Modern Machine Learning Algorithms: Applications in Nuclear Physics

by

Robert Solli

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

August 7, 2019

Contents

1 Introduction **7**
- 1.1 Research question and hypotheses 7
- 1.2 Why machine learning? 7

I Theory and Experimental Background **9**

2 Machine Learning Theory **11**
- 2.1 Introduction 11
- 2.2 Model Fitting 11
  - 2.2.1 On information 12
  - 2.2.2 Over and under-fitting 14
- 2.3 Logistic Regression 17
- 2.4 Linear Regression 17
  - 2.4.1 Regularization 20
  - 2.4.2 Batch Normalization 21
- 2.5 Performance validation 23
  - 2.5.1 Performance metrics 23
  - 2.5.2 Labeled samples 24
  - 2.5.3 Cross validation 24
  - 2.5.4 The bias-variance relationship 24
- 2.6 Hyperparameters 24
  - 2.6.1 Hand holding 25
  - 2.6.2 Grid Search 25
  - 2.6.3 Random Search 26
- 2.7 Gradient Descent 27
  - 2.7.1 Momentum Gradient Descent 31
  - 2.7.2 Stochastic & Batched Gradient Descent 31
  - 2.7.3 adam 33
- 2.8 Neural Networks 34
  - 2.8.1 Backpropagation 36
  - 2.8.2 Neural architectures 38
  - 2.8.3 Activation functions 38

2.8.4 Convolutional Neural Networks 39
2.9 Recurrent Neural Networks 42
2.9.1 Introduction to recurrent neural networks 42
2.9.2 Long short-term memory cells 44
2.10 Autoencoders 46
2.10.1 Introduction to autoencoders 46
2.10.2 Variational Autoencoder 47
2.10.3 Optimizing the variational autoencoder 49
2.10.4 Regularizing Latent Spaces 50
2.10.5 Deep Recurrent Attentive Writer 51
2.10.6 Deep Clustering 54
2.11 Neural architectures 54
2.11.1 Classification 55
2.11.2 Clustering 55
2.11.3 Pre-trained networks 56

3 Experimental background **59**
3.1 Introduction 59
3.2 Active Target Time Projection Chambers 59
3.2.1 A note on nuclear physics 59
3.2.2 AT-TPC details 60
3.3 Data 61
3.3.1 Simulated ^{46}Ar events 61
3.3.2 Full ^{46}Ar events 61
3.3.3 Filtered $^{4}6Ar$ events 62

II Implementation **63**

4 Methods **65**
4.1 Introduction 65
4.2 TensorFlow 65
4.2.1 The computational graph 66
4.3 Deep learning algorithms 69
4.3.1 Convolutional Autoencoder 71
4.3.2 DRAW 73
4.4 Hyperparameter search architecture 73

III Results **75**

5 Experimental setup and design **77**

6 Classification results **81**
6.1 VGG16 81
6.2 Convolutional Autoencoder 84
6.3 DRAW 85

7 Clustering of AT-TPC events **87**
7.1 Convolutional Autoencoder results 87
7.2 DRAW results 87

IV Discussion and Conclusion **89**
7.3 Discussion 91
    7.3.1 Classification 91

Appendices **95**

A Kullback-Leibler divergence of of Gaussian distributions **97**

B Neural network architectures **99**

C Results on simulated data **101**

8 Notes **103**

List of Figures

2.1 Illustrating overfitting with polynomial regression 16
2.2 Geometric interpretation of the L_1 and L_2 regularization and the squared error cost 22
2.3 Why randomsearch works 27
2.4 Sub-optimal gradient descent 29
2.5 Optimal gradient descent 29
2.6 The impact of η on performance 30
2.7 Exponential decay in momentum gradient descent 32
2.8 Effect of the batch size on performance 33
2.9 Fully connected neural network illustration 35
2.10 Convolutional layer illustration 41
2.11 Original LeNet architecture 42
2.12 Recurrent neural network cell 43
2.13 Archetypes of recurrent neural architectures 45
2.14 DRAW network architecture 52

3.1 Chart of the nuclides 60

4.1 FCN forward pass in TensorFlow 67
4.2 Graph representation of the forward pass of a simple FCN 68
4.3 68
4.4 Computing gradients and performing back-propagation in TensorFlow 69

6.1 VGG16 performance on labeled subsets 83

7.1 Difference between generative and discriminative latent spaces . . 92
7.2 Illustrating differences in the shapes of the MSE and BCE-loss . . 93

Todo list

🟧 Need to add abbreviations to list . . . 7
🟧 add some plots of linear data with noise, regression line and show the errors? . . . 20
🟧 add subsection on dropout and batchnorm . . . 21
🟧 maybe dropout? . . . 21
🟧 add description of n-labled algo? . . . 24
🟧 write out section on CV . . . 24
🟧 write out section on bias-variance . . . 24
🟧 make proper figure for ann . . . 35
🟧 there should be a note on the importance of initialization of the weights 38
🟧 update notation with layer indexation, and algorithm for backprop? . . 38
🟧 Make activation function plots and write out section on act. functions . 39
🟧 Include mathematical formulation of the convolutional layer . . . 40
🟧 Citation needed. Also should I include example of denoising autoencoders ? Maybe a description at least.. Link to notebook maybe? . . . 46
🟧 citation InfoVAE and β-VAE . . . 46
🟧 citation? . . . 47
🟧 citation? Comph-phys 2 compendium? . . . 47
🟧 write out latent sample/loss subsects . . . 54
🟧 clean up section on model architectures . . . 57
🟧 Added part from previous results, needs molding . . . 61
🟧 figure of 3D simulated track and 2D representation . . . 61
🟧 describe the physics in a bit more detail boy . . . 62
🟧 write filtered section . . . 62
🟧 add plots of events in 2d and 3d . . . 62
🟧 add table with data descriptions. N samples, N labelled . . . 62
🟧 whats that damn abbreviation? . . . 67
🟧 SKLEARN CITATION . . . 71
🟧 add image from each experiment to results subsection header . . . 77
🟧 correct appendix link and add performance tables? . . . 81
🟧 add plot with reconst/loss vs f1 scores . . . 84
🟧 add architecture tables, note on latent divergence? . . . 84

🟧 add table of architectures 84
🟧 add n-labeled plots 84

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 2.8 then followed by autoencoders in section 2.10 and finally recurrent neural networks in 2.9.

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 simply defined as the negative loss. Gradient descent is discussed in some detail in section 2.7.

---

[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.

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 2.5.4. 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

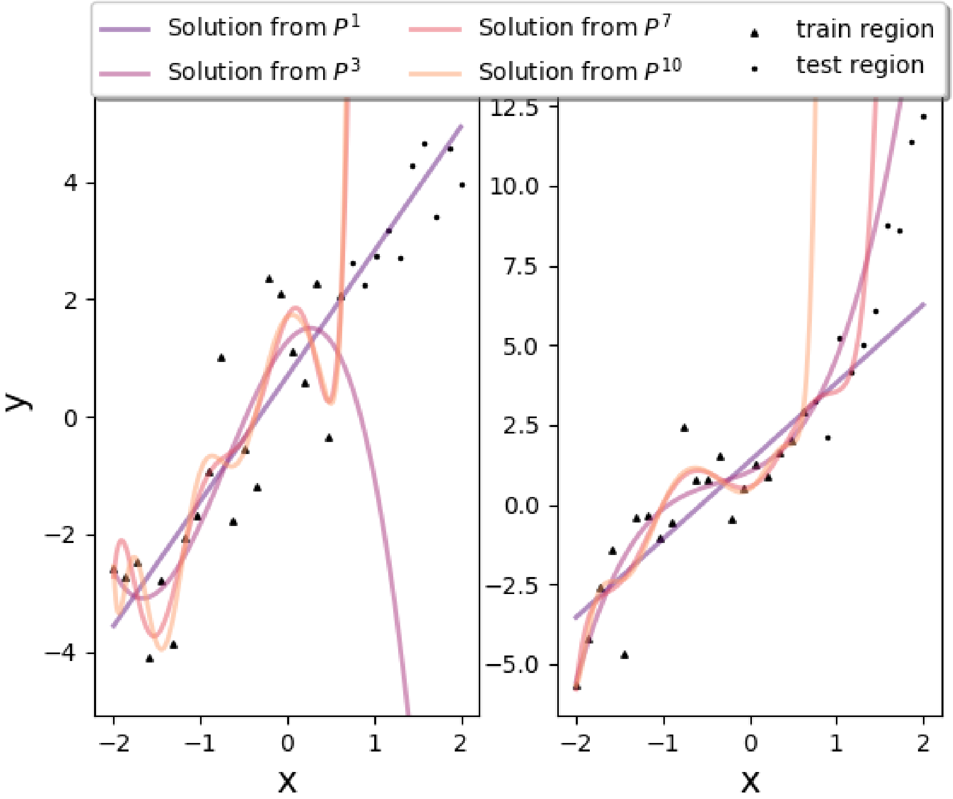


Figure 2.1: Polynomial regression of varying degrees on data drawn from a linear distribution on the left and a cubic distribution on the right. Models of varying complexity indicated by their basis P^n are fit to the train data and evaluated on the test region. We observe that the higher order solutions 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

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

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 data-matrix $\mathbf{x}$ 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 which gives rise to the problem of overfitting. Overfitting describes a process where the model strongly fits to the noise part of the data and to a lesser degree the signal. With the consequence that while performance on the data the model is fit on increases it rapidly deteriorates outside that region. As much of current research deals with somehow bridging the barriers between different regions of data, or entirely different distributions, reducing the ability for a model to overfit is crucial in most applications.

Finding measures to reduce overfitting has been a goal for machine learning researchers for near on 50 years. The first modern breakthrough was adding a constraint on the coefficients cumulative magnitude. The first successful implementation used a threshold on the squared L_2-norm to limit coefficients. This form of restriction proved hugely beneficial for the simple reason that it restricted the models ability to express all of its complexity. Introduced in 1970 by Hoerl and Kennard (1970) the addition of a norm-constraint to linear regression was dubbed *ridge* regression. Experiments with different norms were carried out in the years following the elegant discovery by Hoerl and Kennard (1970). Perhaps most influential of them is the use of the L_1-norm, first successfully implemented by Tibshirani (1996). As the norms have different geometric expressions the consequence of their addition was evident in the types of solutions generated by their inclusion. The inclusion of an L_1-norm to the linear regression cost-function proved to be challenging as had no closed form solution and thus required iterative methods like gradient descent, described in detail in section 2.7.

