Numpy, TF, and Visualization

Start by importing necessary packages

We will begin by importing necessary libraries for this notebook. Run the cell below to do so.

```
import numpy as np
import matplotlib.pyplot as plt
import math
import tensorflow as tf
from scipy.special import erf
```

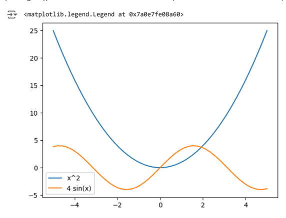
Visualizations

Visualization is a key factor in understanding deep learning models and their behavior. Typically, pyplot from the matplotlib package is used, capable of visualizing series and 2D data.

Below is an example of visualizing series data.

```
x = np.linspace(-5, 5, 50) # create a linear spacing from x = -5.0 to 5.0 with 50 steps

y1 = x**2      # create a series of points {y1}, which corresponds to the function f(x) = y^2
y2 = 4*np.sin(x) # create another series of points {y2}, which corresponds to the function f(x) = 4*sin(x)  NOTE: we have to use np.sin and not math.sin as math.sin will only act on individual values
# to use math.sin, we could have used a list comprehension instead: y2 = [math.sin(xi) for xi in x]

# by default, matplotlib will behave like MATLAB with hold(True), overplotting until a new figure object is created
plt.plot(x, y1, label="x^2")        # plot y1 with x as the x-axis series, and label the line "x^2"
plt.plot(x, y2, label="4 sin(x)")   # plot y2 with x as the x-axis series, and label the line "4 sin(x)"
plt.legend()                        # have matplotlib show the label on the plot
```

```
<matplotlib.legend.Legend at 0x7a0e7fe08a60>
```



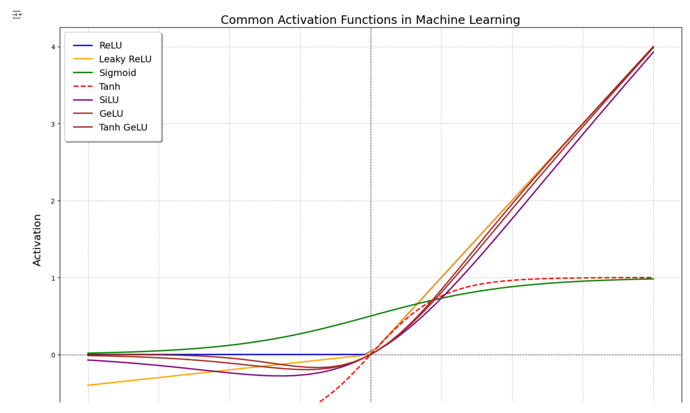
More complex formatting can be added to increase the visual appeal and readability of plots (especially for paper quality figures). To try this out, let's consider plotting a few of the more common activation functions used in machine learning. Below, plot the following activation functions for $x \in [-4, 4]$:

- ReLU: $max(x, 0)$
- Leaky-ReLU: $max(0.1 \cdot x, x)$
- Sigmoid: $\sigma(x) = 1/(1 + e^{-x})$
- Hyperbolic Tangent: $\tanh(x) = (e^{x} - e^{-x})/(e^{x} + e^{-x})$
- SiLU: $x \cdot \sigma(x)$
- GeLU: $x \cdot \frac{1}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$
- tanh GELU: $x \cdot \frac{1}{2}\left(1 + \tanh\left(\frac{x}{\sqrt{2}}\right)\right)$

Plot the GELU and tanh GELU using the same color, but with tanh using a dashed line (tanh is a common approximation as the error-function is computationally expensive to compute). You may also need to adjust the legend to make it easier to read. I recommend using ChatGPT to help find the formatting options here.

