Rapid Hurricane Damage Assessment using CNNs

Deborah Ugaldes and George Whittington

TABLE OF CONTENTS

01

Introduction

What is a convolutional neural network?

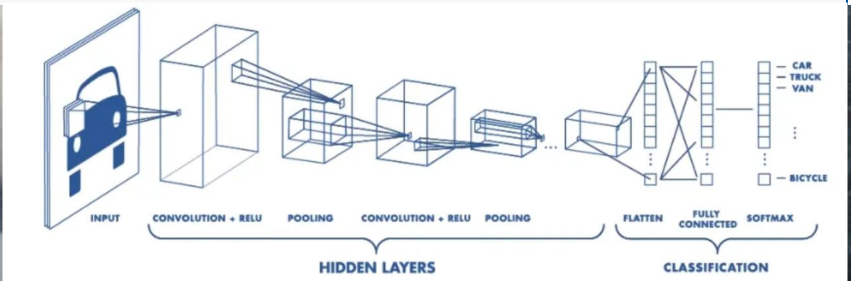
What is a convolutional neural network?

Neural networks are a subset of machine learning. They are made up of node layers, an input layer, some hidden layers, and an output layer.

Information goes in the first layer, is processed by the middle layers, and an answer is produced in the last layer. They learn by strengthening connections between the layers that lead to a predicated answer.

Convolutional neural networks (CNN) specializes in image, speech, or audio inputs. It includes convolutional layers, pooling layers, and fully-connected layers.

Each layer focuses on a specific feature *(such as colors or lines)*. As the image moves through the layers of the CNN, the network begins to recognize larger elements or shapes of the objects until it identifies the subject of the image.

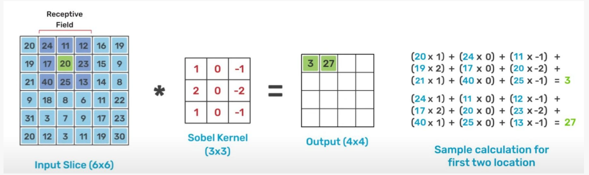


A Closer Look at CNN Layers

Convolutional Layer:

This is the core building block of a CNN. It's components include input data, a filter, and a feature map.

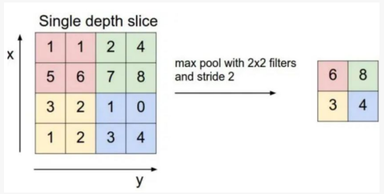
We used a 3x3 kernel *(filter)* that moved across the image checking if a feature was present. It applies a dot product between the input pixels and the filter which is sent to the output array.


Receptive Field
20 24 11 12 16 19
19 17 20 23 15 9
21 40 25 13 14 8
9 18 8 6 11 22
31 3 7 9 17 23
20 12 3 11 19 30
Input Slice (6x6)
1 0 -1
2 0 -2
1 0 -1
Sobel Kernel (3x3)
3 27
Output (4x4)
(20 x 1) + (24 x 0) + (11 x -1) + (19 x 2) + (17 x 0) + (20 x -2) + (21 x 1) + (40 x 0) + (25 x -1) = 3
(24 x 1) + (11 x 0) + (12 x -1) + (17 x 2) + (20 x 0) + (23 x -2) + (40 x 1) + (25 x 0) + (13 x -1) = 27
Sample calculation for first two location

Pooling Layer:

This can also be known as downsampling. This point of this layer is to reduced the number of parameters in the input.

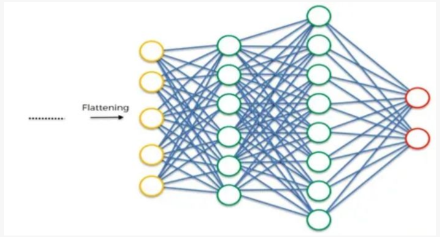
We utilized max-pooling, we used a 2x2 filter that moves across the image It selects one pixel out of the group to move onto the output layer.



Fully-Connected Layer:

This layer takes the "building blocks" from each convolutional layer and strings them together into one piece of information.

This layer essentially makes the final decision as to what the models believes the image to be.

