1 APPENDICES

Various methods defined under the broader scope of machine learning were used to build classifiers to distinguish electron tracklet signals from pion tracklet signals produced during high energy physics (HEP) experiments. These tracklet signals, which were fed as input features to the abovementioned machine learning algorithms, manifested in this project as

As mentioned above in **Error! Reference source not found.**, quarks and gluons are confined by the Strong Force to remain within the bound states of colour-neutral hadrons (e.g. protons and neutrons) and are therefore never found freely in nature. However, the

Below are six examples of simulated tracklet image data, produced by the VAE as explained above.

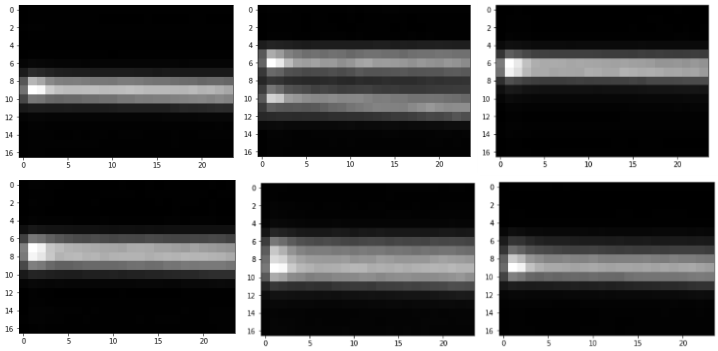


Figure 1: Six examples of simulated data created using a Variational Autoencoder

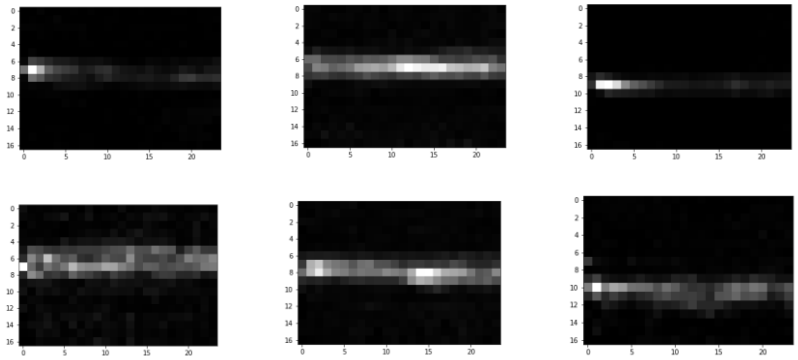


Figure 2: Six examples of simulated data created using a Generative Adversarial Network

Example images:

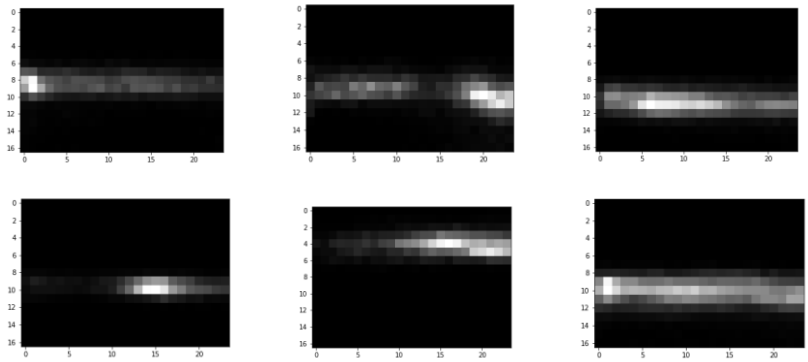


Figure 3: Six examples of simulated data created using an Adversarial Autoencoder

OLD ABSTRACT:

This Masters Dissertation outlines the application of deep learning methods on raw data from the Transition Radiation Detector at CERN as well as on simulated data from the Monte Carlo Event Generator Geant4, in order to achieve the following goals:

i. Particle identification; distinguishing between electrons and pions

To this end, various feedforward neural networks, convolutional neural networks, as well as recurrent neural networks were built using Keras with a TensorFlow back-end, resulting in an ultimate pion efficiency of $\varepsilon_\pi = 1.2\%$ in the $P \leq 2\ GeV$ range, $\varepsilon_\pi = 1.14\%$ in the $2\ GeV < P \leq 3\ GeV$ and $\varepsilon_\pi = 1.51\%$ in the $3\ GeV < P \leq 4\ GeV$ range, all at electron efficiency of $\varepsilon_e \approx 90\%$. The best results were obtained using an incrementally trained convolutional neural network, which was trained on data from particles in increasing momentum ranges sequentially.
Raw data was extracted from the Worldwide LHC Computing grid using the ROOT data analysis framework, a C++ based platform maintained by physicists at CERN. R and Python were used interchangeably during various stages of data exploration, processing, analysis and model-building.

ii. High Energy Physics Event Simulations, Part I: Distinguishing real data from data generated by Geant4

This stage of the project focused on employing convolutional neural networks towards distinguishing real data from simulated data. Data was simulated using Geant4, a Monte Carlo toolkit which simulates the passage of particles through matter. ROOT was used to reconstruct the simulated data to deliver it in a similar format to that given by raw data after processing. A balanced accuracy score of 91.5% (with Sensitivity = 0.8575 and Specificity = 0.9725) was achieved, using a 2D Convolutional Neural Network.
The fact that Geant4 simulations were easily discriminated from real data motivated the third stage of this thesis.

iii. High Energy Physics Event Simulations, Part II: Deep Generative Modeling

Various deep generative models were built to take as input raw TRD data and produce simulated observations which are likely under the training data distribution. Various strategies were employed towards deep generative modelling; with Adversarial Autoencoders giving the best results.

OLD INTRODUCTION:

This Masters Dissertation seeks to apply cutting edge techniques in Machine Learning (ML) towards:

- Particle identification (i.e. classification) of electrons and pions, from raw signal data produced by these particles as they traverse the Transition Radiation Detector (TRD), using various deep learning methods
- Optimizing High Energy Physics Event Simulations by:

  - Distinguishing between real data and data simulated by the Geant4 Monte Carlo simulation environment
  - Building deep generative models, namely Variational Autoencoders, Adversarial Autoencoders and Generative Adversarial Networks for the simulation of TRD data obtained during High Energy Physics (HEP) collision events and quantifying each model's accuracy

The motivation for each of these elements is as follows:

- Accurate particle identification (in particular, electron samples that are as pure as possible) allows physicists at the ALICE (A Large Ion Collider Experiment) experiment to study the properties of the Quark Gluon Plasma (QGP), a primordial state of matter thought to have existed in the early universe. Since this deconfined state of matter rehadronizes quite soon after forming, it cannot be studied directly, but only via its decay products. To this end, accurate particle identification is extremely important
- Being able to distinguish Monte Carlo simulations from real data, could be indicative that Monte Carlo simulations, used for calibration and calculations of detector response functions, etc. are not accurate enough and that they could potentially be tuned via various parameter settings in future studies to increase their accuracy
- Using deep generative models such as Variational Autoencoders (VAEs), Adversarial Autoencoders (AAEs) and Generative Adversarial Networks (GANs), instead of Monte Carlo simulations, could be a desirable future course of action, since these simulations are extremely fast compared to Geant4 simulations; but their practical use is contingent on whether they provide comparable accuracy to Geant4 simulations, as well as their customizability, e.g. is it possible to specify which particle-type, and at which momentum you want to be simulated?

A variety of different software packages were utilized during the course of this project, including ROOT for data extraction, Geant4 for event simulation, Python and R for statistical analysis and Keras with a Tensorflow back-end for deep learning implementations.

The highest balanced accuracy in distinguishing Geant4 simulated data from true raw data was 91.5%. This was also achieved using a convolutional network, discussed in

Quantum mechanics explains the emergence of unique physical properties in different elements, which arise from their exact electronic structures. Quantum field theories explain

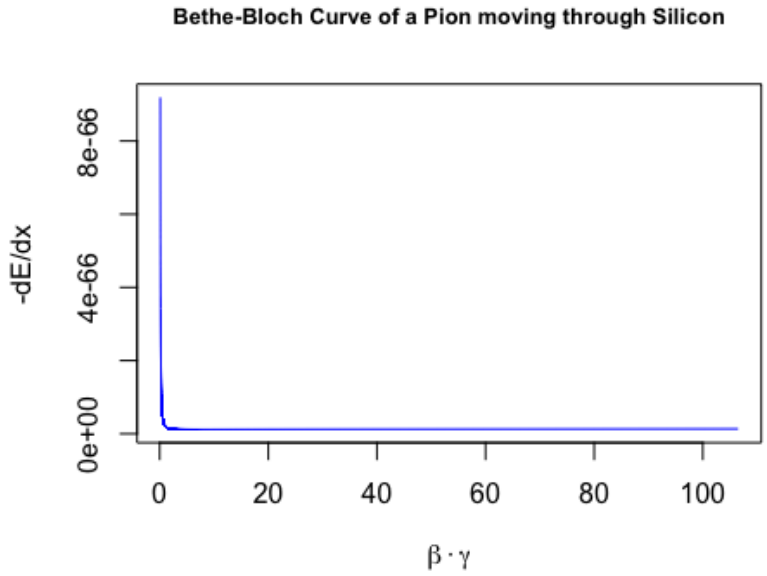
Chapter 1: Appendices

APPENDIX A: PLOTTING THE BETHE-BLOCH EQUATION

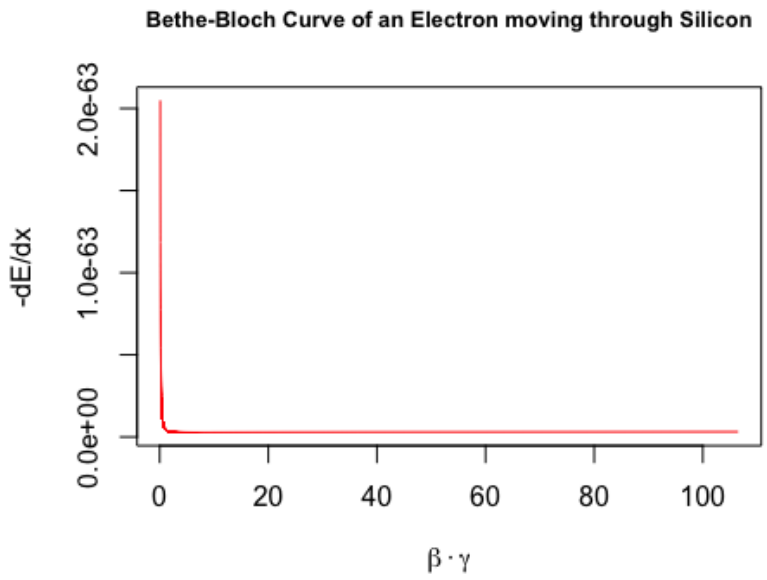
Create a Bethe-Bloch function:

```
#Planck's constant:
h <- 6.62607004e-34

#Speed of light m/s
c <- 299792458

#Fine structure constant
alpha <- 1/137

#Mass of an electron Mass/GeV

m.e <- 0.005

#Density n, atomic number Z, the fraction of the speed of light the pa
rticle is moving at, beta, and the particle's velocity v are specified
as parameters to the equation


dE.dx <- function(n,Z,v,beta){
  -4 * pi * h^2 * c^2 * alpha^2 * ((n * Z)/(m.e * v^2)) * log(((2 * be
ta^2 * gamma^2 * c^2 * m.e)/(I.e)) - beta^2,base=exp(1))
}

#For an electron traversing a silicon detector:

v <- seq(0.1*c,c,100000)

beta <- v/c

#Lorentz factor

gamma <- 1/(sqrt(1-(v^2/c^2)))

n <- 1



Z <- 14

#Effective ionization potential of the material

I.e <- 10 * Z

electron.y = dE.dx(n=n,Z=Z,v=v,beta=beta)

require(latex2exp)

## Loading required package: latex2exp
```

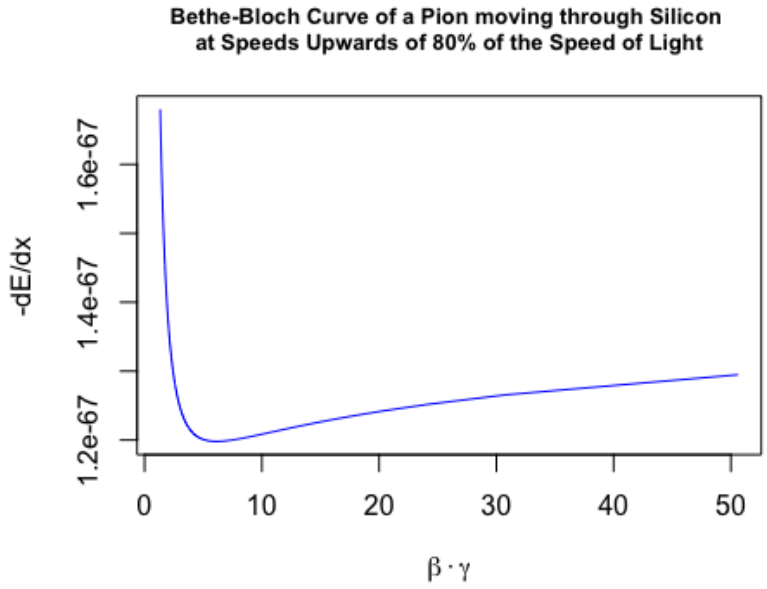
```
m.e <- 273.13*m.e

pion.y = dE.dx(n=n,Z=Z,v=v,beta=beta)



plot(x=beta*gamma, y=-pion.y,type="l",main="Bethe-Bloch Curve of a Pio
n moving through Silicon", xlab = TeX("$\\beta\\cdot\\gamma$"),ylab=Te
X("$-dE/dx$"),col="blue",cex.main=0.8)
```



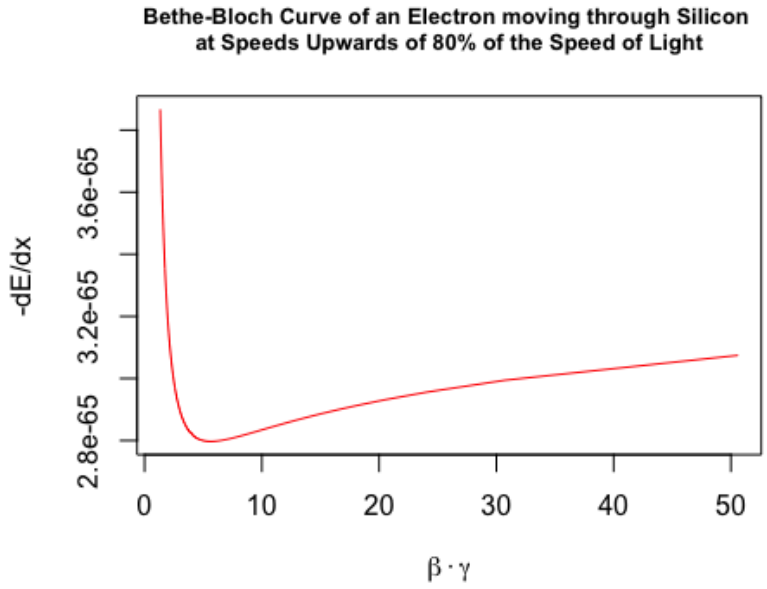

```
plot(x=beta*gamma, y=-electron.y,type="l",main="Bethe-Bloch Curve of a
n Electron moving through Silicon", xlab = TeX("$\\beta\\cdot\\gamma$"
),ylab=TeX("$-dE/dx$"),col="red",cex.main=0.8)
```



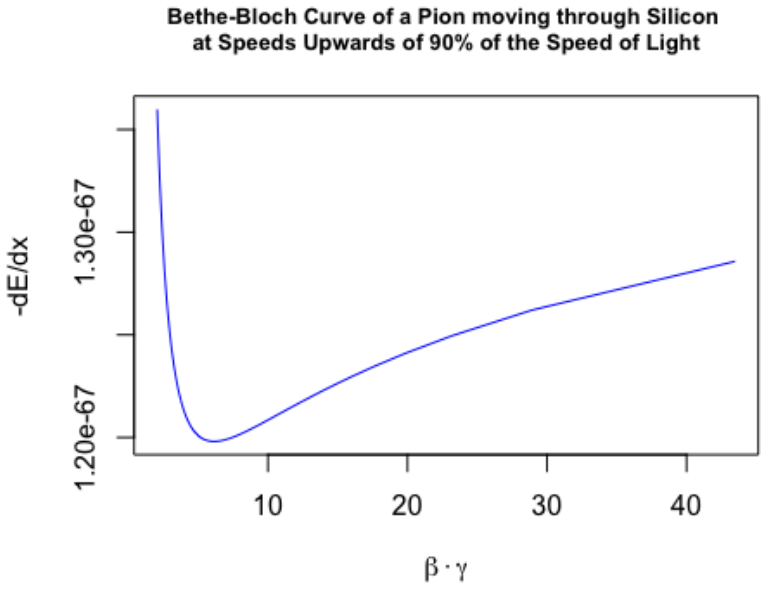

```
v <- seq(0.8*c,c,100000)

beta <- v/c

#Lorentz factor

gamma <- 1/(sqrt(1-(v^2/c^2)))

n <- 1

m.e <- 0.005

electron.y = dE.dx(n=n,Z=Z,v=v,beta=beta)

m.e <- 273.13*m.e

pion.y = dE.dx(n=n,Z=Z,v=v,beta=beta)

plot(x=beta*gamma, y=-pion.y,type="l",main="Bethe-Bloch Curve of a Pio
n moving through Silicon \nat Speeds Upwards of 80% of the Speed of Li
ght", xlab = TeX("$\\beta\\cdot\\gamma$"),ylab=TeX("$-dE/dx$"),col="bl
ue",cex.main=0.8)
```



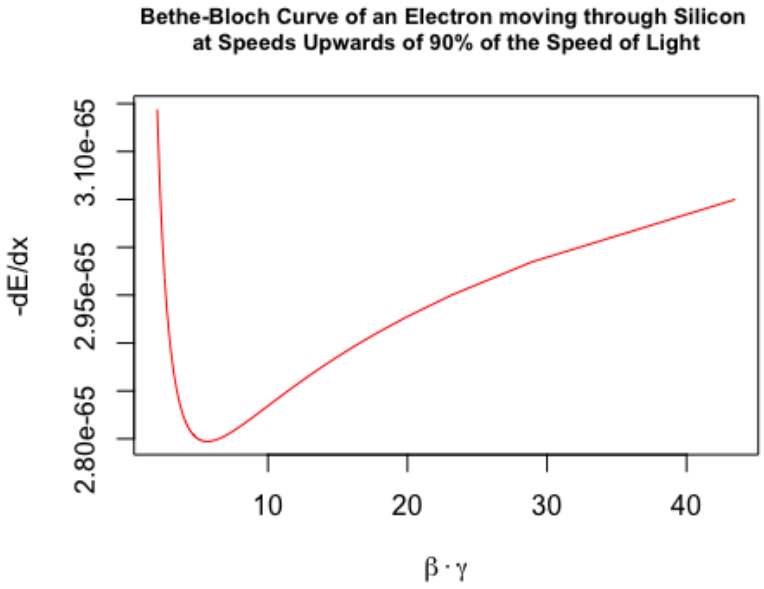

```
plot(x=beta*gamma, y=-electron.y,type="l",main="Bethe-Bloch Curve of a
n Electron moving through Silicon \nat Speeds Upwards of 80% of the Sp
eed of Light", xlab = TeX("$\\beta\\cdot\\gamma$"),ylab=TeX("$-dE/dx$"
),col="red",cex.main=0.8)
```




```
v <- seq(0.9*c,c,100000)

beta <- v/c

#Lorentz factor

gamma <- 1/(sqrt(1-(v^2/c^2)))

n <- 1

m.e <- 0.005

electron.y = dE.dx(n=n,Z=Z,v=v,beta=beta)

m.e <- 273.13*m.e

pion.y = dE.dx(n=n,Z=Z,v=v,beta=beta)

plot(x=beta*gamma, y=-pion.y,type="l",main="Bethe-Bloch Curve of a Pio
n moving through Silicon \nat Speeds Upwards of 90% of the Speed of Li
ght", xlab = TeX("$\\beta\\cdot\\gamma$"),ylab=TeX("$-dE/dx$"),col="bl
ue",cex.main=0.8)
```



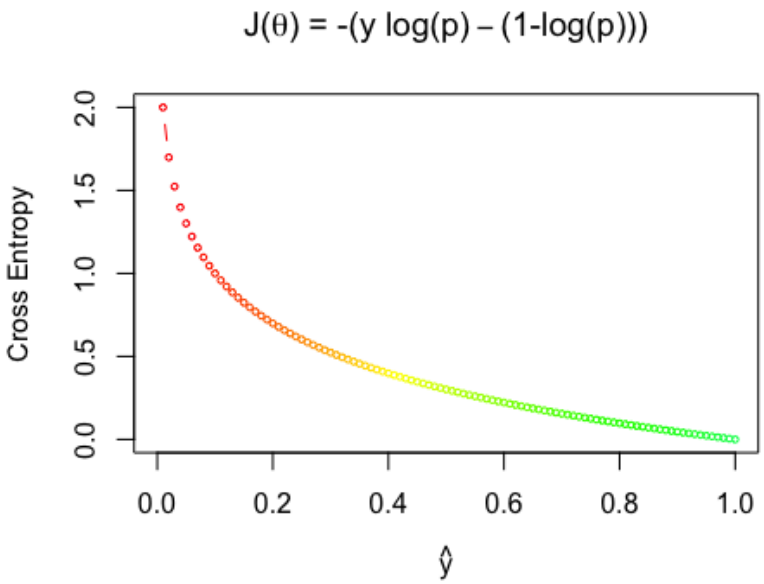

```
plot(x=beta*gamma, y=-electron.y,type="l",main="Bethe-Bloch Curve of a
n Electron moving through Silicon \nat Speeds Upwards of 90% of the Sp
eed of Light", xlab = TeX("$\\beta\\cdot\\gamma$"),ylab=TeX("$-dE/dx$"
),col="red",cex.main=0.8)
```


Bethe-Bloch Curve of an Electron moving through Silicon
at Speeds Upwards of 90% of the Speed of Light
-dE/dx
2.80e-65
2.95e-65
3.10e-65
10
20
30
40
β · γ

APPENDIX B: PLOTTING BINARY CROSS-ENTROPY

Define a function to plot the binary cross-entropy loss function:

```
cross.entropy <- function(y,p){
  -(y * log(p,base = 10) + ((1-y)*(1 - log(p,base=10))))
}

#if the predicted class is 1:

y <- 1

p <- seq(0,1,0.01)

loss <- cross.entropy(y,p)

require(latex2exp)

## Loading required package: latex2exp

plot(x=p,y=loss, type="b", col=rainbow(250),cex=0.5, main = TeX("J($\\
theta$) = -(y log(p)-(1-log(p)))"),ylab = "Cross Entropy", xlab = TeX(
"$\\hat{y}$"))
```



APPENDIX C: THE ANATOMY OF AN ALIROOT ANALYSIS TASK

In AliROOT, all analysis tasks inherit from the base class **`AliAnalysisTaskSE`** (where SE stands for Single Event), which in turn is derived from the base class **`AliAnalysisTask`.**

All analysis tasks done in AliROOT inherit the following base methods from **`AliAnalysisTaskSE`**:

```
        AliAnalysisTaskSE::AliAnalysisTaskSE();//constructor1

        AliAnalysisTaskSE::AliAnalysisTaskSE(const
char*);//constructor2

        AliAnalysisTaskSE::~AliAnalysisTaskSE();//destructor

        AliAnalysisTaskSE::UserCreateOutputObjects();//user-
defined output objects (results of physics analyses, which can
be attached to output files)

        AliAnalysisTaskSE::UserExec(Option_t*);//event loop,
called for each event in the analysis: checks conditions for
inclusion, accesses physics objects, fills histograms or other
data containers with attributes from event

        AliAnalysisTaskSE::Terminate(Option_t*); //deallocates
memory after all steps in analysis have completed
```

The final element of an analysis task in AliROOT is the (.C) macro file, which creates and configures an instance of the particular C++ class.

The Class Header (.h)

Reproduced and modified from (35):

```
#ifndef AliAnalysisTaskMyTask_H //include guard (aids in prevention of double
inclusion, which may result from including parent and child classes, leading
to multiple definitions for class members)
#define AliAnalysisTaskMyTask_H //part of include guard

class AliAnalysisTaskMyTask : public AliAnalysisTaskSE //we define a class
AliAnalysisTaskMyTask, which inherits from the base class AliAnalysisTaskSE
{
public:
   // two class constructors, called when a new instance of the class is
created
```

```
   AliAnalysisTaskMyTask();
   AliAnalysisTaskMyTask(const char *name);
   // class destructor, called when this instance of the class is deleted
   virtual                ~AliAnalysisTaskMyTask();
   // called once at beginning of runtime
   virtual void           UserCreateOutputObjects();
   // called for each event
   virtual void           UserExec(Option_t\* option);
   // called at end of analysis
   virtual void           Terminate(Option_t\* option);

//class members

     private:
       AliAODEvent*  fAOD;           //!<! pointer to a single input event
       TList*        fOutputList;    //!<! pointer to an output list, which
holds all the output objects of the analysis
       TH1F*         fHistPt;        //!<! pointer to a histogram containing
the transverse momentum (Pt) spectrum

//note that the !<! expression above is seen and evaluated by ROOT and is
used in the generation of ROOT documentation

//ClassDef definition:

  /// \cond CLASSDEF //surrounding comments for documentation generation
  ClassDef(AliAnalysisTaskMyTask, 1); //this is a C pre-processor macro, used
when class derives from TObject: it contains member declarations and inserts
a few new members into the class, version number is incremented from 1 when
definition of class changes
  /// \endcond}; //surrounding comments for documentation generation

#endif //part of include guard
```

The Class Implementation (.cxx)

Reproduced and modified from (35):

```
//include statements for UserCreateOutputObjects:

    #include "TList.h" //TList class, an instance of which will
contain a histogram in this example
    #include "TH1F.h" //ROOT 1-dimensional histogram class with one float per
channel

//include statement for UserExec:

  #include "AliAODEvent.h"

//implementation of class constructors:
AliAnalysisTaskMyTask::AliAnalysisTaskMyTask() : AliAnalysisTaskSE(),
//members of the class are initialized in the constructors with their default
values, if default values are not specified, these will be filled with random
values, which could lead to unexpected behaviour
           fAOD{0}, fOutputList{0}, fHistPt{0}
       {
           // This first constructor is the ROOT IO constructor, memory
should not be allocated here
       }
```

```
//in the second constructor, below, the input and output objects handled by
the class are defined

        AliAnalysisTaskMyTask::AliAnalysisTaskMyTask(const char* name) :
AliAnalysisTaskSE(name),
            fAOD{0}, fOutputList{0}, fHistPt{0}
        {
//input object is a TChain
            DefineInput(0, TChain::Class());
//output object is a TList
            DefineOutput(1, TList::Class());
        }

//implementation of the UserCreateOutputObjects class:

AliAnalysisTaskMyTask::UserCreateOutputObjects()
    {
        // create a new TList that OWNS its objects
        fOutputList = new TList();
        fOutputList->SetOwner(true);

        // create a histogram:
//from ROOT's online documentation, this is the constructor for a TH1F:
//TH1F (const char *name, const char *title, Int_t nbinsx, Double_t xlow,
Double_t xup)
//seen below, we give the histogram the pointer name defined in the header
file and give the histogram plot the same title, we define the histogram
itself to have 100 bins on an x-axis bounded by [0,100]
        fHistPt = new TH1F("fHistPt", "fHistPt", 100, 0, 100);
//add the histogram to the output list:
        fOutputList->Add(fHistPt);

        // add the list to our output file
        PostData(1,fOutputList); //calling PostData() notifies client tasks
of the fOutPutList data container that its contents have changed
    }

//UserExec: the "event loop" (operations defined here are called for each
event in the analysis):

AliAnalysisTaskMyTask::UserExec(Option_t*)
    {
      // get an input event from the analysis manager and cast it as an
AliAODEvent
      fAOD = dynamic_cast<AliAODEvent*>(InputEvent());

      // check if there actually is an event, and throw a fatal exception
with error message if not
      if(!fAOD)
        ::Fatal("AliAnalysisTaskMyTask::UserExec", "No AOD event found, check
the event handler.");

// Loop over all the tracks in the event and fill the histogram

        // get the number of tracks in the input event
        int iTracks{fAOD->GetNumberOfTracks()};

        // iterate through all the tracks in the event:
        for(int i{0}; i < iTracks; i++) {
//get the current track, cast it as an AliAODTrack
```

```
            AliAODTrack* track = static_cast<AliAODTrack*>(fAOD-
>GetTrack(i));
//if the track variable does not exist after the above operation, continue to
the next iteration of the loop
            if(!track) continue;

            // here we do some track selection
            if(!track->TestFilterbit(128) continue;

            // get the transverse momentum of the track and fill the
histogram with this data
            fHistPt->Fill(track->Pt());
        }
        // save the output list
        PostData(1, fOutputList);
    }
```

The AddTask macro (.C)

Reproduced and modified from (35):

```
//this file instantiates our class, defines its input and
output, and connects it to the analysis manager

AliAnalysisTaskMyTask* AddMyTask(TString name = "name") {
//get a pointer to the analysis manager
  AliAnalysisManager *mgr =
AliAnalysisManager::GetAnalysisManager();

  // resolve the name of the output file
  TString fileName = AliAnalysisManager::GetCommonFileName();
  fileName += ":MyTask";      // create a subfolder in this file

  // create an instance of the analysis task
  AliAnalysisTaskMyTask* task = new
AliAnalysisTaskMyTask(name.Data());

  // add this task to the analysis manager
  mgr->AddTask(task);

  // connect the manager to the task's input container
  mgr->ConnectInput(task,0,mgr->GetCommonInputContainer());
  // connect the manager to the task's output container (TList)
  mgr->ConnectOutput(task,1,mgr-
>CreateContainer("MyOutputContainer", TList::Class(),
AliAnalysisManager::kOutputContainer, fileName.Data()));

  // important: return a pointer to this task
  return task;
}
```

APPENDIX D: SOFTWARE ENVIRONMENT, PACKAGES & UTILITIES

1.1 Software Environment

1.1.1 AiROOT

AliROOT was built locally using alidock Docker container.

AliROOT was built from source on the hep01 server hosted at UCT

1.1.2 R Statistical Software

Packages

1.1.3 ROOTR

1.1.4 Keras & Tensorflow

1.1.5 Utilities

Makefiles

```
all: gridfiles.md5

gridfiles.xml: query.sh
	./$< > $@

gridfiles.md5: gridfiles.xml
	xsltproc /alice/data/util/xml2md5.xsl $< > $@

download: $(shell cut -c 49- files.md5)

/alice/data/%:
	mkdir -p $(dir $@)
	alien_cp alien:$@ file:$@
```

User specified aliases in ~/.bashrc

```
# User specific aliases and functions
alias initialize_aliroot='/cvmfs/alice.cern.ch/bin/alienv enter
VO_ALICE@AliPhysics::vAN-20180902-1'

alias my_alice='alienv -w /alice/gviljoen/alice/sw enter
VO_ALICE@AliPhysics::latest'
```

Remote Editing

Rsync

```
rsync -av --stats --progress --include="*/" --include="*.txt" --
exclude="*.C" --exclude="*.cxx" --exclude="*.h" --exclude="*.root" --
exclude="*.ps" --exclude="*.d" --exclude="*.so" --exclude "*.proc"
gviljoen@hep01.phy.uct.ac.za:/alice/gviljoen/trdpid/adj_sim/test .
```

X11 Forwarding

Atom packages:

- Remote Atom Server
- PlatformIO-IDE-Terminal

Killing a process being listened to on the remote port 52698:

List processes that are owned by me:

```
ps aux | grep gviljoen
```

Find the sshd process being listened to on port 52698 and kill it, by running:

```
kill -9 $processid
```

In this case, here is the suspect process (`$processid = 28525`):

```
gviljoen 28525  0.0  0.0 119612  2168 ?        S    13:02   0:00
sshd: gviljoen@pts/0
```

1.1.6 Python

APPENDIX E: RUNNING AND MONITORING ROOT ANALYSIS TASKS

Once one is happy with the analysis task defined, one first needs to enter AliPhysics, by using one of the user-defined aliases, e.g.:

```
initialize_aliroot
```

Then, one gets a token from alien, to access the grid. This token will be valid for 24 hours. Since my CERN username is not the same as my username on HEP01, the command is:

```
alien-token-init username
```

Once the above commands have been run, one can run the analysis task on the grid, by setting the following parameters in the analysis macro (ana.C):

```
  Bool_t local = kFALSE;
  Bool_t gridTest = kFALSE;
```

Adding the appropriate run number and output directory:

```
alienHandler->AddRunNumber(265377);

alienHandler->SetGridWorkingDir("new-wd-momentum-test");
alienHandler->SetGridOutputDir("outDir265378");
```

Setting the run mode and starting the analysis:

```
alienHandler->SetRunMode("full");
//alienHandler->SetRunMode("terminate"); //this is run for
merging stages
mgr->StartAnalysis("grid");
```

Assuming that one has added the appropriate CERN certificates, one can then view, manage and download the output of one's jobs on the MonALISA grid monitoring site for ALICE see Figure 4 for an example screenshot of user job monitoring and Figure 5 for the user interface for viewing the directory structure for the ALICE grid, in particular the user's working directory:

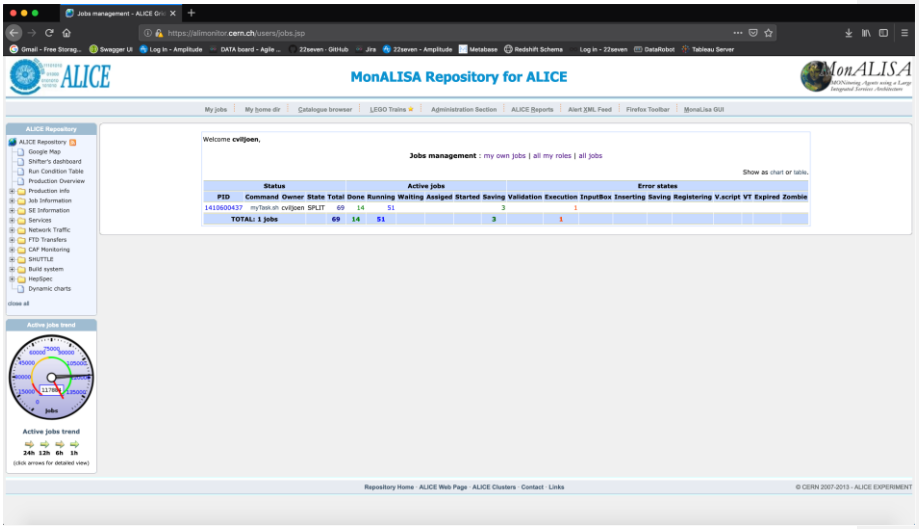


Figure 4: MonALISA Alice grid monitoring site, user jobs at url: https://alimonitor.cern.ch/users/jobs.jsp

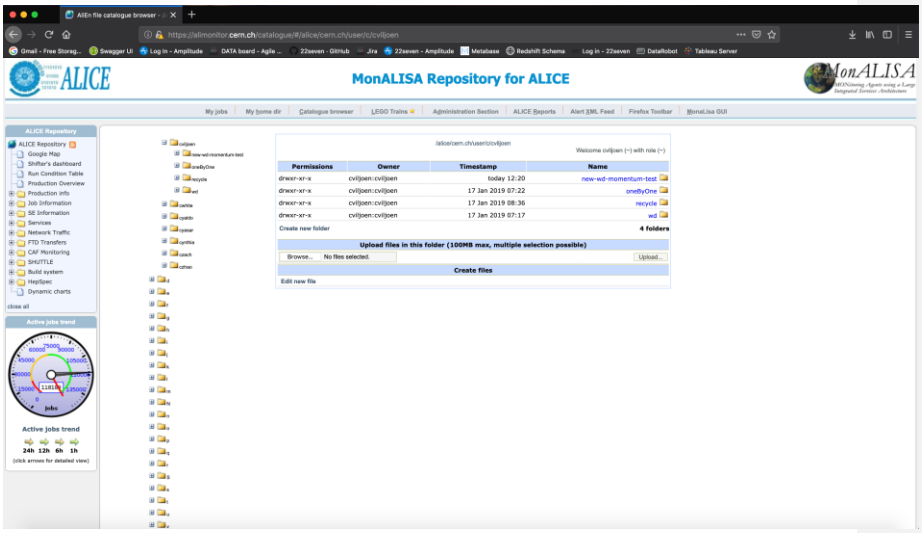


Figure 5: User working directory structure on MonALISA at url: https://alimonitor.cern.ch/catalogue/#/alice/cern.ch/user/c/cviljoen

In Figure 6, a screenshot shows how subjobs belonging to a masterjob can be tracked by clicking on the process ID on the MonALISA jobs management webpage:

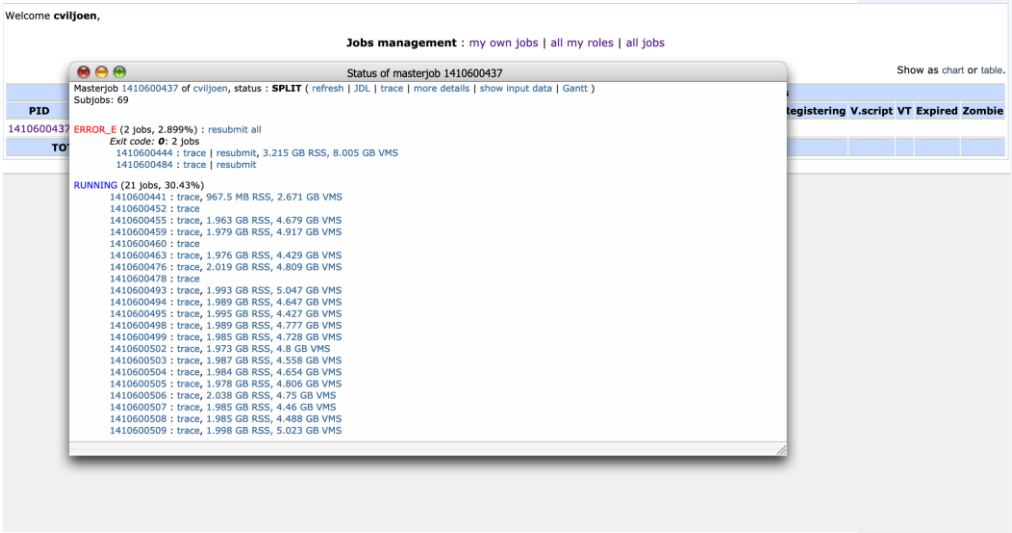


Figure 6: Tracking the status of subjobs of a master-job, by clicking on the process id (PID)

One can resubmit errored subjobs by browsing through the various error states in the "Status of masterjob" view and clicking on "resubmit all" for all processes that are in a specific error state.

The trace of a subjob (see Figure 7 for an example screenshot) can give hints as to what caused a specific subjob to fall into an error state. In this case the job has an error state "ERROR_E", i.e. "Error in Execution", since the job is using too much memory (memory and storage limits are allocated to each user and overusing either can downgrade the priority of a user's jobs).

The alien shell can be accessed by running

```
aliensh
```

This gives access to the alien terminal, which is not strictly a bash terminal, but has similar commands, for instance the shell command to forcefully and recursively remove a directory:

```
rm -rf directory
```

would be achieved on an alien terminal by running:

```
rmdir directory
```

Killing a job is done in a similar fashion to the normal shell workflow, i.e. running

```
ps
```

To list the currently active processes and

```
kill $(process-id)
```

To kill a process and its attendant subprocesses, in case you figured out that you made a mistake and want to terminate a running process early for whatever reason.

