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
```

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]

# 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
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-6-3a4f87d7b1a3> in <cell line: 1>()
----> 1 x =np.linspace(-5, 5, 50) # create a linear spacing from x = -5.0 to 5.0 with 50 steps
      2
      3 y1 = x**2      # create a series of points {y1}, which corresponds to the function f(x) = y^2
      4 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
      5 # to use math.sin, we could have used a list comprehension instead: y2 = [math.sin(xi) for xi in x]

NameError: name 'np' is not defined
```

Next steps: Explain error

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 $x \in [-4, 4]$:

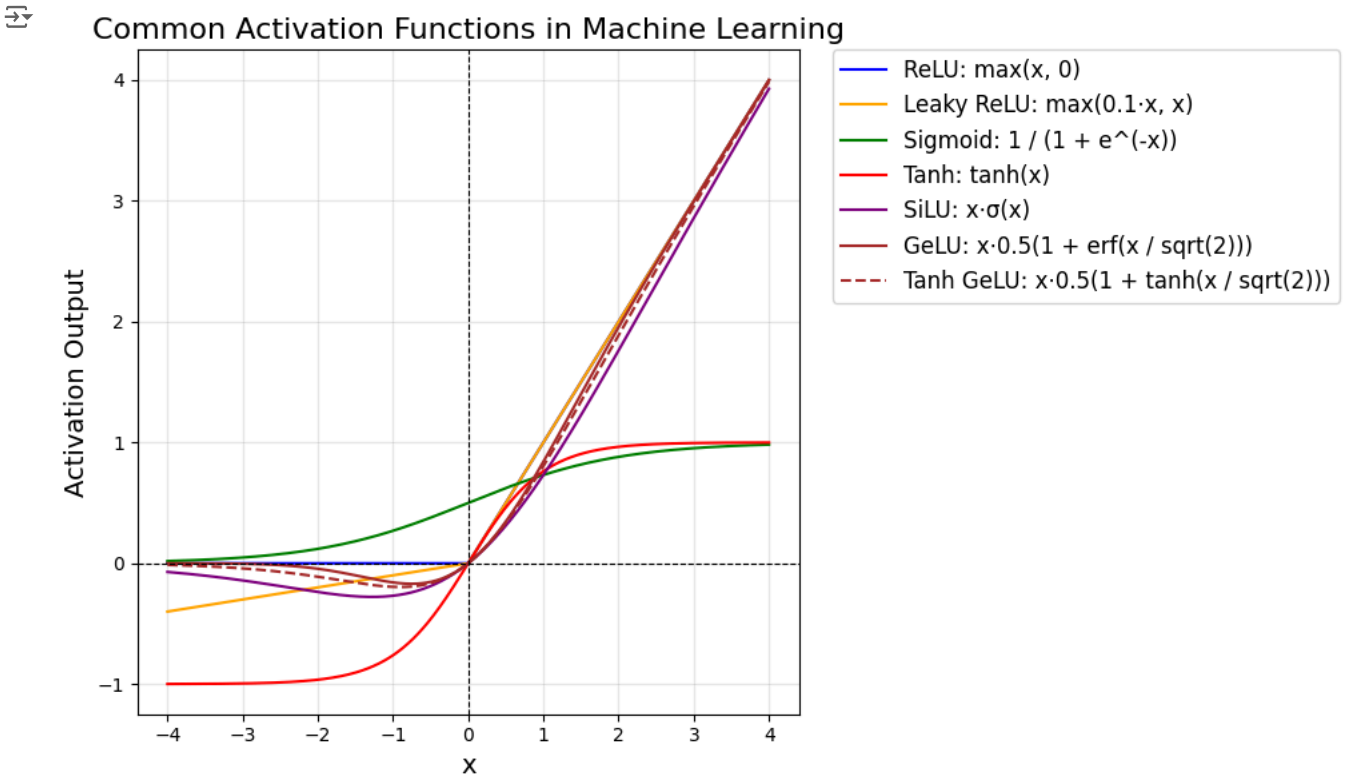
- ReLU: $max(x, 0)$
- Leaky-ReLU: $max(0.1 \cdot x, x)$
- Sigmoid: $\sigma(x) = 1/(1 + e^{-x})$
- Hyperbolic Tangent: $\tanh(x) = (e^{x} - e^{-x})/(e^{x} + e^{-x})$
- SiLU: $x \cdot \sigma(x)$
- GeLU: $x \cdot \frac{1}{2}\left(1 + \text{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$
- tanh GELU: $x \cdot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

