Zaid Abdulrehman
Professor Thomason
CSCI 499

# Objectives

In this task, my objective was to create an LSTM pipeline which could classify an action and a target from a provided sentence. The dataset, ALFRED, was written by humans and described an agent on the screen. My task was split into processing the data properly, deciding on a model architecture, and experimenting with hyperparameters.

# Implementation

## Data Processing

Data required for the task was provided in a JSON format. Upon loading the data, it became clear that the data was split into episodes. Sentences were grouped based on the agent they were describing. This added an element of complexity to the task, so in order to avoid complexities in feeding the data to the model, I eliminated this grouping. The data was a collection of sentences, from which a mapping from integers to words was created. The size of the mapping was predetermined in the utils.py. Using this mapping, I tokenized every word in the feature set. Similarly, I tokenized the action and target labels. A large focus during this portion of the task was ensuring compatibility with the data loader. To achieve this, I made sure to use a NumPy array for manipulating the data and looped through the data by index to add padding tokens.

## Model Architecture

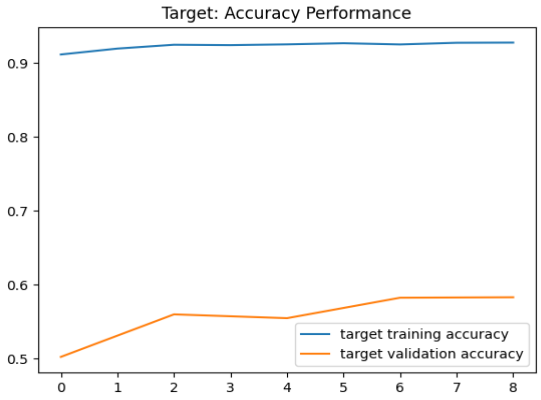
For this task, I tested three main model architectures. I first tested a model architecture consisting of an embedding layer, a max pool layer, an LSTM, and two fully connected layers. I chose this model as it follows from the first text classification model we discussed in class of a max pool layer followed by a fully connected layer. In my testing, the model's performance was very poor, the model achieved an action accuracy of 90% and achieved an unreliable target accuracy of 35%. My initial hypothesis for the poor performance of this model was the inadequacy of the LSTM. I hypothesized that in trying to improve either the target loss or the action loss in the LSTM, the model was negatively impacting the accuracy of both. Based on

this hypothesis, I created a second model which was the same as the first model but with output from the max pool layer to two different LSTMs. One LSTM was for the targets, and the other for actions. The two networks were completely separate and output data to their own fully connected network. Both LSTMs took input from the same embedding layer. I found this configuration to be successful and achieved a target accuracy of 73% and action accuracy of 92%. However, I would not test both of my first two models further and record metrics due to the fact that: both models took an excessive amount of training time, hindering much experimentation with hyperparameters; both models used a max pool layer which defeated the idea of utilizing an lstm. As such, continuing with one of the two models would have led to concerns about the ability of the model to generalize as well as the reliability of any good results achieved. I considered using a completely separate model for targets and actions, but decided against this due to the inability to capture trends between actions and targets with this approach. I sought to create a third model which could be extensively tested with different hyperparameters and did not utilize a max pool layer. Originally, I simply removed the max pool layer from the second model. However, the model was unable to complete an epoch within one hour (with the fewest hidden size possible), so I abandoned the 2 LSTM approach. The third model, which I tested and have implemented for this task, is simple: an embedding layer, one LSTM, two fully connected layers. With this model, I was able to test hyperparameters extensively and did not need to rely on any unreliable technique such as using a max pool as input for an LSTM.

