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

August 30, 2019

Contents

List of Figures

Todo list

Abstract

In this thesis a novel filtering thechinque of AT-TPC noise events is presented using clustering techniques on the latent space produced by a Variational Autoencoder(VAE)

Chapter 1

Introduction

1.1   Research question and hypotheses

In this thesis we explore the following research question and hypotheses

(**R0**): To what degree are we able to separate event classes from an AT-TPC experiment.

with the related hypotheses

There exists a mapping from raw AT-TPC data to some lower dimensional vector space $\mathbf{z}$ such that:

(**H0**): a set of hyperplanes in this space separates the event types present

(**H1**): the mapping clusters the event types present in disjoint sets under some distance metric

1.2   Why machine learning?

- Advent of large amounts of data

- Precedence from High energy

- Promise of discovery from unsupervised methods

- Need for exploration

Need to add abbreviations to list

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'll introduce them in turn using a couple of simple models; linear and logistic regression. Before those discussions we briefly outline the process of model fitting and briefly introduce the difference in models with a known or unknown outcome.

Fitting models to data is the formal framework by which much of modern science is underpinned. In most sciences the researcher has a need 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 the estimating the Hamiltonian of a simple binary spin system like the Ising model, to more complex methods like variational Markov chain Monte Carlo which is used for 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 $\hat{\mathbf{y}}$,

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

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 parameters θ take different forms for different models but in general it can be thought of as a set of matrices that is used to transform the input to the output dimension. Additionally the output of the function can be multivariate, discrete or continuous which informs the choice of model for a particular problem. In this first part of the 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 $y = f(x; \theta)$. In the event that f is a probability density the aforementioned 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.

There is also a good amount of manipulation of expectation values in both this chapter and the following, and so we introduce the notation used throughout the thesis here. Let $p(x)$ be a normalized probability density function, i.e.

$$1 = \int_{-\infty}^{\infty} p(x)dx. \tag{2.1}$$

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

$$\langle f(x) \rangle_p := \int_{-\infty}^{\infty} f(x)p(x)dx. \tag{2.2}$$

Some special expectations are notable because of their wide applicability, we concern ourselves primarily with the mean and variance of a distribution. Those expectations are also known as the first and second moment of a distribution are defined as

$$\mu := \langle x \rangle_p, \tag{2.3}$$
$$\sigma^2 := \langle x^2 \rangle_p - \langle x \rangle_p^2. \tag{2.4}$$

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 system states, $\mathbf{X}$. However this thesis deals largely with the problem of modeling when one only has access to the system states. However 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 are able to estimate the performance of our model in the real world. To elaborate the need for this separation we explore the concepts of over-fitting and under-fitting to the data later in this chapter.

2.2   Linear Regression

Modern machine learning has its foundations in the familiar framework of linear regression. Many fairly interesting problems can be cast as systems of linear problems, and as such there are multitudes of ways to solve the problem. The most straight forward of which is with just a little bit of linear algebra.

   We begin by defining the constituents of our model: let our data be denoted as a matrix $\mathbf{X} \in \mathcal{R}^{m \times n+1}$ where m is the number of data-points and n the number of features. The $+1$ factor in the dimension is to accommodate the addition of a column of ones to our data as a convenient placeholder for the model intercept. Note also that the term features is used broadly in machine learning literature and denotes 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 $\mathbf{w} \in \mathcal{R}^{n+1 \times k}$. Generally the outcome-dimension k can be greater than one when estimating multi-variate outcomes. For illustrative purposes we limit this section and the following to the case where $k = 1$. The linear regression model is then the transformation of the input using the weight matrix, i.e.

$$\mathbf{y} = \mathbf{X}\mathbf{w}. \tag{2.5}$$

Finally let the outcome we wish to approximate be given as a vector $\hat{\mathbf{y}} \in \mathcal{R}^m$. We'll 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 coming 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 the first derivative in terms of the model parameters. The squared error is defined in terms of the Euclidean vector norm

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

which we apply to the vector difference between model output and true values and take the square to give us a nice differentiable form. In mathematical terms we define the squared error objective as

$$\mathcal{O} = ||\hat{\mathbf{y}} - \mathbf{y}||_2^2. \tag{2.6}$$

An objective function $\mathcal{O}$ defines some optimization problem to minimize or maximize. In machine learning these are mostly commonly cast as minimization problems. Objective functions for finding minima are termed cost functions in machine learning literature and we'll adopt that name 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.

$$\theta^* = \arg\min_{\theta} \mathcal{C}(\hat{\mathbf{y}}, f(\mathbf{X}; \theta)). \tag{2.7}$$

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

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

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

$$\nabla_{\mathbf{w}}\mathcal{C} = \nabla_{\mathbf{w}}(\hat{\mathbf{y}} - \mathbf{X}\mathbf{w})^T(\hat{\mathbf{y}} - \mathbf{X}\mathbf{w}), \tag{2.8}$$
$$= -2\mathbf{X}^T\hat{\mathbf{y}} + 2\mathbf{X}^T\mathbf{X}\mathbf{w}, \tag{2.9}$$
$$\mathbf{0} = -2\mathbf{X}^T\hat{\mathbf{y}} + 2\mathbf{X}^T\mathbf{X}\mathbf{w}, \tag{2.10}$$
$$\mathbf{X}^T\hat{\mathbf{y}} = \mathbf{X}^T\mathbf{X}\mathbf{w}, \tag{2.11}$$
$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\hat{\mathbf{y}}. \tag{2.12}$$

This problem is of course solvable with a plethora of other tools, most notably we have the ones that don't 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 fairly simple and is rarely applicable to complex systems. Additionally we've not mentioned the assumptions made to ensure the validity of the model, the most important of which concerns the measurement errors which should be independent identical normally distributed. But given it's relative simplicity and the fact that an analytical solution can be found we'll use it for reference in understanding the following sections.

2.3   Over and under-fitting

When estimating an unknown function from 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're 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 obviously 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 to 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,

$$\hat{y}_i = P(\mathbf{x}_i) + \epsilon_i. \tag{2.13}$$

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. In machine learning, with modern computing resources, it turns out to be much easier to make a model too complex than having it be not complex enough. Frankle et al. (2019) and Frankle and Carbin (2018) 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 simple and robust method of doing this is creating a disjoint hold out set of the data which is not used during the estimating 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 we'll consider a one-dimensional polynomial regression problem[1]. Two sets of outcomes were generated from a process P as in equation 2.13 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 ɴᴜᴍᴘʏ (van der Walt et al. (2011)). Additionally we split the sets in disjoint subsets for training and testing. The data and the models are shown in figure 2.1. In the figure we observe 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

---

[1]We note that this section follows closely that of section 2 in Mehta et al. (2019), we also refer to this paper for a more in-depth introduction to machine learning for physicists.

that increases performance in the training domain, but 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 drawn from P^3 the linear model looses 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" (Mehta et al. (2019)). 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.
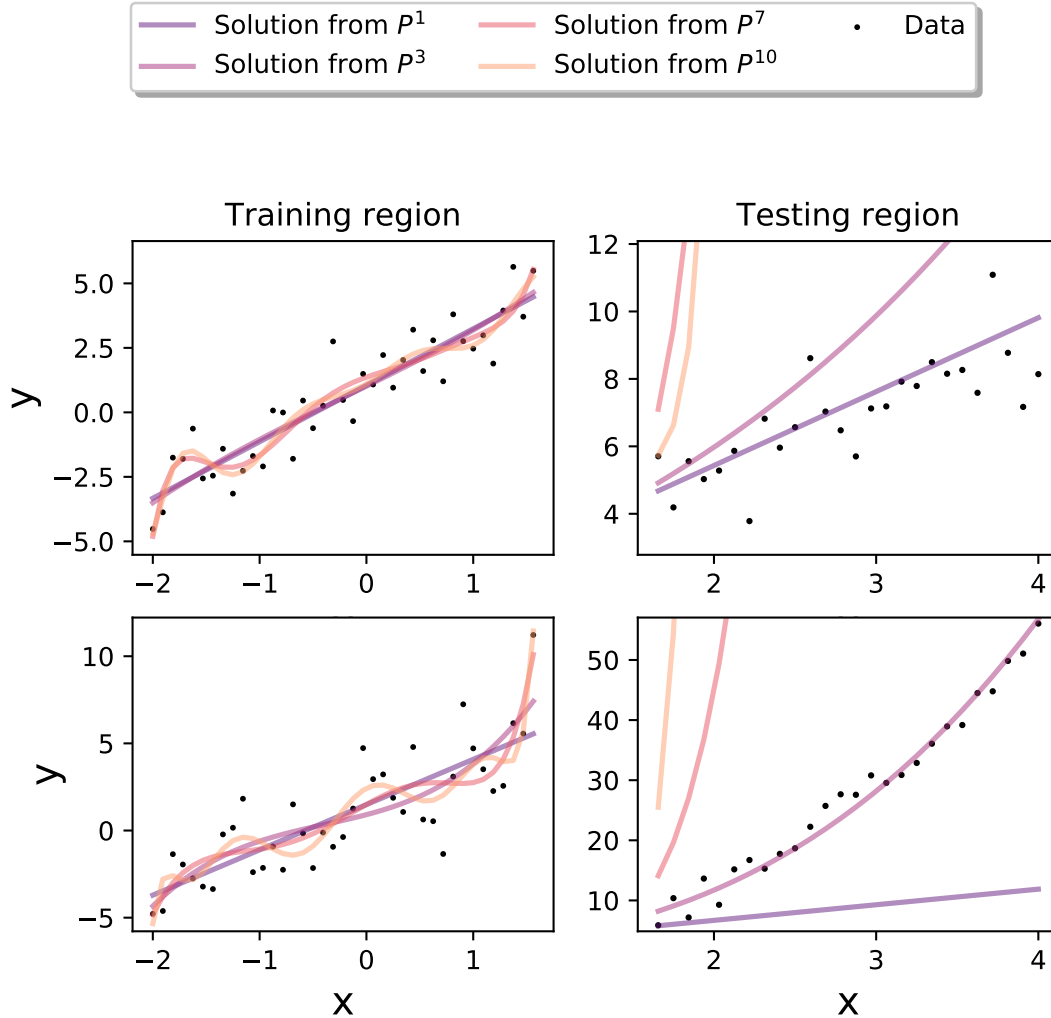
Figure 2.1: Polynomial regression of varying degrees on data drawn from a linear distribution on above and a cubic distribution on the bottom. Models of varying complexity indicated by their basis P^n are fit to the train data and evaluated on the test region, shown in the left and right columns. We 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 column 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 does not have the capability to 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.

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 least squares cost function can be decomposed in three contributions[2], namely

$$E_{out} = \langle C(\hat{\mathbf{y}}^t, f(\mathbf{x}^t; \theta_S^*)) \rangle_{S, \epsilon} = \text{Bias}^2 + \text{Variance} + \text{Noise}. \qquad (2.14)$$

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 2.13 that we decompose the target values as a contribution from a true process and an error term ϵ. The expectation over the cost then has a contribution from the noise, represented by the last term of the decomposition. Furthermore the expectation in equation 2.14 is taken over a model with optimized parameters θ^*, but that optimization can be thought to be a function of the selected data used in the optimization. We can then view $f(\mathbf{x}; \theta_S^*)$ as a stochastic functional which varies over the selected data $s = \{\hat{\mathbf{y}}, \mathbf{x}\}$ used for training (Mehta et al. (2019)). The expectation over S is then an expectation over the set of different choices of training data, i.e. $S = \{s_i, s_{i+1}, \dots\}$. We also use the superscript t on the outcome and state variables to indicate that they are from the testing set, variables without that subscript are implied to be from the training set. Using the derived quantities from Mehta et al. (2019), and the notation described in the previous section and equation 2.13, we can then define the bias as

> add some details on derivation?

$$\text{Bias}^2 := \sum_i (\hat{\mathbf{y}}_i^t - \langle f(\mathbf{x}_i^t; \theta_S^*) \rangle_S)^2. \qquad (2.15)$$

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, again using notation from this section,

$$\text{Variance} := \sum_i \langle (f(\mathbf{x}_i; \theta_S^*) - \langle f(\mathbf{x}_i; \theta_S^*) \rangle_S) \rangle_S. \qquad (2.16)$$

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. Over-fitting is then a behavior intrinsically tied to the amount of data we have to train on (Mehta et al. (2019)).

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

---

[2]This is shown by Mehta et al. (2019) and we refer to that work for details on the derivation
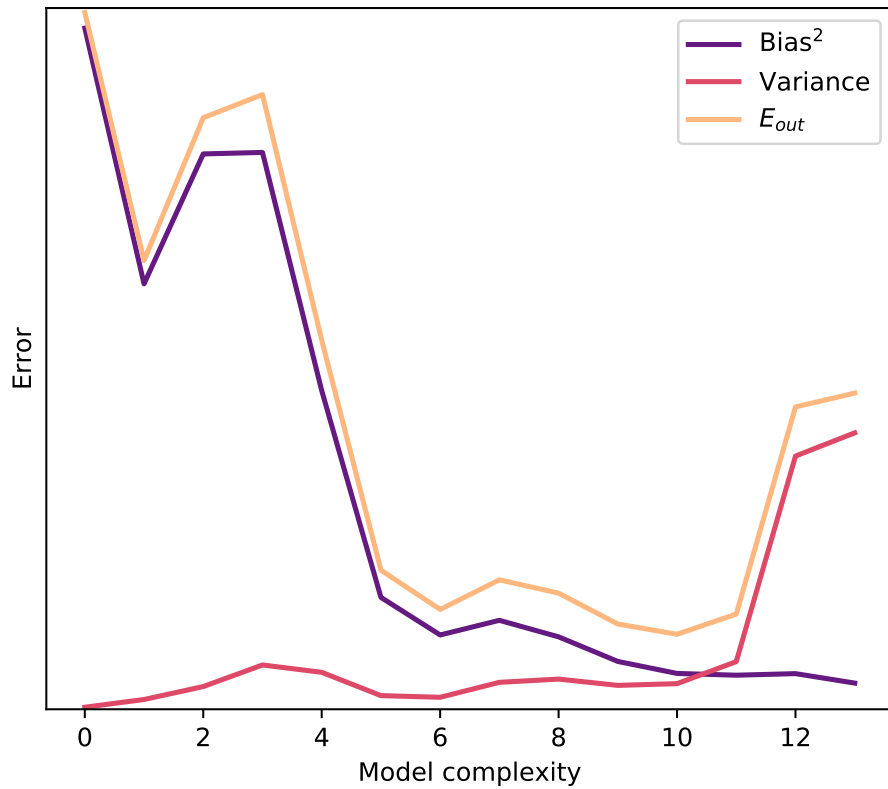
Figure 2.2: Decomposition of the bias and variance from the overall test-set error E_{out}. A set of polynomials of varying degrees are fit to a known function using randomly selected training samples. The polynomial degree is denoted by the x-axis label. With the set of polynomials we evaluate the bias and variance terms shown in equation 2.15 and 2.16. 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.

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 giving 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 greatly beneficial if the recognition of reactions translate 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 models ability to express all of its complexity. Introduced in 1970 by Hoerl and Kennard (1970) 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 Hoerl and Kennard (1970). Perhaps most influential of them is the use of the L_1-norm, first successfully implemented by Tibshirani (1996). 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, copied from Mehta et al. (2019), 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 where as 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
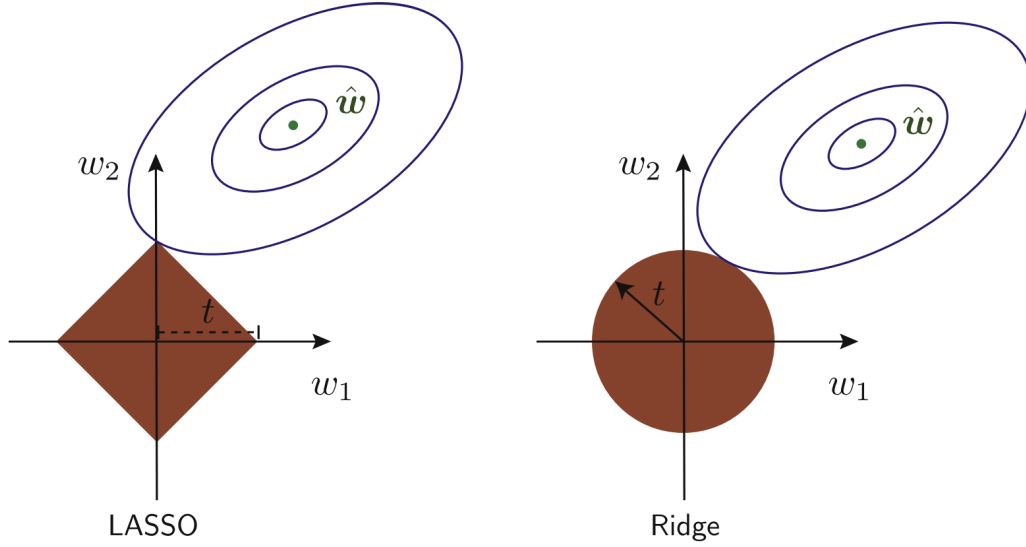
Figure 2.3: Demonstrating the effect of a regularization constraint on a 2-variable optimization. The blue ovals represent the squared error, as it is quadratic in the parameters w_i. And the shaded brown region represents the restriction on the values of w_i, s.t. the only eligible values for the parameters 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. (2019), which in turn adapted a figure from Friedman et al. (2001)

optimization problem, for example in the case of least squares regression

$$\theta^* = \underset{||\theta||_2^2 < t}{\arg\min} ||y_i - f(\mathbf{x}_i; \theta)||_2^2. \tag{2.17}$$

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 $\mathbf{x} \in \mathcal{R}^n$ as

$$L_p(\mathbf{x}) = \left(\sum |x_i|^p \right)^{\frac{1}{p}}. \tag{2.18}$$

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. Wile the L_1 term is commonly called the Manhattan or taxicab-distance, aptly named as one can think of it as the length of the city blocks a cab-driver drives 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

$$C(\hat{y}_i, f(\mathbf{x}_i; \theta)) = (\hat{y}_i - f(\mathbf{x}_i; \theta))^2 + \lambda \sum ||\theta_i||_2^2. \qquad (2.19)$$

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

$$C(\hat{\mathbf{y}}, \mathbf{Xw}) = (\hat{\mathbf{y}} - \mathbf{Xw})^T(\hat{\mathbf{y}} - \mathbf{X}^T\mathbf{w}) - \lambda\mathbf{w}^T\mathbf{w}, \qquad (2.20)$$

$$\nabla_{\mathbf{w}}C = -2\mathbf{X}^T\hat{\mathbf{y}} + 2\mathbf{X}^T\mathbf{Xw} - 2\lambda\mathbf{w}. \qquad (2.21)$$

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

$$\mathbf{0} = -\mathbf{X}^T\hat{\mathbf{y}} + (\mathbf{X}^T\mathbf{X} - \lambda\mathbf{I})\mathbf{w}, \qquad (2.22)$$

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X} - \lambda\mathbf{I})^{-1}\mathbf{X}^T\hat{\mathbf{y}}. \qquad (2.23)$$

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 2.7 by adding a penalty to parameters having high values. This is especially useful in cases where features are co-variate 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're left with other ways of estimating its value. As models grow more complex several of these parameters which influence optimization but are not model parameters need to be added; they're collectively known as hyperparameters.

