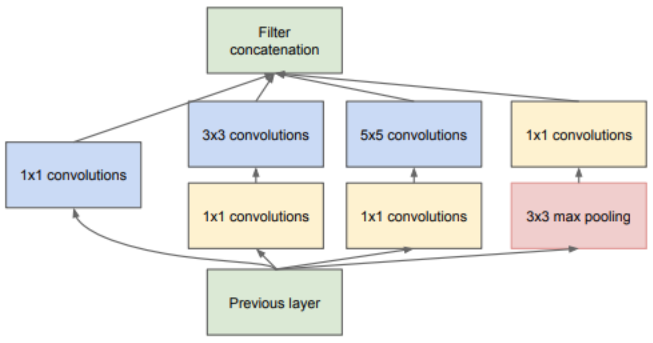
Using Convolutional Neural Network to Classify House Numbers from Google Street View Dataset

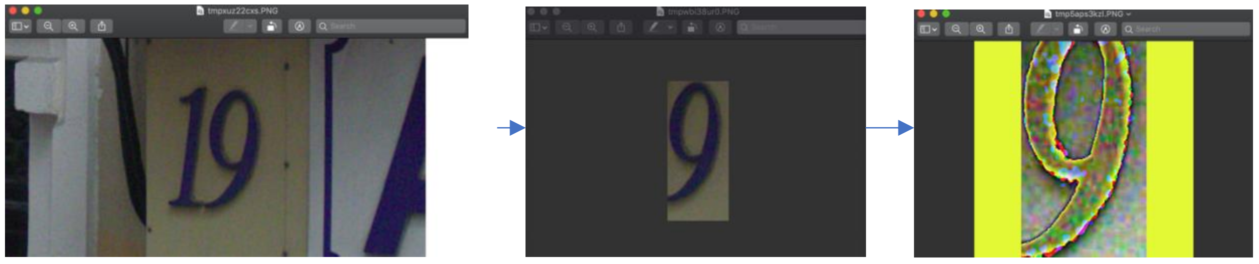
Khwaja Mustafa Sidiqi
ksidiqi@gatech.edu

There are three commonly used classical CNN LeNet-5, AlexNet, and VGG 16. They are mainly comprised of simply stacked convolution layers. The more modern architectures for convolutional networks focus on new innovation and efficacy [5]. Some of the well know modern CNN are Inception, ResNet, ResNeXt, and DenseNet. In particular, Inception was developed in 2014 by Google commonly referred to as GoogleNet. Inception was crafted to increase the depth and width of the neural network without increasing computational need. The following diagram illustrates the parallelism and the use of 1x1 to aggregate the results from different scale [2]. Some of the key takeaway from this research paper is how computationally intense large convolution filter are. A convolution with large filter like 7x7 is computationally disproportionally expensive in compare to its benefit. The paper suggests stacking 3x3 convolution can be computationally inexpensive in contrast to having a single 5x5. Following this paradigm, Inception model is known to have a really long depth.



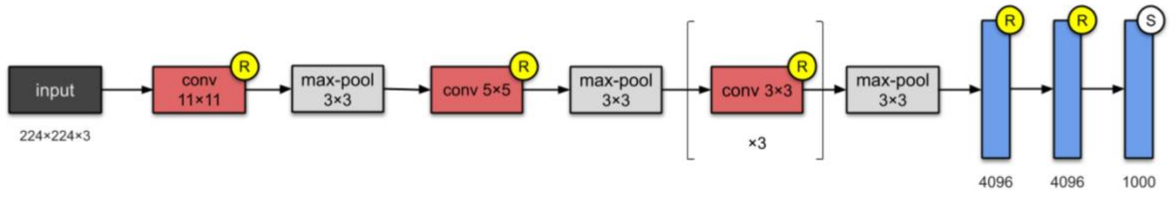
(b) Inception module with dimension reductions

There are four main parts to my CNN: getting the data, creating a model, training and evaluating the model, and predicting. I implemented three different ways of loading data: mat file, jpg, and video. Moreover, I utilized pytorch library dataset and dataloader and best practices to implement batching [3]. Additionally, I used Google cloud with nvdia tesla v100 for this project.



The following photos shows how a jpg file is cropped and transformed before sending it to the model with a proper label for training. Transformation includes resizing, center crop, and normalizing. It is worth noting that the training data included bounding box that allowed me to crop and zoom into the right digit. For images without bounding box, I used combination of image pyramid, sliding window, indices remap, and non-maximum suppression. For each level in the gaussian pyramid a tall bounding box extracts part of the image to send over to the model. The sliding windows have approximately 75% intersection to prevent splitting a digit. Image pyramid introduces a new complication with indexing. A bounding box at the top of the pyramid needs to be mapped back to the base of the pyramid. I resolved this issue by keeping a second matrix of the indices that gets resized exactly like the image. Thus, at the bottom of the pyramid the indices matrix at (x…x2, y…y2) will map to single location on the top of the pyramid. Lastly, we will get multiple bounding box enclosing a single digit due to intersection of the sliding windows. Ideally, these boxes should either be merged or only the most centered bounding box be kept. I used non-maximum suppression to resolve this problem. Pytorch library provides ops.nms we can utilize.

