Lecture Notes: 600.475 Neural Networks

Matt Gormley

7 October 2014

Contents

1 Backpropagation

The backpropagation algorithm is a general method for computing the gradient of a neural network. Here we generalize the concept of a neural network to include any computational circuit. Applying the backpropagation algorithm on these circuits amounts to repeated application of the chain rule.

This general algorithm goes under many other names: automatic differentiation (AD) in the reverse mode, analytic differentiation, module-based AD, autodiff, etc.

Below we define a forward pass which computes the output bottom-up, and a backward pass which computes the derivatives of all intermediate quantities top-down.

1.1 Defining Computational Circuits

The graphical representation of a neural network leaves much to be desired. What non-linear function is used? What does it mean to connect one variable to another? How can we make this more precise?

Computational circuits provide a clearer visual representation of the computation and help provide a clearer intuition for how to apply backpropagation.

1.2 Chain Rule

At the core of the backpropagation algorithm is the chain rule. The chain rule in its familiar form allows us to differentiate a function f defined as a the composition of two functions g and h such that $f = (g \circ h)$.

If the inputs and outputs of f, g, and h are all scalars, then we obtain the familiar form of the chain rule:

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx} \tag{1}$$

or equivalently,

$$f'(x) = (g \circ h)'(x) \tag{2}$$
$$= g'(h(x))h'(x) \tag{3}$$

Suppose instead the inputs and outputs of the functions are vector-valued variables, $f : \mathbb{R}^K \to \mathbb{R}^I$. Let $\boldsymbol{y} = g(\boldsymbol{u})$ and $\boldsymbol{u} = h(\boldsymbol{x})$ where $\boldsymbol{x} = \{x_1, x_2, \ldots, x_K\}$, $\boldsymbol{y} = \{y_1, y_2, \ldots, y_I\}$, and $\boldsymbol{u} = \{u_1, u_2, \ldots, u_J\}$. Then the chain rule must sum over all the intermediate quantities.

$$\frac{dy_i}{dx_k} = \sum_{j=1}^{J} \frac{dy_i}{du_j}\frac{du_j}{dx_k}, \quad \forall i, k \tag{4}$$

1.3 Small Example

This section demonstrates automatic differentiation as applied to a simple example of only one input variable.

The goal is to compute $J = \cos(\sin(x^2) + 3x^2)$ on the forward pass and the derivative $\frac{dJ}{dx}$ on the backward pass.

Forward	Backward			
$J = cos(u)$	$\frac{dJ}{du}$ += $-sin(u)$			
$u = u_1 + u_2$	$\frac{dJ}{du_1}$ += $\frac{dJ}{du}\frac{du}{du_1}$,	$\frac{du}{du_1} = 1$	$\frac{dJ}{du_2}$ += $\frac{dJ}{du}\frac{du}{du_2}$,	$\frac{du}{du_2} = 1$
$u_1 = sin(t)$	$\frac{dJ}{dt}$ += $\frac{dJ}{du_1}\frac{du_1}{dt}$,	$\frac{du_1}{dt} = \cos(t)$		
$u_2 = 3t$	$\frac{dJ}{dt}$ += $\frac{dJ}{du_2}\frac{du_2}{dt}$,	$\frac{du_2}{dt} = 3$		
$t = x^2$	$\frac{dJ}{dx}$ += $\frac{dJ}{dt}\frac{dt}{dx}$,	$\frac{dt}{dx} = 2x$		

2 Neural Network Training

Choose each of the following:

- **Loss Function:** $J = \ell(\hat{\boldsymbol{y}}, \boldsymbol{y}^*) \in \mathbb{R}$
- **Neural Network:** $\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$

Training Find parameters $\boldsymbol{\theta}$ that minimize the objective function over the entire training set $\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N$: Define the loss / error as function of the parameters for each training instance i.

$$\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^{N} E_i(\boldsymbol{\theta})$$
$$\text{where } E_i(\boldsymbol{\theta}) = \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i^*)$$

SGD Stochastic gradient descent minimizes this objective iteratively:

1. Choose a starting point $\boldsymbol{\theta}$.
2. While not converged:
   - Choose a step size $\eta_t > 0$.
   - Sample a training instance i.
   - Take a small step following the gradient down.

$$\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla E_i(\boldsymbol{\theta}) \tag{5}$$

What is the gradient? It's just a vector of derivatives. For shorthand we can define $J = E_i(\boldsymbol{\theta})$, then the gradient is $\nabla E_i(\boldsymbol{\theta}) = [\frac{dJ}{d\theta_1}, \frac{dJ}{d\theta_2}, \ldots, \frac{dJ}{d\theta_K}]$.