Hyperparameters are parameters in the model that have to be heuristically determined that may impact performance without having a closed form derivative with respect to 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 are expensive because each configuration of parameters is accompanied by a full training of the model. In this section we'll discuss the general process of tuning hyperparameters in general, and then we'll introduce specific parameters that need tuning in subsequent sections pertaining to particular architectures or modeling choices. 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 do the 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 they are discussed for each model type or neural network cell type a suggested scale will also be discussed.

2.6.1   Hand holding

The most naive way of doing hyperparameter optimization is to manually tune the values by observing changes in performance metrics. Being naive and unfounded

it is rarely used in modern modeling pipelines, outside the prototyping phase. For this thesis we use a hand held approach to get a rough understanding of the ranges of values over which to apply a more well reasoned approach.

2.6.2   Grid Search

Second on the ladder of naiveté is the exhaustive search of hyperparameter configurations. This in this paradigm one defines a multi dimensional grid of parameters that is evaluated. 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 in a linear regression we would want to find the optimal value for 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 however 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 in the event that the hyperparameters are disconnected i.e. 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 $\leftarrow$ [ ];
for $\lambda_k \in \lambda$ **do**
    for $\eta_k \in \eta$ **do**
        opt $\leftarrow$ optimizer(η_k);
        model_instance $\leftarrow$ model(λ_k);
        metrics $\leftarrow$ model_instance.fit($\mathbf{X}, \hat{\mathbf{y}}$, opt) ;
        log.append((metrics, (λ_k, η_k)))
    end
end

2.6.3   Random Search

Surprisingly the hyperparameter search method that has proven to be among the most effective is a simple random search. Bergstra et al. (2012) showed empirically the inefficiency of doing grid search and proposed the simple remedy of doing randomized configurations of hyperparameters. The central argument of the paper is elegantly presented in figure 2.4. Observing that grid search is both more computationally expensive and has significant shortcomings for complex modalities in the loss functions we approach the majority of hyperparameter

searches in this thesis by way of random searches. The algorithm for the random search is very simple in that it just 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.
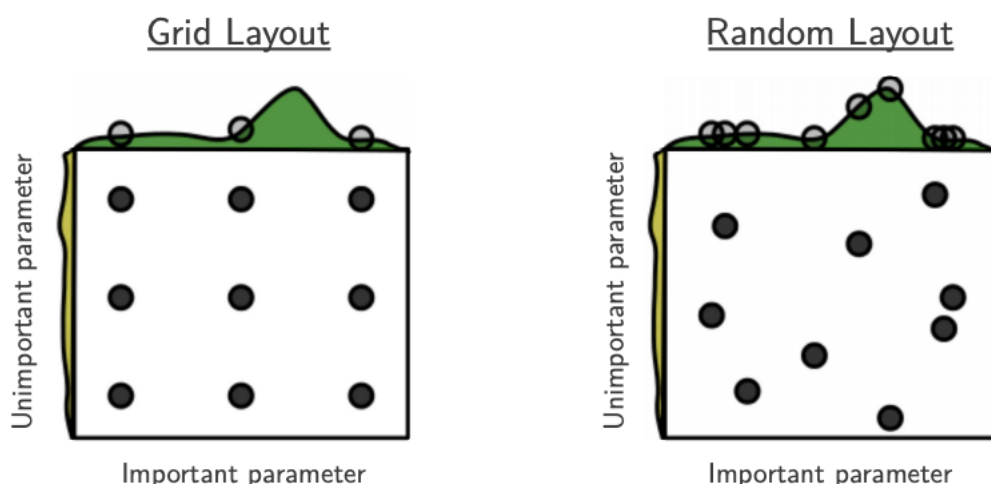
Figure 2.4: Figure showing the inefficiency of grid search. The shaded areas on the axis represent the unknown variation of the cost as a function of that axis-parameter. Since the optimum of the loss with respect to a hyperparameter might be very narrowly peaked a grid search might miss the optimum in it's entirety. A random search is less likely to make the same error as shown by Bergstra et al. (2012). Figure copied from Bergstra et al. (2012)

2.7   On information

Up until now we have been dealing with predicting a real valued outcome, but this is not always the goal. A very common 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 define tasks where the goal is to predict a discrete outcome like the species of an animal or thermodynamic state of a system. Transitioning the type of goal our model has then also necessitates some new terminology. In this section we'll briefly touch on some fundamental concepts in information theory needed to construct models that perform a classification task.

In information theory one considers the amount of chaos in in a process and how much one needs to know to characterize such a process. As we'll see this

ties into concepts well known to physicists from statistical and thermal physics. As a quick refresher we mention that processes that are more random 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 normal way as

$$I := -\log(p(\mathbf{x})), \tag{2.24}$$

where $p(\mathbf{x})$ is the probability of a given event $\mathbf{x}$ occurring. One of the quantities that turn out to have very wide 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;

$$H(p(\mathbf{x})) := -\langle I(\mathbf{x})\rangle_{p(\mathbf{x})}. \tag{2.25}$$

Depending on the choice of base of the logarithm this functional has different names, but the interpretation is largely the same as it measures the degree of randomness in the system. One of the widest used bases is log base 2 know as the Shannon entropy which describes how many bits of information 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

$$H(p,q) := -\sum_{\mathbf{x}} p(\mathbf{x})\log(q(\mathbf{x};\theta)). \tag{2.26}$$

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 MLE (Maximum Likelihood Estimate) 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

$$p(S|\theta) := \prod_i q(\mathbf{x}_i;\theta)^{\hat{y}_i} - (1 - q(\mathbf{x}_i;\theta))^{1-\hat{y}_i}. \tag{2.27}$$

We want to maximize this functional with respect to the parameters θ. The product sum is problematic in this regard as it's gradient is likely to vanish

---

[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.

as the number of terms increase, to circumvent this we take the logarithm of the likelihood defining the log-likelihood. Since the likelihood is defined 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 likelihood as the logarithmic function is monotonic [4]

$$\mathcal{C}(\mathbf{x}, y, \theta) = -\log(p(S|\theta)) = -\sum_i y_i \log(q(\mathbf{x}_i; \theta)) + (1 - y_i)(q(\mathbf{x}_i; \theta)). \quad (2.28)$$

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

$$\theta^* = \arg\min_\theta \mathcal{C}(\mathbf{x}, \hat{y}, \theta). \quad (2.29)$$

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 solution is most often not analytically solvable and so in machine learning the solution of these optimization problems is often found by of gradient descent on the construct. 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 (Rosenblatt (1958)). The perceptron uses the same transformation as in equation 2.5 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

---

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

equation 2.5. 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

$$o = \frac{p}{1 - p}. \tag{2.30}$$

Since p is bounded in the unit interval unfortunately the odds are bounded in $\mathcal{R}^+$. 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 $\mathbf{w}$

$$\log\left(\frac{p_i}{1 - p_i}\right) = \mathbf{x}_i\mathbf{w}. \tag{2.31}$$

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

$$q(y = 1|\mathbf{x}_i; \mathbf{w}) = q(\mathbf{x}_i\mathbf{w}) = \frac{1}{1 + e^{-\mathbf{x}_i\mathbf{w}}}. \tag{2.32}$$

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 2.28, i.e.

$$P(S|\mathbf{w}) = -\sum_i y_i \log q(\mathbf{x}_i\mathbf{w}) + (1 - y_i) \log\left(1 - q(\mathbf{x}_i\mathbf{w})\right). \tag{2.33}$$

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

$$\nabla_{\mathbf{w}}q(\mathbf{x}_i\mathbf{w}) = q(\mathbf{x}_i\mathbf{w})(1 - q(\mathbf{x}_i\mathbf{w}))\mathbf{x}_i, \tag{2.34}$$

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

$$\nabla_{\mathbf{w}} \log q(\mathbf{x}_i\mathbf{w}) = 1 - q(\mathbf{x}_i\mathbf{w})\mathbf{x}_i. \tag{2.35}$$

We can then write out the derivative of the cost as

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

$$= -\sum_i \left(y_i - q(\mathbf{x}_i\mathbf{w})\right)\mathbf{x}_i. \tag{2.37}$$

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 2.38. 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,

$$y_i = \mathbf{x}_i \mathbf{w}. \tag{2.38}$$

In addition to equation 2.38 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

$$\mathcal{O} = \arg\min_{\mathbf{w}} ||\hat{\mathbf{y}} - \mathbf{X}\mathbf{w}||^2. \tag{2.39}$$

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

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2), \tag{2.40}$$

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

$$\hat{y}_i \sim \mathcal{N}(\mathbf{x}_i \mathbf{w}, 0). \tag{2.41}$$

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

$$p(x|\mu, \sigma) := \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{\sigma^2}}. \tag{2.42}$$

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

$$\langle \hat{y} \rangle = \langle y + \epsilon \rangle, \tag{2.43}$$

$$\langle \hat{y} \rangle = \langle y \rangle + \langle \epsilon \rangle, \tag{2.44}$$

$$\langle \hat{y} \rangle = \mathbf{x}_i^T \mathbf{w}. \tag{2.45}$$

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.

$$\langle \hat{y} \rangle^2 + \langle \hat{y}^2 \rangle = \sigma^2. \tag{2.46}$$

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 2.27

$$p(S|\theta) = \prod_i^n \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(\hat{y}_i - \mathbf{x}_i \mathbf{w})^2}{\sigma^2}}, \tag{2.47}$$

$$= \left(\frac{1}{\sqrt{2\pi\sigma^2}} \right)^n \prod_i^n e^{-\frac{(\hat{y}_i - \mathbf{x}_j \mathbf{w})^2}{\sigma^2}}, \tag{2.48}$$

$$= \left(\frac{1}{\sqrt{2\pi\sigma^2}} \right)^n e^{-\sum_i \frac{(\hat{y}_i - \mathbf{x}_j \mathbf{w})^2}{\sigma^2}}, \tag{2.49}$$

$$= \left(\frac{1}{\sqrt{2\pi\sigma^2}} \right)^n e^{-\frac{(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w})^T (\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w})}{\sigma^2}}. \tag{2.50}$$

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

$$\theta^* = \arg\max_\theta p(S|\theta). \tag{2.51}$$

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

$$\log(p(S|\theta)) = n \log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w})^T(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w})}{\sigma^2}. \qquad (2.52)$$

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

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

$$= -\frac{1}{\sigma^2}(-2\mathbf{X}^T y + 2\mathbf{X}^T\mathbf{X}\mathbf{w}),$$

$$= -\frac{1}{\sigma^2}2\mathbf{X}^T(\hat{\mathbf{y}}_i - \mathbf{X}\mathbf{w}),$$

$$\mathbf{0} = -\frac{2}{\sigma^2}(\mathbf{X}^T\hat{\mathbf{y}}_i - \mathbf{X}^T\mathbf{X}\mathbf{w}),$$

$$\mathbf{X}^T\mathbf{X}\mathbf{w} = \mathbf{X}^T\hat{\mathbf{y}}_i.$$

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

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\hat{\mathbf{y}}. \qquad (2.53)$$

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 are shown 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, and for many functions and functionals this has a closed form solution. For functionals of complicated functions, like neural networks, 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 a 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 an update weight η

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla_{\mathbf{x}} f(\mathbf{x}_n). \tag{2.54}$$

We known that the gradient vector is in the direction of the steepest ascent for the function, moving towards a minimum then simply requires going exactly the opposite way. The parameter controlling the size of this variable step is $\eta \in \mathcal{R}_+$. This parameter is dubbed the learning rate in machine learning which 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 completely with the introduction of value decay to the learning rate. While too large a 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. Additionally since the learning rate exists outside the gradient it's 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 a multivariate logistic regression. However as suggested by Karpathy (2019) we can indirectly observe the impact of the choice of learning rate 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'll use as a reference when training the models used in this thesis.

Despite its simplicity gradient descent and it's cousins have shown to solve remarkably complex problems despite its obvious flaw: convergence is only guaranteed to a local minimum. While much more computationally efficient than higher order methods using a first order method brings with them some problems of their own. In particular there are two problems that need to be solved and we'll discus some modifications to the gradient descent procedure proposed to remedy them both.

(**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

The importance of the methods we discuss in the coming sections are also covered in some detail in a paper by Sutskever et al. (2013). A longer and more in depth overview of the methods themselves can be found in Ruder (2016)

(a)   (b)
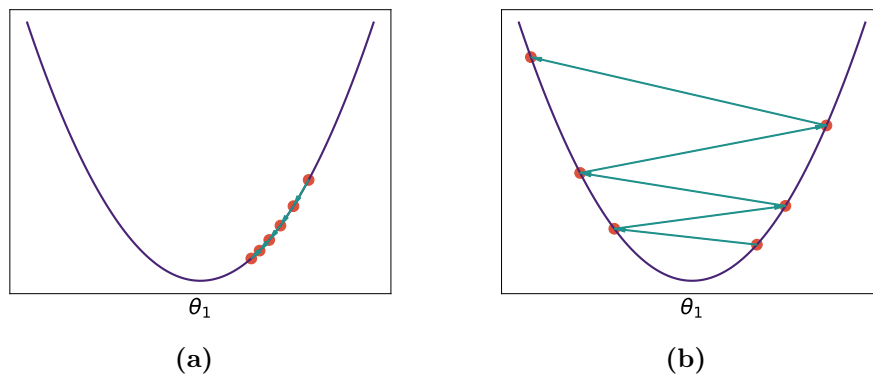
Figure 2.5: Gradient descent on a simple quadratic function showing the effect of too small, **(a)**, and too large, **(b)**, value for the learning rate η
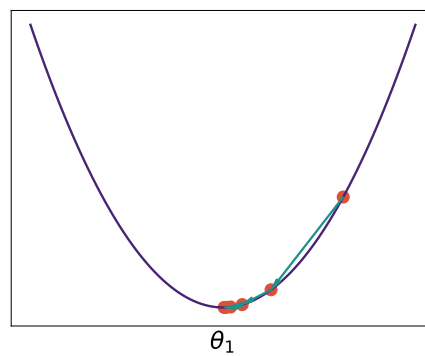
Figure 2.6: Complement to figure 2.5 where we show the effect of a good learning rate on a gradient descent procedure. The gradient descent procedure is performed on a quadratic function.
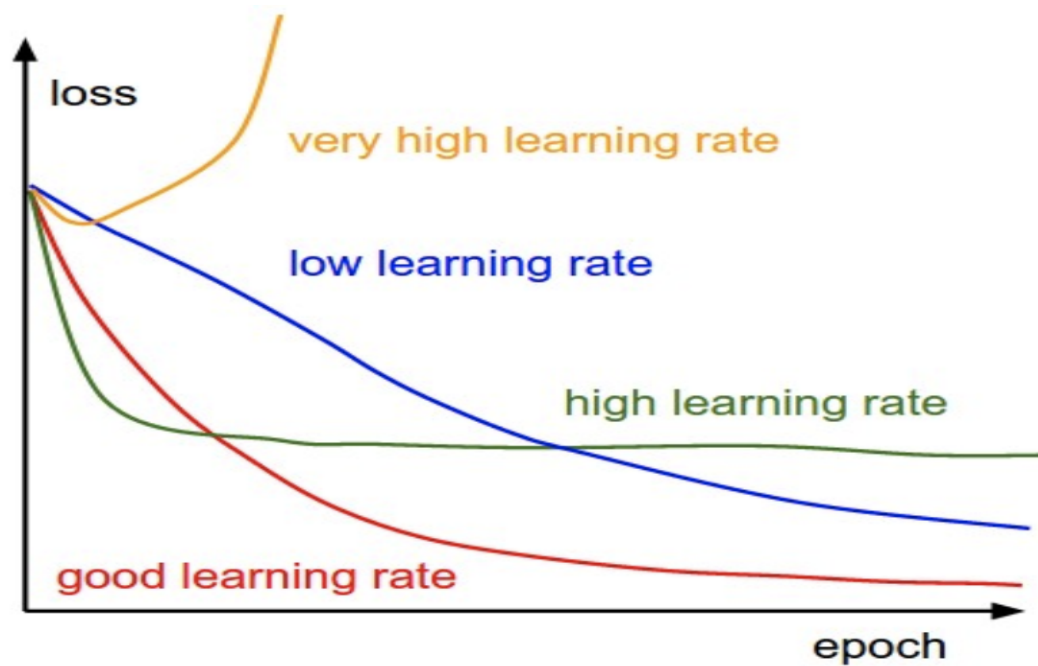
Figure 2.7: A hand-drawn figure showing the impact of the choice of the learning rate parameter on the shape of the loss function. The optimal choice has a nice slowly decaying shape that we will use as a benchmark when tuning the learning rate in our applications. Copied from the cs231 course material from Stanford authored by Karpathy (2019).

2.10.1 Momentum Gradient Descent

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

$$\begin{aligned} \mathbf{v}_n &= \beta\mathbf{v}_{n-1} + (1-\beta)\nabla f(\mathbf{x}_n), \\ \mathbf{x}_{n+1} &= \mathbf{x}_n - \eta\mathbf{v}_n. \end{aligned} \tag{2.55}$$

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 there is backwards in 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 bee-lining 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

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

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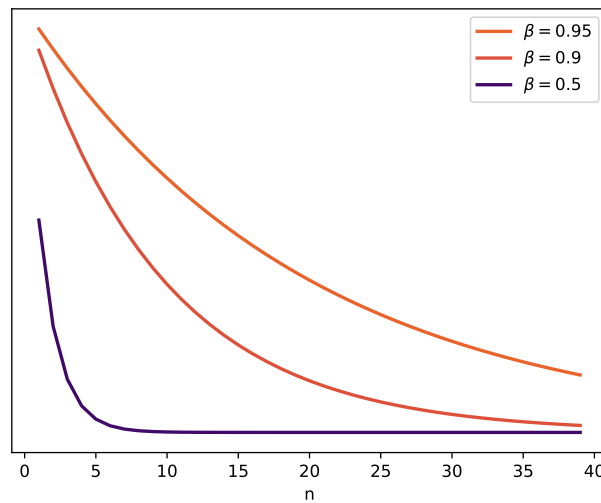
Figure 2.8: A figure illustrating the decay rate of different choices of β. The lines go as β^n and shows that one can quite easily infer how many last steps is included for each choice. A good starting value for the parameter has been empirically found to be $\beta = 0.9$ for many applications. In this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

which encourages exploration of the loss-surface rather than strong convergence to the nearest minimum. If we set the batch size $N = 1$ 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 on performing full-set gradient updates (Keskar et al. (2016)). This effect is illustrated in figure 2.9.

