# 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
```

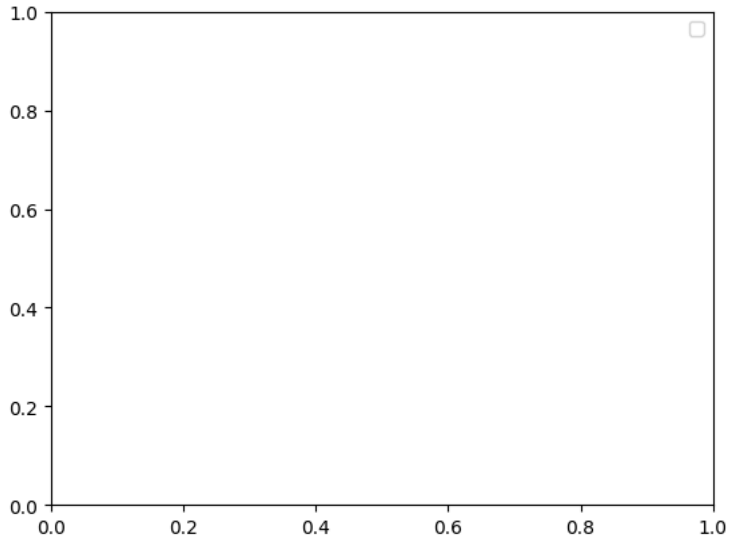
## Visualizations

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
# to use math.sin, we could have used a list comprehension instead: y2 = [math.sin(xi) for xi in x]


plt.legend()                        # have matplotlib show the label on the plot
```

```
<ipython-input-3-50d218eb0245>:8: UserWarning: No artists with labels found to put in legend.  Note that artists whose label start w
  plt.legend()                        # have matplotlib show the label on the plot
<matplotlib.legend.Legend at 0x78738dfac2d0>
```




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: $x \cdot \frac{1}{2}\left(1 + \tanh\left(\frac{x}{\sqrt{2}}\right)\right)$

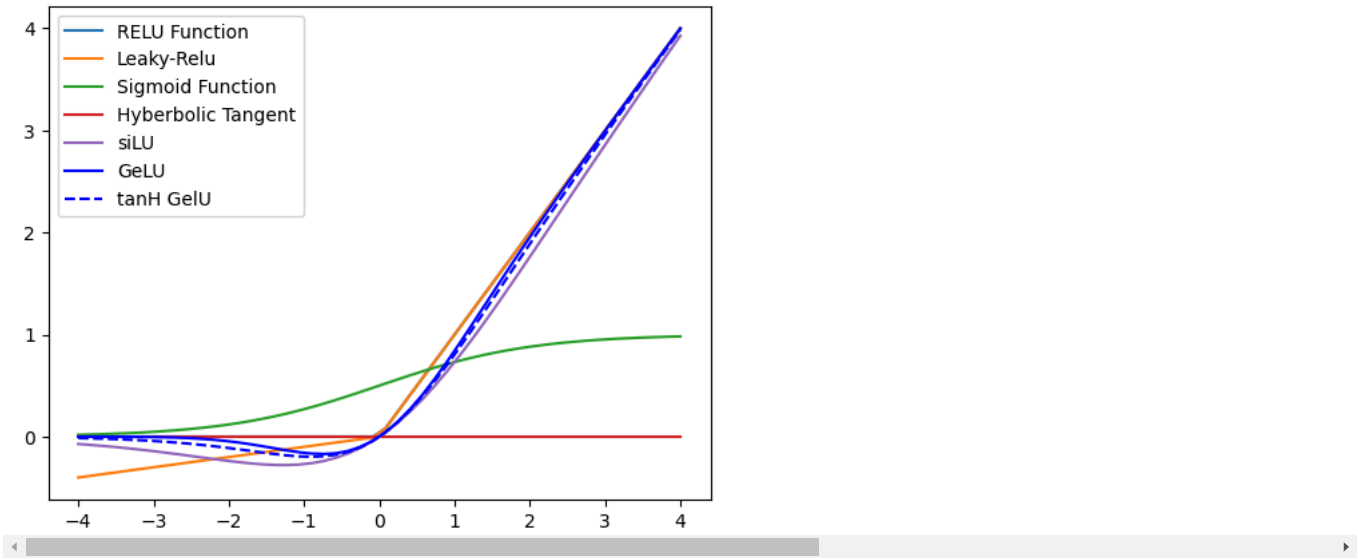
Plot the GELU and tanh GELU using the same color, but with tanh using a dashed line (tanh is a common approximation as the error-function is computationally expensive to compute). You may also need to adjust the legend to make it easier to read. I recommend using ChatGPT to help

