Technical Note

On implementing backpropagation with moment neural network in MATLAB

Yang Qi

**Table of content**

1. General set-up
2. Derivation of backpropagation
3. Efficient MATLAB implementation for the activation function
4. **General set-up**

Schematics

Some notations:

is the mean firing rate

is the variance of firing rate

is the correlation coefficient (for neurons within the same layer)

Layer index = n

Neuron index (in layer n-1) = j

Neuron index (in layer n) = i

Neuron index (in layer n+1) = k

I use redundant notation for the sake of clarity.

Also, I use prime ( and ) for neurons in the same layer to improve clarity.

A complicated activation function:

Loss function where y is the target value, and assume it does not depend on rho.

(More about loss function later)

Forward equations

For a first approximation, we force for all layers, then

For general correlation, use matrix notation

Then

(not sure about here… maybe better use Einstein’s notation)

Loss/error/cost function

Or maybe I should use std instead of var to keep the balance

Note the second term,

1. **Derivation of backpropagation**

General derivation (Eq.1)

First, if n is the output layer, and can be directly calculated (Eq.2)

Next, if n is a hidden layer:

Note terms with blue highlight is zero so above simplifies into:

Define:

Then we have,

Similarly,

Define,

Then we have,

[NB. Error corrected on 2020.7.8]

From Eq.1,

Match coefficients with that in Eq.1 we get a 2D recursive formula,

In matrix notation,

The square matrix here is the Jacobian matrix of the activation function . If n+1 is the output layer, then the colored vectors can be calculated direct from the loss function (Eq2).

1. **Efficient MATLAB implementation for the activation function**

Activation function is the mapping:

In the following we give its definition and numerical implementation.

[2020.7.2: highlighted term was wrong in the code, now corrected]

For clarity, denote:

Then the activation function is given by as:

[Note Tref was missing in Feng’s paper. It makes quite a big difference for small T! Reason Tref should be included: Suppose input mean and var is fixed => var T is fixed. Now imagine Tref increases to infinity. The variance of firing rate should vanish if Tref is included and it should be constant otherwise. Clearly the latter is false as virtually no spike should be observed.]

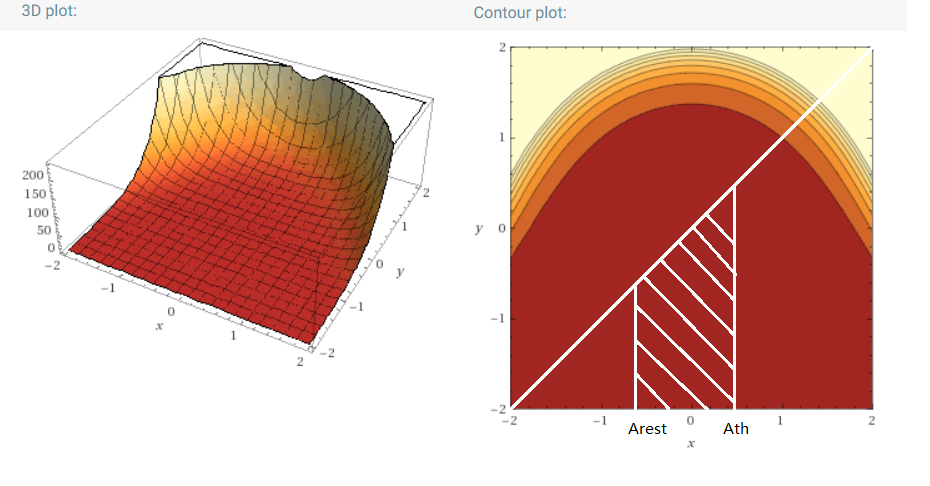
**Numeric implementation for g(x) and its integral**

This function is monotonic and it converges to zero for and diverges for . Numerically this integral presents a type of instability. To fix this, observe

where erfcx is the “scaled complementary error function”, which has a built-in MATLAB function specifically design to avoid numerical overflow. The integration is simply handled by MATLAB built-in function integral to achieve specified error tolerance.

