Deep Learning assignment #1
Akshat Mishra | Spring 2025 | netid: am15111

Question 1.
Expressivity of neural networks. Recall that the functional form for a single neuron is given by $y = \sigma(\langle w, x \rangle + b)$, where x is the input and y is the output. In this exercise, assume that x and y are 1- dimensional (i.e., they are both just real-valued scalars) and σ is the step activation: $\sigma(u) = 1$ for $u > 0$ and 0 otherwise. We will use multiple layers of such neurons to approximate pretty much any function f. There is no learning/training required for this problem; you should be able to guess/derive the weights and biases of the networks by hand.

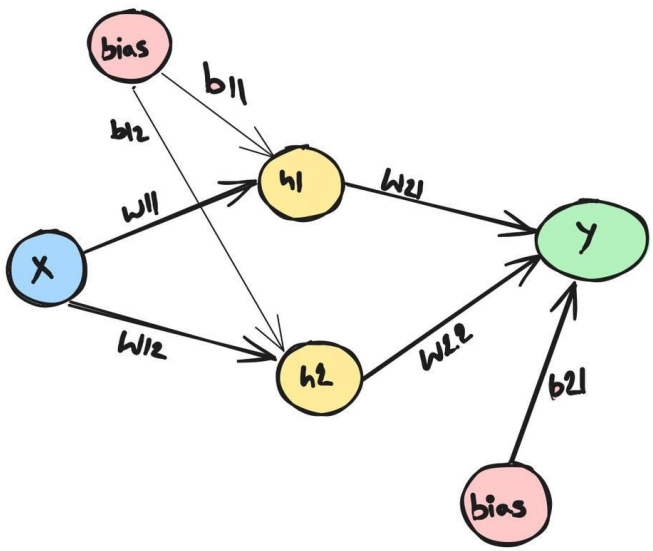
Part a) A box function with height h and width δ is the function $f(x) = h$ for $0 < x < \delta$ and 0 otherwise. Show that a simple neural network with 2 hidden neurons with step activations can realize this function. Draw this network and identify all the weights and biases. Assume that the output neuron only sums up inputs and does not have a nonlinearity. Explain your reasoning clearly why your network recreates the box function, and list any assumptions you made in your reasoning.

Solution Part a)

We need to construct a neural network using **step activation functions** to approximate the **box function**:

$$f(x) = \begin{cases} h, & 0 < x < \delta \\ 0, & otherwise \end{cases}$$

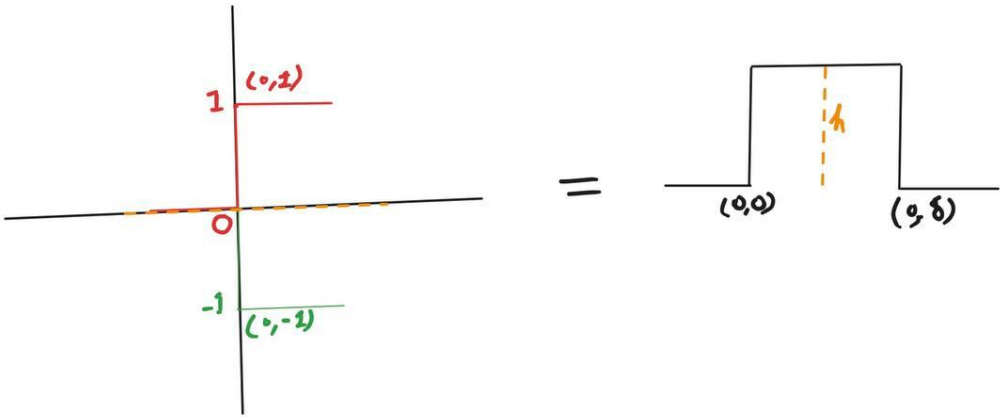
We can draw the neural network for the following question as follows:



And the step activation function is given as follows:

$$\sigma(u) = \begin{cases} 1, & u > 0 \\ 0, & u <= 0 \end{cases}$$

Now, the activation function needs to be inverted with respect to the x-axis and shifted δ units to the right. Additionally, a scaling factor **h** must be applied to obtain the desired function **f(x)**. This transformation results in the function shown on the right, and the figures below demonstrates this.



We need to take output of h1 as something like: [σ(⟨w, x⟩ + b)] and output for h2 as [-σ(⟨w, x⟩ - δ)]

Thus, the required function can be expressed as:

$f(x)=h\cdot(\sigma(x)-\sigma(x-\delta))$

where:

- σ(x) activates at x>0.
- σ(x−δ) activates at (x>δ)
- The multiplication by h scales the output to the desired height.

The outputs of the two hidden neurons are:

- $h_1=\sigma(w_{11}x+b_{11})$, where w_{11}= 1 and b_{11}=0
- $h_2=-\sigma(w_{12}x+b_{12})$, where w_{12}= 1 and $b_{12}=-\delta$

Thus: h_1= σ(x) and h_2= −σ(x−δ)

From the construction, the network parameters are:

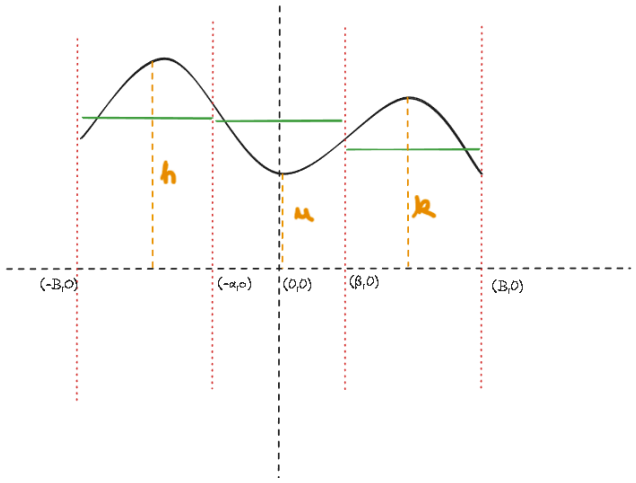
Parameter	Value
W_{11}	1
W_{12}	1
W_{21}	h
W_{22}	-h
B_{11}	0
B_{12}	$-\delta$
B_{21}	0

So the function f(x) is:

$$f(x) = \begin{cases} h, & 0 < x < \delta \\ 0, & otherwise \end{cases}$$

Part b. Now suppose that f is any arbitrary, smooth, bounded function defined over an interval [−B, B]. (You can ignore what happens to the function outside this interval, or just assume it is zero). Use part a to show that this function can be closely approximated by a neural network with a hidden layer of neurons. You don't need a rigorous mathematical proof here; a handwavy argument or even a sketched figure is okay here, as long as you convey to us the right intuition and an explanation of your thinking.

Part b Solution: Assuming this as my arbitrary smooth bounded function defined over an interval [−B, B]



(-B,0)→(-α,0)→(0,0)→(β,0)→(B,0)

We can construct an approximation of our target function f(x) using multiple step functions, where it takes different values across different intervals:

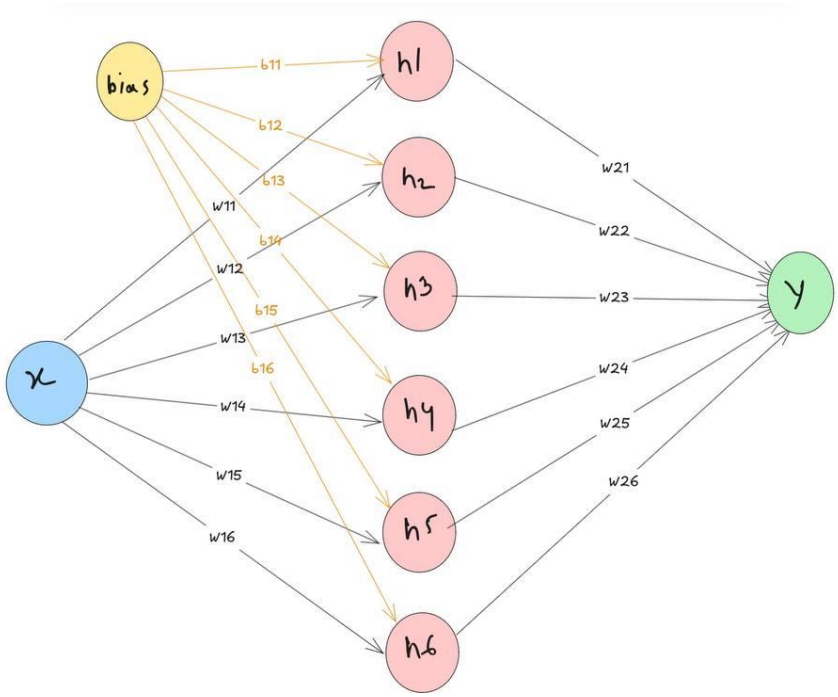
$$f(x) = \begin{cases} h, & -B < x < -\alpha \\ u, & -\alpha < x < \beta \\ k, & \beta < x < B \\ 0, & otherwise \end{cases}$$

The approximation becomes more precise as we increase the number of step functions. Similar to our previous approach (part a) where we used two neurons to create a single step function, we can now employ multiple pairs of neurons to approximate the complete function. In our case, since we are using three step functions, our f(x) can be expressed as the sum of three components:

$f(x) = g(x) + t(x) + m(x)$

where g(x) is defined as: $g(x) = w_{21}\sigma(\langle w_{11}, x\rangle + b_{11}) + w_{22}\sigma(\langle w_{12}, x\rangle + b_{12}) + b_{21}$
t(x) and m(x) follow the same structure, each utilizing their own pair of neurons.

