




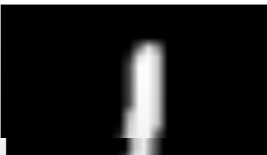


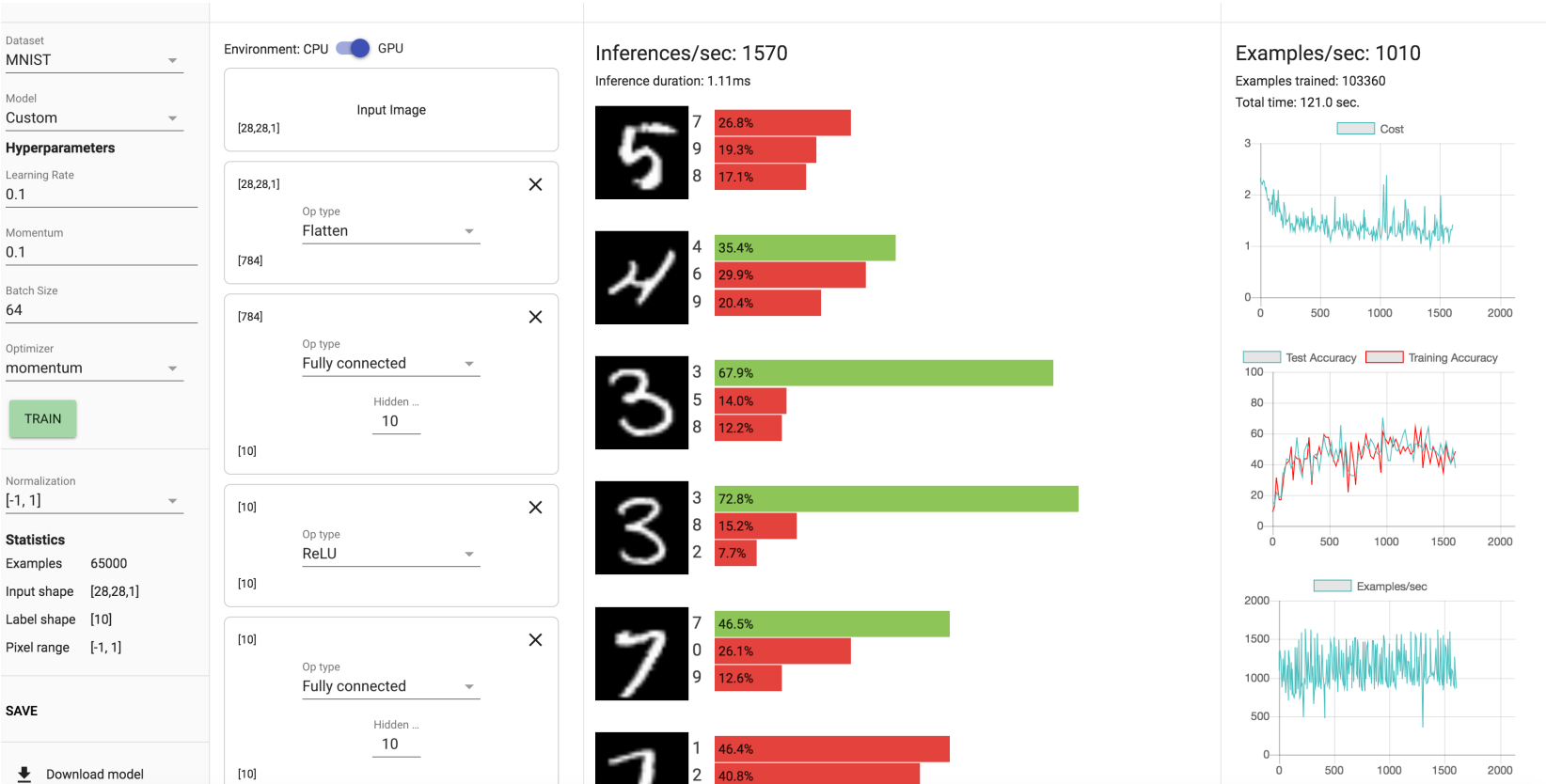
DATA	MODEL	INFERENCE
<div>Dataset MNIST</div> <div>Model Custom</div> <div>Hyperparameters</div> <div>Learning Rate 0.1</div> <div>Momentum 0.1</div> <div>Batch Size 64</div> <div>Optimizer momentum</div> <div>TRAIN</div> <div>Normalization [-1, 1]</div> <div>Statistics</div> <div>Examples65000</div> <div>Input shape[28,28,1]</div> <div>Label shape[10]</div> <div>Pixel range[-1, 1]</div> <div>SAVE</div> <div>Download model</div>	<div>Environment: CPU GPU</div> <div><div>Input Image</div><div>[28,28,1]</div></div> <div><div>[28,28,1]</div><div>Op type Flatten</div><div>[784]</div></div> <div><div>[784]</div><div>Op type Fully connected</div><div>Hidden ... 10</div><div>[10]</div></div> <div><div>[10]</div><div>Op type ReLU</div><div>[10]</div></div> <div><div>[10]</div><div>Op type Fully connected</div><div>Hidden ... 300</div><div>[300]</div></div>	<div>Inferences/sec: 1</div> <div>Inference duration: 0.993ms</div> <div><div></div><div>210.8%</div><div>810.4%</div><div>49.9%</div></div> <div><div></div><div>210.8%</div><div>810.4%</div><div>49.9%</div></div> <div><div></div><div>210.8%</div><div>810.4%</div><div>49.9%</div></div> <div><div></div><div>210.8%</div><div>810.4%</div><div>49.9%</div></div> <div><div></div><div>210.8%</div><div>810.4%</div><div>49.9%</div></div> <div><div></div><div>210.8%</div><div>810.4%</div><div>70.3%</div></div>

