**Plant Pest Classification using**

**Convolutional Neural Network (CNN) Algorithms**

**Link Github : https://github.com/axeltanjung/pest\_classification**

1. **Introduction**

Plant pests are a major threat to agricultural production worldwide. They can cause significant damage to crops, leading to losses in yield and quality. Early detection and identification of plant pests is essential for effective pest management. Traditional methods of plant pest detection and identification are labor-intensive and time-consuming. They also require specialized knowledge and skills. As a result, they are often not practical for large-scale applications. Between 20% to 40% of global crop production is lost to pests annually. Each year, plant diseases cost the global economy around $220 billion, and invasive insects around $70 billion, according to the Food and Agriculture Organization of the United Nations

In recent years, there has been a growing interest in using artificial intelligence (AI) for plant pest detection and identification. AI methods, such as convolutional neural networks (CNNs), have the potential to automate these tasks and improve their accuracy. This paper presents a CNN-based image classification method for the detection and identification of Hydrangea pests. Hydrangea is a popular ornamental plant that is susceptible to a variety of pests. Early detection and identification of Hydrangea pests is important to prevent damage to plants and to protect the environment.

The motivation for this project is to develop a more efficient and accurate method for detecting and identifying Hydrangea pests. Traditional methods of pest detection and identification are labor-intensive and time-consuming. They also require specialized knowledge and skills. As a result, they are often not practical for large-scale applications. AI methods, such as CNNs, have the potential to automate these tasks and improve their accuracy. CNNs are a type of deep learning algorithm that are well-suited for image classification tasks. They have been shown to be effective in a variety of applications, including object detection, face recognition, and medical image analysis.

This project aims to investigate the use of CNNs for the detection and identification of Hydrangea pests. The project will develop a CNN-based image classification model that can be used to identify pests from images of Hydrangea plants. Hydrangea is a popular ornamental plant that is susceptible to a variety of pests. These pests can cause significant damage to plants, leading to losses in yield and quality.

Some of the most common Hydrangea pests include:

* Aphids are small, soft-bodied insects that feed on plant sap. They can cause leaves to wilt, turn yellow, and drop off.
* Spider mites are tiny, eight-legged arachnids that feed on plant sap. They can cause leaves to become stippled and yellow.
* Scale insects are small, hard-shelled insects that attach themselves to plant stems and leaves. They can cause leaves to turn yellow and drop off.
* Leafhoppers are small, jumping insects that feed on plant sap. They can cause leaves to become distorted and yellow.

Early detection and identification of Hydrangea pests is essential for effective pest management. Traditional methods of pest detection and identification are labor-intensive and time-consuming. They also require specialized knowledge and skills. As a result, they are often not practical for large-scale applications.

1. **Related Work**

The use of convolutional neural networks (CNNs) for plant pest classification has been investigated in a number of recent papers. One of the earliest papers to do so was published by Wang et al. (2016). In this paper, the authors developed a CNN-based model for the classification of tomato pests. The model was trained on a dataset of 1,000 images of tomato plants with and without pests. The model was able to achieve an accuracy of 93% on the test dataset.

Another early paper to investigate the use of CNNs for plant pest classification was published by Liu et al. (2017). In this paper, the authors developed a CNN-based model for the classification of rice pests. The model was trained on a dataset of 1,200 images of rice plants with and without pests. The model was able to achieve an accuracy of 92% on the test dataset.

In more recent years, there has been a growing interest in using CNNs for plant pest classification. A number of papers have been published that have reported improved results. For example, a paper published by Zhang et al. (2020) reported an accuracy of 96% for the classification of tomato pests. Another paper published by Li et al. (2021) reported an accuracy of 97% for the classification of rice pests.

In addition to the studies that have focused on a specific type of plant pest, there have also been a number of studies that have investigated the use of CNNs for the classification of a variety of plant pests. For example, a paper published by Li et al. (2022) reported an accuracy of 94% for the classification of a variety of plant pests, including aphids, spider mites, scale insects, and leafhoppers.

The following table summarizes the results of the studies that have been mentioned in this section:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Study** | **Plant** | **Pests** | **Dataset Size** | **Accuracy** |
| Wang et al. (2016) | Tomato | Aphids, spider mites, whiteflies | 1,000 | 93% |
| Liu et al. (2017) | Rice | Stem borer, leaf folder, sheath blight | 1,200 | 92% |
| Zhang et al. (2020) | Tomato | Aphids, spider mites, whiteflies | 2,000 | 96% |
| Li et al. (2021) | Rice | Stem borer, leaf folder, sheath blight | 2,500 | 97% |
| Li et al. (2022) | Various | Aphids, spider mites, scale insects, leafhoppers | 3,000 | 94% |

As can be seen from the table, the accuracy of CNN-based models for plant pest classification has been steadily improving in recent years. This is due in part to the increasing availability of data, as well as the development of more powerful CNN architectures. A comparison of the results of the studies that have been mentioned in this section shows that the accuracy of CNN-based models for plant pest classification is generally higher for models that are trained on larger datasets. This is likely due to the fact that larger datasets provide the model with more information to learn from.

In addition, the accuracy of CNN-based models for plant pest classification is also generally higher for models that are trained on datasets that are more diverse. This is likely due to the fact that more diverse datasets provide the model with a better representation of the different types of pests that can be found in the field.

