Neural Networks

Neural networks go back many decades. They were a hot topic in the late 1980s, before many newer methods were invented.

The newer methods, like lasso, random forests, and boosting, required less computing power and less tinkering to get good performance.

But neural networks have recently become a very hot topic because of major improvements in performance associated with changes in their design. They also have a new name: **deep learning**.

Deep learning is intimately associated with **artificial intelligence**, including **machine translation** and **generative AI** like ChatGPT, Claude, and DeepSeek.

Modern AI uses **Large Language Models**, or **LLMs**, which are enormous deep learning models trained on vast amounts of data.

If given a large enough dataset, LLMs can work remarkably well at tasks like image recognition, and making fake photos and videos.

These slides are based on Chapter 10 of ISLR/ISLP.

Many neural networks can be thought of as nonlinear regression models with a particular structure and a great many parameters.

A simple **feed-forward neural network** has the form

$$\begin{aligned} f(\boldsymbol{x}) &= \beta_0 + \sum_{k=1}^{K} \beta_k h_k(\boldsymbol{x}) \\ &= \beta_0 + \sum_{k=1}^{K} \beta_k \, g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_j\right) \end{aligned} \tag{1}$$

Here $\boldsymbol{x}$ is the input vector and K is the number of **hidden units**.

The user picks the number of **hidden layers**, in the case of Figure 18.1 just one, and the number of hidden units in each layer, in this case 5.

The user also chooses the form of the **activation function** $g(z)$. More about this later.

To simplify notation, write

$$A_k = h_k(\boldsymbol{x}) = g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} x_j\right). \tag{2}$$

Then (1) can be written as

$$f(\boldsymbol{x}) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k, \tag{3}$$

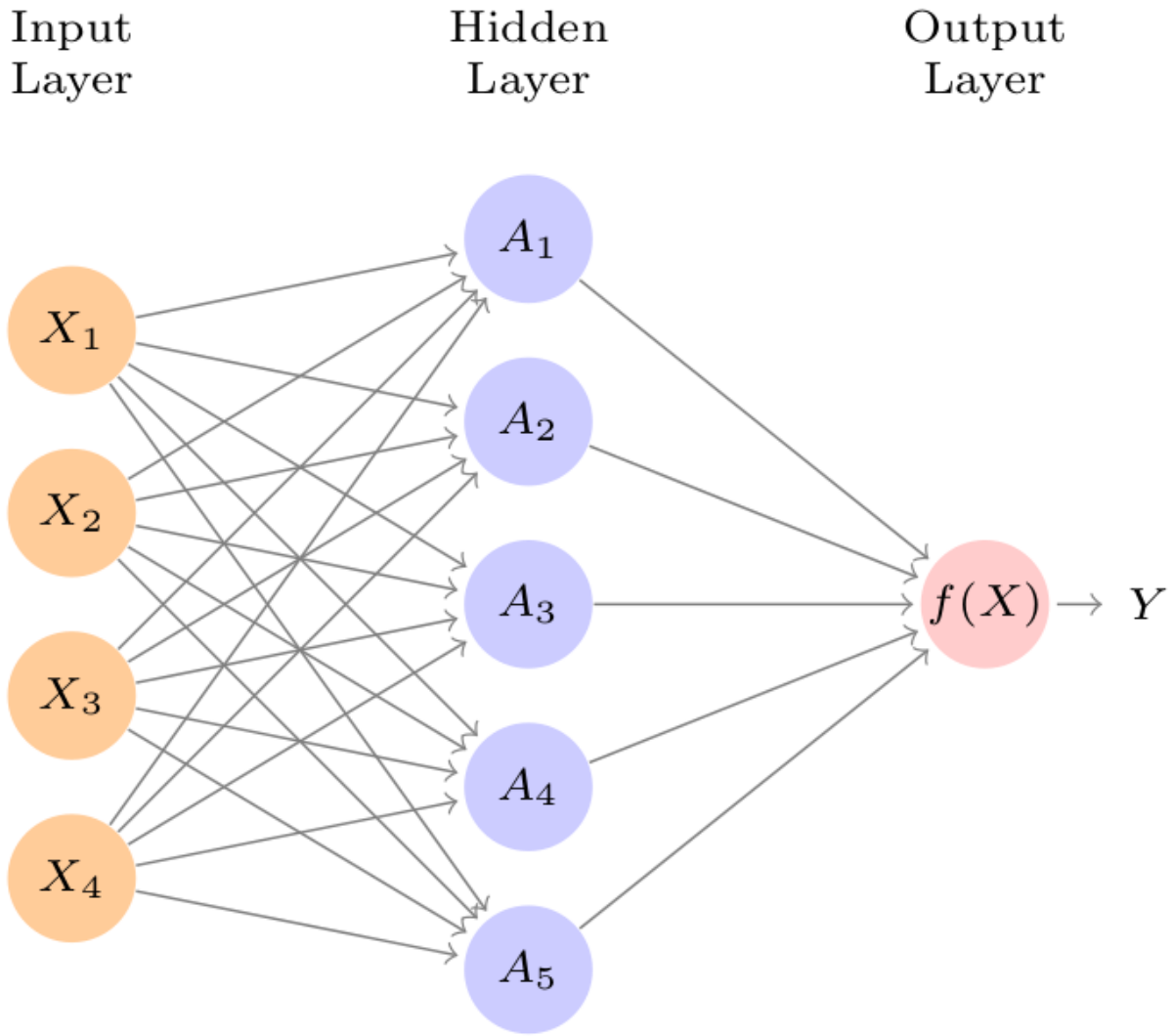
where $A_k = A_k(\boldsymbol{x})$. The A_k are called **activations**.

