5-7-2020

Towards natural language understanding in text-based games

Anthony Snarr
James Madison University

Towards Natural Language Understanding in Text-Based Games

An Honors College Project Presented to

the Faculty of the Undergraduate

College of Computer Science

James Madison University

by Anthony 'AJ' Snarr

April 2020

Accepted by the faculty of the Computer Science dept, James Madison University, in partial fulfillment of the requirements for the Honors College.

FACULTY COMMITTEE:

Project Advisor: Nathan Sprague, Ph.D.

Reader: David Bernstein, Ph.D.

Reader: Kevin Molloy, Ph.D.

HONORS COLLEGE APPROVAL:

Bradley R. Newcomer, Ph.D.,
Dean, Honors College

PUBLIC PRESENTATION

The public presentation requirement has been removed for this semester

Table of Contents

Page

List of Figures

Page

Acknowledgements

Thank you to Dr. Bernstein and Dr. Molloy from the Computer Science department, and Dr. Newcomer from the Honors College for giving their time to read through and review my capstone thesis. And thank you to my capstone advisor, Dr. Sprague from Computer Science, for giving his time to help me throughout this project. His advice was invaluable in helping me to complete this project.

Abstract

TOWARDS NATURAL LANGUAGE UNDERSTANDING IN TEXT-BASED GAMES

Anthony ‘AJ’ Snarr

James Madison University,

Project Advisor: Nathan Sprague, Ph.D.

Text-based games are a very promising space for language-focused machine learning. Within them are huge hurdles in machine learning, like long-term planning and memory, interpretation and generation of natural language, unpredictability, and more. One problem to consider in the realm of natural language interpretation is how to train a machine learning model to understand a text-based game’s objective. This work considers treating this issue like a machine translation problem, where a detailed objective or list of instructions is given as input, and output is a predicted list of actions. This work also explores how a supervised learning system might learn long-term planning and memory through the example of an oracle that always knows the best path. In this exploration, the work here shows that finding this best path is infeasible.

1 Introduction

1.1 TextWorld

Text-based games are games made up of complex, interactive worlds where the state of the world itself is only ever presented to the player in the form of text descriptions. Likewise, the player can also only interact with the world via writing text, almost like navigating through an interactive book. A prime example of text-based games is the famous game, Zork. The opening scene of which is shown in Figure 1 below.

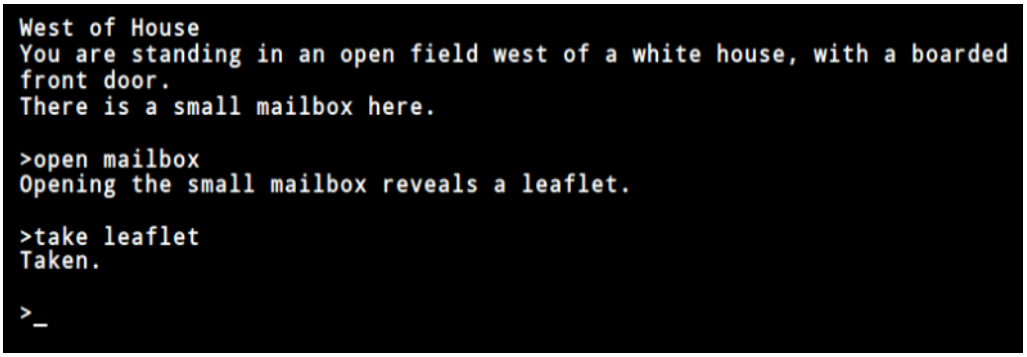


Figure 1 Zork Opening [1]

Text-based games are a very promising space for language-focused machine learning [1]. To play a game well, not only does an agent need to have some understanding of natural language, but it also requires the ability to plan ahead and reason in complex ways based on its observations. According to Côté et al. in [1], this task can easily involve long-term memory and planning, exploration via trial and error, and common sense. Additionally, text in a player's observations only hints at the real state of the world. An agent cannot know if it has been given all the information about its surroundings, making planning more complex.

In a push to help explore and further the area of machine learning in text-based games, the *TextWorld* framework was developed [1]. TextWorld is a framework consisting of an engine capable of generating and presenting text-based games, all with different worlds, objectives,

language, and reward density. Because the framework can be used to randomly generate an unlimited number of different games, data generated from it can be extremely useful in training learning agents. An abundance of varying data helps especially in transfer learning and generalization.

TextWorld makes a very good training ground for a range of machine learning problems: most notably problems in the realms of natural language and reinforcement learning. It is important to note that optimal play of TextWorld games is an open research problem that requires solving many smaller-scale problems first. To help with the focus on smaller subproblems, the framework provides ways of customizing how a game will be presented; for example, a game could have a brief or detailed objective. Since objectives and text descriptions in the game are generated at runtime, an agent playing the game also has the option of peeking at parts of the underlying game state at any time. Viewing pieces of the underlying state allows the agent to have more information about its surroundings. For example, the agent could see the objects in the current room, like doors, cookbooks, and the like. This ability can be very useful in reinforcement and even supervised learning.

1.2 Supervised Learning for TextWorld

Originally, the only goal of this project was to attempt to create a learning agent that would use supervised learning to play TextWorld games with a defined objective. At each step in the game the agent would predict the best action it could take, using TextWorld's features to get a helpful list of possible actions at its current state in the game. The model would train with supervised learning, in theory, by having another subprogram, an oracle, peek into the underlying state of the game and find an optimal path or paths through the game. During training, this oracle could use techniques like imitation learning [2] to teach the real agent how to

play. The reasoning behind this idea is that the training method of supervised learning would be *much* simpler than techniques like reinforcement learning, which have already been explored more in playing text-based games [3]. Much of this idea of an oracle was based on the concept of a referenced "winning policy," mentioned in [1], which would be provided by the TextWorld API. The "winning policy" [1] sounded like it was always a path to the goal from the player's current position, implying a way in TextWorld games to very quickly generate a path to a goal state in the game. An oracle might be able to use similar methods to the generation of TextWorld's winning policy to find a good path to the goal. After exploring this problem for some time, I discerned finding a true optimal path to be infeasible.

1.3 Reading Instructions as Machine Translation

Instead of playing an entire TextWorld game in a simpler way, the rest of this work focuses on how a machine learning model could begin to interpret a game's objective. In TextWorld, there are two kinds of objectives that a game can be generated with, 'detailed,' and 'brief.' A detailed objective in TextWorld is very explicit; in natural language, it describes all the things a player would need to do to win the game going from the start to the goal. In this way, the path to win the game is encoded in this objective description, just in a different format from the commands the player would actually type in-game.

This work frames the problem of pulling key information out of a 'detailed' objective in TextWorld as a machine translation problem—where the text in the detailed objective is considered an input language, which the machine learning model will "translate" into the text making up the sequence of commands to run to complete the game. From a probabilistic perspective, this means that given a sequence of words in a detailed objective, x, and a matching list of commands, y, the goal is to find a target sequence $y = \text{argmax}_y P(y \mid x)$ [4]. That is, the

model is to find a target sequence y that maximizes the conditional probability of words forming a command sequence, given a detailed objective.

1.4 Organization of Paper

The rest of this paper will be organized into three main topics:

1. Machine learning and TextWorld background, and related work
2. Methods and results in supervised learning for TextWorld
3. Methods and results in interpreting instructions as a machine translation problem

The section on machine learning and TextWorld background, and related work is divided into two chapters: one which describes necessary background on machine learning concepts and what a TextWorld game looks like, and another chapter that goes over related works. Both of these chapters cover a range of separate topics and works. The related works chapter will be split into one part that describes related works in TextWorld, and another that describes related works in machine translation.

The general section pertaining to supervised learning for TextWorld will contain an introduction to that initial part of this project, and then the methods and results used in exploring that problem. It will also contain reasoning for why training a supervised learning system with the perfect shortest path in a TextWorld game is infeasible for this project.

The final section on machine translation is split into two chapters. The first will introduce framing interpreting instructions as a machine translation problem, and why this method is chosen. The second chapter in this general section of the paper will go over this project's methods and results for the problem. The paper will end with a chapter on final conclusions, and another on future research.

2 Machine Learning Background

2.1 Supervised, Unsupervised, and Reinforcement Learning

In machine learning, there are three overall types of models one can train: supervised, unsupervised, and reinforcement learning models. The two relevant to the background of this paper are supervised and reinforcement learning. The simplest of these three types is supervised learning. A supervised learning algorithm learns by being trained on labeled data, in other words, each given data point also has a correct answer attached to it. During training, a learning model might be given a data point, and be asked to predict something from that data. Because the data is paired with a correct answer, the model would then be immediately told if it got the answer right or wrong, allowing it to learn immediately from its decision. An example of a supervised learning problem would be predicting if an image contains a bird or does not contain a bird. A model for this task would be trained with labeled images either containing birds or not containing birds. Each time the model was given an image, it would predict "bird" or "not bird" and then be told how correct it was based on the image's label. One of the hardest parts about training a supervised learning model is not always training the model itself, but getting large amounts of labeled data [21].

Whereas supervised learning involves training a model with labeled data, and some immediate feedback of what the correct answer is, reinforcement learning has no 1-1 labeled data. An example of a reinforcement learning problem would be teaching a baby robot how to walk. The robot cannot be told it definitively has found the "correct answer" when it decides to make a certain muscle movement. Getting from point A to point B will require several movements in synchrony, and only a good combination of movements over time will help it start to move its body forward. Therefore, the baby robot can only be told it is doing the right thing

when it has gotten definitively closer to its goal, like successfully lifting one of its legs up without falling over. This encouragement to the baby robot, given as rewards as the robot gets progressively closer to its goal, is the reinforcement.

2.2 Neural Networks

Neural networks are a form of computational system used for machine learning.

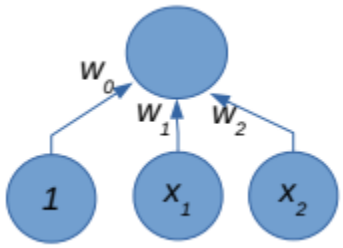


Figure 2.1 A simple 1-layer neural network [13]

In Figure 2.1 is a very simple example of a neural network. This network takes in an input vector, $\boldsymbol{x} = \langle x_1, x_2 \rangle$, and puts it through a linear function using the weights w_1, w_2 as the slopes, and w_0 as the offset from the origin. In this context w_0 is called the "bias weight." Given $\boldsymbol{w} = \langle w_1, w_2 \rangle$, this equation looks like:

$$\hat{y} = \boldsymbol{x} \cdot \boldsymbol{w} + w_0 = (x_1 w_1 + x_2 w_2) + w_0$$

Now say, $w_0 = 1, w_1 = 2,$ and $w_2 = -2$, then an input $\boldsymbol{x} = \langle 5, 4 \rangle$ would cause the network to predict:

$$\hat{y} = (x_1 * w_1 + x_2 * w_2) + w_0 = (5 * 2 + 4 * -2) + 1 = 3$$

This network will "learn" to predict the right output by being given many example inputs, and many example correct answers to those inputs. Each time it makes a prediction from one of those examples, it will look at the expected answer, and adjust its weights very slightly using gradient descent on some "loss function" that indicates how badly the network is doing.

In the previous example, let us say the expected answer was 15. A good loss function for a network like this one is absolute error $L(\boldsymbol{w}) = |y - (\boldsymbol{w} \cdot \boldsymbol{x} + w_0)|$. In this case, the neural network would adjust its weights from $w_0, w_1, w_2 = 1, 2, -2$ to $1.12, 2.60, -1.52$. These new weights would change that old prediction from 3 to 8, getting it closer to the ideal value 15. As this network sees more and more varying input and expected output, it will slowly adjust its weights until it can find a group of weights that give it good predictions on the data as a whole.

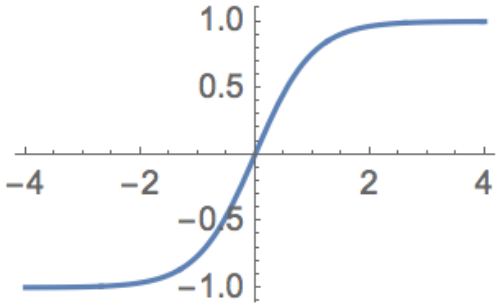


Figure 2.2 tanh function

Neural networks are not limited to fit only straight lines and planes to data. Usually, output from weights in a network will be fed through a nonlinear function, like the $tanh$ function shown in Figure 2.2. Any nonlinearity will reshape outputs to something less linear. It is more typical to have a network made up of several layers of weights feeding into each other, with several nonlinearities in-between. In Figure 2.3 below is a diagram of a simple feed-forward network.

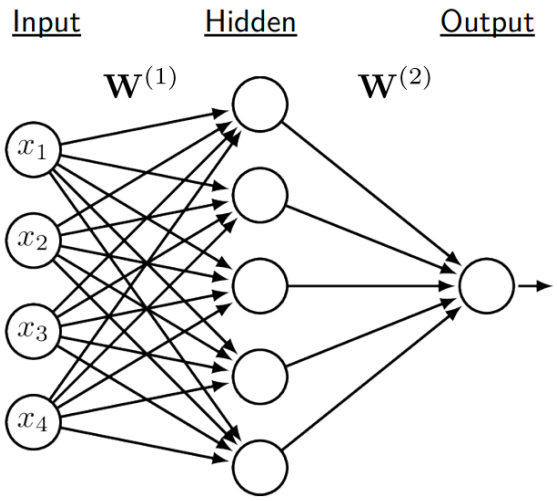


Figure 2.3 A simple feed-forward network [14]

The advantage of multi-layer neural networks is that having multiple non-linearities fed into each other allows the network to learn to remap data between layers however it needs to. In the network shown in Figure 2.3, it will still take in 4 inputs, $\boldsymbol{x} = \langle x_1, x_2, x_3, x_4 \rangle$, and give one output $\hat{y}$, but at the output it will effectively be able to fit any nonlinear model it needs to the data.

In the case of problems where one wants to have a network predict one option from a discrete set of classes, it is useful to have multiple outputs: one output representing the likelihood of each class. One-hot-encoded output (described in section 2.5) is characteristic of a network like this one. For turning these outputs into a list of probabilities, functions like softmax (described in section 2.4) are very useful.

2.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a form of neural network that can process an arbitrary number of inputs over time. What makes them special is their ability to process input in sequence. Unlike a standard neural network, a recurrent neural network keeps an internal state in between inputs, and then applies the same weights to each successive input with respect to its

internal state. In a way, a recurrent network "remembers" something about inputs it has seen in the past and uses that knowledge to decide what to do with new inputs.

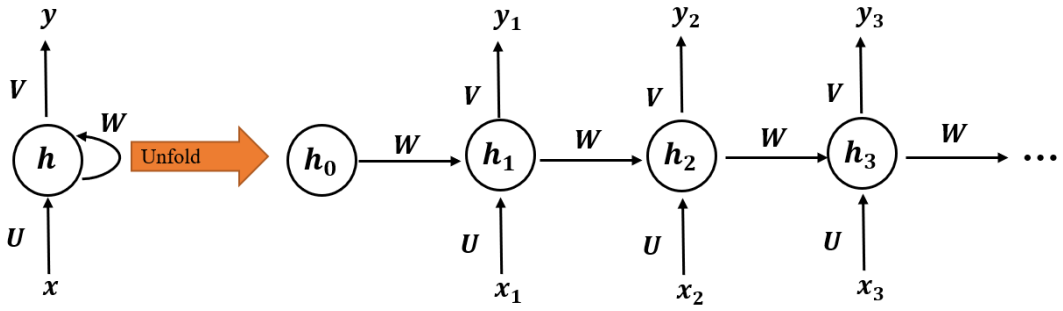


Figure 2.4 A recurrent neural network

Figure 2.4 shows an example of one kind of recurrent network. Each input is initially passed through a set of weights, U, which decided what will be changed in the internal state, h_i. Then, output at that step is determined by taking the current internal state and multiplying it with a set of weights, V. Between each step, the previous internal state, h_i is fed through the weights, W, to become h_{i+1}.

Recurrent neural networks are useful in any kind of data where ordered patterns matter, like in sequences of text. Some text, like the sentence "Hello, my name is Jane" could be processed as a sequence of characters, a sequence of words, or more, all depending on how the RNN is set up. If the network processed characters, it could first be fed numbers representing 'H,' then 'e,' 'l,' 'l,' and so on. When processing words, input must be prepared for the network first. One way to prepare words for an RNN would be to assign a number to each word in the network's vocabulary, and then feed a number-encoded sequence of the text into the network. So, "hello my name is hello" might become "0 1 2 3 0".

2.4 Softmax Activations

The softmax function is a function that takes in a vector of K real numbers, in this case representing some number of classes to predict, and normalizes it into a probability distribution. Given a vector $\boldsymbol{v}$ of length K, the standard softmax function is defined as:

$$\sigma(\boldsymbol{v})_i = \frac{e^{v_i}}{\sum_{j=1}^{K} e^{v_j}} \text{ for each element in } \boldsymbol{v}$$

Softmax will take every element in $\boldsymbol{v}$, whether positive or negative, and output a respective value that is between 0 and 1. The sum of every value in the vector produced by $\sigma(\boldsymbol{v})$ is 1.

