y = -0.5$, create a function to compute the loss $\mathcal{L} = (y - \widehat{y})^2$.

Ans. 1.3:

```
# Ex. 1.3

y = -0.5
def squared_error(y, y_hat):
       return (y-y_hat)**2

loss = squared_error(y, y_hat)
print(loss)
```

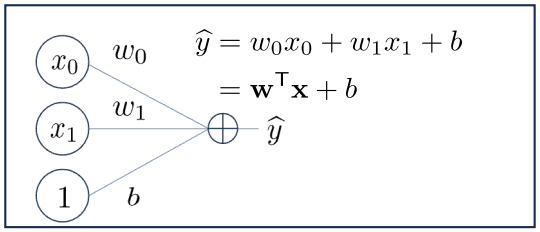
```
tensor([[3.0276]], grad_fn=<PowBackward0>)
```

It is important to note that $\mathcal{L}$ is typically a real-valued scalar

You can also look at the module torch.nn.MSELoss

Ex. 1.4: Optimizing using gradient descent

Supervised algorithm



Loss

$$\mathcal{L} = (y - \widehat{y})^2$$

Now that we know the loss value, we can optimize its weights and biases using an optimizer. Generally, the stochastic gradient descent approach is employed, which computes via

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial \mathbf{w}},$$
$$b_{\text{new}} = b_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial b},$$

where μ is the learning rate and $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} \in \mathbb{R}^{2\times 1}$ and $\frac{\partial \mathcal{L}}{\partial b} \in \mathbb{R}$ are the gradients of the weights and biases, respectively. Use `loss.backward()` to compute these gradients and print out its value via `model.w.grad` and `model.b.grad`.

Ans. 1.4:

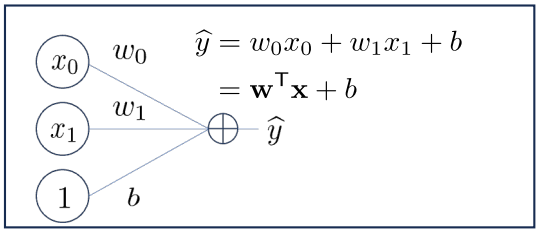
```
model = Linear()
y_hat = model(x)
loss = squared_error(y, y_hat)
model.zero_grad()
loss.backward()

print(model.w.grad)
print(model.w.grad.shape)
print(model.b.grad)
print(model.b.grad.shape)

tensor([[1.7400], [1.3920]])
torch.Size([2, 1])
tensor(3.4800)
torch.Size([])
```

In PyTorch, the gradients are accumulated after each backward operation.

It is important to reset the gradients via zero_grad before backward
Try without it!

Ex. 1.5: Updating the weight

Supervised algorithm



Loss

$$\mathcal{L} = (y - \widehat{y})^2$$

Now that we know the loss value, we can optimize its weights and biases using an optimizer. Generally, the gradient descent approach is employed, which computes via

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial \mathbf{w}},$$
$$b_{\text{new}} = b_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial b},$$

where μ is the learning rate and $\frac{\partial \mathcal{L}}{\partial \mathbf{w}} \in \mathbb{R}^{2\times 1}$ and $\frac{\partial \mathcal{L}}{\partial b} \in \mathbb{R}$ are the loss gradient of the weights and biases, respectively.

Obtain the updated weights $\mathbf{w}_{\text{new}}$ and b_{new} with $\mu = 0.01$. Then, obtain $\widehat{y}$ and $\mathcal{L}$ using the updated weights.

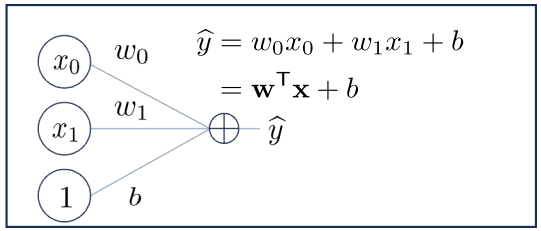
Ans. 1.5:

```
w_new = w - learning_rate * w_grad
b_new = b - learning_rate * b_grad

y_hat = tensor([[1.1909]], grad_fn=<AddBackward0>)
loss = tensor([[2.8593]], grad_fn=<PowBackward0>)
```

Notice that the loss has decreased.
This serves as an easy check to confirm that the gradient update is successful.

Ex. 1.6: Train for more steps

Supervised algorithm



Loss

$$\mathcal{L} = (y - \widehat{y})^2$$

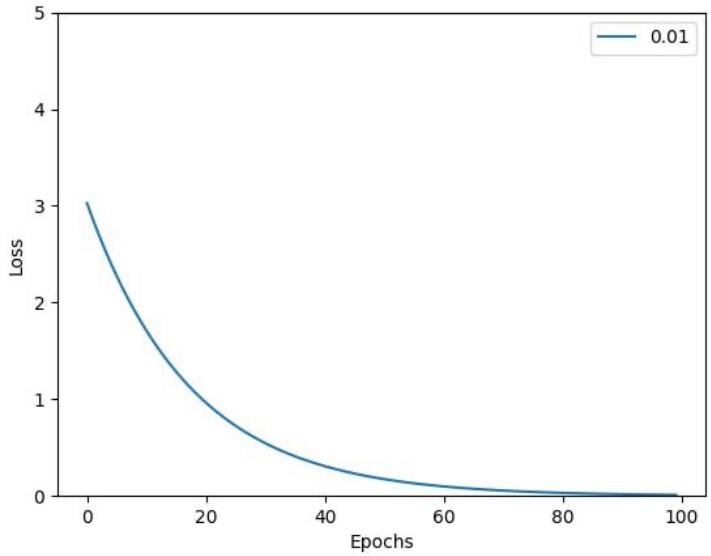
Optimization approach

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial \mathbf{w}},$$
$$b_{\text{new}} = b_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial b},$$

The optimization process is repeated for all data points in a dataset for stochastic gradient descent (SGD). Optimizing through all data points in a dataset means it has been trained for an *epoch*.

For simplicity of this exercise, we shall optimize the linear approach to the same data point. Optimize the linear algorithm using SGD for 100 epoch. Store the results of $\mathcal{L}$ at each epoch and plot it.

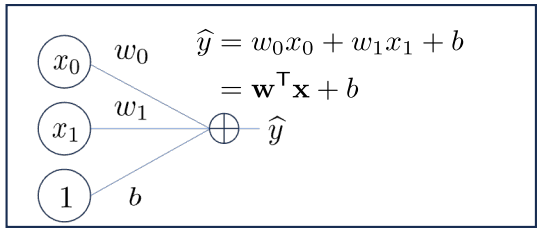
Ans. 1.6




```
epochs = 100
# initialize model
model = Linear()
learning_rate = 0.01
losses = []
for i in range(epochs):

    # apply algorithm
    y_hat = model(x)

    loss = squared_error(y_hat, y)
    losses.append(loss.detach().item())

    # zero previous gradients before computing
    gradient
    model.zero_grad()
    # compute gradient
    loss.backward()

    # optimize weights
    optimize_weights(model, learning_rate)
```

Ex. 1.7: Effects of different learning rate

Supervised algorithm



Loss

$$\mathcal{L} = (y - \widehat{y})^2$$

Optimization approach

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial \mathbf{w}},$$

$$b_{\text{new}} = b_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial b},$$

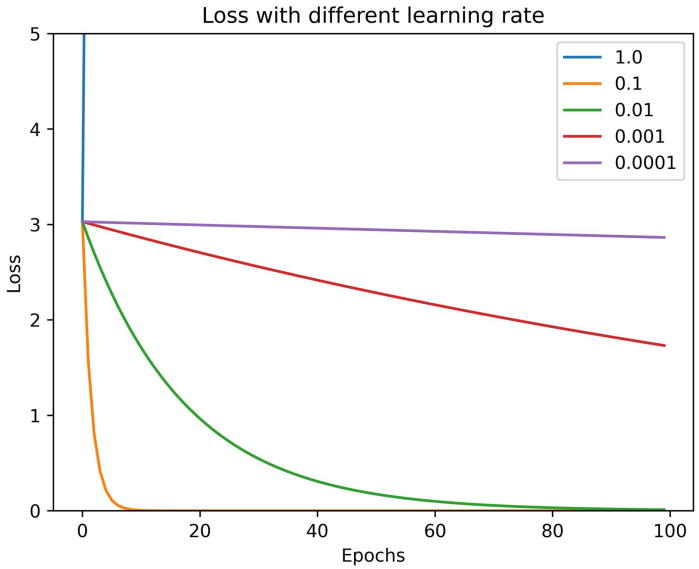
The learning rate is an important *hyperparameter* for training neural networks.

For simplicity of this exercise, we shall run the training in Ex. 1.5 for learning rate in $\{1.0, 0.1, 0.01, 0.001\}$, and store the results of $\mathcal{L}$ at each learning rate and plot it.

Answer 1.7:

Optimization approach

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial \mathbf{w}},$$
$$b_{\text{new}} = b_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial b},$$

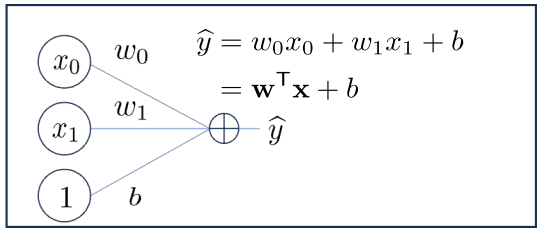


- The learning rate is a vital *hyperparameter* for training neural networks.
- Notice that the setting learning for this linear approach has mainly one trade-off: convergence speed and instabilities during training.
- As shown from the plot, a low learning rate can result in slow convergence, while a large one can cause significant weight updates, leading to non-convergence.

We will look into this effect more in week 4 onwards.

Eg. 2: Gradient descent

Supervised algorithm



Loss

$$\mathcal{L} = (y - \widehat{y})^2$$

Optimization approach

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial \mathbf{w}},$$
$$b_{\text{new}} = b_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial b},$$

Derive the gradient of the loss w.r.t $\mathbf{w} = [w_0, w_1]^\mathsf{T}$ in terms of $\mathbf{x}$,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \left[\frac{\partial \mathcal{L}}{\partial w_0}, \frac{\partial \mathcal{L}}{\partial w_1}\right]^\mathsf{T},$$

where $\mathcal{L} = (\widehat{y} - y)^2$.

Start by expressing $\mathcal{L}$ in terms of w_0, w_1, b, x_0, x_1, and y. Then, perform partial derivative w.r.t. w_0 where the other parameters are **constants** (including w_1)

Eg. 2:

Derive the gradient of the loss w.r.t $\mathbf{w} = [w_0, w_1]^\mathsf{T}$ in terms of $\mathbf{x}$,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \left[\frac{\partial \mathcal{L}}{\partial w_0}, \frac{\partial \mathcal{L}}{\partial w_1}\right]^\mathsf{T}, \qquad \begin{aligned} \widehat{y} &= w_0 x_0 + w_1 x_1 + b \\ &= \mathbf{w}^\mathsf{T}\mathbf{x} + b \end{aligned}$$

where $\mathcal{L} = (\widehat{y} - y)^2$.

Start by expressing $\mathcal{L}$ in terms of w_0, w_1, b, x_0, x_1, and y.

$$\begin{aligned} \mathcal{L} &= (\widehat{y} - y)^2 \\ &= (\widehat{y}^2 - 2\widehat{y}y + y^2) \end{aligned}$$

$$\begin{aligned} \widehat{y}^2 &= (w_0 x_0 + w_1 x_1 + b)^2 \\ &= (w_0 x_0)^2 + (w_1 x_1)^2 + b^2 \\ &\quad + 2b(w_0 x_0 + w_1 x_1) + 2(w_0 x_0 w_1 x_1) \end{aligned}$$

$$\widehat{y}y = y(w_0 x_0 + w_1 x_1 + b)$$

Eg. 2: con't

Derive the gradient of the loss w.r.t $\mathbf{w} = [w_0, w_1]^\mathsf{T}$ in terms of $\mathbf{x}$,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \left[\frac{\partial \mathcal{L}}{\partial w_0}, \frac{\partial \mathcal{L}}{\partial w_1}\right], \qquad \begin{aligned}\widehat{y} &= w_0 x_0 + w_1 x_1 + b \\ &= \mathbf{w}^\mathsf{T}\mathbf{x} + b\end{aligned}$$

where $\mathcal{L} = (\widehat{y} - y)^2$.

Start by expressing $\mathcal{L}$ in terms of w_0, w_1, b, x_0, x_1, and y.

$$\mathcal{L} = (\widehat{y} - y)^2 = (\widehat{y}^2 - 2\widehat{y}y + y^2)$$

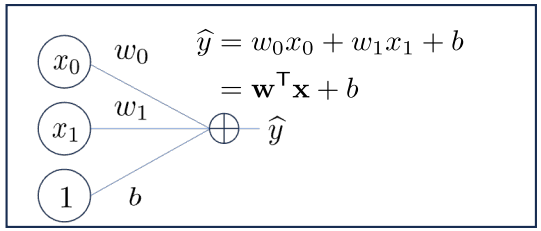
$$\widehat{y}^2 = (w_0 x_0 + w_1 x_1 + b)^2 = (w_0 x_0)^2 + (w_1 x_1)^2 + b^2 + 2b(w_0 x_0 + w_1 x_1) + 2(w_0 x_0 w_1 x_1)$$

$$\widehat{y}y = y(w_0 x_0 + w_1 x_1 + b)$$

$$\begin{aligned}\frac{\partial \mathcal{L}}{\partial w_0} &= \frac{\partial}{\partial w_0}(w_0^2 x_0^2 + 2b w_0 x_0 + 2 w_0 x_0 w_1 x_1 - 2y w_0 x_0) \\ &= 2 w_0 x_0^2 + 2b x_0 + 2 x_0 w_1 x_1 - 2y x_0 \\ &= 2x_0(w_0 x_0 + w_1 x_1 + b - y)\end{aligned}$$

Ex. 2: Calculate the gradient

Supervised algorithm



Loss

$$\mathcal{L} = (y - \widehat{y})^2$$

Optimization approach

$$\mathbf{w}_{\text{new}} = \mathbf{w}_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial \mathbf{w}},$$
$$b_{\text{new}} = b_{\text{old}} - \mu \frac{\partial \mathcal{L}}{\partial b},$$

Derive the gradient of the loss w.r.t $\mathbf{w} = [w_0, w_1]^\mathsf{T}$ in terms of $\mathbf{x}$,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \left[\frac{\partial \mathcal{L}}{\partial w_0}, \frac{\partial \mathcal{L}}{\partial w_1}\right]^\mathsf{T},$$

where $\mathcal{L} = (\widehat{y} - y)^2$.

Compute the gradient function for $\frac{\partial \mathcal{L}}{\partial w_1}$. Compare the results with that of the previous part.

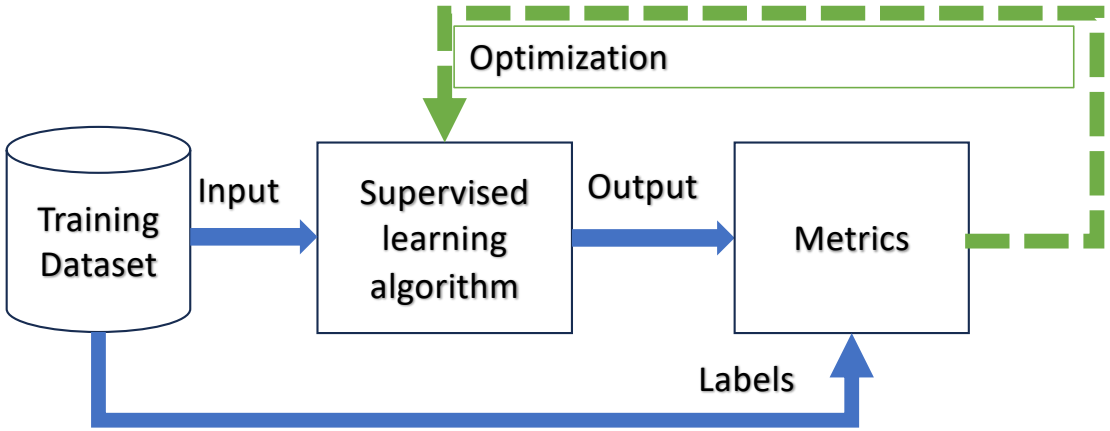
NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

EE4414: Week 3

By: Zhi-Wei Tan

Course coordinator: Prof Yap

Review of last week components

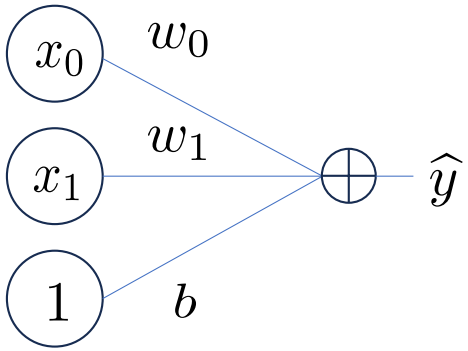


Ch. 1: Linear approaches

- One of the most fundamental operations in neural networks is called linear transformation or linear mapping.

