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

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 [1].

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 [2] 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 [2].

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

Finding the minima or maxima of a functions is a well-known process, perhaps especially so in physics. In machine learning most, if not all, cost optimization problems are cast as minimization problems, and we will do the same here.

The basic idea of gradient descent is that a function $F(x)$, $x \equiv (x_1, \ldots, x_n)$, decreases fastest if one goes from x in the direction of the negative gradient $-\nabla F(x)$. It can be shown that if

$$x_{k+1} = x_k - \gamma_k \nabla F(x_k),$$

with $\gamma_k > 0$, then for γ_k small enough, $F(x_{k+1}) \leq F(x_k)$. This means that for a sufficiently small γ_k we are always moving towards smaller function values, i.e a minimum. The first point, x_0, is an initial guess for the minimum. It could be chosen at random, or you could exploit some prior knowledge if available. The parameter γ_k is often referred to as step length or *learning rate*. We will be using the latter term in this thesis.

Ideally the sequence $\boldsymbol{x}_{kk=0}$ converges to a global minimum of the function F. We do not generally know if the minimum we find is local or global, unless we have the special case where F is a convex function. In this case all local minima are global minima, and gradient descent can converge to the global solution. However, gradient descent is sensitive to the choice of learning rate γ_k. As mentioned above $F(x_{k+1}) \leq F(x_k)$ is only guaranteed for sufficiently small γ_k. If the learning rate is too small the method will converge slowly. If it is too large we can experience erratic behaviour.

2.6.1 Stochastic Gradient Descent

The stochastic gradient descent (SGD) method address some of the shortcomings of the normal gradient descent method by introducing randomness. The cost function we wish to optimize can almost always be expressed as a sum

over n data points $\{x_i\}_{i=1}^n$.

$$\mathcal{C}(\boldsymbol{w}) = \sum_{i=1}^{n} c_i(\boldsymbol{x}_i \boldsymbol{w}) \tag{2.13}$$

Which gives us the ability to find the gradient as a sum over i gradients

$$\nabla_{\boldsymbol{w}} \mathcal{C}(\boldsymbol{w}) = \sum_{i}^{n} \nabla_{\boldsymbol{w}} c_i(\boldsymbol{x}_i \boldsymbol{w}) \tag{2.14}$$

Stochasticity/randomness is included by only taking the gradient on a subset of data called *minibatches*. Let the size of each minibatch be denoted M. Given a set of n datapoints, this gives us n/M minibatches, which we will denote B_k, with $k = 1, \ldots, n/M$.

Now, the procedure for calculating the gradient is now an approximation. Instead of summing over all data points, we sum over the data points in one minibatch, chosen at random each gradient step. This means that one gradient step is now given by

$$w_{j+1} = w_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_{\boldsymbol{w}} c_i(\boldsymbol{x}_i, w) \tag{2.15}$$

where k is chosen at random with equal probability from $[1, n/M]$. Iterating over the number of minibatches is commonly referred to as an *epoch*. When training a model it is typical to choose the number of epochs and then iterate over the number of minibatches each epoch. There are two important gains from this introduced stochasticity.

- Decreased chance that our optimization scheme gets stuck in a local minima.
- If the size of each minibatch is small relative to the number of datapoints, the computation of the gradient is much cheaper.

2.6.2 adam

Optimization of the training process has been a focus in the machine learning community. Introduced by Kingma and Lei Ba [3], *adam* has become the default choice of optimizer for a large number of machine learning applications. The algorithm keeps track of two moving averages; the average of the gradient (m_t) and the squared gradient (v_t). Related to these two quantities are two hyperparameters, $\beta_1, \beta_2 \in [0, 1)$, which control the exponential decay rates of these moving averages. The moments are described mathematically as

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.16}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta : 2) g_t^2 \tag{2.17}$$

where g_t is the gradient w.r.t the objective function at timestep t. In the paper, the authors also describe a problem with the initializing the moving averages as vectors of 0's - it leads to moment estimates that are biased towards zero, especially during the initial timesteps, and with small decay rates (βs close to 1). They do, however, provide a simple countermeasure to this bias, leading to the bias-currected moment estimates given by

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.18}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.19}$$

where β_i^t reads as β_i to the power t. The final parameter update is given by

$$x_{n+1} = x_n - \gamma \frac{\hat{m}_t}{\hat{v}_t + \epsilon} \tag{2.20}$$

where γ is the learning rate and ϵ is added to avoid divide by zero for small gradient values. The authors propose default settings for the hyperparameters, which are $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\gamma_1 = 10^{-3}$, and $\epsilon = 10^{-8}$. These are the values used for every model trained in this thesis, unless otherwise is specifically indicated.

