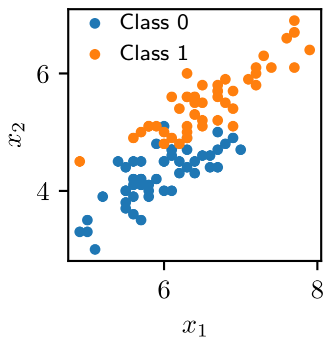
EMAT31530, Part 3: Linear classification

Laurence Aitchison

1 An loss function for classification error

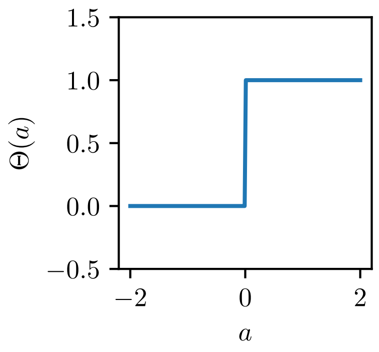
Doing regression by gradient descent is pretty straightforward, because all the distances between the real data, y_i and targets, $\hat{y}(\mathbf{x}_i)$ are continuous and differentiable (almost everywhere). We can therefore define the loss as the sum-of-squared-errors. However, things become much harder when it comes to classification. In classification, we take inputs (here, x_0 and x_1) and try to determine which of a fixed number of discrete classes the datapoint belongs to:



Looking back at regression, the starting point was to define a prediction-function which takes x_0 and x_1 as inputs, gives predictions as outputs, and has parameters that you can tweak. For instance, if we have two classes (class 0 and class 1) we could use,

$$\hat{y}_{\mathbf{w}}(\mathbf{x}) = \Theta(\mathbf{x}\mathbf{w}) \tag{1}$$

where $\Theta(a)$ is the Heaviside step function, which returns 0 when the input argument is negative ($a < 0$), and returns 1 otherwise.



As $\hat{y}_{\mathbf{w}}(\mathbf{x})$ returns 0 or 1, and is parameterised in terms of a weight-vector, $\mathbf{w}$, we can treat the output from $\hat{y}_{\mathbf{w}}(\mathbf{x})$ as a prediction of the class (class 0 or class 1). We can then optimize the weights in order to give better predictions.

Key problem: what loss function to use? The obvious choice is classification error:

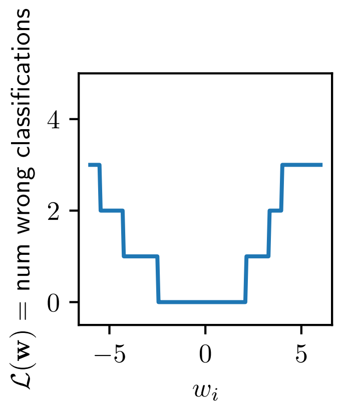
$$\mathcal{L}(\mathbf{w}) = \text{number of wrong classifications} \tag{2}$$

$$\mathcal{L}(\mathbf{w}) = \sum_{i=1}^{N} \text{different}(y_i, \hat{y}_{\mathbf{w}}(\mathbf{x}_i)) \tag{3}$$

where,

$$\text{different}(y, \hat{y}) = \begin{cases} 0 & \text{if } y = \hat{y} \\ 1 & \text{otherwise} \end{cases} \tag{4}$$

The problem is that the number of wrong classifications is always an integer (0, 1, 2 etc.)



This function isn't continuous. In fact, the gradient is flat (almost) everywhere! So we can't do gradient descent!

Instead, we need an alternative loss, which is continuous and differentiable (almost everywhere). How do we get such a loss? We're going to set up classification as maximum-likelihood in a probabilistic model. To introduce the basic idea, we first need to take a detour through maximum-likelihood for coin flipping...

