

Notes on Simple Nets

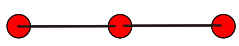

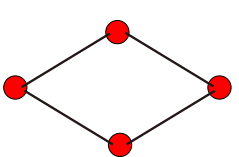
1 Overall plan

(v.1: Masud, August 2019)

The current idea of the project:

Fitness landscape of 3-node and 4-node neural networks

Restricting to one input node and one output node, we can think of three network geometries:

| | |
|--|--|
| <p>A</p>  <p>B</p>  <p>C</p>  | <p>(A) One hidden layer, containing a single hidden node. 4 parameters.</p> <p>(B) Two hidden layers, each containing a single node. 6 parameters.</p> <p>(C) One hidden layer, consisting of two nodes. 7 parameters.</p> |
|--|--|

We can then explore and describe the landscape of the loss function for the same (or similar) learning task in these three cases. In addition, we can explore and describe how the gradient descent moves us along this landscape.

The three networks have respectively 4, 6 and 7 parameters.

The primary learning task with these networks will be to learn a scalar function. The function $f(x) = x^2$ seems the most natural to concentrate on.

We still have to specify the domain of the function (interval from which x is chosen) over which to concentrate on. We might want to use the sigmoid function in the definition of the network; in that case the outputs are limited to $[0, 1]$. Hence it makes sense to use the interval $[-1, 1]$.

The 3-node network (A) is so limited that it is not able to approximate the parabolic shape in the whole range $[-1, 1]$ (please double-check this statement), so we will probably have to limit it to $[0, 1]$.

The choice of activation function defines the network, i.e., the loss function will depend on the activation function. Let's start with the sigmoid whenever we can, and possibly play with using other activation functions later.

1.1 The functions expressed by the toy networks

For the 3-node network A, the output as a function of the input x is

$$[A] \quad N(x) = \sigma\left(w_{11}^2 \sigma(w_{11}^1 x + b_1^1) + b_1^2\right) \quad (1)$$

where the superscript/subscript notation is that of Nielsen, see, e.g., the diagram in

<http://neuralnetworksanddeeplearning.com/chap2.html>

The same notation is probably standard elsewhere as well. The superscripts are layer indices and the in the subscript ij for the w 's, the first is an index for the nodes of the previous layer and the second is the index for the nodes of the current layer. (The input layer is called layer 0 here.) This elaborate notation is probably overkill for the toy networks we will be using; we will use simpler names later on when we focus on the individual networks.

The problem of learning the x^2 function with the network A is simply the problem of least-square-fitting the above nonlinear function to a dataset of the form $\{(u, u^2)\}$.

Similarly, the functions expressed by the other two networks are

$$[B] \quad N(x) = \sigma\left(w_{11}^3 \sigma\left(w_{11}^2 \sigma(w_{11}^1 x + b_1^1) + b_1^2\right) + b_1^3\right) \quad (2)$$

and

$$[C] \quad N(x) = \sigma\left(w_{11}^2 \sigma(w_{11}^1 x + b_1^1) + w_{21}^2 \sigma(w_{12}^1 x + b_2^1) + b_1^2\right) \quad (3)$$

We should probably use simpler notations while discussing the individual networks.

1.2 Choice of activation function

Should we choose activation functions that are well-suited for learning the x^2 function, or choose those that are bad for the present purpose? The suitability of the activation function probably affects the nature of the fitness landscape, so both might be interesting. However, let's start with choosing activation functions that allow us to do a relatively good job.

Given the nature of the problem — fitting a nonlinear (quadratic) function — a piecewise linear function will probably not do a good job. (Maybe we should check this statement.) This rules out the ReLU function.

So let's concentrate on the sigmoid function

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

and the softplus function

$$\sigma(x) = \log(1 + e^x)$$

The softplus is a rounded version of the ReLU.

