ResNet Algorithm for Image Detection

Implemenation Author: Matthew Yeseta

ResNet, Residual Network, is a type of deep neural network that makes use of skip connections to jump over some layers. This allows ResNet to mitigate the vanishing gradient problem. This problem typically occurs in very deep networks, where the gradient tends to become too small, preventing the network from learning effectively.

The architecture includes residual units, where the network learns the residual of the function rather than the full transformation. This full transformaion is use for training this Resnet deep networks, in order to assist in better performance and improved image recognition detection.

In my code, a scaled version of the ResNet architecture has residual blocks in order to help the model to learn challning representations in the set of leaf images. Leaf images subsequeenly are process multiple convolutional lays and dense layers. Utilize with skip connections to aid the Resent to retain key or vial features at different levels of abstraction.

In [16]:
```
import numpy as np # linear algebra
import pandas as pd # data processing, CSV file I/O (e.g. pd.read_csv)

import os
for dirname, _, filenames in os.walk('/kaggle/input'):
    for filename in filenames:
        print(os.path.join(dirname, filename))
```

In [17]:
```
import os
import numpy as np
import tensorflow as tf
from tensorflow import keras
from tensorflow.keras import layers
from sklearn.preprocessing import LabelEncoder
import deeplake
import matplotlib.pyplot as plt
from sklearn.metrics import classification_report, accuracy_score
import random
#!pip install deeplake
```

Key Classes and Their Functions: ImageDataLoader: This class is responsible for loading the dataset from DeepLake. It retrieves the image data from the dataset and limits the number of images to a specified maximum (in this case, 300). It also splits the data into training, validation, and test sets.

Methods: init: Loads the dataset and retrieves up to 300 images. split_data: Splits the images into 75% training, 7% validation, and 18% test data. ResNetImageProcessor: This class processes the images by resizing and normalizing them, preparing them for input into the ResNet model. It includes an optional method to crop the image, which can be useful when working with images with excess padding or noise.

Methods: crop_image: Crops out unnecessary zero-pixel regions in the images. process_image: Converts each image to grayscale (if needed), resizes it to 175x175 pixels, and normalizes the pixel values. ImageResidualUnit: This class defines the residual unit, a key building block of ResNet architecture. A residual unit allows the network to learn residual functions with reference to the input, which helps in overcoming vanishing gradient problems and improving the training of deep networks.

Methods: init: Initializes the residual unit layers (main and skip layers). call: Defines the forward pass of the residual unit, combining the input (skip connection) with the output from the main convolutional layers. ResNetModel: This class constructs the overall ResNet model. It defines the architecture using residual units and includes methods to compile the model and add residual blocks.

Methods: init: Initializes the ResNet model. build_model: Constructs the ResNet model using convolutional layers, residual units, and dense layers. add_residual_units: Adds multiple residual units to the model for better feature extraction and deep learning capabilities. ImagePredictionModel: This is the main class responsible for orchestrating the image loading, preprocessing, model training, and displaying predictions. It integrates the functionality of ImageDataLoader, ResNetImageProcessor, and ResNetModel.

Methods: init: Initializes the data loader, encoder, and ResNet model. run: Splits the dataset, preprocesses the images, trains the model, and generates predictions. view_predictions: Visualizes the predicted and original images side by side for comparison.

In [18]:
```
class ImageDataLoader:
    def __init__(self, dataset_path, max_images=300):
        self.dataset = deeplake.load(dataset_path)  # Using `load()` for Deeplake 3.x
        self.image_paths = self._load_image_paths(max_images)

    def _load_image_paths(self, max_images):
        images = []
        for i, sample in enumerate(self.dataset['images']):
            images.append(sample.numpy())
            if len(images) >= max_images:
                break
        return images

    def split_data(self):
        train_idx = int(len(self.image_paths) * 0.75)
```

```
        val_idx = int(len(self.image_paths) * 0.82)
        return (self.image_paths[:train_idx],
                self.image_paths[train_idx:val_idx],
                self.image_paths[val_idx:])
```

