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

April 29, 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 [22].

Likewise, variational autoencoders (Section 2.3.3) have been successfully used to extract a biologically relevant latent space from cancer transcriptomes [25]. 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 [7]. 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 [6].

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** [20]) 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 $x_1 w_1 + ... + x_n w_n$, before applying an **activation function** σ [9].

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:

$$y = \sigma(\boldsymbol{x} \cdot \boldsymbol{w}) \tag{2.1}$$
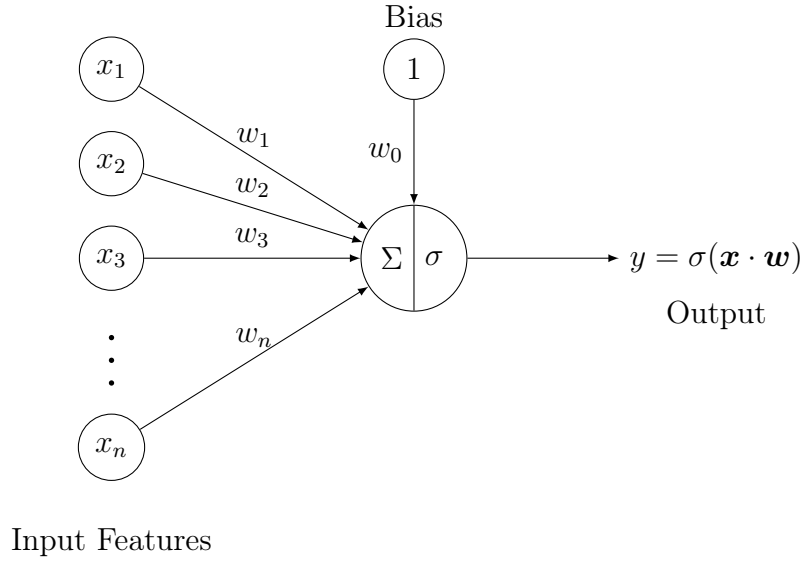
Figure 2.1: A perceptron

The activation function applied will depend on what the neuron is being used for. Some common activation functions and their uses are [9]:

- **Linear**

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

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

- **Sigmoid**

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.3}$$

  Used to give output values in the range $[0, 1]$, for example to give a probability of a class in binary classification (two classes). Was previously used as the non-linear function in hidden layers but has seen reduced usage with the increased popularity of ReLU.

- **ReLU**

$$\sigma(z) = max(0, z) \tag{2.4}$$

  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 [18].

2.1.2   Deep feedforward networks

Deep feedforward networks are made up of multiple perceptrons arranged into **layers** (and are often called multilayer perceptrons). 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. (N.B. I will use the terms deep feedforward network and neural network interchangably).
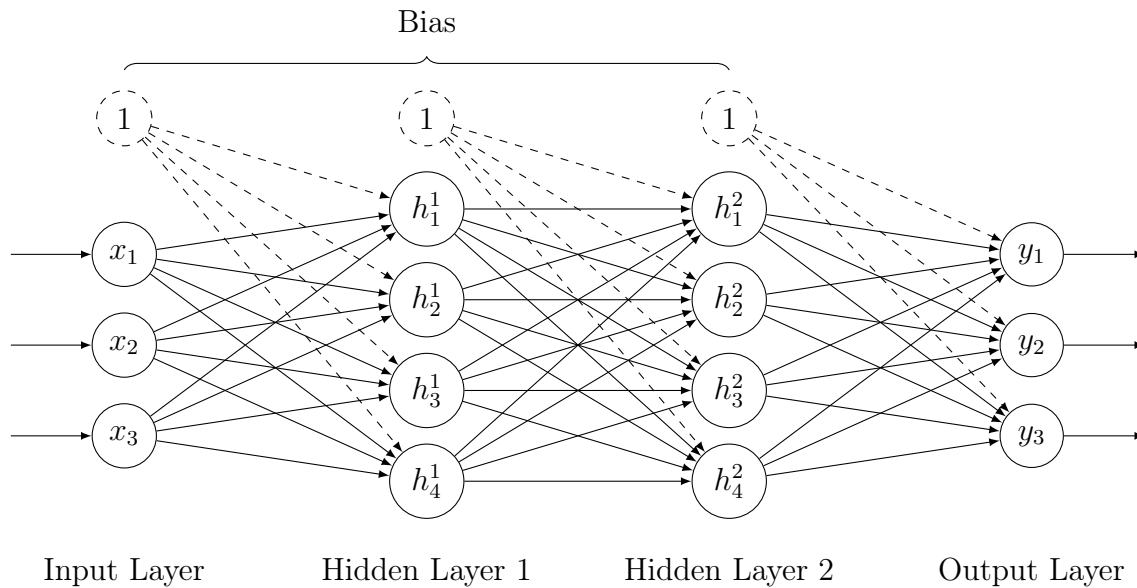
Figure 2.2: Illustration of a 4-layer deep feedforward network