Question 1

```
# Define activation functions
def relu(x):
    return np.maximum(x, 0)

def leaky_relu(x):
    return np.maximum(0.1 * x, x)

def sigmoid(x):
    return 1 / (1 + np.exp(-x))

def tanh(x):
    return np.tanh(x)

def silu(x):
    return x * sigmoid(x)

def gelu(x):
    return x * 0.5 * (1 + erf(x / np.sqrt(2)))

def tanh_gelu(x):
    return x * 0.5 * (1 + tanh(x / np.sqrt(2)))

# Generate x values
x = np.linspace(-4, 4, 50)

# Generate y values for each activation function
y_relu = relu(x)
y_leaky_relu = leaky_relu(x)
y_sigmoid = sigmoid(x)
y_tanh = tanh(x)
y_silu = silu(x)
y_gelu = gelu(x)
y_tanh_gelu = tanh_gelu(x)

# Plot
plt.figure(figsize=(14, 10))

# Plot each function
plt.plot(x, y_relu, label='ReLU', color='blue', linewidth=2)
plt.plot(x, y_leaky_relu, label='Leaky ReLU', color='orange', linewidth=2)
plt.plot(x, y_sigmoid, label='Sigmoid', color='green', linewidth=2)
plt.plot(x, y_tanh, label='Tanh', linestyle='--', color='red', linewidth=2)
plt.plot(x, y_silu, label='SiLU', color='purple', linewidth=2)
plt.plot(x, y_gelu, label='GeLU', color='brown', linewidth=2)
plt.plot(x, y_tanh_gelu, label='Tanh GeLU', color='brown', linewidth=2)

# Add labels and title
plt.xlabel('x', fontsize=16)
plt.ylabel('Activation', fontsize=16)
plt.title('Common Activation Functions in Machine Learning', fontsize=18)

# Customize legend
plt.legend(loc='upper left', fontsize=14, frameon=True, shadow=True, borderpad=1)

# Add grid
plt.grid(True, linestyle='--', alpha=0.6)

# Highlight axes
plt.axhline(0, color='black', linewidth=0.8, linestyle='--', alpha=0.7)
plt.axvline(0, color='black', linewidth=0.8, linestyle='--', alpha=0.7)

# Tight layout for better spacing
plt.tight_layout()

# Show the plot
plt.show()
```



Answer to the following questions from the the plot you just created:

1. Which activation function is the least computationally expensive to compute?
2. Are there better choices to ensure more stable training? What downfalls do you think it may have?
3. Are there any cases where you would not want to use either activation function?

Question 2

Answer:

1.Computationally Least Expensive: ReLU, because it only has to perform a max(x, 0) operation.

2.What Works Best for Stability vS Gradients: Tanh, Sigmoid, SiLU, and GeLU offer smoother gradients but:

Also Tanh/Sigmoid: Have vanishing gradients and are computationally expensive

SiLU/GeLU: More stable but also expensive.

3.When Not to Use:

ReLU: Do not use if you have a dead neuron problem.

Sigmoid/Tanh: Don't use in deep networks (vanishing gradients).

SiLU/GeLU: Use cautiously in resource-constrained settings.

Leaky ReLU: Is not the best if smooth gradient flow is the preference..

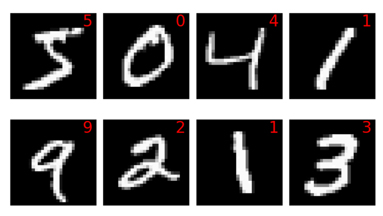
Visualizing 2D data

In many cases, we also want the ability to visualize multi-dimensional data such as images. To do so, matplotlib has the imshow method, which can visualize single channel data with a heatmap, or RGB data with color.

Let's consider visualizing the first 8 training images from the MNIST dataset. MNIST consists of hand drawn digits with their corresponding labels (a number from 0 to 9).

