Charles London

A tool for prediction of phenotype from cell genotype

Alternate: Semi-supervised learning with autoencoders for classification of gene expression data

Computer Science Tripos – Part II

Trinity College

May 8, 2019

Proforma

Name:	**Charles London**
College:	**Trinity College**
Project Title:	**A tool for phenotype prediction from cell genotype**
Examination:	**Computer Science Tripos – Part II, March 2019**
Word Count:	**1587**[1]
Project Originator:	Prof P. Lió
Supervisors:	Prof P. Lió & Helena Andres Terre

Original Aims of the Project

Work Completed

All that has been completed appears in this dissertation.

Special Difficulties

Learning how to incorporate encapulated postscript into a LaTeX document on both Ubuntu Linux and OS X.

[1] This word count was computed by `detex diss.tex | tr -cd '0-9A-Za-z \n' | wc -w`

Declaration

I, [Name] of [College], being a candidate for Part II of the Computer Science Tripos [or the Diploma in Computer Science], hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed [signature]

Date [date]

Contents

List of Figures

Acknowledgements

Chapter 1

Introduction

The stated aim of this project was to develop a semi-supervised autoencoder-based method for classifying cells into phenotypes [1] using genetic data. To this end a range of autoencoder based semi-supervised models have been implemented, ranging from fairly simple to state-of-the-art. These models were then evaluated on selected gene expression datasets, including the Cancer Genome Atlas expression data.

1.1 Motivation

Gene expression is the process of synthesising proteins from the gene via RNA. A transcriptome is the set of all RNA molecules in a cell or population of cells. While every cell with a nucleus in an organism has the same DNA and genes, the cells differ in function, and this depends on which genes are being actively expressed. This in turn means the transcriptome contains information from both genetic and epigenetic sources [5]. Epigenetic differences are differences in the phenotype without alterations to the DNA (e.g. DNA methylation). Therefore, as this project aims to classify cells into phenotypes, gene expression data is used.

Biological labs worldwide perform transcriptome analysis, resulting in huge amounts of gene expression data being generated. Much of this is shared or available online, giving researchers access to huge amounts of data. The development of RNA-Seq using next generation sequencing has also resulted in increased amounts of gene expression data, being more accurate and cost effective than previous methods. However, while many of the datasets generated in different experiments may include transcriptomes for the same species of organism, the majority of the time the experiments are measuring different phenotypes of the organism. This means that a researcher wanting to analyse or predict a specific phenotype is unable to use much of the available data, being limited to only those labelled with the desired phenotype.

[1] the observable characteristics and traits of an organism

Semi-supervised learning attempts to leverage unlabelled data to improve the accuracy of the machine learning algorithm on a supervised task. Autoencoders have a long history of being used in semi-supervised learning problems, having been used early on to improve deep networks by pretraining them using stacked denoising autoencoders (Section 2.4.2) and recently having been used to achieve state of the art semi-supervised performance as part of the ladder network (Section 2.4.4). The main reason for their use is that autoencoders are good at learning important features of data in an unsupervised manner, and these features are often useful in improving supervised performance.

Therefore, with the use of semi-supervised autoencoder models it should be possible to leverage the data without the desired phenotype to improve performance in predicting the phenotype.

The reason for choosing autoencoder-based models to use with gene expression data is that they are implemented using neural networks. This is advantageous because neural networks work very well on non-linear data and are also able to effectively analyse data with very high dimensionality (number of features). Transcriptomes can often contain several thousand genes, and so any model used must be able to cope with this level of dimensionality.

1.2 Related work

Stacked denoising autoencoders have previously been used with gene expression data to derive the most informative genes for distinguishing between healthy and cancerous cells [25].

Likewise, variational autoencoders (Section 2.3.3) have been successfully used to extract a biologically relevant latent space from cancer transcriptomes [28]. They and the semi-supervised variant (Section 2.4.3) have also been used to model the change in the gene expression of tumours in response to certain drugs [8]. he semi-supervised model in the paper, Dr.VAE, jointly models both the drug response and the treatment outcomes.

Ladder networks have also been used in biologically relevant ways, achieving state of the art accuracy in the binary cancer classification problem using gene expression data [7].

Chapter 2

Preparation

2.1 Neural networks

This project involves using **autoencoders** as a basis for semi-supervised learning and classification of gene expression data. Autoencoder architectures are all based on **deep feedforward networks** so it is important to have a good understanding of the basic principles of these networks.

2.1.1 The perceptron

The perceptron (also known as the **artificial neuron** [21]) is the basic element of the neural network. It takes in inputs $(x_1, ..., x_n)$ and multiplies these by **weights** $(w_1, ..., w_n)$, giving a linear combination of the inputs $x_1w_1 + ... + x_nw_n$, before applying an **activation function** σ [10].

It can be helpful to think of the inputs and weights as vectors $\boldsymbol{x}$ and $\boldsymbol{w}$, giving the combination as $\boldsymbol{x} \cdot \boldsymbol{w}$. A **bias** term $x_0 = 1$ may also be included and multiplied by weight w_0, giving $\boldsymbol{x} = [1, x_1, ..., x_n]$ and $\boldsymbol{w} = [w_0, w_1, ..., w_n]$. Intuitively the bias can be thought of as the intercept on a graph - if the neuron should have an output when all the inputs are zero it will be unable to model this correctly without a bias.

The function computed by the neuron is then:

$$y = \sigma(\boldsymbol{x} \cdot \boldsymbol{w}) \tag{2.1}$$

The activation function applied will depend on what the neuron is being used for. The main activation functions used in this project are:

- **Linear**

$$\sigma(z) = z \tag{2.2}$$

Used in regression problems, when a real-valued output that can take any value is required.

- **ReLU**

$$\sigma(z) = max(0, z) \tag{2.3}$$

Used in the hidden layers of deep feedforward networks (described in the next section) to provide non-linearity, allowing the network to learn more complicated functions. Has seen massive uptake due to performance benefits over sigmoid [19].

2.1.2 Multilayer perceptrons

Multilayer perceptrons are made up of multiple perceptrons arranged into **layers**. Each neuron in a layer has its own set of weights $\boldsymbol{w}$ and takes the ouputs of the previous layer (or the inputs to the network if the neuron is in the first layer) $\boldsymbol{x}$ in order to compute the output value y for the neuron. This value is then used as an input to the next layer, or part of the output of the network if the neuron is in the output layer.

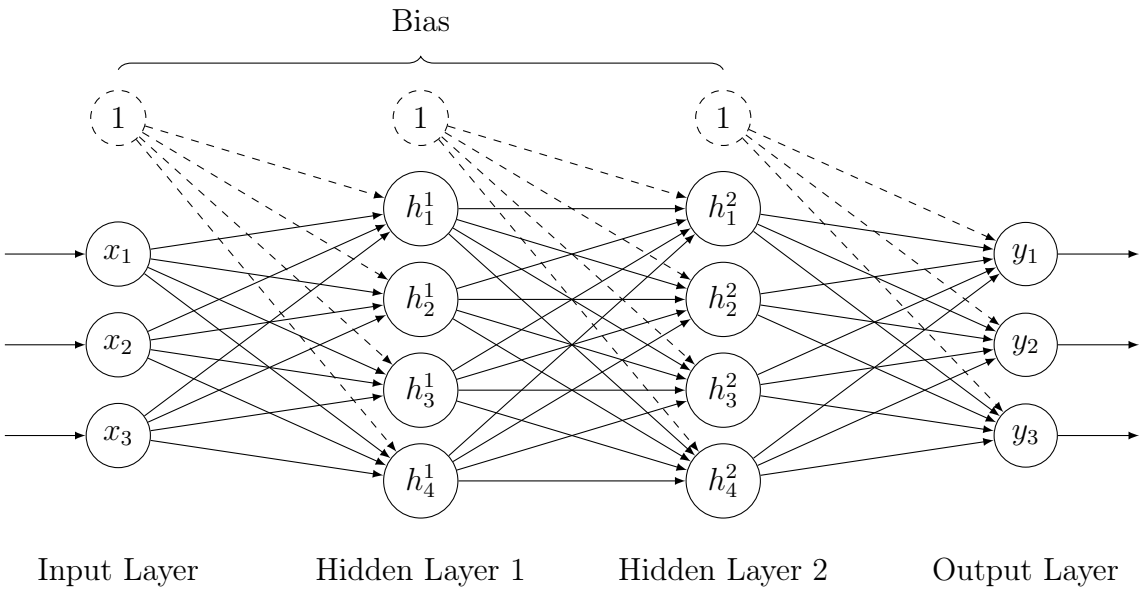


Figure 2.1: Illustration of a 4-layer multilayer perceptron

The main reason for neural networks with multiple layers is that they can model much more **complex non-linear relationships** [9]. than single neurons and single layer models are able to. Using non-linear activation functions such as ReLU (2.3) allow neurons to model simple non-linear functions, and combining these into layers allows them to model these more complex non-linear relationships.

I will denote the function computed by a neural network as $f_{\boldsymbol{\theta}}(\boldsymbol{x})$, where $\boldsymbol{\theta}$ denotes the trainable parameters of the neural network.

2.1.3 Neural networks for classification

A dataset for classification training is given as pairs of numerical **features** and a **label**: $(\boldsymbol{x}, y)$. These features are the input to the network, and the label is the target. A label cannot be directly utilised by the a neural network as they operate on numerical data. Therefore it is necessary to transform the data into a numerical representation.

One-hot encoding assigns an integer i to each class and makes a vector of length n (where n is the number of classes), setting the ith element to one and all other elements to zero. For example, with three classes, red, green, and blue the one hot labels would be $[1, 0, 0]$, $[0, 1, 0]$ and $[0, 0, 1]$. The network then has an output node corresponding to each class. This provides no implicit ordering over the classes (a problem found in integer coding) as each output node is equally important to the network [2].

The **loss** of the network is the difference between the output of the network and the correct label. Concrete details are given in the implementation, but loss should be high when the network is performing poorly, and low when it is performing well.

2.1.4 Training the network

Neural networks usually have their weights initialised to small random values. This means that on the first pass the output of the network will likely be very far from the correct value. Training a neural network involves adjusting its trainable parameters ($\boldsymbol{\theta}$) until the network reaches an acceptable loss or accuracy. This is done by computing the gradients of the trainable parameters with respect to the loss, and updating the parameters to move the loss towards a minima.

Updating the weights

Forward propagation is the flow of information through the network from input features to the output values. From these output values the loss is calculated. The trainable parameters of a neural network can then be updated by first finding the gradient of the loss with respect to these trainable parameters. The process of computing these gradients is called **backpropagation** and a full derivation can be found in Appendix A. The important notation is:

- θ is a vector of all the trainable parameters in the network
- $J(\boldsymbol{\theta})_k$ is the loss for the kth sample in the training set
- $\nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})_k$ is a vector of gradients of the loss with respect to all the trainable parameters

Once the gradients for all the weights have been calculated the weights are updated. By moving the weights a small step in the direction of steepest negative gradient the loss should decrease as the parameters are shifting it towards a minima. This means that the

model is getting better at modelling the training set. Taking these small steps is called **gradient descent**. The weight update rule is:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \sum_k \frac{\partial J(\boldsymbol{\theta})_k}{\partial \boldsymbol{\theta}} \tag{2.4}$$

$$:= \boldsymbol{\theta} - \eta \sum_k \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})_k \tag{2.5}$$

$$:= \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \sum_k J(\boldsymbol{\theta})_k \tag{2.6}$$

η is a **hyperparameter**. Hyperparameters are parameters of the model that are not trained but are set by the user before training. η is called the **learning rate** and controls how large of a step is taken each time the weight is updated. If the learning rate is too high the weights can “jump" over minima, and even diverge out of a minima, but if the rate is too low it can take too long to converge, or get stuck in a less desirable local minima. The step size is also proportional to the magnitude of the gradient, allowing the optimizer to take a larger step towards a minima if the gradient is steeper.

Batch learning

??

A neural network is generally run over multiple samples for each iteration, but not all the samples in the dataset. This is called mini-batch gradient descent and it is the most popular because it has stability advantages over stochastic gradient descent (one sample at a time) and computational advantages over batch (the entire dataset every iteration). Stochastic gradient descent has more noise as each update is based on an individual example, causing the weights to jump around more. It also doesn't take advantage of the parallelisation possiblity of much hardware - it's much more efficient to send multiple samples through at once. Batch has the problem of storing the whole dataset in memory, and only making one update per pass through the dataset can mean the model takes longer to converge to the best parameters. Mini-batch manages to have less noise than stochastic due to averaging over multiple samples, while also taking advantage of parallelisation and not requiring huge amounts of memory.

2.2 The manifold hypothesis

The **dimensionality** of data is the number of features needed to specify the data. For example, a very popular machine learning dataset is the MNIST dataset, containing 28x28 grayscale images of handwritten digits. The dimensionality of each datapoint is 784, as that is the number of pixels specifying each image.

The manifold hypothesis suggest that high dimensional data can actually be viewed as lying on or near to a lower dimensional manifold embedded in this higher dimensional space.

The precise definition of an n-dimensional manifold is “a topological space that is locally Euclidean", i.e. there is a neighbourhood around each point on the manifold that can be describe as n-dimensional Euclidean space. However, it is much easier to think about with an intuitive example - imagine that all the datapoints in a dataset lie on piece of paper, and so can be described with two features, an x and y axis. If the paper is taken and scrunched up and twisted it now has a three dimensional shape, but the data still lies on a two-dimensional manifold and, by unscrunching the paper, can still be described with only two features.

