Homework 4

Brandon Amaral, Monte Davityan, Nicholas Lombardo, Hongkai Lu

2022-10-03

1 Murphy Chapter 13 & ISLR Chapter 10 Summaries

Murphy Chapter 13

13.1 Introduction A simple way of increasing the flexibility of a model is to perform a feature transformation $\phi(\mathbf{x})$, which then makes the model

$$f(\mathbf{x};\theta) = \mathbf{W}\phi(\mathbf{x}) + \mathbf{b}.$$

Having to specify the the feature transformation by hand is limiting, so a natural extension is to give the feature extractor its own parameters θ_2 to get

$$f(\mathbf{x};\theta) = \mathbf{W}\phi(\mathbf{x};\theta_2) + \mathbf{b},$$

where $\theta = (\theta_1, \theta_2)$ and $\theta_1 = (\mathbf{W}, \mathbf{b})$. If we repeat this recursively, we compose L functions to get

$$f(\mathbf{x};\theta) = f_L(f_{L-1}(\cdots(f_1(\mathbf{x}))\cdots)),$$

where $f_\ell(\mathbf{x}) = f(\mathbf{x};\theta_\ell)$ is the function at layer ℓ. This is the basis of deep neural networks (DNNs).

The term "DNN" is a larger family of models in which differential functions are composed into any kind of directed acyclic graph, mapping input to output. A feedforward neural network or multilayer perceptron (MLP) is a simple example where the DAG is a chain.

An MLP assumes a fixed-dimensional input $\mathbf{x} \in \mathbb{R}^D$, commonly called `structured data'' or`tabular data'' in an $N \times D$ design matrix.

13.2 Multilayer Perceptrons (MLPs) A perceptron is a deterministic version of logistic regression, i.e.

$$f(\mathbf{x};\theta) = \mathbb{I}(\mathbf{w}^\mathbf{T}\mathbf{x} + b \geq 0) = H(\mathbf{w}^\mathbf{T}\mathbf{x} + b),$$

where $H(a)$ is the heaviside step function or linear threshold function. The decision boundaries represented by perceptrons are linear, so they are limited in what they can represent.

13.2.1 The XOR Problem The goal is to learn a function that computes the exclusive OR of its two binary inputs. The data is not linearly separable, so a perceptron cannot represent this mapping. We can overcome this by stacking multiple perceptrons on top of each other. This is a multilayer perceptron.

To solve the XOR problem, we can use 3 perceptrons h_1, h_2, and y. There are 2 input nodes x_1, x_2, two hidden units h_1, h_2, and the output node y. The hidden units and output node also receive bias terms from constant nodes. The first hidden unit computes $h_1 = x_1 \wedge x_2$ by using appropriately set weights. The input nodes x_1 and x_2 are weighted by 1.0 with bias term -1.5, and so h_1 will fire if and only if x_1 and x_2 are both on, since

$$\mathbf{w}_1^T\mathbf{x} - b_1 = [1.0, 1.0]^T[1, 1] - 1.5 = 0.5 > 0.$$

Similarly, $h_2 = x_1 \vee x_2$ and $y = \overline{h_1} \wedge h_2$, where $\overline{h} = \neg h$ is the NOT operation. So,

$$y = f(x_1, x_2) = \overline{(x_1 \wedge x_2)} \wedge (x_1 \vee x_2),$$

which is exactly the XOR function. By generalizing this example, an MLP can represent any logical function.

13.2.2 Differential MLPs Suppose we define the hidden units z_l at each layer l to be a linear transformation of the hidden units at the previous layer passed elementwise through an activation function $\phi : \mathbb{R} \to \mathbb{R}$. Then

$$z_l = f_l(z_{l-1}) = \phi_l(\mathbf{b}_l + \mathbf{W}_l z_{l-1}),$$

or equivalently,

$$z_{kl} = \phi_l \left(b_{kl} + \sum_{j=1}^{K_{l-1}} w_{jkl} z_{jl-1} \right).$$

The quantity that is passed to the activation function is the pre-activations

$$\mathbf{a}_l = \mathbf{b}_l + \mathbf{W}_l z_{l-1},$$

so $x_l = \phi_l(\mathbf{a}_l)$. If L of these functions are composed together, one can compute the gradient of the output using the chain rule. This is also known as backpropagation. Passing the gradient to an optimizer, we can minimize some training objective.

13.2.3 Activation Functions Any differential activation function can be used at each layer, but using a linear activation function reduces the model to a regular linear model, i.e.

$$f(\mathbf{x}; \theta) \propto \mathbf{W}_L \mathbf{W}_{L-1} \cdots \mathbf{W}_1 \mathbf{x} = \mathbf{W}' \mathbf{x},$$

so it is important to use nonlinear activation functions. A few common shoices are the sigmoid (logistic function)

$$\sigma(a) = \frac{1}{1 + e^{-a}}$$

or the tanh function. However, the sigmoid function saturates at 1 for large positive inputs and at 0 for large negative inputs. The tanh function saturates at -1 and +1.

In saturated regions, the gradient of the output with respect to the input will be close to zero, so any gradient signal from higher layers will not be able to propagate back to earlier layers. This makes it hard to train the model using gradient descent.

The most common non-saturating activation function is ReLU,

$$ReLU(a) = \max(a, 0) = a\mathbb{I}(a > 0).$$

13.2.5 The Importance of Depth It can be shown that an MLP with one hidden layer is a universal function approximator, i.e. it can model any suitably smooth function, given enough hidden units. However, experimental and theoretical arguments have shown that deep networks are more effective than shallow ones. Later layers can use features learned by earlier layers, i.e. it's defined in a hierarchical way.