We will use the tensorflow keras dataset library to load the dataset, and then visualize the images with a matplotlib subplot. Because we have so many images, we should arrange them in a grid (4 horizontal, 2 vertical), and plot each image in a loop. Furthermore, we can append the label to each image using the matplotlib utility.

```
from tensorflow.keras.datasets import mnist

(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

# Define the grid dimensions
rows, cols = 2, 4

# Create a figure and axes for the grid
fig, axes = plt.subplots(rows, cols, figsize=(8, 5))

# Iterate through the grid
for i in range(rows):
    for j in range(cols):
        index = i * cols + j
        ax = axes[i, j]

        # Display the image
        ax.imshow(train_images[index], cmap='gray')

        # Display the label on top of the image in red text
        ax.text(0.9, 0.9, str(train_labels[index]), color='red',
                transform=ax.transAxes, fontsize=24,
                ha='center', va='center')

        # Turn off axis labels
        ax.axis('off')

# Adjust spacing and layout
plt.tight_layout()
```



Another popular image dataset for benchmarking and evaluation is CFAR-10. This dataset consists of small (32 x 32 pixel) RGB images of objects that fall into one of 10 classes:

0. airplane
1. automobile
2. bird
3. cat
4. deer
5. dog
6. frog
7. horse
8. ship
9. truck

Plot the first 32 images in the dataset using the same method above.

Question 3

```
from tensorflow.keras.datasets import cifar10

(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()

rows, cols = 4, 8

# Create a figure and axes for the grid
fig, axes = plt.subplots(rows, cols, figsize=(10, 8))

# Iterate through the grid
for i in range(rows):
    for j in range(cols):
        index = i * cols + j
        ax = axes[i, j]

        # Display the image
        ax.imshow(train_images[index], cmap='gray')

        # Display the label on top of the image in red text
        ax.text(0.9, 0.9, str(train_labels[index]), color='red',
                transform=ax.transAxes, fontsize=24,
                ha='center', va='center')

        # Turn off axis labels
        ax.axis('off')

# Adjust spacing and layout

plt.tight_layout()
```

Visualizing Tensors

Aside from visualzing linear functions and images, we can also visualize entire tensors from DL models.

```
# first, let's download an existing model to inspect
model = tf.keras.applications.VGG16(weights='imagenet')

# can then print the summary of what the model is composed of
print(model.summary())
```

Model: "vgg16"

Layer (type)	Output Shape	Param #
input_layer_7 (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(None, 112, 112, 128)	147,584
block2_pool (MaxPooling2D)	(None, 56, 56, 128)	0
block3_conv1 (Conv2D)	(None, 56, 56, 256)	295,168
block3_conv2 (Conv2D)	(None, 56, 56, 256)	590,080
block3_conv3 (Conv2D)	(None, 56, 56, 256)	590,080
block3_pool (MaxPooling2D)	(None, 28, 28, 256)	0
block4_conv1 (Conv2D)	(None, 28, 28, 512)	1,180,160
block4_conv2 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_conv3 (Conv2D)	(None, 28, 28, 512)	2,359,808
block4_pool (MaxPooling2D)	(None, 14, 14, 512)	0
block5_conv1 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv2 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_conv3 (Conv2D)	(None, 14, 14, 512)	2,359,808
block5_pool (MaxPooling2D)	(None, 7, 7, 512)	0
flatten (Flatten)	(None, 25088)	0
fc1 (Dense)	(None, 4096)	102,764,544
fc2 (Dense)	(None, 4096)	16,781,312
predictions (Dense)	(None, 1000)	4,097,000

