Reconstructing Particle Trajectories in 2D

Gido Verheijen s1008055*
Radboud University

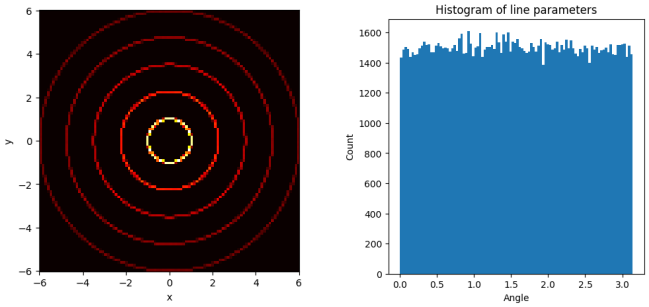
I report on a two-step approach for track reconstruction in particle physics using machine learning. A Siamese-Network-inspired convolutional network, called the classifier, is trained to determine if hits belong to the same track. Subsequently, a grouping process based on similarity scores is employed to assign coordinates to tracks. The limitations of regression models, such as static input size and poor generalization, motivate this approach. Experimental results highlight the potential of machine learning for accurate track identification in particle physics experiments. Further research can refine and extend the proposed methodology for broader applications.

When particles collide in a large particle collider like in the ATLAS experiment at the LHC, many new particles are generated and propelled in various directions. Identifying the charge and momentum of all particles resulting from a collision is crucial for studying them in these experiments. A network of detectors is placed around the collision area, each of which registers 'hits', interactions between the particles and the detectors. These hits are utilized to reconstruct the paths or 'tracks' along which the particles traveled after the collision.

In the present letter, I present my results of a deep learning network, which utilizes a simplified model to reconstruct particle tracks based on their hits.

This simplified model differs from the classical problem in the following ways. Firstly the particles traveling through the detectors can only travel in two dimensions, instead of three dimensionalities of the real world. Secondly, the particles are assumed to travel in straight lines, instead of the 4-5 parametric helix functions that result from the particles interacting with the magnetic fields [1]. Lastly, we assume that all particles originate in the origin. These three requirements collectively result in track descriptions that can be fully characterized by a single parameter—the line angle α.

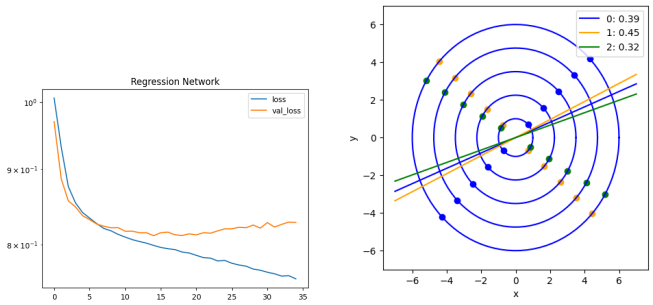
To create data for the neural networks I created a simulation that could generate hits in the simplified model. Five circular detectors were created each of which was centered at the origin, with radii ranging from 2 to 6 with steps of 1 between them. Then a number of tracks could be generated by their angle, which then could be used to generate the intersection points of the tracks and the detector circles, which were returned as the hits found by the detectors. To enhance realism, some augmentations were applied to the input data. Specifically, the coordinates were shuffled to remove any grouping of coordinates from the same track. I simulated 50000 experiments each having three tracks, and observed if the results were as expected. FIG 1 shows the distribution of datapoints across both the angle space and the hit space. From these two graphs it is clear that the data is well distributed. Note that the angles only range from 0 to π, this was done to for the reason that the lines have no directionality in this simulation. This means that a track with an angle of 0.5π has the same hit coordinates as one with an angle of 1.5π. To make it easier for the neural networks to predict the angles, I chose to exclude angles above π in this particular experiment. However, if directionality were significant, the angle generation could easily be adjusted to include the missing angles.



(a) (b)

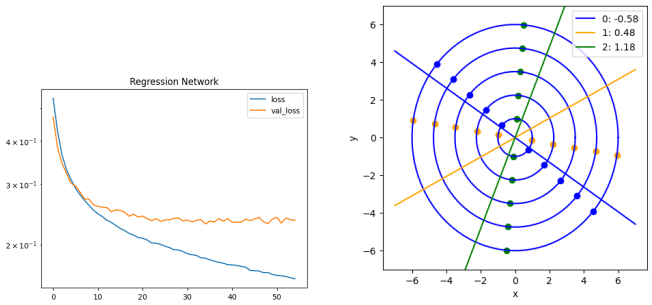
Figure 1: Two histograms showing the distribution of the randomly sampled data from the simulation. (a) The 2d histogram of hits generated by 150,000 random angle tracks. (b) The histogram shows the distribution of angles in the 150,000 randomly generated tracks. It is clear that we are sampling from a flat distribution.