In this section we will introduce the different regularizations as additional contributions to the cost-function. As well as a brief exploration of their different geometrical shapes which provides the reasoning for why, and in what manner, these terms reduce the expressibility of the model. We begin with the general L_p norm, which 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. Wile the L_1 term is 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 a ridge regularization term to the squared error cost

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

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. The geometrical interpretation of this is shown 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.

Regularization then reduces the probability of overfitting by limiting the expressed complexity of a model. In the example of polynomial regression lasso regularization forces many of the coefficients to be zero-valued in such a way that it still performs maximally.

add subsection on dropout and batchnorm

2.4.2 Batch Normalization

maybe dropout?

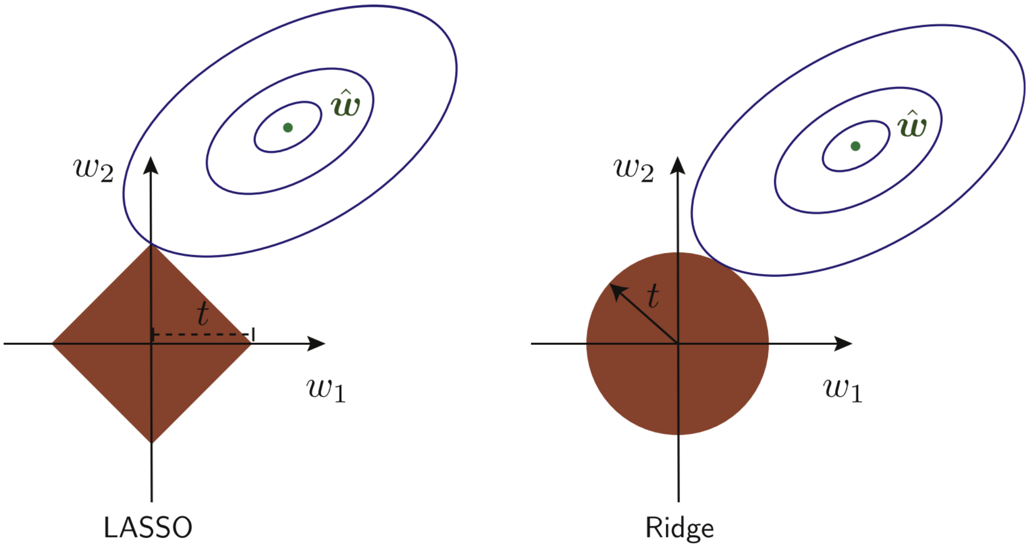


Figure 2.2: Demonstrating the effect of a regularization constraint on a 2-variable optimization. The blue ovals represent the squared error, as it is quadratic in the parameters w_i. And the shaded brown region represents the restriction on the values of w_i, s.t. the only eligible values for the parameters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

The threat of overfitting hangs as a specter over most machine learning applications. Regularization, as discussed in section 2.4.1, outlines the tools researchers use to minimize the risk of overfitting. What remains then is the measurement of the performance of the model, and our confidence in that performance. We've already outlined the most simple tool to achieve this in section 2.2.2; simply split the data in disjoint sets and train on one, measure on the other. As a tool this works best when there is lots of data from which to sample or the purpose of the algorithm is predictive in nature. In this thesis however the purpose is exploratory and labeled data is scarce. Before delving into how to estimate the out-of-sample error we first have to discuss the performance metrics we will use to measure error.

2.5.1 Performance metrics

We measure the performance in the semi-supervised case, outlined in section 2.11, by accuracy of the linear classifier (logistic regression), and the $f1$ score. 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 2.30. Accuracy is bounded in the interval $[0, 1]$

$$\text{accuracy} = \frac{TP + TN}{FN + TN + TP + FP} \tag{2.30}$$

The accuracy as presented in equation 2.30 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 and total $f1$ score, as it allows for comparisons with earlier work on the same data from Kuchera et al. (2019). The $f1$ score is defined in terms of the precision and recall of the prediction. Which are simply defined as true positives weighted by the false positives and negatives. We define recall and precision in equations 2.31 and 2.32 respectively.

$$\text{recall} = \frac{TP}{TP + FP} \tag{2.31}$$

$$\text{precision} = \frac{TP}{TP + FN} \tag{2.32}$$

The $f1$ score is then defined as the harmonic mean of precision and recall for each class. Formally it is given as shown in equation 2.33.

$$f1 = 2\frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} \tag{2.33}$$

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.

2.5.2 Labeled samples

One of the principal challenges with the AT-TPC data discussed in this thesis is that labeled data is challenging to acquire. In the best case scenario it's still computationally intensive to label individual events. In the worst case scenario the current monte carlo based fitting methods might not be able to separate event types of interest from background noise and unknown reactions.

It is then interesting to quantify the effect of the amount of accessible labeled data on a semi-supervised approach as listed in section 2.11. Starting from a random, small, sample of the labeled data we train a classifier on a subset of the labeled data iteratively adding to that subset.

add description of n-labled algo?

2.5.3 Cross validation

To estimate the out-of-sample error one can use simple statistical tools. The premise is that by iteratively selecting what data the model gets to train on and what it doesn't we can compute a less biased estimate of the out of sample error, compared to simply taking the training performance. This idea of iterative sampling is known collectively as cross validation, and the manner in which the sampling is conducted specifies the type of cross validation performed.

write out section on CV

2.5.4 The bias-variance relationship

In statistical learning theory we

write out section on bias-variance

2.6 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.

2.6.1 Hand holding

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

2.6.2 Grid Search

Second on the ladder of naiveté is the exhaustive search of hyperparameter configurations. This in this paradigm one defines a multi dimensional grid of parameters that is evaluated. If one has a set of N magnitudes of the individual parameter sets $s = \{n_i\}_{i=0}^{N}$ with values of the individual parameters γ_k and where $n_i = |\{\gamma_k\}|$ then the complexity of this search is $\mathcal{O}(\prod_{n \in s} n)$. For example in a linear regression we would want to find the optimal value for the learning rate

$\eta = \{\eta_k\}$ and the regularization strength $\lambda = \{\lambda_k\}$ then this search is a double loop as illustrated in algorithm 1. In practice however the grid search is rarely used as the computational complexity scales exponentially with the number and resolution of the parameters. The nested nature of the for loops is also extremely inefficient in the event that the hyperparameters are disconnected i.e. neither co- or contra-variant.

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

```
Data: Arrays of float values $\lambda$, $\eta$
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
```

2.6.3 Random Search

Surprisingly the hyperparameter search method that has proven to be among the most effective is a simple random search. Bergstra et al. (2012) showed empirically the inefficiency of doing grid search and proposed the simple remedy of doing randomized configurations of hyperparameters. The central argument of the paper is elegantly presented in figure 2.3. 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.

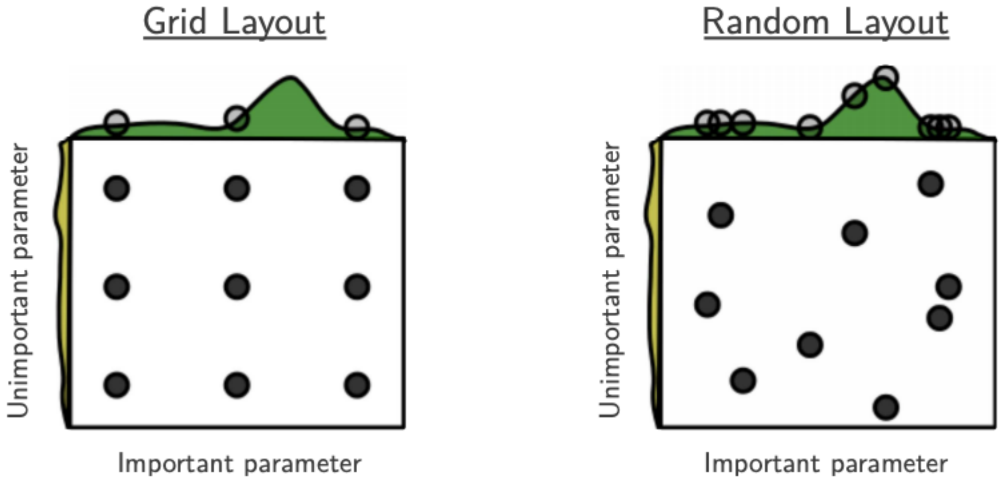


Figure 2.3: Figure showing the inefficiency of grid search. The shaded areas on the axis represent the unknown variation of the cost as a function of that axis-parameter. Since the optimum of the loss with respect to a hyperparameter might be very narrowly peaked a grid search might miss the optimum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

2.7 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 2.43.

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \eta \nabla f(\mathbf{x}_n) \tag{2.34}$$

Equation 2.43 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.

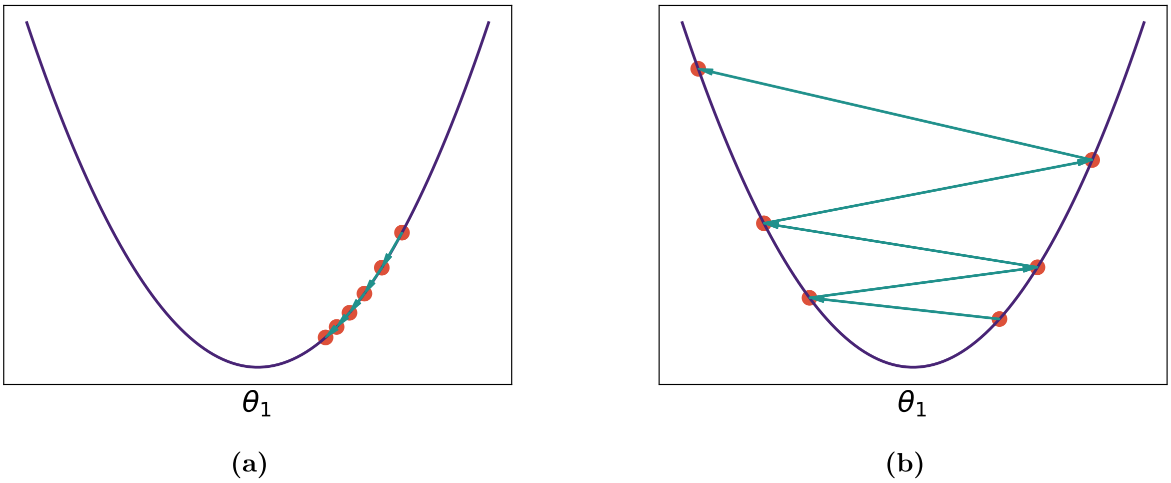


Figure 2.4: Gradient descent on a simple quadratic function showing the effect of too small, **(a)**, and too large, **(b)**, value for the learning rate η

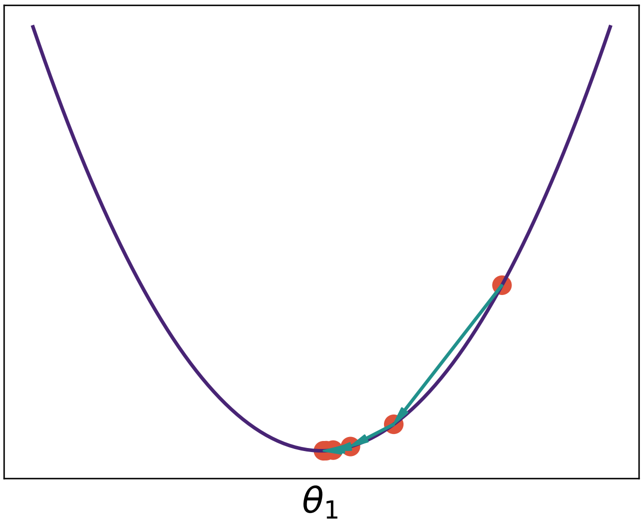


Figure 2.5: Complement to figure 2.4 where we show the effect of a good learning rate on a gradient descent procedure. The gradient descent procedure is performed on a quadratic function.