This means that the SGD update can either be written as the vector update in Eq. (5), or as an update of each model parameter as in Eq. (6) below

$$\theta_j^{(t+1)} = \theta_j^{(t)} - \eta_t \frac{dJ}{d\theta_j} \tag{6}$$

Prediction Given a new instance $\boldsymbol{x}$, predict the output $\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x})$.

2.1 Loss Functions

Some common loss functions for the case where $y \in \{0, 1\}$.

	Forward	Backward
Regression	$J = \frac{1}{2}(y - y^*)^2$	$\frac{dJ}{dy} = y - y^*$
Cross Entropy	$J = y^* \log(y) + (1 - y^*) \log(1 - y)$	$\frac{dJ}{dy} = y^* \frac{1}{y} + (1 - y^*) \frac{1}{y - 1}$

2.2 Binary Logistic Regression

Binary logistic regression can be interpreted as a computational circuit. To compute the derivative of some loss function (below we use regression) with respect to the model parameters, we can apply backpropagation.

Note that the output y below is the probability that the output label takes on the value 1.

The forward pass computes $J = \frac{1}{2}\left(\left(\frac{1}{1+\exp(\sum_{j=0}^{D}\theta_j x_j)}\right) - y^*\right)^2$. The backward pass computes $\frac{dJ}{d\theta_j} \forall j$.

Forward	Backward	
$J = \frac{1}{2}(y - y^*)^2$	$\frac{dJ}{dy} = y - y^*$	
$y = \frac{1}{1+\exp(a)}$	$\frac{dJ}{da} = \frac{dJ}{dy}\frac{dy}{da}, \frac{dy}{da} = \frac{\exp(a)}{(\exp(a)+1)^2}$	
$a = \sum_{j=0}^{D}\theta_j x_j$	$\frac{dJ}{d\theta_j} = \frac{dJ}{da}\frac{da}{d\theta_j}, \frac{da}{d\theta_j} = x_j$	$\frac{dJ}{dx_j} = \frac{dJ}{da}\frac{da}{dx_j}, \frac{da}{dx_j} = \theta_j$

2.3 2-Layer Neural Network

Backpropagation for a 2-layer neural network looks very similar to the logistic regression example above. We have added a hidden layer z corresponding to the latent features of the neural network.

Note that our model parameters θ are defined as the concatentation of the vector β (parameters for the output layer) with the vectorized matrix α (parameters for the hidden layer).

Forward	Backward	
$J = \frac{1}{2}(y - y^*)^2$	$\frac{dJ}{dy} = y - y^*$	
$y = \frac{1}{1+\exp(b)}$	$\frac{dJ}{db} = \frac{dJ}{dy}\frac{dy}{db}, \frac{dy}{db} = \frac{\exp(b)}{(\exp(b)+1)^2}$	
$b = \sum_{j=0}^{D}\beta_j z_j$	$\frac{dJ}{d\beta_j} = \frac{dJ}{db}\frac{db}{d\beta_j}, \frac{db}{d\beta_j} = z_j$	$\frac{dJ}{dz_j} = \frac{dJ}{db}\frac{db}{dz_j}, \frac{db}{dz_j} = \beta_j$
$z_j = \frac{1}{1+\exp(a_j)}$	$\frac{dJ}{da_j} = \frac{dJ}{dz_j}\frac{dz_j}{da_j}, \frac{dz_j}{da_j} = \frac{\exp(a_j)}{(\exp(a_j)+1)^2}$	
$a_j = \sum_{i=0}^{M}\alpha_{ji} x_i$	$\frac{dJ}{d\alpha_{ji}} = \frac{dJ}{da_j}\frac{da_j}{d\alpha_{ji}}, \frac{da_j}{d\alpha_{ji}} = x_i$	$\frac{dJ}{dx_i} = \frac{dJ}{da_j}\frac{da_j}{dx_i}, \frac{da_j}{dx_i} = \sum_{j=0}^{D}\alpha_{ji}$

2.4 Numerical Differentiation

While this document is focused on reverse-mode automatic differentiation, numerical differentiation provides a convenient method for testing gradients computed by autodiff. Unfortunately, in practice, it suffers from issues of floating point precision. Therefore, it is typically only appropriate to use this on small examples with an appropriately chosen epsilon close to machine epsilon.

$$\frac{\partial}{\partial \theta_i} E(\boldsymbol{\theta}) \approx \frac{(E(\boldsymbol{\theta} + \epsilon \cdot \boldsymbol{d}) - E(\boldsymbol{\theta} - \epsilon \cdot \boldsymbol{d}))}{2\epsilon \cdot d_i} \tag{7}$$

