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
```

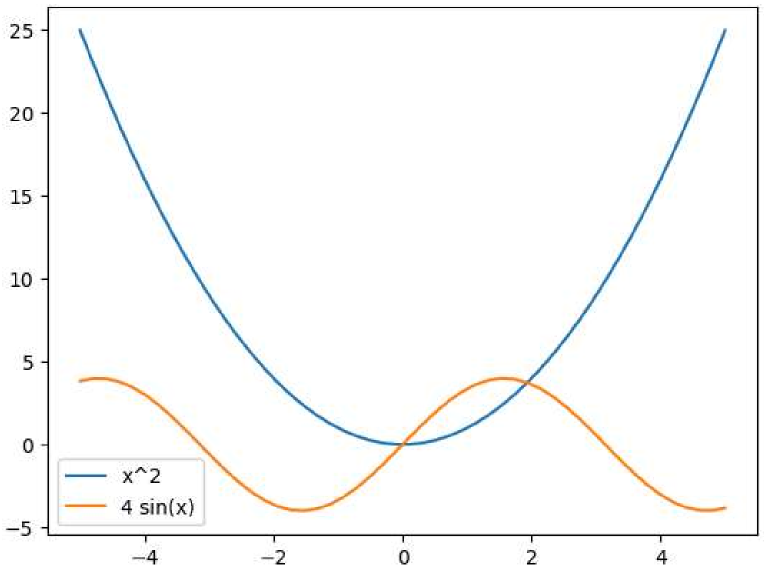
Visualizations

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.
# 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 0x7c8c24c93490>
```



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

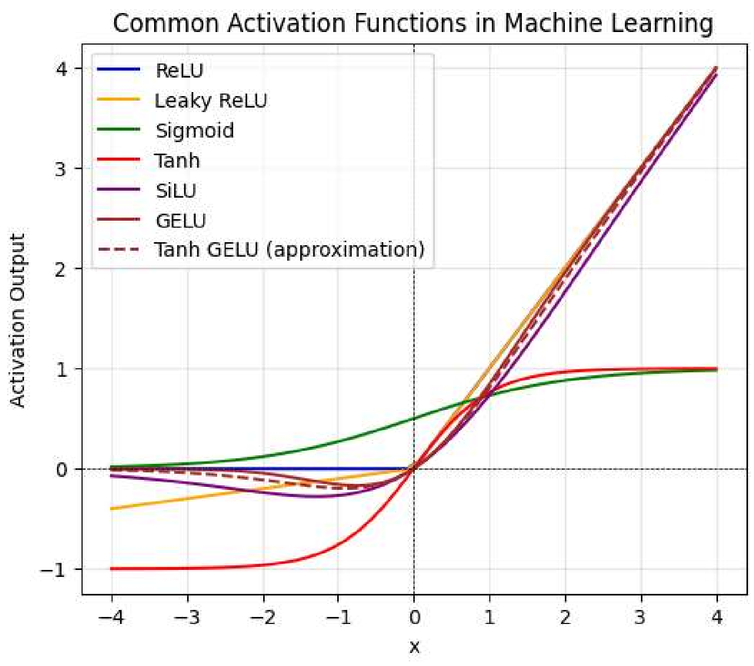
- ReLU: $max(x, 0)$
- Leaky-ReLU: $max(0.1 \cdot x, x)$
- Sigmoid: $\sigma(x) = 1/(1 + e^{-x})$
- Hyperbolic Tangent: $\tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$
- SiLU: $x \cdot \sigma(x)$

- GeLU: $x \cdot \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x}{\sqrt{2}}\right)\right)$
- 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 x values
x = np.linspace(-4, 4, 50)

# Define 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 / np.sqrt(2)))
tanh_gelu = x * 0.5 * (1 + np.tanh(x / np.sqrt(2)))

# Plot the activation functions
plt.figure(figsize=(6, 5))
plt.plot(x, relu, label='ReLU', color='blue')
plt.plot(x, leaky_relu, label='Leaky ReLU', color='orange')
plt.plot(x, sigmoid, label='Sigmoid', color='green')
plt.plot(x, tanh, label='Tanh', color='red')
plt.plot(x, silu, label='SiLU', color='purple')
plt.plot(x, gelu, label='GELU', color='brown')
plt.plot(x, tanh_gelu, '--', label='Tanh GELU (approximation)', color='brown')  # Dashed line for Tanh GELU

# Add labels, legend, and grid
plt.title('Common Activation Functions in Machine Learning')
plt.xlabel('x')
plt.ylabel('Activation Output')
plt.axhline(0, color='black', linewidth=0.5, linestyle='--')
plt.axvline(0, color='black', linewidth=0.5, linestyle='--')
plt.legend()
plt.grid(alpha=0.3)
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

1) The ReLU is the least computationally expensive to compute. It requires only a simple comparison operation making it highly efficient. 2) SiLU and GELU functions are smooth and non-linear, which can improve gradient flow and prevent vanishing or exploding gradient. GELU, in particular, is widely used in transformer-based models like BERT, as it provides smoother transitions compared to ReLU.Tanh is zero-centered, which can sometimes help with faster convergence compared to sigmoid, especially in models where the mean activation around zero is desirable.