- This transformation can be interpreted in multiple ways:
  - Projection, Rotation, Scaling, …
  - **Decision boundary: SVM**
  - Weightage of features: McCulloch–Pitts (MCP) neuron

$$\widehat{y} = w_0 x_0 + w_1 x_1 + b$$



Learning objective

- Interpreting a linear classifier with the Decision boundary

Ex. 1.1 : A linear classifier

A supervised linear algorithm computes via

$$\widehat{y}^{(i)} = \mathbf{w}^{\mathsf{T}}\mathbf{x}^{(i)} + b,$$

where $\mathbf{w} = [1 \;\; 3]^{\mathsf{T}} \in \mathbb{R}^{2\times 1}$, $b = 0$ are respectively, its weights and bias and $\mathbf{x}^{(i)}$ and $\widehat{y}^{(i)}$ are its ith input and output, respectively.

Use torch.nn.Module to construct it. (refer to previous week)

Then, compute its outputs given $x^{(0)} = [1 \;\; 0.5]^{\mathsf{T}}$ and $x^{(1)} = [0.1 \;\; -0.3]^{\mathsf{T}}$.

Ans. 1.1 : A linear classifier

```
class Linear(torch.nn.Module):
    def __init__(self):
        super().__init__()
        w = torch.tensor([[1.0],[3.0]])
        b = torch.tensor(0.0)
        self.w = torch.nn.Parameter(w, requires_grad=True)
        self.b = torch.nn.Parameter(b, requires_grad=True)

    def forward(self, x):
        return self.w.T @ x + self.b

model = Linear()
input_features = [torch.tensor([[1.0], [0.5]], requires_grad=False),
torch.tensor([[0.1], [-0.3]], requires_grad=False)]
output_features = []
for i, x in enumerate(input_features):
    y_hat = model.forward(x)
    output_features.append(y_hat)


[tensor([[2.5000]], grad_fn=<AddBackward0>),
tensor([[-0.8000]], grad_fn=<AddBackward0>)]
```

Ex. 1.2 : A linear classifier

A supervised linear algorithm computes via

$$\widehat{y}^{(i)} = \mathbf{w}^{\mathsf{T}}\mathbf{x}^{(i)} + b,$$

where $\mathbf{w} = [1 \;\; 3]^{\mathsf{T}} \in \mathbb{R}^{2\times 1}$, $b = 0$ are, respectively, its weights and bias and $\mathbf{x}^{(i)}$ and $\widehat{y}^{(i)}$ are its ith input and output, respectively.

Suppose $x^{(0)} = [1 \quad 0.5]^{\mathsf{T}}$ is classified as a feature in $\mathcal{C} = 1$ while $x^{(1)} = [0.1 \;\; -0.3]^{\mathsf{T}}$ is $\mathcal{C} = -1$, how can we perform classification by inferring from $\widehat{y}^{(i)}$?

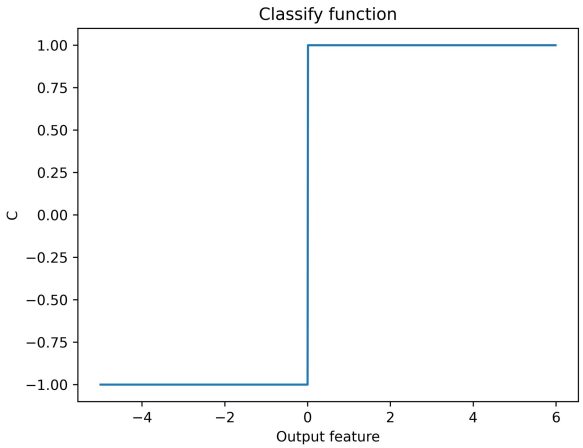
Use the if else condition to perform the classification.

Ans. 1.2: A linear classifier

```
def classify(y_hat):
    if y_hat > 0:
        return 1
    else:
        return -1

classifications = []
for y_hat in output_features:
    C = classify(y_hat)
    classifications.append(C)
```

```
[1, -1]
```

$$C = \begin{cases} 1; & \text{if } \widehat{y}^{(i)} > 0; \\ -1; & \text{if } \widehat{y}^{(i)} \leq 0; \end{cases}$$



In general, this is the way support vector machines (SVM) perform classification. The decision boundary in this case when $y^{(i)} = 0$

Ex. 1.3 : A linear classifier

A supervised linear algorithm computes via

$$\widehat{y}^{(i)} = \mathbf{w}^{\mathsf{T}}\mathbf{x}^{(i)} + b,$$

where $\mathbf{w} = [1 \;\; 3]^{\mathsf{T}} \in \mathbb{R}^{2\times 1}$, $b = 0$ are respectively, its weights and bias and $\mathbf{x}^{(i)}$ and $\widehat{y}^{(i)}$ are its ith input and output, respectively.

(No code) Find the equation of the decision boundary, $\widehat{y} = 0$ and express it in terms of elements x_0, x_1, w_0, w_1, and b with x_1 being the subject. Note that $\widehat{y} = x_0 w_0 + x_1 w_1 + b$.

Then, construct the function for decision boundary, and plot its graph with x_0 being the x-axis, and x_1 being the y-axis.

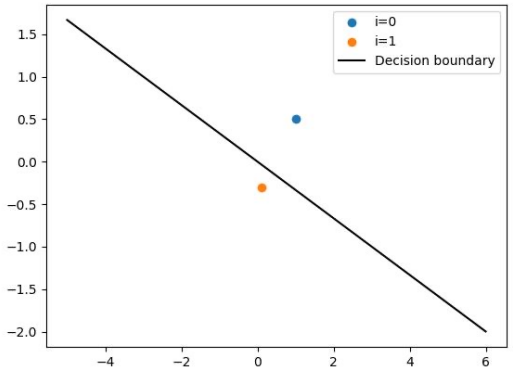
Ans 1.3:

(No code) Find the equation of the decision boundary, $\widehat{y} = 0$ and express it in terms of elements x_0, x_1, w_0, w_1, and b with x_1 being the subject. Note that $\widehat{y} = x_0 w_0 + x_1 w_1 + b$.
Given $\widehat{y} = 0$,

$$x_0 w_0 + x_1 w_1 + b = 0$$
$$x_1 w_1 = -x_0 w_0 - b$$
$$x_1 = -x_0 \frac{w_0}{w_1} - \frac{b}{w_1}$$

Then, construct the function for decision boundary, and plot its graph with x_0 being the x-axis, and x_1 being the y-axis.

```
def decision_boundary(x0, w0, w1, b):
        x1 = -x0* w0/w1 - b/w1
return x1
```



Ex. 1.4: Training the SVM

The loss function to optimize the weights and biases of the linear algorithm such that it maximizes the margin between classes is defined as

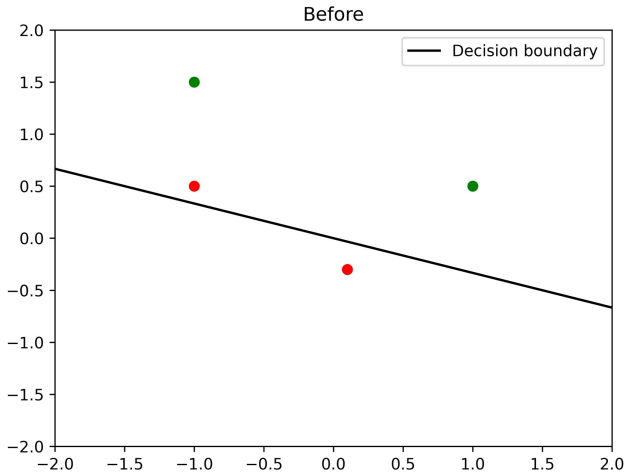
$$\mathcal{L} = \max\left(0, 1 - \mathcal{C}^{(i)}\widehat{y}^{(i)}\right),$$

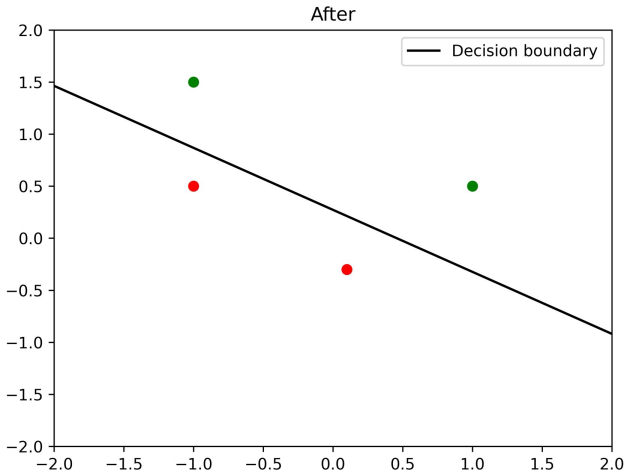
where $\mathcal{C}^{(i)}$ and $\widehat{y}^{(i)}$ are, respectively, the class and the estimated output feature for the ith data point.

For this exercise, declare the loss function and train the linear algorithm.

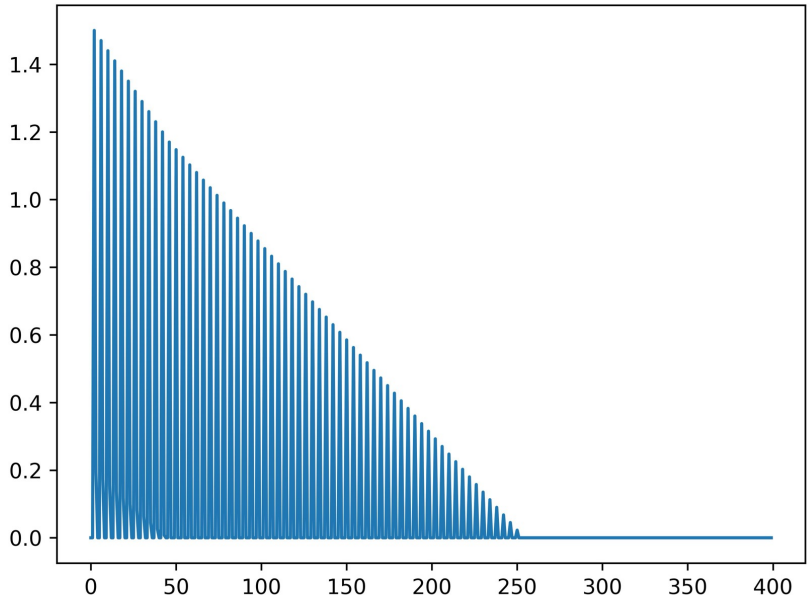
Ans. 1.4

```
def loss_function(c_hat, c):
        return torch.max(torch.tensor(0), 1-c * c_hat)
```





Ans 1.4:



$$\mathcal{L} = \max\left(0, 1 - \mathcal{C}^{(i)}\widehat{y}^{(i)}\right),$$

- Notice that although the loss decreases, it fluctuates between each data point.
- One way to understand why this happens can be via the loss function.
- Homework (after practice): Figure out why the loss behaves in this manner.

Ex. 1.5: Training the SVM

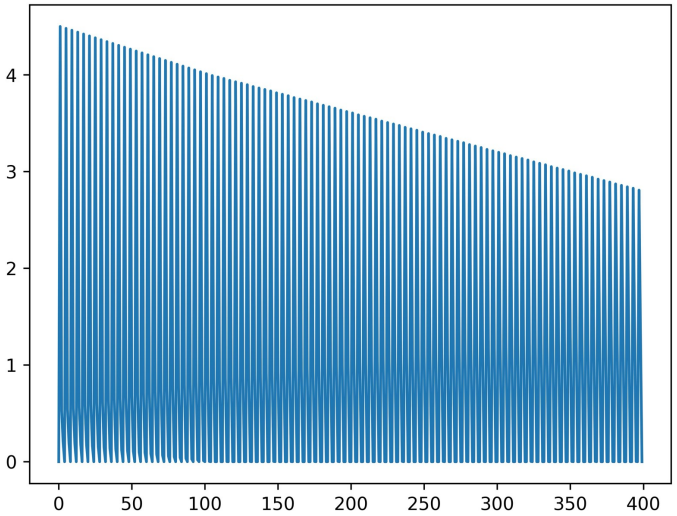
The loss function to optimize the weights and biases of the linear algorithm such that it maximizes the margin between classes is defined as

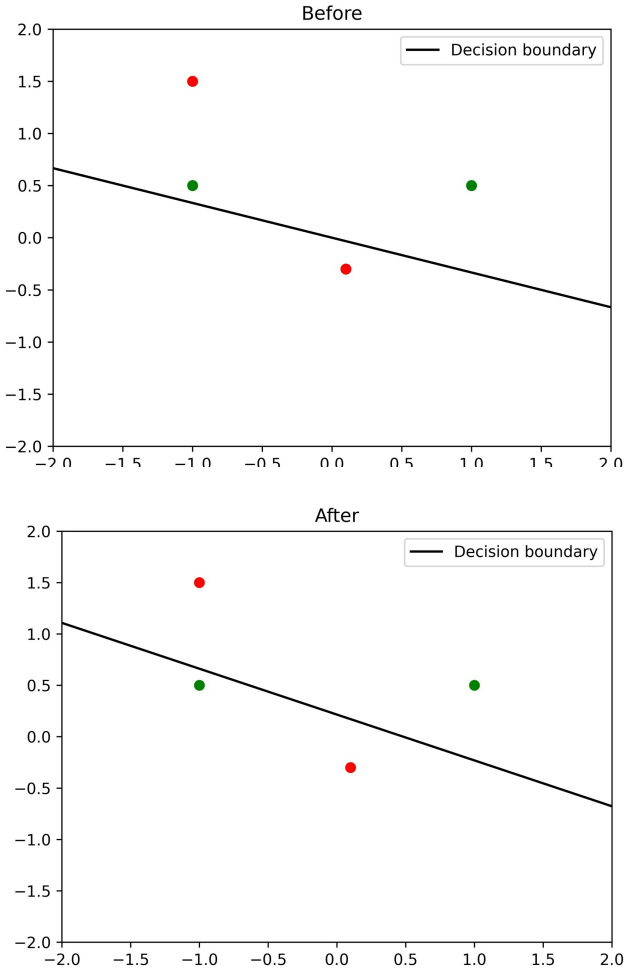
$$\mathcal{L} = \max\left(0, 1 - \mathcal{C}^{(i)}\widehat{y}^{(i)}\right),$$

where $\mathcal{C}^{(i)}$ and $\widehat{y}^{(i)}$ are, respectively, the class and the estimated output feature for the ith data point.

Using the previous code, change the labels $\widehat{y}^{(i)}$ from 1 to -1 or vice versa, but there should be two of each, such that despite training, the decision boundary does not properly separate the two classes.

Ans. 1.5:


0
1
2
3
4
0
50
100
150
200
250
300
350
400


Before
Decision boundary
After
Decision boundary

Reviews and further considerations

- In general, SVM, decision trees (XGBoost), etc., are better suited for low data and computational resources applications than deep learning approaches.
- However, embeddings from deep-learning techniques can be used as input features to these techniques.
- SVM approach works for non-linear feature spaces, one may employ kernels (such as circles, etc) to fit the feature space.
- Recommended library for work with SVM: scikit-learn

Briefing for practice

In this coding practice, you are required to

1. Submit this paper with each part's print output
2. Email the completed Jupyter Notebook (name as GroupNum-SeatNum-MatricNum) to **zhiwei.tan@ntu.edu.sg** with subject IE4414-GroupNum-SeatNum-MatricNum.

This practice contributes **10%** to your final score, with two sections each contributing **5%**. Each section has multiple questions, and in a newly created Jupyter Notebook, **each question should be answered in a cell** and indicates the question number, e.g., 1.1, via comments or markdown. You should run the Jupyter Notebook in Google Colab (https://colab.research.google.com/).

You are **allowed** to use your laptop and access the internet while doing this practice. However, you are considered **cheating** if you were to take photos, screen share, and/or perform any communication with any external party or parties.

You **will employ your matriculation number** as the variables for the practice, unless otherwise stated. Each number in your matriculation number represents a scalar value following the format U$abcdefg$X. For example, if your matriculation number is U1234567C, then $a = 1$, $b = 2$, $c = 3$, and so forth.