The primary source of the following information is *Deep Learning* by Goodfellow et al. [8]. The naming convention **deep** comes about because the networks contain multiple layers. Deep is usually used to refer to networks with multiple hidden layers. **Feedforward** comes from the fact that information in the network flows forward from layer to layer, with no connections allowing the information to move backwards to previous layers.

The main reason for neural networks with multiple layers is that they can model much more **complex non-linear relationships** than single neurons and single layer models are able to. Using non-linear activation functions such as ReLU (2.4) 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 (in the simplest case the weights).

2.1.3 Supervised learning

There are two main problems that neural networks are usually used to solve: **regression** and **classification** [10]:

- Regression problems involve predicting the value of a quantity that is continuous and real-valued: $f_{\boldsymbol{\theta}}(\boldsymbol{x}) \in \mathbb{R}$.

- Classification problems involve predicting which class (of a pre-determined set) an input belongs to: $f_{\boldsymbol{\theta}}(\boldsymbol{x}) \in [l_0, ..., l_n]$.

Supervised learning involves taking a **dataset** $s = ((\boldsymbol{x_0}, y_0), ..., (\boldsymbol{x_n}, y_n))$ and using that to train a network to find a function that gives good predictions for y given $\boldsymbol{x}$: $f_{\boldsymbol{\theta}}(\boldsymbol{x}) = \hat{y} \approx y$.

Neural networks for classification

This project has focused exclusively on classification problems, and so a more in-depth description is necessary.

A dataset for classification training is given as pairs of numerical **features** and a **label**: $(\boldsymbol{x}, y)$. These features are the input to our network, and the label is our 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 [3].

The output from the network is then a real number for each class. This does not lend itself easily to a loss function (more explanation in next section) and can also be difficult to interpret for humans. Therefore the **softmax** function is used as the activation function in the ouput layer:

$$\sigma(y_j) = \frac{e^{y_j}}{\sum_{i=1}^{K} e^{y_i}}, \text{ for the } jth \text{ output in K output values} \tag{2.5}$$

This give a value in the range $[0, 1]$ for every class, the sum of which is 1. It is now possible to view the output of the model as a probability distribution over the classes [2], which allows the model to encode uncertainty and is easily interpretable by humans. It also lends itself well to a loss function.

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. The difference between the correct value and the network output is called the **loss**. 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 a loss function, which measures how well a neural network is performing on the training set, and updating the parameters to decrease the loss.

Loss functions

The two main loss functions used in this project are **mean square error** and **cross entropy** loss:

- **Mean square error**

$$J(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} (f_{\boldsymbol{\theta}}(\boldsymbol{x}^i) - y^i)^2 \tag{2.6}$$

  The mean square error is the mean of the squared difference between the real values and the predicted values for multiple datapoints. If the output from the network and the correct values are vectors it is calculated by taking the Euclidean distance between them ($|f_{\boldsymbol{\theta}}(\boldsymbol{x}^i) - \boldsymbol{y}^i|^2$) and squaring it. This loss function is defined for all real numbers and so is used in regression problems. It is also used as the reconstruction loss in most of the autoencoders in this project.

- **Cross entropy**

$$J(\boldsymbol{\theta}) = -\frac{1}{n} \sum_{i=1}^{n} \boldsymbol{y}^i \cdot \log f_{\boldsymbol{\theta}}(\boldsymbol{x}^i) \tag{2.7}$$

  This loss function is used for multiclass classification problems. $\boldsymbol{y}^i$ is the one-hot representation of the label, and the scalar product is taken with the logarithm of the softmax output probabilities. This comes out to $\log f_{\boldsymbol{\theta}}(\boldsymbol{x}^i)_j$ where the jth label is the correct one. Taking the negative of this means that if the probability of the correct label is high then the loss is very low ($\lim_{x \to 1} -\log x = 0$) and if the probability is low then the loss is very high ($\lim_{x \to 0} -\log x = \infty$).

