Non-linear Function Approximating Neural Network Trained by the Levenberg-Marquardt Algorithm

ECE 174 Fall 2021 Mini Project 2

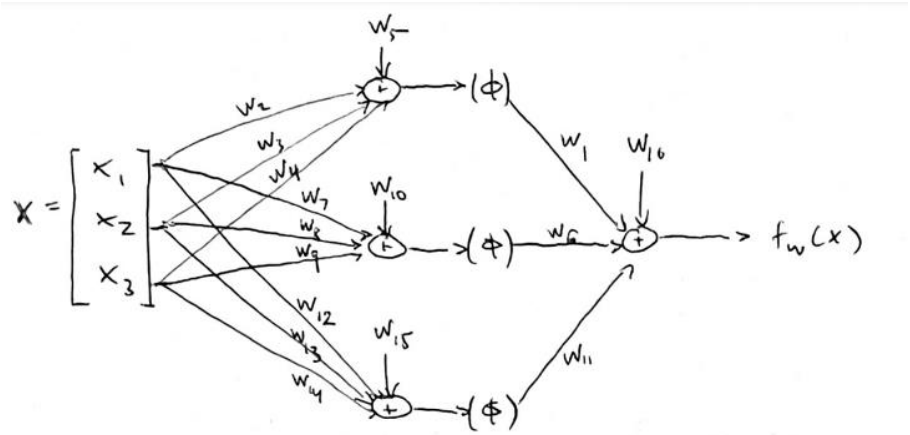
Hao Le

NOTE: source code includes main.py script which is meant to be ran. It will demonstrate all the experiments and produced figures outlined in this report. However, the number of training iterations can be and should be reduced in the script to allow faster run time for demonstration purposes to the grader.

Mathematical background

Neural network design

We have a neural network designed as follows:



It takes an input vector of 3 elements, and by passing these elements through the network with weights and biases (weights 1-16), we essentially perform a non-linear mapping of the 3D vector to a 1D scalar. The objective is to learn such weights and biases so that the neural network acts as a specific non-linear function. We train the neural network by giving examples of how it should behave i.e. we give it random 3D vectors along with their corresponding non-linearly mapped scalars, where the non-linear map is some function we define.

If we have N randomly generated points, x_1 to x_N, along with the scalars that result from non-linearly mapping them, y_1 to y_N, then we can define the residual vector as:

$$r(w) = \begin{bmatrix} f_w(x^{(1)}) - y^{(1)} \\ f_w(x^{(2)}) - y^{(2)} \\ \vdots \\ f_w(x^{(N)}) - y^{(N)} \end{bmatrix} = \begin{bmatrix} r_1(w) \\ r_2(w) \\ \vdots \\ r_N(w) \end{bmatrix}$$

Every row of the residual vector is the error between what the neural network thinks is the non-linearly mapped scalar and what the actual scalar should be, for every randomly generated point. Ideally, we

want all rows to be zero, meaning that our neural network perfectly mimics a non-linear function – this implies that we also want the norm of the residual vector to be zero. However, our neural network is limited to how many weights it has and its architecture. The neural network can never perfectly replicate the behavior of a defined non-linear function, so the residual norm is never zero – we can however heuristically minimize the residual norm, through the Levenberg – Marquardt algorithm.

Levenberg – Marquardt algorithm for minimizing a non-linear loss function

---

Algorithm 18.3 LEVENBERG–MARQUARDT ALGORITHM FOR NONLINEAR LEAST SQUARES

given a differentiable function $f : \mathbf{R}^n \to \mathbf{R}^m$, an initial point $x^{(1)}$, an initial trust parameter $\lambda^{(1)} > 0$.

For $k = 1, 2, \ldots, k^{\max}$

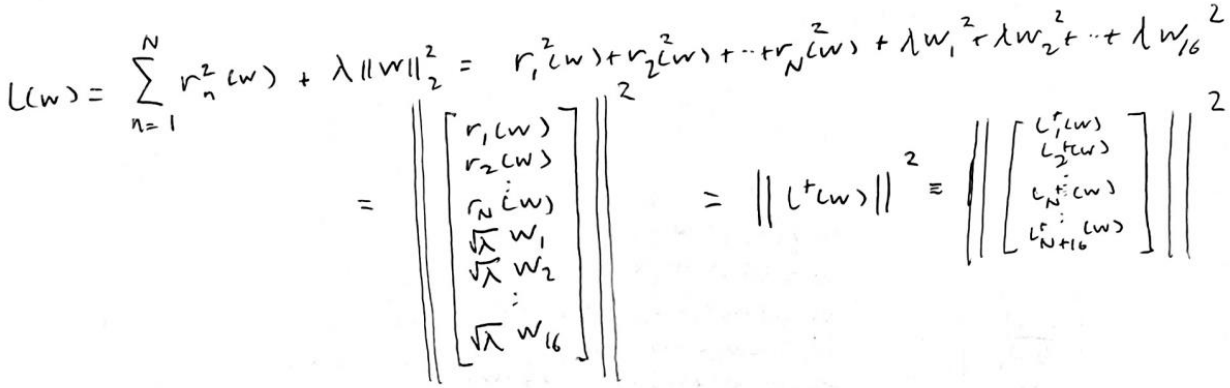
1. *Form affine approximation at current iterate.* Evaluate the Jacobian $Df(x^{(k)})$ and define
$$\hat{f}(x; x^{(k)}) = f(x^{(k)}) + Df(x^{(k)})(x - x^{(k)}).$$
2. *Compute tentative iterate.* Set $x^{(k+1)}$ as minimizer of
$$\|\hat{f}(x; x^{(k)})\|^2 + \lambda^{(k)}\|x - x^{(k)}\|^2.$$
3. *Check tentative iterate.*
If $\|f(x^{(k+1)})\|^2 < \|f(x^{(k)})\|^2$, accept iterate and reduce λ: $\lambda^{(k+1)} = 0.8\lambda^{(k)}$.
Otherwise, increase λ and do not update x: $\lambda^{(k+1)} = 2\lambda^{(k)}$ and $x^{(k+1)} = x^{(k)}$.

