Homework

On Wednesday, go to our class website. There will be instructions on how to submit the homework.

You will need to have your group number, the accuracy on your test data, the time it took to train, and a 2 by 2048 matrix with the weights in the activation functions for each node. The first row of the matrix is for the hidden layer that receives the inputs, and has the form (first weight, first intercept, second weight, second intercept, ..., 1024th weight, 1024th intercept). The second row has the same pattern for the hidden layer that produces the output.

This is an experiment. There has been work that studies how weights/intercepts change in deep networks, and I am curious to see if we can replicate those findings.

3. Back-Propagation and Training

The idea for training a deep NN is to optimize its set of hyperparameters $\left(\text{let } \theta = \{w_1, w_2, w_3, b_1, b_2, b_2\}\right)$ in order to approximate a function of interest. The quality of the approximation is evaluated using a cost function $J(\vec{x}, y; \theta)$ such as mean squared error or KL Divergence.

To adjust these hyperparameters, their gradients are calculated with respect to the cost function. Since any particular w_i or b_i is related to the cost function $J(\vec{x}, y; \theta)$ through a function composition, the **chain rule** is used to calculate partial derivatives. Due to the structure of the network, partial derivatives must be calculated starting at the output layer and working recursively backwards in a process known as **back-propagation**.

Consider the very simple neural network with a single scalar input x, two hidden layers, a single node at each layer, and differentiable activation functions:

$$x \longrightarrow h_1 \longrightarrow h_2 \longrightarrow y$$

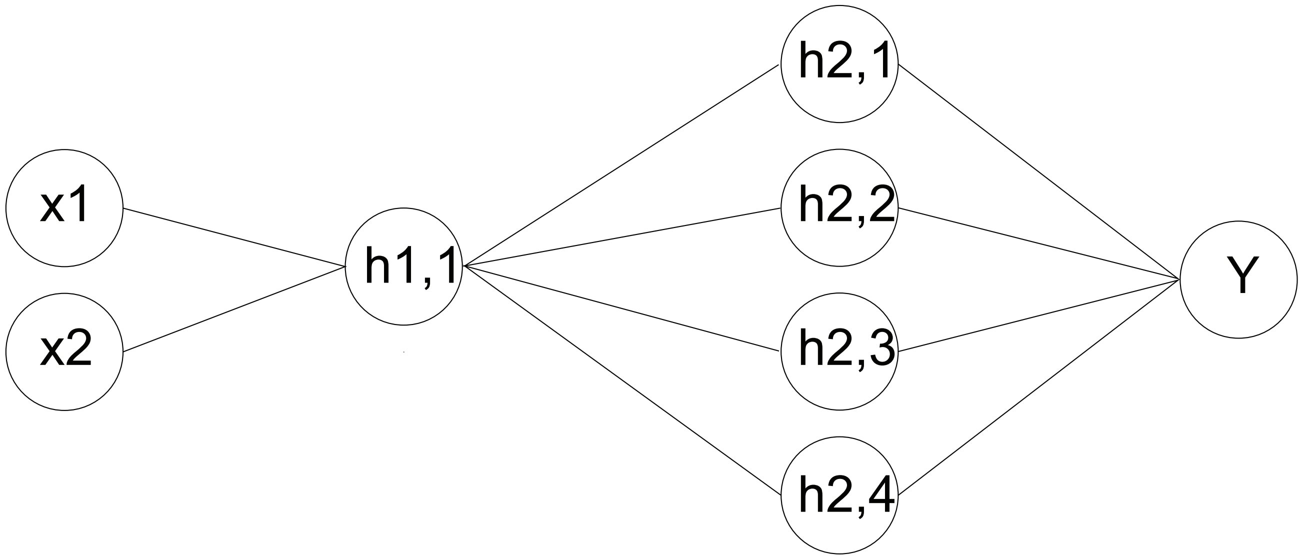
$$x \longrightarrow \sigma_1\big(w_1 x + b_1\big) \longrightarrow \sigma_2\big(w_2 h_1 + b_2\big) \longrightarrow \sigma_{out}\big(w_3 h_2 + b_3\big)$$

The $\frac{\partial J(\vec{x},y;\,\theta)}{\partial y}$ is easily observable (since we directly observe the output layer), however the chain rule must be used to find the partial derivatives of each $\frac{\partial J(\vec{x},y;\,\theta)}{\partial w_i}$ or $\frac{\partial J(\vec{x},y;\,\theta)}{\partial b_i}$. In the example above,

$$\frac{\partial J(\vec{x},y;\,\theta)}{\partial w_1} = \frac{\partial J(\vec{x},y;\,\theta)}{\partial y}\frac{\partial y}{\partial h_2}\frac{\partial h_2}{\partial h_1}\frac{\partial h_1}{\partial w_1}$$

Chain Rule: $\frac{d}{dx}f(g(x)) = f'(g(x))g'(x)$.

An Example



If $h_{1,1} = \sigma(w_{1,1}x_1 + w_{1,2}x_2 + b_1)$, then $\frac{\partial J(\vec{x},y;\,\theta)}{\partial h_{1,1}}$ can be calculated as

$$\frac{\partial J(\vec{x},y;\,\theta)}{\partial h_{1,1}} = \frac{\partial J(\vec{x},y;\,\theta)}{\partial y} \sum_{i=1}^{4} \frac{\partial y}{\partial h_{2,i}} \frac{\partial h_{2,i}}{\partial h_{1,1}}.$$

