Computational Mathematics

PROJECT #3 WITH MACHINE LEARNING
2017/2018

Ahmad Alleboudy | Lanpei Li
[ahmad.alleboudy@outlook.com] | [lilanpei@gmail.com]

Contents

1. Optimization Problems Solved

The main problem approached is the UNIPI AA1 CUP challenge, a regression problem utilizing data collected from noisy sensors.
However, during the building process of our lightweight library, we extensively utilized; the Monk's problem datasets, used for verifying the correctness of a machine learning algorithm.
In the following sections, we provide details on the development journey of our hopefully, MEGA project, for 12 Credits of the computational mathematics course, please enjoy ☺

1) MONK'S PROBLEMS DATASETS

The MONK's problem were introduced as artificial problems to verify the correctness of a machine learning algorithm and to compare different algorithms.

Which makes them a very great means for building one's own machine learning algorithms.

They are three classification problems, sharing the same artificial domain of a robot described by six attributes

1- x1: head shape:
   round, square, octagon
2- x2: body shape:
   round, square, octagon
3- x3: is smiling?
   yes, no
4- x4: holding an object of the following:
   sword, balloon, flag
5- x5: jacket color:
   red, yellow, green, blue
6- x6: has tie?
   yes, no

Each problem defines a logical expression that should be evaluated;

Problem M1:

(Head shape = body shape) or (jacket color = red)

From 432 possible examples, 124 were randomly selected for the training set. There were no misclassications.

Problem M2:

Exactly two of the six attributes have their first value.

(E.g.: body shape = head shape = round implies that robot is not smiling, holding no sword, jacket color is not red and has no tie, since then exactly two (body shape and head shape) attributes have their first value) From 432 possible examples, 169 were randomly selected. Again, there was no noise.

Problem M3:

(Jacket color is green and holding a sword) or (jacket color is not blue and body shape is not octagon)

From 432 examples, 122 were selected randomly, and among them there were 5% misclassications, i.e. noise in the training set.

We only used this dataset to build the skeleton of our library and managed to achieve 100% accuracy on problem 1 and 2 using both adam and SGD and 95% accuracy on problem 3, which are the perfect results to check the sanity of the implementation.

A quick hint on these results is available under:
https://docs.google.com/document/d/1volOPbnfBgjSr-iws1uocb9STbOj7EJMAdh5VG40dBA/edit?usp=sharing

The experiments for the Monk's problems are omitted as they are not relevant to the challenge, all of the following plots and experiments are only related to the AA1 CUP challenge datasets

2) AA1 CUP CHALLENGE

The provided data are of the format:

Training set:
id,input1,input2,input3,input4,input5,input6,input7,input8,input9,input10,target_x,target_y

And it has 1016 records, for building and selecting the model

Blind Test set:
id,input1,input2,input3,input4,input5,input6,input7,input8,input9,input10

And it has 315 records, for the final challenge

HIGHLIGHTS ON THE THEORETICAL ASPECTS

The motivation for this section is to shed the light on the type of the optimization problem being solved in order to understand better the behavior of the algorithms implemented and to be able to analyze their behavior.

The problem we are trying to solve is building a regressor that can map the inputs to their respective outputs in the provided dataset [it should also be able to generalize for unseen examples] and we wish it does that with the minimum error possible.

Let's formalize that!

We provide a hypothesis function h represented by our machine learning algorithm [the regressor]

and we define a cost function "mean squared error" that judges the quality of what our hypothesis outputs against the actual outputs provided in the dataset.

Our cost function in a basic form is half MSE $=\frac{1}{2n}\sum_{i=0}^{n}||h(x_i, weights) - y_i||^2$

Where we have n examples in our dataset, y is a ground truth (target output) and h(x_i) is our hypothesis applied to example x_i

We wish to minimize that cost function, but we have a few questions in mind to know what kind of optimization problem we are dealing with:

1- Is this a convex function? [If yes, then we have a single minimum point and we are doing convex optimization, our algorithm should converge to that point, otherwise this means there are many possible solutions some of them are local minima and our algorithm would need to be able to find the best possible solution]

2- Is it differentiable? [We care for that our function is differentiable so that we can in our search for a minimum point, with our iterative algorithms, use its derivative, [a function is decreasing against the direction of its derivative at any point, it is our hope to find a way to a minima!]]

Answering these two questions would depend on our used machine learning algorithm

so, is it convex?

The answer when utilizing a MLP with nonlinear activations, same as we do, is simply no!

Because our hypothesis h is not convex in its parameters, can we make sure?

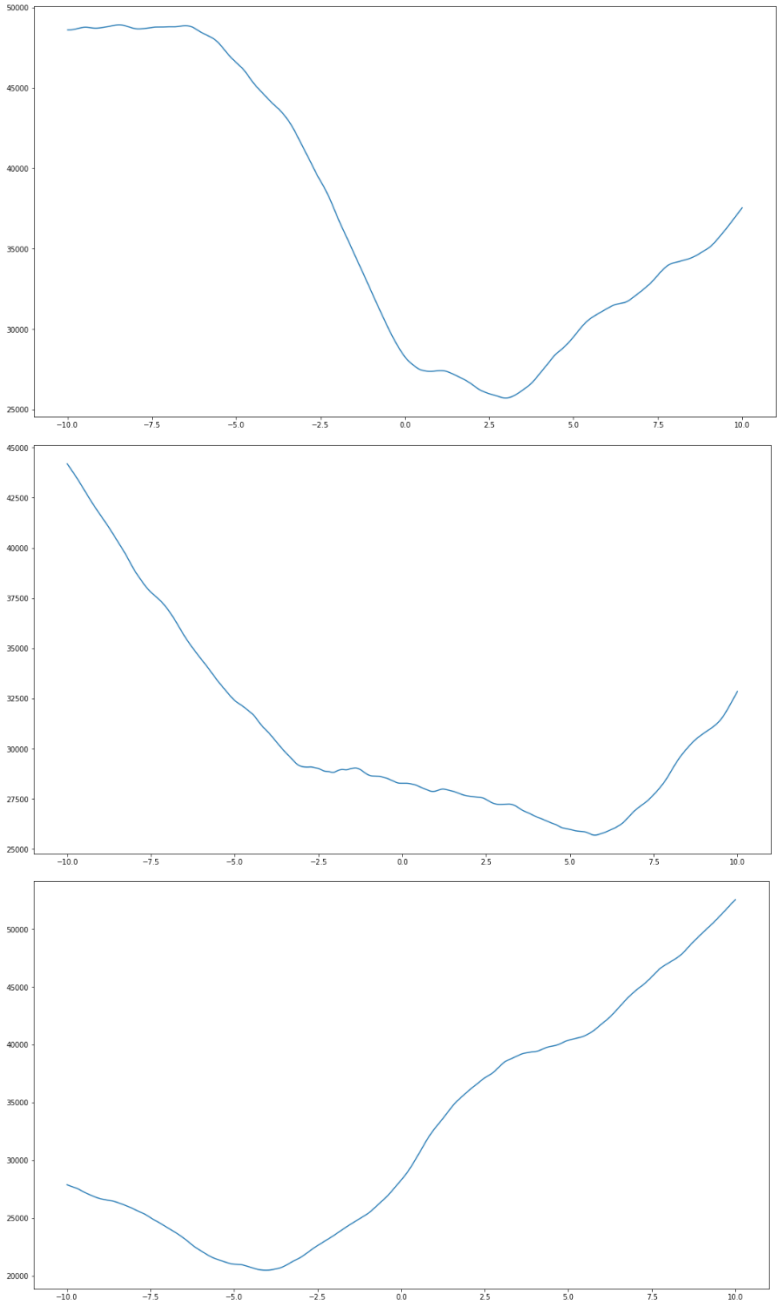
Yes, let's plot it!

But it is in a very high number of dimensions, due to the many weights as parameters we have in it, so, following Ian Goodfellow's answer on Quora [5], we need to plot a cross section of the function, perhaps repeatedly along different random directions and if we find one plot that is not convex then the whole function is not convex by counter example.

Experiment steps:

1- Pick a fixed initial value on our function [initial weights]
2- Pick a random direction d in the domain space [which could be simply represented by random small weights values]
3- for that picked direction change the network weights by initialWeights+s*d where s is a step [+ve or –ve]in the random direction
4- for each value change for the step s, compute the corresponding cost function
5- plot the step on the x axis and the cost function on the y axis

The following are plots generated from an experiment that could be rerun under project MLPTestDemo> Program.cs> function `CheckConvexity`



For how to run the Experiments, please check the 4) Examples section in 5. Implementation

How about the gradient computations of the cost function?

In order to minimize the function we need to find

How do our trainers deal with this problem?

Adam is utilizing running first and second order moments to rectify the learning rate automatically during the training, it also already incorporates the idea of a momentum in it.

SGD+Momentum, the idea of the momentum is to accumulate acceleration from previous steps taken by the algorithm, making it susceptible to local minimum points and small bumps in the function during the search for a solution, however, it has a fixed learning rate over all of the training process [possible to make sort of a timed reduction as we advance in the training process].

In the case of using a linear model, our hypothesis is convex, thus the cost function, that is why gradient descent [and other LLS solvers] will always find the sole global minima of our MSE.

OUR LOSS FUNCTION AND ITS GRADIENT

So, our cost function in a basic form is half MSE $=\frac{1}{2n}\sum_{i=0}^{n}||h(x_i, weights) - t_i||^2$

With the hypothesis of our model being $h(x_s, weights)$

But where did that cost function come from?

We have our dataset examples $x_i\ for\ i\ from\ 1\ to\ n$ and their corresponding set of targets [ground truth values] $t_i\ for\ i\ from\ 1\ to\ n$

We are assuming our regressor will learn to predict a value h(x_i,weights) for the example x_i as if it was a value drawn from a normal distribution centered around the real actual value t_i as its mean with a variance σ^2

So, the probability of having exactly the set of targets {t} for our dataset {x} is

$$P(t|x, weights, \sigma^2) = \prod_{i=1}^{n} N(t_i|h(x_i, weights), \sigma^2)$$

This product looks ugly, eh? So, for mathematical convenience we are going to maximize the log of this function [logarithms chew products and give them as sums instead 😊 + we can get rid of the exponent inside of the normal distribution formula!] and since the log function is monotonically increasing in its arguments [given an increasing sequence $\{x_i\}$ [x here could be anything, e.g f(y) where f is a function applied to a point y], $\ln(x_{i+1}) > \ln(x_i)$ i.e. preserving the order of the sequence], then maximizing the log of our function is same as maximizing the function itself .

So, we end up with

$$\ln(P(t|x, weights, \sigma^2)) = \ln(\prod_{i=1}^{N} \frac{1}{\sqrt{(2\pi\sigma^2)}} e^{-\frac{(h(x_i, weights)-t_i)^2}{2\sigma^2}})$$

which is

$$\ln(P(t|x, weights, \sigma^2)) = \ln(\Big(\frac{1}{\sqrt{2\pi\sigma^2}}\Big)^n \prod_{i=1}^{N} e^{-\frac{(h(x_i, weights)-t_i)^2}{2\sigma^2}})$$

$$\ln\big(P(t|x, weights, \sigma^2)\big) = \ln(\Big(\frac{1}{\sqrt{2\pi\sigma^2}})^n\Big) + \sum_{i=1}^{n} -\frac{(h(x_i, weights) - t_i)^2}{2\sigma^2})$$

$$= -\frac{1}{2\sigma^2}\sum_{i=}^{n}(h(x_i, weights) - t_i)^2 - \frac{n}{2}\ln(2\pi\sigma^2)$$

Intuitively we should minimize the variance around our target values as much as possible to make sure our model is predicting exactly the target values [the smaller variance of a normal distribution = the more the points drawn from the distribution are closely centered around the mean]

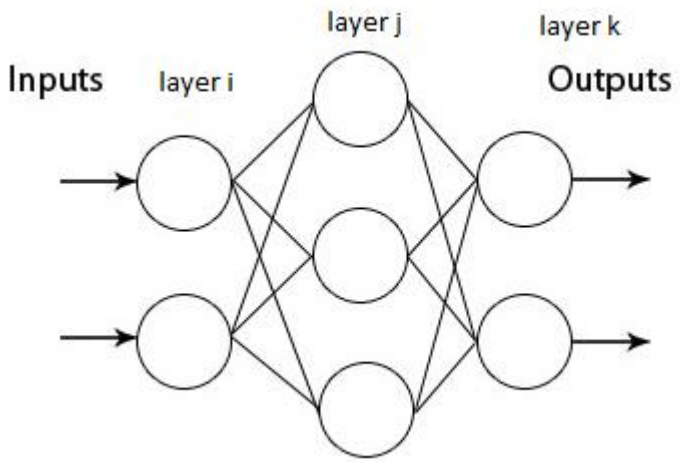
Taking the derivative w.r.t the variance and setting it to zero we get that

$$\sigma^{2*} = \frac{1}{n}\sum_{i=1}^{n}(h(x_i, weights) - t_i)^2$$

Our goal using our trainers here is to find the optimum set of weights that minimizes the variance. Which is exactly the same as minimizing the half MSE function we mentioned at the beginning of this section.

[the half is for mathematical convenience when we take its derivative later!]

Our architecture is a simple two-layer MLP of the shape:



Where we have an input layer i of 10 units, a single hidden layer j of 100 units, an output layer k of two units.

Following we provide our notations and the mathematical expressions needed

Symbol	Description
h()	An activation function
x_i	denotes input unit i. And, are same as z_i since the input layer utilizes a liner [identity] activation
z_i	Is h(x_i) where h() is the activation function and in the case of the input layer it is just linear identity so, $z_i = x_i$
w_{ji}	Weight w on the link between unit i in the input layer and unit j in the hidden layer
w_{kj}	Weight w on the link between unit j in the hidden layer and unit k in the output layer
a_j	The weighted [by weights w_{ji}] sum of the signals z_i coming from the input layer i to unit j in the hidden layer a_j=$\sum_i w_{ji} z_i$
z_j	Is h(a_j) where h() is the tanh() activation function in the case of the hidden unit
a_k	The weighted sum of the signals z_j coming from layer j to unit k in the output layer a_k=$\sum_j w_{kj} z_j$
z_k	Is the output at unit k after applying the activation function h() at layer k [which is a linear function for the output layer k] on a_k the weighted [by weights w_{kj}] sum of the input signals $z_j$coming to the unit k from the hidden layer j preceding it

t_k	Is the target value at unit k [for an example n]
E_n	The error [loss function] at example n $= \frac{1}{2}\sum_k (z_k - t_k)^2$
$\frac{\partial E_n}{\partial w_{ji}}$	The partial derivative of the error E_n function with respect to the weight w_{ji} $= \frac{\partial E_n}{\partial a_j}\frac{\partial a_j}{\partial w_{ji}}$ according to the chain rule We call $\frac{\partial E_n}{\partial a_j}$ as δ_j *and* $\frac{\partial a_j}{\partial w_{ji}}$ *is actually* z_i So, $\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i$
δ_k	Is the error at the output layer k = $z_k - t_k$ Since $\delta_k = \frac{\partial E_n}{\partial a_k}$ and $E_n = \frac{1}{2}\sum_k (z_k - t_k)^2$ and $a_k = z_k = h(a_k)$ and h() is linear [identity] for the output layer k
δ_j	Is the error at the hidden unit j which is = $\frac{\partial E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k}\frac{\partial a_k}{\partial a_j}$ Which is = $h'(a_j)\sum_k w_{kj}\delta_k$ since $a_k = \sum_j w_{kj} z_j$ *and* $z_j = h(a_j)$, *h is the activation function at layer j* which is the tanh activation function

Note: why δ_x are often called errors[14]?

For the output $\delta_k = z_k - t_k$ is proportional to the difference between what the network outputs and the ground truth, so, in a sense how "off" is the result or the "local error" for that neuron.

Then:
$\delta_j = h'(a_j)\sum_k w_{kj}\delta_k$

For each hidden unit, the value δ_j is proportional to the sum of the weighted local errors δ_k of the output layer connected to it, making it also encoding a sense of local error the hidden neurons are providing.

For bookkeeping, our cost function in its majestic matrix form when using our MLP is:

$$\frac{1}{2N}\sum_{n=1}^{N}\left(\left(\mathrm{W_{kxj}}\tanh\left(W_{jxi}x_{nix1}\right) - t_{nkx1}\right)^2\right.$$

Where $\mathrm{W_{kxj}}$ is the weight matrix between the hidden and output layer.

W_{jxi} is the weight matrix between the input and hidden layer.

x_{nix1} is the column vector of the input signals for example n and t_{nkx1} is the corresponding column vector for the output of that same example n

So, for backpropagation, we:

1- apply a forward propagation of an example x_n to our network, computing the activations a_j then $z_j = h(a_j)$
2- evaluate δ_k for all the units in the output layer
3- computing δ_j values using the values acquired for δ_k
4- then computing the derivative $\frac{\partial E_n}{\partial w_{ji}}$=$\delta_j z_i$

There is one major issue during training though, the tanh function we are using as activations in the hidden layer would saturate if the weights are updated to very large or very small values at any step during the training falling in the trap of the "vanishing gradient" when computing $\delta_j = h'(a_j)\sum_k w_{kj}\delta_k$

Since the tanh at large values [typically >=4] is almost a constant equal to 1 and at small values [typically <=-4] is almost a constant equal to -1.

If this happens, the gradient will be too small almost 0 making the training by changing the weights in the direction against the gradient with the learning step almost 0.

<u>Continuity of the gradient of our cost function</u>

After breaking down the gradient computations above, we notice that the part controlling the properties of the loss function and its gradient is $\delta_j = h'(a_j)\sum_k w_{kj}\delta_k$

Our hypothesis with single input x ->weight w1 -> hidden[with tanh activation] ->weight w2 -> and single output unit

$y = w_2 tanh(w_1 x)$

$E = Loss\ function = \frac{1}{2N}\sum_N (y_n - t_n)\text{^}2$

$For\ a\ single\ example = ½\ (y-t)^2$

$= ½\ (w_2\ tanh(w_1 x) - t)^2$

$\frac{\partial E}{\partial w_1} = (w_2\ tanh(w_1 x) - t) * x w_2 sech^2(w_1 x)$

$\frac{\partial E}{\partial w_2} = (w_2\ tanh(w_1 x) - t)(\tanh(w_1 x))$

$But\ w_2 \epsilon R,$

$\lim_{w_2 \to inf} (w_2\ tanh(w_1 x) - t) * x w_2 sech^2(w_1 x) = \inf$

It is unbounded, so, it can't be globally Lipschitz.

