Last block:

Build an encoder and a decoder to:

Accept noisy image, produce clean version

By:

Constructing loss
Applying various tricks for good behavior
Adam/SGD/something to get minimal loss on training data

Idea:

exploit this machinery to classify image; running eg is real vs denoised

Classification

Take a feature representation and apply a label

eg credit card transaction (valid/fraud)

By some procedure

usually, constructed out of training data

crucial point:

classifier should work well on future data

Hugely useful, very broad idea

Model example: is image denoiser output, or is it original?

Evaluating classification

Accuracy:

fraction of classification attempts that result in right label

Error rate:

fraction of classification attempts that result in wrong label

Bayes error:

error rate that the very best possible classifier makes

usually, very hard to know

Evaluating classification

One class is boring:

Binary classifiers:

Two classes (yes/no; real/denoise; valid/fraud) etc.

Error rate never greater than 50%

False positive rate = fraction of negative examples labelled positive
(Type I error rate)

False positive rate = fraction of negative examples labelled positive
(Type II error rate)

Error rates interact!

Evaluating classification

Always:

evaluate on data not used to train

(otherwise evaluation is biased optimistic)

compare to baselines

(reasonable alternatives)

report TP rate and FP rate

Train error, test error, overfitting

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.

Estimating error rate

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?

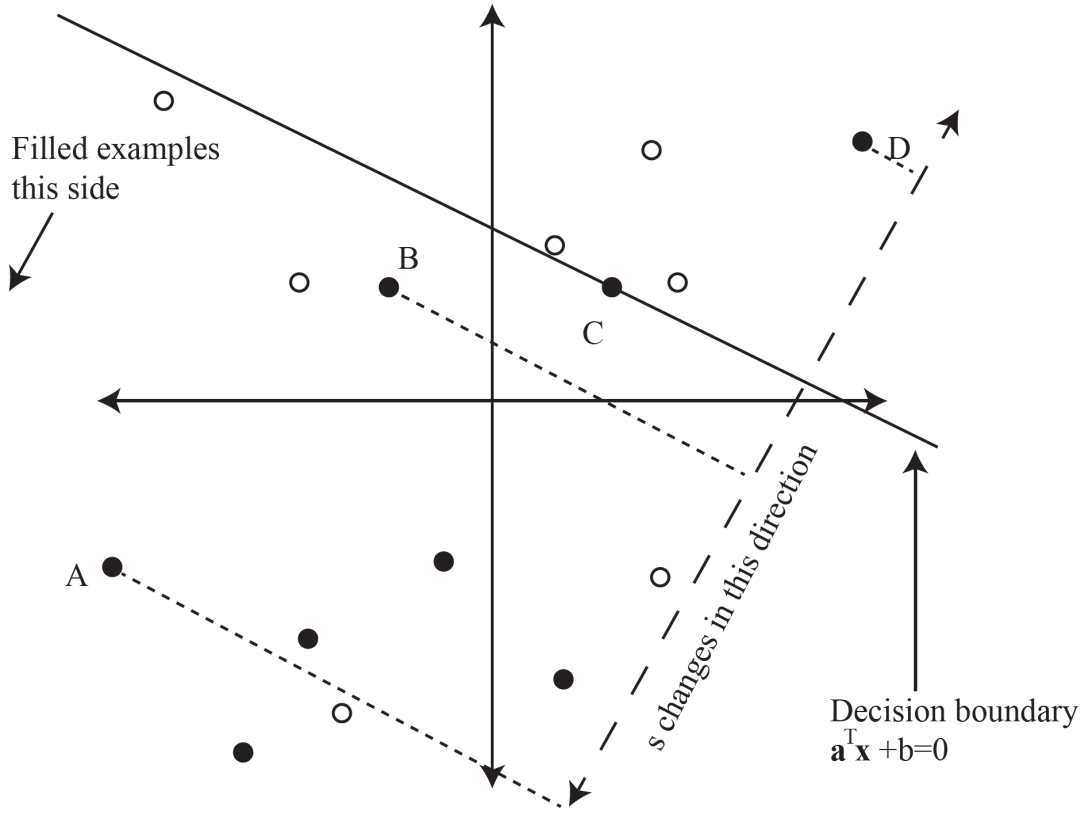
Cross validation

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*.