approximately 1600/sec with 1460 examples/ second/.

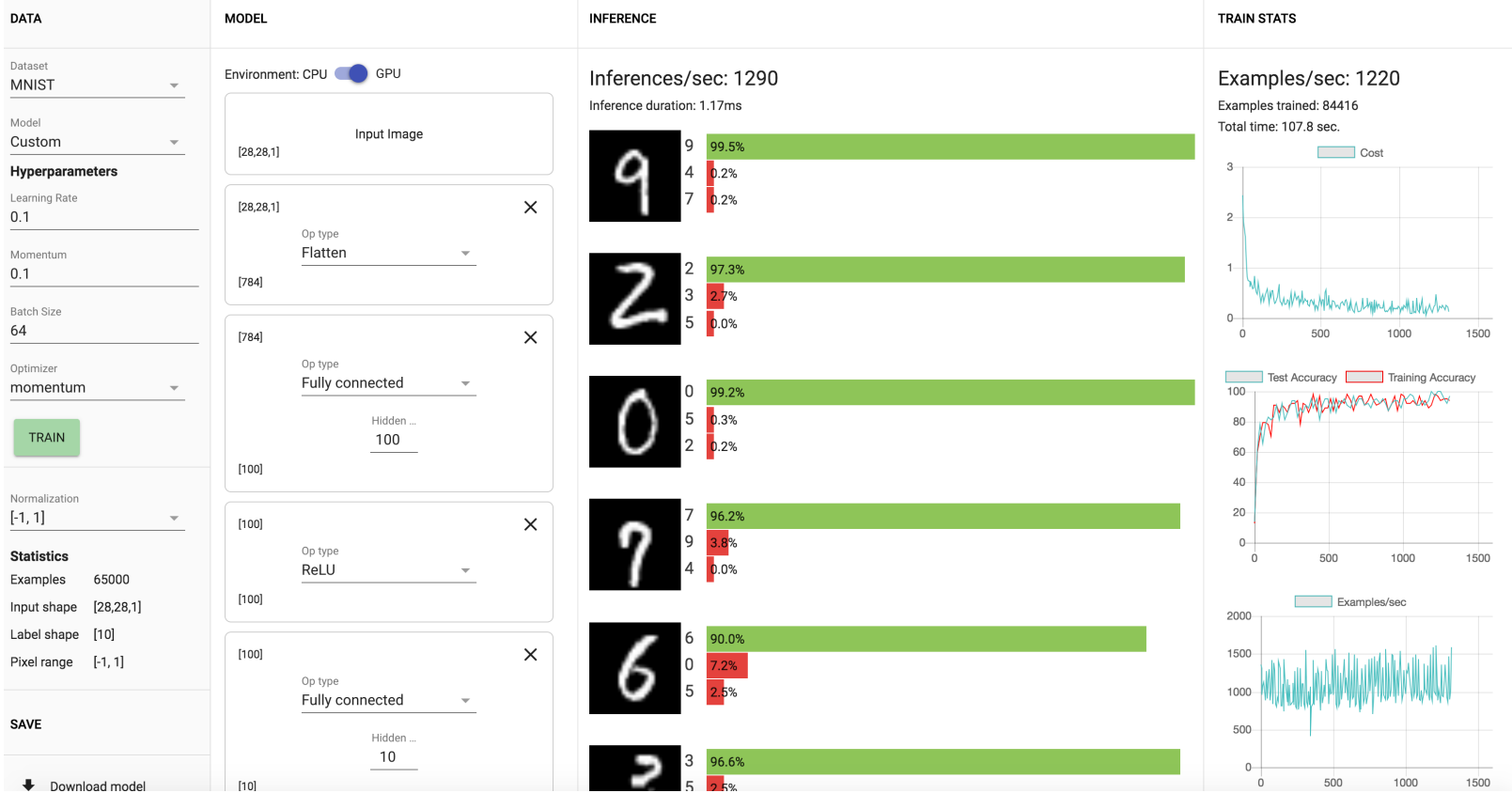
Start training and you should see the accuracy plummet to zero, with terrible results. What’s going on? Hint: Notice that many of the probabilities will print as Nan%. Document these results and write up your ideas for why this happens.

