Machine Learning and Pattern Recognition

Traiko Dinev <traiko.dinev@gmail.com>

July 1, 2019

NOTE: This partially follows Machine Learning and Pattern Recognition, a masters level course at the University of Edinburgh.
NOTE: Note this "summary" is NOT a reproduction of the course materials nor is it copied from the corresponding courses. It was entirely written and typeset from scratch.

1 Linear Regression

Consider the vector of features $\mathbf{x} = \langle x_1, x_2, \ldots, x_D \rangle$, where D is the number of dimensions. Then we want to learn this transformation (i.e. learn $\mathbf{w}$):

$$\mathbf{f} = X\mathbf{w} + b \tag{1}$$

where X is a $N \times D$ matrix, containing N examples of D dimensions. We have N targets, our training vector y of size $N \times 1$. This corresponds to computing $\mathbf{w}^T\mathbf{x}^{(i)}$, i.e. the dot product between the weights and training examples for each i. We can use mean squared error:

$$\min_{\mathbf{w}} \quad \sum_i (y^{(i)} - f^{(i)})^2 = (\mathbf{y} - \mathbf{f})^T(\mathbf{y} - \mathbf{f})$$

This has a linear solution of the form $\mathbf{w} = \underbrace{(X^TX)^{-1}X^T}_{\text{pseudo-inverse}}\mathbf{y}$. We can also introduce basis functions *phi*:

$$f(\mathbf{x}) = \sum_k w_k \ \phi_k(\mathbf{x})$$

$$\phi(\mathbf{x}) = \exp(-(\mathbf{x} - \mathbf{c})^T(\mathbf{x} - \mathbf{c})/h^2) \qquad \text{Radial Basis Function}$$

$$\sigma(\mathbf{x}) = \frac{1}{\exp(-\mathbf{v}^T\mathbf{x} - b)} \qquad \text{Sigmoid}$$

1.1 Regularization

We can add L2 regularization, which penalizes big weights.

$$\hat{E}(\mathbf{w}) = E(\mathbf{w}) + \lambda\mathbf{w}^T\mathbf{w} \tag{2}$$

which we can solve analytically to get something similar:

$$\mathbf{w}^* = (X^TX + \lambda\mathbb{I})^{-1}X^T\mathbf{y} \tag{3}$$

Alternatively, we can augment the $\mathbf{y}$ vector and X matrix to obtain the same results:

$$\mathbf{y}' = \begin{pmatrix} \mathbf{y} \\ \mathbf{0}_k \end{pmatrix}$$
$$\Phi' = \begin{pmatrix} \Phi \\ \sqrt{\lambda}\mathbb{I}_k \end{pmatrix}$$

2 Error Bars

We normally train on a *training set*, find best hyperparameters (like the L2 coefficients) on a *validation set* and judge only once on a *test set*.
Now let's say we compute some error measure or loss $L(y, f(x))$, like the mean squared error above. If we have a test set, we can compute the mean of this error:

$$L_{\text{test}} = \frac{1}{M}\sum_{m=1}^{M} L(y_m, f(x_m)) = \frac{1}{M}\sum_m L_m \tag{4}$$

This is a sum of test errors, which means that we can *most of the time* apply the **Central Limit Theorem** and say that L_{test} is approximately Gaussian. We need to have a finite variance and all the x_m's need to be independent, but we as things in ML often are, we assume things work until proven otherwise.

The x's actually come from an **input distribution**, which we don't know. That doesn't stop us from computing its mean and variance from independent samples (different than the sets above):

$$\mu \approx \bar{x} = \frac{1}{N}\sum_n x_n$$
$$\sigma^2 \approx \bar{\sigma}^2 = \frac{1}{N-1}\sum_n (x_n - \bar{x})^2$$

Now we can also compute the variance of the mean. If we collected a lot of sets, we can compute the variance:

$$\text{var}[\bar{x}] = \frac{1}{N^2}\sum_i \text{var}[x_n] = \frac{\sigma^2}{N} \approx \frac{\bar{\sigma}^2}{N} \tag{5}$$

Using the rules of variance (google variance of a sum). We can apply the same reasoning to the errors, where we replace x_n above by $L(x_n, f(x_n))$. This combined with the central theorem reasoning gives us a "confidence interval" like so:

$$\mu \pm \hat{\sigma}/\sqrt{N} \tag{6}$$

TL;DR: Report standard error bars.

3 Normal Distributions

A random variable is Gaussian (normal) if it has the following pdf:

$$p(y) = \mathcal{N}(y; \mu, 1) = \frac{1}{\sqrt{2\pi}} e^{-\frac{1}{2}(y-\mu)^2} \tag{7}$$

Standard normals have a mean of 0. We can scale and shift a standard normal like so:

$$z = \sigma x + \mu$$
$$x = \frac{z - \mu}{\sigma}$$

We then substitute the above in Equation 7 and scale the PDF to be normalized:

$$p(z) = \mathcal{N}(z; \mu, \sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2\sigma^2}(z-\mu)^2} \tag{8}$$

3.1 Multivariate Gaussians

The joint probability density of D univariate Gaussians will give us a multivariate one:

$$p(\mathbf{x}) = \prod_d p(x_d) = \frac{1}{(2\pi)^{D/2}} e^{-\frac{1}{2}\mathbf{x}^T\mathbf{x}} \tag{9}$$

This has a diagonal covariance matrix ($\mathbf{x}^T\mathbf{x}$). If we transform a variable as the one above:

$$\mathbf{y} = A\mathbf{x} \tag{10}$$

Then we can show that $\text{cov}[\mathbf{y}] = \Sigma = AA^T$. Also, assuming A is invertible, $\mathbf{x} = A^{-1}\mathbf{y}$. Substituting and normalizing, we obtain:

$$p(\mathbf{y}) = |2\pi\Sigma|^{-1/2} e^{-\frac{1}{2}\mathbf{y}^T\Sigma^{-1}\mathbf{y}} \tag{11}$$

Shifting the distribution ($\mathbf{z} = \mathbf{y} + \mu$), we obtain:

$$p(\mathbf{z}) = |2\pi\Sigma|^{-1/2} e^{-\frac{1}{2}(\mathbf{z}-\mu)^T\Sigma^{-1}(\mathbf{z}-\mu)} \tag{12}$$

Covariance matrices need to be **positive semi-definite**. This ensures that the above matrix is invertible and the hence the transformation doesn't "reduce" dimensionality:

$$\mathbf{z}^T\Sigma\mathbf{z} \geq 0, \text{for all real real } \mathbf{z} \tag{13}$$

4 Simple Classifiers

In classification targets $y^{(n)}$ are either $\{0, 1\}$ or $\{-1, +1\}$. Always 1-hot encode categorical variables: $\mathbf{x} = [0\ 1\ 0\ 0\ \ldots]^T$.

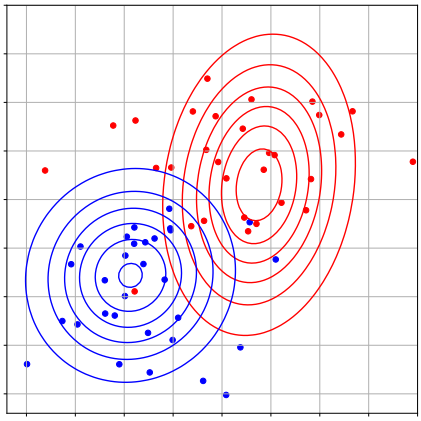
4.1 Generative Normal Models

We can assume that features come from a normal distribution for each class:

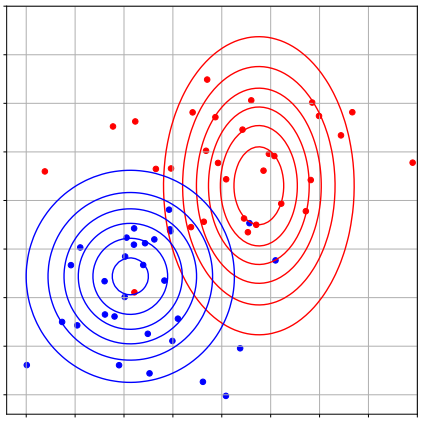
$$p(\mathbf{x}|y = k) = \mathcal{N}(\mathbf{x}, \boldsymbol{\mu}_k, \Sigma_k)$$

We can fit the mean and covariance to the mean and covariance of the vectors. Using Bayes' rule, we can get a probability for a label belonging to class k:

$$\begin{aligned} P(y = k|\mathbf{x}) &\propto p(\mathbf{x}|y = k)P(y = k) \\ &\propto \mathcal{N}(\mathbf{x}; \mu_k, \Sigma_k)\pi_k \end{aligned}$$



(a) A normal classifier



(b) A "naive" normal classifier. The multivariate Gaussian has no diagonal covariance, hence it can only stretch, but not rotate.

To get priors, we can just count:

$$P(y = k) = \pi_k \approx \frac{\Sigma_n I(y^{(n)} = k)}{N}$$

At test time we just compute probabilities for each class, normalize and hey-presto, classifier:

$$P(y = k|\mathbf{x}^{(\text{test})}, \theta) = \frac{s_k}{\sum_{k'} s_{k'}}$$

where $\theta = \{\boldsymbol{\mu}_k, \Sigma_k, \pi_k\}$, the learned parameters and $s_k = P(y = k|\mathbf{x})$ as above.

