CLASSIFICATION PROBLEMS WITH REWARD-MODULATED INFERENC

BUCK SHLEGERIS, U5192430

ABSTRACT. Denoising autoencoders are a kind of neural net which can be used for both supervised learning and unsupervised learning: for supervised learning, you train them to maximise the likelihood of the data they're trying to predict, and for unsupervised learning you train them to competently compress and decompress their input. The combination of these two aims of the neural net is called reward modulation. This project investigated neural nets and reinforcement learning in general, and RMI using denoising autoencoders for a classification problem in particular.

1. INTRODUCTION

Machine learning is a field of computer science focussed on finding algorithms to make predictions based on experience. In this report, I'll briefly summarize the goals of three areas of machine learning: supervised learning, unsupervised learning, and reinforcement learning.

1.1. Supervised learning. Supervised learning is the area of machine learning which is concerned with learning functions from examples of input and output from the function.

For example, spam classification is the task of learning a function from a string representing an email to a boolean representing whether the email is spam or not. More generally, the classification task requires the learning agent to learn a function from inputs to a set of labels.

Another common supervised learning task is to map from input values to a continuous space. This is called regression. Some examples of regression tasks include learning to predict income based on a selection of data on a person, or learning to predict SOMETHING ELSE

1
If we want to, we can view classification to a set of \( n \) labels as a regression problem learning a function called fit to a vector in \( \mathbb{R}^n \), where the \( i \)th entry is how well the input fits into the \( i \)th category. We then say that the model classifies the input into the label with the best fit. That is,

\[
\text{classification}(x) = \arg \max_i \text{fit}(x)_i
\]

One problem with supervised learning is that it requires a large, labelled training set, which might be difficult or annoying to acquire in practice.

1.1.1. *The classification task.* The classification task is a common problem in machine learning. You have a set of inputs in one space, and you want to learn a classification function which transforms them to another smaller space of categories. For example, we might want to take in a bunch of faces as bitmaps and categorize them by race and age, or we might want to take in a bunch of PayPal transactions and classify them into money laundering and not money laundering.

One of the earlier major successes for machine learning was in handwritten digit recognition, for use in post offices. One standard task which this report uses extensively is the MNIST digits training set. The MNIST database is a collection of 60,000 pairs of a handwritten digit and its label between 0 and 9.

There are a lot of different algorithms which can be used for the classification task. One of the simplest is \( k \)-nearest-neighbours. In \( k \)-nearest-neighbours, you choose a distance function over the set of inputs. Then, to predict the label of an input \( x \), you look through your training set for the \( k \) closest vectors to \( x \). You then look at those \( k \) vectors, and classify \( x \) with the most common label among them.

In an information-theoretic sense, the expected amount of information gained from each observation falls over time, because initially you have no idea what the classification will be, but as your model improves, you’re less surprised by each sequential observation.

1.1.2. *The regression task.* In the regression task, the learning agent tries to learn a continuous function from past examples of data. For example, we might try to predict someone’s
income from a variety of data about them.

A variety of non-neural net techniques for regression are popular. They weren’t the focus of this project, so I won’t discuss them.

1.2. **Unsupervised learning.** Unsupervised learning is the area of machine learning which tries to find patterns in unlabelled data. It can be viewed as trying to learn a function \( p(\vec{x}) \) which assigns a probability density to all vectors \( \vec{x} \) in some space. One way of stating the goal of unsupervised learning is that you want to maximise the likelihood of data which is actually generated by the process you’re learning from.

This problem is closely related to compression: if you can learn a model of your data which assigns higher probability to more likely inputs, you can express those likely inputs in less space.

In this project, we’re mostly interested in unsupervised learning as a preprocessing step for the supervised learning and reinforcement learning problems which we’re focussing on. The patterns which unsupervised learning algorithms notice are often useful insights for the learning problem we’re actually looking at. For example, unsupervised learning of images quickly detects the correlation of nearby pixels to each other: if one pixel is red, it’s suddenly more likely that the pixels around it are red. So the unsupervised learning process is going to learn features of the input like ‘it has a red patch here’. These features are often easier to learn from than the original input was.

1.3. **Reinforcement learning.** Reinforcement learning is another area of machine learning, with a slightly different phrasing of the problem.

In RL, we have an agent who is learning how to navigate in an environment. Every step, the agent sees an observation and receives a reward. It then chooses an action.