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.*

The decision boundary



$$\text{maps } \mathbf{x} \text{ to } u(\mathbf{x}; \mathbf{a}, b) = (\mathbf{a}^T\mathbf{x} + b),$$

Choosing a, b

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

Choosing a, b - II

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

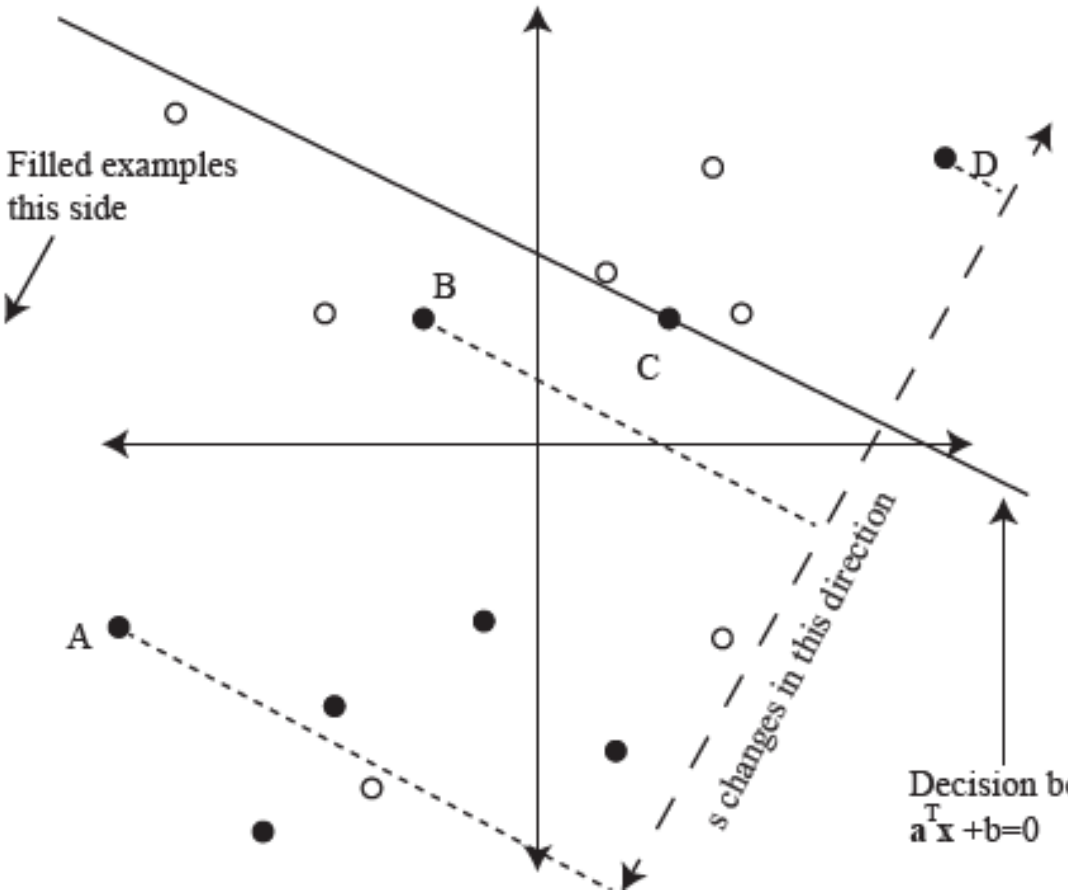
$$\mathcal{L}_{lr} = \sum_{i \in \mathcal{S}} \left[u_i \left(\frac{1 - y_i}{2} \right) - \log \left(1 + e^{u_i} \right) \right]$$