4.2 Naive Bayes

If the features are discrete, we can assume they are independent (big, very wrong assumption) and fit a model like this:

$$P(\mathbf{x}|y = k, \theta) = \prod_d P(x_d|y = k; \theta) = \prod_d \theta_{d,k}^{x_d}(1 - \theta_{d,k})^{1-x_d}$$

Here $\theta_{d,k} = P(x_d = 1|y = k)$.

We can also make the Normal Model above "naive" by making the covariance matrices diagonal, ignoring dependence (covariances) between features.

5 Logistic Regression

5.1 Gradient Descent

From the chain rule, we have $\frac{dw}{dt} = w_x \frac{dx}{dt} + w_y \frac{dy}{dt} + w_z \frac{dz}{dt}$. In vector notation, this becomes $\frac{dw}{dt} = \nabla w \cdot \frac{\mathbf{r}}{dt}$, where $\mathbf{r}$ is a curve in 3 dimensions. Thus:

$$\nabla w = \langle w_x, w_y, w_z \rangle$$

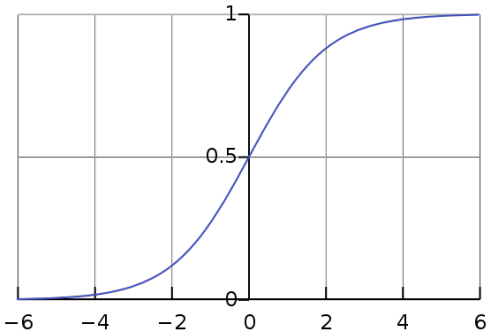


Figure 2: Logistic Sigmoid (from wikipedia)

The gradient vector's values depend on the point where we evaluate it. It is also perpendicular to a level surface $w = c$ by the following proof:

Proof: For any curve $\vec{r} = \vec{r}(t)$ that is inside the level surface $w = c$, the velocity is in the tangent plane. This follows from the fact that derivatives are tangents to the function curve from 1-D calculus. Then by the chain rule above $\frac{dw}{dt} = \nabla w \cdot \frac{\vec{r}}{dt} = \frac{c}{dt} = 0$. Hence the gradient is perpendicular to the surface.

Directional derivatives. These are derivatives in the direction of some vector $\hat{\mathbf{u}}$. Treat $\frac{\partial w}{\partial x}$ as the derivative along the x-axis. In other words, this is a slice of the function as given by the x-plane. Now let's have a curve $\vec{r}(t)$, such that it has a *normal* velocity $\hat{\mathbf{u}} = \langle a, b \rangle$. We can assume $|\hat{\mathbf{u}}| = 1$, since we are only interested in the direction. In other words:

$$\begin{cases} x(s) &= x_0 + as \\ y(s) &= y_0 + bs \end{cases}$$

Now s is the change in the direction of the curve. We can ask then what $\frac{\partial w}{\partial s}$ is. Well, by the chain rule:

$$\frac{\partial w}{\partial s} = \nabla w \cdot \frac{d\mathbf{r}}{ds} = \nabla w \cdot \hat{\mathbf{u}}$$

But from algebra we know:

$$\frac{\partial w}{\partial s} = \nabla w \cdot \hat{\mathbf{u}} = |\nabla w||\hat{\mathbf{u}}| \cos\theta = |\nabla w| \cos\theta$$

This is maximized when $\cos\theta = 1$, i.e. when the gradient $\hat{\mathbf{u}}$ is in the direction of ∇w. Hence the gradient points to the direction of steepest increase.

5.2 So what?

Patience. This can be applied to machine learning with the following idea. We normally define a loss or cost function L (or J as often seen in statistics). This is a function of our weights/ parameters. We usually aim to minimize said cost function. Well, if we know the direction of its steepest increase, $\nabla_{\mathbf{w}} L$, we know the direction of its steepest decrease - $-\nabla_{\mathbf{w}} L$. Hence we can take small steps in that direction:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla_{\mathbf{w}}$$

5.3 Logistic Regression.. finally

We want to classify things, which we can interpret as computing the probability $P(y = 1|\mathbf{x}, \mathbf{w})$. Enter the sigmoid:

$$f(\mathbf{x}; \mathbf{w}) = \sigma(\mathbf{w}^T \mathbf{x}) = \frac{1}{1 + e^{\mathbf{w}^T \mathbf{x}}}$$

See Figure 2. We also need to define our loss function. We define another **super-duper important** concept in ML - the likelihood of the data given the weights:

$$\begin{aligned}
\text{Likelihood} &= P(\mathcal{D}|\mathbf{w}) = P(\mathbf{x}^{(1)}, y^{(1)}, \ldots, \mathbf{x}^{(N)}, y^{(N)} \mid \mathbf{w}) \\
&= \underbrace{\prod_i P(\mathbf{x}^{(i)}, y^{(i)}| \mathbf{w})}_{\text{independent x's}} = \prod_i P(y^{(i)}| \mathbf{x}^{(i)}, \mathbf{w}) P(\mathbf{x}^{(i)} \mid \mathbf{w}) \\
&= \prod_i P(y^{(i)}| \mathbf{x}^{(i)}, \mathbf{w}) \underbrace{P(\mathbf{x}^{(i)})}_{\text{ignore weights}} = \prod_i P(y^{(i)}| \mathbf{x}^{(i)}, \mathbf{w})
\end{aligned}$$

We ignore the weights, since our model doesn't tell us anything about the input distribution. I.e. no matter the weights, the probability of having $\mathbf{x}$ will remain 1. Another way of saying this is that our model is **discriminative** as opposed to **generative**. We don't "generate" anything, everything has a probability 1 of being generated. [1]

We often minimize the negative **log-likelihood**, in this case primarily because it transforms the product into a sum. With some careful math, we can calculate it and fit logistic regression models.

5.4 Softmax Regression

If there are K classes, we can one-hot encode them as:

$$\mathbf{y} = [0\ 0\ 0\ \ldots\ 0\ 1\ 0\ \ldots\ 0]$$

where we have a 1 for the target class. We fit K weights, $\mathbf{w}^{(k)}$, such that our output contains K probabilities, for each class:

$$f_k = \frac{e^{(\mathbf{w}^{(k)})^T \mathbf{x}}}{\sum_{k'} e^{(\mathbf{w}^{(k')})^T \mathbf{x}}}$$

where we normalize by the sum of all outputs so that we get a proper probability distribution. We can use batch gradient descent to minimize the negative log-likelihood. The likelihood would be the following then:

$$\begin{aligned}
\nabla_{\mathbf{w}}^{(k)} \text{ LL} &= \nabla_{\mathbf{w}^{(k)}} \log \prod P(y_c = 1 \mid \mathbf{x}, W) \\
&= \sum_n \nabla_{\mathbf{w}^{(k)}} \log P(y_c = 1 \mid \mathbf{x}, W) \\
&= \sum_n (\mathbf{y}_k^{(n)} - f_k(\mathbf{x}^{(n)})) \mathbf{x}^{(n)}
\end{aligned}$$

5.5 Robust Regression

Sometimes we get outliers in the data we cannot filter in the pre-processing stage. We assume that for each observation there is a probability it isn't part of the data. Or, in math terms:

$$P(m) \ = \ \text{Bernoulli}(m; 1 - \epsilon) = \begin{cases} 1 - \epsilon & \text{m} = 1 \\ \epsilon & \text{m} = 0 \end{cases}$$

[1] *I know this is sloppy but I've looked and I haven't found a better (mathematically rigorous) explanation for this. Suggestions are welcome.*

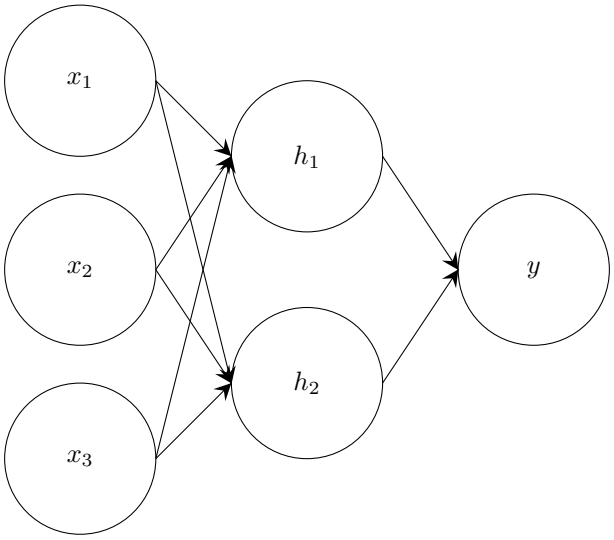


Figure 3: Neural Network illustration. The inputs are x_1, x_2, x_3, the hidden layers are h_1, h_2 and y is the output.

then we pick a random label when $m = 0$:

$$P(y = 1 \mid \mathbf{x}, \mathbf{w}, m) = \begin{cases} \sigma(\mathbf{w}^T \mathbf{x}) & m = 1 \\ \frac{1}{2} & m = 0 \end{cases} \tag{14}$$

The likelihood for a single example becomes:

$$\begin{aligned} P(y = 1 \mid \mathbf{x}, \mathbf{w}) &= \sum_m P(y = 1, m \mid \mathbf{x}, \mathbf{w}) \\ &= \sum_m P(y = 1 \mid \mathbf{x}, \mathbf{w}, m) P(m) \\ &= (1 - \epsilon)\sigma(\mathbf{w}^T \mathbf{x}) + \frac{\epsilon}{2} \end{aligned}$$

We can then derive the gradient for this and maximize the likelihood. We can treat ϵ as a hyperparameter and optimize over a validation set or set $\epsilon = \sigma(b)$ and optimize $b = \text{logit}(\epsilon) = \log \frac{\epsilon}{1-\epsilon}$.