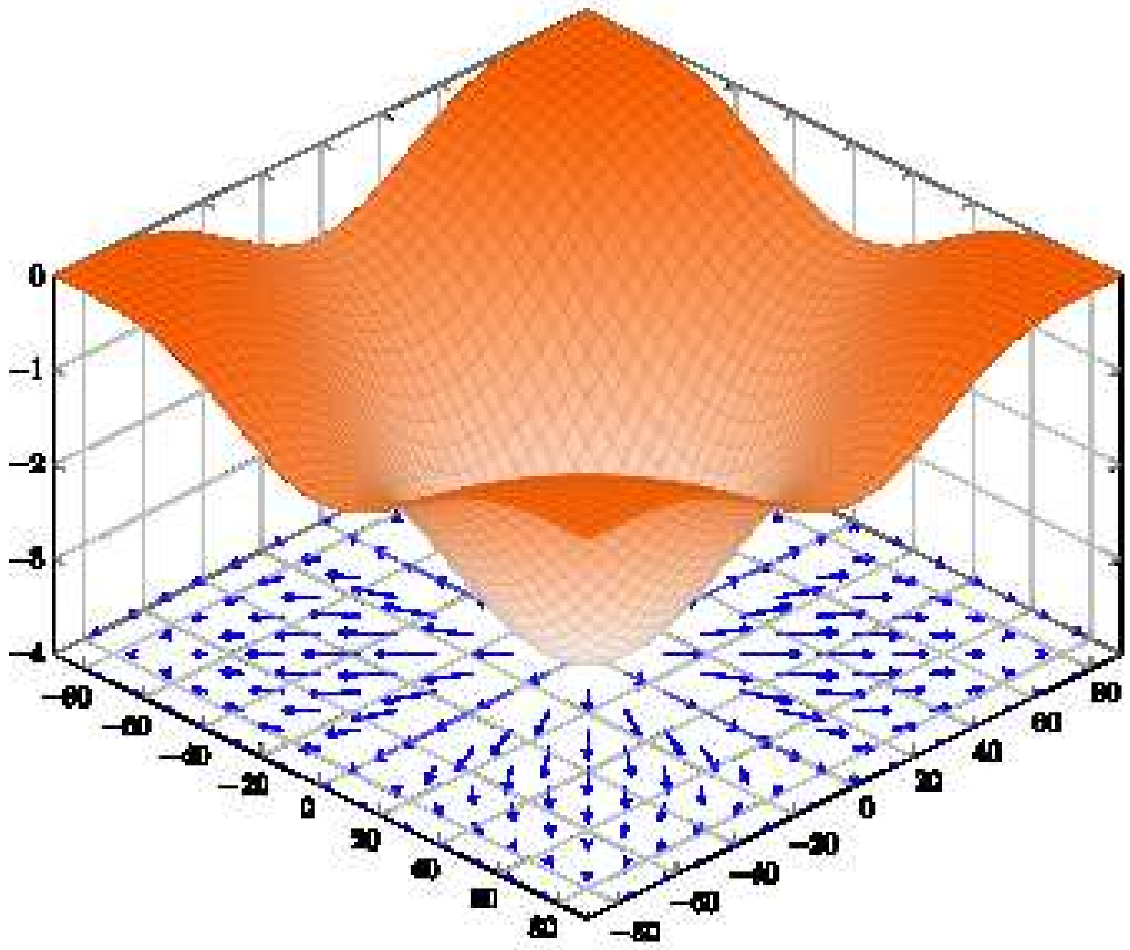
Once this has been found, the gradient of the cost function with respect to $\vec{w}_1$ can be calculated using back-propagation through

$$\nabla_{\vec{w}} J(\vec{x}, y;\, \theta) = \left(\, \nabla_{\vec{w}}\, h_{1,1}\right) \frac{\partial J(\vec{x}, y;\, \theta)}{\partial h_{1,1}}$$

Recall that for a three-dimensional system,

$$\nabla f(x, y, z) = \frac{\partial f}{\partial x}(1,0,0) + \frac{\partial f}{\partial y}(0,1,0) + \frac{\partial f}{\partial z}(0,0,1).$$

The gradient at a point gives the direction of steepest change in the function.


0
−1
−2
−3
−4
−80
−60
−40
−20
0
20
40
60
80
−80
−60
−40
−20
0
20
40
60
80

The General Back-Propagation Algorithm treats each node in the computational graph of a neural network not as scalars, vectors, or matrices, but as tensors of appropriate dimensions.

Tensor calculus is exactly the same as vector calculus. We denote the gradient of output z with respect to a tensor $\mathbf{X}$ as $\bigtriangledown_{\mathbf{X}} z$. In a sense, we have flattened out the tensor $\mathbf{X}$ as a vector. Just as a vector has integer indices and a matrix entry is indexed by its row and column, a 3-dimensional tensor has indices that are triples of integers. If $\mathbf{Y} = g(\mathbf{X})$ and $z = f(\mathbf{Y})$, then the chain rule for the tensor is:

$$\bigtriangledown_{\mathbf{X}} z = \sum_{j \in \{\textbf{index set of } \mathbf{Y}\}} \left(\bigtriangledown_{\mathbf{X}} \mathbf{Y}_j \times \frac{\partial z}{\partial \mathbf{Y}_j} \right)$$

If $\mathbf{X} = \mathbf{AB}$ and $z = f(\mathbf{X})$ and $\bigtriangledown_{\mathbf{X}} z = \mathbf{G}$ then

$$\bigtriangledown_{\mathbf{A}} z = \mathbf{G}\mathbf{B}^T$$
$$\bigtriangledown_{\mathbf{B}} z = \mathbf{A}^T\mathbf{G}$$

The intuition behind back-propagation is not about the rules of differentiation. Let u_1 be the input, u_n be the output, and $u_{2\leq i\leq n-1}$ be the intermediate nodes in a computation graph. Let D=$\{\bigtriangledown_{u_i} u_n \mid i = 1, \ldots, n\}$. The goal is to find the weights and intercepts by working backwards from u_n all the way to u_1, moving (sort of) in the direction of most improvement. The gradients of all parameters are needed in doing optimization.

The “sort of” reflects the fact that stochastic gradient descent takes steps in the best direction at a given point. It does not move continuously.

3.1 Some Back-Propagation Details

There are many other issues and technicalities that arise in training a deep network.

One common practice is to scale all the inputs to lie between 0 and 1. Unscaled inputs can slow or destabilize the training. The transformation is $(x - \text{min})/(\text{max} - \text{min})$.

It is also good to scale the outputs to lie between 0 and 1. Unscaled outputs in regression problems can cause exploding gradients, which also cause the training to fail.

Scaling the data has the useful property of putting all the weights and intercepts in the nodes on a common footing, so that one can better assess the important variables that contribute to the output.

A difficulty with scaling arises when one applies the trained network to new data. The new data, when given the same scaling as was used with the training data, may not lie between 0 and 1. If the overage or underage is small, that is not a serious problem. If it is large, there can be an issue.