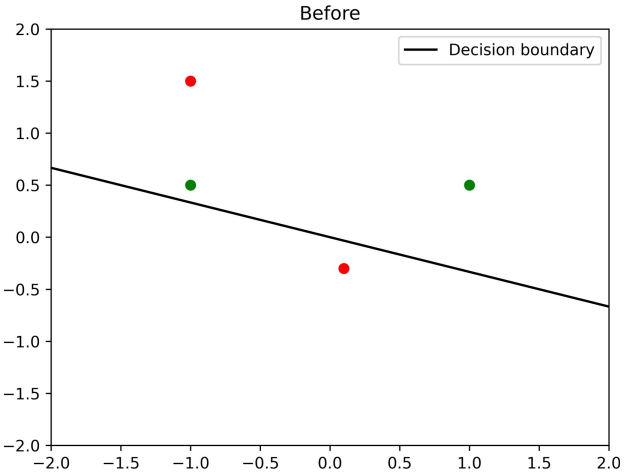
NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

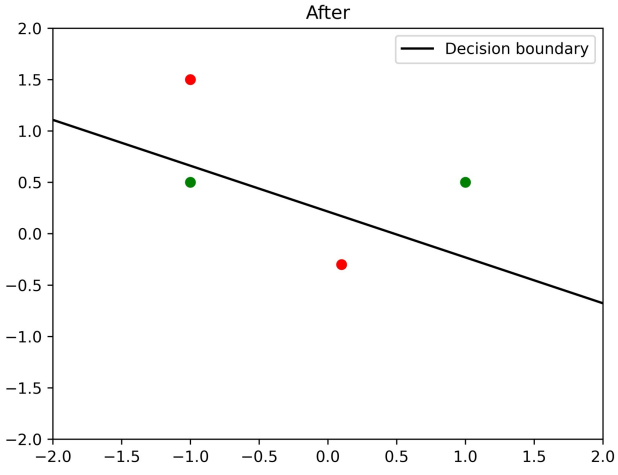
EE4414: Week 4

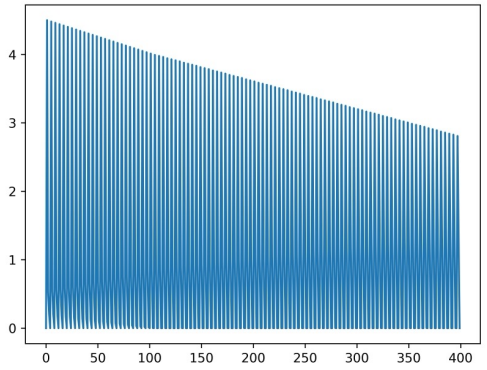
By: Zhi-Wei Tan

Course coordinator: Prof Yap

Review of last week: Training the SVM



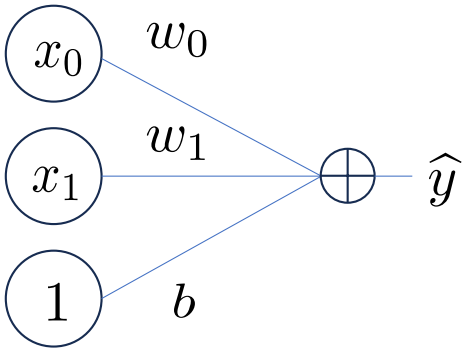


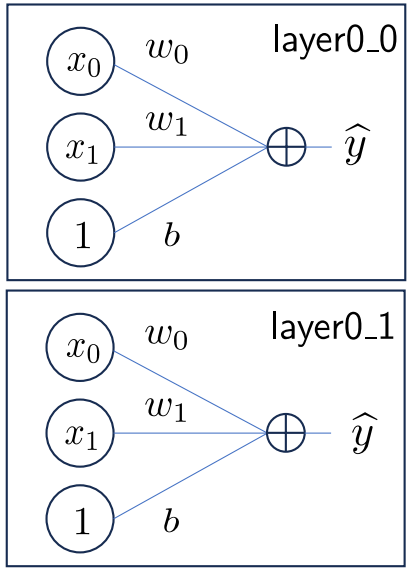


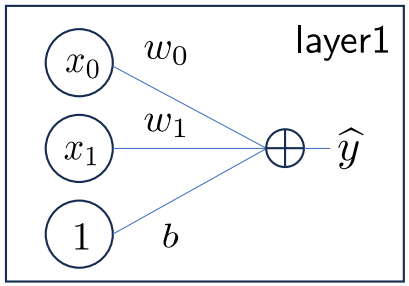
Learning objective

- Overcome the XOR problem with a two-layer feedforward neural network (NN)
- Activation function
- Decision boundary for a two-layer NN

Ex. 1.1 :Constructing the two-layer feedforward neural network







Initialize three Linear modules each with weight $\mathbf{w} \in \mathbb{R}^{2\times 1}$ and bias $b \in \mathbb{R}$. Then, apply the forward function for each of them using the provided input feature $\mathbf{x}$

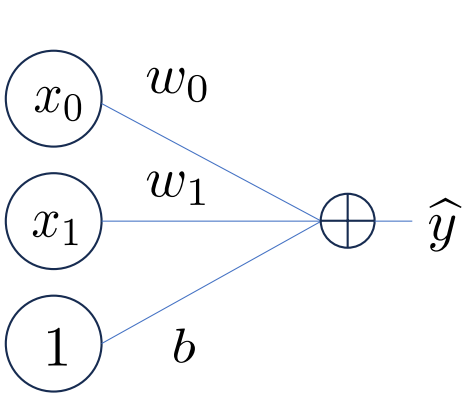
- layer0_0: $\mathbf{w} = [0.8, -0.5]^\mathsf{T}, b = 0.5$
- layer1_0: $\mathbf{w} = [-0.3, -0.1]^\mathsf{T}, b = -0.2$
- layer1_1: $\mathbf{w} = [-0.5, -0.4]^\mathsf{T}, b = -0.8$

Ans. 1.1

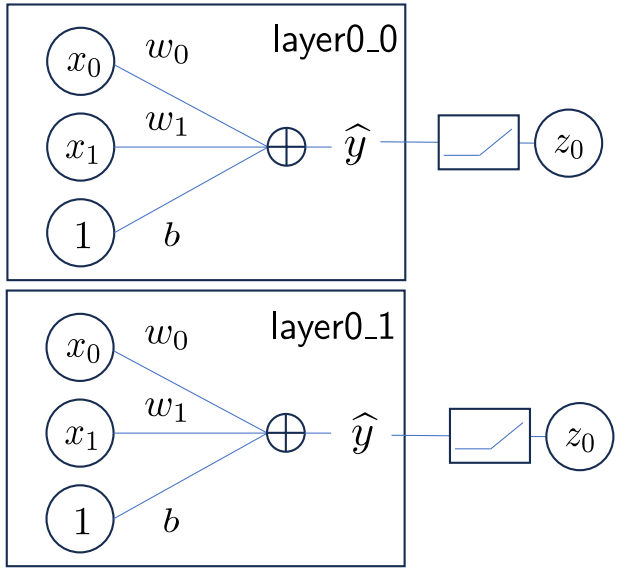
```
class Linear(torch.nn.Module):
    def __init__(self, w0, w1, b):
    super().__init__()
    w = torch.tensor([[w0],[w1]])
    b = torch.tensor(b)
    self.w = torch.nn.Parameter(w, requires_grad=True)
    self.b = torch.nn.Parameter(b, requires_grad=True)
def forward(self, x):
    # return self.w.T @ x + self.b
    # x: *, 2, 1
    # w: 2, 1
    return self.w.T @ x + self.b

layer0_0 = Linear(0.8, -0.5, 0.5)
layer0_1 = Linear(-1.0, -0.1, -0.2)
layer1 = Linear(-0.5, -0.4, -0.2)

x = torch.tensor([[0.3], [0.1]])
# apply forward for each linear layer and print its result
y_hat0_0 = layer0_0.forward(x)
y_hat0_1 = layer0_1.forward(x)
y_hat1 = layer1.forward(x)
```

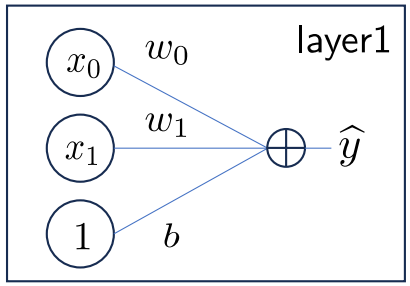
```
tensor([[0.6900]], grad_fn=<AddBackward0>)
tensor([[-0.3000]], grad_fn=<AddBackward0>)
tensor([[-0.9900]], grad_fn=<AddBackward0>)
```

Ex. 1.2 :Constructing the two-layer feedforward neural network



$$\widehat{y} = w_0 x_0 + w_1 x_1 + b$$
$$= \mathbf{w}^{\mathsf{T}} \mathbf{x} + b$$





The rectified linear unit (ReLU) activation function can be expressed via

$$z = \mathsf{max}(0, \widehat{y}).$$

For this exercise, apply ReLU to the output of layer0_0 and layer1_0.

Ans. 1.2

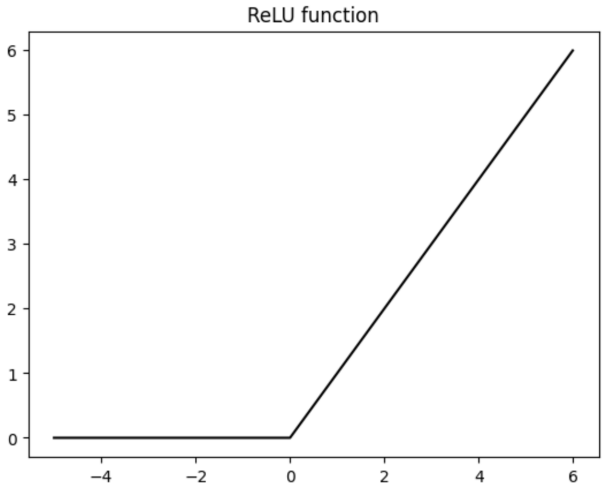
The rectified linear unit (ReLU) activation function can be expressed via

$$z = \max(0, \widehat{y}).$$

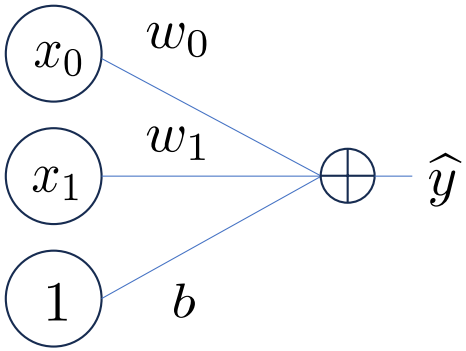
For this exercise, apply ReLU to the output of layer0_0 and layer1_0.

```
def relu(y_hat):
    return torch.max(torch.zeros_like(y_hat), y_hat)


# apply relu to y_hat0_1 and y_hat0_0
z_0 = relu(y_hat0_0)
z_1 = relu(y_hat0_1)
```

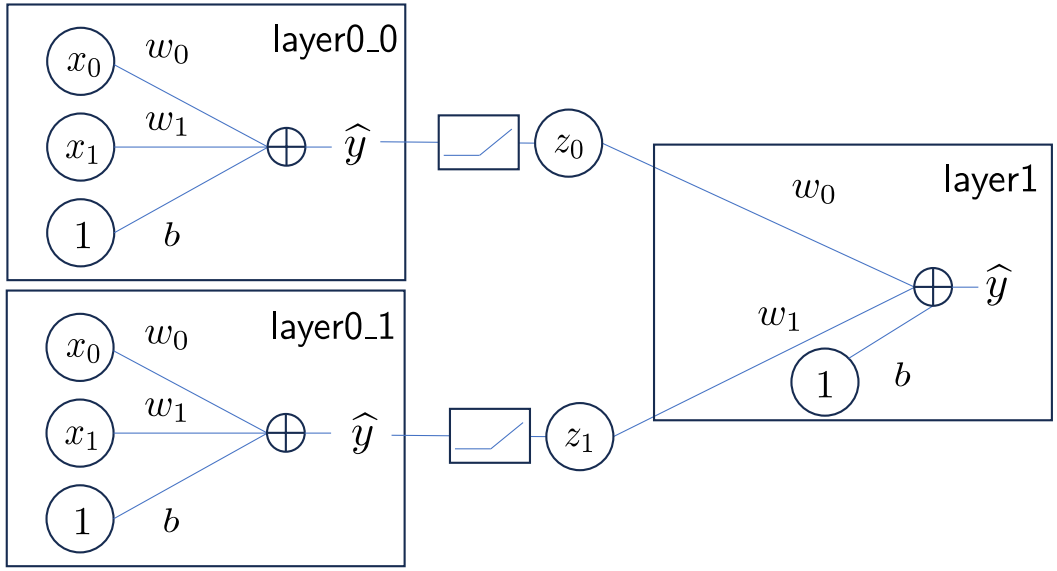
```
tensor([[0.6900]], grad_fn=<MaximumBackward0>)
tensor([[0.]], grad_fn=<MaximumBackward0>)
```



Ex. 1.3 :Constructing the two-layer feedforward neural network



$$\widehat{y} = w_0 x_0 + w_1 x_1 + b$$
$$= \mathbf{w}^\mathsf{T}\mathbf{x} + b$$



Obtain $\mathbf{z} = [z_0, z_1]^\mathsf{T} \in \mathbb{R}^{*\times 2\times 1}$ by concatenating using reverse index. Here, the variable $*$ denotes any dimensions.

Then, with $\mathbf{z}$, apply the forward operation of layer1.

Ans. 1.3

-2 -1

Obtain $\mathbf{z} = [z_0, z_1]^\mathsf{T} \in \mathbb{R}^{*\times 2\times 1}$ by concatenating using reverse index. Here, the variable $*$ denotes any dimensions.

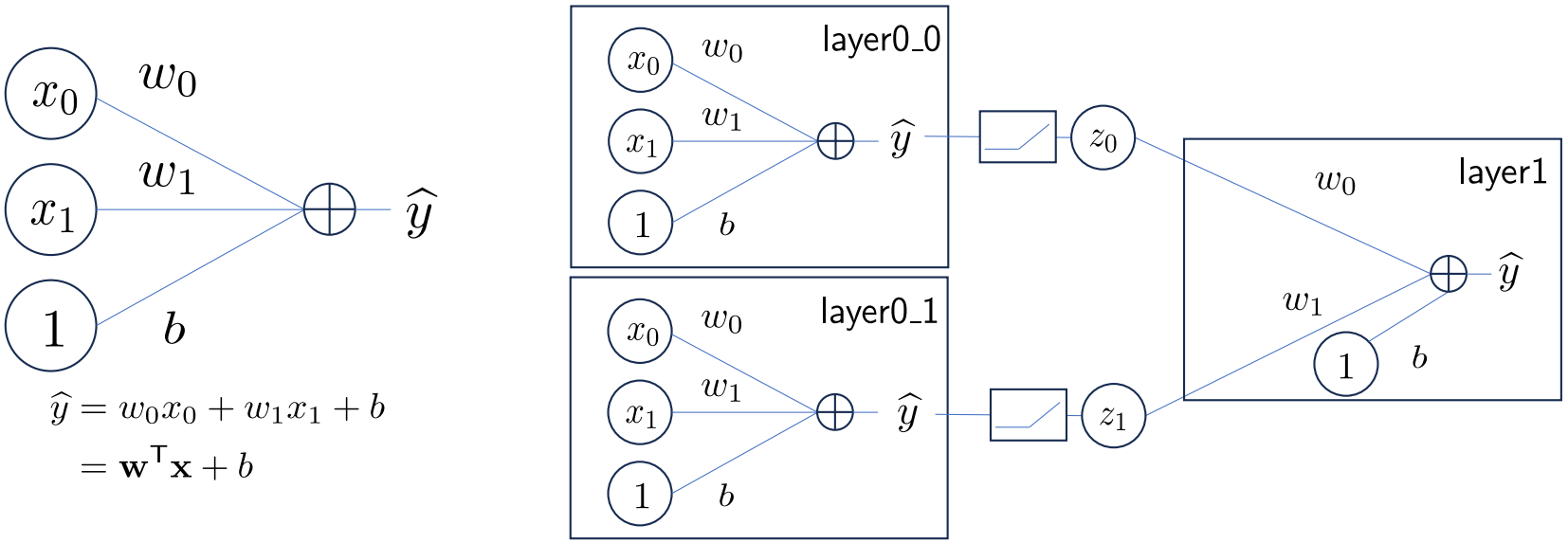
Then, with $\mathbf{z}$, apply the forward operation of layer1.

```
z = torch.cat([z_0, z_1], dim=-2)
print(z.shape)
y_hat = layer1.forward(z)
print(y_hat)
```

We employ reverse indexing so that the approach is broadcastable to any dimension (*).

This is also a consequence of PyTorch employing last N dimensions for the operation.

Ex. 1.4 :Constructing the two-layer feedforward neural network



Use torch.nn.Module to encapsulate the two-layer feedforward NN.

Then, obtain its output by stacking the two inputs.