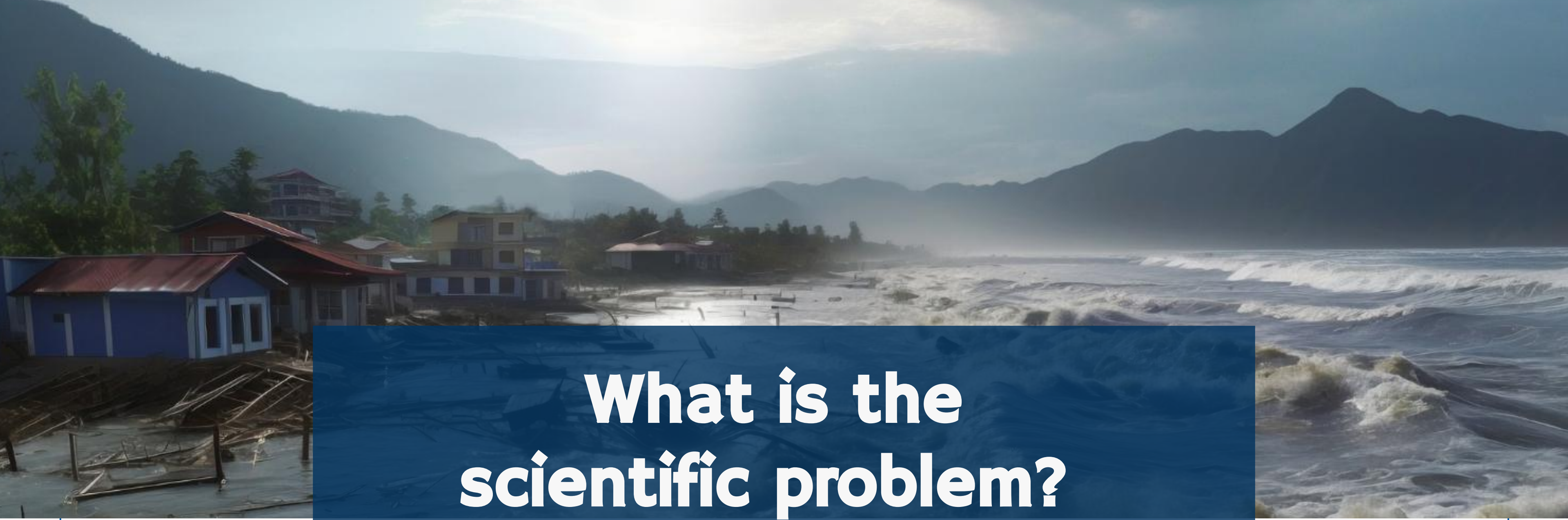


The Prevalence of Hurricanes

- Hurricanes are among the **most devastating** natural disasters, causing extensive damage to infrastructure, displacing communities, and requiring rapid response efforts.

- The intensity, frequency, and duration of North Atlantic hurricanes have been **increasing** since the early 1980s.

- The contributions of human and natural causes to these increases and still unknown. Hurricane-associated storms are projected to **increase** as the climate continues to warm.



What is the scientific problem?

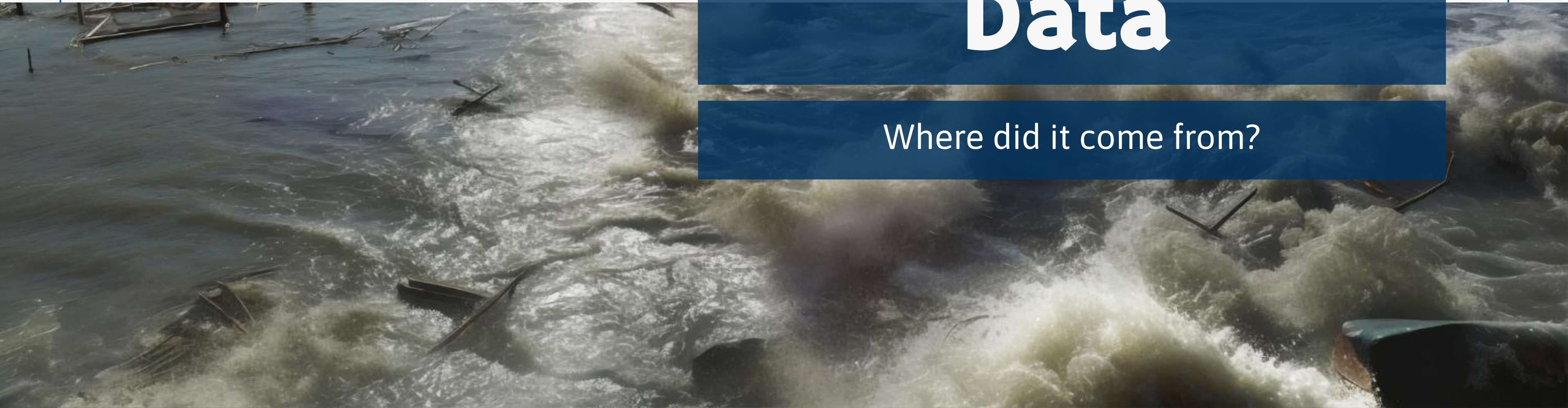
Efficient damage assessment is essential for emergency managers to allocate resources effectively and provide timely assistance to affected areas. Traditionally damage assessment includes time and labor intensive ground surveys to record the number of flooded or damaged buildings.

