Guess-a-Sketch

Lucia Zacek (llz22), Urvashi Deshpande (uad3), John Fernandez (jmf433)

Presentation

https://drive.google.com/file/d/1KiRD0MVFUcbW-8oZ4qk2G7hSSWhfZpFh/view?usp=sharing

Github (note: this does not include the most recent RF model as the file size was too large)

https://github.coecis.cornell.edu/llz22/Guess-a-Sketch

AI Keywords: Computer Vision, Machine Learning, Deep Learning, Convolutional Neural Networks, Support Vector Machines, Random Forest, Game Playing

Application Setting:

We developed an AI model to classify drawings based on the online game Skribbl.io, which is similar to pictionary. Players take turns either drawing a specific word or guessing what another player is drawing. Our model takes in data in the form of black-and-white sketches and word length, then predicts a label for each image. This model was trained to work exclusively in the constrained environment of Skribbl.io, where the length of each word is known. During development, we have limited the setting to a word bank of 250 labels.

No other students were involved in the development or testing of our models

Part 1: Project Description

The Original Plan:

Initially, we wanted our models to very closely resemble a human playing the game Skribbl.io. We had two main objectives for our models: one, that they would be able to classify drawings with a high accuracy, and two, that they would be able to do so faster than a human player. We used the dataset linked in the references below to train our models. It contains 20,000 black-and-white images, split equally into 250 different classes.

We originally intended to construct three different models to make these image classifications: SVM, Random Forest, and CNN. Each of these models would take in a set of testing data in the form of images and associated word lengths, then try to classify them based on a predetermined word bank. For the final model, we would select the most accurate (as measured on a validation and testing set) model among the three listed above.

As further goals, we wanted the models to be able to take in partially completed images (or images in the process of being drawn) and make accurate predictions. We also wanted to expand the word bank from the original 250 labels to the full Skribbl.io dictionary. This would more fully model the Skribbl.io setting, where players can guess what image is being drawn as it is in process. We would then reward the model for guessing correctly quickly, and expand our evaluation of the model to include speed.

The Modifications:

As we started working on developing the Random Forest and CNN models, we ended up having to dedicate significantly more time to processing the data and debugging than expected. As a result of spending more time on these two models, we unfortunately were unable to implement the SVM model.

There was some overhead where we had to develop scripts to construct training/validation/testing splits from the original dataset, as well as overhead in downsampling the images. The original image sizes in the reference data was 1111x1111, and we had to downsample to 50x50 for the random forest model and 256x256 for the CNN model. To do this, we used computer vision techniques including applying gaussian filters before downsampling (see AI Core).

Another modification we made was increasing the number of models we were training. To take full advantage of the Skribbl.io setting where word length is known, we decided to train one model per word length. As a result, we ended up training a total 16 models each for word lengths between 2 and 18. This gave each individual model more predictive power by limiting the set of words it had to differentiate between. We implemented this for both the Random Forest and CNN model.

While the Random Forest model was time-consuming to develop, it was able to achieve 22% accuracy without many modifications. However, memory limitations meant that we were unable to train the Random Forest model on the full dataset or to the full depth. We were only able to train this model on 7500 images, to a maximum depth of 500 per tree, sampling 500 pixels per split, and constructing 10 trees per forest. Additionally, we implemented bagging as a variance-reduction technique.

The CNN model was challenging in that it was originally struggling to learn the patterns in the images, and consistently predicted the same word for all testing images. We made the same modification as in Random Forest with creating a new CNN classifier for each word length. Further, while we did not specify parameters in our original plan, we had to do a lot of parameter tuning in order to achieve a model that could actually learn patterns in our data. This process of iterative development is explained further in the AI Core and evaluation sections.

As a result of the time spent fine-tuning the Random Forest and CNN models, we ended up not implementing the further goals. This includes the goals to pass in work-in-progress testing images to evaluate speed and expanding the word bank. Our final result was a set of two models for classifying completed black-and-white images in the limited 250-word bank, given word length as in the Skribbl.io setting. While the models could theoretically work on incomplete images, we have not tested our models in this setting and cannot guarantee their speed or accuracy.

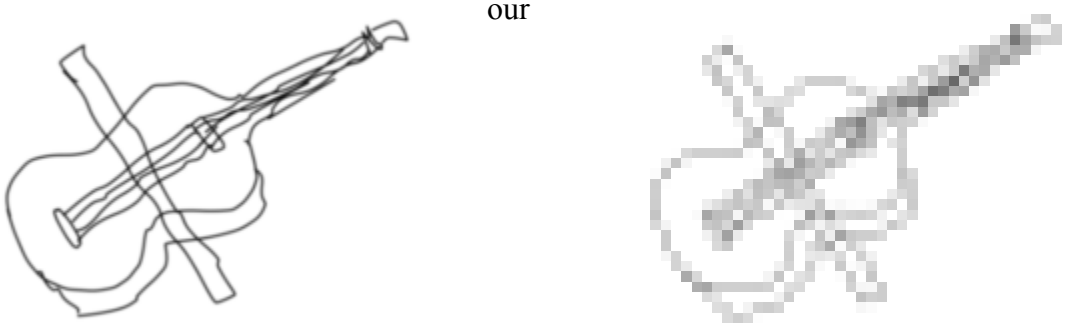
The AI Core:

At the core of our project were several concepts from computer vision and machine learning, as follows.

