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

July 25, 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 using non-sequential and sequential neural network models

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 hyperplane 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

Machine Learning Theory

2.1 Introduction

The research question being explored in this thesis is to what degree we can extract compressed information about physical events from the AT-TPC experiment using modern machine learning methods. To achieve this we employ the DRAW algorithm (Gregor et al. (2015)) and variations of a traditional autoencoder. The DRAW algorithm is built of neural network components in a joint architecture comprised of a variational autoencoder wrapped in a set of long term short term memory cells. Each of the components are discussed in their own sections starting with the neural network in section 2.7 then followed by autoencoders in section 2.9 and finally recurrent neural networks in 2.8.

To arrive at the DRAW network we need to introduce the optimization of the log likelihood function using a binary cross-entropy cost function. In it's simplest form this optimization problem occurs in the formulation of the logistic regression mode introduced in section 2.3. As part of the derivation of the variational autoencoder cost the same optimization problem of the log likelihood will be applied. Likewise we introduce gradient descent methods and regularization, crucial components of modern machine learning, in the familiar framework of linear regression in section 2.4.

We hypothesize that this compressed information can be used to linearly separate events in classes, possibly using relatively small amounts of data. Furthermore we hypothesize that we can construct an implicit clustering based on emergent structures in the latent space. In the experiment at hand we hope to separate events with proton or a carbon as the reaction output.

2.2 Model Fitting

The process of 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 the theory at hand. In physics we construct models to describe complex natural phenomena by which we can make predictions or infer inherent properties of the system from. The models we use vary from simple linear models describing the nearest neighbor ising model in statistical mechanics to variational markov chain monte-carlo simulations for many-body quantum mechanics. Common for all these applications is the need to fit the model to the data at hand. The model would describe something general about systems similar the one under scrutiny and the fitting procedure is the way by which the model is tailored to the system at hand.

In this thesis we consider a special case of model fitting commonly known as function approximation. Wherein an unknown function $\hat{f}(\mathbf{X}) = \hat{y}$ is approximated by an instance of a model $f_\theta(\mathbf{X}) = y$. We generally don't have a good ansatz for the form of $\hat{f}$. The subscript θ denotes the model parameters we can adjust 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. 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. 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 such 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 overfitting and underfitting to the data this chapter, but first we introduce some simple tools and terminology from statistical learning theory and information theory that is used throughout this thesis.

2.2.1 On information

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 re-state 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

$$I = -\log(p(x)) \tag{2.1}$$

We usually wish to have knowledge of a system, however, obtained by the expectation over information. This expectation is called the entropy of the system and is defined in a familiar way as

$$H(p(x)) = -\langle I(x)\rangle_{p(x)} = -\sum_x p(x)\log(p(x)) \tag{2.2}$$

Depending on the choice of base of the logarithm this functional has different names. Perhaps widest used is log base 2 know as the Shannon entropy; describing how many bits of information we need to fully describe the process underlying $p(x)$. 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 x $p(x)$ with the model distribution $q_\theta(x)$. In this thesis we will in general use greek subscripted letters on distributions to denote models. This measure is called the cross-entropy and is defined as

$$H(p, q) = -\sum_x p(x)\log(q_\theta(x)) \tag{2.3}$$

Tying the cross entropy to model optimization requires a quantity to optimize. We define the maximum likelihood estimate (MLE) which represents the probability of seeing the data, i.e. the set of tuples $\eta_i = \{\mathbf{x}_i, y_i\}$, at hand given our model and parameters. Given the feature vectors with binary class labels $S = \{\eta_i\}$ the likelihood of our model is defined as

$$p(S|\theta) = \prod_i q_\theta(x_i)^{y_i} - (1 - q_\theta(x_i))^{1-y_i} \tag{2.4}$$

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 as the number of terms increase, to circumvent this we take the logarithm of the likelihood defining the log-likelihood. Optimizing the log-likelihood yields the same optimum as for the likelihood as the logarithmic function is monotonic. [1]

$$\mathcal{L}(\mathbf{x}, y, \theta) = \log(p(S|\theta)) = \sum_i y_i \log(q_\theta(x_i)) + (1 - y_i)(q_\theta(x_i)) \tag{2.5}$$

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

$$\theta^* = \arg\max_\theta \mathcal{L}(\mathbf{x}, y, \theta) \tag{2.6}$$

This formulation of the MLE for binary classification can be extended to the case of simple regression where one shows the mean squared error is the functional to optimize for. Common to most applications in machine learning is the solution of these optimization problems by the use of gradient descent on the cost, usually

[1] 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.

simply defined as the negative loss. Gradient descent is discussed in some detail in section 2.6.

2.2.2 Over and under-fitting

When fitting an unknown function to data it is often not clear what complexity is suitable for the model. Additionally compounding this problem is the ever present threat of various noise signals and measurement errors present in the data. Further complicating the issue is the nature of machine learning problems: we're almost always interested in extrapolating to unseen regions of data, in the machine learning vernacular these are sets of data we call TEST-sets. Data used to fit the model is called TRAIN-sets. To illustrate this we'll consider the case of the one dimensional problem of polynomial regression. 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. The concepts of over and under-fitting go hand-in-hand with two other concepts we'll introduce in this chapter, regularization and the bias-variance relationship. Firstly however we'll briefly introduce the concepts in over and under-fitting models. This topic is strongly related to the concepts of complexity, and the Vapnik-Chervonenkis theory of statistical learning, the details of which are outside the scope of this thesis. We start by considering a process we wish to model that is on the form shown in equation

$$y_i = P(x_i) + \epsilon_i \tag{2.7}$$

Our goal is to create a model $f(x_i;\theta)$ of parameters θ that best approximates our true distribution $p(y_i|x_i)$. To evaluate the quality of our model we use the formalism introduced in the previous sub-section 2.2.1. Which is to say we need to introduce a cost-function whose minimum w.r.t the parameters yields the optimal parameters θ^*, i.e.

$$\theta^* = \arg\min_{\theta} \mathcal{C}(y_i, f(x_i;\theta)) \tag{2.8}$$

Typically $C(\cdot,\cdot)$ is something like a squared distance, or a cross entropy like we introduced in section 2.2.1. If the measurement errors ϵ_i from equation 2.7 are independent identically distributed Gaussian variables then the method of least squares and the squared distance cost are appropriate. With those assumptions we form the basis for linear regression, a foundational model we elaborate on in section 2.4. For probability-like outcomes the cross entropy is a more common choice, represented by the other cornerstone of machine learning; logistic regression. We detail this method also in a later section 2.3.

In equation 2.7 the ϵ_i term expresses a noise term at that point, and the function $P(\cdot)$ is the true process which we are interested in modeling but whose shape is hidden from us. It is important to note that for data with no noise, most of the problems and cautions we describe in this chapter do not apply, however measuring any physical phenomenon inherently carries with it some noise. We wish to model this principally unknown process expressed by y_i by using a polynomial of degree n, let P^n be the set of polynomials which we can construct a polynomial of degree n to fit to the observation.

The distinguishing features of overfitting are shown in figure 2.1 where we fit polynomials of varying degrees to data drawn from a true distribution following a first and third order polynomial respectively. In the figure we observe the higher order models are being fit to spurious trends in the data that we can attribute to the noise. The higher expressibility of the model then leads to it capturing features of the noise that increase accuracy in the training domain but we observe that this rapidly deteriorates in the testing region. That the model follows these noise-generated features is called overfitting. Conversely when we increase the complexity of the data to be drawn from P^3 the linear model looses the ability to capture the complexities of the data and is said to be underfit.

To quantify the quality of the model during optimization we measure the change in the value of the cost function. In the machine learning community the best practice for this in settings were we train on tuples of response variables and data, e.g. $s_i = \{y_i, \mathbf{x}_i\}$ is to split the data in disjoint sets. From the full data one selects a subset of around $\sim 20\%$ or so that is withheld from training. After training we can evaluate the cost function on this data to create an unbiased estimate the out-of-sample error

$$E_{out} = C(y_{test}, f(x_{test}; \theta^*)) \tag{2.9}$$

During training we split the data yet again in two disjoint subsets, the larger of which the model is trained on. The training data gives us another measure of how good the model is, the in-sample-error, or E_{in}. Lastly the data that is not seen during that iteration but can be randomly selected from the train data we call validation and is our measure of when we should stop the optimization. The training error will likely decrease but for complex models this validation error E_{val} will diverge and signal that the optimization should be terminated. We investigate the relationships between each of these errors and the model complexity in section 2.4.3. As to the exact size of the different partitions is largely a heuristic decision made by the amount of data available. It is also important to note that some models operate without a ground-truth labeling, or target, y_i. The models investigated in this thesis are either completely divorced from the ground truth variables during training or only use them in an auxiliary step. We show that this separation allows well trained models to estimate the ground truth using surprisingly few samples.

We've conspicuously left out the fitting procedure in the paragraphs above. Generally the degree of over or under-fitting is not dependent on the fitting

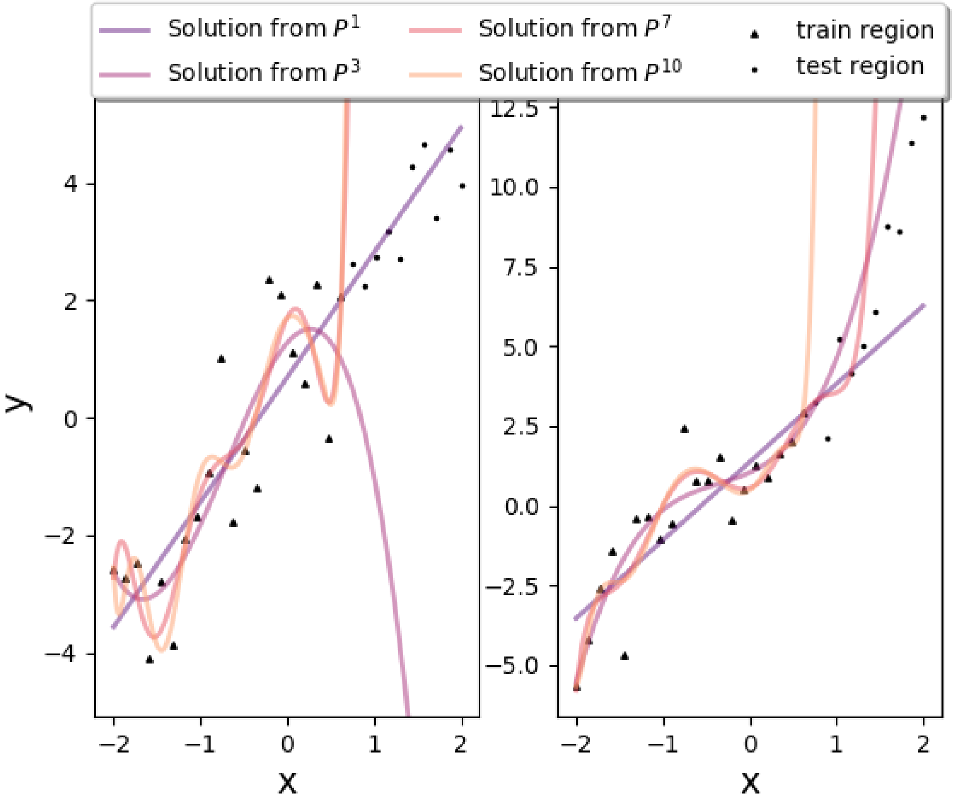


Figure 2.1: Polynomial regression of varying degrees on data drawn from a linear distribution on the left and a cubic distribution on the right. Models of varying complexity indicated by their basis P^n are fit to the train data and evaluated on the test region. We observe that the higher order solutions follow what we observe to be spurious-noise generated features in the data. This is what we call overfitting. On the right hand side we observe that the model with appropriate complexity, $f(x_i) \in P^3$, follows the true trend also in the test region while the linear and higher order models miss. The linear model does not have the capability to express the complexities of the data and is said to be underfit. Additionally we observe that the solutions of both sides degrade rapidly inside the testing region. Extending our models to previously unseen regions is very challenging.

schema, but more on the model which we use to make predictions. 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 networks can be expressed by sub-networks contained in the modern very deep very complex neural networks. As a consequence we primarily concern ourselves then with understanding and proposing remedies to overfitting.

