MACHINE LEARNING: DEEP LEARNING

Eric Nalisnick
Johns Hopkins University

Last Update:

February 2, 2025

Contents

1 Supervised Learning with Univariate Linear Models

The first topic we will discuss is predictive modeling using linear models—that is, models that are linear in their parameters. This will provide the building blocks with need to eventually stack these 'shallow' models into the 'deep' models that give this course its title.

1.1 Predictive Modeling

Consider the task of *predictive modeling.* Imagine that we are creating a system that, given a medical image, can predict if the patient has a particular disease, e.g. pneumonia. Such a system will be used by bringing patients into the clinic to perform the imaging, and then once the image is taken, the image will be passed to some sort of predictive model, that will generate the prediction that the radiologist will consider to inform their diagnosis. Let $\mathcal{X}$ denote a feature space, which in the example above, is the space of all valid medical images. You can think of this as a matrix in which entries are the pixel values, intensities, or some other property of the image. Let $\mathcal{Y}$ denote the label / response space, which in the above setting is a discrete encoding of the potential diseases. Our goal is to design some model $\hat{\mathrm{y}} = f(\mathbf{x})$ that takes features $\mathbf{x} \in \mathcal{X}$ as input and outputs an accurate prediction $\hat{y} \in \mathcal{Y}$.

Data Generating Process Ideally, we want the above model $f(\mathbf{x})$ to match the true underlying process that generated the data. In the medical imaging example, this means that $f(\mathbf{x})$ would faithfully capture whatever is the underlying medical process that results in a person, with that given image, having the biological conditions that present as their true clinical diagnosis. Mathematically, we can say that the world generates these diseases according to a distribution $\mathbb{P}(\mathrm{y}|\mathbf{x})$, and thus the goal of predictive modeling is to have $f(\mathbf{x}) = \mathbb{P}(\mathrm{y}|\mathbf{x})$. Although, in practice, we are often satisfied with a close approximation.

Training Data As we will see below, we will construct $f(\mathbf{x})$ in a data-driven way. That is, instead of just hand-engineering rules or some other function for $f(\mathbf{x})$, we will *learn* a good predictive model from *data* that represents or encodes our problem of interest. Ideally, we would like to know and work with $\mathbb{P}(\mathrm{y}|\mathbf{x})$ directly, but this is usually never the case in practice. And if we did have access to $\mathbb{P}(\mathrm{y}|\mathbf{x})$, why then would we need to train a model $f(\mathbf{x})$? In practice, we usually just have samples from $\mathbb{P}(\mathrm{y}|\mathbf{x})$, i.e. $\mathrm{y} \sim \mathbb{P}(\mathrm{y}|\mathbf{x})$. For the features $\mathbf{x}$, we will also assume we have samples from another underlying generative process $\mathbf{x} \sim \mathbb{P}(\mathbf{x})$. One could try to model $\mathbb{P}(\mathbf{x})$ in addition to $\mathbb{P}(\mathrm{y}|\mathbf{x})$; this is usually called *generative modeling*, a topic we will get into later in the course. We will assume that, for purposes of training data, we are able to collect N samples of feature-label pairs, making our N-element training set $\mathcal{D} = \{(\boldsymbol{x}_n, y_n)\}_{n=1}^{N}$.

1.2 Univariate Linear Model for Real-Valued Responses

We will now get into our first (or many) concrete instantiations of $f(\mathbf{x})$, and we will start with a (seemingly) simple function: the line, with one slope parameter. Assume for the time being that the features $\mathrm{x} \in \mathbb{R}$ and $\mathrm{y} \in \mathbb{R}$ are both real-valued, unconstrained scalar variables. We will define the *univariate linear model* as $f(\mathrm{x}; \mathrm{w}) \triangleq \mathrm{w} \cdot \mathrm{x}$, where x is a scalar feature value and w is a scalar parameter that we wish to learn from data. To pick apart the notation, $f(\mathrm{x}; \mathrm{w})$ means that we have a function of the features x and the function is determined by parameters w. This model encodes a very simple predictive relationship: the prediction $\hat{y}$ is proportional or inversely proportional to the feature value x.

Loss Function Given an N-sample training set $\mathcal{D}$ and the linear model $f(\mathrm{x}; \mathrm{w})$, the next step is to fit the model to the data. An intuitive way to do this is to define a *loss function* that quantifies how far off the model's predictions are from the observed data. While we will later given a complete recipe for deriving loss functions, one natural choice for real-valued, unconstrained data is the squared loss: $\ell\left(f; x, y\right) = \left(f(\mathrm{x}) - y\right)^2$. Clearly, this will be zero when $f(\mathrm{x}) = y$ and grow quadratically as $f(x)$ makes worse and worse predictions. Also notice that this loss doesn't care if the predictions under or over estimate y, which could be inappropriate for some applications. For example, in the American game show *The Price is Right*, contestants had to guess the sale price of items, and if they overestimated the price, they instantly lost. If you were building an AI agent to play The Price is Right, you would certainly want to train it with a loss function that treats over- and under- estimates differently. Now that we have devised a loss for one data point, we can compute the loss over the full training set by summing the losses for each data point:

$$\ell(\mathrm{w}; \mathcal{D}) \;=\; \frac{1}{N}\sum_{n=1}^{N} \ell(\mathrm{w}; x_n, y_n) \;=\; \frac{1}{N}\sum_{n=1}^{N} \left(f(x_n; \mathrm{w}) - y_n\right)^2 \;=\; \frac{1}{N}\sum_{n=1}^{N} \left(\mathrm{w} \cdot x_n - y_n\right)^2 . \quad (1)$$

Note that these loss functions are a function of *the model*, with the data treated as a constant, because we want to assess how well the model fits the data and not vice versa.

Optimizing a Loss Function Now that we have defined a loss function, we want to use it to find the best setting of the parameter w. This boils down to the following optimization problem:

$$\begin{aligned} w^* &= \arg\min_{\mathrm{w}} \; \ell(\mathrm{w}; \mathcal{D}) \\ &= \arg\min_{\mathrm{w}} \; \frac{1}{N}\sum_{n=1}^{N} \left(f(x_n; \mathrm{w}) - y_n\right)^2 \\ &= \arg\min_{\mathrm{w}} \; \frac{1}{N}\sum_{n=1}^{N} \left(\mathrm{w} \cdot x_n - y_n\right)^2 . \end{aligned} \quad (2)$$

Thus, w^* will be the parameter that minimizes the squared distance between the model predictions and the training responses y. It is unlikely the value of the loss will be exactly zero when computed using $f(x_n; w^*)$, so when we speak of 'minimizing the loss,' it is constrained by the best training performance achievable under the fixed model class—which in this case, is the univariate linear model. The loss function might be able to be driven to exactly zero if we were to choose a different model, especially one that can represent more flexible functions than a line.

