Tracking by Neural Nets

Arash Jofrehei

August 14, 2015

Supervisor: Dr. Jean-Roch Vlimant

1 Project Outline

1.1 Track Reconstruction vs Machine Learning Tracking

I spent my first week studying about current track reconstructing methods.
Current methods start with two points and then for each layer loop through all possible hits to find proper hits to add to that track.
Another idea would be to use this large number of already reconstructed events and/or simulated data and train a machine on this data to find tracks given hit pixels. Training time could be long but real time tracking is really fast.
Simulation might not be as realistic as real data but tracking efficiency is 100 percent for that while by using real data we would probably be limited to current efficiency.
The fact that this approach can be a lot faster and even more efficient than current methods by using simulation data can make it a great alternative for current track reconstruction methods used in both triggering and tracking.

1.2 strategy

First we should define tracking in machine's language.
We give a neural network some questions (hit pixels) and answers (tracks) to these questions.
Neural net will train itself on this dataset.
We ask new questions and neural net answers.
Final step is translating these answers to our language and validating results.

2 Simplified Detector and Tracks

We had to start with simplified problem to see if this idea works.
Assume that detector has four flat parallel layers and each layer has 25 (5 by 5) cells. We also tried other number of cells per layer which I will explain later.
We have 4 tracks. We started with random number of tracks but then decided

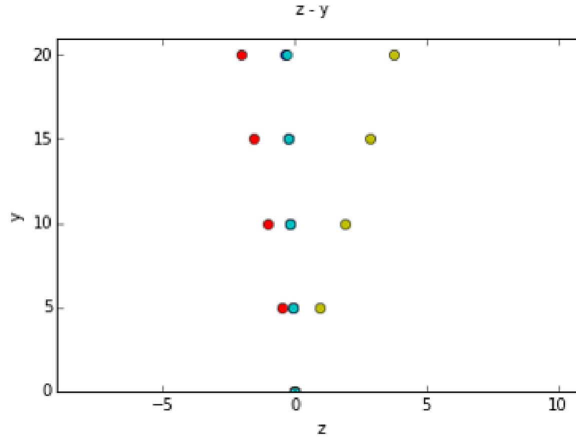

Figure 1: tracks in x axis view

to use fixed number of tracks to make this problem simpler. We chose 4 because it would lead to a reasonable hit density for 5 by 5 layers. Each track is part of a circle with almost random parameters. Track planes are perpendicular to z-y plane (z is the beam line axis). All tracks start from the origin of space, hit all layers and do not collide with each other. (figure 1 and 2)
We find hits in a way that we choose random points in 2nd and 4th layer considering limitations told above and then pass a circle through these two points and origin of space and find hits in other two layers. Then find cells that contain these points and finally check that all layers have been hit and tracks do not pass through the same cell.

3 Dataset

Input layer of neural net is a binary vector each component of which represents a cell in the detector and is 1 if the cell has been hit and 0 otherwise.
Suppose that we sort tracks in an ascending order by their radius and assign an index to each track. (indices start from 1)
Then divide these indices by number of tracks. Now for each hit the output would be a number between 0 and 1 while it's 1 if the hit is in the track with highest radius.

$$\text{output for a pixel} = \begin{cases} 0 & \text{if pixel hasn't been hit} \\ \dfrac{\text{index of track which has passed through that pixel}}{\text{number of tracks}} & \text{if pixel has been hit} \end{cases}$$

You can see input vs output plot in figure 3.

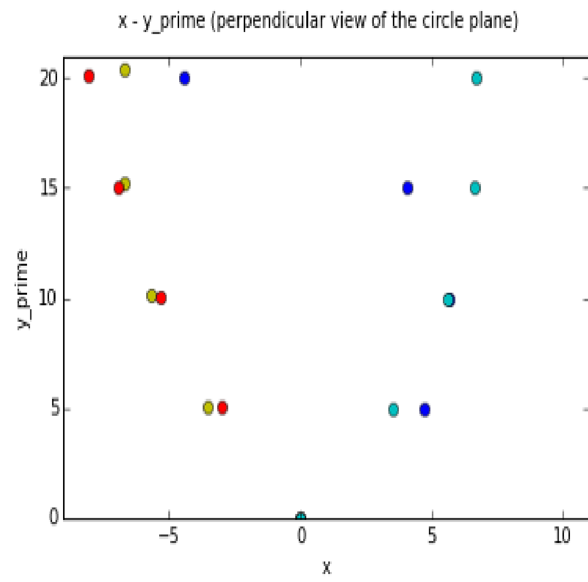

Figure 2: perpendicular view of circle planes

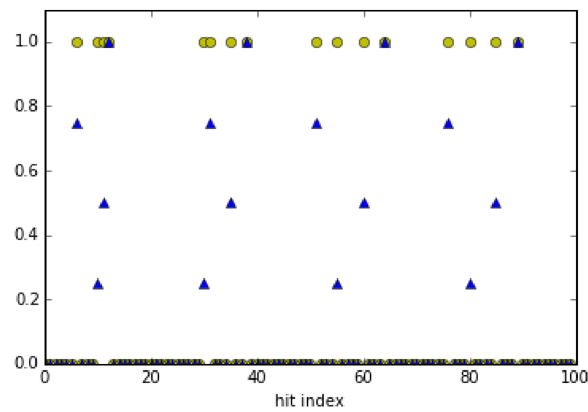

Figure 3: input (yellow circles) vs output (blue triangles)

4 Libraries

4.1 Pybrain

PyBrain[1] is a modular machine learning library for Python. I trained neural nets with it during 2nd, 3rd and 4th week. It's a pretty good library for starters but accessing its training parameters is usually impossible or hard. As these parameters have a vital effect on speed and accuracy of neural network results and should be tuned for each network, I had to move on to another library.
Another reason to leave PyBrain was that it doesn't support GPU while using GPU would be necessary for complicated neural networks with long training times.

