Neural-Network-2

Here we will learn rest concept given we will be solving MNIST Lab Problems for DeepML
GitHub Link : Link

Let's understand the problem statement and why it's is a big deal

---

The problem

Description:

Build a complete neural network from scratch using only NumPy. You must implement forward propagation, backward propagation (backprop), and parameter updates manually. No PyTorch, TensorFlow, or autograd libraries allowed. Implement a simple MLP that achieves ≥85% accuracy on MNIST by manually computing all gradients.

Contstraints:

1. Use ONLY numpy - no PyTorch, TensorFlow, JAX, or autograd libraries
2. Manually implement forward and backward passes
3. Manually compute gradients for all parameters
4. Implement at least: Linear layers, ReLU, Softmax, Cross-Entropy loss
5. Must support mini-batch training

Technical Specifications:

Time Limits:

Dev: 120s
Test: 300s

Memory Limit:

2 GB

Model Specifications:

MLP architecture: Input(784) → Hidden(128, ReLU) → Output(10, Softmax). You implement everything from scratch.

Approved Libraries:

`numpy`

Question -1

Why it is such a big deal to write in Numpy ?

1. **Translation of Calculus into code**
2. **The Shape hell**
    1. In PyTorch, you define `Linear(784, 128)` and the library handles the matrix multiplication shapes for you. In NumPy, you have to manually manage the dimensions of every dot product.
    2. In the backward pass, you must transpose matrices (`.T`) to align gradients. A single missing `.T` might not throw an error (if dimensions happen to match coincidentally), but it will produce mathematical garbage, and the model simply won't learn.
3. **Deriving the Gradients (The Calculus)**
4. **Numerical Instability (The `NaN` Trap)**
   You are required to implement **Softmax** and **Cross-Entropy**.
   Naive Implementation: `softmax(x) = exp(x) / sum(exp(x))`
   The Crash: If `x` contains a large number (e.g., 100), `exp(100)` overflows to Infinity. Then `Inf / Inf` becomes `NaN` (Not a Number). Your training crashes instantly.
5. Vectorization is Mandatory (Constraint: 120s)

Approach

1. Achieve Modularity => Code will be long and explanatory
2. Achieve compact while fulling constraints
3. Something that we can submit to Lab website to see what is the accuracy
4. Move to Linear algebra Module (Numpy) to Deep learning Module (PyTorch).
5. Move from neural network approach to some different approach that is ever cleaner and efficient

Here what ever we will Learn as concept can be found in `Neural-Networks-3` Note

---

Let's understand the data

as **Always design the network to fit the data, never the other way around.**
Data-First is the only scalable strategy

Why "Data-First" ?

In software development, you write code to handle inputs. In ML, the **data dictates the architecture**.

- **Dimensionality:** If your data is a time-series of 50 points, a massive ResNet (designed for 224x224 images) is useless.
- **Information Density:** High-density data (images, audio) requires deep, convolutional, or attention-based layers. Low-density data (sparse spreadsheets) often performs better with simpler, shallow dense networks or tree-based models (XGBoost).
- **The "Capacity" Trap:** If you pick a complex network (like a Transformer) for a simple problem (like predicting housing prices from 5 variables), the network will memorize the noise in the training data (overfitting) rather than learning the pattern.

Without starting even a single line of code. we should consider

Phrase-1: The "Data Interrogation"

Before you open your IDE, answer these questions about your data:

1. **What is the Shape?** (Vector? Matrix? Sequence? Graph?) $\rightarrow$ This decides your **Input Layer**.
2. **What is the Distribution?** (Is it Gaussian? Is it skewed? Are values between 0-1 or -1000 to +1000?) $\rightarrow$ This decides your **Normalization** and **Initialization** strategies.
3. **What is the Output?** (Binary class? Multi-class? Real number?) $\rightarrow$ This decides your **Output Layer** (Sigmoid vs Softmax vs Linear) and **Loss Function**.
   we will understand the requirements of more points to ask ourselves with time.

Phase 2: The "Stupid Baseline"

Never start with a complex Neural Network.

- If it's tabular data: Run a Linear Regression or Random Forest.
- If it's images: Run a simple nearest-neighbor or a 1-layer perceptron.
- **Why?** You need a baseline. If your massive, complex Neural Network cannot beat a simple Random Forest, your network design is flawed, or the data doesn't have enough signal for deep learning.

Phase 3: Network Design (The "Fun" Part)

Now that you understand the data, you build the network to solve specific problems the data presents:

- *Data has spatial correlation (images)?* $\rightarrow$ Add **Conv2D** layers.
- *Data has temporal dependence (text/stock prices)?* $\rightarrow$ Add **RNN/LSTM** or **Attention** mechanisms.

- *Network is dying (gradients vanishing)?* → Add **Skip Connections** (Residuals) or **Batch Norm**.

There is a exception in **Transfer Learning**, where you pick the model first and adjust the data to fit it

If you are doing Image Classification, you don't reinvent the wheel. You take a standard architecture (like ResNet50 or EfficientNet) that is proven to work.

- **Why?** Because these models have learned "universal features" (edges, textures) from millions of images.
- **The Trade-off:** In this case, you *must* resize/crop your images to 224x224 and normalize them exactly how the model expects, because you are leveraging pre-existing weights.

There are multiple ways to get the MNIST Data

1. from kaggle
2. from author's website
3. There are many libraries

Before even testing with a large datasets. We should follow "Unit Testing" hierarchy to save yourself hours of debugging.

Level 1: The "Sanity Check" (Synthetic Data)

Before you even touch MNIST, can your network solve `XOR` or `make_moons`?

- **Source:** Generate it yourself inside the script.
- **Why:** It is 2D. You can visualize the decision boundary. If your network can't separate two curved lines, it definitely won't recognize a handwritten "7".

Level 2: The "Integration Test" (MNIST)

Once `make_moons` works, move to MNIST.
Why: It is high-dimensional (784 inputs) but highly structured. It has a known "solution"
What if it has not !! We will see.

Best Practice : Load Data function

```
def load_mnist():
    """
    Downloads and parses MNIST dataset using only numpy/standard lib.
    """
    base_url = "https://github.com/rossbar/numpy-tutorial-data-
mirror/blob/main/"
    data_sources = {
```

```
        "training_images": "train-images-idx3-ubyte.gz",  # 60,000 training
images.
        "test_images": "t10k-images-idx3-ubyte.gz",  # 10,000 test images.
        "training_labels": "train-labels-idx1-ubyte.gz",  # 60,000 training
labels.
        "test_labels": "t10k-labels-idx1-ubyte.gz",  # 10,000 test labels.
    }
    # Use responsibly! When running notebooks locally, be sure to keep local
    # copies of the datasets to prevent unnecessary server requests
    headers = {
        "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:10.0)
Gecko/20100101 Firefox/10.0"
    }
    request_opts = {
        "headers": headers,
        "params": {"raw": "true"},
    }

    data_dir = "../_data"
    os.makedirs(data_dir, exist_ok=True)

    for fname in data_sources.values():
        fpath = os.path.join(data_dir, fname)
        if not os.path.exists(fpath):
            print("Downloading file: " + fname)
            resp = requests.get(base_url + fname, stream=True,
**request_opts)
            resp.raise_for_status()  # Ensure download was succesful
            with open(fpath, "wb") as fh:
                for chunk in resp.iter_content(chunk_size=128):
                    fh.write(chunk)
    mnist_dataset = {}
    # Images
    for key in ("training_images", "test_images"):
        with gzip.open(os.path.join(data_dir, data_sources[key]), "rb") as
mnist_file:
            mnist_dataset[key] = np.frombuffer(
                mnist_file.read(), np.uint8, offset=16
            ).reshape(-1, 28 * 28)

    # Labels
    for key in ("training_labels", "test_labels"):
        with gzip.open(os.path.join(data_dir, data_sources[key]), "rb") as
```

```
mnist_file:
            mnist_dataset[key] = np.frombuffer(mnist_file.read(), np.uint8,
offset=8)

    x_train, y_train, x_test, y_test = (
      mnist_dataset["training_images"],
      mnist_dataset["training_labels"],
      mnist_dataset["test_images"],
      mnist_dataset["test_labels"],
    )
    return x_train, y_train, x_test, y_test
```

Which could have done in a single line in PyTorch

```
from torchvision import datasets, transforms
train_dataset = datasets.MNIST(root='./data', train=True, download=True,
transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True,
transform=transform)
```

but these are objects in torch. and another thing is, as MNIST is popular it is already in the library. but most datasets we have to write custom `Data Loader`. (it is just one of the lesson we will take from this implementation)

Let's comeback to the data loader code

Writing a data loader from scratch (like load_mnist or the custom folder loader) is a skill called **Data Engineering**. It is distinct from designing neural networks but is arguably 80% of the work in real-world AI.

The Concept: The "ETL" Pipeline

Every data loader follows the **ETL** pattern: **E**xtract, **T**ransform, **L**oad.

1. EXTRACT (Get the raw bytes)

- **Question:** Where does the data live and what format is it in?
- **Scenario A (Web):** It's a URL. -> Use urllib or requests.
- **Scenario B (Disk):** It's a folder structure. -> Use os.walk or glob.
- **Scenario C (Compressed):** It's .zip or .gz. -> Use zipfile or gzip.
- **Thought:** I need to get the file handle open so I can read the raw binary 0s and 1s.

2. TRANSFORM (Parse and Formatting)

- **Question:** How do I turn "bytes" into "numbers"?
- **The Spec Check:** You must look up the file specification.
  - MNIST Spec: "The first 16 bytes are header info. The rest are pixels." -> **Action:** f.read(16) to skip, f.read() to get data.
  - Image Spec (JPG/PNG): "Complex compression headers." -> **Action:** Don't write a parser manually. Use PIL.Image.open() or cv2.imread().
  - CSV Spec: "Text separated by commas." -> **Action:** line.split(',').
- **The Conversion:**
  - Binary buffer -> np.frombuffer (Fastest)
  - Image object -> np.array(image)
  - String -> float(string)

3. LOAD (Normalization & Batching)

- **Question:** What does my Neural Network expect?
- **Shape:** (Batch_Size, Dimensions). Do I need to flatten it? img.reshape(-1, 784).
- **Type:** Neural nets hate Integers (0-255). They love Floats (0.0-1.0). -> **Action:** .astype(np.float32) / 255.0.
- **Labels:** Are they strings ("cat")? -> **Action:** Map them to integers (0).

Step-by-Step Guide to writing load_mnist

Here is exactly how I wrote that function, line by line, based on this thought process.

Step 1: Read the Documentation

as the documentation is not in the original website Link. but
I would have went to the MNIST website.

Where it is clearly written,
MNIST uses a file format called **IDX**, which is an old-school, rigid, binary layout.
It has nothing to do with gzip itself. Link

the **IDX file header is exactly 16 bytes for images**.
That's the format definition.

IDX Format (Images)

Offset	Size	Meaning
0–3	4 bytes	magic number

Offset	Size	Meaning
4–7	4 bytes	number of images
8–11	4 bytes	number of rows
12–15	4 bytes	number of cols
16+	remainder	pixel data, unsigned bytes

My Takeaway: The data I actually want (the pixels) starts at **byte 16**. Everything before that is metadata.

Step 2: Handle the Compression

The file ends in .gz. Python has a built-in library for this, called gzip

```
import gzip
with gzip.open(filepath, "rb") as f: # 'rb' = Read Binary
    ...
```

Step 3: Fast Reading (Buffers vs Loops)

Now you need to read the data.

- **The naive way (Slow):** Read one byte, convert to int, read next byte... Python loops are too slow for 60,000 * 784 bytes.
- **The NumPy way (Fast):** "Hey OS, take this chunk of memory and pretend it is a NumPy array."

This is where `np.frombuffer` comes in

```
# The spec said the data starts at byte 16.
# So, we tell numpy: "Skip the first 16 bytes (offset=16),
# and treat the rest as unsigned 8-bit integers (uint8)."
data = np.frombuffer(file_content, dtype=np.uint8, offset=16)
```

Step 4: Reshaping (The Logic)

np.frombuffer returns a 1D array (a long flat line of numbers). You need to structure it.

- **Thought:** "I know I have 60,000 images, and each is 28x28."
- **Logic:**
  - If I want a flat input for a Linear Layer: reshape(-1, 784)
  - If I want images for a CNN: reshape(-1, 28, 28)

(Note: -1 tells NumPy "You calculate this dimension based on the total size".)

The code now become

```
with gzip.open(train_file, "rb") as mnist_file:
    train_data = np.frombuffer(
        mnist_file.read(), np.uint8, offset=16
        ).reshape(-1, 28 * 28)
```

but we have 4 files. `training_images` , `test_images` , `training_labels` , `test_labels` so we have to handle them in the same way. so i can use for loop.

