D604 Task 1: Neural Networks (Image Classification)

Jesse Byers

College of Information Technology, Western Governors University

Dr. Sherin Aly (Instructor)

June 17, 2025

D604 Task 1: Neural Networks (Image Classification)

Part I: Research Question

A1: Research Question

To what extent can a neural network accurately distinguish between common crop and weed seedlings?

A2: Objectives or Goals

The goal of this project is to create and train a neural network to accurately classify and differentiate between 12 common crop and weed seedlings. If the model can make predictions with sufficient accuracy, then the model outputs could be used to guide decisions around automated weeding protocols and crop care. As a result, the farming community would be able to increase crop yields, using fewer resources such as water and fertilizer, because they would be able to remove weeds at an early stage before they consume the main crop's resources.

A3: Neural Network Type

A Convolutional Neural Network (CNN) was chosen for this project and implemented using PyTorch.

A4: Neural Network Justification

A Convolutional Neural Network (CNN) uses deep learning to enable tasks such as object detection, image classification into categories, or anomaly detection in images. When used for image classification, they require a large input set of structured and labeled image data representing samples from each class to be classified. During training, the processed image data is fed through a number of locally-connected layers of the network that perform different operations on the image data in order to detect meaningful patterns. The convolutional layers in the network use matrices as a filter that slides over the entire image and creates various feature

maps. These feature maps are able to detect meaningful features such as horizontal or vertical lines, edges, textures, etc. Activation functions are used to introduce non-linearity, which allows the network to detect and learn more complex patterns in the image data. Pooling layers are then used to reduce the size of the feature maps, which allows the neural network to operate more efficiently with fewer parameters. Finally, fully connected layers are used after the locally-connected layers to collect the outputs and make a final prediction.

During training and validation, the image data is fed through the layers as described above. This is referred to as the forward pass and produces an output as the probability that an image represents each of the possible classes. Next, a loss function is used to compare the actual class to the model's predicted class. The results of this calculation are used in the backpropagation process, which feeds the difference back through the model's parameters to adjust the weights, which should lead to higher levels of accuracy.

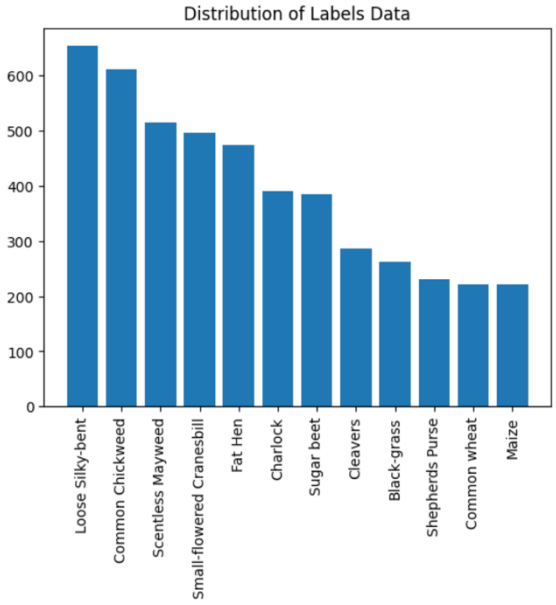
CNNs are not the only option for a neural network designed for image classification. Multilayer Perceptrons, or MLPs, can also be used for this purpose. However, MLPs only use fully connected layers and flatten the input image into a long vector, which can lead the model to miss some spatial relationships. Since CNNs use locally-connected layers, they can pick up on more complex spatial relationships with a smaller number of parameters. Therefore, a CNN is a better choice for this seedling identification problem.

Part II: Data Preparation

B1a: Data Visualization

The original dataset included the following distribution of data:

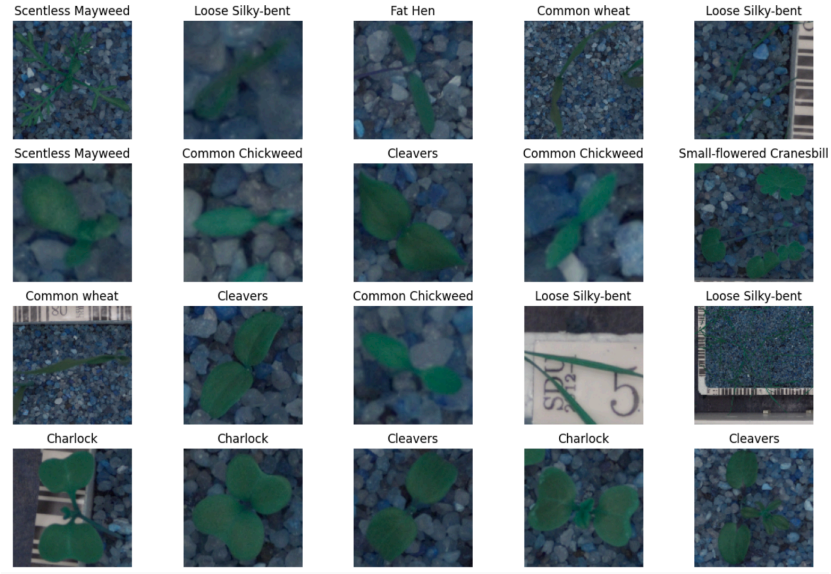
Label	
Loose Silky-bent	654
Common Chickweed	611
Scentless Mayweed	516
Small-flowered Cranesbill	496
Fat Hen	475
Charlock	390
Sugar beet	385
Cleavers	287
Black-grass	263
Shepherds Purse	231
Common wheat	221
Maize	221



The distribution shows that the original dataset is unbalanced in terms of the representation of each of the 12 plant species classes. Unbalanced datasets have the potential to impair model accuracy because the model can learn to over-predict the classes that are over-represented in the dataset. Steps to rebalance the dataset will be described in a later section.

B1b: Sample Images

The screenshot below depicts 20 sample images, chosen at random from the original dataset, with their associated labels.

