Flooding Damage Detection from Post-Hurricane Satellite Imagery Based on Convolutional Neural Networks

Bixuan Huang
George Washington University
bixuan@gwu.edu

Weining Hu
George Washington University
whu369@gwu.edu

Abstract

In this project, we applied convolutional neural networks (CNN) to detect flooding damages from the 2017 Hurricane Harvey satellite images. To train the model, we used a dataset posted on Kaggle consisting of images labeled with two types of condition, which are "damage" or "no damage." We experimented with three different CNN architectures, which are VGG-16, Resnet50, and a custom convolutional architecture. Our best model, the custom model, was able to achieve an accuracy of over 90% on the test set. In this paper we describe the dataset we used, present related work, explain how we iterated on our model to come up with the final version, and finally evaluate our results.

1 Introduction

When a hurricane makes landfall, emergency managers need to assess the situation and flooding damage so that they can make efficient response to the disaster and allocate resources for future rescue and recover work. Traditional practice to assess the disaster situation and damage largely relies on emergency response crews and volunteers to drive around the affected area and make assessments accordingly. However, this process is labor intensive and time-consuming. Thanks to the blooming development in the filed of computer vision studies, scientists find another way to improve the efficiency of building disaster assessment. More specifically, they applied image classification algorithms to distinguish damaged buildings and flooding area from the ones still intact. In this project, we explored varied state-of-the-art convolutional neural networks and use them to train and classify the 2017 post-Harvey Hurricane satellite images.

1.1 Dataset

We obtained the data[1] from Kaggle, which gives 128 X 128 X 3-pixel RGB satellite images of the Greater Houston area before and after Hurricane Harvey in 2017. The dataset also labels all images into two classes, either no damage or damage. The training set[2] consists of 10,000 examples, while the validation set[3] is composed of 2,000 examples. The test set[4] contains 9,000 images. The class distribution of the dataset is displayed in Table 1:

[1] https://www.kaggle.com/kmader/satellite-images-of-hurricane-damage
[2] The training set folder is named as "*train_another*."
[3] The validation set folder is named as "*validation_another*."
[4] The test set folder is named as "*test_another.*"

Labels / Folders	Damage	No Damage
Training set	5,000	5,000
Validation set	1,000	1,000
Test set (Unbalanced)	8,000	1,000

Table 1: Distribution of the dataset

As it is shown in Table 1, each of the training set and validation set has an even distribution across the two classes. Therefore, there is no need to apply image augmentation to increase the images from the underrepresented class. The dataset contains a balanced test set (titled as "*test*" in the zip file) and an unbalanced test set (titled as "*test_another*"). To preserve the original distribution of the two classes, we chose to test our models on the unbalanced dataset rather than the balanced test set. A few example images from the dataset are show in Figure 1.

Damage No Damage

Figure 1: Example images from the dataset

1.2 Problem Statement

The goal of this project is to predict whether a given image from the hold-out test set contain flooding damage or no damage. Additionally, we compare the performances between a custom architecture with fewer convolutional layers and the pre-trained models with more convolutional layers. Our evaluation matrix is the accuracy for each class, supplemented with a confusion matrix that highlights which class is better recognized than the other.

1.3 Related Work

The dataset was released by two students, Quoc Dung Cao and Youngjun Choe, from the University of Washington. They were the first ones who applied image classification algorithms to classify the 2017 Hurricane Harvey satellite images. In December 2018, the *Institute of Electrical and Electronics Engineering Journal* accepted their research paper and published their work, which highlighted the findings that convolutional neural networks can automatically annotate flooded/damaged area on post-hurricane satellite images with approximately 97% accuracy.[5] Cao and Choe trained their networks using the *Keras* framework and on a shorter version of the VGG-16 network.

[5] Cao, Quoc Dung, and Youngjun Choe. "Building Damage Annotation on Post-Hurricane Satellite Imagery Based on Convolutional Neural Networks." *arXiv preprint arXiv:1807.01688* (2018). URL: https://arxiv.org/abs/1807.01688

To make new contributions to the existing work that were completed by Cao and Choe, we used *PyTorch* as our framework to train neural networks on. In addition to VGG-16, we trained our model using another pre-trained model, Resent50. Lastly, we also built a custom model with less convolutional layers to compare with the pre-trained models.