Taken from "Introduction to Applied Linear Algebra" by S. Boyd and L. Vandenberghe.

We have a loss function defined as:

$$L(w) = \sum_{n=1}^{N} r_n^2(w) + \lambda \|w\|_2^2$$

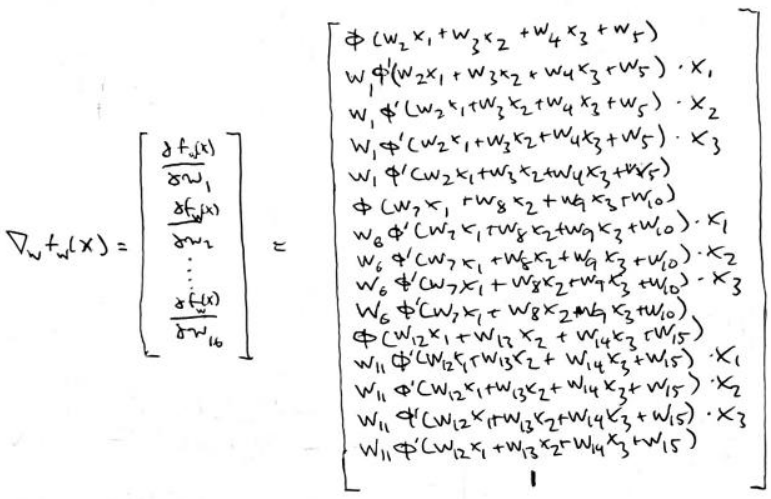
Abstractly, we are minimizing the residual error of the network as well as the weights vector norm, with a regularization coefficient lambda. This is to prevent overfitting, and keep weights from "exploding", taking on extreme values.



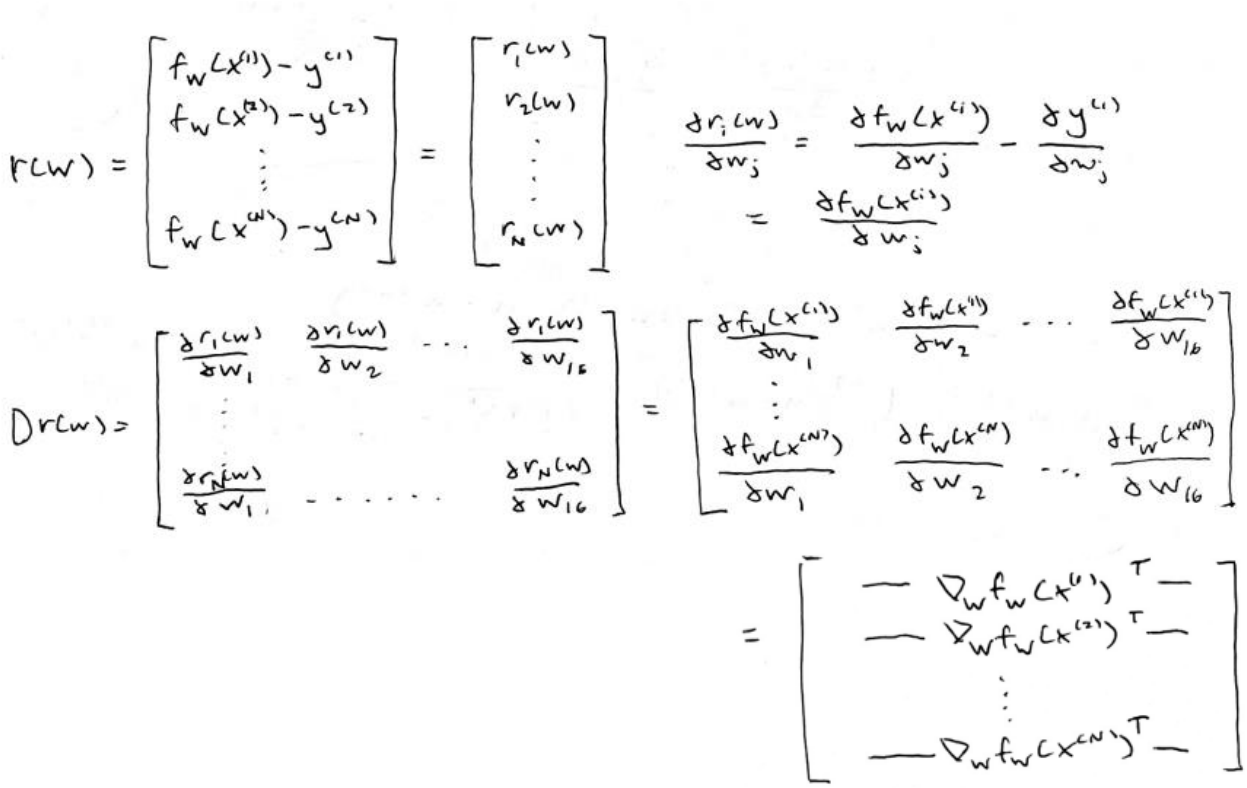
We can rephrase the loss function as the squared norm of a single vector. Thus, we can apply the LM algorithm to minimize the loss function $l^+(w)$'s norm which in turn also minimizes its square norm. We will be referring to $l^+(w)$ as the loss function from now on, even though technically it should be its norm squared.