**Numeric implementation for h(x)**

The integrant function over the (x,u) plane is shown below, with the shaded area marking the domain of integration.

****

I implement the integrant numerically using the form:

**exp[(x+u)\*(x-u)]\*[erfcx(-u)]^2**

which provides the best numeric stability. The integration is handled by MATLAB built-in function integral2.

Note that in all of the above, the integrals are guaranteed to be well behaved when , that is, when . However, when the integrals may blow up, especially when . This is just how the function behaves not due to numerical implementation. Luckily this shouldn’t be an issue for biologically plausible values of .

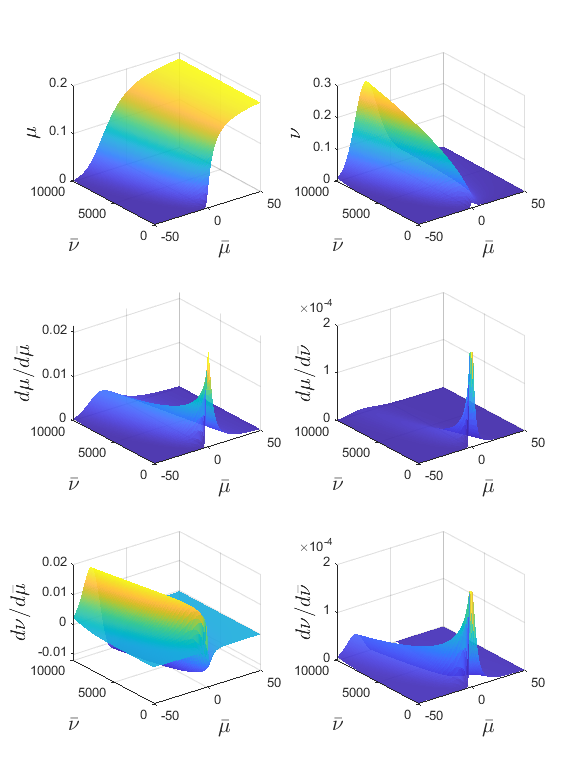
**Jacobian matrix of the activation function**

Note since this part is all element-wise operations, we drop the layer and neuron indices for clarity.

where

where the derivatives of A is given earlier. In MATLAB I implement J as a N x 2 x 2 array.

**Visualization of the activation function and its partial derivatives.**



NB. Output kHz; output kHz^2. The dynamic range of the activation function is limited therefore target output requires tuning for regression problem.

Ideally keep target as well as the output of each layer to be kHz and kHz^2.

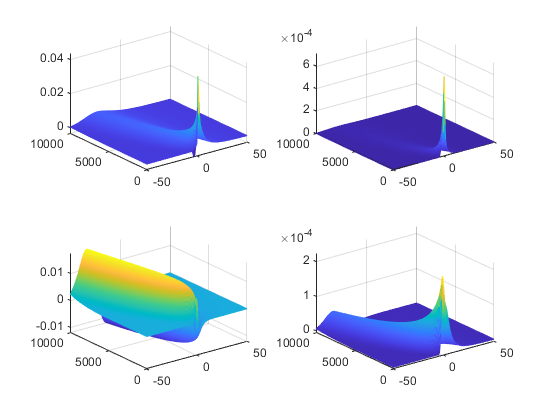
Similarly, the input and the weights should be adjusted so the

Note NaN values when or when is small while

Small variance cases would result in numeric instability and it should be replaced by either interpolation or theoretical sigmoid (how).

[plotted with maf\_validation.m]

Compare above semi-analytical partial derivatives with numerical ones:



[2020.7.2] Derivatives with **with respect to was off by an order of magnitude (due to mistake in dA/dv). Now fixed.**

Note on the case when :

Underlying assumption of diffusion approximation is broken down, and output is simply periodic spikes.