2 Pretrained Models

To begin with this project, we chose VGG-16 and Resnet50 as our pre-trained models. Both models are representative state-of-the-art architectures for convolutional neural networks. The most unique thing about VGG-16 is that instead of having a large number of hyper-parameters, it only performs 3 X 3 convolutional layers and 2 X 2 max pooling layers from the beginning to the end (Figure 2). However, the downside of the VGG-16 architecture is that it is very expansive to evaluate and uses significantly more memory and parameters (approximately 138 million parameters) than the other architectures (Figure 3).

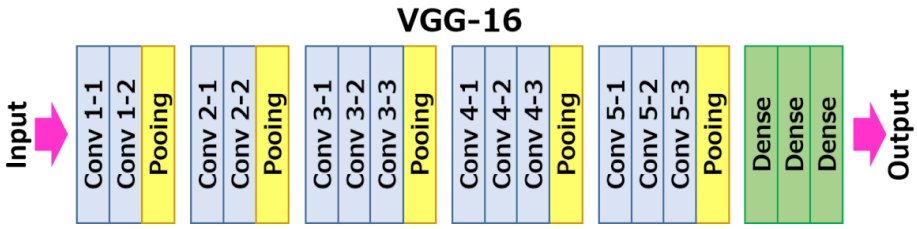

Figure 2. VGG-16 architecture

Model	Size	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
Xception	88 MB	0.790	0.945	22,910,480	126
VGG16	528 MB	0.713	0.901	138,357,544	23
VGG19	549 MB	0.713	0.900	143,667,240	26
ResNet50	99 MB	0.749	0.921	25,636,712	168
InceptionV3	92 MB	0.779	0.937	23,851,784	159
InceptionResNetV2	215 MB	0.803	0.953	55,873,736	572
MobileNet	16 MB	0.704	0.895	4,253,864	88
MobileNetV2	14 MB	0.713	0.901	3,538,984	88
DenseNet121	33 MB	0.750	0.923	8,062,504	121
DenseNet169	57 MB	0.762	0.932	14,307,880	169
DenseNet201	80 MB	0.773	0.936	20,242,984	201
NASNetMobile	23 MB	0.744	0.919	5,326,716	-
NASNetLarge	343 MB	0.825	0.960	88,949,818	-

Figure 3. Pretrained models comparison[6]

[6] https://datascience.stackexchange.com/questions/39177/which-is-the-fastest-image-pretrained-model

Resnet50 is similar to VGG-16 but with the additional identify mapping capability (Figure 4). The uniqueness of Resnet50 is that its "skip connection" capability allows a model to stack additional layers and build a deeper network, meanwhile it can offset the vanishing gradient by allowing the network to skip through layers that it feels they are less relevant in training. Compared with VGG-16, Resnet50 trains much faster while allowing for much deeper networks.

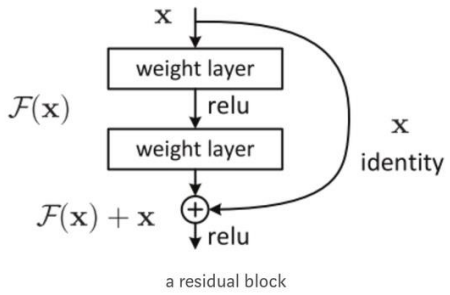

Figure 4. Resnet residual block diagram with identity mapping

2.1 Models Set Up

Our approach was to experiment with two different pretrained architectures. Once we had the models run successfully, we began running hyperparameter tuning experiments. The following are the major changes that we made to customize the pretrained architecture for our own use.

Image Augmentation: Since all pretrained architectures require the height and width of an input image to have at least 224 pixels, we resize the image size from 128 X 128 to 224 X 224. Additionally, all pretrained models expect input images to be normalized in the same way.[7] We loaded the images to a range of [0, 1] by converting the image from a *numpy* type to a *tensor* and then normalized the images using mean = [0.485, 0.456, 0.406] and std = [0.229, 0.224, 0.225] (Figure 5).

```
train_tfroms = transforms.Compose([transforms.Resize((224, 224)),
                                   transforms.RandomHorizontalFlip(),
                                   transforms.ColorJitter(),
                                   transforms.ToTensor(),
                                   transforms.Normalize(mean=[0.485, 0.456, 0.406],
                                                        std=[0.229, 0.224, 0.225])])
```

Figure 5. Code for image augmentation

Fully Connected Layer: All of the pretrained models were trained on the *ImageNet* library and therefore the last fully connected layer has 1000 outputs. In order to tailor the pretrained models for our own use, we first froze all of the existing lower convolutional layers in the VGG-16 network, and then we replace the classifier (fully connected) part of the network with a custom classifier (Figure 6). In that custom classifier, we set the number of outputs in the last fully connected layer to 2, which equals to the number of classes in our data set.