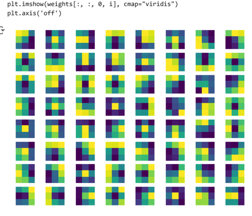
Total params: 138,357,544 (527.79 MB)

```
# we can also print the model layers based on index to better understand the structure
for i,layer in enumerate(model.layers):
  print(f"{i}: {layer}")
```

```
0: <InputLayer name=input_layer_7, built=True>
1: <Conv2D name=block1_conv1, built=True>
2: <Conv2D name=block1_conv2, built=True>
3: <MaxPooling2D name=block1_pool, built=True>
4: <Conv2D name=block2_conv1, built=True>
5: <Conv2D name=block2_conv2, built=True>
6: <MaxPooling2D name=block2_pool, built=True>
7: <Conv2D name=block3_conv1, built=True>
8: <Conv2D name=block3_conv2, built=True>
9: <Conv2D name=block3_conv3, built=True>
10: <MaxPooling2D name=block3_pool, built=True>
11: <Conv2D name=block4_conv1, built=True>
12: <Conv2D name=block4_conv2, built=True>
13: <Conv2D name=block4_conv3, built=True>
14: <MaxPooling2D name=block4_pool, built=True>
15: <Conv2D name=block5_conv1, built=True>
16: <Conv2D name=block5_conv2, built=True>
17: <Conv2D name=block5_conv3, built=True>
18: <MaxPooling2D name=block5_pool, built=True>
19: <Flatten name=flatten, built=True>
20: <Dense name=fc1, built=True>
21: <Dense name=fc2, built=True>
22: <Dense name=predictions, built=True>
```

Not all of these layers contain weights, for example, MaxPooling2D is a stateless operation, and so is Flatten. Conv2D and Dense are the two layer types that can be visualized. That said, let's visualize the filter kernels in the first convoluton layer.

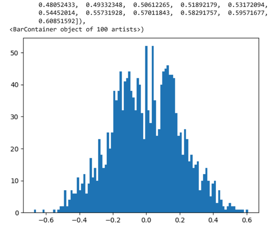
```
# next we can extract som
layer = model.layers[1] # Get the first convolutional layer
weights = layer.get_weights()[0]

n_filters = weights.shape[-1]

for i in range(n_filters):
    plt.subplot(8, 8, i+1)  # Assuming 64 filters, adjust if necessary
    plt.imshow(weights[:, :, 0, i], cmap="viridis")
    plt.axis('off')
```



Aside from visualizing the weights directly, we can also compute and visualize the weight distribution using a histogram.

```
# we can use the mean and var (variance) functions built in to calculate some simple statistics
print(f"weight tensor has mean: {weights.mean()} and variance: {weights.var()}")

# we need to call .flatten() on the tensor so that all the histogram sees them as a 1D array. Then we can plot with 100 bins to get a bit more resolution in the histogram.
plt.hist(weights.flatten(), bins=100)
```

```
weight tensor has mean: -0.0024379086680710316 and variance: 0.04272466152906418
(array([ 1.,  0.,  0.,  0.,  1.,  0.,  0.,  0.,  0.,  1.,  0.,  1.,  2.,
         2.,  7.,  2.,  4.,  7.,  6.,  6., 11.,  7.,  9., 12.,  6.,  9.,
        17., 11., 14., 10., 23., 18., 14., 15., 25., 20., 25., 38., 35.,
        38., 44., 32., 41., 42., 44., 38., 36., 32., 42., 40., 31., 23.,
        52., 32., 28., 52., 35., 24., 26., 40., 44., 45., 46., 43., 43.,
        42., 31., 24., 25., 18., 26., 23., 16., 18., 14., 15., 16.,  7.,
        10., 12., 14., 10.,  5.,  9., 10.,  3.,  7.,  4.,  2.,  1.,  2.,
         3.,  2.,  2.,  1.,  1.,  1.,  1.,  0.,  1.]),
 array([-0.67140007, -0.65860093, -0.64580172, -0.63300258, -0.62020344,
        -0.60740429, -0.59460509, -0.58180594, -0.5690068 , -0.55620766,
        -0.54340845, -0.53060931, -0.51781017, -0.50501096, -0.49221182,
        -0.47941267, -0.4666135 , -0.45381436, -0.44101518, -0.42821604,
        -0.41541687, -0.40261772, -0.38981855, -0.37701941, -0.36422023,
        -0.35142106, -0.33862191, -0.32582274, -0.3130236 , -0.30022442,
        -0.28742528, -0.27462611, -0.26182696, -0.24902779, -0.23622863,
        -0.22342947, -0.21063031, -0.19783115, -0.185032  , -0.17223284,
        -0.15943368, -0.14663452, -0.13383536, -0.12103619, -0.10823704,
        -0.09543788, -0.08263872, -0.06983955, -0.0570404 , -0.04424123,
        -0.03144208, -0.01864292, -0.00584376,  0.0069554 ,  0.01975456,
         0.03255372,  0.04535288,  0.05815204,  0.0709512 ,  0.08375036,
         0.09654953,  0.10934868,  0.12214784,  0.134947  ,  0.14774616,
         0.16054532,  0.17334448,  0.18614364,  0.1989428 ,  0.21174197,
         0.22454113,  0.23734029,  0.25013945,  0.26293859,  0.27573776,
         0.28853691,  0.30133608,  0.31413525,  0.3269344 ,  0.33973357,
         0.35253271,  0.36533189,  0.37813103,  0.39093021,  0.40372935,
         0.41652852,  0.42932767,  0.44212684,  0.45492601,  0.46772516,
         0.48052433,  0.49332348,  0.50612265,  0.51892179,  0.53172094,
         0.54452014,  0.55731928,  0.57011843,  0.58291757,  0.59571677,
         0.60851592]),
 <BarContainer object of 100 artists>)
```

