On Gradient Descent and Local vs. Global Optimum

The Loss Surfaces of Multilayer Networks

Anna Choromanska achoroma@cims.nyu.edu
Mikael Henaff mbh305@nyu.edu
Michael Mathieu mathieu@cs.nyu.edu
Gérard Ben Arous benarous@cims.nyu.edu
Yann LeCun yann@cs.nyu.edu

Courant Institute of Mathematical Sciences
New York, NY, USA

Abstract

We study the connection between the highly non-convex loss function of a simple model of the fully-connected feed-forward neural network and the Hamiltonian of the spherical spin-glass model under the assumptions of: i) variable independence, ii) redundancy in network parametrization, and iii) uniformity. These assumptions enable us to explain the complexity of the fully decoupled neural network through the prism of the results from random matrix theory. We show that for large-size decoupled networks the lowest critical values of the random loss function form a layered structure and they are located in a well-defined band lower-bounded by the global minimum. The number of local minima outside that band diminishes exponentially with the size of the network. We empirically verify that the mathematical model exhibits similar behavior as the computer simulations, despite the presence of high dependencies in real networks. We conjecture that both simulated annealing and SGD converge to the band of low critical points, and that all critical points found there are local minima of high quality measured by the test error. This emphasizes a major difference between large- and small-size networks where for the latter poor quality local minima have non-zero probability of being recovered. Finally, we prove that recovering the global minimum becomes harder as the network size increases and that it is in practice irrelevant as global minimum often leads to overfitting.

1 Introduction

Deep learning methods have enjoyed a resurgence of interest in the last few years for such applications as image recognition [Krizhevsky et al., 2012], speech recognition [Hinton et al., 2012], and natural language processing [Weston et al., 2014]. Some of the most popular methods use multi-stage architectures composed of alternated layers of linear transformations and max function. In a particularly popular version, the max functions are known as ReLUs (Rectified Linear Units) and compute the mapping $y = \max(x, 0)$ in a pointwise fashion [Nair and Hinton, 2010]. In other architectures, such as convolutional networks [LeCun et al., 1998a] and maxout networks [Goodfellow et al., 2013], the max operation is performed over a small set of variable within a layer.

The vast majority of practical applications of deep learning use supervised learning with very deep networks. The supervised loss function, generally a cross-entropy or hinge loss, is minimized using some form of stochastic gradient descent (SGD) [Bottou, 1998], in which the gradient is evaluated using the back-propagation procedure [LeCun et al., 1998b].

The general shape of the loss function is very poorly understood. In the early days of neural nets (late 1980s and early 1990s), many researchers and engineers were experimenting with relatively small networks, whose convergence tends to be unreliable, particularly when using batch optimization. Multilayer neural nets earned a reputation of being finicky and unreliable, which in part caused the community to focus on simpler method with convex loss functions, such as kernel machines and boosting.

However, several researchers experimenting with larger networks and SGD had noticed that, while multilayer nets do have many local minima, the result of multiple experiments consistently give very similar performance. This suggests that, while local minima are numerous, they are relatively easy to find, and they are all more or less equivalent in terms of performance on the test set. The present paper attempts to explain this peculiar property through the use of random ma-

Appearing in Proceedings of the 18^{th} International Conference on Artificial Intelligence and Statistics (AISTATS) 2015, San Diego, CA, USA. JMLR: W&CP volume 38.

We conjecture that both simulating annealing and SGD converge to the band of low criticial points, and that all criticial points found are local minima of high quality measured by the test error. ... it is in practice irrelevant as global minimum often leads to overfitting.

Note: Critical points are *maxima*, *minima*, and *saddle points*.

Activation functions

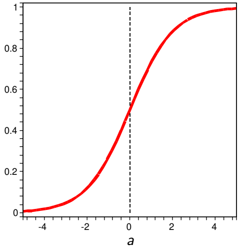
Discrimination functions of the form $y(\mathbf{x}) = \mathbf{w}^T\mathbf{x} + w_0$ are simple linear functions of the input variables $\mathbf{x}$, where distances are measured by means of the dot product.