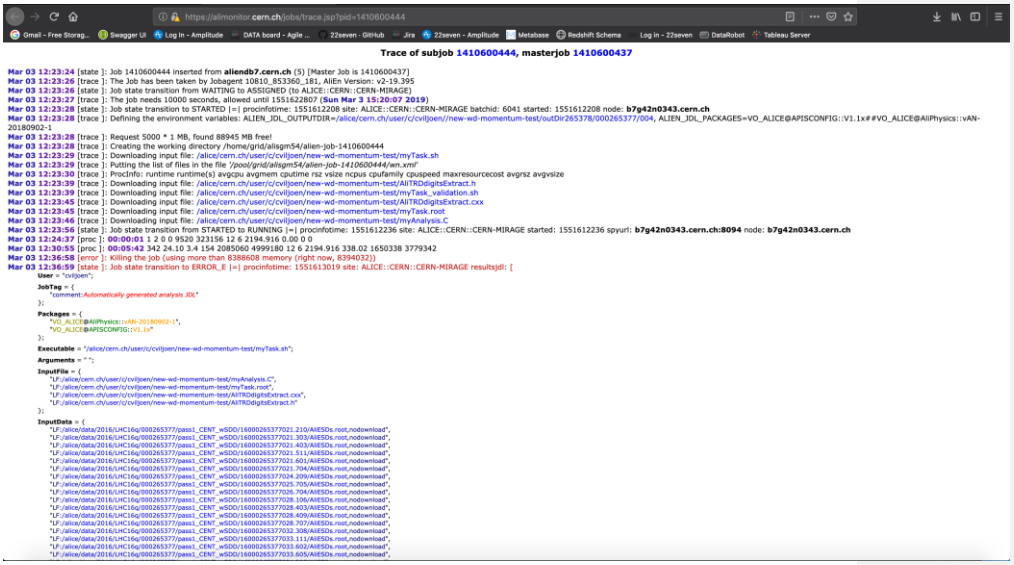


Figure 7: Example trace of a subjob on MonALISA

APPENDIX F: DATA EXTRACTION, DATA QUALITY AND DATA PRE-PROCESSING

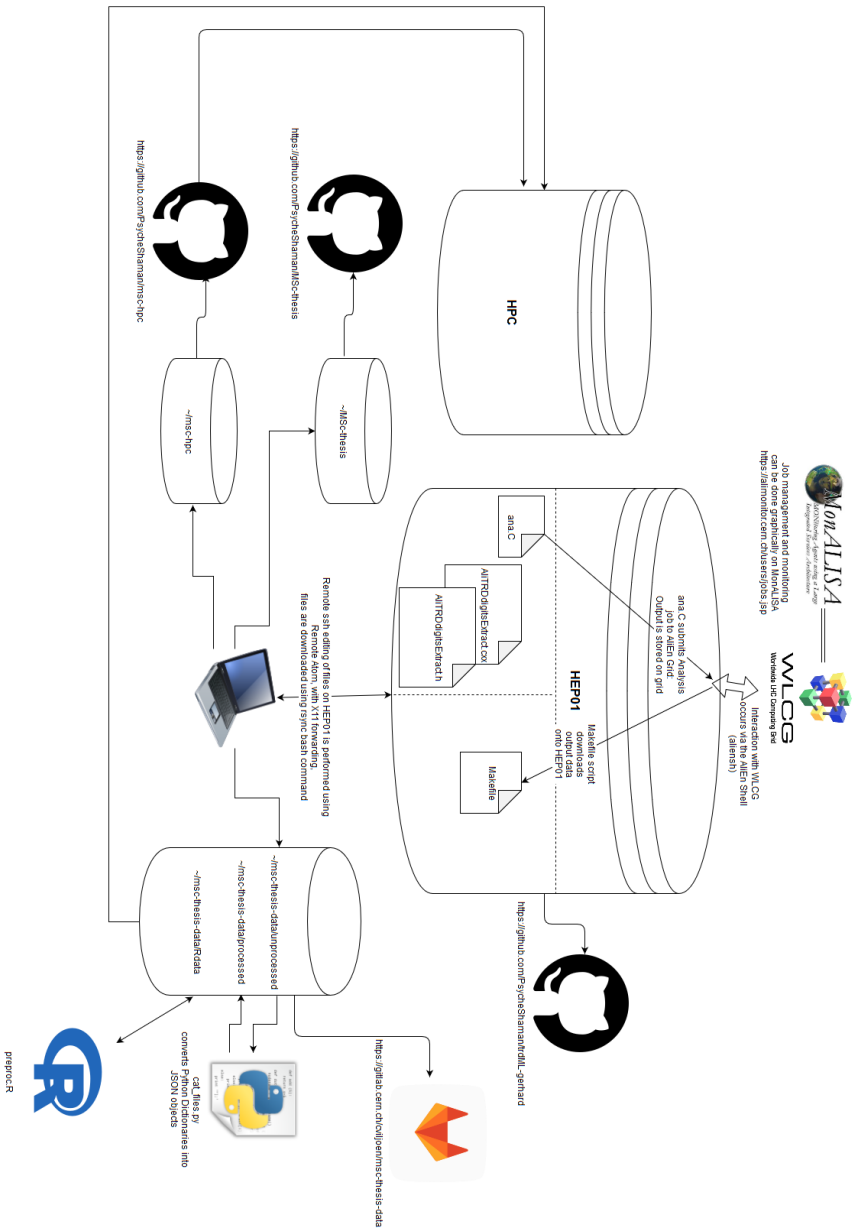


Figure 8: Data pre-processing environment and repository logic

1.1.7 Running Digits Extract Task on AliEn Grid

- Stage 1
- Stage 2
- Stage 3

1.1.8 Data Extraction from WLCG

From Alien to HEP01

Makefile

From HEP01 to Local Machine

Into data backup directory: https://gitlab.cern.ch/cviljoen/msc-thesis-data

Directory structure

```
scp -r
gviljoen@hep01.phy.uct.ac.za:/alice/cern.ch/user/c/cviljoen/wd/o
utDir265377/000265377/ .
```

```
rsync -av --stats --progress
gviljoen@hep01.phy.uct.ac.za://alice/cern.ch/user/c/cviljoen/wd/
od/ .
```

From Local Machine to HPC

1.1.9 Data Quality Assessment and Descriptive Statistics

1.1.10 File Merging and Conversion of Python Dictionaries to JSON Objects

Cat_files.py

1.1.11 Loading JSON Files into R Environment and Data Wrangling for Deep Learning

Wrangle.R

1.1.12 UCT HPC Cluster

Compiler variables set in ~/.R/Makevars

```
CC = gcc -std=gnu99
```

Install.packages command needs to be modified to write packages in a directory where there are permissions and where the CRAN mirror is set, dependencies=TRUE allows R to read the Makevars compiler variables.

install.packages(pkgs="keras",lib="/scratch/username",
repos="https://cloud.r-project.org",dependencies=TRUE)

APPENDIX G: OLD METHODS SECTION

Model 1

An initial benchmark feedforward model was built, compiled and trained, according to the following lines of Python code:

```
model1 = Sequential([
    Dense(256, input_shape=(24,)),
    Activation('relu'),
    Dense(128),
    Activation('relu'),
    Dense(128),
    Activation('relu'),
    Dense(64),
    Activation('relu'),
    Dense(2),
    Activation('softmax')
])

model1.compile(loss='categorical_crossentropy',
              optimizer='rmsprop',
              metrics=['accuracy'])


history = model1.fit(x_train, y_train,
              epochs=epochs,
              validation_split=0.15,
              shuffle=True,
              verbose=2)
```

The input features to this model were as follows:

- Time-bin sums across all pads, divided by the mean of the entire x sample
- Missing data removed
- Electrons oversampled (the electron sample in the training data was taken thrice)

This model was trained on 5778261 samples and validated on 642029 samples.

As can be seen in 0, this model failed to train, so an approach was taken to account for class imbalances using a different method, and by starting with a very simple architecture and sequentially adding complexity to the model.

Sequential Model Building

Stage 1

All electron tracks were included and a pion sample twice the size of the electron sample was added, before partitioning data into a training and test set.

Class imbalances were accounted for by allowing error in electron classification to contribute proportionately more to the loss function.

As per the approach followed by the current classification neural network in production at the TRD, timebins were compressed by summing across three timebins at a time, to create 8 new features, which were added to the 24 timebins already available to the neural network as input features.

Class weights were accounted for as follows:

```
class_weights = class_weight.compute_class_weight('balanced',
                                                  np.unique(y_train),
                                                  y_train)

class_weights = {0:class_weights[0],1:class_weights[1]}
```

The model was built, compiled and trained according to the following Python code:

```
sgd = optimizers.SGD(lr=0.01, clipvalue=0.5)

model1_dropout_0_5 = Sequential([
    Dense(32, input_shape=(32,)),
    Activation('relu'),
    Dense(2),
    Activation('softmax')
])

batch_size=32

model1_dropout_0_5.compile(loss='binary_crossentropy',
              optimizer=sgd,
              metrics=['accuracy'])


history = model1_dropout_0_5.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_split=0.1,
              shuffle=True,
              verbose=2,
              class_weight=class_weights)
```

Model was trained on 883591 samples, and validated on 98177 samples, due to undersampling of pions.

As can be seen in 0, this model did train, in contrast to the first model, although it did not achieve validation accuracy above 75%.

Stage 2

Following the successful running of the above model, the same model was run, without compensating for class imbalances by undersampling pions, but by maintaining the proportionally greater contribution to the loss function by the underrepresented “electron” class.

This model was trained on 2720444 samples and validated on 302272 samples.

Stage 3

After successful training of the single layer, 32 node neural network above, a larger network was constructed as follows, using the same dataset, optimizer, etc.

```
model1_dropout_0_5 = Sequential([
    Dense(128, input_shape=(32,)),
    Activation('relu'),
    Dense(128),
    Activation('relu'),
    Dense(2),
    Activation('softmax')
])
```

As can be seen in 0, increasing the model capacity in this way does have its benefits in terms of accuracy, without seeming to overfit too much, therefore the next model was built with much higher capacity.

Stage 4

```
model1_dropout_0_5 = Sequential([
    Dense(128, input_shape=(32,)),
    Activation('relu'),
    Dense(128),
    Activation('relu'),
    Dense(128),
    Activation('relu'),
    Dense(128),
    Activation('relu'),
    Dense(128),
    Activation('relu'),
    Dense(2),
    Activation('softmax')
])
```

While this model was slightly more accurate than those discussed before, it is clear when looking at 0 that the model was not generalizable to the validation set, therefore, before increasing model complexity, regularization in the form of dropout was introduced as follows:

1.1.13 2D Convolutional Neural Networks

Stage 1

Trained on 770981 samples, validated on 85655 samples:

```
epochs = 100

model = Sequential()
model.add(Conv2D(16, (2, 2),
padding='valid',input_shape=(17,24,1),data_format="channels_last"))
model.add(Activation('relu'))
```

```
model.add(MaxPooling2D(pool_size=(3, 3)))
model.add(Flatten())
model.add(Dense(256))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(2))
model.add(Activation('softmax'))


sgd = tensorflow.keras.optimizers.SGD(lr=0.01, clipvalue=0.5)

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

history=model.fit(x_train, y_train,
              epochs=epochs,
              validation_split=0.1,
              shuffle=True)
```

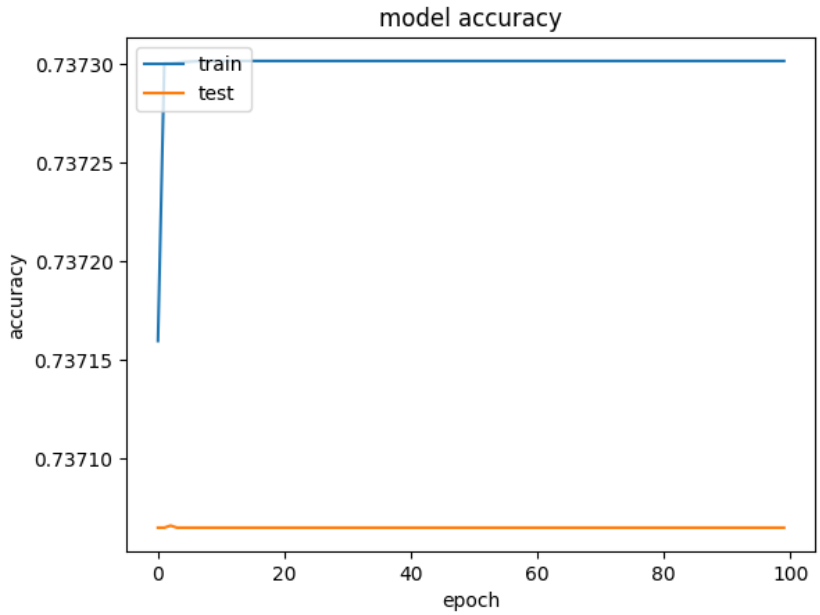
Stage 2

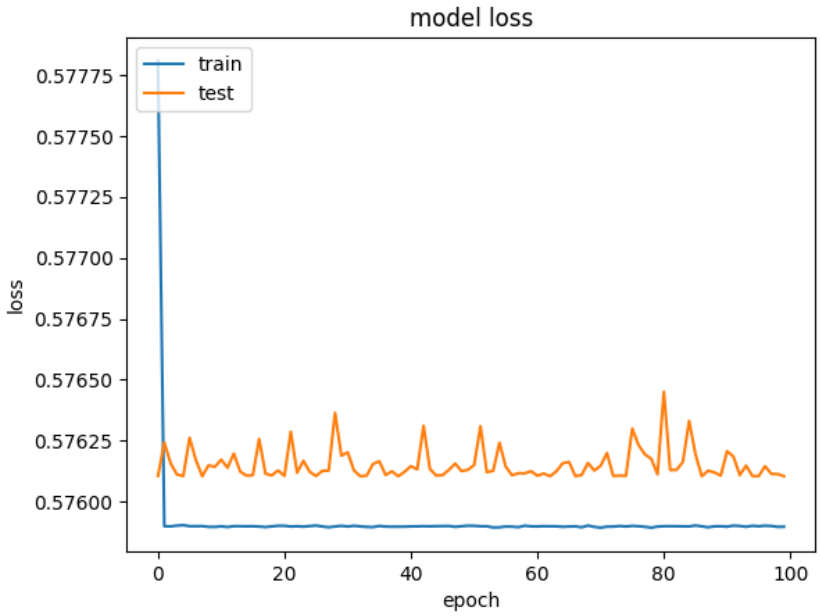
```
epochs = 100

model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
input_shape=(x_train.shape[1],x_train.shape[2],x_train.shape[3]),data_
format="channels_last"))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(2, activation='softmax'))

sgd = tensorflow.keras.optimizers.SGD(lr=0.01, clipvalue=0.5)

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

batch_size=32

history=model.fit(x_train, y_train,
              epochs=epochs,
              validation_split=0.1,
              shuffle=True)
```

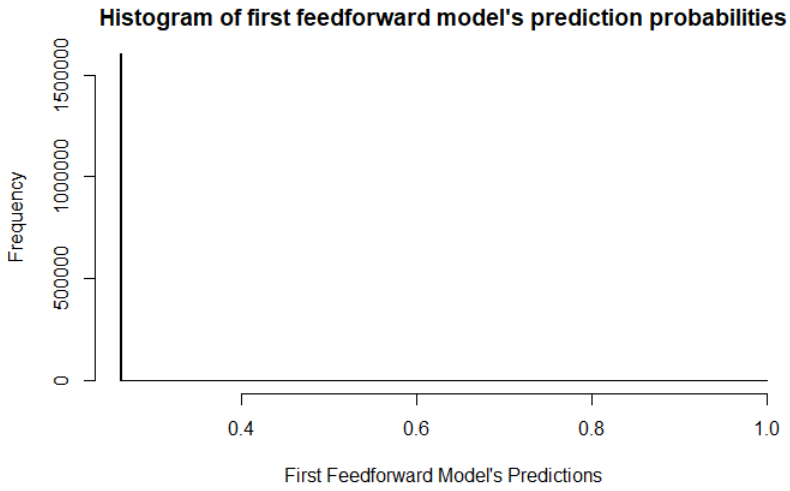
1.1.14 Recurrent Neural Networks

APPENDIX H: OLD RESULTS SECTION

Model 1

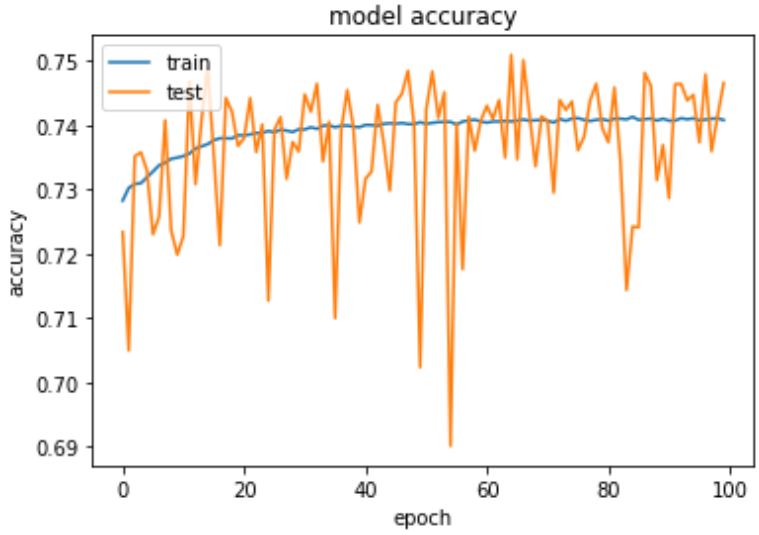


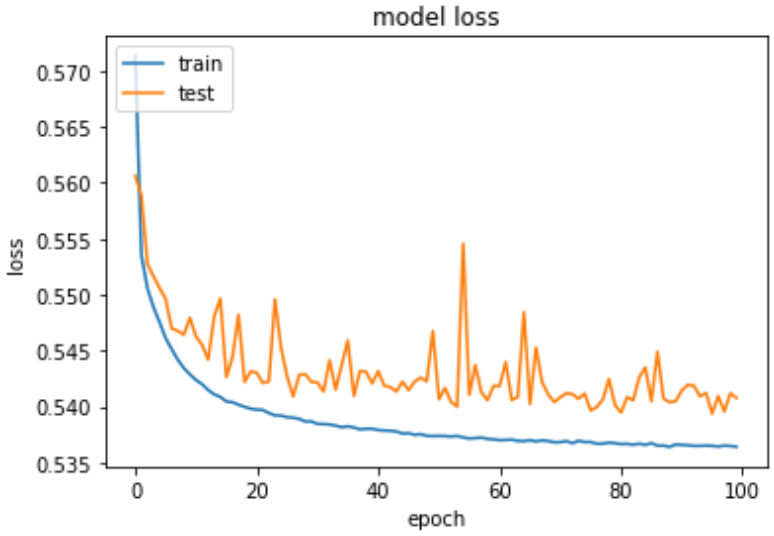


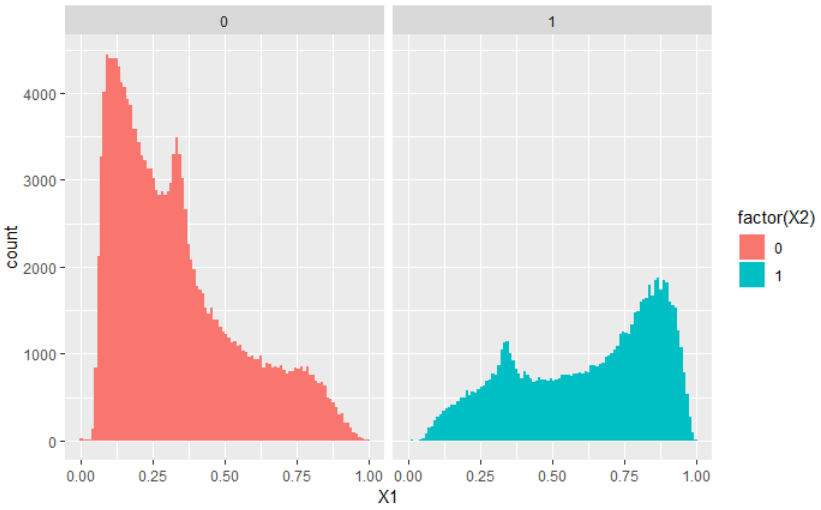


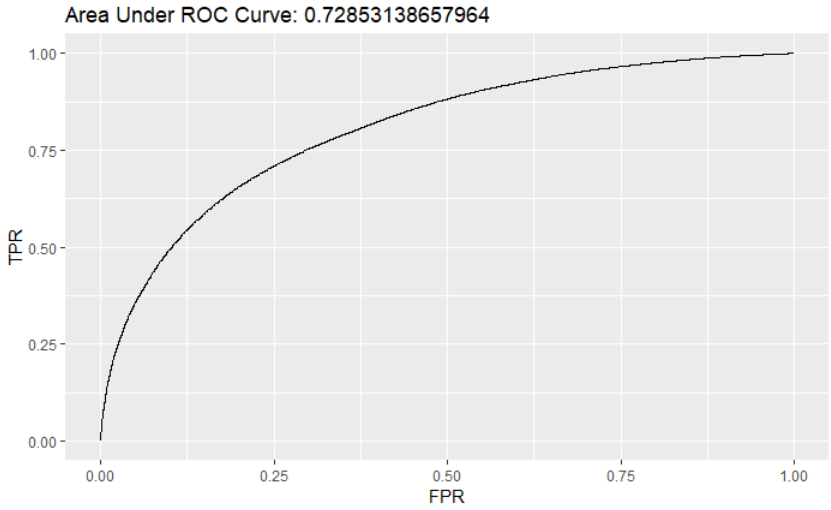
Sequential Model Building

Stage 1

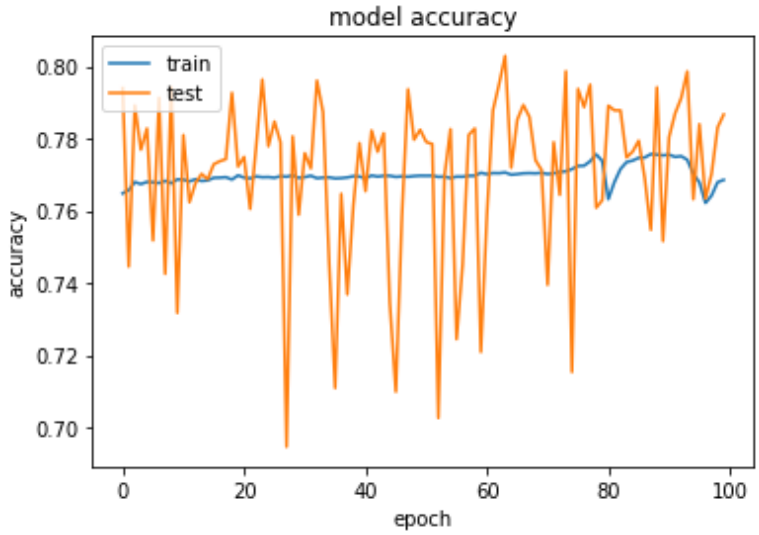


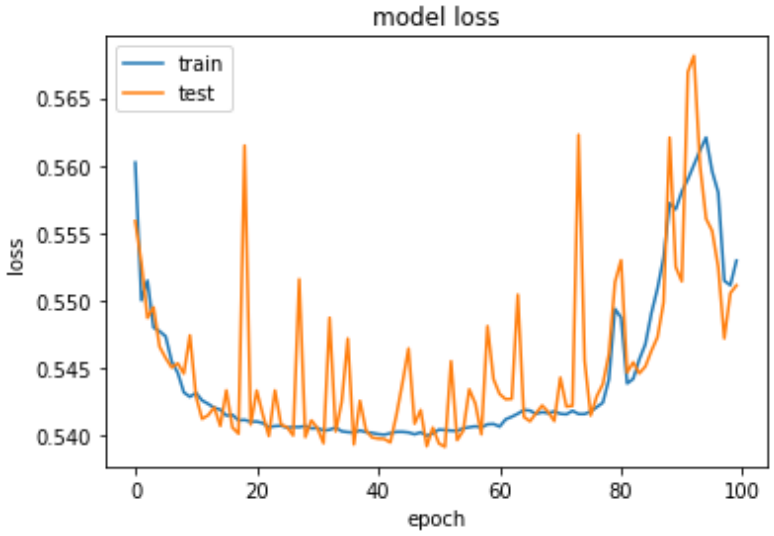

model loss
train
test
loss
epoch
0.570
0.565
0.560
0.555
0.550
0.545
0.540
0.535
0
20
40
60
80
100

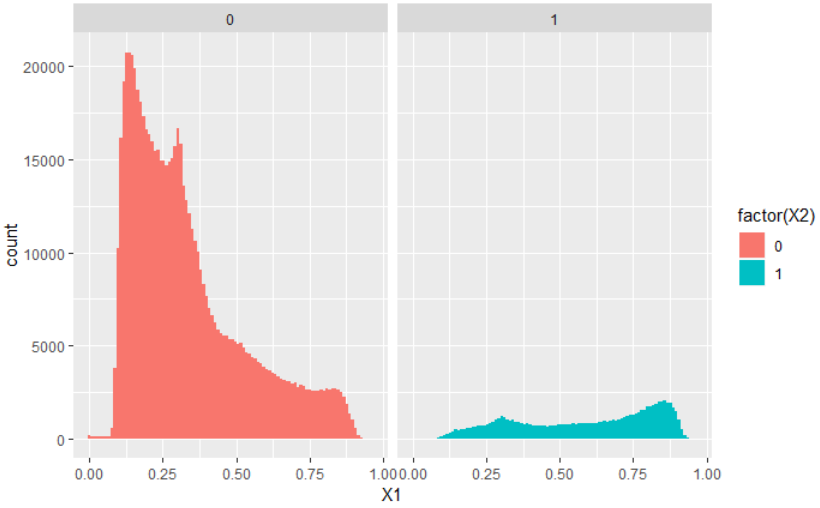

0
1
count
X1
factor(X2)
0
1
4000
3000
2000
1000
0
0.00
0.25
0.50
0.75
1.00

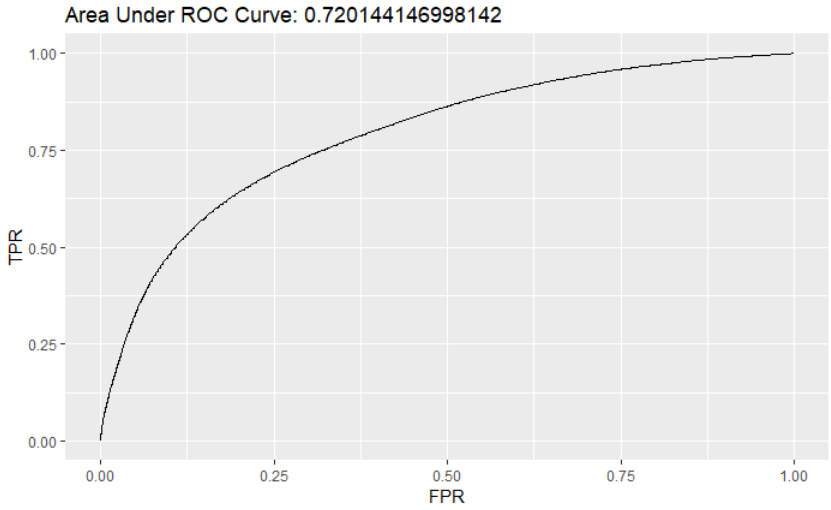

Area Under ROC Curve: 0.72853138657964
TPR
FPR
1.00
0.75
0.50
0.25
0.00
0.00
0.25
0.50
0.75
1.00

Stage 2

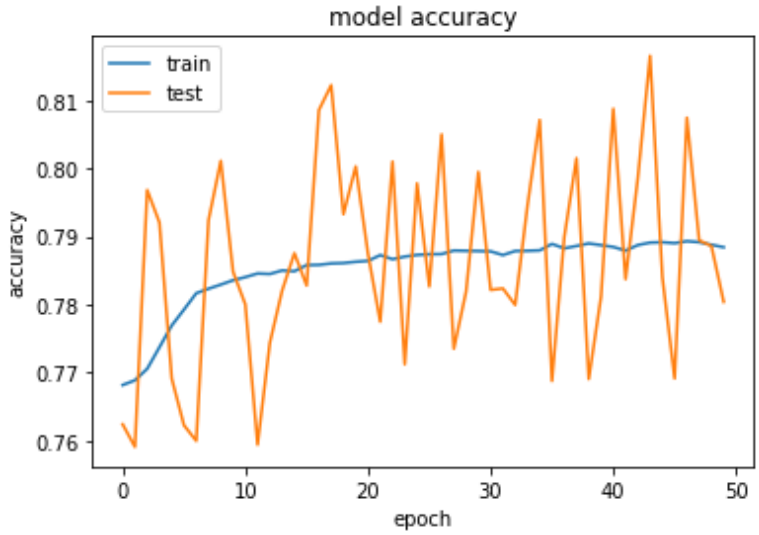

model accuracy
train
test
accuracy
epoch
0.80
0.78
0.76
0.74
0.72
0.70
0
20
40
60
80
100

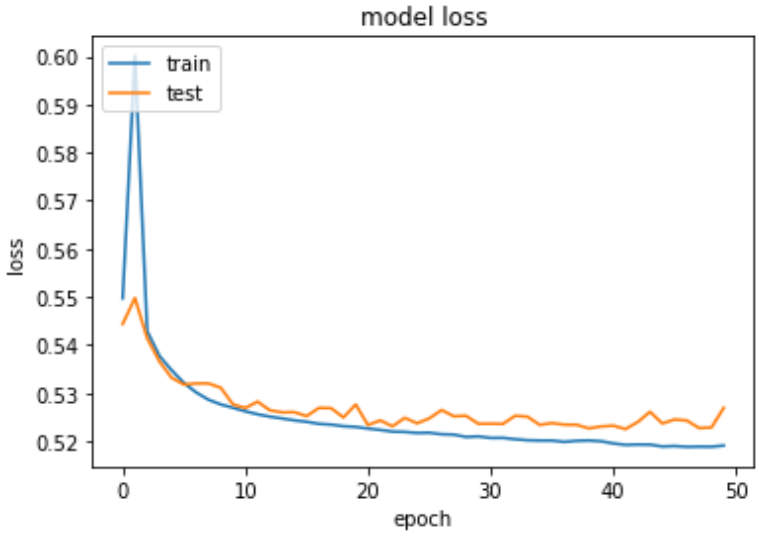

model loss
train
test
loss
epoch
0.565
0.560
0.555
0.550
0.545
0.540
0
20
40
60
80
100

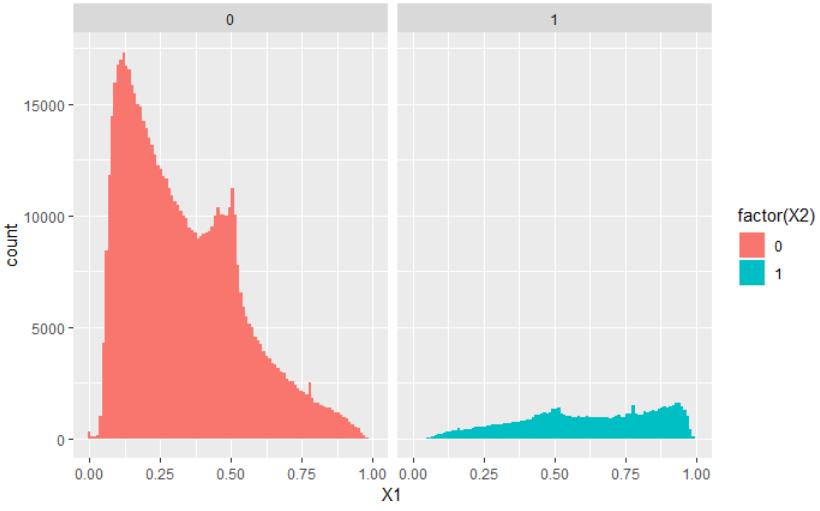

0
1
count
X1
factor(X2)
0
1
20000
15000
10000
5000
0
0.00
0.25
0.50
0.75
1.00

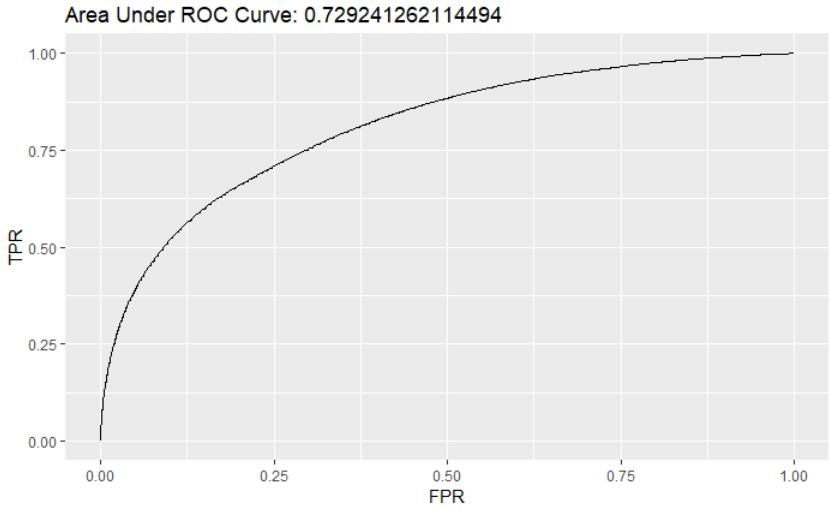

Area Under ROC Curve: 0.720144146998142
1.00
0.75
0.50
0.25
0.00
TPR
0.00
0.25
0.50
0.75
1.00
FPR

Stage 3

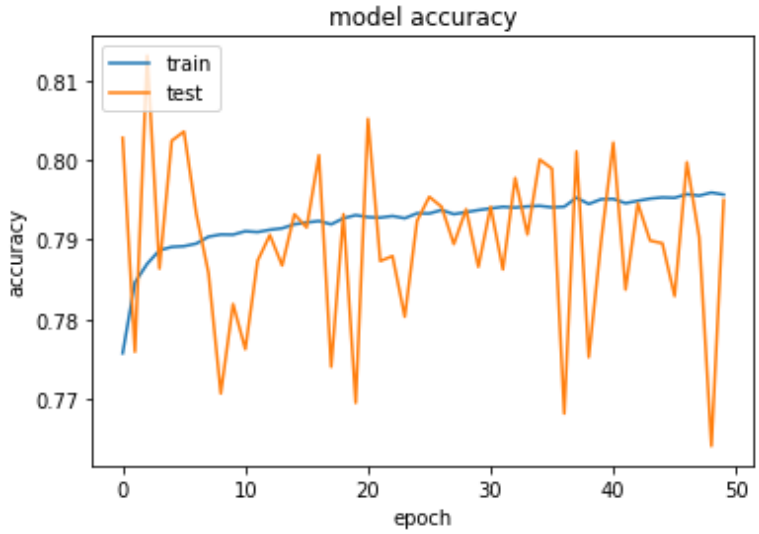

model accuracy
train
test
0.81
0.80
0.79
0.78
0.77
0.76
accuracy
0
10
20
30
40
50
epoch

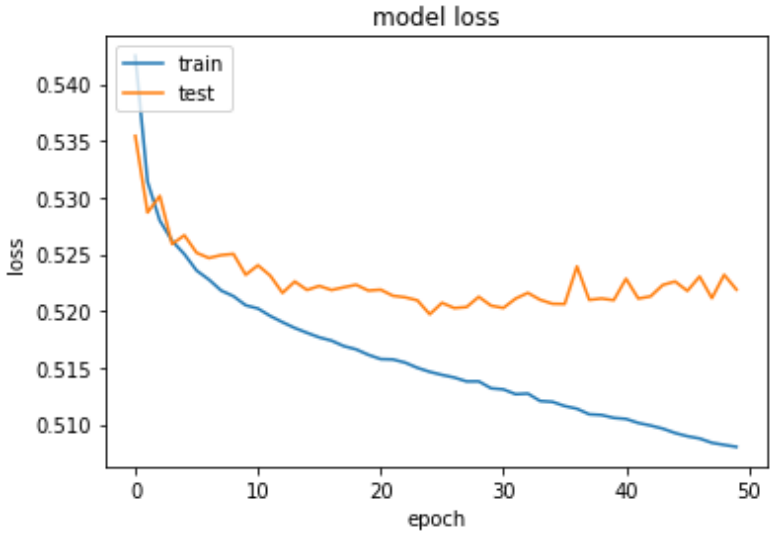

model loss
train
test
loss
epoch
0.60
0.59
0.58
0.57
0.56
0.55
0.54
0.53
0.52
0
10
20
30
40
50


0
1
count
X1
factor(X2)
0
1
15000
10000
5000
0
0.00
0.25
0.50
0.75
1.00


Area Under ROC Curve: 0.729241262114494
TPR
FPR
1.00
0.75
0.50
0.25
0.00
0.00
0.25
0.50
0.75
1.00

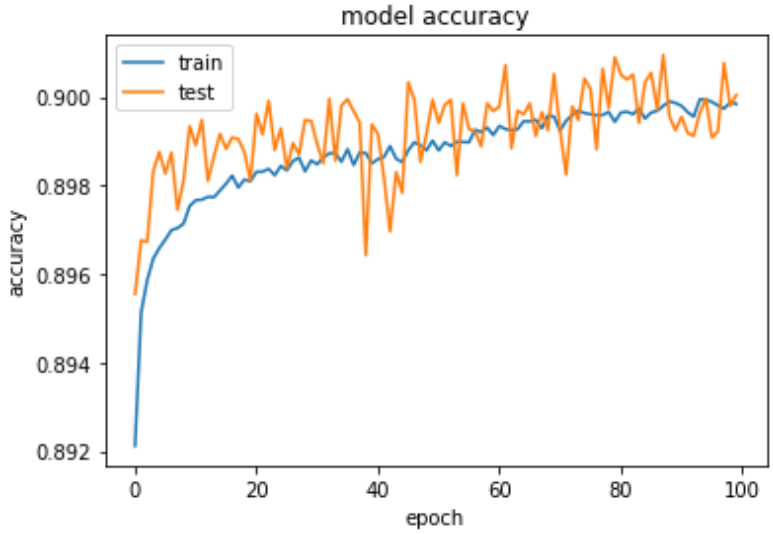
Stage 4

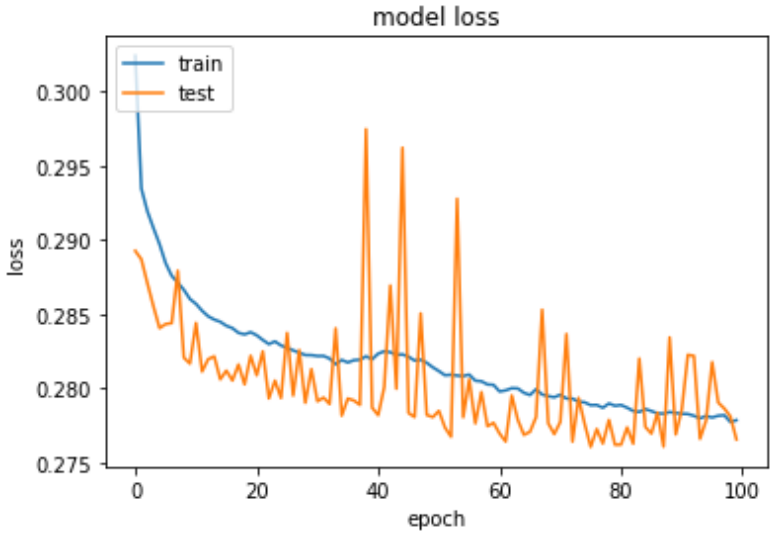

model accuracy
train
test
accuracy
epoch
0.81
0.80
0.79
0.78
0.77
0
10
20
30
40
50



