MACHINE LEARNING: DEEP LEARNING

Eric Nalisnick
Johns Hopkins University

Last Update:

April 9, 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: $\ell(f; x, y) = (f(\mathrm{x}) - y)^2$. 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 $p(\mathrm{z}) = q(\mathrm{z})$:

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

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:

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

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:

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

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:

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

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):

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

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:

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

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:

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

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:

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

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

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

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:

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

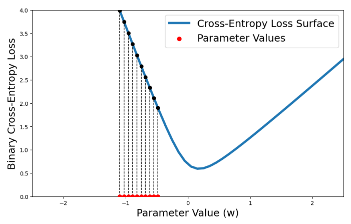
where $\alpha > 0$ is the *learning rate* or *step size* that controls how aggressively we move down the path of steepest descent at each iteration. It is common to set α to be large for the early iterations and then slowly decay it, taking smaller and smaller step sizes, when the procedure gets close 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.

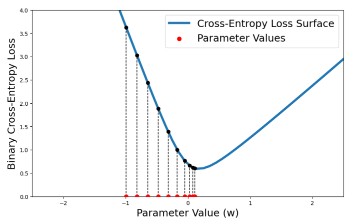
$$\begin{aligned}
\frac{d}{d\mathrm{w}}\ell(\mathrm{w};\mathcal{D}) \;&=\; \frac{d}{d\mathrm{w}}\left[\frac{1}{N}\sum_{n=1}^{N} -\log \mathrm{Bernoulli}\left(y_n; \pi = \texttt{logistic}(f(x_n;\mathrm{w}))\right)\right] \\
&=\; \frac{1}{N}\sum_{n=1}^{N} -y_n \frac{d}{d\mathrm{w}}\left[\log \texttt{logistic}(\mathrm{w}\cdot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

Using the definition of logistic regression as 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 *confident* and *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 with *maximal* confidence. Plugging this derivative into the gradient descent equation, we have:

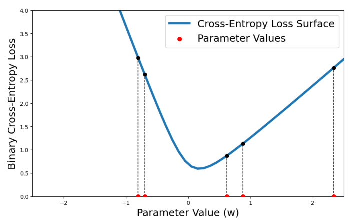
$$\begin{aligned}
w_{t+1} \;&=\; w_t \;-\; \alpha \cdot \frac{d}{dw_t}\ell(w_t;\mathcal{D}) \\
&=\; w_t \;-\; \alpha\left[\frac{1}{N}\sum_{n=1}^{N}(\texttt{logistic}(w_t\cdot x_n) - y_n)\cdot x_n\right].
\end{aligned} \tag{14}$$



(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 ($\alpha = 1.5$).

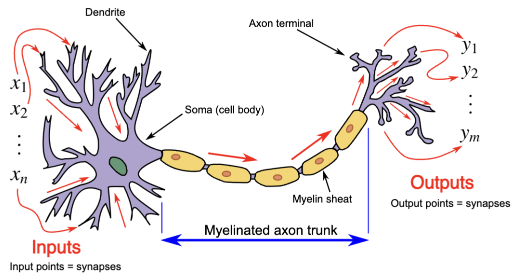
We would implement the above equation numerically by guessing an initial value, w_0, making a prediction using w_0 (i.e. evaluate the logistic output) for all training points, compute the derivative by combining the predictions, the labels and features, as shown above, and lastly updating the weight 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 found. 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.

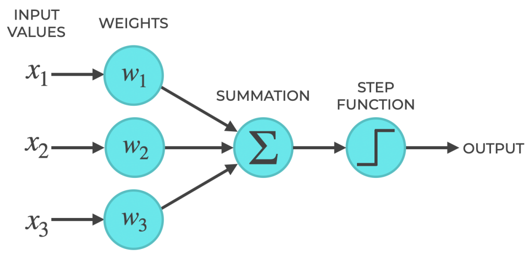
1.7 Models of Artificial Neurons & the Perceptron

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 Artificial 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 $\boldsymbol{\theta}^T\mathbf{w} = \mathrm{b} \cdot \mathrm{x}_0 + \mathrm{w} \cdot \mathrm{x}_1 = \mathrm{w} \cdot \mathrm{x}_1 + \mathrm{b}$. We will discuss this representation in more detail in the next section. 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 neurons 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

In the previous section, we have built up our foundation for specifying statistical predictive models and their loss functions. We also covered how to train them using steepest descent. Now we will look to expand these models such that they can have multiple input features and multiple output features. The underlying principles will primarily be the same, but now as parameters and outputs will be vector-valued, we will need to use vector calculus for deriving the corresponding learning updates.

2.1 Multiple Features and Feature Expansions

We will now consider regression models for which the input features are multi-dimensional. In turn, we will now need to specify multiple weight parameters—one for each input dimension. Thus, our (generalized) regression models will be specified as:

$$\mathbb{E}\left[\mathrm{y}|\mathbf{x}\right] \;\triangleq\; g^{-1}\left(\mathbf{w}^T\mathbf{x}\right), \quad \text{where} \;\; \mathbf{w} = \begin{bmatrix} \mathrm{w}_1 \\ \vdots \\ \mathrm{w}_D \end{bmatrix} \in \mathbb{R}^D \;\; \text{and} \;\; \mathbf{x} = \begin{bmatrix} \mathrm{x}_1 \\ \vdots \\ \mathrm{x}_D \end{bmatrix} \in \mathbb{R}^D, \qquad (15)$$

and where $g^{-1}(\cdot)$ is still the (inverse) link function and y is the scalar label. The most obvious case where this formulation is applicable is for applications that have multiple feature observations. For example, y might be a binary indicator representing the presence of heart disease, and $\mathbf{x}$ could be various health features such as age, blood pressure, weight, cortisol levels, etc. We will now consider two other examples.

Vector Representation of Offset Parameter One simple but common use for the vector representation, even when there is only input feature, is to encode the offset a.k.a. bias parameter. In the perceptron model, this was b. The regression models we considered so far (except the perceptron) had just one parameter, meaning that we can interpret it as modeling the slope of a line. However, we might also want to model the bias / offset from the x-axis. Or in logistic regression, the offset determines where the decision boundary of $p(\mathrm{y}|\mathbf{x}) = 0.5$ resides. We can treat the offset parameter like any other weight by appending a constant value of one to every feature vector. Thus we have:

$$\mathbb{E}\left[\mathrm{y}|\mathrm{x}\right] \;\triangleq\; g^{-1}\left(\mathrm{w}_1 \cdot \mathrm{x} + \mathrm{w}_0\right) \;=\; g^{-1}\left(\mathbf{w}^T\mathbf{x}\right), \quad \text{where } \mathbf{w} = \begin{bmatrix}\mathrm{w}_0 \\ \mathrm{w}_1\end{bmatrix} \in \mathbb{R}^D \text{ and } \mathbf{x} = \begin{bmatrix}1 \\ \mathrm{x}\end{bmatrix} \in \mathbb{R}^D,$$

where w_0 is the offset parameter and w_1 is the weight that represents the slope parameter, like before. While this may seem to be just a notational trick, this allows for cleaner programmatic implementations since the offset parameter does not have to be treated differently than the other weight parameters. In this course, we will usually assume offset parameters are treated in this way, by appending a constant 1 to the feature vectors.

Polynomial Basis Expansion Linear models may seem limited, at first glance, to encoding only straight lines. Yet, one trick to make linear models more expressive while not adding much computational overhead is to use a *feature expansion.* Let's assume we have a scalar data $\mathrm{x} \in \mathbb{R}$ and $\mathrm{y} \in \mathbb{R}$. Our previous models for this data used only one parameter and thus encoded the slope of a line through the origin. Yet what if the relationship between the features and labels is more complex than that? One simple trick is to create 'dummy' features by replicating the existing feature. One simple but still very expressive way is to use a K-degree *polynomial basis.* This means that we create a new feature vector by taking x to powers up to degree K:

$$\tilde{\mathbf{x}} \;=\; \left[\mathrm{x}^0\ \mathrm{x}^1\ \mathrm{x}^2\ \ldots\ \mathrm{x}^K\right]^T \;=\; \left[1\ \mathrm{x}\ \mathrm{x}^2\ \ldots\ \mathrm{x}^K\right]^T.$$

Then when we use this expanded feature set in the linear model, the model encodes a K-degree polynomial, with the $K+1$ parameters / weights serving as the coefficients:

$$\mathbb{E}\left[\mathrm{y}|\mathrm{x}\right] \;\triangleq\; g^{-1}\left(\mathbf{w}^T\tilde{\mathbf{x}}\right) \;=\; g^{-1}\left(\sum_{k=0}^{K} \mathrm{w}_k \cdot \mathrm{x}^k\right), \quad \text{where } \mathbf{w} = \begin{bmatrix}\mathrm{w}_0 \\ \vdots \\ \mathrm{w}_K\end{bmatrix} \in \mathbb{R}^{K+1} \text{ and } \tilde{\mathbf{x}} = \begin{bmatrix}1 \\ \vdots \\ \mathrm{x}^K\end{bmatrix} \in \mathbb{R}^{K+1}.$$

Figure 3 shows a simulation in which the true function we are trying to model is a $K = 7$ degree polynomial. The plots show when real-valued regression is run with degrees ranging from one (linear) to fifteen. We see that once the degree of the model surpasses the degree of the true function ($K = 10$), then the model is well-fit to the data.

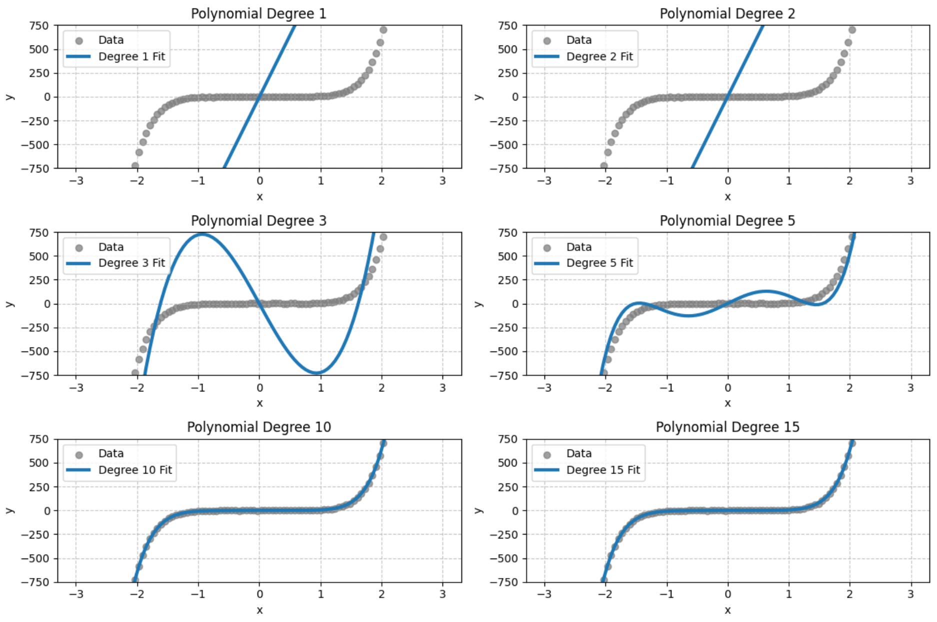


Figure 3: *Linear Regression with Polynomial Basis.* The data represents a 7th-degree polynomial. We see that as the degree of the model's feature basis increases, the model can express richer and richer functions.

2.2 Revisiting Gradient Descent with Multivariate Derivatives

Now that we have seen vector-based models, fitting them will require that we compute the derivative for each parameter. This will require that we adopt tools from multivariate calculus—namely, the gradient operator. Consider an arbitrary differentiable function $f : \mathbb{R}^D \mapsto \mathbb{R}$, meaning that its input is a D-dimensional vector and its output is a scalar. We define the *gradient* of this function to be:

$$\nabla_{\boldsymbol{x}} f(\boldsymbol{x}) \;=\; \left[\frac{df}{d\boldsymbol{x}}\right]^T \;=\; \left[\frac{\partial f(\boldsymbol{x})}{\partial x_1} \cdots \frac{\partial f(\boldsymbol{x})}{\partial x_d} \cdots \frac{\partial f(\boldsymbol{x})}{\partial x_D}\right].$$

Notice that the gradient is *row vector* $\mathbb{R}^{1\times D}$; this will be important when we start to chain together derivatives when differentiating function compositions.

Returning to the steepest descent equations, their multivariate version can be written with the gradient operator as follows:

$$\boldsymbol{w}_{t+1} \;=\; \boldsymbol{w}_t \;-\; \alpha \cdot \left[\nabla_{\boldsymbol{w}_t} \ell\left(\boldsymbol{w}_t; \mathcal{D}\right)\right]^T \tag{16}$$

where α is still the step size (a.k.a. learning rate), $\boldsymbol{w}_t$ is the parameter vector, and $\nabla_{\boldsymbol{w}_t} \ell\left(\boldsymbol{w}_t; \mathcal{D}\right)$ is the gradient of the loss function w.r.t. the parameter vector. As we assume that $\boldsymbol{w}_t$ is a column vector, we need to re-orient the result of the gradient (which is assumed to be a row vector) in order to perform the update via subtraction.

In the univariate case, we thought of steepest descent as forming a tangent line that guides how we descend down the curve. Now we should have a multi-dimensional surface in mind, and the gradient gives us, upon each evaluation, a (hyper)-plane that guides our descent. A demonstration for logistic regression with two parameters is shown in Figure 4.

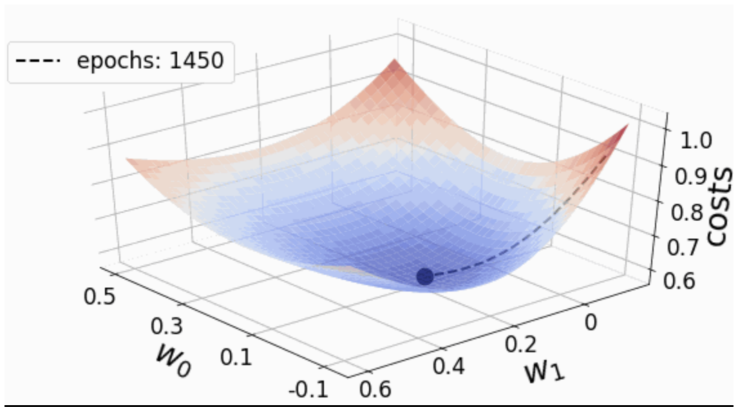


Figure 4: *Optimization Trajectory for Logistic Regression.* This figure shows the path (dashed line) through the optimization surface taken for gradient descent on a logistic regression model. The two parameters are shown on the $x-$ and z-axes and the loss (a.k.a. cost) is shown on the y-axis. Optimization is run for 1450 iterations through the training data (a.k.a. *epochs*).

The two parameters are shown on the $x-$ and z-axes and the cross-entropy loss function (a.k.a. cost function) is shown on the y-axis. The trajectory of the intermediate parameter values is shown by the dashed line, with the final parameter setting denoted by the circle.

