DEEP NEURAL NETWORKS FOR SPOKEN DIALOG SYSTEMS

CALLUM MAIN
H00147950

21 August 2014

COMPUTER SCIENCE
SCHOOL OF MATHEMATICAL AND COMPUTER SCIENCES

DISSERTATION SUBMITTED AS PART OF THE REQUIREMENTS FOR THE AWARD OF THE DEGREE OF MSc IN ARTIFICIAL INTELLIGENCE
Statement of Non-Plagiarism

DECLARATION

I, Callum Main, confirm that this work submitted for assessment is my own and is expressed in my own words. Any uses made within it of the works of other authors in any form (e.g., ideas, equations, figures, text, tables, programs) are properly acknowledged at any point of their use. A list of the references employed is included.

Signed:

Date: 21/8/2014
Acknowledgement

I want to thank my supervisor Dr. Verena Rieser for all of her support and guidance as well as giving me opportunities to present my work and gain more experience. Additionally I want to thank her for helping me discover a field that I am very interested in and passionate about.

I want to also thank my co-supervisor Dr. Zhuoran Wang for all of his help with the technical details that I did not have enough experience with to understand fully. I also want to thank him for helping me to understand some of mathematical details that underlie the project.
Abstract

Deep learning, particularly deep neural networks, has revolutionized automatic speech recognition (ASR) and also has shown first promising results in belief state estimation in spoken dialogue systems [Henderson et al., 2013].

In dialog systems, ”state tracking” – sometimes also called ”belief tracking” – refers to accurately estimating the user’s goal as a dialog progresses. Accurate state tracking is desirable because it provides robustness to errors in speech recognition, and helps reduce ambiguity inherent in language within a temporal process like dialogue.

This work extends previous research by Henderson et al. [2013] by extending it to use more the more modern stacked Restricted Boltzmann Machine with pretraining model. We train and evaluate our method on the Dialog State Tracking Challenge (DSTC1) data. We experiment with parameter initializations using pre-training techniques on large-scale unlabelled data, and compare the resulting models with Henderson et al.’s original method, as well as the other models entered in the DSTC. An initial version of this approach was presented at the Edinburgh Deep Learning Workshop on May 6, 2014.
1 Introduction

1.1 Motivation

One of the major issues in Spoken Dialog Systems (SDS) is that the automatic speech recognition (ASR) and spoken language understanding (SLU) components can be error prone resulting in poor overall performance. This necessitates the dialog state tracking component of SDSs as it provides robustness to some of the errors that can be introduced in the ASR and SLU components of the SDS. Additionally recent advances of Deep Learning particularly in the area of ASR has motivated the investigation of deep learning in the area of dialog state tracking.

Henderson et al. [2013] have shown some promising initial results in the area of deep learning for dialog state tracking however they have not used some more recent and more sophisticated deep learning models. The question is now do these more recent developments in deep learning give us even better performance than the basic initial model. We will investigate the use of stacked Restricted Boltzmann Machines (RBMs) with layerwise pretraining for dialog state tracking and compare it with the other models entered in the Dialog State Tracking Challenge and in particular the model by Henderson et al. [2013].

1.2 Objectives

The main goal of the project is to extend the previous work of Henderson to be able to use the more recent stacked RBM with pretraining model. We also want to compare this model to other similar models to see how this newer Deep Learning model compares to other models and as a result gain some insight into the potential for Deep Learning in dialog state tracking.

Additionally I have some personal goals, first to gain an understanding of the theory that underpins deep learning and the potential of deep learning in other fields outside of SDSs. Second to gain an understanding of SDSs and in particular dialog state tracking,
this is particularly important with the rise of the various personal digital assistants such as Siri, Google Now, and Cortana. Finally to gain more experience with the Python programming language, python is a desirable language in the area of machine learning and several machine learning toolkits have been developed for python. Prior to starting this project I only had a basic knowledge of Python as I had only really used it in one or two courses.
2 Literature Review

2.1 Deep Learning and Deep Neural Networks

2.1.1 Motivation

In over half a decade of research, there have been many advances in the creation of intelligent machines or artificial intelligence and due to the rise of “big data”, in recent years, computers now have the ability to access vast amounts of information about the world. However it is a monumental task to be able to allow computers to be able to use this information to tackle hard problems such as natural language processing or computer vision. Researchers have developed various learning algorithms and techniques in order to convert the data they have into a representation that is understandable to machines so that this knowledge can be used for tasks, most notably understanding things such as speech or images. However, according to Bengio [2009], the current algorithms are very limited in terms of their capability, they are unable to “understand scenes and describe them in natural language ... except in very limited settings”. This shows that there is a clear need for newer learning algorithms which can encapsulate more of the information required to enable computers to be able to do more complicated tasks such as understand images or speech. This is where deep architecture learning comes into play, the aim is to learn abstractions from the data going from low level features to higher level concepts that are formed through the composition of these low level features. The main issue now becomes how do we then train these architectures such that it can learn the features and resulting abstractions without the need for human intervention.

In addition, there are some more theoretical motivations as to why developing learning algorithms for deep architectures is important. According to Bengio [2009], it is sometimes the case that a function cannot be expressed in an efficient manner, that is to say that it uses few elements in a network, if the depth of the network is insufficiently deep. As a result, developing effective learning algorithms for these architectures
is important as they may allow us to represent functions that are otherwise too complex to represent with a shallow architecture using a smaller number of elements. We can formalize this by defining what it means for an expression to be compact, “the expression of a function is compact when it has few computation elements, i.e. few degrees of freedom that need to be tuned by learning” [Bengio, 2009]. The result is that such compact representations demonstrate better generalization properties than non-compact representations. However, there is one issue, how deep do networks need to be to allow for compact representations of complex functions that are required for tasks like natural language processing? Bengio [2009] suggests that there is no universal minimum depth that works for all problems, it is in fact problem specific. So when developing new learning algorithms, they need to not be problem specific in the sense that should try to determine the depth required from the data set itself.

2.1.2 Previous Work in Deep Learning

Deep learning is very closely tied with Artificial Neural Networks, one of the key algorithms that is used in training various deep learning architectures is the backpropagation or gradient descent algorithm. Backpropagation was first described in the context of a training algorithm for neural networks by Rumelhart et al. [1988], allowing neural networks to be able to be used a computation tools. The idea behind the backpropagation is that the inputs are fed into a neural network, the outputs are then calculated using the weights and the input, then the error between this output and the desired output is calculated, finally this error is fed back through the network and the weights are adjusted accordingly. This algorithm however has some issues, the main one being that it is a very slow and computationally expensive algorithm. As a result it fell out of favor as a training algorithm for a period of time, however due to the recent large increases in computer hardware, most notably the GPU, the algorithm has seen more use as these hardware improvements have significantly reduced computation time. In addition, back-
propagation has issues with getting stuck at local minima as the starting point in weight space is randomized. Never the less, backpropagation has had some success in deep learning in various different architectures.

