Neural Networks (and Gradient Ascent)

Frank Wood

December 3, 2009

Generalized Regression

Until now we have focused on *linear* regression techniques.

We generalized linear regression to include nonlinear functions of the inputs – we called these features.

The remaining regression model remained linear in the parameters. i.e.

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j1=}^{M} w_j \phi_j(\mathbf{x})\right)$$

where $f()$ is the identity or is invertible such that a transform of $\mathbf{t}$ can be employed.

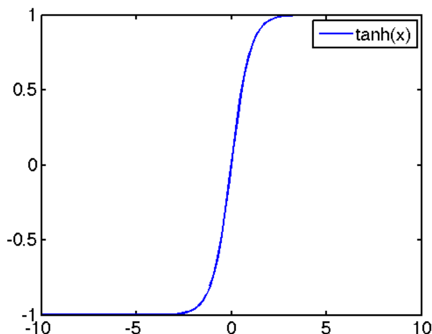
Our goal has been to learn $\mathbf{w}$. We've done this using least squares or penalized least squares in the case of MAP estimation.

Fancy $f()$'s

What if $f()$ is not invertible? Then what? Can't use transformations of $\mathbf{t}$.

Today (to start):

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



tanh regression (like logistic regression)

For pedagogical purpose assume that tanh() can't be inverted.

Or that we observe targets that are $t_n \in \{-1, +1\}$ (note – not continuous valued!)

Let's consider a regression(/classification) function

$$y(\mathbf{x}_n, \mathbf{w}) = \tanh(\mathbf{x}_n \mathbf{w})$$

where **w** is a parameter vector and **x** is a vector of inputs (potentially features). For each input **x** we have an observed output t_n which is either minus one or one.

We are interested in the general case of how to learn parameters for such models.

tanh regression (like logistic regression)

Further, we will use the error that you are familiar with, namely, the squared error. So, given a matrix of inputs $\mathbf{X} = [\mathbf{x}_1 \cdots \mathbf{x}_n]$ and a collection of output labels $\mathbf{t} = [t_1 \cdots t_n]$ we consider the following squared error function

$$E(\mathbf{X}, \mathbf{t}, \mathbf{w}) = \frac{1}{2} \sum_{n} (t_n - y(\mathbf{x}_n, \mathbf{w}))^2$$

We are interested in minimizing the error of our regressor/classifier. How do we do this?

Error minimization

If we want to minimize

$$E(\mathbf{X}, \mathbf{t}, \mathbf{w}) = \frac{1}{2} \sum_n (t_n - y(\mathbf{x}_n, \mathbf{w}))^2$$

w.r.t. $\mathbf{w}$ we should start by deriving gradients and trying to find places where the they disappear.

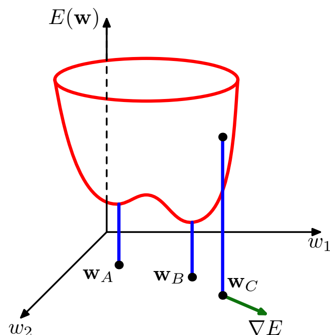


Figure taken from PRML, Bishop 2006

Error gradient w.r.t. $\mathbf{w}$

The gradient of

$$\begin{aligned}\nabla_{\mathbf{w}} E(\mathbf{X}, \mathbf{t}, \mathbf{w}) &= \frac{1}{2} \sum_n \nabla_{\mathbf{w}} (t_n - y(\mathbf{x}_n, \mathbf{w}))^2 \\ &= -\sum_n (t_n - y(\mathbf{x}_n, \mathbf{w})) \nabla_{\mathbf{w}} y(\mathbf{x}_n, \mathbf{w})\end{aligned}$$

A useful fact to know about tanh() is that

$$\frac{d \tanh(a)}{db} = (1 - \tanh(a)^2) \frac{da}{db}$$

which makes it easy to complete the last line of the gradient computation straightforwardly for the choice of $y(\mathbf{x}_n, \mathbf{w}) = \tanh(\mathbf{x}_n \mathbf{w})$, namely

$$\nabla_{\mathbf{w}} E(\mathbf{X}, \mathbf{t}, \mathbf{w}) = -\sum_n (t_n - y(\mathbf{x}_n, \mathbf{w}))(1 - \tanh(\mathbf{x}_n \mathbf{w})^2) \mathbf{x}_n$$

Solving

Unreasonable pedagogical assumptions aside, it is clear that algebraically solving