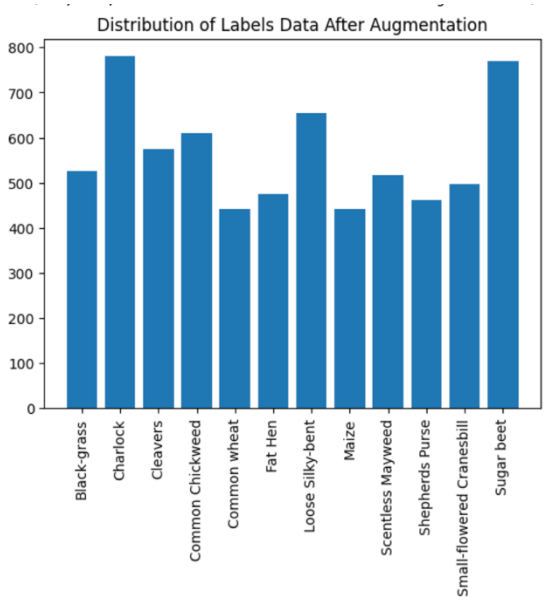


B2: Augmentation and Justification

As described above, the original dataset was unbalanced in terms of its representation of each plant species class, with a range from only 221 samples for Maize and Common Wheat, to 654 samples of Loose Silky-bent. If Maize and Common Wheat are the target crops and the rest of the plants are weeds, this under-representation in the dataset would likely lead to a model that over-identifies crop images as weeds.

The first data augmentation strategy is intended to better balance the dataset by increasing its size. In a real-world scenario, I would ask for more real images of the under-represented classes before building and training a model, which should not be an unrealistic request for a large agricultural company investing in this project.

As a proxy for this, in this project, I decided to augment the dataset by artificially increasing the number of images of each class that had fewer than 400 samples by duplicating the original images. This resulted in a new dataset with the following distribution, with all classes having at least 400 samples.




```
# double images from least-represented
classes (with label counts under 400)

for item in labeled_images[0:4750]:
  if item[1] in [1, 11, 2, 0, 9, 4, 7]:
    labeled_images.append(item)

# ["Charlock", "Sugar beet", "Cleavers",
"Black-grass", "Shepherds Purse",
"Common wheat", "Maize"] (less than 400 samples)
```

The implications of this method of data augmentation to increase the sample size of under-represented classes will be further discussed in section G4: Improvements.

The second data augmentation approach was intended to increase the diversity of the training images, and was only performed on the training set of data while creating the DataLoaders.

```
# define transforms to augment training data
training_transforms = [
    # random alteration of rotation, translation, shear
    transforms.RandomAffine(scale=(0.9, 1.1), translate=(0.1, 0.1),
degrees=10),
    transforms.RandomHorizontalFlip(0.5),
    transforms.RandomVerticalFlip(0.5),
    ]

train_transform = transforms.Compose(training_transforms)

…
# Create the datasets and DataLoaders
train_dataset = CustomDataset(train_final_images, train_labels,
transform=train_transform)
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
```

This code causes the training images to undergo transformations around the vertical and horizontal orientation, as well as rotation and translations from the original image orientation. These transformations are applied when the DataLoaders are created. The transformations result

in a more robust training set with more variation, which will lead to better training and less over-fitting of the model to training data.

B3: Normalization Steps

In normalization, the image data is re-centered, which increases consistency across input images and leads to better and more efficient training, resulting in better speed and accuracy. All data (training, validation, and testing) was normalized using the following code:

```
# define transforms to normalize validation and test data
normalize_transforms = [
    transforms.ToTensor(),
    transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5)),
    ]


normalize_transform = transforms.Compose(normalize_transforms)


val_final_images = []
for i, img in enumerate(val_images):
  val_final_images.append(normalize_transform(img))


test_final_images = []
for i, img in enumerate(test_images):
  test_final_images.append(normalize_transform(img))


train_final_images = []
for i, img in enumerate(train_images):
  train_final_images.append(normalize_transform(img))
```

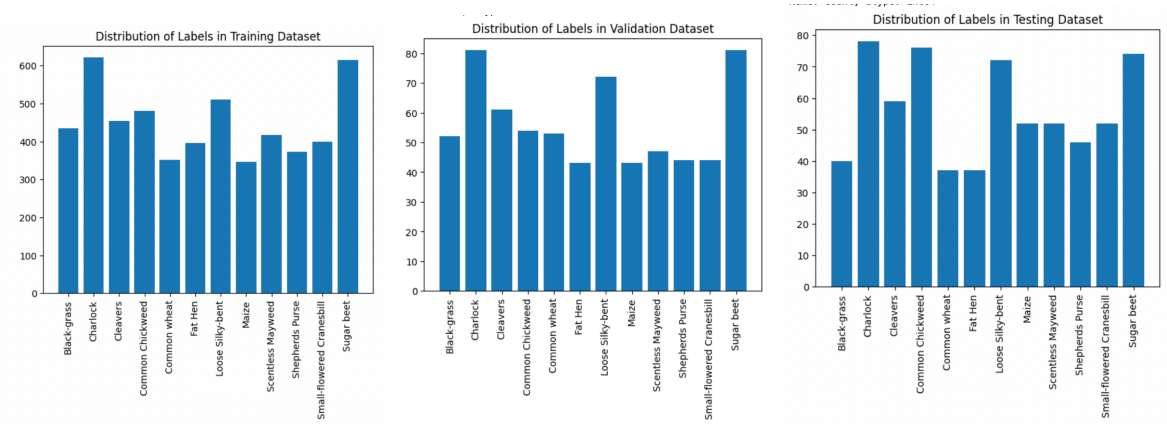
The values chosen for the `transforms.Normalize()` function sets the mean and standard deviation of all three color channels to 0.5.