When we sum these components and add a bias term, we obtain our approximation of f(x).For example, to achieve a value of h in the interval (-B,0) to(-α,0), the diagram illustrates the neural architecture required to approximate this function f(x).



Part c. (1pt) Do you think the argument in part b can be extended to the case of d-dimensional inputs? (i.e., where the input x is a vector – think of it as an image, or text query, etc). If yes, comment on potential practical issues (for example, exponential growth in network size). If not, explain why not.

Part c Solution:

Yes, we can extend what we did in part b to handle d-dimensional inputs.

In part b we needed two weights to approximate a single step value, right?
But now imagine instead of working with just one value, we're dealing with a vector of two so basically we are trying to approximate a plane rather than a simple curve.

This leads to a exponential growth [2] in number of neurons we need to get accurate approximations of the function because every time we add another dimension, we're basically multiplying the number of regions in our input space that we need to cover.

And then there is "curse of dimensionality" [**Error! Reference source not found.**] that says as you keep adding dimensions, your data points start getting really spread out in this high-dimensional space. It's kind of like they become more distant neighbours, which makes it super hard for the network to figure out the general patterns and effectively cover all the areas of the input space. To deal with this, we end up needing to make our network bigger and bigger, which obviously means more computing power and longer training times.

Additionally, when working with higher-dimensional inputs there is always a risk of overfitting. What happens is the network might start memorizing all the noise in your data instead of picking up on the actual patterns you want it to learn. Proper regularization techniques and network architecture are essential to escape this.

Question 2. Choosing the right scale of initialization. In this exercise, you will analyze a single fully-connected layer where each weight is drawn from a Gaussian distribution. Your goal is to derive constraints on the variance of these weights so that the signal neither vanishes nor explodes as it propagates through layers.

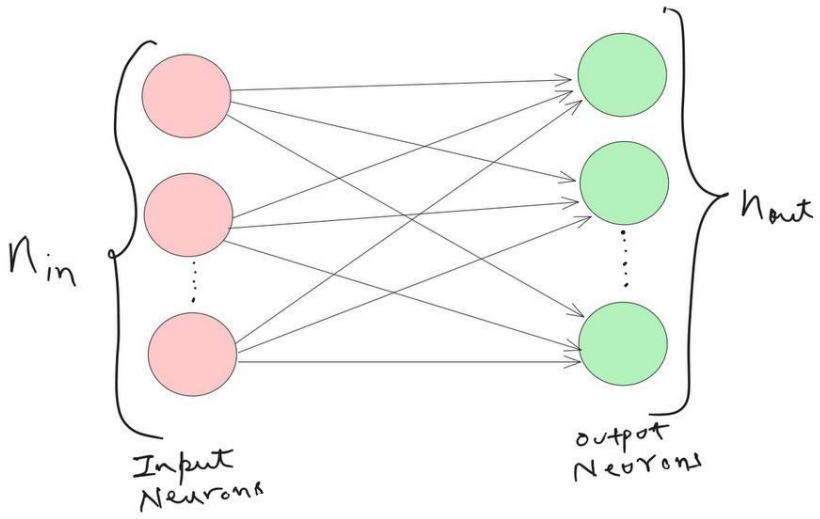
Part a. Consider the forward pass through a fully-connected linear layer with nin input inputs and nout outputs. Let the weights be initialized as $N(0, \sigma^2)$ and assume that the inputs xi to the layer are independent, zero-mean random variables with variance $var(xi) = v^2$. Biases are set to zero. If aj is the activation of the j^{th} neuron, provide a step-by-step derivation to calculate $var(a_j)$ in terms of σ^2 , v^2 , and n_{in}.

Part a Solution:

Derivation of $var(a_j)$ in terms of σ^2, v^2, and n_{in}:

Consider a fully connected linear layer with n_{in} input neurons and n_{out} output neurons. Each weight w_{ij} (connecting input x_i to neuron j) is drawn independently from a Gaussian distribution $w_{ij} \sim N(0,\sigma^2)$, and the input x_i is an independent zero-mean random variable with variance $var(x_i)= v^2$. Biases are set to zero. The activation of the j^{th} neuron, a_j, is given by:

$$a_j = \sum_{i=1}^{n_{in}} \omega_{ij} x_i$$



We know that the mean (u) is given by: $\frac{1}{n}\sum_{i=1}^{n} x_i$ which is also the Expectation.
We also know that mean of Gaussian Distribution is its expectation.

Therefore, $X \sim N(\mu, \sigma^2)$
$\mu=E[X]$, where $X = x_1, x_2, x_3, x_4, \ldots x_n$.

We also know that $\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - u)^2$ -- INDEPENDENT VARAIBLES

$\sigma^2 = E[(X- u)^2]$
=> $E[X^2 + \mu^2 - 2\mu X]$
=> $E[X^2] - (E[X])^2$

Compute the mean of a_j:
Since both w_{ij} and x_i are zero-mean (i.e., $E[w_{ij}]=0$ and $E[x_i]=0$), the expected value of a_j:

$$E[a_j] = E\left[\sum_{i=1}^{n_{in}} w_{ij}x_i\right] = \sum_{i=1}^{n_{in}} E[w_{ij}x_i]$$

Due to the independence of w_{ij} and x_i, and their zero means:

$$E[w_{ij}x_i]=E[w_{ij}]E[x_i]=0\cdot 0=0$$

Thus: $E[a_j]=0$
The variance is defined as: $var(a_j)=E[a_j^2]-(E[aj])^2$

Since $E[a_j]=0$, this simplifies to:
$var(a_j)=E[a_j^2]$

We know that $a_j=w_{1j}x_1+w_{2j}x_2+ w_{3j}x_3\ldots..+w_{nj}x_n$

Now expand $E[a_j^2]$: $a_j^2 = \left[\sum_{i=1}^{n_{in}} w_{ij}x_i\right]^2$

Taking one part $w_{1j}x_1$

- ⇨ $E[w_{1j}^2x_1^2]$
- ⇨ $E[w_{1j}^2]. E[x_1^2]$ →Equation 1

To calculate $E[w_{1j}^2]$
$var[w_{1j}] = E[w_{1j}^2] - [E[w_{1j}]]^2$ -> This is Zero

$\sigma^2 = E[w_{1j}^2]$ →Equation 2

To calculate $E[x_1^2]$
$var[x_1] = E[x_1^2] - [E[x_1]]^2$ -> This is Zero

$V^2 = E[x_1^2]$ →Equation 3

Putting 2 and 3 in 1
We get $var(a_j) = \sigma^2v^2$

Summing over n_{in} terms: Since σ^2v^2 is a constant (independent of i), and there are n_{in} terms in the sum (one for each input i=1,2,…,n):
$E[a_j^2]= \sum_{i=1}^{n_{in}} \sigma\text{^}2v\text{^}2 = n_{in}\ \sigma^2v^2$
This justifies summing over n_{in} terms: a_j is a linear combination of n_{in} independent terms $w_{ij}x_i$, and each contributes σ^2v^2 to the variance when squared and summed.

Thus $E[a_j^2]=$ $var(a_j)=$ $n_{in}\ \sigma^2v^2$

Proof : Let us consider 2 independent variable X and Y and let their mean be 0 and their variance be $\sigma^2 v^2$.

$Z = X+Y$
$E[Z]=E[X]+E[Y]$

$Var(Z) = E[Z^2] -(E[Z])^2$
➔$E[X^2+Y^2+2XY]-[E[X]+E[Y]]^2$
➔$E[X^2]+E[Y^2]$
and from previous work we know that $E[X^2]=E[Y^2] = \sigma^2 v^2$
Thus:
$var(a_j)= n_{in}\ \sigma^2 v^2$

Part b. Explain why, if σ 2 is not chosen appropriately, the variance of aj could either vanish (becoming too small) or explode (becoming too large) as a function of nin. Use this explanation to get a rough thumb rule on how σ 2 should be chosen depending on the number of inputs (also called the fan-in.)