3 For Presentation

Recipe for ML

$$\begin{aligned}
\{\boldsymbol{x}_i, \boldsymbol{y}_i\}_{i=1}^N \\
\hat{\boldsymbol{y}} = f_{\boldsymbol{\theta}}(\boldsymbol{x}_i) \\
\ell(\hat{\boldsymbol{y}}, \boldsymbol{y}_i) \in \mathbb{R} \\
\boldsymbol{\theta}^* = \arg\min_{\boldsymbol{\theta}} \sum_{i=1}^N \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i) \\
\boldsymbol{\theta}^{(t+1)} = \boldsymbol{\theta}^{(t)} - \eta_t \nabla \ell(f_{\boldsymbol{\theta}}(\boldsymbol{x}_i), \boldsymbol{y}_i)
\end{aligned}$$

Linear, Logistic, NNs

$$\begin{aligned}
f_{\boldsymbol{\theta}}(\boldsymbol{x}) &= y \\
y &= h(\boldsymbol{\theta} \cdot \boldsymbol{x}) \\
&\text{where } h(a) = a \\
f_{\boldsymbol{\theta}}(\boldsymbol{x}) &= y = h(\boldsymbol{\theta} \cdot \boldsymbol{x}) \\
&\text{where } h(a) = \frac{1}{1 + \exp(a)}
\end{aligned}$$

$$\begin{aligned}
y = f_{\boldsymbol{\theta}}(\boldsymbol{x}) &= h(\boldsymbol{\theta} \cdot \boldsymbol{x}) \\
&\text{where } h(a) = \frac{1}{1 + \exp(a)}
\end{aligned}$$

CS 475 Machine Learning: Lecture 6
Information Theory

1 Information Theory

Information theory is the study of the transmission of bits across a noisy channel.
The currency of information theory is "bits"
How many bits do I need to encode information?
The model is a channel with a sender and receiver. I want to send you information. How many bits do I need to do it? How expensive is information?

I have a coin {Heads, Tails}. I want to send you the result of the coin flip. On average, how many bits do I need? (1 bit)

- Heads - $\langle 0 \rangle$
- Tails - $\langle 1 \rangle$

Of course, not everything fits into 1 bit.
Horse race with 4 horses. How many bits? (2 bits)

- Horse A - $\langle 0, 0 \rangle$
- Horse B - $\langle 0, 1 \rangle$
- Horse C - $\langle 1, 0 \rangle$
- Horse D - $\langle 1, 1 \rangle$

Let's say the sender and receiver know extra information.
Distribution over each horse winning the race.

- Horse A - $\frac{1}{2}$
- Horse B - $\frac{1}{4}$
- Horse C - $\frac{1}{8}$
- Horse D - $\frac{1}{8}$

Can we do better than 2 bits?

- Horse A - $\langle 0 \rangle$
- Horse B - $\langle 1, 0 \rangle$
- Horse C - $\langle 1, 1, 0 \rangle$
- Horse D - $\langle 1, 1, 1 \rangle$

Notice that I now have up to 3 bits, but only for unlikely events.
How many on average?
$.5 \times 1 + .25 \times 2 + .125 \times 3 + .125 \times 3 = 1.75$ bits

1.1 Entropy

In information theory, entropy is the uncertainty associated with a random value.
We can ask how uncertain are we with the random value (horse race) we are receiving.
The expected value (number of bits) in the message.

The entropy of a discrete random variable X is:

$$H(X) = E(I(X))$$

E is the expected value function
$I(X)$ is the information content of the message/random variable X
We can write this out as:

$$H(X) = \sum_{i=1}^{n} p(x_i) I(x_i) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i)$$

First part- weigh each event's information by the probability that it occurs
Second part- the amount of bits needed to store the information.
Consider the horse race. For an event that occurs $\frac{1}{2}$ the time we need:

$$-\log_2 p(x_i) = -\log_2 \frac{1}{2} = 1\text{bit}$$

For an event that occurs $\frac{1}{4}$ the time we need:

$$-\log_2 p(x_i) = -\log_2 \frac{1}{4} = 2\text{bits}$$

So to know how much information we need for the horse race, use the entropy of the message:

$$H(X) = -\sum_{i=1}^{n} p(x_i) \log_b p(x_i) = -\frac{1}{2}\log_2 \frac{1}{2} - \frac{1}{4}\log_2 \frac{1}{4} - \frac{1}{8}\log_2 \frac{1}{8} - \frac{1}{8}\log_2 \frac{1}{8} = 1.75\text{bits}$$

1.2 Notes on Entropy

High entropy- the distribution is uniform. We can't predict which events will happen. More bits needed.
Low entropy- the distribution is peaked. We can predict which events will happen. Less bits needed.