Despite the success of training deep neural networks using gradient descent, there were still some fundamental problems that needed to be addressed. Hochreiter and Schmidhuber [1997] identified and rigorously analyzed why deep neural networks could not be effectively trained using gradient descent and the answer is now know as the "vanishing gradient problem". In these networks the gradient of the error signal decays as it is propagated to the lower layers making it impossible to effectively train all the layers leading to poor generalizations. As a result, this problem drove most of the research in deep learning for the next two decades as new learning algorithms would be required in order to exploit deep learning architectures.

However it wasn’t until 2006 when Geoffrey E. Hinton [2006] came up with an efficient learning algorithm for training deep architectures. This new algorithm showed that better results could be achieved if the network was pre-trained layer by layer using unsupervised learning. The main idea here is that the first layer is trained with an unsupervised learning algorithm to set the initial parameters of that layer. Then the output of this layer is connected to the input of the next layer and this process is repeated layer by layer for the number of desired layers. This network can then be further tuned towards a given problem using the normal methods. The architecture described here is known as a Deep Belief Network which are generative architectures composed of multiple layers of hidden variables which in this case are Restricted Boltzmann Machines.

2.1.3 Energy Models and Restricted Boltzmann Machines

Boltzmann Machines have had a large impact on the field of deep learning as a result of Hinton et al devising a method for effectively and efficiently training them. Boltzmann Machines are in essence stochastic networks derived from the earlier recurrent neural
network architecture known as the Hopfield Network. Fully Visible Boltzmann machines have connections between all hidden and visible nodes in the network, this makes it impractical for real problems and so a variant of the Boltzmann Machine was developed called the Restricted Boltzmann Machine (RBM) where this is constrained so there are only connections between hidden and visible nodes. However, in order to properly understand Boltzmann Machines and RBMs, we must first discuss some basic concepts of energy-based models which form the foundations for the mathematics behind Boltzmann Machines. LeCun and Huang [2005] define a Energy-Based Model as a model that "associates a scalar energy \( E(W, Y, X) \) to each configuration of \( X, Y \)" where \( X \) is the set of inputs and \( Y \) is the set of variables that are defined as a result of the input. We can then use this energy-based model to define a probabilistic model [Bengio, 2009]:

\[
P(x) = \frac{e^{-Energy(x)}}{Z} \tag{1}
\]

Where \( Z \) is the partition function inspired by statistical mechanics:

\[
Z = \sum_x e^{-Energy(x)} \tag{2}
\]

However in many cases the variables in the model are not all observable so we have to introduce a hidden component to our model:

\[
P(x, h) = \frac{e^{-Energy(x,h)}}{Z} \tag{3}
\]

Since we are only concerned with non-hidden units we can find the marginal distribution by summing over all of the hidden states.

\[
P(x) = \sum_h e^{-Energy(x,h)} \frac{1}{Z} \tag{4}
\]

We can then use the notion of free energy from physics to define:
\[ \mathcal{F}(x) = -\log \sum_h e^{-E(x,h)} \quad (5) \]

We can then combine (4) and (5) to write

\[ P(x) = \frac{e^{-\mathcal{F}(x)}}{Z} \quad (6) \]

We can then use the energy equation from the Boltzmann Machine to calculate the gradient of the log-likelihood of our model. However because we have connections between nodes from the same layer, this is very computationally expensive and thus makes it hard to train Boltzmann Machines.

As a result, in order to make it possible to effectively train this model, so instead we must use an RBM which, as mentioned above, is no longer completely connected, the only connections are between a hidden and visible unit which aren’t in the same layer. We can then write the energy of this model as [Hinton, 2012]:

\[ E(v, h) = - \sum_{i \in \text{visible}} a_i v_i - \sum_{j \in \text{hidden}} b_i h_j - \sum_{i,j} v_i h_j w_{ij} \quad (7) \]

where \( v_i, v_j \) are the states of the hidden unit \( i \) and hidden unit \( j \), \( a_i, b_j \) are their biases and \( w_{ij} \) is the difference between them.

### 2.1.4 Stochastic Gradient Descent and Contrastive Divergence

Now that we have established a model for RBMs we can then train this model using an algorithm known as Contrastive Divergence (CD). Since stochastically approximating the gradient of the log-likelihood of models like RBMs is difficult another objective function known as the contrastive divergence which can be approximated more easily. However, since CD is closely related to Stochastic Gradient Descent (SGD), which is also an important algorithm for training DNNs in combination with backpropagation, it makes sense to first go over what SGD is.
In Machine Learning we often have optimization problems where we want to find some set of parameters, in this case weights $w$, such that a target objective function $Q(w)$ is minimized. One method for computing these weights, provided that $Q(w)$ is differentiable, is gradient descent, this method approaches a local minimum by taking steps towards the negative of the gradient. The update equation for gradient descent is given by [Bottou, 2012]:

$$w_{t+1} = w_t - \alpha \nabla Q(w_t) = w_t - \frac{1}{n} \sum_{i=1}^{n} \nabla Q_i(w_t) \quad (8)$$

Where $\alpha$ is the learning rate that is picked to help the algorithm converge and depends on the problem. The major issue with this method is that for large data sets it quickly becomes intractable to compute the gradient over the whole data set at each time step. This motivates the need to approximate this so that it can be used for larger data sets, this is where SGD comes into play. In SGD we pick a random sample from the data set and then use it to compute the gradient of the objective function. The update equation is now written as

$$w_{t+1} = w_t - \alpha \nabla Q(w) \quad (9)$$

In Machine Learning a popular objective function is the log-likelihood and in the RBM the gradient of the log-likelihood of a training vector with respect to a weight can be written as [Hinton, 2012]:

$$\frac{\partial P(v)}{\partial w_{ij}} = \langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model} \quad (10)$$

from this we can write a learning rule for SGD:

$$\Delta w_{ij} = \epsilon (\langle v_i h_j \rangle_{data} - \langle v_i h_j \rangle_{model}) \quad (11)$$
where $\epsilon$ is the learning rate, $\langle v_i h_j \rangle_{\text{data}}$ is the expected value of nodes $v_i h_j$ connected by weight $w_{ij}$ when the visible layer is “clamped” to the input data, and $\langle v_i h_j \rangle_{\text{model}}$ is the expected value of nodes $v_i h_j$ connected by weight $w_{ij}$ when the clamp is removed and the system is “free running”. Taking advantage of the lack of connections between hidden units in an RBM, we can easily obtain a sample of $\langle v_i h_j \rangle_{\text{data}}$. The probability of setting the binary state of each hidden unit $h_j$ to 1 given a randomly selected training value, $v$, is given by:

$$p(h_j = 1|v) = \sigma \left( b_j + \sum_i v_i w_{ij} \right)$$  \hspace{1cm} (12)

where $\sigma$ is the logistic sigmoid function $\sigma(x) = \frac{1}{1+e^{-x}}$. We can similarly obtain the probability of setting the state of a visible unit to 1 given a hidden vector:

$$p(v_i = 1|h) = \sigma \left( a_i + \sum_j h_j w_{ij} \right)$$  \hspace{1cm} (13)

