Peter Ott
G00833253
pott2@gmu.edu
CS747 - Final Project

Vision Transformers for Segmentation

Vision Transformers (ViT) are an extension of the latest sweeping innovation in deep learning: the Transformer. The researchers of the ViT paper have adapted transformers to ingest images and do not use any convolutional layers in their method. The ViT network is able to produce binary or multi class outputs given input images. My work here continues from where theirs stops. I implement an extension to the ViT architecture to allow the model to output pixels of the same size as the input image. I train and test on an image segmentation task, however I theorize that the uses of this model style can be applied to other domains where the output is an image, such as enhancing image quality or style transfer.

Architecture Disclaimer

I started by building it to the spec that the researchers described in the ViT paper. When having difficulties in getting the tensor operations just right, I used the official implementation[1] in JAX that is linked in the original research paper as a reference. I also found a 3rd party implementation[2] by the Keras team and it was quite similar to my own version because both were built off of the same blueprints. There are some differences, the Keras example implementation combined the positional embeddings and dimensional projection into a single layer. The Keras version also added the embeddings to the projected patches, when the diagram in the paper seems to imply concatenation. I did take some information from the Keras example in order to get my own version working. This is only for the base ViT model, it has no code about the new work I am adding-on in terms of image segmentation. I would like to make that known as to avoid any honor code concerns. My own original work starts after the base Transformer model and ViT model.

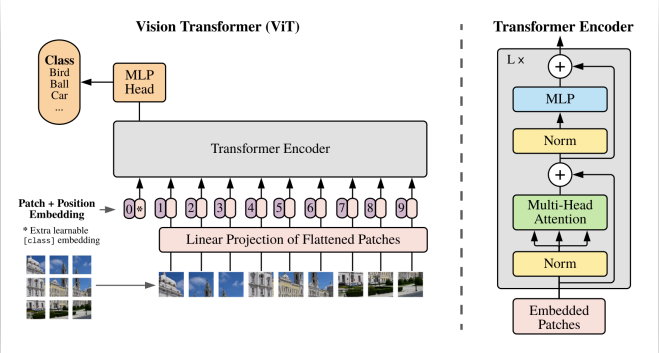

Fig. 1
ViT Architecture Overview
taken from original ViT Paper

Standard ViT Architecture

The standard ViT architecture (seen in Fig. 1) takes in images and cuts them into patches. These patches are flattened and projected into a latent space; a positional embedding is added as well. The input image has now become a sequence that the transformer encoder can

[1] https://github.com/google-research/vision_transformer

[2] https://keras.io/examples/vision/image_classification_with_vision_transformer/

process. After the layers of transformer blocks, the network will output a dense vector of features and classification can be done with that. For my Image Segmentation addition to the ViT architecture, the final classification head is not included, instead the output features are then used for further processing.

Extending ViT for Image Segmentation

The new capability my model will have is for semantic image segmentation, however the model is built such that it could theoretically do any image-to-image task, not just segmentation. For image segmentation, the model will need to output a 2D image that is "colored" based on class predictions. Historically, this task is done with a U-Net style architecture that downsizes the image while enriching with features, then it upsized back up to the input resolution. U-Net does this with convolution and deconvolution layers. Other image processing neural network also can accomplish this task, such as those in the ResNet family.

To output pixels, the ViT model needs extra layers on the end to facilitate this behavior. In my preliminary searching, I found one research paper that also modifies ViT for image segmentation, however they fall-back to using convolution operations to accomplish the goal. My objective is to stay true to the goal of the original ViT authors and abandon convolutions completely.

My architecture for ViT for Image Seg can be seen in Fig. 2. This novel architecture takes inspiration from the U-Net model. ViT for Image Seg cuts the image into patches, runs a transformer, then it does things backwards, like U-Net. It reintroduces the original patches and