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.

2.3 Logistic Regression

2.4 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. In this section we'll detail the derivation of linear regression in the formalism of a maximum likelihood estimate, as it is in this formalism the thesis writ large is framed. Linear regression can be expressed on a general form as the linear relationship expressed in equation 2.10 where we don't specify the basis of $\mathbf{w}$, but we are free to model using polynomial, sinusoidal or ordinary Cartesian basis-sets.

$$\hat{y}_i = \mathbf{x}_i^T \mathbf{w} + b \tag{2.10}$$

In addition to equation 2.10 we introduce the error term $\epsilon_i = y_i - \hat{y}_i$ which is the difference between the models prediction, $\hat{y}_i$, and the actual value y_i. The goal of linear regression is to minimize this error, in mathematical terms we have an optimization problem on the form

$$\mathcal{O} = \underset{\mathbf{w}, b}{\arg\min} \, ||\mathbf{y} - \mathbf{X}\mathbf{w}||^2 \tag{2.11}$$

In physics there are many relationships which have successfully been modeled with linear regression. From the very simple single regression problem of Ohm's law to more complex problems like identifying the Hamiltonian in a thermodynamic system like the Ising model. The latter of the two is elegantly demonstrated in Mehta et al. (2019). The central assumption of linear regression, that provides the opportunity for a closed form solution, is the independent identically distributed (IID) nature of ϵ_i. We assume that the error is normally distributed with zero-mean and identical variance across all samples, e.g.

$$\epsilon_i \sim \mathcal{N}(0, \sigma^2) \tag{2.12}$$

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

$$\hat{y}_i \sim \mathcal{N}(\mathbf{x}_i^T \mathbf{w}, 0) \tag{2.13}$$

For simplicity we include the intercept term, b, in $\mathbf{w}$ and extend the full datamatrix $\mathbf{x}$ with a column of ones to compensate. Where we use $\mathcal{N}(\mu, \sigma^2)$ to denote a Gaussian normal distribution which has a probability density function (PDF) defined as

$$p(x|\mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{\sigma^2}} \tag{2.14}$$

This allows us to consider the real outcomes y_i as a set of normally distributed variables too. By the linearity of the expectation operator we have

$$\langle y \rangle = \langle \hat{y} + \epsilon \rangle \tag{2.15}$$

$$\langle y \rangle = \langle \hat{y} \rangle + \langle \epsilon \rangle \tag{2.16}$$

$$\langle y \rangle = \mathbf{x}_i^T \mathbf{w} \tag{2.17}$$

And by the exact same properties we have that the variance of the prediction is the variance of the error term

$$\langle y \rangle^2 + \langle y^2 \rangle = \sigma^2 \tag{2.18}$$

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 2.4

$$
\begin{aligned}
p(S|\theta) &= \prod_i^n p(y_i) && (2.19)\\
&= \prod_i^n \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(y_i - \mathbf{x}^T\mathbf{w})^2}{\sigma^2}} && (2.20)\\
&= \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^n \prod_i^n e^{-\frac{(y_i - \mathbf{x}^T\mathbf{w})^2}{\sigma^2}} && (2.21)\\
&= \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^n e^{-\sum_i \frac{(y_i - \mathbf{x}_i^T\mathbf{w})^2}{\sigma^2}} && (2.22)\\
&= \left(\frac{1}{\sqrt{2\pi\sigma^2}}\right)^n e^{-\frac{(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})}{\sigma^2}} && (2.23)
\end{aligned}
$$

We recall from section 2.2 that the best parameters of a model can be estimated with

$$\theta^* = \arg\max_\theta p(S|\theta) \tag{2.24}$$

To find the optimal values we then want to take the derivative w.r.t the parameters and find a saddle point, 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 2.2 re-familiarizing ourselves with the log-likelihood

$$\log(p(S|\theta)) = n\log(\frac{1}{\sqrt{2\pi\sigma^2}}) - \frac{(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})}{\sigma^2} \tag{2.25}$$

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

$$
\begin{aligned}
\nabla_\mathbf{w} \log(p(S|\theta)) &= \nabla_\mathbf{w}\left(-\frac{1}{\sigma^2}(\mathbf{y} - \mathbf{X}\mathbf{w})^T(\mathbf{y} - \mathbf{X}\mathbf{w})\right) \\
&= -\frac{1}{\sigma^2}(-2\mathbf{X}^T y + 2\mathbf{X}^T\mathbf{X}\mathbf{w}) \\
&= -\frac{1}{\sigma^2}2\mathbf{X}^T(\mathbf{y} - \mathbf{X}\mathbf{w}) \\
0 &= -\frac{2}{\sigma^2}(\mathbf{X}^T\mathbf{y} - \mathbf{X}^T\mathbf{X}\mathbf{w}) && \text{setting derivative to zero} \\
\mathbf{X}^T\mathbf{X}\mathbf{w} &= \mathbf{X}^T\mathbf{y} && \text{multiplying away constants}
\end{aligned}
$$

Which ultimately supplies us with the solution, equal to the least squares derivation, of the optimal parameters. Which we present in equation 2.26

$$\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y} \tag{2.26}$$

Which is the same solution for the parameters as we get with ordinary least squares. 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 derivative might not be well defined. It's also important to note that the

add some plots of linear data with noise, regression line and show the errors?

2.4.1 Regularization

With the advent of modern computing resources researchers gained the ability to operate very complex models. Giving rise to the problem of overfitting to noise in the data, as described in section 2.2.2. Finding measures to reduce overfitting has been a goal for near on 50 years. The first modern breakthrough was adding a coefficient restriction on their cumulative magnitude by the L_2-norm. 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) this modification on 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). The inclusion of a penalty by the L_1-norm proved challenging as it was not solvable analytically and required iterative methods like gradient descent, described in detail in section 2.6.

In this section we will introduce the different regularizations as additional contributions to the cost function. As well as geometrically exploring the reasoning for why these terms reduce the expressibility of the model. The L_p norm is defined as.

$$L_p(\mathbf{x}) = \left(\sum |x_i|^p\right)^{\frac{1}{p}} \tag{2.27}$$

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. The squared L_2 norm was added to form ridge regression (Hoerl and Kennard (1970)). For Lasso (Tibshirani (1996)) the authors used an L_1 term, commonly called the Manhattan or taxicab-distance. The Manhattan distance is 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 the ridge regularization term to the squared error cost, shown in equation 2.28

$$C(y_i, f(\mathbf{x}_i;\theta)) = (y_i - f(\mathbf{x}_i;\theta))^2 + \sum |\theta_i|^2 \tag{2.28}$$

Conceptually the regularization term added to the cost function modifies what parameters satisfy the arg min in equation 2.8 by adding a penalty to parameters having high values. This is especially useful in cases where features are co-variate or the data is noisy. Adding a regularization term is equivalent to solving a constrained optimization problem e.g.

$$\theta^* = \underset{||\theta||_2^2 < t}{\arg\min} ||y_i - f(\mathbf{x}_i;\theta)||_2^2 \tag{2.29}$$

The representation as a constrained optimization is useful to understand the impact of this form of regularization. Graphically this is demonstrated in figure 2.2, 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 the squared error, or cross entropy. 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.

add subsection on dropout and batchnorm

2.4.2 Batch Normalization

maybe dropout?

2.4.3 The bias-variance relationship

In statistical learning theory we

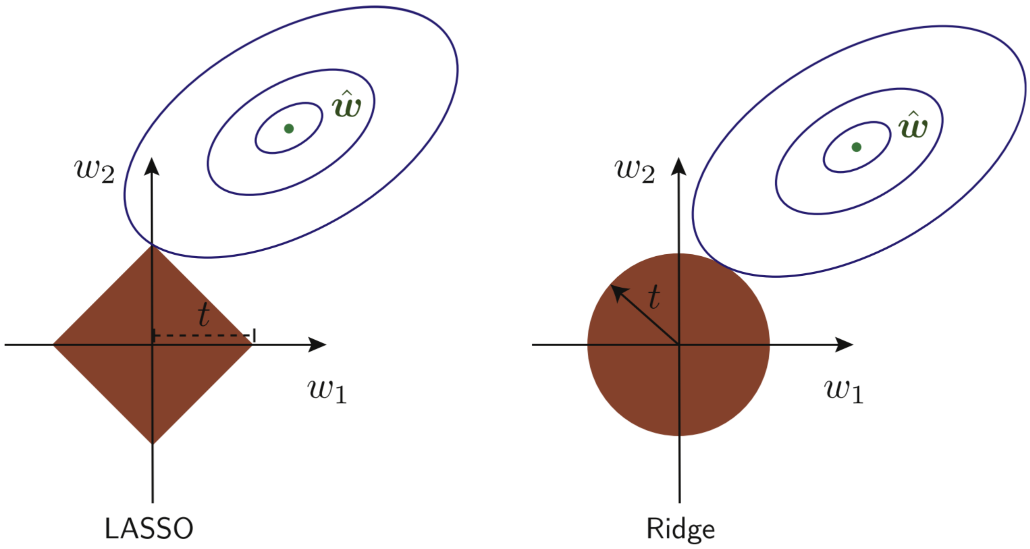


Figure 2.2: Demonstrating the effect of a regularization constraint on a 2-variable optimization. The blue ovals represent the squared error, as it is quadratic in the parameters w_i. And the shaded brown region represents the restriction on the values of w_i, s.t. the only eligible values for the parameters 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)

2.5 Hyperparameters

In fitting a model to data we adjust the parameters to fit some objective. However there are several parameters to 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 are called hyperparameters and are 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 and the optimizer parameters like the learning rate η and eventual momentum parameters $\beta_1, \beta_2, \dots$ 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.

2.5.1 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.