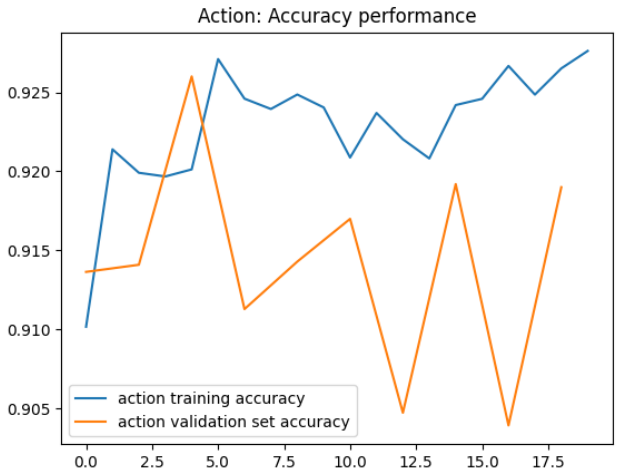
## Testing and Results

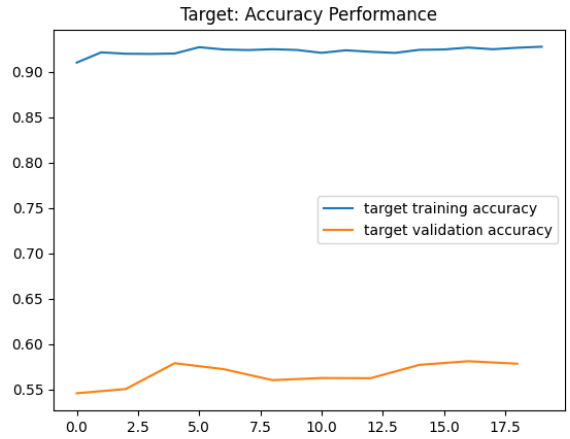
As part of testing, I have tried over 80 combinations of batch size, embedding dimension size, LSTM hidden size, epochs, and optimizers, and shape configurations. My approach was to establish a heuristic for the batch size, embedding dimension size, and optimizer with respect to the hidden_size. Before starting my search, I focused on my model's shape. After removing the

max pool layer, the shape of the input to the LSTM and onwards, had each word token as its own dimension. While this did not interfere with the output of my model, it lead to a noticeable decline in speed. To mitigate this, I reshaped the output of the embedding layer and the input size of the LSTM so that each word token did not have a separate dimension. This lead to a significant speedup in training time and allowed me to test more hyperparameters. The first hyperparameter which I focused on was the embedding dimension. I started by testing values under 40. I tested small embedding dimensions with a hidden size of 100, 120, 200, and 512 and got similar results where the action accuracy plateaued at 94% but the target accuracy varied and was at best 58%. The chart below displays the performance of the model with the

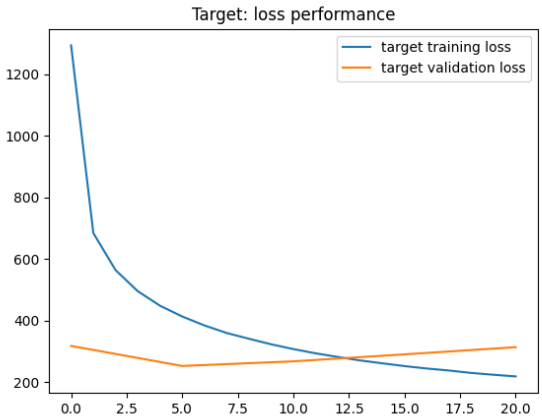



SGD optimizer, a hidden size of 512 and an embedding size of 32. With this, I had my first heuristic, which was to ensure that my embedding dimension was sufficiently large. Next, I focused on the optimizer. For multi-class classification, stochastic gradient descent and the Adam optimizer are supported in PyTorch and are most commonly used. I began by testing a small embedding dimension with a small and large hidden_size, and the Adam optimizer. I noticed that with the Adam optimizer, the delta between training accuracy and validation accuracy was relatively small for targets. However, for my model architecture, the action accuracy performance was very unreliable. I then tested the Adam optimizer with a large

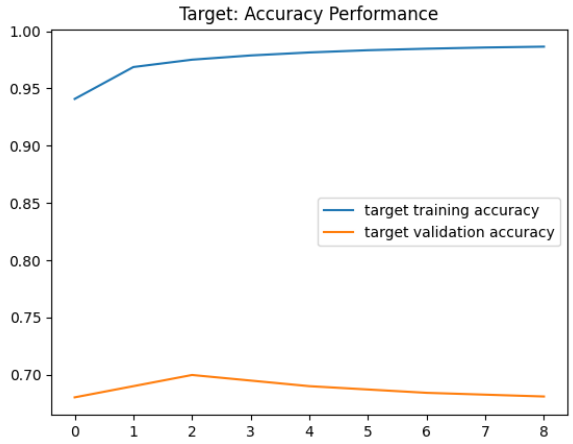


embedding layer (shown above), and although I noticed an overall improvement, the trends remained the same. With this, I chose the SGD optimizer. Next, I tested a model with an embedding size over 60, with a small hidden_size and the SGD optimizer.