Ans: 1.4

```
class TwoLayerNN(torch.nn.Module):

    def __init__(self):
        super().__init__()
        self.layer0_0 = Linear(0.8, -0.5, 0.5)
        self.layer0_1 = Linear(-1.0, -0.1, -0.2)
        self.layer1 = Linear(-0.5, -0.4, -0.2)
    def forward(self, x):

        # x: *, 2, 1
        y_hat0_0 = self.layer0_0.forward(x)
        y_hat0_1 = self.layer0_1.forward(x)

        # y_hat0_0: *, 1, 1
        # y_hat0_1: *, 1, 1
        z_0 = relu(y_hat0_0)
        z_1 = relu(y_hat0_1)

        # z_0: *, 1, 1
        # z_1: *, 1, 1
        z = torch.cat([z_0, z_1], dim=-2)
        # z: *, 2, 1
        # print(z.shape)
        y_hat = self.layer1.forward(z)
        return y_hat
```

```
x = torch.stack([x1, x2], dim=0)

print("x", x.shape)
model = TwoLayerNN()
y_hat = model(x)
```

```
x shape torch.Size([2, 2, 1])

tensor([[[-1.0450]], [[-1.0450]]], grad_fn=<AddBackward0>)
```

Ans: 1.4

```
class TwoLayerNN(torch.nn.Module):

    def __init__(self):
        super().__init__()
        self.layer0_0 = Linear(0.8, -0.5, 0.5)
        self.layer0_1 = Linear(-1.0, -0.1, -0.2)
        self.layer1 = Linear(-0.5, -0.4, -0.2)
    def forward(self, x):

        # x: *, 2, 1
        y_hat0_0 = self.layer0_0.forward(x)
        y_hat0_1 = self.layer0_1.forward(x)

        # y_hat0_0: *, 1, 1
        # y_hat0_1: *, 1, 1
        z_0 = relu(y_hat0_0)
        z_1 = relu(y_hat0_1)

        # z_0: *, 1, 1
        # z_1: *, 1, 1
        z = torch.cat([z_0, z_1], dim=-2)
        # z: *, 2, 1
        # print(z.shape)
        y_hat = self.layer1.forward(z)
        return y_hat
```

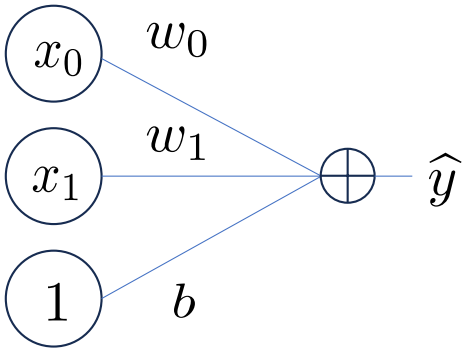
```
x = torch.stack([x1, x2], dim=0)
x = x.unsqueeze(0)
print("x", x.shape)
model = TwoLayerNN()
y_hat = model(x)
```

```
x shape torch.Size([1, 2, 2, 1])

tensor([[[[-1.0450]], [[-1.0450]]]], grad_fn=<AddBackward0>)
```

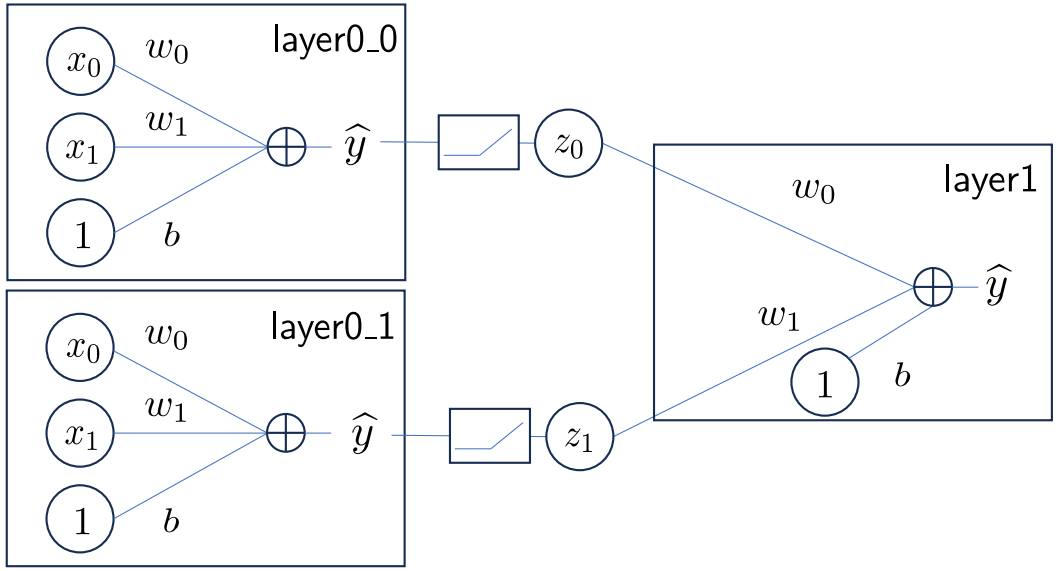
For Ex.1.5 Please make this changes first

```
# the labels for XOR
(before)
labels = [torch.tensor([1.0]),torch.tensor([-1.0]), torch.tensor([1.0]),
torch.tensor([-1.0])]

(after)
labels = [torch.tensor([[1.0]]),torch.tensor([[-1.0]]), torch.tensor([[1.0]]),
torch.tensor([[-1.0]])]
```

Ex. 1.5 :Constructing the two-layer feedforward neural network



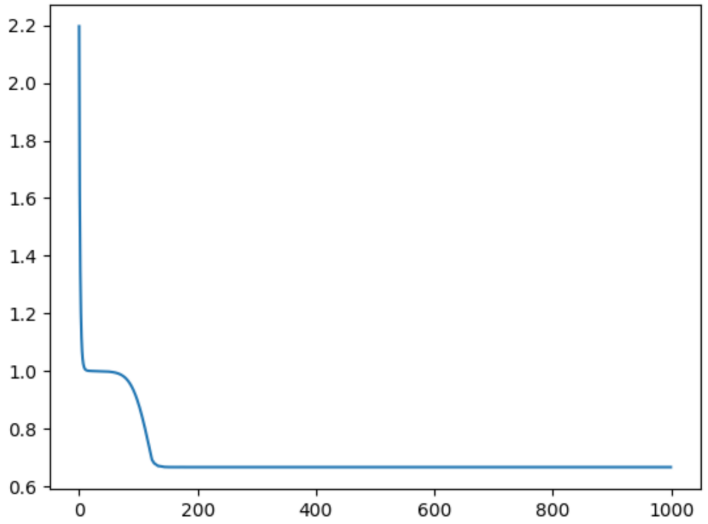
$$\widehat{y} = w_0 x_0 + w_1 x_1 + b$$
$$= \mathbf{w}^\mathsf{T}\mathbf{x} + b$$

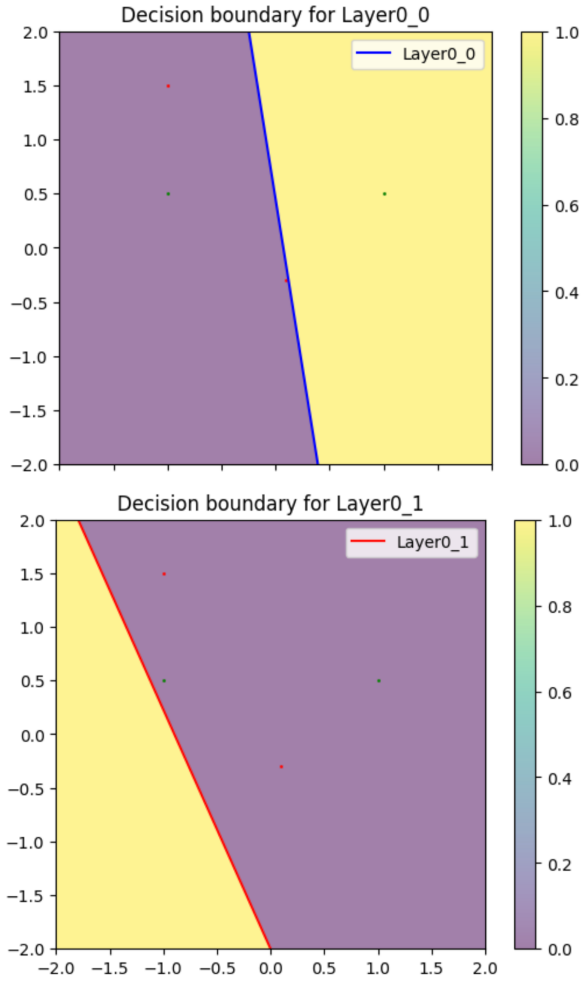


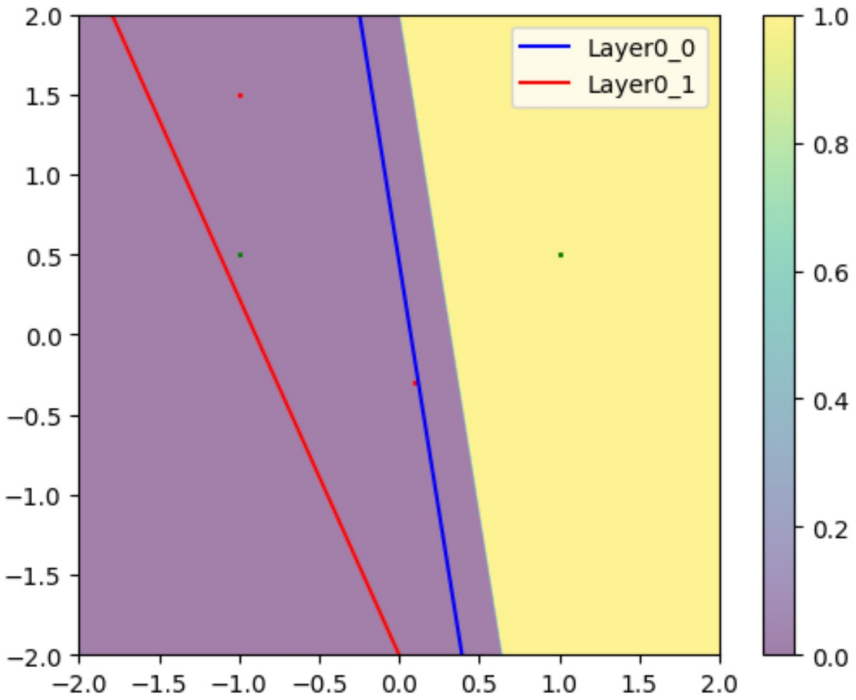
Complete the training loop. Use mean squared error $\sum_i^I (\widehat{y}^{(i)} - y^{(i)})^2$ as the loss function.
Then, run the next script cell which plots the response of the two-layer feedforward neural network.

Ans 1.5

```
def loss_function(y_hat, y):
        return torch.mean((y_hat - y)**2)



for epoch in range(epochs):

    # stack the input features and labels along the first dim
    x = torch.stack(input_features, 0)
    y = torch.stack(labels)
    # apply model
    y_hat = model.forward(x)
    loss = loss_function(y_hat, y)
    losses.append(loss.detach().item())
    # zero_grad
    model.zero_grad()
    # perform backward
    loss.backward()

    # optimize weights
    for layer in [model.layer0_0, model.layer0_1, model.layer1]:
        optimize_weights(layer, learning_rate)
```




Decision boundary for Layer0_0
Layer0_0
Decision boundary for Layer0_1
Layer0_1

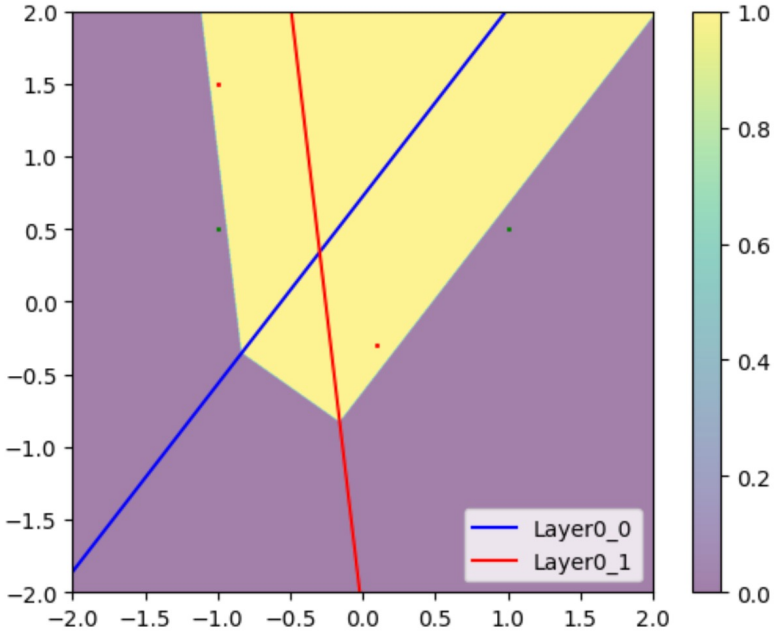

Layer0_0
Layer0_1

Free and easy: Plot the response with different activation, loss function

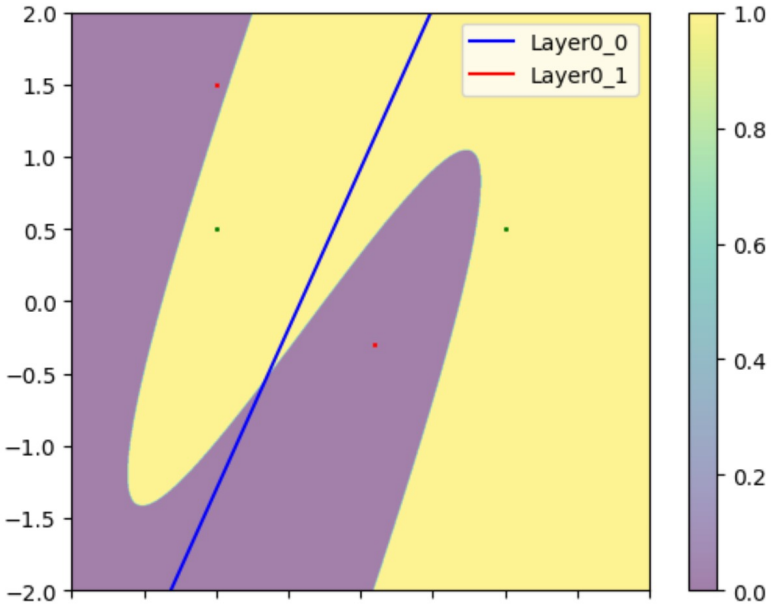
Copy the class TwoLayerNN and rename it.
Then, employ a different activation function (other than ReLU) such as sigmoid, tanh, from torch.nn.functional, or loss function.
After that, train and plot the response of the newly configured neural network.

Ans. 1.5

Trained with different initial weights and biases



Employed tanh as activation function



NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

EE4414: Week 5 HBL

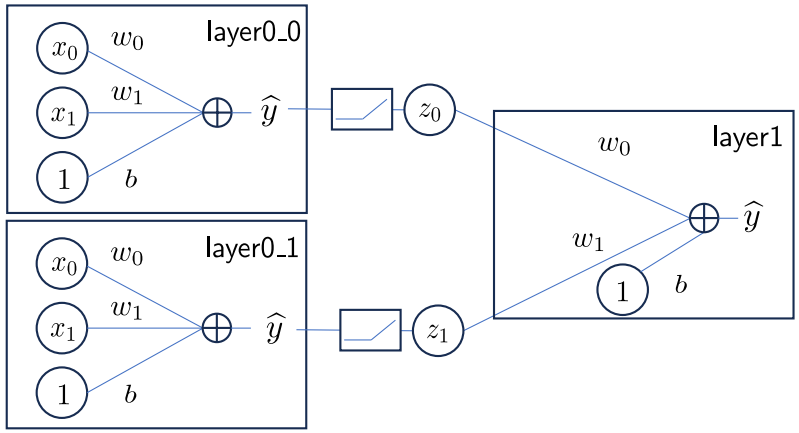
By: Zhi-Wei Tan

Course coordinator:
Prof Yap

Learning objective

- Calculate the loss gradient for two-layer NN
  - And its consequence given different activation functions

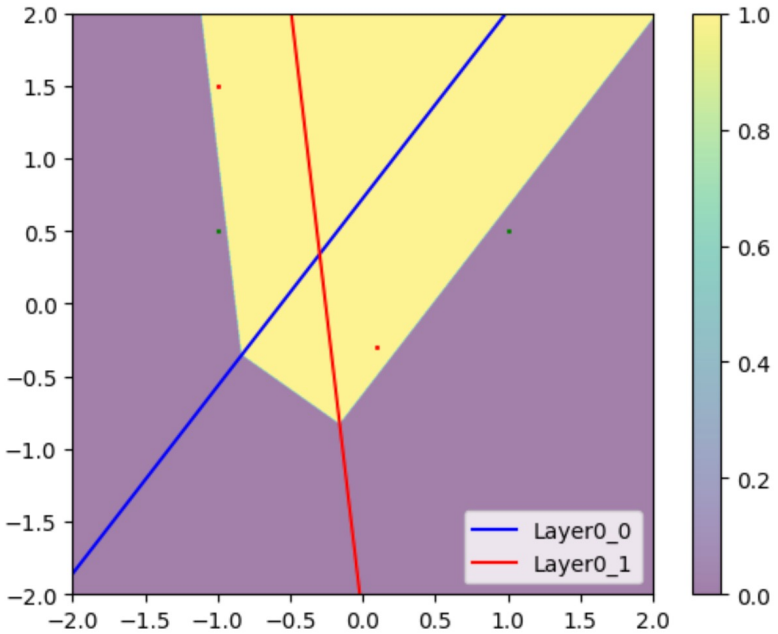
Review of last week: Training the two-layer neural network



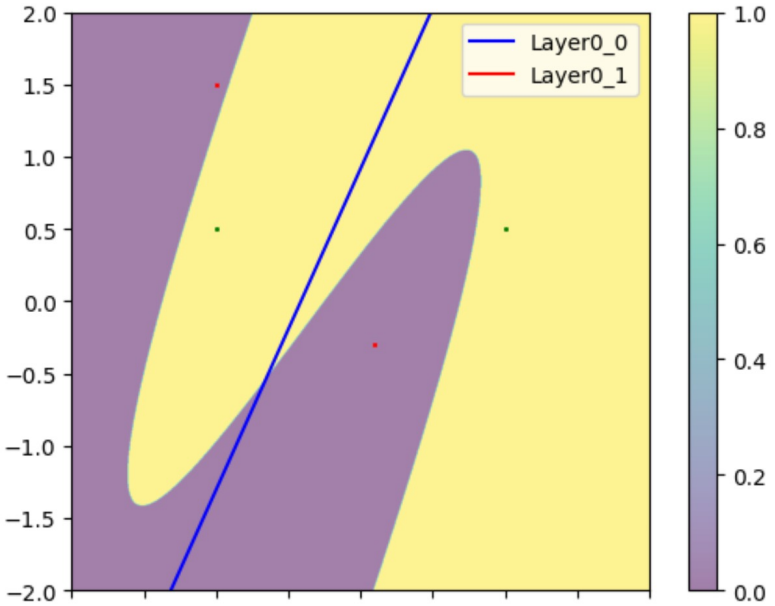
We employed three Linear modules and ReLU activations to construct a two-layer neural network (NN).
We demonstrated that the approach can solve the XOR problem space.