2.5.2 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 $\lambda$, $\eta$
Result: log of performance for each training
Initialization ;
log ← [ ];
for $\lambda_k \in \lambda$ do
    for $\eta_k \in \eta$ do
        opt ← optimizer($\eta_k$);
        model_instance ← model($\lambda_k$);
        metrics ← model_instance.fit($\mathbf{X}, \hat{\mathbf{y}}$, opt) ;
        log.append((metrics, ($\lambda_k$, $\eta_k$)))
    end
end
```

2.5.3 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 2.3. 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.

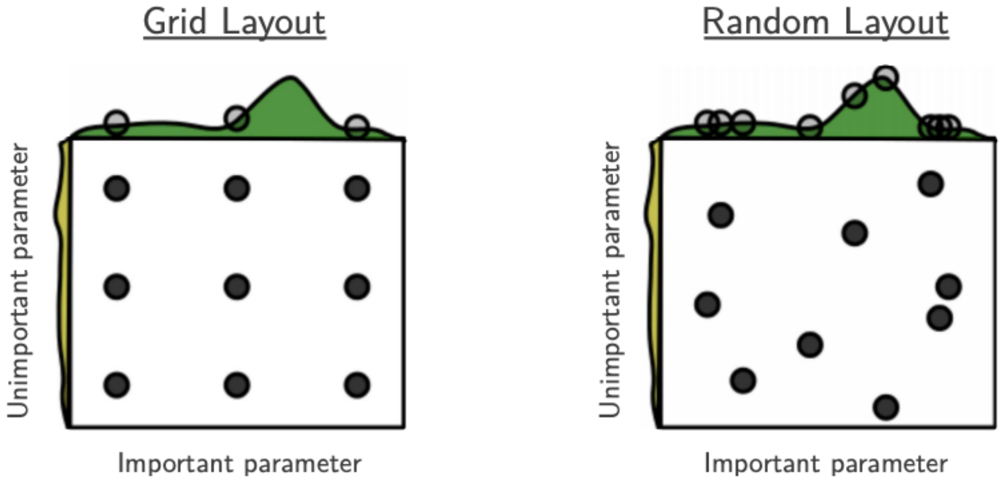


Figure 2.3: Figure showing the inefficiency of grid search. The shaded areas on the axis represent the unknown variation of the cost as a function of that axis-parameter. Since the optimum of the loss with respect to a hyperparameter might be very narrowly peaked a grid search might miss the optimum 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)

2.6 Gradient Descent

The process of finding minima or maxima, known collectively as extrema, of a function well trodden ground for physicists. 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 extrema 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. Mehta et al. (2019) shows that even relatively simple simple model like logistic regression with L_1 regularization is transcendental in the first derivative of the cost. For both too-complex or analytically unsolvable first derivatives we then turn to iterative methods for the gradient. Iterative methods of the n-th order for finding extrema involves updating parameters based on the direction of the set of n-th order partial derivatives. 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 of many variables as shown in equation 2.39.

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla f(\mathbf{x}_n) \tag{2.30}$$

Equation 2.39 is the hammer which regards any neural network as a nail. 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. 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 also. The choice of eta 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 2.4 shows the effects of choosing the values for the learning rate poorly, while figure 2.5 shows the effect of a well chosen eta which finds the minimum in just a few steps.

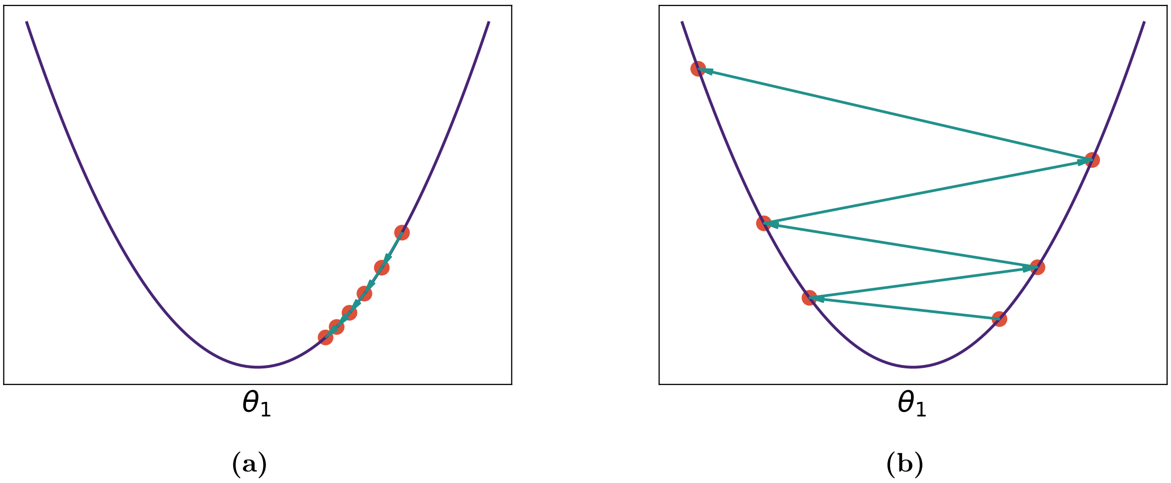


Figure 2.4: Gradient descent on a simple quadratic function showing the effect of too small, **(a)**, and too large, **(b)**, value for the learning rate η

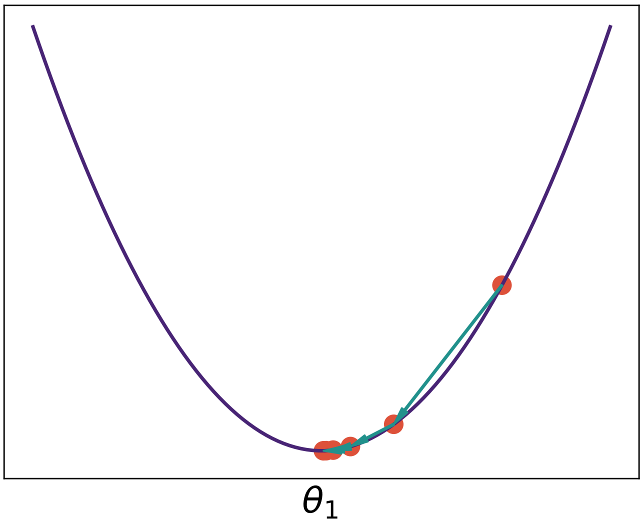


Figure 2.5: Complement to figure 2.4 where we show the effect of a good learning rate on a gradient descent procedure. The gradient descent procedure is performed on a quadratic function.

While in the case of the convex function we can directly inspect the progress this is not feasible for the high-dimensional updates required for a neural network. In the Stanford course authored by Karpathy (2019) they point out that one can indirectly observe the impact of the choice of learning rate from the shape of the loss as a function of epoch. This impact is shown in figure 2.6 which we'll use as a reference when training the models used in this thesis.

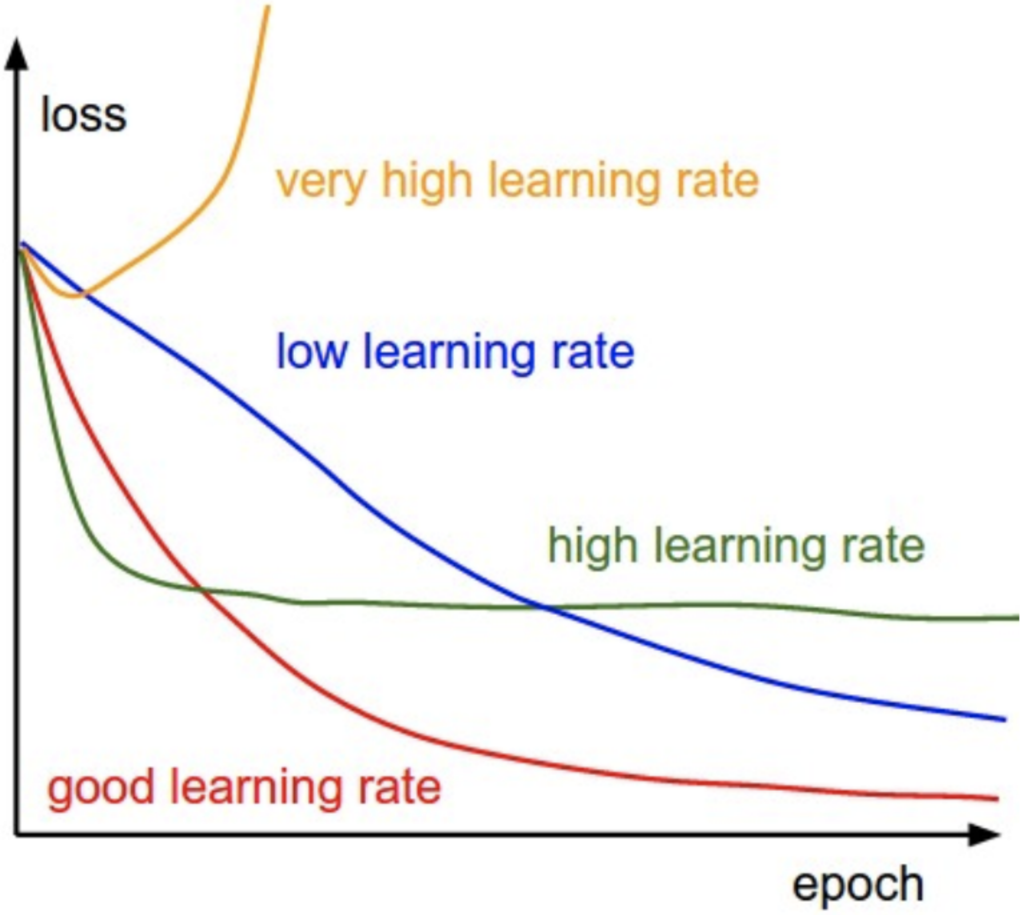


Figure 2.6: A hand-drawn figure showing the impact of the choice of the learning rate parameter on the shape of the loss function. The optimal choice has a nice slowly decaying shape that we will use as a benchmark when tuning the learning rate in our applications. Copied from the cs231 course material from Stanford authored by Karpathy (2019)

Using a first order method, while much more computationally efficient than higher order methods, bring with them some problems of their own. In particular there are two problems that need to be solved and we'll go through a couple of methods 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)

2.6.1 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 in equation 2.31

$$\begin{aligned} \mathbf{v}_n &= \beta \mathbf{v}_{n-1} + (1-\beta)\nabla f(\mathbf{x}_n) \\ \mathbf{x}_{n+1} &= \mathbf{x}_n - \eta \mathbf{v}_n \end{aligned} \tag{2.31}$$

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

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

So each $\mathbf{v}_t$ is then an exponentially weighted average over all the previous gradients. The factor $1-\beta$ then controls how much of a view there is backwards in the iteration. The factor is then reasonably restricted to avoid overpowering by recent gradients to $\beta \in [0, 1]$. How many steps in the past sequence that this average "sees" we illustrate in figure 2.7. 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

2.6.2 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

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 2.5

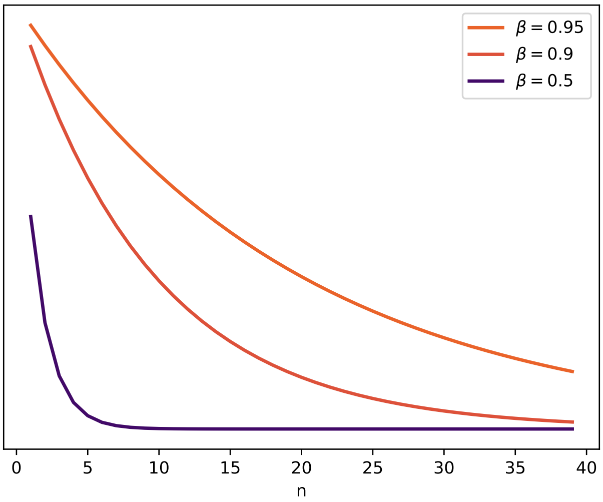
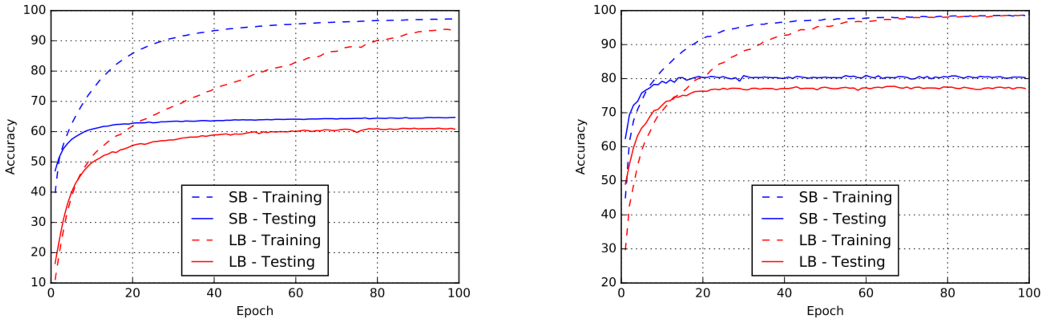


Figure 2.7: A figure illustrating the decay rate of different choices of β. The lines go as β^n and shows that one can quite easily infer how many last steps is included for each choice. A good starting value for the parameter has been empirically found to be $\beta = 0.9$ for many applications. In this thesis we'll use a gaussian distribution around this value as a basis for a random search.

behavior is batching which involves taking the gradient with only a limited partition of the data and updating the parameters. This creates noise in the gradient which encourages exploration of the loss-surface rather than strong convergence to the nearest minimum. If we set the batch size $N = 1$ 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 2.8.



(a) Network F_2

(b) Network C_1

Figure 2.8: 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)

2.6.3 adam

One of the major breakthroughs in modern machine learning is the improvements on the optimization scheme of gradient descent from the most simple version introduced in equation 2.39 to the adam paradigm described 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

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 2.5

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla\mathcal{L}(\mathbf{x}_{t,i}) \tag{2.32}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla\mathcal{L}(\mathbf{x}_{t,i})^2 \tag{2.33}$$

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

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.34}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.35}$$

Which is then used to update the model parameters in a familiar way, equation 2.36 gives the adam update rule.

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{\eta}{\hat{v}_n + \epsilon}\hat{m}_n \tag{2.36}$$

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 $\beta_2 > \beta_1$. Lastly then we consider the hyperparameters required for the usage of adam, β_1 and β_2 are in principle both needed to be tuned but we restrict the tuning to β_1 in this thesis to limit the number of parameters needed for tuning. We list the parameters and their scale in tble 2.2.

2.7 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. The output can be categorical or real-valued in nature. A simple illustration of two neurons in one layer is provided in figure 2.9

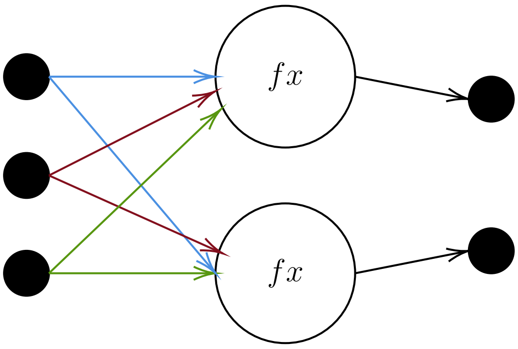


Figure 2.9: An illustration of the graph constructed by two artificial neurons with three input nodes. Colored lines illustrate that each of the input nodes are connected to each of the neurons in a manner we denote as fully-connected.