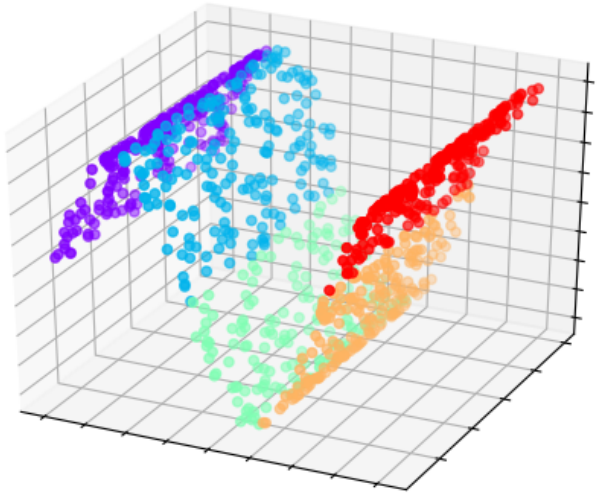
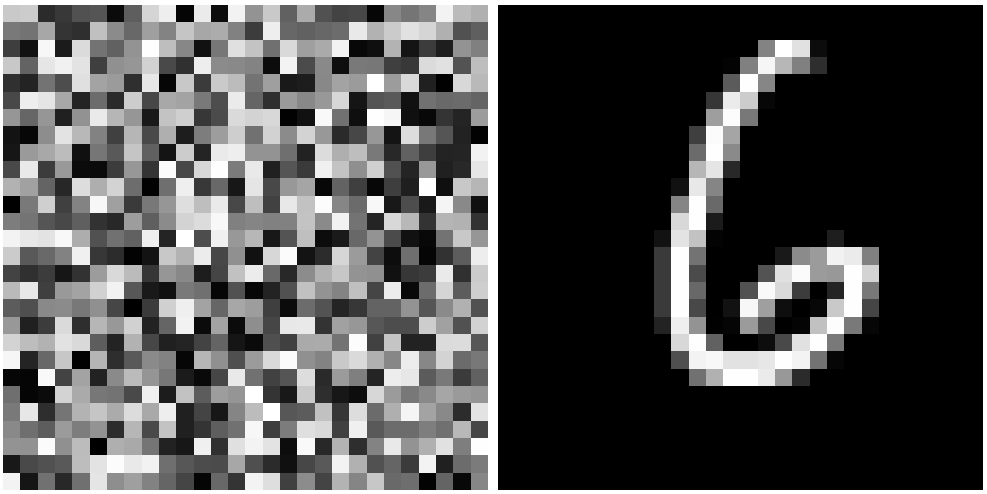


Figure 2.2: Points on a 2-dimensional manifold embedded in 3-dimensional space

Therefore, what the manifold hypothesis is suggesting is that many features in high dimensional data are actually redundant, and the data can be described using fewer features. Again this can be seen intuitively by the fact that the set of 784 pixel images that look like a recognisable digit is a very small subset of the set of 784 pixel images - if the pixel values were selected randomly from between 0 and 255 the picture would look like random noise, leading to the conclusion that there is a lot of redundancy in the MNIST features.



(a) Pixel values selected randomly (b) MNIST digit

Figure 2.3: Demonstrating redundancy of features in MNIST dataset

The manifold hypothesis then leads to **non-linear dimensionality reduction** techniques. By finding this lower dimensional mainfold the data can be explained with fewer features and possibly be more easily separated and classified. Going back to the paper example, linear dimensionality reduction techniques (e.g. principal component analysis) would be able to find the 2D embedding if the paper were rotated, translated or stretched, but would be unable to unscrunch the paper, as that amounts to a non-liner embedding. Autoencoders are used for non-linear dimensionality reduction, and are described in the next section.

2.3 Autoencoders

Autoencoders use neural networks in an unsupervised way to try and learn new **latent** representations of the data, typically with reduced dimensionality (i.e. there are fewer features in the latent data than in the input data). The use of neural networks allow it to learn a non-linear mapping from the data to the latent representation, the advantages of which were explained in the previous section.

Autoencoders are made up of two parts, the encoder and the decoder. The encoder takes in the original data, $\boldsymbol{x}$, and outputs a latent representation, $\boldsymbol{z} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$. The decoder then takes in this latent representation and attempts to reconstruct $\boldsymbol{x}$, outputting $\hat{\boldsymbol{x}} = h_{\boldsymbol{\phi}}(\boldsymbol{z})$. This allows an unsupervised problem to be turned into a supervised problem, using $\boldsymbol{x}$ as the target.

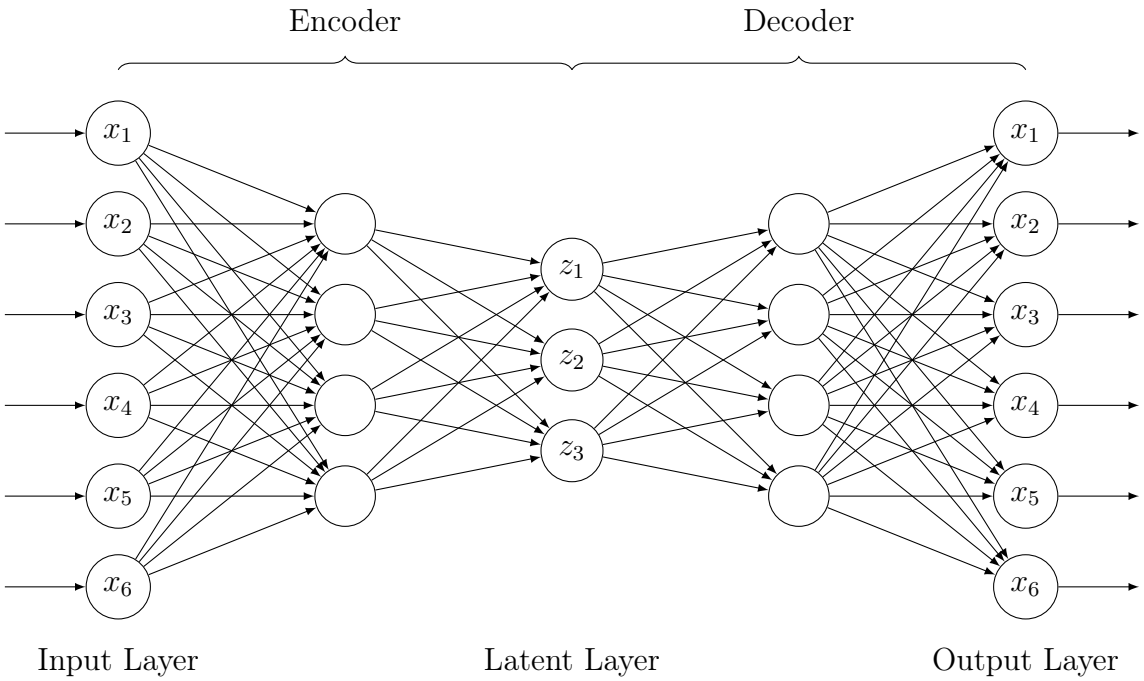


Figure 2.4: Illustration of a simple autoencoder

2.3.1 Simple autoencoders

Simple autoencoders, as in the figure above, constrain the network by making the number of latent features smaller than the number of input features. This prevents the network from simply learning the identity function, and hopefully results in an informative latent space.

Training

Autoencoders can be trained end to end using backpropagation as explained in section A. In order to do this they need a loss function, and the simplest and most widely used is MSE loss (**??**). The target x and output $\hat{x}$ are both vectors and so the Euclidean distance is used as the loss per datapoint. Both the encoder weights θ and decoder weights ϕ are trained at the same time; the gradients are backpropagated through the decoder to the latent layer and then back through the encoder every iteration.

2.3.2 Denoising autoencoders

Denoising autoencoders corrupt the input data with noise (e.g. by adding Gaussian noise or setting some of the features to 0) giving $\tilde{\boldsymbol{x}}$, which is then used as the input to the network. However the target is the uncorrupted input data, $\boldsymbol{x}$. The aim is to force the autoencoder to learn a better set of features because it not only has to reconstruct the

data but also has to remove the noise. They can be trained in the same way as a simple autoencoder.

Denoising autoencoders were used extensively to pre-train deep neural networks in the early 2010s, but this has become less popular with the introduction of new **non-saturating** activation functions such as ReLU which have helped deal with the vanishing gradient problem [Appendix ??].

2.3.3 Variational autoencoders

Variational autoencoders are based on Bayesian inference, and differ from other autoencoders in that the encoder outputs the parameters of a probability distribution over the latent variables, rather than a single configuration. Variational autoencoders were introduced by Kingma and Welling in 2013 and have become one of the most popular unsupervised learning techniques.

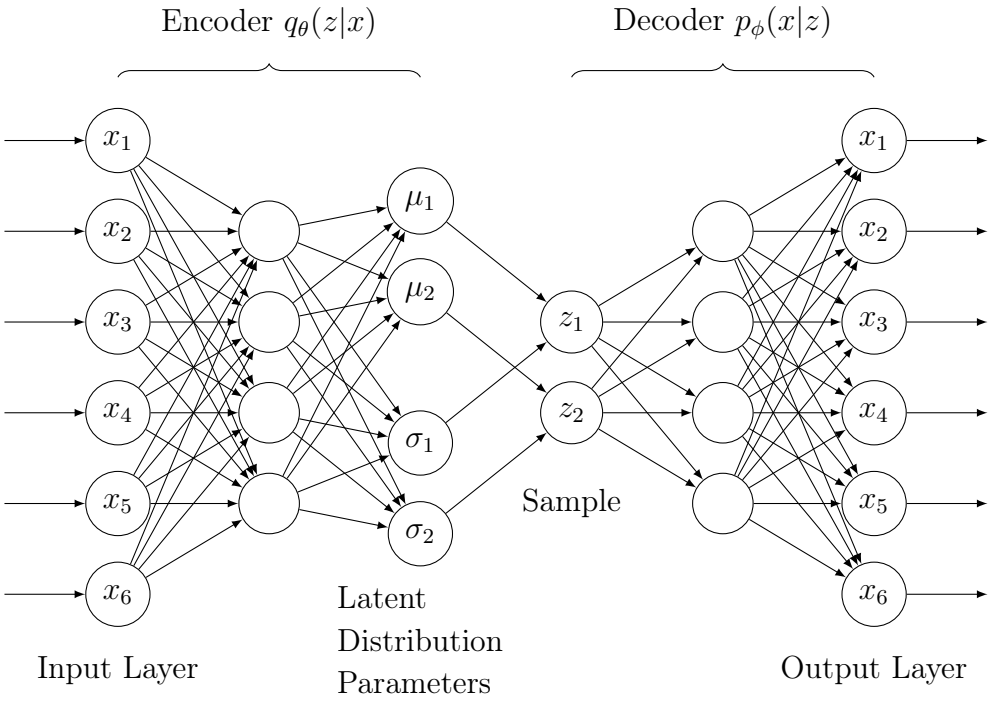


Figure 2.5: Illustration of a Gaussian variational autoencoder

The central idea of variational autoencoders is that the data x has been generated from some lower dimensional latent representation z. Each datapoint x_i is generated by:

- sampling z_i from the prior distribution over z: $z_i \sim p(z)$,
- sampling x_i from the conditional distribution $p(x|z)$ (known as the likelihood): $x_i \sim p(x|z = z_i)$

The prior $p(z)$ can be thought of as constraining the possible space of latent variables. For example, in the MNIST dataset a normal autoencoder could place 3s written in different

styles in different areas of n-dimensional Euclidean space (where n is the dimensionality of z) as the latent representation is a vector of unconstrained real numbers. This is detrimental to learning a meaningful latent space, and by constraining this space with a prior it should force the model to keep the representations of similar datapoints close.

The decoder of the VAE is a neural network that models the likelihood, $p_\phi(x|z)$. Once the VAE is trained new samples similar to those in the training set can be generated by sampling from the prior and passing this into the decoder; this is why VAE's are often referred to as generative models.

In most situations where unsupervised learning is useful only x is known. The goal is then to infer z. Inference in the model refers to finding good values of the latent variables given the data. This can be done by computing the posterior, $p(z|x)$. Using Bayes rule we have:

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} \tag{2.7}$$

The denominator $p(x)$ is known as the evidence and calculating it is intractable as it has to be computed by marginalizing out z:

$$p(x) = \int p(x|z)p(z)dz \tag{2.8}$$

Computing this integral requires exponential time as it has to be computed over all the possible configurations of the latent variables. Therefore the posterior is approximated with another simpler distribution $q(z|x)$ defined so that it is tractable. The most popular distribution, and the one used in this project, is the Gaussian. This can then be modelled by a neural network $q_\theta(z|x)$. This is the encoder.

Training

The loss function used for a variational autoencoder can be derived by maximizing the probability of the evidence. This makes sense intuitively, as a good model should maximise the probability of the real data [24].

$$\begin{aligned}
\log p(x) &= \log \int p(x|z)p(z)dz && \text{Law of total probability} \\
&= \log \int p(x|z)p(z)\frac{q(z|x)}{q(z|x)}dz \\
&= \log \left(\mathbb{E}_q \left[\frac{p(x|z)p(z)}{q(z|x)} \right] \right) \\
&\geq \mathbb{E}_q \left[\log \frac{p(x|z)p(z)}{q(z|x)} \right] && \text{Jensen's inequality} \\
&= \mathbb{E}_q \left[\log p(x|z) + \log \frac{p(z)}{q(z|x)} \right] \\
&= \mathbb{E}_q[\log p(x|z)] - D_{KL}(q(z|x)||p(z))
\end{aligned}$$

The final line of this equation is the **evidence lower bound** (ELBO). Maximizing the ELBO maximizes the probability of the evidence in the model, meaning the model fits the data as well as possible. The two terms in the ELBO correspond to the negative reconstruction loss and the **Kullback-Leibler divergence** between the computed posterior and the prior distribution of the latent variables. The KLD measures the difference between two probability distributions, and here measures the difference between the computed posterior and the real prior. This acts as a regularizing term, constraining the distribution of the latent variables to be close to the prior. Taking the negative of the ELBO gives the loss function for the variational autoencoder. Minimizing this loss function is equivalent to maximising the ELBO.

The Gaussian VAE is the most common and the one used in this project. The prior $p(z)$ is chosen to be the standard Gaussian, $\mathcal{N}(0, 1)$, for every latent variable. The output from the encoder is a vector of means $\boldsymbol{\mu}$ and standard deviations $\boldsymbol{\sigma}$ of normal distributions. During training, data is fed in mini-batches into the encoder and latent variables are then sampled from the encoder output: $\boldsymbol{z_i} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$. These are fed into the decoder which outputs a reconstruction $\hat{\boldsymbol{x}}$. The loss for each datapoint is then computed as the MSE loss (**??**) between $\hat{\boldsymbol{x}}$ and $\boldsymbol{x}$ and the KLD between $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\sigma}^2)$ and $\mathcal{N}(0, 1)$.

It is not possible to backpropagate the reconstruction loss through the drawing of the random sample and so **the reparameterization trick** is used. An explanation can be found in Appendix B.