Gaussian Pyramid

In order to condense our data further for our two models to train more efficiently, we used Gaussian Pyramids in order to downsample our images. Given that our data contained 20,000 images, each being of size 1111x1111, we wanted to downsample each image to be of size 256x256. In order to do this while retaining as much information as possible, we used the concept of Gaussian Pyramids. The key idea here is that we first blur the image and then downsample to half the size. By doing this, we are able to prevent any aliasing artifacts that may occur otherwise had we not blurred the image first. Since we started with an 1111x1111 image, we first blurred the image and then downsampled the image into a 512x512 image. Typically, you want to half the image every time you blur in order to retain the information in the image at every step. Since for this first downsample we did not have a perfect 1024x1024 image, we reduced the image to 512x512 to obtain a cleaner downsample when we downsampled for the second time. Given this new 512x512 image, we once again blurred this image before downsampling in order to ensure that our downsampled 256x256 image did not contain any aliased artifacts that may impede our models ability to train.

At first, we trained our models by just downsampling an image normally from the original 1111x1111 shape into 50x50. Doing so, we lost a lot of information and got an aliasing effect apparent on most of our images. Comparing these original downsampled images that were not blurred at all and just downsampled all at once to our final datasets we trained our model on created a very apparent difference in our models' ability to learn. For example, the following are two images selected from our training data we used to train our models. The image on the left hand side is from our



256x256 final dataset that is blurred between subsampling twice and the right hand side is the 50x50 image that was just subsampled from 1111x1111 to 50x50.

256x256 image **50x50 image**

Random Forest

When developing the Random Forest model, we not only implemented the model from scratch, but also expanded on it using machine learning concepts. We felt that Random Forest would be interesting to use in this setting due to its ability to recognize patterns over sequences of pixels, even if it was not translation invariant. In constructing the model, we were able to take advantage of several aspects of our setting, including the assumption that word length is known and the knowledge that many training images are informationally sparse. For the former, we constructed a forest of 10 trees for each word length. For the latter, we randomly sampled 500 pixels as potential tree splits, making sure to not sample from pixels that were white for all images. While we would have liked to further expand our model (for example, increasing depth would have allowed it to detect more patterns on a macro-scale), we were unfortunately limited by our computing power.

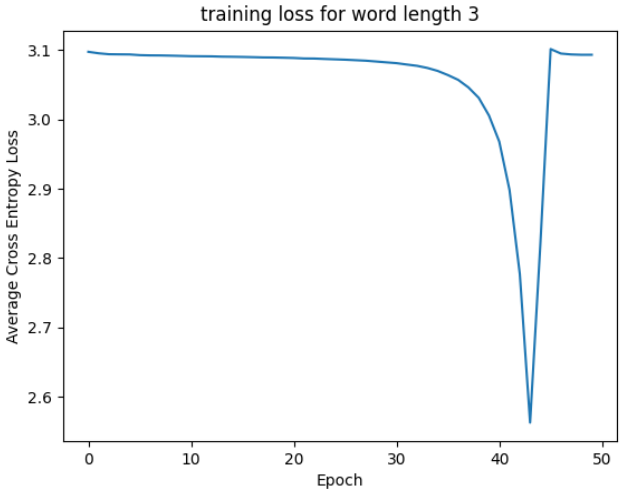
However, we were able to implement some machine learning strategies such as bagging in the construction of the Random Forest model. When constructing each tree in each forest, we subsampled a new dataset of size n (7500) from the original dataset. Since we sampled with replacement, this allowed some images to show up multiple times in the training datasets. We decided to use bagging with the intention of reducing the variance of the results. By constructing 10 trees per forest and taking the most common prediction for a given image among these trees, the idea is that the most common predicted label will have smaller variance than if we had simply trained a single tree. Naturally, this increased the computing time and power required to train the model.