2 Maximum likelihood for coin flipping

When tossing a biased coin, our random variable, x, can take on values of 1 or 0. In coin flipping, $x = 1$ corresponds to heads and $x = 0$ corresponds to tails. In classification (which we'll get on to), $x = 1$ corresponds to class 1, while $x = 0$ corresponds to class 0 (logically enough). Because this is a biased coin, there is a parameter, p, which controls the probability of heads vs tails (or class 0 vs class 1). Specifically,

$$\mathrm{P}\left(x = 1|p\right) = p \tag{5a}$$
$$\mathrm{P}\left(x = 0|p\right) = (1 - p). \tag{5b}$$

The probability of heads *or* tails adds to 1,

$$1 = \mathrm{P}\left(x = 1|p\right) + \mathrm{P}\left(x = 0|p\right) = p + (1 - p). \tag{6}$$

For the maths, it'll be super-useful to have a form for the probability which takes x as an input (rather than just having two different forms for $x = 1$ and $x = 0$, as in Eq. 5a). We can write this as,

$$\mathrm{P}\left(x|p\right) = p^x (1 - p)^{1-x} \tag{7}$$

Note that there isn't really a "derivation" for this. It just happens to be an expression gives the right answer when we substitute $x = 1$ or $x = 0$,

$$\mathrm{P}\left(x = 1|p\right) = p^1 (1 - p)^0 = p \times 1 = p, \tag{8}$$
$$\mathrm{P}\left(x = 0|p\right) = p^0 (1 - p)^1 = 1 \times (1 - p) = (1 - p). \tag{9}$$

Now, lets consider a classic problem: we have a dataset of tosses from a biased coin, $x_1, \dots, x_N$, and our goal is to estimate the probability, p, that generated these coin tosses. To do that, we find the p that makes the observed data most probable (i.e. maximum likelihood),

$$\mathcal{L}(p) = \log \mathrm{P}\left(x_1, \dots, x_N|p\right) \tag{10}$$

Specifically, we maximize the maximize the *log* probability. As the logarithm is a monotonically increasing function, this gives the right answer. Specifically, the value of p at which $\mathrm{P}(x_1, \dots, x_N|p)$ is maximized is the same as the value of p at which $\log \mathrm{P}(x_1, \dots, x_N|p)$ is maximized,

$$p^* = \underset{p}{\operatorname{argmax}} \, \mathrm{P}(x_1, \dots, x_N|p) = \underset{p}{\operatorname{argmax}} \log \mathrm{P}(x_1, \dots, x_N|p) \,. \tag{11}$$

Moreover, using logarithms is very useful for two reasons discussed in the Prerequisites:

- Logarithms turn products into sums, and its much easier to differentiate sums than products.
- Raw probabilities can be very large or very small once we take products over a large number of datapoints. That can cause numerical issues (i.e. over/underflow).

Now, we assume that coin tosses are independent, so $\mathrm{P}(x_1, \dots, x_N|p) = \prod_{i=1}^{N} \mathrm{P}(x_i|p)$, and

$$\mathcal{L}(p) = \log \prod_{i=1}^{N} \mathrm{P}(x_i|p) \tag{12}$$

The log turns the product into a sum,

$$\mathcal{L}(p) = \sum_{i=1}^{N} \log \mathrm{P}(x_i|p) \tag{13}$$

Substituting the log-probability for tossing a biased coin (Eq. 7),

$$\mathcal{L}(p) = \sum_{i=1}^{N} \log \left(p^{x_i} (1-p)^{1-x_i}\right) \tag{14}$$

The log again turns the product into a sum,

$$\mathcal{L}(p) = \sum_{i=1}^{N} \left(\log\left(p^{x_i}\right) + \log\left((1-p)^{1-x_i}\right)\right) \tag{15}$$

And the log turns powers into products,

$$\mathcal{L}(p) = \sum_{i=1}^{N} \left(x_i \log p + (1-x_i) \log(1-p)\right) \tag{16}$$

Applying the sum to each term separately, and noting that p is independent of i,

$$\mathcal{L}(p) = \underbrace{\left(\sum_{i=1}^{N} x_i\right)}_{\text{a constant number}} \log p + \underbrace{\left(N - \sum_{i=1}^{N} x_i\right)}_{\text{another constant number}} \log(1-p) \tag{17}$$

So we can treat this as,

$$\mathcal{L}(p) = a \log p + b \log(1 - p) \tag{18}$$

where,

$$a = \sum_{i=1}^{N} x_i \qquad\qquad b = N - \sum_{i=1}^{N} x_i \tag{19}$$

Notice that we've extracted the sums over data into constant multipliers, a and b. This has simplified the loss, $\mathcal{L}(p)$, dramatically. So finding the maximum-likelihood value for p is now a tractable, univariate calculus problem!