Review of last week: Training the two-layer neural network

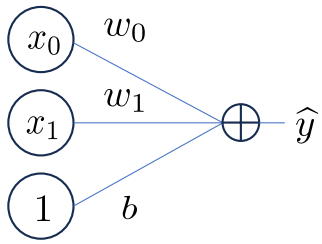
Trained with different initial weights and biases



Employed tanh as activation function



Ch. 1: Backpropagation



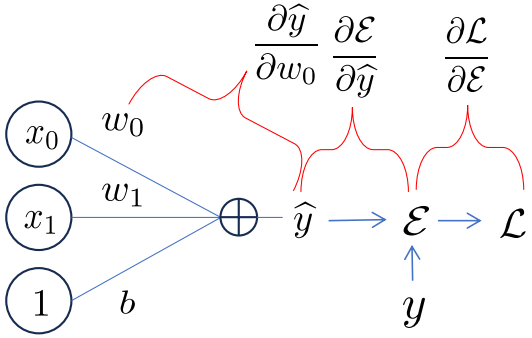
In the second week, we achieved the computation of loss gradients for a single linear approach to optimize its weights and biases. Recall that we obtain

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = \left[\frac{\partial \mathcal{L}}{\partial w_0} \frac{\partial \mathcal{L}}{\partial w_1} \right]^{\mathsf{T}},$$

where $\mathcal{L} = (\widehat{y} - y)^2$ with $\widehat{y} = x_0 w_0 + x_1 w_1 + b$. We obtained the same values computed by the PyTorch library.

However, there is a simpler way to achieve that, and it is via chain rule.

Eg. 1. 1:



Employing chain rule, we obtain

$$\frac{\partial \mathcal{L}}{\partial w_0} = \frac{\partial \mathcal{L}}{\partial \mathcal{E}} \frac{\partial \mathcal{E}}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial w_0},$$

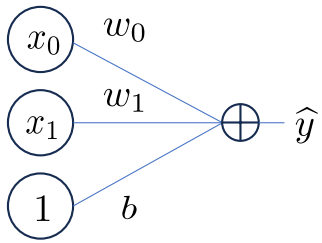
where $\mathcal{E} = (\widehat{y} - y)$ is the error term, $\frac{\partial \mathcal{L}}{\partial \mathcal{E}} = 2\mathcal{E} = 2(\widehat{y} - y)$, and $\frac{\partial \mathcal{E}}{\partial \widehat{y}} = 1$.

For this exercise, calculate the equation for $\frac{\partial \widehat{y}}{\partial w_0}$. Then, fill in the equation function deriv_y_over_w0 in the exercise.

Ans 1.1:

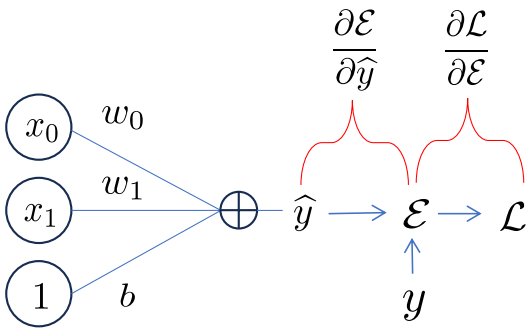
Employing chain rule, we obtain For this exercise, calculate the equation for $\frac{\partial \widehat{y}}{\partial w_0}$. Then, fill in the equation function deriv_y_over_w0 in the exercise.

$$\begin{aligned}\frac{\partial \widehat{y}}{\partial w_0} &= \frac{\partial}{\partial w_0} x_0 w_0 + x_1 w_1 + b \\ &= x_0\end{aligned}$$




```
def get_deriv_y_hat_over_w0(x):
        return x[0:1]
```

Ex 1.2:



Employing chain rule, derive the equation

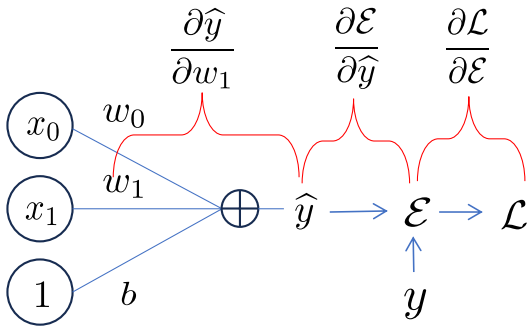
$$\frac{\partial \mathcal{L}}{\partial w_1}.$$

Then, fill in the equation for the function deriv_y_hat_over_w1.
Then, compute $\frac{\partial \mathcal{L}}{\partial w_1}$ and compare its value with the one computed by PyTorch.

Ans 1.2:

Employing chain rule, we obtain

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \mathcal{E}} \frac{\partial \mathcal{E}}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial w_1},$$



where

$$\frac{\partial \widehat{y}}{\partial w_1} = \frac{\partial}{\partial w_1} x_0 w_0 + x_1 w_1 + b$$
$$= x_1$$

$$\frac{\partial \mathcal{L}}{\partial w_0} = \frac{\partial \mathcal{L}}{\partial \mathcal{E}} \frac{\partial \mathcal{E}}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial w_0},$$

Same values!

```
def get_deriv_y_hat_over_w1(x):
        return x[1:2]

deriv_loss_over_error = get_deriv_loss_over_error(y_hat, y)
deriv_error_over_y_hat = get_deriv_error_over_y_hat(y_hat)
deriv_y_hat_over_w1 = get_deriv_y_hat_over_w1(x)
gradient_w1 = deriv_loss_over_error * deriv_error_over_y_hat * deriv_y_hat_over_w1
```

Ex 1.3:

Employing chain rule, derive $\frac{\partial \mathcal{L}}{\partial b}$.

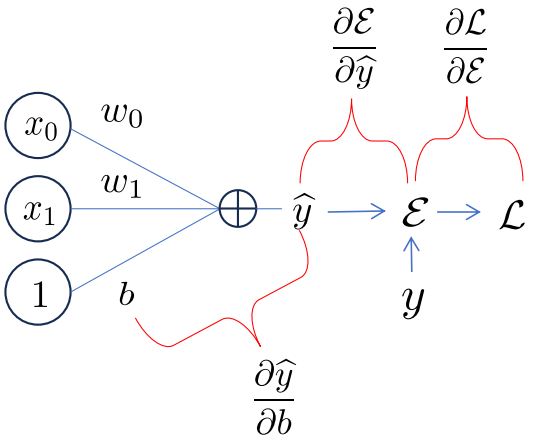
Then, fill in the equation for the function deriv_y_hat_over_b.

After that, compute $\frac{\partial \mathcal{L}}{\partial b}$ and compare its value with the one computed by PyTorch.

Ans. 1.3:

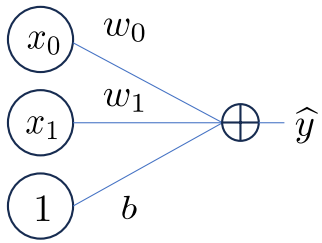
Employing chain rule, we obtain

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial \mathcal{E}} \frac{\partial \mathcal{E}}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial b},$$



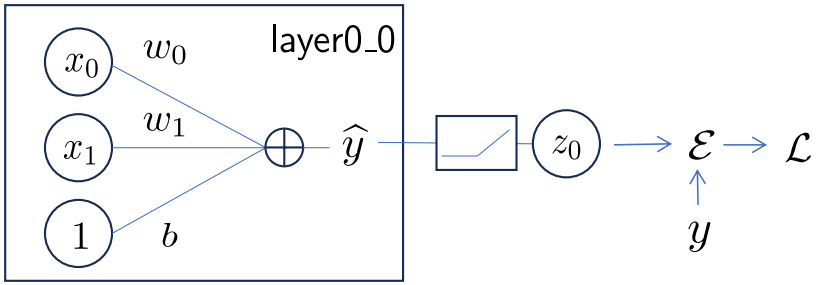
where

$$\frac{\partial \widehat{y}}{\partial w_1} = \frac{\partial}{\partial b} x_0 w_0 + x_1 w_1 + b$$
$$= 1$$




```
def get_deriv_y_hat_over_b(x):
        return torch.ones_like(x)[0:1]

deriv_loss_over_error = get_deriv_loss_over_error(y_hat, y)
deriv_error_over_y_hat = get_deriv_error_over_y_hat(y_hat)
deriv_y_hat_over_b = get_deriv_y_hat_over_b(x)
gradient_b = deriv_loss_over_error * deriv_error_over_y_hat * deriv_y_hat_over_b
```

Ex. 1.4:



For the above approach employ chain rule and derive equations for

$$\frac{\partial \mathcal{L}}{\partial w_0}$$

where $\mathcal{L} = \mathcal{E}^2 = (z_0 - y)^2$, $\mathcal{E} = (z_0 - y)$, and

$$z = \mathsf{max}\,(0, \widehat{y}) = \begin{cases} \widehat{y}; & \text{for } \widehat{y} > 0; \\ 0; & \text{otherwise.} \end{cases}$$

Then, fill in the equation in get_deriv_relu. After that, compute $\frac{\partial \mathcal{L}}{\partial w_0}$ and compare its value with the one computed by PyTorch.

Ans. 1.4:

Employing chain rule,

$$\frac{\partial \mathcal{L}}{\partial w_0} = \frac{\partial \mathcal{L}}{\partial z_0}\frac{\partial z_0}{\partial w_0} = \frac{\partial \mathcal{L}}{\partial \mathcal{E}}\frac{\partial \mathcal{E}}{\partial z_0}\frac{\partial z_0}{\partial \widehat{y}}\frac{\partial \widehat{y}}{\partial w_0}$$

where $\mathcal{L} = \mathcal{E}^2 = (z_0 - y)^2$, $\mathcal{E} = (z_0 - y)$, and

$$z = \mathsf{max}\,(0, \widehat{y}) = \begin{cases} \widehat{y}; & \text{for } \widehat{y} > 0; \\ 0; & \text{otherwise,} \end{cases}$$

Same operations as Ex. 1.1!

$$\frac{\partial \mathcal{L}}{\partial \mathcal{E}} = 2\mathcal{E} \qquad \frac{\partial \mathcal{E}}{\partial z} = 1 \qquad \frac{\partial \widehat{y}}{\partial w_0} = x_0$$

$$\frac{\partial z}{\partial \widehat{y}} = \begin{cases} 1; & \text{for } \widehat{y} > 0; \\ 0; & \text{otherwise.} \end{cases}$$

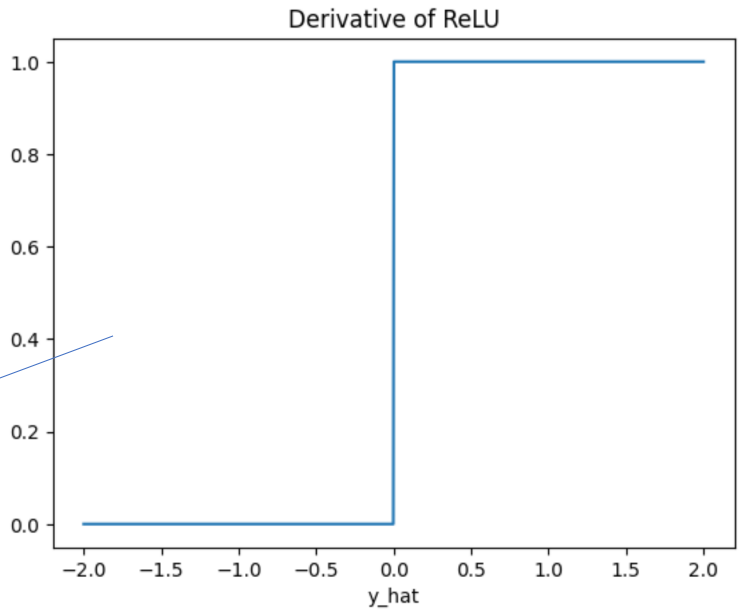
Ans. 1.4

```
def get_deriv_relu(y_hat):
    return 1/ y_hat * torch.max(y_hat, torch.zeros_like(y_hat))
```

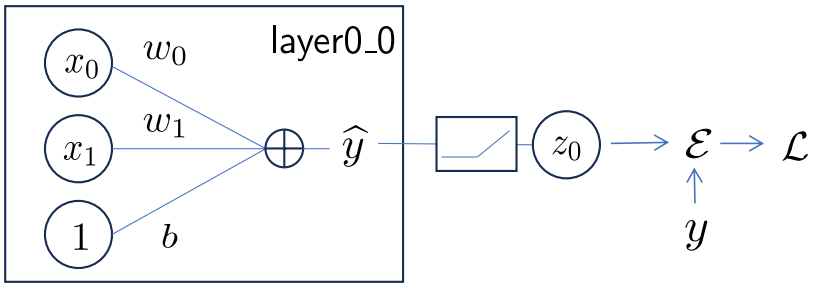
```
def get_deriv_relu(y_hat):
    return y_hat >0
```

$$\frac{\partial z}{\partial \widehat{y}} = \begin{cases} 1; & \text{for } \widehat{y} > 0; \\ 0; & \text{otherwise.} \end{cases}$$

When $\widehat{y} \leq 0$, $\frac{\partial \mathcal{L}}{\partial w_0} = 0$, $\frac{\partial \mathcal{L}}{\partial w_1} = 0$, and $\frac{\partial \mathcal{L}}{\partial b} = 0$.
Hence, no weight updates.



Ex. 1.4.1 (optional):



For the above approach employ chain rule and derive equations for

$$\frac{\partial \mathcal{L}}{\partial w_0}$$

where $\mathcal{L} = \mathcal{E}^2 = (z_0 - y)^2$, $\mathcal{E} = (z_0 - y)$, and

$$z = \sigma(\widehat{y}) = \frac{1}{1 + e^{-\widehat{y}}}$$

Then, fill in the equation in get_deriv_sigmoid. After that, compute $\frac{\partial \mathcal{L}}{\partial w_0}$ and compare its value with the one computed by PyTorch.

Ans. 1.4.1:

$$z = \sigma(\widehat{y}) = \frac{1}{1 + e^{-\widehat{y}}}$$

Let $u = 1 + e^{-\widehat{y}}$, then

$$\frac{\partial z}{\partial \widehat{y}} = \frac{\partial z}{\partial u} \frac{\partial u}{\partial \widehat{y}}, \tag{1}$$

where

$$\begin{aligned} \frac{\partial z}{\partial u} &= \frac{\partial}{\partial u} u^{-1} \\ &= -u^{-2} \\ &= -\left(1 + e^{-\widehat{y}}\right)^{-2}, \end{aligned} \tag{2}$$

and

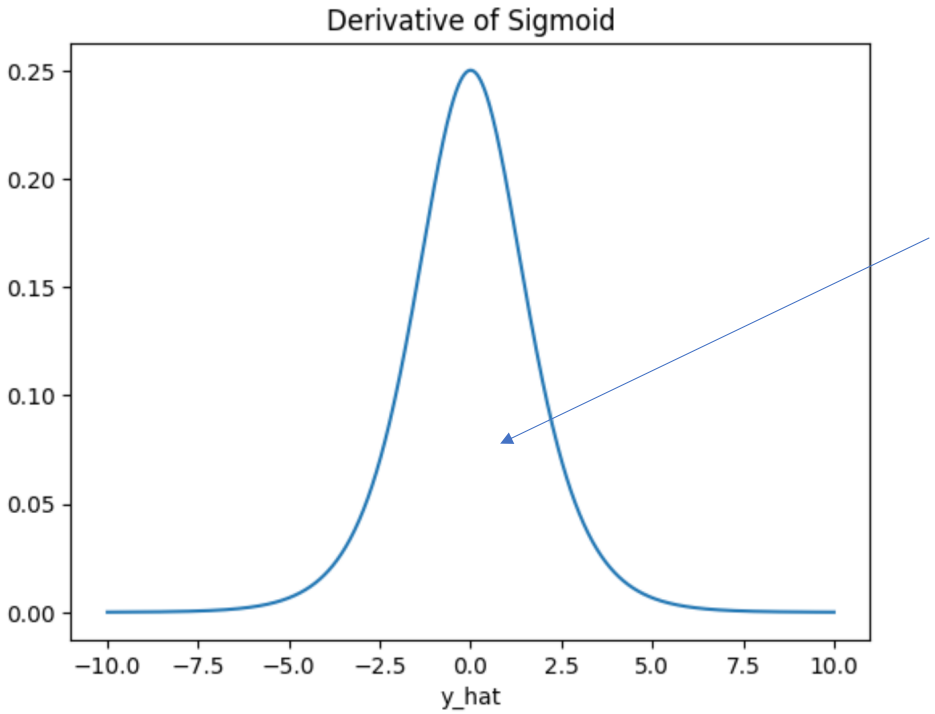
$$\frac{\partial u}{\partial \widehat{y}} = \frac{\partial}{\partial \widehat{y}}(1 + e^{-\widehat{y}}) = -e^{-\widehat{y}}. \tag{3}$$