Let us consider the non-linear *logistic sigmoid* activation function $g(\cdot)$ for limiting the output to $(0, 1)$, that is,

$$y(\mathbf{x}) = g(\mathbf{w}^T\mathbf{x} + w_0),$$

where

$$g(a) = \frac{1}{1 + \exp(-a)}$$

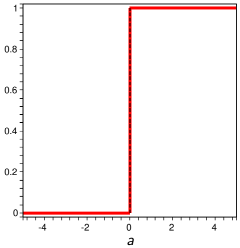


Single-layer network with a logistic sigmoid activation function can also output probabilities (rather than geometric distances).

Activation functions (cont.)

Heaviside step function:

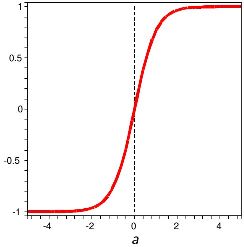
$$g(a) = \begin{cases} 0 & \text{if } a < 0 \\ 1 & \text{if } a \geq 0 \end{cases}$$



Hyperbolic tangent function:

$$g(a) = \tanh(a) = \frac{\exp(a) - \exp(-a)}{\exp(a) + \exp(-a)}$$

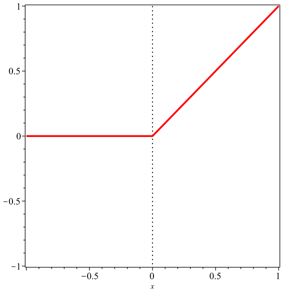
Note, $\tanh(a) \in (-1, 1)$



Activation functions (cont.)

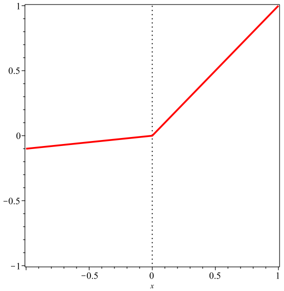
Rectified Linear Unit (ReLU) function:

$$g(a) = \max(0, a)$$



Leaky ReLU

$$g(a) = \max(0.1 \cdot a, a)$$



Online/Mini-Batch/Batch Learning

Online learning:

- Update weight $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \eta \frac{\partial E^{(i)}}{\partial \mathbf{w}}$ (pattern by pattern).

This type of online learning is also called *stochastic gradient descent*, it is an approximation of the true gradient.

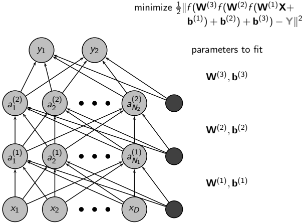
Mini-Batch Learning: Partition $\mathcal{X}$ randomly in subsets $\mathcal{B}^1, \mathcal{B}^2, \ldots, \mathcal{B}^S$ and

- Update weight $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \eta \frac{1}{|\mathcal{B}^s|} \sum_s^S \frac{\partial E^{(s)}}{\partial \mathbf{w}}$ by computing derivatives for each pattern in subset $\mathcal{B}^s$ separately and then sum over all patterns in $\mathcal{B}^s$.

Batch learning:

- Update weight $\mathbf{w}^{(i+1)} = \mathbf{w}^{(i)} - \eta \frac{1}{N} \sum_{n=1}^N \frac{\partial E^{(n)}}{\partial \mathbf{w}}$ by computing derivatives for each pattern separately and then sum over all patterns.

Learning in Neural Networks with Backpropagation



Core idea:

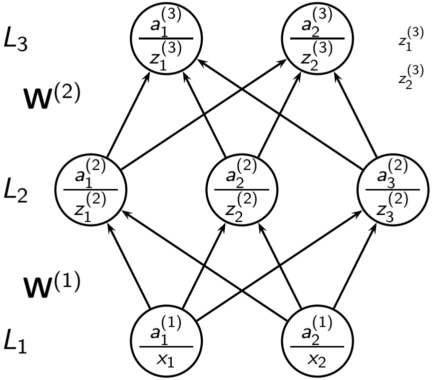
- Calculate error of loss function and change weights and biases based on output.
- These "error" measurements for each unit can be used to calculate the partial derivatives.
- Use partial derivatives with gradient descent for updating weights and biases and minimizing loss function.

