Data preparation: what did you do?

I started by downloading the data in its entirety and using Python's PIL (pillow) library to load each image. I converted these pillow objects into numpy arrays, and then stored a nested array of numpy arrays, where each internal array is a representation of a single image. To make this easier to test (and avoid OOM issues), I decided to collect up all the file_paths initially, and then take a random sample of the file paths, and only load these images. I then performed a train/val/test split of the data (65%, 24.5%, 10.5% respectively), and passed this split onto Part 2 for training.

I also had an alternate implementation (which I submitted as my best_model), and for this implementation, I used the pretrained Swin-T transformer architecture. To load the data for this particular model, I used a custom ImageDataset class, with standard torch image dataset methods (len() and getitem()), along with DataLoader objects to lazily load the image data. I thought that the lazy-loading property of the pytorch implementation I had was super important (especially considering that the transformer I was fine-tuning was quite a bit larger than the test models), and if I was submitting my original model as my "best_model," I would've tried to incorporate lazy-loading into that part of the code as well.

For the torch implementation, I also decided to add a few preprocessing/image modification steps. The input for the SwinT architecture was required to be 224x224x3, so I had to resize the images as they were loaded in. Furthermore, I also reasoned a lot about the nature of the hurricane image dataset. Theoretically, the drone capturing these images (and by extension the images themselves), should be independent of their orientation and if they were flipped over the x/y-axis. Thus, to augment the data, and make the transformer resilient to relying on particular regions/features of the dataset that may have had some biases through collection, I decided to apply vertical and horizontal flips randomly to the data, along with random rotation amounts between 0 - 270 degrees. Lastly, I had a 65% train, and 17% val/test split. If I could go back, I would likely increase the train% to 75% but I originally lowered it for testing purposes and later forgot to reset it. Because I trained my entire model on that particular split, I decided it would be too computationally expensive to retrain it again.

Model design: which architectures did you explore and what decisions did you make for each?

The basic architectures that I explored were the dense ANN, the lenet-5 (standard) and the modified lenet-5 (architecture based on the paper that was provided). I was mostly interested in building a custom architecture, but here are the decisions I made for all of my basic architectures.

DenseANN:
For the dense-ann, I wanted a simple baseline for the overall accuracy I could achieve. My architecture first flattens the (128, 128, 3) input image into a 1D vector, which (unfortunately!) discards spatial relationships but is necessary for a dense network. This vector is fed into two hidden layers with 512 and 256 neurons, respectively, using the relu activation function to promote faster training and avoid vanishing gradients. To avoid the risk of overfitting with the dense network, I added a Dropout(0.5) layer in between the hidden layers to randomly deactivate 50% of the neurons every train step. This would help the network stay more robust and better recognize features in the images. The final output layer is a single Dense neuron with a sigmoid activation, producing a probability value between 0 and 1 for binary classification. The model was compiled using the Adam optimizer and binary_crossentropy loss, which is a pretty standard setup.

LeNet-5
My architecture's core is slightly modified (to fit this particular problem's constraints) but is generally in the form of the LeNet-5 architecture. Its core consists of two Conv2D layers with (5, 5) filters, the first having 6 filters and the second having 16. These layers learn spatial features, offering a significant advantage over the DenseANN. I used the more modern relu activation instead of the classic tanh activation, as it was supposed to perform better generally. After each convolutional layer, a MaxPooling2D ((2,2)) layer downsamples the feature maps, making features more robust to translation. After flattening the output of the conv/pool stack, the data passes through the two classic LeNet-5 dense layers (120 and 84 neurons) before reaching the final single-neuron sigmoid output layer for binary classification. The compilation settings were the same as the DenseANN, using the Adam optimizer and bce loss.

Modified LeNet-5:
The last architecture that I implemented was based on the provided paper and represents a much deeper, more modern CNN design. Instead of two conv layers, this model stacks four conv2d layers, all using smaller (3, 3) filters, which is a more modern way to extract feature representations. The filter depth steadily increases (32 -> 64 -> 128 -> 128) as the spatial dimensions are reduced by MaxPooling2D((2, 2)) layers. The paper also added a Dropout(0.5) layer after flattening which helped to prevent overfitting. This layer is followed by a large Dense(512, relu) layer and then finally a Dense(1, sigmoid) output layer. This model also had an Adam optimizer and bce loss.