4.2 Theanets

Theanets[2] is another machine learning library. Unlike PyBrain, there are a lot of training algorithms and parameters which should be tuned for each network.It's also built on Theano[3] so supports GPU.
Theanets uses downhill[4], a library to optimize functions (cost function in this case) with lots of options on algorithms and their parameters.

5 Algorithms and Their Parameters

5.1 General Parameters

learning rate : This is almost the most important parameter. It represents step sizes while approaching toward minimums and varied between 0.00001 to 0.1 for different networks and algorithms that I worked with.
Here are some important facts that I figured out while tuning learning rate:
Constant step size usually doesn't work. You have to start with relatively larger step sizes to avoid local minima (This model has a lot of local minima about which I'll talk later) and after finding the best valley, you have to decrease step size to reach its minimum.
By looking through some neural net codes and contacting other coders, I figured out that sometimes experienced coders find algorithms for decreasing step size based on their special network but neither I was that experienced nor we had any clue about topology or shape of cost function of our network. Usually no one knows exactly how cost function shape is and people use some general algorithms. There are available algorithms which decrease step size themselves based on an initial learning rate and a history of how cost function has reacted after each step in a way that they start with large steps and decrease step size near minima.

[1] http://pybrain.org
[2] http://theanets.readthedocs.org/en/stable/quickstart.html
[3] http://deeplearning.net/software/theano/
[4] http://downhill.readthedocs.org/en/stable/guide.html

Even by using these algorithms, we need to decrease learning rate manually, too. The reason behind that is that algorithms decrease step size near minimums (more flat areas) and the closer you get the smaller step size tends to be but you might never get that close to a minimum by a step size to make the algorithm decrease it so you have to decrease it manually. I figured out that the best time to decrease learning rate is when cost function starts to be still because if I decrease it too soon, it would increase training time a lot and more importantly, increases the probability of getting stuck in a local minimum and if I don't decrease learning rate when cost function starts to converge, usually after a very long time it reaches it's minimum while I could reach there a lot sooner by decreasing learning rate.
In Theano and Theanets, learning rate is stored as a constant as the initial step size and algorithm adapts step sizes during training and after changing learning rate, it re-initializes step size to new given learning rate. This is why we have to be careful when decreasing learning rate as we may even increase step size by decreasing learning rate. I usually decrease learning rate 2 or 3 times during training process.

momentum : In each step of training a neural network, we update network parameters (weights) by subtracting a number proportional to step size and gradient of cost function with respect to weights from each weight (Different algorithms use different approaches but the main idea is as told above). By using momentum, we also divide weights by a certain factor (momentum) in each step which prevents noises and long jumpy steps. The problem though, is that if I use momentum from the starting point, it slows down training process a lot as we are too far from minimums at first and we actually need those big jumps. The good thing about momentum compared to learning rate is that changing momentum improves results even after full convergence so I decided to use a momentum about 0.9 in final part of training process.

dropout : By using dropout for a layer in network, in each updating step, we kill each neuron in that layer with a probability of dropout value. Killing **n** neurons means that we ignore those neurons in that step and update other neurons exactly like that layer had **n** less neurons.
Using dropout helps us to prevent over-fitting, a condition when we train on our training samples too much that we lose the generalization of results and network gives worse results on validation samples. The best and logical way to prevent over-fitting is increasing samples but for large networks I needed relatively more samples which were impossible to store so I used a dropout about 0.8 to 0.9 for hidden layers. I should mention that we cannot use small values for dropout as killing a lot of neurons at each step will somehow make a training process meaningless.

batch and epoch parameters : During each iteration, the optimizer instance processes training data in small pieces (**batch-size**) called mini-batches. Each mini-batch is used to compute a gradient estimate for cost function, and the

parameters are updated by a small amount. In each epoch, a fixed number of mini-batches (**validation-size**) are processed. After a fixed number of epochs (**validate-every**) have taken place, the cost is then evaluated using a fixed number of mini-batches from the validation dataset.
Optimization epochs continue to occur, with occasional validations, until the loss on the validation dataset fails to make sufficient progress (more than **min-improvement** percent of lost) for long enough(**patience** times). Optimization halts at that point. [5] (bold parameters have to be set when defining a trainer)

Leif Johnson [6] : "I think of full batch vs mini-batch as a trade-off between time and accuracy. With a full batch, you spend more time computing an accurate (at least, the most accurate that you can get with your data) estimate of the gradient. With a mini-batch, you spend less time but get a noisier gradient estimate."

Andrej Karpathy [7] : "Usually you want to use batch size of 1. This basically controls how accurate the gradient steps of your network will be. If you let the network see 100 examples in a batch, it will be able to estimate a much better value for gradient before it actually takes the step. However, in practice a value of 1 (and having an appropriately small learning rate) is probably the best way to go."