2.4 Semi-supervised learning

Semi-supervised learning involves leveraging large amounts of unlabelled data to increase model performance on supervised learning tasks. In most fields unlabelled data is much easier to obtain than labelled data. As I mentioned in the Introduction, there is a wealth of gene expression data for many different organisms available online, with the only problem being that often the phenotype a researcher wants to study is not measured or included with this data. With standard supervised learning this data is not usuable, but semi-supervised learning can use it to improve performance.

2.4.1 Dimensionality reduction (M1)

The simplest semi-supervised model in this project relies on the manifold hypothesis. It uses a variational autoencoder to construct a reduced dimnesionality feature representation of the data that should cluster similar samples together. The idea is that it should then be easier to classify the datapoints, even with a limited amount of labelled data. The classifier used in this project is a neural network with softmax outputs. This differs from the implementation used by Kingma and Welling which uses a transductive support vector machine, and is done for simplicity of the project.

The model has to be trained in two stages, with the autoencoder trained on the combined labelled and unlabelled data, and the classifier trained on the latent representation of the labelled data. Once training is complete data can be classified by first passing it through the encoder and then through the classifier.

While a latent representation should hopefully lead to easier classification, at least some information is usually lost during dimensionality reduction and so often other methods are preferable.

2.4.2 Network pre-training

Stacked denoising autoencoders are a way of pre-training deep networks one layer at a time. Each hidden layer in the network is trained as part of a one layer denoising autoencoder, with the weights of the layer to be trained as the encoder and a new temporary layer as the decoder. The autoencoder takes the ouput from the previous layer in the network and uses this as the input, injecting noise before passing it through the autoencoder and attempting to reconstruct the clean input. The reconstruction loss is then backpropagated throught the autoencoder only (no other layers of the deep network) and the weights are updated. The loss computed by the autoencoder is referred to as an unsupervised "local denoising criterion" [27] as it does not require a label and is computed only for one layer at a time rather than the whole network. This allows the large amount of unlabelled data to be leveraged in training each layer.

The network is trained greedily, beginning with the first hidden layer. Once the reconstruction loss for the denoising autoencoder has converged for the layer the decoder is discarded, and the next layer is trained in the same way, using the output of the previously trained layer as input.

Once this unsupervised pre-training is finished the model is then *fine-tuned* by running normal supervised training, backpropagating classification loss through the entire network and updating the weights.

The idea behind the unsupervised pre-training with a stacked denoising autoencoder is that it provides a good prior to the supervised training. The pre-training procedure provides an initialization point for the supervised training where the parameters are restricted, hopefully to an area closer to the global minimum for the loss function [4].

2.4.3 The semi-supervised VAE

The semi-supervised VAE is a model due to Kingma et al. [15] that extends the VAE to include label information. The assumption used is that the data $\boldsymbol{x}$ is generated from both a discrete label $\boldsymbol{y}$ and a continuous latent representation $\boldsymbol{z}$, which are marginally independent of each other. Therefore $\boldsymbol{y}$ encodes the class of the data, while $\boldsymbol{z}$ encodes everything else. In the MNIST dataset this means that $\boldsymbol{y}$ encodes what digit the character is, while $\boldsymbol{z}$ encodes the style. The generative model (the decoder) works by sampling $\boldsymbol{y}$

from a categorical distribution $p(y)$ and by sampling $\boldsymbol{z}$ from a continuous distribution $p(z)$ (usually a Gaussian), before computing $p(x|y,z)$ using a neural network $p_\phi(x|y,z)$.

(a) The generative model (b) The inference model

Figure 2.6: The semi-supervised VAE
(shading indicates whether a variable is found in the dataset)

In the semi-supervised model there are two inference cases. When the data is labelled the question is of how to infer z from x and y. Despite x and y being marginally independent, they are not necessarily conditionally independent, and so both are used as input to the encoder. Another reason for this is that it helps the encoder to learn to separate the representation of y and z, so that z contains no information about the label. The semi-supervised VAE in this project uses a Gaussian distribution as the tractable distribution for z (like the unsupervised VAE), and the encoder models $q(z|x,y)$ with a network $q_\theta(z|x,y)$. For labelled data the model should maximise the evidence $p(x,y)$, the probability the model assigns to the real data. This leads to a variant of the ELBO by marginalizing out z:

$$
\begin{aligned}
\log p(x,y) &= \log \int p(x,y,z)dz \\
&= \log \int p(x|y,z)p(y)p(z)dz \qquad \text{Independence of } z \text{ and } y \\
&= \log \int p(x|y,z)p(y)p(z)\frac{q(z|x,y)}{q(z|x,y)}dz \\
&= \log \left(\mathbb{E}_q \left[\frac{p(x|y,z)p(y)p(z)}{q(z|x,y)} \right] \right) \\
&\geq \mathbb{E}_q \left[\log \frac{p(x|y,z)p(y)p(z)}{q(z|x,y)} \right] \\
&= \mathbb{E}_q \left[\log p(x|y,z) + \log p(y) + \log \frac{p(z)}{q(z|x,y)} \right] \\
&= \mathbb{E}_q[\log p(x|y,z) + \log p(y)] - D_{KL}(q(z|x,y)||p(z)) \\
&= -\mathcal{L}(x,y)
\end{aligned}
$$

This is very similar to the ELBO for the normal VAE, except that there is now a prior over y. This encodes previous knowledge about the distribution of the classes y, penalizing the model more when the label is of a low probability class. This is unimportant for the labelled data as the previous knowledge is drawn from this data, but becomes important for the unlabelled data, and the connection between the two can be seen in the derivation

below. A good model maximises the ELBO and therefore minimises $\mathcal{L}(x, y)$, which is used as the loss function.

When the data is unlabelled the problem is of inferring both y and z from x, $q(y, z|x)$. The evidence in the unlabelled case is $p(x)$, and the ELBO can be derived by marginalizing out both y and z:

$$\begin{aligned}
\log p(x) &= \log \sum_y \int p(x, y, z) dz \\
&= \log \sum_y \int p(x, y, z) \frac{q(z, y|x)}{q(z, y|x)} dz \\
&= \log \sum_y \int p(x, y, z) \frac{q(z|x, y)q(y|x)}{q(z|x, y)q(y|x)} dz \\
&\geq \sum_y q(y|x) \int q(z|x, y) \log \frac{p(x, y, z)}{q(z|x, y)q(y|x)} dz \qquad \text{Jensen's inequality} \\
&= \sum_y q(y|x) \int q(z|x, y) \log \frac{p(x, y, z)}{q(z|x, y)} dz - \sum_y q(y|x) \log q(y|x) \int q(z|x, y) dz \\
&= \sum_y q(y|x)(-\mathcal{L}(x, y)) + \mathcal{H}(q(y|x)) \\
&= -\mathcal{U}(x, y)
\end{aligned}$$

The marginalization of y is done by summation because the labels are discrete. Looking at the penultimate line of the derivation there is the term $q(y|x)$. This is classification, inferring y from x. This classifier is parameterised by a neural network, and outputs a categorical distribution over the labels using the softmax function (**??**). The summation term this is part of is referred to as "classification as inference" by Kingma [15]. For each label the labelled loss with respect to the data and that label is calculated and then multiplied by the probability of the label. This means that if a particular label leads to a bad reconstruction, implying that the label was incorrect, and the classifier assigns a high probability to that (likely incorrect) label, the loss to the classifier will be very high, and minimizing this loss should lead to better classification. The prior over y in $\mathcal{L}(x, y)$ becomes important here as it discourages the classifier from assigning a high probability to an unlikely class too often. All of this means that the classifier can learn directly from unlabelled data, with the small amount of labelled data providing a guide to good reconstructions.

The pipeline for labelled and unlabelled data is therefore slightly different at the moment. Labelled data is fed directly into the encoder along with its label, and the loss function $\mathcal{L}(x, y)$ is calculated and backpropagated through the network. Unlabelled data is first put through the classifier, before it is fed into the encoder once with each label allowing $\mathcal{U}(x)$ to be computed and backpropagated through the network.

This means that at no point does the classifier learn directly from the labelled data, unnecessarily disadvantaging the model. To remedy this the labelled loss function is

modified to include an extra term, the cross entropy loss (**??**) between the real label and the label the classifier outputs for the data. This modified version is then:

$$\mathcal{J}(x, y) = \mathcal{L}(x, y) - \alpha(y \cdot \log q(y|x)) \tag{2.9}$$

α is a hyperparameter that controls the weighting of the supervised loss, and is configured depending on the amount of labelled and unlabelled data available.

The model can now learn to classify from both labelled and unlabelled data at the same time, and so does not require pretraining the way the previous models did.

2.4.4 The ladder network

The ladder network is the most recent of the models included in this project having first been described by Valpola in 2014 [26], and then expanded upon by Rasmus et al. in 2015 [20]. It has some similarity to the SDAE, using an unsupervised local denoising criterion, but the structure of the model is very different. The model is made up of an encoder and decoder, like an auotencoder, but with lateral connections between the layers in the encoder and decoder.

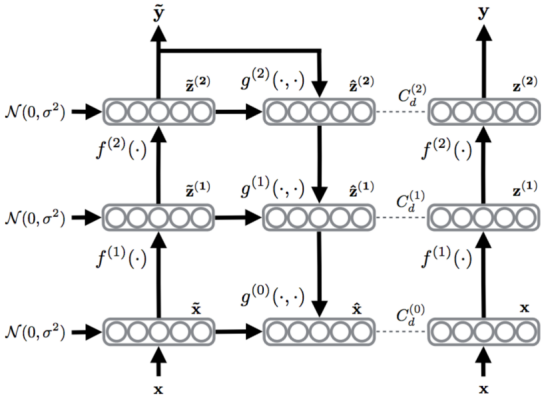


Figure 2.7: Illustration of the ladder network
(Source: Rasmus et al. [20])

The ladder network is a hierarchical latent variable model. These differ from the latent variable models looked at so far (autoencoder, VAE) in that they attempt to represent the data using multiple layers of latent variables, rather than a single layer of latent variables. In a normal autoencoder or VAE the single latent layer z has to encode everything about the data x, otherwise the reconstruction it generates will be very poor. With a hierarchical model each layer models only some information about x, with the higher layers able to be more abstract, as they don't have to model the lower level details encoded in the lower layers. For example, using MNIST, z in a simple autoencoder cannot just be the digit label as this does not provide enough information to reconstruct the original image well. In a hierarchical latent variable model the highest level latent variable is able to be very abstract and only encode the label, as the other layers in the model provide less abstract

information (e.g about style and location) that can lead to a good reconstruction. The reason this works in the ladder network is due to the lateral connections, which "leak" information from layers in the encoder to the decoder. This means that each decoder layer receives information both from the previous decoder layer and the corresponding encoder layer. The encoder layer passes some information which the subsequent encoder layers then no longer have to model, as the decoder will receive it directly from the encoder layer.

This structure with the representation becoming more abstract in higher level layers is also analagous to how supervised learning works, with the layers further into the network modelling more complicated and abstract features with the final layer just outputting a class label (most abstract).

At the input to each decoder layer $u^{(l)}$ the corresponding encoder representation $z^{(l)}$ and the output of the previous decoder layer $u^{(l+1)}$ are combined together using a combinator function g. g has trainable parameters, but this leads to a problem where the lowest possible unsupervised loss can be achieved by g in the first layer learning to copy $z^{(0)}$ and completely ignoring the $u^{(l+1)}$, which corresponds to copying the input directly to the output at the bottom layer. This completely short circuits the autoencoder, and in order to prevent this, Gaussian noise (sampled from $\mathcal{N}(0,1)$)) is added to the input to each layer in the encoder. Each encoder layer then has the representation $\tilde{z}^{(l)}$ and this noise means that just copying over the input no longer minimises the cost function.

However if the unsupervised cost function used is simply the reconstruction loss between the decoder output $\hat{x}$ and the encoder input x the first layer of the network still has a disproportionate influence on the loss. In order to remedy this Valpola proposes adding local denoising criterion. This involves adding a cost function at each layer of the decoder, namely the mean square error between $g(\tilde{z}^{(l)}, u^{(l+1)})$, which is $\hat{z}^{(l)}$, a denoised representation of $\tilde{z}^{(l)}$, and $z^{(l)}$, the clean representation from layer l of the encoder. In order to generate both the clean and noisy encoder representations the encoder is run twice per training iteration, once without the added noise to generate $z^{(l)}$, and once with noise to generate $\tilde{z}^{(l)}$. These local cost functions require all the layers to learn in order to make a meaningful representation that can be denoised well. Each layer loss is multiplied by a hyperparameter λ_l according to how important the denoising cost of the layer is, before being summed together to give a final unsupervised cost function per sample of:

$$\mathcal{U}(x) = \sum_l |z^{(l)} - \hat{z}^{(l)}|^2 \tag{2.10}$$

The model as explained so far allows the ladder network to learn abstract features in the higher layers. However, without supervised data and a supervised cost function the features learned are unlikely to be useful for the classification task. The small amount of labelled data is used as a guide for this. The encoder of the model is the original classifier so the data is passed through the encoder and cross entropy loss is computed between

the output and the labels. During training noise is added at the output of each layer, the same as one run of supervised learning, as the noise acts as a regularizer [29].

An overview of the training process can be found in Section 3.1.9

2.5 Requirements analysis

By taking the success criteria from the project proposal (Appendix C) I constructed a set of tasks that must be completed for the project and ranked them by their priority. The success criteria are:

- The implemented models achieve close to original paper performance on the MNIST dataset
- The final chosen model achieves better prediction accuracy than supervised learning alone on genetic datasets
- A tool is built that takes in file paths to unlabelled and labelled data and trains a classifier based on this

The first success criteria differs from the project proposal. After reading the papers of the models to be implemented (Section 2.6) I realised that the most important models, the semi-supervised variational autoencoder and the ladder network, both had benchmarks given on the MNIST database of handwritten digits [18]. As the stated aim of generating synthetic data was to ensure that the models were working correctly, I decided that a better measure of the correctness of the models was whether they achieved (close to) the accuracy reported in the papers.

