Supervised Learning in a Low-Energy Nuclear Physics Experiment

by

Geir Tore Ulvik

Thesis

for the degree of

Master of Science

Faculty of Mathematics and Natural Sciences
University of Oslo

January 2020

Contents

List of Figures

Acknowledgements

Acknowledgement text like lorem ipsum all up in here.

Abstract

This is the abstract text.

Chapter 1

Introduction

In this thesis, we the gap between simulation and experiment. Modelling the world around us to predict behaviour and increase our understanding is a key concept in physics, and science as a whole. Simulating physical processes often lay the groundwork for verifying theory, and designing experiments. For many branches of physics, and especially nuclear physics, there has been an exponential growth in data generated from experiments. This brings several challenges, the most immediate of which may be "how do we process it all?". If there are multiple types of reactions occurring in an experiment, how can we separate them when there are billions of events? In the later years, machine learning algorithms have gained popularity for accomplishing this task, alleviating the need for humans to sift through vast amounts of data by hand. In this thesis, we explore the leveraging of simulated experiments to train machine learning models and subsequently apply them to data gathered from real-world experiments.

Nuclear physics seeks to understand the building blocks of the universe - nuclei - and from that understanding build a comprehensive and predictive model of them. The vast majority of nuclei discovered so far are not stable, and the study of many of them requires specialized equipment.

Chapter 2

Theory

In this thesis we explore the use of supervised learning on simulated and experimental nuclear physics data. To build a foundation for the following discussion, we introduce the fundamental concepts underpinning the advanced machine learning techniques of today.

We note that the following chapter follows closely the work of Morten Hjorth-Jensen[2], Mehta et. al[3], Goodfellow et. al[4], and the University of Stanfords course CS231n[1].

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

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

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

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

This requires matrix $\mathbf{X}^T\mathbf{X}$ to be invertible to get the solution [5].

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 [3] 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 the derivation of this equation, we refer to the work of Mehta et. al [3].

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. The **no free lunch theorem** for machine learning [6] states that, averaged over all possible data-generating distributions, every classification algorithm has the same error rate when classifying previously unobserved points. In other

words,in some sense, no machine learning algorithm is universally any better than anyother. The most sophisticated algorithm we can design has the same average performance (over all possible tasks) as merely predicting that every point belongs to the same class. Fortunately, this holds only when we average over *all* possible data-generating distributions[4]. This means that we do not seek a universally optimal classifier for all problems. Rather, for each learning problem we face, we can use knowledge of the problem to build a set of preferences into the learning algorithm, such that the algorithm becomes better suited to the specific problem.

As an example, we can modify the cost function for linear regression with the inclusion of **weight decay**. Instead of minimizing only the mean squared error, as defined in equation 2.5, we minimize a sum of the MSE and a criterion that expresses a preference for the weights to have smaller squared L_2 norm. This takes the form

$$\mathcal{C}(\boldsymbol{w}) = MSE + \lambda \boldsymbol{w}^T \boldsymbol{w}, \tag{2.12}$$

where λ is a value that controls the strength of our preference for smaller weights, chosen ahead of training. λ is often referred to as a *hyperparameter*. When $\lambda = 0$, there is no preference, and the cost function becomes the regular MSE. As λ increases it forces the weights to become smaller. Minimization of the cost function when $\lambda > 0$ results in weights that make a tradeoff between fitting the training data and being small[4]. We will come back to the concept of regularization in the context of deep learning later in the chapter.

2.5 Logistic Regression

Differently to the task in linear regression, or regression problems as a whole, classification problems are concerned with outcomes taking the form of discrete variables. The descrete variable could be a category, and the task at hand might be to determine whether an images is of a dog or a cat. In this case the category can take one of two value - we have a binary outcome. One of the most basic examples of a classifier algorithm is logistic regression, and it serves as a stepping stone towards neural networks and deep learning. Taking its name from the task itself, the categories are typically called *classes*, which we will use going forward.

We consider the case where the dependent variables, also called the responses or the outcomes, y_i are discrete and only take values from $k = 0, \ldots, K-1$ (K classes).

The goal is to predict the output classes from the inputs $\boldsymbol{X} \in \mathbb{R}^{n \times p}$ made of n samples, each of which carries p features or predictors. The primary goal is to identify the classes to which new unseen samples belong.

Let us specialize to the case of two classes only, with outputs $y_i = 0$ and

$y_i = 1$. Our outcomes could represent the status of a credit card user that could default or not on her/his credit card debt. That is

$$y_i = \begin{bmatrix} 0 & \text{no} \\ 1 & \text{yes} \end{bmatrix}. \tag{2.13}$$

The perceptron is an example of a "hard classification" model. We will encounter this model when we discuss neural networks as well. Each datapoint is deterministically assigned to a category (i.e $y_i = 0$ or $y_i = 1$). In many cases, it is favorable to have a "soft" classifier that outputs the probability of a given category rather than a single value. For example, given x_i, the classifier outputs the probability of being in a category k. Logistic regression is the most common example of a soft classifier. In logistic regression, the probability that a data point x_i belongs to a category $y_i = \{0, 1\}$ is given by the logit function (or Sigmoid) which is meant to represent the likelihood of a given event,

$$\sigma(x) = \frac{1}{1 + e^{-x}} = \frac{e^x}{1 + e^x}. \tag{2.14}$$

Note that $1 - \sigma(x) = \sigma(-x)$.
We assume now that we have two classes with y_i either 0 or 1. Furthermore we assume also that we have only two parameters w in our fitting of the Sigmoid function, that is we define probabilities

$$\sigma(y_i = 1|x_i, \boldsymbol{w}) = \frac{\exp(w_0 + w_1 x_i)}{1 + \exp(w_0 + w_1 x_i)},$$
$$\sigma(y_i = 0|x_i, \boldsymbol{w}) = 1 - \sigma(y_i = 1|x_i, \boldsymbol{w}),$$

where $\boldsymbol{w}$ are the weights we wish to extract from data, in our case w_0 and w_1. Note that we used

$$\sigma(y_i = 0|x_i, \boldsymbol{w}) = 1 - \sigma(y_i = 1|x_i, \boldsymbol{w}). \tag{2.15}$$

In order to define the total likelihood for all possible outcomes from a dataset $\mathcal{D} = \{(y_i, x_i)\}$, with the binary labels $y_i \in \{0, 1\}$ and where the data points are drawn independently, we use what is commonly referred to as the "Maximum Likelihood Estimation". We aim thus at maximizing the probability of seeing the observed data. We can then approximate the likelihood in terms of the product of the individual probabilities of a specific outcome y_i. That is

$$P(\mathcal{D}|\boldsymbol{w}) = \prod_{i=1}^{n} \left[\sigma(y_i = 1|x_i, \boldsymbol{w})\right]^{y_i} \left[1 - \sigma(y_i = 1|x_i, \boldsymbol{w}))\right]^{1-y_i}$$

from which we obtain the log-likelihood and our cost function.

$$\mathcal{C}(\boldsymbol{w}) = \sum_{i=1}^{n} \left(y_i \log \sigma(y_i = 1|x_i, \boldsymbol{w}) + (1 - y_i) \log \left[1 - \sigma(y_i = 1|x_i, \boldsymbol{w}) \right] \right). \tag{2.16}$$

Reordering the logarithms, we can rewrite the cost function as

$$\mathcal{C}(\boldsymbol{w}) = \sum_{i=1}^{n} \left(y_i(w_0 + w_1 x_i) - \log(1 + \exp(w_0 + w_1 x_i)) \right). \tag{2.17}$$

The maximum likelihood estimator is defined as the set of parameters that maximize the log-likelihood where we maximize with respect to $\boldsymbol{w}$. Since the cost (error) function is just the negative log-likelihood, for logistic regression we have that

$$\mathcal{C}(\boldsymbol{w}) = -\sum_{i=1}^{n} \left(y_i(w_0 + w_1 x_i) - \log(1 + \exp(w_0 + w_1 x_i)) \right). \tag{2.18}$$

This equation is known in statistics as the **cross entropy**. Finally, we note that just as in the example of linear regression, we often supplement the cross-entropy with additional regularization terms, usually L_1 and L_2 regularization.

The cross entropy is a convex function of the weights $\boldsymbol{w}$, therefore any local minimizer is a global minimizer. Minimizing this cost function with respect to the two parameters w_0 and w_1 we obtain

$$\frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial w_0} = -\sum_{i=1}^{n} \left(y_i - \frac{\exp(w_0 + w_1 x_i)}{1 + \exp(w_0 + w_1 x_i)} \right), \tag{2.19}$$

and

$$\frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial w_1} = -\sum_{i=1}^{n} \left(y_i x_i - x_i \frac{\exp(w_0 + w_1 x_i)}{1 + \exp(w_0 + w_1 x_i)} \right). \tag{2.20}$$

Let us now define a vector $\boldsymbol{y}$ with n elements y_i, an $n \times p$ matrix $\boldsymbol{X}$ which contains the x_i values and a vector $\boldsymbol{\sigma}$ of fitted probabilities $\sigma(y_i|x_i, \boldsymbol{w})$. We can rewrite the first derivative of the cost function in a more compact form

$$\frac{\partial \mathcal{C}(\boldsymbol{w})}{\partial \boldsymbol{w}} = -\boldsymbol{X}^T (\boldsymbol{y} - \boldsymbol{\sigma}). \tag{2.21}$$

If we in addition define a diagonal matrix $\boldsymbol{W}$ with elements $\sigma(y_i|x_i, \boldsymbol{w})(1 - \sigma(y_i|x_i, \boldsymbol{w})$, we can obtain a compact expression of the second derivative as

$$\frac{\partial^2 \mathcal{C}(\boldsymbol{w})}{\partial \boldsymbol{w} \partial \boldsymbol{w}^T} = \boldsymbol{X}^T \boldsymbol{W} \boldsymbol{X}. \tag{2.22}$$

Within a binary classification problem, we can easily expand our model to include multiple predictors. Our ratio between likelihoods is then with p predictors

$$\log(\frac{\sigma(\boldsymbol{wx})}{1-\sigma(\boldsymbol{wx})}) = w_0 + w_1x_1 + w_2x_2 + \cdots + w_px_p. \tag{2.23}$$

Here we defined $\boldsymbol{x} = [1, x_1, x_2, \ldots, x_p]$ and $\boldsymbol{w} = [w_0, w_1, \ldots, w_p]$ leading to

$$\sigma(\boldsymbol{wx}) = \frac{\exp(w_0 + w_1x_1 + w_2x_2 + \cdots + w_px_p)}{1 + \exp(w_0 + w_1x_1 + w_2x_2 + \cdots + w_px_p)}. \tag{2.24}$$

Next we look at methods for minimizing these functions.