Hypothesis made by our accelerated gradient methods used for guaranteed convergence results [15]:

1- the cost function is convex
2- the gradient of the cost function is Lipschitz continuous with L>0

And with fixed step size $t \leq \frac{1}{L}$ it is satisfied that $f\left(x^{(k)}\right) - f\left(x^{(*)}\right) \leq \frac{2\left\|\left|x^{(0)} - x^{(*)}\right|\right\|^2}{t(k+1)^2}$

Achieving the optimal $O(\frac{1}{k^2})$ convergence rate I.e., to get $f(x^{(k)}) - f(x^{(*)}) \leq \epsilon$, need $O(\frac{1}{\sqrt{\epsilon}})$ epochs.

However, as we just saw in the sections above:

1- our cost function is non-convex
2- its gradient is not globally Lipschitz.

So, we can no longer guarantee these convergence promises anymore!

However, as done by [3], we can still show convergence is possible with practical experiments instead as we do in the following section where we provide empirical results of a comparison we set between our implementation of Adam against SGD+Momentum[Nesterov]

The settings and hyperparameters used in these experiments are the result of our exhaustive tuning in section "2. The implemented solution methods and how good are out-of-sample results"

THE TRAINERS WE IMPLEMENTED

Adam Optimizer

Quoting from [3]

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Note: *ϵ is used to prevent a division by zero when updating the parameters before end while*

The stochastic gradient descent + Momentum both the classic [new weights Updates = momentum rate* old weights updates +learning rate the weights gradient] [4]

$$v_{t+1} = \mu v_t - \varepsilon \nabla f(\theta_t) \qquad (1)$$

$$\theta_{t+1} = \theta_t + v_{t+1} \qquad (2)$$

and Nestrov [new weights Updates = momentum rate* old weights updates + learning rate*the weights gradient at the updated weights][4]

$$v_{t+1} = \mu v_t - \varepsilon \nabla f(\theta_t + \mu v_t) \qquad (3)$$

$$\theta_{t+1} = \theta_t + v_{t+1} \qquad (4)$$

Where v_t is the weights updates at step t and θ_t are the weights at step t
μ is the momentum rate and epsilon is the learning rate.

ANALYSIS AND EXPECTED BEHAVIOR OF ADAM AND SGD+MOMENTUM

According to the analysis in the paper [3], ADAM always outperformed SGD+Momentum[Nesterov] in all of the provided settings, so, we would restrict our test to Adam vs SGD+Nestrov Momentum, they are our final models that made it to the Finish line after all.

The following experiments utilize the 60percent dataset, which is 60% train [60percentTrain.txt] 40% validation[60percenttest.txt] splits of the data available under: UsedFiles\TrainValSplits

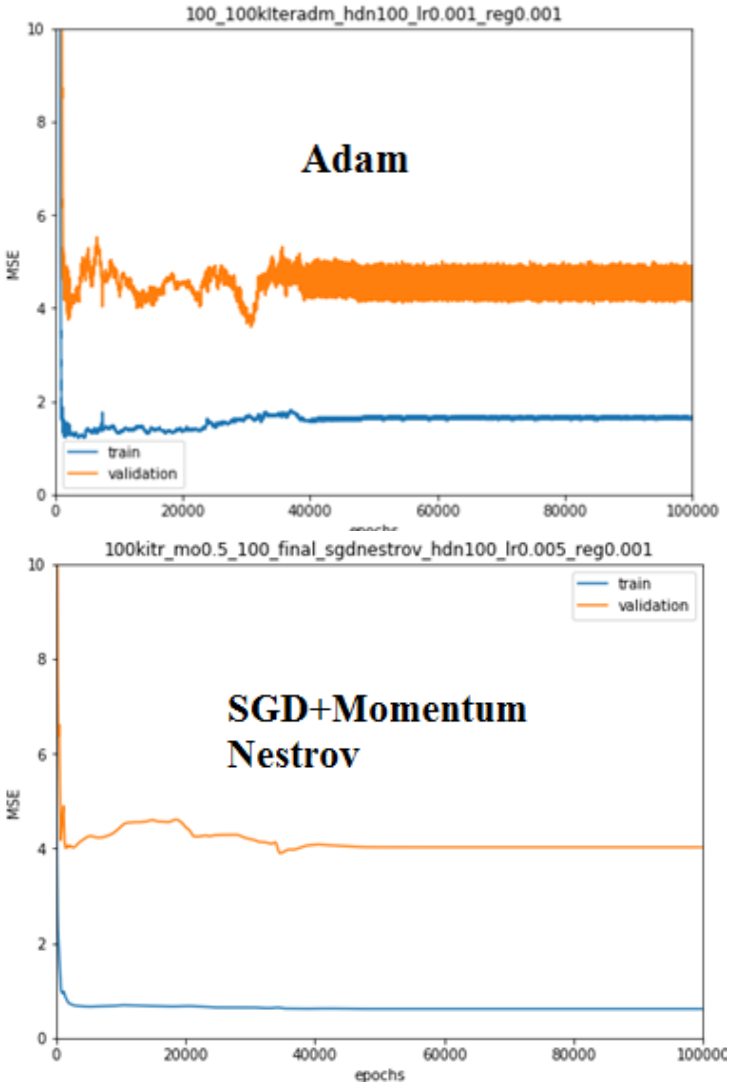
Each row in the dataset is made of 12 values, the first 10 are the inputs and the last 2 are the corresponding outputs.
More details are present in section 2. Implemented solution methods and in section 6. Provided files and Folder.

We trained two models with the following settings:
1- With A stochastic gradient descent optimizer +Nestrov momentum

Number Of Epochs = 100000; Batch Size = 10; Learning Rate = 0.001; regularization = L2; regularization Rate = 0.001; Nestrov = true; Momentum Rate = 0.5; Number Of Hidden Units = 100;

2- with Adam Optimizer

Number Of Epochs = 100000; batch Size = 10; Learning Rate = 0.001; regularization = Regularizations.L2; regularization Rate = 0.001; Number Of Hidden Units = 100;



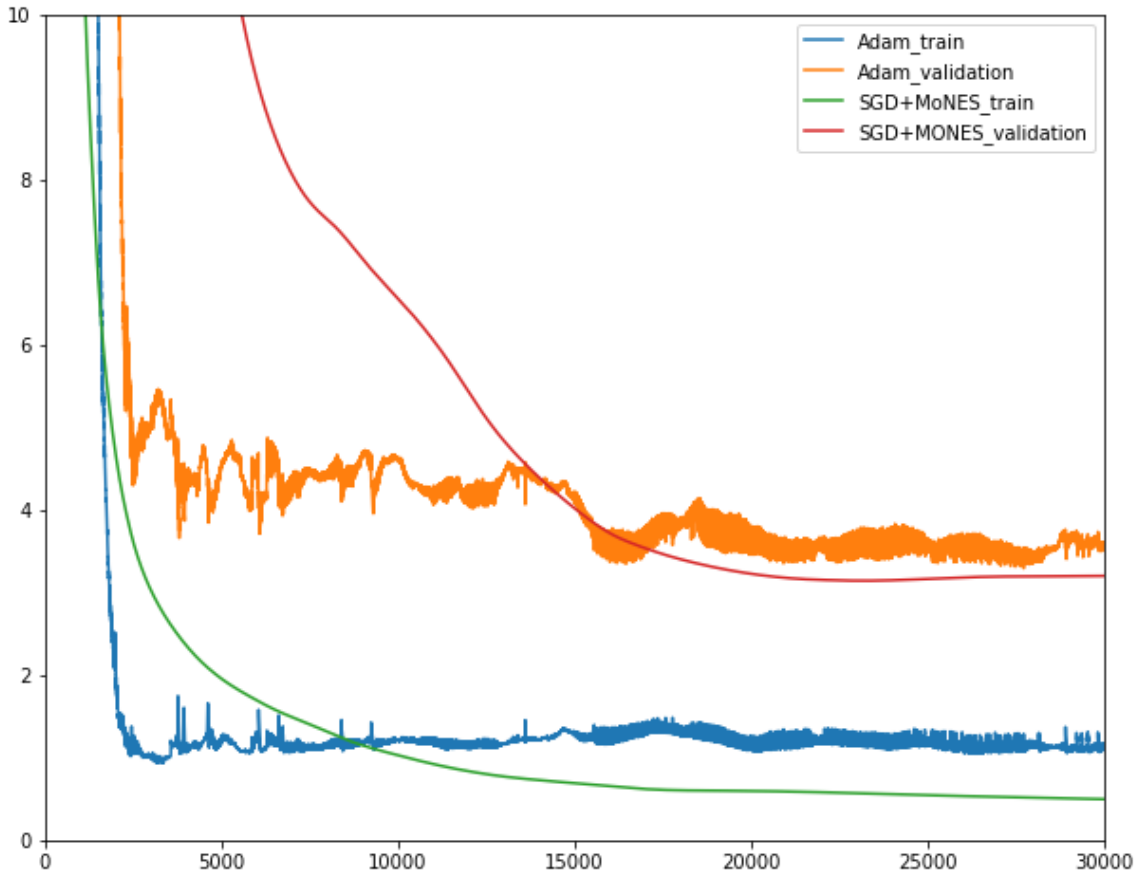
We can easily notice that both algorithms converge around 45k epochs, but for this problem SGD+Nesterov momentum seem much more stable than Adam.

To fight off the instability issue, perhaps utilizing a larger batch size would be a better idea reducing the stochasticity of our loss function.

We reran the experiment but with Batch size of 50, 30k epochs each:

	SGD+Momentum[Nesterov]	**Adam**
MEE	1.353	1.435
MSE	3.200	3.583
Elapsed Time	5053507 ms	5287189 ms

The two plots together for the second experiment with a larger batch size to reduce the stochasticity in our gradient updates:



Still we can find SGD+Momentum[Nesterov] is more stable and converging to a better solution, although they both seem to have a similar convergence rate after about 20k epochs.

Our conclusion is that this difference could be due to the problem being solved.

MODALITY OF THE LOSS FUNCTION

Perhaps it was quite clear from the cross sections taken of the loss function that it is unimodal with a single global minimum point, that is why our algorithms had no troubles finding it and never getting stuck in local minima or saddle points.

As we have mentioned, both SGD+Momentum and Adam are of the accelerated gradient methods, both accumulate momentum escaping local minima and saddle points.

To verify that we have a single global minimum, we retrained our model several times utilizing different initial weights at each time and compared the acquired weights:

Seed	Euclidean Norm
1	24.386
73	23.522
40	23.703
4	24.917
2	23.833
15	23.887

And here we report the Euclidean distance between the acquired vector weights

Seed\Seed	1	73	40	4	2	15
1	0	33.89	32.758	33.755	32.784	32.629
73	33.89	0	33.008	33.96	33.51	33.182
40	32.758	33.008	0	33.837	34.024	32.566
4	33.755	33.96	33.837	0	33.209	33.247
2	32.784	33.51	34.024	33.209	0	33.124
15	32.629	33.182	32.566	33.247	33.124	0

And after 20k epochs:

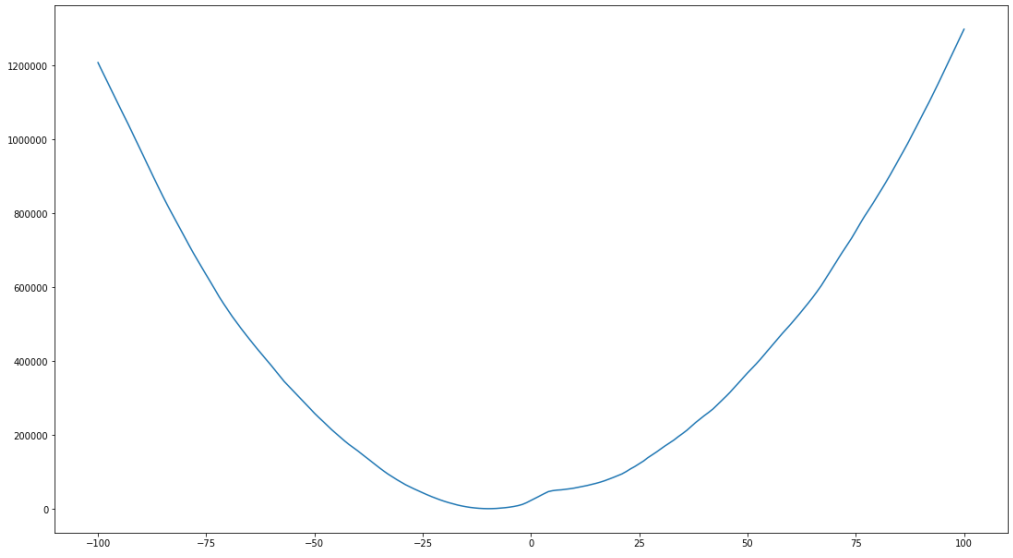
Seed	Euclidean Norm
1	21.276
73	21.368
40	21.432
4	21.255
2	21.585
15	21.563

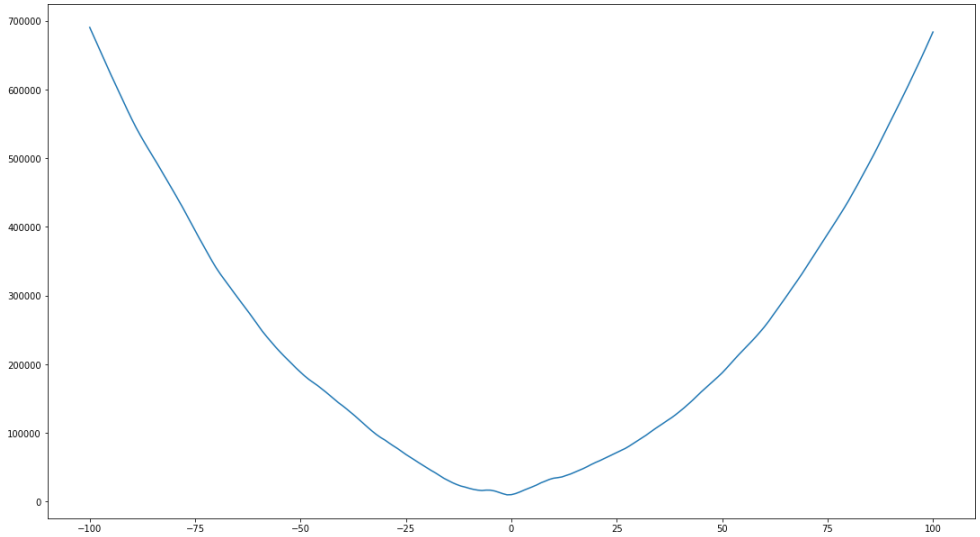
Seed\Seed	1	73	40	4	2	15
1	0	29.974	29.979	29.858	30.219	30.558
73	29.974	0	30.747	30.572	30.655	30.726
40	29.979	30.747	0	30.122	30.046	29.742
4	29.858	30.572	30.122	0	30.451	29.3
2	30.219	30.655	30.046	30.451	0	31.054
15	30.558	30.726	29.742	29.3	31.054	0

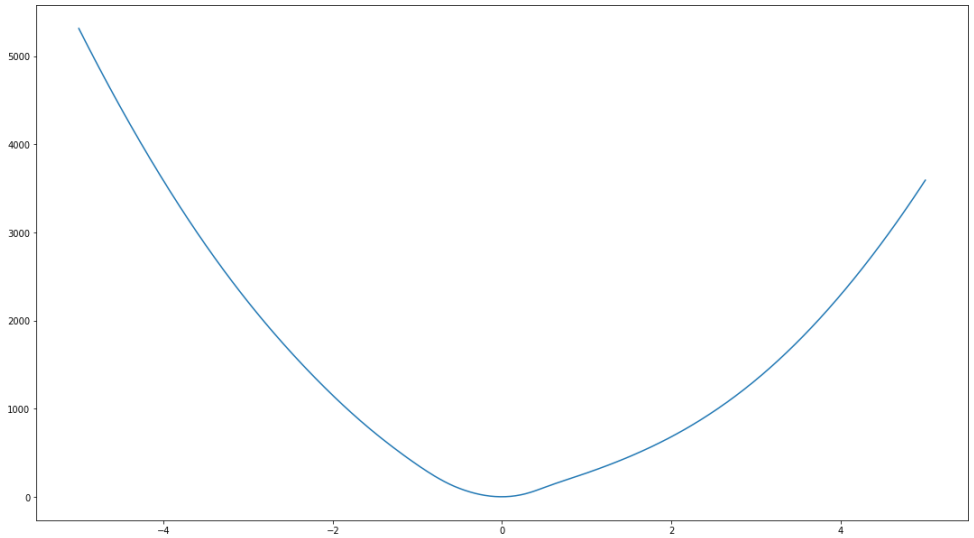
The norms are getting to similar values and the Euclidean distances between the weights vector are getting smaller, an interesting experiment would be to train the models with the different seeds in parallel and interrupting the training each maybe 5k epochs then report these numbers and resume training til the next 5k epochs and so on, to see where these numbers are converging, but unfortunately we are running too late on the deadline for submitting our documents for the graduation session, so, we are adding this to the future work.

But for now, we can at least draw a cross section of the cost function at a larger scale than before:

The following three plots are using the same method for plotting the cross section, but with the change of the scale for the step taken along the random direction picked it is now from -100 to 100 instead.


1200000
1000000
800000
600000
400000
200000
0
−100
−75
−50
−25
0
25
50
75
100


700000
600000
500000
400000
300000
200000
100000
0
−100
−75
−50
−25
0
25
50
75
100


5000
4000
3000
2000
1000
0
−4
−2
0
2
4

2. The implemented solution methods and how good are out-of-sample results

1) MLP

The main algorithm is a multilayer perceptron topology with a Tanh activation along with its trainer algorithms as required.

All of the experiments reported here are utilizing the Grot weight initialization using the same seed to generate the random weights

a. Adam Optimizer

We also provided our Adam trainer implementation as the algorithm of our choice from the class of accelerated gradient methods

For the Adam optimizer, we introduced the following hyperparameters;

Learning Rates, here it represents the learning rate Upper Bound for adam

Regularization Rates

A grid search is utilized to decide the final values of the hyperparameters:

Learnig Rate Upper Bound	0.001,0.005,0.01
Regularization Rate	0,0.001,0.01
Number Of hidden units	From 10 to 100 with a step of 10

Table 1 – **Adam hyperparameters**

The line search used:
Is a simple fixed step in the case of SGD+Momentum with the possibility to increase or decrease it utilizing the resilient setting

And is of course updated in the case of the Adam.

