Fine Grained Visual Classification with NABirds

Colin Arwood

Problem Overview

Fine Grained Visual Classification (FGVC) is a hard problem in the realm of Computer Vision, and will be an important benchmark for Artificial Intelligence, and Computer Vision moving forward. It is one thing to be able to distinguish between an image of a car and an image of a bird as they are relatively dissimilar. But can a computer learn to recognize different types of cars? And different types of birds?

The dataset that I chose was the NABirds dataset, a large (48,000 images) dataset with 555 categories created by the Cornell Lab of Ornithology. It is an updated version of the CUB200 dataset, with both creators of CUB contributing to NABirds. I chose this dataset because I have a new interest in birds as of this year, and have taken up birding as a hobby.

FGVC is hard because birds of different species may look very similar. Examples of the difficulty in differentiating birds can be seen below by comparing two different types of Warblers: Nashville Warbler (left) and MacGillivray's Warbler (Right). Images taken from NABirds dataset.

Additionally, their can be large discrepancies in birds of the same class, different morphs of the same bird, variety of positions and setting, changing plumage between seasons, as well as sexual dimorphism in species make species identification more difficult. Fortunately, NABirds differentiates between morphs, as well as maturity and sex; it labels the birds accordingly (as well as provides a class hierarchy on how they are related, though this is typically unused in FGVC). However, variation in setting and positions remain, the example below is of the Barn Owl, images taken from the dataset showing these discrepancies in one class:

Data Preprocessing

The first issues that I ran into were with regards to the quantity of the data as well as shape and size. As we can see above in the images of Barn Owls, the images are not all the same shape. Some are taken in portrait mode, some are taken in landscape, and others are fairly square. There is not exactly a perfect solution to this problem, my solution was reshaping the images to 300x300, which results in some horizontal or vertical compression depending on the input image. The images are then cropped to 224x224 (randomly when training in hopes of preventing overfitting, more on overfitting later). Additionally the data is normalized such that the image's intensity and lighting effects are mitigated as best they can.

The dataset is fairly large (~10 GB), which was slightly too large for my computer and certainly too large for my GPU which had 6GB of VRAM (GTX1660 Super was used for training). This also presented a unique challenge to me, luckily, a variety of online resources exist, and I figured out how to use the DataLoaders that Torch provides, and can read the images in the `__getitem__()` call, and we use TorchVision's `default_loader` to load the images.

Solution 1: CNN

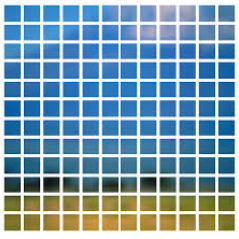
Expectedly, the first way that I chose to tackle this problem was using a Convolutional Neural Network (CNN). CNN's have been prevalent and used in State of the Art vision classification algorithms since AlexNet in 2012. I initially tried to use a fairly simple model with a few layers, but quickly found that the model was simply not complex enough to score well on NABirds dataset. Fine Grained Image Classification is a hard problem, specifically for shallow and "simplistic" models.

After researching State of the Art FGVC models, I chose to make the CNN deeper, but not spend a whole lot of time on the CNN because I was very interested in building a Vision Transformer. I ended up doing a relatively simple 11 layer architecture for my CNN before moving to Vision Transformers. I could have spent more time making it deeper, finetuning models and architectures like ResNet50 or implementing a similar Residual CNN, but instead I focused on Transformers.

Solution 2: Vision Transformer (VIT)

As said above, I read about Vision Transformers, and was very interested in what it would take to implement one on my own. Transformers are a fairly new architecture introduced in the 2017 paper "Attention Is All You Need". Transformers were first used for Sequence-to-Sequence learning (such as translation from English to French), and this is how I was introduced to them in CSCI1470. In homework for the class, we had to implement both a transformer block for both encoding and decoding, and a transformer model that uses the transformer block.

However, more recent work has shown that transformers can be used for a variety of other things by separating the encoding and decoding logic, and Transformers can perform well at Sequence-to-Vector learning with its encoding logic, and Vector-to-Sequence learning with decoding. An example of Sequence-to-Vector learning can be found here, with text classification.