The first method that was used to determine the track parameters was a simple dense regression model. This model consisted of 5 dense layers of sizes 256, 128, 64, and 32 interspersed by dropout layers, that converted the (30,2) input into a (3,1) output. The input of this model was 30 (x,y) coordinates of different hits corresponding to three tracks, of which the angles are outputted by the model. Some optimizations were done to make the model perform slightly better. Both the input and the output of the regression model were normalized. Rescaling the input and output data to be normalized was expected to improve model performance by ensuring the majority of the data falls within a suitable range for training and prediction. The network utilized the MSE loss, and the ADAM optimizer [2] with a learning rate of $10^{-}4$. The network was trained for a maximum of 100 epochs, but used an early stopping metric to stop once the validation



(a) (b)

Figure 2: The data for the first regression model with pure random inputs and outputs (a) The training graph belonging to the network (b) An example of a predicted output of the network, it is clear that the predictions are quite bad. Note that this algorithm only predicts the parameter values, the grouping of variables was done by hand.



(a) (b)

Figure 3: The data for the second regression model with pure random inputs and sorted outputs (a) This graph illustrates the training progress of the second regression model. It shows the loss or error value as a function of the number of training epochs. Compared to the first regression model, the training graph of the second model demonstrates a noticeable improvement. However, the model's performance is still not optimal. (b) An example of a predicted output of the network, the predictions are much better than the first regression model but still not very good. Note that this algorithm only predicts the parameter values, the grouping of variables was done manually.

loss had not improved in the last 20 epochs. A validation set of 10% of the training set was used to track the performance of the model throughout its training process.

The performance of the regression model was found to be unsatisfactory. As can be seen in FIG 2, the network quickly stopped learning at a very high error and the angles it predicted were quite bad. This makes sense on a theoretical basis, as the network does not take into account that any permutation of the input and output data must result in the same results. This makes it much harder for the network to learn any underlying patterns in the data, making it bad at generalizing to data it has never seen before.

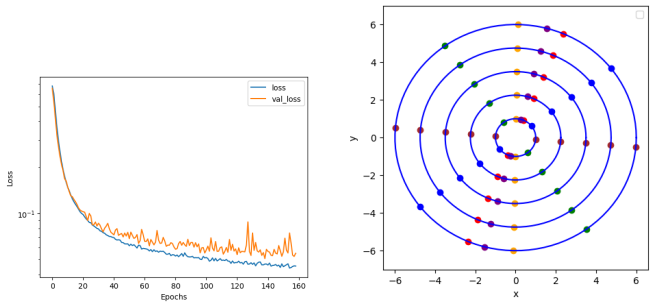
To make up for this a little, we can do some basic permutations on the training data. FIG 3 shows the improvement that can already be obtained by just sorting the output angles by their size. The results of this network were a lot better already than the previous one, we can see that due to the validation loss being much lower than in the previous case. To get an estimate on how stable the network is at predictions, we can look at the uncertainty of the predictions by predicting a couple of times and seeing what the variance is. Due to the dropout layers used, each prediction will be slightly different, and the standard deviation of these predictions gives us this metric. The improved regression model has a mean standard deviation of 0.2, this shows that the network is not very stable in its predictions, giving wildly varied results each time.

More improvements could be obtained by sorting the input values in a similar way. One could for instance first give all hits that were obtained from the first detector ordered by their x value, then the hits obtained from the second detector etc. However, it should be noted that the effect of this additional permutation was not verified. This is because the regression model is tailored to a specific case and does not generalize well to the full problem.

A big limitation of the previous regression model is the static input size of the model. In the real world not every track contains the same amount of hits, nor do we know for sure how many tracks exist in our input data. The (30,2) input and (3,1) output of the regression model specifically only work when given three tracks that each contain exactly 10 hits. We would like the model to be able to get the tracks from a random set of hits, which means that a regression model does not work for this problem.

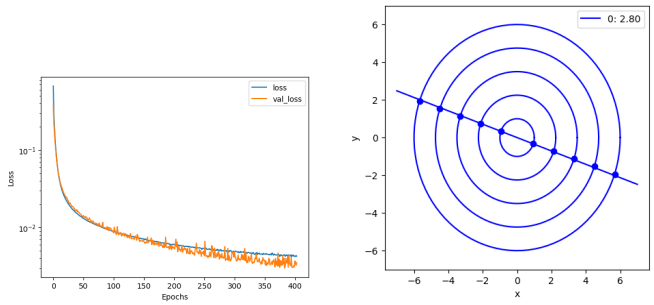
Instead, a two-step approach was used to determine the track parameters. The first step of this approach involved utilizing a model that was inspired by Siamese Neural Networks [3]. The basic use of these networks is to take two input values, and outputs whether the two inputs are similar or not. While siamese networks usually consists out of two twin networks that each output a value, I have combined them into a single convolutional network, that gives a (2,1) Softmax [4] output that decides whether two coordinates belong to the same track or not. Softmax was chosen here such that the output represented the chance that the two hits either belonged to the same track, or were part of different tracks.