Logistic Regression with Multiple Features Lets consider the multi-feature version of the logistic regression model: $\mathbb{E}[\mathrm{y}|\mathbf{x}] = s\left(\mathbf{w}^T\mathbf{x}\right)$ where, as before, $\mathbf{x} \in \mathbb{R}^D$ are the features and $\mathbf{w} \in \mathbb{R}^D$ are the parameters. Yet now the label is binary valued, $\mathrm{y} \in \{0, 1\}$, and $s(\mathrm{z}) = 1/(1 + \exp\{-\mathrm{z}\})$ is the logistic function. This model would be trained by the binary cross-entropy loss function for an N-sized data set:

$$\ell\left(\mathbf{w}; \mathcal{D}\right) \;=\; \frac{1}{N}\sum_{n=1}^{N} -y_n \log\left\{s\left(\mathbf{w}^T\boldsymbol{x}_n\right)\right\} - (1-y_n)\log\left\{1 - s\left(\mathbf{w}^T\boldsymbol{x}_n\right)\right\}.$$

The gradient w.r.t. the weight vector is:

$$\begin{aligned}
&\nabla_{\mathbf{w}}\ \ell\left(\mathbf{w}; \mathcal{D}\right) \\
&= \nabla_{\mathbf{w}}\left[\frac{1}{N}\sum_{n=1}^{N} -y_n \log\left\{s\left(\mathbf{w}^T\boldsymbol{x}_n\right)\right\} - (1-y_n)\log\left\{1 - s\left(\mathbf{w}^T\boldsymbol{x}_n\right)\right\}\right] \\
&= \frac{1}{N}\sum_{n=1}^{N} -y_n \cdot \nabla_{\mathbf{w}}\left[\log\left\{s\left(\mathbf{w}^T\boldsymbol{x}_n\right)\right\}\right] - (1-y_n)\cdot\nabla_{\mathbf{w}}\left[\log\left\{1 - s\left(\mathbf{w}^T\boldsymbol{x}_n\right)\right\}\right] \\
&= \frac{1}{N}\sum_{n=1}^{N} -y_n \cdot \frac{1}{s\left(\mathbf{w}^T\boldsymbol{x}_n\right)}\cdot\nabla_{\mathbf{w}}\left[s\left(\mathbf{w}^T\boldsymbol{x}_n\right)\right] - (1-y_n)\cdot\frac{-1}{1 - s\left(\mathbf{w}^T\boldsymbol{x}_n\right)}\cdot\nabla_{\mathbf{w}}\left[s\left(\mathbf{w}^T\boldsymbol{x}_n\right)\right].
\end{aligned}$$

Focusing on just the gradient of the logistic function, we have:

$$\begin{aligned}
\nabla_{\mathbf{w}}\ s\left(\mathbf{w}^T \boldsymbol{x}_n\right) &= s\left(\mathbf{w}^T \boldsymbol{x}_n\right) \cdot \left(1 - s\left(\mathbf{w}^T \boldsymbol{x}_n\right)\right) \cdot \nabla_{\mathbf{w}}\left[\mathbf{w}^T \boldsymbol{x}_n\right] \\
&= s\left(\mathbf{w}^T \boldsymbol{x}_n\right) \cdot \left(1 - s\left(\mathbf{w}^T \boldsymbol{x}_n\right)\right) \cdot \left[\frac{\partial\ \mathbf{w}^T \boldsymbol{x}_n}{\partial\ w_1} \cdots \frac{\partial\ \mathbf{w}^T \boldsymbol{x}_n}{\partial\ w_D}\right] \\
&= s\left(\mathbf{w}^T \boldsymbol{x}_n\right) \cdot \left(1 - s\left(\mathbf{w}^T \boldsymbol{x}_n\right)\right) \cdot \left[x_{n,1} \ldots x_{n,D}\right] \\
&= s\left(\mathbf{w}^T \boldsymbol{x}_n\right) \cdot \left(1 - s\left(\mathbf{w}^T \boldsymbol{x}_n\right)\right) \cdot \boldsymbol{x}_n^T.
\end{aligned}$$

Plugging this back into the expression above, we have:

$$\begin{aligned}
&= \frac{1}{N} \sum_{n=1}^{N} -y_n \cdot \frac{1}{s\left(\mathbf{w}^T \boldsymbol{x}_n\right)} \cdot s\left(\mathbf{w}^T \boldsymbol{x}_n\right) \cdot \left(1 - s\left(\mathbf{w}^T \boldsymbol{x}_n\right)\right) \cdot \boldsymbol{x}_n^T \\
&\qquad - (1 - y_n) \cdot \frac{-1}{1 - s\left(\mathbf{w}^T \boldsymbol{x}_n\right)} \cdot s\left(\mathbf{w}^T \boldsymbol{x}_n\right) \cdot \left(1 - s\left(\mathbf{w}^T \boldsymbol{x}_n\right)\right) \cdot \boldsymbol{x}_n^T \\
&= \frac{1}{N} \sum_{n=1}^{N} \left[-y_n \cdot \left(1 - s\left(\mathbf{w}^T \boldsymbol{x}_n\right)\right) - (y_n - 1) \cdot s\left(\mathbf{w}^T \boldsymbol{x}_n\right)\right] \cdot \boldsymbol{x}_n^T \\
&= \frac{1}{N} \sum_{n=1}^{N} \left[s\left(\mathbf{w}^T \boldsymbol{x}_n\right) - y_n\right] \cdot \boldsymbol{x}_n^T.
\end{aligned}$$

This result would be plugged into Equation 16 and iterated until convergence.

2.3 Multiple Output Dimensions

Just as we considered models with multiple input dimensions, we will also want to consider models with multiple *output* dimensions. This is useful for, for example, predicting the trajectory of an object, as its spatial coordinate in two or three dimensions would need to be output by the model. Multi-dimensional outputs are also crucial for defining predictive models for *categorical* data, as we will see in an example below. In general, the GLM formulation for a K-dimensional label denoted by the *vector* $\mathbf{y}$ is:

$$\begin{aligned}
\mathbb{E}\left[\mathbf{y}|\mathbf{x}\right] &\triangleq g^{-1}\left(\mathbf{W}^T \mathbf{x}\right), \quad \text{where } \mathbf{x} = \begin{bmatrix} \mathrm{x}_1 \\ \vdots \\ \mathrm{x}_D \end{bmatrix} \in \mathbb{R}^D \text{ and} \\
\mathbf{W} &= \left[\mathbf{w}_1 \ldots \mathbf{w}_K\right] = \begin{bmatrix} \mathrm{w}_{1,1} & \cdots & \mathrm{w}_{1,K} \\ \vdots & \ddots & \vdots \\ \mathrm{w}_{D,1} & \cdots & \mathrm{w}_{D,K} \end{bmatrix} \in \mathbb{R}^{D \times K}.
\end{aligned} \tag{17}$$

The two crucial differences needed to arrive at this multi-output formulation are (1) there is now a $(D \times K)$-parameter *matrix* to transform the D-dimensional input features into a K-dimensional output, and (2) the inverse link function is applied to a vector input. It depends on the model formulation whether the link function is applied element-wise or has dependencies across dimensions. Let's examine the case of the latter below.

Example: Categorical Labels One of the most common tasks in machine learning (and therefore in deep learning as well) is *classification*: categorizing a given set of input features into one of K discrete, disjoint groups. We have already seen an example of a binary classifier (i.e. two groups), which was logistic regression, and so this can be thought of as a higher dimensional generalization. The usual way to represent this encoding is via a so-called 'one hot' vector representation, meaning that one element takes on value 1 and the rest are 0. For example, $\mathbf{y}^T = [0,0,1,0]$ means that there are $K = 4$ total categories, and this label is denoting membership in the third category. The natural choice of distribution for data of this type is the *categorical distribution* (a.k.a. the multinoulli):

$$p(\mathbf{y};\boldsymbol{\pi}) \ = \ \text{Categorical}(\mathbf{y};\boldsymbol{\pi}) \ \triangleq \ \prod_{k=1}^{K} \pi_k^{\mathrm{y}_k},$$

where the distribution's parameters are represented by the K-dimensional vector $\boldsymbol{\pi}$, which has the constraints $\pi_k \in [0,1]$ and $\sum_k \pi_k = 1$. We can interpret these parameters as the probability of each class / category, and therefore the probabilities must sum to one to be well-defined.

Now defining a GLM with the categorical distribution, we have:

$$\mathbb{E}\left[\mathbf{y}|\mathbf{x}\right] \ = \ \boldsymbol{\pi} \ = \ g^{-1}\left(\mathbf{W}^T\mathbf{x}\right)$$

where, like with the Bernoulli, the success probabilities of the classes are the distribution's mean. And as above, we will use a $(D \times K)$-dimensional matrix of parameters. The canonical inverse link function for this setting is known as the *softmax* function:

$$\texttt{softmax}_j(\mathbf{z}) \ \triangleq \ \frac{\exp\{z_j\}}{\sum_{k=1}^{K}\exp\{z_k\}}, \quad \text{where} \quad \mathbf{z} \in \mathbb{R}^K \tag{18}$$

and where the subscript j in $\texttt{softmax}_j(\mathbf{z})$ denotes the function's jth output dimension. Summing over all dimensions in the denominator is what enforces the constraints imposed by interpreting the output as probabilities. In general, we have $\texttt{softmax} : \mathbb{R}^K \mapsto \Delta_K$ where Δ_K denotes the K-dimensional *simplex*, the space of all positive K-dimensional vectors that sum to one. Note that the 'softmax' function is mis-named, as it really is performing a soft 'argmax.'

Putting this all together, we can derive the *categorical cross-entropy loss function* as:

$$\begin{aligned}
\boldsymbol{W}^* &= \arg\min_{\mathbf{W}} \ \mathbb{E}_{\mathbb{P}(\mathbf{x})}\mathbb{KLD}\left[\ \mathbb{P}(\mathbf{y}|\mathbf{x}) \ || \ \text{Categorical}\left(\mathbf{y}; \boldsymbol{\pi} = \texttt{softmax}(f(\mathbf{x}; \mathbf{W}))\right)\ \right] \\
&= \arg\min_{\mathbf{W}} \ \mathbb{E}_{\mathbb{P}(\mathbf{x})}\mathbb{E}_{\mathbb{P}(\mathbf{y}|\mathbf{x})}\left[-\log \text{Categorical}\left(\mathbf{y}; \boldsymbol{\pi} = \texttt{softmax}(f(\mathbf{x}; \mathbf{W}))\right)\right] \quad \text{(drop entropy term)} \\
&\approx \arg\min_{\mathbf{W}} \ \frac{1}{N}\sum_{n=1}^{N} -\log \text{Categorical}\left(\boldsymbol{y}_n; \boldsymbol{\pi}_n = \texttt{softmax}(f(\boldsymbol{x}_n; \mathbf{W}))\right) \quad \text{(Monte Carlo approximation)} \\
&= \arg\min_{\mathbf{W}} \ \frac{1}{N}\sum_{n=1}^{N} -\log\left\{\prod_{k=1}^{K} \pi_{n,k}^{y_{n,k}}\right\} \\
&= \arg\min_{\mathbf{W}} \ \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} -y_{n,k} \cdot \log \pi_{n,k} \\
&= \arg\min_{\mathbf{W}} \ \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} -y_{n,k} \cdot \log \texttt{softmax}_k(f(\boldsymbol{x}_n; \mathbf{W})) \\
&= \arg\min_{\mathbf{W}} \ \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} -y_{n,k} \cdot \log \frac{\exp\{\mathbf{w}_k^T \boldsymbol{x}_n\}}{\sum_{j=1}^{K}\exp\{\mathbf{w}_j^T \boldsymbol{x}_n\}} \\
&= \arg\min_{\mathbf{W}} \ \frac{1}{N}\sum_{n=1}^{N}\sum_{k=1}^{K} -y_{n,k} \cdot \left[\mathbf{w}_k^T \boldsymbol{x}_n - \log\left\{\sum_{j=1}^{K}\exp\{\mathbf{w}_j^T \boldsymbol{x}_n\}\right\}\right].
\end{aligned}$$

We will need to use gradient descent to perform this optimization. We'll start by deriving the gradient for the one particular column of $\mathbf{W}$ that corresponds to the dimension for which $\mathrm{y}_k = 1$; call it $\mathbf{w}_k$:

$$\begin{aligned}
\nabla_{\mathbf{w}_k} \ \ell(\mathbf{W}; \mathcal{D}) &= \frac{1}{N}\sum_{n=1}^{N} \nabla_{\mathbf{w}_k} \sum_{k=1}^{K} -y_{n,k} \cdot \left[\mathbf{w}_k^T \boldsymbol{x}_n - \log\left\{\sum_{j=1}^{K}\exp\{\mathbf{w}_j^T \boldsymbol{x}_n\}\right\}\right] \\
&= \frac{1}{N}\sum_{n=1}^{N} -\nabla_{\mathbf{w}_k}\left[\mathbf{w}_k^T \boldsymbol{x}_n - \log\left\{\sum_{j=1}^{K}\exp\{\mathbf{w}_j^T \boldsymbol{x}_n\}\right\}\right] \quad (y_{n,k} = 1 \text{ so we can drop it}) \\
&= \frac{1}{N}\sum_{n=1}^{N} -\left[\nabla_{\mathbf{w}_k}\left[\mathbf{w}_k^T \boldsymbol{x}_n\right] - \frac{1}{\sum_{j=1}^{K}\exp\{\mathbf{w}_j^T \boldsymbol{x}_n\}}\nabla_{\mathbf{w}_k}\left[\sum_{j=1}^{K}\exp\{\mathbf{w}_j^T \boldsymbol{x}_n\}\right]\right] \\
&= \frac{1}{N}\sum_{n=1}^{N} -\left[\boldsymbol{x}_n^T - \frac{\exp\{\mathbf{w}_k^T \boldsymbol{x}_n\}}{\sum_{j=1}^{K}\exp\{\mathbf{w}_j^T \boldsymbol{x}_n\}}\nabla_{\mathbf{w}_k}\left[\mathbf{w}_k^T \boldsymbol{x}_n\right]\right] \\
&= \frac{1}{N}\sum_{n=1}^{N} -\left[\boldsymbol{x}_n^T - \pi_{n,k} \cdot \boldsymbol{x}_n^T\right] \\
&= \frac{1}{N}\sum_{n=1}^{N} -(1 - \pi_{n,k}) \cdot \boldsymbol{x}_n^T \\
&= \frac{1}{N}\sum_{n=1}^{N} (\pi_{n,k} - 1) \cdot \boldsymbol{x}_n^T.
\end{aligned}$$

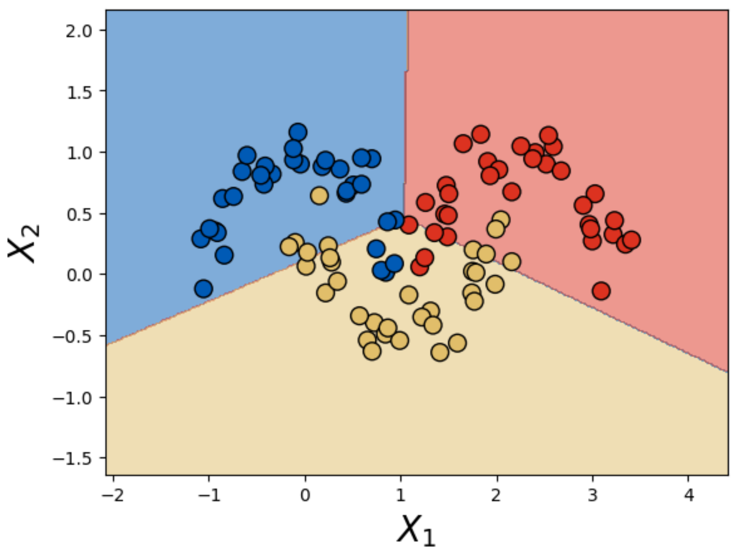


Figure 5: *Example of a Multi-Class Linear Classifier (i.e. Categorical Regression).* The plots shows the feature space; training data points are shown by the scatter plot and colored according to their true class assignment. The background is shaded according to how a 3-class linear classifier would categorize the data points.

