GENERIC MASTER THESIS TITLE

by

Geir Tore Ulvik

THESIS

for the degree of

MASTER OF SCIENCE

Faculty of Mathematics and Natural Sciences
University of Oslo

November 2020

Abstract

This is the abstract text.

To someone

This is a dedication to my cats.

Acknowledgements

I acknowledge my acknowledgements.

Contents

1 Introduction **1**

2 Theory **3**
2.1 Linear Regression 4
2.2 Over- and underfitting 5
2.3 The Bias-Variance Tradeoff 6
2.4 Regularization 7
2.5 Logistic Regression 7
2.6 Gradient Descent 7
2.6.1 Stochastic Gradient Descent 7
2.7 Neural Networks 8
2.7.1 Artificial Neurons 8
2.7.2 The Feed Forward Neural Network 9
2.7.3 Activation Functions 12
2.7.4 Types of neural networks 13
2.7.5 Backpropagation 13
2.8 Assessing the Performance of Models 13
2.8.1 Imbalanced Data in Classification 13
2.8.2 Confusion Matrix 14
2.8.3 Receiver Operating Characteristic 14
2.9 Nuclear Science 15
2.9.1 Shell Structure 15

3 Method **17**
3.1 Pretrained network 18

4 Results **19**
4.1 Classification 20
4.1.1 Simulated data 20
4.1.2 Experimental data 20
4.2 Regression 22
4.2.1 Simulated data 22

5 Discussion **25**

6 Conclusion **27**

Chapter 1

Introduction

Start your chapter by writing something smart. Then go get coffee.

Chapter 2

Theory

2.1 Linear Regression

Suppose you have a data set $\mathcal{L}$ consisting of the data $\mathbf{X}_\mathcal{L} = \{(y_i, x_i), i = 0 \dots n-1\}$. Each point is associated with a scalar target y_i, and a vector $\boldsymbol{x}$ containing values for p input features. Assuming the target variable y_i is linear in the inputs, it can be written as a linear function of the features, given by

$$y_i = w_0 x_{i,0} + w_1 x_{i,1} + ... + w_{p-1} x_{i,p-1} + \epsilon_i, \tag{2.1}$$

where $\boldsymbol{w} = (w_0, w_1, \dots, w_{p-1})^T$ is a vector of length p containing unknown values, and ϵ are the errors in our estimate. This gives us a system of linear equations, which can be written in matrix form as

$$\boldsymbol{y} = \boldsymbol{X}\boldsymbol{w} + \epsilon, \tag{2.2}$$

where

$$\mathbf{X} = \begin{bmatrix} x_{0,0} & x_{0,1} & x_{0,2} & \dots & x_{0,p-1} \\ x_{1,0} & x_{1,1} & x_{1,2} & \dots & x_{1,p-1} \\ x_{2,0} & x_{2,1} & x_{2,2} & \dots & x_{2,p-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{n-1,0} & x_{n-1,1} & x_{n-1,2} & \dots & x_{n-1,p-1} \end{bmatrix} \tag{2.3}$$

The unknown values $\boldsymbol{w}$ are commonly referred to as *weights* in machine learning literature. To find the best possible weights $\boldsymbol{w}$ we want a suitable quantity to optimize - a **cost function**, $\mathcal{C}$ (also referred to as an **objective function**). An example of such a function is the squared error - or the Euclidian vector norm, defined as

$$L_2(\boldsymbol{x}) = ||\boldsymbol{x}||_2 = \left(\sum x_i^2\right)^{\frac{1}{2}}. \tag{2.4}$$

From this we define the cost function

$$\mathcal{C} = ||\hat{\boldsymbol{y}} - \boldsymbol{y}||_2^2. \tag{2.5}$$

In machine learning, it is most common to cast the optimization as a minimization problem ("minimize the cost"). Our task is then to find an approximation

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w} \tag{2.6}$$

which minimizes this cost function. To find the minimum we need a differentiation. To simplify that process, we rewrite the cost function on matrix form