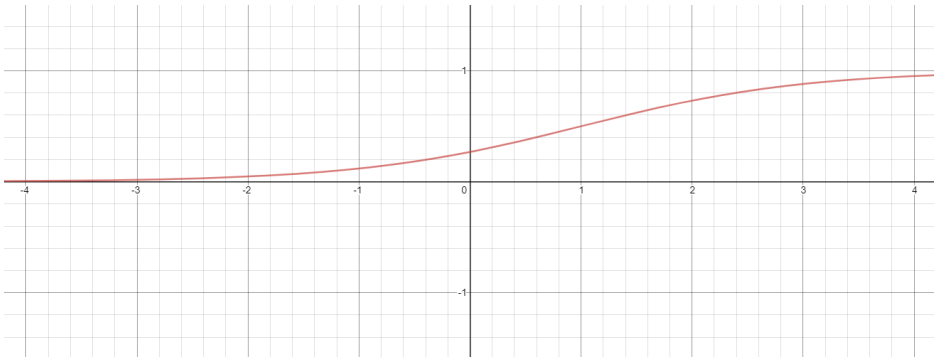


Figure 2.5 Example of a softmax function

In neural networks, softmax is sometimes applied as an activation function in the final layer, in other words to the outputs of the network.

2.5 One-Hot Encoding

In designing a neural network it is very easy to shape output to look like a vector of probabilities. For example, a neural network trying to predict "rock," "paper," or "scissors" might be designed to output a 3-dimensional vector, where a vector $\langle .10,\ .10,\ .80 \rangle$ might indicate a 10% predicted probability of "rock," 10% probability of "paper," and 80% predicted probability of "scissors." One-hot encoding is useful when predicting text with a neural network because it is designed to take advantage of this easy shape.

Consider a network designed to predict text made up of sentences with only 5 different words “hello,” “goodbye,” “my,” “good,” and “friend,” corresponding to numbers 1, 2, 3, 4, and 5 respectively. Converting the sentence “hello my good friend” to numbers (‘tokens’) would produce “1 3 4 5.” A network predicting the last word in the sentence (“friend”) could be designed to output something close to the number ‘5,’ but it would be more useful if it could predict something like $\langle 0.01, 0.01, 0.01, 0.01, 0.96 \rangle$, a list of probabilities, one for each word. This is where one-hot-encoding helps. A one hot encoded version of the sentence “1 3 4 5” would be: “$\langle 1, 0, 0, 0, 0 \rangle, \langle 0, 0, 1, 0, 0 \rangle, \langle 0, 0, 0, 1, 0 \rangle, \langle 0, 0, 0, 0, 1 \rangle$,” where each word is represented by a vector containing all 0’s, and a 1 (100% probability) in the position for the word it represents. One hot encoding makes output from a neural network very convenient and informative. The main disadvantage with the process is that as the network’s vocabulary size becomes larger, each one-hot-encoded vector must become larger, such that space requirements are increased linearly $O(n)$ where n is the vocabulary size.

3 Related Work

3.1 Related Works in TextWorld

3.1.1 About Playing TextWorld

TextWorld games are always worlds made up of some number of rooms, filled up with some number of interactable objects, and embedded with a quest. A quest is the player's goal in the game, for example it might be to reach a room while holding a certain key, or to drop a pepper on the ground for no apparent reason. Below is an example of playing a game in TextWorld, played via the command line.

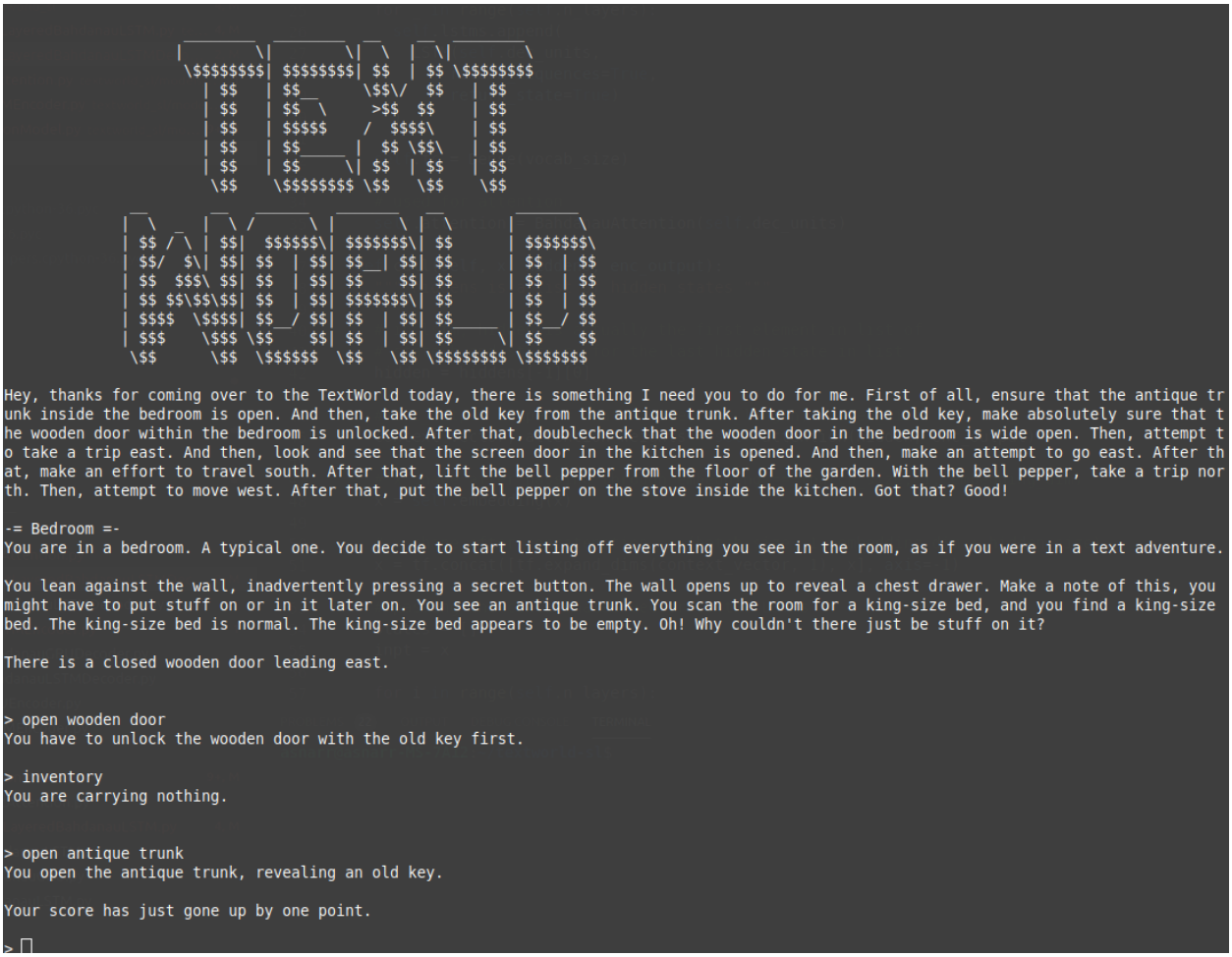


Figure 3.1 A game of TextWorld

Each game in TextWorld begins with a title, “TEXT WORLD,” and an objective that describes the quest, followed by the description of where the player is. There are two kinds of objectives that TextWorld games can be generated with: ‘detailed’ objectives, like in the game in Figure 3.1, and ‘brief’ objectives. Detailed objectives are very explicit and describe everything a player needs to do to win. If the game shown in Figure 3.1 had been generated with a brief objective, it would probably have said, “Put the bell pepper on the stove inside the kitchen,” without giving further context.

3.1.2 Neural Approaches to Conversational AI

Machine learning algorithms processing text in the form of natural language have been the focus of research for years. The task of language-understanding has generally focused on in three areas: question answering agents, task oriented dialogues, and social chatbots [5]. Researchers in the past year and a half have even tried applying these techniques to TextWorld itself.

3.1.3 Playing Text-Adventure Games with Graph-Based Deep Reinforcement Learning

In their work, Ammanabrolu et. al, frame the question of finding the correct command at each step in a TextWorld game into a question-answering problem [3]. Playing a game can be thought of as continuously asking the question, “What is the right action to perform in this situation?” In order to have the context to answer a question, Ammanabrolu et. al. introduce a “knowledge graph” that represents all of the player’s knowledge about the current game state. This knowledge graph stores 3-tuples of $\langle subject, relation, object \rangle$ in text form, where directed edges in the graph hold the relation from a subject to respective objects. An example knowledge graph is shown in Figure 3.2.

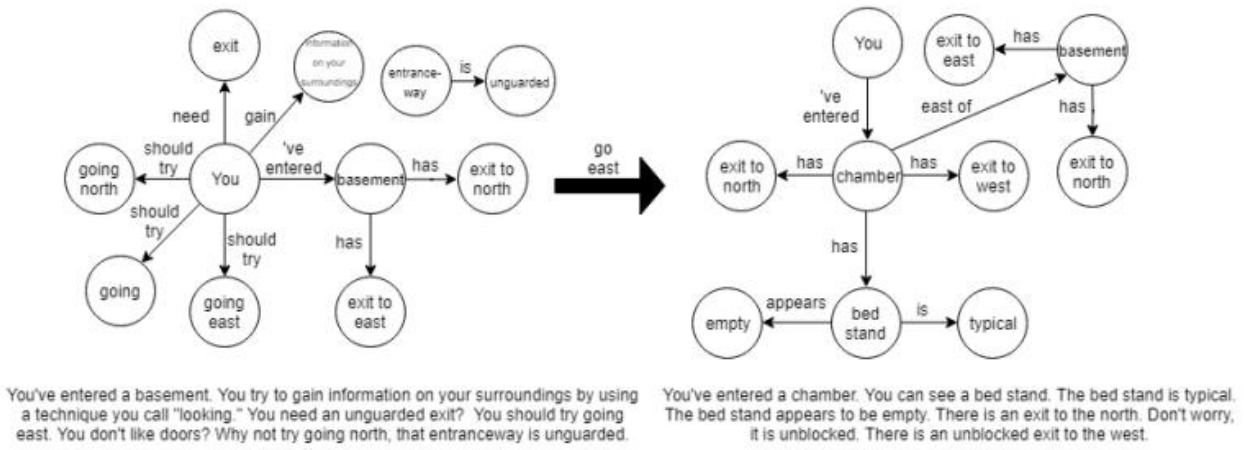


Figure 3.2 Graph state update example [3]

At each step in the game, this knowledge graph is used to help prune the action space and eliminate all but the top k possible commands, which are chosen using an action scoring function. They compare performance with the baseline LSTM-DQN developed in [19] and are able to reach the solution 40% faster on a small game with 10 rooms, 20 objects and a quest length of 5 [3]. On a large game with 20 rooms, 40 objects, no agents are able to get the maximum score, but the best model is able to reach the same score faster than LSTM-DQN.

3.1.4 Towards Solving Text-based Games by Producing Adaptive Action Spaces

In their paper, Tao et. al. try three generative models that attempt to generate a set of all valid commands for a given context [20]. Predicting what commands are available is a necessary step in playing text-based games like TextWorld, because there is no list given to the player that tells them what commands are available; possible actions to write must be inferred from a given context. Toa et. al. try three approaches:

- a pointer-softmax model that uses beam search to generate multiple commands
- a hierarchical recurrent model with pointer-softmax generating multiple commands at once
- a pointer-softmax model generating multiple commands at once

Of the three, the two generating multiple commands at once perform by far the best. One variation of the pointer-softmax model generating multiple commands at once achieves ~98% precision and around 95% recall on commands seen. This model is also able to correctly generate 80% of commands never seen before in training [20].

3.2 Related Works in Machine Translation

3.2.1 Long Short Term Memory (LSTM)

A critical problem in recurrent neural networks (RNNs) has always been keeping state over a long period of time. In the realm of text understanding, a RNN taking in one word at a time might have to remember a term 30 words back to even slightly pick up the semantic meaning of its current word. For example, in the text, "The dog was excited by the approaching mailman on the sidewalk. It was so wound up it barked," a RNN would have to have to be able to look back far across all its input just to see that 'it' has some relation to the dog. The long short-term memory network (LSTM), presented by Hochreiter et. al. [6], attempts to solve this problem through the use of internal gates that function as a way of deciding what to forget from the previous state, and also what to selectively remember from the current state.

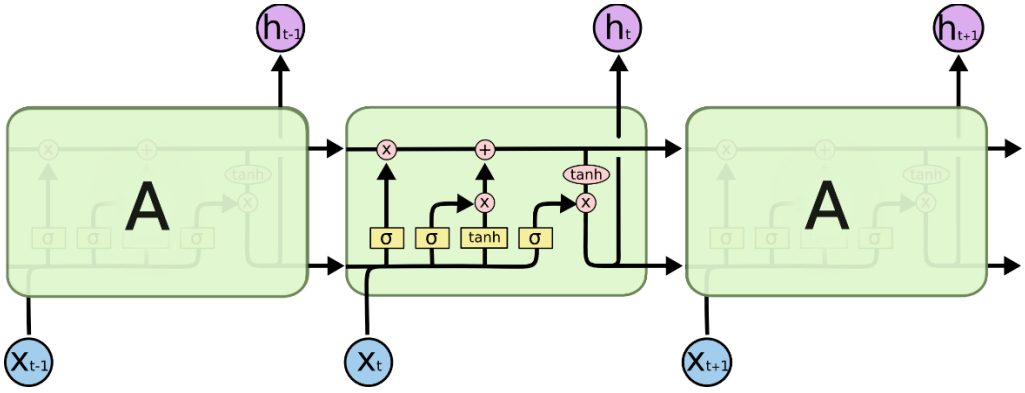


Figure 3.3 LSTM Gates Diagram [17]

The LSTM proves extremely useful across a variety of tasks, like handwriting recognition, speech recognition, and more, given that it is properly used. There are an immense

number of papers expanding on the use and construction of LSTMs. As well as development of similar mechanisms, like the gated recurrent unit (GRU), which, among others has gained some traction [7].

3.2.2 Encoder-Decoder Architecture

Machine translation, the problem of translating from one language to another, has traditionally been reliant on statistical machine translation (SMT) systems [8]. Encoder-decoder architecture, proposed by Cho et. Al. in 2014 [9], was one of the first steps towards usable neural machine translation (NMT) systems. Neural machine translation refers to the application of deep learning and neural networks to solve the problem of translation.

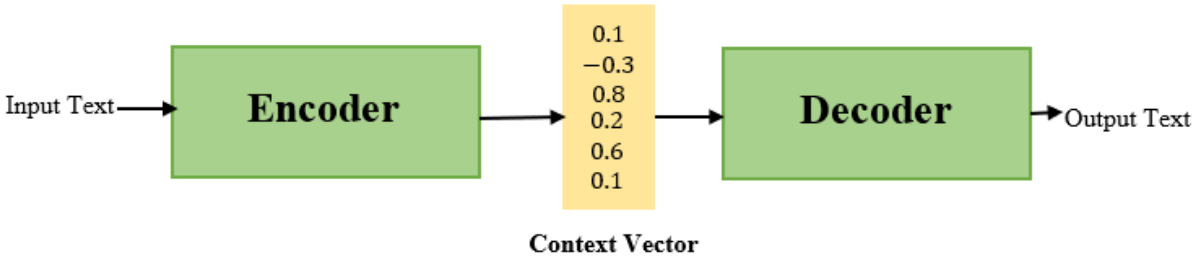


Figure 3.4 Most Basic Encoder-Decoder

Encoder-decoder architecture [9] is characterized by two recurrent neural networks. The first network, called the encoder, is fed an input sequence to be translated, which it will encode into some useful representation of that source sequence. The second network, the decoder, then does the task of decoding that representation into an output sequence, the translation. Modern best practice NMT systems now almost exclusively use some form of an encoder-decoder architecture [8].

3.2.3 Encoder-Decoder with Attention Mechanisms

An issue with the basic encoder-decoder architecture is that the encoder needs to be able to compress all information it has about a source sequence into a fixed-length vector, to be

passed to the decoder. This fixed-length vector might make it difficult for the network to handle translation of long sentences. In their paper on aligning output sequences with input, Bahdanau et. al. [4], propose use of an attention model, which tries to score which parts of the input sequence are most relevant to the current prediction in the output sequence. For example, take the English sentence, “Behind President Lincoln, Booth fired the gun at point blank range.” In Spanish, this sentence translates to “Detrás del presidente Lincoln, Booth disparó su arma a quemarropa.” When predicting the last Spanish word, “quemarropa,” the most relevant words in the English sequence will be “at point blank range,” and so the attention model will probably score those words higher. Later works have expanded on the use of an attention mechanism in encoder-decoder architectures [10]. Since Bahdanau et. al. [4], other papers [10] have also proved an increase in performance over basic models using an attention mechanism, especially in longer translations.

3.2.4 Teacher Forcing

Normally, a recurrent neural network used for machine translation will work recursively, where it uses the last token it predicted as an input for the next prediction.