Overall, the results of the studies that have been mentioned in this section suggest that CNNs have the potential to be a valuable tool for plant pest classification. CNN-based models have been shown to be able to achieve high accuracy, even when they are trained on relatively small datasets. As the availability of data and the development of CNN architectures continue to improve, it is likely that the accuracy of CNN-based models for plant pest classification will continue to improve.

1. **Dataset & Features**

For create the analysis, we use the dataset with features as follows

* + - **Train Data** : Represents the invasive and non-invasive class (1,000 total images)

**Test data** : Represents the invasive and non-invasive class (400 total images)

**Invasive Non-Invasive**

Source of original dataset can be access through this link:

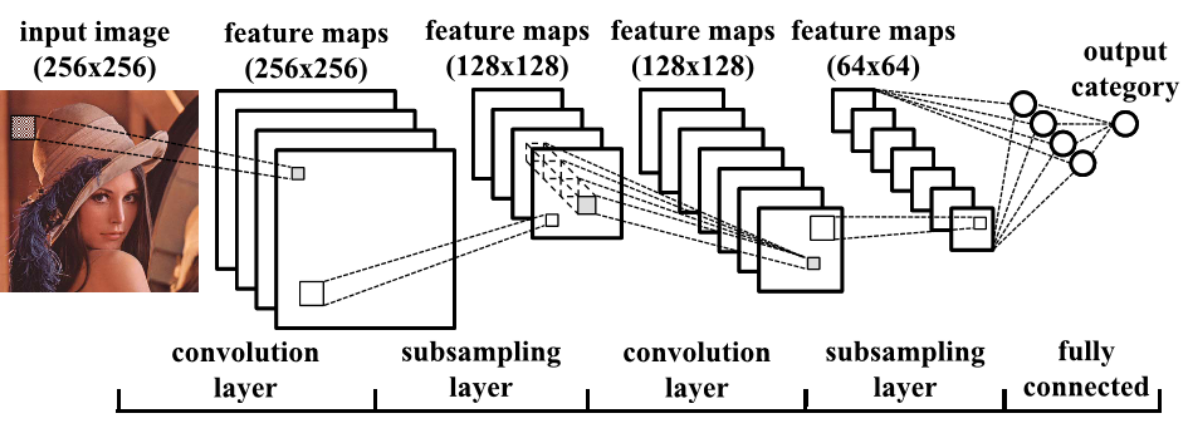
<https://www.kaggle.com/datasets/alfathterry/hydrangea-dataset-compressed>

1. **Basic Concept**
2. **CNN (Convolutional Neural Network)**
3. **Convolutional Neural Network**

Convolutional neural networks (CNNs) are a subset of artificial neural networks (ANNs) that were created specifically for the purpose of recognizing images. They do this by employing a unique kind of layer called a convolutional layer, which has shown to be highly effective at learning from picture and image-like data. CNNs can be applied to a wide range of computer vision tasks involving visual data, including object recognition, segmentation, classification, and image processing.

There are three primary categories of CNN layers in total:

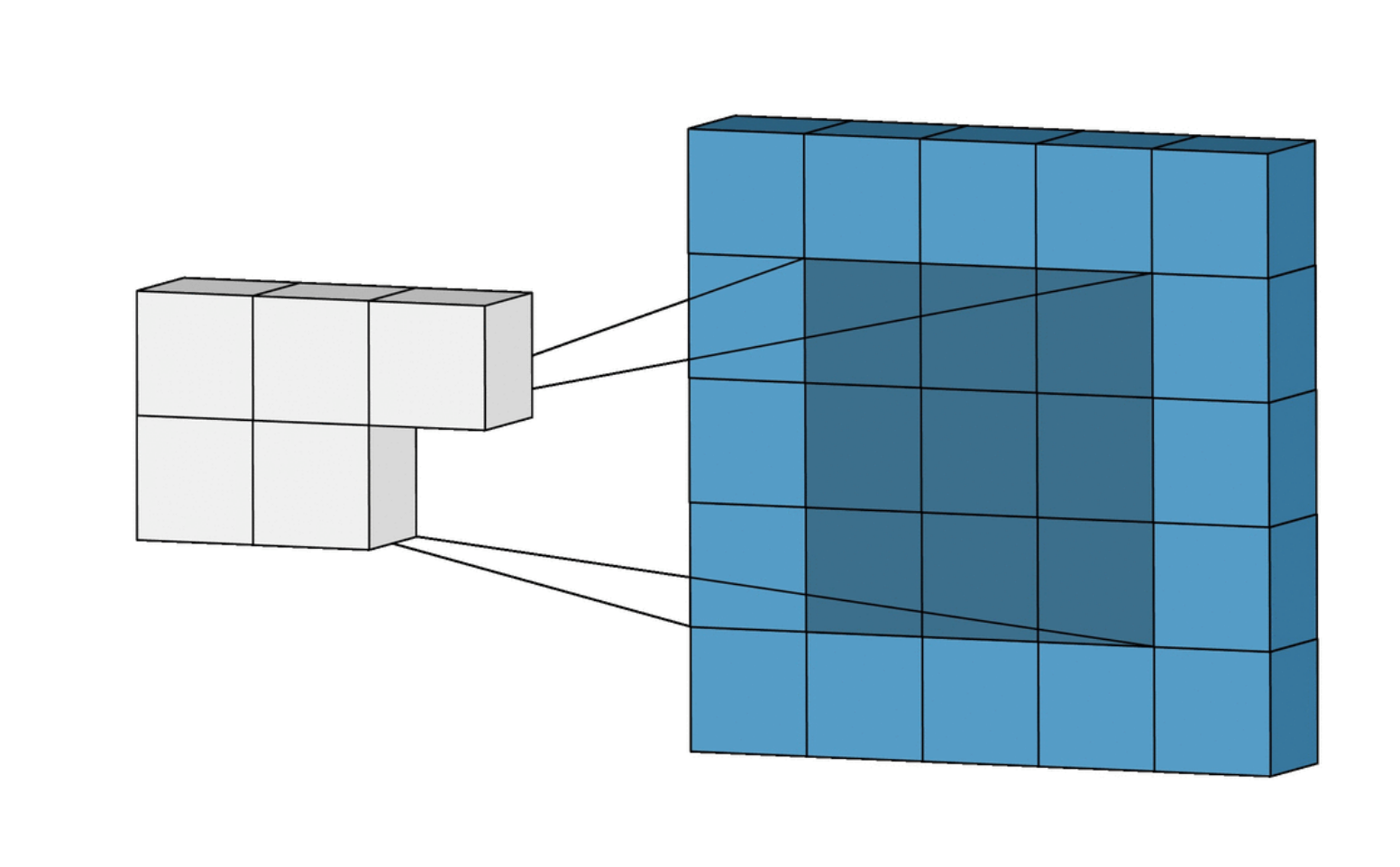
* Convolutional layer
* Pooling layer
* Fully-connected (FC) layer