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 [4] 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.

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(z) \tag{2.21}$$

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,

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \tag{2.22}$$

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

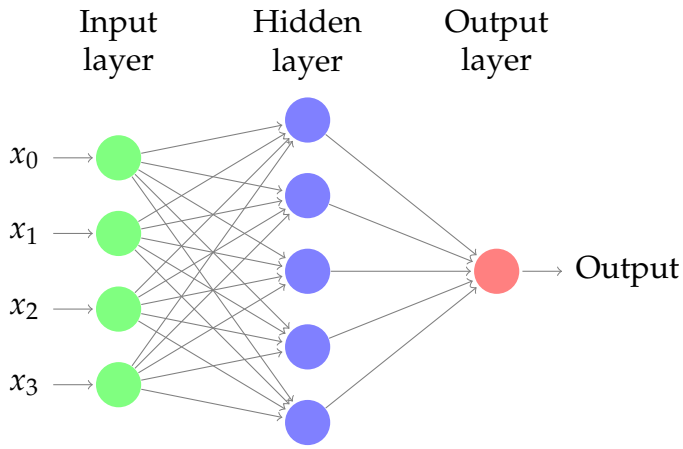

Figure 2.1: An example of a simple, feed-forward neural network architecture. Each input x_i is fed to each node in the hidden layer, where the value of the activation function $f(z)$ is calculated and passed on to the output layer. In this case the output layer consists of only one node.

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right) \tag{2.23}$$

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,

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right) \tag{2.24}$$

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,

$$y_i^2 = f^2\left(\sum_{j=1}^{N} w_{ij}^2 y_j^1 + b_i^2\right) \tag{2.25}$$

$$= f^2\left[\sum_{j=1}^{N} w_{ij}^2 f^1\left(\sum_{k=1}^{M} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{2.26}$$

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

$$y_i^3 = f^3\left(\sum_{j=1}^{N} w_{ij}^3 y_j^2 + b_i^3\right) \tag{2.27}$$

$$= f_3\left[\sum_j w_{ij}^3 f^2\left(\sum_k w_{jk}^2 f^1\left(\sum_m w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right] \tag{2.28}$$

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

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\dots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\dots\right) + b_k^2\right) + b_1^3\right] \tag{2.29}$$

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

$$f(x) = c_1 f(c_2 x + c_3) + c_4 \tag{2.30}$$

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

$$\boldsymbol{y}_2 = f_2(\boldsymbol{W}_2\boldsymbol{y}_1 + \boldsymbol{b}_2) = f_2\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}\right). \tag{2.31}$$

The activation of node i in layer 2 is

$$y_i^2 = f_2\Big(w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2\Big) = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right). \tag{2.32}$$

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*

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.33}$$

and the *hyperbolic tangent*

$$f(x) = tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}. \tag{2.34}$$

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

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)), \tag{2.35}$$

and the hyperbolic tangents is

$$\frac{\partial tanh(x)}{\partial x} = 1 - tanh^2(x) \tag{2.36}$$

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. [5], takes the following form

$$\text{ReLU}(x) = f(x) = \begin{cases} x, & \text{if } x > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{2.37}$$

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

$$H(x) = f'(x) = \begin{cases} 1, & \text{if } x > 0 \\ 0, & \text{otherwise.} \end{cases} \tag{2.38}$$

2.7.4 Backpropagation

As we have seen now in a feed-forward network, we can express the final output of our network in terms of basic matrix-vector multiplications. The unknown quantities are our weights w_{ij} and we need to find an algorithm for changing them so that our errors are as small as possible. This leads us to the famous backpropagation algorithm [6].