By applying multiple linearly activated layers to our NN makes it worst because no matter how many layers or neurons to our network the result will be always a linear function. Each layer would get the weighted values from the previous linear function and will calculate a weighted sum on that input - it will fire every time on another linear activation function.

Train the new model. How well does it perform? Then make the first FC model wider by increasing the number of units to 100. Does this make a difference? Document the results for these questions on your webpage.



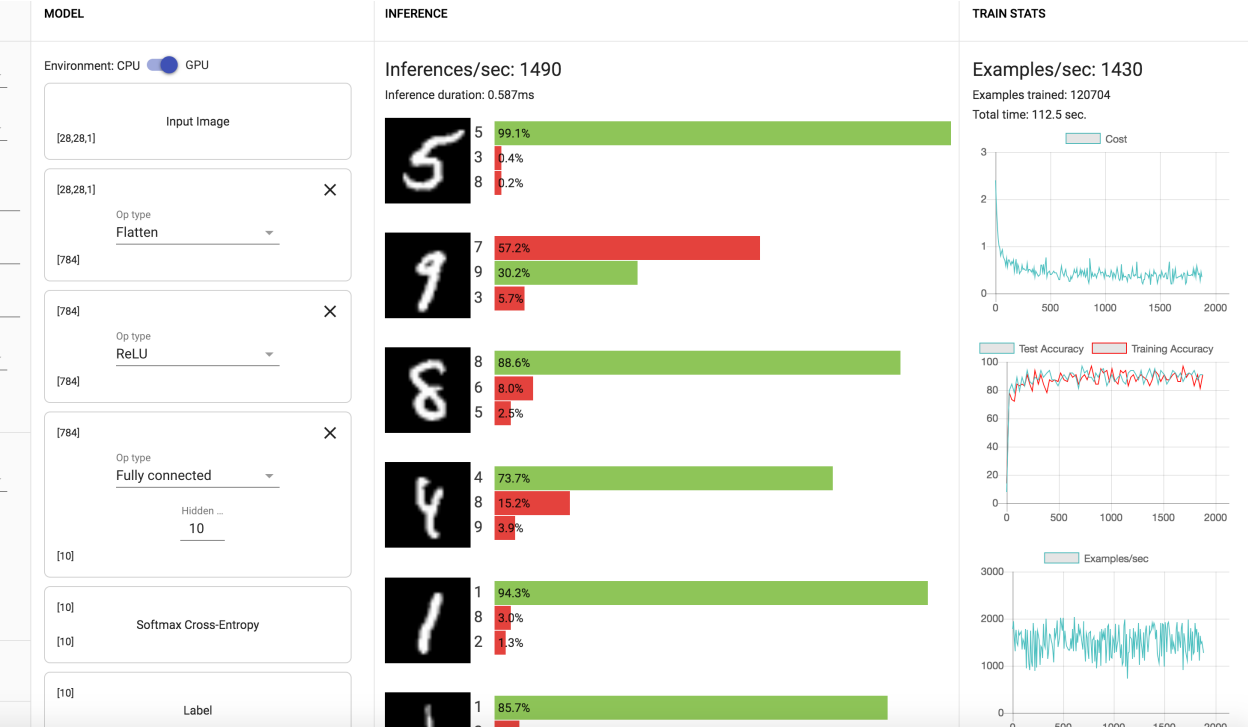
The model starts really bad at the beginning, slowly increasing the accuracy over time. The results are as low as 37% avg. I wonder if this is produced because of the RELU activating only too little amount of neurons.