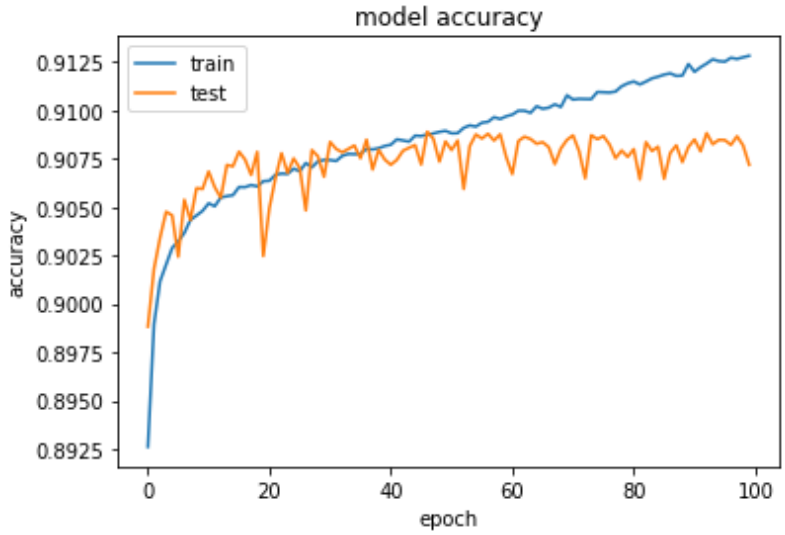
1.1.15 Convolutional Neural Networks

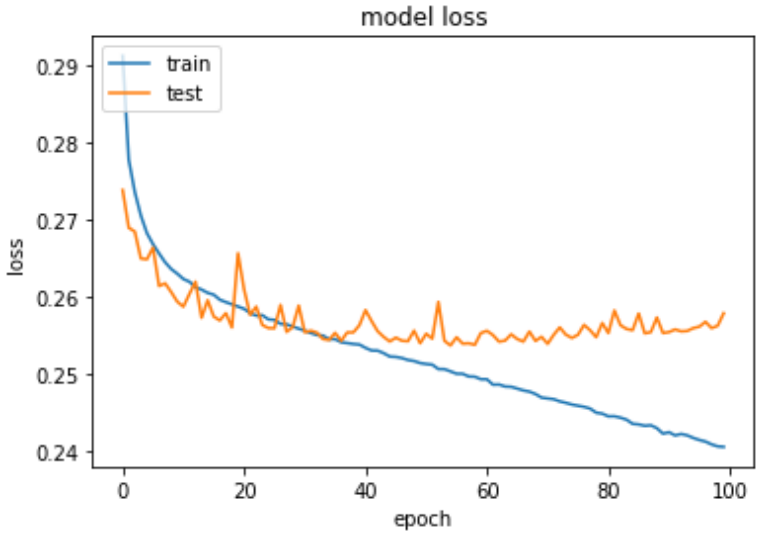
Stage 1



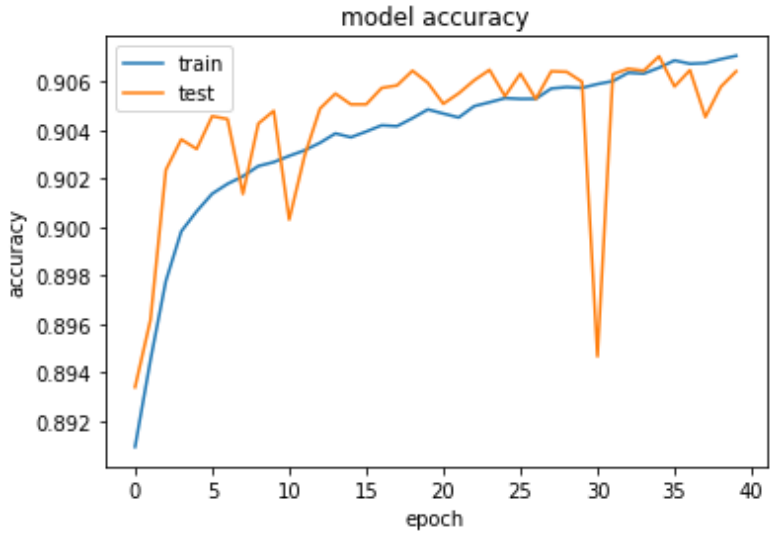

model loss
train
test
loss
epoch
0.300
0.295
0.290
0.285
0.280
0.275
0
20
40
60
80
100

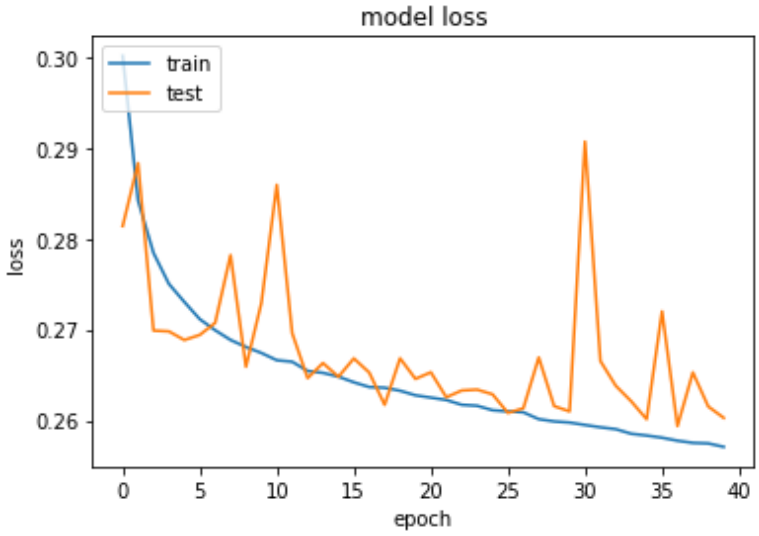
Stage 2


model accuracy
train
test
accuracy
epoch
0.9125
0.9100
0.9075
0.9050
0.9025
0.9000
0.8975
0.8950
0.8925
0
20
40
60
80
100


model loss
train
test
loss
epoch
0.29
0.28
0.27
0.26
0.25
0.24
0
20
40
60
80
100

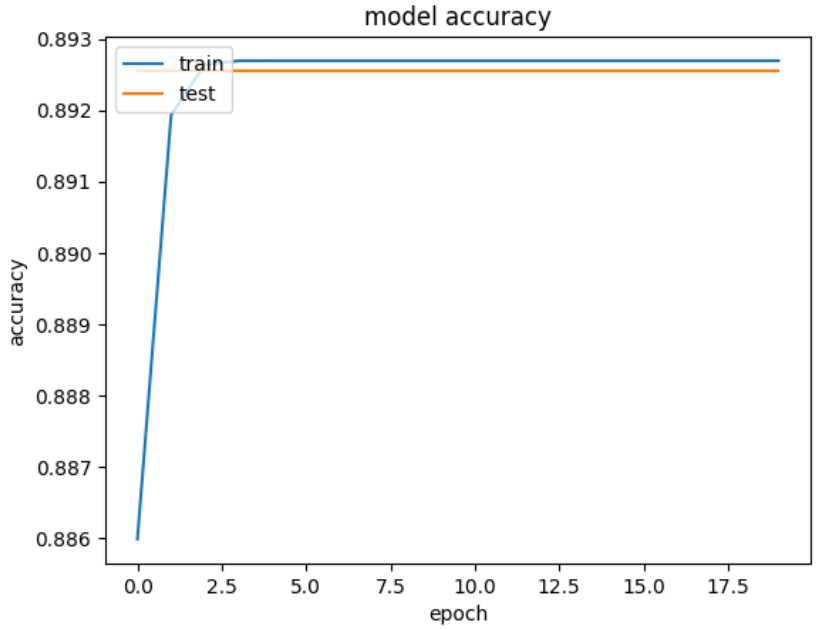
Stage 3

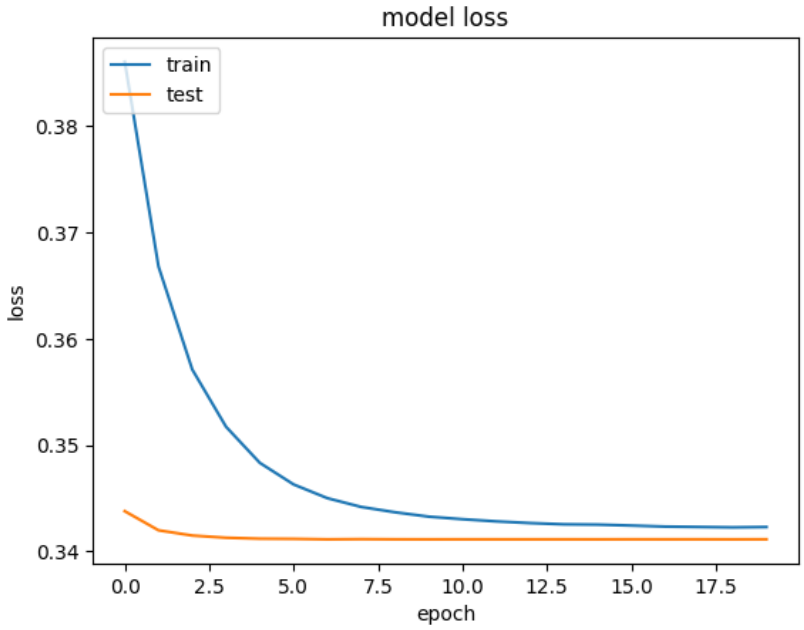

model accuracy
train
test
accuracy
epoch
0.906
0.904
0.902
0.900
0.898
0.896
0.894
0.892
0
5
10
15
20
25
30
35
40



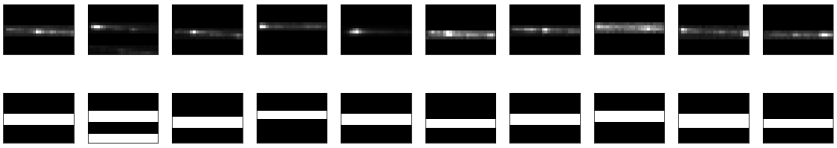
Stage 4

Stage 5





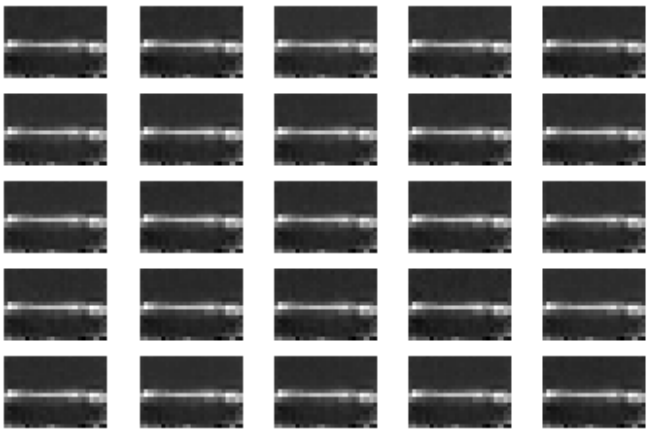
1.2 Autoencoders



Boundary-Seeking Generative Adversarial Network

100 Latent dimensions
Adam optimizer with learning rate = 0.000001 and a batch size of 32
Generator with 8 hidden layers with 128, 256, 256, 256, 512, 512, 512 and 1024 nodes, using leaky ReLU activation and an output layer using tanh activation
Convolutional discriminator using two convolutional layers, max-pooling and 5 hidden layers with 1024, 512, 256, 128 and 64 nodes and a single node output layer with sigmoid activation.

Example output after 29000 epochs:



APPENDIX I: STANDARD MODEL VERTICES

The properties of the bosons in the associated quantum field theory for the various forces of the Standard Model (i.e. QCD for the strong force, EWT (QED for the electromagnetic force and for the weak force), along with their coupling with the spin-half fermions, are illustrated by three-point interaction vertices of a gauge boson with an incoming and outgoing fermion. Each of these interactions also has an associated coupling strength g [1].

A particle will only couple with the force-carrying boson if it carries the interaction's charge, for instance quarks are the only particles that carry colour charge and are therefore the only particles that can participate in the strong interaction with a gluon; similarly, only charged particles can interact with photons; but since all 12 of the fundamental fermions listed in **Error! Reference source not found.** carry the weak isospin charge involved in the weak interaction, they all participate in this interaction [1].

The weak charged-current interaction differs from the other forces in that it is involved in the coupling of different flavour fermions. The $W^+ \, and \, W^-$ bosons carry charges +e and –e respectively, so in order for electric charge to be conserved, this interaction can only occur between pairs of fermions that differ by one unit of electric charge [1].

Figure 9 shows the main Standard model interaction vertices in the form of Feynman diagrams.

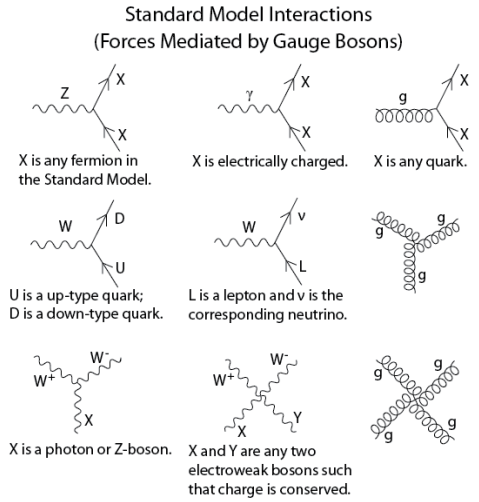


Figure 9: Standard model interaction vertices [2]

APPENDIX J: DATA CENTER

CERN has a data centre with over 174,000 processor cores, 150,000 Terabytes (TB) of Disk space and over 1,000 TB of random access memory (RAM) [14]; this main datacentre is connected both to its extension in Budapest, Hungary and the multi-tier Worldwide LHC Computing Grid (WLCG), all of which operates at a data transfer rate of around 10 Gigabytes/second (GiB/s).

This is wrong:

To calculate the centre-of-mass energy at collision-time, we do:

$$E_{MC} = \sqrt{\bar{s}} = \sqrt{(\sum_{i=1}^{2} E_i)^2 - (\sum_{i=1}^{2} \boldsymbol{p}_i)^2} = \sqrt{2^2 - 6.5^2} = 13 \text{ TeV [1]}$$

This equation is derived from the relativistic relationship between energy and momentum, where the rest energy (invariant mass of a particle) is the familiar $E_0 = mc^2$ and the kinetic energy from acceleration is $p_{tot} = p^2 + c^2$. To simplify the equations, the speed of light, c is set at a constant $c = 1$ [23].

TOTEM and LHCf are smaller experiments focused on particles emitted in the forward direction during non-central collisions, TOTEM investigates particles produced during non-central collisions on either side of the CMS experiment, while LHCf does the same for non-central collisions at the ATLAS experiment [25]. LHCf uses some of these forwardly thrown particles produced at the LHC as a simulated source of cosmic rays to complement the calibration and interpretation of large-scale cosmic ray experiments [29].

MoEDAL is the most recent experiment at CERN and searches for a hypothetical magnetic monopole particle; theoretically envisioned, the magnetic monopole would be a subatomic particle with its own magnetic charge, whose evidence of existence would manifest as extensive damage to the MoEDAL detector [30].

. The ROOT forums allow users of the platform to report bugs and suggest fixes and in this way contribute to the platform without being part of the official development team

Upon installation, running the following line in a Unix terminal

```
> echo $ROOTSYS
```

will print the symbolic path to the top of the ROOT directory, e.g.

```
/Users/gerhard/root
```

Looking at the contents of this directory, $ROOTSYS/bin contains executables such as the main ROOT executable, daemons for remote ROOT file access and authentication of parallel processing capabilities, etc.

$ROOTSYS/lib contains the libraries for the C++ interpreter, image manipulation, ROOT base classes, as well as interfaces with event generators.

Additional directories exist, i.e. $ROOTSYS/tutorials which contains example .C macro files, $ROOTSYS/test which contains .cxx files and $ROOTSYS/include which contains the .h header files.

Looking at the detector geometry in more detail, we first find, closest to the collision area, a central barrel part for measuring photons, electrons, hadrons, as well as a forward muon spectrometer, all of which is embedded in a large solenoid magnet and which covers polar angles between 45°-135°. Moving outward from the first layer of the central barrel, we find an inner tracking system (ITS, Figure 10), consisting of 6 planes of silicon pixel detectors (SPD), silicon drift detectors (SDD) and silicon strip detectors (SSD), which provide for high resolution particle detection [40].

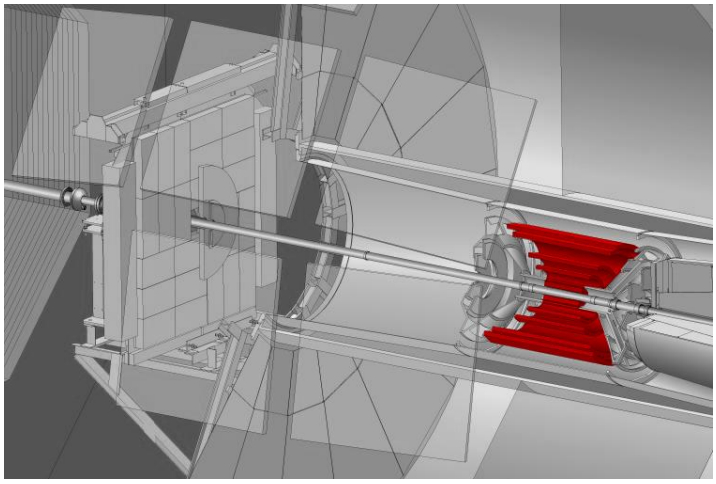


Figure 10: ALICE Inner Tracking System (SPD, SDD, SSD) [39].

The main functions of the ITS are: 1) the reconstruction of secondary vertices in the decay of strange- and heavy flavour particles, 2) particle identification and tracking of particles with low momentum, and 3) improving the resolution of impact parameters and momentum. The outer SSD detectors have analog readout for particle identification via dE/dx (see section **Error! Reference source not found. Error! Reference source not found.**), in the non-relativistic (i.e. low P_T) region.

Next, as we move outwards, we find the Time-Projection Chamber (TPC, Figure 11).

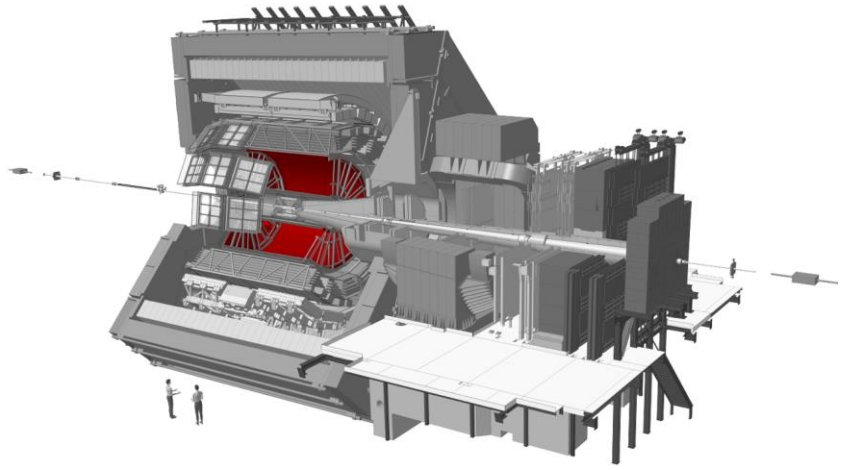


Figure 11: ALICE TPC [39].

As the main tracking detector, the TPC is a conservative system, sacrificing data volume and speed for redundant tracking mechanisms, which guarantee reliable performance, by ensuring good double-track resolution and by minimising space charge distortions [40].

After the TPC, we find three Time of Flight (TOF) particle identification arrays (Figure 12).

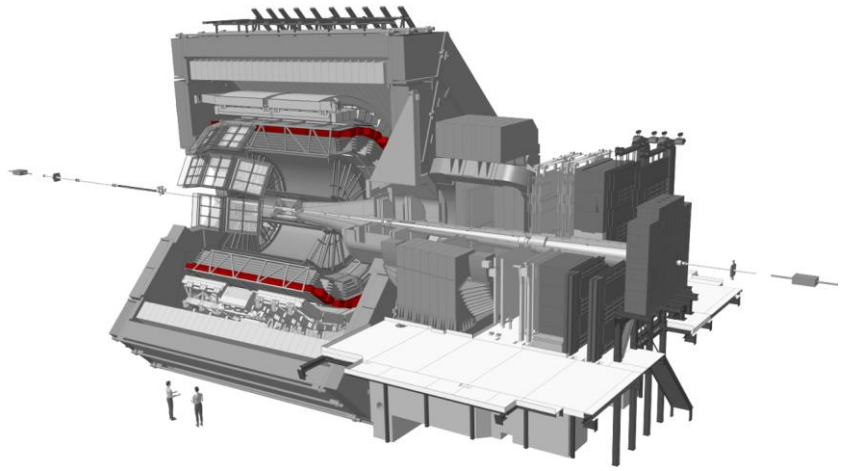


Figure 12: ALICE TOF [39].

Optimized for large acceptance and particle identification in the average momentum range, the TOF covers an area of 140 m² with 160 000 individual cells, the TOF offers time resolution of 100 ps.

Next, we find Ring Imaging Cherenkov Detectors (HMPID, Figure 13).

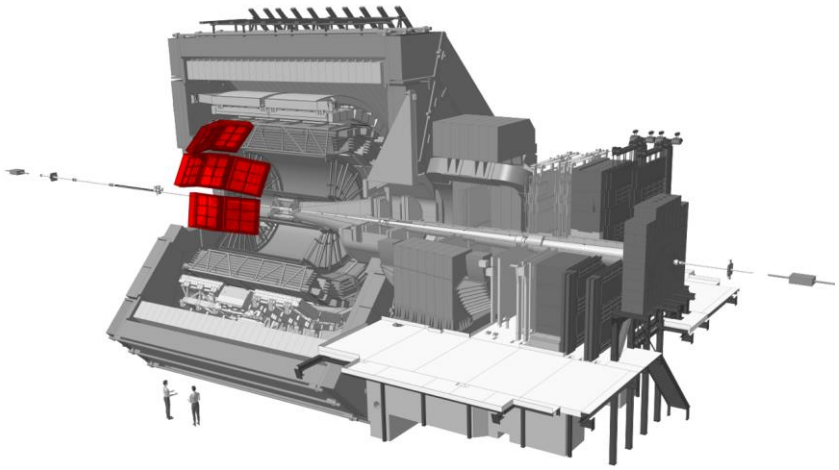


Figure 13: ALICE HMPID [39].

A single-arm detector consisting of an array of proximity focusing ring imaging Cherenkov counters, the HMPID extends particle identification (especially the identification of hadrons) towards a higher spectrum of momentum [40].

After the HMPID detectors, we get to the Transition Radiation Detector (TRD, Figure 14), part of the overarching topics of this dissertation.

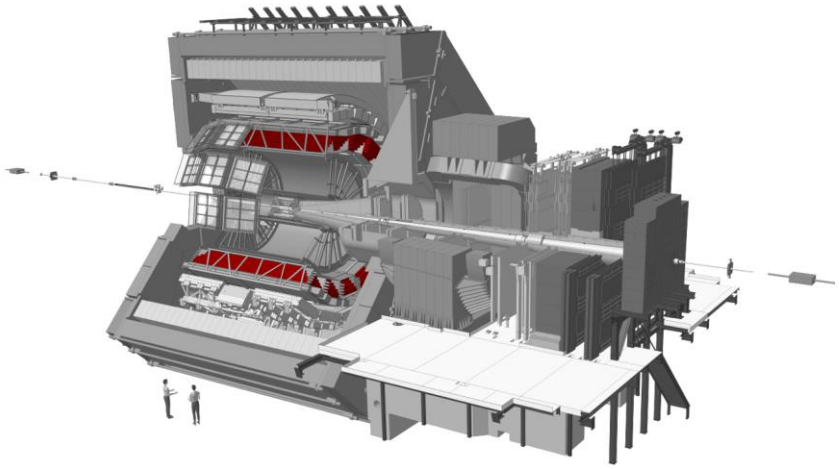


Figure 14: ALICE TRD [39]

The TRD identifies electrons of high momentum, above 1 GeV/c, to quantify production rates of quarkonia and heavy quarks in the mid rapidity (relativistic velocity) range [40]. Six time expansion wire chambers filled with $Xe - CO_2$ are used in conjunction with attendant composite polystyrene radiators to distinguish electrons from other particles by comparing their actual energy deposition in the detector to their characteristic dE/dx curves [40].

The outer layers of the central barrel are occupied by two electromagnetic calorimeters (PHOS, Figure 15, and EMCal, Figure 16).

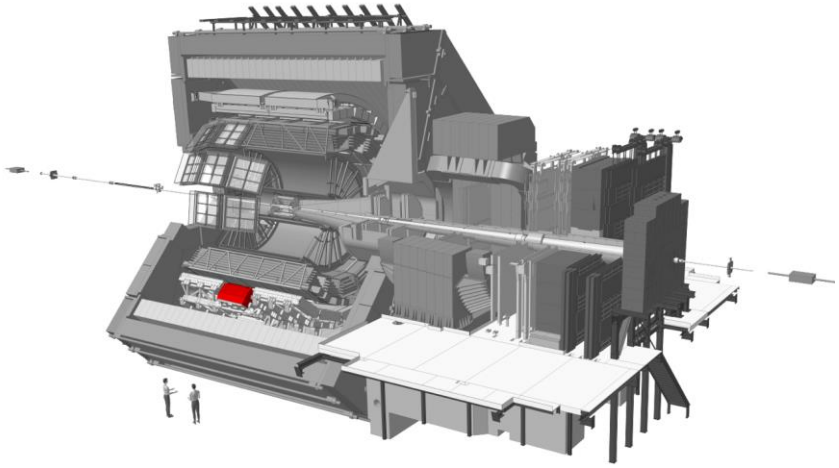


Figure 15: ALICE PHOS [39].

Another single-arm detector, PHOS is an electromagnetic calorimeter which gives a high-granularity and -resolution view of photons, to distinguish their production mechanisms (i.e. whether they arise from thermal emission or hard QCD processes). Scintillating $PbWO_4$ crystals amplify the signal to give good resolution of lower energy photons. Charged particles are vetoed by a set of multiwire chambers, inwardly adjacent to PHOS [40].

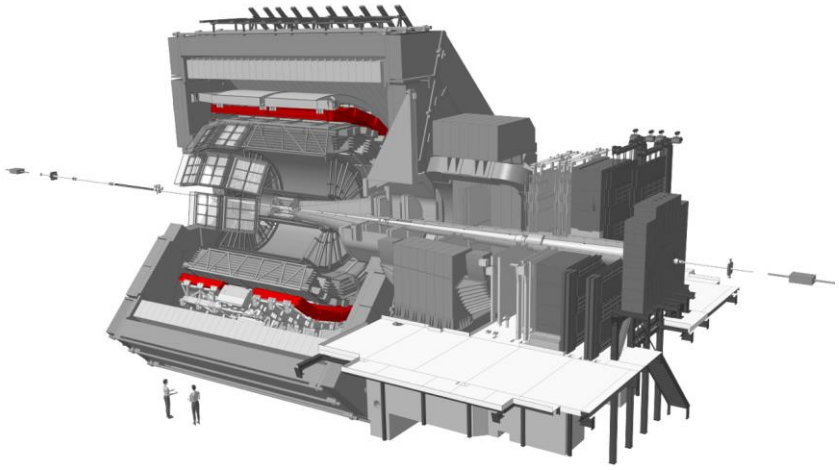


Figure 16: ALICE EMCal [39].

EMCal is a lead-scintillator sampling calorimeter, larger than PHOS, it is used in the measurement of jet production rates and fragmentation functions (functions used to calculate the probability that specific observed final states arise from a given quark or gluon) [40].

All of the detectors in the central barrel, except for HMPID, EMcal and PHOS, cover the full azimuth, i.e. they can detect particles at all angles around the central collision area [40].

Outside of the central barrel, a variety of smaller detector elements are found (V0, T0, PMD, FMD, ZDC) that are involved in the triggering of data collection for a specific event, as well as global event characterization [40].

The forward muon arm (covering angles between 2°-9° relative to the collision centre) completes the picture of the ALICE detector (Figure 17). It consists of 14 planes of triggering and tracking chambers, as well as various muon absorbers and its own dipole magnet [40].

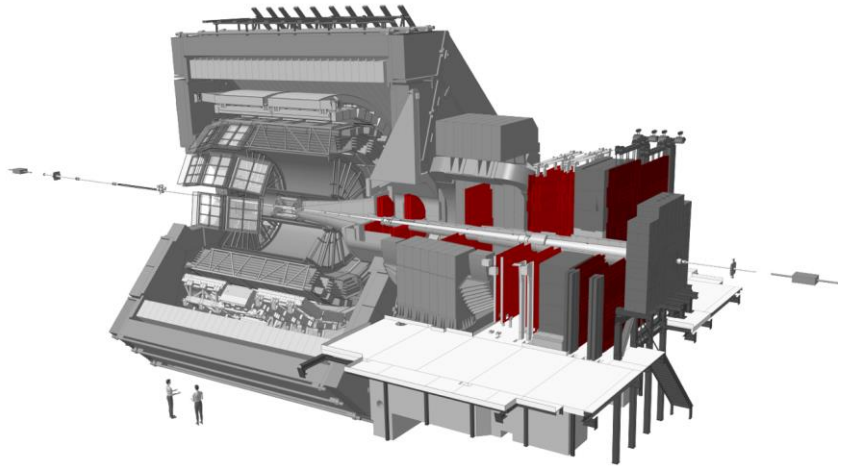


Figure 17: ALICE forward Muon arm [39].

The measurement of heavy-quark resonance production is fulfilled by the Muon spectrometer, its small angle relative to the beam-line allows acceptance down to zero transverse momentum. It is made up of a composite absorber and ten thin cathode strip planes acting as high granularity tracking stations. An additional muon filter and four Resistive Plate Chambers are employed in the processes of triggering and muon identification. The muon spectrometer is protected from secondary particles produced in the beam pipe, by a 60 cm-thick absorber tube [40].

1.2.1 A Note on Geometry

Figure 18 serves as a guide to understanding the coordinate system used at the LHC and in this thesis.

The point of beam intersection (the collision centre) acts as the zero-point in geometric coordinate expressions (x = 0, y = 0, z = 0). Cylindrical coordinates are specified from this origin, with the z-axis pointing along the beam line (with positive z coordinates indicated along this plane in the direction of the muon arm) [40].

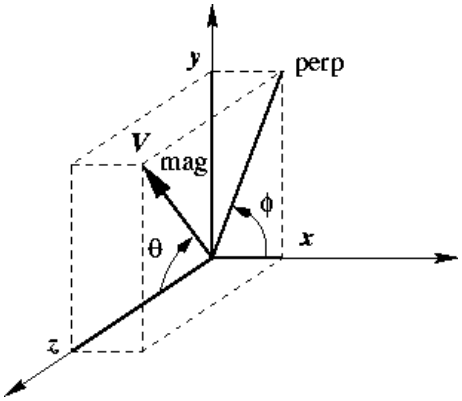


Figure 18: Cylindrical coordinates as used in geometric coordinate specifications for measurements made in experiments conducted at the LHC [42].

Where appropriate, traditional Cartesian coordinates are used, for instance when talking about the location of a detector element. In these cases, the y-axis proceeds from the origin in the direction of the wires in the Multi-Wire Proportional Chambers (MWPC, discussed in section **Error! Reference source not found.**) and also indicates the direction of deflection in the magnetic field, the x-axis proceeds from the origin in the direction of electron drift [40].

In order to specify the cylindrical coordinates (ρ, θ, ϕ) of a point P, one can firstly obtain ρ, by measuring the distance from the origin to point P. Next, one would project a line from P onto a point Q on the xy-plane, to obtain θ, as the angle between the positive x-axis and the line segment from the origin to point Q. Finally, one would calculate ϕ as the angle between the positive z-axis and the line segment from the origin to point P [43].

Commented [GV1]: I don't think I got this right. Phi and Eta seem to be the same value (angle from x-axis)

An additional geometric term used in HEP literature is pseudorapidity, η, which is a specification of a particle's angle relative to the beam (z-) axis.

1.3 Deep Learning within the Context of Artificial Intelligence and Machine Learning

Artificial Intelligence (AI) is a branch of Computer Science concerned with getting computers to perform tasks that are characteristic of those performed by the human mind. The field of AI encompasses both hard-coded rule-based programs (known as the knowledge base approach to AI, which has largely remained ineffective), as well as Machine Learning, which is an approach to AI which aims to get computers to perform these tasks without explicitly coding the solutions for them [38].
The success of Machine Learning algorithms is largely determined by the representation of the data fed through them. Often, a large amount of an AI practitioner's time is dedicated to engineering the right feature-set to hand to a simple machine learning algorithm [38].
Representation learning is a solution to feature generation in which ML is applied, not only to map from a feature set to an output, but also towards automatically learning the most useful representation of the data; usually this representation will encompass identifying the major factors of variation which effectively explain the observed data and discarding those which are not useful to the algorithm [38].
Deep Learning is an approach to representation learning which constructs useful representations based on a combination of simpler representations. In fact, the basic unit of a neural network is the perceptron, which in itself is a very simple function, but once compiled into a Multi-layer Perceptron, the rich texture of the input data distribution can be very accurately captured, since useful features discovered in the first layers of such a neural network can be combined in various ways to create additional useful features [38].

Continuing with the image classification example, an early layer of a convolutional neural network may detect edges in an image, the next layer may detect corners and shadows, and layers further down will ideally detect actual visual elements (faces, car lights, arms, etc.) [38].

In the case of machine learning for image classification, which loosely ties back to some of the aims in this project, it is not always immediately obvious as to which features will be informative to an ML algorithm. For example, feeding raw pixel values into a linear regression model should not be very effective, since images vary in terms of positional information, lighting, sharpness, rotation, etc. [38]

1.4 Mathematical Background for Deep Learning

1.4.1 Rosenblatt's Perceptron

The original Rosenblatt paper [39] outlining the concept of the "perceptron" aimed to develop a theory to explain: 1. How sensory information is detected by biological organisms, 2. how that information is subsequently processed and stored and 3. how mental comprehension or organismal behaviour (which he termed "preference for a particular response") was driven by the first two processes.

He outlined a mathematical framework for these mechanisms, at the hand of the following constructs:
1. **S-points:** sensory units which can possess any of a number of response curves based on the signal strength of incoming information
2. **A-units:** association cells located in an "association area" A_{II}, which in some of his models was preceded by a "projection area" A_I
3. S-points are connected in specific ways to A-units and forward their stimulus response to them, in the form of an inhibitory or an excitatory impulse
4. $\boldsymbol{\theta}$: A threshold value assigned to each A-unit dictates whether it will fire, based on the algebraic sum of excitatory and inhibitory signals received, from either S-points or preceding A-units
5. The connections between S-points and A-units, and between A-units themselves is random, and not all elements of such a network are connected to each other
6. Response units, $R_1, R_2, \dots, R_n$, receive a large number of inputs from the A_{II} set, called its source-set, and have feedback mechanisms to A-units in its source set. [39]

He put forth various models for response curve summation and how these networks would learn [39], but while the mathematical constructs he proposed were oversimplifications of the complexity of biological brains, they were found to be extremely useful in training computers to emulate their capabilities.

For k = $l,\ l-1, \ldots, 1$ hidden layers, $h^{(l)}, h^{(l-1)}, \ldots, h^{1}$, we compute the element-wise gradient on the layer's output (before the non-linear activation function is applied):

$$g \leftarrow \nabla_{a^{(k)}} J = g \odot f'\left(a^{(k)}\right)$$

And the gradients on the weights and the bias term:

$$\nabla_{W^{(k)}} J = g\, h^{(k-1)T} + \lambda \nabla_{W^{(k)}} \Omega(\theta)$$

$$\nabla_{b^{(k)}} J = g + \lambda \nabla_{b^{(k)}} \Omega(\theta)$$

Here, λ represents the weight decay penalty, where the size of the weights are constrained, in a manner inversely proportional to λ . A regularizer $\Omega(\theta)$ is added to the loss, where θ contains all the weight and bias parameters.

This gradient is then propagated to the activations of the preceding layer:

$$g \leftarrow \nabla_{h^{(k-1)}} J = W^{(k)T} g$$

Regularization in deep learning models often involves limiting the capacity (the hypothesis space) of an ANN by introducing a parameter norm penalty $\Omega(\theta)$ to the loss function J. The loss function regularized in this fashion is denoted by J̃, as follows:

$$\tilde{J}(\theta, X, y) = J((\theta, X, y) + \alpha\Omega(\theta)$$

Where α is a weighting hyperparameter, determining the extent of contribution of the parameter norm penalty to the magnitude of the regularized loss function J̃, i.e. setting α = 0 eliminates regularization and increasing its value results in more regularization [38].

Various norms Ω can be used in such a setup and can be applied to the entire set of network parameters θ or a specific subset, e.g. all the weights can be regularized, but all the bias terms can be set to escape regularization, because weights encode the interaction between two variables under a variety of circumstances, whereas bias terms only affect the output of one variable [38].

Ideally, each ANN layer should have its own α coefficient, but doing so increases the search space for the optimal value, so a global α is sometimes used in practice [38].

A Note on Norms

Norms are a means of measuring the size of a vector, by mapping them to non-negative values, by satisfying the following properties:

1. $f(x) = 0 \implies x = 0$
2. $f(x + y) \leq f(x) + f(y)$
3. $\forall \alpha \in \mathbb{R},\ f(\alpha x) = |\alpha| f(x)$

In general, the L^p norm is specified by:

$$||x||_p = \left(\sum_i |x_i|^p\right)^{\frac{1}{p}}$$

L^2: Weight Decay Regularization

The L^2 parameter norm is a simple regularization strategy which shrinks the weights of an ANN closer to the origin by adding the squared and weighted parameter norm penalty

$$\lambda\Omega(\theta) = \lambda||w||_2^2 = \frac{\lambda}{2}w^T w$$

to the objective function [38].

The L^2 norm is known as the Euclidean norm, because it gives the magnitude of the Euclidean distance from the origin to the point defined by x. It is squared in this regularization technique for computational efficiency, because calculating the derivative with respect to each component of the unsquared L^2 norm involves all its elements, whereas the derivative for each component of the squared L^2 norm depends only on the corresponding element of x [38].

Figure 19 (RHS) illustrates the manner in which introducing an L^2 norm penalty introduces an additional constraint on the objective function, i.e. having to minimize the magnitude of the L^2 norm in addition to minimizing the loss function causes the weights to be shrunk, since this larger regularized loss function is interpreted as having higher variance [38].

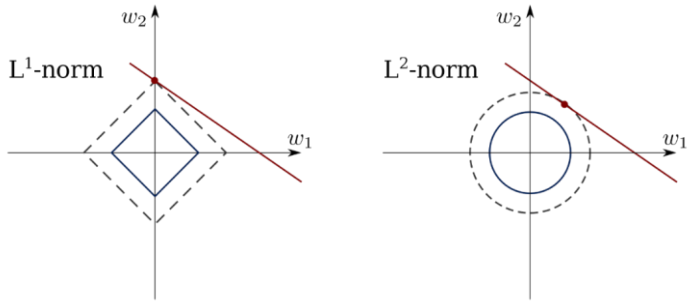


Figure 19: L^1 and L^2 norm penalties

Commented [GV2]: Need to find a better image and explanation

L^1 Regularization

L^1 regularization adds a slightly different weighted parameter norm penalty

$$\lambda\Omega(\theta) = \lambda||w||_1 = \lambda \sum_i |w_i|$$

to the objective function.

When L^1 regularization is used, as the sum of the absolute values of the weights of the ANN increases, the loss function will also increase, as it does for L^2 regularization; but in contrast to L^2 regularization, L^1 allows weights to be shrunk down to zero, resulting in a more sparse neural network, depending on the magnitude of the weighting parameter λ. This phenomenon allows for better feature selection, by reducing the amount of connections in the network and therefore removing the influence of some features on its output [38].

When using either L^1 or L^2 regularization, care has to be taken to select the right level for λ, since a large λ could result in the backpropagation algorithm getting trapped in a local minimum or where the weights are shrunk by so much that they can't impart any useful information to the next layer [38].