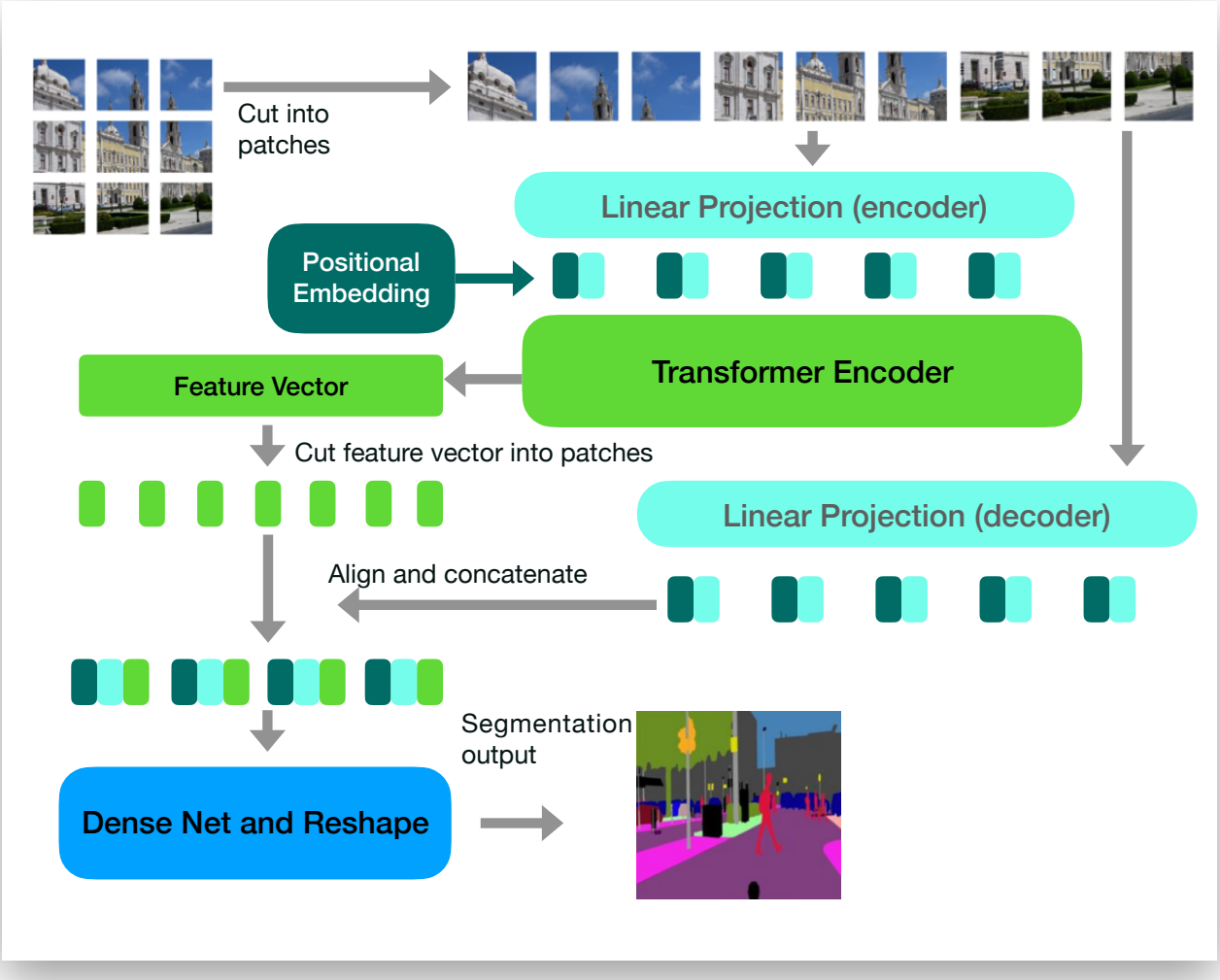

Fig. 2
ViT for Image Segmentation Architecture Overview

combines them with enriched information from the transformer body. Then a final prediction head is used to convert the enriched latent patches into pixel values.

Dataset

For image segmentation, I chose the CityScapes dataset[3]. It is a very expansive dataset that has many forms of labels for all sorts of purposes beyond just segmentation, such as depth estimation. For this project I am using the fine-grained segmentation labels. A sample data item

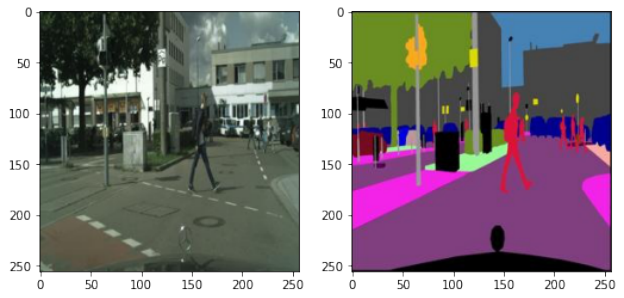

Fig. 3
Sample from CityScapes dataset
Left: input, Right: segmentation labels

can be seen in Fig. 3. Inputs are standard color images, and outputs are “images” that are colored per class. These images are photos taken from the hoods of cars that have driven around various cities in Europe. The training set has 2,975 image pairs and the test set has 500. The images are 256x256 in resolution with standard RGB color channels per pixel. I have converted the target images color range of 0-255 to floats of 0.0-1.0 for ease of modeling use. This means that displaying model outputs must be multiplied by 255 to restore the original color range.