The model performs better with accuracies over 98 % (2 minutes training) Introducing RELU and adding more units to the first FC layer makes a difference in terms of speed and "lightness' of the network because of its outputs producing -I guess- a sparse activation of the neurons (if some are 0 then they don't activate). Therefore is less computationally expensive due to the use of a simpler mathematical operations.

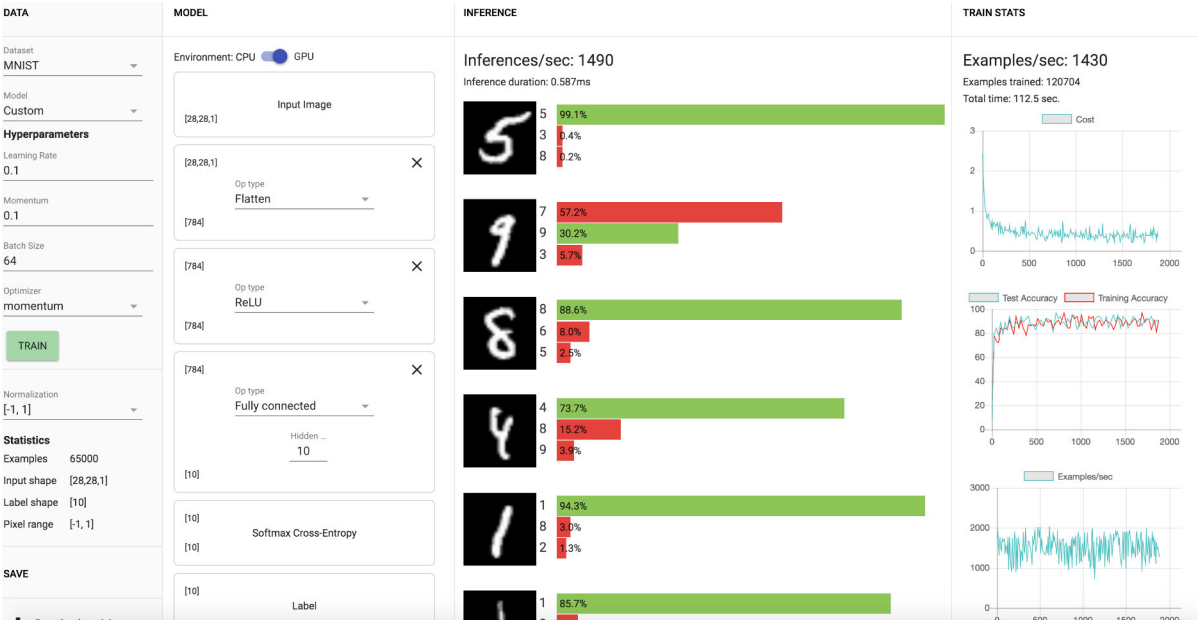
1. Train your MNIST model with 1,2,3,4, and 5 FC layers, with ReLU between them. For each, use the same hyperparameters, and the same number of hidden units (except for the last layer). What were the training times and accuracy? Do you see any overfitting? What can you conclude about how many layers to use? Include screenshots of the Training Stats for each of your examples.

CASE 1: INPUT > FLATTEN > FC(100) > RELU > SoftMax > Label - Training time 120 sec.