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:

$$\boldsymbol{\theta} := \boldsymbol{\theta} - \eta \sum_k \frac{\partial J(\boldsymbol{\theta})_k}{\partial \boldsymbol{\theta}} \tag{2.8}$$

$$:= \boldsymbol{\theta} - \eta \sum_k \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta})_k \tag{2.9}$$

$$:= \boldsymbol{\theta} - \eta \nabla_{\boldsymbol{\theta}} \sum_k J(\boldsymbol{\theta})_k \tag{2.10}$$

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

The basic form of gradient descent outlined above uses one constant learning rate (η) for updating all of the weights in the network. Recently most neural network implementations have instead used **Adam**, an adaptation of gradient descent that holds a learning rate per trainable parameter, and updates this based on the mean and variance of the gradients.

Batch learning

The reason for including a summation term over training set samples in both the loss functions ((2.6) & (2.7)) and the weight update rule (2.10) is because neural networks are generally run over multiple samples each time they are run, rather than being run over a single example. When performing gradient descent there are three main ways:

- **Stochastic** gradient descent is computed over a single sample per iteration.

- **Mini-batch** gradient descent uses multiple samples per iteration, but less than the whole dataset.

- **Batch** gradient descent uses the entire dataset per iteration

The most popular of these approaches is mini-batch, as it has stability advantages over stochastic and computational advantages over batch. 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 sampels 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.

Summary

One complete pass through the dataset is called an **epoch**. 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 2.1 Train neural network via mini-batch gradient descent

---
1: **procedure** TRAINING(i, $\mathcal{D}$, η, `model`, `loss_function`)
2:     **for** i epochs **do**
3:         **for** mini-batch $\mathcal{M}$ in $\mathcal{D}$ **do**
4:             `data, labels` $= \mathcal{M}$
5:             `out` $=$ `model(data)`
6:             $\mathcal{J} =$ `loss_function(out, labels)`
7:             `model.`$\boldsymbol{\theta} =$ `model.`$\boldsymbol{\theta}$ - $\eta\nabla_{\boldsymbol{\theta}}\mathcal{J}$

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.
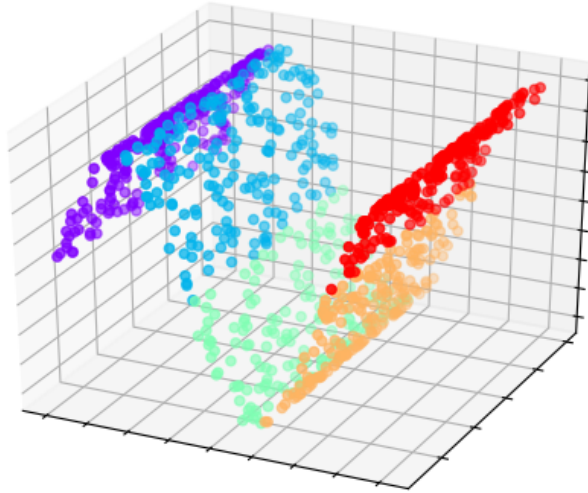
Figure 2.3: Points on a 2-dimensional manifold embedded in 3-dimensional space

