Lecture 3, Part 2: Training a Classifier

Roger Grosse

1 Introduction

Now that we've defined what binary classification is, let's actually train a classifier. We'll approach this problem in much the same way as we did linear regression: define a model and a cost function, and minimize the cost using gradient descent. The one thing that makes the classification case harder is that it's not obvious what loss function to use. We can't just use the classification error itself, because the gradient is zero almost everywhere! Instead, we'll define a *surrogate loss function*, i.e. an alternative loss function which is easier to optimize.

1.1 Learning Goals

- Understand why classification error and squared error are problematic cost functions for classification.
- Know what cross-entropy is and understand why it can be easier to optimize than squared error (assuming a logistic activation function).
- Be able to derive the gradient descent updates for all of the models and cost functions mentioned in this lecture and to implement the learning algorithms in Python.
- Know what hinge loss is, and how it relates to cross-entropy loss.
- Understand how binary logistic regression can be generalized to multiple variables.
- Know what it means for a function to be convex, how to check convexity visually, and why it's important for optimization.
  - Algebraically proving functions to be convex is beyond the scope of this course.
- Know how to check the correctness of gradients using finite differences.

2 Choosing a cost function

Recall the setup from the previous lecture. We have a set of training examples $\{(\mathbf{x}^{(i)}, t^{(i)})\}_{i=1}^N$, where the $\mathbf{x}^{(i)}$ are vector-valued inputs, and $t^{(i)}$ are binary-valued targets. We would like to learn a binary linear classifier, where we compute a linear function of $\mathbf{x}^{(i)}$ and threshold it at zero:

$$y = \begin{cases} 1 & \text{if } \mathbf{w}^\top \mathbf{x} > 0 \\ 0 & \text{otherwise.} \end{cases} \tag{1}$$

In the last lecture, our goal was to correctly classify every training example. But this might be impossible if the dataset isn't linearly separable. Even if it's possible to correctly classify every training example, it may be undesirable since then we might just overfit!

How can we define a sensible learning criterion when the dataset isn't linearly separable? One natural criterion is to minimize the number of misclassified training examples. We can formalize this with the **classification error** loss, or the **0-1 loss**:

$$\mathcal{L}_{0-1}(y,t) = \begin{cases} 0 & \text{if } y = t \\ 1 & \text{otherwise.} \end{cases} \tag{2}$$

As always, the cost function is just the loss averaged over the training examples; in this case, that corresponds to the **error rate**, or fraction of misclassified examples. How do we make this small?

2.1 Attempt 1: 0-1 loss

As a first attempt, let's try to minimize 0-1 loss directly. In order to compute the gradient descent updates, we need to compute the partial derivatives $\partial\mathcal{L}_{0-1}/\partial w_j$. Rather than mechanically deriving this derivative, let's think about what it means. It means, how much does $\mathcal{L}_{0-1}$ change if you make a very small change to w_j? As long as we're not on the classification boundary, making a small enough change to w_j will have *no effect* on $\mathcal{L}_{0-1}$, because the prediction won't change. This implies that $\partial\mathcal{L}_{0-1}/\partial w_j = 0$, as long as we're not on the boundary. Gradient descent will go nowhere. (If we are on the boundary, the cost is discontinuous, which certainly isn't any better!) OK, we certainly can't optimize 0-1 loss with gradient descent.

Near the end of the course, when we discuss reinforcement learning, we'll see an algorithm which can minimize 0-1 loss directly. It's nowhere near as efficient as gradient descent, though, so we still need the techniques of this lecture!

2.2 Attempt 2: linear regression

Since that didn't work, let's try using something we already know: linear regression. Recall that this assumes a linear model and the squared error loss function:

$$y = \mathbf{w}^\top \mathbf{x} + b \tag{3}$$

$$\mathcal{L}_{\text{SE}}(y,t) = \frac{1}{2}(y-t)^2 \tag{4}$$

We've already seen two ways of optimizing this: gradient descent, and a closed-form solution. But does it make sense for classification? One obvious problem is that the predictions are real-valued rather than binary. But that's OK, since we can just pick some scheme for binarizing them, such as thresholding at $y = 1/2$. When we replace a loss function we trust with another one we trust less but which is easier to optimize, the replacement one is called a **surrogate loss function**.