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

To minimize we take the derivative with respect to the weights $\boldsymbol{w}$, and find the minima by setting the derivative equal to zero

$$\begin{aligned}
\nabla_{\boldsymbol{w}}\mathcal{C} &= \nabla_{\boldsymbol{w}}(\hat{\boldsymbol{y}} - \boldsymbol{X}\boldsymbol{w})^T(\hat{\boldsymbol{y}} - \boldsymbol{X}\boldsymbol{w}), && (2.7)\\
&= -2\boldsymbol{X}^T\hat{\boldsymbol{y}} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w}, && (2.8)\\
\mathbf{0} &= -2\boldsymbol{X}^T\hat{\boldsymbol{y}} + 2\boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w}, && (2.9)\\
\boldsymbol{X}^T\hat{\boldsymbol{y}} &= \boldsymbol{X}^T\boldsymbol{X}\boldsymbol{w}, && (2.10)\\
\boldsymbol{w} &= (\boldsymbol{X}^T\boldsymbol{X})^{-1}\boldsymbol{X}^T\hat{\boldsymbol{y}}. && (2.11)
\end{aligned}$$

This requires matrix $\mathbf{X}^T\mathbf{X}$ to be invertible to get the solution [**?**].

The residuals $\boldsymbol{\epsilon}$ are given by

$$\boldsymbol{\epsilon} = \boldsymbol{y} - \hat{\boldsymbol{y}} = \boldsymbol{y} - \boldsymbol{X}\boldsymbol{w},$$

and with

$$\boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) = 0,$$

we have

$$\boldsymbol{X}^T\boldsymbol{\epsilon} = \boldsymbol{X}^T(\boldsymbol{y} - \boldsymbol{X}\boldsymbol{w}) = 0,$$

meaning that the solution for $\boldsymbol{w}$ is the one which minimizes the residuals. This method of regression is known as Ordinary Least Squares.

2.2 Over- and underfitting

In machine learning, when fitting a model to a data set the goal is nearly always to predict values or classify samples from regions of data the model has not seen. This is not a simple task, especially taking into consideration that data is rarely, if ever, noiseless. When extrapolating to unseen regions we must take steps to ensure the model complexity is appropriate - we want it to fit the signal, not the noise. First off - what do the terms "overfit" and "underfit" mean? An overfit model will typically perform well during the fitting procedure, but when presented with data outside the fitted region its performance decreases considerably. An underfit model lacks the expressive power to capture core signal variations in the data. Mehta et. al [**?**] demonstrates this concept through polynomial regression.

In machine learning literature and practice, you will encounter the concept of splitting the available data into two - training data and test data. We fit, or 'train' the model on the training data, and then assess the performance of the model on the test data. This practice lets us evaluate whether the model is overfitting to unseen data by comparing performance on the training data and test data.

2.3 The Bias-Variance Tradeoff

Considering the same dataset $\mathcal{L}$ consisting of the data, let us assume that the true data is generated from a noisy model

$$\mathbf{y} = f(\boldsymbol{x}) + \boldsymbol{\epsilon},$$

where ϵ is normally distributed with mean zero and standard deviation σ^2.

In our derivation of the ordinary least squares method we defined an approximation (2.6) to the function f in terms of the weights $\boldsymbol{w}$ and the input matrix $\boldsymbol{X}$ which together define our model, that is $\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w}$.