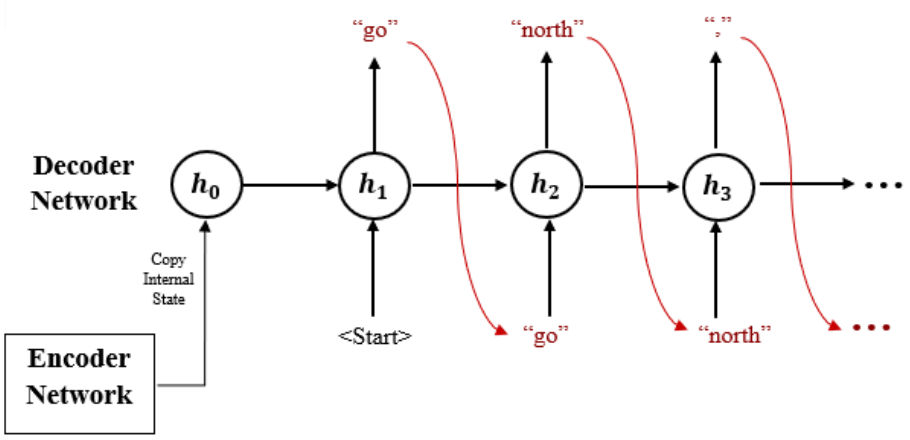


Figure 3.5 Without Teacher Forcing

In teacher forcing, proposed in [12], instead of feeding in predictions from the last time step, shown in Figure 3.5, during training, inputs at each time step are fixed, so that they will always be the previous expected output. Fixed inputs are shown in Figure 3.6.

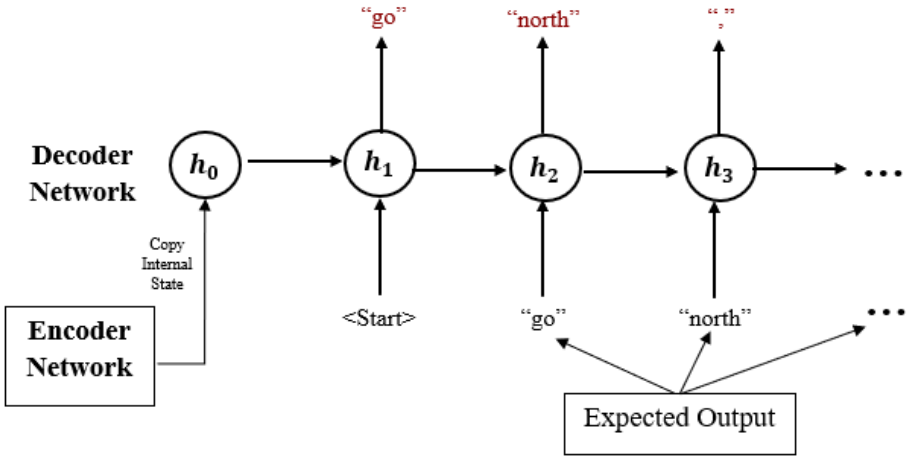


Figure 3.6 With Teacher Forcing

Fixing inputs at each step to be previous expected outputs forces the model to train as if it was making the correct predictions each time, giving it a head start in training and in theory, speeding up training time.

3.2.5 Google's Neural Machine Translation System

Google's NMT system, presented in Wu et. al. [8], was hailed as the best translation architecture at the time it came out, and greatly outperforms traditional statistical systems. It uses 8 LSTM layers stacked on top of each other for its encoder network, with the first of the 8 set up to be bi-directional on input; where "bidirectional" refers to the idea that it both reads input forward and backward to get added context. The decoder network is also made up of 8 LSTM layers, with an attention mechanism based on Bahdanau et. al. [4], consisting of a feed-forward network with a single hidden layer. Part of the goal in creation of the system was to make the encoder and decoder networks as deep as possible i.e. to make as many stacked LSTM layers as possible. Wu et. al. were able to get up to 8 stacked LSTMs by making very specialized custom

LSTM layers that interacted with each other through what Wu et. al. call “residual connections” between the layers [8].

4 Supervised Learning for TextWorld

The original goal of this project was to attempt to create a learning agent that would use supervised learning to play TextWorld games with a defined objective. The learning agent would train to play TextWorld games by being taught to follow the example of an oracle that knows the shortest path. Creating this 'oracle' would be one part of the project. To work, in any given TextWorld game, this oracle would have access to the underlying game state and find shortest paths through the game by running a graph search through the graph of game states.

This work finds the problem of searching for the shortest path(s) to a goal in TextWorld to be intractable. This whole chapter describes the initial work done to reach that conclusion. This research is significant because it highlights one avenue to solving this problem which does not seem to be feasible.

4.1 Data Generation

Testing any machine learning models required a dataset containing different TextWorld games. Here includes a short description of the code that was used to interface with the TextWorld API.

4.1.1 Code for Data Generation

Among other supporting code, the codebase used for this project includes a simple class used for generating games through TextWorld's API, compiling them, and saving them to a file. Using a dedicated class for game generation is advantageous as it allows for added randomness to the parameters normally passed into TextWorld, and enables use of default parameters.

Italics below means a set of min, max values can be specified instead of the shown value. Here are the options available in the class for game generation:

- *nb_rooms (int):*
  Number of rooms in the game.
- *nb_objects (int):*

Number of objects in the game.
- *nb_parallel_quests (int):*
  Number of parallel quests, i.e. not sharing a common goal. 1 = no parallel quests.
- *quest_length (int):*
  Number of actions that need to be performed to complete the game.
- *quest_breadth (int):*
  Number of subquests per independent quest. It controls how nonlinear a quest can be (1 means a linear questline).
- *quest_depth (int):*
  Number of actions that need to be performed to solve a subquest
- filename (str):
  Filename (not path) of the compiled game (.ulx). Also, the source (.ni) and metadata (.json) files will be saved along with it.
- force_recompile (bool):
  If True, recompile game even if it already exists in a file.
- is_detailed_objective (bool):
  Whether or not to use a detailed objective, which says step by step what to do.
- regenerate_seeds (bool):
  If True, will regenerate seeds each time before generating a new game.
- seeds:
  An optional set of seeds that can be used for game generation. These include:
    * `'map'`: control the map generation;
    * `'objects'`: control the type of objects and their location;
    * `'quest'`: control the quest generation;
    * `'grammar'`: control the text generation.
  Any seeds not given are randomly generated.

To play a game, this project's codebase includes a class that executes the actual steps of the game, using decisions made by an 'Agent' interface and logging filtered output using a special 'Writer' interface. Interfaces like these are used so that any agent can be switched out easily by passing in a different agent to the player, and different writers can be used for either dataset generation (writing to a file) or simply printing to std out, for testing.

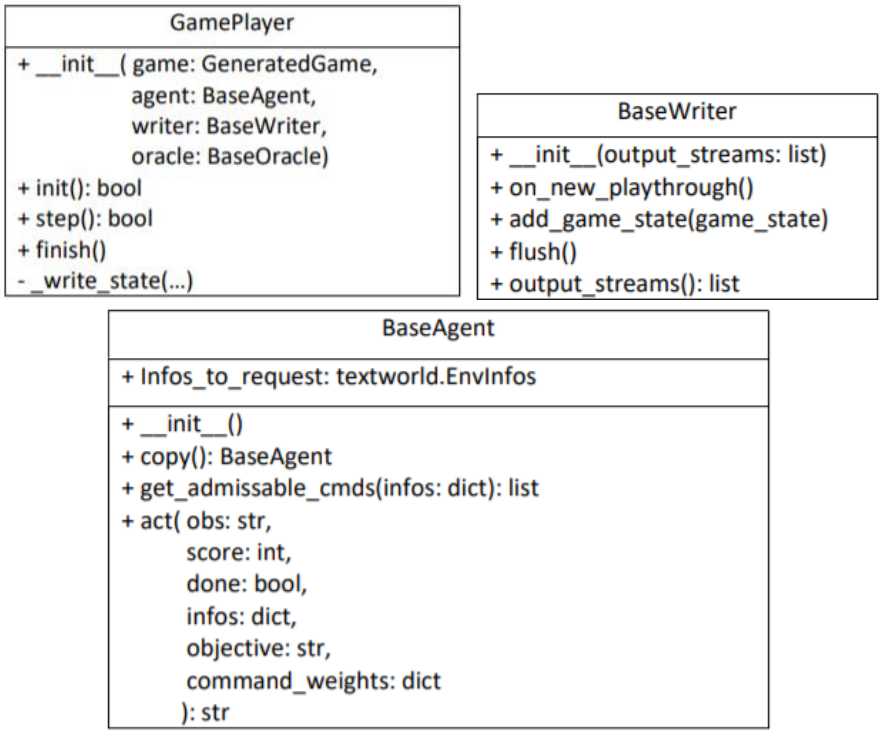


Figure 4.1 Data Generation UML

4.1.2 Getting the Textworld Winning Policy

As described in [1], the “winning_policy” in TextWorld is a sequence of commands that can be used to win the game, stored in a Python list. This winning policy is important to the idea of a supervised learning implementation because its existence implies there is a very fast way to generate paths to a goal state in TextWorld games. Assuming that this winning_polcicy in the TextWorld framework shows the best path to the goal from the current state, it must either be generated somehow at runtime or efficiently encoded at game generation. The supervised learning implementation proposed here relies on some oracle that could similarly find winning paths quickly. If the winning_policy did turn out to be a shortest path from *any* state in the game to a goal state, this searching would effectively be done for the oracle already.

A "winning_policy” that contains a list of actions that can be used to win the game is not in TextWorld’s higher level API, the part of TextWorld that players would normally have access

to; but, it does exist in the lower level parts of the huge TextWorld framework. It turned out that a winning policy can be found two layers behind the higher level API, but it does not change once the game starts. It is simply a list of commands, closely resembling the objective, that will take a player from the starting state to a goal state. This winning policy is also *not* guaranteed to be the shortest path to the goal; it is not an optimal policy. Because this list is not generated dynamically (from any state the player may be in), and because it is not an optimal path, another method to very quickly generate a path to the goal would need to be found in order to implement the oracle.

It is also important to note that this winning_policy list used inside TextWorld's backend will always work as a non-optimal path from the start state to the goal state because games generated by TextWorld always end up being deterministic. While text-based games in general have some randomness to them (text games are stochastic), in practice generated TextWorld games do not.

An example **detailed** objective, and the respective winning policy:

Detailed Objective:

```
You are now playing a exciting session of TextWorld! Here is
how to play! First stop, make an attempt to take a trip
south. With that accomplished, attempt to go to the west.
Following that, make an effort to take a trip west. With
that done, head north. Next, go east. With that over with,
pick up the candy bar from the table. And then, place the
candy bar on the floor of the cookery. And if you do that,
you're the winner!
```

Resulting Winning Policy:

```
['go south', 'go west', 'go west', 'go north', 'go east',
'take candy bar from table', 'drop candy bar']
```

4.2 Exploring Path Generation

In order for the oracle to train an agent using correct decisions at each point in a game, there needed to be some way of generating a path or set of paths from the current position to the goal. A winning path can be thought of as a list of actions that will get a player from his current position to a goal. Ideally, generated winning paths should be ones that would get the player to a goal state the fastest.

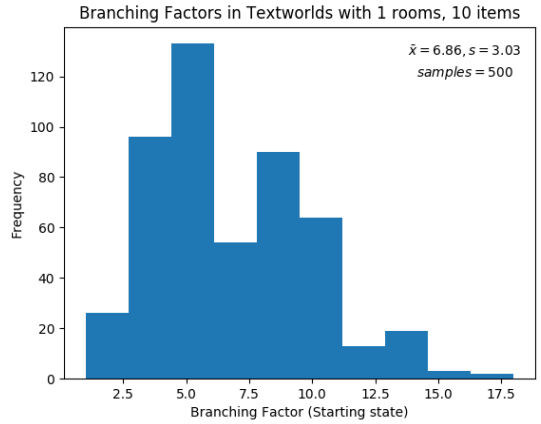
4.2.1 Analyzing Branching Factors

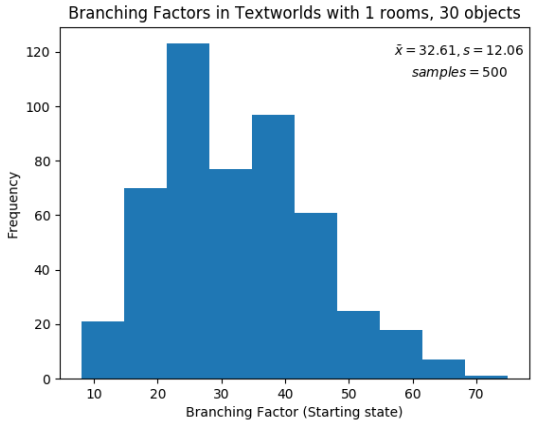
In thinking about path generation, the first steps considered in this project were to explore how complex a search might be. Any game of TextWorld can be thought of as a collection of possible game states, where one game state might have a player in a specific room, with some specific inventory, and some specific number of objects in each room. Any actions the player makes, like moving north, will change the game to some different state, in this case with the player saved in a different area. This collection of possible states and actions connecting them can be thought of as a directed graph, where nodes are states, and edges are the actions that move a player from one game state to another. As such, searching for a path to a goal state in TextWorld is the same thing as a search through a uniformly weighted directed graph.

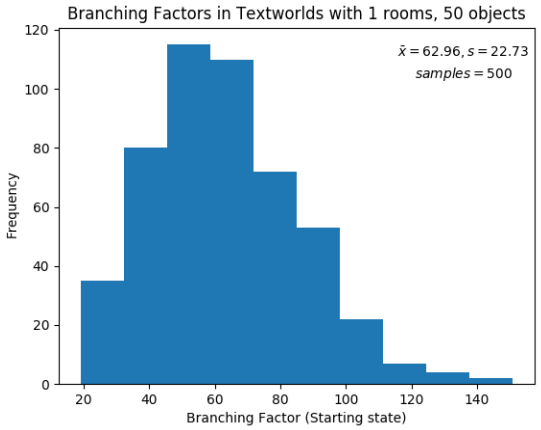
To explore path generation, this work first analyzes what branching factors in the graph of game states might look like across a variety of TextWorld games, each generated with different parameters. Note that in TextWord, there are many commands in the game that will not change the player's state, like "examine," "look," or "inventory." As far as this analysis goes, these commands do not affect the branching factor as they can be ignored during a search.

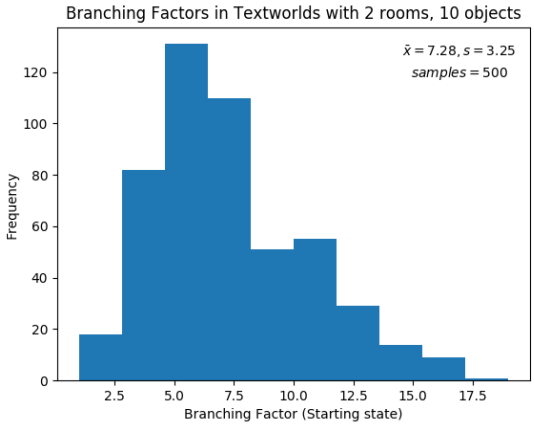
This analysis was done by taking random samples of branching factors (the number of state-affecting commands) from TextWorld games generated with *r* number of rooms and *o*

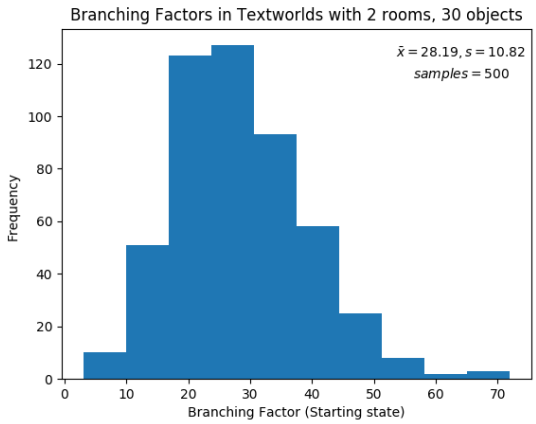
number of objects distributed throughout the world. Each sample was taken from the first state in a newly generated text world.