The issue now becomes how do we get $\langle v_i h_j \rangle_{\text{model}}$ as it is intractable to compute directly. One way of doing this is by initializing the state of the visible units to a random state and then performing Markov Chain Monte Carlo (MCMC) Gibbs Sampling where all of the hidden units are updated in parallel using equation 12 and then the visible units are updated using equation 13 [Hinton, 2012]. If this sampling was repeated for an infinite number of steps we would get the exactly value of $\langle v_i h_j \rangle_{\text{model}}$ for calculating the gradient and a large number of steps would give us a reasonable approximation. A better way of doing this is instead of initializing the visible units states to a random state, we initialize it to a vector from the training data. Once again we then update the states for the hidden units using equation 12, we then obtain a “reconstruction” by updating the visible units using equation 13. We then repeat this for a small number steps $n$ and denote our contrastive divergence algorithm with $n$ steps of alternating Gibbs sampling $CD_n$ [Hinton, 2012].
2.1.5 Applications of Deep Learning in Natural Language Processing

Deep Learning techniques have shown some promising results in recent years, especially in the areas of automatic speech recognition (ASR) and image processing. Up until recently, [RAD, 2013] ASR systems have been dominated by systems using a combination of Gaussian mixture models combined with Hidden Markov Models to account for the variability that is inherent in speech signals. In addition, these systems also require some manual feature selection in order to get more accurate results. As a result, researchers from Microsoft [RAD, 2013] have applied deep learning techniques to try and solve this problem since deep learning models, specifically Deep Neural Networks (DNN) in this case, have the ability to learn features from the data on their own. The results showed that DNN systems showed noticeable improvement over the tradition GMM-HMM systems in terms of recognition error, and showed even better performance when given more primitive feature data.

Additionally they investigated this ability of DNNs to be a “universal learner” by performing two sets of experiments where the data involved different acoustic sources as well as different languages [RAD, 2013]. In one of the sets of experiments they investigated the effect of using data of different bandwidth to train the DNN. In this instance they used narrowband data which was treated as wideband data with half of the data missing, they then used that to train a wideband model. The results showed that there was reduction in word error rate (WER) using both the narrowband and wideband data, as a result this allows them to exploit the narrowband data for the model which they have gotten from other applications. The second set of experiments involved multilingual speech recognition, a task that is well suited for DNNs as DNNs have the power to represent the complex relationships that are universal to the target languages. The network is initially trained using all of the data from all of the languages, then the top layer is discarded and retrained using the target language. The results showed that using this additional data reduced the WER than just training with the target language,
this is interesting because it means that languages with less data can take advantage of languages with large amounts of data because of the DNNs ability to transfer phonetic information from multiple languages.

In addition to the area of speech recognition, deep learning architectures have shown promise in the area of image processing and multimodal learning. In the real world, data can come from many different channels such as audio and visual signals, so in order to get a complete picture of the world, multiple modalities need to be fused together in a joint representation. However, because these modalities have different properties, they have to be represented in different ways which as a result makes it difficult to learn relationships between two different modalities. Srivastava and Salakhutdinov [2012] propose a model using Deep Boltzmann Machines (DBM), to learn a joint representation over the different input channels, in this case they use user tagged images to learn a joint distribution over images and text. One of the interesting properties of this model is that it can be used to generate data for missing modalities by “clamping the observed modalities at the inputs and sampling the hidden modalities from the conditional distribution by running the standard alternating Gibbs sampler” [Srivastava and Salakhutdinov, 2012]. This is important because in real world applications it is very likely that there will be missing modalities. In addition, from their experiments, Srivastava and Salakhutdinov [2012] found that this ability to “fill in the gaps” of the missing modalities helps improve the overall performance of the model.

2.2 Spoken Dialogue Systems and State Tracking

In recent years, due to the rise of data-driven approaches, much like in the area of deep learning, spoken dialog systems (SDS) have become more robust and adaptive. As a result there has a been a shift from the traditional rules based systems towards these data driven statistical systems. One of the main areas of focus with these statistical approaches is the dialogue manager, which is the component of the SDS that controls
the overall behavior of the dialog system. Before discussing the dialogue manager in greater detail, I will first give a brief overview of the SDS architecture.

2.2.1 SDS Architecture

As can be seen in figure 1 SDSs in general are composed of three parts: the input, the output, and the controller [Reiser and Lemon, 2011]. These components can be broken down further with the input being composed of an automatic speech recognition (ASR) module and a spoken language understanding (SLU) module. The ASR module takes in speech from the user and then converts it into text which is then passed to the SLU module which extracts the meaning from the text. The controller is composed of a dialogue manager which is in this case composed of a state tracker and action selector. The state tracker maintains the overall state of the dialogue and then the action selector decides which action to take based on the state and is trained using reinforcement learning. Finally the output is composed of the natural language generation (NLG) module and the text to speech (TTS) engine. The NLG module takes the action that has been selected by the dialogue manager and converts it into text which is then passed to the TTS engine which converts the text into spoken language. The main area of interest for this paper is the state tracker which will be discussed in greater detail in the
2.2.2 State Tracking in Statistical SDS

Before describing the state tracker it is important understand what state tracking is, according to the Dialogue State Tracking Challenge [Williams et al., 2013], state tracking is “accurately estimating the user’s goal as a dialog progresses”. This is desirable as it enables the creation of more robust systems as uses techniques to reduce errors due to errors from the ASR module. However, according to Williams, there are issues with accurately estimating the dialogue state as ASR and SLU modules are prone to errors so the true state of the dialogue is only partially observable. One of the popular ways of solving this issue is to maintain a distribution of a set of hidden dialogue states. The hidden dialogue states contain all of the unobservable elements of the dialogue such as the users actual goal and the users actual action. We can then describe the ”belief state” of the dialogue as collection of dialogue states and their associated probability of correctness and by maintaining the belief state we can track the state of the dialogue. This belief state can then be updated by taking the N-Best list from the SLU and assigning each item a probability of correctness, then the next belief state is determined by the n-best list, the probability of correctness of each item and the previous belief state.