13.2.7 Connection with Biology Consider a model of a single neuron. A neuron k fires, denoted by $h_k \in \{0, 1\}$, depending on the activity of its inputs, denoted by $\mathbf{x} \in \mathbb{R}^D$, as well as the strength of the incoming connections, denoted by $\mathbf{w}_k \in \mathbb{R}^D$. The weighted sum of the inputs is $a_k = \mathbf{w}_k^T \mathbf{x}$, and we can view the weights as "wires'' connecting the inputs x_d to neuron h_k, which are analogous to dendrites in a real neuron. The weighted sum is then compared to a threshold b_k, and if the activation exceeds the threshold, the neuron fires, analogous to a neuron emitting an electrical output or action potential. We can model the behavior of the neuron with $h_k(\mathbf{x}) = H(\mathbf{w}_k^T \mathbf{x} - b_k)$, where $H(a) = \mathbb{I}(a > 0)$ is the Heaviside function. This is the McCulloch-Pitts model of the neuron.

Artificial Neural Networks differ from biological brains in many ways, including:

- Most ANNs use backprop to modify connection strengths, but brains have no way to do this and instead use local update rules for adjusting synaptic strengths

- Most ANNs are strictly feedforward, but real brains have many feedback connections, which may act like priors to be combined with bottom up likelihoods from the sensory system to compute posteriors
- Most ANNs use simple neurons with weighted sums passed through nonlinear functions, but biological neurons have complex dendritic tree structures with complex spatio-temporal dynamics
- Most ANNs are smaller in size and connection numbers than biological brains
- Most ANNs model a single function, but biological brains are complex systems which implement many different kinds of functions

As the tasks we want our machines to solve become harder, we may be able to gain insights from neuroscience and cognitive science for how to solve such tasks in an approximate way.

13.3 Backpropagation The backpropagation algorithm can be used to compute the gradient of a loss function applied to the output of the network with respect to the parameters in each layer. The gradient can then be optimized by another algorithm.

In the case where the computation graph is a simple linear chain of stacked layers, backprop is equivalent to repeated applications of the chain rule in calculus, but it can be generalized to arbitrary directed acyclic graphs, also known as automatic differentiation.

13.3.1 Forward vs Reverse Mode Differentiation Consider a mapping of the form $\mathbf{o} = f(\mathbf{x})$, where $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{o} \in \mathbb{R}^m$. We assume f is defined as a composition of continuous functions:

$$f = f_4 \circ f_3 \circ f_2 \circ f_1$$

where $f_1 : \mathbb{R}^n \to \mathbb{R}^{m_1}$, $f_2 : \mathbb{R}^{m_1} \to \mathbb{R}^{m_2}$, $f_3 : \mathbb{R}^{m_2} \to \mathbb{R}^{m_3}$, and $f_4 : \mathbb{R}^{m_3} \to \mathbb{R}^m$. We can compute the Jacobian $\mathbf{J}_f(\mathbf{x}) = \frac{\partial \mathbf{o}}{\partial \mathbf{x}} \in \mathbb{R}^{m \times n}$ using the chain rule

$$\begin{aligned}\frac{\partial \mathbf{o}}{\partial \mathbf{x}} &= \frac{\partial \mathbf{o}}{\partial \mathbf{x}_4}\frac{\partial \mathbf{x}_4}{\partial \mathbf{x}_3}\frac{\partial \mathbf{x}_3}{\partial \mathbf{x}_2}\frac{\partial \mathbf{x}_2}{\partial \mathbf{x}} = \frac{\partial f_4(\mathbf{x}_4)}{\partial \mathbf{x}_4}\frac{\partial f_3(\mathbf{x}_3)}{\partial \mathbf{x}_3}\frac{\partial f_2(\mathbf{x}_2)}{\partial \mathbf{x}_2}\frac{\partial f_1(\mathbf{x})}{\partial \mathbf{x}} \\ &= \mathbf{J}_{f_4}(\mathbf{x}_4)\mathbf{J}_{f_3}(\mathbf{x}_3)\mathbf{J}_{f_2}(\mathbf{x}_2)\mathbf{J}_{f_1}(\mathbf{x})\end{aligned}$$

Recall that

$$\mathbf{J}_f(\mathbf{x}) = \frac{\partial f(\mathbf{x})}{\partial \mathbf{x}} = \begin{pmatrix} \frac{\partial f_1}{\partial x_1} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \cdots & \frac{\partial f_m}{\partial x_n} \end{pmatrix} = \begin{pmatrix} \nabla f_1(\mathbf{x})^T \\ \vdots \\ \nabla f_m(\mathbf{x})^T \end{pmatrix} = \left(\frac{\partial f}{\partial x_1}, \cdots, \frac{\partial f}{\partial x_n} \right) \in \mathbb{R}^{m \times n},$$

where $\nabla f_i(\mathbf{x})^T \in \mathbb{R}^{1 \times n}$ is the ith row ($i = 1, \ldots, m$) and $\frac{\partial f}{\partial x_j} \in \mathbb{R}^m$ is the jth column ($j = 1, \ldots, n$).

If $n < m$, it's more efficient to compute $\mathbf{J}_f(\mathbf{x})$ for each column $j = 1, \ldots, n$ using forward mode differentiation. Below we show the pseudocode.