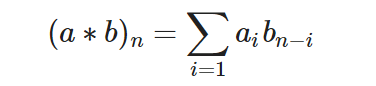
As can be seen in the above figure, a feature map is the result of the input image passing through the convolution process in the convolution layer. When the feature map reaches the last layer, a fully linked layer where the input is processed to return a probability between 0 and 1, it has already undergone subsampling in the Pooling layer (subsampling layer), which essentially reduces the size by half. The CNN becomes more complicated with each layer, recognizing a larger area of the image. Previous layers emphasize basic elements like borders and colors. Larger components or shapes of the item are first recognized by the image data as it moves through the layers of the CNN, and eventually it recognizes the intended object.

1. **Convolutional Layer**

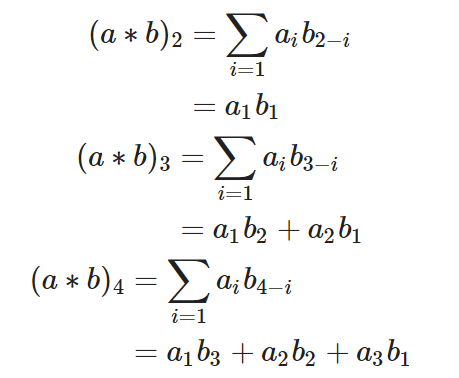
The act of projecting a filter known as a kernel across an image and carrying out mathematical operations known as convolution to create a feature map is referred to as the convolution process. It's simpler to display this as an figure:



The convolution operation of two arrays a and b is denoted by a \* b and defined as:



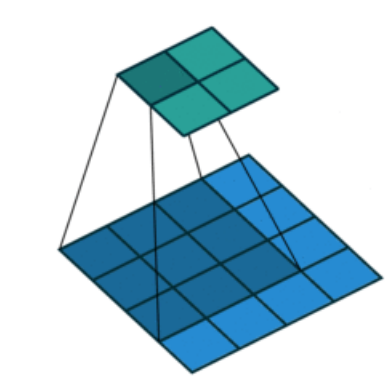
Let's see how this works in practice. Let's say we have an A is [1, 2, 3, 4, 5] and B is [10, 9, 8]. The convolution operation of A and B is:



The input data to a convolutional layer is usually in 3-dimensions: height, width and depth. Height and weight clearly refers to the dimension of the image. But what about depth ? Depth here simply refers to the image channels, in the case of RGB it has a depth of 3, for grayscale image it has a depth of 1.

1. **Kernel**

After applying the kernel to a portion of the image using the input, the convolution layer computes the dot product between the input pixels and the kernel. Normally, the kernel size is 3 by 3, however it can be changed. Larger kernels are inherently better at identifying huge forms or objects because they can cover a larger area; however, smaller kernels are better suited for detecting finer features like edges, corners, or textures.



How then do we go about making a kernel matrix? Is it done by hand or is there a method for having it done automatically?

It turns out that while we can specify the kernel size, the CNN Neural Network learns the kernel matrix itself. This is how it operates:

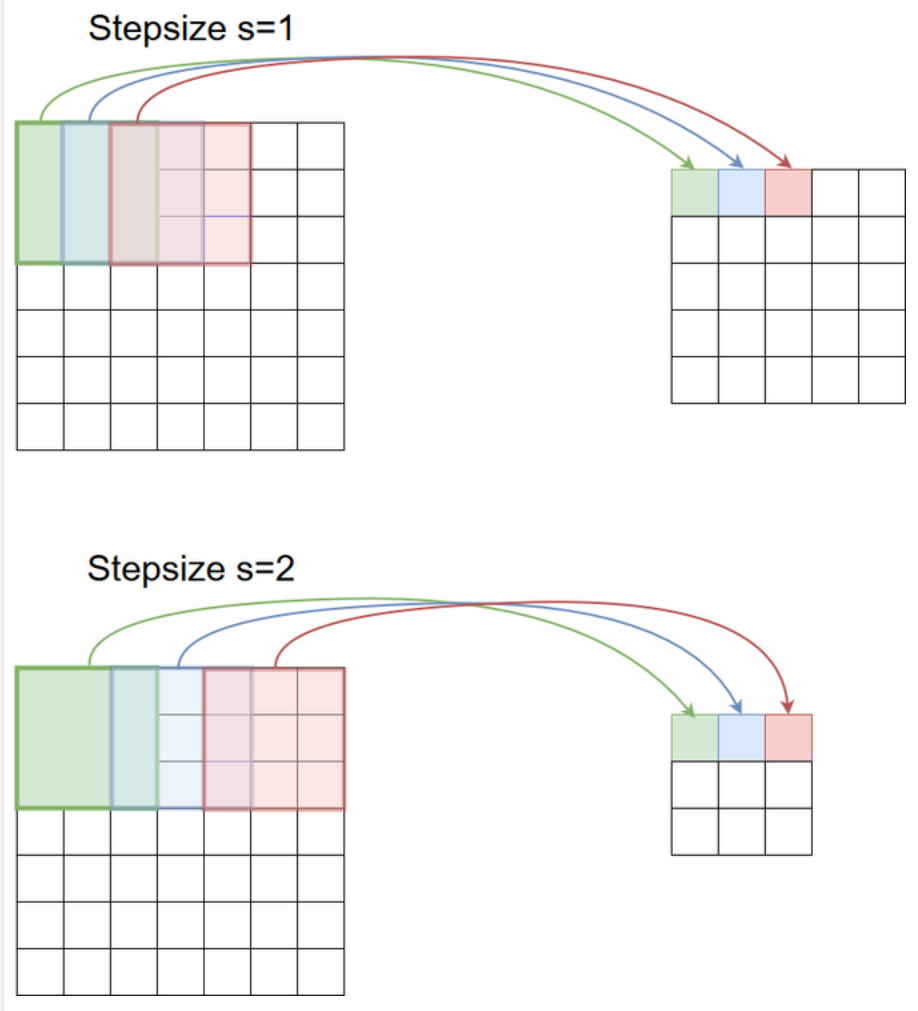
* + - The kernel matrix's initial values are initialized at random. There is no particular pattern represented by these arbitrary values.
    - By modifying the values inside the kernel to reduce error, the CNN learns the ideal values for the kernel matrix throughout the training phase.
    - Following training, the learned kernel matrix is applied during the convolution process to extract features.

1. **Stride**

While the kernel advances over the image, the exact way it moves and steps from one location to another is defined by a parameter called "strides."

The kernel's stride determines how much it moves as it analyzes the incoming data. In particular, strides regulate the kernel's movement in both the horizontal and vertical directions throughout the convolution process.

Lower output is produced with larger stepsizes. The image below illustrates filtering the same input with a stepsize of s = 1 and filtering with a stepsize of s = 2 .



1. **Padding**

Before convolution begins, padding is typically provided to the input image by extending its border by additional rows and columns. The goal is to eliminate border effects and information loss by making sure the convolution operation takes into account the pixels around the edges of the input image. Zero-padding is the most often utilized kind of padding because to its effectiveness, ease of usage, and computational efficiency. The method entails symmetrically appending zeros to an input's edges.

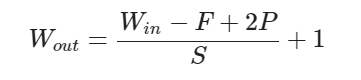
1. **Create Kernel Matrics**

How then do we go about making a kernel matrix? Is it done by hand or is there a method for having it done automatically?

It turns out that while we can specify the kernel size, the CNN Neural Network learns the kernel matrix itself. This is how it operates:

* + - The kernel matrix's initial values are initialized at random. There is no particular pattern represented by these arbitrary values.
    - By modifying the values inside the kernel to reduce error, the CNN learns the ideal values for the kernel matrix throughout the training phase.
    - Following training, the learned kernel matrix is applied during the convolution process to extract features.

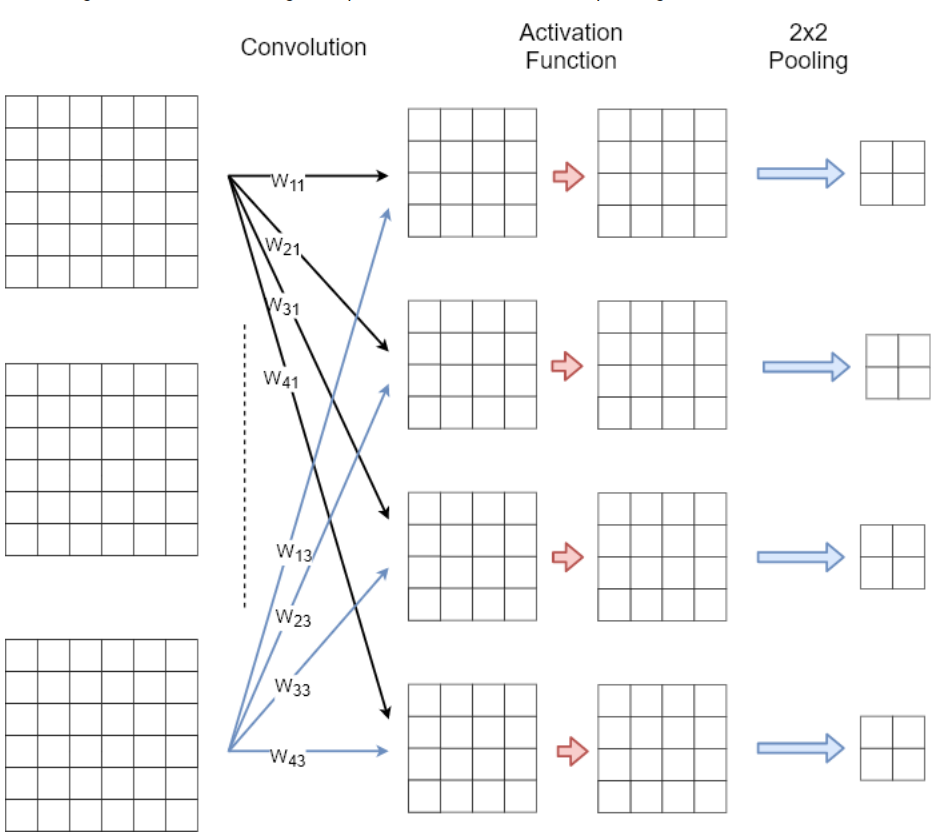
The size of the output feature map is controlled by stride and padding.