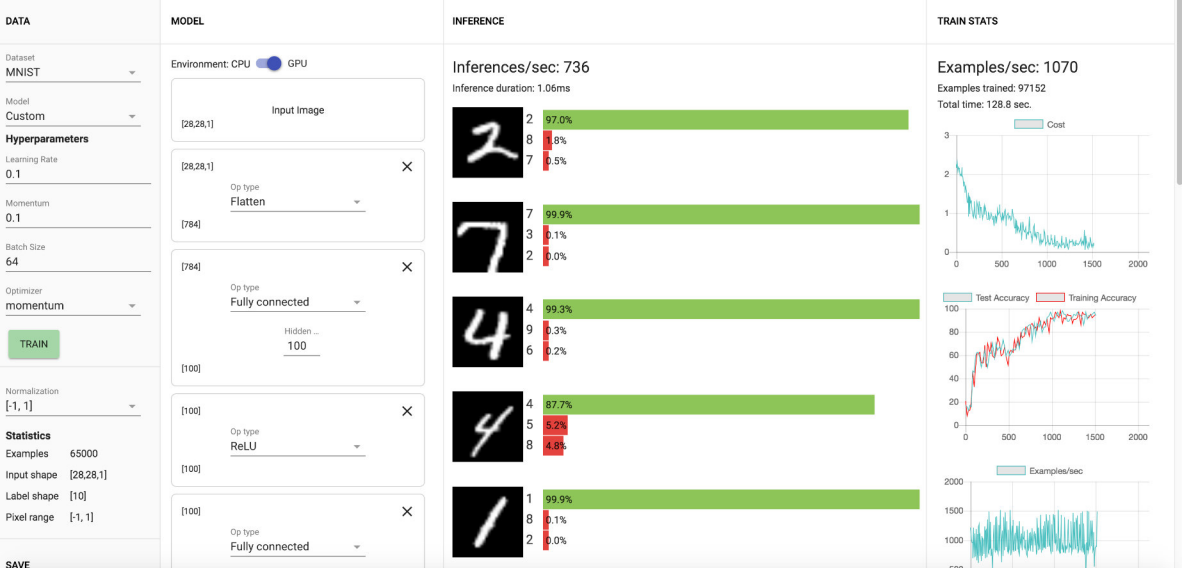
The model has an accuracy close to 91%, with an average of 1450 inferences per second and approximately 1400 Examples per second.

CASE 1: INPUT > FLATTEN > FC(100) > RELU > SoftMax > Label - Training time 120 sec. approx



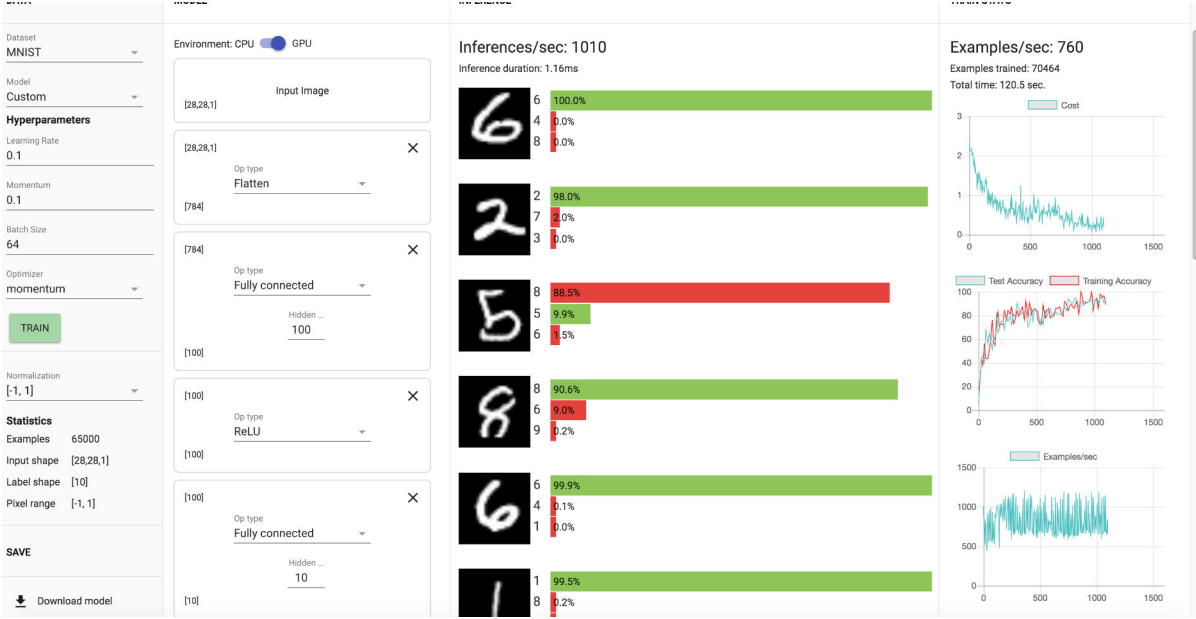
The model has an accuracy close to 91%, with an average of 1450 inferences per second and approximately 1400 Examples per second.