$\mathbf{x}_1 := \mathbf{x}$
$\mathbf{v}_j := \mathbf{e}_j \in \mathbb{R}^n$ for $j = 1, \ldots, n$
for $k = 1 : K$ do
    $\mathbf{x}_{k+1} = f_k(\mathbf{x}_k)$
    $\mathbf{v}_j := \mathbf{J}_{f_k}(\mathbf{x}_k)\mathbf{v}_j$ for $j = 1, \ldots, n$
Return $\mathbf{o} = \mathbf{x}_{K+1}, [\mathbf{J}_f(\mathbf{x})]_{:,j} = \mathbf{v}_j$ for $j = 1, \ldots n$

If $n > m$, it's more efficient to compute $\mathbf{J}_f(\mathbf{x})$ for each row $i = 1, \ldots, m$. This can be done using reverse mode differentiation. We show the pseudocode below:

$\mathbf{x}_1 := \mathbf{x}$
for $k = 1 : K$ do
    $\mathbf{x}_{k+1} = f_k(\mathbf{x}_k)$
$\mathbf{u}_i := \mathbf{e}_i \in \mathbb{R}^m$ for $i = 1, \ldots, m$
for $k = K : 1$ do
    $\mathbf{u}_i^T := \mathbf{u}_i^T \mathbf{J}_{f_k}(\mathbf{x}_k)$ for $i = 1, \ldots, m$
Return $\mathbf{o} = \mathbf{x}_{K+1}, [\mathbf{J}_f(\mathbf{x})]_{i,:} = \mathbf{u}_i^T$ for $i = 1, \ldots m$

13.3.2 Reverse Mode Differentiation for Multilayer Perceptrons We now suppose that each layer can have optional parameters $\theta_1, \ldots, \theta_4$, with a mapping of the form $\mathcal{L} : \mathbb{R}^n \to \mathbb{R}$. Consider the ℓ_2 loss for a MLP with one hidden layer:

$$\mathcal{L}((\mathbf{x}, \mathbf{y}), \theta) = \frac{1}{2}||\mathbf{y} - \mathbf{W}_2 \phi(\mathbf{W}_1 \mathbf{x})||_2^2.$$

We can represent this as the following feedforward model:

$$\begin{aligned}
\mathcal{L} &= f_4 \circ f_3 \circ f_2 \circ f_1 \\
\mathbf{x}_2 &= f_1(\mathbf{x}, \theta_1) = \mathbf{W}_1 \mathbf{x} \\
\mathbf{x}_3 &= f_2(\mathbf{x}_2, \emptyset) = \phi(\mathbf{x}_2) \\
\mathbf{x}_4 &= f_3(\mathbf{x}_3, \theta_3) = \mathbf{W}_2 \mathbf{x}_3 \\
\mathcal{L} &= f_4(\mathbf{x}_4, \mathbf{y}) = \frac{1}{2}||\mathbf{x}_4 - \mathbf{y}||^2.
\end{aligned}$$

The notation $f_k(\mathbf{x}_k, \theta_k)$ denotes the function at layer k, where $\mathbf{x}_k$ is the previous output and θ_k are the optional parameters for this layer.

The algorithm for finding the gradient with respect to the parameters in the earlier layers is given by the following pseudocode

// Forward pass
$\mathbf{x}_1 := \mathbf{x}$
for $k = 1 : K$ do
    $\mathbf{x}_{k+1} = f_k(\mathbf{x}_k, \theta_k)$
// Backward pass
$\mathbf{u}_{K+1} := 1$
for $k = K : 1$do
    $\mathbf{g}_k := \mathbf{u}_{k+1}^T \dfrac{\partial f_k(\mathbf{x}_k, \theta_k)}{\partial \theta_k}$
    $\mathbf{u}_k^T := \mathbf{u}_{k+1}^T \dfrac{\partial f_k(\mathbf{x}_k, \theta_k)}{\partial \mathbf{x}_k}$
// Output
Return $\mathcal{L} = \mathbf{x}_{K+1}, \nabla_x \mathcal{L} = \mathbf{u}_1, \{\nabla_{\theta_k} \mathcal{L} = \mathbf{g}_k : k = 1 : K\}$

13.3.4 Computation Graphs Modern DNNs can combine differentiable components in more complex ways than a chain structure, as in MLPs. The resulting computation graph must correspond to a directed acyclic graph, where each node is a differentiable function of all its inputs.

In general,

$$\frac{\partial \mathbf{o}}{\partial \mathbf{x}_j} = \sum_{k \in \text{Ch}(j)} \frac{\partial \mathbf{0}}{\partial \mathbf{x}_k} \frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_j},$$

where the sum is over all children k of node j. The $\frac{\partial \mathbf{o}}{\partial \mathbf{x}_k}$ gradient vector is already computed for each child k, which gets multiplied by the Jacobian $\frac{\partial \mathbf{x}_k}{\partial \mathbf{x}_j}$ of each child.

The computation graph can be computed ahead of time to define a static graph or computed just in time by tracing the execution of the function on an input argument. Dynamic graphs, whose shapes change depending on the values computed by the function, are easier to work with using the second approach.

13.4 Training Neural Networks The standard approach to fit deep neural networks to data is to use maximum likelihood estimation, by minimizing the negative log likelihood:

$$L(\theta) = -\log p(D|\theta) = -\sum_{n=1}^{N} \log p(y_n|x_n;\theta).$$

We can use the back-propagation algorithm to compute the gradient of this loss and pass it on to an pre-made optimizer, but this may not always work well. We should tune the step size learning rate to ensure convergence to a good solution. We can mitigate exploding gradients (where gradients become very large) by incorporating gradient clipping, where we cap the magnitude of the gradient if it becomes too large. However, vanishing gradients can be more difficult to solve: we may modify the activation functions at each layer to prevent gradients from taking extreme values; we may modify the DNN architecture so that updates are additive rather than multiplicative; we may modify the DNN architecture to standardize the activations at each layer; and/or we may carefully choose the initial values of the parameters.