This is an extremely general framework, in which many other machine learning problems can be expressed. For example, we can express supervised learning as a problem where the agent see an input, and has an action corresponding to each output that the agent could have. The agent then recieves a reward depending on the accuracy of its prediction, and
then sees an observation corresponding to the correct label.

RL is an attractive framework for researchers interested in general AI. In theory, we could give an RL agent any number of different tasks with no explanation of how the tasks work, then let the agent figure that out itself. This indeed works, on problems from balancing a pole [Sutton, 1984] to flying a helicopter [Abbeel et al., 2007] to playing Pacman [Szita and Lörincz, 2007].

One of the essential parts of solving RL problems is choosing a bunch of assumptions you want to make about the environment. This is necessary, because the general RL problem is so general that you can’t actually make any conclusions about it without making further assumptions. This problem is part of the general problem of induction.

The bandit problem, which we’ll look at below, is the RL problem where we make the assumption that the environment has only one state and never changes over time. We’ll also look at a few other, more general assumptions. For the most general solution to these problems, we can use Solomonoff’s prior and take actions according to that. This agent is called AIXI.[Hutter, 2007]

In RL, both episodic and continuous environments are considered. Episodic environments are environments where the task can be broke into similar, separate episodes. For example, playing games of chess or Tetris, or finding your way through a maze. In this report, we only considered episodic tasks, but many of the preprocessing methods we investigated would work for continuous tasks.

1.3.1. The $n$-armed bandit. One simple example of an environment is called the $n$-armed bandit. The agent is in a casino, and has bought 100 turns, which it can use on any of $n$ different poker machines. Each poker machine has a probability distribution over different prizes: maybe it gives you either $100 or $10 based on a coin toss. Or maybe it gives you a prize between $0 and $50 uniformly at random.

However, the agent doesn't have any reason to believe that all of the poker machines have the same distribution. It therefore has a dilemma: how should it maximise its winnings given that it wants to both try a few different machines to find out how good they are
(‘exploring’) but also pull the best machine repeatedly (‘exploiting’)? This explore-exploit dilemma is a central question in reinforcement learning.

1.3.2. Approaches to the n-armed bandit. Part of the solution to the RL problem is having a way to estimate the value of doing particular actions. This process is called value estimation. We write our estimation of the value of a state as \( v(s) \). We often want to talk about the ideal correct value estimation function, and we denote that \( v^*(s) \). In the case of the bandit problem, the easiest approach to value estimation is to just assume that the value of an action is just the average reward received over all the previous times you’ve taken that action.

This is only justified if we know that the arms of the bandit have stationary distributions, which means that their distribution does not change over time. This is an example of a kinda sketchy thing which we can assume to make the problem easier to solve.

If we didn’t know that the problem was stationary, and we suspected that the value of the actions might change over time, one way of dealing with it would be to weight our observations by how recently that happened. We can do this with exponentially decaying weights by simply storing our value approximation for every state, then updating when we see a reward \( r \) and a learning rate \( \alpha \):

\[
\text{(2)} \quad v_{\text{new}}(s) = (1 - \alpha) \cdot v_{\text{old}}(s) + \alpha \cdot r
\]

The other problem is to determine the optimal action to take after you know the value estimates. The simplest strategy which works reasonably well is called an \( \epsilon \)-greedy strategy. In this strategy, we pick a value of \( \epsilon \), such as 10%. In that case, we do the optimal action 90% of the time, and a random action 10% of the time. This ends up exploring enough of the time that it will eventually discover a strategy which is at most 10% worse than the optimal strategy: it does the best action most of the time, but does a potentially stupid action whenever it explores. For any \( \delta \) away from the optimal value, you can choose an \( \epsilon \) such that an \( \epsilon \)-greedy strategy with that \( \epsilon \) gets an average reward within \( \delta \) of the optimal policy.

A better strategy is to decrease \( \epsilon \) over time. At the start, exploration is highly valuable, because you have a lot of moves in the future which can be informed by information you
gather now. In contrast, when you’re nearly at the end of the problem, new information isn’t that useful to have. A strategy which eventually decreases $\epsilon$ to 0 turns out to be asymptotically optimal.

We can quantify this by explicitly calculating the value of information at different points in the problem, and from that you can calculate an analytical solution to the optimal $\epsilon$ function. However, this relies on even more sketchy assumptions: Sutton and Barto recommend against it [Sutton and Barto, 1998].