Let's talk about downloading the data

1. we can download the whole data and then read, convert or reshape (This will consume excessive memory if the file is very large.)
2. or we can allows to iterate over the response content in chunks instead of downloading the entire response content into memory.

we will see the 2nd option

there is a base URL :

```
base_url = "https://github.com/rossbar/numpy-tutorial-data-
mirror/blob/main/"
```

There are 4 files to download

https://github.com/rossbar/numpy-tutorial-data-mirror/blob/main/train-images-idx3-ubyte.gz

https://github.com/rossbar/numpy-tutorial-data-mirror/blob/main/t10k-images-idx3-ubyte.gz

https://github.com/rossbar/numpy-tutorial-data-mirror/blob/main/train-labels-idx1-ubyte.gz

https://github.com/rossbar/numpy-tutorial-data-mirror/blob/main/t10k-labels-idx1-ubyte.gz

in which the file name differ so

```
data_sources = {
    "training_images": "train-images-idx3-ubyte.gz",  # 60,000 training
images.
    "test_images": "t10k-images-idx3-ubyte.gz",  # 10,000 test images.
    "training_labels": "train-labels-idx1-ubyte.gz",  # 60,000 training
labels.
    "test_labels": "t10k-labels-idx1-ubyte.gz",  # 10,000 test labels.
}
```

Now we just have to add `base_url` + `data_sources[i]`

we created a dict to support both download and read logic we discussed above. How ? let's see

Before that there is 2 more things to discuss

1. How can we download a file with python

we use built-in library called `request`.

```
resp = requests.get(url, stream=True, **request_opts)
```

requests.get(...): This is the core function from the requests library that sends an HTTP GET request to the specified URL.

- `url` : the full URL for the file to be downloaded
- `stream` : This is crucial for downloading large files efficiently. When stream=True, the requests library doesn't immediately download the entire response content into memory. Instead, it allows you to iterate over the response content in chunks (as seen in the `for chunk in resp.iter_content(chunk_size=128):` loop later). This prevents your program from consuming excessive memory if the file is very large.
- `other arguments` : - `request_opts `**: This is a Python syntax for "unpacking" a dictionary into keyword arguments. It's equivalent to writing headers=headers, params={"raw": "true"} directly inside the requests.get()` call. This allows for a more organized way to pass multiple options.

- **resp**: The return value is a Response object from the requests library. This object contains all the information about the server's response, including the status code, headers, and the content of the downloaded file.

why we need the header and params in `request` object?

let's see

```
# Use responsibly! When running notebooks locally, be sure to keep local
# copies of the datasets to prevent unnecessary server requests
headers = {
    "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:10.0) Gecko/20100101
Firefox/10.0"
}

request_opts = {
    "headers": headers,
    "params": {"raw": "true"},
}
```

- **headers**: This is a Python dictionary that defines HTTP headers. When your program makes an HTTP request to a server, it sends these headers along with the request.
- **"User-Agent"**: This is a standard HTTP header that identifies the client (the program or browser) making the request. Many web servers check the User-Agent header.
- `request_opts` : Another dictionary, this one is used to collect various options that will be passed to the `requests.get()` function. This makes the call to `requests.get()` cleaner, as you can pass all options together.
  - `"headers": headers` : This tells the `requests` library to use the `headers` dictionary we defined above for this request.
  - `"params": {"raw": "true"}` : This specifies URL query parameters.
  - **What are query parameters?** They are key-value pairs appended to a URL after a question mark (e.g., `www.example.com/api?key1=value1&key2=value2`).
  - **Why `{"raw": "true"}` ?** The `base_url` points to a GitHub repository mirror. When you navigate to a file on GitHub, you usually see the file rendered within the GitHub website interface. To get the actual, raw content of the file (e.g., the pure binary data of a `.gz` file), you often need to add a parameter like `?raw=true` to the URL. This dictionary item will be translated by `requests` into that URL parameter.

    > this applicable to this case of download may not work for all case

- **resp.raise_for_status()**: This method is called on the Response object to check if the HTTP request was successful.
  - If the HTTP status code is between 200 and 299 (indicating success, like 200 OK), this method does nothing.
  - If the status code indicates an error (e.g., 404 Not Found, 500 Internal Server Error, 403 Forbidden), it will **raise an HTTPError exception**.
  - **Purpose**: This is a concise way to ensure that the download actually started correctly before attempting to process the (potentially empty or error-containing) response content. If an error occurs, the program will stop early with a clear exception, rather than trying to write corrupt or incomplete data.

2. we have to store it somewhere right ?

```
import os
data_dir = "../_data"
os.makedirs(data_dir, exist_ok=True)
```

and for file name to store in the gzip portion in the loop

```
fpath = os.path.join(data_dir, fname)
```

where the fname is a key from `data_source` we defined earlier.

Now the code become

```
base_url = "https://github.com/rossbar/numpy-tutorial-data-
mirror/blob/main/"

data_sources = {

  "training_images": "train-images-idx3-ubyte.gz",  # 60,000 training
images.
  "test_images": "t10k-images-idx3-ubyte.gz",  # 10,000 test images.
  "training_labels": "train-labels-idx1-ubyte.gz",  # 60,000 training
labels.
  "test_labels": "t10k-labels-idx1-ubyte.gz",  # 10,000 test labels.

}

# Use responsibly! When running notebooks locally, be sure to keep local
# copies of the datasets to prevent unnecessary server requests

headers = {

    "User-Agent": "Mozilla/5.0 (X11; Linux x86_64; rv:10.0) Gecko/20100101
Firefox/10.0"

}

request_opts = {

    "headers": headers,
    "params": {"raw": "true"},

}

data_dir = "../_data"
os.makedirs(data_dir, exist_ok=True)

for fname in data_sources.values():
    fpath = os.path.join(data_dir, fname)
    if not os.path.exists(fpath):
        print("Downloading file: " + fname)
        resp = requests.get(base_url + fname, stream=True, **request_opts)
        resp.raise_for_status()  # Ensure download was succesful
        with open(fpath, "wb") as fh:
```

```
            for chunk in resp.iter_content(chunk_size=128):
                fh.write(chunk)

mnist_dataset = {} # another dictionary to help in return statement

# Images
for key in ("training_images", "test_images"): # for execution of logic for
both training images and test images
    with gzip.open(os.path.join(data_dir, data_sources[key]), "rb") as
mnist_file:
        mnist_dataset[key] = np.frombuffer(
            mnist_file.read(), np.uint8, offset=16
        ).reshape(-1, 28 * 28)

# Labels
    for key in ("training_labels", "test_labels"):
        with gzip.open(os.path.join(data_dir, data_sources[key]), "rb") as
mnist_file:
            mnist_dataset[key] = np.frombuffer(mnist_file.read(), np.uint8,
offset=8)
```

the `mnist_dataset` dictionary would look like this

```
{
    "training_images": [NumPy Array (60000, 784)],
    "test_images": [NumPy Array (10000, 784)],
    "training_labels": [NumPy Array (60000,)],
    "test_labels": [NumPy Array (10000,)]
}
```

Now it's downloaded and stored in appropriate format but we have to pack it into a function to reuse it as many time as possible.

this is how the code written at the top for loading mnist dataset called `load_mnist`

1. Modular and Explanation

Colab: Link

1. why Base class ?

```
class Layer:
  """
```

```
    Base Class for all layers
    """
    def forward(self, input):
      raise NotImplementedError

    def backward(self, grad_output):
      raise NotImplementedError
```

In Python, you strictly *don't* need a base class due to **duck typing**. However, including it serves two critical purposes in systems design:

1. **The Contract (Interface Enforcement)**
   - The base class defines a **contract**. It tells any future developer (or you, three months from now) that *every* layer must implement `forward(x)` and `backward(grad)`.
   - Without this, you might accidentally name a method `predict()` in one layer and `forward()` in another, breaking your training loop.
   - It allows you to iterate generic lists (like `layers = [Linear, ReLU, Linear]`) without checking types. The loop just calls `.forward()` blindly, trusting the contract.
2. **Shared Utility (Don't Repeat Yourself)**
   In a more advanced version, the Base Class becomes the home for shared logic, such as:
   - **Serialization:** A `save_weights()` method that works for *any* layer.
   - **Training Modes:** A flag like `self.training = True` (critical for Dropout or Batch Norm) is best stored in the base class so you can toggle the whole network with `model.train()` or `model.eval()`.
     don't get overwhelmed, we will see everything in detail #implementation

Linear layer

```
class Linear(Layer):
    def __init__(self, input_dim, output_dim):
        # He Initialization (Good for ReLU)
        self.W = np.random.randn(input_dim, output_dim) * np.sqrt(2.0 /
input_dim)
        self.b = np.zeros((1, output_dim))

        # Gradients
        self.dW = None
        self.db = None

        # Cache for backward pass
        self.input_cache = None
```

```
    def forward(self, x):
        """
        x: shape (batch_size, input_dim)
        """
        self.input_cache = x
        return np.dot(x, self.W) + self.b

    def backward(self, grad_output):
        """
        grad_output: gradient of loss w.r.t output of this layer
                     shape (batch_size, output_dim)
        """
        # 1. Calculate Gradients for Weights and Biases
        self.dW = np.dot(self.input_cache.T, grad_output)
        self.db = np.sum(grad_output, axis=0, keepdims=True)

        # 2. Calculate Gradient for the Input (to pass to the previous
layer)
        grad_input = np.dot(grad_output, self.W.T)

        return grad_input
```

Q1. What Linear Layer does ?

At its core, a Linear Layer (also called a Fully Connected or Dense layer) performs a **linear transformation** on the data.
Mathematically, it calculates the equation of a line (or hyperplane) in high dimensions:

$$y = xW + b$$

- x **(Input):** The features coming in (e.g., pixel values).
- W **(Weights):** The "strength" of the connection between inputs and outputs. This is what the network learns.
- b **(Bias):** An offset that allows the activation to shift up or down, independent of the input.

that why we did the following in forward pass

```
return np.dot(x, self.W) + self.b
```

Q2. why would i consider forward and backward layer ?

Neural Networks work in a two-step cycle. You cannot train a network without both.

1. **Forward (Prediction):**

- Data flows from Input $\rightarrow$ Output.
- We calculate the result (scores) using current weights.
- **Goal:** Calculate the Loss (Error).

2. **Backward (Learning):**
   - Error information flows from Output $\rightarrow$ Input.
   - We use the **Chain Rule** of calculus to figure out *who is responsible for the error*.
   - **Goal:** Calculate Gradients (dW, db) to update the weights so the error is lower next time.

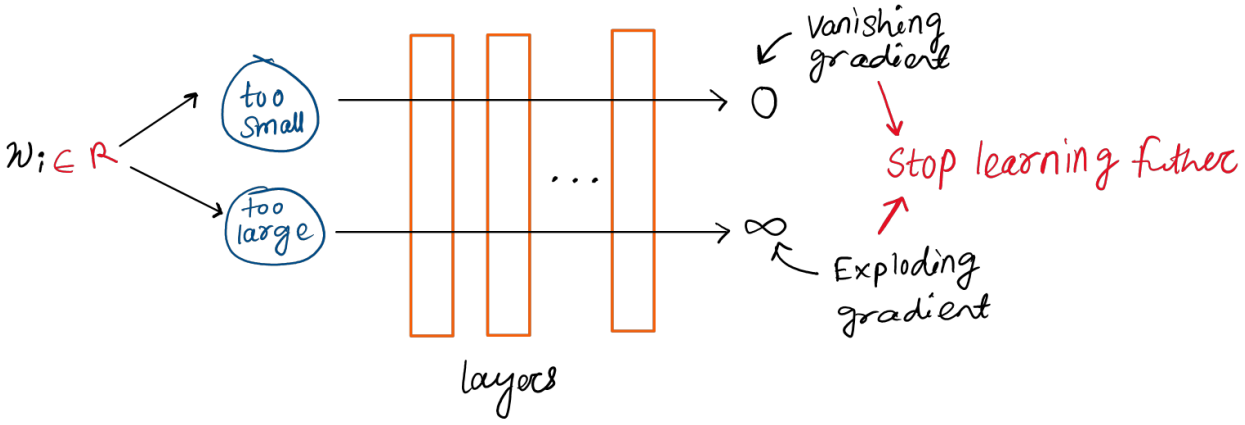
If you only implemented `forward`, you would have a prediction machine that never learns.

Q3. What is He initialization ? How it is good for ReLU networks instead of random initialization ?

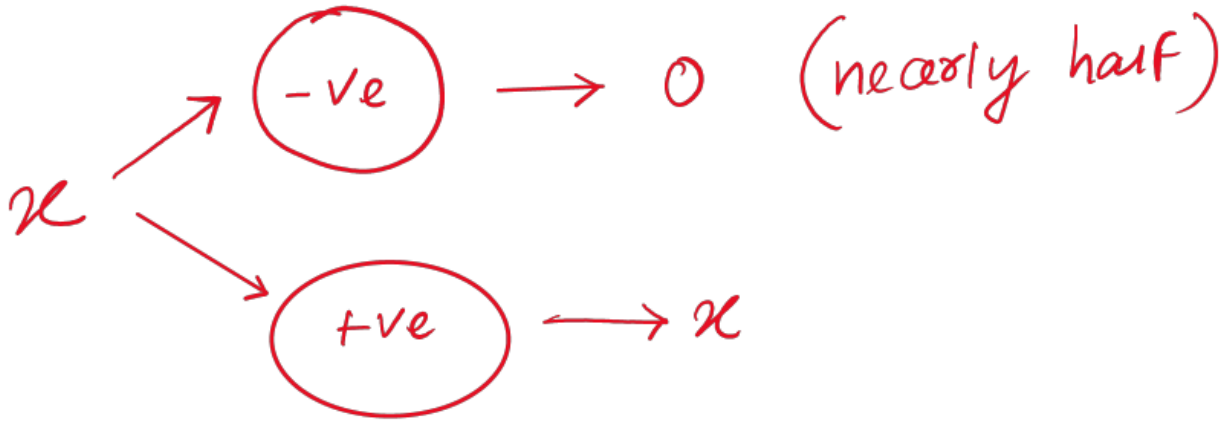
Source: Link

The Problem:
If you initialize weights randomly (standard normal distribution), the values can be too small or too big.

