HW4

Reinforcement

+

Deep Learning

Name: Jordan Wu Section R3 CRN: 69112
Shih-Jie Chang Section Q3 CRN:31425
date:4/21/2019

1.1

1.1.1

During training, the agent first updates the Q table using the SARSA algorithm given current state (which is passed into the act function and then discretized into the smaller state space) and the previous state+action (stored in self.s and self.a). We call the previous state and action s and a, and the current state s_prime. The reward is calculated by first checking if the last action caused the agent to die, which is given by the dead argument. Then checking if the agent ate a food by checking if points argument increased from the previous. If nothing happened, we give a penalty of -0.1.

To find the next action the agent should take, take the argmax of the exploration function given all actions and the current state s_prime. As required by the MP, tiebreaking is done in reverse with respect to the action index, from right, left, down, up.

In the final step, we check if the snake is dead, and reset the game if it is. Only if the snake is alive, we increase the N for the action we just calculated. We also save the current state, action, and points so it can be used in the next iteration, given the snake is still alive.

1.1.2

During testing, we only do step 2, which is calculating the next action. In addition, we also ignore the exploration function, and instead just return the best action based on the Q value for the current state. We keep the tie-breaking priority.

1.2

1.2.1

To optimize Ne C and gamma, I created an wrapper script that iterated through various hyperparameter values, trained for 10000 episodes, then took the 1000 episode test average. For Ne and C, I iterated 10, 20, 30, … up to 100. For gamma, I iterated 0.1, 0.2, … up to .9.

However, when we compared the performance of these hyperparameters with the default's performance at 20000+ train_episodes, there's no real difference.

Best hyperparameters for 10000 train_episodes
Ne = 20
C = 80
Gamma = 0.5

1.2.2

train_episodes = 10000
Training takes 20.178129196166992 seconds

1.2.3

Test_episodes = 1000
Average Points = 21.995

1.3

In the modified MDP, we did away with the first 2 state parameters regarding where the wall is in relation to the head. Instead we encoded this information directly in the last 4 parameters, which checks for any “objects”, wall or body, in the four directions relative to the head.

As such, the new state model is a 6-tuple vs the 8-tuple of the default.
(food_x, food_y, object_top, object_bottom, object_left, object_right)

In addition, we also modified the reward model to give a penalty of -10 for deaths. This made the snake much more “risk averse”.

This modified MDP significantly boosted the “rate of learning”. By the 10000 train episode, we can get an average above 20 points consistently. This increase can be attributed to the reduced state space the agent needs to learn.

Results:
Hyperparameters:
Train_episode = 10000
Test_episodes = 1000
Gamma(default) = 0.7
Ne(default) = 40
C(default) = 40

Average points (test) = 24.741

2:

2.1

For the affine_forward function, I decided to concatenate the biases to the weights as an extra row and add an 1 column to A. Doing so allows calculation of Z in one numpy matrix multiplication operation.

For relu_forward and backward, I tried various forms, such as with fancy indexing, np.maximum, but I found np.where to be the fastest since it doesn't need to do copying.

The rest of the functions, I used numpy function for all the operations as much as possible.