1.3.3. Bandit feedback. In the classification task, the learning agent is told the correct answer every time it tries to classify something. This is called full feedback.

Bandit feedback is a variation of this problem, more naturally described in the language of RL. We now consider the different labels which the agent outputs to be actions, and it receives a reward in $[0, 1]$ after every action. We can imagine this as the agent seeing images appear on a screen, and having to choose a button to press in response to the images, and receiving some number of cents depending on what they pressed.

The most direct translation of the classification problem to bandit feedback environment is just the environment where you get a reward of 1 if you classified the input image correctly. The main difference here is that if you got it wrong, you don’t learn what the correct answer is. The output from the classifier, which we were previously thinking of as posterior probability, is now interpreted as an expected reward from hitting that button.

We can learn from this kind of feedback too. Instead of getting feedback on all of our predictions, we only get feedback on the prediction that corresponds to the action we actually took. This means that the agent gets significantly less information from this approach, particularly when the agent is just starting out. As the agent gets better at guessing, the feedback becomes more useful to it: if it’s pretty sure that the digit it sees is a 2 or an 7, and it guesses 7, and it was wrong, it can use that as evidence that the digit was actually a 2. Whereas if the agent has no idea what’s happening, all it learns was that the given input didn’t correspond to a 2, and has very little idea what other digit it would correspond to.
We can talk about the differences between full feedback and bandit feedback from an information theory perspective. Suppose that we’re looking at a task where there are \( n \) different labels, and each is evenly represented in the training set. Suppose that at some timestep, the agent has probability \( p \) of classifying correctly. Now, if we make the potentially dodgy assumption that whenever the agent guesses, it splits its probability mass evenly over the other options.

In the full feedback environment, the agent has a \( p \) chance of seeing the label which it expects, and has a \((1 - p)\) chance of being surprised by the label, in which case it will see something which it thought it had a \( \frac{1-p}{n} \) chance of seeing. So the expected information gain is:

\[
- p \log p - (1 - p) \log \left( \frac{1 - p}{n} \right)
\]

In the bandit feedback environment, if the agent is wrong with its guess, then it just sees a 0 reward, which it thought it had a \( 1 - p \) chance of seeing. So its information is:

\[
- p \log p - (1 - p) \log(1 - p)
\]

We can plot these, as in figure 1. We see that the ratio between the bandit feedback and full feedback constantly increases. We can observe from this that the advantage of full feedback falls as the agent gets better.

Semi-supervised learning is an area of machine learning where the learning agent has a bunch of input examples, only some of which are labelled. This is obviously relevant to the problem of using supervised learning techniques for RL. A wide variety of techniques have been applied to this problem [Chapelle et al., 2006]. Most of them aren’t related to the neural nets which we focussed this research project on, so I won’t discuss them here.

Depending on how we want to treat this bandit feedback problem, we can worry about the explore-exploit dilemma or not. If we are using algorithms which are allowed to know that for any given input, only one action will get a reward, then they can make more inferences. Such an assumption is justified in this case. However, algorithms which make that assumption can be generalized less easily to other domains.
1.3.4. **Finite Markov decision processes.** A finite Markov decision process, or MDP, is an environment where the behaviour of the environment at a given time can be described entirely by its current state. For example, the dynamics of a bouncing ball can be described entirely by its current state and current velocity. The dynamics of a MDP can be described by a function from states and actions to the probability distribution over new states and rewards.

Finite MDPs are the largest well understood class of RL environments [Sutton and Barto, 1998]. Many practical problems can be approximated as finite MDPs, even if they’re aren’t quite.

One common strategy to solve MDPs is to progressively learn approximations to the value of every state-action pair, and progressively learn approximations to the dynamics function.
of the MDP. The true value function for a particular policy \( \pi \) is denoted \( q^* (s, a) \), and we call the approximation which the agent is actually using and working on \( q_\pi(s, a) \). We can estimate this with Monte-Carlo methods.

The Monte-Carlo method for estimating the optimal action-value function of a state is pretty simple. The agent starts out by deciding on some policy. Now, the agent wanders around the environment with this policy for a while. As this goes, we’re getting a bunch of experience about the world. In the bandit case, there’s only one state, so the expected value of an action is the average value we’ve seen from that state in the past. In this MDP case, we need to take future rewards into account. We can define the optimal action value function [Sutton and Barto, 1998] as:

\[
q^\pi(s, a) = \mathbb{E} \left( \sum_{t=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a \right)
\]

where \( \gamma \) is a discounting parameter, \( R_t, A_t, \) and \( S_t \) are respectively the reward, action, and state at timestep \( t \).