But there's still a problem. Suppose we have a positive example, i.e. $t = 1$. If we predict $y = 1$, we get a cost of 0, whereas if we make the wrong prediction $y = 0$, we get a cost of $1/2$; so far, so good. But suppose we're really confident that this is a positive example, and predict $y = 9$. Then we pay a cost of $\frac{1}{2}(9-1)^2 = 32$. This is far higher than the cost for $y = 0$, so the learning algorithm will try very hard to prevent this from happening.

That's not bad in itself, but it means that something else might need to be sacrificed, if it's impossible to match all of the targets exactly. Perhaps the sacrifice will be that it incorrectly classifies some other training examples.

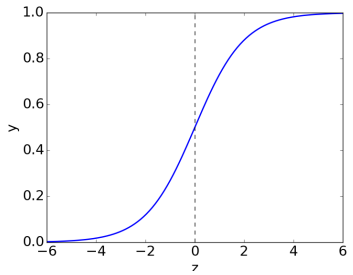
2.3 Attempt 3: logistic nonlinearity

The problem with linear regression is that the predictions were allowed to take arbitrary real values. But it makes no sense to predict anything smaller than 0 or larger than 1. Let's fix this problem by applying a **nonlinearity**, or **activation function**, which squashes the predictions to be between 0 and 1. In particular, we'll use something called the **logistic function**:

If you predict $y > 1$, then regardless of the target, you can decrease the loss by setting $y = 1$. Similarly for $y < 0$.

$$\sigma(z) = \frac{1}{1 + e^{-z}}. \tag{5}$$

This is a kind of **sigmoidal**, or S-shaped, function:



What's important about this function is that it increases monotonically, with asymptotes at 0 and 1. (Plus, it's smooth, so we can compute derivatives.)

Another advantage of the logistic function is that calculations tend to work out very nicely.

We refine the model as follows:

$$z = \mathbf{w}^\top \mathbf{x} + b \tag{6}$$

$$y = \sigma(z) \tag{7}$$

$$\mathcal{L}_{\text{SE}}(y, t) = \frac{1}{2}(y - t)^2. \tag{8}$$

Notice that this model solves the problem we observed with linear regression. As the predictions get more and more confident on the correct answer, the loss continues to decrease.

To derive the gradient descent updates, we'll need the partial derivatives of the cost function. We'll do this by applying the Chain Rule twice: first to compute $\mathrm{d}\mathcal{L}_{\text{SE}}/\mathrm{d}z$, and then again to compute $\partial\mathcal{L}_{\text{SE}}/\partial w_j$. But first, let's note the convenient fact that

This is equivalent to the elegant identity $\sigma'(z) = \sigma(z)(1 - \sigma(z))$.

$$\begin{aligned}\frac{\partial y}{\partial z} &= \frac{e^{-z}}{(1 + e^{-z})^2} \\ &= y(1 - y).\end{aligned} \tag{9}$$

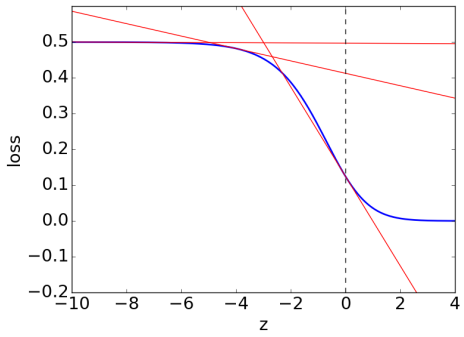


Figure 1: Visualization of derivatives of squared error loss with logistic nonlinearity, for a training example with $t = 1$. The derivative $\mathrm{d}\mathcal{J}/\mathrm{d}z$ corresponds to the slope of the tangent line.