The ANN produces an output by a "forward pass". If we let the input to an ANN be $x \in \mathbb{R}^N$, and letting the matrix $W \in \mathbb{R}^{N \times D}$ be a representation of the weight matrix forming the connections between the input and the artificial neurons. Lastly we define the activation function $a(x)$ as a monotonic, once differentiable, function on $\mathbb{R}^1$. The function $a(x)$ determines the complexity of the neural network together with the number of neurons per layer and number of layers. For any complex task the activation takes a non-linear form which allows for the representation of more complex problems. A layer in a network implements what we will call a forward pass as defined in function 2.37.

$$\hat{y} = a(\langle x|W\rangle)_D \tag{2.37}$$

In equation 2.37 the subscript denotes that the function is applied element-wise and we denote the matrix inner product in bra-ket notation with $\langle\cdot|\cdot\rangle$. Each node is additionally associated with a bias node ensuring that even zeroed-neurons can encode information. Let the bias for the layer be given as $b \in \mathbb{R}^D$ in keeping with the notation above. Equation 2.37 then becomes:

$$\hat{y} = a(\langle x|W\rangle)_D + b \tag{2.38}$$

As a tie to more traditional methods we note that if we only have one layer and a linear activation $a(x) = x$ the ANN becomes the formulation for a linear regression model. In our model the variables that need to be fit are the elements of W that we denote W_{ij}. While one ordinarily solves optimization problem for the linear regression model by matrix inversion, we re-frame the problem in more general terms here to prime the discussion of the optimization of multiple layers and a non linear activation function. The objective of the ANN is formulated in a "loss function", which encodes the difference between the intended and achieved output. The loss will be denoted as $\mathcal{L}(y, \hat{y}, W)$. Based on whether the output is described by real values, or a set of probabilities this function, $\mathcal{L}$, takes on the familiar form of the Mean Squared Error or in the event that we want to estimate the likelihood of the output under the data; the binary cross-entropy. We will also explore these functions in some detail later. The ansatz for our optimization procedure is given in the well known form of a gradient descent procedure in equation 2.39

$$W_{ij} \leftarrow -\eta \frac{\partial \mathcal{L}}{\partial W_{ij}} + W_{ij} \tag{2.39}$$

2.7.1 Backpropagation

In the vernacular of the machine learning literature the aim of the optimization procedure is to "train" the model to perform better on the regression, reconstruction or classification task at hand. Training the model requires the computation of the total derivative in equation 2.39. This is also where the biological metaphor breaks down, as the brain is almost certainly not employing an algorithm so crude as to be formulated by 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 w.r.t the output of the ANN to the gradient w.r.t the individual neuron weights in the layers of the ANN. The algorithm begins with computing the total loss, here exemplified with the squared error function, in equation 2.40

$$E = \mathcal{L}(y, \hat{y}, W) = \frac{1}{2} \sum_n \sum_j (y_{nj} - \hat{y}_{nj})^2 \tag{2.40}$$

The factor one half is included for practical reasons to cancel the exponent under differentiation. As the gradient is multiplied by an arbitrary learning rate η this is ineffectual on the training itself. The sums define an iteration over the number of samples, and number of output dimensions respectively. Taking the derivative of 2.40 w.r.t the output, $\hat{y}$, we get

$$\frac{\partial E}{\partial \hat{y}_j} = \hat{y}_j^M - y_j \tag{2.41}$$

Recall now that for an ANN with M layers the output fed to the activation function is

$$x_j^M = \langle a^{M-1} | W^M \rangle + b_j \tag{2.42}$$

Where the superscript in the inner product denote the output of the second-to-last layer and the weight matrix being the last in the layers. The vector x_j is then fed to the activation to compute the output

$$\hat{y}_j^M = a(x_j^M) \tag{2.43}$$

The activation function was classically the sigmoid (logistic) function but during the last decade the machine learning community has shifted to largely using the rectified linear unit (ReLU) as activation. Especially after the success of Krizhevsky et al. (2012) with AlexNET in image classification. Depending on the output (be it regression or classification) it might be useful to apply the identity transform or a soft max function in the last layer. This does not change the derivation except to change the derivatives in the last layer. We here exemplify the back propagation with the ReLU, which has the form

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.44}$$

The ReLU is obviously monotonic and its derivative can be approximated with the Heaviside step-function.

$$H(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.45}$$

We again make explicit that the choice of equations 2.40, 2.44 and 2.45 is not a be-all-end-all solution but chosen for their ubiquitous nature in modern machine learning. We then return to equation 2.41 and manipulate the expression via the chain rule

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial x_j} \tag{2.46}$$

The second derivative of the r.h.s we know from our choice of the activation to be equation 2.45, inserting to evaluate the expression we find

$$\frac{\partial E}{\partial x_j^M} = (\hat{y}_j - y_j) H(x_j^M) \tag{2.47}$$

To complete the derivation we further apply the chain rule to find the derivative in terms of the weight matrix elements.

$$\frac{\partial E}{\partial w_{ij}^M} = \frac{\partial E}{\partial x_j^M} \frac{\partial x_j^M}{\partial w_{ij}^M} \tag{2.48}$$

Recall the definition of x_j as the affine transformation defined in equation 2.37. The derivative of the inner product w.r.t the matrix elements is simply the previous layers output. Inserting this derivative of equation 2.42 we have the expression for our derivatives of interest.

$$\frac{\partial E}{\partial w_{ij}} = (\hat{y}_j - y_j)H(x_j)\mathrm{ReLU}(x_i^{M-1}) \tag{2.49}$$

Separately we compute the derivatives of 2.47 in terms of the bias nodes.

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial b_j} = (\hat{y}_j - y_j)H(x_j) \cdot 1 \tag{2.50}$$

This procedure is then repeated for the earlier layers computing the $\partial E/\partial w$ as we go. 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 x_j. As long as it is once differentiable in terms of w_{ij} we are free to pick this transformation also.

there should be a note on the importance of initialization of the weights

2.7.2 Activation functions

2.7.3 Convolutional Neural Networks

There are a couple of glaring problems with neural network layers as they were introduced in section 2.7. Firstly they are hardly efficient either by memory or by flops, secondly there is the question of invariance. Developed primarily for images convolutional neural networks aims at increasing both efficiency and modeling power when faced with data that has some translational symmetry. For image-data this is intuitively a strong assumption since the object of interest can have many different positions on the canvas and still be the same object. However convolutional neural networks do not address the two other evident symmetries for image data; rotation and scale. In short the advantage of convolutional layers is an allowance for a vastly reduced number of parameters if there is some translational symmetry in the data at the cost of much higher demands of memory. The convolutional forward pass is illustrated in figure 2.10. A $n \times n$ kernel, a matrix of weights, is convolved with the input image by taking the inner product with a $n \times n$ patch of the image iteratively moving over the entire input. The convolution is computed over the entire depth of the input, i.e. along the channels of the image. Thus the kernel maintains a $n \times n$ 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 2.51 for each filter.