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.

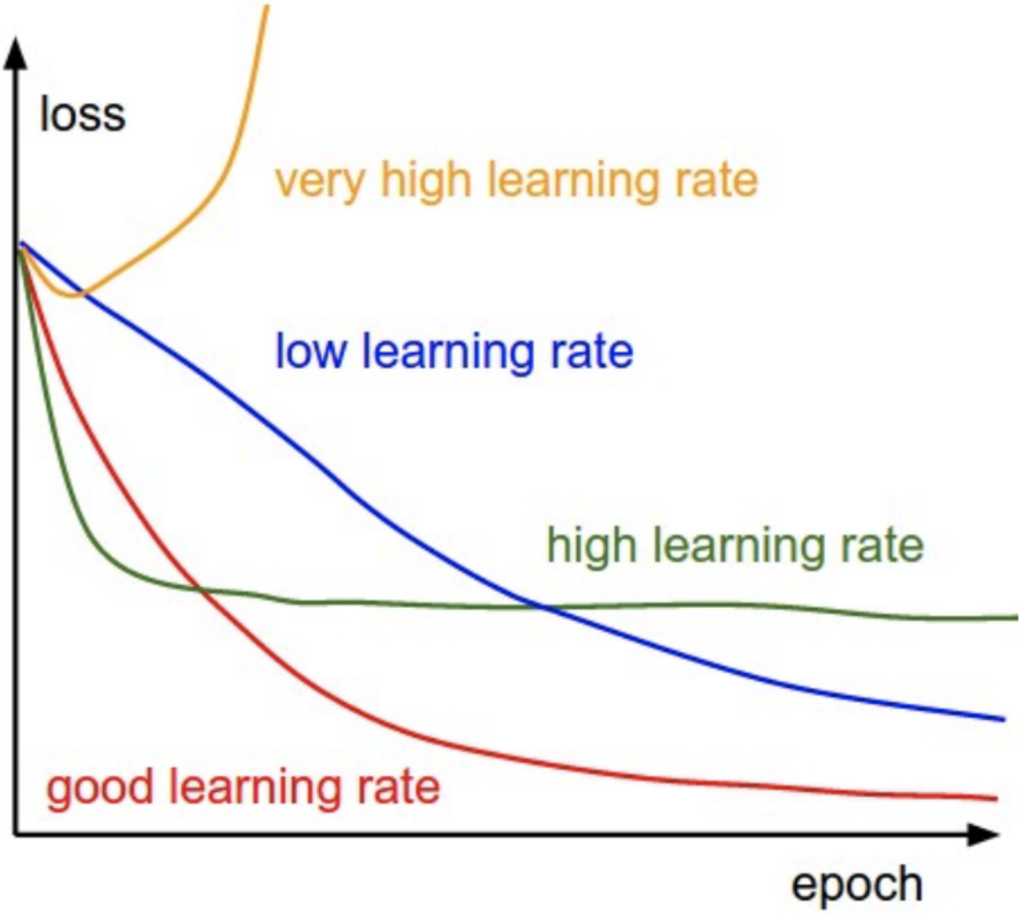


Figure 2.6: A hand-drawn figure showing the impact of the choice of the learning rate parameter on the shape of the loss function. The optimal choice has a nice slowly decaying shape that we will use as a benchmark when tuning the learning rate in our applications. Copied from the cs231 course material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

2.7.1 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 2.35

$$\begin{aligned} \mathbf{v}_n &= \beta \mathbf{v}_{n-1} + (1-\beta)\nabla f(\mathbf{x}_n) \\ \mathbf{x}_{n+1} &= \mathbf{x}_n - \eta \mathbf{v}_n \end{aligned} \tag{2.35}$$

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

2.7.2 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 2.6

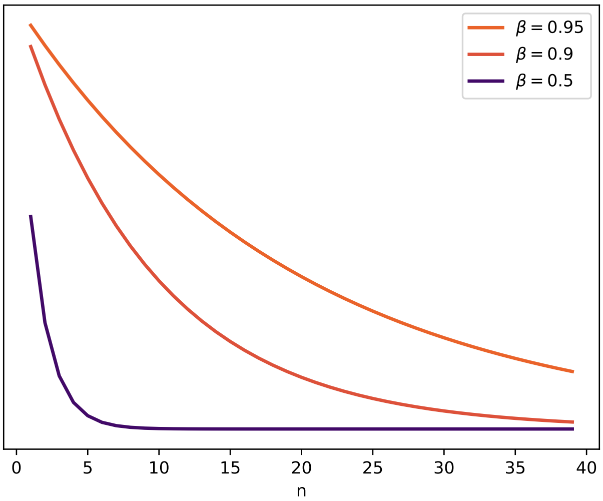
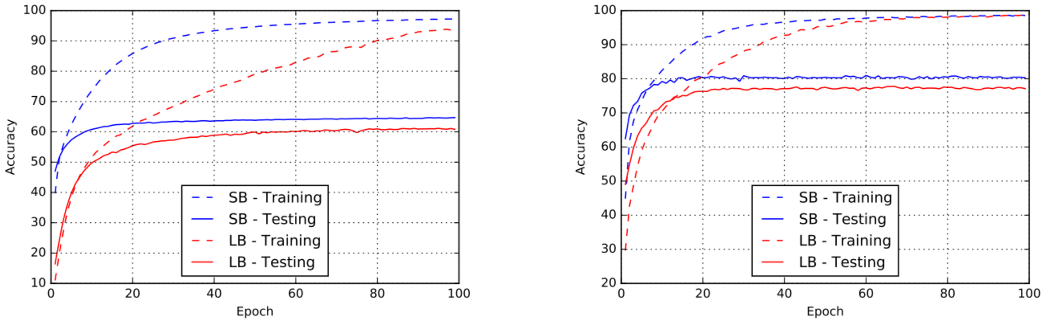


Figure 2.7: A figure illustrating the decay rate of different choices of β. The lines go as β^n and shows that one can quite easily infer how many last steps is included for each choice. A good starting value for the parameter has been empirically found to be $\beta = 0.9$ for many applications. In this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

2.7.3 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 2.43 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 2.6

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla\mathcal{L}(\mathbf{x}_{t,i}) \tag{2.36}$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\nabla\mathcal{L}(\mathbf{x}_{t,i})^2 \tag{2.37}$$

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

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \tag{2.38}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{2.39}$$

Which is then used to update the model parameters in a familiar way, equation 2.40 gives the adam update rule.

$$\mathbf{x}_{n+1} = \mathbf{x}_n - \frac{\eta}{\hat{v}_n + \epsilon}\hat{m}_n \tag{2.40}$$

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.

2.8 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

make proper figure for ann

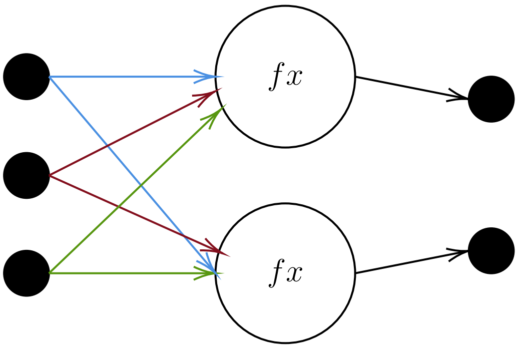


Figure 2.9: An illustration of the graph constructed by two artificial neurons with three input nodes. Colored lines illustrate that each of the input nodes are connected to each of the neurons in a manner we denote as fully-connected.

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 2.41.

$$\hat{y} = a(\langle x|W \rangle)_D \tag{2.41}$$

In equation 2.41 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 2.41 then becomes:

$$\hat{y} = a(\langle x|W \rangle)_D + b \tag{2.42}$$

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 2.43

$$W_{ij} \leftarrow -\eta \frac{\partial \mathcal{L}}{\partial W_{ij}} + W_{ij} \tag{2.43}$$

2.8.1 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 2.43. 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 2.44

$$E = \mathcal{L}(y, \hat{y}, W) = \frac{1}{2} \sum_n \sum_j (y_{nj} - \hat{y}_{nj})^2 \tag{2.44}$$

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 2.44 w.r.t the output, $\hat{y}$, we get

$$\frac{\partial E}{\partial \hat{y}_j} = \hat{y}_j^M - y_j \tag{2.45}$$

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

$$x_j^M = \langle a^{M-1} | W^M \rangle + b_j \tag{2.46}$$

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

$$\hat{y}_j^M = a(x_j^M) \tag{2.47}$$

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

$$\mathrm{ReLU}(x) = \begin{cases} x, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.48}$$

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

$$H(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases} \tag{2.49}$$

We again make explicit that the choice of equations 2.44, 2.48 and 2.49 is not a be-all-end-all solution but chosen for their ubiquitous nature in modern machine learning. We then return to equation 2.45 and manipulate the expression via the chain rule

$$\frac{\partial E}{\partial x_j} = \frac{\partial E}{\partial \hat{y}_j} \frac{\partial \hat{y}_j}{\partial x_j} \tag{2.50}$$

The second derivative of the r.h.s we know from our choice of the activation to be equation 2.49, inserting to evaluate the expression we find

$$\frac{\partial E}{\partial x_j^M} = (\hat{y}_j - y_j) H(x_j^M) \tag{2.51}$$

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

$$\frac{\partial E}{\partial w_{ij}^M} = \frac{\partial E}{\partial x_j^M} \frac{\partial x_j^M}{\partial w_{ij}^M} \tag{2.52}$$

Recall the definition of x_j as the affine transformation defined in equation 2.41. The derivative of the inner product w.r.t the matrix elements is simply the previous layers output. Inserting this derivative of equation 2.46 we have the expression for our derivatives of interest.

$$\frac{\partial E}{\partial w_{ij}} = (\hat{y}_j - y_j)H(x_j)\mathrm{ReLU}(x_i^{M-1}) \tag{2.53}$$

Separately we compute the derivatives of 2.51 in terms of the bias nodes.

$$\frac{\partial E}{\partial b_j} = \frac{\partial E}{\partial x_j}\frac{\partial x_j}{\partial b_j} = (\hat{y}_j - y_j)H(x_j) \cdot 1 \tag{2.54}$$

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

update notation with layer indexation, and algorithm for backprop?