Thereafter we found the parameters $\boldsymbol{w}$ by optimizing the means squared error via the so-called cost function

$$\mathcal{C}(\boldsymbol{X}, \boldsymbol{w}) = \frac{1}{n}\sum_{i=0}^{n-1}(y_i - \hat{y}_i)^2 = \mathbb{E}\left[(\boldsymbol{y} - \hat{\boldsymbol{y}})^2\right].$$

We can rewrite this as

$$\mathbb{E}\left[(\boldsymbol{y} - \hat{\boldsymbol{y}})^2\right] = \frac{1}{n}\sum_i (f_i - \mathbb{E}\left[\hat{\boldsymbol{y}}\right])^2 + \frac{1}{n}\sum_i (\hat{y}_i - \mathbb{E}\left[\hat{\boldsymbol{y}}\right])^2 + \sigma^2.$$

The three terms represent the square of the bias of the learning method, which can be thought of as the error caused by the simplifying assumptions built into the method. The second term represents the variance of the chosen model and finally the last terms is variance of the error $\boldsymbol{\epsilon}$.

For a derivation of this equation, we refer to the work of Mehta et. al [**?**]. The authors also provide a list of universal lessons to keep in mind, which we quote here:

- Fitting is not predicting. Fitting existing data well is fundamentally different from making predictions about new data.
- Using a complex model can result in overfitting. Increasing a model's complexity (i.e number of fitting parameters) will usually yield better results on the training data. However when the training data size is small and the data are noisy, this results in *overfitting* and can substantially degrade the predictive performance of the model.
- For complex datasets and small training sets, simple models can be better at prediction complex models due to the bias-variance tradeoff. It takes less data to train a simple model than a complex one. Therefore, even though the correct model is guaranteed to have better predictive performance for an infinite amount of training data (less bias), the training errors stemming from finite-size sampling (variance) can cause simpler models to outperform the more complex model when sampling is limited.

2.4 Regularization

With the computing resources available today, increasing model complexity to deal with underfitting is usually a simple task. However, this computational freedom has led to overfitting being the common challenge to overcome.

2.5 Logistic Regression

Differently to linear regression, classification problems are concerned with outcomes taking the form of discrete variables. For a specific physical problem, we'd like to identify its state, say whether it is an ordered of disordered system. (cite?) One of the most basic examples of a classifier algorithm is logistic regression.

For a logistic regressor to improve it needs a way to track its performance. This is the purpose of a cost function. Essentially, the cost function says something about how wrong the model is in classifying the input. The objective in machine learning, and logistic regression, is then to minimize this error.

The cost function used in this project is called the **cross-entropy**, or the 'negative log likelihood', and takes the form

$$\mathcal{C}(\boldsymbol{w}) = -\sum_{i=1}^{n} \left(y_i(w_0 + w_1 x_i) - \log\left(1 + \exp\left(w_0 + w_1 x_i\right)\right)\right) \tag{2.12}$$

2.6 Gradient Descent

Minimizing the cost function is done using Gradient Descent. The gist of it is that to optimize the weights or coefficients, and biases to minimize the cost function, we can change their values to

$$\frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial \boldsymbol{w}} = -\boldsymbol{X}^T\left(\hat{y} - \hat{p}\right) \tag{2.13}$$

2.6.1 Stochastic Gradient Descent

The stochastic gradient descent method address some of the shortcomings of the normal gradient descent method. The gradient descent method is for instance sensitive to the choice of learning rate (cite?).

The underlying idea of stochastic gradient descent comes from observing that the cost function we want to minimize, almost always can be written as a sum over n data points. (cite?). Which gives

$$C(w) = \sum_{i=1}^{n} c_i(\mathbf{x}_i w) \tag{2.14}$$

(cite?)

This means that we also can find the gradient as a sum over i gradients as follows:

$$\Delta_w C(w) = \sum_{i}^{n} \Delta_w c_i(\mathbf{x}_i w) \tag{2.15}$$

(cite?)

Randomness is included by only taking the gradient on a subset of data.

2.7 Neural Networks

Artificial neural networks (ANN) are computational systems that can learn to perform tasks by considering examples, generally without being programmed with any task-specific rules. It is supposed to mimic a biological system, wherein neurons interact by sending signals in the form of mathematical functions between layers. All layers can contain an arbitrary number of neurons, and each connection is represented by a weight variable. The field of artificial neural networks has a long history of development, and is closely connected with the advancement of computer science and computers in general.