Ensembled Models

Bagging and other model averaging techniques involve training a multitude of models and allowing each of them to vote towards the outcome, making use of the principle that a number of Deep Learning models which have each been set up differently should not all make the same "cognitive errors" when learning useful representations to inform accurate predictions on the test set [38].

Bagging, in particular, requires construction of multiple training datasets by sampling with replacement from the full training dataset, resulting in around a third of the full training observations not being present in each of the resampled training sets, and different observations being missing in each [38].

Since random weight initialization and random minibatch selection can result in slightly different weight parameterisation, even when the same architecture is trained multiple times on the same dataset, model averaging is a highly reliable way to reduce overfitting [38].

Boosting is an alternative approach to ensembled methods, which actually increases the capacity of the ensemble by learning based on the variance of previous neural networks by adding additional neural networks sequentially, or even by incrementally introducing hidden units to a single ANN [38].

Early Stopping

By saving the parameter setting at the conclusion of each epoch during training, one can return the network to the parameter setting where the validation error was at its lowest (the point at which the network started overfitting to the training set) [38].

One can also prevent a model from passing that point by specifying early stopping criteria, which will kick the neural network out of training when a defined minimum improvement on the validation error has not occurred for a defined number of epochs [38].
Computational efficiency is maintained by checking the abovementioned conditions at specified training intervals, i.e. not checking whether early stopping criteria have been met after each epoch. Storing parameter settings can be made more efficient by saving in a slower form of memory, such as hard disk space to keep available random-access memory or GPU memory space sufficient for model training [38].

Once early stopping has been reached, the checkpointed model can be trained further by adding the previously held out validation data to the training data and monitoring the objective function as a guide for when to interrupt training [38].

Alternatively, once early stopping criteria are reached, one can retrain a completely new neural network, with the same hyperparameters as the stopped network, for the number of epochs it ran, but this time using the full training + validation data for training [38].

Early stopping is often used in conjunction with other regularization techniques, since it is unobtrusive towards the learning dynamics, i.e. it does not change how the neural network arrives at its optimal weights, it simply changes when to stop adjusting them to prevent overfitting [38].

Full convolutions result from applying enough zero-padding to allow each pixel to be visited k times in each direction of the convolution operation, and therefore should result in an output with m+k-1 pixels. This results in output pixels near the border being influenced by fewer pixels than output pixels near the centre, making the kernel harder to train [38].

The ideal amount of padding generally lies between the amount of padding required to achieve valid- and same convolutions [38].

1.5 Recurrent Neural Networks

Recurrent neural networks (RNNs) are specifically designed to process sequential values. Parameter sharing is an essential aspect of RNNs and facilitate the detection of patterns that could potentially occur in more than one place in the sequence; this family of ANNs also accounts for sequences of differing length [38].

Parameter sharing in RNNs manifest in the form that each element of the output is a function of previous elements of the output within a specified range and is updated using the same rule used to update previous elements of the output [38].

The input vector to an RNN will be vectors $x^{(t)}$ with timestep t consisting of a range from 1, …, τ [38].

Recurrent neural networks extend the concept of a computational graph to include cyclical connections, where the present value of a variable is understood to have an influence on its future value [38].

1.5.1 Computational Graphs

Computational graphs are visual depictions which formalize a set of operations applied to an input vector, for example the computational graph formalizing the ReLU activated output of a hidden unit, i.e. $H = max\{0, WX + b\}$, would look as follows:

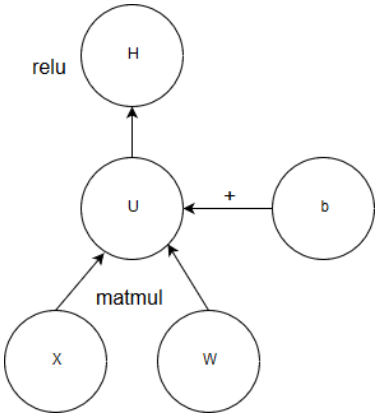


Figure 20: ReLU activated hidden unit in a Neural Network depicted as a computational graph

In RNNs, computational graphs manifest as repetitive chains of operations which result in parameter sharing across neural network architectures [38].

As an example, a dynamical system is classically expressed as:

$$s^{(t)} = f(s^{(t-1)}; \theta)$$

Here, the state of the system at time t, $s^{(t)}$, explicitly depends on its state at the previous time step (t-1).

This graph can be unfolded for a finite number of timesteps, τ, by applying the above expression $\tau - 1$ times, e.g. if τ=3:

$$s^{(3)} = f\left(s^{(2)}; \theta\right)$$
$$= f\left(f(s^{(1)}; \theta); \theta\right)$$

The above equation can be represented as an acyclic graph, which does not make use of recurrence, as follows:

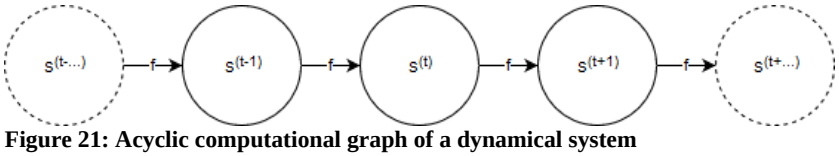


Figure 21: Acyclic computational graph of a dynamical system

If we extend this to express the dynamical system's state at any point being informed by all the previous states of the system, the equation becomes:

$$s^{(t)} = f(s^{(t-1)}, x^{(t)}; \theta)$$

This is the basic formula upon which RNNs are built, where the "states" of the system are the neural network's hidden units, i.e.

$$h^{(t)} = f(h^{(t-1)}, x^{(t)}; \theta)$$

[38].

1.5.2 Long Short-Term Memory

Long Short-Term Memory (LSTM) recurrent neural networks are a highly successful class of RNN which deals with the problem of exploding or vanishing gradients introduced by other RNN implementations by enforcing constant error flow through the internal states of special units, called memory cells, shown in a computational graph in Figure 22 [42].

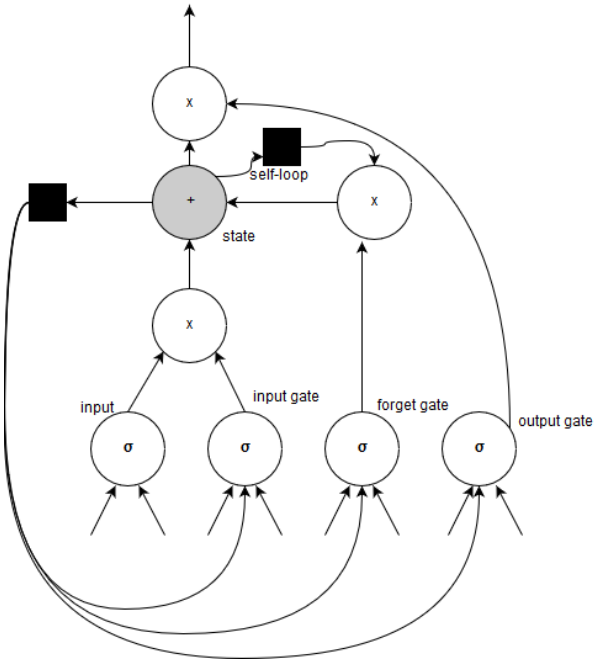


Figure 22: Graph-based representation of special LSTM units

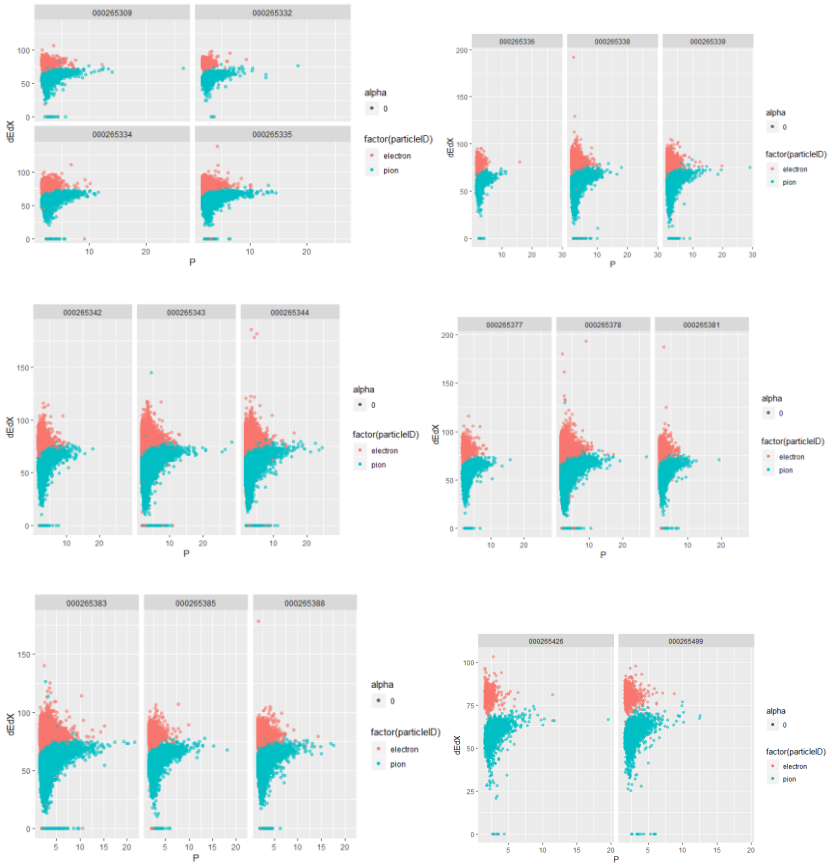
Multiplicative input and output gates protect other units from irrelevant inputs and currently irrelevant stored memory states, respectively. Each memory cell as shown above consists of a central linear unit with a self-connection which is fixed [42]. In this way, a memory cell can decide whether or not to save information about its current state, based on inputs from other memory cells.

1.6 Electron and Pion Counts per Run

What follows below is a graphical overview of the number of electrons and pions which were measured in each run, from which it is immediately obvious that there is an overwhelming majority of pions detected.

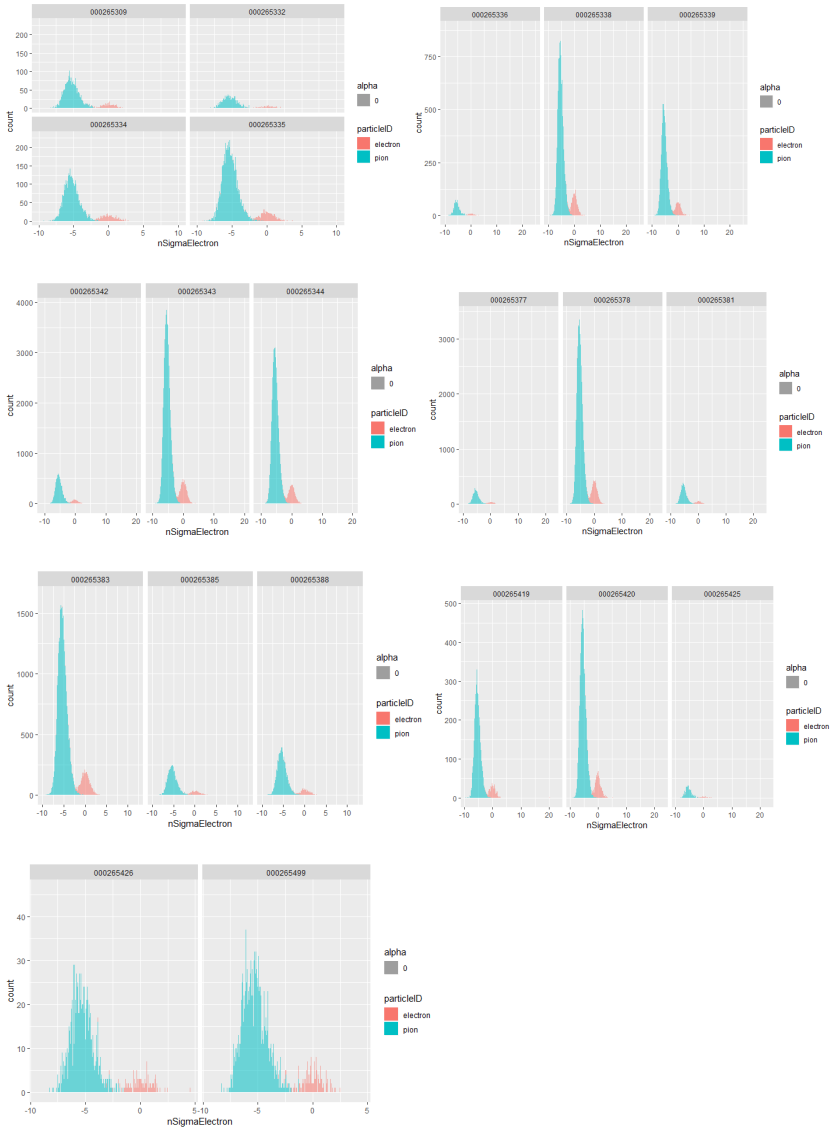
1.7 Bethe Bloch Curve per Run for Electrons and Pions

The plots below depict the energy loss as a function of detector material traversed, i.e. the Bethe-Bloch curves for electrons and pions, respectively. It is clear from these plots that electrons deposit proportionately more energy than pions in the lower GeV range.



1.8 nσ Electron per Run for Electrons and Pions

In the plots below, the statistical estimate for electron and pion identification is depicted, it is clear from the plots below that particles with a low nσ Electron value have been classified as electrons. Pions are centred around an nσ Electron value of around -5.


000265309
000265332
000265334
000265335
000265336
000265338
000265339
000265342
000265343
000265344
000265377
000265378
000265381
000265383
000265385
000265388
000265419
000265420
000265425
000265426
000265499
count
nSigmaElectron
alpha
0
particleID
electron
pion

Unsupervised Feature Extraction

Two forms of Unsupervised Feature Extraction were tested for usefulness in generating features that could be useful during particle identification, i.e. Principal Component Analysis (PCA) and T-distributed stochastic neighbour embedding, the discussion of the mathematical mechanics of these methods lies outside the scope of this thesis. While it does seem promising to use these models to effectively cancel-out some of the noise in the signal by only keeping the major decorrelated factors of variation in the dataset, they are computationally expensive and therefore were not employed to this end. They do however serve as interesting ways to visualize data and their results are therefore shown below.

Principal Component Analysis

It is hard to tell from this perspective whether electrons somehow "surround" pions in this two-dimensional space, based on raw pixel data, or whether they were just plotted last. It might have been possible to find a separating soft margin in kernel space using a Support Vector Machine, but this dataset proved much too large for PCA to run, even on a balanced subset of ~500 000 tracklets.

Figure 23: 2D PCA, pion = 0 (red), electron =1 (blue)

T-distributed stochastic neighbour embedding (t-SNE)

While t-SNE seems to separate the data much better, feeding its results into an SVM are equally unfeasible, but makes for an interesting plot with various clusters of points where particles seem to group, at least seemingly on average, according to their particle IDs.

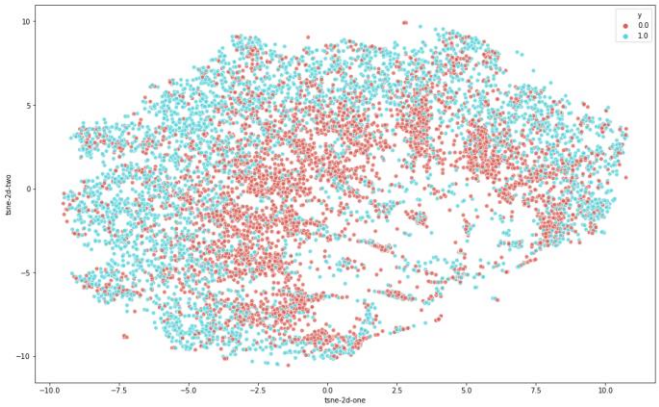


Figure 24: t-SNE

Semi-Supervised Feature Extraction

Autoencoder Dimension Reduction

An autoencoder was built using the H2O framework [51]. Both the input and target values were the flattened pixel values of ~500 000 tracklet images, with an architecture of 256:128:3:128:256, trained for 600 epochs, the autoencoder has to find a vector of length 3 that explains enough variability in the data to reconstruct it with as low as possible loss. If it does well, it should extract 3 features that explain as much variability in the data as possible and are therefore not highly correlated with one another, an approximation of what happens in PCA. Figure 25 and Figure 26 show a 2D and a 3D projection of the weights of the hidden bottleneck layer of width 3. Interestingly, it seems to reverse the encapsulation seen in PCA, i.e. here pions seem to surround electrons, instead of the other way around, but again this could be because of the plotting sequence, since a different plotting library was used.

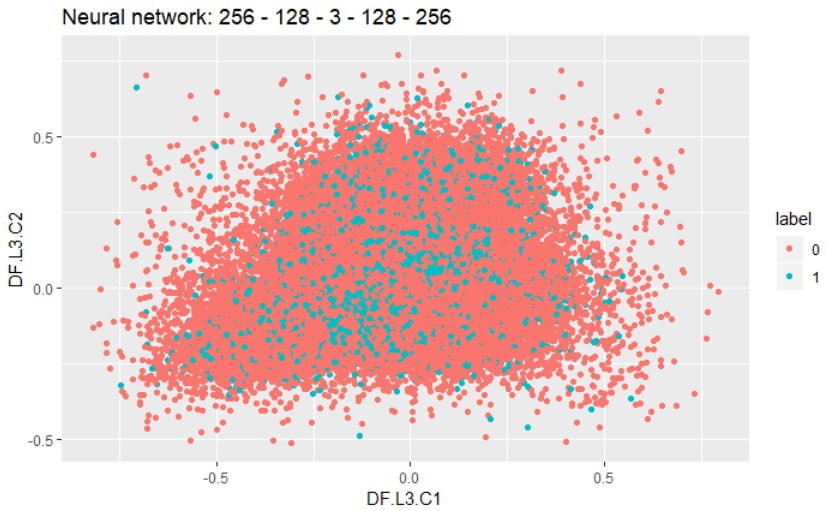


Figure 25: Two Principal Components derived from an AE's latent variables

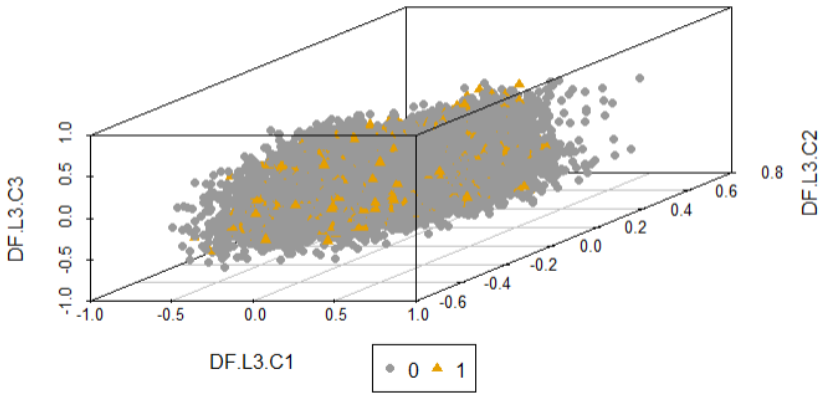


Figure 26: The three Principal Components derived from the same AE's (256:128:3:128:256) latent variables

ROOT is freely available for download from [25] and can be installed using precompiled binaries or built from source using the GNU g++ complier on Unix platforms, such as Linux or MacOSX; Windows 10 64-bit users can make use of the Ubuntu subsystem or locally hosted Linux Virtual Machines to install and use ROOT, but native Microsoft Windows is not supported [24].

To put this into perspective, the Relativistic Heavy Ion Collider (RHIC), located at the Brookhaven National Laboratory in New York, has a circumference of 3.8 km [12], Fermilab's Tevatron, which is no longer in operation, was 6.3 km in circumference [13] and the KEKB accelerator in Tsukuba, Japan also has a circumference of around 3 km [14].

It is also the most powerful particle accelerator in the world, compared to RHIC, which operates at $E_{CM} \approx 200\ GeV$ [12], the Tevatron, which reached $E_{CM} \approx 1.8\ TeV$ [13] and KEKB at $E_{CM} \approx 10.58\ GeV$ [16].

1.9 Stage 1 of Model Building: Establishing Naïve Benchmarks & Generating Features by Hand

The initial approach towards particle identification entailed manual feature generation. Initially a feature set was created, which took as input the following variables:

- Column sums of image pixels
- Five number summaries (Minimum, First Quartile, Median, Mean, Third Quartile and Maximum values) for the non-zero pixels in an image
- The number of non-zero rows in an image
- The column means of images in pixels
- The row means of images in pixels

Applying column-wise scaling and centering, to normalise the dataset as follows:

$$c = \frac{c - mean(c)}{sd(c)}$$

A linear regression model was used as a very simple benchmark to compare future results against, using the abovementioned dataset, which contained around 99% pions. Using a naïve t_{cut} of the 99th percentile of the output distribution of the linear model, a single tracklet pion efficiency of $\varepsilon_\pi = 0.91\%$ at electron efficiency $\varepsilon_e = 3.92\%$ was achieved.
Following this, t_{cut} was optimized by maximizing the area under the ROC curve when assessing the linear regression model's results, which improved electron acceptance, but decreased pion rejection results to $\varepsilon_\pi = 24.92\%$ at electron efficiency $\varepsilon_e = 66.85\%$.
A benchmark for deep learning was also established using this dataset, by upsampling electrons with replacement to arrive at a sample of equal size to the pion sample.
A fully connected neural network with architecture 256:128:64:32:16:2, with ReLU activations in the hidden layers and softmax activation in the output layer was trained for 500 epochs and achieved $\varepsilon_\pi = 13.15\%$ at $\varepsilon_e = 62\%$. This model's accuracy and loss curves are shown in Figure 28.

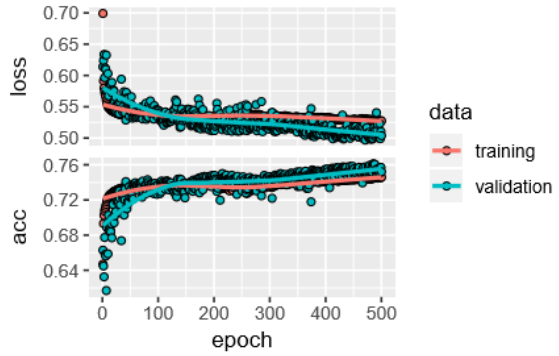


Figure 27

Figure 28: Training and Loss Curves for Naïve Benchmark Deep Learning Model

Next, an additional feature set was created as follows:

- The sum of all pixel values in each image
- Column sums of pixel values, scaled by dividing by the mean value of the entire matrix
- The lagged differences of the time evolution of pulse height obtained in the previous feature set, with both lag=1 and lag=2
- Further binning of the column sums, by adding the window sum of three columns at a time as an additional 8 features

A neural network with the following architecture 512:512:256:128:2, with ReLU activation functions in the hidden layers and softmax activation in the output layer managed to increase electron acceptance slightly, but with the attendant sacrifice of very high pion contamination, i.e. $\varepsilon_\pi = 63.1\%$ at $\varepsilon_e = 68\%$.
It should be noted that class imbalances were not being accounted for at this stage and therefore the accuracy and loss curves shown in Figure 29 appear misleadingly successful. In addition t_{cut} was not optimised and was set to $t_{cut} = 0.5$ and the probabilities for single tracklets were at this stage not combined to reconstruct Bayesian probabilities for the full track case as explained in **Error! Reference source not found.**.
Figure 30 shows that there is not much separability in the probability distribution obtained from this model, with most estimates lying between 0.3 and 0.6.

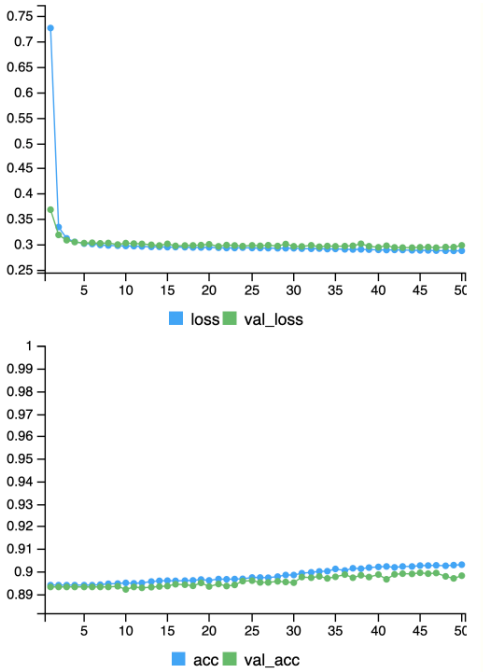


Figure 29

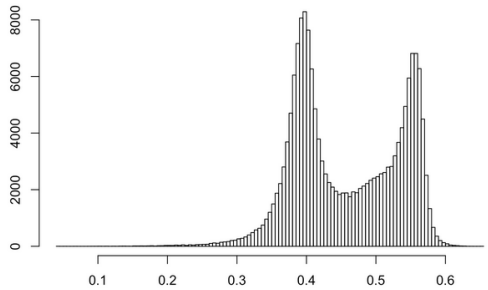


Figure 30

1.9.1 1D Convolutional Neural Networks

Figure 36, Figure 37, Figure 38, Figure 39 and Figure 40 summarise the results obtained for 1D Convolutional Neural Networks during the second stage of model building.

Although much fewer model 1D Convolutional architectures will built, a deeper convolutional architecture seems (from looking at Figure 36) to be a good strategy in solving the problem of particle identification. The best-performing model had four 1D Convolutional Layers, but it didn't perform that much better than the other models which had less convolutional layers. It was also the only model with such a deep convolutional architecture, so it's hard to say whether low pion efficiency truly depends that heavily on the depth of the convolutional architecture of a CNN.

Interestingly, all 1D CNN models used Tanh activation functions in the hidden dense architecture, whilst 2D CNNs mostly relied on Sigmoid activations in their hidden dense layers. The range of outputs from a Tanh activation is [-1,1], whereas the range of a Sigmoid activation is [0,1] (see Figure 31). The Tanh range is more symmetrical than that of the Sigmoid activation function and maps negative input values to negative output values, which provides for stronger gradients. Since the sigmoid function can only output positive values, it saturates faster (i.e. if the pre-activation input to a sigmoid function is close to zero, the gradient on that neuron might vanish, rendering it untrainable).

It is possible that the high performance of 1D CNNs, despite the fact that they were fed data with less information (column sums, instead of full images) can be partly explained by the choice of activation function. In addition, the fact that the input data to 2D CNNs are extremely sparse, with most pixels being close to zero could have rendered a lot of the neurons in the early layers saturated at zero and therefore untrainable. Perhaps a better normalisation strategy in the case of 2D CNNs could have been: $x = \frac{x - mean(x)}{sd(x)}$, which would have resulted in a zero centred and scaled pixel distribution, instead of an arbitrarily distributed pixel distribution in the range [0,1], especially considering the choice of activation function.

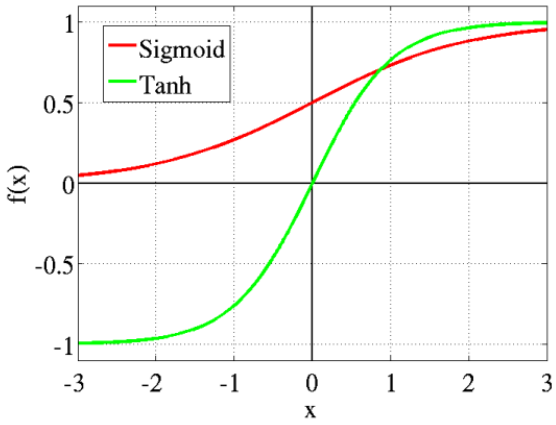


Figure 31: Sigmoid vs Tanh activation

Another thing to consider is that no activation functions were applied to the Convolutional Layers in all the 1D CNNs that were built, i.e. $\phi(x) = x$, it is not possible to discern what the effect of this strategy was when comparing 1D CNNs, since all these models were built with no activation functions, but it's worth noting that the fourth worst 2D CNN also employed this strategy, although this is entangled with the high learning rate that was used in this particular 2D CNN.

1.9.2 LSTM Networks

An interesting feature of this dataset is that it can be framed in multiple ways (as an image for 2D CNNs, as an array for Dense Fully Connected Neural Networks and as a timeseries for LSTM Networks).

By transposing each image matrix, an input feature set with the following dimensions can be obtained : $n \times 24_{timesteps} \times 17_{features}$. LSTM Networks did give the second lowest pion efficiency of all models trained, but one might have expected them to perform better than 2D CNNs, since a TRD signal is technically more of a timeseries than an image. Perhaps their inferior performance can be explained by noise in the dataset (perhaps summing 3 columns sequentially, but maintaining the number of rows could have been a better feature set, but generally one would hope that a deep learning model will do that kind of feature extraction by itself). There is not much explainable variability in terms of number of layers, number of nodes, etc. It might be deduced that using LSTM layers that go backwards is not a successful strategy, since the models that employed such a strategy were among the lower end of the performance spectrum.

1.9.3 Dense (Fully Connected) Networks

These networks resulted in the lowest performance of all models, but very few of them were built. Using a deeper architecture with a lower learning rate was slightly more successful than other strategies, and designing a custom set of features was also not successful (Model M3 in **Error! Reference source not found.**), but this aspect is confounded with a high learning rate and very small architecture. These plots are included mainly for the sake of completeness.

Chapter 1: Appendices

Whilst they may not be very interpretable, the weights for the four convolutional layers of this model are plotted in Figure 32, Figure 33, Figure 34 and Figure 35 below.

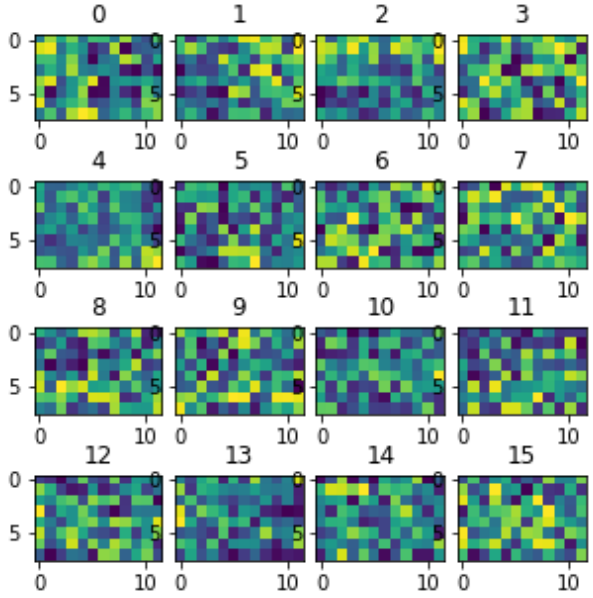


Figure 32: Weights of first convolutional layer

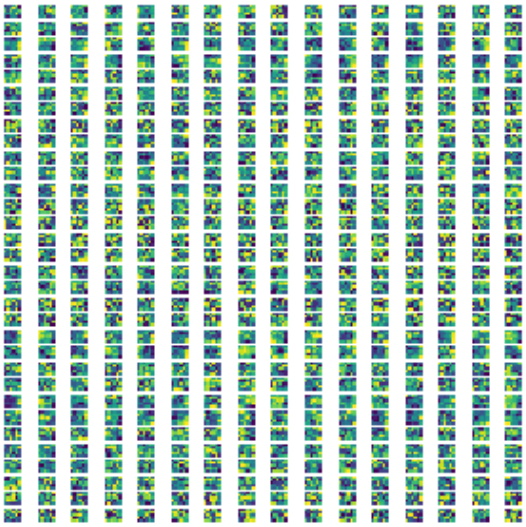


Figure 33: Weights of second convolutional layer

Figure 34: Weights of third convolutional layer

Figure 35: Weights of fourth convolutional layer

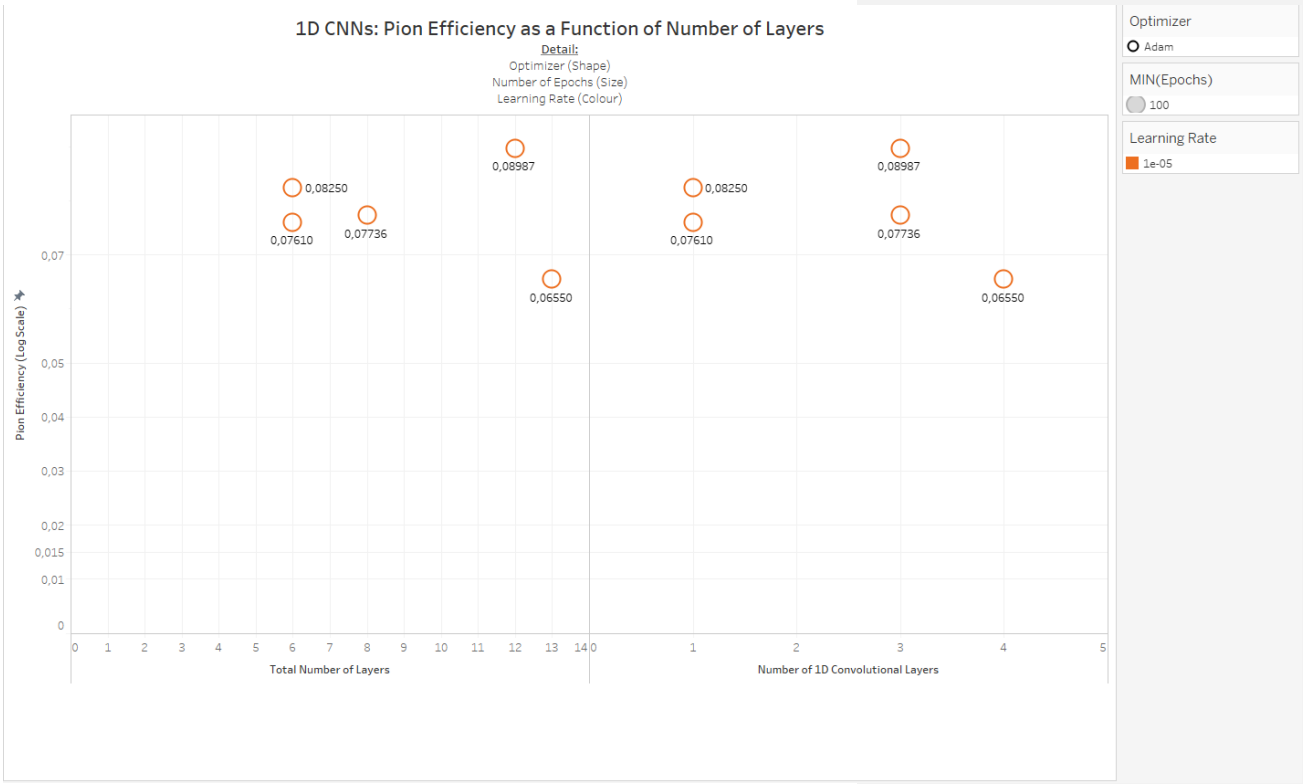

1D CNNs: Pion Efficiency as a Function of Number of Layers
Detail:
Optimizer (Shape)
Number of Epochs (Size)
Learning Rate (Colour)
Pion Efficiency (Log Scale)
0,07
0,05
0,04
0,03
0,02
0,015
0,01
0
0,08250
0,07610
0,07736
0,08987
0,06550
0,08250
0,07610
0,08987
0,07736
0,06550
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14
Total Number of Layers
0 1 2 3 4 5
Number of 1D Convolutional Layers
Optimizer
Adam
MIN(Epochs)
100
Learning Rate
1e-05

Figure 36

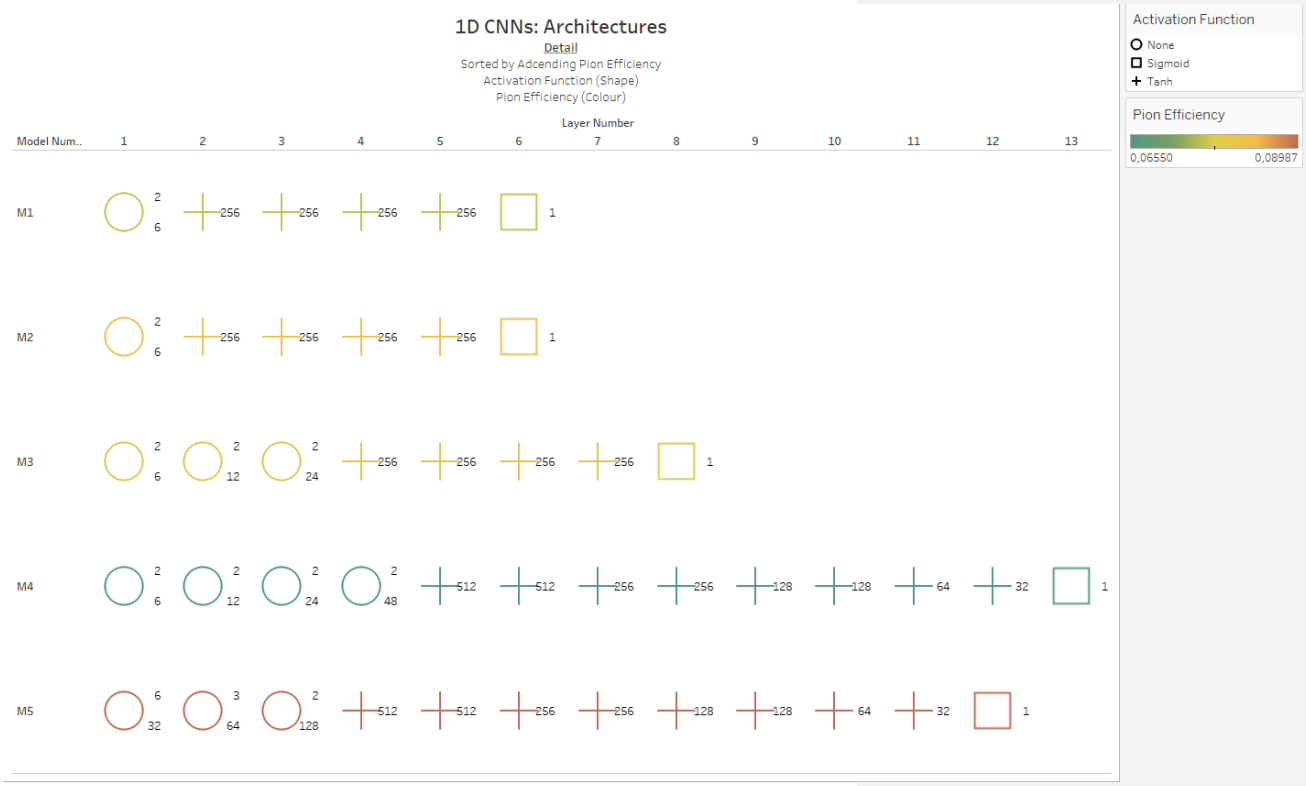

1D CNNs: Architectures
Detail
Sorted by Adcending Pion Efficiency
Activation Function (Shape)
Pion Efficiency (Colour)
Layer Number
Model Num..
1
2
3
4
5
6
7
8
9
10
11
12
13
M1
M2
M3
M4
M5
Activation Function
None
Sigmoid
Tanh
Pion Efficiency
0,06550
0,08987

