Super Hexagon Bot

Kevin Baichoo (kbaichoo),[1] Peter Do (peterhdo)[1]

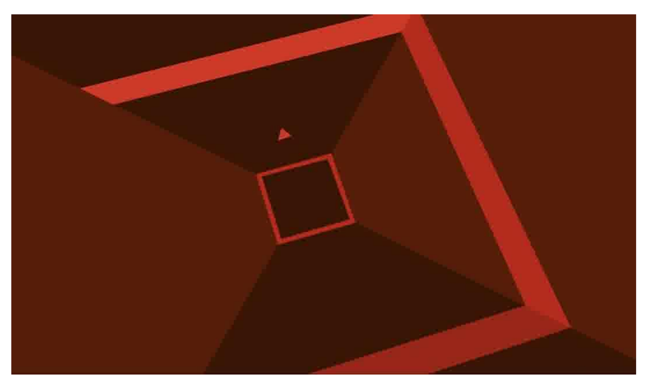
[1]Department of Computer Science, Stanford University

Stanford
Computer Science

Overview

- **Problem**: How can we use neural networks that classify images in order to play Super Hexagon well?
- **Solution**: We present two different models (one based on **MNIST** and one utilizing **ResNet**) and a bootstrapping framework that allows us to play Super Hexagon with neural networks in real-time.

Predictions Based on Screenshot Data



"Left" Classification

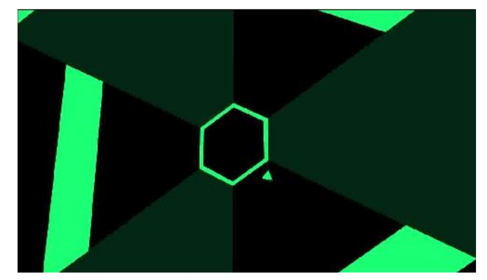
"Right" Classification

"Stay" Classification

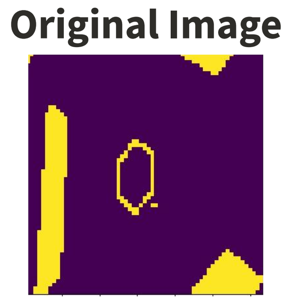
- We modeled the task as an image classification problem
  - Models are implemented in Python using PyTorch
- 3 classifications: move "left", move "right", and "stay"
  - These commands allow the player (triangle) to avoid the wall obstacles to proceed further in the game
- Game loop runs with python script, taking screenshots, passing them into the model, and pressing keys based on the model's output

- **Data Collection**: We used a combination of self-recorded and YouTube video data that we fed into FFMPEG for screenshots.
- With these screenshots, we manually classified training, validation, and test data sets, being sure to balance our data
  - Representative dataset skewed towards staying
  - We were able to augment the data size (by 4x) by rotating the image 90 degrees (Rotation Symmetry)
  - 1K+ Data points

Features



Original Image



Transformed Image

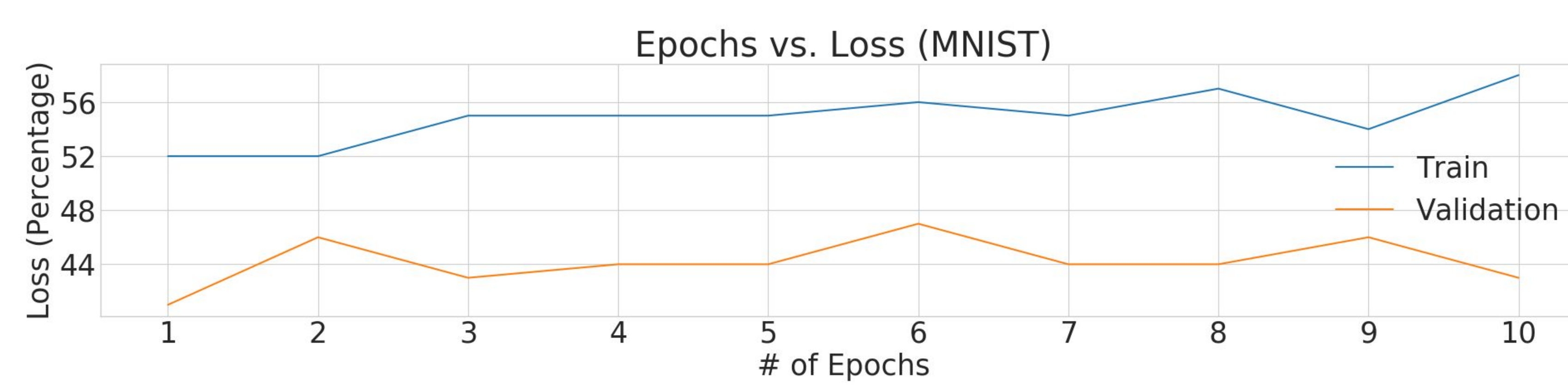
- We transformed the images to leave only the minimal set of relevant features in the image
  - Knowing where the walls / player is enough to make good decisions
- Transformations:
  - Grayscale (single channel)
  - Resizing (downsampling to 64x64)
  - Binarization