2.8.2 Neural architectures

When creating a neural network on a computer with finite resources a principled consideration must be made on the width and depth of the network. These terms are common in machine learning literature and describes how many nodes per layer and how many layers a network consists of, respectively. A discussion of this consideration is neatly summarized in the work of Lin et al. (2017). The authors provide strong reasoning for prioritizing deep networks over wide ones. They show that one can view many physical systems that generate the data a causal hierarchy (see figure 3 in Lin et al. (2017) for an illustration). This representation of stepwise transformations intuitively lends itself well to representation by a sequences of layers. As each layer contains a transformed, compressed, representation of the data. This bottleneck property of information compression is also what motivates the use of autoencoders, neural networks that compress and reconstruct the input, as a tool when unlabeled data is plentiful and labeled data is scarce.

2.8.3 Activation functions

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

Make activation function plots and write out section on act. functions

2.8.4 Convolutional Neural Networks

There are a couple of glaring problems with neural network layers as they were introduced in section 2.8. 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 2.55 for each filter.

$$O = W - K + 1 \tag{2.55}$$

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 2.55, namely:

$$O = \frac{W - K + 2P}{S} + 1 \tag{2.56}$$

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.

Include mathematical formulation of the convolutional layer

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.

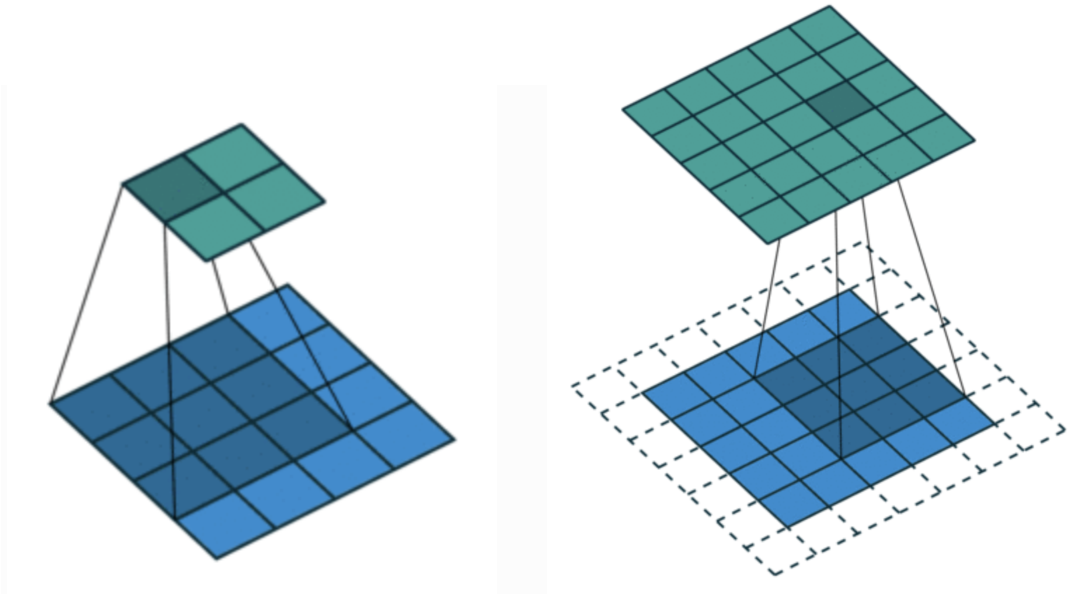
Since then there have been major revolutions in architecture for the state-of-the-art. Inception modules showed that combinations of filters are functionally the same as ones with larger kernels yet maintain fewer parameters (Szegedy et al. (2014)). Residual networks used skip connections, passing the 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].

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] 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 2.46 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)

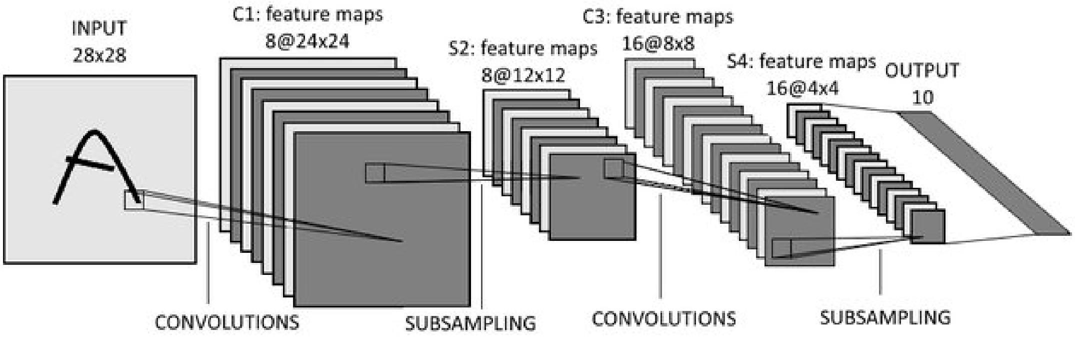


Figure 2.11: The architecture Lecun et al. (1998) used when introducing convolutional layers. Each convolutional layer maintains N kernels with initially randomized weights. These N filters act on the same input, but will extract different features from the input owing to their random initialization. The output from a convolutional layer then has size determined by equation 2.56 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.

2.9 Recurrent Neural Networks

2.9.1 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

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 2.57.

$$\langle [X_t, h_t] | W \rangle + b = h_{t+1} \tag{2.57}$$

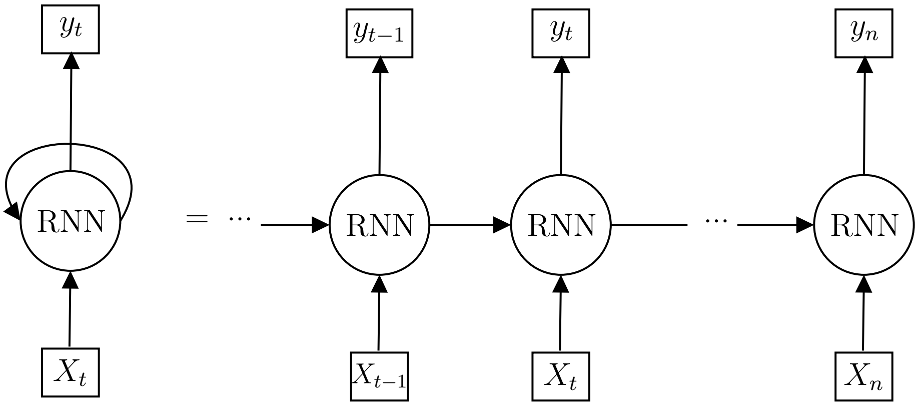


Figure 2.12: A graphical illustration of the RNN cell. The self-connected edge in the left hand side denotes the temporal nature we unroll on the right side. The cell takes as input a state vector and an input vector at time t, and outputs a prediction and the new state vector used for the next prediction. 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 2.8 trained with back-propagation.

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).

2.9.2 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

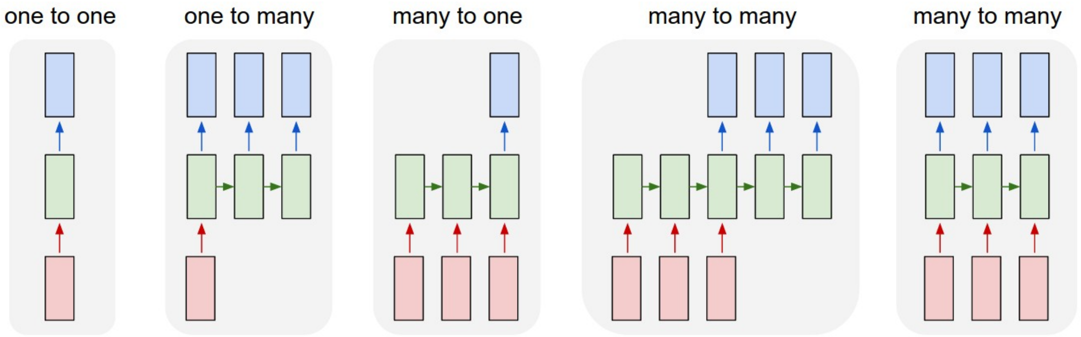


Figure 2.13: The advent of recurrent networks enabled machine learning researchers to both model complex sequential behaviors like understanding patterns in text as well as using sequences to predict a single multivariate outcome and more. The leftmost figure **1)** represents an ordinary neural network,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

2.10 Autoencoders

2.10.1 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

$$\mathcal{O} = \underset{\phi,\psi}{\arg\min}\, ||X - \phi \circ \psi||^2 \tag{2.58}$$

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 2.8. 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

2.10.2 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 2.10.1 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

$$p(z|x,\beta) = \frac{p(z,x|\beta)}{\int_z p(z,x|\beta)} \tag{2.59}$$

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?