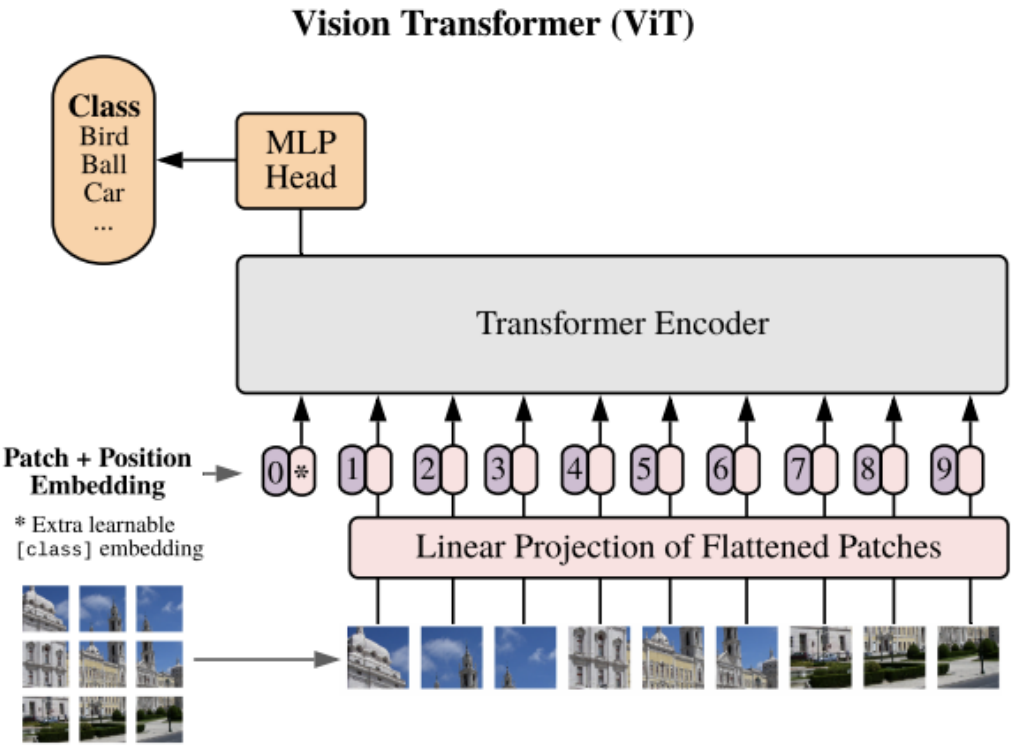
Images can also be thought of as sequences, and in "An Image Is Worth 16x16 Words: Transformers For Image Recognition At Scale", Google Research showed that Transformers can be extremely effective at classification if given enough data. This works by breaking the images down into patches like seen to the right. To create the patches, I used einops, which provides an easy API to do things like creating patches. Additionally, all einops functions are able to be backpropagated. Einops is a widely used library in Vision Transformers specifically, and the API works for most major Python DL libraries.

However each patch is a `(patch_w, patch_h, channels)` array and the transformer expects a 1d array. This can be solved in two ways, either a convolutional layer that looks at each patch and outputs the desired channels, or by flattening each patch and then passing it through a Dense Layer. I chose to do a Dense layer as we are able to both create patches and flatten the image in 1 call of einops.

Additionally, there is a little bit more magic that goes into a Transformer classifier that is different from Seq-to-Seq models, and that is adding a classification token. A classification token is essentially another "patch" that is added at the beginning of the sequence. This is used for classification, and the output of the other patches are essentially disregarded at the prediction layer. We also need to add positional encoding to the patches which tells the transformer how patches are positioned.

I chose to not reimplement the Transformer Block since Torch provides already implemented layers, and the fact that I had already implemented a Transformer in other classes. I mainly focused on the preprocessing required to create a VIT.