- **Too small:** As data passes through many layers, the signal shrinks to zero ("Vanishing Gradients"). The network stops learning.
- **Too large:** The signal explodes to infinity.



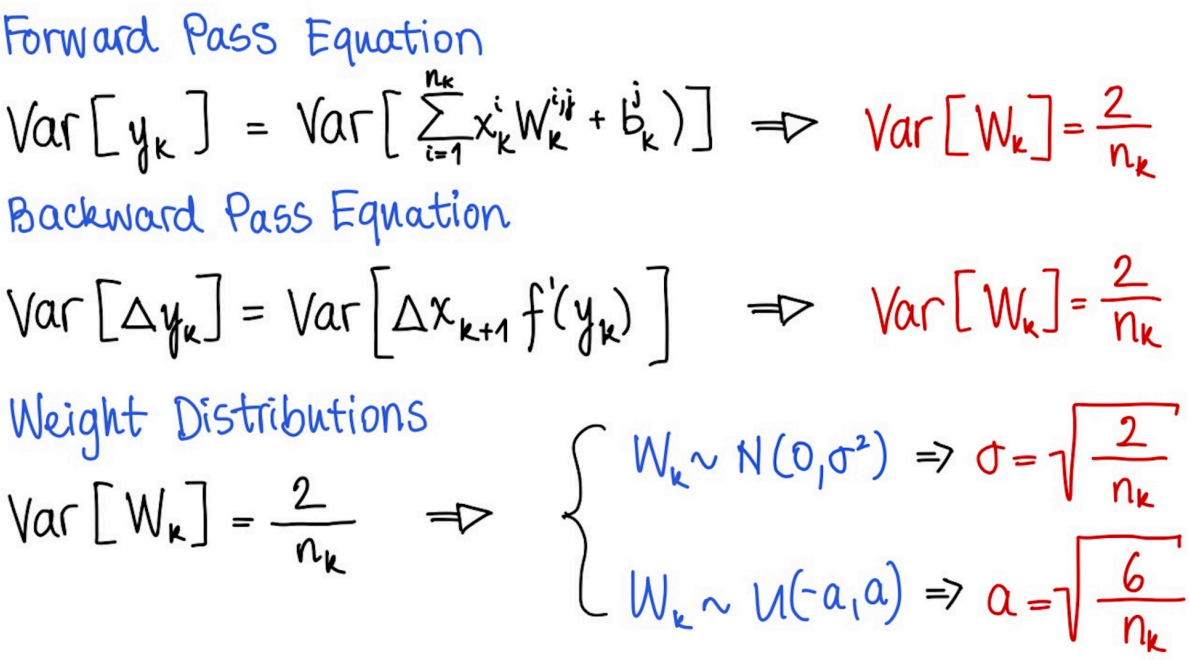
The Context (ReLU):
A ReLU activation function ($f(x) = \max(0, x)$) sets all negative numbers to zero. Effectively, it "kills" half of the neurons in every layer on average. This **halves the variance** of the signal flowing through the network.



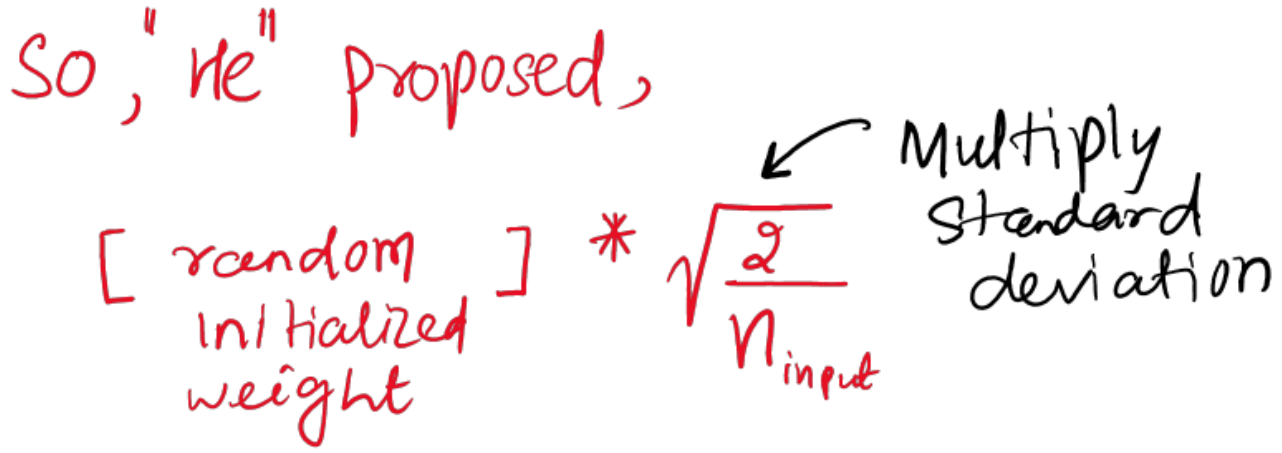
The Solution (He Initialization):

Proposed by Kaiming He, this method initializes weights with a variance of $\frac{2}{n_{in}}$ instead of $\frac{1}{n_{in}}$.

- **Why 2?** To compensate for the fact that ReLU kills half the signal. It doubles the variance of the weights to keep the signal strength constant deeper in the network.
- **Code:** `np.random.randn(...) * np.sqrt(2/input_dim)` as normal distribution else would have multiplied by `np.sqrt(6/input_dim)`



credit: Math proof towards data science



Q4. What if i had to implement different activation function ? What would be my initialization strategy ?

If you change the activation function, you change how the signal variance flows. You must change the initialization strategy.

- **For Sigmoid or Tanh:** Use **Xavier (Glorot) Initialization**.
  - These functions are linear-ish near zero but do not "kill" half the inputs like ReLU.
  - **Formula:** `np.random.randn(...) * np.sqrt(1/input_dim)` (or sometimes $\sqrt{2/(n_{in} + n_{out})}$).
- **For SELU:** Use **LeCun Normal Initialization**.

> **General Rule:** The goal of initialization is always to keep the variance of the input equal to the variance of the output so the signal flows cleanly.

Homework:

#homework

Understand `Xavier` and `LeCun` implementation with `Sigmoid` and `SELU` in code

Q5. Let's talk more about weights, bias, input_cache and returning value

- `self.W` : The learnable matrix. Shape: `(Input_Features, Output_Features)` .
- `self.b` : The learnable bias vector. Shape: `(1, Output_Features)` .
- `self.input_cache` :
  - **Why do we save this?** Look at the gradient calculation for W: $\frac{\partial L}{\partial W} = x^T \cdot \text{grad\_output}$.
  - To calculate the gradient update (`dW`) during the **Backward** pass, we need the original input `x` from the **Forward** pass. Since `backward` happens later, we must "cache" (save) `x` in memory.
- `return grad_input` **(in backward):**

- This is crucial. The current layer has calculated its own gradients (dW, db), but the layer *before* it also needs to know the gradients to update itself.
- We return the gradient with respect to the input ($\frac{\partial L}{\partial x}$) to pass the "error signal" down to the previous layer.

Q6. What is forward and backward doing here ?

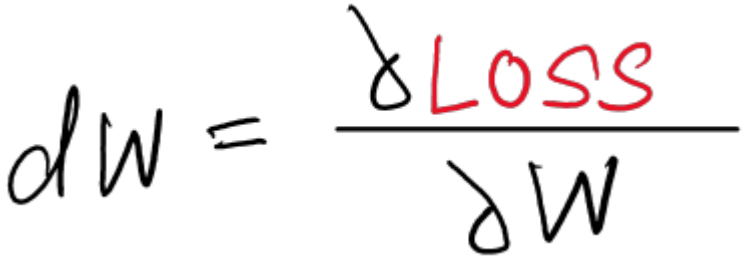
Forward:

```
return np.dot(x, self.W) + self.b
```

Simply calculates $Y = XW + B$. It saves `x` into `self.input_cache` because we need it later.

Backward:
The function receives `grad_output` (The gradient coming from the layer ahead / usually the loss function). It has three jobs:

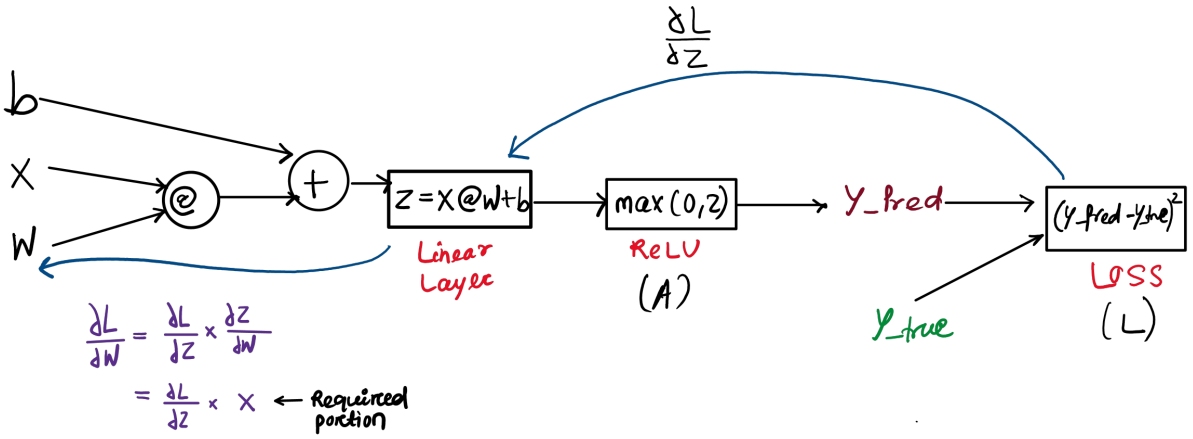
1. **Calculate Weight Gradient (`dW`):**
   - `self.dW = np.dot(self.input_cache.T, grad_output)`
   - This tells us: "How much should we change the Weights to decrease error?"



   - What does this actually mean?

     1. **"The Loss"**: This is the final error calculated at the very end of the network (e.g., "How wrong was the prediction?").
     2. **"With respect to W"**: This asks: *"If I nudge this specific weight value up by a tiny amount, how much will the Final Loss change?"*
        Okay so you will have no doubt once you see the computation graph.

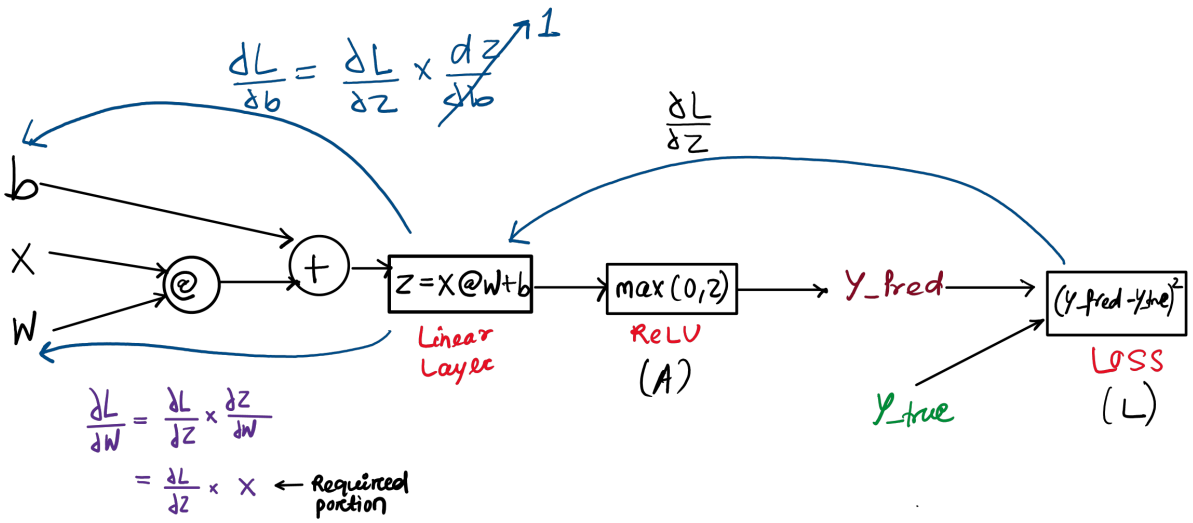
     Forward Pass ($\rightarrow$)
     Backward Pass ($\leftarrow$)



that's why we directly wrote

```
self.dW = np.dot(self.input_cache.T, grad_output)
```

2. **Calculate Bias Gradient (`db`):**
   - `self.db = np.sum(grad_output, axis=0, ...)`
   - This tells us: "How much should we change the Bias to decrease error?"
   - Same goes for `self.db`



Here $\frac{\partial z}{\partial b}$ is 1 as if we increase `b` by 1 then `z` increase by 1 so, the derivative is the rate of change. implies the derivative here will be 1

3. **Pass the Gradient Back (`return ...`):**
   - `return np.dot(grad_output, self.W.T)`
   - This calculates the gradient w.r.t the **Input**.
   - This tells the *previous* layer: "Here is the error that *you* contributed to."
     Here the same goes for derivative of loss w.r.t the input.