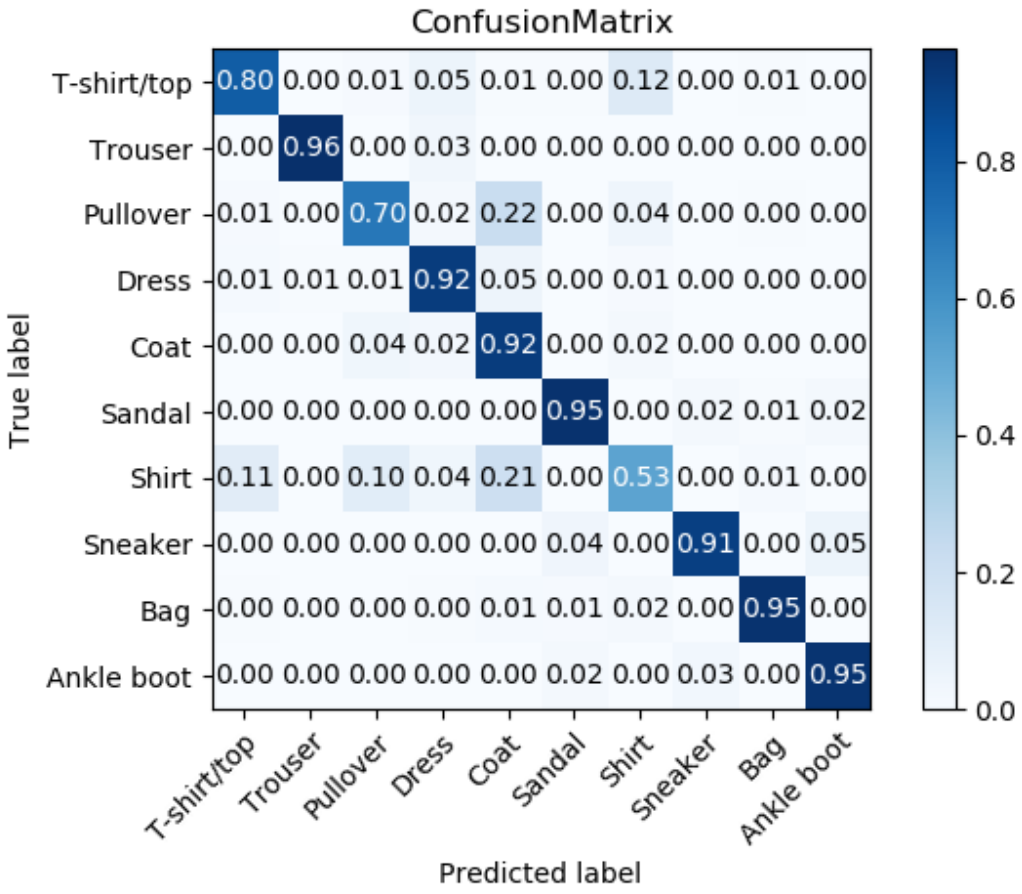
The final result was about 2.5 seconds per epoch, which is pretty good.