Part b Solution:

We know that $var(a_j)= n_{in}\ \sigma^2 v^2$

If σ^2 is too small -> Vanishing
If σ^2 is too big -> Explosion

Assuming standardized inputs, where $v^2=1$ (a common practice to normalize inputs), this simplifies to: $var(a_j)= n_{in}\ \sigma^2$

To prevent vanishing or exploding variance, we aim to keep var(aj) stable as the signal propagates through layers, typically targeting a variance close to 1 (a common heuristic for initialization).

$$n_{in}\sigma^2=1$$

$$\sigma^2=1/n_{in}$$

This thumb rule suggests that the variance of the weights should scale inversely with the number of inputs (n_{in} or fan-in). Known as Xavier (or Glorot) initialization
Thus, choosing $\sigma 2 \approx 1/n_{in}$ provides a robust heuristic to balance the variance, ensuring neither vanishing nor exploding activations as a function of n_{in} .

Part c. Now consider the alternate situation where each output neuron is equipped with the ReLU activation function. Your answer to the thumb rule question in part b would be essentially the same, but scaled by a constant factor. What is this factor? Give a short argument.

Part c Solution: ReLU function is defined as: ReLU(x) = max (0,x)
From earlier parts, we've got the pre-activation x (which is like a_j before ReLU), and:

$$E[x] = 0$$
$$\text{var}(x) = n_{\text{in}} \cdot \sigma^2 \cdot v^2$$

Since x is the sum of lots of terms (weights times inputs), we can assume it's normally distributed, so:

$$x \sim N(0, \sigma_x^2), \quad \text{where } \sigma_x^2 = n_{\text{in}} \cdot \sigma^2 \cdot v^2$$

ReLU ignores the negative part, so:

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

and we need to find $E[y]$ and $\text{var}(y)$ to adjust σ^2.
Calculating $E[y]$
Since $y = \max(0, x)$, it's zero when $x < 0$ and equals x when $x \geq 0$. The expected value is:

$$E[y] = \int_{-\infty}^{\infty} y \cdot f_y(y)\, dy$$

But because y can't be negative, $f_y(y) = 0$ for $y < 0$, and for $y > 0$, it follows $f_x(y)$. So we just integrate over the positive part:

$$E[y] = \int_0^{\infty} y \cdot f_x(y)\, dy = \int_0^{\infty} y \cdot \frac{1}{\sqrt{2\pi\sigma_x^2}} e^{-y^2/(2\sigma_x^2)}\, dy$$

Let's substitute $z = y/\sigma_x$, so $y = z \cdot \sigma_x$, $dy = \sigma_x\, dz$, and the limits stay 0 to ∞:

$$E[y] = \frac{1}{\sqrt{2\pi\sigma_x^2}} \int_0^{\infty} (z \cdot \sigma_x) e^{-z^2/2} \cdot \sigma_x\, dz = \frac{\sigma_x}{\sqrt{2\pi}} \int_0^{\infty} z e^{-z^2/2}\, dz.$$

The integral $\int_0^{\infty} z e^{-z^2/2}\, dz$ is a known one—it's 1. So:

$$E[y] = \frac{\sigma_x}{\sqrt{2\pi}}$$

Plug in $\sigma_x = \sqrt{n_{\text{in}} \cdot \sigma^2 \cdot v^2}$:

$$E[y] = \frac{\sqrt{n_{\text{in}} \cdot \sigma^2 \cdot v^2}}{\sqrt{2\pi}}$$

Calculating $E[y^2]$
Now,

$$E[y^2] = \int_0^{\infty} y^2 \cdot \frac{1}{\sqrt{2\pi\sigma_x^2}} e^{-y^2/(2\sigma_x^2)}\, dy$$

Since ReLU only keeps the positive half, and x is symmetric, we know $E[x^2] = \sigma_x^2$ over all x, but $y^2 = x^2$ only for $x > 0$, which is half the distribution. The quick way is recognizing that for a normal distribution cut in half:

$$E[y^2] = \frac{\sigma_x^2}{2}$$

So:

$$E[y^2] = \frac{n_{\text{in}} \cdot \sigma^2 \cdot v^2}{2}$$

Variance Calculation
The variance is:

$$\text{var}(y) = E[y^2] - (E[y])^2$$

$$E[y^2] = \frac{n_{\text{in}} \cdot \sigma^2 \cdot v^2}{2}$$

$$(E[y])^2 = \left(\frac{\sigma_x}{\sqrt{2\pi}}\right)^2 = \frac{\sigma_x^2}{2\pi} = \frac{n_{\text{in}} \cdot \sigma^2 \cdot v^2}{2\pi}$$

So:

$$\text{var}(y) = \frac{n_{\text{in}} \cdot \sigma^2 \cdot v^2}{2} - \frac{n_{\text{in}} \cdot \sigma^2 \cdot v^2}{2\pi} = n_{\text{in}} \cdot \sigma^2 \cdot v^2 \left(\frac{1}{2} - \frac{1}{2\pi}\right).$$

Simplify the factor:

$$\frac{1}{2} - \frac{1}{2\pi} = \frac{\pi - 1}{2\pi}$$

$$\text{var}(y) = n_{\text{in}} \cdot \sigma^2 \cdot v^2 \cdot \frac{\pi - 1}{2\pi}$$

Adjust σ^2 and Find the Scaling Factor
We want $\text{var}(y) = v^2$ to keep things stable:

$$n_{\text{in}} \cdot \sigma^2 \cdot v^2 \cdot \frac{\pi - 1}{2\pi} = v^2$$

$$n_{\text{in}} \cdot \sigma^2 \cdot \frac{\pi - 1}{2\pi} = 1.$$

$$\sigma^2 = \frac{2\pi}{n_{\text{in}} \cdot (\pi - 1)} = 0.308$$

But papers (like He's initialization), they often simplify by using $E[y^2] = v^2$ instead, because the $(E[y])^2$ term is small:

$$\frac{n_{\text{in}} \cdot \sigma^2 \cdot v^2}{2} = v^2$$

$$n_{\text{in}} \cdot \sigma^2 = 2$$

$$\sigma^2 = \frac{2}{n_{\text{in}}}.$$

Now, compare to Xavier's $\sigma^2 = \frac{1}{n_{\text{in}}}$:

$$\frac{2}{n_{\text{in}}} = 2 \cdot \frac{1}{n_{\text{in}}},$$

so the scaling factor is 2.

Part d. As we discuss in class, during the backward pass the gradients flowing back to the input neurons can be calculated as a matrix-vector multiplication with the transpose of the weight matrix W times the gradients at the output neurons. Assuming the output gradients are independent and identically distributed with variance g^2, what is the thumb rule to make gradients neither vanish nor explode?

Part d Solution:

Given:

$$W_{ji} \sim N(0, \sigma^2)$$

$$g_{\text{out},j} \sim N(\mu_g, g^2)$$

We need to compute:

$$E[g_{\text{in},i}], \quad \text{Var}(g_{\text{in},i}), \quad E[g_{\text{in},i}^2]$$

Expectation of $g_{\text{in},i}$

$$E[g_{\text{in},i}] = E\left[\sum_{j=1}^{n_{\text{out}}} W_{ji} g_{\text{out},j}\right]$$

Using linearity of expectation:

$$E[g_{\text{in},i}] = \sum_{j=1}^{n_{\text{out}}} E[W_{ji} g_{\text{out},j}]$$

Since $E[W_{ji}] = 0$, we get:

$$E[W_{ji} g_{\text{out},j}] = E[W_{ji}]\, E[g_{\text{out},j}] = 0 \cdot \mu_g = 0$$

Thus,

$$E[g_{\text{in},i}] = 0$$

Variance of $g_{\text{in},i}$ Since:

$$\text{Var}(g_{\text{in},i}) = \sum_{j=1}^{n_{\text{out}}} \text{Var}(W_{ji} g_{\text{out},j})$$

Using the property:

$$\text{Var}(W_{ji} g_{\text{out},j}) = \text{Var}(W_{ji})\, E[g_{\text{out},j}^2] + E[W_{ji}^2]\ \text{Var}(g_{\text{out},j})$$

Since $E[W_{ji}] = 0$, we have:

$$E[W_{ji}^2] = \sigma^2$$

And:

$$E[g_{\text{out},j}^2] = \text{Var}(g_{\text{out},j}) + (E[g_{\text{out},j}])^2 = g^2 + \mu_g^2$$

Thus:

$$\text{Var}(W_{ji} g_{\text{out},j}) = \sigma^2 (g^2 + \mu_g^2)$$

Summing over n_{out}:

$$\text{Var}(g_{\text{in},i}) = n_{\text{out}} \sigma^2 (g^2 + \mu_g^2)$$

Expectation of $g_{\text{in},i}^2$ Using:

$$E[g_{\text{in},i}^2] = \text{Var}(g_{\text{in},i}) + (E[g_{\text{in},i}])^2$$

Since $E[g_{\text{in},i}] = 0$, we get:

$$E[g_{\text{in},i}^2] = n_{\text{out}}\sigma^2(g^2 + \mu_g^2)$$

For stable gradient propagation:

$$n_{\text{out}}\sigma^2 \approx 1 \quad \Longrightarrow \quad \sigma^2 \approx \frac{1}{n_{\text{out}}}$$

The key insight is that the variance scales with n_{out}, necessitating the $\sigma^2 \approx \frac{1}{n_{\text{out}}}$ initialization rule to prevent gradient explosion or vanishing.