From a high level, the LM-algorithm works iteratively, where each iteration is performed as follows:

1. Use the previous iterations weights, or if this is the first iteration random weights, w^k to do a first-order Taylor approximation of the loss function. This requires the Jacobian of the loss function.

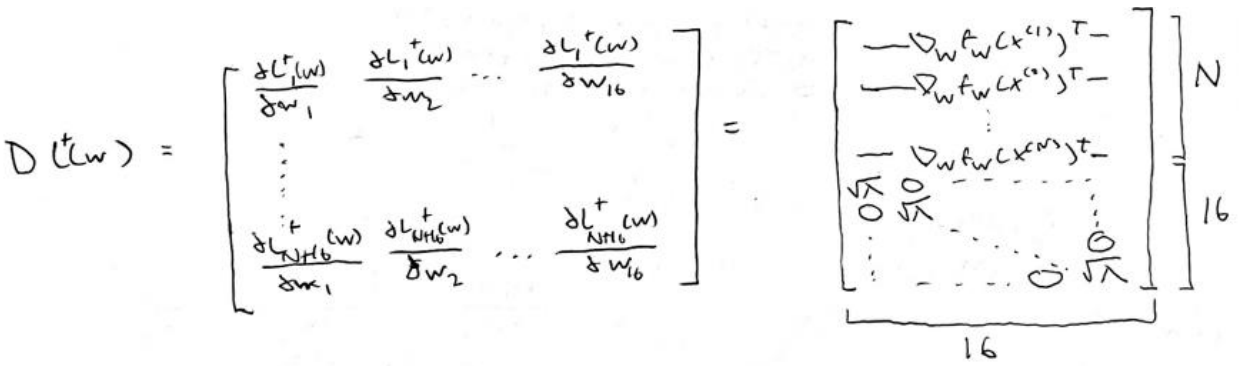


To find the Jacobian, start with finding the gradient vector of the neural network function with respect to the weights as follows. Phi in this case represents the tanh function – its derivate is 1 – tanh^2 which is implemented in the source code. The chain rule is used.



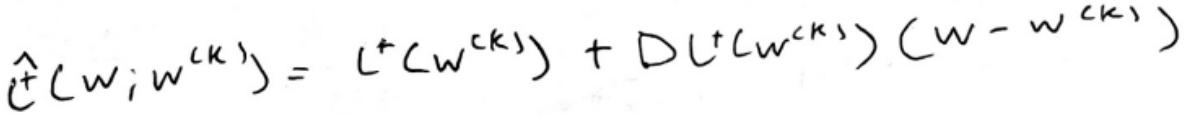
The Loss function includes the prediction residuals, r(w), and the Jacobian of $l^+(w)$ will perform partial derivatives of each $r_i(w)$ to i=N with respect to the weights. This can be simplified by realizing that the partial derivative with respect to a weight of $r_i(w)$ is simply the partial derivate of the neural network function, evaluated at data point X_i, with respect to the weight – this is because the partial derivate of y_i , a constant, is 0.

Thus, the Jacobian of $l^+(w)$ will include Dr(w) on top which is the transposes of the neural network function gradient vectors evaluated for each data point X_i. However, this is not the full $l^+(w)$ Jacobian because there are still the regularization terms left.

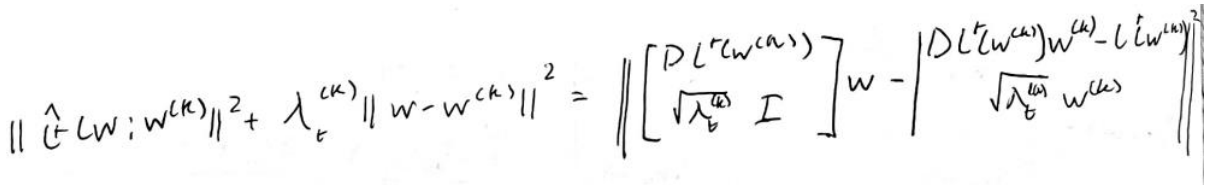


By partial derivates with respect to the weights of the last 16 rows of $l^+(w)$, we yield a 16 x 16 diagonal matrix with the root of the loss regularization term in the diagonals. Thus, the Jacobian of $l^+(w)$, $Dl^+(w)$ becomes a (N + 16) x 16 matrix.