Stopping criteria:
For each setting, we trained with initially 1000 epochs, then examined all of the learning curves, to decide on a proper stopping criterion

we found that the runs are converging at very different number of epochs, some are actually flickering, some diverging and some still require more training epochs, which made it difficult to set a unified criterion for early stopping or decide the last number of epochs for the training on the whole dataset we decided to utilize regularization + a standard stopping criterion[2] i.e. no improvement in the loss anymore after a quite excessive number of epochs which could be examined easily from the learning curve of the runs, to decide the final number of epochs to be 5000 epochs for Adam's experiments

Here we report the top 20 smallest average MSE experiments run with Adam,

We used k-folds cross validation with k=5.
The following table reports for the top 10 experiments, the average MSE of the 5-folds for each experiment hyperparameters setting ordered by the average MSE ascending.

How to read the title?
hdn{x}=means number of hidden units in the hidden layer is x
k{5}= 5-fold experiment
lr{x} = learning rate is x
reg{x} = regularization is x

mo{x} = momentum value for SGD+Momentum

	name	avgMSE
73	hdn20_k5_lr0.001_reg0.001	2.869099
64	hdn30_k5_lr0.001_reg0.001	2.941200
37	hdn60_k5_lr0.001_reg0.001	2.945531
22	hdn80_k5_lr0.005_reg0.001	2.974550
55	hdn40_k5_lr0.001_reg0.001	2.980975
46	hdn50_k5_lr0.001_reg0.001	2.991854
19	hdn80_k5_lr0.001_reg0.001	3.024321
10	hdn90_k5_lr0.001_reg0.001	3.041776
1	hdn100_k5_lr0.001_reg0.001	3.045349
68	hdn30_k5_lr0.005_reg0.01	3.050320
31	hdn70_k5_lr0.005_reg0.001	3.064476
76	hdn20_k5_lr0.005_reg0.001	3.068522
67	hdn30_k5_lr0.005_reg0.001	3.103125
59	hdn40_k5_lr0.005_reg0.01	3.106687
13	hdn90_k5_lr0.005_reg0.001	3.119385
58	hdn40_k5_lr0.005_reg0.001	3.139477
50	hdn50_k5_lr0.005_reg0.01	3.139826
77	hdn20_k5_lr0.005_reg0.01	3.146856
82	hdn10_k5_lr0.001_reg0.001	3.156930
40	hdn60_k5_lr0.005_reg0.001	3.164501

Table 2 - Top 10 experiments

Here we report the a few experiments plots, for a comprehensive list of experiments plots and results, please find the folder adamexperimentsfinalPLOTS

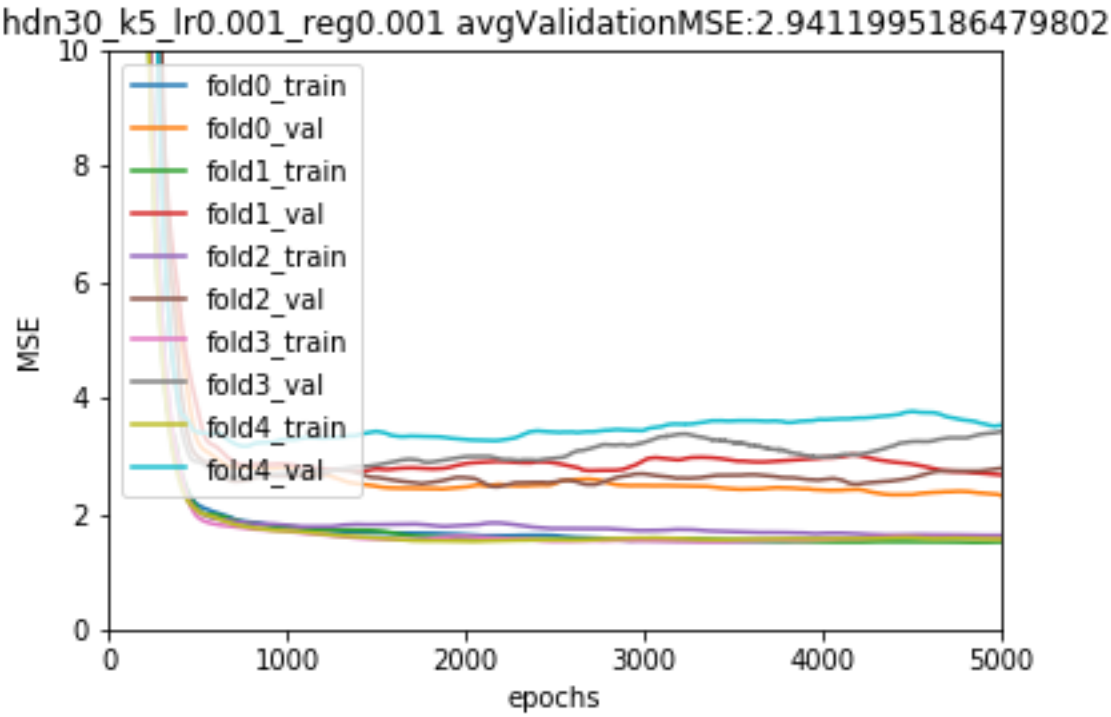


Figure 1: Learning Curve of hdn30_k5_lr0.001_reg0.001

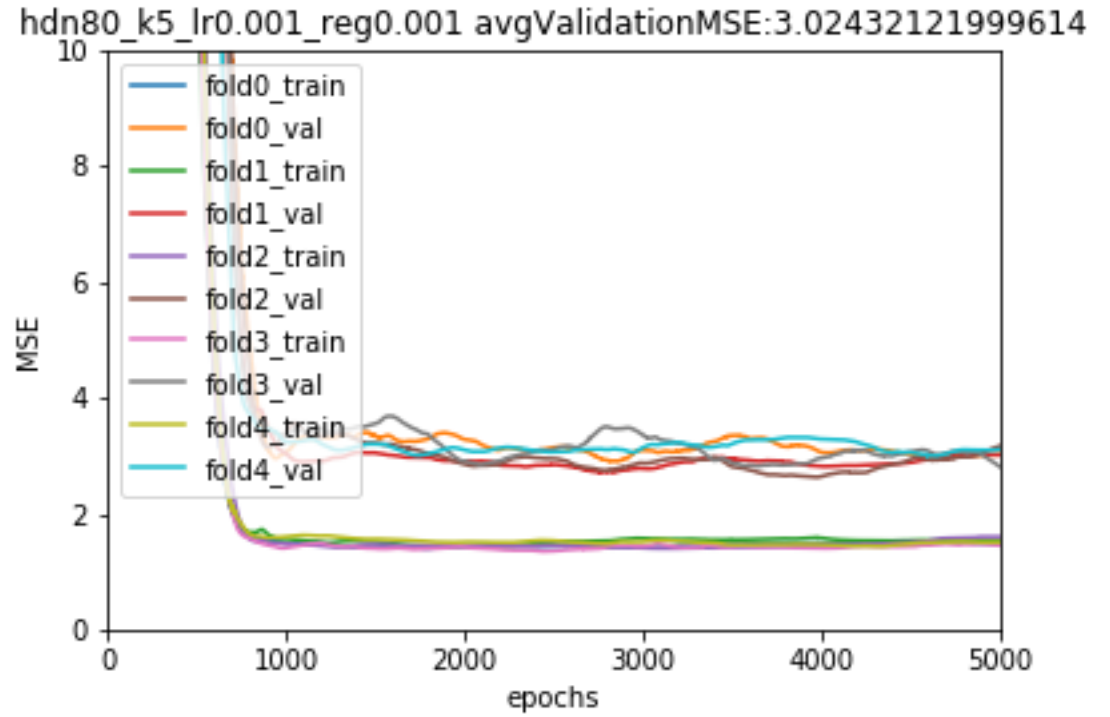


Figure 2: Learning Curve of hdn80_k5_lr0.001_reg0.001

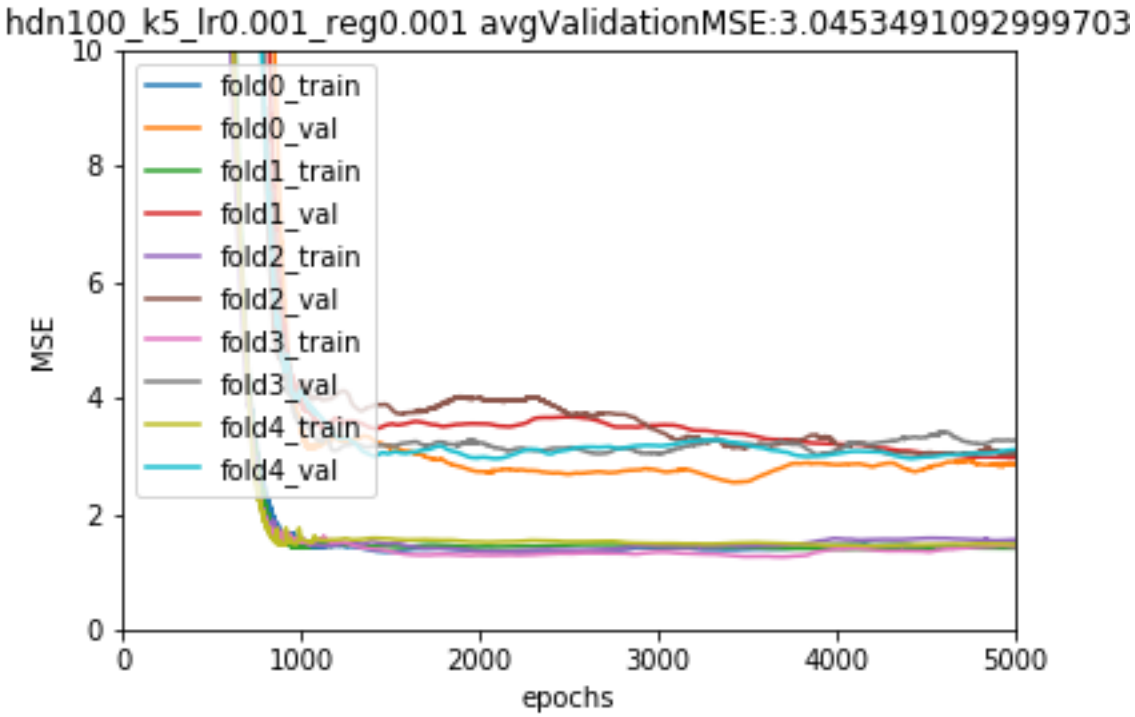


Figure 3: Learning Curve of hdn100_k5_lr0.001_reg0.001

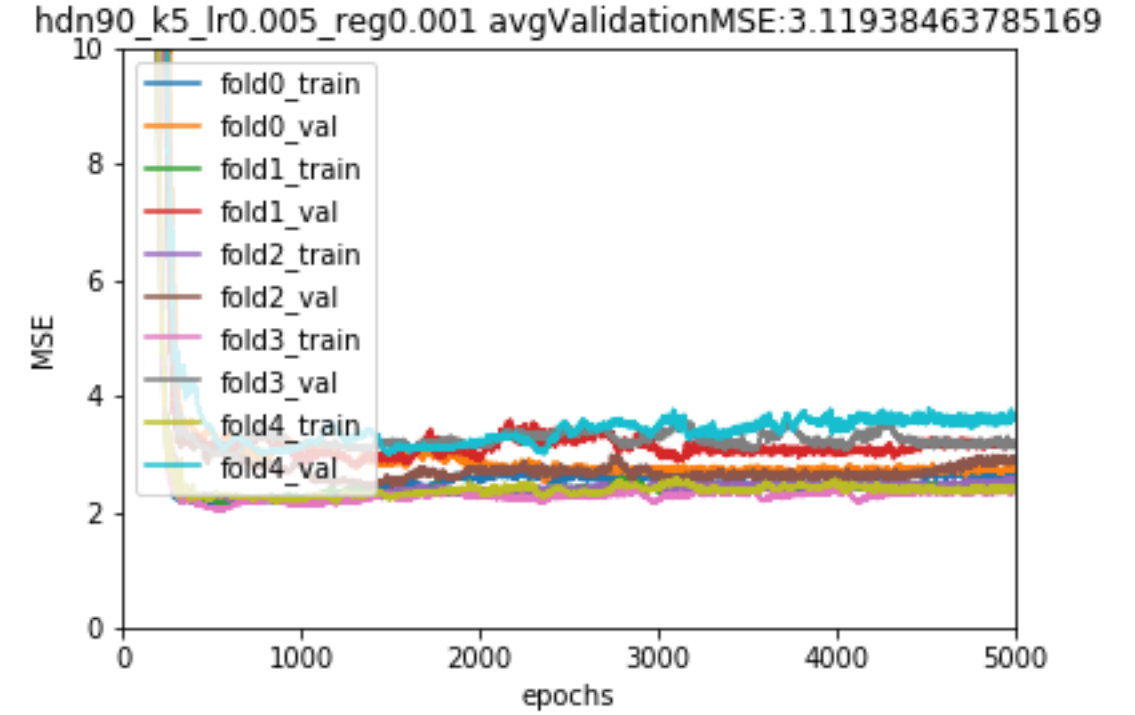


Figure 4: Learning Curve of hdn90_k5_lr0.005_reg0.001

The final choice is based on the following properties we defined:

1- it belongs to the top 20 least average MSEs
2- its curve is quite stable, [not so much flickering]
3- the validation curves are not performing radically different
4- the plot shows potential to either improve for more than 5000 epochs or at least already converging, in the former case, we will re-run the winner experiment to an extended number of epochs
5- vague plots where validation MSE is starting lower than training MSE are not considered

Perhaps the winner could be hdn100_lr0.001_reg0.001 as it belongs to the top 20, the curves are stable, not performing radically different and actually showing a potential to run for more than 5000 epochs and still improve, alternatively hdn80_lr0.001_reg0.001 seems also as a good choice, but without the same potential to improve after 5k epochs.

We decided a final experiment using the traditional Train/validation/test cross validation strategy with a 60% training, 20% validation and 20% testing reporting validation MSE and MEE for testing and validation splits, choosing the best of both based on the validation MSE score

Reproducing the same experiments results:
Please run the project AA1_CUP Program.cs for SGD+momentum and Adam screening

Simple usage example:
SGD+Momentum[Nestrov]

Number Of Epochs = 5000; batch Size = 10; learning Rate = 0.001; regularization = L2; regularization Rate = 0.001; nestrov = true; momentum = 0.5; Number Of Hidden Units = 100;

And Adam:

Number Of Epochs = 5000; batch Size = 10; learning Rate = 0.001; regularization = L2; regularization Rate = 0.001; Number Of Hidden Units = 100;

b. SGD+Momentum

We implemented a trainer utilizing the stochastic gradient descent algorithm with the following hyper parameters

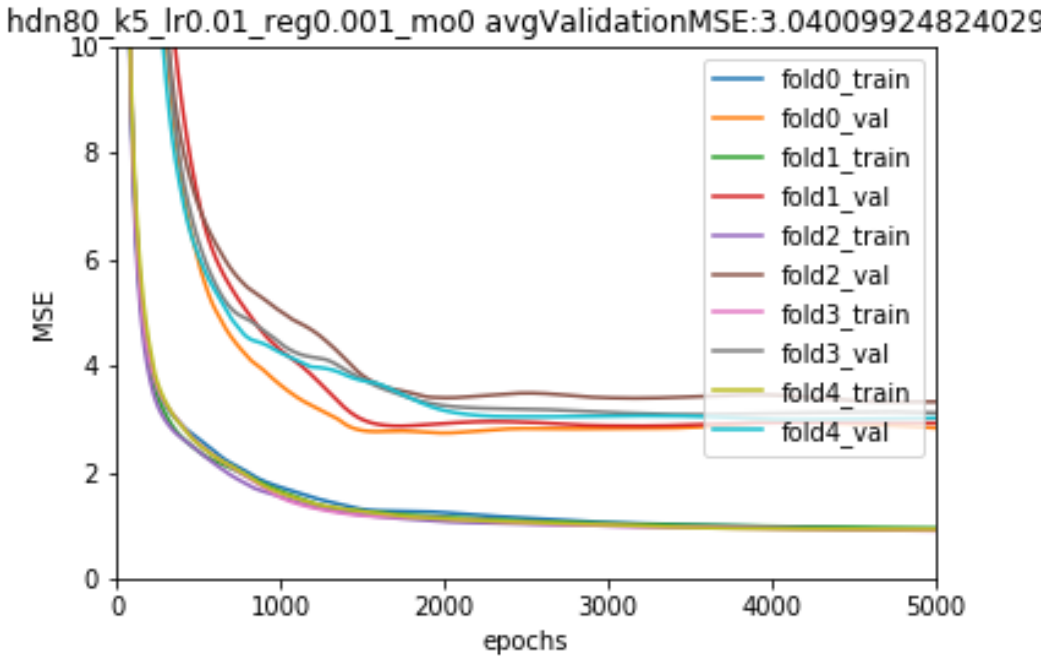
SGD+Momentum hyperparameters:

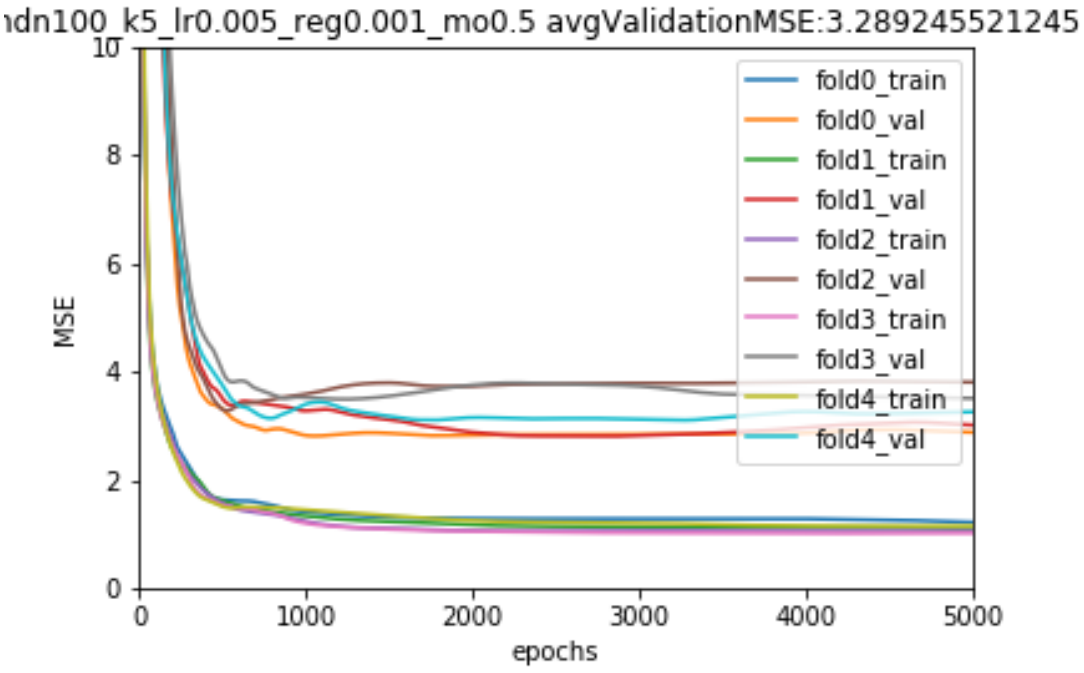
Momentum [both classic and nesterov implementations]
For the SGD+Momentum optimizer, we introduced the following hyperparameters;
Momentum rates [0, 0.5]
Learning Rates [0.005, 0.01]
Regularization Rates [0, 0.001]
Nesterov: [on or off]
SGD+Momentum [Nesterov] top 20 experiments report:

