CHAPTER 20

The Elements of Classification

A *classifier* is a procedure that accepts a set of features and produces a class label for them. Classifiers are immensely useful, and find wide application, because many problems are naturally classification problems. For example, if you wish to determine whether to place an advert on a web-page or not, you would use a classifier (i.e. look at the page, and say yes or no according to some rule). This is a two class classifier, but in many cases it is natural to have more classes – a *multi-class classifier*. You can think of sorting laundry as applying a multi-class classifier. You can think of doctors as complex multi-class classifiers: a doctor accepts a set of features (your complaints, answers to questions, and so on) and then produces a response which we can describe as a class. The grading procedure for any class is a multi-class classifier: it accepts a set of features — performance in tests, homeworks, and so on — and produces a class label (the letter grade).

20.1 CLASSIFICATION: GENERAL IDEAS

A classifier is usually trained by obtaining a set of labelled training examples and then searching for a procedure that optimizes some cost function which is evaluated on the training data. Performance on training data doesn't really matter. What matters is performance on run-time data, which may be extremely hard to evaluate because one often does not know the correct answer for that data. For example, you wish to classify credit-card transactions as safe or fraudulent. You could obtain a set of transactions with true labels, and train with those. But what you care about is new transactions, where it would be very difficult to know whether the classifier's answers are right. To be able to do anything at all, the set of labelled examples must be representative of future examples in some strong way.

Remember this: *A classifier is a procedure that accepts a set of features and produces a label. Classifiers are trained on labelled examples, but the goal is to get a classifier that performs well on data which is not seen at the time of training. Training a classifier requires labelled data that is representative of future data*

20.1.1 The Error Rate, and Other Summaries of Performance

You can summarize the performance of any particular classifier using the *error* or *total error rate* (the percentage of classification attempts that gave the wrong answer) and the *accuracy* (the percentage of classification attempts that give the

right answer). Recall, you want to know what the classifier does on run-time data, so these numbers need to be computed on data *that wasn't used to train the classifier* and that is "like" the data you will use it on.

For most practical cases, even the best choice of classifier will make mistakes. For example, an alien tries to classify humans into male and female, using only height as a feature. Whatever the alien's classifier does with that feature, it will make mistakes. This is because the classifier must choose, for each value of height, whether to label the humans with that height male or female. But for the vast majority of heights, there are some males and some females with that height, and so the alien's classifier must make some mistakes. The minimum expected error rate obtained with the best possible classifier applied to a particular problem is known as the *Bayes risk* for that problem. In most cases, it is hard to know what the Bayes risk is.

The error rate of a classifier is not that meaningful on its own, because you don't usually know the Bayes risk for a problem. It is more helpful to compare a particular classifier with some natural alternatives, sometimes called *baselines*. The choice of baseline for a particular problem is almost always a matter of application logic. The simplest general baseline is a know-nothing strategy. Imagine classifying the data without using the feature vector at all — how well does this strategy do? If each of the C classes occurs with the same frequency, then it's enough to label the data by choosing a label uniformly and at random, and the error rate for this strategy is $1 - 1/C$. If one class is more common than the others, the lowest error rate is obtained by labelling everything with that class. This comparison is often known as *comparing to chance*.

If the data has only two labels the highest possible error rate is 50% — if you have a classifier with a higher error rate, you can improve it by switching the outputs. If one class is much more common than the other, training becomes more complicated because the best strategy – labelling everything with the common class – becomes hard to beat.

Remember this: *Classifier performance is summarized by either the total error rate or the accuracy, computed on data that (a) wasn't used to train the classifier and (b) is reasonably representative of what you will use in practice. You will very seldom know what the best possible performance for a classifier on a problem is. You should always compare performance to baselines. Chance is one baseline that can be surprisingly strong.*

The error rate is a fairly crude summary of the classifier's behavior. For a two-class classifier and a 0-1 loss function, one can report the *false positive rate* (the percentage of negative test data that was classified positive) and the *false negative rate* (the percentage of positive test data that was classified negative). Note that it is important to provide both, because a classifier with a low false positive rate tends to have a high false negative rate, and vice versa. As a result, you should be suspicious of reports that give one number but not the other. Alternative numbers

that are reported sometimes include the *sensitivity* (the percentage of true positives that are classified positive) and the *specificity* (the percentage of true negatives that are classified negative).

The false positive and false negative rates of a two-class classifier can be generalized to evaluate a multi-class classifier, yielding the *class confusion matrix*. This is a table of cells, where the i, j'th cell contains the count of cases where the true label was i and the predicted label was j (some people show the fraction of cases rather than the count). The first thing to look at in a table like this is the diagonal; if the largest values appear there, then the classifier is working well.