find the formatting options here.

**Question 1**

```
x = np.linspace(-4, 4, 50) # create a linear spacing from x = -4.0 to 4.0 with 50 steps
# create and plot the functions below
y3 = np.maximum(x,0)
y4 = np.maximum(x/10,x)
y5 = (np.exp(x))/(1+np.exp(x))  #sigmoid
y6 = 1j*np.tan(x)
y7 = x*y5
y8 = 0.5 * x * (1 + erf(x / np.sqrt(2)))
y9 = 0.5 * x * (1 + tf.tanh(x / np.sqrt(2)))

plt.plot(x,y3,label="RELU Function")
plt.plot(x,y4,label="Leaky-Relu")
plt.plot(x,y5,label="Sigmoid Function")
plt.plot(x,y6,label="Hyberbolic Tangent")
plt.plot(x,y7,label="siLU")
plt.plot(x, y8, color="blue", label="GeLU")
plt.plot(x, y9, color="blue", linestyle='--',label="tanH GelU")

plt.legend()                        # have matplotlib show the label on the plot
```

```
/usr/local/lib/python3.11/dist-packages/matplotlib/cbook.py:1709: ComplexWarning: Casting complex values to real discards the imagir
  return math.isfinite(val)
/usr/local/lib/python3.11/dist-packages/matplotlib/cbook.py:1345: ComplexWarning: Casting complex values to real discards the imagir
  return np.asarray(x, float)
<matplotlib.legend.Legend at 0x78738be3e750>
```




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

1. Which activation function is the least computationally expensive to compute?

RELU Function, since from a hardware point-of-view, it will require only Comparator for computing the max(x,0) function, that can be implemented using subtractors, whereas remaining activation functions will require more than subtractors and additional hardware.

2. Are there better choices to ensure more stable training? What downfalls do you think it may have?

Leaky ReLU is a strong, practical choice for stable training. It addresses the dying ReLU problem by maintaining a small gradient for negative inputs. It's computationally efficient and avoids the vanishing gradient issue common with sigmoid or tanh. It works well across various network depths and architectures without requiring significant hyperparameter tuning.

Downfalls: The slope for negative inputs might need tuning, which introduces hyperparameter complexity.

3. Are there any cases where you would not want to use either activation function?

ReLU and Leaky ReLU are not smooth; their piecewise linear nature can make optimization less stable for tasks requiring fine-grained adjustments (e.g., in complex generative models or continuous control tasks). Furtheromre, while ReLU avoids the vanishing gradient for positive inputs, it outputs zero for all negative inputs, potentially causing dead neurons. Leaky ReLU's linear behavior for both positive and negative inputs can amplify outliers in certain datasets, leading to instability. For networks requiring zero-centered activations, neither function is zero-centered, which can cause biases to accumulate in the network and slow down training. For tasks with Very Large or Very Small Input values, they don't naturally normalize activations, which can result in very large or very small values propagating through the network.