B4: Train - Validation - Test

The augmented dataset was split into three subsets for training (80%), validation (10%), and testing (10%). The screenshots below show the code used for the split, as well as the distribution of classes represented within each subset. These percentages were chosen to maximize the overall size of the training set (80%) because the overall dataset size is not terribly large. Using only 10% for training and validation should be a sufficient amount for those purposes. As the screenshots show, each class has over 300 samples in the training set, and over 35 samples in the validation and testing sets.

```
# split data into training (80%), validation (10%), and testing (10%) datasets

train_len = int(len(labeled_images) * 0.8)
val_len = int((len(labeled_images) - train_len) / 2)
test_len = int(len(labeled_images) - train_len - val_len)

train_subset, val_subset, test_subset = torch.utils.data.random_split(
    labeled_images, [train_len, val_len, test_len]
)

print(f"Length of training subset: {len(train_subset)}")
print(f"Length of validation subset: {len(val_subset)}")
print(f"Length of testing subset: {len(test_subset)}")
```

```
Length of training subset: 5398
Length of validation subset: 675
Length of testing subset: 675
```



B5: Target Encoding

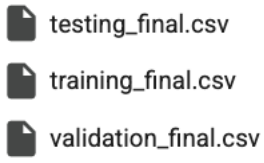
SciKit-Learn's label encoder was used to encode the categorical labels (plant species) into integer values before training the model. The loop also creates a plant_map dictionary to make it easier to decode the integer values back into the original plant species names.

```
# encode labels using LabelEncoder

encoder = LabelEncoder()

labels["Label"] = encoder.fit_transform(labels["Label"])


# create plant_map variable and print encoder mapping

print("Label mapping:")

plant_map = {}

for i, item in enumerate(encoder.classes_):

    plant_map[i] = item

    print(f"{item} : {i}")

print(plant_map)
```

```
Label mapping:
Black-grass : 0
Charlock : 1
Cleavers : 2
Common Chickweed : 3
Common wheat : 4
Fat Hen : 5
Loose Silky-bent : 6
Maize : 7
Scentless Mayweed : 8
Shepherds Purse : 9
Small-flowered Cranesbill : 10
Sugar beet : 11
```

B6: Datasets Copy

The submission includes CSV files of the training, validation, and testing datasets.




```
# save final datasets to csv

# training dataset with labels (normalized and

augmented)

train_zip = list(zip(train_final_images,

train_labels))

df1 = pd.DataFrame(train_zip)

df1.to_csv('training_final.csv', index=False)


# validation dataset with labels (normalized)

val_zip = list(zip(val_final_images, val_labels))

df2 = pd.DataFrame(val_zip)

df2.to_csv('validation_final.csv', index=False)


# testing dataset with labels (normalized)

test_zip = list(zip(test_final_images, test_labels))

df2 = pd.DataFrame(test_zip)

df2.to_csv('testing_final.csv', index=False)
```

Part III: Network Architecture

E1: Model Summary Output

The final model was created using PyTorch's neural network module, with the following architecture:

```
# build CNN model

class Net(nn.Module):
    def __init__(self, n_classes=12):

        super(Net, self).__init__()

        self.model = nn.Sequential(
            nn.Conv2d(in_channels=3, out_channels=16, kernel_size=3,
stride=1, padding=1),
            nn.BatchNorm2d(16),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2), # 16x64x64

            nn.Conv2d(in_channels=16, out_channels=32, kernel_size=3,
stride=1, padding=1),
            nn.BatchNorm2d(32),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),  # 32x32x32

            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3,
stride=1, padding=1),  # -> 64x8x8
            nn.BatchNorm2d(64),
            nn.ReLU(),
            nn.MaxPool2d(kernel_size=2, stride=2),  # 64x16x16

            nn.AdaptiveAvgPool2d((1, 1)),   # reduces to (batch_size, 64, 1,
1)
            nn.Flatten(),
            nn.Dropout(0.3),
            nn.Linear(64, n_classes)
        )

    def forward(self, x):
        return self.model(x)

model = Net()
```

```
Net(
  (model): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (2): ReLU()
    (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (4): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (6): ReLU()
    (7): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (8): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (10): ReLU()
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): AdaptiveAvgPool2d(output_size=(1, 1))
    (13): Flatten(start_dim=1, end_dim=-1)
    (14): Dropout(p=0.3, inplace=False)
    (15): Linear(in_features=64, out_features=12, bias=True)
  )
)
```

After training, the best model's summary and parameters were printed:

```
# Print best model information
model.load_state_dict(torch.load('best_model.pth'))

print("Model Summary:")
print(summary(model, (3, 128, 128)))

print("Model Parameters:")
for name, param in model.named_parameters():
    if param.requires_grad:
        print(name, param.data.shape)
```

```
Model Summary:
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1         [-1, 16, 128, 128]             448
       BatchNorm2d-2         [-1, 16, 128, 128]              32
              ReLU-3         [-1, 16, 128, 128]               0
         MaxPool2d-4           [-1, 16, 64, 64]               0
            Conv2d-5           [-1, 32, 64, 64]           4,640
       BatchNorm2d-6           [-1, 32, 64, 64]              64
              ReLU-7           [-1, 32, 64, 64]               0
         MaxPool2d-8           [-1, 32, 32, 32]               0
            Conv2d-9           [-1, 64, 32, 32]          18,496
      BatchNorm2d-10           [-1, 64, 32, 32]             128
             ReLU-11           [-1, 64, 32, 32]               0
        MaxPool2d-12           [-1, 64, 16, 16]               0
AdaptiveAvgPool2d-13             [-1, 64, 1, 1]               0
          Flatten-14                   [-1, 64]               0
          Dropout-15                   [-1, 64]               0
           Linear-16                   [-1, 12]             780
================================================================
Total params: 24,588
Trainable params: 24,588
Non-trainable params: 0
----------------------------------------------------------------
Input size (MB): 0.19
Forward/backward pass size (MB): 11.38
Params size (MB): 0.09
Estimated Total Size (MB): 11.66
----------------------------------------------------------------
None
```

```
Model Parameters:
model.0.weight torch.Size([16, 3, 3, 3])
model.0.bias torch.Size([16])
model.1.weight torch.Size([16])
model.1.bias torch.Size([16])
model.4.weight torch.Size([32, 16, 3, 3])
model.4.bias torch.Size([32])
model.5.weight torch.Size([32])
model.5.bias torch.Size([32])
model.8.weight torch.Size([64, 32, 3, 3])
model.8.bias torch.Size([64])
model.9.weight torch.Size([64])
model.9.bias torch.Size([64])
model.15.weight torch.Size([12, 64])
model.15.bias torch.Size([12])
```

E2a - E2b: Number of Layers and Types of Layers

The model includes 16 layers, including several different types, as described below. The layers are organized into three blocks of locally-connected layers, followed by one block of fully-connected layers to generate the model output.

