Introduction to Artificial Intelligence
198:440
Project 3

Larry Scanniello

July 2024

1 Input and Output, Dataset

Each data point in my training dataset represented some information known to the training bot (bot 3 of project 2), such as the probability distribution or current location, at a snapshot in time during the simulation. The corresponding targets for each each data point were a number between 0 and 4, each representing either a move of up, down, left, right, or sense that resulted from the information represented by the data point. I found that the structure of the input data plays a very large role in whether or not a model succeeds. Here, I will detail my attempts at structuring/preprossessing the input data.

My first attempt at structuring the input data was to have the input data be a large vector. The first two components were the index the bot is currently at, the next two components were the index the bot is previously at, the next 1600 components were the flattened layout, and the next 1600 components were the flattened probability distribution.

My second attempt was to condense everything onto one 1600-length flat vector. I had the current location as 10000, the previous location as 7500, the probabilities in their corresponding index multiplied by 1600, and closed squares as -1.

I then tried to have the input data be one (small) vector, where the first two components were the current index, the next two components were the previous index, and the next twenty indices were, in order, the top ten indices in the probability distribution with the highest probability.

I then tried to have the input data be the current index, previous index, and highest probability index.

All of these attempts at structuring the input data were unsuccessful. In hindsight, I was simultaneously making the mistake of over-complicating my models, which definitely worked against me, and if I tried some of these input configurations again with simpler models, it's possible that I'd get better results. But what I learned is that it is best to structure the input data in the most obvious way for the computer to understand, so that the computer has to do as little heavy lifting as possible to figure out the relationships in the data.

I was ultimately able to make better models by following the advice of the professor, and having my input data be stacked arrays. I tried different variations, but I landed on having the first array be the ship's layout, the second be the zeros except for a 1 in the bot's current location and .5 in the previous location (if applicable), the third be all zeros except for a 1 in the space with the highest probability, and the fourth be the current probability distribution times 1000 (to keep things nicer numerically). One variation I tried was to replace the probability distribution for each data point with the probability distribution from the most recent time the training bot did UFCS, since it was really this distribution that the bot was using

```
class StationaryMouseNetwork(nn.Module):
  def __init__(self):
    super(StationaryMouseNetwork, self).__init__()
    self.conv_1 = torch.nn.Conv3d( in_channels = 1, out_channels = 40, stride = 1 , bias = True, kernel_size=(1,40,40))
    self.linear = torch.nn.Linear( in_features= 40*4, out_features= 200, bias = True)
    self.linear_2 = torch.nn.Linear( in_features = 200, out_features = 5, bias = True )

  def forward(self, input_tensor):
    reshaped = torch.reshape( input_tensor, (-1,1,4,40,40) )
    conv_results = self.conv_1( reshaped )
    conv_results = torch.nn.Tanh()( conv_results )
    flattened = torch.nn.Flatten()( conv_results )
    prelogits = self.linear( flattened )
    prelogits = torch.nn.ELU()(prelogits)
    logits = self.linear_2( prelogits )
    probabilities = nn.Softmax( dim = 1 )( logits )
    return logits, probabilities
```

Figure 1: StationaryMouseNetwork (and basically StochMouseNetwork)

to plan its moves, and although I got decent results with this, I decided against it for training my final models, since especially for the stationary mouse case, this may confuse the model.

I was conflicted about including the array with highest probability index. My bot 3 from project 2 (the training bot) makes decisions solely based on the highest probability square at the time. I wasn't sure if including this array was innocuous prepossessing, or if it was making a full decision/setting a priority on behalf of my ML model bot, when the bot should be learning what to do from the probability distribution alone. Since the results were so much better, and the highest probability index is information readily availability to the neural network bot, I decided to include it in my array stack.

I gathered data by running about 1500 simulations for each the stationary and stochastic case, generating about 300k to 400k data points, one data point for each time step in each of the simulations. This generated many gigabytes of data, but also in some ways, may not have been enough, since for example, a neural bot only at the start of a simulation only has 1500 references for how to start. It might be advantageous to generate data specifically at the start of simulations, or at the end of simulations, or in certain scenarios it is important for the neural bot to learn.