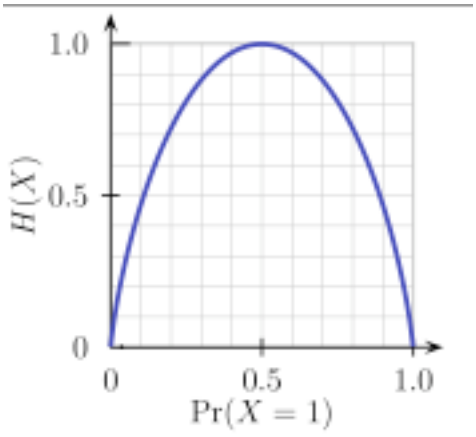

Figure shows the entropy for a coin. If the coin has equal probability of heads vs. tails,

then high entropy (full bit needed). Otherwise, less bits.

1.3 Conditional Entropy

What if we both already know some information. How many more bits are needed?
Example, you knew that horse A or B won, but not sure which. Do I still need 1.75 bits? Obviously not.
Define $H(Y|X = x)$- the number of bits needed to send Y given that we both know $X = x$.
Its the same as entropy but for only the cases when $X = x$.
The full expected condition entropy is $H(Y|X)$ where we average over all the values that X can take in $H(Y|X = x)$.

$$\begin{aligned} H(Y|X) &= \sum_{x\in X} p(x)H(Y|X=x) && (1)\\ &= -\sum_{x\in X} p(x)\sum_{y\in Y} p(y|x)\log p(y|x) && (2)\\ &= -\sum_{x\in X}\sum_{y\in Y} p(y,x)\log p(y|x) && (3)\\ &= -\sum_{x\in X, y\in Y} p(y,x)\log p(y|x) && (4)\\ &= -\sum_{x\in X, y\in Y} p(y,x)\log \frac{p(y,x)}{p(x)} && (5)\\ & && (6) \end{aligned}$$

1.4 Information Gain

Now that we can 1) quantify how much information is in a message and 2) how much that reduces when both sides know information:
We can talk about information savings.
I want to send Y with as few bits as possible. How many bits could I save if we both knew X?
In terms of horse race: I want to say that horse A won the race, how many bits would I save if we both knew it was horse A or B?
Information gain: how much information have we gained if you knew X?

$$IG(Y|X) = H(Y) - H(Y|X)$$

Intuitively, X has a high information gain with respect to Y if, knowing X , it takes many fewer bits to transmit Y.

CS 475 Machine Learning: Lecture 10
Notes for SVMs*

Prof. Mark Dredze

1 SVM Derivation

Let's start by writing the quadratic program

$$\begin{aligned} \arg\min_{\mathbf{w}} \quad & \tfrac{1}{2}||\mathbf{w}||^2 \\ s.t. \quad & y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1 = 0, \forall i \end{aligned} \tag{1}$$

The purpose of the $\frac{1}{2}$ will be apparent momentarily. We will ignore the bias term since it is necessary for some of the derivation.

This form of the objective can be solved directly using quadratic programming, which minimizes (or maximizes) a quadratic function with linear constraints. This is a convex function so we will find a single global optimum. There is a single hyperplane that satisfies all of the constraints. The single result is from the normalization of $\mathbf{w}$.

1.1 Dual

In these types of optimization problems, we call this the objective the primary problem. However, we can also define a dual problem. The dual problem and the primary problem have the same solution. If the primary problem minimizes an objective, then we can rewrite it as a problem that maximizes the objective.

Let's rewrite this objective using Lagrange multipliers. We introduce a multiplier for each constraint: α_i such that $\alpha_i \geq 0$. We have n such variables, so you can guess these will become the dual variables.

Using these multipliers we write the Lagrange function:

$$L(\mathbf{w}, b, \alpha) = \frac{1}{2}||\mathbf{w}||^2 - \sum_{i=1}^{N} \alpha_i\{y_i(\mathbf{w}\cdot\mathbf{x}_i + b) - 1\} \tag{2}$$

Notice the $-$ in front of the multipliers since this is minimization.
Sanity check: This equation is minimized when the norm of $\mathbf{w}$ is 0 and all margins are 1.

Let's start by taking the derivative of L with respect to b since that is easy.

$$\frac{\partial L}{\partial b} = -\sum_{i=1}^{N} \alpha_i y_i \tag{3}$$

Setting that equal to 0 gives the optimal solution for b, which is just a constraint on the values of α.

$$0 = \sum_{i=1}^{N} \alpha_i y_i \tag{4}$$

*You will notice that the notation I use differs from the Bishop book. I have chosen this alternate notation to remain consistent with the majority of the literature on machine learning. This means that when you read papers in the literature you will be familiar with their notation.