The value of a state-action pair can now be estimated as the average discounted future reward we’ve seen from that state action pair.

1.3.5. Observation pre-processing. Humans don’t try to learn maps from the activation of the rods and cones in our eyes to the correct action to take. We have dedicated areas of our brain for recognising objects in our field of vision, and then we make decisions based on what we saw. Likewise, it might not make sense for an RL agent to try to make decisions directly from its observations: we might want to do some kind of input preprocessing, which is a map from the input space to some smaller or more useful space, and then do RL algorithms on that smaller space. This is called feature reinforcement learning.

This is one nice application of unsupervised learning to reinforcement learning. If we run an unsupervised learning algorithm on our inputs to learn a variety of features in them, we can give the compressed version to our agent.
Note that this represents again another assumption about the environment: that it can be meaningfully compressed by an unsupervised learning technique. Sometimes it works, sometimes it doesn’t.

One major problem with preprocessing the input without regard to the actual task is that the unsupervised learning might dedicate a lot of effort to describing areas of the data which don’t matter to the task at hand. Perhaps the task lets you ignore large swathes of the input and only concentrate on one small part of it.

It would be a lot nicer to mix this step with the actual task, so that the agent can work on learning how to read the input at the same time as it is getting some idea of what aspects of the input are important.

1.4. Neural nets. So we want to come up with good ways of mapping from vectors to vectors. Whatever the function is, it needs to be fast and simple to manipulate mathematically.

The obvious function from vectors to vectors is a linear transformation represented by $f(\cdot) = A \cdot + b$. That can be viewed as a bunch of connections from input nodes to output nodes, where the entry $A_{i,j}$ is the strength of the connection between the $i$th input node and the $j$th output node, and $b_j$ is the natural output of the $j$th output node.

Back in the early days of machine learning, this reminded researchers of how the human brain works: we have neurons with different strengths of connections between each other. So this kind of model, made up of matrix multiplications, was called a neural network.

Human neural networks also have non-linearities in their outputs. This turns out to be vital to recognising complex patterns. We represent this by putting in non-linear functions over the outputs of the neural network. We write that function as $s$.

1.4.1. Logistic regression. Previously, we were describing our classification function as a function from our input set $\mathcal{X}$ to our label set $\mathcal{Y}$. One way of implementing this classification function is to implement a ‘fit’ function, from samples in $\mathcal{X}$ to vectors in $\mathbb{R}^{1}$, where the $i$th component of the vector is some measure of how well the input fits into the $i$th category.
in \( Y \). We then classify the input into the category which has the highest fit score.

If we look at the problem this way, we get the added advantage that we can talk about how well inputs fit into categories, not just which category we assigned to them. In fact, we can interpret the output vector of the logistic regression as a posterior probability vector, by running it through a function like the vector sigmoid function \( s \), which takes any vector in \( \mathbb{R}^n \) to a vector in \( \mathbb{R}^n \) whose L1 norm is one.

Logistic regression is a linear model that works along these lines. Our fit function is just a linear transformation:

\[
LR(\vec{x}) = s(W \vec{x} + \vec{b})
\]

So our model is just a matrix \( W \) of size \( |\mathcal{X}| \times |\mathcal{Y}| \), and a bias vector \( b \) of size \( |\mathcal{Y}| \).

The next question is how we train this model to become better. Before we can do that, we need to establish a measure of how good the model is, which we’ll call a loss function: the goal of our learning agent is to minimize their loss. One obvious such loss function would be the proportion of input samples that it classifies incorrectly. However, that loss function
is not differentiable with respect to the model, which means that we can’t use it directly to
train the model.

