Investigating Reward Modulation in Denoising Autoencoders

Matthew Alger
The Australian National University

In this report, we describe basic concepts of machine learning, and introduce and test a mechanism of combining supervised and unsupervised learning called reward modulation. Having reward modulation of 0 means that training is fully unsupervised, and having modulation of 1 means that training is fully supervised.

Tests were performed by changing reward modulation over time in three different ways: as a step function (equivalent to the common machine learning technique of fine tuning), as a linear function, and as a hyperbolic function. This was applied to denoising autoencoders, which were then used to classify MNIST images and solve simple MDPs.

While changing reward modulation over time seems to have a noticable impact on the classification task, there is no obvious impact on the MDP tasks. This may be due to the small size of the problems involved, hence more testing is required on larger grids and mazes.

Introduction

Machine learning algorithms are a class of algorithms which learn a model of a class of data given some set of input data. This model may be used to accomplish a variety of tasks, such as classification and regression.

There are three main forms of machine learning. These are supervised learning, unsupervised learning, and reinforcement learning.

This report will outline some common concepts in machine learning, and introduce a mechanism of combining supervised and unsupervised learning called reward modulation. Note that this is not the same as semi-supervised learning — reward modulation requires labelled information for all inputs, whereas semi-supervised learning does not. This is because the motivations behind semi-supervised learning and reward modulation differ: With semi-supervised learning, the computer is trying to learn from an incomplete set of data, but with reward modulation, we are trying to find a better solution by requiring that any parameters we find must represent a good model of the inputs.

Supervised Learning

Supervised learning is a form of machine learning where the computer is given as input a set of pairs, with each pair representing an input and the desired output. The computer is then tasked with learning a map between the inputs and the outputs. Examples of supervised learning problems include classification, regression, and sequence prediction.

Classification

Classification is a task where the computer aims to assign the correct label to a given vector. Once the computer has been trained, it can be given a vector and it will return the best label it can find. A notable example of a classification task is handwritten digit recognition, in which the computer is given an image of some handwritten digit, and it must then assign as a label the digit that the image represents. The MNIST dataset (LeCun et al., 2010), shown in figure 1, is commonly used as input for this task.

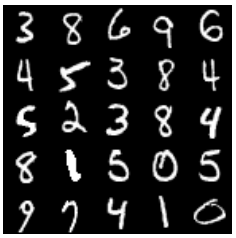


Figure 1: The MNIST dataset contains 70000 28 $\times$ 28 px images of handwritten digits, which can each be interpreted as a vector in $\mathbb{R}^{784}$.

In training for a classification task, the computer is given a set of learning data $\mathcal{D}$. This consists of pairs of input vectors and their associated labels, which we will refer to as $\vec{x}$ and k respectively. It then attempts to find a good map between input vectors and labels. The exact method of this learning process depends on the algorithm used.

Regression

Regression is a task where the computer aims to find the relationship between one or more input variables and a real-valued output variable. After the computer has been trained, it can be given values of the input variables, and it will return the value of the dependent variables. This is identical to the problem of classification, but with a real-valued output instead of discrete categories.

During training, the computer is given a set $\mathcal{D}$ of pairs of values of independent variables and the associated values of the dependent variables. These are both represented as vectors, called $\vec{x}$ and $\vec{y}$ respectively. The computer then attempts to find a function approximating the relationship between independent and dependent variables. As with classification, the exact method of this function approximation depends on the algorithm used.

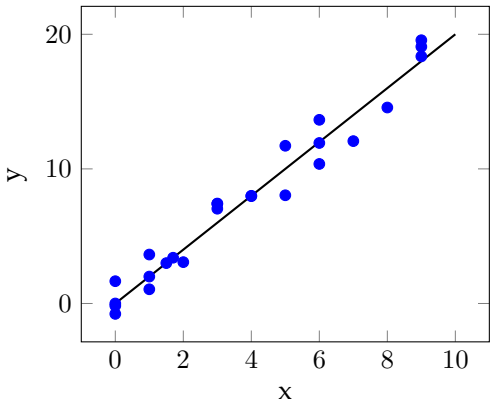


Figure 2: A linear regression finds a line of best fit for some data points.

The accuracy of the learned function depends heavily on the method of learning used. For example, the linear regression algorithm can only find linear relationships.

Sequence Prediction

Sequence prediction is a task where the computer is given a vector representing the current state of some system, and aims to predict what the next state of this system will be. Once the computer has been trained, it can be given one state of a system and will return a prediction of the next state. During training, the computer is supplied a set of pairs of states and the associated next states, which it will learn a map between. An example of a state prediction task would be predicting the effects of gravity on some object, where the computer is supplied the position and velocity of the object and is tasked with predicting the subsequent position and velocity of the object.

Unsupervised Learning

Unsupervised learning is a form of machine learning where the computer is only given input data, without any associated expected output data, and is tasked with finding patterns in it. Examples of unsupervised learning problems that will be covered in this report include model learning and denoising.

Model Learning

Model learning is a task where the computer aims to find a different representation of some class of input data. This representation either provides a more abstract form of input data, or simply provides a different form of the input data (for example, the representation may be simply a higher dimensional projection of the input data, allowing it to be more easily separated for a classficiation problem). In both cases, the new representation may allow other machine learning algorithms to work more effectively, by using the new representation as input instead of the raw data.

The computer is trained by supplying it with a set of input data, which it then learns to identify key features of. Usually, the computer will have some secondary task to accomplish which will be used to validate how good the model is — for example, denoising autoencoders attempt to learn to reduce noise in input data, and learn a model as a side-effect.

Examples of algorithms that learn models are autoencoders and restricted Boltzmann machines.

Denoising

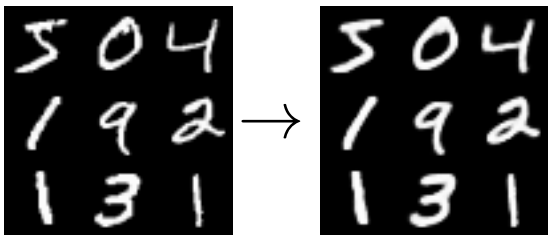


Figure 3: Original MNIST digits, and their denoised versions generated by a denoising autoencoder.

Denoising is a task where the computer learns to accept input data and return a less noisy version of that data, such as the example in figure 3. This is closely related to model learning, as the algorithms used to denoise data also learn a model as a side-effect.