6 Neural Networks

This section is taken from by Bachelor's thesis, so it might not read very well in a summary kind of way.

Neural networks are universal function approximators inspired by biological neurons (e.g. Goodfellow, Bengio, and Courville 2016, Chapter 6). The main building blocks of neural networks are called *neurons* and the networks are often drawn as directed graphs, as in figure 3. This graph illustrates the *flow* of the network, from the *inputs* x_1, x_2, x_3 to the *output* y. Each circle in the figure is called a neuron and each of the three vertical slices is called a layer. The layers that are not the input or output layers are called *hidden* layers.
In what is called the *supervised learning* scenario, the neural network is trained to predict the outputs y given the inputs $\mathbf{x}$ (see next section). If y is real-valued, then the task is called *regression* and is what we will attempt to do in the context of ABC.

We can define the transformations equivalently from the inputs to the hidden layer and from the hidden units to

output(s) via a system of equations:

$$\mathbf{h} = W^{(1)}\ \mathbf{x} \tag{15}$$

$$y = W^{(2)}\ \mathbf{h} \tag{16}$$

Thus we can increase or decrease the size of the input vector in subsequent layers of the network depending on the shapes of the matrices $W^{(1)}$ and $W^{(2)}$.

We apply a piecewise non-linearity called the *activation function* after each matrix multiplication, i.e. at each neuron; without it, the model could be collapsed to a single layer:

$$\mathbf{h} = h^{(1)}(W^{(1)}\ \mathbf{x}) \tag{17}$$

$$y = h^{(2)}(W^{(2)}\ \mathbf{h}) \tag{18}$$

Choices for activation functions $h(\mathbf{x})$ include hyperbolic tangent (tanh), sigmoids ($\sigma(x)$) and rectified linear units (relu):

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{19}$$

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{20}$$

$$\text{relu}(x) = \max(0, x) \tag{21}$$

Additionally, parametric rectifier units allow the activation to be non-zero (and hence, the derivative as well):

$$\text{prelu}(x) = \begin{cases} x & \text{if } x > 0 \\ a\ x & \text{otherwise} \end{cases} \tag{22}$$

The parameter a is learned akin to the rest of the neural network parameters.

Rectified (and parametric) linear units generally learn faster than tanh and sigmoid units and are preferred in state-of-the-art applications (Glorot, Bordes, and Bengio 2011). One improvement we will make is to replace the previously used tanh activations with relu units.

We can think of the hidden layers of the network as *learned* feature transformations. If the outputs y are real-valued and unbounded, i.e. in a regression task, we normally use a linear activation for the final layer, so we can think of the outputs from the previous hidden layer as features or *basis* functions for the linear output neuron.

$$y = W^{K-1}\ \mathbf{h}^{(K-1)} \tag{23}$$

where we have K layers, including the output.

6.1 Training Neural Networks

To train the neural network, we use *backpropagation* (Rumelhart, Hinton, and Williams 1985). More specifically, we use more optimized versions of backpropagation, namely *Adam* (Kingma and Ba 2014) and *RMSProp* (Tieleman and Hinton 2012).

Before we train, we specify a *loss function*. The loss function gives us a measure of "how bad" it is to differ from the true output y, where $\hat{y}$ is the output of the neural network. We then train on a set of examples $\{\mathbf{x}^{(i)}, \mathbf{y}^{(i)}\}_{i=0}^{N}$ via backpropagation. For a regression task, we use the mean-squared error loss function:

$$\mathcal{L}(y, \hat{y}) = \frac{1}{N} \sum_{i=1}^{N} (y^{(i)} - \hat{y}^{(i)})^2 \tag{24}$$

Backpropagation computes the derivative of the weight matrices $W^{(i)}$ with respect to the loss function for a *batch* of training examples from the training set. It then moves the weight in the direction of the negative gradient, thus minimizing the loss function. The backpropagation algorithm takes as a parameter a *learning rate*, which can be optimized. There are also more advanced versions, which often perform better, such as Adam (Kingma and Ba 2014) and RMSProp (Tieleman and Hinton 2012).

Algorithm 1 Basic Backpropagation Algorithm

1: initialize learning rate η
2: **while** Error is still high (see Early Stopping) **do**
3: $\quad \mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \mathbf{w}$ ▷ Update weights
4: **end while**

6.2 Modern Deep Neural Networks

Using the simple backpropagation algorithm and neural networks introduced so far does not scale well to larger models and datasets (Goodfellow, Bengio, and Courville 2016, Chapter 7). We can employ a number of techniques to improve the learning procedure. We describe here Batch Normalization (Ioffe and Szegedy 2015), Early Stopping (Prechelt 2012) and Dropout (Srivastava et al. 2014), as well as *regularization* in general.

A problem with larger networks is *overfitting*. Normally, we train the network on a training set and then evaluate the performance on a separate *validation set*, unseen during training. If the network learned the underlying structure of the data, then we say it *generalizes* and we expect the loss on the validation set to be comparable to that on the training set. Networks with more free parameters (weights) as compared to the number of training examples can, however, learn to "remember" quirks in the training instances instead of learning such an underlying structure. In this thesis, in the context of ABC, we have the benefit of procedurally generating the data, so we do not have issues with the number of examples; we can have as many as needed.

Early Stopping is a relatively easy way to prevent overfitting (Prechelt 2012). Consider figure 4. The training error normally will approach a very small number if the model has enough capacity. The validation error, however, usually starts increasing after it reaches a (sometimes local) minimum. If we continue to train, we no longer learn useful information, but rather quirks in the training set. Thus early stopping monitors the validation error at each epoch and stops training if the validation error continues to *increase* for a number of epochs. We then revert to the best epoch so far.

Batch Normalization forces the activations of different layers in the network to be normalized (Ioffe and Szegedy 2015). This has been shown to help with poor initialization strategies and large input values by forcing outputs of layers to take on the same distribution, thus helping learning. Using batch normalization after each (fully-connected) layer helps the network by eliminating the need to learn the different output distributions of each layer. The following transformation is used after applying the activation functions on each layer:

$$\hat{z}_i = \frac{h_i - \mu_i}{\sqrt{\sigma_i^2 + \epsilon}} \tag{25}$$

$$h_i = \gamma_i \ \hat{z}_i + \beta_i \tag{26}$$

Here h_i are the outputs of each hidden unit and μ_i and σ_i are minibatch mean and standard deviation, computed over training examples. γ_i and β_i are *learned* scaling parameters and ϵ is a small value to prevent division by 0.

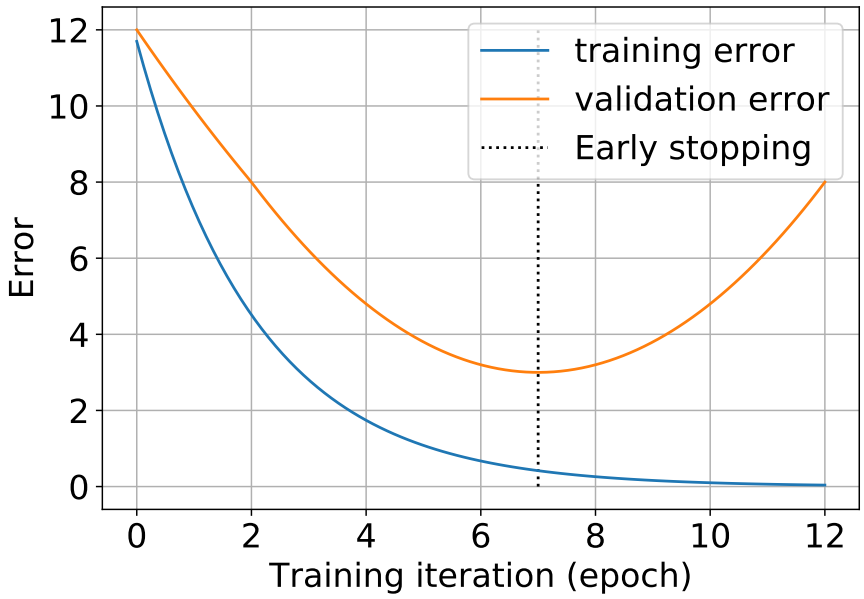


Figure 4: Neural Network overfitting illustration. The training error almost reaches 0, but the validation error starts to increase as the networks begins to learn structure present only in the training set.