This process can be significantly expedited by using **satellite data** and a **CNN** to identify damaged structures.

02

Data

Where did it come from?



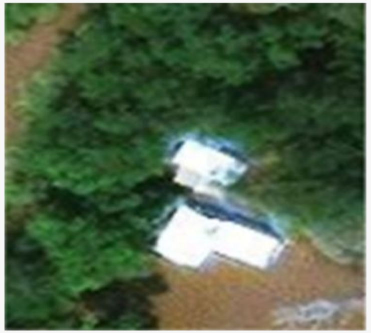
Data Source

IEEE DataPort

This data set contains images labeled with either (a) **damage** or (b) **no damage** representing the state of buildings and infrastructure from the greater Houston area after Hurricane Harvey in 2017.

The data is divided into training, validation, testing, and unbalanced sets.

	Damage (a)	No Damage (b)
Training Set	5,000	5,000
Validation Set	1,000	1,000
Test set	1,000	1,000
Unbalanced set	8,000	1,000


(a)

(b)

03

Method

Preprocessing, the model, training/validation, and analysis.

Preprocessing the Images

First we converted the images to tensors- which is essentially a three-dimensional array housing each color channel.

When reading in the training data set, we also chose to randomly horizontally flip the images to add variation. We chose to do this step because it would increase diversity of the training set to help generalization.

Damage

No Damage

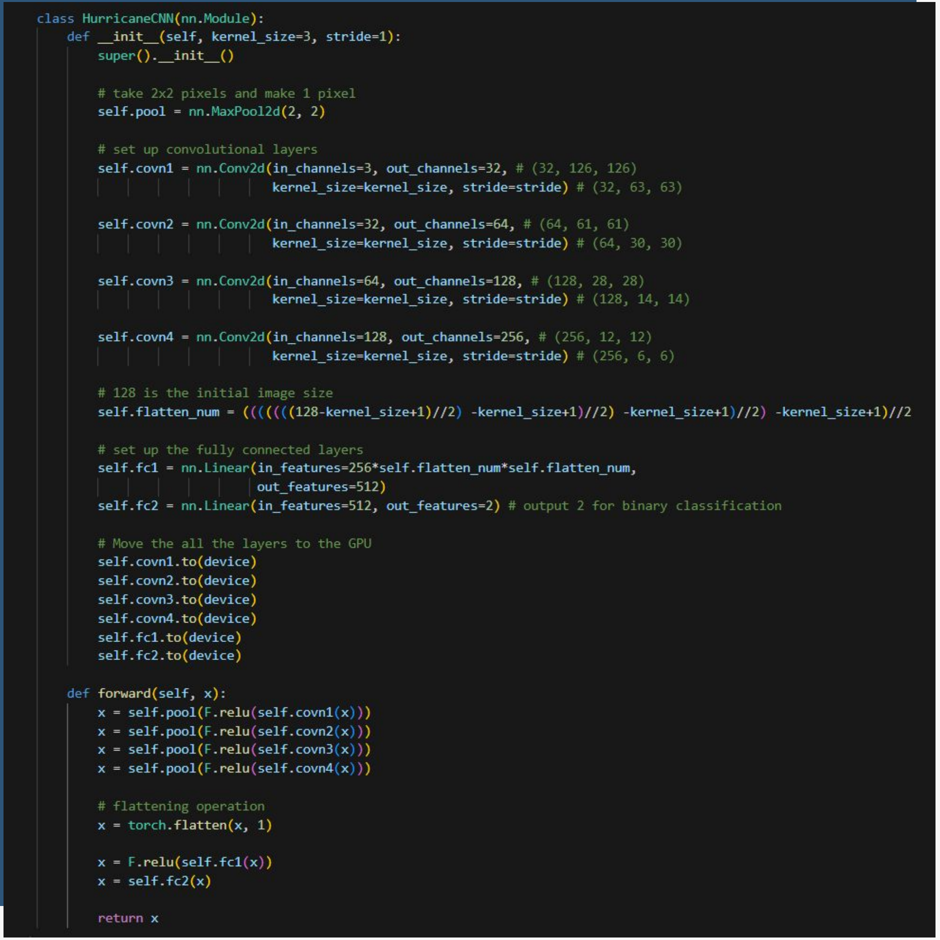
Preprocessing Continued

There are many steps one can take to preprocess image data, the main method we used was normalizing color channels.