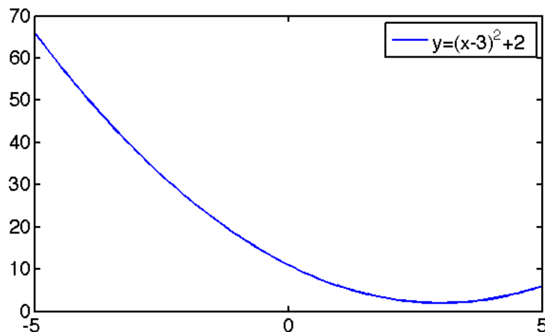
$$\begin{aligned}\nabla_{\mathbf{w}} E(\mathbf{X}, \mathbf{t}, \mathbf{w}) &= -\sum_n (t_n - y(\mathbf{x}_n, \mathbf{w}))(1 - \tanh(\mathbf{x}_n \mathbf{w})^2)\mathbf{x}_n \\ &= \mathbf{0}\end{aligned}$$

for all the entries of **w** will be troublesome if not impossible.

This is OK, however, because we don't always have to get an analytic solution that directly gives us the value of **w**. We can arrive at it's value numerically.

Calculus 101

Even simpler – consider *numerically* minimizing the function



How do you do this?

Hint, start at some value x_0, say $x_0 = -3$ and use the gradient to "walk" towards the minimum.

Calculus 101

The gradient of $y = (x-3)^2 + 2$ (or derivative w.r.t. x) is $\nabla_x y = 2(x-3)$.

Consider the sequence $x_n = x_{n-1} - \lambda \nabla_{x_{n-1}} y$

It is clear that if λ is small enough that this sequence will converge to $\lim_{n\to\infty} x_n \to 3$.

There are several important caveats worth mentioning here

- If λ (called the learning rate) is set too high this sequence might oscillate
- Worse yet, the sequence might diverge.
- If the function has multiple minima (and/or saddles) this procedure is *not* guaranteed to converge to the minimum value.

Arbitrary error gradients

This is true for any function that one would like to minimize.

For instance we are interested in minimizing prediction error $E(\mathbf{X}, \mathbf{t}, \mathbf{w})$ in our "logistic" regression/classification example where the gradient we computed is

$$\nabla_{\mathbf{w}} E(\mathbf{X}, \mathbf{t}, \mathbf{w}) = -\sum_n (t_n - y(\mathbf{x}_n, \mathbf{w}))(1 - \tanh(\mathbf{x}_n \mathbf{w})^2)\mathbf{x}_n$$

So starting at some value of the weights $\mathbf{w}_0$ we can construct and follow a sequence of guesses until convergence

$$\mathbf{w}_n = \mathbf{w}_{n-1} - \lambda \nabla_{\mathbf{w}_{n-1}} E(\mathbf{X}, \mathbf{t}, \mathbf{w})$$

Arbitrary error gradients

Convergence of a procedure like

$$\mathbf{w}_n = \mathbf{w}_{n-1} - \lambda \nabla_{\mathbf{w}_{n-1}} E(\mathbf{X}, \mathbf{t}, \mathbf{w})$$

can be assessed in multiple ways:

- The norm of the gradient grows sufficiently small
- The function value change is sufficiently small from one step to the next.
- etc.

Gradient Min(Max)imization

There are several other important points worth mentioning here and avenues for further study

- If the objective function is *convex*, such learning strategies are guaranteed to converge to the global optimum. Special techniques for convex optimization exist (e.g. Boyd and Vandenberghe, http://www.stanford.edu/~boyd/cvxbook/).
- If the objective function is not convex, multiple restarts of the learning procedure should be performed to ensure reasonable coverage of the parameter space.
- Even if the objective is not convex it might be worth the computational cost of restarting multiple times to achieve a good set of parameters.
- The "sum over observations" nature of the gradient calculation makes online learning feasible.
- More (much more) sophisticated gradient search algorithms exist, particularly ones that make use of the curvature of the underlying function.