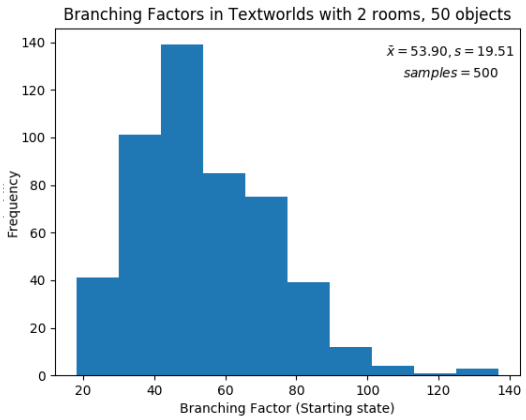




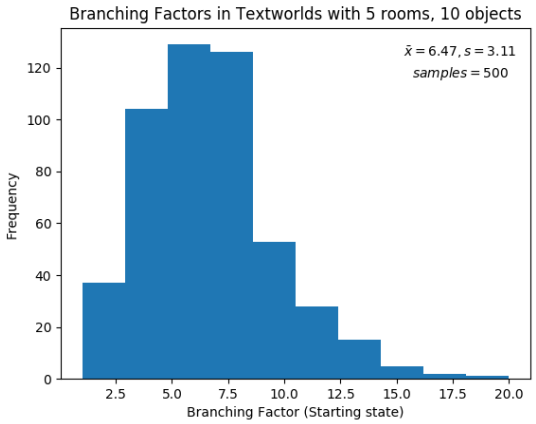


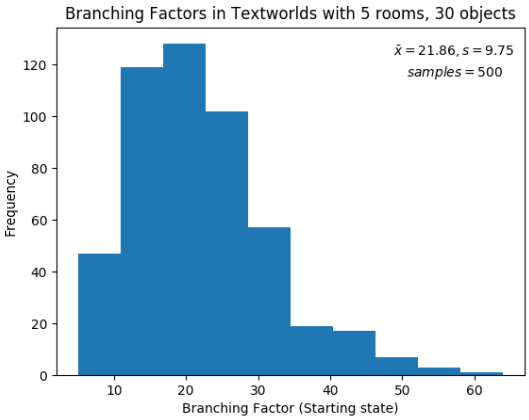


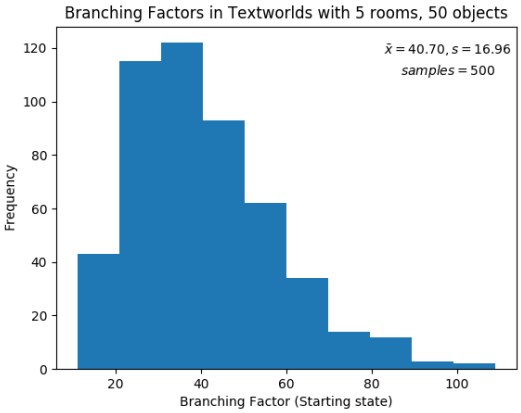


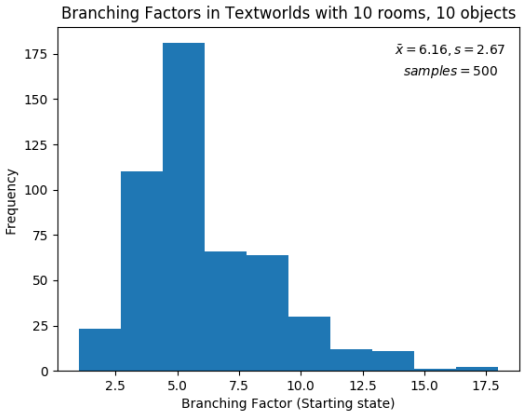


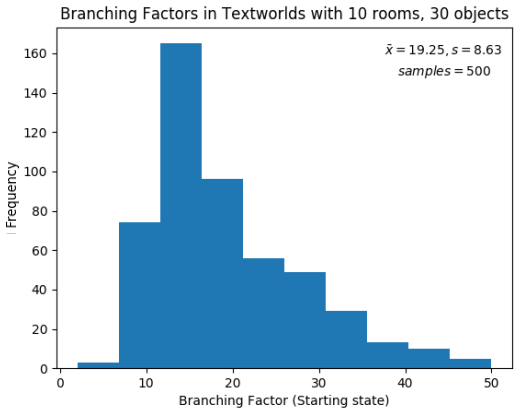
Figures 4.2.1-4.2.6 Branching Factor Charts

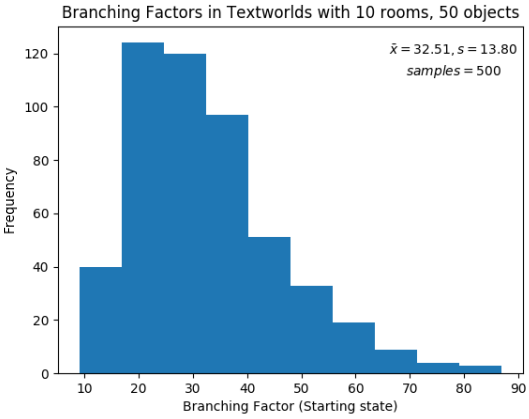












Figures 4.2.7-4.2.12 Branching Factor Charts

Repeated below are some of these statistics in a more compact format:

Branching Factors (sample size 500)

	1 Room	2 Rooms	5 Rooms	10 Rooms
10 Objects	x=6.86 s=3.03	x=7.28 s=3.25	x=6.47 s=3.11	x=6.16 s=2.67
30 Objects	x=32.61 s=12.06	x=28.19 s=10.82	x=21.86 s=9.75	x=19.25 s=8.63
50 Objects	x=62.96 s=22.73	x=53.90 s=19.51	x=40.70 s=16.96	x=32.51 s=13.80

Figure 4.3 Branching Factors Table

Hypothesis:

Average branching factor would go up noticeably with the number of objects, but would go down noticeably with room size. A world with 10 rooms and 10 objects for example would have a noticeably lower branching factor than a world with only 1 room and 10 objects, but variance would go up with the number of rooms, as interactable objects might be distributed in many more ways with more rooms.

Analysis:

Average branching factor is strongly impacted by the number of objects, but surprisingly not as affected by the number of rooms. In a TextWorld game with only 10 objects, changes in the number of rooms seem to have almost no noticeable effect. As object numbers increase, change in rooms seems to have a more noticeable effect on average, but this change does not appear to be great. Interestingly, as the number of rooms goes up, the standard deviation in branching factors seems to go down, instead of up. I am not sure what could cause that kind of change.

Potential problems in samples:

All these samples were taken from the first state in a TextWorld game. Could branching factor be different in the first state? Might it go up later as containers are unlocked or opened, possibly revealing more objects?

It is important to note that in a single textworld, branching factor can change in each game state. A player may enter a room with a different number of exits and objects to interact with. Or a player might pick up an apple, and then have both options to either eat or drop it, causing an increase in branching factor.

The branching factors found in this experiment appear notably high for a graph search that must be done once per game state in an instance where hundreds of thousands of game states must be recorded and trained on. In a small textworld, say with 5 rooms and 30 objects, some game states just 2 standard deviations above the mean can have branching factors of around 40, meaning just searching every game state 3 actions away from the current state would mean visiting 40^3 states. In a larger TextWorld game, this branching factor could be huge.

4.2.2 Further Analyzing the Search Space

At any point in the game, a TextWorld game has a player, some number of rooms (r), and some number of objects (obj) distributed throughout those rooms and the player's inventory. As the player can also pick up, move, and place down objects in different rooms, the total number of states in a game can be represented by the formula:

$$n_{states} = (r + 1)^{obj} + r$$

Let us picture a 'normal-sized' TextWorld game with $r = 5$ and $obj = 30$. In this game, the 30 objects can be stored in 5 rooms + the player's inventory, and the player can be in 5 different locations:

$$n_{states} = (5 + 1)^{30} + 5 = 2.2e23$$

The number of searchable states in this normal TextWorld game is huge – far too many to search every state, or even travel too deep into the graph, considering branching factors of around 40 for games with $obj = 30$ and $r = 5$.

In a uniformly weighted graph such as this one, the standard shortest-path searching algorithm is breadth-first-search (BFS). Given a graph search with branching factor b and search depth d, the time complexity of breadth first search is $O(b^d)$. In a TextWorld game with branching factor of 10, like the example at the end of section 4.2.1, this equation becomes $O(40^d)$ for some depth d.

While time complexity is an obstacle, the main disadvantage with a BFS is its space complexity. During its search, a BFS must store a huge portion of the states in the graph, creating a space complexity of $O(V) = O(b^d)$. In this large graph of game states, the space complexity of BFS makes it infeasible. The solution to this problem is to use an iterative-deepening search (IDS), which reduces space complexity to $O(d)$ at the cost of a small number of repeated visits to nodes in the graph; however, a pure IDS still becomes infeasible in its time complexity. With a time complexity of $O(40^d)$, searching deep into the graph for a goal state would grow exponentially with the depth. For an oracle that needs to run hundreds of thousands of searches, this would not be fast enough.

4.3 Conclusions on Supervised Learning for TextWorld

Unfortunately, training a supervised learning system in TextWorld by teaching it to follow a perfect shortest path does not turn out to be feasible. The problem of finding even a single shortest path to a goal in the TextWorld state space is infeasible in anything but very small TextWorld games. Generating a distribution of paths to train with, instead of a single best path, is even less feasible. Since a pure IDS will not work in the full state space, further research into this idea might involve finding how to generate close-to-best paths, or ideally a distribution of paths with which a supervised learning agent could train.

5 Instruction-Following as Machine Translation

As stated before, in this modified addition to the project, this work explores how a machine learning model could begin to interpret a text-based game's objective. Specifically, this work considers viewing the problem of pulling key information out of a detailed objective in TextWorld as a machine translation problem—where the text in the detailed objective is considered an input language, which the machine learning model will "translate" into the text making up the sequence of commands to run to complete the game. Formally defined, machine translation is the problem where, given a source sentence, x, the goal is to find a target sentence $y = \text{argmax}_y P(y \mid x)$ [4].

Here again is an example of a detailed objective, and its respective list of commands:

Detailed Objective:

> `You are now playing a exciting session of TextWorld! Here is how to play! First stop, make an attempt to take a trip south. With that accomplished, attempt to go to the west. Following that, make an effort to take a trip west. With that done, head north. Next, go east. With that over with, pick up the candy bar from the table. And then, place the candy bar on the floor of the cookery. And if you do that, you're the winner!`

Resulting Winning Policy:

> `['go south', 'go west', 'go west', 'go north', 'go east',`
>
> `'take candy bar from table', 'drop candy bar']`

There are many challenges to making this transformation from objective to a list of actions. First of all, there is no one-to-one conversion from any phrase in the objective to a command in the resulting list. For example, just expressing the command "open chest" in the objective could be: "then, open the chest," "make sure the chest is open," "and then, doublecheck

that the chest is ajar," and many more different wordings. Many sentences in the objective may also have nothing to do with the resulting list of commands, as in the first two sentences and the last sentence in the example objective above. These generated sentences can be a wide range of phrases, are also not necessarily placed in the objective, and in rare cases, can be sprinkled throughout.

Viewing this transformation as a machine translation problem has many advantages. First, it may seem odd to view transforming the objective into a set of commands in the same way as translating from one language to another, since both texts are in English; but looking more closely, both texts have different rules of syntax and only vaguely similar vocabularies. They can almost be thought of as texts that both have the same meaning, but written in two different languages. This nature of the two texts makes this problem work very well as a machine translation task. Another advantage of viewing this transformation as a machine translation problem is that a lot of research has gone into neural machine translation in the past. Much of the results of this research can be leveraged to solve this problem, for example encoder-decoder architecture as described in [7].

It is important to note that purpose of this project is not just to solve this problem, but to also show that it *can* be solved using machine learning. This work uses and focuses on neural machine translation, i.e. machine translation using machine learning and neural networks; but, one could make the argument that this complication could be avoided. Looking at the example TextWorld objective above, it looks like it would be possible to manually write a program that would piece together the translation from manually recorded phrases in the objective, maybe using some kind of table. One disadvantage of this simpler approach would be that a table of translations would be huge. Since there is no one-to-one translation, there would need to be a

huge number of patterns created just to capture one command. Another disadvantage of this phrase-based approach is that it would not be as easily generalizable to other vocabularies apart from the default vocabulary in TextWorld. Part of the point of trying to solve this problem is to solve it in a way that is generalizable. There are also other reasons why a machine learning solution is focused on here.

By showing that this problem can be solved using machine learning, it leaves open of the possibility of other machine learning models being able to build off progress made in this solution. For example, using parts of this solution to help develop a full agent that could play text-based games. A benefit of using neural networks to perform this task is that, as a network learns to predict a list of commands from a TextWorld detailed objective, it must also learn how to interpret and store some meaningful representation of TextWorld objectives. A neural network that knows how to pull some meaning out of more detailed objectives like the one above might in the future be able to be used to help interpret some meaning from much less detailed, more open-ended objectives in TextWorld games.

In order to attack this as a machine translation problem, I consult research conducted over the last few years in the area of neural machine translation, which specifically focuses around solving the problem of translation using neural networks. This work most notably focuses on research developments in translation like the encoder-decoder architecture [7] and attentional mechanisms like in [4], as well as the state-of-the-art neural machine translation architecture developed by Google [8] [11].

6 Methods and Results

Following the path of many other neural machine translation (NMT) systems [4][8][9][10], the models proposed here are based off of the encoder-decoder architecture proposed in [7]. The following sections describe three neural machine translation systems which build off of each other, as well several variations of each one. As a starting point, the first few of these models were built off of code from a Keras tutorial at [18].

To reiterate, encoder-decoder architecture in its most basic form consists of two recurrent neural networks: an encoder model and a decoder model. The encoder model will read in the source sequence and transform it into some 'context vector' containing some meaning of the source, which is then fed into the decoder, which will decode the context into a target sequence.

6.1 Initial Dataset Generation

Testing the NMT systems described here required a new dataset to hold translations. Here I call this dataset TWTranslationDataset. It is formatted in this way:

objective on this line
winning_policy on this line
(###)
objective on this line
winning_policy on this line
(###)
objective on this line
winning_policy on this line
(###)

Here is an example (objective line italicized):

Get ready to pick stuff up and put it in places, because you've just entered TextWorld!

Your first objective is to attempt to travel west. That done, attempt to go south. Then, travel east. After that, move south. And then, retrieve the non-euclidean keycard in the

bedroom. Then, make an attempt to move north. Then, insert the non-euclidean keycard into the non-euclidean locker's lock to unlock it. And then, open the non-euclidean locker inside the studio. Got that? Good!

go west, go south, go east, go south, take non-euclidean keycard, go north, unlock non-euclidean locker with non-euclidean keycard, open non-euclidean locker

(###)

Initial generation of training, validation, and testing data ran into two problems. First, data generation started out very slow. It ran just fast enough to generate about 1000 TextWorld games (objective-winning policy pairs) an hour, but the goal was to get about 50-150 thousand of these pairs at least. Second, the dataset, when loaded, was loaded all at once into main memory. This use of memory was not a problem with a few thousand translations, but it became more of an issue as the size of the dataset got bigger.

The final data-generation-related code tries to address both of these problems. To speed up data generation, generation of multiple TextWorld games is parallelized on different processors. This was a very simple task, as the generation process is embarrassingly parallelizable. The main machine used for generating data in this project has 12 cores, so this change ended up boosting data generation to about 10 times the speed it was before.

To relieve memory used by the dataset, the final code written for data generation includes an implementation of a Keras “Sequence,” which allows models to load the data from secondary storage as it is needed during training, validation, or testing.

All generated data is split into three datasets: A training set, a validation set, and a test set. The training set and validation set are stored together in one file, where the first 10% of the

translations in that file are put into the validation set during runtime, and the last 90% are put into the training set. The validation set is used in each model to help tune the number of epochs used during training. The test set is used only for sample predictions throughout this paper, and the accuracy on the full test set is only shown once when evaluating the final model in section 6.5. All data in each set is stored in plain text, and converted to a one-hot-encoded format during runtime.

Once data generation had been parallelized, it was possible to generate about 10,000 objective-winning policy pairs per hour. The full dataset generated has 500,000 of these pairs, totaling to about 50 hours of computing time. This full dataset of 500,000 translations takes up 643MB.

6.2 My First Encoder-Decoder

The initial model I tested in order to get a feel for the data and encoder-decoder architectures in general. It is the most basic encoder-decoder architecture one can make.

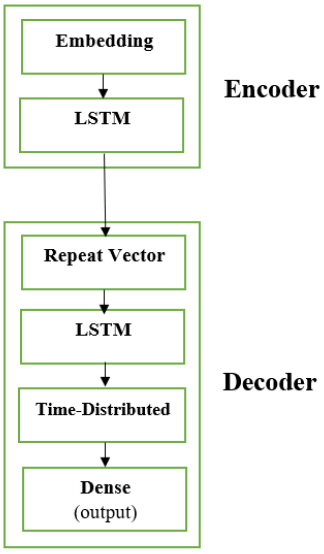
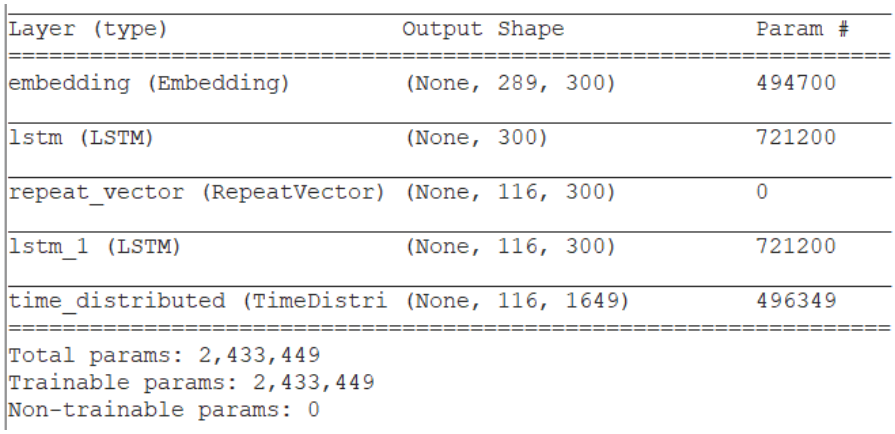


Figure 6.1 Simple Encoder-Decoder Model



Layer (type)	Output Shape	Param #
embedding (Embedding)	(None, 289, 300)	494700
lstm (LSTM)	(None, 300)	721200
repeat_vector (RepeatVector)	(None, 116, 300)	0
lstm_1 (LSTM)	(None, 116, 300)	721200
time_distributed (TimeDistri	(None, 116, 1649)	496349

Total params: 2,433,449
Trainable params: 2,433,449
Non-trainable params: 0

Figure 6.2 Simple Encoder-Decoder Shape

The model consists of a single LSTM layer for the encoder, which will output a context vector that is fed into a single decoder LSTM layer. Given an input sequence of length n, and full output sequence of length m, the context vector is fed to the decoder m times, each time creating one more word (token) in the output sequence. Each output is run through a dense layer with a softmax activation, once for each output token. Input is run through an embedding layer, which transforms each input token into a vector.