Good-bye b ☺

Let's take the derivative of L with respect to $\mathbf{w}$ for one of the positions in $\mathbf{w}$.

$$\frac{\partial L}{\partial w_j} = w_j - \sum_{i=1}^{N} \alpha_i y_i x_{i,j} \tag{5}$$

We see that the derivative for a value in $\mathbf{w}$ depends on a combination of values from $\mathbf{x}$ weighted by α. The derivative for the full vector $\mathbf{w}$ is then just:

$$\frac{\partial L}{\partial \mathbf{w}} = \mathbf{w} - \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i \tag{6}$$

Setting the derivative equal to 0 and solving for $\mathbf{w}$:

$$\mathbf{w} = \sum_{i=1}^{N} \alpha_i y_i \mathbf{x}_i \tag{7}$$

Here comes the magic.

We now can use these as constraints and obtain a new objective. This takes some slick algebra, which we'll skip.

$$\tilde{L}(\alpha) = \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} \alpha_i \alpha_j y_i y_j \mathbf{x}_i \mathbf{x}_j \tag{8}$$

with constraints

$$\alpha_i \geq 0 \quad \forall i$$
$$\sum_{i=1}^{n} \alpha_i y_i = 0$$

We can then maximize this dual objective, which minimizes our primal objective by finding the values for α that satisfy the constraints.

Sanity check: when is this maximized? Let's go back to the slides and look at how predictions are made.

2 Dual Perceptron

Let's look at $\mathbf{w}$ in the perceptron. If we've seen lots of examples of a perceptron then many of them have contributed to $\mathbf{w}$.

What is $\mathbf{w}$ after no examples? $\mathbf{w} = 0$

What is $\mathbf{w}$ after the ith examples?

$$\mathbf{w}^{i+1} = \mathbf{w}^{i} + y_i \mathbf{x}_i \tag{9}$$

What about $\mathbf{w}$ after $i-1$ examples?

$$\mathbf{w}^{i} = \mathbf{w}^{i-1} + y_{i-1} \mathbf{x}_{i-1} \tag{10}$$

So we can write $\mathbf{w}^{i+1}$ as:

$$\mathbf{w}^{i+1} = \mathbf{w}^{i-1} + y_{i-1} \mathbf{x}_{i-1} + y_i \mathbf{x}_i \tag{11}$$

If we continue until $i = 0$ we get

$$\mathbf{w}^{i+1} = \mathbf{0} + \sum_{i=1}^{n} y_i \mathbf{x}_i \tag{12}$$

Of course, we only make an update if we make a mistake. Let's write $\mathcal{M}$ as the set of examples for which we made a mistake.

$$\mathbf{w}^{i+1} = \mathbf{0} + \sum_{i \in \mathcal{M}} y_i \mathbf{x}_i \tag{13}$$

So we see that $\mathbf{w}$ is a linear combination of the inputs $\mathbf{x}_i$. In this case, α_i is just an indicator if we made a mistake $\alpha_i = 1$ or if we got the example correct $\alpha_i = 0$.

What is the prediction rule for the perceptron?

$$\hat{y} = \text{sign}(\mathbf{w} \cdot \mathbf{x}) \tag{14}$$

But now we can express $\mathbf{w}$ in terms of $\mathbf{x}$ and our indicator α.

$$\hat{y} = \text{sign}(\{\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i\} \cdot \mathbf{x}) \tag{15}$$

and moving $\mathbf{x}$ inside the sum we see.

$$\hat{y} = \text{sign}(\sum_{i=1}^{n} \alpha_i y_i \mathbf{x}_i \mathbf{x}) \tag{16}$$

This looks very similar to an SVM. The difference is really in how we set α_i. For the perceptron, we set them at each point based a local decision. In SVM, we set them globally by looking at all of the points.

Notice that since we no longer have $\mathbf{w}$, this is a *dual perceptron*. We have n variables α to solve instead of m variables for w.

CS 475 Machine Learning: Lecture 3
Notes for Regression

Prof. Mark Dredze

1 Maximum Likelihood for Gaussians

Let's begin by looking at Maximum Likelihood for Gaussians. We are given a dataset that contains points that we believe have been sampled from a Gaussian distribution. Our set of points is given by $\mathbf{X} = \{\mathbf{x}_i\}$, where $\mathbf{x}_i \in \mathcal{R}^M$. We would like to estimate the parameters of this Gaussian: μ and $\mathbf{\Sigma}$. We want to measure the likelihood that these points (our data $\mathcal{D}$) were generated by the parameters μ and $\mathbf{\Sigma}$.