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

$$\mathcal{C}(\boldsymbol{w}) = \sum_{i=1}^{n} c_i(\boldsymbol{x}_i \boldsymbol{w}) \tag{2.25}$$

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

$$\nabla_{\boldsymbol{w}} \mathcal{C}(\boldsymbol{w}) = \sum_{i}^{n} \nabla_{\boldsymbol{w}} c_i(\boldsymbol{x}_i \boldsymbol{w}) \tag{2.26}$$

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

$$w_{j+1} = w_j - \gamma_j \sum_{i \in B_k}^{n} \nabla_{\boldsymbol{w}} c_i(\boldsymbol{x}_i, w) \tag{2.27}$$

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 [7], *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

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \tag{2.28}$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta : 2) g_t^2 \tag{2.29}$$

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

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{2.30}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{2.31}$$

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

$$x_{n+1} = x_n - \gamma \frac{\hat{m}_t}{\hat{v}_t + \epsilon} \tag{2.32}$$

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 [8] 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.

$$y = f\left(\sum_{i=1}^{n} w_i x_i\right) = f(z) \tag{2.33}$$

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,

$$z_i^1 = \sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1 \tag{2.34}$$

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

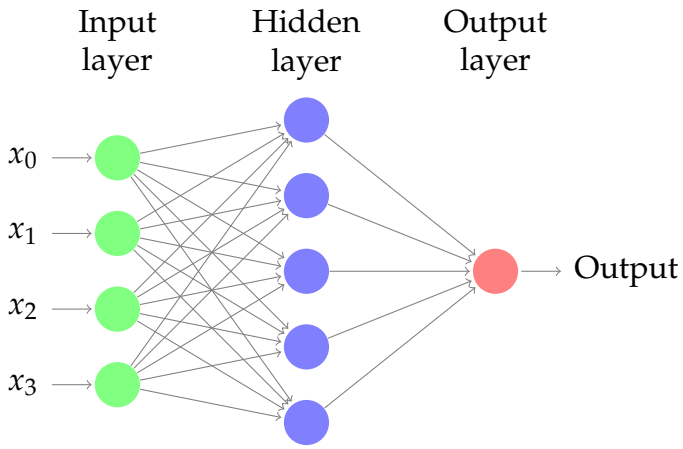

Figure 2.1: An example of a simple, feed-forward neural network architecture. Each input x_i is fed to each node in the hidden layer, where the value of the activation function $f(z)$ is calculated and passed on to the output layer. In this case the output layer consists of only one node.

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

$$y_i^1 = f(z_i^1) = f\left(\sum_{j=1}^{M} w_{ij}^1 x_j + b_i^1\right) \tag{2.35}$$

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,

$$y_i^l = f^l(u_i^l) = f^l\left(\sum_{j=1}^{N_{l-1}} w_{ij}^l y_j^{l-1} + b_i^l\right) \tag{2.36}$$

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,

$$y_i^2 = f^2\left(\sum_{j=1}^{N} w_{ij}^2 y_j^1 + b_i^2\right) \tag{2.37}$$

$$= f^2\left[\sum_{j=1}^{N} w_{ij}^2 f^1\left(\sum_{k=1}^{M} w_{jk}^1 x_k + b_j^1\right) + b_i^2\right] \tag{2.38}$$

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

$$y_i^3 = f^3\left(\sum_{j=1}^{N} w_{ij}^3 y_j^2 + b_i^3\right) \tag{2.39}$$

$$= f_3\left[\sum_j w_{ij}^3 f^2\left(\sum_k w_{jk}^2 f^1\left(\sum_m w_{km}^1 x_m + b_k^1\right) + b_j^2\right) + b_1^3\right] \tag{2.40}$$

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

$$y_i^{l+1} = f^{l+1}\left[\sum_{j=1}^{N_l} w_{ij}^3 f^l\left(\sum_{k=1}^{N_{l-1}} w_{jk}^{l-1}\left(\ldots f^1\left(\sum_{n=1}^{N_0} w_{mn}^1 x_n + b_m^1\right)\ldots\right) + b_k^2\right) + b_1^3\right] \tag{2.41}$$

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

$$f(x) = c_1 f(c_2 x + c_3) + c_4 \tag{2.42}$$

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

$$\boldsymbol{y}_2 = f_2(\boldsymbol{W}_2\boldsymbol{y}_1 + \boldsymbol{b}_2) = f_2\left(\begin{bmatrix} w_{11}^2 & w_{12}^2 & w_{13}^2 \\ w_{21}^2 & w_{22}^2 & w_{23}^2 \\ w_{31}^2 & w_{32}^2 & w_{33}^2 \end{bmatrix} \cdot \begin{bmatrix} y_1^1 \\ y_2^1 \\ y_3^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \\ b_3^2 \end{bmatrix}\right). \tag{2.43}$$

The activation of node i in layer 2 is

$$y_i^2 = f_2\Big(w_{i1}^2 y_1^1 + w_{i2}^2 y_2^1 + w_{i3}^2 y_3^1 + b_i^2\Big) = f_2\left(\sum_{j=1}^{3} w_{ij}^2 y_j^1 + b_i^2\right). \tag{2.44}$$

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* as seen previously in equation 2.14, and the *hyperbolic tangent*

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}. \tag{2.45}$$

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

$$\frac{\partial \sigma(x)}{\partial x} = \sigma(x)(1 - \sigma(x)), \tag{2.46}$$

and the hyperbolic tangents is

$$\frac{\partial tanh(x)}{\partial x} = 1 - tanh^2(x) \tag{2.47}$$

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. [9], takes the following form

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

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

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

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 [10].

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

$$\boldsymbol{\delta}^L = f'(\boldsymbol{z}^L) \circ \frac{\partial \mathcal{C}}{\partial(\boldsymbol{a}^L)}. \tag{2.50}$$

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

$$\frac{\partial \mathcal{C}(\hat{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}. \tag{2.51}$$

It is now possible to rewrite our previous equation for δ_j^L (2.50) as

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial z_j^L} = \frac{\partial \mathcal{C}}{\partial a_j^L}\frac{\partial a_j^L}{\partial z_j^L}, \tag{2.52}$$

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

$$\frac{\partial \mathcal{C}(\boldsymbol{W}^L)}{\partial w_{jk}^L} = \delta_j^L a_k^{L-1}, \tag{2.53}$$

and

$$\delta_j^L = f'(z_j^L)\frac{\partial \mathcal{C}}{\partial a_j^L}, \tag{2.54}$$

and

$$\delta_j^L = \frac{\partial \mathcal{C}}{\partial b_j^L}, \tag{2.55}$$

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

$$\delta_j^l = \sum_k \delta_k^{l+1} w_{kj}^{l+1} f'(z_j^l), \tag{2.56}$$

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

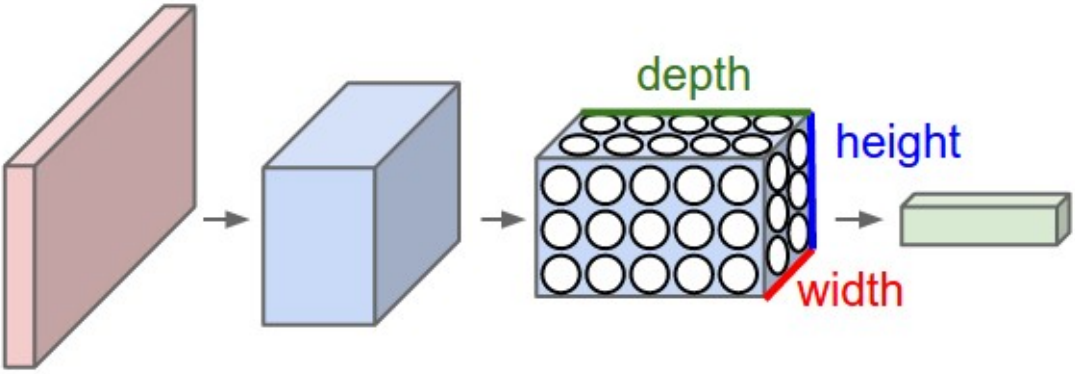

Figure 2.2: A CNN arranges its neurons in three dimensions (width, height, depth), as visualized in one of the layers. Every layer of a CNN transforms the 3D input volume to a 3D output volume of neuron activations. In this example, the red input layer holds the image, so its width and height would be the dimensions of the image, and the depth would be 3 (Red, Green, Blue channels). Image borrowed from CS231n's github page [1] 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 [1].

2.8 Performance metrics

To go further into the backround for the chosen metrics, we need to establish four quantities that appear in most, if not all discussion of the topic. These are

- True positive (TP) - Double event classified as double event
- True negative (TN) - Single event classified as single event
- False positive (FP) - Single event classified as double event
- False negative (FN) - Double event classified as single event

Which type of event is "positive" and "negative" is an arbitrary choice. From these terms we can define some properties of a classifier. **Sensitivity** (or **True positive rate** (TPR), or **recall**) measures the fraction of positive samples in the data that are correctly classified as positive.

$$\text{sensitivity} = \frac{\text{number of true positives}}{\text{number of true positives} + \text{number of false negatives}} = \frac{TP}{TP + FN}$$

Specificity (or **True negative rate** (TNR)) measures the fraction of negative samples in the data that are correctly classified as negative.

$$\text{specificity} = \frac{\text{number of true negatives}}{\text{number of true negatives} + \text{number of false positives}} = \frac{TN}{TN + FP}$$

Precision (or **Positive preditive value** (PPV)) measures the fraction of samples classified as positive that are correctly classified.

$$\text{precision} = \frac{\text{number of true positives}}{\text{number of true positives} + \text{number of false positives}} = \frac{TP}{TP + FP}$$

2.8.1 Accuracy

The accuracy is a well known measure of performance, but not always a good one. It's simply the fraction of all samples that were correctly classified. Using the terms above we have

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

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 in 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

	Predicted Cat	Predicted Dog	Predicted Rabbit
True Cat	**5**	3	0
True Dog	2	**3**	1
True Rabbit	0	2	**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 j but should've predicted class i. 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 filtered 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 we recommend Fawcett[11].

2.8.3 F1-Score

The F1 score is a also a measure of accuracy of the model, but it accounts for more than regular accuracy. It is defined as

$$F_1 = 2 \cdot \frac{\text{precision} \cdot \text{recall}}{\text{precision} + \text{recall}} = \frac{TP}{TP + \frac{1}{2}(FP + FN)},$$

which is the harmonic mean of precision and recall. Why is this better than accuracy? By including both precision and recall, you are combining two metrics that both tell you something about how good the model is at classifying your "positive" class.

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. Summarized, a few key goals nuclear physics aims to achieve are

- 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

Before we delve deeper into the experiment, we outline the nomenclature and terms used. 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

There are several types of decays that can occur within nuclei. Among them we find:

- Beta decay - β
- Photon decay - γ
- Alpha decay - α (^{4}He)
- Proton/Neutron decay
- Gamma ray
- Internal conversion electron

This experiment focuses on beta decay, photon decay and internal conversion. Decays can also occur as chains of decays before a nucleus reaches a stable state. When measuring decay spectroscopy, there are three primary pieces of information that we are interested in:

- energy - gives an idea of the energy difference between the initial and final state for decays with characteristic energies.
- half-lives - what is the time between when a state was populated and when it was depopulated?
- branching ratios - when there is a choice of final states, we want to know which states are preferentially populated.