In this simple encoder-decoder all text is tokenized at a word level, meaning each word in the input and output corresponds to a single number ("token"). All input is in the form of sequences of word tokens, and output is in a one-hot-encoded format, with a vocab size equal to 1.5 times the size of the dictionary used by the TextWorld framework. The vocab size is higher than the vocab size in TextWorld to account for extra punctuation and contractions. During training, all input and output sequences are padded at the end with zeros ("null tokens"), so that all input sequences are the same length n and all output sequences are the same length m. Keeping input and output sequences at the same length makes it easier to parallelize training by using a GPU to process multiple sequences at once. In practice, n is the length of the longest input sequence (TextWorld objective) and m is the length of the longest output sequence (winning policy) observed in the training, testing, and validation sets.

As a loss function during training, the model uses categorical cross-entropy:

$$-\mathcal{LL}(\boldsymbol{w}) = -\log\left(\frac{e^{\hat{y}_p}}{\sum_{i=1}^{m} e^{\hat{y}_i}}\right)$$

Where $\hat{\boldsymbol{y}}$ is the one-hot-encoded vector of predictions for each word, and $\hat{\boldsymbol{y}}_p$ is the prediction for the expected target word. The exact accuracy function used is unknown, and is hidden within the implementation of Keras, the library used for much of this project.

6.2.1 Testing the Simple Encoder-Decoder Model

To test this simple architecture, I trained three versions of this encoder-decoder, each with variations of the input embedding layer, which turns each token in the input into a multidimensional vector. Each version was trained on a tiny dataset of 700 translations for 15 epochs, and predictions came out to be mostly gibberish. The reported results from one are shown in Figure 6.3 below.

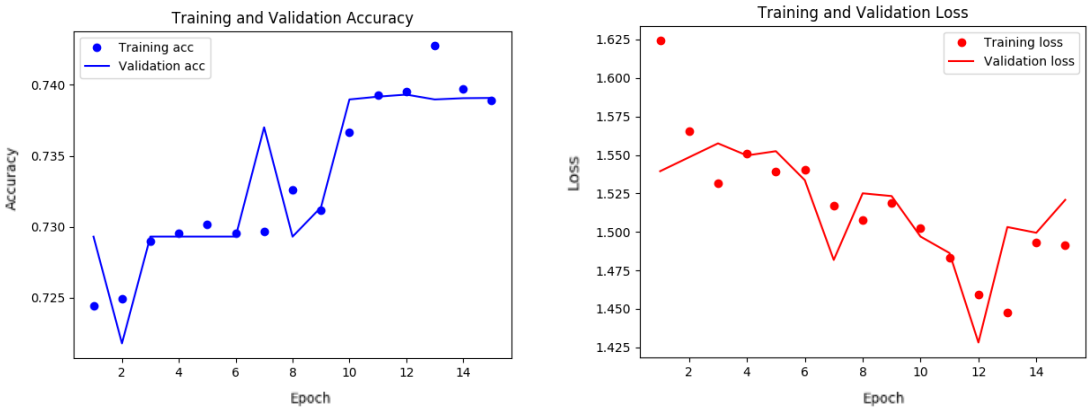


Figure 6.3 One of three tests

The predictions from these tests of the very simple model are mostly gibberish. Yet, the highest accuracies reported by these three models comes out to be around 73-74%. The reported accuracy seems much higher than it should be.

After looking more closely at what the networks were doing, this strangely high accuracy makes sense. Because all input and output to the network is padded with 0's to create sequences

of fixed length, most of the expected outputs are 0s, null word tokens. If each null token is counted towards the "accuracy" of a prediction, the reported accuracies make much more sense.

To give an example, the tokenized and zero-padded version of the sequence "go north , go south , go west" might look like "1 2 3 1 4 3 1 5 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0". Even if the network outputs "1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0," which would translate to "go", because 42/50 of the tokens in the expected sequence are 0's, in its own eyes it would still get 43/50 of the output tokens correct, even though only the first word, "go" matched anything that matters.

Because the goal of the network is to learn to get the best accuracy it can on all the input, it could learn to have a bias towards predicting as many null word tokens as possible. Another test was run to confirm this theory. In this test, the first of the three simple models was trained on the same 700-translation data for 50 epochs. At the end, the network still reported 70% accuracy, but no matter what, it would always predict the word "go", and nothing else.

When analyzing the training data, in practice, it turns out that most sequences the network encounters are much shorter than the max output sequence length. Normally, the zero-padded output length is around 115 tokens long (the length of the longest output sequence). In a second test, the model was set up to only consider a short, maximum output length of 9, and then was trained again on the same data. Afterwards the model predicted much more reasonable output, closer to the expected output when expected output was shrunken to 9 words.

To fix this problem of training with null tokens, the best model was set to use custom training weights on each expected output of the network. For each output sequence, there is a

weight of 1 (max attention) on every expected output token up to and including the first null token, and 0 (no attention) on every output after that.

Below are the results before and after adding training weights (trained for 50 epochs on a tiny dataset of 700 translations).

Before:

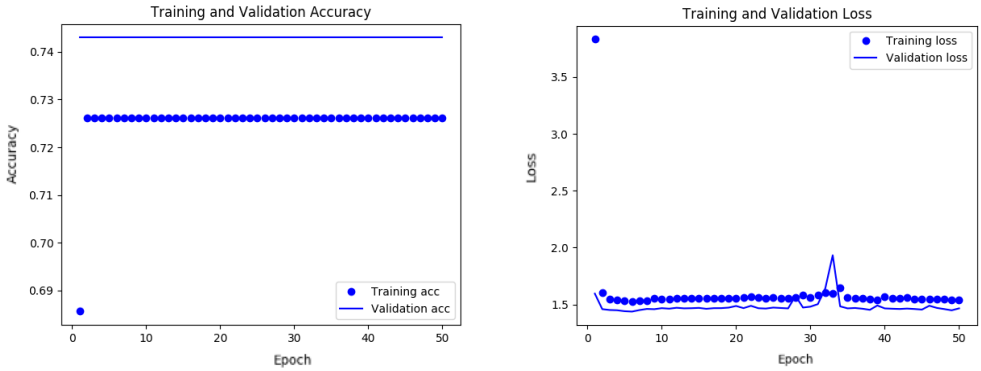


Figure 6.4.1 Simple encoder-decoder before learning weights

Test set predictions before training weights (just showing outputs):

```
_____Prediction 0_____
Expected Output: "go south , go south , take american style key from stand , unlock american style box with american style key , open american style box , take honeydew from american style box , go north , insert honeydew into "
Actual Output: ""
_____Prediction 1_____
Expected Output: "go east , open type 4 safe , take keycard from type 4 safe , go west , unlock safe with keycard , open safe , take latchkey from safe , go east , drop "
Actual Output: ""
_____Prediction 2_____
Expected Output: "go north , go east , go east , go south , go west , take pair of pants from counter , insert pair of pants into "
Actual Output: ""
```

After:

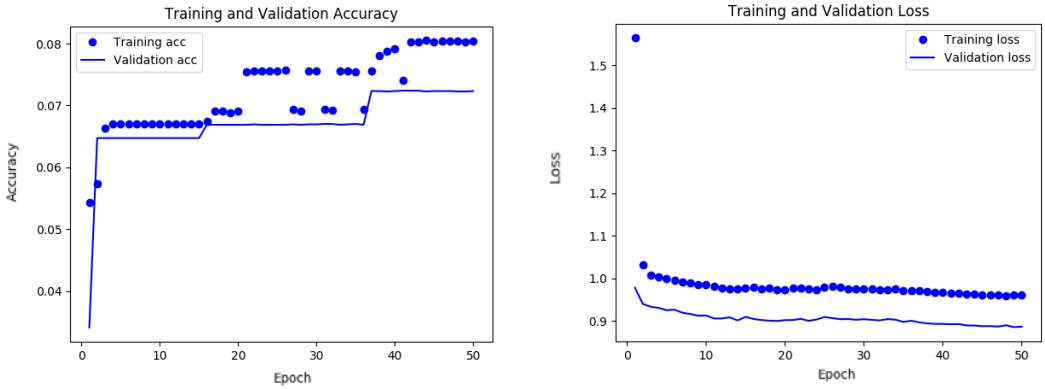


Figure 6.4.2 Simple encoder-decoder after learning weights

Test set predictions after adding training weights (just showing outputs):

```
_____Prediction 0_____
Expected Output: "go south , go west , go west , go south , go
east , go east , take staple from stand , go west , drop staple "
Actual Output: "go with , go , , , , , , , , , , , , , , , , , , ,
, , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , ,
, , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , ,
, , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , "
_____Prediction 1_____
Expected Output: "go west , go south , go south , go east , go
north , take chocolate bar from rack , drop chocolate bar "
Actual Output: "go with , go , , , , , , , , , , , , , , , , , , ,
, , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , ,
, , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , ,
, , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , "
_____Prediction 2_____
Expected Output: "go south , go west , go north , take latchkey ,
go south , go east , drop latchkey "
Actual Output: "go with , go , , , , , , , , , , , , , , , , , , ,
, , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , , ,
```

, ,
, "

After adding training weights, the reported accuracy goes down to something that looks much more fitting of the predictions that the model gave. The predictions afterward also show that even with this tiny amount of data, the model is learning to predict less null tokens, and is instead starting to gravitate towards more real word tokens like “go,” “,”, and “with.”

6.3 A Better Encoder-Decoder Model

The simple encoder-decoder architecture discussed in sections 6.1-6.2 has a lot that could be improved upon. For one, the decoder must create the entire output sequence from just the single context vector given by the encoder. With just that context given, each time it predicts an output token, it will have to guess the next token with little knowledge of earlier predicted tokens in the output sequence.

Another thing that could be improved is the lack of attention given to specific parts of the decoded input at each time step in the output sequence. Each part of the output sequence must predict from the entire decoded context vector. There is less opportunity for the network to give attention to only relevant meaning from the input for a specific token in the output.

The simple model described next is based on encoder-decoder architecture as proposed in [7] and [9], and improves upon these areas.

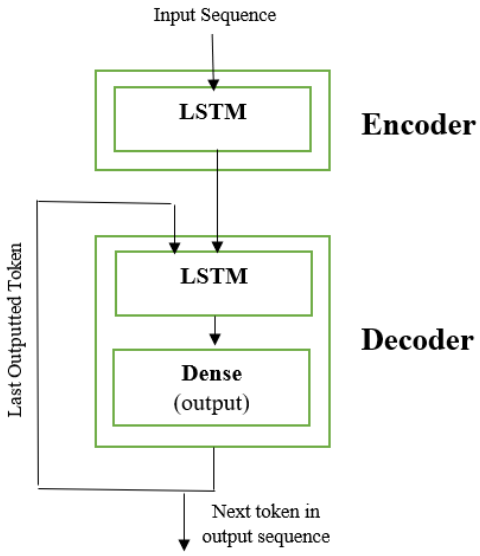


Figure 6.5 Recursive Encoder-Decoder Model

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, None, 1649)]	0	
input_2 (InputLayer)	[(None, None, 1649)]	0	
lstm (LSTM)	[(None, 300), (None,	2340000	input_1[0][0]
lstm_1 (LSTM)	[(None, None, 300),	2340000	input_2[0][0] lstm[0][1] lstm[0][2]
dense (Dense)	(None, None, 1649)	496349	lstm_1[0][0]

Figure 6.6 Recursive Encoder-Decoder Shape

In the new, recursive model described in Figure 6.5-6.6, the encoder remains about the same. It is still made up of a single LSTM layer. The decoder here now is changed to have different inputs. Instead of taking in the context vector produced by the encoder, the initial internal state of the decoder LSTM layer is set to the resulting state of the LSTM layer in the encoder. The decoder will also take in a different input. During training this input will look like the last token that the decoder model would have predicted. Output at each decoder time step is fed into a dense layer with a softmax activation. Unlike in the last decoder model, this model will output predictions for one output token at a time, instead of the entire output sequence at once.

In this simple recursive model, training and prediction are treated differently. During prediction, the decoder model will perform more like the model in Figure 3.5. At each timestep, the decoder will take in the predicted token from the previous timestep, and an internal state. For the first token in the output sequence, the decoder is passed in the final internal state of the encoder, and a special sequence start token. Then, the decoder makes its first prediction. After the first prediction, and every prediction after that, the output token of the decoder, and its final internal state, are passed recursively back in during each prediction. Prediction will stop after the decoder either outputs a null token, or the max output length is reached.

In training, teacher forcing [12] is used, and all inputs are fixed to always be the expected outputs from the previous step. Figure 3.6 shows what a training step would look like. Just like the modified version of the simple model in section 6.2, this recursive encoder-decoder is trained using zero-padded sequences and produces one-hot-encoded output sequences. Likewise, it uses training weights in order to only train on non-zero word tokens in the expected output sequence.

The loss function used for this model is once again categorical cross-entropy, with an unknown accuracy function, hidden within the implementation of Keras. The loss function and the accuracy function respect the training weights.

6.3.1 Testing the Simple Recursive Model

Here the simple 1-layer encoder 1-layer decoder model is trained for 50 epochs on 50,000 translations (50,000 different TextWorld games).

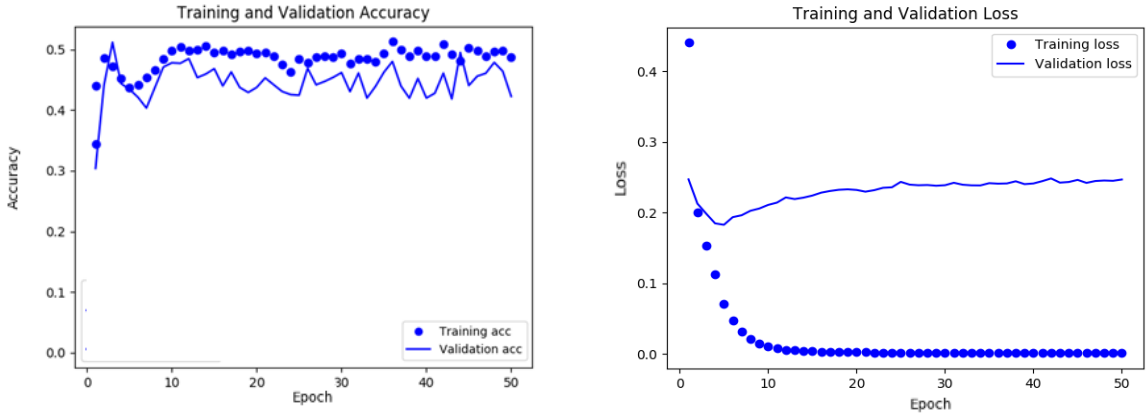


Figure 6.7 Training Simple Recursive Model With Weights

Sample test set predictions (showing output only):

```
_____Prediction 0_____

Expected Output: "go west , take cake scented keycard from
bookshelf , unlock cake scented locker with cake scented keycard
, open cake scented locker , take folder from cake scented locker
, go east , put folder on armchair "

Actual Output: "go north , take spherical type from wide , unlock
spherical entered south spherical type , open spherical entered ,
go with , take got rectangular box from locker , unlock got
rectangular passkey south got rectangular box , open got
rectangular passkey , take chest from got rectangular passkey ,
unlock because south chest , open because , go with , take c from
amazing try of , go ! , handled c "

_____Prediction 1_____

Expected Output: "go south , open door , go east , go east , take
bowl from pan , go west , drop bowl "

Actual Output: "go with , take play chest from wide , go ! ,
unlock play because south play chest , open play because , go
north , take shirt from fantastic "

_____Prediction 2_____

Expected Output: "go north , go west , go south , go south , go
east , take passkey , put passkey on shelf "
```

```
Actual Output: "go with , take chest from bedchamber , go ! ,
unlock keycard south chest , open keycard , take box from keycard
, go with , handled box "
```

It is obvious from the validation loss in Figure 6.7 that this model needs much less training before it starts overfitting on the training data. After only around 6 or 7 epochs, validation loss starts rising and keeps going, indicating strong overfitting. Based on the accuracy shown in the chart on the left, this model also appears to perform much better at its best than the very simple encoder-decoder developed before. The recursive model reports almost 50% accuracy at its best, compared to the less than 10% done by the previous simpler model. It is notable that a part of the gain in accuracy probably has to do with the increase in training data available to the new simple recursive model. It is also interesting that the training loss seems to reach zero very quickly, implying that the network has stopped learning from the training data, yet the reported training accuracy only stays around 50%. The model does not seem to be able to memorize, or learn the training data. I am not sure what to make of these results. The same issues arise in Figures 6.8-6.11 with loss approaching close to 0, but low reports of training accuracy. I am uncertain if this could be a bug in the way the loss is calculated, or perhaps in another part of the code. Ultimately, I moved on to work on a similar design to this model using an attention mechanism and a custom loss and accuracy function, which seemed to fix this issue; so this issue was never explored much further.

6.3.2 Improving the Recursive Model

In research by Wu et. al. in [8], they show that both the encoder and decoder implementations work best when they have more layers to “capture subtle irregularities in the input and output languages” [8]. However, as more layers are added, the network becomes slower and more difficult to train. Wu et al. found that in large scale translation tasks like the one

Google was working on, stacking layers worked well up to 4 stacked LSTM layers in both the encoder and decoder, barely with 6, and poorly beyond 8. Taking this into account, and trying to test performance with more layers in the recursive network, more variations to the model were made that added extra layers to the encoder and decoder in different arrangements, and compared the results. Diagrams and results can be found below.