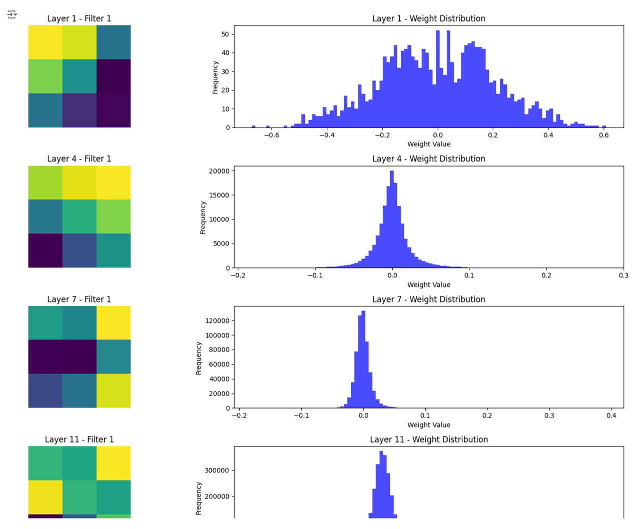


Look through the other weight tensors in the network and note any patterns that can be observed. Plot some examples in a subplot grid (include at least 4 plots). You can also overplot on the same subplot if you find that helpful for visualization.

