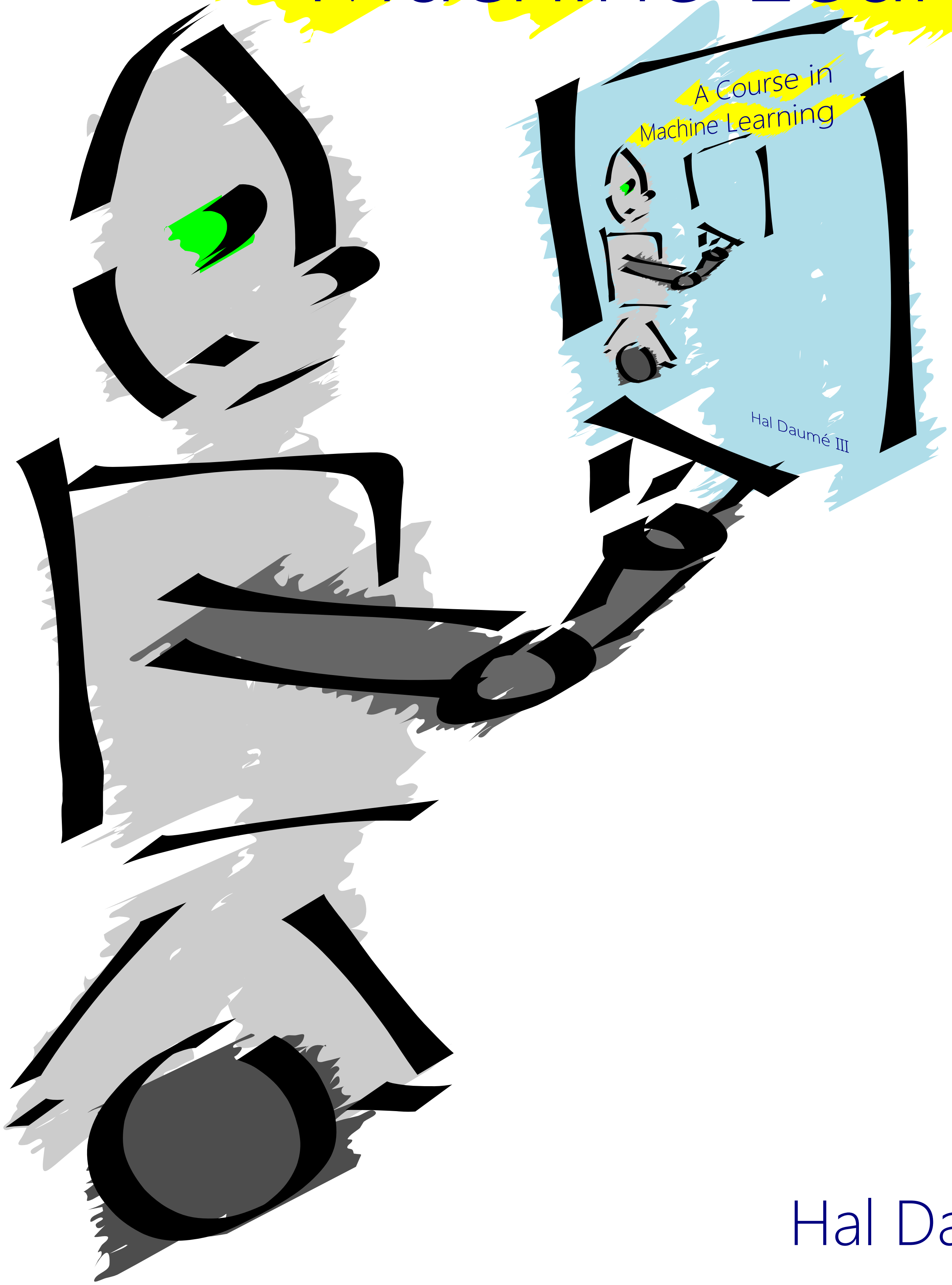
A Course in Machine Learning



Hal Daumé III

6 | Linear Models

The essence of mathematics is not to make simple things complicated, but to make complicated things simple. – Stanley Gudder

Learning Objectives:

- Define and plot four surrogate loss functions: squared loss, logistic loss, exponential loss and hinge loss.
- Compare and contrast the optimization of 0/1 loss and surrogate loss functions.
- Solve the optimization problem for squared loss with a quadratic regularizer in closed form.
- Implement and debug gradient descent and subgradient descent.

Dependencies:

In Chapter **??**, you learned about the perceptron algorithm for linear classification. This was both a *model* (linear classifier) and *algorithm* (the perceptron update rule) in one. In this section, we will separate these two, and consider general ways for optimizing linear models. This will lead us into some aspects of optimization (aka mathematical programming), but not very far. At the end of this chapter, there are pointers to more literature on optimization for those who are interested.

The basic idea of the perceptron is to run a particular algorithm until a linear separator is found. You might ask: are there better algorithms for finding such a linear separator? We will follow this idea and formulate a learning problem as an explicit optimization problem: find me a linear separator that is not too complicated. We will see that finding an "optimal" separator is actually computationally prohibitive, and so will need to "relax" the optimality requirement. This will lead us to a **convex** objective that combines a loss function (how well are we doing on the training data?) and a regularizer (how complicated is our learned model?). This learning framework is known as both **Tikhonov regularization** and **structural risk minimization**.

6.1 *The Optimization Framework for Linear Models*

You have already seen the perceptron as a way of finding a weight vector $\boldsymbol{w}$ and bias b that do a good job of separating positive training examples from negative training examples. The perceptron is a **model** and **algorithm** in one. Here, we are interested in *separating* these issues. We will focus on linear models, like the perceptron. But we will think about other, more generic ways of finding good parameters of these models.

The goal of the perceptron was to find a **separating hyperplane** for some training data set. For simplicity, you can ignore the issue of overfitting (but just for now!). Not all data sets are linearly separable. In the case that your training data *isn't* linearly separable, you might want to find the hyperplane that makes the *fewest errors* on the training data. We can write this down as a formal mathematics **optimization problem** as follows:

$$\min_{w,b} \quad \sum_n \mathbf{1}[y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 0] \tag{6.1}$$

In this expression, you are optimizing over two variables, $\boldsymbol{w}$ and b. The **objective function** is the thing you are trying to minimize. In this case, the objective function is simply the **error rate** (or **0/1 loss**) of the linear classifier parameterized by $\boldsymbol{w}, b$. In this expression, $\mathbf{1}[\cdot]$ is the **indicator function**: it is one when $(\cdot)$ is true and zero otherwise.

? You should remember the $y\boldsymbol{w} \cdot \boldsymbol{x}$ trick from the perceptron discussion. If not, re-convince yourself that this is doing the right thing.

We know that the perceptron algorithm is guaranteed to find parameters for this model if the data is linearly separable. In other words, if the optimum of Eq (6.1) is zero, then the perceptron will efficiently find parameters for this model. The notion of "efficiency" depends on the margin of the data for the perceptron.

You might ask: what happens if the data is *not* linearly separable? Is there an efficient algorithm for finding an optimal setting of the parameters? Unfortunately, the answer is *no.* There is no polynomial time algorithm for solving Eq (6.1), unless P=NP. In other words, this problem is NP-hard. Sadly, the proof of this is quite complicated and beyond the scope of this book, but it relies on a reduction from a variant of satisfiability. The key idea is to turn a satisfiability problem into an optimization problem where a clause is satisfied exactly when the hyperplane correctly separates the data.

You might then come back and say: okay, well I don't really need an *exact* solution. I'm willing to have a solution that makes one or two more errors than it has to. Unfortunately, the situation is really bad. Zero/one loss is NP-hard to even *appproximately minimize*. In other words, there is no efficient algorithm for even finding a solution that's a small constant worse than optimal. (The best known constant at this time is $418/415 \approx 1.007$.)

However, before getting too disillusioned about this whole enterprise (remember: there's an entire chapter about this framework, so it must be going somewhere!), you should remember that optimizing Eq (6.1) perhaps isn't even what you want to do! In particular, all it says is that you will get minimal *training error.* It says nothing about what your *test error* will be like. In order to try to find a solution that will *generalize* well to test data, you need to ensure that you do not overfit the data. To do this, you can introduce a **regularizer** over the parameters of the model. For now, we will be vague about what this regularizer looks like, and simply call it an arbitrary function $R(\boldsymbol{w}, b)$.

This leads to the following, **regularized objective**:

$$\min_{\boldsymbol{w},b} \quad \sum_n \mathbf{1}[y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b) > 0] + \lambda R(\boldsymbol{w}, b) \tag{6.2}$$

In Eq (6.2), we are now trying to optimize a *trade-off* between a solution that gives low training error (the first term) and a solution that is "simple" (the second term). You can think of the maximum depth hyperparameter of a decision tree as a form of regularization for trees. Here, R is a form of regularization for hyperplanes. In this formulation, λ becomes a **hyperparameter** for the optimization.

The key remaining questions, given this formalism, are:

- How can we adjust the optimization problem so that there *are* efficient algorithms for solving it?
- What are good regularizers $R(\boldsymbol{w}, b)$ for hyperplanes?
- Assuming we can adjust the optimization problem appropriately, what algorithms exist for efficiently solving this regularized optimization problem?

We will address these three questions in the next sections.

? Assuming R does the "right thing," what value(s) of λ will lead to overfitting? What value(s) will lead to underfitting?

6.2 Convex Surrogate Loss Functions

You might ask: why is optimizing zero/one loss so hard? Intuitively, one reason is that small changes to $\boldsymbol{w}, b$ can have a large impact on the value of the objective function. For instance, if there is a positive training example with $\boldsymbol{w}, \boldsymbol{x} \cdot +b = -0.0000001$, then adjusting b upwards by 0.00000011 will decrease your error rate by 1. But adjusting it upwards by 0.00000009 will have no effect. This makes it really difficult to figure out good ways to adjust the parameters.

To see this more clearly, it is useful to look at plots that relate *margin* to *loss*. Such a plot for zero/one loss is shown in Figure 6.1. In this plot, the horizontal axis measure the margin of a data point and the vertical axis measures the loss associated with that margin. For zero/one loss, the story is simple. If you get a positive margin (i.e., $y(\boldsymbol{w}\cdot\boldsymbol{x} + b) > 0$) then you get a loss of zero. Otherwise you get a loss of one. By thinking about this plot, you can see how changes to the parameters that change the margin *just a little bit* can have an enormous effect on the overall loss.

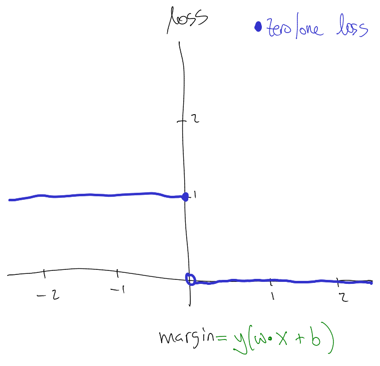


Figure 6.1: plot of zero/one versus margin

You might decide that a reasonable way to address this problem is to replace the non-smooth zero/one loss with a smooth approximation. With a bit of effort, you could probably concoct an "S"-shaped function like that shown in Figure 6.2. The benefit of using such an S-function is that it is smooth, and potentially easier to optimize. The difficulty is that it is not **convex**.

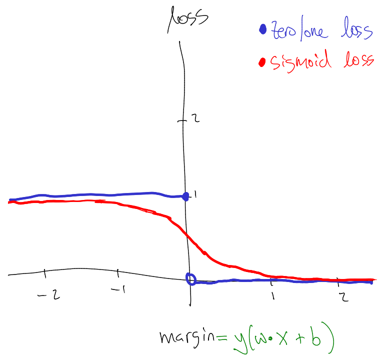
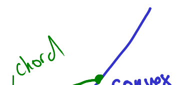


Figure 6.2: plot of zero/one versus margin and an S version of it



If you remember from calculus, a convex function is one that looks like a happy face ($\smile$). (On the other hand, a **concave** function is one that looks like a sad face ($\frown$); an easy mnemonic is that you can hide under a con**cave** function.) There are two equivalent definitions of a concave function. The first is that it's second derivative is always non-negative. The second, more geometric, defition is that any **chord** of the function lies above it. This is shown in Figure **??**. There you can see a convex function and a non-convex function, both with two chords drawn in. In the case of the convex function, the chords lie above the function. In the case of the non-convex function, there are parts of the chord that lie below the function.

Convex functions are nice because they are *easy to minimize*. Intuitively, if you drop a ball anywhere in a convex function, it will eventually get to the minimum. This is not true for non-convex functions. For example, if you drop a ball on the very left end of the S-function from Figure 6.2, it will not go anywhere.

This leads to the idea of **convex surrogate loss functions**. Since zero/one loss is hard to optimize, you want to optimize something else, instead. Since convex functions are easy to optimize, we want to approximate zero/one loss with a convex function. This approximating function will be called a **surrogate loss**. The surrogate losses we construct will always be *upper bounds* on the true loss function: this guarantees that if you minimize the surrogate loss, you are also pushing down the real loss.