This test compares four deeper configurations of the recursive model with stacked LSTM layers (each uses training weights and is trained on 50,000 translations):

3 Encoder Layers, 1 Decoder Layer

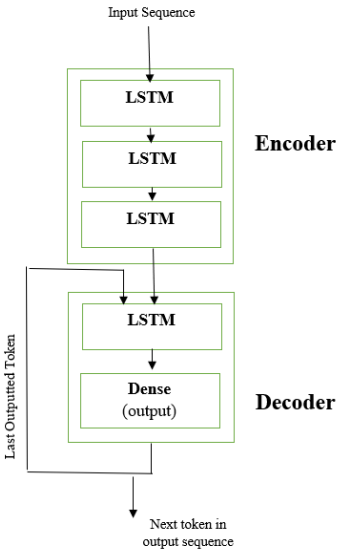


Figure 6.8.1 3-layer encoder, 1-layer decoder

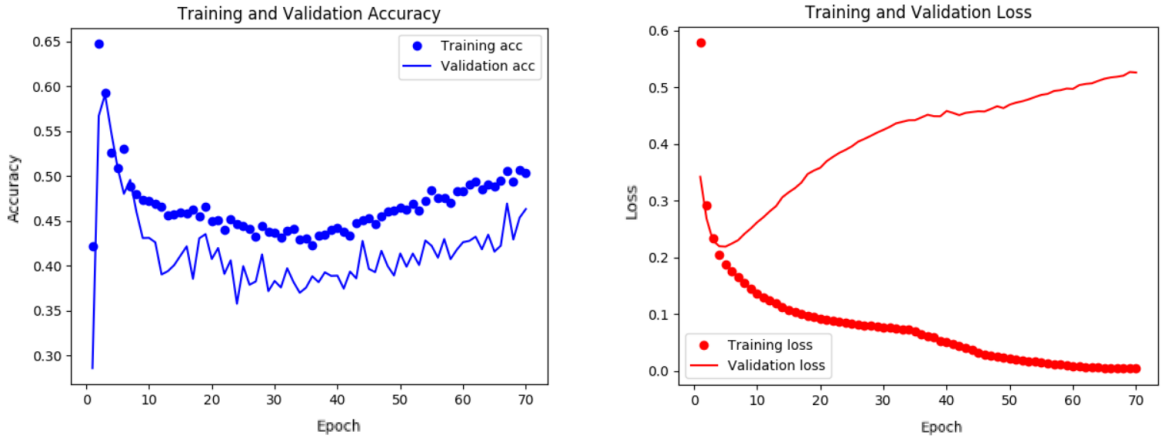


Figure 6.8.2 Training 3-layer encoder, 1-layer decoder

Sample test set predictions (output only):

```
_____Prediction 0_____

Expected Output: "go west , go north , go north , go east , go
south , take cashew from safe , eat cashew "

Actual Output: "go south , open ensure chest , take latchkey from
ensure chest , unlock good is latchkey , open good , go you ,
take broccoli from pair "

_____Prediction 1_____

Expected Output: "go south , go south , go east , go east , go
north , go west , take latchkey from formless locker "

Actual Output: "go south , open ensure chest , take latchkey from
ensure chest , unlock good is latchkey , open good , go ! , take
restroom from board "

_____Prediction 2_____

Expected Output: "go north , go west , go south , take type w
passkey from shelf , go north , go east , unlock type w chest
with type w passkey , open type w chest , take keycard from type
w chest , put keycard on desk "

Actual Output: "go north , go ! , go south , go south , go you ,
go you , take attempt from objects , go ! , play attempt up
objects "
```

1 Encoder Layers, 3 Decoder Layers

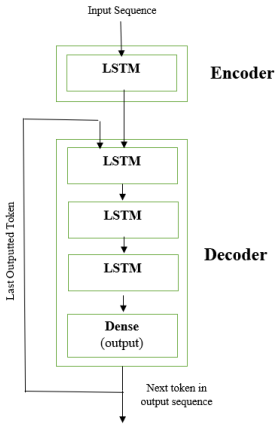


Figure 6.9.1 1-layer encoder, 3-layer decoder

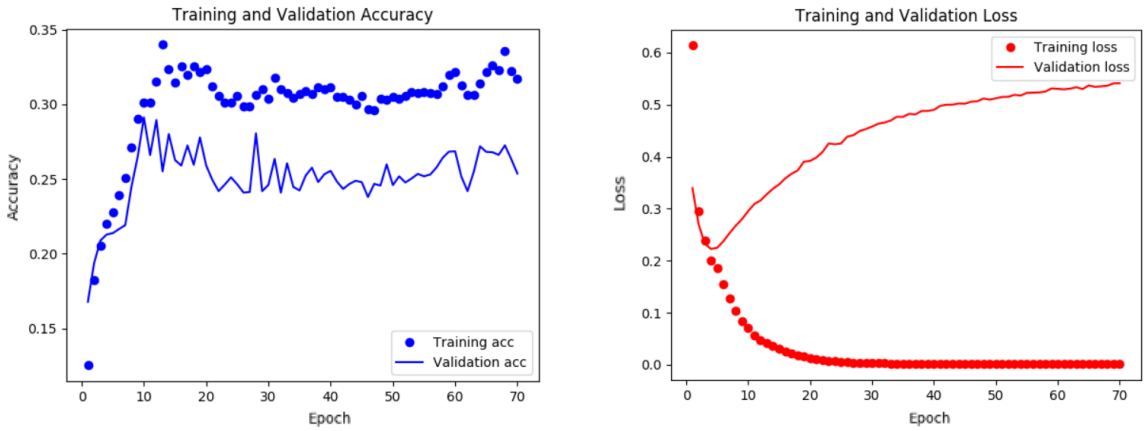


Figure 6.9.2 Training 1-layer encoder, 3-layer decoder

Sample test set predictions (output only):

```
_____Prediction 0_____

Expected Output: "go south , open rectangular locker , take
spherical key from rectangular locker , go south , go west ,
unlock spherical gate with spherical key , open spherical gate ,
go north , take cookie from mantelpiece "

Actual Output: "go take of north open board from , go go go go go
go go go go go go go go go go go go go go go go go go go go go go
go go go go go go go go go go go take rectangular north paper
-euclidean spare north take from attic mantelpiece go go go go go
go go go go go go go north board over south take from cup comic
go go go go go go go go go go go you're cup candy take , go go go
```

go go go go go go go go go go north take attic top open go take , go go "

_____Prediction 1_____

Expected Output: "go west , go north , go east , go east , go south , take latchkey , go west , unlock chest with latchkey , open chest , take candy bar from chest "

Actual Output: "go take of north take attic stop picked go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go take , go go go take rectangular assure go go take attic bear go go go go go go go go go go north take attic wide north go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go go north board board over larva go "

_____Prediction 2_____

Expected Output: "go north , take type g keycard , go south , unlock type g passageway with type g keycard , open type g passageway , go east , take shadfly "

Actual Output: "go take of attic "

2 Encoder Layers, 2 Decoder Layers

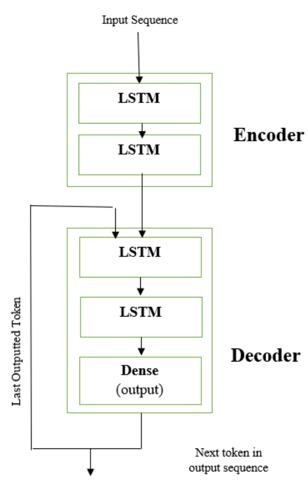


Figure 6.10.1 2-layer encoder, 2-layer decoder

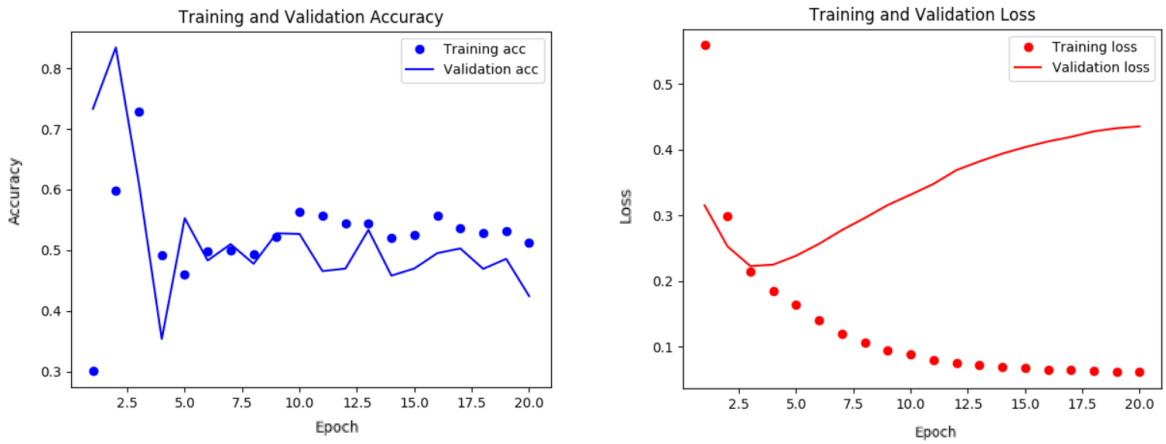


Figure 6.10.2 Training 2-layer encoder, 2-layer decoder

Sample test set predictions (output only):

```
_____Prediction 0_____

Expected Output: "go west , take type r keycard , go east ,
unlock type r safe with type r keycard , open type r safe , take
hendersons style passkey from type r safe , unlock hendersons
style safe with hendersons style passkey , open hendersons style
safe , insert keycard into hendersons style safe "

Actual Output: "go put key "

_____Prediction 1_____

Expected Output: "go north , go north , go west , go south , go
south , take scarf , put scarf on shelf "

Actual Output: "go put key , go north , go go put , unlock of of
-pit of of of of of of recliner "

_____Prediction 2_____

Expected Output: "go south , take microsoft style latchkey , go
north , unlock microsoft style box with microsoft style latchkey
, open microsoft style box , take latchkey from microsoft style
box , unlock chest with latchkey , open chest , take stick of
butter from chest , put stick of butter on stand "

Actual Output: "go go go go go go go go go go go you , over
strip k episode k "
```

3 Encoder Layers, 3 Decoder Layers

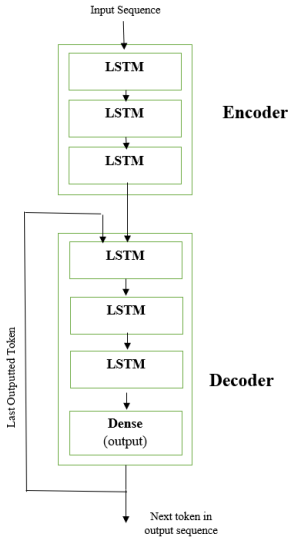


Figure 6.11.1 3-layer encoder, 3-layer decoder

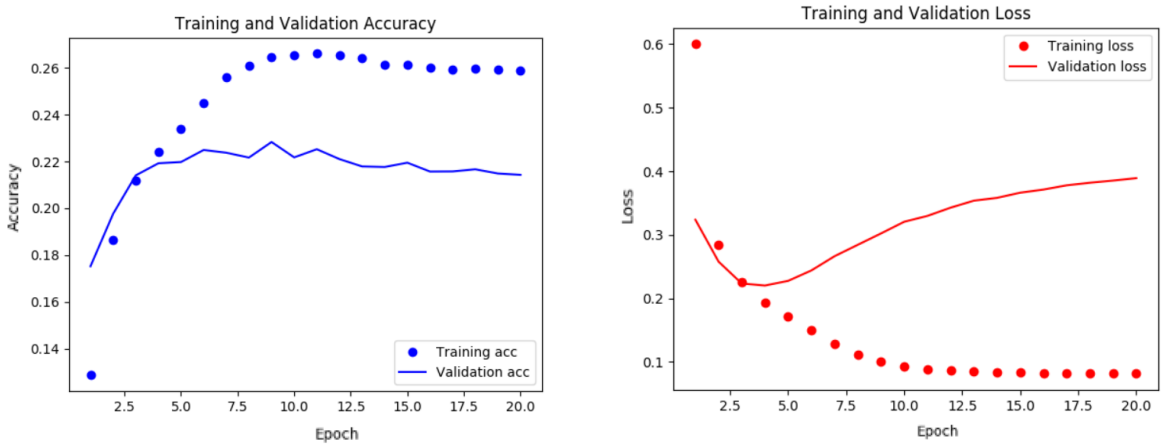


Figure 6.11.2 Training 3-layer encoder, 3-layer decoder

Sample test set predictions (output only):

```
_____Prediction 0_____

Expected Output: "go west , go west , go south , go south , go
east , go north , take passkey from mantelpiece "

Actual Output: "take from of from of from from shirt go go go go
go go go go go go go go go go go go go go go go go go go go go go
go go go go go go go go go go go go go go go go go go go go go go
go go go go go go go go go go go go go go go go go go go go go go
go go go go go go go go go go go go go go go go go go go go go go
go go go go go go go go go go go go go go go go go go go go go go
go go go go go go go go go go go go go go go go go "
```

_____Prediction 1_____

Expected Output: "take keycard from counter , unlock box with keycard , open box , take type v passkey from box , unlock type v door with type v passkey , open type v door , go west , take latchkey from bench "

Actual Output: "take from of from of from from shirt go "

_____Prediction 2_____

Expected Output: "go north , go west , go south , take keycard from case , go east , unlock locker with keycard , open locker , take pear from locker , drop pear "

Actual Output: "take from of from of from from shirt go "

Evaluating Performance of the Four Stacked Recursive Models

These four configurations of layers create surprisingly large differences in accuracy. The configuration with the most accurate predictions was without a doubt the 2-deep encoder, 2-deep decoder configuration. At its highest point, it reported almost 75% validation accuracy, before it started overfitting to the training data. Comparing the 3 encoder, 1 decoder and 1 encoder, 3 decoder configurations, it is notable that the one with more encoder layers reported more than twice as much accuracy (at its highest point) than the configuration with more decoder layers. This seems to indicate that a more complicated encoder is much more important than a more complicated decoder. This kind of result would make sense, as the detailed objective is usually

much longer, and much more varied in vocabulary and semantic structure than the output list of commands.

Seeing that the 2 encoder, 2 decoder model performed the best, one would think that the 3-layer encoder, 3-layer decoder would perform better, based off of the results from Wu et al. where up to 4 layers in each part of the model worked well [8]. However, it seems to get the lowest reported accuracy out of all the models, peaking at a little over 20% before it starts overfitting.

Looking at the example predictions on test data for the 2 encoder, 2 decoder model, the model has made some noticeable progress on previous versions. Output predicted from test data seems much closer to comma-separated commands, like in the prediction "`go put key , go north , go go put , …`".

Despite the small amounts of progress models like the 2 encoder, 2 decoder version makes, each of these four models during training have training loss reach or very closely approach 0, implying that the model is no longer learning, yet reported training accuracy only stays around 50-60% in the best of the four models at this stage. These results seem to imply that the design used for these models is not enough to make full translations from TextWorld objectives to actions.

6.4 Developing the Final Translation Model

While the 2-layered encoder, 2-layered decoder recursive model improves a lot in reported accuracy over other models, it still has much that can be improved upon. The longer input and output sequences are, the faster the predicted translations seem to break down, with complete gibberish predicted at full scale translations, as demonstrated in the sample test

predictions above. The model is not prepared to handle input and output sequences as long as the ones in these TextWorld objectives.

Attentional mechanisms, proposed in [4], have been demonstrated to get much better results on longer input sequences than encoder-decoder models without attention [10]. There are three kinds of basic attentional models that most encoder-decoder models use: Bahdanau Attention, as proposed by Bahdanau et. al. in [4], and the global attentional model and local attentional model proposed by Luong et. al. in [10]. For simplicity's sake, and to follow Google's implementation in [8], this work uses the simpler Bahdanau-based attentional model.

To help get started writing an encoder-decoder using a custom attentional model, code here was modelled off of a tutorial on neural machine translation in the TensorFlow Core docs [15]. Basing the design of the attentional model off of the Bahdanau Attention implementation in [8], this work first tests two attention-using models, one using LSTM layers as the RNN in the encoder and decoder, and one using GRU (gated recurrent unit) layers.