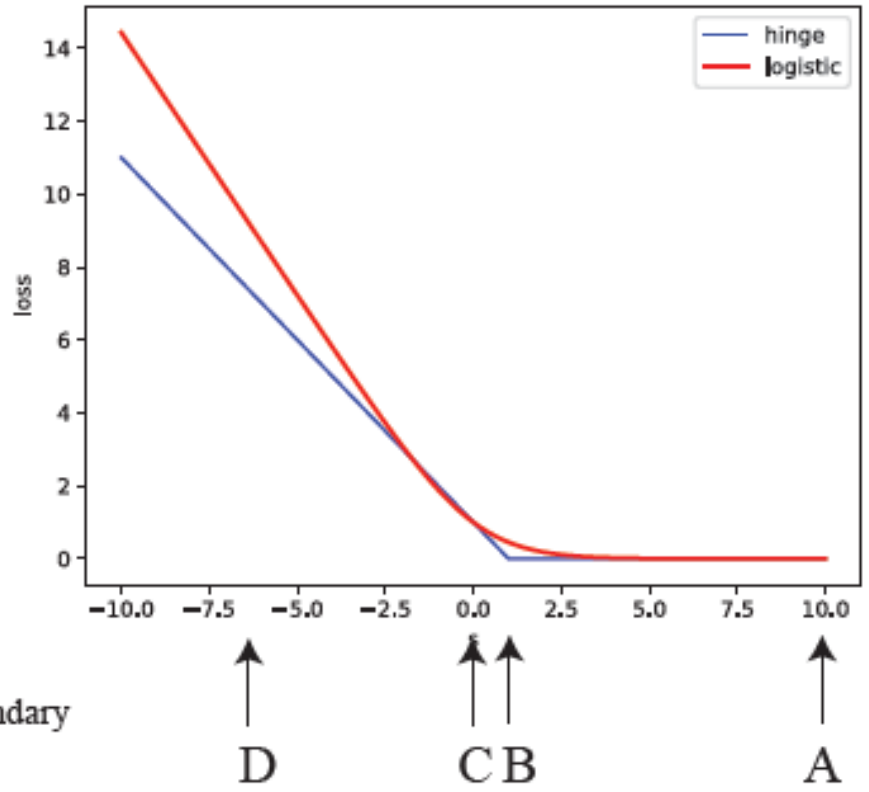
Choose a, b to maximize log-likelihood

equivalently minimize negative log-likelihood

Logistic regression

Logistic regression





Cross entropy between distributions

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

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

If they're the same, entropy of that dist

If different, bigger

Cross entropy loss

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.

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).

Hinge loss

Write $s_i = y_i u_i = y_i(\mathbf{a}^T\mathbf{x} + b)$.