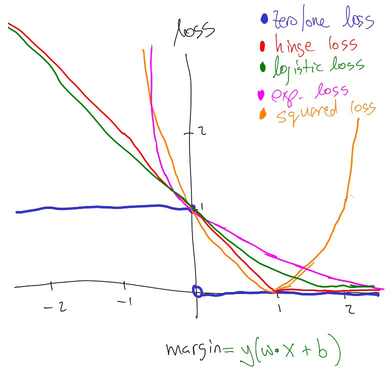


Figure 6.4: surrogate loss fns

There are four common surrogate loss function, each with their own properties: **hinge loss**, **logistic loss**, **exponential loss** and **squared loss**. These are shown in Figure 6.4 and defined below. These are defined in terms of the true label y (which is just $\{-1,+1\}$) and the predicted value $\hat{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b$.

$$\text{Zero/one:} \qquad \ell^{(0/1)}(y,\hat{y}) = \mathbf{1}[y\hat{y} \leq 0] \tag{6.3}$$

$$\text{Hinge:} \qquad \ell^{(\text{hin})}(y,\hat{y}) = \max\{0, 1 - y\hat{y}\} \tag{6.4}$$

$$\text{Logistic:} \qquad \ell^{(\text{log})}(y,\hat{y}) = \frac{1}{\log 2}\log\left(1 + \exp[-y\hat{y}]\right) \tag{6.5}$$

$$\text{Exponential:} \qquad \ell^{(\text{exp})}(y,\hat{y}) = \exp[-y\hat{y}] \tag{6.6}$$

$$\text{Squared:} \qquad \ell^{(\text{sqr})}(y,\hat{y}) = (y - \hat{y})^2 \tag{6.7}$$

In the definition of logistic loss, the $\frac{1}{\log 2}$ term out front is there simply to ensure that $\ell^{(\text{log})}(y,0) = 1$. This ensures, like all the other surrogate loss functions, that logistic loss upper bounds the zero/one loss. (In practice, people typically omit this constant since it does not affect the optimization.)

There are two big differences in these loss functions. The first difference is how "upset" they get by erroneous predictions. In the case of hinge loss and logistic loss, the growth of the function as $\hat{y}$ goes negative is linear. For squared loss and exponential loss, it is super-linear. This means that exponential loss would rather get a few examples a little wrong than one example really wrong. The other difference is how they deal with very confident correct predictions. Once $y\hat{y} > 1$, hinge loss does not care any more, but logistic and exponential still think you can do better. On the other hand, squared loss thinks it's just as bad to predict $+3$ on a positive example as it is to predict -1 on a positive example.

6.3 *Weight Regularization*

In our learning objective, Eq (**??**), we had a term correspond to the zero/one loss on the training data, plus a **regularizer** whose goal was to ensure that the learned function didn't get too "crazy." (Or, more formally, to ensure that the function did not overfit.) If you replace to zero/one loss with a surrogate loss, you obtain the following objective:

$$\min_{\boldsymbol{w},b} \quad \sum_n \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n + b) + \lambda R(\boldsymbol{w}, b) \tag{6.8}$$

The question is: what should $R(\boldsymbol{w}, b)$ look like?

From the discussion of surrogate loss function, we would like to ensure that R is convex. Otherwise, we will be back to the point where optimization becomes difficult. Beyond that, a common desire is that the components of the weight vector (i.e., the w_ds) should be small (close to zero). This is a form of **inductive bias**.

Why are small values of w_d good? Or, more precisely, why do small values of w_d correspond to *simple functions*? Suppose that we have an example $\boldsymbol{x}$ with label $+1$. We might believe that other examples, $\boldsymbol{x}'$ that are nearby $\boldsymbol{x}$ should also have label $+1$. For example, if I obtain $\boldsymbol{x}'$ by taking $\boldsymbol{x}$ and changing the first component by some small value ϵ and leaving the rest the same, you might think that the classification would be the same. If you do this, the difference between $\hat{y}$ and $\hat{y}'$ will be exactly ϵw_1. So if w_1 is reasonably small, this is unlikely to have much of an effect on the classification decision. On the other hand, if w_1 is large, this could have a large effect.

Another way of saying the same thing is to look at the derivative of the predictions as a function of w_1. The derivative of $\boldsymbol{w}, \boldsymbol{x} \cdot +b$ with respect to w_1 is:

$$\frac{\partial \boldsymbol{w}, \boldsymbol{x} \cdot +b}{\partial w_1} = \frac{\partial \left[\sum_d w_d x_d + b\right]}{\partial w_1} = x_1 \tag{6.9}$$

Interpreting the derivative as the rate of change, we can see that the rate of change of the prediction function is proportional to the individual weights. So if you want the function to change slowly, you want to ensure that the weights stay small.

One way to accomplish this is to simply use the norm of the weight vector. Namely $R^{(\mathsf{norm})}(\boldsymbol{w}, b) = ||\boldsymbol{w}|| = \sqrt{\sum_d w_d^2}$. This function is convex and smooth, which makes it easy to minimize. In practice, it's often easier to use the squared norm, namely $R^{(\mathsf{sqr})}(\boldsymbol{w}, b) = ||\boldsymbol{w}||^2 = \sum_d w_d^2$ because it removes the ugly square root term and remains convex. An alternative to using the sum of squared weights is to use the sum of absolute weights: $R^{(\mathsf{abs})}(\boldsymbol{w}, b) = \sum_d |w_d|$. Both of these norms are convex.

? Why do we not regularize the bias term b?

In addition to small weights being good, you could argue that *zero* weights are better. If a weight w_d goes to zero, then this means that feature d is not used at all in the classification decision. If there are a large number of irrelevant features, you might want as many weights to go to zero as possible. This suggests an alternative regularizer: $R^{(\mathsf{cnt})}(\boldsymbol{w}, b) = \sum_d \mathbf{1}[x_d \neq 0]$.

This line of thinking leads to the general concept of ***p*-norms**. (Technically these are called ℓ_p (or "ell p") norms, but this notation clashes with the use of ℓ for "loss.") This is a family of norms that all have the same general flavor. We write $||\boldsymbol{w}||_p$ to denote the p-norm of $\boldsymbol{w}$.

? Why might you not want to use $R^{(\mathsf{cnt})}$ as a regularizer?

$$||\boldsymbol{w}||_p = \left(\sum_d |w_d|^p \right)^{\frac{1}{p}} \tag{6.10}$$

You can check that the 2-norm exactly corresponds to the usual Euclidean norm, and that the 1-norm corresponds to the "absolute" regularizer described above.

? You can actually identify the $R^{(\mathsf{cnt})}$ regularizer with a p-norm as well. Which value of p gives it to you? (Hint: you may have to take a limit.)

When p-norms are used to regularize weight vectors, the interesting aspect is how they trade-off multiple features. To see the behavior of p-norms in two dimensions, we can plot their **contour** (or **level-set**). Figure 6.5 shows the contours for the same p norms in two dimensions. Each line denotes the two-dimensional vectors to which this norm assignes a total value of 1. By changing the value of p, you can interpolate between a square (the so-called "max norm"), down to a circle (2-norm), diamond (1-norm) and pointy-star-shaped-thing ($p < 1$ norm).

In general, smaller values of p "prefer" sparser vectors. You can see this by noticing that the contours of small p-norms "stretch" out along the axes. It is for this reason that small p-norms tend to yield weight vectors with many zero entries (aka **sparse** weight vectors). Unfortunately, for $p < 1$ the norm becomes non-convex. As you might guess, this means that the 1-norm is a popular choice for sparsity-seeking applications.

Figure 6.5: `loss:norms2d`: level sets of the same p-norms

? The max norm corresponds to $\lim_{p \to \infty}$. Why is this called the max norm?

Math Review | Gradients

... be sure to do enough to do the closed for squared error

Figure 6.6:

Algorithm 22 GradientDescent($\mathcal{F}$, K, η_1, ...)

```
1: $z^{(0)} \leftarrow \langle 0, 0, \dots, 0 \rangle$            // initialize variable we are optimizing
2: for $k = 1 \dots K$ do
3:    $g^{(k)} \leftarrow \nabla_z \mathcal{F}|_{z^{(k-1)}}$            // compute gradient at current location
4:    $z^{(k)} \leftarrow z^{(k-1)} - \eta^{(k)} g^{(k)}$            // take a step down the gradient
5: end for
6: return $z^{(K)}$
```

6.4 Optimization with Gradient Descent

Envision the following problem. You're taking up a new hobby: blindfolded mountain climbing. Someone blindfolds you and drops you on the side of a mountain. Your goal is to get to the peak of the mountain as quickly as possible. All you can do is feel the mountain where you are standing, and take steps. How would you get to the top of the mountain? Perhaps you would feel to find out what direction feels the most "upward" and take a step in that direction. If you do this repeatedly, you might hope to get the the top of the mountain. (Actually, if your friend promises always to drop you on purely concave mountains, you *will* eventually get to the peak!)

The idea of gradient-based methods of optimization is exactly the same. Suppose you are trying to find the maximum of a function $f(\boldsymbol{x})$. The optimizer maintains a current estimate of the parameter of interest, $\boldsymbol{x}$. At each step, it measures the **gradient** of the function it is trying optimize. This measurement occurs *at* the current location, $\boldsymbol{x}$. Call the gradient $\boldsymbol{g}$. It then takes a step in the direction of the gradient, where the size of the step is controlled by a parameter η (eta). The complete step is $\boldsymbol{x} \leftarrow \boldsymbol{x} + \eta \boldsymbol{g}$. This is the basic idea of **gradient ascent**.

The opposite of gradient ascent is **gradient descent**. All of our learning problems will be framed as *minimization* problems (trying to reach the bottom of a ditch, rather than the top of a hill). Therefore, descent is the primary approach you will use. One of the major conditions for gradient ascent being able to find the true, **global minimum**, of its objective function is convexity. Without convexity, all is lost.

The gradient descent algorithm is sketched in Algorithm 6.4. The function takes as arguments the function $\mathcal{F}$ to be minimized, the number of iterations K to run and a sequence of learning rates

$\eta_1, \dots, \eta_K$. (This is to address the case that you might want to start your mountain climbing taking large steps, but only take small steps when you are close to the peak.)

The only real work you need to do to apply a gradient descent method is be able to compute derivatives. For concreteness, suppose that you choose exponential loss as a loss function and the 2-norm as a regularizer. Then, the regularized objective function is:

$$\mathcal{L}(\boldsymbol{w}, b) = \sum_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \frac{\lambda}{2}||\boldsymbol{w}||^2 \tag{6.11}$$

The only "strange" thing in this objective is that we have replaced λ with $\frac{\lambda}{2}$. The reason for this change is just to make the gradients cleaner. We can first compute derivatives with respect to b:

$$\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial}{\partial b}\sum_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \frac{\partial}{\partial b}\frac{\lambda}{2}||\boldsymbol{w}||^2 \tag{6.12}$$

$$= \sum_n \frac{\partial}{\partial b}\exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + 0 \tag{6.13}$$

$$= \sum_n \left(\frac{\partial}{\partial b} - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right)\exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] \tag{6.14}$$

$$= -\sum_n y_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] \tag{6.15}$$

Before proceeding, it is worth thinking about what this says. From a practical perspective, the optimization will operate by updating $b \leftarrow b - \eta\frac{\partial \mathcal{L}}{\partial b}$. Consider positive examples: examples with $y_n = +1$. We would hope for these examples that the current prediction, $\boldsymbol{w} \cdot \boldsymbol{x}_n + b$, is as large as possible. As this value tends toward ∞, the term in the exp[] goes to zero. Thus, such points will not contribute to the step. However, if the current prediction is small, then the exp[] term will be positive and non-zero. This means that the bias term b will be *increased*, which is exactly what you would want. Moreover, once all points are very well classified, the derivative goes to zero.

? This considered the case of positive examples. What happens with negative examples?

Now that we have done the easy case, let's do the gradient with respect to $\boldsymbol{w}$.

$$\nabla_{\boldsymbol{w}}\mathcal{L} = \nabla_{\boldsymbol{w}}\sum_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \nabla_{\boldsymbol{w}}\frac{\lambda}{2}||\boldsymbol{w}||^2 \tag{6.16}$$

$$= \sum_n (\nabla_{\boldsymbol{w}} - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b))\exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \lambda\boldsymbol{w} \tag{6.17}$$

$$= -\sum_n y_n\boldsymbol{x}_n \exp\left[-y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\right] + \lambda\boldsymbol{w} \tag{6.18}$$

Now you can repeat the previous exercise. The update is of the form $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta\nabla_{\boldsymbol{w}}\mathcal{L}$. For well classified points (ones that are tend toward $y_n\infty$), the gradient is near zero. For poorly classified points, the gradient points in the direction $-y_n \boldsymbol{x}_n$, so the update is of the form $\boldsymbol{w} \leftarrow \boldsymbol{w} + c y_n \boldsymbol{x}_n$, where c is some constant. This is just like the perceptron update! Note that c is large for very poorly classified points and small for relatively well classified points.

By looking at the part of the gradient related to the regularizer, the update says: $\boldsymbol{w} \leftarrow \boldsymbol{w} - \lambda \boldsymbol{w} = (1-\lambda)\boldsymbol{w}$. This has the effect of *shrinking* the weights toward zero. This is exactly what we expect the regulaizer to be doing!

The success of gradient descent hinges on appropriate choices for the step size. Figure 6.7 shows what can happen with gradient descent with poorly chosen step sizes. If the step size is too big, you can accidentally step over the optimum and end up oscillating. If the step size is too small, it will take way too long to get to the optimum. For a well-chosen step size, you can show that gradient descent will approach the optimal value at a fast *rate*. The notion of convergence here is that the *objective value* converges to the true minimum.