Now turn to the dimensions for which $\mathrm{y}_j = 0$; call one of them $\mathbf{w}_i$:

$$\begin{aligned}
\nabla_{\mathbf{w}_i} \ell(\mathbf{W}; \mathcal{D}) &= \frac{1}{N} \sum_{n=1}^{N} \nabla_{\mathbf{w}_i} \sum_{k=1}^{K} -y_{n,k} \cdot \left[\mathbf{w}_k^T \boldsymbol{x}_n - \log \left\{ \sum_{j=1}^{K} \exp\{\mathbf{w}_j^T \boldsymbol{x}_n\} \right\} \right] \\
&= \frac{1}{N} \sum_{n=1}^{N} -\nabla_{\mathbf{w}_i} \left[\mathbf{w}_k^T \boldsymbol{x}_n - \log \left\{ \sum_{j=1}^{K} \exp\{\mathbf{w}_j^T \boldsymbol{x}_n\} \right\} \right] \qquad (\nabla_{\mathbf{w}_i} \mathbf{w}_k^T \boldsymbol{x}_n = 0) \\
&= \frac{1}{N} \sum_{n=1}^{N} \frac{1}{\sum_{j=1}^{K} \exp\{\mathbf{w}_j^T \boldsymbol{x}_n\}} \nabla_{\mathbf{w}_i} \left[\sum_{j=1}^{K} \exp\{\mathbf{w}_j^T \boldsymbol{x}_n\} \right] \\
&= \frac{1}{N} \sum_{n=1}^{N} \frac{\exp\{\mathbf{w}_i^T \boldsymbol{x}_n\}}{\sum_{j=1}^{K} \exp\{\mathbf{w}_j^T \boldsymbol{x}_n\}} \nabla_{\mathbf{w}_i} \left[\mathbf{w}_i^T \boldsymbol{x}_n \right] \\
&= \frac{1}{N} \sum_{n=1}^{N} \pi_{n,i} \cdot \boldsymbol{x}_n^T.
\end{aligned}$$

Lastly, for some nicer book keeping, we can combine the equations to get a unified update rule:

$$\nabla_{\mathbf{w}_j} \ell(\mathcal{D}; \mathbf{W}) = \frac{1}{N} \sum_{n=1}^{N} (\pi_{n,j} - y_{n,j}) \cdot \boldsymbol{x}_n^T, \tag{19}$$

which recovers the first gradient when $y_{n,j} = 1$ and the second when $y_{n,j} = 0$. Finally, the gradient descent update for the jth weight vector is:

$$\boldsymbol{w}_{j,t+1} = \boldsymbol{w}_{j,t} - \alpha \cdot \nabla_{\boldsymbol{w}_{j,t}} \ell(\boldsymbol{W}_t; \mathcal{D}),$$

and we would perform that update for all of the $j \in [1, K]$ weights vectors.

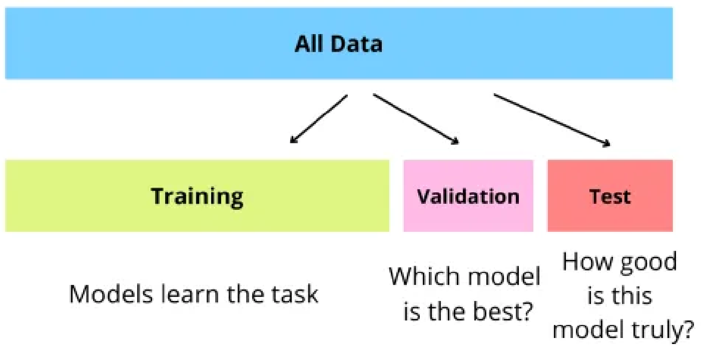


Figure 6: *Splitting the Available Data for Training, Validation, and Testing*

3 Model Evaluation, Model Selection, and Capacity Control

Until now, we have been primarily concerned with *training* predictive models. But there are other important issues that we so far ignored: the related topics of model evaluation, model selection, and regularization. These pertain to the issues of *how well do we expect the model to perform when deployed to the real world?* And, *when we have multiple models, which one should we deploy?*

3.1 Model Evaluation

Recall that the goal of predictive modeling is to approximate the true, data-generating process: $p(\mathrm{y}; f(\mathbf{x})) \approx \mathbb{P}(\mathrm{y}|\mathbf{x})$. Yet we don't have direct access to $\mathbb{P}(\mathrm{y}|\mathbf{x})$; thus, how should we evaluate if our model will work when we deploy it to the real world (e.g. embed it within a self-driving car, integrate it into our web application, etc)? Recall that we only see samples from our generative process, which we call 'data': $\mathcal{D} \sim \mathbb{P}(\mathrm{y}, \mathbf{x})$. If $\mathcal{D}$ is all of the data that we have access to during model development, and we use all of $\mathcal{D}$ to train the model, then assessing the model's performance on $\mathcal{D}$ again isn't ideal. This is like, given some practice problems for which you already know the answers, using those same practice problems to study for the exam. While performance on those known problems tells you something about test-time performance, it does not simulate the actual process of having to answer truly never-before-seen questions. Similarly, we want to test our models by having them make predictions on data points that they have never seen before. Thus, when evaluating models, it's of the utmost importance that we evaluate them on 'held-out' data—that is, data that was not used for training. One simple quantity to compute is the loss function used for training but with the held-out data plugged in for the training data:

$$\ell\left(\mathbf{W}^{*}; \mathcal{D}_{\text{held-out}}\right)$$

where $\mathbf{W}^{*}$ are the parameters that were found by training and $\mathcal{D}_{\text{held-out}}$ is the never-before-seen data.

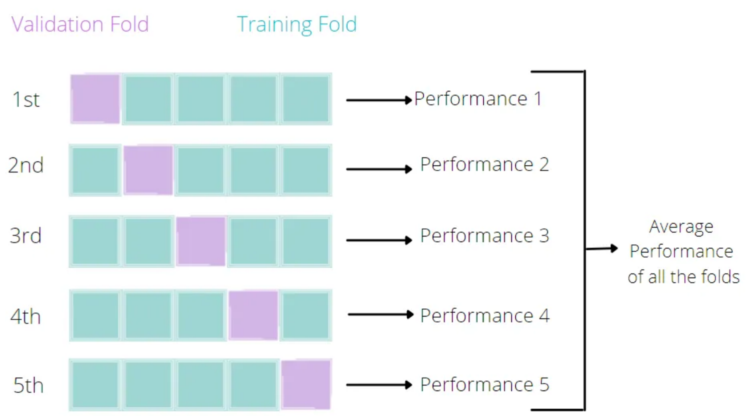


Figure 7: *An Example of 5-Fold Cross Validation.*

3.1.1 Train-Validation-Test Splits

Yet we are often limited in the amount of data that we have and thus must use it sparingly to perform the aforementioned held-out evaluation. One common way to do this is to split the finite data set that we are given into three pieces, with the largest piece used for training ($60\% - 80\%$), the next biggest used for validation ($30\% - 10\%$), and the final part used for testing ($20\% - 10\%$). See Figure 6 for a diagram. As the name implies, the training split is used for training the model. The 'validation' split is used for selecting among multiple models: for example, if you fit multiple models, each with different hyperparameter settings such as step size or polynomial degree. Every time the validation split is used to revise the model and improve the parameter settings, it loses its value. When you believe you have finally selected the best model, you can use the test split to calculate the final performance that is expected on the data that will be seen when the model is deployed and receives data from the real world.

If there is not enough available data to make three sufficiently large subsets, one strategy is to use the training set for both training and validation by creating multiple 'folds.' The procedure is visualized in Figure 7 for 5 folds. The training data is split into five subsets, with four being used for training and one for validation. The subset that is used for validation is varied across all five partitions. Note that this then *requires re-training the model five times!* Thus, while the evaluation procedure is more data efficient, we must pay in the computational cost of re-training the model. The most extreme form of this procedure is *leave-one-out cross validation*, where all but one data points are used for training and the remaining point is used for validation. Thus this procedure is extremely expensive since it requires re-training the model as many times as there are data points. Generally, using more folds gives better estimates of the validation error, but the choice must be balanced with the computational considerations.

3.1.2 Example Evaluation Metrics

Now let's examine some example evaluation metrics for real-valued regression and classification.

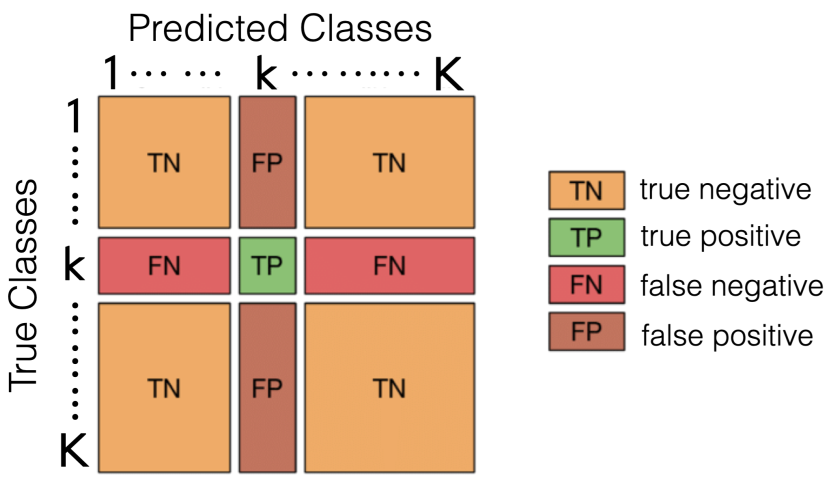


Figure 8: *Confusion Matrix.* The entries of a confusion matrix are the counts of the number of times an instance of the ground-truth class denoted by the row indices is classified as the class denoted by the column indices. Hence, the confusion matrix of a classifier that never makes a mistake on the tested data has zero entries for all off-diagonal elements.

Real-Valued Regression Recall that when training a model for real-valued regression, we usually use the squared error function. When evaluating these models, we usually use a slightly different quantity called the *root-mean-squared error* (RMSE):

$$\texttt{RMSE}\left(\mathbf{w}^*; \mathcal{D}_{\text{held-out}}\right) \ = \ \sqrt{\frac{1}{M}\sum_{m=1}^{M}\left(\mathbb{E}\left[y_m|f\left(\boldsymbol{x}_m;\mathbf{w}^*\right)\right] \ - \ y_m\right)^2}$$

where M is the number of evaluation points and $\mathbb{E}\left[y_m|f\left(\boldsymbol{x}_m;\mathbf{w}^*\right)\right] = (\mathbf{w}^*)^T\mathbf{x}$ for real-valued regression. The reason why the square root operation is added is so that the units of the error are on the same scale as the label's.

Classification, a.k.a. Categorical Regression For classification tasks, we can also use the training loss evaluated (i.e. cross-entropy error) on held-out data to evaluate the model. However, this doesn't fully capture how classifiers are used when they are deployed. The cross-entropy loss will consider the precise value of $\mathbb{E}\left[y_m|f\left(\boldsymbol{x}_m;\mathbf{w}^*\right)\right]$, but in practice, we often need to make a discrete choice. *Which class / category does this instance before too?* If the goal of the classifier is to filter out spam, we need it to make a clear decision whether to let the incoming email go into the inbox or into the junk folder. One metric that directly measure this decision-making is *accuracy*:

$$\texttt{Accuracy}(\mathbf{w}^*; \mathcal{D}_{\text{held-out}}) \ = \ \frac{1}{M}\sum_{m=1}^{M}\mathbb{I}\left[\texttt{round}\left(\mathbb{E}[y_m|f(\boldsymbol{x}_m;\mathbf{w}^*)]\right) = y_m\right],$$

where, if $y_m \in \{0,1\}$, $\texttt{round}\left(\mathbb{E}[y_m|f(\boldsymbol{x}_m;\mathbf{w}^*)]\right)$ simply means that we return 1 if $\mathbb{E}[y_m|f(\boldsymbol{x}_m;\mathbf{w}^*)] \geq 0.5$ and 0 otherwise. If $y_m \in \{1,\ldots,K\}$, then $\texttt{round}\left(\mathbb{E}[y_{m,k}=1|f(\boldsymbol{x}_m;\mathbf{w}^*)]\right) = \arg\max_k \mathbb{E}[y_{m,k} = 1|f(\boldsymbol{x}_m;\mathbf{w}^*)]$. Usually this is implemented by taking the maximum dimension of the probability vector that is produced by the softmax transformation.

Accuracy is an aggregate metric and therefore can hide disparate performance across subclasses. That is, maybe the classifier identifies two out of three classes well but does poorly discriminating the third one. This might result in an accuracy that is indistinguishable from doing moderately well across all classes. If this is the fear, then we can use a *confusion matrix* to better assess per-class performance. See Figure 8 for a visualization. The entries of a confusion matrix are the counts of the number of times an instance of the ground-truth class denoted by the row indices is classified as the class denoted by the column indices. Mathematically, we can write this as:

$$C_{i,j} \;=\; \sum_{m=1}^{M} \mathbb{I}[y_m = i] \cdot \mathbb{I}[\hat{y}_m = j]$$

where y_m is the true class of the mth instance and $\hat{y}_m$ is the predicted class of the mth instance. Thus, the confusion matrix of a classifier that never makes a mistake on the tested data has zero entries for all off-diagonal elements. Then entry $C_{k,k}$ is the count of the number of times an instance of class k was correctly classified as such.

If there is an imbalanced number of classes, then accuracy may not be a good metric. Consider the case of identifying credit card fraud. Fraud only happens in a small minority of cases—say, 1 out of 1000—and therefore a classifier that always predicts 'no fraud' will have an accuracy of about 99.9%. This classifier, of course, would actually be useless because we presumably built it with the motivation of detecting fraud.

Two metrics that are useful when there is class imbalance are *precision* and *recall*. They can easily be defined by considering the columns and rows of the confusion matrix, respectively:

$$\texttt{Precision}_k(\boldsymbol{C}) \;=\; \frac{C_{k,k}}{\sum_{j=1}^{K} C_{j,k}}, \quad \texttt{Recall}_k(\boldsymbol{C}) \;=\; \frac{C_{k,k}}{\sum_{j=1}^{K} C_{k,j}}.$$

The precision for the kth class is the number of times class k was predicted correctly ($C_{k,k}$) divided by the total number of times class k was predicted, i.e. the sum of the elements in the kth column $\left(\sum_{j=1}^{K} C_{j,k}\right)$. On the other hand, *recall* is the fraction of times the classifier successfully identified class k ($C_{k,k}$) out of all of the instances of class k, i.e. the sum of the elements in the kth row $\left(\sum_{j=1}^{K} C_{k,j}\right)$. Returning to the example of detecting credit card fraud, while that classifier would have a high accuracy, it's precision and recall for the 'fraud' class would be zero. If one wishes to simultaneously quantify precision and recall, all within one value, the *F1* score is the harmonic mean of precision and recall:

$$\texttt{F1}_k(\boldsymbol{C}) \;=\; 2 \cdot \frac{\texttt{Precision}_k(\boldsymbol{C}) \cdot \texttt{Recall}_k(\boldsymbol{C})}{\texttt{Precision}_k(\boldsymbol{C}) + \texttt{Recall}_k(\boldsymbol{C})}.$$

The harmonic mean is used so that the smaller values are emphasized in the aggregation. All of the above metrics are defined class-wise, but the 'macro' versions can be computed by averaging the values over all classes.