- ***Convolutional Layers (Conv2d)*** - These locally-connected layers are responsible for extracting features from the input data. They do this by passing filters over regions of the input image, creating feature maps that detect aspects such as lines and edges. The summary shows that the first convolutional layer, for example, has an input of 3 channels (red, blue,

green) and produces an output of 16 channels, which are 16 different feature maps of 128 x 128 pixels.

- ***Batch Normalization Layers (BatchNorm2d)*** - These layers normalize the outputs of the previous layer so they can be used as inputs to the next layer. This normalization improves performance and training outcomes, but does not change the shape of the output.
- ***Activation Functions (ReLU)*** - The Rectified Linear Unit function ensures that only positive signals are passed on to the next layer of the model. This introduces non-linearity and helps to ensure that the most prominent features are detected, as well as more complex patterns. Again, this layer does not change the shape of the output, but only adjusts the values of the outputs.
- ***Pooling Layers (MaxPool2d, AdaptiveAvgPool2d)*** - Pooling layers reduce the feature map size, which makes the model's learning more generalized and efficient. Max Pooling considers a region of pixels (defined by the window size and stride) and uses the maximum value to represent the entire region. In Adaptive Average Pooling, only the output shape is defined, and the pooling operation adapts by using different-sized regions and strides, and taking the average of the values within the region. Using Max Pooling in the locally-connected layers allows one to focus on the most prominent features, while using Adaptive Average Pooling in the fully-connected layers gives flexibility to work with varying input sizes if needed.
- ***Flatten Layer (Flatten)*** - This layer flattens the input into a one-dimensional vector, so it can be processed in the fully-connected layers.
- ***Dropout Layer (Dropout)*** - This layer implements regularization to reduce overfitting. It works by randomly dropping out a certain number of neurons in the layer. This prevents

overfitting by adding some noise and forcing the model to develop multiple pathways to determine a prediction.

- ***Linear Layer (Linear)*** - This layer is a fully-connected layer. It takes the output of the flattening layer (a vector with a size of 64) and outputs 12 elements (one for each plant species class). Each element of the output is the probability that the original input image belongs to that class.

E2c: Nodes per Layer

In general, the model architecture was chosen in an effort to progressively increase the number of channels at each layer (depth), while decreasing the input size (in terms of pixel grid). This progression allows the model to discover more meaningful, complex patterns in the image while conserving resources.

The number of nodes per layer can be computed by multiplying the number of channels by the layer's output size (height x width). The model summary output below has been annotated to include the number of nodes at each layer in the rightmost column, in bold:

Layer (type)	Output Shape	Param #	**Nodes #**
Conv2d-1	[-1, **16, 128, 128**]	448	**262,144**
BatchNorm2d-2	[-1, **16, 128, 128**]	32	**262,144**
ReLU-3	[-1, **16, 128, 128**]	0	**262,144**
MaxPool2d-4	[-1, **16, 64, 64**]	0	**65,536**
Conv2d-5	[-1, **32, 64, 64**]	4,640	**131,072**
BatchNorm2d-6	[-1, **32, 64, 64**]	64	**131,072**
ReLU-7	[-1, **32, 64, 64**]	0	**131,072**
MaxPool2d-8	[-1, **32, 32, 32**]	0	**32,768**
Conv2d-9	[-1, **64, 32, 32**]	18,496	**65,536**
BatchNorm2d-10	[-1, **64, 32, 32**]	128	**65,536**
ReLU-11	[-1, **64, 32, 32**]	0	**65,536**
MaxPool2d-12	[-1, **64, 16, 16**]	0	**16,384**
AdaptiveAvgPool2d-13	[-1, **64, 1, 1**]	0	**64**
Flatten-14	[-1, **64**]	0	**64**
Dropout-15	[-1, **64**]	0	**64**
Linear-16	[-1, **12**]	780	**12**

E2d: Number of Parameters

The number of parameters is provided in the model summary and can also be calculated using the following formula:

```
num_params = (height x width x input_channels x output_channels) + output_channels
```

The weights are based on the filters that are developed from the model, calculated by multiplying the height, width, and number of channels in the input and output. The bias is the number of output channels. The third column in the chart above (section E2c) shows the number of parameters at each layer. The output from the model params in the screenshot below summarizes the weights at each layer (the four dimensions of the tensor) as well as the bias:

```
Model Parameters:
model.0.weight torch.Size([16, 3, 3, 3])
model.0.bias torch.Size([16])
model.1.weight torch.Size([16])
model.1.bias torch.Size([16])
model.4.weight torch.Size([32, 16, 3, 3])
model.4.bias torch.Size([32])
model.5.weight torch.Size([32])
model.5.bias torch.Size([32])
model.8.weight torch.Size([64, 32, 3, 3])
model.8.bias torch.Size([64])
model.9.weight torch.Size([64])
model.9.bias torch.Size([64])
model.15.weight torch.Size([12, 64])
model.15.bias torch.Size([12])
```

E2e: Activation Functions

As discussed above, the Rectified Linear Unit function ensures that only positive signals are passed on to the next layer of the model. If the input is positive, the value is output without alteration, but if the input is negative, it is output as zero. This introduces non-linearity and helps to ensure that the most prominent features are detected, as well as more complex patterns. This layer does not change the shape of the output, but only adjusts the values of the outputs. While there are several activation functions that could have been used, ReLU was a good choice because it is appropriate for multi-class classification problems, and is simple and computationally efficient.