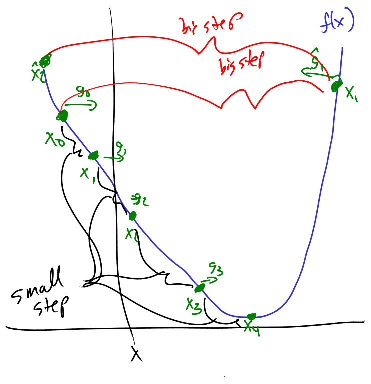


Figure 6.7: good and bad step sizes

Theorem 7 (Gradient Descent Convergence)**.** *Under suitable conditions*[1]*, for an appropriately chosen constant step size (i.e.,* $\eta_1 = \eta_2, \dots = \eta$*), the* **convergence rate** *of gradient descent is* $\mathcal{O}(1/k)$*. More specifically, letting* $\boldsymbol{z}^*$ *be the global minimum of* $\mathcal{F}$*, we have:* $\mathcal{F}(\boldsymbol{z}^{(k)}) - \mathcal{F}(\boldsymbol{z}^*) \leq \frac{2||\boldsymbol{z}^{(0)} - \boldsymbol{z}^*||^2}{\eta k)}$.

[1] Specifically the function to be optimized needs to be **strongly convex**. This is true for all our problems, provided $\lambda > 0$. For $\lambda = 0$ the rate could be as bad as $\mathcal{O}(1/\sqrt{k})$.

? A naive reading of this theorem seems to say that you should choose huge values of η. It should be obvious that this cannot be right. What is missing?

The proof of this theorem is a bit complicated because it makes heavy use of some linear algebra. The key is to set the learning rate to $1/L$, where L is the maximum **curvature** of the function that is being optimized. The curvature is simply the "size" of the second derivative. Functions with high curvature have gradients that change quickly, which means that you need to take small steps to avoid overstepping the optimum.

This convergence result suggests a simple approach to deciding when to stop optimizing: wait until the objective function stops changing by much. An alternative is to wait until the *parameters* stop changing by much. A final example is to do what you did for perceptron: early stopping. Every iteration, you can check the performance of the current model on some held-out data, and stop optimizing when performance plateaus.

6.5 *From Gradients to Subgradients*

As a good exercise, you should try deriving gradient descent update rules for the different loss functions and different regularizers you've learned about. However, if you do this, you might notice that *hinge loss* and the 1-norm regularizer are not differentiable everywhere! In particular, the 1-norm is not differentiable around $w_d = 0$, and the hinge loss is not differentiable around $y\hat{y} = 1$.

The solution to this is to use **subgradient** optimization. One way to think about subgradients is just to not think about it: you essentially need to just ignore the fact that you forgot that your function wasn't differentiable, and just try to apply gradient descent anyway.

To be more concrete, consider the hinge function $f(z) = \max\{0, 1 - z\}$. This function is differentiable for $z > 1$ and differentiable for $z < 1$, but not differentiable at $z = 1$. You can derive this using differentiation by parts:

$$\frac{\partial}{\partial z} f(z) = \frac{\partial}{\partial z} \begin{cases} 0 & \text{if } z > 1 \\ 1 - z & \text{if } z < 1 \end{cases} \tag{6.19}$$

$$= \begin{cases} \frac{\partial}{\partial z} 0 & \text{if } z > 1 \\ \frac{\partial}{\partial z}(1 - z) & \text{if } z < 1 \end{cases} \tag{6.20}$$

$$= \begin{cases} 0 & \text{if } z \geq 1 \\ -1 & \text{if } z < 1 \end{cases} \tag{6.21}$$

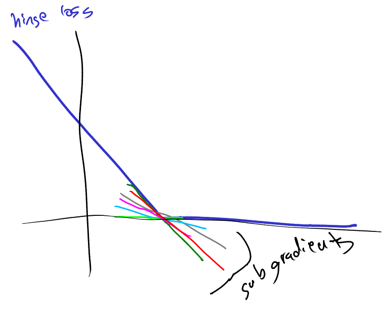


Figure 6.8: hinge loss with sub

Thus, the derivative is zero for $z < 1$ and -1 for $z > 1$, matching intuition from the Figure. At the non-differentiable point, $z = 1$, we can use a **subderivative**: a generalization of derivatives to non-differentiable functions. Intuitively, you can think of the derivative of f at z as the tangent line. Namely, it is the line that touches f at z that is always below f (for convex functions). The subderivative, denoted $\boldsymbol{\partial} f$, is the *set* of all such lines. At differentiable positions, this set consists just of the actual derivative. At non-differentiable positions, this contains all slopes that define lines that always lie under the function and make contact at the operating point. This is shown pictorally in Figure 6.8, where example subderivatives are shown for the hinge loss function. In the particular case of hinge loss, any value between 0 and -1 is a valid subderivative at $z = 0$. In fact, the subderivative is always a closed set of the form $[a, b]$, where a and b can be derived by looking at limits from the left and right.

This gives you a way of computing derivative-like things for non-differentiable functions. Take hinge loss as an example. For a given example n, the subgradient of hinge loss can be computed as:

$$\boldsymbol{\partial}_{\boldsymbol{w}} \max\{0, 1 - y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b)\} \tag{6.22}$$

$$= \boldsymbol{\partial}_{\boldsymbol{w}} \begin{cases} 0 & \text{if } y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 1 \\ y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) & \text{otherwise} \end{cases} \tag{6.23}$$

$$= \begin{cases} \boldsymbol{\partial}_{\boldsymbol{w}} 0 & \text{if } y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 1 \\ \boldsymbol{\partial}_{\boldsymbol{w}} y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) & \text{otherwise} \end{cases} \tag{6.24}$$

$$= \begin{cases} \boldsymbol{0} & \text{if } y_n(\boldsymbol{w} \cdot \boldsymbol{x}_n + b) > 1 \\ y_n \boldsymbol{x}_n & \text{otherwise} \end{cases} \tag{6.25}$$

Algorithm 23 HINGEREGULARIZEDGD($\mathbf{D}$, λ, $MaxIter$)

```
 1: $\boldsymbol{w} \leftarrow \langle 0, 0, \dots 0\rangle$ , $b \leftarrow 0$            // initialize weights and bias
 2: for $iter = 1 \dots MaxIter$ do
 3:   $\boldsymbol{g} \leftarrow \langle 0, 0, \dots 0\rangle$ , $g \leftarrow 0$       // initialize gradient of weights and bias
 4:   for all $(\boldsymbol{x},y) \in \mathbf{D}$ do
 5:     if $y(\boldsymbol{w} \cdot \boldsymbol{x} + b) \leq 1$ then
 6:       $\boldsymbol{g} \leftarrow \boldsymbol{g} + y\,\boldsymbol{x}$                   // update weight gradient
 7:       $g \leftarrow g + y$                          // update bias derivative
 8:     end if
 9:   end for
10:   $\boldsymbol{g} \leftarrow \boldsymbol{g} - \lambda\boldsymbol{w}$                        // add in regularization term
11:   $\boldsymbol{w} \leftarrow \boldsymbol{w} + \eta\boldsymbol{g}$                             // update weights
12:   $b \leftarrow b + \eta g$                                  // update bias
13: end for
14: return $\boldsymbol{w}$, $b$
```

MATH REVIEW | MATRIX MULTIPLICATION AND INVERSION

...

Figure 6.9:

If you plug this subgradient form into Algorithm 6.4, you obtain Algorithm 6.5. This is the **subgradient descent** for regularized hinge loss (with a 2-norm regularizer).

6.6 Closed-form Optimization for Squared Loss

Although gradient descent is a good, generic optimization algorithm, there are cases when you can do better. An example is the case of a 2-norm regularizer and squared error loss function. For this, you can actually obtain a *closed form* solution for the optimal weights. However, to obtain this, you need to rewrite the optimization problem in terms of matrix operations. For simplicity, we will only consider the *unbiased* version, but the extension is Exercise **??**. This is precisely the **linear regression** setting.

You can think of the training data as a large matrix $\mathbf{X}$ of size $N \times D$, where $X_{n,d}$ is the value of the dth feature on the nth example. You can think of the labels as a column ("tall") vector $\boldsymbol{Y}$ of dimension N. Finally, you can think of the weights as a column vector $\boldsymbol{w}$ of size D. Thus, the matrix-vector product $\boldsymbol{a} = \mathbf{X}\boldsymbol{w}$ has dimension N. In particular:

$$a_n = [\mathbf{X}\boldsymbol{w}]_n = \sum_d \mathbf{X}_{n,d} w_d \tag{6.26}$$

This means, in particular, that $\boldsymbol{a}$ is actually the predictions of the model. Instead of calling this $\boldsymbol{a}$, we will call it $\hat{\boldsymbol{Y}}$. The squared error says that we should minimize $\frac{1}{2}\sum_n(\hat{Y}_n - Y_n)^2$, which can be written in vector form as a minimization of $\frac{1}{2}\left|\left|\hat{\mathbf{Y}} - \mathbf{Y}\right|\right|^2$.

This can be expanded visually as:

Verify that the squared error can actually be written as this vector norm.

$$\underbrace{\begin{bmatrix} x_{1,1} & x_{1,2} & \dots & x_{1,D} \\ x_{2,1} & x_{2,2} & \dots & x_{2,D} \\ \vdots & \vdots & \ddots & \vdots \\ x_{N,1} & x_{N,2} & \dots & x_{N,D} \end{bmatrix}}_{\mathbf{X}} \underbrace{\begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_D \end{bmatrix}}_{w} = \underbrace{\begin{bmatrix} \sum_d x_{1,d}w_d \\ \sum_d x_{2,d}w_d \\ \vdots \\ \sum_d x_{N,d}w_d \end{bmatrix}}_{\hat{\mathbf{Y}}} \approx \underbrace{\begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_N \end{bmatrix}}_{\hat{\mathbf{Y}}} \tag{6.27}$$

So, compactly, our optimization problem can be written as:

$$\min_{w} \quad \mathcal{L}(w) = \frac{1}{2}\left|\left|\mathbf{X}w - \mathbf{Y}\right|\right|^2 + \frac{\lambda}{2}\left|\left|w\right|\right|^2 \tag{6.28}$$

If you recall from calculus, you can minimize a function by setting its derivative to zero. We start with the weights w and take gradients:

$$\nabla_w \mathcal{L}(w) = \mathbf{X}^\top(\mathbf{X}w - \mathbf{Y}) + \lambda w \tag{6.29}$$
$$= \mathbf{X}^\top\mathbf{X}w - \mathbf{X}^\top\mathbf{Y} + \lambda w \tag{6.30}$$
$$= \left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right) w - \mathbf{X}^\top\mathbf{Y} \tag{6.31}$$

We can equate this to zero and solve, yielding:

$$\left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}\right) w - \mathbf{X}^\top\mathbf{Y} = 0 \tag{6.32}$$
$$\Longleftrightarrow \quad \left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_D\right) w = \mathbf{X}^\top\mathbf{Y} \tag{6.33}$$
$$\Longleftrightarrow \quad w = \left(\mathbf{X}^\top\mathbf{X} + \lambda\mathbf{I}_D\right)^{-1}\mathbf{X}^\top\mathbf{Y} \tag{6.34}$$

Thus, the *optimal* solution of the weights can be computed by a few matrix multiplications and a matrix inversion. As a sanity check, you can make sure that the dimensions match. The matrix $\mathbf{X}^\top\mathbf{X}$ has dimension $D{\times}D$, and therefore so does the inverse term. The inverse is $D{\times}D$ and $\mathbf{X}^\top$ is $D{\times}N$, so that product is $D{\times}N$. Multiplying through by the $N{\times}1$ vector $\mathbf{Y}$ yields a $D{\times}1$ vector, which is precisely what we want for the weights.

Note that this gives an *exact solution*, modulo numerical innacuracies with computing matrix inverses. In contrast, gradient descent will give you progressively better solutions and will "eventually" converge to the optimum at a rate of $1/k$. This means that if you want an answer that's within an accuracy of $\epsilon = 10^{-4}$, you will need something on the order of one thousand steps.

For those who are keen on linear algebra, you might be worried that the matrix you must invert might not be invertible. Is this actually a problem?

The question is whether getting this exact solution is always more efficient. To run gradient descent for one step will take $\mathcal{O}(ND)$ time, with a relatively small constant. You will have to run K iterations, yielding an overall runtime of $\mathcal{O}(KND)$. On the other hand, the closed form solution requires constructing $\mathbf{X}^\top\mathbf{X}$, which takes $\mathcal{O}(D^2N)$ time. The inversion take $\mathcal{O}(D^3)$ time using standard matrix inversion routines. The final multiplications take $\mathcal{O}(ND)$ time. Thus, the overall runtime is on the order $\mathcal{O}(D^3 + D^2N)$. In most standard cases (though this is becoming less true over time), $N > D$, so this is dominated by $\mathcal{O}(D^2N)$.

Thus, the overall question is whether you will need to run more than D-many iterations of gradient descent. If so, then the matrix inversion will be (roughly) faster. Otherwise, gradient descent will be (roughly) faster. For low- and medium-dimensional problems (say, $D \leq 100$), it is probably faster to do the closed form solution via matrix inversion. For high dimensional problems ($D \geq 10,000$), it is probably faster to do gradient descent. For things in the middle, it's hard to say for sure.