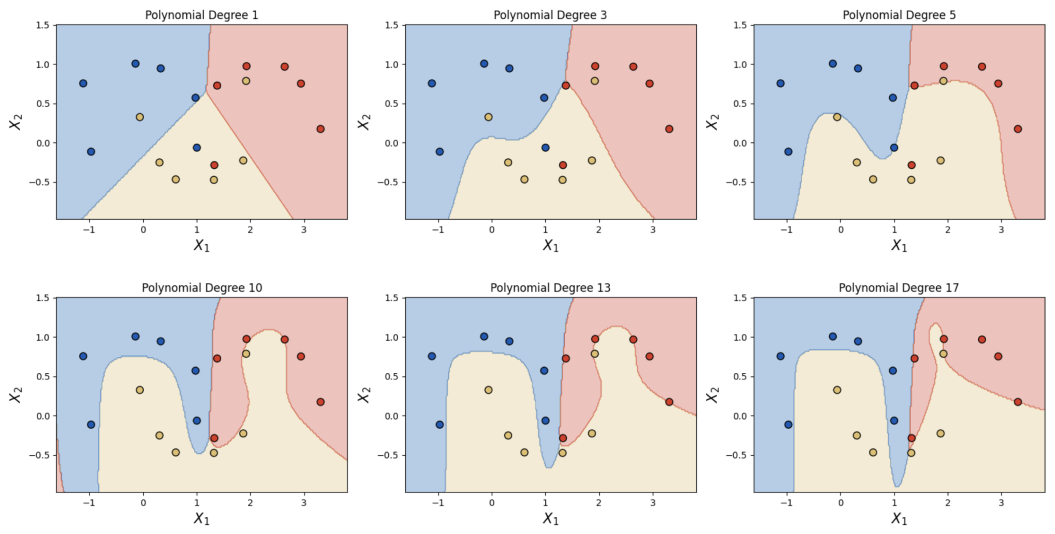


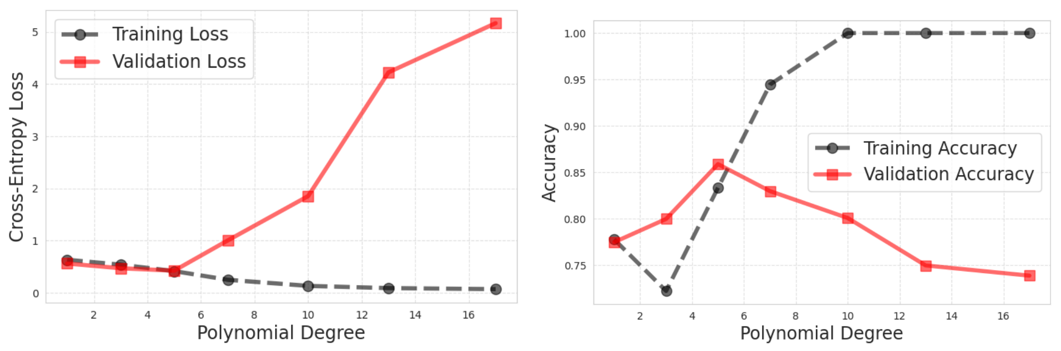
Figure 9: *Classifiers with a Polynomial Basis of Increasing Degree.* The sequence of plots shows the resulting decision boundaries when we increase the degree of the polynomial basis function.

3.2 Model Selection

Having the held-out validation set is integral for performing *model selection*: choosing the best out of a candidate set of models. This is especially important to do when the candidate models vary in their complexity and expressive power. Examine Figure 9; it shows the classification boundaries obtained via categorical regression with a polynomial basis of increasing degree. The classification boundaries between categories are linear in the case of the top left figure and become highly non-linear, culminating in the 17-degree basis shown in the bottom right. *Which of these models is the best classifier to deploy to the real world?*

We can answer this question by plotting the validation loss on the y-axis and the polynomial degree on the x-axis. See Figure 10a for this plot, showing the cross-entropy loss for the validation data in red and for the training data in black. As the polynomial degree increases, the training loss strictly decreases, as the model can fit the data better and better. However, the validation loss (red) does not follow the same monotonic pattern. Instead, the validation loss decreases for degrees 1, 3, and 5, but it increases for all degrees thereafter. This is because the extra model capacity afforded by the higher degrees is only useful for modeling noise in the training data, not for exactly signals that will *generalize* to other data sets. Thus, we would choose the degree 5 polynomial model to deploy to our application. Figure 10 is the same plot but with accuracy as the quality metric instead of cross-entropy loss. Here we see that the same polynomial degree is identified as best (a degree of 5). Yet, interestingly, accuracy is not monotonic and actually decreases from degree 1 to degree 3 and then continues upward from there, finally reaching an accuracy of 100%. An accuracy of 100% should always bring suspicious: either you have an extremely easy problem, a very small dataset, have overfit the model, or otherwise have a bug in your code.

Fortunately, the magic of cross validation (i.e. evaluation on a held-out set) is designed exactly for these situations. The model builder has the natural desire to define more and more expressive models, hoping that the additional flexibility translates into better predic-



(a) Cross-Entropy Loss (b) Classification Accuracy

Figure 10: *Performance on Training vs Held-Out Validation Data.* Cross-entropy loss vs polynomial degree of the classifier is shown in (a); accuracy vs polynomial degree is shown in (b).

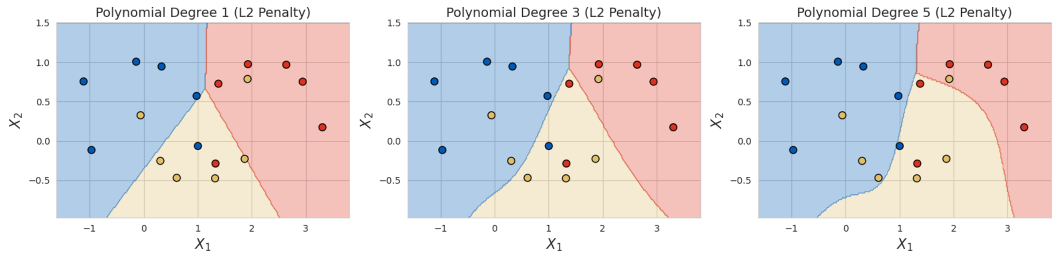


Figure 11: *Classifiers with Polynomial Basis and L2-Regularized Weights.* The sequence of plots shows the resulting decision boundaries when we increase the degree of the polynomial basis function.

tive performance. Yet this desire is counter balanced by cross validation, as it allows us to see if that extra capaticity is being used to model signal or just noise that is specific to the training set.

3.3 Capacity Control

While evaluating on held-out data is perhaps the best method for controlling model capacity and selecting the appropriately powerful model, it is expensive since it is data driven, and high-quality data is all but always expensive. There exists some general techniques for controlling model capacity without data, and I will give an brief overview of two here: regularization penalties and ensembling.

Regularization Penalty Regularization penalties are an additional term that is added to the loss function that penalizes more complex models:

$$\widetilde{\ell}(\mathbf{w};\mathcal{D},\lambda) \ = \ \ell(\mathbf{w};\mathcal{D}) \ + \ \lambda \cdot \mathcal{R}(\mathbf{w})$$

where $\ell(\mathbf{w};\mathcal{D})$ is the original training loss function, which will purely reward the model's fit to the training data, $\lambda \in \mathbb{R}^{+}$ is a hyperparameter that controls the strength of the regularization, or in other words, the relative trade-off between data fit and complexity

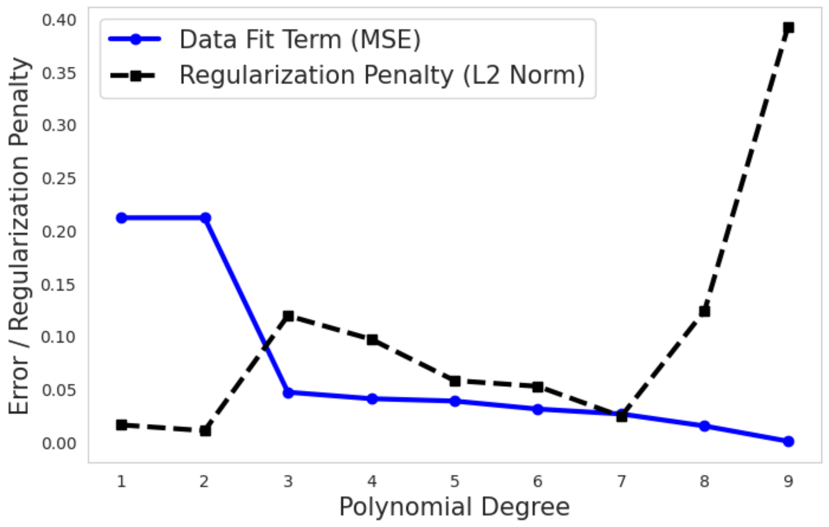


Figure 12: *Data Fit vs Complexity Penalty.* The sequence of plots shows the resulting decision boundaries when we increase the degree of the polynomial basis function.

control, and $\mathcal{R}(\mathbf{w})$ is the regularization penalty that rewards simpler models. One very common and easy way to control a model's capacity is simply by penalizing the weights from moving away from zero. A common implementation is to choose the $L2$ norm:

$$\widetilde{\ell}(\mathbf{w}; \mathcal{D}, \lambda) \ = \ \ell(\mathbf{w}; \mathcal{D}) \ + \ \lambda \cdot \sum_{d=1}^{D} \mathrm{w}_d^2,$$

where the square is applied so that we equally penalize large positive and negative weights. Figure 12 shows the same example of categorical regression with a polynomial basis, but now an L2 penalty has been applied to the weights. Comparing these decision boundaries to the ones in the first row of Figure 5, we see that the decision boundaries are relatively smoother and more closely resemble that of the first-order model's straight-line boundaries. Figure 12 shows how the two terms interact in a real-valued regression example; the data-fit term is in blue, adnd the regularization penalty (without λ) is the black dotted line. We see that at first, the regularization penalty is low as the data fit is poor. Then at degree three, the data fit improves dramatically yet the regularization penalty jumps up as well, indicating a more complex model has been found. The two stay relatively level until degree seven when the data fit makes modest improvements but the regularization penalty skyrockets, indicating that the gains in data fit are only coming at the 'cost' of using increasingly complex models—which is likely a sign of overfitting and poor model generalization. This information is nice in that we can see some hints at how well-fit the model is without using held-out validation data. However, these insights are not as robust as looking at the error on held-out data and also comes at the price of having another hyperparameter to choose (λ). Fortunately, the model is usually less sensitive to the setting of λ than the polynomial degree. If this were not the case, then one would need to intensively cross-validate λ, and if one needs to do that, that data might just as well be used to tune the polynomial degree directly.

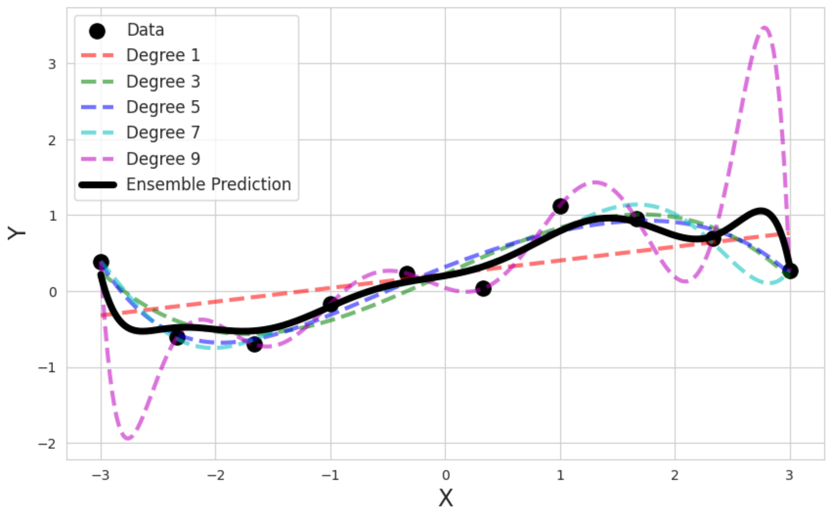


Figure 13: *Ensemble of Polynomial Regressors of Varying Degrees.*

Ensembling Another technique that is easy to implement and controls complexity is *ensembling* a collection of models. This is where we train several models, and instead of using just one to make predictions, we combine the predictions of all models. The intuition behind why this is beneficial is that 'overfit models often overfit in different ways,' and therefore, the differences across models cancel out, leaving only the similarities that correspond to true signal. For real-valued regression, models can be ensembled simply by averaging their mean functions:

$$\bar{f}(\mathbf{x}; \mathbf{w}_1, \ldots, \mathbf{w}_J) \;=\; \frac{1}{J}\sum_{j=1}^{J} \mathbb{E}\left[\mathrm{y}|f(\mathbf{x}; \mathbf{w}_j)\right] \;=\; \frac{1}{J}\sum_{j=1}^{J} \mathbf{w}_j^T \mathbf{x}$$

where J is the total number of models in the ensemble. Notice that for linear models, averaging the mean functions is equivalent to averaging the model parameters: $\frac{1}{J}\sum_{j=1}^{J} \mathbf{w}_j^T \mathbf{x} = \left(\frac{1}{J}\sum_{j=1}^{J} \mathbf{w}_j^T\right)\mathbf{x}$; though this is not true in general (e.g. for logistic regression). Figure 13 shows an ensemble of real-value regressors using a polynomial basis that ranges over degrees 1, 3, 5, 7, and 9 (colored dashed lines). The ensemble output is shown by the black solid line. Note how the black line represents a flexible but restrained model fit—going through more data points than most of the simple models but doesn't have the drastic turns of the 9-degree polynomial.

For categorical regression models, there are two ways to ensemble them. The first is called *voting*: each model makes a prediction, and the class that was predicted most frequently across the ensemble is the aggregate prediction:

$$\bar{f}_{\text{vote}}(\mathbf{x}; \mathbf{w}_1, \ldots, \mathbf{w}_J) \;=\; \underset{k \in [1,K]}{\arg\max} \sum_{j=1}^{J} \hat{y}_{j,k}$$

where $\hat{y}_{j,k}$ is the one-hot-encoded prediction for label k from model j. Another technique is to aggregate the outputs of the inverse link functions directly. For example, for the softmax we have:

$$\bar{f}_{\text{soft}}(\mathbf{x}; \mathbf{w}_1, \dots, \mathbf{w}_J) \;=\; \underset{k \in [1,K]}{\arg\max} \; \sum_{j=1}^{J} \texttt{softmax}_k \left(f(\mathbf{x}; \mathbf{w}_j) \right)$$

where $\texttt{softmax}_k(\cdot)$ denotes the softmax's output for class k. Ensembling is attractive since it doesn't require too much thought: as long as we can train a sufficiently diverse set of models, then it should would and not be too sensitive to the settings of any one particular model. However, the drawback comes at the cost of computation. Not only do we have to train J models, we usually need to deploy all J models to the real world as well, since unlike with linear models for real-valued regression, there is usually not a way to combine all J sets of parameters into one object while maintaining the full benefits of the ensemble.

4 Feedforward Neural Networks

We have so far only considered 'shallow' models—namely, linear models. We have worked with shallow and 'narrow' models defined by have a fixed, given feature set. Yet we have also considered shallow and 'wide' models obtained by having a dynamic feature set, e.g. a polynomial basis of arbitrary degree. Now we will consider 'deep' models that will adapt and change the feature basis via learning. Or in other words, the feature basis itself will only have a very generic form, and the data itself will guide how the features are transformed into different representations. While we saw that polynomial basis functions can represent very complex functions, the degree of the polynomial is determined by us and has a very specific influence on the model. With deep models, we will not have to make such strong choices about what is the best representation space of the features.