What I understood by changing batch size:
By using a large batch size you increase stability of updating steps and each step takes relatively more time as we have to compute gradients for more samples and then take a step proportional to average gradient. While using small batch sizes, weights and therefore cost function change in a noisy way and to control this noisy behavior, we have to decrease learning rate (even by a factor of 100 in some cases). Although large batches cause greater training time for each epoch, overall training time is much more longer while using small batches because we waste a lot of steps jumping around in a noisy way and more importantly, we would have smaller step sizes.
A more important problem with using small batches is that using small step sizes sometimes leads to getting stuck in local minima.
Although if we solve local minimum and training time problem, using small batches will lead to better final results because we would use the full capacity of our samples individually and independently.
Using batch sizes that are a multiple of processor's warp size (32 for my case) will speed up training process a lot as matrix operations are defined in a way that will be faster this way.
As I said above to improve final results we need to see each sample individually and independently but too small batches cause problems. I have experienced

[5] http://downhill.readthedocs.org/en/stable/guide.html#batches-epochs
[6] Computer Science doctoral student at The University of Texas at Austin and a contributer of Theanets
[7] Stanford Computer Science Ph.D. student

that we'd better set batch size and dataset size as co-primes so while looping through batches, we would never have same batches and this means we will have both stability and independence. Another approach is shuffling samples after each epoch which worked a little better than co-prime approach. I used this in batch-loading approach which I'll explain later.
In conclusion, I decided to use larger batches (1024 for example) first and after getting closer to minimum, I decrease batch size to 32 or 64 and I also use approaches that I talked about above to avoid local minima and also see each sample independently.

5.2 Algorithms

Theanets uses downhill library to optimize cost function. You can find theories behind all algorithms and expected parameters for each algorithm on downhill site[8]. Pybrain only uses **Stochastic Gradient Decent (sgd)** as a simple and basic optimization method but sgd is neither accurate nor fast and sgd is not common these days anymore unless for some special networks. I tried almost all algorithms, but found **resilient backpropagation (rprop)** , **rmsprop** and **adadelta** the best. Steps in these methods are determined by the history of optimization as I described before in learning rate section. adadelta doesn't have a learning rate and uses an alternative step size. Although it has shown better performance in some image processing neural networks like CIFAR-10, the fact that I couldn't have access to step sizes for each epoch made adadelta inappropriate for my case. For simple neural networks, rmsprop and rprop had almost the same performance unless rprop was a lot faster and rmsprop has two more parameters to tune which makes it hard to reach the same performance as rprop. For more complicated neural nets, rmsprop stopped working sufficiently. After tuning all parameters for a few networks, rmsprop results got relatively better but still not as good as rprop so I decided to use rprop.

I think it would worth mentioning that the main challenging problem in this project is tuning parameters specially for larger networks. Because performance gets much more sensible to parameters while using large networks and training time increases which again, makes it harder to tune parameters. Usually I try to find best parameters for small and medium networks and then either look for tuned parameters around these values for large networks or just use the same values if network is too large and untunable.

6 Activation Functions

There are a lot of activation functions available for Theanets.[9]
The best activation function for output layer was sigmoid(logistic) which wasn't hard to guess as we expect outputs to be between 0 and 1.

[8]http://downhill.readthedocs.org/en/stable/guide.html#optimization-algorithms
[9]http://theanets.readthedocs.org/en/stable/creating.html

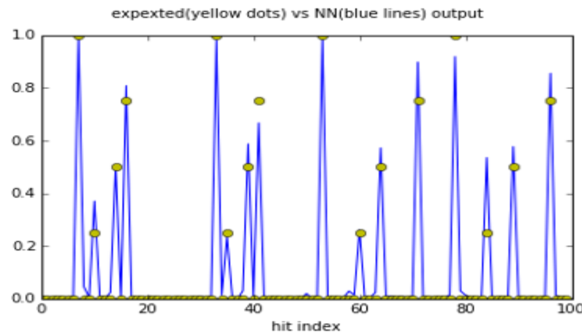

Figure 4: expected(yellow dots) vs neural network(blue lines) output

For hidden layer(s) the best ones were relu and sigmoid. Usually, wherever relu works fine, maxout should make it better because maxout acts exactly like relu unless it assigns more free parameters (weights) to each neuron but it couldn't improve results and even made it worse in some cases.
The problem with relu is that it acts like linear for positive inputs and doesn't have an upper limit on its outputs and more importantly it's not continuous and is 0 for all negative inputs. This discreteness somehow kills some neurons while training. Reacting in a same way to two or more different neurons can be the same as keeping one and ignore others. This problem makes cost function to get stuck in local minima specially for small networks. I tried both relu and sigmoid for XOR gate problem and figured out that there's no way to avoid local minima (XOR has a lot of local minima) while using relu for hidden and output layer. For my case cost function decreases much more smoother (avoids local minima) while using sigmoid (although this network still has a lot of local minima) so I decided to use sigmoid.

7 Assembling

Neural network output is a vector with float components between 0 and 1 (figure 4). We need to translate this output to see how it has assembled hits to form tracks. Almost the inverse procedure that we created expected output from tracks.
I had two main approaches and worked on them to make them faster and also more compatible to next step which is validation. I should mention that all assembling approaches work the same when neural network output gets close to expected output.

