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). 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. 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*.

Predict

True	0	1	2	3	4	Class error
0	151	7	2	3	1	7.9%
1	32	5	9	9	0	91%
2	10	9	7	9	1	81%
3	6	13	9	5	2	86%
4	2	3	2	6	0	100%

TABLE 20.1: *The class confusion matrix for a multiclass classifier. 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).*

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). Table 20.1 gives an example. This is a class confusion matrix from a classifier built on a dataset where one tries to predict the degree of heart disease from a collection of physiological and physical measurements. There are five classes $(0 \ldots 4)$. The i, j'th cell of the table shows the number of data points of true class i that were classified to have class j. As I find it hard to recall whether rows or columns represent true or predicted classes, I have marked this on the table. For each row, there is a *class error rate*, which is the percentage of data points of that class that were misclassified. 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. This clearly isn't what is happening for table 20.1. Instead, you can see that the method is very good at telling whether a data point is in class 0 or not (the class error rate is rather small), but cannot distinguish between the other classes. This is a strong hint that the data can't be used to draw the distinctions that we want. It might be a lot better to work with a different set of classes.

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 training data has been selected and so isn't exactly like the test 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 our 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. This is likely to be possible. 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 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.*

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 16 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.

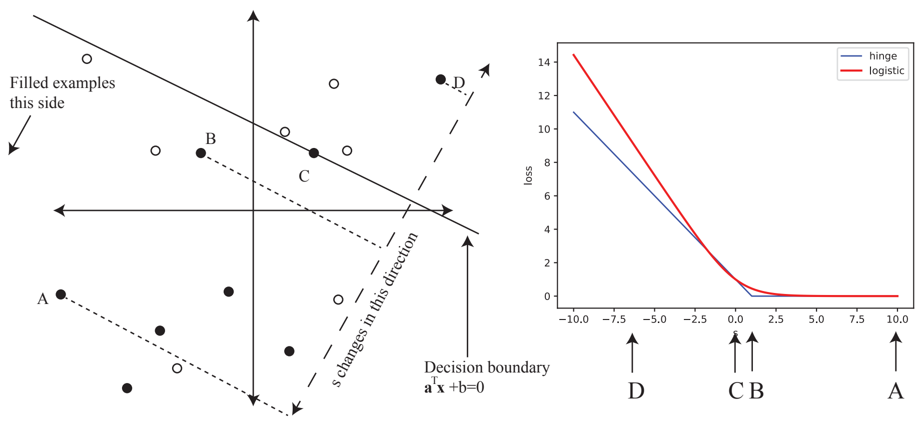
Filled examples this side
B
C
D
A
s changes in this direction
Decision boundary
$\mathbf{a}^T\mathbf{x}+b=0$
hinge
logistic
loss
s
D
C B
A

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.*

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 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

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

(you should check this; **exercises**). It would be natural to choose $\mathbf{a}$, b to maximize this likelihood, a procedure known as *logistic regression*

20.2.3 The Cross Entropy Loss

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

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

where the sum is over all elements with non-zero terms in p and q. 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

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

This means that you can interpret the log-likelihood as a comparison between the predicted distribution and the model distribution for each data item.

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

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

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

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

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

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

$$\sum_i \mathcal{L}_{\text{hinge}}(s_i).$$

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 **??**).

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 also be on the wrong side of 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.

20.3 BUILDING A CLASSIFIER

The feature representation $\mathbf{x}$ used to classify the image patches could come from a fixed encoder recovered from an autoencoder. There is no good reason to do this, and several good reasons not to. It is mildly inconvenient to train an encoder on one problem, and use it on another. Worse, an encoder trained to do one thing may not be good at another. The encoder parameters were chosen to be good at denoising images, rather than to be good at distinguishing between real and denoised patches.

Much more natural 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. 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 16.2.3, 16.2.1 (and variants in Section 17.3) apply. The main open question is good ways to turn a block of encoded features into a vector.

20.3.1 Pooling

Chapters 16 and 17 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 $\mathbf{a}^T\mathbf{C}\mathbf{u} + \mathbf{a}^T\mathbf{d} + b$, which is just a different linear classifier. Similarly, applying a fully connected 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.

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

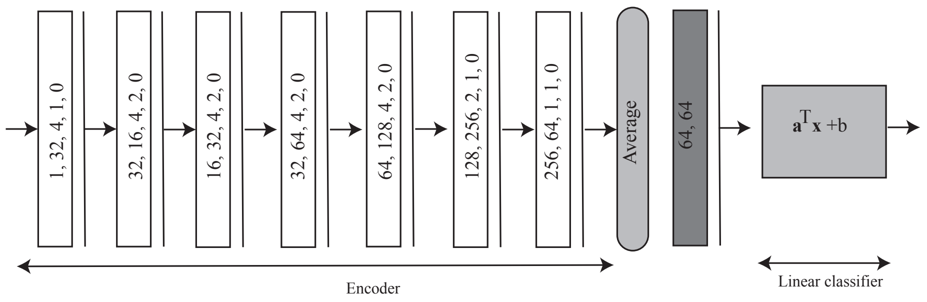

FIGURE 20.2: *A (very simple) classifier built out of convolutional layers, a pooling layer, a fully connected layer, and a linear classifier. Each light block represents a convolutional layer (arguments are, in order: input dimension, output dimension, kernel size, stride, padding); vertical lines are 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.*

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 16.2.1.