The questions we want to ask are how do changes in the biases and the weights in our network change the cost function and how can we use the final output to modify the weights? To derive these equations let us start with a plain regression problem and define our cost function

$$\mathcal{C}(\boldsymbol{W}) = \frac{1}{2}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2,$$

where the y_i's are our n targets (the values we want to reproduce), while the outputs of the network after having propagated all inputs $\hat{x}$ are given by y_i. Below we will demonstrate how the basic equations arising from the back-propagation algorithm can be modified to study classification problems with K classes.

With our definition of the targets $\boldsymbol{y}$, the outputs of the network $\hat{\boldsymbol{y}}$ and the inputs $\boldsymbol{x}$ we define now the activation z_j^l of node j of the l-th layer as a function of the bias, the weights which add up from the previous layer $l-1$ and the outputs $\boldsymbol{a}^{l-1}$ from the previous layer as

$$z_j^l = \sum_{i=1}^{M_{l-1}} w_{ij}^l a_i^{l-1} + b_j^l,$$

where b_k^l are the biases from layer l. Here M_{l-1} represents the total number of nodes of layer $l-1$. We can rewrite this in a more compact form as the matrix-vector products we discussed earlier,

$$\boldsymbol{z}^l = \left(\boldsymbol{W}^l\right)^T \boldsymbol{a}^{l-1} + \boldsymbol{b}^l.$$

With the activation values $\boldsymbol{z}^l$ we can in turn define the output of layer l as $\boldsymbol{a}^l = f(\boldsymbol{z}^l)$ where f is our activation function. In the examples here we will use the sigmoid function discussed in our logistic regression lectures. We will also use the same activation function f for all layers and their nodes. It means we have

$$a_j^l = f(z_j^l) = \frac{1}{1+e^{-(z_j^l)}}.$$

From the definition of the activation z_j^l we have

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

and

$$\frac{\partial z_j^l}{\partial a_i^{l-1}} = w_{ji}^l.$$

With our definition of the activation function we have that (note that this function depends only on z_j^l)

$$\frac{\partial a_j^l}{\partial z_j^l} = a_j^l(1-a_j^l) = f(z_j^l)(1-f(z_j^l)).$$

With these definitions we can now compute the derivative of the cost function in terms of the weights. Let us specialize to the output layer $l = L$. Our cost function is

$$\mathcal{C}(\boldsymbol{W}^L) = \frac{1}{2}\sum_{i=1}^n (\hat{y}_i - y_i)^2 = \frac{1}{2}\sum_{i=1}^n \left(a_i^L - y_i\right)^2,$$

The derivative of this function with respect to the weights is

$$\frac{\partial \mathcal{C}(\boldsymbol{W}^L)}{\partial w_{jk}^L} = \left(a_j^L - t_j\right)\frac{\partial a_j^L}{\partial w_{jk}^L},$$

The last partial derivative can easily be computed and reads (by applying the chain rule)

$$\frac{\partial a_j^L}{\partial w_{jk}^L} = \frac{\partial a_j^L}{\partial z_j^L}\frac{\partial z_j^L}{\partial w_{jk}^L} = a_j^L(1-a_j^L)a_k^{L-1},$$

We have thus

$$\frac{\partial \mathcal{C}(\boldsymbol{W}^L)}{\partial w_{jk}^L} = \left(a_j^L - t_j\right) a_j^L (1 - a_j^L) a_k^{L-1},$$

Defining

$$\delta_j^L = a_j^L(1 - a_j^L)\left(a_j^L - t_j\right) = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial (a_j^L)},$$

and using the Hadamard product of two vectors we can write this as

$$\boldsymbol{\delta}^L = f'(\boldsymbol{z}^L) \circ \frac{\partial \mathcal{C}}{\partial (\boldsymbol{a}^L)}. \tag{2.39}$$