Regarding non-saturating activation functions, the most common in ReLU, the rectified linear unit, defined as $ReLU(a) = \max(a, 0) = aI(a > 0)$, which simply "turns off" negative inputs, passing through positive inputs unchanged. Another is "leaky ReLU" (sometimes called parametric ReLU) $LReLU(a;\alpha) = \max(\alpha a, a)$ where $0 < a < 1$. Other popular choices is "ELU" and "SELU", which have the advantage of being smooth functions. A popular choices for image classification include "swish" (also known as SiLU) and GELU.

One more solution to the vanishing gradient problem for DNNs is using a ResNet (residual network), a feed-forward model where each layer is a residual block $F_l^{'} = F_l(x) + x$. This makes the model easier to train while still having the same number of parameters as a model without residual connections.

We may employ a heuristic initialization scheme to ensure our model doesn't end up with exploding activations or gradiants. If we sample our initialized parameters from standard normal, we can set $\sigma^2 = \frac{2}{n_{in}+n_{out}}$, a technique known as "Xavier initialization" or "Glorot initialization." Xavier/Glorot rae recommended for linear, tanh, logistic, and softmax activation functions. If the number of inputs is equal to the number of outputs, we have the special case of LeCun initialization using the same formula, and this is recommended for SELU. "He initialization" is related to LeCun, but we would use $\sigma^2 = 2/n_{in}$; this is recommended when using ReLU activation functions.

13.5 Regularization We employ regularization to prevent our DNN from overfitting the training data. The simplest method is "early stopping," where we stop the training procedure when the error on the validation set starts to increase. Another is "weight decay," where we impose a prior on the parameters and use MAP estimation, equivalent to l_2 regularization of the objective. A third method is encouraging model sparsity using model compression with l_1 regularization (or ARD). A fourth method is "dropout" where we randomly (per-example) turn off all the outgoing connections from each neuron with probability p. "Dropout can dramatically reduce overfitting and is very widely used." We may even use dropout at test time (called Monte Carlo dropout), resulting in an ensemble of networks, each with slightly different and sparse graph structures. In our effort to prevent overfitting the model to training data, we may use implicit regularization, where we explicitly encourage stochastic gradient descent to find "flat" flexible minima, using either entropy SGD, sharpness aware minimization, stochastic weight averaging (SWA), or other related techniques.

ISLR Chapter 10

Deep learning is currently a very active research area in both the machine learning and artificial intelligence communities. While neural networks gained significant popularity in the 1980s, the advent of SVMs, boosting, and random forests led neural networks to fall from favor; these methods were more automatic and produced greater results than poorly trained neural networks. However, after 2010, neural networks again rose in prominence, often under the name "deep learning," thanks to advancements in techniques, increase in available computing power, and high profile successes in image/video classification, speech, and text modeling. A major part in the increased success of neural network techniques today is the availability of increasingly massive training datasets, thanks to the wide-scale digitization of science and industry.

10.1 Single Layer Neural Networks A neural network takes an input vector of p variables $X = (X_1, X_2, ..., X_p)$ and builds a nonlinear function $f(X)$ to predict the response Y. A visual structure of a single-layer feed-forward neural network includes: 1) A set of p features $X_1, ...X_p$ called the "input layer" 2) Arrows indicating that the inputs from the input layer feed into the K hidden units in the hidden layer. 3) A set of K hidden units (note that we get to choose K) 4) Arrows indicating the hidden layers feeding into the output

The neural network has the form

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k h_k$$

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right)$$

First, the K activations A_k in the hidden layer are computed as functions of the input features. X_p,

$$A_k = h_k(X) = g(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j)$$

Here, $g(z)$ is a nonlinear "activation function" specified in advance. Consider each A_k as a different transformation $h_k(X)$ of the original input features. Then, all of the K hidden layer feed into the output layer, essentially forming a linear regression model.

$$f(X) = \beta_0 + \sum_{k=1}^{K} \beta_k A_k$$

In the day of NNs, we favored the sigmoid $\sigma(a)$ activation function for g, the same function use in logistic regression to generate probabilities (between 0 and 1) from a linear function. In 2020 and beyond, the preferred choice of activation function for g is "ReLU" (rectified linear unit), though there are many other options. ReLU takes the maximum of either the input or 0, essentially transforming all negative inputs into a 0, and passing positive inputs unchanged. Although ReLU thresholds at zero, because we apply it to a linear function, the constant term w_{k0} translates the inflection point left or right on the number line.

It is essential that our activation function be nonlinear; otherwise, $f(X)$ would collapse to a simple linear model $X_1, ..., X_p$. Note that interaction effects can arise from even a sum of two nonlinear transformations, a useful feature.

We must estimate all unknown parameters β_k and w_{kj} to fit a neural network. If we see a qualitative response, we typically use squared-error loss, choosing parameters to minimize $\sum_{i=1}^{n}(y_i - f(x_i))^2$.

10.2 Multilayer Neural Networks Instead of a single layer, modern neural networks typically have multiple hidden layers with many units per layer. While a single layer with a large number of units can approximate most functions, the task of discovering good solutions to complex problems is made much easier with multiple layers, each of modest size.

Our neural networks often have multiple outputs. When our output is a class label, we can represent the various labels with a vector $Y = (Y_0, Y_1, ..., Y_m)$ of m dummy variable, each with a "1" in the position corresponding to the label, and "0" in each other position. This single output selection is called "one-hot encoding.