4.1 Adaptive Features and the Importance of Non-Linearities

The key idea behind *deep learning* is to have a feature representation (or basis) that is flexible and determined by the data, not by the model builder's assumptions. Thus, in general, we can think of deep models as having the general form:

$$\mathbb{E}[\mathrm{y}|\mathbf{x}] \;=\; g^{-1}\left(\mathbf{w}^T \psi(\mathbf{x})\right) \tag{20}$$

where g is the link function (as usual), $\mathbf{w}$ are parameters to be learned, and $\psi(\mathbf{x})$ is a new representation of $\mathbf{x}$ that *depends on* $\mathbf{x}$ *itself and will have its own parameters.* This is unlike in polynomial regression in that the feature basis was fixed across all possible feature vectors, though would change with the specifics of the features themselves.

So how should we build this adaptive basis $\psi(\mathbf{x})$? The first idea one might have is to start simple and make $\psi(\mathbf{x})$ a linear model as well:

$$\psi(\mathbf{x}; \mathbf{U}) \;=\; \mathbf{U}^T \mathbf{x}$$

where $\mathbf{U} \in \mathbb{R}^{D \times D'}$ is a matrix of real-values that transforms $\mathbf{x}$ from a D-dimensional representation into a new representation of size D'. Plugging this expression into the model above, we have:

$$\mathbb{E}[\mathrm{y}|\mathbf{x}] \;=\; g^{-1}\left(\mathbf{w}^T \psi(\mathbf{x}; \mathbf{U})\right) \;=\; \mathbf{w}^T \left(\mathbf{U}^T \mathbf{x}\right),$$

which means that we would first transform the feature vector with $\mathbf{U}$ and then multiply it with $\mathbf{w}$.

However, there is a problem with the above construction: this model is no more flexible and expressive than the original model $g^{-1}\left(\mathbf{w}^T\mathbf{x}\right)$! This is because a composition of linear transformations is no more expressive than one linear transformation, i.e. $\mathbf{w}^T\left(\mathbf{U}^T\mathbf{x}\right) = \left(\mathbf{w}^T\mathbf{U}^T\right)\mathbf{x}$, which could be equivalently represented just with a linear transformation of the same shape / size as $\mathbf{w}$. Thus, to break out of the space of linear transformations, we need to introduce some non-linear transform. We'll call it $\phi(\mathbf{z})$, which usually acts element-wise:

$$\phi(\mathbf{z}) \ = \ [\phi(z_1), \ldots, \phi(z_D)], \qquad \mathbf{z} \in \mathbb{R}^D.$$

Now introducing this non-linear function into the model we had before, we have:

$$\mathbb{E}[\mathrm{y}|\mathbf{x}] \ = \ g^{-1}\left(\mathbf{w}^T\psi(\mathbf{x}; \mathbf{U}, \phi(\cdot))\right) \ = \ \mathbf{w}^T\phi\left(\mathbf{U}^T\mathbf{x}\right).$$

By definition of ϕ being non-linear, we have successfully made it so that $\mathbf{w}^T\phi\left(\mathbf{U}^T\mathbf{x}\right) \neq \phi\left(\mathbf{w}^T\mathbf{U}^T\right)\mathbf{x}$. Of course, there are many, many choices of $\phi(\cdot)$ for which the above holds. Next we will see some specific implementations.

4.2 Neural Networks

Neural networks (NNs) (a.k.a. *artificial neural networks*) are a specific implementation of the adaptive-basis-function framework described above. Really, the only thing left to introduce is the specific terminology that has become attached to them due to the influences and inspirations of biological intelligence.

Terminology Keeping the same notation as above, both $\mathbf{w}$ and $\mathbf{U}$ are called 'weights', just like in regular linear regression, and are parameters that need to be learned from data. Again, this is why NNs can potentially be very powerful: they learn an adaptive representation from the data. The non-linear transform $\phi(\cdot)$ is called the 'activation function' or 'transfer function,' with the former name arising from the idea of biological neuron 'activating' or 'firing'. Thus, one element of the new representation $\phi_d(\mathbf{U}^T\mathbf{x})$ is usually called a 'neuron' or 'hidden unit.' The collection of all elements $\phi(\mathbf{U}^T\mathbf{x}) = \left[\phi_1(\mathbf{U}^T\mathbf{x}), \ldots, \phi_D(\mathbf{U}^T\mathbf{x})\right]$ is called a 'hidden layer.' The description 'hidden' emphasizes that (i) the representation is neither directly the input or output variables but rather some intermediate variables, and (ii) we do not exactly know how to interpret $\phi(\mathbf{U}^T\mathbf{x})$ since the representation is more free-form than, say, a polynomial basis. Another common term is to call the value $\mathbf{U}^T\mathbf{x}$ the pre-activation (vector) since it serves as the input to the activation function ϕ. To make things a bit more confusing, NNs are sometimes referred to as *multilayer perceptrons* (MLPs) as they can be seen as stacking (or more precisely, composing) multiple perceptron architectures. Lastly, we call a NN of this form 'fully connected' since all input values can influence all of the hidden units, and all hidden units can influence the output(s).

Activation Functions The choice of activation function ϕ is actually quite crucial to the success of NNs. We will briefly introduce some popular choices here and revisit the topic later, after talking about how to train NNs. See Figure for an overview of four activation

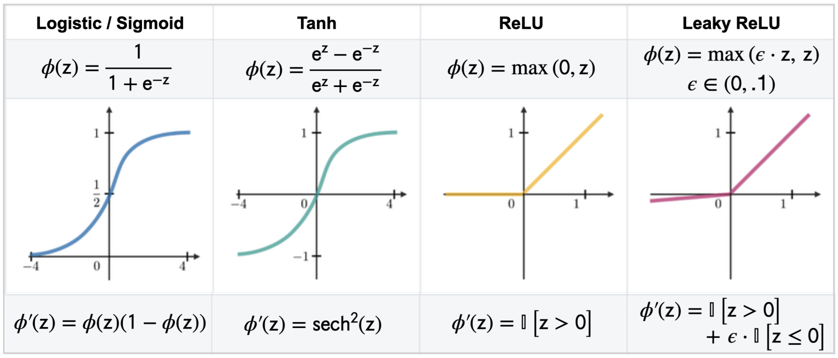


Figure 14: *Common Activation Functions.* Four functions that are commonly used for the activation functions of a NN's hidden layers. Their derivatives are given in the bottom row.

functions. One we have already seen before: the logistic (sigmoid) function. The second one from the left, the hyperbolic tangent function (tanh), looks similar to the logistic in shape, but notice that it's range is from $(-1, 1)$, being symmetrical about the x-axis, whereas the logistic's output is never negative. The last two are variants of the *rectified linear unit* (ReLU). In the simplest implementation, the ReLU simply preserves the pre-activation value when it is positive sets it to zero when negative. The Leaky ReLU behaves similarly, except that it allows some information to 'leak' to the left of zero. As we will discuss more later, this is to make gradient-based training a bit easier. Note that, though the ReLU and Leaky RelU are piecewise linear functions with just two pieces (meaning that they are linear functions one one side of the origin), this rather small amount of non-linearity is still sufficient to prevent NN's from collapsing back to linear functions, as would happen if we used the identity function as the activation.

4.3 Deep Neural Networks

In the preceding subsection, we introduced a NN with one hidden layer. But why stop there? One can simply repeat the process of applying a linear transformation followed by a non-linear activation function to create NNs of arbitrary 'depth'. We can then define *deep NNs* (DNNs) of depth $(L+2)$—with L of the layers being hidden—via the following two equations:

$$\mathbb{E}[\mathrm{y}|\mathbf{x}] \;=\; g^{-1}\left(\mathbf{w}_L^T \mathbf{h}_L\right), \quad \mathbf{h}_l \;=\; \phi\left(\mathbf{W}_{l-1}^T \mathbf{h}_{l-1}\right), \quad \text{where} \quad \mathbf{h}_0 = \mathbf{x} \in \mathbb{R}^{D_0}, \tag{21}$$

$\mathbf{w}_L \in \mathbb{R}^{D_L}$ (or a $(D_L \times K)$-matrix for K dimensional outputs), $\mathbf{h}_l$ is of dimension D_l, and $\mathbf{W}_{l-1} \in \mathbb{R}^{D_{(l-1)} \times D_l}$. Yet note that the variable $\mathbf{h}_l$ is really just notation for the function composition. This goes back to the above point: the hidden layers are really just computational operations that allow us to define a more expressive model, and they do not have the strong semantic interpretations that the input and output 'layers' have. A DNN

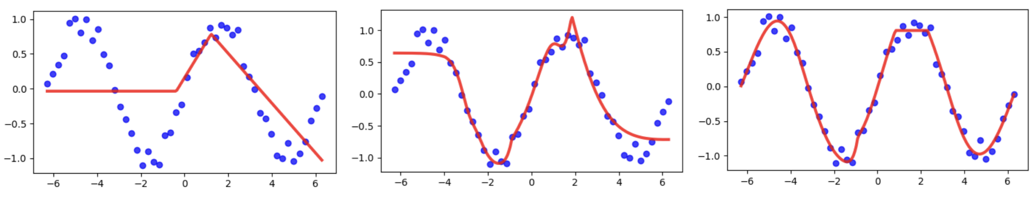

(a) 1 Hidden Layer (b) 2 Hidden Layers (c) 3 Hidden Layers

Figure 15: *Increasing the Depth of a Neural Network.* The three plots above show when neural networks of increasing depth are fit to the sine function. All hidden layers have three hidden units that use Tanh activation functions.

can be equivalently written just in terms of the function compositions:

$$\begin{aligned}
\mathbb{E}[\mathrm{y}|\mathbf{x}] &= g^{-1}\left(\mathbf{w}_L^T\mathbf{h}_L\right) \\
&= g^{-1}\left(\mathbf{w}_L^T\phi\left(\mathbf{W}_{L-1}^T\mathbf{h}_{L-1}\right)\right) \\
&= g^{-1}\left(\mathbf{w}_L^T\phi\left(\mathbf{W}_{L-1}^T\phi\left(\mathbf{W}_{L-2}^T\mathbf{h}_{L-2}\right)\right)\right) \\
&= g^{-1}\left(\mathbf{w}_L^T\phi\left(\mathbf{W}_{L-1}^T\phi\left(\mathbf{W}_{L-2}^T\phi\left(\mathbf{W}_{L-3}^T\mathbf{h}_{L-3}\right)\right)\right)\right) \\
&\vdots
\end{aligned} \tag{22}$$

and so on, continuing to write each hidden layer in terms of the previous one until we stop upon reaching $\mathbf{h}_0 = \mathbf{x}$. Starting from the input layer and evaluating all hidden layers to produce an output is called *forward propagation*. I have written the above expression as having the same activation function applied at every depth in the NN. This is usually the case, but nothing is preventing us from having different activation functions at different layer (or even within a layer). Doing that could possibly improve the model in cases, but cross-validating all of those choices / configurations would probably not be worth the effort. Choosing the depth and width of the layers is usually a more critical hyperparameter to tune. Figure 15 demonstrates the power of adding more hidden layers, as the harmonic function can be better and better approximated as the NN goes from one to three hidden layers.

Incorporating the Offset Parameter Recall that for linear models, we assumed that the input feature vector always had a constant element of one so that we could implement the offset / bias / intercept parameter just by multiplication. We want to do the same thing for DNNs, and therefore at every hidden layer, we need to assume a constant value of one is appended to the hidden layer:

$$\mathbf{h}_l = \left[\phi\left(\mathbf{W}_{l-1}^T\mathbf{h}_{l-1}\right),\ 1\right].$$

Going forward, we will assume the hidden layers are always implemented using this definition unless otherwise stated.

Why depth vs width? Before we get too excited about making our NNs as deep as possible, it is worth considering: why can't we making our NNs powerful just by making one or two hidden layers really wide? Of course, we are allowed to pick the dimensionality of the hidden layers, so nothing is stopping us from making them on the order of a thousand or even a million units wide. In fact, there is actually a theorem called the *Universal Approximation Theorem*[1] that says that, if you can make the NN wide enough, it can approximate just about any function you would possibly ever need in practice. Thus, at least in theory, depth is not necessary to define expressive NNs. *However*, the theorem does not prescribe how wide you need to make the NN, and in fact, you may need exponentially more hidden units than you have feature dimensions. There has also been theoretical work showing that, for a fixed number of parameters, a deeper network can represent functions that the corresponding wide network cannot (Telgarsky, 2016). Yet the simple answer is that people have found that increasing an NN's depth is a much more effective way to use parameters and just generally works better in practice. In the future, perhaps we will have new architectures or learning algorithms that better take advantage of width than depth.

4.4 Gradient-Based Learning of Neural Networks

We can learn the parameters of a neural network via gradient descent, following a similar recipe to how we implemented it for linear models. We will use the same loss functions as considered earlier—such as squared error or cross-entropy error. For example, if we are using a DNN with L hidden layers for a real-valued regression task $\mathrm{y} \in \mathbb{R}$, we would have:

$$\begin{aligned} \ell\left(\mathbf{W}_0, \ldots, \mathbf{W}_l, \ldots, \mathbf{w}_L; \mathcal{D}\right) &= \frac{1}{N}\sum_{n=1}^{N}\left(\mathbb{E}[y_n|\boldsymbol{x}_n] - y_n\right)^2 \\ &= \frac{1}{N}\sum_{n=1}^{N}\left(\mathbf{w}_L^T \boldsymbol{h}_{n,L} - y_n\right)^2 \\ &= \frac{1}{N}\sum_{n=1}^{N}\left(\mathbf{w}_L^T \phi\left(\mathbf{W}_{L-1}\boldsymbol{h}_{n,L-1}\right) - y_n\right)^2 \\ &\vdots \end{aligned} \tag{23}$$

and so one, writing each hidden layer in terms of the previous one. $\mathbf{W}_0, \ldots, \mathbf{W}_l, \ldots, \mathbf{w}_L$ are all parameters in the DNN. Given a loss function, we then need to compute the gradient of the weights at all layers ($\mathbf{W}_l \ \forall l$):

$$\mathbf{W}_l^{t+1} \ = \ \mathbf{W}_l^t \ - \ \alpha \cdot \nabla_{\mathbf{W}_l^t}\, \ell\left(\mathbf{W}_0^t, \ldots, \mathbf{W}_l^t, \ldots, \mathbf{w}_L^t; \mathcal{D}\right) \tag{24}$$

where t indexes the gradient descent iteration.