The model is usually a transformation from the input data to the other representation of the input data. This transformation encodes the general features learned from the class of input data.

Denoising autoencoders are an example of a denoising algorithm.

Reinforcement Learning

Reinforcement learning is a form of machine learning where an agent is in an environment where it observes the state of the environment and may then take a number of actions. Each action changes the state of the environment in some way. The agent receives some scalar value called a reward after each action. The aim is to maximise long-term accumulated reward.

The reinforcement learning tasks described here are represented by Markov decision processes.

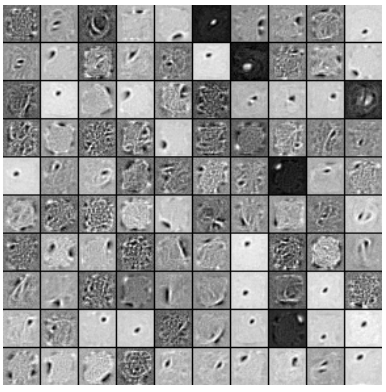


Figure 4: Part of the transformation learned by the denoising autoencoder in figure 3. These images are element-wise multiplied by input data and added together to get the learned representation of the input data.

Markov Decision Processes

A Markov decision process (MDP) is a set of states $\mathcal{S}$, a set of actions $\mathcal{A}$ available at each state, a (possibly non-deterministic) function $s : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$ taking a state and an action to another state, and a reward function $R : \mathcal{S} \to \mathbb{R}$. They model environments in which an agent can take actions to move through the states, and eventually receive some kind of reward.

An example of a MDP is the mountain car problem. The agent is a car in a valley, and it can accelerate left, right, or not at all at any point in time. If the car reaches a certain height, it can exit the valley, and receive a reward. However, the car cannot accelerate fast enough to escape the valley under its own power. The aim of the car is to reach a certain height up the side of the valley, a goal which it cannot achieve without utilising the shape of the valley to gain enough speed. When the car reaches the given height, it receives a reward. The state here comprises the position and speed of the car.

For learning methods that require a "reward" at each state proportional to how good that state is, we can run the MDP until the end state is reached, and then assign each state a value based on future rewards. This is called the value of a state. This can be given as the instantaneous reward of the state itself, plus some constant $0 < \gamma \leq 1$ multiplied by the value of the next state. In this way, a value is assigned to each state. This is described in equation 1, where $\vec{\boldsymbol{x}}_i$ is the ith state, $V(\vec{\boldsymbol{x}}_i)$ is the reward assigned to that state, and $R(\vec{\boldsymbol{x}}_i)$ is the instantaneous reward at that state.

$$V(\vec{\boldsymbol{x}}_i) = R(\vec{\boldsymbol{x}}_i) + \gamma V(\vec{\boldsymbol{x}}_{i+1}) \tag{1}$$

In general, the value of the state $\vec{\boldsymbol{x}}$ taking the action k may be written $V(\vec{\boldsymbol{x}}, k)$.

Solutions to MDPs with a discrete, finite set of possible actions can be found using methods similar to those used in classification problems. The state vector is given as input to some algorithm, and the "labels" returned by the algorithm are interpreted as actions. By restricting the reward given to the range $[0, 1]$, we can interpret the expected probabilities in the output vector as expected rewards, and require the computer to minimise the difference between expected and actual reward.

In a similar way, techniques used in regression can also be applied to MDPs with continuous actions.

Algorithms and Implementation

There are many different ways to implement supervised learning, unsupervised learning, and reinforcement learning problems. Some algorithms for these tasks include logistic regression, multi-layer perceptrons, autoencoders, and denoising autoencoders.

Logistic Regression

Logistic regression is a form of supervised learning which learns to classify input data into discrete categories, called labels.

Like all supervised learning, the input data is a set of pairs of corresponding inputs and outputs. A set of such input data is denoted $\mathcal{D}$ and an element in the input data is denoted $(\vec{\boldsymbol{x}}, k)$. $\vec{\boldsymbol{x}}$ represents the input vector and k represents the corresponding output label.

To classify an input $\vec{\boldsymbol{x}}$, logistic regression applies a linear transformation and a translation to $\vec{\boldsymbol{x}}$. The resulting output is then passed through the softmax function, such that each component of the final output is in the range $(0, 1)$, and the sum of all components is 1. We call this output $\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}})$, and it is defined in equation 2, where W is

called the weight matrix and $\vec{\boldsymbol{b}}$ is called the bias vector. Together, we refer to W and $\vec{\boldsymbol{b}}$ as the model, and denote it θ. $\vec{\boldsymbol{y}}$ is entirely defined by θ, so we denote $\vec{\boldsymbol{y}}$ for some specific θ as $\vec{\boldsymbol{y}}_\theta$.

$$\vec{\boldsymbol{y}}_\theta(\vec{\boldsymbol{x}}) = \text{softmax}(W\vec{\boldsymbol{x}} + \vec{\boldsymbol{b}}) \tag{2}$$

The entries of $\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}})$ look a lot like probabilities, so we interpret them as such: The ith entry of $\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}})$, denoted $y(\vec{\boldsymbol{x}})_i$, is the predicted probability that $\vec{\boldsymbol{x}}$ carried the ith label. As such, classification is simple: The predicted label, l, is the value of i such that $y(\vec{\boldsymbol{x}})_i$ is maximised[1], as shown in equation 3.

$$l_\theta(\vec{\boldsymbol{x}}) = \operatorname*{argmax}_i(y_\theta(\vec{\boldsymbol{x}})_i) \tag{3}$$

The aim of logistic regression, then, is to *find* θ such that the number of incorrect labellings is minimised. One way to represent this is as a loss function $\mathcal{L}(\mathcal{D}, \theta)$. If θ is optimal, then $\mathcal{L}(\mathcal{D}, \theta)$ will be 0 for any $\mathcal{D}$. We can write $\mathcal{L}$ as the average number of incorrectly labelled inputs, as shown in equation 4.

$$\mathcal{L}(\mathcal{D}, \theta) = \frac{1}{|\mathcal{D}|} \sum_{(\vec{\boldsymbol{x}}, k) \in \mathcal{D}} \begin{cases} 1 & l_\theta(\vec{\boldsymbol{x}}) = k \\ 0 & l_\theta(\vec{\boldsymbol{x}}) \neq k \end{cases} \tag{4}$$