The models compared in this project in order to decide on the best semi-supervised method to use were:

- **A variant of the Kingma M1 model** [15]. A variational autoencoder trained on combined labelled and unlabelled data followed by a neural network classifier trained on the latent representation of the labelled data.
- **A stacked denoising autoencoder** [27].
- **A semi-supervised variational autoencoder**. [15] Also referred to as the Kingma M2 model.
- **A ladder network** [20]

With these models selected and the success criteria defined above the requirements can be constructed:

Requirement	Priority
Implement simple multilayer perceptron	Medium
Implement Kingma M1 model	Medium
Implement stacked denoising autoencoder	Medium
Implement semi-supervised autoencoder	High
Semi-supervised autoencoder achieves close to original paper accuracy on MNIST	Medium
Implement ladder network	High
Ladder network achieves close to original paper accuracy on MNIST	Medium
Process the Cancer Genome Atlas gene expression data	High
Evaluate and compare performance of models on MNIST	Low
Evaluate and compare performance of models on TCGA data	High
Implement saliency for best performing model (extension)	Low

Table 2.1: Requirements for a successful project

2.6 Starting point and reading

I had some previous knowledge about neural networks and machine learning techniques through completing *Introduction to data science* and *Artificial intelligence I* as part of Part IB, and completing the machine learning course by Andrew Ng on Coursera. However, I had no previous experience with autoencoders, semi-supervised learning or Bayesian inference, and also little experience of optimising a machine learning model. To this end I decided upon a list of essential reading: [11] [9] [14] [15] [27] [26] [20].

2.7 Software engineering

All the models in this project were developed iteratively. They were first constructed in their most basic form, and tested on MNIST data to determine whether they were working correctly. Enhancements such as early stopping and other hyperparameter optimisations were added in stages, and the models were tested at each stage to ensure that the new code had not led to any problems.

2.8 Resources

2.8.1 Language and libraries

I made the decision to use Python, as it has support for the deep learning library PyTorch. After trying both PyTorch and Tensorflow, and looking at a comparison of the features

offered, I decided to use PyTorch, as it felt more flexible, allowing the user to easily stop and start training at any point, and allowing much easier access to intermediate variables for debugging. In Tensorflow all operations have to be run using a Session object, and the only variables a user can see are those returned from the session object.

I used the PyCharm IDE for writing my Python code as I was familiar with Jetbrains IDEs. PyCharm includes useful features like autocomplete, and has good integration with git.

2.8.2 Hardware

All of the programming was done on my laptop (Macbook Pro 2014, 2.6 GHz Intel Core i5, with 8GB of RAM, a 128GB SSD and 128GB SD card).

Running the models, especially on the highly dimensional gene expression data, is extremely computationally intensive, and runs much quicker on a GPU. To this end I was given access to NVIDIA Titan X and P100 GPUs by Prof. Lió.

2.8.3 Backing up

To back up my project and dissertation files and code I stored them in a git repository synced with a remote repository on GitHub. I also made regular backups of my entire SSD to an external hard drive.

2.9 Summary

This section should have provided an overview of the theory behind this project, and of the requirements that must be completed to ensure the project is a success.

Chapter 3

Implementation

This section of the dissertation is focused on the steps involved in constructing, training and evaluating the models used in this project. All the models were written from scratch in PyTorch, using the original papers and their implementations (in Theano) as starting points. This section also covers the tuning of the model hyperparameters, and the subsequent evaluation of the models.

3.1 Model and class structure

3.1.1 PyTorch data structures

The most important computational unit in PyTorch is the **tensor**. Tensors are similar to multidimensional arrays, but they are able to utilise GPUs to perform operations in parallel. This is important because the running of a neural network involves thousands of matrix operations, many of which can be performed in parallel, as they don't depend on each other. GPUs have high memory bandwidth and are optimised to perform simple operations in parallel, making them well suited for machine learning applications.

Tensors have the property `requires_grad` which tells PyTorch whether to record the operations that happen to the tensor. If this is true the tensor records all operations on it, and build up a **computation graph**. The computation graph is a directed acyclic graph containing input tensors as the roots and output tensors as leaves and has `Functions` as internal nodes. These functions are actually the expressions that are applied to the tensors during the forward pass. The construction of this graph allows for backpropagation to be computed easily by performing a backwards pass on this graph to compute the gradients. The graph is created during the forward pass, which is why they are referred to as **dynamic** computation graphs.

Parameters are tensors that are trainable parameters of the network, for example the weights and biases. The parameter class is a wrapper that tells a **module** whether to include a tensor in the parameters object of the module.

Modules are the base class for all models in PyTorch, and are combinations of parameters and functions that will be applied to input data when the `forward` method of the module is called. All models subclass `nn.Module` and these modules can be nested to allow more complicated models to be constructed from basic constituent parts. For example the `nn.Linear` module contains weight and bias parameters, and multiplies the input data by the weights and adds the bias in the forward method.

3.1.2 My base modules

My codebase contains four base modules that are used to construct the more complicated higher level models. The simplest of these base classes are Classifier, Encoder and Decoder, and the difference between them is mainly semantic and used to keep the code clearer.

The Classifier constructor takes as input the dimensionality of the input data, a list of hidden layer sizes, and the number of classes in the output, and constructs a simple multilayer perceptron module using `nn.Linear` for each layer. It uses ReLU as the activation function in all the hidden layers, because it is non-saturating, meaning that it does not suffer from the vanishing gradients problem, where the gradients of the early layers of the neural network become very small and so update very slowly. It uses a linear function on the output because the implementation of cross entropy loss in PyTorch computes log softmax inside it, and so computing softmax in the classifier is not necessary.

The Encoder is very similar to the classifier, except the constructor also takes in an activation function that is applied to the output of the network. This becomes important when the encoder is used in the SDAE. The Decoder constructor is the same as the Encoder to make constructing Encoder/Decoder pairs as simple as possible. It takes in the same inputs but reverses them to create the complementary decoder, also including an output function if required.

The Variational Encoder is a separate module from the Encoder, because the forward pass has three outputs. The output of the network is two vectors of the same dimension, μ and σ, the parameters of the posterior Gaussian distributions. It also contains a module that samples from these distributions to create the hidden vector z using the reparameterization trick (Appendix B). The output of calling `forward` on the Variational Encoder is then z, μ and σ.

Weight initialisation

Random initialisation is required in neural networks to give them the best chance to find minima of the loss, as the starting point can greatly effect the gradients in the network and the direction the parameters move in. If the weights were deterministically initialised the network would always find the same minima, which may not be optimal. Weight initialisation can help models converge more easily as reported by Glorot and Bengio [6].

The default weight initialisation in PyTorch is Xavier initialisation which involves drawing the weights and biases from a normal distribution with the parameters below:

$$\mu = 0 \qquad \sigma = \frac{1}{\sqrt{n_{l-1}}}$$

where n_{l-1} is the number of neurons in the previous layer

3.1.3 Class structure

The models used in this project all required certain methods to allow the scripts used for running the training and evaluation to be as model agnostic as possible. To this end I implemented a base class `Model` for all the models to ensure they all had the same methods. `Model` also subclassed `nn.Module` to ensure that the models adhered to PyTorch standards.

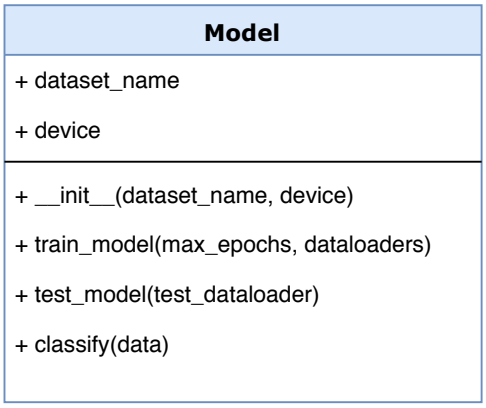


Figure 3.1: Model base class

The `dataset_name` field is used to save the model state in the correct place when using early stopping, and the `device` field is used to ensure that the GPU is used if available, and all new tensors created are placed on the GPU. Not placing the tensors on the correct device leads to errors as PyTorch cannot perform operations on tensors located on different devices.

Each model has different fields depending on the structure of the model, so initialisation of the models could not be a general method. However, for evaluation I implemented a `hyperparameter_loop` method per model which took the same arguments for all models:

```
hyperparameter_loop(fold, validation_fold, state_path, results_path,
dataset_name, dataloaders, input_size, num_classes, max_epochs, device)
```

This method loops through a pre-selected set of hyperparameters to determine the best performing. This was possible because the hyperparameters chosen for searching over were not dependent on the input dataset, and were instead selected to give good general performance e.g. the ladder network hyperparameter search constructs a ladder network

with 1-4 hidden layers, with the deeper model winning on more complicated datasets and the shallower on less complicated.

3.1.4 Model training

All the models were trained using mini-batch gradient descent, as described in Section **??**. One complete pass through the dataset is called an **epoch** and training involves running forward and backpropagation for a certain number of epochs to minimise the loss on the training set and cause the model to converge to a good approximation to the real function. The pseudocode for training the network is shown below.

Algorithm 3.1 Train neural network via mini-batch gradient descent

```
procedure TRAINING(i, 𝒟, η, model, loss_function)
    for i epochs do
        for mini-batch ℳ in 𝒟 do
            data, labels = ℳ
            out = model(data)
            𝒥 = loss_function(out, labels)
            model.θ = model.θ - η∇_θ 𝒥
```

3.1.5 Multilayer perceptron

The multilayer perceptron is a fully supervised model implemented to compare the performance between a fully supervised approach and the semi-supervised approaches used in the project. The model simply uses the Classifier base module and trains it on only the labelled data using the Adam optimizer for gradient descent.

3.1.6 M1

The M1 model is partially taken from the Kingma paper [15]. It is made up of a VAE Encoder, Decoder and Classifier module. The VAE Encoder and Decoder are initially trained in an unsupervised way using both the labelled and unlabelled data. In order to train them I made the decision to normalise all the data to the range [0-1] and to use a Sigmoid function [1] on the output of the Decoder, to ensure the output range was the same as the input. This was done for numerical stability reasons, as it was found that using unnormalised or standardised data as input to the VAE would cause the variance of the latent distribution to become infinitely small and eventually lead to the loss and output of the VAE becoming `nan`.

[1] the sigmoid function is of the form $f(x) = \frac{1}{1+e^{-x}}$ and has the domain of the real numbers and range [0-1]

Once the VAE Encoder and Decoder are trained the Decoder is discarded. The labelled data is passed through the VAE Encoder, and the samples from the distributions are used as input to the Classifier. The Classifier is then trained using this labelled data, using cross entropy loss as the loss function.

This model relies on the manifold hypothesis. The training of the VAE hopefully clusters similar samples together by finding a lower dimensional manifold, allowing the Classifier to more easily separate the classes using only a small amount of labelled data.

Once the Classifier is trained the full pipeline for classifying new data involves first passing the data through the VAE Encoder and then through the Classifier.

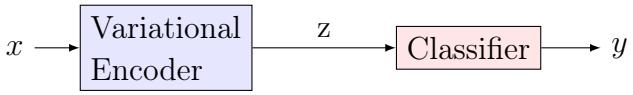


Figure 3.2: The classification pipeline of the trained M1 model

3.1.7 Stacked denoising autoencoder

The SDAE was constructed by first constructing a list of Encoders. Each Encoder had the same input size as the latent dimension of the previous Encoder. All the Encoders use ReLU as the output activation function, and the final output layer is a simple `nn.Linear` layer with no output activation. This means the structure of the network is very similar to the Classifier, but it enables easy unsupervised pretraining.

The unsupervised pretraining is performed by taking each encoder, starting with the first hidden layer, and adding the corresponding Decoder. This is where the pairing of the Encoder and Decoder is very useful, as by extracting the layer sizes of the Encoder they can be passed straight into the Decoder contructor to create the complementary Decoder. Then Gaussian noise is added to the input data, using the PyTorch function `torch.randn_like(data)`, which creates a Gaussian noise tensor of the same size as the data that can simply be added to the data. This Encoder/Decoder pair is then trained like a normal autoencoder using mean square error loss, and this is the local denoising criterion.

Once each hidden layer has been trained like this, the whole network is trained using labelled data to ensure that the hidden layers are detecting the right features for the task. Classification just involves passing the data through all the Encoders and then through the output layer.

3.1.8 M2

The M2 model is made up of a Classifier, a Variational Encoder and a Decoder. It is again taken from the Kingma paper [15]. Data is first passed through the Classifier to get a prediction for the label. The loss functions used were given in Section 2.4.3, and

depend on whether the data is labelled or unlabelled. If the data is labelled then the data is passed into the Variational Encoder with the correct label, and the output of the Variational Encoder is passed into the Decoder which attempts to reconstruct the data. If the data is unlabelled then the data and each possible label are passed into the Variational Encoder, and the reconstruction loss is computed for all the combinations. This seems like it would greatly increase the training time, but this can actually be done by creating a larger tensor with every unlabelled sample in the batch paired with each label. This operations on this larger tensor can be computed in parallel on the GPU so the performance hit is not as drastic as this operation might suggest. The overall structure of the model is shown below:

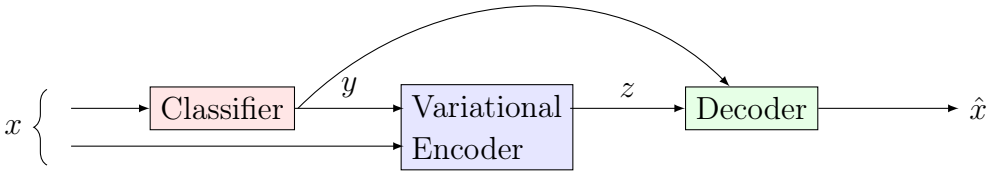


Figure 3.3: Model structure of M2

The main diffculties in implementing this model came from conflicting information found online, and the lack of other implementations to view. The original paper can be quite hard to decipher, often skipping over difficult derivations, and at one time seemingly contradicting itself in the space of two lines. One of the original blog posts I viewed [12] did not pass the label into the encoder, and after corresponding with the author I found it was due to a different interpretation of a line in the paper. However, Kingma's implementation (available at: https://github.com/dpkingma/nips14-ssl) does do this and so that is the model I followed. A helpful reference implementation I found is available at https://github.com/wohlert/semi-supervised-pytorch, but I believe my implementation is easier to read and also performs better, as I use the inbuilt PyTorch models for computing the cross entropy loss and use the Kingma way of computing the KLD [14]. I was able to contribute to the Wohlert repository too, finding a bug in the variational autoencoder code where the natural logarithm of the variance was restricted to values greater than zero (this is incorrect).

This model again requires that the data be normalised to the range [0-1] (as in M1) as otherwise the loss explodes and becomes `nan` as the variance of the latent distribution becomes very small.

Once the model is trained the only part used in classification is the Classifier.