In natural sciences, ANNs have already found numerous applications. In statistical physics, they have been applied to detect phase transitions in 2D Ising and Potts models, lattice gauge theories, and different phases of polymers, or solving the Navier-Stokes equation in weather forecasting. Deep learning has also found interesting applications in quantum and nuclear physics.

The applications are not limited to the natural sciences. There is a plethora of applications in essentially all disciplines, from the humanities to life science and medicine.

2.7.1 Artificial Neurons

A model of artificial neurons was first developed by McCulloch and Pitts in 1943 [**?**] to study signal processing in the brain and has later been refined by others. The general idea is to mimic neural networks in the human brain, which is composed of billions of neurons that communicate with each other by sending electrical signals. Each neuron accumulates its incoming signals, which must exceed an activation threshold to yield an output. If the threshold is not overcome, the neuron remains inactive, i.e. has zero output. This behaviour inspired a simple mathematical model for an artificial neuron.

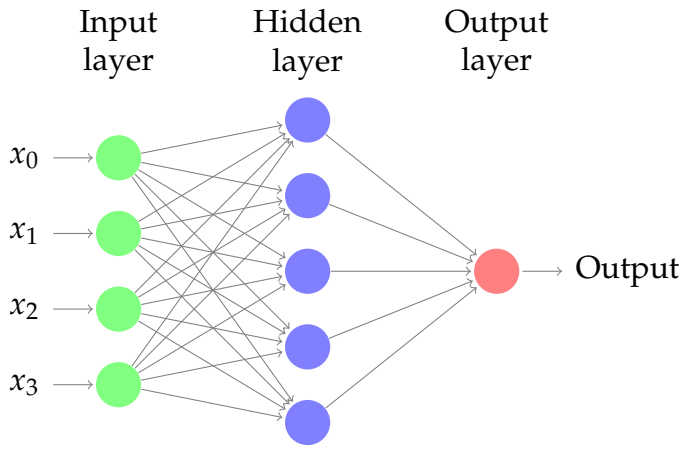

Figure 2.1: An example of a simple, feed-forward neural network architecture. Each input x_i is fed to each node in the hidden layer, where the value of the activation function $f(z)$ is calculated and passed on to the output layer. In this case the output layer consists of only one node.

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(z) \tag{2.16}$$

Here, the output y of the neuron is the value of its activation function, which receives as input a weighted sum of signals $x_i, \ldots, x_n$ received by n other neurons. Neurons are often referred to as "nodes" or "units" in machine learning literature, and we will use these interchangeably in the following sections.

2.7.2 The Feed Forward Neural Network

A network of only one neuron such as the one described above is typically referred to as a *perceptron*. The simplest network structure contains a single layer of N such nodes, and is most often called a *single-layer perceptron*. Adding additional layers of nodes, so-called *hidden layers*, results in a type of feed-forward neural network (FFNN), typically referred to as a Multilayer Perceptron (MLP) (see figure 2.1). The example is also a fully connected network, as every node in a layer is connected to every node in the next. The name "feed-forward" stems from the fact that information flows in only one direction: forward through the layers. First, for each node i in the first hidden layer, we calculate a weighted sum z_i^1 of the input coordinates x_j,

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \tag{2.17}$$

Here b_i is the *bias* which is needed in case of zero activation weights or inputs. How to fix the biases and the weights will be discussed below. The value of z_i^1 is the argument to the activation function f_i of each node i, The variable M stands for all possible inputs to a given node i in the first layer. We define the output y_i^1 of all neurons in layer 1 as

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right) \tag{2.18}$$

where we assume that all nodes in the same layer have identical activation functions, hence the notation f. In general, we could assume in the more general case that different layers have different activation functions. In this case we would identify these functions with a superscript l for the l-th layer,

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right) \tag{2.19}$$

where N_l is the number of nodes in layer l. When the output of all the nodes in the first hidden layer are computed, the values of the subsequent layer can be calculated and so forth until the output is obtained.