Minimising the loss function would then give a suitable θ. However, this loss function is not differentiable, meaning that the common methods of minimisation based on differentiation will not work. To this end, we instead design a differentiable loss function based on the probability predictions in $\vec{\boldsymbol{y}}$. Good values for θ will result in correct labels being assigned high probabilities in $\vec{\boldsymbol{y}}$. We can invert this, and instead look for values of θ such that $y(\vec{\boldsymbol{x}})_k \to 1$. To phrase this as a minimisation problem, we can take the negative log of $y(\vec{\boldsymbol{x}})_k$ — as $y(\vec{\boldsymbol{x}})_k \to 1$, $-\log y(\vec{\boldsymbol{x}})_k \to 0$. This is described in equation 5. We call this particular loss function the negative log-likelihood.

$$\mathcal{L}(\mathcal{D}, \theta) = \sum_{(\vec{\boldsymbol{x}}, k) \in \mathcal{D}} -\log y_\theta(\vec{\boldsymbol{x}})_k \tag{5}$$

We now want to find θ. This can be accomplished by a process called gradient descent, in which we repeatedly change θ based on how much it affects the loss across some data set $\mathcal{D}$. This is described in equation 6. The value r is called the learning rate.

$$\theta_{i+1} = \theta_i - r \sum_{(z \in \mathcal{D})} \frac{d\mathcal{L}(\{z\}, \theta)}{d\theta} \tag{6}$$

Gradient descent requires iterating over the entire data set $\mathcal{D}$ for each epoch. This is sometimes slow, so we might instead use stochastic gradient descent for each item $z \in \mathcal{D}$ (equation 7). This is sometimes faster at the cost of accuracy.

$$\theta_{i+1} = \theta_i - r \frac{d\mathcal{L}(\{z\}, \theta)}{d\theta} \tag{7}$$

These methods can also be combined using batches. A subset of the data set, called a batch, is used with equation 6 to train θ. This is then repeated until all data has been used to train θ. The process is then repeated.

Since logistic regression is essentially a linear transformation, it can only accurately classify data that are linearly separable — that is, data in n dimensions that can be split into categories by an n-dimensional hyperplane.

[1]Note that we don't really need to apply softmax at all to classify, since the largest entry in $\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}})$ is at the same index whether or not we apply softmax. However, applying softmax allows us to interpret the entries as probabilities, which is useful for other tasks.

The concepts behind logistic regression can be easily extended to other supervised learning problems. For example, by removing the softmax from the definition of $\vec{y}$, we can perform regression with $\vec{y}$ as the output vector.

Multilayer Perceptrons

Multilayer perceptrons are similar to logistic regression in that they also classify input data into discrete categories, but they are more powerful: They can also classify data that are not linearly separable. This is accomplished with the use of what are called hidden layers — an input vector $\vec{x}$ is non-linearly mapped to a "hidden" vector $\vec{h}$, which is non-linearly mapped again to obtain an output vector $\vec{y}$. This mapping can be repeated for arbitrarily many hidden layers. These non-linearities are what allow multilayer perceptrons to classify non-linearly separable data.

Like logistic regression, a multilayer perceptron can also be represented by a model θ of weight matrices $W_1, W_2, \cdots, W_n$, and bias vectors $\vec{b}_1, \vec{b}_2, \cdots, \vec{b}_n$. A multilayer perceptron with n hidden layers taking an input vector $\vec{x}$ to an output vector $\vec{y}_\theta(\vec{x})$ would be represented by the following equations, where s is a non-linear function such as tanh or sigmoid and $\vec{h}_{n_\theta}(\vec{x})$ is the nth hidden layer:

$$\vec{h}_{1_\theta}(\vec{x}) = s(W_1\vec{x} + \vec{b}_1) \tag{8}$$
$$\vec{h}_{2_\theta}(\vec{x}) = s(W_2\vec{h}_{1_\theta}(\vec{x}) + \vec{b}_2) \tag{9}$$
$$\vdots$$
$$\vec{y}_\theta(\vec{x}) = s(W_{n+1}\vec{h}_{n_\theta}(\vec{x}) + \vec{b}_{n+1}) \tag{10}$$

The cost function is the same as for logistic regression — the negative 'log-likelihood '(equation 5) — and gradient descent is again used with this cost function on θ.

Denoising Autoencoders

An autoencoder is a form of unsupervised learning, where the input vector $\vec{x}$ is acted upon non-linearly to produce an output vector $\vec{x}'(\vec{x})$, called the reconstruction of $\vec{x}$. This is essentially a multilayer perceptron, with an autoencoder with one hidden layer described by equations almost identical to equations 8 and 10:

$$\vec{h}_\theta(\vec{x}) = s(W_1\vec{x} + \vec{b}_1) \tag{11}$$
$$\vec{x}'_\theta(\vec{x}) = s(W_2\vec{h}(\vec{x}) + \vec{b}_2) \tag{12}$$

where the model θ describes both $\vec{h}_\theta$ and $\vec{x}'_\theta$, and comprises weight matrices W_1 and W_2, and bias vectors $\vec{b}_1$ and $\vec{b}_2$. A common restricting is setting W_2 such that $W_2 = W_1^T$. This is called tied weights.

The autoencoder is designed to learn a map between the input $\vec{x}$, some number of hidden layers $\vec{h}_1, \vec{h}_2, \cdots, \vec{h}_n$, and an output $\vec{x}'$, such that $\vec{x} = \vec{x}'(\vec{x})$. The key idea behind this is that the hidden layers will be some alternate, more abstract representation of the input data, from which the data can be reconstructed as $\vec{x}'$. The hidden layers can then be used as input to some other learning process, such as logistic regression.

One possible cost function would be as follows:

$$\mathcal{L}(\mathcal{D}, \theta) = \sum_{\vec{x}\in\mathcal{D}} ||\vec{x} - \vec{x}'(\vec{x})|| \tag{13}$$

Another common cost function is the reconstruction cross-entropy (Vincent et al., 2008). This is shown in the following equation, with d as the dimension of the input:

$$\mathcal{L}(\mathcal{D}, \vec{x}') = -\sum_{i=0}^{d-1} \Big(x_i \log x'_i(\vec{x}) + (1 - x_i) \log(1 - x'_i(\vec{x})) \Big) \tag{14}$$

Clearly, as $\vec{x}'(\vec{x})$ grows more similar to $\vec{x}$, this value will decrease, making it a suitable minimisation function.

