Modern Machine Learning Algorithms: Applications in Nuclear Physics

by

Robert Solli

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

October 3, 2019

Contents

List of Figures

Todo list

Acknowledgments

Who to thank
Reading: Tommy, Sigmund, Geir, Øyvind, Maiken
Academic: Morten, Michelle, Daniel
Sanity: Maiken

Abstract

In this thesis, we introduce the application of convolutional autoencoder neural networks to two-dimensional projections of particle tracks from the ^{46}Ar resonant proton scattering experiment recorded by an active target time-projection chamber (AT-TPC). We also build on recent works applying pre-trained models from the image analysis community to this data.

Machine learning presents an interesting avenue for research as traditional methods of analysis of AT-TPC data are both computationally expensive and fits all particle tracks against the event type of interest - the latter presents a challenge when the space of reactions is not known prior to the analysis.

We explore the performance of the autoencoder networks and a pre-trained VGG16 model on two tasks: a semi-supervised classification task and the unsupervised clustering of particle tracks. On the semi-supervised task, we find that a logistic regression classifier trained on the latent space of these models performs very well on simulated data, with a $f1 > 0.9$. The VGG16 latent achieves this result with as few as $N = 100$ samples, as does the convolutional autoencoder when trained on the VGG16 representations of the particle tracks. Furthermore, we found that the autoencoder model reduces the variability in the identification of proton events by 64% from the estimate made by a logistic regression classifier trained on the VGG16 latent space on real experimental data.

On the clustering task, we found that a K-means algorithm applied to the simulated data in the VGG16 latent space forms almost perfect clusters, with an $ARI > 0.8$. Additionally, the VGG16+K-means approach finds high purity clusters of proton events for real experimental data. We also explore the application of neural networks to clustering by implementing the mixture of autoencoders algorithm. With this model we improved clustering performance on the real experimental data from $ARI = 0.17$ to $ARI = 0.40$. However, the neural network clustering suffers from severe stability issues prohibiting its application to new experiments.

Chapter 1

Introduction

In this thesis, we address a long-standing challenge in modern nuclear physics. The efficiency needed from detectors used in the analysis of rare nuclides means that much of the collected data is not pertinent to the questions we ask. One way of filtering out these unwanted events is to use machine learning algorithms. Machine learning is a field of study with elements from mathematics and computer science that deals with pattern recognition and function approximation. In this thesis, we explore the application of machine learning to the segmentation of events from a rare isotope detector.

Nuclear physics is the pursuit of understanding nuclides, which are the building blocks of the visible universe. These building blocks are made up of protons and neutrons. However, while most matter we interact with is stable, the vast majority of the hitherto discovered nuclides are not. Understanding these unstable nuclides are essential to our understanding of the universe, as well as having implications in medicine and the industry. Detecting these rare nuclides require very sensitive and specialized equipment. One such piece of equipment is the active target time-projection chamber (AT-TPC) detector. When running, it can detect on the order of 10^4 events each hour, producing terabytes of data each day it is active. In this thesis, we work with data from an AT-TPC detector constructed and commissioned at the National Cyclotron Laboratory located on the Michigan State University campus.

The two main topics we address in this thesis is how many events from known reactions do we need to construct a good model, and given the absence of known reactions can we segment events by reaction type.

1.1 Machine learning

In modern science, data analysis has become ubiquitous. For many applications, it is now possible to collect data with samples numbering from several thousand to billions, which has transformed the modeling needs in those sciences. Machine

learning is closely tied to this development. Machine learning models tend to have a large number of degrees of freedom, which means they need large volumes of data to perform well. These models have for example been used to beat human professionals in chess, and go[1] from just knowing the rules of the game [1]. They also fundamentally affect our on-line interactions[2]. However, machine learning models also include simpler methods, like ordinary least squares regression.

At the heart of modern machine learning is the Artificial Neural Network family of algorithms. These models, which are based on systems of biological neurons, have shown themselves to be very good at approximating complex functions [2]. They are also fundamentally flexible algorithms and can are applied to image analysis, time-series prediction, or regression tasks with relative ease.

In this thesis, we will consider a particular property of Neural Networks. We know that they can achieve a high degree of compression of complex input and that these compressions can inform us of salient differences in the input space. We consider the space of nuclear reactions in an experiment in this thesis and apply methods that can discover structure without being explicitly informed of what to look for.

1.2 Why machine learning?

Traditionally, data from experiments recorded with an AT-TPC is analyzed with a Monte Carlo method. In this framework, each event is treated as a potential candidate for the reaction of interest. Subsequently, several physical parameters are tested to find the best fit for that event. Once all events in the dataset are fit, a threshold value for the fit-statistic is chosen. Events that are below this threshold value are then said to be events of interest, and the rest are discarded. Some of the events of interest are bound to be discarded with this method, which is problematic when the interesting reactions are rare.

This method of analysis turns out to be very computationally costly. Additionally, the presumption that each event is considered a possible candidate for the reaction of interest can be problematic. In the case where the breadth of possible reactions is not known, fitting against the reaction of interest can give unexpected results. Lastly, the fitting method requires that the records are of complete tracks, but this is not always the case.

In this thesis, we then propose computationally feasible models, that do not explicitly fit against the reaction of interest, and who are agnostic to the completeness of the tracks, from the machine learning literature.

[1] Go is a strategy game played on a grid, and is like chess in that it rewards long-term planning.

[2] Facebook invests heavily in machine learning research, and apply insights from this research to dictate what we view and interact with on their platforms.

1.3 Ethical considerations

Reproducibility is a cornerstone of the scientific process, and in computational science version control systems like Github make it possible to track development. Additionally, version control provides opportunities for other researchers to reproduce results easily. To further this standard of research we make code from this thesis available for all to access from our repository.

In addition to being applied in quantum and statistical mechanics, and the design of robots with robust movement abilities[3], machine learning has also been applied to the detection of sexual orientation from facial images [3]. The latter gives rise to serious ethical concerns as LGBTQ persons are still persecuted in many nations, algorithms like these then have the potential to influence these persons livelihood, as well as their mental and physical health. The algorithms developed for this thesis are latent variable models for the segmentation of data, and it is easy to transfer these algorithms to different arenas e.g., the segmentation of people in discrete groups.

It is difficult to say what moral obligations the researcher carries in this instance. However, this is not unfamiliar territory, as the development of nuclear weaponry is closely tied with the emergence of modern nuclear theory. From this history, we can infer that the openness of the scientific discourse is a necessary ingredient in providing strategies for dealing with complex issues such as these. That is not to say that this discourse is sufficient. Being critical of our role as researchers in the development of these algorithms is essential Additionally, we must engage with society and lawmakers on these issues, as part of the responsibility of being a scientist.

1.4 Structure of the thesis

As machine learning is still nascent in its application to nuclear physics, we begin with a thorough introduction to the theory in the first chapter of part I. We continue in the second chapter with an introduction to the models implemented and applied in this thesis. The last chapter of part I is devoted to elaborating on the experiment, and data, that forms the basis for our analysis.

Part II is devoted to the details of the implementation. We have chosen the Python programming language as the basis for the implementation because of the mature machine learning libraries developed for the language.

Lastly, in parts III and IV, we present our findings and discuss those. We also present avenues for further research based on our findings.

[3]Boston dynamics design and manufacture robots with exceptional adaptability to varying environments

Part I

Theory and Experimental Background

Chapter 2

Fundamental Machine Learning Concepts

2.1 Introduction

In this thesis, we explore the application of advanced machine learning methods to experimental nuclear physics data. To properly understand the framework of optimization, validation, and challenges we face, we will introduce these using two models: linear and logistic regression. Before those discussions, we briefly outline the process of model fitting and introduce the difference in models where there is a known versus an unknown outcome.

Fitting models to data is the formal framework by which much of modern science is underpinned. In most scientific research, the researcher needs to formulate some model that represents a given theory. In physics, we construct models to describe complex natural phenomena which we use to make predictions or infer inherent properties about the natural world. These models vary from estimating the Hamiltonian of a simple binary spin system like the Ising model, to more sophisticated methods like variational Markov chain Monte Carlo models, which are used in many-body quantum mechanics.

We view this process as approximating an unknown function $\hat{f}$ which takes a state $\mathbf{X}$ as input and gives some output $\hat{\mathbf{y}}$,

$$\hat{f}(\mathbf{X}) = \hat{\mathbf{y}}. \tag{2.1}$$

To approximate this function we use an instance of a model: $f(\mathbf{X}; \theta) = \mathbf{y}$, where we don't necessarily have a good ansatz for the form of f or the parameters θ. The model can take on different forms depending on the purpose, but the parameters θ can be thought of as a set of matrices that transform the input into the output. Additionally, the output of the function can be multi-variate, discrete, or continuous, which informs the choice of model for a particular problem. In this

first part of this chapter, we consider a single real-valued outcome. Additionally we note that in this thesis we use a notation on the form $f(y|x;\theta)$ which reads as the function f with output y given x and the parameters θ, and is equivalent to the notation $y = f(x;\theta)$. If f is a probability density the relationship above reads as the probability of y given x and parameters θ, the former is a more common notation for probabilities and the latter more common for continuous real-valued outcomes.

The theory we present in this thesis is built on the understanding of expectation values, and how how they behave. Here we define some key properties of expectation values in general. Let $p(x)$ be a normalized probability density function, i.e.

$$1 = \int_{-\infty}^{\infty} p(x)dx. \tag{2.2}$$

The expectation of a function, f, of x is then defined as

$$\langle f(x)\rangle_p := \int_{-\infty}^{\infty} f(x)p(x)dx. \tag{2.3}$$

Some particular expectations are notable because of their ubiquitousness. Here we concern ourselves primarily with the mean and variance of a distribution. These expectations are also known as the first and second moment of a distribution and are defined as

$$\mu := \langle x\rangle_p, \tag{2.4}$$
$$\sigma^2 := \langle x^2\rangle_p - \langle x\rangle_p^2. \tag{2.5}$$

Returning to the question of approximating $\hat{f}$ we begin with the objective of the model: to minimize the discrepancy, $g(|\hat{y} - y|)$, between our approximation and the true target values. An example of the function g is the mean squared error function used in many modeling applications, notably in linear regression which we explore in section 2.2.

In this paradigm, we have access to the outcomes of our process, $\hat{y}$, and the states, $\mathbf{X}$. In machine learning parlance, this is known as supervised learning.

However, this thesis deals mainly with the problem of modeling when one only has access to the system states. These modeling tasks are known as unsupervised learning tasks. As the models are often similar in supervised and unsupervised learning the concepts, terminology, and challenges inherent to the former are also ones we have to be mindful of in the latter.

Approximating functions with access to process outcomes starts with the separation of our data into two sets with zero intersection. This is done so that we can estimate the performance of our model in the real world. To elaborate on

the need for this separation, we explore the concepts of over-fitting and under-fitting to data later in this chapter.

2.2 Linear Regression

Modern machine learning has part of its foundations from the familiar linear regression framework. With its popularity, linear regression also has a multitude of solution strategies. The most straight forward of which is with some simple linear algebra and calculus.

We begin by defining the constituents of our model: let the data be denoted as a matrix $\mathbf{X} \in \mathbb{R}^{m \times n+1}$, where m is the number of data-points and n the number of features. We add the $+1$ factor to describe the addition of a column of ones to our data as a convenient placeholder for the model intercept. Note also that the term features are used broadly in machine learning literature and denote the measurable aspects of our system; in the 1D Ising model, the n would denote the number of spins and m the number of measurements we made on that system. Furthermore let the parameter, or *weight*, matrix be given as $\mathbf{w} \in \mathbb{R}^{n+1 \times k}$. Generally, the outcome-dimension k can be greater than one when estimating multi-variate outcomes. We limit this section and the following to the case where $k = 1$ for illustrative purposes. The linear regression model is then the transformation of the input using the weight matrix, i.e.

$$\mathbf{y} = \mathbf{X}\mathbf{w}. \tag{2.6}$$

Finally, let the outcome we wish to approximate be given as a vector $\hat{\mathbf{y}} \in \mathbb{R}^m$. We will not concern ourselves overly with the properties of the process that generates $\hat{\mathbf{y}}$ in this section as this is elaborated on in the following sections, and assume that that the outcome has no noise.

The challenge is then finding the weights that give correct predictions from data. To measure the quality of our model, we introduce the squared distance between our predictions and $\hat{\mathbf{y}}$, which also gives us a path to optimization. By differentiating with respect to the model parameters, we can find the optimal solution for the problem. The squared error is defined in terms of the Euclidean vector norm

$$L_2(\mathbf{x}) = ||\mathbf{x}||_2 = \left(\sum x_i^2\right)^{\frac{1}{2}},$$

In mathematical terms we define the squared error objective as

$$\mathcal{O} = ||\hat{\mathbf{y}} - \mathbf{y}||_2^2. \tag{2.7}$$

An objective function $\mathcal{O}$ defines some optimization problem to minimize or maximize. In machine learning, these are most commonly cast as minimization prob-

lems. Objective functions for finding minima are termed cost functions in machine learning literature, and we will adopt that nomenclature moving forward. In this thesis, we use the symbol $\mathcal{C}$ to indicate such functions, and the optimization problem is then finding the minimum w.r.t the parameters θ^*, i.e.

$$\theta^* = \arg\min_{\theta} \mathcal{C}(\hat{\mathbf{y}}, f(\mathbf{X}; \theta)). \tag{2.8}$$

We use the starred notation to indicate the optimal parameters for the given cost function.

Returning to the least squares problem the task is finding the optimal parameters now requires a differentiation, and to aid in that we write the objective as a matrix inner product

$$\begin{aligned}\mathcal{C} &= ||\hat{\mathbf{y}} - \mathbf{y}||_2^2, \\ \mathcal{C} &= (\hat{\mathbf{y}} - \mathbf{X}\mathbf{w})^T(\hat{\mathbf{y}} - \mathbf{X}\mathbf{w}).\end{aligned}$$

Since the problem is one of minimization we take the derivative with respect to the model parameters and locate its minima by setting it to zero

$$\begin{aligned}\nabla_{\mathbf{w}}\mathcal{C} &= \nabla_{\mathbf{w}}(\hat{\mathbf{y}} - \mathbf{X}\mathbf{w})^T(\hat{\mathbf{y}} - \mathbf{X}\mathbf{w}), && (2.9)\\ &= -2\mathbf{X}^T\hat{\mathbf{y}} + 2\mathbf{X}^T\mathbf{X}\mathbf{w}, && (2.10)\\ \mathbf{0} &= -2\mathbf{X}^T\hat{\mathbf{y}} + 2\mathbf{X}^T\mathbf{X}\mathbf{w}, && (2.11)\\ \mathbf{X}^T\hat{\mathbf{y}} &= \mathbf{X}^T\mathbf{X}\mathbf{w}, && (2.12)\\ \mathbf{w} &= (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\hat{\mathbf{y}}. && (2.13)\end{aligned}$$

This problem is analytically solvable with a plethora of tools. Most notably, we have the ones that do not perform the matrix inversion $(\mathbf{X}^T\mathbf{X})^{-1}$ as this inverse is not unique for data that does not have full rank.

Admittedly, the least-squares linear regression model is relatively simple and is rarely applicable to complex systems. Additionally, we have not discussed the assumptions made to ensure the validity of the model. Most important of which concerns the measurement errors, which we assume to be identical independent normally distributed.

Given the relative simplicity of the linear regression model and the fact that an analytical solution can be found, we will use it for reference in understanding the subsequent sections in this chapter.

2.3 Over and under-fitting

When fitting an unknown function to data, it is often not clear what complexity is suitable for the model. Additionally compounding this problem is the ever-present threat of various noise signals and measurement errors present in the data. Further complicating the issue is the nature of machine learning problems: we are almost always interested in extrapolating to unseen regions of data.

In the previous section on linear regression, we operated on the premise that the outcome $\hat{y}$ we wish to model is a perfect noiseless record of nature, which does not translate. The task we're faced with is then to determine an appropriate complexity for the model which lets us extrapolate to unseen regions, and that fits the signal and not the noise in the outcome.

To begin the discussion on more realistic systems we start by re-defining the outcome we wish to model, $\hat{y}$, as a decomposition of the true unknowable process P which acts as a function of the system state $\mathbf{x}$ and a stochastic noise term ϵ,

$$\hat{y}_i = P(\mathbf{x}_i) + \epsilon_i. \tag{2.14}$$

It is now necessary to introduce some more terminology to tackle the problems introduced by noisy data. Firstly we define the terms over and under-fitting. A model is said to be over-fit if it is excessively complex, and thus when fit models the noise strongly. Over-fit models will tend to perform very well during fitting but will rapidly deteriorate outside the domain it was trained on. Under-fit models are models that do not have enough expressive power to capture the variations in the data. With modern computing resources, it is much easier to make a model too complex than not complex enough. Frankle et al. [4] and Frankle and Carbin [5] show that, in fact, most complex models could be fully expressed using just parts of the original model. As a consequence of this, we focus primarily on the effect of, and how to avoid, over-fitting.

Secondly, we need to establish a framework for evaluating if a model is over or under-fit. A robust and straightforward method of doing this is creating a disjoint hold-out set of the data which is not used during the estimation of the model parameters. In machine learning vernacular, these are sets of data we call testing-set and the data used to fit the model is called the training-set.

To illustrate the problems created by noise in the data we'll consider a one-dimensional polynomial regression problem[1]. Two sets of outcomes were generated from a process P as in equation 2.14 using linear and cubic polynomials with an added noise-term. We attempt to model these processes with polynomials of a few select degrees n, which we fit using a least-squares approximation. The fitting was performed using the python package `numpy` [7]. Additionally, we split the sets into disjoint subsets for training and testing. The data and the models

[1] We note that this section follows closely that of section 2 in Mehta et al. [6], we also refer to this paper for a more in-depth introduction to machine learning for physicists.

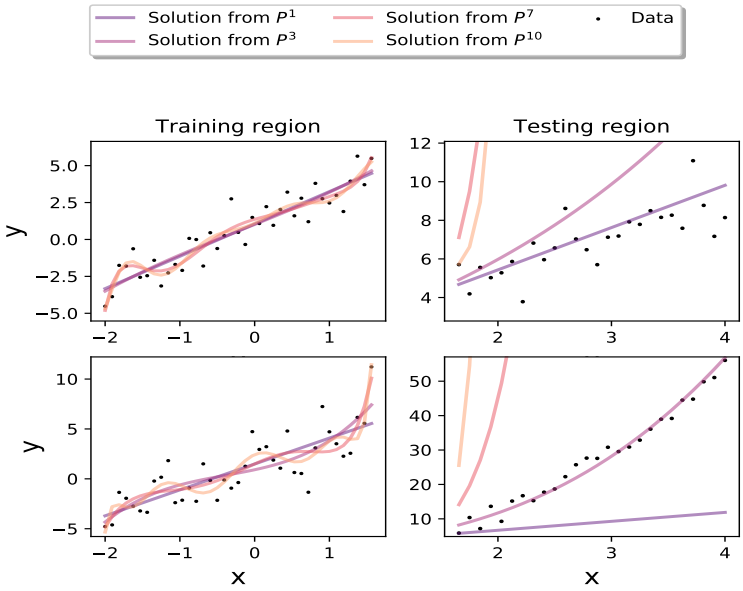


Figure 2.1: Polynomial regression of varying degrees on data drawn from a linear distribution on above and a cubic distribution on the bottom. Models of varying complexity indicated by their basis P^n are fit to the train data and evaluated on the test region, shown in the left and right columns. We observe that the higher-order solutions follow what we observe to be spurious-noise generated features in the data. This is what we call over-fitting. In the bottom row we observe that the model with appropriate complexity, $f(x_i) \in P^3$, follows the true trend also in the test region while the linear and higher-order models all miss. The linear model cannot express the complexities of the data and is said to be under-fit. Additionally, we observe that the higher-order polynomials degrade in performance extremely rapidly outside the training region.

are shown in figure 2.1. In the figure, we observe that the higher-order polynomials follow spurious trends in the data generated by the noise factor. The higher expressibility of the model leads to capturing features of the noise that increases performance in the training domain that rapidly deteriorates in the testing region. In the top column, the linear model outperforms the more complex models in the testing region. Conversely, when we increase the complexity of the true process to be data that is drawn from a polynomial in P^3 the linear model loses the ability to capture the complexities of the data and is said to be under-fit.

The previous paragraphs contain some important features that we need to keep in mind going forward. We summarize them here for clarity:

- "Fitting is not predicting" [6]. There is a fundamental difference between fitting a model to data and making predictions from unseen samples.

- Generalization is hard. Making predictions in regions of data not seen during training is very difficult, making the importance of sampling from the entire space during training that much more vital.

- Complex models often lead to overfitting. While usually resulting in better results during training in the cases where data is noisy or scarce, predictions are poor outside the training sample.

2.4 The bias-variance relationship

Understanding over and under-fitting is very important to understanding the challenges faced when doing machine learning. As it turns out those concepts are fundamentally tied to the out-of-sample error, E_{out}, for which the mean squared error (MSE) cost function can be decomposed in three contributions[2], namely

$$E_{out} = \langle C(S, f(\mathbf{x}; \theta_{S_t})) \rangle_{S,\epsilon} = \text{Bias}^2 + \text{Variance} + \text{Noise}. \tag{2.15}$$

This expectation is rather compact and so before we move on to explaining the bias and variance start by elaborating on S and ϵ. Recall from equation 2.14 that we decompose the target values as a contribution from a true function $\hat{f}$, and an error term ϵ

$$\hat{y}_i = \hat{f}(\mathbf{x}_i) + \epsilon. \tag{2.16}$$

In this section, we assume that the noise is distributed as $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

The expectation in equation 2.15 is taken over a model with optimized parameters θ_{S_t}, whose value are a function of the selected data used in the optimization, $S_t = \{(\hat{y}_i, \mathbf{x}_i)\}$. We can then view our model, $f(\mathbf{x}; \theta_{S_t})$, as a stochastic functional which varies over the selected data used for training. The expectation over S is then an expectation over the set of different choices of training data.

With the derived quantities from appendix B, and equation 2.14, we can then define the bias as

$$\text{Bias}^2 = \sum_i (\hat{\mathbf{y}}_i - \langle f(\mathbf{x}_i; \theta_{S_t}) \rangle_S)^2. \tag{2.17}$$

The bias term is analogous to the squared error cost and gives and estimate to the degree of under-fitting the model is susceptible to. Building on this we have the variance term

$$\text{Variance} = \sum_i \langle (f(\mathbf{x}_i; \theta_{S_t}) - \langle f(\mathbf{x}_i; \theta_{S_t}) \rangle_S) \rangle_S. \tag{2.18}$$

[2]We show this derivation in appendix B

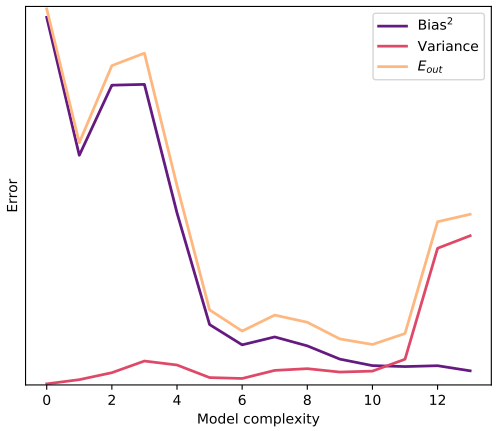


Figure 2.2: Decomposition of the bias and variance from the overall test-set error E_{out}. A set of polynomials of varying degrees are fit to a known function using randomly selected training samples. The polynomial degree is denoted by the x-axis label. With the set of polynomials we evaluate the bias and variance terms shown in equation 2.17 and 2.18. In the figure we observe the characteristic relationship where the out of sample error decreases steadily with complexity until the models start to over-fit as measured by the variance, and the E_{out} increases as a consequence.

For clarity we note that the summation is over the testing set elements. The variance term relates to the consistency in the testing region, and as such describes the degree of over-fitting the model is exhibiting.

The bias-variance relationship describes a fundamental challenge in most domains of machine learning; fitting a more complex model requires more data. Which has led to an explosion in the acquisition of vast amounts of data in the private sector. With the recent development in nuclear physics where data is becoming abundant this opens the avenue to exploring more complex models than has previously been possible. We illustrate this tension in figure 2.2 where polynomials of varying degrees are fit to a linear combination of exponential functions. We observe the characteristic relationship between bias and variance wherein the out of sample error E_{out} starts out decreasing with complexity, but increases exponentially as a threshold is reached. Note also that we illustrate the concept of the irreducible error that for a model class, i.e. polynomials, there is an irreducible error term owing to the contribution from the noise.

2.5 Regularization

With the advent of modern computing resources, researchers gained the ability to operate very complex models. This gave rise to the problem of over-fitting. Consequentially while performance on the data the model is fit to increases, it rapidly deteriorates outside that region. As much of current research deals with somehow bridging the barriers between different regions of data, or entirely different distributions, reducing the chance of a model over-fitting is crucial in most applications. In this thesis, we tackle data that come from multiple different instances of the same nuclear-experiment, which means training on one instance should translate to another. Additionally, it is hugely beneficial if the recognition of reactions translates between different experiments.

Finding measures to reduce over-fitting has been a goal for machine learning researchers for near on 50 years. The first modern breakthrough was adding a constraint on the cumulative magnitude of the coefficients to linear regression systems. This form of restriction proved hugely beneficial for the simple reason that it restricted the model's ability to express all of its complexity. Introduced in 1970 by Hoerl and Kennard [8] the addition of a L_2 norm-constraint to linear regression was dubbed *ridge* regression. Experiments with different norms were carried out in the years following the elegant discovery by [8]. Perhaps most important of them is the use of the L_1-norm, first successfully implemented by Tibshirani [9]. As the norms have different geometric expressions, the consequence of their addition to the cost was evident in the types of solutions generated by their inclusion. We illustrate this geometry in figure 2.3, where the lasso penalty is shown to result in a constrained region for the parameter inside a region with vertices pointing along the feature axes. Intuitively this indicates that for a L_1 penalty, the optimal solution is a sparse one whereas many parameters as possible are zero while still minimizing cost. For L_2 ridge regression these vertices are not present, and the region has an even boundary along the feature axes resulting in solutions where most parameter values are small. The figure is made using the understanding that adding a regularization term is equivalent to solving a constrained optimization problem, for example in the case of least squares regression

$$\theta^* = \underset{||\theta||_2^2 < t}{\arg\min} ||y_i - f(\mathbf{x}_i; \theta)||_2^2. \tag{2.19}$$

The inclusion of an L_1-norm to the linear regression cost-function proved to be challenging as had no closed-form solution and thus required iterative methods like gradient descent, described in detail in section 2.10.

We still have to show how these additional contributions add to the cost-function. We begin by defining the general L_p norm of a vector $\mathbf{x} \in \mathbb{R}^n$ as

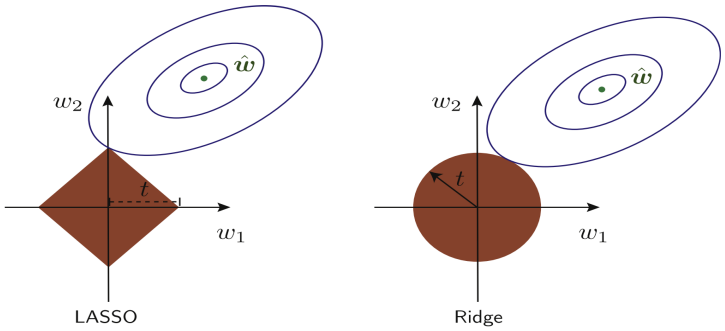


Figure 2.3: Demonstrating the effect of a regularization constraint on a 2-variable optimization. The blue ovals represent the squared error, as it is quadratic in the parameters w_i. Moreover, the shaded brown region represents the restriction on the values of w_i, s.t. the only eligible values for the parameters are inside this region. Since the L_1 norm has these vertices on the feature axis, we expect that the contour of the cost will touch a vertex. Consequently generating a sparse feature representation. The L_2 norm does not have these protrusions and will then generally intersect with the cost-contour somewhere that generates a linear combination of features that all have small coefficients. Figure copied from Mehta et al. [6], which in turn is adapted from a figure in Friedman et al. (2001)

$$L_p(\mathbf{x}) = \left(\sum |x_i|^p\right)^{\frac{1}{p}}. \tag{2.20}$$

A common notation for the $L_p(\cdot)$ norm that we will also use in this thesis is $L_p(\cdot) = ||\cdot||_p$. We note that the familiar euclidian distance is just the L_2 norm of a vector difference. While the L_1 term is commonly called the Manhattan or taxicab-distance, aptly named as one can think of it as the distance a cab-driver would drive from one house to another.

Modifying the cost function then is as simple as adding the normed coefficients. To demonstrate we add a ridge regularization term to the squared error cost with λ determining the strength of the regularization, while the rest of the symbols have their usual meaning. The modified cost function is then

$$\mathcal{C}(\hat{y}_i, f(\mathbf{x}_i;\theta)) = (\hat{y}_i - f(\mathbf{x}_i;\theta))^2 + \lambda \sum ||\theta_i||_2^2. \tag{2.21}$$

Assuming now that the model f is a linear regression model we can derive the solution by substituting in the model from section 2.2 and repeating the procedure of taking the derivative with respect to the parameters. We begin by substituting in and taking the derivative

$$\mathcal{C}(\hat{\mathbf{y}}, \mathbf{X}\mathbf{w}) = (\hat{\mathbf{y}} - \mathbf{X}\mathbf{w})^T(\hat{\mathbf{y}} - \mathbf{X}^T\mathbf{w}) - \lambda\mathbf{w}^T\mathbf{w}, \tag{2.22}$$

$$\nabla_{\mathbf{w}} C = -2\mathbf{X}^T\hat{\mathbf{y}} + 2\mathbf{X}^T\mathbf{X}\mathbf{w} - 2\lambda\mathbf{w}. \tag{2.23}$$

Following from the section on linear regression to find the optimal parameters we find the zero intersection of the derivative and solve for the parameters

$$\mathbf{0} = -\mathbf{X}^T\hat{\mathbf{y}} + (\mathbf{X}^T\mathbf{X} - \lambda\mathbf{I})\mathbf{w}, \tag{2.24}$$

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} - \lambda\mathbf{I})^{-1}\mathbf{X}^T\hat{\mathbf{y}}. \tag{2.25}$$

The solution is very close to that of the ordinary least squares problem with an added term in the matrix inversion. This addition turns out to be very convenient as it ensures the resulting matrix has full rank, which avoids some of the potential problems when trying to estimate the inverse.

Conceptually the regularization term added to the cost function modifies what parameters satisfy the arg min in equation 2.8 by adding a penalty to parameters having high values. This is especially useful in cases where features are covariate, or the data is noisy. Regularization then reduces the probability of overfitting by limiting the expressed complexity of a model. In the example of polynomial regression lasso regularization forces many of the coefficients to be zero-valued in such a way that it still performs maximally. In that way, regularization is addressing the challenge posed by the balance between bias and variance presented in section 2.4 as reducing the expressibility of the model is in effect a reduction of complexity reducing variance at the cost of an increased bias.

2.6 Hyperparameters

In the previous section on regularization, we introduced the regularization strength λ without much fanfare. However, that innocuous-seeming inclusion has quite far-reaching consequences. All of which stem from the question of how do we determine the value of the parameter λ? There is no analytical solution for the optimal value, and so we are left with other ways of estimating its value. Either by educated guesswork or through some principled schema. More sophisticated models require several of these parameters. Moreover, they will often strongly affect optimization. Collectively these parameters are known as hyperparameters.

Hyperparameters are parameters in the model that have to be empirically determined, which often impact performance and who do not have a closed-form derivative in the optimization problem. These parameters have proven to be vitally important to the optimization of machine learning models. In the simple

linear or logistic regression case the hyperparameters include the choice of regularization norm (ordinarily the L_1 or L_2 norms) and the regularization strength λ. The choices of all these parameters are highly non-trivial because their relationship can be strongly co- or contra-variant. Additionally, the search for these parameters is expensive because each configuration of parameters requires re-training the model.

In this section, we will discuss the general process of tuning hyperparameters in general. Subsequently, we will introduce specific parameters that need tuning that pertain to particular architectures used in this thesis. Whichever scheme is chosen for the optimization they each follow the same basic procedure:

1. Choose hyperparameter configuration
2. Train model
3. Evaluate performance
4. Log performance and configuration

When searching for hyperparameter configurations for a given model, it becomes necessary to define a scale for the variable. Together with the range, the scale defines the interval on which we search. That is, the scale defines the interval width on the range of the parameter. Usually, the scale is either linear or logarithmic, though some exceptions exist. As we introduce hyperparameters for each model, a suggested scale will also be discussed.

2.6.1 Hand holding

The most naive way of doing hyperparameter optimization is to tune the values by observing changes in performance metrics manually. This approach is very naive and rarely used in modern modeling pipelines outside the prototyping phase. For this thesis, we use a handheld approach to get a rough understanding of the ranges of values over which to apply a well-reasoned approach.

2.6.2 Grid Search

Second on the ladder of naiveté is the exhaustive search of hyperparameter configurations. In this paradigm, one defines a multi-dimensional grid of parameters over which the model we evaluate the model. This approach has two principal pitfalls, the first is computational: If one has a set of N magnitudes of the individual parameter sets $s = \{n_i\}_{i=0}^{N}$ with values of the individual parameters γ_k and where $n_i = |\{\gamma_k\}|$ then the complexity of this search is $\mathcal{O}(\prod_{n \in s} n)$. For example we want to estimate the learning rate $\eta = \{\eta_k\}$ and the regularization strength $\lambda = \{\lambda_k\}$, then this search is a double loop as illustrated in algorithm 1.

In practice, the grid search is rarely used as the computational complexity scales exponentially with the number and resolution of the parameters. The nested nature of the for loops is also extremely inefficient if the hyperparameters are disconnected. That is, neither co- or contra-variant.

Algorithm 1: Showing a grid search hyperparameter optimization for two hyperparameters η and λ

```
Data: Arrays of float values λ, η
Result: log of performance for each training
Initialization ;
log ← [ ];
for λ_k ∈ λ do
    for η_k ∈ η do
        opt ← optimizer(η_k);
        model_instance ← model(λ_k);
        metrics ← model_instance.fit(X, ŷ, opt) ;
        log.append((metrics, (λ_k, η_k)))
    end
end
```

2.6.3 Random Search

Surprisingly, one of the hyperparameter search algorithms that proved to be among the most effective is a simple random search. Bergstra and Bengio [10] showed the inefficiency of doing grid search empirically and proposed the simple remedy of doing randomized configurations of hyperparameters. The central argument of the paper is elegantly presented in figure 2.4. Where the authors observe that a grid search is both more computationally expensive and has significant shortcomings for complex modalities in the loss functions. As such, we approach the majority of hyperparameter searches in this thesis by way of random searches. The algorithm for the random search is straightforward. It requires one to draw a configuration of hyperparameters and run a fitting procedure N times and log the result. In terms of performance, both grid and random search can be parallelized with linear speedups.

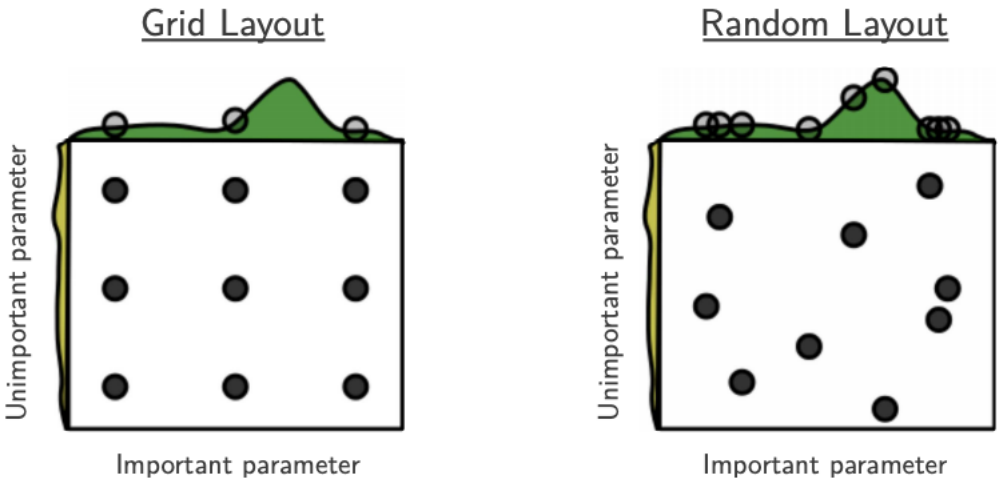


Figure 2.4: Figure showing the inefficiency of grid search. The shaded areas on the axis represent the unknown variation of the cost as a function of that axis-parameter. Since the optimum of the loss with respect to a hyperparameter might be very narrowly peaked a grid search might miss the optimum entirety. A random search is less likely to make the same error as shown by Bergstra and Bengio [10]. Figure copied from Bergstra and Bengio [10]

2.7 On information

Up until now, we have been dealing with predicting a real-valued outcome. However, this is not always the goal. A prevalent task in machine learning is predicting the odds, or probability, of some event or confidence in a classification. The term classification is a general term in machine learning literature which defines a task where the goal is to predict a discrete outcome. Examples include predicting the species of an animal, or thermodynamic state of a system. Transitioning the type of goal our model has necessitates some new terminology. In this section, we will briefly touch on some fundamental concepts in information theory needed to construct models that perform a classification task.

One of the fundamental sizes in information theory is the amount of chaos in a process. As well as how much one needs to know to characterize the same process. A process can be the toss of a coin, or roll of a dice. These concepts tie into well-known phenomena to physicists from statistical and thermal physics. As a quick refresher, we mention that more random processes possess more information in this formalism, i.e., a rolling die has more information than a spinning coin. We define the information of an event in the usual way as

$$I := -\log(p(\mathbf{x})), \tag{2.26}$$

where $p(\mathbf{x})$ is the probability of a given event $\mathbf{x}$ occurring. One of the quantities that have extensive applications is the expectation over information, known as the entropy of a system. We define the entropy as just that, the expectation over the information:

$$H(p(\mathbf{x})) := -\langle I(\mathbf{x})\rangle_{p(\mathbf{x})}. \tag{2.27}$$

Depending on the choice of the base of the logarithm, this functional has different names. However, the interpretation is mostly the same. Entropy measures the degree of randomness in the system. The base two entropy, known as the Shannon entropy, describes how many bits we need to fully describe the process underlying $p(\mathbf{x})$.

In machine learning, or indeed may other applications of modeling, we wish to encode a process with a model. We can then measure the amount of bits (or other units of information) it takes to encode the underlying process, $p(\hat{y}|\mathbf{x})$, with a model distribution $q(y|\mathbf{x};\theta)$. We re-iterate that in this thesis, we will, in general use the semi-colon notation to denote model parameters. The measure of information lost by encoding the original process with a model is called the cross-entropy and is defined as

$$H(p,q) := -\sum_{\mathbf{x}} p(\mathbf{x})\log(q(\mathbf{x};\theta)). \tag{2.28}$$

With the cross-entropy, we have arrived at a way to measure information lost by using the model q, which means we can use the cross-entropy as a tool to optimize the model parameters. We begin by simply considering a binary outcome y_i as a function of a state $\mathbf{x}_i$ and define the Maximum Likelihood Estimate (MLE) as the probability of seeing the data given our model and parameters. Let the data be a set consisting of tuples[3], $s_i = (\mathbf{x}_i, \hat{y}_i)$, and denote that set as $S = \{s_i\}$ then the likelihood of our model is defined as

$$p(S|\theta) := \prod_i q(\mathbf{x}_i;\theta)^{\hat{y}_i} - (1 - q(\mathbf{x}_i;\theta))^{1-\hat{y}_i}. \tag{2.29}$$

We want to maximize this functional with respect to the parameters θ. The product sum is problematic for the optimization, as its gradient will likely vanish with increased terms . To circumvent this, we take the logarithm of the likelihood and define the log-likelihood. Since we define the likelihood as a maximization problem, we define the negative log-likelihood as the corresponding minimization problem. Optimizing the log-likelihood yields the same optimum as for the

[3]a tuple is a data structure consisting of an ordered set of different elements. It differs from a matrix in that the constituent elements need not be of the same dimension.

likelihood as the logarithmic function is monotonic [4]

$$\mathcal{C}(\mathbf{x}, y, \theta) = -\log(p(S|\theta)) = -\sum_i y_i \log(q(\mathbf{x}_i;\theta)) + (1-y_i)(q(\mathbf{x}_i;\theta)). \quad (2.30)$$

Where we observe this is simply the cross-entropy for the binary case. The optimization problem is then

$$\theta^* = \arg\min_\theta \mathcal{C}(\mathbf{x}, \hat{y}, \theta). \quad (2.31)$$

This formulation of the MLE for binary classification can be extended to the case of linear regression where one shows the mean squared error is the functional to optimize for. The MLE is for most models not analytically solvable, and so in machine learning the solution of these optimization problems is found by gradient descent. Gradient descent is discussed in some detail in section 2.10. The first place we find that we need iterative methods is in the section immediately following this, where we discuss the second principal machine learning algorithm; logistic regression.

2.8 Logistic Regression

As mentioned previously a good portion of machine learning has the objective of identifying what class a given sample is drawn from. As a problem it has a very natural formation as classification is something we do both explicitly and implicitly every day. Visually identifying what animal the next-door neighbor is taking for a walk or when it is safe to cross the road are some classification tasks that we are very good at. In physics classification also holds significant interest. Whether it is identifying phase transitions from the state configuration of a thermodynamic system, or identifying reaction constituents from particle tracks which is the objective of this thesis.