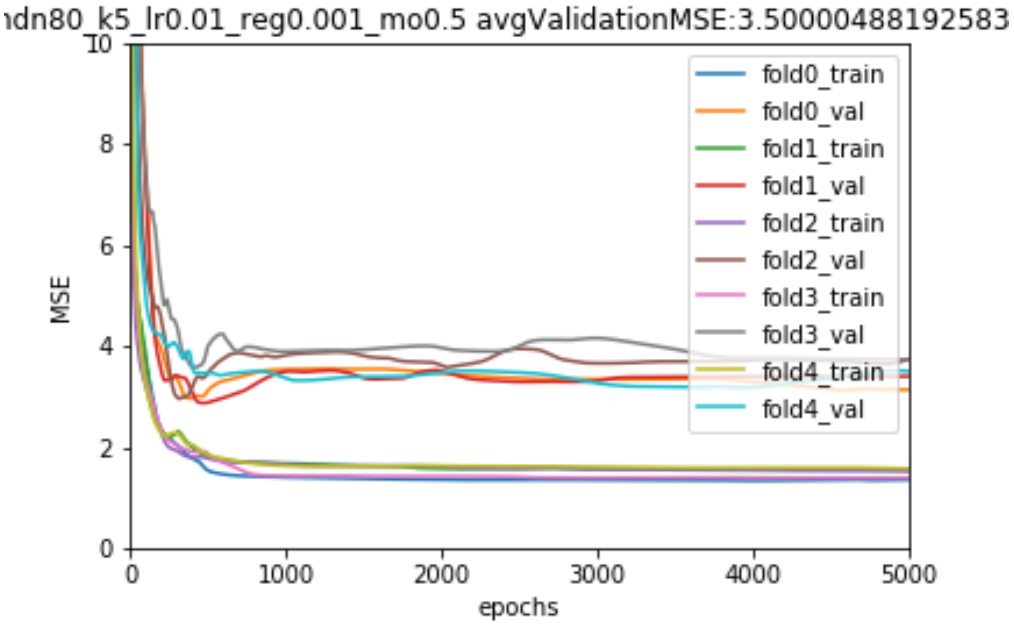
39	hdn20_k5_lr0.01_reg0.001_mo0.5	2.879580
33	hdn20_k5_lr0.005_reg0.001_mo0	2.928588
27	hdn40_k5_lr0.01_reg0.001_mo0	2.976110
25	hdn40_k5_lr0.005_reg0.001_mo0	3.011973
35	hdn20_k5_lr0.01_reg0.001_mo0	3.017527
11	hdn80_k5_lr0.01_reg0.001_mo0	3.040099
1	hdn100_k5_lr0.005_reg0.001_mo0	3.049197
19	hdn60_k5_lr0.01_reg0.001_mo0	3.069511
37	hdn20_k5_lr0.005_reg0.001_mo0.5	3.077974
17	hdn60_k5_lr0.005_reg0.001_mo0	3.088708
9	hdn80_k5_lr0.005_reg0.001_mo0	3.155544
29	hdn40_k5_lr0.005_reg0.001_mo0.5	3.251418
31	hdn40_k5_lr0.01_reg0.001_mo0.5	3.267038
5	hdn100_k5_lr0.005_reg0.001_mo0.5	3.289246
3	hdn100_k5_lr0.01_reg0.001_mo0	3.292397
21	hdn60_k5_lr0.005_reg0.001_mo0.5	3.292790
13	hdn80_k5_lr0.005_reg0.001_mo0.5	3.380744
7	hdn100_k5_lr0.01_reg0.001_mo0.5	3.397745
15	hdn80_k5_lr0.01_reg0.001_mo0.5	3.500005
23	hdn60_k5_lr0.01_reg0.001_mo0.5	3.589723

Table 6 - Top 20 experiments

A Few of the top Plots:







For a complete list of experiments results and learning curves plot, please visit the folder sgdnestrovfinal

Again, with the same criteria we defined for picking the model earlier in Adam's report,

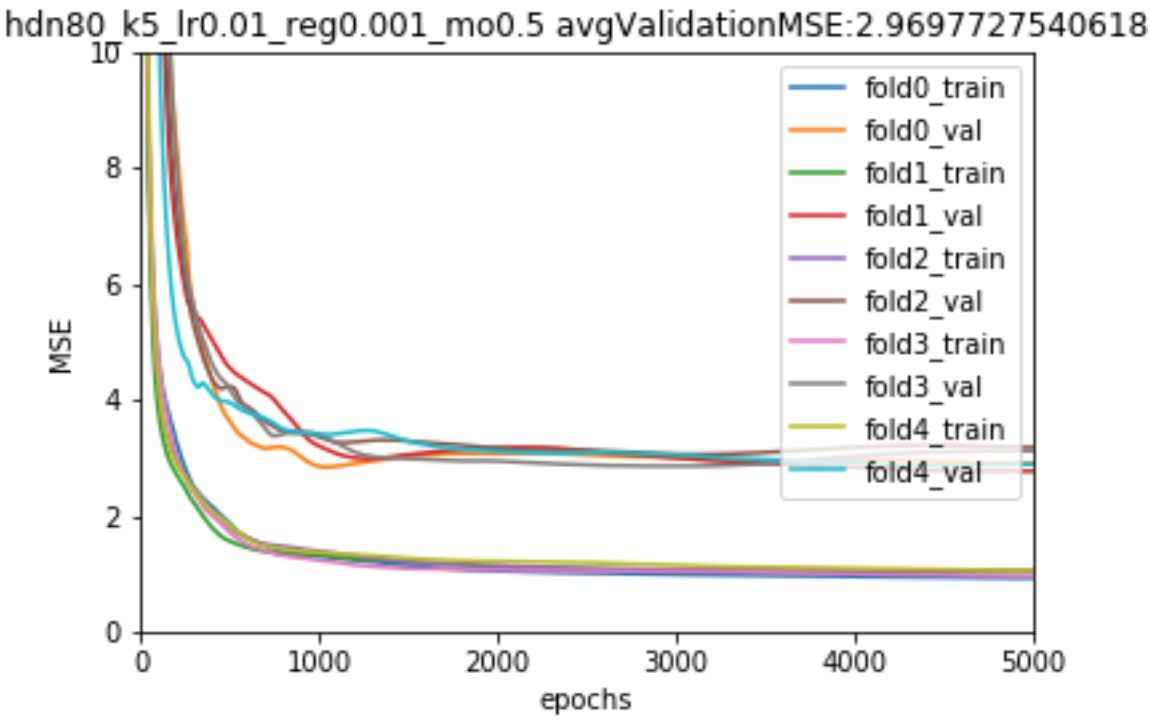
sgd+ nestrov

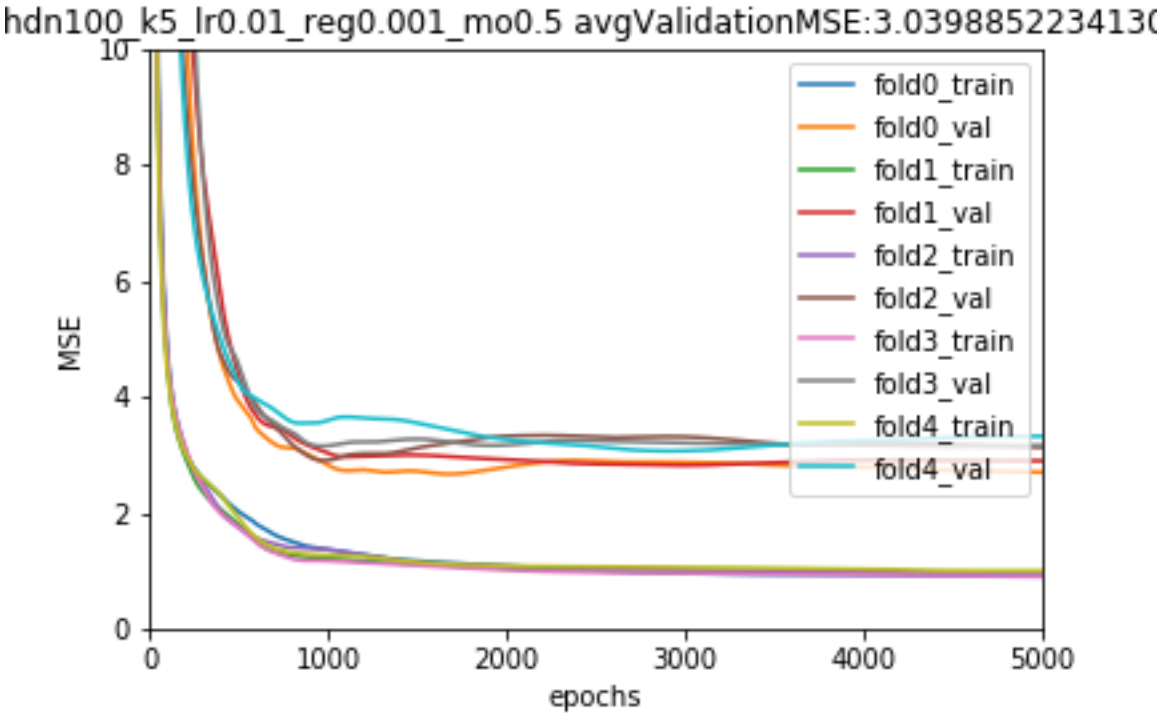
hdn100_lr0.005_reg0.001_mo0.5

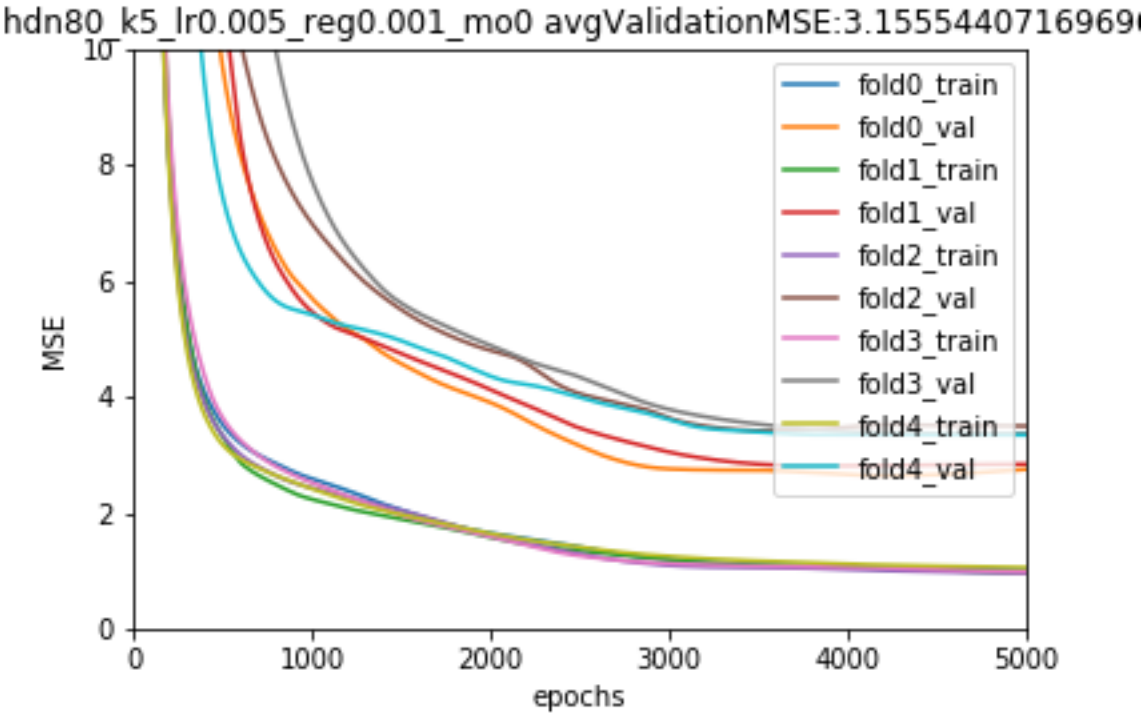
hidden 100
LearningRate 0.005
Regularization 0.001
Momentum 0.5
SGD+Momentum [Classic Momentum] [No Nesterov] top 20 experiments report:

33	hdn20_k5_lr0.005_reg0.001_mo0	2.928588
15	hdn80_k5_lr0.01_reg0.001_mo0.5	2.969773
27	hdn40_k5_lr0.01_reg0.001_mo0	2.976110
39	hdn20_k5_lr0.01_reg0.001_mo0.5	2.982251
5	hdn100_k5_lr0.005_reg0.001_mo0.5	3.006829
23	hdn60_k5_lr0.01_reg0.001_mo0.5	3.007029
29	hdn40_k5_lr0.005_reg0.001_mo0.5	3.011795
25	hdn40_k5_lr0.005_reg0.001_mo0	3.011973
35	hdn20_k5_lr0.01_reg0.001_mo0	3.017527

21	hdn60_k5_lr0.005_reg0.001_mo0.5	3.032281
37	hdn20_k5_lr0.005_reg0.001_mo0.5	3.033151
7	hdn100_k5_lr0.01_reg0.001_mo0.5	3.039885
11	hdn80_k5_lr0.01_reg0.001_mo0	3.040099
13	hdn80_k5_lr0.005_reg0.001_mo0.5	3.040472
1	hdn100_k5_lr0.005_reg0.001_mo0	3.049197
19	hdn60_k5_lr0.01_reg0.001_mo0	3.069511
17	hdn60_k5_lr0.005_reg0.001_mo0	3.088708
9	hdn80_k5_lr0.005_reg0.001_mo0	3.155544
31	hdn40_k5_lr0.01_reg0.001_mo0.5	3.280586
3	hdn100_k5_lr0.01_reg0.001_mo0	3.292397







We pick hdn80_k5_lr0.01_reg0.001_mo0.5 based on the same previous criteria to train on the train 60% cross validation 40% splits

c. Final Models

Here we report the Final Models selected from the previous k-fold validation strategy to be trained on the 60% training 40% validation splits for a final cross validation to choose the CUP challenge model

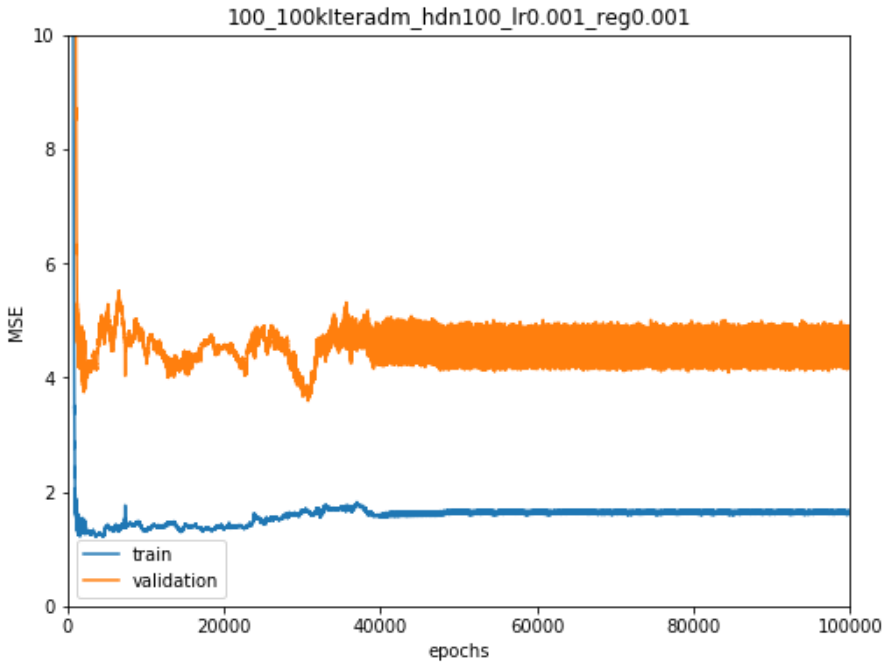


Figure showing the learning curve of the model trained with adam hdn100_lr0.001_reg0.001
Validation MEE:1.498 MSE:3.897

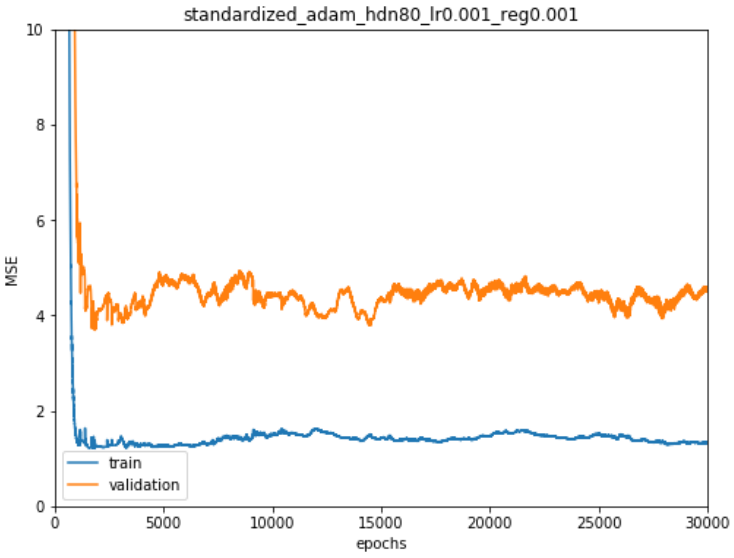


Figure showing the learning curve of the model trained with adam hdn80_lr0.001_reg0.001
Validation MEE:1.669 MSE:4.510