Preprocessing was important because normalizing the images helps the model learn better. Making all the colors on the same scale prevents some colors from overpowering others, this assist the model in focusing on the important features.

Calculating these scales specifically from our training images help the model learn even faster and more accurate for these specific type of image.

The Model



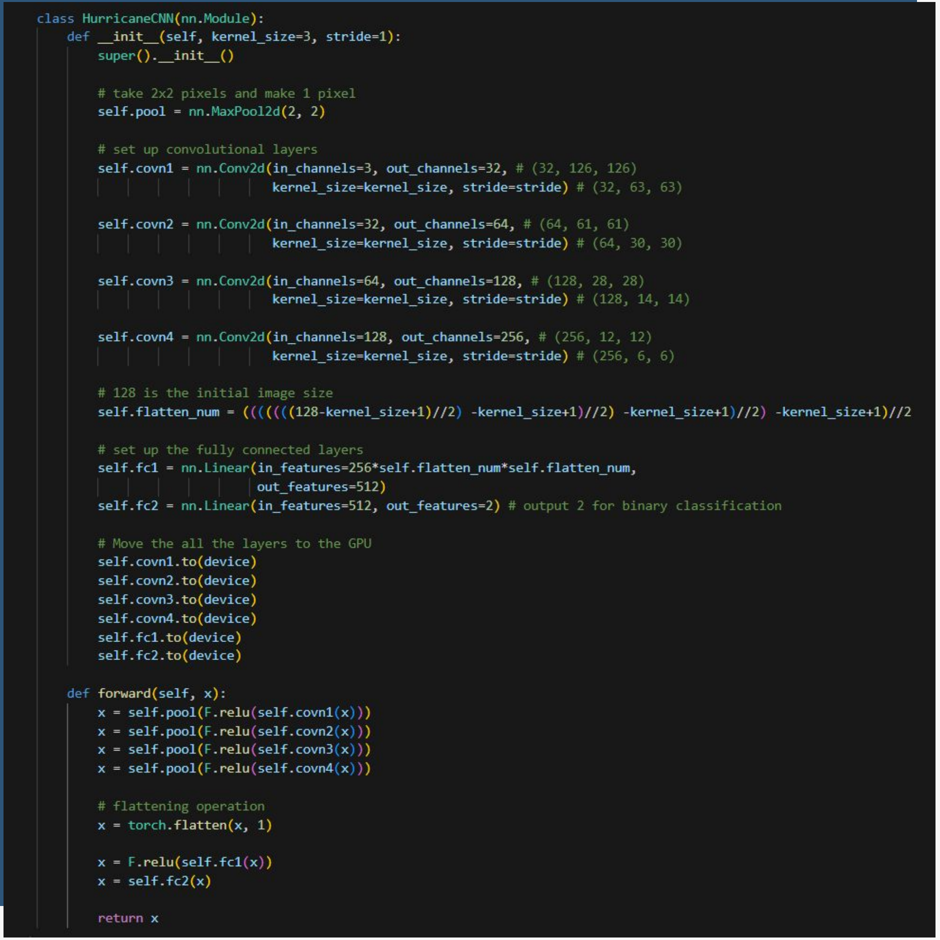
```python
class HurricaneCNN(nn.Module):
    def __init__(self, kernel_size=3, stride=1):
        super().__init__()

        # take 2x2 pixels and make 1 pixel
        self.pool = nn.MaxPool2d(2, 2)

        # set up convolutional layers
        self.covn1 = nn.Conv2d(in_channels=3, out_channels=32, # (32, 126, 126)
                               kernel_size=kernel_size, stride=stride) # (32, 63, 63)

        self.covn2 = nn.Conv2d(in_channels=32, out_channels=64, # (64, 61, 61)
                               kernel_size=kernel_size, stride=stride) # (64, 30, 30)

        self.covn3 = nn.Conv2d(in_channels=64, out_channels=128, # (128, 28, 28)
                               kernel_size=kernel_size, stride=stride) # (128, 14, 14)

        self.covn4 = nn.Conv2d(in_channels=128, out_channels=256, # (256, 12, 12)
                               kernel_size=kernel_size, stride=stride) # (256, 6, 6)

        # 128 is the initial image size
        self.flatten_num = (((((((128-kernel_size+1)//2) -kernel_size+1)//2) -kernel_size+1)//2) -kernel_size+1)//2

        # set up the fully connected layers
        self.fc1 = nn.Linear(in_features=256*self.flatten_num*self.flatten_num,
                             out_features=512)
        self.fc2 = nn.Linear(in_features=512, out_features=2) # output 2 for binary classification

        # Move the all the layers to the GPU
        self.covn1.to(device)
        self.covn2.to(device)
        self.covn3.to(device)
        self.covn4.to(device)
        self.fc1.to(device)
        self.fc2.to(device)

    def forward(self, x):
        x = self.pool(F.relu(self.covn1(x)))
        x = self.pool(F.relu(self.covn2(x)))
        x = self.pool(F.relu(self.covn3(x)))
        x = self.pool(F.relu(self.covn4(x)))

        # flattening operation
        x = torch.flatten(x, 1)

        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

Convolutional Layers

- We have four convolutional layers. The “in” channel is how many feature maps the layer receives and the “out” is how many feature maps the layer generates as output.

- So, we start with a size of **3** *(RGB channels)* and through the convolutional layer it outputs **32** feature maps. Those outputs show one ***specific pattern*** found each.

- The next layer continues to find even more complex layers, and so on, through all four layers.