(v.1: John, August 2019) For the sigmoid activation function, we can already say something about the fitness landscape where some of the network parameters are large. If we denote the affine transformation from layer $i - 1$ to layer i by ω_i :

$$\omega_i(a_{i-1}) = w_i a_{i-1} + b_i,$$

then, for a given network input x , the fitness of the network, as a function of ω_i can be written as

$$L(\omega_i) = L \circ \sigma_n \circ \omega_n \circ \cdots \circ \omega_{i+1} \circ \sigma_i \circ \text{Eval}_{a_{i-1}}(\omega_i),$$

where a_{i-1} is the activation vector of layer $i - 1$ after x has been fed into the network and $\text{Eval}_{a_{i-1}}$ is the evaluation map: $\text{Eval}_{a_{i-1}}(\omega_i) = \omega_i(a_{i-1})$. The exterior derivative of this is:

$$dL(\omega_i) = dL(d\sigma_n)(d\omega_n) \cdots (d\omega_{i+1})(d\sigma_i)d\text{Eval}_{a_{i-1}}(\omega_i),$$

where the Jacobi matrix $d\omega_i = W_i$ is the i -th weight matrix and $d\sigma_i$ is a diagonal matrix whose diagonal is obtained by evaluating $\frac{d\sigma}{dx} = \sigma(x)(1 - \sigma(x))$ on each component of the vector $z_i = \omega_i(a_{i-1})$. Heuristically, if any of the network parameters defining the map ω_i are large then some of the components of the output $\omega_i(a_{i-1})$ may also be large. If that is the case then the derivative of the sigmoid function evaluated on that component will be close to zero and the corresponding component of $dL(\omega_i)$ will be small,

suggesting the fitness landscape is almost flat in directions where the large parameters change. Looking at a component of the vector $\omega_i(a_{i-1})$:

$$(\omega_i(a_{i-1}))_k = b_j + \sum_j W_{kj}(a_{i-1})_j$$

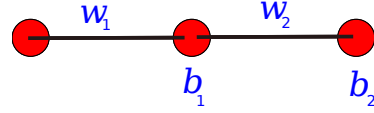
This component will be large, for the given input, if the network parameters are large and don't cancel each other. Hence, we can at least say the fitness landscape is almost flat, where any one the network parameters defining ω_i is large, in the direction that increases/decreases that parameter while keeping the other parameters fixed.

2 The 3-node network

(v.1: Masud, August 2019)

2.1 Setup

Let's simplify the notation; one index should suffice. The four parameters are w_1 and b_1 (determining the hidden-neuron value from the input value x) and w_2 and b_2 (which determine the final output from the hidden-neuron value).



$$\begin{aligned} z_1 &= w_1 x + b_1 \\ z_2 &= w_2 \sigma(z_1) + b_2 \\ \text{output} &= \sigma(z_2) \end{aligned}$$

The function represented by the network is

$$N(x) = \sigma(w_2 \sigma(w_1 x + b_1) + b_2) \quad (4)$$

So the quantity we want to analyze is

$$L(w_1, w_2, b_1, b_2) = \int_{x_{\min}}^{x_{\max}} [N(x) - x^2]^2 \quad (5)$$

In practice we want to discretize this integral and have a sum over a grid of x values. Hopefully the answer does not depend much on the discretization.

Define

$$x_i = x_{\min} + \frac{x_{\max} - x_{\min}}{n_T} i \quad \text{with } i = 0, 1, \dots, n_T$$

The subscript T could stand for ‘test’, as the set of numbers $\{(x_i, x_i^2)\}$ can be regarded as the test set used to define the fitness function or loss function. However, for the present toy problem, there is probably no harm to use the same set as the training set.

The loss/fitness function we aim to analyze is thus

$$L(w_1, w_2, b_1, b_2) = \frac{x_{\max} - x_{\min}}{n_T} \sum_{i=0}^{n_T} [N(x_i) - x_i^2]^2 \quad (6)$$

We expect that the results should not depend on n_T as long as it is ‘large enough’; perhaps even $n_T = 20$ is large enough.

If some other target function $f(x)$ is being trained for instead of $f(x) = x^2$, one simply replaces x_i^2 by $f(x_i)$ in the above expression.