Problem: At which magnitude one shall change e.g. weight $W_{ij}^{(1)}$ based on error of y_2?

Learning in Neural Networks with Backpropagation (cont.)

Input: x_1, x_2, output: $a_1^{(3)}, a_2^{(3)}$, target: y_1, y_2 and $g(\cdot)$ is activation function. NN calculates[2] $g(\mathbf{W}^{(2)} g(\mathbf{W}^{(1)}\mathbf{x}))$.

$$E(\mathbf{W}) = \frac{1}{2}\left[(a_1^{(3)} - y_1)^2 + (a_2^{(3)} - y_2)^2\right] = \frac{1}{2}\|\mathbf{a}^{(3)} - \mathbf{y}\|^2$$



$$z_1^{(3)} = W_{10}^{(2)} a_0^{(2)} + W_{11}^{(2)} a_1^{(2)} + W_{12}^{(2)} a_2^{(2)} + W_{13}^{(2)} a_3^{(2)} \qquad a_1^{(3)} = g(z_1^{(2)})$$

$$z_2^{(3)} = W_{20}^{(2)} a_0^{(2)} + W_{21}^{(2)} a_1^{(2)} + W_{22}^{(2)} a_2^{(2)} + W_{23}^{(2)} a_3^{(2)} \qquad a_2^{(3)} = g(z_2^{(2)})$$

Forward pass

$$\underbrace{\mathbf{z}^{(3)}}_{2\times 1} = \underbrace{\mathbf{W}^{(2)}}_{2\times 4}\underbrace{\mathbf{a}^{(2)}}_{4\times 1} \qquad \mathbf{a}^{(3)} = g(\mathbf{z}^{(3)})$$

$$z_1^{(2)} = W_{10}^{(1)} x_0 + W_{11}^{(1)} x_1 + W_{12}^{(1)} x_2 \qquad a_1^{(2)} = g(z_1^{(2)})$$

$$z_2^{(2)} = W_{20}^{(1)} x_0 + W_{21}^{(1)} x_1 + W_{22}^{(1)} x_2 \qquad a_2^{(2)} = g(z_2^{(2)})$$

$$z_3^{(2)} = W_{30}^{(1)} x_0 + W_{31}^{(1)} x_1 + W_{32}^{(1)} x_2 \qquad a_3^{(2)} = g(z_3^{(2)})$$

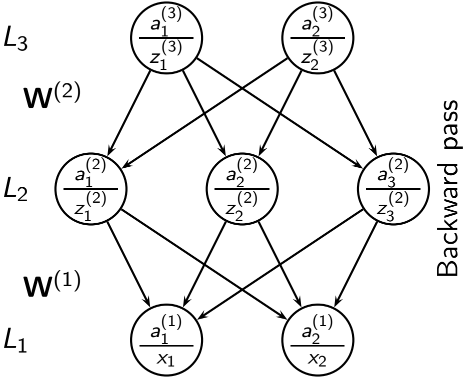
$$\underbrace{\mathbf{z}^{(2)}}_{3\times 1} = \underbrace{\mathbf{W}^{(1)}}_{3\times 3}\underbrace{\mathbf{x}}_{3\times 1} \qquad \mathbf{a}^{(2)} = g(\mathbf{z}^{(2)})$$

[2] Notation adapted from Andew Ng's slides.

Learning in Neural Networks with Backpropagation (cont.)

For each node we calculate $\delta_j^{(l)}$, that is, error of unit j in layer l, because $\frac{\partial}{\partial W_{ij}^{(l)}} E(\mathbf{W}) = a_j^{(l)} \delta_i^{(l+1)}$. Note $\odot$ is element wise multiplication.

$$E(\mathbf{W}) = \frac{1}{2}\left[(a_1^{(3)} - y_1)^2 + (a_2^{(3)} - y_2)^2\right] = \frac{1}{2}\|\mathbf{a}^{(3)} - \mathbf{y}\|^2$$



$$\boldsymbol{\delta}^{(3)} = (\mathbf{a}^{(3)} - \mathbf{y}) \odot g'(\mathbf{z}^{(3)})$$

$$\boldsymbol{\delta}^{(2)} = (\mathbf{W}^{(2)})^T \delta^{(3)} \odot g'(\mathbf{z}^{(2)})$$

Note $\boldsymbol{\delta}^{(1)}$ is the input, so no term.