> **Remember this:** *When more detailed evaluation of a classifier is required, look at the false positive rate and the false negative rate. Always look at both, because doing well at one number tends to result in doing poorly on the other. The class confusion matrix summarizes errors for multiclass classification.*

20.1.2 Overfitting and Cross-Validation

Choosing and evaluating a classifier takes some care. The goal is to get a classifier that works well on future data *for which we might never know the true label*, using a training set of labelled examples. This isn't necessarily easy. For example, think about the (silly) classifier that takes any data point and, if it is the same as a point in the training set, emits the class of that point; otherwise, it chooses randomly between the classes.

The *training error* of a classifier is the error rate on examples used to train the classifier. In contrast, the *test error* is error on examples not used to train the classifier. Classifiers that have small training error might not have small test error, because the classification procedure is chosen to do well on the training data. This effect is sometimes called *overfitting*. Other names include *selection bias*, because the classifier has been selected to do well on the training data, and *generalizing badly*, because the classifier must generalize from the training data to the test data.

The effect occurs because the classifier has been chosen to perform well *on the training dataset*. An efficient training procedure is quite likely to find special properties of the training dataset that aren't representative of the test dataset, because the training dataset is not the same as the test dataset. The training dataset is typically a sample of all the data one might like to have classified, and so is quite likely a lot smaller than the test dataset. Because it is a sample, it may have quirks that don't appear in the test dataset. One consequence of overfitting is that classifiers should always be evaluated on data that was not used in training.

Now assume that you want to estimate the error rate of the classifier on test data. You cannot estimate the error rate of the classifier using data that was used to train the classifier, because the classifier has been trained to do well on that data, which will mean the error rate estimate will be too low. An alternative is to separate out some training data to form a *validation set* (confusingly, this is

sometimes called a test set), then train the classifier on the rest of the data, and evaluate on the validation set. The error estimate on the validation set is the value of a random variable, because the validation set is a sample of all possible data you might classify. But this error estimate is *unbiased*, meaning that the expected value of the error estimate is the true value of the error.

However, separating out some training data presents the difficulty that the classifier will not be the best possible, because we left out some training data when we trained it. This issue can become a significant nuisance when we are trying to tell which of a set of classifiers to use — did the classifier perform poorly on validation data because it is not suited to the problem representation or because it was trained on too little data?

You can resolve this problem with *cross-validation*, which involves repeatedly: splitting data into training and validation sets uniformly and at random, training a classifier on the training set, evaluating it on the validation set, and then averaging the error over all splits. Each different split is usually called a *fold*. This procedure yields an estimate of the likely future performance of a classifier, at the expense of substantial computation. A common form of this algorithm uses a single data item to form a validation set. This is known as *leave-one-out cross-validation*.

Remember this: *Classifiers usually perform better on training data than on test data, because the classifier was chosen to do well on the training data. This effect is known as overfitting. To get an accurate estimate of future performance, classifiers should always be evaluated on data that was not used in training.*

20.2 A SIMPLE BINARY CLASSIFIER

Here is a simple problem whose solution is useful. You wish to tell whether an image is the output of a denoiser, or just an image. Intuition should tell you that this will work. For example, denoising by Gaussian smoothing will produce an image that is slightly blurred, and it should be possible to tell whether an image is a little blurred.

This section assumes that each image is represented by a known feature vector of fixed dimension (Section 20.3 treats how to obtain this feature vector). The classifier will accept this feature vector, then produce a number. Ideally, that number is positive for for any image that comes out of the denoiser, and negative for any that is a real image patch.

20.2.1 From Features to Label with a Linear Classifier

Assume you have a feature vector $\mathbf{x}$ that describes an image well. You must map this feature vector to a *label* which identifies the class of the image. In the current case, the label is either "real" or "denoised", but much richer alternatives will be important (Chapter **??**). A straightforward choice is a *linear classifier*, which

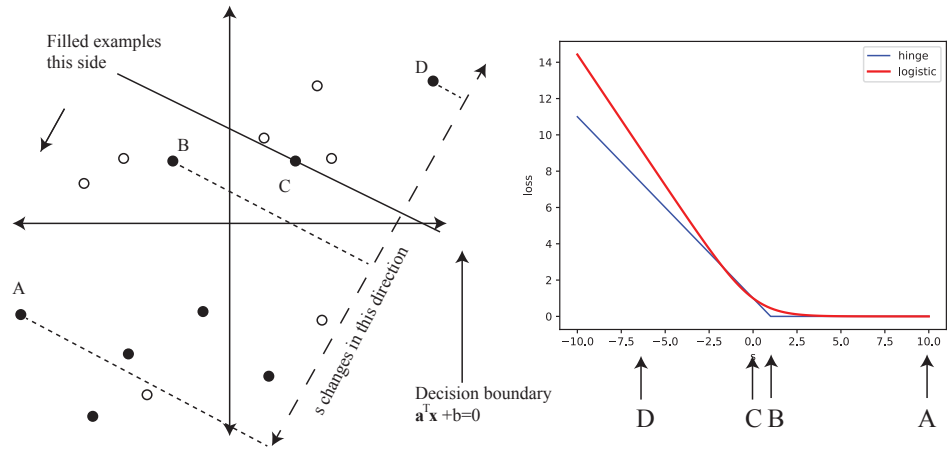

FIGURE 20.1: **Left:** *A visualization of a linear classifier in a 2D feature space (so $f = 2$) to illustrate the constraints on a classification loss. The example labelled D should have large loss, because it is on the wrong side of the boundary and far away from the boundary. The example labelled C should have a medium loss. It is on the boundary, but should be some way to the right side. This is because there are likely future examples close to it, and some of those might be on the wrong side of the boundary. The example labelled B should have a zero loss, because it is far enough on the right side of the boundary.* **Right:** *Plots of the hinge and cross entropy loss for filled examples, keyed to the example labels, to show how these losses meet the constraints.*