(a) Network F_2 (b) Network C_1
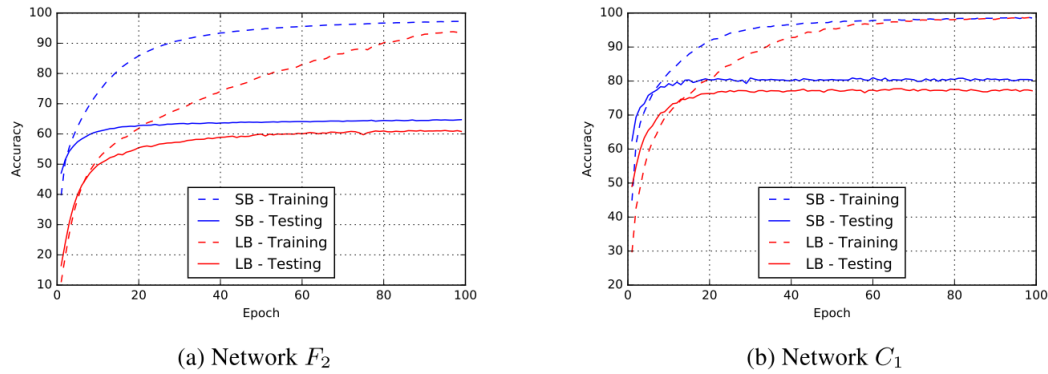
Figure 2.9: Showing the effect of batch sizes on a fully connected and shallow convolutional network in figure (a) and (b) respectively. The smaller batch-sizes are consistently able to find minima of a higher quality than the large batch versions of the same network. The networks were trained on common machine learning datasets for illustrative purposes. Figure taken from Keskar et al. (2016)

2.10.3   adam

One of the major breakthroughs in modern machine learning is the improvements made to gradient descent from the most simple version introduced in equation 3.12 to the adam paradigm introduced by Kingma and Lei Ba (2015). Since it's conception adam has become the de facto solver for most 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 a exponentially decaying average over previous first and second moments of the derivative. Physically this is akin to maintaining a 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

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1)\nabla f(\mathbf{x}_{t,i}), \qquad (2.56)$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2)\nabla f(\mathbf{x}_{t,i})^2. \qquad (2.57)$$

The update now maintains two β parameters analogous to the simple momentum update rule. In the paper Kingma and Lei Ba (2015) 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

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \qquad (2.58)$$
$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \qquad (2.59)$$

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

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{\eta}{\hat{v}_n + \epsilon}\hat{m}_n. \qquad (2.60)$$

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 labeled 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 TP (True Positive) predictions and the TN (True Negatives) divided by the total number of samples. We will use the FP (False Positives) and FN (False Negatives) 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

$$\text{accuracy} := \frac{TP + TN}{FN + TN + TP + FP}. \tag{2.61}$$

One of the principal failings of accuracy as presented in equation 2.61 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. (2019). 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 2.62 and 2.63 respectively.

$$\text{recall} = \frac{TP}{TP + FP} \tag{2.62}$$

$$\text{precision} = \frac{TP}{TP + FN} \tag{2.63}$$

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 2.64.

$$f1 = 2 \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{2.64}$$

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   Labeled samples

One of the principal challenges with the experimental data discussed in this thesis is that labeled 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 labeled data on a semi-supervised approach as listed in section 4.6. Starting from a random, small, sample of the labeled data we train a classifier on a subset of the labeled 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 2.14 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. For some applications we do not have measured outcomes of states and are left with only the data itself. In that 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 (Wishart and Neyman (1950)), 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 follow 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 looses 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're trying to solve. A naive approach is to find the eigenvectors of the co-variance matrix and project the data along those with the highest corresponding eigenvalues. This method is called PCA (principal component analysis, Marsland (2009)) 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 $\mathbf{Z} = \langle \mathbb{X}^T \mathbf{X} \rangle$ be the covariance matrix of our data. The principal components, $\mathbf{p}_i$, are then the eigenvectors of Z

$$\lambda_i \mathbf{Z} = \mathbf{p}_i \mathbf{Z}, \tag{2.65}$$

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, $\mathbf{x}^T \mathbf{x}'$, and the Gaussian kernel, $e^{-\|\mathbf{x}-\mathbf{x}'\|_2^2}$. The representations of the data in the kernel space is never evaluated, we only compute the associated inner-product space allowing for high dimensional kernels to be used (Schölkopf et al. (1996)).

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 the neural network, and more specifically the autoencoder. The neural network architecture known as autoencoders describe

models which aim to achieve efficient information-bottlenecking of complex data by reconstruction. To enact 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 $\psi : \mathcal{R}^n \to \mathcal{R}^m$, $n >> m$, 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. $\phi : \mathcal{R}^m \to \mathcal{R}^n$, and is known as the decoder. And the objective of this model is the reconstruction of the input which means the compression map ψ is optimized to capture salient information about the data. This 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 (Fertig et al. (2018)).

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 labeled data exists and where there exists some subset of labeled data. Autoencoders and how to encourage semantic information in the compressed representation is discussed in greater detail in section 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 ARS (adjusted rand score) 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 Hubert and Arabic (1985) the ARS computes the agreement between the two clusterings d_i and c_j. As with the supervised metrics we use the $scikit-$learn implementation of the ARS (Pedregosa et al. (2011)). The computa-

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

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

tion is performed as shown in equation 2.66:

$$\text{ARS} = \frac{\sum \binom{n_{ij}}{2} - \left[\sum \binom{x_i}{2} \sum \binom{y_j}{2} \right] / \binom{n}{2}}{\frac{1}{2} \left[\sum \binom{x_i}{2} + \sum \binom{y_j}{2} \right] - \left[\sum \binom{x_i}{2} \sum \binom{y_j}{2} \right] / \binom{n}{2}}. \tag{2.66}$$

In equation 2.66 we introduce the terms x_i and y_i which are just the row-wise and column wise sums of the 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

3.1 Introduction

Deep learning is the region of machine learning that concerned with neural network based algorithms. A neural network is a computational structure which generalizes the logistic and linear regression framework introduced in the previous chapter to include many intermediary computations before making a prediction.

The research question being explored in this thesis is to what degree we can extract salient information about different nuclear reactions occurring in a AT-TPC (active target time projection chamber) experiment using modern deep learning methods. To achieve this we employ the DRAW algorithm (Gregor et al. (2015)) and variations of a traditional autoencoder in conjunction with logistic regression and various clustering algorithms. Both of these architectures are firstly used to explore our ability to create class-separating compressions. Secondly we modify those algorithms slightly to investigate the feasibility of applying clustering techniques on the compressed space.

Building up to the algorithms used for analysis in this thesis we start by introducing the basic neural framework in section 3.2. Moving forward we discuss the algorithm for optimizing such networks with a gradient descent procedure as discussed in section 2.10. We also consider the fundamental constituents of the neural network; the layer structure and the activation function. A neural network consists of multiple layers that transform the input in a way such that it can be used in e.g. classification. This layer can be an inner product like how we formulate linear regression in equation 2.38 stacked on top of each-other, or something more complex considering the structure of the data. Images, time-series and other data types warrant different models, all of which we touch on briefly. Between each of these layers a non-linear function is applied to further enhance the expressibility of the model, the choice of which 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 differ-

ent models in conjunction with choice tools for the measurement of performance and a framework for hyper-parameter tuning.

3.2   Neural Networks

While the basis for the modern neural network was laid more than a hundred years ago in the late 1800's what we think of as neural networks in modern terms was proposed by McCulloch and Pitts (1943). They described a computational structure analogous to a human neuron. Dubbed an Artificial Neural Network (ANN) it 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 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

make proper figure for ann
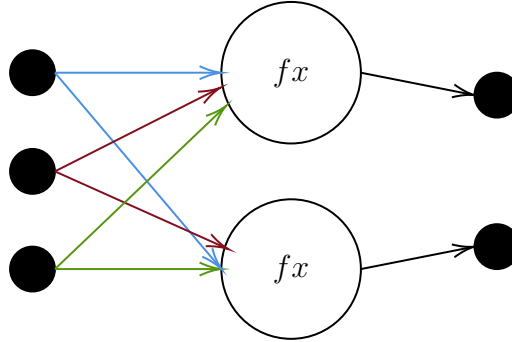
Figure 3.1: An illustration of the graph constructed by two artificial neurons with three input nodes. Colored lines illustrate that each of the input nodes are connected to each of the neurons in a manner we denote as fully-connected.

The ANN produces an output by a "forward pass". 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

$$\mathbf{W}^{[l]} \; \forall \; l \in \{1, \, 2, \, \dots, l\},$$

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. Using 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.

A layer in a network implements what we will call a forward pass which transforms the input to an intermediate representation $\mathbf{a}$ by the matrix product with the layers weights and subsequently applying an activation

$$\mathbf{a}^{[1]} = f(\mathbf{x}\mathbf{W}^{[1]})_D. \tag{3.1}$$

In equation 3.1 the subscript denotes that the function is applied element-wise.

Each node is additionally associated with a bias, ensuring that even zero valued neurons 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 3.1 then becomes

$$\mathbf{a}^{[1]} = f(\mathbf{x}\mathbf{W}^{[1]} + \mathbf{b}^{[l]})_D. \tag{3.2}$$

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

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

$$\mathbf{y} = o(\mathbf{a}^{[1]}\mathbf{W}^{[2]} + \mathbf{b}^{[2]})_D. \tag{3.4}$$

The difference between the intermediate representations $\mathbf{a}^{[l]}$ and the output $\mathbf{y}$ is largely semantic. We find it useful to separate the notation to provide clarity for what constitutes the last layer output, and because the last layer of a network usually has an activation which differs from the rest of the network. We denote this with the function $o(\cdot)$. 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 in the output we use the soft-max function. For a network with n layers the soft-max function is defined as

$$o(\mathbf{z})_i = \frac{e^{z_i^{[n]}}}{\sum_i e^{z_i^{[n]}}}, \tag{3.5}$$

where we introduce input to the activation function at each layer z_j. We explicitly define this input as

$$z_j^{[l]} = a_i^{[l-1]} W_{ij}^{[l]} + b_j^{[l]}, \tag{3.6}$$

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

$$a_j^{[l]} = f(z_j^{[l]})_D. \tag{3.7}$$

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 we can then be described in terms of its forward pass as

$$\mathbf{a}^{[1]} = f(\mathbf{x}\mathbf{W}^{[1]} + \mathbf{b}^{[1]})_D, \tag{3.8}$$

$$\mathbf{a}^{[2]} = f(\mathbf{a}^{[1]}\mathbf{W}^{[2]} + \mathbf{b}^{[2]})_D, \tag{3.9}$$

$$\vdots \tag{3.10}$$

$$\mathbf{y} = o(\mathbf{a}^{[n-1]}\mathbf{W}^{[n]} + \mathbf{b}^{[n]})_D. \tag{3.11}$$

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 $b_j^{[l]}$. 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 $b_j^{[l]}$. 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 the familiar form of the MSE (mean squared error) or in the event that we want to estimate the probability of an event under the data; the binary cross-entropy. We discuss the cost-function in general and especially the MSE and BCE (binary cross-entropy) in chapter 2. Regardless if the cost the optimization problem is solved by a gradient descent procedure which we re-introduce on update form as

$$W_{ij}^{[l]} \leftarrow W_{ij}^{[l]} - \eta \frac{\partial \mathcal{C}}{\partial W_{ij}^{[l]}}. \tag{3.12}$$

3.2.1   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, recon-

struction or classification task at hand. Training the model requires the computation of the total derivative in equation 3.12. This is also where the biological metaphor breaks down, as the brain is almost certainly not employing gradient descent.

Backpropagation, or automatic differentiation, first described by Linnainmaa (1976) 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, and having multiple layers complicate this somewhat.

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

$$E = \mathcal{C}(\hat{\mathbf{y}}, \mathbf{y}) = \frac{1}{2n} \sum_n \sum_j (\hat{y}_{nj} - y_{nj})^2. \tag{3.13}$$

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 3.13 w.r.t the model output $y_j = a_j^{[l]}$

$$\frac{\partial E}{\partial y_j} = \hat{y}_j - a_j^{[l]}. \tag{3.14}$$

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 ReLU (rectified linear unit). This shift was especially apparent after the success of Krizhevsky et al. (2012). 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.

$$\text{ReLU}(x) = f(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise.} \end{cases} \tag{3.15}$$

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

$$H(x) = f'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise.} \end{cases} \tag{3.16}$$

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 choice of equations 3.13, 3.15 and 3.16 is not a be-all-end-all solution but chosen for their ubiquitous nature in modern machine learning. Some forays have even been made into non-linear activations or composite interactions that mimic arithmetic operation.

add citations

Returning to the optimization problem we start to unravel the backpropagation algorithm. We use equation 3.14 to find the derivatives in terms of the last parameters, i.e. $W_{ij}^{[n]}$ and $b_j^{[n]}$

$$\frac{\partial E}{\partial W_{ij}^{[n]}} = \frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j^{[n]}}\frac{\partial z_j^{[n]}}{\partial W_{ij}^{[n]}}, \tag{3.17}$$

$$= \frac{\partial E}{\partial y_j}o'(z_j^{[n]})\frac{1}{\partial W_{ij}^{[n]}}\left(a_i^{[n-1]}W_{ij}^{[n]} + b_j^{[n]}\right), \tag{3.18}$$

$$= (\hat{y}_j - y_j)o'(z_j^{[n]})a_i^{[n-1]}. \tag{3.19}$$

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

$$\frac{\partial E}{\partial b_j^{[n]}} = (\hat{y}_j - y_j)o'(a_j^{[n]}). \tag{3.20}$$

Repeating this procedure layer by layer is the process that defines the backpropagation algorithm. From equations 3.19 and 3.20 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,

$$\delta_j^n = (\hat{y}_j - y_j)o'(z_j^{[n]}). \tag{3.21}$$

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

$$\mathbf{a} \circ \mathbf{b}. \tag{3.22}$$

This short-hand lets us define equations 3.19 and 3.20 in a more compact way

$$\frac{\partial E}{\partial w_{ij}^{[n]}} = \delta_j^n a_i^{[n-1]}, \tag{3.23}$$

$$\frac{\partial E}{\partial b_j^{[n]}} = \delta_j^n. \tag{3.24}$$

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

$$\frac{\partial z_j^{[l]}}{\partial w_{ij}^{[l]}} = a_i^{[l-1]}, \tag{3.25}$$

$$\frac{\partial z_j^{[l]}}{\partial b_j^{[l]}} = 1. \tag{3.26}$$

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

$$\delta_j^l = \sum_k \frac{\partial E}{\partial a_k^{[l+1]}} \frac{\partial a_k^{[l+1]}}{\partial z_k^{[l+1]}} \frac{\partial z_k^{[l+1]}}{\partial a_j^{[l]}} \frac{\partial a_j^{[l]}}{\partial z_j^{[l]}}, \tag{3.27}$$

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

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 3.28 giving a final expression for δ_j^l,

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{jk}^{[l+1]} f'(z_j^{[l]}). \tag{3.29}$$

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

$$\frac{\partial E}{\partial w_{jm}^{[l]}} = \delta_j^l a_m^{[l-1]}, \tag{3.30}$$

$$\frac{\partial E}{\partial b_j^{[l]}} = \delta_j^l. \tag{3.31}$$

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**

$\quad\left| \begin{array}{l} \frac{\partial E}{\partial w_{jm}^{[l]}} \leftarrow \hat{\delta}_j^{l+1} a_m^{[l]}; \\[2mm] \frac{\partial E}{\partial w_{jm}^{[l]}} \leftarrow \hat{\delta}_j^{l+1}; \\[2mm] \delta_j^{l+1} \leftarrow \sum_k \delta_k^{l+1} w_{jk}^{[l+1]} f'(z_j^{[l]}) \end{array} \right.$

return $\frac{\partial E}{\partial w_{ij}^{[l]}}$ *and* $\frac{\partial E}{\partial b_j^{[l]}}$

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

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 that not only are we free to chose the activation function remarkably freely 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.

3.2.2   Neural 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 common in machine learning literature and describes 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. (2017). 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. (2017) for an illustration). This representation of stepwise transformations intuitively lends itself well to representation

by a sequences of layers. As each layer contains a transformed, compressed, representation of the data. This bottleneck property of information compression is also what motivates the use of autoencoders, neural networks that compress and reconstruct the input, as a tool when unlabeled data is plentiful and labeled data is scarce.

3.2.3   Activation functions

Building neural networks depend in a 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'll 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, while they are now mostly of interest for educational purposes they still see some use in some niche models. Through logistic regression we've already been introduced to the logistic sigmoid function, used in early classifying neural networks. Mathematically it has the form

$$\sigma(x) = \frac{1}{1 - e^{-x}}. \tag{3.32}$$

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

$$\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x)) \tag{3.33}$$

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

$$\tanh x = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{3.34}$$

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

$$\frac{d \tanh x}{dx} = 1 - \tanh^2 x \tag{3.35}$$

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
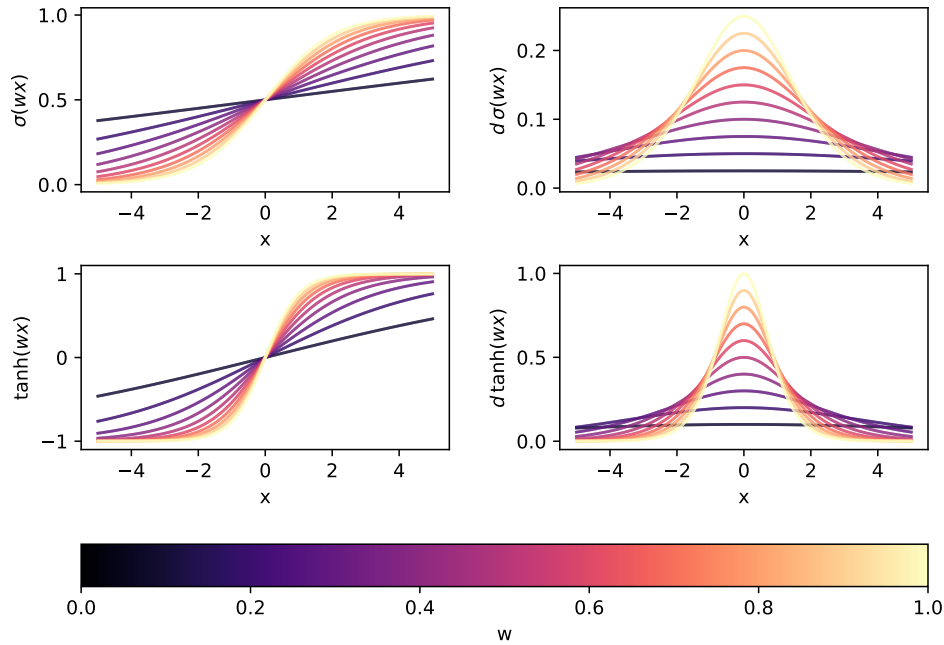
Figure 3.2: Sigmoid activation functions and their derivatives, the colorization indicates that the function is multiplied with a second variable. We observe that the derivative deteriorates to zero as the function moves away from zero. The derivative going to zero means that a network using sigmoid activations generally struggle with saturated gradients which slows down training dramatically.

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 which leads us to the next family of activation functions.