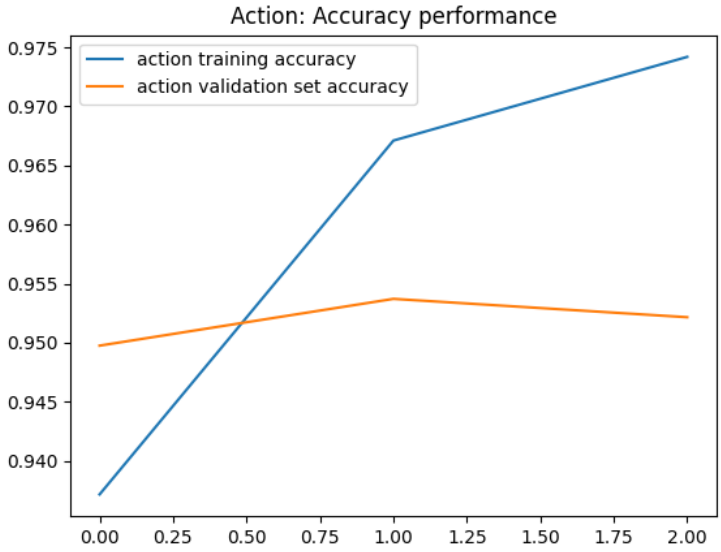



I immediately noticed that models with a small (less than 100) hidden_size were overfitting over many epochs. This was very surprising to me since I expected the smaller models to generalize well. I suspect that this overfitting likely has to do with many duplicate sentences present in the data. I did not notice any improvement in target or action accuracy when switching to models of hidden size greater than 180. Furthermore, I did not find a significant improvement with models of size 100-180, however, I decided that a model of medium size would be the best compromise between a model which requires too much data and time to train and a model which may not be able to model complex dependencies. So, I began to experiment with the number of epochs and



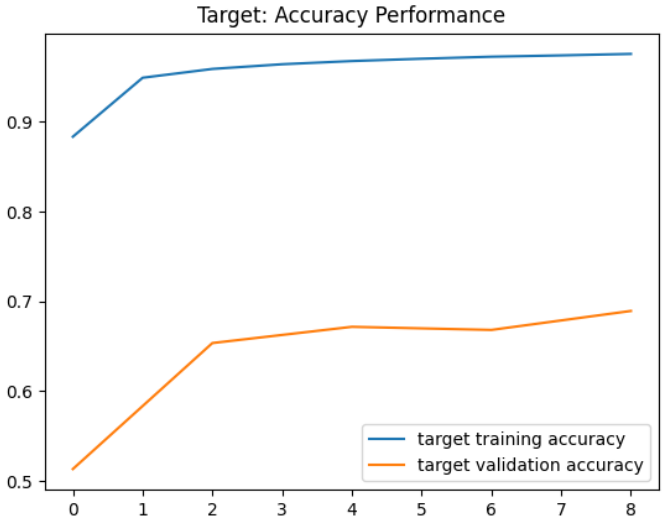

batch size. Having tried a batch size of 20, I reduced the size to 8 epochs and kept the batch size at 500, which I had previously used to train very large configurations. My results showed that the model was still overfitting, but to a lesser extent. The action accuracy remained near 95%, I was, however, focused on optimizing the relatively low target accuracy. So, I reduced the batch size as well as the epochs. I decided on three epochs based on the graph above, and

subsequent trials. After my testing, I decided to use: embedding dimension = 80, hidden_size = 150, epochs = 3, batch size = 100, learning rate = 0.1, and SGD optimizer.




I achieved an action accuracy of 95.2%

My target accuracy is 69.89%.



Although 3 epochs may be considered unusually low, I believe that using a batch size of 100 accounts for the low number of epochs. The model consistently reaches this accuracy and does not overfit. Although the model has not been tested extensively with 2 layers due to slow training times, the model has been tested with a combination of larger and smaller hidden sizes and embedding dimensions as described. The model has also been tested with two different optimizers. For this reason, I believe the hyperparameters chosen for my model are a good set.