2.2

Epochs = 10

Avg Classification Rate: 0.857

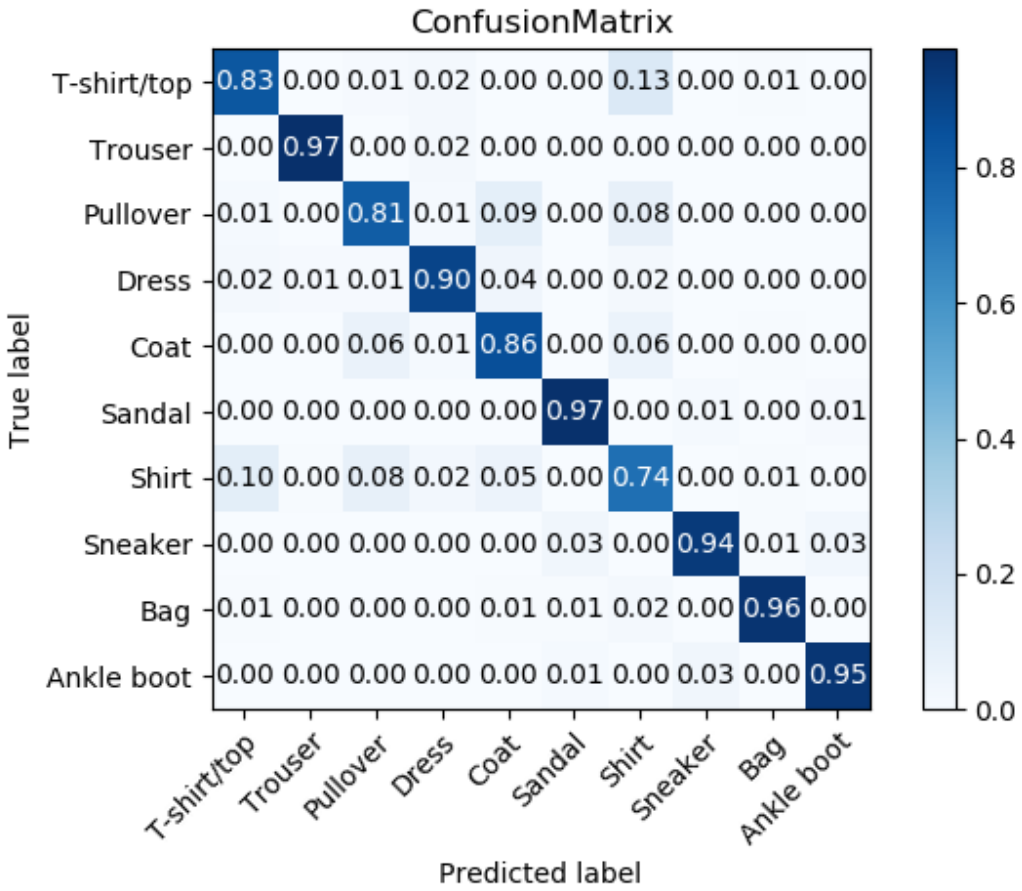
Runtime (Core i7 8700): 25.599



Epochs = 30

Avg Classification Rate: 0.8911

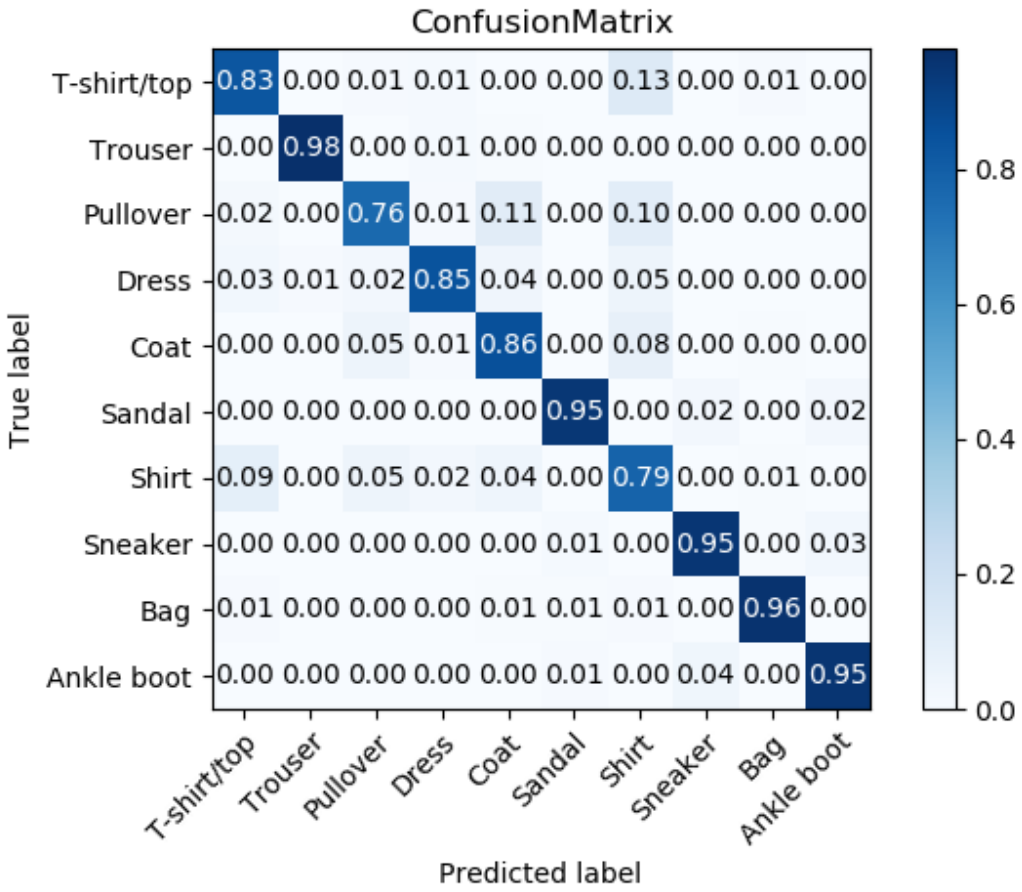
Runtime (Core i7 8700): 77.739s



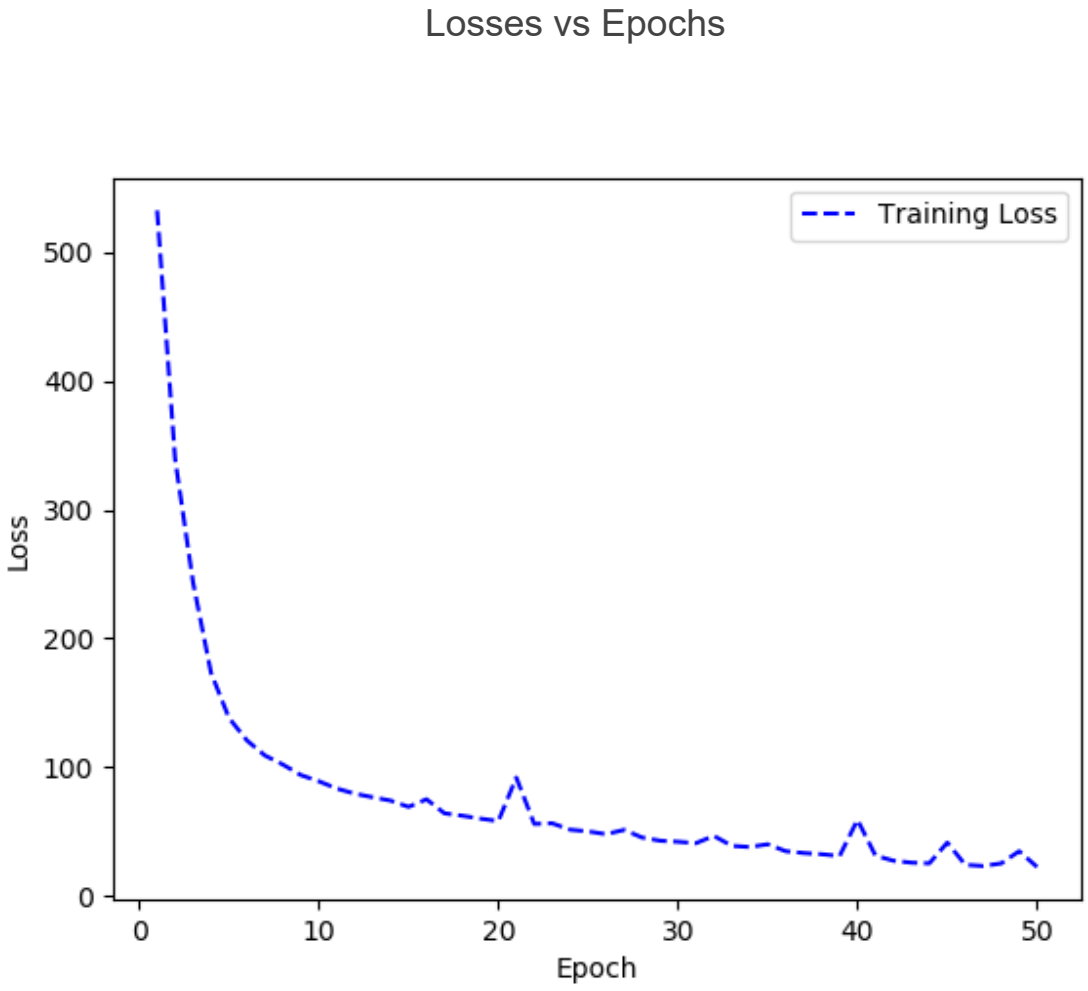
Epochs = 50

Avg Classification Rate: 0.8882

Runtime (Core i7 8700): 127.168s



2.3



2.4

We can see the losses decreases very rapidly, and by the 10th epoch it steadys out.
We can also see small spikes in losses throughout, but more pronounced at the 20th and 40th epoch.
Spikes in losses are often unavoidable when using mini-batch gradient descent, since updates of weights in one batch can cause the losses for another batch to spike.

2.5 (Extra Credit)

We used Pytorch implemented a CNN model inspired bo LeNet-5 and the model described in lecture.

For implementation, I followed these tutorials:
https://seba-1511.github.io/tutorials/beginner/blitz/neural_networks_tutorial.html
https://www.youtube.com/watch?v=LgFNRIFxuUo
https://blog.algorithmia.com/convolutional-neural-nets-in-pytorch/

A CNN works by preprocessing the input image through layers of convolution filters to create output feature maps. These output feature are then fed into a classic deep neural network for classification. The advantage of this preprocessing by convolution is that the output feature maps "summarizes" spatial relationship in the input data, making it much more meaningful when fed into the DNN. For example, some of the convolution filters can be doing edge detection, while others can be doing some blurring to remove noise. When compared to traditional image preprocessing approach, the trainability of the convolution filters allow for the net to "find" the optimal filters by itself.