Instead, we define loss to be the likelihood of the observed label given the input:

\[ L(\theta, \vec{x}, y) = -\log(LR(\vec{x})_y) \]

We don’t want our model to overfit the data: that is, we want it to generalize well to
inputs it hasn’t seen, potentially at the expense of maximising its likelihood of observed
data. One way to do this is to add in a term corresponding to the simplicity of the model,
for example the L1 norms of \( W \) and \( b \). This is called a regularization parameter.

\[ L(\theta, \vec{x}, y) = -\log(LR(\vec{x})_y) + R(\theta) \]

Instead of talking about the loss with respect to a single input \( \vec{x} \) and label \( y \), we can talk
about the loss over a whole dataset of inputs \( D \) and outputs \( \vec{y} \). That involves altering the
previous equation to:

\[ L(\theta, D, \vec{y}) = \sum_{i \in |D|} L(\theta, D_i, \vec{y}_i) + R(\theta) \]

So the optimal model can be written as:

\[ \theta^*(D, \vec{y}) = \arg\min_{\theta} (L(\theta, D, \vec{y})) \]

Solving that argmin is really hard in practise, because the model is in the space \( (\mathbb{R}^{|x|+|y|} \times \mathbb{R}^{|y|}) \), which is really big. So we have to resort to standard mathematical optimization tech-
niques.

One way of solving mathematical optimization problems is the gradient descent algo-

rithm, shown in figure 3. In this algorithm, we take successive steps towards a better cost of
the model \( \theta \). This method is slow because it involves calculating the cost of the model with
while last update was bigger than $\epsilon$ do
  $W \leftarrow W - \alpha \frac{\partial L(\theta, D, \vec{y})}{\partial W}$
  $b \leftarrow b - \alpha \frac{\partial L(\theta, D, \vec{y})}{\partial b}$
end while

**Figure 3.** Gradient descent

<table>
<thead>
<tr>
<th>for input $\vec{x}$ and label $y \in (D, \vec{y})$ do</th>
</tr>
</thead>
<tbody>
<tr>
<td>$W \leftarrow W - \alpha \frac{\partial L(\theta, \vec{x}, y)}{\partial W}$</td>
</tr>
<tr>
<td>$b \leftarrow b - \alpha \frac{\partial L(\theta, \vec{x}, y)}{\partial b}$</td>
</tr>
<tr>
<td>end for</td>
</tr>
</tbody>
</table>

**Figure 4.** Stochastic gradient descent

respect to the entire data set every time.

Alternatively, we can take a single example and update the model to reduce its cost from just this input. This is called stochastic gradient descent. See figure 4.

It turns out to be more computationally efficient if we use stochastic gradient descent with respect to a small set of vectors. We call this collection of typically 10-50 vectors a minibatch. That’s called minibatch gradient descent. If we had the minibatch size be 1, it would be equivalent to stochastic gradient descent, and if the minibatch size were the size of the entire data set, it would be equivalent to the usual gradient descent.

Logistic regression is a linear classifier. This means that its fit function is a linear function of the input. It can thus only solve classification problems when you can draw a hyperplane between the different categories. Lots of things aren’t linearly separable: the simplest example is the xor function.

1.4.2. **Multilayer perceptrons.** Multilayer perceptrons are a simple classifier which can distinguish data which isn’t linearly separable. Instead of mapping directly from inputs to outputs, we map linearly from the inputs to an internal, ‘hidden’ layer. The values of this hidden layer are then all put through a non-linear activation function. The output is a linear
combination of these hidden layer values.

\[ \vec{y} = W_2 \cdot s(W_1 \cdot \vec{x} + \vec{b}_1) + \text{vec}b_2 \]

where \( s \) is some non-linear function. Common choices include \( \tanh \) and \( (1 + e^x)^{-1} \).

The power in this comes from the non-linearity. Without the non-linearity, the compositions of two linear transformations is obviously a linear transformation.

We train MLPs by basically the same means as we train logistic regression: optimization of the parameters by some normal optimization method.

Of course, we can have multiple internal layers, each of which feeds into the next after a non-linearity, eg in figure 6. This doesn’t increase the space of models which the multilayer perceptron can learn, but it does improve the multilayer perceptron's ability to perceive complex patterns.
1.4.3. **Denoising autoencoders.** Both LR and MLP are supervised learning models. Denoising autoencoders (dAs) are an unsupervised learning model.

In a denoising autoencoder, the neural net is given as input a noisy version of some input. Its task is to reconstruct the original input. Mathematically, it’s a map like this, from $\mathcal{X}$ back to $\mathcal{X}$:

$$\tilde{x}' = s(W_2 \cdot s(W_1 \cdot n(x) + b_1) + b_2)$$

(12)

It has four parameters, two matrices of size $|\mathcal{X}| \times |\mathcal{X}|$ and one vector in $\mathbb{R}^{|\mathcal{X}|}$. As above, $s$ is a logistic function. $n$ is a noise function. Typically the noise function works by randomly setting some proportion of the input vector components to 0. 30% seems to be a fairly good amount of noise to add for the problems we looked at.