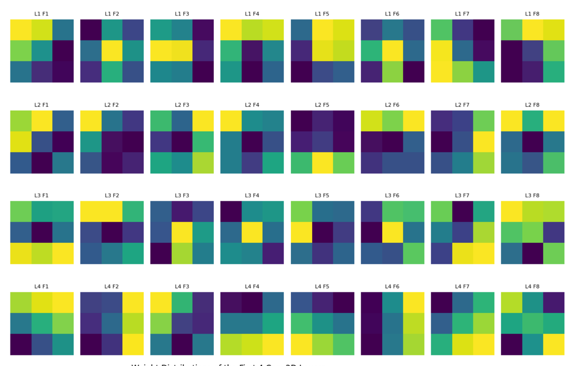
Question 4

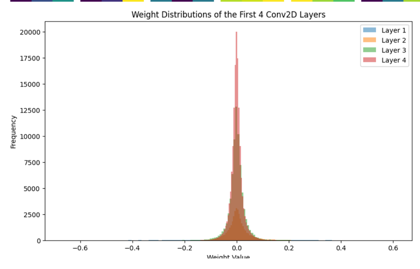
```
# enter plot code below
import matplotlib.pyplot as plt
```

```
import tensorflow as tf
import numpy as np

# Load the VGG16 model
model = tf.keras.applications.VGG16(weights='imagenet')

# Extract Conv2D layers' weights
conv_layers = [layer for layer in model.layers if isinstance(layer, tf.keras.layers.Conv2D)]
weights_list = [layer.get_weights()[0] for layer in conv_layers]

# Function to visualize filters and their distributions
def visualize_weights(weights_list, layer_indices):
    plt.figure(figsize=(16, 12))

    for idx, layer_idx in enumerate(layer_indices):
        weights = weights_list[layer_idx]
        n_filters = weights.shape[-1]

        # Visualize the first filter of the layer
        plt.subplot(len(layer_indices), 2, 2 * idx + 1)
        plt.imshow(weights[:, :, 0, 0], cmap="viridis")
        plt.axis('off')
        plt.title(f"Layer {layer_idx+1} - Filter 1", fontsize=12)

        # Plot weight distribution
        plt.subplot(len(layer_indices), 2, 2 * idx + 2)
        plt.hist(weights.flatten(), bins=100, alpha=0.7, color='blue')
        plt.xlabel("Weight Value")
        plt.ylabel("Frequency")
        plt.title(f"Layer {layer_idx+1} - Weight Distribution", fontsize=12)

    plt.tight_layout()
    plt.show()

# Visualize selected layers
visualize_weights(weights_list, [0, 3, 6, 10])  # Adjust layer indices as needed
```




```
import matplotlib.pyplot as plt
import tensorflow as tf

# Load the VGG16 model
model = tf.keras.applications.VGG16(weights='imagenet')

# Print model summary (optional for inspection)
# print(model.summary())

# Get weights of specific layers (e.g., Conv2D layers)
conv_layers = [layer for layer in model.layers if isinstance(layer, tf.keras.layers.Conv2D)]
weights_list = [layer.get_weights()[0] for layer in conv_layers]

# Visualize the weights of the first four Conv2D layers
plt.figure(figsize=(12, 8))
for i in range(4):
    weights = weights_list[i]
    n_filters = weights.shape[-1]

    # Visualize the first 8 filters of the layer
    for j in range(8):
        plt.subplot(4, 8, i * 8 + j + 1)
        plt.imshow(weights[:, :, 0, j], cmap="viridis")
        plt.axis('off')
        plt.title(f"L{i+1} F{j+1}", fontsize=8)

plt.tight_layout()
plt.show()

# Analyze weight distributions
plt.figure(figsize=(10, 6))
for i in range(4):
    weights = weights_list[i]
    plt.hist(weights.flatten(), bins=100, alpha=0.5, label=f"Layer {i+1}")
plt.xlabel("Weight Value")
plt.ylabel("Frequency")
plt.title("Weight Distributions of the First 4 Conv2D Layers")
plt.legend()
plt.show()
```