Learning in Neural Networks with Backpropagation (cont.)

Backpropagation = forward pass & backward pass

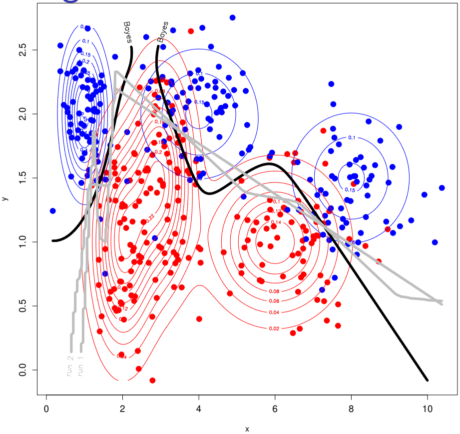
Given labeled training data $(\mathbf{x}_1, \mathbf{y}_1), \ldots, (\mathbf{x}_N, \mathbf{y}_N)$.
Set $\Delta_{ij}^{(l)} = 0$ for all l, i, j. Value Δ will be used as accumulators for computing partial derivatives.
For $n = 1$ to N

- Forward pass, compute $\mathbf{z}^{(2)}, \mathbf{a}^{(2)}, \mathbf{z}^{(3)}, \mathbf{a}^{(3)}, \ldots, \mathbf{z}^{(L)}, \mathbf{a}^{(L)}$
- Backward pass, compute $\boldsymbol{\delta}^{(L)}, \boldsymbol{\delta}^{(L-1)}, \ldots, \boldsymbol{\delta}^{(2)}$
- Accumulate partial derivate terms, $\boldsymbol{\Delta}^{(l)} := \boldsymbol{\Delta}^{(l)} + \boldsymbol{\delta}^{(l+1)}(\mathbf{a}^{(l)})^T$

Finally calculated partial derivatives for each parameter:
$\frac{\partial}{\partial W_{ij}^{(l)}} E(\mathbf{W}) = \frac{1}{N} \Delta_{ij}^{(l)}$ and use these in gradient descent.

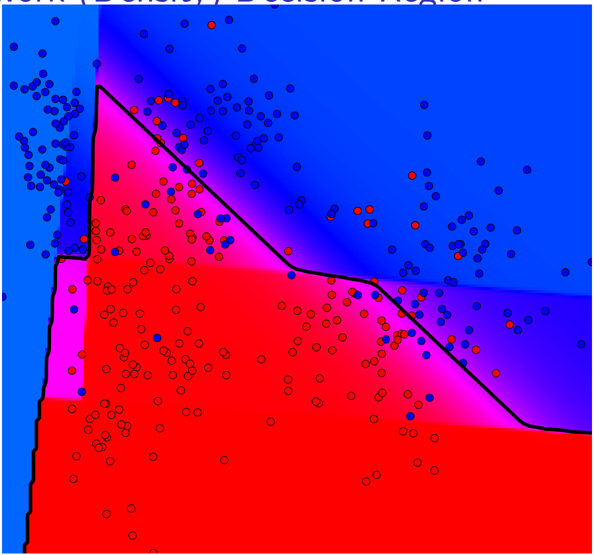
See interactive demo.

Bayes Decision Region vs. Neural Network



Points from blue and red class are generated by a mixture of Gaussians. Black curve shows optimal separation in a Bayes sense. Gray curve shows neural network separation of two independent backpropagation learning runs.