State tracking is an important part of Statistical SDSs, particularly in Partially Observable Markov decision process (POMDP)-based SDSs. POMDPs have been used to deal with the uncertainty that is inherent in SDSs and also incorporating three separate techniques for dealing with this issue: parallel dialog state hypotheses, local use of confidence scores, and automated planning [Young et al., 2013]. Before we talk about POMDPs in the context of SDSs I will first review some of the concepts for POMDPs.

A POMDP is defined as a tuple, \((S, A, O, T, \Omega, R)\) where \(S\) is the set of states, \(A\) is the set of actions, \(O\) is the set of observations, \(T\) is the set of transition probabilities, \(\Omega\)
is the set of conditional observation probabilities, and $R$ is the reward function. Then at each time step, the world is in some state $s \in S$ which is not known exactly, then the belief state $b$, with initial state $b_0$, is a distribution over the states. We can then define $b(s)$ as the probability of being in a particular state $s$. Then an action $a \in A$ is selected and a reward $r(s,a)$ is received, and the state of the system transitions to a new state $s'$ which depends on the previous state $s$ and the action $a$ taken. The system receives an observation $o' \in O$ and then the new belief is updated using [Young et al., 2013]:

$$b'(s') = \eta \Omega(o'|s',a) \sum_{s \in S} T(s'|s,a)b(s)$$

(14)

Where $\eta = 1/P(o'|b,a)$ is a normalization constant. Updating this belief state throughout the dialogue is known as belief monitoring. As mentioned previously, when an action is selected a reward is given to the system based on this action and the state the system is in. We can then define the cumulative, infinite-horizon, discounted reward [Young et al., 2013] as

$$\Theta = \sum_{t=0}^{\infty} \lambda^t r_t(s,a)$$

(15)

Where $\lambda$ is the geometric discount factor with $0 \leq \lambda \leq 1$. The goal of the system is to choose actions in such a way that maximizes the expectation value of this reward, this is called a policy $\pi$. The optimal policy denoted $\pi^*$ is then given by

$$\pi^* = \arg \max_{\pi} \left( \sum_{t=0}^{\infty} \lambda^t E[R(s,a)|b_0,\pi] \right)$$

(16)

While POMDPs give a framework for SDSs that helps to deal with uncertainty that is inherent in dialogue, there are still some issues with state tracking. One of the major issues as identified by Williams is correctly estimating each of the components of the model, if there are errors in the model then the performance of the system is decreased. In addition he noted that the current generative models have trouble discriminating between correct and incorrect hypothesis which implies that performance gains are due
entirely to ASR accuracy gains and not due to belief tracking allowing for better correct
decisions to be made. Another issue is that in real applications, such as personal digital
assistants, there is typical more than one domain for the dialogue, meaning that systems
will have to be scaled to work in the multi-domain or open-domain environment.

One way of dealing with the issue of poor discrimination in generative models is to
use a discriminative model which can keep accurate estimations of the state the system is
in. However discriminative models generally cannot track more than a few states at once
and correct hypothesis might be erroneously discarded. Metallinou et al. [2013] propose
a discriminative model which can maintain a distribution over an arbitrary amount of hy-
pothesis as well as having accurate probability estimates. According to Metallinou et al.
[2013] “discriminative approaches to dialog state tracking directly predict the correct
state hypothesis by leveraging discriminatively trained conditional models of the form
\( b(g) = P(g|f) \), where \( f \) are features extracted from various sources, e.g. ASR, SLU, dia-
log history, etc.” These features can be split into two categories: General features which
provide information about the overall meta-hypothesis that none of the given hypothesis
are correct, and hypothesis-specific which provide information about the correctness of
a particular hypothesis [Metallinou et al., 2013]. One of the issues with this approach is
that as the number of hypothesis increase, the number of feature functions to be learned
increases quadratically. As a result, in practice, this limits the number of hypothesis
that can be used by discriminative models which limits the potential performance of the
model, thus motivating a new model which can take advantage of discriminative models
[Metallinou et al., 2013].

2.2.3 State Tracking and the Dialogue State Tracking Challenge

Since dialog state tracking (DST) is an important problem in SDSs, recently a challenge
known as the Dialogue State Tracking Challenge (DTSC) has been created to create a
common testbed for DST. In addition to training and test data, as well as evaluation
tools, the DSTC provides baseline systems to measure performance against. In order to understand the baseline system I will be using to evaluate against I will give a brief overview of the system as described in the paper by Wang and Lemon [2013]. This system is a simple rule based model where the rules are applied to “observed system actions and partially observable user acts, without using any knowledge obtained from external resources” [Wang and Lemon, 2013]. The reasoning behind this approach is to determine how much information can be determined by only looking at SLU n-best lists which are error prone and as a result determine a better lower bounding for performance of machine learning approaches to the problems. In addition they also look to measure the system against machine learning based systems to get a better understanding of the strengths and weaknesses of the current machine learning techniques in the context of DST.

The system operates by executing an action and then receiving an observation at each time step, where the observation is an SLU n-best list which contains dialog acts which do not take slot-value arguments (e.g. affirm() or negate()) or gives one or more slot-value pairs (e.g. deny(route = 64a) or inform(date.day=today, time.ampm=am)) [Wang and Lemon, 2013]. We can then track our belief over the users goals by using the confidence score from the SLU n-best list as the probability a dialogue action occurs at the current turn as well as some “common sense” rules. So then the first step in each turn is to split acts from the n-best best list into single slot-value pairs and merge any identical slot-value pairs by summing their confidence scores. The next step is to apply the rules to update the current belief of the system depending on the user act at that turn as well as the machine action for that turn for some of the rules. One thing to note is that this update sometimes does not yield a valid belief meaning that the the probability of the belief summed over all values at a given slot is greater than 1. The solution is to normalize our marginal probability over all the values for a given slot [Wang and Lemon, 2013]. In the end there are some clear issues with this approach, most importantly that
this method cannot do any sort of error correlation as all of the dialogue acts as it makes
the assumption that they are all independent events, this leaves room for improvement
with machine learning techniques which can model error correlations. The system was
then evaluated with regards to hypothesis accuracy with a series of 4 test sets and it
performs as well or better than the majority of the systems in the first three test sets.

2.3 Deep Learning for State Tracking in Spoken Dialogue Systems

State tracking in SDSs is a complex task with issues that it needs to deal with such as
noisy conditions and ambiguity. Due to recent advances in speech research showing the
potential of deep learning, Deep Neural Networks (DNNs) have been explored as a model
for state tracking as they can model the complex interactions in dialogue features. In
addition due to the recent success of discriminative models in the DTSC, DNNs are also
attractive because they are also discriminative models. A recent paper by Henderson
et al. [2013] describes a Deep Neural Network approach to dialogue state tracking for
the DSTC, the following section will give an overview of method described in this paper.