Now how should we find the exact form of w^*. Fortunately, for linear models, we can do this exactly and in 'closed form,' meaning that we can get an explicit equation for w^*. This will not be the case for most of the course, and we'll often have to resort to approximate, numerical techniques. Yet, in all cases, we will reply upon tools from calculus.

Recall that the points at which a derivative equals zero represent the *critical points* of a function, meaning that that point can be a maxima, minima, or saddle point. For this linear model with the squared loss, fortunately there is just one (non-trivial) critical point and it represents the global minimum. While a proper course on optimization would go into the details of validating this claim, we will mostly ignore these details since deep learning methodologies often need to reply upon so many approximations that such proofs are not that informative of practice.

Moving on to the mechanics of taking the derivative of the loss with respect to the model parameter, we have:

$$\begin{aligned}
\frac{d}{d\mathrm{w}}\ell(\mathrm{w};\mathcal{D}) \ &= \ \frac{d}{d\mathrm{w}}\left[\frac{1}{N}\sum_{n=1}^{N}(\mathrm{w}\cdot x_n - y_n)^2\right] \\
&= \ \frac{1}{N}\sum_{n=1}^{N}\frac{d}{d\mathrm{w}}\left[(\mathrm{w}\cdot x_n - y_n)^2\right] \\
&= \ \frac{1}{N}\sum_{n=1}^{N}2\cdot(\mathrm{w}\cdot x_n - y_n)\cdot\frac{d}{d\mathrm{w}}\left[\mathrm{w}\cdot x_n - y_n\right] \\
&= \ \frac{1}{N}\sum_{n=1}^{N}2\cdot(\mathrm{w}\cdot x_n - y_n)\cdot x_n \\
&= \ \frac{2}{N}\left\{\left(\sum_{n=1}^{N}\mathrm{w}\cdot x_n^2\right) - \left(\sum_{n=1}^{N}y_n\cdot x_n\right)\right\}.
\end{aligned} \tag{3}$$

Now we can find w^* by setting the derivative to zero and solving for w:

$$\begin{aligned}
0 \ = \ \frac{d}{d\mathrm{w}}\ell(\mathrm{w};\mathcal{D}) \ &= \ \frac{2}{N}\left\{\left(\sum_{n=1}^{N}\mathrm{w}\cdot x_n^2\right) - \left(\sum_{n=1}^{N}y_n\cdot x_n\right)\right\} \\
\implies 0 \ &= \ \left(\sum_{n=1}^{N}\mathrm{w}\cdot x_n^2\right) - \left(\sum_{n=1}^{N}y_n\cdot x_n\right) \\
\implies \sum_{n=1}^{N}\mathrm{w}\cdot x_n^2 \ &= \ \sum_{n=1}^{N}y_n\cdot x_n \\
\implies \mathrm{w} \ &= \ \frac{\sum_{n=1}^{N}y_n\cdot x_n}{\sum_{n=1}^{N}x_n^2} \ \triangleq \ w^*.
\end{aligned} \tag{4}$$

We have finally arrived at the ‘trained’ version of our model: computing $\sum_{n=1}^{N}y_n \cdot x_n / \sum_{n=1}^{N}x_n^2$ will give the value that we should plug in for the optimal parameter w^*.

Vectorized Version The *Graphics processing unit* (GPU) and linear algebra libraries of a modern computers make *vectorized* implementations much faster—i.e. writing your computations as vector or matrix products will make your code much faster than using for-loops. We can do this for the simple linear model above as follows. Firstly, regarding the data, we can write the collection of N features as $\boldsymbol{x} = [x_1, \ldots x_N]^T$, and similarly, the responses as $\boldsymbol{y} = [y_1, \ldots y_N]^T$. Now the vectorized form of the loss function is:

$$\ell(\mathrm{w};\mathcal{D}) \ = \ \frac{1}{N} \ || \, \mathrm{w}\cdot\boldsymbol{x} - \boldsymbol{y} \, ||_2^2 \tag{5}$$

where $||\cdot||_2^2$ is the squared (Euclidean) two norm. Following the same derivation as above but keeping the vector notation, the optimal setting of the weights can then be written in vectorized form as: $w^* = \left(\boldsymbol{y}^T\boldsymbol{x}\right) / \left(\boldsymbol{x}^T\boldsymbol{x}\right)$.

1.3 Maximum Likelihood Estimation: A General Recipe

While sensible, the above procedure we used for finding w^* could still seem arbitrary and unsound. For example, recalling that the goal of predictive modeling is to capture $\mathbb{P}(\mathrm{y}|\mathbf{x})$, how does what we did relate to $\mathbb{P}(\mathrm{y}|\mathbf{x})$? Moreover, are they other choices than the squared loss function? We will now give a general procedure for deriving optimization objectives known as *maximum likelihood estimation*.

Statistical Divergences Yet before introducing maximum likelihood estimation, we need to visit the concept of a statistical *divergence*. A divergence is like a loss function but applied to probability distributions. The most commonly employed divergence is the *Kullback–Leibler divergence* (KLD):

$$\mathbb{KLD}[p(\mathrm{z})||q(\mathrm{z})] \ \triangleq\ \mathbb{E}_{p(\mathrm{z})}\left[\log\frac{p(\mathrm{z})}{q(\mathrm{z})}\right] \ =\ \int_{\mathrm{z}} p(\mathrm{z})\left(\log\frac{p(\mathrm{z})}{q(\mathrm{z})}\right)d\mathrm{z},$$

where z is the random variable of interest, and we want to compare two distributions over z: $p(\mathrm{z})$ vs $q(\mathrm{z})$. The KLD is an information theoretic quantity that represents the number of bits lost when $q(\mathrm{z})$ is used to approximate $p(\mathrm{z})$. This means that the KLD is *not* symmetric: $\mathbb{KLD}[p(\mathrm{z})||q(\mathrm{z})]$ does not necessarily equal $\mathbb{KLD}[q(\mathrm{z})||p(\mathrm{z})]$, thus making the order of the arguments important. However, no matter the order of the arguments, the KLD will be exactly zero when $p(\mathrm{z}) = q(\mathrm{z})$:

$$\mathbb{KLD}[p(\mathrm{z})||p(\mathrm{z})] \ =\ \mathbb{E}_{p(\mathrm{z})}\left[\log\frac{p(\mathrm{z})}{p(\mathrm{z})}\right] \ =\ \mathbb{E}_{p(\mathrm{z})}\left[\log 1\right] \ =\ \log 1 \ =\ 0.$$

There are other divergences, such as the (squared) Hellinger divergence:

$$\mathcal{H}^2[p(\mathrm{z})||q(\mathrm{z})] \ \triangleq\ 1 - \int_{\mathrm{z}} \sqrt{p(\mathrm{z})\cdot q(\mathrm{z})}\ d\mathrm{z}.$$

The Hellinger divergence is symmetric, but unfortunately, it is less commonly employed due to it having an integral that is usually more difficult to evaluate. Both the Hellinger and KLD are members of the family of f-divergences.