For output, all of the models I made have a softmax as their last layer, which output a 5D vector, each component interpretable as probability corresponding to either up, right, down, left, or sense. This gives the neural network bot some more flexibility, because if it can't move in a direction, it can pick the direction with the next highest probability.

I also tried letting the output include the directions northeast, southeast, southwest, and northwest. I thought I could get better classifications since the bot is often moving in two directions, not just one. To get the training data, I used the destination index from my training bot's UFCS to determine the direction. I couldn't get this to work, but I may give it another try later.

In this project, I fixed $\alpha = \frac{-\ln(0.5)}{25-1} \approx .028881133$.

2 Models

Both the stationary and stochastic models had similar structure, with one 3d convolutional layer and two linear layers into a softmax. The key ingredient to the success of these models was using a $1 \times 40 \times 40$ kernel on the convolutional layer. This mean that each array in the stack was examined separately. The most important local feature of the data is that the different arrays in the stack were their own arrays, and

using the convolutional neural network in this way accounted for that. The sizes of each layer can be seen in figure 1. The only real difference between the stationary and stochastic network is that the stochastic neural net has more out features (300) in the first linear layer, to add some complexity. I genuinely found that a setup like this worked best for both cases. I was more adventurous with my neural network for randomly generated layouts.

There was a lot of trial and error that went into determining these models. I observed (no data, just my experience) a significant negative correlation between the overall complexity of the model and ease of training the model. Whenever I tried to make a larger model with five layers, each with 1000+ features, or a convolutional layer with small kernel and stride and many out channels, it became impossible to get the loss to go down. I think if I wanted a complicated model like this to work, I would need to use a smaller step size on the optimizer, or more strategically initialize the weights (as the professor mentioned in class).

On the other hand, I did not really observe much correlation between model success and other design choices. The first activation function used was tanh, and the second activation function was ELU. I like tanh, and the ELU because it is a smoother version of the ReLU, but a lot of my intuition here is based on just vibes. One reason I used the ELU last because it can preserve more input structure than tanh or sigmoid (although it's not clear how important this preservation of input structure is). I had roughly the same success interchanging them, or using sigmoid, or other activation functions. I think at least for my data, the activation functions seemed to not make as big of an impact as, say, the overall model complexity. I am sure if I worked more with ML, differences would become apparent, but I not notice an especially pronounced difference between activation functions.

3 Training Process

I primarily used the Adam optimizer. I heard it was the gold standard in optimization, and it generally lived up to this and worked well for me. Sometimes, when not getting the results I had hoped for, I would try stochastic gradient descent, but this did not yield any remarkable results, doing at best as well as Adam, or often worse than Adam.

I think there was a relationship between the scale of my data and the step size I could use without error, but I am not totally sure. At first, when I tried using the .01 step size, when the values of the arrays in my data were not scaled roughly between 0 and 1, the loss would eventually become nan. I switched to a .001 step size when this happened, which seemed to fix the problem. When I switched to roughly scaling the array values between 0 and 1, I was able to switch the step size back to .01. There seemed to be some correlation between the scale and the step size I could use, but more investigation is needed to determine if that's really the case.

I experimented with regularization a couple of times. I tried using both L_2 regularization and dropout. It didn't seem like overfitting was too big of a problem, especially in the stationary fixed layout cases. So I did not feel a lot of pressure to get this exactly right. For L_2 regularization, I never mastered the art of finding the correct value of λ, the constant coefficient. If λ was too large, then the weights would completely take over the loss, and the model would break. If λ was too small, no regularization would happen. It was never clear if I hit the sweet spot of λ such that the weights did not completely dominate the loss, but contributed just enough to make the model better. Dropout didn't really work much better, making it much harder to train the same model it was applied on, maybe not working so great with localized data/convolutional layers.

4 Results

My results for StationaryMouseNetwork and StochMouseNetwork can be seen in figures 2 and 3.