Solve the equation,

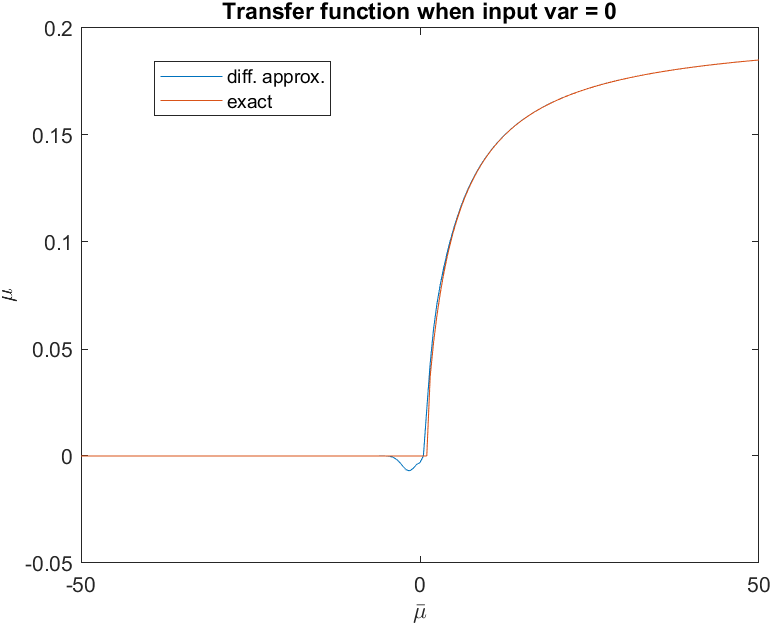
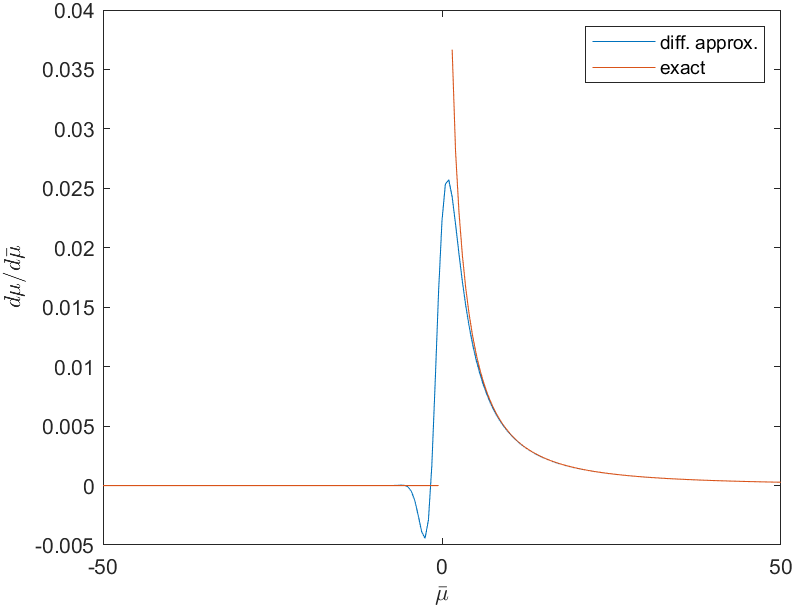
We have:

[for , and 0 otherwise]

Not entirely sure about the last equation here…

Use these values whenever any of it contains NaN. But will this make backprop any better? Note at , the slope of is infinite! Could make things worse…

Validation of above formula:

Finally putting the pieces of the puzzle together.

Update rule

where is the learning rate and is the moment.