maps $\mathbf{x}$ to $u(\mathbf{x}; \mathbf{a}, b) = (\mathbf{a}^T\mathbf{x} + b)$, then uses the sign of that value to classify. Equivalently, a linear classifier constructs a hyperplane in the feature space. Data items that map to one side of the hyperplane are real and data items that map to the other side are denoiser outputs. The parameters $\mathbf{a}$ and b are chosen to get the best performance (many more details below). You might object that this mapping is too simple to achieve what is wanted. But the the feature vector is a high dimensional representation of the image, so there is a good chance of finding a linear classifier that separates the two. It will turn out that the feature vector is the product of a learned encoder, meaning *you can adjust the encoder to get the feature vector that works best with a linear classifier*. All this is easily extended to deal with more than two classes (Section 21.1).

It is sometimes useful to think in terms of the *decision boundary* of a classifier. This is the boundary (or set of boundaries) between volumes in feature space that have constant label. The decision boundary of a linear classifier is a hyperplane in feature space.

Remember this: *A two-class linear classifier maps a feature vector* $\mathbf{x}$ *to one of two classes, using*

$$sign(\mathbf{a}^T\mathbf{x} + b)$$

where $\mathbf{a}$ *and* b *are parameters of the classifier which are chosen to get strong performance on training data. Linear classifiers are much more powerful than you might expect at first glance. The decision boundary of a linear classifier is a hyperplane in feature space.*

20.2.2 Logistic Regression

The next step is to choose the parameters of the linear classifier to get good behavior from the classifier. The recipe used in Chapter 17 applies: construct a loss, then use some optimization procedure to minimize the loss. Notice that you can't use training error to adjust the parameters $\mathbf{a}$ and b. Gradient descent on error rate won't work, because the gradient is zero almost everywhere. Instead, some approximation to the error rate is required.

One natural approximation is to interpret $u(\mathbf{x}; \mathbf{a}, b)$ in terms of a probability. Use the model

$$u(\mathbf{x}; \mathbf{a}, b) = \log \left[\frac{P(\text{denoise}|\mathbf{x})}{P(\text{real}|\mathbf{x})} \right].$$

This means a data item with positive u is likely to be from the denoiser, and more likely to be from the denoiser if $|u|$ is larger. A data item with a negative u is likely to be real, and more likely so if $|u|$ is larger. In particular

$$P(\text{denoise}|\mathbf{x}) = \frac{e^u}{1+e^u} \text{ and } P(\text{real}|\mathbf{x}) = \frac{1}{1+e^u}.$$

Call this distribution the *predictive distribution* for the i'th example, and write $P(\cdot; u_i)$. Now write $\mathcal{S}$ for the set of N examples, where each example has the form $(\mathbf{x}_i, y_i)$, and

$$y_i = \begin{cases} 1 & \text{if } i\text{'th example is real} \\ -1 & \text{otherwise} \end{cases}$$

Then the log-likelihood of the dataset under this model is

$$\sum_{i \in \mathcal{S}} \left[u_i \left(\frac{1-y_i}{2} \right) - \log\left(1 + e^{u_i}\right) \right]$$

(you should check this; **exercises**). It would be natural to choose $\mathbf{a}$, b to maximize this likelihood, a procedure known as *logistic regression*. The usual procedure minimizes the negative log-likelihood (also logistic regression). You should think of this as minimizing a loss, and good practice regards a loss as a mean over a set of

data items, so write

$$\mathcal{L}_{lr}(\mathbf{a}, b) = \frac{-1}{N} \sum_{i \in \mathcal{S}} \left[u_i \left(\frac{1 - y_i}{2} \right) - \log \left(1 + e^{u_i} \right) \right].$$

> **Remember this:** *Logistic regression chooses the parameters of a two-class linear classifier by minimizing the negative log-likelihood, which takes an easy form.*

20.2.3 The Cross Entropy Loss

The *cross-entropy* between a discrete distribution p – a reference distribution – and another discrete distribution on the same space q is

$$H_x(p, q) = -\mathbb{E}[p] [\log q] = -\sum_u p_u \log q_u$$

where the sum is over all elements with non-zero terms in p and q. For fixed p and varying q, the largest value could be arbitrarily large (**exercises**), but the smallest value is $H(p) = -\sum_u p_u \log p_u$ (**exercises**). You could use

$$\mathcal{L}_{xe} = \frac{1}{N} H_x(p, q(\mathbf{a}, b))$$

as a loss.