3.1.9 Ladder

The ladder network doesn't make use of any of the base modules, as it requires a lot of extra parts. My original implementation was based on an implementation found at https://github.com/abhiskk/ladder. However, this implementation is very slow, as at one

point tensors on the GPU are shifted onto the CPU to perform operations using numpy, which I found was unnecessary. The implementation also did not achieve the desired accuracy on MNIST. For this reason I based my implementation on a Tensorflow implementation written by Rinu Boney (available at https://github.com/rinuboney/ladder), who works at Curious AI, the same company as Rasmus, the author of the 2015 paper [20]. The code required many modifications to move from Tensorflow to PyTorch, but my implementation is now the fastest and most accurate PyTorch implementation of the model I could find on the internet.

While the model is described in some detail in Section 2.4.4 the training process is as below:

1. Data is passed into the clean encoder and the output from each layer is saved.
2. Data is passed through the noisy encoder, and output from each layer is saved.
3. The final output from the noisy encoder is the classification output, and supervised loss is computed with it.
4. The output from the noisy encoder is fed into the decoder. At each layer l of the decoder $g(\tilde{z}^{(l)}, u^{(l+1)})$ is calculated, and the unsupervised loss for the layer is computed.
5. The unsupervised losses are multiplied by hyperparameter λ_l and summed together.
6. The supervised and unsupervised losses are summed and backpropagated through the network and the weights are updated.

The Python code used to implement this process is:

```
for batch_idx, (labelled_data, unlabelled_data) in enumerate(
    zip(cycle(supervised_dataloader), unsupervised_dataloader)):
self.ladder.train()

self.optimizer.zero_grad()

labelled_images, labels = labelled_data
labelled_images = labelled_images.to(self.device)
labels = labels.to(self.device)

unlabelled_images, _ = unlabelled_data
unlabelled_images = unlabelled_images.to(self.device)

inputs = torch.cat((labelled_images, unlabelled_images), 0)

batch_size = labelled_images.size(0)

# pass data through encoders clean
y, clean = self.ladder.forward_encoders(inputs, 0.0, True, batch_size)
# pass data through encoders noisy
y_c, corr = self.ladder.forward_encoders(inputs, self.noise_std, True, batch_size)

# compute supervised cross entropy cost
supervised_cost = self.supervised_cost_function.forward(labeled(y_c, batch_size),
                                                        labels)

# pass output of encoder through decoder
z_est_bn = self.ladder.forward_decoders(F.softmax(y_c), corr, clean, batch_size)
```

```
zs = clean['unlabeled']['z']

# compute unsupervised reconstruction cost for each layer
unsupervised_cost = 0
for l in range(self.L, -1, -1):
    unsupervised_cost += self.unsupervised_cost_function.forward(
        z_est_bn[l], zs[l]) * self.denoising_cost[l]

loss = supervised_cost + unsupervised_cost

# backpropagate loss through network and update weights
loss.backward()
self.optimizer.step()
```

For the most part this was implemented without difficulty, though I did discover early on that the loss sometimes became `nan`. By adding a backward hook to the model that would notify me when the values became `nan` I discovered that this was due to the fact that the ladder requires the data be standardised, which involves dividing the features in the data by the standard deviation of the features. However, in the MNIST dataset the pixels on the edge of the images are always black, and so have a constant value of zero over all the images. The standard deviation of these features is therefore also zero, and dividing by this leads to data becoming `nan`.

3.2 Hyperparameter optimisation

While the weights and biases in a network can be trained by backpropagation, there are parameters of the model that cannot be trained in this way, but still affect the performance of the network. These are hyperparameters and they are set before the training begins. Optimising these can be crucial in getting the best performance from a model.

3.2.1 Grid search

The most common method for hyperparameter optimisation, and the one used in this project, is grid search. The researcher provides several likely values for the hyperparameters, and the model is trained with each combination of these. A validation set is used to compare the performance of the models, and the best set of hyperparameters is selected this way.

However, optimising hyperparameters this way results in every combination having to be run again for each additional value of each hyperparameter used. Depending on how long it takes each model to train, this can result in hours more training time. Therefore, I chose to optimise over a small number of hyperparameters that I feel are most likely to affect the performance of the models.

3.2.2 Number of hidden layers

The number of layers in a model is important because it controls the capacity [9] of the neural network. If the capacity of the network is too high it can **overfit** to the training data, resulting in poor generalization to new examples, and if the capacity is too low it can **underfit**, resulting in poor fitting to the train data. Therefore choosing the number of layers is an important hyperparameter. The universal approximation theorem states that a neural network with a single hidden layer can approximate any function, but the layer has to be exponentially large to give the same capacity as a multi-layer network; instead adding new layers is the preferred method, with the deeper layers learning more complex and relevant features to the task.

3.2.3 Hidden layer size

Larochelle et al. [17] found that neural networks with constant layer size usually perform better than those where the layer size increases or decreases throughout the network. When optimising the number of layers I use a constant layer size for each hidden layer.

Bengio et al. [1] found that using a first hidden layer larger than the input layer often resulted in better performance. However due to the high dimensionality of gene expression data (over 20,000 genes in the TCGA data) I decided against this. I use a constant layer size for all the hidden layers, using the same layer size for all the different models to allow for a fairer comparison.

3.2.4 Latent dimension of autoencoders

The size of the latent dimension for an autoencoder affects the quality of the representation learned. If the dimension is too small the autoencoder may not be able to encode all the important features, whereas a latent dimension that is too large can give the autoencoder too much freedom, causing it to not cluster important samples together.

This hyperparameter is only optimisable for the M1 and M2 models, as the hidden dimensions for the autoencoder in the SDAE and ladder are controlled by the layer size and number of classes.

3.2.5 Learning rate

The learning rate is a hyperparameter that controls how large of a step is taken each time the weights are updated. It is an often optimised hyperparameter because if it is too large it can overshoot loss minima, resulting in worse performance, and if it is too small it can take a very long time for the gradient descent to converge. However, I have chosen not to search over different learning rates because of the additional time it would take, and the fact that I am using the Adam optimizer. The Adam optimizer keeps a learning rate per

parameter, and updates these learning rates according to the first and second moments of the gradients with respect to each parameter [13]. This means that while an initial learning rate still has to be provided, it affects the performance much less.

3.2.6 Early stopping

The number of epochs that a learning algorithm is trained for also affect how well it performs. Not training for long enough can result in poor peformance and underfitting, as the model has not had enough time to learn the relevant features, while training for too long can be wasted compute time if the model is not improving, or even lead to overfitting.

Luckily, this is one of the easiest hyperparameters to optimise, and does not have to be included in a grid search. With the use of a validation set that the network is not trained on the performance of each model on unseen data can be measured after every epoch. Unseen data is necessary because evaluating a model on training data will give overfitted models excellent performance, while the actual model generalises poorly and is unusable. If the performance has improved the state of the model is saved. After a certain number of epochs without the performance improving the training is stopped, and the best performing model state is loaded.

3.3 Data processing

An important part of a machine learning project is the pre-processing of the data. This is used to ensure good model performance and to partition the data to allow unbiased evaluation.

3.3.1 Datasets

MNIST handwritten digit database

The MNIST dataset is one of the most popular in machine learning, being used to benchmark new models is many papers.

Samples	60,000 train & 10,000 test
Inputs	28x28 b/w images
Number of classes	10 - digits 0-9
Balanced	Yes

Table 3.1: MNIST dataset

In this project it is again used for benchmarking, and for ensuring that the models are performing similarly to their original implementations. It does have several similarities

with gene expression data though, being highly dimensional, but with only a few features being relevant to classification. In general, only a very few pixels are actually needed to distinguish between the different digits (and most pixels near the edge of the image are completely useless, being black in almost all the images), while in gene expression it is usually only a few genes that control each phenotype.

The Cancer Genome Atlas

The Cancer Genome Atlas is a project cataloguing sequencing data for several different types of cancer. The data generated by the TCGA Research Network (https://www.cancer.gov/tcga) is available online at https://portal.gdc.cancer.gov. In this project I used data generated using RNA-Seq to attempt to classify the different types of cancer using only their gene expression.

Samples	11,060
Inputs	20,350 genes with values given as $\log_2(\text{TPM}+1)$ [2]
Number of classes	33 - different cancer types
Balanced	No

Table 3.2: MNIST dataset

Some of the samples in this dataset are missing gene expression values, and these samples are not used for comparing semi-supervised model performance. However imputation of missing data is an important topic in bioinformatics and so I cover this in Section 3.3.3.

3.3.2 Data normalisation

Data normalisation can help neural networks by allowing gradient descent to reach a minima more easily. Features having large ranges can result in larger gradients of the loss, resulting in larger steps being taken at the weights. This can cause the gradient descent to oscillate around minima and take far longer to reach a good value. The two most common forms of data normalisation are standardisation and normalisation to the range [0-1]. Having all the features with similar scales is also important for saliency computation, as different scales result in different gradients of the output with respect to the input, preventing them from being directly comparable.

- **Standardisation** involves scaling all the features so that they have mean 0 and variance 1. This is done be computing the mean and standard deviation of the features from the training set, and then subtracting the mean from each feature and

[2] TPM is transcripts per million, where the total number of reads mapped to a gene is normalised by the length of the gene

dividing by the standard deviation.

$$x_{std} = \frac{x - \mu_x}{\sigma_x} \tag{3.1}$$

- **Normalisation** into the range [0-1] is done by finding the maximum and the minimum for every feature from the training set, subtracting the minimum from every feature and dividing by the maximum minus the minimum.

$$x_{norm} = \frac{x - x_{max}}{x_{max} - x_{min}} \tag{3.2}$$

For the MNIST dataset I used normalisation, as the pixels take values between 0 and 255 and so scaling this to be between 0 and 1 is a simple and intuitive transform. It is also the transform used in the code for the ladder network paper [20], and similar to the transform used in the semi-supervised VAE paper [15] (here they set pixel values to either zero or one, without the range inbetween).

For the gene expression datasets I tried both standardisation and normalisation. Standardisation worked well for the non-VAE based models, but the M1 and M2 models experienced significant numeric instability, resulting in the variances of the normal distribution in the latent dimension becoming very small, which resulted in the KLD loss exploding. This eventually lead to the output of the autoencoder becoming `nan`. From what I could discern this is a somewhat common problem in VAEs, especially when the dimensionality of the data is higher than the number of samples in the dataset. The VAEs responded well to normalised data, while the other models experienced similar or worse performance. Therefore standardisation was used for the MLP, SDAE and ladder network and normalisation was used for M1 and M2.

3.3.3 Data imputation

The TCGA dataset contains the gene expression levels for 20,350 genes, but in many of the samples some of these genes are missing. This can be dealt with by simply discarding the samples with missing genes, but this can be a significant number of samples, and removing a large chunk of the dataset is not a good way to improve a model. Instead it may be possible to impute the missing genes, or drop the genes entirely, reducing the number of features but keeping the number of samples. In Section 4.2.1 there is a comparison of the performance of dropping the genes, replacing the missing values with the feature mean, and replacing the missing values with zero.

3.3.4 Data partitioning

Evaluating models and hyperparameter optimisation

Evaluating the performance of a machine learning algorithm should be done on a test dataset that is separate from the dataset the algorithm is trained on, to prevent overfitting leading to overestimating the performance of the model. If the dataset is not particularly large this can be done using **k-fold cross validation**, where the data is split into k different partitions. $k-1$ of these partitions are used for training, while the kth is used to test the performance after training. The accuracy for the model is then computed by taking the average of all the accuracies for each fold.

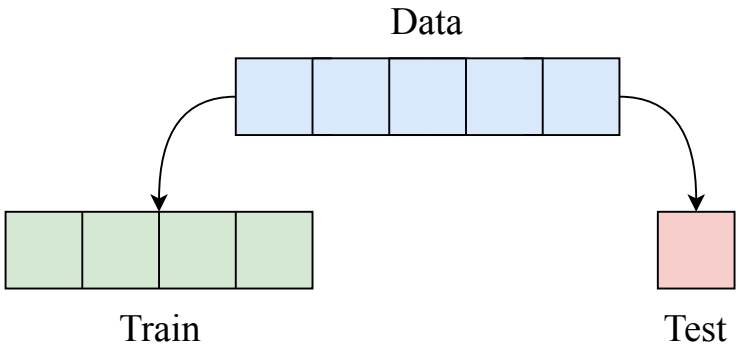


Figure 3.4: One split of 5-fold cross validation

The k-fold cross-validation used in this project is **stratified**. This keeps the proportion of each class in each fold close to the proportion of each class in the overall dataset. This is especially important when classes are unbalanced, as performing non-stratified cross-validation could result in biasing the model towards an uncommon class, or not including any samples of a class in the fold. It has also been shown by Kohavi [16] stratified cross-validation generally has lower bias and variance when estimating model accuracies than non-stratified.

In order to perform hyperparameter optimisation (Section 3.2) there needs to be a validation dataset, to compare the performance of each set of hyperparameters. Performing hyperparameter optimisation on the test set will lead to overestimating the ability of the model to perform on unseen data, as the hyperparameters have been chosen to perform best on the test set. Therefore, another split has to be made to generate a validation set. The most common way to do this is **nested k-fold cross validation**: partition the training data into another k folds; compute the validation performance for each set of hyperparameters k times; take the hyperparameters that perform best on average and train a model with those hyperparameters on all the training data. However this method has a couple of problems. Firstly if there are n hyperparameter sets to test over this method will take nk^2 iterations, and even if a model is quick to train the time this takes quickly becomes very large. It also means that there is no validation set used for the final computation of the model. As I am using early stopping a validation set is always required

because the number of epochs to train to get best performance can be quite variable and depend on the model weight initialisation.

Therefore I instead partitioned the kth fold in two, into a validation set and a test set. The hyperparameters are optimised using the validation set, and the performance of each model is compared. The best performing model is then run on the test set and the acccuracy recorded. The test and validation sets are then swapped and optimisation is performed again. This reduces the number of iterations to $2nk$ and also ensures that there is always a validation set to perform early stopping, while still using all the available data as train, validation, or test data at some point.