To understand classification algorithms we begin from the simplest algorithm in classification; the perceptron [11]. The perceptron uses the same transformation as in equation 2.6 and determines the class from the sign of the prediction. This is a rather crude representation of the problem and so we seek to refine it somewhat. The challenge lies in predicting a bounded variable like a probability $p \in [0, 1]$ with a principally unbound transformation like in equation 2.6. To construct a feasible model we begin by defining the odds of an event, which is simply the ratio of probability for an event happening to the probability of it not happening, that is

$$o = \frac{p}{1-p}. \quad (2.32)$$

[4] it can be shown that for optimization purposes any monotonic function can be used, the logarithm turns out to be practical for handling the product sum and exponents.

Since p is bounded in the unit interval unfortunately the odds are bounded in $\mathbb{R}^+$. Again the logarithm comes to the rescue as the logarithm of a positively bounded variable is unbounded, and so we define the log-odds as the output of our model given a data-point $\mathbf{x}_i$ and the model parameters $\mathbf{w}$

$$\log\left(\frac{p_i}{1-p_i}\right) = \mathbf{x}_i\mathbf{w}. \tag{2.33}$$

As a reminder we note that we add a column of ones to the data and thus include the intercept in the weights.

We can then transform the log-odds back to a model for the probability which gives us the formulation for logistic regression

$$q(y_i = 1|\mathbf{x}_i; \mathbf{w}) = q(\mathbf{x}_i\mathbf{w}) = \frac{1}{1+e^{-\mathbf{x}_i\mathbf{w}}}. \tag{2.34}$$

The term on the right-hand side is the logistic sigmoid function, which has the notable properties one needs from a function that models probabilities e.g. $f(x) = 1 - f(1-x)$. With a binary outcome we can plug this model directly into the MLE cost defined in equation 2.30, i.e.

$$P(S|\mathbf{w}) = -\sum_i y_i \log q(\mathbf{x}_i\mathbf{w}) + (1-y_i)\log\left(1 - q(\mathbf{x}_i\mathbf{w})\right). \tag{2.35}$$

Finding the optimal values for the parameters $\mathbf{w}$ is again a matter of finding the minimum of the cost. Noting first that the derivative of the logistic sigmoid is

$$\nabla_{\mathbf{w}} q(\mathbf{x}_i\mathbf{w}) = q(\mathbf{x}_i\mathbf{w})(1 - q(\mathbf{x}_i\mathbf{w}))\mathbf{x}_i, \tag{2.36}$$

which is known as the sigmoid derivative identity in the machine learning literature. Additionally the derivative of the log model is straightforwardly computed as

$$\nabla_{\mathbf{w}} \log q(\mathbf{x}_i\mathbf{w}) = 1 - q(\mathbf{x}_i\mathbf{w})\mathbf{x}_i. \tag{2.37}$$

We can then write out the derivative of the cost as

$$\nabla_{\mathbf{w}}\mathcal{C} = -\sum_i y_i(1 - q(\mathbf{x}_i\mathbf{w}))1 - y_i(-)q(\mathbf{x}_i\mathbf{w})\mathbf{x}_i, \tag{2.38}$$

$$= -\sum_i \left(y_i - q(\mathbf{x}_i\mathbf{w})\right)\mathbf{x}_i. \tag{2.39}$$

Unfortunately this derivative is transcendental in $\mathbf{w}$ which means that there is no closed form solution w.r.t. the parameters. Finding the optimal values for the parameters is then a problem that has to be solved with iterative methods. The

same methods are used to fit very complex machine learning methods and as such proper understanding of these underpin the understanding of complex machine learning methods as much as understanding linear and logistic-regression. We discuss gradient descent in some detail in section 2.10, but first we re-visit linear regression with the MLE formalism fresh in mind.

2.9 Revisiting linear regression

We've seen the applications of the maximum likelihood estimate to classification in section 2.8, but the same formalism can very easily be applied to regression. In this section we'll detail the derivation of linear regression solution in the formalism of a maximum likelihood estimate, as it is in this formalism the thesis writ large is framed. Re-introducing linear regression we define the model on a general form as the linear relationship expressed in equation 2.40. The basis of $\mathbf{w}$ is left unspecified, but we are free to model using polynomial, sinusoidal or ordinary Cartesian basis-sets. Using the terminology introduced earlier in this chapter our model is then,

$$y_i = \mathbf{x}_i \mathbf{w}. \tag{2.40}$$

In addition to equation 2.40 we introduce the error term $\epsilon_i = y_i - \hat{y}_i$ which is the difference between the models prediction, y_i, and the actual value $\hat{y}_i$. The goal of linear regression is to minimize this error, in mathematical terms we have an optimization problem on the form

$$\mathcal{O} = \arg\min_{\mathbf{w}} ||\hat{\mathbf{y}} - \mathbf{X}\mathbf{w}||^2. \tag{2.41}$$

The central assumption of linear regression, that provides the opportunity for a closed form solution, is the assumption of Independent Identically Distributed (IID) ϵ_i's. We assume that the error is normally distributed with zero-mean and identical variance, σ^2,across all samples, e.g.

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2), \tag{2.42}$$

and similarly we consider the model predictions to be normally distributed, but with zero variance, e.g.

$$\hat{y}_i \sim \mathcal{N}(\mathbf{x}_i \mathbf{w}, 0). \tag{2.43}$$

We use $\mathcal{N}(\mu, \sigma^2)$ to denote a Gaussian normal distribution with mean μ and variance σ^2 which has a probability density function defined as

$$p(x;\mu,\sigma) := \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{\sigma^2}}. \tag{2.44}$$

This allows us to consider the real outcomes y_i as a set of normally distributed variables as well. By the linearity of the expectation operator we can then compute the expectation of the outcome

$$\langle \hat{y}_i \rangle = \langle y_i + \epsilon \rangle, \tag{2.45}$$
$$\langle \hat{y}_i \rangle = \langle y_i \rangle + \langle \epsilon \rangle, \tag{2.46}$$
$$\langle \hat{y}_i \rangle = \mathbf{x}_i \mathbf{w}. \tag{2.47}$$

The expectation of the error term is zero following the definitions of the expectation we presented in section 2.1. Following the exact same properties we have that the variance of the prediction is the variance of the error term σ^2, i.e.

$$\langle \hat{y}_i \rangle^2 + \langle \hat{y_i}^2 \rangle = \sigma^2. \tag{2.48}$$

In concise terms we simply consider our outcome as a set of IID normal variables on the form $y_i \sim \mathcal{N}(\mathbf{x}_i^T \mathbf{w}, \sigma^2)$. The likelihood of the linear regression can then be written using the same tuple notation as for equation 2.29

$$p(S|\theta) = \prod_i^n \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(\hat{y}_i - \mathbf{x}_i\mathbf{w})^2}{\sigma^2}}, \tag{2.49}$$
$$= \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^n \prod_i^n e^{-\frac{(\hat{y}_i - \mathbf{x}_i\mathbf{w})^2}{\sigma^2}}, \tag{2.50}$$
$$= \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^n e^{-\sum_i \frac{(\hat{y}_i - \mathbf{x}_i\mathbf{w})^2}{\sigma^2}}, \tag{2.51}$$
$$= \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^n e^{-\frac{(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w})^T(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w})}{\sigma^2}}. \tag{2.52}$$

We recall from section 2.7 that the best parameters of a model is defined as

$$\theta^* = \arg\max_\theta p(S|\theta). \tag{2.53}$$

To find the optimal values we then want to take the derivative w.r.t the parameters and find a minimum. But as we saw before this is impractical, if not impossible, with the product sum in the likelihood. To solve this problem we repeat the log-trick from section 2.7 re-familiarizing ourselves with the log-likelihood

$$\log(p(S|\theta)) = n\log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w})^T(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w})}{\sigma^2}. \tag{2.54}$$

Taking the derivative with respect to the model parameters and setting to zero we get

$$\begin{aligned}
\nabla_{\mathbf{w}} \log(p(S|\theta)) &= \nabla_{\mathbf{w}} \left(-\frac{1}{\sigma^2}(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w})^T(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w}) \right), \\
&= -\frac{1}{\sigma^2}(-2\mathbf{X}^T y + 2\mathbf{X}^T\mathbf{X}\mathbf{w}), \\
&= -\frac{1}{\sigma^2}2\mathbf{X}^T(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w}), \\
\mathbf{0} &= -\frac{2}{\sigma^2}(\mathbf{X}^T\hat{\mathbf{y}}_i - \mathbf{X}^T\mathbf{X}\mathbf{w}), \\
\mathbf{X}^T\mathbf{X}\mathbf{w} &= \mathbf{X}^T\hat{\mathbf{y}}_i.
\end{aligned}$$

Which ultimately supplies us with the solution for of the optimal parameters

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\hat{\mathbf{y}}. \tag{2.55}$$

An important note is that the MLE solution is equal to the least squares derivation we performed in section 2.2.

2.10 Gradient Descent

The process of finding minima or maxima of a function is well-trodden ground for physicists. These points along a function are collectively known as extrema, and with Newton and Leibniz's formulation of calculus, we were given analytical procedures for finding extrema by the derivative of functions. The power of these methods is demonstrated by the sheer volume of problems we cast as minimization or maximization objectives. From first-year calculus, we know that a function has an extremum where the derivative is equal to zero. Moreover, for many functions and functionals this has a closed-form solution. For functionals of complicated functions this becomes impractical or impossible.

We showed in section 2.8 that even a relatively simple model like logistic regression is transcendental in the first derivative of the cost. For both too-complex or analytically unsolvable first derivatives, we then turn to iterative methods of the gradient. In this thesis, we will restrict discussions to gradient descent, which is an iterative method of the first order for used for finding function minima. All the optimization problems are thus cast as minimization problems to fit in this framework. We begin by considering the simplest form of gradient

descent of a function, f, of many variables $\mathbf{x}$ and a weighting term for the update, η

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla_{\mathbf{x}} f(\mathbf{x}_n). \tag{2.56}$$

We know that the gradient vector is in the direction of the steepest ascent for the function, moving towards a minimum then requires going exactly the opposite way. The parameter controlling the size of this variable step is $\eta \in \mathbb{R}_+$. This parameter is dubbed the learning rate in machine learning and is the term we will be using. Choosing η is extremely important for the optimization as too low values slow down convergence to a crawl, and can even stall the optimization entirely with the introduction of value decay to the learning rate. Conversely, too high a value for the learning rate jostles the parameter values around in such a way that we might miss the minimum entirely. Figure 2.5 shows the effects of choosing the values for the learning rate poorly, while figure 2.6 shows the effect of a well-chosen eta which finds the minimum in just a few steps. We note that the learning rate is a hyper-parameter, as discussed in section 2.6.

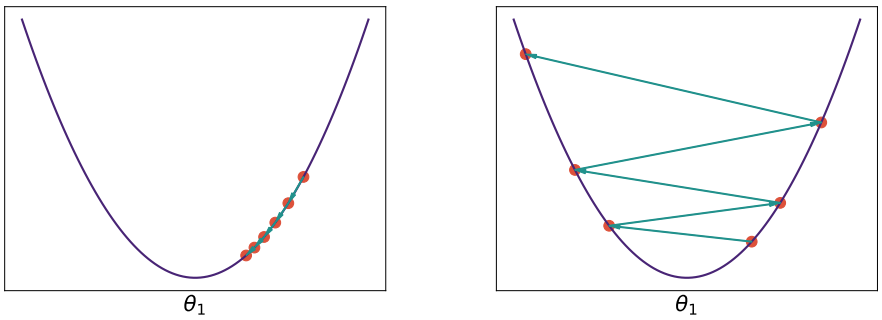
Directly inspecting the progress is not feasible for the high-dimensional updates required for a neural network, or even a multivariate logistic regression. However, as suggested by Karpathy [12], we can indirectly observe the impact of the choice of learning rate. We infer the impact from the shape of the loss as a function of the epoch. In machine learning, an entire gradient update using all the available training data is called an epoch. This impact is shown in figure 2.7, which we will use as a reference when training the models used in this thesis.

Despite its simplicity gradient descent and its cousins have shown to solve remarkably complex problems despite one obvious flaw: convergence is only guaranteed to a local minimum. While first-order methods are much more computationally efficient than higher-order methods, their use brings with them some problems of their own. In particular, there are two problems that need to be solved:

(**C0**): Local minima are usually common in the loss function landscape, traversing these while not getting stuck is problematic for ordinary gradient descent

(**C1**): Converging to a minimum can be slow or miss entirely depending on the configuration of the method

In this section, we will discuss some modifications to the gradient descent procedure proposed to remedy them both. The importance of the methods we discuss in the following sections is also covered in detail by Sutskever et al. [13]. A more extensive and in-depth overview of the methods themselves can be found in Ruder [14].



(a) **(b)**

Figure 2.5: Gradient descent on a simple quadratic function showing the effect of too small, **(a)**, and too large, **(b)**, value for the learning rate η

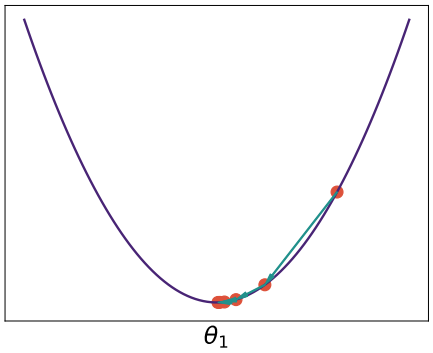


Figure 2.6: Complement to figure 2.5 where we show the effect of a good learning rate on a gradient descent procedure. The gradient descent procedure is performed on a quadratic function.

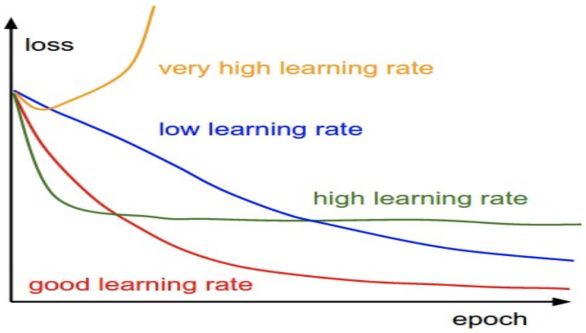


Figure 2.7: A hand-drawn figure which shows the impact of the choice of the learning rate parameter on the shape of the loss function. The optimal choice slowly decays, and we will use that shape as a benchmark when tuning the learning rate in our applications. Copied from the cs231 course material from Stanford authored by Karpathy [12].

2.10.1 Momentum Gradient Descent

The first problem of multiple local minima has a proposed solution that to physicists is intuitive and straightforward: add momentum. For an object in a gravity potential with kinetic energy to not get stuck in local minima of the potential, it has to have enough momentum. While at the same time not having so much that it overshoots the global minimum entirely. It is then with a certain familiarity that we introduce the momentum update which replaces the ordinary gradient with an exponential average over the previous steps controlled by a parameter β

$$\begin{aligned} \mathbf{v}_n &= \beta \mathbf{v}_{n-1} + (1-\beta)\nabla f(\mathbf{x}_n), \\ \mathbf{x}_{n+1} &= \mathbf{x}_n - \eta \mathbf{v}_n. \end{aligned} \tag{2.57}$$

To understand the momentum update we need to decouple the recursive nature of the $\mathbf{v}_t$ term and it's associated parameter β. This understanding comes from looking at the recursive term for a few iterations and observing that this is simply a weighted sum

$$\begin{aligned} \mathbf{v}_n &= \beta(\beta \mathbf{v}_{t-1} + (1-\beta)\nabla f(\mathbf{x}_{n-1}) + (1-\beta)\nabla f(\mathbf{x}_n), \\ \mathbf{v}_n &= \beta(\beta(\beta \mathbf{v}_{t-2} \\ &\qquad + (1-\beta)\nabla f(\mathbf{x}_{n-2})), \\ &\qquad + (1-\beta)\nabla f(\mathbf{x}_{n-1})), \\ &\qquad + (1-\beta)\nabla f(\mathbf{x}_n). \end{aligned}$$

Each $\mathbf{v}_t$ is then an exponentially weighted average over all the previous gradients. This representation indicates that the factor $1-\beta$ controls how much of a view

Table 2.1: Hyperparameter table for momentum gradient descent. These parameters have to be tuned without gradient information, we discuss ways to achieve this in section 2.6

Name	Default value	Scale	Description
β	0.9	Gaussian normal	Exponential decay rate of the momentum step
η	10^{-3}	Linear	Weight of the momentum update

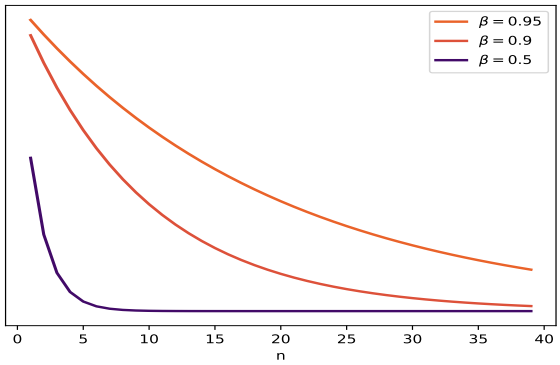


Figure 2.8: A figure illustrating the decay rate of different choices of β. The lines go as β^n and show that one can quite easily infer how many of the previous steps is included for each choice. A good starting value for the parameter has been empirically found to be $\beta = 0.9$. In this thesis we will use a Gaussian distribution around this value as a basis for a random search.

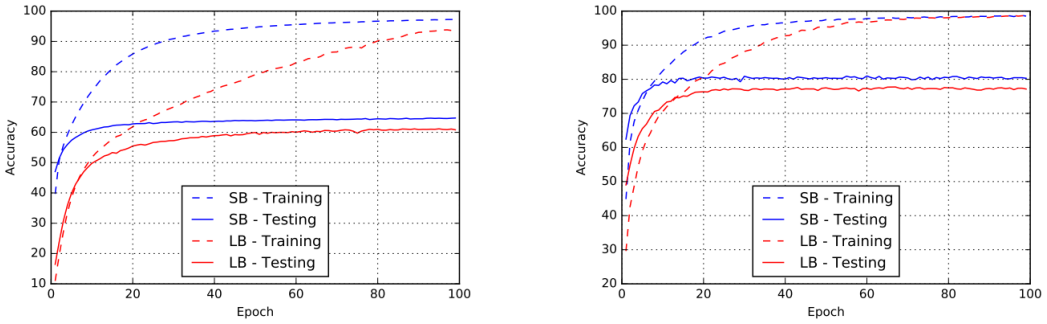
we have of the history of the iteration. Leading to the conclusion that β must be reasonably restricted to $\beta \in [0, 1]$ to avoid overpowering the new gradient. How many steps in the past sequence that this average "sees" we illustrate in figure 2.8. Adding momentum is then a partial answer to the challenge of how to overcome both local minima and saddle regions in the loss function curvature. To summarize we list the parameters that need tuning for a gradient descent with momentum in table 2.1

2.10.2 Stochastic & Batched Gradient Descent

In the preceding sections, we discussed gradient descent as an update we do over the entire data-set. This procedure creates a gradient with minimal noise, pointing directly to the nearest minimum. For most complex models that beelining behavior is something to avoid.

One of the most powerful tools to avoid this behavior is batching, which involves taking the gradient with only a limited partition of the data and updating the parameters. This creates noise in the gradient, which encourages exploration

of the loss-surface rather than strong convergence to the nearest minimum. If we set the batch size to $N = 1$ we arrive at a special case of batched gradient descent known in statistics and machine learning nomenclature as stochastic gradient descent (SGD). As the naming implies, SGD aims to include the noisiness we wish to introduce to the optimization procedure. Both batched gradient descent and SGD show marked improvements compared to performing full-set gradient updates [15]. Batches larger than one are usually employed when the computational cost of SGD becomes prohibitive. We show the effect of batch sizes on performance in figure 2.9.



(a) Network F_2 (b) Network C_1

Figure 2.9: Showing the effect of batch sizes on a fully connected and shallow convolutional network in figure (a) and (b) respectively. The smaller batch-sizes are consistently able to find minima of a higher quality than the large batch versions of the same network. The networks were trained on common machine learning datasets for illustrative purposes. Figure taken from Keskar et al. [15]

2.10.3 adam

One of the breakthroughs in modern machine learning is the improvements made to gradient descent from the most simple version introduced in equation 2.56 to the adam paradigm introduced by Kingma and Lei Ba [16]. Since its conception adam has become the de facto solver for many ML applications. Conceptually, adam ties together stochastic optimization in the form of batched data, momentum, and adaptive learning rates. The latter of which involves changing the learning rate as some function of the epoch, of the magnitude of the derivative, or both. Adding to the momentum part adam maintains an exponentially decaying average over the previous first and second moments of the derivative. Physically this is akin to maintaining velocity and momentum for an inertial system. Mathematically we describe these decaying moments as functions of the first and second moments of the gradient

Name	Default value	Scale	Description
β_1	0.9	Gaussian normal	Exponential decay rate of the fist moment of the gradient
β_2	0.999	Gaussian normal	Exponential decay rate of the second moment of the gradient
η	10^{-3}	Linear	Weight of the momentum update

Table 2.2: Hyperparameter table for adam. These parameters have to be tuned without gradient information, we discuss ways to achieve this in section 2.6

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla f(\mathbf{x}_{t,i}), \tag{2.58}$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\nabla f(\mathbf{x}_{t,i})^2. \tag{2.59}$$

The update now maintains two β parameters analogous to the simple momentum update rule. In the paper Kingma and Lei Ba [16] describe an issue where zero-initialized m_t and v_t are biased towards zero, especially when the decay is small. To solve this problem, they introduce bias-corrected versions of the moments

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \tag{2.60}$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \tag{2.61}$$

Which is then used to update the model parameters in the now familiar way

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{\eta}{\hat{v}_n + \epsilon}\hat{m}_n. \tag{2.62}$$

The authors provide suggested values for $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 1 \times 10^{-8}$. They also recommend that one constricts the values for such that $\beta_2 > \beta_1$.

To summarize, we consider the hyperparameters required for the usage of adam. Both β_1 and β_2 are in principle needed to be tuned, but we restrict the tuning to β_1 in this thesis and freeze the value of β_2 to retain some semblance manageability in the hyperparameter space. The parameters and their scales are listed for convenience in table 2.2.

2.11 Performance validation

The threat of overfitting hangs as a specter over most machine learning applications. Regularization, as discussed in section 2.5, outlines the tools researchers

use to minimize the risk of overfitting. What remains then is the measurement of the performance of the model, and our confidence in that performance. We've already outlined the most simple tool to achieve this in section 2.3; simply split the data in disjoint sets and train on one, measure on the other. As a tool this works best when there is lots of data from which to sample or the purpose of the algorithm is predictive in nature. In this thesis however the purpose is exploratory and labelled data is scarce. Before delving into how to estimate the out-of-sample error we first have to discuss the performance metrics we will use to measure error.

2.11.1 Supervised performance metrics

In this thesis we measure the performance of a classifier with the $f1$ score. However it is helpful to first discuss a simpler metric of performance; the accuracy. Intuitively the accuracy is very satisfying as it is simply the percentage of correct classifications made. The accuracy is then computed from the True Positive (TP) predictions and the True Negatives (TN) divided by the total number of samples. We will use the False Positives (FP) and False Negatives (FN) later and so introduce their abbreviation here.

We note that the accuracy is related to the rand index which we will use to measure unsupervised performance, with the distinction that for accuracy we know the ground truth during training. The accuracy is then defined as

$$\text{accuracy} := \frac{TP + TN}{FN + TN + TP + FP}. \tag{2.63}$$

One of the principal failings of accuracy as presented in equation 2.63 is that it does not account for class imbalance. Consider a problem where one class occurs as 99% of the sample, a trivial classifier predicting only that class will achieve an accuracy of acc = 0.99. This is for obvious reasons a problematic aspect of accuracy. Though a simple remedy is to measure multiple metrics of performance, or to change measurements altogether. We chose the $f1$ score per-class and total $f1$ score, as it allows for comparisons with earlier work on the same data from Kuchera et al. [17]. The $f1$ score is defined in terms of the precision and recall of the prediction. Which are simply defined as true positives weighted by the false positives and negatives. We define recall and precision in equations 2.64 and 2.65 respectively.

$$\text{recall} = \frac{TP}{TP + FP} \tag{2.64}$$

$$\text{precision} = \frac{TP}{TP + FN} \tag{2.65}$$

The $f1$ score is then defined as the harmonic mean of precision and recall for each class. Formally it is given as shown in equation 2.66.

$$f1 = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{2.66}$$

Note that the $f1$ score does not take into account the FN predictions. But in nuclear event detection the now flourishing amount of data weights the problem heavily in favor of optimizing for TP and FP predictions, and so the $f1$ score is a well suited performance measure for this problem.

2.11.2 labelled samples

One of the principal challenges with the experimental data discussed in this thesis is that labelled data is challenging to acquire, if not impossible to acquire. In the best case scenario it's still computationally intensive to label individual events and in the worst case scenario the current Monte Carlo based fitting methods might not be able to separate event types of interest from background noise and unknown reactions.

It is then interesting to quantify the effect of the amount of accessible labelled data on a semi-supervised approach as listed in chapter 5. Starting from a random, small, sample of the labelled data we train a classifier on a subset of the labelled data iteratively adding to that subset.

2.11.3 Cross validation

To estimate the out-of-sample error as discussed in section 2.4 one can use simple statistical tools. The premise is that by iteratively selecting what data the model gets to train on and what it doesn't we can compute a less biased estimate of the out of sample error, compared to simply taking the training performance. The division in sub-sets mimics the expectation computed in equation 2.15 over the data-selection sensitive model parameters θ^*_S.

This idea of iterative sampling is known collectively as cross validation, and the manner in which the sampling is conducted specifies the type of cross validation performed. In this thesis we use the technique called k-fold cross-validation.

The algorithm consists of separating the data in k equally sized folds. A fold is a a tuple of a corresponding target and data sub-set. If the data is very biased or k is high it might be useful to ensure that each fold roughly follows the global class distribution. A model is then trained on all but one of the folds, and the out of sample error is estimated on the last fold. This is repeated such that all folds are left out exactly once, creating a k-long vector with performance estimates. The average of which then represents our estimation of the true performance of the model.

2.12 Unsupervised learning

In section 2.7 we presumed the data was comprised of system states and measured outcomes of the set of states. However, for some applications, we do not have measured outcomes of states and are left with only the data itself. In this case, we often wish to extract information about the differences in the data from the data itself. Clustering is one such approach wherein similarities are measured between points of data and grouped based on respective similarities.

Clustering analysis is well illustrated by the k-means algorithm [18], which is to clustering what logistic regression is to classification. In the k-means algorithm, one uses the euclidean distance between each point in the data and objects in data space called centroids to allocate clusters. The simplest version of the k-means algorithm initializes random cluster centroids and follows an update rule where new centroids are allocated by computing the mean of all points that are closest to the given centroid. This procedure is repeated until convergence of the centroid locations. Of course, this algorithm is dependent on the cluster initialization and performs poorly in high-dimensional spaces where the euclidean distance loses its discriminatory power.

The latter restriction can be addressed by representing the data in a much smaller dimensional space. Choosing this representation then becomes the primary challenge we are trying to solve. A naive approach is to find the eigenvectors of the covariance matrix and project the data along the axes with the highest corresponding eigenvalues. This method is called PCA (principal component analysis, [19]) and has been a staple in the machine learning community for decades. The PCA defines a dimensionality reduction by a projection of the data along the major axes of variation. More formally we let $\mathbf{Z} = \langle \mathbf{X}^T \mathbf{X} \rangle$ be the covariance matrix of our data. The principal components, $\mathbf{p}_i$, are then the eigenvectors of Z

$$\lambda_i \mathbf{Z} = \mathbf{p}_i \mathbf{Z}, \tag{2.67}$$

with eigenvalues λ_i. The immediately apparent problem with PCA is that it captures linear axes of variation, while the data might very well have non-linear relationships. In principle, one can modify this property by using a kernel trick to evaluate the projections along kernelized principal components. Common choices for the kernel are the linear kernel, $\mathbf{x}^T \mathbf{x}'$, and the Gaussian kernel, $e^{-||\mathbf{x}-\mathbf{x}'||_2^2}$. The representations of the data in the kernel space are never evaluated; we only compute the associated inner-product space allowing for high dimensional kernels to be used [20].

In practice, methods with more flexibility have seen more use, but PCA is still often the first step in an exploratory analysis pipeline. Of the more flexible non-linear methods, we will focus on the neural network, more specifically the autoencoder. The neural network architecture known as autoencoders describes

models that aim to achieve efficient information-bottlenecking of complex data by reconstruction. In order to form an efficient information bottle-neck, the auto-encoder enacts two non-linear mappings. One from the input dimension to a much lower-dimensional projection that is the map $\psi : \mathbb{R}^n \to \mathbb{R}^m$, $n >> m$, which we call the encoder network. The second part of the model then maps from the lower-dimensional projection to the original dimension, i.e., $\phi : \mathbb{R}^m \to \mathbb{R}^n$, and is known as the decoder. The objective of this model is then the reconstruction of the input, which means the compression map ψ is optimized to capture salient information about the data. These compressions are interesting because they have been shown to convey semantic information through the compression given some restrictions on the compression that we discuss in the next chapter [21].

In this thesis, we will use autoencoder models to explore the semantic information when processing events from a nuclear physics experiment. We will explore both the case where no labelled data exists and where there exists some subset of labelled data. Autoencoders and how to encourage semantic information in the compressed representation is discussed in greater detail in chapter 4

2.13 Unsupervised performance metrics

When performing clustering and other unsupervised methods, the goal is to estimate some class belonging without the model having knowledge of this correspondence between class and data. This disconnect makes measuring performance a little more abstract. In general, there is a separation between performance metrics that assumes that one has access to ground-truth labels and those that do not. We will principally use the adjusted rand score (ARS) which formally defines the agreement between two separate clusterings adjusted for random chance. To compute the ARS, we first need to define the contingency table, similar to the confusion matrix from classification, which lists the coinciding elements of one clustering with another. A general representation of this is shown in table 2.3.

Introduced by [22] the ARS computes the agreement between the two clusterings d_i and c_j. As with the supervised metrics we use the scikit −learn imple-

Table 2.3: General form of a contingency table. d_n and c_n are classes measured by two different processes on the same data. The n_{ij}'s then describe how many samples are in clusters d_i and simultaneously in c_j

	c_1	c_2	$\cdots$
d_1	n_{11}	n_{12}	$\cdots$
d_2	n_{21}	n_{22}	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$

mentation of the ARS [23]. The computation is performed as shown in equation 2.68:

$$\text{ARS} = \frac{\sum \binom{n_{ij}}{2} - \left[\sum \binom{x_i}{2} \sum \binom{y_j}{2}\right] / \binom{n}{2}}{\frac{1}{2}\left[\sum \binom{x_i}{2} + \sum \binom{y_j}{2}\right] - \left[\sum \binom{x_i}{2} \sum \binom{y_j}{2}\right] / \binom{n}{2}}. \tag{2.68}$$

In equation 2.68 we introduce the terms x_i and y_i which are just the row-wise and column-wise sums of the contingency table, respectively.

If we take one of the clusterings to be a ground-truth measurement, the ARS becomes a clear measure of performance.

Chapter 3

Deep learning theory

Deep learning describes the subfield of machine learning which uses neural network-based algorithms. A neural network is a computational structure that generalizes the logistic and linear regression framework introduced in the previous chapter.

This thesis explores to what degree we can extract salient information about different nuclear reactions occurring in an active target time projection chamber (AT-TPC) experiment using modern deep learning methods. To achieve this, we explore the latent spaces of various deep learning algorithms. Mainly we employ the deep recurrent attentive writer (DRAW) algorithm [24] and variations of a traditional autoencoder, in conjunction with logistic regression and various clustering algorithms. Both of the deep learning architectures are first used to explore our ability to create class-separating compressions. We then modify those algorithms to investigate the feasibility of clustering techniques on the compressed space.

Building up to the algorithms used for analysis in this thesis, we start by introducing the underlying neural network framework in section 3.1. Moving forward, we discuss the algorithm for optimizing such networks with a gradient descent procedure. We discuss gradient descent in some detail in section 2.10. Moreover, we also consider the fundamental constituents of the neural network: the layer structure and the activation function. To round out the chapter, we discuss ways to modify both the layer structure and activation function to fit different kinds of data.

A neural network consists of multiple layers that transform the input such that it can be used in e.g., classification. Each layer is traditionally expressed as an inner product, similar to how we formulated linear regression in equation 2.40, stacked on top of each other. The layers may also be some more complex transformation. We will touch different formulations for neural networks used image analysis, and for time-series. Between each layer, a non-linear function is applied to enhance the expressibility of the model further. Choosing the correct nonlinearity has been a subject of debate for the last decade in deep learning literature.

Lastly, we introduce the analysis pipelines used in this work. We utilize different models in conjunction with choice tools for the measurement of performance and a framework for hyper-parameter tuning.

3.1 Neural networks

While the basis for the modern neural network was laid more than a hundred years ago, modern neural networks were proposed by McCulloch and Pitts [25]. They described a computational structure analogous to a set of biological neurons. The premise is that a biological neuron accepts signals from many other adjacent neurons, and if it receives a large enough signal it fires, and emits a signal to those neurons it is connected to.

Dubbed an artificial neural network (ANN) this computational analogue takes input from multiple sources, weights that input and produces an output if the signal from the weighted input is strong enough. A proper derivation will follow but for the moment we explore this simple intuition. These artificial neurons are ordered in layers, each successively passing information forward to a final output. Depending on the application, the output can be categorical or real-valued in nature. Each layer uses a matrix multiplication and a non-linear function to transform the input space in a way that condenses information, as most applications use transformations that reduce the dimensionality substantially before making a prediction.

A simple illustration of two neurons in one layer is provided in figure 3.1.

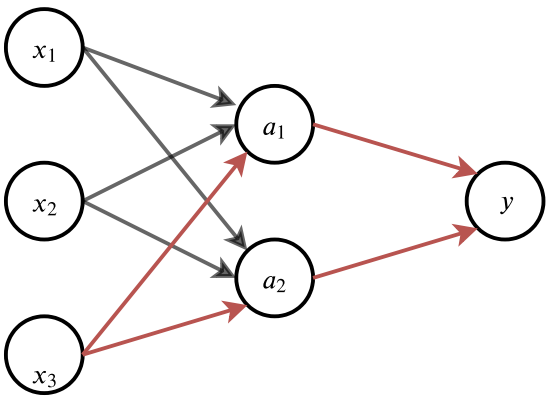


Figure 3.1: An illustration of the graph constructed by two artificial neurons with three input nodes. Colored lines show the flow of information from input to the predicted value. For details on notation see the text.

The ANN produces an output by what we call a forward pass. This defines all the mathematical operations from the input to the output of the network. More formally, let the input to an ANN be $\mathbf{x} \in \mathbb{R}^N$, and the matrix $\mathbf{W}^{[1]} \in \mathbb{R}^{N \times D}$ be

a representation of the weight matrix forming the connections between the input and the artificial neurons, each layer has its own weights which we denote with bracketed superscripts. For a network with n layers each layer then maintains a weight matrix

$$\mathbf{W}^{[l]} \; \forall \; l \in \{1, 2, \ldots, n\}, \tag{3.1}$$

which are used to transform the input to a space which enables regression or classification.

Lastly we define the activation function $f(x)$ as a monotonic, once differentiable, function on $\mathbb{R}^1$. The function $f(x)$ determines the complexity of the neural network together with the number of neurons per layer and number of layers. The use of a non-linear activation is what allows for the ANN to represent more complex problems than we were able to with logistic and linear regression.

We are now ready to fully describe the forward pass, which transforms the input to an intermediate representation $\mathbf{a}$. The simplest such transformation is the fully connected, or *dense*, layer. It bears close resemblance to the formulation of linear and logistic regression and is defined as

$$\mathbf{a}^{[1]} = f(\mathbf{x}\mathbf{W}^{[1]})_D, \tag{3.2}$$

for the first layer in the network with inputs $\mathbf{x}$. In equation 3.2 the subscript, D, denotes that the function is applied element-wise on the output of dimension $\mathbb{R}^D$.

Each node is additionally associated with a bias, ensuring that even zero valued weights in $\mathbf{W}^{[l]}$ can encode information. Let the bias for the layer be given as $\mathbf{b} \in \mathbb{R}^D$ in keeping with the notation above. Equation 3.2 then becomes

$$\mathbf{a}^{[1]} = f(\mathbf{x}\mathbf{W}^{[1]} + \mathbf{b}^{[1]})_D. \tag{3.3}$$

To tie the notation back to more traditional methods we note that if we only have one layer and a linear activation $f(x) = x$ the ANN becomes a formulation of a linear regression model. Keeping the linear regression formulation in mind we illustrate the difference by showing the full forward pass of a two-layer neural network, where we also introduce the output activation function $o(\cdot)$

$$\mathbf{a}^{[1]} = f(\mathbf{x}\mathbf{W}^{[1]} + \mathbf{b}^{[1]})_D, \tag{3.4}$$

$$\mathbf{y} = o(\mathbf{a}^{[1]}\mathbf{W}^{[2]} + \mathbf{b}^{[2]})_D. \tag{3.5}$$

The difference between the intermediate representations $\mathbf{a}^{[l]}$ and the output $\mathbf{y}$ is largely semantic. However, we find it useful to separate the notations in an effort to provide clarity, and because the last layer of a network usually has an

activation which differs from the rest of the network. This output activation $o(\cdot)$ is usually formulated depending on the analysis at hand. For regression tasks o is commonly just a linear activation. Conversely for classification the logistic sigmoid is used for binary outcomes, or in the event of multiple classes we use the soft-max function. For a network with n layers the soft-max function is defined as

$$o(\mathbf{z})_i = \frac{e^{z_i^{[n]}}}{\sum_j e^{z_j^{[n]}}}, \tag{3.6}$$

where we introduce our notation for the input to the activation at each layer as $z_j^{[l]}$. We explicitly define this input as

$$z_j^{[l]} = a_i^{[l-1]} W_{ij}^{[l]} + b_j^{[l]}, \tag{3.7}$$

such that the intermediate representation as seen by the next layer is

$$a_j^{[l]} = f(z_j^{[l]})_D. \tag{3.8}$$

We seamlessly transition to a network of many layers n which is a simple extension of the two-layer network presented above. A multi-layer network can then be described in terms of its forward pass as

$$\mathbf{a}^{[1]} = f(\mathbf{x}\mathbf{W}^{[1]} + \mathbf{b}^{[1]})_D, \tag{3.9}$$

$$\mathbf{a}^{[2]} = f(\mathbf{a}^{[1]}\mathbf{W}^{[2]} + \mathbf{b}^{[2]})_D, \tag{3.10}$$

$$\vdots \tag{3.11}$$

$$\mathbf{y} = o(\mathbf{a}^{[n-1]}\mathbf{W}^{[n]} + \mathbf{b}^{[n]})_D. \tag{3.12}$$

Furthermore we note that the dimensions of the weight matrices are largely user specified, excepting the first dimension of $\mathbf{W}^{[1]}$ which maps to the input dimension, and the second dimension of the last layer which maps to the output. Otherwise the first dimension is chosen to fit with the previous output and the second is specified as a hyperparameter. Recall from section 2.6 that hyperparameters have to be specified and tuned outside the ordinary optimization procedure and are usually related to the complexity of the model and so cannot be arbitrarily chosen.

We can now turn to the process of optimizing the model. In a neural network the variables that need to be fit are the elements of $\mathbf{W}^{[l]}$ that we denote $W_{ij}^{[l]}$, and the biases which we denote with $b_j^{[l]}$. And while one can solve the linear regression optimization problem by matrix inversion, the multi-layer neural net does not have a closed form first derivative for $W_{ij}^{[l]}$ or $b_j^{[l]}$ because of the non-linearities between each layer. We are then forced to turn to iterative methods of the gradient, which we previously introduced in section 2.10

Based on whether the output is described by real values or a set of probabilities the cost takes on different forms, just as for linear and logistic regression. In the real case we use the now familiar mean squared error (MSE) cost, or in the event that we want to estimate a probability of an outcome; the binary cross-entropy. We discuss these cost-functions in general and especially the MSE and binary cross-entropy (BCE) earlier on in chapter 2. Regardless of the cost the optimization problem is solved by a gradient descent procedure discussed in section 2.10. We we re-introduce the update form for the parameters as

$$W_{ij}^{[l]} \leftarrow W_{ij}^{[l]} - \eta \frac{\partial \mathcal{C}}{\partial W_{ij}^{[l]}}. \tag{3.13}$$

3.1.1 Backpropagation

In the vernacular of the machine learning literature the aim of the optimization procedure is to train the model to perform optimally on the regression, reconstruction or classification task at hand. Training the model requires the computation of the total derivative in equation 3.13. This is also where the biological metaphor breaks down, as the brain is probably not employing gradient descent.

Backpropagation of errors by automatic differentiation, first described by Linnainmaa [26], is a method of computing the partial derivatives required to go from the gradient of the loss to individual parameter derivatives. Conceptually we wish to describe the slope of the error in terms of our model parameters, but having multiple layers complicate this somewhat.

The backpropagation algorithm begins with computing the total loss, here exemplified with the squared error function,

$$E = \mathcal{C}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2n} \sum_n \sum_j (\hat{y}_{nj} - y_{nj})^2. \tag{3.14}$$

The factor one half is included for practical reasons to cancel the exponent under differentiation. As the gradient is multiplied by the learning rate η, this is ineffectual on the training itself.