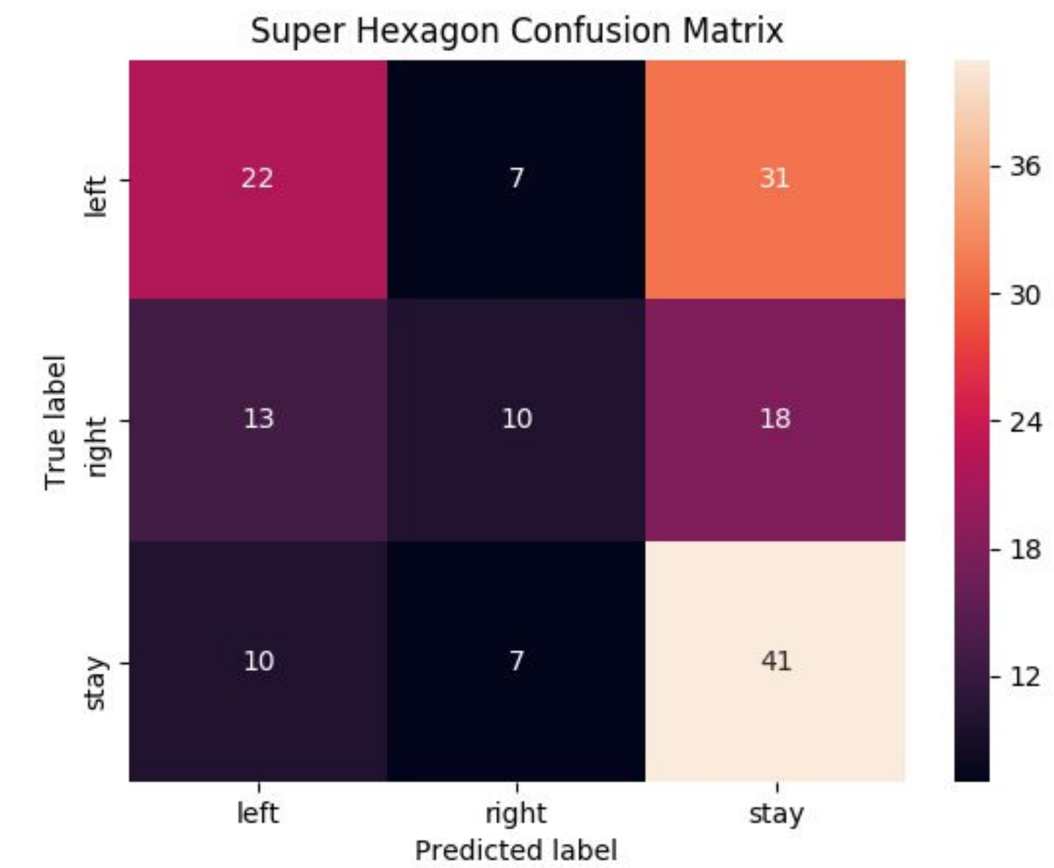
Models

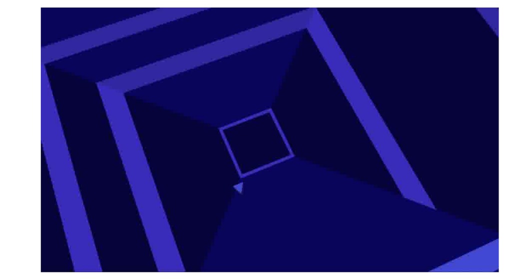
1. MNIST-based classifier
   - 2 conv. layers each followed by max pool, then hooked up to two linear layers; ReLu activations; Softmax on final output
   - SGD optimizer with NLL Loss function
   - Quicker to train, with decent results (around 50% accuracy across validation/test)
2. ResNet
   - Utilized PyTorch's implementation of ResNet (18 and 50 layers) without pretraining (tried pretrained model)
   - ResNet was able to fit the training data well with Adam optimizer and Cross Entropy Loss
   - Overfitting was a concern, mitigated with early stopping

Results

Model	Train	Validation	Test
Modified MNIST	63%	41%	48%
ResNet50	59%	40%	45%







Original Image

Saliency Map

Discussion

- There is a significant amount of bias towards predicting "stay" (as seen in the confusion matrix)
  - High error rate on "left" and "right" classification
  - However,"left"/"right" can both be valid
- Overall, our networks do seem able to determine which features are relevant in classification and play the game in real-time with some success
- Our MNIST-based model likely needs more layers to be more expressive.
- Nontrivial to tune ResNet for our purposes without overfitting.

Future

- We plan to refine the models (via hyperparameter tuning, data augmentation, data regularization, transformations, etc.) in order to improve classification accuracy.
- We may experiment with other types of networks (RNNs) or other techniques (RL) to compare results

References

1. **Deep Residual Learning for Image Recognition**: https://arxiv.org/pdf/1512.03385.pdf
2. **Densely Connected Convolutional Networks**: https://arxiv.org/pdf/1608.06993.pdf
3. **MNIST Classifier**: https://github.com/pytorch/examples/tree/master/mnist
4. **Generating Saliency Maps**: https://github.com/utkuozbulak/pytorch-cnn-visualizations