Rectifiers

On the other end of the scale rectified activations have gained traction in later years. We introduced the ReLU and it's derivative in the previous section, but we will discuss a popular cousin here. The LReLU (leaky rectified linear unit) 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
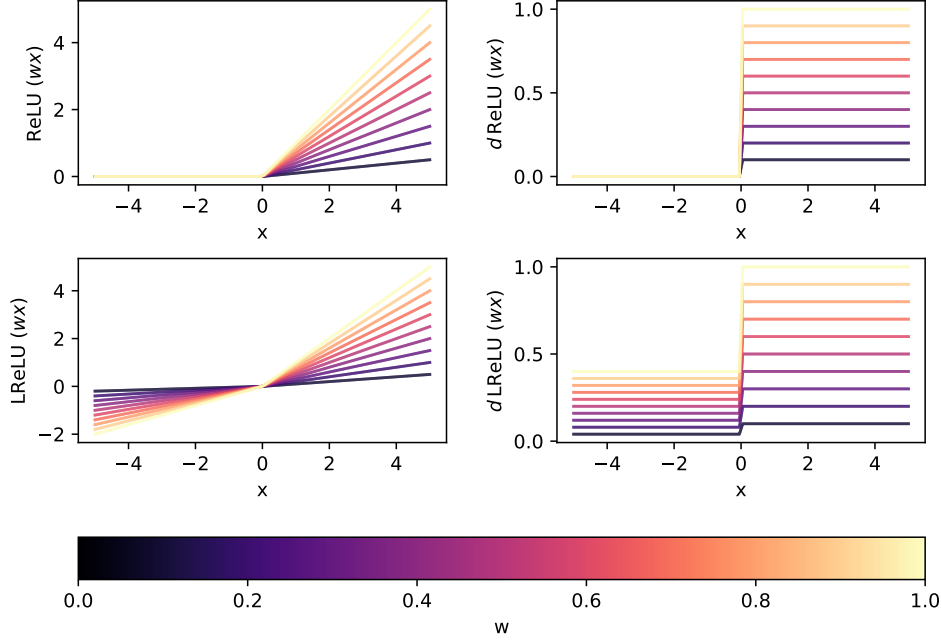
Figure 3.3: Rectifier activation functions and their derivatives, the colorization indicates that the function is multiplied with a second variable. In contrast with the sigmoid activations the derivative is non-zereo in $\mathcal{R}^+$ for the ReLU and non-zero on the entirety of $\mathcal{R}$ for the LReLU.

$$\text{LReLU}(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha \cdot x & \text{otherwise,} \end{cases} \tag{3.36}$$

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

$$\frac{d}{dx}\text{LReLU}(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ \alpha & \text{otherwise.} \end{cases} \tag{3.37}$$

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 is non-zero for a large range of value 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 a layers gradient 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'll discuss in the next section on the regularization of neural networks.

3.3   Deep learning regularization

As neural networks can be made arbitrarily complex reducing the degree of over-fitting is an important concern. Regularization of deep learning use 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 over-fitting 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 a network modeling a given process, or the optimal values for the gradient descent hyperparameters etc.

Weight constraints are similarly convenient additions to the cost function. The only modification required from 2.19 is adding a summation term over the layers. For a L_2 regularized neural net the cost function can then be written as

$$C(\hat{y}_i, f(\mathbf{x}_i; \theta)) = (\hat{y}_i - f(\mathbf{x}_i; \theta))^2 + \lambda \sum_l \sum_{ij} ||w_{ij}^{[l]}||_2^2. \qquad (3.38)$$

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

Dropout

Armed with the knowledge that neural networks usually are more complex than the problem at hand has need Srivastava et al. (2014) proposed a wonderfully simple remedy; dropout. The premise is very 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 some 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 (2015) as a means to address 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 the gradients we do not do it with respect to how much the weights in the preceding layer changes. This slows down training substantially and is what Ioffe and Szegedy (2015) describe 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 denotes 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

$$\mu_k = \frac{1}{n} \sum_i^n a_{ik}^{[l]}. \tag{3.39}$$

Secondly we compute the variance of the feature as

$$\sigma_k^2 = \sum_i (a_{ik}^{[l]} - \mu_k)^2. \tag{3.40}$$

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

$$\hat{a}_{ik}^{[l]} = \frac{\hat{a}_{ik}^{[l]} - \mu_k}{\sqrt{\sigma_k^2 + \epsilon}}, \tag{3.41}$$

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

$$\mathrm{BN}_{\gamma,\beta}(a_{ik}^{[l]}) = \gamma \hat{a}_{ik}^{[l]} + \beta. \tag{3.42}$$

The primary effect of batch normalization is then to speed up training, and empirically Ioffe and Szegedy (2015) shows that learning rates orders of magnitude larger can be used with the batch normalization procedure inserted after each layer. As bonus effects 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 this can add a substantial number of parameters, and 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.

3.3.1   Convolutional Neural Networks

Equipped with an understanding of the constituent parts of a fully connected neural network as described in section 3.2 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 AT-TPC (active target time projection chamber), which is the subject of analysis for this thesis, 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. (2019). Keeping that in mind we begin our discussion of convolutional networks with a comparison and contrast to ordinary fully connected, or dense, networks discussed previously in this chapter.

There are a couple of glaring problems with neural network layers as they were introduced in section 3.2. Firstly the number of parameters can quickly get out of hand, in a dense layer the number of parameters are the product of the input and output nodes. For a layer with 10^3 inputs and 10^2 nodes that is 10^5 parameters for just one layer. Secondly comes the question of structure. As convolutional layers were developed primarily for images, the forward pass is constructed in a way which captures any local structures in an efficient manner. However convolutional neural networks do not address variations in the rotation and scale of the objects in the image. 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, $\mathbf{F}$, are 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

$$(\mathbf{F} * \mathbf{I})_{ijk} = \sum_{f=-n'}^{n'} \sum_{g=-m'}^{m'} \sum_{h=1}^{c} I_{i+f, j+g, h} \cdot F_{fghk}. \qquad (3.43)$$

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 3.43 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 that moves one pixel from left to right per step over a square image the output is then a square matrix with size as defined in equation 3.44 for each filter.

$$O = W - K + 1 \qquad (3.44)$$

Where W is the width/height of the input and K the width/height of the kernel. In practice, however, it is 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. This operation has a very visually intuitive representation which we show in figure 3.4. The full computation of the down-sizing with these effects then is a modified version of equation 3.44, namely:

$$O = \frac{W - K + 2P}{S} + 1 \qquad (3.45)$$

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 simply selects the highest value from the set of disjoint $m \times m$ patches of the input. Where m is the pooling number, and we note that $m = 2$ results in a halving of the input in both width and height.

Originally proposed by Lecun et al. (1998) 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

(a)   **(b)**
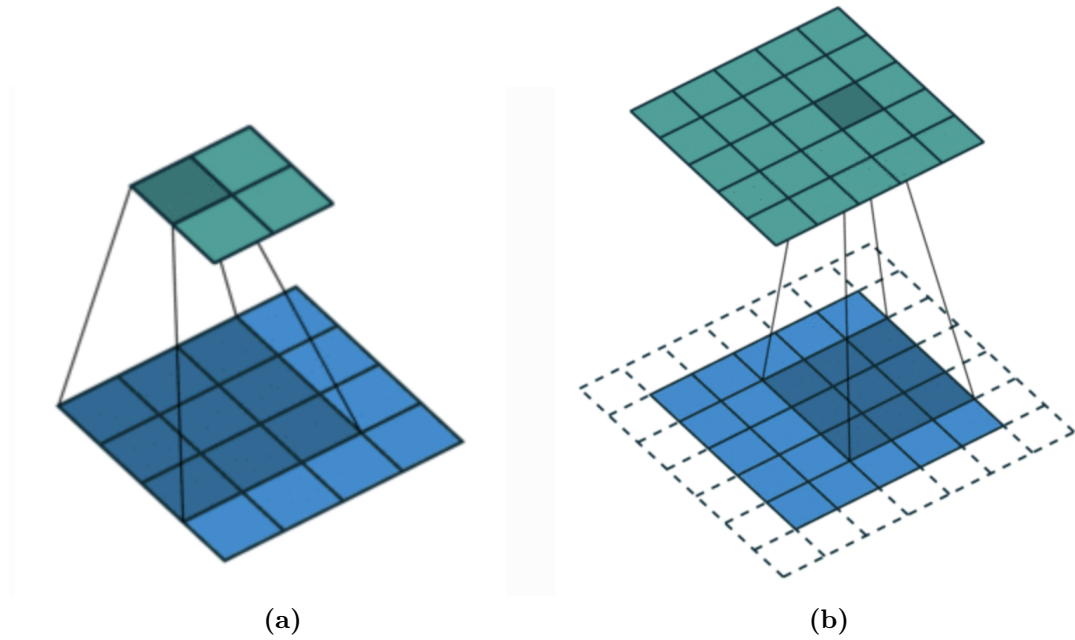
Figure 3.4: Two examples of a convolutional layers forward pass, which is entirely analogous to equation 3.6 for fully connected layers. The convolutional layer maintains a N kernels, or filters, that slides over the input taking the dot product for each step, this is the convolution of the kernel with 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 (2016)
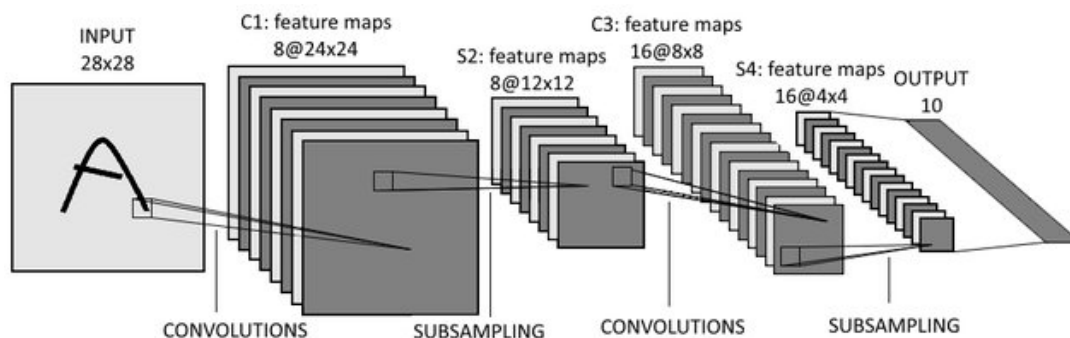
Figure 3.5: The architecture Lecun et al. (1998) used when introducing convolutional layers. Each convolutional layer maintains N kernels with initially randomized weights. These N filters act on the same input, but will extract different features from the input owing to their random initialization. The output from a convolutional layer then has size determined by equation 3.45 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 Max-Pooling.

in 2012 (Krizhevsky et al. (2012)) 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 abused 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 (Szegedy et al. (2014)). Residual networks used skip connections, passing the original data forward to avoid vanishing graadients, and batch normalization (discussed in section 3.3). In this thesis however, the number of classes and amount of data is still far less complex than the cases where these technologies 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 profit 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 sys-

---

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

tem memory so the authors ended up implementing an impressive load-sharing paradigm (Krizhevsky et al. (2012)). Modern consumer grade GPUs have up to $10GB$ of memory 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.

3.4   Recurrent Neural Networks

3.4.1   Introduction to 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 when the network predicts. The model predictions enact a sequence which has led to applications in the generation of text, 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 (1989). A simple graphical representation of the RNN cell is presented in figure 3.6
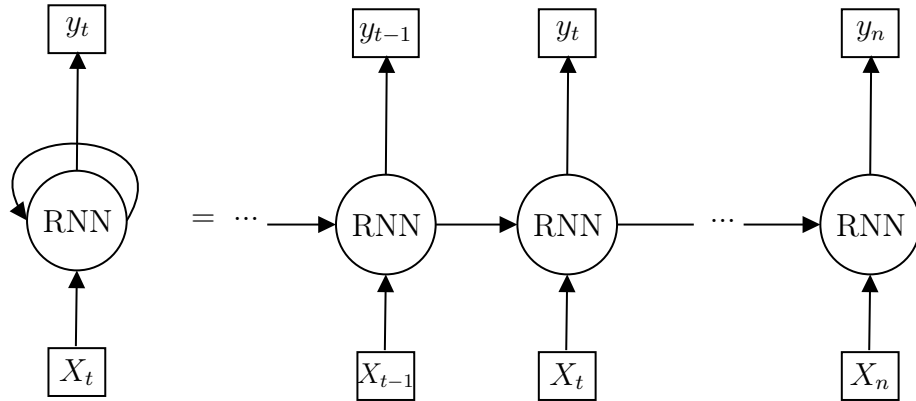
Figure 3.6: A graphical illustration of the RNN cell. The self-connected edge in the left hand side denotes the temporal nature we unroll on the right side. The cell takes as input a state vector and an input vector at time t, and outputs a prediction and the new state vector used for the next prediction. 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 3.2 trained with back-propagation.

The memory encoded by the RNN cell is encoded as a state variable. And while figure 3.6 gives a good intuition we will elaborate this by introducing the surprisingly simple forward pass structure for simple 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 3.46.

$$\langle [X_t, h_t] | W \rangle + b = h_{t+1} \tag{3.46}$$

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 the machine learning community 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
   $\mathbf{F} \leftarrow \textbf{concatenate}((h_t, \mathbf{X}_t), \text{axis}= 1)$;
   $\mathbf{h}_{t+1} \leftarrow \textbf{matmul}(\mathbf{F}, \mathbf{W}) + \text{b}$;
   return $\mathbf{h}_{t+1}$

Recurrent architectures present the researcher with a set of tools to not only model sequences but using a sequential structure to avoid common problems with e.g. variational autoencoders. This is the solution proposed by Gregor et al. (2015) with their DRAW algorithm that sequentially draws on a canvas to create realistic looking output images where a non-sequential autoencoder encounters 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

one to one   one to many   many to one   many to many   many to many
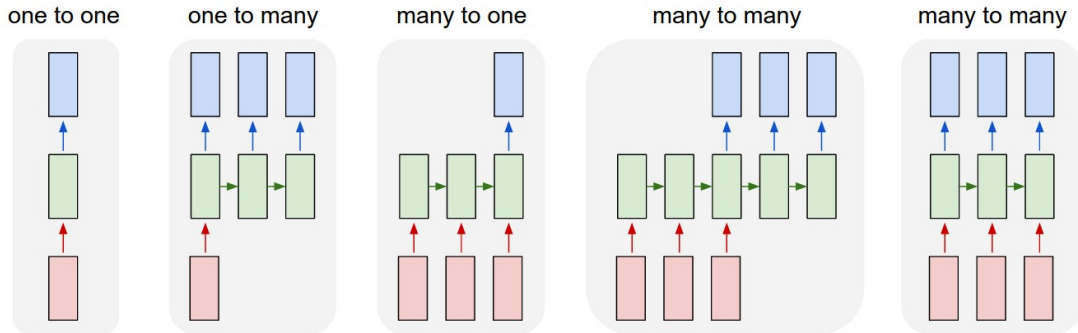
Figure 3.7: The advent of recurrent networks enabled machine learning researchers to both model complex sequential behaviors like understanding patterns in text as well as using sequences to predict a single multivariate outcome and more. The leftmost figure **1)** represents an ordinary neural network, 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 Karpathy (2015)

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 (2015).

3.4.2   Long short-term memory cells

- natural extension of RNN, where RNN carries the entire long-term dependency LSTMs introduce forgetting parts of the history

- implements gates in the architecture using sigmoid activations

Chapter 4

Autoencoders

4.1 Introduction to autoencoders

The primary challenge we face with data from the AT-TPC (active target time projection chamber) is that most, if not all, data is not labeled. Given that fact and the huge amount of data 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 over some data by reconstruction. The interesting part of the algorithm is in many applications that it is in the family of latent variable models. This means that the model encodes the data into a much lower dimensional latent space before reconstruction. 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 non linear map ψ

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

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

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

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

$$\mathcal{O} = \underset{\phi, \psi}{\arg \min} \, ||X - \phi \circ \psi||^2, \tag{4.1}$$

where the $\circ$ operator denotes function composition in the standard manner. As the name implies the encoder creates a lower-dimensional "encoded" representation of the input. This objective function is optimized by a mean-squared-error cost 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 PCA (principal component analysis)(Marsland (2009)) which we introduced in chapter 2. In practice the non-linear maps, ψ and ϕ, are most often parametrized by neural networks. We described neural networks some detail in section 3.2.

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 $P(\mathcal{X})$.

The first of these employed a Kullback-Leibler divergence and were dubbed variational autoencoders, or VAEs (Kingma and Welling (2013)). 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 (Higgins et al. (2017), Zhao et al. (2017), Fertig et al. (2018)).

4.2   The Variational Autoencoder

Originally presented by Kingma and Welling (2013) 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 is only interesting as a way of describing the latent space. Our efforts largely 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.

---

[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 MNIST dataset

4.2.1   The variational autoencoder cost

In section 4.1 we presented the structure of the autoencoder rather loosely. 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 makes clear how we aim to impose 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 from. 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, $p(z|x)$ 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

$$p(z|x) = \frac{p(x|z)p(z)}{p(x)}. \tag{4.2}$$

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

$$p(x) = \int_z p(x|z)p(z) \tag{4.3}$$

The integral in the denominator is intractable for most interesting problems, as the pace over z is often combinatorially large.

This is also the same problem that MCMC (Markov chain Monte Carlo) methods are commonly applied to. In physics this family of algorithms has been applied to solve many-body problems in quantum mechanics primarily by gradient descent on variational parameters . citation-Comph-phys 2 compendium

Variational Bayesian methods we can then summarize as being techniques for estimating computationally intractable integrals as the one expressed in 4.3. To derive a solution we begin by introducing the Kullback-Leibler divergence (KL-divergence) (Kullback and Leibler (1951)) which is a measure of how much two distributions are alike, it is important to not that it is however not a metric. We define the KL-divergence in equation 4.4 from a probability measure P, to another Q, by their probability density functions p, q over the set $x \in \mathcal{X}$

$$D_{KL}(P||Q) = -\int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx, \tag{4.4}$$

$$= \langle \log\left(\frac{p(x)}{q(x)}\right) \rangle_p. \tag{4.5}$$

In the context of the VAE the KL-divergence is a measure of the quality of P approximating Q (Burnham et al. (2002)). The first part of deriving the cost is then to introduce an approximation of the evidence.

We also introduce the ELBO (evidence lower bound) as an approximation to the evidence following the derivation laid out by Kingma and Welling (2013). The EBLO function is defined as

$$ELBO(q) = \langle \log(p(z,x)) \rangle - \langle \log(q(z|x)) \rangle. \tag{4.6}$$

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

$$ELBO(q) = \langle \log(p(z)) \rangle + \langle \log(p(x|z)) \rangle - \langle \log(q(z|x)) \rangle, \tag{4.7}$$

$$= \langle \log(p(x|z)) \rangle - D_{KL}(q(z|x)|p(z)) \tag{4.8}$$

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

$$f(\langle x \rangle) \geq \langle f(x) \rangle. \tag{4.9}$$

From the definition of the evidence we then have

$$\log(p(x)) = \log \int_z p(x|z)p(z), \tag{4.10}$$

$$= log \int_z p(x|z)p(z) \frac{\psi(z|x)}{\psi(z|x)}, \tag{4.11}$$

$$= log \langle p(x|z)p(z)/\psi(z|x) \rangle, \tag{4.12}$$

$$\overset{(J)}{\geq} \langle \log(p(x|z)p(z)/\psi(z|x)) \rangle, \tag{4.13}$$

$$\geq \langle \log(p(x|z)) \rangle + \langle \log(p(z)) \rangle - \langle \log(\psi(z|x)) \rangle, \tag{4.14}$$

$$\geq \langle \log(p(x|z)) \rangle - D_{KL}(\psi(z|x)||p(z)). \tag{4.15}$$

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

Finally we 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

$$D_{KL}(\Psi||Q) = \langle \log \left(\frac{\psi(z|x)}{q(z|x)} \right) \rangle_z, \tag{4.16}$$

$$= \langle \log \left(\psi(z|x) - \log q(z|x) \right) \rangle_z. \tag{4.17}$$

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

$$D_{KL}(\Psi||Q) = \langle \log \left(\psi(z|x) \right) - \log \left(\phi(x|z)q(z) \right) + \log(q(x)) \rangle_z, \tag{4.18}$$

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

$$D_{KL}(\Psi||Q) - \log q(x) = \langle \log \psi(z|x) - \log q(z) - \log \left(\phi(x|z) \right) \rangle_z, \tag{4.19}$$

Note that the term $-\langle \log \left(\phi(x|z) \right) \rangle_z$ 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

$$\log(p(x)) - D_{KL}(\Psi||Q) = \langle \log \left(\phi(x|z) \right) \rangle_z - D_{KL}(\psi(z|x)||q(z)). \tag{4.20}$$

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 4.2. This problem is solved by recognizing that the right hand side is the ELBO and so our model fits on the lower bound of the evidence.