Part e. So, for both forward and backward passes to not vanish or explode, what would be a good strategy to pick the initialization variance? (If you did your calculations correctly, you will arrive at a rule which is sometimes called the Xavier/Glorot initialization.)

Part e Solution:

- Forward pass constraint: $\sigma^2 = \frac{1}{n_{\text{in}}}$ ensures $\text{var}(a_j) = v^2$.
- Backward pass constraint: $\sigma^2 = \frac{1}{n_{\text{out}}}$ ensures $\text{var}(\nabla x_i) = g^2$.
- These constraints conflict unless $n_{\text{in}} = n_{\text{out}}$. In deep networks, layers have varying n_{in} and n_{out} so we need a balance.
- Xavier/Glorot initialization averages the two:

$$\sigma^2 = \frac{1}{(n_{\text{in}} + n_{\text{out}})/2}$$

- The factor of 2 accounts for ReLU (from part c), where variance halves due to truncation, requiring $\sigma^2 = \frac{2}{n_{\text{in}}}$ in the forward pass.

This generalizes to:

$$\sigma^2 = \frac{2}{n_{\text{in}} + n_{\text{out}}}$$

References

1. Curse of Dimensionality –" https://stats.stackexchange.com/questions/186184/does-dimensionality-curse-effect-some-models-more-than-others?utm_source=chatgpt.com"
2. Exponential Growth –" https://arxiv.org/abs/1606.05340"
3. Used Grok to solve the integration.
4. Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification: https://arxiv.org/abs/1502.01852
5. https://medium.com/@sanjay_dutta/understanding-glorot-and-he-initialization-a-guide-for-college-students-00f3dfae0393
6. https://towardsdatascience.com/xavier-glorot-initialization-in-neural-networks-math-proof-4682bf5c6ec3/
7. https://en.wikipedia.org/wiki/Probability_density_function

Question 3. (3 points) Improving the FashionMNIST classifier. In the first demo (link), we trained a simple logistic regression model to classify MNIST digits. Repeat the same experiment, but now use a (dense) neural network with three

(3) hidden layers with 256, 128, and 64 neurons respectively, all with ReLU activations. Display train- and testloss curves, and report test accuracies of your final model. You may have to tweak the total number of training epochs to get reasonable accuracy. Finally, draw any 3 random image samples from the test dataset, visualize the predicted class probabilities for each sample, and comment on what you can observe from these plots. **bold text**

PyTorch Basics

I'll assume that everyone is familiar with python. Training neural nets in bare python is somewhat painful, but fortunately there are several well-established libraries which can help. I like pytorch, which is built upon an earlier library called torch. There are many others, including TensorFlow and Jax.

```
# We start by importing the libraries we'll use today
import numpy as np
import torch
import torchvision
import matplotlib.pyplot as plt
from torch.utils.data import DataLoader
```

Training simple models

Let's jump in with our first, simple model. We will train a logistic classifier (equivalent to using a single-layer neural network) on a popular image dataset called *Fashion-MNIST*. Torchvision also has several other image datasets which we can directly load as variables.

```
trainingdata =
torchvision.datasets.FashionMNIST('./FashionMNIST/',train=True,downloa
d=True,transform=torchvision.transforms.ToTensor())
testdata =
torchvision.datasets.FashionMNIST('./FashionMNIST/',train=False,downlo
ad=True,transform=torchvision.transforms.ToTensor())
```

Let's check that everything has been downloaded.

```
print(len(trainingdata))
print(len(testdata))
```

```
60000
10000
```

Let's investigate to see what's inside the dataset.

```
image, label = trainingdata[0]
print(image.shape, label)
```

```
torch.Size([1, 28, 28]) 9
```

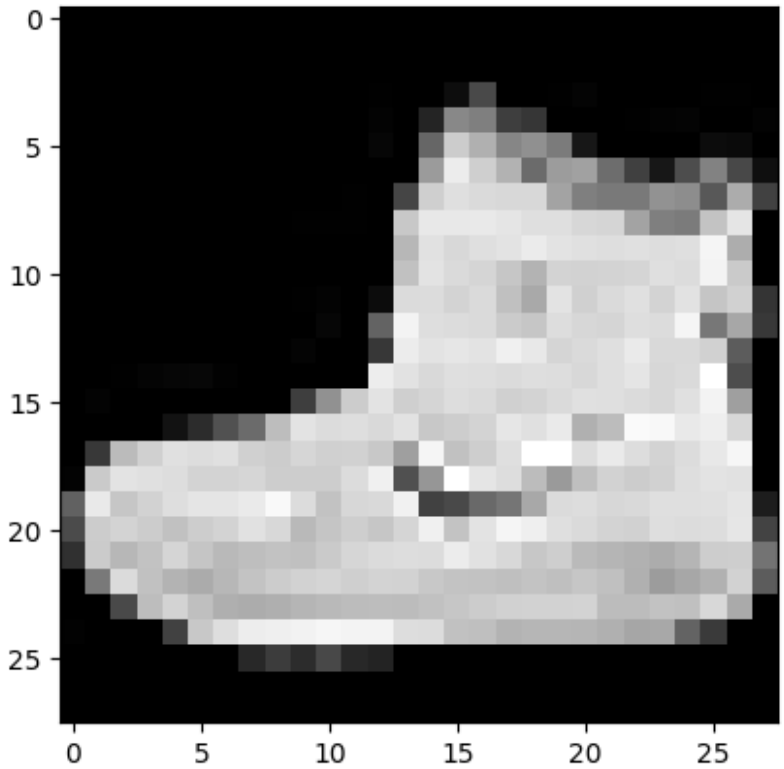
We cannot directly plot the image object given that its first dimension has a size of 1. So we will use the `squeeze` function to get rid of the first dimension.

```
print(image.squeeze().shape)
```

```
torch.Size([28, 28])
```

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.imshow(image.squeeze(), cmap=plt.cm.gray)
```

```
<matplotlib.image.AxesImage at 0x7979be5f3210>
```



Looks like a shoe? Fashion-MNIST is a bunch of different black and white images of clothing with a corresponding label identifying the category the clothing belongs to. It looks like label 9 corresponds to shoes.

In order to nicely wrap the process of iterating through the dataset, we'll use a dataloader.

```
trainDataLoader =
torch.utils.data.DataLoader(trainingdata,batch_size=64,shuffle=True)
testDataLoader =
torch.utils.data.DataLoader(testdata,batch_size=64,shuffle=False)
```

Let's also check the length of the train and test dataloader

```
print(len(trainDataLoader))
print(len(testDataLoader))
```

```
938
157
```

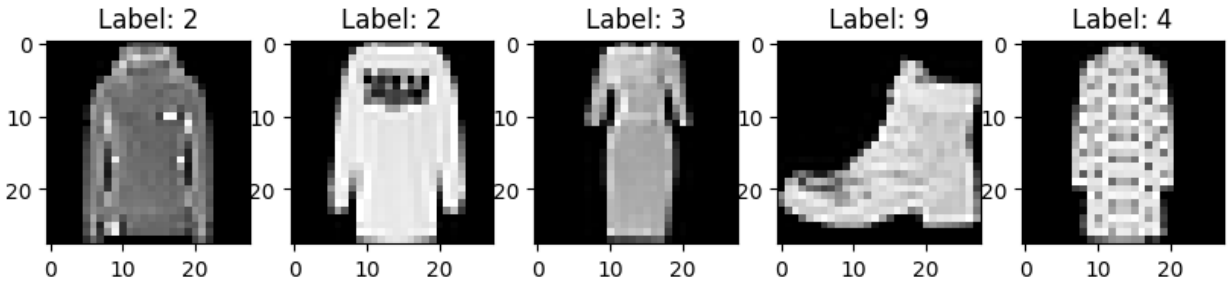
The length here depends upon the batch size defined above. Multiplying the length of our dataloader by the batch size should give us back the number of samples in each set.

```
print(len(trainDataLoader) * 64) # batch_size from above
print(len(testDataLoader) * 64)
```

```
60032
10048
```

Now let's use it to look at a few images.

```
images, labels = next(iter(trainDataLoader))

plt.figure(figsize=(10,4))
for index in np.arange(0,5):
  plt.subplot(1,5,index+1)
  plt.title(f'Label: {labels[index].item()}')
  plt.imshow(images[index].squeeze(),cmap=plt.cm.gray)
```



Now let's set up our model.