This approach may not work as well as intended - the autoencoder may simply learn the identity function (Vincent et al., 2008; Bengio, 2009), which would not result in a useful abstraction $\vec{h}$. To avoid this, we introduce an extra step: We add noise to the input vector, and then task the autoencoder with denoising it. This is described in equations 15 and 16, once again for a single hidden layer autoencoder, where $\vec{n}$ is a mask vector of random ones and zeros and $\vec{x} * \vec{n}$ is the element-wise product of $\vec{x}$ and $\vec{n}$.

$$\vec{h}_\theta(\vec{x}) = s(W_1(\vec{x} * \vec{n}) + \vec{b}_1) \tag{15}$$

$$\vec{x}'_\theta(\vec{x}) = s(W_2\vec{h}(\vec{x}) + \vec{b}_2) \tag{16}$$

This is called a denoising autoencoder. Using the transformation described by θ in a regular autoencoder will result in denoised versions of the input vector $\vec{x}$, such as those shown in figure 3.

Denoising autoencoders are more effective than non-denoising autoencoders at generating alternative representations of inputs (Vincent et al., 2008).

Denoising autoencoders can also be stacked to improve their abstractions. In this case, the hidden layer from a denoising autoencoder is used as input to a second denoising autoencoder, which may then have its hidden layer used as input to a third, and so on.

Reward Modulation

As previously stated, denoising autoencoders can be used to perform supervised tasks by using their hidden layer as input to a supervised algorithm such as logistic regression. The denoising autoencoder is then trained in the usual way, alongside the logistic regression. This kind of combined supervised–unsupervised learning can be far more effective than using purely supervised learning (Larochelle et al., 2012; Deep Learning, 2014)

A common technique when using this method of supervised learning is called fine tuning. Fine tuning is the technique of training the supervised and unsupervised components of the algorithm separately for some amount of time, and then training the whole algorithm exclusively on the loss from the supervised component. This has good results (Bengio, 2009; Deep Learning, 2014), but begs the question of whether there is a benefit from instead *smoothly* changing from unsupervised to supervised learning over time.

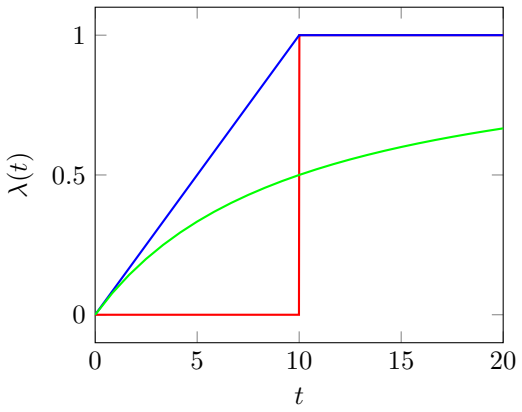


Figure 5: Different modulation functions $\lambda(t)$. Step is red, linear is blue, and hyperbolic is green.

This is reward modulation. If we write the denoising autoencoder's original loss function as $\mathcal{F}_\theta$, and the logistic regression loss function as $\mathcal{R}_\theta$, then the loss function of the denoising autoencoder becomes

$$\mathcal{L}(\mathcal{D}, \theta) = (1 - \lambda(t))\mathcal{F}_\theta + \lambda(t)R_\theta \tag{17}$$

where $\lambda(t)$ is a function that takes the current epoch and returns the modulation, a real number in the range $[0, 1]$. If $\lambda(t) = 0$, we have effectively entirely unsupervised learning. If $\lambda(t) = 1$, we have supervised learning only. If $\lambda(t) = \begin{cases} 0 & t < 50 \\ 1 & t \geq 50 \end{cases}$, then we recover the process of fine tuning described above.

In this report we investigate three forms of $\lambda(t)$. These are listed below, with p as some constant parameter:

- λ as a step function, $\lambda(t) = \begin{cases} 0 & t < p \\ 1 & t \geq p \end{cases}$
- λ as a linear function, $\lambda(t) = \frac{t}{p}$
- λ as a hyperbolic function, $\lambda(t) = 1 - \frac{p}{p+t}$

The value of $\lambda(t)$ is constrained to the range $[0, 1]$. These are plotted in figure 5 with $p = 10$.

We aim to find whether changing reward modulation over the course of training improves the quality and speed of solutions to various machine learning tasks.

Experimental Design

We conducted three experiments: one classification experiment, and two MDP experiments.

Classification Task — Digit Recognition

Outline

The first experiment was a classification task, performed on the MNIST dataset (LeCun et al., 2010). The goal of the task was to correctly classify images from the MNIST dataset as the digits they represent.

This experiment was performed using two denoising autoencoders, each with one hidden layer. The first denoising autoencoder was a standard denoising autoencoder, and the second denoising autoencoder (referred to here as the RM denoising autoencoder) changed reward modulation while training.

The denoising autoencoders were trained on inputs $(\vec{\boldsymbol{x}}, k)$, with $\vec{\boldsymbol{x}} \in [0, 1]^{28 \times 28}$ as an image from the MNIST dataset, and $k \in \{0, 1, \cdots, 9\}$ as its associated label. The number of incorrect labellings were recorded after each epoch, and these are plotted in figures 7a, 7b, 7c, and 7d.

Implementation

The denoising autoencoders were implemented using the Python libraries NumPy (van der Walt et al., 2011) and Theano (Bergstra et al., 2010). NumPy was used to manipulate the large amounts of input data as matrices, and Theano was used to symbolically manipulate these matrices to form the denoising autoencoder itself.