2. Using $Dl^+(w)$, we construct an affine approximation of $l^+(w)$ at w^k , l hat:



3. Do ordinary linear least squares to minimize l hat and find w^{k+1} :



Derivation of matrices of linear least squares problem found on page 391 of "Introduction to Applied Linear Algebra" by S. Boyd and L. Vandenberghe.

By adding the trust lambda regularization term, we ensure that the minimizing weight w^{k+1} does not "jump" far from the previous iteration's w^k . This is l hat degrades in accuracy in terms of approximating $l^+(w)$ as w drifts farther away from w^k. The approximation could say that it is at a

minimum, but when plugging the weights back into the original $l^+(w)$, we may find that the loss norm has actually increased.

4. If we plug in the determined w^{k+1} weights into the original loss function and find that $l^+(w^{k+1})$'s norm is larger than $l^+(w^k)$'s norm i.e. the loss function has actually increased as a result of the affine approximation, we increase the trust lambda (by some coefficient) and re-minimize the approximation, but now with more freedom to choose a farther w^{k+1}. On the contrary, if the loss norm has increased, reduce the trust lambda, then the next iteration will use the approximation minimizing w^{k+1} to construct its own approximation.
   Reducing the trust lambda ensures local minimums are approached slowly and are not missed; increasing the trust lambda discourages local maximums. The amount of reduction/increase is to be determined by observing convergence performance, as specified in the source code.

The LM algorithm is traverses the "landscape" of the loss function, where each position is a 16-dimension weight vector. It gravitates towards areas where loss gets lower, but can be stuck in a local minimum i.e., the loss function converges. To control how susceptible the algorithm is to local convergence, we tune the loss and LM trust regularization lambda terms and experiment with the initial "guess" of the weights as this determines the locality of nearby local minimums.

Algorithm stop condition

The implementation of LM in the source code uses a threshold to determine when to stop. By checking if the current loss, $l^+(w^{k+1})$ norm squared, is smaller than a set threshold, in this case 0.1 by default, the algorithm stops if the loss is small enough, which is the main goal (the ideal goal is zero, but as explained previously getting close to zero is good enough and achievable).

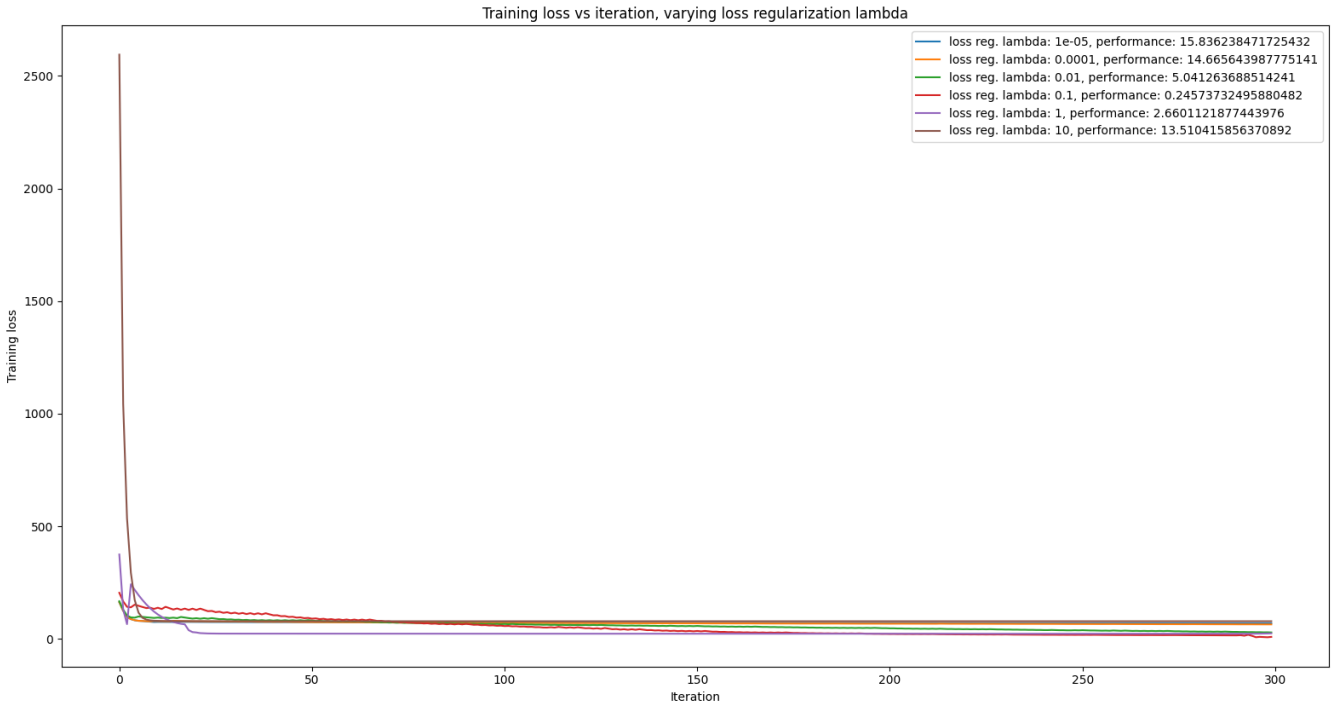
Experiments

The non-linear function used will be as follows: $g(x) = x_1 * x_2 + x_3$

Testing performance is determined as follows: for every test point x_N, find the squared error of $f_w(x_N) - g(x_N)$. Then sum all of the squared errors to find the total squared error for all testing points. The lower the squared error, the better performance.