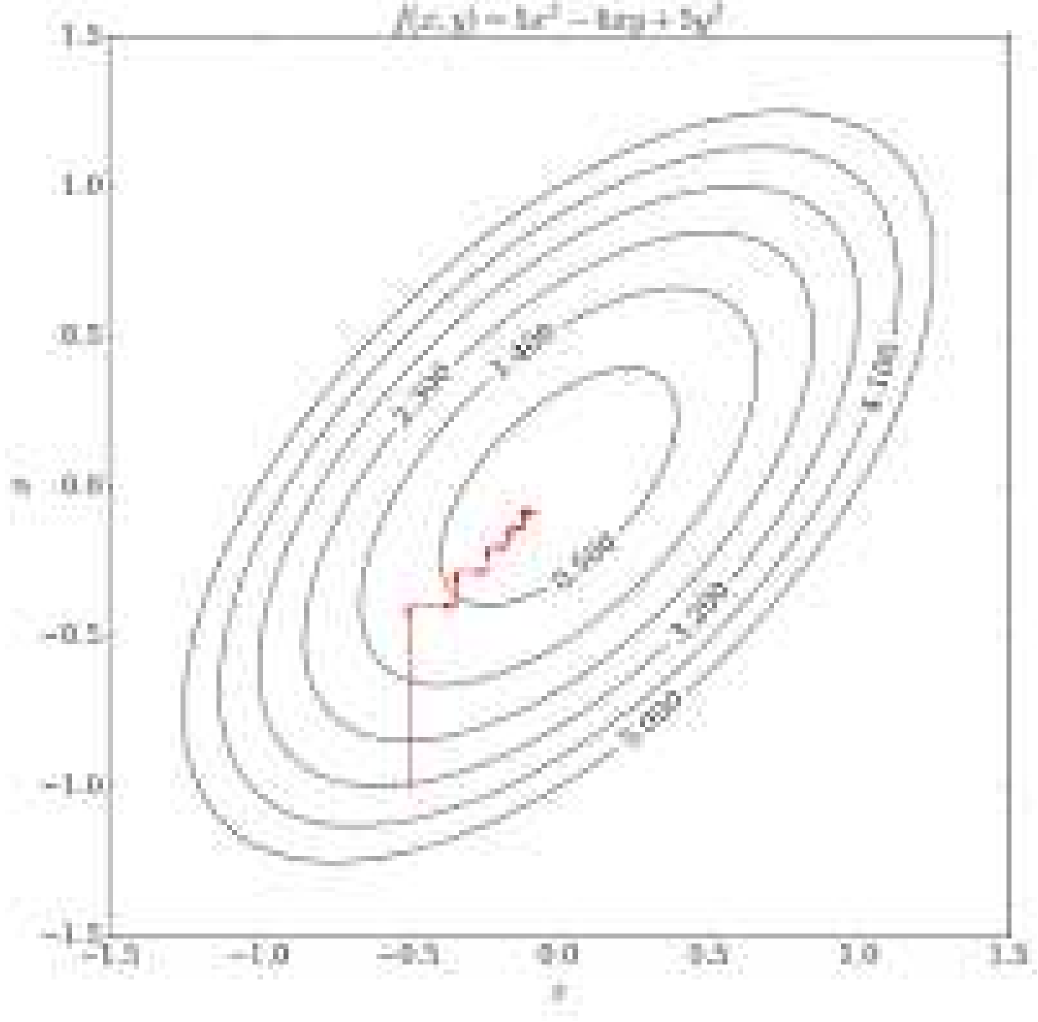
With some BIG data sets, it may be hard to find the maximum and the minimum.

Some people normalize the input and output data by subtracting the mean and dividing by the standard deviation, but this makes the strong assumption that the inputs and outputs have a Gaussian distribution.

Our previous discussion of back-propagation implicitly assumed that all of the data were used in training the NN. In practice, that is time consuming.

Recall optimization by coordinate gradient descent. One cycles through the components of the input vectors, taking the derivative with respect to each coordinate, and moving to the location on the cost-function surface that minimizes the cost function along that coordinate.

With deep NNs, there are millions of coordinates and often millions of data points. Gradient descent is too slow. Instead, one uses **stochastic gradient descent** (SGD).


$f(x,y) = 5x^2 - 8xy + 5y^2$
1.5
1.0
0.5
0.0
−0.5
−1.0
−1.5
y
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
x
0.400
1.200

In SGD, one trains with subsamples of the data, called **minibatches**. (Batch methods train with all the data, on-line methods train with one observation at a time, and minibatch methods are the compromise used in essentially all deep NN training.)

The size of the minibatch is determined by several factors:

- Large batches more accurately estimate the gradient.
- Small batches underuse multicore processors.
- For parallel processing, memory demands scale with minibatch size.
- With GPUs, batch sizes that are powers of 2 are efficient (32 to 256).
- Wilson and Martinez (2003) report that small minibatches regularize.

The standard approach is to choose the data in the minibatches at random from the training data. But as a statistician, I wonder if it is smarter to choose training data that are very dissimilar (at first) with respect to an appropriate metric, and then becoming more similar.

However, in many datasets, the initial ordering of the data can show significant dependence (e.g., successive frames in a movie, digits written by the same person). Such dependence introduces bias when estimating the gradient.

The standard approach is to randomly rearrange the initial data, and then work with the rearrangement going forward.

The SGD algorithm is:

1. Set a learning rate ϵ_k.
2. Pick an initial parameter value of weights and intercepts $\vec{\theta}_1$.
3. For $k = 1, \ldots,$stopping criterion
   - sample a minibatch of size m, $(\vec{x}^{(1)}, \ldots, \vec{x}^{(m)})$ and corresponding $(y^{(1)}, \ldots, y^{(m)})$
   - estimate the gradient $\hat{g} = \frac{1}{m}\big(\bigtriangledown_{\vec{\theta}} \sum_{i=1}^{m} J(\vec{x}^{(i)}, y^{(i)}, \vec{\theta}_k)$
   - update so $\vec{\theta}_{k+1} \leftarrow \vec{\theta}_k - \epsilon_k \hat{g}$.

The learning rate determines how much the parameter changes in the direction of the estimated gradient.

Usually, for k between 1 and τ, one sets

$$\epsilon_k = (1 - \alpha_k)\epsilon_0 + \alpha_k \epsilon_\tau$$

for $\alpha_k = k/\tau$. After the τth iteration, it is constant. But there are reasons to let the step size go to zero.

To choose ϵ_0, ϵ_τ and τ, one can monitor a plot of the approximate empirical cost function against time. Commonly, ϵ_τ is 1% of ϵ_0 and τ is perhaps 300.

If ϵ_0 is too large, then SGD is unstable and wildly overshoots the true minimizer and then overcorrects in the next iteration.

The number of training iterations is called the number of **epochs**.

The previous work assumed that the cost function was differentiable (which requires differentiable activation functions). But ReLU and many other activation functions are not differentiable.

The optimization world has many tricks for handling non-differentiable cost functions. Subgradients are one tool. If $f : D \to \mathbb{R}$ is a real-valued convex function, then a vector $\vec{v} \in \mathbb{R}^d$ is a subgradient at a point x_0 if for all $\vec{x} \in D$ one has $f(\vec{x}) - f(\vec{x}_0) \geq \vec{v}'(\vec{x} - \vec{x}_0)$.