Homework:

#homework

complete the derivate of x w.r.t L part with computation graph

Q7. What would have written in actual torch.nn Linear class for the same logic ?

In PyTorch, you almost **never write the backward pass manually**. PyTorch uses a system called **Autograd**. You define the forward pass, and it remembers the operations to build the backward pass automatically.

Here is what the Source Code logic looks like in PyTorch (simplified):

```
import torch
import torch.nn as nn
import math

class PyTorchLinear(nn.Module):
    def __init__(self, in_features, out_features):
        super().__init__()
        # PyTorch wraps weights in a Parameter so it tracks gradients
automatically
        self.weight = nn.Parameter(torch.Tensor(out_features, in_features))
        self.bias = nn.Parameter(torch.Tensor(out_features))
        self.reset_parameters()

    def reset_parameters(self):
        # PyTorch default init (Kaiming Uniform usually)
        nn.init.kaiming_uniform_(self.weight, a=math.sqrt(5))
        if self.bias is not None:
            fan_in, _ = nn.init._calculate_fan_in_and_fan_out(self.weight)
            bound = 1 / math.sqrt(fan_in)
            nn.init.uniform_(self.bias, -bound, bound)

    def forward(self, input):
        # The actual math happens here.
        # F.linear performs (input @ weight.T) + bias
        return torch.nn.functional.linear(input, self.weight, self.bias)

    # NO def backward() needed!
    # PyTorch's Autograd engine handles dW, db, and passing gradients back.
```

Crazy right ?

We will see in PyTorch section. How it is being done.

Next is ReLU Block

```
class ReLU(Layer):
  def __init__(self):
```

```
    self.mask = None

  def forward(self, input):
    self.mask = input > 0
    return input * self.mask

  def backward(self, grad_output):
    return grad_output * self.mask
```

Concept of ReLU

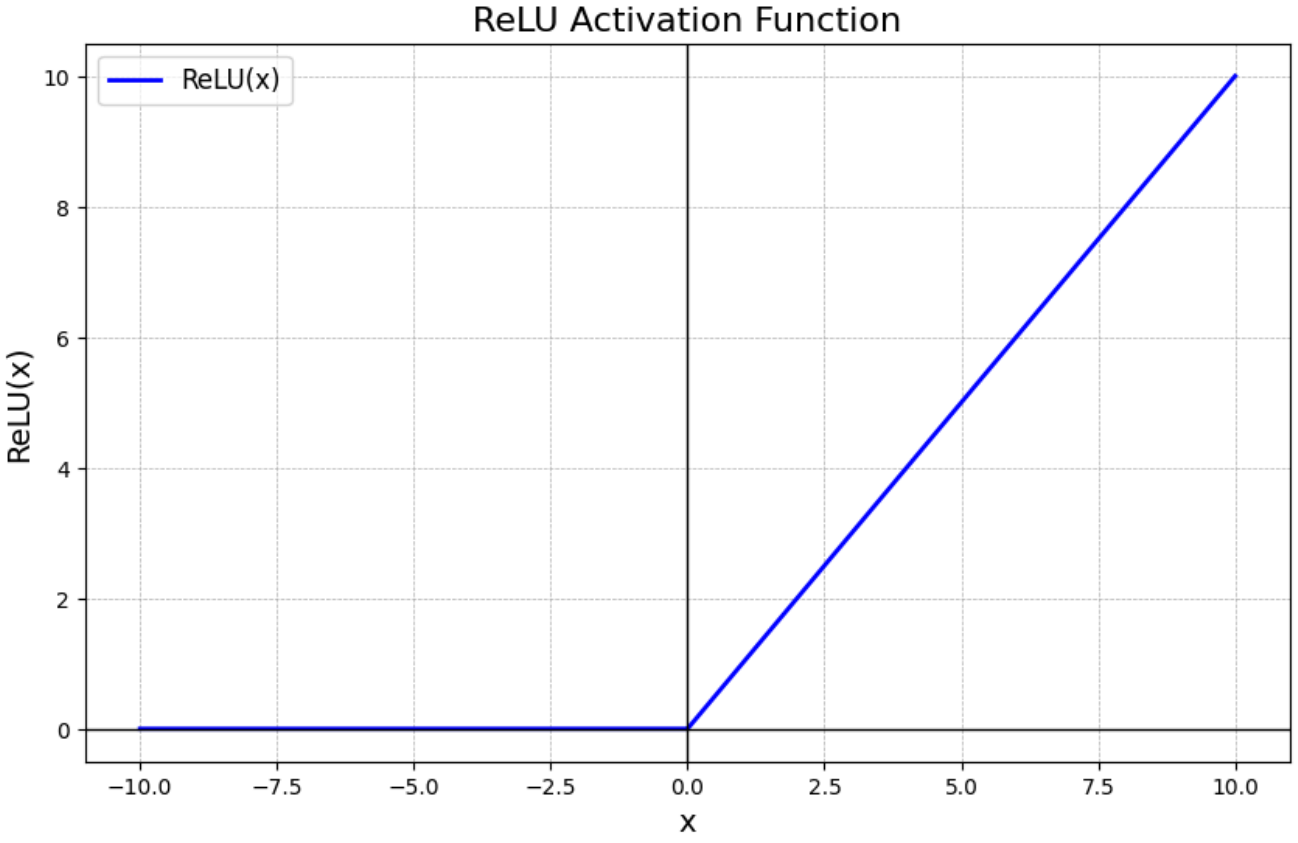
The code implements the **Rectified Linear Unit** function:

$$f(x) = \max(0, x)$$

In the line `return input * self.mask`:

1. **If** $x > 0$: The mask is `True` (which acts like `1`). The result is $x \times 1 = x$.
2. **If** $x \leq 0$: The mask is `False` (which acts like `0`). The result is $x \times 0 = 0$.

This introduces **Non-Linearity** into the network. Without layers like this, a deep neural network would mathematically collapse into a single Linear Regression model, no matter how many layers you stack.



source: GeekforGeeks

How does Python/NumPy handle the multiplication `input * self.mask`?

Answer:
This relies on **implicit typecasting**.

- `input` is a Float array.
- `self.mask` is a Boolean array.

When you multiply a Number by a Boolean in Python/NumPy:

- `True` is treated as **1.0**
- `False` is treated as **0.0**

So, `input * self.mask` is effectively a computationally efficient way to write "Keep the value if True, zero it out if False."

Concept of Mask

Answer:

- **What it is:** `self.mask` is a NumPy boolean array of the same shape as the `input`. It contains `True` where the input pixel/neuron was positive, and `False` where it was negative or zero.
- **Why save it:** We save it because of the **Backward Pass**.
  When calculating gradients later, we need to know **which neurons were active** during the forward pass. If a neuron was "off" (negative input) during the forward pass, it does not contribute to the error, and its gradient should be killed (set to 0). We cache this state so `backward` can use it.
- Here cache is just storing the value in a variable (like here the variable is mask)

it's gradient (it's backward pass)

Q4. Explain the logic in `backward`: `grad_output * self.mask`.

Answer:
given $y = f(x) = max(0, x)$ here,
This applies the **Chain Rule**.
We want to find $\frac{\partial L}{\partial x}$ (the gradient w.r.t input).

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot \frac{dy}{dx}$$

1. $\frac{\partial L}{\partial y}$: This is `grad_output` (passed in from the next layer).
2. $\frac{dy}{dx}$: This is the local derivative of the ReLU function.
   - Derivative of x (when $x > 0$) is **1**.

- Derivative of 0 (when $x < 0$) is **0**.

So, mathematically, the local derivative IS the mask!

- If mask is True (1): Gradient passes through unchanged ($grad \times 1$).
- If mask is False (0): Gradient is killed ($grad \times 0$).

Home work

#homework

Apply other Activation function in the code file. (don't forget to change the respective recommended initialization). we already discussed the outcome of changing Activation function. But you have to implement that.
my suggestions:

1. Sigmoid / tanh
2. Softmax
3. Leaky ReLU
4. GELU
5. SiLU (Swish)
   and see how it boost performance.
   we will cover later anyway #implementation

Loss Function

```
class SoftmaxCrossEntropy:
  def forward(self, logits, y_true):
    # Numerical Stability: Shift values so max is 0
    shift_logits = logits - np.max(logits, axis=1, keepdims=True)
    self.probs = np.exp(shift_logits) / np.sum(np.exp(shift_logits), axis=1,
keepdims=True)

    # Calculate Loss ( Negative Log Likelihood )
    m = y_true.shape[0]
    # We use array indexing for efficient extraction of correct log-probs
    log_likelihood = -np.log(self.probs[range(m), np.argmax(y_true,
axis=1)])
    loss = np.sum(log_likelihood) / m
    return loss

  def backward(self, y_true):
    # The magic simplified derivatives: (Probabilities - labels)/BatchSize
```

```
        m = y_true.shape[0]
        return (self.probs - y_true) / m
```

Q1. What are `logits` and why do we shift them (`logits - max`)?

Code: `shift_logits = logits - np.max(logits, axis=1, keepdims=True)`

- **What are Logits?**
  Logits are the raw, unbounded scores coming out of the last Linear layer (before any activation). They can be anything: -1000, 50, 0.01, etc. (basically the `y_pred`)
- **The Problem (Numerical Instability):**
  Softmax uses the exponential function: e^x.
  If a logit is a large number (e.g., 1000), `np.exp(1000)` results in `inf` (Infinity) because computers cannot store numbers that large. This causes the code to crash or return `NaN`.
- **The Solution (The Max Trick):**
  Mathematically, Softmax is invariant to shifts.

  $$\frac{e^x}{e^x + e^y} = \frac{e^{x-C}}{e^{x-C} + e^{y-C}}$$

  By subtracting the maximum value from every score, the largest number becomes **0**, and the rest become negative.
  - $e^0 = 1$
  - $e^{-huge} = \text{very small number}$
    This guarantees we never overflow, yet the resulting probabilities remain mathematically identical.
    Crazy !! Huuh

Q2. What is `self.probs` calculating?

Code: `self.probs = np.exp(shift_logits) / np.sum(...)`

This is the standard **Softmax** formula:

$$P(class_i) = \frac{e^{score_i}}{\sum_j e^{score_j}}$$

It transforms raw scores into a **Probability Distribution**:

1. All values are positive (because e^x is always positive).
2. All values sum to 1.0.
   Why exponential ?
   to make values positive

Q3. Explain the complex indexing: `self.probs[range(m), np.argmax(y_true, ...)]`

Code: `log_likelihood = -np.log(self.probs[range(m), np.argmax(y_true, axis=1)])`

- **The Goal:** Cross Entropy Loss only cares about the prediction confidence for the **correct class**. If the image is a "Dog", we only care what probability the model assigned to "Dog". We ignore the probabilities for "Cat" or "Car".
- `np.argmax(y_true, axis=1)`:
  Since `y_true` is One-Hot encoded (e.g., `[0, 0, 1]`), this converts it back to an integer index (e.g., `2`). This gives us the column index of the correct class.
- `range(m)`:
  This generates row indices `[0, 1, 2, ... batch_size]`.
- **Combined:**
  This line acts like a "Zipper". It goes to Row 0 and picks the correct column; goes to Row 1 and picks the correct column. It extracts exactly one probability per image—the one corresponding to the ground truth label.
  Let's talk more about it

Cross entropy loss

$$L = -\sum_{k=1}^{K} y_k \log(p_k)$$

credit: Chris Hughes Medium article

Where:

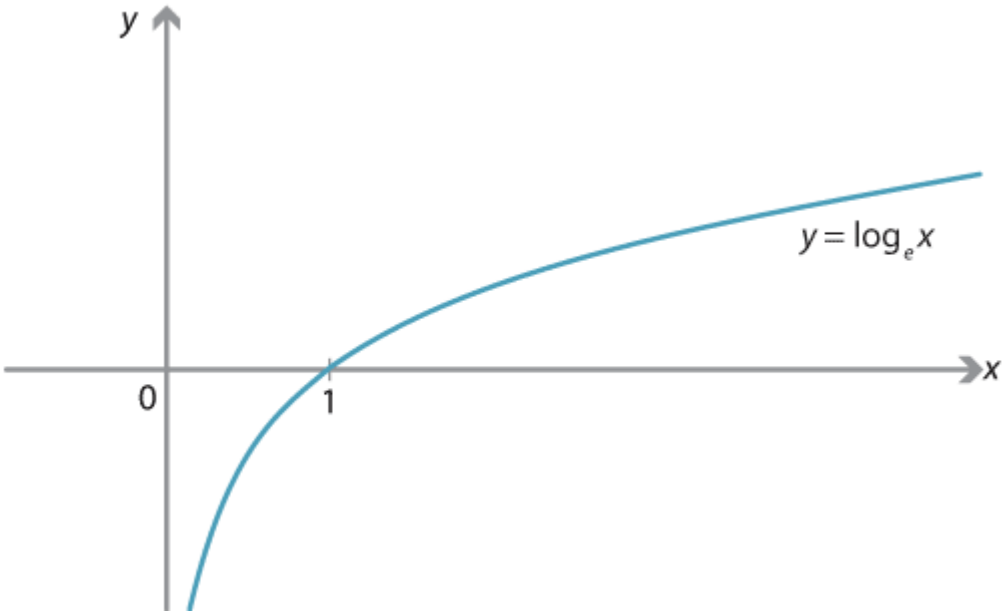
- y_k is the true probability of class k, typically represented as 1 for the correct class and 0 for all other classes.
- p_k is the predicted probability of class k

cross entropy loss is more sensitive to changes for low-confidence predictions on the correct class. This encourages the model to quickly resolve uncertainty by increasing confidence in correct predictions, while heavily penalizing confident mistakes.