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

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

# Define the range of x values
x = np.linspace(-4, 4, 500)
```

```
# Define the activation functions
relu = np.maximum(x, 0)
leaky_relu = np.maximum(0.1 * x, x)
sigmoid = 1 / (1 + np.exp(-x))
tanh = np.tanh(x)
silu = x * sigmoid
gelu = x * 0.5 * (1 + erf(x / (2**0.5)))
tanh_gelu = x * 0.5 * (1 + np.tanh(x / (2**0.5)))

# Create the plot
plt.figure(figsize=(10, 6))

# Plot each activation function
plt.plot(x, relu, label="ReLU: max(x, 0)", color="blue")
plt.plot(x, leaky_relu, label="Leaky ReLU: max(0.1·x, x)", color="orange")
plt.plot(x, sigmoid, label="Sigmoid: 1 / (1 + e^(-x))", color="green")
plt.plot(x, tanh, label="Tanh: tanh(x)", color="red")
plt.plot(x, silu, label="SiLU: x·σ(x)", color="purple")
plt.plot(x, gelu, label="GeLU: x·0.5(1 + erf(x / sqrt(2)))", color="brown")
plt.plot(x, tanh_gelu, label="Tanh GeLU: x·0.5(1 + tanh(x / sqrt(2)))", color="brown", linestyle="--")

# Customize the plot
plt.title("Common Activation Functions in Machine Learning", fontsize=16)
plt.xlabel("x", fontsize=14)
plt.ylabel("Activation Output", fontsize=14)
plt.axhline(0, color="black", linestyle="--", linewidth=0.8)  # Add a horizontal line at y=0
plt.axvline(0, color="black", linestyle="--", linewidth=0.8)  # Add a vertical line at x=0
plt.grid(alpha=0.3)

# Adjust the legend for readability
plt.legend(fontsize=12, loc="upper left", bbox_to_anchor=(1.05, 1), borderaxespad=0.)
plt.tight_layout()  # Adjust layout to fit the legend
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

Double-click (or enter) to edit

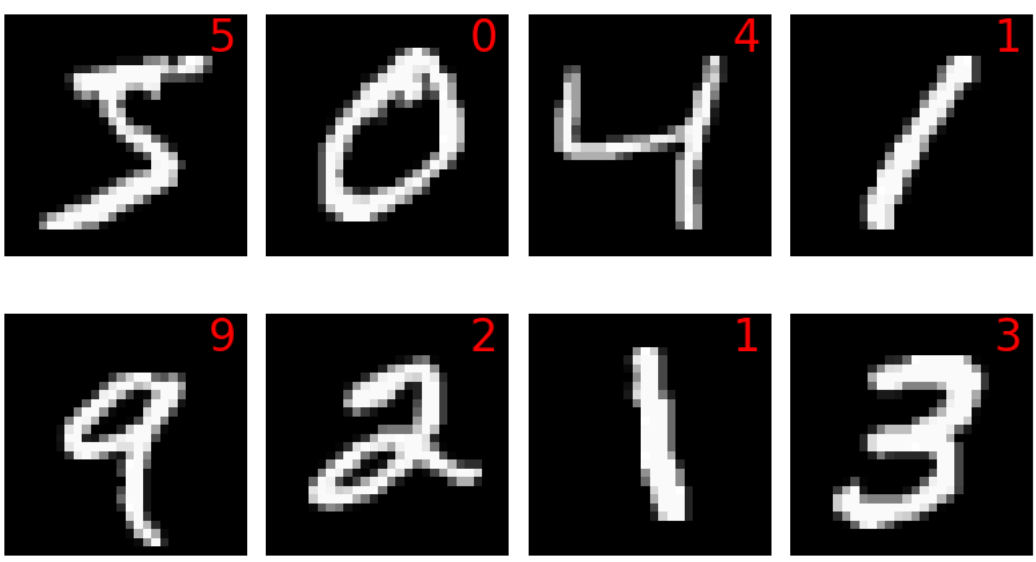
Visualizing 2D data

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.

```
import matplotlib.pyplot as plt  # Import matplotlib.pyplot
from tensorflow.keras.datasets import mnist

# Load MNIST dataset
(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()  # Adjust the spacing
plt.show()  # Display the plot
```