$$O = W - K + 1 \tag{2.51}$$

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. The full computation of the down-sizing with these effects then is a modified version of equation 2.51, namely:

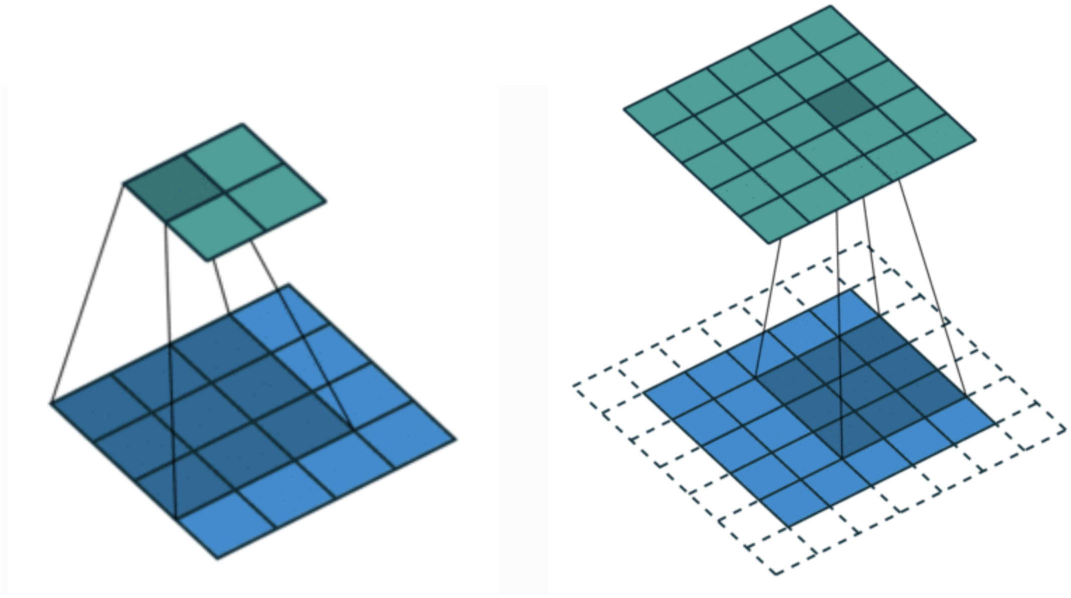
$$O = \frac{W - K + 2P}{S} + 1 \tag{2.52}$$

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 in 2012 (Krizhevsky et al. (2012)) using connected graphical processing units (GPUs). A GPU is a specialized device constructed to write data to display to a computer screen. This involves large matrix-multiplications which 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 2.4.2). 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[2].

[2] 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



(a)

(b)

Figure 2.10: Two examples of a convolutional layers forward pass, which is entirely analogous to equation 2.42 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)

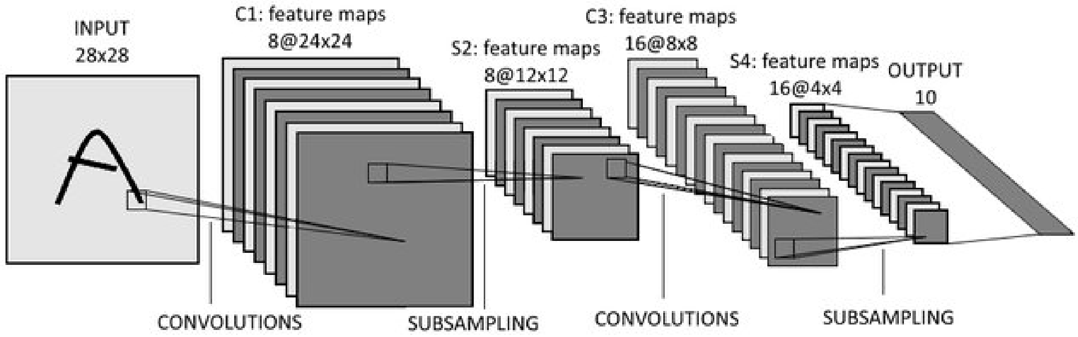


Figure 2.11: The architecture Lecun et al. (1998) used when introducing convolutional layers. Each convolutional layer maintains N kernels with initially randomized weights. These N filters act on the same input, but will extract different features from the input owing to their random initialization. The output from a convolutional layer then has size determined by equation 2.52 multiplied by the number of filters N. Every t-th layer will down-sample the input, usually by a factor two. Either by striding the convolution or by MaxPooling.

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 system 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.

2.8 Recurrent Neural Networks

2.8.1 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 2.12

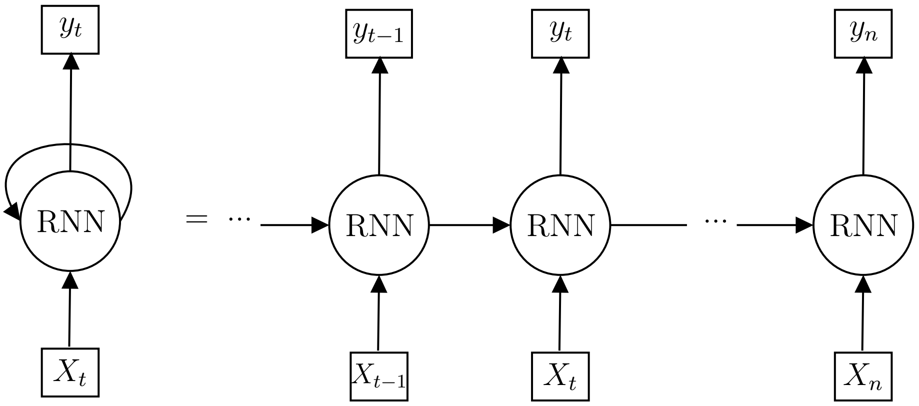


Figure 2.12: A graphical illustration of the RNN cell. The self-connected edge in the left hand side denotes the temporal nature we unroll on the right side. The cell takes as input a state vector and an input vector at time t, and outputs a prediction and the new state vector used for the next prediction. 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 2.7 trained with back-propagation.

The memory encoded by the RNN cell is encoded as a state variable. And while figure 2.12 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 2.53.

$$\langle [X_t, h_t] | W \rangle + b = h_{t+1} \tag{2.53}$$

Where the weight matrix W and bias vector b are defined in the usual manner. Looking back at figure 2.12 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 2. 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 2: 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, **W**, and bias, *b*, which will be updated by back-propagation of errors in the standard way.

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

Recurrent architectures present the researcher with a set of tools to not only model sequences but 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 the researcher principally has five basic choices regarding the sequentiality of the model. We represent these five in figure 2.13, which is copied from Karpathy (2015).

2.8.2 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

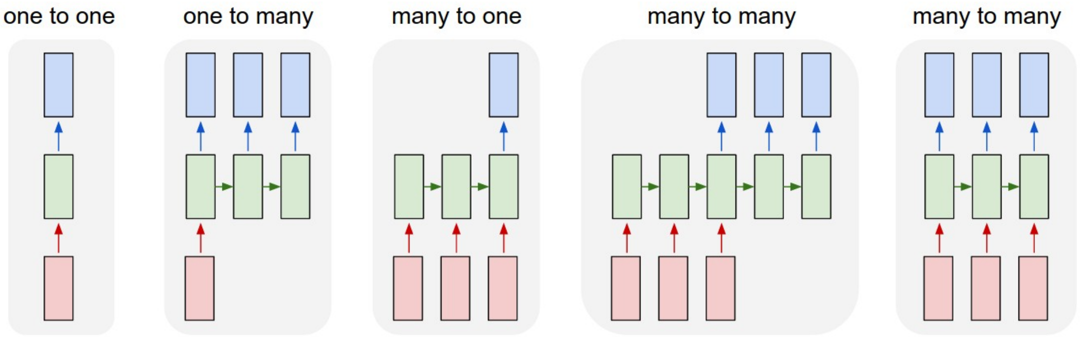


Figure 2.13: The advent of recurrent networks enabled machine learning researchers to both model complex sequential behaviors like understanding patterns in text as well as using sequences to predict a single multivariate outcome and more. The leftmost figure **1)** represents an ordinary neural network, 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)

2.9 Autoencoders

2.9.1 Introduction to autoencoders

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. Which is to say the model encodes the data into a lower dimensional latent space before reconstruction. The goal, of course, is to learn the 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. Where 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 that gives the best possible reconstruction, i.e the objective $\mathcal{O}$ is given as

$$\mathcal{O} = \arg\min_{\phi,\psi} ||X - \phi \circ \psi||^2 \tag{2.54}$$

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 more commonly through a binary crossentropy for data normalized to the range $[0, 1]$. This representation can be useful for identifying whatever information-carrying variations are present in the data. This can be thought of as an analogue to Principal Component Analysis (PCA) (Marsland (2009)). In practice the non-linear maps, ψ and ϕ, are most often parametrized and optimized as ANNs. ANNs are described in detail in section 2.7. The autoencoder was used perhaps most successfully in a denoising tasks. More recently the Machine Learning community discovered that one could impose a regularizing divergence term on the latent distribution allow for the imposition of structure in the latent space. The first of these employed a Kullback-Leibler divergence and was dubbed "Variational Autoencoders", or VAEs, (Kingma and Welling (2013)). While VAEs lost the contest for preeminence as a generative algorithm to adversarial networks they remain a fixture in the literature of expressive latent variable models with development focusing on the expressibility of the latent space ().

Citation needed. Also should I include example of denoising autoencoders ? Maybe a description at least.. Link to notebook maybe?

citation InfoVAE and β-VAE

2.9.2 Variational Autoencoder

Originally presented by Kingma and Welling (2013) the Variational Autoencoder (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 before training an ANN on the output the VAE seeks to control the latent space of the model. The goal is to be able to generate samples from the unknown distribution over the data. Imagine trying to draw a sample from the distribution of houses, we'd be hard pressed to produce anything remotely useful but this is the goal of the VAE. 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 and importantly discerning whether class membership, be it a physical property or something more abstract [3]xw is encoded.

The variational autoencoder cost

In section 2.9.1 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. We define the joint probability distribution of some hidden variables z and our data x conditional on some β. In a traditional modeling context we would coin z as including model parameters and β would then denote the hyperparameters. The variational problem is phrased in terms of finding the posterior over z, given β

$$p(z|x, \beta) = \frac{p(z, x|\beta)}{\int_z p(z, x|\beta)} \tag{2.55}$$

citation?

The integral in the denominator is intractable for most interesting problems . This is also the same problem that Markov Chain Monte Carlo (MCMC) methods aim at solving. 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?

Next we introduce 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 2.56 from a probability measure P, to another Q, by their probability density functions p, q over the set $x \in \mathcal{X}$.

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

$$D_{KL}(P||Q) = -\int_{-\infty}^{\infty} p(x)\log\left(\frac{p(x)}{q(x)}\right)dx \tag{2.56}$$

$$= \langle \log\left(\frac{p(x)}{q(x)}\right)\rangle_p \tag{2.57}$$

In the context of the VAE the KL-divergence is a measure of dissimilarity of P approximating Q (Burnham et al. (2002)). The derivation then sensibly starts with a KL-divergence.