Dropout is another simple regularization technique (Srivastava et al. 2014), that can prevent overfitting. During training we disconnect each neuron with a certain probability. This forces the other neurons to learn when some of the features are not present, thus making the network more flexible. At test time all the neurons are used and the outputs are scaled accordingly.

L2 Regularization is another way to prevent overfitting. We add a penalty term to the loss function for the entire neural network consisting of the squared absolute value of the weights for each layer. This forces the weights of the network to be small, thus ensuring the gradient updates are also small. It also prevents the activations from *exploding*.

$$\hat{\mathcal{L}}(y, \hat{y}, \{W^{(i)}\}_{i=1}^{K}) = \frac{1}{N}\sum_{i=1}^{N}(y^i - \hat{y}^i)^2 + \sum_{i,j,k}\lambda(W_{i,j}^{(k)})^2 \tag{27}$$

where λ is the regularization strength.
Note for all neural network models used in this thesis both the input data and the targets were standardized. This helps learning by avoiding large values for any neuron and therefore large gradients that become unstable:

$$\hat{x} \;=\; \frac{x - \langle x \rangle}{\sqrt{\mathrm{Var}[x]}} \tag{28}$$

7 Autoencoders and PCA

7.1 Autoencoders

Autoencoders learn transformations or representations of data. The basic idea is to create a bottleneck somewhere in the middle of the network so that networks learn to represent the data in a smaller number of dimensions. Thus autoencoders are functions that return the input approximately:

$$f(\mathbf{x}) \approx \mathbf{x} \tag{29}$$

We can use a neural network with a hidden size less than the input, like so:

$$\mathbf{h} = g^{(1)}(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$$
$$\mathbf{f} = g^{(2)}(W^{(2)}\mathbf{h} + \mathbf{b}^{(1)})$$

where $W^{(1)}$ is $K \times D$ and $K << D$. We can additionally impose a regularization penalty to further constrain the network. Sparse autoencoders, for instance, impose an L1 regularization. More about why that drives weights to zero below.

Denoising autoencoders learn to reconstruct the data when some of the inputs are missing or swapped or with added noise. Those are useful for removing noise from real-world images.

7.2 Principal Component Analysis (PCA)

We compute the covariance matrix of the data $\Sigma = \frac{1}{N}X^T X$. We want to reduce dimensionality (from D to K) in a way that maximizes the variance along the K new axis. This can be done by computing the eigenvectors of the covariance matrix and then projecting the data along those axis. These eigenvectors are also orthogonal to each other; if we keep all dimensions (i.e. project onto all eigenvectors), no information is lost. This process is also equivalent to a linear autoencoder with a square loss function (TODO: proof).

Often we standardize the data (i.e. $\frac{x-\mu}{\sigma}$) which means we can omit the $1N$ in the calculation above.

7.3 Singular Value Decomposition

Or SVD decomposes an $N \times D$ matrix into a product of a $N \times K$ matrix U, a diagonal $K \times K$ matrix S and $K \times D$ matrix V^T:

$$X \approx USV^T$$

The columns of U contain the eigenvectors of XX^T and the rows of U and the columns of V^T give K-dimensional embeddings of the rows and columns of X, respectively.

7.4 Probabilistic PCA

We want to embed the data $X = \{\mathbf{x}^{(n)}\}$ into a K-dimensional space, such that $K < D$. Let's assume that each point is embedded into a latent variable $\mathbf{z}^{(n)}$. We assume that the latent variables are normally distributed:

$$\mathbf{z}^{(n)} \sim \mathcal{N}(\mathbf{0}, \mathbb{I})$$

We also assume that the original data is generated via a projection:

$$\mathbf{x}^{(n)} \mid \mathbf{z}^{(n)} \sim \mathcal{N}(W\mathbf{z}^{(n)}, \sigma^2\mathbb{I}) \tag{30}$$

where the matrix W is $D \times K$. σ is known as the noise term. We are interested in estimating W and σ. Marginalizing out the $\mathbf{z}^{(n)}$, we arrive at:

$$\mathbf{x}^{(n)} \sim \mathcal{N}(\mathbf{0}, WW^T + \sigma^2\mathbb{I}) \tag{31}$$

This is then fitted via e.g. maximum likelihood.

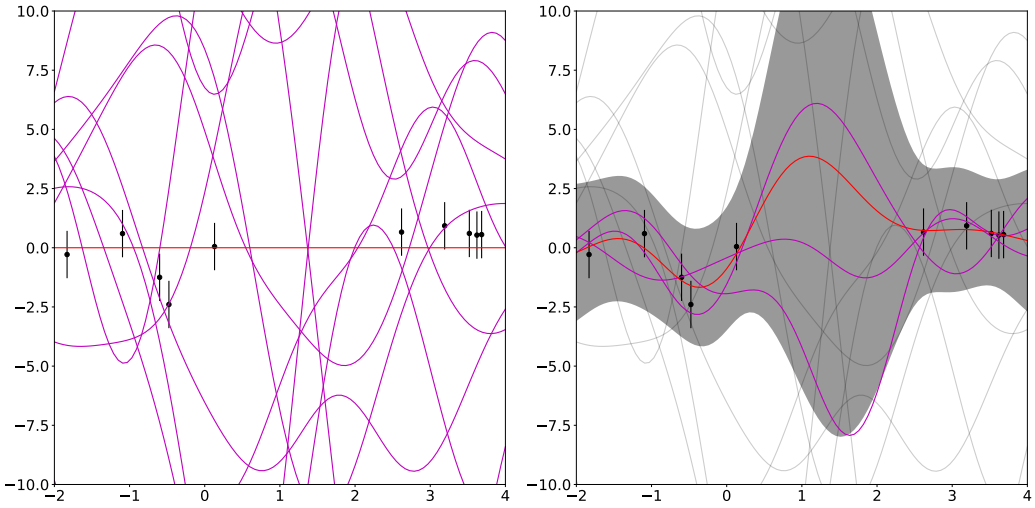


Figure 5: Example Bayesian Regression with basis functions. The left graph has fits sampled from the prior. The shaded region in the right graph is centered at the posterior mean with one standard deviation plotted. The red line is the mean fit. The bars on each datapoint represent the noise σ_y.

8 Bayesian Modeling

We can adopt a probabilistic model for regression. Our training data consists of pairs $(\mathbf{x}, y)$. We can assume a simple model:

$$P(y \mid \mathbf{x}, \mathbf{w}) = \mathcal{N}(y; f(\mathbf{x}, \mathbf{w}), \sigma_y^2)$$

where $f(\mathbf{x}, \mathbf{w})$ is any function. A quadratic regression would have a quadratic expansion, for example. We can then fit using maximum likelihood or in practice by minimizing the negative log-likelihood:

$$\begin{aligned} -\log P(\mathbf{y} \mid X, \mathbf{w}) &= -\sum_n log P(y^{(n)} \mid \mathbf{x}^{(n)}, \mathbf{w}) \\ &= \frac{1}{2\sigma_y^2} \sum_n \left[(y^{(n)} - f(\mathbf{x}^{(n)}, \mathbf{w})) \right] + \frac{N}{2} \log(2\pi\sigma_y^2) \end{aligned}$$

which is equivalent to fitting by least squares. Neat.

9 Bayesian Regression

We need to set-up two things - a probabilistic model, like the one above, and a prior belief for our model. A simple way to do this would be as follows:

$$\begin{aligned} P(\mathbf{w}) &= \mathcal{N}(\mathbf{w}; \mathbf{0}, \sigma_w^2 \mathbb{I}) \\ P(y \mid \mathbf{x}, \mathbf{w}) &= \mathcal{N}(y; \mathbf{w}^T \phi(\mathbf{x}^{(n)}), \sigma_y^2) \end{aligned}$$

where this is a linear regression and $\phi(\mathbf{x}^{(n)})$ represents any transformation we can apply. We can thus compute our posterior beliefs about the weights $\mathbf{w}$ given our training examples (using Bayes' theorem, of course):

$$
\begin{aligned}
P(\mathbf{w} \mid \mathcal{D}) &\propto P(\mathcal{D} \mid \mathbf{w}) P(\mathbf{w}) \\
&\propto \mathcal{N}(\mathbf{w}; \mathbf{0}, \sigma_w^2 \mathbb{I}) \prod_n \mathcal{N}(y^{(n)}; \mathbf{w}^T \phi(\mathbf{x}^{(n)}), \sigma_y^2) \\
&\propto \mathcal{N}(\mathbf{w}; \mathbf{0}, \sigma_w^2 \mathbb{I})\ \mathcal{N}(\mathbf{y}; \Phi \mathbf{w}, \sigma_y^2 \mathbb{I})
\end{aligned}
$$

This is also a Gaussian and can be derived (Murphy). If $\mathbf{w}_0 = \mathbf{0}$ for the prior and $V_0 = \sigma_w^2 \mathbb{I}$, then:

$$
\begin{aligned}
V_n &= \sigma_y^2 (\sigma_y^2 V_0^{-1} + \Phi^T \Phi)^{-1} \\
\mathbf{w}_n &= V_n V_0^{-1} \mathbf{w}_0 + \frac{1}{\sigma_y^2} V_N \Phi^T \mathbf{y}
\end{aligned}
$$

where $\mathbf{w}_N$ is the posterior mean and V_n - the posterior covariance.