```
class HurricaneCNN(nn.Module):
    def __init__(self, kernel_size=3, stride=1):
        super().__init__()

        # take 2x2 pixels and make 1 pixel
        self.pool = nn.MaxPool2d(2, 2)

        # set up convolutional layers
        self.covn1 = nn.Conv2d(in_channels=3, out_channels=32, # (32, 126, 126)
                               kernel_size=kernel_size, stride=stride) # (32, 63, 63)

        self.covn2 = nn.Conv2d(in_channels=32, out_channels=64, # (64, 61, 61)
                               kernel_size=kernel_size, stride=stride) # (64, 30, 30)

        self.covn3 = nn.Conv2d(in_channels=64, out_channels=128, # (128, 28, 28)
                               kernel_size=kernel_size, stride=stride) # (128, 14, 14)

        self.covn4 = nn.Conv2d(in_channels=128, out_channels=256, # (256, 12, 12)
                               kernel_size=kernel_size, stride=stride) # (256, 6, 6)

        # 128 is the initial image size
        self.flatten_num = (((((((128-kernel_size+1)//2) -kernel_size+1)//2) -kernel_size+1)//2) -kernel_size+1)//2

        # set up the fully connected layers
        self.fc1 = nn.Linear(in_features=256*self.flatten_num*self.flatten_num,
                             out_features=512)
        self.fc2 = nn.Linear(in_features=512, out_features=2) # output 2 for binary classification

        # Move the all the layers to the GPU
        self.covn1.to(device)
        self.covn2.to(device)
        self.covn3.to(device)
        self.covn4.to(device)
        self.fc1.to(device)
        self.fc2.to(device)

    def forward(self, x):
        x = self.pool(F.relu(self.covn1(x)))
        x = self.pool(F.relu(self.covn2(x)))
        x = self.pool(F.relu(self.covn3(x)))
        x = self.pool(F.relu(self.covn4(x)))

        # flattening operation
        x = torch.flatten(x, 1)

        x = F.relu(self.fc1(x))
        x = self.fc2(x)