Figure 37

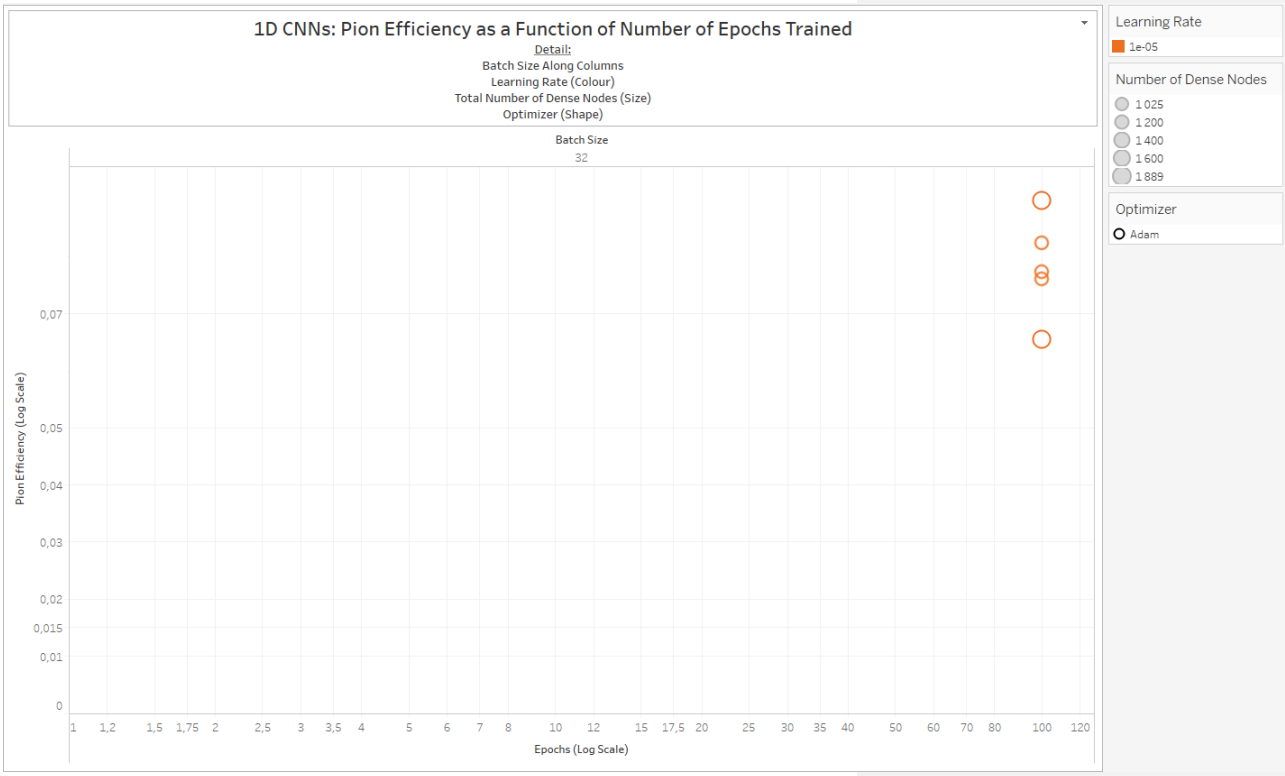

1D CNNs: Pion Efficiency as a Function of Number of Epochs Trained
Detail:
Batch Size Along Columns
Learning Rate (Colour)
Total Number of Dense Nodes (Size)
Optimizer (Shape)
Batch Size
32
Pion Efficiency (Log Scale)
0,07
0,05
0,04
0,03
0,02
0,015
0,01
0
1
1,2
1,5
1,75
2
2,5
3
3,5
4
5
6
7
8
10
12
15
17,5
20
25
30
35
40
50
60
70
80
100
120
Epochs (Log Scale)
Learning Rate
1e-05
Number of Dense Nodes
1 025
1 200
1 400
1 600
1 889
Optimizer
Adam

Figure 38

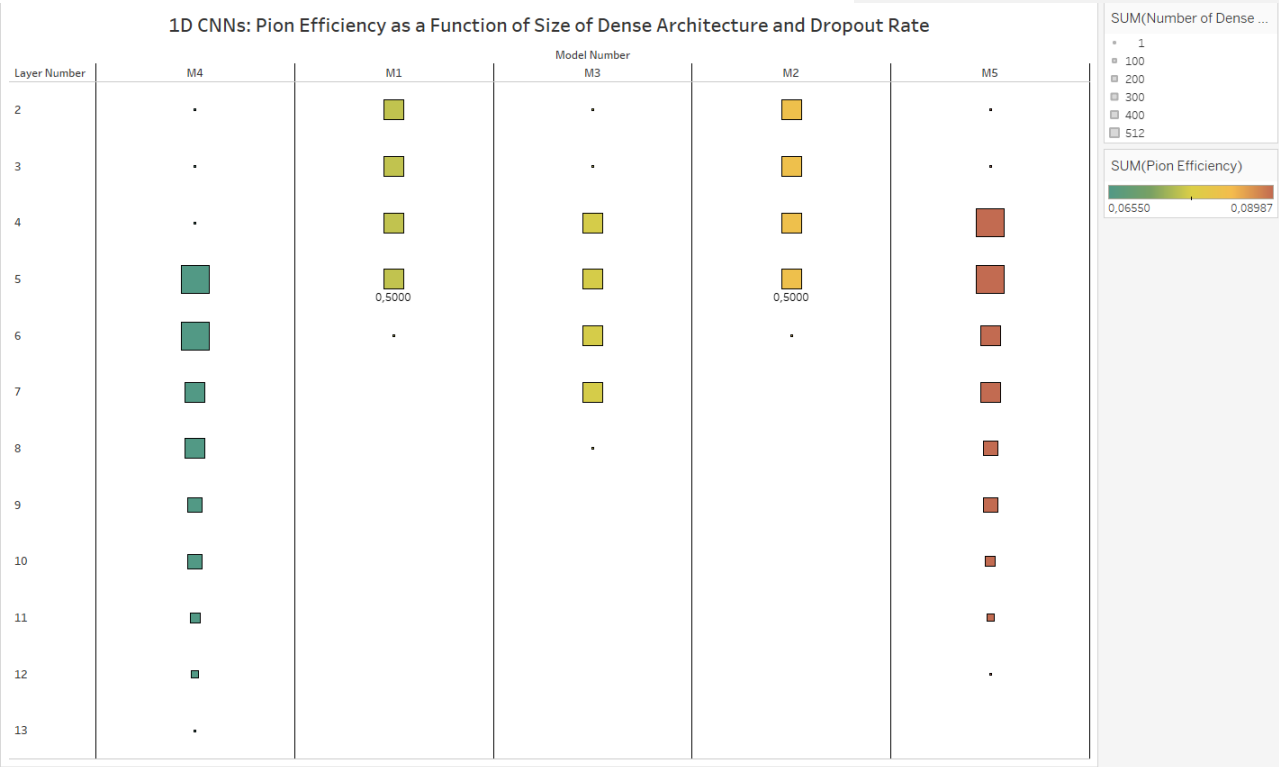

1D CNNs: Pion Efficiency as a Function of Size of Dense Architecture and Dropout Rate
Model Number
Layer Number
M4
M1
M3
M2
M5
2
3
4
5
6
7
8
9
10
11
12
13
0,5000
0,5000
SUM(Number of Dense ...
1
100
200
300
400
512
SUM(Pion Efficiency)
0,06550
0,08987

Figure 39

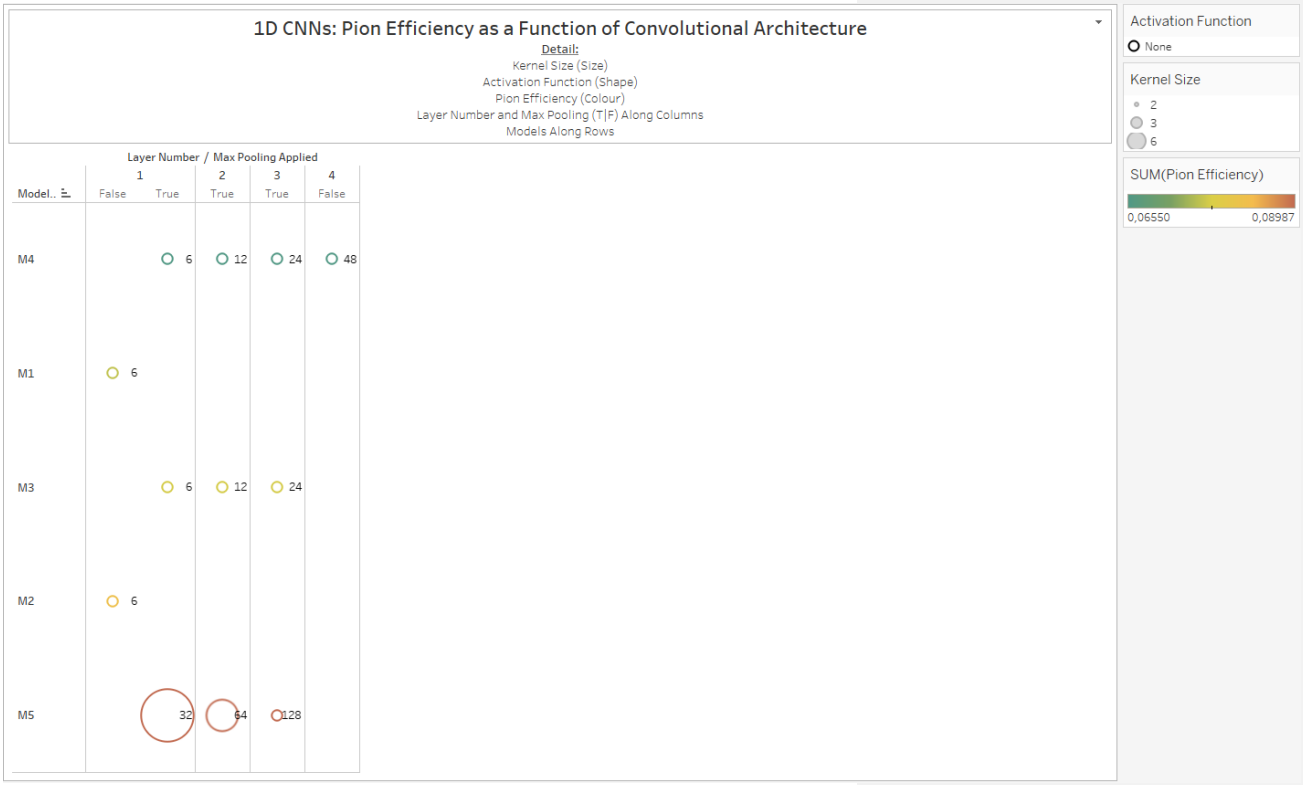

1D CNNs: Pion Efficiency as a Function of Convolutional Architecture
Detail:
Kernel Size (Size)
Activation Function (Shape)
Pion Efficiency (Colour)
Layer Number and Max Pooling (T|F) Along Columns
Models Along Rows
Layer Number / Max Pooling Applied
1
2
3
4
Model..
False
True
True
True
False
M4
6
12
24
48
M1
6
M3
6
12
24
M2
6
M5
32
64
128
Activation Function
None
Kernel Size
2
3
6
SUM(Pion Efficiency)
0,06550
0,08987

Figure 40

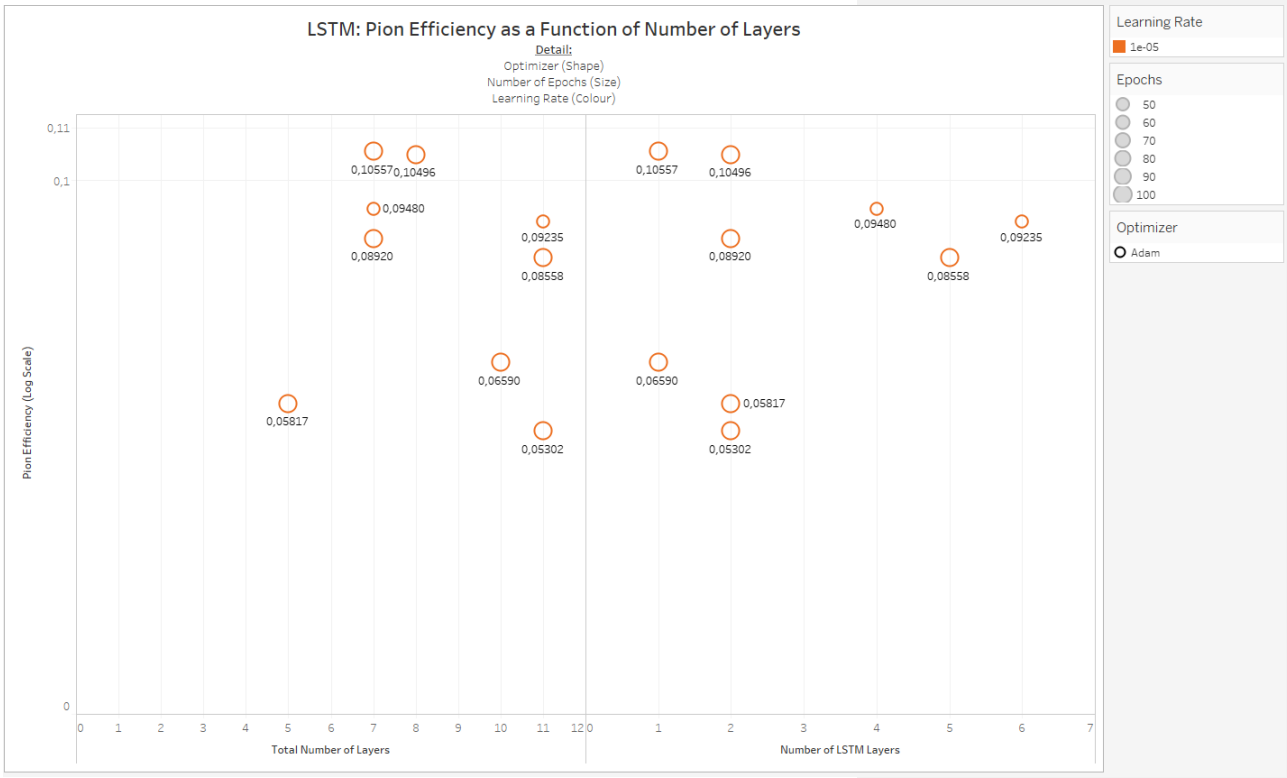

LSTM: Pion Efficiency as a Function of Number of Layers
Detail:
Optimizer (Shape)
Number of Epochs (Size)
Learning Rate (Colour)
Pion Efficiency (Log Scale)
0,11
0,1
0
0,10557
0,10496
0,09480
0,09235
0,08920
0,08558
0,06590
0,05817
0,05302
Total Number of Layers
Number of LSTM Layers
Learning Rate
1e-05
Epochs
50
60
70
80
90
100
Optimizer
Adam

Figure 41

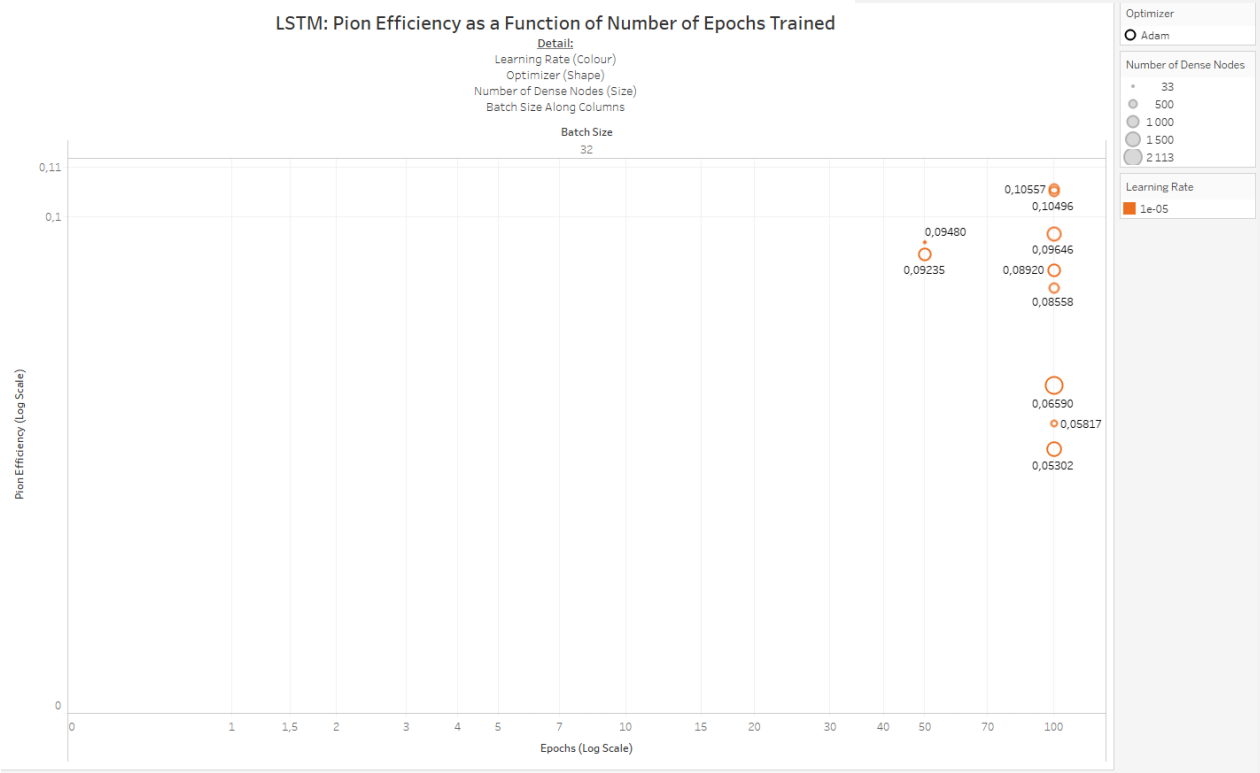

LSTM: Pion Efficiency as a Function of Number of Epochs Trained
Detail:
Learning Rate (Colour)
Optimizer (Shape)
Number of Dense Nodes (Size)
Batch Size Along Columns
Batch Size
32
Pion Efficiency (Log Scale)
0,11
0,1
0
0,10557
0,10496
0,09480
0,09646
0,09235
0,08920
0,08558
0,06590
0,05817
0,05302
0
1
1,5
2
3
4
5
7
10
15
20
30
40
50
70
100
Epochs (Log Scale)
Optimizer
Adam
Number of Dense Nodes
33
500
1 000
1 500
2 113
Learning Rate
1e-05

Figure 42

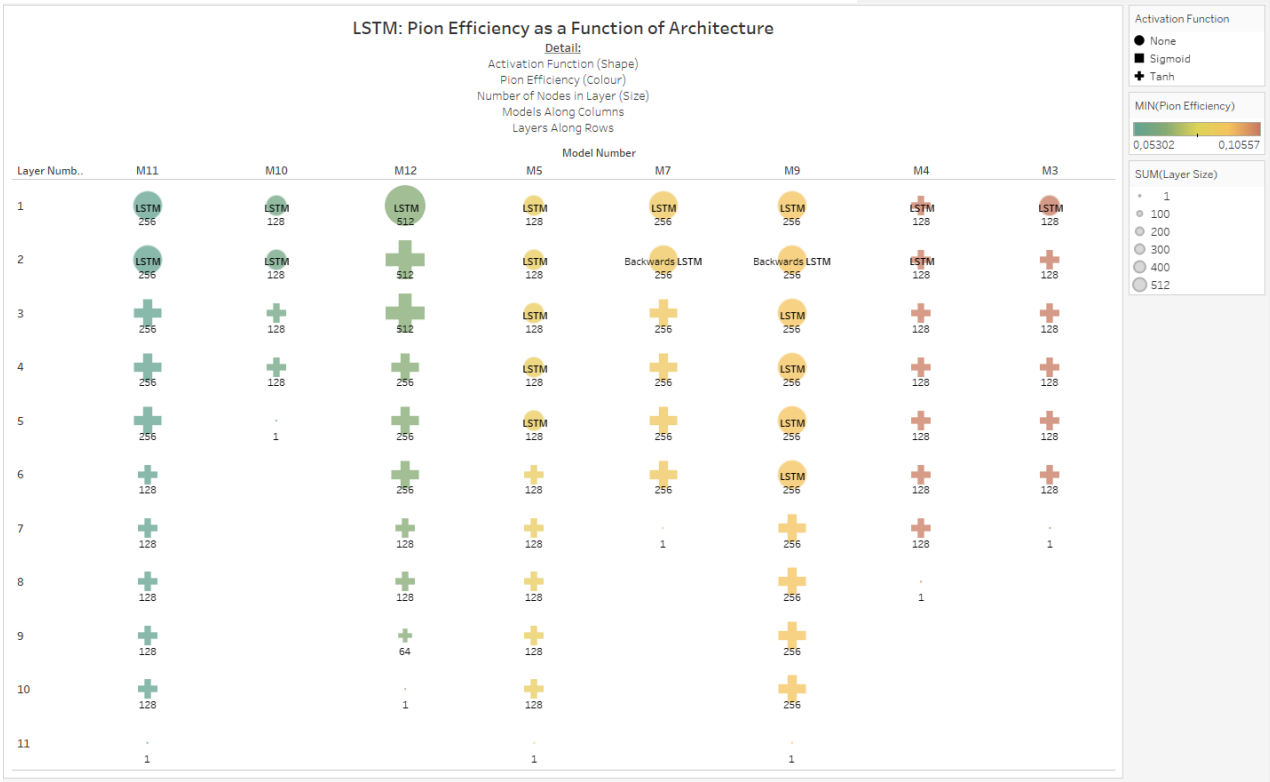

LSTM: Pion Efficiency as a Function of Architecture
Detail:
Activation Function (Shape)
Pion Efficiency (Colour)
Number of Nodes in Layer (Size)
Models Along Columns
Layers Along Rows
Model Number
Layer Numb..
M11
M10
M12
M5
M7
M9
M4
M3
Activation Function
None
Sigmoid
Tanh
MIN(Pion Efficiency)
0,05302
0,10557
SUM(Layer Size)
1
100
200
300
400
512

Figure 43

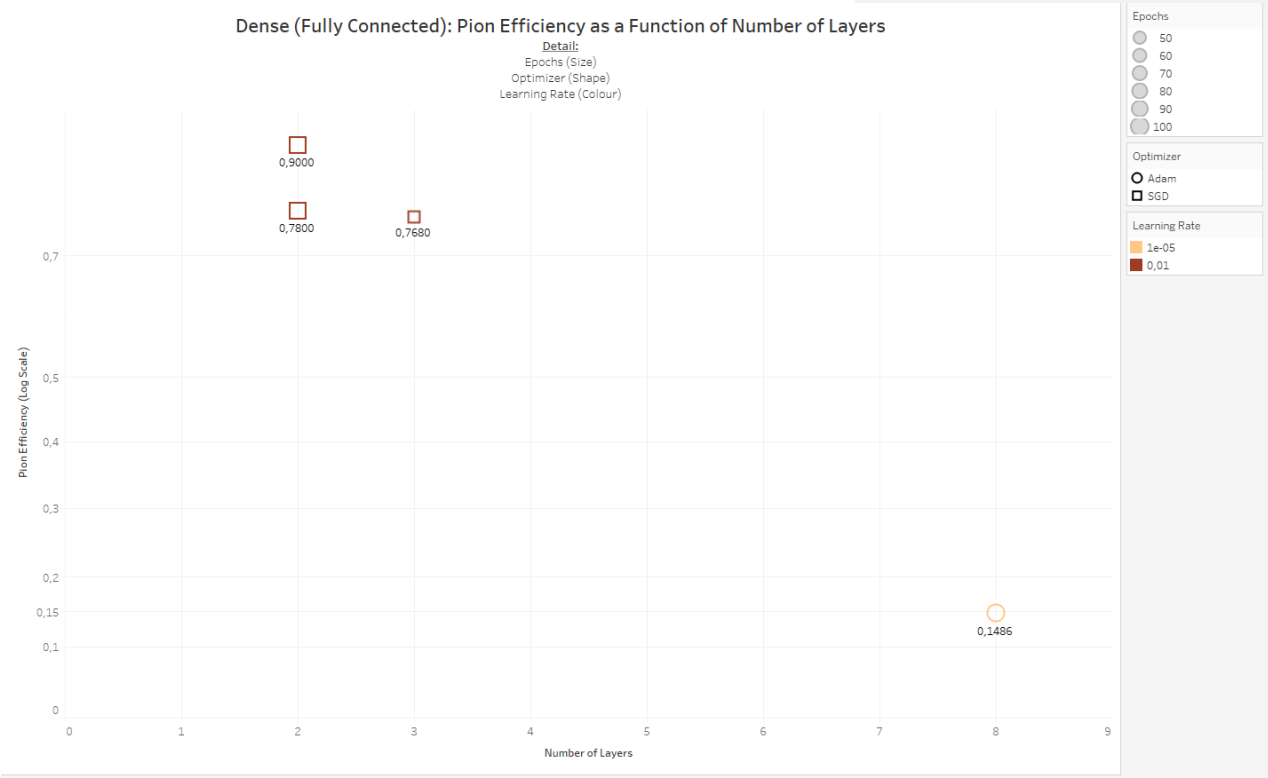

Dense (Fully Connected): Pion Efficiency as a Function of Number of Layers
Detail:
Epochs (Size)
Optimizer (Shape)
Learning Rate (Colour)
0,9000
0,7800
0,7680
0,1486
Pion Efficiency (Log Scale)
0,7
0,5
0,4
0,3
0,2
0,15
0,1
0
0
1
2
3
4
5
6
7
8
9
Number of Layers
Epochs
50
60
70
80
90
100
Optimizer
Adam
SGD
Learning Rate
1e-05
0,01

Figure 44

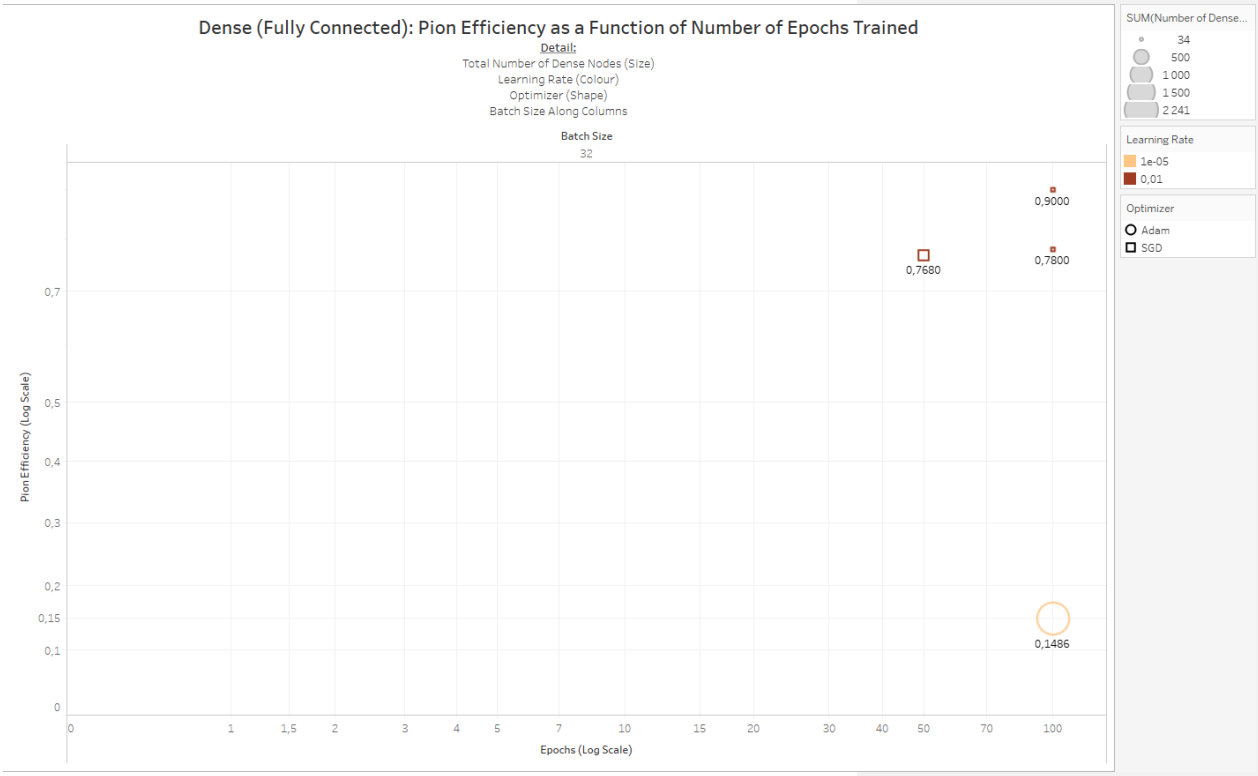

Dense (Fully Connected): Pion Efficiency as a Function of Number of Epochs Trained
Detail:
Total Number of Dense Nodes (Size)
Learning Rate (Colour)
Optimizer (Shape)
Batch Size Along Columns
Batch Size
32
Pion Efficiency (Log Scale)
0,7
0,5
0,4
0,3
0,2
0,15
0,1
0
0,9000
0,7680
0,7800
0,1486
0
1
1,5
2
3
4
5
7
10
15
20
30
40
50
70
100
Epochs (Log Scale)
SUM(Number of Dense...
34
500
1 000
1 500
2 241
Learning Rate
1e-05
0,01
Optimizer
Adam
SGD

Figure 45


Dense (Fully Connected): Pion Efficiency as a Function of Architecture
Detail:
Number of Dense Nodes (Size)
Activation Function (Shape)
Pion Efficiency (Colour)
Models Along Columns
Layers Along Rows
Model Number
Layer Numb..
M4
M3
M2
M1
1
1 024
128
32
32
2
512
128
2
2
3
256
2
4
128
5
128
6
128
7
64
8
1
SUM(Number of Dense...
1
2
32
64
128
256
512
1 024
Activation Function
ReLU
Sigmoid
SoftMax
MIN(Pion Efficiency)
0,1486
0,9000

As the name suggests, transition radiation occurs when a particle transits across a dielectric boundary, this radiation is often measured in particle detectors to inform track reconstruction. Multiple boundaries are typically required to increase radiation yield, and since highly relativistic particles emit transition radiation that extends into the X-ray domain, the TRD utilizes gases with high proton-number (Z) to absorb this radiation, resulting in a high yield of energy deposition relative to the energy lost via ionization [29].

1.10 A Brief History of Atomic Theory

The earliest model for the atom can be traced back to 400 BCE, when Democritus proposed that the entire universe consisted of fundamental particles, or "Atoms", which cannot be divided any further.
In 1803, Dalton refined this model by stating that these indivisible atoms can have distinguishing chemical and physical traits and that they combine to form chemical compounds.
Then, in 1987, JJ Thompson discovered the electron and proposed a – subsequently proven to be incorrect – theory for subatomic structure, in which negatively charged electrons were embedded amongst positive charges within an atom.
Rutherford, Marsder and Geiger disproved this model in 1911, with their seminal alpha-particle scattering experiment and put forth a more accurate model for the atom, in which most of the atom consists of empty space, with a dense core of positively charged protons surrounded by an electron shell.
In 1913, Bohr refined this model further, indicating that electrons orbit the positively charged atomic core at distinct energy levels. While this model did explain the emission spectrum of Hydrogen, it could not explain the emission spectra of any of the other elements.
Between 1924 – 1928, De Broglie, Heisenberg and Schrödinger each separately developed a similar quantum paradigm, where electrons have wave-like properties and appear in much more complex orbitals. This is still the accepted theory of atomic structure today.
There have been some additions made to the quantum theory, as new information has come to light: a neutral subatomic particle, the neutron, was discovered in 1932, which solved the puzzle of why atoms were found to be nearly twice as heavy as expected based on proton number; this discovery also disproved Dalton's second law, which stated that all atoms of a specific element were identical; which resulted in the concept of isotopes (atoms with the same number of protons, but differing numbers of neutrons) being proposed. In the same year, Cockroft and Walton split the atom for the first time, by bombarding Lithium atoms with protons, splitting them into two Helium particles.
The 1950s brought about a new era in nuclear physics, in which particle accelerators with collision energies of a few hundreds of MeVs became affordable, along with cosmic ray and inelastic proton-scattering experiments; since this time, a whole host of subatomic elements have been discovered, many of which are unstable. The discovery of these new particles has led, over time, to the development and refinement of the modern Standard Model of Particle Physics.

(an electron Volt, eV, is a unit of energy, equivalent to the amount of work required to accelerate a single electron through a potential difference of 1 Volt),

1.11 LHC Runs Used

- 000265377
- 000265378
- 000265309
- 000265332
- 000265334
- 000265335
- 000265336
- 000265338
- 000265339
- 000265342
- 000265343
- 000265344
- 000265381
- 000265383
- 000265385
- 000265388
- 000265419
- 000265420
- 000265425
- 000265426
- 000265499

ana.C (https://github.com/PsycheShaman/trdML-gerhard/blob/master/ana.C), modified from a version developed by other collaborators in the SA-ALICE group,

This script interfaces with https://github.com/PsycheShaman/trdML-gerhard/blob/master/AliTRDdigitsExtract.cxx, also modified from a previously developed C++ file.

, i.e.

- The run-number and event number which the track was obtained from
- The V_0 Track ID, identifying the primary vertex from which the track originated
- A track number, which is a unique identifier for the track in that event and run number
- A PDG code, which is 11 for electrons -11 for positrons, 211 and -211 for positively and negatively charged pions, respectively
- n-σ electron and n-σ pion which are the number of standard deviations away from the expected electron- and pion signal, respectively
- The transverse momentum of the particle
- Information about the angle at which the particle is traveling, i.e. Eta, Theta and Phi
- dEdX estimate from the TPC
- The detector number, row and column the particle went through in layers 1-6 (indexed from 0-5 because of Python's indexing strategy)
- The raw data signal caused by the track as it traversed the six layers of the TRD

1.11.1 Alternative ways of visualizing tracklet signals

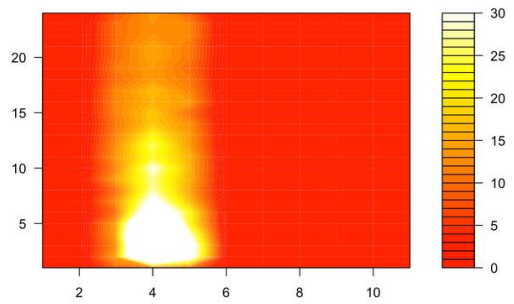


Figure 46: Filled contour map of a single pion tracklet's signal, with pads along the *x* axis (columns) and timebins along the *y* axis (rows)

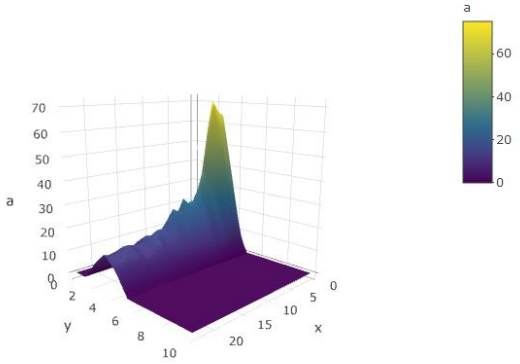


Figure 47: 3D surface plot of the signal of a single pion tracklet's signal, with timebins along *x*, pads along *y* and pulse height along *a*.

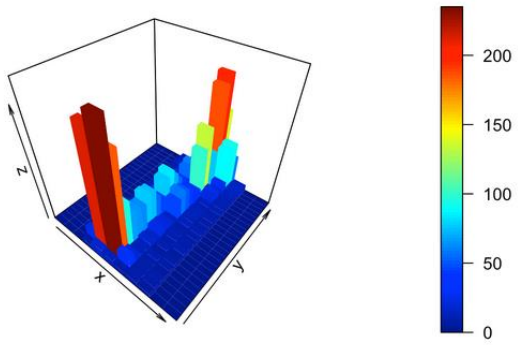


Figure 48: 3D histogram of a single electron tracklet's signal, with pads along *x*, timebins along *y* and pulse height along *z*.

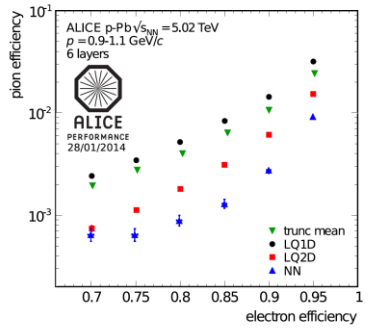


Figure 49: Pion efficiency as a function of electron efficiency for the various particle identification methods discussed [30].

In order to distinguish muons originating from particle-particle collisions from muons originating from cosmic rays, cosmic runs were performed at ALICE; this data, along with energy loss and transition radiation measurements from test-beams at the proton synchrotron (CERN PS) in 2004, and proton-proton (pp-) collisions performed at $\sqrt{s}$ = 7 TeV at ALICE, provides reference distribution data used for particle identification in the ALICE TRD (see Figure 50 for a plot showing the most probable signal dependence on βγ [30]).

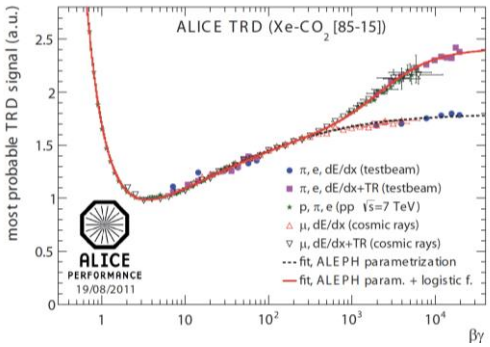


Figure 50: Reference distributions for most probable signal dependence on βγ in the TRD, i.e. from measurements taken in pp-runs, test beams and measurements from cosmic rays [30].

A commonly used nonlinear transformation ϕ, or activation function, in modern deep learning algorithms is the rectified linear unit (the ReLU function), which is simply an affine transformation, of the form $\phi(f_a(x)) = max\{0, f_a(x)\}$ [33].

1.11.1.1.1 *Batch and Minibatch*

The process of minimizing the objective function **J**, can be made more efficient by sampling a "minibatch" of training examples at each iteration, this process also compensates for

redundancy in the training data, where many observations are effectively contributing the same information regarding the gradient of the loss function [33].
Using smaller batches can also prevent overfitting; this regularizing feature is optimal when using a batch size of one with a very small learning rate to maintain stability because the gradient will have high variance in this case, but the combination of a slow learning rate and high number of iterations for each epoch can result in very long compute time during training [33].

1.11.1.1.2 *Stochastic Gradient Descent*

Stochastic gradient descent (SGD) is a commonly used optimization algorithm that makes use of the average gradient of the loss function over a minibatch of training examples as an unbiased estimate of the true gradient; along with a learning rate ϵ, which decreases over time to compensate for noise in the gradient introduced by stochasticity of the process. The learning rate at iteration i is denoted as ϵ_i. The learning rate is often set to decay linearly until iteration τ, i.e.

$$\epsilon_i = (1 - \alpha)\epsilon_0 + \alpha\epsilon_\tau$$

where $\alpha = \frac{i}{\tau}$. After iteration τ, ϵ is commonly leaved constant [33].

1.11.1.1.3 *Momentum*

Momentum, in the context of deep learning, is a method which results in accelerated learning compared to SGD, by taking an exponentially decaying moving average of past gradients into account when updating weights during backpropagation. A weighting hyperparameter $\alpha \epsilon [0,1)$, determines the rate of decay of previous gradients in determining this so-called momentum [33].

A simplified representation of the SGD algorithm with momentum looks as follows:

Given:

- Learning rate $\boldsymbol{\epsilon}$
- Momentum decay parameter $\boldsymbol{\alpha}$
- Initial velocity **v**
- Initial parameter to be updated $\boldsymbol{\theta}$

While stopping criteria is unmet, DO:

1. Sample minibatch of size m from training data
2. Compute gradient:

$$g \leftarrow \frac{1}{m}\nabla_\theta \sum_i L\left(f\left(x^{(i)};\theta\right), y^{(i)}\right)$$

3. Update velocity:

$$v \leftarrow \alpha v - \epsilon g$$

4. Update parameter:

$$\theta \leftarrow \theta + v$$

Commonly used values of α are 0.5, 0.9 and 0.99, and similarly to the learning rate, α can also be adapted over time [33].