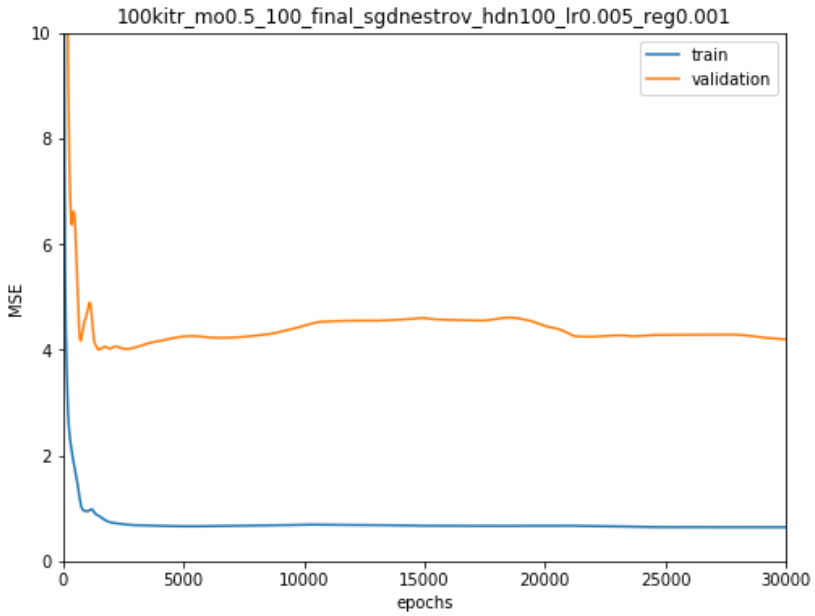


Figure showing the learning curve of the model trained with SGD+Nestrov
hdn100_lr0.005_reg0.001_m00.5
Validation MEE:1.614 MSE:4.356

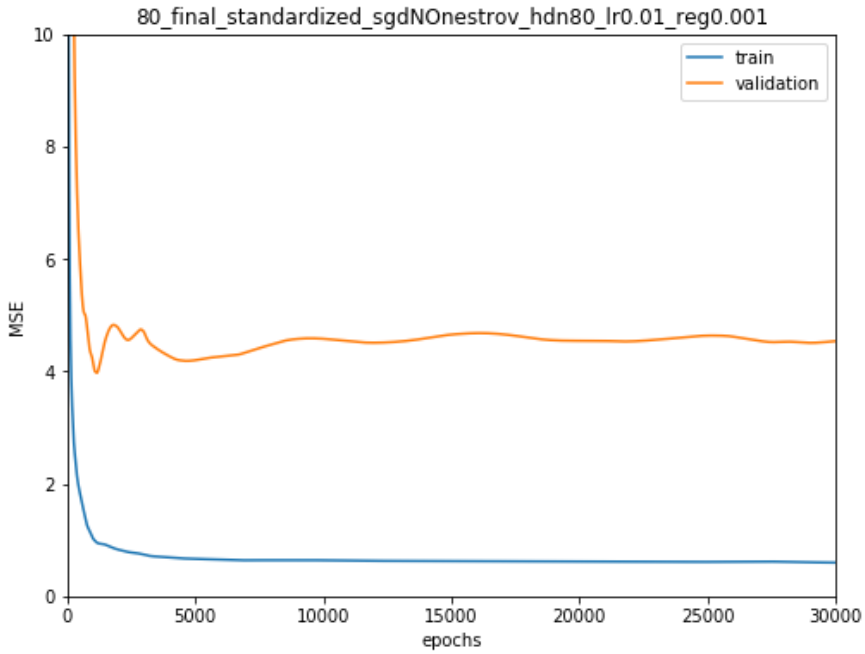
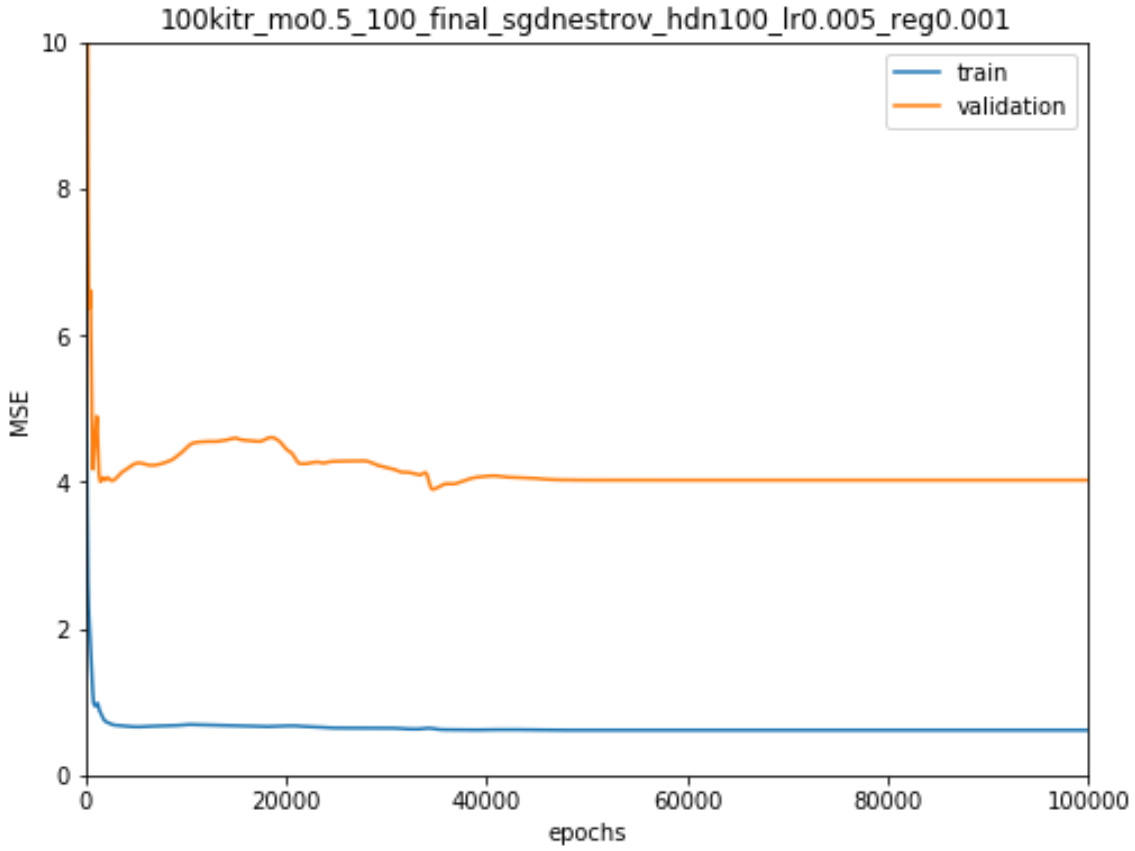


Figure showing the learning curve of the model trained with sgd+Classic
hdn80_lr0.01_reg0.001_m00.5
Validation MEE:1.635 MSE:4.537

Validation	adam hdn100_lr0.001_rego.001	adam hdn80_lr0.001_rego.001	SGD+Nestrov hdn100_lr0.005_rego.001_m00.5	sgd+Classic hdn80_lr0.01_rego.001_m00.5
MEE	1.498	1.669	1.614	1.635
MSE	3.897	4.510	4.356	4.537

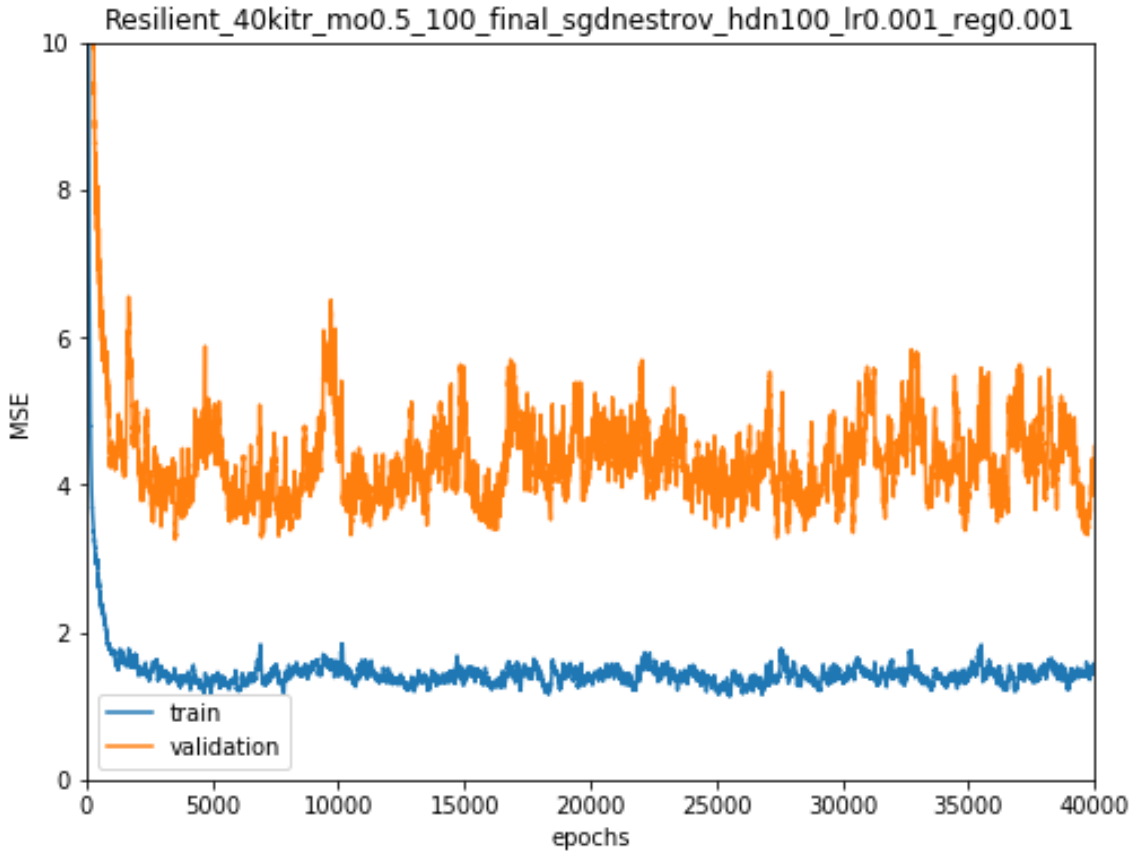
So, we think the model trained with SGD+Nestrov hdn100_lr0.005_rego.001_m00.5 has the highest potential, we retrained it for 100k epochs on the same dataset



And it seems to have reached a stable result with validation
Validation MEE:1.547 MSE:4.023

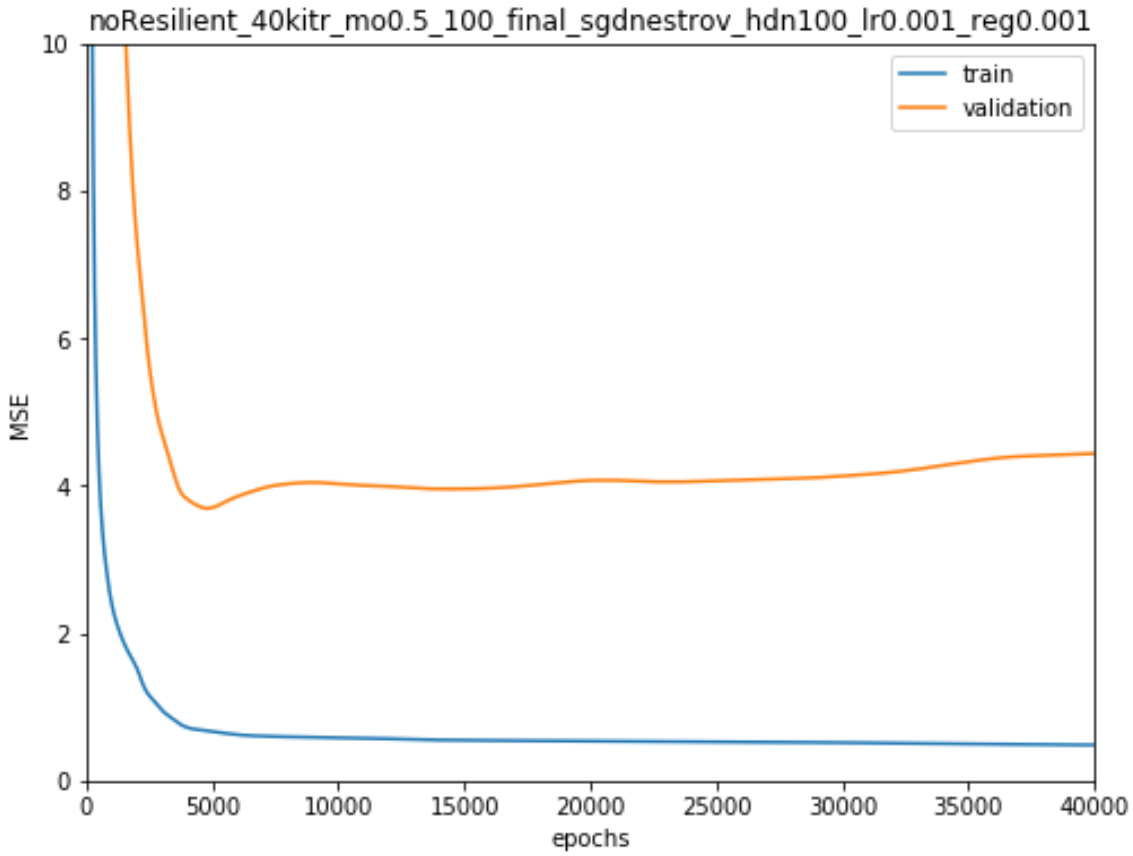
It gets stable after ~35k epochs maintaining the lowest validation MEE and MSE loss reported so far, so, we decided to use it for the final challenge.

A feature we tried to introduce to speedup the learning process, resilient weights updates, simply use a learning rate of learningRate*2 if the sign of the previous gradient is the same as the current one and learningRate/2 if the sign is different, mimicking the idea of speeding up when we get a similar direction as previous and slowing down if the the direction changes, however, it wasn't as successful as advertised, unfortunately, thus, not used for the final training.



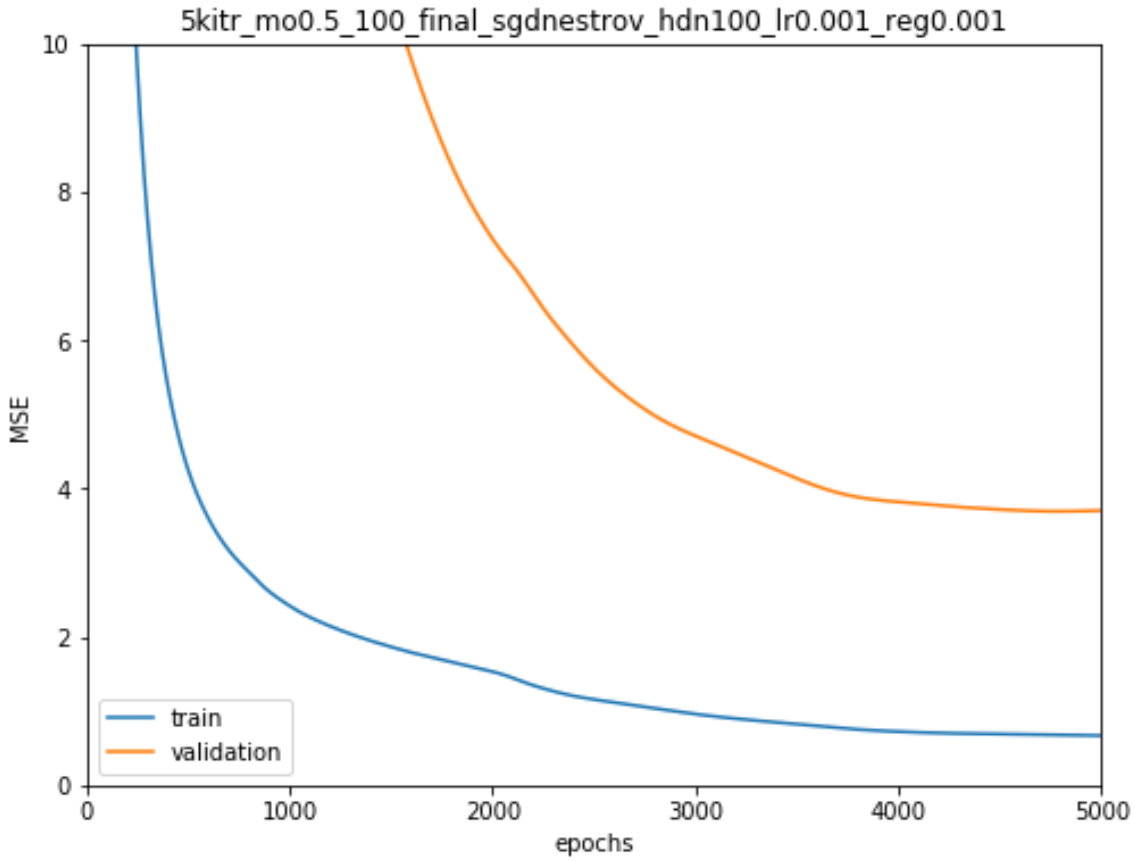
Same model with resilient, unfortunately, it is unstable
MEE:1.6319 MSE:4.515

We also experimented a little with the data preprocessing, the following is the learning curve with data standardization instead of normalization:



MEE:1.569 MSE:4.435

The curve is even smoother with the best MSE around 5k epochs, so, for the final challenge we are only training for 5k epochs



Trained again but til only 5k epochs
MEE:1.462 MSE:3.701

d. Comparison Against an off-the-shelf Library

One last comparison is introduced with off-the-shelf solutions using the same 60% 40% train/validation splits:

For the same model with the same stochastic gradient descent trainer

Learning rate = 0.001
Momentum[Nestrov] rate =0.5
L2 Regularization rate=0.001
number of hidden neurons = 100
Number of Epochs = 5000
Batch size =10

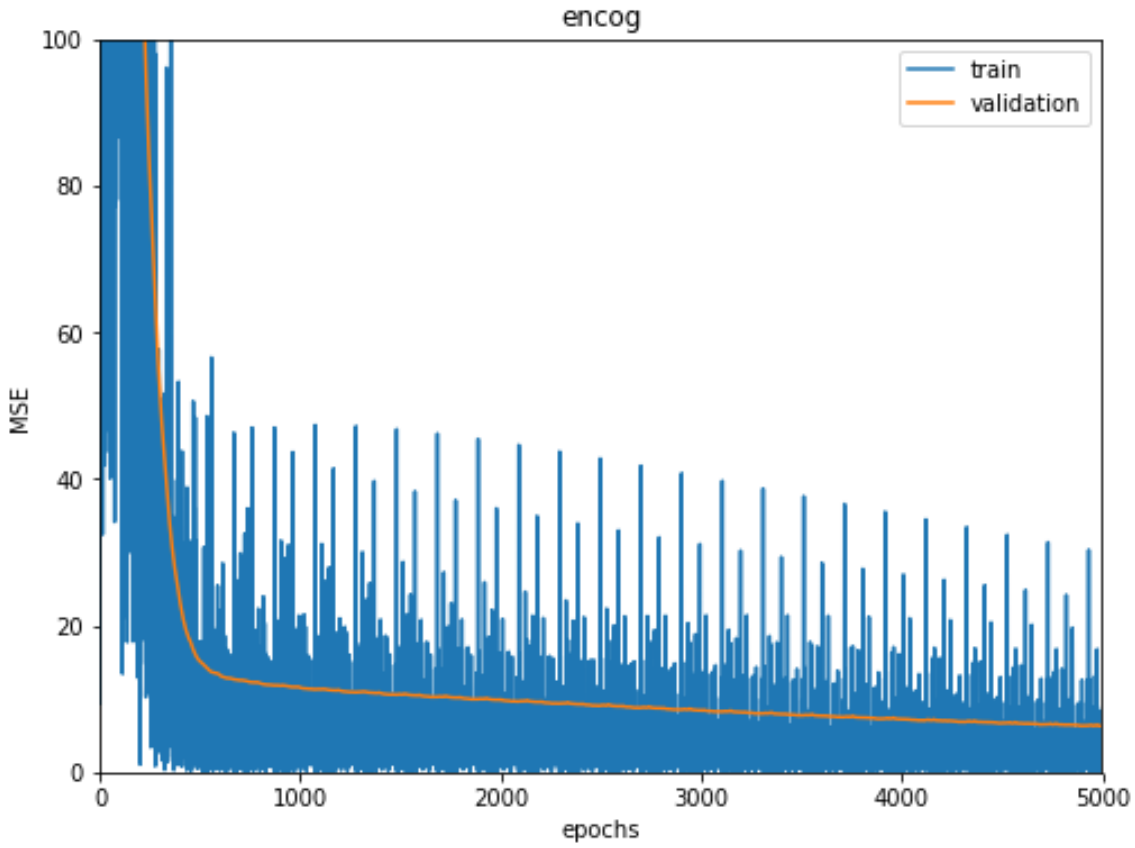


Figure Showing Encog learning curve on the same dataset, unfortunately, it is very unstable with the result Validation MEE2.153, MSE6.321

	Time	**Validation MEE**
Encog's	27386 ms	2.153
Ours	1000192 ms	1.462

Possible main reason for this difference in the MEE results aside from the implementation is the weights initialization and the random seed used.