Each denoising autoencoder mapped an input $\vec{\boldsymbol{x}}$ to a hidden layer $\vec{\boldsymbol{h}}(\vec{\boldsymbol{x}})$, which was then mapped to a reconstruction of the original input $\vec{\boldsymbol{x}}'(\vec{\boldsymbol{x}})$ using tied weights. A logistic regression was also implemented on top of the denoising autoencoders, mapping the hidden vectors $\vec{\boldsymbol{h}}(\vec{\boldsymbol{x}})$ to output vectors $\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}})$. The denoising autoencoders are described in equations 18 to 20, with W and A as the denoising autoencoder and logistic regression weight matrices respectively, and $\vec{\boldsymbol{b}}, \vec{\boldsymbol{b}}'$, and $\vec{\boldsymbol{c}}$ as the autoencoder, reconstruction, and logistic regression bias vectors respectively.

$$\vec{\boldsymbol{h}}(\vec{\boldsymbol{x}}) = \text{sigmoid}(W\vec{\boldsymbol{x}} + \vec{\boldsymbol{b}}) \tag{18}$$

$$\vec{\boldsymbol{x}}'(\vec{\boldsymbol{x}}) = \text{sigmoid}(W^T\vec{\boldsymbol{h}}(\vec{\boldsymbol{x}}) + \vec{\boldsymbol{b}}') \tag{19}$$

$$\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}}) = \text{softmax}(A\vec{\boldsymbol{h}}(\vec{\boldsymbol{x}}) + \vec{\boldsymbol{c}}) \tag{20}$$

$\vec{\boldsymbol{h}}(\vec{\boldsymbol{x}})$ contained 700 nodes, as this was found during initial testing to be a good balance between solution quality and speed. $y_i(\vec{\boldsymbol{x}})$ represented the likelihood that $\vec{\boldsymbol{x}}$ carried the label i.

The model of each denoising autoencoder was trained by stochastic gradient descent (equation 7) on the weight matrices and bias vectors, using batch sizes of 20. The learning rates r of the denoising autoencoders were a function of the current epoch t, with $r(t) = 0.1 \times \frac{4}{4+t}$, such that the learning rates decreased over time. Each denoising autoencoder was trained for 40 epochs.

After each trial, the model was evaluated by predicting the labels of 10000 test images in the MNIST dataset that were not used to train the model. The percentage of incorrect labels was recorded. The error rates found for the non-RM denoising autoencoder are plotted against epochs in figure 7a, and the error rates found for the RM denoising autoencoders are plotted against epochs in figures 7b, 7c, and 7d.

Both denoising autoencoders used two loss functions — one loss function for the unsupervised component, and one loss function for the logistic regression. The standard denoising autoencoder used the reconstruction cross-entropy for its unsupervised loss function (equation 14), and the negative log-likelihood for its logistic regression loss function (equation 5). The RM denoising autoencoder used the same loss function for the logistic regression component, but used a reward modulating, epoch-dependent loss function for the unsupervised component, which combined the reconstruction cross-entropy loss with the supervised loss from the logistic regression. This is described in equation 23.

$$\mathcal{F}(\mathcal{D}, \vec{\boldsymbol{x}}') = \sum_{i=0}^{9} \left(x_i \log x_i'(\vec{\boldsymbol{x}}) + (1 - x_i) \log(1 - x_i'(\vec{\boldsymbol{x}})) \right) \tag{21}$$

$$\mathcal{R}(\mathcal{D}, \vec{\boldsymbol{y}}) = \sum_{(\vec{\boldsymbol{x}}, k) \in \mathcal{D}} -\log y_\theta(\vec{\boldsymbol{x}})_k \tag{22}$$

$$\mathcal{L}(\mathcal{D}, \vec{\boldsymbol{x}}', \vec{\boldsymbol{y}}) = (1 - \lambda(t))\mathcal{F}(\mathcal{D}, \vec{\boldsymbol{x}}') + \lambda(t)\mathcal{R}(\mathcal{D}, \vec{\boldsymbol{y}}) \tag{23}$$

Three different functions $\lambda(t)$ were trialled. These were step modulation (equation 24), linear modulation (equation 25), and hyperbolic modulation (equation 26).

$$\lambda(t) = \begin{cases} 0 & t < p \\ 1 & t \geq p \end{cases} \tag{24}$$

$$\lambda(t) = \frac{t}{p} \tag{25}$$

$$\lambda(t) = 1 - \frac{p}{p + t} \tag{26}$$

The trials were repeated for different values of the parameter p. These values were 0, 1, 2, 4, 10, 15, and 20 for step modulation; 1, 2, 4, 10, 20, 40, and 80 for linear modulation; and $\frac{1}{2}$, $\frac{5}{2}$, $\frac{9}{2}$, $\frac{13}{2}$, 10, 12, 20, and 200 for hyperbolic modulation. These values were chosen to represent a wide range of values, so it would be possible to determine which region of values is better for use in reward modulation.

MDP Task — Grid

Outline

The second experiment was a navigation task. An agent was placed in a square grid and tasked with navigating to the bottom-right corner.

The agent was given as input the coordinates of its current location. These coordinates were represented by images from the MNIST dataset, such that one input contained two images. The agent then output an action as

an integer in $\{0, 1, 2, 3\}$. The action was interpreted as a direction for the agent to move: 0 for right, 1 for up, 2 for left, and 3 for right. If the agent would ever attempt to move outside the grid, it would instead not move at all.

Each epoch, the agent was allowed to navigate the grid from random starting coordinates. The coordinates of the agent were input to the agent. The output was used to choose the direction to move the agent with probability $1-\epsilon$, where ϵ was a constant parameter. With probability ϵ, the agent would instead move randomly. This process was repeated until the agent reached the bottom-right corner of the grid, or until the agent had made 1000 moves.

Once the agent had finished navigating the grid, a reward was given. This reward was either 0 if the agent did not reach the bottom-right corner, or 1 if it did. This reward was then used to calculate the discounted future reward for each state, as in equation 1, with $\gamma = 0.9$. The agent was then trained based on the state-action-reward data obtained during all previous navigations.

Implementation

The grid was implemented as an object which stored the location of the agent, as well as information about the width and height of the grid.

The agent was implemented in two different ways: firstly with a denoising autoencoder, and secondly with a RM denoising autoencoder as described previously. The autoencoders had 700 nodes in their hidden layers, and (as with classification) used a learning rate of $r(t) = 0.1 \times \frac{4}{4+t}$, where t is the epoch. Each autoencoder had a logistic regression layer implemented alongside it, which used the hidden layer as input. Both autoencoders and their logistic regression layers are described by equations 18 to 20.

The input to the autoencoders was two images from the MNIST dataset, represented by a vector $\vec{\boldsymbol{x}} \in [0, 1]^{28\times28\times2}$. The output vector $\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}})$ from the logistic regression was interpreted as a list of expected rewards, with the ith component of $\vec{\boldsymbol{y}}(\vec{\boldsymbol{x}})$ representing the expected reward gained from taking the ith action at state $\vec{\boldsymbol{x}}$.