The output of neuron i in layer 2 is thus,

$$y_i^2 = f^2\left(\sum_{j=1}^{N} w_{ij}^2 y_j^1 + b_i^2\right) \tag{2.20}$$

$$= f^2\left[\sum_{j=1}^{N} w_{ij}^2 f^1\left(\sum_{k=1}^{M} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{2.21}$$

where we have substituted y_k^1 with the inputs x_k. Finally, the ANN output reads

$$y_i^3 = f^3\left(\sum_{j=1}^{N} w_{ij}^3 y_j^2 + b_i^3\right) \tag{2.22}$$

$$= f_3\left[\sum_j w_{ij}^3 f^2\left(\sum_k w_{jk}^2 f^1\left(\sum_m w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right] \tag{2.23}$$

We can generalize this expression to an MLP with l hidden layers. The complete functional form is,

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots\right) + b_k^2\right) + b_1^3\right] \tag{2.24}$$

which illustrates a basic property of MLPs: The only independent variables are the input values x_n. This confirms that an MLP, despite its quite convoluted mathematical form, is nothing more than an analytic function, specifically a mapping of real-valued vectors $\boldsymbol{x} \in \mathbb{R}^n \to \boldsymbol{y} \in \mathbb{R}^m$.

Furthermore, the flexibility and universality of an MLP can be illustrated by realizing that the expression is essentially a nested sum of scaled activation functions of the form

$$f(x) = c_1 f(c_2 x + c_3) + c_4 \tag{2.25}$$

where the parameters c_i are weights and biases. By adjusting these parameters, the activation functions can be shifted up and down or left and right, change slope or be rescaled which is the key to the flexibility of a neural network.

We will now introduce a more convenient notation for the activations in an ANN. We can represent the biases and activations as layer-wise column vectors $\boldsymbol{b}_l$ and $\boldsymbol{y}_l$, so that the i-th element of each vector is the bias b_i^l and activation y_i^l of node i in layer l respectively.

We have that $\boldsymbol{W}_l$ is an $N_{l-1} \times N_l$ matrix, while $\boldsymbol{b}_l$ and $\boldsymbol{y}_l$ are $N_l \times 1$ column vectors. With this notation, the sum becomes a matrix-vector multiplication, and we can write the equation for the activations of hidden layer 2 (assuming three nodes for simplicity) as

$$\boldsymbol{y}_2 = f_2(\boldsymbol{W}_2\boldsymbol{y}_1 + \boldsymbol{b}_2) = f_2\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}\right). \tag{2.26}$$

The activation of node i in layer 2 is

$$y_i^2 = f_2\Big(w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2\Big) = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right). \tag{2.27}$$

This is not just a convenient and compact notation, but also a useful and intuitive way to think about MLPs: The output is calculated by a series of matrix-vector multiplications and vector additions that are used as input to the activation functions. For each operation $\boldsymbol{W}_l\boldsymbol{y}_{l-1}$ we move forward one layer.

2.7.3 Activation Functions

Other than its connectivity, the choice of which activation function(s) to employ is one of the defining properties of a neural network. Not just any function will do, however, and there are several restrictions imposed on any applicable function. An activation function for an FFNN must be

- Non-constant
- Bounded
- Monotonically-increasing
- Continuous

As linear functions are not bounded, the second requirement exludes this entire family of functions. The output of a neural network with linear activation functions would be nothing more than a linear function of the inputs. We need to introduce some form of non-linearity to be able to fit non-linear functions. The most common examples of such functions are the logistic *sigmoid*

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.28}$$

and the *hyperbolic tangent*

$$f(x) = tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{2.29}$$

In addition to meeting the requirements, these functions also have derivatives that are relatively cheap to compute. The sigmoid's derivative is