Example: Scalar NN Let's build our intuition by working out the gradient equation for a simple NN in which the weights are all scalars. Moreover, let's assume the task is real-valued regression, and the NN has two hidden layers, no offset parameters, scalar inputs and outputs, and logistic activations. It can be implemented like so:

$$\mathbb{E}[\mathrm{y}|\mathrm{x}] = \mathrm{w}_2 \cdot \mathrm{h}_2, \quad \mathrm{h}_2 = \phi(\underbrace{\mathrm{w}_1 \cdot \mathrm{h}_1}_{\mathrm{a}_2}), \quad \mathrm{h}_1 = \phi(\underbrace{\mathrm{w}_0 \cdot \mathrm{x}}_{\mathrm{a}_1}),$$

[1] `https://en.wikipedia.org/wiki/Universal_approximation_theorem`

where a is an intermediate variable we will use to denote the pre-activation. Starting with the derivative for w_2, we have:

$$\begin{aligned}\frac{d}{d\mathrm{w}_2}\ \ell(\mathrm{w}_2,\mathrm{w}_1,\mathrm{w}_0;\mathcal{D}) &= \frac{1}{N}\sum_{n=1}^{N} \frac{d\ \ell_n}{d\ \mathbb{E}[y_n|x_n]}\frac{d\ \mathbb{E}[y_n|x_n]}{d\ \mathrm{w}_2} \\ &= \frac{1}{N}\sum_{n=1}^{N} \underbrace{2\cdot((\mathrm{w}_2\cdot h_{n,2})-y_n)}_{\frac{d\ \ell_n}{d\ \mathbb{E}[y_n|x_n]}}\ \underbrace{h_{n,2}}_{\frac{d\ \mathbb{E}[y_n|x_n]}{d\ \mathrm{w}_2}}\end{aligned}$$

Moving on to the derivative for w_1, we have:

$$\begin{aligned}\frac{d}{d\mathrm{w}_1}\ \ell(\mathrm{w}_2,\mathrm{w}_1,\mathrm{w}_0;\mathcal{D}) &= \frac{1}{N}\sum_{n=1}^{N} \frac{d\ \ell_n}{d\ \mathbb{E}[y_n|x_n]}\frac{d\ \mathbb{E}[y_n|x_n]}{d\ h_{n,2}}\ \frac{d\ h_{n,2}}{d\ a_{n,2}}\ \frac{d\ a_{n,2}}{d\ \mathrm{w}_1} \\ &= \frac{1}{N}\sum_{n=1}^{N} \underbrace{2\cdot((\mathrm{w}_2\cdot h_{n,2})-y_n)}_{\frac{d\ \ell_n}{d\ \mathbb{E}[y_n|x_n]}}\ \underbrace{\mathrm{w}_2}_{\frac{d\ \mathbb{E}[y_n|x_n]}{d\ h_{n,2}}}\ \underbrace{h_{n,2}\cdot(1-h_{n,2})}_{\frac{d\ h_{n,2}}{d\ a_{n,2}}}\ \underbrace{h_{n,1}}_{\frac{d\ a_{n,2}}{d\ \mathrm{w}_1}}\end{aligned}$$

Then finally for w_0, we have:

$$\begin{aligned}&\frac{d}{d\mathrm{w}_0}\ \ell(\mathrm{w}_2,\mathrm{w}_1,\mathrm{w}_0;\mathcal{D}) \\ &\quad= \frac{1}{N}\sum_{n=1}^{N} \frac{d\ \ell_n}{d\ \mathbb{E}[y_n|x_n]}\frac{d\ \mathbb{E}[y_n|x_n]}{d\ h_{n,2}}\ \frac{d\ h_{n,2}}{d\ a_{n,2}}\ \frac{d\ a_{n,2}}{d\ h_{n,1}}\frac{d\ h_{n,1}}{d\ a_{n,1}}\frac{d\ a_{n,1}}{d\ \mathrm{w}_0} \\ &\quad= \frac{1}{N}\sum_{n=1}^{N} \underbrace{2\cdot((\mathrm{w}_2\cdot h_{n,2})-y_n)}_{\frac{d\ \ell_n}{d\ \mathbb{E}[y_n|x_n]}}\ \underbrace{\mathrm{w}_2}_{\frac{d\ \mathbb{E}[y_n|x_n]}{d\ h_{n,2}}}\ \underbrace{h_{n,2}\cdot(1-h_{n,2})}_{\frac{d\ h_{n,2}}{d\ a_{n,2}}}\ \underbrace{\mathrm{w}_1}_{\frac{d\ a_{n,2}}{d\ h_{n,1}}}\ \underbrace{h_{n,1}\cdot(1-h_{n,1})}_{\frac{d\ h_{n,1}}{d\ a_{n,1}}}\ \underbrace{x}_{\frac{d\ a_{n,1}}{d\ \mathrm{w}_0}}\end{aligned}$$

There are a few observations to make here. The first is that the $\frac{d\ \ell_n}{d\ \mathbb{E}[y_n|x_n]}$ partial derivative is present in all of the calculations, which makes sense since this is what connects the NN to the label y_n to provide the supervision signal. From there, the signal flows backward into one or more hidden layers. Next we see that the partial derivatives $\frac{d\ \mathbb{E}[y_n|x_n]}{d\ h_{n,2}}$ and $\frac{d\ h_{n,2}}{d\ a_{n,2}}$ are present in the total derivatives for w_1 and w_0. Again, this makes sense since these parameters reside earlier in the model than both $\mathbb{E}[y_n|x_n]$ and $h_{n,2}$. This suggests that, in our computational implementations, there are savings to be had by caching these partial derivatives for multiple use.

4.5 Backpropagation

As we saw with the example of a scalar NN, we can think of the prediction error signal as flowing backwards from the loss function towards the input layer. This is intuitive: if *forward propagation* sends the signal from the input to the DNN output, then *backpropagation* sends the signal from the output (i.e. the deepest part of the network) towards to shallow layers. This can be seen in the general form of the chain rule as applied above:

$$\frac{d}{d\mathbf{W}_l}\ell(\mathbf{W}_0,\ldots,\mathbf{W}_l,\ldots,\mathbf{w}_L;\mathcal{D}) = \frac{d\ell}{d\mathbb{E}[\mathrm{y}|\mathbf{x}]}\ \frac{d\mathbb{E}[\mathrm{y}|\mathbf{x}]}{d\mathbf{h}_L}\ \frac{d\mathbf{h}_L}{d\mathbf{h}_{L-1}}\ \frac{d\mathbf{h}_{L-1}}{d\mathbf{h}_{L-2}}\ \cdots\ \frac{d\mathbf{h}_{l+2}}{d\mathbf{h}_{l+1}}\ \frac{d\mathbf{h}_{l+1}}{d\mathbf{W}_l}. \tag{25}$$

Notice the repeated structure of $(d\mathbf{h}_{l+1}/d\mathbf{h}_l)$, which means that all parameter involved in computation of $\mathbf{h}_l$ share all the same partial derivatives required to compute $(d\ell/d\mathbf{h}_l)$! Thus we trade off computational complexity for memory.

Forward vs Reverse Mode Give the sequence of partial derivatives above, you can evaluate them in two different orders: left-to-right or right-to-left, i.e.:

$$\left(\left(\frac{d\ell}{d\mathbf{h}_L}\frac{d\mathbf{h}_L}{d\mathbf{h}_{L-1}}\right)\frac{d\mathbf{h}_{L-1}}{d\mathbf{h}_{l+1}}\right)\frac{d\mathbf{h}_{l+1}}{d\mathbf{W}_l} \quad \text{vs} \quad \frac{d\ell}{d\mathbf{h}_L}\left(\frac{d\mathbf{h}_L}{d\mathbf{h}_{L-1}}\left(\frac{d\mathbf{h}_{L-1}}{d\mathbf{h}_{l+1}}\frac{d\mathbf{h}_{l+1}}{d\mathbf{W}_l}\right)\right).$$

The left-to-right version starts the computation at the last layer and therefore is known as *reverse mode.* This is the typical way that backpropagation is implemented since, again, the partial derivatives are computed starting at the deepest layers. This is most efficient when the output dimension is smaller than the input dimension—which is usually the case—since the highest-dimensional (sub)gradients will be computed last. *Forward mode* is the right-to-left version, and it is more computationally efficient when the input dimension is smaller than the output dimension, which follows the same logic that motivates reverse mode.

Exploding and Vanishing Gradients The shared partial derivatives that makes backpropagation computationally efficient also has a downside: if any of the intermediate derivatives $(d\mathbf{h}_{l+1}/d\mathbf{h}_l)$ are problematic, then this problem will propagate to all parameters that reside at all earlier layers. Due to the multiplicative structure of the chain rule, by 'problematic' we usually mean we are worried about derivatives that are very large or very small. The former is called the *exploding gradient* problem, and the latter is called the *vanishing gradient* problem. To make the vanishing gradient version explicit, let's assume that partial derivative $(d\mathbf{h}_{l+1}/d\mathbf{h}_l) \approx 0$. We then have that:

$$\frac{d\ell}{d\mathbf{h}_L}\frac{d\mathbf{h}_L}{d\mathbf{h}_{l+1}}\cancelto{\approx 0}{\frac{\mathbf{h}_{l+1}}{\mathbf{h}_l}}\frac{\mathbf{h}_l}{\mathbf{h}_{l-1}}\frac{\mathbf{h}_{l-1}}{\mathbf{h}_{l-2}}\cdots \approx 0$$

for any parameter at a layer earlier than $\mathbf{h}_l$. The same logic would follow for exploding gradients: one really large partial derivative can blow up the whole chain. For many years, it was difficult to train DNNs for this very problem, as we did not yet have the tools and tricks to effectively stabilize the gradients across multiple hidden layers.

Saturating Activation Functions One common cause of vanishing gradients is what's call 'saturating' activation functions. This means that the input to the activation function is in a regime in which the output of the function is relatively flat and therefore has a derivative near zero. Recall the logistic activation function's derivative: $\phi'(\mathrm{z}) = \phi(\mathrm{z})(1-\phi(\mathrm{z}))$. How can the logistic saturate and cause the gradient signal to vanish? This would happen when $\phi(\mathrm{z}) \approx 0$ or $\phi(\mathrm{z}) \approx 1$. Next consider the ReLU activation; when would it saturate? As it's derivative is $\phi'(\mathrm{z}) = \mathbb{I}[\mathrm{z} > 0]$, the function can saturate when the input is negative. In fact, when many of a DNN's ReLUs are evaluating to zero, this is called having 'dead ReLUs.' However, unlike the logistic, the ReLU only saturates to one side of the origin, not both (like the logistic and tanh). Lastly, consider the Leaky RelU. The hyperparameter ϵ prevents it from ever having a derivative of zero. Rather, for negative inputs the derivative is simply 'small.' Thus, the Leaky ReLU can never fully saturate.

4.5.1 Vectorized Implementation

When implementing DNNs in practice, we often want to make use of matrix / vector products as much as possible, since these are fast to implement with modern software and on graphics processing units. The implementation I have given above aims to follow the model definition given for linear models, but to reduce computational overhead, we'd like to make some modest changes in practice. Firstly, let's assume that the feature vectors are D-dimensional and the labels are one-hot-encoded K-dimensional vectors. We will then assume we see N training points, which we can aggregate into matrices $\boldsymbol{X}$ and $\boldsymbol{Y}$ with both have N rows, i.e.:

$$\boldsymbol{X} = \begin{bmatrix} \boldsymbol{x}_1^T \\ \vdots \\ \boldsymbol{x}_N^T \end{bmatrix}, \qquad \boldsymbol{Y} = \begin{bmatrix} \boldsymbol{y}_1^T \\ \vdots \\ \boldsymbol{y}_N^T \end{bmatrix}.$$

Moving on to the weight parameters $\mathbf{W}_l$, originally we defined the NN as requiring $\mathbf{W}_l$ to undergo a transpose operation—again, to follow the GLM implementation. But in practice, we can simply implement the hidden layers using the product in the other direction in order to keep the data points as the rows and the hidden units as the columns:

$$\boldsymbol{H}_l \ = \ \phi\left(\boldsymbol{H}_{l-1}\mathbf{W}_{l-1}\right)$$

where $\boldsymbol{H}_{l-1}$ is of size $(N \times D_{l-1})$, $\mathbf{W}_{l-1}$ is of size $(D_{l-1} \times D_l)$, and $\boldsymbol{H}_l$ is of size $(N \times D_l)$. Finally, for the output layer and loss, we have:

$$\ell\left(\mathbf{W}_0, \ldots, \mathbf{W}_L; \boldsymbol{X}, \boldsymbol{Y}\right) \ = \ -\frac{1}{N} \cdot \mathbf{1}^T \left(\boldsymbol{Y} \odot \log \texttt{softmax}\left(\boldsymbol{H}_L \mathbf{W}_L\right)\right) \mathbf{1}$$

where $\odot$ denotes the element-wise product and $\mathbf{1}^T(\cdot)\mathbf{1}$ is just a fancy way of writing that we are going to sum over all $N \times K$ elements of $\boldsymbol{Y} \odot \log \texttt{softmax}\left(\boldsymbol{H}_L\mathbf{W}_L\right)$. While this product has $N \times K$ elements, only N are active since there is only one 1 in each row of $\boldsymbol{Y}$.

Binary classification and real-valued regression would be setup in similar ways. For example, for real-valued regression, the loss function would be

$$\ell\left(\mathbf{W}_0, \ldots, \mathbf{W}_L; \boldsymbol{X}, \boldsymbol{Y}\right) \ = \ -\frac{1}{N} \cdot \mathbf{1}^T \left(\boldsymbol{H}_L \mathbf{W}_L - \mathbf{Y}\right)^2 \mathbf{1}$$

where $\boldsymbol{Y} \in \mathbb{R}^{N \times K}$ for K output dimensions and when assuming that the underlying Normal distribution has a diagonal covariance matrix and $(\cdot)^2$ acts element-wise.

General Form for Backpropagation We can derive a general form for the vectorized version of backpropagation's gradient equation. Recall that for many of the nice choices of inverse link functions discussed in this course (e.g. logistic for binary classification, softmax for multi-class classification, identity for real-valued regression), the gradient w.r.t. the model output $\mathbf{H}_L\mathbf{W}_L$ (i.e. the output before the link is applied) has the nice form:

$$\nabla_{\mathbf{H}_L\mathbf{W}_L}\ \ell\left(\mathbf{W}_0, \ldots, \mathbf{W}_L; \boldsymbol{X}, \boldsymbol{Y}\right) \ = \ \frac{1}{N}\left(\mathbb{E}\left[\mathbf{Y}|\mathbf{X}\right] - \boldsymbol{Y}\right)^T$$

where the transpose is taken because we define the gradient to be in row orientation. Taking the gradient one-step further into the hidden units, we have in column-form:

$$\frac{1}{N}\left(\mathbb{E}\left[\mathbf{Y}|\mathbf{X}\right]-\boldsymbol{Y}\right)\mathbf{W}_L^T\ \in\mathbb{R}^{N\times D_L}.$$

Then the activation function is applied element-wise, which means the gradient has an element-wise product:

$$\frac{1}{N}\left(\mathbb{E}\left[\mathbf{Y}|\mathbf{X}\right]-\boldsymbol{Y}\right)\mathbf{W}_L^T\odot\phi'(\mathbf{A}_L)\ \in\mathbb{R}^{N\times D_L}.$$

The backpropagating one step further into $\mathbf{H}_{L-1}$, we have:

$$\frac{1}{N}\left(\mathbb{E}\left[\mathbf{Y}|\mathbf{X}\right]-\boldsymbol{Y}\right)\mathbf{W}_L^T\odot\phi'(\mathbf{A}_L)\mathbf{W}_{L-1}^T\ \in\mathbb{R}^{N\times D_{L-1}}.$$

Then once again through the hidden units we have:

$$\frac{1}{N}\left(\mathbb{E}\left[\mathbf{Y}|\mathbf{X}\right]-\boldsymbol{Y}\right)\mathbf{W}_L^T\odot\phi'(\mathbf{A}_L)\mathbf{W}_{L-1}^T\odot\phi'(\mathbf{A}_{L-1})\ \in\mathbb{R}^{N\times D_{L-1}}.$$

Notice how the $\mathbf{W}^T\ \phi'(\mathbf{A})$ structure is repeated. This means that for a weight matrix at arbitrary depth l, we have the general form:

$$\nabla_{\mathbf{W}_l}\ \ell\left(\mathbf{W}_0,\ldots,\mathbf{W}_L;\boldsymbol{X},\boldsymbol{Y}\right)\ =\ \frac{1}{N}\left[\left(\mathbb{E}\left[\mathbf{Y}|\mathbf{X}\right]-\boldsymbol{Y}\right)\prod_{j=L}^{l+1}\mathbf{W}_j^T\odot\phi'(\mathbf{A}_j)\right]^T\mathbf{H}_l, \qquad (26)$$

where switching back to row form allows the chain rule to be expressed just as left-to-right matrix multiplication. The term $\left[\left(\mathbb{E}\left[\mathbf{Y}|\mathbf{X}\right]-\boldsymbol{Y}\right)\prod_{j=L}^{l+1}\mathbf{W}_j^T\odot\phi'(\mathbf{A}_j)\right]$ is of size $N\times D_{l+1}$. Transposing it gives a matrix of size $D_{l+1}\times N$. $\mathbf{H}_l$ is of size $N\times D_l$. Thus the resulting is product is of size $D_{l+1}\times D_l$, which exactly the size of $\mathbf{W}_l^T$, which again is what we should expect by assuming the row-form of the gradient.