[7] https://pytorch.org/docs/stable/torchvision/models.html

```
# Freeze early layers
for param in model.parameters():
    param.requires_grad = False
n_inputs = model.classifier[6].in_features
n_classes = 2

# Add on classifier
model.classifier[6] = nn.Sequential(
    nn.Linear(n_inputs, 256), nn.ReLU(), nn.Dropout(0.2),
    nn.Linear(256, n_classes))
```

```
for param in model.parameters():
    param.requires_grad = False

n_inputs = model.fc.in_features
n_classes = 2
model.fc = nn.Sequential(
    nn.Linear(n_inputs, 256), nn.ReLU(), nn.Dropout(0.2),
    nn.Linear(256, n_classes))
```

Figure 6. Code for adding a custom classifier in the pre-trained models

Criterion: Since we are tackling a binary classification problem, we set the criterion as *CrossEntropyLoss* (Figure 7). This criterion is used to predict the probability for each one of the two classes for each example. Since *CrossEntropyLoss* already contains a *Softmax* layer in the last layer of the network, there is no need to add an additional *Softmax* layer in our network.

```
criterion_vgg = nn.CrossEntropyLoss()
criterion_resnet50 = nn.CrossEntropyLoss()
```

Figure 7. Code for setting the criterion for both pretrained models.

2.2 Hyperparameter Tuning

Once we had established a successful base architecture, we sought to improve our performance through hyperparameter searches. The following present the main hyperparameters adjusted.

Batch Size: We started modeling by loading 18 images to the *Dataloader.* However, when we increase the batch size in the *Dataloader* to more than 100, the time used to train the VGG-16 model for 30 epochs can last for as long as 1 hour. Therefore, to spend our time wisely, we decided to limit the batch size to less than 80, although this in return limits the number of choices that we can make when choosing the mini-batch size.

Learning Rate: We started to train the VGG-16 model by choosing the default learning rate (0.00002) recommended in Cao and Choe's paper. We then experimented with *PyTorch learning rate finder* in an attempt to find a better learning rate for our pretrained models (Figure 8).[8] This library is made based on Leslie N. Smith's cyclical learning rates theory.[9] She argues that typically, a good static learning rate can be found half-way on the descending loss curve. By implementing the *torch-lr-finder*, we were able to find the best learning rate for Resnet50 (Figure 9), but the searching results for the VGG-16 architecture did not go well (Figure 10).

[8] https://github.com/davidtvs/pytorch-lr-finder
[9] https://arxiv.org/abs/1506.01186

```
optimizer_resnet50 = torch.optim.Adam(model_resnet50.parameters(), lr=0.00001, weight_decay=1e-6)
lr_finder_resnet50 = LRFinder(model_resnet50, optimizer_resnet50, criterion_resnet50, device=device)
lr_finder_resnet50.range_test(train_loader, end_lr=0.08, num_iter=200, step_mode="linear")
result_resnet50 = lr_finder_resnet50.history
plot_loss_resnet50 = result_resnet50["loss"]
plot_lr_resnet50 = result_resnet50["lr"]
plt.xticks(rotation=45)
plt.title("Resnet50: Learning Rate")
plt.plot(plot_lr_resnet50, plot_loss_resnet50, color = "skyblue")
plt.savefig('resnet50_best_lr.png')
```