Example - Data for tanh regression/classification

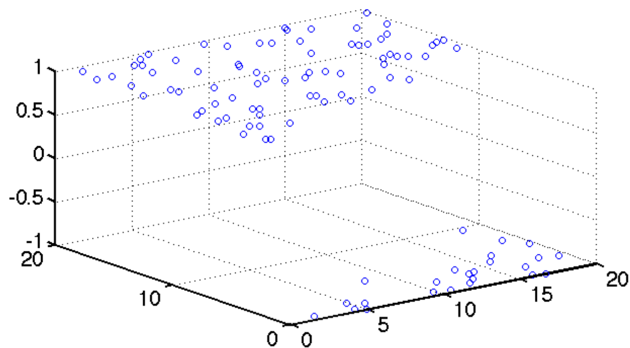


Figure: Data in $\{+1, -1\}$

"Generative model" =

n = 100;
*x = [rand(n,1) rand(n,1)]*20;*
y = x[-2;4]¿2;*
y = y+ (y==0)-1;*

Example - Result from Learning

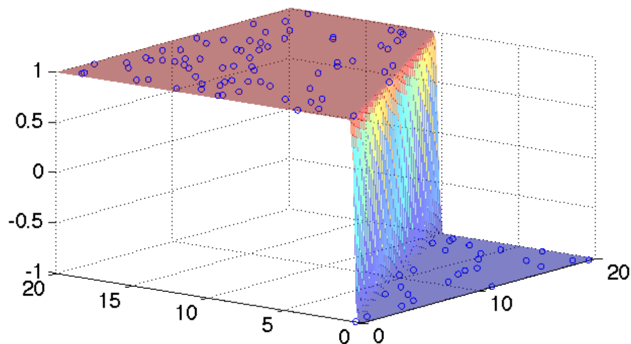


Figure: Learned regression surface.

Run logistic_regression/tanh_regression.m

Two more hints

1. Even analytic gradients are not required!
2. (Good) software exists to allow you to minimize whatever function you want to minimize (matlab: fminunc)

For both, note the following. The definition of a derivative (gradient) is given by

$$\frac{df(x)}{dx} = \lim_{\delta \to 0} \frac{f(x+\delta) - f(x)}{\delta}$$

but can be approximated quite well by a fixed size choise of δ, i.e.

$$\frac{df(x)}{dx} \approx \frac{f(x+.00000001) - f(x)}{.00000001}$$

This means that learning algorithms can be implemented on a computer using given nothing but the objective function to minimize!

Neural Networks

It is from this perspective that we will approach neural networks.

A general two layer feedforward neural network is given by :

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right)$$

Given what we have just covered, if given as set of targets $\mathbf{t} = [t_1 \cdots t_n]$ and a set of inputs $\mathbf{X} = [\mathbf{x}_1 \cdots \mathbf{x}_n]$ one should straightforwardly be able to learn $\mathbf{w}$ (the set of all weights w_{kj} and w_{ji} for all combinations kj and ji) for any choice of $\sigma()$ and $h()$.

Neural Networks

It is from this perspective that we will approach neural networks.

A general two layer feedforward neural network is given by :

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right)$$

Given what we have just covered, if given as set of targets $\mathbf{t} = [t_1 \cdots t_n]$ and a set of inputs $\mathbf{X} = [\mathbf{x}_1 \cdots \mathbf{x}_n]$ one should straightforwardly be able to learn $\mathbf{w}$ (the set of all weights w_{kj} and w_{ji} for all combinations kj and ji) for any choice of $\sigma()$ and $h()$.

Neural Networks

Neural networks arose from trying to create mathematical simplifications or representations of the kind of processing units used in our brains.

We will not consider their biological feasibility, instead we will focus on a particular class of neural network – the multi-layer perceptron, which has proven to be of great practical value in both regression and classification settings.

Neural Networks

To start – there should be list of important features and caveats

1. Neural networks are *universal approximators*, meaning that

   > *a two-layer network with linear outputs can uniformly approximate any continuous function on a compact input domain to arbitrary accuracy provided that the network has a sufficiently large number of hidden units [Bishop, PRML, 2006]*

2. *but...* How many hidden units?
3. Generally the error surface as a function of the weights is non-convex leading to a difficult and tricky optimization problem.
4. The internal mechanism by which the network represents the regression relationship is not usually examinable or testable in the way that linear regression models are. i.e. What's the meaning of a statement like, the 95% confidence interval for the ith hidden unit weight is $[.2, .4]$?