Moreover, notice that despite the data, x_i, being discrete, the objective is a continuous and differentiable function of the parameter, p. As such, we can find the most likely probability, p^*, by doing gradient descent! If we did that, we'd end up with the sensible answer (see Exercises),

$$p^* = \tfrac{1}{N} \sum_{i=1}^{N} x_i. \tag{20}$$

3 A smooth, maximum-likelihood objective for binary classification

In the previous section, we saw that we can get a continuous, differentiable loss for discrete data by setting the problem up as maximum-likelihood. So how do we apply the same idea to get a differentiable loss for binary classification? The basic idea is that we first do the usual thing of taking the dot-product of the input vector with some weights,

$$\ell = f_{\mathbf{w}}(\mathbf{x}_i) = \mathbf{x}_i \mathbf{w}. \tag{21}$$

The problem is that the result, ℓ, could be from $-\infty$ to ∞, while we want a probability, between 0 and 1. If the input features, $\mathbf{x}_i$ are bounded (e.g. they could be bounded between 0 and 1), then we could find bounds on $\mathbf{w}$ that ensure that the result of this dot-product is always between 0 and 1. But that sounds hard. And it won't work in more complex architectures like neural networks, where we can't find the required constraints on the parameters.

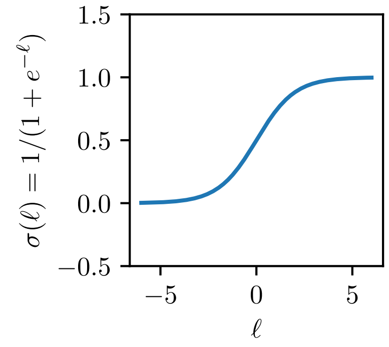
As such, we need to do something else. In particular, we don't take take ℓ to be the probability; instead, we take ℓ to be the "logits", which could be anything from $-\infty$ to ∞. We can transform from the logits to the probability by applying the "logistic sigmoid" function (or sigmoid[1] for short), denoted $p = \sigma(\ell)$. The

[1]From Wikipedia: a sigmoid function is a mathematical function having a characteristic "S"-shaped curve or sigmoid curve, so the logistic sigmoid is a specific function with an "S"-shaped curve.

sigmoid function takes any ℓ (from $-\infty$ to ∞) and returns a number from 0 to 1:

$$\sigma(\ell) = \frac{1}{1+e^{-\ell}} \tag{22}$$

Graphically, the sigmoid looks like,



Now, we can use the logistic sigmoid to give the probability for class 1 for input $\mathbf{x}$,

$$p_{\mathbf{w}}(\mathbf{x}) = \sigma(f_{\mathbf{w}}(\mathbf{x})) \tag{23}$$

$$\mathrm{P}\left(y=1|\mathbf{x}_i\right) = p_{\mathbf{w}}(\mathbf{x}_i). \tag{24}$$

4 A smooth, maximum-likelihood objective for multi-class classification

In the previous section, we were looking at classification problems where there's two possible classes. But almost always, there's multiple classes. For instance, we might want to classify handwritten digits as 0–9, in which case, there's ten classes.

Again, to train a model for multi-class classification, we need an objective. Classification error (Eq. 2) also makes sense in the multi-class setting, but again, can't be used as an objective as it is non-differentiable. As in the binary classification setting, we need an objective that is a continuous, differentiable function of the parameters (weights). To get such an objective, we're again going to consider a probability distribution over classes. In multi-class classification, the probability distribution is over the integers 0–9 (or in general, over the integers 0–C, where C is the number of classes), instead of just over $\{0, 1\}$, as in binary classification in the previous section.

The first problem is that previously, we just had one scalar logits for each datapoint, $\ell = f_{\mathbf{w}}(\mathbf{x}_i) = \mathbf{x}_i \mathbf{w}$, and we can't turn just a single scalar into a probability distribution over multiple classes. Instead, in multi-class classification, we use a whole *vector* of logits, $\boldsymbol{\ell}$, for each datapoint. This vector has length C, so there's one logits for each class,

$$\underbrace{\boldsymbol{\ell}}_{1\times C} = \mathbf{f}_{\mathbf{W}}(\mathbf{x}) = \underbrace{\mathbf{x}}_{1\times D} \underbrace{\mathbf{W}}_{D\times C}. \tag{25}$$

Note that we get a vector of logits by replacing the usual $D \times 1$ weight vector, $\mathbf{w}$, with a $D \times C$ weight matrix, $\mathbf{W}$.