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 2.60 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

$$D_{KL}(P||Q) = -\int_{-\infty}^{\infty} p(x) \log \left(\frac{p(x)}{q(x)} \right) dx \tag{2.60}$$

$$= \langle \log \left(\frac{p(x)}{q(x)} \right) \rangle_p \tag{2.61}$$

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

$$D_{KL}(\Psi||Q) = \langle \log \left(\frac{\psi(z|x)}{q(z|x)} \right) \rangle_\psi \tag{2.62}$$

$$= \langle \log \left(\psi(z|x) \right) \rangle_\psi - \langle \log \left(p(x|z)q(z) \right) \rangle_\psi + \log \left(p(x) \right) \tag{2.63}$$

$$= \langle \log \left(\frac{\psi(z|x)}{q(z)} \right) \rangle_\psi - \langle \log \left(p(x|z) \right) \rangle_\psi + \log \left(p(x) \right) \tag{2.64}$$

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

$$\log(p(x)) - D_{KL}(\Psi||Q) = \langle \log \left(p(x|z) \right) \rangle_\psi - \langle \log \left(\frac{\psi(z|x)}{q(z)} \right) \rangle_\psi \tag{2.65}$$

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 2.59. 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

$$ELBO(q) = \langle \log(p(z,x)) \rangle - \langle \log(q(z|x)) \rangle \tag{2.66}$$

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

$$ELBO(q) = \langle \log(p(z)) \rangle + \langle \log(p(x|z)) \rangle - \langle \log(q(z|x)) \rangle \tag{2.67}$$

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

$$\begin{aligned}
\log(p(x)) &= \log \int_z p(x|z)p(z) && (2.68)\\
&= log \int_z p(x|z)p(z)\frac{q(z|x)}{q(z|x)} && (2.69)\\
&= log\langle p(x|z)p(z)/q(z|x)\rangle && (2.70)\\
&\overset{\text{(J)}}{\geq} \langle \log(p(x|z)p(z)/q(z|x))\rangle && (2.71)\\
&\geq \langle \log(p(x|z))\rangle + \langle \log(p(z))\rangle - \langle \log(q(z|x))\rangle && (2.72)
\end{aligned}$$

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 2.65. 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 2.8.1. 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 2.10.

2.10.3 Optimizing the variational autoencoder

From section 2.10 we observe that the optimization is split in two. A reconstructive term that approximates the log evidence which we can train with a squared error or cross entropy cost. Secondly we have a divergence term over the parametrized and theoretical latent distribution. We would like to simplify the second to conserve computational resources. Thankfully this is simple given some assumptions on the target latent distribution. Let the target distribution $p(z|x)$ be a multivariate normal distribution with zero means and a diagonal unit variance matrix, i.e. $p(z|x) \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$. The neural network approximation then follows $q(z|x) \sim \mathcal{N}(\mu, \Sigma)$. The normalized probability density function for the normal Gaussian is defined as

$$p(x) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}}exp(-\frac{1}{2}(x-\mu)^T\Sigma^{-1}(x-\mu)), \tag{2.73}$$

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

$$D_{KL}(p_1||p_2) = \frac{1}{2}\left(\log\frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)\right). \tag{2.74}$$

Which we derive in appendix A. Substituting p_1 and p_2 for the model and target distributions we get

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

or more conveniently

$$D_{KL}(q||p) = \frac{1}{2}\sum_i -\log\sigma_i^2 - 1 + \sigma_i^2 + \mu_i^2. \tag{2.75}$$

We note that equation 2.75 satisfies the equality that the divergence is zero when the target and model distributions are equal. An important feature of the Kullback-Leibler divergence is that it operates point-wise on the probability densities. Which was the topic of the argument presented by Zhao et al. (2017) when proposing alternate measures for regularizing the latent space. The intuitive alternative to a point-wise measurement is comparing the moments of the distribution and minimize their difference.

2.10.4 Regularizing Latent Spaces

As introduced in section 2.10.2 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, which fits the moments of the latent distribution instead of measuring a point-wise divergence. By using any positive definite kernel $k(\cdot,\cdot)$[4] we define the MMD divergence as

$$D_{MMD} = \langle k(z,z')\rangle_{p(z),p(z')} - 2\langle k(z,z')\rangle_{q(z),p(z')} + \langle k(z,z')\rangle_{q(z),q(z')}. \tag{2.76}$$

---

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

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

2.10.5 Deep Recurrent Attentive Writer

One of the central challenges of the ELBO as presented in equation 2.66 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.

$$\hat{\mathbf{x}} = \mathbf{x} - \sigma(\mathbf{c_{t-1}}) \tag{2.77}$$
$$\mathbf{r}_t = \text{read}(\mathbf{x}_t, \hat{\mathbf{x}}_t,) \tag{2.78}$$
$$\mathbf{h}_t^{enc} = \text{RNN}^{enc}(\mathbf{h}_{t-1}^{enc}, [\mathbf{r}_t, \mathbf{h}_{t-1}^{dec}]) \tag{2.79}$$
$$\mathbf{z}_t = \text{latent}(\mathbf{h}_t^{enc}) \tag{2.80}$$
$$\mathbf{h}_t^{dec} = \text{RNN}^{dec}(\mathbf{h}_{t-1}^{dec}, \mathbf{z}_t) \tag{2.81}$$
$$\mathbf{c}_t = \mathbf{c}_{t-1} + \text{write}(\mathbf{h}_t^{dec}) \tag{2.82}$$

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.

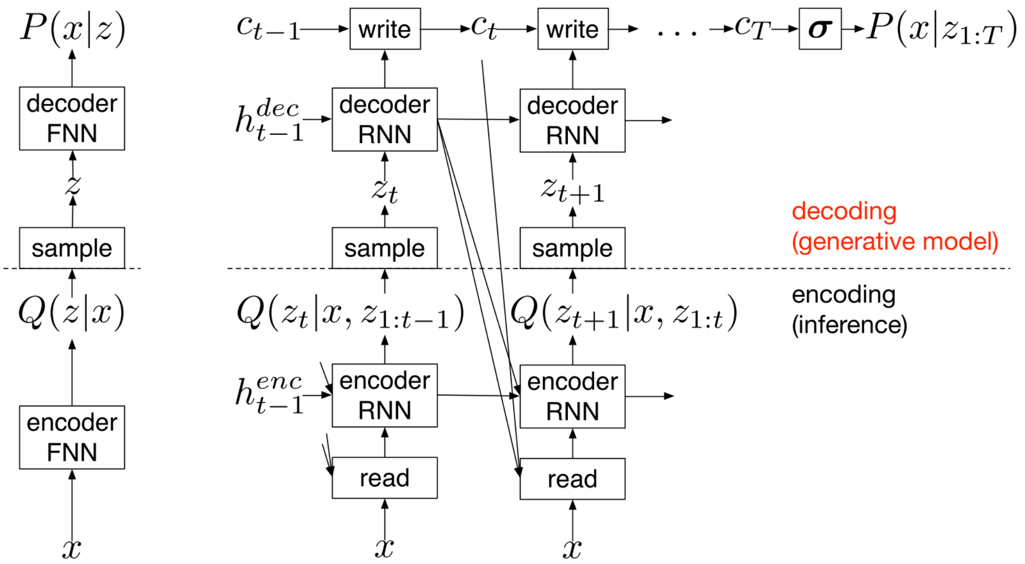

$P(x|z)$
decoder FNN
z
sample
$Q(z|x)$
encoder FNN
x
c_{t-1}
write
c_t
write
$\dots$
c_T
σ
$P(x|z_{1:T})$
h_{t-1}^{dec}
decoder RNN
decoder RNN
z_t
z_{t+1}
sample
sample
decoding
(generative model)
encoding
(inference)
$Q(z_t|x, z_{1:t-1})$
$Q(z_{t+1}|x, z_{1:t})$
h_{t-1}^{enc}
encoder RNN
encoder RNN
read
read
x
x

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

$$\mu_X^i = g_X + (i - N/2 - 0.5)\delta \tag{2.83}$$

$$\mu_Y^j = g_Y + (j - N/2 - 0.5)\delta \tag{2.84}$$

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

$$\tilde{g}_x,\, \tilde{g}_y,\, \log \sigma^2,\, \log \gamma = \text{Dense}(\mathbf{h}_t^{dec}) \tag{2.85}$$

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.

$$g_x = \frac{W+1}{2}(\tilde{g}_x + 1) \tag{2.86}$$

$$g_y = \frac{H+1}{2}(\tilde{g}_y + 1) \tag{2.87}$$

$$\tag{2.88}$$

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 $F_y \in \mathcal{R}^{N \times H}$.

$$F_x[i, w] = \frac{1}{Z_x} e^{-\frac{(w - \mu_x^i)^2}{2\sigma^2}} \tag{2.89}$$

$$F_y[j, h] = \frac{1}{Z_y} e^{-\frac{(h - \mu_y^j)^2}{2\sigma^2}} \tag{2.90}$$

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 2.91

$$\text{read}(\mathbf{x}, \hat{\mathbf{x}}, \mathbf{h}^{dec}_{t-1}) = \gamma[F_y \mathbf{x} F_x^T, F_y \hat{\mathbf{x}} F_x^T] \tag{2.91}$$

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.

$$\text{write}(w_t) = \hat{\gamma} \hat{F}_y^T w_t \hat{F}_x \tag{2.92}$$

Notice the transposition order in equation 2.92 is reversed with respect to the order in equation 2.91.

Latent samples and loss

write out latent sample/loss subsects

2.10.6 Deep Clustering

2.11 Neural architectures

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

Labeled data is, however, both require a high time investment and possibly high amounts of computing power to separate. Having access to a fully unsupervised method of separating classes of events can then be hugely beneficial to researchers. In the event where we don't have access to labeled data we have to discover emergent clusterings in the data without knowledge about the class distribution. In this case we still use an autoencoder model, but different demands have to be made of the latent space. This is the *clustering* scheme.

2.11.1 Classification

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

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

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

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

We will determine the best autoencoder architecture for each dataset listed in section 3.3. The best autoencoder is measured by performance in identifying separate classes by the logistic regression model on a test-set of data.

2.11.2 Clustering

To attempt clustering of event data we will follow the pipeline as outlined in Guo et al. (2017). The modeling has two distinct steps, pre-training of the convolutional autoencoder and training with regularization towards the pseudo-labels.

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

2. Compute latent representations of the full dataset

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

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

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

2.11.3 Pre-trained networks

Following the precedent of Kuchera et al. (2019) we will consider representations of our events through the lens of a pre-trained network. In the Machine Learning community it is not uncommon to publish packaged models with fitted parameters from image recognition contests. These models are trained on millions of images and classify between hundreds of distinct classes. In their work Kuchera et al. (2019) use the VGG16 architecture trained on imagenet to classify AT-TPC events, in this thesis we will build on the understanding of using this pre trained networks in event classification by using VGG16 as an element in the end-to-end training of autoencoders. The VGG16 network is one of six analogous networks proposed by Simonyan and Zisserman (2015), they were runners up in the ISLVRC(ImageNet large scale visual recognition competition) of 2014 (Russakovsky et al. (2015)). The network architectures are fairly simple, for VGG16 there are sixteen layers in the network. The first thirteen of which are convolutional layers with exclusively 3×3 kernels. The choice of the kernel size is based on the fact that a stacked 3×3 is equivalent to larger kernels in terms of the receptive field of the output. Three 3×3 kernels with stride 1 have a 7×7 receptive field, but the larger kernel has 81% more parameters and only one non-linearity (Simonyan and Zisserman (2015)). Stacking the smaller kernels then both contribute computationally, as a regularization factor because of the reduce number of parameters and increased discriminatory power because of the added non-linearities. The full architecture is detailed in appendix B.1.