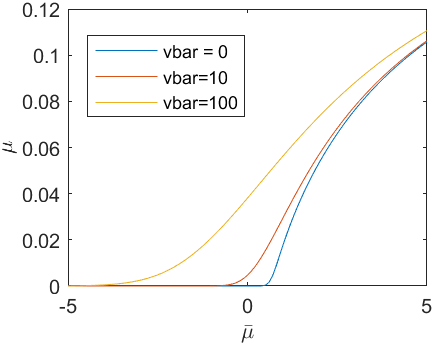
How to adjust hyperparameters

Two conditions must be met when choosing hyperparameters:

This condition ensures that the integrals in the activation function converges (in the analytical sense). When , the activation function actually blows up if it also happens that .

This condition is quite easy to satisfy as long as the correlation matrix has predominantly positive entries.

The activation function flattens out outside this range, which may result in flat gradients thus vanishing changes in the weights.



One way to satisfy this condition is by having a balanced network such that the number of excitatory neurons and the number of inhibitory neurons are roughly the same.

If the layer has only excitatory neurons (such as the input layer), then the following tweak can be used. Assume and each entry in the weight matrix is independent and identically distributed r.v. according to some distribution, and also assume that the number of neurons is large. Then we require,

where N is the number of presynaptic neurons. To proceed, suppose that the presynaptic neural activation has a well-behaved mean value of

About transduction factor for input images

The grayscale values (range between 0 and 1) of the pixels in the raw images of MNIST has a mean value of approximately 0.25. A transduction factor of **0.02** is sufficient to bring it to the desired .

Similarly for output target, scale it to .

**Loss function: MSE**

This is bad for classification problem with one-hot target.

**Softmax cross-entropy loss**

Where is the target label.

Result works similarly for .

Alternatively,

By chain rule, just multiply by to get the derivative with respect to .

**Custom cross-entropy loss?**

Since mu and var are already probabilistic, I should replace softmax with a custom probability.

Where is, for instance, a Gamma distribution. [?]

Assuming output neurons fire independently, then f is separable.

Which has mean and variance . I actually don’t need x here…

(NB the original softmax function is a probability across output neurons, not over the values of output variables; If I use this Gamma distribution then I have imposed a stronger condition)

POINT: x here is the spike count! I want to match my spike count to the target!

Use Poisson for now?

Interpreting the cross-entropy loss as minimizing the KL divergence between 2 distributions

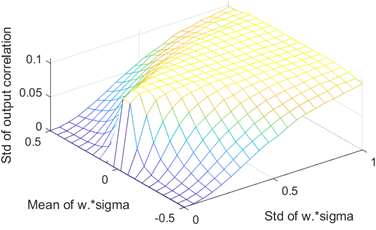
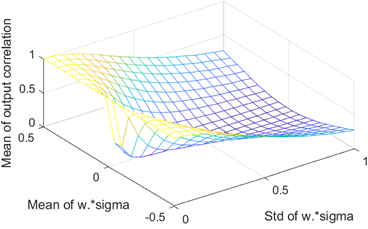
Referene:

<https://gombru.github.io/2018/05/23/cross_entropy_loss/>

<https://rohanvarma.me/Loss-Functions/>

Regarding correlation map

Let the correlation matrix of the first layer be a random positive definite matrix, and let of the first layer be normally distributed. We calculate correlation matrix of the next layer and observe how it changes with the mean and std of .



（sorry the x and y-axes in above figures should be swapped)

A number of observations:

1. The mean of the output correlation is always positive.
2. The std of the output correlation has an upper bound of approx. 0.12, which is slightly bigger than the std of the input correlation of 0.1. Implication: as long as the has a high variance, the std of output correlation should not vanish. [Vanish means all entries in rho become the same value]
3. Unfortunately, the output correlation matrix is not positive definite for nearly all explored cases (which means that the variance calculated using these correlation matrices may become negative)

Regarding scaling of the activation function (nondimensionalization)

[Discarded: below results are incorrect since Tref is not included in the calculation of ]

We are interested in the upper and the lower bound on the activation function (in the limiting cases)

First it is trivial to show that both and (the output) are greater than zero.