For a NN of multiple layers, the first hidden layer is defined as it was for a single layer NN. However, the second (and beyond) layers treats the activations A_k^m of the previous hidden layer as inputs, computing new activations. The second layer would be:

$$A_l^{(2)} = h_l^{(2)}(X)$$

$$A_l^{(2)} = g(w_{l0}^{(2)} + \sum_{k=1}^{K_1} w_{lk}^{(2)} A_k^{(1)})$$

Here, we notice that each of the activations in the second layer $A_l^{(2)} = h_l^{(2)}(X)$ is a function of the input vector X. This would also be the case with more hidden layers, since through a chain of transformations, the network builds up fairly complex transformations of X that ultimately feed into the output layer as features.

Notice the use of additional superscript notation, such as $h_l^{(2)}$ and $w_{lj}^{(2)}$, to indicate that the layer the activations and weights (coefficients) belong to layer 2. Also note the matrix notation W_1 represents the entire matrix of weights that feed from the inputs to the first hidden layer L_1, so the subscript indicates which layer the weights are used to feed INTO.

The intercept or "bias" term of each set of weights adds an additional parameter per layer.

At the output layer, the first step is to compute a different linear model (similar to the single layer model above) for each response. For a model with two hidden layers, we would have:

$$Z_m = \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} h_l^{(2)}(X)$$

$$Z_m = \beta_{m0} + \sum_{l=1}^{K_2} \beta_{ml} A_l^{(2)}$$

for each of m outputs. We may use a matrix called B to store all of these final weights. Note that if these were all separate quantitative (not qualitative) responses, we would simply let each $f_m(X) = Z_m$ and be done. However, if we wish our estimates to represent class probabilities $f_m(X) = Pf(Y = m|X)$, then we use a "softmax" activation function

$$f_m(X) = Pr(Y = m|X) = \frac{e^{Z_m}}{\sum_{l=0}^{m} e^{Z_l}}$$

for each of m outputs. The softmax function makes each of m outputs behave like probabilities which are non-negative and together sum to 1. Even though our goal in this example is to classify each input vector, this model actually estimates a probability for each of the classes. Then the classifier assigns the input vector to the class with the greatest probability.

Since this example network's response is qualitative, we search for coefficient estimates that minimize the negative multinomial log-likelihood AKA cross-entropy. If the response were quantitative, we would instead minimize squared-error loss.

10.3 Convolution Neural Networks Convolutional neural networks (CNNs) evolved and gained notoriety for classifying images, though they can show success on a range of problems. CNNs recognize specific patterns or features anywhere in an image that distinguish each particular object class. The recognition starts at the granular level of a square handful of of pixels, then builds toward higher-level features, like outlines or body parts. A convolutional neural network builds up this hierarchy using two specialized types of hidden layers: "convolution layers" and "pooling layers."

A convolution layer is made up of a large number of convolution filters, each determining whether particular local features are present in an image. They rely on "convolution" which is repeatedly multiplying matrix elements and then summing the results. As an example, consider a 4x3 matrix convolved with a 2x2 filter matrix. The convolution filter is applied to every 2x2 submatrix of the original image in order to obtain the convolved image. "If a 2x2 submatrix of the original image"resembles" the convolution filter, then it will have a large value in the convolved image. Note that convolution filters are small $l_1 \times l_2$ arrays, though not necessarily square. A standard practice for image processing is to use predefined filters for image processing. However, with other CNNs, the filters are learned for the specific classification task

In ISLR's image processing CNN example, the 32x32 pixel input images are in color, so the images have three channels: red, green, and blue, each a 32x32 feature map. So, a single convolution filter will also have three channels, though they may have different filer weights. The convolution results from each are summed to form a two-dimensional output feature map. From here, specific color information is no longer passed directly onward. If K different convolution filters are used at the first hidden layer, we get K 2D output feature maps, which can be represented as a single 3D feature map with the third dimension being K. Typically, the ReLU activation function is applied to the convolved image. We may view this as a distinct layer in the CNN, called the "detector layer."

Pooling layers condense a large image into a smaller image. A common method is max pooling, where the single maximum value in each non-overlapping 2×2 block of pixels in an image is used, reducing the image size by a factor of 2^2 while still providing location invariance

"The number of convolution filters in a convolution layer is like the number of units at a particular hidden layer in a fully-connected neural network." This defines the number of channels in resulting 3D feature maps. After the first round of convolutions, we have a feature map with considerably more channels than the three original input channels. To reduce the size of the feature map, we then apply a max-pool layer. Each subsequent convolving layer takes the 3D feature maps from previous layer as input and treats it like a single multi-channel image. Since channel feature maps are reduced in size after each pooling, we typically increase the number of filters in the next convolving layer as a counterbalance. We may repeat several convolving layers before a pool, effectively increasing the dimension of the filter.

We repeat these operations until the pooling has reduced each channel feature map to just a few pixels in each dimension, at which point we say the feature maps are "flattened." Now that the pixel can be treated as separate units, we feed them into one (or more) fully-connected layers before reaching the output layer. We finish the classification with a softmax activation for the classes.

Data augmentation is an important tool for image modeling. Each training image is replicated several times with slight variations or distortions which leave them easily recognizable by humans, such as mirror, shearing, white noise, or cropping. Each transformed image keeps the label of the original. Thus, the training set size is increased considerably, our CNN is somewhat protected against overfitting, and our CNNs is regularized.

Much of the work in fitting a CNN is in learning the convolution filters at the hidden layers as these are like the coefficients of a CNN. For new CNNs, We can use pre-trained hidden layers for new problems with much smaller training sets (we call this weight freezing), and just train the last few layers of the network, requiring much less data.