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

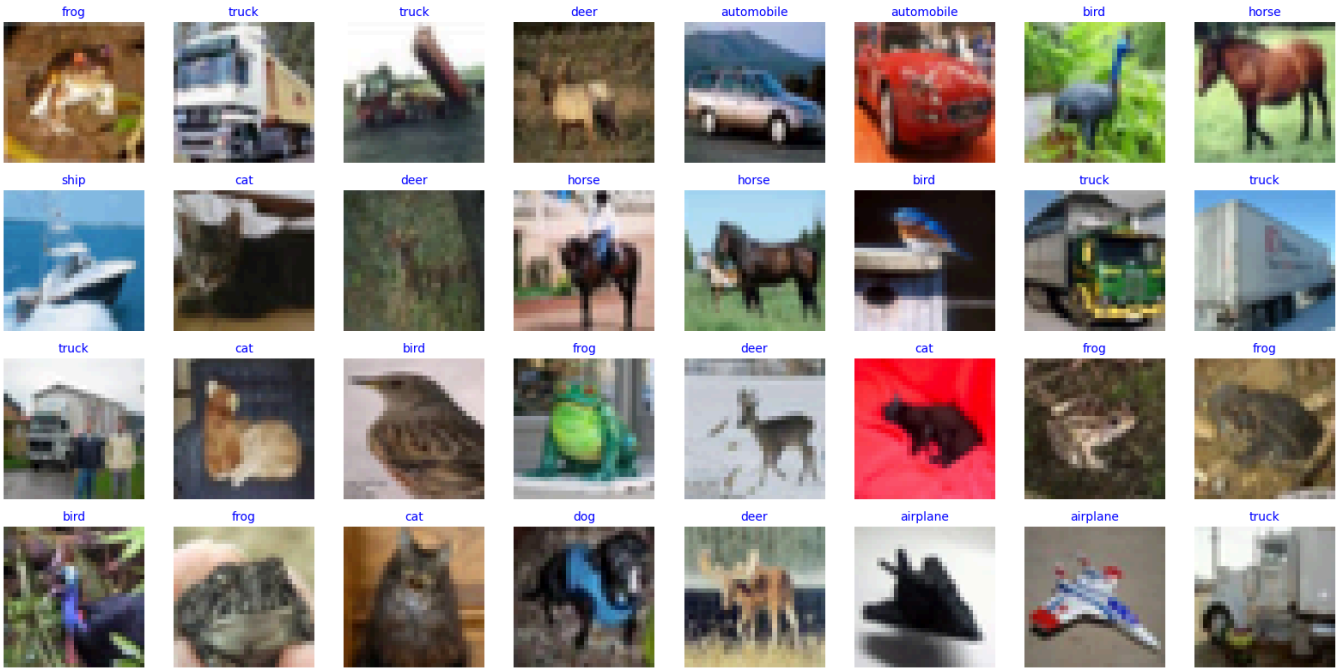
7. horse
8. ship
9. truck

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

Question 3

```
import matplotlib.pyplot as plt
from tensorflow.keras.datasets import cifar10

# Load the CIFAR-10 dataset
(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()

# Define the CIFAR-10 class labels
class_labels = [
    "airplane", "automobile", "bird", "cat", "deer",
    "dog", "frog", "horse", "ship", "truck"
]

# Define the grid dimensions (4 rows x 8 columns = 32 images)
rows, cols = 4, 8

# Create a figure and axes for the grid
fig, axes = plt.subplots(rows, cols, figsize=(16, 8))

# Iterate through the grid to display the first 32 images
for i in range(rows):
    for j in range(cols):
        index = i * cols + j  # Calculate image index
        ax = axes[i, j]

        # Display the image
        ax.imshow(train_images[index])

        # Display the class label on top of the image
        ax.set_title(class_labels[train_labels[index][0]], fontsize=10, color='blue')

        # Turn off axis ticks for clarity
        ax.axis('off')

# Adjust spacing between images
plt.tight_layout()
plt.show()
```

```
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())
```

```
# 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}")
```

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.