Architecture Search and Training

As is my default practice, I started my architecture search with a very small network, and it was not able to learn any features at all. Most parameters need to be set at 256 or much higher for the model to begin to learn anything meaningful. The learning rate must also be set quite high in order for the model to learn in a reasonable amount of time. The final model was trained for 100 epochs with a learning rate of 0.0015 and weight decay of 0.0001 (via AdamW optimizer). MeanAbsoluteError was the loss function as it proved to help the model learn with more stability over MeanSquaredError. Total training time takes about 45 minutes, so I was able to through a few revisions each day. My best parameters can be seen in Fig. 4, and the full details can be seen in the python notebook.

Model training was slow, not just in time, but in the progression of loss decreasing and accuracy increasing. After the initial quick increase, the learning slows down as the model approaches its maximum. The validation metrics are also more noisy than I would like, but this is as stable as I could get them with normalization and dropout. Training graphs can be seen in Fig. 4. I believe with a larger and more diverse dataset, this architecture could be make to go further. Obviously more weights and more compute could help as well.

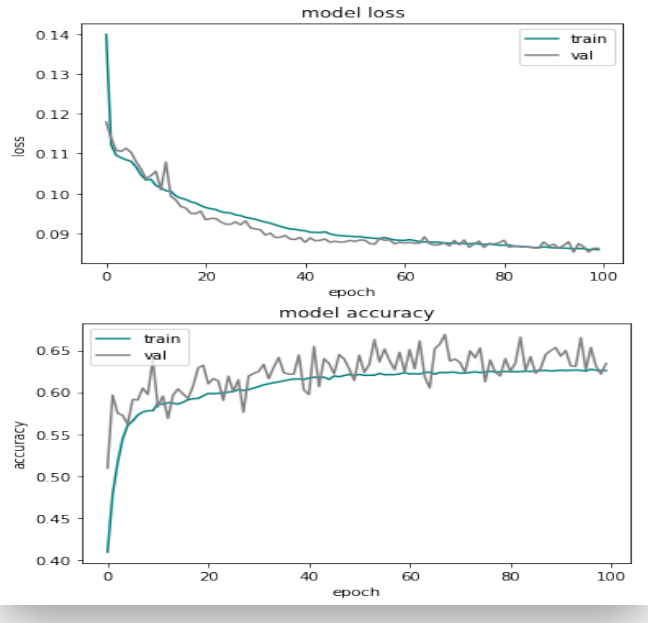

Fig. 4
Train and Val - Loss and Acc

[3] https://www.cityscapes-dataset.com

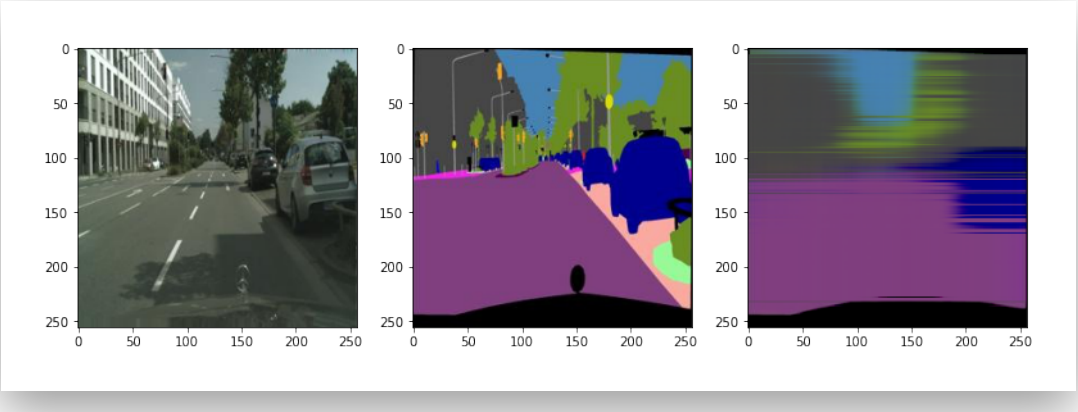

Fig. 5
Left to right: input, true labels, prediction

Final Model

After many parameter iterations I have settled on a rather large model. The model has 60mil total parameters, all are trainable. (model.summary not included to save space, but it can be seen in the notebook) The output from my final model looks approximately correct, however there are horizontal patterns that show up on most output images, sometimes vertical ones as well. My first thought was that my reshaping was buggy, however I have thoroughly checked and that is not the case. A sample output can be seen in Fig. 5. The final metrics are visualized in Fig. 6 - these are a bit strange because they increase for val and test. An overview of the parameters for the best model are in Fig. 7. Metrics for a handful of models are hosted on a leaderboard on PapersWithCode.com. My model is almost at the level of the "DeepLab" model from 2015 on this dataset. Considering my model was trained for about an hour and with no transfer learning, I say that is pretty good. The ResNet-38 model does a great job at roughly 80% accuracy. So for now, traditional convolutions are still king for image segmentation. Though nothing is able to surpass the ~85% accuracy threshold, so clearly this is a difficult task and perhaps a more innovative use of Vision Transformers can be used to further increase the performance.