The key intuition is:

- The more confident the model is in predicting the correct outcome, the lower the loss.

- The more confident the model is in predicting the wrong outcome, the higher the loss.
  why are we saying this ?
  because
  y_k is a class (a constant)
  p_k is the probability of input being belong to class `k`. if more confident. more probability.
  $log(p_k)$ will be more as graph of log is



  Credit: Link

so multiplying with constant and and sum of these for all k class and negative of whole will be as min as possible. So the loss will be minimum.

and

Why is this behavior useful?

This behavior is crucial for **Backpropagation**.

1. **When the model is Confident & Wrong:** The loss is huge $\rightarrow$ The Gradient is huge $\rightarrow$ The weights get updated **aggressively**. It forces the model to "change its mind" immediately.
2. **When the model is Confident & Correct:** The loss is tiny $\rightarrow$ The Gradient is tiny $\rightarrow$ The weights don't change much. The model stabilizes.

Let's go back to the line
now we know what np.argmax(y_true, axis=1) (think of what would happen if axis = 0) and range(m) so it's now just `self.prob[[0,1,..., m-1], [ each number between 1 to K class for m-1 data ]]` and this is nothing but the zipper analogy
where `self.prob[[index value of range(m), same index of argmax one which return 1 to k ]]`
example let's say range(3) = `[0, 1, 2]`
and np.argmax(y_true, axis = 1) = `[1, 7, 9]` (let's say there are 3 data and 10 classes)

so the zipper would give `[0,1], [1, 7], [2, 9]` and these are the 2d indexing in self.prop which would give a single value for which we calculate -np.log(values from self.prob).

Q4. Why `-np.log` (Negative Log)?

- **The Math:** Let's say We want the probability of the correct class (p) to be **1**.
  - $\log(1) = 0$ (Perfect, 0 Loss)
  - $\log(0.1) = -2.3$ (Bad)
  - $\log(0.01) = -4.6$ (Terrible)
- **The Negative Sign:**
  Since logs of probabilities (0 to 1) are negative, we flip the sign to make the Loss positive. so now We want to **minimize** this positive number. else we would have wanted to maximize the negative number if the -ve sign in np.log ommited.

Q5. Why divide by `m` in `forward` and `backward`?

Code: `loss = ... / m` and `return ... / m`

- **Normalization:** `m` is the Batch Size (e.g., 64).
- If we didn't divide by `m`, the Loss would grow huge if we increased the batch size.
- We want the Loss to represent the **Average Error per Image**, not the total error of the batch. This ensures that our Learning Rate (`lr`) works effectively regardless of whether we train with batch size 32 or 1024.

Q6. Why is `backward` so simple? (`probs - y_true`)

Code: `return (self.probs - y_true) / m`

This is a mathematical miracle known as the **Softmax-with-Loss Derivative**.

If you tried to derive Softmax alone, you would get a messy Jacobian matrix. If you derived Cross Entropy alone, you get $\frac{-1}{p}$.
But when you combine them ($L = -\log(\text{Softmax}(z))$) and take the derivative w.r.t the logits (z), the complicated parts cancel out perfectly:

$$\frac{\partial L}{\partial z} = \text{Predicted\_Prob} - \text{Actual\_Label}$$

This implies:

- If Prediction = 1.0 and Truth = 1.0, Gradient = 0 (Stop learning).
- If Prediction = 0.2 and Truth = 1.0, Gradient = -0.8 (Push logits up strongly).

So forward look like this

logits ← y_Pred

(for stability) Shift_logits ← y_Pred - max(y_Pred)

$$Prob \leftarrow \frac{e^{shift\_logits}}{\sum e^{shift\_logits}}$$ ← vector ← Constant

≃ Prob of each class for each Inputs

m ← no. of Examples

log_liklihood = − log(Prob) ↙ vector

$$loss = \frac{\sum log\_liklihood}{m}$$ ← Average loss

which we converted to code

For backward, we have to calculate derivative of loss w.r.t to logits
we know

$$\frac{d}{dx} \log(x) = \frac{1}{x}$$

so,

$$\frac{d}{dx}(-\log(Prob)) = \frac{-1}{Prob}$$

Now the loss function look like this,

$$loss = \frac{\sum log\_liklihood}{m}$$

and

$$\text{log\_likelihood} = -\log(\text{Prob})$$

derivative w.r.t logits

$$\frac{\partial L}{\partial \text{logits}} = \frac{\partial L}{\partial \text{Prob}} \times \frac{\partial \text{Prob}}{\partial \text{logits}}$$

$$-\frac{1}{\text{Prob}} \times (\dots)$$

before final answer we have to know

$$\log(\text{Probs}) = \log\left(\frac{e^{\text{logits}}}{\sum e^{\text{logits}}}\right)$$

$$= \log(e^{\text{logits}}) - \log(\sum e^{\text{logits}})$$

$$= \text{logits} - \log(\sum e^{\text{logits}})$$

and

$$\frac{d \log(\text{Probs})}{d \text{ logits}} = \frac{d \text{ logit}}{d \text{ logits}} - \frac{\partial \log(\sum e^{\text{logits}})}{\partial \text{ logits}}$$

first one

$$\frac{d \text{ logits}}{d \text{ logits}} = \begin{cases} 0 & \text{if not Same} \\ 1 & \text{if Same} \end{cases}$$

second one

$$\frac{\partial \log(\Sigma e^{logits})}{\partial\, logits} = \frac{1}{\Sigma e^{logits}} \times \frac{\partial(\Sigma e^{logits})}{\partial\, logits}$$

$$= \frac{1}{\Sigma e^{logits}} \times \Sigma \frac{\partial e^{logits}}{\partial\, logits}$$

$$= \frac{1}{\Sigma e^{logits}} \times \Sigma e^{logits} \frac{\partial logits}{\partial logis}$$

(we have seen this before)

$$let \quad \frac{\partial logits}{\partial\, logits} = \delta_{jk}$$

the j is for numerator index and k for denominator index

now continuing the derivative

$$= \frac{1}{\Sigma e^{logits}} \times \Sigma e^{logits} \delta_{jk}$$

$$= \frac{e^{logits}}{\Sigma e^{logits}}$$

← only correct one survive as multiplied by 1 & incorrect don't as multiplied by 0

$$= Prob$$

Back to our sub main derivative of log liklihood w.r.t to logits

$$\frac{d\ log(Probs)}{d\ logits} = \frac{d\ logits}{d\ logits} - \frac{\partial\ log(\Sigma e^{logits})}{\partial\ logits}$$

$$\downarrow \quad = \delta_{jk} - Prob$$

$$\frac{1}{Prob}\ \frac{d\ Probs}{d\ logits} = \delta_{jk} - Prob$$

$$\frac{d\ Probs}{d\ logits} = Prob(\delta_{jk} - Prob)$$

$$\frac{\partial L}{\partial logits} = \frac{\partial L}{\partial\ Prob} \times \frac{\partial\ Prob}{\partial\ logits}$$

$$-\frac{1}{Prob} \times Prob(\delta_{jk} - Prob)$$

$$= \quad Prob - \delta_{jk}$$

$$\uparrow$$

$$y\_true$$

$$= \quad Prob - y\_true$$

That's why only 2 line backward where the derivative is simple by subtracting the probability with y_true

```
return (self.probs - y_true) / m
```

Crazy right !!
if you didn't able understand then here is the source. You can try to understand yourself or can ask me anytime.

Homework

#homework
What are the other loss function we can try ? derive them , do changes in code and see the empire breaking and building
My suggestions

1. MSE (No, why Not ?)
2. Binary cross entropy ?
3. Sparse Categorical Cross Entropy
4. Hinge Loss (SVM Loss)
5. Huber Loss
6. Categorical cross entropy ?
7. Focal Loss

Optimizer

Before the code Let's understand why Adam needed when there was SGD ? What is the concept of SGD ?

Sources

1. Yt Video (my recommendation) Link
2. a funny blog : Link

The blog itself explain each and everything. i am impressed. but we will see the Notes in my next portion note of neural network Link

Back to the Code

```
class Adam:

  def __init__(self, layers, lr = 0.001, beta1=0.9, beta2=0.999, eps = 1e-
8):
    self.layers = [l for l in layers if hasattr(l, 'W')] # Only Layers with
```

```
weights
    self.lr = lr
    self.beta1 = beta1
    self.beta2 = beta2
    self.eps = eps
    self.t = 0

    # Initialize Momentum (m) and RMSProp (v) caches
    self.m_W = {l: 0 for l in self.layers}
    self.v_W = {l: 0 for l in self.layers}
    self.m_b = {l: 0 for l in self.layers}
    self.v_b = {l: 0 for l in self.layers}

  def step(self):
    self.t += 1
    for l in self.layers:
      # Update Weights (W)
      self.m_W[l] = self.beta1 * self.m_W[l] + (1 - self.beta1) * l.dW
      self.v_W[l] = self.beta2 * self.v_W[l] + (1 - self.beta2) * (l.dW **
2)

      # Bias Correction
      m_hat_w = self.m_W[l] / (1 - self.beta1**self.t)
      v_hat_w = self.v_W[l] / (1 - self.beta2**self.t)

      l.W -= self.lr * m_hat_w / (np.sqrt(v_hat_w) + self.eps)

      # Update Biases (b) - (Same logit omitted for brevity, but needed in
prod )
      self.m_b[l] = self.beta1 * self.m_b[l] + (1 - self.beta1) * l.db
      self.v_b[l] = self.beta2 * self.v_b[l] + (1 - self.beta2) * (l.db **
2)

      m_hat_b = self.m_b[l] / (1 - self.beta1 ** self.t)
      v_hat_b = self.v_b[l] / (1 - self.beta2 ** self.t)

      l.b -= self.lr * m_hat_b / (np.sqrt(v_hat_b) + self.eps)
```

Q1. Why do we filter layers with `if hasattr(l, 'W')`?

```
self.layers = [l for l in layers if hasattr(l, 'W')]
```

Answer:
Not every layer in a neural network has "learnable parameters."

- **Linear Layers:** Have Weights (`W`) and Biases (`b`). They need updating.
- **ReLU / Softmax:** These are just mathematical functions. They have no weights to learn. If we didn't filter them out, the optimizer would try to access `relu.W`, which doesn't exist, and the code would crash.

Q2. What are `beta1` (0.9) and `beta2` (0.999)?

Answer:
These are the **Decay Rates** (or "forgetting factors"). They control how much history we keep.

- `beta1` **(0.9):** Controls **Momentum (m)**. It says: "Keep 90% of the previous speed, and add 10% of the current gradient." It makes the movement smooth and heavy.
- `beta2` **(0.999):** Controls **RMSProp/Velocity (v)**. It says: "Keep 99.9% of the historical variance." It ensures the scaling factor changes very slowly over time.

Q3. Why are `self.m_W` and `self.v_W` dictionaries?

```
self.m_W = {l: 0 for l in self.layers}
```

Answer:
Adam needs to remember the history for **every single weight matrix** independently.

- Layer 1 needs its own momentum.
- Layer 2 needs its own momentum.
  By using the Layer object `l` as the **key** in the dictionary, we can easily retrieve the specific memory cache for that specific layer during the loop. We initialize them to `0` because at the start, we haven't moved yet.

Q4. What is the math behind the `m_W` update?

```
self.m_W[l] = self.beta1 * self.m_W[l] + (1 - self.beta1) * l.dW
```

Answer:
This calculates the **First Moment** (Mean) of the gradients.

- **Concept:** This is **Momentum**.
- **Analogy:** Imagine pushing a heavy ball down a hill.
  - `self.m_W[l]` : The current speed of the ball.
  - `l.dW` : The slope of the hill right now.
  - If the slope changes direction temporarily (noise), the heavy ball keeps going in the previous direction (`beta1 * m`).