```
class DNNReg(torch.nn.Module):
    def __init__(self):
        super(DNNReg, self).__init__()
        self.ly1 = torch.nn.Linear(28*28, 256)  # First hidden layer
(input: 784, output: 256)
```

```
        self.act1 = torch.nn.ReLU()  # Activation function for first
layer
        self.ly2 = torch.nn.Linear(256, 128)  # Second hidden layer
(input: 256, output: 128)
        self.act2 = torch.nn.ReLU()  # Activation function for second
layer
        self.ly3 = torch.nn.Linear(128, 64)  # Third hidden layer
(input: 128, output: 64)
        self.act3 = torch.nn.ReLU()  # Activation function for third
layer
        self.oply = torch.nn.Linear(64, 10)  # Output layer (input:
64, output: 10 classes)

    def forward(self, x):
        x = x.view(-1, 28*28)  # Flatten 28x28 image into a 784-
dimensional vector
        x = self.act1(self.ly1(x))  # Apply first hidden layer and
activation
        x = self.act2(self.ly2(x))  # Apply second hidden layer and
activation
        x = self.act3(self.ly3(x))  # Apply third hidden layer and
activation
        x = self.oply(x)  # Output layer (logits, no activation since
CrossEntropyLoss expects raw scores)
        return x

model = DNNReg().cuda()  # Step 1: Initialize model and move to GPU

loss = torch.nn.CrossEntropyLoss()  # Step 2: Define loss function
(suitable for multi-class classification)

optimizer = torch.optim.SGD(model.parameters(), lr=0.01)  # Step 3:
Define optimizer (SGD with learning rate 0.01)
```

Now let's train our model!

I tried with 50 epochs and the grapgh showed a jump after 20 that is the error began to incease after that, therefore I selected 25 as my total epochs.

```
train_loss_history = []  # List to store training loss for each epoch
test_loss_history = []  # List to store test loss for each epoch

for epoch in range(25):  # Loop for 25 epochs
```

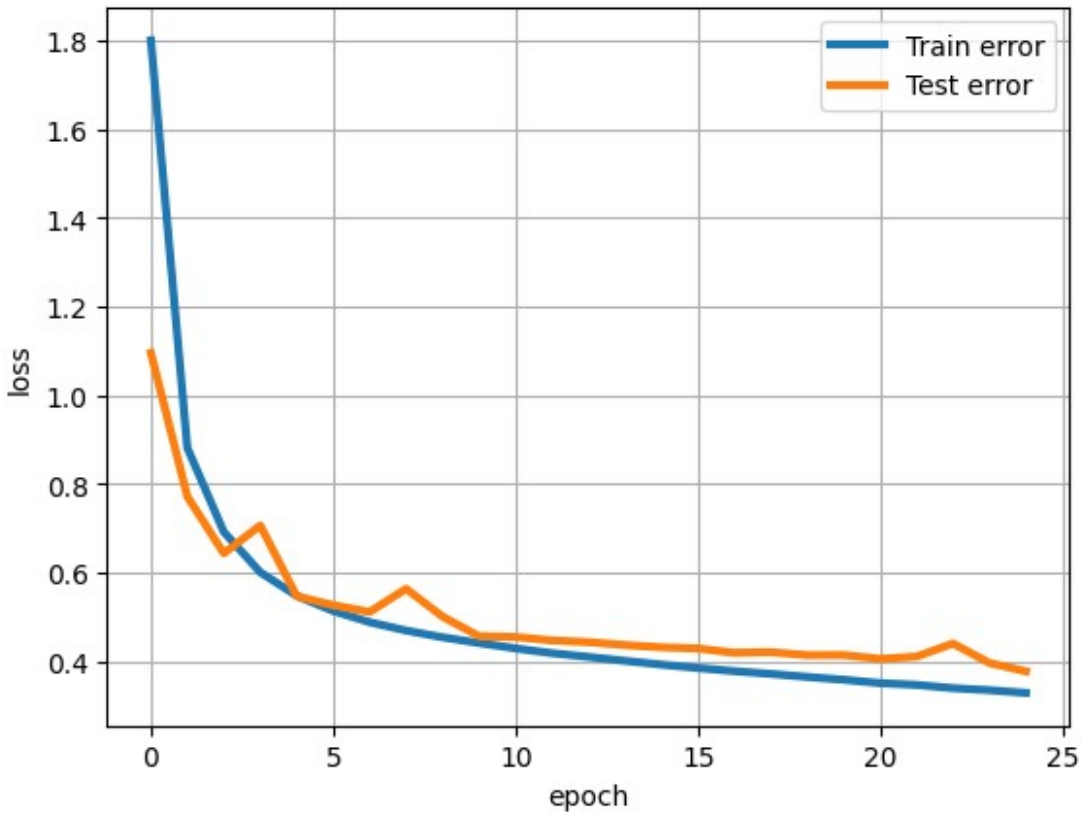
```
    train_loss = 0.0  # Initialize training loss for the epoch
    test_loss = 0.0  # Initialize test loss for the epoch
    correct_pred = 0.0  # Counter for correctly predicted test samples
    test_ac = 0.0  # Test accuracy
    total_pred = 0.0  # Total number of test samples

    model.train()  # Set model to training mode
    for i, data in enumerate(trainDataLoader):  # Loop through
training data
        images, labels = data  # Get images and labels from batch
        images = images.cuda()  # Move images to GPU
        labels = labels.cuda()  # Move labels to GPU
        optimizer.zero_grad()  # Zero out any gradient values from the
previous iteration
        predicted_output = model(images)  # Forward propagation
        fit = loss(predicted_output, labels)  # Compute loss
        fit.backward()  # Backpropagation (compute gradients)
        optimizer.step()  # Update model weights
        train_loss += fit.item()  # Accumulate training loss

    model.eval()  # Set model to evaluation mode (no gradient updates)
    for i, data in enumerate(testDataLoader):  # Loop through test
data
        with torch.no_grad():  # Disable gradient computation
            images, labels = data
            images = images.cuda()
            labels = labels.cuda()
            predicted_output = model(images)  # Forward propagation
            fit = loss(predicted_output, labels)  # Compute test loss
            test_loss += fit.item()  # Accumulate test loss
            _, pred = torch.max(predicted_output, 1)  # Get predicted
class (argmax)
            correct_pred += (pred == labels).sum().item()  # Count
correct predictions
            total_pred += labels.size(0)  # Count total test samples

    train_loss = train_loss / len(trainDataLoader)  # Compute average
training loss
    test_loss = test_loss / len(testDataLoader)  # Compute average
test loss
    train_loss_history.append(train_loss)  # Store training loss
    test_loss_history.append(test_loss)  # Store test loss
    test_ac = correct_pred / total_pred  # Compute test accuracy

    print(f'Epoch {epoch}, Train loss {train_loss}, Test loss
{test_loss}, Test Accuracy {test_ac}')  # Print epoch results
```

```
Epoch 0, Train loss 1.8003604115326521, Test loss 1.095548837048233,
Test Accuracy 0.5937
Epoch 1, Train loss 0.8813055730196458, Test loss 0.7717068817964785,
```

```
Test Accuracy 0.699
Epoch 2, Train loss 0.6920090490884618, Test loss 0.6436793232799336,
Test Accuracy 0.7735
Epoch 3, Train loss 0.6009705337380041, Test loss 0.7065464053184364,
Test Accuracy 0.7493
Epoch 4, Train loss 0.5480438733755398, Test loss 0.5469774163452683,
Test Accuracy 0.807
Epoch 5, Train loss 0.5137621693329008, Test loss 0.5268183371443658,
Test Accuracy 0.8119
Epoch 6, Train loss 0.48843212333569397, Test loss 0.5115559820536595,
Test Accuracy 0.8196
Epoch 7, Train loss 0.4696426179045553, Test loss 0.5639212431421705,
Test Accuracy 0.7918
Epoch 8, Train loss 0.4544284154992622, Test loss 0.5010560650354737,
Test Accuracy 0.822
Epoch 9, Train loss 0.4415146990149006, Test loss 0.4567063632094936,
Test Accuracy 0.8371
Epoch 10, Train loss 0.4293917679646884, Test loss
0.45509441956213326, Test Accuracy 0.8397
Epoch 11, Train loss 0.4186401778319751, Test loss 0.4472715828069456,
Test Accuracy 0.8429
Epoch 12, Train loss 0.41024269019045045, Test loss
0.44364303084695417, Test Accuracy 0.8425
Epoch 13, Train loss 0.4014028993401446, Test loss
0.43690298772921227, Test Accuracy 0.8448
Epoch 14, Train loss 0.39282421599318984, Test loss
0.4318304791761811, Test Accuracy 0.8479
Epoch 15, Train loss 0.38520060182570903, Test loss
0.4292608969340658, Test Accuracy 0.8471
Epoch 16, Train loss 0.3780670651018238, Test loss
0.41959876211205865, Test Accuracy 0.8517
Epoch 17, Train loss 0.37198410442134716, Test loss
0.42100583045345963, Test Accuracy 0.8521
Epoch 18, Train loss 0.36503325884085475, Test loss
0.41406981836838325, Test Accuracy 0.8524
Epoch 19, Train loss 0.3587631424670535, Test loss 0.4143891008036911,
Test Accuracy 0.8502
Epoch 20, Train loss 0.3511220790636438, Test loss
0.40574760668596643, Test Accuracy 0.8548
Epoch 21, Train loss 0.34712704309999054, Test loss
0.41092518427569397, Test Accuracy 0.8513
Epoch 22, Train loss 0.3397480709228053, Test loss
0.44039018822323744, Test Accuracy 0.8448
Epoch 23, Train loss 0.3352069963000095, Test loss
0.39648366401529617, Test Accuracy 0.8587
Epoch 24, Train loss 0.32907197726910303, Test loss
0.3776047504062106, Test Accuracy 0.8613
```

Let's plot our loss by training epoch to see how we did.