With (2) and (3), (1) can be expressed via

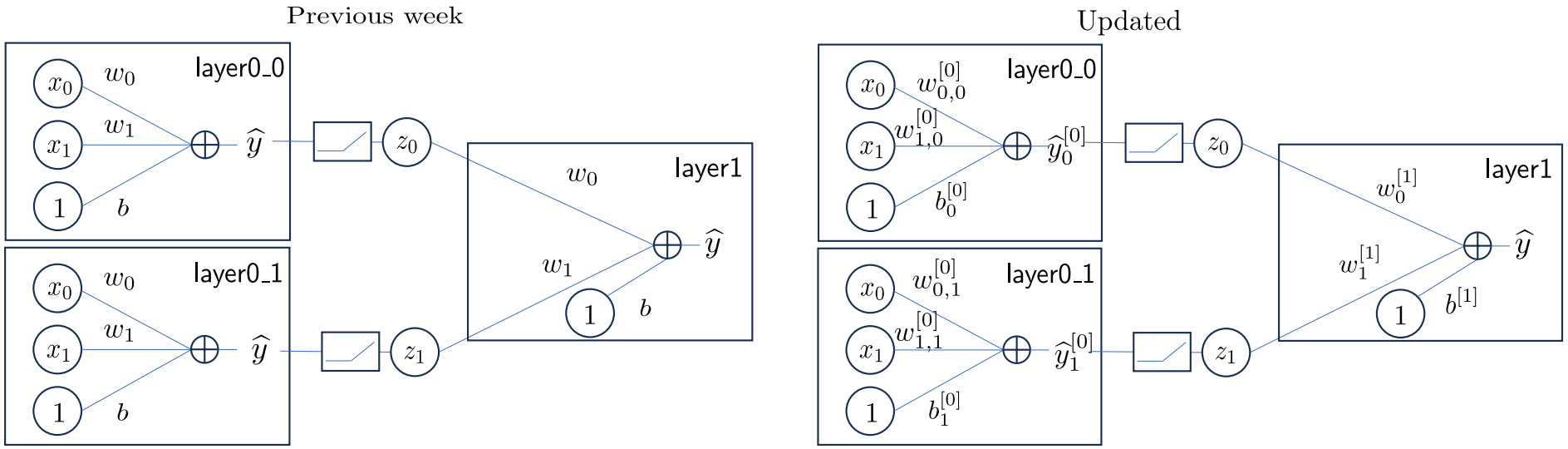
$$\frac{\partial z}{\partial \widehat{y}} = \frac{e^{-\widehat{y}}}{(1 + e^{-\widehat{y}})^2}$$

Ans. 1.4.1:

$$
\begin{aligned}
\frac{\partial z}{\partial \widehat{y}} &= \frac{e^{-\widehat{y}}}{\left(1+e^{-\widehat{y}}\right)^2} \\
&= \frac{e^{-\widehat{y}}+1-1}{\left(1+e^{-\widehat{y}}\right)^2} \\
&= \frac{1}{\left(1+e^{-\widehat{y}}\right)} - \frac{1}{\left(1+e^{-\widehat{y}}\right)^2} \\
&= \frac{1}{\left(1+e^{-\widehat{y}}\right)}\left(1 - \frac{1}{\left(1+e^{-\widehat{y}}\right)}\right) \\
&= \sigma\left(\widehat{y}\right)\left(1-\sigma\left(\widehat{y}\right)\right)
\end{aligned}
$$

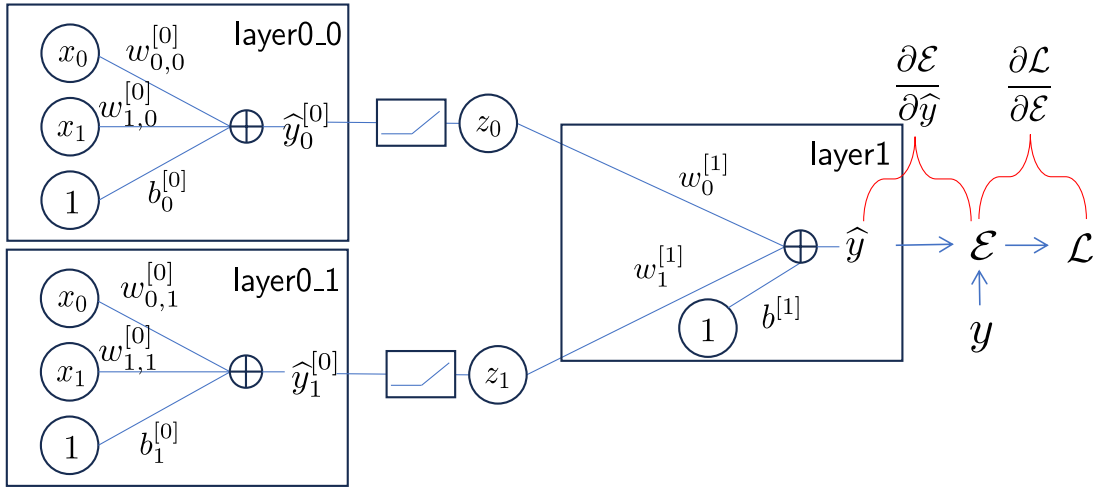


Explanation of notations



The variable $w_{i,j}^{[l]}$ denotes the weight with i, j, and l denoting the indices of the feature space, neuron, and layer, respectively. This does not affect the code and helps with identification for the subsequent exercise.

Eg. 1.5



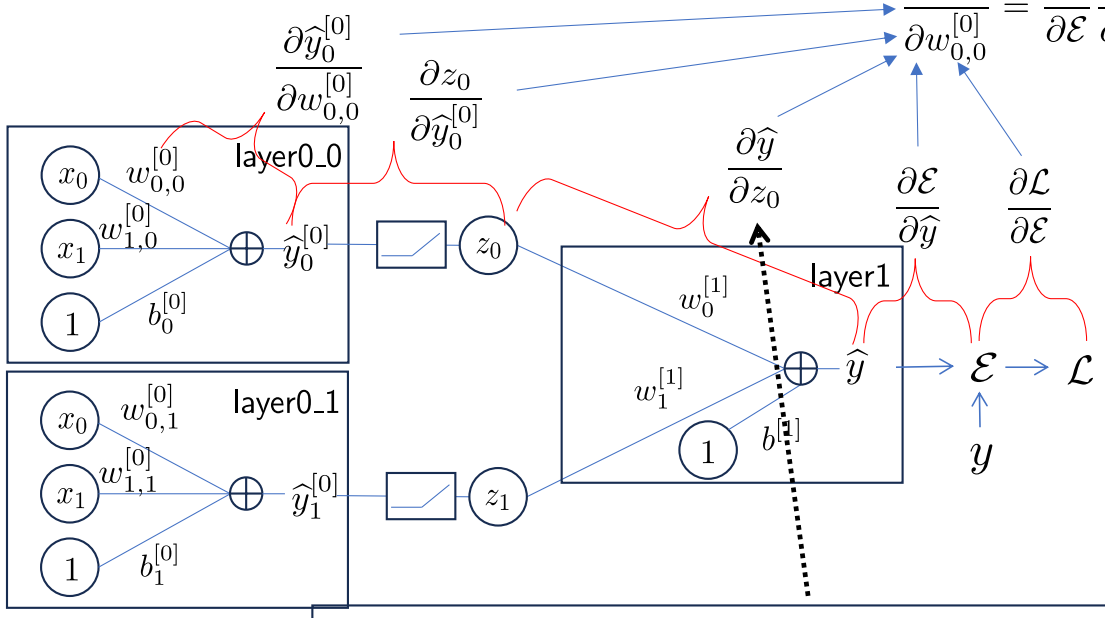
For the above two-layer NN approach employ chain rule and derive equations for

$$\frac{\partial \mathcal{L}}{\partial w^{[0]}_{0,0}}$$

where $\mathcal{L} = \mathcal{E}^2 = (\widehat{y} - y)^2$, $\mathcal{E} = (\widehat{y} - y)$, $\widehat{y} = z_0 w^{[1]}_0 + z_1 w^{[1]}_1 + b^{[1]}$, $z_0 = \mathsf{max}(0, \widehat{y}^{[0]}_0)$, and $z_1 = \mathsf{max}(0, \widehat{y}^{[0]}_1)$.

Ans. 1.5: Visual way

$$\frac{\partial \mathcal{L}}{\partial w_{0,0}^{[0]}} = \frac{\partial \mathcal{L}}{\partial \mathcal{E}} \frac{\partial \mathcal{E}}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_0} \frac{\partial z_0}{\partial \hat{y}_0^{[0]}} \frac{\partial \hat{y}_0^{[0]}}{\partial w_{0,0}^{[0]}}$$



For this part, we do not compute $\frac{\partial \hat{y}}{\partial w_0^{[1]}}$ as further expansion will not lead to $\frac{\partial \hat{y}}{\partial w_0^{[0]}}$.

Ans. 1.5: Mathematical way

Using chain rule,

$$\frac{\partial \mathcal{L}}{\partial w_{0,0}^{[0]}} = \frac{\partial \mathcal{L}}{\partial \mathcal{E}} \frac{\partial \mathcal{E}}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial w_{0,0}^{[0]}},$$

where

$$\begin{aligned}
\frac{\partial \widehat{y}}{\partial w_{0,0}^{[0]}} &= \frac{\partial z_0 w_0^{[1]}}{\partial w_{0,0}^{[0]}} + \underbrace{\frac{\partial z_1 w_1^{[1]}}{\partial w_{0,0}^{[0]}} + \frac{\partial b^{[1]}}{\partial w_{0,0}^{[0]}}}_{=0} \\
&= \frac{\partial z_0 w_0^{[1]}}{\partial z_0} \frac{\partial z_0}{\partial w_{0,0}^{[0]}} \\
&= \frac{\partial z_0 w_0^{[1]}}{\partial z_0} \frac{\partial z_0}{\partial \widehat{y}_0^{[0]}} \frac{\partial \widehat{y}_0^{[0]}}{\partial w_{0,0}^{[0]}}
\end{aligned}$$

Note that z_1, $w_1^{[1]}$ and $b_1^{[1]}$ are not functions of $w_{0,0}^{[0]}$.

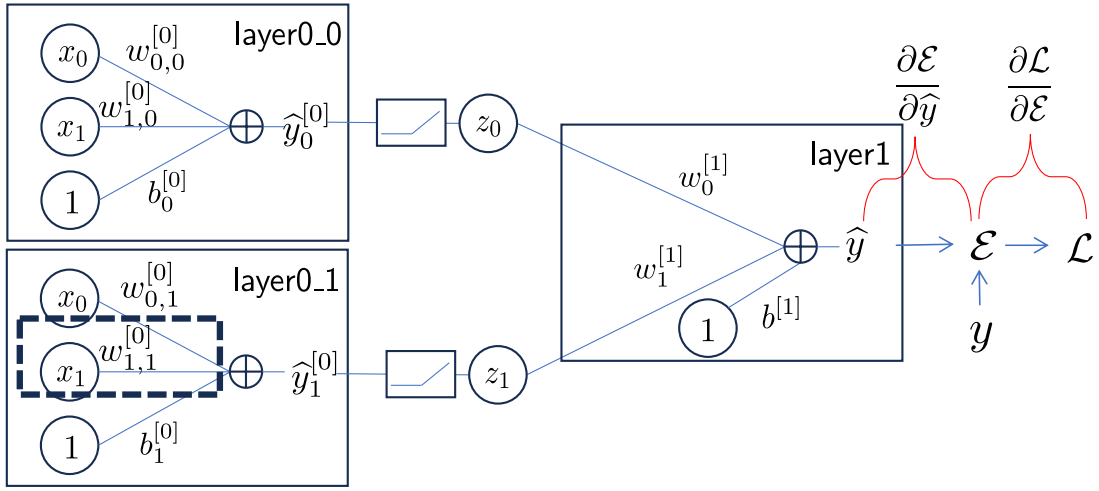
Ans. 1.5: Computation of each component

$$\frac{\partial \mathcal{L}}{\partial w_{0,0}^{[0]}} = \frac{\partial \mathcal{L}}{\partial \mathcal{E}} \frac{\partial \mathcal{E}}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial z_0} \frac{\partial z_0}{\partial \widehat{y}_0^{[0]}} \frac{\partial \widehat{y}_0^{[0]}}{\partial w_{0,0}^{[0]}}$$

$\mathcal{L} = \mathcal{E}^2 = (\widehat{y} - y)^2$, $\mathcal{E} = (\widehat{y} - y)$, $\widehat{y} = z_0 w_0^{[1]} + z_1 w_1^{[1]} + b^{[1]}$, $z_0 = \mathsf{max}(0, \widehat{y}_0^{[0]})$, and $z_1 = \mathsf{max}(0, \widehat{y}_1^{[0]})$.

$$\frac{\partial \mathcal{L}}{\partial \mathcal{E}} = 2\mathcal{E} \qquad \frac{\partial \mathcal{E}}{\partial \widehat{y}} = 1 \qquad \frac{\partial \widehat{y}}{\partial z_0} = w_0^{[1]} \qquad \frac{\partial z_0}{\partial \widehat{y}_0^{[0]}} = \frac{1}{\widehat{y}_0^{[0]}} \mathsf{max}\left(0, \widehat{y}_0^{[0]}\right) \qquad \frac{\partial \widehat{y}_0^{[0]}}{\partial w_{0,0}^{[0]}} = x_0$$

Ex. 1.6



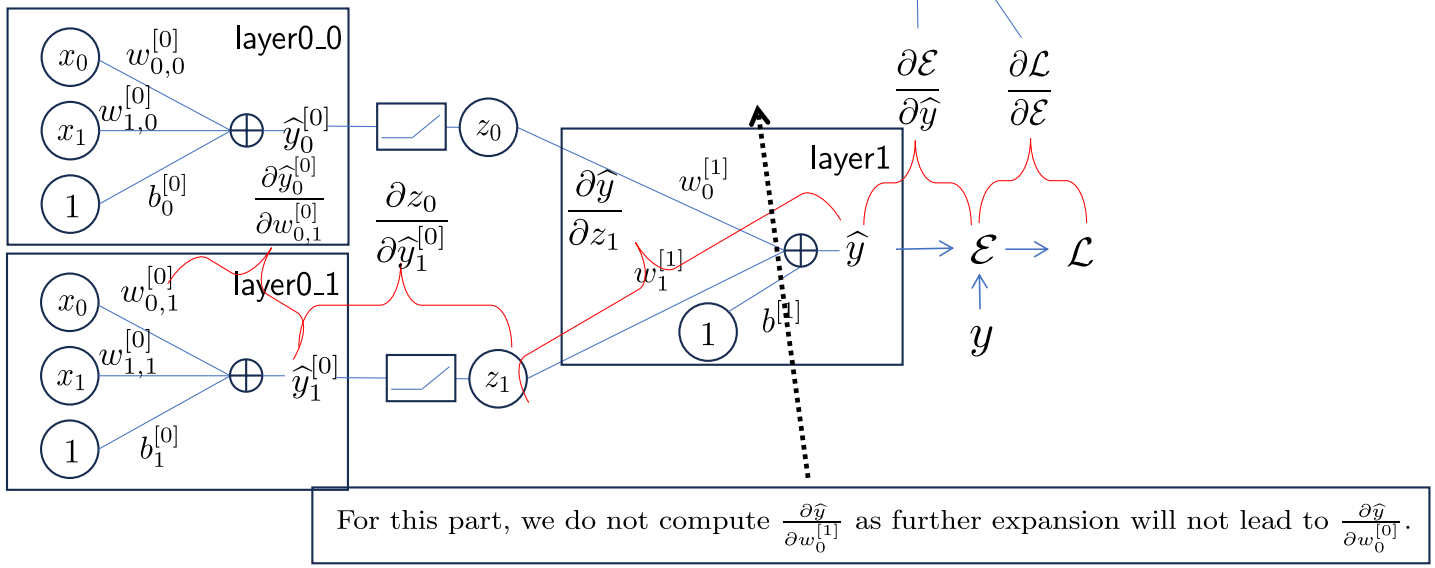
For the above two-layer NN approach employ chain rule and derive equations for

$$\frac{\partial \mathcal{L}}{\partial w^{[0]}_{1,1}}$$

where $\mathcal{L} = \mathcal{E}^2 = (\widehat{y} - y)^2$, $\mathcal{E} = (\widehat{y} - y)$, $\widehat{y} = z_0 w^{[1]}_0 + z_1 w^{[1]}_1 + b^{[1]}$, $z_0 = \mathsf{max}(0, \widehat{y}^{[0]}_0)$, and $z_1 = \mathsf{max}(0, \widehat{y}^{[0]}_1)$.
noindent After that, compute $\frac{\partial \mathcal{L}}{\partial w^{[0]}_{0,0}}$ and compare its value with the one computed by PyTorch.

Ans. 1.6: Visual way

$$\frac{\partial \mathcal{L}}{\partial w_{0,0}^{[0]}} = \frac{\partial \mathcal{L}}{\partial \mathcal{E}} \frac{\partial \mathcal{E}}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial z_0} \frac{\partial z_0}{\partial \widehat{y}_0^{[0]}} \frac{\partial \widehat{y}_0^{[0]}}{\partial w_{0,0}^{[0]}}$$



For this part, we do not compute $\frac{\partial \widehat{y}}{\partial w_0^{[1]}}$ as further expansion will not lead to $\frac{\partial \widehat{y}}{\partial w_0^{[0]}}$.