This is an important expression. The second term on the right-hand side measures how fast the cost function is changing as a function of the jth output activation. If, for example, the cost function doesn't depend much on a particular output node j, then δ_j^L will be small, which is what we would expect. The first term on the right measures how fast the activation function f is changing at a given activation value z_j^L.
Notice that everything in the above equations is easily computed. In particular, we compute z_j^L while computing the behaviour of the network, and it is only a small additional overhead to compute $f'(z_j^L)$. The exact form of the derivative with respect to the output depends on the form of the cost function. However, provided the cost function is known there should be little trouble in calculating

$$\frac{\partial \mathcal{C}}{\partial (a_j^L)}$$

With the definition of δ_j^L we have a more compact definition of the derivative of the cost function in terms of the weights, namely

$$\frac{\partial \mathcal{C}(\hat{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}. \tag{2.40}$$

It is now possible to rewrite our previous equation for δ_j^L (2.39) as

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}, \tag{2.41}$$

which can also be interpreted as the partial derivative of the cost function with respect to the biases b_j^L, namely

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L}\frac{\partial b_j^L}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial b_j^L},$$

That is, the error δ_j^L is exactly equal to the rate of change of the cost function as a function of the bias.

We now have three equations that are essential for the computations of the derivatives of the cost function at the output layer. These equations are needed to start the algorithm and they are

$$\frac{\partial \mathcal{C}(\boldsymbol{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}, \tag{2.42}$$

and

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial a_j^L}, \tag{2.43}$$

and

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L}, \tag{2.44}$$

A consequence of the above equations is that when the activation a_k^{L-1} is small, the gradient term, that is the derivative of the cost function with respect to the weights, will also tend to be small. From this we gather that the weight changes (or "learns") slowly when we minimize the weights via gradient descent.

Another feature is that when the activation function (in this case sigmoid), is rather flat when we move towards its limit values 0 and 1. In these cases, the derivatives of the activation function will also be close to zero, meaning again that the gradients will be small and the network learns slowly again.
We need a fourth equation and we are set. We are going to propagate backwards to determine the weights and biases. To do so we need to represent the error in the layer before the final one $L-1$ in terms of the errors in the final output layer. Replacing the final layer L with a general layer l, we have

$$\delta_j^l = \frac{\partial \mathcal{C}}{\partial z_j^l}.$$

We want to express this in terms of the equations for layer $l+1$. Using the chain rule and summing over all k entries we have

$$\delta_j^l = \sum_k \frac{\partial \mathcal{C}}{\partial z_k^{l+1}}\frac{\partial z_k^{l+1}}{\partial z_j^l} = \sum_k \delta_k^{l+1}\frac{\partial z_k^{l+1}}{\partial z_j^l},$$

and recalling that

$$z_j^{l+1} = \sum_{i=1}^{M_l} w_{ij}^{l+1} a_i^l + b_j^{l+1},$$

with M_l being the number of nodes in layer l, we arrive at

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l), \tag{2.45}$$

The four equations provide us with a way of computing the gradient of the cost function.

First, we set up the input data $\boldsymbol{x}$ and the activations $\boldsymbol{z}_1$ of the input layer and compute the activation function and the pertinent outputs $\boldsymbol{a}^1$.

Secondly, we perform the feed-forward until we reach the output layer and compute all $\boldsymbol{z}_l$ of the input layer and compute the activation function and the pertinent outputs $\boldsymbol{a}^l$ for $l = 2, 3, \ldots, L$. Next we compute the ouput error $\boldsymbol{\delta}^L$ by computing all

$$\delta_j^L = f'(z_j^L) \frac{\partial \mathcal{C}}{\partial a_j^L}.$$

Then we compute the back propagate error for each $l = L-1, L-2, \ldots, 2$ as

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l).$$

Finally, we update the weights and the biases using gradient descent for each $l = L-1, L-2, \ldots, 2$ and update the weights and biases according to the rules

$$w_{jk}^l \leftarrow= w_{jk}^l - \eta \delta_j^l a_k^{l-1},$$

$$b_j^l \leftarrow b_j^l - \eta \frac{\partial \mathcal{C}}{\partial b_j^l} = b_j^l - \eta \delta_j^l,$$

The parameter η is the learning parameter discussed in connection with the gradient descent methods.