Now we have a vector of C logits, $\boldsymbol{\ell}$. But we are again, left with the problem of how to set the probability for each class, given that the individual logits values can be anywhere from $-\infty$ to ∞. We use the so-called "softmax" function,

$$p_c(\mathbf{x}) = \text{softmax}\left(\boldsymbol{\ell} = \mathbf{f}_{\mathbf{W}}(\mathbf{x})\right) = \frac{e^{\ell_c}}{\sum_{c'} e^{\ell_{c'}}} \tag{26}$$

where c is the class label. Remember that the probabilities must all be non-negative and sum to 1. Non-negativity is straightforward, as the logits are real, so all the exponentials are non-negative. Further, we can check that the resulting probabilities sum to 1,

$$\sum_c p_c(\mathbf{x}) = \frac{\sum_c e^{\ell_c}}{\sum_{c'} e^{\ell_{c'}}} = 1 \tag{27}$$

as the numerator and denominator are the same.

5 Maximum likelihood vs cross-entropy loss

While the maximum likelihood perspective is the right way to motivate the binary and multi-class classification objectives, deep learning types don't use this viewpoint. Instead, they typically say that in classification, we use a "cross entropy loss". Mathematically, this is just the negative log-probability, so it is equivalent to the maximum likelihood framework above (maximizing the log-probability is equivalent to minimizing the negative log-probability),

$$\mathcal{L} = -\sum_i \log p_{y_i}(\mathbf{x}_i). \tag{28}$$

Remember, this is a sum over datapoints, and p_{y_i} picks out the predicted probability of the actual target class.

Why do they call it a "cross-entropy"? If we look at the general definition of cross-entropy, it is a function of *two* probability distributions,

$$H(q, p) = -\sum_c q_c \log p_c \tag{29}$$

where the sum is now over image classes, rather than datapoints. If we make a couple of choices, this becomes equal to the loss for a single datapoint. Specifically, we take q_c to be a distribution which puts all its mass at the true target class, and we take p_c to be our predicted probabilties,

$$q_c = \delta_{c,y_i} \tag{30}$$

$$p_c = p_c(\mathbf{x}_i). \tag{31}$$

Then,

$$H(q, p) = -\sum_c q_c \log p_c \tag{32}$$

$$H(q, p) = -\sum_c \delta_{c,y_i} \log p_c(\mathbf{x}_i). \tag{33}$$

The Kronecker-delta picks out the element of the sum for which $c = y_i$,

$$H(q, p) = -\log p_{y_i}(\mathbf{x}_i). \tag{34}$$

And this is exactly our loss for the ith datapoint. We can simplify this loss, and rename it to a form that is more usual in deep learning. In the multiclass setting:

$$\text{cross entropy}(\boldsymbol{\ell}, y) = -\ell_y + \log \sum_c \exp(\ell_c). \tag{35}$$

where y is the target class label, and $\boldsymbol{\ell}$ is the vector of logits that would usually come from the neural network. In the binary classification setting,

$$\text{cross entropy}(\ell, y) = \begin{cases} \log(1 + e^{-\ell}) & \text{if } y = 1 \\ \log(1 + e^{\ell}) & \text{if } y = 0 \end{cases} \tag{36}$$

6 Gradient descent for binary classification

Now, we have discussed the loss function for binary and multiclass classification, we need to think about how to actually use this objective to optimize the weights. We'll focus on the binary setting (to keep things slightly simpler) and use gradient descent (i.e. compute the gradient of the loss wrt the weights). But first-things-first, it turns out to be useful to rearrange the objective itself into a simpler form. Remember that the objective is the log-probability of class-labels under our model,

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \log \mathrm{P}\left(y_i | \mathbf{w}, \mathbf{x}_i\right) \tag{37}$$

Using the coin-flipping probability from Eq. (7)

