Frankenstein

Thomas, Lola, and Jacob

Motivating Paper

Interested in the problem of image classification and computer vision, Tatsunami and Taki propose a model called Sequencer (2022). Their objective is to rival the accuracy of Vision Transformer (ViT) networks at lower computational cost. As opposed to using self-attention layers, they use LSTMs to model long-range dependencies and mitigate the "vanishing gradient problem." In particular, they propose a bidirectional LSTM, which they term BiLSTM. Their two LSTM layers identify top/bottom oriented patterns and left/right oriented patterns in parallel. Additionally, by excluding self-attention layers, Sequencer is more resolution flexible than existing image classifiers.

However, the authors admit that there is more work to be done to understand exactly how Sequencer learns. By construction, it is expected that Sequencer learns orthogonally oriented patterns that are repeated across the input sequence. They offer some plots of the Sequencer's effective receptive fields (ERFs) to compare to existing architectures. Sequencer's ERFs always form a distinct crucifix pattern that is drastically different from existing models, suggesting that it learns differently. Visualizing hidden states of the model reveals that they interact with each other over the horizontal and vertical directions as expected given the BiLSTM processing layers.

As it relates to our project, we were inspired by this paper but do not replicate their network exactly. What was interesting to us was the concept that long-range dependencies, rather than simple global dependencies, matter for image recognition. In this course, the models we used for images and the models we used for sequences were distinct so we are interested in whether combining the flavor of an RNN, via LSTM, and a CNN would yield a more effective image classifier model.

Our Goal

To train a model that uses LSTM and other techniques from class to identify sketches drawn by the user.

Our Data

Google's Quick, Draw! is a crowdsourced collection of 50 million different black and white sketches of particular shapes that a user is prompted to draw in under 20 seconds. Our model is trained on a selection of these shapes that were chosen for being fairly distinct from one another. They are also well drawn overall on a visual inspection of the data. Some prompts like "diamond" are drawn like the gem 50% of the time while the rest of the time they are drawn like the shape. Lastly, the shapes are relatively simplistic and can be drawn in only a few strokes. The complete list of shapes chosen is:

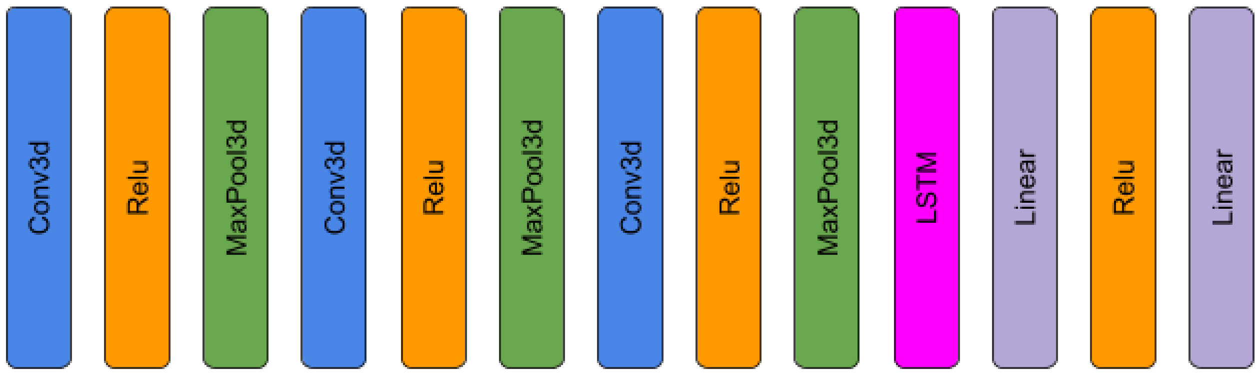
1. Circle
2. Cloud
3. Donut
4. Eye
5. Fish
6. Hexagon
7. Hockey stick
8. Hourglass
9. House
10. Ice cream
11. Line
12. Mountain
13. Octagon
14. Paper clip
15. Smiley face
16. Square
17. Sun
18. Triangle

The raw data is a single json file per shape with entries that contain an array of strokes in what we term stroke form. Each stroke array is *[stroke_1, stroke_2,....]* where each *stroke_i* is a 2D array of (x,y) pairs in the form *[[x_1,x_2,...,x_n] , [y_1,y_2,...,y_n]]*. The original sketches are on a 256 by 256 grid.

We parse the raw data and split it into training, test, and validation sets formatted in the standard json format that the *DataManager.py* module from the labs expects. Sketches originally in stroke form are converted into one tensor of *pad_len* bitmaps of size *m* by *m* where *m* is the specified output size and *pad_len* is the specified maximum number of strokes allowed per image. We cap the number of bitmaps at *pad_len = 8* to save memory and computation time based on an analysis of the distributions of the number of strokes in the sketches. The zeroth layer of the formatted stroke array is a bitmap just containing the zeroth stroke. The first layer will be the previous bitmap with the first stroke added on. The second layer will be the previous bitmap with the second stroke added on and so on. If a particular sketch has fewer than *pad_len* number of strokes, then after we reach the layer where the sketch is complete, all later layers will be a zero-filled tensor.