2.7.5 Convolutional Neural Networks

Convolutional Neural Networks (CNNs) share quite a few similarities with ordinary neural networks, and all the concepts developed for neural networks so far still apply. The difference is that CNNs assume the inputs to be images.

A problem with regular neural networks is that they scale poorly to large images. As an example, consider an image of size $32 \times 32 \times 3$ (32 wide, 32 high, 3 color channels), so a single fully-connected neuron in a first hidden layer

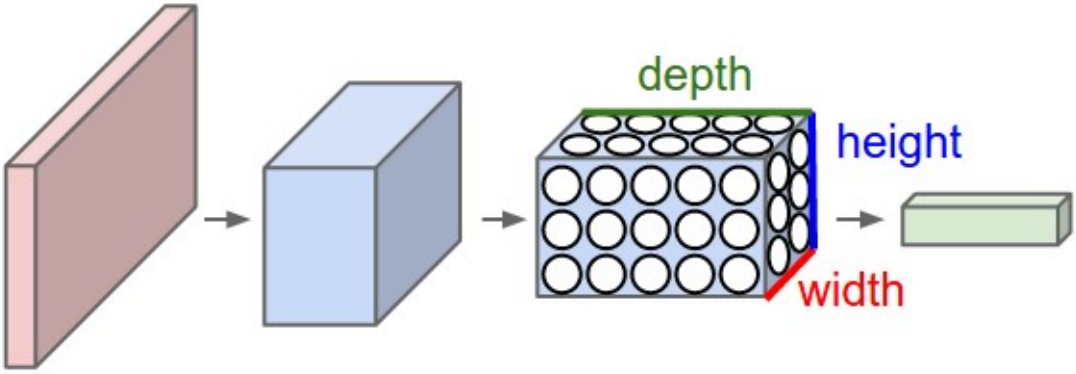

Figure 2.2: A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels). Image borrowed from CS231n's github page [**?**] on CNNs

of a regular Neural Network would have $32 \times 32 \times 3 = 3072$ weights. This amount still seems manageable, but clearly this fully-connected structure does not scale to larger images. For example, an image of more respectable size, say $200 \times 200 \times 3$, would lead to neurons that have $200 \times 200 \times 3 = 120,000$ weights. Adding several such neurons then quickly increases the number of parameters, which in turn increases the risk of overfitting.

CNNs take advantage of the fact that the input consists of images and they constrain the architecture in a more sensible way. In particular, unlike a regular NN, the layers of a CNN have neurons arranged in 3 dimensions: width, height, depth. (Note that the word depth here refers to the third dimension of an activation volume, not to the depth of a full NN, which can refer to the total number of layers in a network.) The above example of an image with an input volume of activations has dimensions $32 \times 32 \times 3$ (width, height, depth respectively). See figure 2.2 for an illustration.

The neurons in a layer will only be connected to a small region of the layer before it, instead of all of the neurons in a fully-connected manner. Moreover, the final output layer could for this specific image have dimensions $1 \times 1 \times 10$, because by the end of the CNN architecture we will reduce the full image into a single vector of class scores, arranged along the depth dimension.

A simple CNN is a sequence of layers, and every layer of a CNN transforms one volume of activations to another through a differentiable function. We use three main types of layers to build CNN architectures: Convolutional Layer, Pooling Layer, and Dense (fully connected, exactly as seen in regular Neural Networks). We will stack these layers to form a full CNN architecture.

A simple CNN for image classification could have the architecture:

- **INPUT** ($32 \times 32 \times 3$) will hold the raw pixel values of the image, in this case an image of width 32, height 32, and with three color channels R,G,B.
- **CONV** (convolutional)layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume such as $[32 \times 32 \times 12]$ if we decided to use 12 filters.
- **RELU** layer will apply an elementwise activation function, such as the $max(0, x)$ thresholding at zero. This leaves the size of the volume unchanged ($[32 \times 32 \times 12]$).
- **POOL** (pooling) layer will perform a downsampling operation along the spatial dimensions (width, height), resulting in volume such as $[16 \times 16 \times 12]$.
- **DENSE** (i.e. fully-connected) layer will compute the class scores, resulting in volume of size $[1 \times 1 \times 10]$, where each of the 10 numbers correspond to a class score, such as among the 10 categories of the MNIST images we considered above . As with ordinary Neural Networks and as the name implies, each neuron in this layer will be connected to all the numbers in the previous volume.