The sums over n and j enumerate the number of samples, and output dimensions respectively. Finding the update for the parameters then starts with taking the derivative of equation 3.14 w.r.t the model output $y_j = a_j^{[l]}$

$$\frac{\partial E}{\partial y_j} = \hat{y}_j - a_j^{[l]}. \tag{3.15}$$

We have dropped the data index, as the differentiation is independent under the choice of data. In practice the derivative of each sample in the batch is averaged together for the gradient update of each parameter.

The activation function, f, has classically been the logistic sigmoid function, but during the last decade the machine learning community has largely shifted to using the Rectified linear unit (ReLU). This shift was especially apparent after the success of Krizhevsky et al. [27]. In this section we then exemplify the backpropagation algorithm with a network with ReLU activation. The ReLU function is defined in such a way that it is zero for all negative inputs and the identity otherwise, i.e.

$$\text{ReLU}(x) = f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{3.16}$$

The ReLU is obviously monotonic and its derivative can be approximated with the Heaviside step-function which we denote with $H(x)$ and is mathematically expressed as

$$H(x) = f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{3.17}$$

Common to most neural network activations the computation of the derivative is very light-weight. In the case of the *ReLU* function the computation of the derivative uses the mask needed to compute the activation itself, requiring no extra computing resources.

It is important to note that the cost and activation as introduced in equations 3.14, 3.16 and 3.17 is not a be-all-end-all solution, but chosen for their ubiquitous nature in modern machine learning.

Returning to the optimization problem we start to unravel the backpropagation algorithm. We use equation 3.15 to find the derivatives in terms of the last parameters, i.e. $W_{ij}^{[n]}$ and $b_j^{[n]}$

$$\frac{\partial E}{\partial W_{ij}^{[n]}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j^{[n]}}\frac{\partial z_j^{[n]}}{\partial W_{ij}^{[n]}}, \tag{3.18}$$

$$= \frac{\partial E}{\partial y_j} o'(z_j^{[n]}) \frac{1}{\partial W_{ij}^{[n]}} \left(a_i^{[n-1]} W_{ij}^{[n]} + b_j^{[n]} \right), \tag{3.19}$$

$$= (\hat{y}_j - y_j) o'(z_j^{[n]}) a_i^{[n-1]}. \tag{3.20}$$

The differentiation of the error w.r.t to b_j can be similarly derived to be

$$\frac{\partial E}{\partial b_j^{[n]}} = (\hat{y}_j - y_j) o'(a_j^{[n]}). \tag{3.21}$$

Repeating this procedure layer by layer is the process that defines the backpropagation algorithm. From equations 3.20 and 3.21 we discern a recursive pattern

in the derivatives moving to the next layer. Before writing out the full backpropagation term we will introduce some more notation that makes bridging the gap to an implementation considerably simpler. From the repeating structure in the aforementioned equations we define the first operation needed for backpropagation,

$$\delta_j^n = (\hat{y}_j - y_j)o'(z_j^{[n]}). \tag{3.22}$$

Note that this is an element-wise Hadamard product and not an implicit summation, expressed by the subscript index in $\hat{\delta}_j^n$. The element-wise product of two matrices or vectors is denoted as

$$\mathbf{a} \circ \mathbf{b}. \tag{3.23}$$

This short-hand lets us define equations 3.20 and 3.21 in a more compact way

$$\frac{\partial E}{\partial w_{ij}^{[n]}} = \delta_j^n a_i^{[n-1]}, \tag{3.24}$$

$$\frac{\partial E}{\partial b_j^{[n]}} = \delta_j^n. \tag{3.25}$$

From the iterative nature of how we construct the forward pass we see that the last derivative in the chain for each layer, i.e. those in terms of the weights and biases, have the same form

$$\frac{\partial z_j^{[l]}}{\partial w_{ij}^{[l]}} = a_i^{[l-1]}, \tag{3.26}$$

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1. \tag{3.27}$$

These derivatives together with a general expression for the recurrent term δ_j^l are then the pieces we need to compute the parameter update rules. By summing up over the connecting nodes, k, to the layer, l, of interest δ_j^l can be expressed as

$$\delta_j^l = \sum_k \frac{\partial E}{\partial a_k^{[l+1]}} \frac{\partial a_k^{[l+1]}}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}}, \tag{3.28}$$

$$\delta_j^l = \sum_k \delta^{l+1} \frac{\partial z_k^{[l+1]}}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}}. \tag{3.29}$$

From the definitions of the $z_j^{[l]}$ and $a_j^{[l]}$ terms we can then compute the last derivatives. These are then inserted back into 3.29, giving a final expression for δ_j^l,

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{jk}^{[l+1]} f'(z_j^{[l]}). \tag{3.30}$$

Finally the weight and bias update rules can then be written as

$$\frac{\partial E}{\partial w_{jm}^{[l]}} = \delta_j^l a_m^{[l-1]}, \tag{3.31}$$

$$\frac{\partial E}{\partial b_j^{[l]}} = \delta_j^l. \tag{3.32}$$

To finalize the discussion on the algorithm we illustrate how backpropagation might be implemented in algorithm 2.

Algorithm 2: Backpropagation of errors in a fully connected neural network for a single sample $\mathbf{x}$.

Data: Iterables $\mathbf{a}^{[l]}$ $\mathbf{z}^{[l]}$ $\mathbf{W}^{[l]}$ $\mathbf{b}^{[l]}$ $\forall\, l \in [1, 2, \ldots, n]$
Input: $\frac{\partial E}{\partial \mathbf{y}}$, $o'(\mathbf{z}^{[\mathbf{n}]})$, $f'(\cdot)$
Result: Two iterables of the derivatives $\frac{\partial E}{\partial w_{ij}^{[l]}}$ and $\frac{\partial E}{\partial b_j^{[l]}}$

Initialization;
$\delta_j^n \leftarrow \frac{\partial E}{\partial \mathbf{y}} \circ o'(\mathbf{z}^{[\mathbf{n}]})$;
Compute derivatives;
for $l \in [n-1, \ldots, 1]$ **do**
  $\frac{\partial E}{\partial w_{jm}^{[l]}} \leftarrow \hat{\delta}_j^{l+1} a_m^{[l]}$;
  $\frac{\partial E}{\partial w_{jm}^{[l]}} \leftarrow \hat{\delta}_j^{l+1}$;
  $\delta_j^{l+1} \leftarrow \sum_k \delta_k^{l+1} w_{jk}^{[l+1]} f'(z_j^{[l]})$
return $\frac{\partial E}{\partial w_{ij}^{[l]}}$ *and* $\frac{\partial E}{\partial b_j^{[l]}}$

The backward propagation framework is highly generalizable to variations of activation functions and network architectures. The two major advancements in the theory of ANNs are both predicated on being fully trainable by the backpropagation of errors. Before we consider these improvements made by the introduction of recurrent neural networks (RNN) and convolutional neural networks (CNN), we remark again on the strength of this algorithm. We are not only free to chose the activation function from a wide selection, the backpropagation algorithm also makes no assumptions on the transformation that constructs z_j.

As long as it is once differentiable we are free to choose a different approach. This flexibility of the framework is part of the resurgent success of deep learning in the last decade.

3.1.2 Neural network architectures

When creating a neural network on a computer with finite resources, a principled consideration must be made on the width and depth of the network. These terms are standard in machine learning literature and describe how many nodes per layer, and how many layers a network consists of respectively. A discussion of this consideration is neatly summarized in the work of Lin et al. [2]. The authors provide strong reasoning for prioritizing deep networks over wide ones. They show that one can view many physical systems that generate the data a causal hierarchy (see figure 3 in Lin et al. [2] for an illustration). This representation of stepwise transformations intuitively lends itself well to representation by a sequence of layers. This intuition builds on the fact that each layer contains a transformed, compressed, representation of the data. It is this bottleneck property of information compression that motivates the use of autoencoders, neural networks that compress and reconstruct the input when unlabelled data is plentiful and labelled data is scarce.

3.1.3 Activation functions

Building neural networks depend in large part on the choice of the non-linearity that acts on the output from each layer. They are intrinsically tied to the attributes of the optimization scheme. Gradients have to pass backwards through all the layers to adjust the weights to make better predictions in the next iterations. In this section, we will discuss the attributes, strengths, and weaknesses of the four principal functions used as activations in neural networks.

Sigmoid activation functions

Two sigmoid functions have been traditionally used in neural networks, and while they are now mostly of interest for educational purposes, they still see some use in niche models. Through logistic regression, we have already been introduced to the logistic sigmoid function. The logistic sigmoid was for the same reasons used in early classifying neural networks. Mathematically it has the form

$$\sigma(x) = \frac{1}{1 - e^{-x}}. \tag{3.33}$$

As with the ReLU activation we introduced in the previous section the sigmoid activation has a derivative which is very cheap to compute, namely

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x)) \tag{3.34}$$

The second sigmoid activation is the hyperbolic tangent function. It saw widespread use at the start of the 2000s, especially in recurrent neural networks which we will discuss in greater detail later in this chapter. It is defined as

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.35}$$

We observe that the hyperbolic tangent is simply a shifted and scaled version of the logistic sigmoid. Its derivative is slightly more expensive than for the logistic sigmoid, and it is given as

$$\frac{d \tanh x}{dx} = 1 - \tanh^2 x \tag{3.36}$$

The challenge with the sigmoid functions is that the gradient saturates. A saturated gradient occurs where there is little or no change in the values as a function of x. We illustrate the sigmoid activation functions in figure 3.2. From this figure, we observe that for relatively small values the gradient goes to zero, which makes optimization hard. Additionally, we note one of the convenient properties of the sigmoid functions; their values are capped. This prohibits the gradient from exploding but also causes problems in the gradient. We also note that the logistic sigmoid is capped at zero, like the ReLU function, while the hyperbolic tangent is bounded at -1. The hyperbolic tangent then has the option to emit a bounded anti-signal.

Rectifiers

On the other end of the scale rectified activations have gained traction in later years. We introduced the ReLU and its derivative in the previous section, but we will discuss a popular cousin here. The LReleaky rectified linear unit (LU) takes the ReLU and modifies it slightly by adding a small slope to the negative part of the activation. Mathematically we define the LReLU as

$$\text{LReLU}(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha \cdot x & \text{otherwise,} \end{cases} \tag{3.37}$$

where α is a small positive real number. The differentiation of the LReLU is then

$$\frac{d}{dx}\text{LReLU}(x) = \begin{cases} 1 & \text{if } x > 0 \\ \alpha & \text{otherwise.} \end{cases} \tag{3.38}$$

Immediately it becomes clear that for the rectifier functions we have the opposite problem from what we encountered for the sigmoid activations. As the derivative

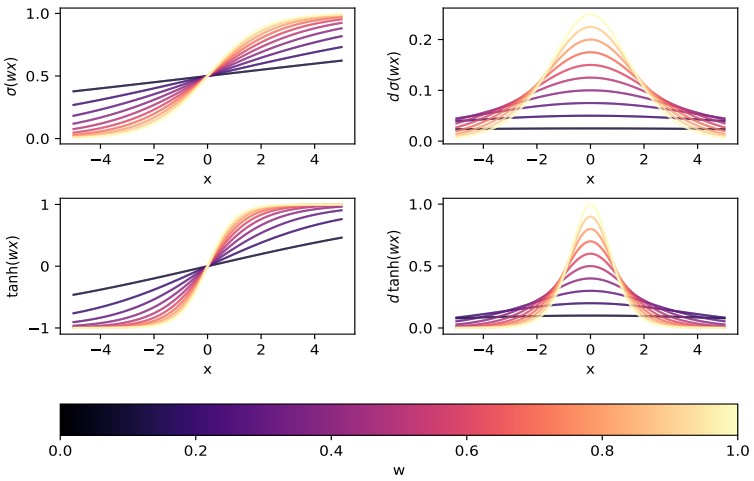


Figure 3.2: Sigmoid activation functions and their derivatives. The colorization indicates the value of the multiplicative weight w. The weight is multiplied with function argument x, which is what we differentiate with respect to. We observe that the derivative deteriorates to zero as the function moves away from zero. The derivative going to zero means that a network using sigmoid activations generally struggle with saturated gradients which slows down training dramatically.

is non-zero for a large range of values, we no longer have a concern for vanishing gradients, but the activation is not positively bounded. This leads to a problem where the gradient might explode, leading to an unstable optimization that fails to find a minimum of the cost. We illustrate the defining features of the rectifiers in figure 3.3

One heuristic method of handling exploding gradients that has seen wide applications is gradient capping. When computing the gradients, one chooses a maximum value for e.g., the norm of the gradient of a layer and shrink any update that crosses that threshold. The downside of this is that it can slow down training substantially. More elegant solutions are usually employed in conjunction with a capped gradient which we will discuss in the next section on the regularization of neural networks.

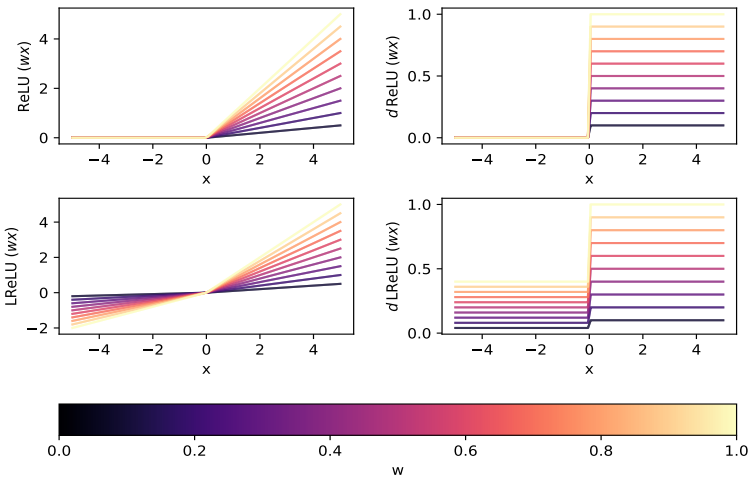


Figure 3.3: Rectifier activation functions and their derivatives, the colorization indicates that the function is multiplied with a second variable. In contrast with the sigmoid activations the derivative is non-zereo in $\mathbb{R}^+$ for the ReLU and non-zero on the entirety of $\mathbb{R}$ for the LReLU.

3.2 Deep learning regularization

As neural networks can be made arbitrarily complex, reducing the degree of overfitting is an important concern. Regularization of deep learning uses both tried and true algorithms, as we discussed in chapter 2, and some specialized tools developed for use with neural networks specifically.

Traditional measures of regularization like using cross-validation and weight constraints are both important features of reducing overfitting in deep learning algorithms. We introduced cross-validation in section 2.11.3 as a way to estimate appropriate model complexities. Cross-validation is model agnostic and can be applied to estimate an appropriate complexity for an arbitrary model or the optimal values for the gradient descent hyperparameters.

Weight constraints are similarly convenient additions to the cost function. The only modification required from equation 2.21 is adding a summation term over the layers. For a L_2 regularized neural net the cost function can then be written as

$$C(\hat{y}_i, f(\mathbf{x}_i; \theta)) = (\hat{y}_i - f(\mathbf{x}_i; \theta))^2 + \lambda \sum_{lij} ||w_{ij}^{[l]}||_2^2. \tag{3.39}$$

More specific tools for neural networks have been developed in later years, and we will discuss two of these that has seen extensive application in recent works.

Dropout

Armed with the knowledge that neural networks are usually more complex than the problem at hand warrants, Srivastava et al. [28] proposes a wonderfully simple remedy: dropout. The premise is simple: after the activation of a layer, randomly set d of these activations to zero. The fraction of the activations that d represents is called the dropout-rate and is typically chosen to be a few tenths. Dropout adds a regularizing effect by having the network increase the redundancy of its prediction and thus forcibly reducing the complexity of the network.

Batch Normalization

The second addition we discuss more directly addresses a problem in the optimization, and adds a regularizing effect only as a bi-product of the intended purpose. Batch normalization was introduced by Ioffe and Szegedy [29] as a means to address the internal covariance shift in neural networks.

Conceptually one can think of the challenge presented by an internal covariance shift as an unintended consequence of the gradient descent procedure. When adjusting weights in a layer its gradients, we do adjust them with respect to the change in the weights of the preceding layers. This slows down training substantially and is what Ioffe and Szegedy [29] describes as the internal covariance shift. To reduce the effects of this problem the authors propose to scale and center the outputs of each layer, before feeding it to the next in line. The normalization happens over each batch of training data and consists of two steps. Let the output from a layer be given as $a_k^{[l]}$ in keeping with previous notation. Furthermore, let the samples in a batch be indicated by the index i. Thus the activations can be denoted as a matrix $a_{ik}^{[l]}$. The batch normalization procedure then begins by computing the batch-wise mean, μ_k, per feature k for a batch of size n

$$\mu_k = \frac{1}{n}\sum_i^n a_{ik}^{[l]}. \tag{3.40}$$

Secondly we compute the variance of the feature as

$$\sigma_k^2 = \sum_i (a_{ik}^{[l]} - \mu_k)^2. \tag{3.41}$$

We can then compute the normalized activations using the mean and variance of the features, mathematically we then compute the normalized activations as

$$\hat{a}_{ik}^{[l]} = \frac{\hat{a}_{ik}^{[l]} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}, \tag{3.42}$$

where ϵ is a small number large enough to make the denominator non-zero, usually $\epsilon = 1 \times 10^{-8}$. The final part of the batch normalization procedure is to let the

network determine a scale and shift of the new features, adding new features to the gradient descent procedure. The final expression for the features being fed forward is then

$$\mathrm{BN}_{\gamma,\beta}(a_{ik}^{[l]}) = \gamma \hat{a}_{ik}^{[l]} + \beta. \tag{3.43}$$

The primary effect of batch normalization is then to speed up training. [29] shows that orders of magnitude larger learning rates may be used in experiments. Additionally, there is a contribution to the regularization by providing non-deterministic outputs as the normalization parameters are dependent on the other samples in a batch.

As a bonus effect, the procedure lessens the potential problems with exploding and vanishing gradients: By placing the normalization layer before a sigmoid layer the probability of the activations being in the active region of the functions increases. Conversely, for the rectifier units placing the normalization after the activation lessens the chance for an exploding value in the gradient.

For all the merits of batch normalization, a fundamental challenge still remains. Since the normalization is over the feature axis, it adds a substantial number of parameters. For models who heavily use normalization, the scale and shift parameters can add up to $20\% - 30\%$ of the weights in a model. Additionally, we observe that the computation involves a square root and a squaring operation which principally necessitates a full 64 bit precision in the floating-point computation.

3.2.1 Convolutional Neural Networks

Equipped with an understanding of the constituent parts of a fully connected neural network as described in section 3.1, we can now introduce different transformations used in modern neural networks. We begin this discussion with the introduction of the convolutional layer. Originally designed for image analysis tasks convolutional networks are now ubiquitous tools for sequence analysis, images, and image-like data.

Data from the active target time projection chamber (AT-TPC), can be represented in a two-dimensional projection. While these projections do not exhibit all the same properties as traditional images, the analysis tools used for images are still applicable as shown by Kuchera et al. [17]. The primary differences owe to the fact that the AT-TPC data lacks invariance under translation, rotation, and scale: Simply because the physics would change under variations of these properties. However, invariance under these properties is commonly applied in image analysis to improve generalization. Researchers typically employ tools such as translating objects or zooming to increase the amount of data available artificially.

We begin our discussion of convolutional networks with a comparison to ordinary fully connected, or *dense*, networks introduced previously in this chapter.

There are a couple of challenges with neural network layers as they were introduced in section 3.1. Firstly, the number of parameters can quickly get out of hand, as the number of parameters in a dense layer is the product of the input and output nodes. For example, a layer with 10^3 inputs and 10^2 nodes contains 10^5 parameters. Another challenge the dense layer faces is the local structure in the data. As convolutional layers were developed primarily for images, the forward pass is constructed in a way that captures local structures in an efficient manner. In short, the advantage of convolutional layers is an allowance for a vastly reduced number of parameters at the cost of much higher demands of memory. The increased memory cost comes from the fact that convolutional networks make many transformations of the input at each layer that must be stored.

The increased memory cost comes from how we construct the convolutional layer. Each layer maintains k filters, or kernels, each of which is a $n \times m$ matrix of weights. To compute the forward pass a stack of k filters, $\mathbf{F}$, is convolved with the input by taking the inner product with a sliding $n \times m$ patch of the image thus iteratively moving over the entire input, $\mathbf{I}$ with size $h \times w \times c$. Mathematically we express the forward pass with the convolution operator $*$, and it can then be written in terms of an element of the output as

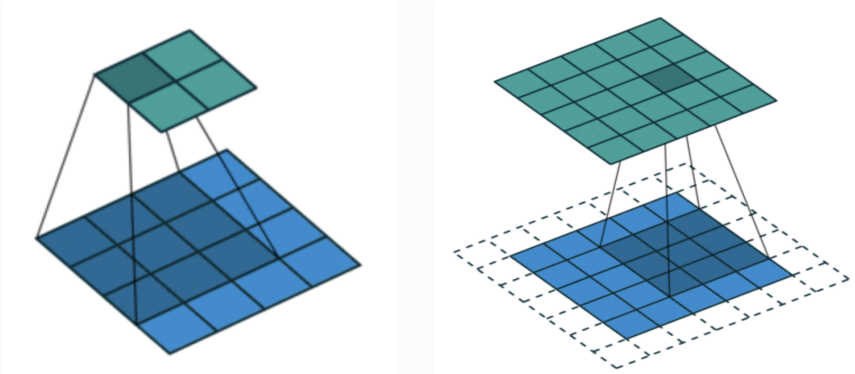
$$(\mathbf{F} * \mathbf{I})_{ijk} = \sum_{f=-n'}^{n'} \sum_{g=-m'}^{m'} \sum_{h=1}^{c} I_{i+f,\, j+g,\, h} \cdot F_{fghk}. \tag{3.44}$$

We iterate over the primed filter dimensions $n' = \lfloor \frac{n}{2} \rfloor$ and $m' = \lfloor \frac{m}{2} \rfloor$ in place of the non-primed dimensions to correctly align the input with the center element of the kernels. For this reason n and m are usually chosen to be odd integers. Equation 3.44 is illustrated for $c = 1$ in figure 3.4

The convolution is computed over the entire depth of the input, i.e. along the channels of the image. Thus the kernel maintains a $n \times m$ matrix of weights for each layer of depth in the previous layer. For a square kernel of size K that moves one pixel from left to right per step over a square image of size W the output is then a square matrix with size O, i.e.

$$O = W - K + 1. \tag{3.45}$$

In practice, it is often beneficial to pad the image with one or more columns/rows of zeros such that the kernel fully covers the input. Additionally, one can down-sample by moving the kernel more than one pixel at a time. This is called the stride of the layer and has a very visually intuitive representation that is illustrated in figure 3.4. The full computation of the down-sizing with these effects then is a modified version of equation 3.45, namely:



(a) **(b)**

Figure 3.4: Two examples of a convolutional layers forward pass, which is entirely analogous to equation 3.7 for fully connected layers. The convolutional layer maintains a N kernels, or filters, that slides over the input taking the dot product for each step, this is the convolution of the kernel with the input. In **(a)** a 3×3 kernel is at the first position of the input and produces one floating point output for the 9 pixels it sees. The kernel is a matrix of weights that are updated with backpropagation of errors. An obvious problem with **(a)** is that the kernel center cannot be placed at the edges of the image, we solve this by padding the image with zeros along the outer edges. This zero-padding is illustrated in **b** where zeros are illustrated by the dashed lines surrounding the image. The kernel then convolves over the whole image including the zeroed regions thus loosing less information. Figure copied from Dumoulin and Visin [30]

$$O = \frac{W - K + 2P}{S} + 1. \tag{3.46}$$

The modification includes the addition of an additive term from the padding, P, and a division by the stride (i.e., how many pixels the kernel jumps each step), S. Striding provides a convenient way to down-sample the input, which lessens the memory needed to train the model. Traditionally MaxPooling has also been used to achieve the same result. MaxPooling is a naive down-sampling algorithm that selects the highest value from the set of disjoint $m \times m$ patches of the input, where m is the pooling number. In practice, $m = 2$ has been the most common value for MaxPooling as it results in a halving of the input in both width and height.

Originally proposed by LeCun et al. [31] convolutional layers were used as feature extractors, i.e., to recognize and extract parts of images that could be fed to ordinary fully connected layers. The use of convolutional layers remained in partial obscurity for largely computational reasons until the rise to preeminence when Alex Krizhevsky et al. won a major image recognition contest in 2012

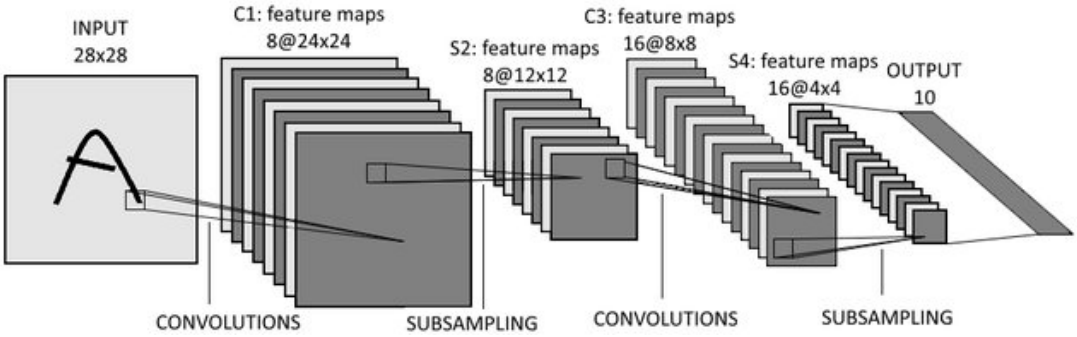


Figure 3.5: The architecture LeCun et al. [31] used when introducing convolutional layers. Each convolutional layer maintains N kernels with initially randomized weights. These N filters act on the same input but will extract different features from the input owing to their random initialization. The output from a convolutional layer then has size determined by equation 3.46 multiplied by the number of filters N. Every t-th layer will down-sample the input, usually by a factor two. Either by striding the convolution or by MaxPooling.

[27] using connected GPUs (graphics processing unit). A GPU is a specialized device constructed to write data to display to a computer screen, which involves large matrix-multiplications. This property is what Krizhevsky et al. used to achieve the entrance of convolutional networks truly to the main-stage of machine learning.

Since then, there have been major revolutions in architecture for the state-of-the-art. Inception modules showed that combinations of filters are functionally the same as ones with larger kernels, yet maintain fewer parameters [32]. Residual networks used skip connections, passing the original data forward to avoid vanishing gradients, and batch normalization (discussed in section 3.2). In this thesis, however, the number of classes and amount of data is still far less complex than the cases where these improvements have really shown their worth[1].

A small digression on GPUs

Usually, these devices are used in expensive video operations such as those required for visual effects and video games. They are specialized in processing large matrix operations which is exactly the kind of computational resource neural networks profits from. The major bottle-neck they had to solve was the problem of memory; at the time a GPU only had about $3GB$ of memory. They were however well equipped to communicate without writing to the central system memory, so the authors ended up implementing an impressive load-sharing paradigm [27]. Modern consumer-grade GPUs have up to $10GB$ of memory

[1]The AT-TPC produces data on the order of 10^5 samples and 10^0 classes while inception-net and residual nets deal with datasets on the order of millions of samples and 10^3 classes

and have more advanced sharing protocols further cementing them as ubiquitous in machine learning research. In this thesis, all models were run on high-end consumer-grade GPUs hosted by the AI-HUB computational resource at UIO.

3.3 Recurrent Neural Networks

The recurrent neural network (RNN) models a unit that has "memory". The memory is encoded as a state variable, which is ordinarily concatenated with the input to make a prediction. The model predictions typically enact a sequence which has led to applications text generation, time series predictions, and other serialized applications. RNNs were first discussed in a theoretical paper by Jordan, MI in 86' but implemented in the modern temporal sense by Pearlmutter [33]. A simple graphical representation of the RNN cell is presented in figure 3.6

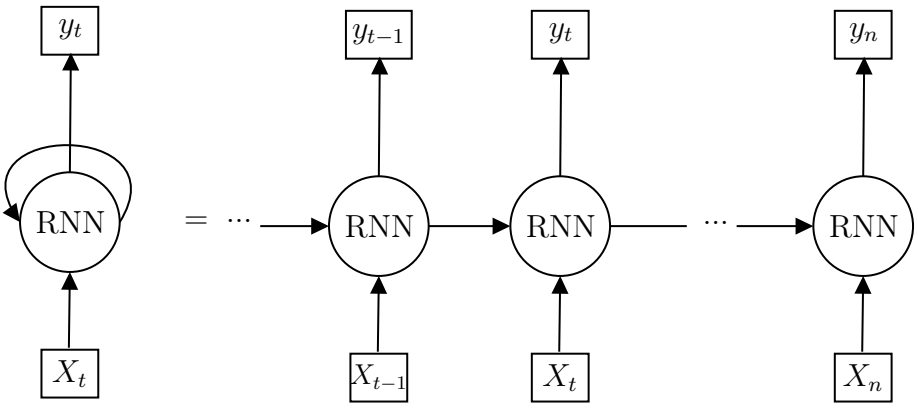


Figure 3.6: A graphical illustration of the RNN cell. The self-connected edge in the left hand side denotes the temporal nature we unroll on the right side. The cell takes as input a state vector and an input vector at time t, and outputs a prediction and the new state vector used for the next prediction. Internally the simplest form this operation takes is to concatenate the state vector with the input and use an ordinary dense network as described in section 3.1 trained with back-propagation.

The memory encoded by the RNN cell is encoded as a state variable. Figure 3.6 gives a good intuition for a RNN cell, but we will elaborate on this by introducing the surprisingly simple forward pass structure for ordinary RNN cells. Let X_t be the input to the RNN cell at time-step from zero to n, $\{0 \leq t \leq n : t \in \mathcal{Z}\}$ and h_t be the state of the RNN cell at time t. Let also y_t be the output of the RNN at time t. The nature of X and y are problem specific but a common use of these network has been the prediction of words in a sentence, such that X is a representation of the previous word in the sentence and y the prediction over the set of available words for which comes next. Our cell can then be simply formulated as in equation 3.47.

$$\langle [X_t, h_t] | W \rangle + b = h_{t+1} \tag{3.47}$$

Where the weight matrix W and bias vector b are defined in the usual manner. Looking back at figure 3.6 the output should be a vector in y space and yet we've noted the output as being in the state space of the cell. This is simply a convenience lending flexibility to our implementation, the new state is produced by the cell and transformed to the y space by use of a normal linear fully connected layer. This is a common trick in machine learning: leaving the inner parts of the algorithm extremely problem agnostic and using end-point layers to fit the problem at hand. To further clarify, we show the forward pass for a simple one-cell RNN in algorithm 3. The forward pass is remarkably simple and flexible all the same. To model complex systems one can use the output from one RNN cell, h_{t+1}, as the input to another RNN cell that maintains its own state.

Algorithm 3: Defining the forward pass of a simple one cell RNN network. The cell accepts the previous state and corresponding data-point as input. These are batched vectors both, and so one usually concatenates the vectors along the feature axis to save time when doing the matrix multiplication. The cell maintains a weight matrix, $\mathbf{W}$, and bias, b, which will be updated by back-propagation of errors in the standard way.

Result: $\mathbf{h}_{t+1}$
Input: $\mathbf{h}_t$, $\mathbf{X}_t$
Data: $\mathbf{W}$, b
$\mathbf{F} \leftarrow$ **concatenate**((h_t, $\mathbf{X}_t$), axis= 1);
$\mathbf{h}_{t+1} \leftarrow$ **matmul**($\mathbf{F}$, $\mathbf{W}$) + b;
return $\mathbf{h}_{t+1}$

Recurrent architectures present the researcher with a set of tools to not only model sequences, but to use a sequential structure to avoid common problems with e.g. variational autoencoders. Gregor et al. [24] uses this aspect of recurrent networks in their deep recurrent attentive writer (DRAW) algorithm, which sequentially draws on a canvas to create realistic looking output images. In a non-sequential autoencoder we encounter the challenge that a given pixels activation is not conditioned on it's neighbors activation. In part this problem is what gives rise to the blurriness observed in the output from variational autoencoders. When designing a neural network the researcher principally has five basic choices regarding the sequentiality of the model. We represent these five in figure 3.7, which is copied from Karpathy [34].

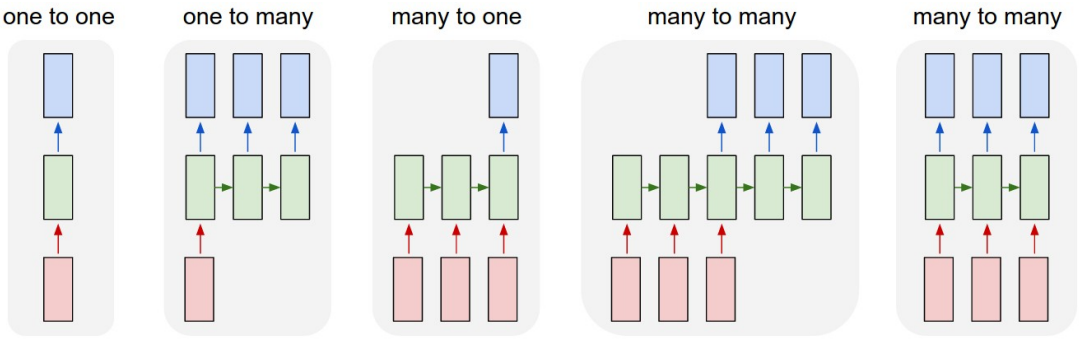


Figure 3.7: The advent of recurrent networks enabled machine learning researchers to both model complex sequential behaviors like understanding patterns in text as well as using sequences to predict a single multivariate outcome and more. The leftmost figure **1)** represents an ordinary neural network, where the rectangles are matrix objects, red for input, blue for output and green for intermediary representations and the arrows are matrix operations like concatenation multiplication etc. **2)** shows a recurrent architecture for sequence output e.g. image captioning. Where the information about the previous word gets passed along forward by the state of the previous cell. **3)** transforming a sequence of observations to a single multivariate outcome. The classical example of which is sentiment prediction from text. **4), 5)** Sequenced input and output can either be aligned as in the latter or misaligned as in the former. An example of synced sequence to sequence can be phase prediction from a time series of a thermodynamic system. Un-synced applications include machine translation where sentences are processed then output in another language. Figure copied from [34]

3.3.1 Long short-term memory cells

One of the principal challenges for the RNN cell is in the handling of the state of the cell. The state acts as a sort of memory, but without moderation it tries to recall the entire history of the sequence. A solution to this problem was proposed by [35] in their description of the long short-term memory (LSTM) network. The LSTM network introduced the ability for the network to selectively forget part of its memory. This was a boon to the language processing community, as they regularly tackle problems with both long and short term relationships.

The LSTM network implements a series of layers inside each cell. By combining sigmoid, $\sigma\cdot$, and the hyperbolic tangent function, $\tau(\cdot)$, they compute pointwise manipulations of the state $\mathbf{h}_t$, previous output $\mathbf{c}_t$ and input $\mathbf{x}_t$. Lastly, we define the input gate $\mathbf{i}_t$, output gate $\mathbf{o}_t$ and forget gate $\mathbf{f}_t$, each with their own weight matrices $\mathbf{W}_{\{i,f,o,c\}}$. The gates are then computed as

$$\mathbf{i_t} = \sigma(\mathbf{W}_i[\mathbf{h}_{t-1}, \mathbf{x}_t]), \tag{3.48}$$
$$\mathbf{f_t} = \sigma(\mathbf{W}_f[\mathbf{h}_{t-1}, \mathbf{x}_t]), \tag{3.49}$$
$$\mathbf{o_t} = \sigma(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t]). \tag{3.50}$$

Following from these definitions, we can compute the new state and cell outputs

$$\hat{\mathbf{c}}_t = \tau(\mathbf{W}_o[\mathbf{h}_{t-1}, \mathbf{x}_t]), \tag{3.51}$$
$$\mathbf{c}_t = \mathbf{i}_t \circ \hat{\mathbf{c}}_t + \mathbf{f}_t \circ \mathbf{c}_{t-1}, \tag{3.52}$$
$$\mathbf{h}_t = \mathbf{o}_t \circ \tau(\mathbf{c}_t). \tag{3.53}$$

Recall the Hadamard product denoted by $\circ$, and the concatenation of two vectors denoted by square brackets. In the equations above we have also followed the convention of including a column of ones in the data, which bakes the biases into the weight matrices.

The LSTM network was designed to allow for the network to forget parts of the input. However, it also solves a vanishing gradient problem as the gradient flows through the network by the cell outputs.

Chapter 4

Autoencoders

In the previous chapters we discussed the components which are commonly used to make neural network models. These include dense, convolutional and recurrent layers, as well as different types of activation functions and regularization techniques. In this chapter we will discuss the autoencoder family of deep learning algorithms. All of the algorithms discussed in this chapter are constructed using the elements from the previous chapter. But we frame them in a very general way such that we are free to implement the algorithms using convolutions, dense layers, or a combination of those depending on the problem at hand.

As mentioned in the beginning of the previous chapter this thesis explores the feasibility of using latent spaces to model nuclear physics events. In the context of this thesis a latent space is a representation of the data when passed through a set of trained neural network layers. One way of training a latent space is by enacting a compression of the data and using this to reconstruct the original input; this is the essence of the autoencoder algorithms.

4.1 Introduction to autoencoders

The primary challenge we face with data from the active target time projection chamber (AT-TPC) is that labelling data is not always possible for a given experiment. Given that fact, and the considerable amount of data available, we turn to dimensionality reduction methods to possibly express the different physics occurring in this data. One such set of methods is the autoencoder family of neural network algorithms.

An autoencoder is an attempt at learning a distribution of data by reconstruction. This means that the model encodes the data into a much lower dimensional latent space before projecting back into the original space of data. The goal, as a matter of course, is then to learn the true distribution, $P(\mathcal{X})$, over the data with some parametrized model $Q(\mathcal{X};\theta)$. The model consists of two discrete parts: an encoder and a decoder. The encoder is, in general, a nonlinear map ψ

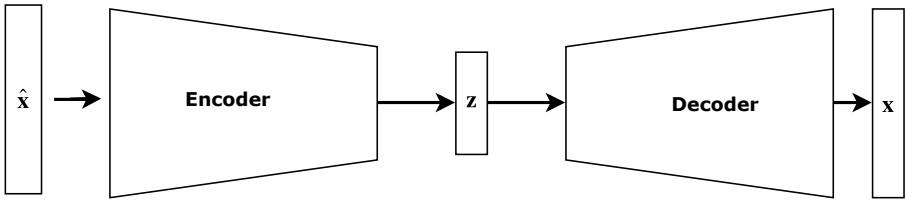


Figure 4.1: Schematic of an autoencoder model. A sample $\hat{\mathbf{x}}$ is compressed to a lower-dimensional representation $\mathbf{z}$, which is subsequently reconstructed to the original space by the decoder. The tapered shape of trapezoid which represents the networks indicates if the network is compressing or uncompressing a representation.

$$\psi : \mathcal{X} \to \mathcal{Z}, \tag{4.1}$$

where $\mathcal{X}$ and $\mathcal{Z}$ are spaces with $\dim(\mathcal{X}) > \dim(\mathcal{Z})$. The second part of the model is the decoder that maps back to the original space

$$\phi : \mathcal{Z} \to \mathcal{X}. \tag{4.2}$$

An illustration of a general autoencoder architecture is included in figure 4.1.

The objective is then to find the configuration of the two maps ϕ and ψ which gives the best possible reconstruction. That is the objective $\mathcal{O}$ for the model is given as

$$\mathcal{O} = \arg\min_{\phi,\psi} \sum_i ||\mathbf{x}_i - \phi(\psi(\mathbf{x}_i))||^2 \tag{4.3}$$

As the name implies the encoder creates a lower-dimensional "encoded" representation of the input. A mean-squared-error cost optimizes this objective function in the event of real-valued data, but just as commonly through a binary cross-entropy for data normalized to the range $[0, 1]$. This representation can be useful for identifying whatever information-carrying variations are present in the data. This can be thought of as an analogue to principal component analysis (PCA)[19] which we introduced in chapter 2. In practice, the non-linear maps, ψ and ϕ, are most often parametrized by neural networks. We refer to section 3.1 for a detailed discussion on deep learning and neural networks.

The autoencoder has previously been successfully implemented in de-noising tasks. More recently, the Machine Learning community discovered that one could impose a constraining term on the latent distribution to allow for the imposition of structure in the latent space. The goal in mind was to create a generative

algorithm, a class of models used to sample from the distribution $P(\mathcal{X})$. For example, to generate bespoke images of houses.

The first of these employed a Kullback-Leibler divergence to impose structure on the latent space. These models were dubbed variational autoencoders or VAEs by Kingma and Welling [36]. While VAEs lost the contest for preeminence as a generative algorithm to adversarial[1] networks, they remain a fixture in the literature of expressive latent variable models with development focusing on the salience of the latent space ([37], [38], [21]).

4.2 The Variational Autoencoder

Originally presented by Kingma and Welling [36] the VAE is a twist upon the traditional autoencoder. Where the applications of an ordinary autoencoder largely extended to de-noising, with some authors using it for dimensionality reduction, the VAE seeks to control the latent space of the model. The goal of the VAE is then to be able to generate samples from the unknown distribution over the data. In this thesis, the generative properties of the algorithm are only interesting as a metric to describe the latent space. Our efforts mainly concentrate on the latent space itself. Specifically, the focus is discerning whether class membership, be it a physical property or something more abstract [2] is encoded.