Now for the Chain Rule:

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}z} &= \frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z} \\ &= (y-t)y(1-y) \qquad (10) \\ \frac{\partial\mathcal{L}_{\mathrm{SE}}}{\partial w_j} &= \frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}z}\frac{\partial z}{\partial w_j} \\ &= \frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}z}\cdot x_j. \qquad (11)\end{aligned}$$

Done! Why don't we go one step further and plug Eqn. 10 into Eqn. 11? The reason is that our goal isn't to compute a *formula* for $\partial\mathcal{L}_{\mathrm{SE}}/\partial w_j$; as computer scientists, our goal is to come up with a *procedure* for computing it. The two formulas above give us a procedure which we can implement directly in Python. One advantage of doing it this way is that we can reuse some of the work we've done in computing the derivative with respect to the bias:

At this point, you should stop and sanity check the equations we just derived, e.g. checking that they have the sign that they ought to. Get in the habit of doing this.

Reusing computation of derivatives is one of the main insights behind backpropagation, one of the central algorithms in this course.

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}b} &= \frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}z}\frac{\partial z}{\partial b} \\ &= \frac{\mathrm{d}\mathcal{L}_{\mathrm{SE}}}{\mathrm{d}z} \qquad (12)\end{aligned}$$

If we had expanded out the entire formula, it might not be obvious to us that we can reuse computation like this.

So far, so good. But there's one hitch. Let's suppose you classify one of the training examples extremely wrong, e.g. you confidently predict a negative label with $z = -5$, which gives $y \approx 0.0067$, for a positive example (i.e. $t = 1$). Plugging these values into Eqn 10, we find that $\partial\mathcal{L}_{\mathrm{SE}}/\partial z \approx -0.0066$. This is a pretty small value, considering how big the mistake was. As shown in Figure 1, the more confident the wrong prediction, the *smaller* the gradient is! The most badly misclassified examples will have hardly any effect on the training. This doesn't seem very good. We say the learning algorithm does not have a strong **gradient signal** for those training examples.

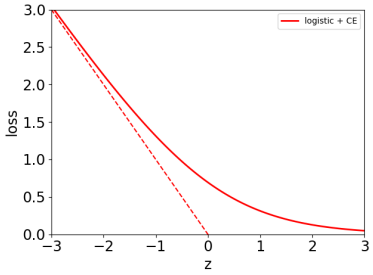


Figure 2: Plot of cross-entropy loss as a function of the input z to the logistic activation function.

The problem with squared error loss in the classification setting is that it doesn't distinguish bad predictions from extremely bad predictions. If $t = 1$, then a prediction of $y = 0.01$ has roughly the same squared-error loss as a prediction of $y = 0.00001$, even though in some sense the latter is more wrong. This isn't necessarily a problem in terms of the cost function itself: whether 0.00001 is inherently much worse than 0.01 depends on the situation. (If all we care about is classification error, they're essentially equivalent.) But *from the perspective of optimization*, the fact that the losses are nearly equivalent is a big problem. If we can increase y from 0.00001 to 0.0001, that means we're "getting warmer," but this doesn't show up in the squared-error loss. We'd like a loss function which reflects our intuitive notion of getting warmer.

Think about how the argument in this paragraph relates to the one in the previous paragraph.

Actually, the effect discussed here can also be beneficial, because it makes the algorithm *robust*, in that it can learn to ignore mislabeled examples. Cost functions like this are sometimes used for this reason. However, when you do use it, you should be aware of the optimization difficulties it creates!

2.4 Final touch: cross-entropy loss

The problem with squared-error loss is that it treats $y = 0.01$ and $y = 0.00001$ as nearly equivalent (for a positive example). We'd like a loss function which makes these very different. One such loss function is **cross-entropy (CE)**. This is defined as follows:

You'll sometimes see cross-entropy abbreviated XE.

$$\mathcal{L}_{\text{CE}}(y, t) = \begin{cases} -\log y & \text{if } t = 1 \\ -\log 1 - y & \text{if } t = 0 \end{cases} \tag{13}$$

In our earlier example, we see that $\mathcal{L}_{\text{CE}}(0.01, 1) = 4.6$, whereas $\mathcal{L}_{\text{CE}}(0.00001, 1) = 11.5$, so cross-entropy treats the latter as much worse than the former.

When we do calculations, it's cumbersome to use the case notation, so we instead rewrite Eqn. 13 in the following form. You should check that they are equivalent:

$$\mathcal{L}_{\text{CE}}(y, t) = -t \log y - (1 - t) \log 1 - y. \tag{14}$$

See if you can derive the equations for the asymptote lines.