Suppose , and , we get,

Note the asymptotic expansion as :

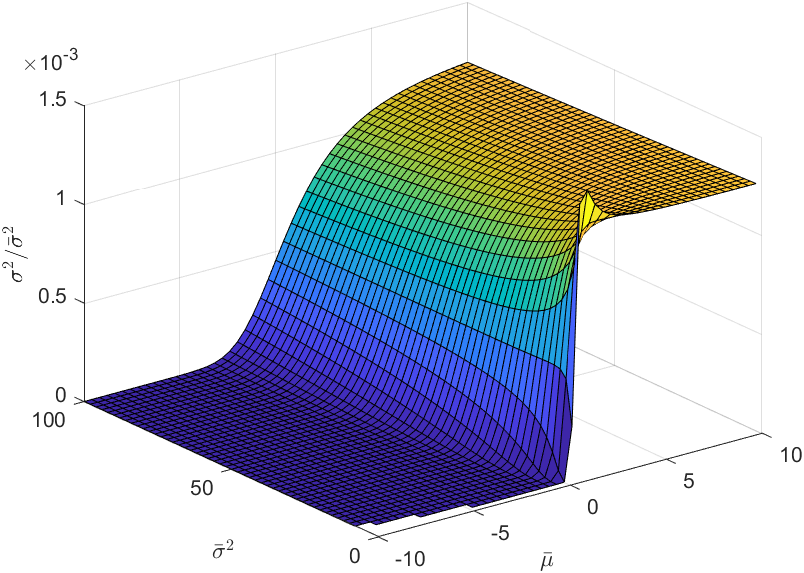
Therefore,

For the variance the situation is a bit more complicated. From numeric evaluation it can be shown that the following limit exists,

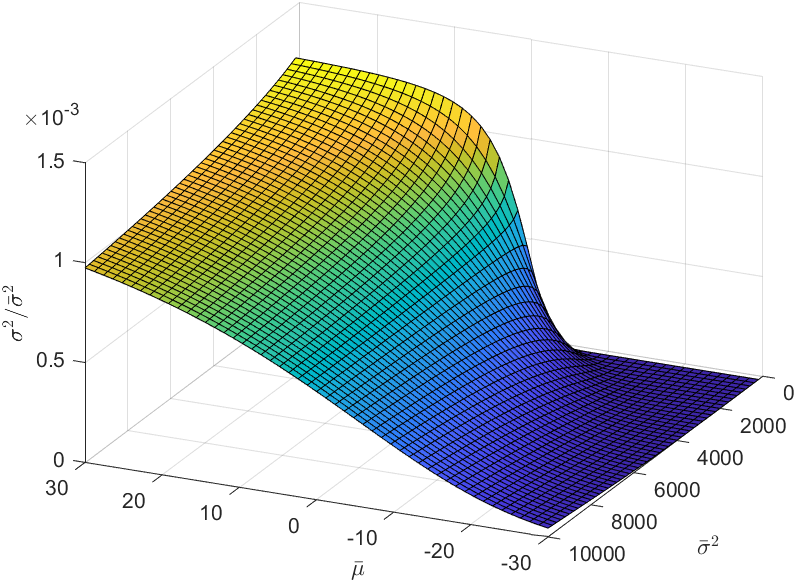
To find this limit, again perform the following expansion:

Where

Therefore,

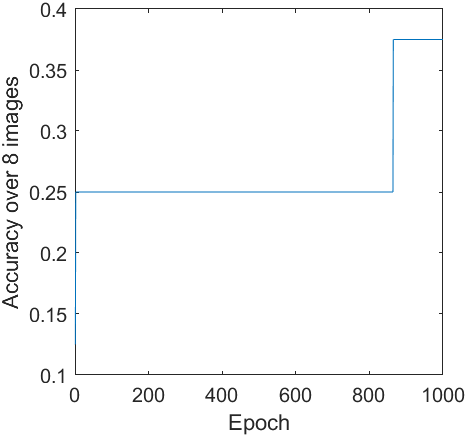
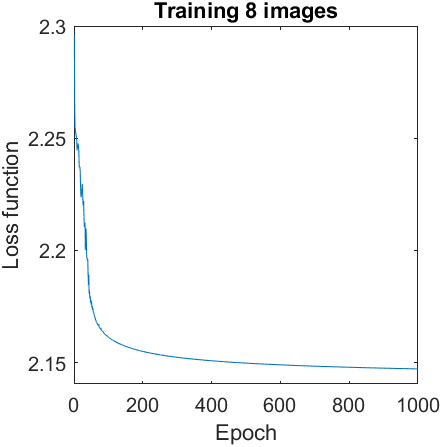


Numeric evaluation of activation function is extremely stable over a wide range of input values. I don’t understand how warning for infinity or nan could occur. Where did it appear?



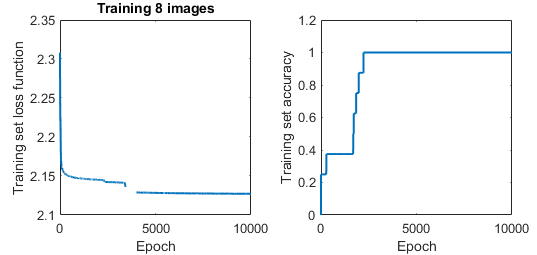
Training results

Case 1: MNIST. Train only 8 images over 1000 epoch



0001\_mnist\_8\_images\_test.mat

Improved results over 1e4 epoch.



Test set of 80 images show that the test accuracy is: **0.425** (note since only 8 training samples are used whereas there are a total of 10 digits, theoretical max accuracy is 0.8)

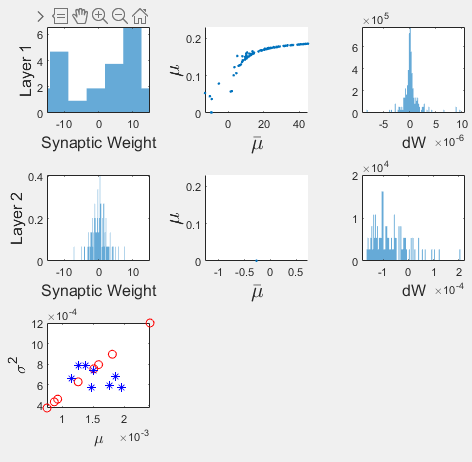
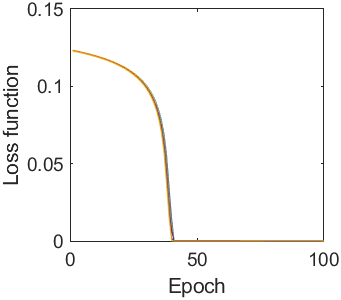
During the training, the loss function end up with NaN due to failed numerical integration of certain samples. But for some reason it recovered shortly afterwards (probably dW who are NaN has being treated as 0 so other ‘healthy’ samples can continue push for learning.)

**Training MNN for optimal Bayesian inference**

0002\_reg\_8x10\_samples.mat

Randomly generated 80 samples (10 minibatches each with size 8).