All of the parameters $\beta_0, \ldots, \beta_K$ and $w_{k0}, \ldots, w_{kp}$ for $k = 1, \ldots, K$ need to be estimated simultaneously.

The w_{kj} for $j > 0$ are called **weights** instead of slope coefficients. The w_{k0} are called **biases** instead of intercepts.

If $p = 4$ and $K = 5$, there are $5 + 5 \times 5 = 30$ parameters.

Figure 18.1 A neural network with one hidden layer



The inputs enter the model in the **input layer**.

They are then transformed into activations in the **hidden layers**.

- Until a decade or so ago, most neural networks involved one or just a few hidden layers.
- But modern deep-learning models typically involve many hidden layers. That's why they are called "deep".

It is essential that the activation function be nonlinear. Otherwise, $f(\boldsymbol{x})$ just collapses to the equivalent of a linear regression function.

For many years, activation functions were **sigmoid functions**, of which the logistic function

$$g(z) = \frac{1}{1+\exp(-z)} = \frac{\exp(z)}{1+\exp(z)} \tag{4}$$

was particularly popular. So was $\tanh(x)$, which equals $\big(\exp(2x)-1\big)/\big(\exp(2x)+1\big)$.

But, quite recently, the logistic function has been replaced by the **rectified linear unit**, or **ReLU**, function

$$g(x) = \max(0, x). \tag{5}$$

ISLR/ISLP gives the impression that ReLU replaced logistic because it is cheaper to compute. That is not the main reason.

Sigmoid functions seem natural, because they map smoothly from the real line to an interval. However, they have a very important deficiency.

When the argument is small, a sigmoid function will be close to 0, and when the argument is large, it will be close to 1. Thus the function is very flat in these cases. It is said to **saturate**.

Changing the weights has little effect when an activation function is saturated. The optimization procedure does not know where to look.

This a severe identification problem. It is called the **vanishing gradient** problem.

Although the gradient for the ReLU function completely vanishes when $z \leq 0$, it never vanishes when $z > 0$.

Changing the intercept, or bias, will change the inflection point below which the gradient vanishes.

- The problem of vanishing gradients tends to be especially severe for models with several layers.
- If saturation occurs for any layer, making changes to the weights for lower layers will have little impact on the model fit.
- It is not a coincidence that deep learning and the ReLU function arrived at the same time.
- Deep learning models with sigmoid activations would be very hard to estimate.