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \log(p_{\mathbf{w}}(\mathbf{x}_i)^{y_i}(1 - p_{\mathbf{w}}(\mathbf{x}_i))^{1-y_i}) \tag{38}$$

The log turns products into sums,

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \left[\log(p_{\mathbf{w}}(\mathbf{x}_i)^{y_i}) + \log((1 - p_{\mathbf{w}}(\mathbf{x}_i))^{1-y_i})\right]. \tag{39}$$

Next, the log turns powers into products, (mirroring the coin flipping result in Eq. 16)

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \left[y_i \log(p_{\mathbf{w}}(\mathbf{x}_i)) + (1 - y_i) \log((1 - p_{\mathbf{w}}(\mathbf{x}_i)))\right]. \tag{40}$$

Substituting the value of $p_{\mathbf{w}}(\mathbf{x}_i)$ (Eq. 23), (keeping the sigmoid in the first term, but using the explicit form for the sigmoid in the second term),

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \left[y_i \log \sigma(\mathbf{w}\mathbf{x}_i) + (1 - y_i) \log \left(1 - \frac{1}{1 + e^{-\mathbf{w}\mathbf{x}_i}}\right)\right] \tag{41}$$

Using, $1 = (1 + e^{-\mathbf{w}\mathbf{x}_i})/(1 + e^{-\mathbf{w}\mathbf{x}_i})$,

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \left[y_i \log \sigma(\mathbf{w}\mathbf{x}_i) + (1 - y_i) \log \left(\frac{1 + e^{-\mathbf{w}\mathbf{x}_i}}{1 + e^{-\mathbf{w}\mathbf{x}_i}} - \frac{1}{1 + e^{-\mathbf{w}\mathbf{x}}}\right)\right], \tag{42}$$

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \left[y_i \log \sigma(\mathbf{w}\mathbf{x}_i) + (1 - y_i) \log \frac{1 + e^{-\mathbf{w}\mathbf{x}_i} - 1}{1 + e^{-\mathbf{w}\mathbf{x}_i}}\right], \tag{43}$$

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \left[y_i \log \sigma(\mathbf{w}\mathbf{x}_i) + (1 - y_i) \log \frac{e^{-\mathbf{w}\mathbf{x}_i}}{1 + e^{-\mathbf{w}\mathbf{x}_i}}\right]. \tag{44}$$

Multiplying the numerator and denominator of the fraction by $e^{\mathbf{w}\mathbf{x}_i}$,

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \left[y_i \log \sigma(\mathbf{w}\mathbf{x}_i) + (1 - y_i) \log \frac{(e^{-\mathbf{w}\mathbf{x}_i})e^{\mathbf{w}\mathbf{x}_i}}{(1 + e^{-\mathbf{w}\mathbf{x}_i})e^{\mathbf{w}\mathbf{x}_i}}\right], \tag{45}$$

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \left[y_i \log \sigma(\mathbf{w}\mathbf{x}_i) + (1 - y_i) \log \frac{1}{1 + e^{\mathbf{w}\mathbf{x}_i}}\right] \tag{46}$$

Finally, notice that the final term is another sigmoid,

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} [y_i \log \sigma(\mathbf{w}\mathbf{x}_i) + (1 - y_i) \log \sigma(-\mathbf{w}\mathbf{x}_i)] \tag{47}$$

Now, we want to differentiate this objective wrt a weight. This is a bigger, more complicated and scarier function than any we've seen before. But things are going to be okay if we write out a stepwise process for computing the function, and apply the chain rule everywhere! Specifically, we're going to compute the chain-rule using the following sequence of computations. We're first going to compute the product of features and weights,

$$a^i = \sum_{\lambda} w_\lambda X_{\lambda i} \tag{48}$$

Then we're going to apply the sigmoid to a^i (for the first term) and $-a^i$ (for the second term),

$$s_0^i = \sigma(-a^i) \qquad\qquad s_1^i = \sigma(a^i), \tag{49}$$

Then we're going to apply the log to both of the resulting probabilities

$$l_0^i = \log s_0^i \qquad\qquad l_1^i = \log s_1^i, \tag{50}$$

Then we're going to compute the objective,

$$\mathcal{L}(\mathbf{w}) = -\sum_{i=1}^{N} \left[y_i l_1^i + (1 - y_i) l_0^i \right]. \tag{51}$$