1.11.1.1.4 *Nesterov Momentum*

A momentum variant based on the accelerated gradient method proposed by Nesterov, Nesterov momentum updates parameters according to the following rules:

$$v \leftarrow \alpha v - \epsilon \nabla_\theta \left[\frac{1}{m} L\left(f\left(x^{(i)}; \theta + \alpha v\right); y^{(i)}\right)\right]$$

$$\theta \leftarrow \theta + v$$

Nesterov momentum differs from standard momentum in that the gradient is evaluated after velocity is applied, whereas in standard momentum, the gradient is evaluated first, before velocity is calculated and applied, as can be seen in the following algorithm for Nesterov momentum, compared to that for standard momentum shown above [33].

Given:

- Learning rate $\boldsymbol{\epsilon}$
- Momentum decay parameter $\boldsymbol{\alpha}$
- Initial velocity **v**
- Initial parameter to be updated $\boldsymbol{\theta}$

While stopping criteria is unmet, DO:

1. Sample minibatch of size m from training data
2. Apply interim update:

$$\tilde{\theta} \leftarrow \theta + \alpha v$$

3. Compute interim gradient:

$$g \leftarrow \frac{1}{m} \nabla_{\tilde{\theta}} \sum_i L\left(f\left(x^{(i)}; \tilde{\theta}\right), y^{(i)}\right)$$

4. Update velocity:

$$v \leftarrow \alpha v - \epsilon g$$

5. Update parameter:

$$\theta \leftarrow \theta + v$$

1.11.1.1.5 *Adaptive Learning Rates*

Since the learning rate is a very important hyperparameter and difficult to set; a variety of algorithms have been developed by the deep learning community that dynamically modify the learning rate as training progresses.

1.11.1.1.5.1 AdaGrad

The AdaGrad algorithm adapts each one of a model's parameters by scaling them inversely proportional to the square root of the summed historical square values of their individual gradients. In doing so, AdaGrad ensures that parameters that have a greater influence on the objective function (i.e. those that contribute a larger partial derivative to the objective function) have a rapidly shrinking learning rate α, whereas those with smaller partial derivatives of the objective function decrease their learning rate much more slowly [33].

1.11.1.1.5.2 RMSProp

RMSProp is an adaptation of AdaGrad that uses an exponentially weighted moving average, therefore not taking into account all historical gradients but in essence forgetting gradients that fall outside of a specified length scale, ρ [33].

Optimization

The essential optimization objective in deep learning is to find the optimal set of hyperparameters θ to minimize the objective function $J(\theta)$ [33].
Adaptive learning rates, utilization of the second derivative of the loss function during training and various parameter initialization- and other advanced strategies can be employed to make the training/ optimization process more effective [33].

Adam

Originating as an acronym for "adaptive moments", the Adam algorithm is generally touted as an optimization strategy robust to various settings of hyperparameters. Adam combines features of momentum and RMSProp, by using momentum to estimate the first moment of the gradient and by applying bias corrections to both the first and second order moments of the gradient [33].

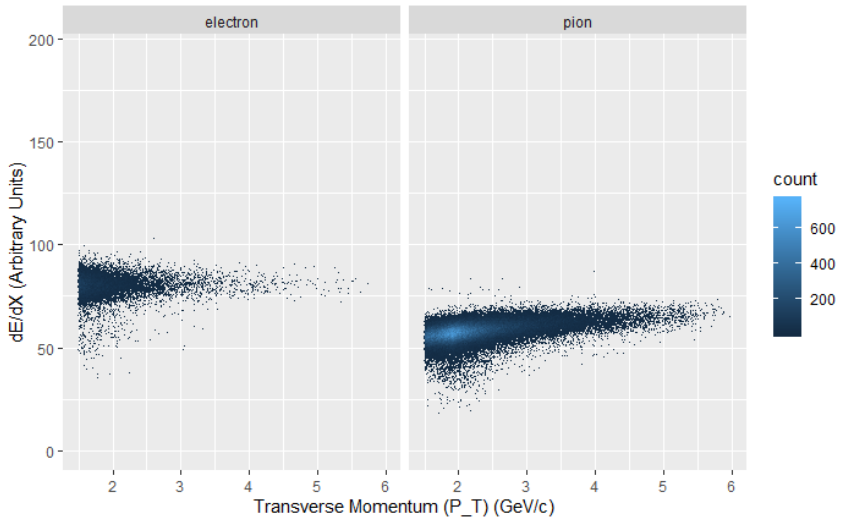


Figure 51: Don't know if I should rather keep this graph, it looks cleaner, there's a weird bump in the 2GeV range when looking at P instead of P_T

1.11.2 Exploratory Data Analysis

Some additional EDA follows, to look for interesting features in the dataset.

1.11.2.1.1 *Mean pixel Value per TRD Layer*

Figure 52 shows how the mean pixel value decreases from the innermost to outermost layers, a first indication that calibration would have increased the pion efficiencies obtained in this project. Much more granular calibration tactics were employed in earlier work done in this area, including chamber gain calibration on a run-by-run basis, as well as pad-by-pad calibration. Since the measurement mechanism employed during data-taking is an analog-to-digital transformation using an immense array of extremely sensitive sensors, the signal obtained from different detector layers and different pads, during different environmental conditions across a long time period, should ideally not be treated as coming from the same measuring device, since these devices will each have their own underlying signal distribution, which was not fed to the models trained in this project.

A form of calibration which feeds some of the above factors of variation as one-hot encoded variables to neural networks for both particle identification and deep generative modelling was investigated, but this meta-information alone explodes to upwards of 42GiB when represented in this fashion. The possibility of encoding this meta-information to a lower dimension using Autoencoders was explored, but the computational cost was deemed to be too much effort for potential pay-off, since there are not equal amounts of signals that originated from each combination of detector, layer and pad, for the neural network to learn useful representations from this information without overfitting, especially to spurious patterns from pads which have produced very little data.

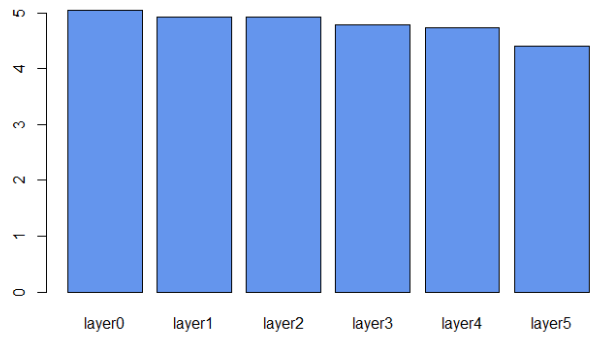


Figure 52: Mean pixel Values per TRD layer, for all runs

1.11.3 The Convolution Function

In practice, data fed to a CNN usually consists of a grid of vectors, effectively adding a depth- (or channel-) dimension to the usual width- and height- dimensions of a grid. In deep learning packages such as Tensorflow, indices of values in such a tensor specify 4 locations, i.e. each value has a row-, column-, channel- and observation index [33].

1.11.3.1 The Convolution Operation

Given an element of a 3-D input tensor $\boldsymbol{V}_{i,j,k}$ i.e. a value in the ith channel, jth row and kth column of a single training observation $\boldsymbol{V}$, which is convolved by a 4-D kernel tensor $\mathbf{K}$ to generate an element in an output tensor specified by $\boldsymbol{Z}_{i,j,k}$, then $\boldsymbol{K}_{i,j,k,l}$ represents the strength of the connection between an element in channel i of the output tensor ($\boldsymbol{Z}_{i,:,:}$) and an element in channel j of the input tensor ($\boldsymbol{V}_{j,:,:}$), offset by k rows and l columns between the input and output element. Thus $\boldsymbol{Z}_{i,j,k}$ is calculated as follows:

$$\boldsymbol{Z}_{i,j,k} = \sum_{l,m,n} \boldsymbol{V}_{l,j+m,k+n-1} \boldsymbol{K}_{i,l,m,n}$$

where the summation over the indices is for all valid indices in the tensor [33].

Pooling with down-sampling is achieved by reducing the number of pooling operations relative to the number of detector units, by introducing a stride greater than one. See Figure 53 for an illustration of the effect of using different stride widths with the same pool-width. It is straightforward to see that down-sampling applied in this manner will lead to fewer inputs to process, reduced memory requirements and improved statistical efficiency [33].

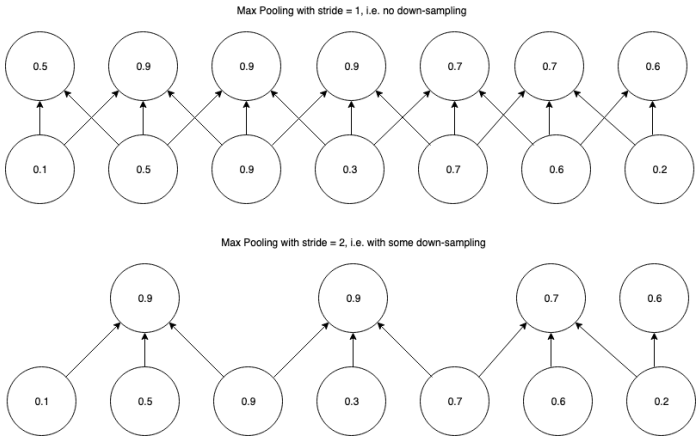


Figure 53: An illustration of the concept of max pooling, using pool-width of 3 with a stride of one (top panel) vs a stride of two (bottom panel) [36].

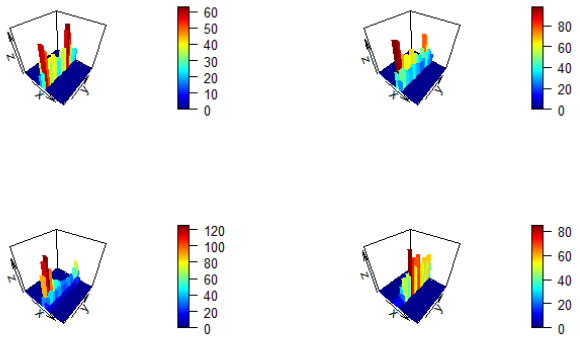


Figure 54: 3D Histograms of Four Randomly Sampled Geant4 Pion Tracklets

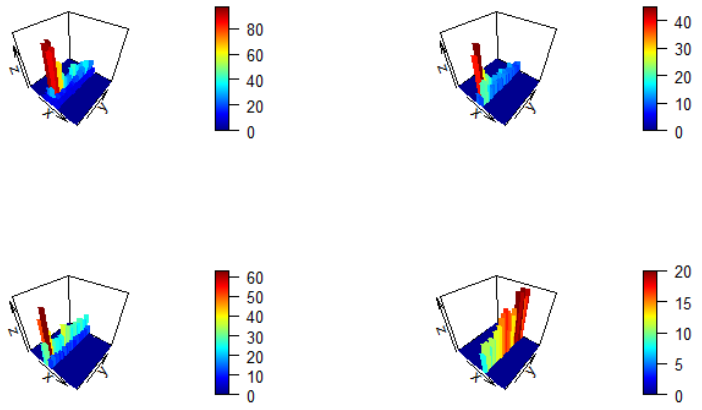


Figure 55: 3D Histograms of Four Randomly Sampled Real Pion Tracklets

Towards developing a prototype for event simulation, the following models were built:

- Autoencoders
- Variational Autoencoders
- Various types of Generative Adversarial Networks

The following repositories contain code used to build and run Deep Generative Models for this project:

https://github.com/PsycheShaman/deep-gen

https://github.com/PsycheShaman/Keras-GAN (forked from https://github.com/eriklindernoren/Keras-GAN and adapted to be able to work with data from this project).

Below is a quick summary of the various Deep Generative Models used under the broader GAN category; which also shows how adjusting certain parameters lead to different results, illustrated by the accompanying images.

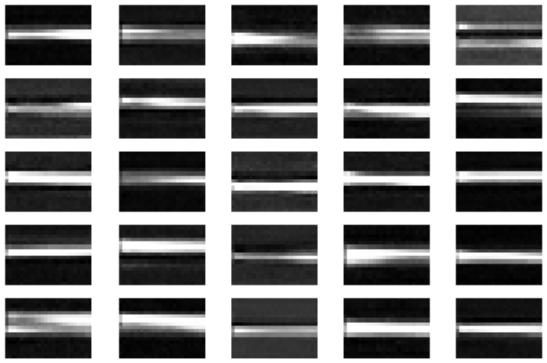
1.11.3.2 Adversarial Autoencoder

1.11.3.2.1 *Version 1*

10 Latent dimensions
Adam optimizer with learning rate = 0.002 and beta1 = 0.5
Batch size = 32

Encoder with 2 hidden layers with 512 nodes each, using leaky ReLU activation
Decoder with 2 hidden layers with 512 nodes each, using leaky ReLU activation and an output layer with tanh activation
Discriminator with two hidden layers, with 512 and 256 nodes respectively and sigmoid activation in the single-node output layer

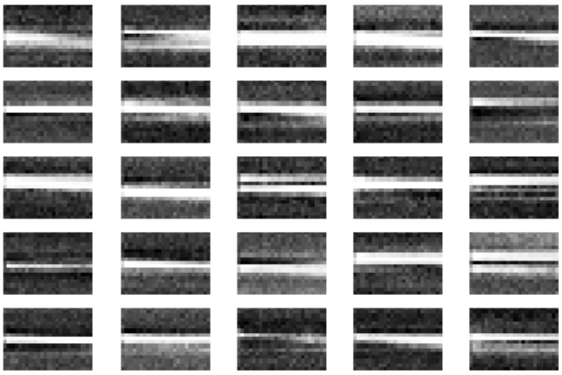
Sample result after 19800 epochs:



1.11.3.2.2 *Version 2*

100 Latent dimensions
Adam optimizer with learning rate = 0.00002 and beta1 = 0.5
Batch size = 32
Encoder with 3 hidden layers with 512 nodes each, using leaky ReLU activation
Decoder with 3 hidden layers with 512 nodes each, using leaky ReLU activation and an output layer with tanh activation
Discriminator with two hidden layers, with 512 and 256 nodes respectively and sigmoid activation in the single-node output layer

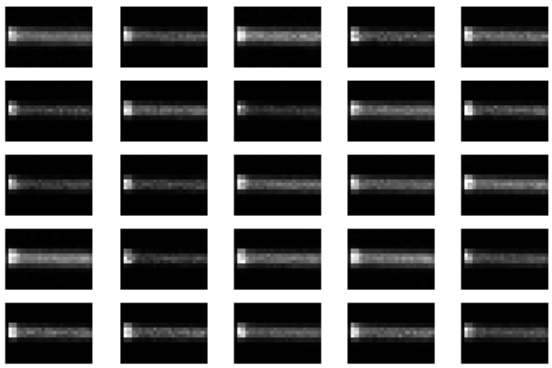
Sample result after 30800 epochs:



1.11.3.2.3 *Version 3*

8 Latent dimensions
Adam optimizer with learning rate = 0.000002 and beta1 = 0.5
Batch size = 32
Encoder with 7 hidden layers with 1024, 512, 256, 128, 64, 32 and 16 nodes respectively, using leaky ReLU activation
Decoder with 4 hidden layers with 128, 256, 512 and 1024 nodes respectively, using leaky ReLU activation and an output layer with tanh activation
Discriminator with 4 hidden layers, with 1024, 512, 256 and 128 nodes respectively and sigmoid activation in the single-node output layer

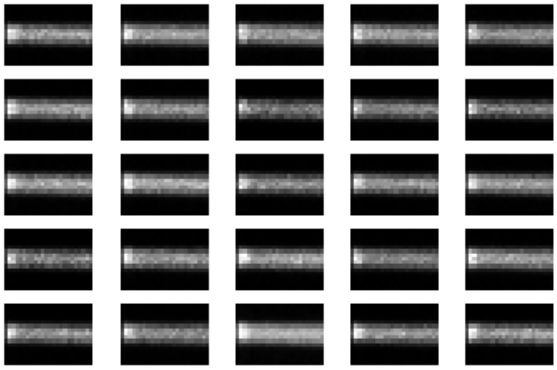
Sample result after 23600 epochs:



1.11.3.2.4 *Version 4*

12 Latent dimensions
Adam optimizer with learning rate = 0.000002 and beta1 = 0.5
Batch size = 32
Encoder with 5 hidden layers with 1024, 512, 512, 128 and 128 nodes respectively, using leaky ReLU activation
Decoder with 4 hidden layers with 256, 256, 512 and 1024 nodes respectively, using leaky ReLU activation and an output layer with tanh activation
Discriminator with 8 hidden layers, with 512 nodes each and sigmoid activation in the single-node output layer

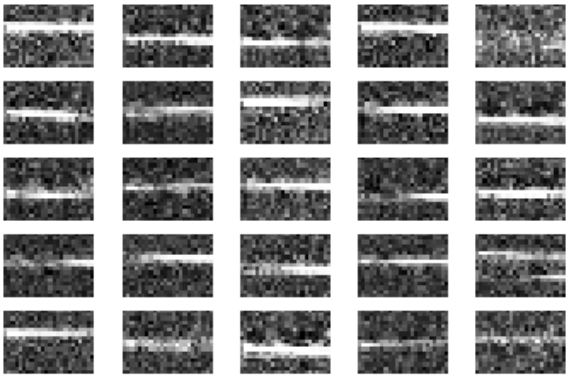
Sample result after 73200 epochs:



1.11.3.3 Bidirectional Generative Adversarial Network

100 Latent dimensions
Encoder with 2 hidden layers with 512 units each, using Leaky ReLU activation and employing Batch Normalization with momentum = 0.8 after each
Generator with the same architecture as the encoder, with an additional dense layer of the same dimensions as the number of pixels in an image, with tanh activation
Discriminator with 3 hidden layers with 1024 nodes each using leaky relu activation and a single node output layer with sigmoid activation function
Trained using Adam Optimizer with Learning Rate = 0.0002 and Beta1=0.9 and a batch size of 32

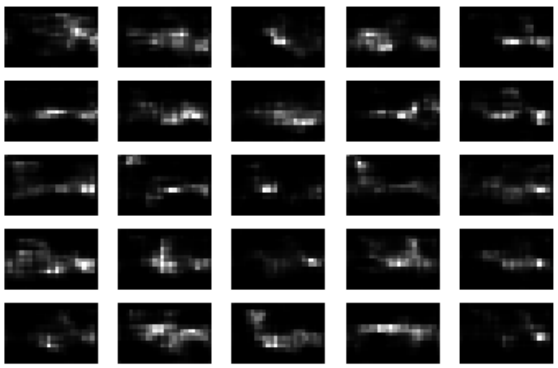
Example output after 39600 epochs:



Deep Convolutional Generative Adversarial Network

100 Latent dimensions
Generator with Dense layer with 3072 nodes, reshaped to $4 \times 6 \times 128$ *with 2D Upsampling, followed by 3 2D convolutional layers, with the first two followed by Batch normalization and 2D Upsampling, using ReLU activation in the first 4 layers and tanh in the final output layer*
Discriminator with 4 2D convolutional layers, with Batch Normalization applied after the second, third and fourth layer and zero-padding after the second layer, using leaky ReLU activation in the hidden layer and sigmoid in the output node.

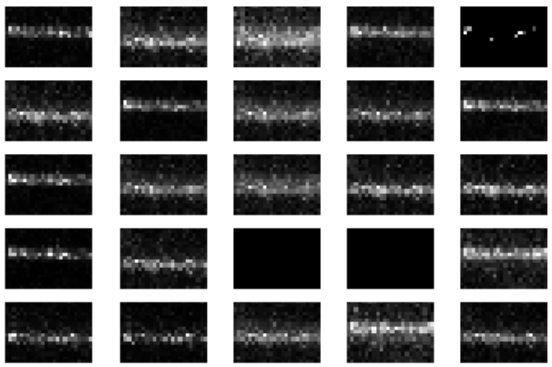
Example output after 57500 epochs:



Generative Adversarial Network

100 Latent dimensions
Two separate Adam Optimizers used for Generator and Discriminator, i.e. with learning rate = 0.00002 and beta1=0.5 for Discriminator and learning rate = 0.00001 and beta1=0.5 for Generator
Generator with 12 hidden layer with 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000, 1100,1200 hidden units, using leaky ReLU activation and a dense output layer of the desired output image dimensions with tanh activation, including Batch Normalization with momentum = 0.8 after each hidden layer
Discriminator with 7 hidden layers with 1024, 1024, 512, 512, 256, 256 and 128 units, respectively, using leaky ReLU activation and a single node output layer using sigmoid activation.

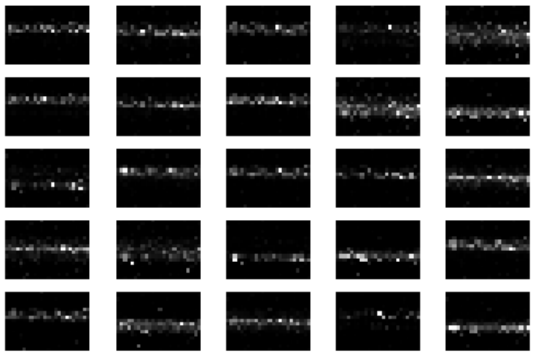
Example Output after 66400 epochs:



Least Squares Generative Adversarial Network

100 Latent dimensions
Adam Optimizer with learning rate = 0.00002 and beta1 = 0.5 using batch size = 32
Generator with 6 hidden dense layers with 256, 256, 512, 512, 1024 and 1024 hidden layers using leaky ReLU activation and an output layer with tanh activation, using Batch Normalization with momentum = 0.8 after each hidden layer
Discriminator with 8 hidden layers with 1024, 1024, 512, 512, 256, 256, 128 and 128 hidden layer and a single node output layer with no activation function

Example output after 64600 epochs:



1.11.4 GANs Stage 2

1.11.4.1 GAN “Hacks”

Various deep neural network architectures were developed towards particle identification, using Keras with a Tensorflow back-end.

The following repositories host the code used to build and train feedforward-, convolutional- and LSTM neural networks towards particle identification:

https://github.com/PsycheShaman/msc-hpc

https://github.com/PsycheShaman/hpc-mini

https://github.com/PsycheShaman/MSc-thesis

A summary of the input tensors used for the various types of deep learning architectures mentioned above is as follows:

- Feed-forward neural networks were fed with summarised data, i.e. the original $n \times 17 \times 24$ matrices were collapsed down to 17 row-sums and 24 column-sums, to arrive at a wide input matrix with dimensions $n \times 41$. Some networks were also trained with additional hand-designed features
- 2D convolutional neural networks were fed data with one image channel added to the $n \times 17 \times 24$ matrices, to produce an input tensor of dimensions $n \times 17 \times 24 \times 1$
- 1D convolutional neural networks were fed data with one channel added to the 24 column-sums, to produce an input matrix of $n \times 24 \times 1$
- LSTM neural networks were fed data where the $n \times 17 \times 24$ matrices were transposed to have dimensions $n \times 24 \times 17$, i.e. 24 time-bins with 17 features

During this stage of model building, class imbalances were accounted for by downsampling the pion sample to be equal in size to the electron sample.

Data was normalized as follows:

$$x = \frac{x - max\ (x)}{max\ (x)}$$

The SLURM-managed High-Performance Computing Cluster at UCT was utilized extensively to test the performance of various deep learning architectures, enabling one to train various deep learning models in parallel.

Over the next few pages, an assessment of the usefulness of various model architectures and hyperparameter settings developed during this stage is conducted at the hand of plots.

1.11.5 2D Convolutional Neural Networks

All 2D Convolutional Neural Networks developed for Particle Identification are summarised in **Error! Reference source not found.**, Figure 83, Figure 84, Figure 85 and Figure 86.

Although it is quite difficult to make any absolute statements about which model architecture is best suited to this problem, there are some conclusions that can be drawn from the plots below.

It is immediately apparent from **Error! Reference source not found.** that the worst-performing models were all trained with a fairly high learning rate. These models were also trained with different optimizers compared to better-performing models, but this is confounded with the fact that the Adam optimizer started being used at the same time that the learning rate was reduced. Using the correct learning rate is essential, since using a very high learning rate will result in the gradient descent algorithm jumping over possible minima, whereas a learning rate which is too low will struggle to converge and might get stuck in local minima.

50-100 epochs seem to be enough iterations to train decently performing models, but the more important aspect to monitor is whether training loss and validation loss start to diverge, which indicates that the model is starting to overfit to the training set. It is hard to monitor this aspect when training on a server, but this can be achieved by piping the output of the Python script used for training to a text file, setting the Keras training verbosity to level 2 and monitoring the model's progress by calling the `tail -F out.txt` bash command. As a general rule of thumb, training models on this dataset for more than 100 epochs is not useful and will most likely result in overfitting. In terms of computational cost, when running Convolutional Networks on the full dataset for 100 epochs, training time can take up to 2-3 days.

From **Error! Reference source not found.**, one can see that is is difficult to establish a clear guideline as to the number of layers (convolutional- or dense), which will result in a low pion efficiency at high electron efficiency. Models with a total of 8-9 layers seem to perform well in general and models with 6 or less layers slightly worse, although the model which gave the second-best performance only had a total of 4 layers, of which two were convolutional.

Using larger kernel sizes seems to be a good strategy, judging by the top two best-performing models, who both had 8x12 and 5x6 kernels in the first two layers of the network. The best-performing model was one of only two models that had 4 convolutional layers, the other model with this much convolutional depth being in the top 4 highest-performing models as well.

There is no consistent indication of whether using Max Pooling improved performance in this task, but the top two models did not make use of them.

Figure 84 shows that most 2D Convolutional models were trained with a batch size of 32, while using a smaller batch size makes gradient updates slightly more volatile and therefore a little less accurate, using very large batch sizes makes this already computationally expensive procedure take even longer.

While a lot of time was spent on building different architectures, comparatively little time was spent on optimizing hyperparameters, such as learning rate, batch size, dropout rate, weight initialization strategies and testing different standard deviation settings with Gaussian Noise layers.

In retrospect, spending some time on setting up Randomized Grid Searches could have been useful, but this brings us to the point of why pion rejection results in this thesis are not comparable to previous work done on Particle Identification.

The data used in this project was not corrected for chamber gain, nor was pad-by-pad calibration performed on the raw dataset. These two essential steps explain the hard limit on accuracy/ pion efficiency, which seems to be at around 75% accuracy an 1-2% pion efficiency; therefore spending even more time on this phase of the project would be futile, since results on raw data will never compare with results from models trained on properly calibrated data.
A surprising result from Figure 85 is that Dropout was not employed in the model which achieved the lowest pion efficiency score. It is difficult to explain why this is the case, since Dropout should generally enable a model to achieve higher accuracy on unseen data. However, by looking at the training and validation accuracy and loss curves in Figure 80 and Figure 81, one sees that this model did indeed not overfit, despite it not being regularised. Perhaps the large capacity of this network compensated for the lack of regularisation, or perhaps this model just happened to pick up features that generalised well on unseen data by random statistical fluctuations.

1.11.6 Particle Identification Using Deep Learning

1.11.6.1 Accuracy Paradox

As mentioned, when extremely large class imbalances are not accounted for, misleadingly good results may appear to occur during training (see Figure 56), but this level of accuracy just reflects the ratio of the classes in the dataset we're working with (the model learns that it gets the lowest loss when it favours the prediction of pion, since there are so many more pions in our dataset, compared to electrons). This phenomenon is known as the "Accuracy Paradox".

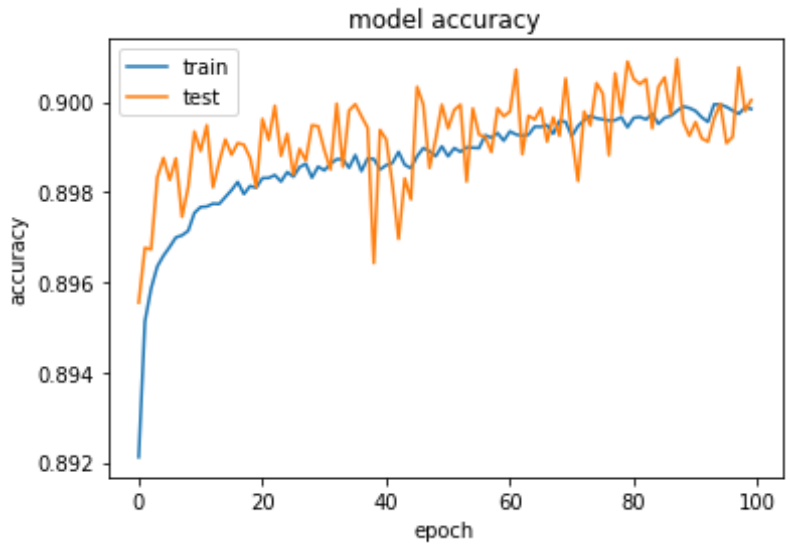


Figure 56: Accuracy Paradox

1.11.6.2 Inherent Limit on the Amount of Information Contained about the Class Label

One of the most salient features of the data used in this project is the inherent limit of the amount of information that the input features x contain about the target feature y.

There seems to be an absolute limit at around 75% training accuracy, regardless of:

- which architecture was used (Figure 57, Figure 58 and Figure 59 show how this problem is potentially solvable by 2D Convolutional, LSTM and 1D Convolutional Neural Networks)

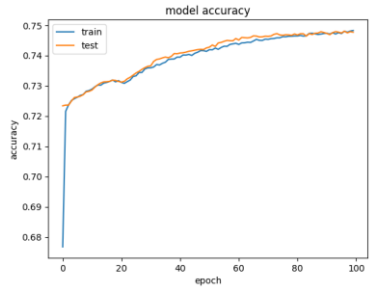


Figure 57: Example of a 2D-Convolutional Network training to high validation accuracy

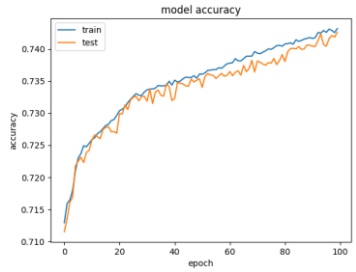


Figure 58: Example of an LSTM Network training to high validation accuracy

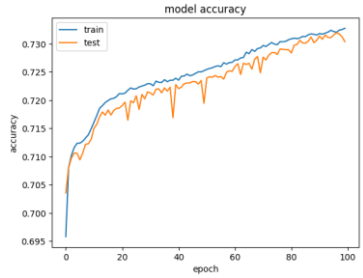


Figure 59: Example of a 1D-Convolutional Neural Network training to high validation accuracy

- the amount of epochs used for training (Figure 60 shows how training a highly successful model for twice the number of epochs only results in eventual overfitting)

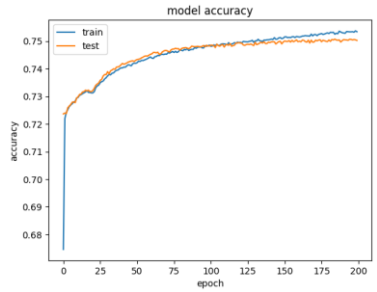


Figure 60: Running a successful model for twice the number of epochs results in minimal gains and eventually, results in overfitting

1.11.6.3 Convolutional Neural Networks

Regardless of the limitations outlined above, 2D convolutional neural networks resulted in the highest performance in general, but interestingly, even a very simplistic convolutional neural network (one convolutional layer, using 16 convolutional filters with kernel size equal to image dimensions, i.e. 17×24 and a single dense layer with 12 nodes) still gave comparable results, i.e. $\varepsilon_\pi = 6.2\%$.

1.11.6.4 1D Convolutional Neural Networks

While 2D Convolutional Neural Networks were more successful than 1D Convolutional Neural Networks, 1D CNNs resulted in similar performance, even though they were fed data with lower dimensions, i.e. a compressed version of the data 2D CNNs were trained on, in the form of column sums of the original 2D image data. It should be noted that many more 2D CNNs were built during this project, but the fact that 1D CNNs gave comparable accuracy provides additional suspicion that there is a limit to the amount of information contained in the 2D images from the TRD about the Particle ID.

1.11.6.5 LSTM Networks

The nature of this dataset is such that it can be framed as an image for Convolutional Neural Networks, but it is essentially a timeseries as well, with columns going across indicating the ADC signal at sequential time intervals and rows indicating where the charge was deposited (in which pad in a specific TRD chamber, row and column).

The lowest pion efficiency using LSTM networks was $\varepsilon_\pi = 8.5\%$, which is much higher than that achieved by the best 2D Convolutional Neural Networks. This was achieved by having 4 LSTM layers return sequences sequentially, followed by a final LSTM network which fed into a dense architecture of 5 fully connected layers of 128 nodes each; although similar performance ($\varepsilon_\pi = 8.9\%$) was obtained using only two LSTM layers, with the second layer going backwards, followed by four dense layers of 256 nodes each. Using 6 LSTM layers,

alternating between going backwards and forwards, with four dense layers of 256 nodes each made pion efficiency drop back down to $\varepsilon_\pi = 9.2\%$.

1.11.6.6 The Effect of Regularization

Figure 61 shows the effect of applying too much Dropout at training time. The left side of the figure shows a model which overfit during training and the right hand side shows the effect of applying a Dropout rate of 0.5 to each layer of the same network.

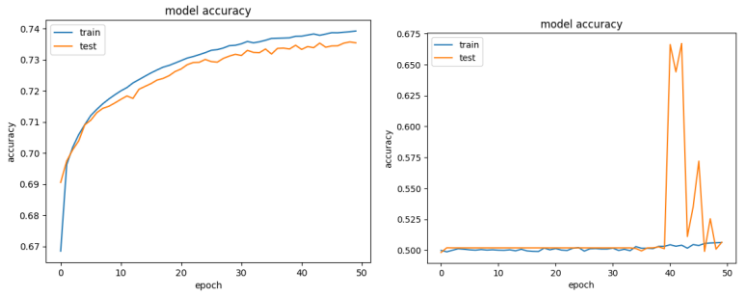


Figure 61: No Dropout vs Same Model with too much Dropout

In general, using a Dropout rate of 0.2 on the fully connected layers of the network proved to be a decent strategy, bearing in mind that other hyperparameters also play a role, e.g. a model employing the abovementioned Dropout rate which resulted in a pion efficiency of $\varepsilon_\pi = 6.5\%$ dropped down to a pion efficiency of $\varepsilon_\pi = 24.1\%$ when the learning rate (using the Adam optimizer) was reduced from $\epsilon = 10^{-4}$ to $\epsilon = 10^{-6}$.

The use of a Gaussian Noise layer as the first layer of various convolutional architectures was employed, but was unsuccessful, as depicted in Figure 62. All models that incorporated Gaussian noise gave pion efficiencies of $\varepsilon_\pi > 45\%$. While there was not much experimentation with different standard deviations of the Gaussian noise layer, this method seems more applicable to image data which is less sparse, since it is only active during training time, and since most of the rows in our input data naturally has a value of 0, adding Gaussian noise will only serve to confuse the model when evaluated at test time.

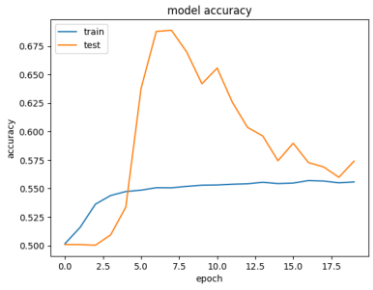


Figure 62: Gaussian Noise with σ=0.2

1.11.7 Deep Generative Models towards High Energy Physics Event Simulations

1.11.7.1 Challenges faced during deep generative modelling

Running on a server is not practical, since training Deep Generative models requires constant qualitative monitoring, especially when label smoothing is used, monitoring metrics that are informative during classification could be misleading if intermediate output images can't be viewed and training can be forced stop if the Generative model is outputting unrealistic images.

Not being able to run on a server prevents parallel training of models. Running locally imposes additional constraints in the form of RAM and processing limitations. Loading the full dataset results in an out of memory error.

Tweaking a single hyperparameter to see its effect would be ideal, but time constraints force one to often change multiple hyperparameters and architectures simultaneously. Using one's intuition as to what the effect of each of a combination of changes induced becomes more important.

While various deep generative architectures can give results that appear vastly different in terms of "style", they all fail to capture the underlying distribution of the training data sufficiently.

A possible reason for this is that the images used for training are quite small and have relatively little information compared to, for instance, images of human faces, where GANs have proven to be quite successful.

While Variational Autoencoders give images that appear less spread-out along the pad dimension than any of the GAN architectures that were used, with less overall noise, they still fail to capture some of the volatility along the time dimension and appear somewhat "smeared out" compared to real data.

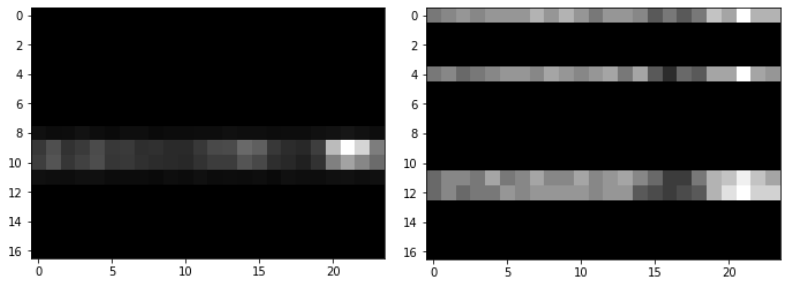


Figure 63: Real images

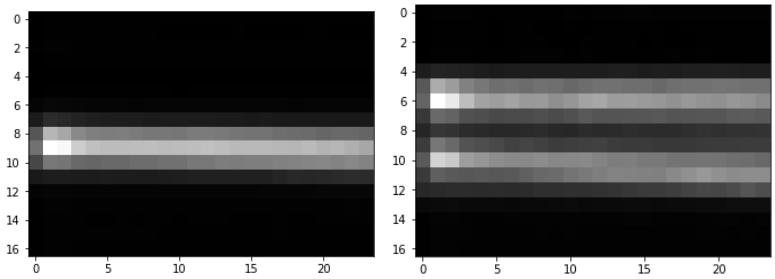


Figure 64: Autoencoder outputs

While Adversarial Autoencoders gave results that were slightly better than the other GAN architectures experimented with, all of the GAN architectures provided disappointing results, possibly due to the fact that GANs are quite hard to train, since there are two neural networks pitted against each other in such a setup and if either of them become dominant during the training process, they can force the other into a parameter space which is not conducive to training either network properly.

Examples of this can be seen in an Adversarial Autoencoder setup with very deep architecture in both the Generator and Discriminator (Figure 65), where after 21400 epochs, the Generator still outputs images that are seemingly random, but nonetheless have similar features across images (for some reason this resulted in low error and therefore became prevalent). This phenomenon is known as Mode Collapse.

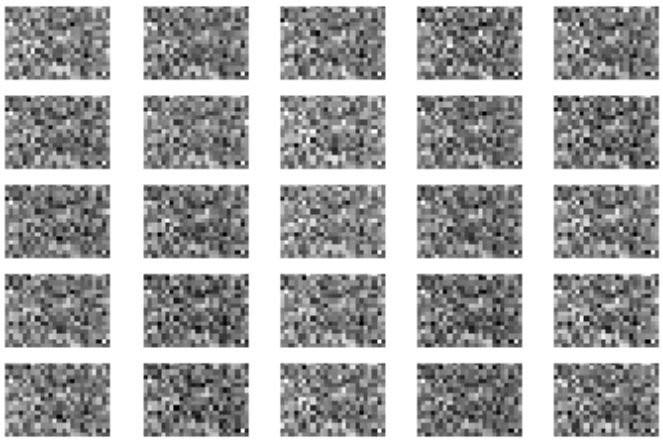


Figure 65: An illustration of Mode Collapse

Batch Normalization is a strategy that ensures that a layer's outputs are normally distributed and therefore prevents the Generative component of the GAN to output images that look very similar, in Figure 66, one can see that the output images of the Bidirectional GAN (which employs this strategy) are highly dissimilar, but there is still a lot of noise around the main signal.