6.7 *Support Vector Machines*

At the beginning of this chapter, you may have looked at the convex surrogate loss functions and asked yourself: where did these come from?! They are all derived from different underlying principles, which essentially correspond to different inductive biases.

Let's start by thinking back to the original goal of linear classifiers: to find a hyperplane that separates the positive training examples from the negative ones. Figure 6.10 shows some data and three potential hyperplanes: red, green and blue. Which one do you like best?

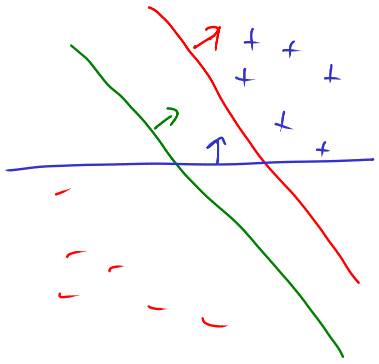


Figure 6.10: picture of data points with three hyperplanes, RGB with G the best

Most likely you chose the green hyperplane. And most likely you chose it because it was furthest away from the closest training points. In other words, it had a large **margin**. The desire for hyperplanes with large margins is a perfect example of an inductive bias. The data does not tell us which of the three hyperplanes is best: we have to choose one using some other source of information.

Following this line of thinking leads us to the **support vector machine** (SVM). This is simply a way of setting up an optimization problem that attempts to find a separating hyperplane with as large a margin as possible. It is written as a **constrained optimization problem**:

$$\min_{\boldsymbol{w},b} \quad \frac{1}{\gamma(\boldsymbol{w},b)} \tag{6.35}$$
$$\text{subj. to} \quad y_n\left(\boldsymbol{w}\cdot\boldsymbol{x}_n + b\right) \geq 1 \qquad (\forall n)$$

In this optimization, you are trying to find parameters that maximize the margin, denoted γ, (i.e., minimize the reciprocal of the margin) subject to the constraint that *all* training examples are correctly classified.

The "odd" thing about this optimization problem is that we require the classification of each point to be greater than *one* rather than simply greater than *zero*. However, the problem doesn't fundamentally change if you replace the "1" with any other positive constant (see Exercise **??**). As shown in Figure 6.11, the constant one can be interpreted visually as ensuring that there is a non-trivial margin between the positive points and negative points.

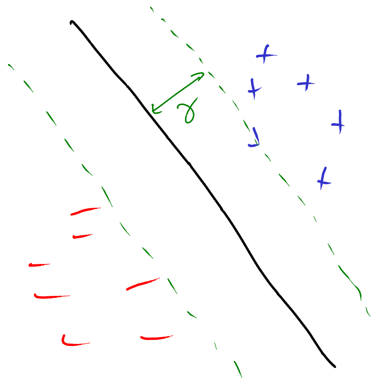


Figure 6.11: hyperplane with margins on sides

The difficulty with the optimization problem in Eq (**??**) is what happens with data that is not linearly separable. In that case, there *is no* set of parameters $\boldsymbol{w}, b$ that can simultaneously satisfy all the constraints. In optimization terms, you would say that the **feasible region** is *empty*. (The feasible region is simply the set of all parameters that satify the constraints.) For this reason, this is refered to as the **hard-margin SVM**, because enforcing the margin is a hard constraint. The question is: how to modify this optimization problem so that it can handle inseparable data.

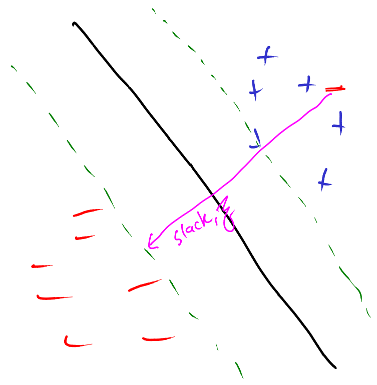


Figure 6.12: one bad point with slack

The key idea is the use of **slack parameters**. The intuition behind slack parameters is the following. Suppose we find a set of parameters $\boldsymbol{w}, b$ that do a really good job on 9999 data points. The points are perfectly classifed and you achieve a large margin. But there's one pesky data point left that cannot be put on the proper side of the margin: perhaps it is noisy. (See Figure 6.12.) You want to be able to pretend that you can "move" that point across the hyperplane on to the proper side. You will have to pay a little bit to do so, but as long as you aren't moving a *lot* of points around, it should be a good idea to do this. In this picture, the amount that you move the point is denoted ξ (xi).

By introducing one slack parameter for each training example, and penalizing yourself for having to use slack, you can create an objective function like the following, **soft-margin SVM**:

$$\min_{\boldsymbol{w},b,\xi} \quad \underbrace{\frac{1}{\gamma(\boldsymbol{w},b)}}_{\text{large margin}} + \underbrace{C\sum_n \xi_n}_{\text{small slack}} \tag{6.36}$$

$$\text{subj. to} \quad y_n\left(\boldsymbol{w}\cdot\boldsymbol{x}_n + b\right) \ge 1 - \xi_n \qquad (\forall n)$$

$$\xi_n \ge 0 \qquad (\forall n)$$

The goal of this objective function is to ensure that all points are correctly classified (the first constraint). But if a point n cannot be correctly classified, then you can set the slack ξ_n to something greater than zero to "move" it in the correct direction. However, for all non-zero slacks, you have to pay in the objective function proportional to the amount of slack. The hyperparameter $C > 0$ controls overfitting versus underfitting. The second constraint simply says that you must not have negative slack.

What values of C will lead to over-fitting? What values will lead to underfitting?

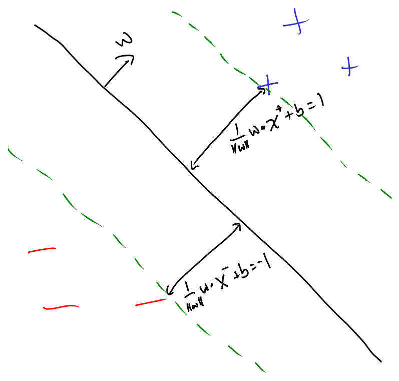
One major advantage of the soft-margin SVM over the original hard-margin SVM is that the feasible region is *never empty*. That is, there is always going to be some solution, regardless of whether your training data is linearly separable or not.

It's one thing to write down an optimization problem. It's another thing to try to solve it. There are a very large number of ways to optimize SVMs, essentially because they are such a popular learning model. Here, we will talk just about one, very simple way. More complex methods will be discussed later in this book once you have a bit more background.

Suppose I give you a data set. Without even looking at the data, construct for me a feasible solution to the soft-margin SVM. What is the value of the objective for this solution?

To make progress, you need to be able to measure the size of the margin. Suppose someone gives you parameters $\boldsymbol{w}, b$ that optimize the hard-margin SVM. We wish to measure the size of the margin. The first observation is that the hyperplane will lie *exactly* halfway between the nearest positive point and nearest negative point. If not, the margin could be made bigger by simply sliding it one way or the other by adjusting the bias b.

By this observation, there is some positive example that that lies exactly 1 unit from the hyperplane. Call it $\boldsymbol{x}^+$, so that $\boldsymbol{w} \cdot \boldsymbol{x}^+ + b = 1$. Similarly, there is some negative example, $\boldsymbol{x}^-$, that lies exactly on the other side of the margin: for which $\boldsymbol{w} \cdot \boldsymbol{x}^- + b = -1$. These two points, $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$ give us a way to measure the size of the margin. As shown in Figure 6.11, we can measure the size of the margin by looking at the difference between the lengths of projections of $\boldsymbol{x}^+$ and $\boldsymbol{x}^-$ onto the hyperplane. Since projection requires a normalized vector, we can measure the distances as:



w
$\frac{1}{\|w\|} w \cdot x^+ + b = 1$
$\frac{1}{\|w\|} w \cdot x^- + b = -1$

Figure 6.13: copy of figure from p5 of cs544 svm tutorial

$$d^+ = \frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^+ + b - 1 \tag{6.37}$$

$$d^- = -\frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^- - b + 1 \tag{6.38}$$

We can then compute the margin by algebra:

$$\begin{aligned}
\gamma &= \frac{1}{2}\left[d^+ - d^-\right] && (6.39)\\
&= \frac{1}{2}\left[\frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^+ + b - 1 - \frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^- - b + 1\right] && (6.40)\\
&= \frac{1}{2}\left[\frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^+ - \frac{1}{||\boldsymbol{w}||} \boldsymbol{w} \cdot \boldsymbol{x}^-\right] && (6.41)\\
&= \frac{1}{2}\left[\frac{1}{||\boldsymbol{w}||}(+1) - \frac{1}{||\boldsymbol{w}||}(-1)\right] && (6.42)\\
&= \frac{1}{||\boldsymbol{w}||} && (6.43)
\end{aligned}$$

This is a remarkable conclusion: the size of the margin is inversely proportional to the norm of the weight vector. Thus, **maximizing the margin is equivalent to minimizing $||w||$!** This serves as an additional justification of the 2-norm regularizer: having small weights means having large margins!

However, our goal wasn't to justify the regularizer: it was to understand hinge loss. So let us go back to the soft-margin SVM and plug in our new knowledge about margins:

$$\min_{\boldsymbol{w},b,\xi} \quad \underbrace{\frac{1}{2}||\boldsymbol{w}||^2}_{\text{large margin}} + \underbrace{C\sum_n \xi_n}_{\text{small slack}} \tag{6.44}$$

$$\text{subj. to} \quad y_n\left(\boldsymbol{w}\cdot\boldsymbol{x}_n + b\right) \geq 1 - \xi_n \qquad (\forall n)$$

$$\xi_n \geq 0 \qquad (\forall n)$$

Now, let's play a thought experiment. Suppose someone handed you a solution to this optimization problem that consisted of weights ($\boldsymbol{w}$) and a bias (b), but they forgot to give you the slacks. Could you recover the slacks from the information you have?

In fact, the answer is yes! For simplicity, let's consider positive examples. Suppose that you look at some positive example $\boldsymbol{x}_n$. You need to figure out what the slack, ξ_n, would have been. There are two cases. Either $\boldsymbol{w}\cdot\boldsymbol{x}_n + b$ is at least 1 or it is not. If it's large enough, then you want to set $\xi_n = 0$. Why? It cannot be less than zero by the second constraint. Moreover, if you set it greater than zero, you will "pay" unnecessarily in the objective. So in this case, $\xi_n = 0$. Next, suppose that $\boldsymbol{w}\cdot\boldsymbol{x}_n + b = 0.2$, so it is not big enough. In order to satisfy the first constraint, you'll need to set $\xi_n \geq 0.8$. But because of the objective, you'll not want to set it any larger than necessary, so you'll set $\xi_n = 0.8$ exactly.

Following this argument through for both positive and negative points, if someone gives you solutions for $\boldsymbol{w}, b$, you can automatically compute the optimal ξ variables as:

$$\xi_n = \begin{cases} 0 & \text{if } y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b) \geq 1 \\ 1 - y_n(\boldsymbol{w}\cdot\boldsymbol{x}_n + b) & \text{otherwise} \end{cases} \tag{6.45}$$

In other words, the optimal value for a slack variable is *exactly* the hinge loss on the corresponding example! Thus, we can write the SVM objective as an *unconstrained* optimization problem:

$$\min_{\boldsymbol{w},b} \quad \underbrace{\frac{1}{2}||\boldsymbol{w}||^2}_{\text{large margin}} + \underbrace{C\sum_n \ell^{(\text{hin})}(y_n, \boldsymbol{w}\cdot\boldsymbol{x}_n + b)}_{\text{small slack}} \tag{6.46}$$