- This reduces "jitter" and helps the optimizer plow through local noise.

Q5. What is the math behind the `v_W` update, and why square (`** 2`)?

```
self.v_W[l] = self.beta2 * self.v_W[l] + (1 - self.beta2) * (l.dW ** 2)
```

Answer:
This calculates the **Second Moment** (Uncentered Variance) of the gradients.

- **Concept:** This is **Adaptive Learning Rate** (RMSProp).
- **Why Square?** We want to know the **magnitude** (volatility) of the gradients, regardless of sign (positive or negative). Squaring makes everything positive.
- **Goal:**
  - If gradients are huge and changing wildly (v is large), we are on unstable terrain $\rightarrow$ **Slow down** (divide by large number).
  - If gradients are tiny (v is small), the terrain is flat $\rightarrow$ **Speed up** (divide by small number).

Q6. The "Bias Correction" blocks are confusing. Why do we need them?

```
m_hat_w = self.m_W[l] / (1 - self.beta1**self.t)
```

Answer:
This fixes the **"Cold Start" problem**.

- **The Problem:** We initialized `m_W = 0`.
  - In step 1, `m = 0.9 * 0 + 0.1 * grad`. Result: `0.1 * grad`.
  - The momentum is **10x smaller** than it should be! It is "biased towards zero."
- **The Fix:** We divide by `(1 - 0.9^t)`.
  - at $t = 1$: Divide by $1 - 0.9 = 0.1$.
  - Result: `(0.1 * grad) / 0.1` = `1.0 * grad`. **Corrected!**
- As t gets large (e.g., $t = 100$), `0.9^100` becomes tiny, the denominator becomes `~1`, and bias correction effectively turns off because the warm-up period is over.

Q7. Breaking down the Final Update Line

```
l.W -= self.lr * m_hat_w / (np.sqrt(v_hat_w) + self.eps)
```

Answer:
This combines everything into the **Parameter Update**. Let's split it:

1. `l.W -=`: Gradient Descent. We move in the opposite direction of the gradient.

2. `self.lr`: The base step size (e.g., 0.001).
3. `m_hat_w`: "Go this way." (The smoothed direction).
4. `/ (np.sqrt(v_hat_w) ...)`: "Normalize the step size."
    - We take the square root to bring `v` (which was squared gradients) back to the same unit/scale as the weights.
    - If `v` is high (steep/volatile), the denominator is big → Step size shrinks.
    - If `v` is low (flat), the denominator is small → Step size grows.
5. `+ self.eps`: Safety. If `v` is 0 (e.g., dead neurons), we would divide by zero and get `NaN`. Adding `1e-8` prevents the crash.

Summary of the Flow

1. **Loop:** Go through every layer.
2. **Accumulate:**
    - Add current gradient to Momentum history (`m`).
    - Add squared current gradient to Variance history (`v`).
3. **Correct:** Fix the fact that we started at 0 (`hat`).
4. **Update:** Move the weights based on Momentum, slowed down by Variance.

Homework

#homework

Implement other optimizers see yourself getting stressed and relieved based on results.
my suggestions

1. Vanila SGD (Stochastic gradient descent)
2. AdamW
3. Lion
4. SGD with Momentum

enough for now we can see more optimizer with time

Cool.. we are close

Not the perfect one, but we will improve our intuition and concepts over time. so keeping the consistency is important.

Accuracy function

```
def get_accuracy(layers, X, Y):
    # Forward pass only
    out = X
    for layer in layers:
        out = layer.forward(out)
```

```
    predictions = np.argmax(out, axis=1)
    true_labels = np.argmax(Y, axis=1)
    return np.mean(predictions == true_labels)
```

Q1. Why do we run the loop `for layer in layers` again? Can't we just use the output from training?

Answer:
We cannot use the output from the training loop because we might be testing on **new data** (like the Test Set).

- **Training:** Updates weights using training data.
- **Evaluation:** Uses the *current* state of the weights to predict on data the network might haven't seen before.

The loop `out = layer.forward(out)` pushes the input data (X) through the entire network (Linear $\rightarrow$ ReLU $\rightarrow$ Linear...) to get the final scores (Logits/Probabilities).

Key Difference: Notice we **do not** call `backward()`. We are just checking answers, not learning.

Q2. What is `np.argmax` and why do we use `axis=1`?

Code: `predictions = np.argmax(out, axis=1)`

Answer:
The network outputs **Probabilities** (or Logits), but humans want **Class Labels** (e.g., "Is it a 5 or a 9?").

- `out`: A matrix of shape `(Batch_Size, 10)`.
  - Example for one image: `[0.05, 0.01, 0.90, 0.04 ...]` (Model is 90% sure it's a "2").
- `np.argmax`: Returns the **Index** of the maximum value.
  - It looks at the array, finds `0.90`, and returns index `2`.
- `axis=1`: This means "Work horizontally across columns."
  - We want the best class **per image** (per row).
  - If we used `axis=0`, we would find the highest probability across the entire batch, which is useless here.

Q3. Why do we run `np.argmax` on `Y` (true_labels) as well?

Code: `true_labels = np.argmax(Y, axis=1)`

Answer:
This is necessary because your labels `Y` are **One-Hot Encoded**.

- **Current Format of Y:** `[0, 0, 1, 0, 0, ...]` (Vector representation of "2").
- **Desired Format:** `2` (Integer representation).

We cannot compare a scalar prediction (e.g., `2`) with a vector (e.g., `[0,0,1...]`). We must convert the One-Hot vector back to a simple integer so we can compare "Apples to Apples."

Note: If our dataset provided labels as simple integers (0-9) originally, this step would not be needed. But our code uses One-Hot.

Q4. How does `np.mean` calculate accuracy?

Code: `np.mean(predictions == true_labels)`

Answer:
This relies on a cool Python/NumPy trick: **Booleans are Integers.**

1. **Comparison (`==`):**
   - `predictions`: `[2, 5, 9]`
   - `true_labels`: `[2, 1, 9]`
   - Result: `[True, False, True]`
2. **Conversion:**
   - Python treats `True` as `1.0`.
   - Python treats `False` as `0.0`.
   - Effective Array: `[1.0, 0.0, 1.0]`
3. **Mean (Average):**
   - Formula: $\frac{\text{Sum of values}}{\text{Count of values}}$
   - Calculation: $\frac{1+0+1}{3} = \frac{2}{3} = 0.666$

So, the mean of the boolean array is exactly equal to the **Accuracy Percentage** (66.6%).

Train

```
def train(X_train, Y_train, X_test, Y_test, epochs=5, batch_size=64):
  print(f"Training on {X_train.shape[0]} samples. Input Dim:
{X_train.shape[1]}")
  # Architecture
  layers = [
      Linear(784, 128),
      ReLU(),
      Linear(128, 10)
  ]

  loss_fn = SoftmaxCrossEntropy()
  optimizer = Adam(layers, lr=0.001) # we will improvement ourselves here in
```

```
next implementation

  num_batches = X_train.shape[0] // batch_size

  for epoch in range(epochs):
    # Shuffle
    perm = np.random.permutation(X_train.shape[0])
    X_shuffled, Y_shuffled = X_train[perm], Y_train[perm]
    epoch_loss = 0

    for i in range(num_batches):
        start = i * batch_size
        x_batch = X_shuffled[start:start+batch_size]
        y_batch = Y_shuffled[start:start+batch_size]

        # Forward
        out = x_batch
        for layer in layers:
            out = layer.forward(out)

        # Loss
        loss = loss_fn.forward(out, y_batch)
        epoch_loss += loss

       # Backward
        grad = loss_fn.backward(y_batch)
        for layer in reversed(layers):
            grad = layer.backward(grad)

        # Update
        optimizer.step()

    # Evaluate at end of epoch
    avg_loss = epoch_loss / num_batches
    train_acc = get_accuracy(layers, X_train[:1000], Y_train[:1000]) #
Estimate on subset
    test_acc = get_accuracy(layers, X_test, Y_test)
    print(f"Epoch {epoch+1}/{epochs} | Loss: {avg_loss:.4f} | Train Acc:
{train_acc:.4f} | Test Acc: {test_acc:.4f}")
  return layers
```

Setup

```
layers = [
    Linear(784, 128),
    ReLU(),
    Linear(128, 10)
]
optimizer = Adam(layers, lr=0.001)
```

Q1. Why are the dimensions 784, 128, and 10?

- **784:** This is the Input Layer. MNIST images are 28×28 pixels. $28 \times 28 = 784$. We flattened the 2D image into a 1D vector.
- **128:** This is the Hidden Layer size. It's a hyperparameter (a design choice). You could choose 64, 256, or 512. It represents "how much capacity" the network has to learn patterns.
- **10:** This is the Output Layer. MNIST has digits 0–9, so we need exactly 10 output scores (probabilities).

Q2. Why do we pass `layers` into the `Adam` optimizer?

The Optimizer needs to know **what** to modify. By passing the list of `layers`, the optimizer scans them, finds the ones that have weights (`self.W`, `self.b`), and registers them so it can update them later using `.step()`.

The Epoch Loop & Shuffling

```
for epoch in range(epochs):
    perm = np.random.permutation(X_train.shape[0])
    X_shuffled, Y_shuffled = X_train[perm], Y_train[perm]
```

Q3. What is an Epoch?

An Epoch is one complete pass through the entire training dataset (seeing all 60,000 images once). If `epochs=5`, the model sees every image 5 times total.

Q4. Why do we shuffle (`np.random.permutation`) at the start of every epoch?

Crucial Concept: If you don't shuffle, the model might learn the **order** of the data rather than the features.

- *Example:* If your data is sorted (all 0s, then all 1s...), the model will overfit to "0s" in the first 100 batches and forget everything else.
- Shuffling ensures every batch is a random mix, which stabilizes the Gradient Descent.

The Mini-Batch Loop

```
num_batches = X_train.shape[0] // batch_size
for i in range(num_batches):
    start = i * batch_size
    x_batch = X_shuffled[start : start+batch_size]
    y_batch = Y_shuffled[start : start+batch_size]
```

Q5. Why do we use `//` (Integer Division)?

We need a whole number of batches. If we have 60,000 images and batch size is 64, $60000/64 = 937.5$. We can't run "half a loop", so `//` truncates it to 937 loops.

Q6. What is slicing doing here (`start : start+batch_size`)?

This extracts a "Mini-Batch."

- **Iteration 0:** Indices `0` to `64`.
- **Iteration 1:** Indices `64` to `128`.
- This allows us to train on small chunks of data. Training on all 60,000 at once is too slow and consumes too much RAM.

The Forward Pass

```
out = x_batch
for layer in layers:
    out = layer.forward(out)
```

Q7. How does this loop work?
This is the "Bucket Brigade."

1. `out` starts as the raw images.
2. **Layer 0 (Linear):** Takes images, outputs raw numbers. `out` becomes those numbers.
3. **Layer 1 (ReLU):** Takes those numbers, sets negatives to 0. `out` becomes the filtered numbers.
4. **Layer 2 (Linear):** Takes filtered numbers, calculates class scores.
   The final `out` holds the predictions.

Loss Calculation

```
loss = loss_fn.forward(out, y_batch)
epoch_loss += loss
```

Q8. Why do we calculate loss here?

The loss function tells us "How wrong were we?" for this specific batch.

- `epoch_loss += loss` : We accumulate the loss just for printing later. It helps us see the average error over the whole epoch.

The Backward Pass (Chain Rule)

```
grad = loss_fn.backward(y_batch)
for layer in reversed(layers):
    grad = layer.backward(grad)
```

Q9. Why do we reverse the layers (`reversed(layers)`)?

This is **Backpropagation**.

- The error is calculated at the **End** (Output).
- We need to carry that error back to the **Start** (Input).
- We must process `Linear(Output)` $\rightarrow$ `ReLU` $\rightarrow$ `Linear(Input)` .

Q10. What is happening to the variable `grad` ?

It acts like a baton in a relay race.

1. `loss_fn` calculates the initial gradient (Error signal).
2. It passes `grad` to the last Linear layer.
3. That layer calculates its `dW` (for update) and returns a *new* `grad` (for the previous layer).
4. This repeats until the input.

The Update

```
optimizer.step()
```

Q11. What happens inside `.step()` ?

The optimizer looks at every layer. It sees that `layer.dW` and `layer.db` were just calculated in the backward pass.
It applies the math:

$$W_{new} = W_{old} - \text{LearningRate} \times \text{AdaptiveGradient}$$

It modifies `layer.W` directly. The model is now slightly smarter than it was 1 millisecond ago.

Evaluation

```
avg_loss = epoch_loss / num_batches
train_acc = get_accuracy(layers, X_train[:1000], Y_train[:1000])
test_acc = get_accuracy(layers, X_test, Y_test)
```

Q12. Why do we divide `epoch_loss` by `num_batches`?

`epoch_loss` is the *sum* of errors. If we have more batches, the sum is naturally bigger. Dividing gives us the **Average Loss**, which is easier to compare across different experiments.

Q13. Why do we use `X_train[:1000]` instead of the full `X_train`?