We begin this chapter with the discussion of the VAE as the framework for deriving the cost, and the methodology underpins the formalism on how we view the regularization of latent spaces.

4.2.1 The variational autoencoder cost

In section 4.1 we introduced the structure of the autoencoder. For the VAE, which is a more integral part of the technology used in the thesis, a more rigorous approach is warranted. We will here derive the loss function for the VAE in such a way that clarifies how we aim to impose a known structure of the latent space.

We begin by considering the family of problems encountered in variational inference, where the VAE takes its theoretical inspiration. Specifically, we will derive the VAE as a solution to a Bayesian variational inference problem. Bayesian inference is a framework for linking some observed values x to some hypothesized latent variable z.

This family of techniques all begin with a consideration of the problem from Bayes' rule, which relates the probability of seeing our model given our data,

[1] Adversarial networks are a pair of networks where one aims to generate realistic samples, and the other aims to separate between fake and real samples

[2] examples include discerning whether a particle is a proton or electron, or capturing the "five-ness" of a number in the handwritten digits MNIST dataset

$p(z|x)$ to the odds ratio of seeing our model $p(x|z)p(z)$ to the evidence $p(x)$. Bayes' rule can then be expressed mathematically as

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}. \tag{4.4}$$

The left-hand side is termed the posterior distribution, which describes the updated knowledge given some prior belief expressed by the probability of the model and the likelihood which we know from chapter 2. The last part of this equation is the denominator which is commonly referred to as the evidence. It is also what causes the problem to be challenging. To see why we introduce some common re-writing tools used in statistical analysis. Beginning with the evidence, which can be expressed as the integral of the joint distribution $p(x, z) = p(x|z)p(z)$ such that

$$p(x) = \int_z p(x|z)p(z) \tag{4.5}$$

The integral in the denominator of equation 4.4 is intractable for most interesting problems, as the space over z is usually combinatorially large.

Another standard tool for solving this problem is Markov chain Monte Carlo (MCMC) methods. In physics, this family of algorithms has been applied to solve many-body problems in quantum mechanics primarily by gradient descent on variational parameters [39].

We can then summarize variational Bayesian methods as being techniques for estimating computationally intractable integrals as the one expressed in 4.5. To derive a solution, we begin by introducing the KL-divergence [40], which is a measure of how much two distributions are alike. It is important to note that it is, however, not a metric. We define the KL-divergence in equation 4.6 from a probability measure P to another Q, by their probability density functions p, q over the set $x \in \mathcal{X}$

$$D_{KL}(P||Q) = -\int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx, \tag{4.6}$$

$$= \langle \log\left(\frac{p(x)}{q(x)}\right) \rangle_p. \tag{4.7}$$

In the context of the VAE, the KL-divergence is a measure of the quality of P approximating Q [41]. The first part of deriving the cost is then to introduce an approximation of the evidence.

We introduce the evidence lower bound (ELBO) as an approximation to the evidence following the derivation laid out by [36]. The EBLO function is defined as

$$ELBO(q) = \langle \log(p(z,x)) \rangle - \langle \log(q(z|x)) \rangle. \tag{4.8}$$

To fit the VAE cost we rewrite the ELBO in terms of the conditional distribution of x given z

$$\begin{aligned} ELBO(q) &= \langle \log(p(z)) \rangle + \langle \log(p(x|z)) \rangle - \langle \log(q(z|x)) \rangle, && (4.9) \\ &= \langle \log(p(x|z)) \rangle - D_{KL}(q(z|x)|p(z)) && (4.10) \end{aligned}$$

Finally, the ELBO can be written as a lower bound on the evidence using Jensen's inequality (J) for concave functions. Mathematically we express the inequality as

$$f(\langle x \rangle) \geq \langle f(x) \rangle. \tag{4.11}$$

From the definition of the evidence we then have

$$\begin{aligned} \log(p(x)) &= \log \int_z p(x|z)p(z), && (4.12) \\ &= log \int_z p(x|z)p(z)\frac{\psi(z|x)}{\psi(z|x)}, && (4.13) \\ &= log \langle p(x|z)p(z)/\psi(z|x) \rangle, && (4.14) \\ &\overset{(J)}{\geq} \langle \log(p(x|z)p(z)/\psi(z|x)) \rangle, && (4.15) \\ &= \langle \log(p(x|z)) \rangle + \langle \log(p(z)) \rangle - \langle \log(\psi(z|x)) \rangle, && (4.16) \\ \log(p(x)) &\geq \langle \log(p(x|z)) \rangle - D_{KL}(\psi(z|x)||p(z)). && (4.17) \end{aligned}$$

Showing that the ELBO is indeed a lower bound on the log evidence.

We can then move on to the VAE cost. We begin by defining $q(z|x)$ to be the posterior distribution over the latent variable $z \in \mathcal{Z}$, conditional on our data $x \in \mathcal{X}$ with evidence $p(x)$ and latent prior $q(z)$, with an associated probability measure Q as per our notation above. Let then the parametrized distribution over the latent space enacted by the encoder be given as $\psi(z|x)$, and an associated probability measure Ψ. The quality of our encoder can then be decomposed as

$$\begin{aligned} D_{KL}(\Psi||Q) &= \langle \log \left(\frac{\psi(z|x)}{q(z|x)} \right) \rangle_z, && (4.18) \\ &= \langle \log \left(\psi(z|x) - \log q(z|x) \right) \rangle_z. && (4.19) \end{aligned}$$

From Bayes' rule, we can re-state the posterior by introducing the decoder $\phi(x|z)$. Continuing the derivation we then have

$$D_{KL}(\Psi||Q) = \langle \log(\psi(z|x)) - \log(\phi(x|z)q(z)) + \log(q(x)) \rangle_z, \tag{4.20}$$

We identify that the evidence can be taken outside the expectation. Re-arranging the terms separates the model from the optimization targets

$$D_{KL}(\Psi||Q) - \log q(x) = \langle \log \psi(z|x) - \log q(z) - \log(\phi(x|z)) \rangle_z, \tag{4.21}$$

Note that the term $-\langle \log(\phi(x|z)) \rangle_z$ is the negative log-likelihood of our decoder network which we can optimize with the cross-entropy as discussed in section 2.8. We also identify that we can re-write the right side to include another KL divergence between the latent prior and the encoder. Flipping the sign then gives us the VAE cost

$$\log(p(x)) - D_{KL}(\Psi||Q) = \langle \log(\phi(x|z)) \rangle_z - D_{KL}(\psi(z|x)||q(z)). \tag{4.22}$$

We are still bound by the intractable integral defining the evidence $p(x) = \int_z p(x,z)$ which is the same integral as in the denominator in equation 4.4. This problem is solved by recognizing that the right-hand side is the ELBO and so we conclude that the VAE fits on the lower bound of the evidence.

Kingma and Welling [36] showed that this variational lower bound on the marginal likelihood of our data is feasibly approximated with a neural network when trained with backpropagation and gradient descent methods. That is, we estimate the derivative of the ELBO with respect to the neural network parameters, as described by the backpropagation algorithm in section 3.1.1 and iteratively improve the neural network with a gradient descent algorithm.

4.3 Optimizing the variational autoencoder

From equation 4.22, we observe that the optimization is split in two. The first is a reconstruction term that approximates the log evidence, which we can train with a squared error or cross-entropy cost. Secondly, we have a divergence term over the parametrized and theoretical latent distribution. We want to simplify the second to conserve computational resources. Thankfully this is computationally cheap given a specific prior latent distribution. Let the target distribution $p(z|x)$ be a multivariate normal distribution with zero means and a diagonal unit variance matrix, i.e. $p(z|x) \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. And accordingly the neural network approximation then follows $\psi(z|x) \sim \mathcal{N}(\mu, \Sigma)$. The normalized probability density function for the normal Gaussian is defined as

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} exp(-\frac{1}{2}(\mathbf{x}-\mu)^T\Sigma^{-1}(\mathbf{x}-\mu)), \tag{4.23}$$

and the Kullback-Leibler divergence for two multivariate Gaussians is given by

$$D_{KL}(p_1||p_2) = \frac{1}{2}\left(\log\frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2 - \mu_1)^T\Sigma_2^{-1}(\mu_2 - \mu_1)\right), \quad (4.24)$$

which we derive in appendix A. Substituting p_1 and p_2 with the model distribution $\psi(z|x)$ and a latent prior $p(\mathbf{z}) \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ we get

$$\begin{aligned} D_{KL}(q||p) &= \frac{1}{2}\left(-\log|\Sigma_1| - n + tr(I\Sigma_1) + \mu_2^T I\mu_2\right) \\ &= \frac{1}{2}\left(-\log|\Sigma_1| - n + tr(\Sigma_1) + \mu_2^T\mu_2\right), \end{aligned}$$

or more conveniently

$$D_{KL}(q||p) = \frac{1}{2}\sum_i -\log\sigma_i^2 - 1 + \sigma_i^2 + \mu_i^2. \quad (4.25)$$

We note that equation 4.25 satisfies the equality that the divergence is zero when the target and model distributions are equal. The VAE model can then be enacted with a pair of neural networks acting as an encoder and decoder. To parametrize the sample, we compute a mean vector μ and a variance vector σ^2 from the encoded representation. The latent sample is then given as

$$\mathbf{z} = \mu + \sigma^2 \cdot \epsilon, \quad (4.26)$$

where ϵ is a stochastic vector drawn from $\mathcal{N}(\mathbf{0}, \mathbf{1})$. A schematic of a VAE model is shown in figure 4.2.

An important feature of the Kullback-Leibler divergence is that it operates point-wise on the probability densities. This was the point of contention in the paper by Zhao et al. [38]. In the paper, the authors propose alternate measures for regularizing the latent space. The intuitive alternative to a point-wise measurement is comparing the moments of the distribution and minimize their difference.

4.3.1 Mode collapse

When training a VAE, we are balancing the evidence and prior latent distribution loss terms. Kingma and Welling [36] note that equation 4.22 is unbalanced in favor of the latent space regularization. As a consequence, the un-weighted loss suffers from mode collapse. When the reconstruction is under-valued in the optimization, the model will quickly learn the prior, without encoding the input. As the prior has become uninformative, the decoder proceeds to learn a rough representation. This representation usually covers as many of the outputs as possible. Mode collapse can then often be visually identified by the output being a vague cloud that looks like an average of all the data.

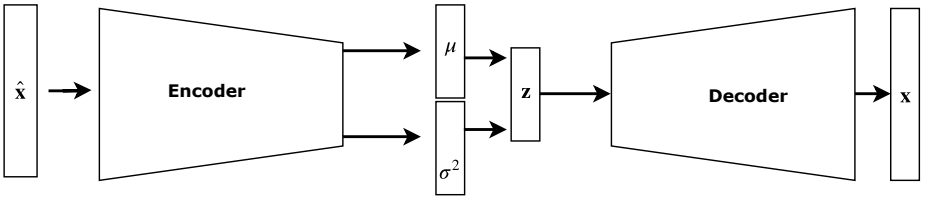


Figure 4.2: Schematic of a variational autoencoder model. A sample $\hat{\mathbf{x}}$ is compressed to a lower-dimensional representation $\mathbf{z}$, which is parametrized by a mean vector, μ, and a variance vector σ^2. Subsequently, we reconstruct the original input with the decoder.

4.4 Regularizing Latent Spaces

As introduced in section 4.2 the latent space of an autoencoder can be regularized to satisfy some distribution. The nature and objective of this regularization has been the subject of debate in machine learning literature since Kingma's original VAE paper in 2014. Two of the seminal papers published on the topic is the $\beta - VAE$ paper by Higgins et al. [37] introducing a Lagrange multiplier to the traditional KL divergence term, and the Info-VAE paper by Zhao et al. [38] criticizing the choice of a KL-divergence on the latent space. Where they further build on the $\beta - VAE$ proposition that the reconstruction and latent losses are not well balanced, and show that one can replace the KL-divergence term with another strict divergence and empirically show better results with these. In particular they show strong performance with a Maximum-Mean Discrepancy (MMD) divergence, which fits the moments of the latent distribution instead of measuring a point-wise divergence. By using any positive definite kernel $k(\cdot, \cdot)$[3] we define the MMD divergence as

$$D_{MMD} = \langle k(z, z') \rangle_{p(z),p(z')} - 2\langle k(z, z') \rangle_{q(z),p(z')} + \langle k(z, z') \rangle_{q(z),q(z')}. \tag{4.27}$$

Which does not have a convenient closed form like the the Kullback-Leibler divergence and so adds some computational complexity. In this thesis we use mixture of Gaussian distributions of the prior $p(z)$ to encourage class separability in the latent space.

Recent research by Seybold et al. [42], amongst others, points to the challenge of the model collapsing to an autodecoder. In other words a sufficiently complex decoder can learn to reconstruct the sample independent of the latent sample [42]. To combat this problem they introduce a change in the optimization objective by adding a second decoder term to the optimization task

[3]We will not probe deeply into the mathematics of kernel functions but they are used in machine learning most often for measuring distances, or applications in finding nearest neighbors. They are ordinarily proper metric functions. Some examples include the linear kernel: $k(x, x') = x^T x'$ or the popular radial basis function kernel $k(x, x) = e^{-\frac{||x-x'||^2}{2\sigma}}$

$$\langle \phi(x|z) + \lambda \phi'(x'|z) \rangle + \beta D(p||\psi). \tag{4.28}$$

The second decoder term reconstructs a different representation of the sample x, and the change is dubbed as a duelling decoder model. In this work we will consider a duelling decoder that reconstructs a 2D charge projection reconstruction, which is the ordinary decoder, and a decoder that reconstructs a charge distribution or the measured net charge. As reactions happen and the charged particles move through the gas in the AT-TPC the amount of gas ionized varies and as such we expect that this second reconstruction objective will improve the amount of semantic information in the latent expressions.

4.5 Deep Recurrent Attentive Writer

One of the central challenges of the ELBO as presented in equation 4.8 is that the probability of a pixel in the output being activated is not conditional on whether the pixels surrounding it has is activated. The deep recurrent attentive writer (DRAW) aims to solve this problem by creating an iterative algorithm which updates the canvass multiple times [24]. In this thesis, we make three central modifications to the algorithm.

- Originally DRAW views parts of the input conditioning the latent sample $\mathbf{z_t}$ on differently sized patches of the input image. We modify the model such that the model gets glimpses of the same size at each time step. This is done to make samples comparable between time steps in line with the work of Harris et al. [43]

- The attentive part of DRAW as described by Gregor et al. [24] is a set of Gaussian filters that pick out a part of the input allowing the image to focus on discrete regions. We modify the algorithm to allow the use of a convolutional network as a feature extractor.

- Latent samples from DRAW are originally described in the framework of the VAE where the latent sample is drawn from a normal distribution i.e. $\mathbf{z}_t \sim \mathcal{N}(Z_t|\mu_t, \sigma_t)$. Since then proposals have been made for autoencoders that do not require this stochasticity in the forward-pass and as such the latent samples can be generated from fully connected layers, e.g. the InfoVae architecture proposed by Zhao et al. [38]

At the core of the DRAW algorithm sits an encoder- and a decoder network, making it part of the autoencoder sub-family of neural networks. A pair of recurrent LSTM cells then enact this familiar framework. We use the same notation as Gregor et al. [24] and denote the encoder with with RNN^{enc} whose output at

time t is $\mathbf{h}_t^{enc}$, and the decoder with RNN^{dec}. The form of the encoder/decoder pair is determined by the read/write functions which we discuss in the next section. Next the encoder output, $\mathbf{h}_t^{enc}$, is used to draw a latent sample, $\mathbf{z}_t$, using a function latent$(\cdot)$ which is determined by the form of the latent loss. At each time-step, the algorithm produces a sketched version of the input c_t, which is used to compute an error image, $\hat{\mathbf{x}}_t$, that feeds back into the network. The following equations from Gregor et al. [24] summarizes the DRAW forward pass

$$\hat{\mathbf{x}} = \mathbf{x} - \sigma(\mathbf{c_{t-1}}), \tag{4.29}$$
$$\mathbf{r}_t = \text{read}(\mathbf{x}_t, \hat{\mathbf{x}}_t), \tag{4.30}$$
$$\mathbf{h}_t^{enc} = \text{RNN}^{enc}(\mathbf{h}_{t-1}^{enc}, [\mathbf{r}_t, \mathbf{h}_{t-1}^{dec}]), \tag{4.31}$$
$$\mathbf{z}_t = \text{latent}(\mathbf{h}_t^{enc}), \tag{4.32}$$
$$\mathbf{h}_t^{dec} = \text{RNN}^{dec}(\mathbf{h}_{t-1}^{dec}, \mathbf{z}_t), \tag{4.33}$$
$$\mathbf{c}_t = \mathbf{c}_{t-1} + \text{write}(\mathbf{h}_t^{dec}), \tag{4.34}$$

where $\sigma(\cdot)$ denotes the logistic sigmoid function. The iteration then consists of an updating canvass $\mathbf{c}_t$ which informs the next time-step. We outline the architecture in figure 4.3.

4.5.1 Read and Write functions

The read/write functions are paired processing functions that create a subsampled representation of the input. The trivial read function is a concatenation of the error image with the input. In a similar vein, the trivial write function is a weight transformation from the decoder output to the input dimension. This pair of functions were not considered for this work, outside testing of the algorithm.

Instead of the trivial implementations, the DRAW authors implement grids of Gaussian filters to extract patches of smoothly varying location and size [24]. To control the patch the authors compute centres, g_X and g_Y, and stride, which controls the size, of a $N \times N$ patch of Gaussian filters over the $H \times W$ input image. These filters are collected in matrices F_x and F_y that we use to extract a part of the image with the read function and project to the input space with the write function. The mean location of those filters are computed from the centres, g_X and g_Y, and the stride δ. From Gregor et al. [24] the means at row i and column j are defined as

$$\mu_X^i = g_X + (i - N/2 - 0.5)\delta, \tag{4.35}$$
$$\mu_Y^j = g_Y + (j - N/2 - 0.5)\delta. \tag{4.36}$$

The attention parameters are computed from a fully connected layer connecting the decoder state to a 4-tuple of floating point numbers i.e

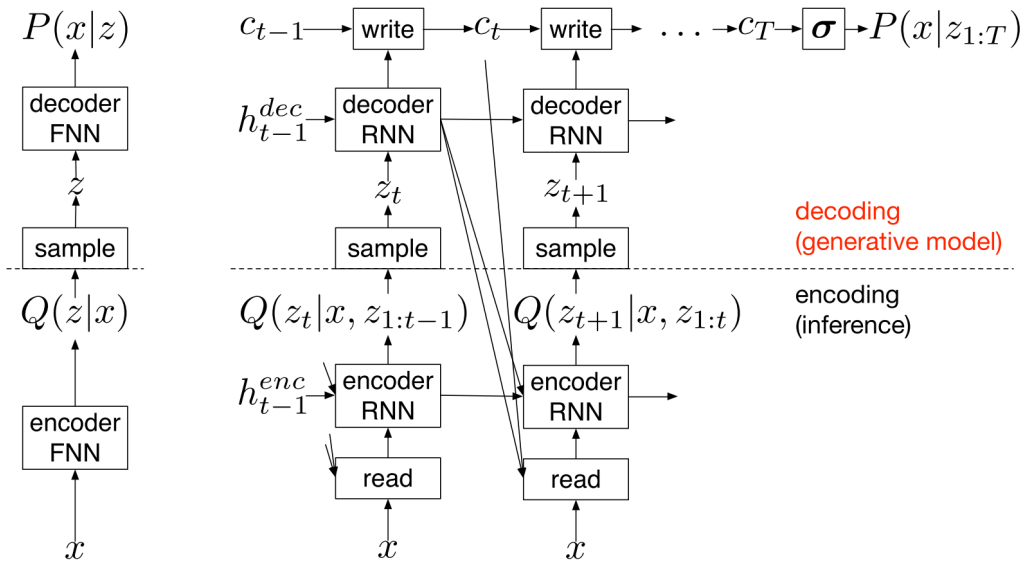


Figure 4.3: Left: an ordinary oneshot encoderdecoder network. **Right:** DRAW network that iteratively constructs the canvas using RNN cells as the encoder/decoder pair. The final output is then iteratively constructed using a series of updates on a canvass, c_t. DRAW function read that process the input and feeds this to the encoder which outputs a latent sample $\mathbf{z}_t$. The latent sample, in turn, acts as input to the decoder part of the network which modifies the canvass using a write function that mirrors the read operation.

$$\tilde{g}_x,\ \tilde{g}_y,\ \log\sigma^2,\ \log\gamma = \text{Dense}(\mathbf{h}_t^{dec}), \tag{4.37}$$

where σ^2 is the isotropic variance of the Gaussian filters, and γ the multiplicative intensity of the filtering. We parametrize the σ^2 and γ variables as being log-transformed to ensure positivity by exponentiation prior to use. Gregor et al. makes an additional transformation on the centres to ensure that the initial patch roughly covers the entire input image. The transformation is made with respect to the input width, W, and height, H, giving

$$g_x = \frac{W+1}{2}(\tilde{g}_x + 1), \tag{4.38}$$

$$g_y = \frac{H+1}{2}(\tilde{g}_y + 1). \tag{4.39}$$

$$\tag{4.40}$$

The above equations included terms to compute and scale δ, which we instead elect to estimate as a constant hyperparameter. The combination between the

number of filters N and δ determines the size of the input region passed to the encoder. By forcing these glimpses to be equally sized, we hypothesize will ensure the comparability of latent samples. Setting δ as a hyper-parameter was inspired by the work of Harris et al. [43].

Given the scaled center we can then compute the filter-banks $F_x \in \mathbb{R}^{N \times W}$ and $F_y \in \mathbb{R}^{N \times H}$

$$F_x[i, w] = \frac{1}{Z_x} e^{-\frac{(w - \mu_x^i)^2}{2\sigma^2}}, \tag{4.41}$$

$$F_y[j, h] = \frac{1}{Z_y} e^{-\frac{(h - \mu_y^j)^2}{2\sigma^2}}, \tag{4.42}$$

where we denote a point in the input with $(h,\, w)$, and a point in the attention patch with $(i,\, j)$. The filters-banks are multiplied with a normalization constant s.t. $\sum_w F_x[i, w] = 1$, and we define the constant Z_y in the same way.

Finally, we define the read and write functions with attention parameters. The read operation reads a patch from the input and the error image and returns their concatenation to the encoder, and the write function returns an array that adds to the current canvass c_t. From Gregor et al. [24] the read function is defined as

$$\text{read}(\mathbf{x}, \hat{\mathbf{x}}, \mathbf{h}_{t-1}^{dec}) = \gamma[F_y \mathbf{x} F_x^T, F_y \hat{\mathbf{x}} F_x^T]. \tag{4.43}$$

For the write function we compute a new set of attention parameters which we denote as, e.g. $\hat{\gamma}$. Subsequently, we compute a dense layer transform from the current decoder state to a matrix $w_t \in \mathbb{R}^{N \times N}$ to ensure the matrix multiplications are sane. The write function is then defined as

$$\text{write}(w_t) = \hat{\gamma} \hat{F}_y^T w_t \hat{F}_x. \tag{4.44}$$

Notice the transposition order in equation 4.44 is reversed with respect to the order in equation 4.43.

4.5.2 Latent samples and loss

Optimizing the DRAW algorithm is almost entirely analogous to the procedure for the variational autoencoder. We operate with a divergence over our latent samples and a latent prior, as well as a reconstruction term parameterizing the log evidence. In not so many words, we still have a cross-entropy loss over the reconstruction and input as well as a divergence term from our latent samples.

As the DRAW model creates a sequence of latent samples, the considerations for the latent loss changes somewhat. In the DRAW algorithm our encoder parametrizes a distribution $q(\mathbf{z}_t | \mathbf{h}_t^{enc})$ which we want to model as being drawn

from some prior $p(\mathbf{z}_t)$. As with the variational autoencoder, we let the prior be a multivariate isotropic Gaussian. The latent loss, L_z, is then a sum over individual divergence terms for each time-step

$$L_z = \sum_t^T D_{KL}(q||p). \tag{4.45}$$

Given the same prior as for the variational autoencoder we can apply the same derivation of the closed form divergence. Previously we parametrized the latent sample with a mean and standard deviation vector, repeating this procedure the loss becomes

$$L_z = \frac{1}{2}\sum_t^T(\mu_t^2 + \sigma_t^2 - \log\sigma_t^2) - \frac{T}{2}. \tag{4.46}$$

Similarly, the maximum mean discrepancy loss is computed from equation 4.45 replacing the Kullback-Leibler divergence with the terms from equation 4.27.

4.6 Deep Clustering

One of the holy grails of machine learning is achieving a general clustering algorithm, as retrieving labelled samples is often a very costly process. In some cases, labelled data is unavailable altogether. This is the case for some of the AT-TPC experiments, and so discovering clustering algorithms for event data is of some academic interest.

Clustering algorithms based on neural networks are known collectively as deep clustering algorithms. Many of which are based on autoencoder architectures. In this thesis, we will focus on two such algorithms: the deep clustering with convolutional autoencoders (DCEC) algorithm, developed by Guo et al. [44]. Also, the mixture of autoencoders (MIXAE) model, developed by [45].

4.6.1 Deep Clustering With Convolutional Autoencoders

The DCEC architecture is at its core a simple convolutional autoencoder. To convert it to a clustering algorithm, Guo et al. [44] adds a fully connected transformation to a soft class assignment and a loss term for that assignment.

To describe the DCEC we begin by letting the convolutional autoencoder be given in terms of the encoder $\psi(\mathbf{z}|\mathbf{x};\theta_e)$ and decoder $\phi(\mathbf{x}|\mathbf{z};\theta_d)$, where the θ indicates the neural network parameters and $z \in \mathbb{R}^D$. Furthermore let the algorithm maintain K cluster centers $\{\mu_j\}^K$, where $j \in [0, 1, \ldots, N]$ denote the clusters. These cluster centres are trainable parameters which map the latent samples to a soft assignment by a Student's t-distribution, in the model we parametrize them with a matrix μ_{ij}. The assignment is then given as

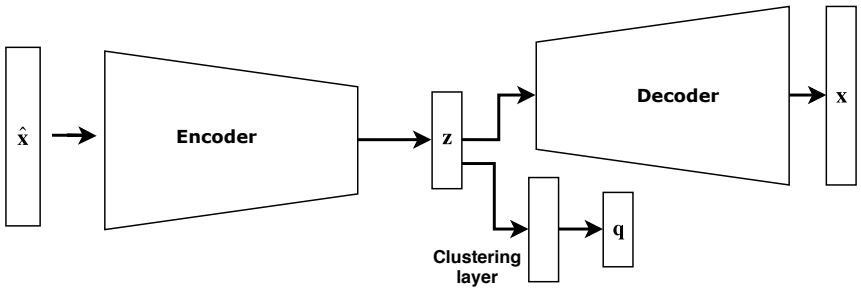


Figure 4.4: Schematic of a DCEC model. A sample $\hat{\mathbf{x}}$ is compressed to a lower-dimensional representation $\mathbf{z}$. The latent sample is then fed both to a decoder which reconstructs the input, but also to a clustering layer which computes the soft assignments $\mathbf{q}$. See the text for further details. Figure adapted from Guo et al. [44]

$$q_{ij} = \frac{(1 + ||\mathbf{z}_i - \mu_j||_2^2)^{-1}}{\sum_j (1 + ||\mathbf{z}_i - \mu_j||_2^2)^{-1}}. \tag{4.47}$$

The matrix elements q_{ij} are then the probability of the latent sample $\mathbf{z}_i$ belonging to cluster j. A schematic of the model is shown in figure 4.4 to illustrate the structure of the DCEC model.

To define the clustering loss over the soft assignments, we must first compute a target distribution p_{ij}. This distribution represents the confidence of the assignment q_{ij}. We define the 'target distribution as

$$p_{ij} = \frac{q_{ij}^2 / \sum_i q_{ij}}{\sum_j q_{ij}^2 / \sum_i q_{ij}}. \tag{4.48}$$

It is important to note that these distributions are not chosen arbitrarily. The soft assignments are computed in a way that is analogous to the t-SNE method described by Van Der Maaten and Hinton [46]. Furthermore, the distribution p_{ij} is chosen to improve cluster purity and emphasize the assignments with high confidence, according to [47]. The loss is then computed as the KL-divergence between p_{ij} and q_{ij}, i.e

$$\mathcal{L}_z = D(P||Q) = \sum_i \sum_j p_{ij} \log \frac{p_{ij}}{q_{ij}}. \tag{4.49}$$

Guo et al. [44] show that the target distribution should not be updated with each epoch. Instead, it needs to be changed on a regular schedule. They found that for the handwritten digits dataset MNIST a suitable update was once every $T = 140$ epochs.

Training the DCEC algorithm is split into two phases. First, the convolutional autoencoder is trained until convergence with no regularization on the latent space. Secondly, the cluster centres are initialized using a k-means algorithm and which is then used to compute the target distribution for the first T epochs. Lastly, the algorithm is trained with the KL-divergence loss and the original reconstruction term. We can then write the total cost as the sum of the reconstruction, $\mathcal{L}_x$, and clustering-loss, $\mathcal{L}_z$ as

$$\mathcal{L} = \mathcal{L}_x + \gamma \mathcal{L}_z, \tag{4.50}$$

where γ is a weighting term for the clustering loss. [44] empirically set $\gamma = 0.1$ for their experiments.

The fundamental challenge that DCEC faces is that it is dependent on a K-means solution that is good enough after the pre-training of the convolutional autoencoder. As the K-means algorithm is susceptible to outliers, scale differences in the latent axes, and assumes that the clusters are isotropic Gaussians.

4.6.2 Mixture of autoencoders

Another way of representing the clusters is by having multiple latent spaces representing the underlying manifolds that describe each class. This is the central idea in the Mixture of autoencoders (MIXAE) algorithm, introduced by [45]. To ensure that each autoencoder represents a cluster, we attach a soft-max classifier to the set of latent samples. This classifier assigns a cluster to each autoencoder, coupling the latent space and the reconstructions. The soft-max classifier is trained to output cluster probabilities and penalized for collapsing to assigning one cluster only. The remaining task is then to connect the cluster assignments to the autoencoder reconstructions. This is achieved by multiplying by the cluster confidence with the reconstruction error of each autoencoder.

More formally let $\{\mathbf{z}_j^{(i)}\}^N$ be the set of N latent samples from each of the N auto-encoders from a single sample image $\hat{\mathbf{x}}^{(i)} \in \mathbb{R}^{t \times v}$ with height t and width v. Furthermore, let the soft cluster assignments be given as $\{p_j^{(i)}\}^N$ and the reconstructed samples be given as $\{\mathbf{x}_j^{(i)}\}$. The reconstruction loss is then the sum over each autoencoder multiplied with each cluster assignment, i.e.

$$\mathcal{L}_x = \sum_j p_j^{(i)} \mathcal{C}(\mathbf{x}_j^{(i)}, \hat{\mathbf{x}}^{(i)}), \tag{4.51}$$

where $\mathcal{C}(\cdot)$ is a cost function like the mean squared error or a cross-entropy.

We ensure that the soft cluster assignments encourage clustering by adding two terms to the total loss. The first is a simple entropy term which, when minimized, encourages the assignments to be one-hot vectors. Mathematically this loss can then be written as the entropy $S(\cdot)$ of a cluster assignment $\mathbf{p}$

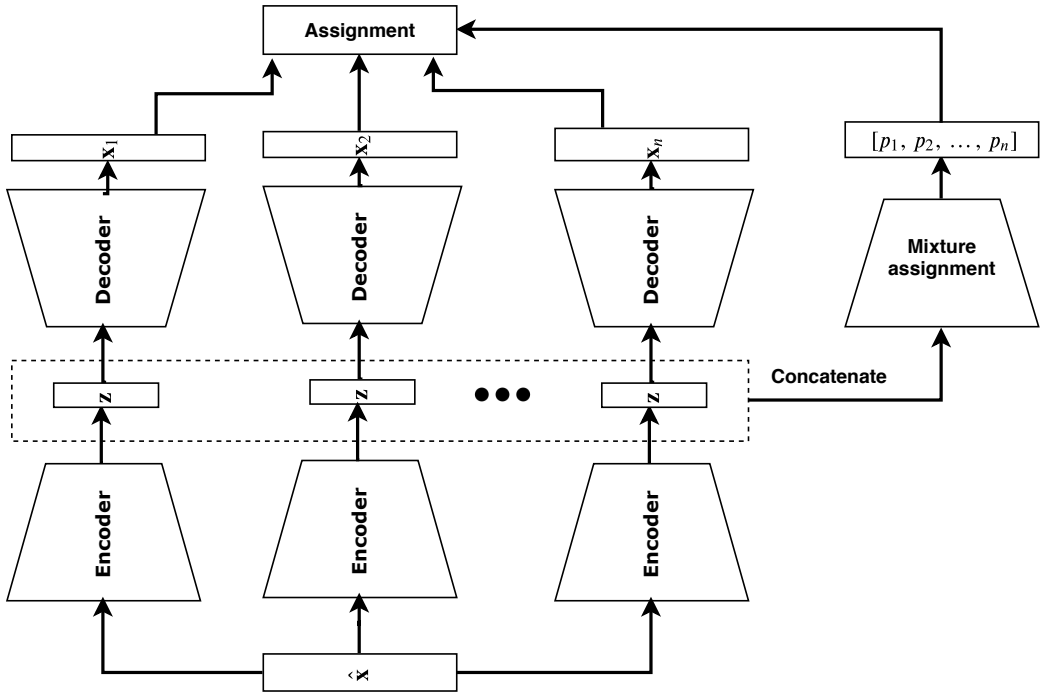


Figure 4.5: Schematic of a DCEC model. A sample $\hat{\mathbf{x}}$ is compressed to set of lower-dimensional representations $\{\mathbf{z}^{(i)}\}$ by N autoencoders. These samples are concatenated and passed through an auxiliary assignment network that predicts a confidence of cluster belonging for each autoencoder. For further details see the text. Figure adapted from Zhang et al. [45]

$$S(\mathbf{p}^{(i)})_{\text{sample}} = -\sum_j p_j^{(i)} \log p_j^{(i)}. \tag{4.52}$$

The last ingredient in the loss is then the term which discourages the trivial solution where only one cluster is being assigned. This loss is a batch-wise entropy term, given a mini-batch of β samples the loss is computed as

$$\begin{aligned} S(\{\mathbf{p}^{(i)}\}_i)_{\text{batch}} &= \left(-\sum_j \bar{p}_j \log \bar{p}_j \right)^{-1}, \\ \bar{\mathbf{p}} &= \frac{1}{\beta} \sum_i^{\beta} \mathbf{p}^{(i)}. \end{aligned} \tag{4.53}$$

The negative exponent in equation 4.53 owes to the fact that we want to maximize the batch-wise entropy.

From equations 4.51, 4.52, and 4.53 we can then compose the full loss for the MIXAE model. [45] note that the optimization depends heavily on the weighting of the different terms and so we introduce the weighting hyperparameters α, γ and θ, and write the total loss over a mini-batch as

$$\begin{aligned}\mathcal{L}_{\text{total}}(\{\hat{\mathbf{x}}^{(i)}\}_i^{\beta}) = \frac{1}{\beta}\sum_i \Big(&\frac{t \cdot v}{\theta}\mathcal{L}_x(\mathbf{p}^{(i)}, \mathbf{x}^{(i)}, \hat{\mathbf{x}}^{(i)}) \\ &+ \alpha S(\mathbf{p}^{(i)})_{\text{sample}}\Big) \\ &+ \gamma S(\{\mathbf{p}^{(i)}\}_i)_{\text{batch}}.\end{aligned} \tag{4.54}$$

Zhang et al. [45] note that this algorithm has a couple of shortcomings. The batch-entropy term encourages the cluster assignments to be uniformly distributed within the classes. However, it does have a second minimum when the assignments are all equally likely. For biased datasets with significant variations in the number of samples in each class, this can create problems for the algorithm. Additionally, it suffers from the same problem that K-means does in that we need to guess at the number of clusters present in the data, which may not be known.

Chapter 5

Neural architectures

The research question explored in this thesis necessitates two separate architectures. In the case where we have access to some labelled data the goal is to learn as much as possible from the event distribution and create an informative representation which is used to train a classifier on the labelled data. Learning the data distribution is done with an autoencoder network, then the informative representation ins the compressed latent space of the model This regime is semi-supervised as most of the training time is spent trying to learn an informative compression over the data-distribution. Additionally this approach seeks to lessen the probability of overfitting by using a very simple model as the classifier on the compressed representation. This is the *classification* scheme.

Access to labelled data is not guaranteed, however. Labeling data requires a very well determined system with very disparate reaction events. In the ^{46}Ar AT-TPC experiment the proton and carbon products are different enough to produce visually distinct tracks, but this is not generally the case. Having access to a fully unsupervised method of separating classes of events can then be hugely beneficial to researchers. In the event where we don't have access to labelled data we have to discover emergent clusterings in the data without knowledge about the class distribution. In this case we still use an autoencoder model, but different demands have to be made of the latent space. This is the *clustering* scheme.

5.0.1 Classification

We will leverage the autoencoder to gain information about the event distribution from the volume of unsupervised data. The modeling pipeline for the classification task is then concisely summarized with:

1. Train Autoencoder end-to-end on full data with a select regularization on the latent space until converged or it starts to overfit.

2. Use the encoder to produce latent representations of the labelled data

3. Train a logistic regression model using the latent representations of the labelled data

We will determine the best autoencoder architecture for each dataset listed in section 6.3. The best autoencoder is measured by performance in identifying separate classes by the logistic regression model on a test-set of data.

5.0.2 Clustering

In the case where labelled data is not available, we must turn to unsupervised methods. For the present work, we implemented two clustering algorithms based on autoencoding neural networks: the deep convolutional embedded clustering (DCEC) and the mixture of autoencoders (MIXAE) algorithm.

The MIXAE is trained end-to-end on the individual AT-TPC datasets and subsequently evaluated by the clustering metrics presented in section 2.13.

The DCEC modelling pipeline has two distinct steps: first, we pre-train the convolutional autoencoder and subsequently we apply the clustering layer loss which trains the clustering part of the algorithm. Training the DCEC algorithm then follows following pipeline:

1. Train autoencoder end-to-end on the full dataset without regularizing the latent space.

2. Compute latent representations of the full dataset

3. Determine initial centroids from a K-means fit of the latent representations of the full dataset

4. Train the autoencoder end-to-end on the full dataset with an added regularization of the soft cluster assignments to the target distribution of pseudo labels.

In the same manner as for the clasification task we will search over autoencoder architectures and will select the highest performing model by it's performance on the labelled subset of the datasets.

5.0.3 Pre-trained networks

Following the precedent of Kuchera et al. [17] we will consider representations of our events through the lens of a pre-trained network. In the Machine Learning community it is not uncommon to publish packaged models with fitted parameters from image recognition contests. These models are trained on datasets with millions of images and classify between hundreds of distinct classes, one such is

the imagenet dataset. In their work Kuchera et al. [17] use the VGG16 architecture trained on imagenet to classify AT-TPC events, in this thesis we will build on the understanding of using these pre-trained networks in event classification by using VGG16 as an element in the end-to-end training of autoencoders. The VGG16 network is one of six analogous networks proposed by Simonyan and Zisserman [48], they were runners up in the ISLVRC(ImageNet large scale visual recognition competition) of 2014 [49]. The network architectures are fairly simple, for VGG16 there are sixteen layers in the network. The first thirteen of which are convolutional layers with exclusively 3×3 kernels. The choice of the kernel size is based on the fact that a stacked 3×3 is equivalent to larger kernels in terms of the receptive field of the output. Three 3×3 kernels with stride 1 have a 7×7 receptive field, but the larger kernel has 81% more parameters and only one non-linearity [48]. Stacking the smaller kernels then contributes to a lower computational cost. Additionally there is a regularizing effect from the lowered number of parameters, and increased explanatory power from the additional non-linearities. The full architecture is detailed in appendix C.1.

We include the pre-trained VGG16 network in the autoencoder architectures in one of the three possible configurations. The pre-trained network can either:

1. Have their parameters fixed, thus creating a new representation of the input in terms of this particular model. In this way the autoencoder does not reconstruct from the image x but rather from the representation $VGG16(x)$. The decoder is here not a mirror of the encoder

2. Have their parameters be trainable. In this configuration we use the pre-trained network as the encoder function itself and encode to a lower dimensional space for the latent representation which is used for the reconstruction. The decoder is here not a mirror of the encoder

3. Have their parameters be randomly initialized. In other words we can simply use the architecture of the network but not the pre-trained weights. This is just a normal autoencoder, with a mirrored encoder-decoder pair.

We choose the first configuration for the interface between the autoencoder models and the VGG16 model.

Additionally, we compare the autoencoder models with a baseline VGG16 model. Which is to say that the performance of the methods proposed in this thesis will be measured against the performance of simply passing the events through the pre-trained VGG network and then to a logistic regression classifier for the *classification* scheme. And to a K-means algorithm for the *clustering* scheme.

Chapter 6

Experimental Background

The experiment, which is the topic of analysis in this thesis, was conducted at the facility for rare isotope beams (FRIB) located on the Michigan State University (MSU) campus. As the name implies, the FRIB offers researchers the ability to study isotopes far from stability. These isotopes are short-lived, and not normally occurring. Applications of the studies conducted at the FRIB include furthering the understanding of nuclear structure, nuclear astrophysics, and have applications in medicine and industry.