A pre-trained network can be included in the architectures in three distinct configurations. The pre-trained network can either:

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

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

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

As a baseline for the modeling pipelines we consider the VGG16 model without updating weights. Which is to say that the performance of the methods proposed in this thesis will be measured against the performance of simply passing the events through the pre-trained VGG network and then to a logistic regression classifier for the *classification* scheme. And to a K-means algorithm for the *clustering* scheme.

clean up section on model architectures

Chapter 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

3.2.1 A note on nuclear physics

In this thesis we primarily concern ourselves with analysis methods that are agnostic to the physics in the system. One can argue that this is both a strength of the methodology and a weakness. As a consequence the discussion of the physical system will brief. For a more in-depth treatment of the physics see Bradt (2017).

With that in mind we turn to the central pursuits of nuclear physics: understanding the structure of the nucleus. Nuclides are described in terms of the number of protons, Z, and neutrons N and their total mass number $A = Z + N$. They are further categorized by equal components; nuclides with an equal number of protons are called *isotopes*, equal number of neutrons *isotones* and with the same mass *isobars*. The first modern fully-formed theory of nuclear structure, the nuclear shell model, was focused around the observation that certain isotopes and isotones were much more stable than others. As it happened these stable nuclei were regularly spaced around certain numbers of constituent protons and neutrons. These numbers are called magic numbers and describe nuclides that are much more tightly bound than the next number, as a consequence they are very stable and exhibit long half-lives. These magic numbers are: 2, 8, 20, 28, 50, 82, 126. Some nuclides are even doubly-magic, which is to

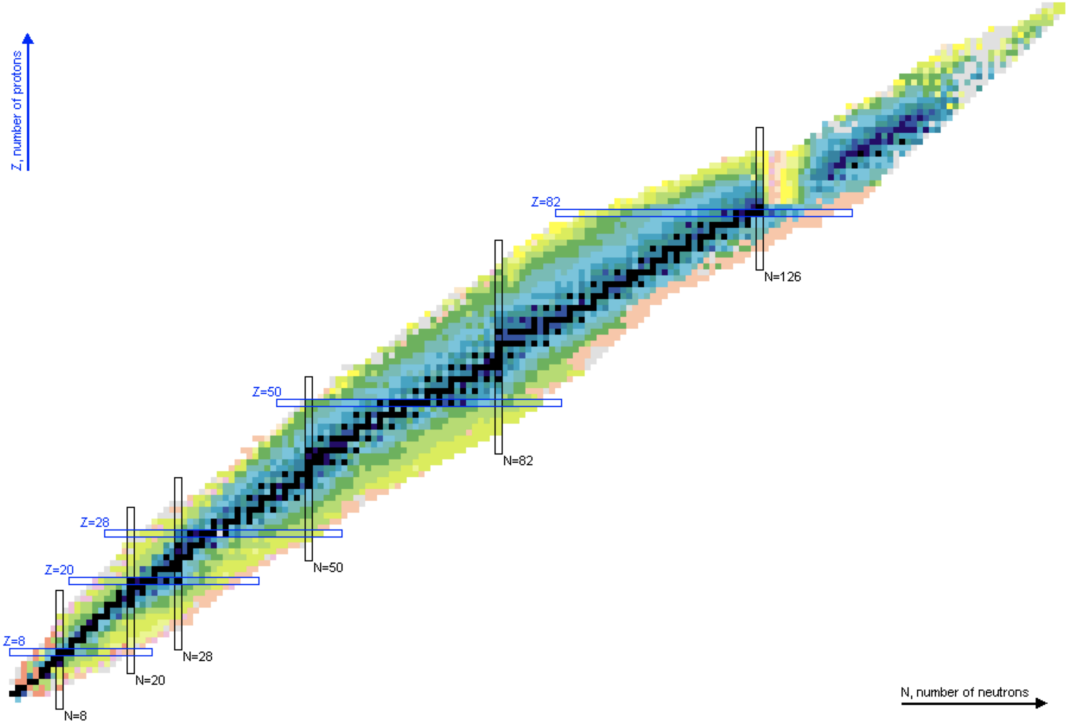


Figure 3.1: Chart of the known nuclides. The number of protons are given along the vertical axis, and neutrons along the horizontal. The color indicates the half-life of the nucleus with darker colors representing longer times. Lines of isotones and isotopes along the magic numbers are indicated by rectangles in the figure. Retrieved from Sonzogni (2019)

say both Z and N are magic numbers. One area of active research is around the $N = 28$ isotones. Predictions from the nuclear shell model indicates that these isotones should have an approximately spherical structure. This has been disproved experimentally as deformities appear when removing protons from the spherical nucleus ^{48}Ca. Which brings us to ^{46}Ar which lies in a region between the spherical ^{48}Ca and the lighter isotones that are known to be deformed. The location of ^{46}Ar makes it an object of some academic interest, and served as the commissioning of the AT-TPC (Active Target Time Projection Chamber) at the NSCL.

Isobaric analogues

3.2.2 AT-TPC details

1. Introduce the TPC and AT-TPC

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

3. Introduce and discuss the physics of the AT-TPC experiments
4. Discuss the need for ML in event categorization in the AT-TPC experiments

3.3 Data

In this thesis we will work with data from the $^{46}Ar(p, p')$ experiment conducted at the national superconducting cyclotron laboratory (NSCL) located on the Michigan state university campus. Both data produced with simulation tools and data recorded from the active target time projection chamber (AT-TPC). For the experimental data we use data collected from a single run of the experiment.

3.3.1 Simulated ^{46}Ar events

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

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

Added part from previous results, needs molding

3.3.2 Full ^{46}Ar events

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

figure of 3D simulated track and 2D representation

	Simulated	Real	Filtered
Total	8000	51891	49169
Labeled	2400	1774	1582

Table 3.1: Description of the data used for analysis. In principle we can simulated infinite data, but it is both quite simple and not very interesting outside a case for a proof-of-concept

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

describe the physics in a bit more detail boy

3.3.3 Filtered $^{4}6Ar$ events

As we saw in the previous section the events pick up

write filtered section

add plots of events in 2d and 3d

add table with data descriptions. N samples, N labelled

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

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

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

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

```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 2.8. 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 2.8. 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.

```
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

4.3 Deep learning algorithms

All the models in this thesis are implemented in the python programming language using the tensorflow api for deep learning. All the models are open source and can be found in the github repository https://github.com/ATTPC/VAE-event-classification. In this section we will be detailing the general framework that the models have been built on. The structure is straightforward with a model class implementing shared functions and usage between models. Two subclasses are implemented, one for the sequential DRAW model (discussed in section 2.10.5) and one for the non-sequential convolutional autoencoder 2.10. 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 2.10.4) to be used for the experiment. As well as whether or not to restore the weights of a previous run from a directory, this directory is supplied to the train method of LatentModel class.

After initialization and before training the subclasses of LatentModel needs to construct the computational graph defining the forward pass. As well as the pertinent operations, these include the loss components, the latent space sample and the backwards gradient descent pass. This is done via a wrapper function compile_model defined in LatentModel that takes two dictionaries for the graph and loss configuration. They are subclass specific and will be elaborated on later in sections 4.3.1 and 4.3.2. 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 2.7 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 2.10.6. 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 2.10.6

SKLEARN CITATION

4.3.1 Convolutional Autoencoder

The convolutional autoencoder class ConVae is implemented as a subclass of LatentModel. It implements the construction of the computational graph and the compute operations needed for the available losses. The graph is manipulated with respect to a supplied configuration dictionary. Which includes options for additional regularization terms like batch-normalization and instructs the class on what losses to compile the model with.

Computational graph

The private[3] function which enacts the computational graph is _ModelGraph. It accepts arguments for the strength and type of regularization on the kernel and bias parameters as well as the activation function to be used for the internal representations and the projection to output space.

Inside the method the placeholder variable ConVae.x is defined. The placeholder defines the entry point of the forward pass and is where tensorflow allocates the batched data in the training operation. Depending on whether the model is instructed to use the VGG16 representation of the data or a specified encoder structure it applies dense weight transformations with non-linearities or runs a series of convolutional layers, respectively. Each convolutional layer is specified with a kernel size, a certain number of filters and the striding of the convolutions. We also use a trick from Guo et al. (2017) to ensure that the padding is chosen such that the reconstruction is unambiguous.

The padding is set to preserve the input dimensionality and is only reduced in dimensionality with striding or max-pooling. Depending on whether the output width(height) is an integer multiple of 2^n, where n is the number of layers, the last convolution is adjusted to have no zero-padding if this is the case.

After each layer the specified non-linearity is applied. This is one of the sigmoid activations (logistic sigmoid or hyperbolic tangent) or the rectified linear unit family of activations[4]. If the model configuration specifies to use batch normalization this is applied before sigmoid functions and after rectified units. The reason for different points of application relates to the challenges of the respective activation families; sigmoids' saturate and so the input should be scaled and rectified units explode so the output is scaled. The output from the convolutional layers is then an object with dimensions $\mathrm{h} = (\mathrm{o},\ \mathrm{o},\ \mathrm{f})$ where f is the number of filters in the last layer and $\mathrm{o} = \frac{H}{2^n}$ with n denoting the number of layers with stride 2 or the count of MaxPool layers and H the input image size.

---

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

The tensor output from the convolutional layers is then transformed to the latent space with either a simple dense transformation, e.g. $\mathrm{z} = \mathrm{Dense(flatten(h))}$. Or if a variational loss is specified a pair mean and standard deviation tensors is constructed with dense transformations from h. Using the re-parametrization trick shown by Kingma and Welling (2013) a sample is generated by $\mathrm{z} = \mathrm{mu} + \mathrm{sigma}{*}\mathrm{epsilon}$ where epsilon is a stochastic tensor from the multivariate uniform normal distribution, $\mathcal{N}(0, 1)$. The mean and standard deviation tensors are stored to be used in the computation of the loss. The latent sample is also stored for later retrieval or the computation of non-variational costs.

After a sample z is drawn the reconstruction is computed with either a mirrored decoder for the naive autoencoder structure or for a VGG16 representation of the data a reconstruction is computed based on a specified decoder structure. The VGG16 representation has the same call structure, but with a boolean flag to the model use_vgg that indicates that the configuration is explicitly for the decoder.