9.1 Test Predictions

So we have our posterior $\mathbf{w}$. Now we want, conditioned on the training data $\mathcal{D} = \{\mathbf{x}^{(n)}, y^{(n)}\}$ to give predictions for a new test point $\mathbf{x}^*$ with output y^*. We introduce our weights using the sum rule:

$$
P(y^* \mid \mathbf{x}^*, \mathcal{D}) = \int P(y^*, \mathbf{w} \mid \mathbf{x}^*, \mathcal{D})\ \mathrm{d}\mathbf{w}
$$

Now we split the above using the product rule:

$$
\begin{aligned}
P(y^* \mid \mathbf{x}^*, \mathcal{D}) &= \int P(y^* \mid \mathbf{w}, \mathbf{x}^*, \mathcal{D})\ P(\mathbf{w} \mid \mathbf{x}^*, \mathcal{D})\ \mathrm{d}\mathbf{w} \\
&= \int P(y^* \mid \mathbf{w}, \mathbf{x}^*)\ P(\mathbf{w} \mid \mathcal{D})\ \mathrm{d}\mathbf{w}
\end{aligned}
$$

where $\mathbf{w}$ does not depend on the test input $\mathbf{x}^*$. The test prediction also doesn't depend on the training data if we know $\mathbf{w}$, the parameters. Now, we know the posterior for $\mathbf{w}$:

$$
P(\mathbf{w} \mid \mathcal{D}) = \mathcal{N}(\mathbf{w}; \mathbf{w}_N, V_N)
$$

For linear models the above integral is Gaussian. Expecting that, we can write:

$$
\begin{aligned}
y^* &= f(\mathbf{x}^*) + \nu = \mathbf{w}^T \mathbf{x}^* + \nu \\
\nu &\sim \mathcal{N}(0, \sigma_y^2)
\end{aligned}
$$

This matches what we previously descibed as:

$$
P(y^* \mid \mathbf{w}) = \mathcal{N}(y^*; \mathbf{w}^T \mathbf{x}^*, \sigma_y^2)
$$

We can thus show that:

$$
P(y^* \mid \mathcal{D}, \mathbf{x}^*) = \mathcal{N}(y^*; \mathbf{w}_N^T \mathbf{x}^*, (\mathbf{x}^*)^T V_n \mathbf{x}^* + \sigma_y^2)
$$

9.2 Decision Making

The above answer is a distribution of y^*. Sometimes we need to ouput a single prediction, a guess. Thus we need to write a loss function $L(y, y^*)$, which says how bad it is to guess y^* if the actual output is y. We can compute the expected loss:

$$c = \mathbb{E}_{P(y \mid \mathcal{D})}[L(y, y^*)] = \int L(y, y^*)\ P(y \mid \mathcal{D})\ \mathrm{d}y$$

For square loss, $L(y, y^*) = (y - y^*)^2$, we can differentiate the cost c with respect to our guess:

$$\frac{\partial c}{\partial y^*} = \mathbb{E}_{P(y \mid \mathcal{D})} \frac{\partial L(y, y^*)}{\partial y^*} = \mathbb{E}_{P(y \mid \mathcal{D})}[2(y - y^*)] = 2(\mathbb{E}_{P(y \mid \mathcal{D})}[y] - y^*)$$

Setting this to 0, the optimal guess is the posterior mean, $y^* = \mathbb{E}_{P(y \mid \mathcal{D})}[y]$. Thus if we only need the guess, using Bayesian regression does not change much at all. In fact this can be shown to correspond to a L2-regularized standard linear regression.

10 Bayesian Logistic Regression

For logistic regression, the likelihood function (as above) is defined as:

$$\begin{aligned} P(\mathcal{D} \mid \mathcal{M}) &= P(\mathcal{D} \mid \mathbf{w}) = P(\mathbf{x}^{(1)}, y^{(1)}, \ldots, \mathbf{x}^{(N)}, y^{(N)} \mid \mathbf{w}) \\ &= \underbrace{\prod_i P(\mathbf{x}^{(i)}, y^{(i)} \mid \mathbf{w})}_{\text{independent x's}} = \prod_i P(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w})\ P(\mathbf{x}^{(i)} \mid \mathbf{w}) \\ &= \prod_i P(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}) \quad \underbrace{P(\mathbf{x}^{(i)})}_{\text{ignore weights}} \quad = \prod_i P(y^{(i)} \mid \mathbf{x}^{(i)}, \mathbf{w}) \end{aligned}$$

When fitting via maximum likelihood, we do something like:

$$\mathbf{w}^* = \arg\max_{\mathbf{w}} [\log\ P(\mathbf{y} \mid X, \mathbf{w}) - \mathbf{w}^T \mathbf{w}]$$

or, as most optimizers are optimized for minimizing functions, equivalently minimize the negative log-likelihood. Bayesian logistic regression begins the same way as Bayesian linear regression:

$$P(\mathbf{w} \mid \mathcal{D}) = \frac{P(\mathcal{D} \mid \mathbf{w}) P(\mathbf{w})}{P(\mathcal{D})} \propto P(\mathcal{D} \mid \mathbf{w}) P(\mathbf{w})$$

To make predictions (for a datapoint $y^*, \mathbf{x}^*$), we can use probability theory:

$$\begin{aligned} P(y^* \mid \mathbf{x}^*, \mathcal{D}) &= \int P(y^*, \mathbf{w} \mid \mathbf{x}^*, \mathcal{D}) \mathrm{d}\mathbf{w} \\ &= \int P(y^* \mid \mathbf{x}^*, \mathbf{w}) P(\mathbf{w} \mid \mathcal{D}) \mathrm{d}\mathbf{w} \end{aligned}$$

The problem, of course, is that we can't really compute the posterior $P(\mathbf{w} \mid \mathcal{D})$. It never really is Gaussian as it were in the linear regression case. Thus we must be clever. In reality, if we assume a Gaussian prior, then posterior is this:

$$\begin{aligned} P(\mathbf{w}) &= \mathcal{N}(\mathbf{w}; \mathbf{0}, \sigma_w^2 \mathbb{I}) \\ P(\mathbf{w} \mid \mathcal{D}) &= \mathcal{N}(\mathbf{w}; \mathbf{0}, \sigma_w^2 \mathbb{I}) \prod_n \sigma(\mathbf{w}^T \mathbf{x}^{(n)} z^{(n)}) \end{aligned}$$

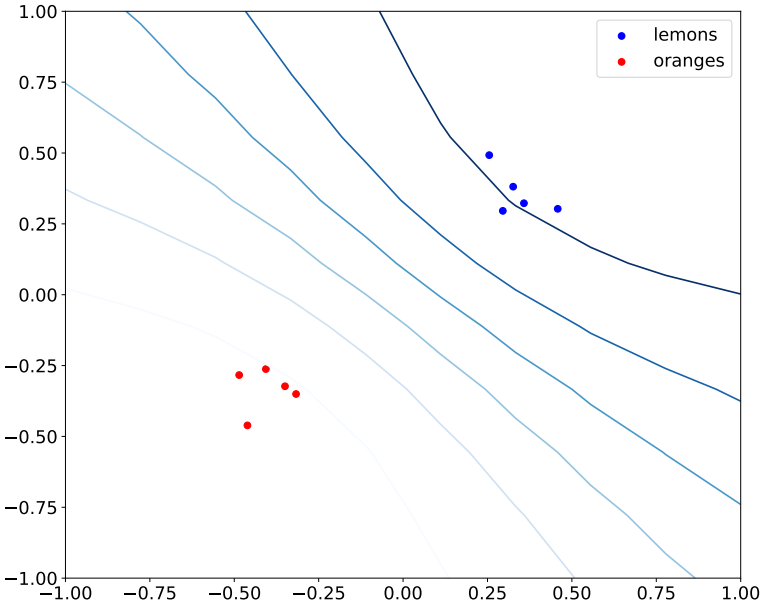


Figure 6: Bayesian Logistic Regression via Importance Sampling

where $z^n = \{+1, -1\}$ instead of $\{0, 1\}$. If we notice that $\sigma(-a) = 1 - \sigma(a)$, then the above follows. Either way, this product of sigmoids and a Gaussian is nasty and very slow to integrate numerically above. We can do importance sampling to predict like so:

10.1 Importance Sampling

$$\begin{aligned} P(y^* \mid \mathbf{x}^*, \mathcal{D}) &= \mathbb{E}_{P(\mathbf{w} \mid \mathcal{D})}[P(y^* \mid \mathbf{x}^*, \mathbf{w})] \\ &\approx \frac{1}{S} \sum_{s=1}^{S} P(y \mid , \mathbf{x}^*, \mathbf{w}^{(s)}) \end{aligned}$$

where $\mathbf{w}^{(s)} \sim P(\mathbf{w} \mid \mathcal{D})$. Sampling from the posterior $P(\mathbf{w} \mid \mathcal{D})$ is hard. We can do many things, like MCMC, but perhaps the easiest is importance sampling:

$$\begin{aligned} P(y^* \mid \mathbf{x}^*, \mathcal{D}) &= \int P(y^* \mid \mathbf{x}^*, \mathbf{w}) P(\mathbf{w} \mid \mathcal{D}) \, \mathrm{d}\mathbf{w} \\ &= \int P(y^* \mid \mathbf{x}^*, \mathbf{w}) \frac{P(\mathcal{D} \mid \mathbf{w}) P(\mathbf{w})}{P(\mathcal{D})} \, \mathrm{d}\mathbf{w} \\ &= \mathbb{E}_{P(\mathbf{w})}[P(y^* \mid \mathbf{x}^*, \mathbf{w}) \frac{P(\mathcal{D} \mid \mathbf{w})}{P(\mathcal{D})}] \\ &= \mathbb{E}_{P(\mathbf{w})}[P(y^* \mid \mathbf{x}^*, \mathbf{w}) \frac{P(\mathcal{D} \mid \mathbf{w})}{\int P(\mathcal{D} \mid \mathbf{w}) P(\mathbf{w}) \, \mathrm{d}\mathbf{w}}] \\ &= \mathbb{E}_{P(\mathbf{w})}[P(y^* \mid \mathbf{x}^*, \mathbf{w}) \frac{P(\mathcal{D} \mid \mathbf{w})}{\mathbb{E}_{P(\mathbf{w})}[P(\mathcal{D} \mid \mathbf{w})]}] \end{aligned}$$

Now we can do importane sampling in the denominator and in the entire expression, perhaps even storing samples from $P(\mathbf{w})$, the prior. The end result looks like this:

$$P(y^* \mid \mathbf{x}^*, \mathcal{D}) \approx \frac{1}{S} \sum_{s} P(y^* \mid \mathbf{x}^*, \mathbf{w}^{(s)}) \frac{P(\mathcal{D} \mid \mathbf{w}^{(s)})}{\frac{1}{S'} \sum_{s'} P(\mathcal{D} \mid \mathbf{w}^{(s')})}$$

where $w^{(s)} \sim P(\mathbf{w})$. The bias of this, of course, is humongous. Figure 6 shows an example result of this procedure.

Another thing we can do is:

10.2 Maximum a Posteriori

which is just this:

$$\mathbf{w}^* = \arg\max_{\mathbf{w}}[\log P(\mathbf{w} \mid \mathcal{D})] = \arg\max_{\mathbf{w}}[\log P(\mathcal{D} \mid \mathbf{w})P(\mathbf{w})]$$

c.f. maximum likelihood. Really we only added the prior here. Onwards to:

10.3 Laplace Approximation

Let's assume that the posterior $p(\mathbf{w} \mid \mathcal{D})$ is Gaussian. That's not a very good assumption, putting it mildly. We can try matching the best Gaussian to it. We first find the maximum likelihood, the most probable value of $\mathbf{w}$, which we label $\mathbf{w}^*$, as usual:

$$\begin{aligned}
\mathbf{w}^* &= \arg\max_{\mathbf{w}} P(\mathbf{w} \mid \mathcal{D}) \\
&= \arg\max_{\mathbf{w}} \ \log \frac{P(\mathcal{D}, \mathbf{w})}{P(\mathcal{D})} \\
&= \arg\max_{\mathbf{w}} \ \log P(\mathcal{D}|\mathbf{w})P(\mathbf{w})
\end{aligned}$$

The last line follows from the fact that $P(\mathcal{D})$ is independent of $\mathbf{w}$, i.e. a constant in the optimization problem. We also can't really evaluate $P(\mathcal{D})$, since it's a nasty integral. So far, so good. We can define the energy as follows:

$$\begin{aligned}
E(\mathbf{w}) &= -\log P(\mathbf{w}, \mathcal{D}) \\
\mathbf{w}^* &= \arg\min_{\mathbf{w}} E(\mathbf{w})
\end{aligned}$$

Then we compute the Hessian, the matrix of second derivatives. This gives us the curvature of the function:

$$H_{i,j} = \frac{\partial^2 E(\mathbf{w}^*)}{\partial w_i \ \partial w_j}$$

The energy of a Gaussian distribution would be, up to a constant:

$$E_{\mathcal{N}}(w) = \frac{(w-\mu)^2}{2\sigma^2}$$

The minimum is at $w^* = \mu$ and the second derivative - $H = 1/\sigma^2$. Generalizing to a higher dimension, we get:

$$E_{\mathcal{N}}(\mathbf{w}) = \frac{1}{2}(\mathbf{w}-\mu)^T \Sigma^{-1}(\mathbf{w}-\mu)$$

where $\mathbf{w}^* = \mu$ and $H = \Sigma^{-1}$ or $\Sigma = H^{-1}$. Thus matching energies, we get:

$$P(\mathbf{w} \mid \mathcal{D}) \approx \mathcal{N}(\mathbf{w}; \mathbf{w}^*, H^{-1})$$

To approximate $P(\mathcal{D})$, we can do:

$$P(\mathbf{w} \mid \mathcal{D}) = \frac{P(\mathbf{w}, \mathcal{D})}{P(\mathcal{D})} \approx \mathcal{N}(\mathbf{w}; \mathbf{w}^*, H^{-1}) = \frac{|H|^{1/2}}{(2\pi)^{D/2}} \exp(-\frac{1}{2}(\mathbf{w}-\mathbf{w}^*)^{\mathrm{T}} H(\mathbf{w}-\mathbf{w}^*))$$

This is true for any $\mathbf{w}$, so let's evaluate it for $\mathbf{w}^*$, giving us:

$$P(\mathbf{w} \mid \mathcal{D}) = \frac{P(\mathbf{w}, \mathcal{D})}{P(\mathcal{D})} \approx \mathcal{N}(\mathbf{w}^*; \mathbf{w}^*, H^{-1}) = \frac{|H|^{1/2}}{(2\pi)^{D/2}}$$

which gives us:

$$P(\mathcal{D}) \approx P(\mathbf{w}^*, \mathcal{D}) \; |2\pi H^{-1}|^{1/2}$$

10.4 Test Predictions

For a test point $y^*, \mathbf{x}^*$, we need to compute:

$$\begin{aligned} P(y^* \mid \mathbf{x}^*, \mathcal{D}) &= \int P(y^*, \mathbf{w} \mid \mathbf{x}^*, \mathcal{D}) \; \mathrm{d}\mathbf{w} \\ &= \int P(y^* \mid \mathbf{x}^*, \mathbf{w}) \; P(\mathbf{w} \mid \mathcal{D}) \; \mathrm{d}\mathbf{w} \end{aligned}$$

Using the Laplace approximation above, we know that $P(\mathbf{w} \mid \mathcal{D}) \approx \mathcal{N}(\mathbf{w}; \mathbf{w}^*, H)$, we have the following integral:

$$\begin{aligned} P(y^* = 1 \mid \mathbf{x}, \mathcal{D}) &\approx \int \sigma(\mathbf{w}\mathbf{x}^{\mathrm{T}}) \mathcal{N}(\mathbf{w}; \mathbf{w}^*, H^{-1}) \; \mathrm{d}\mathbf{w} \\ &= \mathbb{E}_{\mathcal{N}(\mathbf{w}; \mathbf{w}^*, H^{-1})}[\sigma(\mathbf{w}\mathbf{x}^{\mathrm{T}})] \end{aligned}$$

Calling $a = \mathbf{w}^*\mathbf{x}^{\mathrm{T}}$ and changing variables:

$$\begin{aligned} P(a) &= \mathcal{N}(a; \mathbf{w}^*\mathbf{x}^{\mathrm{T}}, \mathbf{x}^{\mathrm{T}} H^{-1} \mathbf{x}) \\ P(y = 1 \mid \mathbf{x}, \mathcal{D}) &= \int \sigma(a) \; \mathcal{N}(a; \mathbf{w}^*\mathbf{x}^{\mathrm{T}}, \mathbf{x}^{\mathrm{T}} H^{-1} \mathbf{x}) \; \mathrm{d}a \end{aligned}$$

which we can then compute numerically. From Murphy Section 8.4.4.2, we can derive another approximation:

$$\begin{aligned} P(y = 1 \mid \mathbf{x}, \mathcal{D}) &\approx \sigma(\kappa \mathbf{w}^* \mathbf{x}) \\ \kappa &= \frac{1}{\sqrt{1 + \frac{\pi}{8} \mathbf{x}^{\mathrm{T}} V \mathbf{x}}} \end{aligned}$$

11 Optimizing Hyperparameters in a Bayesian Setting

We normally opt to choose simpler models, that is models that assign a high probability to the training data, over models that have an high probability of generating many datasets. We can thus specify the probability:

$$\begin{aligned} P(\mathcal{D} \mid \mathcal{M}) = P(X, \mathbf{y} \mid \mathcal{M}) &= P(\mathbf{y} \mid X, \mathcal{M}) \underbrace{P(X \mid \mathcal{M})}_{1} \\ &= P(\mathbf{y} \mid X, \mathcal{M}) = \int P(\mathbf{y}, \mathbf{w} \mid X, \mathcal{M}) \; \mathrm{d}\mathbf{w} = \int P(\mathbf{y} \mid X, \mathbf{w}, \mathcal{M}) \; P(\mathbf{w} \mid \mathcal{M}) \; \mathrm{d}\mathbf{w} \end{aligned}$$

The probability of the data given the model, i.e. $P(\mathcal{D} \mid \mathcal{M})$ is the **marginal likelihood** of the model. We can use it to select between different models $\mathcal{M}_k$.