        return x
```

Fully-Connected Layer

- The first fully-connected layer takes in flattened feature maps from the last convolutional layer. Which represents all extracted spatial features combined into one vector per image.

- It outputs **512** higher level combinations of features that the model learned that are potentially important in figuring out the final classification.

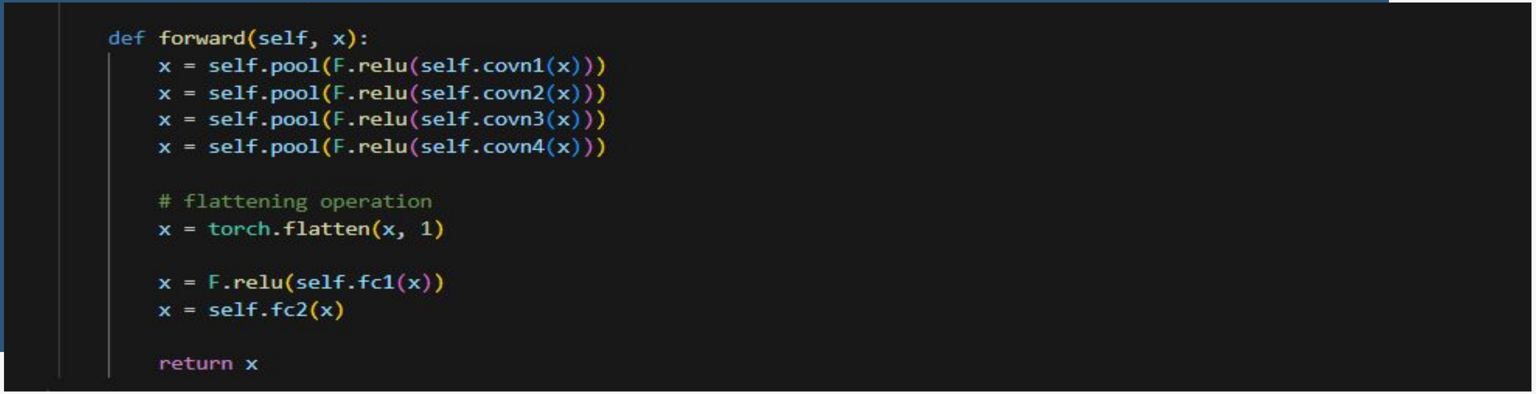
- The second fully-connected layer takes the 512 combinations of features and outputs two scores for the network's confidence of the image's classification.

The Model

Forward Function

- The forward function is the path the data takes through the network. It goes through each convolution then passes through the **ReLU function** *(acts as a nonlinear activation- setting any negative values in the output to 0 while leaving any positive numbers unchanged, helping the network learn complex patterns).*

- Then the image passes through the pooling function to extract increasingly complex spatial features while reducing dimensionality. Essentially, keeping the most noteworthy parts of the images. The features maps are then flattened and passed through the fully-connected layers.

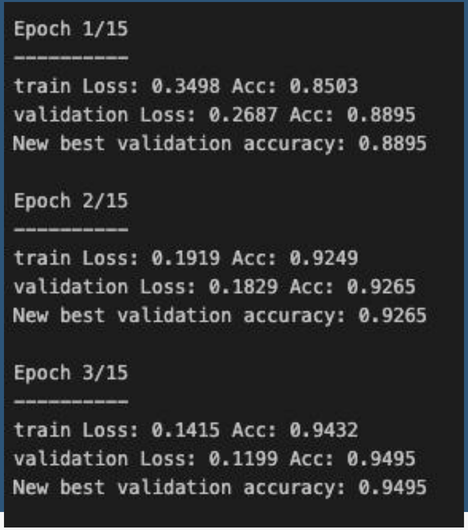


```
def forward(self, x):
    x = self.pool(F.relu(self.covn1(x)))
    x = self.pool(F.relu(self.covn2(x)))
    x = self.pool(F.relu(self.covn3(x)))
    x = self.pool(F.relu(self.covn4(x)))

    # flattening operation
    x = torch.flatten(x, 1)

    x = F.relu(self.fc1(x))
    x = self.fc2(x)

    return x
```

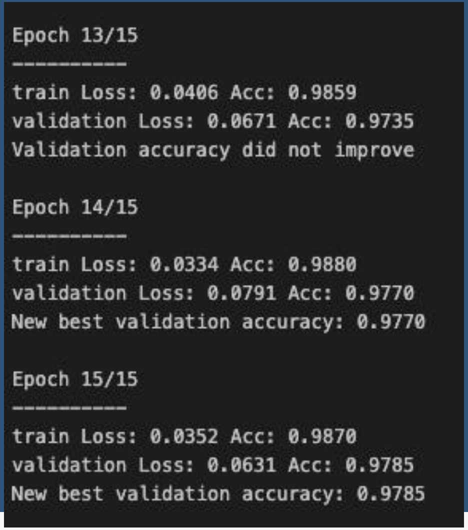
Training and Validation

For our model, we took the approach of having a training set and validation set that we also used during the training process. Because of the format of the data, we did not have to split our data into these sets as it was done for us.

Our model trained over 15 epochs *(one full run through of the data)*. We put the model through a training and validation phase where the loss and accuracy of the model was recorded.



```
Epoch 1/15
----------
train Loss: 0.3498 Acc: 0.8503
validation Loss: 0.2687 Acc: 0.8895
New best validation accuracy: 0.8895

Epoch 2/15
----------
train Loss: 0.1919 Acc: 0.9249
validation Loss: 0.1829 Acc: 0.9265
New best validation accuracy: 0.9265