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)), \tag{2.30}$$

and the hyperbolic tangents is

$$\frac{\partial tanh(x)}{\partial x} = 1 - tanh^{2}(x) \tag{2.31}$$

However, the sigmoid functions suffer from saturating gradients. This occurs when the functions value changes little to nothing with changes in the value of x. This has lead to a continuous search for alternatives, and one of the most popular activation functions to day is the *Rectified Linear Unit* (ReLU). The function, made especially popular after the success of Krizhevsky et al. [**?**], takes the following form

$$\text{ReLU}(x) = f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{2.32}$$

This function is certainly monotonic, and we can approximate its derivative with the Heaviside step-function, denoted $H(x)$ on the following form

$$H(x) = f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{2.33}$$

2.7.4 Types of neural networks

Convolutional Neural Network

A different variant of FFNNs are *convolutional neural networks* (CNNs), which have a connectivity pattern inspired by the animal visual cortex. Individual neurons in the visual cortex only respond to stimuli from small sub-regions of the visual field, called a receptive field. This makes the neurons well-suited to exploit the strong spatially local correlation present in natural images. The response of each neuron can be approximated mathematically as a convolution operation. (figure to come)

Convolutional neural networks emulate the behaviour of neurons in the visual cortex by enforcing a *local* connectivity pattern between nodes of adjacent layers: Each node in a convolutional layer is connected only to a subset of the nodes in the previous layer, in contrast to the fully-connected FFNN. Often, CNNs consist of several convolutional layers that learn local features of the input, with a fully-connected layer at the end, which gathers all the local data and produces the outputs. They have wide applications in image and video recognition.

2.7.5 Backpropagation

2.8 Assessing the Performance of Models

If classification accuracy is not enough to gauge whether a model is performing well, or well in the desired way, alternative way to measure performance must be explored. For cases of imbalanced data there are a few widely used methods that reveal information about the model that the simple accuracy metric can't.

2.8.1 Imbalanced Data in Classification

The physical world is rarely balanced.

A common challenge in classification is imbalanced data, in which a large amount of the labeled data belongs to just one or a few of the classes. For binary classification, if 90% of the data belongs to one of the classes, then the

classifier is likely to end up placing every single input in that class, as it will bring its accuracy to 90%. Technically, this accuracy is correct, but it's not very useful since the decision isn't at all affected by the features of the input. Accuracy alone isn't a good enough measure of performance to reveal this.

2.8.2 Confusion Matrix

A confusion matrix is an n by n matrix containing correct classifications on the diagonal, and false positives and negatives in the off-diagonal elements. An example of such a matrix could be the following table: In the table above

	True Cat	True Dog	True Rabbit
Predicted Cat	**5**	2	0
Predicted Dog	3	**3**	2
Predicted Rabbit	0	1	**11**

Table 2.1: Confusion matrix for an example classification where the classes are Cat, Dog and Rabbit. Correct classifications in bold.

(2.1), the diagonal elements $i = j$ are the correct classifications, while the other elements correspond to cases where the model predicted class i but should've predicted class j. The confusion matrix thus gives information about false positives and false negatives, in addition to classification accuracy. This is very useful in cases where for example false positives can be readily ignored or filtere later, but false negatives may have severe consequences. An example of this could be detection of cancer, in which a false positive can be ruled out from further testing, while a false negative may lead to a patient being sent home when actually needing help. For a more in-depth look at confusion matrices see [**?**].

2.8.3 Receiver Operating Characteristic

The Receiver Operating Characteristic (ROC) is a widely used measure of a classifiers performance . The performance is measured as the effect of the true positive rate (TPR) and the false positive rate (FPR) as a function of thresholding the positive class. To evaluate the ROC curve for a model, traditionally the Area Under the Curve (AUC) is used, which ranges from 0 (an ideal "opposite" classifier) to 1.0 (an ideal classifier) with 0.5 indicating a random choice classifier, For a thorough explanation of ROC curves and the underlying concepts, see [**?**].