10.4 Document Classification To study document classification, consider an example of predicting the positive or negative sentiment of an IMBd movie review. We need to "featurize" the document, or define a set of predictors. The "bag-of-words" model is the simplest and most common option here, where the presence or absence of a particular word (from a defined list of common words) is indicated by a 1 or a 0. If we use a 10,000 word dictionary, then each feature vector will be a 10,000 units long, filled with 0 and 1. Since most of the matrix's entries are filled with 0, with only a few percent filled with 1, we call this matrix sparse. Most of the values are the same, so it can be stored efficiently in a spare matrix format.

If we use a two-class neural network, we essentially have a nonlinear logistic regression model with a redundancy in the softmax function, since for K classes, we really only need to estimate $K - 1$ sets of coefficients.

10.5 Recurrent Neural Networks When data sources are sequential, they need special treatment when building predictive models. For example: documents such as books, reviews, articles, and tweets; time series of temperature, rainfall, wind speed, and air quality; financial time series; and recorded speech, music, and other sounds; or handwriting. "In a recurrent neural network (RNN), the input object X is a sequence. The output Y can also be a sequence, but may also be a scalar. A sequence such as $X = X_1, X_2, ..., X_L$ may be the input, where each X_l is a vector, perhaps one-hot encoding for the lth word on a preset language dictionary. A hidden-layer sequence may be $\{A_L\}_1^L = \{A_1, A_2, ..., A_L\}$. Each A_l feeds into the output layer and produces a prediction O_l for YY. The most relevant output is O_L, the last of the output sequence. Note that to process each element in the sequence, the same W, U, and B are used; they are not functions of l. This is a form of"weight sharing" used by RNNs, similar to the use of filters in CNNs. From beginning to end, the activations A_l accumulate a history of what has been seen before, so that the learned context can be used for prediction.

For regression problems, the loss function for an observation (X, Y) is $(Y - O_L)^2$, which interestingly references only the final output O_L. Then, with n input sequence/response pairs (x_i, y_i), the parameters are found by minimizing the sum of squares $\sum_{i=1}^{n}(y_i - o_{iL})^2$. Note that for some learning tasks, the response is also a sequence, so the output sequence $\{O_1, O_2, ..., O_L\}$ is explicitly needed.

We can use RNNs to predict a time series, though we will often find that day-to-day observations are not independent of each other—they may be auto-correlated, where values nearby tend to be similar to each other. For example, we may seek to predict daily measurements log trade volume v_t on the NYSE. Consider pairs of observations (v_t, v_{t-l}), a "lag" of l days apart. "If we take all such pairs in the v_t series and compute their correlation coefficient, this gives the autocorrelation at lag l." Interestingly, in these cases, the response variable is also a predictor, since we use past values of v_t to predict values in the future. We may compare the power of our model against a "straw man", a simple and sensible prediction that can be used as a baseline for comparison.

Section 10.6 When to Use Deep Learning The text shows three way to determine when to use deep learning: 1: Occam's razor principle: when faced with several methods that give roughly equivalent performance, pick the simplest. 2: According to the text, "if we can produce models with the simpler tools that perform as well, they are likely to be easier to fit and understand, and potentially less fragile than the more complex approaches." 3: When the sample size of the data is extremely large and the interpretability of the model is not our main priority, we should use Deep Learning to be our first choice.

Section 10.7 Fitting a Neural Network When fitting a neural netwrok model, the text provide two general strategies: 1. Slow Learning 2. Regularization

10.7.1 Backpropagation Backpropagation is a process that the act of differentiation assigns a fraction of the residual to each of the parameters via the chain rule.

10.7.2 Regularization and Stochastic Gradient Descent Stochastic Gradient Descent is a process that we sample a small fraction of n observations' summation of the derivatives of $R_i(\theta)$ with respect to β_k and w_kj each time we compute a gradient step if n is large. In this process, Regularization is essential to avoid overfitting.

10.7.3 Dropout Learning Dropout learning is a way to randomly remove a fraction ϕ of the units in a layer when fitting the model.

10.7.4 Network Tuning There are three ways provided for network tuning in the text, which are "The number of hidden layers, and the number of units per layer", "The number of hidden layers, and the number of units per layer", and "The number of hidden layers, and the number of units per layer".

Section 10.8 Interpolation and Double Descent Double descent is a phenomenon that the test error has a U-shape before the interpolation threshold is reached, then it descends again as an increasingly flexible model is fit.That may applied to deep learning as well. However, even though double descent can occasionally occur in neural networks model, we do not rely on this behavior typically; meanwhile, it is important to know that the bias-variance trade-off always holds.

2 Reproduction of ISLR Lab 10.9

10.9.1 A Single Layer Network on the Hitters Data

First, We load the Hitters data and split it to training and test set.

```
Gitters <- na.omit(Hitters)
n <- nrow(Gitters)
set.seed(13)
ntest <- trunc(n/3)
testid <- sample(1:n, ntest)
```

Then, we fit a linear model and obtain the mean absolute prediction error, which is 254.6687.

```
lfit <- lm(Salary ~., data=Gitters[-testid,])
lpred <- predict(lfit, Gitters[testid,])
with(Gitters[testid,], mean(abs(lpred - Salary)))
```

Next, we fit a lasso model. Similarly, we calculate the absolute prediction error of the lasso model, which is slightly less than the result of linear model.

```
library(glmnet)
x <- scale(model.matrix(Salary ~. -1, data=Gitters))
y <- Gitters$Salary

cvfit <- cv.glmnet(x[-testid, ], y[-testid], type.measure='mae')
cpred <- predict(cvfit, x[testid,], s='lambda.min')
mean(abs(y[testid] - cpred))
```