Epoch 3/15
----------
train Loss: 0.1415 Acc: 0.9432
validation Loss: 0.1199 Acc: 0.9495
New best validation accuracy: 0.9495
```

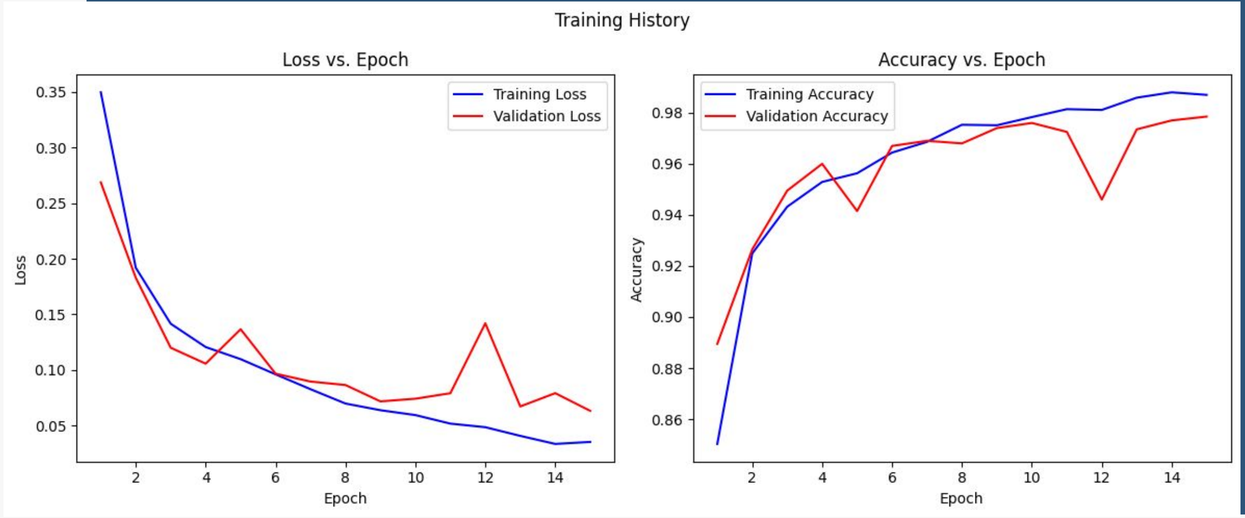


```
Epoch 13/15
----------
train Loss: 0.0406 Acc: 0.9859
validation Loss: 0.0671 Acc: 0.9735
Validation accuracy did not improve

Epoch 14/15
----------
train Loss: 0.0334 Acc: 0.9880
validation Loss: 0.0791 Acc: 0.9770
New best validation accuracy: 0.9770