Our CNN implements 2 layers of "convolution", where each layer involves

1) Convolution, we chose 5x5 kernels for both layers to follow the LeNet-5 model, plus it gave lower losses than 3x3 kernels.
2) Max-Pooling, which increases generality and reduces overfitting, while also reducing the output feature map size.
3) ReLU non-linearity. For the same purpose as in DNNs. Convolution is a linear operation, so it is associative. Without non-linearity, multiple layers can be simplified to a linear classifier.
4) Normalization, more generally known to reduce internal covariate shift, which causes exploding gradients and overfitting.

The output feature map of the last convolution layer are fed into a 2 layer DNN with identical architecture to part 2's DNN.

We used cross entropy for the loss function, and optimized using mini-batch gradient descent to for comparison to the DNN we implemented previously.

During training, the CNN follows similar steps as what we implemented. A batch is fed forward into the net, and outputs are calculated. The CrossEntropy class automatically calculates the loss based on predicted and actual label. Pytorch's built in variable type stores the cache values, and by calling loss.backward(), it calculates the gradients for each variable in the net. The selected optimizer then applies the gradient by calling optimizer.step()

The CNN was comparable to the DNN we implemented.
The CNN was much slower. Each epoch took about 24 seconds. Since pytorch supported CUDA, I ran the net on GPU instead, and each epoch took less than 2 seconds.

Results

Hyperparameters:

Epochs: 20

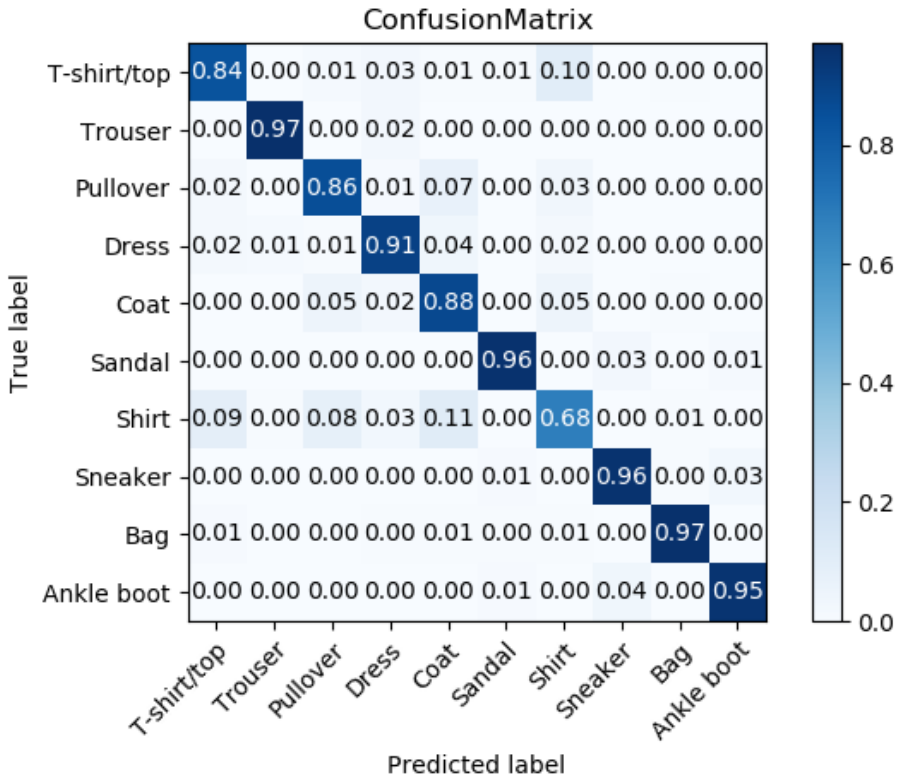
Batch Size: 200

Learn Rate: 0.01

Average Classification Rate: 0.8968

Training Time (GPU): 39.42998027801514

Rate Per Class: [0.853, 0.978, 0.829, 0.905, 0.795, 0.962, 0.765, 0.964, 0.965, 0.952]



Loss vs Epoch