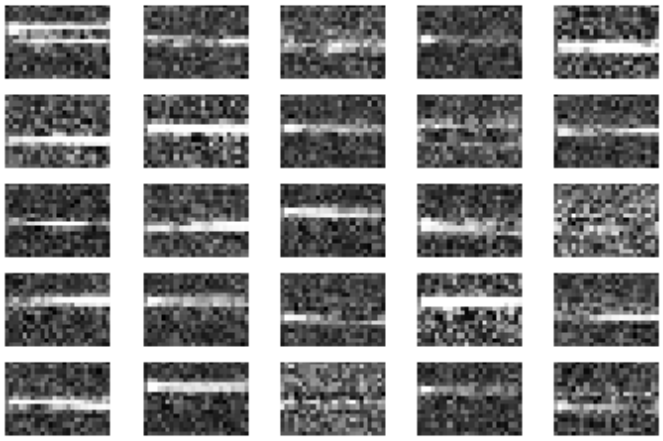


Figure 66: Illustrating the effect of Batch normalization to prevent output images from looking highly similar

It is interesting to note that while one might expect GANs that employ convolutional layers to give better results than fully connected dense layers, the Deep Convolutional GANs gave some of the most unrealistic simulated images (Figure 67). This could be due to the fact that convolutional layers introduce a prior that features are translationally invariant, i.e. they can appear anywhere in the image. While this might be a useful assumption during particle identification, it does not seem to hold for when simulating the data used in this project.

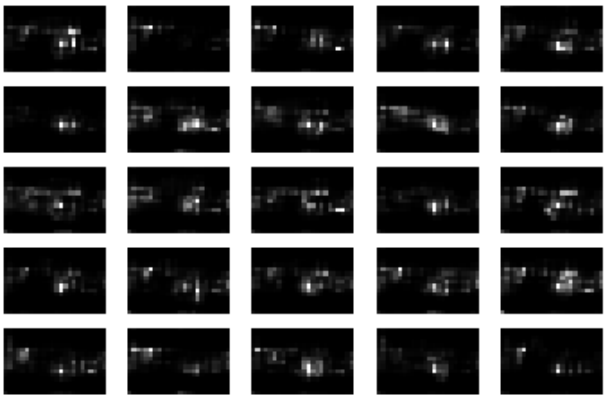


Figure 67: Illustrating how convolutional architectures in a GAN setup results in features that might exist in the training distribution, but do not appear in the right place

Training after a certain number of epochs does not necessarily result in additional gains in performance, as can be seen in Figure 68, Figure 69 and Figure 70, there is not much change in the output of a Least Squares GAN when training for an additional 293 000 epochs after 94 600 epochs and training for an additional 61 400 epochs after that eventually results in images that look less realistic than those produced during earlier epochs.

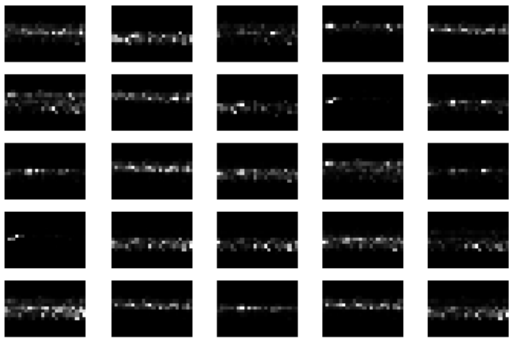


Figure 68: Least squares GAN after 94600 epochs

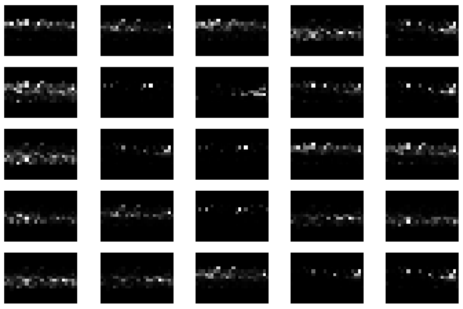


Figure 69: Least squares GAN after 387600 epochs

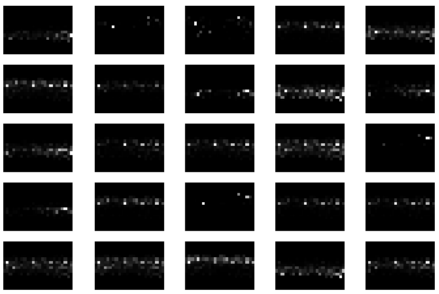


Figure 70: Least squares GAN after 449000 epochs

1.12 Conclusions

1.12.1 Particle Identification

While neural networks are very good at coming up with their own feature sets to find nested functions that can solve classification problems, what's more important in a deep learning project is making sure that the input data contains sufficient information about the class label.

While the work done in this project did not achieve comparable results to work done before, pad-per-pad calibration was performed on data in the work done before, which proved to be invaluable in normalizing the data for environmental and electronic variations which occur during data taking and affect how the signal manifests, a process which cannot be solvable by deep learning techniques.

1.12.2 Simulations

Probably the most important result of this dissertation is the fact that Geant4 simulations are easily distinguishable from real data. Whilst deep generative techniques do not appear to be able to give similar performance compared to Geant4 for this particular problem, there could be interesting ways to combine the two methods, e.g. by transforming the output given by Geant4 with deep generative techniques to make it appear more like real data.

1.13 Outlook and Future Work

1.13.1 Particle Identification

Future work should focus on applying additional data pre-processing steps before particle identification is applied. The data used for particle identification in this thesis was raw data from the TRD, whereas previous work used data which was calibrated pad-by-pad per run; however there does not seem to be much promise for arriving at increased accuracy using advanced deep learning methods compared to work that was done before

1.13.2 Simulations

As mentioned in the Conclusions section above, there should be scope to use the output of Geant4 simulations as a latent space to produce more realistic simulations, alternatively, a more realistic approach would entail tuning parameters used during simulation in the following script https://github.com/alisw/AliRoot/blob/master/TRD/TRDbase/AliTRDSimParam.cxx, a lot of hard-coded parameters could be adjusted recursively by using the loss function of a neural network which is used to distinguish between the two data sets as a parameter to be maximized, i.e. to produce simulated data that becomes harder to distinguish from real data as the multidimensional input parameters maximize the loss function of the discriminating neural network more, a setup similar to what occurs in GANs.

1.13.3 Outlook

Tensorflow and the Keras library are immensely useful for rapid prototyping of various deep learning architectures, whether they be for classification or regression problems. Whilst there exists a toolkit for machine learning within the AliRoot infrastructure, there should be benefits to exploring how using more modern deep learning libraries within the environment of AliRoot could extend its capabilities as Machine Learning becomes a more mature and effective field.

After applying the classification based on t_{cut}, the histograms for the 6-tracklet estimate, for both electrons and pions is shown below:

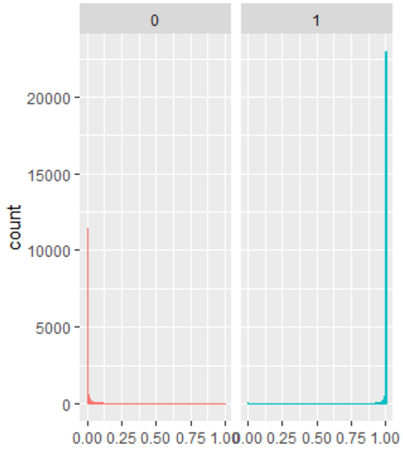


Figure 71: Combined output probabilities for true electrons (1, blue) and true pions (0, red)

for example, binary cross entropy:
$J(\theta) = -(y\,log(p)) - (1 - log(p))$, where p is the model's estimate for the probability of an observation of being of a particular class y [33].
Figure 72 shows how, as $p_{model}(y|x)$ approaches the true y (in this binary classification example, $y = 1$), the binary cross entropy loss function approaches 0.

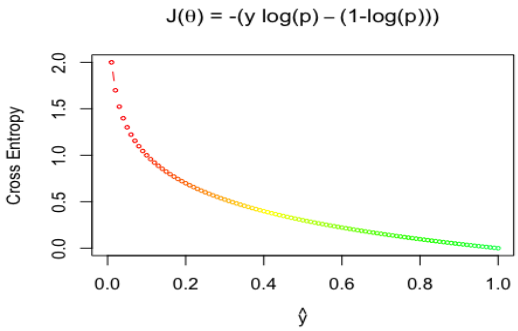


Figure 72: Illustration of the descent towards zero, of the Binary Cross Entropy Loss Function as ŷ, or $p_{model}(y|x)$, approaches the true y.

As can be seen in Figure 73, there is a decreasing number of signals returned per layer, moving outwards from the innermost layers

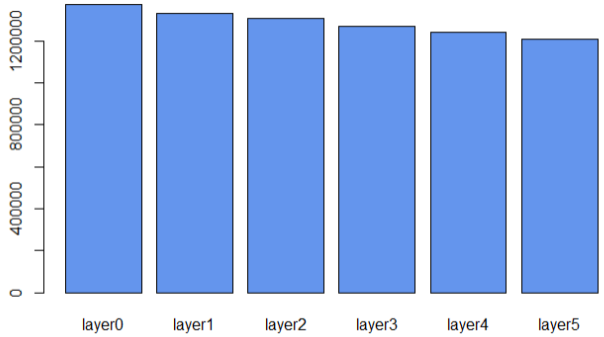


Figure 73: Number of Entries per TRD Layer Number, for all Runs

LIST OF ABBREVIATIONS AND ACRONYMS

ALICE	A Large Ion Collider Experiment
TRD	Transition Radiation Detector
CERN	European Organization for Nuclear Research
QGP	Quark Gluon Plasma
LHC	Large Hadron Collider
WLCG	Worldwide LHC Computing Grid
QCD	Quantum Chromodynamics
ML	Machine Learning
PbPb	Lead-Lead Collisions
e^-	Electron
π	Pion
QED	Quantum Electrodynamics
p	Proton
n	Neutron
v_e	Electron Neutrino
v_μ	Muon Neutrino
v_τ	Tau Neutrino
LSTM	Long Short-Term Memory
VAEs	Variational Autoencoders
GANs	Generative Adversarial Networks
nσ-electron	The TPC's estimate for how many standard deviations away from the expected signal for an electron a particle is
eV	Electron Volt
u	Up quark
d	Down quark
s	Strange quark
c	Charm quark
b	Bottom quark
t	Top Quark
μ^-	Muon
τ^-	Tau lepton
EWT	Electroweak Theory
c	The speed of light
$\langle O \rangle$	Vacuum expectation value
QFT	Quantum Field Theory
g	Coupling strength of standard model interaction vertices
τ	Characteristic mean lifetime of subatomic particles
HEP	High Energy Physics

fm	Femtometer
s	Seconds
T	Temperature
UNESCO	United Nations Educational, Scientific and Cultural Organization
TB	Terabytes
RAM	Random Access Memory
GiB/s	Gigabytes per second
RHIC	Relativistic Heavy Ion Collider
Z	Atomic number
H_2	Hydrogen
$\sqrt{s}$	Centre of Mass Energy
p	Momentum
pPb	Proton-Lead Collisions
Pb	Lead
PS	Proton Synchrotron
SPS	Super Proton Synchrotron
NbTi	Niobium-titanium
K	Degrees Kelvin
°C	Degrees Celcius
T	Tesla
P_{vac}	Vacuum Pressure
atm	Atmosphere
ATLAS	A Toroidal LHC Apparatus
CMS	Compact Muon Solenoid
LHCb	Large Hadron Collider beauty
TOTEM	The TOTal cross section, Elastic scattering and diffraction dissociation Measurement at the Large Hadron Collider
MoEDAL	The Monopole & Exotics Detector at the LHC
OOP	Object Oriented Programming
GNU	Gnu's Not Unix
OS	Operating System
T_c	Critical Temperature
m	Meter
ITS	Inner Tracking System
SPD	Silicon Pixel Detectors
SDD	Silicon Drift Detectors
SSD	Silicon Strip Detectors
dE/dx	Energy loss per unit pathlength
P_T	Transverse Momentum
TPC	Time Projection Chamber
TOF	Time of Flight
m²	Square meters
ps	Picoseconds
HMPID	Ring Imaging Cherenkov Detectors

$Xe - CO_2$	Xenon - Carbon Dioxide Gas
PHOS	Photon Spectrometer
EmCal	Electromagnetic Calorimeter
$PbWO_4$	Lead Tungstate
V0	ALICE V0 Detector
T0	ALICE Fast timing and trigger detector
PMD	Photomultiplicity detector
FMD	Forward multiplicity detector
ZDC	Zero Degree Calorimeters
cm	Centimetre
η	Pseudorapidity
MWPC	Multi-wire Proportional Chamber
ns	Nanoseconds
ADC	Analog to digital converter
γ	Relativistic Factor
LQ1D	One-dimensional likelihood
LQ2D	Two-dimensional likelihood
AI	Artificial Intelligence
MLPs	Multilayer Perceptrons
ReLU	Rectified Linear Unit
J	Objective function
SGD	Stochastic Gradient Descent
ϵ_i	Learning rate at iteration i
$\boldsymbol{\alpha}$	Momentum decay parameter
v	Velocity
$\boldsymbol{\theta}$	Parameter set
g	gradient
$\widetilde{\theta}$	Interim parameter update
ρ	Length scale
Adam	Adaptive Moments
CNN	Convolutional Neural Network
RNN	Recurrent Neural Network
ANN	Artificial Neural Network
$s^{(t)}$	State of a dynamical system at timestep t
Z	Latent space
I	Identity matrix
$\mathcal{D}$	Kullback-Leibler divergence
μ	Mean
Σ	Multidimensional standard deviation
σ	Standard deviation
D	Discriminative Neural Network
G	Generative Neural Network
BiGANs	Bidirectional Generative Adversarial Networks
LSGANs	Least Squares Generative Adversarial Networks
H_0	Null Hypothesis
t	Test statistic

t_{cut}	Threshold value for test statistic
α	Significance level
$1-\beta$	Power
ε_e	Electron Efficiency
ε_π	Pion Efficiency
PDG	Particle Data Group
V_0	Primary vertex
γ_1	Skewness

Again, it needs to be stressed that the poorer performance of particle identification obtained during this project is most probably fully accounted for by the quality of data was used here, in comparison to pad-by-pad calibrated data fed to much simpler neural fully connected neural networks in the previous work done.

An interesting fact about the results obtained during the final stage of model building is the fact that the data it was trained on was not scaled in any way. Perhaps this was also a contributing factor to the low pion efficiencies obtained, possibly because the signal arrays are very sparse, with most pixels being equal to zero, allowing the gradients during backpropagation to flow more strongly when the other pixels are also close to zero, but this is just speculation. For the most part, neural networks remain proverbial black boxes.

List of Tables

List of Figures

Keras allows one to resume training of a model, by increasing the number of epochs in the keras::fit() function to the total number of desired epochs, i.e. number of epochs (previous training round) + additional epochs to be run, and specifying an initial_epochs argument, which should be the final epoch number from the previous training round.

Using this functionality, a 2D Convolutional Network was trained on particles in the $P \leq 2GeV$ range for 10 epochs (note that the class imbalance at this momentum range is less pronounced, containing a ratio of around 76 pions: 24 electrons), Figure 74 shows the accuracy and loss curves obtained.

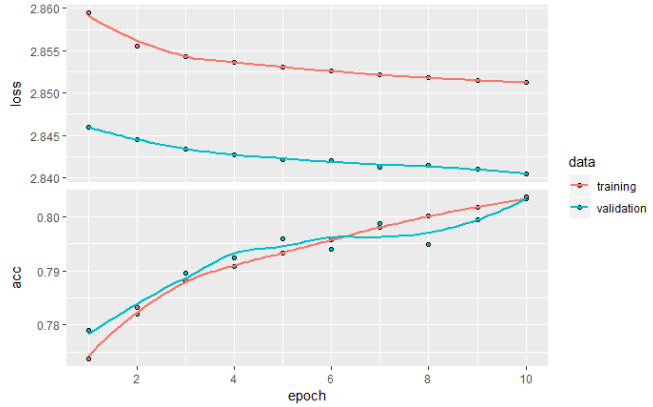


Figure 74

When it became clear that the model was not yet overfitting, but had some statistical power (validation accuracy ~80% compared to what a majority class classifier would give, i.e. ~76%), the model was trained for an additional 20 epochs (Figure 75).

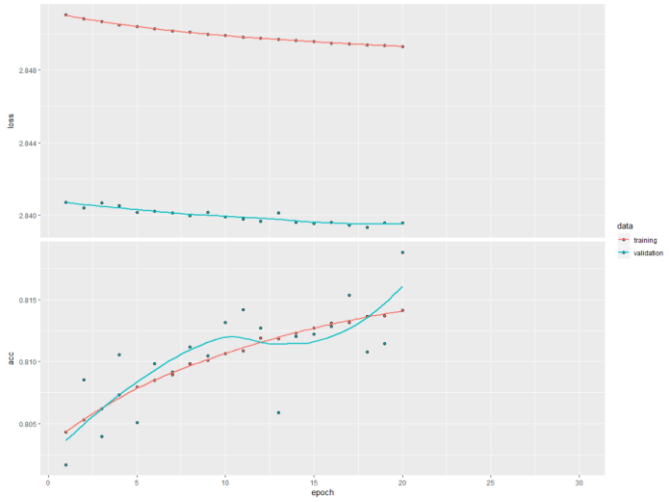


Figure 75

This model performed as follows on an independent test set of particles in the same momentum range that it was trained on:
$\varepsilon_\pi = 1.2\%$ at $\varepsilon_e = 89.99\%$, which is much better than anything obtained during the second stage of model building.
Surprisingly, this model performed almost equally well on particles in the following momentum bin, i.e. performance on particles with momenta $2\ GeV < P \leq 3GeV$ was $\varepsilon_\pi = 1.4\%$ at $\varepsilon_e = 90\%$.

Another feature of Keras is that the state of the optimizer of a saved model can be discarded and only the weights of a previously trained model can be loaded into a new model with the same architecture. Using this functionality training was continued, using the pretrained model, on particles in the $2\ GeV < P \leq 3GeV$ range (Figure 76).

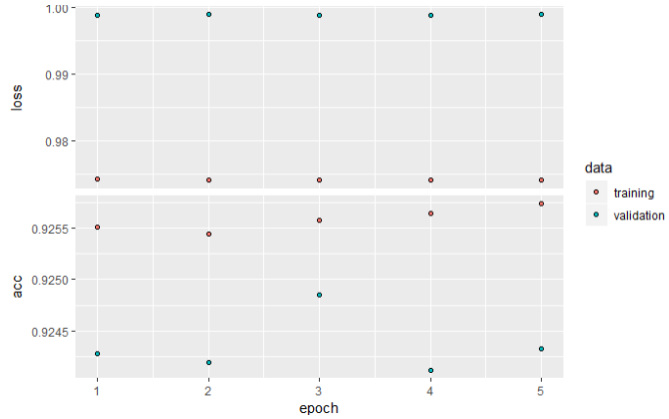


Figure 76

When evaluated on a test set in this momentum range ($2\ GeV < P \leq 3GeV$), performance increased slightly to $\varepsilon_\pi = 1.14\%$ at $\varepsilon_e = 89.99\%$
And when tested on particles in the next momentum range ($3\ GeV < P \leq 4GeV$), performance remained surprisingly high: $\varepsilon_\pi = 1.16\%$ at $\varepsilon_e = 89.86\%$.

This model was then trained for a further 5 epochs on particles in the $3\ GeV < P \leq 4\ GeV$ range (Figure 77)

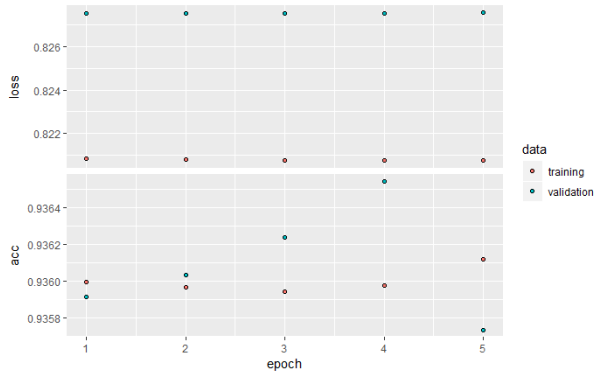


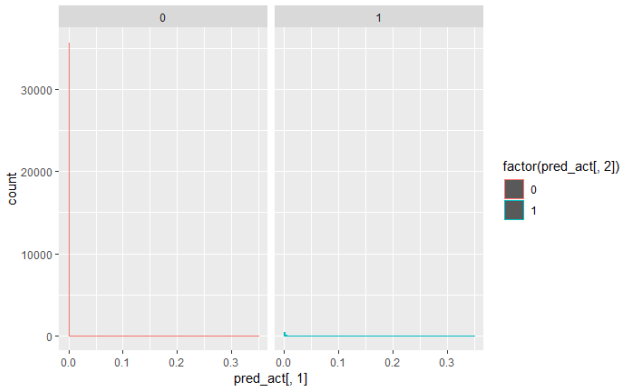
Figure 77

Performance actually decreased when tested on particles in the $3\ GeV < P \leq 4\ GeV$ range, to $\varepsilon_\pi = 1.51\%$ at electron efficiency $\varepsilon_{e^-} = 89.99\%$. This might be explained by the fact that there is a much smaller proportion of electrons in this momentum range than in the lower GeV ranges. Additionally, when tested on particles in the next momentum range, the model's generalizability breaks down and predicts "pion" everywhere.

Table 1

Prediction/Actual	0	1
0	36713	2359
1	0	0

Pion efficiency at 90 % electron efficiency cannot be calculated here, since the optimization algorithm for finding t_{cut} does not converge, because the distribution of the combined probability that a particle is an electron is very narrow, close to zero.



i.e. we arrive at $\varepsilon_\pi = 0\%$ at $\varepsilon_e = 0\%$

The incrementally trained model, which at this stage has already become extremely biased towards predicting pion with a very high certainty, was then trained on particles in the $4\ GeV < P \leq 5\ GeV$ range (Figure 78).

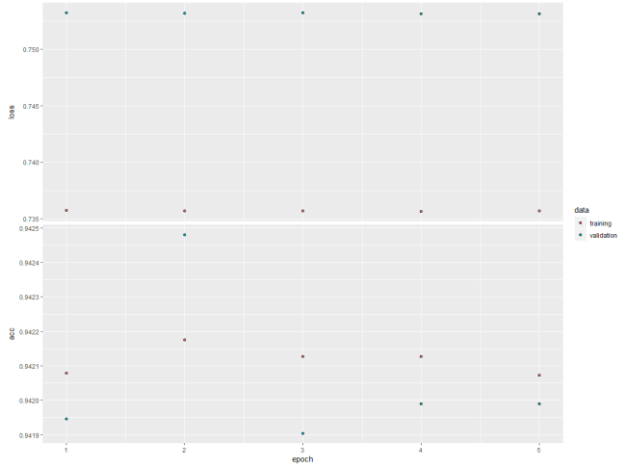


Figure 78

The incrementally trained model seems to increase in accuracy, but only because, at this stage of training, it has seen a lot more pions than electrons, and the focal loss function is no longer able to mitigate the extreme class imbalance. There is no change in the confusion matrix, reported above, prior to training for additional epochs in this high GeV range.

The model was not trained for particles in the $5\ GeV < P \leq 6\ GeV$ range as a separate training set, since it had already started to overfit (beyond rescue) to the imbalanced class distribution.

Error! Reference source not found. summarises the results from the final stage of model-building, in order to allow for comparison with work that had been done in the past on particle identification using the TRD (for that reason **Error! Reference source not found.**, which was also shown in Section **Error! Reference source not found.** is repeated here for comparison).

1.14 The Structure & Organization of this Dissertation

Chapter **Error! Reference source not found.**: **Error! Reference source not found.** begins with a history of atomic theory from Democritus to the Standard Model of Particle Physics, outlines the fundamental particles and forces and the standard model vertices, which explains their interactions; then touches upon two ways in which particles interact with matter relevant to understanding the data used in this thesis, followed by a quick overview of the Quark Gluon Plasma. Next, the European Organization for Nuclear Research (CERN) experiment is discussed, in terms of its establishment, particle acceleration hardware and the various experiments conducted at the LHC today, as well as a discussion of its currently used software packages for data analysis (ROOT) and simulation (Geant4).

Chapter **Error! Reference source not found.**: **Error! Reference source not found.** goes into detail about the ALICE detector, focussing on the Transition Radiation Detector (TRD) and methods currently used for particle identification in the TRD, with a brief overview of their performance.

Chapter **Error! Reference source not found.**: **Error! Reference source not found.** in troduces Deep Learning within the larger context of Machine Learning and Artificial Intelligence, then discusses the original theory upon which modern deep learning is based: Rosenblatt's perceptron. After this, the mathematical background for deep learning used in this dissertation is discussed, including feedforward neural networks, backpropagation, methods for regularization, convolutional- and recurrent neural networks and two types of generative models, namely variational autoencoders and generative adversarial networks.

Chapter **Error! Reference source not found.**: **Error! Reference source not found.** is a short chapter explaining the statistical tests used for particle selection in this thesis at the hand of: hypotheses, significance level and power. It also introduces the concepts of electron- and pion efficiency, which is important in understanding the Results section of this dissertation.

Chapter **Error! Reference source not found.**: Data outlines the format of the data used in this thesis, along with some example plots of TRD tracklet signals, electron and pion counts per run, Bethe Bloch curves for electrons and pions per run as well as nσ-electron plots for electrons and pions per run.

Chapter **Error! Reference source not found.** is the **Error! Reference source not found.** se ction of this dissertation.

Chapter **Error! Reference source not found.** is the **Error! Reference source not found.** se ction of this dissertation, where only the most successful results are discussed.

Chapter **Error! Reference source not found.** contains the **Error! Reference source not found.**, including a deeper analysis of results, as well as an outline of future work which can be done in this area.

“

A man may imagine things that are false, but he can only understand things that are true, for if the things be false, the apprehension of them is not understanding.

”

—Sir Isaac Newton

1.14.1 Discussion of Most Successful Model from Stage Two of Model Building

At the conclusion of the second stage of model building, the following 2D convolutional neural network architecture (Figure 79) achieved the lowest pion efficiency $\varepsilon_{\pi} = 2.2\%$ at electron efficiency $\varepsilon_{e} = 90\%$:

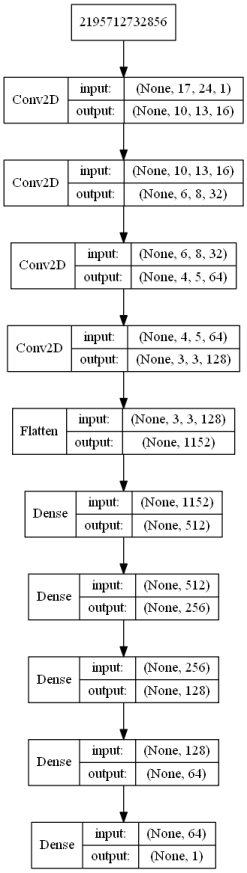


Figure 79: Particle Identification Model Architecture

Using the Adam optimizer with learning rate = 0.00001, trained for 100 epochs with a batch size of 32, using binary cross-entropy as the loss function to be optimized.

The training and validation accuracy and loss graphs for this model are depicted below.

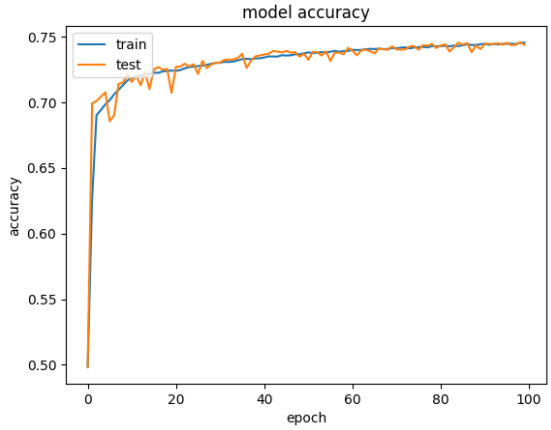


Figure 80: Training vs Validation Accuracy

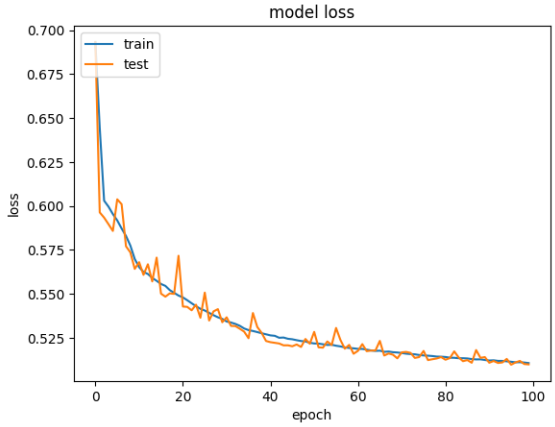


Figure 81: Training vs Validation Loss

Pion efficiency at 90% electron efficiency was calculated by writing a function which, when minimized, finds the cut-off point (critical region) in the distribution of the single node output layer which results in 90% of true electrons being classified as electrons, on a per-tracklet basis. This was done for tracks which left signal in all 6 layers of the TRD, allowing one to construct a test statistic taking all 6 estimates for the track into account, as outlined in **Error! Reference source not found. Error! Reference source not found.**.

The result of the minimization process is shown in Figure 82: any track which receives a combined probability above t_{cut} (shown as a vertical red line) was classified as an electron, otherwise it was classified as a pion.

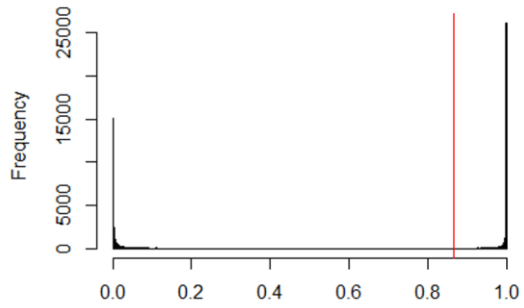


Figure 82: t-statistic selection allowing for 90% electron efficiency

Table 2 shows the obtained confusion matrix, with predicted labels along the rows and actual labels along the columns, here the diagonal of the confusion matrix are particles that were classified correctly:

Table 2: Confusion Matrix for Particle Identification

Prediction/Actual	0	1
0	47 833	4 893
1	1 088	44 028

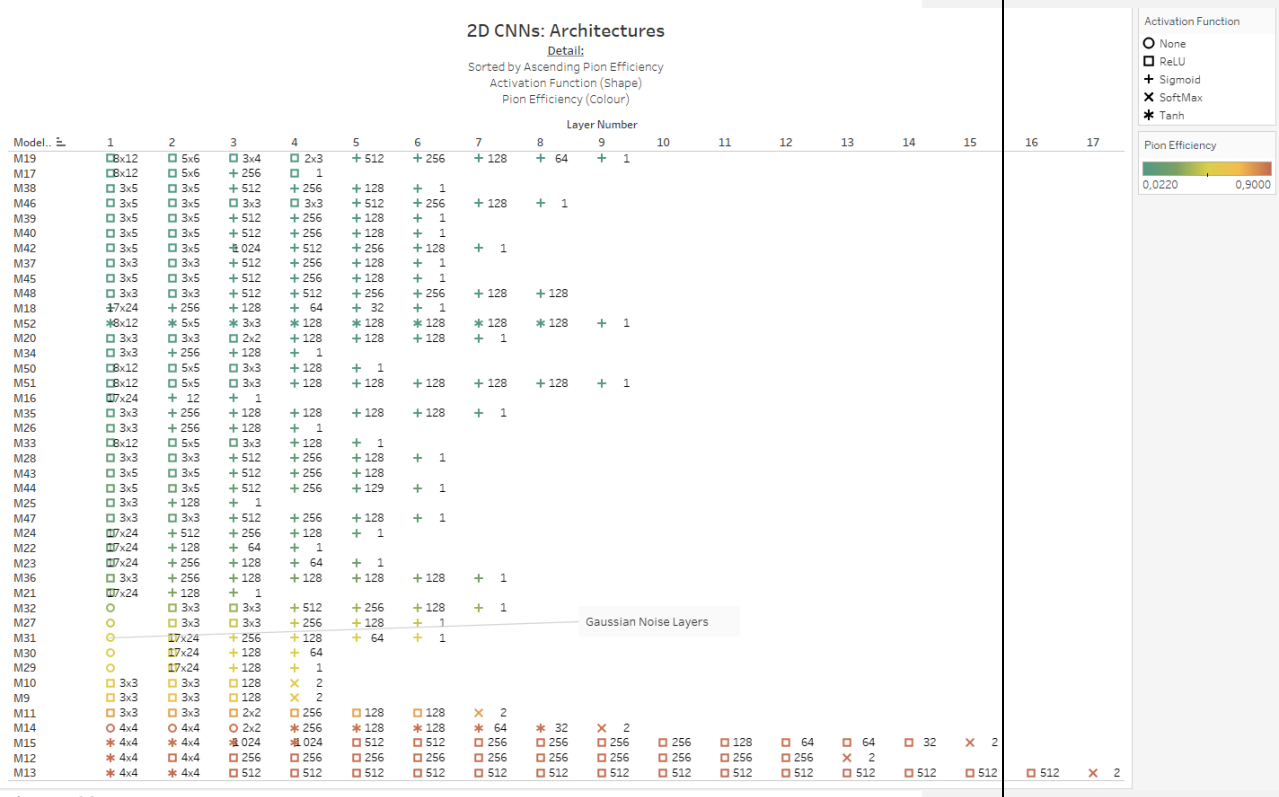

2D CNNs: Architectures
Detail:
Sorted by Ascending Pion Efficiency
Activation Function (Shape)
Pion Efficiency (Colour)
Layer Number
Model...
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
M19 8x12 5x6 3x4 2x3 512 256 128 64 1
M17 8x12 5x6 256 1
M38 3x5 3x5 512 256 128 1
M46 3x5 3x5 3x3 3x3 512 256 128 1
M39 3x5 3x5 512 256 128 1
M40 3x5 3x5 512 256 128 1
M42 3x5 3x5 1024 512 256 128 1
M37 3x3 3x3 512 256 128 1
M45 3x5 3x5 512 256 128 1
M48 3x3 3x3 512 512 256 256 128 128
M18 7x24 256 128 64 32 1
M52 8x12 5x5 3x3 128 128 128 128 128 1
M20 3x3 3x3 2x2 128 128 128 1
M34 3x3 256 128 1
M50 8x12 5x5 3x3 128 1
M51 8x12 5x5 3x3 128 128 128 128 128 1
M16 7x24 12 1
M35 3x3 256 128 128 128 128 1
M26 3x3 256 128 1
M33 8x12 5x5 3x3 128 1
M28 3x3 3x3 512 256 128 1
M43 3x5 3x5 512 256 128
M44 3x5 3x5 512 256 129 1
M25 3x3 128 1
M47 3x3 3x3 512 256 128 1
M24 7x24 512 256 128 1
M22 7x24 128 64 1
M23 7x24 256 128 64 1
M36 3x3 256 128 128 128 128 1
M21 7x24 128 1
M32 3x3 3x3 512 256 128 1
M27 3x3 3x3 256 128 1
M31 7x24 256 128 64 1
M30 7x24 128 64
M29 7x24 128 1
M10 3x3 3x3 128 2
M9 3x3 3x3 128 2
M11 3x3 3x3 2x2 256 128 128 2
M14 4x4 4x4 2x2 256 128 128 64 32 2
M15 4x4 4x4 1024 1024 512 512 256 256 256 256 128 64 64 32 2
M12 4x4 4x4 256 256 256 256 256 256 256 256 256 256 256 2
M13 4x4 4x4 512 512 512 512 512 512 512 512 512 512 512 512 512 512 2
Gaussian Noise Layers
Activation Function
None
ReLU
Sigmoid
SoftMax
Tanh
Pion Efficiency
0,0220
0,9000

Figure 83

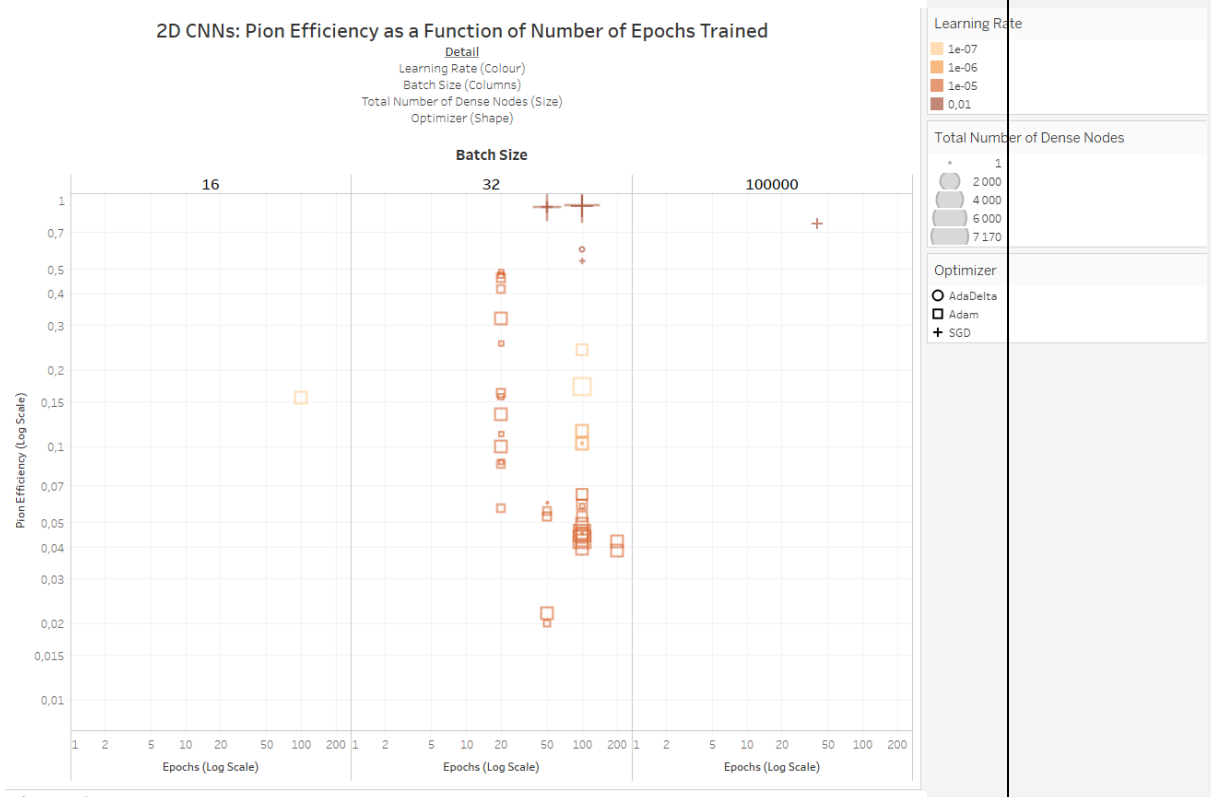

2D CNNs: Pion Efficiency as a Function of Number of Epochs Trained
Detail
Learning Rate (Colour)
Batch Size (Columns)
Total Number of Dense Nodes (Size)
Optimizer (Shape)
Batch Size
16
32
100000
Pion Efficiency (Log Scale)
1
0,7
0,5
0,4
0,3
0,2
0,15
0,1
0,07
0,05
0,04
0,03
0,02
0,015
0,01
1 2 5 10 20 50 100 200
Epochs (Log Scale)
Learning Rate
1e-07
1e-06
1e-05
0,01
Total Number of Dense Nodes
1
2 000
4 000
6 000
7 170
Optimizer
AdaDelta
Adam
SGD