In this model the tracker must maintain a distribution over $S_{t,s} \cup \{\text{other}\}$, where $S_{t,s}$
is the set of possible values for slot $s$ which have occurred as a SLU hypothesis at or
prior to turn $t$. Other in this instance is the probability that the user’s actual goal has
yet to appear as a SLU hypothesis. The architecture for the neural network for this
model is shown in figure 2 [Henderson et al., 2013].

In this architecture there are feature functions $f_i(t, v)$ for $i = 1 \ldots M$ which return
a real number for $v$ given some information about the SLU hypothesis as well as the
machine’s action at turn $t$ [Henderson et al., 2013]. The specific features that were used
in this implementation will be discussed later in more detail. We then choose a window
size $T$, which fixes the size the input layer for a given value $v$ such that the feature
functions are summed for for turns less than the current turn minus the window size
[Henderson et al., 2013]. Using the equations from our feed forward architecture depicted
Figure 2: Neural Network Architecture for calculating the probability distribution for each value $v$ in the set $S_{t,s}$

in 2 we can then write the overall distribution of the tracker as:

\[
P(s = v) = e^{\frac{E(t,v)}{Z}}
\]

\[
P(s \notin S_{t,s}) = e^{B/Z}
\]

\[
Z = e^{B} + \sum_{v' \in S_{t,s}} e^{E(t,v')}
\]

The Model is then trained using Stochastic Gradient Descent with mini-batches to minimize the log-likelihood of the parameters. Then the window size was selected such that “it is large enough to capture enough of the sequence of the dialog, whilst ensuring sufficient data to train the weights” [Henderson et al., 2013]. In the evaluation of the system they determined that a window size of 10 seems to work best. Next the feature set that is used in the model was determined, the features used were broken down into 4 sets of features $F_1 \ldots F_4$. $F_1$ only includes the SLU score, $F_2$ includes the features which are effected by the user act and values, $F_3$ further includes the user acts and the
machine acts, and finally $F_4$ includes the features that are effected by the system act and value [Henderson et al., 2013]. From the experiments they determined that using more features "monotonically increases the performance of the tracker". Next the overall structure of the architectures in terms of the number of hidden layer and the number of units per hidden layer, the best structure was found to be a three layer network with 20 units in the first layer, 10 in the next later, and 2 in the final layer. Finally the network was initialized with three different methods: Separate where a different model is trained for each slot, Single Model where one model is trained for all of the slots, and Shared Initialization where a single is model is trained for a few epochs and then this model is used to train separate models for each slot. Experiments using all three methods were conducted and Shared Initialization performed the best out of the three methods of initialization.

This system will provide the basis for the system I will be implement, using the same 12 feature functions and the window and will use the data collected from the Spoken Dialog Challenge where the users evaluated SDSs which provided information about the Pittsburgh bus system. The new model will be developed using stacked RBMs using Geoffrey E. Hinton [2006] method for layer-wise initiation for training deep structures.

3 Methodology

3.1 Data

As mentioned previously, the Dialog State Tracking Challenge (DSTC) provides a testbed including testing data, training data, evaluation tools, as well as baseline trackers. The first DSTC is concerned with the domain of bus route information in Pittsburgh and the users goal in the dialog was static. The data consists of log files, ‘log.json’, for each dialog, as well as ‘label.json’ which is a label object for datasets that are labeled. The data comes from 6 systems, each consisting of a combination of one of three dialog managers and one of two speech recognizers. The three dialog managers are:
• 0, A MDP based system that maintains a single dialog state and has a hand-crafted policy

• 1, A POMDP based system for tracking states with a hand-crafted policy

• 2, A POMDP based similar to system 1 but with a policy learned using reinforcement learning

The two speech recognizers are:

• 0, A Gaussian Mixture Model - Hidden Markov Model (GMM-HMM) based system that has been intentionally degraded

• 1, A GMM-HMM based system that has been optimized for the domain

The two combinations with dialog manager 2 and both of the speech recognizers make up the test set which as 1117 dialogs and the other combinations of dialog managers and speech recognizers for a total of 2118 dialogs.

3.2 Experiments

As mentioned previously I will be implementing a system using RBMs and greedy layer-wise pre-training using the 12 feature functions from Henderson et al. [2013] and evaluating it against both the Henderson system and the systems from the DSTC. The overall problem as mentioned previously the general problem it to maintain a probability distribution over a set of dialog states to track the overall user’s goal. A dialog state has three components, goals, method, and requested slots. Goals refers to a value that has been specified for the user for a certain slot or ‘None’ if the user has not specified a value for the given slot. Method refers to how the user is interacting with the system and is broken down into three general categories: ’by constraints’ where the user specifies a goal for that slot, ‘by alternatives’ where the user asks for an alternative suggestion, and ‘by name’ where the user asks for information about a specific item by name. Requested
slots refers to the slots where the user has requested information and as a result the system most inform the user.

The DSTC provides tools for evaluating these systems including a python script that evaluates the system with respect to a set of metrics which are referred to as ‘featured metrics’. These featured metrics are Accuracy, L2 norm, and ROC CA 5, where accuracy refers to the number of turns where the system’s 1-best hypothesis is correct divided by the total number of turns, the L2 norm refers to the L2 norm between the distribution of scores from the system and the labels, and the ROC CA 5 refers to the proportion of hypothesis that are correctly accepted when there is at most a 5% false accept rate. Schedule refers to which turns are used when computing the metric, in this case only turns which contain some information about a part of the state or if the correct label is None. Label scheme refers to how the data is labeled, in this case the components of the state are labeled using the most recently asserted values from the user. All three of these metrics are calculated for the join goals, the method, and the combined requested slots giving a total of 9 items to optimize where the Accuracy and ROC metrics are maximized and the L2 norm metric is minimized.

4 Requirements Analysis

4.1 Objectives

The overall goal of this project is to implement a deep learning architecture and evaluate it using the metrics and tools provided by the Dialogue State Tracking Challenge. This can be further broken down into five objectives:

1. Present Initial Design at the Edinburgh Deep Learning Workshop to get feedback from experts

2. Develop the feature extraction and vectorization component to extract the required information for training
3. Build the new system using RBMs and greedy layer-wise pre-training using the feature functions

4. Evaluate this system with respect to the first system and the other DSTC systems

5. Present the results of the system in the form of a poster presentation

4.2 Requirements

As can be seen from the objectives, there is essentially two stages to the project, feature extraction and vectorization and developing the stacked RBM system. Both of these stages are needed to complete the model and acquire results. Then if time permits further research can be investigated, this will be discussed in more detail in the Future Work section.