Downfalls: ReLU-While computationally efficient, ReLU suffers from the "dying ReLU problem", where neurons can get stuck with zero gradients if their outputs become negative for all inputs. Leaky ReLU-Leaky ReLU addresses the dying ReLU issue but introduces a small negative slope, which might slightly increase computational cost and may not always yield significant benefits. Tanh and Sigmoid-Both can suffer from the vanishing gradient problem, where gradients become extremely small in deep networks. This can slow or stall training. GELU and SiLU-While effective, these functions are more computationally expensive compared to ReLU or Leaky ReLU. GELU, in particular, involves the error function which can be slow.

3)ReLU: Not ideal when there is a high likelihood of encountering the dying ReLU problem, especially in very deep networks without careful weight initialization. Leaky ReLU: May not be the best choice when strict non-linearity is required, as the small slope in the negative region can leak information. Sigmoid: Avoid in deep networks due to the vanishing gradient problem and non-zero-centered outputs, which can slow convergence. Tanh: Can still suffer from vanishing gradients, especially in very deep networks. It is also computationally more expensive than ReLU. GELU and SiLU: May not be ideal in resource-constrained environments or when computational efficiency is a priority due to their higher cost.

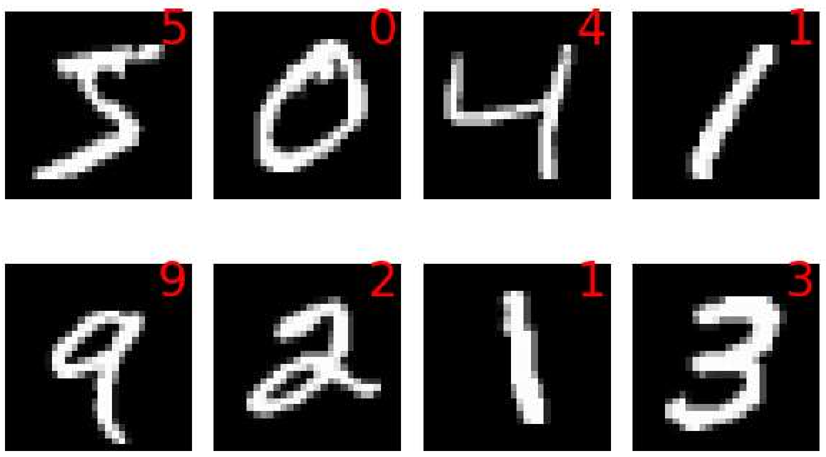
Visualizing 2D data

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=(6, 4))

# 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()
```



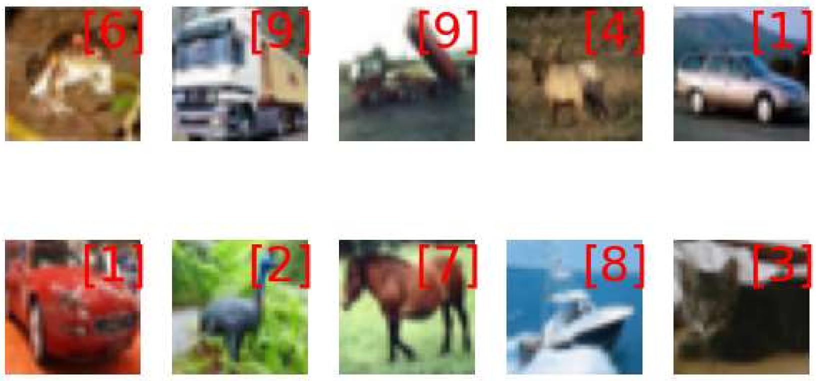
Another popular image dataset for benchmarking and evaluation is CFAR-10. This dataset consists of small (32 x 32 pixel) RGB images of objects that fall into one of 10 classes:

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
import matplotlib.pyplot as plt
(train_images, train_labels), (test_images, test_labels) = cifar10.load_data()
# Define the grid dimensions
rows, cols = 2, 5
# Create a figure and axes for the grid
fig, axes = plt.subplots(rows, cols, figsize=(6, 4))
# 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(0.8, 0.8, 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())
```

```
Downloading data from https://storage.googleapis.com/tensorflow/keras-applications/vgg16/vgg16_weights_tf_dim_ordering_tf_kern
553467096/553467096 ━━━━━━━━━━━━━━━━━━━━ 2s 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)
```