Remember, the logistic function squashes y to be between 0 and 1, but cross-entropy draws big distinctions between probabilities close to 0 or 1. Interestingly, these effects cancel out: Figure 2 plots the loss as a function of z. You get a sizable gradient signal even when the predictions are very wrong.

When we combine the logistic activation function with cross-entropy loss, you get **logistic regression**:

$$\begin{aligned} z &= \mathbf{w}^\top \mathbf{x} + b \\ y &= \sigma(z) \\ \mathcal{L}_{\text{CE}} &= -t \log y - (1-t) \log 1 - y. \end{aligned} \tag{15}$$

Now let's compute the derivatives. We'll do it two different ways: the mechanical way, and the clever way. Let's do the mechanical way first, as an example of the chain rule for derivatives. Remember, our job here isn't to produce a formula for the derivatives, the way we would in calculus class. Our job is to give a procedure for computing the derivatives which we could translate into NumPy code. The following does that:

The second step of this derivation uses Eqn. 9.

$$\begin{aligned} \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}y} &= -\frac{t}{y} + \frac{1-t}{1-y} \\ \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}z} &= \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}y}\frac{\mathrm{d}y}{\mathrm{d}z} \\ &= \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}y} \cdot y(1-y) \\ \frac{\partial \mathcal{L}_{\text{CE}}}{\partial w_j} &= \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}z}\frac{\partial z}{\partial w_j} \\ &= \frac{\mathrm{d}\mathcal{L}_{\text{CE}}}{\mathrm{d}z} \cdot x_j \end{aligned} \tag{16}$$

This can be translated directly into NumPy (exercise: how do you vectorize this?). If we were good little computer scientists, we would stop here. But today we're going to be naughty computer scientists and break the abstraction barrier between the activation function (logistic) and the cost function (cross-entropy).

2.5 Logistic-cross-entropy function

There's a big problem with Eqns. 15 and 16: what happens if we have a positive example ($t = 1$), but we confidently classify it as a negative example ($z \ll 0$, implying $y \approx 0$)? This is likely to happen at the very beginning of training, so we should be able to handle it. But if y is small enough, it could be smaller than the smallest floating point value, i.e. **numerically zero**. Then when we compute the cross-entropy, we take the log of 0 and get $-\infty$. Or if this doesn't happen, think about Eqn. 16. Since y appears in the denominator, $\mathrm{d}\mathcal{L}_{\text{CE}}/\mathrm{d}y$ will be extremely large in magnitude, which again can cause numerical difficulties. These bugs are very subtle, and can be hard to track down if you don't expect them.

What we do instead is combine the logistic function and cross-entropy loss into a single function, which we term **logistic-cross-entropy**:

$$\mathcal{L}_{\text{LCE}}(z,t) = \mathcal{L}_{\text{CE}}(\sigma(z),t) = t \log(1 + e^{-z}) + (1-t)\log(1+e^{z}) \tag{17}$$

This still isn't numerically stable if we implement it directly, since e^z could blow up. But most scientific computing environments provide a numerically

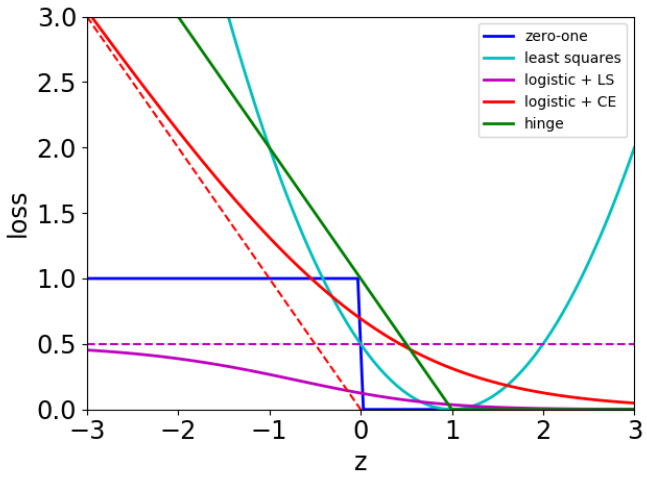


Figure 3: Comparison of the loss functions considered so far.