Varying loss regularization lambda term

In this experiment, we see how varying the lambda regularization term in the loss function affects convergence and performance of the neural network. The initial trust lambda is kept at 1, and the initial weights are generated randomly from a uniform distribution of -10 to 10. The number of training iterations is kept at 300. Each trained neural network is evaluated on 500 random test points.



From the plot of training loss, we see that for all lambdas, it decreases quickly, and converges, indicating that the LM algorithm is working.

As lambda increases from a small to large value, the rate at which training loss decreases is slower. For example, the blue line which represents the smallest tested lambda shows the loss converges very

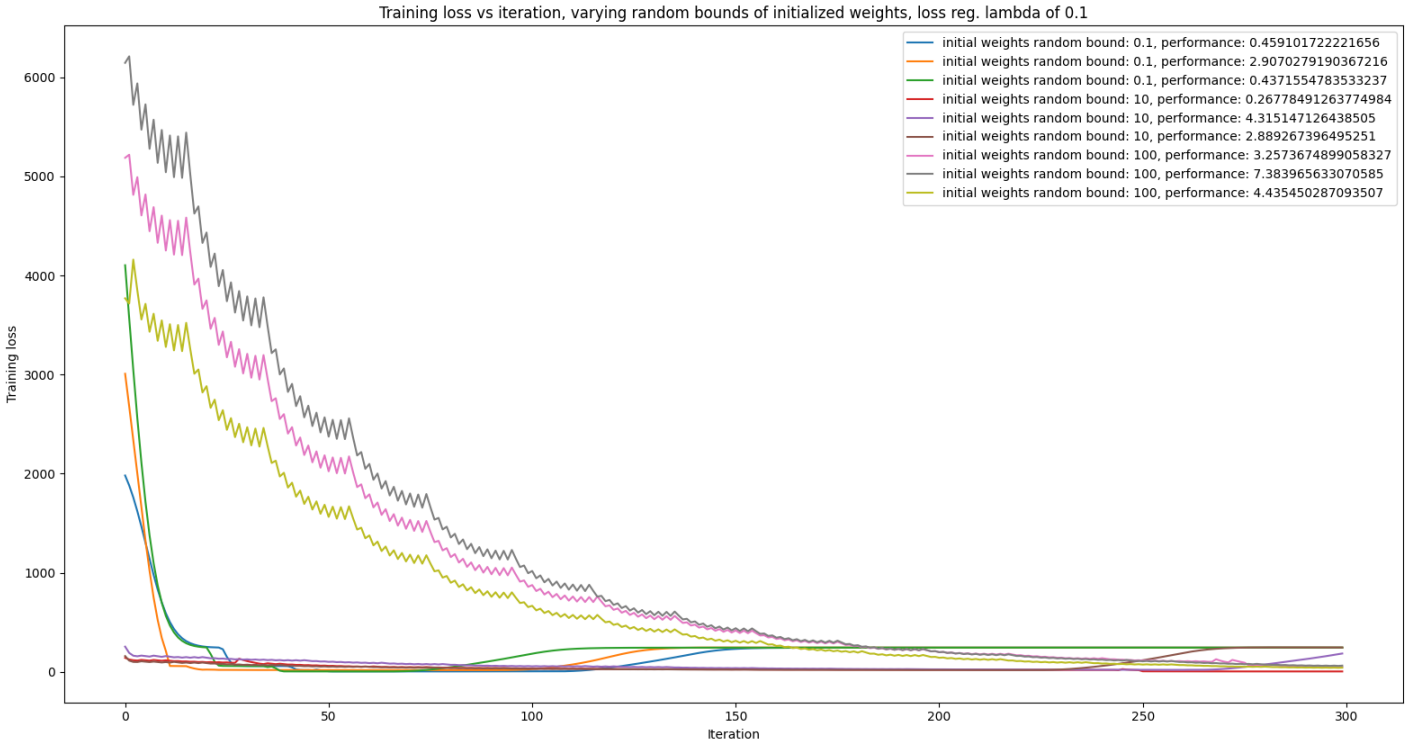
quickly, around 5-10 iterations. However, looking at red and blue lines with higher lambdas, these exhibit gradual decreases in training loss, however the loss at which they converge is lower than that of blue's. In other words, lower lambdas yield quick convergence, and larger lambdas yield slower convergence but at a lower local minimum. It is also worth noting that the brown line with the highest tested lambda also exhibits a gradual loss decrease rate but appears to have converged to the same local minimum as lower lambdas – it has been "sucked" into it.