2.9 Nuclear Science

2.9.1 Shell Structure

- Comprehensive and predictive model of atomic nuclei
  - Evolving structure of atomic nuclei as a function of protons and neutrons from first principles
- Understanding the origin of the elements
  - Explosive nucleosynthesis
- Use of atomic nuclei to test fundamental symmetries
- Search for new applications of isotopes and solutions to societal problems

Nomenclature

A nucleus Y has Z protons and N neutrons with a mass of $A = Z + N$. This is written as ${}^{\mathrm{A}}_{\mathrm{Z}}\mathrm{Y}_{\mathrm{N}}$. For a given nucleus there may be several

- Isotopes - nuclei with the same number of protons, but varying number of neutrons
  - ^{66}Ni, ^{67}Ni, ^{68}Ni, ^{69}Ni, ^{70}Ni
- Isotones - nuclei with the same number of neutrons, but varying number of protons
  - ^{70}Zn, ^{69}Cu, ^{68}Ni, ^{67}Co, ^{66}Fe
- Isobars - nuclei with the same number of nucleons A
  - ^{68}Fe, ^{68}Co, ^{68}Ni, ^{68}Cu, ^{68}Zn

Chapter 3

Method

3.1 Pretrained network

Chapter 4

Results

4.1 Classification

Our primary objective is to separate experimental data into two possible categories. Supervised learning typically leverages large amounts of labeled data to learn representations of the classes present in the inputs. A challenge in Physics is that experiments generate enormous amounts of data, but it is not labeled. Hand-labeling data is time-consuming and cost-ineffective. A possible solution to this challenge is to train models on simulated data. The idea being that simulated and experimental data are similar enough that the learned patterns from simulated data are, to some degree, transferrable to experimental data. The simulated data contains two classes of events - a single electron and a double electron.

4.1.1 Simulated data

In table 4.1 the performance of each model trained is reported using the f1-score. The model architecture for each model is described in (ref appendix). As a benchmark, we are including a state of the art pretrained network ([]VGG HERE) applied to the data using the approach described in 3.1. To give a broader picture of the performance we also include the calculated confusion matrix values in table 4.2. Even if f1-scores are similar, there may still be differences in either of the confusion matrix values.

4.1.2 Experimental data

Table 4.1: Mean F1-scores for classification of simulated data using multiple models. Error estimates are the standard deviation in results from k-fold cross-validation with $K = 5$ folds.

Logistic	Dense	Convolutional	Pretrained VGG16
0.734 $\pm 7.727 \times 10^{-3}$	0.907 $\pm 1.329 \times 10^{-2}$	0.959 $\pm 6.286 \times 10^{-3}$	0.894 $\pm 1.591 \times 10^{-2}$

Table 4.2: Mean confusion matrix values for classification of simulated data using multiple models. Error estimates are the standard deviation in results from k-fold cross-validation with $K = 5$ folds.

	TN	FP	FN	TP
Logistic	1.28×10^5 $\pm 1.546 \times 10^4$	6.16×10^4 $\pm 1.546 \times 10^4$	4.39×10^4 $\pm 1.118 \times 10^4$	1.46×10^5 $\pm 1.118 \times 10^4$
Dense	1.72×10^5 $\pm 1.125 \times 10^4$	1.85×10^4 $\pm 1.125 \times 10^4$	1.73×10^4 $\pm 5.205 \times 10^3$	1.73×10^5 $\pm 5.205 \times 10^3$
Convolutional	1.89×10^5 $\pm 7.287 \times 10^2$	1.39×10^3 $\pm 7.290 \times 10^2$	1.37×10^4 $\pm 2.692 \times 10^3$	1.76×10^5 $\pm 2.692 \times 10^3$
Pretrained VGG16	1.79×10^5 $\pm 9.079 \times 10^3$	1.15×10^4 $\pm 9.080 \times 10^3$	2.71×10^4 $\pm 9.409 \times 10^3$	1.63×10^5 $\pm 9.408 \times 10^3$