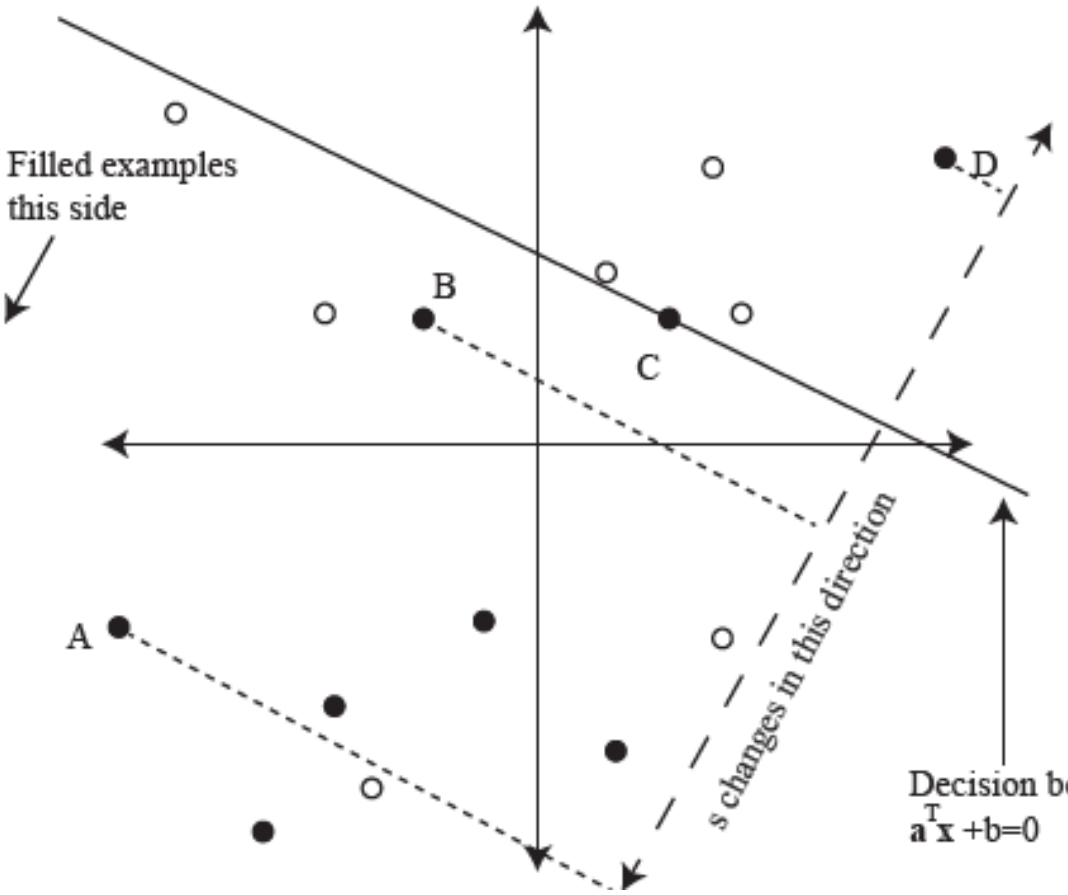
to zero (and so the example gets closer to the hyperplane on the right side), the logistic loss grows. If s_i is a lot smaller than zero (and so the example is far from the hyperplane and on the wrong side), the loss grows close to linearly in s_i There are other loss functions that have this behavior. The *hinge loss* function

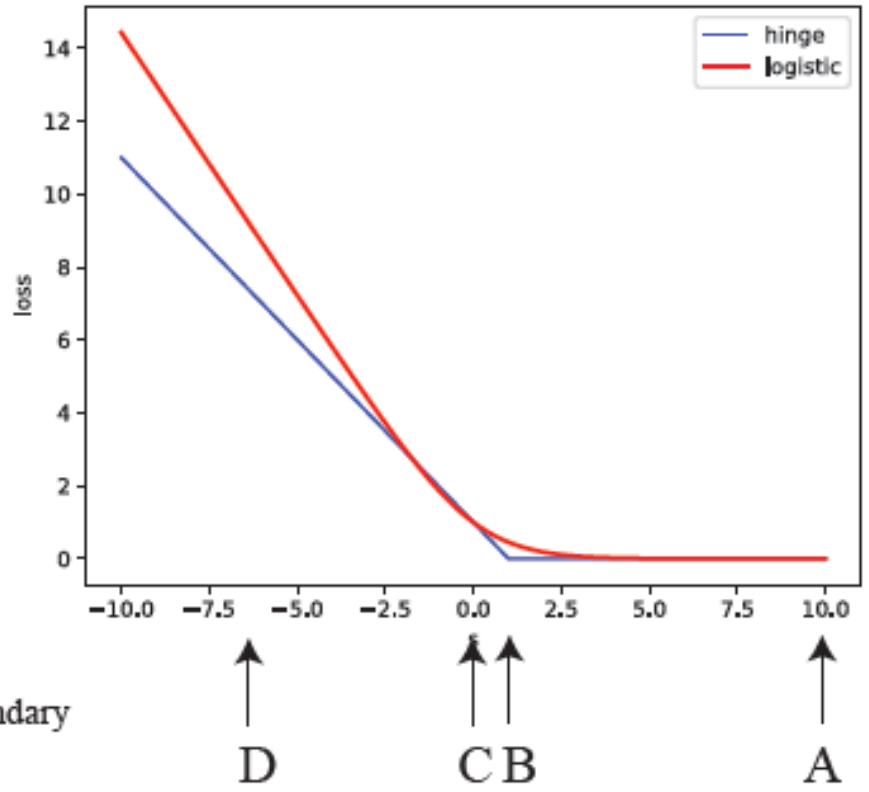
$$\mathcal{L}_{\text{hinge}}(s) = \max(1 - s, 0)$$