In [19]:

```
class ResNetImageProcessor:
    @staticmethod
    def crop_image(image):
        nonzero_indices = np.argwhere(image != 0)
        y_min, y_max = nonzero_indices[:, 0].min(), nonzero_indices[:, 0].max()
        x_min, x_max = nonzero_indices[:, 1].min(), nonzero_indices[:, 1].max()
        return image[y_min:y_max, x_min:x_max]

    @staticmethod
    def process_image(image_paths):
        X = []
        for img in image_paths:
            resized_img = tf.image.resize(img, (175, 175))  # Keeping the image in RGB
            X.append(np.array(resized_img / 255.0, dtype=np.float16))
        return np.array(X)


class ImageResidualUnit(keras.layers.Layer):
    def __init__(self, filters, strides=1, activation="selu", **kwargs):
        super().__init__(**kwargs)
        self.activation = keras.activations.get(activation)
        self.main_layers = [
            layers.Conv2D(filters, 3, strides=strides, padding="same"),
            layers.BatchNormalization(),
            self.activation,
            layers.Conv2D(filters, 3, strides=1, padding="same"),
            layers.BatchNormalization()
        ]
        self.skip_layers = []
        if strides > 1:
            self.skip_layers = [
                layers.Conv2D(filters, 1, strides=strides, padding="same"),
                layers.BatchNormalization()
            ]

    def call(self, inputs):
        Z = inputs
        for layer in self.main_layers:
            Z = layer(Z)
        skip_Z = inputs
```

```
        for layer in self.skip_layers:
            skip_Z = layer(skip_Z)
        return self.activation(Z + skip_Z)

class ResNetModel:
    def __init__(self):
        self.model = self.build_model()

    def build_model(self):
        model = keras.models.Sequential([
            layers.Conv2D(512, 7, strides=2, padding="same", activation="selu", input_shape=(175, 175, 3)),
            layers.Dropout(0.2),
            layers.Conv2D(128, 3, strides=1, padding="same", activation="selu"),
            layers.MaxPool2D(pool_size=3, strides=1, padding="same")
        ])

        self.add_residual_units(model)

        model.add(layers.Flatten())
        model.add(layers.Dense(128, activation='selu'))
        model.add(layers.Dropout(0.4))
        model.add(layers.Dense(64, activation='selu'))
        model.add(layers.Dropout(0.3))
        model.add(layers.Dense(4, activation='softmax'))

        model.compile(loss='sparse_categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
        return model


    def add_residual_units(self, model):
        filters = [128] * 2 + [64] * 2
        for f in filters:
            model.add(ImageResidualUnit(f, strides=2))

class ImagePredictionModel:
    def __init__(self, dataset_path, epoch=1, max_images=300):
        self.loader = ImageDataLoader(dataset_path, max_images=max_images)
        self.encoder = LabelEncoder()
        self.model = ResNetModel()
        self.epoch = epoch


    def run(self):
        # Split the data into train, validation, and test sets
        X_train, X_val, X_test = self.loader.split_data()
```

```python
        X_train = ResNetImageProcessor.process_image(X_train)
        X_val = ResNetImageProcessor.process_image(X_val)
        X_test = ResNetImageProcessor.process_image(X_test)

        history = self.model.model.fit(X_train, np.zeros(len(X_train)),
                                       validation_data=(X_val, np.zeros(len(X_val))),
                                       epochs=self.epoch, batch_size=32)

        prob_pred_ResNet = self.model.model.predict(X_test)
        y_pred_ResNet = np.argmax(prob_pred_ResNet, axis=1)
        self.display_results(history.history, X_test, y_pred_ResNet)

    def display_results(self, history, X_test, y_test):
        fig, axs = plt.subplots(1, 2, figsize=(20, 7))
        plt.title("ResNet Model")

        axs[0].plot(history['accuracy'], label='Accuracy')
        axs[0].plot(history['val_accuracy'], label='Val Accuracy')
        axs[0].legend()
        axs[0].grid()

        axs[1].plot(history['loss'], label='Loss')
        axs[1].plot(history['val_loss'], label='Val Loss')
        axs[1].legend()
        axs[1].grid()

        plt.show()
```