E3a: Loss Function

```
# loss function (categorical cross-entropy)
criterion = nn.CrossEntropyLoss()
```

Cross-entropy loss was chosen as the loss function for this project. Cross-Entropy Loss is a good choice for CNNs performing multi-class image classification, which is the aim of this project. This loss function measures the difference between the predicted probability for each class label (final output) and the true label from the input data. This loss calculation is then used to adjust the weights of the parameters to penalize the incorrect predictions, leading to a model that performs more accurately over time.

E3b: Optimizer

```
# optimizer
optimizer = torch.optim.Adam(model.parameters(), lr=0.001,
weight_decay=1e-4)
```

The Adam optimizer was used for this model. This optimizer was a good choice because it is more flexible and adaptable than other optimizers, such as SGD. The Adam optimizer has an adaptive learning rate, meaning that it can adjust its learning rate for individual parameters. This adaptability can also lead to faster training times.

E3c: Learning Rate

```
# scheduler
scheduler = optim.lr_scheduler.ReduceLROnPlateau(optimizer, 'min',
patience=5, factor=0.5)
```

In this model, I used a learning rate scheduler in order to dynamically adjust the learning rate as training continued. The base rate was set at 0.001 as a parameter of the Adam optimizer. The Adam optimizer adjusts the learning rate for each parameter, so this initial value is just a baseline starting point. I then set up a scheduler so that the learning rate could be refined as the training progresses. The scheduler pays close attention to the validation loss values and reduces the learning rate when the loss crosses specific thresholds. Specifically, in this model, the

learning rate is adjusted when the validation loss starts to plateau. If the loss is at a plateau for 5 epochs (the patience parameter), it will reduce the learning rate by half (the factor parameter).

E3d: Stopping Criteria

```
    if valid_loss < best_val_loss:
        best_val_loss = valid_loss
        torch.save(model.state_dict(), 'best_model.pth')  # Save best
model
        counter = 0
    else:
        counter += 1
        print(f"EarlyStopping counter: {counter} / {patience}")
        if counter >= patience:
            print("Early stopping triggered.")
            break
```

This model used stopping criteria to avoid overtraining. The number of epochs for training and validation was initially set to 75. The stopping criteria logic was developed as shown above. A counter initialized at 0 kept track of how many times the validation loss was smaller than the best loss encountered in a previous epoch. Once the counter reaches the patience level (set to 7 epochs), the training and validation loop ends. The model state is saved as the best model each time the current model has the smallest loss. Therefore, when the stopping criteria are met, the best trained model has already been saved, and can be loaded from the state at which it exhibited the best loss. The patience criteria of 7 seems to be a good choice. In training the model, the patience reached 5 or 6 a few times, while still achieving better loss values and accuracy, and resetting the counter to zero. While a smaller patience value would have sped up the training, it would have been at the cost of a few percentage points of accuracy.

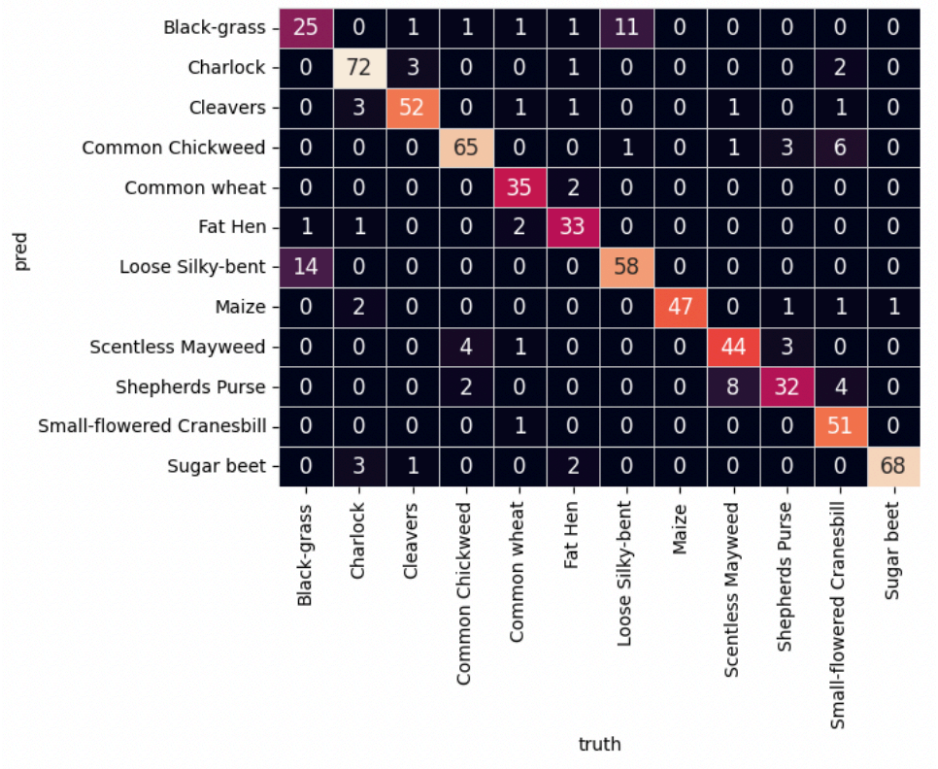
E4: Confusion Matrix

```
# plot confusion matrix

gt = pd.Series(actuals, name='Ground Truth')
predicted = pd.Series(preds, name='Predicted')

confusion_matrix = pd.crosstab(gt, predicted)
```

```
confusion_matrix.index = plant_map.values()

new_column_names = []
for col in confusion_matrix.columns.values:
    new_column_names.append(plant_map[col])
confusion_matrix.columns = new_column_names

display(confusion_matrix)

# visualize confusion matrix
fig, sub = plt.subplots()
with sns.plotting_context("notebook"):

    ax = sns.heatmap(
        confusion_matrix,
        annot=True,
        fmt='d',
        ax=sub,
        linewidths=0.5,
        linecolor='lightgray',
        cbar=False
    )
    ax.set_xlabel("truth")
    ax.set_ylabel("pred")
```

A confusion matrix shows the performance of the model on test data and allows us to compare performance across each of the 12 classes. The y-axis shows predictions for each of the 12 classes, while the x-axis shows true values for each of the 12 classes. Perfect predictive performance would show up as a colorful diagonal line from the top left to the bottom right, with all other cells showing 0 and being represented in black.