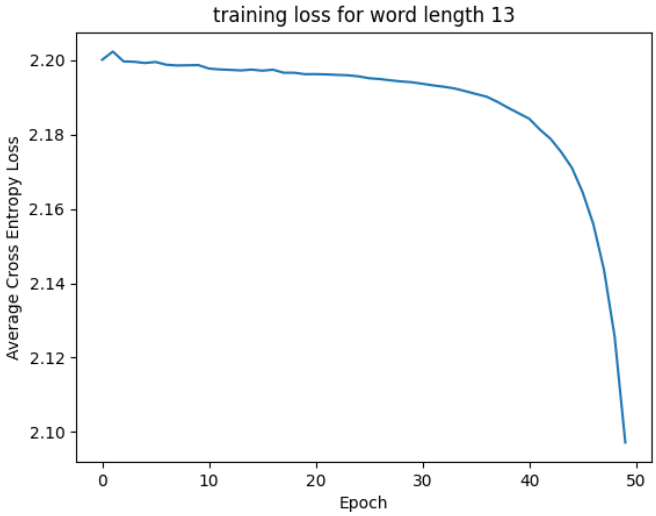
CNN Model

To develop the CNN model, we referred back to our resources from our previous machine learning course and reviewed PyTorch's Conv2d library, which we used heavily in this project. Initially we had three convolutional layers and a fully connected layer. We were also training images from all 250 classes together before making the modification described above, where we train a separate CNN for each word length. In the first iteration of our model, we trained for 10 epochs with a much smaller learning rate, and we noticed that the model always predicted the same label, which suggested that it wasn't actually learning anything. The accuracy of this initial model was 6.4%, the same as randomly guessing. Another challenge we faced in the beginning was figuring out how to feed images into the CNN model using the data loading methods we created for the Random Forest, and how to fine-tune the channels in our convolutional layers. While ensuring that the images and labels were being loaded correctly into the model was a more trivial issue compared to problems we faced later, it took away some time from the overall project which we would have otherwise used to implement more interesting features.

In our second attempt at fixing the model, we added a pooling layer before the fully connected layer, and a separate function to train separate CNN models for each word length present in the 250 classes. In total, there were 16 different models trained, similarly to Random Forest. We also increased the number of epochs to 30, 40, and 50 to check whether it was just a matter of not having trained the model on enough examples, and tested many different learning rates between 0.0001 and 1. The accuracy went up only slightly to around 8%, and after consulting with our TA, we added another two convolutional layers and added the Gaussian downsampling to make sure that the images retained the integrity of the drawings they were trying to represent. We found that the model was very sensitive to the learning rate. With large learning rates, the model failed to converge, and simply oscillated between predicting different labels for all images. Small learning rates led to extremely slow learning, to the point where little to no progress was visible in the first 10-20 epochs of learning.

Finding a middle ground was challenging because we wanted a learning rate small enough to consistently learn while not overshooting when stepping. We also had time limitations; for example, we could not simply keep making the learning rate smaller and number of epochs larger as we wanted to be able to train the model overnight. Below is a training loss curve (not from our final model) that illustrates this difficulty:





These models to the left were trained using the exact same parameters, with a learning rate of 0.01. Notice that the loss for word length 13 consistently decreased, and still had further room to decrease if we increased the number of epochs. On the other hand, the loss for word length 3 initially decreased and then spiked up at a later epoch. While decreasing the learning rate further could help reduce the chance of an overshoot as shown in the learning curve, the model was already learning very slowly and we did not have the overhead to significantly increase the number of epochs.

Instead of further decreasing the learning rate and increasing the number of epochs, we decided to take advantage of other machine learning techniques such as learning rate decay and momentum.

The first modification we made was to stabilize the performance of models with fewer classes. We found that using the same learning rate for all models was ineffective; models with fewer classes tended to oscillate

significantly more when using the same learning rate as those with more classes. To improve this, we set the learning rate to be variable based on the number of classes:

$$lr = \frac{0.01}{(1 + \frac{1}{num\ classes})}$$

As the number of classes approaches infinity, this learning will approach 0.01. For models with fewer classes, the initial learning rate would be as small as 0.005. Making this change significantly stabilized the loss curves of models with fewer classes, and did not affect the models with more classes.

Although this change helped make learning more consistent across the different models, it did not address the overshoot issue described above, which often also happened in the presence of many classes. Two machine learning techniques we applied to try to address were learning rate decay and momentum.

We added a decay rate of 0.99, such that the learning rate in a subsequent epoch is 99% of the previous. This would allow the learning rate to still be large initially, when learning is slow. It would also allow the learning rate to gradually decay such that these "overshoots" are less likely further into training, when the learning rate has significantly decayed.

Our second modification was adding a momentum factor to our SGD optimizer. We tried a momentum of 0.5. There were two major advantages to this:

1. The model would be less likely to get stuck in local minima. We identified that guessing a single label was likely a local minima, as the loss of our model would typically stop decreasing and simply plateau after one of these "overshoots".
2. Once the model begins learning, the momentum will build more inertia in the direction of learning, leading to faster learning.
3. This same inertia generated could help overcome some of the noise from randomly sampling batches and iterating through the dataloader each epoch.

Overall, we hoped that making these modifications would allow us to overcome the challenges we faced above, preventing the model from oscillating between guesses and preventing large overshoots. See the evaluation section for our final quantified results.