**Question 2**

Double-click (or enter) to edit

## Visualizing 2D data

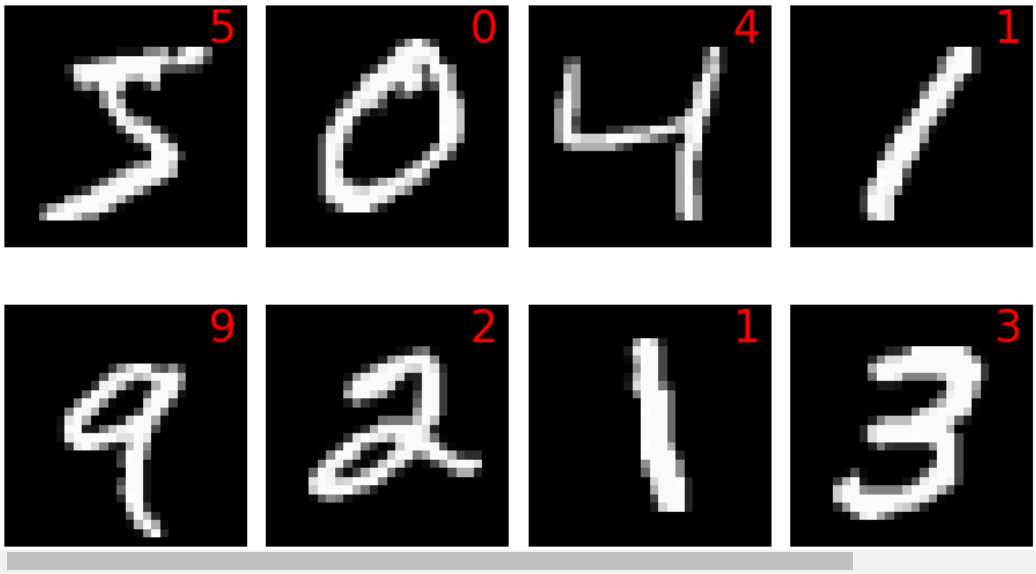
In many cases, we also want the ability to visualize multi-dimensional data such as images. To do so, matplotlib has the imshow method, which can visualize single channel data with a heatmap, or RGB data with color.

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()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11490434/11490434 ━━━━━━━━━━━━━━━━━━━━ 0s 0us/step
```




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()

# add your plotting code below
# Define the grid dimensions
rows, cols = 4, 8

# 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()
```

```
Downloading data from https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
170498071/170498071 ━━━━━━━━━━━━━━━━━━━━ 2s 0us/step
```


## 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())
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels.h5
553467096/553467096 ━━━━━━━━━━━━━━━━━━━━ 5s 0us/step
Model: "vgg16"
```

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_layer (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 |

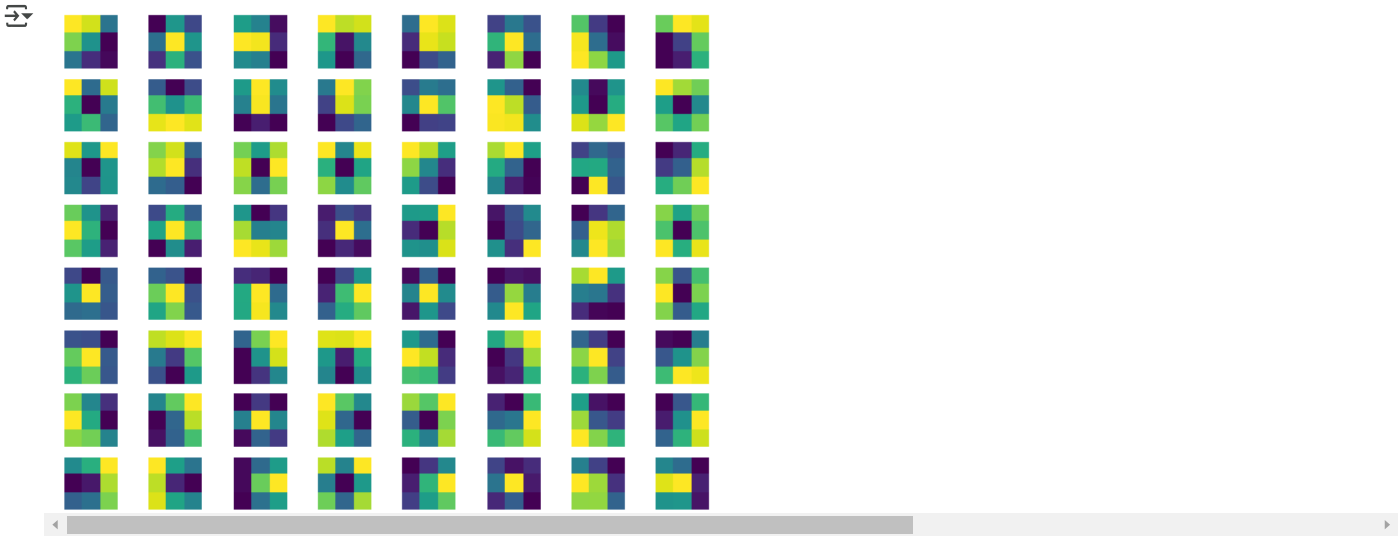
```
Total params: 138,357,544 (527.79 MB)
Trainable params: 138,357,544 (527.79 MB)
Non-trainable params: 0 (0.00 B)
None
```