Models as Probability Distributions Previously, we defined the model just as a generic function $f(\mathrm{x})$. Now we will be more particular interpreting $f(\mathrm{x})$, embedding it within a distribution function. This will, firstly, allow us to give a probabilistic interpretation to the model itself, unlocking operations such as marginalization, sampling, etc. Secondly, it will allow us to apply a statistical divergence to the model, directly quantifying the gap between the model and the true generative process $\mathbb{P}(\mathrm{y}|\mathrm{x})$. To do this, we will pick a probability distribution $p(\mathrm{y};\theta)$, where y still denotes the label and θ denotes the parameter(s). For example, for the normal distribution $\theta = \{\mu, \sigma\}$, the mean and the standard deviation respectively. Since in our running example $\mathrm{y} \in \mathbb{R}$, technically we can pick any distribution with support over all the real numbers—e.g. normal, Laplace, student-t, etc.—but each comes with its own probabilistic assumptions. For example, the student-t distribution has heavier tails than the normal distribution, meaning that building a model with the student-t assumes that we will see more outlying points.

We will consider the general construction again in a later section. For now, let's assume that we choose $p(\mathrm{y};\theta)$ to be a normal distribution. Now taking our linear model $f(\mathrm{x}) = \mathrm{w}\cdot\mathrm{x}$, we will use this model to parameterize just the mean μ:

$$\begin{aligned} p\left(\mathrm{y}; f(\mathrm{x})\right) &\triangleq \text{Normal}\left(\mathrm{y}; \mu = f(\mathrm{x}), \sigma\right) \\ &= \frac{1}{\sigma\sqrt{2\pi}} \ \exp\left\{-\frac{(\mu - \mathrm{y})^2}{2\sigma^2}\right\} \\ &= \frac{1}{\sigma\sqrt{2\pi}} \ \exp\left\{-\frac{(f(\mathrm{x}) - \mathrm{y})^2}{2\sigma^2}\right\} \\ &= \frac{1}{\sigma\sqrt{2\pi}} \ \exp\left\{-\frac{(\mathrm{w}\cdot\mathrm{x} - \mathrm{y})^2}{2\sigma^2}\right\}. \end{aligned} \tag{6}$$

The parameter σ will also have to be set somehow. We could set it with the same function, tying the mean and standard deviation, e.g. $\sigma = \exp\{f(\mathrm{x})\}$, where the $\exp\{\cdot\}$ ensures that the standard deviation is positive. However, this would mean that as the mean increases, so does the standard deviation, which is an assumption that would be inappropriate for many applications. Alternatively, we could set the standard deviation via a second linear model: $\sigma = \exp\{f'(\mathrm{x})\} = \exp\{\mathrm{u}\cdot\mathrm{x}\}$, where $\mathrm{u} \in \mathbb{R}$ is another parameter that defines this second linear model. In the statistics literature, regression models that have a σ that is constant w.r.t. x are called *homoskedastic*. If σ varies with x, then the model is called *heteroskedastic*. Both of the cases above, where $\sigma = \exp\{f(\mathrm{x})\}$ or $\sigma = \exp\{f'(\mathrm{x})\}$, are heteroskedastic. In this course, for simplicity, we will often assume the models are homoskedastic.

Divergence as an Optimization Objective We now have the pieces in place to derive the maximum likelihood estimation procedure—the standard procedure we will use to train models throughout the course. Maximum likelihood estimation means that we seek to minimize the KLD between the true generative distribution and our probabilistic predictive model:

$$\begin{aligned} \mathbb{KLD}\left[\ \mathbb{P}(\mathrm{y}|\mathrm{x})\ ||\ p(\mathrm{y}; f(\mathrm{x}))\ \right] &= \mathbb{E}_{\mathbb{P}(\mathrm{y}|\mathrm{x})}\left[\log \frac{\mathbb{P}(\mathrm{y}|\mathrm{x})}{p(\mathrm{y}; f(\mathrm{x}))}\right] \\ &= \underbrace{\mathbb{E}_{\mathbb{P}(\mathrm{y}|\mathrm{x})}\left[\log \mathbb{P}(\mathrm{y}|\mathrm{x})\right]}_{-\mathbb{H}[\mathbb{P}(\mathrm{y}|\mathrm{x})]} - \mathbb{E}_{\mathbb{P}(\mathrm{y}|\mathrm{x})}\left[\log p(\mathrm{y}; f(\mathrm{x}))\right] \\ &= \mathbb{E}_{\mathbb{P}(\mathrm{y}|\mathrm{x})}\left[-\log p(\mathrm{y}; f(\mathrm{x}))\right] \quad - \quad \underbrace{\mathbb{H}\left[\mathbb{P}(\mathrm{y}|\mathrm{x})\right]}_{\text{constant w.r.t. } p(\mathrm{y}; f(\mathrm{x}))} \end{aligned} \tag{7}$$

where $\mathbb{H}[p(\mathrm{x})] = \int_{\mathrm{x}} p(\mathrm{x})(-\log p(\mathrm{x}))d\mathrm{x}$ denotes the differential entropy of the distribution $p(\mathrm{x})$. $\mathbb{H}\left[\mathbb{P}(\mathrm{y}|\mathrm{x})\right]$ denotes the entropy of the true generating process $\mathbb{P}(\mathrm{y}|\mathrm{x})$, and thus it does not involve the model. This matters because we will eventually optimize this KLD w.r.t. $f(\mathrm{x})$, and in turn, terms that are not a function of $f(\mathrm{x})$ 'fall out of' the optimization problem. We can see this explicitly when we take the derivative w.r.t. the model parameters, since $\frac{d}{d\mathrm{w}}\mathbb{H}\left[\mathbb{P}(\mathrm{y}|\mathrm{x})\right] = 0$.