The gradient is therefore,

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_\nu} = -\sum_{i=1}^{N} \left[y_i \frac{\partial l_1^i}{\partial w_\nu} + (1 - y_i) \frac{\partial l_0^i}{\partial w_\nu} \right]. \tag{52}$$

Applying the chain rule,

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_\nu} = -\sum_{i=1}^{N} \left[y_i \frac{\partial l_1^i}{\partial s_1^i} \frac{\partial s_1^i}{\partial a_1^i} \frac{\partial a^i}{\partial w_\nu} + (1 - y_i) \frac{\partial l_0^i}{\partial s_0^i} \frac{\partial s_0^i}{\partial a^i} \frac{\partial a_0^i}{\partial w_\nu} \right]. \tag{53}$$

Now, we need to compute the individual derivatives in the chain rule. To start, we look at the linear combination of weights and features,

$$\frac{\partial a^i}{\partial w_\nu} = \frac{\partial}{\partial w_\nu} \left[\sum_{\lambda} w_\lambda X_{\lambda i} \right] = \sum_{\lambda} \delta_{\nu\lambda} X_{\lambda i} = X_{\nu i} \tag{54}$$

(see the last exercise of "Mathematical Prerequisites" for details). For the sigmoid,

$$\frac{\partial s_0^i}{\partial a^i} = \frac{\partial \sigma(-a_0^i)}{\partial a_0^i} = -\sigma(a^i)\sigma(-a^i) \tag{55}$$

$$\frac{\partial s_1^i}{\partial a^i} = \frac{\partial \sigma(a_1^i)}{\partial a_1^i} = \sigma(a^i)\sigma(-a^i) \tag{56}$$

(You're going to work out the form for this derivative in the exercises.) And finally the log,

$$\frac{\partial l_0^i}{\partial s_0^i} = \frac{\partial \log s_0^i}{\partial s_0^i} = \frac{1}{s_0^i} = \frac{1}{\sigma(-a^i)} \tag{57}$$

$$\frac{\partial l_1^i}{\partial s_1^i} = \frac{\partial \log s_1^i}{\partial s_1^i} = \frac{1}{s_0^i} = \frac{1}{\sigma(a^i)} \tag{58}$$

Putting everything back together, we get something reasonably simple,

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_\nu} = -\sum_{i=1}^{N} \left[y_i \frac{\partial l_1^i}{\partial s_1^i} \frac{\partial s_1^i}{\partial a_1^i} \frac{\partial a_1^i}{\partial w_\nu} + (1-y_i) \frac{\partial l_0^i}{\partial s_0^i} \frac{\partial s_0^i}{\partial a_0^i} \frac{\partial a^i}{\partial w_\nu} \right] \tag{59}$$

$$\frac{\partial \mathcal{L}(\mathbf{w})}{\partial w_\nu} = -\sum_{i=1}^{N} X_{i\nu} \left[y_i \frac{\sigma(a^i)\sigma(-a^i)}{\sigma(a^i)} + \frac{-\sigma(a^i)\sigma(-a^i)}{\sigma(-a^i)}(1-y_i) \right] \tag{60}$$

$$= -\sum_{i=1}^{N} X_{i\nu} \left[y_i \sigma(-a^i) - (1-y_i)\sigma(a^i) \right] \tag{61}$$

While we *could* compute these derivatives, it was pretty darn tedious. It would be great if we could get the computer to do it for us! Next week, we're going to look at neural networks themselves, and see that PyTorch somehow magically computes the gradients for us. The week after, we're going to see precisely *how* PyTorch does this magic!