CASE 2: INPUT > FLATTEN > FC(100) > RELU > FC(10) > RELU > FC(10) > RELU > SoftMax > Label - Training time 120 sec. approx



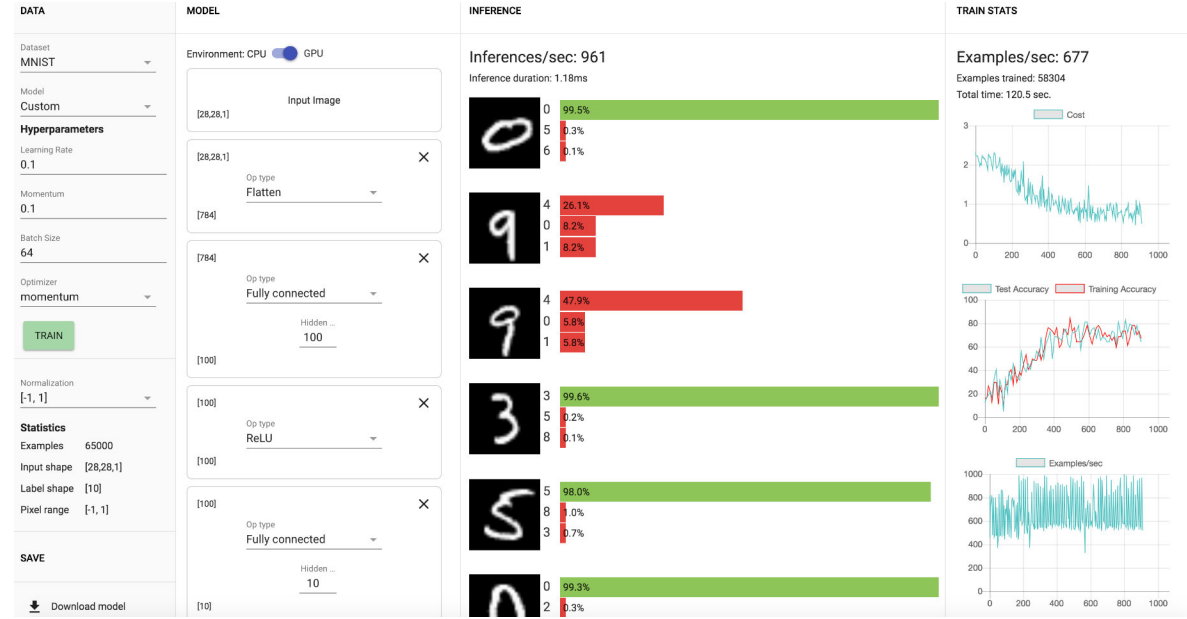
The model performs slower and compared to the previous one it takes longer in getting good results. after 90 seconds it reaches good results. with less inferences and less examples

CASE 3: INPUT > FLATTEN > FC(100) > RELU > FC(10) > RELU > FC(10) > RELU > FC(10) > RELU > SoftMax > Label - Training time 120 sec. approx



by adding one more FC layer and another RELU in between, it performs slightly better than the previous one.