Neural Network (Density) Decision Region



Overfitting/Underfitting & Generalization

Consider the problem of polynomial curve fitting where we shall fit the data using a polynomial function of the form:

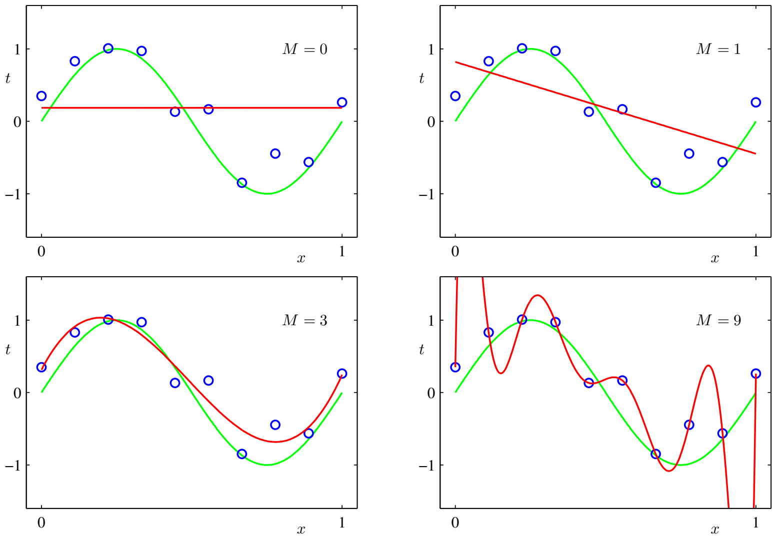
$$y(x, \mathbf{w}) = w_0 + w_1 x + w_2 x^2 + \ldots + w_M x^M = \sum_{j=0}^{M} w_j x^j.$$

We measure the misfit of our predictive function $y(x, \mathbf{w})$ by means of error function which we like to minimize:

$$E(\mathbf{w}) = \frac{1}{2} \sum_{i=1}^{N} (y(x_i, \mathbf{w}) - t_i)^2$$

where t_i is the corresponding target value in the given training data set.

Polynomial Curve Fitting



Polynomial Curve Fitting (cont.)

	$M = 0$	$M = 1$	$M = 3$	$M = 9$
$w_0^\star$	0.19	0.82	0.31	0.35
$w_1^\star$		-1.27	7.99	232.37
$w_2^\star$			-25.43	-5321.83
$w_3^\star$			17.37	48568.31
$w_4^\star$				-231639.30
$w_5^\star$				640042.26
$w_6^\star$				-1061800.52
$w_7^\star$				1042400.18
$w_8^\star$				-557682.99
$w_9^\star$				125201.43

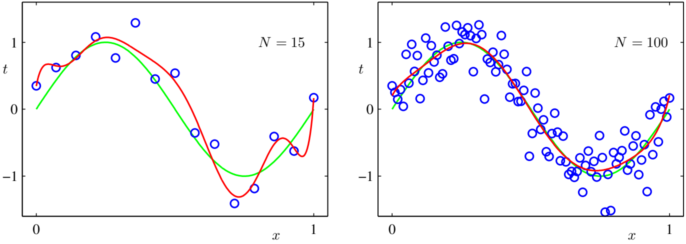
Table: Coefficients $\mathbf{w}^\star$ obtained from polynomials of various order. Observe the dramatically increase as the order of the polynomial increases (this table is taken from Bishop's book).

Polynomial Curve Fitting (cont.)

Observe:

- if M is too small then the model underfits the data
- if M is too large then the model overfits the data

If M is too large then the model is more flexible and is becoming increasingly tuned to random noise on the target values. It is interesting to note that the overfitting problem become less severe as the size of the data set increases.



ImageNet Classification with Deep ConvolutionalNeural Networks: "The easiest and most common method to reduce overfitting on image data is to artificially enlargethe dataset using label-preserving transformation."

Polynomial Curve Fitting (cont.)

One technique that can be used to control the overfitting phenomenon is the *regularization*.