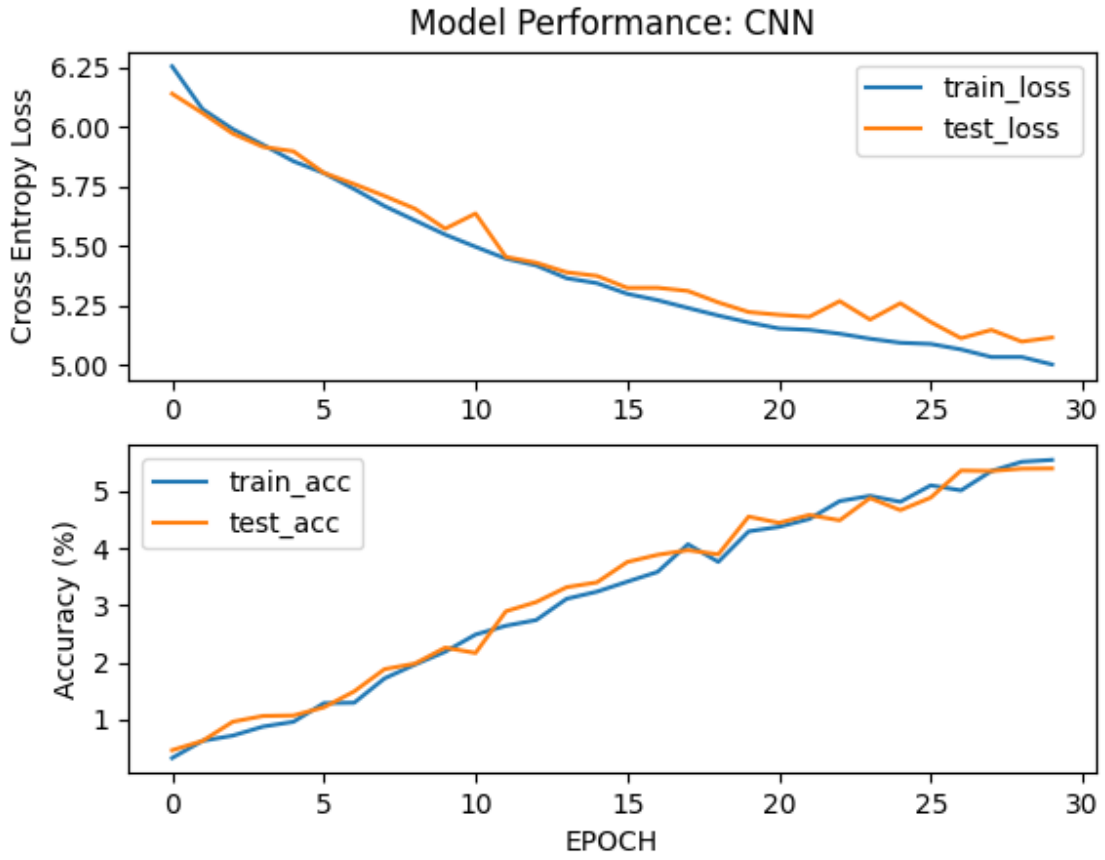
The Figure below shows a good overview of a VIT architecture, diagram taken from the VIT paper by Google.