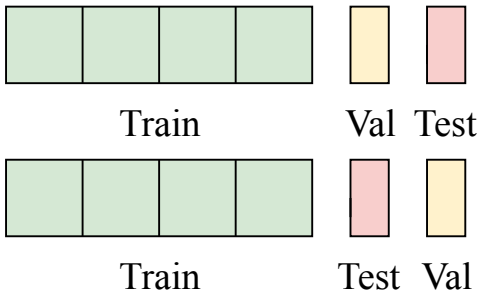


Figure 3.5: Splitting test set into a test and val set for hyperparameter optimisation

Labelled splits

While the datasets used in this project include labels for all the samples, the focus of this project is on semi-supervised learning. In order to give an overall view of the performance for each dataset I performed evaluation of the models with different amounts of labelled data. For each number of labelled samples (n) used, I selected n samples from the train set in a stratified way to use as the labelled dataset. The unlabelled dataset was then all the remaining samples in the train set, and these were assigned the dummy label -1 to ensure that they could not accidentally be used in supervised training.

Parallelisation

While the cross-validation changes made above reduce the number of iterations to a more manageable size, even a model with a fairly short training time will take a long time to complete the evalutaion. To this end I computed the indices of test/val/train and labelled splits I would use in advance, and serialised these using `pickle` [3]. This allows all the models to use the exact same folds, allowing for a fair comparison. It also allows each script to run hyperparameter optimisation over only one fold. The Cambridge High Performance Cluster has 90 GPUs, and so running many smaller jobs in parallel results in a much quicker finish time than running several larger jobs. To this end I also wrote several `bash` scripts to schedule slurm jobs for each model, number of labelled examples and fold.

[3] `pickle` is the most popular Python library for serialisation

3.4 Saliency

Neural networks often act as a black box, with no indication to the programmer or user of what the network is doing, and what it deems to be important. One way of extracting that is through the use of saliency maps (first introduced in [22]), which intend to give the user some idea of which inputs are the most important in determining the output class. The way this is done is by computing the partial derivatives of the input data with respect to the output class of the network. If the input values have been scaled to have similar ranges then this partial derivative should be larger for for inputs that are more important to determining the class. A large positive partial derivative means that increasing the value of that input makes the class more likely, and a large negative partial derivative means increasing that input decreases the likelihood of that class.

A variant of this uses guided backpropagation [23], a variant of backpropagation where the gradient is only backpropagated through a ReLU if the error gradient is greater than zero. When used to construct saliency maps for images it has been shown to make much clearer images, and highlight better the features the network thinks are important.

I implemented both of these methods, basing my implementation on this very simple implementation: https://github.com/Ema93sh/pytorch-saliency. The guided backpropagation version works by registering a hook on each ReLU module that sets the gradient being backpropagated through it to zero if it is less than zero. The code below implements this: by using the saved forward ReLU outputs as a mask, and setting any negative gradient input values to zero it ensures that the backpropagated gradient leaving the rectifier is only non-zero if both the forward pass output and the backwards pass input gradient are greater than zero.

```
def relu_backward_hook_function(self, module, grad_in, grad_out):
    """
    If there is a negative gradient, change it to zero
    """
    # Get last forward output
    corresponding_forward_output = self.forward_relu_outputs[-1]
    positive_mask = (corresponding_forward_output > 0).type_as(
        corresponding_forward_output)

    modified_grad_out = positive_mask * torch.clamp(grad_in[0], min=0.0)

    del self.forward_relu_outputs[-1]  # Remove last forward output
    return (modified_grad_out,)
```

3.5 Command line tool

One of the success criteria of this project was to build a tool, and so the `main.py` file in the repository provides a command line tool for building a semi-supervised learning model. The tool uses many of the techniques described earlier in this section, but many of

the decisions made are based on results in the evaluation section, so an overall description of the tool is available in Section 4.3.

3.6 Repository overview

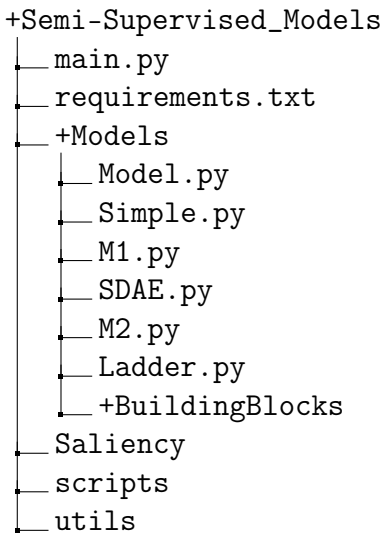


Figure 3.6: Directory structure of the repository

The top level of the repository contains the files `main.py` and `requirements.txt`. These is the script for running the tool, and the packages I am using in the virtual environment respectively. `Models` contains all the code for the models used in this project, including the base modules in `BuildingBlocks` and the abstract class `Model.py`. `scripts` contains all the python and bash scripts used in obtaining the results for the evaluation section, but this folder was created for tidiness and the scripts have to be moved up one level into `Semi-Supervised_Models` to be run. `utils` contains functions for loading data, along with the code for early stopping and making the MNIST and TCGA folds. `Saliency` contains all the code for generating saliency maps for MNIST.

Chapter 4

Evaluation

In this section I will compare the performance of the semi-supervised models on both the MNIST and TCGA datasets, and will compare these to a fully supervised MLP operating on only the labelled data. I then discuss how these factored into the choices made for a more general tool.

4.1 MNIST results

The MNIST results are important for establishing whether the ladder and M2 models have been correctly implemented, as they should achieve close to paper accuracy. It should also give some indication of how the models are expected to perform on the gene expression data.

The MNIST dataset has a designated train and test set, so to obtain the results for this I performed 5-fold cross validation over the training set, using 48,000 examples for testing and 12,000 for validation and then computing the accuracy of the model on the 10,000 test samples. Accuracy is computed as the number of datapoints in the test set where the label was correctly predicted divided by the total number of datapoints in the test set, given below as a percentage of points correctly classified.

Number of labelled samples	**Models**				
	MLP	**SDAE**	**M1**	**M2**	**Ladder**
100	71.75 ± 0.82	74.86 ± 0.85	49.00 ± 0.98	92.34 ± 0.52	96.64 ± 0.35
1000	88.86 ± 0.62	91.97 ± 0.53	83.44 ± 0.73	96.57 ± 0.36	98.03 ± 0.27
3000	93.88 ± 0.47	95.60 ± 0.40	88.55 ± 0.62	97.19 ± 0.32	98.35 ± 0.25
All	98.41 ± 0.25	98.58 ± 0.23	90.30 ± 0.58	98.45 ± 0.24	98.95 ± 0.20

Table 4.1: MNIST 5-fold cross-validation percentage accuracies

4.1.1 Performance comparisons

The MLP and SDAE do not have standard accuracies that can be obtained from papers, but the slight performance boost the SDAE provides over the MLP when only some of the data is labelled is what was expected.

The results for M1 are much worse than those in the original Kingma paper [15], and this is due to the use of a neural network as the classifier on the latent dimension. This neural network can only use the labelled samples, while the Kingma paper uses a transductive support vector machine, which is able to utilise the unlabelled samples in seperating the classes. In order to not overcomplicate the project I decided against implementing a TSVM and so the results are poor. Part of the reason the results are worse than even the basic MLP are that the supervised learning cannot effect the weights of the VAE. If the VAE is encoding useless information the supervised update steps are unable to prevent this.

The results for M2 are comparable to those in the Kingma paper, and for the 100 labelled samples are actually better, averaging 92.34 compared to 88.03. I believe that these improvements are due to performing hyperparameter optimisation over the latent dimension of the model, whereas the Kingma paper used a fixed latent size of 50.

The ladder results are again comparable, with my implementation averaging 96.64 compared to 98.94 by Rasmus [20] for 100 labelled samples. The discrepancy is likely due to the slight differences in model structure. I optimise over 1 to 4 hidden layers of fixed size, while Rasmus uses 5 hidden layers with an overcomplete (1000 neurons) first hidden layer.

4.2 TCGA results

4.2.1 Missing data

As some of the samples in the TCGA dataset are missing expression levels for certain genes I decided to compare the results (using an MLP and all of the data as labelled data) of three different techniques for removing the missing values. The first technique was to just drop all the columns of genes with any missing values, reducing the number of genes from 20,350 to 16334. The second technique involved replacing all the missing gene expression values with zero, and the third involved replacing them with the mean of the expression level for all the samples. The results are summarized in the table below:

Drop genes	**Zero**	**Mean**
95.09 ± 0.40	95.67 ± 0.38	95.81 ± 0.37

Table 4.2: TCGA data imputation 10-fold cross-validation percentage accuracies

Computing a paired t-test helps discern if any one method could be considered better.

Mean/Drop genes	**Mean/Zero**	**Drop genes/Zero**
1.723	0.543	−1.753

Table 4.3: t statistics for difference between imputation folds

None of these t statistics are statistically significant using a two-tailed t-test with p=0.05, and so we cannot reject the null hypothesis that all the imputation methods have the same performance. However, dropping the genes makes the least sense, as that way some real data is lost.

4.2.2 Semi-supervised results

The results in this section were obtained by computing cross-validation as in Section 3.3.4. All samples with missing data were dropped from the dataset to avoid bias due to the imputation type.

One caveat with these results is that they represent an upper bound on the performance of the models, as a validation set is used that is sometimes larger than the number of labelled examples in the training set. This is obviously not feasible in real life, and so the results when used in a production setting will likely not be as good. However, this validation set was required for a fair comparison, as the number of layers and epochs that get the best performance from each model greatly differs from model to model.

Accuracy

Number of labelled samples	**Models**				
	MLP	**SDAE**	**M1**	**M2**	**Ladder**
100	77.54 ± 0.85	78.69 ± 0.84	68.83 ± 0.95	$\mathbf{90.92} \pm 0.59$	87.56 ± 0.67
500	91.09 ± 0.58	91.32 ± 0.57	91.08 ± 0.58	$\mathbf{93.73} \pm 0.49$	93.26 ± 0.51
1000	93.02 ± 0.52	93.53 ± 0.50	92.20 ± 0.55	93.90 ± 0.49	$\mathbf{94.08} \pm 0.48$
All	96.47 ± 0.37	96.13 ± 0.39	93.67 ± 0.50	96.53 ± 0.37	$\mathbf{96.69} \pm 0.37$

Table 4.4: TCGA 10-fold cross-validation percentage accuracies

As can be seen from the results in the table there is no performance benefit in using the M1 model over the simple MLP, but the accuracy achieved nevertheless shows that M1 is probably learning an informative manifold. The SDAE shows a slight performance advantage, but clearly performs less well than M2 and the ladder.

The ladder and M2 both show a significant performance advantage over the MLP, and by computing the a t-test between the three models it can be shown which models perform statistically better with each amount of labelled data.

Number of labelled samples	**MLP/M2**	**MLP/Ladder**	**M2/Ladder**
100	−20.61	−13.40	5.36
500	−7.86	−6.47	1.98
1000	−2.89	−4.59	−1.41
All	−0.30	−2.36	−1.03

Table 4.5: TCGA 10-fold t-statistics between MLP, ladder and M2

The significance threshold for a two fold t-test, using p=0.05 and 9 degrees of freedom, is 2.26. Therefore, looking at these results the ladder network significantly outperforms the MLP at every amount of labelled data. The M2 model outperfoms the MLP significantly until all the data is labelled. At this point the performance should be almost exactly the same as the MLP, as the training of the classifier in the M2 model is only controlled by the supervised cross entropy loss. The ladder is outperformed by the M2 model with 100 labelled samples, but with more labelled samples the difference between them is not statistically significant.

Matthews correlation coefficient

Accuracy is not always the best metric to use when evaluating a machine learning model, expecially when the classes in the model are imbalanced, as it can mean that predicting just the most populous classes results in good accuracy. For binary classification one of the preferred methods is using the Matthews correlation coefficient, which is regarded as a balanced measure even when the classes are unbalanced because it takes into account the balance ratios of the four possible categories (true positive, true negative, false positive, false negative) [3]. There is a multiclass variant of this coefficient that ranges between a minimum value of between −1 and 0 (depending on the distribution) and +1, with +1 meaning perfect correlation between the predicted and actual labels and 0 meaning no correlation. This is calculated as below for k classes:

$$\mathrm{MCC} = \frac{\sum_k \sum_l \sum_m C_{kk} C_{lm} - C_{kl} C_{mk}}{\sqrt{\sum_k (\sum_l C_{kl})(\sum_{k'|k' \neq k} \sum_{l'} C_{k'l'})} \sqrt{\sum_k (\sum_l C_{lk})(\sum_{k'|k' \neq k} \sum_{l'} C_{l'k'})}} \tag{4.1}$$

Calculating this for M2 and the ladder network gives:

Number of labelled samples	**M2**	**Ladder**
100	0.9041	0.8683
500	0.9337	0.9286
1000	0.9354	0.9372
All	0.9631	0.9649

Table 4.6: Multiclass Matthews correlation coefficient

This is actually not much more informative than the accuracy scores, giving very similar results.

Confusion matrix

Viewing the confusion matrix can sometimes give a better insight into where a model is successful and unsuccessful. A confusion matrix has the actual labels on the y-axis and the predicted labels on the x-axis. Each cell contains the number of datapoints with the real label on the y-axis that have been predicted the label on the x-axis.