This network, to which I will from this point refer to as the classifier, was trained on 50,000 pairs of coordinates of which half were of the same track and half of



(a) (b)

Figure 4: The data for the classification model (a) The training graph belonging to the distinguishing network (b) An example of a predicted output of the network. Note that this network only groups the hits, it does not produce the track parameters.



(a) (b)

Figure 5: The data for the angle regressor model (a) The training graph belonging to the network (b) An example of a predicted output of the network. Note that this network predicts the track parameters, it does not group the hits.

different tracks. These coordinates were inputted into a 1D convolutional layer with size 2 and stride 2. This was done to make the network less dependent on the order of the input coordinates, as the convolution would make the permutation of the two coordinates matter less. Then the output of the convolutional layer was flattened and connected to a couple of dense layers. The model was trained for a maximum of 1000 epochs, with an early stopping metric with patience 20. The model used a binary cross entropy [5] loss function, and used Adam [2] with a learning rate of 10^{-4} as its optimizer.

Then to group the input coordinates by their track, we can go over each coordinate and compare them to other coordinates. More specifically, we assign the first coordinate to the first group, then if the second coordinate is similar enough (according to a threshold value that can be specified) it will be put into the same group, if not then it will be put into a new group. The third coordinate is then compared to another coordinate from each existing group and is added to the group with which it has the highest score (as long as that score is above the threshold). This process continues for all input values until they are all part of a track.

This algorithm works quite well, as can be seen in FIG 4. The error of the network is quite low, and the grouping works very well. The network struggles to differentiate tracks that are very close together, that is tracks that have a very similar angle, but this is not that big of a problem as in the real world problem the particles are expected to be well separated [1].

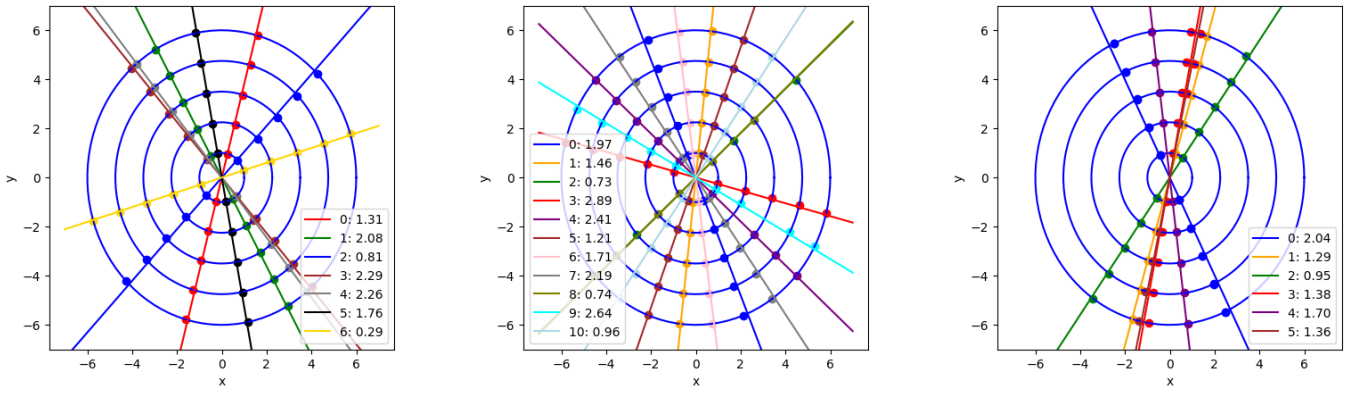
Then to go from the grouped coordinates per track to the track parameters, we need to use another model. While one can in theory just use a linear fit, or the average of a set of arctangents, we will not be using either of these methods for this letter. This was done because in the real-world problem, the lines are not straight, and this would therefore not be a very good representation of the problem if we assumed this to be the case here. Therefore we just assume that the track only depends on a single variable, but we cannot tell the model how this track depends on it, except for the fact that we are looking for an angle as an output.

The model that I created to predict the track parameter used a convolutional neural network as a base again. This particular structure was chosen to deal with the variable input length of the model. The model needed to be able to handle any number of (x,y) coordinates as an input, and give the track parameter as an output.