Epoch 15/15
----------
train Loss: 0.0352 Acc: 0.9870
validation Loss: 0.0631 Acc: 0.9785
New best validation accuracy: 0.9785
```

Training History

We recorded the best validation set accuracy for each epoch, if the validation accuracy did not increase for at least more than 3 epochs in a row we would end the training session early.

As seen in the graph to the side- there was a big spike at epoch 12. There are multiple possibilities for this: this could have simply been an unlucky batch or the model could have started to be overfit.



04

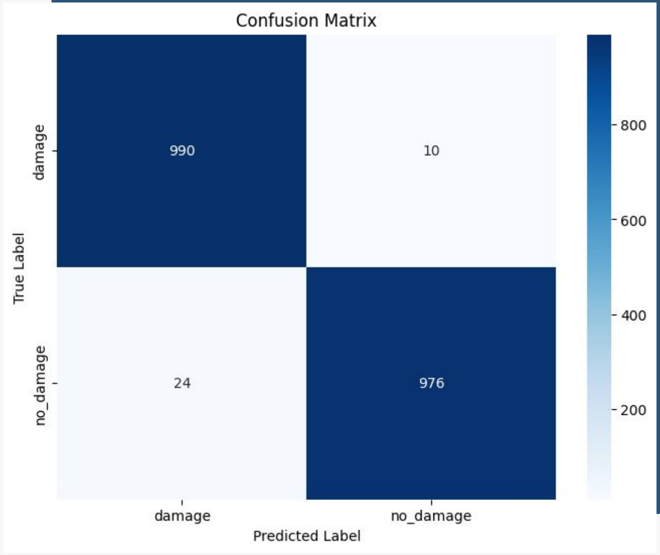
Results

What did we find?

Our Results:

The model achieves **excellent** performance on the balanced test set, with a high precision *(0.98-0.99)* and recall *(0.98-0.99)* for both "damage" and "no damage" classes.

The confusion matrix shows **very few** misclassifications, indicating a good ability to distinguish between damaged and undamaged images in a balanced scenario.

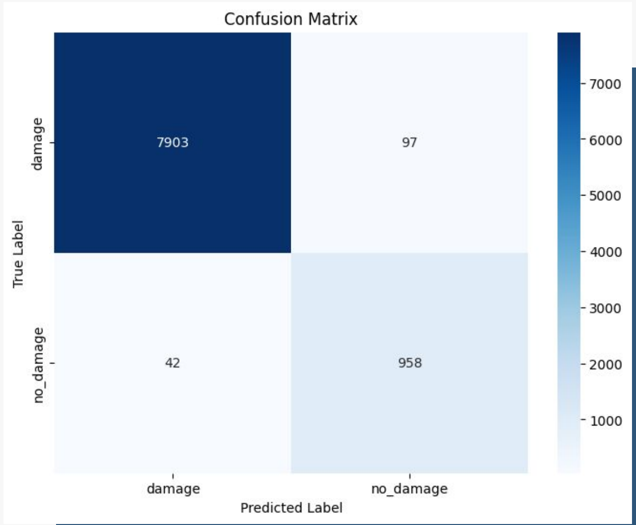


	Precision	Recall	F1-Score
0 (Damage)	0.98	0.99	0.98
1 (No Damage)	0.99	0.98	0.98
Accuracy			0.98
Macro Average	0.98	0.98	0.98
Weighted Average	0.98	0.98	0.98

Our Results: Unbalanced

On the unbalanced test set, the model exhibits **high** performance overall, with excellent precision *(0.99)* and recall *(0.99)* for the majority class (damaged). The "no damage" class shows a slightly lower precision *(0.91)* and recall *(0.96)*, and the confusion matrix indicates 42 images of "no damage" were missed, suggesting this is an area for potential **improvement**.

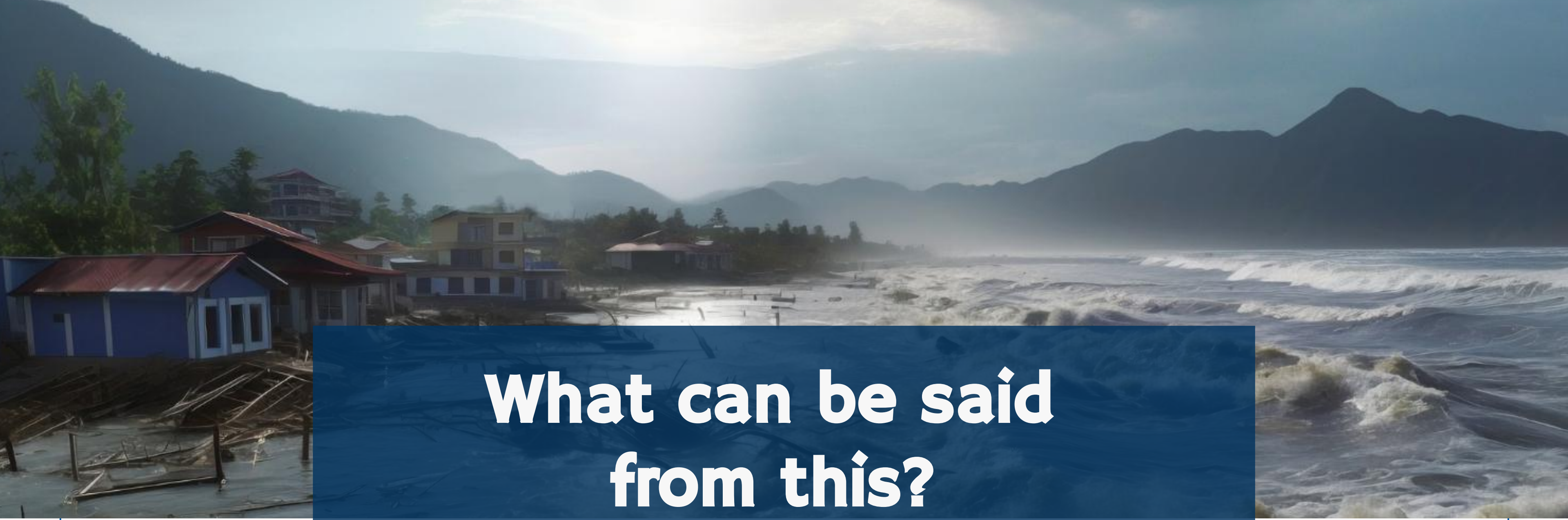
	Precision	Recall	F1-Score
0 (Damage)	0.99	0.99	0.99
1 (No Damage)	0.91	0.96	0.93
Accuracy			0.98
Macro Average	0.95	0.97	0.96
Weighted Average	0.99	0.98	0.98



05

Conclusion

What does this mean?



What can be said from this?

Based on our findings, and similar models made, we believe that utilizing convolutional neural networks to categorize satellite images after hurricanes should be a growing interest point. With more research and fine tuning, it could be a **very accurate** method to assess and assist the areas affected following a hurricane.

Future models could be trained on other natural disasters *(such as wildfires or tornados)* to further assist in relief assistance.

References

What are convolutional neural networks?. IBM. (2024). https://www.ibm.com/think/topics/convolutional-neural-networks

Cao, Q. D. (2018). *Detecting damaged buildings on Post-Hurricane satellite imagery based. IEEE DataPort.*https://ieee-dataport.org/open-access/detecting-damaged-buildings-post-hurricane-satellite-imagery-based-customized