Sorting : I iteratively loop through all detector layers and assemble maximum outputs together.
For example for a detector with 4 layers and 25 cells per layer, I divide the 100 dimensional neural network output into 4 parts (1 to 25, 26 to 50, 51 to 75, 76

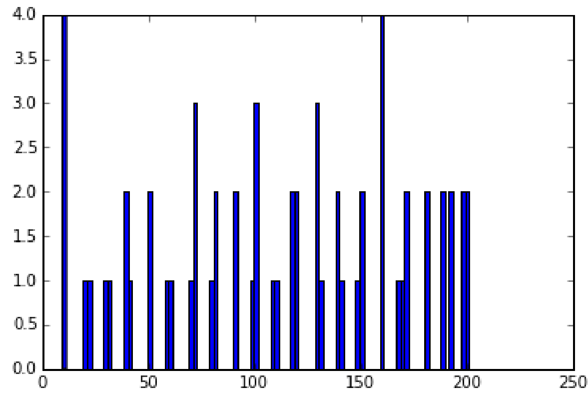

Figure 5: Histogram of a clustering input sample. It's not our neural net output, just a sample to show how clustering algorithm works. It was created in a way that it should have 20 clusters with random number of inputs in each cluster

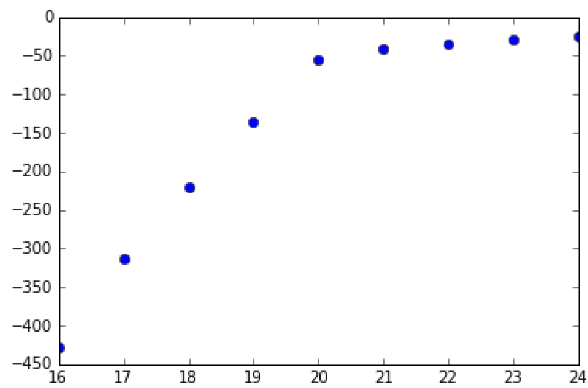

Figure 6: Clustering scores for different number of clusters. Notice that it's found 20 as breaking point as was expected for clustering input in figure 5

to 100) and then find maximum outputs for each part. These four hits then would be considered as hits of a track. Then I do the same thing for second highest outputs and so on.

Advantage : Sometimes network cannot separate outputs well enough in (0,1) interval (needed for clustering approach) but usually keeps the arrangement for each layer.

Clustering : Network outputs are clustered to unknown number of clusters which represents number of tracks and then hits in each cluster would be assembled together.
Clustering algorithm: Given the network output vector, first I put a low cut on outputs (0.1 or 0.05 works fine) then save all outputs in a matrix each row of which is [hit index of output , output value] and sort this matrix by it's second column. Now I have to cluster the second column but there was no available clustering algorithm to cluster into unknown number of clusters so I found a way which works fine at least for this one dimensional clustering problem.
Given the outputs (second column of matrix) I calculate differences between each output and its next output. Then I cluster these differences into two clusters, small differences which show close outputs and large differences which represent a gap between two output clusters. Now expected number of output clusters (number of tracks) would be number of gaps plus one.
Although, we have to make sure that we have found the right number of tracks. To do this, I cluster outputs to other number of clusters in a range around found number. Clustering algorithms return a clustering score which is a chi-square like value. This value would increase by increasing number of clusters but after plotting these scores for those different number of clusters, you can see that it has a breaking point after which plot gets more flat which means clustering inputs have been to separated (figure 5 and 6). So that breaking point would be our revised number of clusters. To find that breaking point I cluster differences between each score and next one into two groups and accept the separating point of these two groups as breaking point. After finding this number, we have to cluster neural network outputs into this number of clusters and keep all hits in a cluster as a track.

Another clustering approach that I tried, used the fact that neural network outputs should be symmetric and clustered them in a way that we keep this symmetry but it was much more slower than the other approach so we decided not to use it.
One problem with our clustering method is that sometimes it merges close tracks so we have tracks with more hits than number of layers but I finally decided to ignore this problem because first, we wanted this clustering approach to work for most general case in which we may even have tracks with more hits than number of layers due to noise and second, each approach that I tried to fix this problem would destroy generalization of clustering method in a different way while better networks can separate outputs well enough and I had sorting

288 method for worse networks. Still, improving this approach can be one of next
289 steps on this project. I should mention that goodness of hits which I'll talk
290 about it in next section uses only clustering assembling so reported results for
291 that would vary by improving this assembling approach.
292
293 Advantage : In general, we don't know how many tracks we have and more
294 importantly, this method works even if a particle doesn't hit some layers.
295

296 # 8 Validation

297 The whole idea of validating network results is as follows. A **good hit** is a hit
298 which has been found correctly. A **good track** is a track for which a certain
299 ratio of hits have been found correctly. A **good event** is an event with certain
300 ratio of good tracks.
301 After assembling hits I save all hits separately (in a matrix each row of which
302 contains hits of a certain track). At first I used to compare tracks in a way that I
303 iteratively compared a track with highest energy from expected tracks with the
304 highest energy track from neural network tracks but that's not actually what
305 we want. Neural network doesn't have to find energy index of a track correctly,
306 it has to just assemble hits correctly and then while fitting a trajectory through
307 hits, we will find track's energy. Results with the first approach are reported
308 with an "old" label. In the new approach, for each event, after saving all hits in
309 those matrices, for each track in expected output, I loop through all neural net
310 tracks to find a track that completely matches with that and then delete that
311 neural net track, then I do the same thing but accept one mistake and so on
312 and meanwhile, I count good hits and also good tracks. I do this for all events
313 and then calculate an average value for event, track and hit efficiency but keep
314 all the data and don't replace any data by its average value (I though we may
315 need them for comparison at next steps).