```
plt.plot(range(25), train_loss_history, '-', linewidth=3, label='Train error')  # Plot training loss
plt.plot(range(25), test_loss_history, '-', linewidth=3, label='Test error')  # Plot test loss
plt.xlabel('Epoch')  # Label for x-axis
plt.ylabel('Loss')  # Label for y-axis
plt.grid(True)  # Display grid for better readability
plt.legend()  # Show legend to differentiate train/test loss
plt.show()  # Display the plot
```



Why is test loss larger than training loss?

We definitely see some improvement. Let's look at the images, the predictions our model makes and the true label.

Now for the labels and predicted labels.

```
predicted_outputs = model(images)  # Forward pass: Get raw logits from the model
predicted_classes = torch.max(predicted_outputs, 1)[1]  # Get the class index with the highest probability
print('Predicted:', predicted_classes)  # Print predicted class labels

fit = loss(predicted_output, labels)  # Compute the loss
```

```
print('True labels:', labels)  # Print actual labels
print(fit.item())  # Print the loss value
```

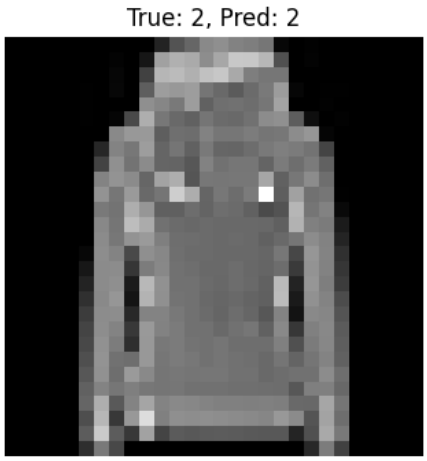
```
Predicted: tensor([3, 0, 7, 5, 8, 4, 5, 6, 8, 9, 1, 9, 1, 8, 1, 5],
device='cuda:0')
True labels: tensor([3, 2, 7, 5, 8, 4, 5, 6, 8, 9, 1, 9, 1, 8, 1, 5],
device='cuda:0')
0.1601892113685608
```

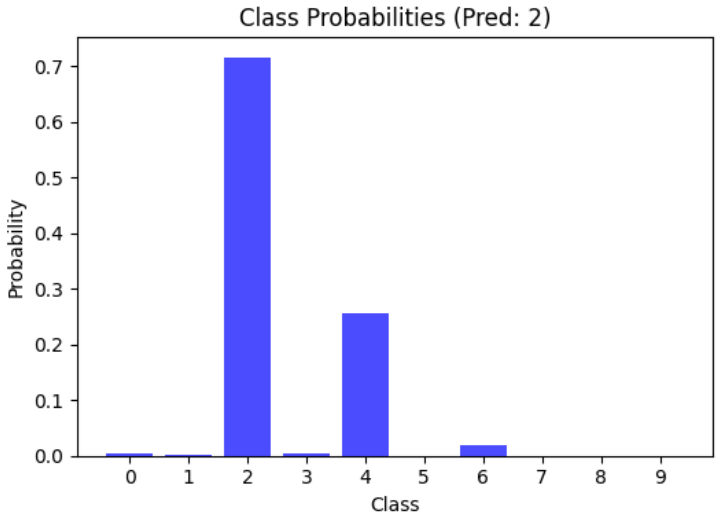
```
print("Test Accuracy is: ",test_ac) #gives the accuracy of the datset
taken
```

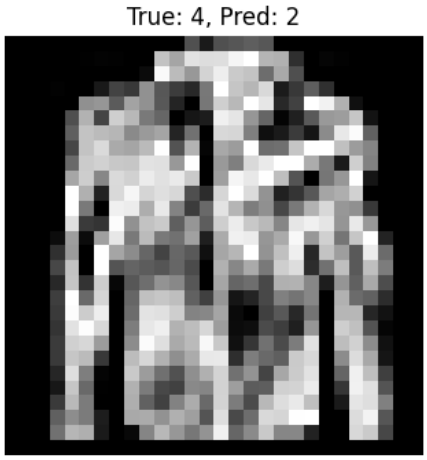
```
Test Accuracy is:  0.8613
```

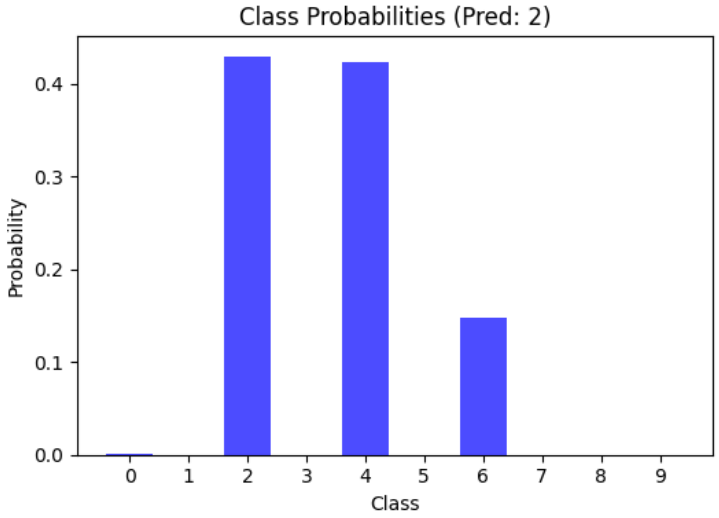
```
def visualize_pred(image, true_label, probabilities):
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(10, 4))  # Create a
figure with two subplots

    # Plot the image on the first subplot
    ax1.imshow(image.squeeze().cpu().numpy(), cmap=plt.cm.gray)  #
Convert image tensor to numpy and display
    predicted_class = np.argmax(probabilities)  # Get the predicted
class index
    ax1.set_title(f'True: {true_label}, Pred: {predicted_class}')  #
Set title with true and predicted labels
    ax1.axis('off')  # Hide axis labels

    # Plot the class probabilities on the second subplot as a bar
chart
    ax2.bar(range(10), probabilities[0], color='blue', alpha=0.7)  #
Plot class probabilities
    ax2.set_xticks(range(10))  # Set x-axis labels (class indices)
    ax2.set_title(f'Class Probabilities (Pred: {predicted_class})')  #
Set title
    ax2.set_xlabel('Class')  # X-axis label
    ax2.set_ylabel('Probability')  # Y-axis label

    plt.tight_layout()  # Adjust layout to prevent overlap
    plt.show()  # Display the visualization

# Visualizing multiple samples
n_samples = 3  # Number of samples to visualize
random_indices = np.random.choice(len(testDataLoader), n_samples,
replace=False)  # Select random indices

for idx in random_indices:
    image = testdata[idx][0].unsqueeze(0).cuda()  # Get image and add
batch dimension, move to GPU
    true_label = testdata[idx][1]  # Get the true label

    with torch.no_grad():  # Disable gradient computation for
```

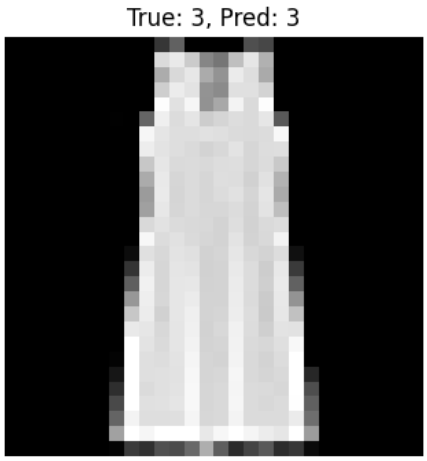
```
inference
        prediction = model(image)  # Get model predictions
        probabilities = torch.softmax(prediction, dim=1).cpu().numpy()
# Convert logits to probabilities