Ans. 1.6: Computation of each component

$$\frac{\partial \mathcal{L}}{\partial w_{0,1}^{[0]}} = \frac{\partial \mathcal{L}}{\partial \mathcal{E}} \frac{\partial \mathcal{E}}{\partial \widehat{y}} \frac{\partial \widehat{y}}{\partial z_1} \frac{\partial z_1}{\partial \widehat{y}_1^{[0]}} \frac{\partial \widehat{y}_1^{[0]}}{\partial w_{0,1}^{[0]}}$$

$\mathcal{L} = \mathcal{E}^2 = (\widehat{y} - y)^2$, $\mathcal{E} = (\widehat{y} - y)$, $\widehat{y} = z_0 w_0^{[1]} + z_1 w_1^{[1]} + b^{[1]}$, $z_0 = \mathsf{max}(0, \widehat{y}_0^{[0]})$, and $z_1 = \mathsf{max}(0, \widehat{y}_1^{[0]})$.

$$\frac{\partial \mathcal{L}}{\partial \mathcal{E}} = 2\mathcal{E} \qquad \frac{\partial \mathcal{E}}{\partial \widehat{y}} = 1 \qquad \frac{\partial \widehat{y}}{\partial z_1} = w_1^{[1]} \qquad \frac{\partial z_1}{\partial \widehat{y}_1^{[0]}} = \frac{1}{\widehat{y}_1^{[0]}} \mathsf{max}\left(0, \widehat{y}_1^{[0]}\right) \qquad \frac{\partial \widehat{y}_1^{[0]}}{\partial w_{0,1}^{[0]}} = x_0$$

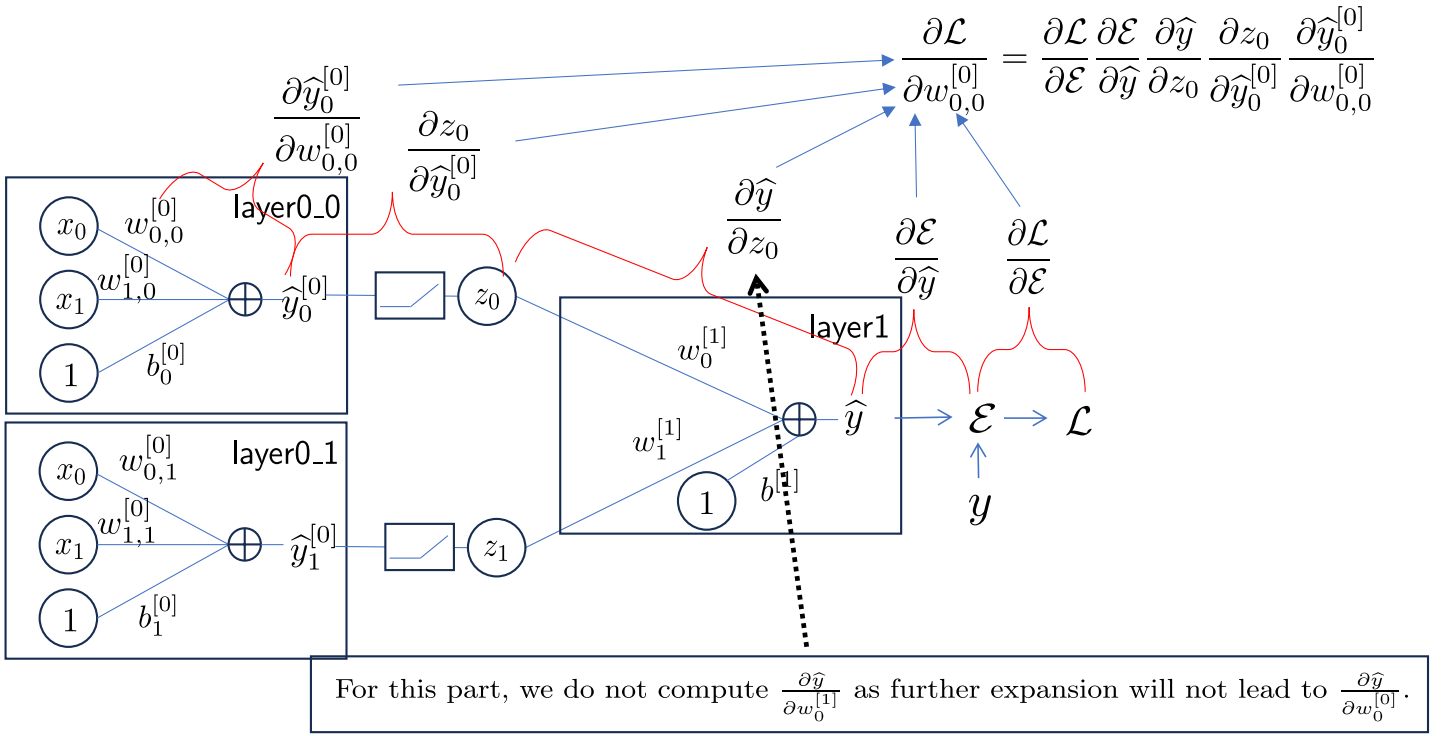
NANYANG TECHNOLOGICAL UNIVERSITY SINGAPORE

EE4414: Week 6

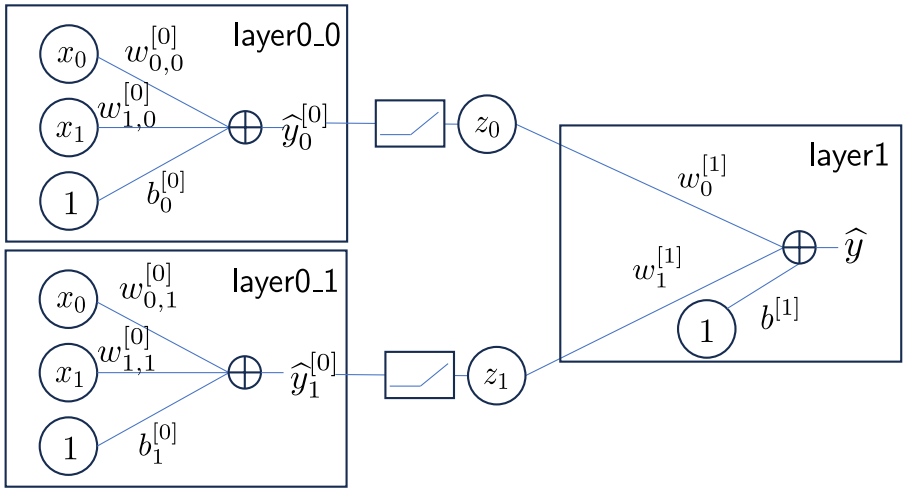
By: Zhi-Wei Tan

Course coordinator: Prof Yap

Review of last week HBL: Backpropagation

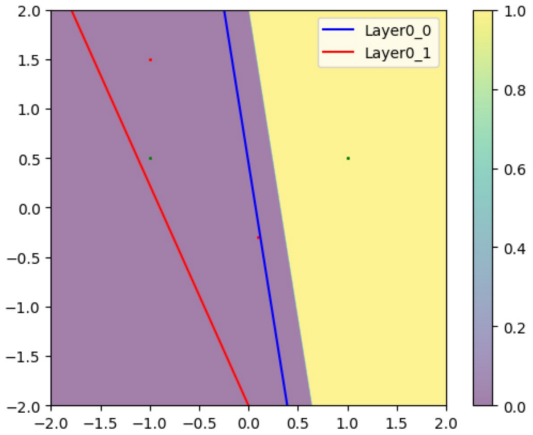


Review of week 4: Training the two-layer NN

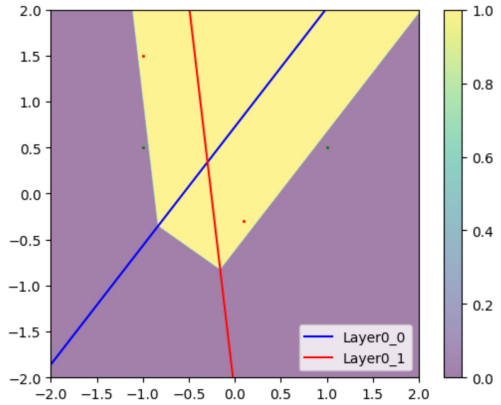


We employed three Linear modules and ReLU activations to construct a two-layer neural network (NN).
We demonstrated that the approach can solve the XOR problem space.

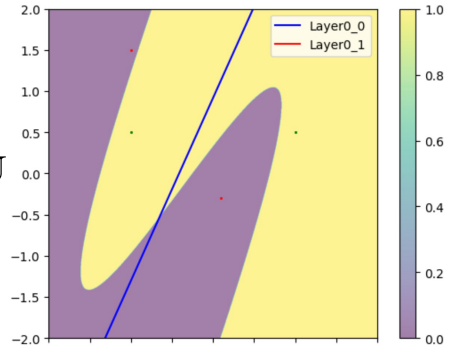
Review of week 4: Hyperparameter search



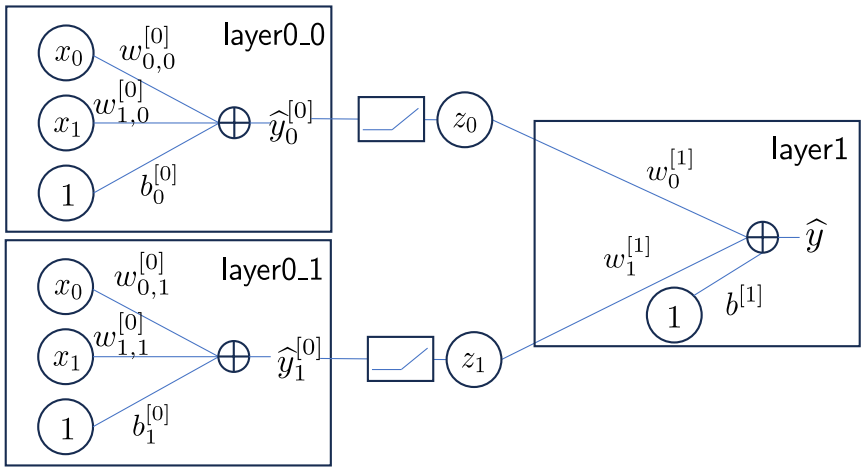
Trained with different initial weights and biases



Employed tanh instead of ReLU as activation function



Ex. 1.1: Random weight initialization



Modify the Linear approach to employ weights drawn from a uniform distribution, i.e., $\mathcal{U}(-0.2, 0.2)$.
Employ a manual seed before initializing the model.

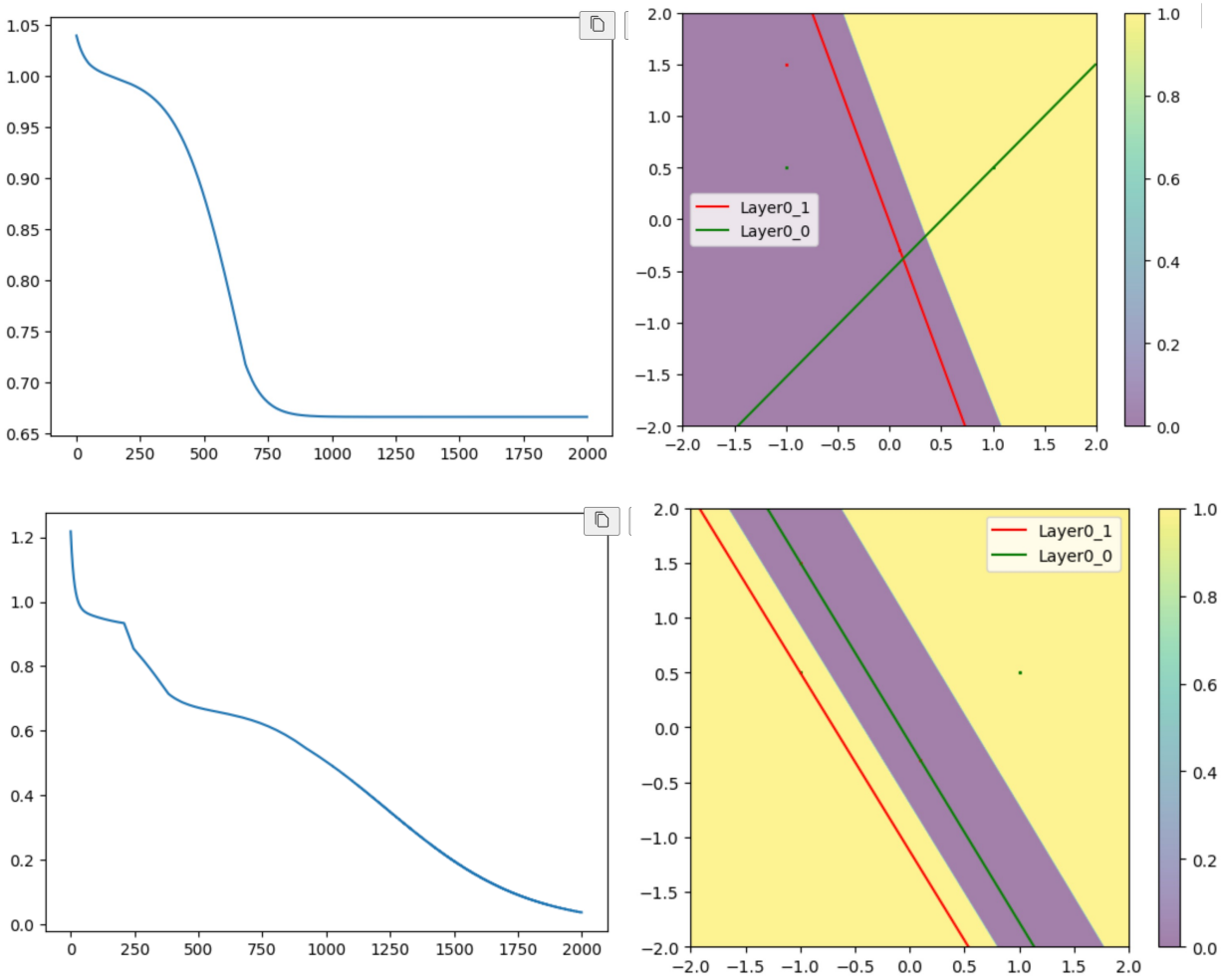
Ans. 1.1

```
def __init__(self,num_in_features=2,
    num_out_features=1):
    super().__init__()
    # use in features and out
    w = torch.FloatTensor(num_in_features, num_out_features)
    w = torch.nn.init.uniform_(w, -0.2, 0.2)
    b = torch.FloatTensor(1)
    b = torch.nn.init.uniform_(b, -0.2, 0.2)
    self.w = torch.nn.Parameter(w, requires_grad=True)
    self.b = torch.nn.Parameter(b, requires_grad=True)
```

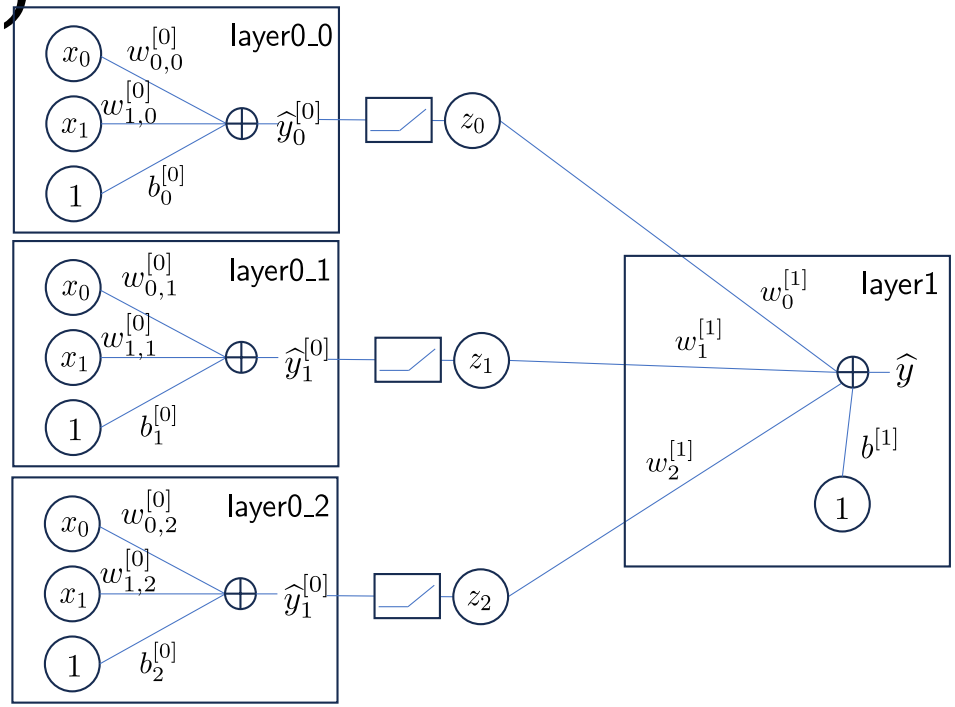
More ways to define is found in torch.nn.init

```
torch.manual_seed(8)
```

```
Linear(2, 1)
```


Layer0_1
Layer0_0
Layer0_1
Layer0_0

Ex. 1.2: Using more neurons to fit (going wider)