The reconstruction cross-entropy (equation 14) was used as the loss function for the denoising autoencoder component of the agents. The average absolute difference between the expected and the actual reward was used as the loss function for the logistic regression component. This is described in equation 27, where $\vec{\boldsymbol{x}}$ is a state vector, k is the action taken at that state, and $R(\vec{\boldsymbol{x}}, k)$ is the reward given from taking action k at state $\vec{\boldsymbol{x}}$.

$$\mathcal{L}(\mathcal{D}, \vec{\boldsymbol{y}}) = \sum_{(\vec{\boldsymbol{x}},k)\in\mathcal{D}} ||y(\vec{\boldsymbol{x}})_k - R(\vec{\boldsymbol{x}}, k)|| \tag{27}$$

The agent was randomly placed into a grid, and navigated the grid as described in the outline. ϵ, the chance that the agent would act randomly, was varied over time as $\epsilon(t) = \frac{1}{1+t}$, where t is the epoch. The states observed over the course of navigating the grid were then stored, along with the corresponding action that the agent took and the discounted future reward that was received as a result.

Previously stored state-action-reward pairs were used as the data set in training. The training was accomplished using stochastic gradient descent (equation 7) with batch sizes of 20.

Validation was accomplished by placing the agent in the top-left corner of the grid, and measuring how many moves it took to reach the goal of the bottom-right corner. During validation, ϵ was set to 0. If the agent made 1000 moves without finding the goal, the validation step would be terminated and a value of 1000 recorded.

Three different forms of reward modulation were trialled in the RM agent. These were step modulation (equation 24) with $p = 50$, linear modulation (equation 25) with $p = 25$ and $p = 50$, and hyperbolic modulation (26) with $p = 50$ and $p = 1$[2].

[2]The parameter p for linear modulation was changed from 50 to 25 between experiments, and the parameter p for hyperbolic modulation changed from both 50 and 1 to just 1 between experiments. This change was made in light of new data from the classification task. Ideally, the older trials would be re-run with the new parameters, but due to the computational power required to run trials, this was unfortunately

The non-RM agent and the RM agents were trained for 60 epochs on a 3×3 grid and a 5×5 grid. The results for these tests are plotted in figure 8a and figure 8b respectively.

MDP Task — Maze

Outline

The third experiment was a harder navigation task. The maze task was identical to the grid task, except that the grid had walls placed between some cells. This formed a maze similar to the maze pictured in figure 6. The walls were generated using a randomised depth-first search.

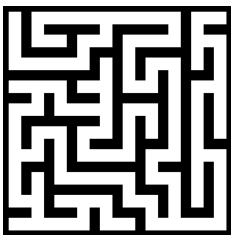


Figure 6: A grid maze.

As with the grid task, the agent was given as input the coordinates of its current location, which were once again represented by images from the MNIST dataset. Actions were again interpreted as directions in which to move. If the agent would ever attempt to move outside the grid or into a wall, it would instead not move at all.

Each epoch, the agent was allowed to navigate the maze from random starting coordinates. Random actions would be taken with probability $\epsilon(t) = \frac{1}{1+t}$ where t is the epoch. The navigation stage stopped when the agent reached the goal, or when the agent had taken more than 1000 moves and had not found the goal.

Once the agent had finished navigating the grid, a reward was given. This reward was either 0 if the agent did not reach the bottom-right corner, or 1 if it did. This reward was then used to calculate the discounted future reward for each state, as in equation 1, with $\gamma = 0.9$. The agent was then trained based on the state-action-reward data obtained during all previous navigations.

Implementation

The maze was implemented as an object which stored the location of the agent, as well as the location of the walls of the maze.

The agents used the same code and parameters as for the grid task, with the navigation step modified to account for the existence of walls.

As with the grid task, three different forms of reward modulation were trialled in the RM agent. These were step modulation (equation 24) with $p = 50$, linear modulation (equation 25) with $p = 25$, and hyperbolic modulation (26) with $p = 1$.

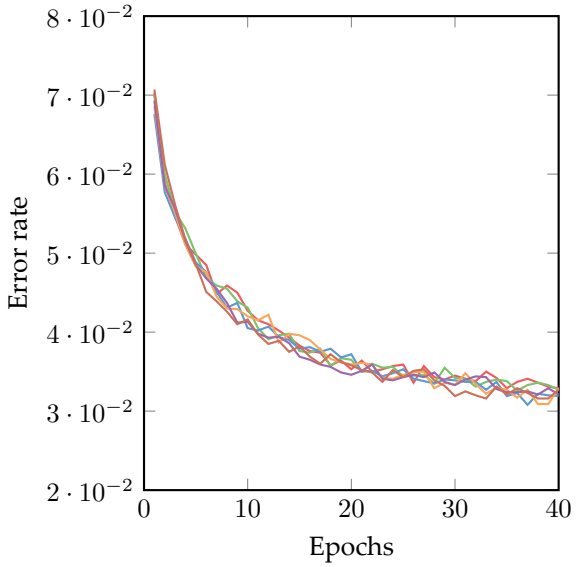
The agents were trained for 60 epochs on a 3×3 maze. These results are plotted in figure 9. The agents were also trained for 60 epochs on a 5×5 maze, but no agents reached the goal within the allowed number of moves.

not possible.