stable **log-sum-exp** routine.[1] In numpy, this is `np.logaddexp`. So the following code would compute the logistic-cross-entropy:

```
E = t * np.logaddexp(0, -z) + (1-t) * np.logaddexp(0, z)
```

Now to compute the loss derivatives:

$$\begin{aligned}\frac{\mathrm{d}\mathcal{L}_{\mathrm{LCE}}}{\mathrm{d}z} &= \frac{\mathrm{d}}{\mathrm{d}z}\left[t\log(1+e^{-z}) + (1-t)\log(1+e^{z})\right] \\ &= -t\cdot\frac{e^{-z}}{1+e^{-z}} + (1-t)\frac{e^{z}}{1+e^{z}} \\ &= -t(1-y) + (1-t)y \\ &= y - t\end{aligned} \tag{18}$$

This is like magic! We took a somewhat complicated formula for the logistic activation function, combined it with a somewhat complicated formula for the cross-entropy loss, and wound up with a stunningly simple formula for the loss derivative! Observe that this is exactly the same formula as for $\mathrm{d}\mathcal{L}_{\mathrm{SE}}/\mathrm{d}y$ in the case of linear regression. And it has the same intuitive interpretation: if $y > t$, you made too positive a prediction, so you want to shift your prediction in the negative direction. Conversely, if $y < t$, you want to shift your prediction in the positive direction.

This isn't a coincidence. The reason it happens is beyond the scope of this course, but if you're curious, look up "generalized linear models."

2.6 Another alternative: hinge loss

Another loss function you might encounter is **hinge loss**. Here, y is a real value, and $t \in \{-1, 1\}$.

$$\mathcal{L}_{\mathrm{H}}(y, t) = \max(0, 1 - ty)$$

Hinge loss is plotted in Figure 3 for a positive example. One useful property of hinge loss is that it's an upper bound on 0–1 loss; this is a useful property

[1] The log-sum-exp trick is pretty neat. `https://hips.seas.harvard.edu/blog/2013/01/09/computing-log-sum-exp/`

for a surrogate loss function, since it means that if you make the hinge loss small, you've also made 0–1 loss small. A linear model with hinge loss is known as a **support vector machine (SVM)**:

$$y = \mathbf{w}^\top \mathbf{x} + b \tag{19}$$

$$\mathcal{L}_{\text{H}} = \max(0, 1 - ty) \tag{20}$$

If you take CSC411, you'll learn a lot about SVMs, including their statistical motivation, how to optimize them efficiently and how to make them nonlinear (using something called the "kernel trick"). But you already know one optimization method: you already know enough to derive the gradient descent updates for an SVM.

Interestingly, even though SVMs came from a different community and had a different sort of motivation from logistic regression, the algorithms behave very similarly in practice. The reason has to do with the loss functions. Figure 3 compares hinge loss to cross-entropy loss; even though cross-entropy is smoother, the asymptotic behavior is the same, suggesting the loss functions are basically pretty similar.

All of the loss functions covered so far is shown in Figure 3. Take the time to review them, to understand their strengths and weaknesses.

3 Multiclass classification

So far we've talked about binary classification, but most classification problems involve more than two categories. Fortunately, this doesn't require any new ideas: everything pretty much works by analogy with the binary case. The first question is how to represent the targets. We could represent them as integers, but it's more convenient to use a **one-hot vector**, also called a **one-of-K encoding**:

$$\mathbf{t} = \underbrace{(0, \ldots, 0, 1, 0, \ldots, 0)}_{\text{entry } k \text{ is } 1} \tag{21}$$

Now let's design the whole model by analogy with the binary case.

First of all, consider the linear part of the model. We have K outputs and D inputs. To represent a linear function, we'll need a $K \times D$ **weight matrix**, as well as a K-dimensional **bias vector**. We first compute the intermediate quantities as follows:

$$\mathbf{z} = \mathbf{W}\mathbf{x} + \mathbf{b}. \tag{22}$$

This is the general form of a linear function from $\mathbb{R}^D$ to $\mathbb{R}^K$.