When the cost function is not convex, other tricks are available, especially when the cost function is a difference of two convex functions (SCAD, LASSO, and many others). The Difference of Convex Functions Algorithm applies.

Another trick is to use weight decay. This performs regularization that improves generalization and thus predictive accuracy on the test data.

In weight decay, after each iteration, one multiplies the weights by a number slightly less than 1. This is very much like the ridge regression penalty on the sum of the squares of the coefficients in the regression. It is equivalent to including a ridge regression penalty on the cost function

Other complications arise from use of 32-bit versus 64-bit floating point data types. These types of problems are handled almost automatically by TensorFlow and PyTorch.

Cost functions in deep NNs must be written as averages, and they depend only on the weights, intercepts, and training data. Quadratic cost is often used for regression, and is

$$\sum_j (\hat{y}_j - y_j)^2$$

and often has an additional penalty on the sum of the squares or absolute values of the weights and intercepts, to shrink them towards zero.

Similarly, one can have a cost function that reflects mean absolute error. This does not weight large discrepancies as strongly, and it may also have a penalty term. Other options are Huber loss and SCAD.

If one is doing classification with M categories, where the empirical probability in the training sample for the mth category is $\hat{p}_m$, one commonly uses cross entropy.

$$-\sum_{1}^{M}\sum_{j} I(\hat{y}_j \textbf{ correctly labelled}) \ln(\hat{p}_m).$$

Another standard choice is Kullback-Leibler divergence, or

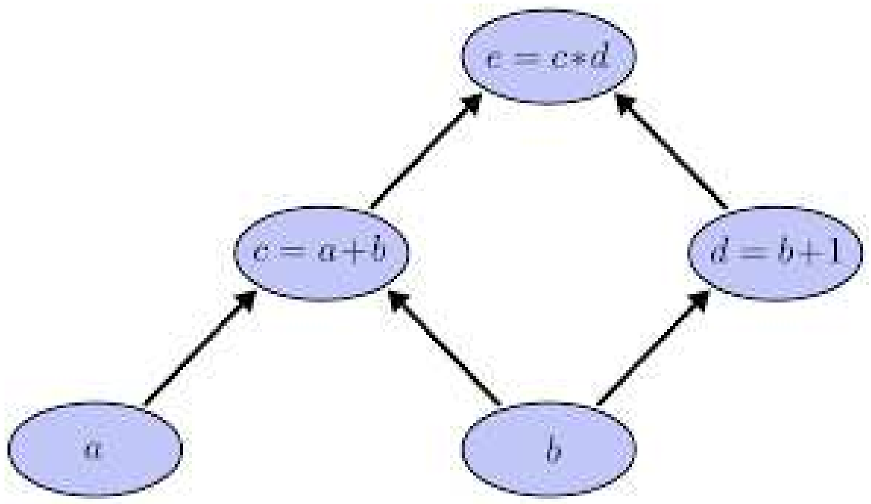
$$\sum_{m} = 1^{M}\hat{p}_m \ln \hat{p}_m / q_m$$

where q_m is the proportion of the minibatch or training sample that is correctly classified. (This can be written as an average.)

A **computational graph** is a way of representing the steps in a calculation. Each node in the graph represents a variable (scalar, vector, matrix, tensor, or something more complicated).

When a variable y is found by applying an operation to variable x, one draws a directed edge from x to y.

The computational graph helps to represent complex calculations



Consider an NN with a single hidden layer. We train it with minibatch SGD. Each minibatch is represented by $\boldsymbol{X}$ with labels $\boldsymbol{y}$.

To simplify things, assume all intercepts are 0. We then compute the hidden features as $\max\{0, \boldsymbol{X}\boldsymbol{W}^{(1)}\}$ (i.e., we use ReLU activation).

The predictions of the unnormalized log-probabilities are $\boldsymbol{H}\boldsymbol{W}^{(2)}$. We use a cross-entropy operation to compute the cost between the targets $\boldsymbol{y}$ and the probability distribution determined by the unnormalized log-probabilities.

The cross-entropy defines the cost, and minimizing it performs maximum likelihood estimation for the classifier. But it is good to add a regularization term with penalty λ, so the total cost is

$$J = J_{MLE} + \lambda\left(\sum_{i,j}(\mathbf{W^{(1)}_{(i,j)}})^{\mathbf{2}} + (\mathbf{W^{(2)}_{(i,j)}})^{\mathbf{2}}\right)$$

$$\frac{\partial J}{\partial \mathbf{W^{(2)}}} = 2\lambda\mathbf{W^{(2)}} + \mathbf{H^T}\frac{\partial \mathbf{J}}{\partial \mathbf{U^{(2)}}}$$

$$\frac{\partial J}{\partial \mathbf{W^{(1)}}} = 2\lambda\mathbf{W^{(1)}} + \mathbf{X^T}\frac{\partial\left(\frac{\partial \mathbf{J}}{\partial \mathbf{U^{(2)}}}(\mathbf{W^{(2)}})^{\mathbf{T}}\right)}{\partial \mathbf{U^{(1)}}}$$

This training is represented in the following computational graph.