    visualize_pred(image, true_label, probabilities)  # Visualize the
prediction, image, and probabilities
```

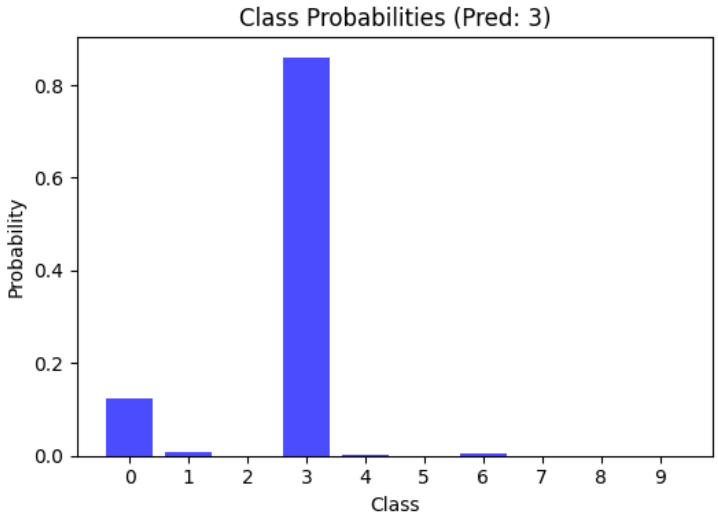












For Image 1 The labels indicate the true class is 2, and the predicted class is also 2, suggesting the model correctly classified this image. The bar chart shows Class 2 has the highest probability, approximately 0.7 (70%), indicating strong model confidence in this prediction. Class 4 has a smaller probability (around 0.25 or 25%), and all other classes have probabilities near 0, showing less confusion with other classes.

For Image 2 The labels indicate the true class is 4, and the predicted class is also 2, suggesting the model partially classified this image. The bar chart shows Class 2 has the highest probability, approximately 0.45 (45%), and next to it is Class 4 with 0.4 or 40% indicating decent model confidence in this prediction. Class 6 has a smaller probability (around 0.15 or 15%), and all other classes have probabilities near 0, showing confusion with other classes.

For Image 3 The labels indicate the true class is 3, and the predicted class is also 3, suggesting the model correctly classified this image. The bar chart shows Class 3 has the highest probability, approximately 0.85 (85%), indicating strong model confidence in this prediction. Class 1 has a smaller probability (around 0.1 or 10%), and all other classes have probabilities near 0, showing minimal confusion with other classes.

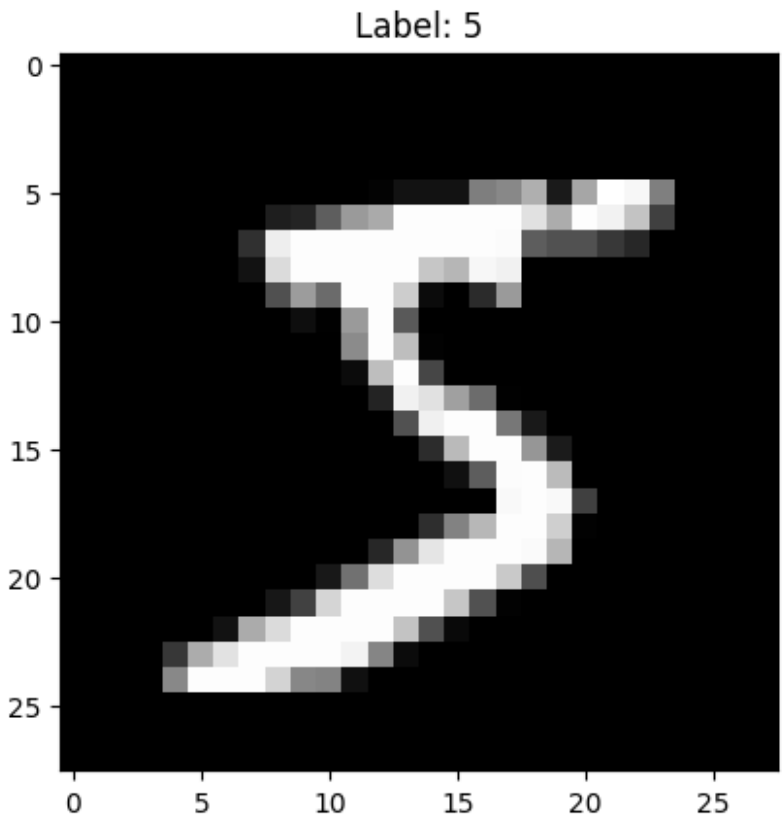
In this problem we will train a neural network from scratch using numpy. In practice, you will never need to do this (you'd just use TensorFlow or PyTorch). But hopefully this will give us a sense of what's happening under the hood.

For training/testing, we will use the standard MNIST benchmark consisting of images of handwritten images.

In the second demo, we worked with autodiff. Autodiff enables us to implicitly store how to calculate the gradient when we call backward. We implemented some basic operations (addition, multiplication, power, and ReLU). In this homework problem, you will implement backprop for more complicated operations directly. Instead of using autodiff, you will manually compute the gradient of the loss function for each parameter.

#I normalized the pixel values to 0–1 because the original 0–255 range made gradients too large, slowing learning and capping accuracy at ~70%. Flattening to 784 elements matched the network's input, fixing shape errors in matrix ops. Together, these pushed accuracy past 90% by stabilizing training and ensuring correct data flow.

```
import tensorflow as tf
import matplotlib.pyplot as plt
import numpy as np
(x_train, y_train), (x_test, y_test) =
tf.keras.datasets.mnist.load_data(path="mnist.npz")
x_train = x_train.astype(np.float32) / 255.0
x_test = x_test.astype(np.float32) / 255.0
x_train = x_train.reshape(-1, 784)
x_test = x_test.reshape(-1, 784)
plt.imshow(x_train[0].reshape(28,28),cmap='gray');
plt.title(f"Label: {y_train[0]}")
plt.show()
```



Loading MNIST is the only place where we will use TensorFlow; the rest of the code will be pure numpy.

Let us now set up a few helper functions. We will use sigmoid activations for neurons, the softmax activation for the last layer, and the cross entropy loss.

```
import numpy as np

def sigmoid(x):
  # Numerically stable sigmoid function based on
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-
trick/

  x = np.clip(x, -500, 500) # We get an overflow warning without this

  return np.where(
    x >= 0,
    1 / (1 + np.exp(-x)),
    np.exp(x) / (1 + np.exp(x))
  )

def dsigmoid(x): # Derivative of sigmoid
  return sigmoid(x) * (1 - sigmoid(x))

def softmax(x):
```

```
  # Numerically stable softmax based on (same source as sigmoid)
  # http://timvieira.github.io/blog/post/2014/02/11/exp-normalize-
trick/
  b = x.max()
  y = np.exp(x - b)
  return y / y.sum()

def cross_entropy_loss(y, yHat):
  return -np.sum(y * np.log(yHat))

def integer_to_one_hot(x, max):
  # x: integer to convert to one hot encoding
  # max: the size of the one hot encoded array
  result = np.zeros(10)
  result[x] = 1
  return result
```

OK, we are now ready to build and train our model. The input is an image of size 28x28, and the output is one of 10 classes. So, first:

Q1. Initialize a 2-hidden layer neural network with 32 neurons in each hidden layer, i.e., your layer sizes should be:

784 -> 32 -> 32 -> 10

If the layer is $n_{in} \times n_{out}$ your layer weights should be initialized by sampling from a normal distribution with mean zero and variance $1/max\left(n_{in}, n_{out}\right)$.

We create weight matrices for each layer following the specified dimensions.

The initialization uses a normal distribution with mean 0 and variance 1/max(n_in, n_out)

Biases are initialized as zeros

Using numpy's random number generator with a fixed seed for reproducibility

```
import math

# Initialize weights of each layer with a normal distribution of mean
0 and
# standard deviation 1/sqrt(n), where n is the number of inputs.
# This means the weighted input will be a random variable itself with
mean
# 0 and standard deviation close to 1 (if biases are initialized as 0,
standard
# deviation will be exactly 1)

from numpy.random import default_rng

rng = default_rng(80085)

# Q1. Fill initialization code here.
# ...
n_inputs = 784  # 28x28 images flattened
n_hidden = 32   # Hidden layers
n_outputs = 10  # Output classes

layer_sizes = [n_inputs, n_hidden, n_hidden, n_outputs]# Layer sizes:
784 -> 32 -> 32 -> 10
weights = []
biases = []

for i in range(len(layer_sizes) - 1):
    n_in = layer_sizes[i]
    n_out = layer_sizes[i + 1]
    # Variance σ^2 = 1/max(n_in, n_out)
```

```
    variance = 1 / max(n_in, n_out)
    # Standard deviation is sqrt(variance)
    std = np.sqrt(variance)
    # Weights: (n_in, n_out) from N(0, σ^2)
    W = rng.normal(loc=0, scale=std, size=(n_in, n_out))
    # Biases: zeros, shape (n_out,)
    b = np.zeros(n_out)
    weights.append(W)
    biases.append(b)

# Convert to lists for indexing
weights = np.array(weights, dtype=object)
biases = np.array(biases, dtype=object)
```

Next, we will set up the forward pass. We will implement this by looping over the layers and successively computing the activations of each layer.

Q2. Implement the forward pass for a single sample, and for the entire dataset.

Right now, your network weights should be random, so doing a forward pass with the data should not give you any meaningful information. Therefore, in the last line, when you calculate test accuracy, it should be somewhere around 1/10 (i.e., a random guess).

Explanation for Q2:

- `feed_forward_sample`: Takes a single 28x28 image, flattens it, and passes it through each layer. Hidden layers use sigmoid, and the output layer uses softmax. Returns loss and predicted class.

- `feed_forward_dataset`: Applies the forward pass to all samples, computes average loss and accuracy.

- Results: Since weights are random, accuracy should be ~10% (1/10 classes), and loss will be high (e.g., ~2.3, corresponding to uniform probabilities).