For definiteness, as we explore our different networks with different activation functions, maybe it makes sense to concentrate on the setup described here: using the $f(x) = x^2$ function in the range $(x_{\min}, x_{\max}) = (-1, 1)$, with $n_T = 100$.

2.2 Experimenting with Mathematica: Some observations

Played with minimizing the loss function using the minimizers provided by Mathematica, using the `FindMinimum[]` function. This function can take the `Method` option: the method can be chosen as one of `Newton`, `PrincipalAxis`, `InteriorPoint`, `QuasiNewton`, `ConjugateGradient`. Have not dug into the details of any of these algorithms; presumably it's not very important. Maybe later we can simply re-explore this system with the Stochastic Gradient Descent, but I assume we will get the same results.

Let's provide results below for $n_T = 100$.

The performance of this primitive network as a learner/predictor is rather limited — it seems not able to fit even roughly the x^2 function in the domain $[-1, 1]$.

2.2.1 Fitting to the domain $x \in [-1, 1]$, sigmoid

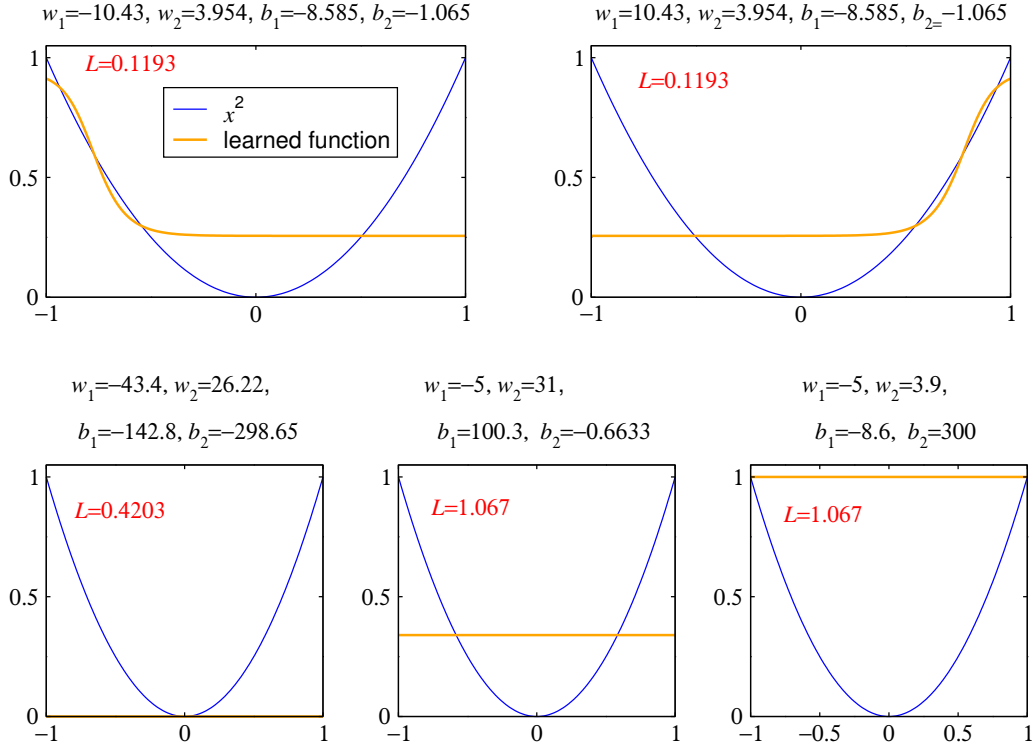
Results for the case of a sigmoid activation function, learning the $f(x) = x^2$ function in the range $(x_{\min}, x_{\max}) = (-1, 1)$, with $n_T = 100$.

Before presenting results, we comment on the expected dependence on the b_2 parameter. For finite values of the other three parameters, if b_2 is very large and positive, the output is 1 for any input, and if b_2 is very large and negative, the output will be 0 for any input value of x . This also means that the cost function L changes from one plateau (≈ 0.4203) to another (≈ 1.067) as b_2 is changed from large negative to large positive.