316 # 9 Models and Performances

317 I started working on a wide range of models from second week. Our neural net is
318 feed forward and fully connected. During 2nd, 3rd and middle 4th week we saw
319 exactly no sign of any promising results which wasn't too strange as no similar
320 neural networks was available to learn from so we had no idea about which
321 topology, activation functions, dataset size and ... to use. A short summary of
322 what I did in those 3 weeks is as follows:
323 At first we had 10 hidden layers while each layer had 100 neurons (like input
324 and output). Detector layers were 100 by 100 (representing a 10 cm by 10 cm
325 piece of real detector). Each event had a random number of tracks between
326 1000 and 4000. Dataset size was roughly between 100 to 1000 while 70 percent
327 of samples were used for training and the rest for testing while training and also

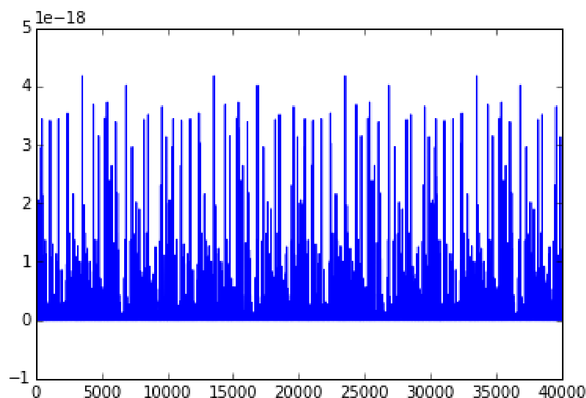

Figure 7: a typical neural net result achieved for 100 by 100 layer and random number of tracks between 1000 and 4000 - notice that neural net output is almost zero for all cells

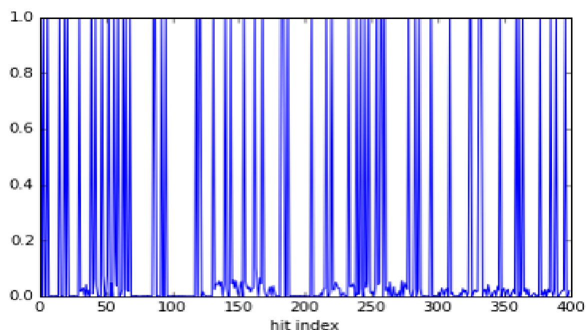

Figure 8: started to get promising results by reducing module size and number of tracks

328 final validation. (figure 7)
329 After failing to achieve any promising results, we decided to make this problem
330 simpler by reducing module size from 100 to 20 and then 10, using fixed number
331 of tracks and using less tracks (4 tracks) to have a more reasonable hit density in
332 detector layers. Neural network predictions started to change from completely
333 random numbers and move toward expected shapes but still it was far from any
334 promising result.(figure 8)
335 From late 4th week I started to use Theanets. Training got a lot faster by
336 using different algorithms and also using GPU (about 500 times faster which
337 was mostly a result of using algorithms like rprop). I also figured out that
338 dataset size should roughly be something between 10 to 30 times number of free
339 parameters in model so I had a too small dataset. I also decided to simplify
340 topology of model to reduce training time and also number of free parameters

341 and therefore needed dataset size. So I tried 1 to 6 hidden layers each with the
342 same size as input and output and also reduced module size from 10 to 5 (5 by
343 5 layers).
344 By looking at figures 7 and 8, it's obvious that neural network predictions were
345 too bad to use any assembling or validation method (any figure of merit would
346 be 0) so I used to compare cost functions in first 4 weeks but by simplifying
347 topology and using theanets we started to get better results so I needed to
348 develop those assembling and validation methods I talked about in previous
349 sections.
350 I also had some problems while using theanets with GPU (monitoring cost func-
351 tion while training, passing some training parameters used to cause problems,
352 results behaved a lot different with CPU and GPU and ...). It took me 2 weeks
353 until I could use it completely and efficiently. I also figured out all those facts
354 about algorithms and their parameters and activation functions and ... during
355 5th to 7th week.
356 From late 5th week, we tried to increase number of neurons per hidden layer.
357 While using fixed number of hidden layers (we tried 1 hidden layer) increas-
358 ing number of neurons improved results. Adding more hidden layers improves
359 results, too. By comparing training time and performance, I figured out that
360 it's better to use more neurons per each hidden layer than input and output
361 but not too much because after a certain number of neurons, adding another
362 layer would be much more efficient. Although we didn't know these fact until
363 the 7th week because at first, adding more free parameters (either by adding
364 layers or neurons per layer or ...) to neural network would lead to worse results.
365 The problem was that as I talked about it before, for larger neural nets tuning
366 parameters tends to be a really hard task and we need more training samples
367 and more importantly, figuring out if cost function has converged to something
368 or we have to wait more becomes really challenging about which I will talk later.
369 You can find best track performances achieved by using different number of hid-
370 den layers and neurons per layer in figures 9 and 10.
371
372 As complete convergence of a model usually takes a long time, sometimes I stop
373 training process when there's no sign of converging to a good efficiency and
374 that's one of the reasons that almost half of models that I tried are not reported
375 in figure 12. Although they mostly have the same topologies but with different
376 parameters and activation functions.
377
378 A summary of final results:
379 For more accurate models with track efficiency more than 48 percent, about
380 81 percent of hits have been found correctly which is 13 out of 16 hits and it
381 doesn't change a lot when track efficiency varies in this range.
382 51 to 54 percent of tracks are fully reconstructed for 5 by 5 models.
383 About 88 percent of tracks are reconstructed with one mistake.
384 About 30 percent of events are fully reconstructed.
385 Notice that by each mistake in assembling hits, we lose 2 out of 16 hits and 2
386 out of 4 tracks so on average, neural network has about 1 to 1.5 mistakes.