Figure 18.2 compares the logistic and ReLU functions.

They don't look very similar!

Notice how the logistic function saturates at both ends.

Figure 18.2 The logistic and ReLU activation functions

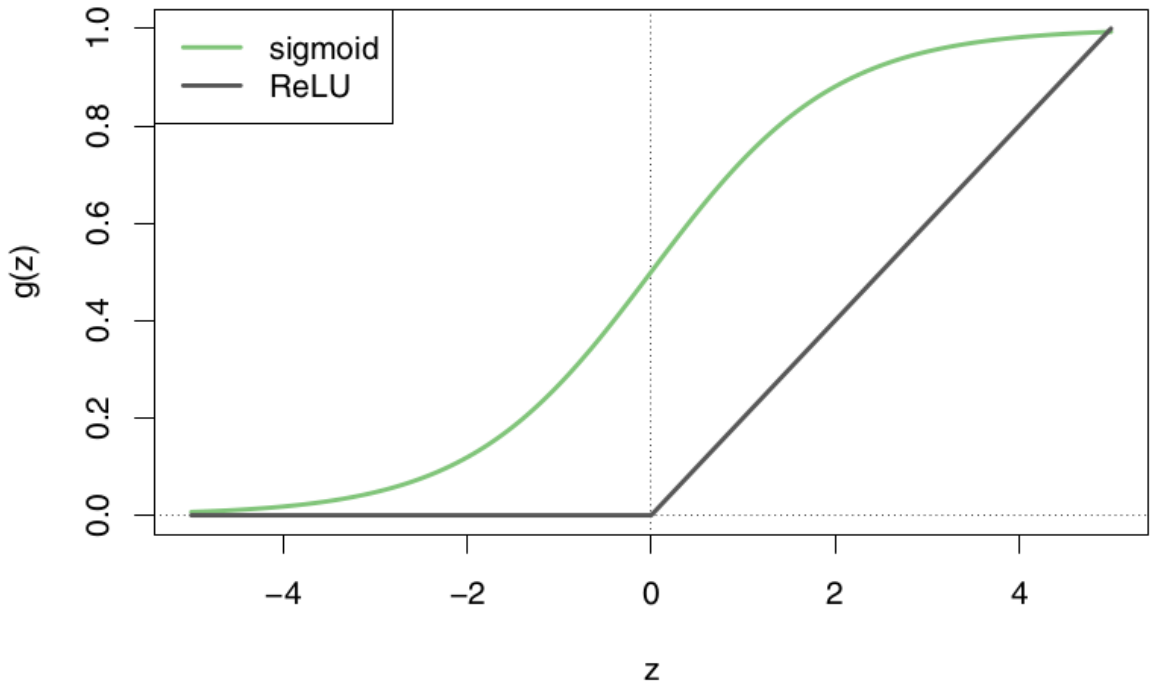


FIGURE 10.2. *Activation functions. The piecewise-linear* `ReLU` *function is popular for its efficiency and computability. We have scaled it down by a factor of five for ease of comparison.*

The sum of two nonlinear activation functions of two variables can yield an interaction term. Suppose that

$$h_1(\boldsymbol{x}) = (x_1 + x_2)^2 \tag{6}$$

$$h_2(\boldsymbol{x}) = (x_1 - x_2)^2. \tag{7}$$

Then if we set $w_{01} = w_{02} = 0$, $w_{11} = w_{12} = w_{21} = 1$, $w_{22} = -1$, $\beta_1 = 1/4$, and $\beta_2 = -1/4$ in $f(\boldsymbol{x})$, given in (1), and substitute these $h(\cdot)$ functions into $f(\boldsymbol{x})$, we obtain

$$f(\boldsymbol{x}) = \frac{1}{4}\left((x_1 + x_2)^2 - (x_1 - x_2)^2\right) = x_1 x_2. \tag{8}$$

Thus we obtain a very simple interaction term by taking a weighted sum of two activation functions.