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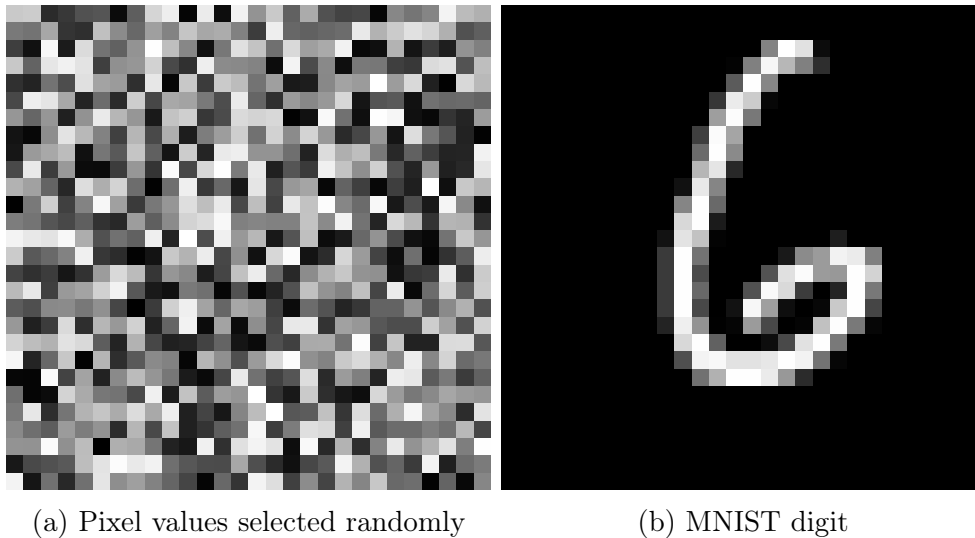
(a) Pixel values selected randomly   (b) MNIST digit

Figure 2.4: 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 seperated 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 $\hat{\boldsymbol{x}} = h_{\phi}(\boldsymbol{z})$. This allows an unsupervised problem to be turned into a supervised problem, using $\boldsymbol{x}$ as the target.
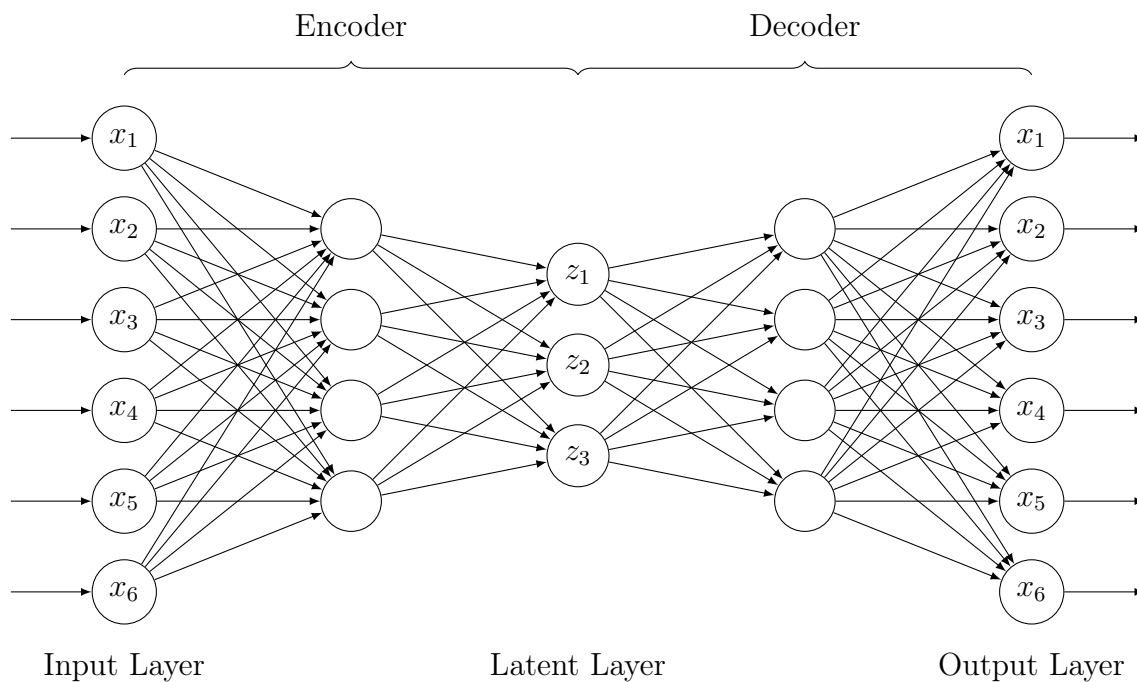
Figure 2.5: Illustration of a simple autoencoder