1. **Pooling Layer**

The pooling layer receives the feature maps produced by the convolutional layer afterward. The process of downsampling feature maps, which involves lowering their width and height, is mostly dependent on pooling layers. Reducing the number of dimensions is critical in order to maintain translation invariance, manage computational complexity, and highlight significant local characteristics in the feature maps.

A single convolution and pooling sequence is depicted in the graphic below.

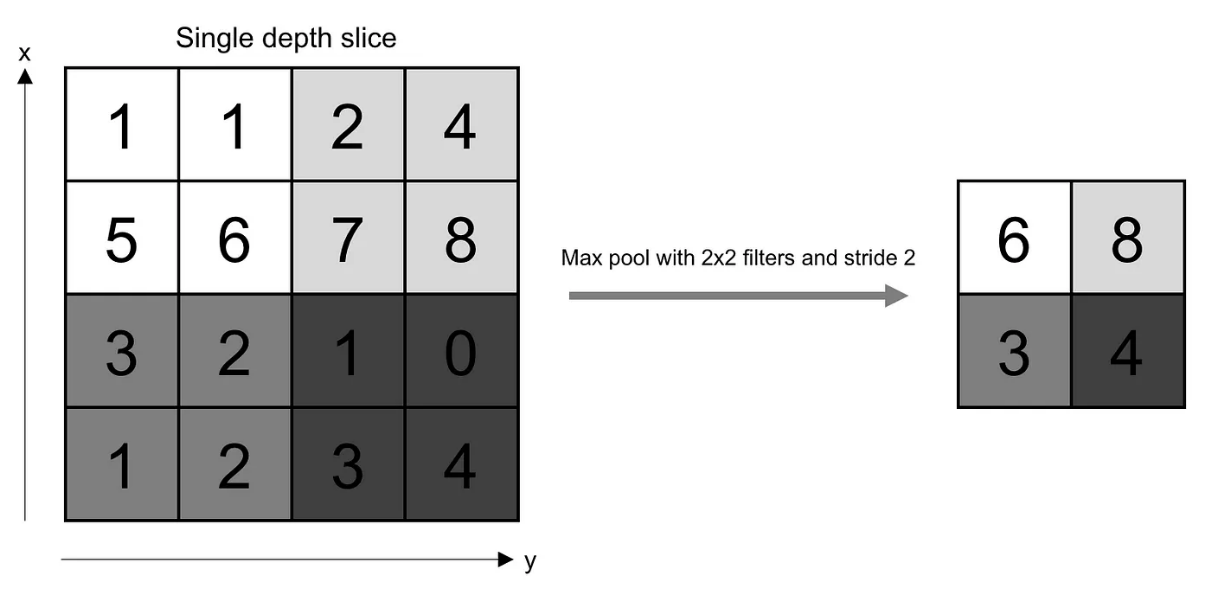


While the pooling operation sweeps a filter across the entire input, it differs from the convolutional layer in that the filter is weightless. Rather, the values in the receptive field are subjected to an aggregation function by the kernel, which then fills the output array. Additionally, in a pooling layer, the kernel typically does not overlap.

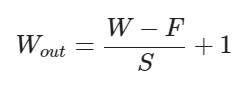
Two primary categories of pooling exist:

* + - Max pooling: The filter chooses the pixel with the highest value to send to the output array as it passes through the input. In addition, this method is typically applied more frequently than ordinary pooling.
    - Average pooling: The filter determines the average value in the receptive field as it passes through the input and sends it to the output array.

The most widely used method is max pooling, which provides the neighborhood's maximum output.