Results

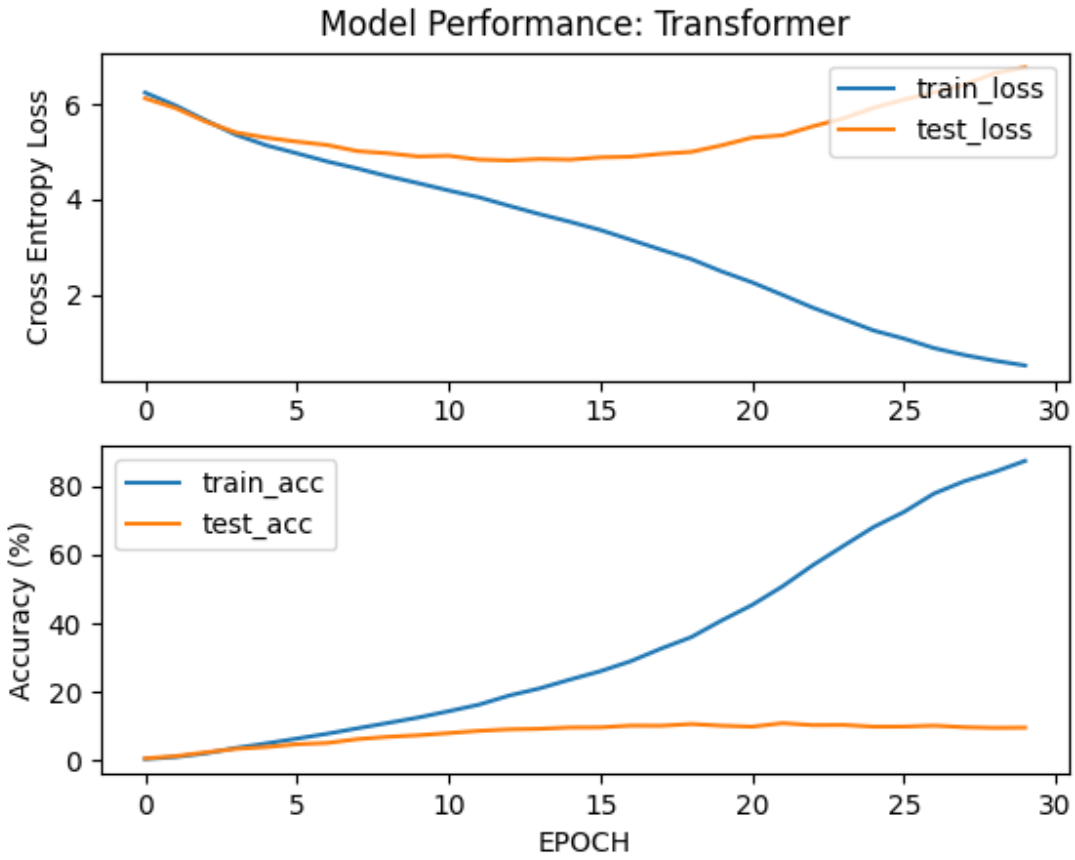
Overal, I thought that the results were a little lackluster compared to my expectations when I started the project. Looking back with everything that I know now, I understand why my results are relatively poor. I will go into more detail about why the results are poor in both CNN and VIT cases.

For the CNN, the model is simply not complex enough. A relatively simple CNN cannot learn to distinguish between the nuances of the different bird categories. We can see below that learning is extremely slow, and after 30 epochs the model only gets around 5% accuracy! If I were to continue with a CNN, I would build a large, deep, residual network that is far better at learning the small nuances.



The opposite issue exists for the transformer. The model is very good at learning to distinguish between similar images, but instead of learning species, it learns the training images. Overfitting is a substantial problem in the transformer architecture, and test loss peaks around 15 epochs. The highest performant transformer that I found for a simple VIT architecture was 128 Transformer Dimension (size of input-output for each patch), with a

Dropout of 0.5 to reduce overfitting. Performance seen below. The model reaches 10.84% test accuracy on epoch 22, but test loss is minimized at epoch 13.



As we can see the model drastically overfits due to the complexity of the model. I know now that Transformers need a lot of data in order to properly train and be able to generalize to real world data or the testing data. This can be accomplished by fine tuning existing VITs, training on other FGVC datasets like FGVC-Aircraft, or other ways. The dataset I chose was not enough to train models to be extremely accurate and performant.

Possible Future Work

It would be foolish to say that I would not change this project if I were to do it again, or say that there is not more to do. If I were to do this project again (FGVC), I would probably do it with more data/datasets, and build a model that is good at FGVC in general and then fine tune it for each dataset.

However, I learned a lot while working on this project, and read quite a few papers that put me out of my comfort zone a little bit. Some of the papers were a little hard to digest,

and when I started the project, I was looking at some of the state of the art models like the Swin Transformer, and they were difficult to read. I think that I should have started with a regular VIT, and now I feel more comfortable reading the paper, and that is something that I could do in the future.

If I were to continue to work on this project, I would increase the amount of data first and foremost, and then I would read papers on FGVC and choose something, Part-Selection Modules caught my *attention* over the course of the project. All in all, I learned a lot over the course of the project, and I am glad that I chose to build a VIT since it was a completely new model to me.