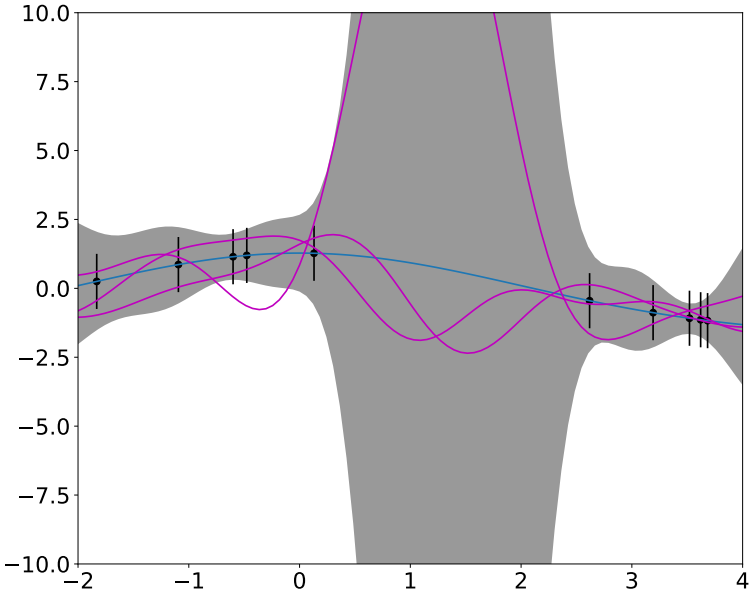


Figure 7: Gaussian Process posterior

(*Yet another example from MLPR*) If we have a linear model with:

$$P(\mathbf{w} \mid \sigma_w) = \mathcal{N}(\mathbf{w}; \mathbf{0}, \sigma_w^2 \mathbb{I})$$
$$P(y \mid \mathbf{w}, \sigma_y) = \mathcal{N}(y; \mathbf{w}^{\mathrm{T}}\mathbf{x}, \sigma_y^2)$$

We then fit σ_w and σ_y by maximizing the marginal likelihood:

$$P(\mathbf{y} \mid X, \sigma_w, \sigma_y) = \int P(\mathbf{y} \mid X, \mathbf{w}, \sigma_y)\ P(\mathbf{w} \mid \sigma_w)\ \mathrm{d}\mathbf{w}$$

12 Gaussian Processes

Gaussian processes are just big Gaussians. Let's say we have a training set of values $\mathbf{y}$ and inputs X and a test set of values (unknown) $\mathbf{f}_*$ and test inputs X_*. If we assume that everything comes from a Gaussian, i.e.:

$$P\left(\begin{bmatrix}\mathbf{y}\\ \mathbf{f}_*\end{bmatrix}\right) = \mathcal{N}\left(\begin{bmatrix}\mathbf{y}\\ \mathbf{f}_*\end{bmatrix}; \mathbf{0}; \begin{bmatrix}K(X,X) + \sigma_y^2\mathbb{I} & K(X,X_*)\\ K(X_*,X) & K(X_*,X_*)\end{bmatrix}\right)$$

This really is the essence of the thing. The K functions are the kernels that represent the covariance between datapoints. This is enough to describe any dataset; in fact a non-zero mean Gaussian process is always equivalent to a zero mean one (TODO: proof). The kernels are defined as:

$$K(X,Y)_{i,j} = k(\mathbf{x}^{(i)}, \mathbf{z}^{(j)})$$

for two test inputs $\mathbf{x}$ and $\mathbf{z}$. For training inputs we also have noise variance σ_y^2. Let's call the noise:

$$\nu_i \sim \mathcal{N}(0, \sigma_y^2)$$

then for the cross-covariance:

$$\begin{aligned}\operatorname{cov}(y_i, f_{*,j}) &= \mathbb{E}[y_i f_{*,j}] - \mathbb{E}[y_i]\mathbb{E}[f_{*,j}] \\ &= \mathbb{E}[(f_i + \nu_i) f_{*,j}] \\ &= \mathbb{E}[f_i f_{*,j}] + \underbrace{\mathbb{E}[\nu_i]}_{0}\mathbb{E}[f_{*,j}] \\ &= \mathbb{E}[f_i f_{*,j}] = k(\mathbf{x}^{(i)}, \mathbf{x}^{*,j})\end{aligned}$$

Once we specify the kernel k, we can extract the distribution of the test datapoints by simply marginalizing the big Gaussian:

$$\begin{aligned}P(\mathbf{f}_* \mid \mathcal{D}, \mathcal{M}) = \mathcal{N}(&\mathbf{f}_*; \\ &K(X_*, X)(K(X, X) + \sigma_y^2 \mathbb{I})^{-1}\mathbf{y}, \\ &K(X_*, X_*) - K(X_*, X)(K(X, X) + \sigma_y^2 \mathbb{I})^{-1}K(X, X_*) \\ &)\end{aligned}$$

which is the posterior over test datapoints.

12.1 Gaussian Kernel

A common choice for a kernel is the Gaussian kernel:

$$k(\mathbf{x}^i, \mathbf{x}^j) = \sigma_f^2 \exp\left(-\frac{1}{2}\sum_{d=1}^{D}(x_d^{(i)} - x_d^j)^2 / l_d^2 \right)$$

We can fit σ_f and l_d by maximizing the marginal likelihood.

13 Variational Inference

Not going into details here. The idea is to minimize the Kullback-Leibler Divergence between two distributions. We can use it instead of the Laplace Approximation for Bayesian Logistic Regression above.

$$D_{KL}(p||q) = \int p(\mathbf{x}) \log \frac{p(\mathbf{x})}{q(\mathbf{x})} \, \mathrm{d}\mathbf{x}$$

We can minimize the KL divergence $D_{KL}(q(\mathbf{w};\alpha) \mid\mid P(\mathbf{w} \mid \mathcal{D}))$ in the logistic regression case. We are trying to match a parametric function p with some parameters captured in α to the posterior distribution of the weights $P(\mathbf{w} \mid \mathcal{D})$:

$$\begin{aligned}D_{KL}(q(\mathbf{w};\alpha) \mid\mid P(\mathbf{w} \mid \mathcal{D})) &= \int q(\mathbf{w};\alpha) \log \frac{q(\mathbf{w};\alpha)}{P(\mathbf{w} \mid \mathcal{D})} \, \mathrm{d}\mathbf{w} \\ &= -\int q(\mathbf{w};\alpha) \log P(\mathbf{w} \mid \mathcal{D}) \, \mathrm{d}\mathbf{w} + \underbrace{\int q(\mathbf{w};\alpha) \log q(\mathbf{w};\alpha) \, \mathrm{d}\mathbf{w}}_{\text{negative entropy}, -H(q)}\end{aligned}$$

The first term is large if we put probability on regions where the posterior is small. The second term encourages more spread-out distributions with high entropy.

Using Bayes' theroem in the KL divergence formula:

$$D_{KL}(q \mid\mid p) = \underbrace{\mathbb{E}_q[\log q(\mathbf{w})] - \mathbb{E}_q[\log P(\mathcal{D} \mid \mathbf{w})] - \mathbb{E}_q[\log P(\mathbf{w})]}_{J(q)} + \log P(\mathcal{D})$$

We thus minimize $J(q)$, also known as ELBO (Evidence Lower Bound). We can get a bound on the $P(\mathcal{D})$ term, the marginal likelihood, by observing the KL divergence is non-negative:

$$\log P(\mathcal{D}) \geq -J(q)$$

13.1 Minimizing the KL-divergence for a Gaussian function

Let $\alpha = \{\mathbf{m}, V\}$, the mean and co-variance of a Gaussian distribution. We will be (numerically) minimizing the $J(q)$ with respect to α. Thus:

$$q(\mathbf{w}; \alpha) = \mathcal{N}(\mathbf{w}; \mathbf{m}, V)$$

We first Cholesky decompose V to get $V = LL^{\mathrm{T}}$, L being a lower-triangular matrix with positive diagonal entries. We also optimize $\log \sigma_w$; otherwise we might make σ_w negative.

$$L = \begin{cases} \log L_{i,j} & i = j \\ L_{i,j} & i \neq j \end{cases}$$

We assume a Gaussian prior:

$$P(\mathbf{w}) = \mathcal{N}(\mathbf{w}; \mathbf{0}, \sigma_{\mathbf{w}}^{\mathbf{2}} \mathbb{I})$$

We can write two of the terms of $J(q)$ in closed form:

$$\begin{aligned} \mathbb{E}_{\mathcal{N}(\mathbf{w};\mathbf{m},V)}[\mathcal{N}(\mathbf{w}; \boldsymbol{\mu}, \Sigma)] &= \mathbb{E}_{\mathcal{N}(\mathbf{w};\mathbf{m},V)}\left[-\frac{1}{2}(\mathbf{w} - \boldsymbol{\mu})^{\mathrm{T}} \Sigma^{-1} (\mathbf{w} - \boldsymbol{\mu}) \right] - \frac{1}{2} \log |2\pi\Sigma| \\ \mathbb{E}_q[\log q(\mathbf{w})] &= \mathbb{E}_{\mathcal{N}(\mathbf{w};\mathbf{m},V)}[\log \mathcal{N}(\mathbf{w}; \mathbf{m}, V)] = -\frac{D}{2} - \frac{1}{2} \log |2\pi V| \\ \mathbb{E}_q[\log P(\mathbf{w})] &= \frac{1}{2\sigma_w^2} \left[\mathrm{Tr}(V) + \mathbf{m}^{\mathrm{T}} \mathbf{m} \right] + \frac{D}{2} \log(2\pi\sigma_w^2) \end{aligned}$$