We begin by defining $q(z|x)$ to be the true posterior distribution over the latent variable $z \in \mathcal{Z}$, conditional on our data $x \in \mathcal{X}$ with a true posterior distribution $p(x)$ and $q(z)$, with an associated probability measure Q as per our notation above. Let then the distribution over the latent space parametrized by the autoencoder be given as $\psi(z|x)$, where the autoencoder parametrizes a distribution $\eta(x)$, and an associated probability measure Ψ. And recalling Bayes rule for conditional probability distributions $p(z|x) = (p(x|z)p(z)/p(x)$ we then have

$$D_{KL}(\Psi||Q) = \langle \log\left(\frac{\psi(z|x)}{q(z|x)}\right)\rangle_\psi \tag{2.58}$$

$$= \langle \log\left(\psi(z|x)\right)\rangle_\psi - \langle \log\left(p(x|z)q(z)\right)\rangle_\psi + \log\left(p(x)\right) \tag{2.59}$$

$$= \langle \log\left(\frac{\psi(z|x)}{q(z)}\right)\rangle_\psi - \langle \log\left(p(x|z)\right)\rangle_\psi + \log\left(p(x)\right) \tag{2.60}$$

Note that the term $-\langle \log\left(p(x|z)\right)\rangle_\psi$ is the log likelihood of our decoder network which we can optimize with the cross entropy as discussed in section 2.3. Rearranging the terms we arrive at the variational autoencoder cost

$$\log(p(x)) - D_{KL}(\Psi||Q) = \langle \log\left(p(x|z)\right)\rangle_\psi - \langle \log\left(\frac{\psi(z|x)}{q(z)}\right)\rangle_\psi \tag{2.61}$$

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 2.55. The solution appears by approximating the KL-divergence up to an additive constant by estimating the evidence lower bound (ELBO). This function is defined as

$$ELBO(q) = \langle \log(p(z,x))\rangle - \langle \log(q(z|x))\rangle \tag{2.62}$$

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

$$ELBO(q) = \langle \log(p(z))\rangle + \langle \log(p(x|z))\rangle - \langle \log(q(z|x))\rangle \tag{2.63}$$

Finally the ELBO can be related to the VAE loss by applying Jensen's inequality (J) to the log evidence

$$\begin{aligned}
\log(p(x)) &= \log \int_z p(x|z)p(z) \quad &(2.64)\\
&= log \int_z p(x|z)p(z)\frac{q(z|x)}{q(z|x)} \quad &(2.65)\\
&= log\langle p(x|z)p(z)/q(z|x)\rangle \quad &(2.66)\\
&\overset{(J)}{\geq} \langle \log(p(x|z)p(z)/q(z|x))\rangle \quad &(2.67)\\
&\geq \langle \log(p(x|z))\rangle + \langle \log(p(z))\rangle - \langle \log(q(z|x))\rangle \quad &(2.68)
\end{aligned}$$

Which shows that the function enacted by the ELBO approximates the VAE loss up to a constant i.e. the KL loss on the RHS in equation 2.61. 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 2.7.1. We note again that in the above notation we would parametrize the distribution $p(x|z)$ as a neural network, in machine learning parlance called the generator network and denoted as ϕ in section 2.9.

2.9.3 Regularizing Latent Spaces

As introduced in section 2.9.2 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 (Kingma and Welling (2013)). Two of the seminal papers published on the topic is the $\beta - VAE$ paper by Higgins et al. (2017) introducing a weight on 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, using any positive definite kernel $k(\cdot, \cdot)$[4] it is defined as:

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

$$D_{MMD} = \langle k(z, z') \rangle_{p(z),p(z')} - 2\langle k(z, z') \rangle_{q(z),p(z')} + \langle k(z, z') \rangle_{q(z),q(z')} \tag{2.69}$$

2.9.4 Deep Recurrent Attentive Writer

One of the central challenges of the ELBO as presented in equation 2.62 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 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.

$$\hat{\mathbf{x}} = \mathbf{x} - \sigma(\mathbf{c_{t-1}}) \tag{2.70}$$

$$\mathbf{r}_t = \text{read}(\mathbf{x}_t, \hat{\mathbf{x}}_t,) \tag{2.71}$$

$$\mathbf{h}_t^{enc} = \text{RNN}^{enc}(\mathbf{h}_{t-1}^{enc}, [\mathbf{r}_t, \mathbf{h}_{t-1}^{dec}]) \tag{2.72}$$

$$\mathbf{z}_t = \text{latent}(\mathbf{h}_t^{enc}) \tag{2.73}$$

$$\mathbf{h}_t^{dec} = \text{RNN}^{dec}(\mathbf{h}_{t-1}^{dec}, \mathbf{z}_t) \tag{2.74}$$

$$\mathbf{c}_t = \mathbf{c}_{t-1} + \text{write}(\mathbf{h}_t^{dec}) \tag{2.75}$$

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 2.14.

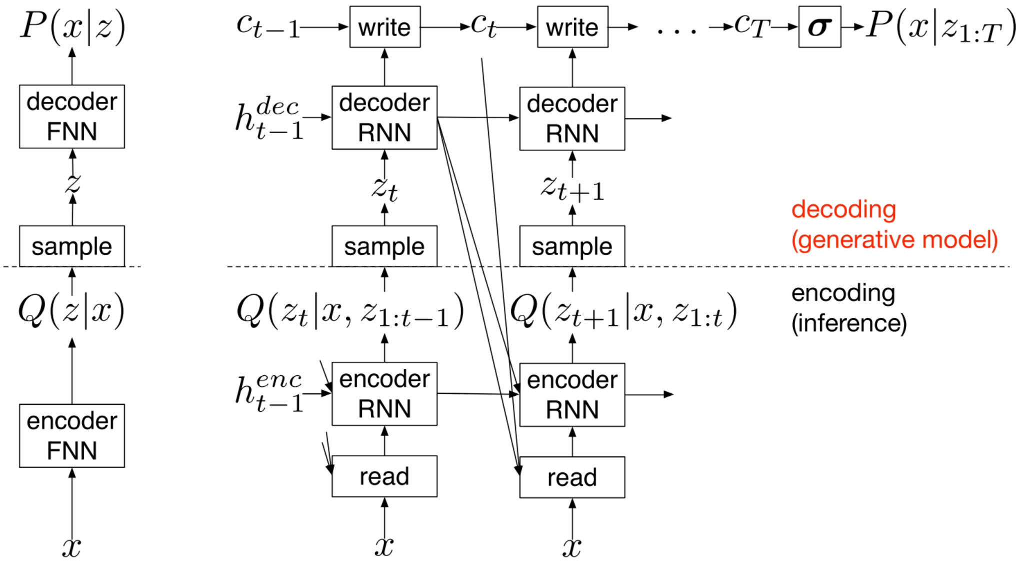


Figure 2.14: Left: an ordinary oneshot encoderdecoder network. **Right:** DRAW network that iteratively constructs the canvas using RNN cells as the encoder/decoder pair. The final output is then iteratively constructed using a series of updates on a canvass, c_t. DRAW function read that process the 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.

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

$$\mu_X^i = g_X + (i - N/2 - 0.5)\delta \tag{2.76}$$

$$\mu_Y^j = g_Y + (j - N/2 - 0.5)\delta \tag{2.77}$$

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

$$\tilde{g}_x,\, \tilde{g}_y,\, \log\sigma^2,\, \log\gamma = \text{Dense}(\mathbf{h}_t^{dec}) \tag{2.78}$$

Where σ^2 is the isotropic variance of the Gaussian filters, and γ the multiplicative intensity of the filtering. They are taken to be logarithmic to ensure positivity by the exponential function. Gregor et. al makes an additional transformation on the centers to ensure that the initial patch roughly covers the entire input image.

$$g_x = \frac{W+1}{2}(\tilde{g}_x + 1) \tag{2.79}$$

$$g_y = \frac{H+1}{2}(\tilde{g}_y + 1) \tag{2.80}$$

$$\tag{2.81}$$

The above equations included terms to compute and scale δ, that we have eliminated and added it instead as a constant hyperparameter. Given the scaled centers, input dimensions $H \times W$ we can compute the filter-banks $F_x \in \mathcal{R}^{N\times W}$ and $F_y \in \mathcal{R}^{N\times H}$.

$$F_x[i, w] = \frac{1}{Z_x} e^{-\frac{(w-\mu_x^i)^2}{2\sigma^2}} \tag{2.82}$$

$$F_y[j, h] = \frac{1}{Z_y} e^{-\frac{(h-\mu_y^j)^2}{2\sigma^2}} \tag{2.83}$$

Where we denote a point in the input $(h,\, w)$, and a point in the attention patch $(i,\, j)$. The filters-banks are multiplied with a normalization constant s.t. $\sum_w F_x[i,w] = 1$, Z_y is defined in the same way. We are now finally ready to define the read and write functions with attention parameters. The read operation returns a concatenation of the same patch extracted from the input and error images. The write function returns an array that is added to the current canvass c_t. From Gregor et al. (2015) read function is defined by equation 2.84

$$\text{read}(\mathbf{x}, \hat{\mathbf{x}}, \mathbf{h}^{dec}_{t-1}) = \gamma[F_y \mathbf{x} F_x^T, F_y \hat{\mathbf{x}} F_x^T] \tag{2.84}$$

For the write function we define a new set of attention parameters e.g. $\hat{\gamma}$. And 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.

$$\text{write}(w_t) = \hat{\gamma} \hat{F}_y^T w_t \hat{F}_x \tag{2.85}$$

Notice the transposition order in equation 2.85 is reversed with respect to the order in equation 2.84.

Latent samples and loss

write out latent sample/loss subsects

2.9.5 Deep Clustering

Chapter 3

Experimental background

3.1 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?

3.2 Active Target Time Projection Chambers

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

3.3 Data

In this thesis we will work with data from the $Ar(p, p')$ 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)

Part II

Implementation

Chapter 4

Methods

4.1 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 2, 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 2 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.

4.2 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. 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.

4.2.1 The computational graph

To understand the program flow of the later algorithm implementations we begin by introducing the fundamental concepts of TensorFlow code [1]. The heart of which is the computational graph. A code snippet with an associated graph is included in figure 4.1 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] 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.