Yet, notice that Equation 7 involves a particular value of the features x. Or in other words, both the model and true distribution are conditioned on a particular feature value, and we are evaluating the KLD only at those features. Of course, in the training data, we have multiple feature observations. This means that there is another distribution to be concerned with: the distribution that generates the features, $\mathbb{P}(\mathrm{x})$. We do not model this distribution directly. Rather, we will incorporate it into the maximum likelihood formulation by adding an outer expectation over $\mathbb{P}(\mathrm{x})$:

$$\mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{KLD}\left[\ \mathbb{P}(\mathrm{y}|\mathrm{x})\ ||\ p(\mathrm{y}; f(\mathrm{x}))\ \right] \ = \ \mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{E}_{\mathbb{P}(\mathrm{y}|\mathrm{x})}\left[-\log p(\mathrm{y}; f(\mathrm{x}))\right] - \mathbb{E}_{\mathbb{P}(\mathrm{x})}\left[\mathbb{H}\left[\mathbb{P}(\mathrm{y}|\mathrm{x})\right]\right]$$

Putting everything together and taking the model parameter to again be w, we have the complete maximum likelihood optimization problem:

$$\begin{aligned} w^* &= \operatorname*{arg\,min}_{\mathrm{w}}\ \mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{KLD}\left[\ \mathbb{P}(\mathrm{y}|\mathrm{x})\ ||\ p(\mathrm{y}; f(\mathrm{x};\mathrm{w}))\ \right] \\ &= \operatorname*{arg\,min}_{\mathrm{w}}\ \mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{E}_{\mathbb{P}(\mathrm{y}|\mathrm{x})}\left[-\log p(\mathrm{y}; f(\mathrm{x};\mathrm{w}))\right]\ -\ \mathbb{E}_{\mathbb{P}(\mathrm{x})}\left[\mathbb{H}\left[\mathbb{P}(\mathrm{y}|\mathrm{x})\right]\right] \\ &= \operatorname*{arg\,min}_{\mathrm{w}}\ \mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{E}_{\mathbb{P}(\mathrm{y}|\mathrm{x})}\left[-\log p(\mathrm{y}; f(\mathrm{x};\mathrm{w}))\right] \end{aligned} \tag{8}$$

where, again, the entropy term drops out because it does not involve the model and in turn, the parameter w that we are optimizing.

Monte Carlo Approximation There is one remaining obstacle to solving the optimization problem in Equation 8. It requires taking the expectation w.r.t. the true generative process, $\mathbb{P}(\mathrm{y}, \mathrm{x})$. As stated above, we do not have access to this distribution except through the samples that constitute our training data: $\mathrm{y} \sim \mathbb{P}(\mathrm{y}|\mathrm{x})$ and $\mathrm{x} \sim \mathbb{P}(\mathrm{x})$. Fortunately, this allows us to compute what's called a *Monte Carlo* (MC) approximation of the expectation; for a generic distribution $P(\mathrm{x})$ and a function of the random variable $\phi(\mathrm{x})$, this is:

$$\mathbb{E}_{P(\mathrm{x})}\left[\phi(\mathrm{x})\right] \ \approx\ \frac{1}{S}\sum_{s=1}^{S}\phi(x_s), \quad x_s \sim P(\mathrm{x}),$$

where $S \in \mathbb{N}^+$ is the number of samples. While this approximation is valid for any number of samples (above zero), the approximation becomes better and better as $S \to \infty$, becoming exact only asymptotically. Applying the MC expectation to Equation 8, we have:

$$\begin{aligned} w^* &= \operatorname*{arg\,min}_{\mathrm{w}}\ \mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{E}_{\mathbb{P}(\mathrm{y}|\mathrm{x})}\left[-\log p(\mathrm{y}; f(\mathrm{x};\mathrm{w}))\right] \\ &\approx \operatorname*{arg\,min}_{\mathrm{w}}\ \frac{1}{N}\sum_{n=1}^{N} -\log p\left(y_n; f(x_n;\mathrm{w})\right) \end{aligned} \tag{9}$$

where the sum is over the training samples $\{(x_n, y_n)\}_{n=1}^{N}$. It follows that, the larger training set we have, the better we should be approximating our true optimization target, $\mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{KLD}\left[\ \mathbb{P}(\mathrm{y}|\mathrm{x})\ ||\ p(\mathrm{y}; f(\mathrm{x};\mathrm{w}))\ \right]$.

Deriving the Squared Loss Function We will now work through an end-to-end derivation and eventually arrive at the same loss function used in Equation 2. Keeping with the same setup, we assume $f(\mathrm{x};\mathrm{w}) = \mathrm{w}\cdot\mathrm{x}$ and $p(\mathrm{y}; f(\mathrm{x};w)) = \mathrm{N}(\mathrm{y}; \mu = f(\mathrm{x};w), \sigma = 1)$, where the normal distribution's variance is fixed at one (i.e. the homoskedastic assumption). The final optimization problem is then:

$$\begin{aligned}
w^* &= \arg\min_{\mathrm{w}} \; \mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{KLD}\left[\; \mathbb{P}(\mathrm{y}|\mathrm{x}) \;||\; p(\mathrm{y}; f(\mathrm{x};\mathrm{w}))\;\right] \\
&= \arg\min_{\mathrm{w}} \; \mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{E}_{\mathbb{P}(\mathrm{y}|\mathrm{x})}\left[-\log p(\mathrm{y}; f(\mathrm{x};\mathrm{w})\right] \quad \text{(drop entropy term)} \\
&\approx \arg\min_{\mathrm{w}} \; \frac{1}{N}\sum_{n=1}^{N} -\log p\left(y_n; f(x_n;\mathrm{w})\right) \quad \text{(Monte Carlo approximation)} \\
&= \arg\min_{\mathrm{w}} \; \frac{1}{N}\sum_{n=1}^{N} -\log \mathrm{N}(y_n; \mu_n = f(x_n;\mathrm{w}), \sigma = 1) \\
&= \arg\min_{\mathrm{w}} \; \frac{1}{N}\sum_{n=1}^{N} -\log\left\{\frac{1}{\sigma\sqrt{2\pi}} \;\exp\left\{-\frac{(\mu_n - y_n)^2}{2\sigma^2}\right\}\right\} \\
&= \arg\min_{\mathrm{w}} \; \frac{1}{N}\sum_{n=1}^{N} -\log\left\{\frac{1}{\sqrt{2\pi}} \;\exp\left\{-\frac{(f(x_n;\mathrm{w}) - y_n)^2}{2}\right\}\right\} \qquad (10) \\
&= \arg\min_{\mathrm{w}} \; \frac{1}{N}\sum_{n=1}^{N} -\log\exp\left\{-\frac{(f(x_n;\mathrm{w}) - y_n)^2}{2}\right\} + \log\left\{\sqrt{2\pi}\right\} \\
&= \arg\min_{\mathrm{w}} \; \frac{1}{N}\sum_{n=1}^{N} \frac{1}{2}(f(x_n;\mathrm{w}) - y_n)^2 + \log\left\{\sqrt{2\pi}\right\} \\
&= \arg\min_{\mathrm{w}} \; \frac{1}{2}\frac{1}{N}\sum_{n=1}^{N}(f(x_n;\mathrm{w}) - y_n)^2 \quad \text{(drop } \sqrt{2\pi} \text{ constant)} \\
&= \arg\min_{\mathrm{w}} \; \frac{1}{2}\frac{1}{N}\sum_{n=1}^{N}(\mathrm{w}\cdot x_n - y_n)^2
\end{aligned}$$