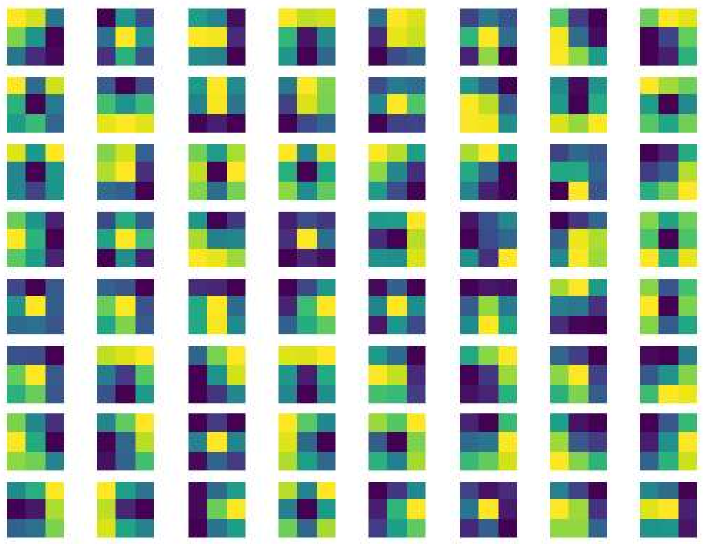
```
# we can also print the model layers based on index to better understand the structure
for i,layer in enumerate(model.layers):
  print(f"{i}: {layer}")
```

```
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.

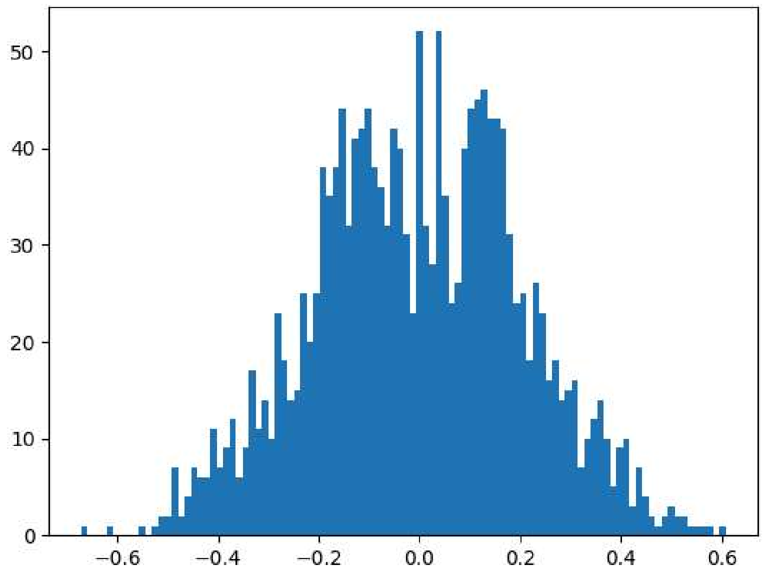
```
# next we can extract som
layer = model.layers[1] # Get the first convolutional layer
weights = layer.get_weights()[0]