Kingma and Welling (2013) 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 3.2.1.

4.3   Optimizing the variational autoencoder

From section 4 we observe that the optimization is split in two. A reconstructive 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 would like to simplify the second to conserve computational resources. Thankfully this is simple given some assumptions on the target 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

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} exp(-\frac{1}{2}(\mathbf{x} - \mu)^T \Sigma^{-1}(\mathbf{x} - \mu)), \qquad (4.21)$$

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

$$D_{KL}(p_1||p_2) = \frac{1}{2}\left(\log\frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2 - \mu_1)^T\Sigma_2^{-1}(\mu_2 - \mu_1)\right), \quad (4.22)$$

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

$$D_{KL}(q||p) = \frac{1}{2}\sum_i -\log\sigma_i^2 - 1 + \sigma_i^2 + \mu_i^2. \qquad (4.23)$$

We note that equation 4.23 satisfies the equality that the divergence is zero when the target and model distributions are equal. An important feature of the Kullback-Leibler divergence is that it operates point-wise on the probability densities. Which was the topic of the argument presented by Zhao et al. (2017) when proposing 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.

4.4   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 the 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. (2017) introducing a Lagrange multiplier to the traditional KL divergence term, and the Info-VAE paper by Zhao et al. (2017) 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

$$D_{MMD} = \langle k(z, z')\rangle_{p(z),p(z')} - 2\langle k(z, z')\rangle_{q(z),p(z')} + \langle k(z, z')\rangle_{q(z),q(z')}. \qquad (4.24)$$

Which does not have a convenient closed form like the the Kullback-Leibler divergence and so adds some computational complexity.

Recent research by Seybold et al. (2019), 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 (Seybold et al. (2019)). To combat this problem they introduce a change in the optimization objective by adding a second decoder term to the optimization task

$$\langle \phi(x|z) + \lambda \phi'(x'|z)\rangle + \beta D(p||\psi). \qquad (4.25)$$

The second decoder term reconstructs a different representation of the sample x, and the change is dubbed as a dueling decoder model. In this work we will consider a dueling 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 4.6 is that the probability of a pixel in the output being activated is not conditional on whether the pixels surrounding it has is activated. This means that the entire canvas is

---

[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 $k(x, x) = e^{-\frac{||x-x'||^2}{2\sigma}}$

conditioned on a single sample. The Deep Recurrent Attentive Writer (DRAW) aims to solve this problem by creating an iterative algorithm updates parts of, or the whole canvass, multiple times (Gregor et al. (2015)). 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. (2019)

- The attentive part of DRAW as described by Gregor et al. (2015) 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 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. (2017)

At the core of the DRAW algorithm sits a pair of encoder of decoder networks, making it part of the autoencoder sub-family of neural networks. This familiar core is then wrapped in a recurrent framework with LSTM cells that acts as the encoder/decoder pair. We use the same notation as Gregor et al. (2015) 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 that will be discussed in the next section. Next the encoder hidden state, $\mathbf{h}_t^{enc}$, 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 forward into the network. The following equations from Gregor et al. (2015) summarizes the DRAW forward pass

$$\hat{\mathbf{x}} = \mathbf{x} - \sigma(\mathbf{c_{t-1}}), \tag{4.26}$$

$$\mathbf{r}_t = \text{read}(\mathbf{x}_t, \hat{\mathbf{x}}_t), \tag{4.27}$$

$$\mathbf{h}_t^{enc} = \text{RNN}^{enc}(\mathbf{h}_{t-1}^{enc}, [\mathbf{r}_t, \mathbf{h}_{t-1}^{dec}]), \tag{4.28}$$

$$\mathbf{z}_t = \text{latent}(\mathbf{h}_t^{enc}), \tag{4.29}$$

$$\mathbf{h}_t^{dec} = \text{RNN}^{dec}(\mathbf{h}_{t-1}^{dec}, \mathbf{z}_t), \tag{4.30}$$

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \text{write}(\mathbf{h}_t^{dec}), \tag{4.31}$$

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 4.1.
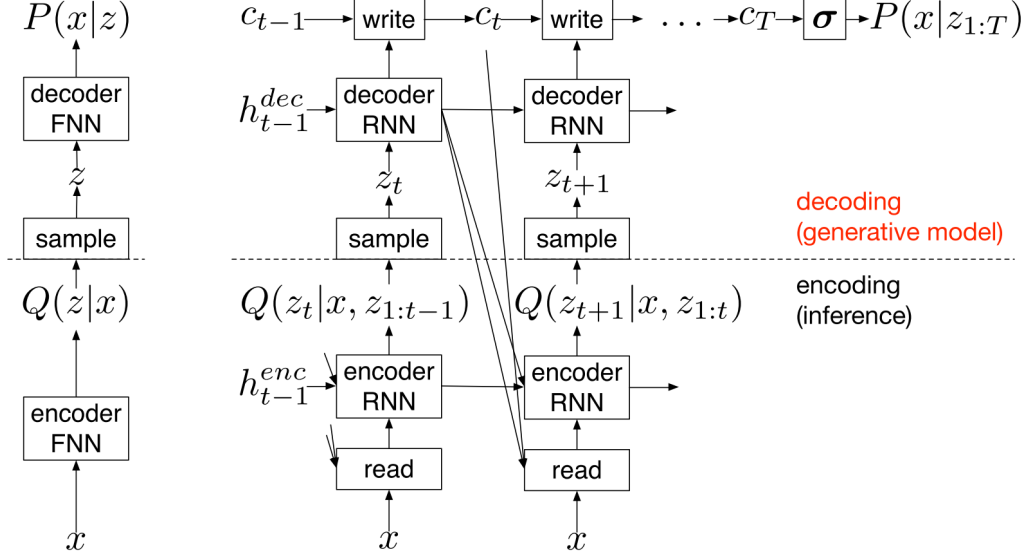
Figure 4.1: Left: an ordinary oneshot encoderdecoder network. **Right:** DRAW network that iteratively constructs the canvas using RNN cells as the encoder/decoder pair. The final output is then iteratively constructed using a series of updates on a canvass, c_t. DRAW function read that process the input 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.

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 versions of which is just the concatenation of the error image with the input for the read-function and a weight transformation from the decoder state to the output dimension as the write. This pair of functions have not been considered for this work.

Instead of the trivial implementations the DRAW network implemented grids of Gaussian filters to extract patches of smoothly varying location and size (Gregor et al. (2015)). To control the patch the authors compute centers, 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. The mean location of those filters are computed from

the centers, g_X and g_Y, and the stride δ. From Gregor et al. (2015) the means at row i and column j are defined as

$$\mu_X^i = g_X + (i - N/2 - 0.5)\delta, \tag{4.32}$$

$$\mu_Y^j = g_Y + (j - N/2 - 0.5)\delta. \tag{4.33}$$

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

$$\tilde{g}_x, \tilde{g}_y, \log \sigma^2, \log \gamma = \text{Dense}(\mathbf{h}_t^{dec}), \tag{4.34}$$

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 in a logarithm to ensure positivity by exponentiating them prior to use. Gregor et. al makes an additional transformation on the centers 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

$$g_x = \frac{W + 1}{2}(\tilde{g}_x + 1), \tag{4.35}$$

$$g_y = \frac{H + 1}{2}(\tilde{g}_y + 1). \tag{4.36}$$

$$\tag{4.37}$$

The above equations included terms to compute and scale δ, which we elect to estimate as a constant hyperparameter. As the combination between the number of filters N and δ determines the size of the input region passed to the encoder. 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. (2019).

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

$$F_x[i, w] = \frac{1}{Z_x} e^{-\frac{(w - \mu_x^i)^2}{2\sigma^2}}, \tag{4.38}$$

$$F_y[j, h] = \frac{1}{Z_y} e^{-\frac{(h - \mu_y^j)^2}{2\sigma^2}}, \tag{4.39}$$

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. To add to the output the write function returns an array that is added to the current canvass c_t. From Gregor et al. (2015) the read function is defined as

$$\text{read}(\mathbf{x}, \hat{\mathbf{x}}, \mathbf{h}_{t-1}^{dec}) = \gamma[F_y \mathbf{x} F_x^T, F_y \hat{\mathbf{x}} F_x^T]. \tag{4.40}$$

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 $w_t \in \mathcal{R}^{N \times N}$ to ensure the matrix multiplications are sane. The write function is then defined as

$$\text{write}(w_t) = \hat{\gamma} \hat{F}_y^T w_t \hat{F}_x. \tag{4.41}$$

Notice the transposition order in equation 4.41 is reversed with respect to the order in equation 4.40.

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

$$L_z = \sum_t^T D_{KL}(q||p). \tag{4.42}$$

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

$$L_z = \frac{1}{2} \sum_t^T (\mu_t^2 + \sigma_t^2 - \log \sigma_t^2) - \frac{T}{2}. \tag{4.43}$$

Similarly the maximum mean discrepancy loss is computed from equation 4.42 replacing the Kullback-Leibler divergence with the terms from equation 4.24.

4.5.3   Deep Clustering

4.6   Neural architectures

The research question explored in this thesis necessitates two separate architectures. In the case where we have access to some labeled 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 labeled 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.

Labeled data is, however, both require a high time investment and possibly high amounts of computing power to separate. 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 labeled 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.

4.6.1   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 labeled data

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

We will determine the best autoencoder architecture for each dataset listed in section 5.3. The best autoencoder is measured by performance in identifying separate classes by the logistic regression model on a test-set of data.

4.6.2   Clustering

To attempt clustering of event data we will follow the pipeline as outlined in Guo et al. (2017). The modeling has two distinct steps, pre-training of the

convolutional autoencoder and training with regularization towards the pseudo-labels.

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 subset of labeled data.

4.6.3   Pre-trained networks

Following the precedent of Kuchera et al. (2019) 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 millions of images and classify between hundreds of distinct classes. In their work Kuchera et al. (2019) use the VGG16 architecture trained on imagenet to classify AT-TPC events, in this thesis we will build on the understanding of using this 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 (2015), they were runners up in the ISLVRC(ImageNet large scale visual recognition competition) of 2014 (Russakovsky et al. (2015)). 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 (Simonyan and Zisserman (2015)). Stacking the smaller kernels then both contribute computationally, as a regularization factor because of the reduce number of parameters and increased discriminatory power because of the added non-linearities. The full architecture is detailed in appendix B.1.

A pre-trained network can be included in the architectures in three distinct 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.

   As a baseline for the modeling pipelines we consider the VGG16 model without updating weights. 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.

clean up section on model architectures

Chapter 5

Experimental background

5.1 Introduction

1. Describe the FRIB facility and its goals

2. Describe the advent and use of ML in High Energy particle physics

3. Contextualize the need for ML in Nuclear physics, what has changed?

5.2 Active Target Time Projection Chambers

5.2.1 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 Bradt (2017).

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 $A = Z + N$. They are further categorized by equal components; nuclides with an equal number of protons are called *isotopes*, equal number of neutrons *isotones* and with the same mass *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
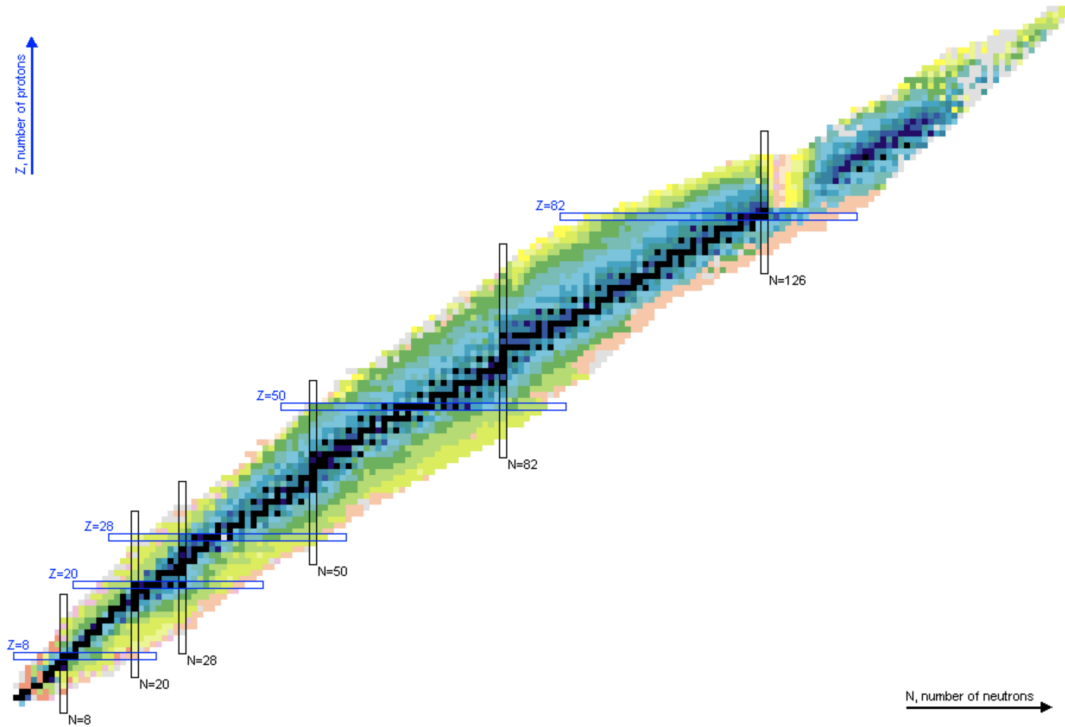
Figure 5.1: Chart of the known nuclides. The number of protons are given along the vertical axis, and neutrons along the horizontal. The color indicates the half-life of the nucleus with darker colors representing longer times. Lines of isotones and isotopes along the magic numbers are indicated by rectangles in the figure. Retrieved from Sonzogni (2019)

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 indicates that these isotones should have an approximately spherical structure. This has been disproved experimentally as deformities appear when removing protons from the spherical nucleus ^{48}Ca. Which 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 (Active Target Time Projection Chamber) at the NSCL.

Isobaric analogues

5.2.2   AT-TPC details

1. Introduce the TPC and AT-TPC

2. Introduce the objective and of the AT-TPC experiments

3. Introduce and discuss the physics of the AT-TPC experiments

4. Discuss the need for ML in event categorization in the AT-TPC experiments

5.3 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. Both data produced with simulation tools and data recorded from the active target time projection chamber (AT-TPC). For the experimental data we use data collected from a single run of the experiment.

5.3.1 Simulated ^{46}Ar events

The simulated AT-TPC tracks were simulated with the `pytpc` package developed at the NSCL (Bradt et al. (2017)). Using the same parameters as for the $Ar^{46}(p, p)$ experiment a set of $N = 4000$ events were generated per class. The events are generated in the same format as the semi-raw experimental data. That is they are represented as peak-only 4-tuples of $e_i = (x_i, y_i, t_i, c_i)$. Each event is then a set of these four-tuples: $\epsilon_j = \{e_i\}$ creating a track in three dimensional space with charge amplitude for each point. To process these events with the algorithms implemented for this thesis we chose to represent these 3D tracks as 2D images with charge represented as pixel images. For the analysis we chose to view the x-y projection of the data.

To emulate the real-data case we set a subset of the simulated data to be labeled and treat the rest as unlabeled 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 $\mathbf{X}_L \subset \mathbf{X}_F$.

5.3.2 Full ^{46}Ar events

The events analyzed in this section were retrieved from the on-going AT-TPC experiment at Michigan State University. In the experiment a beam of a particular isotope is accelerated and directed into a chamber with a gas that acts as the reaction medium and target. As reactions occur between the gas and beam ejected electrons from these drift towards the anode and the Micromegas measuring the impact over time from the reactions. The measuring apparatus is very sensitive, and though filtering is performed such that only the peaks of deposited charge the events are noisy in the ^{46}Ar experiment subject to analysis in this thesis. There is probable structural noise that can be attributed to electronics cross-talk and possible interactions with cosmic background radiation and other

Added part from previous results, needs molding

figure of 3D simulated track and 2D representation

	Simulated	Real	Filtered
Total	8000	51891	49169
Labeled	2400	1774	1582

Table 5.1: Description of the data used for analysis. 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

sources of charged particles. Indeed one of the confounding factors is that there is currently not an understanding of the physics of the major contributing factors to this noise.

describe the physics in a bit more detail boy

5.3.3   Filtered $^4 6Ar$ events

write filtered section

As we saw in the previous section the events pick up

add plots of events in 2d and 3d

add table with data descriptions. N samples, N labelled

Part II

Implementation

Chapter 6

Methods

6.1 Introduction

In this chapter we will illustrate the research pipeline applied to the experimental data from the AT-TPC. We will show the implementation of the algorithms described in section 3, and their performance on simulated data to illustrate their workings and to establish a baseline for the inquiry into the real experimental data.

The implementation of the algorithms described in chapter 3 have been implemented for this thesis in the python programming language (van Rossum and Python development team (2018)). Parts of algorithms displayed for demonstration or exposition have been developed in the numerical python framework numpy (van der Walt et al. (2011)). While variational autoencoder and DRAW algorithms were implemented in the tensorflow framework (Abadi et al. (2015)). The choice of python as the framework in which to develop this thesis was made for the ease of rapid prototyping and the extensive availability of mature numerical libraries like the ones cited above. Plots of numerical performance and data visualization was achieved with the matplotlib graphics package for python (Hunter (2007)).

We begin by describing the tensorflow framework as it with that api that the algorithms are implemented.

6.2 TensorFlow

The numerical framework TensorFlow is a development for deep learning tasks developed by Google Brain starting in 2011 (Abadi et al. (2015)). It implements a total pipeline for a very large variety of machine learning architectures. At the core of the library is the graph structure constructed during runtime. Built for iteration the update of tensor objects is statically determined in a manner close

to traditional compiled languages[1]. Python indexing and iteration in loops are notoriously slow but can be sped up considerably to the point where for modestly sized computations the gain of switching to a C style language is negligible. This speed up is achieved largely by avoiding pythons built in iterables and loop structures where possible, relying instead on interfaces to optimized C or C++ code for very efficient matrix operations.

Part of the efficiency loss that numpy sustains is another python peculiarity; for most operations (adding, multiplying, etc.) the default behavior of numpy is to return a new object according to the broadcasting rules of the input with inferred element types. This is obviously a costly behavior while very much pythonic in spirit. Later versions (> 1.9.0) allow for more operations to define an output array destination. While this allows numpy to catch up somewhat in speed the TensorFlow address to this problem solves both the challenge of object allocation and the computation of gradients so emblematic of modern machine learning.

6.2.1   The computational graph

To understand the program flow of the later algorithm implementations we begin by introducing the fundamental concepts of TensorFlow code [2]. The heart of which is the computational graph. A code snippet with an associated graph is included in figure 6.1 showing a simple program that computes a weight transformation of some input with a bias. The cost is included as the bracketed ellipses and is choses specifically for the problem. When the forward pass is unrolled it becomes available for automatic differentiation.