In terms of speed, definitely their code is better optimized, in this project we didn't focus on code speed.

To reproduce similar results, please run AA1_Encog.program.cs with the correct paths to the standardized 60% training and 40% test datasets

2) LINEAR LEAST SQUARES

The second task was to provide a basic version of the linear least squares problem formulation and solver of our choice.

We provide the following three solvers:

a. Normal equations

As seen in the lecture, the simplest solver with the cheapest cost despite its instability, we introduced a simple implementation for a normal equations solver to our problems

b. SVD

A more robust, but also costly solver

c. Plain Gradient Descent

The plain gradient descent algorithm just updating the weights of the linear model with a fixed step as a learning rate against the gradient of the loss of the model without a line search.

Following we provide a comparison of the results acquired by the three solvers on both of the introduced problems

The Following is a performance comparison between the three solutions for the CUP problem:

	Time	Validation MEE
SVD solution	72 ms	18.221
Normal Equations Solution	30 ms	18.221
Gradient Descent Solution	862 ms for 1k iterations with a degree of 1, bias and learning rate of 0.1	2.214

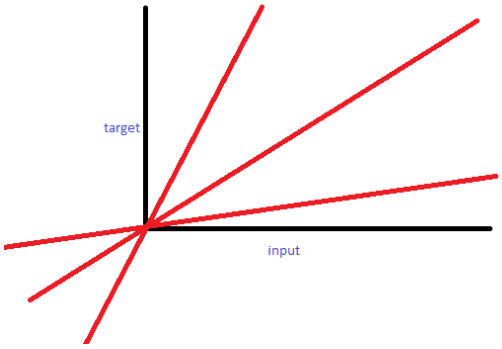
Table 7 – The performance comparison between the three solutions

To reproduce the results, please run `CupTestingLLS` in MLPTestDemo.Program.cs with the correct path to the train and test splits.

To obtain the data splits, please run the python notebook "SikitLearnLLS.ipynb" up to cell 7 inclusive.

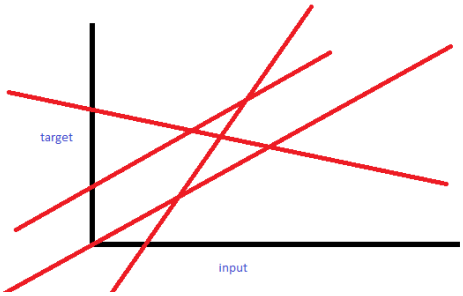
From these experiments provided, we believe the iterative solution seen in the Optimization lessons is a clear winner, because it benefits from a bias term as well, it is able to find a much better solution than all of the others, despite its drawback of the computational cost.

Why is that?
In a simple 2D example as shown in the following plot:



the iterative solution would be fitting a [red] line that passes by the data points and through the origin where it tries to find the perfect slope for it representing the hypothesis ax=t [a is the weight of the model, x is the input feature and t is the corresponding output] and trying to find the best 'a' as weight, exactly the same as the exact solutions in the optimal case.

But with a bias, it actually also has the freedom not to pass by the origin where all of the input features are zero and go up or down on the y axis to find an even better solution with the hypothesis ax+b=t where b is the bias value

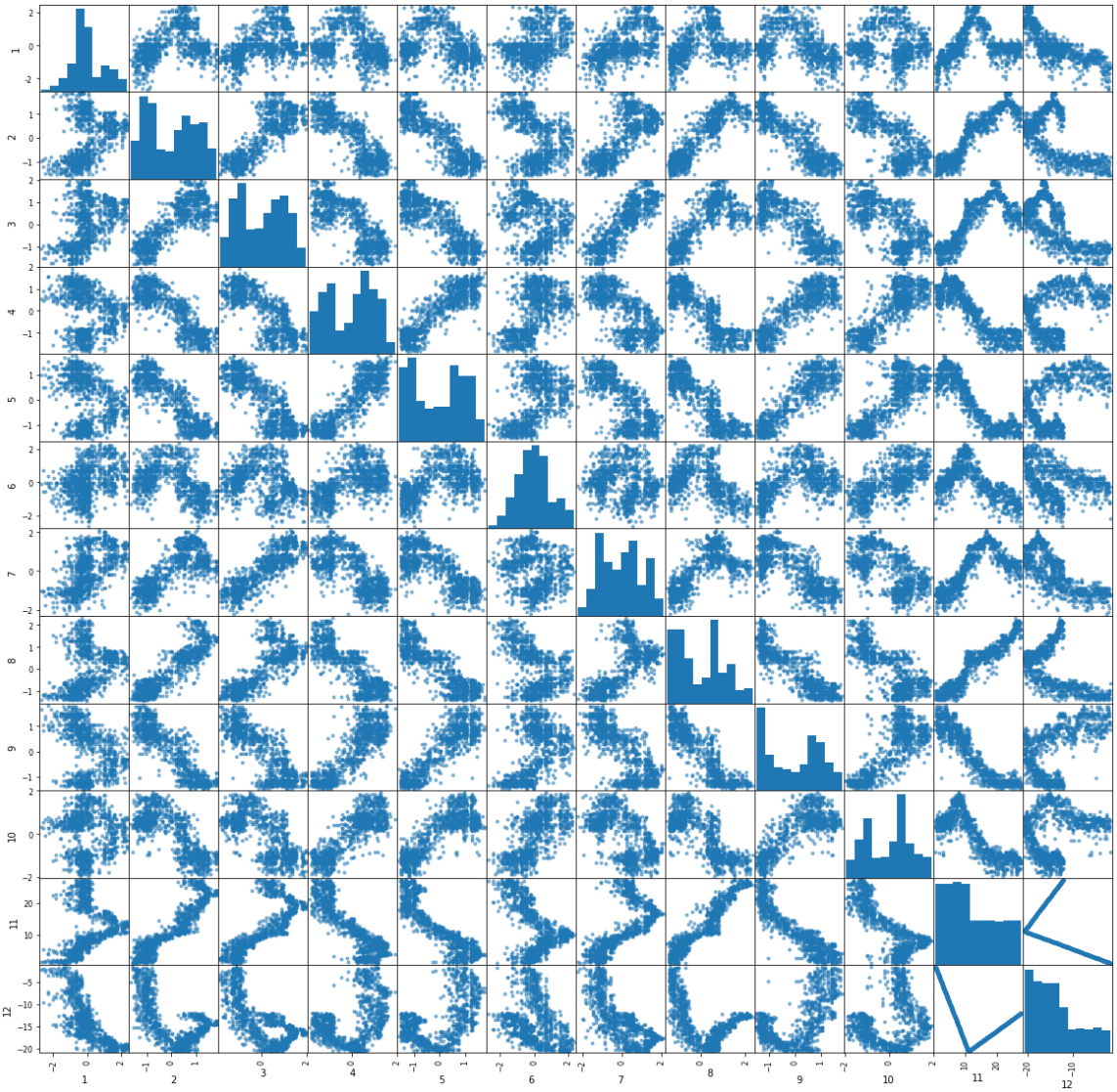


So, the bias does not depend on any of the input features allowing the hyperplane described by our hypothesis to fit data that doesn't pass through the origin.
And if it is not needed, the algorithm will just learn to set it to zero returning to the first case.

3. On the capacity& efficiency comparison of the solutions

To answer these points, we first conducted a few steps to help understand the nature of the provided data in the notebook DataUnderstanding.ipynb

Perhaps the most important step is the following figure:



Showing the correlations between the 10 inputs among themselves and also the two outputs.

Any model we use shall capture the correlation between the inputs and the outputs which is mostly non-linear, making any non-linear model in a complete disadvantage!

Since, unlike the linear model, MLPs can better capture such relations, perhaps it is already a clear winner and we have already seen that in the report results in the previous sections of the report.

	Time	Validation MEE
LLS: SVD solution	72 ms	18.221
LLS: Normal Equations Solution	30 ms	18.221
LLS: Gradient Descent Solution	4469 ms for 5k iterations with a degree of 1 and learning rate of 0.1	2.214
MLP: SGD+Nestrov Momentum	1000192 ms 100 hidden neurons, .001 learning rate and regularization rate, momentum 0.5 5k iterations	1.462

Table showing The performance comparison between the LLS and MLP solutions

From this we conclude that in terms of the model capacity, ability to find a good solution and generalization, MLPs are a clear winner, but in terms of computational cost, LLS will be favored

4. LLS implemented vs Off-the-shelf

Unfortunately, LLS is not available in C# off-the-shelf, however, we provide here a comparison between our implementation and that of scikit-learn.org

	Time	Validation MEE
Our LLS+SVD	73 ms	18.221
Scikit-learn LLS+SVD	231 ms	18.221
Our LLS+GradientDescnet	862 ms for 1k iterations with a degree of 1 and learning rate of 0.1	2.214
Scikit-learn LLS+GradientDescnet [solver='sag']	19 ms for 1k iterations with a degree of 1 and learning rate of 0.1	18.221

Table 8 – The performance comparison between ours and that of scikit-learn.org

For reproducing the results:

1- Run the python notebook "SikitLearnLLS_MEE_SavingTrainDataStandardized.ipynb"
We split the training set of the CUP challenge into two subsets, 60% training and 40% testing, save the data splits to be used by C# as well
And the book forms the problem and solves it with SVD and Gradient Descnet
2- Call the function `CupTestingLLS` in MLPTestDemo.Program.cs with the correct path to the train and test splits.

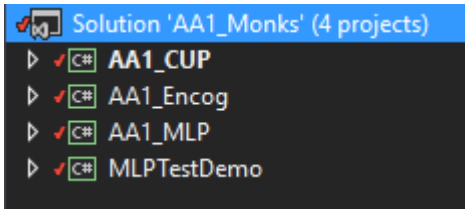
5. Implementation

In this section, we are glad to introduce our implementation of the work done in the form of a lightweight library with several features and possibility to extend.
The whole work is available on the following Git repository:
https://github.com/lilanpei/Monks

1) FEATURES

1. Possibility to build, train and test Multilayer perceptrons with different settings;
2. Number of layers, number of neurons per layer
3. Different activation functions: Sigmoid, Tanh, Relu
4. Different weight initialization mechanisms: Uniform, Xavier, He
5. MLP Trainers: Stochastic Gradient Descent + Momentum [classic and nesterov],Adam Optimizer, L2 regularization
6. Possibility to build and train a linear regression model
7. Data managers and interfaces to help handle regression and classification datasets
8. A solution to help screening the experiments and decide the final hyper parameters to choose the best model for the problem being solved
9. And most importantly, the ability to extend and add to the library more solvers, activations, or even different machine learning algorithms later by implementing our interfaces

2) SOLUTION PROJECTS



The projects implemented in the solution

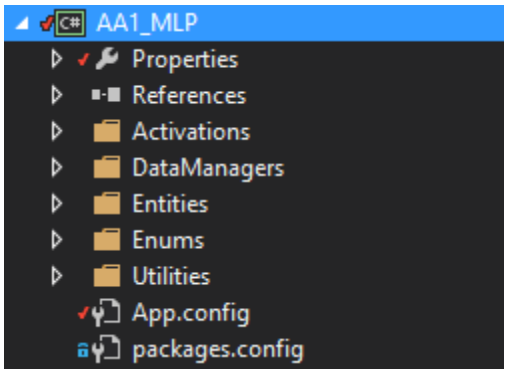
We have mainly four projects as follows:

1. AA1_CUP is a reusable project for screening a model
2. AA1_Encog is a project used for comparing our selected model against an off-the-shelf library, namely; Encog by heatonresearch.com
3. AA1_MLP is a project for building, training and testing MLPs and Linear regression models
4. MLPTestDemo is a project for testing the features of the models, trainers and the introduced settings in AA1_MLP, we used extensively this project with the Monk's problem datasets to build the features of AA1_MLP

Our sincere apologies for the bad namings [developers issues ☺]

The following are more details about each project and its usage:

AA1_MLP

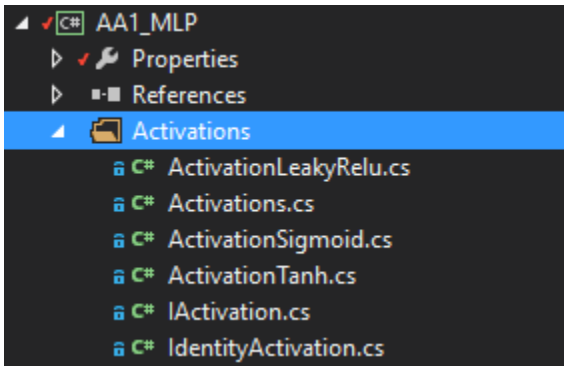


The main dish on our menu here☺

Activations is a folder providing activation function of MLPs
DataMangers is for providing managers to load training and testing datasets
Entities provides the models implementations along with their solvers/optimizers
Enums aggregates our used enums
Utilities holds our own extension methods for C# along with utilities and tools to facilitate the development process and the usage of the code

The following are more details on each folder of the AA1_MLP project

Activations



IActivation.cs provides and interface that must be implemented by any activation function desired with two functions:
CalculateActivation:

```
        Vector<double> CalculateActivation(Vector<double> x);
```

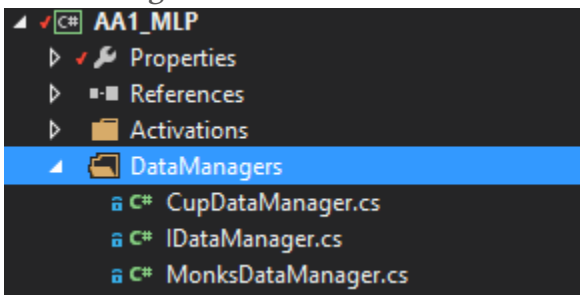
Given a vector of doubles x, applies the activation in a pointwise fashion and returns the result
And Calculate Derivative

```
        Vector<double>  CalculateDerivative(Vector<double> x);
```

Applies the derivative of the activation in a pointwise fashion on the input vector

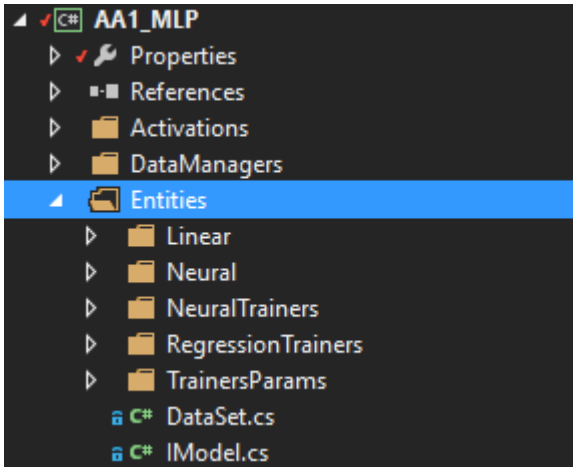
ActivationLeakyRelu.cs is a simple leaky relu activation function [x > 0 ? x : 0.01 * x]
Activations.cs is an enum listing all of the activations [currently unused, but for future refactoring we intend to utilize it]
ActivationSigmoid.cs is the sigmoid activation function
ActivationTanh.cs is the tanh activation function
IdentityActivation.cs is an identity activation function given x, return x, this is useful for layers without activation like the input layer in our case

DataManagers



A data manager is a class handling the loading and parsing of dataset files, it must provide an implementation for the IDataManger.cs interface function LoadData.
In the DataManagers folder, we provide our data managers for cup and monk's datasets, handling the parsing from the provided csv files and encoding the monks datasets into categorical features

Entities

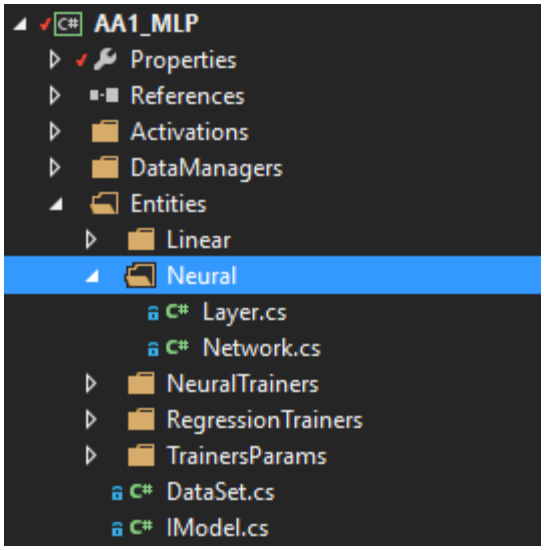


Contains our implementations for Linear and Neural models along with their trainers and the parameters required per trainer.
DataSet.cs is a class representing a dataset with two properties; 1-Inputs, a matrix of doubles, each row in it represents one example in the dataset provided, 2- Outputs, also a

matrix of doubles representing the corresponding targets per example
IModel.cs is a serializable abstract class providing a single function "Predict", taking a vector of inputs [a row in the dataset for an example] and returns a vector of doubles representing the corresponding predicted output,
Any Model provided must implement the IModel.cs abstract class.