In fact, if $w_2 = 0$ then $N(x) = \sigma(b_2)$ and $L = \int dx [\sigma(b_2) - f(x)]^2$, and there is no dependence on the first-layer parameters w_1 and b_1 .

Now let's look at some data.

In the figure, some parameter values returned by the Mathematica minimizers are shown, together with the corresponding learned function.



The points shown on the top really are minima: one can see this by varying one of the parameters and holding the others fixed at this point. (Shown below.) These two minima are mirrors of each other, with w_1 value flipped.

The parameter values on the bottom row are not really minima — the loss function is constant in those parameter regions. These are regions where one of the $b_{1,2}$ is very large. The cost function tends to become constant in such regions. Presumably, this is a feature of the sigmoid function.

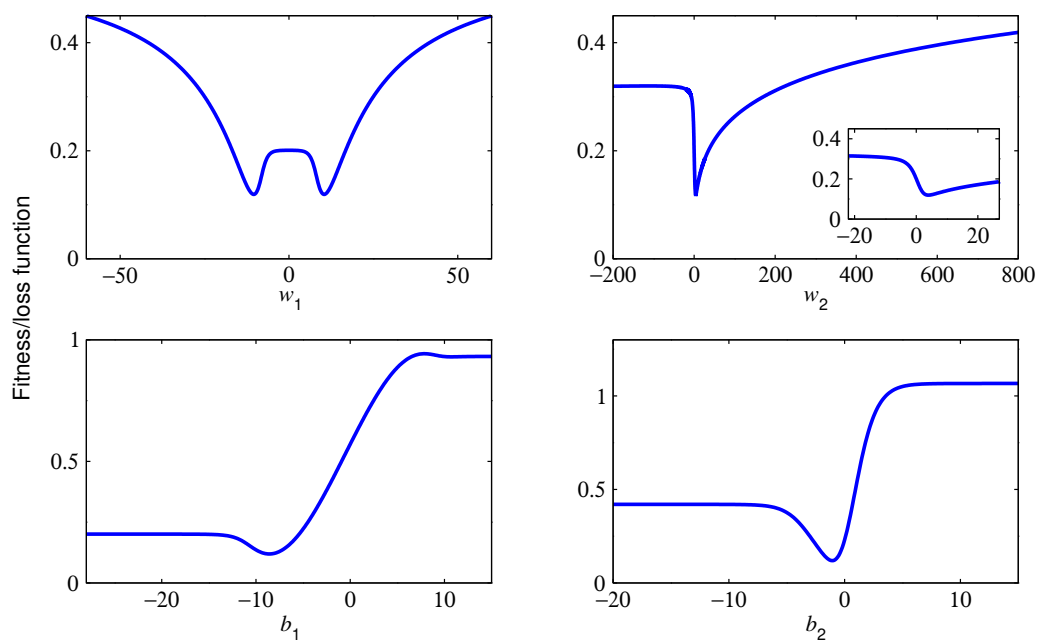
It would be good to understand this better, e.g., whether the loss function becomes constant as any one of the parameters become very large.

The experimentation strongly suggests that there is only the pair or minima described above. It would be good to prove or disprove this rigorously!!

Are there interesting structures other than the minima, e.g., saddle points?

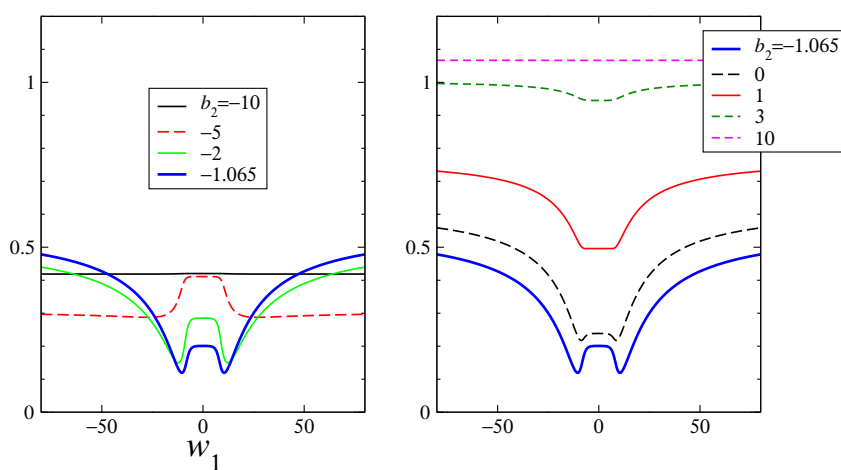
Visualizing the structure near the pair of minima. Below we show the variation of the cost function in the four directions. In each plot, one parameter is varied while the other three are held fixed at the minimal value.