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 (2.6). 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.
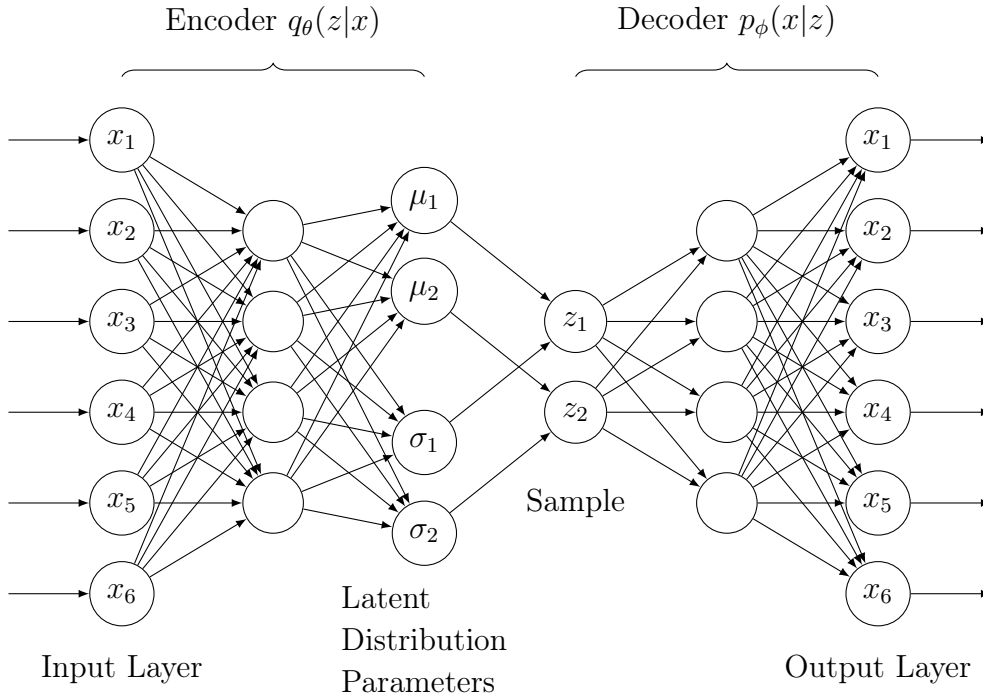
Figure 2.6: Illustration of a Gaussian variational autoencoder

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:

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)} \tag{2.11}$$

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

$$p(x) = \int p(x|z)p(z)dz \qquad (2.12)$$

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 [21].

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

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 (2.6) 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 unsupervised learning tasks. In most fields unlabelled data is much easier to obtain than labelled data. For example in the field of computer vision it is very easy to find huge amounts of unlabelled data on the iternet, but finding labelled data can involve hiring human annotators to label the images, an expensive and time consuming process.

In the field of genomics there are often cases in which data that is generated in different labs can be usefully combined in machine learning models (e.g. The Cancer Genome Atlas). However, the different labs often measure different characteristics of the organisms, and so only a small subset of this combined dataset may have the labels required for a task. In supervised learning the rest of the dataset is now useless, but a semi-supervised model can leverage this data.

Perhaps the simplest semi-supervised learning method is **self-training**. This involves training a classification model on the small amount of available labelled data. This model is then run on the unlabelled data, and the datapoints for which the model is most confident of the label are added to the labelled dataset. The labelled dataset then becomes larger with (hopefully) mostly correct labels giving the model has a larger training dataset. The methods used in this project are described below, with more in-depth explanations in the implementation section.

2.4.1 Dimensionality reduction

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" [24] 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. [13] 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
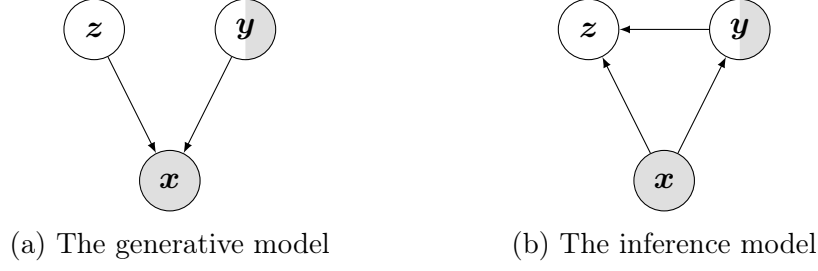
Figure 2.7: The semi-supervised VAE
(shading indicates whether a variable is found in the dataset)