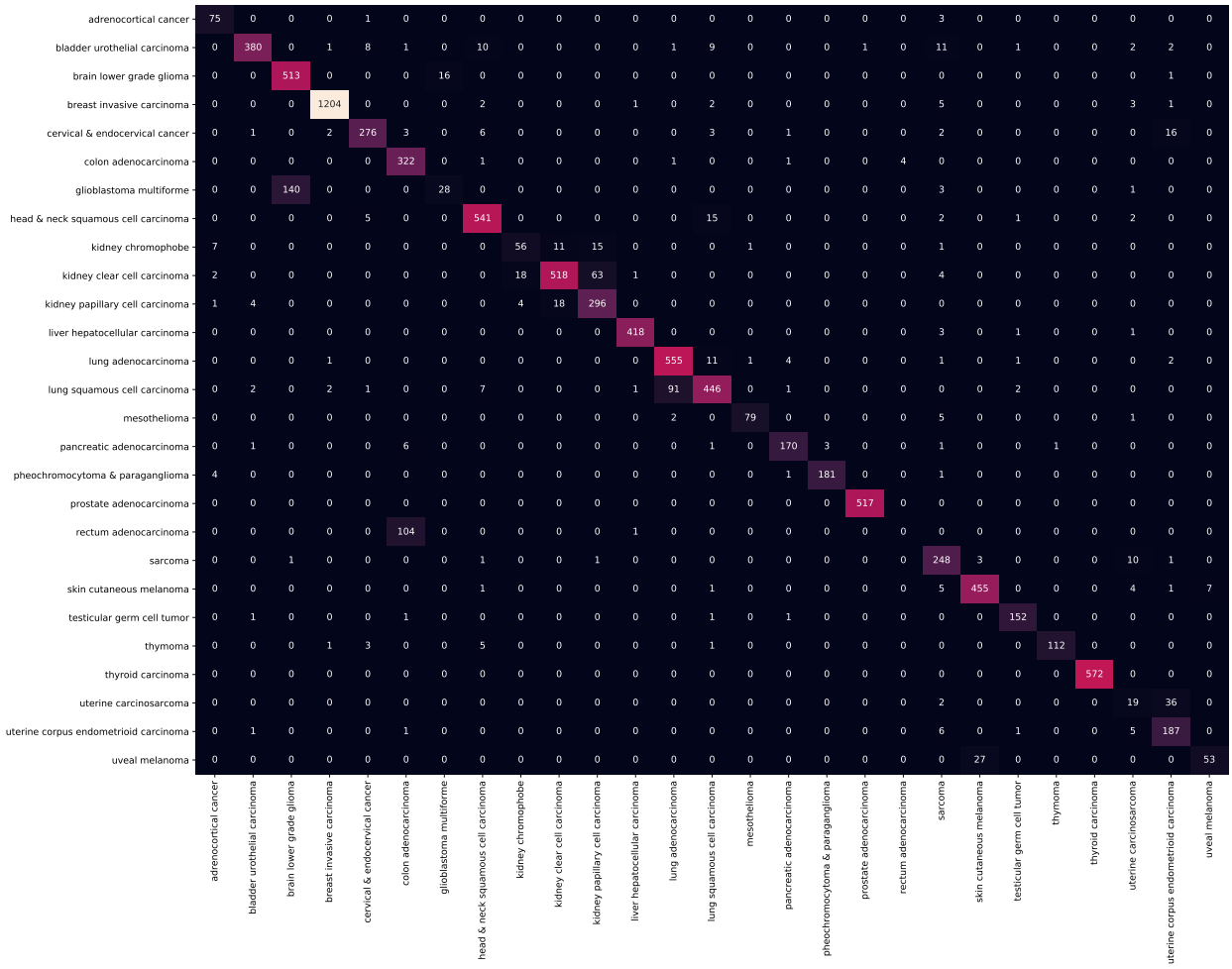


Figure 4.1: Confusion matrix for M2 model with 100 labels

The confusion matrices for the TCGA data are very large due to the large number of classes in the data, and so only this one is included here. A larger copy, and the confusion matrix for the ladder network can be found in Appendix

Both the ladder and M2 matrices are very similar, and show the same weaknesses. *Rectum adenocarcinoma* is misclassified almost 100% of the time by both models, as it is a very similar cancer to *colon adenocarcinoma*, with both being the same type of cancer occuring in a very similar location in the body. The other often misclassified cancers include *brain lower grade glioma* and *glioblastoma mutliforme*, both gliomas located in the brain with differing severity, and *uterine carcinosarcoma* and *uterine corpus endometriod carcinoma*.

The difficulty in classifying these using gene expression is to do with the similarities in gene expression within cells of a certain tissue. There is also the fact that the boundary between the grades of cancer cannot always be well defined, depending on a range of factors including rate of growth and necrosis that can cause the cancer to move between grades.

However, when the number of labelled examples is increased these misclassifications are reduced, with only the rectum and colon adrenocarcinoma significantly misclassifed with all the labels.

Ensemble learning

One way of improving the performance of a classification model is combining the classifications of two different models which may have learned slightly different features and class boundaries. The combination of the models can then lead to an improvement in the accuracy as the different features and boundaries can combine to make a better classifier. The simplest way of doing this is to take the softmax output of the different models and sum them together. Dividing by the number of models then gives a new probability distribution that is the average of the previous models. Taking the maximum of this distribution is the predicted label, and if the models have learned complementary features this will improve the accuracy. Combining the M2 and ladder predictions seems unlikely to result in a large boost in performance due to the similar successes and failures witnessed in their confusion matrices, but the results are below:

Number of labelled samples	**Accuracy**	**MCC**
100	90.89 ± 0.59	0.9036
500	94.00 ± 0.49	0.9365
1000	94.22 ± 0.48	0.9388
All	96.82 ± 0.36	0.9662

Table 4.7: Accuracy and MCC for an average of M2 and ladder

These results are statistically significant improvements over M2 and the ladder network for different amounts of labelled data.

Number of labelled samples	**Ensemble/M2**	**Ensemble/Ladder**
100	−1.17	5.34
500	4.11	3.39
1000	3.42	1.08
All	2.24	0.87

Table 4.8: TCGA 10-fold t-statistics between Ensemble, M2 and ladder

This combined model performs significantly better than M2 (at p=0.05 level) on for all except 100 labelled samples, where it does not perform significantly worse. It is significantly better than the ladder network for both 100 and 500 labeled samples.

It seems that the model therefore is the best of both worlds, outperforming the individual best models, and therefore will factor into the decision of how a more general tool should be built.

4.3 Making a tool

The information in the previous section informed the choices I made in constructing a tool, found in `main.py`.

In the interest of simplicity the tool is a simple command line tool where the only arguments a user provides are the file to load the data from, the mode the tool should be in and the name of the folder to save any outputs in.

There are two modes, train and classify. In both the data is first loaded in, and if there are any missing values they are imputed using mean-value imputation to avoid losing data.

In train mode the labelled data is first split in two, to perform a very simple 2-fold cross validation to find the best hyperparameters for two models, M2 and the ladder. The validation accuracies are also saved to a file called `accs.csv`, to give the user some indication of how the model is likely to perform. Once the hyperparameters have been selected, both models are trained using these parameters on the entire dataset. The models are then saved to the output folder.

In classify mode the models are loaded from the specified output folder, and the data is passed through both. The outputs are combined as described in Section 4.2.2, and the results are saved to the outputs folder.

Diagram

Chapter 5

Conclusion

This dissertation has discussed the researching, implementation and evaluation of several semi-supervised models for use with genomic data. It has also discussed techniques for processing data and improving model performance, and has used the lessons learned from this to construct a simple command line tool.

5.1 Success Criteria

All the major success criteria of the project have been achieved.

The implementations of the models performed as well as their original paper implementations on the MNIST dataset, showing that they were implemented correctly. All the models are now available online at https://github.com/Clondon98/Semi-Supervised_Models, and include the best performing PyTorch implementation of the semi-supervised VAE and ladder that I was able to find online. This repository also allows the models to be easily used on different datasets, while other implementations are restricted to MNIST.

The M2 and ladder model also succeeded in achieving statistically significant performance improvements over the fully supervised model on gene expression data, and an average of both of the models achieved even higher performance. This then led to the development of a command line tool that can be trained and used for classification very simply, without any knowledge of machine learning techniques being required.

I also managed to implement an extension that calculated the saliency of inputs to the network, and while time ran out to include it in the tool its results on MNIST showed that it was locating important features for the classification of the characters.

5.2 Further Work

Some areas of the project that could be explored further include:

- Using VAEs to generate additional unlabelled datapoints and adding these to the dataset to see if they improve performance.
- Using a semi-supervised VAE to generate additional labelled datapoints and seeing if these help improve supervised learning performance.
- Discovering the most important genes for phenotypes using saliency and comparing those to highly important genes found through other methods.

5.3 Final remarks

I hope that the models used in this project can be refined further to find real use in the fields of medicine and biology. The models and tool developed in this project are all freely available online and will hopefully be of use to future researchers interested in semi-supervised learning.

Bibliography

[1] Yoshua Bengio. "Practical Recommendations for Gradient-Based Training of Deep Architectures". In: *Neural Networks: Tricks of the Trade - Second Edition*. 2012, pp. 437–478. DOI: 10.1007/978-3-642-35289-8_26. URL: https://doi.org/10.1007/978-3-642-35289-8%5C_26.

[2] Jason Brownlee. *Why One-Hot Encode Data in Machine Learning?* (Accessed on 04/18/2019). 2017. URL: https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/.

[3] Davide Chicco. "Ten quick tips for machine learning in computational biology". In: *BioData Mining* 10.1 (Dec. 8, 2017), p. 35. ISSN: 1756-0381. DOI: 10.1186/s13040-017-0155-3. URL: https://doi.org/10.1186/s13040-017-0155-3.

[4] Dumitru Erhan et al. "Why Does Unsupervised Pre-training Help Deep Learning?" In: *J. Mach. Learn. Res.* 11 (Mar. 2010), pp. 625–660. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1756006.1756025.

[5] E. R. Gibney and C. M. Nolan. "Epigenetics and gene expression". In: *Heredity* 105 (May 12, 2010). Review, URL: https://doi.org/10.1038/hdy.2010.54.

[6] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics, AISTATS 2010, Chia Laguna Resort, Sardinia, Italy, May 13-15, 2010*. 2010, pp. 249–256. URL: http://jmlr.org/proceedings/papers/v9/glorot10a.html.

[7] Guray Golcuk, Mustafa Anil Tuncel, and Arif Canakoglu. "Exploiting Ladder Networks for Gene Expression Classification". In: *Bioinformatics and Biomedical Engineering*. Ed. by Ignacio Rojas and Francisco Ortuño. Cham: Springer International Publishing, 2018, pp. 270–278. ISBN: 978-3-319-78723-7.

[8] Anna Goldenberg et al. "Dr.VAE: improving drug response prediction via modeling of drug perturbation effects". In: (Mar. 2019). DOI: 10.1093/bioinformatics/btz158. eprint: http://oup.prod.sis.lan/bioinformatics/advance-article-pdf/doi/10.1093/bioinformatics/btz158/28492125/btz158.pdf. URL: https://doi.org/10.1093/bioinformatics/btz158.

[9] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.

[10] Sean Holden. *Lecture slides for Artificial Intelligence I*. 2018. URL: https://www.cl.cam.ac.uk/teaching/1718/ArtInt/ai1-18.pdf.

[11] Sean Holden. *Lecture slides for Machine Learning and Bayesian Inference*. 2018. URL: https://www.cl.cam.ac.uk/teaching/1718/MLBayInfer/ml-bayes-18.pdf.

[12] Brian Keng. *Semi-supervised Learning with Variational Autoencoders | Bounded Rationality*. 2017. URL: http://bjlkeng.github.io/posts/semi-supervised-learning-with-variational-autoencoders/.

[13] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*. 2015. URL: http://arxiv.org/abs/1412.6980.

[14] Diederik P. Kingma and Max Welling. "Auto-Encoding Variational Bayes". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*. 2014. URL: http://arxiv.org/abs/1312.6114.

[15] Diederik P. Kingma et al. "Semi-Supervised Learning with Deep Generative Models". In: *CoRR* abs/1406.5298 (2014). arXiv: 1406.5298. URL: http://arxiv.org/abs/1406.5298.

[16] Ron Kohavi. "A Study of Cross-validation and Bootstrap for Accuracy Estimation and Model Selection". In: *Proceedings of the 14th International Joint Conference on Artificial Intelligence - Volume 2*. IJCAI'95. Montreal, Quebec, Canada: Morgan Kaufmann Publishers Inc., 1995, pp. 1137–1143. ISBN: 1-55860-363-8. URL: http://dl.acm.org/citation.cfm?id=1643031.1643047.

[17] Hugo Larochelle et al. "Exploring Strategies for Training Deep Neural Networks". In: *Journal of Machine Learning Research* 10 (2009), pp. 1–40. URL: https://dl.acm.org/citation.cfm?id=1577070.

[18] Yann LeCun and Corinna Cortes. "MNIST handwritten digit database". In: (2010). URL: http://yann.lecun.com/exdb/mnist/.

[19] Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted Boltzmann Machines". In: *Proceedings of the 27th International Conference on International Conference on Machine Learning*. ICML'10. Haifa, Israel: Omnipress, 2010, pp. 807–814. ISBN: 978-1-60558-907-7. URL: http://dl.acm.org/citation.cfm?id=3104322.3104425.

[20] Antti Rasmus et al. "Semi-Supervised Learning with Ladder Networks". In: *CoRR* abs/1507.02672 (2015). arXiv: 1507.02672. URL: http://arxiv.org/abs/1507.02672.

[21] Eyal Reingold. *What are Artificial Neural Networks*. (Accessed on 04/18/2019). URL: http://www.psych.utoronto.ca/users/reingold/courses/ai/cache/neural2.html.

[22] Karen Simonyan, Andrea Vedaldi, and Andrew Zisserman. "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps". In: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Workshop Track Proceedings*. 2014. URL: http://arxiv.org/abs/1312.6034.

[23] Jost Tobias Springenberg et al. "Striving for Simplicity: The All Convolutional Net". In: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Workshop Track Proceedings*. 2015. URL: http://arxiv.org/abs/1412.6806.

[24] *SVI Part I: An Introduction to Stochastic Variational Inference in Pyro — Pyro Tutorials 0.3.1 documentation*. 2017. URL: https://pyro.ai/examples/svi_part_i.html.

[25] V. Teixeira, R. Camacho, and P. G. Ferreira. "Learning influential genes on cancer gene expression data with stacked denoising autoencoders". In: *2017 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*. Nov. 2017, pp. 1201–1205. DOI: 10.1109/BIBM.2017.8217828.

[26] Harri Valpola. "From neural PCA to deep unsupervised learning". In: *CoRR* abs/1411.7783 (2014). arXiv: 1411.7783. URL: http://arxiv.org/abs/1411.7783.

[27] Pascal Vincent et al. "Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion". In: *J. Mach. Learn. Res.* 11 (Dec. 2010), pp. 3371–3408. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1756006.1953039.

[28] Gregory P. Way and Casey S. Greene. "Extracting a biologically relevant latent space from cancer transcriptomes with variational autoencoders". In: *Pacific Symposium on Biocomputing. Pacific Symposium on Biocomputing* 23 (2018), pp. 80–91.

[29] Richard M. Zur et al. "Noise injection for training artificial neural networks: A comparison with weight decay and early stopping". In: *Medical Physics* 36 (2010). URL: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2771718/.