The following screenshot shows the confusion matrix for the trained model:



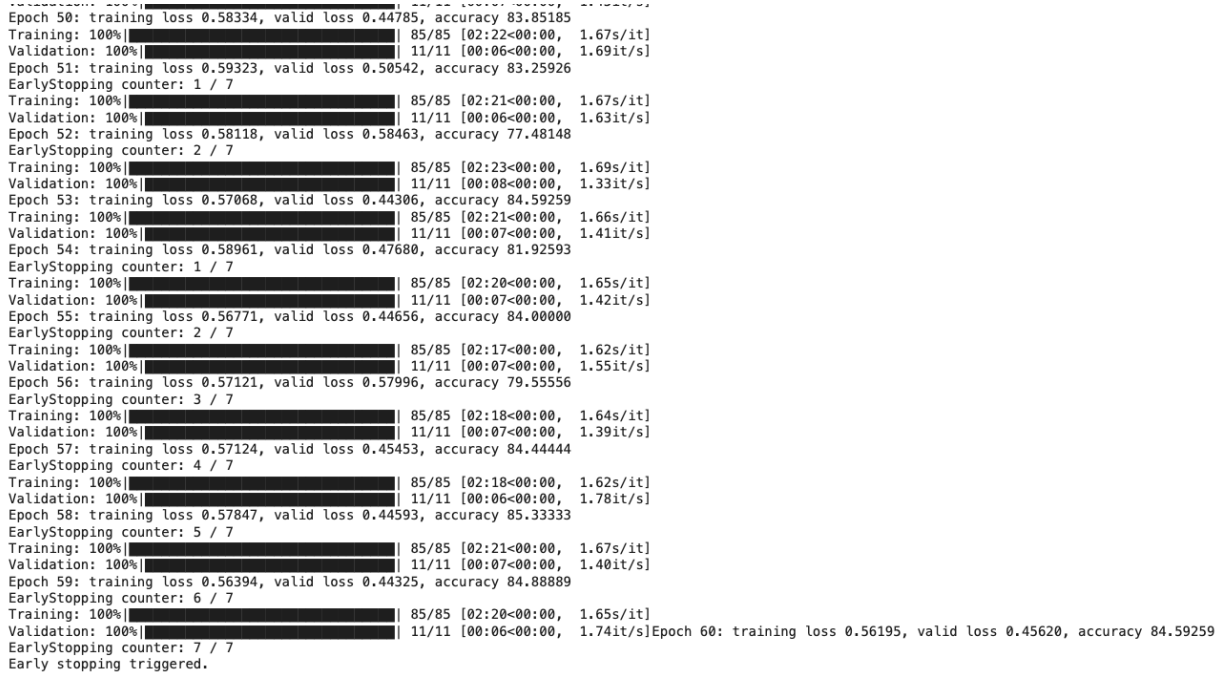
The matrix shows that the model performed relatively well across all classes, with only a small number of missed predictions overall. The only problematic pair that shows up in the matrix is Black-grass and Loose Silky-bent. The matrix shows that 14 instances of Black-grass were misidentified as Loose Silky-bent, while 11 instances of Loose Silky-bent were misidentified as Black-grass. The implications of this finding will be discussed further in the final section of the report.

Part IV: Model Evaluation

F1a: Stopping Criteria Impact

Using stopping criteria was crucial in training the model using the right number of epochs, without overtraining. As discussed above, I set the training and validation loop to a maximum of 75 epochs, with a stopping criterion of 7. The model was trained for 60 epochs

before hitting that criterion. The screenshot below shows the summary of the last 10 epochs of training:



The chart below shows the training loss, validation loss, and accuracy of the last 10 epochs. If a smaller patience value were used, the training would have ended earlier, with a greater loss and lower accuracy for the model. In fact, the patience counter hit 6 before resetting to zero again, twice. If the patience had been 6, training would have ended before the model achieved 80% accuracy, but with a value of 7, it was able to achieve over 85% accuracy. It is possible that even greater accuracy could be achieved with a larger patience value, but that needs to be weighed against training resources. As is, the training took over 2 hours, with accuracy improvements getting smaller and smaller as time went on. Thus, I feel comfortable with the stopping criteria used.

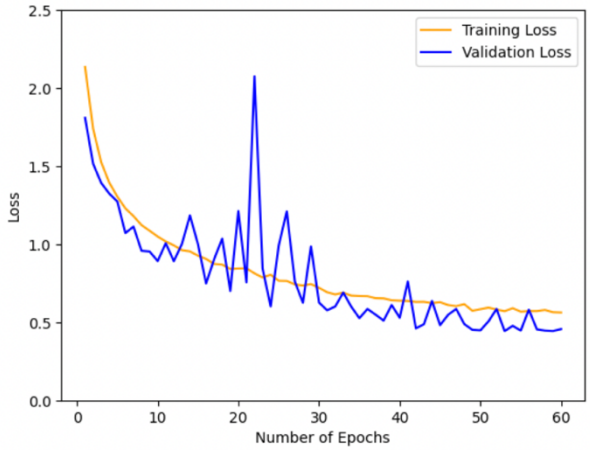
50	51.0	0.593228	0.505419	83.259259
51	52.0	0.581176	0.584625	77.481481
52	53.0	0.570677	0.443062	84.592593
53	54.0	0.589615	0.476802	81.925926
54	55.0	0.567713	0.446564	84.000000
55	56.0	0.571214	0.579962	79.555556
56	57.0	0.571242	0.454535	84.444444
57	58.0	0.578466	0.445928	85.333333
58	59.0	0.563941	0.443255	84.888889
59	60.0	0.561954	0.456198	84.592593

F1b: Evaluation Metrics

```
val_accuracy = 100.0 * correct / total
```

I used accuracy to evaluate the trained model's performance on the validation data. Accuracy represents the percentage of predictions that match the true label on the validation dataset. The best model had a validation set accuracy of 85.3%, meaning that it made a correct prediction about 85% of the time, and made an incorrect prediction about 15% of the time on the validation dataset.