Now interpret the label for the i'th data item as a model probability distribution, by writing $p_i(\text{real}) = (1 + y_i)/2$ and $p_i(\text{denoise}) = 1 - p_i(\text{real}) = (1 - y_i)/2$. One of these is 1 and the other 0 for each data item, and there is a different distribution for each data item. Write m_i for the i'th such example distribution and $P(\cdot; u_i)$ for the distribution predicted by the classifier for the i'th item. Notice that the logistic loss is constructed out of cross-entropy terms, so

$$\begin{aligned}
-N\mathcal{L}_{lr} &= \sum_{i \in \mathcal{S}} \left[u_i \left(\frac{1 - y_i}{2} \right) - \log \left(1 + e^{u_i} \right) \right] \\
&= \sum_{i \in \mathcal{S}} \left[\left(\frac{1 - y_i}{2} \right) [u_i - \log (1 + e^{u_i})] + \left(\frac{1 + y_i}{2} \right) [-\log (1 + e^{u_i})] \right] \\
&= \sum_{i \in \mathcal{S}} [p_i(\text{real}) \log P(\text{real}|u_i) + p_i(\text{denoise}) \log P(\text{denoise}|u_i)] \\
&= -\sum_{i \in \mathcal{S}} H(m_i, P(\cdot; u_i)) \\
&= N\mathcal{L}_{xe}.
\end{aligned}$$

This means that you can interpret $\mathcal{L}_{lr}$ as a comparison between the predicted distribution and the model distribution for each data item. This is convenient

when you are not certain of the label, but have some probability that you believe (**exercises**).

> **Remember this:** *Minimizing cross-entropy loss is equivalent to minimizing the negative log-likelihood. However, cross-entropy loss is convenient when you are not certain of the label, but have some probability (that you believe) that a data item has a particular label.*

20.2.4 The Logistic Loss

Write $s_i = y_i u_i = y_i(\mathbf{a}^T\mathbf{x} + b)$. The *logistic loss* function is given by

$$\mathcal{L}_{\text{logistic}}(s) = \frac{1}{\log 2}\left[\log\left(1 + e^{-s}\right)\right]$$

Then, by recalling that $\log\left(1 + e^{f}\right) = f + \log\left(1 + e^{-f}\right)$, you can show that the log-likelihood for logistic regression is

$$\mathcal{L}_{lr} = \frac{(\log 2)}{N}\sum_{i \in \mathcal{S}} \mathcal{L}_{\text{logistic}}(s_i)$$

(though the $\log 2$ factor is often ignored).

> **Remember this:** *Derive a loss equivalent to the negative log-likelihood by applying the logistic loss to $s_i = y_i u_i$.*

20.2.5 The Hinge Loss

The logistic loss has a helpful geometric interpretation in terms of the hyperplane $\mathbf{a}^T\mathbf{x} + b = 0$. If the i'th example is correctly classified and far from the hyperplane, s_i is large and positive, and so $\mathcal{L}_{\text{logistic}}(s_i)$ is very close to zero. As s_i gets closer to zero (and so the example gets closer to the hyperplane on the right side), the logistic loss grows. If s_i is a lot smaller than zero (and so the example is far from the hyperplane and on the wrong side), the loss grows close to linearly in s_i There are other loss functions that have this behavior. The *hinge loss* function

$$\mathcal{L}_{\text{hinge}}(s) = \mathsf{max}\left(1 - s, 0\right)$$

has this behavior as well. Recall $s_i = y_i(\mathbf{a}^T\mathbf{x}_i + b)$. The hinge loss for a dataset is

$$\frac{1}{N}\sum_{i \in \mathcal{S}} \mathcal{L}_{\text{hinge}}(s_i).$$

If the example is correctly classified and far from the hyperplane, s is larger than 1, and so the hinge loss is zero. If the example is correctly classified and close to the hyperplane, s is less than 1, and so the hinge loss is positive and gets bigger as the example gets closer to the hyperplane. If the example is incorrectly classified, the loss is positive, and the loss grows as the example gets further from the hyperplane (Figure 20.1).

For both the hinge and the logistic loss, there is some cost to having an example close to the hyperplane even if it is on the right side. This effect helps ensure that the classifier performs well on test examples. You should expect future test examples to occur near to training examples. If (say) a training example is on the right side of the hyperplane, but is close to it, there is some possibility that some other, future example that is near the training example might be on the wrong side of the hyperplane. This kind of future error is more likely if the training example is closer to the hyperplane. This means it is a good idea for the loss to have a *margin*. A training example that is on the right side, but close to the hyperplane, should have loss greater than zero, and the loss should get bigger for examples that are closer to the hyperplane.