Using an M-dimensional Gaussian the likelihood of a single instance is

$$p(\mathbf{x}|\mu, \mathbf{\Sigma}) = \frac{1}{\sqrt{(2\pi)^M}} \frac{1}{\sqrt{|\mathbf{\Sigma}|}} \exp\left\{-\frac{1}{2}(\mathbf{x}-\mu)^T\mathbf{\Sigma}^{-1}(\mathbf{x}-\mu)\right\}$$

where $|\mathbf{\Sigma}|$ is the determinant of $\mathbf{\Sigma}$. The likelihood of our data is the product of the likelihood of each individual example.

$$p(\mathcal{D}|\mu, \mathbf{\Sigma}) = \prod_{i=1}^{N} \frac{1}{\sqrt{(2\pi)^M}} \frac{1}{\sqrt{|\mathbf{\Sigma}|}} \exp\left\{-\frac{1}{2}(\mathbf{x}_i-\mu)^T\mathbf{\Sigma}^{-1}(\mathbf{x}_i-\mu)\right\}$$

To simplify the function, we take the log of the likelihood.

$$\mathcal{L} = \log p(\mathcal{D}|\mu, \mathbf{\Sigma}) = -\frac{NM}{2}\log(2\pi) - \frac{N}{2}\log|\mathbf{\Sigma}| - \frac{1}{2}\sum_{i=1}^{N}(\mathbf{x}_i-\mu)^T\mathbf{\Sigma}^{-1}(\mathbf{x}_i-\mu)$$

We now have a function of variables $\mathbf{\Sigma}$ and μ. We can find the maximum of this function, ie. the point of highest likelihood, by taking the derivative of the function with respect to each of the variables. We then set the derivatives to 0 and solve for the variable.

Since only the last term depends on μ so we can get:

$$\frac{\partial \mathcal{L}}{\partial \mu} = \sum_{i=1}^{N} \mathbf{\Sigma}^{-1}(\mathbf{x}_i - \mu) = 0$$

Moving the μ terms to the right hand side and multiplying by $\mathbf{\Sigma}$, we get a solution for μ

$$\mu_{ML} = \frac{1}{N}\sum_{i=1}^{N}\mathbf{x}_i$$

This is the maximum likelihood solution for the variable μ.

The same approach for $\mathbf{\Sigma}$ is much more involved. It is illustrative to try the 1-dimensional case, where $\mathbf{\Sigma}$ is actually a 1×1 matrix (ie. a scalar) which we indicate by σ. Using the 1-dimesional Gaussian we get:

$$\sigma^2_{ML} = \frac{1}{N}\sum_{i=1}^{N}(\mathbf{x}_i - \mu_{ML})(\mathbf{x}_i - \mu_{ML})^T$$

What are the expectations of these parameters, meaning what values do we expect to get for these parameters?

$$\mathbb{E}[\mu_{ML}] = \mu$$
$$\mathbb{E}[\mathbf{\Sigma}_{ML}] = \frac{N-1}{N}\mathbf{\Sigma}$$

Notice that the expectation of μ is correct, it is equal to the true expectation of the Gaussian. However, the expectation of $\mathbf{\Sigma}$ is biased, it underestimates the true parameter by $\frac{N-1}{N}$. Maximum likelihood estimation thinks that there is less variance than truly present in the data, meaning that it over-fits the observed data. Over-fitting means that we have high variance (in the bias vs. variance sense): we are too trusting of our observed data. In contrast, we believe there is lower variance (not in the bias vs. variance sense) for the Gaussian.

When $N \to \infty$, we have infinite data, then this isn't a problem. However, with limited data, this becomes an issue.

If the estimator for μ is unbiased, why is estimator for $\mathbf{\Sigma}$ biased? The estimator for the variance is based on the sample mean ($\mathbf{X}$), and not the true mean of the distribution. That means that sometimes the mean is too low, and sometimes the mean is too high, when compared to the actual true mean of the distribution. When we compute μ_{ML} these biases cancel each other out (the "too high" estimates balance out the "too low" estimates.) However, variance squares the sample mean, which means that even when its under-estimates (misses the true mean by a negative number), these numbers become positive (in the square.) When all our estimates are off by a positive number, the positive and negative (relative) estimates no longer are canceled, so we end up over-estimating the distribution's mean.[1]

Note that we could modify the maximum likelihood estimator for variance to overcome this problem:

$$\mathbf{\Sigma} = \frac{1}{N-1}\sum_{i=1}^{N}(\mathbf{x}_i - \mu_{ML})(\mathbf{x}_i - \mu_{ML})^T$$

2 Maximum Likelihood for Least Squares Regression

In regression, we are given a set of examples $\mathbf{x}_i$, where $\mathbf{x}_i \in \mathcal{R}^M$ and $y_i \in mathcalR$. We assume that these examples are generated by a linear function $f_{\mathbf{w}}(\mathbf{x})$ and are then permuted by Gaussian noise.