Following are more details on the folders contained in Entities

Entities.Neural



The subfolder "Neural" contains our implementations for Layer.cs a MLP layer and Network.cs a MLP.
Layer.cs contains the following properties:

```
int NumberOfNeurons { get; set; }
IActivation Activation { get; set; }
bool Bias { get; set; }
Vector<double> LayerActivationsSumInputs { get; set; }//The inputs to the layer's
neurons
Vector<double> LayerActivations { get; set; }//the output from the layer's neurons
after applying the activation on LayerActivationsSumInputs
Vector<double> Delta { get; set; }//layer local error
```

And it also provides the implementation of a forward propagation function that takes

```
public Vector<double> ForwardPropagation(Vector<double> inputOfPrevLayer,
Matrix<double> weights, bool debug = false)
```

```
"inputOfPrevLayer" the output of the previous layer, before multiplying the
weights between the two layers with it
"weights"the weights between the current layer and the previous one
```

The function multiplies the inputOfPrevLayer column vector by the weights matrix and sets the result to LayerActivationsSumInputs
Then applies the activation function on it and set the result in LayerActivations which is

returned by the function eventually.

Networks.cs has two properties:
A list of layers that will hold the layers of the network and a list of weight matrices that will hold the weights matrices between the layers of the network

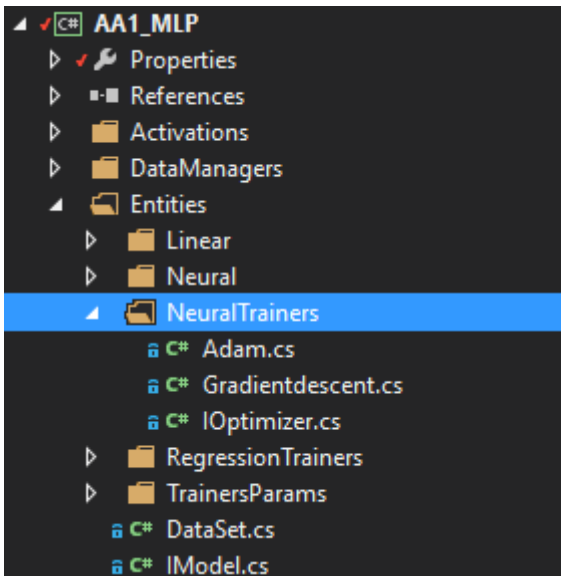
```
List<Layer> Layers { get; set; }
List<Matrix<double>> Weights { get; set; }
```

In the constructor of the Network, we provide three weight initialization mechanisms 1- Uniform, initializes the weights by values drawn from a uniform distribution between -0.7 to 0.7
2- Xavier, initializes the weights with values drawn from normal distributions with 0 mean and a variance of (2/fanIn+fanOut) where FanIn and FanOut are the number of incoming connections to the layer's neurons and the number of leaving connections from each neuron
3- He, similar to Xavier but with only fainIn considered.

In both of the given problems a simple uniform initialization should be sufficient since the architecture we provided for the network is definitely too shallow.

As the Network represents a model, it inherits the IModel abstract class and provides an implementation for its "predict function" which simply passes the signal of the input through the network and provides the network output.

Entities.NeuralTrainers



Holds the implementations for our optimizers,
An Optimizer algorithm must inherit the IOptimizer.cs abstract class and provide the implementation to its function Train

```
public abstract List<double[]> Train(TrainerParams trainParams);
```

Also, it needs to provide a TrainerParams class representing the required parameters for the algorithm [more in the TrainerParams is introduced shortly in the TrainersParams

folder]
The Train function returns a list of arrays of doubles that will hold the learning curve of the training process

Adam.cs
Provides an implementation of the Adam optimizer, our chosen algorithm of the class of accelerated gradient methods
Quoting from [3]

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

```
//The simple implementation of adam at the beginning of the professor's paper
weightsUpdates[y] = weightsUpdates[y] / numberOfBatchExamples;
firstMoment[y] = passedParams.beta1 * prevFirstMoment[y] + (1 -
passedParams.beta1) * (-1 * weightsUpdates[y]);
secondMoment[y] = passedParams.beta2 * prevSecondMoment[y] + (1 -
passedParams.beta2) * weightsUpdates[y].PointwisePower(2);
mhat[y] = firstMoment[y] / (1 - Math.Pow(passedParams.beta1, adamUpdateStep));
vhat[y] = secondMoment[y] / (1 - Math.Pow(passedParams.beta2, adamUpdateStep));
var finalUpdates = (passedParams.learningRate *
mhat[y]).PointwiseDivide((vhat[y].PointwiseSqrt() + passedParams.epsilon));
passedParams.network.Weights[y] -= finalUpdates;
```

Gradientdescent.cs
Provides our implementation of the stochastic gradient descent + Momentum both the classic [new weights Updates = momentum rate* old weights updates +learning rate the weights gradient] [4]

$$v_{t+1} = \mu v_t - \varepsilon \nabla f(\theta_t) \quad (1)$$

$$\theta_{t+1} = \theta_t + v_{t+1} \quad (2)$$

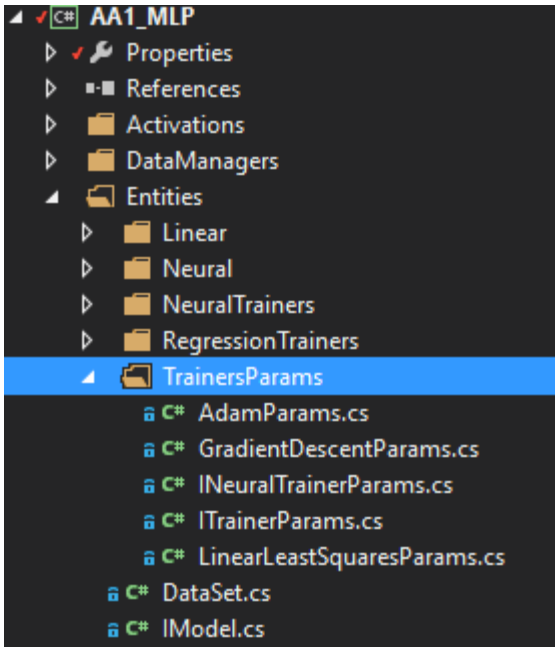
and Nestrov [new weights Updates = momentum rate* old weights updates + learning rate*the weights gradient at the updated weights][4]

$$v_{t+1} = \mu v_t - \varepsilon \nabla f(\theta_t + \mu v_t) \quad (3)$$
$$\theta_{t+1} = \theta_t + v_{t+1} \quad (4)$$

Where v_t is the weights updates at step t and Theta_t are the weights at step t
mu is the momentum weight and epsilon is the learning rate.

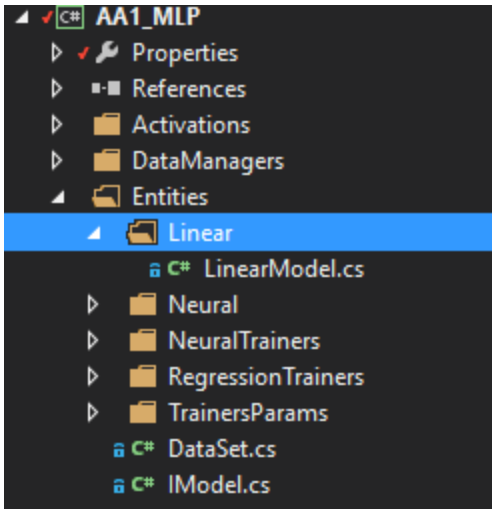
With the possibility of using an L2 regularization by penalizing the weights with 2*regularization rate * current weights

Entities.TrainersParams



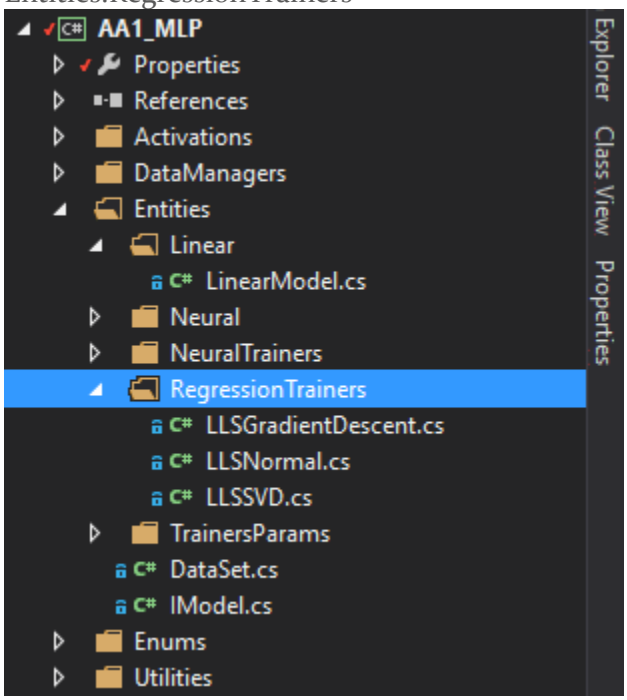
Holds the trainers parameters needed to pass around a trainer's hyperparameters, each provided trainer needs to have a class for its parameters inheriting the TrainerParams class and adding any additional parameters required by the optimizer
We see also a Params.cs for each solver or trainer we have where ITrainerParams is inherited by the LinearLeastSquaresParams.cs for the linear models params and INeuralTrainerParams.cs is inherited by the MLP optimizers params[AdamParams.cs and gradientDescnetParams.cs]

Entities.Linear



Contains the LinearModel.cs a simple linear model inheriting IModel and implementing its function predict, also providing properties required by a LinearModel
Weights is the weights [single row]matrix of the model
Degree, a property we introduced to provide a Polynomial regression model,
Set to higher than one, while using an LLSGradientDescnt solver, will utilize higher order features in the model, for example set to two the features and each feature squared will be considered, set to three, the features, the squared features and the cubed features will be considered

Entities.RegressionTrainers



Holds our implementation for the linear regression solver of our choice, we provide three solvers

LLSGradientDescent.cs is the plain vanilla gradient descent, with a learning rate parameters as a step without line search, or any other hyperparameters
LLSNormal.cs a simple normal equations solver

```
//Assuming the data matrix A has linearly independent rows or columns to properly use the pseudo inverse otherwise we might have a problem!
//Ax=b  ->   A'Ax=A'b -> x = (inv(A'A))A'b ->pinv(A)b  where A is our data, b is our vector of targets
```

LLSSVD.cs a simple SVD solver;
Quoting from the CM lecture Linear Algebra part [1]

Least squares with the SVD

One can solve least-squares problem also with (thin) $A = USV^T$.
Same derivation as with QR:

$$\|Ax - b\| = \|USV^T x - b\| = \|S\underbrace{V^T x}_{=y} - U^T b\|$$

$$= \left\| \begin{bmatrix} \sigma_1 y_1 \\ \sigma_2 y_2 \\ \vdots \\ \sigma_n y_n \\ 0 \\ \vdots \\ 0 \end{bmatrix} - \begin{bmatrix} u_1^T b \\ u_2^T b \\ \vdots \\ u_n^T b \\ u_{n+1}^T b \\ \vdots \\ u_m^T b \end{bmatrix} \right\|$$

If all the σ_n are different from 0, the minimum is when $y_i = \frac{u_i^T b}{\sigma_i}$ (and then $x = Vy$).

Putting everything together, one gets

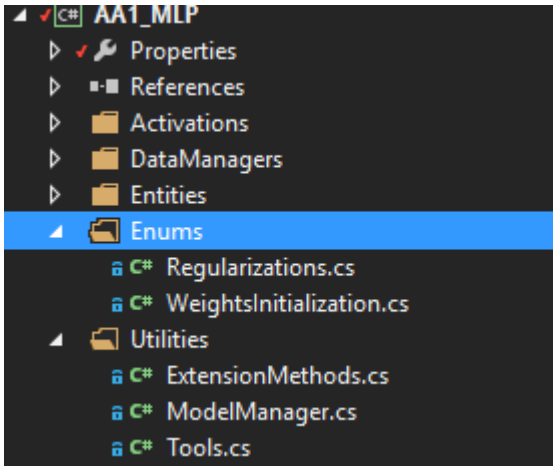
$$x = \sum_{i=1}^{n} v_i \frac{u_i^T b}{\sigma_i}.$$

Again, we need only the thin SVD to compute it.

Note that the small σ_i's contribute more to the solution (unless also $u_i^T b \approx 0$).

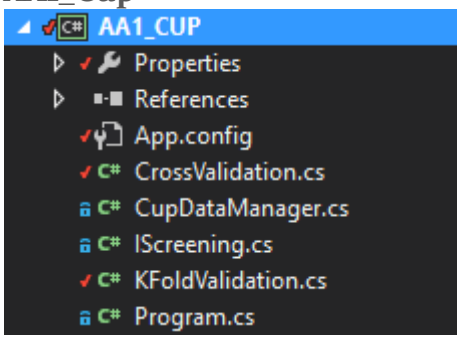
```
var svd = trainParams.trainingSet.Inputs.Svd();
int r = trainParams.trainingSet.Inputs.Rank();
var d = svd.U.Transpose().Multiply(trainParams.trainingSet.Labels);
var temp = CreateMatrix.Dense<double>(svd.VT.ColumnCount, 1);
temp.SetSubMatrix(0, 0, d.SubMatrix(0, r, 0,
1).Column(0).PointwiseDivide(svd.S.SubVector(0, r)).ToRowMatrix().Transpose());
passedParams.model.Weights = svd.VT.Transpose().Multiply(temp);
```

Enums and Utilities



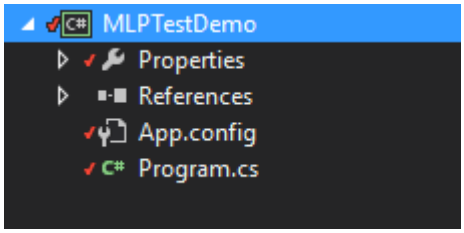
Enums holds holds our enums for Regularizations.cs [either L2 or None for now]
WeightsInitialization.cs [He, Uniform and Xavier]
Utilities holds our extension methods to C#, we found we needed a shuffle function for a C# List<> and a matrix to vector multiplication and a vector to matrix multiplication functions
Both returning a single row matrix as the result
Tools.cs provides a compute accuracy function for a model
ModelManager.cs to save and load a model

AA1_Cup



The AA1_CUP project is provided mainly to screen a model by running experiments using train/validation/test utilizing CrossValidation.cs or k-fold cross validation utilizing KFoldValidation.cs
Any validation strategy needs to implement the IScreening.cs interface and provide an implementation to the "Screen" function.
CupDataManager.cs is a Data Manger implementing the IDataManager.cs abstract class in AA1_MLP providing the body for the LoadData function
Program.cs is the tradition C# class with a main function to run code in a console app utilizing the screening mechanisms implemented

MLPTestDemo



Program.cs is providing the main function doing several tests using the different models we had on the monk's datasets, we utilized this project mainly for building our library.

3) EXAMPLES

In this section we introduce some examples on how to reproduce the results of the experiments done so far.

Plotting a cross section of the Cost function to prove or disprove its convexity.

We load the dataset

```
AA1_MLP.DataManagers.CupDataManager dm = new
AA1_MLP.DataManagers.CupDataManager();
DataSet trainDS = dm.LoadData(dsLocation, 10, 2, standardize: true);
```

```
Fix an initial set of weights, simply by defining a MLP with random initial
weights
```

```
var nOriginal = new Network(new List<Layer>() {
                    new Layer(new ActivationIdentity(),true,10),
                    new Layer(new ActivationTanh(),true,100),
                    new Layer(new ActivationIdentity(),false,2),
                    }, false, AA1_MLP.Enums.WeightsInitialization.Xavier);
```

We pick 100 random directions [random sets of very small weights] each time plotting the function along the picked direction[with a range of -10 to 10 and a step of 0.01,these numbers are acquired empirically by trial and zooming on the best range for the plots to show non-convexity]

```
for (int i = 0; i < 100; i++)
            {
                Console.WriteLine(i);

List<double[]> weightValVsCost = GeneratePlot(trainDS, nOriginal, -10, 10, 0.01);

File.WriteAllLines(@"xcurve" + i + ".txt", weightValVsCost.OrderBy(s =>
s[0]).Select(x => string.Join(",", x)).ToArray());
            }

            Console.WriteLine();
```

Then we plot the generated file with the MSE on the y axis and the line step on the x axis

The function GeneratePlot in the same file is quite hacky, we build a network n that we don't really care about its weights values, it is a simple container for the weights values, so that we don't change the original initial point

```
            //building the architecture
            Network n = new Network(new List<Layer>() {
                     new Layer(new ActivationIdentity(),true,10),
                     new Layer(new ActivationTanh(),true,100),
                   //   new Layer(new ActivationLeakyRelu(),true,40),

                     new Layer(new ActivationIdentity(),false,2),
                     }, false, AA1_MLP.Enums.WeightsInitialization.Xavier);
```

We build a matrix with weights of very small random values representing the random direction in the weights space

```
            Network nRand = new Network(new List<Layer>() {
                     new Layer(new ActivationIdentity(),true,10),
                     new Layer(new ActivationTanh(),true,100),
                   //   new Layer(new ActivationLeakyRelu(),true,40),

                     new Layer(new ActivationIdentity(),false,2),
                     }, false, AA1_MLP.Enums.WeightsInitialization.Normal);
```

We add step*direction Network weights nRand to the original initial weights and set them to the container network n
And compute the cost function of n at each step

```
            double MSE = 0, MEE = 0;
            List<double[]> weightValVsCost = new List<double[]>();
            for (double i = start; i <= end; i += step)
            {
                for (int layerIdx = 0; layerIdx < n.Weights.Count; layerIdx++)
                {
                    n.Weights[layerIdx] = nOriginal.Weights[layerIdx] + (i *
nRand.Weights[layerIdx]);

                }


                var log = ModelManager.TesterCUPRegression(trainDS, n, out MEE,
out  MSE);
                weightValVsCost.Add(new double[] { i, MSE });
            }
            return weightValVsCost;
```

Training a MLP with the implemented algorithms:

Simple usage example:
SGD+Momentum[Nestrov]

```
AA1_MLP.DataManagers.CupDataManager dm = new
AA1_MLP.DataManagers.CupDataManager();
DataSet trainDS = dm.LoadData("D:\\dropbox\\Dropbox\\Master Course\\SEM-
3\\ML\\CM_CUP_Datasets\\60percenttrain.txt", 10, 2, standardize: true);
DataSet testDS = dm.LoadData("D:\\dropbox\\Dropbox\\Master Course\\SEM-
3\\ML\\CM_CUP_Datasets\\60percenttest.txt", 10, 2, standardize: true);
GradientDescentParams passedParams = new GradientDescentParams();
Gradientdescent trainer = new Gradientdescent();
passedParams.numberOfEpochs = 5000;
passedParams.batchSize = 10;
passedParams.trainingSet = trainDS;
passedParams.validationSet = testDS;
passedParams.learningRate = 0.001;
passedParams.regularization = Regularizations.L2;
passedParams.regularizationRate = 0.001;
passedParams.nestrov = true;
passedParams.resilient = false;
passedParams.resilientUpdateAccelerationRate = 2;
passedParams.resilientUpdateSlowDownRate = 0.5;

passedParams.momentum = 0.5;
passedParams.NumberOfHiddenUnits = 100;

LastTrain(testDS, passedParams, trainer, "5kitr_mo0.5_100_final_sgdnestrov_hdn");
```

And Adam:

```
AA1_MLP.DataManagers.CupDataManager dm = new
AA1_MLP.DataManagers.CupDataManager();
DataSet trainDS = dm.LoadData("D:\\dropbox\\Dropbox\\Master Course\\SEM-
3\\ML\\CM_CUP_Datasets\\60percenttrain.txt", 10, 2, standardize: true);
DataSet testDS = dm.LoadData("D:\\dropbox\\Dropbox\\Master Course\\SEM-
3\\ML\\CM_CUP_Datasets\\60percenttest.txt", 10, 2, standardize: true);
AdamParams passedParams = new AdamParams();
IOptimizer trainer = new Adam();
passedParams.numberOfEpochs = 5000;
passedParams.batchSize = 10;
passedParams.trainingSet = trainDS;
passedParams.validationSet = testDS;
passedParams.learningRate = 0.001;
passedParams.regularization = Regularizations.L2;
passedParams.regularizationRate = 0.001;
passedParams.NumberOfHiddenUnits = 100;

LastTrain(testDS, passedParams, trainer, "5kitr_mo0.5_100_final_sgdnestrov_hdn");
```

The function `LastTrain` takes a training setting and dumps the log history of the learning curve train and validation MSEs along with the validation MEE in a text file and also the trained model as well.