5 Professional, Legal, and Ethical Issues

As this project is focused towards a more theoretical evaluation of our approach to the problem, there are few ethical issues. The main issue is data anonymization since the data being used was collected from real people using Spoken Dialog Systems. However this has already been handled by the organizers of the Dialogue State Tracking Challenge and they data we are given both to train with and to test with has been anonymized. In addition, the system will be evaluated using the test data provided according to the metrics presented in the methodology chapter and will not be evaluated by actual users.

6 Project Plan

As mentioned in the requirements analysis there are five goals for this project:

1. Present Initial Design at the Edinburgh Deep Learning Workshop to get feedback from experts
2. Develop the feature extraction and vectorization component to extract the required information for training

3. Build the new system using RBMs and greedy layer-wise pre-training using the feature functions

4. Evaluate this system with respect to the first system and the other DSTC systems

5. Present the results of the system in the form of a poster presentation

These goals represent the two main components for this project which are the feature extraction and vectorization, and the system that uses stacked RBMs in combination with layer-wise pre-training described in Geoffrey E. Hinton [2006]. The RBM system can be broken down into a set of tasks: development, training, and testing. In addition, the project consists of a written report component as well as a poster presentation, all of these tasks with time frames can be seen in the schedule in figure 3.

Figure 3: Gantt chart showing the schedule for the project

In this plan I have accounted for the possibility of something such as illness preventing me from completing the whole projects, the new deliverable becomes only system 1 as system 1 is the minimal subset of the project. I have also accounted for the possibility that the time estimates are too conservative and I have extra time, other things that could be implemented are discussed in the Future Work section.
7 Implementation

7.1 Feature Extraction and Vectorization

In order to be able to be used for training our model, the log files containing the dialogs must be preprocessed to extract the features from the dialog. In order to facilitate comparison between the new model using the newer stacked RBM with pre-training and the original feedforward neural network, the same set of 12 features was used. Before talking about how the features are extracted from the log files, it makes sense to talk a little about how the logs are formatted.

The logs are in the JSON format where each log file corresponds to a single call. The top level structure of the log includes the session id, the session time, the session date, the system-specific information for the call, and finally the turns which is an array of dictionaries, one for each turn. Each turn consists of the input received by the dialog manager, system-specific information associated with the turn, the turn index, the restart flag, and finally the output generated by the dialog manager. For vectorization we’re only concerned with the input and output components of each turn as they contain all of data used to capture the features.
As can be seen in figure 5, the input component of each turn consists of a live entry, a batch entry, and audio file, a start time, and an end time. The live and batch entries have the same format and contain the recognition results produced in development and offline respectively. Since the original system used the live recognition results, we will also only look at the live recognition results. The recognition results then consists of two parts, the ASR results and the SLU results, we’re only interested in the SLU results as the features only look at information from the SLU results and do not take into account the ASR results. The SLU results is a list of all the SLU hypotheses for the given turn and each hypothesis consists of a dialog act which consists of an act type and zero or more slot value pairs depending on the dialog act type. Each hypothesis also has an associated score which for the live results is a real number in the range [0,1].
Figure 5: Example of system input for a log file

The system output has a similar format, each call consists of a start time, and end time, a transcript of the words spoken by the system, and finally the dialog acts. The dialog acts consists of an array of dialog acts that describe the the transcript. It should also be noted that some of the dialog act types are specific to the system and visa versa.
Our 12 features are then [Henderson et al., 2013]:

1. **SLU score**: the score assigned by the SLU to the user asserting \( s=v \).

2. **Rank score**: \( 1/r \) where \( r \) is the rank of \( s=v \) in the SLU n-best list, or 0 if it is not on the list.
3. **Affirm score**; SLU score for an affirm action if the system just confirmed s=v.

4. **Negate score**; as previous but with negate.

5. **Go back score**; the score assigned by the SLU to a goback action matching s=v.

6. **Implicit score**; 1 - the score given in the SLU to a contradictory action if the system just implicitly confirmed s=v, otherwise 0.

7. **User act type**; a feature function for each possible user act type, giving the total score of the user act type in the SLU. Independent of s & v.

8. **Machine act type**; a feature function for each possible machine act type, giving the total number of machine acts with the type in the turn. Independent of s & v.

9. **Cant help**; 1 if the system just said that it cannot provide information on s=v, otherwise 0.

10. **Slot confirmed**; 1 if s=v’ was just confirmed by the system for some v’, otherwise 0.

11. **Slot requested**; 1 if the value of s was just requested by the system, otherwise 0.

12. **Slot informed**; 1 if the system just gave information on a set of bus routes which included a specific value of s, otherwise 0.

The main loop to extract the features is fairly straightforward, we loop through each call in our dataset and first reset our feature extractor to clear the history. Then we loop through all of the turns in that call, extracting the features at each turn and appending it to our list of turns. Finally we append each call to our list of sessions. The process for extracting the features at each turn is also straightforward, we first get the relevant machine acts, SLU hypotheses, and labels if the data set has labels, adding the
machine acts and Slot Value hypotheses from the SLU hypotheses to our history. Next we create our feature vector and update the features that are independent of Slot-Value pair. Next we loop through the hypotheses and copy our feature vector for each slot value pair. Then we loop through the slot-value hypotheses and update the features for each slot-value pair, this includes the SLU score, the Rank score, the features related to the user acts and the features related to system acts. Finally we create our data point consisting of “x” our dictionary of features at the given turn, and “y” the goal labels for the turn.

7.2 Stacked Restricted Boltzmann Machine

As mentioned previously, our model for dialog state tracking uses stacked restricted which are pre-trained layer by layer before being stacked together and combined with a logistic layer to tie our outputs. The model is implemented in Python using the Theano library [Bergstra et al., 2010] and incorporates code from the Deep Learning Tutorials for the RBM class and DBN class. Since the original model was set up to work with the MNIST dataset, some changes had to be made to allow for the model to work on our new dataset. First, the way the dataset is loaded needed to be changed to load the new dataset properly so that our feature vectors can be used to train our model.

Pertaining for our model works exactly the same as the original model, during the pretraining phase we can take advantage of the unlabeled datasets and use all of the data we have available to train the model. The general algorithm for pretraining is as follows:

1. Train an RBM using the input X as the visible layer to be used as the first layer.

2. Transform X using the first layer to obtain data for second layer by either sampling or computing mean activation of hidden units.

3. Train the second layer using this transformed data as training examples.
4. Repeat steps 2 and 3 for the desired number of layers.

In our case we loop through each of the layers, get the cost and updates of each of our RBM layers by performing $k$ steps of contrastive divergence where $k = 1$ by default. We then use this to compile our Theano function with the inputs being the batch index and learning rate and the outputs being the costs. We then append this to our list of pretraining functions and repeat this process for each of our minibatches.