For these experiments the focus is neutron-rich nuclei. Some conventions and units used in the experiments are

- Time - can range between picoseconds ($10^{-12}s$) to years.
  - $t_{1/2}$ - half-life
  - λ - decay constants
  - τ - lifetime
- Energy - electron Vol (eV) - the energy needed to move an electron across a potential difference of one volt.
- Branching ratios - given as the percentage chance of the state reaching each of the possible final states.

Beta Decay

The majority of nuclei decay through beta decay. There are three types:

$$\beta^- : {}^{\mathrm{A}}_{\mathrm{Z}}\mathrm{Z}_{\mathrm{N}} \rightarrow {}_{\mathrm{Z}+1}{}^{\mathrm{A}}\mathrm{Y}_{\mathrm{N}-1} + e^- + \nu + Q_\beta \tag{2.57}$$

$$\beta^+ : {}^{\mathrm{A}}_{\mathrm{Z}}\mathrm{Z}_{\mathrm{N}} \rightarrow {}_{\mathrm{Z}-1}{}^{\mathrm{A}}\mathrm{Y}_{\mathrm{N}+1} + e^+ + \nu + Q_\beta \tag{2.58}$$

$$EC : {}^{\mathrm{A}}_{\mathrm{Z}}\mathrm{Z}_{\mathrm{N}} + e^- \rightarrow {}_{\mathrm{Z}-1}{}^{\mathrm{A}}\mathrm{Y}_{\mathrm{N}+1} + \bar{\nu} + Q_\beta \tag{2.59}$$

where EC is Electron Capture. These three decays either turn protons into neutrons, or vice versa. In every case there are neutrinos (ν) being emitted.

Gamma-ray decay

Decay by photon emission, where a nucleus in some excited state J_i^π with energy E_i decays to a lower energy state J_f^π with energy E_f, through the emission of a photon with energy E_γ. Each state J is characterized by a specific spin and parity. Gamma-rays must carry away at least one unit of angular momentum, L, to accomplish such a transition. The amount of angular momentum able to be carried away is bounded according to

$$|J_i - J_f| \leq L \leq J_i + J_f \tag{2.60}$$

Tyically the lowest possible L dominates. We can also characterize these transitions in terms of their electric or magnetic character, given by

$$\Delta\pi(EL) = (-1)^L \tag{2.61}$$

$$\Delta\pi(ML) = (-1)^{L+1} \tag{2.62}$$

Gamma ray decay leads to a characteristic energy representative of the difference between the initial and final state. We want to measure transition rates. States always have the potential to decay either through photon or electron emission. Electron emission dominates at higher Zs and lower energies, but is in principle always a possibility.

Internal Conversion

The internal conversion process competes with gamma decay. Electromagnetic interactions between the nucleus and an atomic electron leads to the atomic electron being ejected from the atom. This happens without the emission of gamma rays. The energy of the emitted electron can be calculated, and is given by

$$E_{IC} = E_{\text{transition}} - E_{\text{electron binding energy}} \tag{2.63}$$

This will give a quantized set of possible energies, as the emitted electron can come from different shells in the atom.

2.9.1 Experiment

Small, inorganic $CeBr_3$ (Cerium Bromide) scintillator. The detector itself is abobut 3mm thick and a few cm in dimension perpendicular to the beam. Accellerated ions are stopped in the scintillator (implanted). The central detector observes ions and decays, of which decays can be any one of gamma rays, beta decay electrons, or internal conversion electrons.

- The detector is constructed as a 16x16 grid of what we will refer to as *pixels*. Note that the scintillator itself is not pixellated, but the Position-Sensitive Photomultiplier Tube (PSPMT) is.

- Ions identified event-by-event are implanted in the detector. Position (which pixel) and arrival time are recorded for all implanted ions.
- Some characteristic time later a decay is detected. Position and time of decays are recorded.
- Decays are correlated to ions using spatial and temporal information
- Time scales
  - Beta decay: $10^{-3}s$
  - Gamma decay: 10^{-15} to $10^{-3}s$
- Gamma rays coincident with decays are then associated with the decay of a particular ion

The ability to correlate decays to specific ions is dependent on the rate of ions implanted. Electrons are typically on the order of a few MeV. Because you need a fairly large amount of material to completely stop an electron with energies of 5-10MeV, what is actually detected is the energy left in the detector by the electron as it passes through.

2.9.2 Data

Examples of detector images generated by both simulations and experiments are shown in figure 2.3. For both types of data we have selected examples of single events, and double events that would be considered 'easy' and 'hard' to label correctly by eye. This difficulty of labelling or 'classifying' is expected to increase as the separation distance between the events decreases. This is especially apparent in the simulated examples, where the 'hard' or 'close' double events are near indistinguishable from the single event. Simulated data is generated using GEANT4 [12], and contains two million simulated single and double events, balanced at a 50/50 ratio. Positions of origin are uniformly distributed in the detector image, and event energies are uniformly distributed between 0 to 1MeV. Pixel intensities range from 0 to 10000.

The experimental data is taken from a recent beta-decay experiment. This data differs from the simulated data in some ways:

- The positions of events in the detector images are expected to follow a more gaussian distribution rather than the uniform distribution. This is due to the nature of the experiment and how the particle beam is formed.
- Energy fluctuations in the scintillator makes experimental data look more 'noisy' than simulated data.

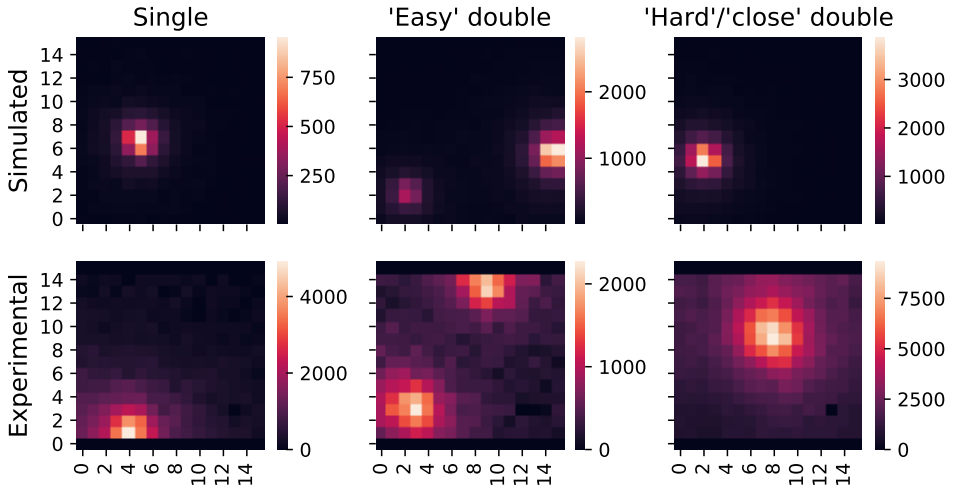

Figure 2.3: Sample images from simulated and experimental datasets. The top row contains simulated samples and the bottom row contains experimental samples. The labeling as "easy" and "hard" double events is based on experience from hand-labeling, and theoretical expectations.

- The ratio of single events to double events is not expected to be balanced. Rather, we expect to see a much larger amount of single events than double events.

- Some pixels in the experimental detector images will be set to zero if that pixel exhibits erratic behaviour. For some specific pixels this is true for the entire dataset. These pixels are the entire rows 0 and 15, plus pixel (13,3).

In our analysis of experimental data, we use two datasets. One set of 260147 decay events from a beta-decay experiments, and one set of 100000 decay events where we have two additional energy-related attributes available. These are DDAS[13]-energies and fit-energies. Both are measured using the PSPMT dynode signal. The DDAS energy is the energy provided by the onboard signal processing algorithms of the DDAS system. It is determined from the digitized detector signal using a trapezoidal filter. The fit energy is determined offline by analyzing digitized detector signals. The detector signal is modelled using a constant baseline plus a logistic rise convoluted with an exponential decay. This model signal is fit to the recorded pulse using standard nonlinear regression methods, and the fit energy is given by the amplitude of the best fit to the recorded detector signal.

The relationship between the arbitrary energy values determined by DDAS or the fitting and real energies define a set of calibration parameters which can

be compared to calibration parameters determined using source data. Ideally, the calibration parameters determined using machine learning models and the calibration parameters determined using source data should be the same or similar.

2.9.3 Simulated datasets

To explore the effect of some properties found in the experimental data, we make three simulated datasets. The first (a) without any modifications. The second (b) is designed to monitor the effects occurrences of 'dead' pixels in experimental data may have on model performance. When a pixel is unresponsive or displays otherwise erratic behaviour, the value of that pixel is set to zero in the given detector image. For some select pixels, this is the case for the entire dataset. The pixels are the entire top and bottom rows, and the pixel located at (13,3) when viewing an image as a 16x16 x,y-grid. The third dataset (c) is designed to monitor any possible bias imposed on the models when trained on a balanced dataset. As mentioned above, the expectation is that experimental data largely consists of single decays, with double decays being much more of a rarity. When training a model on a balanced dataset, we may impose a bias towards predicting a somewhat balanced amount of each class. Intentionally reducing the presence of one class in the dataset may provide some insight into whether or not this happens. In this case, we attempt to approximate the experimental data by reducing the presence of double decays to only 5% of the total number of decays.

Chapter 3

Method and Implementation

The programming language chosen for implementing experiments performed in this thesis is Python. Python has quickly grown to become one of the most popular languages overall, and especially in the machine learning community. Together with its extensive amount of available libraries, it allows for fast prototyping of solutions, and excellent readability. For machine learning purposes there are several libraries commonly used in the natural sciences. The code developed for data processing and analysis in the thesis is available in two GitHub repositories:

- https://github.com/geirtul/master_analysis contains scripts and notebooks used in analysis of the data, as well as trained models and experiment logs.
- https://github.com/geirtul/master_scripts contains an installable python module with data import scripts and various helper functions used by the analysis repository.

Additionally, for a briefer overview and introduction to analysis of beta-decay data using machine learning we have made a smaller example repository available at https://github.com/geirtul/event_classification_example.

3.1 TensorFlow

The TensorFlow[14] library is developed by Google, and is one of the most used libraries for machine learning in Python. It allows for designing complex learning algorithms efficiently, and also includes the Keras API[15], allowing easy-to-follow implementations of standard architectures and pipelines. Using this API, with TensorFlow as the backend framework, we build and train all models in this thesis.