Part 2: Evaluation

Because we did not get to implement the further goals of making the model quickly guess work-in-progress images, all of our evaluation focused on the accuracy of the models on completed images. For the random forest model, we were curious if we could achieve an accuracy better than randomly guessing. Theoretically, we believed that the CNN model should achieve higher accuracy than Random Forest because of the translation invariance.

Random Forest

At the time of training the random forest model, which was early in the semester, we were only using a training/testing split. We asked ourselves whether we could **construct a random forest model that actually learns on non-traditional data (images) and achieves**

better-than-random accuracy? Our first construction of this model did not take into account word length, and it was only able to achieve a final testing accuracy of 2%. Had it guessed randomly, it would have achieved an accuracy of:

$$\frac{1}{num\ classes} = \frac{1}{250} = 0.004$$

In other words, the random guessing accuracy would be 0.4%. While it was a good sign that random forest was doing better than randomly guessing, 2% was a very poor accuracy and we sought to increase it further.

To increase accuracy, we asked ourselves **how can we take advantage of the Skribbl.io setting to make better assumptions about our data?** We came to the conclusion that there were two major assumptions we could make: black-and-white sketches are sparse, and word lengths are known. To take advantage of the former, we only subsampled pixels (when searching for ideal node splits) that were not sparse. For the latter, we constructed a different forest for each word length. Then the random guessing accuracy of a particular word length could be defined as:

$$r_i = \frac{1}{n\ word\ length\ i}$$

However, because each word length has a variable number of classes, this was a challenging metric to standardize. Instead, we evaluated the performance of our *overall* set of all 16 forests. To do this, we passed in a testing set of 1250 images and word lengths. Each image was tested on the model for its respective word length. Our final, overall accuracy was **21.84%.** This was a significant improvement over the original version. We evaluated this accuracy against the random guessing accuracy given word length:

$$\frac{1}{avg\ num\ classes} = \frac{1}{16} = 0.0625$$

We felt that this was a success; traditionally, random forest is not a strategy that is recommended for images because it does poorly when images are translated to other parts of the canvas space. We were curious whether there are use cases where random forest can actually perform with decent accuracy. We found that in our case, where we have a relatively uniform dataset and all images generally take up the entire canvas space, having translation invariance is not a critical issue and random forest can still perform with accuracy significantly better than guessing.

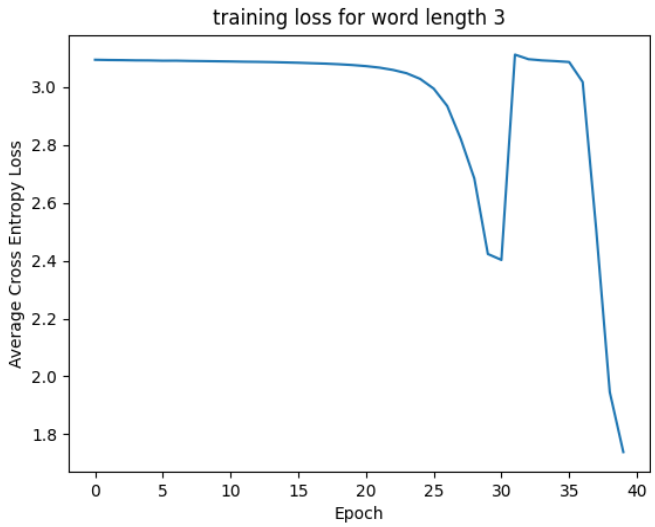
CNN

In constructing the CNN model, we not only wanted to beat the random guessing accuracy, but we also wanted to beat the random forest accuracy. Traditionally CNN is used for images due to its ability to recognize local patterns with translation invariance. As a result, we asked **whether CNN will perform better than Random Forest on these sketches?**

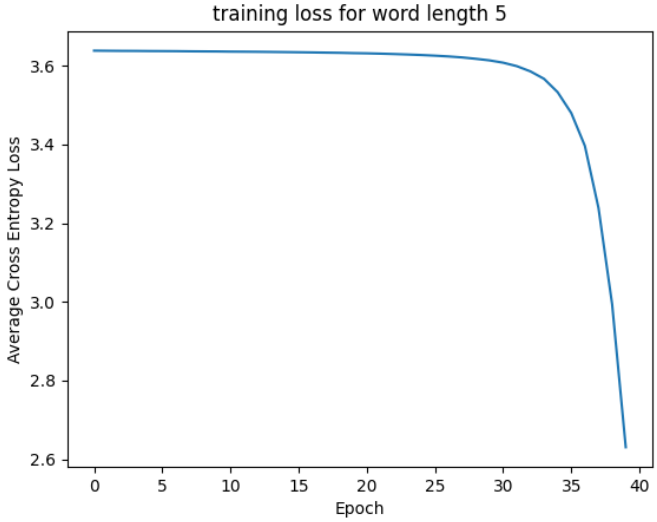
A lot of our challenges in constructing the CNN model came from the question of **how can we optimize our model parameters to increase testing accuracy? What parameters are important when it comes to the optimizer for achieving fast and accurate learning?** As

described in the AI Core section, our initial model used a flat learning rate for all word lengths, did not have a decay rate, and did not employ momentum in gradient descent.