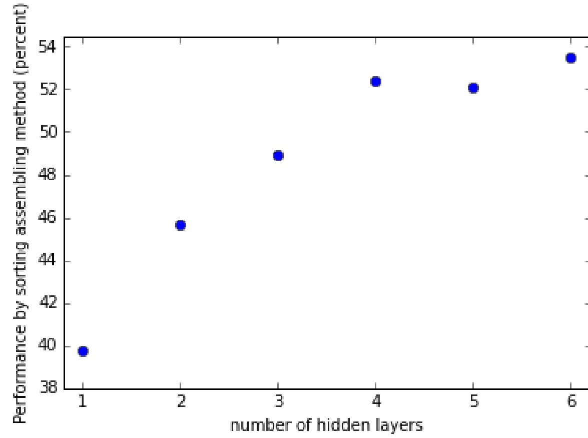

Figure 9: best track efficiencies achieved using sorting assembling method (percent); detector layers are 5 by 5; each hidden layer has 100 neurons as input and output; each value has a maximum error bar about 1 to 2 percent due to probable early stopping or small validation dataset

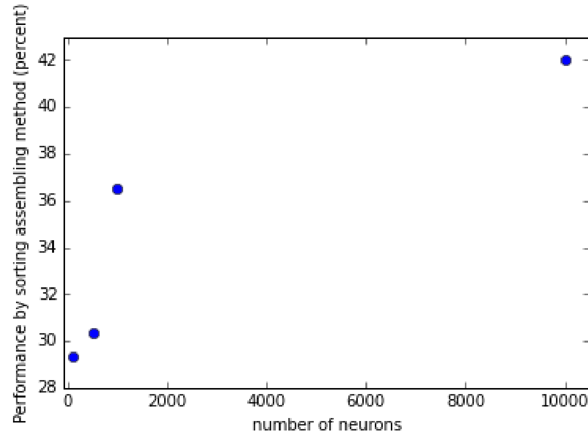

Figure 10: best track efficiencies achieved using sorting assembling method (percent); detector layers are 5 by 5; used one hidden layer

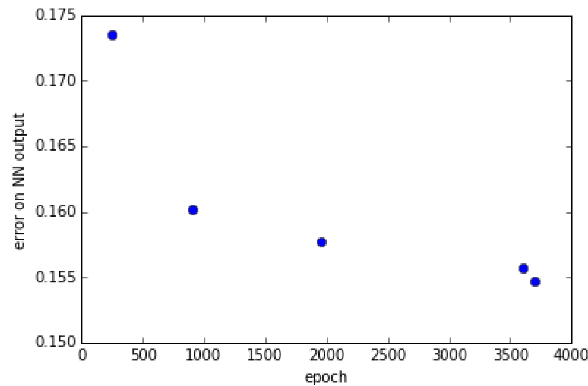

Figure 11: error on neural network results for 4 tracks case where expected outputs are 0.25, 0.5, 0.75 and 1; You can see how this error converges to around 0.15 for a model with 6 hidden layer, each with 100 neurons and 5 by 5 detector layers

387 10 by 10 models have a lot better performances compared to 5 by 5 models
388 with similar topologies but I couldn't try large neural networks for them due to
389 memory errors and these reported models were trained for about 15 hours but
390 cost function was still decreasing for them so they need more training. I'll talk
391 about these two issues later.

392 10 Observations

393 I should emphasize on two important facts that I encountered during training
394 these models: Early stopping and local minima
395 Cost function keeps decreasing after each training epoch but after a while it
396 changes too slow and by considering the fact that training time for each epoch
397 for models that I worked with in this project were relatively long a challenging
398 part was deciding when to stop training process. One way was using min-
399 improvement that I talked about before but this approach doesn't work well
400 when cost function doesn't have a smooth converging behavior. By looking at
401 figures 13 and 14 you can see that cost function seems to converge at first but
402 after a lot of epochs it decreases by 0.5 which is a really vital value.
403 As I mentioned before, there were a lot of local minima in this neural net (fig-
404 ures 15 and 16). I looked through literature and figured out that every local
405 minimum can be smoothed by adding enough training samples but still one chal-
406 lenging problem that coders encounter to is flat areas which cannot be solved
407 completely by adding samples and also I had limitations on dataset size. So I
408 tried to study this problem in XOR problem which as I told before, has a lot of
409 local minima. Simplicity of XOR makes it too fast and it's easy to try different
410 ideas on it. I mentioned some methods to avoid local minima like using learn-

Sorting (new)							
5C	20K	50K	100K	200 K	300K	500K	700K
1L	28.12	29.3	39.76	_	_	_	_
2L	_	25	_	45.7	_	_	_
3L	_	_	_	_	48.91	_	_
4L	_	_	_	_	_	52.4	_
5L	_	_	_	_	_	52.1	_
6L	_	_	_	_	_	_	53.51

Clustering							
5C	20K	50K	100K	200K	300K	500K	700K
1L	30	30.1	40	_	_	_	_
2L	_	28	_	46.85	_	_	_
3L	_	_	_	_	48.6	_	_
4L	_	_	_	_	_	51.4	_
5L	_	_	_	_	_	54	_
6L	_	_	_	_	_	_	52.7

5C-1L-1000N				
	maxout-50K	sigmoid-50K	relu-100K	sigmoid-100K
sorting (old)	24.25	28.12	_	_
sorting (new)	_	33.25	32.87	36
clustering	32.25	33.8	35.75	41