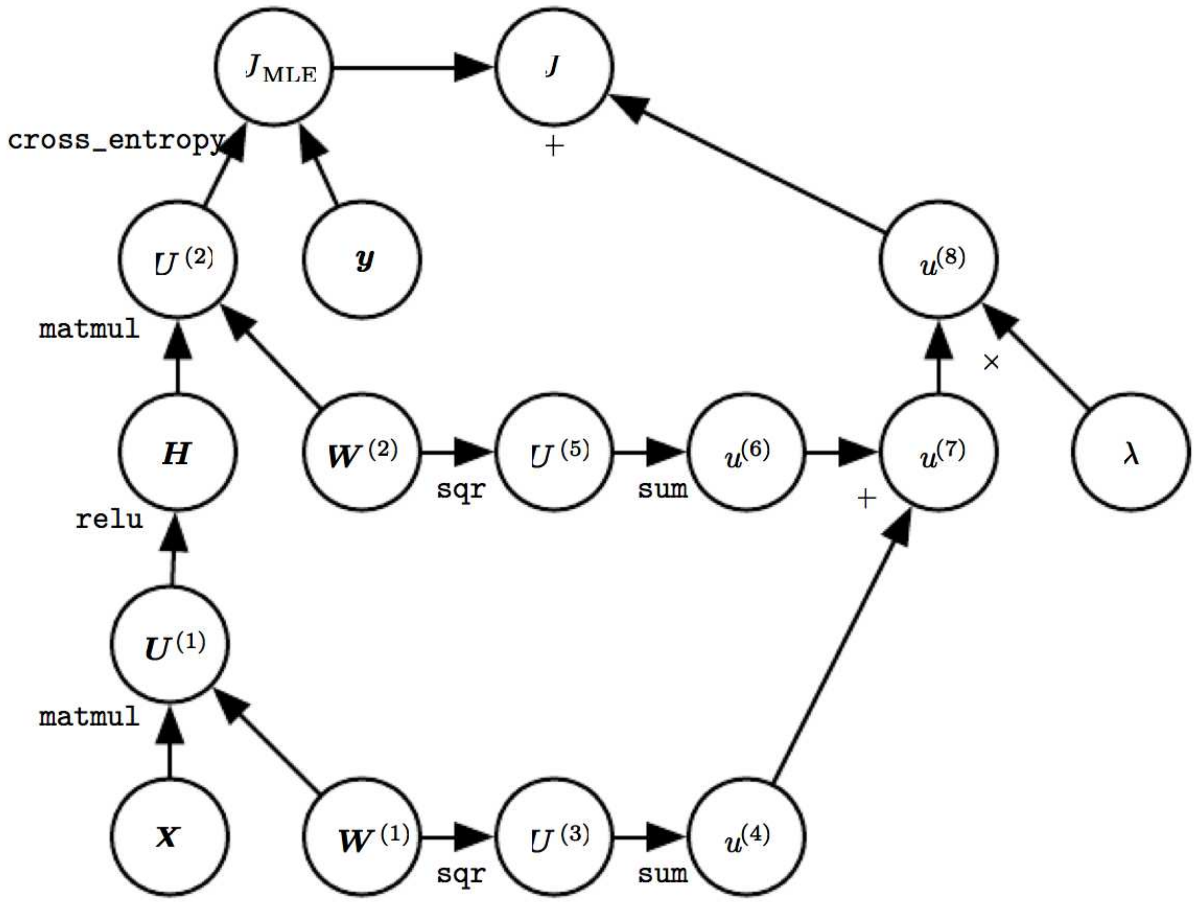


Figure 6.11: The computational graph used to compute the cost used to train our example of a single-layer MLP using the cross-entropy loss and weight decay.

3.2 Network Architecture

Deeper networks with the same number of units but more layers have empirically been found to often **generalize** better. This means that the perform better on test sets.

This is particularly the case for problems with many abstractions (such as image recognition) and this occurs because the deeper network can construct more complex functions through larger numbers of compositions.

Often the most impactful parameter to tune when training a network is its depth.

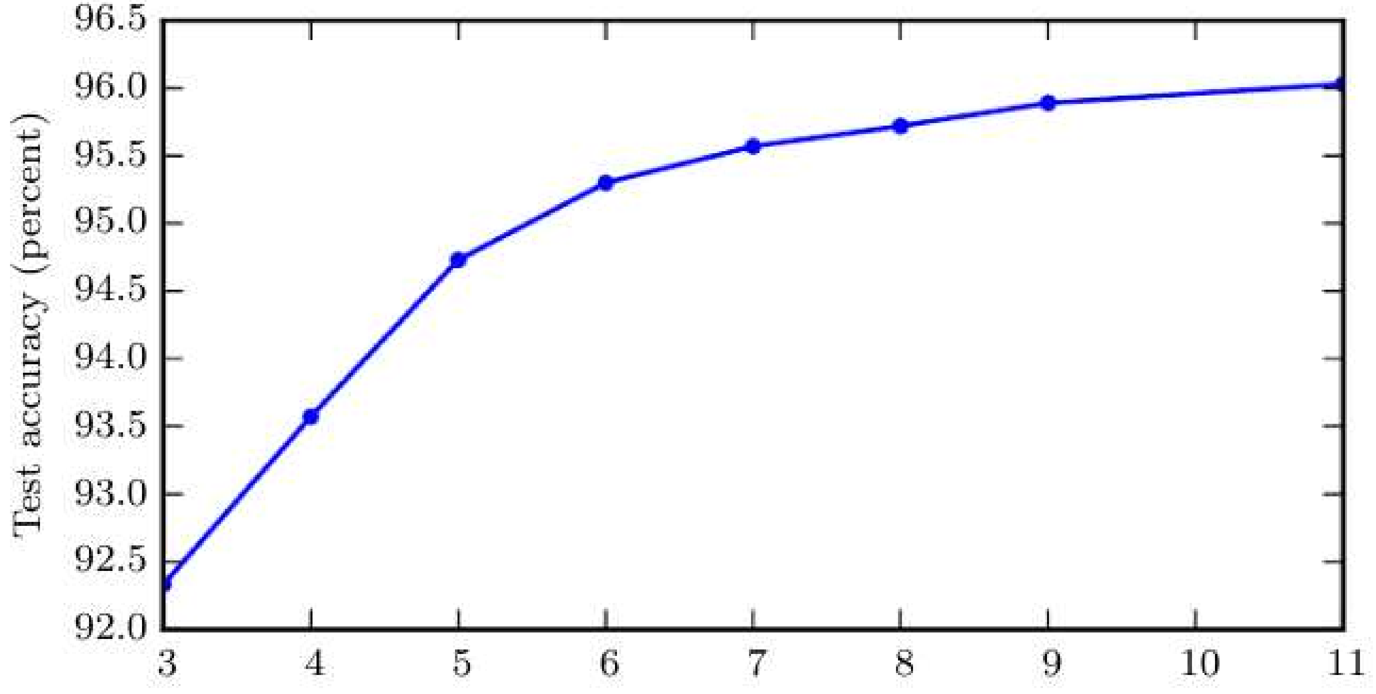


Figure 6.6: Effect of depth. Empirical results showing that deeper networks generalize better when used to transcribe multidigit numbers from photographs of addresses. Data from Goodfellow *et al.* (2014d). The test set accuracy consistently increases with increasing depth. See figure 6.7 for a control experiment demonstrating that other increases to the model size do not yield the same effect.

NB: To train a deeper or larger network sufficient amounts of data must exist to prevent overfitting.