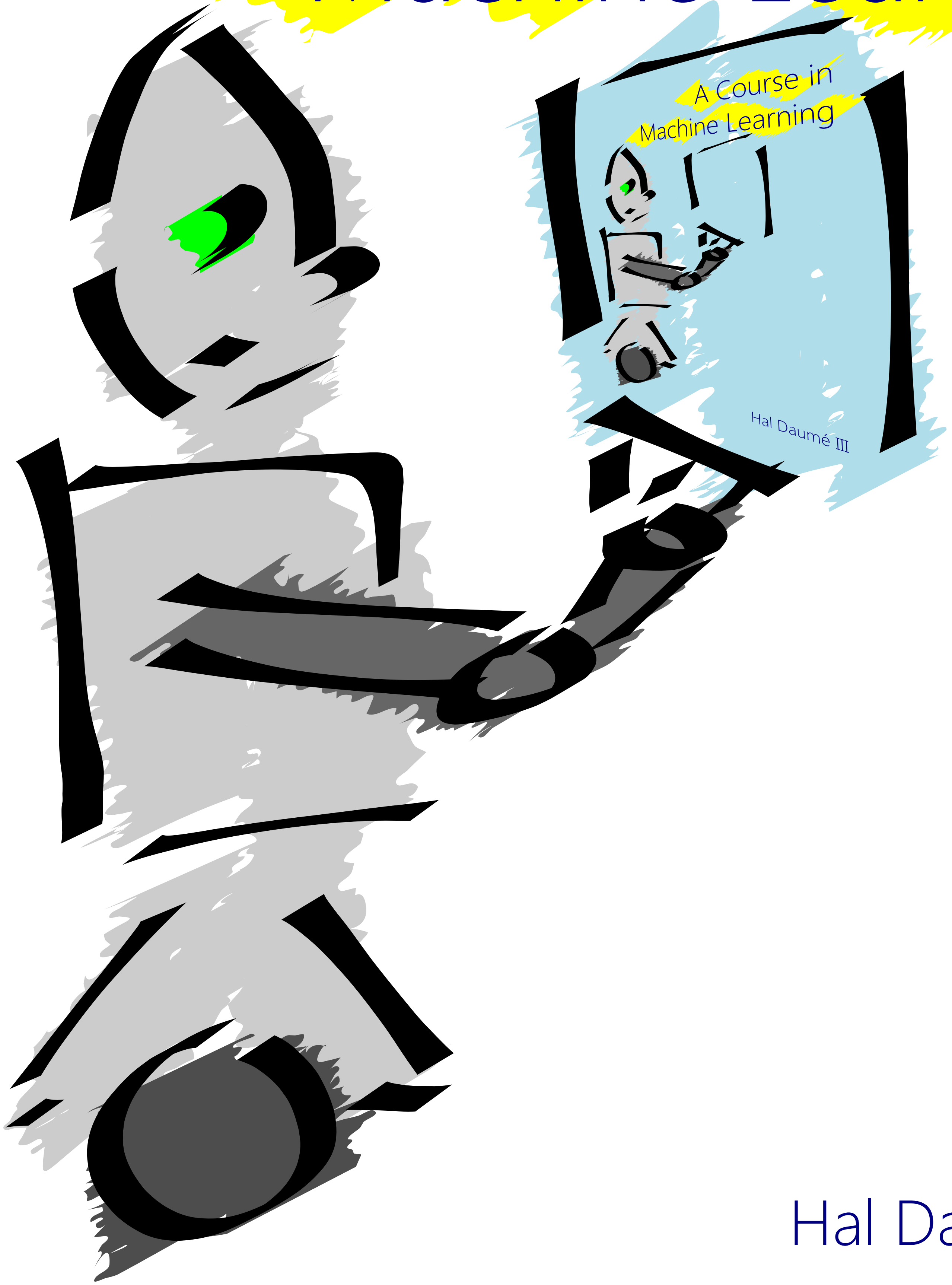
Multiplying this objective through by λ/C, we obtain exactly the regularized objective from Eq (6.8) with hinge loss as the loss function and the 2-norm as the regularizer!

TODO: justify in term of one dimensional projections!

6.8 Exercises

Exercise 6.1. TODO. . .

A Course in Machine Learning



Hal Daumé III

8 | Neural Networks

–

Learning Objectives:

- Explain the biological inspiration for multi-layer neural networks.
- Construct a two-layer network that can solve the XOR problem.
- Implement the back-propogation algorithm for training multi-layer networks.
- Explain the trade-off between depth and breadth in network structure.
- Contrast neural networks with radial basis functions with k-nearest neighbor learning.

Dependencies:

The first learning models you learned about (decision trees and nearest neighbor models) created complex, **non-linear** decision boundaries. We moved from there to the perceptron, perhaps the most classic linear model. At this point, we will move *back* to non-linear learning models, but using all that we have learned about linear learning thus far.

This chapter presents an extension of perceptron learning to non-linear decision boundaries, taking the biological inspiration of neurons even further. In the perceptron, we thought of the input data point (e.g., an image) as being directly connected to an output (e.g., label). This is often called a **single-layer network** because there is one layer of weights. Now, instead of directly connecting the inputs to the outputs, we will insert a layer of "hidden" nodes, moving from a single-layer network to a **multi-layer network**. But introducing a non-linearity at inner layers, this will give us non-linear decision boundaires. In fact, such networks are able to express almost any function we want, not just linear functions. The trade-off for this flexibility is increased complexity in parameter tuning and model design.

8.1 *Bio-inspired Multi-Layer Networks*

One of the major weaknesses of linear models, like perceptron and the regularized linear models from the previous chapter, is that they are linear! Namely, they are unable to learn arbitrary decision boundaries. In contrast, decision trees and KNN *could* learn arbitrarily complicated decision boundaries.

One approach to doing this is to chain together a collection of perceptrons to build more complex **neural networks**. An example of a **two-layer network** is shown in Figure 8.1. Here, you can see five inputs (features) that are fed into two **hidden units**. These hidden units are then fed in to a single **output unit**. Each edge in this figure corresponds to a different weight. (Even though it looks like there are three layers, this is called a two-layer network because we don't count

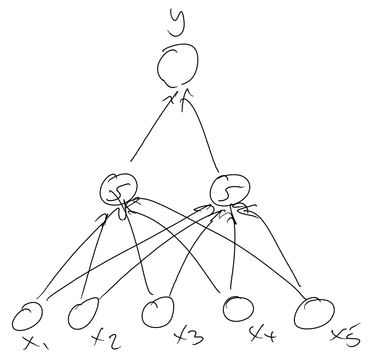


Figure 8.1: picture of a two-layer network with 5 inputs and two hidden units

the inputs as a real layer. That is, it's two layers of *trained* weights.)

Prediction with a neural network is a straightforward generalization of prediction with a perceptron. First you compute activations of the nodes in the hidden unit based on the inputs and the input weights. Then you compute activations of the output unit given the hidden unit activations and the second layer of weights.

The only major difference between this computation and the perceptron computation is that the hidden units compute a *non-linear* function of their inputs. This is usually called the **activation function** or **link function**. More formally, if $w_{i,d}$ is the weights on the edge connecting input d to hidden unit i, then the activation of hidden unit i is computed as:

$$h_i = f\left(\boldsymbol{w}_i \cdot \boldsymbol{x}\right) \tag{8.1}$$

Where f is the link function and $\boldsymbol{w}_i$ refers to the vector of weights feeding in to node i.

One example link function is the **sign** function. That is, if the incoming signal is negative, the activation is -1. Otherwise the activation is $+1$. This is a potentially useful activiation function, but you might already have guessed the problem with it: it is non-differentiable.

EXPLAIN BIAS!!!

A more popular link function is the **hyperbolic tangent** function, tanh. A comparison between the sign function and the tanh function is in Figure 8.2. As you can see, it is a reasonable approximation to the sign function, but is convenient in that it is differentiable.[1] Because it looks like an "S" and because the Greek character for "S" is "Sigma," such functions are usually called **sigmoid** functions.

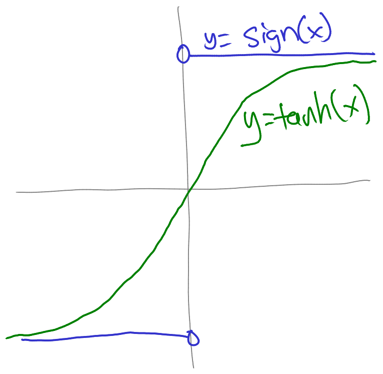


Figure 8.2: picture of sign versus tanh

[1] It's derivative is just $1 - \tanh^2(x)$.

Assuming for now that we are using tanh as the link function, the overall prediction made by a two-layer network can be computed using Algorithm 8.1. This function takes a matrix of weights $\mathbf{W}$ corresponding to the first layer weights and a vector of weights $\boldsymbol{v}$ corresponding to the second layer. You can write this entire computation out in one line as:

$$\hat{y} = \sum_i v_i \tanh(\boldsymbol{w}_i \cdot \hat{\boldsymbol{x}}) \tag{8.2}$$

$$= \boldsymbol{v} \cdot \tanh(\mathbf{W}\hat{\boldsymbol{x}}) \tag{8.3}$$

Where the second line is short hand assuming that tanh can take a vector as input and product a vector as output.

? Is it necessary to use a link function at all? What would happen if you just used the identify function as a link?

Algorithm 24 TwoLayerNetworkPredict($\mathbf{W}$, $\boldsymbol{v}$, $\hat{\boldsymbol{x}}$)

```
1: for $i$ = 1 to number of hidden units do
2:     $h_i \leftarrow \tanh(\boldsymbol{w}_i \cdot \hat{\boldsymbol{x}})$        // compute activation of hidden unit $i$
3: end for
4: return $\boldsymbol{v} \cdot \boldsymbol{h}$        // compute output unit
```

The claim is that two-layer neural networks are more expressive than single layer networks (i.e., perceptrons). To see this, you can construct a very small two-layer network for solving the XOR problem. For simplicity, suppose that the data set consists of four data points, given in Table 8.1. The classification rule is that $y = +1$ if an only if $x_1 = x_2$, where the features are just ± 1.

y	x_0	x_1	x_2
+1	+1	+1	+1
+1	+1	-1	-1
-1	+1	+1	-1
-1	+1	-1	+1

Table 8.1: Small XOR data set.

You can solve this problem using a two layer network with two hidden units. The key idea is to make the first hidden unit compute an "or" function: $x_1 \vee x_2$. The second hidden unit can compute an "and" function: $x_1 \wedge x_2$. The the output can combine these into a single prediction that mimics XOR. Once you have the first hidden unit activate for "or" and the second for "and," you need only set the output weights as -2 and $+1$, respectively.

? Verify that these output weights will actually give you XOR.

To achieve the "or" behavior, you can start by setting the bias to -0.5 and the weights for the two "real" features as both being 1. You can check for yourself that this will do the "right thing" if the link function were the sign function. Of course it's not, it's tanh. To get tanh to mimic sign, you need to make the dot product either really really large or really really small. You can accomplish this by setting the bias to $-500,000$ and both of the two weights to $1,000,000$. Now, the activation of this unit will be just slightly above -1 for $\boldsymbol{x} = \langle -1, -1 \rangle$ and just slightly below $+1$ for the other three examples.

? This shows how to create an "or" function. How can you create an "and" function?

At this point you've seen that one-layer networks (aka perceptrons) can represent any linear function and only linear functions. You've also seen that two-layer networks can represent non-linear functions like XOR. A natural question is: do you get additional representational power by moving beyond two layers? The answer is partially provided in the following Theorem, due originally to George Cybenko for one particular type of link function, and extended later by Kurt Hornik to arbitrary link functions.

Theorem 9 (Two-Layer Networks are Universal Function Approximators)**.** *Let F be a continuous function on a bounded subset of D-dimensional space. Then there exists a two-layer neural network $\hat{F}$ with a finite number of hidden units that approximate F arbitrarily well. Namely, for all $\boldsymbol{x}$ in the domain of F, $\left|F(\boldsymbol{x}) - \hat{F}(\boldsymbol{x})\right| < \epsilon$.*

Or, in colloquial terms "two-layer networks can approximate any function."

This is a remarkable theorem. Practically, it says that if you give me a function F and some error tolerance parameter ϵ, I can construct a two layer network that computes F. In a sense, it says that going from one layer to two layers completely changes the representational capacity of your model.

When working with two-layer networks, the key question is: how many hidden units should I have? If your data is D dimensional and you have K hidden units, then the total number of parameters is $(D+2)K$. (The first $+1$ is from the bias, the second is from the second layer of weights.) Following on from the heuristic that you should have one to two examples for each parameter you are trying to estimate, this suggests a method for choosing the number of hidden units as roughly $\lfloor \frac{N}{D} \rfloor$. In other words, if you have tons and tons of examples, you can safely have lots of hidden units. If you only have a few examples, you should probably restrict the number of hidden units in your network.

The number of units is both a form of inductive bias and a form of regularization. In both view, the number of hidden units controls how complex your function will be. Lots of hidden units $\Rightarrow$ very complicated function. Figure **??** shows training and test error for neural networks trained with different numbers of hidden units. As the number increases, training performance continues to get better. But at some point, test performance gets worse because the network has overfit the data.

8.2 *The Back-propagation Algorithm*

The back-propagation algorithm is a classic approach to training neural networks. Although it was not originally seen this way, based on what you know from the last chapter, you can summarize back-propagation as:

$$\text{back-propagation} = \text{gradient descent} + \text{chain rule} \tag{8.4}$$

More specifically, the set up is *exactly* the same as before. You are going to optimize the weights in the network to minimize some objective function. The only difference is that the predictor is no longer linear (i.e., $\hat{y} = \boldsymbol{w} \cdot \boldsymbol{x} + b$) but now non-linear (i.e., $\boldsymbol{v} \cdot \tanh(\mathbf{W}\hat{\boldsymbol{x}})$). The only question is how to do gradient descent on this more complicated objective.

For now, we will ignore the idea of regularization. This is for two reasons. The first is that you already know how to deal with regularization, so everything you've learned before applies. The second is that *historically*, neural networks have not been regularized. Instead, people have used **early stopping** as a method for controlling overfitting. Presently, it's not obvious which is a better solution: both are valid options.

To be completely explicit, we will focus on optimizing squared error. Again, this is mostly for historic reasons. You could easily replace squared error with your loss function of choice. Our overall objective is:

$$\min_{\mathbf{W},\boldsymbol{v}} \quad \sum_n \frac{1}{2}\left(y_n - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x}_n)\right)^2 \tag{8.5}$$

Here, f is some link function like tanh.

The easy case is to differentiate this with respect to $\boldsymbol{v}$: the weights for the output unit. Without even doing any math, you should be able to guess what this looks like. The way to think about it is that from $\boldsymbol{v}$s perspective, it is just a linear model, attempting to minimize squared error. The only "funny" thing is that its inputs are the activations $\boldsymbol{h}$ rather than the examples $\boldsymbol{x}$. So the gradient with respect to $\boldsymbol{v}$ is just as for the linear case.