Because we want the overall transformation to take a vector back to itself, it might make sense to have the second weight matrix just be the transpose of the first. This makes the denoising autoencoder much more computationally efficient.
We can measure the distance between the input and the reconstructed input a few different ways. The most obvious is just to take the L1 or L2 norm of their difference. Alternatively, if our vectors are in $[0, 1]^d$, we can define the cross entropy loss. If we interpret the vector as a vector of probabilities, then the cross entropy loss from vector $\vec{x}$ to $\vec{y}$ is the entropy of the distribution $y$ if we think that we’re actually looking at $x$. The equation is:

$$L_H(\vec{x}, \vec{z}') = - \sum_{k=1}^{\dim \vec{x}} \vec{x}_k \log \vec{z}'_k + (1 - \vec{x}_k) \log(1 - \vec{z}'_k)$$

In a dA, the noise functions as a kind of regularization. If there were no regularization, and the hidden layer was as large as the input layer, the function would try to learn the identity function, which isn’t useful. The addition of random noise gives it an incentive to actually try to learn features of the input.

So each hidden node can be described by a vector of how strongly each input pixel is fed into that node. We can draw a hidden layer visually by drawing, for every pixel in our input images, the coefficient of that pixel in the hidden layer. If we draw a few different hidden nodes like this, you see things like in figure 8. Most layers are a collection of nearby pixels which the dA has learned are strongly correlated and so are useful for compressing
the input image.

**Figure 8.** Layers in a denoising autoencoder.

If we run the denoising autoencoder on some input images, we get output like in figure 9.

**Figure 9.** Denoising digits.

1.4.4. dAs for classification. dAs are good at finding features of our input which are highly correlated and useful for describing it. Because of this, we can use dAs for supervised learning by adding a logistic regression layer, and having the cost of the model overall be a sum of the supervised cost (i.e., the likelihood of the observed classification) and unsupervised cost (the distance between the input and the reconstructed input). The dA is simultaneously learning to make a good denoising filter, and learning a logistic regression on the hidden layer, which is hopefully easier to classify from.
The weighting between supervised cost and unsupervised cost is called the reward modulation. The whole point of this report is varying that mix and seeing what happens.

It is time consuming to train multiple layers of a neural network. Because of this, it is common to eventually stop training the hidden layer and just translate the final logistic regression layer when using dAs for classification. This is called fine-tuning. It is basically the same as varying the mix between supervised and unsupervised cost with a step function. The usefulness of fine-tuning is evidence for the plausibility of varying reward modulation as a means of improving the performance of neural nets.

Just as we could stack hidden layers in MLPs, we can stack dAs, as shown in figure 10.

1.5. **Reward-modulated inference.** The proposal in [Sunehag and Hutter, 2014] which this report was investigating is that varying the balance between the supervised and unsupervised costs in an agent. This seems intuitively plausible for a variety of different reasons. For example, in the bandit feedback classification problem, the agent is getting much less
supervised feedback at the start of the experiment.

To express this in an equation, we’re using a cost function made of a supervised cost $l_{\text{supervised}}$ and an unsupervised cost $l_{\text{unsupervised}}$, and we want to make a single cost function $l$. We’re considering different functions of time $m(t)$, such that

$$l(x, t) = m(t) \cdot l_{\text{supervised}} + (1 - m(t)) \cdot l_{\text{unsupervised}}$$

(14)

This unifies the two mechanisms of the supervised and unsupervised learning with just one mechanism, which is conceptually simpler and more elegant.

1.6. Engineering a simpler neural net library. I spent a bit of time writing a nice object-oriented wrapper for neural net layers. Theano’s abstraction goes to the level of functions, which seems lower level than it needs to be. These neural nets can usually be expressed as a combination of layers. My code abstracts away the layers. For example, a complete listing of my linear regression code is provided in figure 11.

The essential abstractions provided by my code are:

- A SnazzyNet class, which represents an abstract neural net. Classes which inherit from it must define instance variables input, output, predictor and cost. The code can change from receiving full feedback to bandit feedback just by changing the predictor and cost. The class defines a learn method, which takes some input and learns from it, a get_regression method which takes an input and returns the output from the final layer in the neural net, and an a get_classification method which just argmaxes the previous method.