Speed.
Calculating accuracy requires running the forward pass.

- Running it on 60,000 images takes time.
- Running it on a random subset of 1,000 images gives a "good enough" estimate of how well the model is learning, without slowing down the training loop.

Q14. Why check `test_acc` (Test Accuracy)?

This checks for **Overfitting**.

- If `Train Acc` is 99% but `Test Acc` is 80%, the model is just memorizing the training images (cheating).
- We want both numbers to go up together.

What was my thought process writing this code piece ?

Phase 1: The Setup (Architecture & Strategy)

Thought 1: "What does my model look like?"

- *Requirement:* I need to define the structure.
- *Input:* MNIST images are 28x28 = **784** pixels.
- *Output:* digits 0-9 = **10** classes.
- *Hidden:* I need a hidden layer to learn shapes/curves. Let's pick **128** neurons.
- *Activation:* I need non-linearity. **ReLU** is the standard.
- *Code:* `layers = [Linear(784, 128), ReLU(), Linear(128, 10)]`

Thought 2: "How do I measure success and improve?"

- *Loss:* This is a classification problem. I need **Softmax Cross Entropy**.
- *Optimizer:* SGD is too slow/unstable for a first try. **Adam** is the "just works" option.
- *Code:* `loss_fn = SoftmaxCrossEntropy()`, `optimizer = Adam(...)`

Phase 2: The Data Strategy (Batching)

Thought 3: "I can't process 60,000 images at once."

- *Problem:* Matrix multiplication of (60000, 784) is heavy on RAM and slow.
- *Solution:* Divide and conquer (**Mini-Batch Gradient Descent**).
- *Decision:* Let's use a batch size of 64.
- *Math:* Total iterations = `Total Samples / Batch Size`.
- *Code:* `num_batches = X.shape[0] // batch_size`

Thought 4: "If I feed data in the same order, the model will cheat."

- *Problem:* If the data is sorted (0s, then 1s...), the model learns "Oh, the first 10 minutes are always 0s".
- *Solution:* **Shuffle** the data at the start of every epoch.
- *Code:* `perm = np.random.permutation(...)`, `X_shuffled = X[perm]`

Phase 3: The Learning Loop (Forward -> Loss -> Backward -> Update)

Thought 5: "How do I get a prediction?" (Forward Pass)

- *Logic:* Take the batch (x) and pass it through Layer 1 -> Activation -> Layer 2.
- *Implementation:* I can't hardcode `layer1.forward()`, then `layer2.forward()` because I might change the architecture later.
- *Solution:* Loop through the list of layers.
- *Code:* `for layer in layers: out = layer.forward(out)`

Thought 6: "How wrong was I?" (Loss Calculation)

- *Logic:* Compare prediction (`out`) with truth (`y_batch`).
- *Side Note:* I need to track the total loss to print a readable average later.
- *Code:* `loss = loss_fn.forward(...)`, `epoch_loss += loss`

Thought 7: "Who is responsible for the error?" (Backward Pass)

- *Logic:* This is the Chain Rule.

1. Get the gradient from the Loss function.
2. Pass it *backwards* through the network.
3. Linear Layer 2 needs the gradient from Loss.
4. ReLU needs the gradient from Linear Layer 2.
5. Linear Layer 1 needs the gradient from ReLU.

- *Code:* `grad = loss_fn.backward(...)`, `for layer in reversed(layers): ...`

Thought 8: "Fix the weights." (Update)

- *Logic:* Now that every layer knows its gradient (`dW`), apply the Adam update rule.
- *Code:* `optimizer.step()`

Phase 4: Feedback & Evaluation

Thought 9: "Is it actually learning?"

- *Problem:* Watching the Loss go down (e.g., from 2.3 to 0.5) is good, but I don't know if it's actually guessing digits correctly.
- *Solution:* Calculate **Accuracy**.
- *Performance:* calculating accuracy on all 60,000 training images is slow. I'll just check a subset (first 1000) to be fast.
- *Safety:* I must check **Test Accuracy** to make sure I'm not memorizing (Overfitting).
- *Code:* `train_acc = get_accuracy(...)`, `test_acc = get_accuracy(...)`

Let's see the code in action

```
# 1. Download Data
X_train_raw, y_train_raw, X_test_raw, y_test_raw = load_mnist()

# 2. Preprocess Labels (One Hot)
Y_train = one_hot_encode(y_train_raw)
Y_test = one_hot_encode(y_test_raw)

# 3. Train
model = train(X_train_raw, Y_train, X_test_raw, Y_test, epochs=10,
batch_size=64)
```

```python
# 1. Download Data
X_train_raw, y_train_raw, X_test_raw, y_test_raw = load_mnist()

# 2. Preprocess Labels (One Hot)
Y_train = one_hot_encode(y_train_raw)
Y_test = one_hot_encode(y_test_raw)

# 3. Train
model = train(X_train_raw, Y_train, X_test_raw, Y_test, epochs=10,
batch_size=64)
```

```
Downloading file: t10k-labels-idx1-ubyte.gz
Training on 60000 samples. Input Dim: 784
Epoch 1/10 | Loss: 4.3534 | Train Acc: 0.9215 | Test Acc: 0.9136
Epoch 2/10 | Loss: 0.5685 | Train Acc: 0.9324 | Test Acc: 0.9170
Epoch 3/10 | Loss: 0.3147 | Train Acc: 0.9426 | Test Acc: 0.9254
Epoch 4/10 | Loss: 0.2499 | Train Acc: 0.9480 | Test Acc: 0.9330
Epoch 5/10 | Loss: 0.2068 | Train Acc: 0.9520 | Test Acc: 0.9374
Epoch 6/10 | Loss: 0.1970 | Train Acc: 0.9512 | Test Acc: 0.9404
Epoch 7/10 | Loss: 0.1789 | Train Acc: 0.9579 | Test Acc: 0.9444
Epoch 8/10 | Loss: 0.1752 | Train Acc: 0.9549 | Test Acc: 0.9429
Epoch 9/10 | Loss: 0.1574 | Train Acc: 0.9667 | Test Acc: 0.9543
Epoch 10/10 | Loss: 0.1576 | Train Acc: 0.9638 | Test Acc: 0.9486
```

10 epochs -> 96%+ accuracy
few questions

1. increasing epochs, will increase the accuracy ?
2. How would you explain the test accuracy fluctuation ?
3. do batch size matter ? How it would affect the performance
4. How can you make this code worst performing ? (interesting as this is exactly you should not do right ?)

Improvements we can do

1. adding training boolean, parameters method for optimizing parameters rather than layers in train function and layers class
2. Sequential container instead of List in train function
3. logsumexp for numerical stability
4. adamw implementation

Submission-1

```python
import numpy as np

class NeuralNetwork:
    '''
    Build a neural network from scratch using only NumPy.
    Required architecture: 784 â†' 128 (ReLU) â†' 10 (Softmax)
    '''
    def __init__(self, input_size=784, hidden_size=128, output_size=10,
lr=0.01):
        '''
        Initialize network parameters.
```

```
        Use small random initialization (e.g., Xavier/He initialization).
        '''
        # Initialize weights using He Initialization (good for ReLU)
        # Formula: randn * sqrt(2 / input_nodes)
        self.lr = lr
        self.W1 = np.random.randn(input_size, hidden_size) * np.sqrt(2.0 /
input_size)
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * np.sqrt(2.0 /
hidden_size)
        self.b2 = np.zeros((1, output_size))

        # Cache for backward pass
        self.cache = {}

    def forward(self, X):
        '''
        Forward pass through the network.
        Args:
            X: Input batch, shape (N, 784)
        Returns:
            probs: Class probabilities, shape (N, 10)
        Must cache intermediate values for backward pass!
        '''
        # Layer 1: Linear
        Z1 = np.dot(X, self.W1) + self.b1

        # Layer 1: Activation (ReLU)
        A1 = np.maximum(0, Z1)

        # Layer 2: Linear
        Z2 = np.dot(A1, self.W2) + self.b2

        # Layer 2: Activation (Softmax)
        # Numerical stability: shift values so max is 0
        shift_logits = Z2 - np.max(Z2, axis=1, keepdims=True)
        exp_scores = np.exp(shift_logits)
        probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True)

        # Cache values needed for backward pass
        self.cache = {'Z1': Z1, 'A1': A1, 'Z2': Z2}
        return probs
```

```python
    def backward(self, X, y, probs):
        '''
        Backward pass - compute gradients for all parameters.
        Args:
            X: Input batch, shape (N, 784)
            y: True labels, shape (N,)
            probs: Predicted probabilities from forward pass, shape (N, 10)
        Returns:
            loss: Scalar cross-entropy loss
        Must update self.W1, self.b1, self.W2, self.b2 using computed
gradients!
        '''
        m = X.shape[0] # Batch size
        # 1. Compute Loss (Cross Entropy)
        # Extract the probability assigned to the correct class
        correct_logprobs = -np.log(probs[range(m), y] + 1e-15) # + epsilon
for safety
        loss = np.sum(correct_logprobs) / m

        # 2. Compute gradient of loss w.r.t. softmax output (dZ2)
        # Formula: (Predicted_Probs - One_Hot_Labels) / Batch_Size
        dZ2 = probs.copy()
        dZ2[range(m), y] -= 1
        dZ2 /= m

        # 3. Backprop through linear layer 2
        # dW2 = A1.T @ dZ2
        dW2 = np.dot(self.cache['A1'].T, dZ2)
        db2 = np.sum(dZ2, axis=0, keepdims=True)

        # 4. Backprop through ReLU
        # dA1 = dZ2 @ W2.T
        dA1 = np.dot(dZ2, self.W2.T)
        # ReLU derivative: 1 if Z1 > 0 else 0
        dZ1 = dA1 * (self.cache['Z1'] > 0)

        # 5. Backprop through linear layer 1
        dW1 = np.dot(X.T, dZ1)
        db1 = np.sum(dZ1, axis=0, keepdims=True)

        # 6. Update all parameters using gradients (Standard SGD)
        self.W1 -= self.lr * dW1
        self.b1 -= self.lr * db1
```

```
        self.W2 -= self.lr * dW2
        self.b2 -= self.lr * db2
        return loss

    def train_step(self, X, y):
        '''
        Complete training step: forward + backward + update.
        Args:
            X: Input batch, shape (N, 784)
            y: True labels, shape (N,)
        Returns:
            loss: Scalar loss value
        '''
        probs = self.forward(X)
        loss = self.backward(X, y, probs)
        return loss

    def predict(self, X):
        '''
        Predict class labels.
        Args:
            X: Input batch, shape (N, 784)
        Returns:
            predictions: Predicted class labels, shape (N,)
        '''
        probs = self.forward(X)
        return np.argmax(probs, axis=1)
```

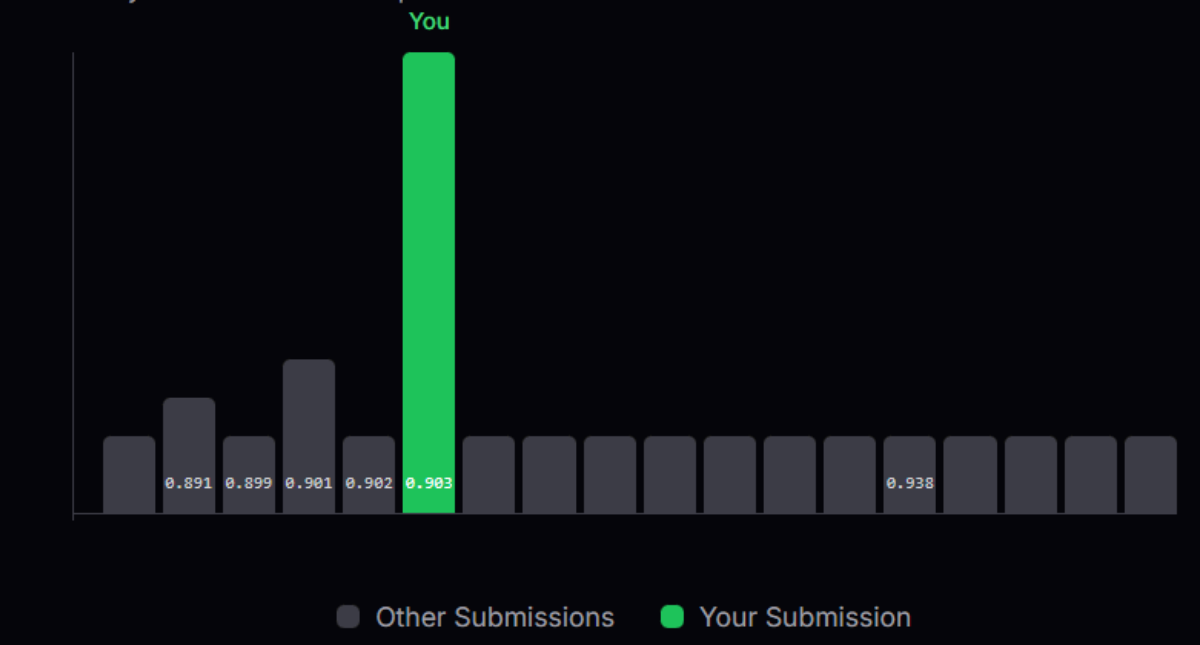
The result is 90.3% accuracy still less than average 91.2% so, next we will chase this number.