5C-2L-100-1000N-100K	
sorting (new)	clustering
31.87	33

5C-1L-500N-100K	
sorting (new)	clustering
30.37	_

5C-1L-10000N-50K	
sorting (old)	clustering
26.2	33.5

5C-1L-10000N-500K	
sorting (new)	clustering
42	40.13

10C-1L-1000N-100K	
sorting (new)	clustering
43	44.87

10C-1L-2000N-200K	
sorting (new)	clustering
47.87	50.625

10C-1L-2000N-400K	
sorting (new)	clustering
49.25	51.875

10C-1L-4000N-500K	
sorting (new)	clustering
50	52.75

10C-2L-2000N-1M	
sorting (new)	clustering
49.3	50.7

10C-3L-2000N-1M	
sorting (new)	clustering
57	54.2

15C-1L-2000N-130K	
sorting (new)	clustering
35.5	37.5

15C-1L-3000N-200K	
sorting (new)	clustering
37	_

Figure 12: best achieved track efficiency for some models ; C: module cell (cells per edge) ; L: hidden layer ; N: neurons per hidden layer ; K: thousand samples ; M: million samples

411 ing rate properly or co-prime approach or shuffling dataset but finally I found
412 another approach which although requires more study in main network, works
413 better than other approaches for XOR problem. I tried to look into network
414 parameters (weights) for XOR problem with sigmoid layers (relu almost never
415 works) and see how these local minimums occur. I figured out that getting stuck
416 in a local minimum depends on initial distribution of weights a lot. For example
417 by only working on the sign of each initial weight I could completely avoid local
418 minima in XOR problem. Although this approach might not seem wise as we
419 are using results to find results (we need to train XOR once and find final signs
420 and then initialize weights with the same signs but arbitrary absolute value)
421 but sign of weights doesn't seem to be too much information and probably can
422 be obtained by a pre-trainer, too.
423 I also looked into final weights for our neural network and figured out that
424 weights for each layer have almost the same distribution. For a 5 by 5 model
425 with 6 hidden layers each with 100 neurons bias weights were some numbers
426 around 1 or 2 and neuron weights had a distribution with two peaks at 0.003
427 and -0.003 while default initial values for all weights are set by a normal dis-
428 tribution with deviation around 1 and mean 0. I set initial values manually for
429 this model and cost function converged a lot faster and smoother. Distribution
430 of weights for the same model with 5 layers was almost the same, too. Although
431 I worked on this method on my 8th (last) week and didn't have time to look
432 into it with more details. One of next steps can be looking into weights while
433 training and investigate their behavior and also compare weight distributions
434 for different topologies.
435

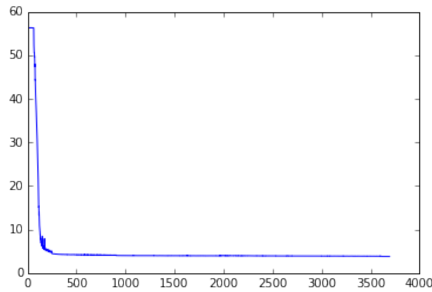

Figure 13: cost function on validation data

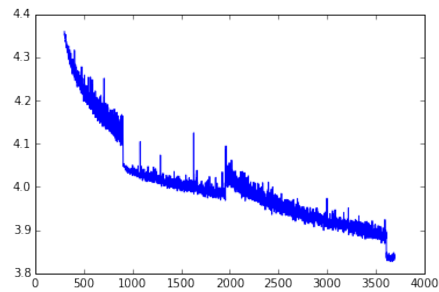

Figure 14: same plot after removing first 300 epochs

Beside these two problems, I encountered another problem while working on 10 by 10 models with 4 tracks. After a certain point, hit and track efficiency were almost constant while cost function was still decreasing. As cost function was decreasing on both training and validating data, it couldn't be an over-fitting problem so most probably definition of cost function had some problems. Current definition is as follows:

$$\text{cost function} = \frac{\sum_{\text{output vector components}} (\text{expected output} - \text{neural net output})^2}{\text{output vector dimension}}$$

I figured out that for latest models, neural networks learns to set zeros in input to values close to zero in output really fast and then tunes nonzero values. It's possible that trainer reduces cost function by reducing those values near zero after a certain point and although it's not noticeable for each component, we have a lot zeros in input which would make a noticeable overall difference in cost function which means trainer is wasting a lot of steps. We can modify cost function after a while to make it less sensible to zeros or modify activation function of output layer by adding a below cut on outputs and considering all values near zero as zero (so training steps wouldn't waste any step for tuning them) to solve this problem. I talked about this with Leif Johnson and he also believed that modifying cost function in the middle of training can be useful in my case.

We can also modify cost function to make it less sensible to inner layers of detector because hits are mostly restricted to center area of these layers so trainer has relatively more dataset to train inner areas of inner layers while for higher layers, hits have been spread all over the surface of layers. This can also be seen by plotting neural net outputs so we can see that on average, we have more accurate results for inner layers of detector.