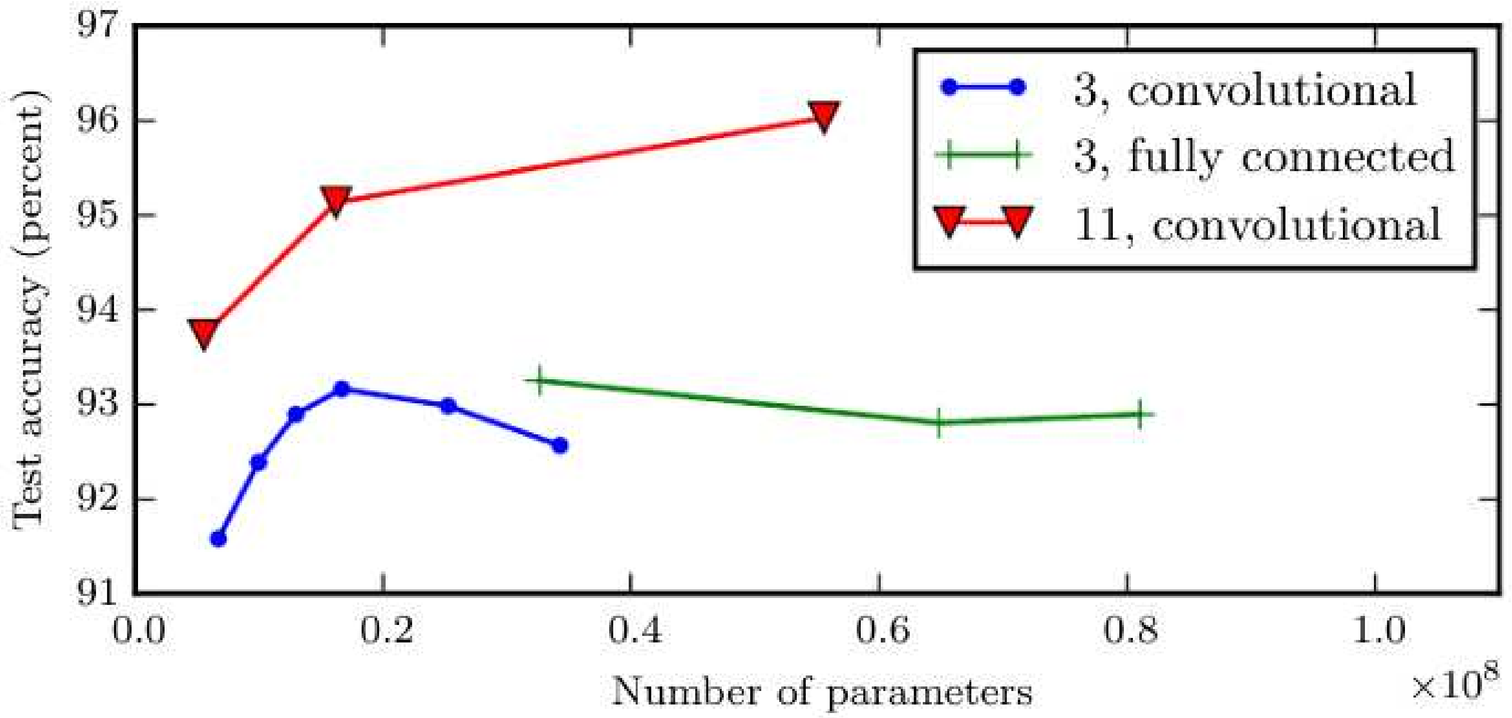


Fig. 6.7 from (Goodfellow et.al 2015) showing that the depth of layers (represented by the different colored lines) has a much larger effect than increasing the number of units.

However, while there are gains to generalization, deeper networks are harder to train for two reasons:

- The problem is more non-convex, because of a larger number of connections between the layers.
- A deeper network is more likely to experience the **vanishing/exploding gradient problem** when trained with back-propagation.

The capacity of a classifier or a regression refers to its flexibility in describing a wide range of situations. Too much flexibility is bad—it introduces high variance but low bias (e.g., 1-nearest neighbor classifiers). And too little flexibility is also bad—one has small variance but large bias (e.g., linear discriminant analysis).

3.3 Evaluating Model Performance

To assess model fit in complex, computer-intensive situations, the ideal strategy is to hold out a random portion of the data, fit a model to the rest, then use the fitted model to predict the response values from the values of the explanatory variables in the hold-out sample.

This allows a straightforward estimate of predicted mean squared error (PMSE) for regression, or predictive classification error, or some similar fit criterion. But this wastes data.

Also, we usually need to compare fits among *many* models. If the same hold-out sample is re-used, then the comparisons are not independent and (worse) the model selection process will tend to choose a model the overfits the hold-out sample, causing spurious optimism.

Cross-validation is a procedure that balances the need to use data to select a model and the need to use data to assess prediction.

Specifically, v-fold cross-validation is as follows:

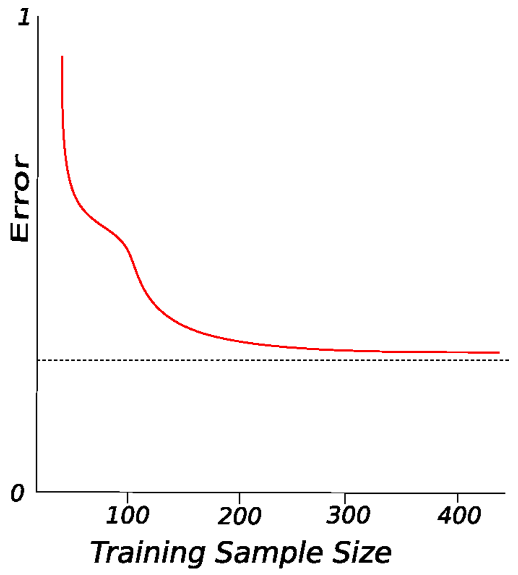
- randomly divide the sample into v portions;
- for $i = 1, \ldots, v$, hold out portion i and fit the model from the rest of the data;
- for $i = 1, \ldots, v$, use the fitted model to predict the hold-out sample;
- average the PMSE over the v different fits.

One repeats these steps (including the random division of the sample!) each time a new model is assessed.

If $v = n$, then cross-validation has low bias but possibly high variance, and computation is lengthy. If v is small, say 4, then bias can be large. A common choice is 10-fold cross-validation.

Leave-one-out cross-validation takes $v = n$, and calculates the predicted value for each observation using all of the other data. It is almost unbiased but the variance can be large, and it is computationally expensive.

Picking v is tricky. To minimize mean squared error, one must balance the variance in the estimate against the bias. One strategy is to plot the error as a function of the size of the training sample—when the curve levels off, don't increase v.



Cross-validation is not perfect—some dependency remains, and it can absorb a lot of computer time. But shortcuts exist.