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 (2.5). The summation term this is part of is referred to as "classification as inference" by Kingma [13]. 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 (2.7) between the real label and the label the classifier outputs for the data. This modified version is then:

$$\mathcal{J}(x, y) = \mathcal{L}(x, y) - \alpha(y \cdot \log q(y|x)) \tag{2.13}$$

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 [23], and then expanded upon by Rasmus et al. in 2015 [19]. It can be thought of as building on top of the SDAE, using an unsupervised local denoising criterion in order to improve the supervised performance of a deep feedforward network by using unlabelled data. However, instead of training each layer with its own denoising autoencoder, it instead adds a single deep decoder of the same size as the supervised network to the model, with lateral connections with trainable weights between the equivalent layers in the encoder and decoder.
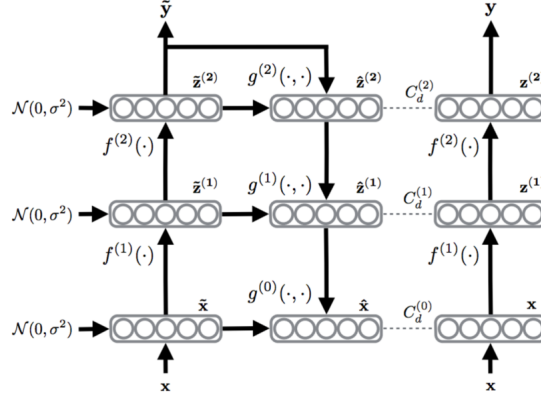
Figure 2.8: Illustration of the ladder network
(Source: Rasmus et al. [19])

As described by Valpola, the ladder network is a form of 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 hierarchies of latent variables. The model contains layers of latent variables, with the higher layers describing *some* of the variance of lower layers by $p(z^{(l)}|z^{(l+1)})$ ($z^{(0)} = x$). In a normal autoencoder 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 some information about x, with the higher level latent variables (further from the data) able to be more abstract, as they don't have to encode the lower level details encoded in the previous

(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 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.

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). For example in convolutional neural networks (used extensively in vision), visualisation of the feature maps learned in image classification tasks have shown they are hierarchical. The lower layers have high spatial resolution for detecting low-level features like edges and the higher layers have lower spatial resolution but can capture more abstract and complex features (like people) [17].

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 MSE loss (2.6) can be achieved by g learning to copy $z^{(l)}$ 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 $\hat{x}$ and 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 MSE (2.6) 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:

$$\mathcal{U}(x) = \sum_l |z^{(l)} - \hat{z}^{(l)}|^2 \tag{2.14}$$

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 (2.7) 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 [26].

While the illustration of the model (Fig. 2.8) makes it look like it has three networks, the first and third column are both the encoder, with the first being noisy and the second being clean. A final overview of the training process is then:

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.

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 [16]. 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** [13]. 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** [24].

- **A semi-supervised variational autoencoder**. [13] Also referred to as the Kingma M2 model.

- **A ladder network** [19]

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:

- Lecture slides for *Machine learning and Bayesian inference* by Holden [10]

- *Deep learning* by Goodfellow et al. [8]

- *Auto-encoding variational Bayes* by Kingma et al. [12]

- *Semi-supervised learning with deep generative models* by Kingma et al. [13]

- *Stacked denoising autoencoders: learning useful representations in a deep network with a local denoising criterion* by Vincent et al. [24]

- *From neural PCA to deep unsupervised learning* by Valpola [23]

- *Semi-supervised learning with ladder networks* by Rasmus et al. [19]

2.7 Resources

2.7.1 Language and libraries

I made the decision to use Python, as it has support for two of the most popular and intuitive deep learning libraries, PyTorch and Tensorflow. Both frameworks are computationally similar, using a tape-based automatic differentiation system for backpropagation, and a C/C++ engine for speed. After reading through an introduction to both languages, 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.

2.7.2 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.

2.7.3   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.

2.8   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

3.1 Model and class structure

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 [8] 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. [15] 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 is the layer size in the SDAE, and the number of classes in the ladder network.

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 [11]. 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

3.3.1   Datasets

During the course of this model I ran the models on three different 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 $\log_2(\text{TPM} + 1)$ [1]
Number of classes	33 - different cancer types
Balanced	No

Table 3.2: MNIST dataset

The statistics in the table are for the entire dataset. However, some of the samples are missing expression values for certain genes, and for the testing of the semi-supervised models I discard these samples to use only those with complete data. Data imputation is still an important topic in genomics, and I talk about this in Section .