Remember this: *The hinge loss is very like the logistic loss. Each has the important features that: correctly classifying an item with a large positive value of $s_i = y_i u_i$ is free; correctly classifying the item with a small positive value of s_i has a cost; and incorrectly classifying the item (so s_i negative) is expensive, and more so as s_i gets more negative. These constraints are natural. Ensuring that there is some cost to having a training item close to the decision boundary, even if it is on the right side, encourages better accuracy on test data.*

20.3 BUILDING A CLASSIFIER

There are a variety of ways to get the feature representation $\mathbf{x}$ used to classify. The data might just come with a good feature (unusual, but not impossible). More usually, you will want to construct the feature. There was a time when it appeared to make sense to do this largely or partially using your judgement. It is now clear that learning the feature is a good idea.

There are two strategies. In the first, you obtain a pretrained encoder and use this to produce features. There are now a number of published encoders that are very strong (Section 36.3). There is strong evidence that encodings from these encoders support a wide range of image classification activities. However, training an encoder that is strong enough to be used for a wide range of tasks is currently very much not for the faint hearted. Doing so requires immense amounts of training data and a variety of schemes. There are sketches in Chapter 36.3, but you are unlikely to be able to go from that information to such an encoder. If you don't want to use a published general purpose encoder, you could train an autoencoder,

then recover a feature representation from that. The classifier that results will not work very well.

The alternative is to build a classifier that accepts an image and predicts a value. If that value is positive, the classifier has labelled the patch a denoised patch; if negative, a real patch. This is equivalent to training the encoder and the classifier together, and is the usual scheme for small classifiers. Mostly, you know how to do this already. It is straightforward to repurpose the tools of Chapter **??** to do so. The classifier consists of a learned encoder that accepts an image and produces an $f \times 1$ dimensional vector which is passed to a linear classifier that produces a number. Sections 20.2.2, 20.2.3 and 20.2.5 offer options for scoring the numbers produced from a training set. The losses are (mostly) differentiable functions of the parameters $\mathbf{a}$ and b, so the machinery of Sections 17.3.4, 17.3.1 (and variants in Section 18.4.1) apply. The main open question is good ways to turn a block of encoded features into a vector.

20.3.1 Pooling

Chapters 17 and 18 showed procedures to produce a learned image representation: Apply a sequence of layers to an image, typically, convolutional, then ReLU, then convolutional, then ReLU, and so on. There could be stride in these layers, so that the block of features gets spatially smaller as it moves up the encoder.

For the result to be a vector, it must be $f \times 1 \times 1$. This could be achieved with stride alone, but an alternative is a *pooling layer* – a layer that reduces the spatial extent of the data block by forming summaries of local windows. Windows may overlap (depending on the API), but often don't. Quite usual is halve the spatial dimension of the image by pooling over non-overlapping 2×2 windows, so mapping from $f \times 2a \times 2b$ to $f \times a \times b$. In *average pooling*, the summary is the mean of the elements in the window in each feature layer, and in *max-pooling*, the summary is the maximum of the elements in the window in each feature layer. These pooling layers have no learnable parameters (unlike, say, a convolutional layer with stride 2). Pooling layers differ by how they react to unusual (outlying) responses from feature detectors. Average pooling will tend to suppress them, whereas max-pooling will tend to emphasize them; there is some evidence that emphasizing them, and so max-pooling, is better on the whole for some classification purposes.

The layers, stride, padding and pooling are arranged so that the $c \times d \times d$ image results in a $g \times s \times s$ block. It is straightforward to turn this into a $g \times 1 \times 1$ block by average pooling over the two spatial dimensions.

20.3.2 Fully Connected Layers

You could regard the $g \times 1 \times 1$ block as a vector (in some APIs, you need to reshape, but this is housekeeping) and simply pass it to a linear classifier. Alternatively, you could transform this vector with a *fully connected layer*, which maps a vector $\mathbf{u}$ to a vector $\mathcal{C}\mathbf{u} + \mathbf{d}$, where the parameters $\mathcal{C}$ and $\mathbf{d}$ are learned, and $\mathcal{C}$ does not need to be square.

Notice that applying a linear classifier $\mathbf{a}^T\mathbf{x} + b$ to the output of a fully connected layer is not particularly interesting, because the result is $\left(\mathcal{C}^T\mathbf{a}\right)^T \mathbf{u} + \left(\mathbf{a}^T\mathbf{d} + b\right)$, which is just a different linear classifier. Similarly, applying a fully con-