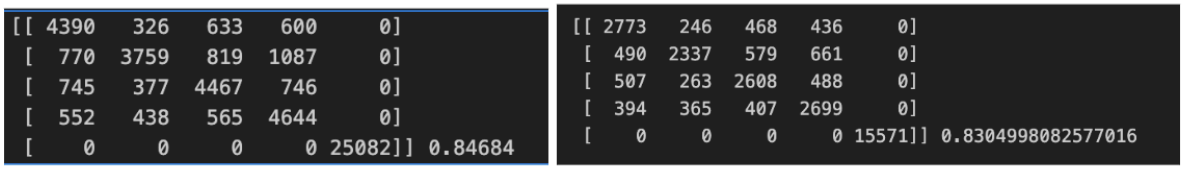

Figure 2: StationaryMouseNetwork, results. L: Training, R: Testing, Average Loss per Data Point: 0.3857

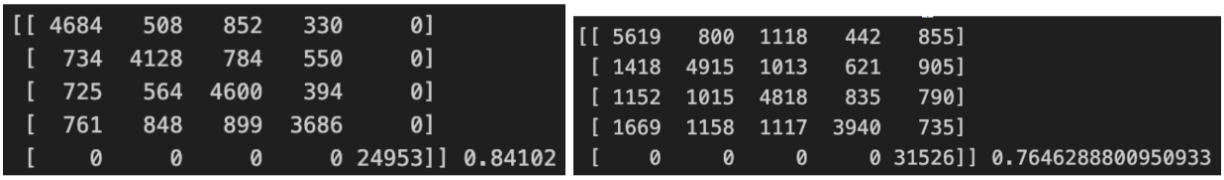

Figure 3: StochMouseNetwork, results. L: Training, R: Testing, Average Loss per Data Point: 0.3945

For both sets of training data, I was able to achieve greater than 80% accuracy on the training data, as well as 83% accuracy on the testing data for the stationary case, and 76% accuracy on the testing data for the stochastic case.

I attribute my modest success to the use of the convolutional layer, which looked at each array in the stack separately. No other change I made to the model or training seemed to have had as big of an impact, other than maybe adding an array to the input data that gave the highest probability index.

I am curious about how to get higher accuracy and lower loss. For lower loss, I don't think the answer is more data. Since the training bot can move in more than one direction at the same time, I don't think it's possible to get perfect accuracy with my current 5-dimensional output. Intuitively, in less dimensions, you shouldn't expect or even want to always be able to fit a line perfectly when doing linear regression, and the case of neural nets is analogous.

Letting the neural network bots do their thing without any outside intervention led to some pretty disappointing results. I defined bot failure to be if the bot took any longer than 5000 moves, and I ended simulations at $t = 5000$. For both bots the success rate was not much higher than 1/3. I wanted to see if the bots would do any better if they were given a little bit of help, so I conducted trials with backtrack prevention, which means the bot is not allowed to move into the space it was previously in, or the space before that, and the bot cannot sense more than once in a row. In figures 4 and 5, summary data for the neural network bots are displayed. All of the data for the neural bots are calculated using data from successful trials only.

No backtrack protection	Success ratio	Mean	Median	Max	Std
Bot 3	1	228.4	222	556	98.33171776
StationaryMouseNetwork bot	0.384	511.1518325	367	4946	635.3408262
With backtrack protection	Success ratio	Mean	Median	Max	Std
Bot 3	1	218.26	208	624	95.95755071
StationaryMouseNetwork bot	0.854	431.6299766	294	4980	599.9892814

Figure 4: Statistics from StationaryMouseNetwork bot, 500 trials, successful trials only (except success ratio)

No backtrack protection	Success ratio	Mean	Median	Max	Std
Bot 3	1	219.336	216	542	94.34804615
StochMouseNetwork bot	0.356	665.2977528	475	4279	650.6283743
With backtrack protection	Success ratio	Mean	Median	Max	Std
Bot 3	1	334.666	277	1614	244.0638384
StochMouseNetwork bot	1	413.966	317	1995	328.0339196

Figure 5: Statistics from StochMouseNetwork bot, 500 trials, successful trials only (except success ratio)

What is surprising is that it seems as though a neural bot without backtrack protection either fails, or it doesn't. It is not the case that we uniformly have simulations that have times to completion between 0 and 5000. For the stationary neural bot with no backtracking, there were only 3 trials that took between 2000 and 5000 time steps. For the stochastic case, there were 8. If you look at a heat map for these bots, what you see is that they get stuck, oscillating between two squares back and forth forever.