Our Neural Network

Our Frankenstein architecture eats tensors of size (BATCH, 8, 64, 64) and begins with three CNN layers. In between each CNN layer, there is a relu layer and max pooling layer. The max pooling layer has kernel size 2x2 and a stride of 2, which reduces the last two dimensions by a factor of two. Each CNN layer also outputs a specified number of channels. Thus, after the last MaxPool3d layer, the output of the model is of shape (BATCH, 8, Num_Channels, 8, 8) where Num_Channels is the number of channels the third CNN layer outputs. Note that by shrinking the last two dimensions of the tensor by a factor of two at each CNN layer, we have dramatically reduced the size of the problem by the time we reach the LSTM. The LSTM portion of the network is composed of three layers and implemented in PyTorch based upon Sak et al. 2014. Lastly we apply a linear, relu, and linear layer to compute linear regression, the negative log of the probabilities, and classify the output probabilities. A diagram of these layers is shown below.



The model is able to recognize how a shape is constructed because the formatted sketch input is a tensor of sequential bitmaps documenting how the sketch is composed of individual strokes. The neural network has the ability to not only identify patterns in the final shape, but also learns what stroke patterns and sequences produce a specific shape. In particular, the LSTM layer can look for patterns across the 8 or fewer strokes and weight their importance towards classifying the final shape in a way that the CNN can not do on its own. It is important to note that when the input reaches the LSTM layer, it is still a batched four-dimensional tensor (the original three dimensions with an additional channel dimension) with eight 'frames' (i.e. (BATCH, 8, NUM_CHANNELS, WIDTH, HEIGHT)), so the LSTM layer can correctly analyze the sequential stroke data.

The regular CNN-only model that is referenced throughout the paper is identical to the Frankenstein model, but the LSTM portion of the network is omitted.

Training

The data used to train our models was a subset of Google's Quick, Draw! Dataset. This subset had 10,000 examples of each of the 18 classification categories the model was trained to detect. We partitioned this data into a training set with 80% of the data, a validation set with 10% of the data, and a test set with 10% of the data. Thus, we had ~144,000 8x64x64 tensors to store in memory during training. To avoid running out of RAM, we utilized a torch DataLoader object that stored our images on disk as JSON files and read them in as necessary during training.

The "stroke" data from Google's Quick, Draw! dataset is provided for 256x256 images. If we did a 1:1 conversion when generating our three-dimensional bitmapped images, our dimension *m* from the data section would be 256. Thus, with *max_len* set to 8, our unbatched input tensors stored in JSON files would have shape (8, 256, 256). To increase the speed of training, we applied a maxpool function to each 256x256 bitmapped image with 4x4 kernel and a 4 unit stride to truncate each image in our input sequence to size 64x64. Thus, with *max_len* set to 8, our non-batched input tensors to the model were of shape (8, 64, 64). Note that images with less than eight strokes were padded with two-dimensional tensors filled with 0 to reach a 0th dimension of 8.

We utilized mini-batch gradient descent with batch size 32 to train both our Frankenstein model and the CNN-only model. Thus, the batched tensors fed to each model had shape (32, 8, 64, 64). Each model was trained for 40 epochs, and in each training instance, we selected the version of the model that performed the best on the validation set after completing a given epoch.

The Interface

In *paintCanvas.py*, we provide the user an interface to draw sketches on a 64 by 64 grid. Users can draw sketches that contain up to eight strokes and click on "clear" to restart them. Once complete, users can click on "classify" to have either Frankenstein (by default) or a more basic CNN only model (by clicking on switch) that is the same as Frankenstein without the LSTM layer classify their image as one of the eighteen trained shapes in the bottom right corner. Additionally, by using the "switch" button and reclassifying an image under a different model, you can

Extensions

Given more time, there are several interesting extensions we have identified. Because the sketches are from an online game where users draw, the shapes tended to fill the available grid and were often oriented in some natural direction. Thus, Frankenstein is not trained on different scaled sketches or various orientations of most of them. One could perhaps find a more robust dataset to train on. Or, one could augment the dataset we used, by randomly shrinking and rotating the sketches.

Originally, we wanted the model to tell you the size and orientation of the shape it classified, that way it could replace your drawing with a properly sized and rotated generic image of the same shape. To do this, the training data we used would have to be augmented with true answers about the size and orientation of the sketches in order to do this. With millions of sketches, this was not feasible to do however we did consider writing scripts that could compute this data from the sketch. Another issue with this idea was that some shapes do not have a true orientation. For ice cream it is clear that the cone should point down and rotations are variations from that. But what is the normal orientation of a sun sketch and what does it mean to rotate the sun 90 degrees clockwise?

Lastly, it would have been interesting to have the model automatically classify and replace shapes it sees as they are being drawn. For example, I draw the cone of an ice cream cone and then the model sees that and finishes the sketch for me. One challenge with this knowing when to run your model on the partially finished sketch. For example, the cone of an ice cream come might be an ice cream cone and it might also be a partially finished triangle. One solution to this would be to potentially run the sketch through the model after every stroke and only replace the partial sketch if the model classifies the sketch with a certain confidence that is higher than some bar that we set.

Sources

Haşim Sak, Andrew Senior, & Françoise Beaufays. (2014). Long Short-Term Memory Based Recurrent Neural Network Architectures for Large Vocabulary Speech Recognition.

Yuki Tatsunami, & Masato Taki. (2022). Sequencer: Deep LSTM for Image Classification.