For the finetuning component of training we separate our training set by slot and fine tune each slot separately using only the data from that slot. However if a certain slot does not have any data in the training set for a particular slot, we combine all of the data and use it to train that slot. If the test set does not have any examples of a particular slot we ignore that slot completely and continue training. When finetuning we make use of the early stopping method to prevent our model from overfitting. Early stopping works by keeping a held out set of examples to be used for validation and then when training we look at a certain number of examples based on a factor known as patience, and if our model improvement is above a certain threshold we keep train, otherwise we stop. Then the process during training is we loop through our minibatches and first compute the average cost, then if we need to validate the model based on the validation frequency, we calculate the average validation loss and if it is better than our best previous loss by enough we increase the patience. Then finally we test it on the test set and calculate the average loss which is the test score accuracy.
8 Experiments

One of the main issues with the various deep learning methods is that there are many hyperparameters that have to be set. However, there are not any concrete rules for how to pick these parameters so a lot of work goes into tuning these parameters to ensure the best performance. For our model we looked at several different parameters including the size of the minibatch, the fine turning and pretraining learning rates, and the shape of the network. For each experiment we varied one of the parameters and kept the others constant with our base line having 10 pretraining epochs, 10 fine tuning epochs, a fine tuning learning rate of 0.1, a pretraining learning rate of 0.01, and layer sizes 20-20-20. Additionally we used training sets train 1a and train 1b, train 1a for fine tuning and validation, and test 1 for testing.

When looking at our results we will be looking at the accuracy score for each of the slot groups which are route, from.desc, to.desc, from.neighborhood, to.neighborhood, from.monument. These slot groups represent the possible slots that the users actions can contain. For example if the system were to ask “Where are you leaving from?” and the user responds with “CMU”, then the user act would be inform(from.monument=cmu). As mentioned previously, our domain is the Pittsburgh Bus System so our slots represent the various locations users can go to and from as well as the different bus routes.
8.1 Batch Size

The batch size refers to the number of samples that are used in a minibatch for doing CD-k in pretraining stage and SGD in the finetuning stage. As can be seen in figure 7, increasing the size of the batch increases the accuracy of the system. This due to the fact that using more samples allows us to estimate the gradient of our dataset more accurately, however using too large of batches can degrade the overall performance. In our case there is only a slight degradation for increasing the batch size beyond 20, with a batch size of 50 performing only slightly worse than a batch size of 20, which has the overall best performance.

![Figure 7: Accuracy for each slot with varying minibatch size.](image)

Figure 7: Accuracy for each slot with varying minibatch size.
8.2 Fine Tuning Learning Rate

The fine tuning learning rate controls the size of the weight and bias updates during the fine tuning state of training. If the learning rate is too high the size of the weights may explode and lead to poor performance. As can be seen in figure 8, it appears as though the fine turning learning rate has little effect on the overall performance of the system. One possible explanation of this could be that our search space is fairly small and pretraining narrows the search space further meaning that only a little fine tuning needs to be done. Another possible reason is that the pretraining learning rate might be too high and narrowing the search space enough that little fine tuning needs to be done.

Figure 8: Accuracy for each slot with varying fine tuning learning rate.
8.3 Pretraining Learning Rate

The fine tuning learning rate controls the size of the weight and bias updates during the fine tuning state of training. As can be seen in figure 9, we get similar performance when the learning rate is 0.1 and when the learning rate is 0.001 and similar performance when the learning rate is 0.01 and 0.0001. It makes sense that a higher learning would give better performance as we are only running for 10 epochs to reduce computation time. However the reason that the performance would increase again at 0.001 is a bit harder to understand. It could possibly be that with the pretraining rate at 0.01 the weights end up getting stuck in a bad part of the parameter space and that by reducing the learning rate we are able to get out of this area.

![Figure 9: Accuracy for each slot with varying pretraining learning rate.](image)

Figure 9: Accuracy for each slot with varying pretraining learning rate.
8.4 Layer Shape

The shape of the network refers to the arrangement of the layers in the network for example a funneling architecture decreases in layer size with each subsequent layer. For our experiments we test two funneling architectures, one where the layer size decreases in subsequent and one where the layer size increases in subsequent layers. We also tried two other architectures, one where the middle layer is smaller than the other two layers and one where the middle layer is larger than the outside layers. As can be seen in figure 10, the 2-10-20 and the 10-20-10 architecture architectures perform better than the 20-10-2 and the 20-10-20. This seems to point to the fact that having fewer units in the first layer leads to better performance.

![Figure 10: Accuracy for each slot with varying layer shapes.](image)

Figure 10: Accuracy for each slot with varying layer shapes.
9 Evaluation

As mentioned before there is a lot of tuning to be done to determine the best parameters for the model to get the best performance and due to time constraints we had to stop at some point and pick the best parameters to evaluate the model with. The parameters used for testing were 10 pretraining epochs, 10 fine tuning epochs, a fine tuning learning rate of 0.1, a pretraining learning rate of 0.01, batch size of 20, and layer sizes 20-20-20. Additionally, we used the same training sets, train 1a and train 1b for pretraining and finetuning as we did for experiment with the parameters. With the DSTC we have 4 different test sets to test against, each having slightly different structure due to the different systems being used to produce the data sets. Due to time and resource constraints we only evaluated accuracy and compared it all of the other systems for each slot group excluding date and time. In the case of test sets 2 and 3 we only had instances of route, from.desc, and to.desc so the accuracy is only shown for these slots.

9.1 Test 1 - Large Quantity of Same System Data

Test Set 1 consists of calls that are very similar in structure to those in training sets 1 and 2 to test performance when there is a large quantity of same system data. As a result the performance in general for this test is the highest in general for each of the test sets. Our system did reasonably well on this test set, achieving accuracies above 80% for each of the slots but performed slightly worse than other systems on the slots to.monument, from.mounment and to.neighborhood. Since we did not take advantage of all the data available that is similar for this task and because we did not tune specifically for this task, it makes sense that our model would perform slightly worse than other models, however the overall performance on the tast is still fairly good. We only performed worse than the average on the to.monument, from.monument, and to.neighborhood slots but outperformed all of the other systems on the to.desc, from.desc and route slots.
9.2 Test 2 - Large Quantity of Similar System Data

Test Set 2 consists of calls that are somewhat similar in structure to those in training sets 1 and 2, to test performance when there is a large quantity of similar system data. As mentioned previously, this test set does not contain examples from certain slots, so we can only compare the slots to.desc, route, and from.desc. As can be seen in figure 12, our model performs fairly well compared to the other systems for these slots. In fact our model achieves the highest performance out of all of the models we are looking at.
9.3 Test 3 - Small Quantity of Similar System Data

Test Set 3 consists of calls that are similar in structure to those in training set 3, to test performance when there is a small quantity of similar system data. As mentioned previously, this test set does not contain examples from certain slots, so we can only compare the slots to.desc, route, and from.desc. Our system performed very well on this test set, outperforming all of the other systems for the slots route and from.desc. It should be noted that we are not using any of the training data that is similar to this test set, this implies that we are able to learn a more general representation and thus perform well regardless of how similar the test set is to the data we train on.
Figure 13: Box plot depicting Accuracy for each slot for test set 3, where the black dot represents our model and the blue cross represents the mean.