```
# 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, 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.

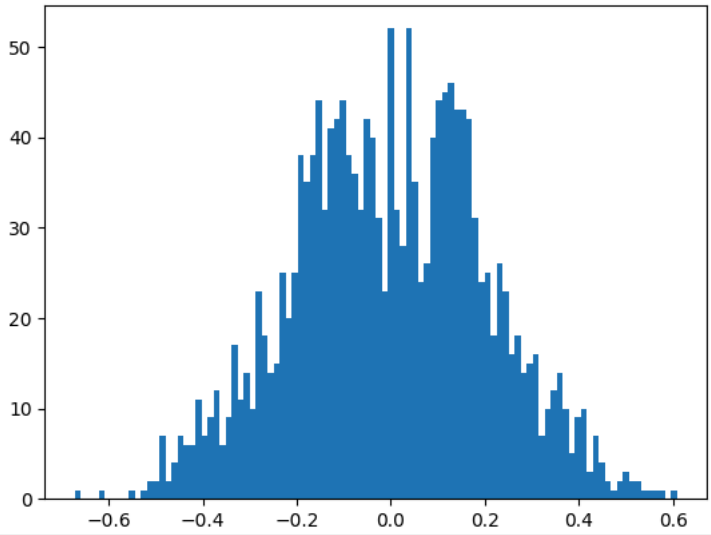
```
# next we can extract som
layer = model.layers[1] # Get the first convolutional layer
weights = layer.get_weights()[0]

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
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**

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

# Load pre-trained VGG16 model
model = tf.keras.applications.VGG16(weights='imagenet')

# Print model summary and layers
print(model.summary())
for i, layer in enumerate(model.layers):
    print(f"{i}: {layer}")

# Extract and visualize first convolutional layer weights
layer = model.layers[1]  # First Conv layer
weights = layer.get_weights()

if weights:  # Ensure the layer has weights
    weights = weights[0]  # Get weight tensor
    n_filters = weights.shape[-1]  # Number of filters

    plt.figure(figsize=(10, 10))
    for i in range(min(64, n_filters)):  # Show up to 64 filters
        plt.subplot(8, 8, i + 1)
        plt.imshow(weights[:, :, 0, i], cmap="viridis")
        plt.axis('off')
```

```
            plt.axis('off')
        plt.suptitle("First Convolutional Layer Filters")
        plt.show()

        # Compute basic statistics
        print(f"Weight tensor has mean: {weights.mean()} and variance: {weights.var()}")

        # Plot histogram of weights
        plt.figure(figsize=(8, 5))
        plt.hist(weights.flatten(), bins=100, color='blue', alpha=0.7)
        plt.title("Weight Distribution (Layer 1)")
        plt.xlabel("Weight Value")
        plt.ylabel("Frequency")
        plt.show()

    # Explore and plot other convolutional layers
    layers_to_inspect = [1, 3, 5, 7]  # Example layer indices (should be Conv layers)
    plt.figure(figsize=(12, 8))

    for idx, layer_idx in enumerate(layers_to_inspect):
        layer = model.layers[layer_idx]
        weights = layer.get_weights()

        if weights:  # Check if the layer has weights
            weights = weights[0]

            plt.subplot(2, 2, idx + 1)
            plt.hist(weights.flatten(), bins=100, alpha=0.7, label=f'Layer {layer_idx}')
            plt.xlabel("Weight Value")
            plt.ylabel("Frequency")
            plt.legend()

    plt.suptitle("Weight Distributions Across Layers")
    plt.show()
```