Figure 8. Code for implementing *torch-lr-finder*

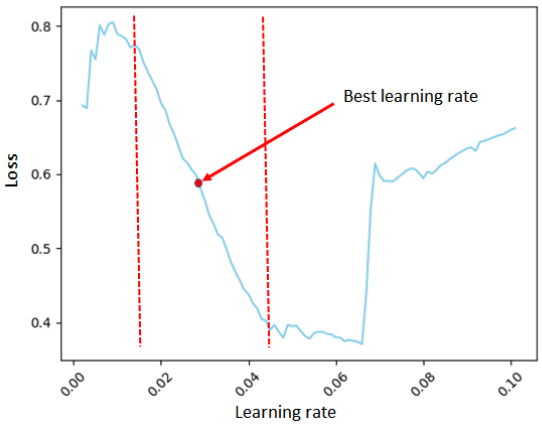

Figure 9. Searching the best learning rate for Resnet50

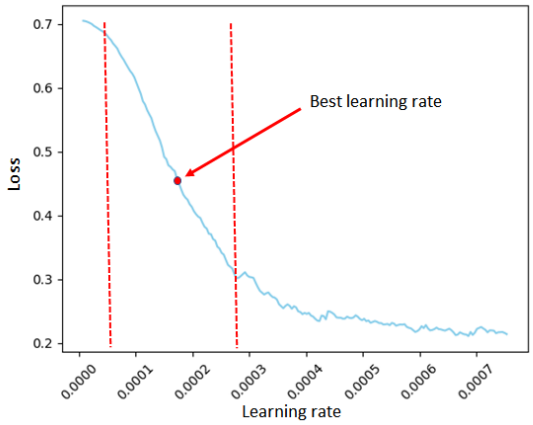

Figure 10. Searching the best learning rate for VGG-16

In fact, the pre-trained models did not perform well using the best learning rate yielded by *torch-lr-finder*. In this project, the best learning rates that we determined for all of our models were not from automated program, but instead derived through multiple experiments and comparisons where we manually picked the learning rate.

Mini-Batch Size: We experimented with a batch size as large as 40. However, the performances are less desirable than using a smaller mini-batch size. Our results show that a smaller mini-batch size between 10 to 20 achieves better performances than using a larger size.

3 Custom Model

3.1 Model Architecture

As for the custom model, we built a network that contains 8 layers with weights. As illustrated in Figure 11, the model follows the block design pattern, where one convolutional layer (filter of 3×3 or 6×6)), one *ReLU* activation layer, one *Batch Normalization* layer, one pooling layer (factor 2, *Max Pooling* or *Average Pooling*) are stacked as a single learning block. The whole model is formed by three learning blocks, followed by two fully connected layers. We chose *Stochastic gradient descent (SGD)* as the model optimizer and *CrossEntropyLoss* as the function loss. The figure below displays only the major weighted layers of the network. Activation function layers and dropout layers are eliminated from the diagram for aesthetic purposes.

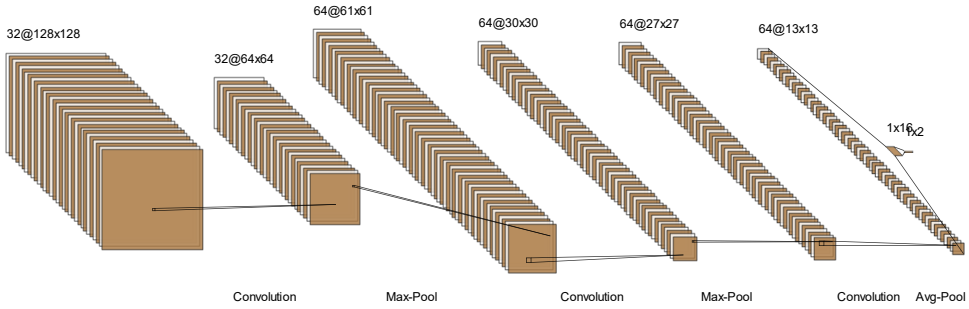

Figure 11. LeNet Style Diagram of the Custom Network[10]

3.2 Hyperparameters Tuning

Once we had established a successful base architecture, we sought to improve our performance through hyperparameter searches. The following present the main hyperparameters adjusted.

Batch Size: Based on the batch size above for the pre-train model, we started modeling by loading 50 images to the *Dataloader.* Less batch size limits our choice in Mini-Batch Size, so we increased the batch size. With the experience of increasing the batch size in the *Dataloader* to train pre-train model, to spend our time wisely, we decided to limit the batch size to 100.