4.5.2 Skip Connections

While using (Leaky) ReLU activations helps with the vanishing gradient problem, it is not a foolproof solution. A simple yet robust solution called *skip connections* (a.k.a. *residual connections*) that is now ubiquitous was proposed and popularized by He et al. (2016). The idea is that the non-linear transformation should not directly parameterize the hidden units but rather an additive change from the previous hidden units:

$$\mathbf{h}_l\ =\ \phi\left(\mathbf{W}_{l-1}^T\mathbf{h}_{l-1}\right)\ +\ \mathbf{h}_{l-1}. \qquad (27)$$

We can see the non-linear transformation parameterizes the *residual* simply by moving the previous hidden units to the left-hand-side:

$$\mathbf{h}_l\ -\ \mathbf{h}_{l-1}\ =\ \phi\left(\mathbf{W}_{l-1}^T\mathbf{h}_{l-1}\right).$$

Parameterizing the hidden units in this way should ensure that gradient information can ‘backpropagate’ into previous hidden units even if $\phi\left(\mathbf{W}_{l-1}^T\mathbf{h}_{l-1}\right)$ is ‘dead’ or saturates to zero because then we will just have an identity transformation: $\mathbf{h}_l \approx \mathbf{h}_{l-1}$.

Example: Scalar NN Let's again return to our scalar NN to build intuition. Still assume the task is real-valued regression, and the NN has two hidden layers, no offset parameters, scalar inputs and outputs, and logistic activations. Yet now the NN has one skip connection from h_1 to h_2:

$$\mathbb{E}[\mathrm{y}|\mathrm{x}] = \mathrm{w}_2 \cdot \mathrm{h}_2, \quad \mathrm{h}_2 = \phi(\underbrace{\mathrm{w}_1 \cdot \mathrm{h}_1}_{\mathrm{a}_2}) \ + \ \mathrm{h}_1, \quad \mathrm{h}_1 = \phi(\underbrace{\mathrm{w}_0 \cdot \mathrm{x}}_{\mathrm{a}_1}),$$

where a is an intermediate variable we will use to denote the pre-activation. The derivatives w.r.t. w_2 and w_1 stay exactly the same in their form as given above, though the underlying semantics have changed since h_2 is defined differently. The derivative w.r.t. w_0 changes to:

$$\begin{aligned}
&\frac{d}{d\mathrm{w}_0} \ \ell(\mathrm{w}_2, \mathrm{w}_1, \mathrm{w}_0; \mathcal{D}) \\
&= \frac{1}{N}\sum_{n=1}^{N} \ \frac{d\ \ell_n}{d\ \mathbb{E}[y_n|x_n]} \frac{d\ \mathbb{E}[y_n|x_n]}{d\ h_{n,2}} \ \left(\frac{d\ h_{n,2}}{d\ a_{n,2}} \frac{d\ a_{n,2}}{d\ h_{n,1}} + \frac{d\ h_{n,2}}{d\ h_{n,1}}\right) \ \frac{d\ h_{n,1}}{d\ a_{n,1}} \frac{d\ a_{n,1}}{d\ \mathrm{w}_0} \\
&= \frac{1}{N}\sum_{n=1}^{N} \ \frac{d\ \ell_n}{d\ \mathbb{E}[y_n|x_n]} \frac{d\ \mathbb{E}[y_n|x_n]}{d\ h_{n,2}} \ \left(\frac{d\ h_{n,2}}{d\ a_{n,2}} \frac{d\ a_{n,2}}{d\ h_{n,1}} + 1\right) \ \frac{d\ h_{n,1}}{d\ a_{n,1}} \frac{d\ a_{n,1}}{d\ \mathrm{w}_0} \\
&= \frac{1}{N}\sum_{n=1}^{N} \underbrace{2 \cdot ((\mathrm{w}_2 \cdot h_{n,2}) - y_n)}_{\frac{d\ \ell_n}{d\ \mathbb{E}[y_n|x_n]}} \ \underbrace{\mathrm{w}_2}_{\frac{d\ \mathbb{E}[y_n|x_n]}{d\ h_{n,2}}} \ \left(\underbrace{h_{n,2} \cdot (1 - h_{n,2})}_{\frac{d\ h_{n,2}}{d\ a_{n,2}}} \ \underbrace{\mathrm{w}_1}_{\frac{d\ a_{n,2}}{d\ h_{n,1}}} \ + 1\right) \underbrace{h_{n,1} \cdot (1 - h_{n,1})}_{\frac{d\ h_{n,1}}{d\ a_{n,1}}} \ \underbrace{x}_{\frac{d\ a_{n,1}}{d\ \mathrm{w}_0}} \ .
\end{aligned}$$

The crucial 'link' in the chain rule is the term:

$$\frac{d}{dh_{n,1}} \ [\phi(\mathrm{w}_1 \cdot h_{n,1}) \ + \ h_{n,1}] \ = \ \frac{d\ h_{n,2}}{d\ a_{n,2}} \frac{d\ a_{n,2}}{d\ h_{n,1}} + \frac{d\ h_{n,2}}{d\ h_{n,1}} \ = \ \frac{d\ h_{n,2}}{d\ a_{n,2}} \frac{d\ a_{n,2}}{d\ h_{n,1}} + 1,$$

which means that even if $\phi'(\mathrm{w}_1 \cdot h_{n,1})$ vanishes, then $(d\mathrm{h}_2/d\mathrm{h}1) = 1$, which still allows backpropagation back to the earlier layers.

Changing Dimensions The above implementation of the skip connection requires that $\mathbf{h}_l$ and $\mathbf{h}_{l-1}$ be the same dimensionality. If we wish to change dimensions from layer l to $l+1$, we need to incorporate another matrix of parameters; let's call them $\mathbf{U}$:

$$\mathbf{h}_l \ = \ \phi\left(\mathbf{W}_{l-1}^T \mathbf{h}_{l-1} \ \right) \ + \ \mathbf{U}^T \mathbf{h}_{l-1}$$

where $\mathbf{U} \in \mathbb{R}^{D_{l-1} \times D_l}$, which should be the same size as $\mathbf{W}_{l-1}$. The parameters $\mathbf{U}$ would also need trained with gradient descent. The skip connection should still prevent vanishing gradients just so long as $\mathbf{U} \neq \mathbf{0}$.

4.5.3 Initializations

Skip connections allow backpropagation signals to bypass hidden layers whose gradient has vanished, but that does not stop the gradient from vanishing within a hidden layer. And of course, if many of the layers' gradients vanish, we would be using an effectively shallower

DNN. To make sure the gradient computation through a particular layer does not vanish / explode, it is very important that the DNN's weights are initialized sensibly. However, the particular initialization strategy depends on choice of activation functions. We give two below: one for sigmoidal (S-shaped) activations, one for ReLUs. In both cases, the goal is to have the initialization of the weights to automatically adjust to the size of the architecture. The more parameters we have, the more likely they are to generate large values during the forward pass (especially during the early iterations of gradient descent).

Xavier Initialization For sigmoidal activations such as logistic and tanh, the following *Xavier* initialization (Glorot and Bengio, 2010) scheme is effective:

$$\mathbf{W}_l \;\sim\; \texttt{Uniform}\left(\frac{-\sqrt{6}}{\sqrt{D_l + D_{l+1}}},\; \frac{\sqrt{6}}{\sqrt{D_l + D_{l+1}}}\right) \tag{28}$$

where D_l and D_{l+1} are the number of hidden units at the current and next layers. The offset / bias / intercept parameters should be initialized all to zero.

He Initialization For ReLU activations, the *He* initialization (He et al., 2016) is suitable:

$$\mathbf{W}_l \;\sim\; \texttt{Normal}\left(0,\; \frac{2}{D_l}\right) \tag{29}$$

where D_l are the number of units at the current hidden layer. Again, the bias / offset / intercept parameters should be set to zero.

4.5.4 Normalization Layers

While proper initialization ensures the weights are at a good setting at the start of gradient descent, nothing is stopping them from becoming numerically problematic during training. This is the problem *normalization* techniques such as *Batch Normalization* (BatchNorm) and *Layer Normalization* (LayerNorm) attempt to solve. These are typically applied before the activation function so that the input to the activation is scaled appropriately and does not saturate. Although with ReLU activations, BatchNorm has been found to be slightly more effective if applied after the activation. For both techniques defined below, assume we have a matrix of pre-activations $\mathbf{A} \in \mathbb{R}^{N \times D}$, with N being the number of instances / data points and D being the number of dimensions at this hidden layer.

BatchNorm BatchNorm applies the following normaliziation to each element $\mathrm{a}_{n,d}$ of $\mathbf{A}$:

$$\tilde{\mathrm{a}}_{n,d} \;=\; \texttt{BatchNorm}\left(\mathrm{a}_{n,d}; \mathbf{A}, \gamma_d, \beta_d\right) \;=\; \beta_d \;+\; \gamma_d \;\cdot\; \frac{a_{n,d} \;-\; \mathbb{E}[\mathrm{a}_{\cdot,d}]}{\sqrt{\mathrm{Var}[\mathrm{a}_{\cdot,d}] + \epsilon}}$$

where $\beta_d \in \mathbb{R}$ and $\gamma_d \in \mathbb{R}$, $d \in [1, D]$, are per-dimension parameters to be learned, $\epsilon \in \mathbb{R}^+$ is a small positive constant for numerical stability, $\mathbb{E}[\mathrm{a}_{\cdot,d}]$ is the mean (first moment) of the d-th dimension as computed empirically over the N samples contained within $\mathbf{A}$, and $\mathrm{Var}[\mathrm{a}_{\cdot,d}]$ is the variance (second moment) of the d-th dimension as computed empirically over the N samples contained within $\mathbf{A}$. In other words, BatchNorm performs its normalization by computing the mean and variance statistics for each column of $\mathbf{A}$. At test time, BatchNorm is 'turned off,' meaning that the γ and β parameters are no longer updated and $\mathbb{E}[\mathrm{a}_{\cdot,d}]$ and $\mathrm{Var}[\mathrm{a}_{\cdot,d}]$ are fixed to whatever their values are for the final training iteration.

LayerNorm LayerNorm, on the other hand, computes the mean and variance parameters across *the rows of* $\mathbf{A}$. Its transformation can be written as:

$$\bar{\mathrm{a}}_{n,d} \;=\; \mathtt{LayerNorm}\left(\mathrm{a}_{n,d}; \mathbf{A}, \gamma_d, \beta_d\right) \;=\; \beta_d \;+\; \gamma_d \;\cdot\; \frac{a_{n,d} \;-\; \mathbb{E}[\mathrm{a}_{n,\cdot}]}{\sqrt{\mathrm{Var}[\mathrm{a}_{n,\cdot}] + \epsilon}},$$

where $\beta_d \in \mathbb{R}$ and $\gamma_d \in \mathbb{R}$, $d \in [1, D]$, are per-dimension parameters to be learned, $\epsilon \in \mathbb{R}^+$ is (again) a small positive constant for numerical stability, $\mathbb{E}[\mathrm{a}_{n,\cdot}]$ is the mean (first moment) of the n pre-activation vector as computed empirically over the D *dimensions* contained within $\mathbf{a}_n$, and $\mathrm{Var}[\mathrm{a}_{n,\cdot}]$ is the variance (second moment) of the n-th per-activation vector as computed empirically over the D dimensions contained within $\mathbf{a}_n$. In other words, LayerNorm performs its normalization by computing the mean and variance statistics for each *row* of $\mathbf{A}$. At test time, the mean and variance of the per-instance pre-activations can be computed just the same as they were during training. The only difference is that the γ and β parameters will no longer be updated.

One may worry that applying these normalization techniques can discard valuable information about the scale of the pre-activations. This is not too much of a worry since the normalization is applied on the hidden units, which don't have a strong interpretation in the way that data does, so we are quite free to change the representations to make computation easier and the DNN will learn to propagate information within these constraints. Moreover, ReLU activations are 'scale free,' meaning that for $\mathrm{z} \in \mathbb{R}^+$, we have $\mathrm{z} \cdot \mathtt{ReLU}(\mathrm{a}) = \mathtt{ReLU}(\mathrm{z} \cdot \mathrm{a})$. This means that it is more meaningful to the DNN if the ReLU is inactive vs active rather than its precise output value.

4.6 Capacity Control

Like with polynomial regressors of high degrees, DNNs are extremely flexible, and sometimes we wish to control their complexity. Below I give two simple and popular ways to do that.

4.6.1 Weight Decay

$$\widetilde{\ell}(\mathbf{W}_0, \ldots, \mathbf{W}_L; \mathcal{D}, \lambda) \;=\; \ell(\mathbf{W}_0, \ldots, \mathbf{W}_L; \mathcal{D}) \;+\; \lambda \cdot \sum_{l=1}^{L} ||\mathbf{W}_l||_2^2\,,$$

4.6.2 Ensembling and Dropout

5 Stochastic, Adaptive Optimizers

5.1 Mini-Batch Gradient Descent

$$\begin{aligned} \nabla_{\mathbf{W}_l^t}\, \ell\left(\mathbf{W}_0^t, \ldots, \mathbf{W}_l^t, \ldots, \mathbf{w}_L^t; \mathcal{D}\right) \approx\; & \nabla_{\mathbf{W}_l^t}\, \ell\left(\mathbf{W}_0^t, \ldots, \mathbf{W}_l^t, \ldots, \mathbf{w}_L^t; \mathcal{B}\right) \\ &= \frac{1}{B} \sum_{b=1}^{B} \nabla_{\mathbf{W}_l^t}\, \ell\left(\mathbf{W}_0^t, \ldots, \mathbf{W}_l^t, \ldots, \mathbf{w}_L^t; (\boldsymbol{x}_b, \mathrm{y}_b)\right) \end{aligned} \tag{30}$$

5.2 Momentum

$$\mathbf{V}_l^t \;=\; \beta \cdot \mathbf{V}_l^{t-1} \;+\; \nabla_{\mathbf{W}_l^t}\, \ell\left(\mathbf{W}_0^t, \ldots, \mathbf{W}_l^t, \ldots, \mathbf{w}_L^t; \mathcal{B}\right)$$

$$\mathbf{W}_l^{t+1} \;=\; \mathbf{W}_l^t \;-\; \alpha \cdot \mathbf{V}_l^t$$

5.3 Adaptive Moment Estimation (Adam)

$$\mathbf{V}_l^t \;=\; \beta_1 \cdot \mathbf{V}_l^{t-1} \;+\; (1-\beta_1) \cdot \nabla_{\mathbf{W}_l^t}\, \ell\left(\mathbf{W}_0^t, \ldots, \mathbf{W}_l^t, \ldots, \mathbf{w}_L^t; \mathcal{B}\right)$$

$$\mathbf{S}_l^t \;=\; \beta_2 \cdot \mathbf{S}_l^{t-1} \;+\; (1-\beta_2) \cdot \Big(\nabla_{\mathbf{W}_l^t}\, \ell\left(\mathbf{W}_0^t, \ldots, \mathbf{W}_l^t, \ldots, \mathbf{w}_L^t; \mathcal{B}\right)\Big)^2$$

$$\hat{\mathbf{V}}_l^t = \frac{\mathbf{V}_l^t}{1-\beta_1^t}, \quad \hat{\mathbf{S}}_l^t = \frac{\mathbf{S}_l^t}{1-\beta_2^t}$$

$$\mathbf{W}_l^{t+1} \;=\; \mathbf{W}_l^t \;-\; \frac{\alpha}{\sqrt{\hat{\mathbf{S}}_l^t} + \epsilon} \odot \hat{\mathbf{V}}_l^t$$

6 Convolutional Neural Networks

7 Models for Sequential Data

7.1 Recurrent Neural Networks

Simple Recurrence

Long Short-Term Memory (LSTM)

Gated Recurrent Unit (GRU)

7.2 Overview of Architectures

One-to-Many

Many-to-One

Aligned Many-to-Many

Unaligned Many-to-Many

7.3 Unaligned Sequence-to-Sequence with Encoder-Decoder Architecture

8 Attention & Transformers

The sequence-to-sequence models that were introduced around 2014 have a problem that is somewhat analogous to the problem solved by LSTMs: the information that should be in the hidden state varies through time, and for the encoder-decoder architecture in particular, the information accessible to the decoder is bottlenecked by the encoder's last hidden representation. This motivated the *attention mechanism*: a way to dynamically look back through time and retrieve the information that is relevant to the current time step.