A lambda of 0.1 yields the lowest end training loss and correspondingly the best test performance (lower is better since we are trying to reduce the residual norm squared vector of every test point)

Varying random distribution spread of initialized weights

In this experiment, we see how varying the spread of the randomly initialized weights affect convergence and performance of the neural network. The idea is by greatly varying the initial "guess" of where a local minimum can be, we expose the algorithm to new localities of minimums, increasing our chances of getting the best overall minimum across multiple runs. The initial trust lambda is kept at 1, and the initial weights are generated randomly from a uniform distribution of varying bounds (for example from -1 to 1 or -100 to 100). The number of training iterations is kept at 300. Each trained neural network is evaluated on 500 random test points.

For each bound, we train the neural network on three different initial weights of the same random spread – this is to determine if the algorithm takes the same path and converges to the same minimum, as indicated by the loss-iteration plot. We hypothesize that the wider the spread, the loss curves of the same spread should be far apart and converge to different values.

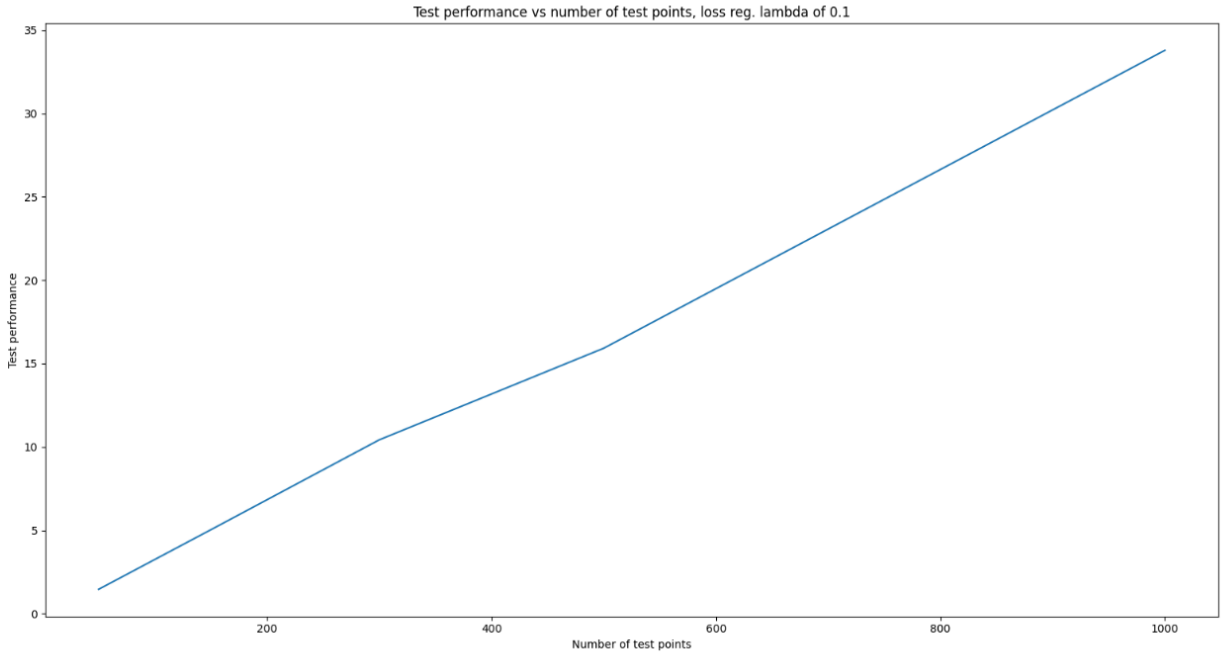


Upon observation, it is clear that the smallest spread of -0.1 to 0.1, represented by blue, orange, and light blue lines, follow very similar loss patterns. Around 100-150 iterations, they all converge to a local minimum of around 400. This pattern also follows for the spread of -10 to 10, though the convergence happens later, at around 300 iterations. For the largest spread of -100 to 100, convergence does not happen for the iterations tested, and loss gradually decreases. All three trials of the same spread also yield different loss curves, implying they are searching different routes for local minimums.

Focusing on performance, we see that the lowest spread yielded the best performance – this is because the loss reached the lower values, but did not converge to them in the long run. We expect to see the same performance, or even better for higher lambdas had the number of iterations been increased.

Varying the number of test points

In this experiment, we see how varying the number of test points affects the testing performance metric. The initial trust lambda is kept at 1, and the initial weights are generated randomly from a uniform distribution limited to -10 and 10. The number of training iterations is kept at 500.



We observe a linear increase in the testing performance (getting worse) as the number of testing points increase. This implies that the squared error of the residual norms increases – this makes sense because the neural network is not a perfect replication of the non-linear function. Visually, by adding more points, the neural network has a harder time to “fit” to the points – the residuals gradually accumulated, and the error increases proportionally to itself squared.