3.2 Building and training a model

Building a machine learning model using TensorFlow and Keras is a straightforward process. By utilizing the `Sequential` class, we can stack the desired layers with given properties, and TensorFlow takes care of the rest.

```
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, InputLayer
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score, f1_score
from helper_functions import normalize_image_data
```

```
# Load images and labels.
DATA_PATH = "../data/"
images = np.load(DATA_PATH+"images_training.npy")
labels = np.load(DATA_PATH+"labels_training.npy")

# Dense layer requires one-dimensional inputs.
images = images.reshape(images.shape[0], 256)

# Split data into training and validation
x_idx = np.arange(images.shape[0])
train_idx, val_idx, not_used1, not_used2 = train_test_split(
    x_idx, x_idx, test_size = 0.25
)

# Initialize the Sequential model
model = Sequential()
# Add Input layer
model.add(InputLayer(input_shape=(images.shape[1],)))
# Add hidden layer
model.add(Dense(64, activation='relu'))
# Add output layer.
model.add(Dense(1, activation='sigmoid'))

# Finally, compile the model and print a summary.
# Loss function and optimizer is set during compilation.
model.compile(
    loss='binary_crossentropy',
    optimizer='adam',
    metrics=['accuracy']
)
model.summary()

# Set parameters for training
epochs = 20
batch_size = 32
# The validation_data keywork expects a 'tuple' (val_x, val_y)
# so we make one on the fly
history = model.fit(
    x=normalize_image_data(images[train_idx]),
    y=labels[train_idx],
    validation_data=(normalize_image_data(images[val_idx]),
        labels[val_idx]),
    epochs=epochs,
    batch_size=batch_size,
)

# Predict on the validation set
pred = model.predict([normalize_image_data(images[val_idx])])
# Convert sigmoid values from prediction to integers so it
    works with the function.
result = pred > 0.5
```

```
# Calculate the f1 score for evaluation
f1 = f1_score(labels[val_idx], result)
```

Listing 3.1: End-to-end example of building, training, and evaluating a model.

An end-to-end example of how to build, train and evaluate a model using a small sample of simulated data is shown in 3.1.

3.3 Pretrained network and feature extraction

A common strategy when working with representation- or transfer learning, is to use a *pretrained* model. State-of-the-art (SOTA) models typically require enormous hardware resources and considerable time to train, but simply passing inputs through them without any backpropagation does not. They are trained on millions of inputs, and we seek to exploit their ability to generalize. The SOTA models serve as feature extractors, and we train our own, more specialized models to make predictions from the extracted features. Many such models are available through the Keras API, and we have used parts of VGG16 [16] as part of this feature extraction. The steps involved are as follows:

- Initialize a `Sequential` model like shown in 3.1.
- Add an `InputLayer` suited to our inputs to the model. In our case this is (16, 16, 3) because VGG16 is trained on RGB images. Our images only have one channel, but we can create 'pseudo-channels' by concatenating our images along the last axis.
- Loop over layers in the pretrained VGG16 model, adding them one by one to our Sequential model until the maximum number of layers have been added. Due to `MaxPooling2D` layers, which cut the size of the inputs to the next layer in half, the number of layers we can use is lower than the number of layers in the full VGG16 model.
- Set each extracted layer's *trainable* attribute to False. We don't want to adjust weights in this large, complex network.
- Add a `Flatten` layer, which creates a one-dimensional vector of the extracted features.
- Add a desired number of `Dense` layers to build a top-level network which will take the extracted features as input.

We are essentially treating the SOTA model's layers as a good initialization for our final model.

Chapter 4

Results

The first task we face is whether or not we can use machine learning algorithms to separate two types of events in a simulated nuclear physics dataset. This being possible is the minimum requirement a model must meet if we are to apply it to experimental data. We present the results of training five different models on three simulated datasets, as described in section 2.9.2). Performance is measured using the $F1$ score and confusion matrix. The models range from the simple logistic regressor, to a deeper CNN architecture with multiple layers. Additionally, we include a model based on VGG16, as outlined in section **??**, fine-tuned on simulated data. Somewhat separate from classification, our second objective is predicting positions of origin and energies associated with events in the dataset. We follow the same strategy as for classification, with the logistic model replaced by a linear regressor. The same minimum requirement is valid for regression - models trained on simulated data must be able to make reasonable predictions. By 'reasonable' we mean 'better than random guessing'. For this task the performance is measured using the $R2$ score. The variability in results is estimated using a K-fold cross-validation approach, with $K = 5$ [17]. As a quick recap, the three simulated datasets mentioned are:

a) No changes.

b) Select pixels set to zero throughout the dataset.

c) Select pixels set to zero throughout the data, and an imbalanced number of single and double decays (reduced amount of double decays).

The machine learning experiments conducted in this thesis were performed using the AI-Hub computational cluster at the University of Oslo. This resource consists of three machines with four RTX 2080 Nvidia GPU's (graphics processing unit) each. These cards have 10GB of memory available for the allocation of models.

4.1 Preliminary analysis

We begin by looking into the simulated data, more specifically looking for correlations in the energies and pixel intensities. Note that these results are generated using dataset (a). In figure 4.1 we show the correlation matrix for simulated single and double decays. For single decays, there is a strong correlation between the energy of the event, the sum of intensities in an image, and also between the event energy and the intensity of the highest intensity pixel. The correlation matrix for double events shows similar results. The same, strong correlation is found for $E1 + E2$ in double decays as for $E1$ in

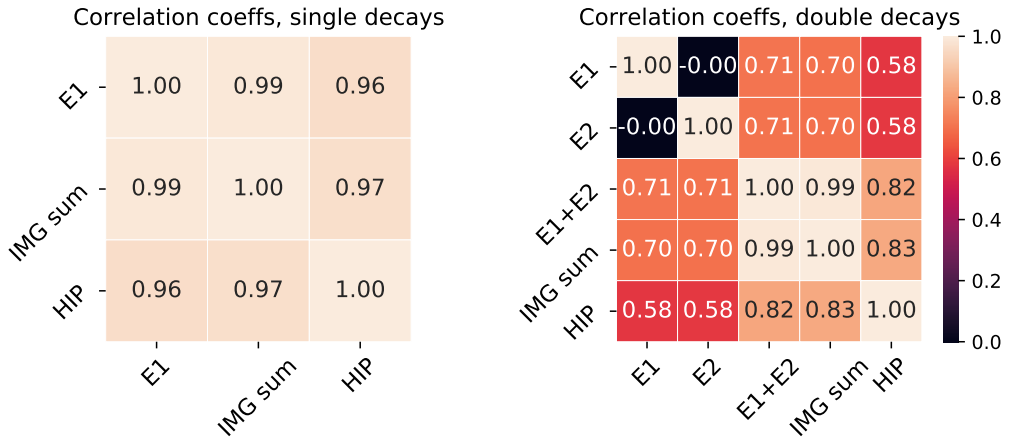

Figure 4.1: Correlation matrices for simulated data, separated into single and double decays. $E1$ and $E2$ are the energies corresponding to event 1 and event 2 in simulated data. For single events there is no event 2. $Sum(image)$ is the sum of all intensities per image in the dataset. HIP is short for Highest Intensity Pixel value.

single decays. That is, the total energy in an event is strongly correlated with the total intensity of pixels in the image.
Next, we investigate some directly comparable quantities shared by simulated and experimental data. The distributions of total intensity and highest intensity pixels in images are shown in figure 4.2. To relate these distributions more closely to the models themselves, the distributions are generated after normalization of the images. As such, these distributions are what the models 'see'. Looking to the top left in figure 4.2 there are a fairly large amount of experimental decays with higher total intensity than what is present in the simulated data. In the bottom left plot, it is also clear that there is a point where simulated single and double decays no longer overlap in total intensities. For the right-hand plots of highest intensity values, there is no such clear difference between the datasets. To provide another point of view for this difference, we plot the total intensity in images as a function of the highest intensity in the images. The plot is shown in figure 4.3, along with linear fits to the data. There is a clear difference between the datasets in that the experimental data has a higher total intensity. We will come back to these results as we review model performance on experimental data.

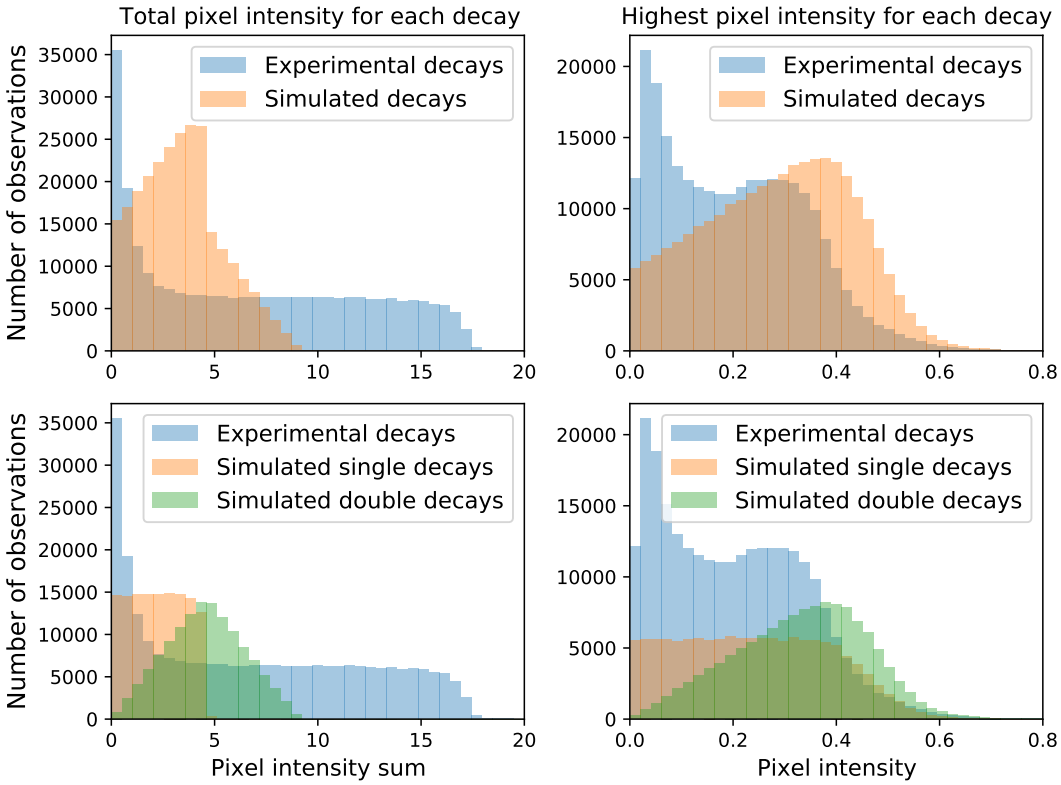

Figure 4.2: Distributions of total pixel intensities and highest intensities in experimental and simulated decays. Top row compares experimental decays and all simulated decays. The bottom row shows the same distributions, but with simulated decays split into single and double events. The calculations are done post normalization, so the maximum possible intensity is 1.0.