8.1 Attention

Attention is defined as follows. Let $\mathfrak{D} = \{(\mathbf{k}_1, \mathbf{v}_1), \dots, (\mathbf{k}_M, \mathbf{v}_M)\}$ be a 'database' of M-tuples representing key-value pairs. These key-value pairs will be real-valued vectors: $\mathbf{k}_m \in \mathbb{R}^{D_k}$, $\mathbf{v}_m \in \mathbb{R}^{D_v}$. Moreover, denote the query vector as $\mathbf{q} \in \mathbb{R}^{D_k}$. Attention over $\mathfrak{D}$ is written as:

$$\texttt{Attention}\,(\mathbf{q}; \mathfrak{D}) \;=\; \sum_{m=1}^{M} \alpha(\mathbf{q}, \mathbf{k}_m) \cdot \mathbf{v}_m \tag{31}$$

where $\{\alpha(\mathbf{q}, \mathbf{k}_1), \dots, \alpha(\mathbf{q}, \mathbf{k}_M)\}$ are the attention weights given to every value in the database. Usually they are constrained such that $0 \le \alpha(\mathbf{q}, \mathbf{k}_m) \le 1$ and $\sum_{m=1}^{M} \alpha(\mathbf{q}, \mathbf{k}_m) = 1$. The intuition is that these weights quantify the degree of similarity between the query and each key. There are two extreme cases on note. When $\alpha(\mathbf{q}, \mathbf{k}_m) = 1$, then the attention mechanism simply returns $\mathbf{v}_m$. When $\alpha(\mathbf{q}, \mathbf{k}_m) = 1/M \;\; \forall m$, attention returns the average value vector: $(1/M) \sum_{m=1}^{M} \mathbf{v}_m$. Thus we can think of attention as computed the 'weighted average' of the value vectors.

Computing the attention weights Computing the attention weights is commonly done by (i) taking the inner product between the query and key vectors, and then (ii) transforming those inner products by the softmax function—the same one we used as the inverse link for multi-class classification. We can write this computation as:

$$\begin{aligned} [\alpha(\mathbf{q}, \mathbf{k}_1), \dots, \alpha(\mathbf{q}, \mathbf{k}_M)] \;&=\; \texttt{softmax}\left(\frac{\mathbf{q}^T \mathbf{K}^T}{\sqrt{D_k}}\right) \\ \text{such that} \;\; \alpha(\mathbf{q}, \mathbf{k}_m) \;&=\; \frac{\exp\left\{\frac{\mathbf{q}^T \mathbf{k}_m}{\sqrt{D_k}}\right\}}{\sum_{j=1}^{M} \exp\left\{\frac{\mathbf{q}^T \mathbf{k}_j}{\sqrt{D_k}}\right\}} \end{aligned} \tag{32}$$

where $\mathbf{K} \in \mathbb{R}^{M \times D_k}$ is a matrix of all the key vectors stacked together such that each row of $\mathbf{K}$ corresponds to a key in the database. Dividing by $\sqrt{D_k}$ is just a normalization heuristic that controls the magnitude of the inner product as the query and key vectors grow in length / dimensionality.

Batched Computation Given a batch of N queries $\mathbf{Q} \in \mathbb{R}^{N \times D_k}$, which represents the query vectors stacked such that each query is a row, we can compute a batched version of attention as follows. This is how it is usually implemented in practice:

$$\texttt{Attention}\left(\mathbf{Q}; \mathfrak{D}\right) \ = \ \texttt{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{D_k}}\right)\mathbf{V}, \tag{33}$$

which will yield an output matrix of size $N \times D_v$, the attention mechanism carried out for all N queries. The linear algebra works out because, first, the product $\mathbf{Q}\mathbf{K}^T$ is of size $N \times M$. Second, applying the softmax to $\mathbf{Q}\mathbf{K}^T/\sqrt{D_k}$ should be done row-wise, such that each row sums to one. Application of the softmax leaves the dimensionality unchanged. We then multiply the $N \times M$ softmax-output with $\mathbf{V} \in \mathbb{R}^{M \times D_v}$, a matrix of all value vectors stacked horizontally. The inner dimensions already match, with the matrix multiplication resulting in a $(N \times D_v)$-sized output.

Multi-Head Attention We may wish to define several attention mechanisms, hoping that ensembling them produces better performance by the database of each representing distinct information. Yet, defining E independent attention mechanisms can be costly in terms of both memory and computation. *Multi-head attention* presents a simple way to balance the benefits of multiple attention mechanisms and these costs. Let $\{\mathbf{A}_1, \ldots, \mathbf{A}_E\}$ denote (batched) outputs of E attention mechanisms such that $\mathbf{a}_e$ is computed as:

$$\mathbf{A}_e \ = \ \texttt{softmax}\left(\frac{\left(\mathbf{Q}\mathbf{W}_{q,e}\right)\left(\mathbf{K}\mathbf{W}_{k,e}\right)^T}{\sqrt{D_{e,k}}}\right)\left(\mathbf{V}\mathbf{W}_{e,v}\right) \tag{34}$$

where $\mathbf{W}_{q,e} \in \mathbb{R}^{D_k \times D_{e,k}}$, $\mathbf{W}_{k,e} \in \mathbb{R}^{D_k \times D_{e,k}}$, and $\mathbf{W}_{v,e} \in \mathbb{R}^{D_v \times D_{e,v}}$ are trainable weight matrices for the e-th attention head. Thus multi-head attention takes a 'base' database and perturbs it by multiplying the queries, keys, and values by parameters that are specific to the e-th head. This drastically saves in memory costs since the overhead of multi-head attention does not scale with M. The final output of multi-head attention is computed by taking a linear transformation of all E head outputs:

$$\texttt{Multi-Head Attention}(\mathbf{A}_1, \ldots, \mathbf{A}_E; \mathbf{W}_o) \ = \ [\mathbf{A}_1, \ldots, \mathbf{A}_E]\mathbf{W}_o$$

where $[\mathbf{A}_1, \ldots, \mathbf{A}_E]$ represents a concatenation of all E heads and is of size $N \times \sum_e D_{e,v}$, and $\mathbf{W}_o$ is a matrix of trainable parameters with size $\sum_e D_{e,v} \times D_o$ and.

Self-Attention Given a sequence of M 'tokens' $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_m, \ldots, \mathbf{x}_M\}$, $\mathbf{x}_m \in \mathbb{R}^D$, we can define *self-attention* as an attention mechanism with the M tokens serving as all queries, keys, and values:

$$\texttt{Self-Attention}\left(\mathbf{Q} = \mathbf{X}; \mathfrak{D} = \{(\mathbf{x}_m, \mathbf{x}_m)\}_{m=1}^{M}\right) \ = \ \texttt{softmax}\left(\frac{\mathbf{X}\mathbf{X}^T}{\sqrt{D}}\right)\mathbf{X}. \tag{35}$$

We can think of self-attention as returning, for each token $\mathbf{x}_m$, a representation that averages the other tokens, with the most weight being places on the tokens that are most similar to $\mathbf{x}_m$ (according to inner their product). A notable aspect of self-attention is that it admits parallel computation. Thinking of the M tokens as a sequence, self-attention can quickly computing similarities across the sequence. An RNN, on the other hand, does not admit such parallel computations as the sequence must be absorbed token-by-token, requiring M

time steps.

8.2 Encoder-Decoder Architecture with Attention

Now let's return to the encoder-decoder sequence-to-sequence model, but now defining the decoder with an attention mechanism. Define a sequence of source tokens $\{\mathbf{x}_1, \ldots, \mathbf{x}_t, \ldots, \mathbf{x}_T\}$, which could be the prompt to a chatbot, and a sequence of target tokens $\{\mathbf{y}_1, \ldots, \mathbf{y}_{t'}, \ldots, \mathbf{y}_{T'}\}$, which could represent the chatbot's response. Recall for the previous version of the seq-to-seq encoder, the encoder output a final context vector that was just the last hidden state: $\mathbf{c} = \mathbf{h}_T$. This is the problem: all the information relevant to the decoder must already be represented within $\mathbf{h}_T$.

Instead, we can define a decoder with attention (that is, an *attentive decoder*) by giving it a context vector that changes over time, $\mathbf{c}_{t'}$, and is computed via attention:

$$\mathbf{c}_{t'} = \sum_{t=1}^{T} \alpha(\mathbf{q} = \mathbf{s}_{t'-1}, \mathbf{k}_t = \mathbf{h}_t) \cdot \mathbf{h}_t \tag{36}$$

where $\mathbf{s}_{t'-1}$ is the decoder's hidden state at the previous time step and $\mathbf{h}_t$ is the encoder's hidden state at time step t. Thus we see that the query will be the decoder's previous hidden state, the keys will be the encoder's hidden states, and the values will also be the encoder's hidden states. This formulation allows the decoder, via its previous hidden state $\mathbf{s}_{t'-1}$, to look into the encoder and retrieve hidden states that are the most similar to the decoder's current state. The context will then be used to compute the next hidden state of the decoder: $\mathbf{s}_{t'} = g(\mathbf{y}_{t'-1}, \mathbf{c}_{t'}, \mathbf{s}_{t'-1})$, where $g(\cdot)$ is again the equation for simple recurrence, an LSTM, a GRU, etc.

8.3 The Transformer

Given the success of attention in seq-to-seq models built from recurrent networks, Vaswani et al. (2017) asked: *is attention all you need?* That is, do we actually need recurrent NNs in our sequence-to-sequence models? Can we just building everything from (self) attention layers? This will certainly have a benefit for computational efficiency, given that self-attention can be computed in parallel. The answer is *yes!*: Vaswani et al. (2017) introduced an encoder-decoder architecture that does not use RNNs, instead using multi-head self-attention. This architecture is called the *Transformer*, and we'll go into its inner workings below.

8.3.1 Encoder

Positional Encodings Before we describe the architecture, we first need to address a flaw in self-attention. Like fully-connected layers, self-attention layers are agnostic to the position of each input. That is, if you train two self-attention layers, each with a different permutation of the tokens, gradient descent (up to optimization pathologies) can train each self-attention layer such that they have the same output. This is not true for convolutional layers and recurrent units, which encode a notion of space and time, respectively. Yet if we wish to apply self-attention to data (such as language) for which its order / time / sequence matters, then this is a problem. A somewhat hack-y but commonly successful way to fix

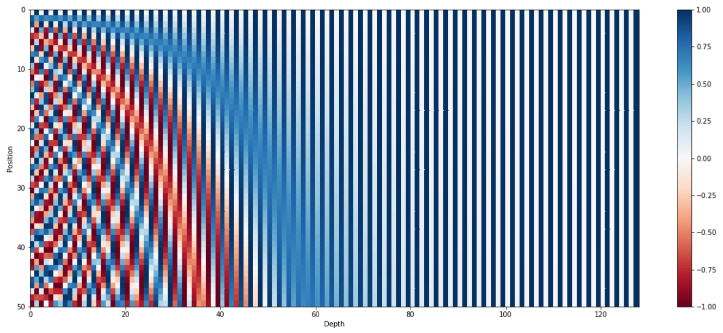


Figure 16: *Visualization of Positional Encodings.* The above heat maps shows the values of **P**, the matrix of positional encodings. Image reproduced from Kazemnejad (2019).

this is via *positional encodings*. We want to add some constant value to the tokens such that their position in the sequence can be known (and used) by the model. Given a T-length sequence $\mathbf{X} \in \mathbb{R}^{T \times D_x}$, its positional encoders are another matrix $\mathbf{P} \in \mathbb{R}^{T \times D_x}$. The two are added together to form a new representation of the sequence:

$$\tilde{\mathbf{X}} \;=\; \mathbf{X} \;+\; \mathbf{P}, \qquad \text{s.t.} \quad p_{t,d2} = \sin\left(\frac{t}{10,000^{2d/D_x}}\right), \quad p_{t,d2+1} = \cos\left(\frac{t}{10,000^{2d/D_x}}\right).$$

A visualization of **P** is provided in Figure 16. The idea is that the sine and cosine functions have a periodicity that changes according to the time index t and the dimension d in order to give a unique positional marker to each value of **X**. One could imagine more naive approaches—such as populating **P** with the indicies directly—but this would drastically vary the scale of **P** and likely overwhelm the information in **X**. The sine and cosine functions do not have this problem since they are bounded in [-1, 1].

Multi-Head Self-Attention

LayerNorm & Skip-Connections

8.3.2 Decoder

8.3.3 Summary of Full Architecture

9 Autoencoders

10 Deep Generative Models

Bibliography

Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the 13th International Conference on Artificial Intelligence and Statistics*, pages 249–256, 2010.

Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image

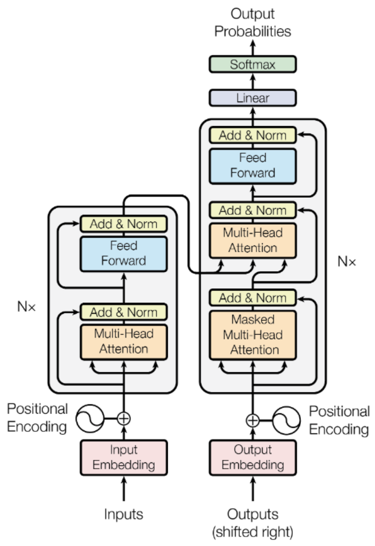


Figure 17: *Transformer Architecture.* Image reproduced from Vaswani et al. (2017).

recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 770–778, 2016.

Amirhossein Kazemnejad. Transformer architecture: The positional encoding. *kazemnejad.com*, 2019. URL `https://kazemnejad.com/blog/transformer_architecture_positional_encoding/`.

Matus Telgarsky. Benefits of depth in neural networks. In *Conference on Learning Theory*, pages 1517–1539. PMLR, 2016.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is All You Need. *Advances in Neural Information Processing Systems*, 30, 2017.