To make things notationally more convenient, let e_n denote the *error* on the nth example (i.e., the blue term above), and let $\boldsymbol{h}_n$ denote the vector of hidden unit activations on that example. Then:

$$\nabla_{\boldsymbol{v}} = -\sum_n e_n \boldsymbol{h}_n \tag{8.6}$$

This is exactly like the linear case. One way of interpreting this is: how would the output weights have to change to make the prediction better? This is an easy question to answer because they can easily measure how their changes affect the output.

The more complicated aspect to deal with is the weights corresponding to the *first* layer. The reason this is difficult is because the weights in the first layer aren't necessarily trying to produce specific values, say 0 or 5 or -2.1. They are simply trying to produce activations that get fed to the output layer. So the change they want to make depends crucially on how the output layer interprets them.

Thankfully, the chain rule of calculus saves us. Ignoring the sum over data points, we can compute:

$$\mathcal{L}(\mathbf{W}) = \frac{1}{2}\left(y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x})\right)^2 \tag{8.7}$$

$$\frac{\partial \mathcal{L}}{\partial \boldsymbol{w}_i} = \frac{\partial \mathcal{L}}{\partial f_i}\frac{\partial f_i}{\partial \boldsymbol{w}_i} \tag{8.8}$$

$$\frac{\partial \mathcal{L}}{\partial f_i} = -\left(y - \sum_i v_i f(\boldsymbol{w}_i \cdot \boldsymbol{x})\right) v_i = -e v_i \tag{8.9}$$

$$\frac{\partial f_i}{\partial \boldsymbol{w}_i} = f'(\boldsymbol{w}_i \cdot \boldsymbol{x})\boldsymbol{x} \tag{8.10}$$

Algorithm 25 TwoLayerNetworkTrain($\mathbf{D}$, η, K, $MaxIter$)

```
W ← D×K matrix of small random values          // initialize input layer weights
v ← K-vector of small random values            // initialize output layer weights
for iter = 1 ... MaxIter do
    G ← D×K matrix of zeros                    // initialize input layer gradient
    g ← K-vector of zeros                      // initialize output layer gradient
    for all (x,y) ∈ D do
        for i = 1 to K do
            $a_i \leftarrow \boldsymbol{w}_i \cdot \hat{\boldsymbol{x}}$
            $h_i \leftarrow \tanh(a_i)$                 // compute activation of hidden unit i
        end for
        $\hat{y} \leftarrow \boldsymbol{v} \cdot \boldsymbol{h}$                      // compute output unit
        $e \leftarrow y - \hat{y}$                      // compute error
        $\boldsymbol{g} \leftarrow \boldsymbol{g} - e\boldsymbol{h}$                    // update gradient for output layer
        for i = 1 to K do
            $\mathbf{G}_i \leftarrow \mathbf{G}_i - ev_i(1 - \tanh^2(a_i))\boldsymbol{x}$   // update gradient for input layer
        end for
    end for
    $\mathbf{W} \leftarrow \mathbf{W} - \eta\mathbf{G}$                   // update input layer weights
    $\boldsymbol{v} \leftarrow \boldsymbol{v} - \eta\boldsymbol{g}$                   // update output layer weights
end for
return W, v
```

Putting this together, we get that the gradient with respect to $\boldsymbol{w}_i$ is:

$$\nabla_{\boldsymbol{w}_i} = -ev_i f'(\boldsymbol{w}_i \cdot \boldsymbol{x})\boldsymbol{x} \tag{8.11}$$

Intuitively you can make sense of this. If the overall error of the predictor (e) is small, you want to make small steps. If v_i is small for hidden unit i, then this means that the output is not particularly sensitive to the activation of the ith hidden unit. Thus, its gradient should be small. If v_i flips sign, the gradient at $\boldsymbol{w}_i$ should also flip signs. The name **back-propagation** comes from the fact that you propagate gradients backward through the network, starting at the end.

The complete instantiation of gradient descent for a two layer network with K hidden units is sketched in Algorithm 8.2. Note that this really is *exactly* a gradient descent algorithm; the only different is that the computation of the gradients of the input layer is moderately complicated.

? What would happen to this algorithm if you wanted to optimize exponential loss instead of squared error? What if you wanted to add in weight regularization?

As a bit of practical advice, implementing the back-propagation algorithm can be a bit tricky. Sign errors often abound. A useful trick is first to keep $\mathbf{W}$ fixed and work on just training $\boldsymbol{v}$. Then keep $\boldsymbol{v}$ fixed and work on training $\mathbf{W}$. Then put them together.

? If you like matrix calculus, derive the same algorithm starting from Eq (8.3).

8.3 Initialization and Convergence of Neural Networks

Based on what you know about linear models, you might be tempted to initialize all the weights in a neural network to zero. You might also have noticed that in Algorithm **??**, this is not what's done: they're initialized to small random values. The question is why?

The answer is because an initialization of $\mathbf{W} = \mathbf{0}$ and $\boldsymbol{v} = \mathbf{0}$ will lead to "uninteresting" solutions. In other words, if you initialize the model in this way, it will eventually get stuck in a bad local optimum. To see this, first realize that on any example $\boldsymbol{x}$, the activation h_i of the hidden units will all be zero since $\mathbf{W} = \mathbf{0}$. This means that on the first iteration, the gradient on the output weights ($\boldsymbol{v}$) will be zero, so they will stay put. Furthermore, the gradient $w_{1,d}$ for the dth feature on the ith unit will be *exactly* the same as the gradient $w_{2,d}$ for the same feature on the second unit. This means that the weight matrix, after a gradient step, will change in *exactly the same way* for every hidden unit. Thinking through this example for iterations $2\dots$, the values of the hidden units will *always* be exactly the same, which means that the weights feeding in to any of the hidden units will be exactly the same. Eventually the model will converge, but it will converge to a solution that does not take advantage of having access to the hidden units.

This shows that neural networks are *sensitive* to their initialization. In particular, the function that they optimize is **non-convex**, meaning that it might have plentiful local optima. (One of which is the trivial local optimum described in the preceding paragraph.) In a sense, neural networks *must* have local optima. Suppose you have a two layer network with two hidden units that's been optimized. You have weights $\boldsymbol{w}_1$ from inputs to the first hidden unit, weights $\boldsymbol{w}_2$ from inputs to the second hidden unit and weights (v_1, v_2) from the hidden units to the output. If I give you back another network with $\boldsymbol{w}_1$ and $\boldsymbol{w}_2$ swapped, and v_1 and v_2 swapped, the network computes *exactly* the same thing, but with a markedly different weight structure. This phenomena is known as **symmetric modes** ("mode" referring to an optima) meaning that there are symmetries in the weight space. It would be one thing if there were lots of modes and they were all symmetric: then finding one of them would be as good as finding any other. Unfortunately there are additional local optima that are *not* global optima.

Random initialization of the weights of a network is a way to address *both* of these problems. By initializing a network with small random weights (say, uniform between -0.1 and 0.1), the network is unlikely to fall into the trivial, symmetric local optimum. Moreover, by training a collection of networks, each with a different random

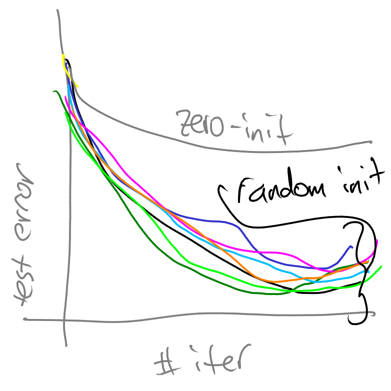


Figure 8.3: convergence of randomly initialized networks

initialization, you can often obtain better solutions that with just one initialization. In other words, you can train ten networks with different random seeds, and then pick the one that does best on held-out data. Figure 8.3 shows prototypical *test-set* performance for ten networks with different random initialization, plus an eleventh plot for the trivial symmetric network initialized with zeros.

One of the typical complaints about neural networks is that they are finicky. In particular, they have a rather large number of knobs to tune:

1. The number of layers
2. The number of hidden units per layer
3. The gradient descent learning rate η
4. The initialization
5. The stopping iteration or weight regularization

The last of these is minor (early stopping is an easy regularization method that does not require much effort to tune), but the others are somewhat significant. Even for two layer networks, having to choose the number of hidden units, and then get the learning rate and initialization "right" can take a bit of work. Clearly it can be automated, but nonetheless it takes time.

Another difficulty of neural networks is that their weights can be difficult to interpret. You've seen that, for linear networks, you can often interpret high weights as indicative of positive examples and low weights as indicative of negative examples. In multilayer networks, it becomes very difficult to try to understand what the different hidden units are doing.

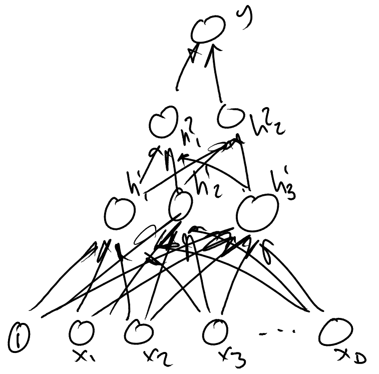


Figure 8.4: multi-layer network

8.4 Beyond Two Layers

The definition of neural networks and the back-propagation algorithm can be generalized beyond two layers to any arbitrary directed acyclic graph. In practice, it is most common to use a layered network like that shown in Figure 8.4 unless one has a very strong reason (aka inductive bias) to do something different. However, the view as a directed graph sheds a different sort of insight on the back-propagation algorithm.

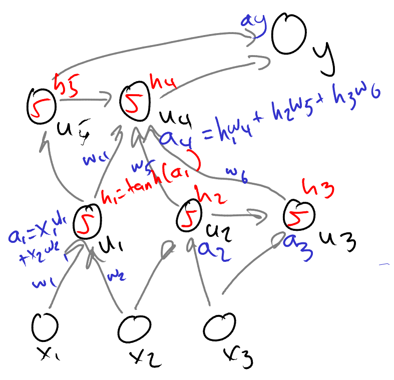


Figure 8.5: DAG network

Suppose that your network structure is stored in some directed acyclic graph, like that in Figure 8.5. We index nodes in this graph as u, v. The activation *before* applying non-linearity at a node is a_u and after non-linearity is h_u. The graph has a single sink, which is the output node y with activation a_y (no non-linearity is performed

Algorithm 26 FORWARDPROPAGATION($\boldsymbol{x}$)

```
1: for all input nodes $u$ do
2:     $h_u \leftarrow$ corresponding feature of $\boldsymbol{x}$
3: end for
4: for all nodes $v$ in the network whose parent's are computed do
5:     $a_v \leftarrow \sum_{u \in par(v)} w_{(u,v)} h_u$
6:     $h_v \leftarrow \tanh(a_v)$
7: end for
8: return $a_y$
```

Algorithm 27 BACKPROPAGATION($\boldsymbol{x}$, y)

```
1: run FORWARDPROPAGATION($\boldsymbol{x}$) to compute activations
2: $e_y \leftarrow y - a_y$                                   // compute overall network error
3: for all nodes $v$ in the network whose error $e_v$ is computed do
4:     for all $u \in par(v)$ do
5:         $g_{u,v} \leftarrow -e_v h_u$                          // compute gradient of this edge
6:         $e_u \leftarrow e_u + e_v w_{u,v}(1 - \tanh^2(a_u))$  // compute the "error" of the parent node
7:     end for
8: end for
9: return all gradients $g_e$
```

on the output unit). The graph has D-many inputs (i.e., nodes with no parent), whose activations h_u are given by an input example. An edge (u, v) is from a parent to a child (i.e., from an input to a hidden unit, or from a hidden unit to the sink). Each edge has a weight $w_{u,v}$. We say that $par(u)$ is the set of parents of u.

There are two relevant algorithms: forward-propagation and back-propagation. Forward-propagation tells you how to compute the activation of the sink y given the inputs. Back-propagation computes derivatives of the edge weights for a given input.

The key aspect of the **forward-propagation** algorithm is to iteratively compute activations, going deeper and deeper in the DAG. Once the activations of all the parents of a node u have been computed, you can compute the activation of node u. This is spelled out in Algorithm 8.4. This is also explained pictorially in Figure 8.6.

Back-propagation (see Algorithm 8.4) does the opposite: it computes gradients top-down in the network. The key idea is to compute an *error* for each node in the network. The error at the output unit is the "true error." For any input unit, the error is the amount of gradient that we see coming from our children (i.e., higher in the network). These errors are computed backwards in the network (hence the name **back-propagation**) along with the gradients themselves. This is also explained pictorially in Figure 8.7.

Given the back-propagation algorithm, you can directly run gradient descent, using it as a subroutine for computing the gradients.