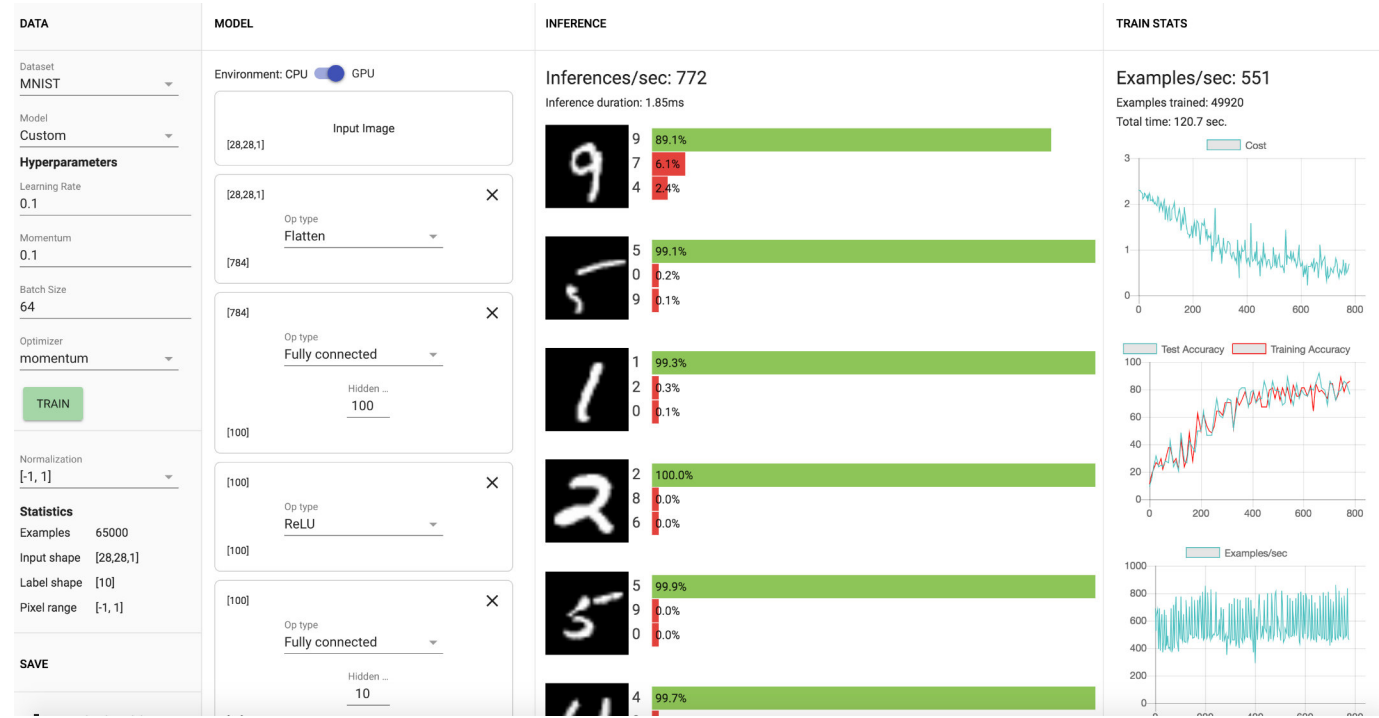
CASE 4: INPUT > FLATTEN > FC(100) > RELU > FC(10) > RELU > FC(10) > RELU > FC(10) > RELU > FC(10) > RELU > SoftMax > Label - Training time 120 sec. approx



Adding another FC and RELU layers makes it worst and slower in terms of both accuracy and speed.

Probably it shows signs of overfitting.

Case 4: INPUT > FLATTEN > FC(100) > RELU > FC(10) > RELU > FC(10) > RELU > FC(10) > RELU > FC(10) > RELU > FC(10) > RELU > SoftMax > Label - Training time 120 sec. approx