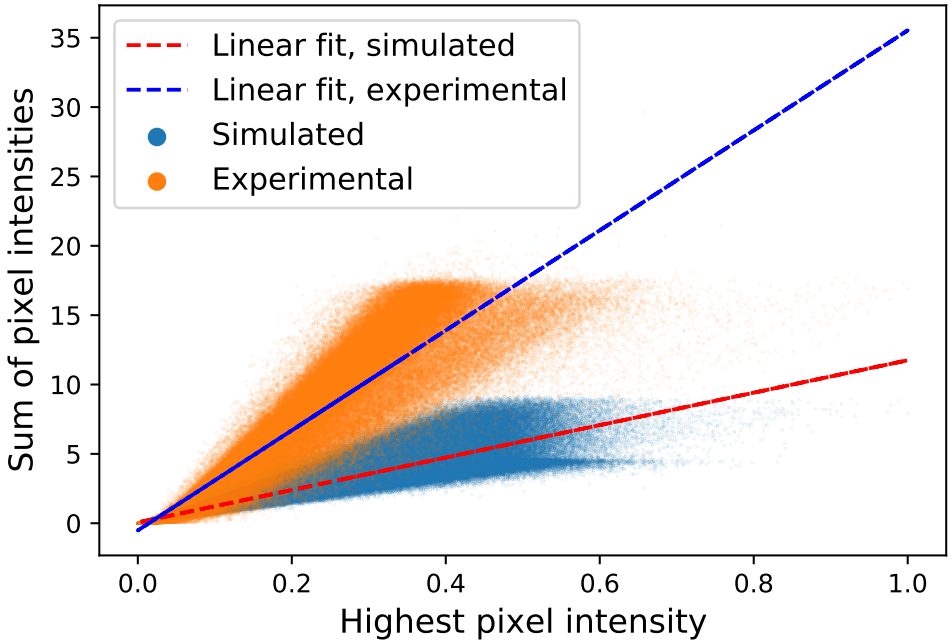

Figure 4.3: Scatterplot of sum of intensities in images vs the highest intensity in the same image, for both experimental and simulated decays. Linear fits give slopes of $a_{experimental} = 36.05$ and $a_{simulated} = 11.69$.

4.2 Classification

4.2.1 Classification on simulated data

In table 4.1 the performance of each model is reported through the estimated $F1$-score, for each of the datasets. As a benchmark for transfer learning, we are including a model based on a state of the art pretrained network[16], as outlined in section 3.3. In in figure 4.4, we show the confusion matrix for prediction on test set data for all the models, including normalized values for each event type. . The F1-scores show decreasing performance for most models when applying modifications to the unmodified simulated dataset. When training on an intentionally imbalanced dataset the models without CNN architectures show a steep decrease in performance. Considering the confusion matrix we see that the logistic regressor and dense network suffer from predicting mostly every sample to be single events. Looking back to the $F1$-scores again, there is not a significant increase in performance between the CNN and the Custom model on unmodified data, but the custom model performs strongly across all datasets, with a low amount of misclassified events relative to the other models. Again looking at the confusion matrices (figure 4.4), Double decays are more often misclassified as single than the opposite. Next, we apply the 'classifiers' to experimental data.

4.2.2 Classification on experimental data

Classification of experimental data poses a different set of challenges when it comes to evaluating our results. We currently only have a small number of events that are hand labelled as double events, which may be used as a form of verification. As mentioned in section 2.9, we expect the number of double events in the experimental data to be much lower than single events. Inspect-

Table 4.1: Test set F1-scores for classification of simulated datausing multiple models. Models are trained on a) unmodified data, b) data where specific pixels are set to zero to mimic 'dead' pixels in experimental data, and c) same as b) and imbalanced to mimic experimental data. Error estimates are the standard deviation in results from k-fold cross-validation with $K = 5$ folds.

	Logistic	Dense	CNN	Pretrained	Custom
F1-score (a)	0.738 $\pm 7.528 \times 10^{-3}$	0.84 $\pm 3.837 \times 10^{-2}$	0.917 $\pm 1.475 \times 10^{-2}$	0.97 $\pm 4.348 \times 10^{-1}$	0.969 $\pm 2.374 \times 10^{-2}$
F1-score (b)	0.733 $\pm 2.298 \times 10^{-3}$	0.732 $\pm 7.453 \times 10^{-2}$	0.796 $\pm 2.310 \times 10^{-2}$	0.966 $\pm 3.070 \times 10^{-3}$	0.932 $\pm 7.587 \times 10^{-3}$
F1-score (c)	0.294 $\pm 8.437 \times 10^{-2}$	0.302 $\pm 8.949 \times 10^{-2}$	0.831 $\pm 1.535 \times 10^{-2}$	0.928 $\pm 3.948 \times 10^{-2}$	0.968 $\pm 1.238 \times 10^{-1}$

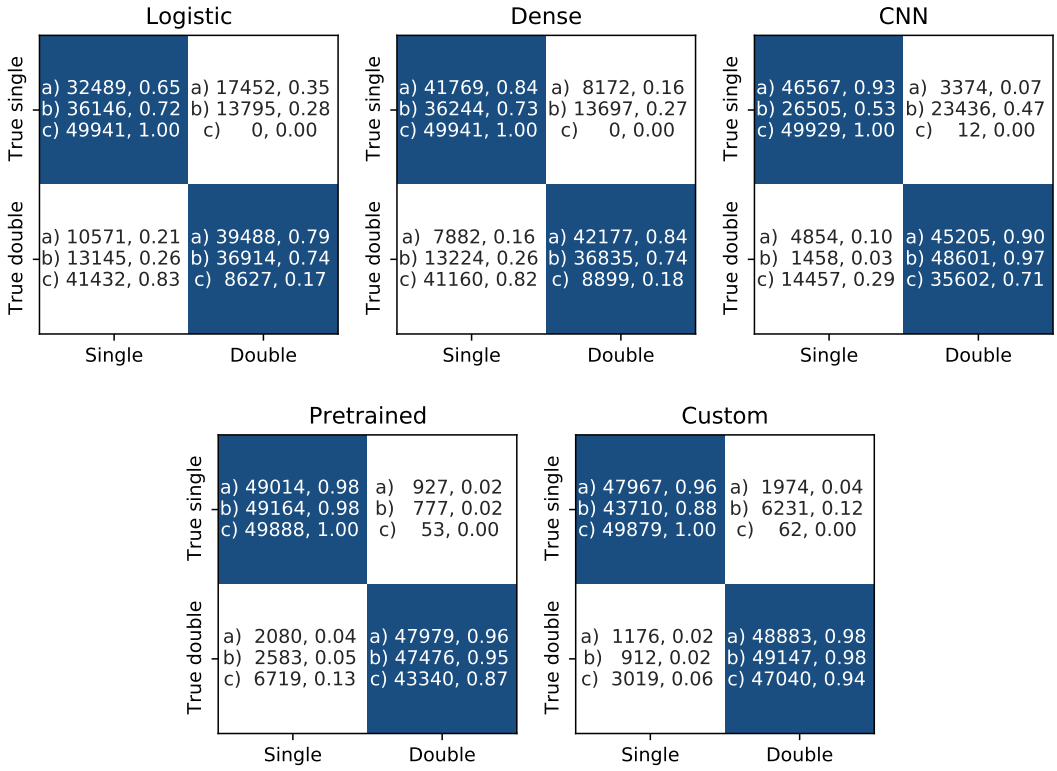

Figure 4.4: Confusion matrices for each model trained on simulated data. For each model and dataset, the number of events and ratio of each event type are given. a) unmodified data. b) select pixels set to zero. c) Same as in b) with the intentionally imbalanced.

Table 4.2: Decay event classification on experimental data, with models trained on: a) unmodified data, b) data where specific pixels are set to zero to mimic 'dead' pixels in experimental data, and c) same as b) and imbalanced to mimic experimental data. The numbers are shown as the normalized ratio of predicted event type, with the actual amount of events predicted of that type below.

	Single (a)	Double (a)	Single (b)	Double (b)	Single (c)	Double (c)
Logistic	0.367 95 565	0.633 164 582	0.375 97 629	0.625 162 518	0.537 139 578	0.463 120 569
Dense	0.357 92 920	0.643 167 227	0.375 97 498	0.625 162 649	0.536 139 405	0.464 120 742
CNN	0.240 62 359	0.760 197 788	0.213 55 512	0.787 204 635	0.226 58 870	0.774 201 277
Pretrained	0.207 53 837	0.793 206 310	0.186 48 449	0.814 211 698	0.225 58 534	0.775 201 613
Custom	0.078 20 219	0.922 239 928	0.069 17 859	0.931 242 288	0.113 29 441	0.887 230 706

ing the ratio of predicted singles to predicted doubles can then be an initial indication of how a model is performing. It is, however, not conclusive. Correctly classified hand labelled events are another indication, but is also not conclusive. In table 4.2, the ratios of predicted singles to predicted doubles are presented for each model trained on each dataset. The actual number of predictions for each class are included below the ratios. Overall there is a strong preference for classifying events as double decays. This makes the validation using hand labelled doubles in table 4.3 somewhat moot, since it is hard to attribute these 'correct' classifications to the models' ability to recognize the double decays. As the number of events is large, manual inspection of each predicted class is not feasible. In figure 4.5 we show the fraction of predicted single events in experimental data as a function of total intensity in each image. Regardless of which simulated dataset a model is trained on, the majority of single events are predicted at low total intensites. The only exception to this is the Dense model trained on dataset *c*. Keep in mind, however, that the total intensities for experimental data span from 0 to 18 (see figure 4.2). To further prod this apparent trend, we perform a simple test using the Custom model trained on dataset *b*. For a set of single events for which we know the model has good performance, we multiply the image intensities with a scaling factor from 0-10. The aim is to see the effect of increasing total intensity in images on the classification accuracy. The result is shown in figure 4.6. From this figure, there is a clear trend towards a lower fraction of events correctly classified as single decays when the intensities in images increases. Note that at a scaling factor of 4 the total intensities in the simulated images approach the highest total intensities in the experimental decay data.

Table 4.3: Decay event classification on 17 labeled samples of experimental data. The 17 samples are all labeled as double events. Models are trained on simulated data with a varying degree of modification: a) unmodified data, b) data where specific pixels are set to zero to mimic 'dead' pixels in experimental data, and c) same as b) and imbalanced to mimic experimental data. The numbers are shown as the normalized ratio of predicted event type, with the actual amount of events predicted of that type below.

	Single (a)	Double (a)	Single (b)	Double (b)	Single (c)	Double (c)
Logistic	0.000 0	1.000 17	0.000 0	1.000 17	0.000 0	1.000 17
Dense	0.000 0	1.000 17	0.000 0	1.000 17	0.000 0	1.000 17
CNN	1.000 17	0.000 0	0.000 0	1.000 17	0.235 4	0.765 13
Pretrained	0.000 0	1.000 17	0.000 0	1.000 17	0.000 0	1.000 17
Custom	0.000 0	1.000 17	0.000 0	1.000 17	0.000 0	1.000 17

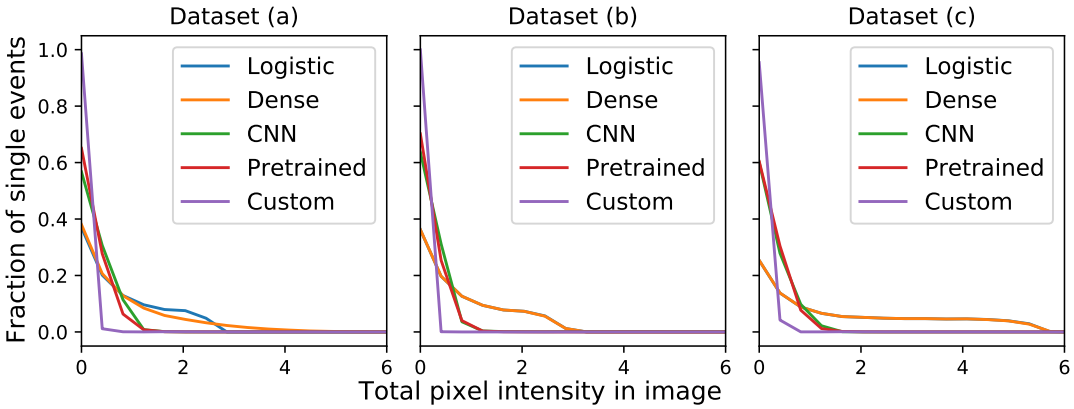

Figure 4.5: Fraction of predicted singles as a function of total intensity in images, for each model trained on simulated data. a) unmodified data. b) select pixels set to zero. c) Same as in b) with the intentionally imbalanced.