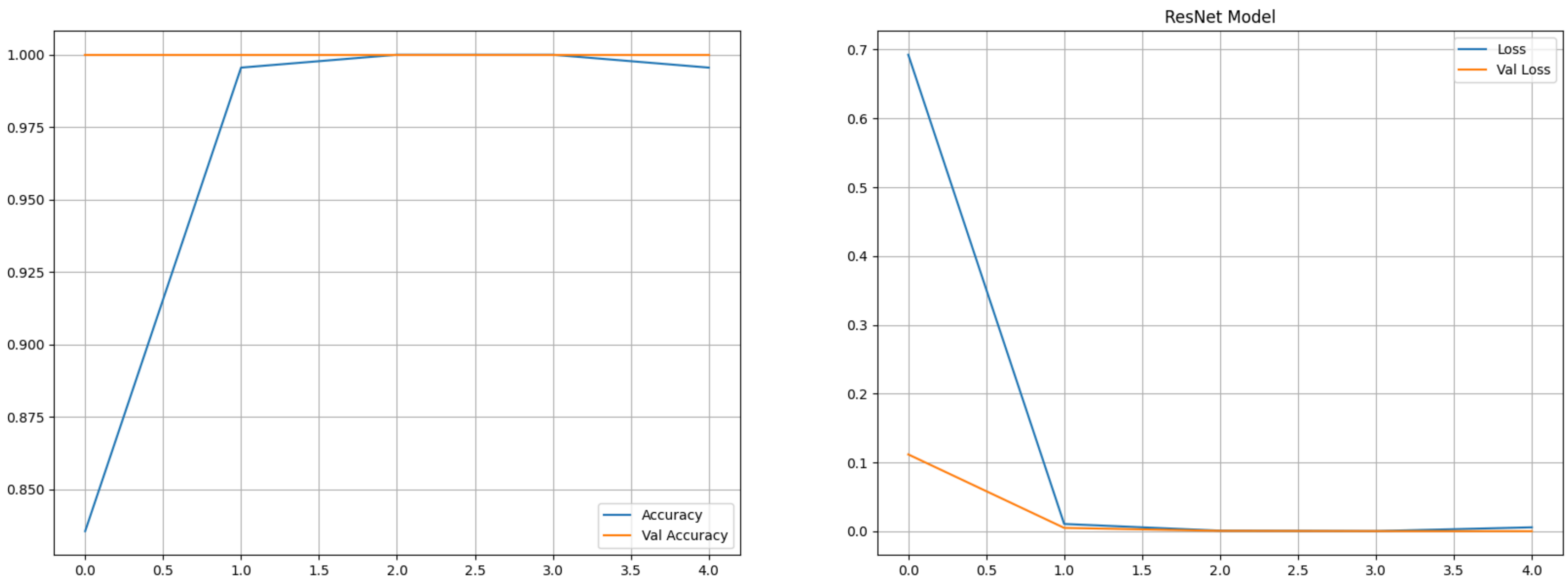
In [20]:

```python
# Main execution logic
if __name__ == '__main__':
    dataset_path = 'hub://activeloop/plantvillage-without-augmentation'
    model = ImagePredictionModel(dataset_path, epoch=5, max_images=300)
    model.run()
```

```
\
Opening dataset in read-only mode as you don't have write permissions.
\
This dataset can be visualized in Jupyter Notebook by ds.visualize() or at https://app.activeloop.ai/activeloop/plantvi
llage-without-augmentation

/
hub://activeloop/plantvillage-without-augmentation loaded successfully.
```

```
Epoch 1/5
8/8 ━━━━━━━━━━━━━━━━━━━━ 102s 11s/step - accuracy: 0.6610 - loss: 1.4767 - val_accuracy: 1.0000 - val_loss: 0.1116
Epoch 2/5
8/8 ━━━━━━━━━━━━━━━━━━━━ 140s 11s/step - accuracy: 0.9952 - loss: 0.0113 - val_accuracy: 1.0000 - val_loss: 0.0048
Epoch 3/5
8/8 ━━━━━━━━━━━━━━━━━━━━ 89s 11s/step - accuracy: 1.0000 - loss: 4.1603e-04 - val_accuracy: 1.0000 - val_loss: 5.7994e-
04
Epoch 4/5
8/8 ━━━━━━━━━━━━━━━━━━━━ 142s 11s/step - accuracy: 1.0000 - loss: 1.9277e-04 - val_accuracy: 1.0000 - val_loss: 1.8677e
-04
Epoch 5/5
8/8 ━━━━━━━━━━━━━━━━━━━━ 90s 11s/step - accuracy: 0.9979 - loss: 0.0034 - val_accuracy: 1.0000 - val_loss: 6.4003e-05
2/2 ━━━━━━━━━━━━━━━━━━━━ 7s 3s/step
```



In [21]:
```python
class ImagePredictionModel:
    def __init__(self, dataset_path, epoch=1, max_images=300):
        self.loader = ImageDataLoader(dataset_path, max_images=max_images)
        self.encoder = LabelEncoder()
        self.model = ResNetModel()
        self.epoch = epoch

    def run(self):
        X_train, X_val, X_test = self.loader.split_data()
        X_train = ResNetImageProcessor.process_image(X_train)
        X_val = ResNetImageProcessor.process_image(X_val)
```

```
        X_test = ResNetImageProcessor.process_image(X_test)

        history = self.model.model.fit(X_train, np.zeros(len(X_train)),
                                       validation_data=(X_val, np.zeros(len(X_val))),
                                       epochs=self.epoch, batch_size=32)

        prob_pred_ResNet = self.model.model.predict(X_test)
        y_pred_ResNet = np.argmax(prob_pred_ResNet, axis=1)
        predicted_images = get_predicted_images(X_test, y_pred_ResNet)
        view_predictions(predicted_images)

    def display_results(self, history, X_test, y_test):
        fig, axs = plt.subplots(1, 2, figsize=(20, 7))
        plt.title("ResNet Model")

        axs[0].plot(history['accuracy'], label='Accuracy')
        axs[0].plot(history['val_accuracy'], label='Val Accuracy')
        axs[0].legend()
        axs[0].grid()

        axs[1].plot(history['loss'], label='Loss')
        axs[1].plot(history['val_loss'], label='Val Loss')
        axs[1].legend()
        axs[1].grid()

        plt.show()
```