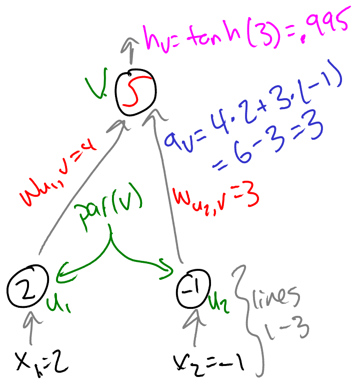
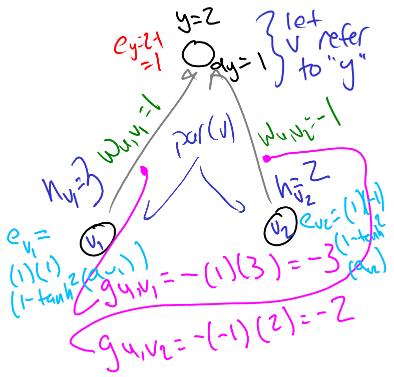


Figure 8.6: picture of forward prop



$y = 2$
$e_y = 2 - 1 = 1$
$a_y = 1$
let v refer to "y"
$w_{u,v_1} = 1$
$w_{u,v_2} = -1$
$par(v)$
$h_{v_1} = 3$
$h_{v_2} = 2$
v_1
v_2
$e_{v_1} = (1)(1)(1 - \tanh^2(a_{v_1}))$
$e_{v_2} = (1)(-1)(1 - \tanh^2(a_{v_2}))$
$g_{u,v_1} = -(1)(3) = -3$
$g_{u,v_2} = -(-1)(2) = -2$

Figure 8.7: picture of back prop

8.5 Breadth versus Depth

At this point, you've seen how to train two-layer networks and how to train arbitrary networks. You've also seen a theorem that says that two-layer networks are universal function approximators. This begs the question: if two-layer networks are so great, why do we care about deeper networks?

To understand the answer, we can borrow some ideas from CS theory, namely the idea of **circuit complexity**. The goal is to show that there are functions for which it might be a "good idea" to use a deep network. In other words, there are functions that will require a huge number of hidden units if you force the network to be shallow, but can be done in a small number of units if you allow it to be deep. The example that we'll use is the **parity function** which, ironically enough, is just a generalization of the XOR problem. The function is defined over binary inputs as:

$$parity(\boldsymbol{x}) = \sum_d \boldsymbol{x}_d \mod 2 \tag{8.12}$$

$$= \begin{cases} 1 & \text{if the number of 1s in } \boldsymbol{x} \text{ is odd} \\ 0 & \text{if the number of 1s in } \boldsymbol{x} \text{ is even} \end{cases} \tag{8.13}$$

It is easy to define a circuit of depth $\mathcal{O}(\log_2 D)$ with $\mathcal{O}(D)$-many gates for computing the parity function. Each gate is an XOR, arranged in a complete binary tree, as shown in Figure 8.8. (If you want to disallow XOR as a gate, you can fix this by allowing the depth to be doubled and replacing each XOR with an AND, OR and NOT combination, like you did at the beginning of this chapter.)

Figure 8.8: `nnet:paritydeep`: deep function for computing parity

This shows that if you are allowed to be deep, you can construct a circuit with that computes parity using a number of hidden units that is linear in the dimensionality. So can you do the same with shallow circuits? The answer is no. It's a famous result of circuit complexity that parity requires exponentially many gates to compute in constant depth. The formal theorem is below:

Theorem 10 (Parity Function Complexity)**.** *Any circuit of depth $K < \log_2 D$ that computes the parity function of D input bits must contain $\mathcal{O}e^D$ gates.*

This is a very famous result because it shows that constant-depth circuits are less powerful that deep circuits. Although a neural network isn't exactly the same as a circuit, the is generally believed that the same result holds for neural networks. At the very least, this gives a strong indication that depth might be an important consideration in neural networks.

? What is it about neural networks that makes it so that the theorem about circuits does not apply directly?

One way of thinking about the issue of breadth versus depth has to do with the number of *parameters* that need to be estimated. By the heuristic that you need roughly one or two examples for every parameter, a deep model could potentially require exponentially fewer examples to train than a shallow model!

This now flips the question: if deep is potentially so much better, why doesn't everyone use deep networks? There are at least two answers. First, it makes the **architecture selection** problem more significant. Namely, when you use a two-layer network, the only hyperparameter to choose is how many hidden units should go in the middle layer. When you choose a deep network, you need to choose how many layers, and what is the width of all those layers. This can be somewhat daunting.

A second issue has to do with training deep models with back-propagation. In general, as back-propagation works its way down through the model, the sizes of the gradients shrink. You can work this out mathematically, but the intuition is simpler. If you are the beginning of a very deep network, changing one single weight is unlikely to have a significant effect on the output, since it has to go through so many other units before getting there. This directly implies that the derivatives are small. This, in turn, means that back-propagation essentially never moves far from its initialization when run on very deep networks.

? While these small derivatives might make training difficult, they might be *good* for other reasons: what reasons?

Finding good ways to train deep networks is an active research area. There are two general strategies. The first is to attempt to initialize the weights better, often by a **layer-wise** initialization strategy. This can be often done using unlabeled data. After this initialization, back-propagation can be run to tweak the weights for whatever classification problem you care about. A second approach is to use a more complex optimization procedure, rather than gradient descent. You will learn about some such procedures later in this book.

8.6 Basis Functions

At this point, we've seen that: (a) neural networks can mimic linear functions and (b) they can learn more complex functions. A reasonable question is whether they can mimic a KNN classifier, and whether they can do it efficiently (i.e., with not-too-many hidden units).

A natural way to train a neural network to mimic a KNN classifier is to replace the sigmoid link function with a **radial basis function** (RBF). In a **sigmoid network** (i.e., a network with sigmoid links), the hidden units were computed as $h_i = \tanh(\boldsymbol{w}_i, \boldsymbol{x}\cdot)$. In an **RBF network**, the hidden units are computed as:

$$h_i = \exp\left[-\gamma_i \, ||\boldsymbol{w}_i - \boldsymbol{x}||^2\right] \tag{8.14}$$

In other words, the hidden units behave like little Gaussian "bumps" centered around locations specified by the vectors $\boldsymbol{w}_i$. A one-dimensional example is shown in Figure 8.9. The parameter γ_i specifies the *width* of the Gaussian bump. If γ_i is large, then only data points that are really close to $\boldsymbol{w}_i$ have non-zero activations. To distinguish sigmoid networks from RBF networks, the hidden units are typically drawn with sigmoids or with Gaussian bumps, as in Figure 8.10.

Figure 8.9: `nnet:rbfpicture`: a one-D picture of RBF bumps

Training RBF networks involves finding good values for the Gassian widths, γ_i, the centers of the Gaussian bumps, $\boldsymbol{w}_i$ and the connections between the Gaussian bumps and the output unit, $\boldsymbol{v}$. This can all be done using back-propagation. The gradient terms for $\boldsymbol{v}$ remain unchanged from before, the the derivates for the other variables differ (see Exercise **??**).

One of the big questions with RBF networks is: where should the Gaussian bumps be centered? One can, of course, apply back-propagation to attempt to find the centers. Another option is to specify them ahead of time. For instance, one potential approach is to have one RBF unit per data point, centered on that data point. If you carefully choose the γs and vs, you can obtain something that looks nearly identical to distance-weighted KNN by doing so. This has the added advantage that you can go futher, and use back-propagation to *learn* good Gaussian widths (γ) and "voting" factors (v) for the nearest neighbor algorithm.

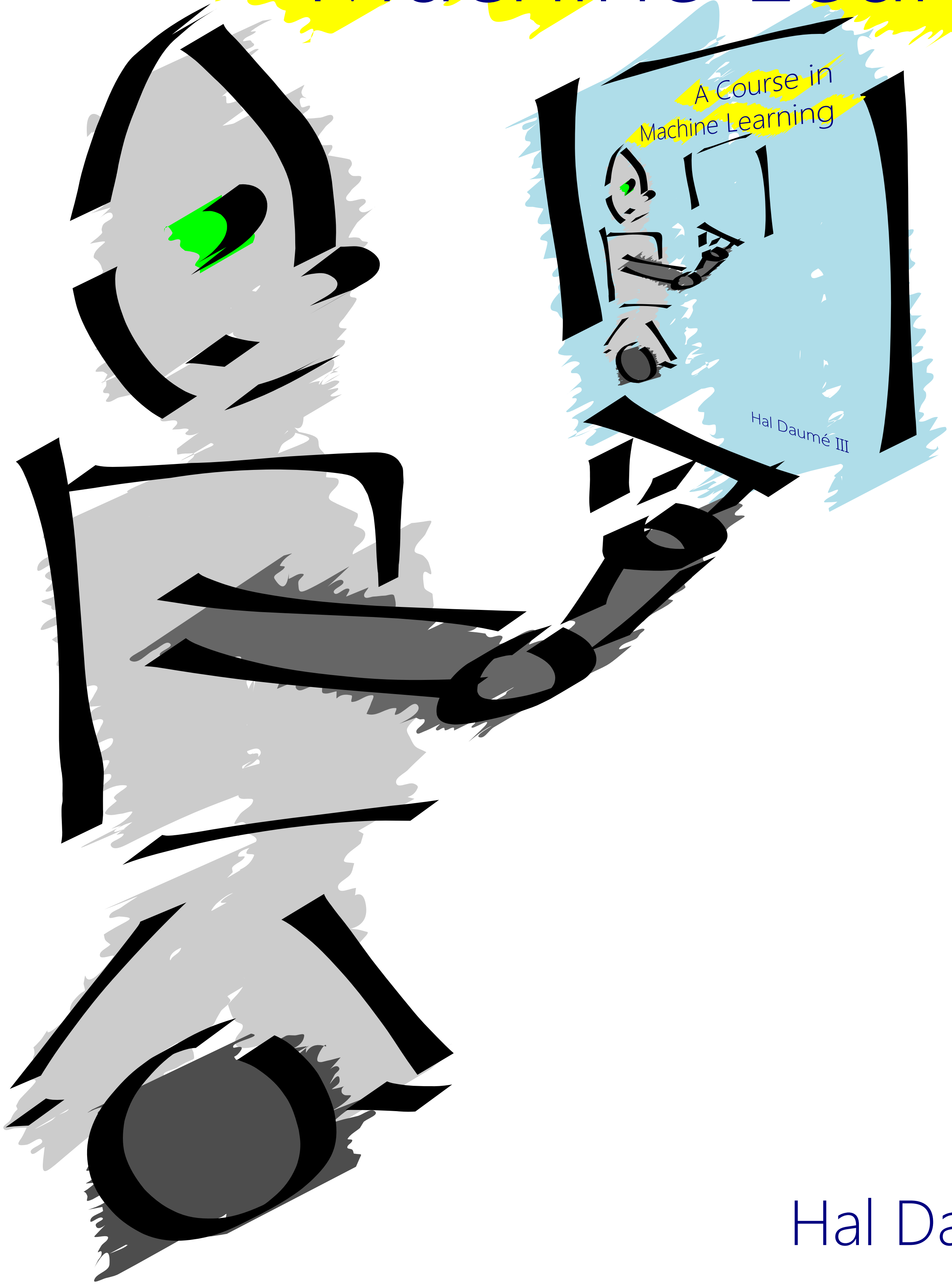
Figure 8.10: `nnet:unitsymbols`: picture of nnet with sigmoid/rbf units

? Consider an RBF network with one hidden unit per training point, centered at that point. What bad thing might happen if you use back-propagation to estimate the γs and $\boldsymbol{v}$ on this data if you're not careful? How could you be careful?

8.7 Exercises

Exercise 8.1. TODO...

A Course in Machine Learning



Hal Daumé III

12 | Efficient Learning

–

Learning Objectives:

- Understand and be able to implement stochastic gradient descent algorithms.
- Compare and contrast small versus large batch sizes in stochastic optimization.
- Derive subgradients for sparse regularizers.
- Implement feature hashing.

So far, our focus has been on *models* of learning and basic algorithms for those models. We have not placed much emphasis on how to learn *quickly*. The basic techniques you learned about so far are enough to get learning algorithms running on tens or hundreds of thousands of examples. But if you want to build an algorithm for web page ranking, you will need to deal with millions or billions of examples, in hundreds of thousands of dimensions. The basic approaches you have seen so far are insufficient to achieve such a massive scale.

In this chapter, you will learn some techniques for scaling learning algorithms. This are useful even when you do not have billions of training examples, because it's always nice to have a program that runs quickly. You will see techniques for speeding up both model training and model prediction. The focus in this chapter is on linear models (for simplicity), but most of what you will learn applies more generally.

Dependencies:

12.1 What Does it Mean to be Fast?

Everyone always wants fast algorithms. In the context of machine learning, this can mean many things. You might want fast training algorithms, or perhaps training algorithms that scale to very large data sets (for instance, ones that will not fit in main memory). You might want training algorithms that can be easily parallelized. Or, you might not care about training efficiency, since it is an offline process, and only care about how quickly your learned functions can make classification decisions.

It is important to separate out these desires. If you care about efficiency at training time, then what you are really asking for are more efficient learning algorithms. On the other hand, if you care about efficiency at test time, then you are asking for *models* that can be quickly evaluated.

One issue that is not covered in this chapter is parallel learning.

This is largely because it is currently not a well-understood area in machine learning. There are many aspects of parallelism that come into play, such as the speed of communication across the network, whether you have shared memory, etc. Right now, this the general, poor-man's approach to parallelization, is to employ ensembles.