Let's start by writing the probability of the data $\mathcal{D}$. We replace the mean of the Gaussian with our prediction function $\mathbf{w}^T \cdot \mathbf{x}_i$. Recall that $\mathbf{w}$ is a $n \times 1$ vector and x is a $n \times 1$ vector, so the dot production of our prediction is a real number. Since the parameters $\mathbf{w}$ are the weights given to each feature, we call $\mathbf{w}$ the weight vector.

We replace y with our prediction function $\mathbf{w}^T \cdot \mathbf{x}_i$ and write the log likelihood of all of our regression data:

$$\log p(\mathcal{D}|\mathbf{w}, \mathbf{\Sigma}) = \log \prod_{i=1}^{N} \mathcal{N}(\mathbf{w}^T \cdot \mathbf{x}_i, \mathbf{\Sigma})$$

We can distribute the log, which breaks the products into sums. For simplicity, we switch to an alternate form of the Gaussian, which replaces $\mathbf{\Sigma}$ with β^{-1}, where β is the precision

[1] See a nice explanation here: `http://stats.stackexchange.com/questions/136673/how-to-understand-that-mle-of-variance-is-biased-in-a-gaussian-distribution`.

(inverse variance) of the Gaussian:

$$\log p(\mathcal{D}|\mathbf{w}, \mathbf{\Sigma}) = \frac{N}{2} \log \beta^{-1} - \frac{N}{2} \log(2\pi) - \frac{\beta}{2} \sum_{i=1}^{N} \{y_i - \mathbf{w}^T \cdot \mathbf{x}_i\}^2$$

To maximize the likelihood we take the gradient with respect to each parameter in $\mathbf{w}$; we write the jth position as w^j.

For simplicity, let's start by assuming that $\mathbf{x}_i$ is of length two. We'll exclude the bias term b (in $y = \mathbf{w}\mathbf{x} + b$) in this presentation, though it is easy to add.

There is only a single term that depends on $\mathbf{w}$ in the likelihood, which means the likelihood is proportional to the function $\mathcal{L}(\mathbf{w})$:

$$\log p(\mathcal{D}|\mathbf{w}, \mathbf{\Sigma}) \propto \mathcal{L}(\mathbf{w}) = -\frac{\beta}{2} \sum_{i=1}^{n} (y_i - w^0 x_i^0 - w^1 x_i^1)^2$$

Here we have expanded the weight vector into each of its components. Taking the gradient with respect to w^1:

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w^1} = -\beta \sum_{i=1}^{n} (y_i - w^0 x_i^0 - w^1 x_i^1)(-x_i^1)$$

We can set this to 0 and solve for w^1 in terms of w^0.

We can do the same step for w^0 to obtain.

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w^0} = -\beta \sum_{i=1}^{n} (y_i - w^0 x_i^0 - w^1 x_i^1)(-x_i^0)$$

We now have two equations and two variables, so we can now solve for w^1 and w^0.

Returning to the general case, we have a weight vector $\mathbf{w}$ of length M. Rather than writing the solution for each w^j separately, let's write them together. As before, the gradient of the log likelihood function with respect to $\mathbf{w}$ only depends on the last term. We can rearrange terms and drop β to yield:

$$\nabla \log p(\mathcal{D}|\mathbf{w}, \beta) = \sum_{i=1}^{n} (y_i - \mathbf{w}^T \mathbf{x}_i) \mathbf{x}_i^T$$

The right hand side evaluations to a vector of values, which is the solution for $\mathbf{w}$. By setting the gradient equal to the 0 vector, we distribute $\mathbf{x}_i^T$ and move in the summation:

$$0 = \sum_{i=1}^{n} y_i \mathbf{x}_i^T - \mathbf{w}^T \left(\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T \right)$$

Rearranging terms to solve for w:

$$\sum_{i=1}^{n} y_i \mathbf{x}_i^T = \mathbf{w}^T \left(\sum_{i=1}^{N} \mathbf{x}_i \mathbf{x}_i^T \right)$$

Notice that each sum is over a matrix, so we have matrix addition.

To move the matrix resulting from the summation over $\mathbf{x}$, we take the inverse and solve for $\mathbf{w}$ to obtain.

$$\mathbf{w} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{Y}$$

where $\mathbf{X}$ is a matrix where each row is a vector $\mathbf{x}_i$.