For example, in a regression model where the estimated values have linear dependence on the observed values, or $\hat{\boldsymbol{y}} = \boldsymbol{H}\boldsymbol{y}$, then often

$$n^{-1}\sum_{i=1}^{n}[y_i - \hat{f}^{-i}(\boldsymbol{x}_i)]^2 = n^{-1}\sum_{i=1}^{n}\left[\frac{y_i - \hat{f}(\boldsymbol{x}_i)}{1 - h_{ii}}\right]^2$$

where $\hat{f}^{-i}(\boldsymbol{x}_i)$ is the leave-one-out cross-validation estimate of f at $\boldsymbol{x}_i$. This reduces computational time by requiring only one calculation of $\hat{f}$.

To avoid calculating h_{ii}, use Generalized Cross-Validation (GCV):

$$n^{-1}\sum_{i=1}^{n}\left[\frac{y_i - \hat{f}(\boldsymbol{x}_i)}{1 - h_{ii}}\right]^2 \approx n^{-1}\sum_{i=1}^{n}\left[\frac{y_i - \hat{f}(\boldsymbol{x}_i)}{1 - \mathbf{tr}(\boldsymbol{H})/n}\right]^2.$$

3.4 Model Complexity

A deep NN concern is the bias-variance tradeoff (or capacity control). One gets the best predictive performance if the family of models strikes the right balance between the **capacity** of the family and accuracy on the test set.

Capacity is the ability of the model to perfectly fit the training sample. If the family is so flexible that it can perform without error on the training sample, then it is fitting noise as well as the signal (i.e., overfitting).

A family with too much capacity is like a biologist with perfect memory, who when shown a dog, cannot identify its species because it has a different number of hairs than any dog previously seen. With too little capacity, he classifies everything with four legs as a dog.

Machine learning theorists have developed clever ways to set bounds on the performance capability of their procedures.

Suppose one has a training sample $\{(y_i, \boldsymbol{x}_i)\}$ and wants to predict a future Y value from measured $\boldsymbol{X}$ values. To do this, one chooses a family of models $\mathcal{F} = \{f(\boldsymbol{x}, \boldsymbol{\theta})\}$ and uses the training sample to find a value of $\boldsymbol{\theta}$ that gives "good" performance.

This structure holds in both regression or classification. Here $f(\cdot, \boldsymbol{\theta}) : \mathbb{R}^p \to \mathbb{R}$, and the family of functions (the model $\mathcal{F}$) is indexed by $\boldsymbol{\theta} \in \boldsymbol{\Theta}$.

In multiple linear regression, $\mathcal{F}$ consists of all p-dimensional hyperflats and $\boldsymbol{\theta}$ are the regression coefficients. In two-class linear discriminant analysis, $\mathcal{F}$ consists of all p-dimensional hyperflats and the $\boldsymbol{\theta}$ are the values in the Mahalanobis-distance rule.

The performance of any model in $\mathcal{F}$ is assessed through a loss function $L[f(\boldsymbol{x}, \boldsymbol{\theta}), y]$. This measures the deviation between the true value y and a predicted value $f(\boldsymbol{x}, \boldsymbol{\theta})$.

For regression, standard loss functions include:

- absolute loss: $L[f(\boldsymbol{x}, \boldsymbol{\theta}), y] = |f(\boldsymbol{x}, \boldsymbol{\theta}) - y|$
- squared error loss: $L[f(\boldsymbol{x}, \boldsymbol{\theta}), y] = [f(\boldsymbol{x}, \boldsymbol{\theta}) - y]^2$.

For binary classification with $y \in \{-1, 1\}$, a standard loss:

- $L[f(\boldsymbol{x}, \boldsymbol{\theta}], y) = I[-yf(\boldsymbol{x}, \boldsymbol{\theta}) > 0]$.

This last is an indicator function that is 1 iff the sign of y is different from the sign of $f(\boldsymbol{x}, \boldsymbol{\theta})$.

Assume that the joint distribution of future values of $(Y, \boldsymbol{X})$ is $P(y, \boldsymbol{x})$. Then the **risk**, or expected loss in a future prediction, is

$$R(\boldsymbol{\theta}) = \int_{\mathbb{R}^p \times \mathbb{R}} L[f(\boldsymbol{x}, \boldsymbol{\theta}), y]\, dP(y, \boldsymbol{x}).$$

But this cannot be calculated without knowing $P(y, \boldsymbol{x})$.

So people use the training sample to calculate the **empirical risk**:

$$R_e(\boldsymbol{\theta}) = \frac{1}{n} \sum_{i=1}^{n} L[f(\boldsymbol{x}_i, \boldsymbol{\theta}), y_i].$$

We want to find a bound on the risk such that with high probability,

$$R(\boldsymbol{\theta}) \leq R_e(\boldsymbol{\theta}) + B.$$

This shows how bad empirical risk is at estimating true risk.

In the context of two-class classification, Vapnik (1995, *Statistical Learning Theory*, Wiley) showed that with probability q,

$$R(\boldsymbol{\theta}) \leq R_e(\boldsymbol{\theta}) + \sqrt{\frac{1}{n}[v \ln n - v \ln v - \ln(q/4)]}$$

where v is a non-negative integer called the **Vapnik-Červonenkis (VC) dimension**.

Note that the bound:

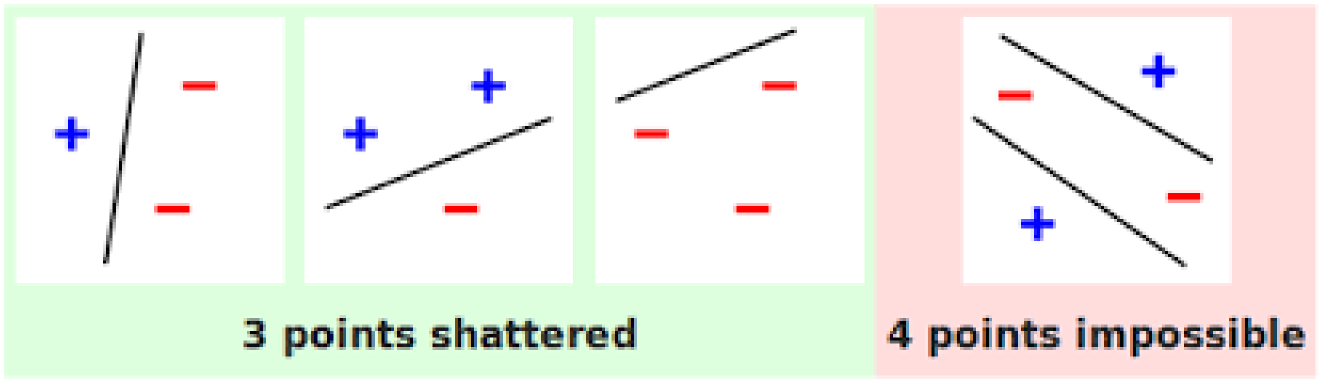
- does not depend on the $P(y, \boldsymbol{x})$;
- assumes that the training sample is a random sample from $P(y, \boldsymbol{x})$;
- is simple to compute, if v is known.