12.2 *Stochastic Optimization*

During training of most learning algorithms, you consider the entire data set simultaneously. This is certainly true of gradient descent algorithms for regularized linear classifiers (recall Algorithm 6.4), in which you first compute a gradient over the entire training data (for simplicity, consider the unbiased case):

$$\boldsymbol{g} = \sum_n \nabla_{\boldsymbol{w}} \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + \lambda \boldsymbol{w} \tag{12.1}$$

where $\ell(y, \hat{y})$ is some loss function. Then you update the weights by $\boldsymbol{w} \leftarrow \boldsymbol{w} - \eta \boldsymbol{g}$. In this algorithm, in order to make a *single* update, you have to look at *every* training example.

When there are billions of training examples, it is a bit silly to look at every one before doing anything. Perhaps just on the basis of the first few examples, you can already start learning something!

Stochastic optimization involves thinking of your training data as a big distribution over examples. A draw from this distribution corresponds to picking some example (uniformly at random) from your data set. Viewed this way, the optimization problem becomes a **stochastic optimization** problem, because you are trying to optimize some function (say, a regularized linear classifier) over a probability distribution. You can derive this intepretation directly as follows:

$$\boldsymbol{w}^* = \arg\max_{\boldsymbol{w}} \sum_n \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + R(\boldsymbol{w}) \qquad \text{definition} \tag{12.2}$$

$$= \arg\max_{\boldsymbol{w}} \sum_n \left[\ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + \frac{1}{N} R(\boldsymbol{w}) \right] \qquad \text{move } R \text{ inside sum} \tag{12.3}$$

$$= \arg\max_{\boldsymbol{w}} \sum_n \left[\frac{1}{N} \ell(y_n, \boldsymbol{w} \cdot \boldsymbol{x}_n) + \frac{1}{N^2} R(\boldsymbol{w}) \right] \qquad \text{divide through by } N \tag{12.4}$$

$$= \arg\max_{\boldsymbol{w}} \mathbb{E}_{(y,\boldsymbol{x}) \sim D} \left[\ell(y, \boldsymbol{w} \cdot \boldsymbol{x}) + \frac{1}{N} R(\boldsymbol{w}) \right] \qquad \text{write as expectation} \tag{12.5}$$

$$\text{where } D \text{ is the training data distribution} \tag{12.6}$$

Given this framework, you have the following general form of an

Algorithm 33 StochasticGradientDescent($\mathcal{F}$, $\mathcal{D}$, S, K, η_1, ...)

```
1: $z^{(0)} \leftarrow \langle 0, 0, \dots, 0 \rangle$                // initialize variable we are optimizing
2: for $k = 1 \dots K$ do
3:     $D^{(k)} \leftarrow S$-many random data points from $\mathcal{D}$
4:     $g^{(k)} \leftarrow \nabla_z \mathcal{F}(D^{(k)})\big|_{z^{(k-1)}}$          // compute gradient on sample
5:     $z^{(k)} \leftarrow z^{(k-1)} - \eta^{(k)} g^{(k)}$              // take a step down the gradient
6: end for
7: return $z^{(K)}$
```

optimization problem:

$$\min_{z} \quad \mathbb{E}_{\zeta}[\mathcal{F}(z, \zeta)] \tag{12.7}$$

In the example, ζ denotes the random choice of examples over the dataset, z denotes the weight vector and $\mathcal{F}(w, \zeta)$ denotes the loss on that example *plus* a fraction of the regularizer.

Stochastic optimization problems are formally *harder* than regular (deterministic) optimization problems because you do not even get access to exact function values and gradients. The only access you have to the function $\mathcal{F}$ that you wish to optimize are noisy measurements, governed by the distribution over ζ. Despite this lack of information, you can still run a gradient-based algorithm, where you simply compute *local* gradients on a current sample of data.

More precisely, you can draw a data point at random from your data set. This is analogous to drawing a single value ζ from its distribution. You can compute the gradient of $\mathcal{F}$ just at that point. In this case of a 2-norm regularized linear model, this is simply $g = \nabla_w \ell(y, w \cdot x) + \frac{1}{N} w$, where (y, x) is the random point you selected. Given this *estimate* of the gradient (it's an estimate because it's based on a single random draw), you can take a small gradient step $w \leftarrow w - \eta g$.

This is the **stochastic gradient descent** algorithm (**SGD**). In practice, taking gradients with respect to a *single* data point might be too myopic. In such cases, it is useful to use a small **batch** of data. Here, you can draw 10 random examples from the training data and compute a small gradient (estimate) based on those examples: $g = \sum_{m=1}^{10} \nabla_w \ell(y_m, w \cdot x_m) + \frac{10}{N} w$, where you need to include 10 counts of the regularizer. Popular batch sizes are 1 (single points) and 10. The generic SGD algorithm is depicted in Algorithm 12.2, which takes K-many steps over batches of S-many examples.

In stochastic gradient descent, it is *imperative* to choose good step sizes. It is also very important that the steps get smaller over time at a reasonable slow rate. In particular, convergence can be guaranteed for learning rates of the form: $\eta^{(k)} = \frac{\eta_0}{\sqrt{k}}$, where η_0 is a fixed, initial step size, typically 0.01, 0.1 or 1 depending on how quickly you expect the algorithm to converge. Unfortunately, in comparisong to gradient descent, stochastic gradient is quite sensitive to the selection of a good learning rate.

There is one more practical issues related to the use of SGD as a learning algorithm: do you *really* select a random point (or subset of random points) at each step, or do you stream through the data in order. The answer is akin to the answer of the same question for the perceptron algorithm (Chapter 3). If you do not permute your data at all, very bad things can happen. If you *do* permute your data once and then do multiple passes over that same permutation, it will converge, but more slowly. In theory, you really should permute every iteration. If your data is small enough to fit in memory, this is not a big deal: you will only pay for cache misses. However, if your data is too large for memory and resides on a magnetic disk that has a slow seek time, randomly seeking to new data points for each example is prohibitivly slow, and you will likely need to forgo permuting the data. The speed hit in convergence speed will almost certainly be recovered by the speed gain in not having to seek on disk routinely. (Note that the story is very different for solid state disks, on which random accesses really are quite efficient.)

12.3 Sparse Regularization

For many learning algorithms, the test-time efficiency is governed by how many features are used for prediction. This is one reason decision trees tend to be among the fastest predictors: they only use a small number of features. Especially in cases where the actual *computation* of these features is expensive, cutting down on the number that are used at test time can yield huge gains in efficiency. Moreover, the amount of memory used to make predictions is also typically governed by the number of features. (Note: this is *not* true of kernel methods like support vector machines, in which the dominant cost is the number of support vectors.) Furthermore, you may simply *believe* that your learning problem can be solved with a very small number of features: this is a very reasonable form of inductive bias.

This is the idea behind sparse models, and in particular, sparse regularizers. One of the disadvantages of a 2-norm regularizer for linear models is that they tend to never produce weights that are *exactly* zero. They get close to zero, but never hit it. To understand why, as a weight w_d approaches zero, its gradient *also* approaches zero. Thus, even if the weight *should* be zero, it will essentially never get there because of the constantly shrinking gradient.

This suggests that an alternative regularizer is required to yield a sparse inductive bias. An ideal case would be the zero-norm regularizer, which simply counts the number of non-zero values in a vector: $||\boldsymbol{w}||_0 = \sum_d [w_d \neq 0]$. If you could minimize this regularizer, you would be explicitly minimizing the number of non-zero features. Unfortunately, not only is the zero-norm non-convex, it's also discrete. Optimizing it is NP-hard.

A reasonable middle-ground is the one-norm: $||\boldsymbol{w}||_1 = \sum_d |w_d|$. It is indeed convex: in fact, it is the tighest ℓ_p norm that *is* convex. Moreover, its gradients do not go to zero as in the two-norm. Just as hinge-loss is the tightest convex upper bound on zero-one error, the one-norm is the tighest convex upper bound on the zero-norm.

At this point, you should be content. You can take your subgradient optimizer for arbitrary functions and plug in the one-norm as a regularizer. The one-norm is surely non-differentiable at $w_d = 0$, but you can simply choose any value in the range $[-1, +1]$ as a subgradient at that point. (You should choose zero.)

Unfortunately, this does not quite work the way you might expect. The issue is that the gradient might "overstep" zero and you will never end up with a solution that is particularly sparse. For example, at the end of one gradient step, you might have $w_3 = 0.6$. Your gradient might have $g_6 = 0.8$ and your gradient step (assuming $\eta = 1$) will update so that the new $w_3 = -0.2$. In the subsequent iteration, you might have $g_6 = -0.3$ and step to $w_3 = 0.1$.

This observation leads to the idea of **trucated gradients**. The idea is simple: if you have a gradient that would step you over $w_d = 0$, then just set $w_d = 0$. In the easy case when the learning rate is 1, this means that if the *sign* of $w_d - g_d$ is different than the sign of w_d then you truncate the gradient step and simply set $w_d = 0$. In other words, g_d should never be *larger* than w_d Once you incorporate learning rates, you can express this as:

$$g_d \leftarrow \begin{cases} g_d & \text{if } w_d > 0 \text{ and } g_d \leq \frac{1}{\eta^{(k)}} w_d \\ g_d & \text{if } w_d < 0 \text{ and } g_d \geq \frac{1}{\eta^{(k)}} w_d \\ 0 & \text{otherwise} \end{cases} \tag{12.8}$$

This works quite well in the case of subgradient descent. It works somewhat less well in the case of *stochastic* subgradient descent. The problem that arises in the stochastic case is that wherever you choose to stop optimizing, you will have just touched a single example (or small batch of examples), which will increase the weights for a lot of features, before the regularizer "has time" to shrink them back down to zero. You will still end up with somewhat sparse solutions, but not as sparse as they could be. There are algorithms for dealing with this situation, but they all have a heuristic flavor to them and are beyond the scope of this book.

12.4 Feature Hashing

As much as speed is a bottleneck in prediction, so often is memory usage. If you have a very large number of features, the amount of memory that it takes to store weights for all of them can become prohibitive, especially if you wish to run your algorithm on small devices. Feature hashing is an incredibly simple technique for reducing the memory footprint of linear models, with very small sacrifices in accuracy.

The basic idea is to replace all of your features with hashed versions of those features, thus reducing your space from D-many feature weights to P-many feature weights, where P is the range of the hash function. You can actually think of hashing as a (randomized) feature mapping $\phi : \mathbb{R}^D \to \mathbb{R}^P$, for some $P \ll D$. The idea is as follows. First, you choose a hash function h whose domain is $[D] = \{1, 2, \dots, D\}$ and whose range is $[P]$. Then, when you receive a feature vector $\boldsymbol{x} \in \mathbb{R}^D$, you map it to a shorter feature vector $\hat{\boldsymbol{x}} \in \mathbb{R}^P$. Algorithmically, you can think of this mapping as follows:

1. Initialize $\hat{\boldsymbol{x}} = \langle 0, 0, \dots, 0 \rangle$
2. For each $d = 1 \dots D$:
   (a) Hash d to position $p = h(d)$
   (b) Update the pth position by adding x_d: $\hat{x}_p \leftarrow \hat{\boldsymbol{x}}_p + x_d$
3. Return $\hat{\boldsymbol{x}}$

Mathematically, the mapping looks like:

$$\phi(\boldsymbol{x})_p = \sum_d [h(d) = p] x_d \quad = \sum_{d \in h^{-1}(p)} x_d \tag{12.9}$$

where $h^{-1}(p) = \{d : h(d) = p\}$.

In the (unrealistic) case where $P = D$ and h simply encodes a permutation, then this mapping does not change the learning problem at all. All it does is rename all of the features. In practice, $P \ll D$ and there will be collisions. In this context, a collision means that two features, which are really different, end up looking the same to the learning algorithm. For instance, "is it sunny today?" and "did my favorite sports team win last night?" might get mapped to the same location after hashing. The hope is that the learning algorithm is sufficiently robust to noise that it can handle this case well.

Consider the kernel defined by this hash mapping. Namely:

$$K^{(\text{hash})}(\boldsymbol{x}, \boldsymbol{z}) = \phi(\boldsymbol{x}) \cdot \phi(\boldsymbol{z}) \tag{12.10}$$

$$= \sum_p \left(\sum_d [h(d) = p] x_d \right) \left(\sum_d [h(d) = p] z_d \right) \tag{12.11}$$

$$= \sum_p \sum_{d,e} [h(d) = p][h(e) = p] x_d z_e \tag{12.12}$$

$$= \sum_d \sum_{e \in h^{-1}(h(d))} x_d z_e \tag{12.13}$$

$$= \boldsymbol{x} \cdot \boldsymbol{z} + \sum_d \sum_{\substack{e \neq d, \\ e \in h^{-1}(h(d))}} x_d z_e \tag{12.14}$$

This **hash kernel** has the form of a linear kernel plus a small number of quadratic terms. The particular quadratic terms are exactly those given by collisions of the hash function.

There are two things to notice about this. The first is that collisions might not actually be bad things! In a sense, they're giving you a little extra representational power. In particular, if the hash function happens to select out feature pairs that benefit from being paired, then you now have a better representation. The second is that even if this doesn't happen, the quadratic term in the kernel has only a small effect on the overall prediction. In particular, if you assume that your hash function is pairwise independent (a common assumption of hash functions), then the *expected value* of this quadratic term is zero, and its variance decreases at a rate of $\mathcal{O}(P^{-2})$. In other words, if you choose $P \approx 100$, then the variance is on the order of 0.0001.

12.5 *Exercises*

Exercise 12.1. TODO...