Varying each parameter around point:
 $w_1=-10.43$, $w_2=3.954$, $b_1=-8.585$, $b_2=-1.065$



What happens away from the minima? Does the fitness function become flat?

Keeping w_2 and b_1 fixed at the values found at the minima, we plot L as a function of w_1 for various values of b_2 .

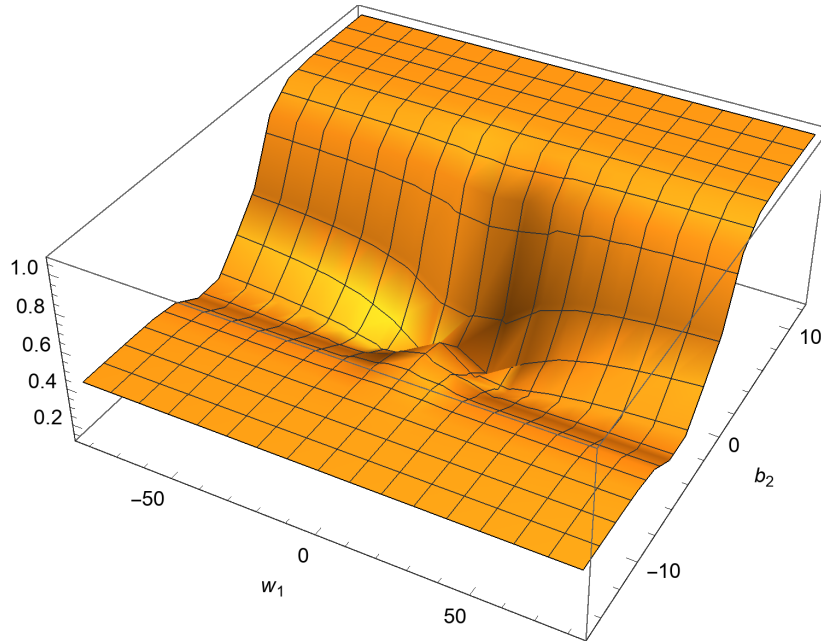


Held fixed:

$$w_2 = 3.954$$

$$b_1 = -8.585$$

As one goes away in either the negative b_2 or the positive b_2 direction, the w_1 -dependence becomes flat. This can also be visualized through a 3D plot:



For large negative b_2 , the loss function settles to a value around 0.42; for large positive b_2 , a value around 1.07. These correspond to predicted functions that are constant at 0 and 1, respectively. They are certainly not minimal!

TODO: add a few plots of the loss function as a function of parameters around other parameter points, e.g., with one of the parameters large.

2.2.2 Fitting to the domain $x \in [0, 1]$

Question: should we do this? Or should we restrict to the same problem and domain $x \in [-1, 1]$ for each of the three networks.

Below some old comments from initial pre-exploration; need to be double-checked.

Using the sigmoid function, the 3-node network even has difficulty fitting in the range $(0,1)$. The Softplus function does a better job.

The landscape seems interesting enough — there are multiple minima. (DOUBLE-CHECK THIS STATEMENT. If true, it is an interesting contrast with fitting in the domain $x \in [-1, 1]$. In the restricted domain the model is not as high-bias as in the larger domain $x \in [-1, 1]$. Maybe the number of minima is related to how unsuitable the model is?)