The activation functions $h_1(\boldsymbol{x})$ and $h_2(\boldsymbol{x})$ are quadratic, not ReLU, so this is not what a neural net would actually do.

But, even though the functional form of $g(z)$ is extremely restrictive, a single hidden layer with enough activations can model all sorts of interactions.

If, in addition, there are many hidden layers, a neural network model based on the ReLU activation function can be extremely flexible.

To estimate a neural network for the regression case, we just minimize

$$\text{SSR}(\boldsymbol{B}) = \sum_{i=1}^{n} \left(y_i - f(\boldsymbol{x}_i, \boldsymbol{B})\right)^2 \tag{9}$$

with respect to the parameter vector $\boldsymbol{\beta}$ and the matrix $\boldsymbol{W}$, which are collected into the (very long) vector $\boldsymbol{B}$.

For a binary or multinomial response, we would instead maximize the loglikelihood.

But there may be billions of parameters in $\boldsymbol{B}$, and the $f(\boldsymbol{x}_i, \boldsymbol{B})$ functions are all nonlinear! We are about to see a "small" example with about 1/4 of a million parameters.

Multilayer Neural Networks

In theory, a single hidden layer with a large number of units has the ability to approximate most functions.

There is a very famous paper by Hornik, Stinchcombe, and White (1989) which proves this; it has 31,583 citations.

Unfortunately, the theory in that paper is very misleading.

Discovering a good approximation is much easier if we use models with multiple layers, each of modest size.

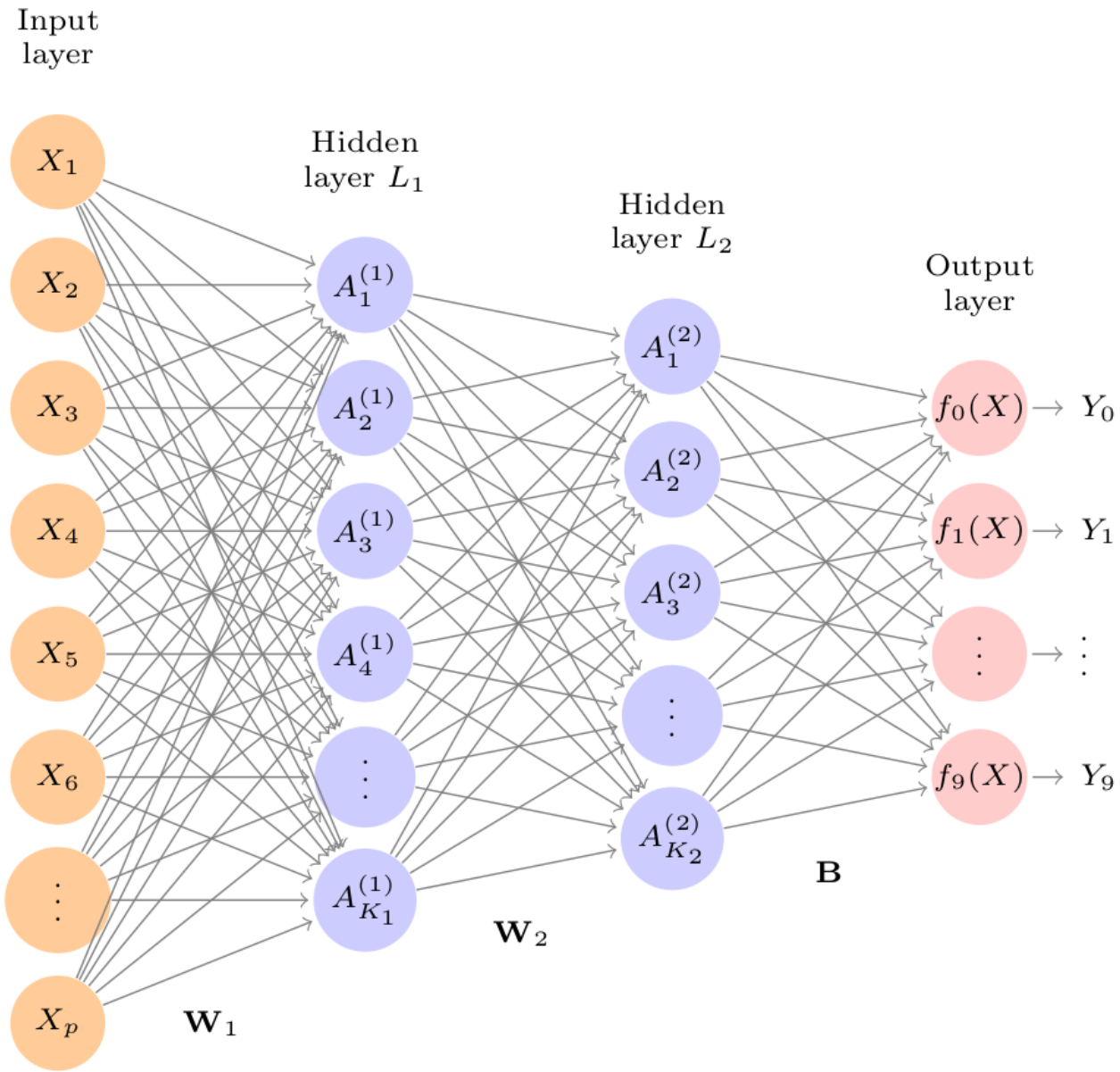
This is what deep learning does. But roughly 25 years passed after the HSW paper before deep learning took off!