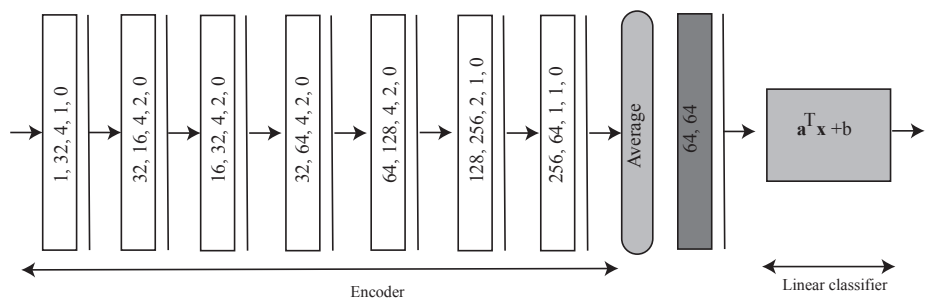

FIGURE 20.2: *A (very simple) classifier built out of convolutional layers, a pooling layer, a fully connected layer, and a linear classifier. Each light block represents a convolutional layer (arguments are, in order: input dimension, output dimension, kernel size, stride, padding); vertical lines are ReLU layers; the gray block with rounded corners pools over all spatial dimensions to produce a vector; and the dark gray block is a fully connected layer.*

nected layer to another fully connected layer directly is not interesting. Instead, each fully connected layer is followed by a ReLU.

It is usual to take the $g \times 1 \times 1$ block, turn it into a vector if your API wants that, then pass it through a fully connected layer and then a ReLU layer at least once and possibly multiple times before applying a linear classifier. Experience teaches that it is helpful to pass high dimensional features to a linear classifier. This creates a minor tension, because big fully connected layers have a lot of parameters in them and can create issues with both inference and learning speed. Further, the classifier will need to be evaluated on images of fixed size (**exercises**), which can be a nuisance.

20.3.3 Training a Classifier

The encoder architecture produces an $f \times 1 \times 1$ block, and the classifier dots that vector with a parameter vector $\mathbf{a}$, adds b, and reports the result. This process is another layer, like the convolutional layers. Fold $\mathbf{a}$ and b into the parameter vector θ. The result is a function that accepts an example image $\mathcal{I}_i$ and produces a number. Write $F(\mathcal{I}_i, \theta)$ for the number that comes out of the classifier.

Choose one of the logistic or hinge losses, and write $\mathcal{C}$ for your chosen loss. Then the loss of applying the classifier to all training examples is

$$\sum_{i \in \text{train}} \mathcal{C}(F(\mathcal{I}_i, \theta), y_i)$$

and stochastic gradient descent can be applied to choose θ as in Section 17.3.1.

20.3.4 Worked Example

Figure 20.2 shows the architecture of a very simple classifier I used to classify real vs. denoised. I trained this classifier using a cross-entropy loss; the optimizer was

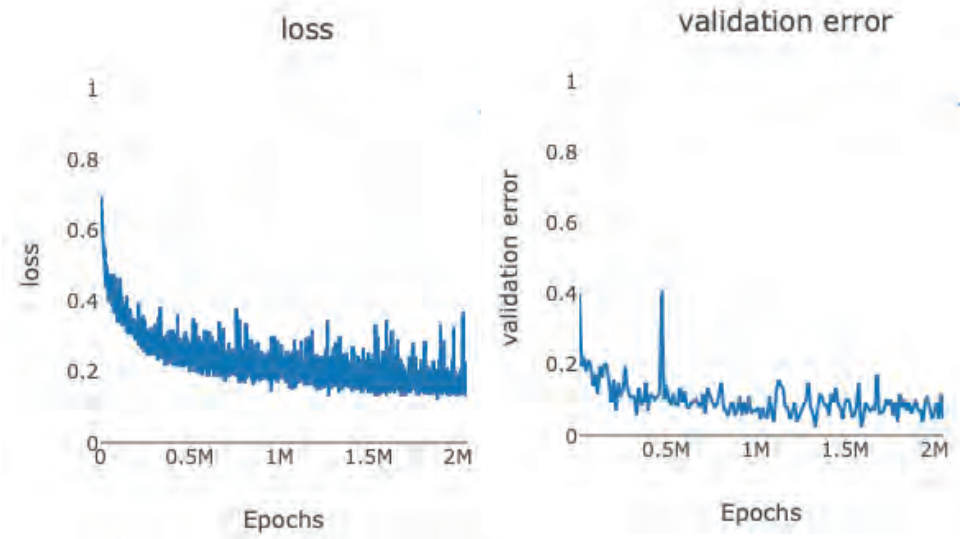

FIGURE 20.3: *A plot of training loss (***left***) and validation error rate (***right***) for the classifier of Figure 20.2, plotted against the number of training images the classifier has seen. These are fairly characteristic of a simple classifier. The loss mostly goes down, but there is some noise, particularly in the early stages of training (where a randomly selected batch may show the classifier effects it hasn't seen before). The validation error mostly goes down, then slows. The validation error is somewhat noisy, because it is measured on batches of 128 images rather than the whole validation set, so there is some chance of an odd batch. Eventually, the validation error rate must stall (it can't go below 0!) but – in this case – is small. There is a very good chance of telling accurately whether an image has been through the autoencoder described in the text or not using a simple classifier – the error rate averaged over the whole validation set is* 0.06 *(so about one in 20 images will be misclassified).*