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

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

Hinge loss vs logistic regression





But what should x_i be?

Idea:

get it from an encoder

what encoder?

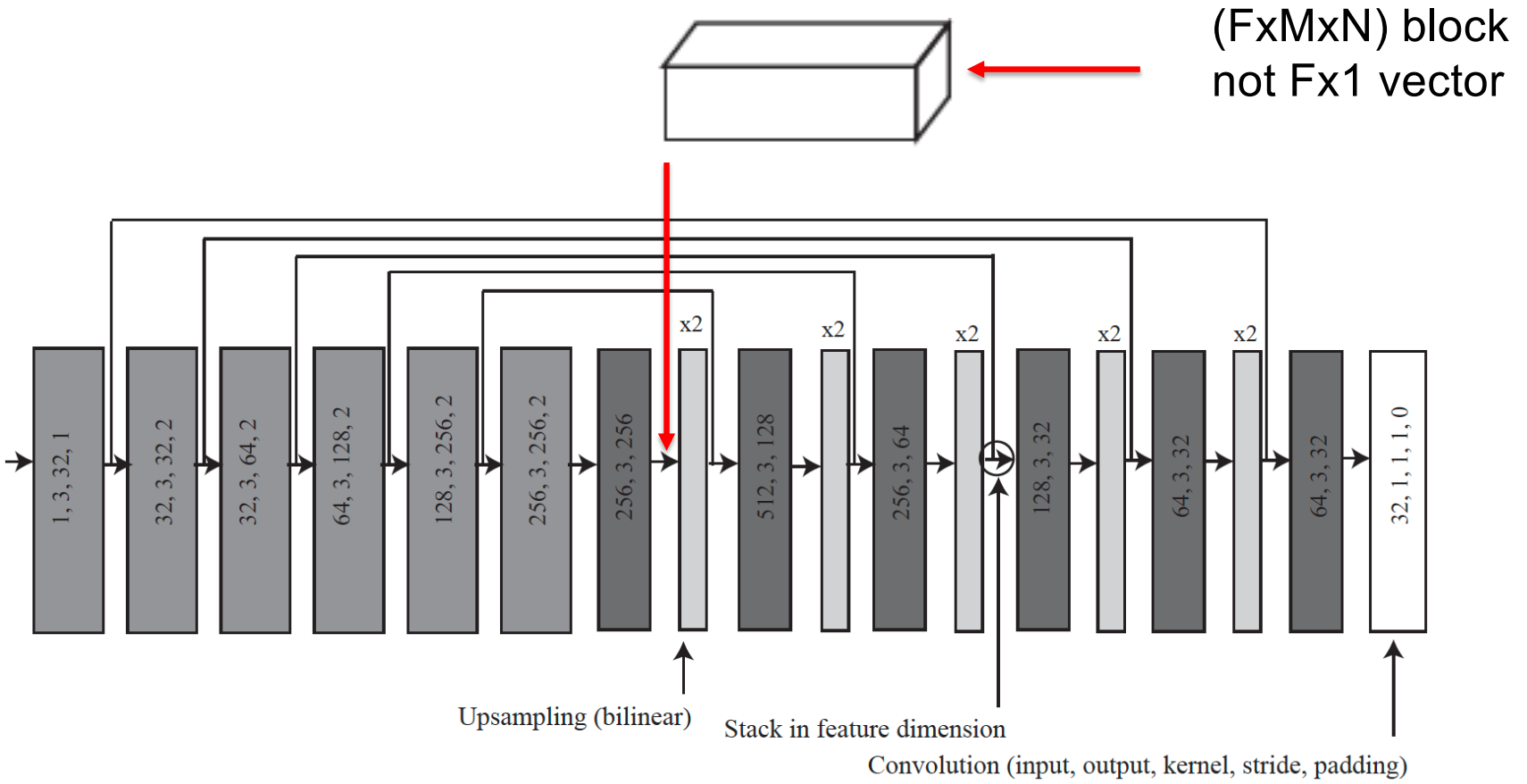
Idea:

train encoder to produce x_i so that classification is best

Issue:

get from encoder output to vector without too much drama

Deeper autoencoder



Pooling

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.

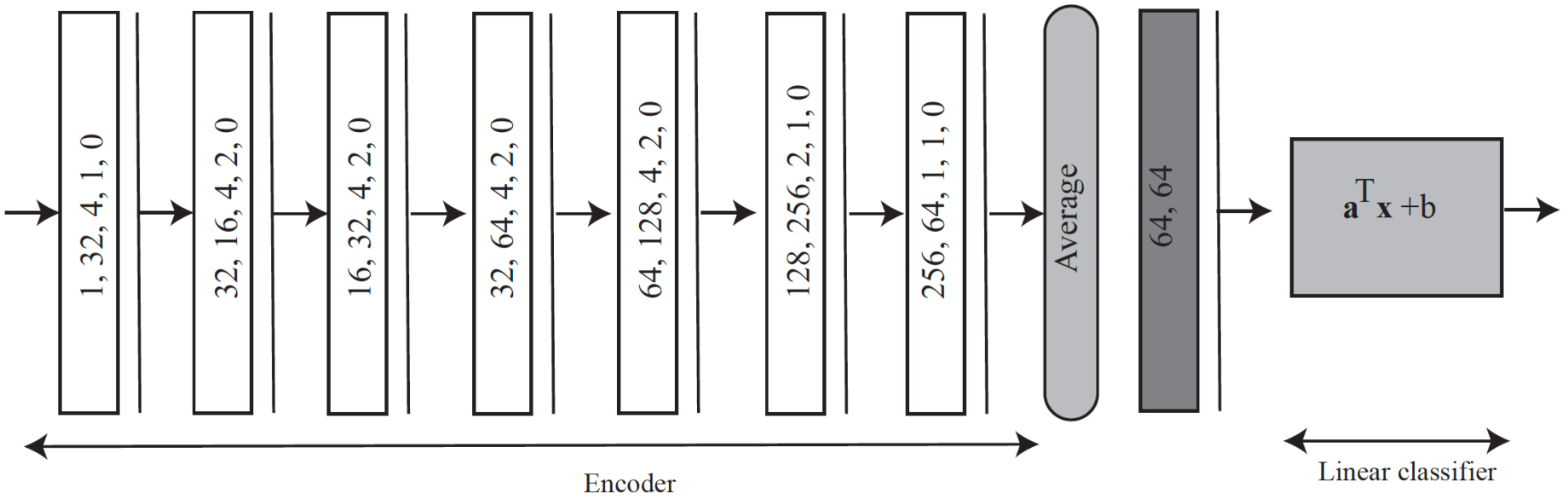
Fully connected layers

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.

Very simple classifier



Training the very simple 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.

Training the very simple classifier

Example case:

- is image just an image, or the output of a denoiser?

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).

Training the very simple classifier