F1c: Visualization



This visualization compares the training loss and the validation loss at each epoch of training. It shows that the training loss dropped rapidly across the first 10-20 epochs before starting to plateau. The validation loss was much more erratic, showing large fluctuations until it somewhat stabilized around 30 or 40. To avoid overfitting, it is important to stop training before the validation loss minimum. This chart suggests that the stopping criteria were effective in stopping the training before the validation loss rose again after epoch 60.

F2: Model Fitness

Overfitting happens when a model learns the training data so well that it performs poorly on new data inputs. It learns both the meaningful features as well as any noise that is in the training data, which therefore leads to decreased performance. Underfitting, on the other hand,

happens when a model is too simple. If a model is too simple, it is not able to learn the complex patterns within the training data, which also leads to poor performance.

In this model, I tried to avoid underfitting by focusing on the complexity of the model. When defining the architecture of the model, I used three convolutional blocks to make sure that the model had the opportunity to learn complex patterns. A simpler model could have been developed with only one convolutional block, but that would likely have led to underfitting due to the simplicity of the model.

I used several different strategies to avoid overfitting in the preprocessing, model creation, and training phases. First, I tried to balance out the dataset by increasing the number of images for the least-represented labels. This helps to avoid the model from over-predicting the most-represented class. Next, I augmented the training images using spatial transformations to include more variety in the training images.

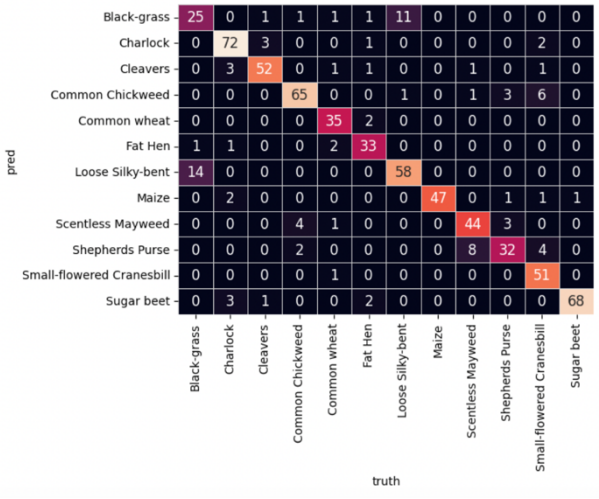
In model creation, I used a Dropout layer in the fully-connected layer to reduce the potential for overfitting. This layer turns off 30% of neurons during training, which requires the model to develop varied pathways and not rely too heavily on specific features. During training, I used stopping criteria that focused on the validation loss minimum to determine the best point to stop training, before running the risk of overtraining.

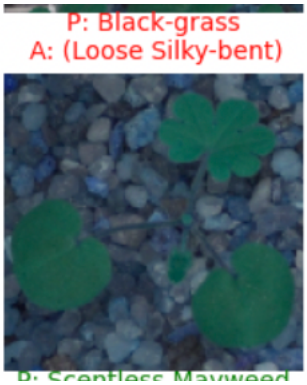
F3: Predictive Accuracy

```
Testing: 100%|████████████████████████████| 11/11 [00:07<00:00,  1.56it/s]Test Loss: 0.431558

Test Accuracy: 86% (582/675)
```

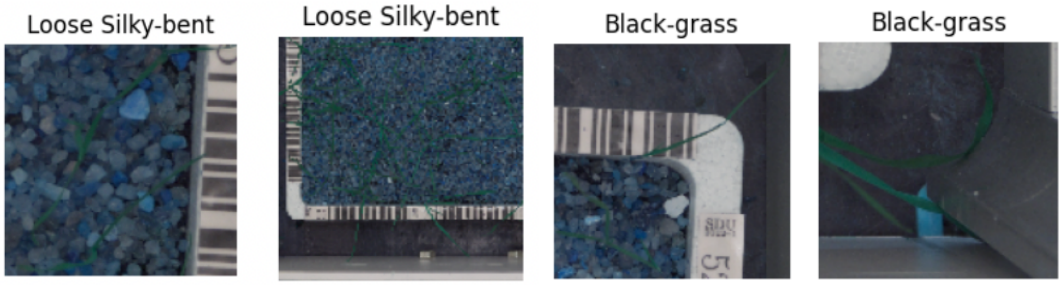
The best model achieved an 86% accuracy rate in classifying images from the test set, with 582 images classified correctly out of the total test set of 675 images. As discussed previously, the confusion matrix shows that the model performed very well across all classes

except for one exception: It tended to make mistakes in distinguishing between Loose Silky-bent and Black-grass.





This image shows one example of this mis-identification in which the model predicted Black-grass for a sample that was actually Loose Silky-bent. The images below show that both species have a similar, grass-like structure, which may make them harder to distinguish from some of the other, broader-leafed species.



Part V: Summary and Recommendations

G1: Code

```
# Save best model
torch.save(model.state_dict(), 'best_model.pth')
…
# Load best model
model.load_state_dict(torch.load('best_model.pth'))
```

After training, the learned parameters of the model, including the weights and biases, are saved. Those parameters are then reloaded before evaluating the model using the testing set data.