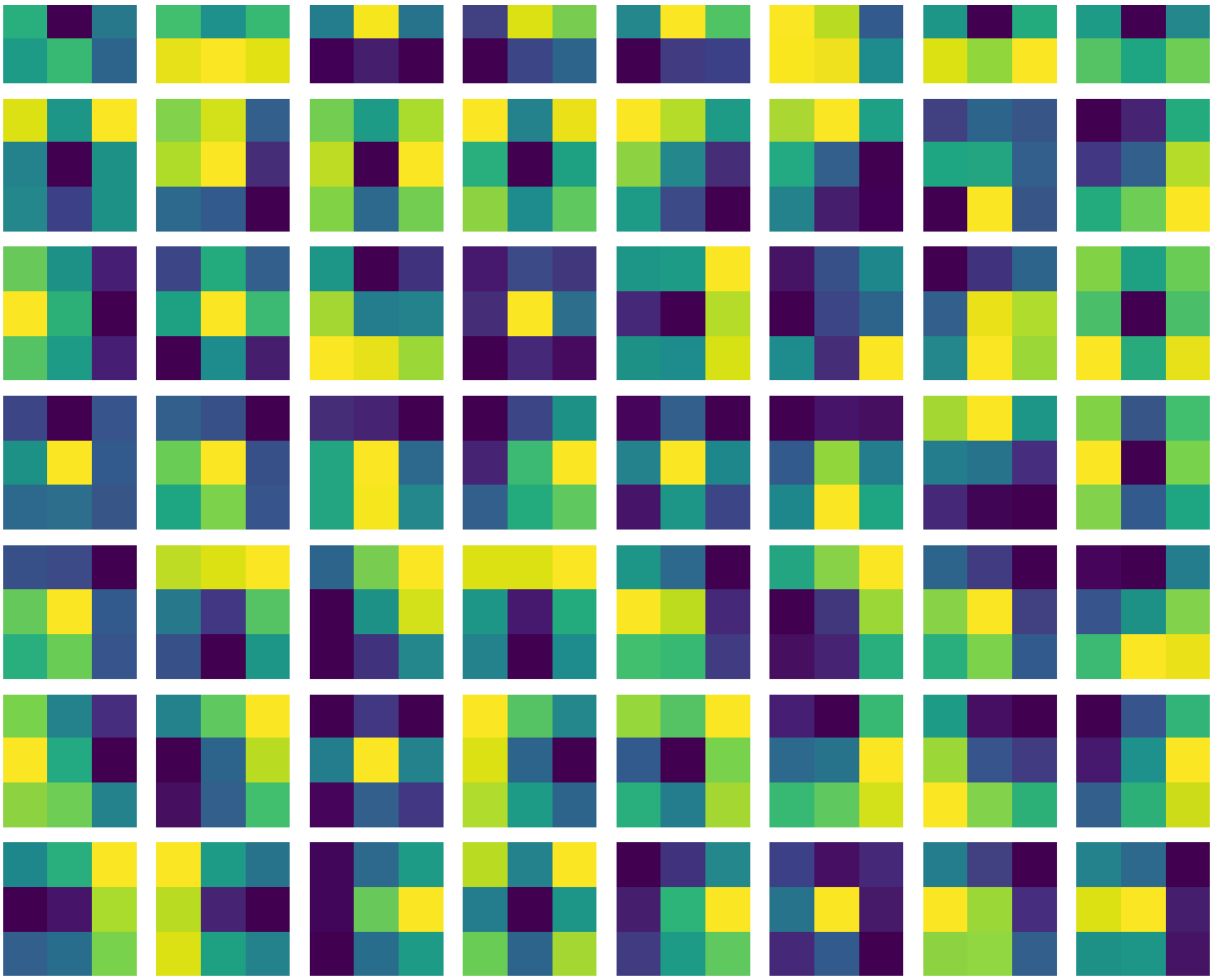
```
import tensorflow as tf
import matplotlib.pyplot as plt

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

# Print model summary to understand the structure
print(model.summary())

# Iterate over model layers to inspect them
for i, layer in enumerate(model.layers):
    print(f"{i}: {layer}")

# Extract the first convolutional layer (index 1 for VGG16)
conv_layer = model.layers[1]  # Conv2D layer
weights = conv_layer.get_weights()[0]  # Get the weights (kernels)

# Print the shape of the weights tensor
print(f"Weights shape: {weights.shape}")  # (filter_height, filter_width, input_channels, num_filters)
```

```
# Visualize the first 64 filter kernels
n_filters = weights.shape[-1]  # Number of filters

# Create a grid to visualize the kernels
plt.figure(figsize=(12, 12))
for i in range(min(n_filters, 64)):  # Display up to 64 filters
    plt.subplot(8, 8, i + 1)  # Arrange in an 8x8 grid
    plt.imshow(weights[:, :, 0, i], cmap="viridis")  # Visualize the kernel for the first channel
    plt.axis('off')
plt.suptitle("Filter Kernels from the First Conv2D Layer", fontsize=16)
plt.tight_layout()
plt.show()

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

# Flatten the weights for histogram visualization
weights_flat = weights.flatten()

# Plot the histogram of the weights
plt.figure(figsize=(8, 6))
plt.hist(weights_flat, bins=100, color='blue', alpha=0.7, label='Weights')
plt.title("Weight Distribution of First Conv2D Layer")
plt.xlabel("Weight Value")
plt.ylabel("Frequency")
plt.legend()
plt.grid(alpha=0.3)
plt.show()

# Visualize weight distributions for other layers
layers_to_visualize = [1, 3, 5, 7]  # Example Conv2D layers (indices may vary by model)
plt.figure(figsize=(12, 8))
for i, layer_idx in enumerate(layers_to_visualize):
    layer = model.layers[layer_idx]
    if isinstance(layer, tf.keras.layers.Conv2D):  # Ensure the layer is Conv2D
        weights = layer.get_weights()[0]
        plt.subplot(2, 2, i + 1)
        plt.hist(weights.flatten(), bins=100, alpha=0.7, label=f"Layer {layer_idx}")
        plt.title(f"Layer {layer_idx} Weight Distribution")
        plt.xlabel("Weight Value")
        plt.ylabel("Frequency")
        plt.legend()
        plt.grid(alpha=0.3)
plt.suptitle("Weight Distributions of Selected Conv2D Layers", fontsize=16)
plt.tight_layout()
plt.show()
```