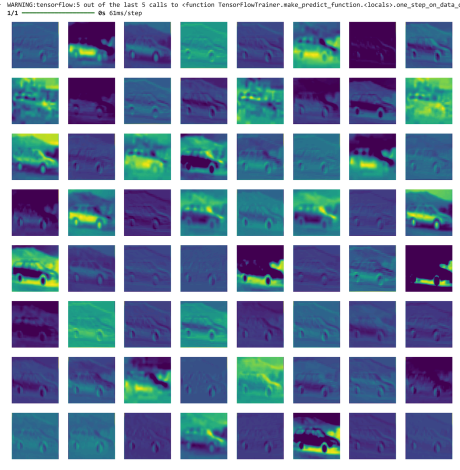




We can also visualize the activations within the network, this is done by applying a forward pass with a data input, and extracting the intermediate result. Below is an example output from the first convolution layer.

```
# Resize and normalize the images to be suitable for VGG16
train_images_resized = tf.image.resize(train_images[4:5], [224, 224])

# Normalize the pixel values to [0,1]
train_images_resized = train_images_resized / 255.0

layer = model.layers[1] # Get the first convolutional layer
intermediate_layer_model = tf.keras.models.Model(inputs=model.input, outputs=layer.output)
activation = intermediate_layer_model.predict(train_images_resized)

# get the feature count from the activations
n_features = activation.shape[-1]

# Set the figure size
plt.figure(figsize=(12, 12))

for i in range(n_features):
    plt.subplot(8, 8, i+1)  # Assuming 64 features, adjust if necessary
    plt.imshow(activation[0, :, :, i], cmap="viridis")
    plt.axis('off')
```

```
WARNING:tensorflow:5 out of the last 5 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x7a0e7e01a050> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to  https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more details.
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 61ms/step
```

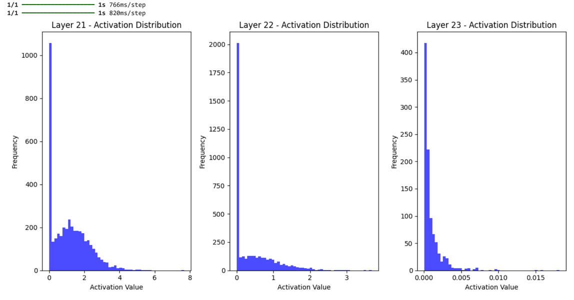


Using the above code for the forward pass, and the layer indices, plot the activation distributions for the final three dense layers.

Question 5

```
# Dense layer plot code below
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np

# Load the VGG16 model
model = tf.keras.applications.VGG16(weights='imagenet')

# Get the final three Dense layers
dense_layers = [layer for layer in model.layers if isinstance(layer, tf.keras.layers.Dense)]

# Resize and normalize an image to pass through the model
(train_images, _), _ = tf.keras.datasets.cifar10.load_data()
train_images_resized = tf.image.resize(train_images[4:5], [224, 224])
train_images_resized = train_images_resized / 255.0

# Forward pass through the model to get activations of the Dense layers
intermediate_outputs = []
for dense_layer in dense_layers:
    intermediate_layer_model = tf.keras.models.Model(inputs=model.input, outputs=dense_layer.output)
    activation = intermediate_layer_model.predict(train_images_resized)
    intermediate_outputs.append(activation.flatten())

# Plot the activation distributions
plt.figure(figsize=(12, 6))
for i, activations in enumerate(intermediate_outputs):
    plt.subplot(1, 3, i + 1)
    plt.hist(activations, bins=50, alpha=0.7, color='blue')
    plt.title(f"Layer {model.layers.index(dense_layers[i]) + 1} - Activation Distribution")
    plt.xlabel("Activation Value")
    plt.ylabel("Frequency")

plt.tight_layout()
plt.show()
```

```
WARNING:tensorflow:6 out of the last 6 calls to <function TensorFlowTrainer.make_predict_function.<locals>.one_step_on_data_distributed at 0x7a0e7e54c310> triggered tf.function retracing. Tracing is expensive and the excessive number of tracings could be due to (1) creating @tf.function repeatedly in a loop, (2) passing tensors with different shapes, (3) passing Python objects instead of tensors. For (1), please define your @tf.function outside of the loop. For (2), @tf.function has reduce_retracing=True option that can avoid unnecessary retracing. For (3), please refer to  https://www.tensorflow.org/guide/function#controlling_retracing and https://www.tensorflow.org/api_docs/python/tf/function for  more details.
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 777ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 766ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 820ms/step
```



What do you notice about the distributions, and how they compare to those of the weight tensors?

Question 6

Activation distributions are dynamic and vary based on input data, often showing broader ranges and sparsity, particularly in layers using ReLU, where many values are zero. In contrast, weight distributions are more symmetric, centered around zero, and have narrower ranges, determined by initialization and regularization. While activations evolve throughout the network to extract features, weights remain static, serving as optimized parameters to guide the transformation of activations. This difference highlights how weights provide a stable framework, while activations reflect the model's real-time processing of input data.