Adam (Section **??**); and I used batches of 128 images. I used 100, 000 images from the ImageNet training set (Section **??**), which I mapped to gray level images at 128×128 resolution. I obtained denoised images by applying the noise of Section **??** to training images, then denoising them with the autoencoder from that section (the one that uses skip connections). I used 20, 000 images from that set as test examples, and constructed denoised test examples as in training examples. This classifier is about as simple as it could be, and still quite easily tells test denoise images from test real images. The behavior of the classifier is summarised in Figure 20.3. Various modifications should lead to an improved classifier (**exercises**). There is a very good chance of telling accurately whether an image has been through the autoencoder described in the text or not using a simple classifier – the error rate averaged over the whole validation set is 0.06 (so about one in 20 images will be misclassified).

20.3.5 Some Routine Classification Tricks

Data augmentation often results in significant improvements. Section **??** suggested that an image that results from augmentation should be "like" a real image. Experience suggests that many augmentations that produce rather unnatural images still improve classification. Most classifiers accept images of fixed size (the ones that appear not to usually just resize the input image). Augmentations that are often useful include: **cropping** and then resizing, including cropping rectangles with an aspect ratio that isn't that of the input image (so training examples look squashed or stretched); and **color adjustments**, many of which result in unnatural looking images. Most APIs will offer extensive augmentation options that are set up to be included in the batch loading procedure (and are often highly optimized). It is often useful to regularize a classifier. Use the procedures of Section **??**; most APIs will do this for you.

The weights of the classifier you train are the values of a random variable (batches, etc. are chosen randomly). This means the classifier may not be the very best possible such. Training more than one, then *ensembling* the result, often produces an easy improvement. You can ensemble by voting, which gets clunky if there are many classes, or by simply adding scores. The difficulty with ensembling is you have to train multiple classifiers. If you don't want to train more than one classifier, you can still get small improvements by applying your trained classifier to multiple crops of the input image, then combine the results. In either case, at inference time, more computation is required.

20.4 YOU SHOULD

20.4.1 remember these facts:

Classifier: definition 380
Classifier performance is summarised by accuracy or error rate . . . 381
Look at false positive rate and false negative rate together 382
Do not evaluate a classifier on training data 383
A two-class linear classifier 385
An expression for log-likelihood of data under a linear classifier . . . 386
Cross-entropy loss is negative log-likelihood 387
Logistic loss yields negative log-likelihood 387
Hinge loss and logistic loss are similar and meet important constraints388

20.4.2 remember these procedures:

20.4.3 exploit these resources:

20.4.4 be able to:

- Explain how a linear classifier works.
- Construct a simple two-class classifier out of an encoder, assorted pooling and fully connected layers, and a linear classifier.
- Apply cross-entropy loss or the hinge loss to train that classifier.
- Apply various tricks to improve performance.

EXERCISES

QUICK CHECKS

20.1. Check that

$$u(\mathbf{x}; \mathbf{a}, b) = \log \left[\frac{P(\text{denoise}|\mathbf{x})}{P(\text{real}|\mathbf{x})} \right]$$

means

$$P(\text{denoise}|\mathbf{x}) = \frac{e^u}{1+e^u} \text{ and } P(\text{real}|\mathbf{x}) = \frac{1}{1+e^u}.$$

20.2. Now write $\mathcal{S}$ for the set of examples, where each example has the form $(\mathbf{x}_i, y_i)$, and

$$y_i = \begin{cases} 1 & \text{if } i\text{'th example is real} \\ -1 & \text{otherwise} \end{cases}$$

Check the log-likelihood of the dataset under this model is

$$\sum_{i \in \mathcal{S}} \left[u_i \left(\frac{1-y_i}{2} \right) - \log \left(1 + e^{u_i} \right) \right].$$

20.3. Recall

$$H_x(p,q) = -\mathbb{E}[p] [\log q] = -\sum_u p_u \log q_u$$

where the sum is over all elements with non-zero terms in p and q. For fixed p and varying q, check the largest value of $H_x(p,q)$ could be arbitrarily large.

20.4. Recall

$$H_x(p,q) = -\mathbb{E}[p] [\log q] = -\sum_u p_u \log q_u$$

where the sum is over all elements with non-zero terms in p and q. For fixed p and varying q, check the smallest value of $H_x(p,q)$ is $H(p) = -\sum_u p_u \log p_u$. **Hint:** Quick if you remember that $q_u \geq 0$ and $\sum_u q_u = 1$, frustrating if you don't.

20.5. Section 20.2.3 has: "This is convenient when you are not certain of the label, but have some probability that you believe." Explain.

20.6. Under the hinge loss, an example that is correctly classified has no loss if $\mathbf{a}^T\mathbf{x} + b > 1$. You can think of the margin as being given by the gap between the hyperplanes $\mathbf{a}^T\mathbf{x} + b = 0$ and $\mathbf{a}^T\mathbf{x} + b = 0$. What happens to the margin if $\mathbf{a}^T\mathbf{a}$ gets bigger?