It also has parameter regimes in which the loss function stays essentially constant, and sometimes very sharp increases near a minimum.

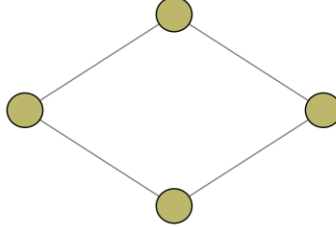


Figure 1

3 The diamond network

(v.1: John, August 2019)

3.1 Introduction

We trained a simple neural net to learn the function $f(x) = x^2$ and explore aspects of the landscape of the related loss function. The network we trained, depicted in Fig.1, has a single input neuron, a single hidden layer with two neurons and a single output neuron. The network we trained uses the sigmoid function as its activation function and so the output of the network is a number in the interval $[0, 1]$. For this reason, we focus on learning the function $f(x) = x^2$ over the domain $[-1, 1]$ so that the range of f matches the range of our network.

To train the network, we generated an array of random 10000 points uniformly distributed in the interval $[0, 1]$ and then computed the value of $f(x)$ for each random point. This gave us a data set to train our network on. A second data set was generated, in the same way, to be used for testing how well the network generalises. The network was trained using the stochastic gradient descent method for 50 epochs with a mini-batch size of 50. The evolution of the network's output as a function of x is shown in Fig.2

Once the network had been trained, we then looked at the performance of the network. The loss function used to evaluate the performance of the net was the squared Euclidean distance between the output of the net and the target output averaged over the test data.

The change in loss due to a variation in each of the network's parameters was calculated. We number the parameters of the network 0 to 6. The weights connecting the first two layers are numbers 0 and 1 while the weights

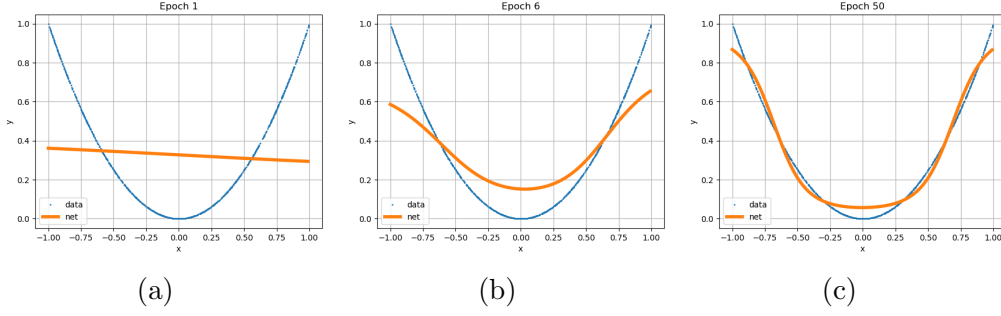


Figure 2

connecting the second and third layers are numbered 2 and 3. The biases of the hidden layers are numbered 4 and 5 while the bias for the output neuron is given the number 6. Given the trained weights and biases $\{\omega_i\}$, we vary a particular parameter by adding a constant $\delta\omega_i$ ranging from -10 to 10 while holding all other parameters fixed. The results are shown in Fig.3. We see each curve having a local minimum at $\delta\omega_i = 0$ indicating the network found a local minimum of the loss function while being trained.

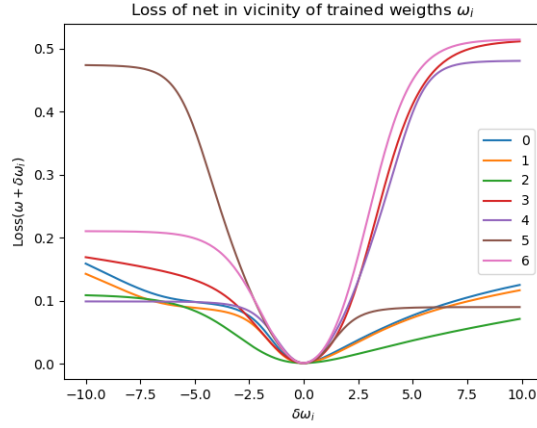


Figure 3