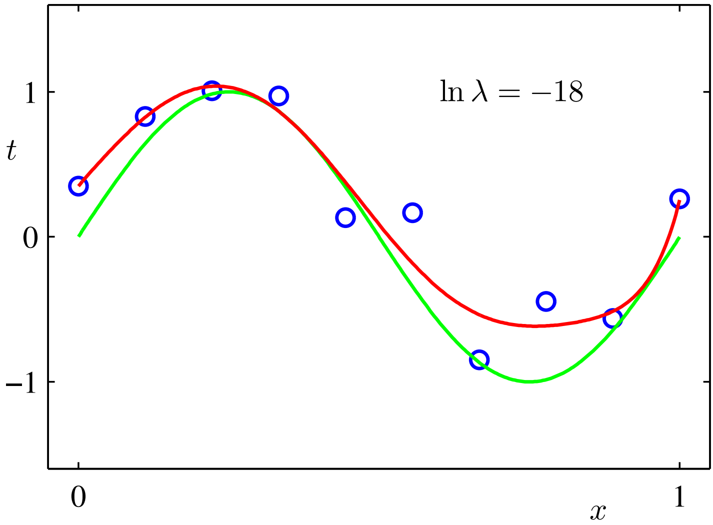
- Regularization involves adding a penalty term to the error function in order to discourage the coefficients from reaching large values.

The modified error function has the form:

$$\widehat{E}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}(y(x_i,\mathbf{w}) - t_i)^2 + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}.$$

By means of the penalty term one reduces the value of the coefficients (shrinkage method).

Regularized Polynomial Curve Fitting $M = 9$



Regularization in Neural Networks

- Number of input/output units is generally determined by the dimensionality of the data set.
- Number of hidden units M is free parameter that can be adjusted to obtain best predictive performance.
- Generalization error is not a simple function of M due to the presence of local minima in the error function.
- One straightforward way to deal with this problem is to increase stepwise the value of M and to choose the specific solution having the smallest test error.

Regularization in Neural Networks (cont.)

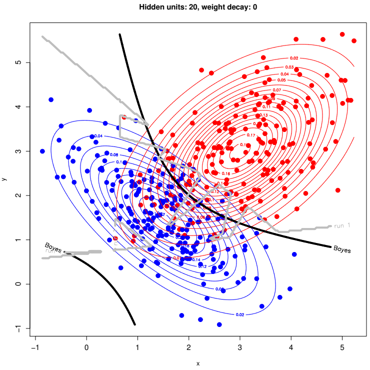
Equivalent to the regularized curve fitting approach, we can choose a relatively large value for M and control the complexity by the addition of a regularized term to the error function.

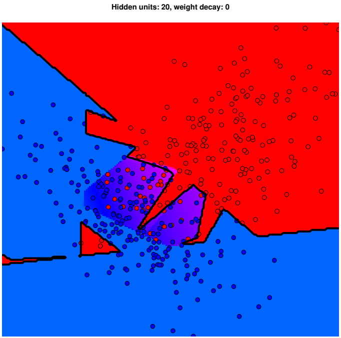
$$\widehat{E}(\mathbf{w}) = E(\mathbf{w}) + \frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

This form of regularization in neural networks is known as *weight decay*.

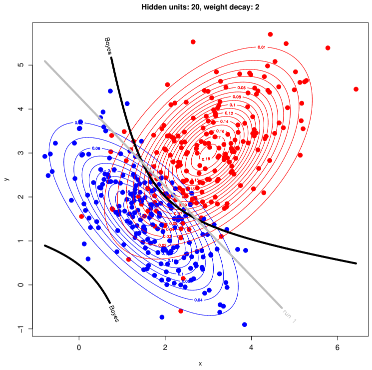
- Weight decay encourages weight values to decay towards zero, unless supported by the data.
- It can be considered as an example of a parameter shrinkage method because parameter values are shrunk towards zero.
- It can be also interpreted as the removal of non-useful connections during training.