A 1D convolutional layer with size and stride 2 was used again at the head of the network to learn the (x,y) coordinates without being dependent on their exact ordering. Then two dense layers transformed the network into the output layer of size (n, 1). It has to be noted that since the input was of size (n, 2), with n being an unknown value, the output needed to be of size (n,1) instead of just being a single value. This was dealt with in the Predictor part of the model as described later in the letter.

The model was trained on a set of training data of size 50000, with a random number of input coordinates. To be specific the chance that each input coordinate was missing in the training set was 0.01. The model was trained with the Adam optimizer with a learning rate of 10^{-4}

At first, the network was trained using the mean squared error loss, but since the output of the network were angles I instead implemented a cyclic part to the error function. This meant that instead of just calculating the MSE between the true value and the predicted value, the loss function first looked if the predicted value was above pi, if so it would subtract pi from the value. The advantage of using this instead of the normal MSE is that the network should become better at predicting values close to the maximum angles. A prediction of π with



(a) (b) (c)

Figure 6: Some predictions done by the predictor algorithm (a) The network performs generally well even when tracks are very similar. (b) When tracks are well spaced, the algorithm barely makes mistakes in classificiation, only making slight errors in the predicted angles. In this case, it misclassified two hits but ended up giving them the correct angle anyway. (c) The main error that the algorithm makes is putting hits of two very similar tracks into a single track as can be seen with the red dots here.

a true value of 0 seems quite bad for a MSE loss, but by taking the cosine value the error suddenly drops to 0 as we would want as those values are effectively the same. This seemed to have a slight increase in performance, as the validation loss decreased from 0.006 to 0.003. The results of which can be seen in FIG 5.

The angle regressor model and the classifier were then combined into a single algorithm that I will refer to as the predictor algorithm.

This predictor algorithm takes in a set of coordinates and returns the angles of any track that it could find in those coordinates. More precisely this algorithm performs the following steps. First of all it takes the input coordinates and normalizes them, such that they are in the right range for the networks to use them. Then the algorithm uses the classifier to group the coordinates into a number of tracks. For each of the coordinate lists belonging to a track, it inputs the values into the angle regressor, which then outputs the track parameter belonging to the set of coordinates. Since the angle regressor outputs multiple values, it takes the average of these values to obtain the track parameter. These values all had a standard deviation of around 0.01 to 0.03. Then the algorithm outputs the grouped coordinates, with the angles that describe the tracks that the groups belong to with the error on those angles.

FIG 6 shows three examples of outputs of the predictor algorithm. The algorithm seems to perform quite well, predicting the tracks each hit belongs to with very high accuracy. The angle regressor is also quite accurate, predicting the angles very accurately with only a small uncertainty. The main performance hickup is due to the classifier. When tracks are too close toghether, the classifier has difficulty telling them apart. Beyond that it sometimes does not classify a hit with the correct track, opting for it to become a new track instead.

Lastly the algorithm is not the fastest, scaling with $\mathcal{O}(n * t)$ where n is the number of hits and t the number of tracks. Since the number of hits is dependent on the number of tracks this scales as a factor of t^2, which means that the algorithm will be slower the more tracks one wants to predict. For the purpose of this exercise, the algorithm could predict everything in a reasonable amount of time.

I have described a method in this paper that can be used to predict the tracks of particles that result from a collision. The two methods described in this letter are quite different, and I would recommend the two-step method over the dense regression model. The two-step model worked by first classifying the coordinates into a list of groups, and then regressing the track parameters from those lists. This method has the advantage of both being more accurate, but also robust against a varying input size. All models, and the code used for this letter can be found on my GitHub page [Link] [6].

The code used in this experiment can in theory be used to determine the track parameters for a three-dimensional path as well, as the entire code is written to support this. Due to time constraints, I was not able to properly run results and document them for this letter, but a follow-up study could be done utilizing the code I have provided as a base. Beyond that, it might be interesting to properly examine whether the models could be used to predict non-straight paths as well, this should also be a matter of training the networks provided on different input data, and changing the loss function of the angle regressor.

* gido.verheijen@ru.nl

[1] S. Caron, "Exam assignment 2023," (2023).
[2] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," (2017), arXiv:1412.6980 [cs.LG].
[3] G. R. Koch (2015) "Siamese Neural Networks for One-Shot Image Recognition".
[4] J. S. Bridle, in *NATO Neurocomputing* (1989).
[5] I. J. Good, Proceedings of the IEE - Part C: Monographs **103**, 200 (1956).
[6] G. Verheijen, "Machine learning particle- and astrophycis," (2023).