where the $\log\left\{\sqrt{2\pi}\right\}$ term is dropped because it does not depend on the parameters w. Thus, the maximum likelihood perspective gives us a more principled justification for use of the squared loss function as well as making its probabilistic assumptions explicit. If we were optimizing a heteroskedastic model w.r.t. the parameters controlling the variance, then this term would not drop from the optimization problem. The last line above is nearly the same as the final form of Equation 2 except for the constant 1/2. Yet the presence of this constant does not change the solution, as the *maximum likelihood estimator* (MLE) for w^* is still the same as derived in Equation 4. In fact, the derivative becomes a bit simpler as the 1/2 cancels the factor of 2 that is introduced when applying the power rule.

1.4 Generalized Linear Models

Above we discussed how a predictive model can be thought of as a probability distribution, with a specific function determining the mean variable. Specifically, we chose $p(\mathrm{y}; f(\mathrm{x})) = \mathrm{N}(\mathrm{y}; \mu = f(\mathrm{x}), \sigma)$. This flexible, modular framework allows us to construct models that meet our constraints or assumptions for the problem at hand. We call a linear model of the following form a *generalized linear model* (GLM):

$$\mathbb{E}_p[\mathrm{y}|\mathrm{x}] \;=\; g^{-1}\left(f(\mathrm{x};\mathrm{w})\right) \;=\; g^{-1}\left(\mathrm{w}\cdot\mathrm{x}\right) \qquad (11)$$

where $f(\mathrm{x};\mathrm{w}) = \mathrm{w}\cdot\mathrm{x}$, the linear model, and $g(\cdot)$ is known as the *link function*. In turn, $g^{-1}(\cdot)$ is called the *inverse link function*. Firstly, notice that in our running example of $p(\mathrm{y}; f(\mathrm{x})) = \mathrm{N}(\mathrm{y}; \mu = f(\mathrm{x}), \sigma)$, there is no g function since $\mu = f(\mathrm{x})$. Or to put it precisely, the link function is simply the identity function. This works in this case since the valid values of μ match the range of $f(\mathrm{x})$, namely all real numbers $\mathbb{R}$. But this will not be the case for all models of interest. Consider binary responses $\mathrm{y} \in \{0, 1\}$. A common distribution with support over binary vales is the *Bernoulli* distribution: $p(\mathrm{y}; \pi) = \pi^{\mathrm{y}} \cdot (1-\pi)^{1-\mathrm{y}}$, where $\pi \in [0,1]$ is the mean parameter. We cannot set $\pi = \mathrm{w}\cdot\mathrm{x}$ in this case since this would mean $\pi \in (-\infty, \infty)$, breaking the definition of the Bernoulli distribution. To fix this issue, we need to choose a g^{-1} function that transforms the output of $f(\mathrm{x})$ onto $(0, 1)$. We will examine one popular implementation for the Bernoulli case in the next section. If $\mathrm{y} \in \mathbb{N}_{\geq 0}$, the non-negative natural numbers, the Poisson distribution is a commonly employed distribution with this support: $p(\mathrm{y}; \lambda) = \lambda^{\mathrm{y}} e^{-\lambda}/\mathrm{y}!$, where $\lambda \in (0, \infty)$. In this case, it is common to choose the inverse link g^{-1} as the exponential function: $\lambda = \exp\{f(\mathrm{x})\}$.

Maximum Likelihood Derivative For the GLM under the maximum likelihood objective, we can write a general form for the chain rule derivative since it will always have the same four components:

$$\frac{d}{d\mathrm{w}}\mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{KLD}\left[\ \mathbb{P}(\mathrm{y}|\mathrm{x})\ ||\ p\left(\mathrm{y}; g^{-1}\circ f(\mathrm{x};\mathrm{w})\right)\ \right]\ = \\ \left(\frac{d}{dp}\mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{KLD}\left[\ \mathbb{P}(\mathrm{y}|\mathrm{x})\ ||\ p\left(\mathrm{y}\right)\right)\ \right]\right)\left(\frac{d}{dg^{-1}}p\left(\mathrm{y}; g^{-1}\right)\right)\left(\frac{d}{df}g^{-1}(f)\right)\left(\frac{d}{d\mathrm{w}}f(\mathrm{x};\mathrm{w})\right).$$

While for linear models $\frac{d}{d\mathrm{w}}f(\mathrm{x};\mathrm{w})$ is usually easy to evaluate, this will be the most computationally intensive term for deep learning models.

1.5 Univariate Logistic Regression for Binary Labels

We will now examine a particular GLM mentioned in the preceding section—namely, a model for binary labels known as *logistic regression*. Assuming $\mathrm{y} \in \{0, 1\}$, we can define the following predictive model:

$$\begin{aligned} p\left(\mathrm{y}; f(\mathrm{x};\mathrm{w})\right) &= \text{Bernoulli}\left(\mathrm{y}; \pi = g^{-1}(f(\mathrm{x};\mathrm{w}))\right) \\ &= \pi^{\mathrm{y}}\cdot(1-\pi)^{1-\mathrm{y}} \\ &= g^{-1}(f(\mathrm{x};\mathrm{w}))^{\mathrm{y}}\cdot(1-g^{-1}(f(\mathrm{x};\mathrm{w})))^{1-\mathrm{y}} \\ &= g^{-1}(\mathrm{w}\cdot\mathrm{x})^{\mathrm{y}}\cdot(1-g^{-1}(\mathrm{w}\cdot\mathrm{x}))^{1-\mathrm{y}}. \end{aligned} \tag{12}$$