Then target scaled appropriately such that

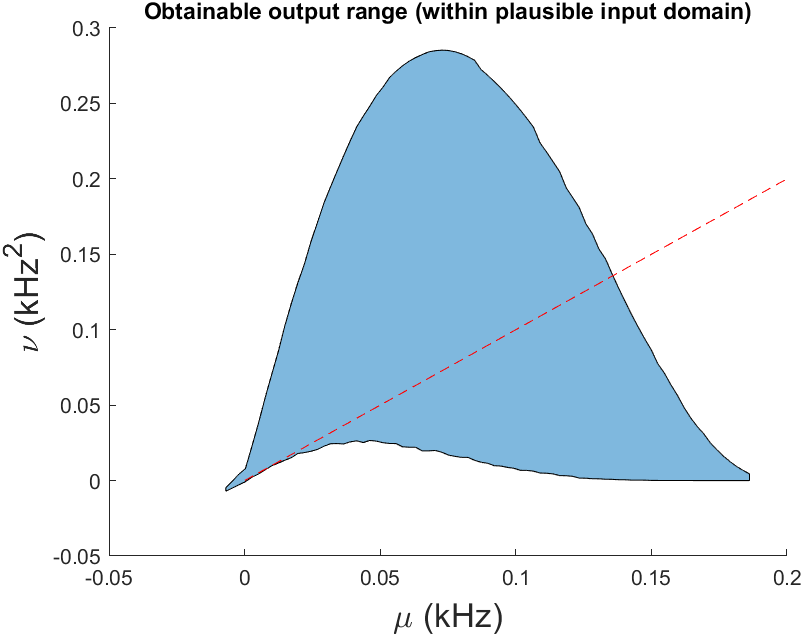


The loss can be reduced without much problem. However, the output won’t hit exactly the target and the relative error remains high. It seems that output mean and var are highly correlated.

Regarding the optimal operation range for MNN

The dynamic range of the activation function is limited. It’s a good idea to tune the network to operate in optimal range (especially so for regression-type training).

Red dashed line: Poisson output (is this line invariant for different time window used for spike count?)



[plotted with output\_uv\_analysis.m]

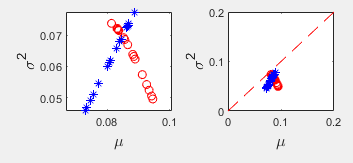
Regression

u1 = 0.1; %kHz, must be in biologically plausible range

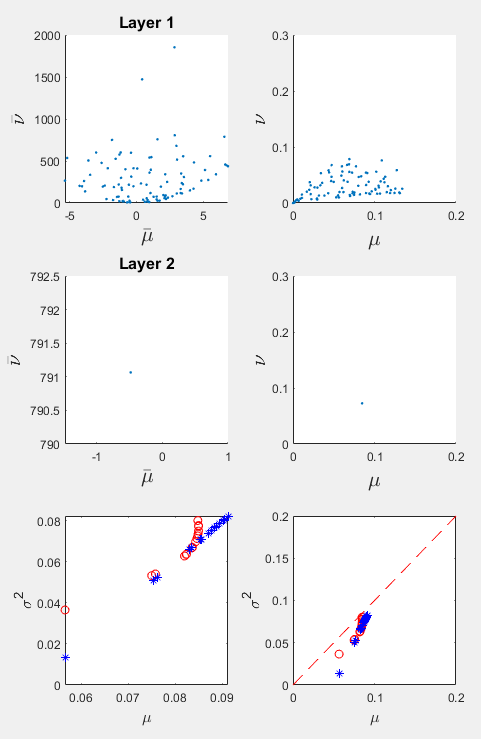
u2 = 0.05;

v1 = 0.1;

v2 = 0.25 + 0.1\*randn;



Another regression



[0006\_reg\_16x600\_samples.mat]

Regarding vanishing gradient

<https://tungmphung.com/sigmoid-tanh-activations-and-their-loss-of-popularity/#:~:text=The%20sigmoid%20and%20tanh%20activation%20functions%20were%20very,post%2C%20we%20explore%20the%20reasons%20for%20this%20phenomenon.>

Batch normalization

Where is the neural activity of neurons within a layer across minibatches.

To take full advantage of the tensor notation, we write

Where is the neural index (within a layer), is the batch index, and is the moment index.

A note on MATLAB implementation:

Index m = 1,2 will be given separated variable names

Index will be a matrix

The forward pass is:

The backward pass

Where repeated indices (except i since normalization is occurs for each neuron) are summed over.