- Were people who studied neural nets misled by the theory?
- Did the saturation problem with sigmoid functions make it hard to estimate multilayer models?
- Was there just not enough computing power in the early 1990s?

Section 10.2 of ISLR/ISLP discusses multilayer networks in the context of a specific problem: recognizing handwritten digits for zip codes.

- The training dataset contains 60,000 observations (coded images), and the test dataset contains 10,000. This is the `MNIST` dataset.
- Each image coded as $28 \times 28 = 784$ pixels, and each pixel is given a greyscale value between 0 and 255.
- Thus the input vector $\boldsymbol{x}$ contains $p = 784$ numbers between 0 and 255, stored in a particular order.
- The output is a number between 0 and 9, but it is stored as 10 dummy variables in the vector $\boldsymbol{y}$.
- A suitable neural network model has two hidden layers, with 256 and 128 units, respectively. See Figure 18.3.
- This model has 235,146 parameters, almost four times the number of observations.

Figure 18.3 Neural network with two hidden layers



The first hidden layer has activations

$$A_k^{(1)} = h_k^{(1)}(\boldsymbol{x}) = g\left(w_{k0}^{(1)} + \sum_{j=1}^{p} w_{kj}^{(1)} x_j\right). \tag{10}$$

The second hidden layer has activations

$$A_\ell^{(2)} = h_\ell^{(2)}(\boldsymbol{x}) = g\left(w_{\ell 0}^{(2)} + \sum_{k=1}^{K_1} w_{\ell k}^{(2)} A_k^{(1)}\right). \tag{11}$$

So the second-layer activations depend directly on the first-layer ones and indirectly on $\boldsymbol{x}$.

No matter how many layers there are, everything ultimately depends on the input vector $\boldsymbol{x}$. For the single-output case, we can write $y = h(\boldsymbol{x})$, but $h(\cdot)$ may be extremely complicated.

Since there are ten responses instead of one, the output layer involves ten different linear models.

These can be written as

$$z_m = \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} h_\ell^{(2)}(\boldsymbol{x}) = \beta_{m0} + \sum_{\ell=1}^{K_2} \beta_{m\ell} A_\ell^{(2)}, \tag{12}$$

for $m = 0, 1, \ldots, 9$.

The z_m are like index functions for a logistic regression. We need to transform them into probabilities using the **softmax activation function**:

$$f_m(\boldsymbol{x}) = \Pr(y = m \,|\, \boldsymbol{x}) = \frac{\exp(z_m)}{\sum_{c=0}^{9} \exp(z_c)}. \tag{13}$$

In the denominator, c denotes class, since we cannot use m.