```python
import tensorflow as tf

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

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

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

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

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

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

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

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

Figure 4.1: This short script describes the setup required to compute a forward pass in a neural network as described in section 2.7. 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 2.7. 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 4.3. 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?

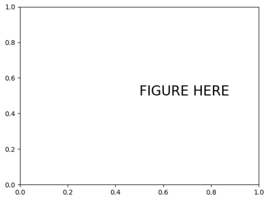


Figure 4.2: A graph representation of the short script in figure 4.1. The nodes represent TensorFlow operation types including variable declarations and operations on those including Add and MatMul operations. In the versions of TensorFlow prior to the release of TF 2.0 this graph did not execute immediately 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)

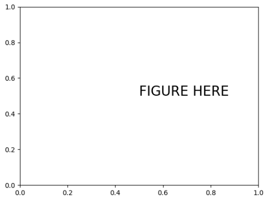


Figure 4.3

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.

```python
import tensorflow as tf

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

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

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

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

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

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

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

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

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

for i in range(epochs):

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

Figure 4.4: 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

4.2.2 Architecture

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 2.9.4) and one for the non-sequential convolutional autoencoder 2.9. 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 wrt. the output(s) for that graph.

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 2.9.3) 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 4.2.2 and 4.2.2. 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 4.4 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 2.6 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 2.9.5. 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 2.9.5

SKLEARN CITATION

Convolutional Autoencoder

DRAW

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, 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.

Part III

Results

4.3 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, clean-real, and full-real datasets. For each dataset we evaluate the performance of the two frameworks of model for both semi supervised classification and clustering tasks. The models are described in terms of their hyperparameters in table 4.1 for the convolutional autoencoder and table 4.2 for the DRAW-analogues.

4.3.1 Performance

We measure the performance in the semi-supervised case by accuracy of the linear classifier (logistic regression), and the $f1$ score. The classifier is trained on a subset of the train set and evaluated on the remainder to estimate the OOS error. The best configuration will then be re-trained and we evaluate this model on the test set to estimate our top performers OOS error. The accuracy is computed in terms of the True Positive (TP) predictions and the True Negatives (TN) divided by the total number of samples. We will use the False Positives (FP) and False Negatives (FN) later and so introduce their abbreviation here. The accuracy is related to the rand index which we will use to measure clustering with the distinction that for accuracy we know the ground truth during training. Mathematically we define the accuracy in equation 4.1. Accuracy is bounded in the interval $[0, 1]$

$$\text{accuracy} = \frac{TP + TN}{FN + TN + TP + FP} \tag{4.1}$$

The accuracy as presented in equation 4.1 does not account for class imbalance, consider for example a problem where one class occurs as 99% of the sample, a trivial classifier predicting only that class will achieve an accuracy of $\text{acc} = 0.99$. This is for obvious reasons a problematic aspect of accuracy and so the remedy is often to measure multiple metrics of performance, we chose the $f1$ score per-class in addition to accuracy. The $f1$ score is defined in terms of the precision and recall of the prediction. They are simply true positives weighted by the false positives and negatives. We define recall and precision in equations 4.2 and 4.3 respectively.

$$\text{recall} = \frac{TP}{TP + FP} \tag{4.2}$$

$$\text{precision} = \frac{TP}{TP + FN} \tag{4.3}$$

The $f1$ score is simply the harmonic mean of precision and recall for each class. In mathematical terms we present it in equation 4.4.

$$f1 = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{4.4}$$

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. We also use the $f1$ score as it has been used in previous studies of the AT-TPC experiments performed at Davidson College in collaboration with the NSCL. Most recently Kuchera et al. (2019) published a paper on supervised methods for track identification, the results from which we will use to evaluate the methods implemented for this thesis.

add image from each experiment to results subsection header

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 2.7.2
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 2.6
β_1	Linear float	Momentum parameter, described in 2.6.1
β_2	Linear float	Second moment momentum parameter. Described in 2.6.3

Table 4.1: 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 2.6
β_1	Linear float	Momentum parameter, described in 2.6.1
β_2	Linear float	Second moment momentum parameter. Described in 2.6.3

Table 4.2: Hyperparameters for the draw algorithm as outlined in section 2.9.4. 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 2.9.4.

4.4 Simulated AT-TPC events

The simulated AT-TPC tracks were simulated with the pytpc package developed at the NSCL. 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

citation

figure of 3D simulated track and 2D representation

4.4.1 Classification of events

From the simulated data we have two classes of events; proton and carbon. Our hypothesis is that we can separate these with a linear classifier, additionally we will investigate how many labeled samples we need to achieve good classification. The investigation of the performance will be separated in the sequential and non-sequential models. The simulated data is very simple and does not necessitate the full hyperparameter search machinery. We begin by considering the non-sequential Autoencoders.

Convolutional Autoencoder classification results[2]

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. 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 $\mathbf{X}_L \subset \mathbf{X}_F$. Furthermore, the tuple γ_L has to be split in training and test sets. The test partition is set to be 30% of the samples. The hyperparameters were tuned with the RandomSearch architecture which searches in a semi structured way over all the parameters given in table 4.1 we list the experiments with their corresponding proton $f1$ score in appendix A.1. We optimize the classification performance using $M = 40$ semi-structured random experiments. To illustrate the difference in quality and convergence of the runs we show the plot of the resulting loss curves of the highest $N = 10$ runs in figure 4.5. In the figure lighter colors represent higher proton $f1$ scores, and as such the same color across the reconstruction and latent losses

[2] The scripts used to produce the results for this section can be found in the repository at: scripts/convae_simulated.py, notebooks/draw_model.ipynb, and notebooks/latent_space_simulated.ipynb

represent the same experiment. We also denote different scales and corresponding markers for the reconstruction and latent losses. We observe that in the top 10 performing runs there are no versions of the model that maintains a VAE-style Kullback Leibler loss on the latent space. And that the very highest performing model maintains no regularization at all, while considerably more complex than the runners up as seen from appendix A.1.

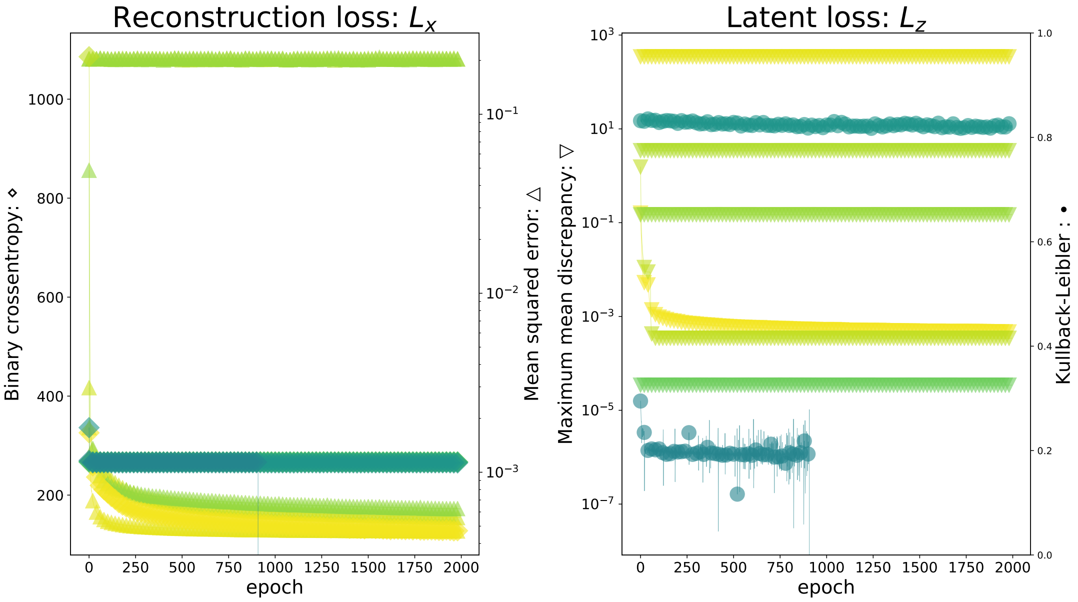


Figure 4.5: Distribution of the $N = 10$ best loss curves from $M = 40$ experiments with a convolutional autoencoder. Lighter colors indicate higher proton $f1$ scores, and as such there is an implied 1-1 correspondence between the plots. The lightest colored line represents a proton score of $f1_p = 0.98$, implying near perfect classification. While the worst performing model included in the plot has a score of $f1_p = 0.92$. The line-markers denote the loss-type for the associated run, with the axis the run is plotted on containing the corresponding legend. We observe that the maximum-mean-discrepancy loss on the latent distribution consistently outperforms the Kullback Leibler loss as none are included in the top 10. The full table is listed in appendix A.1

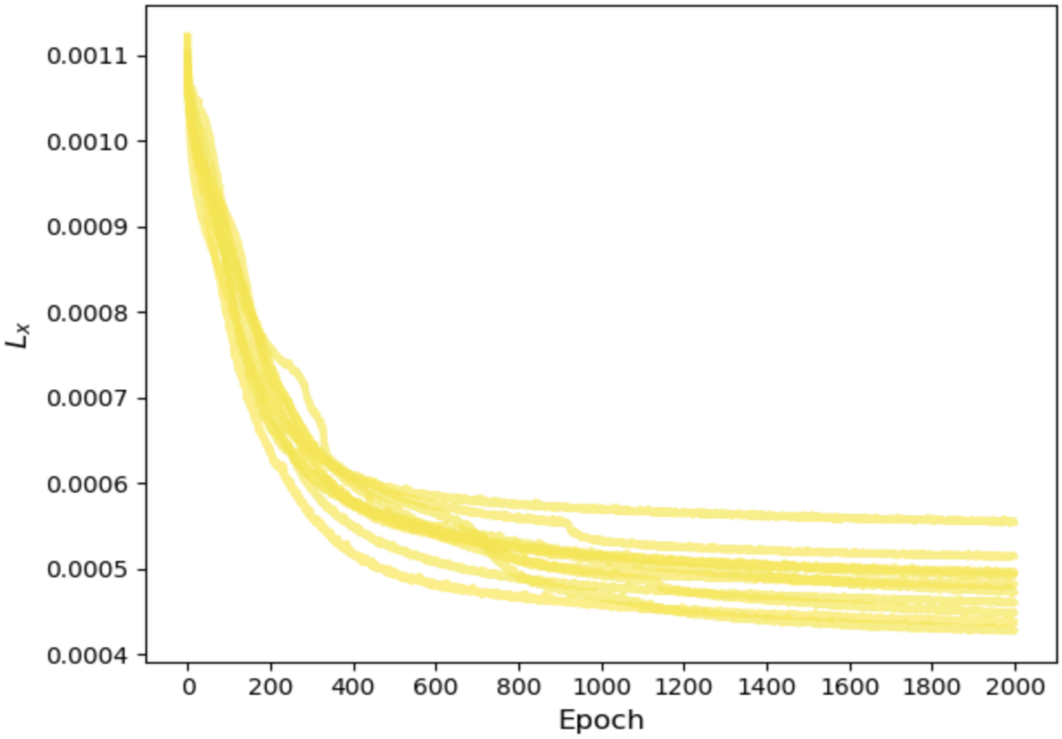
The classifier was a logistic regression model from the scikit −learn python library trained on the latent expressions of the train partition of $\mathbf{X}_L$. The hyperparameters of the best performing model, with the least complexity, is listed in table 4.3. We chose a simpler model as it has fewer variables to fit and as such it is less prone to overfitting effects unsupervised models can exhibit.

Furthermore we wish to estimate the variability of the highest performing model. To achieve this we then re-ran the convolutional autoencoder $N = 10$ times using the configuration listed in table 4.3. The model performs strongly

Hyperparameter	Value
Convolutional parameters:	
Number of layers	3
Kernels	[11, 5, 3]
Strides	[2, 2, 2]
Filters	[16, 16, 8]
Network parameters:	
Activation	ReLU
Latent type	MMD
Latent dimension	3
β	10
Optimizer parameters:	
η	1×10^{-1}
β_1	0.55
β_2	0.99