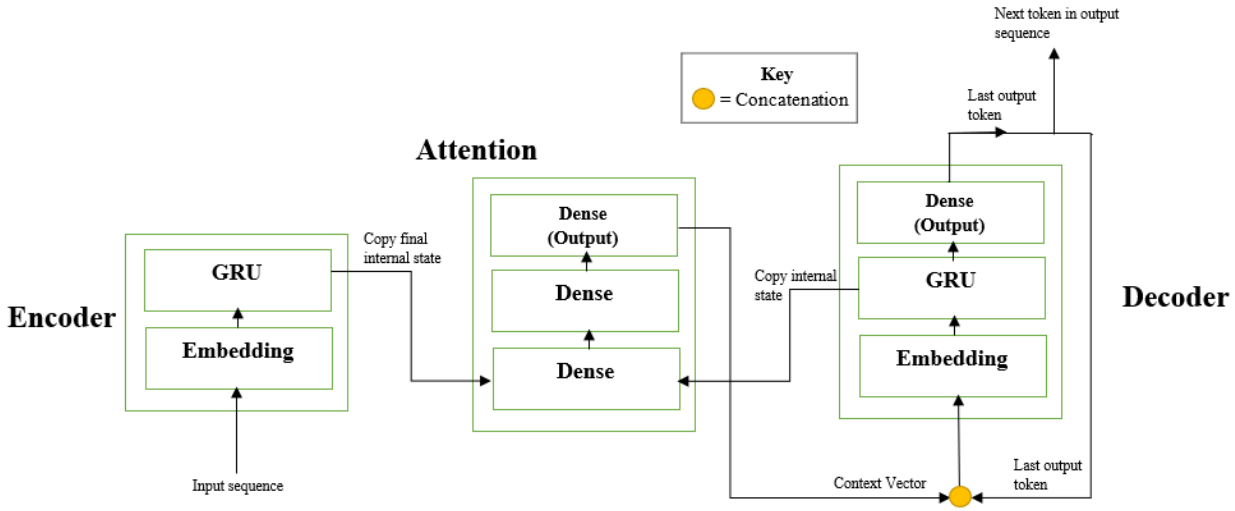


Figure 6.12 1-layer GRU attention model

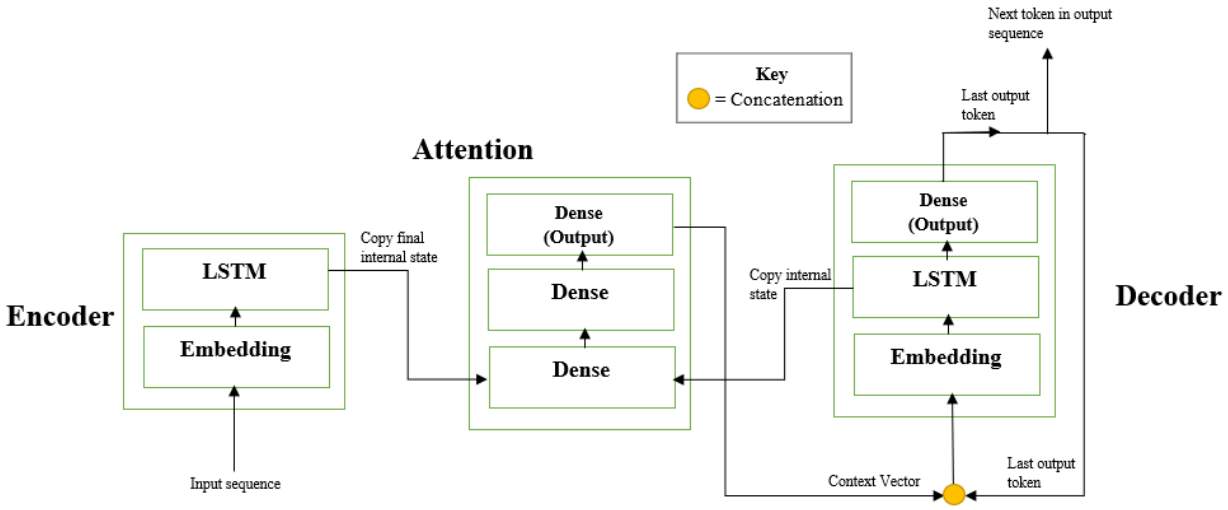


Figure 6.13 1-layer LSTM attention model

Both models are the same, with the only difference being the choice in RNN used in the encoder and decoder. Like models defined in earlier sections, this design trains with zero-padded input and output sequences, and produces one-hot-encoded output.

First, the entire input sequence is processed by the encoder, and at the end, the final hidden state is copied from the encoder and run through the attention model, which will produce a context vector from the given hidden state, like in [4]. For the first prediction in the decoder, this context vector is concatenated with a special '<start>' token, and fed into the decoder. For each prediction after that, the next context vector is produced using the RNN hidden state from the previous decoder step, and is concatenated with the previous predicted token. During training, the system uses teacher forcing, and concatenates the previous expected output token with the context vector instead of the previous predicted token.

Unlike previous models, this design uses a custom implementation for the loss and accuracy functions. As the loss function each model uses a mask applied to sparse categorical cross-entropy. The mask prevents null-tokens in the output sequence from being included in the loss, so they will not affect training. The accuracy function for both models uses a similar mask.

Where $\boldsymbol{y}$ is a set of expected tokens in a batch (including null tokens), $\hat{\boldsymbol{y}}$ is a set of predicted tokens in a batch, and $\boldsymbol{u}$ is the mask, the accuracy function for one timestep in a batch can be defined as

$$total = \text{countNonzero}(\boldsymbol{y})$$

$$acc = \frac{total - \text{countNonzero}(\boldsymbol{u} \odot (\boldsymbol{y} - \hat{\boldsymbol{y}}))}{total}$$

Here $\odot$ is used to indicate an elementwise multiplication. This accuracy function will score each predicted winning policy by how many of the first m tokens match the tokens in the expected winning policy, where m is the number of tokens in the expected winning policy.

6.4.1 Comparing LSTM and GRU-based Models

To compare performance of the two models on TextWorld data, each was trained on 100k translations over 20 epochs.

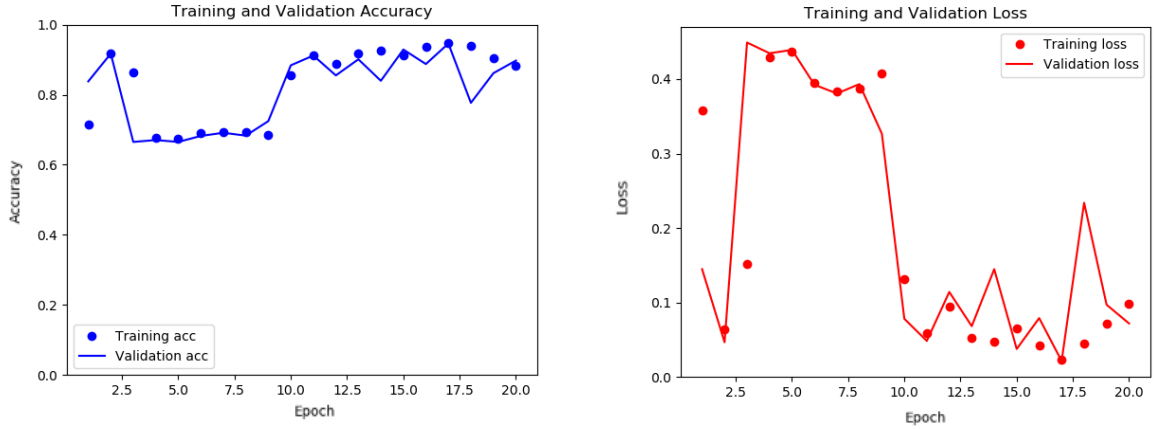


Figure 6.14 Training GRU Bahdanau Attention Model

Sample GRU test set predictions (output only):

```
_____Prediction 0_____
Expected output: "<start> go west , go west , go south , open
locker , take textworld limited edition key from locker , go east
```

, lock textworld limited edition hatch with textworld limited edition key <end> "

Actual output: "<start> go west , go west , go south , open locker , take textworld limited edition key from locker , go east , lock textworld limited edition hatch with textworld limited edition key <end> "

_____Prediction 1_____

Expected output: "<start> take canadian latchkey from board , unlock canadian chest with canadian latchkey , open canadian chest , take latchkey from canadian chest , unlock locker with latchkey , open locker , insert passkey into locker <end>"

Actual output: "<start> take canadian latchkey from board , unlock canadian chest with canadian latchkey , open canadian chest , take latchkey from canadian chest , unlock locker with latchkey , open locker , take latchkey from canadian chest , unlock locker with latchkey , open locker , take latchkey from canadian chest , unlock locker with latchkey , open locker , take latchkey from canadian chest , unlock locker with latchkey , open locker , take latchkey from canadian chest , unlock locker with latchkey , open locker , take latchkey from "

_____Prediction 2_____

Expected output: <start> go south , take key , go north , unlock safe with key , open safe , take microsoft latchkey from safe , unlock microsoft passageway with microsoft latchkey , open microsoft passageway , go east , take legume from plate , eat legume <end> "

Actual output: <start> go south , take key , go north , unlock safe with key , open safe , take microsoft latchkey from safe , unlock microsoft passageway with microsoft latchkey , open microsoft passageway , go east , take legume from plate , eat legume <end> "

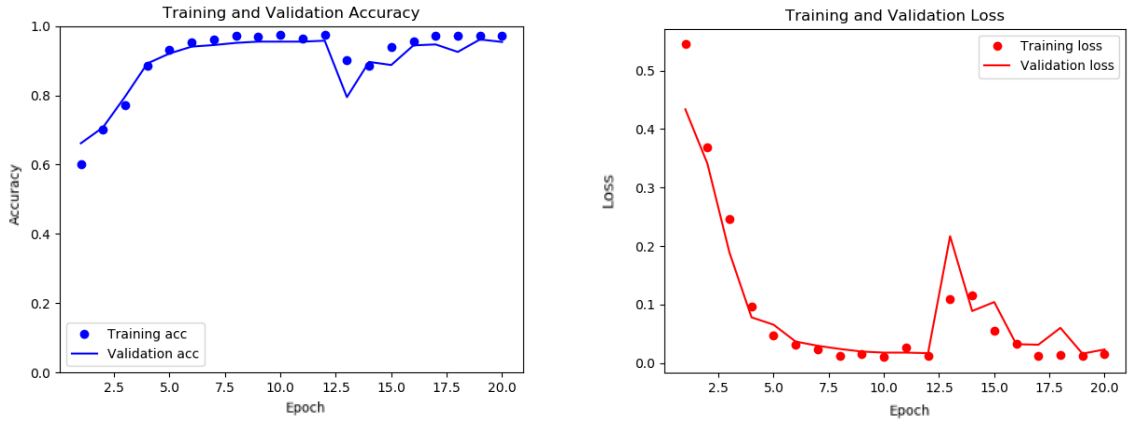


Figure 6.15 Training LSTM Bahdanau Attention Model

Sample LSTM test set predictions (output only):

```
_____Prediction 0_____

Expected output: "<start> go west , go west , go south , open
locker , take textworld limited edition key from locker , go east
, lock textworld limited edition hatch with textworld limited
edition key <end> "

Actual output: "<start> go west , go west , go south , open
locker , take textworld limited edition key from locker , go east
, lock textworld limited edition hatch with textworld limited
edition key <end> "

_____Prediction 1_____

Expected output: "<start> take canadian latchkey from board ,
unlock canadian chest with canadian latchkey , open canadian
chest , take latchkey from canadian chest , unlock locker with
latchkey , open locker , insert passkey into locker <end> "

Actual output: "<start> take canadian latchkey from board ,
unlock canadian chest with canadian latchkey , open canadian
chest , take latchkey from canadian chest , unlock locker with
latchkey , open locker , insert passkey into locker <end> "

_____Prediction 2_____

Expected output: "<start> go south , take key , go north , unlock
safe with key , open safe , take microsoft latchkey from safe ,
unlock microsoft passageway with microsoft latchkey , open
```

```
microsoft passageway , go east , take legume from plate , eat
legume <end> "
Actual output: "<start> go south , take key , go north , unlock
safe with key , open safe , take microsoft latchkey from safe ,
unlock microsoft passageway with microsoft latchkey , open
microsoft passageway , go east , take legume from plate , eat
legume <end> "
```

In both predictions and reported accuracy, both models do exceptionally well compared to the previous models. On average, both models seem to get almost every prediction correct, and only lose accuracy on some predictions. Between the two models, both seem to perform about as well as the other on average, with the LSTM performing slightly better. Between the two, the LSTM also took much less time to train for 20 epochs. Continuing forward, this work uses the LSTM to help with shorter training time.

6.4.2 Comparing Model Depths

To compare different depths in the encoder and decoder this work tests three models with stacked LSTM layers. One with a 2-layer encoder, 2-layer decoder, another with a 3-layer encoder, 3-layer decoder, and another with a 4-layer encoder, 4-layer decoder. In each of these tests, the models were trained on 100k translations. It took about 50 minutes per epoch to train each model, totaling at about 44 hours over 55 epochs per model.

2-layer encoder, 2-layer decoder

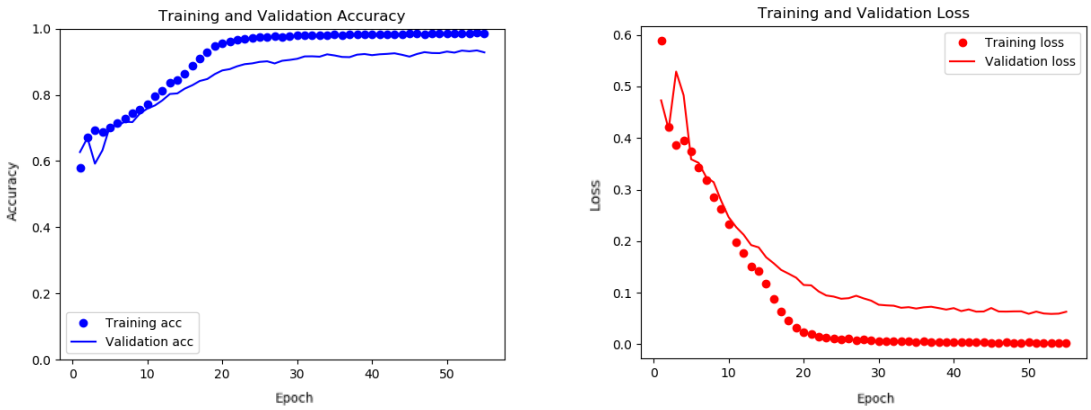


Figure 6.16 Training 2,2 depth attention-based model

Sample test set predictions (output only):

```
_____Prediction 0_____

Expected output: "<start> go west , go west , go south , open
locker , take textworld limited edition key from locker , go east
, lock textworld limited edition hatch with textworld limited
edition key <end> "

Actual output: "<start> go west , go west , go south , open
locker , take textworld limited edition key from locker , go east
, lock textworld limited edition hatch with textworld limited
edition key <end> "

_____Prediction 1_____

Expected output: "<start> take canadian latchkey from board ,
unlock canadian chest with canadian latchkey , open canadian
chest , take latchkey from canadian chest , unlock locker with
latchkey , open locker , insert passkey into locker <end> "

Actual output: "<start> take canadian latchkey from board ,
unlock canadian chest with canadian latchkey , open canadian
chest , take latchkey from canadian chest , unlock locker with
latchkey , open locker , take passkey from locker <end> "

_____Prediction 2_____

Expected output: "<start> go south , take key , go north , unlock
safe with key , open safe , take microsoft latchkey from safe ,
unlock microsoft passageway with microsoft latchkey , open
```

microsoft passageway , go east , take legume from plate , eat legume <end> "

Actual output: "<start> go south , take key , go north , unlock safe with key , open box , take microsoft latchkey from safe , unlock microsoft passageway with microsoft latchkey , open microsoft passageway , go east , take legume from stand , eat legume <end> "

3-layer encoder, 3-layer decoder

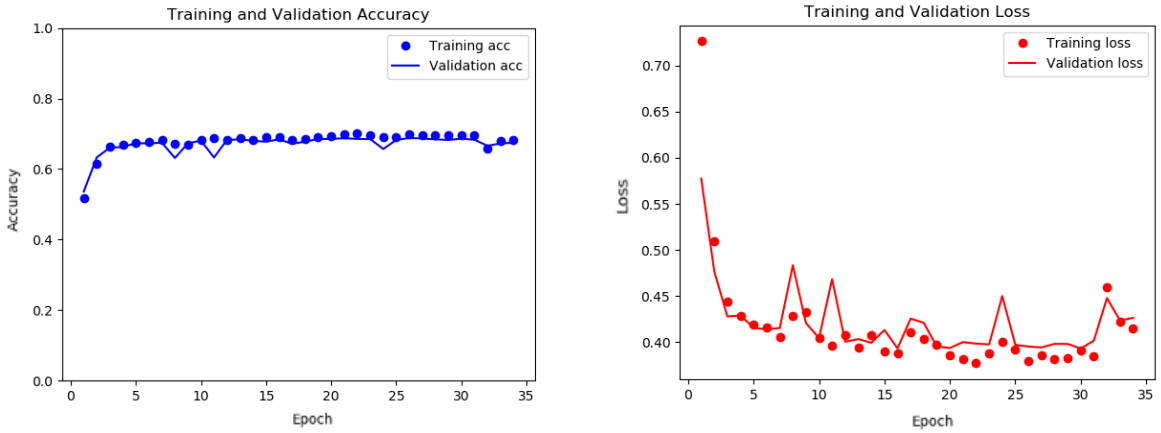


Figure 6.17 Training 3,3 depth attention-based model

Sample test set predictions (output only):

_____Prediction 0_____

Expected output: "<start> go west , go west , go south , open locker , take textworld limited edition key from locker , go east , lock textworld limited edition hatch with textworld limited edition key <end> "

Actual output: <start> go west , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , "