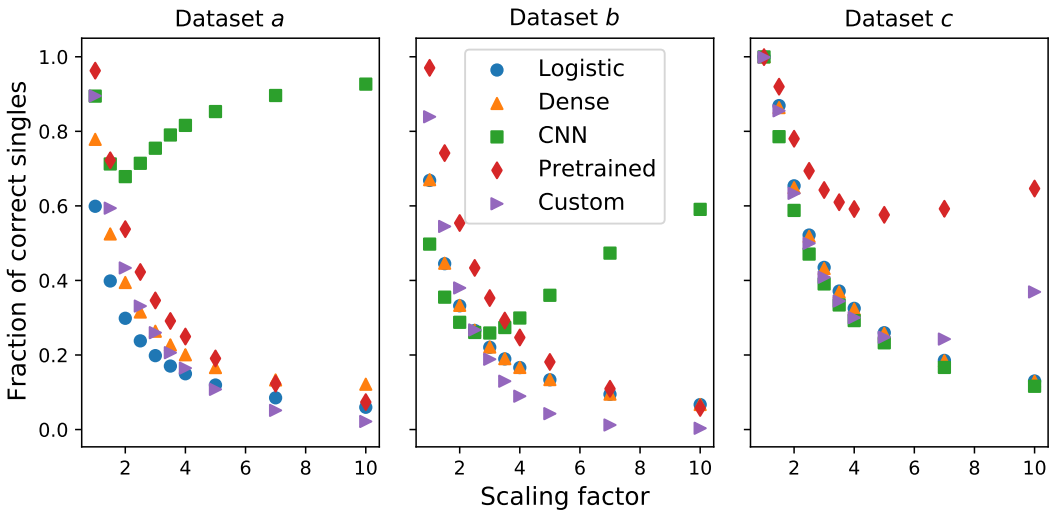

Figure 4.6: Fraction of simulated single events correctly classified as a function of scaling factor used to increase the total intensity in simulated decays. In the title of each plot is the dataset the models were trained on. The dataset used for prediction is dataset *a*.

4.3 Regression

Assuming data has already been classified, we aim to predict the energy of the decays and their position of origin. Because there is a travel distance between the ejection site and the point the energy is deposited, the positions aren't necessarily the locations of the highest-intensity pixels in the detector images. Our hypothesis is then that there are sufficient structures and spacial relationships in the detector images to allow a model to determine these positions. Note that for regression the models are trained on single or double events exclusively. A consequence of this is that for dataset *c* the set of single events is identical to that of set *b*, causing near-identical results between these two sets.

4.3.1 Position of origin

In table 4.4 we present the R2-scores for all models trained on simulated data. All but the simple linear regressor perform excellently. Notably, there is little decrease in $R2$ scores when modifying the training data (datasets *b* and *c*). In table 4.5 we show the mean errors in *mm*. Here we do see a decrease in performance from added modification to training data. The pretrained model sticks out as the more stable of the architectures, and is overall the highest performing model. All the CNN architectures predict with sub-pixel accuracy. In the case of regression on double events, none of the models accurately predict

Table 4.4: Test set R2-scores for regression of positions of origin on simulated data, with models trained on data with: a) no modifications, b) specific pixels set to zero to mimic experimental data, and c) imbalanced dataset in addition to modifications in b) to further mimic experimental data. Error estimates are the standard deviation in results from validation data in k-fold cross-validation with $K = 5$ folds.

	Linear	Dense	CNN	Pretrained	Custom
Single (a)	0.8 $\pm 2.889 \times 10^{-3}$	0.988 $\pm 7.007 \times 10^{-4}$	0.997 $\pm 9.207 \times 10^{-4}$	0.997 $\pm 2.229 \times 10^{-1}$	0.999 $\pm 1.366 \times 10^{-4}$
Single (b)	0.781 $\pm 2.749 \times 10^{-3}$	0.982 $\pm 1.406 \times 10^{-3}$	0.98 $\pm 1.110 \times 10^{-3}$	0.997 $\pm 4.513 \times 10^{-4}$	0.995 $\pm 9.603 \times 10^{-4}$
Single (c)	0.781 $\pm 2.749 \times 10^{-3}$	0.982 $\pm 1.424 \times 10^{-3}$	0.98 $\pm 1.080 \times 10^{-3}$	0.997 $\pm 4.932 \times 10^{-4}$	0.993 $\pm 1.639 \times 10^{-3}$
Double (a)	0.37 $\pm 3.766 \times 10^{-3}$	0.456 $\pm 5.601 \times 10^{-3}$	0.471 $\pm 1.603 \times 10^{-3}$	0.29 $\pm 1.552 \times 10^{-1}$	0.493 $\pm 3.467 \times 10^{-4}$
Double (b)	0.364 $\pm 6.815 \times 10^{-4}$	0.458 $\pm 3.431 \times 10^{-3}$	0.435 $\pm 1.835 \times 10^{-3}$	0.289 $\pm 1.550 \times 10^{-1}$	0.489 $\pm 2.865 \times 10^{-4}$
Double (c)	0.357 $\pm 7.768 \times 10^{-3}$	0.417 $\pm 9.456 \times 10^{-3}$	0.442 $\pm 2.507 \times 10^{-3}$	-0.924 $\pm 8.452 \times 10^{-1}$	0.478 $\pm 4.187 \times 10^{-3}$

Table 4.5: Test set mean error converted to mm for regression of positions of origin on simulated data, with models trained on data with: a) no modifications, b) specific pixels set to zero to mimic experimental data, and c) imbalanced dataset in addition to modifications in b) to further mimic experimental data. Error estimates are the standard deviation in results from validation data in k-fold cross-validation with $K = 5$ folds.

	Linear	Dense	CNN	Pretrained	Custom
Single (a) [mm]	5.68 $\pm 6.427 \times 10^{-1}$	1.36 $\pm 3.382 \times 10^{-1}$	0.699 $\pm 3.843 \times 10^{-1}$	0.676 ± 5.990	0.378 $\pm 1.477 \times 10^{-1}$
Single (b) [mm]	5.95 $\pm 6.824 \times 10^{-1}$	1.7 $\pm 4.791 \times 10^{-1}$	1.81 $\pm 4.221 \times 10^{-1}$	0.707 $\pm 2.694 \times 10^{-1}$	0.904 $\pm 3.938 \times 10^{-1}$
Single (c) [mm]	5.95 $\pm 6.824 \times 10^{-1}$	1.7 $\pm 4.820 \times 10^{-1}$	1.79 $\pm 4.164 \times 10^{-1}$	0.723 $\pm 2.817 \times 10^{-1}$	1.03 $\pm 5.147 \times 10^{-1}$
Double (a) [mm]	10.1 $\pm 7.746 \times 10^{-1}$	9.37 $\pm 9.506 \times 10^{-1}$	9.25 $\pm 5.069 \times 10^{-1}$	10.7 ± 5.002	9.05 $\pm 3.342 \times 10^{-1}$
Double (b) [mm]	10.1 $\pm 3.748 \times 10^{-1}$	9.36 $\pm 7.344 \times 10^{-1}$	9.55 $\pm 5.014 \times 10^{-1}$	10.7 ± 4.999	9.09 $\pm 2.491 \times 10^{-1}$
Double (c) [mm]	10.2 ± 1.299	9.7 ± 1.345	9.49 $\pm 8.811 \times 10^{-1}$	17.6 $\pm 1.171 \times 10^{1}$	9.18 ± 1.033

Table 4.6: Mean distances of predicted position of origin on experimental decays to the center of highest intensity pixel (HIP).Models trained on data with: a) no modifications, b) specific pixels set to zero to mimic experimental data, and c) imbalanced dataset in addition to modifications in b) to further mimic experimental data.

	Mean distance [mm]	Standard Deviation [mm]
LinReg (a)	17.61	9.19
Dense (a)	16.89	13.06
CNN (a)	12.53	7.98
Pretrained (a)	4.59	3.31
Custom (a)	3.88	3.04
LinReg (b)	17.49	8.93
Dense (b)	13.80	10.40
CNN (b)	12.49	8.73
Pretrained (b)	4.80	2.99
Custom (b)	3.50	3.02
LinReg (c)	17.49	8.93
Dense (c)	13.79	10.40
CNN (c)	12.40	8.71
Pretrained (c)	4.66	2.94
Custom (c)	3.48	2.91

positions of origins for both events. The models predict positions on average 9 *mm* off, which is roughly equal to three pixel-widths. In the absence of true positions for experimental decays, we rely on other ways to estimate how regression models perform. In figure 4.7 we look at how the predicted positions are distributed around the highest intensity pixel in each experimental decay event. As a comparison, we plot the same distribution using true values for positions in simulated single decay events. We can see that the distributions overlap quite well up until their respective peaks. Predictions on experimental decays have a wider distribution than the target data, but as we've seen in the classification results this difference in distributions is not a unique case.