Using (13) ensures that the $f_m(\boldsymbol{x})$ are nonnegative and sum to unity.

This works just like using the logistic function to transform an index function into a probability in a logit model.

Note that we really only need to obtain z_m for $c - 1$ classes.

Instead of minimizing a sum of squared residuals, we minimize the negative multinomial loglikelihood

$$-\sum_{i=1}^{n}\sum_{m=0}^{9} y_{im} \log\left(f_m(\boldsymbol{x}_i)\right). \tag{14}$$

This is also called the **cross-entropy**.

Equation (14) is a generalization of the deviance, which can be minimized to obtain estimates for a binary logit model.

We have to minimize it with respect to $\boldsymbol{B}$, which contains all the coefficients in $\boldsymbol{\beta}$, $\boldsymbol{W}_1$, which contains the weights for the first hidden layer, and $\boldsymbol{W}_2$, which contains the weights for the second hidden layer.

There are $1290 + 200,960 + 32,896 = 235,146$ parameters to estimate!

How we can estimate this many parameters will be discussed later. It seems obvious that we need to use some sort of regularization.

Applying the particular neural network described above to the `MNIST` dataset yields remarkably good results.

Table 1: **Test Error Rates for the MNIST Dataset**

Method	Test Error
Neural Network + Ridge	0.023
Neural Network + Dropout	0.018
Multinomial Logit	0.072
Linear Discriminant Analysis	0.127

So the NN blows away multinomial logit. But the latter involves no real nonlinearities and no regularization.

Code to estimate this model is presented in Section 10.9.2. The R book uses `keras`, but the Python book uses `torch`. However `torch` can also be used in `R`, and code is available on the book's website.

Convolutional Neural Networks

Neural networks can work extremely well for classifying images.

Convolutional NNs are designed for this purpose.

- They use two types of hidden layer, called **convolutional layers** and **pooling layers**.
- Convolution layers search for instances of small patterns in the image. Pooling layers downsample these to select subsets that occur together.
- Modern deep-learning neural nets use many convolution layers and many pooling layers.

Studying convolutions is way beyond the level of this course.

The usual definition is an integral:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau) g(t - \tau) d\tau. \tag{15}$$

The convolution layers in neural nets use matrix multiplication, not integrals.

They **convolve** the original image with a **convolution filter**. This involves matrix multiplication of portions of the image by the filter.

For example, if the filter is a 2×2 matrix, we multiply it by every 2×2 submatrix of the image. The result is half the size of the original.

- When a submatrix resembles the filter, the result will be large and will look like the filter.
- When a submatrix does not resemble the filter, the result will be small.

There are many varieties of convolution filter, designed to pick out features of images that the eye (or the A.I.) can use to identify it.

The filters are not usually 2×2, and they are not necessarily square.

The number of convolution filters in a convolution layer is analogous to the number of units in a hidden layer.

A pooling layer condenses the image from the preceding convolution layer.

For example, the "max pooling" operator could take every 2×2 submatrix and replace it by the largest element, thus reducing the number of elements by a factor of four.

There can be multiple convolution layers between two pooling layers.

Once the images get very small, they are "flattened." This means that each pixel is treated as a separate unit.

There are then some conventional hidden layers, with no more convolutions, and finally a softmax activation to choose the class.

Modern image-processing neural nets often use **data augmentation**.

Each training image is replicated many times, with each replicate randomly distorted so that human recognition is unaffected.

This is similar to ridge regularization, which implicitly involves adding fake data (scaled identity matrix added to $\boldsymbol{X}^\top \boldsymbol{X}$).

Data augmentation seems like a really good idea.

- As long as the distortions really would not fool the human eye, the additional training images provide new information.
- With many algorithms, data augmentation is done on the fly. The distorted images are created, used in one training step, and then deleted as the algorithm proceeds.

Image recognition is not yet perfect. See Figure 18.4.