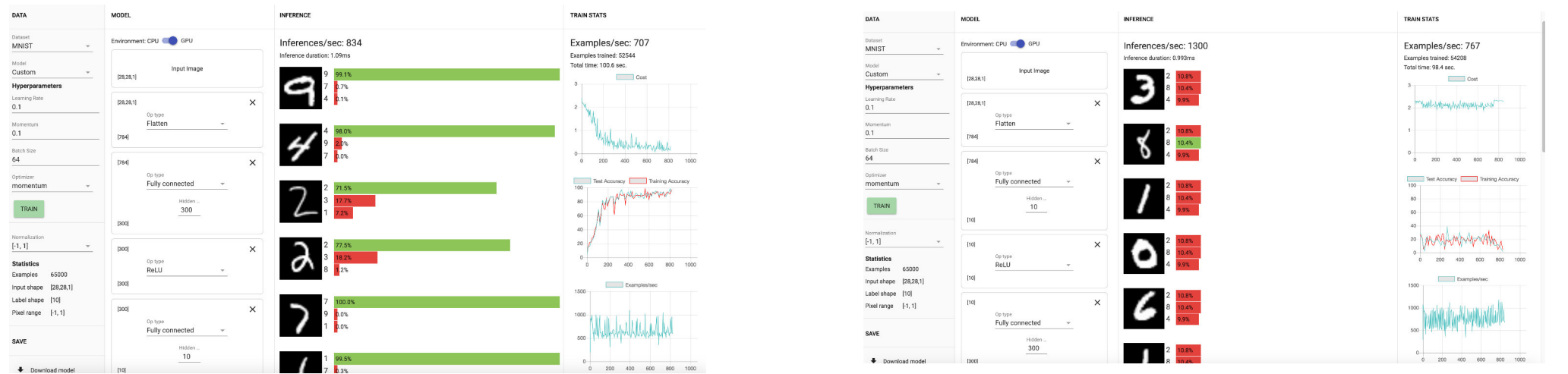
Finally , adding a 5th FC layer, shows that the model performs worst because is overfitting.

In my opinion and based on results, using 3 FC layers with RELU in between performed better than other configurations.

Build a model with 3 FC layers, with ReLU between them. Try making the first layer wide and the second narrow, and vice versa, using the same hyperparameters as before. Which performs better? Why do you think this is?

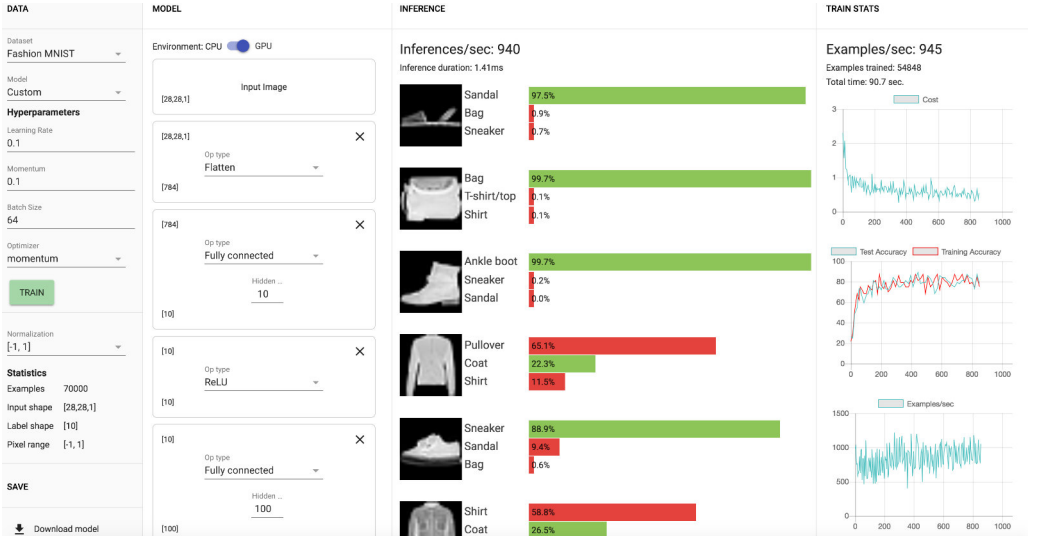
Making the first layer wider (FC(300)) , the NN performs well and fast enough

Inverting the number of units , makes que second NN perform really bad. My guess is that this is due to the small number of neurons in the first layer, the NN doesn't 'learn enough' to pass the values to the next layer.



Results are not very good when using the same configuration in Fashion MNIST.

Changing the hidden units number with the same configuration doesn't make a significant difference in terms of performance.



Repeating the same configuration on CIFAR-10, it performs really bad.

Inverting the number of hidden units doesn't make a difference. Maybe because tensors are more complex than the other 2 datasets