Figure 84

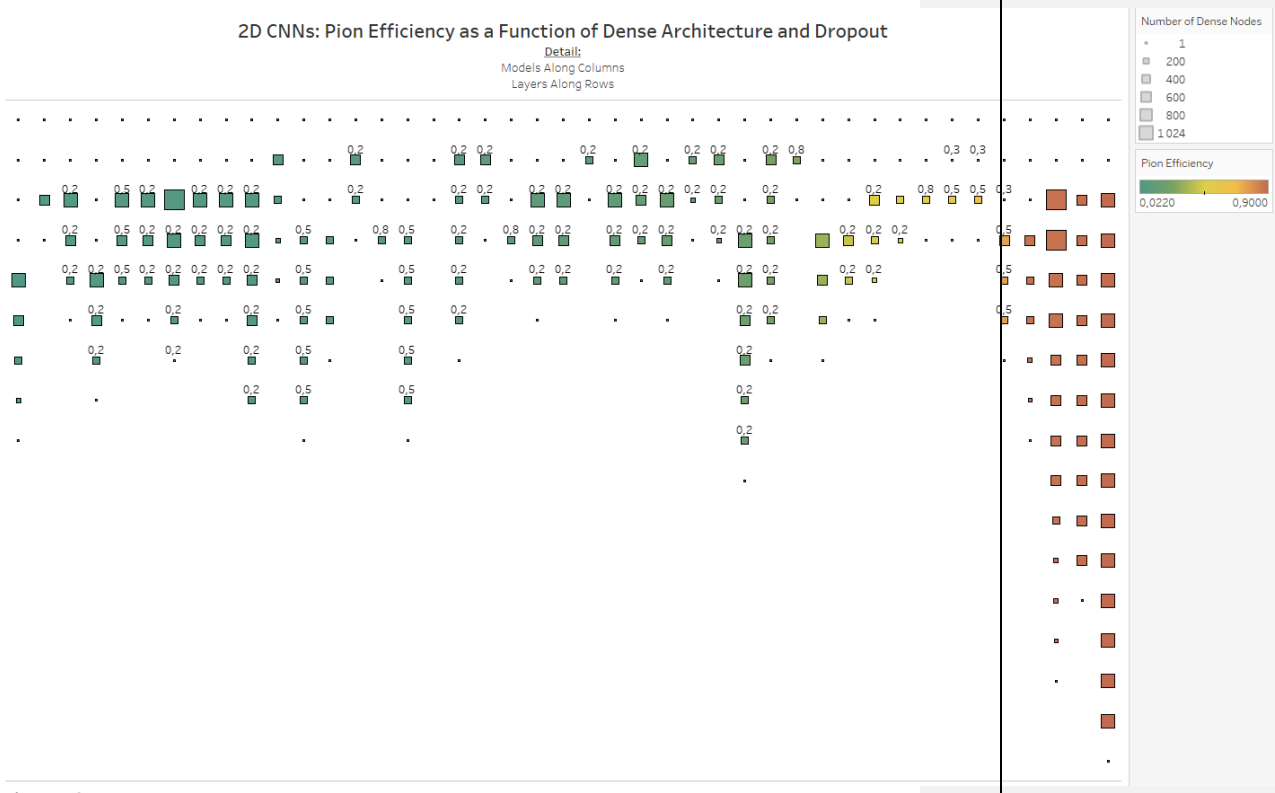

2D CNNs: Pion Efficiency as a Function of Dense Architecture and Dropout
Detail:
Models Along Columns
Layers Along Rows
Number of Dense Nodes
1
200
400
600
800
1 024
Pion Efficiency
0,0220
0,9000

Figure 85

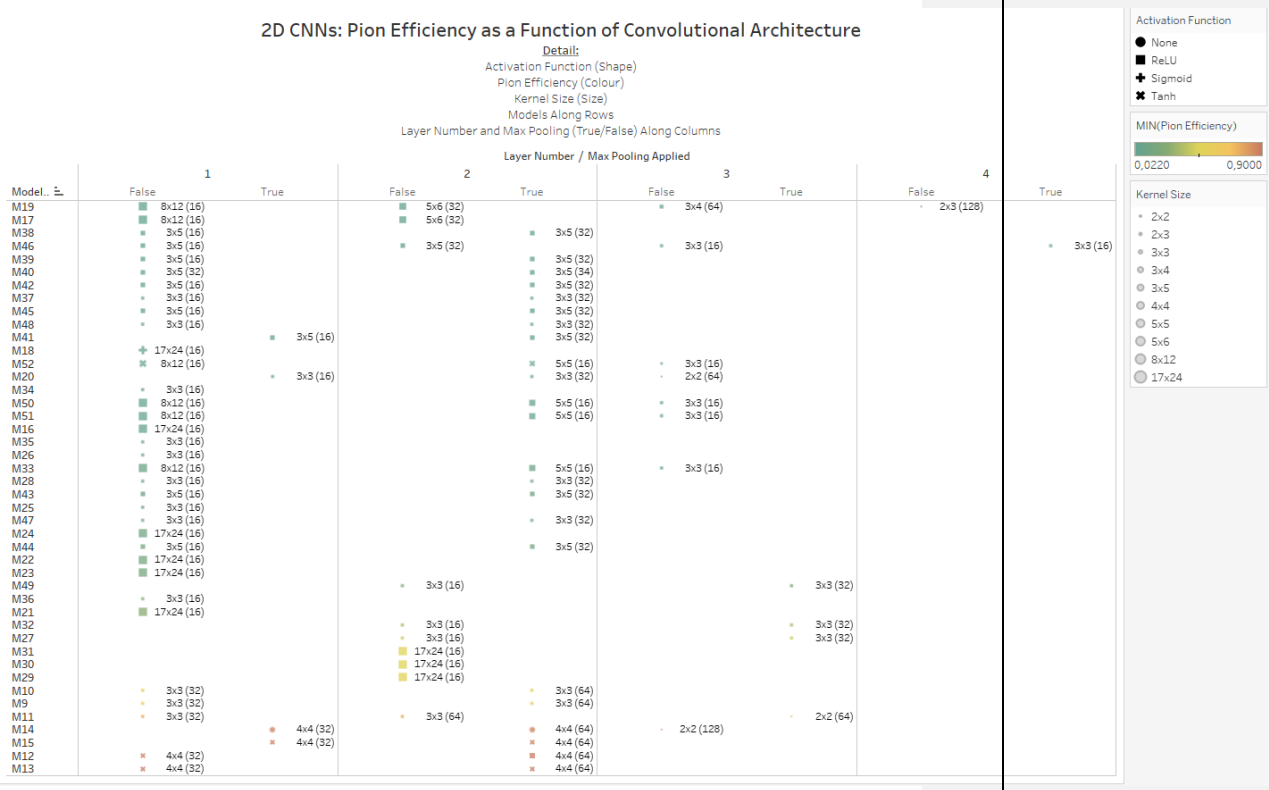

2D CNNs: Pion Efficiency as a Function of Convolutional Architecture
Detail:
Activation Function (Shape)
Pion Efficiency (Colour)
Kernel Size (Size)
Models Along Rows
Layer Number and Max Pooling (True/False) Along Columns
Layer Number / Max Pooling Applied
1
2
3
4
False
True
Model..
M19 8x12 (16) 5x6 (32) 3x4 (64) 2x3 (128)
M17 8x12 (16) 5x6 (32)
M38 3x5 (16) 3x5 (32)
M46 3x5 (16) 3x5 (32) 3x3 (16) 3x3 (16)
M39 3x5 (16) 3x5 (32)
M40 3x5 (32) 3x5 (34)
M42 3x5 (16) 3x5 (32)
M37 3x3 (16) 3x3 (32)
M45 3x5 (16) 3x5 (32)
M48 3x3 (16) 3x3 (32)
M41 3x5 (16) 3x5 (32)
M18 17x24 (16)
M52 8x12 (16) 5x5 (16) 3x3 (16)
M20 3x3 (16) 3x3 (32) 2x2 (64)
M34 3x3 (16)
M50 8x12 (16) 5x5 (16) 3x3 (16)
M51 8x12 (16) 5x5 (16) 3x3 (16)
M16 17x24 (16)
M35 3x3 (16)
M26 3x3 (16)
M33 8x12 (16) 5x5 (16) 3x3 (16)
M28 3x3 (16) 3x3 (32)
M43 3x5 (16) 3x5 (32)
M25 3x3 (16)
M47 3x3 (16) 3x3 (32)
M24 17x24 (16)
M44 3x5 (16) 3x5 (32)
M22 17x24 (16)
M23 17x24 (16)
M49 3x3 (16) 3x3 (32)
M36 3x3 (16)
M21 17x24 (16)
M32 3x3 (16) 3x3 (32)
M27 3x3 (16) 3x3 (32)
M31 17x24 (16)
M30 17x24 (16)
M29 17x24 (16)
M10 3x3 (32) 3x3 (64)
M9 3x3 (32) 3x3 (64)
M11 3x3 (32) 3x3 (64) 2x2 (64)
M14 4x4 (32) 4x4 (64) 2x2 (128)
M15 4x4 (32) 4x4 (64)
M12 4x4 (32) 4x4 (64)
M13 4x4 (32) 4x4 (64)
Activation Function
None
ReLU
Sigmoid
Tanh
MIN(Pion Efficiency)
0,0220
0,9000
Kernel Size
2x2
2x3
3x3
3x4
3x5
4x4
5x5
5x6
8x12
17x24

Figure 86

1.14.2 Variations on the GAN concept used towards Event Simulation in this Dissertation

1.14.2.1 Auxiliary Classifier Generative Adversarial Network

Auxiliary Classifier Generative Adversarial Networks enforce class label conditioned synthesis models, i.e. by feeding a GAN the class label of an image $c \sim p_c$, G is now a function of both a noise vector and the label of the image, i.e. $G(z, c)$, whereas D is not only tasked with classifying whether data is real or simulated, but also the class label of the image. This process has been shown to stabilize training [43].

1.14.2.2 Adversarial Autoencoder

Adversarial Autoencoders match the aggregated posterior of the latent space vector from an autoencoder $q(z) = \int_x q(z|x)p_d(x)dx$ with an arbitrary prior distribution $p(z)$, a process which results in meaningful samples being generated from any sample from any part of the prior space. The decoder function learns a function to map from the imposed prior distribution to the data distribution. In this set-up, the generator of the GAN also acts as the encoder function of the autoencoder, a process which assists the generator in fooling the discriminator of the GAN into misclassifying simulated data as real data [44].

1.14.2.3 Bidirectional Generative Adversarial Network

Bidirectional Generative Adversarial Networks (BiGANs) make use of an encoder function which maps data x into a latent feature space z for the generative model, i.e.: $E: \Omega_x \rightarrow \Omega_z$. Here the discriminator has access to both the (simulated or real) x, as well as its latent encoding z when classifying samples as real or simulated. In this way, BiGANs not only learn how to map from a latent space to data, but how to perform the inverse mapping, i.e. data to latent space [45].

1.14.2.4 Deep Convolutional Generative Adversarial Network

Deep Convolutional Generative Adversarial Networks make use of fully convolutional neural networks for both the generator and discriminator, using strided convolutions to allow these networks to learn their own spatial downsampling in the case of the discriminator and spatial upsampling in the case of the generator. These models also make use of Batch Normalization, which ensures that the input to any hidden unit has zero mean and a variance of 1, a process which compensates for poor initialization strategies, helps with the flow of gradients through complex models and preventing the generator from outputting similar simulated images from any sample of the noise space, a common issue which plagues GANs [46].

1.14.2.5 Least Squares Generative Adversarial Network

Whereas regular GANs use the sigmoid cross entropy loss function, which results in vanishing gradients when samples are generated that are classified as real data, but that are still far from looking like real data, Least Squares Generative Adversarial Networks (LSGANs) use an adaptation of the least squares loss function for the discriminator network, which has been shown to result in images of higher quality and result in networks that are more stable during training [47].

Before commencing deep learning, the dataset was sanitized to ensure that similar ranges for key variables were covered by both real and simulated data, Eta distributions for real and simulated data were similar.

The following cut on Eta was applied to both datasets:

$$|\eta| \leq 0.9$$

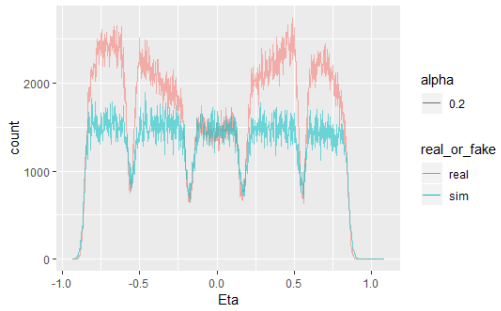


Figure 87: Eta distributions for real and Geant4 simulated data

$n\sigma_\pi$ estimates from the Time Projection Chamber (TPC) were dissimilar for real and simulated data ($n\sigma_\pi$ = -999 is an error flag).

The following cut on $n\sigma_\pi$ was applied to real and simulated data:

$$|n\sigma_\pi| \leq 3$$

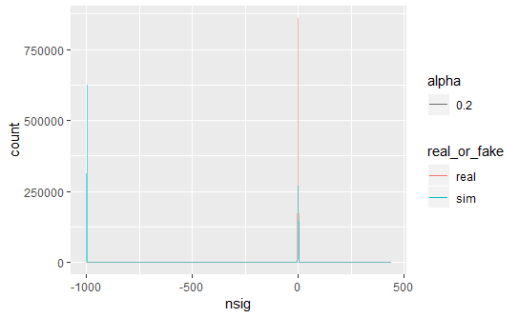


Figure 88: $n\sigma_\pi$ estimate (TPC) distributions for real and Geant simulated data, before cut

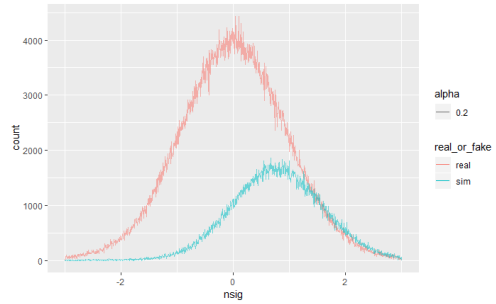


Figure 89: nsigma pion distributions after applying cut

Momentum ranges for real and simulated data were also quite dissimilar, the following cut on momentum was applied to both real and simulated data:

$$1.5 \leq P \leq 20$$

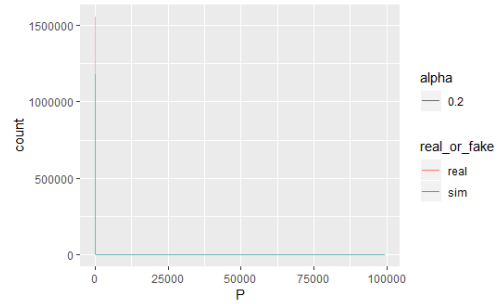


Figure 90: Momentum distributions for both real and Geant4 simulated data, before cut

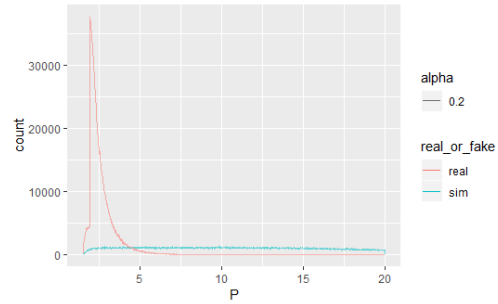


Figure 91: Momentum distributions after cut

Distinguishing Geant4 simulated data from real data proved to be trivial compared to distinguishing real pions from real electrons. What follows is an analysis of features between the two datasets to investigate what differentiates them.

Firstly, the mean and standard deviation of ADC values for real and simulated datasets are not the same, i.e. $\mu_{sim} = 2.178$; $\mu_{real} = 4.527$ and $\sigma_{sim} = 11.989$; $\sigma_{real} = 17.995$. The distribution of simulated data's ADC values is slightly more skewed to the right than that of real data, i.e. $\gamma_{1_{sim}} = 19.170$; $\gamma_{1_{real}} = 17.391$. However, the maximum ADC value for both datasets is the same, i.e. $max_{sim} = max_{real} = 1023$.

Figure 92 and Figure 93 show the distribution of ADC values for simulated and real data, respectively.

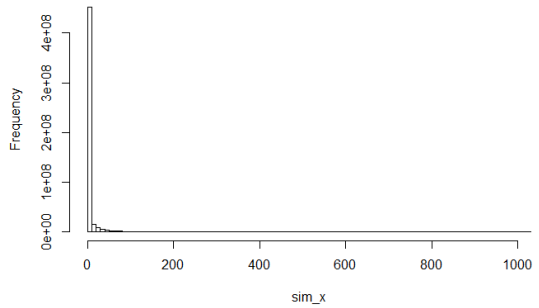


Figure 92: Distribution of ADC values for simulated data

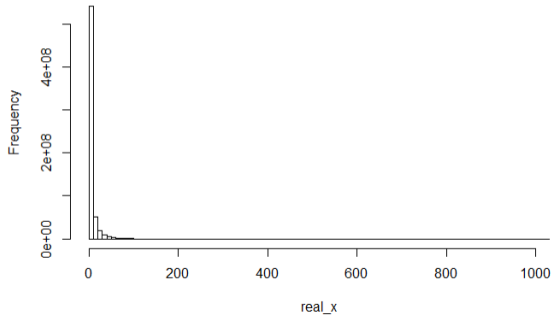


Figure 93: Distribution of ADC values for real data

Looking at the number of pads that don't contain any data (i.e. the number of rowsums of the image that are equal to 0), one sees the following distributions for simulated and real data:

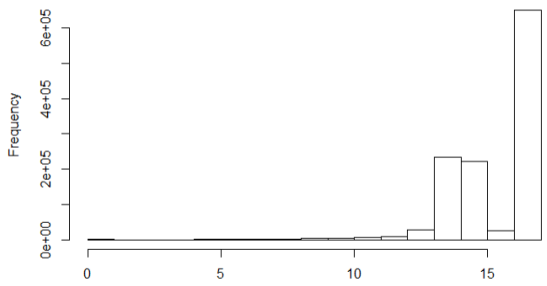


Figure 94: Distribution of the number of pads with no data per image for simulated data

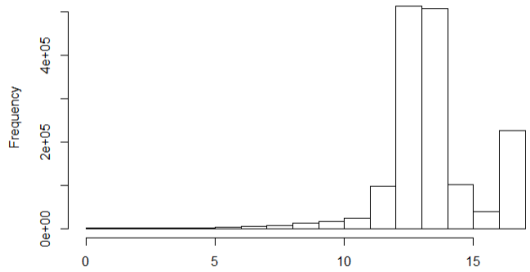


Figure 95: Distribution of the number of pads with no data per image for real data

While these distributions are quite similar, one can see that real data is slightly more skewed towards the left and has far fewer instances of images which are completely devoid of signal, i.e. all pixels in the image are equal to zero and therefore all 17 pads have no signal.

There are also noticeable differences in the distribution of mean ADC values per pad for real and simulated data (see Figure 96 and Figure 97).

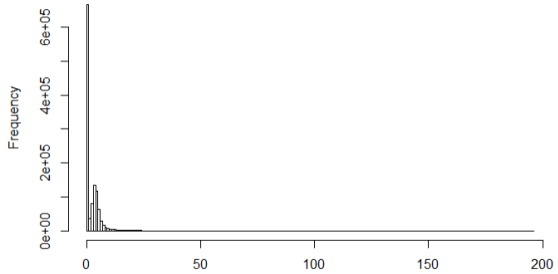


Figure 96: Distribution of mean ADC value per image for simulated data

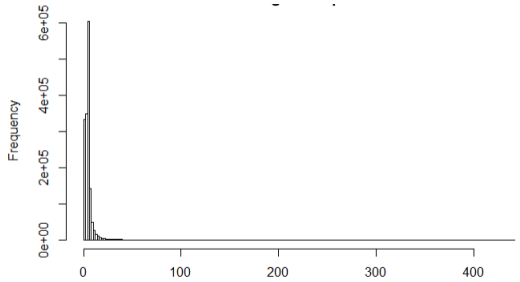


Figure 97: Distribution of mean ADC value per image for real data

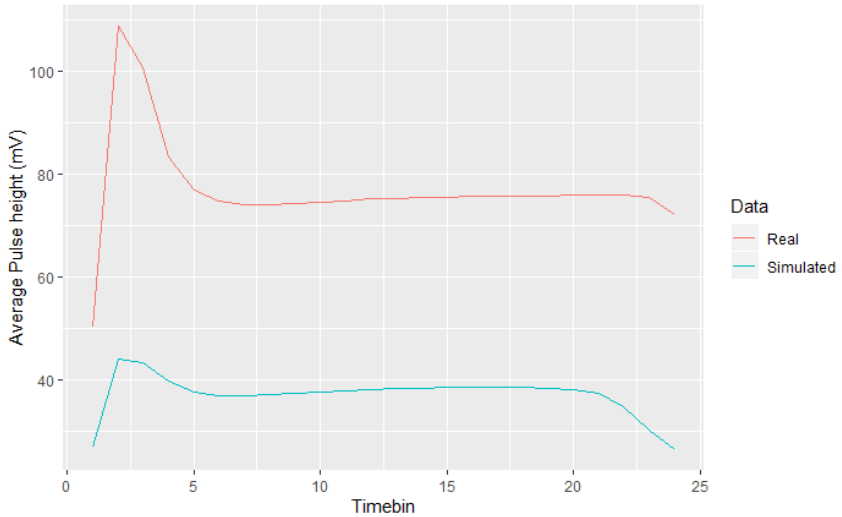


Figure 98: Average Pulse-height as a function of time-bin for Real and Geant4-simulated pion tracklets

Figure 99 illustrates how implementing a convolution with stride = 2, i.e. only sampling every second pixel for convolution, is mathematically equivalent to performing downampling after a convolution applied to all pixels (i.e. stride = 1), followed by downsampling [34].

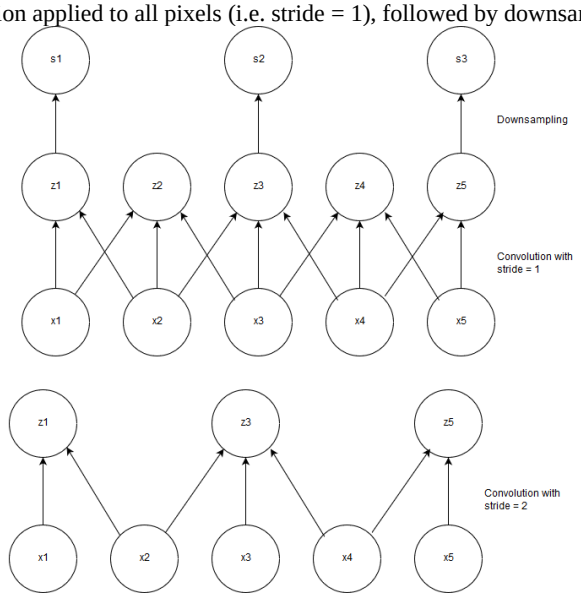


Figure 99: Illustration of mathematical equivalence of implementing a convolution with unit stride followed by downsampling to implementing a convolution with stride = 2.

A simplified diagram depicting this process can be seen in Figure 100.

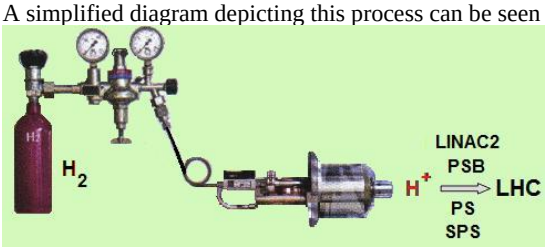
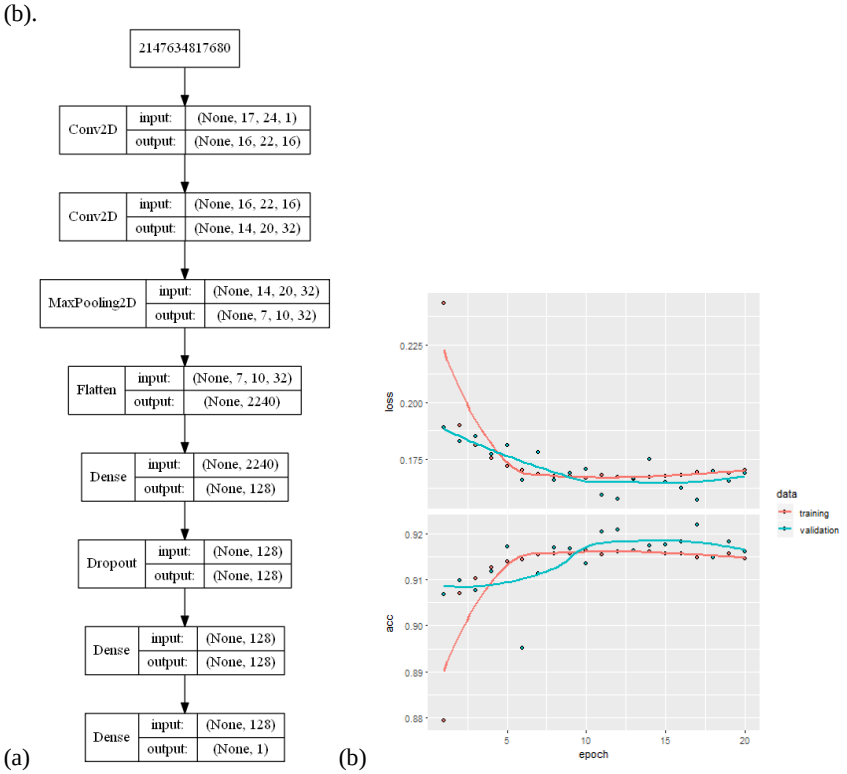


Figure 100: The LHC Proton Source, connected to the Duoplasmatron device, which strips electrons off Hydrogen molecules, to produce the beams of protons which eventually collide within the LHC [13]

The truncated mean signal is the combined signal of Transition Radiation + Specific Ionization Energy, this method focusses on classifying electrons vs pions based on their expected energy loss as per the Bethe Bloch curve shown in

Geant4 simulations were configured using https://github.com/PsycheShaman/trdpid/blob/master/sim/Config.C , simulations were run as per the following shell script: https://github.com/PsycheShaman/trdpid/blob/master/sim/runtest.sh which calls upon the simulation script https://github.com/PsycheShaman/trdpid/blob/master/sim/sim.C the reconstruction script https://github.com/PsycheShaman/trdpid/blob/master/sim/rec.C and the analysis script https://github.com/PsycheShaman/trdpid/blob/master/sim/ana.C in sequence in order to create Monte Carlo simulations in a similar format to raw data analysed during particle identification.

The task of distinguishing simulated from real data was performed using a 2D convolutional neural network, with architecture shown in Figure 101 (a).
Distinguishing Geant4 simulations from real data proved to be a much easier task than distinguishing real electrons from real pions, as depicted in the training graphs in Figure 101 (b).



(a) (b)

Figure 101: Training curves (a) and model architecture (b) for distinguishing real from Geant4-simulated data

Table 3 shows the obtained confusion matrix for the following model architecture:
Table 3: Confusion Matrix for distinguishing between Geant4 vs Real Data

Prediction/Actual	π_{geant}	π_{real}
π_{geant}	42 553	681

$\boldsymbol{\pi_{real}}$	7 069	24 058

While these results look quite believable at first glance, it was quite easy to distinguish 100 000 real data samples vs 100 000 samples simulated with VAE using a CNN to 100% accuracy, as can be seen in Figure 102.

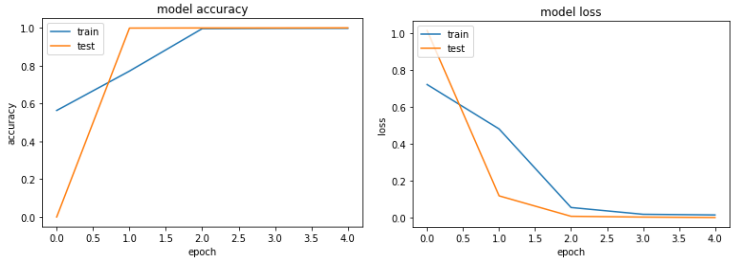


Figure 102: Training accuracy and loss curves for training vs validation data

In order to quantitatively compare the accuracy of the generative models built during the course of the project and Geant4 simulations, a Convolutional Neural Network was trained to 92% accuracy, with 75 000 observations of each type of simulated data combined labeled as 0 and an equivalent amount of real data (75 000 x 4) labeled as 1, used as the training set. Each training set's images were independently scaled to be in the range [-1;1].
Figure 103 shows the distribution of $P(real)$ predictions made by this neural network on an independent test set of size 25 000 for each model type and 25 000 real observations.

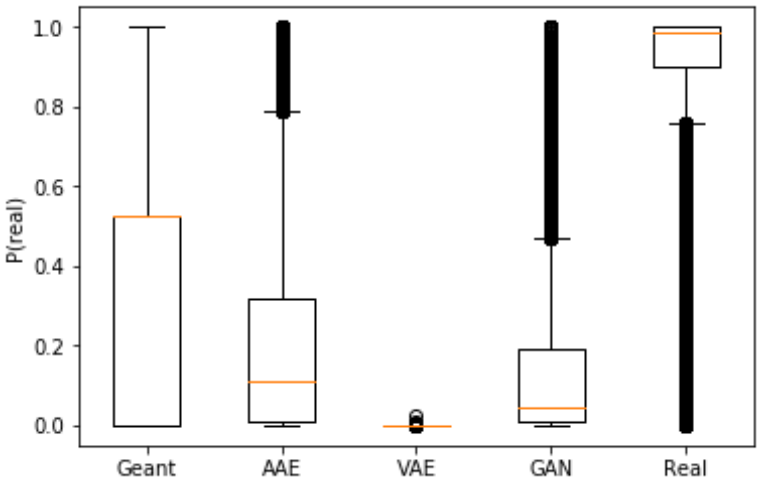
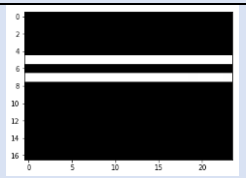
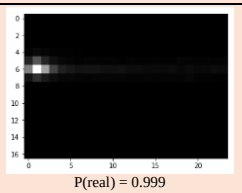
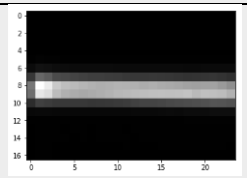
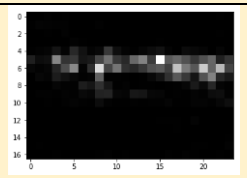
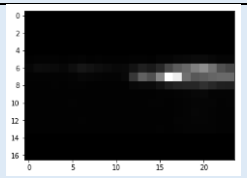
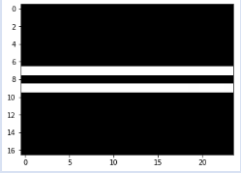
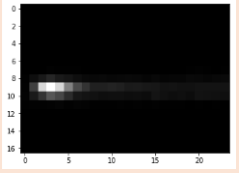
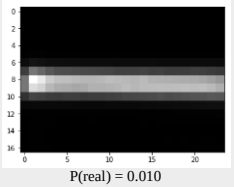
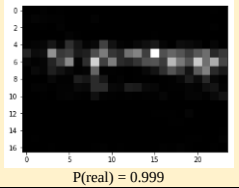
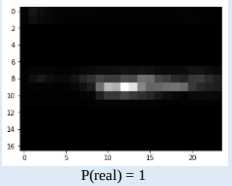
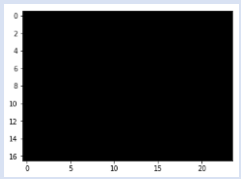
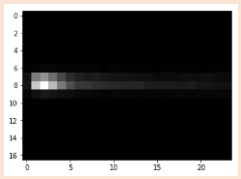
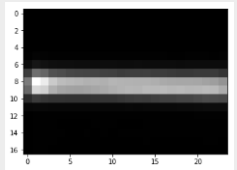
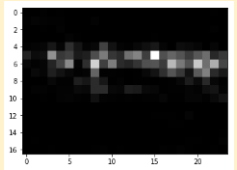
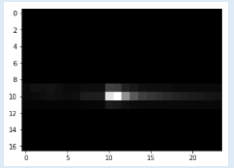
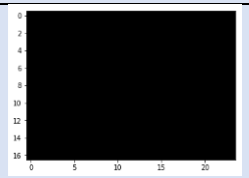
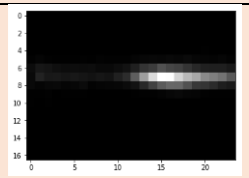
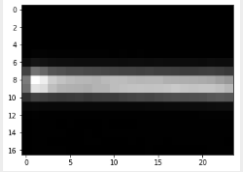
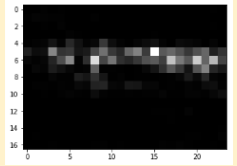
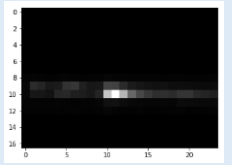
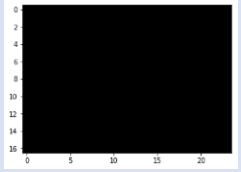
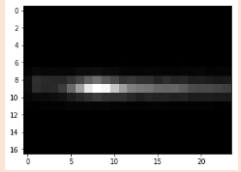
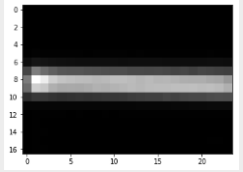
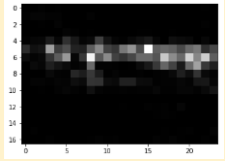
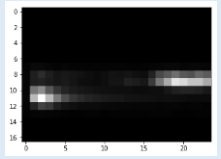
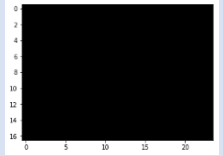
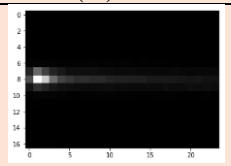
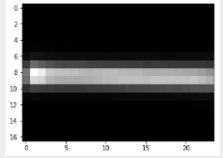
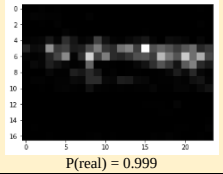
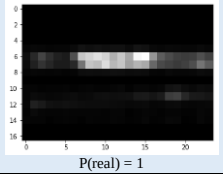


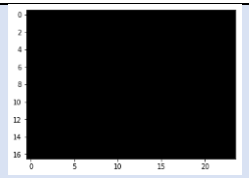
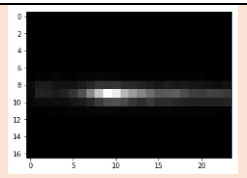
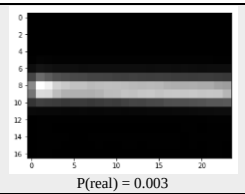
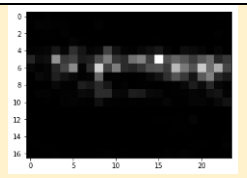
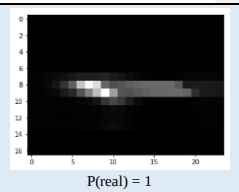
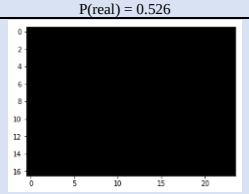
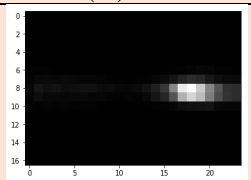
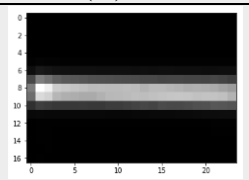
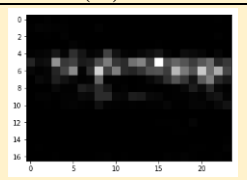
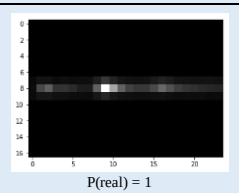
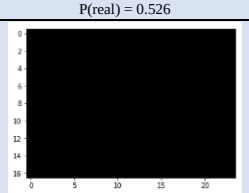
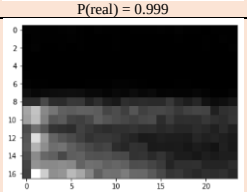
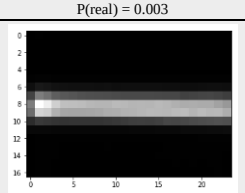
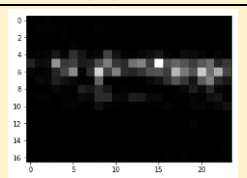
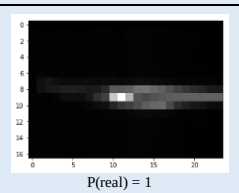
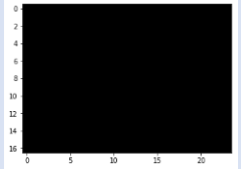
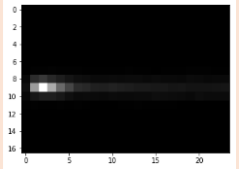
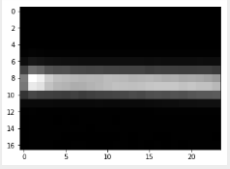
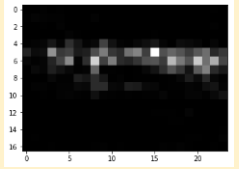
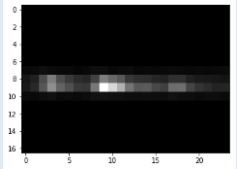
Figure 103: $P(real)$ predictions made (by a neural network trained to discriminate 4 kinds of simulated data from real data) on an independent test set of each type of simulate dataset, as well as real observations. Most real observations received a prediction probability above 0.8, verifying that the model has decent statistical power to discriminate real observations from simulated data, although there appears to be a long tail of outliers that the model could not correctly identified as being real. By looking at this graph, it seems that VAE data was quite easily classified as simulated data (with no probability predictions reaching above 0.1) and can therefore be deemed as inaccurate,

Geant4 data generally received higher probability predictions than the other methods of simulation with a median prediction score of above 0.5 and upper quartile reaching up to 1, but AAE and GAN data both have outlier observations which seem to have fooled this neural network to think they are real as well and can be deemed to be more accurate than VAE data, with AAE data's output distribution slightly more skewed towards 1 than GAN data.

Table 4: For each simulated dataset, as well as real data: the tracklets which received the highest P(real) prediction from the neural network discussed above. This tells an entirely different story and shows that many of the **Geant4** simulated tracklets are completely empty images (a phenomenon which *does* occur in real data, as discussed) and the top two observations appear qualitatively incorrect; although it is theoretically possible that such observations might have existed in the real dataset, it is probably because something went wrong. While it is still difficult to draw any absolute conclusions from looking at such a small sample, **AAE** data seems to be the only dataset whose generative procedure was capable of fully capturing the underlying data distribution. **VAE** data received consistently low probability scores and most samples look very similar, whereas the top **GAN** images appear to all be exactly the same image; while this indicates that the **GAN** managed to create at least one image which fools a discriminative network, it also shows that it failed to capture the underlying distribution as well as the **AAE**, which produced many observations which were hard to discriminate from real data, and which all look dissimilar.

n	Geant	AAE	VAE	GAN	Real
1	 P(real) = 1	 P(real) = 0.999	 P(real) = 0.026	 P(real) = 0.999	 P(real) = 1
2	 P(real) = 0.999	 P(real) = 0.999	 P(real) = 0.010	 P(real) = 0.999	 P(real) = 1

3	 P(real) = 0.526	 P(real) = 0.999	 P(real) = 0.006	 P(real) = 0.999	 P(real) = 1
4	 P(real) = 0.526	 P(real) = 0.999	 P(real) = 0.005	 P(real) = 0.999	 P(real) = 1
5	 P(real) = 0.526	 P(real) = 0.999	 P(real) = 0.004	 P(real) = 0.999	 P(real) = 1
6	 P(real) = 0.526	 P(real) = 0.999	 P(real) = 0.003	 P(real) = 0.999	 P(real) = 1

7	 P(real) = 0.526	 P(real) = 0.999	 P(real) = 0.003	 P(real) = 0.999	 P(real) = 1
8	 P(real) = 0.526	 P(real) = 0.999	 P(real) = 0.003	 P(real) = 0.999	 P(real) = 1
9	 P(real) = 0.526	 P(real) = 0.999	 P(real) = 0.002	 P(real) = 0.999	 P(real) = 1
10	 P(real) = 0.526	 P(real) = 0.999	 P(real) = 0.002	 P(real) = 0.999	 P(real) = 1

Chapter 1: Appendices