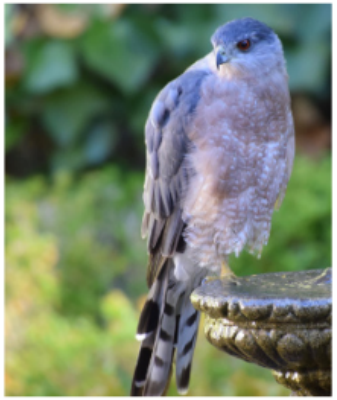
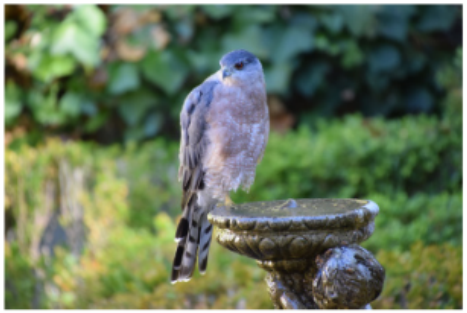
Here the book is using a classifier called `resnet50` that was previously trained using millions of images.

When new images, or new classes of images, become available, we can use pre-trained hidden layers and just estimate the last few layers of the network. Pre-training is a big part of making LLMs viable.

See the `keras` package.

Are there interesting applications for convolutional neural networks in Economics?

Figure 18.4 Image classification using `resnet50`


flamingo		Cooper's hawk		Cooper's hawk	
flamingo	0.83	kite	0.60	fountain	0.35
spoonbill	0.17	great grey owl	0.09	nail	0.12
white stork	0.00	robin	0.06	hook	0.07

Lhasa Apso		cat		Cape weaver	
Tibetan terrier	0.56	Old English sheepdog	0.82	jacamar	0.28
Lhasa	0.32	Shih-Tzu	0.04	macaw	0.12
cocker spaniel	0.03	Persian cat	0.04	robin	0.12

Document Classification

How can we, for example, take a bunch of statements made by the Bank of Canada and convert them into something that can be used to forecast interest rates or explain how the stock market reacted?

This requires us to **featurize** each of a number of documents.

The documents vary in length, were written at different times, by people (or committees) who may or may not be different, and may contain slang or typos (not for B. of C. statements!).

The simplest approach is **bag-of-words**. We just see which words occur in each document.

We need to decide which words to count. Perhaps find the ones that occur most frequently in the entire training set, say the top 10,000.

We may or may not choose to avoid counting words like “a” and “the.”

There are several ways to keep track of the words in a document.

- Use a binary variable to indicate whether a particular word is present; this is what ISLR/ISLP does.
  - In that case, each input vector has, say, 10,000 elements, most of them 0 but a few of them 1.
- Use a count variable for the number of times each word appears.
- Use a number between 0 and 1 to indicate the proportion of counted words each word accounts for.
  - This is just the count divided by the total number of words.

Bag-of-words is being replaced by more advanced methods.

We can count sequential pairs of words, or maybe triplets, instead of individual words.

There may be particular pairs or triplets or quadruplets that often appear in documents of the type we are studying.

We can explicitly treat a document as a sequence of words.

In Section 10.4, ISLR/ISLP provides an application of bag-of-words to IMDb movie reviews. The output is whether a review is positive or negative, presumably as judged by a human.

ISLR/ISLP compares lasso with a two-class neural network with two hidden layers, each with 16 ReLUs.

It uses 23,000 observations for training, 2000 for validation, and an unspecified number for testing.

Lasso is indexed by $-\log(\lambda)$. Penalty term gets smaller as it increases.

The neural nets are indexed by the number of **training epochs**, which we will discuss in the context of how to estimate neural networks.

Increasing the number of epochs allows the model to fit better, almost certainly overfitting as it gets large.

Figure 18.5 shows accuracy (fraction correct).

The book does not report confusion matrices for either method with the tuning parameter chosen optimally for the validation set.