---

[1]We will use the term tensor in this thesis to denote the computational object unless otherwise explicitly stated

[2]The thesis code was written for the latest stable release of TensorFlow prior to the release of TF 2.0. Some modules may have moved, changed name or have otherwise been altered. 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.

```
 1  import tensorflow as tf
 2
 3  # placeholder for input to the computation
 4  x = tf.placeholder(dtype=tf.float32, name="x")
 5
 6  # bias variable for the affine weight transformation
 7  b = tf.Variable(tf.zeros(100))
 8
 9  # weight variable for the affine wegiht transformation with random values
10  W = tf.Variable(tf.random_uniform([784, 100]), tf.float32)
11
12  # activation as a function of the weight transformation
13  a = tf.relu(tf.matmul(W, x) + b)
14
15  # cost computed as a function of the activation
16  # and the target optimization task
17  C = [...]
18
19  # Start session to run the computational graph
20  session = tf.InteractiveSession()
21
22  # Initialize all variables, in this example only the weight
23  # matrix depends on an initialization
24  tf.global_variables_initializer()
25
26  for i in range(epochs):
27      result = session.run(C, feed_dict={x: data[batch_indices]})
28      print(i, result)
```

Figure 6.1: This short script describes the setup required to compute a forward pass in a neural network as described in section 3.2. 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 Abadi et al. (2015)

In general we set up the computational graph to represent the forward pass, or predictive path, of the algorithm. The remainder then is then only 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. The process of automatic differentiation we describe in detail in section 3.2. But in essence the method finds the path on the graph from I to C and then works backwards, adding to the graph as it goes, computing the partial derivatives via the chain rule. We show the gist of this process in figure 6.3. Since the operations on the partial derivatives are defined by the choice of gradient descent variation TensorFlow wraps the computation in optimizer modules for convenience. Defined in `tf.train` these include stochastic gradient descent and ADAM .

whats that damn abbreviation?
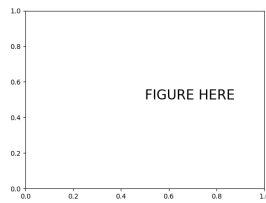
Figure 6.2: A graph representation of the short script in figure 6.1. The nodes represent TensorFlow operation types including variable declarations and operations on those including Add and MatMul operations. In the versions of TensorFlow prior to the release of TF 2.0 this graph did not execute immediately but relied on the session object. The Session.run method takes as arguments input to the graph and the end point(s) and then computes the transitive closure of all the nodes that must be executed in order to obtain said output(s) (Abadi et al. (2015)). Using tensorfow involves setting up a graph once and executing it or a few distinct subgraphs hundreds of thousands of times. This figure is a modified version of figure 2 in Abadi et al. (2015)
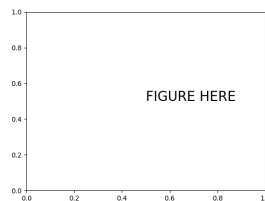
Figure 6.3

We outline a basic TensorFlow script in **??** that goes through the basic steps outlined above. This is the skeleton on which we build the more complex architecture for the DRAW algorithm.

```
 1  import  tensorflow  as  tf
 2
 3  # placeholder  for  input  to  the  computation
 4  x = tf.placeholder(dtype=tf.float32, name="x")
 5
 6  # bias  variable  for  the  affine  weight  transformation
 7  b = tf.Variable(tf.zeros(100))
 8
 9  # weight  variable  for  the  affine  wegiht  transformation  with  random  values
10  W = tf.Variable(tf.random_uniform([784, 100]), tf.float32)
11
12  # activation  as  a  function  of  the  weight  transformation
13  a = tf.relu(tf.matmul(W, x) + b)
14
15  # cost  computed  as  a  function  of  the  activation
16  # and  the  target  optimization  task
17  C = [...]
18
19  # define  optimizer  function  and  compute  gradients
20  # include  optimizer  specific  hyperparameters
21  optimizer = tf.train.AdamOptimizer(eta=0.001)
22  grads = optimizer.compute_gradients(C)
23
24  # define  update  operation
25  opt_op = optimizer.apply_gradients(grads)
26
27  # Start  session  to  run  the  computational  graph
28  session = tf.InteractiveSession()
29
30  # Initialize  all  variables, in  this  example  only  the  weight
31  # matrix  depends  on  an  initialization
32  tf.global_variables_initializer()
33
34  for  i  in  range(epochs):
35
36      # runs  the  grapgh  and  applies  the  optimization  step, running  opt_op
          will
37      # compute  one  gradient  descent  step.
38      result, _ = session.run([C, opt_op], feed_dict={x:
          data[batch_indices]})
39      print(i, result)
```

Figure 6.4: 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

6.3   Deep learning algorithms

All the models in this thesis are implemented in the python programming language using the tensorflow api for deep learning. All the models are open source and can be found in the github repository https://github.com/ATTPC/VAE-event-classification. In this section we will be detailing the general framework that the models have been built on. The structure is straightforward with

a model class implementing shared functions and usage between models. Two subclasses are implemented, one for the sequential DRAW model (discussed in section 4.5) and one for the non-sequential convolutional autoencoder 4. A couple of helper classes are also defined to manage mini-batches and the random search of hyperparameters. 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 model class implements two main functions compute_gradients and compile_model that make calls to the model specific functions constructing the computational graph and subsequently the gradients of a specified cost wrt. the output(s) of that graph. Computing the gradients is a straight forward procedure in TensorFlow given an optimizer class which computes the gradient update operations.

The LatentModel class contains the framework and functions used for common training operations. In the initialization of the class it mostly defines self assignments but two calls are worth notice as we explicitly clean the graph before any operations are defined. Secondly an iteration is made through a configuration dictionary to define 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 or not to restore the weights of a previous run from a directory, this directory is supplied to the train method of LatentModel class.

After initialization and before training the subclasses of LatentModel needs to construct the computational graph defining the forward pass. As well as the pertinent operations, these include the loss components, the latent space sample and the backwards gradient descent pass. This is done via a wrapper function compile_model defined in LatentModel that takes two dictionaries for the graph and loss configuration. They are subclass specific and will be elaborated on later in sections 6.4 and 6.5. The method also sets the compiled flag to True which is a prerequisite for the compute_gradients method.

When the model is compiled the gradients can be computed and the fetch-object for the losses prepared. This general setup is entirely analogous to what was included in the script in listing 6.4 with some small additions. To avoid the problem of exploding gradients we employ the common trick of clipping the gradients by their L_2 norm. This is particularly useful for experiments with $ReLU$ activations. The procedure is implemented in the method compute_gradients. The fetch object contains the loss components, the backwards pass operation as well as the latent sample(s) and decoder state(s). This list of operations (defining a return value for the graph) is fed to a session object for execution at train time or for inference. The runtime philosophy is that a TensorFlow op is not run before the graph gets notice that something that depends on that op is being computed. In the same vein as the compile_model method the compute_gradients method sets the flag grad_op to True when it is through.

The training procedure is implemented in the train method which 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 clustering algorithm described in section 4.5.3. Which uses an off-the-shelf version of the K-means algorithm () to find the initial cluster locations. The main loop of the method iterates over the entire dataset and performs the optimization steps. For the clustering autoencoder an additional step is also included to update the target distribution as described in section 4.5.3

SKLEARN CITATION

6.4 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 available 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 those 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 supplied configuration dictionary passed through the compile_model method. The dictionary includes options for additional regularization terms like batch-normalization and instructs the class on what losses to compile the model with.

6.4.1 Computational graph

The private[3] 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 exploding-gradient susceptible functions and before for functions with a vanishing gradient problem. In TensorFlow the construction of the encoder is coded as

```
1 # excerpt from convolutional_VAE.py
```

---

[3]The term private is used loosely in the context of python as the language does not actually maintain private methods unaccessible to the outside. By convention methods that are prefixed with an underscore are to be treated as private and are not exposed with public apis and in documentation.

```python
# 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
                                )(h1)
                                self.variable_summary(h1)
                    else:
                        a = activations[activation]
                        with tf.name_scope("batch_norm"):
                            if self.batchnorm:
                                h1 = BatchNormalization(
                                    axis=-1, center=True, scale=True,
```

```
          epsilon=1e-4
65                                    )(h1)
66                                    self.variable_summary(h1)
67                          h1 = a(h1)
68
69                      if self.pooling_architecture[i]:
70                          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 in the training operation. 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 runs a series of convolutional layers, respectively. Each convolutional layer is specified with a kernel size, a certain number of filters and the striding of the convolutions. We also use a trick from Guo et al. (2017) to ensure 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. Depending on whether the output width(height) is an integer multiple of 2^n, where n is the number of layers, the last convolution is adjusted to have no zero-padding if this is the case.

After each layer the specified non-linearity is applied. This is one of the sigmoid activations (logistic sigmoid or hyperbolic tangent) or the rectified linear unit family of activations[4]. 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 explode so the output is scaled. The output from the convolutional layers is then an object with dimensions h = (o, o, f) where f is the number of filters in the last layer and $o = \frac{H}{2^n}$ with n denoting the number of layers with stride 2 or the count of MaxPool layers and H the input image size.

The tensor output from the convolutional layers is then transformed to the latent space with either a simple dense transformation, e.g. z = Dense(flatten(h)). Or if a variational loss is specified a pair mean and standard deviation tensors are constructed with dense transformations from h. These are stored as class attributes of the ConVae instance

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

Using the re-parametrization trick shown by Kingma and Welling (2013) a sample is generated by z = mu + sigma∗epsilon where epsilon is a stochastic tensor from the multivariate uniform normal distribution, $\mathcal{N}(0, 1)$. Note that the self.var

---

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

is treated as the log of the variance. The mean and standard deviation tensors are stored to be used in the computation of the loss. The latent sample is also stored retrieval or other usage.

After a sample z is drawn the reconstruction is computed with either a mirrored decoder for the naive autoencoder structure or for a VGG16 representation of the data a reconstruction is computed based on a specified decoder structure. The VGG16 representation 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.

Finally, after the decoding from the latent sample, the output is passed through a sigmoid function if the reconstruction loss is specified as a binary cross-entropy to ensure that the log-term doesn't blow up. Otherwise no transformation is applied on the output.

6.4.2   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 the same manner the losses are configured using a configuration dictionary.

In the configuration dictionary the loss for both the reconstruction and latent spaces is specified. For the reconstruction the model accepts either a mean squared error or binary cross-entropy loss, the cross-entropy is the default.

Each of these losses acts pixel-wise on the output and target images. The reconstruction loss is then stored as a class attribute ConVae.Lx which is monitored by the TensorFlow module TensorBoard for easy monitoring during training. The reconstruction loss is then implemented as

```python
1  # excerpt from convolutional_VAE.py
2  # at https://github.com/ATTPC/VAE-event-classification
3  # from commit ca9b722
4  x_recons = self.output
5  if self.use_vgg:
6      self.target = tf.placeholder(tf.float32)
7  else:
8      self.target = self.x
9  if reconst_loss == None:
10     reconst_loss = self.binary_crossentropy
11     self.Lx = tf.reduce_mean(
12         tf.reduce_sum(reconst_loss(self.target, x_recons), 1)
13     )
14  elif reconst_loss == "mse":
15     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. For a variational autoencoder the loss is a Kullback-Leibler divergence over the latent distribution and 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 4.23. This equation is relatively straightforward to implement as it just implies a sum over the mean and standard deviation tensor. The form of equation 4.23 also makes clear why we parametrize the standard deviation 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 Kullback-Leibler divergence the latent space may be regularized in the style proposed by Zhao et al. (2017). Which measures the We use the radial basis function kernel to compute the maximum mean discrepancy divergence term introduced in equation 4.24

> include implement of MMD?

6.4.3   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 section 4.6 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[5]. 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

> add sim-data to some file hosting

---

[5]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

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.

```
1  import sys
2  sys.path.append("../src/")
3  import matplotlib.pyplot as plt
4  import tensorflow as tf
5  import data_loader as dl
6  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.

```
1  x_full, x_labeled, y = dl.load_simulated("128")
2
3  fig, axs = plt.subplots(ncols=4, figsize=(14, 5))
4  [axs[i].imshow(x_full[i].reshape((128, 128)), cmap="Greys") for i in
       range(4)]
5  [axs[i].axis("off") for i in range(4)]
6  plt.show()
```


Figure 6.5: 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 regu-
larization on the latent space.

```
1  n_layers = 4
2  kernel_architecture = [5, 5, 3, 3]
3  filter_architecture = [8, 16, 32, 64]
4  strides_architecture = [2, 2, 2, 2]
5  pooling_architecture = [0, 0, 0, 0]
6
7  mode_config = {
8      "simulated_mode":False, #deprecated, to be removed
9      "restore_mode":False, #indicates whether to load weights
10     "include_KL":False, #whether to compute the KL loss over the latent
       space
11     "include_MMD":False, #same as above, but for the MMD loss
12     "include_KM":False, #same as above, but K-means. See thesis for a
       more in-depth treatment of these
13     "batchnorm":True, #whether to include batch-normalization between
       layers
14     "use_vgg":False, #whether the input data is from a pre-trained model
15     "use_dd":False, #whether to use the dueling-decoder objective
16 }
17
18 model = ConVae(
19     n_layers,
20     filter_architecture,
21     kernel_architecture,
22     strides_architecture,
23     pooling_architecture,
24     2, #latent dimension,
25     x_full,
26     mode_config=mode_config
27 )
```

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 opti-
mization. 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 reconstruc-
tion, 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.

```
1  graph_kwds = {
2      "activation": "relu",
3      "output_activation": "sigmoid", # applied to the output, necessary
       for BCE
4      "kernel_reg_strength": 1e-5
5  }
6  loss_kwds = {
7      "reconst_loss": None # None is the default and gives the BCE loss
8  }
9  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.

```
1  optimizer = tf.train.AdamOptimizer
2  opt_args = [1e-3, ] #learning rate
3  opt_kwargs = {"beta1": 0.8, "beta2":0.99}
4  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.

```
1  epochs = 200
2  batch_size = 150
3  earlystop = True
4  sess = tf.InteractiveSession()
5
6  lx, lz = model.train(
7      sess,
8      epochs,
9      batch_size,
10     earlystopping=earlystop
11 )
```

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.

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

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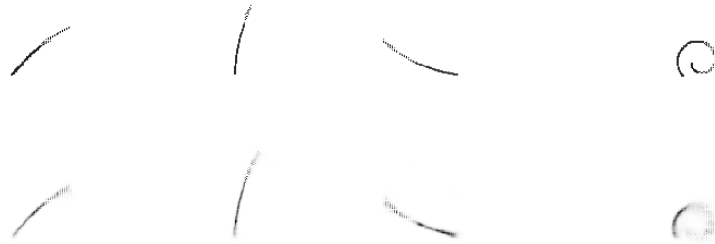

Figure 6.6: Showing four events and their corresponding reconstructions. The reconstructions faithfully reconstruct the artifacts from the simulation procedure but has a fuzzy quality common to the ELBO approximation.

From figure 6.6 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 labeled 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_labeled = x_labeled.reshape((x_labeled.shape[0], -1))
 3 latent_labeled = model.run_large(sess, model.z_seq[0], all_labeled)
 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_labeled[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)
```
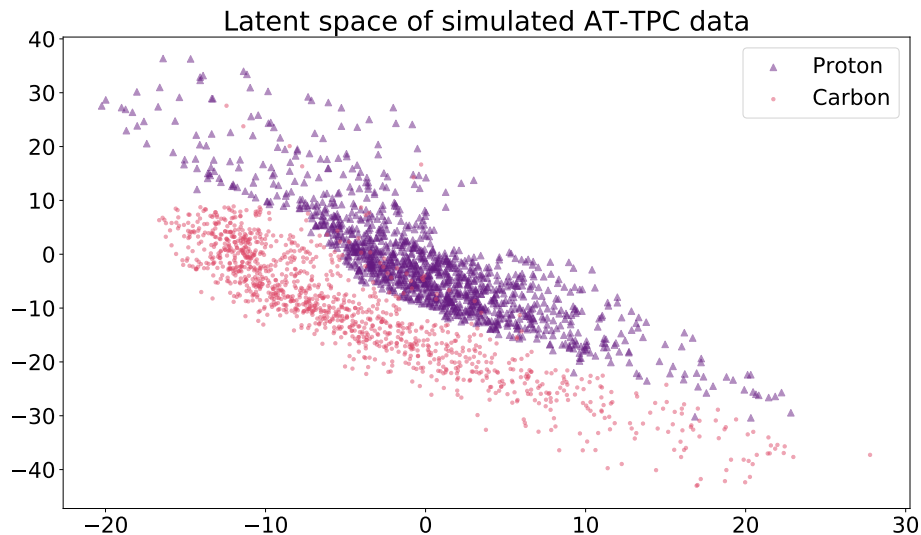

Figure 6.7: 2D latent space representation of the simulated AT-TPC data.

Now that we have prepared a latent space we can do some analysis. The semi-supervised pipeline is aimed at investigating the quality of the latent space and so we begin by

add classifier
stuff

6.5   Deep Recurrent Attentive Writer

Like the convolutional autoencoder discussed in the previous section the DRAW
(Deep Recurrent Attentive Writer) is implemented as a subclass of the LatentModel
class. Consequently it follows the same formula and implements the _ModelGraph
and _ModelLoss methods. We re-iterate from section 4.5 that the DRAW algo-
rithm 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 distribution instead of an
independent one as for the ordinary convolutional autoencoder.

6.5.1   Computational graph

In the same manner as for the ConVae model the construction of the computational
graph starts with a good amount of set-up for the graph. Our implementation
notably includes the option for an increased number of LSTM (Long Term Short
Term) cells, but most important is the extension of the paired read and write
functions to include the option for a paired set of convolutional layers. This
contrasts with the original implementation from Gregor et al. (2015) 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 time-steps
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 were written to follow the style of the set of
equations that yield equation 4.31. 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.

```python
1  # excerpt from draw.py
2  # at https://github.com/ATTPC/VAE-event-classification
3  # from commit 6b64323
4  for t in range(self.T):
5      # computing the error image
6      if t == 0:
7          x_hat = c_prev
8      else:
9          x_hat = self.x - c_prev
10
11     """ Encoder operations """
12     r = self.read(self.x, x_hat, h_dec_prev)
13     if self.batchnorm:
14         r = BatchNormalization(axis=-1, center=True, scale=True,
           epsilon=1e-4)(
15             r
16         )
17     h_enc, enc_state = self.encode(enc_state, tf.concat([r, h_dec_prev],
         1))
18     if self.batchnorm:
19         h_enc = BatchNormalization(
20             axis=-1, center=True, scale=True, epsilon=1e-4
21         )(h_enc)
22
23     """ Compute latent sample """
24     z = self.compute_latent_sample(t, h_enc)
25
26     """ Decoder operations """
27     h_dec, dec_state = self.decode(dec_state, z)
28     # dropout_h_dec = tf.keras.layers.Dropout(0.1)(h_dec, )
29     if self.batchnorm:
30         h_dec = BatchNormalization(
31             axis=-1, center=True, scale=True, epsilon=1e-4
32         )(h_dec)
33
34     self.canvas_seq[t] = c_prev + self.write(h_dec)
35
36     """
37     ... code omitted for brevity
38     """
39
40     """ Storing and updating values """
41     self.z_seq[t] = z
42     self.dec_state_seq[t] = dec_state
43     h_dec_prev = h_dec
44     c_prev = self.canvas_seq[t]
45
46     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 that are passed to the encoder-decoder pair. The location and zoom of that patch is dynamically determined at each time-step. This procedure starts with the read function and its innermost functionality. Recall from equation 4.34 that to compute the filter-

banks used for the extraction of image patches first the parameters have to be dynamically determined. Once those four are determined we compute filter-banks F_x and F_y as they are defined in 4.38 and 4.39. These equations define matrices and so we construct the grid over the exponential using the convenient tf.meshgrid which creates objects with the same dimension from different spacings. To explore the attention parameters outside the model we implemented a numpy version. Conveniently those libraries are rather homogeneous and as such the numpy is implemented to be 1-1 with the TensorFlow version.