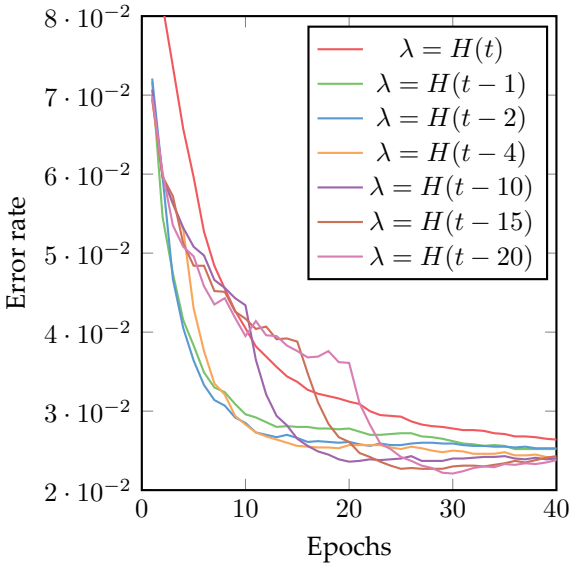
Results

Classification

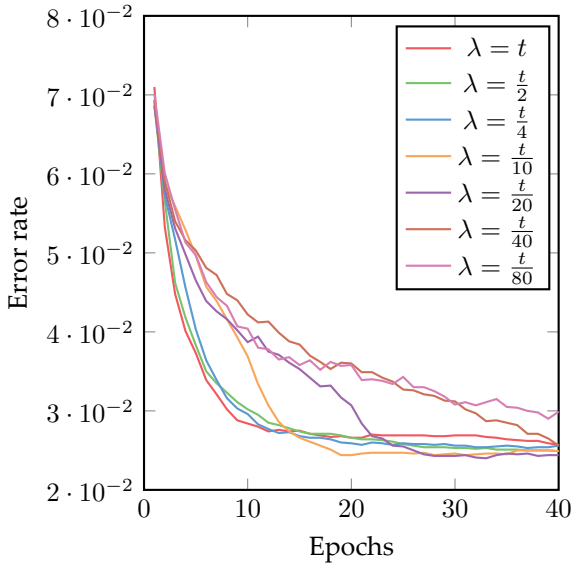
The following plots are of the performance of the trialled denoising autoenocders on the classification task over time.



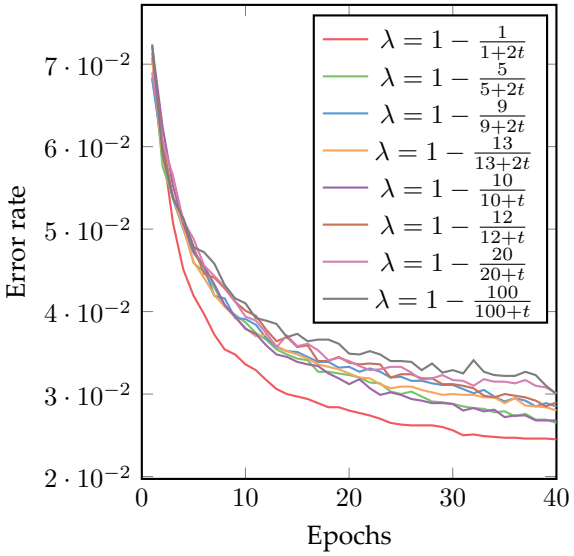
(a) Performance of non-RM denoising autoencoders on the classification task.



(b) Performance of step RM denoising autoencoders. $H(t)$ is the Heaviside step function with $H(0) = 0$.



(c) Performance of linear RM denoising autoencoders with different changes in λ.



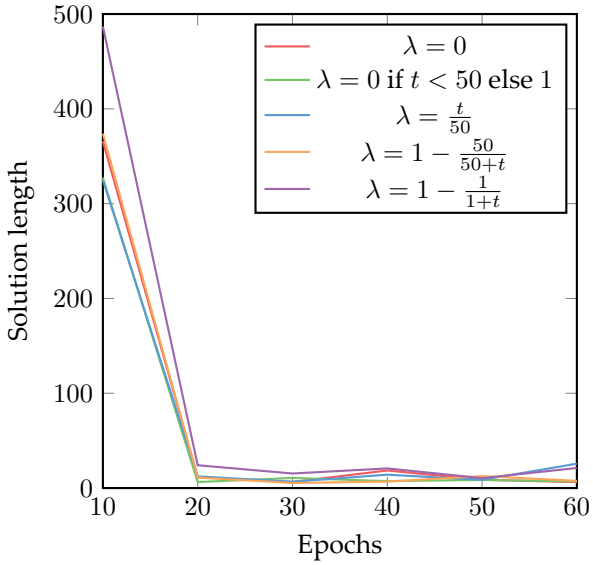
(d) Performance of hyperbolic RM denoising autoencoders with different changes in λ.

RM form	Best final error rate	$\lambda(t)$
None	3.19%	—
Step	2.38%	$H(t-20)$
Linear	2.44%	$\frac{t}{20}$
Hyperbolic	2.45%	$1 - \frac{1}{1+2t}$

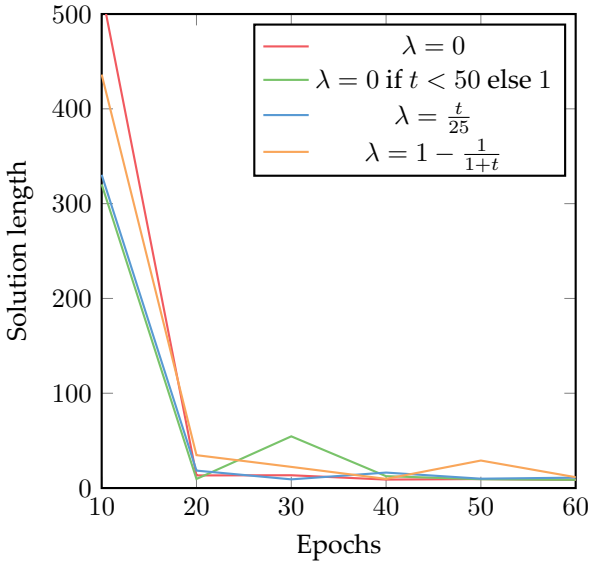
Table 1: Best final error rates of each method of reward modulation

Grid

The following plots are of the solution length of grid worlds. The trials were carried out five times for each form of reward modulation, and then averaged for each epoch. The error rates were then averaged for every ten epochs, and these averages are plotted.



(a) Solution lengths averaged every ten epochs for a 3×3 grid world with different reward modulation over time.



(b) Solution lengths averaged every ten epochs for a 5×5 grid world with different reward modulation over time.

Maze

The following plot is of the solution length of the maze world. The trials were carried out and averaged as for the grid world data.