Next, the activation function. We saw that the logistic function was a good thing to use in the binary case. There's a multivariate generalization of the logistic function called the **softmax function**:

Try plugging in $K = 2$ to figure out how the softmax relates to the logistic function.

$$y_k = \text{softmax}(z_1, \ldots, z_K)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}} \tag{23}$$

Importantly, the outputs of the softmax function are nonnegative and sum to 1, so they can be interpreted as a probability distribution over the K

classes (just like the output of the logistic could be interpreted as a probability). The inputs to the softmax are called the **logits**. Note that when one of the z_k's is much larger than the others, the output of the softmax will be approximately the argmax, in the one-hot encoding. Hence, a more accurate name might be "soft-argmax."

Think about the logits as the "log-odds", because when you exponentiate them you get the odds ratios of the probabilities.

Finally, the loss function. Cross-entropy can be generalized to the multiple-output case:

$$\begin{aligned}\mathcal{L}_{\text{CE}}(\mathbf{y}, \mathbf{t}) &= -\sum_{k=1}^{K} t_k \log y_k \\ &= -\mathbf{t}^\top (\log \mathbf{y}).\end{aligned}$$

You'll sometimes see $\sigma(\mathbf{z})$ used to denote the softmax function, by analogy with the logistic. But in this course, it will always denote the logistic function.

Here, $\log \mathbf{y}$ represents the elementwise log. Note that only one of the t_k's is 1 and the rest are 0, so the summation has the effect of picking the relevant entry of the vector $\log \mathbf{y}$. (See how convenient the one-hot notation is?) Note that this loss function only makes sense for predictions which sum to 1; if you eliminate that constraint, you could trivially minimize the loss by making all the y_k's large.

Try plugging in $K = 2$ to see how this relates to binary cross-entropy.

When we put these things together, we get **multiclass logistic regression**, or **softmax regression**:

$$\begin{aligned}\mathbf{z} &= \mathbf{W}\mathbf{x} + \mathbf{b} \\ \mathbf{y} &= \text{softmax}(\mathbf{z}) \\ \mathcal{L}_{\text{CE}} &= -\mathbf{t}^\top (\log \mathbf{y})\end{aligned}$$

We won't go through the derivatives in detail, but it basically works out exactly like logistic regression. The softmax and cross-entropy functions interact nicely with each other, so we always combine them into a single **softmax-cross-entropy** function $\mathcal{L}_{\text{SCE}}$ for purposes of numerical stability. The derivatives of $\mathcal{L}_{\text{SCE}}$ have the same elegant formula we've been seeing repeatedly, except this time remember that $\mathbf{t}$ and $\mathbf{y}$ are both vectors:

$$\frac{\partial \mathcal{L}_{\text{SCE}}}{\partial \mathbf{z}} = \mathbf{y} - \mathbf{t} \tag{24}$$

Softmax regression is an elegant learning algorithm which can work very well in practice.

4 Convex Functions

An important criterion we often use to compare different loss functions is convexity. Recall that a set $\mathcal{S}$ is convex if the line segment connecting any two points in $\mathcal{S}$ lies entirely within $\mathcal{S}$. Mathematically, this means that for $\mathbf{x}_0, \mathbf{x}_1 \in \mathcal{S}$,

$$(1-\lambda)\mathbf{x}_0 + \lambda \mathbf{x}_1 \in \mathcal{S} \quad \text{for } 0 \le \lambda \le 1.$$

The definition of a **convex function** is closely related. A function f is convex if for any $\mathbf{x}_0, \mathbf{x}_1$ in the domain of f,

$$f((1-\lambda)\mathbf{x}_0 + \lambda \mathbf{x}_1) \le (1-\lambda) f(\mathbf{x}_0) + \lambda f(\mathbf{x}_1) \tag{25}$$