9.4 Test 4 - No Similar System Data

Test Set 4 consists of calls that are different from all of the training sets, to test performance when there is no similar system data available. For this test our system performed the best out of all the tests with accuracies greater than 90% for each of the slots. Compared to other systems, our system performed fairly well, managing to outperform all of the other systems for the slots to.desc, from.desc, from.neighborhood, and to.neighborhood. This makes sense because as mentioned previously, our model is designed to learn more general representations so other systems which are engineered towards similar systems perform worse on this task where as we perform well because we generalize better. This is similar to what Henderson et al. discovered with their model, they performed best on test 4 since there model generalizes better than other models.
Figure 14: Box plot depicting Accuracy for each slot for test set 4, where the black dot represents our model and the blue cross represents the mean.

9.5 Comparison with Henderson et al. Model

In addition to comparing our system to all of the other systems that entered the DSTC 1, we are particularly interested in how our system compares to the Henderson system as our system is an extension of their model using the newer stacked RBM with pretraining model. Our model uses the same 12 features as mentioned previously as well as the windowing function to summarize previous turns from the dialog. In all four test sets our model performed similarly or outperformed the Henderson model particularly on the Test sets 2 and 3 which were noted as the test sets with the worst performance for their model.

9.5.1 Test 1 - Large Quantity of Same System Data

As mentioned previously, test set 1 is designed to test the situation of when there is a large quantity of same system data available to train on. Henderson et al. [2013]
found that their system performed best on test set 1 and test set 4 and as noted in the previous sections, our system has good performance on test set 1 and test set 4 as well. In figure 15 we can see that we outperform their system on the slots to.desc, route, from.desc, and from.neighborhood. Their system then outperforms our system on the slots to.monument, from.monument, and to.neighborhood. However on average our system had better performance with an average performance of 92.1% accuracy versus their 88.1% average accuracy. This can be attributed to the fact that while they had very high performance on the slots they outperformed us on, they had some other slots with much lower performance such as from.desc.

Figure 15: Bar chart depicting the accuracy for each slot for test set 1, the blue bars represent our system and the red bars represent the Henderson system

9.5.2 Test 2 - Large Quantity of Similar System Data

Test set 2 is designed to test the situation of when there is a large quantity of similar system data for training. Henderson et al. [2013] noted that their system had worse
performance on test sets 2 and 3 compared to their performance on test sets 1 and 4. Our system managed to outperform their system on all three of the slots we looked at for this test set. Our system actually outperformed all of the systems on this test set particularly on the to.desc and from.desc slots where our performance was 85.7% and 88.4% versus the 62.6% and 52.9% average accuracy respectively. Our average performance is still very high at 90.7% versus their 67.3% average accuracy.

![Bar chart](image)

Figure 16: Bar chart depicting the accuracy for each slot for test set 2, the blue bars represent our system and the red bars represent the Henderson system

9.5.3 Test 3 - Small Quantity of Similar System Data

Test set 3 is designed to test the situation where we only a limited quantity of similar system data for training. As mentioned previously Henderson et al. [2013] noted that their performance on test set 3 was worse than on test sets 1 and 4. As with test set 2, our system outperforms their system for all three slots and in fact outperforms all of the other systems on the route and from.desc slots. However our average performance
on this test set is the lowest out of all four of the test sets with an average performance of 87.1% compared to 78.4% for their system.

![Figure 17: Bar chart depicting the accuracy for each slot for test set 3, the blue bars represent our system and the red bars represent the Henderson system](image)

9.5.4 Test 4 - No Similar System Data

Test set 4 is designed to test the situation where we have no similar system data to train on. Henderson et al. [2013] noted that their system had good performance on test set 4 because they chose a feature set that would allow them to learn a more general representation rather than trying to exploit the ASR+SLU properties of the other test sets. Our system managed to outperform their system in every slot except for the from.monument monument. This test set also produced the best average results for our system out of all of the test sets with an average performance of 95.3% versus their 84.8% accuracy. It appears as though our model has the same advantages that are gained from using the feature set but the newer stacked RBM with pretraining model
further improves the performance when we do not have similar system data to train on.

Figure 18: Bar chart depicting the accuracy for each slot for test set 4, the blue bars represent our system and the red bars represent the Henderson system

10 Conclusions

We were able to successfully integrate the feature functions from Henderson et al. [2013] with the newer deep learning model using stacked RBMs with pretraining. Overall our model achieved fairly good performance and outperformed all of the other models on certain tasks. Our model also showed noticeable improvements upon the Henderson model and particularly in test sets 2 and 3 where their model had a noticeable weakness. As a result we have shown that Deep Learning techniques are powerful for dialog state tracking and in particular when we don’t have similar data and need to generalize better. However, due to some issues with limited resources and time constraints we weren’t able to utilize all of the data we had and tune the model parameters further to try and squeeze
out as much performance as possible. Additionally, due to the complexity of these sort of models, it can be very hard to understand what is going on when tuning the various parameters and it can be hard to explain why changing a parameter a certain way effects the performance. Also due to time constraints, we were unable to get results for other evaluation metrics to see how our model compares to the other model in terms of the other metrics.

11 Future Work

Since the first DSTC there have been an additional two other DSTCs, DSTC 2 and DSTC 3. In the DSTC 2 we have a new problem where the users goal is no longer static and is allowed to change during the dialog and domain has changed to restaurant information, in the DSTC there is a further domain extension for the DSTC. 3 The domain becomes tourist information in general and not just restaurant information. This challenge is about trying to develop robust state trackers which can work in a domain where there is little labeled data but a large amount of data in other smaller related domains. In addition we could also expand our feature set to try and capture additional features in the dialog. One example might be a feature to capture the error correlation between similar words as ASR error and SLU error are a large source of errors, this feature was initially considered but was not implemented due to time considerations. Additionally the system could be adapted to work directly on the ASR signal to cut out the SLU portion of the SDS in order to remove another source of error. Finally because the general model is very general, the system could also be adapted to work for other systems such as multimodal systems and robotics, only the feature functions would need to be changed to suit the new problem domain.
References