SwinT:
The custom model that I implemented was a fine-tuned Swin Transformer (specifically the "Swin-T" variant, or Swin-Tiny), a SOTA vision transformer architecture that was pretrained on ImageNet. I particularly modified the architecture slightly to fit a binary classification usecase. At a high level, transformers differ from traditional CNNs by relying on self-attention mechanisms rather than convolutions to capture global data relationships. Swin divides images into non-overlapping "patches" (4x4 pixel regions), which are then treated as "tokens" similar to words in NLP. These patches are linearly embedded and fed through multiple transformer stages, where multi-head self-attention computes interactions between all patches, allowing the model to learn complex dependencies across entire image regions rather than just neighbouring regions (ex, how damage in one part of an image relates to another). SwinT is special because it improves efficiency over standard ViTs by using a hierarchical structure (like CNNs), progressively reducing spatial resolution across stages while increasing channel depth, and employing an algorithm called shifted window-based attention. Shifted window is very powerful because it makes the processing time scale linearly instead of quadratically with window size which means that we still retain a large amount of the power within the transformer but we don't have insanely long inference times.

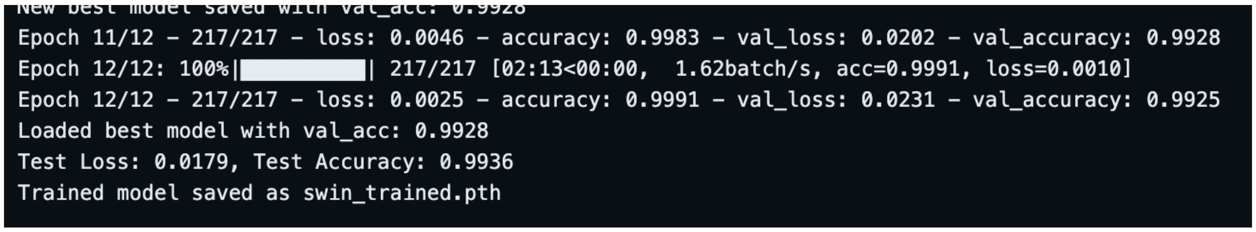
In my code, I loaded the pretrained Swin-T model from Hugging Face's transformers library (microsoft/swin-tiny-patch4-window7-224) and modified the configuration for the final layer. The transformer was originally trained on imagenet, so it had 1000 output classes; I changed the final layer to only have 2 classes instead (damage/no_damage). To train the entire model efficiently, I used a two-step training process. Initially I froze all of the weights within the interior of the model, and only enabled weight-adjustment on the final layer. The goal here was to help the output layer learn to understand and interpret the feature extraction capabilities of the core pretrained architecture. I got to around 95% accuracy here over 12 epochs before I moved onto fine tuning the entire model. I want to note that the lr was pretty high (1e-3) because I just wanted rough convergence. In stage two I fine-tune with a much lower lr anyway.

I then unfroze the entire model's weights and trained for another 12 epochs. I implemented a lr scheduler to decay the learning rate and help prevent overfitting. It turns out that the model converged pretty quick despite the decreased lr, and I had my best results at around 8 epochs of fine-tuning.

To adapt it efficiently, I used a two-stage training process: first, I froze the backbone (the Swin encoder layers) and trained only the classification head with a higher learning rate (1e-3) for 12 epochs, allowing the head to learn from the pretrained features without disrupting them; then, I unfroze the entire model and fine-tuned all parameters with a lower learning rate (1e-5) for another 12 epochs, using a step learning rate scheduler to decay the LR every 5-10 epochs and prevent overfitting I saved the best model based on validation accuracy and evaluated it on a held-out test set, achieving near-perfect performance (99.65% accuracy).

Model evaluation: what model performed the best? How confident are you in your model?

The SwinT model performed the best by far. On the holdout test set that I had, I had an accuracy of 0.9936. I did not test precision/recall but I am extremely confident in my model's performance.



```
New best model saved with val_acc: 0.9928
Epoch 11/12 - 217/217 - loss: 0.0046 - accuracy: 0.9983 - val_loss: 0.0202 - val_accuracy: 0.9928
Epoch 12/12: 100%|██████████| 217/217 [02:13<00:00,  1.62batch/s, acc=0.9991, loss=0.0010]
Epoch 12/12 - 217/217 - loss: 0.0025 - accuracy: 0.9991 - val_loss: 0.0231 - val_accuracy: 0.9925
Loaded best model with val_acc: 0.9928
Test Loss: 0.0179, Test Accuracy: 0.9936
Trained model saved as swin_trained.pth
```

Model deployment and inference: a brief description of how to deploy/serve your model and how to use the model for inference (this material should also be in the README with examples)

Deploying my model is extremely simple. Simply open two separate terminals. In the first terminal, run

```
docker run --rm -p 5000:5000 vnach291/hurr-pred:latest
```

In the second terminal, make sure you have the following directory structure

[parent]/data/damage
[parent]/data/no_damage
grader.py (file contents are provided in the repo for completeness)

Navigate to the parent directory in the second terminal and run the following command.

```
docker run -it --rm -v $(pwd)/data:/data -v
$(pwd)/grader.py:/grader.py -v $(pwd)/project3-results:/results
--entrypoint=python jstubbs/coe379l /grader.py
```

Populate the damage and no_damage folders with damaged and undamaged images respectively, and you can test the model's performance. Enjoy!