In [22]:

```
from PIL import Image
import matplotlib.pyplot as plt
import io

def get_predicted_images(X_test, y_pred):
    predicted_images = []

    for i, img in enumerate(X_test):
        img = (img * 255).astype(np.uint8)
        pil_img = Image.fromarray(img.reshape(175, 175, 3))
        img_byte_array = io.BytesIO()
        pil_img.save(img_byte_array, format='PNG')
        img_byte_array = img_byte_array.getvalue()
        predicted_images.append((img_byte_array, y_pred[i]))

    return predicted_images
```

```
def view_predictions(predicted_images):
    plt.figure(figsize=(15, 10))

    for i in range(5):
        img_data, prediction = predicted_images[i]
        img = Image.open(io.BytesIO(img_data))
        plt.subplot(2, 5, i + 1)
        plt.imshow(img)
        plt.title(f'Predicted Class: {prediction}')
        plt.axis('off')

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

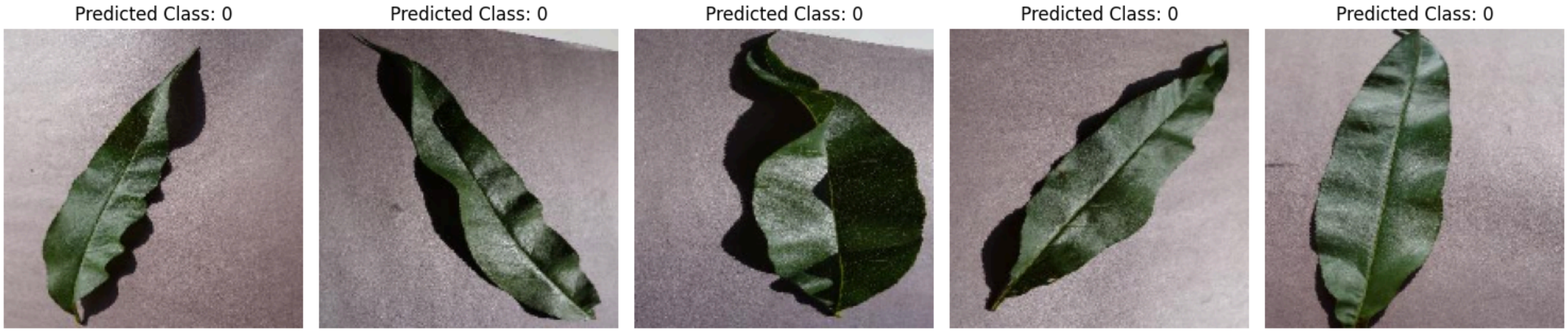
In [23]:

```
# Main execution logic
if __name__ == '__main__':
    dataset_path = 'hub://activeloop/plantvillage-without-augmentation'
    model = ImagePredictionModel(dataset_path, epoch=3, max_images=300)
    model.run()
```

```
\
Opening dataset in read-only mode as you don't have write permissions.
|
This dataset can be visualized in Jupyter Notebook by ds.visualize() or at https://app.activeloop.ai/activeloop/plantvillage-without-augmentation

-
hub://activeloop/plantvillage-without-augmentation loaded successfully.


Epoch 1/3
8/8 ━━━━━━━━━━━━━━━━━━━━ 100s 11s/step - accuracy: 0.6946 - loss: 1.3988 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 2/3
8/8 ━━━━━━━━━━━━━━━━━━━━ 142s 11s/step - accuracy: 1.0000 - loss: 1.0583e-05 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
Epoch 3/3
8/8 ━━━━━━━━━━━━━━━━━━━━ 89s 11s/step - accuracy: 1.0000 - loss: 1.6817e-08 - val_accuracy: 1.0000 - val_loss: 0.0000e+00
2/2 ━━━━━━━━━━━━━━━━━━━━ 7s 3s/step
```


Predicted Class: 0
Predicted Class: 0
Predicted Class: 0
Predicted Class: 0
Predicted Class: 0