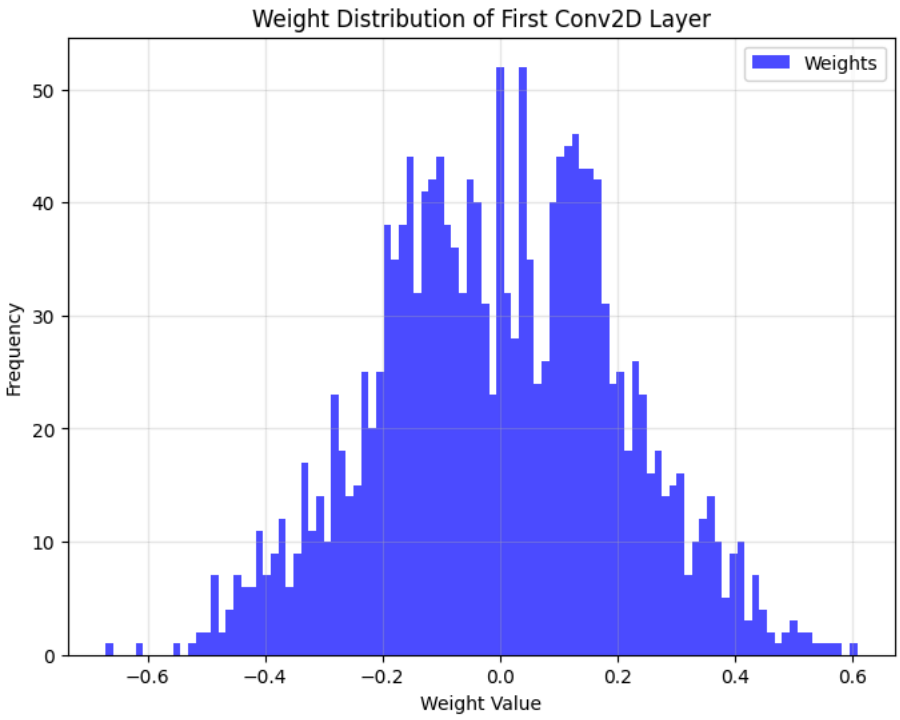
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kernels
553467096/553467096 ━━━━━━━━━━━━━━━━━━━━ **3s** 0us/step
Model: "vgg16"