- A SnazzyLayer class, which represents an abstract layer. Classes which inherit from it must define input_object, which must also be a SnazzyLayer object, output, which is a Theano expression, and output_dimension, which is used to check that the dimensions of all the objects fit together properly.

I also wrote a few layers to be used with this code, such as a LinearTransformationLayer and a TanhLayer.
from SnazzyNets import *
import numpy as np
import random

class LogisticRegressionCBNet(SnazzyNet):
    def __init__(self, input_dimension, output_dimension, learning_rate = 0.5):
        self.input = InputVector(input_dimension)
        self.output_layer = SigmoidTransformationLayer(self.input, output_dimension)

        self.predictor = RewardFeedbackVector(output_dimension)

        self.cost = (self.predictor.distance_loss(self.output_layer) +
                     0.001 * self.output_layer.l1_regularization())

        self.learning_rate = learning_rate

        SnazzyNet.__init__(self)

class LogisticRegressionFullFeedbackNet(LogisticRegressionCBNet):
    def __init__(self, input_dimension, output_dimension, learning_rate = 0.5,
                 learning_rate_half_time=np.inf):
        LogisticRegressionCBNet.__init__(self, input_dimension, output_dimension,
                                          learning_rate, learning_rate_half_time)

        self.predictor = FullFeedbackVector(output_dimension)

        self.cost = (self.predictor.negative_log_loss(self.output_layer) +
                     0.001 * self.output_layer.l1_regularization())

        self.compile_functions()

Figure 11. Code for logistic regression with my neural nets library

With some expansion, this code would make it a lot easier to play around with different
neural net structures for this kind of problem.

2. EXPERIMENTAL DESIGN

The goal of these experiments was to determine whether RMI improves the accuracy of
a learning agent on classification problems.
3. Results

Before investigating RMI, we need to establish some good parameters for the agent, such as its learning rate and the number of nodes in its hidden layer. Here are graphs varying those two variables. Turns out that 700-1300 nodes is pretty much optimal, and a 0.1 learning rate works nicely. See figure 2.

Firstly, let’s try a step RMI curve. This is likely to work, because of the connection to fine tuning which we are aware of from the literature.

Now, let’s try using a hyperbolic RMI curve on this. That is,

\[ m(t) = \frac{1}{1 + kt} \]
FIGURE 13. Performance on MNIST while varying learning rates in dA with LR layer and 700 hidden nodes.

How well does it work?

INSERT GRAPHS HERE. Preliminary results show that it’s good, preferably with sigmoid being evenly balanced at 1 epoch.

Linear reward modulation is when the reward modulation is a linear function over time:

\[ m(t) = \max(0, 1 - kt) \]

Let’s see how that goes.
4. Discussion

We can see clearly from the results that step based RMI works nicely with this supervised learning task. We expected as such, just because it's commonly known that fine-tuning improves the performance of dAs on this kind of problem. It seems to work better with a later step. This is to be expected, because that basically equates to training the dA as a whole for longer before you go to fine tuning.

Hyperbolic RMI works very well. This is an exciting result. Values of $k$ from 0.1 to 1 seemed to work best.

A value of 10 seemed to work best for linear RMI.

All of these results were better than any of the results for a classifying denoising autoencoder which we found in figure 12.
4.1. **Further research.** RMI seems to work very well on some problems. In particular, we expect RMI to have a big advantage in highly structured, high dimensional problems which require some kind of input preprocessing which can be done before the agent knows how to navigate effectively in the environment. I would like to further investigate the following questions:

- How well does RMI work on non-full-feedback environments?
- Is there anything to be gained from using more complex, stacked neural nets to get more general views of the environment?
- Are there better sets of hyperparameters which we could be using?

5. **Conclusion**

Reward modulated inference is a fairly new way of learning from environments in both a supervised and unsupervised fashion simultaneously. Experiments indicate that it seems to
Figure 16. Performance on MNIST while varying $k$ in linear RMI.

work well on some tasks. More research is required to understand how best to use it.

References


Peter Sunehag and Marcus Hutter. Intelligence as inference or forcing Occam on the world. In Ben Goertzel, Laurent Orseau, and Javier Snaider, editors, Artificial General Intelligence,