G2: Neural Network Functionality

This trained neural network functions by taking an input image, running it through a series of layers, and outputting a prediction of which of 12 specific plant species it is most likely to be. Breaking it down into more detail, this is how the trained network works on a new image:

1. A color image with a shape of 3 channels x 128 pixels x 128 pixels is used as input

2. The image data is passed through three locally-connected blocks of convolutional layers, batch normalization, and ReLU activation functions, and max pooling, during which the input is transformed to have more depth (more channels) and smaller size (in terms of the pixel grid)
    a. The convolutional layers use a series of filters, trained with training data, that pick up complex patterns in the image, which are represented by feature maps
    b. The batch normalization layers normalize the data at each step
    c. The ReLU activation functions introduce non-linearity, and only allow the positive activations to pass through to the next layer
    d. The max pooling layers reduce the spatial size of each feature map, which helps with the generalization of patterns

3. Next, the data is flattened from a 4D tensor into a one-dimensional vector, which becomes the input for the last block of fully-connected layers.

4. The final pooling layer then reduces the spatial size of each feature map

5. The dropout layer was used to deactivate a portion of the neurons during each training epoch, which reduced overfitting to the training data

6. Finally, the linear layer maps the features extracted from the input layer to make a prediction of one of the 12 plant species classes. It does this by outputting a probability that the image belongs to each class (a float between 0 and 1).

G3: Business Problem Alignment

The original research question was: “To what extent can a neural network accurately distinguish between common crop and weed seedlings?”. Test results show that the model has an accuracy rate of 86% overall on test data across all species, and the majority of errors come from misidentifications between Loose Silky-bent and Black-grass.

If the primary goal is to differentiate between crop species and weed species for making plant care decisions, then the functional accuracy is much higher than 86%. A quick Google search shows that of the 12 plant species in the dataset, only two are crops (Maize and Common Wheat), and the other 10 are all weeds. With this in mind, the confusion matrix can be reorganized to analyze performance across Common Wheat, Maize, and all weed species aggregated together:

Accuracy by Plant Type (Weed vs. Crop)

PREDICTION			
Weed Species	580 (98.9%)	6 (14.6%)	0 (0.0%)
Common Wheat	2 (0.3%)	35 (85.4%)	0 (0.0%)
Maize	5 (0.8%)	0 (0%)	47 (100.0%)
	Weed Species	***Common Wheat***	***Maize***
	TRUTH		

With this organization, we can see that the model has 98.9% accuracy in identifying weed samples as a weed species, and 100% accuracy in identifying Maize samples. It is weaker in identifying Common Wheat samples, at 85%.

G4: Model Improvement

The main lessons learned in this project relate to evaluating the quality of datasets and the impact of preprocessing on the accuracy of the final model. My first attempt at creating and training the model led to a test accuracy of only 14%. Increasing the size of the data set to address the under-representation of several plant species started to improve the accuracy metric. In addition, making careful decisions about how to augment the training images was essential. Early attempts included transforming aspects related to color and contrast, but these transformations led to poorer results. In previewing some of the images in the dataset, some are very difficult to see due to low contrast, and randomly transforming the hue and contrast likely

made it impossible for the model to learn meaningful features. In contrast, keeping only the spatial transformations improved the model's accuracy.

To improve this model, the main change I would make is to improve the input dataset. The provided data set was unbalanced, with the two crop species having the smallest number of samples. My solution of duplicating images to address the imbalance was not an ideal approach, and was just a proxy for sourcing more real images in a real-world context, given the constraints of this project. In working with the agricultural company, I would explain the importance of quality input images on final model accuracy, and would encourage them to provide more images of Maize and Common Wheat, to ensure that all images are visually clear and meet a minimum baseline quality (especially with the grass-like seedlings), and that there is a minimum number (at least 500) of each individual plant species.

G5. Recommended Course of Action

Based on the aggregate confusion matrix above, I recommend that the model can be used as-is for identifying weed seedlings in the context of Maize farming, since it has 100% accuracy in identifying Maize samples and close to 100% accuracy in identifying the various weed samples as a weed species. Used in this context, the misidentifications between two weed species, Loose Silky-bent and Black-grass, are irrelevant, because both weeds would be treated in the same way (pulled, eliminated with pesticide, etc.).

I do not yet recommend the use of the model to identify weeds in the context of Common Wheat farming. 85% accuracy (true positives) in the identification of the crop is too low, and I would feel more confident in its use if the accuracy were in the mid-nineties. In order to improve the model for the use in wheat farming, I recommend adding additional images of Common

Wheat, as well as Loose Siky-bent and Black-grass, so that the model has more high-quality content to learn from to get better at these specific classifications.

Part VI: Reporting

H: Output

A PDF of the Jupyter notebook is included with the submission.

I. Sources for Third-Party Code

The code for creating the CNN architecture and the training and validation loop was strongly influenced by and adapted from the following course:

Udacity. (n.d.). *Convolutional neural networks.* https://learn.udacity.com/paid-courses/cd1821

The following PyTorch documentation resources were used to develop new code, troubleshoot and debug, and better understand how the code works:

torch.utils.data — PyTorch 2.7 documentation. (2024). Pytorch.org. https://docs.pytorch.org/docs/stable/data.html

torch.nn — PyTorch 2.7 documentation. (2024). Pytorch.org. https://docs.pytorch.org/docs/stable/nn.html

torch.optim — PyTorch 2.7 documentation. (2024). Pytorch.org. https://docs.pytorch.org/docs/stable/optim.html

torchvision.transforms — Torchvision 0.8.1 documentation. (2017). Pytorch.org. https://docs.pytorch.org/vision/0.8/transforms.html

torch-summary. (2020, December 24). PyPI. https://pypi.org/project/torch-summary/

J: Sources

No content was directly quoted, paraphrased, or summarized in developing this project.