Nuclei with very low production rates necessitates a detector with very high efficiency, one of which is the active target time projection chamber (AT-TPC). While in some applications, the reactions of interest can be extracted using traditional testing methods, the low cross-section of relevant interactions necessitates an individual consideration of each reaction. The goal of the analysis is then to extract as many of the events of interest. All the while keeping the sample as pure as possible.

Traditional methods of extraction have been centered around Markov chain Monte Carlo (MCMC) algorithms. The fitting procedure presumes that each event is a reaction of interest, and performs an integration over the reaction vertex, initial momenta, etc. Subsequently, a threshold for the fits was computed to extract the events of interest. This approach has two fundamental challenges, both related to the process of fitting against the positive class. Firstly, if the breadth of reactions is not known prior to the analysis, fitting against the positive class might yield unexpected results. Secondly, the fitting presumes complete tracks in the generated point-cloud of the event. In the event of broken tracks, the fitting will not converge satisfactorily. Additionally, the computational cost of fitting each track with an MCMC algorithm is prohibitively large.

The foray into machine learning is then an attempt to address these challenges. This started with the work by [17] in which the authors successfully train high performing classifiers for the ^{46}Ar experiment performed with the AT-TPC detector. In this thesis, we elaborate on these results by introducing unsupervised techniques for the separation of reactions in the same experiment.

6.1 A note on nuclear physics

In this thesis, we primarily concern ourselves with analysis methods that are agnostic to the physics in the system. One can argue that this is both a strength of the methodology and a weakness. As a consequence, the discussion of the physical system will brief. For a more in-depth treatment of the physics see [50].

With that in mind, we turn to the central pursuits of nuclear physics: understanding the structure of the nucleus. Nuclides are described in terms of the number of protons, Z, and neutrons N and their total mass number $A = Z + N$. Some select nuclides have special names; nuclides with an equal number of protons are called *isotopes*, an equal number of neutrons *isotones* and with the same number of nucleons *isobars*. The first modern fully-formed theory of nuclear structure, the nuclear shell model, was focused around the observation that certain isotopes and isotones were much more stable than others. As it happened, these stable nuclei were regularly spaced around certain numbers of constituent protons and neutrons. These numbers are called magic numbers and describe nuclides that are much more tightly bound than the next number; as a consequence, they are very stable and exhibit long half-lives. These magic numbers are: 2, 8, 20, 28, 50, 82, 126. Some nuclides are even doubly-magic, which is to say both Z and N are magic numbers. One area of active research is around the $N = 28$ isotones. Predictions from the nuclear shell model indicate that these isotones should have an approximately spherical structure. However, this has been disproved experimentally, as deformities appear when removing protons from the spherical nucleus ^{48}Ca. This brings us to ^{46}Ar which lies in a region between the spherical ^{48}Ca and the lighter isotones that are known to be deformed. The location of ^{46}Ar makes it an object of some academic interest and served as the commissioning of the AT-TPC at the NSCL.

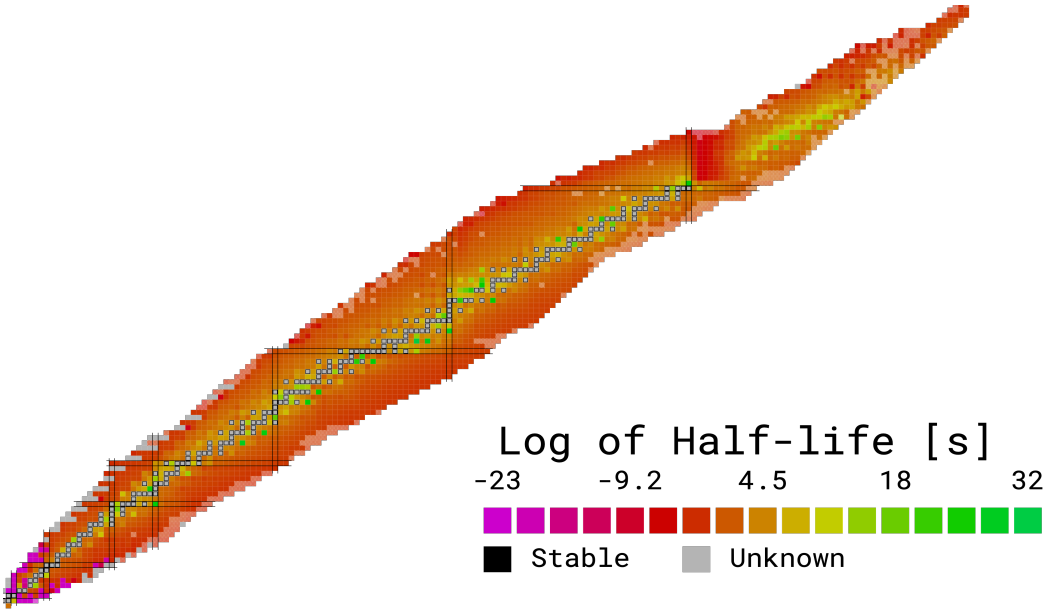


Figure 6.1: Chart of the known nuclides. The number of protons are given along the vertical axis, and neutrons along the horizontal. The color indicates the log half-life of the nucleus. Lines of isotones and isotopes along the magic numbers are indicated by rectangles in the figure. Retrieved from Edward Simpson's website.

6.2 Active Target Time Projection Chambers

The active target time-projection chamber (AT-TPC) is a detector constructed for the detection of low energy beams with very high efficiency. It was constructed at the national superconducting cyclotron laboratory (NSCL) facility on the Michigan state university (MSU) campus. The detector is attached to the ReA3 linear accelerator which provides high-quality rare isotope beams. The low-intensity of these beams necessitates a detector with high efficiency, while the low energies in the interval of 0.3 MeV/u to 6 MeV/u means that solid or liquid targets are not feasible [51]

With the ability to provide high-quality beams of rare isotopes, the ReA3 re-accelerated beam facility provides essential research opportunities in nuclear physics [52].

The detector consists of a cylindrical volume inserted into a solenoid magnet. The magnet, which was designed for medical imaging, applies a 2 T magnetic field which is fairly uniform inside the volume [51]. Inside the detector, the beam enters into a gas-filled volume, where the gas is specified to contain the target nucleus of interest. A sensor plane is placed at the end of the detector, opposite the beam entrance, that records events that happen in the gaseous volume. The sensor plane is a micromegas device consisting of an electron-multiplying mesh

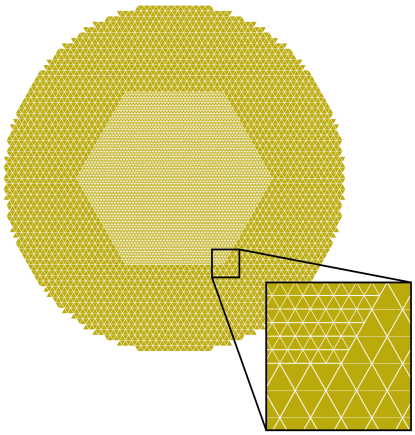


Figure 6.2: Figure showing the pad-plane in the AT-TPC. There are two regions of sensor-pad densities to keep the number of pads reasonable, while ensuring a high resolution in the region with high expected activity. Figure produced with the pytpc package.

and a plane of triangular sensors that record the impacts of impinging electrons, as described by Giomataris et al. [53]. The pad-plane is shown in figure 6.2

To get the ionization electrons from reactions in the detector to the sensor, a 1×10^4 V electrical potential difference is applied between the sensor plane and the beam entrance. Furthermore, the field is kept uniform by a field cage surrounding the volume, which gradually steps down the voltage [51]. A schematic of the AT-TPC is included in figure 6.3

The recorded signal for an event is then a collection of charge measurements from the individual sensor pads of the micromegas. In the experiment, it is assumed that each pad will fire a maximum of one time per event, and as such track reconstruction begins with converting the signal from time buckets to a z coordinate. However, in this thesis, we are only concerned with the x-y projection of the events.

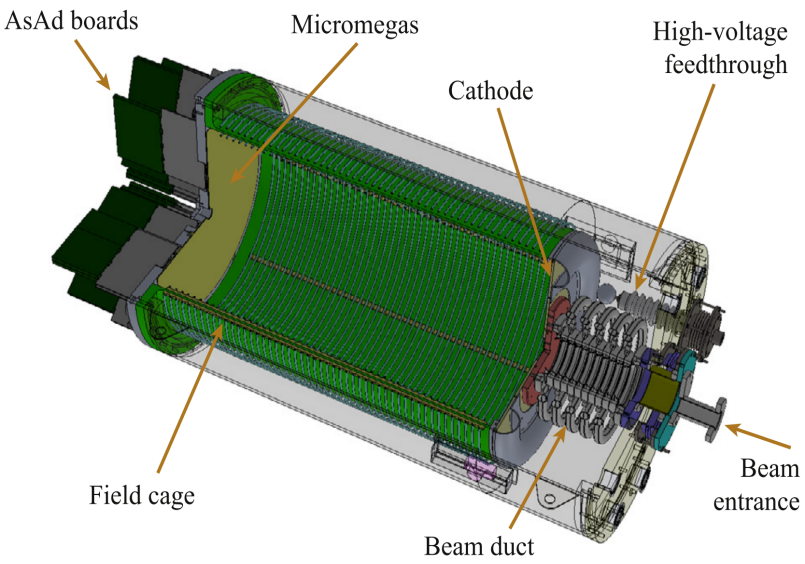


Figure 6.3: Cross section of the AT-TPC detector volume, with the outer shielding made transparent. On the right hand side the beam enters the chamber and some of the signal processing and recording equipment is shown downstream in the figure. Figure copied from [51]

6.3 Data

In this thesis, we will work with data from the $^{46}Ar(p, p')$ experiment conducted at the national superconducting cyclotron laboratory (NSCL) located on the Michigan state university campus. We principally work with two different data sources: data from AT-TPC simulation tools, and data recorded in the AT-TPC experiment proper. The experimental data were recorded from a single run of the experiment, which yields on the order of $\sim 10^4$ events. For the simulated data, we construct two datasets on the order of $\sim 10^3$ and $\sim 10^4$ events, respectively. In this section we give a brief overview of the data, for a more in-depth consideration we refer to [54], [55] and [51].

In this thesis we will explore the machine learning algorithms described in chapters 3 and 4 on three datasets from the AT-TPC, listed in table 6.1. The simulated data will serve as a baseline for performance, while the filtered and full data are real records from the ^{46}Ar experiment. Each of these datasets have different distributions of the constituent classes, which we list in table 6.2. The filtered data differs from the full dataset as it has some post-processing applied to remove noise, which is not the case for the full data. The details of the post-processing are given in later sections. First, we consider the pipeline from raw data to images that the algorithms described in chapter 4 can process.

Table 6.1: Descriptions of number of events in the data used for analysis in this thesis. In principle we can simulated infinite data, but it is both quite simple and not very interesting outside a case for a proof-of-concept

	Simulated	Full	Filtered
Total	8000	51891	49169
labelled	2400	1774	1582

Table 6.2: Event class percentages for each of the datasets used in the analysis conducted in this thesis. The decrease in the "other" class of events from full to filtered data owes to the thresholding of events. If an event contains fewer than 20 data-points it is discarded

	Simulated	Full	Filtered
% Proton	50	25.3	27.6
% Carbon	50	12.0	11.9
% Other	0	62.8	60.1

6.3.1 Data processing

Our data processing pipeline begins with localized point-cloud data in the two-dimensional detector coordinate system, with one time dimension and a corresponding charge measurement. The charge and time measurement are extracted as the peak of this signal over the event, resulting in a maximum of one measurement per pad in the sensor plane. The events range from fairly sparse $< 10^2$ records to being very populated, depending on where in the volume the reaction occurs as well as other noise-generating factors. We centre the charge data to values > 1 by adding the lowest occurring record in the run and apply a log-transform to get values in $\mathbb{R}^+$. Subsequently, we scale by the maximum value in the run to get charge data in the interval $[0, 1]$. Lastly, the transformed charge data is saved in a two-dimensional $128px \times 128px$ array using the matplotlib package provided in the Python programing language [56]. The choice of scaling was made to accommodate a binary cross-entropy loss on a 2D projection, as it presumes the true values to be bounded as probabilities.

We will begin by considering the simulated events, followed by subsequent considerations of the full and filtered experimental data.

6.3.2 Simulated ^{46}Ar events

The simulated AT-TPC tracks were simulated with the pytpc package developed by Bradt et al. [51]. Using the same parameters as for the $Ar^{46}(p, p)$ experiment, a small set of $N = 4000$ events were generated per class, as well as a larger

set of $N = 40000$ events per class. The events are generated as point-clouds, consisting of position data on the x-y plane, a time-stamp and an associated charge value. These point-clouds are transformed into pure x-y projections with charge intensity for the analysis in this thesis using the pipeline described in the previous section.

More formally the events are originally composed of peak-only 4-tuples of $e_i = (x_i, y_i, t_i, c_i)$. The peak-only designation indicates that we use the recorded peak amplitude on each pad, the tuples then correspond to pads that recorded a signal for that event. Each event is then a set of these four-tuples: $\epsilon_j = \{e_i\}$ creating a track in three-dimensional space.

To emulate the real-data case, we set a subset of the simulated data to be labelled and treat the rest as unlabelled data. We chose this partition to be 15% of each class. We denote this subset and its associated labels as $\gamma_L = (\mathbf{X}_L, \mathbf{y}_L)$, the entire dataset which we will denote as $\mathbf{X}_F$. To clarify, please note that $\mathbf{X}_L \subset \mathbf{X}_F$.

We display two simulated events in figure 6.4. The top row illustrates a proton-event, and the bottom a carbon-event.

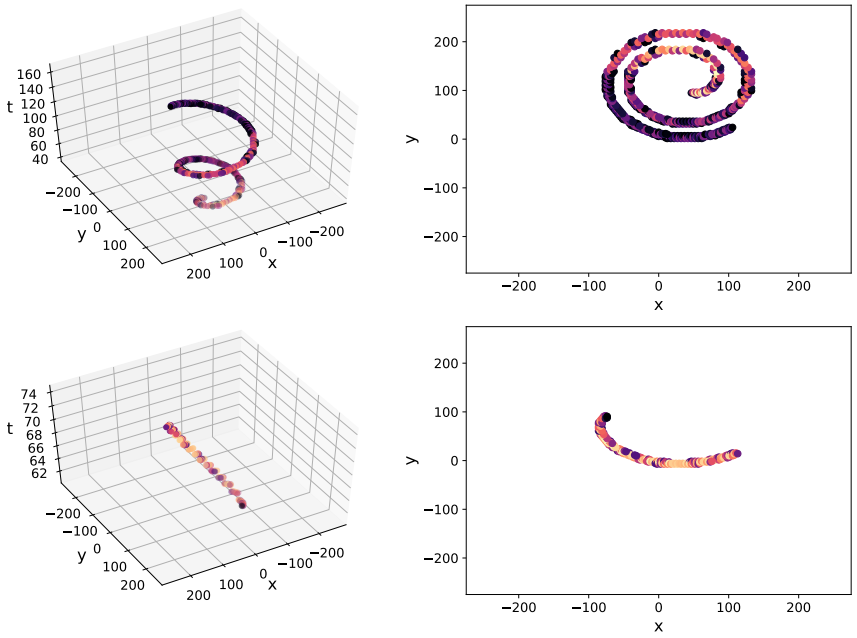


Figure 6.4: Two- and three-dimensional representations of two events from a simulated ^{46}Ar experiment. Each row is one event in two projections, where the lightness of each point indicates higher charge values.

6.3.3 Full ^{46}Ar events

The events analyzed in this section were retrieved from the ^{46}Ar resonant proton scattering experiment recorded with the AT-TPC.

The sensor plane in the AT-TPC is very sensitive, as such there is substantial noise in the ^{46}Ar data. The noise can be attributed to structural noise from electronics cross-talk, and possible interactions with cosmic background radiation, as well as other sources of charged particles. Part of the challenge for this data then comes from understanding of the physics of the major contributing factors to this noise.

We display two different events from the ^{46}Ar experiment in figure 6.5. The top row illustrates an event with a large fraction of noise, while the bottom row shows an event nearly devoid of noise. The very clear spiral structure of the bottom row indicates that this is a proton-event.

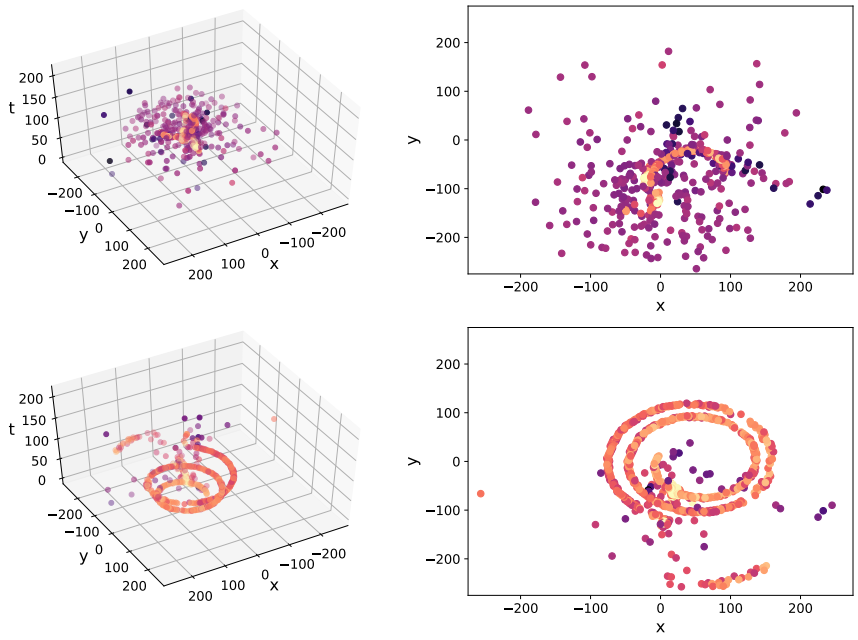


Figure 6.5: Two- and three-dimensional representations of two events from the ^{46}Ar experiment. Each row is one event in two projections, where the lightness of each point indicates higher charge values.

6.3.4 Filtered ^{46}Ar events

As we saw in the previous section, the detector picks up a significant amount of noise. We split the noise broadly in two categories, one being random-uncorrelated noise and the second is structured noise. The former can be quite trivially removed with a nearest-neighbour algorithm that checks if a point in the event is close to any other. To remove the correlated noise, researchers at the NSCL developed an algorithm based on Houghes' transform. This transformation is a common technique in computer vision, used to identify common geometric shapes like lines and circles. Essentially, the algorithm draws many lines (of whatever desired geometry) through each data-point and checks whether these lines intersect with points in the dataset. Locations in parameter space that generate many intersections then become bright spots, allowing us to filter away points that are not close to these points. These algorithms remove a large amount of the unstructured noise and are computationally rather cheap.

We illustrate two filtered events in figure 6.6. These are the same events as shown in figure 6.5, but with the Houghes' and nearest neighbours filtering applied.

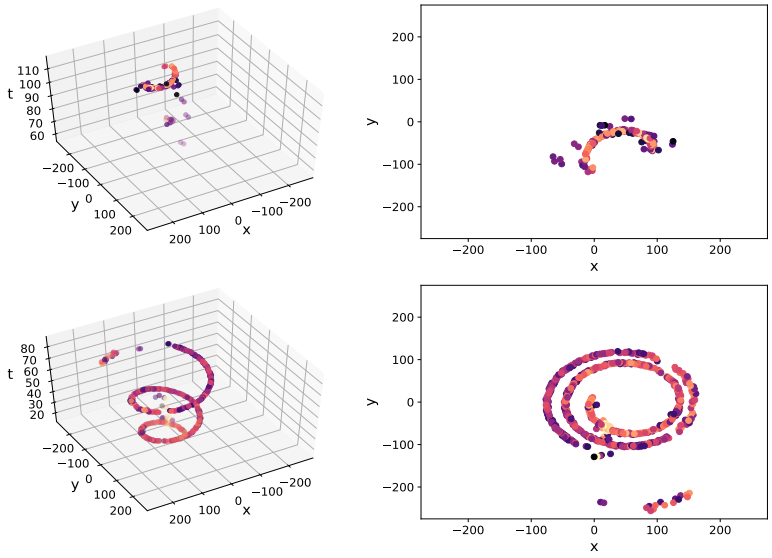


Figure 6.6: Two- and three-dimensional representations of two events from the ^{46}Ar experiment. Each row is one event in two projections, where the lightness of each point indicates higher charge values. These events have been filtered with a nearest neighbors algorithm and a Houghes' transform, described in section 6.3.4

pick out some samples from each class in appendix?

Part II

Implementation

Chapter 7

Methods

In this chapter, we will introduce the algorithms we have implemented for this thesis. Additionally, we will demonstrate their application to simulated active target time-projection chamber (AT-TPC) data. The algorithms were presented in chapter 4, and we refer to this chapter for details on the theory. All the algorithms implemented for this thesis are written in the Python programming language. We chose python for the ability to rapidly prototype an algorithm, as well as the availability of mature machine learning libraries. Plots of algorithmic performance and data visualization were achieved with the matplotlib package [56], and data was managed with the numpy package for numerical python [7].

The algorithms we implemented were written using the TensorFlow library for deep learning. Accordingly, we begin this chapter by describing the TensorFlow framework.

7.1 The TensorFlow library

Google began development of TensorFlow in 2011, and it has since risen to preeminence as one of the two de facto libraries for deep learning in python[1] [57]. The library implements tools for the development of a variety of deep-learning architectures, from fundamental linear algebra operations to interfaces with common layer types. At the core of the library is a computational graph structure constructed for numerical computations, which separates TensorFlow from other libraries for numerical python applications. The graph does not execute an operation when it is defined but rather executes operations when asked to retrieve values for a specific variable in the graph. Writing code in TensorFlow is in this way similar to writing code in a traditional compiled language.

One of the challenges when writing scientific code in python is that indexing and iteration in loops are notoriously slow by default, but they can be sped up considerably. Speeding up python is achieved by avoiding python's built-in iterables

[1] The other is PyTorch, which is developed by Facebook.

and loop structures where possible. Instead we rely on interfaces to heavily optimized Cor C++ code. This is what allows us to perform quite demanding computations with TensorFlow.

7.1.1 The computational graph

To understand the program flow of the algorithms in subsequent sections we begin by introducing the fundamental concepts of TensorFlow code [2]. The heart of which is the computational graph. A graph defines all the operations needed for a model or another series of computations of interest. To retrieve values from the graph, we run it with an input, which is defined as a placeholder tensor[3] object. TensorFlow placeholder objects are special tensors that define the entry-point for the graph and acts as the input for a model, function or class. We include a code snippet with an associated graph in figure 7.1. It shows a simple program that computes a weight transformation of some input with a bias. The cost is included as the bracketed ellipses. When the forward pass is computed, or *unrolled*, it becomes available for automatic differentiation.

For the algorithms implemented in this thesis, we set up the computational graph to represent the forward pass, or predictive path, of the algorithm. The remainder then is then to compute the gradients required to perform gradient descent. TensorFlow provides direct access to find the gradients via tf.gradients(C, [I]k) where [I]k represents the set of tensors we wish to find the gradients of C with respect to. We use these gradients to perform the backpropagation algorithm, described in detail in section 3.1. We show an illustration of the computational graph and the corresponding gradient nodes in figure 7.2. The nodes in the figure represent TensorFlow operation types including variable declarations and operations on those including Add and MatMul operations.

To accommodate different gradient descent schemes, TensorFlow wraps the computation of gradients in optimizer modules. Defined in tf.train these include stochastic gradient descent and ADAM, which we discuss in section 2.10. In this thesis, we will largely be using the ADAM optimizer.

We outline a basic TensorFlow script in 7.3 that goes through the basic steps outlined above.

[2]The thesis code was written for the latest stable release of TensorFlow prior to the release of TF 2.0. Accordingly, some modules may have moved, changed name or have been deprecated. Most notably in the versions prior to TF 2.0 eager execution was not the default configuration and as such the trappings of the implementation includes the handling of session objects.

[3]We follow the TensorFlow nomenclature and define tensors as multidimensional arrays, unless otherwise explicitly stated.

```python
import tensorflow as tf

# placeholder for input to the computation
x = tf.placeholder(dtype=tf.float32, name="x")

# bias variable for the affine weight transformation
b = tf.Variable(tf.zeros(100))

# weight variable for the affine wegiht transformation with random values
W = tf.Variable(tf.random_uniform([784, 100]), tf.float32)

# activation as a function of the weight transformation
a = tf.relu(tf.matmul(W, x) + b)

# cost computed as a function of the activation
# and the target optimization task
C = [...]

# Start session to run the computational graph
session = tf.InteractiveSession()

# Initialize all variables, in this example only the weight
# matrix depends on an initialization
tf.global_variables_initializer()

for i in range(epochs):
    result = session.run(C, feed_dict={x: data[batch_indices]})
    print(i, result)
```

Figure 7.1: This short script describes the setup required to compute a forward pass in a neural network as described in section 3.1. Including more layers is as simple as a for loop and TensorFlow provides ample variations in both cell choices (RNN variations, convolutional cells etc.) and activation functions. This script is a modified version of figure 1 in [57]

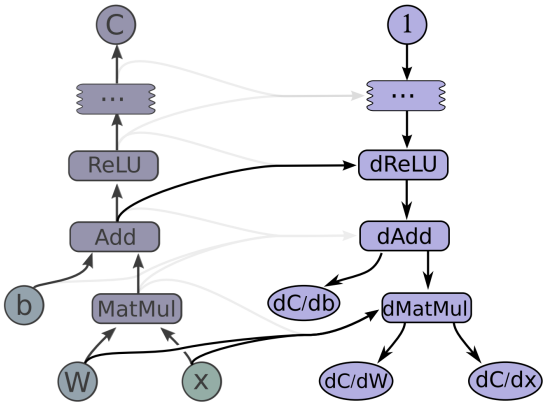


Figure 7.2: A graph representation of the short script in figure 7.1, with respective gradients on the right. This figure is copied from Abadi et al. [57]

```python
import tensorflow as tf

# placeholder for input to the computation
x = tf.placeholder(dtype=tf.float32, name="x")

# bias variable for the affine weight transformation
b = tf.Variable(tf.zeros(100))

# weight variable for the affine wegiht transformation with random values
W = tf.Variable(tf.random_uniform([784, 100]), tf.float32)

# activation as a function of the weight transformation
a = tf.relu(tf.matmul(W, x) + b)

# cost computed as a function of the activation
# and the target optimization task
C = [...]

# define optimizer function and compute gradients
# include optimizer specific hyperparameters
optimizer = tf.train.AdamOptimizer(eta=0.001)
grads = optimizer.compute_gradients(C)

# define update operation
opt_op = optimizer.apply_gradients(grads)

# Start session to run the computational graph
session = tf.InteractiveSession()

# Initialize all variables, in this example only the weight
# matrix depends on an initialization
tf.global_variables_initializer()

for i in range(epochs):

    # runs the grapgh and applies the optimization step, running opt_op
    will
    # compute one gradient descent step.
    result, _ = session.run([C, opt_op], feed_dict={x:
    data[batch_indices]})
    print(i, result)
```

Figure 7.3: A TensorFlow script that uses the `tf.train` module to compute the gradients needed to perform backpropagation of errors on the cost function assigned to the variable `C`. Additionally we show the structure of a session and it's run method to perform a backwards pass with respect to the loss `C`

7.2 Deep learning algorithms

All the models in this thesis are implemented in the python programming language using the TensorFlow library for deep learning. The code is open source and can be found in a github repository https://github.com/ATTPC/VAE-event-classification. In this section we will be detailing the framework built for the algorithms discussed in chapter 4.

The architecture is straightforward, and consists of a model class LatentModel that implements shared functionalities between models. Individual algorithms are then implemented as subclasses of LatentModel, these are discussed in detail in the coming sections. We also define helper-classes to perform hyperparameter searches, and to manage mini-batches of data. Similarly, the clustering algorithm deep convolutional embedded clustering (DCEC) is instantiated as a convolutional autoencoder, while the mixture of autoencoders (MIXAE) algorithm is constructed using the convolutional autoencoder class internally.

Throughout the thesis we follow the convention that classes are named in the CamelCase style, and functions and methods of classes in the snake_case style.

The subclasses of LatentModel implement two main functions: compute_gradients, and compile_model, which call the model specific functions that constructs the computational graph, and subsequently computes the gradients. The gradient computation is done through a TensorFlow optimizer class, which defines the operations needed to update the weights.

The LatentModel class contains the framework and functions used for common training operations. In the initialization it iterates through a configuration dictionary, which defines class variables pertinent to the current experiment. The configuration explicitly defines the type of latent loss (discussed in section 4.4) to be used for the experiment, as well as whether to use the DCEC clustering loss, batch-normalization or if the model should restore the weights from a previous run. These are saved in the model state and referenced at key junctions.

After initialization, but before training the model class needs to construct the computational graph which defines the forward pass of the algorithm, as well as the loss components needed for the backwards pass. These components are wrapped in the method compile_model, defined in LatentModel. It takes two dictionaries as arguments specifying the graph and loss configuration. They are subclass specific and will be elaborated on later in sections 7.3 and 7.4. But they include the specification of regularization strength and type, as well as the type of activation and loss functions. The method also sets the compiled flag to True which is a prerequisite for the compute_gradients method.

When the model is compiled we turn to the computation of the gradients. In this thesis we use TensorFlow optimizer objects to prepare the update operations needed in the training. The compute_gradients method then takes an optimizer object, e.g. AdamOptimizer, and its positional and keyword arguments. Inside the

method the optimizer is instantiated in order to compute the gradients. Lastly, the optimizer prepares the update operations which is fed to the Session object to update the weights. The model is now ready for training.

The training procedure is implemented in the train method of the LatentModel class, and handles both checkpointing of the model to a file, logging of loss values and the training procedure itself. As discussed in section 2.10 we use the adam mini-batch gradient descent procedure. The train method also contains the code to run the pre-training required for the DCEC algorithm, described in section 4.6. As part of that procedure we use an off-the-shelf version of the K-means algorithm, implemented in the scikit –learn package [23] to find the initial cluster locations. The main loop of the method iterates over the entire dataset and performs weight updates. For the DCEC an additional step is also included to update the target distribution p_{ij}.

7.3 Convolutional Autoencoder

The convolutional autoencoder class ConVae is implemented as a subclass of LatentModel. It implements the construction of the computational graph and the compute operations needed for the possible losses. To ascertain that the graph is constructed before the loss is computed, the user never interfaces with the _ModelGraph and _ModelLoss methods that implement the respective functionalities. Instead, the user calls a wrapper method from the parent class LatentModel.compile_model, which compiles the model for training.

The graph is constructed with specifications from configuration dictionaries passed through the compile_model method. The first of which specifies the activation function as well as the strength, and type of, weight regularization. The second dictionary specifies the type of reconstruction loss used in the optimization.

7.3.1 Computational graph

The private[4] function which enacts the computational graph is _ModelGraph. It accepts arguments for the strength and type of regularization on the kernel and bias parameters as well as the activation function to be used for the internal representations, and the projection to output space.

The encoder is constructed with a for loop over the number of layers, using a 2D convolutional layer and best practices for the application of batch-normalization. Wherein the normalization is applied after the activation for

[4]The term private is used loosely in the context of python programs, as the language does not actually maintain private methods. However, methods that are prefixed with an underscore are to be treated as private and are not exposed with public APIs and in documentation by convention.

exploding-gradient susceptible functions and before for functions with a vanishing gradient problem. In TensorFlow, we implement the construction of the encoder as

```
# excerpt from convolutional_VAE.py
# at https://github.com/ATTPC/VAE-event-classification
# from commit ca9b722
kernel_reg=reg.l2
kernel_reg_strength=0.01
bias_reg=reg.l2
bias_reg_strenght=0.00
activation="relu"
output_activation="sigmoid"

activations = {
    "relu": tf.keras.layers.ReLU(),
    "lrelu": tf.keras.layers.LeakyReLU(0.1),
    "tanh": Lambda(tf.keras.activations.tanh),
    "sigmoid": Lambda(tf.keras.activations.sigmoid),
}

self.x = tf.keras.layers.Input(shape=(self.n_input,))
# self.x = tf.placeholder(tf.float32, shape=(None, self.n_input))
self.batch_size = tf.shape(self.x)[0]
self.x_img = tf.keras.layers.Reshape((self.H, self.W, self.ch))(self.x)
h1 = self.x_img  # h1 = self.x_img
shape = K.int_shape(h1)

k_reg = kernel_reg(kernel_reg_strength)
b_reg = bias_reg(bias_reg_strenght)
# ... code omitted for brevity
for i in range(self.n_layers):
                with tf.name_scope("conv_" + str(i)):
                    filters = self.filter_arcitecture[i]
                    kernel_size = self.kernel_architecture[i]
                    strides = self.strides_architecture[i]
                    if i == 0 and pow_2:
                        padding = "valid"
                    else:
                        padding = "same"

                    h1 = Conv2D(
                        filters,
                        (kernel_size, kernel_size),
                        strides=(strides, strides),
                        padding=padding,
                        use_bias=True,
                        kernel_regularizer=k_reg,
                        # bias_regularizer=b_reg
                    )(h1)

                    if activation == None:
                        pass
                    elif activation == "relu" or activation == "lrelu":
                        a = activations[activation]
                        h1 = a(h1)
                        with tf.name_scope("batch_norm"):
                            if self.batchnorm:
                                h1 = BatchNormalization(
                                    axis=-1,
                                    center=True,
                                    scale=True,
                                    epsilon=1e-4
```

```
60                                  )(h1)
61                                  self.variable_summary(h1)
62                          else:
63                              a = activations[activation]
64                              with tf.name_scope("batch_norm"):
65                                  if self.batchnorm:
66                                      h1 = BatchNormalization(
67                                          axis=-1,
68                                          center=True,
69                                          scale=True,
70                                          epsilon=1e-4
71                                      )(h1)
72                                      self.variable_summary(h1)
73                              h1 = a(h1)
74
75                          if self.pooling_architecture[i]:
76                              h1 = tf.layers.max_pooling2d(h1, 2, 2)
```

Inside the method the placeholder variable ConVae.x is defined. The placeholder defines the entry point of the forward pass and is where TensorFlow allocates the batched data when optimizing and making predictions. Depending on whether the model is instructed to use the VGG16 representation of the data or a specified encoder structure it applies dense weight transformations with non-linearities or computes a series of convolutions respectively. Each convolutional layer is specified with a kernel size, a certain number of filters, and the striding. We also use a trick from Guo et al. [44] to determine the padding size, ensuring that the padding is chosen such that the reconstruction size is unambiguous. The padding is set to preserve the input dimensionality and is only reduced in dimensionality with striding or max-pooling. If the input dimension is of size 2^n, where n is the number of layers, the last convolution is adjusted to have no zero-padding.

After each layer, the specified non-linearity is applied. The models accept one of the sigmoid activations (logistic sigmoid or hyperbolic tangent) or the rectified linear unit family of functions as activations[5]. If the model configuration specifies to use batch normalization, this is applied before sigmoid functions and after rectified units. The reason for different points of application relates to the challenges of the respective activation families; sigmoids' saturate and so the input should be scaled, and rectified units are not bounded so the output is scaled. The output from the encoder is then a tensor with dimensions $\mathsf{h} = (\mathsf{o}, \mathsf{o}, \mathsf{f})$ where f is the number of filters in the last layer and $\mathsf{o} = \frac{H}{2^n}$ where n denotes the number of layers with stride 2 or the count of MaxPool layers, and H gives the input image size.

The tensor output from the encoder is then transformed to the latent space with either a simple dense transformation, e.g. $\mathsf{z} = \mathsf{Dense}(\mathsf{flatten}(\mathsf{h}))$. Alternatively, if a variational loss is specified, a pair mean and variance tensors. These are

[5]The model accepts a None argument for the activation in practice for debugging but this is not used for any models in this thesis.

constructed with dense transformations from h. The mean and variance tensors are then stored as class attributes of the ConVae instance:

```
self.mean = Dense(self.latent_dim, kernel_regularizer=k_reg)(h1)
self.var = Dense(self.latent_dim, kernel_regularizer=k_reg)(h1)
```

Using the re-parametrization trick shown by Kingma and Welling [36], we introduce stochasticity to the sample tensor in a manner which allows training by backpropagation. With re-parametrization, the sample is generated as z = self.mean + tf.exp(self.var)*epsilon, where epsilon is a stochastic tensor from the multivariate uniform normal distribution $\mathcal{N}(0, 1)$. Note that the self.var is treated as the log of the variance. Furthermore, he mean and standard deviation tensors are stored as class attributes to be used in the computation of the loss. The latent sample is also set as a class attribute for prediction.

After a sample z is drawn, the decoder computes the reconstruction for a given sample. The decoder is configured according to the instructions supplied in the initialization of the algorithm. In the case that the inputs are from a pre-trained model representation the model has the same call structure but with a boolean flag to the model use_vgg that indicates that the configuration is explicitly for the decoder.

7.3.2 Computing losses

To compute the loss(es) the ConVae implements the second of LatentModels abstract methods; _ModelLoss. Like the graph construction, this method is never called directly but through the interface of the parent class in the LatentModel.compile_model method. In a similar vein, losses are configured using a supplied dictionary.

The reconstruction loss is specified in the configuration dictionary, and the model accepts either a mean squared error or binary cross-entropy loss with cross-entropy as the default. Each of these losses acts pixel-wise on the output and target images. When computed, the construction loss is stored as a class attribute, ConVae.Lx, which is logged to a TensorBoard[6] instance. The reconstruction loss is then implemented as

[6]This is a module for tracking the progress of a model during training. It supports visualizations as well as automatically plotting the model graph, etc.

```python
# excerpt from convolutional_VAE.py
# at https://github.com/ATTPC/VAE-event-classification
# from commit ca9b722
x_recons = self.output
if self.use_vgg:
    self.target = tf.placeholder(tf.float32)
else:
    self.target = self.x
if reconst_loss == None:
    reconst_loss = self.binary_crossentropy
    self.Lx = tf.reduce_mean(
        tf.reduce_sum(reconst_loss(self.target, x_recons), 1)
    )
elif reconst_loss == "mse":
    self.Lx = tf.losses.mean_squared_error(self.target, x_recons)
```

Depending on the configuration, the model then compiles a loss over the latent space. In a variational autoencoder, the loss is a Kullback-Leibler divergence (KL-divergence) of the latent distribution with a multivariate normal distribution with zero mean and unit variance. This has a closed-form solution given a tensor representing the mean and standard deviation, which we derived in equation 4.25. This equation is relatively straightforward to implement, as it just implies a sum over the mean and log-variance tensor. The form of equation 4.25 also makes clear why we parametrize the log-variance and not the variance directly as the exponentiation ensures positivity of the variance.

```python
def kl_loss(self, args):
    """
    kl loss to isotropic zero mean unit variance gaussian
    """
    mean, var = args
    kl_loss = -0.5 * K.sum(
        1 + self.var - K.square(self.mean) - K.exp(self.var), axis=-1
    )
    return tf.reduce_mean(kl_loss, keepdims=True)
```

Alternatively to a KL-divergence the latent space may be regularized in the style proposed by Zhao et al. [38]. In this paradigm we compute a loss over the entire latent sample, and not component wise as the KL-divergence does. We use the radial basis function kernel to compute the maximum mean discrepancy divergence term introduced in equation 4.27, and use the implementation provided by [38] on their website. We choose the prior distribution fore MMD divergence to be a Gaussian mixture of two normal distributions with unit variance and means with a distance $\geq 3\sigma$ from each other.

7.3.3 Applying the framework

To illustrate the use and functionality of the model we'll demonstrate the pipeline for constructing a semi-supervised and clustering version of the architecture using the code written for this thesis. Beginning with the semi-supervised use-case.

These tutorials are also available in the Github repository for the thesis. They are provided as jupyter−notebooks and can be viewed in browser, or hosted locally. The example takes the reader through the entirety of the analysis pipeline as presented in chapter 5 and shows how the model was fit to data as well as post-analysis steps.

The goal of this example is to introduce the reader to the analysis framework used in this thesis. We will go through defining a model with convolutional parameters and fit this model to simulated AT-TPC events. With a 2D latent space this allows us to explore the latent configuration directly, but yields worse reconstructions. The notebook-tutorial walks through the example and is entirely analogous to this section.

We begin by loading the data files. The repository comes equipped with a small data-set of simulated data that can be analyzed. To achieve reasonable run-times a GPU enabled TensorFlow distribution is encouraged[7]. We assume that the script as we walk through it is located in the notebooks/ directory of the repository. We begin by making the necessary imports for the analysis. The packages TensorFlow and matplotlib have to be installed on the system for the tools to work, along with Numpy and pandas. The data_loader module contains functions to load files to numpy arrays, while the module convolutional_VAE contains the model class itself.

add sim-data to some file hosting

```
import sys
sys.path.append("../src/")
import matplotlib.pyplot as plt
import tensorflow as tf
import data_loader as dl
from convolutional_VAE import ConVae
```

Next the simulated data has to be loaded into memory, and we display four events to illustrate what the representation of the data looks like.

```
x_full, x_labelled, y = dl.load_simulated("128")

fig, axs = plt.subplots(ncols=4, figsize=(14, 5))
[axs[i].imshow(x_full[i].reshape((128, 128)), cmap="Greys") for i in
    range(4)]
[axs[i].axis("off") for i in range(4)]
plt.show()
```

[7]If the run-time is too slow the data can be replaced with the MNIST data, which is much smaller in terms of size per data-point

Figure 7.4: Selection of four simulated events in their XY-projection used as targets to reconstruct with the convolutional autoencoder.