Neural network architecture

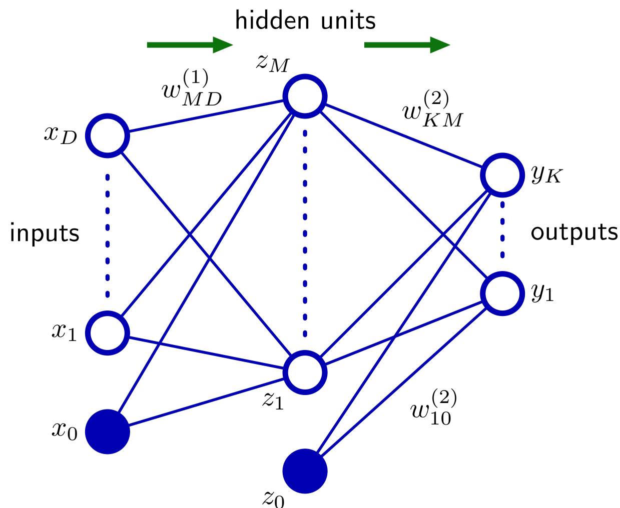


Figure taken from PRML, Bishop 2006

Neural Networks

The specific neural network we will consider is a univariate regression network where there is one output node and the output nonlinearity is set to the identity $\sigma(x) = x$ leaving only the hidden layer nonlinearity $h(a)$ which will will choose to be $h(a) = \tanh(a)$. So

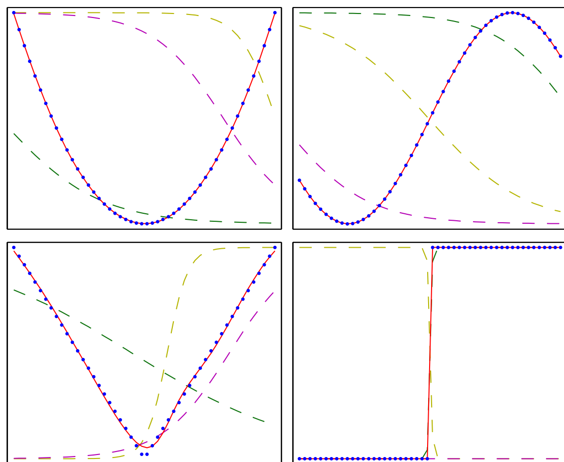
$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)\right)$$

simplifies to

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)$$

Note that the bias nodes $x_0 = 1$ and $z_0 = 1$ are included in this notation.

Representational Power



Four regression functions learned using linear/tanh neural network with three hidden units. Hidden unit activation shown in the background colors.

Figure taken from PRML, Bishop 2006

Neural Network Training

Given a set of input vectors $\{\mathbf{x}_n\}, n = 1, \ldots, N$ and a set of target vectors $\{\mathbf{t}_n\}$ (taken here to be univariate $\{t_n\}$ we wish to minimize the error function

$$E(\mathbf{w}) = \frac{1}{2} \sum_{n=1}^{N} ||\mathbf{y}(\mathbf{x}_n, \mathbf{w}) - \mathbf{t}_n||^2$$

In this example we will assume that $t \in \mathbb{R}$and that t is Gaussian distributed with mean a function of $\mathbf{x}$

$$P(t|\mathbf{x}, \mathbf{w}, \beta) = \mathcal{N}(t|y(\mathbf{x}_n, \mathbf{w}), \beta^{-1})$$

which means that the targets are jointly distributed according to

$$P(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} P(t_n|\mathbf{x}_n, \mathbf{w}, \beta)$$

Neural Network Training and Prediction

If we take the negative logarithm of the error function

$$P(\mathbf{t}|\mathbf{X}, \mathbf{w}, \beta) = \prod_{n=1}^{N} P(t_n|\mathbf{x}_n, \mathbf{w}, \beta)$$

we arrive at

$$\frac{\beta}{2}\sum_{n=1}^{N}\{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2 - \frac{N}{2}\ln\beta + \frac{N}{2}\ln(2\pi)$$

which we can minimize by first minimizing w.r.t. to $\mathbf{w}$ and then β.

Given a trained value of β_{ML} and $\mathbf{w}_{ML}$ prediction is straightforward.

Neural Network Training, Gradient Ascent

We therefore are interested in minimizing (in the case of continuous valued, univariate, neural network regression)

$$E(\mathbf{w}) = \sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2$$

where

$$y(\mathbf{x}, \mathbf{w}) = \sum_{j=0}^{M} w_{kj}^{(2)} h\left(\sum_{i=0}^{D} w_{ji}^{(1)} x_i\right)$$

which we can perform numerically if we have gradient information

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)})$$

where η is a learning rate and $\nabla E(\mathbf{w}^{(\tau)})$ is the gradient of the error function.

Back Propagation

While numeric gradient computation can be used to estimate the gradient and thereby adjust the weights of the neural net, doing so is not very efficient.

A more efficient, if not slightly more confusing method of computing the gradient, is to use *backpropagation*.

Back propagation is a fancy term for a dynamic programming-like way of computing the gradient by running computations backwards on the network.