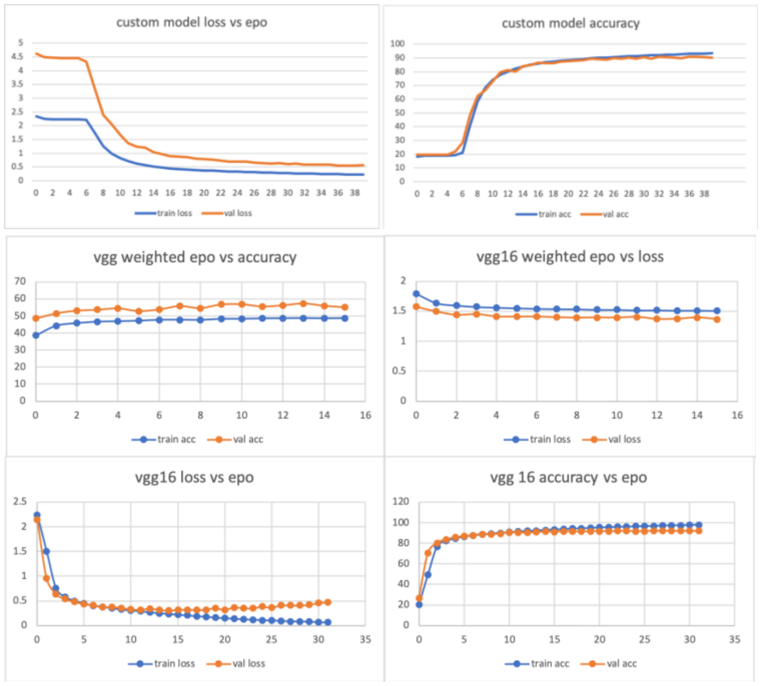
I experimented with three CNN model two imported and one custom: vgg16, vgg16 pre-trained, and a custom model based on AlexNet. For the pretrained VGG, before training I had to freeze the hidden layers, so that when I start training it doesn't override the weights. Moreover, for both VGG models, I had to modify the classifier and limit the number of classes to 11 one for each digit and one for non-digit. For the custom model, I implemented a model based on AlexNet with couple modification. The following diagram illustrates AlexNet architecture. I added two extra conv 3x3 making the last conv chain x5 instead of x3. Moreover, I changed the classifier to 11 classes. I used Pytorch forward step for optimization.



Once a model is instantiated, it needs to trained and validated. I created a function precisely to do that. Eval function inside of ModelTrainEval is the core of the project. It is responsible for training an optimal model. The eval method takes an optimizer, epoch range, training data, and testing data. For criterion I use Cross Entropy Loss and for optimizer I use stochastic gradient descent (SGD) [4]. The decision to use learning rate=0.001 was based on experiment of varying lr from 0.01 to 0.0001. Additionally, Cross Entropy Loss was the preferred object for VGG. To get a deeper understanding of how epoch effects my model, I set epo=50. In each iteration, I switch the model to train mode, iterate through all the training data, and record its performance (loss and accuracy). Then, I switch the model to eval mode and iterate through all the test data without computing the gradient and optimization step since we don't want our validation change the model. Similar to training data, loss and accuracy is recorded. At the end of each iteration, the test data accuracy is compared against the best model so far. If a better accuracy is discovered the best model is updated with the current model. Additionally, there is a way to break out of the training loop without iterating through the full epoch range. If the algorithm detects the model is not improving by much, it breaks out. I implemented this mechanism by keeping track of the testing data accuracy for the last seven iteration in a minimum priority queue. At the end of each iteration if the current accuracy is worst or equal to the least accurate record in the last seven iteration it breaks out of the loop. Before exiting the eval function, I load the best model's states into the current model ensuring the best model is returned. The model is then saved into disk for future usage under the directory "./model"

To classify an image, the model is switched to eval mode. An iterator of images is passed to the model for predication. The output of VGG is passed to a softmax function to calculate the percentage associated with each class. The softmax is required since the official VGG architecture doesn't return probability distribution. If the percentage is less than 95%, then I ignore this image and move on to the next image. If the percentage is greater than 95%, then the pyramid layer, digit, and bounding box is recorded. After iterating through all the images, for each layer of the pyramid we perform non-maximum suppression on all the bounding box with the percentages treated as the scores.