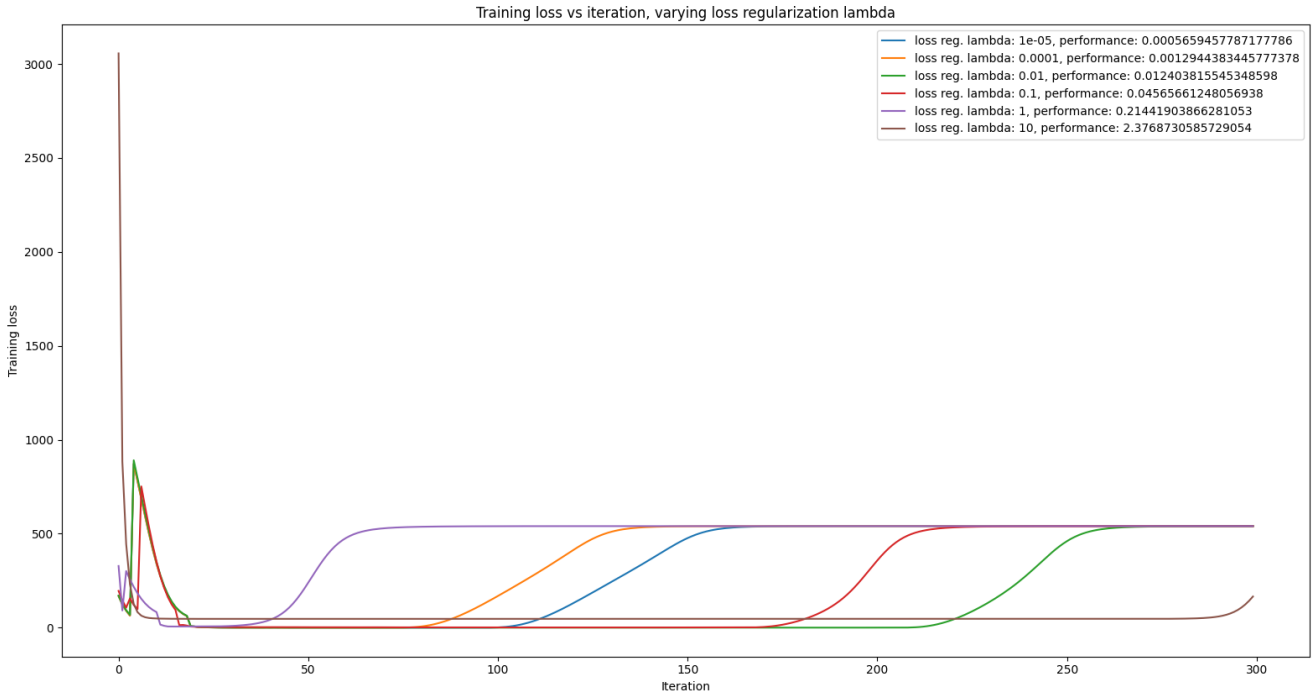
Repeating experiments with different non-linear function

We will repeat the previous experiments but with a new function, this time the linear function