Using this trick:

$$\begin{aligned} \mathbb{E}_{\mathcal{N}(\mathbf{z};0,V)}[\log \mathcal{N}(\mathbf{z}; 0, V)] &= \mathbb{E}[-\frac{1}{2} \mathrm{Tr}(\mathbf{z}^{\mathrm{T}} V^{-1} \mathbf{z})] = -\frac{1}{2} \mathrm{Tr}(\mathbb{E}[\mathbf{z}^{\mathrm{T}} V^{-1} \mathbf{z}]) = -\frac{1}{2} \mathrm{Tr}(\mathbb{E}[\mathbf{z}^{\mathrm{T}} \mathbf{z}] V^{-1}) \\ &= -\frac{1}{2} \mathrm{Tr}(V V^{-1}) = -\frac{1}{2} \mathrm{Tr}(\mathbb{I}_D) = -\frac{D}{2} \end{aligned}$$

where $\mathbb{E}[\mathbf{z}^{\mathrm{T}} \mathbf{z}] = V$ by the definition of covariance for a 0-mean random variable. We also know that $\mathbf{z}^{\mathrm{T}} V \mathbf{z}$ is a number and hence equal to its trace, a.k.a. the "trace trick".

From the Cholesky decomposition:

$$\begin{aligned} \frac{1}{2} \log |V| &= \sum_i \log L_{i,i} \\ \mathrm{Trace}(V) &= \sum_{i,j} L_{i,j}^2 \end{aligned}$$

hence the above reparametrization.

13.2 Log-likelihood term

The nastiest of the three terms is the log-likelihood:

$$-\mathbb{E}_{\mathcal{N}(\mathbf{w};\mathbf{m},V)}[\log P(\mathcal{D} \mid \mathbf{w})] = -\mathbb{E}_{\mathcal{N}(\mathbf{w};\mathbf{m},V)}\left[\sum_{n=1}^{N} \log p(y^{(n)} \mid \mathbf{x}^{(n)}, \mathbf{w})\right]$$

We cannot compute this in closed form. Instead we do a Monter Carlo estimate:

$$-\mathbb{E}_{\mathcal{N}(\mathbf{w};\mathbf{m},V)}[\log P(\mathcal{D} \mid \mathbf{w})] \approx -\sum_{n=1}^{N} \log p(y^{(n)}, \mathbf{x}^{n}, \mathbf{w})$$

where $\mathbf{w} \sim \mathcal{N}\mathbf{m}, V$.

13.3 Gradients of the log-likelihood

Here we do a "reparametrization trick" (gotta love those). To sample a random $\mathbf{w}$, we sample a vector of normal variables $\nu \sim \mathcal{N}(0, \mathbb{I})$ and then transform it (see multivariate Gaussians above): $\mathbf{w} = \mathbf{m} + L\nu$. We can rewrite the expectation:

$$\mathbb{E}_{\mathcal{N}(\mathbf{w};\mathbf{m},V)}[f(\mathbf{w})] = \mathbb{E}_{\mathcal{N}(\boldsymbol{\nu};\mathbf{0},\mathbb{I})}[f(\mathbf{m} + L\boldsymbol{\nu})]$$

Now differentiating:

$$\begin{aligned}
\nabla_{\mathbf{m}}\mathbb{E}_{\mathcal{N}(\mathbf{w};\mathbf{m},V)}[f(\mathbf{w})] &= \mathbb{E}_{\mathcal{N}(\boldsymbol{\nu};\mathbf{0},\mathbb{I})}[\nabla_{\mathbf{m}} f(\mathbf{m} + L\boldsymbol{\nu})] \\
&\approx \nabla_{\mathbf{m}} f(\mathbf{m} + L\boldsymbol{\nu}) \\
\nabla_{L}\mathbb{E}_{\mathcal{N}(\mathbf{w};\mathbf{m},V)}[f(\mathbf{w})] &= \mathbb{E}_{\mathcal{N}(\boldsymbol{\nu};\mathbf{0},\mathbb{I})}[\nabla_{L} f(\mathbf{m} + L\boldsymbol{\nu})] \\
&\approx [\nabla_{L} f(\mathbf{w})]\boldsymbol{\nu}^{\mathrm{T}}
\end{aligned}$$

14 Gaussian Mixture Models

This is a clustering algorithm. We have K classes and each datapoint has a class: $z^{(n)} = \{1, \ldots, K\}$. These are unobserved and drawn from a prior. $z^{(n)} \sim \pi$. The points come from the Gaussian mixtures:

$$P(\mathbf{x}^{(n)} \mid z^{n} = k, \theta) = \mathcal{N}(\mathbf{x}^{(n)}; \boldsymbol{\mu}^{(k)}, \Sigma^{(k)})$$

where $\theta = \{\pi, \{\boldsymbol{\mu}^{(k)}, \Sigma^{(k)}\}\}$. We can fit it via maximum likelihood by maximizing:

$$\log P(\mathcal{D} \mid \theta) = \sum_{n} \log P(\mathbf{x}^{(n)} \mid \theta)$$

where:

$$\begin{aligned}
P(\mathbf{x}^{(n)} \mid \theta) &= \sum_{k} P(\mathbf{x}^{(n)}, z^{(n)} = k \mid \theta) \\
&= \sum_{k} P(\mathbf{x}^{(n)} \mid z^{(n)} = k, \theta) P(z^{(n)} = k \mid \theta) \\
&= \sum_{k} \pi_k \mathcal{N}(\mathbf{x}^{(n)}; \boldsymbol{\mu}^{(k)}, \Sigma)
\end{aligned}$$

We can't solve this analytically, but we can use gradient-based optimizers. We decompose $\Sigma = LL^{\mathrm{T}}$ (Cholesky) and set:

$$L = \begin{cases} \log L_{i,j} & i = j \\ L_{i,j} & i \neq j \end{cases}$$

The $\boldsymbol{\pi}$ vector must represent probabilities. We can use a softmax transformation.

14.1 Expectation Maximization (EM)

This consists of two steps, the **E** and **M** step:

E-step:

$$r_k^{(n)} = P(z^{(n)} = k \mid \mathbf{x}^{(n)}, \theta) = \frac{\pi_k \mathcal{N}(\mathbf{x}^{(n)}; \boldsymbol{\mu}^{(k)}, \Sigma^{(k)})}{\sum_l \pi_l \mathcal{N}(\mathbf{x}^{(n)}; \boldsymbol{\mu}^{(l)}, \Sigma^{(l)})} \tag{32}$$

M-step:

$$\begin{aligned} r_k = \sum_n r_k^{(n)} \pi_k &= \frac{r_k}{N} \\ \boldsymbol{\mu}^{(k)} &= \frac{1}{r_k} \sum_n r_k^{(n)} \mathbf{x}^{(n)} \\ \Sigma^{(k)} &= \frac{1}{r_k} \sum_n r_k^{(n)} \mathbf{x}^{(n)} \mathbf{x}^{(n)\mathrm{T}} - \boldsymbol{\mu}^{(k)} \boldsymbol{\mu}^{(k)\mathrm{T}} \end{aligned}$$

References

[1] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning Internal Representations by Error Propagation*. ICS-8506. Sept. 1985. URL: http://www.dtic.mil/docs/citations/ADA164453 (visited on 01/20/2018).

[2] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. "Deep Sparse Rectifier Neural Networks". In: *PMLR*. Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics. June 14, 2011, pp. 315–323. URL: http://proceedings.mlr.press/v15/glorot11a.html (visited on 01/15/2018).

[3] Lutz Prechelt. "Early Stopping — But When?" In: *Neural Networks: Tricks of the Trade*. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2012, pp. 53–67. ISBN: 978-3-642-35288-1 978-3-642-35289-8. DOI: 10.1007/978-3-642-35289-8_5. URL: https://link.springer.com/chapter/10.1007/978-3-642-35289-8_5 (visited on 01/20/2018).

[4] Tijmen Tieleman and Geoffrey E. Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural Networks for Machine Learning* 4.2 (2012). URL: https://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf.

[5] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *arXiv:1412.6980 [cs]* (Dec. 22, 2014). arXiv: 1412.6980. URL: http://arxiv.org/abs/1412.6980 (visited on 10/25/2017).

[6] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html (visited on 10/25/2017).

[7] Sergey Ioffe and Christian Szegedy. "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift". In: *PMLR*. International Conference on Machine Learning. June 1, 2015, pp. 448–456. URL: http://proceedings.mlr.press/v37/ioffe15.html (visited on 10/25/2017).

[8] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. URL: http://www.deeplearningbook.org.