Appendix A

Backpropagation

Backpropagation is the flow of information from the outputs of the network back through the network. Once the cost function has been computed finding the gradient of the loss with respect to each trainable parameter shows which direction the parameter can be shifted to decrease the loss function.. This derivation of the backpropagation algorithm is based on that found in *Artificial intelligence I* [10]. In the derivations below:

- $J(\boldsymbol{\theta})_k$ is the cost function for the kth sample in the training set
- $w_{i\to j}$ is the weight between node i and node j
- a_j is the value computed by the node j pre-activation ($\sum_k w_{k\to j} z_k$)
- z_j and y_j are the values post-activation ($\sigma(a_j)$), for non-output and output nodes respectively

The value to be calculated for each weight (per sample) is $\frac{\partial J(\boldsymbol{\theta})_k}{\partial w_{i\to j}}$. To do this we use the chain rule of differentiation:

$$\frac{\partial J(\boldsymbol{\theta})_k}{\partial w_{i\to j}} = \frac{\partial J(\boldsymbol{\theta})_k}{\partial a_j}\frac{\partial a_j}{\partial w_{i\to j}} \tag{A.1}$$

$$= \frac{\partial J(\boldsymbol{\theta})_k}{\partial a_j} z_i \tag{A.2}$$

Then for weights connected to the output nodes:

$$\frac{\partial J(\boldsymbol{\theta})_k}{\partial w_{i\to j}} = \frac{\partial J(\boldsymbol{\theta})_k}{\partial a_j} z_i \tag{A.3}$$

$$= \frac{\partial J(\boldsymbol{\theta})_k}{\partial y_j}\frac{\partial y_j}{\partial a_j} z_i \tag{A.4}$$

If our loss function is differentiable with respect to y_j, and y_j is differetiable with respect to a_j then we can calculate this. For backpropagation to work all our loss and activation functions must have this property. Thankfully all the loss and activation functions shown in this dissertation do have that property and so work correctly with backpropagation.

Once the gradients are computed for the weights in the output layer it is possible to calculate them for the next layer, and then the layer after that, etc., as the errors are backpropagated through the network. The computation for the hidden layers is slightly more complicated:

$$\frac{\partial J(\boldsymbol{\theta})_k}{\partial w_{i\to j}} = \frac{\partial J(\boldsymbol{\theta})_k}{\partial a_j} z_i \tag{A.5}$$

$$= \left(\sum_{k \in K} \frac{\partial J(\boldsymbol{\theta})_k}{\partial a_k} \frac{\partial a_k}{\partial a_j} \right) z_i \tag{A.6}$$

where K is the set of all nodes j is connected to in the next layer

By definition:

$$\frac{\partial a_k}{\partial a_j} = \frac{\partial}{\partial a_j} \left(\sum_i w_{i\to k} z_i \right) \tag{A.7}$$

$$= \frac{\partial}{\partial a_j} \left(\sum_i w_{i\to k} \sigma(a_i) \right) \tag{A.8}$$

$$= w_{j\to k} \sigma'(a_j) \tag{A.9}$$

Therefore, for hidden layers:

$$\frac{\partial J(\boldsymbol{\theta})_k}{\partial w_{i\to j}} = \left(\sum_{k \in K} w_{j\to k} \frac{\partial J(\boldsymbol{\theta})_k}{\partial a_k} \right) \sigma'(a_j) z_i \tag{A.10}$$

As the errors are moving backward through the network, and all nodes k are in the layer after j, $\frac{\partial J(\boldsymbol{\theta})_k}{\partial a_k}$ has already been computed and so this gradient can be calculated.

Appendix B

The reparameterization trick

It is not possible to backpropagate the reconstruction loss of a varitaional autoencoder through the random sampling in the latent dimension, because the drawing of the samples is not differentiable with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$. Previously this was solved using a Monte Carlo gradient estimator, but these have extremely high variance and are impractical.Instead the reparameterization trick allows the Gaussian sample to be reparameterized as:

$$\boldsymbol{z_i} = \boldsymbol{\mu} + \boldsymbol{\sigma}\boldsymbol{\epsilon} \tag{B.1}$$

$\boldsymbol{\epsilon}$ is a vector of samples from the standard normal distribution $\mathcal{N}(0, 1)$. The sampling operation is now differentiable with respect to $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ and therefore is differentiable with respect to $\boldsymbol{\theta}$, the trainable parameters of the encoder. As the gradients can be backpropagated through the network it is possible to train the VAE using gradient descent.

Appendix C

Project Proposal

Computer Science Tripos – Part II – Project Proposal

A tool for phenotype prediction from cell genotype

C. London, Trinity College

Originator: Prof P. Lió & H. Andres Terre

19 October 2018

Project Supervisor: Prof P. Lió & H. Andres Terre

Director of Studies: Prof F. Stajano

Project Overseers: Prof J. Daugman & Dr A. Madhavapeddy

Introduction

Phenotype prediction from cell genotype is an important problem in the field of bioinformatics, with usages in agriculture (for selecting crops with highest yield potential), medicine (for predicting likelihood of diseases/mutations), research, and many other fields. With the decreasing cost of genome sequencing this is becoming far more feasible for use in both research and the private sector.

Deep learning techniques provide an opportunity for generating high accuracy predictions of the phenotype from the genotype and as such much research in this area uses techniques such as autoencoders and neural networks to attempt to do this. However, much of the state of the art research is focused on generating predictions for only one type of cell, and so the network is structured specifically towards this cell. This is then not generalisable and not useful in other scenarios.

This project aims to build a tool that will provide these high accuracy predictions and is generalisable to data from different cells. By providing genotype data tagged with the observed phenotype for training, a user should then be able to use this tool to predict the required phenotype.

Starting point

Prior Research

This project is based on similar state of the art work done in papers on DeepGS[**DeepGS**] (genomic selection) and DeepMetabolism[**DeepMetabolism**].

DeepGS was used to predict several phenotypes (grain length, grain hardness, plant height and more for wheat) when given genomic markers, and is now available as an R library. However the method uses convolution and sampling to reduce data dimensionality, whereas the DeepMetabolism uses unsupervised pre-training in the form of an autoencoder to speed the supervised learning up. I believe that this is the better way to do it, and prescribes a less rigid structure on the model.

DeepMetabolism was used to predict three phenotypes of *Escherichia Coli* and uses unsupervised pre-training with an autoencoder. However the autoencoder is used for denoising purposes, and the output, a cleaned version of the input data, is used for predictio instead of using the latent representation. The autoencoder is also structured specifically to correspond to the genome and proteins of *E. Coli* and so the re-usability is limited.

Libraries and codebase

The code for the learning will be written in Python, because Python has excellent support for advanced deep learning libraries. I have written machine learning code in Python before using Scikit-learn, but for this project I will use either Tensorflow or PyTorch as these libraries provide much better support for deep learning applications, as well as allowing for data parallelism and therefore large speed-ups when running on a GPU.

Resources required

For this project I shall mainly use my own laptop, a 2014 Macbook Pro with a dual core Intel i5, 8GB of RAM and 128GB of flash storage. Source code for the project will be kept in a Git repository that will be synced with a repository on Github. The LaTeX source will be stored on my machine, and backed up to both overleaf.com and Google Drive. The entire content of my laptop will be frequently backed up to an external 500GB HDD.

The training of the model will be done using GPUs that can be provided by the Computer Lab, with confirmation from Professor Lió.

Data for training and testing the model will be both artificially generated as part of the project, and provided by the Plant Sciences department.

Work to be done

Reading and further research

- Re-read papers on DeepGS and DeepMetabolism looking for particular insight into the exact models used to see ways they could be improved.
- Deep learning to model the hierarchical structure of the cell[3] - this is less relevant to the project as it involves a neural network that is pre-structured by hand, but still involves genotype-phenotype prediction and so may contain useful insights.
- Principles of gene manipulation and genomics[**Genomics**] Chapters 16 to 20 - a book about genome analysis, involving genomic and transcriptomic data. This is necessary because I have never worked on a project with large scale genome analysis before.
- Reducing the dimensionality of data using neural networks[**Encoders**] - information on reducing the dimensionality of data using autoencoders.
- Sparse autoencoders (lecture notes by Andrew Ng) - more information about using autoencoders to learn the most important features of data unsupervised.
- Meetings with Prof. Haseloff - Prof. Haseloff is a member of the plant sciences department here at the university and has experience working with the Computer Lab on other bioinformatics projects. He should be able to provide real data to use in training, as well as insight into what the synthetic data should look like.

The model

The model for predicting the phenotype will be split into unsupervised and supervised portions.

The unsupervised portion is used to for dimesionality reduction of data, and is effectively a sort of pre-training for the model. Genomic data can be very large depending on the cell and so reducing the dimensionality to only the most important features should significantly speed the model up, and should hopefully provide greater prediction accuracy. This pre-training is likely to involve passing the genomic data into an autoencoder. An autoencoder is made up of two parts, an encoder and a decoder. The encoder includes the input layer and some hidden layers culminating in a final layer that has fewer nodes than the input layer. This layer is the latent representation of the input data. The decoder then

attempts to reconstruct the input data from this representation, and backpropagation is performed to get the reconstruction as close as possible to the original. This latent representation should then be the features of the data that are most important for accurate reconstruction, and is therefore a reduced dimensionality representation of the data.

The supervised portion of the model will take the form of a neural network that takes in this latent representation as input and outputs the predicted phenotype. Backpropagation is performed to train the network to accurately predict the phenotype.

An important feature of the model should be the ability to work with different sized inputs and outputs. For the model to be generalisable to different cells, it will have to be able to take variable sized genome inputs, as the length of a genome sequence varies greatly between organisms. The output size will also not always be constant as the phenotype can take many forms - sometimes it might simply be a number e.g. if the studied phenotype is plant height, whereas other times it could be a class, or, as written in the extensions below, a photograph.

Testing. Testing the model will first be done with synthetic data, to ensure that it is working as it should i.e. recognising patterns and correctly identifying the most important features in the data, and the phenotype that should be associated with a given latent representation. The generation of this synthetic data will form an early part of this project, and will be done with input from Prof. Haseloff as he has significant experience with biological data.

Once this is completed the model will then be trained and tested on real world data to verify its accuracy and usefulness as a real world prediction tool.

The tool

This project should lead to a tool that can be used to predict phenotype without requiring users to know much about the underlying machine learning. To that end the tool should provide a basic front-end where users can add a file with training data, and, once training has completed, provide data that they want predictions for. The difficulty here is dealing with the different phenotypes a user might want, and having a model that can handle very different output requirements.

Success criteria

Evaluation of the success of the project will be based upon the following criteria:

- Better prediction accuracy than supervised training alone on a synthetic dataset (generated as part of the project) - compare the predicted phenotype with the synthetic phenotype generated for a particular pattern in the synthetic genotype data.

- Better prediction accuracy than supervised training alone on real world genotype and phenotype data.
- Producing a tool with a front-end that takes a file-path and trains the model based on the data within the file.

Prediction accuracy can be compared using a statistical significance test. The models will be trained on the same dataset and then tested on data that was not part of the training set. With synthetic data it is easy to generate data that was not part of the training set. With real world data if there is not enough for a separate training set and test set k-fold cross-validation can be performed.

Possible extensions

If the main result is achieved and time is left some possible extensions are:

- Use images of a cell/plant/etc. as the phenotype for training, and generate images as predictions. This would involve using other deep learning techniques to generate the images, likely generative adversarial networks.
- Measuring the importance of inputs to the output. Neural networks are often considered to be a sort of black box where input goes in and a result comes out. It is possible however to use some techniques to discover the importance of the input to the result. This can then be used to pinpoint which genes were most responsible for a certain phenotype. However, results can be misleading and the accuracy is not always good.
- Autoencoder structure based upon cell structure. Some well-studied organisms have extensive prior knowledge about gene interactions available in resources such as the Gene Ontology which would enabling structuring the autoencoder such that it partially mimics the cell. This should speed up training and could increase prediction accuracy. It would not be available for all cells, but if a user was using a well-studied cell the option to use a structured autoencoder would be useful.

Timetable

Planned starting date is 20/10/2018.

1. **20/10/2018 – 3/11/2018**

   Perform the reading and research as stated in the "Work to be done" section. Install Tensorflow and PyTorch and decide which to use by doing small example learning tasks.

2. **4/11/2018 – 18/11/2018**

Write Python code to generate synthetic data for training. Begin building unsupervised model. Begin writing Introduction and Preparation chapters of dissertation.

3. **19/11/2018 – 3/12/2018**

   Train and tune unsupervised model on synthetic data. Ensure that features expected to be important have large impact. Begin building supervised model. Begin writing Implementation chapter of dissertation.

4. **Michaelmas vacation**

   Do supervised training using output from the unsupervised model as input to the network. Make sure model is generalisable to different sized inputs and outputs corresponding to different cells and phenotypes.

   Milestones:

   - Have a working prediction model trained on the synthetic data.
   - Have finished writing dissertation Introduction and Preparation chapters.

5. **15/01/2019 – 29/01/2019**

   Write progress report. Compare accuracy of model against a simple supervised model on artificial data to ensure model has better accuracy. Begin writing Evaluation chapter of dissertation.

6. **30/01/19 – 13/02/19**

   Train model on real data and compare prediction accuracy with only supervised network.

   Milestones:

   - Progress report submitted on 01/02/19.

7. **14/02/19 – 28/02/19**

   Build a front-end that allows a user to pass a file of training data and create a model. Begin Extension 1: using images as phenotype.

   Milestones:

   - Have a working tool allowing users to provide their own data for training.
   - Have completed the Implementation chapter of the dissertation.

8. **01/03/19 – 15/03/19**

   Extension 2: Pinpointing genes that have the greatest effect on phenotype. Ensure full evaluation of all success criteria. Begin writing dissertation Evaluation chapter.

9. **Easter vacation**

   Writing dissertation and finishing any ongoing extensions. Extension 3 if time allows.

10. **23/04/19 – 7/5/19**

    Proof reading, performing any final changes recommended by supervisor, preparing for submission in order to focus on exams.

    Milestone: Have a completed and checked dissertation.

Bibliography

[1] Ma W. & Qiu Z. (2017) *DeepGS: Predicting phenotypes from genotypes using Deep Learning*

[2] Guo W., Xu Y. & Feng X. (2017) *DeepMetabolism: A Deep Learning System to Predict Phenotype from Genome Sequencing*

[3] Ma J., Yu M., Fong S., & Ideker T. (2018) *Using deep learning to model the hierarchical structure and function of a cell*

[4] Primrose S. & Twyman R. (2006) *Principles of Gene Manipulation and Genomics*

[5] Hinton G. & Salakhutdinov R. (2006) *Reducing the Dimensionality of Data with Neural Networks*