Submission Results

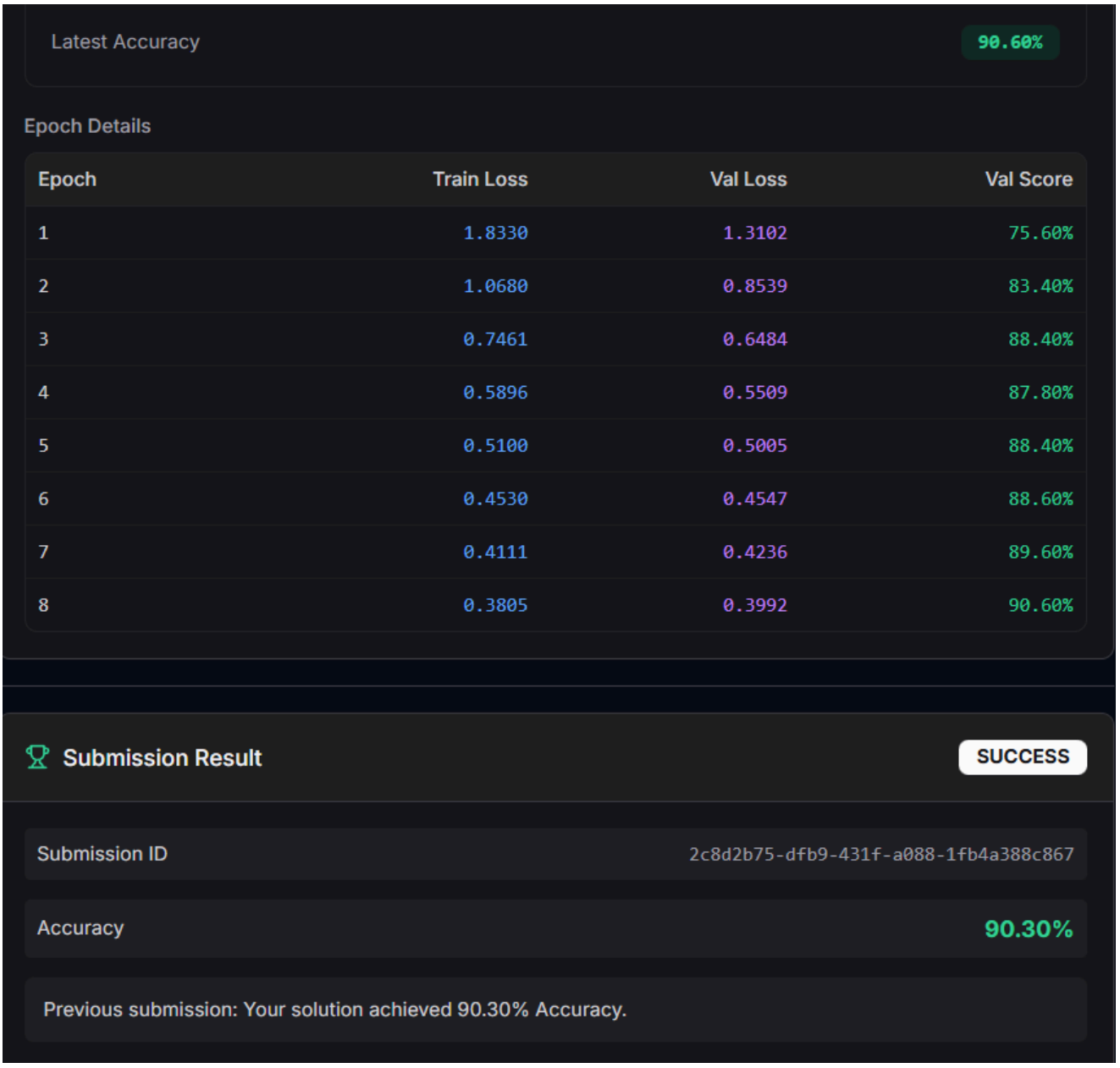
Back to Editor

Score Distribution

See how your submission compares to others



YOUR RANK	YOUR SCORE	TOTAL ATTEMPTS
61%	0.903	38
Above Average	Avg: 0.912	Top: 0.949



Homework

Why this will give a bad result ?
even if not the perfect answer, i want to hear your intuition and hypothesis, that this or that can be the problem.

```
import numpy as np

class NeuralNetwork:
    '''
    Build a neural network from scratch using only NumPy.
    Required architecture: 784 → 128 (ReLU) → 10 (Softmax)
    '''
    def __init__(self, input_size=784, hidden_size=128, output_size=10,
lr=0.01):
        '''
```

```
        Initialize network parameters.
        Use small random initialization (e.g., Xavier/He initialization).
        '''
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.lr = lr

        # Initialization of weights and biases
        self.W0 = np.random.randn(self.input_size, self.hidden_size) /
np.sqrt(2/self.input_size)
        self.b0 = np.zeros((1, self.hidden_size))
        self.W1 = np.random.randn(self.hidden_size, self.output_size) /
np.sqrt(2/self.hidden_size)
        self.b1 = np.zeros((1, self.output_size))

    def forward(self, X):
        '''
        Forward pass through the network.
        Args:
            X: Input batch, shape (N, 784)
        Returns:
            probs: Class probabilities, shape (N, 10)
        Must cache intermediate values for backward pass!
        '''
        # TODO: Implement forward pass
        # Layer 1: X @ W1 + b1, then ReLU
        # Layer 2: hidden @ W2 + b2, then Softmax
        # Cache to store 'activations' for the backward pass
        self.cache = {'A0': X}

        # Layer 1 (ReLU)
        Z1 = X @ self.W0 + self.b0
        self.cache['A1'] = np.maximum(0, Z1) # ReLU

        # Layer 2 (Softmax)
        Z2 = self.cache['A1'] @ self.W1 + self.b1

        # Numerical Stability Trick: subtract max before exp
        exps = np.exp(Z2 - np.max(Z2, axis=1, keepdims=True))
        self.cache['A2'] = exps / np.sum(exps, axis=1, keepdims=True)
        return self.cache['A2']
```

```python
    def backward(self, X, Y, probs):
        '''
        Backward pass - compute gradients for all parameters.
        Args:
            X: Input batch, shape (N, 784)
            y: True labels, shape (N,)
            probs: Predicted probabilities from forward pass, shape (N, 10)
        Returns:
            loss: Scalar cross-entropy loss
        Must update self.W1, self.b1, self.W2, self.b2 using computed
gradients!
        '''
        # TODO: Implement backward pass
        # 1. Compute loss
        # 2. Compute gradient of loss w.r.t. softmax output
        # 3. Backprop through linear layer 2
        # 4. Backprop through ReLU
        # 5. Backprop through linear layer 1
        # 6. Update all parameters using gradients
        # THE PRO MOVE: Combined Softmax + CrossEntropy Derivative
        # dL/dZ2 simply equals (Pred - Y). No complex Jacobian needed.
        m = Y.shape[0]
        # convert labels to one-hot
        Y_onehot = np.zeros_like(probs)
        Y_onehot[np.arange(m), Y] = 1

        dZ2 = self.cache['A2'] - Y_onehot

        # Gradients for Layer 2
        dW1 = (self.cache['A1'].T @ dZ2) / m
        db1 = np.sum(dZ2, axis=0, keepdims=True) / m

        # Backprop to Layer 1 (Derivative of ReLU is just 1 where x>0)
        dZ1 = (dZ2 @ self.W1.T) * (self.cache['A1'] > 0)

        # Gradients for Layer 1
        dW0 = (self.cache['A0'].T @ dZ1) / m
        db0 = np.sum(dZ1, axis=0, keepdims=True) / m

        # SGD Update
        self.W1 -= self.lr * dW1
        self.b1 -= self.lr * db1
        self.W0 -= self.lr * dW0
```

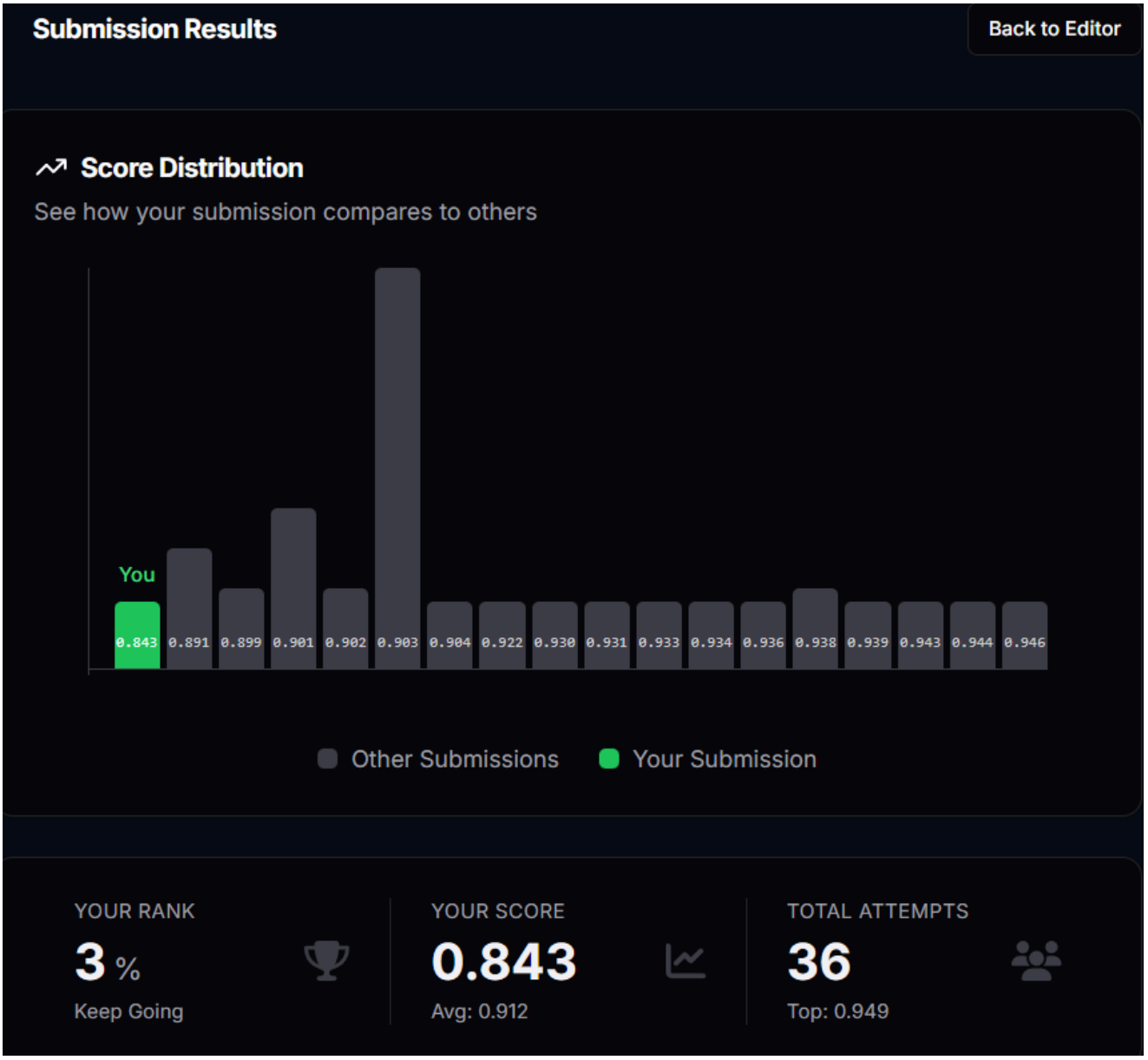
```
        self.b0 -= self.lr * db0

        # loss
        loss = -np.log(probs[np.arange(m), Y] + 1e-9).mean()
        return loss

    def train_step(self, X, y):
        '''
        Complete training step: forward + backward + update.
        Args:
            X: Input batch, shape (N, 784)
            y: True labels, shape (N,)
        Returns:
            loss: Scalar loss value
        '''
        probs = self.forward(X)
        loss = self.backward(X, y, probs)
        return loss

    def predict(self, X):
        '''
        Predict class labels.
        Args:
            X: Input batch, shape (N, 784)
        Returns:
            predictions: Predicted class labels, shape (N,)
        '''
        probs = self.forward(X)
        return np.argmax(probs, axis=1)
```

I got this



an accuracy that is not satisfying the criteria of 0.85 accuracy.

So, What went wrong ?

> Don't skip this

Note:

1. Do all the Homework, and do all steps with your own hand.
2. Implement the logic by your interpretation
3. If you got any doubt Feel free to ask (even if it's silly)
4. As a accountability, Send the answers of HomeWorks (need not be professional or beautiful notes , it can be in paper as well) to me.
5. **Pro Tip:** Try to comment the code in English or Hindi what ever you like yourself in above code or in Colab file given below. This will improve your interpretation. (I forgot

doing so, but good for you though)

Have Fun :)