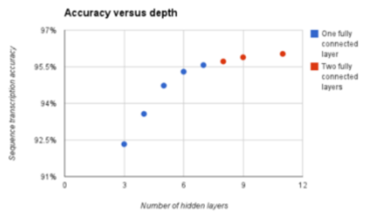
In order to have a good understanding of how each model performs, I plotted the data I recorded during the training phase. For each model I created two charts showing the recorded value for both training data and testing (val) data for each epoch. First chart shows the value of loss, and the second chart shows the model's accuracy.



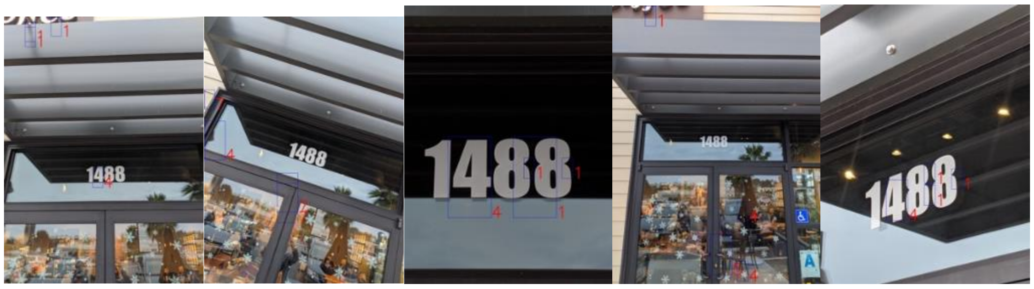
There are couple observations worth noting. First, all the three models braked out of the training loop before reaching epo=50. VGG weighted (pre-trained) finishing the fastest in terms of number of epochs followed by VGG from scratch, and finally the custom model. In term of computational time, custom model finished the fastest followed by VGG16 then VGG16 pre-trained. Second, VGG16 pre-trained started off pretty accurate but didn't adopt well whereas the other models started off pretty bad but adopted well. Third, custom model and VGG16 from scratch had approximately the same accuracy rate for both training data and testing data throughout each epo. I was expecting the accuracy for testing data to be below the training data which isn't the case. I did make sure while I am testing the model with the test data, the model is not learning. Fourth, the model's accuracy for testing data is as follows: VGG16 from scratch, custom model, and VGG16 pre-trained. Thus, for predication I use VGG16 from scratch.

To summarize, Vgg16 from scratch was the most accurate model with 97.845% accuracy on the full training dataset and 91.71% accuracy on the full testing dataset. In a research paper titled "Multi-digit Number Recognition from Street View Imagery using Deep Convolutional Neural Networks", by Google, it is stated, "We have evaluated this approach on the publicly available Street View House Numbers (SVHN) dataset and achieve over 96% accuracy in recognizing street numbers." Assuming the they are referring to the testing data, my model missed the state of art model by 5%. In Machine Learning world, 5% is significant; therefore, there is a lot of room for improvement.

Nonetheless, there are at least two ways I can improve my model. First, introduce random transformations to the training data. For example, randomly rotating each image between (-15, 15) degree; randomly scaling each image by factor of 2; lastly, randomly changing the brightness, contrast and saturation of each image. Performing these random transformations to our training data would increase the accuracy of our testing data. Secondly, using a more modern architecture would improve my accuracy. To support my proposal, the following chart from Google's paper [1] shows the number of hidden layers vs accuracy. The more modern architecture is deeper and wider than VGG16. Hence, using model like Inception would improve the accuracy.



Even with accuracy of 92%, unfortunately, I wasn't able to detect all the digits from an image. I verified the sliding windows does pick up the digits with different scales. The image is passed to the predict function which should give us the right classification 91% of the times. Moreover, anything the VGG16 scores less than 95% I discard it. From my observation, the probability constructed from VGG16 outer layer isn't correct. For example, I get 100% prediction with label 9 when 9 doesn't exist in the image. If I had known this I would have used keras that provides built-in model.predict_proba(X) whereas in pytorch I had to implement by own probability prediction using SoftMax. On a closer inspection all the detected digits do have shape similar to the digit it is tagged. For example, the side of 8 makes 1, the space between two eights make 1, corner of the door at an angle makes 7, ect…



[1] https://storage.googleapis.com/pub-tools-public-publication-data/pdf/42241.pdf
[2] https://arxiv.org/pdf/1409.4842.pdf
[3] https://stanford.edu/~shervine/blog/pytorch-how-to-generate-data-parallel
[4] https://pytorch.org/tutorials/beginner/blitz/cifar10_tutorial.html
[5] https://www.jeremyjordan.me/convnet-architectures/
Video: https://youtu.be/4cDuSlh7vPs