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 Adam (Section 17.3.6); 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 21.1.3 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).

The classifier of Figure 20.2 compares every pixel to every other pixel. You could reasonably expect that you could tell the difference between a denoised image and an image just by looking at image patches. The classifier of Figure 20.4 does this. This classifier computes a feature vector for each image patch; applies a linear

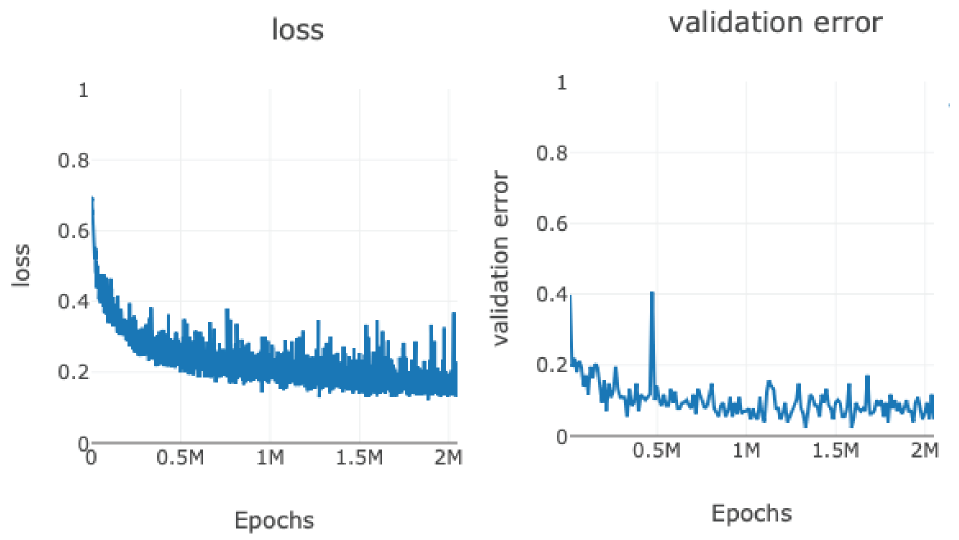

FIGURE 20.3: *A plot of training loss (***left***) and validation error rate (***right***) for the classifier of Figure 20.2, plotted against the number of training images the classifier has seen. These are fairly characteristic of a simple classifier. The loss mostly goes down, but there is some noise, 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).*

classifier to each such feature vector; and then reports (a) the hinge loss and (b) the score, averaged over all patches. It is worth taking a moment to check these statements against the figure. This classifier has the useful property that it checks whether individual image patches look good. The value of this property will become apparent (Section 8.2.2). This isn't a particularly good classifier for this application (validation error rate of 0.28).

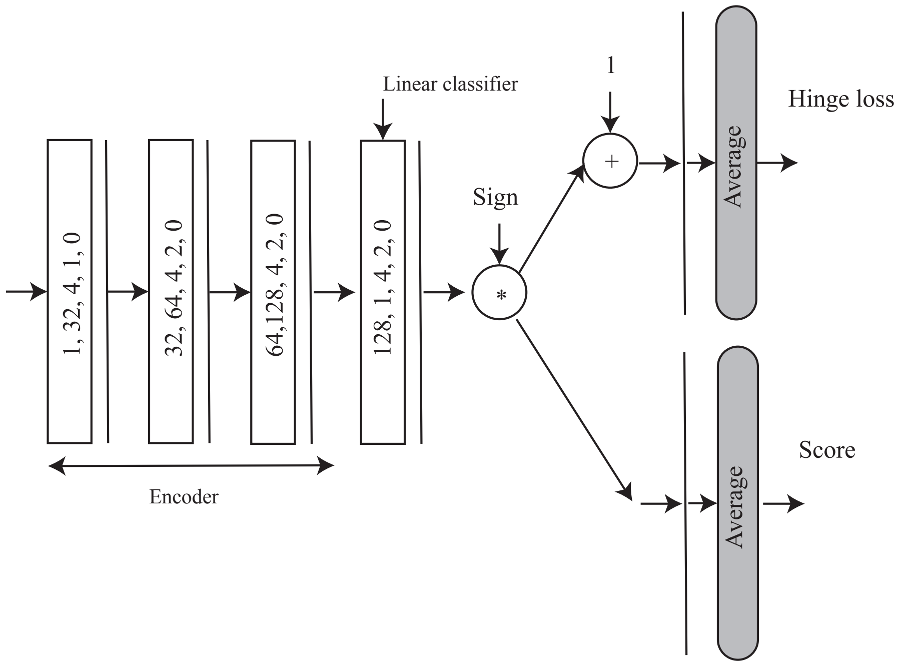

FIGURE 20.4: *This classifier looks at each image patch of a particular size, computes a score for that patch, and reports a loss (or score) averaged across the whole image. The patches overlap, and the size of the patches can be computed from the parameters of the convolutional layers (***exercises***). 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. This classifier is not particularly accurate – the error rate averaged over the whole validation set is* 0.28 *(so about one in four images will be misclassified) – but classifiers built like this will turn out to be useful.*