4.3.2 Energy

In table 4.7 we show the R2-scores for all models trained on simulated data. Performance when predicting single energies is across the board lower than what we saw for positions of origin. On unmodified data, the models are to a large degree able to predict energies, with R2-scores of 0.93 and above. For the modified datasets the CNN suffers greatly and isn't able to account for variances in the data at all. Other models see a less severe effect. In general,

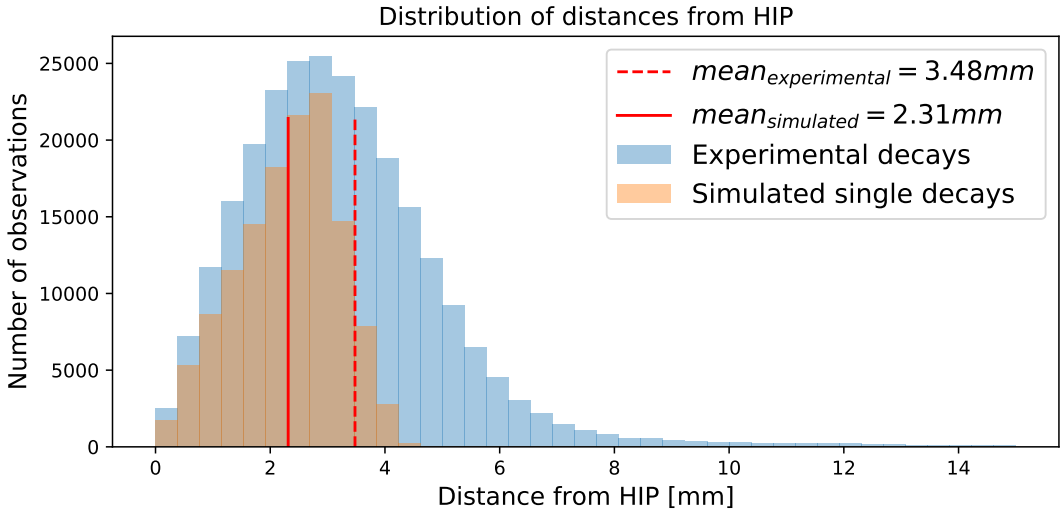

Figure 4.7: Distribution of distances between predicted positions of origin and the highest intensity pixel(HIP) in the corresponding images. The model was a custom cnn architecture, trained on dataset *c*. The simulated single decay distribution is generated with the true potitions of the events, not predicted ones.

the degree of explained variance is considerably reduced when 'dead' pixels are introduced. The same effect is seen in the mean errors shown in table 4.8. With no modifications to the training data, the custom model predicts energies with an error of $0.07MeV$, with the other models close behind. For modified data and double events the error is doubled or more.

Due to the poor performance of models trained on simulated double events, and the expected low frequency of double events in the experimental data, we only apply models trained on simulated single events. From figure 4.1, we know that total image intensity and energy are closely correlated for simulated data, and from $R2$ scores shown in table 4.7 that goodness of fit decreases with added 'dead' pixels. Without true values for energies in experimental data, we show in figure 4.8 the related quantities for experimental decays. Total image intensity decreases with increasing number of pixels with value zero, as expected. The predicted energy follows this same trend very closely. The 'peaks' in the plots correspond to events where there are a fairly large amount of 'dead' pixels, but the decay was registered in a region of the detector with few such pixels. We also calculate the correlation coefficients between total image intensity and predicted energy for all models applied to experimental data. For all models, the correlation coefficients range from 0.95 to 0.999.

Table 4.7: Test set R2-scores for regression of energies on simulated data, with models trained on data with: a) no modifications, b) specific pixels set to zero to mimic experimental data, and c) imbalanced dataset in addition to modifications in b) to further mimic experimental data. Error estimates are the standard deviation in results from validation data in k-fold cross-validation with $K = 5$ folds.

	Linear	Dense	CNN	Pretrained	Custom
Single (a)	0.932 $\pm 3.334 \times 10^{-2}$	0.934 $\pm 3.623 \times 10^{-2}$	0.937 $\pm 4.088 \times 10^{-2}$	0.926 $\pm 3.761 \times 10^{-2}$	0.944 $\pm 2.997 \times 10^{-2}$
Single (b)	0.768 $\pm 2.459 \times 10^{-2}$	0.745 $\pm 2.222 \times 10^{-2}$	0.48 $\pm 2.575 \times 10^{-2}$	0.781 $\pm 1.948 \times 10^{-2}$	0.752 $\pm 3.167 \times 10^{-2}$
Single (c)	0.768 $\pm 2.459 \times 10^{-2}$	0.745 $\pm 2.223 \times 10^{-2}$	0.432 $\pm 2.522 \times 10^{-2}$	0.781 $\pm 1.955 \times 10^{-2}$	0.724 $\pm 2.956 \times 10^{-2}$
Double (a)	0.49 $\pm 3.349 \times 10^{-2}$	0.49 $\pm 3.084 \times 10^{-2}$	0.488 $\pm 4.130 \times 10^{-2}$	0.489 $\pm 3.138 \times 10^{-2}$	0.491 $\pm 3.618 \times 10^{-2}$
Double (b)	0.485 $\pm 3.157 \times 10^{-3}$	0.487 $\pm 2.347 \times 10^{-3}$	0.478 $\pm 7.096 \times 10^{-3}$	0.489 $\pm 4.508 \times 10^{-3}$	0.464 $\pm 3.659 \times 10^{-3}$
Double (c)	0.434 $\pm 4.611 \times 10^{-2}$	0.422 $\pm 4.583 \times 10^{-2}$	0.446 $\pm 4.554 \times 10^{-2}$	0.417 $\pm 3.868 \times 10^{-2}$	0.401 $\pm 4.802 \times 10^{-2}$

Table 4.8: Test set mean error converted to mm for regression of energies on simulated data, with models trained on data with: a) no modifications, b) specific pixels set to zero to mimic experimental data, and c) imbalanced dataset in addition to modifications in b) to further mimic experimental data. Error estimates are the standard deviation in results from validation data in k-fold cross-validation with $K = 5$ folds.

	Linear	Dense	CNN	Pretrained	Custom
Single (a) [MeV]	0.0756 $\pm 5.276 \times 10^{-2}$	0.0742 $\pm 5.499 \times 10^{-2}$	0.0725 $\pm 5.841 \times 10^{-2}$	0.0789 $\pm 5.603 \times 10^{-2}$	0.0683 $\pm 5.001 \times 10^{-2}$
Single (b) [MeV]	0.139 $\pm 4.519 \times 10^{-2}$	0.146 $\pm 4.296 \times 10^{-2}$	0.209 $\pm 4.624 \times 10^{-2}$	0.135 $\pm 4.021 \times 10^{-2}$	0.144 $\pm 5.129 \times 10^{-2}$
Single (c) [MeV]	0.139 $\pm 4.519 \times 10^{-2}$	0.146 $\pm 4.296 \times 10^{-2}$	0.218 $\pm 4.576 \times 10^{-2}$	0.135 $\pm 4.029 \times 10^{-2}$	0.152 $\pm 4.955 \times 10^{-2}$
Double (a) [MeV]	0.206 $\pm 5.286 \times 10^{-2}$	0.206 $\pm 5.074 \times 10^{-2}$	0.207 $\pm 5.862 \times 10^{-2}$	0.207 $\pm 5.116 \times 10^{-2}$	0.206 $\pm 5.496 \times 10^{-2}$
Double (b) [MeV]	0.207 $\pm 1.613 \times 10^{-2}$	0.207 $\pm 1.421 \times 10^{-2}$	0.209 $\pm 2.393 \times 10^{-2}$	0.207 $\pm 1.992 \times 10^{-2}$	0.212 $\pm 1.684 \times 10^{-2}$
Double (c) [MeV]	0.217 $\pm 6.176 \times 10^{-2}$	0.22 $\pm 6.157 \times 10^{-2}$	0.215 $\pm 6.134 \times 10^{-2}$	0.221 $\pm 5.645 \times 10^{-2}$	0.223 $\pm 6.307 \times 10^{-2}$

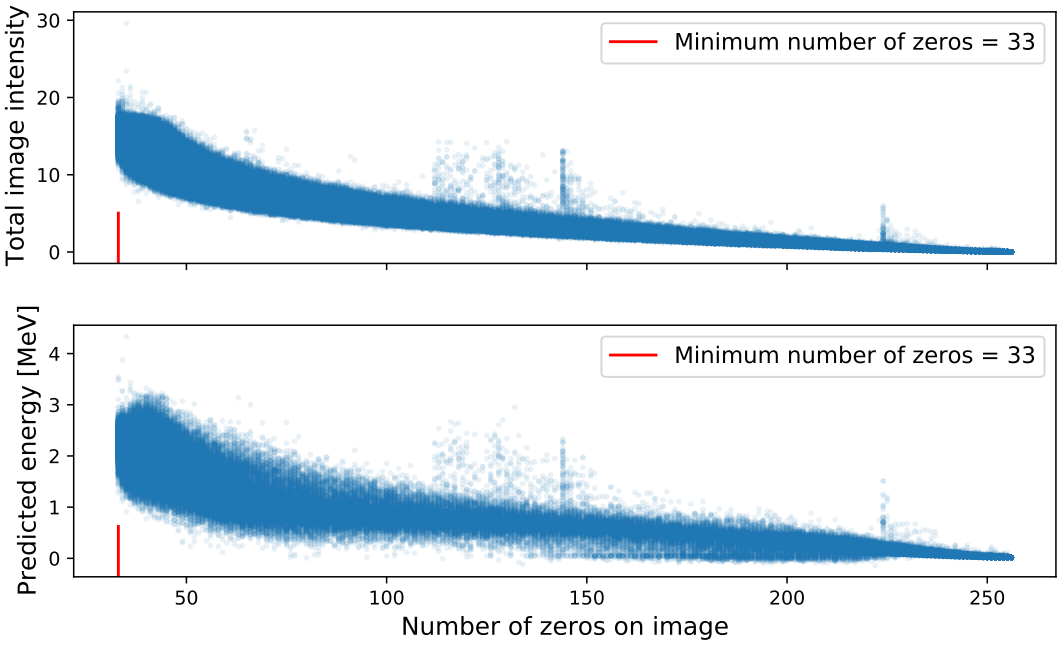

Figure 4.8: Top: Total image intensities as a function of number of pixels with intensity zero in images. Bottom: Predicted energy for experimental decays events as a function of number of pixels with intensity zero in images. The model used for predictions is the Custom architecture model trained on dataset *b*.

Table 4.9: Slopes from 1st order polynomial fit of predicted energies vs. DDAS energies and predicted energies vs. fit energies. *a*, *b*, and *c* specify which simulated dataset the models were trained on. Keep in mind that all these models were trained exclusively on single decay events, thus making dataset *b* and *c* identical in which samples models are trained on.

	Linear	Dense	CNN	Pretrained	Custom
a_{DDAS} (a)	9.12×10^{-5}	8.90×10^{-5}	9.30×10^{-5}	3.63×10^{-5}	8.80×10^{-5}
a_{DDAS} (b)	8.40×10^{-5}	8.16×10^{-5}	8.47×10^{-5}	3.20×10^{-5}	6.40×10^{-5}
a_{DDAS} (c)	8.40×10^{-5}	8.15×10^{-5}	8.53×10^{-5}	3.20×10^{-5}	1.02×10^{-4}
a_{fit} (a)	2.75×10^{-4}	2.69×10^{-4}	2.81×10^{-4}	1.10×10^{-4}	2.66×10^{-4}
a_{fit} (b)	2.53×10^{-4}	2.46×10^{-4}	2.55×10^{-4}	9.64×10^{-5}	1.93×10^{-4}
a_{fit} (c)	2.53×10^{-4}	2.46×10^{-4}	2.57×10^{-4}	9.64×10^{-5}	3.07×10^{-4}

By fitting straight lines,

$$\text{Predicted energy} = a_{\text{DDAS}} \times E_{\text{DDAS}} + b \tag{4.1}$$

$$\text{Predicted energy} = a_{\text{fit}} \times E_{fit} + b \tag{4.2}$$

we can compare the slopes a_{DDAS} and a_{fit} with calibration constants (see section 2.9.2) to determine if our models are predicting reasonable energies. The slopes from fitting a 1st order polynomial to the data are shown in table 4.9. Overall most models show similar slopes within training on the same dataset. The pretrained model, however, shows lower slopes in all cases.