Figure 18.5 Lasso versus neural net for document classification

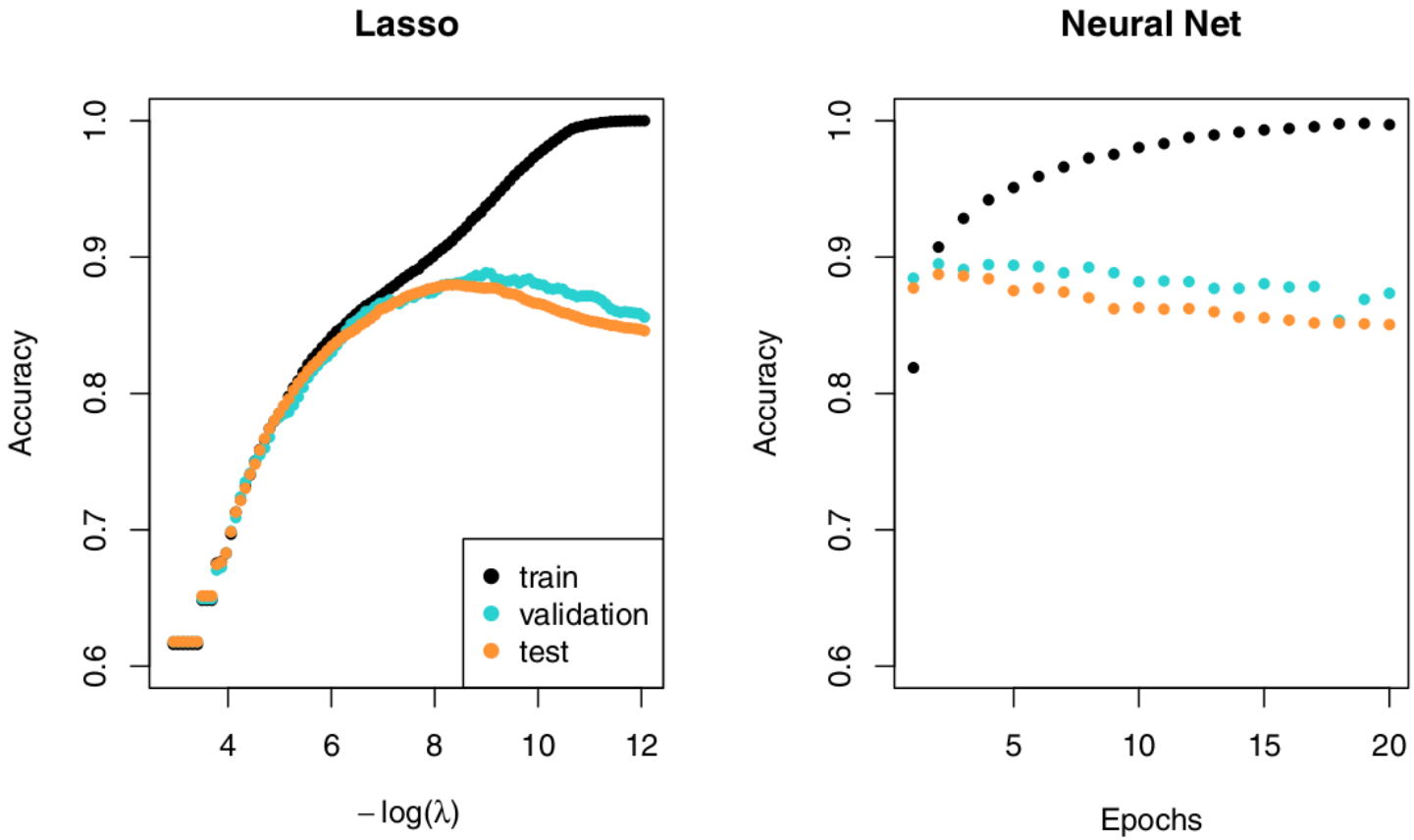


FIGURE 10.11. *Accuracy of the lasso and a two-hidden-layer neural network on the* `IMDb` *data. For the lasso, the x-axis displays* $-\log(\lambda)$*, while for the neural network it displays epochs (number of times the fitting algorithm passes through the training set). Both show a tendency to overfit, and achieve approximately the same test accuracy.*