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

Computing losses

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

Each of these losses acts pixel-wise on the output and target images. The reconstruction loss is then stored as a class attribute ConVae.Lx which is monitored by the TensorFlow module TensorBoard for easy monitoring during training. Depending on the configuration the model then compiles a loss over the latent space. For a variational autoencoder the loss is a Kullback-Leibler divergence over the latent distribution and a multivariate normal distribution with zero mean and unit variance. This has a closed form solution given a tensor representing the mean and standard deviation which we derived in equation 2.75. This equation is relatively straightforward to implement as it just implies a sum over the mean and standard deviation tensor. The form of equation 2.75 also makes clear why we parametrize the standard deviation and not the variance directly as the exponentiation ensures positivity of the variance.

---

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

Alternatively to a Kullback-Leibler divergence the latent space may be regularized in the style proposed by Zhao et al. (2017). Which measures the We use the radial basis function kernel to compute the maximum mean discrepancy divergence term.

4.3.2 DRAW

4.4 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

Chapter 5

Experimental setup and design

The experiments were conducted using the AI-Hub computational resource at the university of Oslo. This resource consists of three machines with four RTX 2080 Nvidia graphics cards each. These cards have about 10GB of memory available to allocate model weights. All experiments described in this section were all computed using this hardware. In this section we lay out the results obtained on the three segments of data: simulated, filtered, and full event-datasets. These are described in greater detail in section 3.3. We explore the models proposed in section 2.11 on two disparate tasks, one of semi-supervised classification and one of clustering. For each task we evaluate the performance on each of the aforementioned datasets using appropriate metrics. The primary objective for the ^{46}Ar experiment was to identify resonant proton scattering events, and so the model s will be evaluated on their ability to separate proton events from the others that occur in the dataset. The broader picture for the application of these models are however applications to experiments where there might be multiple event-types of interest. And so we measure individual class performance wherever appropriate.

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

For reference the models are described in terms of their hyperparameters in table 5.1 for the convolutional autoencoder and table 5.2 for the DRAW-analogues. The classifier is trained on a subset of the labeled set and evaluated on the remainder to estimate the OOS error. The best configuration will then be re-trained and we evaluate this model with k-fold cross validation as outlined in section 2.5.

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 2.8.3
Latent type	Multinomial	One of the latent space regularization techniques (KLD, MMD, clustering loss)
Latent dimension	Integer	The dimensionality of the latent space
β	Logarithmic int	Weighting parameter for the latent term
Batchnorm	Binary	Whether to use batch-normalization in each layer
Optimizer parameters:		
η	Logarithmic float	Learning rate, described in 2.7
β_1	Linear float	Momentum parameter, described in 2.7.1
β_2	Linear float	Second moment momentum parameter. Described in 2.7.3

Table 5.1: 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 2.7
β_1	Linear float	Momentum parameter, described in 2.7.1
β_2	Linear float	Second moment momentum parameter. Described in 2.7.3

Table 5.2: Hyperparameters for the draw algorithm as outlined in section 2.10.5. The implementation of the convolutional read and write functions is a novel contribution to the DRAW algorithm. We investigate which read/write paradigm is most useful for classification and clustering. Additionally as a measure ensuring the comparability of latent sample we fix the δ parameter determining the glimpse size. The effect of δ is explored in detail in the paper by Gregor et al. (2015) and in the earlier section 2.10.5.

add image from each experiment to results subsection header

Chapter 6

Classification results

The training procedure for classification using a semi-supervised regime as the one we'll apply necessitates the same strict separation of labeled data for the classification step as when considering ordinary classification tasks. Details on the modeling pipeline can be found in section 2.2. All models excepting the baseline VGG model were tuned with the RandomSearch architecture which searches in a semi structured way over all the parameters given in table 5.1. The top 40 runs from each model is listed in appendix C.1with their corresponding proton $f1$ score. We optimize the classification performance for each dataset listed in section 3.3. Additionally plots of $f1$ performance as a function of the number of labeled samples is included. As a benchmark we start by measuring the performance using just the pre-trained VGG16 representation of the labeled data of each dataset. The two proposed representation algorithms are then presented with results for each dataset for comparison.

correct appendix link and add performance tables?

6.1 VGG16

As outlined in section 2.11 the pre-trained VGG16 network will serve as the base-line comparison for this work. We chose VGG16 as it has a tried and true performance on labeled AT-TPC data from the ^{46}Ar experiment.(Kuchera et al. (2019)). For each labeled dataset listed in section 3.3 a logistic regression model was fit to the respective VGG16-representation. To estimate the variability in the result a K-fold cross validation approach was taken, with $K = 5$. We report test-$f1$ scores for each class and average for the classification. The results are listed in table 6.1

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

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

Table 6.1: Logistic regression classification results using the VGG16 representation of the labeled data listed in section 3.3. The error is given as the standard deviation in the $f1$ score over the $K = 5$ folds of cross validation.

and so a variability estimate is computed by running this procedure $N = 100$ times. We report the mean and standard deviation for each dataset. The result of this analysis is shown in figure 6.1

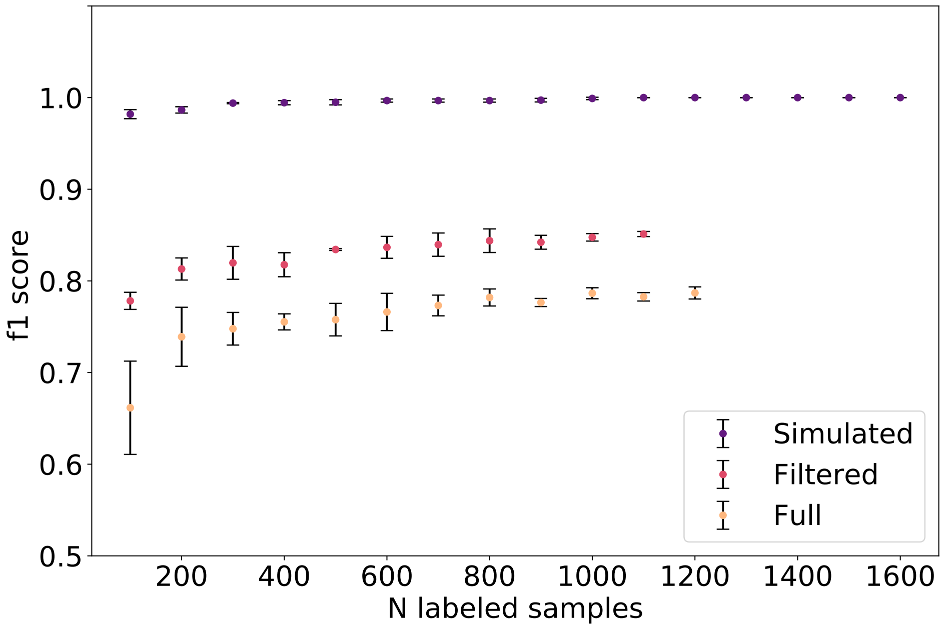


Figure 6.1: VGG16 performance on increasing subsets of labeled data. The error-bars represent the $\pm 1\sigma$ interval. The y axis $f1$ score is computed as the unweighted average of the sample classes.

6.2 Convolutional Autoencoder

To test the hypothesis that classification can be improved by using unsupervised methods to estimate the data distribution is investigated by using a convolutional autoencoder trained end-to-end on the data distribution, and then using the latent representation as input to a logistic regression classifier on the subset of data that has labels. This pipeline is outlined in section 2.11, and the data are described in section 3.3. The convolutional autoencoder has three configurations that we report results from.

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

(Ar1): Using the VGG16 network to compute a representation of the data which is compressed by one or more dense layers and finally reconstructed to the original image by a naively constructed decoder.

(Ar2): Using the VGG16 network as the encoder, adjusting it's weights by the reconstruction loss with a naively constructed decoder.

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

add plot with reconst/loss vs f1 scores

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

add architecture tables, note on latent divergence?

For the best models found by random search we re-compute the performance with $K = 5$ fold cross validation on the logistic regression classifier. We begin with the model using no information from the VGG16 benchmark, i.e. configuration (Ar0). It shows strong performance on the classification task for all datasets. The results are listed in table 6.2

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

add table of architectures

Each of the configurations found by the random search was then evaluated with $K = 5$ fold cross validation to produce estimates of the $f1$ score, listed in table 6.3

add n-labeled plots

	Proton	Carbon	Other	All
Simulated	0.976 $\pm 6.701 \times 10^{-3}$	0.975 $\pm 6.888 \times 10^{-3}$	N/A	0.976 $\pm 6.794 \times 10^{-3}$
Filtered	0.849 $\pm 2.977 \times 10^{-2}$	0.583 $\pm 3.969 \times 10^{-2}$	0.896 $\pm 1.864 \times 10^{-2}$	0.776 $\pm 2.937 \times 10^{-2}$
Full	0.739 $\pm 4.969 \times 10^{-2}$	0.501 $\pm 7.095 \times 10^{-2}$	0.875 $\pm 2.634 \times 10^{-2}$	0.705 $\pm 4.899 \times 10^{-2}$

Table 6.2: Logistic regression classification results using the (Ar0) architecture. The standard error is reported from a $K = 5$ fold cross validation of the logistic regression classifier.

	Proton	Carbon	Other	All
Simulated	0.998 $\pm 1.552 \times 10^{-3}$	0.998 $\pm 1.566 \times 10^{-3}$	N/A	0.998 $\pm 1.559 \times 10^{-3}$
Filtered	0.898 $\pm 3.644 \times 10^{-2}$	0.605 $\pm 3.078 \times 10^{-2}$	0.882 $\pm 2.808 \times 10^{-2}$	0.795 $\pm 3.176 \times 10^{-2}$
Full	0.858 $\pm 2.602 \times 10^{-2}$	0.676 $\pm 4.631 \times 10^{-2}$	0.893 $\pm 1.820 \times 10^{-2}$	0.809 $\pm 3.018 \times 10^{-2}$

Table 6.3: Logistic regression classification results using the (Ar1) architecture. The standard error is reported from a $K = 5$ fold cross validation of the logistic regression classifier.

6.3 DRAW

Chapter 7

Clustering of AT-TPC events

7.1 Convolutional Autoencoder results

7.2 DRAW results

Part IV

Discussion and Conclusion

7.3 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).

7.3.1 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 **??** 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 7.2 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.