CNNs transform the original image layer by layer from the original pixel values to the final class scores. Observe that some layers contain parameters and other don't. In particular, the CNN layers perform transformations that are a function of not only the activations in the input volume, but also of the parameters (the weights and biases of the neurons). On the other hand, the RELU/POOL layers will implement a fixed function. The parameters in the CONV/FC layers will be trained with gradient descent so that the class scores that the CNN computes are consistent with the labels in the training set for each image.
For a more in-depth breakdown of convolutional neural networks, we refer to Stanford's excellent course CS231n, and their text on CNNs [**?**].

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 [7].

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 [7].

2.9 Experimental Background

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

- Low-energy nuclear physics
- Scintillator
- Challenges with the data
- Current methods for fitting
-

The experiment produces data on the order of TerraBytes.

Very summarized goals of nuclear physics.

- Comprehensive and predictive model of atomic nuclei
- Understanding the origin of the elements
- Use of atomic nuclei to test fundamental symmetries
- Search for new applications of isotopes and solutions to societal problems

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
- Isotones - nuclei with the same number of neutrons, but varying number of protons
- Isobars - nuclei with the same number of nucleons A

2.9.1 Data

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. In figure 4.2 we show the performance of the same architectures trained on data where specific pixels are set to 0 to mimic the experimental dataset. in figure 4.1

4.1.2 Experimental data

Table 4.1: Mean F1-scores and roc-auc 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
F1-score	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 $\pm 1.591 \times 10^{-2}$
AUC	0.833 $\pm 6.515 \times 10^{-4}$	0.951 $\pm 1.774 \times 10^{-2}$	0.986 $\pm 2.185 \times 10^{-3}$	0.947 $\pm 8.505 \times 10^{-3}$

Table 4.2: Mean F1-scores and roc-auc scores for classification of simulated data with specific pixels modified, 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
F1-score	0.735 $\pm 2.273 \times 10^{-3}$	0.904 $\pm 8.276 \times 10^{-3}$	0.942 $\pm 3.640 \times 10^{-2}$	0.908 $\pm 1.926 \times 10^{-2}$
AUC	0.832 $\pm 9.779 \times 10^{-4}$	0.952 $\pm 2.604 \times 10^{-3}$	0.98 $\pm 9.848 \times 10^{-3}$	0.953 $\pm 9.530 \times 10^{-3}$