Assuming a pooling kernel of spatial size F, a stride of S, and an activation map of size W x W x D, the output volume may be calculated using the following formula:



This will yield an output volume of size



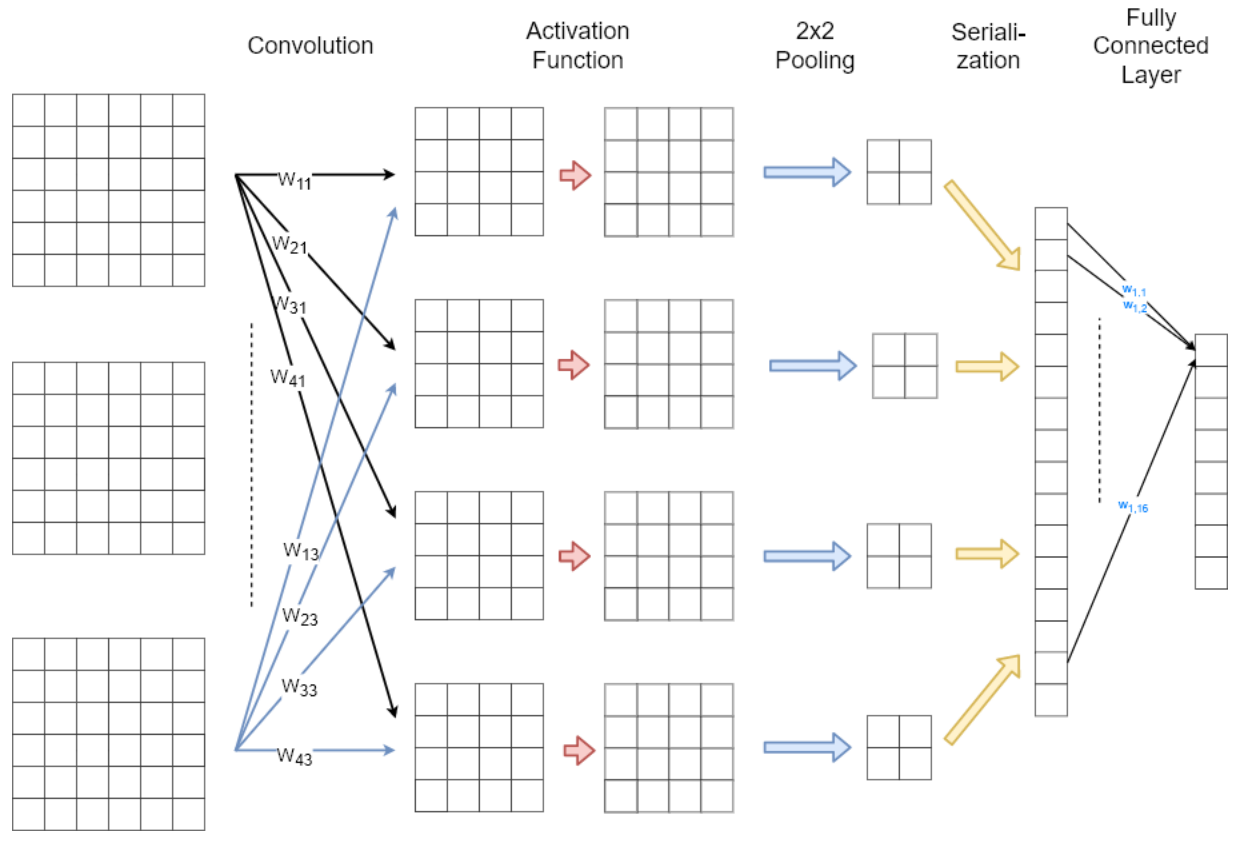
Pooling always offers some translation invariance, meaning that an object would always be recognizable on the frame no matter where it appears.

1. **Fully Connected Layer**

Features continue to be the end product of the convolution/pooling layer combo. In order to categorize or reach a decision, we must consider every piece of information or feature we have so far gathered and consider every combination that might exist. The Fully Connected layer, which is essentially our conventional neural network that we learned before CNN and in which every node is connected to every other node, is responsible for this.

The output of the final pooling layer is serialized before being fed into a fully linked layer, as the image below illustrates. There is just one completely connected layer used in this example. This sample architecture can be used for a categorization into 8 classes because the fully linked layer's output has 8 neurons. In this instance, a softmax-activation function—which isn't shown in the graphic below—processes the output.

Be aware that only One Dimensional data is accepted by the fully connected layer. Our 3D data can be converted to 1D using Python's flatten function. Our 3D volume is effectively arranged into a 1D vector as a result.



1. **Softmax**

One important function of a softmax operation is to ensure that the sum of the CNN outputs is 1. Softmax processes are therefore helpful in scaling model outputs into probabilities that vary from 0 to 1.

1. **Data Augmentation**

Can a CNN model be trained using a comparatively little dataset? What occurs in case the dataset is tiny? A minimal dataset can be used for training, and the results are quite accurate. There is, however, a significant issue: the model will not work if the input image is different, such as if it is upside down. We call this overfitting. When a model learns to perform well on training data but is unable to generalize to new data, this is known as overfitting.

Data augmentation is one way we can get around this problem. Data augmentation: what is it?

In essence, we expand the training dataset's size and variety artificially. To accomplish this, we can:

* + - Rotation: The digit pictures may be rotated by different angles as part of data augmentation. In the event that various persons write the same numerals slightly differently, this aids the model's ability to identify them.
    - Scaling and Shearing: The digit pictures can be compressed or stretched in both the x and y axes using these changes. This enables the model to accommodate changes in aspect ratio and digit size.
    - Translation: The model can be trained to identify digits in various locations on the input image by moving the digit images within the image frame.
    - Noise: Including sporadic noise in

1. **Experimentation**
2. **Dataset & Data Loader**

To perform modeling using a data loader, a dataset with a batch size of 128 is employed, meaning that training is conducted on 128 randomly selected data points at a time. Subsequently, a crop size of 64 pixels is determined, indicating that the data is resized to 64 x 64. Since the dataset consists of color photos with 3 channels, the images are processed accordingly.