Employ an additional neuron in the first layer.
Update the train function to include layer0_2

```
class TwoLayerNN(torch.nn.Module):

    def __init__(self):
        super().__init__()
        self.layer0_0 = Linear(2, 1)
        self.layer0_1 = Linear(2, 1)
        self.layer0_2 = Linear(2, 1)
        self.layer1 = Linear(3, 1)
    def forward(self, x):
        # x: *, 2, 1
        y_hat0_l0 = self.layer0_0.forward(x)
        y_hat1_l0 = self.layer0_1.forward(x)
        y_hat2_l0 = self.layer0_2.forward(x)
        # y_hati_l0: *, 1, 1
        y_hat = torch.cat([y_hat0_l0, y_hat1_l0, y_hat2_l0], dim=-2)
        # y_hat: *, 4, 1
        z = torch.relu(y_hat)
        # z: *, 2, 1
        y_hat = self.layer1.forward(z)

        return y_hat
```

Simpler way to optimize weight and biases

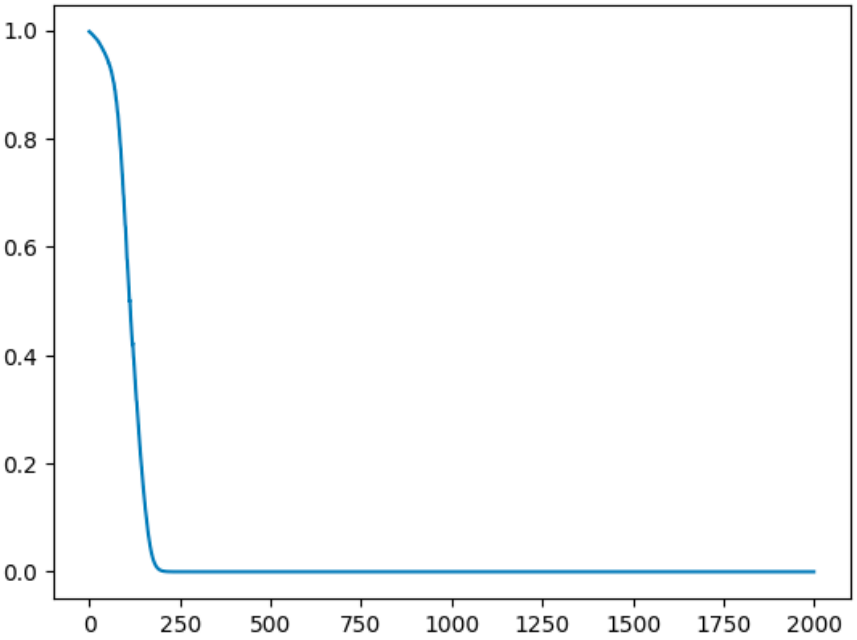
```
for param in model.parameters():
      optimize_weights(param, learning_rate)
```

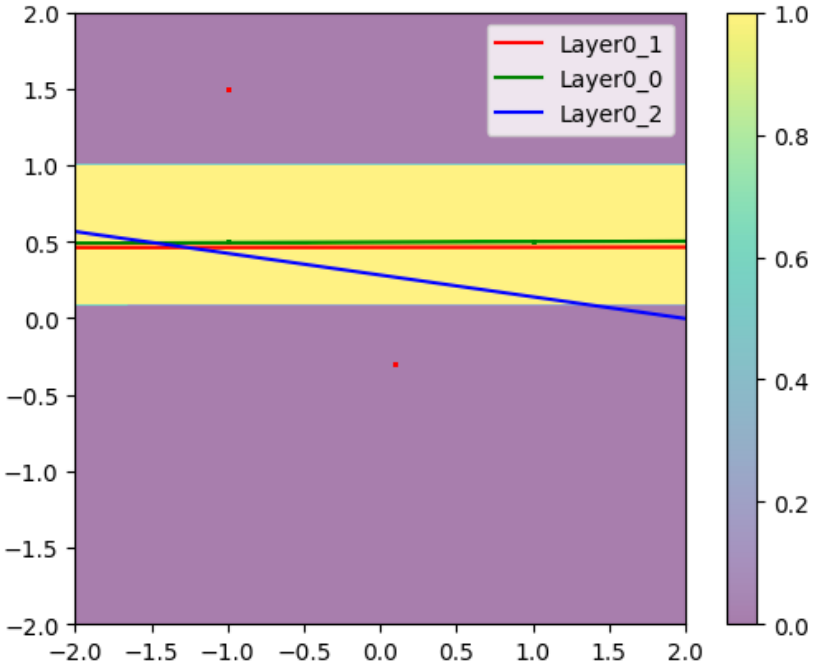
Replace the part where `optimize_weights` is called in `def train()`

```
def optimize_weights(param, learning_rate):

    # update the weights here with learning rate
    param_new = param - learning_rate * param.grad

    # update the model weight
    param.data = param_new
```

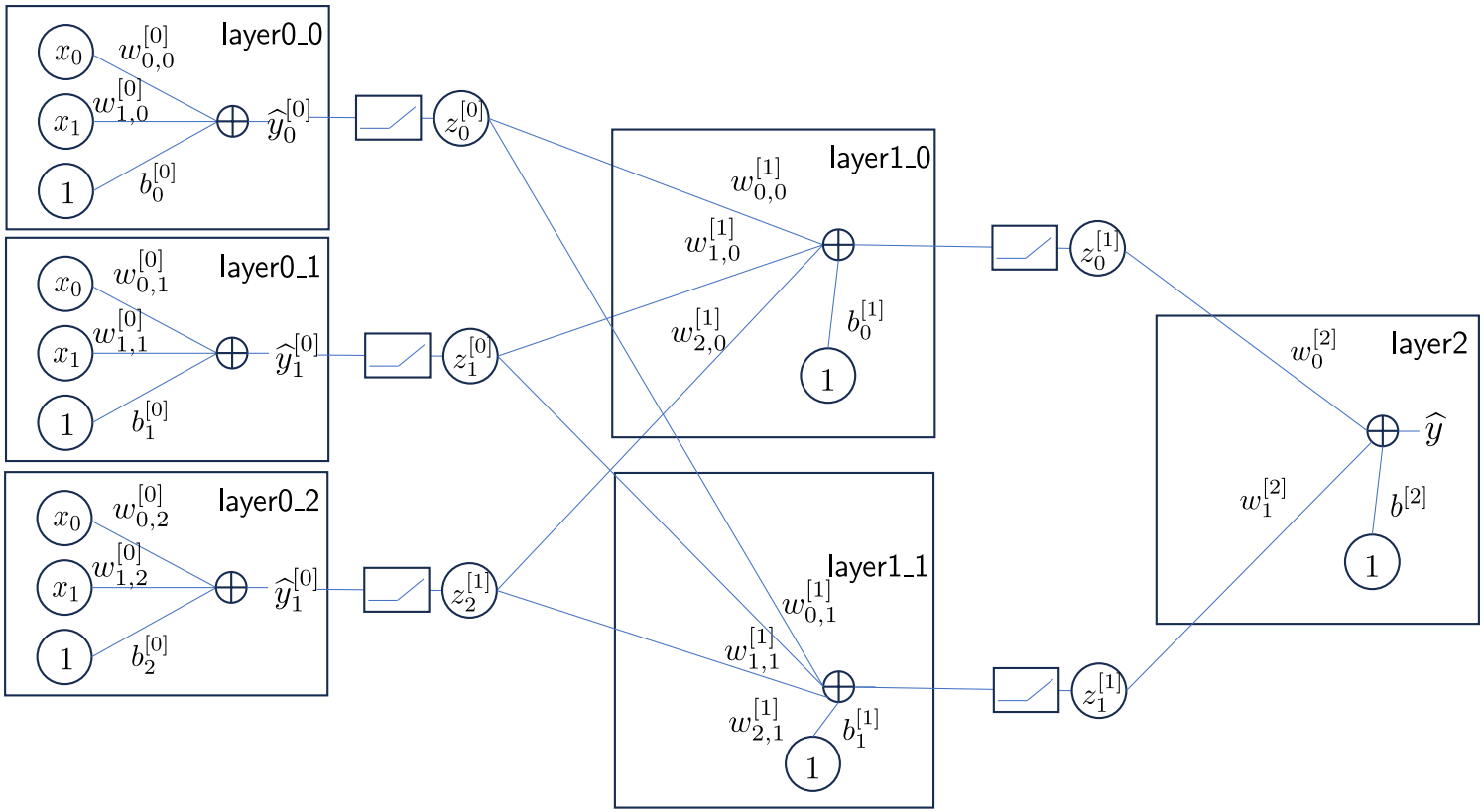
Replace the whole function

An example


1.0
0.8
0.6
0.4
0.2
0.0
0
250
500
750
1000
1250
1500
1750
2000


Layer0_1
Layer0_0
Layer0_2
2.0
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
−2.0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0
1.0
0.8
0.6
0.4
0.2
0.0

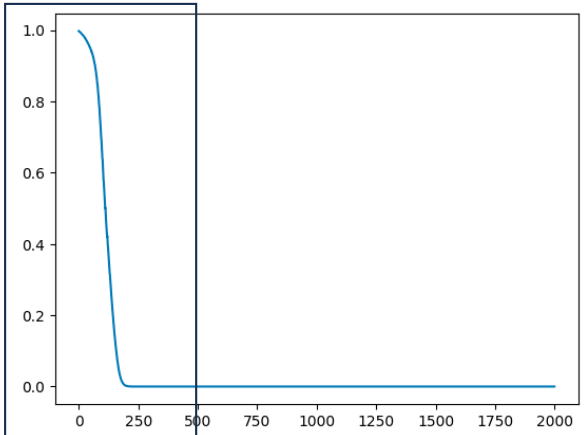
Ex. 1.3: Going deeper…

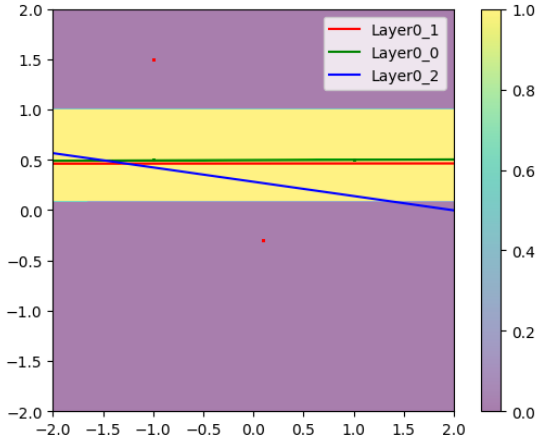


```
class ThreeLayerNN(torch.nn.Module):

    def __init__(self):
    super().__init__()
        self.layer0_0 = Linear(2, 1)
        self.layer0_1 = Linear(2, 1)
        self.layer0_2 = Linear(2, 1)
        self.layer1_0 = Linear(3, 1)
        self.layer1_1 = Linear(3, 1)
        self.layer2 = Linear(2, 1)
```

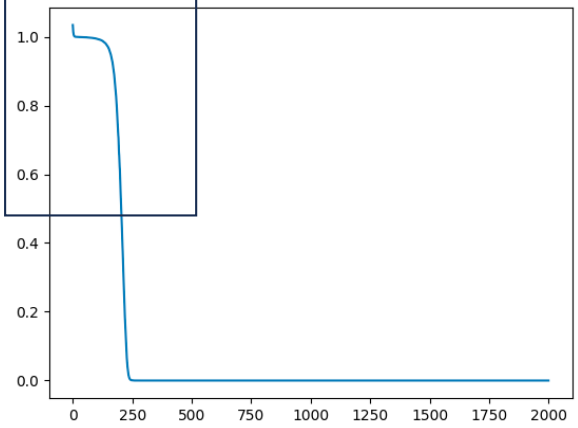
```
def forward(self, x):

    # x: *, 2, 1
    y_hat0_l0 = self.layer0_0.forward(x)
    y_hat1_l0 = self.layer0_1.forward(x)
    y_hat2_l0 = self.layer0_2.forward(x)
    # y_hati_l0: *, 1, 1
    y_hat_l0 = torch.cat([y_hat0_l0, y_hat1_l0, y_hat2_l0], dim=-2)
    # y_hat: *, 4, 1
    z_l0 = torch.relu(y_hat_l0)
    # z: *, 2, 1
    y_hat0_l1 = self.layer1_0.forward(z_l0)
    y_hat1_l1 = self.layer1_1.forward(z_l0)

    y_hat_l1 = torch.cat([y_hat0_l1, y_hat1_l1], dim=-2)
    z_l1 = torch.relu(y_hat_l1)

    y_hat = self.layer2.forward(z_l1)

    return y_hat
```

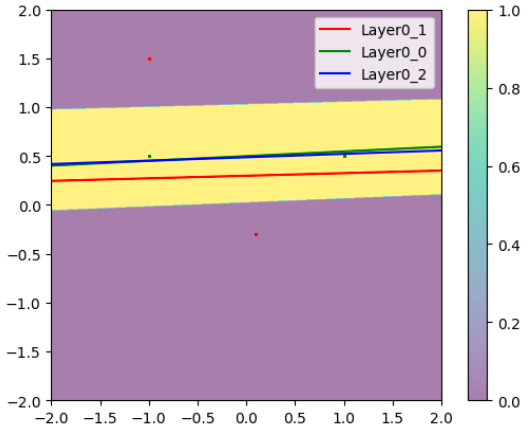
TwoLayerNN





ThreeLayerNN

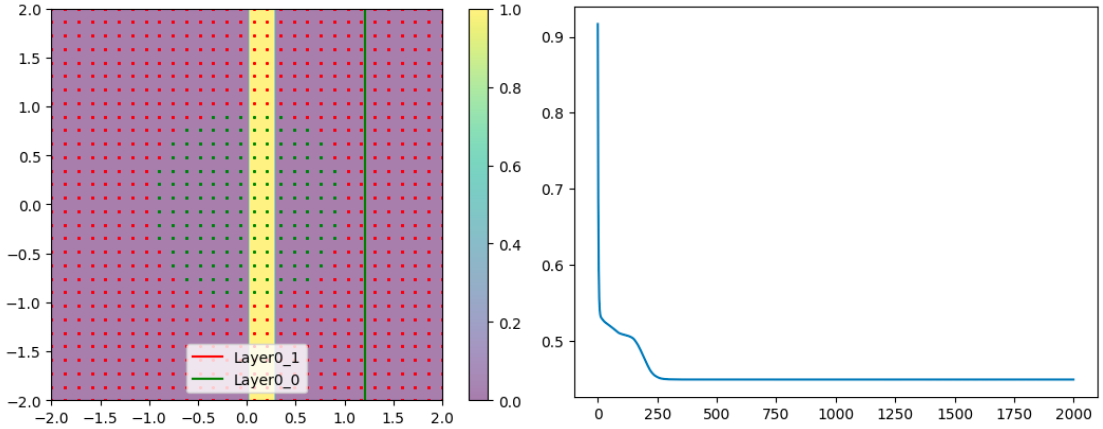




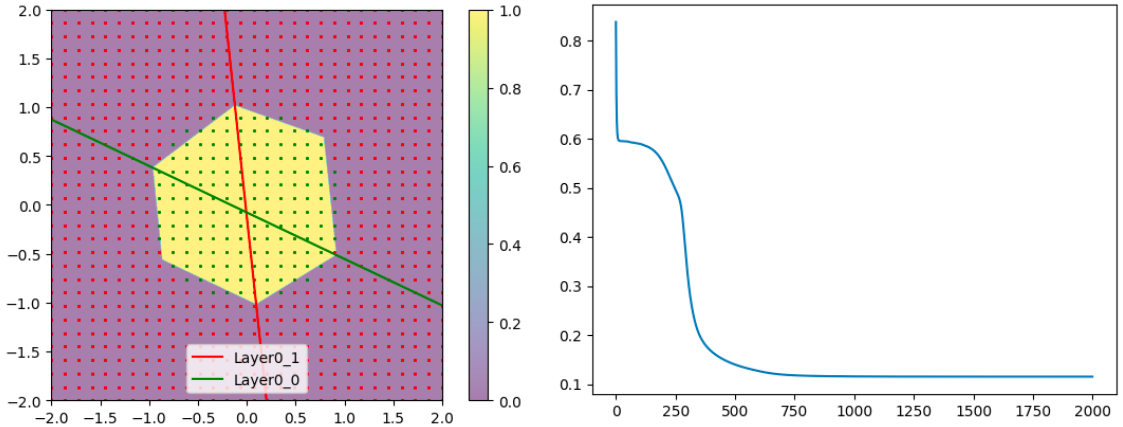
Ex. 1.4: Fitting a circle

- Employ the previous two models (i.e. TwoLayerNN and ThreeLayerNN) and fit the generated circle data

Using two layer



Using three layer



Briefing for practice

In this coding practice, you are required to

1. Submit this paper with each part's print output
2. Email the completed Jupyter Notebook (name as GroupNum-SeatNum-MatricNum) to **zhiwei.tan@ntu.edu.sg** with subject IE4414-GroupNum-SeatNum-MatricNum.

This practice contributes **10%** to your final score, with two sections each contributing **5%**. Each section has multiple questions, and in a newly created Jupyter Notebook, **each question should be answered in a cell** and indicates the question number, e.g., 1.1, via comments or markdown. You should run the Jupyter Notebook in Google Colab (https://colab.research.google.com/).

You are **allowed** to use your laptop and access the internet while doing this practice. However, you are considered **cheating** if you were to take photos, screen share, and/or perform any communication with any external party or parties.

You **will employ your matriculation number** as the variables for the practice unless otherwise stated. Each number in your matriculation number represents a scalar value following the format U$abcdefg$X. For example, if your matriculation number is U1234567C, then $a = 1$, $b = 2$, $c = 3$, and so forth.