7 Exercises

Exercise 1. *Find the maximum-likelihood probability for a biased coin, given a dataset of N tosses, x_i, with $x_i = 0$ representing tails and $x_i = 1$ representing heads. Start from the expression for the log-likelihood in Eq. 17, and solve for the value of p at which the gradient of the log-likelihood is zero.*

Exercise 2. *Show that the gradient of a sigmoid is*

$$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)\sigma(-a) \tag{62}$$

8 Answers

Answer 1. *Now, we find the maximum likelihood value of p by solving for where the gradient is zero,*

$$0 = \frac{\partial \mathcal{L}(p)}{\partial p} \tag{63}$$

$$0 = \left(\sum_{i=1}^{N} x_i\right) \frac{\partial \log p}{\partial p} + \left(N - \sum_{i=1}^{N} x_i\right) \frac{\partial \log(1-p)}{\partial p} \tag{64}$$

It is a standard result that,

$$\frac{\partial \log p}{\partial p} = \frac{1}{p}. \tag{65}$$

But to compute the other derivative, we need to apply the chain rule. Specifically, we use,

$$u = (1-p) \tag{66}$$

$$y = \log(1-p) = \log u. \tag{67}$$

Thus,

$$\frac{\partial y}{\partial p} = \frac{\partial y}{\partial u}\frac{\partial u}{\partial p} \tag{68}$$

$$= \frac{\partial \log u}{\partial u}\frac{\partial}{\partial p}[1-p] \tag{69}$$

$$= \frac{1}{u} \times (-1) \tag{70}$$

$$= -\frac{1}{1-p} \tag{71}$$

Substituting in the derivatives,

$$0 = \left(\sum_{i=1}^{N} x_i\right) \frac{1}{p} - \left(N - \sum_{i=1}^{N} x_i\right) \frac{1}{1-p} \tag{72}$$

To get the p's out of the denominators, we multiply both sides by $p(1-p)$,

$$0 = \left(\sum_{i=1}^{N} x_i\right) \frac{p(1-p)}{p} - \left(N - \sum_{i=1}^{N} x_i\right) \frac{p(1-p)}{1-p} \tag{73}$$

And cancel terms,

$$0 = \left(\sum_{i=1}^{N} x_i\right) (1-p) - \left(N - \sum_{i=1}^{N} x_i\right) p. \tag{74}$$

Now, we separate out all the terms,

$$0 = \left(\sum_{i=1}^{N} x_i\right) - \left(\sum_{i=1}^{N} x_i\right) p - Np + \left(\sum_{i=1}^{N} x_i\right) p. \tag{75}$$

The plus and minus $\left(\sum_{i=1}^{N} x_i\right) p$ *terms cancel,*

$$0 = \left(\sum_{i=1}^{N} x_i\right) - Np. \tag{76}$$

Now, we add Np *to both sides,*

$$Np = \sum_{i=1}^{N} x_i \tag{77}$$

And divide both sides by N*,*

$$p = \tfrac{1}{N} \sum_{i=1}^{N} x_i. \tag{78}$$

Answer 2. *Find the gradient of a sigmoid,*

$$\frac{\partial \sigma(a)}{\partial a} = \frac{\partial}{\partial a} \frac{1}{1 + e^{-a}} \tag{79}$$

Apply the chain rule, with $u = 1 + e^{-a}$*,*

$$\sigma(a) = u^{-1} \tag{80}$$

$$\frac{\partial \sigma(a)}{\partial a} = \frac{\partial u^{-1}}{\partial u} \frac{\partial u}{\partial a} \tag{81}$$

The individual derivatives are,

$$\frac{\partial u^{-1}}{\partial u} = -u^{-2} \tag{82}$$

$$\frac{\partial u}{\partial a} = \frac{\partial}{\partial a} \left[1 + e^{-a}\right] = -e^{-a} \tag{83}$$

Putting everything back together,

$$\frac{\partial \sigma(a)}{\partial a} = \frac{\partial u^{-1}}{\partial u} \frac{\partial u}{\partial a} = \frac{e^{-a}}{u^2} = \frac{e^{-a}}{(1 + e^{-a})^2} \tag{84}$$

Split up the denominator,

$$\frac{\partial \sigma(a)}{\partial a} = \frac{1}{1 + e^{-a}} \frac{e^{-a}}{1 + e^{-a}} \tag{85}$$

The $1/(1+e^{-a})$ *term is a sigmoid. For the other term, we multiply the numerator and denominator by* e^a*,*

$$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)\frac{1}{(1+e^a)} \tag{86}$$

Now, $1/(1+e^a)$ *term is also a sigmoid, just with the argument negated,*

$$\frac{\partial \sigma(a)}{\partial a} = \sigma(a)\sigma(-a). \tag{87}$$