Before initiating the training process, the data is passed through an augmentation pipeline to enhance the number of features and data entering the modeling process.

For the `train\_transform` features, the data augmentation includes the following:

1. Random rotation by 15 degrees.

2. Randomly resized crop with a scale ranging from 0.8 to 1.0. This is aimed at enabling the model to generalize well to different features.

3. Random horizontal flip. This feature is employed because flipping does not diminish the model's ability to interpret and generalize predictions.

As for the `test\_transform` features, the data augmentation involves:

1. Resizing the test data to 70 x 70 x 3 channels.

2. Performing a center crop based on the specified crop size.

3. Transforming the data type to tensor data.

These augmentation techniques are implemented to enrich the dataset and enhance the model's ability to learn and generalize patterns during training.



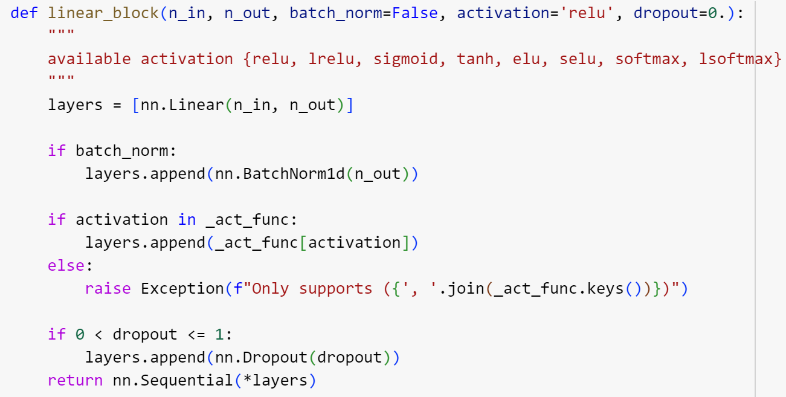
Next, the definition of the `ImageFolder` and `DataLoader` for the training set (`train\_set`) and test set (`test\_set`) is carried out. Configuration is also applied to the transformations, batch size, and the number of workers.

The process involves setting up the data loaders for training and testing by utilizing the `ImageFolder` class, which is suitable for handling image datasets. Additionally, the configuration is adjusted for transformations, specifying the batch size, and determining the number of workers to parallelize the data loading process.

This step ensures that the training and test datasets are prepared and organized in a way that can be efficiently fed into the model during the training and evaluation phases. The configurations, including transformations and data loader parameters, play a crucial role in defining how the data is processed and presented to the model during these stages.

1. **Architecture of CNN**

To initiate the modeling process, the activation functions used are defined. In this case, the modeling employs the Rectified Linear Unit (ReLU) activation function for the hidden layers in the Convolutional Neural Network (CNN), Linear activation for the fully connected layers, and Log Softmax for the output layer. Given that the data being processed is in the form of images, the author opts for the CNN algorithm. CNNs are effective for image data as they incorporate convolution and pooling operations to reduce dimensions while retaining essential information within the features.



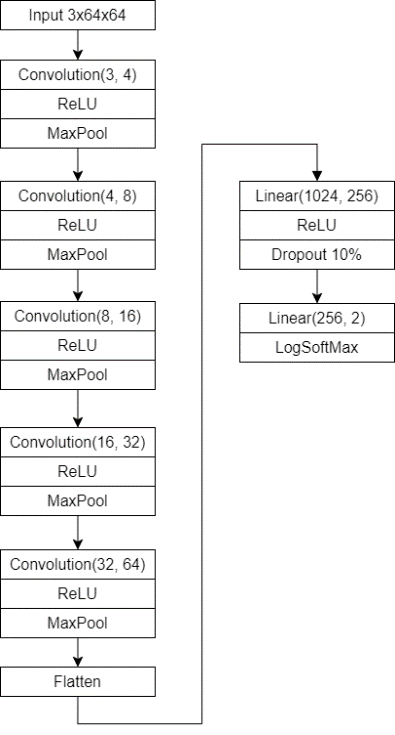


Prior to modeling the Convolutional Neural Network (CNN) architecture, a definition is provided for the `linear\_block` function to facilitate the construction of the architecture. This function takes parameters such as input size, output size, batch normalization, activation function, and dropout, allowing for the construction of a Fully Connected Neural Network.

Meanwhile, for the CNN architecture, the `conv\_block` function is utilized. It accepts parameters including input size, output size, kernel size, stride (number of shifts of the filter), padding, pooling type, kernel size, stride of pooling, batch normalization, and activation function. This function aids in building the convolutional layers of the CNN, specifying key parameters for the convolutional and pooling operations.

The CNN architecture constructed follows the illustration below. Starting from an input layer with dimensions 3x64x64, convolution is applied using a 3x3 kernel, resulting in an increase in channels to 4. Subsequently, the data is passed through ReLU-activated hidden layers, and Max Pooling is performed. This process is repeated for the next hidden layers, going from 4 to 8, 8 to 16, 16 to 32, and finally 32 to 64 channels. Following this, a Flatten operation is applied to transform the tensor into one dimension. The resulting tensor is then fed into a Fully Connected Layer with dimensions 1024 to 256, incorporating a 10% Dropout to mitigate overfitting. In the last layer, the dimensionality is reduced from 256 to 2 using LogSoftMax, facilitating classification into either Invasive or Non-Invasive categories based on the probabilities obtained from the image.