Last, we fit the neural network with library keras. As the text written, it is a challenge to get keras running on computer. The error I got is "CondaSSLError: Encountered an SSL error. Most likely a certificate verification issue." Still figure out a way to get this installing.

```
modnn <- keras_model_sequential() %>%
  layer_dense(units=50, activation='relu', input_shape = ncol(x)) %>%
  layer_dropout(rate=0.4) %>%
  layer_dense(units=1)
```

```
modnn %>% compile(loss='mse', optimizer=optimizer_rmsprop(), metrics=list('mean_absolute_error'))
history <- modnn %>% fit(
  x[-testid, ], y[-testid], epochs=200, batch_size=32,
  validation_data=list(x[testid,], y[testid])
)
plot(history, smooth = FALSE)
```

```
npred <- predict(modnn, x[testid,])
mean(abs(y[testid] - npred))
```

10.9.2 A Multilayer Network on the MNIST Digit Data

```
mnist <- dataset_mnist()
x_train <- mnist$train$x
g_train <- mnist$train$y
x_test <- mnist$test$x
g_test <- mnist$test$y
dim(x_train)
```

```
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
y_train <- to_categorical(g_train, 10)
y_test <- to_categorical(g_test, 10)
```

```
x_train <- x_train/255
x_test <- x_test/255
```

```
modelnn <- keras_model_sequential()
modelnn %>%
  layer_dense(units=256, activation="relu", input_shape=c(784)) %>%
  layer_dropout(rate=0.4) %>%
  layer_dense(units=128, activation='relu') %>%
  layer_dropout(rate=0.3) %>%
  layer_dense(units=10, activation='softmax')

#summary(modelnn)
```

```
modelnn %>% compile(loss='categorical_crossentropy', optimizer=optimizer_rmsprop(), metrics=c("accuracy
```

```
system.time(
  history <- modelnn %>%
    fit(x_train, y_train, epochs=30, batch_size=128, validation_split=0.2)
)
```

```
accuracy <- function(pred, truth) {
   mean(drop(as.numeric(pred)) == drop(truth)) }
modelnn %>% predict(x_test) %>% k_argmax() %>% accuracy(g_test)
```

```
modellr <- keras_model_sequential() %>%
   layer_dense(input_shape = 784, units = 10,
       activation = "softmax")
#summary(modellr)
```

```
modellr %>% compile(loss = "categorical_crossentropy",
     optimizer = optimizer_rmsprop(), metrics = c("accuracy"))
modellr %>% fit(x_train, y_train, epochs = 30,
      batch_size = 128, validation_split = 0.2)
modellr %>% predict(x_test) %>% k_argmax() %>% accuracy(g_test)
```

10.9.3 Convolutional Neural Networks

```
cifar100 <- dataset_cifar100()
names(cifar100)
x_train <- cifar100$train$x
g_train <- cifar100$train$y
x_test <- cifar100$test$x
g_test <- cifar100$test$y
```

```
dim(x_train)
range(x_train[1,,, 1])
```

```
x_train <- x_train / 255
x_test <- x_test / 255
y_train <- to_categorical(g_train, 100)
dim(y_train)
```

```
library(jpeg)
par(mar = c(0, 0, 0, 0), mfrow = c(5, 5))
index <- sample(seq(50000), 25)
for (i in index) plot(as.raster(x_train[i,,, ]))
```

```
model <- keras_model_sequential() %>%
   layer_conv_2d(filters = 32, kernel_size = c(3, 3),
      padding = "same", activation = "relu",
      input_shape = c(32, 32, 3)) %>%
   layer_max_pooling_2d(pool_size = c(2, 2)) %>%
   layer_conv_2d(filters = 64, kernel_size = c(3, 3),
      padding = "same", activation = "relu") %>%
   layer_max_pooling_2d(pool_size = c(2, 2)) %>%
   layer_conv_2d(filters = 128, kernel_size = c(3, 3),
      padding = "same", activation = "relu") %>%
   layer_max_pooling_2d(pool_size = c(2, 2)) %>%
   layer_conv_2d(filters = 256, kernel_size = c(3, 3),
      padding = "same", activation = "relu") %>%
   layer_max_pooling_2d(pool_size = c(2, 2)) %>%
   layer_flatten() %>%
   layer_dropout(rate = 0.5) %>%
   layer_dense(units = 512, activation = "relu") %>%
   layer_dense(units = 100, activation = "softmax")
#summary(model)
```

```
model %>% compile(loss = "categorical_crossentropy",
    optimizer = optimizer_rmsprop(), metrics = c("accuracy"))
#history <- model %>% fit(x_train, y_train, epochs = 30,
history <- model %>% fit(x_train, y_train, epochs = 10,
    batch_size = 128, validation_split = 0.2)
model %>% predict(x_test) %>% k_argmax() %>% accuracy(g_test)
```

10.9.4 Using Pretrained CNN Models

```
img_dir <- "book_images"
image_names <- list.files(img_dir)
num_images <- length(image_names)
x <- array(dim = c(num_images, 224, 224, 3))
for (i in 1:num_images) {
   img_path <- paste(img_dir, image_names[i], sep = "/")
   img <- image_load(img_path, target_size = c(224, 224))
   x[i,,, ] <- image_to_array(img)
}
x <- imagenet_preprocess_input(x)
```

```
model <- application_resnet50(weights = "imagenet")
#summary(model)
```

```
pred6 <- model %>% predict(x) %>%
   imagenet_decode_predictions(top = 3)
names(pred6) <- image_names
print(pred6)
```