Learning Rate: Similarly, the performance of the *Torch-lr-finder* did not perform as good as we expected. We experimented to find the best learning rate by tracking and comparing the learning rate manually and finally, we determined that 0.01 was the most appropriate learning rate for the custom model.

Mini-Batch Size: We experimented with a batch size as large as 100. However, the performances are less desirable than using a smaller mini-batch size. Our results show that a smaller mini-batch size below 25 achieves better performances than using a larger batch size. In the final model, we used the size of 4 for mini-batch.

4 Results

To assess the performance of our models, we use a combination of accuracy and confusion matrix.

4.1 Accuracy

We were able to achieve good results with all three models. As shown in Table 2, both pre-trained models achieved an accuracy over 88%, while the custom model achieved the highest accuracy. This result reveals that even a small convolutional neural network can perform well on this particular dataset. Deep networks

[10] The diagram was created using this online tool: http://alexlenail.me/NN-SVG/LeNet.html

such as VGG-16 and Resnet50 are not necessarily always the best options to solve image classification problems.

Models	Accuracy Rate (%)
VGG-16	88.92
Resnet50	88.43
Custom Model	91.83

Table 2. Accuracy summary across all models

4.2 Confusion Matrix

We examine the confusion matrix for the test set to understand better where misclassifications occur. We see that on the whole, our classification is fairly good, as demonstrated by the strong presence along the diagonal (Figure 12). All three models are most successful at classifying "damage" (Table 3). Particularly, the custom model achieved approximately 93% accuracy in identifying damage images. On the other hand, all three models struggled to classify "no damage" images. Specifically, VGG-16 is among the best to detect no damage images.

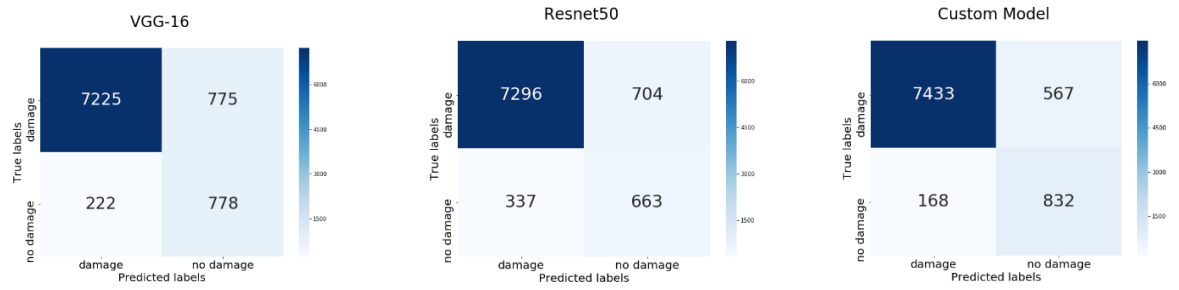

Figure 12. Confusion Matrix of All Three Models (VGG-16, Resnet50 and Custom Model)

Models	Damage Accuracy	No Damage Accuracy	Damage Error	No Damage Error
VGG-16	90.31%	77.8%	9.69%	22.2%
Resnet50	91.2%	66.3%	8.8%	33.7%
Custom Model	92.91%	83.2%	7.08%	16.8%

Table 3. Classification accuracy and error across all classes

4.3 Loss and Accuracy Over Time

Looking at the graphs in Figure 13, we see that neither the Resnet50 and or the VGG-16 model has healthy loss curves. Although both the train loss and validation loss decreased over epochs, the gap between those two losses is quite big. This is a sigh that the models are not learning enough. Due to the time constraint, we were not able to train our data for more epochs. It is possible that when we increase the number of epochs, the training loss and validation loss will ultimately merge together.

Conversely, the custom model yielded a slightly higher validation loss than the training loss. It could be a sign that the custom model is overfitting. For future work, we can try to improve the custom model by

adding more dropout/batch normalization layers, applying more data augmentation, or decreasing the number of parameters of the model.[11]