Our first draft achieved a flat 6.4% accuracy as it did not learn and simply guessed one label for all inputs. As described above, this is equivalent to guessing randomly. We were very unsatisfied with these results, and wanted to get to the core of why the model was unable to learn.

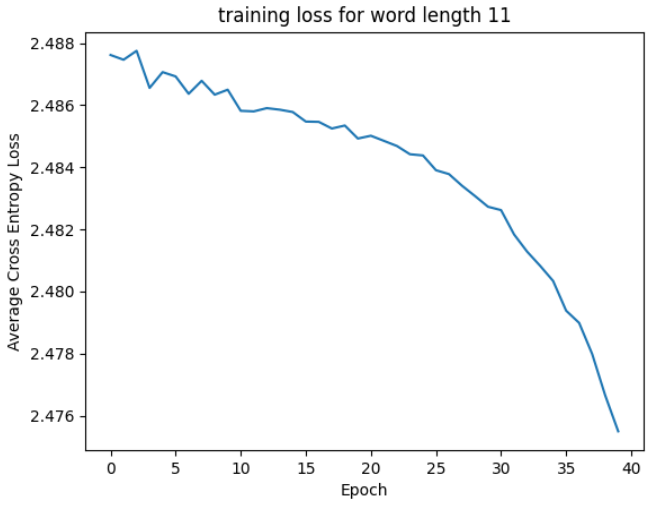


To further the learning objective, our primary goal was to have the training loss decrease consistently over epochs. With guidance from our project mentor Jonathan, the techniques we tried to achieve learning included: downsampling images using gaussian blurs, training for more epochs, trying different base learning rates, using different numbers of channels, employing variable learning rates and rate decay, and momentum.



The initial changes of simply using gaussian blurs, changing the channels, and training for more epochs allowed the model to start learning. However, it still performed poorly with sub-10% final testing accuracy.

The further modifications of optimizing base learning rate, adding learning rate variability based on number of classes, rate decay, and momentum allowed us to achieve a **28.15%** testing accuracy. It also solved some of the issues with overshoot, as seen in the training loss curves to the left.



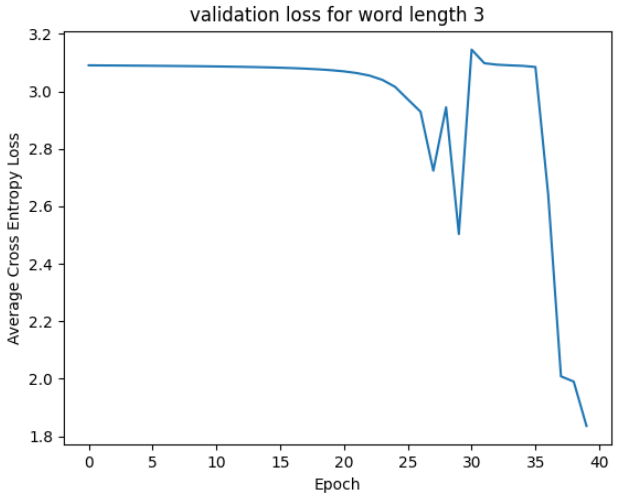
During this particular training instance, we only trained for 40 epochs. Looking at the training loss curves, it seemed to us that the models had not converged yet and 40 was too few epochs.

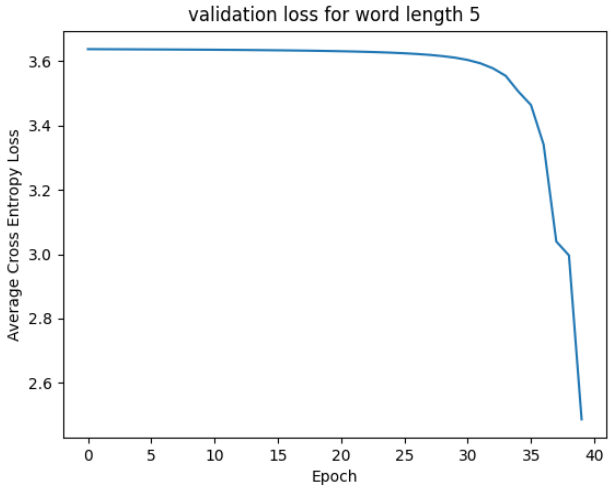
Noticing this led us to another optimization question: **how long should**