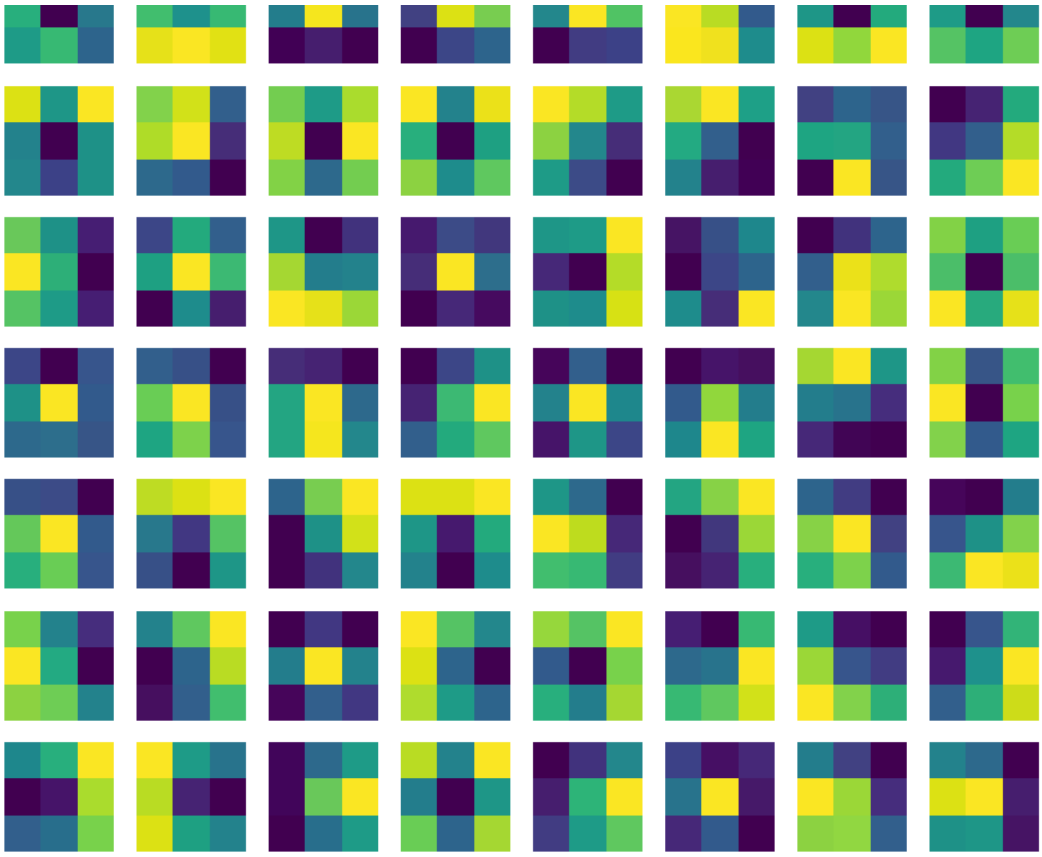
Model: "vgg16"

| Layer (type) | Output Shape | Param # |
|---|---|---|
| input_layer_2 (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)
**Trainable params:** 138,357,544 (527.79 MB)
**Non-trainable params:** 0 (0.00 B)

```
None
0: <InputLayer name=input_layer_2, 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>
```

First Convolutional Layer Filters



```
Weight tensor has mean: -0.0024379086680710316 and variance: 0.04272466152906418
```