Now we need to select the form of g^{-1} such that $g^{-1}: \mathbb{R} \mapsto (0, 1)$. The most common choice of g^{-1} is the *logistic function*, which is where the name *logistic regression* originates:

$$\texttt{logistic}\,(\mathrm{z})\ \triangleq\ \frac{1}{1+\exp\{-\mathrm{z}\}}.$$

We call this the *logistic function* because it is the cummulative distribution function of the standard logistic distribution ('standard' meaning location 0, scale 1). Plugging this into the above expressions gives us our final form of logistic regression:

$$\begin{aligned} p\left(\mathrm{y}; f(\mathrm{x};\mathrm{w})\right) &= \operatorname{Bernoulli}\left(\mathrm{y}; \pi = \texttt{logistic}(f(\mathrm{x};\mathrm{w}))\right) \\ &= \texttt{logistic}(f(\mathrm{x};\mathrm{w}))^{\mathrm{y}} \cdot (1 - \texttt{logistic}(f(\mathrm{x};\mathrm{w})))^{1-\mathrm{y}} \\ &= \left(\frac{1}{1+\exp\{-f(\mathrm{x};\mathrm{w})\}}\right)^{\mathrm{y}} \cdot \left(1 - \frac{1}{1+\exp\{-f(\mathrm{x};\mathrm{w})\}}\right)^{1-\mathrm{y}} \\ &= \left(\frac{1}{1+\exp\{-\mathrm{w}\cdot\mathrm{x}\}}\right)^{\mathrm{y}} \cdot \left(1 - \frac{1}{1+\exp\{-\mathrm{w}\cdot\mathrm{x}\}}\right)^{1-\mathrm{y}}. \end{aligned} \tag{13}$$

Plugging this model into the maximum likelihood objective, we have:

$$\begin{aligned} w^{*} &= \arg\min_{\mathrm{w}} \ \mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{KLD}\left[\ \mathbb{P}(\mathrm{y}|\mathrm{x}) \ || \ \operatorname{Bernoulli}\left(\mathrm{y}; \pi = \texttt{logistic}(f(\mathrm{x};\mathrm{w}))\right)\ \right] \\ &= \arg\min_{\mathrm{w}} \ \mathbb{E}_{\mathbb{P}(\mathrm{x})}\mathbb{E}_{\mathbb{P}(\mathrm{y}|\mathrm{x})}\left[-\log \operatorname{Bernoulli}\left(\mathrm{y}; \pi = \texttt{logistic}(f(\mathrm{x};\mathrm{w}))\right)\right] \quad \text{(drop entropy term)} \\ &\approx \arg\min_{\mathrm{w}} \ \frac{1}{N}\sum_{n=1}^{N} -\log \operatorname{Bernoulli}\left(y_n; \pi = \texttt{logistic}(f(x_n;\mathrm{w}))\right) \quad \text{(Monte Carlo approximation)} \\ &= \arg\min_{\mathrm{w}} \ \frac{1}{N}\sum_{n=1}^{N} -\log\left\{\texttt{logistic}(f(x_n;\mathrm{w}))^{y_n} \cdot (1 - \texttt{logistic}(f(x_n;\mathrm{w})))^{1-y_n}\right\} \\ &= \arg\min_{\mathrm{w}} \ \frac{1}{N}\sum_{n=1}^{N} -y_n \log \texttt{logistic}(f(x_n;\mathrm{w})) \ - \ (1-y_n)\log\left(1 - \texttt{logistic}(f(x_n;\mathrm{w}))\right) \\ &= \arg\min_{\mathrm{w}} \ \frac{1}{N}\sum_{n=1}^{N} -y_n \log \texttt{logistic}(\mathrm{w}\cdot x_n) \ - \ (1-y_n)\log\left(1 - \texttt{logistic}(\mathrm{w}\cdot x_n)\right) \\ &= \arg\min_{\mathrm{w}} \ \frac{1}{N}\sum_{n=1}^{N} -y_n \log\left(\frac{1}{1+\exp\{-\mathrm{w}\cdot\mathrm{x}\}}\right) \ - \ (1-y_n)\log\left(1 - \left(\frac{1}{1+\exp\{-\mathrm{w}\cdot\mathrm{x}\}}\right)\right). \end{aligned}$$

This does not have as intuitive a form as the squared error loss function we examined earlier, but this expression is known as the *binary cross-entropy loss*. The 'cross-entropy' part is a bit of mis-characterization of this particular loss. Cross-entropy is defined as $\mathbb{H}[p(\mathrm{x}), q(\mathrm{x})] = -\mathbb{E}_{p(\mathrm{x})}\left[\log q(\mathrm{x})\right]$ for two distributions $p(\mathrm{x})$ and $q(\mathrm{x})$, which is what we have in step #2 of the above derivation but also in step #2 of Equation 10. Thus, squared error could also be called a 'cross-entropy loss', but when people say that, they usually are assuming the labels take on binary or (as we'll see later) categorical values.

1.6 Gradient Descent

If you try to take the derivative of the cross-entropy loss above, set it to zero, and solve for w, you will find that you cannot isolate w to one side of the equation, meaning that the optimization problem has no 'closed-form' solution. Instead we need to find a numerical solution that will only approximate the true optimum. We will use a procedure known as 'gradient descent', a.k.a. 'steepest descent'. The intuition is that we'll start with an initial guess at the value of w and slowly walk down the loss surface, following the direction of steepest descent according to the derivative at our current point. For a generic function

$\phi(\mathrm{z})$ that we wish to minimize, we can apply gradient descent by iterating the equation:

$$z_{t+1} \;=\; z_t \;-\; \alpha \cdot \frac{d}{dz_t}\phi(z_t)$$

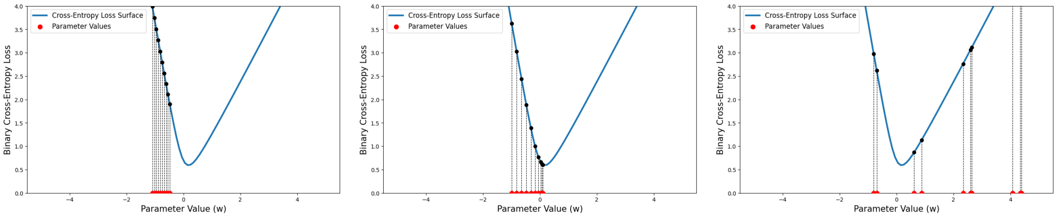
where $\alpha > 0$ is the *learning rate* or *step size* that controls how aggressively we move down the path of steepest descent at each iteration. It is common to set α to be large for the early iterations and then slowly decay it, taking smaller and smaller step sizes, when the procedure gets close to a minimum. We can know if we've approximately converged by tracking the derivative $\frac{d}{dz_t}\phi(z_t)$ since it should be exactly zero at the minimum (or any other critical point—a topic for later in the course).