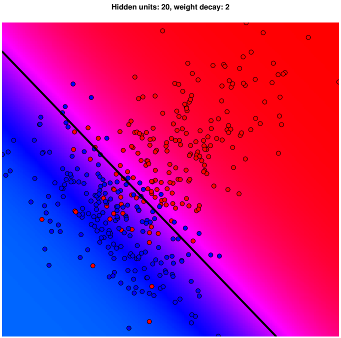
A too Overfitted Neural Network Model



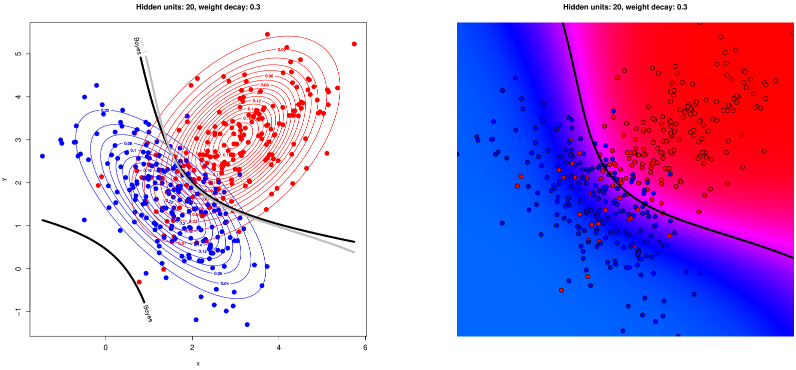


A too Underfitted Neural Network Model



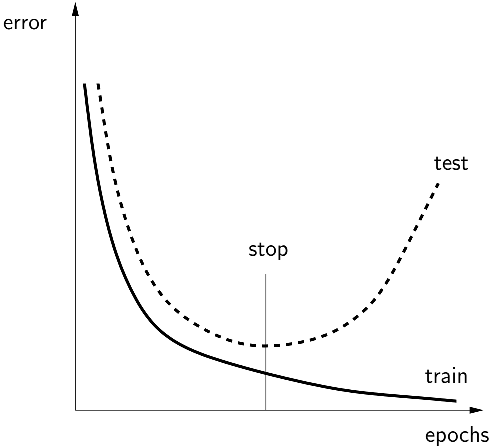


Model Complexity is Properly Penalized

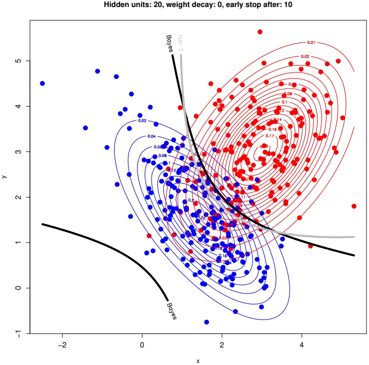


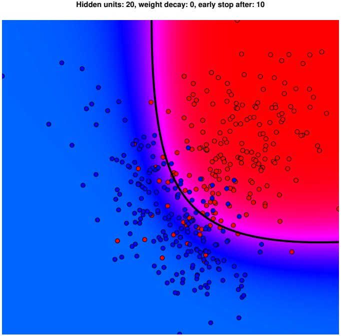
Regularization by Early Stopping

- Another alternative of regularization as a way of controlling the effective complexity of a network is the procedure of *early stopping*.

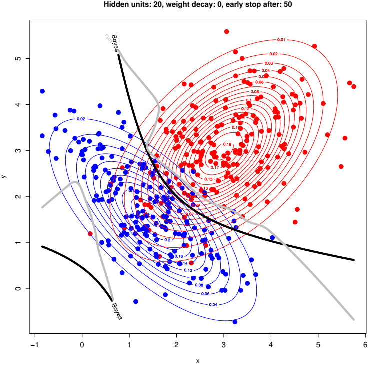


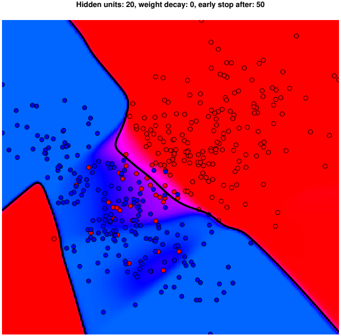
Example Early Stopping after 10 Epochs





Example Early Stopping after 50 Epochs





Example Early Stopping after 100 Epochs