20.7. You are given two linear classifiers trained on a large dataset. They have about the same training error, but $\mathbf{a}_1^T\mathbf{a}_1$ is much bigger than $\mathbf{a}_2^T\mathbf{a}_2$. Use the result of the previous example to choose the one you prefer, and explain why.

20.8. How is your choice in the previous exercise related to the weight decay of Section **??**?

LONGER PROBLEMS

20.9. This exercise explores relationships between a linear classifier, hinge loss, and an optimization problem.

(a) A training dataset for a two-class problem is called *linearly separable* if there exists a linear classifier that can classify every example correctly. Show that this property is equivalent to the property that, for every example, $y_i(\mathbf{a}^T\mathbf{x}_i + b) \geq 1$.

(b) You could obtain the classifier for a linearly separable problem by solving a constrained optimization problem. The constraints would be $y_i(\mathbf{a}^T\mathbf{x}_i + b) \geq 1$ (one per example). Use the argument about margins, above, to explain why $\mathbf{a}^T\mathbf{a}$ is a good objective function.

(c) You should expect that, for most linearly separable d-dimensional datasets, at the stationary point, $d+1$ constraints are active. Why? **Hint:** this is a delicate question, but not as difficult as it may seem. How many points determine a plane in d dimensions?

(d) Now assume that the training dataset is not linearly separable. Choose some $\mathbf{a}$, b. For each example construct $\xi_i = \max 1 - y_i(\mathbf{a}^T\mathbf{x} + b)0$. Show that $y_i(\mathbf{a}^T\mathbf{x}_i + b) \geq 1 - \xi_i$.

(e) Use the $\mathbf{a}$, b from the previous subexample. Assume $\mathbf{x}_i$ is misclassified by this linear classifier. Show that $\xi_i - 1$ is proportional to the perpendicular distance from the decision boundary to $\mathbf{x}_i$.

(f) Now show that the $\mathbf{a}$, b that minimizes

$$(1/2)\mathbf{a}^T\mathbf{a} + \lambda \sum_i \xi_i$$

subject to $y_i(\mathbf{a}^T\mathbf{x}_i + b) \geq 1 - \xi_i$ and $\xi_i \geq 0$ is the $\mathbf{a}$, b that minimizes

$$\sum_i \mathcal{L}_{\text{hinge}}(y_i(\mathbf{a}^T\mathbf{x}_i + b)) + \frac{\mu}{2}\mathbf{a}^T\mathbf{a}$$

for an appropriate choice of λ and μ.

PROGRAMMING EXERCISES

20.10. This is a simple exercise in fitting a two-class linear classifier that uses fixed, known features. Obtain the Wisconsin breast cancer dataset from the UCI machine learning repository. I found it at at `https://archive.ics.uci.edu/dataset/17/breast+cancer+wisconsin+diagnostic`. This dataset, due to William Wolberg, Olvi Mangasarian, Nick Street and W. Street, gives a feature vector for each of 569 needle aspirate biopsies and a diagnosis (benign or malignant). Ignore the ID attribute. **Note:** Logistic regression fits into second order optimization methods rather well, and it is easy to get gradient and hessian in closed form. You could also do stochastic gradient descent. The hinge loss does not fit into second order methods, but can be dealt with using stochastic gradient descent. It is a good idea to try multiple steplengths or consider a steplength schedule.

(a) Split the dataset into 69 validation and 500 training items *at random*. Fit a linear classifier to the 500 training items using logistic regression, then evaluate on the validation items. What accuracy do you get on the validation dataset?

(b) Shift and scale each feature to have zero mean and standard deviation one. Fit a linear classifier to the 500 training items using logistic regression, then evaluate on the validation items. Does this rescaling improve the performance of your classifier? Why?

(c) Now compute the mean and standard deviation of accuracies that you get by repeatedly splitting the dataset into 69 validation and 500 training items *at random*, fitting a linear classifier to the 500 training items using

logistic regression, then evaluating on the validation items. Use at least 10 splits.

(d) Can you improve over the previous results by regularizing the classifier? as a regularizer, use the squared magnitude of $\mathbf{a}$, so that you must now minimize $\mathcal{L}_{lr} + (\lambda/2)\mathbf{a}^T\mathbf{a}$. Choose a regularization weight λ using the following procedure. Select a set of possible weights, using at least five separated by an order of magnitude (so, for example, $1e-4, 1e-3, 1e-2, 1e-1, 1$). Split the dataset at random into 500 working items and 69 test items. Now for each regularization weight, split the working dataset into 450 training items and 50 validation items. Fit the classifier to the 450 training items, and evaluate accuracy on the 50 validation items. Now choose the regularization weight that gave the best performance on the validation items. Using that weight, fit a classifier to all 500 working items and evaluate on the remaining 69 test items. This is the accuracy of your regularized classifier.

(e) Can you improve over the previous results by using the hinge loss rather than logistic regression?