This solution expresses the maximum likelihood solution for $\mathbf{w}$ in terms of our training data $\mathbf{X}$ and $\mathbf{Y}$.

2.1 Loss function

In lecture we decided to use the sum of squares loss function:

$$\sum_{i=1}^{n}(y_i - \mathbf{w} \cdot \mathbf{x}_i)^2$$

We considered two methods for selecting $\mathbf{w}$.

1. Select the $\mathbf{w}$ that maximizes the likelihood of the observed data.
2. Select the $\mathbf{w}$ that minimizes the error on the observed data.

We chose to pursue the first and developed the maximum likelihood solution. Let us now consider the second approach.

Consider again the log-likelihood:

$$\log p(\mathcal{D}|\mathbf{w}, \mathbf{\Sigma}) = \frac{N}{2}\log\beta^{-1} - \frac{N}{2}\log(2\pi) - \frac{\beta}{2}\sum_{i=1}^{N}\{y_i - \mathbf{w}^T \cdot \mathbf{x}_i\}^2$$

As we mentioned before, only the last term depends on our choice of $\mathbf{w}$. Look carefully at this last term and our loss function. You will see they are the same. When we are maximizing the log-likelihood we are in effect minimizing the error function. Therefore, least squares linear regression is maximum likelihood estimation for Gaussians!

3 Regularized Least Squares

Forcing the most likely parameters for our data may cause us to favor parameters that do not generalize on test data. Following Occam's razor, we want to favor simpler models over complex models. Therefore, we want to both minimize the error but enforce a simple choice of parameters.

We can achieve this by rewriting our objective as a combination of two terms.

$$E_{\mathcal{D}}(\mathbf{w}) + \lambda E_W(\mathbf{w})$$

The first term $E_{\mathcal{D}}(\mathbf{w})$ is the error of $\mathbf{w}$ on our data $\mathcal{D}$. The second term is a measure of error of $\mathbf{w}$ on itself, ie. some intrinsic property of $\mathbf{w}$. We call the second term a regularizer, since it imposes some regular constraint over the weight vector. λ indicates how important each term is. We want to minimize both the error on the data and the complexity of $\mathbf{w}$.

We again choose the sum of squares error:

$$E_{\mathcal{D}}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}\{y_i - \mathbf{w}^T \cdot \mathbf{x}_i\}^2$$

To penalize $\mathbf{w}$ we take a similar approach and compute its sum of squares of the weight vector elements:

$$E_W(\mathbf{w}) = \frac{1}{2}\mathbf{w}^T\mathbf{w}$$

We combine these into a single objective.

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}\{y_i - \mathbf{w}^T \cdot \mathbf{x}_i\}^2 + \lambda\frac{1}{2}\mathbf{w}^T\mathbf{w}$$

This function is quadratic in $\mathbf{w}$ and so we can find its solution in closed form thanks to the quadratic formula. If we proceed as before and solve for $\mathbf{w}$ we get:

$$\mathbf{w} = \left(\lambda\mathbf{I} + \mathbf{X}^T\mathbf{X}\right)^{-1}\mathbf{X}^T\mathbf{Y}$$

$\mathbf{I}$ is the identity matrix, the all 0 matrix whose diagonal elements are all 1. Notice we are just increasing the values of $\mathbf{X}^T\mathbf{X}$ where λ indicates by how much. When λ is 0 we have regular least squares. When λ increases, $\mathbf{w}$ gets smaller since λ is in the inverse.

3.1 Generalized Regularization

We can write a more general form for regularization:

$$\mathcal{L}(\mathbf{w}) = \frac{1}{2}\sum_{i=1}^{N}\{y_i - \mathbf{w}^T \cdot \mathbf{x}_i\}^2 + \lambda\frac{1}{2}\sum_{j=1}^{M}|w^j|^q$$

Here we specify the regularizer by setting q. Notice that $q = 2$ is the quadratic regularizer. This is typically called $L2$ regularization.

Another common choice is $q = 1$, which is called $L1$ regularization. Regularized least squares with an $L1$ regularizer is called *Lasso*. Lasso has the property that for large λ the number of 0 terms in $\mathbf{w}$ increases leading to a sparse solution.

Sparse solutions are often desirable. In many applications, we assume that despite the large number of input features, only a few are actually predictive of the label. A sparse $\mathbf{w}$ enforces this assumption. Additionally, we may want to deploy our learned model on a resource limited device, such as a mobile phone. While we could learn a solution using many of the features, we'd prefer a model that used only a few features, reducing the cost of extracting these features and representing a large model.