METABRIC

The METABRIC dataset is a dataset of the gene expression data of breast cancer samples. The task with this data was to classify whether a tumour was to predict whether a patient was ER+ or ER-. ER+ cancers have cells containing receptor proteins that bind to estrogen, and so can be treated using substances that block the binding of estrogen, whereas ER- cancers must be treated in a different way.

Samples	1980
Inputs	1000 genes
Number of classes	2 - whether the cancer is ER+ or ER-
Balanced	No

Table 3.3: MNIST dataset

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

---

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

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 dividing by the standard deviation.

- **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.

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 [19], and similar to the transform used in the semi-supervised VAE paper [13] (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 the evaluation section there is a brief comparison of the performance of dropping the genes, replacing the missing values with the feature mean, and replacing the missing values with zero.

However, to remove any effect of the imputation on the semi-supervised comparisons I removed the samples containing missing genes, leaving 9??? samples.

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 seperate from the dataset the algorithm is trained on, to detect overfitting. 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.
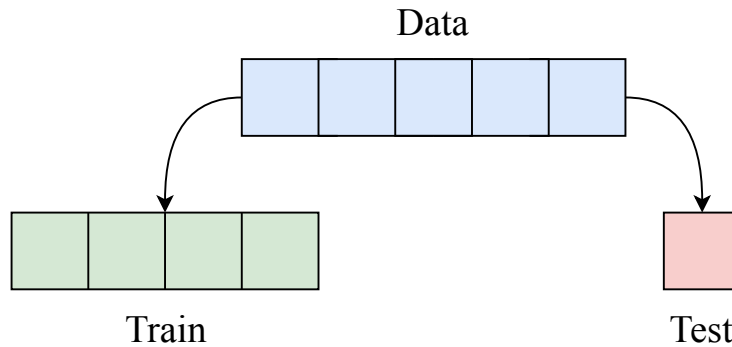
Data

Train   Test

Figure 3.1: 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 [14] 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 my test dataset in two, 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.
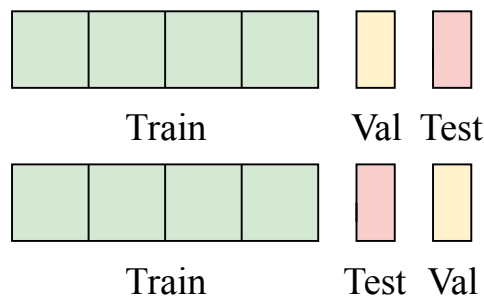
Figure 3.2: Splitting test set into a test and val set for hyperparameter optimisation

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.

While the cross-validation changes made above reduce the number of iterations to

3.4 Pipeline

Chapter 4

Evaluation

4.1 MNIST results

4.2 TCGA results

4.2.1 Missing data

4.2.2 Hyperparameters

4.3 METABRIC results

4.4 Saliency

Chapter 5

Conclusion

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] John S. Bridle. "Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition". In: *Neurocomputing*. Springer Berlin Heidelberg, 1990, pp. 227–236. ISBN: 978-3-642-76153-9.

[3] 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/`.

[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] 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.

[7] 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`.

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

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

[10] 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`.

[11]   Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimiza-
         tion". 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.

[12]   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.

[13]   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.

[14]   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.

[15]   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.

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

[17]   C. Ma et al. "Hierarchical Convolutional Features for Visual Tracking". In: *2015
         IEEE International Conference on Computer Vision (ICCV)*. Dec. 2015, pp. 3074–
         3082. DOI: 10.1109/ICCV.2015.352. URL: https://ieeexplore.ieee.org/
         document/7410709.

[18]   Vinod Nair and Geoffrey E. Hinton. "Rectified Linear Units Improve Restricted
         Boltzmann Machines". In: *Proceedings of the 27th International Conference on In-
         ternational 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.

[19]   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.

[20]   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.

[21]   *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.

[22] 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`.

[23] 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`.

[24] 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`.

[25] 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.

[26] 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* [9]. 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:

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

$\boldsymbol{\epsilon}$ 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*