The problem with the neural bots with no backtracking is that they are short-sighted. They only think one move ahead. This is in contrast to bot 3, which sets a destination, does a UFCS, and operates nine time steps at a time. Without the ability to think ahead, the neural bots get stuck. Although we will see later, I actually was able to (not intentionally) make a neural network bot that gets stuck relatively infrequently without outside help, although it is not the most efficient at catching the mouse.

In the trials where the neural network bots do not get stuck (the summary data displayed in figures 4 and 5 is calculated with these trials), they do not do as well as bot 3, or any of the bots from project 2, but they don't do awfully, taking on average 511 and 665 time steps for the stationary and stochastic case respectively.

With a little help, the neural bots actually do pretty decently. With backtrack protection, they take on average 432 and 413 time steps for the stationary and stochastic cases respectively, which is pretty good.

My bot 3 for project 2 uses UFCS to take paths most likely to yield a mouse. I would like to think to think that my model learned to do this, and learned to set out on paths that are better than just the shortest path to the destination. More testing is needed to see if this is the case, but since using UFCS involves planning multiple moves at a time, and the neural bot can only think 1 move ahead, I am not sure if my neural bot was really able to learn this. I think my model primarily looks at the current location and location with highest probability, and determines its output from that.

One last piece of data that I collected are bumps, which are the number of times the bot tries to move into a space, but can't, because the space is either out of bounds or closed. A bot can bump more than once in one time step, if it tries to go in one direction but can't, and then tries to get in a different direction but can't. A stationary bot without backtracking, on successful trials, bumps an average of 2.25 times per trial, and on unsuccessful trials bumps an average of 40.95 times. However, most unsuccessful trials have zero bumps, with only 31 out of 309 unsuccessful trials having more than zero bumps, and 8 trials having over 1000 bumps. A stationary bot with backtracking protection on successful trials bumps an average of 68.33 times, and on unsuccessful trials bumps an average of 1086.05 times.

A stochastic bot with no backtrack protection has on average 1.04 bumps on successful trials, and 7.88 bumps on unsuccessful trials (again with some outliers above 1000). Each stochastic trial with backtracking was successful, and there were an average of 49.52 bumps.

A bot with backtrack protection bumps much more than a bot without. This suggests that the model itself is doing a good job of outputting valid moves. It's just that these valid moves are not always great for reaching the end goal. With backtrack protection, the moves the bot is forced to make are better for

```
class RandLayoutNetwork(nn.Module):
  def __init__(self):
    super(RandLayoutNetwork, self).__init__()
    self.layoutconv = torch.nn.Conv3d( in_channels = 1, out_channels = 5, stride = (2,1,1) , bias = True, kernel_size=(2,3,3))
    self.botstateconv = torch.nn.Conv2d( in_channels = 1, out_channels = 5, stride = 1 , bias = True, kernel_size=(5,5))
    self.linear = torch.nn.Linear( in_features= 15300, out_features= 500, bias = True)
    self.linear_2 = torch.nn.Linear( in_features = 500, out_features = 5, bias = True )
  def forward(self, input_tensor):
    reshaped = torch.reshape( input_tensor, (-1,1,4,40,40) )
    layoutresult = self.layoutconv(reshaped[:,:,:2])
    highestprobindexresult = reshaped[:,:,2]
    botstateresult = self.botstateconv(reshaped[:,:,3])
    layoutresultflattened = torch.nn.Flatten()(layoutresult)
    destindexflattened = torch.nn.Flatten()(highestprobindexresult)
    botstateflattened = torch.nn.Flatten()(botstateresult)
    flattenedresult = torch.cat([layoutresultflattened,destindexflattened,botstateflattened],dim=1)
    flattenedresulttanh = torch.nn.Tanh()( flattenedresult )
    prelogits = self.linear( flattenedresulttanh )
    prelogits = torch.nn.ELU()(prelogits)
    logits = self.linear_2( prelogits )
    probabilities = nn.Softmax( dim = 1 )( logits )
    return logits, probabilities
```

Figure 6: RandLayoutNetwork

reaching the end goal, but the move with the highest probability that is not moving to a previous space is not always a valid move. I think that bumps are an important measure of the success of a model, since if a model was perfect, it would never try to move into a closed or out of bounds space, and the disallowed moves would have the lowest probability.

5 Bonus: Random Layouts

For each of the previous trials, the layout was fixed. I took on the challenge of training a model that works when the layout is not fixed. I decided I'd collect a large amount of data for this task. I also chose to focus on the stochastic mouse case here.