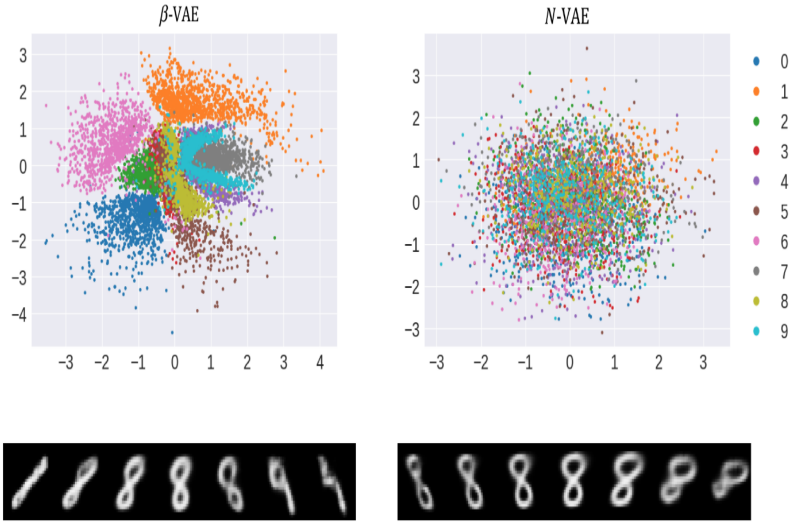


Figure 7.1: Demonstrating the difference of capturing class and feature information in the latent space. On the left the β-VAE pushes the autoencoder to a representation favoring encoded class information in the latent space. The spaces between the class blobs makes for a poor generative algorithm, but for the purpose of classification or even clustering this is strongly preferable. On the right the natural clustering of feature information is demonstrated by the convincingly isotropic nature of the latent space distribution. The subplots under the latent distributions demonstrate reconstructions of a traversal along a latent axis, clearly showing the difference between feature and class information. Figure copied from Antorán and Vivolab (2019)

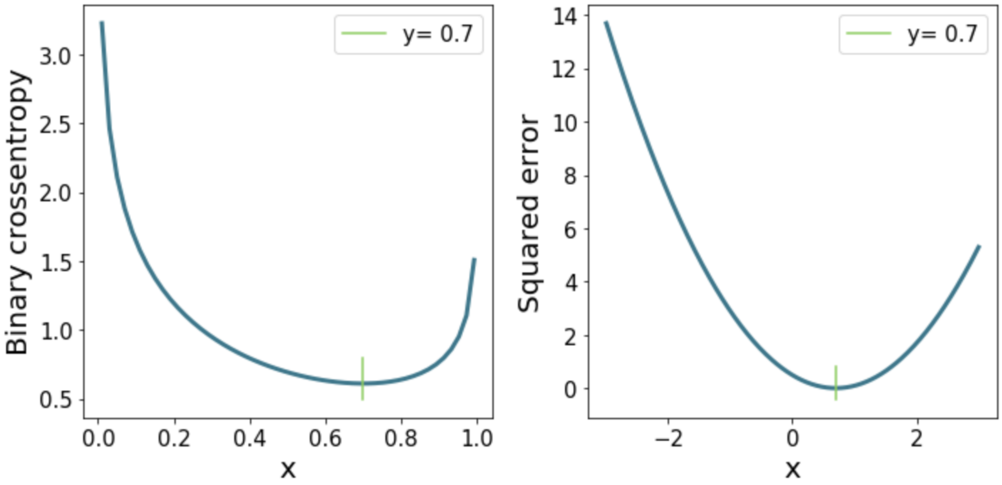


Figure 7.2: Illustrating the difference between the function shapes of binary cross-entropy and the mean squared error for the same target value $y = 0.7$. We observe that though the two functions have the same minimum the landscape surrounding it is very different. Perhaps most telling is the asymmetry 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

Kullback-Leibler divergence of of Gaussian distributions

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

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

And described in full by the mean vector μ and covariance matrix Σ. The Kullback-Leibler divergence between two multivariate Gaussians is then given as

$$\begin{aligned} D_{KL}(p_1||p_2) &= \langle \log p_1 - \log p_2 \rangle_{p_1} \\ &= \langle \frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} + \frac{1}{2}(-(x-\mu_1)^T\Sigma_1^{-1}(x-\mu_1) + (x-\mu_2)^T\Sigma_2^{-1}(x-\mu_2))\rangle. \end{aligned}$$

We abuse the fact that the exponential factors represent an inner-product to apply a trace operator to manipulate the sequence of operations given the trace operators invariance under cyclical permutations i.e. $tr(X^TBX) = tr(BX^TX)$. Furthermore we use the fact that the trace is a linear operator and so commutes with the expectation i.e. $E(tr(BX^TX)) = tr(B\,E(X^TX))$. We also move the logarithm of the covariance determinants outside of the expectations,

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

Conveniently the covariance matrix is defined by the expectation

$$\Sigma := \langle (x-\mu)^T(x-\mu)\rangle, \tag{A.2}$$

giving an evident simplification. For the terms originating from p_2 we will use the definitions of the covariance matrix and the mean vector, i.e. $\mu = \langle x\rangle$ and

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

Returning to the Kullback-Leibler divergence we then have

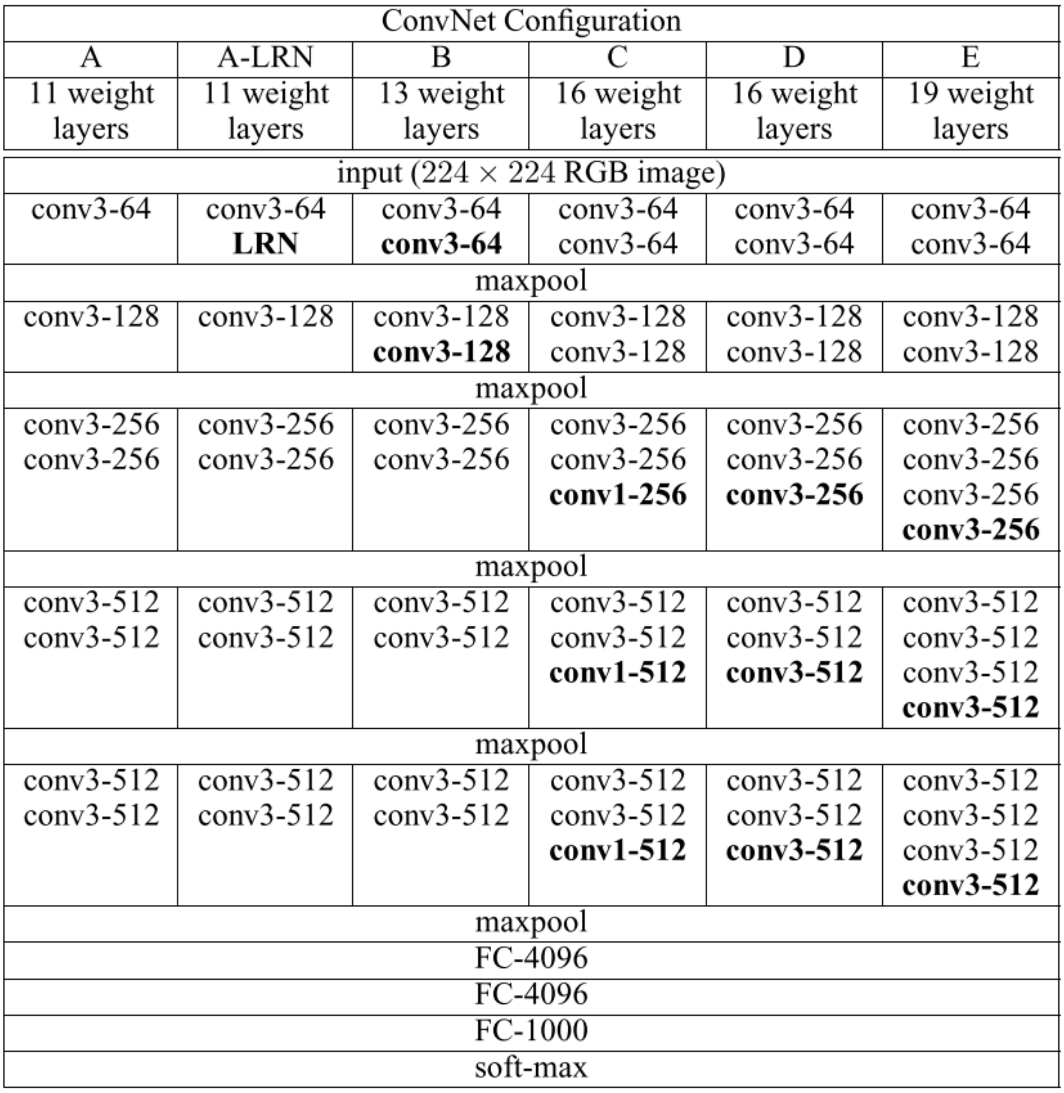
$$\begin{aligned} D_{KL}(p_1||p_2) &= \frac{1}{2}\log\frac{|\Sigma_2|}{|\Sigma_1|} + \frac{1}{2}(-tr(\Sigma_1^{-1}\Sigma_1) + tr(\Sigma_2^{-1}\langle x^T x - 2x\mu_2^T + \mu_2\mu_2^T\rangle)) \\ &= \frac{1}{2}\left(\log\frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}(\Sigma_1 + \mu_1\mu_1^T - 2\mu_1\mu_2^T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

Grouping terms then gives us the final expression for the Kullback-Leibler divergence of two multivariate Gaussians

$$D_{KL}(p_1||p_2) = \frac{1}{2}\left(\log\frac{|\Sigma_2|}{|\Sigma_1|} - n + tr(\Sigma_2^{-1}\Sigma_1) + (\mu_2-\mu_1)^T\Sigma_2^{-1}(\mu_2-\mu_1)\right). \tag{A.3}$$

Appendix B

Neural network architectures



ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

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

Appendix C

Results on simulated data

proton f1-score	N parameters	largest kernel	N layers	latent dimension	latent loss	reconst -ruction loss	activation function	batch -norm	β	β_1	η
0.99	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	reconstruction 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 C.1: Randomsearch runs for the convolutional autoencoder sorted by the resulting proton f1 score of the logistic regression classifier using the latent samples to classify event-types. We note the high occurrence of the maximum mean discrepancy with the higher performing classifications. We also note that simply no latent loss is able to achieve near perfect proton f1 scores.

Chapter 8

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.

Bradt, J., Bazin, D., Abu-Nimeh, F., Ahn, T., Ayyad, Y., Beceiro Novo, S., Carpenter, L., Cortesi, M., Kuchera, M., Lynch, W., Mittig, W., Rost, S., Watwood, N., and Yurkon, J. (2017). Commissioning of the Active-Target Time Projection Chamber. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 875:65–79.

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

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

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

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

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

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

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

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

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

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

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. P., Ramanujan, R., Taylor, J. Z., Strauss, R. R., Bazin, D., Bradt, J., and Chen, R. (2019). Machine learning methods for track classification in the AT-TPC. *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 940:156–167.

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

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

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

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

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

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

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

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

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

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

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

Sonzogni, A. (2019). NuDat 2.7.

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.