Chapter 5

Discussion

5.1 Classification

With the goal of classifying experimental decay data, we set a minimum requirement of first showing that machine learning algorithms can classify simulated decay events. From the $F1$ scores shown in table 4.1 we can see that classification of simulated events is possible. There is a clear performance gap between CNN-based architectures and the logistic regressor and dense neural network. This gap may be expected, as CNN's are specifically designed for machine learning involving images. Of the CNN architechtures, the deeper architectures (Custom 7, Pretrained 8), show less decrease in performance when trained on datasets with 'dead' pixels and imbalanced representation of classes. When training models on an imbalanced dataset these deeper models also appear less prone to simply classify most of events as one class. In figure 4.4 we can see that this is the case for the logistic and dense models.

With the ability to classify simulated events as either single or double decays, we then applied the models trained on simulated data to experimental data. Lacking true labels for experimental data, we look to other expecations, such as the fraction of events predicted as single and double decays, shown in table 4.2. Our expectation is that there is a much larger amount of single decays present in experimental data than double decays. In light of this expectation, the models' performance is not very good. The models performing best on simulated data predict up to 90% or more of the experimental decays to be double decays. This is also the case for models trained on dataset c (imbalanced). In figure 4.2 we saw that there is a difference in total intensity between simulated decays and experimental decays. The experimental decays range higher in total intensity, and in figure 4.3 we also see that a higher maximum intensity in an experimental event corresponds to a higher total intensity than for simulated events. Additionally, the fraction of predicted single events as a function of total intensity in images which we show in figure 4.5, indicate that most single events are predicted for low total intensities in images. In fact, single events are predicted almost exclusively in the region of total intensity where single events are distributed in simulated data. We also concider the trend of decreased fraction of correctly classified single decays in simulated data, presented in figure 4.6. Together, this difference between simulated and experimental data may partially explain why a lot of events are classified as double decays.

5.2 Regression

5.2.1 Positions of origin

With the $R2$ scores shown in table 4.4, and mean errors from table 4.8, prediction of positions of origin seems highly successful for simulated single decays. The portion of explained has low variance with added modifications to the training data, although the mean error increases with added 'dead' pixels. This is to some degree to be expected, as this also degrades the information in the image. In spite of these modifications, models are still able to predict positions of origin with sub-pixel width accuracy for all simulated datasets. Without access to something akin to true positions for experimental decays, properly determining performance is not currently possible. Viewing the distributions of distances from the highest intensity pixel in figure 4.7 indicates at least that models predict reasonable positions. This distribution must be taken with a grain of salt, however. When the number of 'dead' pixels in images increase, the likelihood of the model seeing the 'true' highest intensity pixel decreases. That is, with increasing numbers of 'dead' pixels, the probability that the model is not predicting the position of the actual origin increases. Considering the mean distances and their standard deviations shown in table 4.6, it is clear that overall the pretrained and custom models outclass the others in this task. Note, however, the standard deviations in the mean distances from the highest intensity pixel are almost the size of the mean, indicating wider distributions.

5.2.2 Energy

Based on scores shown in table 4.7 and mean error from 4.8, prediction of energies from simulated events is indeed possible. With the energy being strongly correlated with total intensity in the detector images, we expect the introduction of 'dead' pixels to have a greater impact on energy prediction than for positions, and based on the $R2$ scores (table 4.7) this seems to be the case. This sensitivity to 'dead' pixels may be detrimental to prediction on experimental data. In figure 4.8 we see that the predicted energies on experimental decays vs number of 'dead' pixels closely follows the trend of total image intensity. This likely indicates that the strong correlation between total image intensity and energy found in simulated data is preserved through to predicted energies on experimental decays. This also means that when the number of pixels with value zero increases, it is crucial to know whether that is because of no detected energy or erratic behaviour. Otherwise you cannot determine whether the prediction is that of the event itself, or simply a function of the sum of the inputs. With the sharp decrease in explained variance seen in the $R2$-scores, energy prediction for experimental decays using models trained on simulated

data is likely not particularly successful in this case. This is further supported by the calculated slopes ('calibration constants') based on predicted energies shown in table 4.9. A current working hypothesis is that the simulation data used to train the models probably predicted too many photons per unit energy. This would partly explain good predicted energies for simulated data and poor predicted energies for the real data compared with calibration parameters based on source data.

Chapter 6

Conclusion

Without true positions for experimental data to directly test the models against, making any strong conclusions for these results is hard. Proper verification of performance will have to wait until researchers can test the predicted positions on source data from the experiment.

Appendices

.1 Model architectures

Here we show the base architectures for all models used in this thesis. All layers except final output layers use the ReLU activation function. The output layers depend on the task, as follows:

- Classification - Output uses the sigmoid function.
- Regression - Output has a linear activation.

Additionally, the final output layers for regression differ from those used in classification.

- Positions of origin, single events - Dense(2)
- Positions of origin, double events - Dense(4)
- Energy, single events - Dense(1)
- Energy, double events - Dense(2)

We also note that the compilation of models is different between these tasks. The compile code for models in classification is listed in listing 1, and for regression in listing 2. We include this as the loss function is defined at compile time for Sequential models.

```
model.compile(
    optimizer='adam',
    loss='binary_crossentropy',
    metrics=['accuracy']
)
```

Listing 1: Code for compiling classification models.

```
model.compile(
    optimizer='adam',
    loss='mse',
)
```

Listing 2: Code for compiling regression models.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 1)                 257
=================================================================
```

```
Total params: 257
Trainable params: 257
Non-trainable params: 0
```

Listing 3: Summary of the linear regression model architecture for regression.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 1)                 257
=================================================================
Total params: 257
Trainable params: 257
Non-trainable params: 0
```

Listing 4: Summary of the logistic model architecture for classification.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 8)                 2056
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 9
=================================================================
Total params: 2,065
Trainable params: 2,065
Non-trainable params: 0
_________________________________________________________________
```

Listing 5: Summary of the dense model architecture for classification.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 16, 16, 8)         80
_________________________________________________________________
flatten (Flatten)            (None, 2048)              0
_________________________________________________________________
dense (Dense)                (None, 1)                 2049
=================================================================
Total params: 2,129
Trainable params: 2,129
Non-trainable params: 0
```

Listing 6: Summary of the CNN model architecture for classification.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 16, 16, 32)        320
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 8, 8, 64)          0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 8, 8, 64)          36928
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 8, 8, 64)          36928
_________________________________________________________________
flatten (Flatten)            (None, 4096)              0
_________________________________________________________________
dense (Dense)                (None, 128)               524416
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 129
=================================================================
Total params: 617,217
Trainable params: 617,217
Non-trainable params: 0
```

Listing 7: Summary of the custom model architecture for classification.

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
block1_conv1 (Conv2D)        (None, 16, 16, 64)        1792
_________________________________________________________________
block1_conv2 (Conv2D)        (None, 16, 16, 64)        36928
_________________________________________________________________
block1_pool (MaxPooling2D)   (None, 8, 8, 64)          0
_________________________________________________________________
block2_conv1 (Conv2D)        (None, 8, 8, 128)         73856
_________________________________________________________________
block2_conv2 (Conv2D)        (None, 8, 8, 128)         147584
_________________________________________________________________
block2_pool (MaxPooling2D)   (None, 4, 4, 128)         0
_________________________________________________________________
block3_conv1 (Conv2D)        (None, 4, 4, 256)         295168
_________________________________________________________________
block3_conv2 (Conv2D)        (None, 4, 4, 256)         590080
_________________________________________________________________
block3_conv3 (Conv2D)        (None, 4, 4, 256)         590080
_________________________________________________________________
block3_pool (MaxPooling2D)   (None, 2, 2, 256)         0
```

```
_________________________________________________________________
block4_conv1 (Conv2D)        (None, 2, 2, 512)         1180160
_________________________________________________________________
block4_conv2 (Conv2D)        (None, 2, 2, 512)         2359808
_________________________________________________________________
block4_conv3 (Conv2D)        (None, 2, 2, 512)         2359808
_________________________________________________________________
block4_pool (MaxPooling2D)   (None, 1, 1, 512)         0
_________________________________________________________________
block5_conv1 (Conv2D)        (None, 1, 1, 512)         2359808
_________________________________________________________________
block5_conv2 (Conv2D)        (None, 1, 1, 512)         2359808
_________________________________________________________________
block5_conv3 (Conv2D)        (None, 1, 1, 512)         2359808
_________________________________________________________________
flatten_1 (Flatten)          (None, 512)               0
_________________________________________________________________
dense (Dense)                (None, 10)                5130
_________________________________________________________________
dense_1 (Dense)              (None, 1)                 11
=================================================================
Total params: 14,719,829
Trainable params: 14,719,829
Non-trainable params: 0
```

Listing 8: Summary of the pretrained model architecture for classification.

Bibliography

[1] A. Karpathy. Stanford University CS231n: convolutional neural networks for visual recognition. *URL: http://cs231n. stanford. edu/syllabus. html*, 2017. URL http://cs231n.stanford.edu/.

[2] M. Hjorth-Jensen. Lecture Notes - FYS4155 - Applied Data Analysis and Machine Learning, 2020. URL https://github.com/CompPhysics/MachineLearning.

[3] 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.

[4] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. URL http://www.deeplearningbook.org.

[5] 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.

[6] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997. ISSN 1089778X. doi: 10.1109/4235.585893.

[7] 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.

[8] 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.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton. ImageNet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, 2017. ISSN 15577317. doi: 10.1145/3065386.

URL http://code.google.com/p/cuda-convnet/https://papers.nips.cc/paper/2012/hash/c399862d3b9d6b76c8436e924a68c45b-Abstract.html.

[10] 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.

[11] 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.

[12] S. Agostinelli, J. Allison, K. Amako, et al. GEANT4 - A simulation toolkit. *Nuclear Instruments and Methods in Physics Research, Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 506(3):250–303, jul 2003. ISSN 01689002. doi: 10.1016/S0168-9002(03)01368-8.

[13] S. Liddick. Digital Data Acquisition System (DDAS). URL https://frib.msu.edu/science/instruments/operation.html{#}ddas.

[14] M. Abadi, P. Barham, J. Chen, et al. TensorFlow: A system for large-scale machine learning. In *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016*, pages 265–283, 2016. ISBN 9781931971331. URL https://research.google/pubs/pub45381/www.tensorflow.org.

[15] F. and others Chollet. Keras, 2015. URL https://keras.io.

[16] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, pages 1–14, 2015.

[17] M. Stone. Cross-Validatory Choice and Assessment of Statistical Predictions. *Journal of the Royal Statistical Society: Series B (Methodological)*, 36(2):111–133, 1974. doi: 10.1111/j.2517-6161.1974.tb00994.x. URL https://www.jstor.org/stable/2984809?seq=1{#}metadata{_}info{_}tab{_}contents.