We are now ready to define our model. To instantiate the model a convolutional architecture needs to be specified, in our implementation these are supplied as lists of integers, and a single integer specifying the number of layers. We'll use four convolutional layers and the simplest mode-configuration that uses no regularization on the latent space.

```
n_layers = 4
kernel_architecture = [5, 5, 3, 3]
filter_architecture = [8, 16, 32, 64]
strides_architecture = [2, 2, 2, 2]
pooling_architecture = [0, 0, 0, 0]

mode_config = {
    "simulated_mode":False, #deprecated, to be removed
    "restore_mode":False, #indicates whether to load weights
    "include_KL":False, #whether to compute the KL loss over the latent
    space
    "include_MMD":False, #same as above, but for the MMD loss
    "include_KM":False, #same as above, but K-means. See thesis for a
    more in-depth treatment of these
    "batchnorm":True, #whether to include batch-normalization between
    layers
    "use_vgg":False, #whether the input data is from a pre-trained model
    "use_dd":False, #whether to use the duelling-decoder objective
}

model = ConVae(
    n_layers,
    filter_architecture,
    kernel_architecture,
    strides_architecture,
    pooling_architecture,
    2, #latent dimension,
    x_full,
    mode_config=mode_config
)
```

When the model is defined two steps have to be completed before we train it. Firstly model has to be compiled, which constructs the forward pass and com-

putes the select losses over the outputs from the forward pass. Secondly the gradient-graph has to be computed, as it defines the iterative step for the optimization. For the former the model accepts two dictionaries that specify details of the forward pass; a dictionary graph_kwds which specifies the activation function and a dictionary loss_kwds regularization and the type of loss on the reconstruction, be it cross entropy or mean squared error. When the model is compiled it will print to the console a table of its configuration allowing the researcher to confirm that the model is specified correctly. This print is omitted for brevity but can be found in the notebook.

```
graph_kwds = {
    "activation": "relu",
    "output_activation": "sigmoid", # applied to the output, necessary
    for BCE
    "kernel_reg_strength": 1e-5
}
loss_kwds = {
    "reconst_loss": None # None is the default and gives the BCE loss
}
model.compile_model(graph_kwds, loss_kwds)
```

For the latter the model accepts an object of a TensorFlow optimizer, which should be uninstantiated, and arguments that should be passed to that optimizer object. In this example we choose an adam optimization scheme with $\beta_1 = 0.8$ and $\beta_2 = 0.99$ and a learning rate of $\eta = 1 \times 10^{-3}$. The parameters are explained in detail in section 2.10, but determine the weighting of the first and second moment of the gradient and the size of the change allowed on the parameters respectively.

```
optimizer = tf.train.AdamOptimizer
opt_args = [1e-3, ] #learning rate
opt_kwargs = {"beta1": 0.8, "beta2":0.99}
model.compute_gradients(optimizer, opt_args, opt_kwargs)
```

When the model is compiled and the gradients are computed it is ready to be trained, or alternatively a pre-trained model can be loaded into memory. Model training is performed by specifying a number of epochs to run for and the batch size to use for the optimization. Additionally the model takes a TensorFlow session object which it uses to run parts of the graph including the optimization operations. We also specify that the model should stop before the specified number of epochs with the earlystopping flag if the model converges or starts to overfit.

```
epochs = 200
batch_size = 150
earlystop = True
sess = tf.InteractiveSession()

lx, lz = model.train(
    sess,
    epochs,
    batch_size,
    earlystopping=earlystop
)
```

The training prints the value for the reconstruction, L_x, and latent L_z losses as well as the evaluation of the early-stopping criteria. This record is omitted for brevity, but can be seen in the notebook. After the model is trained we wish to inspect the reconstructions. Computing the reconstructions is done with the session object which feeds an input, in this case four events, to the model and retrieves a specific point on the graph. For this example we retrieve the reconstructions defined as the model output; model.output.

```
sample = x_full[:4].reshape((4, -1))
feed_dict = {model.x:sample}
reconstructions = model.sess.run(model.output, feed_dict)
reconstructions = reconstructions.reshape((4, 128, 128))
```

We reshape the reconstructions to the image dimension and plot them using the same block of code as we did for showing the original events, only adding another row.

```
fig, axs = plt.subplots(nrows=2, ncols=4, figsize=(14, 5))
[axs[0][i].imshow(x_full[i].reshape((128, 128)), cmap="Greys") for i in
    range(4)]
[axs[1][i].imshow(reconstructions[i], cmap="Greys") for i in range(4)]
[(axs[0][i].axis("off"), axs[1][i].axis("off")) for i in range(4)]
plt.show()
```

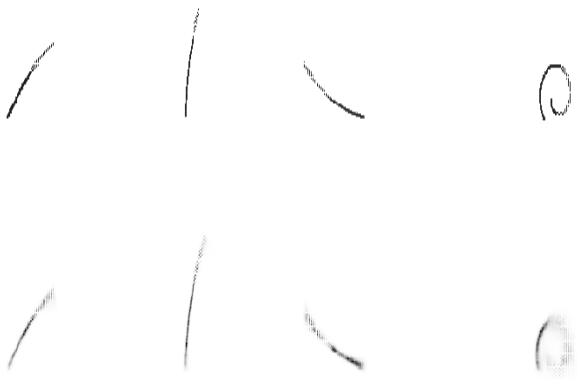


Figure 7.5: Showing four events and their corresponding reconstructions. The reconstructions faithfully reconstruct the artifacts from the simulation procedure but has a fuzzy quality common to the ELBO approximation.

From figure 7.5 we that while the reconstructions are fuzzy they capture the important parts of the input, notably the curvature of the proton. What remains now is the exploration and fitting of the latent space. We begin by computing the latent representation of the labelled subset of the data. This is done with the run_large method which does a computation a few elements at a time as the memory requirements of the computations scale very poorly. The method accept an argument for the session with which to run the required output, what output we wish to retrieve and the input needed to compute that output. In this case we wish to compute the latent representation and so our output is mpdel.z_seq[0]. To preserve homogeneity with the DRAW implementation the latent sample is stored as an iterable.

```
1
2  all_labelled = x_labelled.reshape((x_labelled.shape[0], -1))
3  latent_labelled = model.run_large(sess, model.z_seq[0], all_labelled)
4  fig, ax = plt.subplots(figsize=(14, 8))
5  classes = ["Proton", "Carbon"]
6  cm = matplotlib.cm.get_cmap("magma")
7  colors = [cm(0.3), cm(0.6), cm(0.85)]
8
9
10 for i in range(len(np.unique(y.argmax(1)))):
11     class_samples = latent_labelled[y.argmax(1) == i]
12     mag = np.sqrt((class_samples**2).sum(1))
13     marker = "^" if i == 0 else "."
14     c = "r" if i == 0 else "b"
15     ax.scatter(
16         class_samples[:,0],
17         class_samples[:,1],
18         label=classes[i],
19         alpha=0.5,
20         marker=marker,
21         color=colors[i],
22         #cmap=cmap
23     )
24 ax.set_title("Latent space of simulated AT-TPC data", size=25)
25 ax.tick_params(axis='both', which='major', labelsize=20)
26 ax.legend(loc="best", fontsize=20)
```

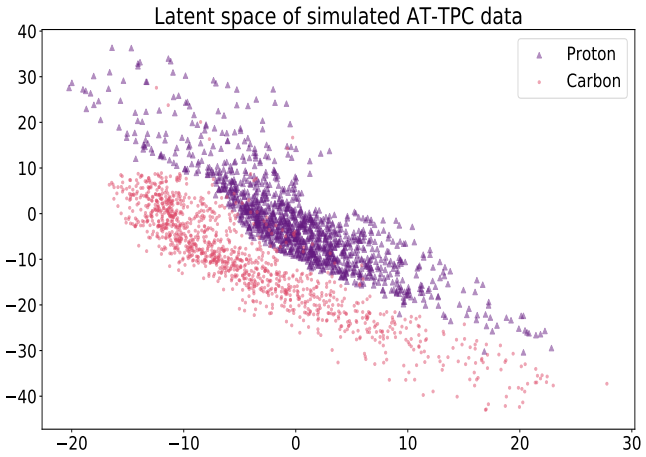


Figure 7.6: 2D latent space representation of the simulated AT-TPC data.

We visually confirm that the resulting latent space shown in figure 7.6 is clearly linearly separable.

7.4 Deep Recurrent Attentive Writer

Like the convolutional autoencoder discussed in the previous section, the deep recurrent attentive writer (DRAW) is implemented as a subclass of LatentModel. Consequently it follows the same formula and implements the _ModelGraph and _ModelLoss methods. We re-iterate from section 4.5 that the DRAW algorithm wraps the autoencoder structure in a recurrent framework, which means that it constructs a set of latent samples and iteratively builds a reconstruction of the input. Each new iteration is fed with an error image from the previous n iterations, making the reconstruction a conditional pixel distribution instead of an independent one which is the case for the ordinary convolutional autoencoder.

Our implementation is inspired by Eric Jang's explanation and implementation of the DRAW algorithm.

7.4.1 Computational graph

The construction of the model is analogous to the ConVae case. We supply configuration dictionaries when initializing the model. However, for the DRAW algorithm, we have to specify the read/write paradigm in addition to the number of pseudo timesteps, T, and the dimensions of the LSTM cells. Our implementation notably includes the option for an increased number of Long Term Short Term (LSTM) cells, but most important is the extension of the paired read and write functions to include the option for a paired set of convolutional networks. This contrasts with the original implementation from Gregor et al. [24] which uses a more traditional overlay of Gaussian filters on the image as a feature extraction tool.

The principal computation is wrapped in a for loop over pseudo timesteps which we label as DRAW.T. In this loop the attributes self.encode and self.decode are the sets of LSTM cells that act as the encoder and decoder networks. As input, the encoder takes features extracted from the canvas $\mathbf{x}_t$ and error image $\hat{\mathbf{x}}_t$ by the read function. Conversely, the write function uses the decoder output to add to the canvas.

The internals of this for-loop was written to follow the style of the set of equations that yield equation 4.34. To adhere to the idiosyncrasies of TensorFlow, we maintain an attribute self.DO_SHARE that ensures that parameters are shared between the iterations in the for-loop.

```
# excerpt from draw.py
# at https://github.com/ATTPC/VAE-event-classification
# from commit 6b64323
for t in range(self.T):
  # computing the error image
  if t == 0:
      x_hat = c_prev
  else:
      x_hat = self.x - c_prev

  """ Encoder operations  """
  r = self.read(self.x, x_hat, h_dec_prev)
  if self.batchnorm:
      r = BatchNormalization(axis=-1, center=True, scale=True,
    epsilon=1e-4)(
          r
      )
  h_enc, enc_state = self.encode(enc_state, tf.concat([r, h_dec_prev],
    1))
  if self.batchnorm:
      h_enc = BatchNormalization(
          axis=-1, center=True, scale=True, epsilon=1e-4
      )(h_enc)

  """ Compute latent sample """
  z  = self.compute_latent_sample(t, h_enc)

  """ Decoder operations """
  h_dec, dec_state = self.decode(dec_state, z)
  # dropout_h_dec = tf.keras.layers.Dropout(0.1)(h_dec, )
  if self.batchnorm:
      h_dec = BatchNormalization(
          axis=-1, center=True, scale=True, epsilon=1e-4
      )(h_dec)

  self.canvas_seq[t] = c_prev + self.write(h_dec)

  """
  ... code omitted for brevity
  """

  """ Storing and updating values """
  self.z_seq[t] = z
  self.dec_state_seq[t] = dec_state
  h_dec_prev = h_dec
  c_prev = self.canvas_seq[t]

  self.DO_SHARE = True
```

As most of the trappings around the model are the same as for the convolutional autoencoder, we do not re-tread that ground. This includes the convolutional read and write functions as they are functionally identical to the encoder and decoder structures in the convolutional autoencoder.

Instead, we will walk through the attentive part of the DRAW algorithm. The goal of the attentive component is to extract patches of the image. These are then passed to the encoder-decoder pair. The location and zoom of that patch are dynamically determined at each time-step. This procedure starts with the read function and its innermost functionality. Recall from equation 4.37 that

computing the filter-banks used for the extraction of image patches requires four parameters to be determined first. Once those four are determined we compute filter-banks F_x and F_y as they are defined in 4.41 and 4.42. These equations define matrices, and so we construct the grid over the exponential using the convenient function tf.meshgrid, which creates objects with the same dimension from different spacings. To explore the attention parameters outside the model class, we implemented a numpy version also. Conveniently those libraries are rather homogeneous, and as such the numpy is implemented in a manner almost one-to-one with the TensorFlow version.

```
# excerpt from numpy_filterbank.py
# at https://github.com/ATTPC/VAE-event-classification
# from commit 6416f96
def filters(self, gx, gy, sigma_sq, delta, gamma, N):
    i = np.arange(N, dtype=np.float32)

    mu_x = gx + (i - N / 2 - 0.5) * delta  #dim = batch_size, N
    mu_y = gy + (i - N / 2 - 0.5) * delta
    a = np.arange(self.H, dtype=np.float32)
    b = np.arange(self.W, dtype=np.float32)

    A, MU_X = np.meshgrid(a, mu_x)  #dim = batch_size, N * self.H
    B, MU_Y = np.meshgrid(b, mu_y)

    A = np.reshape(A, [1, N, self.H])
    B = np.reshape(B, [1, N, self.W])

    MU_X = np.reshape(MU_X, [1, N, self.H])
    MU_Y = np.reshape(MU_Y, [1, N, self.W])

    sigma_sq = np.reshape(sigma_sq, [1, 1, 1])

    Fx = np.exp(-np.square(A - MU_X) / (2 * sigma_sq))
    Fy = np.exp(-np.square(B - MU_Y) / (2 * sigma_sq))

    Fx = Fx / np.maximum(np.sum(Fx, 1, keepdims=True), eps)
    Fy = Fy / np.maximum(np.sum(Fy, 1, keepdims=True), eps)

    return Fx, Fy
```

With F_x and F_y determined the read-representation of the input is trivially computed from 4.43. The same procedure is repeated for the write-function with bespoke parameters determined for that function.

7.4.2 Computing losses

As with the computational graph most of the details about the implementation translate directly. The major difference is that the DRAW algorithm maintains T samples in the latent space and so forms a trajectory in that space. It's implementation is entirely analogous to the variational autoencoder loss and so we omit it for brevity.

7.4.3 Applying the framework

As we did for the convolutional autoencoder we walk through the configuration and fitting of the DRAW model to simulated AT-TPC data to illustrate its usage. This tutorial is also available in notebook form and can be retrieved from our repository and hosted locally.

The code which loads the data is identical to that of the convolutional autoencoder, and so we omit this part for brevity. We then begin by considering the attention hyperparameters. Recall from section 4.5 the number of filters N and the size of the glimpse δ. The combination of these parameters determine the size and zoom of the event, we illustrate the application of these filters in the notebook

```
ref_event = og_events[3]
gx = (128 + 1)/2 * 1.2
gy = (128 + 1)/2 * 1
sigma = 0.2
delta = 0.8
gamma = 1
N = 40
fb = filterbank(*x_full.shape[1:-1])
Fx, Fy = fb.filters(gx, gy, sigma, delta, gamma, N)
filtered = np.einsum("ijk, kl, lm->jm", Fy, ref_event,
    np.squeeze(np.transpose(Fx, [0, 2, 1])))

fig, ax = plt.subplots(ncols=2, figsize=(12, 10))
fig.suptitle("Feature extraction by filters", fontsize=25)
ax[0].imshow(np.squeeze(filtered), cmap="Greys")
ax[0].set_title("Extracted segment", fontsize=20)

ax[1].imshow(ref_event,  cmap="Greys")
ax[1].set_title("Original event", fontsize=20)
plt.tight_layout()
```

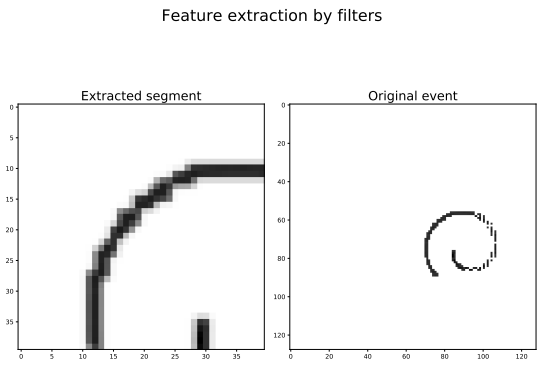


Figure 7.7: DRAW filters applied to a simulated event. Original on the right, and the extracted segment on the left

We can now determine the model with its hyperparameters. The syntax is very similar to the convolutional autoencoder, with the exception that that the user supplies parameters for a convolutional or attention based feature extraction. In code this is represented as:

```
T = 3
enc_size = 256
dec_size = 256
latent_dim = 3
use_attn = False
if use_attn:
    delta_write = delta
    delta_read = delta
    read_N = N
    write_N = N
    attn_config = {
        "read_N": read_N,
        "write_N": write_N,
        "write_N_sq": write_N ** 2,
        "delta_w": delta_write,
        "delta_r": delta_read,
    }
    conv_config = None
    use_conv = None
else:
    n_layers = 4
    kernel_architecture = [5, 5, 3, 3]
    filter_architecture = [32, 16, 8, 4]
    strides_architecture = [2, 2, 2, 2]
    pool_architecture = [0, 0, 0, 0]
    conv_config = {
        "n_layers":n_layers,
        "kernel_size":kernel_architecture,
        "filters":filter_architecture,
        "strides":strides_architecture,
        "pool":pool_architecture,
    }
    attn_config = None
    use_conv = True

mode_config = {
    "simulated_mode":False, #deprecated, to be removed
    "restore_mode":False, #indicates whether to load weights
    "include_KL":False, #whether to compute the KL loss over the latent
    space
    "include_MMD":False, #same as above, but MMD
    "include_KM":False, #same as above, but K-means. See thesis for a
    more in-depth treatment of these
    "batchnorm":False, #whether to include batch-normalization between
    layers
    "use_vgg":False, #whether the input data is from a pre-trained model
    "use_dd":False, #whether to use the duelling-decoder objective
}
```

The model is then instantiated with the following snippet of code:

```
model = DRAW(
    T,
    dec_size,
    enc_size,
    latent_dim,
```

```
    x_full.shape,
    beta=1,
    use_attention=use_attn,
    attn_config=attn_config,
    use_conv=use_conv,
```

The compilation and training code is identical to the convolutional autoencoder, and so we omit it for brevity. Upon completion of the training we can inspect the reconstruction to get an indication of whether the model has converged to satisfaction. The reconstructions are shown in figure 7.8. We observe in the reconstruction that the DRAW algorithm is picking up the hexagonal pad of higher sensor-pad density in the center of the image. Additionally, the model seems to be picking up on the artifacts introduced by the simulation.

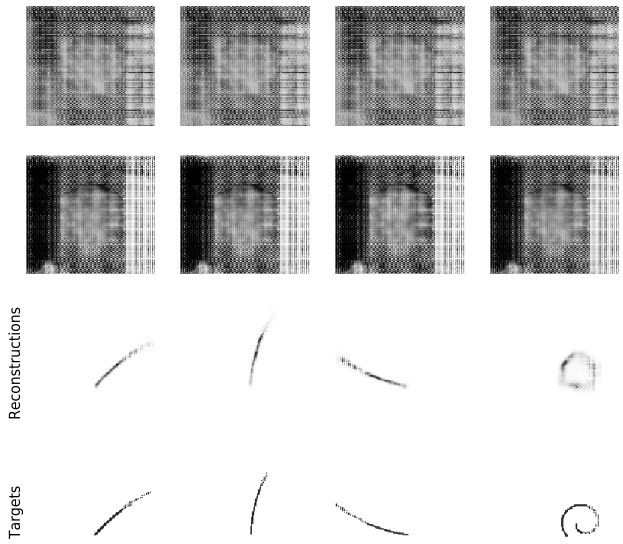


Figure 7.8: DRAW reconstructions on simulated data. Each column is the reconstruction of one event, with the original on the last row. The two first rows are then the canvass as it gets updated, and the second to last row is the finished reconstructions.

It is additionally interesting to inspect the latent space. We chose the latent space to be three dimensional to be able to easily visualize the space. We illustrate the latent space in figure 7.9. In the figure we do not observe the clear linear separability as in figure 7.6, and so we demonstrate the application of a linear regression classifier to the latent space.

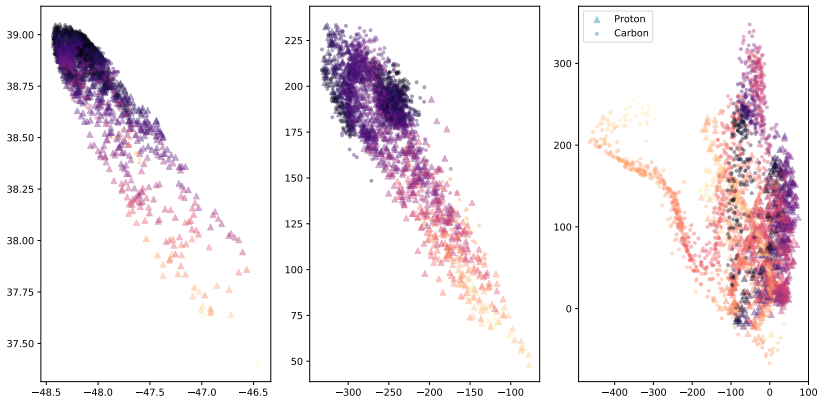


Figure 7.9: DRAW latent space for simulated data, with three time-steps and a three dimensional latent-space. The colors in the scatter-plot indicate the value in the third dimension. We observe some linear separability, but how well the classes separate is not clear.

```
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

lr_train, lr_test, y_train, y_test = train_test_split(latent_labelled, y)
lr_model = LogisticRegression(
  solver="lbfgs",
  class_weight="balanced",
  max_iter=1000,
)
lr_model.fit(lr_train, y_train)
print("Model accuracy: ", lr_model.score(lr_test, y_test))

$ Model accuracy: 0.94
```

7.5 Mixture of autoencoders

The last implementation we'll consider is the mixture of autoencoders (MIXAE) algorithm, which we introduced in section 4.6.2. We implement the model in the mixae_model class found in our repository. Recall that the MIXAE model consists of a set of autoencoders and a predictor network that assign the autoencoders to a cluster. The most interesting part of this architecture is the objective formulation. It consists of the three terms in equation 4.54. To reiterate the components of the loss we haave the ordinary autoencoder reconstruction loss coupled with predicted cluster belongings $p_j^{(i)}$. The clustering assignments are made by an auxiliary network from the concatenated latent vectors $\mathbf{Z} = [\mathbf{z}_1, \mathbf{z}_2, \ldots, \mathbf{z}_n]$.

The implementation of this algorithm is different from the previous discussions of the convolutional autoencoder, and the DRAW network. It is built with the high level Keras library, which builds on TensorFlow and other deep learning

back-ends to provide quicker prototyping of deep learning models. However, the convolutional autoencoders used in the algorithm are ConVae instances. Therefore the usage deviates somewhat from the previous section.

We begin by considering the loss functions for the sample and batch entropies. They are both relatively straight-forward implementations, the sample entropy is implemented as

```
def classification_entropy(y_true, y_pred):
    clipped_pred = tf.clip_by_value(y_pred, 1e-10, 1.0)
    log_pred = tf.math.log(clipped_pred)
    entropy = tf.keras.layers.multiply(
        [
            log_pred,
            y_pred,
        ]
    )
    entropy = - tf.reduce_sum(entropy, axis=0)
    return tf.reduce_mean(entropy)
```

and the batch entropy is coded as

```
def batch_entropy(y_true, y_pred):
    batch_pred_sum = tf.reduce_mean(y_pred, axis=0)
    log_pred_sum = tf.clip_by_value(batch_pred_sum, 1e-10, 1.0)
    log_pred_sum = tf.math.log(log_pred_sum)
    entropy_contents = tf.keras.layers.multiply(
        [
            batch_pred_sum,
            log_pred_sum
        ]
    )
    entropy_contents = tf.reduce_sum(entropy_contents)
    entropy_contents = entropy_contents - 0.9
    batch_ent = - tf.math.divide(1, entropy_contents)
    return batch_ent
```

As with the two previous sections we have written an accompanying notebook that uses and explains the model. We omit it from this section for brevity.

7.6 Hyperparameter search architecture

To tune the hyperparameters of the sequential and non-sequential autoencoders, we implement an object-oriented hyperparameter searching framework. The framework has two modular components: a search algorithm, and model generators which generate model configurations and trains the model.

A parent class, ModelGenerator, defines the variables and the type of model to be generated, e.g. one of ConVAE or DRAW, as well as helper functions to log performance metrics and loss values. The ModelGenerator class is treated as an abstract class and should never be instantiated on its own, only through its children. We have implemented one subclass of ModelGenerator for ConVAE and

DRAW model classes for this purpose.

We implement a simple random-search algorithm for applications in this thesis. The search is conducted with the RandomSearch class, which administrates the search, saves the results to file and handles untoward errors.

Searching can be done with a select sub-set of variables by specifying the static flag to the model-creator. This flag locks some parameters to pre-selected values and searches over the others. For the convolutional autoencoder the static flag holds the convolutional architecture, i.e. kernel sizes stride size and number of layers constant. Other flags are ours for a very wide search and vgg for a VGG16 like architecture.

Part III

Results

Chapter 8

Experimental setup and design

In this chapter, we describe the semi-supervised classification and clustering results obtained on the three active target time projection chamber (AT-TPC) datasets described in section 6.3. The simulated data is used to provide a benchmark for the upper bound on the performance of a given algorithm. This chapter is structured by task first, and algorithm second. As such, we will begin with a consideration of the semi-supervised algorithms before continuing with the clustering task.

We explore the models proposed in chapter 4 on two disparate tasks, one of semi-supervised classification and one of clustering. For each task, we evaluate the performance on each of the datasets using appropriate metrics, which were introduced in section 2.11.

The primary objective for the ^{46}Ar experiment was to identify resonant proton scattering events, but this thesis explores broader applications than this classification task. This broader picture is explicitly geared to the application of the models discussed in this thesis to other AT-TPC experiments. Following this argument, we measure individual class performance wherever appropriate.

The machine learning experiments conducted in this thesis was performed using the AI-Hub computational cluster at the University of Oslo. This resource consists of three machines with four RTX 2080 Nvidia GPU's (graphics processing unit) each. These cards have $\sim$ 10GB of memory available for the allocation of models.

8.0.1 Semi-supervised classification procedure

Our interest in the semi-supervised classification is three fold. Firstly, it provides a bound on the performance we can expect from the clustering algorithms, as the classifier problem is fundamentally an easier task. In addition to the bounding property, the semi-supervised task proved to be an excellent way to explore the types of algorithms we present in this thesis. Secondly, we wish to characterize the performance as a function of the number of available labelled samples. As

previously mentioned, one of the principal challenges with the AT-TPC detector is the cost of finding examples of the positive class, if at all possible. Thirdly, we wish to lay the groundwork for a transfer-learning approach to bridging AT-TPC experiments. Take, for example, resonant proton scattering, which could be a reaction of interest in a different experiment. Applying models trained to recognize these events in the ^{46}Ar experiment could be a feasible approach. The results from the semi-supervised classification experiments are presented in chapter 9

The semi-supervised task is to train an autoencoder model on a large dataset, and evaluate the class separation of the latent space using a logistic regression classifier. We use a logistic regression classifier as we wish to measure qualities of the latent space, and not of the classification per se. To provide an additional benchmark for our algorithms, we measure the performance of a logistic regression classifier using the latent space of a pre-trained image classifier model, which has shown to be effective in the classification of events from the ^{46}Ar experiment [17].

Intrinsic to the measurement of the semi-supervised performance is the budgeting of how many labelled samples one can feasibly extract. Moreover, the principal limitation of the semi-supervised approach is the assumption that the researchers can identify the event class(es) of interest positively. It is then interesting to quantify the change in model performance as a function of how many labelled samples the classification model has to train on. Bear in mind that the representation that the classification model sees is still trained on the full set of events for a given dataset.

For reference, the models are described in terms of their hyperparameters in table D.1 for the convolutional autoencoder and table D.2 for the DRAW-analogues. To determine the best hyperparameters, we perform on the order of $\sim 10^2$ runs, each taking on the order of $\sim 10^1$ minutes on a single Nvidia 2080 GPU. For each configuration of the algorithms, we train a classifier on a subset of the labelled set. Subsequently, this classifier is evaluated on the remainder of the labelled data to estimate the out of sample (OOS) error. The best configuration is then lastly re-trained, and we evaluate the OOS error with k-fold cross validation as outlined in section 2.11.

8.0.2 Clustering procedure

While the semi-supervised classification task provides context and understanding to the AT-TPC experiments, the clustering procedure is a direct attempt at solving challenges associated with these experiments. First and foremost is the challenge of acquiring labelled data of the event of interest. The clustering results are presented in chapter 10

In contrast to the semi-supervised task, the clustering objective presumes that we have access to no labelled data. However, since the purpose of this

thesis is in large part exploratory, we will measure performance on the labelled data provided. We explore the performance of the deep clustering algorithms described in section 4.6 as well as a K-means approach using the latent space of a pre-trained image classifier model.

The clustering task is intrinsically a more challenging task than that of semi-supervised classification, and so more time was spent exploring algorithms to determine which were suitable for clustering AT-TPC data. Before training each of the algorithms in section 4.6 was first configured to reproduce the results from their respective papers. Subsequently, they were applied to the AT-TPC data, with their autoencoder hyperparameters selected empirically from the semi-supervised task. This was done to enable a focus on the clustering components of the algorithm. In particular, we focus on the weighting parameters of the mixture of autoencoders (MIXAE) clustering algorithm.

Chapter 9

Classification results

To prime our discussion on clustering algorithms for AT-TPC data, we first consider the easier problem of semi-supervised classification. The goal of this analysis is to determine whether we can construct latent spaces that separate the known event-types from the ^{46}Ar experiment. We investigate the latent space of a pre-trained model and two different autoencoder structures. Subsequently, we evaluate each of these models on three datasets: simulated, filtered, and full AT-TPC events. The evaluation is performed by training a logistic regression classifier on the latent samples, and we measure performance by the $f1$ score.

The training procedure for classification using a semi-supervised regime necessitates the same strict separation of labelled data for the classification step as when considering ordinary classification tasks. Details on the modelling pipeline are found in section 5. We tuned all models excepting the baseline pre-trained VGG model with the RandomSearch architecture, which searches in a semi structured way over all the parameters given in table D.1. As a benchmark, we start by measuring the performance using just the pre-trained VGG16 representation of the labelled data of each dataset. The two proposed representation algorithms are then presented with results for each dataset for comparison.

9.1 Classification using a pre-trained model

As outlined in chapter 5, the pre-trained VGG16 network will serve as the baseline comparison for this work. We chose VGG16 as it has demonstrated successful performance on labelled AT-TPC data from the ^{46}Ar experiment.[17]. For each labelled dataset listed in section 6.3, a logistic regression model was fit to the respective VGG16-representation. To estimate the variability in the result a K-fold cross validation approach was taken, with $K = 5$. We report test-$f1$ scores for each class and average for the classification. The results are listed in table 9.1

Additionally, the scarceness of labelled data begs the question of how many labelled samples is needed to achieve reliable classification. To estimate this

Table 9.1: Logistic regression classification results using the VGG16 representation of the labelled data listed in section 6.3. The error is given as the standard deviation in the $f1$ score over the $K = 5$ folds of cross validation.

	Proton	Carbon	Other	All
Simulated	0.999 $\pm 1.014 \times 10^{-3}$	0.999 $\pm 1.029 \times 10^{-3}$	N/A	0.999 $\pm 1.022 \times 10^{-3}$
Filtered	0.918 $\pm 5.108 \times 10^{-2}$	0.69 $\pm 4.267 \times 10^{-2}$	0.908 $\pm 2.359 \times 10^{-2}$	0.839 $\pm 3.911 \times 10^{-2}$
Full	0.84 $\pm 4.653 \times 10^{-2}$	0.668 $\pm 4.860 \times 10^{-2}$	0.89 $\pm 1.730 \times 10^{-2}$	0.799 $\pm 3.748 \times 10^{-2}$

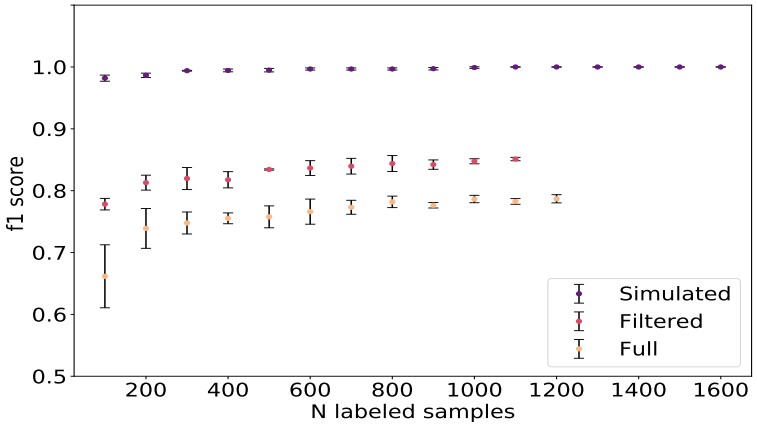


Figure 9.1: VGG16 performance on increasing subsets of labelled data. The error-bars represent the $\pm 1\sigma$ interval from the variability in the selection of subsets. The y axis $f1$ score is computed as the unweighted average of the sample classes.

relationship, we sample increasing subsets of the labelled data, each containing the previously selected data. For each selection, a logistic regression model is fit, and a $f1$ score is computed. This procedure is sensitive data selection effects, and so a variability estimate is computed by running this procedure $N = 100$ times. We report the mean and standard deviation for each dataset. The result of this analysis is shown in figure 9.1

For comparison, we also explore visualizations of the latent space of each of the models in this thesis. The latent spaces are all however in high dimensional spaces, and so we utilize a combination of a linear mapping along axes of variation (PCA) and stochastic mapping via a manifold (t-SNE). The latter renders the axes completely uninterpretable as well as making relative distances incomparable [46]. Distance in t-SNE space is still useful, as we can still get indications of class belonging in the latent space from the visualization. The principal component

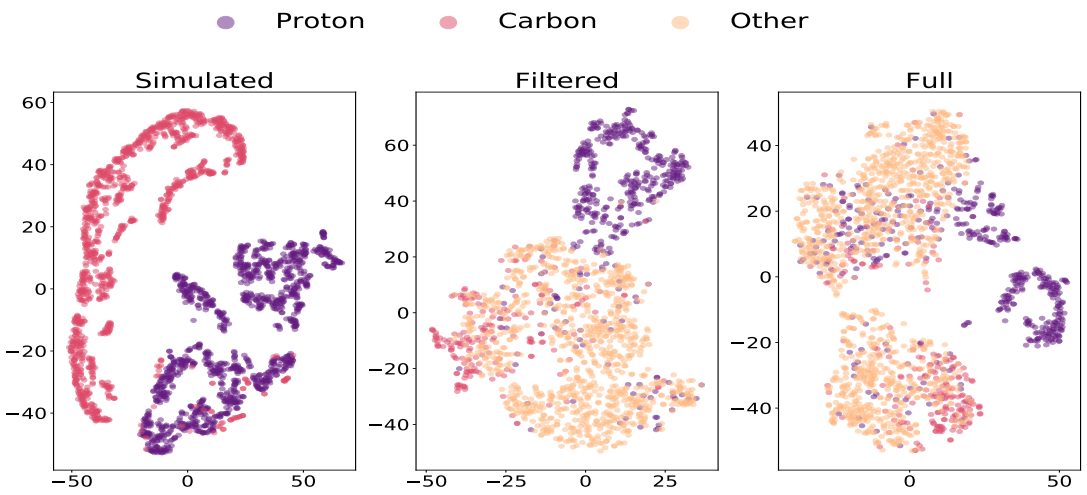


Figure 9.2: Visualization of the latent space from the VGG16 model on the three different data-sets. There is very little mixing in general between the proton class and the other two for all datasets. While the carbon and other categories seem to be mixed for the full and filtered experimental data. The axes have arbitrary non-informative units.

in the t-SNE projection is the perplexity of the model, essentially controlling how many neighbours the algorithms consider, recommended values lie between $perp = 5$ and $perp = 50$ [46]. We chose a perplexity value of $perp = 15$ for all the visualizations. The latent space of the pre-trained VGG16 model is shown in figure 9.2 and demonstrates an evident separation of the proton class with the carbon and "other" classes.

9.2 Convolutional Autoencoder

To test the hypothesis that classification can be improved by using unsupervised methods to estimate the data distribution is investigated by using a convolutional autoencoder trained end-to-end on the data distribution, and then using the latent representation as input to a logistic regression classifier on the subset of data that has labels. This pipeline is outlined in chapter 5, and the data are described in section 6.3. The convolutional autoencoder has three configurations that we report results from.

(Ar0): End-to-end training on data using kernel and filter architectures in a naive manner with decreasing kernel sizes, increasing filter sizes and a mirrored encoder-decoder structure

(Ar1): Using the VGG16 network to compute a representation of the data which

is compressed by one or more dense layers and finally reconstructed to the original image by a naively constructed decoder.

Choosing an architecture for the convolutional autoencoder is the principal challenge to solve. We want to estimate if the reconstruction and optional latent losses relate to the classification accuracy achieved by the logistic regression classifier.

add plot with reconst/loss vs f1 scores

To aid in the understanding of the choice of architecture, we compare the optimal architectures given a dataset. In the event that one dataset finds a configuration of lesser complexity that was not present in the others, a verification run was computed with that configuration to ensure the validity of the performance measurement.

Table 9.2: Hyperparameters that gives the strongest classifier performance on the three simulated, filtered and full datasets.

Hyperparameter	Value		
	Simulated	Filtered	Full
Convolutional parameters:			
Number of layers	3	6	6
Kernels	$[17, 15, 3]$	$[9, 7, 5, 5, 5, 3]$	$[11, 11, 11, 11, 5, 3]$
Strides	2	2	2
Filters	$[2, 16, 64]$	$[8, 4, 16, 16, 16, 16]$	$[16, 16, 16, 16, 32, 32]$
Network parameters:			
Activation	ReLU	LeakyReLU	LeakyReLU
Latent type	MMD	MMD	None
Latent dimension	150	50	100
β	0.01	100	100
Optimizer parameters:			
η	1×10^{-5}	0.0001	0.001
β_1	0.73	0.72	0.25
β_2	0.99	0.99	0.99

For the best models found by random search we re-compute the performance

	Proton	Carbon	Other	All
Simulated	0.969 $\pm 7.350 \times 10^{-3}$	0.968 $\pm 7.326 \times 10^{-3}$	N/A	0.969 $\pm 7.338 \times 10^{-3}$
Filtered	0.876 $\pm 2.447 \times 10^{-2}$	0.605 $\pm 6.682 \times 10^{-2}$	0.905 $\pm 2.782 \times 10^{-2}$	0.795 $\pm 3.970 \times 10^{-2}$
Full	0.744 $\pm 3.146 \times 10^{-2}$	0.618 $\pm 8.593 \times 10^{-2}$	0.851 $\pm 1.403 \times 10^{-2}$	0.738 $\pm 4.381 \times 10^{-2}$

Table 9.3: Logistic regression classification $f1$ scores using the (Ar0) architecture. The standard error is reported from a $K = 5$ fold cross validation of the logistic regression classifier.

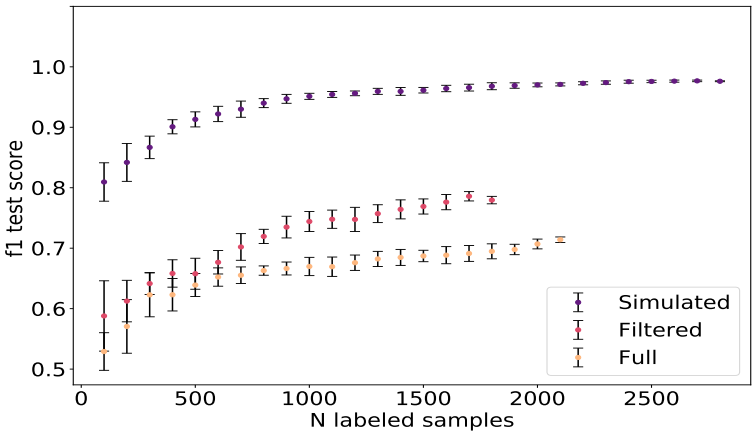


Figure 9.3: Latent space classification performance with a logistic regression classifier on a (Ar0) representation of each dataset. For each dataset a random subsample is drawn and iteratively added to in increments of $n = 100$ datapoints. To estimate the variance of this procedure we repeat the procedure $N = 10$ times.

with $K = 5$ fold cross validation on the logistic regression classifier. We begin with the model using no information from the VGG16 benchmark, i.e. configuration (Ar0). It shows strong performance on the classification task for all datasets. The results are listed in table 9.3

Furthermore, we estimate the performance of the best models as a function of the number of labelled samples it sees. We select a random subsample from the labelled dataset and iteratively add to that dataset in increments of $n = 100$ samples. This procedure is repeated a total of $N = 10$ times to estimate the variability as a function of the selection process. The resulting runs are shown in figure 9.3

Lastly we wish to qualitatively inspect the latent space with a 2D visualization of the latent space. We firstly process the latent space with a $D = 50$ dimensional

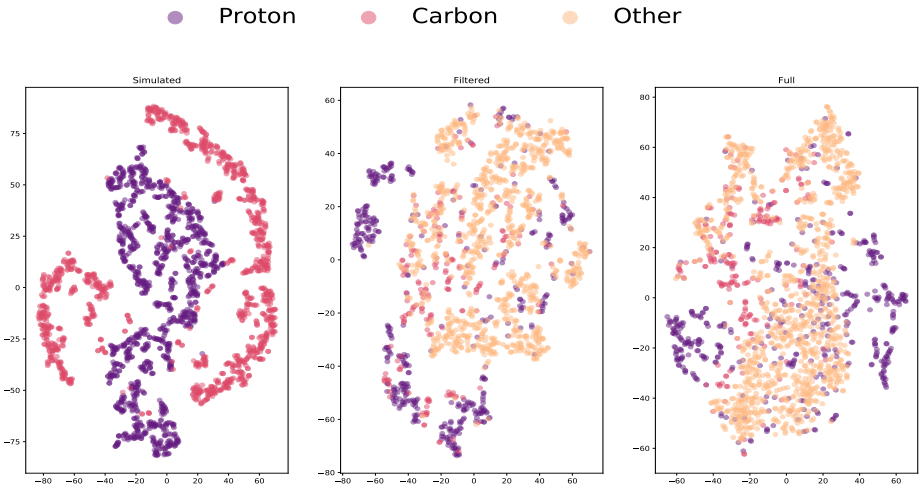


Figure 9.4: Visualizing the latent space of an (Ar0) trained autoencoder. The mapping is a t-SNE projection of the latent space to two dimensions. We re-iterate that the axes have non-informative units.