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 ge
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>)
```



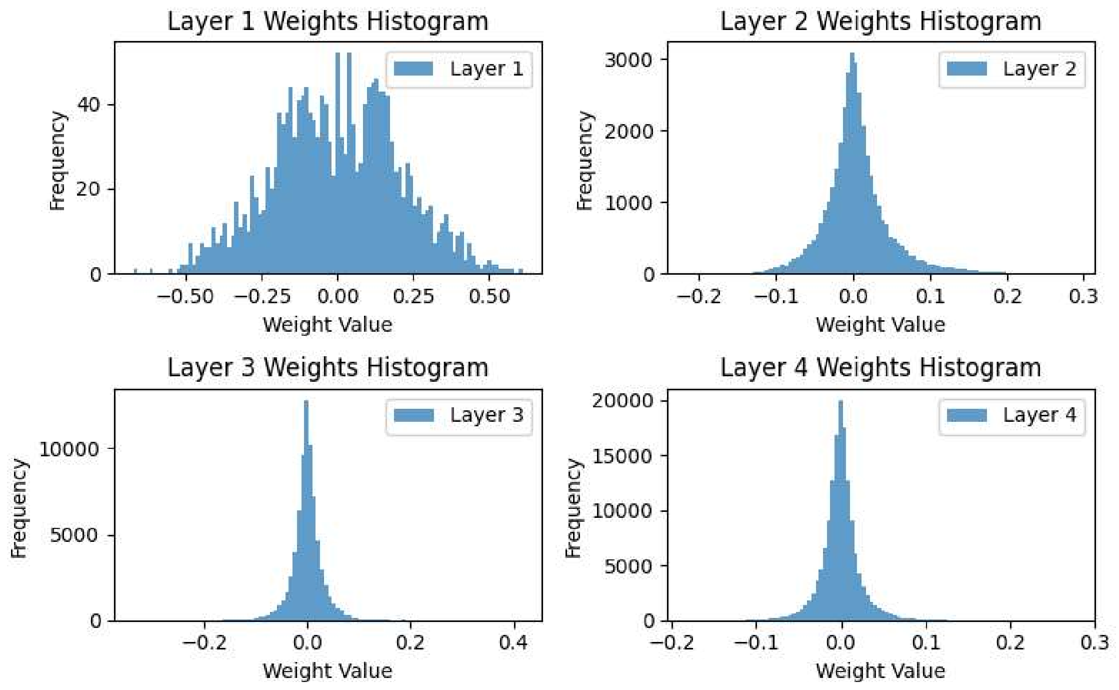
Look through the other weight tensors in the network and note any patterns that can be observed. Plot some examples in a subplot grid (include at least 4 plots). You can also overplot on the same subplot if you find that helpful for visualization.

Question 4

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

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

# Print the model summary
#print(model.summary())

# Retrieve weights from layers that have trainable parameters
weights_list = [layer.get_weights()[0] for layer in model.layers if len(layer.get_weights()) > 0]
```

```
# Create a 2x2 grid for subplots
fig, axes = plt.subplots(2, 2, figsize=(8, 5))

for i, ax in enumerate(axes.flat):
    if i < len(weights_list):
        weights = weights_list[i]

        # Compute mean and variance
        mean = weights.mean()
        variance = weights.var()
        print(f"Layer {i+1}: weight tensor has mean: {mean:.4f} and variance: {variance:.4f}")

        # Plot histogram
        ax.hist(weights.flatten(), bins=100, alpha=0.7, label=f"Layer {i+1}")
        ax.set_title(f"Layer {i+1} Weights Histogram")
        ax.set_xlabel("Weight Value")
        ax.set_ylabel("Frequency")
        ax.legend()
    else:
        # Hide unused subplots
        ax.axis('off')

# Adjust layout and display the plot
plt.tight_layout()
plt.show()
```

```
Layer 1: weight tensor has mean: -0.0024 and variance: 0.0427
Layer 2: weight tensor has mean: 0.0049 and variance: 0.0018
Layer 3: weight tensor has mean: 0.0002 and variance: 0.0010
Layer 4: weight tensor has mean: -0.0003 and variance: 0.0006
```



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.

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

Question 5

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

# Load CIFAR-10 dataset
```

```
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.cifar10.load_data()

# Select a sample image (e.g., the first image from the test set)
image = x_test[0]  # You can choose any index here

# Preprocess the image for VGG16
image = tf.image.resize(image, (224, 224))  # Resize to VGG16 input size
image = np.expand_dims(image, axis=0)  # Add batch dimension

# Convert the image to float32 to make it writeable
image = image.astype(np.float32)

image = tf.keras.applications.vgg16.preprocess_input(image)  # Preprocess for VGG16

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

# Get the activations for the final three dense layers
dense_layer_indices = [-3, -2, -1]  # Assuming the last 3 layers are dense
activations = []
for i in dense_layer_indices:
    activation_model = tf.keras.Model(inputs=model.input, outputs=model.layers[i].output
    activation = activation_model.predict(image)
    activations.append(activation)

# Plot the activation distributions
fig, axes = plt.subplots(3,1,figsize=(8,5))
for i, activation in enumerate(activations):
    axes[i].hist(activation.flatten(), bins=50)
    axes[i].set_title(f"Activation Distribution of Layer{dense_layer_indices[i]}")
    axes[i].set_xlabel("Activation Value")
    axes[i].set_ylabel("Frequency")
plt.tight_layout()
plt.show()
```

```
1/1 ━━━━━━━━━━━━━━━━━━━━ 1s 564ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 367ms/step
1/1 ━━━━━━━━━━━━━━━━━━━━ 0s 444ms/step
```