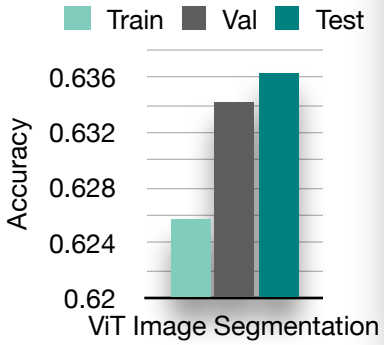

Fig. 6
Best Model Metrics

Best Model Parameters:

Total Params: 60,927,264

Input image: 256x256
Patch size: 16x16
Spatial embedding: 8
Encoder projection: 128

Transformer Blocks: 4
N Attention Heads: 8
Transformer MLP Units: [512, 1024]

Per Patch Features: 32
Decoder Projection: 128
Decoder MLP Units: [512, 512, 512]

Fig. 7
Final revision model parameters

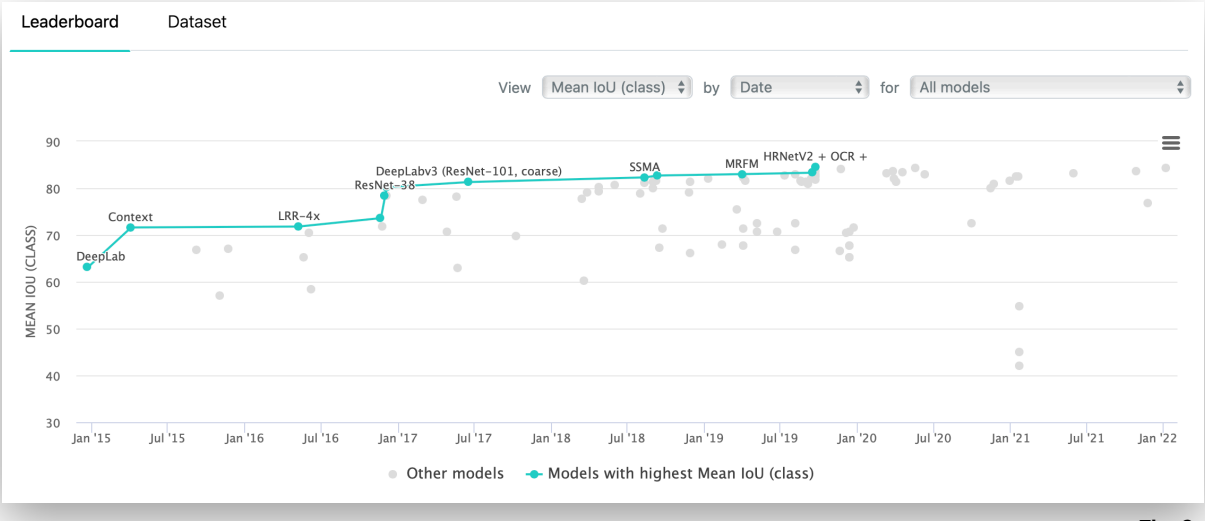

Fig. 8
CityScapes Dataset Leaderboard
paperswithcode.com

Conclusion

Building the transformer block from stock Keras components was a great learning experience in understanding how this revolutionary module functions. While it wasn't the focus of my project, it certainly was a backbone and without it I would not have been able to make my model. It was very enjoyable to combine known techniques (ViT and U-Net) and create something out of it that has not been done before (to my knowledge). I've thought that I might like research and have played around with the idea of pursuing a PhD so maybe I will. At the least, I would like to build more intuition for how to create novel architectures and get a feeling for what is more or less likely to work.

Future Work Ideas

During research and building, I experimented with designing the model as a Seq2Seq architecture. It would run the encoder half of the model as described previously (cut the input image into patches, ingest the patches as a sequence, then produce encodings via a transformer). But then on the decoding side, it would produce its sequence of patches in order starting from the center of the image and spiraling outward. This would simulates how the eye has more "pixels" in the center, so the model could make the center of the image first, then fill in the surrounding details as it works. Unfortunately I did not have enough expertise to do it properly as the masking aspect was confusing to me. I tried a few ways but ultimately could not get the model to compile. I think this would be an interesting path to explore though.

As an easier modification to the current way things are coded up, I believe data augmentation would help the model generalize better as well. For the time constraint, I did not have enough time to figure out how to modify the output image as well. The input is easy, but ensuring the output labels are modified in the same way was taking too much time, so I had dropped the idea. But perhaps in a second iteration of this endeavor it would yield better test metrics.