Parallel Code

The Boolean field `parallelize` in INeuralTrainerParams Allows for the usage of the parallelized version of the functions in the Adam and SGD+Momentum implementations.

The parallel implementations of the functions have “Parallel_” in the beginning of their names.

We made sure to lock the critical part of accumulating the weights updates and we utilized MS parallel for algorithmic skeleton to implement a Barrier design pattern[16] executing each training example on a different thread.

6. Provided Files and Folders

The main project is under the Folder Monks,
please open the solution from \Monks\AA1_Monks\ AA1_Monks.sln

Under Monks\UsedFiles exists the following:

Folder/File	Description
KFoldValidationPlots	the plots of the K-Fold validations reported in section 2>1 MLP
LastModelsPlots	the plots of the last models reported in 2>1>Final Models
ML-17-PRJ lecture package-20171225	the folder provided by prof. Miceli for the CUP [2]
Notebooks	the notebooks used in this project
TrainValSplits	the 60%40% train-validation splits both in regular form and standardized
OMG_ML-CUP17TS.csv	our CUP submission
libs	Where encog and math.NET .dll files live, please reference these files in the projects provided before running them
UsedFiles\FurtherExperiments\60Train40Validation	Contains further output files produced by our library

Our mainly supported files include

- the AA1 CUP challenge .csv files [2]
- .n files as output for saving our MLP models [just a serialization of the trained parameters of the model]
- .csv files representing the training and validation losses of a training experiment
- .png plots of the learning curves

In mode details:

Supported files are the same as the AA1 CUP challenge as input

Output files are CSV files with columns of learning loss, validation loss and the last line is the validation MEE and MSE, to corectly plot a file you output, please feel free to use your

plotting software of choice, for us, we wold use python notebooks available in usedFiles/Notebooks

https://github.com/lilanpei/Monks/tree/master/UsedFiles/Notebooks

A .n file is a trained model binary serialization saved using the ModelManager static method

ModelManager.SaveNetowrk

there is also a ModelManager.LoadNetwork to load a trained model in a .n file

examples of .n files are present in

UsedFiles\FurtherExperiments\60Train40Validation\TrainedModels

Examples of learning curves csv files are present in

UsedFiles\FurtherExperiments\60Train40Validation\LearningCurves

The other two zeros are trainging accuracy and validation accuracy, they are only available for classification problems such as the Monks, thus they are not used for this challenge.

some predicted vs actual comparison for some trained models are present in

UsedFiles\FurtherExperiments\60Train40Validation\PredictedVsActual

7. Future improvements

Moving the parameters from the train function into their respective trainers and keeping only the common parameters

Moving the enums into one folder

Adding SVM

Adding more advanced artificial neural network topologies, namely Convolutional Neural Networks and Recurrent Neural Networks

Parallel computation for better performance

Better names!!!

Adding a complete documentation to the code provided and generating the standard docs

Refactoring the screening project to allow for different models and algorithms without a custom function per each, this will be possible after moving the trainers parameters inside the trainers

8. Special: Code Profiling and improvements

In this section we make an attempt for profiling our code and seeing where most of the execution time is spent and try to find possible improvements to reduce the overall execution time.

First: SGD+Momentum

We refactored the code a bit than before moving Single example computation into a separate function and also the batch computations into a separate function to be able to see where the CPU time is spent.

We started with profiling the Train function implementation of SGD+Momentum and breaking it down to its components

Function Name	Total CPU [ms, %]
AA1_CUP.exe (PID: 1500)	33431 (100.00 %)
[External Code]	33431 (100.00 %)
AA1_CUP.Program::Main	18282 (54.69 %)
AA1_CUP.Program::LastTrain	18272 (54.66 %)
AA1_MLP.Entities.Trainers.Gradientdescent::Train	18172 (54.36 %)
AA1_MLP.Entities.Trainers.Gradientdescent::PerformBatchComputations	13792 (41.26 %)
AA1_MLP.Entities.Trainers.Gradientdescent::ComputeValidationLoss	4365 (13.06 %)

Most of the CPU utilization is consumed in its critical called functions:

1- PerformaBatchComputation where the training computation per batch are carried out
2- ComputeValidationLoss where the validation set is used at each step of the training

The ComputeValidationLoss is merely calling the predict function on each example in the loss and computing the validation MSE to report it during the training for each epoch

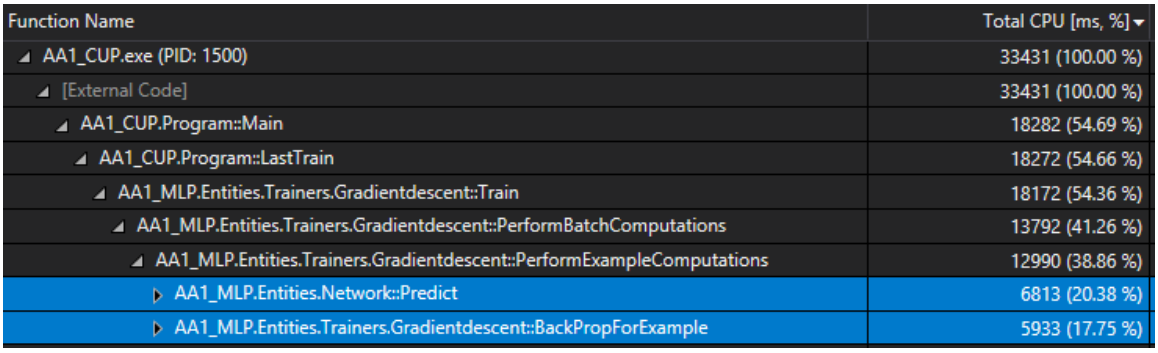
We go further in PerformBatchComputations

Function Name	Total CPU [ms, %]
AA1_CUP.exe (PID: 1500)	33431 (100.00 %)
[External Code]	33431 (100.00 %)
AA1_CUP.Program::Main	18282 (54.69 %)
AA1_CUP.Program::LastTrain	18272 (54.66 %)
AA1_MLP.Entities.Trainers.Gradientdescent::Train	18172 (54.36 %)
AA1_MLP.Entities.Trainers.Gradientdescent::PerformBatchComputations	13792 (41.26 %)
AA1_MLP.Entities.Trainers.Gradientdescent::PerformExampleComputations	12990 (38.86 %)
AA1_MLP.Entities.Trainers.Gradientdescent::UpdateWeights	736 (2.20 %)

Where we find most of the time spent is in the critical functions:

1- PerformExampleComputations
2- UpdateWeights

We go down further in PerformExampleComputations



Function Name	Total CPU [ms, %]
AA1_CUP.exe (PID: 1500)	33431 (100.00 %)
[External Code]	33431 (100.00 %)
AA1_CUP.Program::Main	18282 (54.69 %)
AA1_CUP.Program::LastTrain	18272 (54.66 %)
AA1_MLP.Entities.Trainers.Gradientdescent::Train	18172 (54.36 %)
AA1_MLP.Entities.Trainers.Gradientdescent::PerformBatchComputations	13792 (41.26 %)
AA1_MLP.Entities.Trainers.Gradientdescent::PerformExampleComputations	12990 (38.86 %)
AA1_MLP.Entities.Network::Predict	6813 (20.38 %)
AA1_MLP.Entities.Trainers.Gradientdescent::BackPropForExample	5933 (17.75 %)

In this function, most of the time is spent on the predict function where we do the forward propagation per example then in the BackPropForExample function where the computations for the error corrections are carried out.

A thought would be here to parallelize the computations per example in the batch, since they are actually independent.

Then we update the weights for the batch.

For our final model with a 100 epochs the elapsed time in the current implementation is:

elapsed Time: 10953 ms

After we tried parallelization it became:

elapsed Time: 6712 ms

Another place that can be parallelized is the ValidationLoss computations

And we get an elapsed time of:

elapsed Time:5280 ms

That is a speedup of 48%

The elapsed time for a 5k epoch roughly a 51% speedup:

Parallelized	Sequential
284146 ms	557968 ms

Same procedure is taken with Adam

And we acquire a speedup of 51% as well.

For parallelization usage and notes on the code added, please check the **Parallel Code**

In 3) Examples section under 5. Implementation.

9. Special2: Adamax

Another alternate for Adam proposed by the authors is instead of using a running second moment v, they used 'u' where[3]:

Note that the decay term is here equivalently parameterised as β_2^p instead of β_2. Now let $p \to \infty$, and define $u_t = \lim_{p\to\infty}(v_t)^{1/p}$, then:

$$u_t = \lim_{p\to\infty}(v_t)^{1/p} = \lim_{p\to\infty}\left((1-\beta_2^p)\sum_{i=1}^{t}\beta_2^{p(t-i)}\cdot|g_i|^p\right)^{1/p} \tag{8}$$

$$= \lim_{p\to\infty}(1-\beta_2^p)^{1/p}\left(\sum_{i=1}^{t}\beta_2^{p(t-i)}\cdot|g_i|^p\right)^{1/p} \tag{9}$$

$$= \lim_{p\to\infty}\left(\sum_{i=1}^{t}\left(\beta_2^{(t-i)}\cdot|g_i|\right)^p\right)^{1/p} \tag{10}$$

$$= \max\left(\beta_2^{t-1}|g_1|, \beta_2^{t-2}|g_2|, \ldots, \beta_2|g_{t-1}|, |g_t|\right) \tag{11}$$

Which corresponds to the remarkably simple recursive formula:

$$u_t = \max(\beta_2 \cdot u_{t-1}, |g_t|) \tag{12}$$

Claiming this resulted for them in a more stable algorithm:

Algorithm 2: *AdaMax*, a variant of Adam based on the infinity norm. See section 7.1 for details. Good default settings for the tested machine learning problems are $\alpha = 0.002$, $\beta_1 = 0.9$ and $\beta_2 = 0.999$. With β_1^t we denote β_1 to the power t. Here, $(\alpha/(1-\beta_1^t))$ is the learning rate with the bias-correction term for the first moment. All operations on vectors are element-wise.

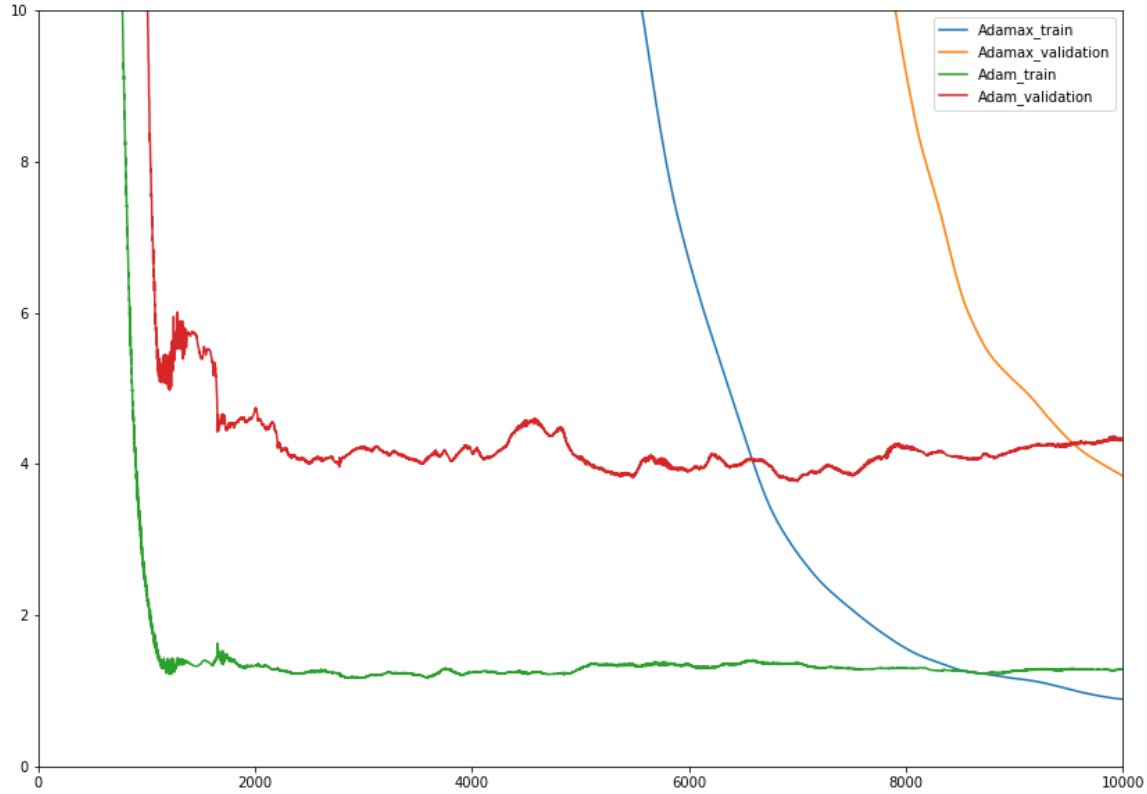
Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $u_0 \leftarrow 0$ (Initialize the exponentially weighted infinity norm)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1-\beta_1) \cdot g_t$ (Update biased first moment estimate)
    $u_t \leftarrow \max(\beta_2 \cdot u_{t-1}, |g_t|)$ (Update the exponentially weighted infinity norm)
    $\theta_t \leftarrow \theta_{t-1} - (\alpha/(1-\beta_1^t)) \cdot m_t/u_t$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

We implemented it and ran an experiment with the following settings:

```
number Of Epochs = 10000;Batch Size = 10;Learning Rate = 0.001;regularization =
Regularizations.L2;regularization Rate = 0.001;Number Of Hidden Units = 100;
Parallelization =False;
Trained over the 60%-40% train/test dataset with both Adam and Adamax
We acquired the following results:
```

Algorithm	MEE	MSE	Elapsed Time (ms)
Adam	1.565	4.313	1242444
Adamax	1.563	3.846	5385730

And following are the acquire learning curves for both algorithms:



As shown, it goes much more stable than the regular Adam as claimed by the authors.
Plus a potential for a better convergence result despite the slow performance possibly due to the max operator.

10. References

[1] Antonio Frangioni, Federico Poloni, UNIPI Computational Mathematics for Learning and Data Analysis course

[2] Alessio Micheli, UNIPI AA1 Machine learning foundations course

[3] Diederik P. Kingma, Jimmy Lei Ba: ADAM: A METHOD FOR STOCHASTIC OPTIMIZATION
https://arxiv.org/pdf/1412.6980.pdf

[4] Ilya Sutskever,James Martens, George Dahl, Geoffrey Hinton, On the importance of initialization and momentum in deep learning
http://www.cs.toronto.edu/~fritz/absps/momentum.pdf

[5] How can you prove that the loss functions in Deep Neural nets are non-convex?
https://www.quora.com/How-can-you-prove-that-the-loss-functions-in-Deep-Neural-nets-are-non-convex#!n=12

[6] EncogML the C# library we compared our MLP results against
http://www.heatonresearch.com/encog/

[7] scikit learn the python library we compared our linear models against
http://scikit-learn.org/stable

[8] Math.Net our library of choice for providing linear algebra to C#
https://www.mathdotnet.com/

[9] Adam Tutorial
https://machinelearningmastery.com/adam-optimization-algorithm-for-deep-learning/

[10] CS231n Convolutional Neural Networks for Visual Recognition
http://cs231n.github.io/neural-networks-3/

[11] Xavier weight initialization
http://proceedings.mlr.press/v9/glorot10a/glorot10a.pdf

[12] He Weight initialization

https://www.cv-foundation.org/openaccess/content_iccv_2015/papers/He_Delving_Deep_into_ICCV_2015_paper.pdf

[13] The MONK's Problems
A Performance Comparison of Dierent Learning Algorithms
S.B. Thrun, J. Bala, E. Bloedorn, I. Bratko, B. Cestnik, J. Cheng, K. De Jong, S. Dzeroski,

S.E. Fahlman, D. Fisher, R. Hamann, K. Kaufman, S. Keller, I. Kononenko, J. Kreuziger, R.S.Michalski, T. Mitchell, P. Pachowicz, Y. Reich H. Vafaie, W. Van de Welde, W. Wenzel, J. Wnek,and J. Zhang

[14] Pattern recognition and Machine learning, by Christopher Bishop.

[15] Accelerated first-order methods by Geoff Gordon & Ryan Tibshirani

Optimization 10-725 / 36-725
https://www.cs.cmu.edu/~ggordon/10725-F12/slides/09-acceleration.pdf

[16] Marco Danelutto, Massimo Torquati, UNIPI: Distributed systems: paradigms and models
http://didawiki.cli.di.unipi.it/doku.php/magistraleinformaticanetworking/spm/start