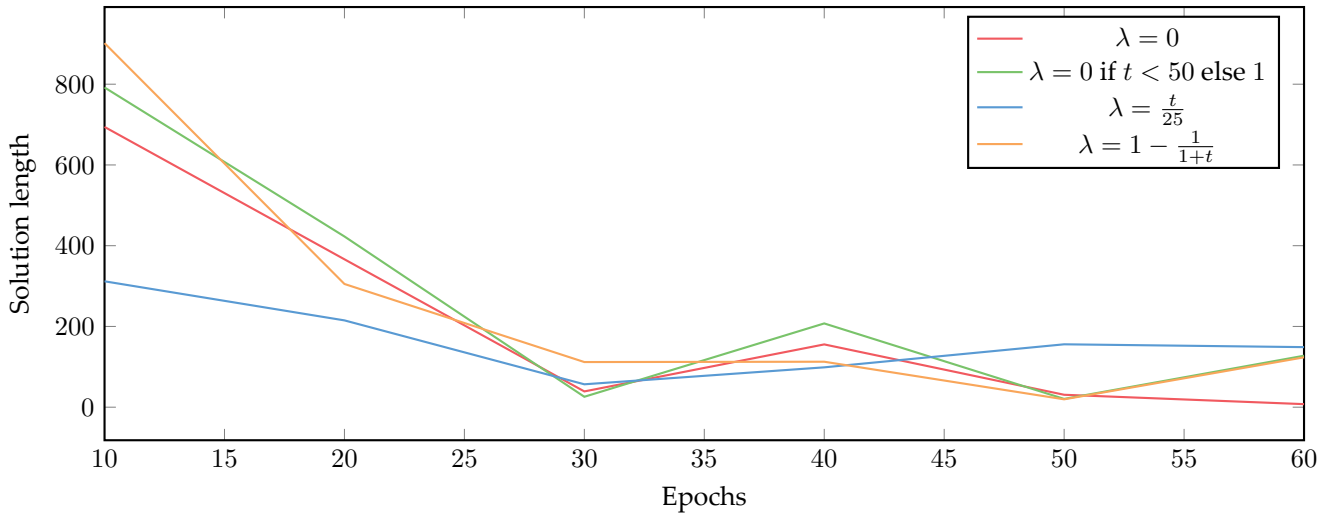


Figure 9: Solution lengths averaged every ten epochs for a 3×3 maze world with different reward modulation over time.

Discussion

Table 1 clearly shows that reward modulation has some noticable effect on error rate for the classification problem: With no reward modulation, the best final error rate was 3.19%, but with all forms of reward modulation, the best final error rate was under 2.45%. While 7a shows that due to the random initial configurations of the systems the error rates have some level of uncertainty, it is also clear from the same figure that this uncertainty is around $\pm 0.1\%$, which is not a large enough error to account for the improvement as a result of reward modulation.

However, no such effect is noticable on the MDP tasks. Figures 8a, 8b, and 9 show no significant difference in performance between the RM agents and the non-RM agent. It is possible that these tasks are too simple to gain improvement — larger grids and mazes, or more complicated MDPs like the grid task with perceptual aliasing, may show differences between RM and non-RM learning. However, we lack the computational power to test these larger problems.

Additionally, there are many free parameters in this experiment, such as the learning rate $r(t)$, the exploration $\epsilon(t)$, the reward decay γ, the batch size, the hidden layer dimension, the supervised and unsupervised cost functions, as well as the reward modulation $\lambda(t)$ itself. Only a limited number of tests could be carried out in this experiment due to the lack of adequate computational power — each trial took several hours to run, and only a few trials could be running at any one time. As such, compromise was necessary in selection of these parameters, such as the choice of 700 as the hidden layer dimension to balance speed and quality of the learned model. This is not ideal, as it is entirely possible that RM may be more or less successful with different choices of these parameters.

The results for classification raise further questions for investigation. The hyperbolic RM (figure 7d) seems to get better with lower values of p in the modulation $\lambda(t) = 1 - \frac{p}{p+t}$. For sufficiently small p, this is essentially one epoch of entirely unsupervised training, followed by almost entirely supervised training. That low values of p result in better solutions faster matches the results for step and linear RM (figures 7b and 7c respectively). When step modulation was used with $\lambda(t) = \begin{cases} 0 & t < 1 \\ 1 & t \geq 1 \end{cases}$ (such that unsupervised training took place for exactly one epoch, followed by entirely supervised training), the error rate very rapidly approached the best final error rate. This was also observed with small values of p in linear modulation $\lambda(t) = \frac{t}{p}$. The same was not observed when there was no initial unsupervised training ($p = 0$ in figure 7b). This implies that the first unsupervised step is very important to getting a good solution. Further investigation may determine whether there is a benefit to starting reward modulation at a non-zero value, such that the first training step is partially unsupervised and partially supervised.

Of additional note is the speed at which an optimal solution was found for the grid and maze problems with MNIST images. The input vectors were 1568-dimensional for a very small problem, and yet the denoising autoencoder agents succeeded in finding the optimal solution within 20 – 30 epochs.

Conclusion

Time-dependent reward modulation seems to greatly improve error rates on the MNIST classification problem, but there is no such effect apparent for MDPs. However, the latter result may be simply due to the small problem size tested. Further trials are needed to determine which reward modulation function is best, and to determine whether there is an effect for larger, more complex MDPs.

References

Yoshua Bengio. Learning deep architectures for AI. *Foundations and Trends® in Machine Learning*, 2(1):1–127, 2009. ISSN 1935-8237. doi: 10.1561/2200000006. URL http://dx.doi.org/10.1561/2200000006.

James Bergstra, Olivier Breuleux, Frédéric Bastien, Pascal Lamblin, Razvan Pascanu, Guillaume Desjardins, Joseph Turian, David Warde-Farley, and Yoshua Bengio. Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.

Deep Learning. Stacked denoising autoencoders [sda]. http://deeplearning.net/tutorial/SdA.html, 2014. URL http://deeplearning.net/tutorial/SdA.html.

Hugo Larochelle, Michael Mandel, Razvan Pascanu, and Yoshua Bengio. Learning algorithms for the classification restricted boltzmann machine. *The Journal of Machine Learning Research*, 13(1):643–669, March 2012. ISSN 1532-4435. URL http://dl.acm.org/citation.cfm?id=2503308.2188407.

Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. MNIST handwritten digit database. http://yann.lecun.com/exdb/mnist/, 2010. URL http://yann.lecun.com/exdb/mnist/.

S. van der Walt, S.C. Colbert, and G. Varoquaux. The NumPy array: A structure for efficient numerical computation. *Computing in Science Engineering*, 13(2):22–30, March 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2011.37.

Pascal Vincent, Hugo Larochelle, Yoshua Bengio, and Pierre-Antoine Manzagol. Extracting and composing robust features with denoising autoencoders. In *Proceedings of the 25th International Conference on Machine Learning*, ICML '08, pages 1096–1103, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-205-4. doi: 10.1145/1390156.1390294. URL http://doi.acm.org/10.1145/1390156.1390294.