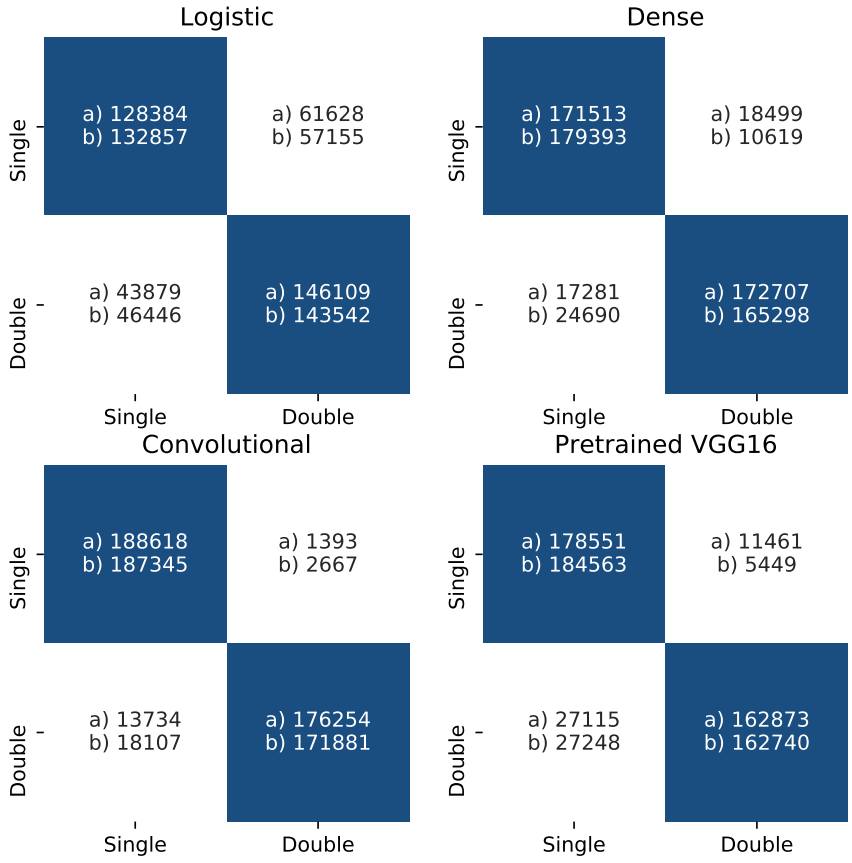

Figure 4.1: Confusion matrices for each model trained on simulated data. a) unmodified data. b) select pixels set to zero.

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 positions of origin, on single events in simulated data with specific pixels set to zero, 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
0.802 $\pm 2.737 \times 10^{-3}$	0.989 $\pm 6.798 \times 10^{-4}$	0.996 $\pm 1.015 \times 10^{-3}$	0.892 $\pm 1.723 \times 10^{-2}$

Table 4.5: 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 4.6: Mean R2-scores for regresson of energy values, on single events in simulated data with specific pixels set to zero, 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
0.965 $\pm 2.459 \times 10^{-2}$	0.97 $\pm 2.261 \times 10^{-2}$	0.969 $\pm 2.405 \times 10^{-2}$	0.949 $\pm 1.418 \times 10^{-2}$

Table 4.7: 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 4.8: Mean R2-scores for regresson of positions of origin, on double events in simulated data with specific pixels set to zero, 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
0.37 $\pm 6.804 \times 10^{-4}$	0.47 $\pm 8.661 \times 10^{-4}$	0.472 $\pm 1.841 \times 10^{-3}$	0.355 $\pm 1.397 \times 10^{-2}$

Table 4.9: 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 $\pm 5.308 \times 10^{-2}$

Table 4.10: Mean R2-scores for regresson of energy values, on double events in simulated data with specific pixels set to zero, 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
0.485 $\pm 3.155 \times 10^{-3}$	0.487 $\pm 2.571 \times 10^{-3}$	0.487 $\pm 3.120 \times 10^{-3}$	0.447 $\pm 1.032 \times 10^{-2}$

Chapter 5

Discussion

Chapter 6

Conclusion

Bibliography

[1] G. James, D. Witten, T. Hastie, and R. Tibshirani. *An introduction to Statistical Learning*, volume 7. 2000. ISBN 978-1-4614-7137-0. doi: 10.1007/978-1-4614-7138-7.

[2] P. Mehta, M. Bukov, C. H. Wang, et al. A high-bias, low-variance introduction to Machine Learning for physicists. *Physics Reports*, 810:1–124, 2019. ISSN 03701573. doi: 10.1016/j.physrep.2019.03.001.

[3] D. P. Kingma and J. L. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*. International Conference on Learning Representations, ICLR, dec 2015. URL https://arxiv.org/abs/1412.6980v9.

[4] W. S. McCulloch and W. Pitts. A logical calculus of the ideas immanent in nervous activity. *The Bulletin of Mathematical Biophysics*, 5(4):115–133, dec 1943. ISSN 00074985. doi: 10.1007/BF02478259. URL https://link.springer.com/article/10.1007/BF02478259.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet Classification with Deep Convolutional Neural Networks. Technical report, 2012. URL http://code.google.com/p/cuda-convnet/https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[6] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. ISSN 1476-4687. doi: 10.1038/323533a0. URL https://doi.org/10.1038/323533a0.

[7] T. Fawcett. An introduction to ROC analysis. *Pattern Recognition Letters*, 27 (8):861–874, 2006. ISSN 01678655. doi: 10.1016/j.patrec.2005.10.010.