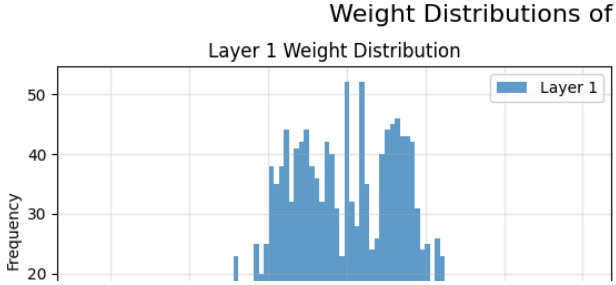
Layer (type)	Output Shape	Param #
input_layer (InputLayer)	(None, 224, 224, 3)	0
block1_conv1 (Conv2D)	(None, 224, 224, 64)	1,792
block1_conv2 (Conv2D)	(None, 224, 224, 64)	36,928
block1_pool (MaxPooling2D)	(None, 112, 112, 64)	0
block2_conv1 (Conv2D)	(None, 112, 112, 128)	73,856
block2_conv2 (Conv2D)	(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

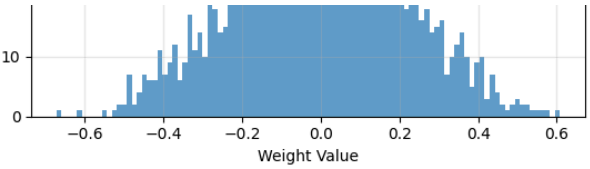
Total params: 138,357,544 (527.79 MB)
Trainable params: 138,357,544 (527.79 MB)
Non-trainable params: 0 (0.00 B)
None
0: <InputLayer name=input_layer, built=True>
1: <Conv2D name=block1_conv1, built=True>
2: <Conv2D name=block1_conv2, built=True>
3: <MaxPooling2D name=block1_pool, built=True>
4: <Conv2D 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>
Weights shape: (3, 3, 3, 64)

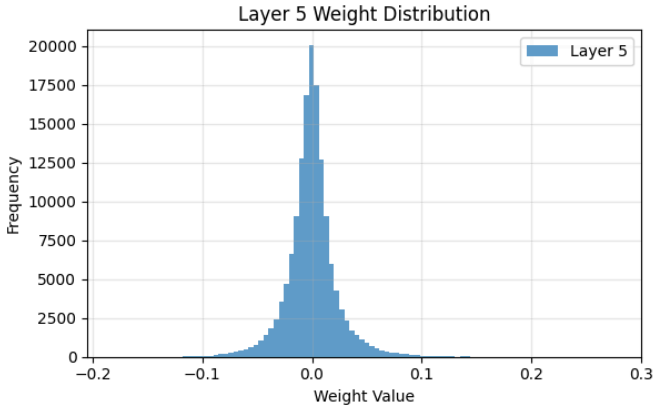


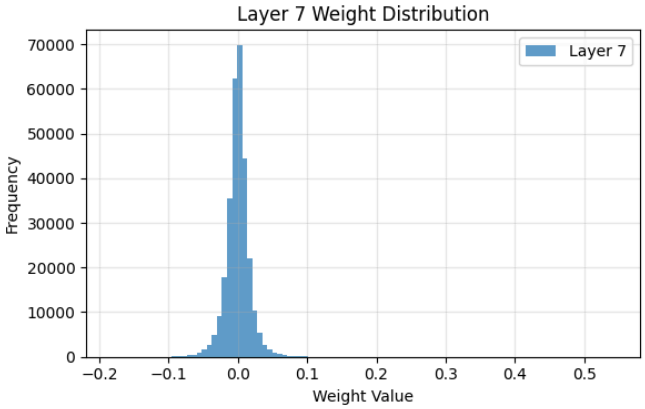
```
Weight tensor has mean: -0.0024379086680710316 and variance: 0.04272466152906418
```






10
0
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
Weight Value


Layer 5 Weight Distribution
Layer 5
Frequency
20000
17500
15000
12500
10000
7500
5000
2500
0
−0.2
−0.1
0.0
0.1
0.2
0.3
Weight Value


Layer 7 Weight Distribution
Layer 7
Frequency
70000
60000
50000
40000
30000
20000
10000
0
−0.2
−0.1
0.0
0.1
0.2
0.3
0.4
0.5
Weight Value