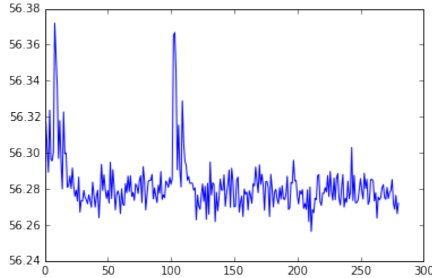

Figure 15: Cost function starts from 300 and seems to converge to 56.3; I have removed first epochs to see that cost function is almost constant even when we zoom in

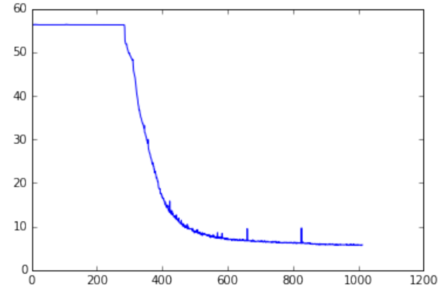

Figure 16: cost function falls down suddenly while it seemed to be constant; I waited this long only because I knew that cost function should be a lot less; it's possible that these final values that I report are local minimums, too but as I've waited for some models for a really long time local minimum at final point has a small probability

11 More Pixels

For 10 by 10, 15 by 15, 20 by 20, layers, data size becomes too large because dimension of input and output increases so we need hidden layers with more neurons and therefore we need more training samples and because of the increase in input and output dimension, each sample will be larger, too. GPU cannot store this amount of data while training anymore.
If we use simple models, we would need less samples (still 3GB for only 1 million samples) but still training time is too much. To solve this timing issue, we can let a simple 5 by 5 model train completely until cost function reaches it's minimum. Then we use this converging function (cost function per epoch for example) and fit it to the piece of curve that we have for complicated model to predict its final results.
Still, to achieve accurate enough neural network predictions, we will eventually need to use more complicated topologies and therefore more samples. I developed a way to save all samples in an external file and for each training epoch, I load a random batch from those samples and train on that which solved the memory issue.
Although I didn't have time to work on fitting method and it can be one of the most important next steps of the project.

As I mentioned before, 10 by 10 models showed better performances with similar topologies even though they were not trained enough and didn't have enough samples. By looking at figures 17 and 18 you can figure out why. 5 by 5 layers don't have enough resolution and hit points in a relatively large area will be

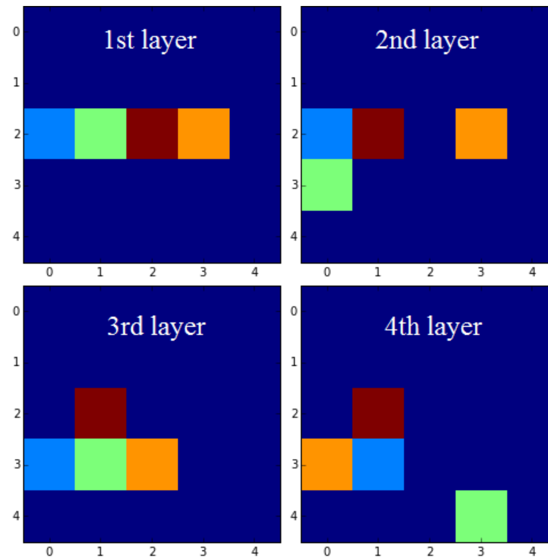

Figure 17: hits for a detector with 5 by 5 layers and with 4 tracks - hits with the same color are part of the same track

485 considered as the same cell which makes it really hard to see track curves and
486 trajectories. Another reason for that might be that hit density has decreased
487 in 10 by 10 case but still even with the same density I suppose that better
488 resolutions will work better and performance is a function of density, resolution
489 and also ratio of layer size on distance between layers and we have to find this
490 function.
491 Low resolution might also lead to a degeneracy in a way that one input can
492 have different outputs. I started to look for this degeneracy but it's really hard
493 to find two same inputs to see if their output are same or not as we have a very
494 large number of combinations for hits in layers which will lead to a large number
495 of different inputs but probably inputs don't have to be completely the same to
496 cause this degeneracy. This problem needs more investigation in details.

497 12 Conclusion and Outlook

498 After searching for best topologies, algorithms, parameters, etc, we could finally
499 train a neural network on this dataset and reconstructed tracks with 1 to 1.5
500 mistakes on average while by considering resolution of detector layers, is most
501 probably the highest achievable performance.
502 To move on to larger models, we needed new methods some of which we have
503 developed already.
504 We have to find convergence point of a cost function without waiting for full
505 convergence.

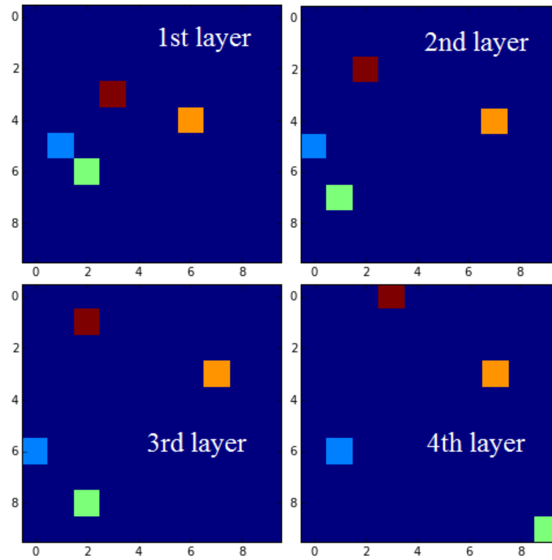

Figure 18: hits for a detector with 10 by 10 layers and with 4 tracks - hits with the same color are part of the same track

506 Work more on initial values for weights.
507 Modify cost function or activation functions to make them compatible with this
508 dataset.
509 Batch-wise loading method and convergence point method will help us to use
510 layers with higher resolution which seem to have better performances.
511 We have to try more tracks because we need a reasonable hit density for large
512 detector layers and also with 4 tracks, each mistake will decrease performance
513 too much. I started to use 8 tracks for 10 by 10 models but didn't have time to
514 see the final results
515 We have to try different neural network structures like recurrent models and try
516 convolutional layers. Eventually, we have to work on more realistic datasets.
517 We have to add noises, use random number of tracks, use more realistic track
518 trajectories and so on.