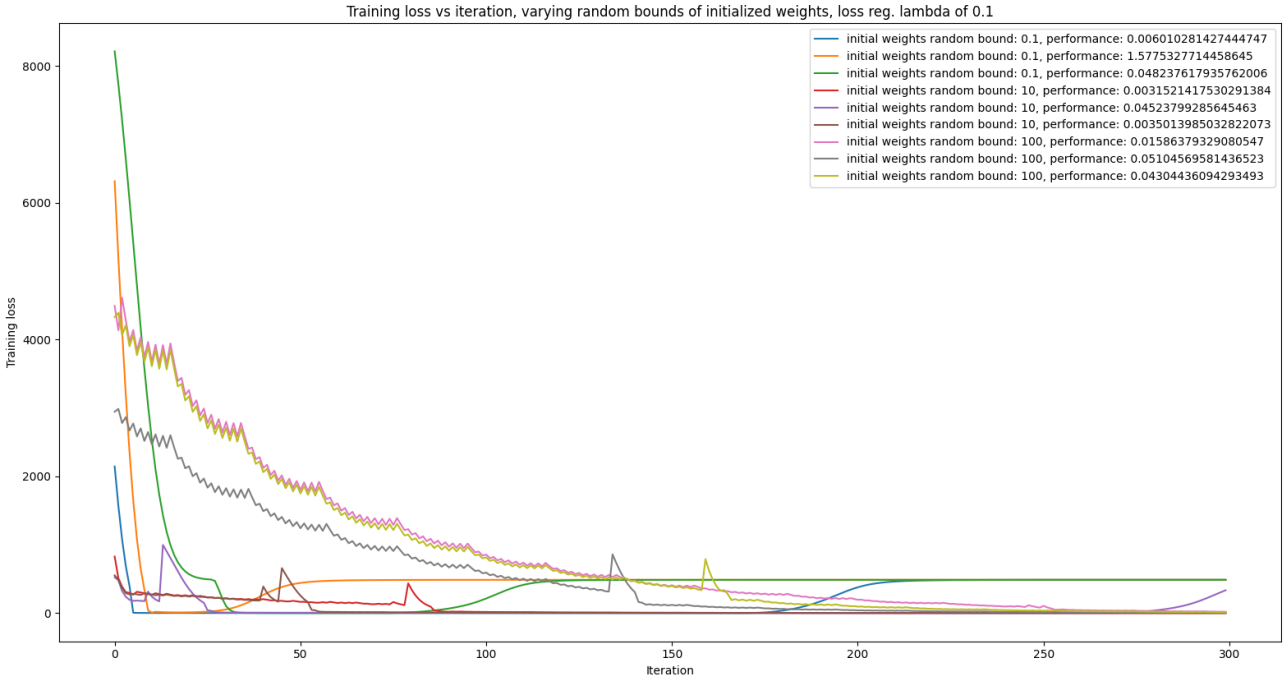
$$g(x) = x_1 + x_2 + x_3 .$$

In theory, a neural network should be able to perfectly replicate this function. However, since our network design includes nonlinearities of bias and tanh, the LM-algorithm will nonetheless try its best to optimize the weights for a minimized loss.

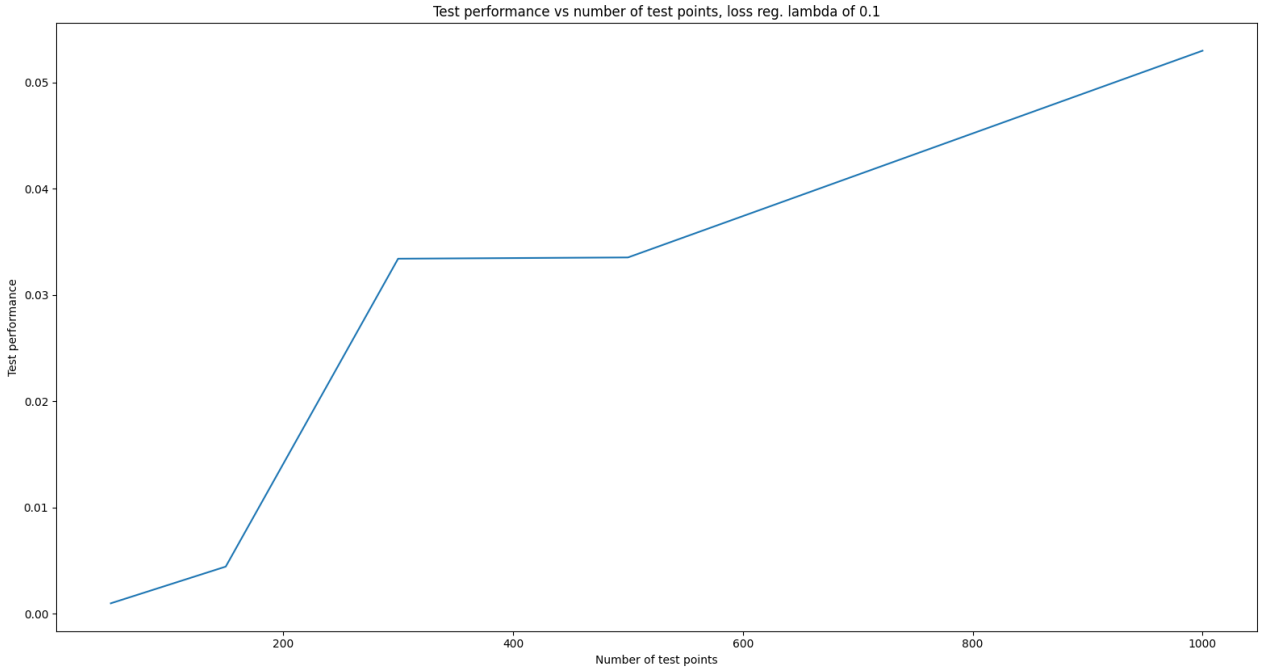
The source code is structured in a way that allows easy swapping of the non-linear function – this is done by **injecting the dependency of the non-linear function into the neural network functions.**



It is immediately obvious that the neural network has an easier time approximating the linear function despite its nonlinearities within the architecture. We see that virtually all lambda values, with the exception of 10, converge to a near zero loss – correspondingly, we see excellent test performance that is also near zero. Interestingly, after converging to near zero, all lambda values “break out” of the local minima and jump to a higher local minimum as iterations increase.



For varying the spread of the initial weights, we also observe similar patterns from the previous experiments. All distributions eventually converge to near zero loss and obtain near zero performance metrics. The larger spread weights tend to converge gradually, and do not experience the “jump” out of the near-zero loss local minimum. It is also worth noting that the smaller distributions experience intermittent spikes in loss – these spikes get higher in frequency as distribution widens.



As expected, we see a linear increase in the performance metric for more test points, but the rate of increase drastically decreases compared to the previous experiment. This is because of how well the neural network can fit to the linear function. In other words, though the addition of more test points incurs more residuals that are squared, the magnitudes of the residuals are very small.

Conclusion

We have successfully coded from scratch an implementation of the Levenberg – Marquardt algorithm that is used to train the weights of a neural network. The neural network then uses the trained weights to behave like a nonlinear function by taking a 3D input vector and nonlinearly mapping it to a scalar output – the error between the scalar output and the desired output, that from a defined nonlinear function – is minimized.