	Proton	Carbon	Other	All
Simulated	0.998 $\pm 1.848 \times 10^{-3}$	0.998 $\pm 1.883 \times 10^{-3}$	N/A	0.998 $\pm 1.866 \times 10^{-3}$
Filtered	0.896 $\pm 3.955 \times 10^{-2}$	0.645 $\pm 7.290 \times 10^{-2}$	0.881 $\pm 3.520 \times 10^{-2}$	0.807 $\pm 4.922 \times 10^{-2}$
Full	0.86 $\pm 2.983 \times 10^{-2}$	0.657 $\pm 8.574 \times 10^{-2}$	0.888 $\pm 2.551 \times 10^{-2}$	0.802 $\pm 4.702 \times 10^{-2}$

Table 9.4: Logistic regression classification $f1$ scores using the (Ar1) architecture. The standard error is reported from a $K = 5$ fold cross validation of the logistic regression classifier.

PCA and subsequently project to two dimensions with a t-SNE mapping of the data. This visualization is shown in figure 9.6 and illustrates a good separation between the proton classes in general.

We repeat this process with using the VGG16 representation as initial input to the autoencoder model. This is configuration (Ar1). In the same manner as for the naive implementation we search over hyper-parameters, with the difference in the dense layer(s) included that transforms the VGG16 representation to the autoencoder latent space.

Each of the configurations found by the random search was then evaluated with $K = 5$ fold cross validation to produce estimates of the $f1$ score, listed in table 9.4

Furthermore, we estimate the performance of the model as a function of the number of latent samples it is shown. In exactly the same manner as we did for

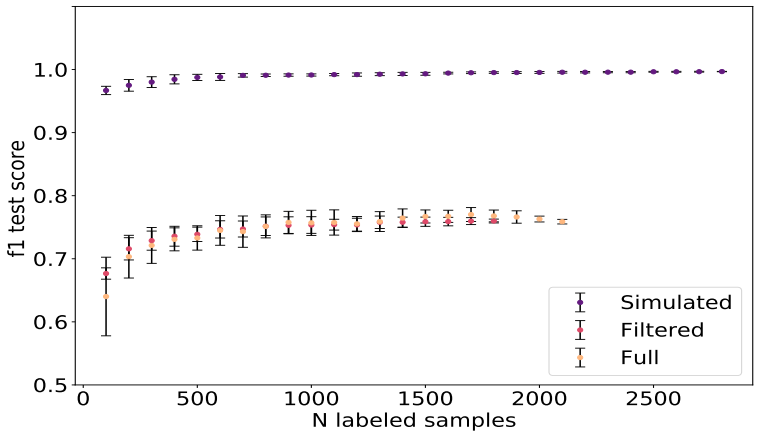


Figure 9.5: Latent space classification performance with a logistic regression classifier on a (Ar1) representation of each dataset. For each dataset a random subsample is drawn and iteratively added to in increments of $n = 100$ datapoints. To estimate the variance of this procedure we repeat the procedure $N = 10$ times.

	Proton	Carbon	Other	All
Histogram	0.781 $\pm 4.580 \times 10^{-2}$	0.638 $\pm 6.482 \times 10^{-2}$	0.863 $\pm 2.487 \times 10^{-2}$	0.761 $\pm 4.516 \times 10^{-2}$
Net charge	0.708 $\pm 1.794 \times 10^{-2}$	0.578 $\pm 6.869 \times 10^{-2}$	0.796 $\pm 2.899 \times 10^{-2}$	0.694 $\pm 3.854 \times 10^{-2}$

Table 9.5: Logistic regression classification $f1$ scores using the (Ar0) architecture, with a duelling decoder addition to the objective. This analysis was performed on full events, and not using a VGG representation. The standard error is reported from a $K = 5$ fold cross validation of the logistic regression classifier.

the (Ar0) architecture. The results of this search is shown in figure 9.5

Lastly, for the architecture we project the latent space for comparison with the non-tuned VGG16 representation.
In addition to the architectures explored above, we investigate the effect of adding a duelling decoder to the objective. We provided two distinct auxiliary representations to reconstruct, grounded in the physics of the experiment. The chosen representations were the charge distribution heuristically chosen to be at the high end of the distribution and the net charge deposited during the event. We perform the analysis on the full events and use their original representation.

The results of those experiments are included in table 9.5. We immediately observe that the addition of the duelling decoder to the objective has a non-zero impact on the performance of the linear classifier on the latent space.

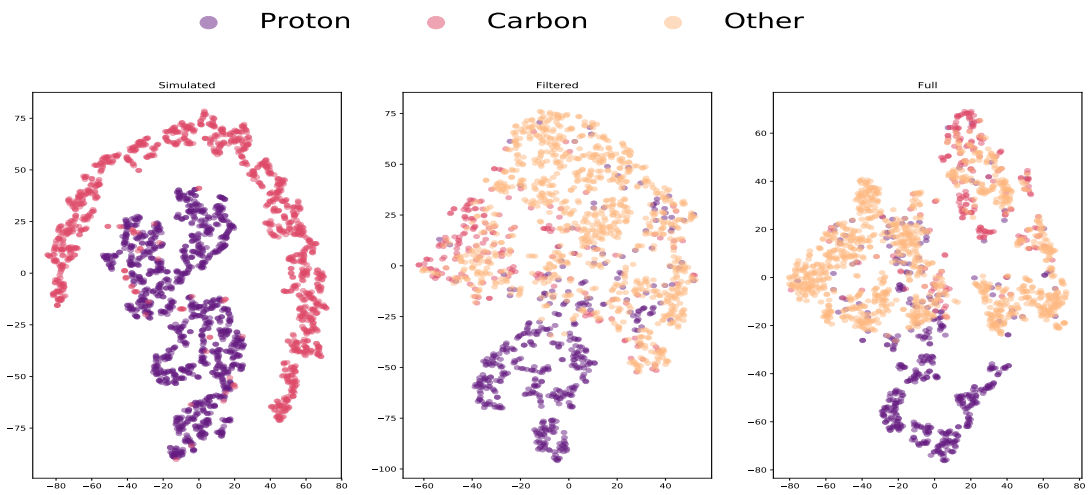


Figure 9.6: Visualizing the latent space of an (Ar1) trained autoencoder. The mapping is a t-SNE projection of the latent space to two dimensions.

9.3 Deep Recurrent Attentive Writer

From the previous section on the convolutional autoencoder, it is clear that we can encode class information in autoencoder latent spaces. It is then interesting to investigate whether we can improve the separation in other ways. In this section, we demonstrate the results from a recurrent model: the deep recurrent attentive writer (DRAW). We discuss the DRAW algorithm in section 4.5.

We begin by considering the semi-supervised classification results. Given the recurrent sampled generated by the DRAW algorithm, we compute classification performance using a flattened representation of the sequence of latent samples. As the model produces T samples from the latent space, we concatenate these, which then acts as input to a logistic regression classifier.

Additionally, we investigate the performance as a function of the number of latent samples, and qualitatively probe the latent space by its t-SNE projection.

The hyperparameters of the algorithm were determined with a random search architecture, equivalently to how we determined the values for the convolutional autoencoder. To keep the search-space feasible, we empirically froze some of the hyperparameters pertaining to the architecture in the read-write function pairs. For the convolutional architecture we used four layers with stride $s = 2$ and kernel sizes $k = [5, 5, 3, 3]$. For the attention parameters, we specified a glimpse size of $\delta = 0.8$ and searched over the number of Gaussian filters N. We used a leaky rectified linear unit and applied the ADAM optimizer in all the model experiments.

The simulated and full datasets achieved optimal performance with a convolutional read/write configuration, while the filtered data showed the strongest

Table 9.6: Hyperparameters that yielded the optimal performance on the semi-supervised task for the DRAW algorithm

Hyperparameter	Simulated	Filtered	Full
Recurrent parameters:			
Dim(encoder)	128	512	256
Dim(decoder)	64	512	256
Network parameters:			
Latent type	MMD	None	MMD
Latent dimension	100	100	10
β	10	None 100	
Batchnorm	False	False	True
Optimizer parameters:			
η	1×10^{-3}	1×10^{-5}	1×10^{-2}
β_1	0.92	0.94	0.81
β_2	0.99	0.99	0.99

Table 9.7: Logistic regression classifier performance on the latent space of the DRAW algorithm.

	Proton	Carbon	Other	All
Simulated	0.971 $\pm 5.259 \times 10^{-3}$	0.97 $\pm 6.067 \times 10^{-3}$	N/A	0.97 $\pm 5.663 \times 10^{-3}$
Filtered	0.86 $\pm 3.769 \times 10^{-2}$	0.613 $\pm 4.098 \times 10^{-2}$	0.899 $\pm 3.435 \times 10^{-2}$	0.791 $\pm 3.768 \times 10^{-2}$
Full	0.77 $\pm 2.378 \times 10^{-2}$	0.616 $\pm 8.465 \times 10^{-2}$	0.85 $\pm 2.331 \times 10^{-2}$	0.745 $\pm 4.391 \times 10^{-2}$

performance with attention parameters. For the filtered data the search yielded filter values $N_{read} = 15$ and $N_{write} = 20$. Moreover, for the full and simulated data, the optimal value for the number of convolutional filters was 8 per layer for all layers. The remainder of the hyperparameters are presented in table 9.6.

We begin by considering the $f1$ scores of the logistic regression classifier on the latent samples. These scores are included in table 9.7, where we note that there seems to be no large deviation from the non-sequential autoencoder.

Additionally, we wish to characterize the latent space by how many latent samples it takes to achieve this optimal performance. We present these performance records in figure 9.7.

Lastly, we wish to describe the latent space in some detail. Like the VGG16 latent space and the non-sequential convolutional autoencoder we project the latent space to a low-dimensional space. The projection is made with the t-SNE

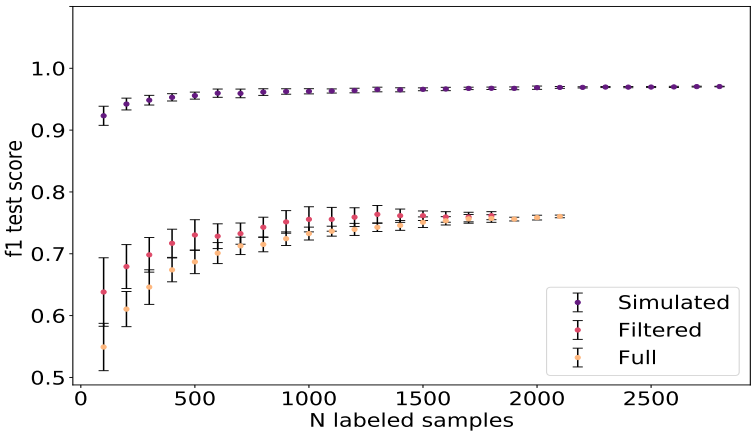


Figure 9.7: Performance of the logistic regression algorithm on the three datasets as a function of number of latent samples. The latent samples are produced with the DRAW algorithm using the hyperparameters presented in table 9.6

algorithm, and the results are presented in figure 9.8

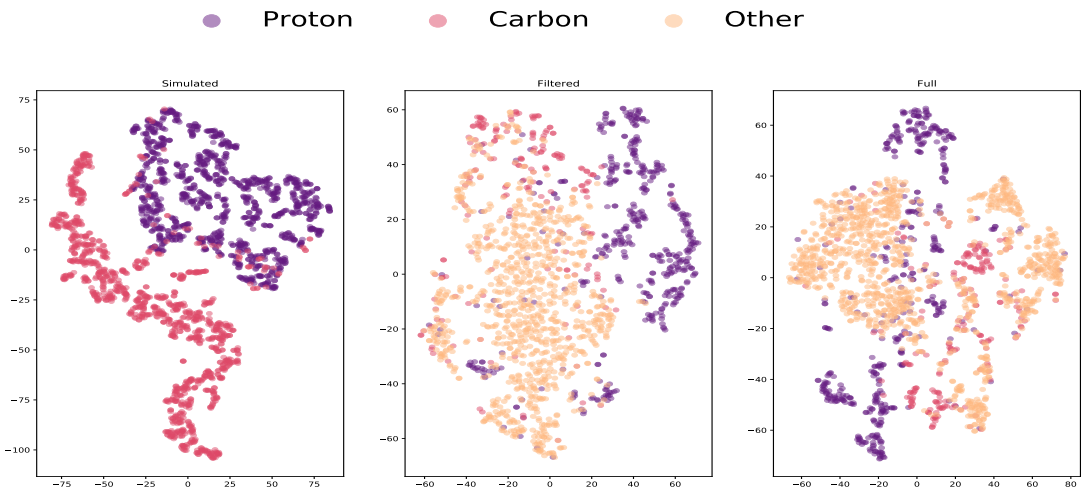


Figure 9.8: t-SNE projection of the DRAW latent space. The latent samples are produced with the DRAW algorithm using the hyperparameters presented in table 9.6

Chapter 10

Clustering of AT-TPC events

The principal challenge in the AT-TPC experiments that we're trying to solve is the reliance on labelled samples in the analysis. While the ^{46}Ar experiment has conveniently visually dissimilar reaction products, which facilitates supervised learning, this is not necessarily the case for other experiments. The ^{46}Ar experiment provides a convenient example where we can then explore unsupervised techniques. In this chapter, we explore the application of clustering techniques to events represented in latent spaces.

We begin by exploring a naive K-means approach on the latent space of a pre-trained network. Subsequently we investigate other clustering methods and two autoencoder based clustering algorithms as outlined in section 4.6.

This chapter builds on the previous results from semi-supervised classification. We observe that we are able to construct high quality latent spaces, which facilitates the investigation of clustering techniques.

The approach for clustering of events is different than the semi-supervised approach in two meaningful ways. First, it's a harder task, as we'll demonstrate. This necessitates a more exploratory approach to the problem. Second, as a a consequence of the challenge the focus will be a bit different than for the semi-supervised approach. We will still utilize the same architectures and models starting with a search over the parameter space over which we measure the performance using the adjusted rand score (ARS) and accuracy defined in section 2.13 and 2.11.1, respectively.

As with the chapter on the semi-supervised results we start with considering the VGG16 pre-trained model as a benchmark.

Lastly, we note that the focus of this work is largely on discovering possible avenues for further research, which requires a broad scan of possible avenues rather than a rigorous analysis of one specific model.

10.1 Clustering using a pre-trained model

As with chapter 5, we also use the VGG16 pre-trained network as a baseline for the clustering performance. We begin by considering a classical K-means approach to clustering. However, the output from the VGG16 network is very high dimensional at some $\sim 8e3$ floats. One of the primary concerns is then the curse of dimensionality, where the ratio of distances goes to one with increasing dimensionality [58]. However, one of the central caveats to this finding is that the elements are uniformly distributed in the space. It is then possible that all the class information lies in some sub-space of the latent data. To investigate this we perform clustering analysis using the full representation, and using the 10^2 principal components only.

10.1.1 K-means

We begin by investigating the K-means clustering algorithm on the VGG16 latent space. As in the previous chapter the VGG16 model is pre-trained on the imagenet dataset creating a set of vectors $\mathbf{x} \in \mathbb{R}^{8192}$. To cluster we use scikit −learn implementation of the K-means algorithm, with default parameters [23]. The results of the clustering runs are included in table 10.1. We observe that we are able to attain near perfect clustering on simulated data, and that there is a sharp decline in performance as we add noise by moving to the filtered and full datasets.

Table 10.1: K-means clustering results on AT-TPC event data. We observe that the performance predictably decreases with the amount of noise in the data.

	Accuracy	ARI
Simulated	0.97	0.89
Filtered	0.74	0.39
Full	0.59	0.17

In addition to the performance measures reported in table 10.1 it is interesting to observe which samples are being wrongly assigned. We achieve this by tabulating the assignments of samples relative to their ground truth labels. From these tables we can infer which classes are more or less entangled with the others. We tabulate the results for each dataset is in figure 10.1. We observe that the proton class is consistently assigned in a pure cluster. Purity is inferred by how much spread there is in the column between the ground truth labels. A high quality cluster will, in addition to being pure, also capture most entries the class represented by the cluster. For example, consider the row corresponding to

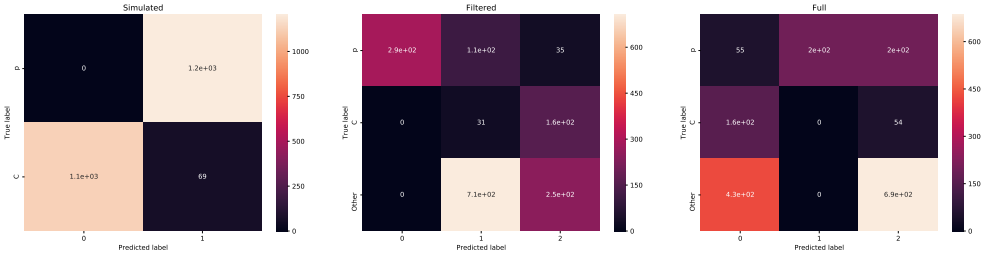


Figure 10.1: Confusion matrices for the K-means clustering of simulated, filtered and full AT-TPC events. The true labels indicate samples belonging to the p (proton), carbon (C), or other classes.

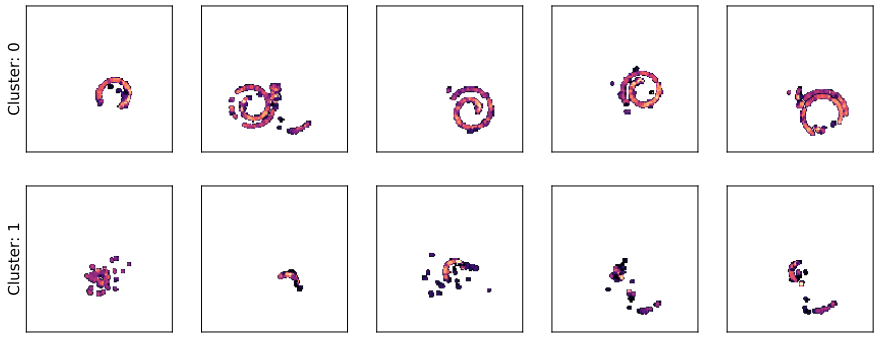


Figure 10.2: Illustrating a sample of proton events from different K-means clusters from the filtered dataset. Each row belongs to a single cluster corresponding to the filtered confusion matrix in figure 10.1

the proton class in 10.1. The column corresponding to the largest entry in the proton row has zero other predicted classes in it. From this, we conclude that the proton cluster is a high quality, high purity cluster.

We repeat this analysis using a PCA dimensionality reduction on the latent space of the VGG16 model. This is done to estimate to what degree the class separating information is encoded in the entirety of the latent space, or in some select regions. The results from the PCA analysis were virtually identical to the results sans the PCA, and so we omit them for brevity.

Furthermore, we wish to further characterize the clusters presented in figure 10.1. To achieve this we sample from the proton samples belonging to different clusters for the filtered and full data.

In addition to the results presented in this section, we performed clustering with a number of different algorithms included in the scikit −learn package. None of them provided any notable differences from the K-means results or were sig-

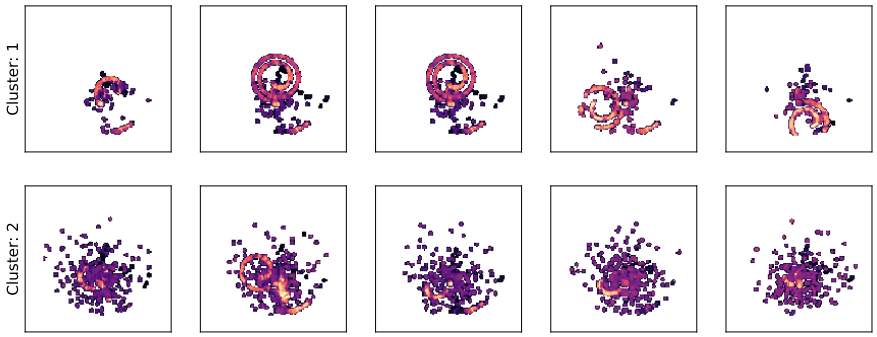


Figure 10.3: Illustrating a sample of proton events from different K-means clusters from the full dataset. Each row belongs to a single cluster corresponding to the full confusion matrix in figure 10.1

nificantly worse. Notably, the DBSCAN algorithm failed to provide any useful clustering results. We find this important as one of the major drawbacks of K-means, and the deep clustering algorithms presented in section 4.6, is that the are all dependent on pre-determining the number of clusters. This is not the case for DBSCAN.

10.2 Deep clustering algorithms

In the previous section we demonstrated a potential for clustering of events from an AT-TPC experiment. To build on this result we will in this section explore the application of the mixture of autoencoders (MIXAE) algorithm introduced in section 4.6. The other algorithm introduced in section 4.6, deep clustering with convolutional autoencoders (DCEC), consistently collapsed to just one cluster for all datasets.

In the MIXAE algorithm the hyper-parameters to adjust are all the ordinary parameters that we introduced in table D.1. In addition to those parameters come the weighting of the loss terms: θ, α and γ. These weighting parameters are attached to the reconstruction loss, sample entropy and batch-wise entropy respectively. [45] note that these parameters are very important for the model performance and so we focus primarily on these.

We empirically freeze the convolutional autoencoder hyperparameters to compress the original 128×128 to a 8×8 pixel grid using four convolutional layers. The parameters chosen for the autoencoders are listed in full in table 10.2.

Table 10.3: Hyperparameter grid for the MIXAE loss weighting terms. The grid is given as exponents for logarithmic scales.

Parameter	Grid	Scale
θ	$[-1, 5]$	Logarithmic
α	$[-5, -1]$	Logarithmic
γ	$[3, 5]$	Logarithmic

Table 10.2: Hyperparameters selected for the autoencoder components of the MIXAE algorithm, see table D.1 for a full description of the parameters.

Hyperparameter	Value
Convolutional parameters:	
Number of layers	4
Kernels	[3, 3, 3, 3]
Strides	[2, 2, 2, 2]
Filters	[64, 32, 16, 8,]
Network parameters:	
Activation	LReLu
Latent type	None
Latent dimension	20
β	N/A
Batchnorm	False
Optimizer parameters:	
η	10^{-3}
β_1	0.9
β_2	0.99

Since there are then only three remaining hyperparameters we choose to perform a coarse grid-search as described in section 7.6.

10.2.1 Simulated AT-TPC data

To train the MIXAE clustering algorithm, we use the large simulated dataset with $M = 80000$ points, evenly distributed between proton- and carbon-events. The algorithm is trained on a subset of 60000 of these samples, and we track performance on the remaining 20000 events.

The grids selected for the search are listed in table 10.3. The search yielded an optimal configuration with

$$\theta = 10^{-1}, \tag{10.1}$$
$$\alpha = 10^{-2}, \tag{10.2}$$
$$\gamma = 10^{5}. \tag{10.3}$$

Finally, for these parameters we re-ran the algorithm $N = 10$ times to investigate the stability of the algorithm. The results are reported in figure 10.4. We observe that while the algorithm can achieve very strong performance, with an $ARI > 0.8$, it fluctuates strongly with repeated experiments. As mentioned in section 4.6.2 the batch-entropy has a second minimum when the cluster confidences are near equal. It is this behavior we observe in 10.4.

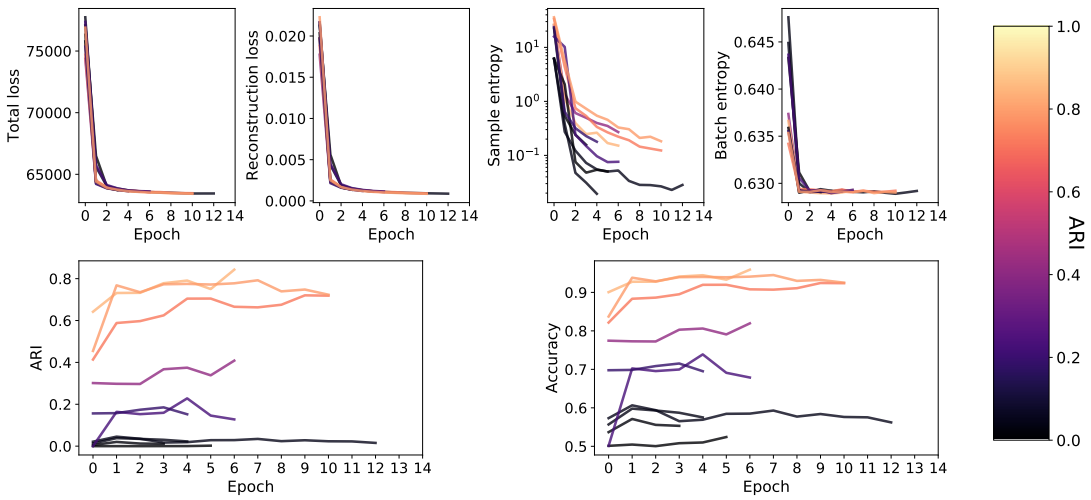


Figure 10.4: Performance for the MIXAE model on simulated AT-TPC data. In the top row the loss components are plotted for each run, and in the bottom row the adjusted rand index (ARI) and clustering accuracy are shown. Each run is color-coded with the ARI achieved at the end of the run.

10.2.2 Filtered AT-TPC data

We repeat the optimization steps in the previous section for the filtered AT-TPC data. The exception being that we allow the algorithm to train on the labelled samples. Beginning with a wide grid equal to the grid used for the simulated data we searched over all parameter configurations to find promising values. We then performed a local search around these values to pin-point the hyperparameter configuration. This search yielded the optimal hyper-parameters

$$\theta = 10^{1}, \tag{10.4}$$
$$\alpha = 10^{-1}, \tag{10.5}$$
$$\gamma = 3.162 \times 10^{3}. \tag{10.6}$$

The results of the runs are included in figure 10.5. We observe ghat the highest performing models reach an $ARI > 0.5$, which is higher than the performance achieved by the K-means algorithm applied to the VGG16 latent space.

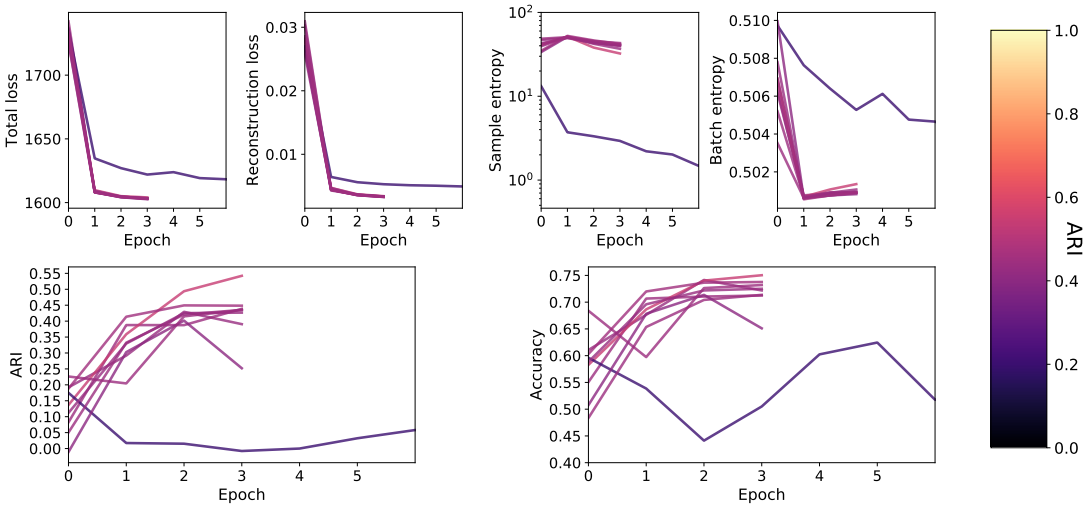


Figure 10.5: Performance for the MIXAE model on filtered AT-TPC data. In the top row the loss components are plotted for each run, and in the bottom row the adjusted rand index (ARI) and clustering accuracy are shown. Each run is color-coded with the ARI achieved at the end of the run.

10.2.3 Full AT-TPC data

As in the two previous sections, we repeat the same procedure of iterative grid searches on the MIXAE loss-weights. Each configuration is re-run a total of $N = 10$ times to capture fluctuations in the performance before a final selection is made on the hyperparameters. For the full dataset the MIXAE hyperparameters converge to the same values as for the clean data, i.e.

$$\theta = 10^{1}, \tag{10.7}$$
$$\alpha = 10^{-1}, \tag{10.8}$$
$$\gamma = 3.162 \times 10^{3}. \tag{10.9}$$

As with the previous datasets we include a plot of the loss curves and performance measures for $N = 10$ runs with the same loss-weight parameters, shown in figure 10.6.

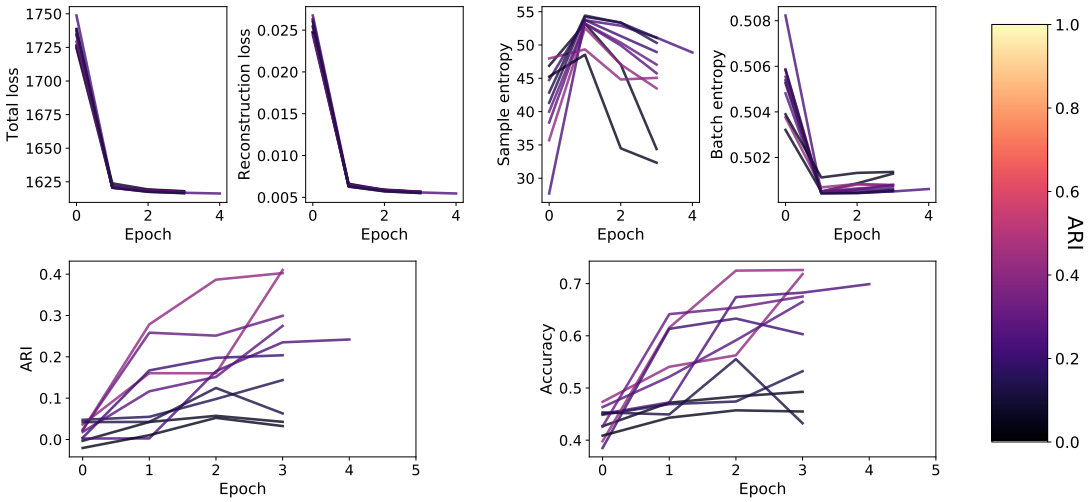


Figure 10.6: Performance for the MIXAE model on un-filtered AT-TPC data. In the top row the loss components are plotted for each run, and in the bottom row the adjusted rand index (ARI) and clustering accuracy are shown. Each run is color-coded with the ARI achieved at the end of the run.

10.2.4 Comparing performance

It is also interesting to compare and contrast the clustering results from the MIXAE model with those of the VGG16+K-means outside the fairly abstract accuracies and rand scores. It is especially interesting to compare the cluster assignments, as they can inform further research efforts in the clustering of AT-TPC events. We illustrate the clustering with confusion matrices, which are shown in figure 10.7. From these matrices we observe that the MIXAE applied to the clean data correctly clusters the noise events. Additionally, it identifies two proton clusters. We observe that these proton clusters are both less pure than the VGG16+K-means clusters, and that there does not seem to be a visually meaningful difference between these clusters. The latter is inferred from figure 10.8.

Applied to the real data the MIXAE correctly separates the proton class, however it is unable to separate the carbon events from the amorphous noise events or from the proton cluster.

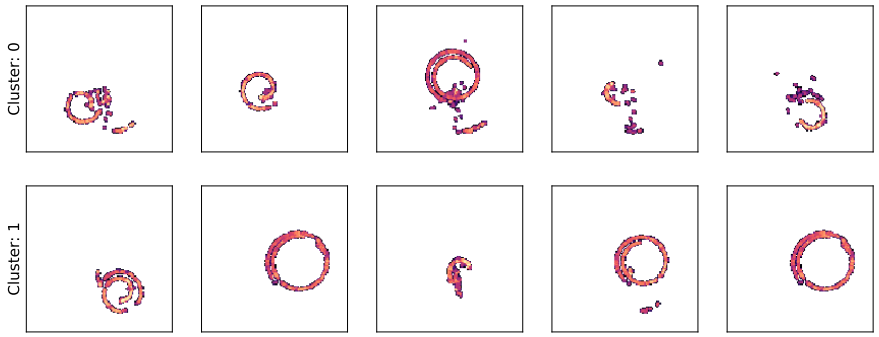


Figure 10.8: Selection of carbon events belonging to different clusters. Each row represents one of the clusters for the filtered data as shown in figure 10.7. There seems to be no clear distinction between these rows.

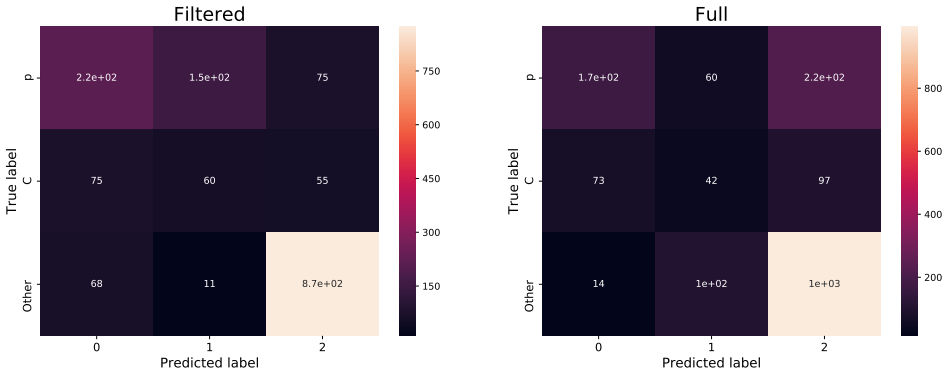


Figure 10.7: Confusion matrices for the MIXAE clustering algorithm on filtered and full AT-TPC events. The true labels indicate samples belonging to the p (proton), carbon (C), or other classes.

Lastly we wish to further investigate if there are systematic differences between proton events that were placed in different clusters for the clean and full data. From figure 10.7 we see that the MIXAE algorithm creates two proton clusters for the filtered data, and places about fifty per-cent of the proton events in a cluster with the amorphous "other" events. We extract some proton events from these clusters to inspect whether systematic differences occur.

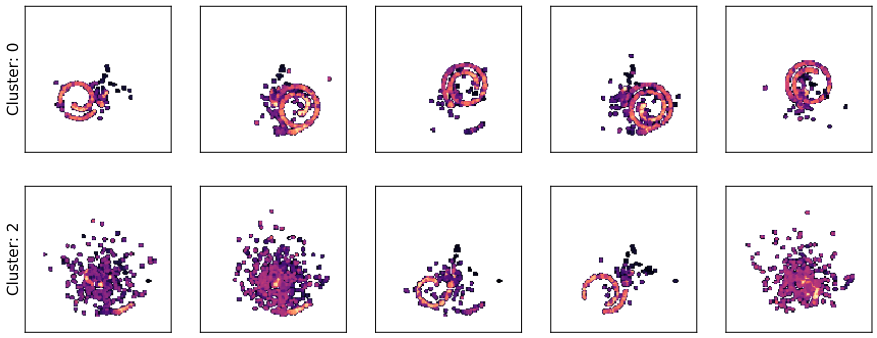


Figure 10.9: Selection of proton events belonging to different clusters. Each row represents one of the clusters for the full data as shown in figure 10.7. We observe a tendency for the more noisy events to be placed in the "other" cluster

Part IV

Discussion, Conclusion and Future Prospects

Chapter 11

Discussion

The primary aim of this chapter is to frame our findings by the goals for our analysis, as presented in sections 8.0.1 and 8.0.2. We divide the discussion into topics of task. First, we will consider the classification performance of our two implemented algorithms on the three different datasets: simulated, filtered and full AT-TPC data. Second, we discuss the unsupervised clustering performance on the same datasets.

11.1 Semi-supervised classification of AT-TPC events

To prime or discussion of the semi-supervised classification results, we briefly restate the goals for the analysis: The core task we aim to accomplish is to quantify the model performance as in terms of the available labelled data. Furthermore, we wish to characterize the latent spaces produced by the different algorithms qualitatively. The performance is contextualized by the work of Kuchera et al. [17], who introduced the application of pre-trained models to AT-TPC data. Lastly, the semi-supervised performance serves as a proof-of-concept for the construction of high-quality latent spaces in an AT-TPC experiment. This proof-of-concept spurred the implementation of autoencoder based algorithms for clustering of AT-TPC data.

As a benchmark, we trained a linear model on the data representations from a pre-trained VGG16 network. This high-performing model from the image analysis community has seen successful applications to the same experimental data; it then follows that it is also a reasonable comparison for our methods [17].

Additionally, we explore the performance of our convolutional autoencoder models trained end-to-end on AT-TPC data. In this section we compare and contrast the pre-trained model and the autoencoder models on the aforementioned analysis objectives.

11.1.1 Pre-trained networks

From table 9.1 it is evident that the VGG16 network, even when trained on an auxiliary task, projects the AT-TPC data to a linearly separable space. In addition to the end-point f1 scores, the performance as a function of labelled data paints a convincing picture of the latent space. This picture is painted by figure 9.1, from which we see that for the simulated data, the linear classifier performs almost perfectly when trained on 100 labelled samples. Additionally, the variance is low for the filtered data but increases significantly for the full data. The relative difference in class occurrences may explain a part of this discrepancy. However, the major difference is in the amount of noise present in the events. An increased variability as a function of noise is an expected consequence, which indicates that pre-processing of data is an important step when employing pre-trained models to AT-TPC data.

11.1.2 Convolutional autoencoder

Using the RandomSearch framework, we were able to find a network configuration for the convolutional autoencoder (CAE) that shows solid performance on the simulated data. With a total $f1$ score of > 0.96, the simulated data strongly indicate that the CAE will perform well for our real data.

Furthermore, we observe that for the filtered and full datasets, the autoencoder encodes a high-quality representation of the data with $f1$ scores > 0.7. We also note that the latent space shows strong class-separability with and without a latent loss. This means that the CAE class separability is more closely tied to the reconstruction objective than the latent prior. It is, however, clear that the maximum mean discrepancy (MMD) regularization is preferable to the variational autoencoder objective. This preference indicates that sample wide measures of regularization are more capable of encoding class-information than the point-based KL objective. This finding is in agreement with the arguments presented by Zhao et al. [38].

From comparing the results in table 9.3 and 9.4 to those in table 9.1 it is clear that while the autoencoder achieves strong class separation the reconstruction objective is somewhat misaligned with a class-separating representation. The misalignment can be attributed to the fact that the pre-trained VGG16 network creates latent spaces which explicitly aims to separate classes. This explicit nature is notably absent from the reconstruction, and latent objectives, of the CAE. Further cementing this argument is the improvement in classification performance when using the VGG16 features as input to the autoencoder. In summary, we argue that when labelled data is present, using pre-trained networks for classification of events is the recommended procedure.

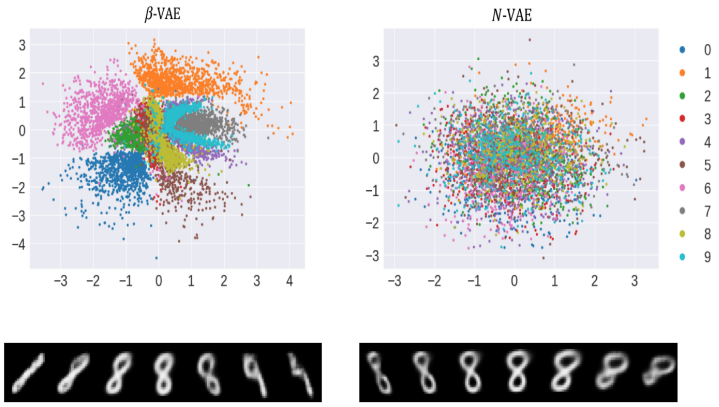


Figure 11.1: Demonstrating the difference of capturing class and feature information in the latent space. On the left, the β-VAE pushes the autoencoder to a representation favouring encoded class information in the latent space. On the right, the natural VAE which has a separate classification from the latent sample exhibits a tight distribution of the latent space. The sub-plots demonstrate the traversal in the output space when sampling from the latent x-axis. Figure copied from Antorán and Vivolab [59].