The VC dimension depends upon the class $\mathcal{F}$. To start, we consider only the two-class discrimination problem, so $f(\boldsymbol{x}, \boldsymbol{\theta}) \in \{-1, 1\}$.

A given set of n points can be labeled in 2^n possible ways. If for any such labeling, there is a member of $\mathcal{F}$ that can correctly assign those labels, then we say that the set of points is **shattered** by $\mathcal{F}$.

The VC dimension for $\mathcal{F}$ is defined as the maximum number of points (i.e., training data) that can be shattered by the elements of $\mathcal{F}$.

Note: If the VC dimension is v, then there exists at least one set of v points that can be shattered. In general, it is not true that *every* set of v points can be shattered.


3 points shattered
4 points impossible

For two-class linear discriminant analysis in $\mathbb{R}^p$, the VC dimension is $v = p + 1$. The risk bound gets large as p increases.

As the elements of $\mathcal{F}$ get more flexible, the VC dimension should increase. But the situation is complex. Consider $\mathcal{F} = \{f(x, \theta)\}$ where

$$f(x,\theta) = \begin{cases} 1 & \textbf{if } \sin(\theta x) > 0 \\ -1 & \textbf{if } \sin(\theta x) \leq 0. \end{cases}$$

Select the points $\{x_i = 10^{-i}\}$ for $i = 1, \ldots, n$. Let y_i be the label of x_i. The one can show that for any choice of labels,

$$\theta = \pi \left[1 + \sum_{i=1}^{n} \frac{1}{2}(1 - y_i)10^{i} \right]$$

gives the correct classification (Levin and Denker). So a one-parameter family can have infinite VC dimension.

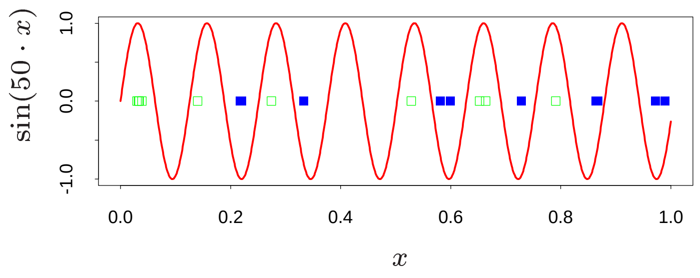


Figure 7.5: *The solid curve is the function* $\sin(50x)$ *for* $x \in [0, 1]$. *The blue (solid) and green (hollow) points illustrate how the associated indicator function* $I(\sin(\alpha x) > 0)$ *can shatter (separate) an arbitrarily large number of points by choosing an appropriately high frequency* α.

Consider the 1-nearest neighbor classifier. This $\mathcal{F}$ has infinite v, since any number of arbitrarily labeled points can be shattered. And the empirical risk is zero (unless two observations with opposite labels coincide). So in this case the VC bound gives no useful information.

Burges (1998; *Data Mining and Knowledge Discovery*, **2**, 121-167) considers a "notebook" classifier for a 50/50 mix of population types A and B. The notebook has m pages; one writes down the labels of the first m training observations, for $m < n$. With all subsequent data, predict type A.

The empirical risk (under standard classification loss) for the first m values is 0; the empirical risk for the next $n - m$ is 0.5. The true risk on future samples is 0.5. And the VC dimension is $v = m$.

The VC-dimension approach to bounding error grew up in the context of support vector machines. So it is worth noting that for nonlinear machines, the VC dimension is $v = \dim(\mathcal{H}) + 1$, where $\mathcal{H}$ is the higher-dimensional space into which the data are mapped. (Recall lecture 2.)

Thus for some kernels, the VC dimension is infinite. To see this in the case the radial basis function kernel, note that one can shatter a set of points by putting a small normal distribution on top of each, and labeling it as needed. (This is like the 1-nearest-neighbor classifier.)

Nonetheless, SVMs peform very well. A good choice of kernel enables parsimonious solutions.

Similarly to classification, one can find a VC-bound on the risk in regression:

$$R(\boldsymbol{\theta}) \leq \frac{R_e(\boldsymbol{\theta})}{(1 - c\sqrt{\delta})_+}$$

where

$$\delta = \frac{a}{n}\left[v + v \ln \frac{bn}{v} - \ln \frac{q}{4}\right].$$

As before, the probability that this bound holds is $1 - q$.

The bound was developed by Cherkassky and Mulier (1998; *Learning From Data*, 108-11). One can tune the constants a, b, c to the application. Cherkassky and Mulier recommend taking $a = b = c = 1$ for regression applications.

The bound tends to be loose.

To use the VC dimension v for regression problems, one needs to extend its definition from classes of dyadic (i.e., +1/-1) functions $\mathcal{F}$ to classes of real-valued functions.

The strategy is to take any class of real-valued functions $\{f(\boldsymbol{x},\boldsymbol{\theta})\}$ and create a set of dyadic functions

$$f_y(\boldsymbol{x},\boldsymbol{\theta}) \begin{cases} 1 & \textbf{if } f(\boldsymbol{x},\boldsymbol{\theta}) - y > 0 \\ -1 & \textbf{if } f(\boldsymbol{x},\boldsymbol{\theta}) - y \leq 0 \end{cases}$$

where y is in the range R of f. The VC-dimension v_y for $\{f_y(\boldsymbol{x},\boldsymbol{\theta})\}$ is now defined in the same way as before.

The VC dimension for the regression class $\mathcal{F}$ is $v = \sup_R\{v_y\}$.

A neural network is described by a directed acyclic graph $G(V, E)$, where V is the set of nodes and E are edges that define the architecture.

The VC dimension of a neural network is bounded as follows (Shalev-Shwartz and Ben David, p. 234-235):

- If the activation functions are the sign function then the VC dimension is at most $\mathcal{O}(|E| * \ln(|E|))$.
- If the activation functions are sigmoids, then the VC dimension is at least $\Omega(|E|^2)$ and at most $\mathcal{O}(|E|^2|V|^2)$.
- If the weights come from a finite set (e.g., quantization, or numbers represented by at most 32 bits in a computer), then, for both activation functions, the VC dimension is at most $\mathcal{O}(|E|)$.

$\Omega(f(n))$ means that for n sufficiently large, the VC dimension is at least $kf(n)$ for some constant k.