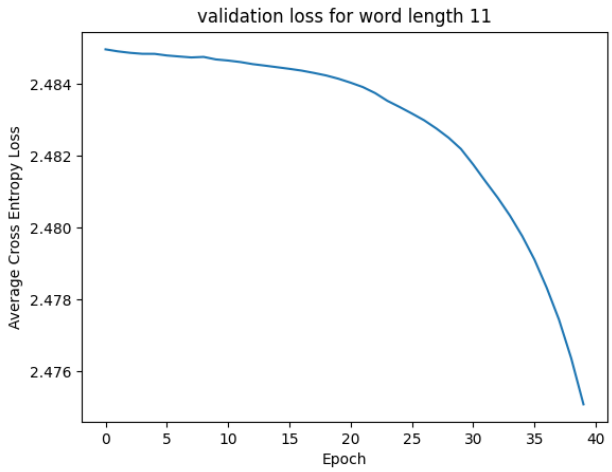
the model train for to maximize accuracy and learning and minimize overfitting?

To tackle this question, we analyzed curves of the validation loss on a set of 2000 validation images each epoch. We had this implemented before training the above model, so these are the validation curves that go alongside the loss curves depicted above. Note that the final training/validation/testing split we used in the model was 16000/2000/2000, and the accuracy of 28.15% was on the testing data of size 2000.

Below are the associated validation curves:



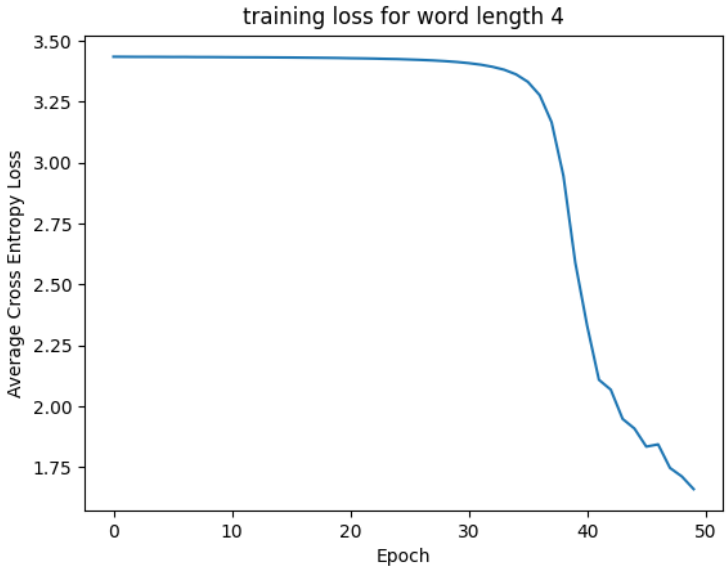


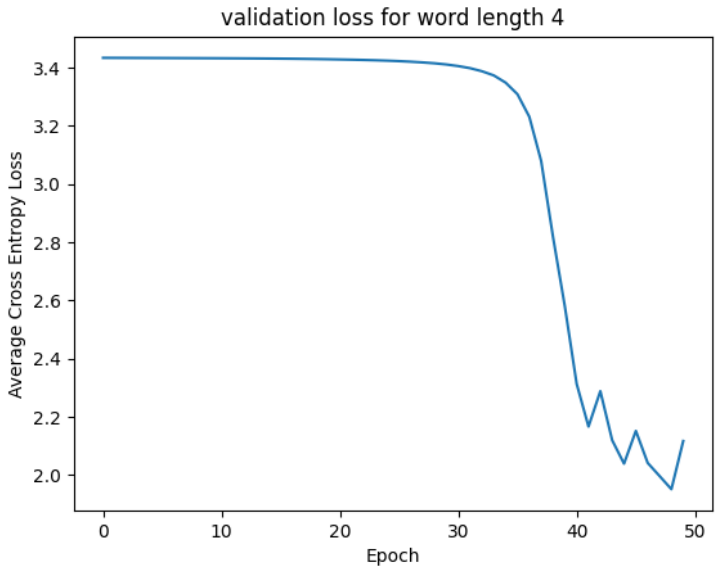


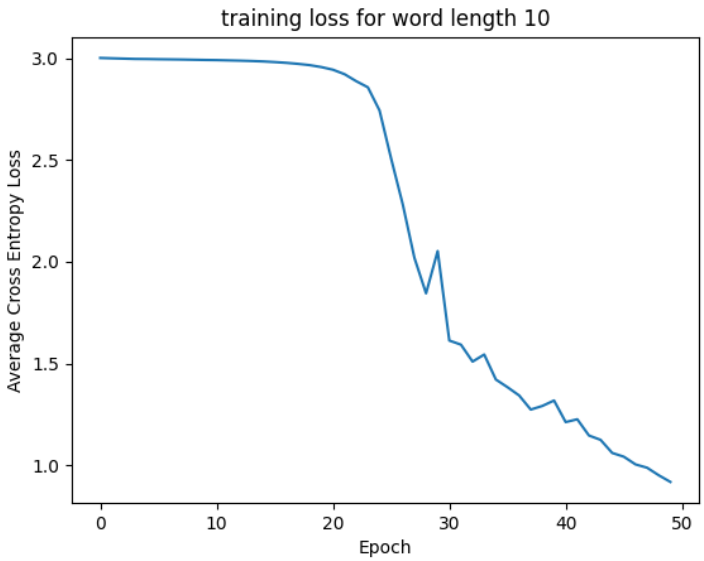
Even though we are not including the loss curves for all 16 lengths in this report, similar patterns were observed across the board. What these loss curves indicated to us was that training to 40 epochs was not yet overfitting.