Applying Gradient Descent to Logistic Regression We will now apply gradient descent to the logistic regression model. The first step will be to take the derivative of the cross-entropy loss w.r.t. the parameter w. When doing this, we will use the fact that the derivative of the logistic function is: $d/dz\, \texttt{logistic}\,(\mathrm{z}) = \texttt{logistic}\,(\mathrm{z}) \cdot (1 - \texttt{logistic}\,(\mathrm{z}))$. I'll leave showing this to the reader as an exercise.

$$\begin{aligned}
\frac{d}{d\mathrm{w}}\ell(\mathrm{w};\mathcal{D}) \;&=\; \frac{d}{d\mathrm{w}}\left[\frac{1}{N}\sum_{n=1}^{N} -\log \mathrm{Bernoulli}\left(y_n; \pi = \texttt{logistic}(f(x_n;\mathrm{w}))\right)\right] \\
&=\; \frac{1}{N}\sum_{n=1}^{N} -y_n \frac{d}{d\mathrm{w}}\left[\log \texttt{logistic}(\mathrm{w}\cdot x_n)\right] \;-\; (1-y_n)\frac{d}{d\mathrm{w}}\left[\log\left(1 - \texttt{logistic}(\mathrm{w}\cdot x_n)\right]\right) \\
&=\; \frac{1}{N}\sum_{n=1}^{N} -y_n \cdot \frac{\texttt{logistic}(\mathrm{w}\cdot x_n)\cdot(1-\texttt{logistic}(\mathrm{w}\cdot x_n))}{\texttt{logistic}(\mathrm{w}\cdot x_n)} \cdot x_n \\
&\qquad\qquad - (1-y_n)\frac{-\texttt{logistic}(\mathrm{w}\cdot x_n)\cdot(1-\texttt{logistic}(\mathrm{w}\cdot x_n))}{1-\texttt{logistic}(\mathrm{w}\cdot x_n)} \cdot x_n \\
&=\; \frac{1}{N}\sum_{n=1}^{N} -y_n \cdot (1-\texttt{logistic}(\mathrm{w}\cdot x_n)) \cdot x_n \;+\; (1-y_n)\texttt{logistic}(\mathrm{w}\cdot x_n)\cdot x_n \\
&=\; \frac{1}{N}\sum_{n=1}^{N} (\texttt{logistic}(\mathrm{w}\cdot x_n) - y_n)\cdot x_n \\
&=\; \frac{1}{N}\sum_{n=1}^{N} (\mathbb{E}_p[\mathrm{y}|x_n] - y_n)\cdot x_n \quad \text{(using the definition of a GLM).}
\end{aligned}$$

Using the definition of a GLM reveals that the derivative takes on a very sensible form: the difference between the expected value of y and its actual value, y_n, scaled by the feature value x_n. When $(\mathbb{E}_p[\mathrm{y}|x_n] - y_n) \approx 0$, the model is a accurate predictor and thus the contribution of this data point is negligible. If the total derivative is zero, then the model is accurately predicting all training points. Plugging this derivative into the gradient descent equation, we have:

$$\begin{aligned}
w_{t+1} \;&=\; w_t \;-\; \alpha \cdot \frac{d}{dw_t}\ell(w_t;\mathcal{D}) \\
&=\; w_t \;-\; \alpha\left[\frac{1}{N}\sum_{n=1}^{N}(\texttt{logistic}(w_t\cdot x_n) - y_n)\cdot x_n\right].
\end{aligned} \tag{14}$$



(a) Step size is too small.

(b) Good step size.

(c) Step size is too high.

Figure 1: *Gradient descent for logistic regression, with varying step sizes.* For each run, gradient descent is run for 10 steps with a fixed step size. In (a), the step size is too small ($\alpha = .02$); in (b), the step size is suitable ($\alpha = .05$); in (c), the step size is too large ($\alpha = 1.5$).

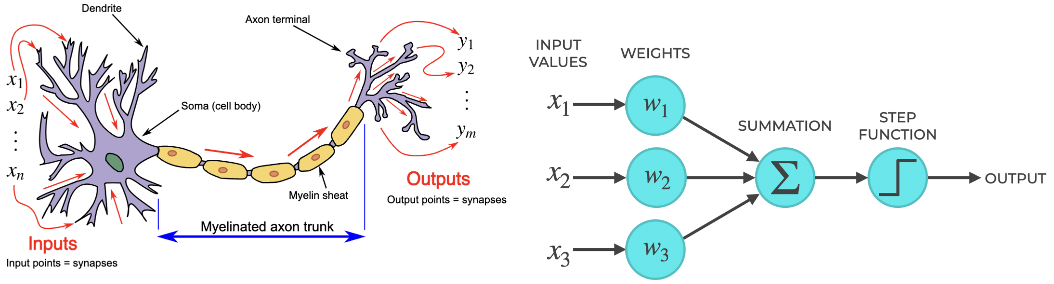
We would implement the above equation numerically by guessing an initial value, w_0, making a prediction using w_0 (i.e. evaluate the logistic output) for all training points, compute the derivative by combining the predictions, the labels and features, as shown above, and lastly updating the weight to arrive at w_1. That process is then repeated for a maximum number of rounds or until the derivative is sufficiently close to zero (e.g.1×10^{-4}).

Figure 1 shows a simulation for a one-parameter logistic regression model. The (binary) cross-entropy loss surface is visualized by the blue line. The intermediate parameter estimates (w_t) are shown along the x-axis in red, and the dotted line connects them to the loss value that they resulted in. Subfigure 1a shows when the step size is too small, as the maximum number of iterations is reached before the minimum is bound. Subfigure 1 shows the other extreme: the step size is too big and so the minimum cannot be found. Instead, the optimizer 'jumps' over the minimum and flys off to the right-hand side of the plot. Subfigure 1b shows when the step size is properly set, as the optimizer gracefully descends to the minimum.