Constructing a salient latent space

We also want to understand why the MMD cost is preferable, in terms of classifier performance, to the VAE objective. This understanding can be brought about by considering a semi-supervised VAE developed by Antorán and Vivolab [59]. In their work, the authors show that the mapping of the latent space to an isotropic Gaussian distribution, as the Kullback-Leibler objective aims to achieve, contributes to the washing out of class information. However, they also show that by predicting a class assignment separate from the latent sample, the KL-divergence strongly encourages feature level information. They demonstrate these qualities on the MNIST handwritten digits dataset, where the feature information is described as, e.g. stroke thickness or skew when drawing a number, while class information is the more esoteric "five"-ness of all drawings of the number five. These differences are illustrated in figure 11.1. On the right, the latent space and generated samples from the natural clustering algorithm. On the left, the same but for the β-VAE. The subplots under the latent distributions demonstrate reconstructions of a traversal along a latent axis, clearly showing the difference between feature- and class information.

Indeed their semi-supervised objective improves the generative aspect of the variational autoencoder by tightening the latent distribution, i.e. achieving a density in the latent space without holes. Tight distributions of latent samples improve the generative properties by having no regions of the prior for which the generated sample is poorly defined.

A variation of this algorithm may apply to transfer-learning problems with AT-TPC data. The challenge with the application stems from the cluster assignments, which requires training with some supervision. A natural application to AT-TPC data is to pre-train this algorithm on simulated data and investigate the clustering on records from an experiment.

11.1.3 Mode collapse and the duelling decoder

An additional challenge attached to the latent space is the threat of mode collapse, as described in section 4.3.1. This problem is investigated in detail by Seybold et al. [42], where the authors propose the duelling decoder objective as a solution to this problem. The duelling decoder adds a second reconstruction term to the objective. This second reconstruction is optimized over a different representation of the data, e.g. reconstructing the edges in an image, its intensity histogram or other transformations. For applications in physics, this is a promising approach as it allows the inclusion of physical properties to the optimization. From the results in table 9.5 as compared to those in table 9.3 there are indications that the duelling decoder architecture contributes to more salient representations in the latent space.

For future work on autoencoder based clustering and classification, the addition of a duelling-decoder objective holds promise. Our results show a marginal increase in performance when predicting the charge histogram of an event, compared to just reconstructing the two-dimensional representation of the event. Further work is required on what representations generate the most salient representations in the latent space. One such representation which we did not explore in this thesis is the Hough's transform of the data. As previously described in section 6.3.4, the Hough's transform is a representation of the geometry in the event. Since the curvature of the track is dependent on the velocity of the electron[1], an accurate description of the geometry would then encode critical physical aspects of the event.

11.2 Classifier performance

From table 9.3 it is clear that while the autoencoder architecture, (Ar0), is able to capture class information, it does not outperform the pre-trained VGG16 in terms of linear separability of its latent space. Adding the VGG16 representations to the autoencoder increased the performance, as we see from table 9.4, but did not increase it beyond the pure VGG16 classifier. Additionally, the performance in terms of the number of latent samples was not improved by the autoencoder

[1] In the AT-TPC configuration with an applied magnetic field, the electron would experience a Lorentz force

architecture when compared to the pre-trained model. The variance is the principal measure of improvement in the performance with low-volume labelled data, as well as the mean performance relative to the end-point mean. From figures 9.1, 9.5 and 9.3 we can infer that the autoencoder algorithms do not meaningfully increase classification performance compared to the pre-trained model in terms of the number of labelled samples.

Visually inspecting the latent space in figure 9.6, we observe the same types of structures for all architectures; local high-density areas that are strung out in the space. For the filtered and full datasets, we observe a slight degradation of proton separation with the naive autoencoder compared to the pure VGG16 representation, which we confirm by the proton $f1$ scores in tables 9.3. Carbon is consistently hard to separate from the amorphous "other" category, and there is no indication that the autoencoder is able to separate them better than the pure VGG16 latent already does. It is important to note that the ^{46}Ar experiment was not designed to detect carbon reactions, and so the confusion with noise sources is not necessarily a cause for concern.

Using the VGG16 representation as an initial encoded representation, the (Ar1) architecture improved performance substantially from the (Ar0) architecture. Moreover, the autoencoder trained on the VGG16 representations showed increased performance on the proton class by lowering the variance by about $\sim 64\%$ from the pure VGG16 classifier. This is evident from table 9.4 and 9.1. When trained on few labelled samples the (Ar1) architecture achieved comparable results to the pre-trained model. We observe from figure 9.3 and 9.5 that the (Ar1) autoencoder exhibits the same patterns of error as the pre-trained model, with very small deviations from the mean for the filtered and simulated data and an error in the second decimal for the full data.

We note that in figure 9.3 and 9.5 the asymptotic performance is not expected to tend to the mean represented their corresponding tables. In the performance estimate as a function of labelled samples, the hold-out set is chosen arbitrarily and held constant, as we are more interested in variability and the shape of the curve than the end-point itself. Conversely, the K-fold approach varies the hold-out set to approximate the expected performance given changes in how we select data to train the classifier.

We then investigate why the reconstruction objective does not aid in classification. We look to a discussion on the principal component analysis (PCA) in regression analysis from Jolliffe [60], where he remarks that the major axes of variation may not be the ones carrying the information needed for regression[2]. We posit that the reconstruction focuses the optimization on these major axes of variation, and if these do not carry the salient class-separating information, the latent space may not carry this information either. Contributing to this argu-

[2]The PCA algorithm is a dimensionality reduction algorithm that projects the data long smartly chosen axes of variation. We refer to section 2.12 for details on the PCA algorithm.

ment is the observation that adding the duelling decoder improves the classifier performance.

In summary, we observe that the autoencoder does not improve classification compared to the pre-trained model latent space. However, there are still interesting avenues to investigate autoencoder models. In particular, the duelling decoder objective could provide an interesting link to the geometry, charge distribution or other physical properties for the latent space.

11.3 Clustering of AT-TPC events

From the semi-supervised results, we know that we can construct class separating latent spaces. The next challenge is then to expand that insight into the unsupervised clustering of events. Clustering is an intrinsically harder task, owing to the unsupervised nature of the problem. This leads to differing requirements on the representation of the data, as a linearly separable latent space is not necessarily suitable for clustering.

The most naive approach to clustering our data is to apply a simple clustering algorithm on the latent representation of an algorithm which we know separates our data. Using a pre-trained model is advantageous because they are generally well studied, but may offer less flexibility. A more sophisticated approach is to use algorithms designed for clustering. These will invariably carry their own assumptions and pitfalls that we have to be wary of.

In this thesis, we have applied a K-means algorithm to the latent space of a pre-trained VGG16 model as our naive approach. Which means that the K-means assumptions bound this approach. These include that the clusters are distributed as Gaussians in the high-dimensional latent space and that we have a good idea of the number of clusters. We also investigate the more sophisticated mixture of autoencoder (MIXAE) and deep convolutional embedded clustering (DCEC) models, the latter of which fails to cluster AT-TPC data to any degree. However, the MIXAE model achieves promising results, but assumes uniformly distributed classes and requires a good idea of the number of clusters present in the data.

The assumption that we have a good idea of the number of clusters present is problematic for AT-TPC data. In the filtered and full datasets, the amorphous "other" class probably consist of a variety of event types. These were empirically placed in the same class for classification purposes when labelling the data, precisely because the researchers were unable to identify these events.

In this section, we will explore and discuss the clustering of AT-TPC data using a pre-trained network and the MIXAE model. We begin by considering the results from clustering on the pre-trained VGG16 model latent space.

11.3.1 Clustering with a pre-trained network

What is immediately clear from the results in table 10.1 is that the VGG16 latent space is able to capture difference between event classes, achieving an adjusted rand index (ARI) score of > 0.88 on the simulated data against the ground-truth labels. Recall that the ARI is a measure of clustering similarities, adjusted for chance. A particularly interesting aspect of the results in table 10.1 is the ability to cluster despite the very high-dimensional latent space. As previously mentioned, Euclidean vector distances can be uninformative in high dimensional spaces, as shown by [58]. We can explain a part of this behaviour by the fact that a majority, on the order of $\sim 80\%$, of the entries in the latent vectors are zero-valued. However, that leaves some $\sim 1.5 \times 10^3$ non-zero elements per sample, which is still very much a high dimensional representation. For the Euclidean distances to have discriminatory power, we must then infer that the space contains some very dense regions in-between regions of minimal density. In such a configuration, the ratios between distances will not tend to unity, and we find some success with K-means clustering. The results for the filtered and full data show worse performance than for the simulated data. However, these results are still promising for the application of clustering algorithms to pre-trained network latent spaces.

To further characterize the performance, we then consider the confusion matrices shown in figure 10.1. From these matrices, we can infer some interesting properties about the types of events placed in different clusters. It seems that the clustering of the filtered data identifies proton events that occur closer to the detector plane, and so have less developed spirals. We confirm this by visually inspecting the events in those clusters, as illustrated in figure 10.2. Similarly, we infer that clustering of full events segments the proton class by the noisiness of the event, placing the more noisy proton events with the amorphous "other" category. We confirm this argument by inspecting the events contained in each cluster, illustrated in figure 10.3.

One of the major strengths of this application is the consistency of the clustering algorithms. The K-means algorithm showed very little, if any, variation in performance when re-running the algorithm.

11.3.2 Clustering with autoencoder based models

In addition to the results from the VGG16+K-means algorithm, we implemented and applied the DCEC and MIXAE algorithms to the ^{46}Ar data. The former failed to produce any salient clusterings of AT-TPC data. We are confident in the implementation of the algorithm as we are able to reproduce the original author's results on the MNIST handwritten digits dataset. Some value was gained from the experiments, as the mode of failure was informative for the choice to implement the MIXAE algorithm. The DCEC would consistently collapse all

samples to one cluster. This exact mode of failure is addressed by the batch entropy loss term introduced in the MIXAE algorithm. Recall from equation 4.53 that the batch entropy encourages the clustering probabilities to be diverse. This also explains the relative imbalance between the weighing of the losses that we found in section 10.2.

We then turn to the MIXAE performance shown in figures 10.4, 10.5 and 10.6. From these figures, we observe that while the MIXAE algorithm is able to achieve better performance than the VGG16+K-means algorithm, it suffers from severe stability problems. The results for the simulated data are noteworthy in this regard. To better understand the out of sample performance for the clustering algorithm, we segregate the training and test data entirely for the simulated data. Coupled with the high variability in the simulated performance, we infer that this method may not generalize well to unseen data. Being unsupervised, this is not as big a caveat for the MIXAE as it would be for a supervised algorithm.

Additionally, the best performing algorithms are systematically not those with the lowest loss-values. Indeed it seems that convergence to a perfect one-hot prediction of cluster belonging is indicative of the training having failed. This is especially clear for the filtered and full datasets whose performance are shown in figures 10.5 and 10.6 respectively. A defining feature of the loss-plots is in the sample loss, as defined in equation 4.52. We observe that for the filtered and full datasets, the sample entropy is quite large for the models with the highest performance, with values $S_{sample} > 10$. Together with the observation that the batch-entropy is at a minimum, we conclude that the cluster assignments are near uniform. For comparison, a perfect one-hot representation has a sample loss of $\sim 10^{-1}$. The initially most plausible candidate for this behaviour is the class imbalance in the data, recall from table 6.2 that the amorphous "other" class represents $\sim 60\%$ of the full and filtered data. Indeed, we observe that the high performing models on the simulated data have an order of magnitude smaller sample loss compared to the models applied to full and filtered data.

To further characterize the performance, we inspect the properties of the clustering results achieved by the highest performing models on the filtered and full datasets. The confusion matrices listed in figure 10.7 initially displays some of the same properties we observed for the VGG16+K-means algorithm. To confirm this behavior, we display some select events from the proton class in figures 10.8 and 10.9. From these plots, we confirm that while the clustering of the full dataset is in agreement with the VGG16+K-means clusters. That is, we observe noisy proton events being placed in the cluster with the "other" class. This is not the case for the filtered dataset. In the filtered data, the different clusters of proton events do not seem to be visually distinguishable.

11.3.3 Comparing clustering methods

Both the pre-trained and autoencoder based clustering methods hold promise. As is the case for classification, the ease of application of the pre-trained methods is a significant boon, but their static nature creates a hard to breach cap on their performance. We also observe an impressive purity in the proton clusters, as shown in figure 10.1, for both the filtered and full datasets. Notably, this purity is absent from the MIXAE clustering results. The increase in performance from the VGG16+K-means is then a result of a stronger segmentation of the "other" class of events.

It remains to be seen if the algorithms explored in this thesis are capable of separating events in future experiments with more similar tracks. A related avenue for future research is then to combine the autoencoder methods discussed here with a duelling decoder objective on Hough transformed events. This representation is promising as it can explicitly encode the geometry in the event.

For on-going experiments with the AT-TPC, our recommendation for clustering is to employ a pre-trained model combined with K-means. Performance validation is, unfortunately, a necessary step. Some positive identifications of events are thus needed to validate the clustering.

In this thesis, we have demonstrated strong performance with the MIXAE algorithm. However, further inquiry is needed to investigate the stability of the algorithm. Other avenues of potential interest are the coupling of clustering with a duelling decoder objective, as well as other autoencoder based clustering algorithms. Lastly, there is a need for the coupling of unsupervised performance metrics to measures of performance against ground truth labels.

Chapter 12

Conclusions and Future Work

In the present work, we explored the segmentation of active target time-projection chamber (AT-TPC) events in neural network latent spaces. This exploration is necessary because traditional methods are both computationally prohibitively expensive and can not be applied to events with broken tracks. Specifically, the goal was to implement autoencoder based models for semi-supervised classification and clustering. We compared these models with a classic pre-trained model from the image analysis community.

Two tasks were proposed to contribute to the exploration of AT-TPC events: a semi-supervised objective which describes the necessary volume of labelled data, and a clustering objective which measures the quality of segmentation without labelled data.

To solve the semi-supervised problem, we implemented two autoencoder-based algorithms: a convolutional autoencoder model with the capacity for two different latent space regularisation, and the sequential deep recurrent attentive writer (DRAW) model. We trained these models on three different sets of AT-TPC data and found the following:

- A convolutional autoencoder can linearly segment its latent space by event type when trained on the ^{46}Ar data.
- Good class separability can be achieved both with and without latent regularisation. When regularised, we found better segmentation in models trained with a Gaussian mixture maximum mean discrepancy loss, than those trained with a variational autoencoder loss.
- The recurrent DRAW model does not offer meaningful improvements to the convolutional autoencoder performance on the semi-supervised task.

Moreover, neither the DRAW algorithm nor the convolutional autoencoder outperformed the pre-trained VGG16 as a function of the available labelled data. This discrepancy indicates that while the reconstruction objective encourages

class separability in the latent space, it does not do so to the degree that a classification objective does even when the classification objective is over a different dataset.

Avenues of academic interest for further research include the construction of new representations for the duelling decoder objective, as well as models that combine autoencoders with generative adversarial networks. Lastly, we analysed two-dimensional projections of AT-TPC events in this work, expanding to include the full three-dimensional cold contribute additional insight.

To address the clustering task, we implemented two algorithms: the deep convolutional embedded clustering (DCEC) algorithm and the mixture of autoencoders algorithm. As in the semi-supervised objective, we compared these algorithms with the performance of a pre-trained VGG16 network. We showed that the pre-trained network latent space could be combined with a simple k-means algorithm for clustering of AT-TPC events. With the VGG16+K-means algorithm, we achieved convincing results on simulated data, as well as promising segmentation of the full and filtered data. Especially notable was the consistent purity of the proton event cluster. With the autoencoder based MIXAE algorithm, we found that an increase in the performance on the clustering task from the VGG16+K-means approach. In conclusions, we found the following for the clustering task:

- Using a K-means algorithm on the VGG16 latent space, we demonstrate strong clustering of simulated data. Additionally, this approach consistently finds high purity proton clusters for both filtered and full ^{46}Ar data.

- Building on the insights from the semi-supervised task, and the failure of the DCEC algorithm, we successfully clustered AT-TPC data with the MIXAE algorithm. We demonstrate that this approach can increase performance on the clustering task compared to the VGG16+K-means algorithm. However, the MIXAE performance is dependent on the loss-weights which we selected based on the performance on the clustering task. We also found challenges with the MIXAE model as it has significant stability problems.

The contribution of the present work is then to both demonstrate the applicability of pre-trained models in the unsupervised clustering of AT-TPC data. Moreover, we have shown that MIXAE model can improve upon this performance. Further research is needed to understand the variability in autoencoder based clustering performance. Additionally, deep clustering is an active field of research and novel methods might provide additional insight.

In summary, we have found promising avenues for research applying both supervised and unsupervised techniques applied to AT-TPC data. With the latter having major implications for experiments in which researchers are unable to separate event types. However, this research is still at an early stage and

for current experiments we recommend the application of pre-trained models for both supervised and unsupervised tasks.

Part V

Appendices

Appendix A

Kullback-Leibler divergence of of Gaussian distributions

A multivariate Gaussian distribution in $\mathbb{R}^n$ is defined in terms of its probability density, which is a complete analogue to its univariate formulation,

$$p(x) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)). \tag{A.1}$$

And described in full by the mean vector μ and covariance matrix Σ. The Kullback-Leibler divergence between two multivariate Gaussians is then given as

$$\begin{aligned} D_{KL}(p_1||p_2) &= \langle \log p_1 - \log p_2 \rangle_{p_1} \\ &= \langle \frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} + \frac{1}{2}(-(x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1) + (x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2)) \rangle. \end{aligned}$$

We abuse the fact that the exponential factors represent an inner-product to apply a trace operator to manipulate the sequence of operations given the trace operators invariance under cyclical permutations i.e. $tr(X^TBX) = tr(BX^TX)$. Furthermore we use the fact that the trace is a linear operator and so commutes with the expectation i.e. $E(tr(BX^TX)) = tr(B\,E(X^TX))$. We also move the logarithm of the covariance determinants outside of the expectations,

$$\begin{aligned} D_{KL}(p_1||p_2) &= \frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} + \frac{1}{2}\langle -tr(\Sigma_1^{-1}(x-\mu_1)^T(x-\mu_1)) + tr(\Sigma_2^{-1}(x-\mu_2)^T(x-\mu_2)) \rangle \\ &= \frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} + \frac{1}{2}(-tr(\Sigma_1^{-1}\langle(x-\mu_1)^T(x-\mu_1)\rangle) + tr(\Sigma_2^{-1}\langle(x-\mu_2)^T(x-\mu_2)\rangle)). \end{aligned}$$

Conveniently the covariance matrix is defined by the expectation

$$\Sigma := \langle (x-\mu)^T (x-\mu) \rangle, \tag{A.2}$$

giving an evident simplification. For the terms originating from p_2 we will use the definitions of the covariance matrix and the mean vector, i.e. $\mu = \langle x \rangle$ and

$$\begin{aligned} \Sigma &= \langle x^T x - 2x\mu^T + \mu\mu^T \rangle \\ \Sigma &= \langle x^T x \rangle - \mu\mu^T. \end{aligned}$$

Returning to the Kullback-Leibler divergence we then have

$$\begin{aligned} D_{KL}(p_1||p_2) &= \frac{1}{2} \log \frac{|\Sigma_2|}{|\Sigma_1|} + \frac{1}{2}(-tr(\Sigma_1^{-1}\Sigma_1) + tr(\Sigma_2^{-1}\langle x^T x - 2x\mu_2^T + \mu_2\mu_2^T \rangle)) \\ &= \frac{1}{2}\left(\log \frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}(\Sigma_1 + \mu_1\mu_1^T - 2\mu_1\mu_2^T + \mu_2\mu_2^T)) \right). \end{aligned}$$

Grouping terms then gives us the final expression for the Kullback-Leibler divergence of two multivariate Gaussians

$$D_{KL}(p_1||p_2) = \frac{1}{2}\left(\log \frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2 - \mu_1)^T \Sigma_2^{-1}(\mu_2 - \mu_1) \right). \tag{A.3}$$

Appendix B

The bias-variance decomposition

In approximating functions we observe a relationship between the complexity of our model and how much data we have available to fit on. Quantizing this relationship helps understand what challenges we face when fitting models to data. We begin by considering the true process which we want to model, decomposed in contributions from the true function we wish to approximate, $\hat{f}$, and a noise term ϵ as

$$\hat{y}_i = \hat{f}(\mathbf{x}_i) + \epsilon, \tag{B.1}$$

where the recorded data are the tuples $s_i = (\hat{y}_i, \mathbf{x}_i)$ and the set of recorded data are denoted as $S = \{s_i\}$. We here assume that the noise is uncorrelated and distributed as $\epsilon \sim \mathcal{N}(0, \sigma^2)$.

Furthermore, assume that we have a procedure to fit a model, $g(\mathbf{x}_i; \theta)$, with parameters θ to a dataset S_k, giving an estimator for unseen data $g(\mathbf{x}_i; \theta_{S_k})$. The quality of this estimator we measure by the squared error cost function, which has the form

$$\mathcal{C}(S, g(\mathbf{x}_i)) = \sum_i (\hat{y}_i - g(\mathbf{x}_i; \theta_{S_k}))^2. \tag{B.2}$$

The relationship we wish to describe is known as the bias-variance decomposition. This decomposition decomposes the expected error on unseen data of our modelling procedure to three discrete contributions, a bias term, a variance term and a noise term. Mathematically it has the form

$$\begin{aligned} \langle \mathcal{C}(S, g(\mathbf{x}_i)) \rangle_{S,\epsilon} = \sum_i & (\hat{f}(\mathbf{x}_i) - \langle g(\mathbf{x}_i; \theta_{S_k}) \rangle_S)^2 \\ & + \langle g(\mathbf{x}_i; \theta_{S_k}) - \langle g(\mathbf{x}_i; \theta_{S_k}) \rangle_S \rangle_S \\ & + \sigma^2. \end{aligned} \tag{B.3}$$

Before we derive this expression we note that the expectation $\langle g_S(\mathbf{x}_i;\theta_{S_k})\rangle_S$ is the expected value of our model, g, on an unseen datum, $\mathbf{x}$, when trained on differing datasets, S_k.

The derivation of the relationship starts with the expectation of the cost with respect to the data selection- and noise-effects. It has the form

$$\langle \mathcal{C}(S, g(\mathbf{x}_i))\rangle_{S,\epsilon} = \sum_i \langle (\hat{y}_i - g(\mathbf{x}_i;\theta_{S_k}))^2\rangle_{S,\epsilon}. \tag{B.4}$$

We introduce a notational shorthand, $g(\mathbf{x}_i;\theta_{S_k}) := g_{S_k}$, for the estimator to maintain clarity in the derivation. The derivation begins by adding and subtracting the expected value of our estimator on the unseen data, and we then have that

$$\begin{aligned}
\langle \mathcal{C}(S, g(\mathbf{x}_i))\rangle_{S,\epsilon} &= \sum_i \langle (\hat{y}_i - g_{S_k})^2\rangle_{S,\epsilon}, && \text{(B.5)}\\
&= \sum_i \langle (\hat{y}_i + \langle g_{S_k}\rangle_S - \langle g_{S_k}\rangle_S - g_{S_k})^2\rangle_{S,\epsilon}, && \text{(B.6)}\\
&= \sum_i [\langle (\hat{y}_i - \langle g_{S_k}\rangle_S)^2\rangle_{S,\epsilon} \\
&\quad + \langle (g_{S_k} - \langle g_{S_k}\rangle_S)^2\rangle_S \\
&\quad + \langle \hat{y}_i - \langle g_{S_k}\rangle_S\rangle_{S,\epsilon} \cdot \langle g_{S_k} - \langle g_{S_k}\rangle_S\rangle_S], && \text{(B.7)}
\end{aligned}$$

where we observe that the cross-term is zero. Further decomposing the $\hat{y}_i$ we can write this as

$$\begin{aligned}
\langle \mathcal{C}(S, g(\mathbf{x}_i))\rangle_{S,\epsilon} &= \sum_i [\langle (\hat{f}(\mathbf{x}_i) + \epsilon - \langle g_{S_k}\rangle_S)^2\rangle_{S,\epsilon} \\
&\quad + \langle (g_{S_k} - \langle g_{S_k}\rangle_S)^2\rangle_S], && \text{(B.8)}\\
&= \sum_i [(\hat{f}(\mathbf{x}_i) - \langle g_{S_k}\rangle_S)^2 \\
&\quad + \langle (g_{S_k} - \langle g_{S_k}\rangle_S)^2\rangle_S \\
&\quad + \sigma^2], && \text{(B.9)}
\end{aligned}$$

where the cross-term from the last transition is zero as the error has zero mean, by assumption.

Appendix C

Neural network architectures

Table C.1: Showing the details of the VGG network architectures. Network D trained on the ImageNet [49] dataset the network known as VGG16 and is what we use in this thesis.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224×224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Appendix D

Model hyperparameters

The convolutional autoencoder and the deep recurrent attentive writer each have many hyperparameters that need to be specified. We provide a complete listing with descriptions in tables D.1 and D.2.

Table D.1: Detailing the hyperparameters that need to be determined for the convolutional autoencoder. The depth and number of filters strongly influence the number of parameters in the network. For all the search-types we follow heuristics common in the field, the network starts with larger kernels and smaller numbers of filters etc.

Hyperparameter	Scale	Description
Convolutional parameters:		
Number of layers	Linear integer	A number describing how many convolutional layers to use
Kernels	Set of linear integers	An array describing the kernel size for each layer
Strides	Set of linear integers	An array describing the stride for each layer
Filters	Set of logarithmic integers	An array describing the number of filters for each layer
Network parameters:		
Activation	Multinomial	An activation function as detailed in section 3.1.3
Latent type	Multinomial	One of the latent space regularization techniques (KLD, MMD, clustering loss)
Latent dimension	Integer	The dimensionality of the latent space
β	Logarithmic int	Weighting parameter for the latent term
Batchnorm	Binary	Whether to use batch-normalization in each layer
Optimizer parameters:		
η	Logarithmic float	Learning rate, described in 2.10
β_1	Linear float	Momentum parameter, described in 2.10.1
β_2	Linear float	Second moment momentum parameter. Described in 2.10.3

Table D.2: Hyperparameters for the draw algorithm as outlined in section 4.5. The implementation of the convolutional read and write functions is a novel contribution to the DRAW algorithm. We investigate which read/write paradigm is most useful for classification and clustering. Additionally as a measure ensuring the comparability of latent sample we fix the δ parameter determining the glimpse size. The effect of δ is explored in detail in the paper by Gregor et al. [24] and in the earlier section 4.5.

Hyperparameter	Scale	Description
Recurrent parameters:		
Readwrite functions	Binary	One of attention or convolutional describing the way draw looks and adds to the canvas.
Nodes in recurrent layer	Integer	Describing the number of cells in the LSTM cells
Network parameters:		
Latent type	Multinomial	One of the latent space regularization techniques (KLD, MMD, clustering loss)
Latent dimension	Integer	The dimensionality of the latent space
β	Logarithmic int	Weighting parameter for the latent term
Optimizer parameters:		
η	Logarithmic float	Learning rate, described in 2.10
β_1	Linear float	Momentum parameter, described in 2.10.1
β_2	Linear float	Second moment momentum parameter. Described in 2.10.3

Bibliography

[1] D. Silver, T. Hubert, J. Schrittwieser, et al. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science*, 362 (6419):1140–1144, dec 2018. ISSN 10959203. doi: 10.1126/science.aar6404.

[2] H. W. Lin, M. Tegmark, and D. Rolnick. Why Does Deep and Cheap Learning Work So Well? *Journal of Statistical Physics*, 168(6):1223–1247, 2017. ISSN 00224715. doi: 10.1007/s10955-017-1836-5.

[3] Y. Wang and M. Kosinski. Deep neural networks are more accurate than humans at detecting sexual orientation from facial images. *Journal of Personality and Social Psychology*, 114(2):246–257, feb 2018. ISSN 00223514. doi: 10.1037/pspa0000098.

[4] J. Frankle, K. Dziugaite, D. M. Roy, and M. Carbin. Stabilizing the Lottery Ticket Hypothesis. Technical report, 2019. URL https://arxiv.org/pdf/1903.01611.pdf.

[5] J. Frankle and M. Carbin. The lottery ticket hypothesis: finding sparse, trainable neural networks. In *International Conference on Learning Representations*, 2019. URL https://arxiv.org/pdf/1803.03635.pdfhttps://openreview.net/forum?id=rJl-b3RcF7.

[6] P. Mehta, M. Bukov, C. H. Wang, et al. A high-bias, low-variance introduction to Machine Learning for physicists. *Physics Reports*, 810:1–124, mar 2019. ISSN 03701573. doi: 10.1016/j.physrep.2019.03.001. URL https://linkinghub.elsevier.com/retrieve/pii/S0370157319300766.

[7] S. van der Walt, S. C. Colbert, and G. Varoquaux. The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30, mar 2011. ISSN 1521-9615. doi: 10.1109/MCSE.2011.37. URL http://ieeexplore.ieee.org/document/5725236/.

[8] A. E. Hoerl and R. W. Kennard. Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12(1):55–67, 1970. URL https://www.math.arizona.edu/{~}hzhang/math574m/Read/RidgeRegressionBiasedEstimationForNonorthogonalProblems.pdf.

[9] R. Tibshirani. Regression Shrinkage and Selection via the Lasso. Technical Report 1, 1996. URL https://www.jstor.org/stable/pdf/2346178.pdf?refreqid=excelsior{%}3A0665690fe41c338bbaa8d3f1883ccb60.

[10] J. Bergstra and Y. Bengio. Random Search for Hyper-Parameter Optimization Yoshua Bengio. Technical report, 2012. URL http://scikit-learn.sourceforge.net.

[11] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological Review*, 65(6):19–27, 1958. URL https://www.ling.upenn.edu/courses/cogs501/Rosenblatt1958.pdf.

[12] A. Karpathy. CS231n Convolutional Neural Networks for Visual Recognition, 2019. URL https://cs231n.github.io/neural-networks-3/.

[13] I. Sutskever, J. Martens, G. Dahl, and G. Hinton. On the importance of initialization and momentum in deep learning. Technical report, 2013. URL http://proceedings.mlr.press/v28/sutskever13.pdf.

[14] S. Ruder. An overview of gradient descent optimization algorithms. Technical report, Insight Centre for Data Analytics, 2016. URL http://caffe.berkeleyvision.org/tutorial/solver.html.

[15] N. S. Keskar, D. Mudigere, J. Nocedal, M. Smelyanskiy, and P. T. P. Tang. On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima. In *International Conference on Learning Representations*, sep 2017. URL http://arxiv.org/abs/1609.04836https://openreview.net/forum?id=H1oyRlYgg.

[16] D. P. Kingma and J. Lei Ba. ADAM: a method for stochastic optimization. In *International Conference on Learning Representations*, 2014. URL https://arxiv.org/pdf/1412.6980.pdfhttps://openreview.net/forum?id=33X9fd2-9FyZd{&}noteId=33X9fd2-9FyZd.

[17] M. P. Kuchera, R. Ramanujan, J. Z. Taylor, et al. Machine learning methods for track classification in the AT-TPC. *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 940:156–167, oct 2019. ISSN 01689002. doi: 10.1016/j.nima.2019.05.097. URL https://linkinghub.elsevier.com/retrieve/pii/S0168900219308046.

[18] J. Wishart and J. Neyman. *Proceedings of the Berkeley Symposium on Mathematical Statistics and Probability*, volume 34. University of California Press, 1950. doi: 10.2307/3610901. URL https://www.jstor.org/stable/3610901?origin=crossref.

[19] S. Marsland. Machine Learning: An Algorithmic Perspective, 2009. ISSN 00368075.

[20] B. Schölkopf, A. Smola, and K.-R. Müller. Component Analysis as a Kernel Eigenvalue Problem. 5(44):1299–1319, 1996. doi: 10.1.1.100.3636. URL https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.100.3636http://www.face-rec.org/algorithms/Kernel/kernelPCA{_}scholkopf.pdf.

[21] E. Fertig, A. Arbabi, and A. A. Alemi. beta-VAEs can retain label information even at high compression. Technical report, 2018. URL http://arxiv.org/abs/1812.02682.

[22] L. Hubert and P. Arabic. Comparing Partitions. *Journal of Classification*, 2:193–218, 1985. URL https://link.springer.com/content/pdf/10.1007{%}2FBF01908075.pdf.

[23] F. Pedregosa, G. Varoquaux, A. Gramfort, et al. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830, 2011. URL http://www.jmlr.org/papers/v12/pedregosa11a.html.

[24] K. Gregor, D. J. Rezende, and D. Wierstra. DRAW: A Recurrent Neural Network For Image Generation. In *International conference on machine learning*, volume 37, pages 1462–1471, 2015.

[25] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, dec 1943. ISSN 0007-4985. doi: 10.1007/BF02478259. URL http://link.springer.com/10.1007/BF02478259.

[26] S. Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT*, 16(2):146–160, jun 1976. ISSN 0006-3835. doi: 10.1007/BF01931367. URL http://link.springer.com/10.1007/BF01931367.

[27] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. In *neural information processing systems*, pages 1097–1105, 2012. URL https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networkshttp://code.google.com/p/cuda-convnet/.

[28] N. Srivastava, G. Hinton, A. Krizhevsky, and R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Technical report, 2014. URL http://jmlr.org/papers/volume15/srivastava14a.old/srivastava14a.pdf.

[29] S. Ioffe and C. Szegedy. Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift. In *International conference*

on machine learning, page 11, 2015. URL https://pdfs.semanticscholar.org/b58f/1529c22d682dbe08ae02ec52587c9da7f270.pdf.

[30] V. Dumoulin and F. Visin. A guide to convolution arithmetic for deep learning. mar 2016. URL http://arxiv.org/abs/1603.07285.

[31] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998. ISSN 00189219. doi: 10.1109/5.726791. URL http://ieeexplore.ieee.org/document/726791/.

[32] C. Szegedy, W. Liu, P. Sermanet, et al. Going deeper with convolutions. Technical report, 2014. URL https://arxiv.org/pdf/1409.4842.pdf.

[33] B. A. Pearlmutter. Learning State Space Trajectories in Recurrent Neural Networks. *Neural Computation*, 1(2):263–269, jun 1989. ISSN 0899-7667. doi: 10.1162/neco.1989.1.2.263. URL http://www.mitpressjournals.org/doi/10.1162/neco.1989.1.2.263.

[34] A. Karpathy. The Unreasonable Effectiveness of Recurrent Neural Networks, 2015. URL https://karpathy.github.io/2015/05/21/rnn-effectiveness/.

[35] S. Hochreiter and J. Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, nov 1997. ISSN 08997667. doi: 10.1162/neco.1997.9.8.1735.

[36] D. P. Kingma and M. Welling. Auto-Encoding Variational Bayes. In *International Conference on Learning Representations*, dec 2014. URL http://arxiv.org/abs/1312.6114https://openreview.net/forum?id=33X9fd2-9FyZd.

[37] I. Higgins, L. Matthey, A. Pal, et al. β-VAE: Learning basic visual concepts with a constrained variational framework. In *International Conference on Learning Representationsr*, 2017. URL https://pdfs.semanticscholar.org/a902/26c41b79f8b06007609f39f82757073641e2.pdfhttps://openreview.net/forum?id=Sy2fzU9gl.

[38] S. Zhao, J. Song, and S. Ermon. InfoVAE: Information Maximizing Variational Autoencoders. In *International conference on machine learning*, page 24, 2018. URL http://arxiv.org/abs/1706.02262.

[39] M. Hjorth-Jensen. Computational Physics 2, 2019. URL https://compphysics.github.io/ComputationalPhysics2/doc/web/course.

[40] S. Kullback and R. A. Leibler. On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86, mar 1951. ISSN 0003-4851. doi: 10.1214/aoms/1177729694. URL http://projecteuclid.org/euclid.aoms/1177729694.

[41] K. P. Burnham, D. R. Anderson, and K. P. Burnham. *Model selection and multimodel inference : a practical information-theoretic approach*. Springer, 2002. ISBN 0387953647. URL https://books.google.no/books?id=fT1Iu-h6E-oC{&}pg=PA51{&}redir{_}esc=y{#}v=onepage{&}q{&}f=false.

[42] B. Seybold, E. Fertig, A. Alemi, and I. Fischer. Dueling Decoders: Regularizing Variational Autoencoder Latent Spaces. may 2019. URL http://arxiv.org/abs/1905.07478.

[43] E. Harris, M. Niranjan, and J. Hare. A Biologically Inspired Visual Working Memory for Deep Networks. Technical report, jan 2019. URL http://arxiv.org/abs/1901.03665https://openreview.net/forum?id=B1fbosCcYm.

[44] X. Guo, X. Liu, E. Zhu, and J. Yin. Deep Clustering with Convolutional Autoencoders. In *neural information processing systems*, pages 373–382. 2017. doi: 10.1007/978-3-319-70096-0_39. URL https://xifengguo.github.io/papers/ICONIP17-DCEC.pdfhttp://link.springer.com/10.1007/978-3-319-70096-0{_}39.

[45] D. Zhang, Y. Sunm, B. Eriksson, and L. Balzano. Deep Unsupervised Clustering Using Mixture of Autoencoders. In *International Conference on Neural Information Processing*, pages 373–382, 2017. URL https://arxiv.org/pdf/1712.07788.pdf.

[46] L. Van Der Maaten and G. Hinton. Visualizing Data using t-SNE. Technical report, 2008. URL http://www.jmlr.org/papers/volume9/vandermaaten08a/vandermaaten08a.pdf.

[47] J. Xie, R. Girshick, and A. Farhadi. Unsupervised Deep Embedding for Clustering Analysis. Technical report, 2015. URL http://arxiv.org/abs/1511.06335.

[48] K. Simonyan and A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*, sep 2015. doi: 10.1.1.740.6937. URL http://www.robots.ox.ac.uk/http://arxiv.org/abs/1409.1556.

[49] O. Russakovsky, J. Deng, H. Su, et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252, dec 2015. ISSN 0920-5691. doi: 10.1007/s11263-015-0816-y. URL http://link.springer.com/10.1007/s11263-015-0816-y.

[50] J. W. Bradt. *Measurement of isobaric analogue resonances of 47Ar with the active target time projection chamber*. PhD thesis, Michigan State University, 2017.

[51] J. Bradt, D. Bazin, F. Abu-Nimeh, et al. Commissioning of the Active-Target Time Projection Chamber. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 875:65–79, dec 2017. ISSN 0168-9002. doi: 10.1016/J.NIMA.2017.09.013. URL https://www.sciencedirect.com/science/article/pii/S0168900217309683.

[52] O. Kester, D. Bazin, C. Benatti, et al. ReA3-The rare isotope reaccelerator at MSU. In *Proceedings of Linear Accelerator Conference*, pages 26–30, Tsukuba, Japan, 2010. URL http://accelconf.web.cern.ch/AccelConf/LINAC2010/papers/mo203.pdf.

[53] Y. Giomataris, P. Rebourgeard, J. Robert, and G. Charpak. MICROMEGAS: a high-granularity position-sensitive gaseous detector for high particle-flux environments. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 376(1):29–35, jun 1996. ISSN 01689002. doi: 10.1016/0168-9002(96)00175-1. URL https://www.sciencedirect.com/science/article/pii/0168900296001751?via{%}3Dihubhttps://linkinghub.elsevier.com/retrieve/pii/0168900296001751.

[54] W. Mittig, S. Beceiro-Novo, A. Fritsch, et al. Active Target detectors for studies with exotic beams: Present and next future. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 784:494–498, jun 2015. ISSN 0168-9002. doi: 10.1016/J.NIMA.2014.10.048. URL https://www.sciencedirect.com/science/article/pii/S0168900214012054.

[55] D. Suzuki, M. Ford, D. Bazin, et al. Prototype AT-TPC: Toward a new generation active target time projection chamber for radioactive beam experiments. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 691:39–54, nov 2012. ISSN 0168-9002. doi: 10.1016/J.NIMA.2012.06.050. URL https://www.sciencedirect.com/science/article/pii/S0168900212007164.

[56] J. D. Hunter. Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95, 2007. ISSN 1521-9615. doi: 10.1109/MCSE.2007.55. URL http://ieeexplore.ieee.org/document/4160265/.

[57] M. Abadi, A. Agarwal, P. Barham, et al. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 265–283, 2016. URL www.tensorflow.org.https://ai.google/research/pubs/pub45381.

[58] C. C. Aggarwal, A. Hinneburg, and D. A. Keim. On the Surprising Behavior of Distance Metrics in High Dimensional Space. pages 420–434. 2001. doi: 10.1007/3-540-44503-x_27. URL https://bib.dbvis.de/uploadedFiles/155.pdf.

[59] J. Antorán and A. M. Vivolab. DISENTANGLING IN VARIATIONAL AUTOENCODERS WITH NATURAL CLUSTERING. 2019. URL https://arxiv.org/pdf/1901.09415.pdf.

[60] I. T. Jolliffe. A Note on the Use of Principal Components in Regression. *Applied Statistics*, 31(3):300–303, 1982. ISSN 00359254. doi: 10.2307/2348005. URL http://automatica.dei.unipd.it/public/Schenato/PSC/2010{_}2011/gruppo4-Building{_}termo{_}identification/IdentificazioneTermodinamica20072008/Biblio/Articoli/PCRvecchio82.pdf.