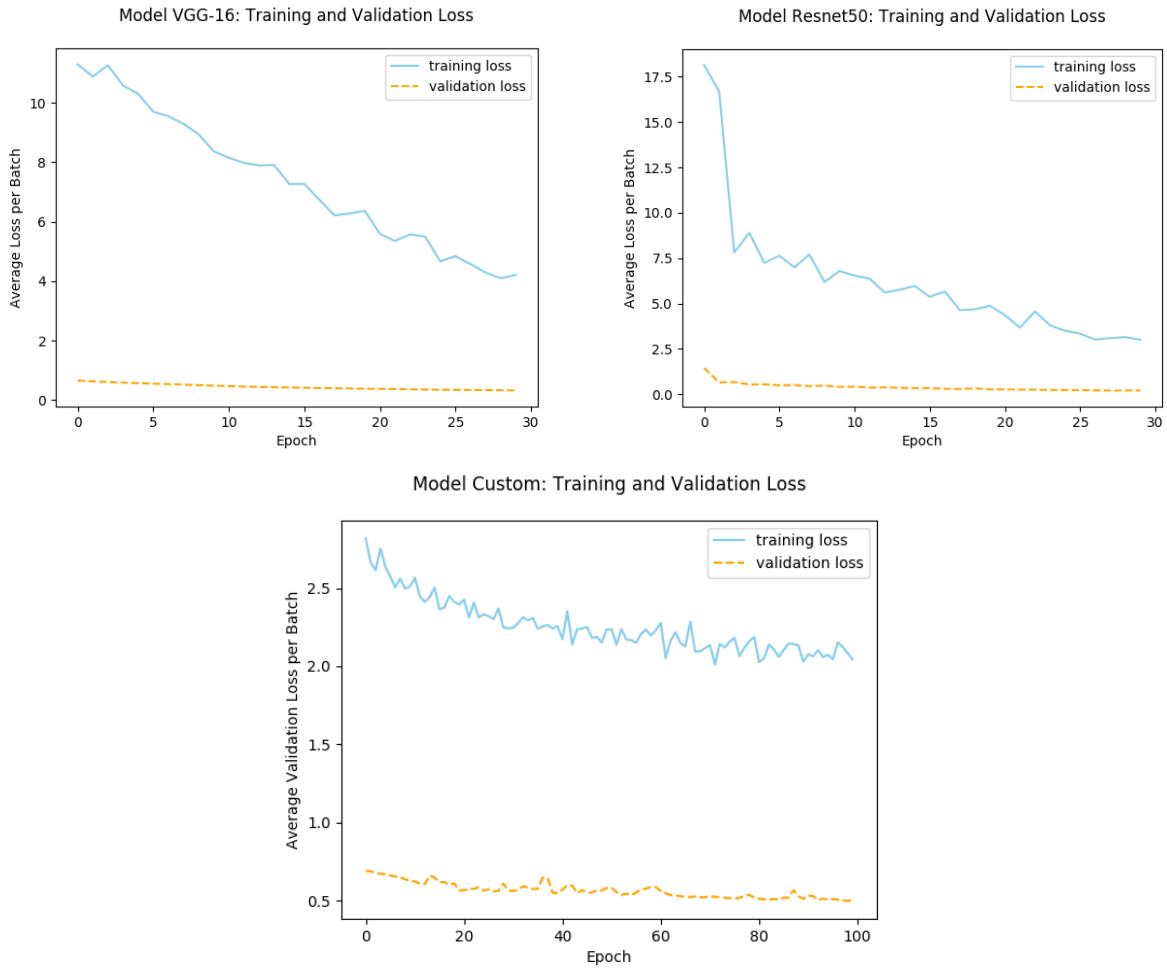

Figure 13. Train loss and validation loss comparison across different models (VGG-16, Resnet50, Custom Model)

5 Conclusion

In this project, we designed a custom convolutional neural network from scratch and two pre-trained models to recognize flooding damage from images that were taken before and after Hurricane Harvey in 2017. The results demonstrated that all of our models were able to achieve acceptable results in comparison to the original work completed by Cao and Choe (their best model achieved a 97% accuracy). Among our three models, the custom model with less convolutional layers performed best in detecting flooding damage, while the VGG-16 model performed the best in classifying no flooding damage images.

To further improve model performance, we would like to implement more epochs to train our pre-trained models, offsetting the loss issues mentioned in section 4.3. Additionally, we wish to experiment with more measures to solve the overfitting issues occurred with the existing custom model. Lastly, we wish to leverage model ensemble techniques to incorporate the strength from all three models.

[11] https://groups.google.com/forum/#!topic/caffe-users/J5205HG7sog