I decided to be a little bit more adventurous with this model, and try an idea that I had that hopefully is not misguided: Using different convolutional configurations for different arrays in my data. My model first splits up the data, so instead of processing one stack of four arrays, I processed separately a stack of two arrays (ship layout and current bot location), plus one array (highest probability index array), plus one more array (current probability distribution array). Since it was particularly important for the ML model to pick up on the relationship between the current bot position and the layout of the ship, I decided to make a special 3d convolutional layer to use on this stack of two arrays, with kernel size $(2, 3, 3)$. The idea is that at least once, if the current location is not on the border, the kernel will look, in 3d, at the bot's current location and all of its surrounding squares at the same time. I used a different scheme for the probability distribution array, using a 5×5 kernel with stride 1. After convolution, everything is then flattened and sent to the next linear layer. I did not put the highest probability index array through any further convolution; I sent it straight to the next linear layer.

This model is much more complex, with 15300 features coming into the first linear layer, and a result, was harder to train. But I was able to get the loss to go down after a while.

One immediate observation is the success rate of this bot with no backtrack protection is much higher than the stationary and stochastic with no backtrack protection. I am not sure why this is the case. One reason may be that I trained this model on a much larger dataset (about 700k data points) than my other models. But I wonder if there is something about the structure of the model itself that lends itself to being more successful. I think its complexity, and maybe even its lower accuracy might help it not get stuck in

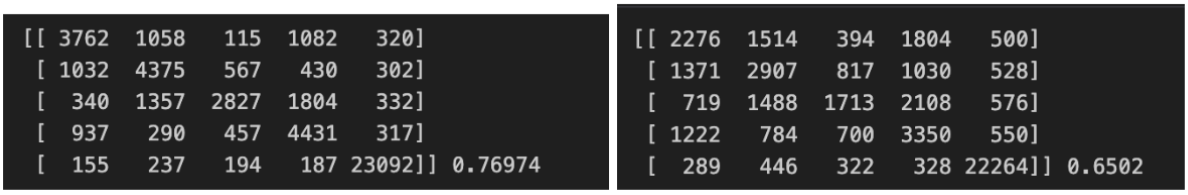

Figure 7: RandLayoutNetwork, results. L: Training, R: Testing, Average Loss per Data Point: .5976

No backtrack protection	Success ratio	Mean	Median	Max	Std
Bot 3	1	327.476	261	1627	258.7790352
RandLayoutNetwork bot	0.962	1020.230769	697.5	4816	979.1318662
With backtrack protection	Success ratio	Mean	Median	Max	Std
Bot 3	1	334.024	259	1724	263.14017
RandLayoutNetwork bot	0.996	690.0863454	484	4910	715.6789431

Figure 8: Statistics from RandLayoutNetwork bot, 500 trials, successful trials only

loops. I think if we used this bot for the fixed layout case, we'd also see higher success rates for the no backtracking cases.

However, the means in general for this bot were higher, and it generally took longer to find the mice, for both the backtracking and no backtracking cases. Where before, in the no backtracking cases, it was rare to see a bot take between 2000 and 5000 time steps (less than 10 trials out of 500), in the 500 trials with no backtrack protection, there were 76 trials that took between 2000 and 5000 time steps. Thus this neural bot, while much less likely to fail completely, often makes the wrong moves, and is not as efficient at catching the mouse. The backtrack protection bot is better, but also suffers from the same inefficiencies. I think there is an accuracy issue here, with the model only achieving .65 accuracy on testing data.

I did try to address the gap between the training accuracy and testing accuracy using L_2 regularization. But the fact that it took so long to train the model made it even harder to find the right value of λ, and I was ever able to get the regularization quite right. I also tried dropout, and even with $p < .5$, the model seemed to refuse to train, although it's possible that with more time I could have seen some better results.

For successful trials with no backtrack protection, the random layout neural bot bumped an average of 318 times. This is considerably higher than the other cases, and to be expected, since we are working with randomly generated layouts. Unlike the other cases, adding backtrack protection gives less bumps, with an average of 244 per trial.

It is the best model so far in terms of success rates, but in terms of bumps, and time to catch the mouse, it could be improved.