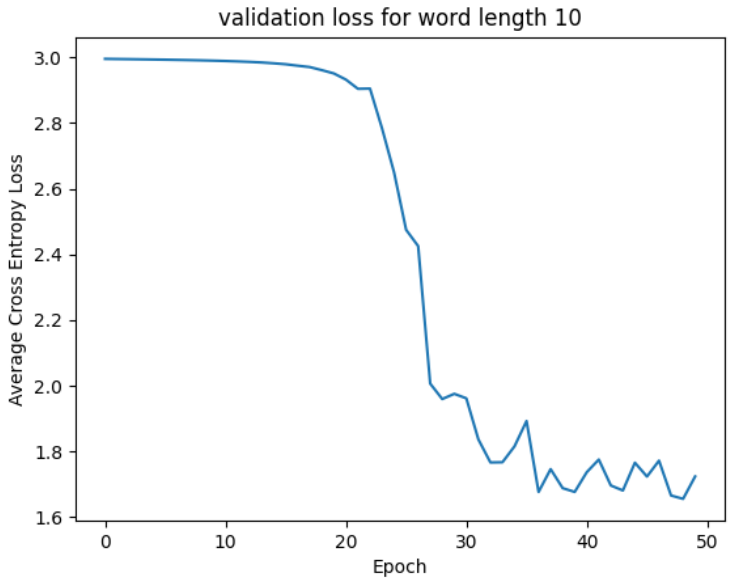
Notably, just as there is still significant room for improvement and a clear downward trend in the training curves, the same is visible in the validation curves.

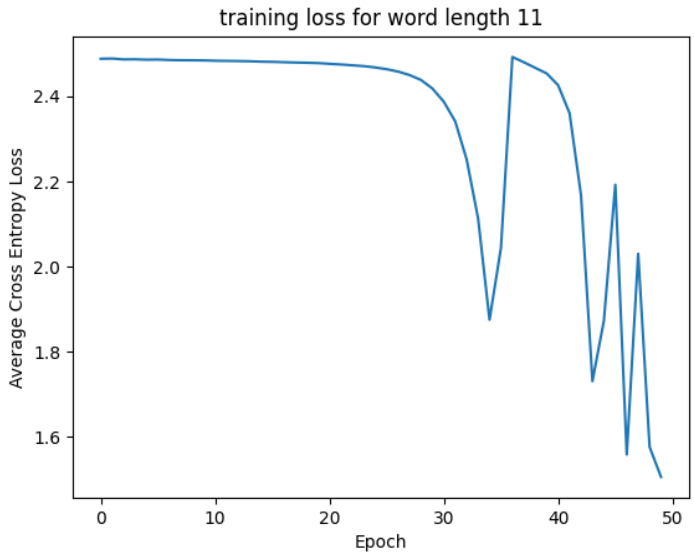
Upon seeing these curves, we decided to retrain the model for 50 epochs, as it seemed that we had not yet hit the point where the model stops learning and starts overfitting. After doing this, our final testing accuracy was **34.6%**. Below are shown a few training and validation curves:

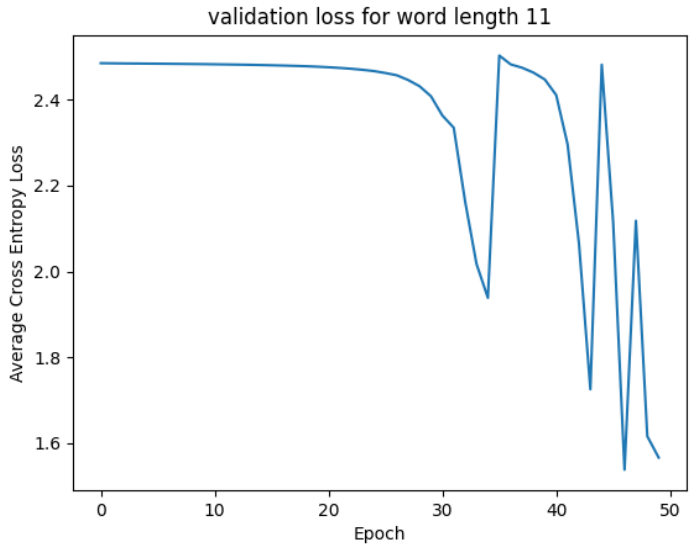

training loss for word length 4
Average Cross Entropy Loss
Epoch


validation loss for word length 4
Average Cross Entropy Loss
Epoch


training loss for word length 10
Average Cross Entropy Loss
Epoch


validation loss for word length 10
Average Cross Entropy Loss
Epoch


training loss for word length 11
Average Cross Entropy Loss
Epoch


validation loss for word length 11
Average Cross Entropy Loss
Epoch

For the sake of brevity, we do not include all training and validation plots in this report. However, you can view them yourself by running `plotLoss(i, "training")` or `plotLoss(i, "validation")` in CNN.py, where i is the desired word length in the range of 3 to 17. Note that there is only a single word associated with lengths 2 and 18.

The general trends we observed from these results were:

- Models for word lengths 3, 5, 6, 7 learned slowly and would have benefited from more epochs.
- Models for lengths 4, 10 learned very effectively.
- Models for lengths 9, 11, 12 still experienced the "spikes" or "overshoots" described previously.
- Models for length 8, 13, 14, 15 began to overfit and would have benefitted from fewer epochs.

Overall, we found that using 50 epochs provided a reasonable balance between optimizing the number of models that learn and minimizing the number of models that overfit. Ideally, we would have trained and optimized the parameters for each word length model individually, but unfortunately we did not get to this goal.

Ultimately, we were satisfied with an accuracy of 34.6% as it met our goals of 30 - 40%+, although it is clear that it is possible to achieve a higher accuracy if we were able to optimize individual models more effectively. We felt that this level of accuracy was a significant improvement over the initial model accuracy of 6.4%, and the use of iterative development allowed us to clearly distinguish what adjustments had been important in allowing the convolutional neural network to learn. Further, we saw from the learning curves that significant further improvement would require more focus on individual optimization of the different models, which was an important learning point for us for future study.

Evaluation Overview

Through researching and exploring our bolded questions, we found that:

- A Random Forest model can still learn effectively on simple drawings, provided they occupy a relatively similar space on the canvas.
- To take advantage of the assumptions of the Skribbl.io setting, we utilized knowledge of the images being sparse (in Random Forest) and knowledge of word length (in both).
- Although Random Forest can learn, CNN still exceeds it in terms of performance on sketches.
- Careful data and image processing is especially important in model design; it is not sufficient to simply downsize images as this can lead to information loss.
- Momentum and Learning Rate Decay are especially important parameters for CNN in a setting where information is sparse, training samples are limited, and the number of classes is large.
- Around 50 epochs provided a reasonable balance between maximizing learning and minimizing overfitting.

- Even in this setting where the only variation between models was due to word length (and the associated classes), optimizing models individually would have been necessary to further improve learning.

We believe that we were successful in answering our questions about the feasibility of Random Forest and about effectively optimizing the convolutional neural network. The only question we asked that requires further follow-up for a satisfactory answer is the question: **how can we optimize our CNN model parameters to increase testing accuracy?** We successfully answered this question in the setting where all 16 models are trained with the same script, where we found that 50 epochs and a base learning rate of 0.01 were ideal. We additionally found that parameters such as momentum, learning rate decay, and learning rate variability (to account for number of classes) were vital to allow the models to learn quickly and more consistently. However, because of the level of variability remaining between the way the individual models learned (described above following the final loss curves), we believe it would be necessary to investigate this question on a model-by-model basis in future training, and this could lead to further improvements. While we were able to achieve our desired accuracy of at least 30% by simply attempting to optimize all models with a single script, optimizing them individually could significantly improve this figure.

Further, we were able to successfully investigate the feasibility of Random Forest in this setting. With a testing accuracy of approximately ~22% compared to CNN's ~35%, it is clear that CNN had a significant lead on Random Forest, and is more suitable even for information-sparse images. However, we have seen that even though images are not a traditional application of Random Forest, it can still successfully learn in this setting and achieve an accuracy better than guessing (which would have been 6.4%).

Overall, we feel that we were successful in constructing a model for the Skribbl.io setting that achieved decent accuracy on completed images. Although we were unable to reach some of our stretch goals and there is still room for further improvement in the model, we learned a significant amount about CNN model design and parameter selection. The level of iterative development especially highlighted which aspects of the CNN's design were crucial in finally allowing it to learn.

References

How do Humans Sketch Objects?

http://cybertron.cg.tu-berlin.de/eitz/projects/classifysketch/

Skribbl.io Base Words

https://skribbliohints.github.io/

Feature Extraction Using Convolution

http://ufldl.stanford.edu/tutorial/supervised/FeatureExtractionUsingConvolution/

Finetuning Torchvision Models

https://pytorch.org/tutorials/beginner/finetuning_torchvision_models_tutorial.html

Downsampling images using Gaussian pyramids

https://docs.opencv.org/4.x/