```python
# 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 4.40. The same procedure is repeated for the write-function with bespoke parameters determined for that function.

6.5.2 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.

6.5.3   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.

We begin by considering the semi-supervised case to explore the latent configurations that yield high quality latent spaces.

Semi-supervised classification

6.6   Hyperparameter search architecture

To tune the hyperparameters of the sequential and non sequential autoencoders we implement an object oriented searching framework. A parent class ModelGenerator defines the logging variables and the type of model to be generated, i.e. 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 in that it should never be instantiated on it's own, only through its children. One subclass is implemented for the ConVAE and DRAW model classes. They share common functionality and maintain a grid over all the search able hyperparameters which we sample from to perform the search.

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 while the sequential DRAW model specifies a convolutional architecture as well as parameters for the read-write paired functions. Other flags are ours for a very wide search and vgg for a VGG16 like architecture.

Searching, saving to file and other utilities are maintained in the RandomSearch class which is instantiated with one of the ModelGenerator subclasses and implements a .search method which performs and logs the search to a specified directory.

implement
static for draw

Part III

Results

Chapter 7

Experimental setup and design

The experiments were conducted using the AI-Hub computational resource at the university of Oslo. This resource consists of three machines with four RTX 2080 Nvidia graphics cards each. These cards have about 10GB of memory available to allocate model weights. All experiments described in this section were all computed using this hardware. In this section we lay out the results obtained on the three segments of data: simulated, filtered, and full event-datasets. These are described in greater detail in section 5.3. We explore the models proposed in section 4.6 on two disparate tasks, one of semi-supervised classification and one of clustering. For each task we evaluate the performance on each of the aforementioned datasets using appropriate metrics. The primary objective for the ^{46}Ar experiment was to identify resonant proton scattering events, and so the model s will be evaluated on their ability to separate proton events from the others that occur in the dataset. The broader picture for the application of these models are however applications to experiments where there might be multiple event-types of interest. And so we measure individual class performance wherever appropriate.

Intrinsic to the measurement of the semi-supervised performance is the budgeting of how many labeled samples one can feasibly extract. And the principal limitation of the semi-supervised approach is the assumption that the researchers are able to positively identify the event class(es) of interest. It is then interesting to quantify the change in model performance as a function of how many labeled 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 7.1 for the convolutional autoencoder and table 7.2 for the DRAW-analogues. The classifier is trained on a subset of the labeled set and evaluated on the remainder to estimate the OOS error. The best configuration will then be re-trained and we evaluate this model with k-fold cross validation as outlined in section 2.11.

105

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 3.2.3
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 7.1: 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
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:		
Dense dimension	Integer	Number of nodes in the dense layer connecting to the latent space
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

Table 7.2: 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. (2015) and in the earlier section 4.5.

add image from each experiment to results subsection header

Chapter 8

Classification results

The training procedure for classification using a semi-supervised regime as the one we'll apply necessitates the same strict separation of labeled data for the classification step as when considering ordinary classification tasks. Details on the modeling pipeline can be found in section **??**. All models excepting the baseline VGG model were tuned with the RandomSearch architecture which searches in a semi structured way over all the parameters given in table 7.1. The top 40 runs from each model is listed in appendix C.1 with their corresponding proton $f1$ score. We optimize the classification performance for each dataset listed in section 5.3. Additionally plots of $f1$ performance as a function of the number of labeled samples is included. As a benchmark we start by measuring the performance using just the pre-trained VGG16 representation of the labeled data of each dataset. The two proposed representation algorithms are then presented with results for each dataset for comparison.

correct appendix link and add performance tables?

8.1 VGG16

As outlined in section 4.6 the pre-trained VGG16 network will serve as the baseline comparison for this work. We chose VGG16 as it has a tried and true performance on labeled AT-TPC data from the ^{46}Ar experiment.(Kuchera et al. (2019)). For each labeled dataset listed in section 5.3 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 8.1

Additionally the scarceness of labeled data begs the question of how much labeled data is needed to achieve strong classification. To estimate this relationship we sample increasing subsets of the labeled data, each containing the previously selected data. For each selection a logistic regression model is fit and a $f1$ core is computed. This procedure is sensitive to which subset selected for fitting first

	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 $\pm 3.748 \times 10^{-2}$

Table 8.1: Logistic regression classification results using the VGG16 representation of the labeled data listed in section 5.3. The error is given as the standard deviation in the $f1$ score over the $K = 5$ folds of cross validation.

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 8.1

For comparison we also explore a visualization 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 (Van Der Maaten and Hinton (2008)). The visualization still has some uses in that a sense of the separation of classes in the latent space can be extracted. The principal component in the t-SNE projection is the perplexity of the model essentially controlling how many neighbors the algorithms considers, recommended values lie between $perp = 5$ and $perp = 50$ (Van Der Maaten and Hinton (2008)). We chose perplexity value of $perp = 15$ for all the visualizations. The latent space of the pre-trained VGG26 model is shown in figure 8.2 and demonstrates an evident separation of the proton class with the carbon and "other" classes.
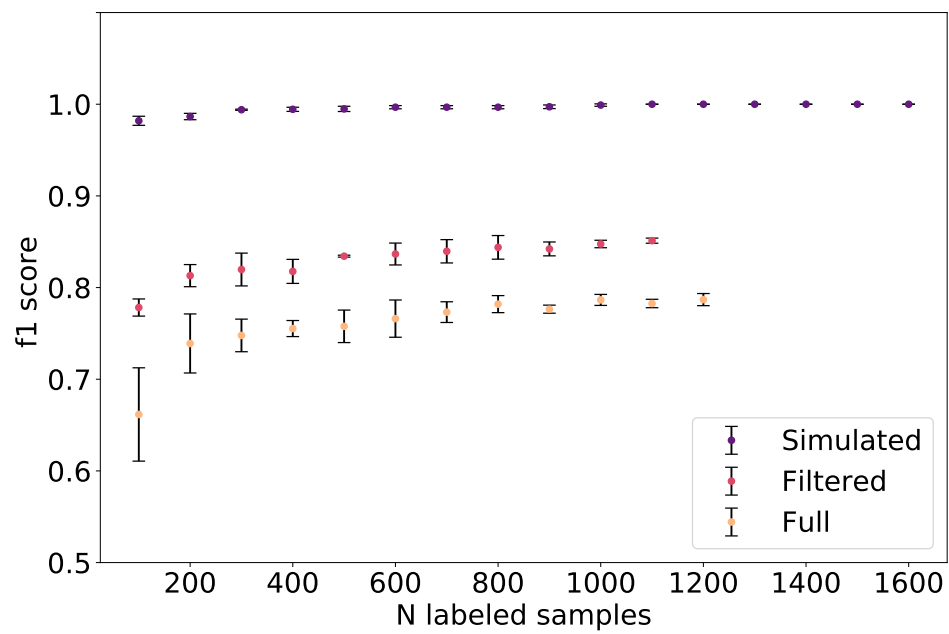
Figure 8.1: VGG16 performance on increasing subsets of labeled data. The error-bars represent the $\pm 1\sigma$ interval from the variability in the selection of subsets. The y axis $f1$ score is computed as the unweighted average of the sample classes.

Figure 8.2: 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.

8.2   Convolutional Autoencoder

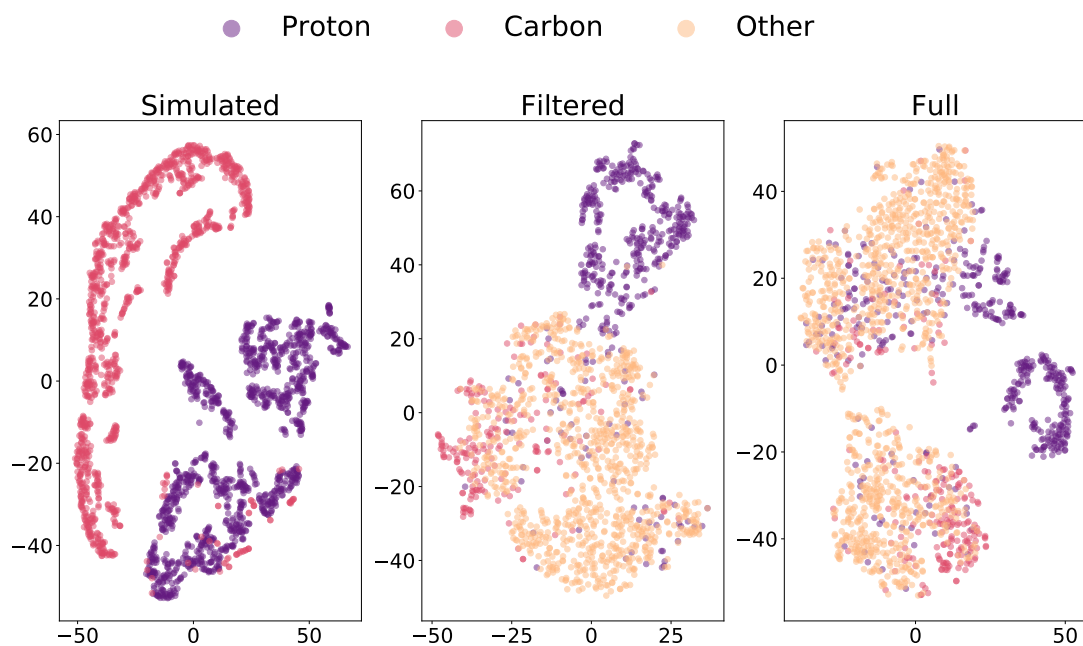To test the hypothesis that classification can be improved by using unsupervised methods to estimate the data distribution is investigated by using a convolutional autoencoder trained end-to-end on the data distribution, and then using the latent representation as input to a logistic regression classifier on the subset of data that has labels. This pipeline is outlined in section 4.6, and the data are described in section 5.3. 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.

(**Ar2**): Using the VGG16 network as the encoder, adjusting it's weights by the reconstruction loss with 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 similarities between the optimal architectures for each of the data-sets. 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.

add architecture tables, note on latent divergence?

For the best models found by random search we re-compute the performance 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 8.3

Furthermore we estimate the performance of the best models as a function of the number of labeled samples it sees. We select a random subsample from the labeled 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 8.3

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 PCA and subsequently project to two dimensions with a t-SNE mapping of the

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

Table 8.2: Hyperparameters that gives the strongest classifier performance on the three simulated, filtered and full datasets.

	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 8.3: 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.
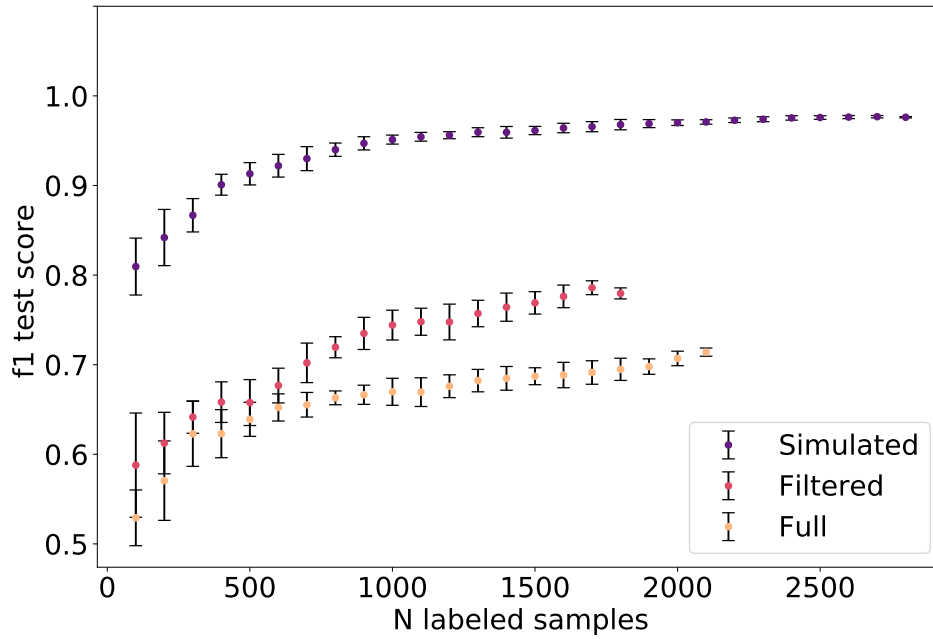

Figure 8.3: Latent space classification performance with a logistic regression classifier on a (Ar0) representation of each dataset. For each dataset a random subsample is drawn and iteratively added to in increments of $n = 100$ datapoints. To estimate the variance of this procedure we repeat the procedure $N = 10$ times.
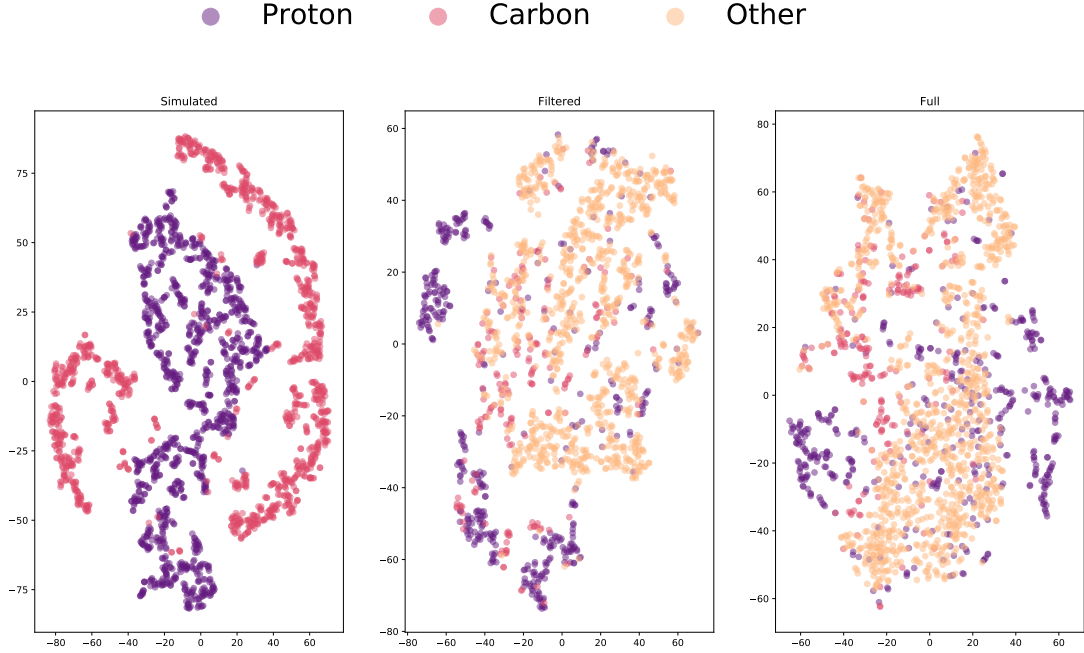
Figure 8.4: Visualizing the latent space of an (Ar0) trained autoencoder. The mapping is a t-SNE projection of the latent space to two dimensions. We re-iterate that the axes have non-informative units.

data. This visualization is shown in figure 8.6 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 8.4

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 the (Ar0) architecture. The results of this search is shown in figure 8.5

Lastly for the architecture we project the latent space for comparison with the non-tuned VGG16 representation.

Lastly we investigate the effect of adding a dueling 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

	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 8.4: 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.
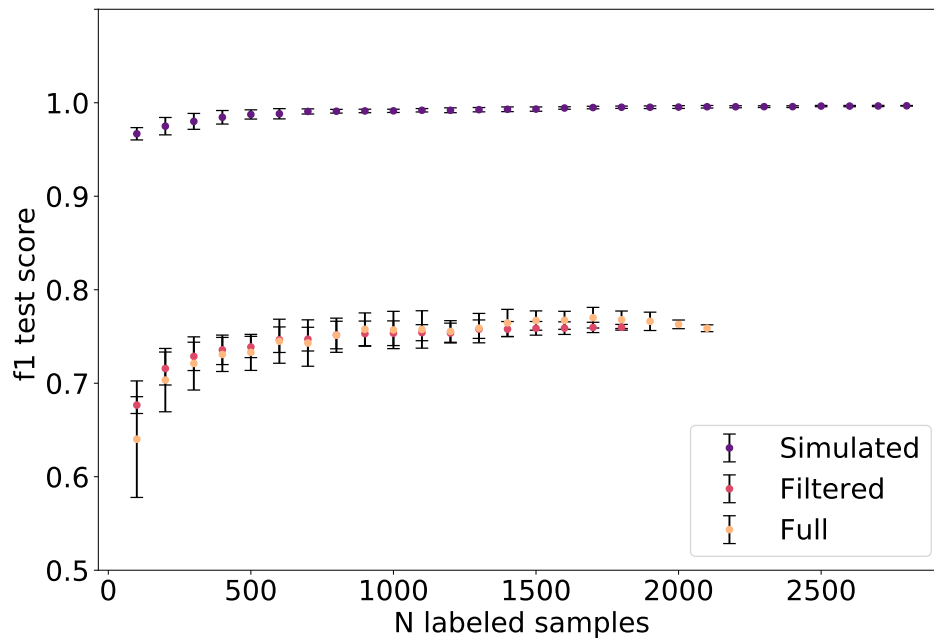

Figure 8.5: Latent space classification performance with a logistic regression classifier on a (Ar1) representation of each dataset. For each dataset a random subsample is drawn and iteratively added to in increments of $n = 100$ datapoints. To estimate the variance of this procedure we repeat the procedure $N = 10$ times.
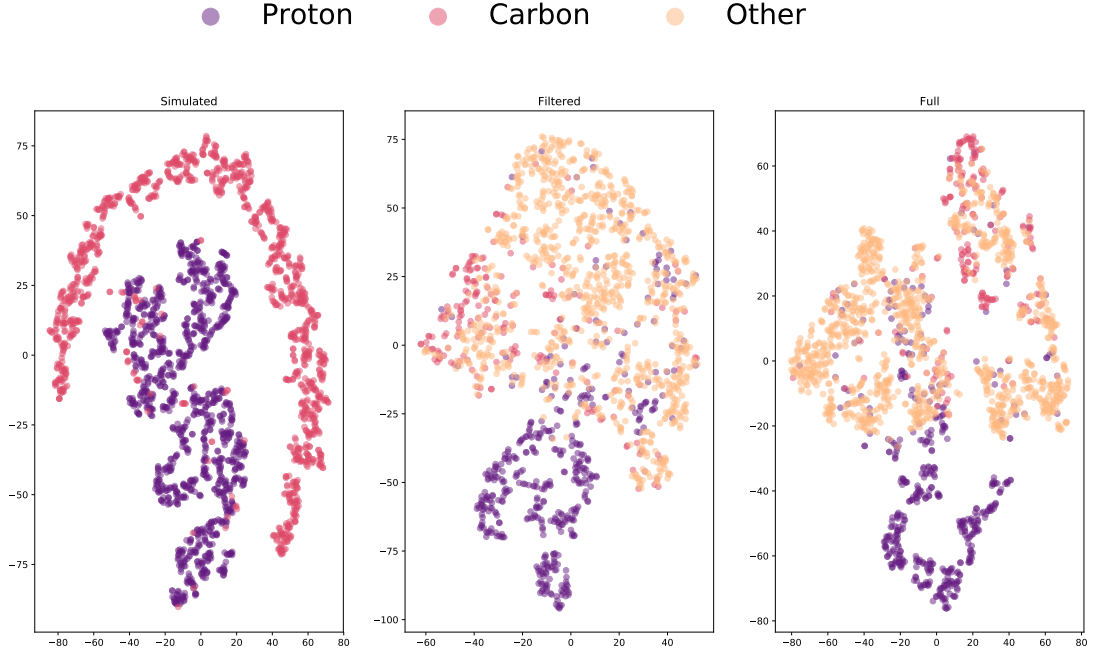
Figure 8.6: Visualizing the latent space of an (Ar1) trained autoencoder. The mapping is a t-SNE projection of the latent space to two dimensions.

charge deposited during the event. The results of those experiments are included in table 8.5. We immediately observe that the addition of the dueling decoder to the objective increases the performance of the linear classifier on the latent space.