Back Propagation

To perform back propagation we need to identify several intermediate variables, the first of which is

$$a_j = \sum_i w_{ji} z_i$$

where a_j is a weighted sum of the inputs to a particular unit (hidden or otherwise). The activiation z_j of a unit is given by

$$z_j = h(a_j)$$

where in our example $h(a) = \tanh(a)$

Here j could be an output.

Back Propagation

What we are interested in computing efficiently is $\frac{dE}{dw_{ji}}$ where

$$E(\mathbf{w}) = \sum_{n=1}^{N} \{y(\mathbf{x}_n, \mathbf{w}) - t_n\}^2$$

we will focus on the individual contribution from a single training input/output pair $\frac{dE_n}{dw_{ji}}$ realizing that the final gradient is the sum of all of the individual gradients.

Note: *Stochatistic gradient ascent approximates the gradient using a single (or small group) of points at a time.*

Back Propagation : Reuse of computation

Our goal is to reuse computation as much as possible. We will do this by constructing a back-ward chaining set of partial derivatives that use computation closer to the output nodes in the calculation of the gradients of the error w.r.t. to weights closer to the input nodes.

To start, note that E_n depends on the weights W_{ji} only through the input a_j to unit j. For this reason we can applye the chain rule to give

$$\frac{dE_n}{dw_{ji}} = \frac{dE_n}{da_j} \frac{da_j}{dw_{ji}}$$

We will denote $\frac{dE_n}{da_j} = \delta_j$. The δ_j's are going to be the quantities we use for dynamic programming.

Back Propagation : Reuse of computation

If we remember that

$$a_j = \sum_i w_{ji} z_i$$

and our new notation $\frac{dE_n}{da_j} = \delta_j$.
We can re-write

$$\frac{dE_n}{dw_{ji}} = \frac{dE_n}{da_j} \frac{da_j}{dw_{ji}}$$

as

$$\frac{dE_n}{dw_{ji}} = \delta_j z_i$$

This is almost it!

Back Propagation : Reuse of computation

For the output layer we have

$$\frac{dE_n}{da_k} = \frac{d}{da_k}\frac{1}{2}(a_k - t)^2 = y_k - t_k$$

From this we can compute the gradient with respect to all of the weights leading to the output layer simply using

$$\frac{dE_n}{dw_{ki}} = \delta_k z_i$$

where i ranges over the hidden layer closest to the output layer and z_i are the activations of that layer.

What remains is to figure out how to use the precomputed δ's to compute the gradients at all remaining hidden layers back to the input nodes. For this we need the chain rule from calculus again.

Back Propagation : Reuse of computation

We want a way of computing δ_j, the error term for an arbitrary hidden unit as a function of the weights and the error terms already computed closer to the output node(s). The definition of δ_j is $\delta_j = \frac{dE_n}{da_j}$

When node j is connected to nodes $k, k = 1, \ldots, K$ the following is true: that the error is a function of the activations at all k nodes and that the activations at each of these nodes is a function of the activation at node j. That means the following is true

$$\frac{dE_n}{da_j} = \sum_k \frac{dE_n}{da_k} \frac{da_k}{da_j}$$

but we have computed $\delta_k = \frac{dE_n}{da_k}$ already for nodes closer to the output already!

Back Propagation : Reuse of computation

To summarize, we know

$$\frac{dE_n}{da_j} = \sum_k \delta_k \frac{da_k}{da_j}$$

We also know $a_k = \sum_j w_{kj} z_j$ and $z_j = h(a_j)$.
This means that

$$\delta_j = \frac{dE_n}{da_j} = \sum_k \delta_k h'(a_j) w_{kj} = h'(a_j) \sum_k \delta_k w_{kj}$$

This means that we can compute the δ's backwards, using only information "local" to each unit.

Further we know that $\frac{dE_n}{w_{ji}} = \delta_j z_i$ which is the "error" at the output side times the activation on the input side. Since the activations are computed on the forward pass and the errors are computed on the backwards pass we are done!

Back Propagation : Full procedure

Error Backpropagation

Repeat for all input/output pairs:

1. Propagate activations forward through the network for an input $\mathbf{x}_n$
2. Compute the δ's for all the units starting at the output layer and proceeding backwards through the network.
3. Compute the contribution to the gradient for that single input (and sum into global gradient computation).

Conclusion

Neural networks are a powerful tool for regression analysis.

Neural networks are not without (significant) downsides. They lack interpretability, they can be difficult to learn, and the model selection issues that arise in any regression problem don't go away.

Further treatment of neural networks include different activation functions, multivalued outputs, classification, and Bayesian neural networks.

Simple trailing question: what would MAP estimation of a neural network look like (with standard weight decay regularization)?