_____Prediction 1_____

Expected output: "<start> take canadian latchkey from board , unlock canadian chest with canadian latchkey , open canadian

```
chest , take latchkey from canadian chest , unlock locker with
latchkey , open locker , insert passkey into locker <end> "

Actual output: "<start> go west , go south , go east , go south ,
go east , go south , go east , go south , go east , go south , go
east , go south , go east , go south , go east , go south , go
east , go south , go east , go south , go east , go south , go
east , go south , go east , go south , go east , go south , go
east , go south , go east , go south , go east , go south , "

_____Prediction 2_____

Expected output: "<start> go south , take key , go north , unlock
safe with key , open safe , take microsoft latchkey from safe ,
unlock microsoft passageway with microsoft latchkey , open
microsoft passageway , go east , take legume from plate , eat
legume <end>

Actual output: "<start> go west , go south , go east , go south ,
go east , go south , go east , go south , go east , go south , go
east , go south , go east , go south , go east , go south , go
east , go south , go east , go south , go east , go south , go
east , go south , go east , go south , go east , go south , go
east , go south , go east , go south , go east , go south , "
```

4-layer encoder, 4-layer decoder

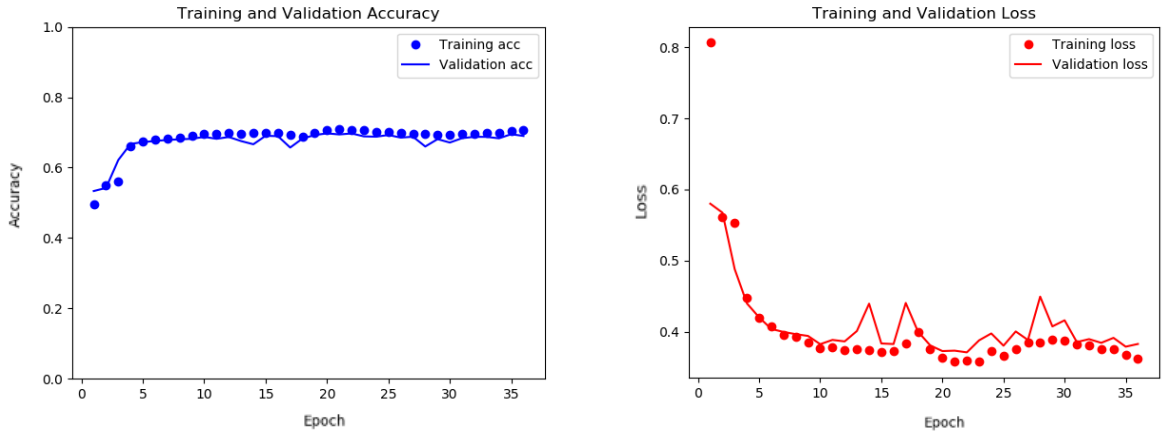


Figure 6.18 Training 4,4 depth attention-based model

Sample test set predictions (output only):

```
_____Prediction 0_____
```

Expected output: "<start> go west , go west , go south , open locker , take textworld limited edition key from locker , go east , lock textworld limited edition hatch with textworld limited edition key <end> "

Actual output: "<start> go north , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , "

_____Prediction 1_____

Expected output: "<start> take canadian latchkey from board , unlock canadian chest with canadian latchkey , open canadian chest , take latchkey from canadian chest , unlock locker with latchkey , open locker , insert passkey into locker <end> "

Actual output: "<start> go north , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , "

_____Prediction 2_____

Expected output: "<start> go south , take key , go north , unlock safe with key , open safe , take microsoft latchkey from safe , unlock microsoft passageway with microsoft latchkey , open microsoft passageway , go east , take legume from plate , eat legume <end> "

Actual output: "<start> go north , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , go south , go east , "

Out of the three configurations, the 2-layer encoder, 2-layer decoder version gets by far the best results when trained on 100k translations. During training, it reports as high as 98% accuracy and 92% validation accuracy, and gets nearly all predictions on the test set completely correct.

6.4.3 The Final Model

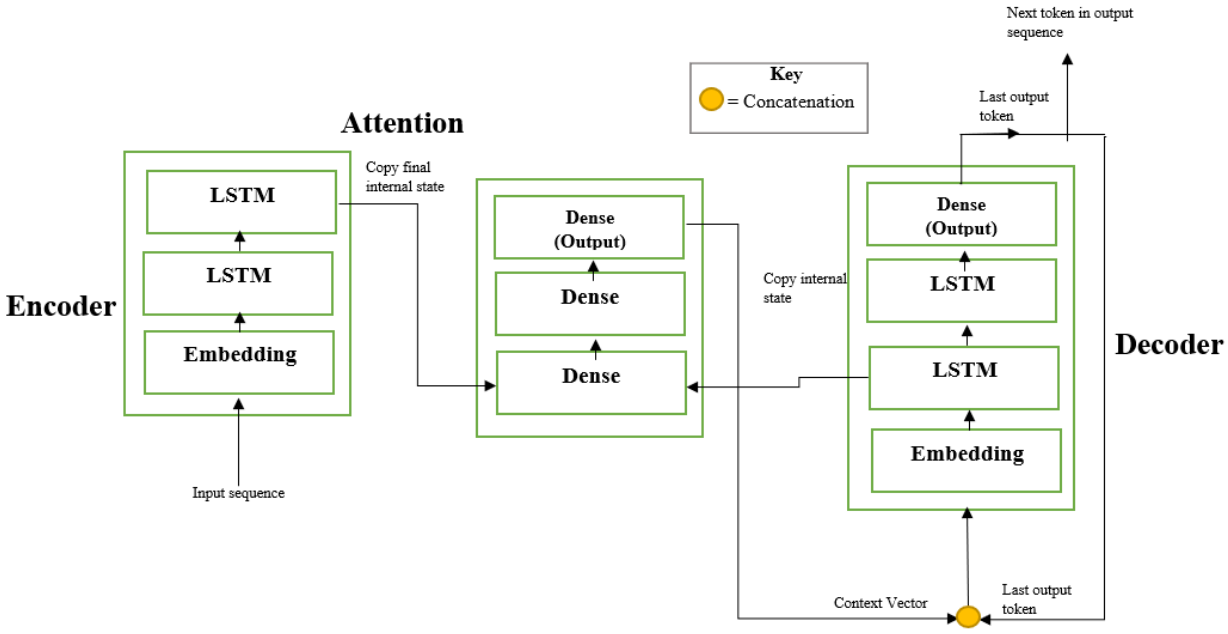


Figure 6.19 Final Model

As the final model, the 2-layer encoder, 2-layer decoder version of the model with attention is used, since it got by far the best performance on 100k translations.

6.5 Final Model Performance

This work tests the performance of the final model by training it on 500k translations until it begins to overfit. This test took about 8 hours of training time per epoch, for a total of 40 hours. The results are below.

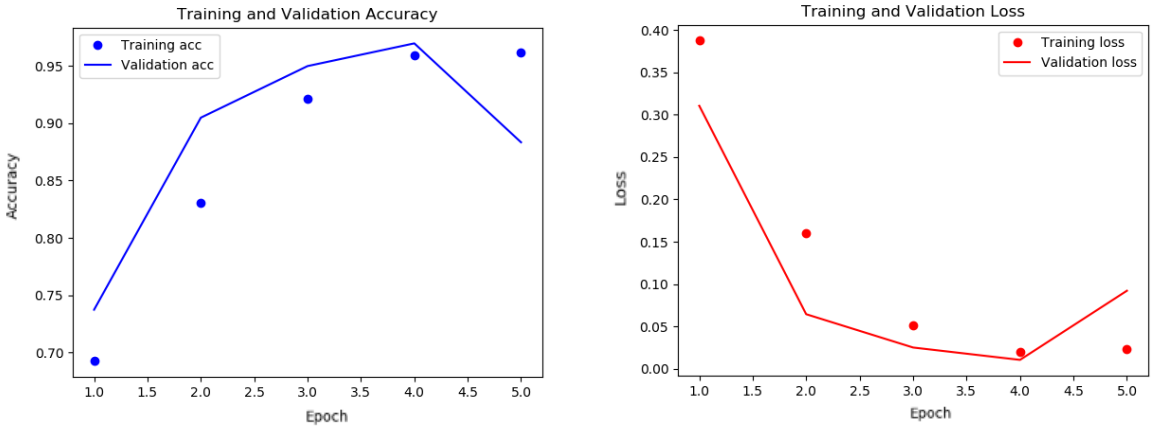


Figure 6.20 Training final model

Test set accuracy: 95%

Sample test set predictions (output only):

```
_____Prediction 0_____

Expected output: "<start> go west , go west , go south , open
locker , take textworld limited edition key from locker , go east
, lock textworld limited edition hatch with textworld limited
edition key <end> "

Actual output: "<start> go west , go west , go south , open
locker , take textworld limited edition key from locker , go east
, lock textworld limited edition hatch with textworld limited
edition key <end> "

_____Prediction 1_____

Expected output: "<start> take canadian latchkey from board ,
unlock canadian chest with canadian latchkey , open canadian
chest , take latchkey from canadian chest , unlock locker with
latchkey , open locker , insert passkey into locker <end> "

Actual output: "<start> take canadian latchkey from board ,
unlock canadian chest with canadian latchkey , open canadian
chest , take latchkey from canadian chest , unlock locker with
latchkey , open locker , take passkey from locker <end> "

_____Prediction 2_____

Expected output: <"start> go south , take key , go north , unlock
safe with key , open safe , take microsoft latchkey from safe ,
```

```
unlock microsoft passageway with microsoft latchkey , open
microsoft passageway , go east , take legume from plate , eat
legume <end> "
Actual output: "<start> go south , take key , go north , unlock
safe with key , open box , take microsoft latchkey from safe ,
unlock microsoft passageway with microsoft latchkey , open
microsoft passageway , go east , take legume from stand , eat
legume <end> "
```

Trained on 500k translations, the final model appears to have a greatly improved validation accuracy after only 4 epochs. At its best, the model reaches 95% training accuracy, 96% validation accuracy, and 95% accuracy on the test set. These results, used to play a TextWorld game, would almost always beat the game, perfectly following the instructions in the objective.

7 Conclusions

In framing the transformation of a TextWorld detailed objective to a list of commands as a machine translation problem, this endeavor found very promising results. This work finds that neural machine translation architecture turns out to work well in interpreting a long set of instructions written in natural language. Using an encoder-decoder model with an attention mechanism, and multiple layers in the encoder and decoder each, it is possible to get at least 95% accuracy on translations from an objective to a respective list of actions. In the development of the final translation model described in this paper, adding an attentional mechanism improved predictions greatly. Before adding an attentional mechanism, the best recursive model could only predict slightly structured gibberish. The simplest model with an attention mechanism predicted most sentences perfectly. Also, compared to earlier recursive models, adding attention almost doubled reported accuracy. It is likely that the strength of adding an attention mechanism comes in its ability to help ignore extra words and sentences in the objective, and generally that it could help process the very long objective text in a way that the decoder can manage.

In the final translation model, it is interesting that increasing the number of LSTM layers in the encoder and decoder models only improved performance up to 2 layers. This contrasts with results by Wu et. al [8], where they were able to get good performance up to 4 stacked LSTM layers. This decline in performance after two LSTM layers may have to do with context from the attention model vanishing in lower layers.

Apart from the problem of translation, this work also explored training a supervised learning system to play TextWorld by following the example of the shortest path. In this problem, this project found searching for this perfect shortest path to be infeasible. Using a pure iterative deepening search, finding the shortest path in a game with as few as 5 rooms and 30

objects can have a time complexity of $O(40^d)$, increasing exponentially with the depth of the search. For an oracle that would need to run hundreds of thousands of searches, this search would not be fast enough.

8 Future Research

While the final model developed in this project gets almost perfect accuracy on most translations, it does not perfectly translate TextWorld game objectives with 100% accuracy yet. It seems likely that getting 100% accuracy, or getting 99.9% accuracy would be possible with an NMT system.

One possible line of research to improve this system would be to figure out a way to increase the depth of the encoder and decoder networks, like in [8]. It is an open question of why translation breaks down after only 2 layers in the encoder and 2 layers in the decoder. Based on research in language translation by Wu et. al. [8], their network seemed to work well up to 4 layers in the encoder and decoder each before it needed special modification through what they call "residual connections" between the recurrent layers.

Along with exploring more depth, another modification to be explored in this network is the use of a beam search algorithm for figuring out the best combination of output tokens [8][11][16]. In the final model for this project, the output sequence is chosen using a greedy strategy that picks the most likely token marked by the decoder at each step. A beam search would keep track of the k mostly likely tokens at each step, and try to make further predictions off of each of those tokens. If a predicted sequence of length l is represented by probabilities of each token, $\boldsymbol{p}_{\hat{y}}$, at the end, the sequence with the highest probability $P(\boldsymbol{p}_{\hat{y}}) = \prod \boldsymbol{p}_{\hat{y}}$ is picked. Beam search has the potential of choosing a much more likely output sequence at the cost of computing time.

Bibliography

[1] M.-A. Côté, Á. Kádár, X. Yuan, B. Kybartas, E. Fine, J. Moore, M. Hausknecht, L. E. Asri, M. Adada, W. Tay, and A. Trischler. TextWorld: A Learning Environment for Text-based Games. *arXiv:1806.11532*, 2018. URL https://arxiv.org/pdf/1806.11532.pdf. PDF.

[2] S. Ross, G.J. Gordon, J.A. Bagnell. A Reduction of Imitation Learning and Structured Prediction to No-Regret Online Learning. *arXiv:* 1011.0686, 2011. URL https://arxiv.org/pdf/1011.0686.pdf. PDF.

[3] P. Ammanabrolu, M. O. Riedl. Playing Text-Adventure Games with Graph-Based Deep Reinforcement Learning. arXiv:1812.01628, 2018. URL https://arxiv.org/pdf/1812.01628.pdf. PDF.

[4] D. Bahdanau, K. Cho, Y. Bengio. Neural Machine Translation by Joinly Learning to Align and Translate. arXiv:1409.0473, 2014. URL https://arxiv.org/pdf/1409.0473.pdf. PDF.

[5] J. Gao, M. Galley, L. Li. Neural Approaches to Conversational AI. 2018.

[6] Hochreiter, Sepp & Schmidhuber, Jürgen. Long Short-term Memory. Neural computation. 1735-80, 1997. URL 10.1162/neco.1997.9.8.1735.

[7] K. Cho, B. van Merrienboer, V. Gulcehre, D. Bahdanau, Y Bengio. On the Properties of Neural Machine Translation: Encoder-Decoder Approaches. arXiv: 1409.1259, 2014. URL https://arxiv.org/pdf/1409.1259.pdf. PDF.

[8] Y. Wu, M. Schuster, Z. Chen, Q.V. Le, M. Norouzi. Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. arXiv: 1609.08144, 2016. URL https://arxiv.org/pdf/1609.08144.pdf. PDF.

[9] K. Cho, B. van Merrienboer, V. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, Y. Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. arXiv:1406.1078, 2014. URL https://arxiv.org/pdf/1406.1078.pdf. PDF.

[10] M. Luong, H. Pham, C. D. Manning. Effective Approaches to Attention-based Neural Machine Translation. arXiv: 1508.04025, 2015. URL https://arxiv.org/pdf/1508.04025.pdf. PDF.

[11] M. Johnson, M. Schuster, Q. Le, M. Krikun, Y. Wu, Z. Chen, N. Thorat, F. Viegas, M. Wattenburg, G. Corrado, M. Hughes, J. Dean. Google's Multilingual Neural Machine Translation System: Enabling Zero-Shot Translation. arXiv: 1611.04558, 2017. URL https://arxiv.org/pdf/1611.04558.pdf. PDF.

[12] R. Williams and D. Zipser. A Learning Algorithm for Continually Running Fully Recurrent Neural Networks. MIT Press, 1989. *Neural Computation*, Vol 1, Issue 2. P.270-280.

[13] N. Sprague. Gradient Descent Exercises. 2020. URL https://w3.cs.jmu.edu/spragunr/machine_learning/activities/gradient_descent/linear_regression.pdf. PDF. Reproduced with permission.

[14] N. Sprague. Multi-Layer Neural Networks. 2020. URL https://w3.cs.jmu.edu/spragunr/machine_learning/lectures/mlp/mlp.pdf. PDF. Reproduced with permission.

[15] TensorFlow. Neural machine translation with attention. URL https://www.tensorflow.org/tutorials/text/nmt_with_attention.

[16] M. Freitag, Y. Al-Onaizan. Beam Search Strategies for Neural Machine Translation. arXiv: 1702.01806, 2017. URL https://arxiv.org/pdf/1702.01806.pdf. PDF.

[17] Colah. Understanding LSTM Networks. github.io, 2015. URL https://colah.github.io/posts/2015-08-Understanding-LSTMs/img/LSTM3-chain.png. PNG.

[18] Sequence to sequence example in Keras (character-level). Keras documentation, 2019. URL https://keras.io/examples/lstm_seq2seq/.

[19] K. Narasimhan, T. Kulkarni, R. Barzilay. Language Understanding for Text-based Games Using Deep Reinforcement Learning. In Conference on Empirical Methods in Natural Language Processing (EMNLP), 2015.

[20] R. Y. Tao, M. Côté. X. Yuan, L. Asri. Towards Solving Text-based Games by producing adaptive action spaces. arXiv: 1812.00855v1, 2018. URL https://arxiv.org/pdf/1812.00855.pdf. PDF.

[21] M. Artetxe, G. Labaka, E. Agirre. Unsupervised Neural Machine Translation. arXiv: 1710.11041, 2017. URL https://arxiv.org/pdf/1710.11041.pdf. URL.