1.7 Models of Artificial Neurons & the Perceptron

So far, the narrative of these notes takes a purely statistical perspective. However, deep learning also has roots in artificial intelligence, which at times has taken inspiration from biological intelligence. The first major step towards formulating a computational model of biological neurons was taken in 1943 by McCulloch and Pitts. A biological neuron can be, very roughly, thought of as a model that takes an input signal (dendrite), performs computation (soma), and passes the output to other connected neurons (axon to synapse to other neuron's dendrite). A visualization is shown in Figure 2a. McCulloch and Pitts proposed a model whose input can be either *excitatory* or *inhibitory*. Its output can then be either *quiet* or *firing*, depending on if the the number of excitatory inputs is equal to or greater than some fixed threshold.

In 1957, Frank Rosenblatt famously implemented a very similar model on an IBM 704 computer, calling it *the perceptron*, and demonstrated that it could 'learn' to classify simple patterns. This demonstration received wide media attention; the New York Times had an article with the headline: "Electronic 'Brain' Teaches Itself." The perceptron can be defined



(a) Biological Neuron

(b) The Perceptron

Figure 2: *Biological vs Artifical Neurons.*

for an input $\mathrm{x} \in \mathbb{R}$ and parameters $\mathrm{w} \in \mathbb{R}$ and $\mathrm{b} \in \mathbb{R}$ as:

$$\hat{\mathrm{y}} \;=\; \psi\left(\mathrm{w} \cdot \mathrm{x} + \mathrm{b}\right), \text{ where } \; \psi(\mathrm{z}) = -1 + 2 \cdot \mathbb{I}\left[\mathrm{z} > 0\right],$$

with $\mathbb{I}[\cdot]$ denoting the indicator function that evaluates to 1 if its argument is true and 0 otherwise. In turn, $\psi(\mathrm{z}) = +1$ if its input is greater than zero and -1 otherwise. Thus $\hat{\mathrm{y}}$ is the prediction, and it should model binary data represented as $\mathrm{y} \in \{-1, +1\}$. The similarity to the McCulloch and Pitts model is that the input x is modulated by a weight w, and if $\mathrm{w} \cdot \mathrm{x} > -\mathrm{b}$, then $\hat{\mathrm{y}}$ 'fires,' having a value of one. Thus $-\mathrm{b}$ serves as the activation threshold proposed by McCulloch and Pitts; though it was discrete in the original model and real-valued here. A visualization of the perceptron is shown in Figure 2b.

The perceptron learning algorithm is as follows for an N-sized data set $\mathcal{D} = \{x_n, y_n\}_{n=1}^{N}$, $\mathrm{x} \in \mathbb{R}$, $\mathrm{y} \in \{-1, +1\}$. The algorithm starts by randomly initializing w and b; denote these values as w_0 and b_0. Then for every time step t, a training pair is selected at random (x_r, y_r), the perceptron's prediction is computed as $\hat{y}_r = \psi(w_0 \cdot x_r + b_0)$, and this prediction is checked for correctness, $\hat{y}_r = y_r$. If the prediction is correct, the process repeats for another feature-label pair. If the prediction is incorrect, then the following update is performed to obtain w_{t+1} and b_{t+1}:

$$w_{t+1} = w_t + y_r \cdot x_r, \quad b_{t+1} = b_t + y_r.$$

This process is repeated for either a maximum number of iterations or until all points in the training data have their label correctly predicted.

The update rule for the perceptron learning algorithm looks a bit mysterious, and it is certainly not clear, at least at a first glance, why iterating them amounts to 'learning.' Yet we can examine some parallels to the steepest descent equations for logistic regression to gain intuition. Let's consider the logistic regression update rule given in Section 1.6. But we will make two simplifying assumptions: the update will be computed for just one data point and with a step size of $\alpha = 1$:

$$w_{t+1} \;=\; w_t - (\mathbb{E}[\mathrm{y}|x] - y) \cdot x \;=\; w_t + (y - \mathbb{E}[\mathrm{y}|x]) \cdot x.$$

By comparing this equation to the perceptron's weight update, the only difference is the factor that is multiplied with the features x: $(y - \mathbb{E}[\mathrm{y}|x])$ for logistic regression (for $\mathrm{y} \in \{0, 1\}$) vs y for the perceptron. Now let's assume logistic regression's output can only take on the extreme values of 0 and 1, i.e. $\mathbb{E}[\mathrm{y}|x] \in \{0, 1\}$. Thus when $\mathbb{E}[\mathrm{y}|x] = y$, their difference will be zero and there will be no update to the parameters. When $\mathbb{E}[\mathrm{y}|x] \neq y$, the difference will either be $+1$ when $y = 1$ or -1 when $y = 0$. Notice that the perceptron learning rule is then recovered *exactly* due to the label support being defined as $\{-1, +1\}$. The update for the second parameter, b, can be recovered by thinking of the features as being a two dimensional vector: $\mathbf{x} = [\mathrm{x}_0, \mathrm{x}_1]^T$ with x_0 always being set to one. Then given the parameter vector $\theta = [\mathrm{b}, \mathrm{w}]^T$, we have $\boldsymbol{\theta}^T \mathbf{w} = \mathrm{b} \cdot \mathrm{x}_0 + \mathrm{w} \cdot \mathrm{x}_1 = \mathrm{w} \cdot \mathrm{x}_1 + \mathrm{b}$. In general, we can think of the perceptron as performing a 'hard' update since its predictions are either completely correct or incorrect. On the other hand, logistic regression's learning rule is more precise since it has a more granular notion of how far off its predictions (as represented by $\mathbb{E}[\mathrm{y}|\mathrm{x}]$) are from the true label.

While the invention of the perceptron generated much excitement, this excitement was short-lived. In 1969, Marvin Minsky and Seymour Papert published a book entitled, *Perceptrons: An Introduction to Computational Geometry.* The authors demonstrated the perceptron's limitations—namely, that it could only learn linear decision functions. Thus the perceptron could not model a simple logical function like XOR (i.e. *exclusive or*). This dissolved enthusiasm for artificial neural networks as a potential path towards artificial intelligence, and symbolic approaches began to be favored instead. This period generally saw the decline in funding for artificial intelligence in the 1970's and 1980's. When similar approaches started to gain traction again in the 1990's, they were often re-branded as 'machine learning' instead of 'artificial intelligence.'

2 Supervised Learning with Multiple Linear Regression

2.1 Multiple Features and Feature Expansions

2.2 Revisiting Gradient Descent with Multivariate Derivatives

2.3 Multiple Output Dimensions

3 Model Evaluation, Model Selection, and Regularization

4 Feedforward Neural Networks

4.1 Adaptive Basis Functions

5 Convolutional Neural Networks

6 Recurrent Neural Networks

7 Transformers

8 Deep Generative Models

9 Uncertainty Estimation

Bibliography