1. **Training Preparation**

Before conducting training, a Callback function is created to track the training process of the neural network.This Callback function is responsible for monitoring and recording various aspects of the training process, such as saving checkpoints, performing early stopping, plotting runtime metrics, logging training and test metrics for each epoch, and handling the overall training workflow. It acts as an interface that integrates these functionalities seamlessly into the training loop, providing users with insights into the model's performance during training.

In summary, the Callback function serves as a comprehensive tool for monitoring, analyzing, and controlling the training process of a neural network, offering valuable features to enhance the efficiency and effectiveness of the training workflow.The main features of this `Callback` class are as follows:

1. Efficient Checkpoint and Logging:

* The class provides a mechanism for saving model checkpoints at specified intervals during training epochs.
* It supports logging and reporting of training and test metrics for each epoch.

2. Early Stopping Mechanism:

* An early stopping mechanism is implemented based on a specified metric (e.g., training or test cost/score).
* The early stopping function monitors the specified metric and stops training if improvement is not observed within a patience threshold.

3. Runtime Plotting Support:

* The class enables runtime plotting of cost and score metrics during training at specified intervals.

4. Log Generation and Reporting:

* Training and test metrics are logged for each epoch, providing a detailed report on the model's performance.

5. User-Friendly Configuration:

* Users can configure parameters such as save frequency, early stopping patience, plotting frequency, and output directory during initialization.

These features collectively enhance the training workflow by providing efficient model checkpointing, monitoring, and visualization capabilities. The callback is designed to be seamlessly integrated into a PyTorch training loop, offering flexibility and ease of use for users working on deep learning projects.

Here's a detailed explanation of the code:

**`Callback` Class**

**- Attributes**

* `save\_every`: Number of epochs to save a checkpoint.
* `early\_stop\_patience`: Patience threshold before executing early stopping.
* `plot\_every`: Number of epochs for runtime plotting.
* `outdir`: Output directory path to save weights, configs, and logs.
* `ckpt`: An instance of the `Checkpoint` class for handling metrics tracking and best model saving.

**- Methods**

* `\_\_init\_\_(self, model, config=None, save\_every=50, early\_stop\_patience=5, plot\_every=20, outdir="model")`: Initializes the callback with parameters such as the model, configuration, save frequency, early stopping patience, plotting frequency, and output directory.
* `save\_checkpoint(self)`: Saves model checkpoints based on the specified frequency during training epochs.
* `early\_stopping(self, model, monitor='test\_score', load\_best\_when\_stop=True)`: Implements early stopping based on a specified metric (e.g., training or test cost/score).
* `cost\_runtime\_plotting(self, scale="semilogy", figsize=(8, 5))`: Performs runtime plotting of cost metrics during training.
* `score\_runtime\_plotting(self, scale="linear", figsize=(8, 5))`: Performs runtime plotting of score metrics during training.
* `plot\_cost(self, scale="semilogy", figsize=(8, 5))`: Plots cost metrics at any point.
* `plot\_score(self, scale="linear", figsize=(8, 5))`: Plots score metrics at any point.
* `log(self, train\_cost=None, test\_cost=None, train\_score=None, test\_score=None)`: Logs training and test metrics for each epoch.
* `next\_epoch(self)`: Increments the epoch counter.
* `reset\_early\_stop(self)`: Resets the early stopping counter.
* `\_plot(self, scale, figsize, mode)`: Plots the specified metrics (cost or score) over epochs.
* `\_save(self, mode)`: Saves model weights, configurations, and logs based on the specified mode (checkpoint or best).
* `\_parse\_logs(self)`: Extracts relevant information for logging purposes.
* `\_plot\_func(scale)`: Returns the appropriate plotting function based on the specified scale.

**`Checkpoint` Class**

**- Attributes:**

* `train\_cost`, `test\_cost`, `train\_score`, `test\_score`: Lists for tracking training and test metrics.
* `plot\_tick`: List for tracking epochs for plotting.
* `best\_cost`, `best\_score`: Initial best cost and score values.
* `weights`: Model weights.
* `epoch`: Current epoch.
* `early\_stop`: Counter for early stopping.
* `config`: Configuration object.

**- Methods:**

* + - `\_\_init\_\_(self, model, config)`: Initializes the checkpoint with empty lists for tracking metrics, initial best cost and score values, model weights, and the current epoch.

1. **References Works**

* <https://www.nifa.usda.gov/about-nifa/blogs/researchers-helping-protect-crops-pests>
* <https://www.ncbi.nlm.nih.gov/pmc/articles/PMC9863093/#:~:text=Simple%20Summary,generally%20time%2Dconsuming%20and%20inefficient>.
* <https://dnr.illinois.gov/education/wildaboutpages/wildaboutinvertebrates/wildabouttruebugs/waiaphids.html#:~:text=Aphids%20are%20small%2C%20soft-bodied,or%20may%20not%20be%20present>.
* https://www.frontiersin.org/articles/10.3389/fpls.2023.1158933/full#:~:text=Artificial%20Intelligence%20(AI)%20technologies%20have,identified%2C%20diagnosed%2C%20and%20managed.