Table 4.3: Hyperparameters that gives the strongest performance with the least complexity on the simulated events. The configuration chosen is listed in appendix A.1

with very little variation in the proton $f1$ scores. We list the $f1$ scores by class in table 4.4 and display the associated loss curves in figure 4.6.



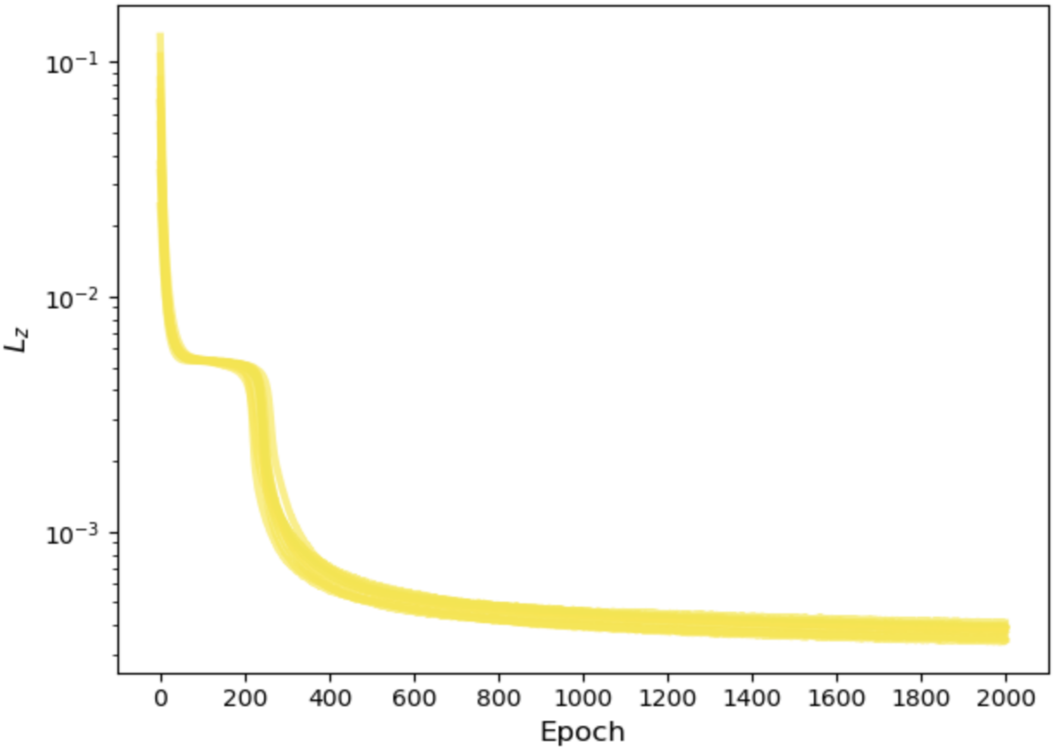


Figure 4.6: Reconstruction and latent loss curves for the $N = 10$ runs of the best performing convolutional autoencoder using the configuration listed in table 4.3. The lightness of color indicates the proton $f1$ score achieved by the given model. The lightest color in the plot corresponds to a score of about $f1_p = 0.98$.

	Proton			Carbon		
	f1	recall	precision	f1	recall	precision
Train	0.88 $\pm 7.03 \times 10^{-2}$	0.88 $\pm 5.93 \times 10^{-2}$	0.86 $\pm 8.56 \times 10^{-2}$	0.90 $\pm 4.39 \times 10^{-2}$	0.89 $\pm 5.32 \times 10^{-2}$	0.87 $\pm 7.37 \times 10^{-2}$
Test	0.88 $\pm 7.07 \times 10^{-2}$	0.89 $\pm 6.40 \times 10^{-2}$	0.87 $\pm 7.67 \times 10^{-2}$	0.90 $\pm 6.06 \times 10^{-2}$	0.89 $\pm 6.80 \times 10^{-2}$	0.88 $\pm 7.11 \times 10^{-2}$

Table 4.4: Performance as measured by $f1$, precision and recall of the convolutional autoencoder as specified in table 4.3 on simulated at-tpc data. The standard errors are estimated by running the simulation $N = 10$ times with random initialization of weights for each experiment

To verify the performance it helps to consider the reconstruction. We would expect a good performing model to be able to reconstruct similar representations to what it was given as input as this structural information then has to have passed through the information bottleneck. In figure 4.7 we show that the reconstruction resembles the original strongly, even discerning the minute breaks in the carbon line caused by some aspect of the simulation. While we also note the blurring typical of the pixel-wise reconstruction.

Lastly we investigate the relationship between the hyperparameters and the proton $f1$ score. Among the reasons for using a random-search procedure for the hyperparameter search is the possibly complex, non-monotonic relationships between parameters and the outcome we wish to measure. As such the ordinary tools of estimating correlation and co-variance must come with caveats. Table 4.5 of correlation between the parameters and the proton $f1$ score, but we note that while none of the correlations were significant - this may be a consequence of violated assumptions and non-monotonic relationships.

DRAW classification results

4.4.2 Clustering of events

Convolutional autoencoder clustering results

DRAW clustering results

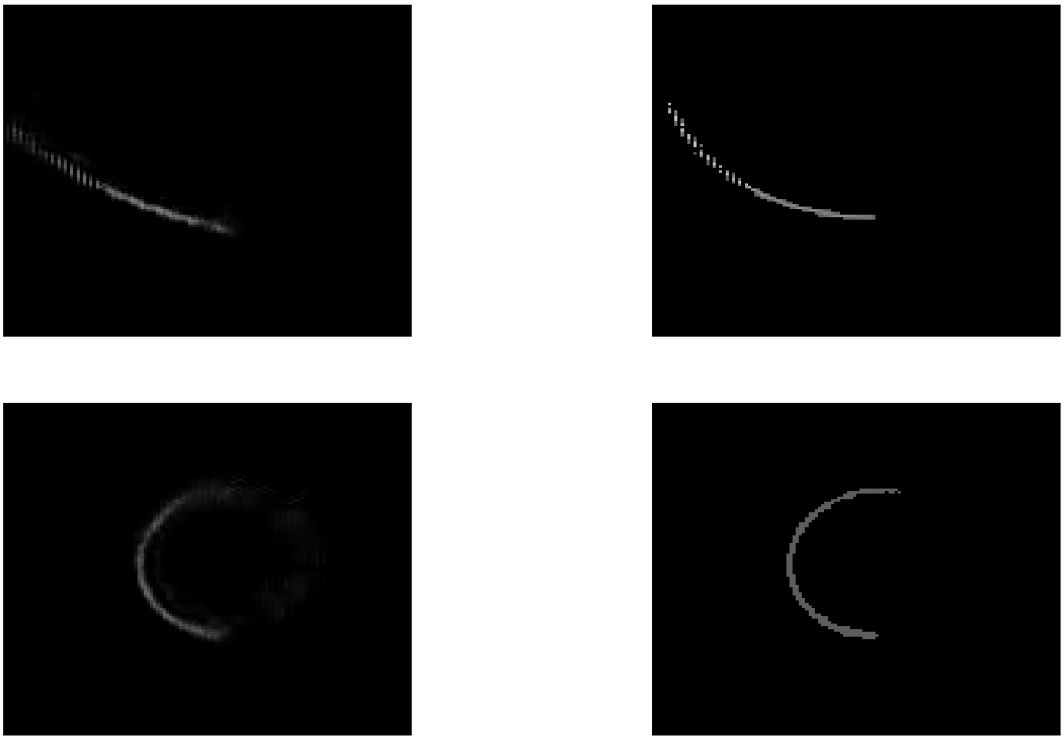


Figure 4.7: Reconstructions and inputs produced by the convolutional autoencoder with the configuration listed in table 4.3. The left column are the reconstructions of the right column which are model inputs. We note that the model is able to pick up even the breaks in the line-structure from the simulations, and that reconstruction of the proton events exhibit a blurring effect common to pixel wise mse or bce losses.

	ρ_s	p
proton f1 score	1	0
N parameters	0.36	0.079
largest kernel	0.38	0.058
N layers	-0.26	0.21
latent dimension	0.17	0.43
batchnorm	0.26	0.21
β	-0.14	0.51
β_1	-0.065	0.76
η	-0.14	0.51
kld	-0.15	0.46
mmd	0.11	0.59
none	-0.012	0.96
bce	0.1	0.63
mse	-0.1	0.63
lrelu	-0.26	0.21
relu	0.26	0.21

Table 4.5: Spearman rank correlation of the proton $f1$ score with the other hyperparameters from appendix A.1. The strongest monotonic relationships are with the number of parameters and the largest kernel, but no relationship is significant at the $\alpha = 0.001$ level. This is perhaps unsurprising as the assumption is that the hyperparameters co-vary differently and in a highly complex manner. Nevertheless there are still indications that batch normalization and maximum mean discrepancy regularization are positively related to the performance.

4.5 Real AT-TPC 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 Ar_{46} experiment subject to analysis in this thesis. There is both structural noise that can be attributed to electronics cross-talk and possible interactions with cosmic background radiation and other 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.

4.5.1 Classification of events

Convolutional autoencoder classification results

DRAW classification results

4.5.2 Clustering of events

Convolutional autoencoder clustering results

DRAW clustering results

4.6 Filtered real AT-TPC events

4.6.1 Classification of events

Convolutional autoencoder classification results

DRAW classification results

4.6.2 Clustering of events

Convolutional autoencoder clustering results

DRAW clustering results

Part IV

Discussion and Conclusion

4.7 Discussion

In this section 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 representations. This performance will be contextualized by measuring against the results on similar tasks in the work of Kuchera et al. (2019).

4.7.1 Classification

Simulated AT-TPC events

Using the RandomSearch framework we were able to find a network configuration for both the convolutional and DRAW networks that shows very strong performance on the simulated data. Figure 4.4 shows that the strongest performance was obtained using a mean squared error loss function for reconstruction. The fact that one loss function is preferred can be simply argued from the function-shape of the respective losses and aspects of gradient descent. We illustrate this discrepancy in figure 4.9 where we note that the cross entropy has an asymmetry that punishes higher predictions in a way that is not physically substantiated. As for the strong preference for the maximum mean discrepancy error this can be explained by a rather subtle difference in the objective of the latent loss. 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.

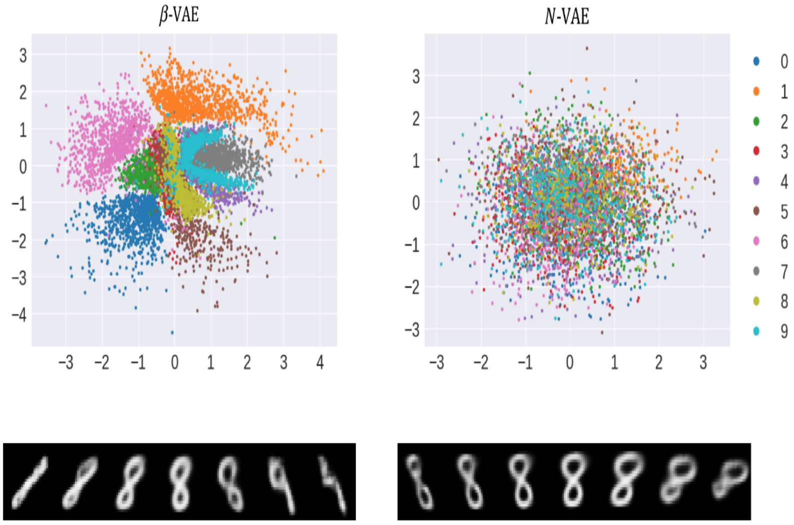


Figure 4.8: Demonstrating the difference of capturing class and feature information in the latent space. On the left the β-VAE pushes the autoencoder to a representation favoring encoded class information in the latent space. The spaces between the class blobs makes for a poor generative algorithm, but 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)

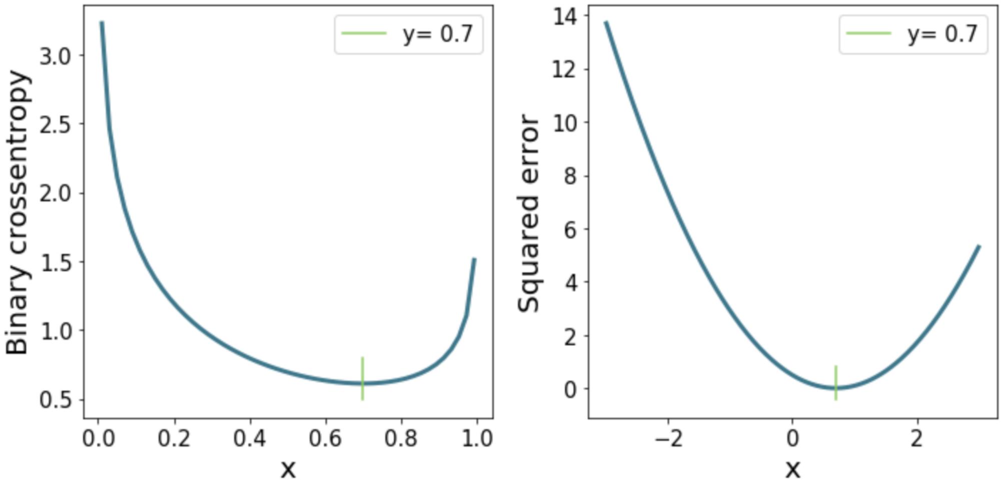


Figure 4.9: Illustrating the difference between the function shapes of binary cross-entropy and the mean squared error for the same target value $y = 0.7$. We observe that though the two functions have the same minimum the landscape surrounding it is very different. Perhaps most telling is the asymmetry of the cross entropy, implicitly telling the optimizer that values above the targets are measurably worse than those below as measured by the steepness of the gradient. For the AT-TPC data this then implies that predicting higher charge values are worse than predicting lower than the target, and implication that is not physically substantiated

Appendices

Appendix A

Results on simulated data

proton f1-score	N parameters	largest kernel	N layers	latent dimension	latent loss	reconstruction loss	activation function	batchnorm	β	β_1	η
0.99	4754	17	3	150	none	mse	relu	False	1e-05	0.73	1e-05
0.99	3798	13	6	50	mmd	bce	lrelu	False	0.001	0.82	0.001
0.98	3636	15	5	150	mmd	mse	lrelu	False	1e-05	0.69	1e-05
0.98	2408	11	3	3	mmd	bce	relu	True	0.1	0.56	0.1
0.98	760	7	5	150	none	bce	lrelu	True	1e-05	0.25	1e-05
0.96	1712	7	3	200	mmd	mse	relu	False	0.1	0.43	0.1
0.95	1740	11	5	50	mmd	bce	relu	False	0.0001	0.53	0.0001
0.94	464	7	5	100	mmd	mse	relu	False	0.0001	0.69	0.0001
0.94	5194	15	4	50	none	mse	relu	True	0.1	0.93	0.1
0.92	9266	17	5	10	none	bce	relu	True	1e-05	0.32	1e-05
0.92	272	5	5	50	mmd	bce	relu	False	0.001	0.46	0.001
0.91	4770	11	5	200	kld	bce	relu	True	0.01	0.67	0.01
0.9	7346	17	4	150	mmd	mse	relu	True	0.0001	0.71	0.0001
0.9	1524	11	4	10	none	bce	lrelu	True	0.001	0.51	0.001
0.89	688	5	3	50	mmd	bce	lrelu	False	0.1	0.81	0.1
0.88	3676	15	4	20	kld	bce	lrelu	False	0.0001	0.28	0.0001
0.83	562	7	6	200	mmd	mse	lrelu	False	0.001	0.37	0.001
0.81	7316	11	6	20	kld	bce	relu	False	0.1	0.63	0.1
0.79	3546	17	5	50	none	bce	lrelu	False	0.0001	0.85	0.0001
0.76	1112	11	5	100	none	mse	relu	False	0.0001	0.92	0.0001
0.75	3416	9	6	10	mmd	bce	lrelu	False	0.01	0.77	0.01
0.74	676	5	3	150	mmd	mse	lrelu	False	0.01	0.57	0.01
0.67	154	5	5	20	none	mse	relu	False	0.001	0.93	0.001
0.65	108	3	6	100	mmd	mse	lrelu	False	1e-05	0.22	1e-05
0.65	14904	17	6	200	none	mse	relu	True	1e-05	0.63	1e-05
0.59	6280	13	6	20	kld	mse	relu	True	0.1	0.63	0.1

proton f1-score	N parameters	largest kernel	N layers	latent dimension	latent loss	reconst -ruction loss	activation function	batch -norm	β	β_1	η
0.57	7008	17	4	10	kld	bce	lrelu	True	0.1	0.82	0.1
0.55	1480	7	5	200	kld	mse	relu	False	0.0001	0.61	0.0001
0.53	6656	11	6	100	kld	mse	relu	False	0.0001	0.39	0.0001
0.53	5984	13	6	200	mmd	mse	lrelu	False	0.01	0.82	0.01
0.52	12824	13	6	200	kld	mse	lrelu	False	0.001	0.77	0.001
0.52	688	5	3	3	mmd	bce	relu	False	0.0001	0.25	0.0001
0.51	1314	9	5	200	kld	mse	lrelu	True	0.01	0.8	0.01
0.5	5140	15	6	3	mmd	bce	relu	False	0.0001	0.46	0.0001
0.5	216	3	6	3	kld	bce	lrelu	True	0.0001	0.25	0.0001
0.5	626	5	3	150	kld	mse	relu	False	0.1	0.91	0.1
0.46	1276	11	3	50	kld	mse	relu	True	0.001	0.84	0.001
0.45	1636	7	4	150	kld	bce	relu	False	0.001	0.45	0.001
0	4658	13	4	200	none	bce	lrelu	False	1e-05	0.28	1e-05

Table A.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 5

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.

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.

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.

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 Ivo Danihelka. *Proceedings of Machine Learning Research*, 37.

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.

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

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*.

Kuchera, M., Ramanujan, R., Taylor, J., Strauss, 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.

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.

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

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 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.

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