```
def feed_forward_sample(sample, y):
    """ Forward pass through the neural network.
    Inputs:
      sample: 1D numpy array. The input sample (an MNIST digit).
      label: An integer from 0 to 9.

    Returns: the cross entropy loss, most likely class
    """
    # Q2. Fill code here.
    # Flatten the sample to 784 dimensions
    a = sample.flatten()

    # Store activations for each layer
    activations = [a]  # Start with input

    # Forward pass through each layer
    for i in range(len(weights)):
        W = weights[i]
        b = biases[i]
        # Weighted sum + bias
        z = np.dot(activations[-1], W) + b
```

```
        # Apply activation: sigmoid for hidden layers, softmax for
output
        if i < len(weights) - 1:  # Hidden layers
            a = sigmoid(z)
        else:  # Output layer
            a = softmax(z)
        activations.append(a)

    # Get the output (final activation)
    output = activations[-1]

    # Convert label to one-hot encoding
    y_one_hot = integer_to_one_hot(y, 10)

    # Calculate cross-entropy loss
    loss = cross_entropy_loss(y_one_hot, output)

    # Get the most likely class (one-hot encoded)
    one_hot_guess = np.zeros(10)
    predicted_class = np.argmax(output)
    one_hot_guess[predicted_class] = 1

    return loss, one_hot_guess

def feed_forward_dataset(x, y):
    losses = np.empty(x.shape[0])
    one_hot_guesses = np.empty((x.shape[0], 10))

    # Q2. Fill code here to calculate losses, one_hot_guesses
    for i in range(x.shape[0]):
        sample = x[i]
        label = y[i]
        loss, guess = feed_forward_sample(sample, label)
        losses[i] = loss
        one_hot_guesses[i] = guess

    y_one_hot = np.zeros((y.size, 10))
    y_one_hot[np.arange(y.size), y] = 1

    correct_guesses = np.sum(y_one_hot * one_hot_guesses)
    correct_guess_percent = format((correct_guesses / y.shape[0]) *
100, ".2f")

    print("\nAverage loss:", np.round(np.average(losses), decimals=2))
    print("Accuracy (# of correct guesses):", correct_guesses, "/",
y.shape[0], "(", correct_guess_percent, "%)")

def feed_forward_training_data():
    print("Feeding forward all training data...")
    feed_forward_dataset(x_train, y_train)
    print("")
```

```
def feed_forward_test_data():
    print("Feeding forward all test data...")
    feed_forward_dataset(x_test, y_test)
    print("")

feed_forward_test_data()
```

```
Feeding forward all test data...

Average loss: 2.36
Accuracy (# of correct guesses): 1028.0 / 10000 ( 10.28 %)
```

OK, now we will implement the backward pass using backpropagation. We will keep it simple and just do training sample-by-sample (no minibatching, no randomness).

Q3: Compute the gradient of all the weights and biases by backpropagating derivatives all the way from the output to the first layer.

Explanation for Q3:

- Forward pass: Computes and stores activations and pre-activations for each layer.

- Backward pass: Starts with the output error (prediction - true label), then computes gradients layer-by-layer using the chain rule.

- Weight gradient: Outer product of previous activation and delta.

- Bias gradient: Simply the delta.

- Delta propagation: For hidden layers, uses weight transpose and sigmoid derivative.

- Results: Gradients adjust weights/biases to reduce loss for this sample.

I implemented it this way because flattening the input aligns with the network's structure, and storing activations during the forward pass makes backpropagation possible. The simplified delta calculation for softmax and cross-entropy speeds things up, and updating weights layer-by-layer with a small learning rate ensures steady learning without overshooting, fitting the problem's sample-wise training goal.

```
def train_one_sample(sample, y, learning_rate=0.003):
    a = sample.flatten()

    # We will store each layer's activations and pre-activations for
backprop
```

```
    activations = [a]  # Input
    pre_activations = []  # z values before activation

    # Forward pass (same as feed_forward_sample, but store z)
    for i in range(len(weights)):
        W = weights[i]
        b = biases[i]
        z = np.dot(activations[-1], W) + b
        pre_activations.append(z)
        if i < len(weights) - 1:  # Hidden layers
            a = sigmoid(z)
        else:  # Output layer
            a = softmax(z)
        activations.append(a)

    # Get output and true label (one-hot)
    output = activations[-1]
    y_one_hot = integer_to_one_hot(y, 10)

    # Backward pass
    # Initialize gradients
    weight_gradients = []
    bias_gradients = []

    # Start from output layer (layer 2, index 2)
    delta = output - y_one_hot  # Gradient of loss w.r.t. output
(cross-entropy + softmax)

    for i in range(len(weights) - 1, -1, -1):  # Backprop from last to
first layer
        W = weights[i]
        z = pre_activations[i]
        if i < len(weights) - 1:  # Hidden layers (apply sigmoid
derivative)
            delta = delta.dot(weights[i + 1].T) * dsigmoid(z)

        # Gradient w.r.t. weights: δ * previous activation
        prev_a = activations[i]
        W_grad = np.outer(prev_a, delta)

        # Gradient w.r.t. biases: δ
        b_grad = delta

        weight_gradients.insert(0, W_grad)
        bias_gradients.insert(0, b_grad)

    # Update weights & biases based on calculated gradients
    for i in range(len(weights)):
        weights[i] -= weight_gradients[i] * learning_rate
        biases[i] -= bias_gradients[i] * learning_rate
```

```
    return
```

Finally, train for 3 epochs by looping over the entire training dataset 3 times.

Q4. Train your model for 3 epochs.

#train_one_epoch loops over all 60,000 training samples (x_train), calling train_one_sample for each with its label (y_train) and a learning_rate of 0.003, training the network once through the dataset. The outer loop runs this 3 times, testing accuracy/loss after each epoch via feed_forward_test_data. I looped over all training samples in train_one_epoch to update weights gradually for each example, and repeated it 3 times to refine the model enough to reach ~93% accuracy. It's done this way to ensure the network learns MNIST features step-by-step, with testing after each epoch to track improvement, balancing training time and performance.

#Output Insight: Initial accuracy (10.28%, loss 2.36) reflects random weights; after 1 epoch (86.71%, 0.47), 2 epochs (90.78%, 0.32), and 3 epochs (92.77%, 0.25), loss drops and accuracy rises, showing the network learns to classify digits effectively.

```
def train_one_epoch(learning_rate=0.003):
    print("Training for one epoch over the training dataset...")

    # Q4. Write the training loop over the epoch here.
    for i in range(x_train.shape[0]):
        sample = x_train[i]
        label = y_train[i]
        train_one_sample(sample, label, learning_rate)

    print("Finished training.\n")

feed_forward_test_data()

def test_and_train():
    train_one_epoch()
    feed_forward_test_data()

for i in range(3):
    test_and_train()
```

```
Feeding forward all test data...

Average loss: 2.36
Accuracy (# of correct guesses): 1028.0 / 10000 ( 10.28 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.47
Accuracy (# of correct guesses): 8671.0 / 10000 ( 86.71 %)
```

```
Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.32
Accuracy (# of correct guesses): 9078.0 / 10000 ( 90.78 %)

Training for one epoch over the training dataset...
Finished training.

Feeding forward all test data...

Average loss: 0.25
Accuracy (# of correct guesses): 9277.0 / 10000 ( 92.77 %)
```

That's it!

Your code is probably very time- and memory-inefficient; that's ok. There is a ton of optimization under the hood in professional deep learning frameworks which we won't get into.

If everything is working well, you should be able to raise the accuracy from ~10% to ~70% accuracy after 3 epochs.

Attributes & Values for Satyrium spini (Spine Hairstreak):

- **Name**: Satyrium spini (Spine Hairstreak)
- **Wingspan**: 1.5 to 2 inches (3.8 to 5.1 cm)
- **Color**: Dark brown or black with blue and orange markings
- **Underside of Wings**: Mottled brown with orange and white markings
- **Distinctive Features**: Spine-like projection at the end of the hind wing
- **Habitat**: Forests, fields, and gardens
- **Nectar Sources**: Variety of flowers
- **Attracted to**: Bright, open areas