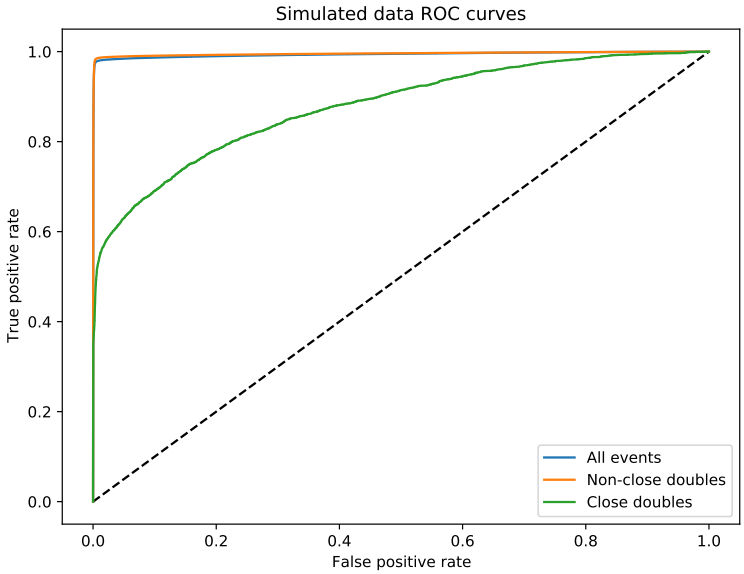

Figure 4.1: generic text

4.2 Regression

On data already classified, we attempt to predict the energy of the events and the position of origin for the electrons. Because there is a travel distance between the ejection site and the scintillator array, the positions aren't necessarily the locations of the highest-intensity pixels in the detector images.

4.2.1 Simulated data

Position - Single electron

Energy - Single electron

Position - Double electron

Energy - Double electron

Table 4.3: Mean R2-scores for regresson of positions of origin, on single events in simulated data, using multiple models. Error estimates are the standard deviation in results from k-fold cross-validation with $K = 5$ folds.

Linear	Dense	Convolutional	Pretrained VGG16
0.801 $\pm 3.664 \times 10^{-3}$	0.99 $\pm 8.185 \times 10^{-4}$	0.997 $\pm 2.401 \times 10^{-4}$	0.891 $\pm 6.864 \times 10^{-3}$

Table 4.4: Mean R2-scores for regresson of energy values, on single events in simulated data, using multiple models. Error estimates are the standard deviation in results from k-fold cross-validation with $K = 5$ folds.

Linear	Dense	Convolutional	Pretrained VGG16
0.926 $\pm 3.691 \times 10^{-2}$	0.931 $\pm 3.364 \times 10^{-2}$	0.936 $\pm 3.282 \times 10^{-2}$	0.935 $\pm 1.945 \times 10^{-2}$

Table 4.5: Mean R2-scores for regresson of positions of origin, on double events in simulated data, using multiple models. Error estimates are the standard deviation in results from k-fold cross-validation with $K = 5$ folds.

Linear	Dense	Convolutional	Pretrained VGG16
0.364 $\pm 5.796 \times 10^{-3}$	0.471 $\pm 1.809 \times 10^{-3}$	0.473 $\pm 2.394 \times 10^{-3}$	0.357 $\pm 1.079 \times 10^{-2}$

Table 4.6: Mean R2-scores for regresson of energy values, on double events in simulated data, using multiple models. Error estimates are the standard deviation in results from k-fold cross-validation with $K = 5$ folds.

Linear	Dense	Convolutional	Pretrained VGG16
0.415 $\pm 6.501 \times 10^{-2}$	0.416 $\pm 6.632 \times 10^{-2}$	0.428 $\pm 5.027 \times 10^{-2}$	0.404 $\pm 5.308 \times 10^{-2}$

Chapter 5

Discussion

Chapter 6

Conclusion