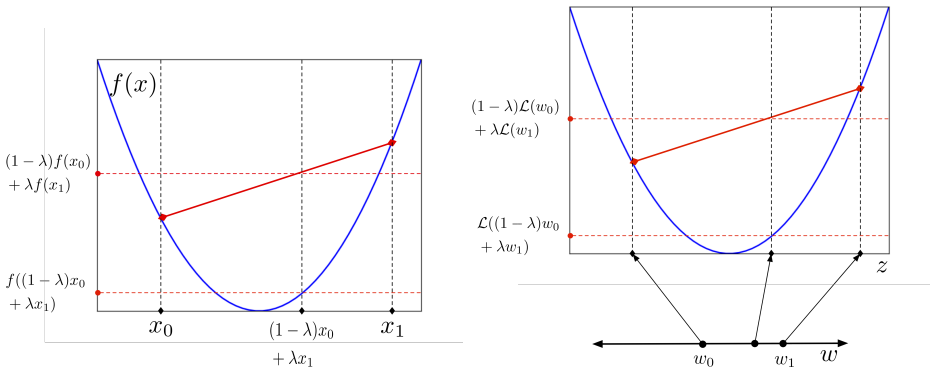


Figure 4: **Left:** Definition of convexity. **Right:** Proof-by-picture that if the model is linear and $\mathcal{L}$ is a convex function of $z = \mathbf{w}^\top \mathbf{x} + b$, then it's also convex as a function of $\mathbf{w}$ and b.

This is shown graphically in Figure 4. Another way to phrase this requirement is that the line segment connecting any two points on the graph of f must lie above the graph of f. Equivalently, the set of points lying above the graph of f must be a convex set. Intuitively, convex functions are bowl-shaped.

Convexity is a really important property from the standpoint of optimization. There are two main reasons for this:

1. All critical points are global minima, so if you can set the derivatives to zero, you've solved the problem.

2. Gradient descent will always converge to the global minimum.

We'll talk in more detail in a later lecture about what can go wrong when the cost function is not convex. Look back at our comparison of loss functions in Figure 3. You can see visually that squared error, hinge loss, and the logistic regression objective are all convex; 0–1 loss, and logistic-with-least-squares are not convex. It's not a coincidence that the loss functions we might actually try to optimize are the convex ones. There is an entire field of research on convex optimization, which comes up with better ways to minimize convex functions over convex sets, as well as ways to formulate various kinds of problems in terms of convex optimization.

Note that even though convexity is important, most of the optimization problems we'll consider in this course will be non-convex, because training a deep neural network is a non-convex problem, even when the loss function is convex. Nonetheless, convex loss functions somehow still tend to be advantageous from the standpoint of optimization.

5 Gradient Checking with Finite Differences

We've derived a lot of gradients so far. How do we know if they're correct? Fortunately, there's an easy and effective procedure for testing them. Recall

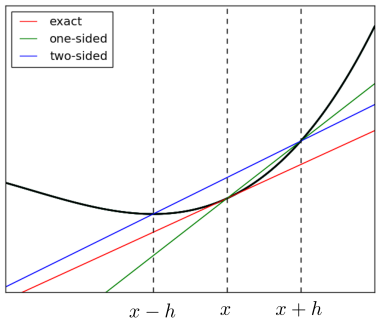


Figure 5: Estimating a derivative using one-sided and two-sided finite differences.

the definition of the partial derivative:

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i, \ldots, x_N)}{h} \tag{26}$$

We can check the derivatives numerically by plugging in a small value of h, such as 10^{-10}. This is known as the method of **finite differences**.

You don't want to implement your actual learning algorithm using finite differences, because it's very slow, but it's great for testing.

It's actually better to use the two-sided definition of the partial derivative than the one-sided one, since it is much more accurate:

$$\frac{\partial}{\partial x_i} f(x_1, \ldots, x_N) = \lim_{h \to 0} \frac{f(x_1, \ldots, x_i + h, \ldots, x_N) - f(x_1, \ldots, x_i - h, \ldots, x_N)}{2h} \tag{27}$$

An example is shown in Figure 5 of estimating a derivative using the one-sided and two-sided formulas.

Gradient checking is really important! In machine learning, your algorithm can often seem to learn well even if the gradient calculation is totally wrong. This might lead you to skip the correctness checks. But it might work even better if the derivatives are correct, and this is important when you're trying to squeeze out the last bit of accuracy. Wrong derivatives can also lead you on wild goose chases, as you make changes to your system which appear to help significantly, but actually are only helping because they compensate for errors in the gradient calculations. If you implement derivatives by hand, gradient checking is the single most important thing you need to do to get your algorithm to work well.