	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 8.5: Logistic regression classification $f1$ scores using the (Ar0) architecture, with a dueling decoder addition to the objective. The standard error is reported from a $K = 5$ fold cross validation of the logistic regression classifier.

8.3　DRAW

Chapter 9

Clustering of AT-TPC events

The approach to the clustering of events is different to the semi supervised approach in a couple of meaningful ways. Firstly it's a harder task, as we'll demonstrate, which necessitates a more exploratory approach to the problem. Secondly 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 ARS (adjusted rand score) and NMI (normalized mutual information) defined in section 2.13. And as with the chapter on the semi-supervised results we start with considering the VGG16 pre-trained model as a benchmark.

9.1 VGG16

As previously discussed in section 4.6 we use the VGG16 pre-trained network as a baseline also 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 $\sim 5e3$ floats. Encumbered by the curse of dimensionality where the ratio of distances goes to one with increasing dimensionality (Aggarwal et al. (2001)) we investigate clustering based of a reduced dimensionality representation of the VGG16 processed event-data.

9.1.1 Simulated events

We begin by investigating the K-means clustering algorithm using the 10 primary principal components

Part IV

Discussion and Conclusion

Chapter 10

Discussion

In this chapter we will review the results presented in the previous section. The section is divided in topics of task, first we will consider the classification performance of our two implemented algorithms on the three different dataset; simulated, cleaned and full datasets. This performance will be contextualized by measuring against the results on similar tasks in the work of Kuchera et al. (2019).

10.1 Classification

Recall that the question we wish to explore in this thesis is whether training an autoencoder has benefits over models trained simply on labeled data. As a benchmark we trained a linear model on the data-representations from a pretrained VGG16 network. This high-performing model from the image analysis community has seen successful applications to the same experimental data, and so is a reasonable comparison for our methods (Kuchera et al. (2019)).

10.1.1 Convolutional autoencoder

Using the RandomSearch framework we were able to find a network configuration for the convolutional autoencoder that shows very strong performance on the simulated data. And strong performance on the filtered and full datasets. From the hyper-parameter search listed in appendix C and the best models found, and listed in table 8.2, we observe that there are no obvious relationships between most of the hyper-parameters and classification performance. The maximum-mean-discrepancy seems to trend with higher performance, as well as a fairly large first kernel. This result can be attributed to the performance landscape being very multi-modal with respect to the classification performance. Then there are many configurations that satisfy a linearly separable latent space. However there is also a large variation in the latent space indicating that there are regions of

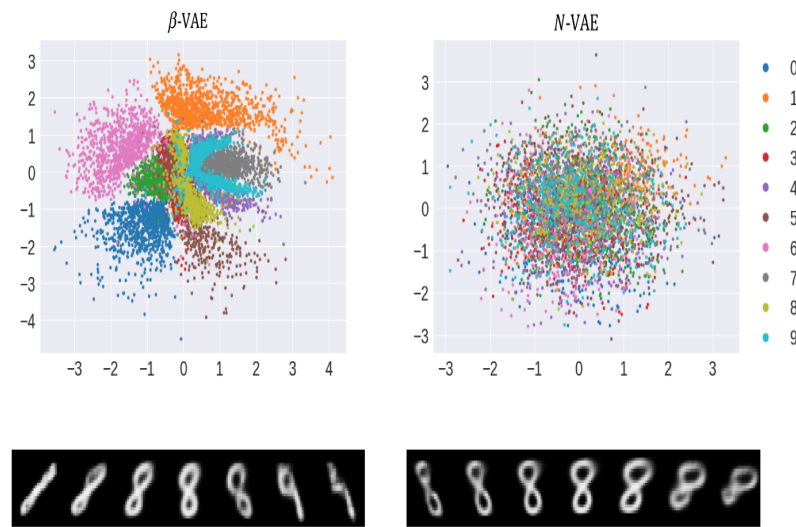add tables for searches with real and filtered

Figure 10.1: 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 favoring encoded class information in the latent space. The spaces between the class blobs makes for a poor generative algorithm, but for the purpose of classification or even clustering this is strongly preferable. On the right the natural clustering of feature information is demonstrated by the convincingly isotropic nature of the latent space distribution. The subplots under the latent distributions demonstrate reconstructions of a traversal along a latent axis, clearly showing the difference between feature and class information. Figure copied from Antorán and Vivolab (2019)

the hyper-parameter space that are unsuited to the purpose of making linearly separable latent spaces.

> Investigate with static random search - isolating some hyper-params

Latent space

Regarding the configuration of the latent space we note a preference for the maximum mean discrepancy term in terms of classifier performance. As Antorán and Vivolab (2019) shows 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, but strongly encourages feature information. Antorán and Vivolab (2019) describes feature information 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.

Indeed this objective works in favor of the variational autoencoder by tightening the distribution, i.e. achieving a density in the latent space without holes, that allows for the generation of new samples without areas in the latent space

that do not have a corresponding output.

An additional challenge attached to the latent space is the problem that the decoder might have the capacity to be trained as an autodecoder, i.e. reconstructing almost independently from the latent sample. This problem is investigated in detail by where they propose the dueling decoder structure. The dueling decoder adds a second reconstruction term to the objective. This second reconstruction is optimized over a different representation of the data. This might be reconstructing the edges in an image, it's 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. For the AT-TPC data this includes predicting the charge-intensity profile or the total charge deposited during the event.

Classifier performance

From table 8.3 it's clear that while the autoencoder architecture, (Ar0), is able to capture class information it does not outperform the pre-trained VGG16 in terms of the linear separability of its latent space. Regarding the performance in terms of the number of latent samples there seems to be no indication that performance was increased in that regard either. Visually inspecting the latent space in figure 8.6 we observe the same separation between proton and carbon events for simulated data. For the filtered and full datasets we observe a slight degradation of proton separation compared to the pure VGG16 representation, which we confirm by the proton $f1$ scores in table 8.3. 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 space is.

Using the VGG16 representation as an initial encoded representation, (Ar1) improved performance substantially from the (Ar0) architecture. The mean performance is close to the VGG16 performance, but the standard error is larger. Inspecting the performance as a function of n-labeled samples we observe that the (Ar1) autoencoder exhibits the same patterns of error with almost zero deviation from the mean for the filtered and simulated data, but with variations in the second decimal for the full data. The same behavior is seen in the pure VGG16 classifiers performance.

The obvious question is then why the reconstruction objective does not aid in classification. To find a plausible answer we look to a discussion on the PCA from Jolliffe (1982) where he remarks that the major axes of variation may not be the ones carrying class information. This relates to the reconstruction objective where we posit that the reconstruction focuses the optimization over these major axes of variation, and if they do not carry the salient class-separating information there is no reason to believe the latent space would. Adding to this argument is the observation that adding the dueling decoder improves the classifier. The rep-

resentations chosen for the auxiliary optimization task are salient to the physical processes underlying the event and so encourages the encoding of class-separating information.

improve this argument by classifier off the latent space of the VGG16 with pca and non pca factors

We note that in figure 8.3 and 8.5 the asymptotic performance is not expected to tend to the mean represented their corresponding tables as the test set is held constant. The K-means approach is then a better estimate of the true mean of the performance on the labeled set.

Appendices

Appendix A

Kullback-Leibler divergence of of Gaussian distributions

A multivariate Gaussian distribution in $\mathcal{R}^n$ is defined in terms of its probability density, which is a complete analogue to its univariate formulation,

$$p(x) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} exp(-\frac{1}{2}(x - \mu)^T \Sigma^{-1}(x - \mu)). \qquad \text{(A.1)}$$

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. $tr(X^T BX) = tr(BX^T X)$. Furthermore we use the fact that the trace is a linear operator and so commutes with the expectation i.e. $E(tr(BX^T X)) = tr(B\, E(X^T X))$. 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

$$\Sigma = \langle x^T x - 2x\mu^T + \mu\mu^T \rangle$$
$$\Sigma = \langle x^T x \rangle - \mu\mu^T.$$

Returning to the Kullback-Leibler divergence we then have

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

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

Neural network architectures

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					

Table B.1: Showing the details of the VGG network architectures. Network D trained on the ImageNet (Russakovsky et al. (2015)) dataset the network known as VGG16 and is what we use in this thesis.

Appendix C

Hyper-parameter search results

F1 score	L_x	L_z	N parameters	largest kernel	N layers	latent dimension	latent loss	reconstruction loss	activation function	batchnorm
0.99	0.000 32	1×10^5	4754	17	3	150	none	mse	relu	False
0.99	1.1×10^3	34	3798	13	6	50	mmd	bce	lrelu	False
0.98	0.000 63	0.000 28	3636	15	5	150	mmd	mse	lrelu	False
0.98	1.1×10^3	0.84	2408	11	3	3	mmd	bce	relu	True
0.97	1.7×10^2	1×10^5	760	7	5	150	none	bce	lrelu	True
0.97	0.0011	0.0038	1712	7	3	200	mmd	mse	relu	False
0.95	1.6×10^2	34	1740	11	5	50	mmd	bce	relu	False
0.94	0.000 76	0.000 75	464	7	5	100	mmd	mse	relu	False
0.94	0.0011	1×10^5	5194	15	4	50	none	mse	relu	True
0.92	1.9×10^2	1×10^5	9266	17	5	10	none	bce	relu	True
0.92	2.5×10^2	0.034	272	5	5	50	mmd	bce	relu	False
0.91	3.2×10^2	1.5×10^2	4770	11	5	200	kld	bce	relu	True
0.91	0.0003	0.01	688	5	4	150	mmd	mse	relu	True
0.9	1.5×10^2	1×10^5	1524	11	4	10	none	bce	lrelu	True
0.9	0.000 33	0.0013	7346	17	4	150	mmd	mse	relu	True
0.89	1.1×10^3	0.034	688	5	3	50	mmd	bce	lrelu	False
0.88	3.8×10^2	9.1×10^{-6}	3676	15	4	20	kld	bce	lrelu	False
0.83	0.000 79	0.0092	562	7	6	200	mmd	mse	lrelu	False
0.81	7.3×10^2	1×10^5	7316	11	6	20	kld	bce	relu	False
0.8	1.1×10^3	1×10^5	3546	17	5	50	none	bce	lrelu	False

Table C.1: Randomsearch runs for the convolutional autoencoder sorted by the resulting proton f1 score of the logistic regression classifier using the latent samples to classify event-types. We note the high occurrence of the maximum mean discrepancy with the higher performing classifications. We also note that simply no latent loss is able to achieve near perfect proton f1 scores.

Chapter 11

Notes

1. L1 regularization on the LSTM cells in the draw network seem to encourage the network to capture "many events". Looks like many spirals in one. While L2 (or sparese) regularization represents the images well. Can we represent the inner workings of the LSTM in some way?

2. Benchmark reconstruction loss for DRAW is at 255 - 1200 nodes, 60 filters, 10 timesteps, L2 regularization, Adam optimizer

3. Nesterov momentum yields suboptimal results. Reconstruction loss of about 1.4 times the loss when using Adam

4. Adadelta yields pure noise reconstructions (short simulation)

5. Adagrad yields localized "clouds" in the output

6. for simulated data it seems we can compress to about $350 \sim 300$ nodes in the encoder lstm. And to 3 dimensions in the latent space

7. In what seems like the minimal compressed state for the simulated data the training seems unstable and will frequently get stuck in local minima or have the gradient explode

8. DRAW without attention seems unable to learn even the simulated distribution at 128 by 128 pixels

9. In the DRAW algorithm the glimpse is specified by an affine weight transformation - but to be comparable it should be constant as a hyperparameter.

10. Implementing the glimpse as a hyperparameter was hugely successful, perhaps surprisingly in decreasing the reconstruction loss. Now remains the task of using the latent representations for classification

11. Two class-classification on the latent space was also hugely successful for simulated data

Bibliography

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., Zheng, X., and Research, G. (2015). TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. Technical report.

Aggarwal, C. C., Hinneburg, A., and Keim, D. A. (2001). On the Surprising Behavior of Distance Metrics in High Dimensional Space. pages 420–434.

Antorán, J. and Vivolab, A. M. (2019). DISENTANGLING IN VARIATIONAL AUTOENCODERS WITH NATURAL CLUSTERING.

Bergstra, J., Ca, J. B., and Ca, Y. B. (2012). Random Search for Hyper-Parameter Optimization Yoshua Bengio. Technical report.

Bradt, J., Bazin, D., Abu-Nimeh, F., Ahn, T., Ayyad, Y., Beceiro Novo, S., Carpenter, L., Cortesi, M., Kuchera, M., Lynch, W., Mittig, W., Rost, S., Watwood, N., and Yurkon, J. (2017). 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.

Bradt, J. W. (2017). *MEASUREMENT OF ISOBARIC ANALOGUE RESONANCES OF 47Ar WITH THE ACTIVE-TARGET TIME PROJECTION CHAMBER*. PhD thesis.

Burnham, K. P., Anderson, D. R., and Burnham, K. P. (2002). *Model selection and multimodel inference : a practical information-theoretic approach*. Springer.

Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning.

Fertig, E., Arbabi, A., and Alemi, A. A. (2018). beta-VAEs can retain label information even at high compression. Technical report.

Frankle, J. and Carbin, M. (2018). THE LOTTERY TICKET HYPOTHESIS: FINDING SPARSE, TRAINABLE NEURAL NETWORKS. Technical report.

Frankle, J., Dziugaite, K., Roy, D. M., and Carbin, M. (2019). Stabilizing the Lottery Ticket Hypothesis. Technical report.

Gregor, K., Com, D., Rezende, D. J., and Wierstra, D. (2015). DRAW: A Recurrent Neural Network For Image Generation. *Proceedings of Machine Learning Research*, 37.

Guo, X., Liu, X., Zhu, E., and Yin, J. (2017). Deep Clustering with Convolutional Autoencoders. In *neural information processing systems*, pages 373–382.

Harris, E., Niranjan, M., and Hare, J. (2019). A Biologically Inspired Visual Working Memory for Deep Networks.

Higgins, I., Matthey, L., Pal, A., Burgess, C., Glorot, X., Botvinick, M., Mohamed, S., Lerchner, A., and Deepmind, G. (2017). β-VAE: LEARNING BASIC VISUAL CONCEPTS WITH A CONSTRAINED VARIATIONAL FRAMEWORK. *ICLR proceedings*.

Hoerl, A. E. and Kennard, R. W. (1970). Ridge Regression: Biased Estimation for Nonorthogonal Problems. *Technometrics*, 12(1):55–67.

Hubert, L. and Arabic, P. (1985). Comparing Partitions. Technical report.

Hunter, J. D. (2007). Matplotlib: A 2D Graphics Environment. *Computing in Science & Engineering*, 9(3):90–95.

Ioffe, S. and Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training b y Reducing Internal Covariate Shift. In *International conference on machine learning*, page 11.

Jolliffe, I. T. (1982). A Note on the Use of Principal Components in Regression. Technical Report 3.

Karpathy, A. (2015). The Unreasonable Effectiveness of Recurrent Neural Networks.

Karpathy, A. (2019). CS231n Convolutional Neural Networks for Visual Recognition.

Keskar, N. S., Mudigere, D., Nocedal, J., Smelyanskiy, M., and Tang, P. T. P. (2016). On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima.

Kingma, D. P. and Lei Ba, J. (2015). ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION. In *ICLR proceedings*.

Kingma, D. P. and Welling, M. (2013). Auto-Encoding Variational Bayes.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *neural information processing systems*, pages 1097–1105.

Kuchera, M. P., Ramanujan, R., Taylor, J. Z., Strauss, R. R., Bazin, D., Bradt, J., and Chen, R. (2019). 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.

Kullback, S. and Leibler, R. A. (1951). On Information and Sufficiency. *The Annals of Mathematical Statistics*, 22(1):79–86.

Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

Lin, H. W., Tegmark, M., and Rolnick, D. (2017). Why Does Deep and Cheap Learning Work So Well? *Journal of Statistical Physics*, 168(6):1223–1247.

Linnainmaa, S. (1976). Taylor expansion of the accumulated rounding error. *BIT*, 16(2):146–160.

Marsland, S. (2009). Machine Learning: An Algorithmic Perspective.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133.

Mehta, P., Bukov, M., Wang, C. H., Day, A. G., Richardson, C., Fisher, C. K., and Schwab, D. J. (2019). A high-bias, low-variance introduction to Machine Learning for physicists. *Physics Reports*.

Pearlmutter, B. A. (1989). Learning State Space Trajectories in Recurrent Neural Networks. *Neural Computation*, 1(2):263–269.

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, É. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12(Oct):2825–2830.

Rosenblatt, F. (1958). THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN 1. Technical Report 6.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. Technical report, Insight Centre for Data Analytics.

Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S., Ma, S., Huang, Z., Karpathy, A., Khosla, A., Bernstein, M., Berg, A. C., and Fei-Fei, L. (2015). ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision*, 115(3):211–252.

Schölkopf, B., Smola, A., and Müller, K.-R. (1996). Component Analysis as a Kernel Eigenvalue Problem. (44):1299—-1319.

Seybold, B., Fertig, E., Alemi, A., and Fischer, I. (2019). Dueling Decoders: Regularizing Variational Autoencoder Latent Spaces.

Simonyan, K. and Zisserman, A. (2015). Very Deep Convolutional Networks for Large-Scale Image Recognition. In *International Conference on Learning Representations*.

Sonzogni, A. (2019). NuDat 2.7.

Srivastava, N., Hinton, G., Krizhevsky, A., and Salakhutdinov, R. (2014). Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Technical report.

Sutskever, I., Martens, J., Dahl, G., and Hinton, G. (2013). On the importance of initialization and momentum in deep learning. Technical report.

Szegedy, C., Liu, W., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2014). Going deeper with convolutions. Technical report.

Tibshirani, R. (1996). Regression Shrinkage and Selection via the Lasso. Technical Report 1.

Van Der Maaten, L. and Hinton, G. (2008). Visualizing Data using t-SNE. Technical report.

van der Walt, S., Colbert, S. C., and Varoquaux, G. (2011). The NumPy Array: A Structure for Efficient Numerical Computation. *Computing in Science & Engineering*, 13(2):22–30.

van Rossum, G. and Python development team, T. (2018). Python Tutorial Release 3.7.0 Guido van Rossum and the Python development team. Technical report.

Wishart, J. and Neyman, J. (1950). *Proceedings of the Berkeley Symposium on Mathematical Statistics and Probability*, volume 34. University of California Press.

Zhao, S., Song, J., and Ermon, S. (2017). InfoVAE: Information Maximizing Variational Autoencoders.