10.9.5 IMDb Document Classification

```
max_features <- 10000
imdb <- dataset_imdb(num_words = max_features)
c(c(x_train, y_train), c(x_test, y_test)) %<-% imdb
```

```
x_train[[1]][1:12]
```

```
word_index <- dataset_imdb_word_index()
decode_review <- function(text, word_index) {
   word <- names(word_index)
   idx <- unlist(word_index, use.names = FALSE)
   word <- c("<PAD>", "<START>", "<UNK>", "<UNUSED>", word)
   idx <- c(0:3, idx + 3)
   words <- word[match(text, idx, 2)]
   paste(words, collapse = " ")
}
decode_review(x_train[[1]][1:12], word_index)
```

```
library(Matrix)
one_hot <- function(sequences, dimension) {
   seqlen <- sapply(sequences, length)
   n <- length(seqlen)
   rowind <- rep(1:n, seqlen)
   colind <- unlist(sequences)
   sparseMatrix(i = rowind, j = colind,
      dims = c(n, dimension))
}
```

```
x_train_1h <- one_hot(x_train, 10000)
x_test_1h <- one_hot(x_test, 10000)
dim(x_train_1h)
nnzero(x_train_1h) / (25000 * 10000)
```

```
set.seed(3)
ival <- sample(seq(along = y_train), 2000)
```

```
library(glmnet)
fitlm <- glmnet(x_train_1h[-ival, ], y_train[-ival],
    family = "binomial", standardize = FALSE)
classlmv <- predict(fitlm, x_train_1h[ival, ]) > 0
acclmv <- apply(classlmv, 2, accuracy,  y_train[ival] > 0)
```

```
par(mar = c(4, 4, 4, 4), mfrow = c(1, 1))
plot(-log(fitlm$lambda), acclmv)
```

```
model <- keras_model_sequential() %>%
   layer_dense(units = 16, activation = "relu",
      input_shape = c(10000)) %>%
   layer_dense(units = 16, activation = "relu") %>%
   layer_dense(units = 1, activation = "sigmoid")
model %>% compile(optimizer = "rmsprop",
   loss = "binary_crossentropy", metrics = c("accuracy"))
history <- model %>% fit(x_train_1h[-ival, ], y_train[-ival],
   epochs = 20, batch_size = 512,
   validation_data = list(x_train_1h[ival, ], y_train[ival]))
```

```
history <- model %>% fit(
    x_train_1h[-ival, ], y_train[-ival], epochs = 20,
    batch_size = 512, validation_data = list(x_test_1h, y_test)
    )
```

10.9.6 Recurrent Neural Networks

```
wc <- sapply(x_train, length)
median(wc)
sum(wc <= 500) / length(wc)
```

```
maxlen <- 500
x_train <- pad_sequences(x_train, maxlen = maxlen)
x_test <- pad_sequences(x_test, maxlen = maxlen)
dim(x_train)
dim(x_test)
x_train[1, 490:500]
```

```
model <- keras_model_sequential() %>%
   layer_embedding(input_dim = 10000, output_dim = 32) %>%
   layer_lstm(units = 32) %>%
   layer_dense(units = 1, activation = "sigmoid")
```

```
model %>% compile(optimizer = "rmsprop",
    loss = "binary_crossentropy", metrics = c("acc"))
#history <- model %>% fit(x_train, y_train, epochs = 10,
history <- model %>% fit(x_train, y_train, epochs = 3,
    batch_size = 128, validation_data = list(x_test, y_test))
plot(history, smooth = FALSE)
```

Sequential Models for Document Classification

```
xdata <- data.matrix(
 NYSE[, c("DJ_return", "log_volume","log_volatility")]
 )
istrain <- NYSE[, "train"]
xdata <- scale(xdata)
```

```
lagm <- function(x, k = 1) {
   n <- nrow(x)
   pad <- matrix(NA, k, ncol(x))
   rbind(pad, x[1:(n - k), ])
}
```

```
arframe <- data.frame(log_volume = xdata[, "log_volume"],
   L1 = lagm(xdata, 1), L2 = lagm(xdata, 2),
   L3 = lagm(xdata, 3), L4 = lagm(xdata, 4),
   L5 = lagm(xdata, 5)
 )
```

```
arframe <- arframe[-(1:5), ]
istrain <- istrain[-(1:5)]
```

```
arfit <- lm(log_volume ~ ., data = arframe[istrain, ])
arpred <- predict(arfit, arframe[!istrain, ])
V0 <- var(arframe[!istrain, "log_volume"])
1 - mean((arpred - arframe[!istrain, "log_volume"])^2) / V0
```

```
arframed <-
    data.frame(day = NYSE[-(1:5), "day_of_week"], arframe)
arfitd <- lm(log_volume ~ ., data = arframed[istrain, ])
arpredd <- predict(arfitd, arframed[!istrain, ])
1 - mean((arpredd - arframe[!istrain, "log_volume"])^2) / V0
```

```
n <- nrow(arframe)
xrnn <- data.matrix(arframe[, -1])
xrnn <- array(xrnn, c(n, 3, 5))
xrnn <- xrnn[,, 5:1]
xrnn <- aperm(xrnn, c(1, 3, 2))
dim(xrnn)
```

```
model <- keras_model_sequential() %>%
   layer_simple_rnn(units = 12,
      input_shape = list(5, 3),
      dropout = 0.1, recurrent_dropout = 0.1) %>%
   layer_dense(units = 1)
model %>% compile(optimizer = optimizer_rmsprop(),
    loss = "mse")
```

```
history <- model %>% fit(
    xrnn[istrain,, ], arframe[istrain, "log_volume"],
#    batch_size = 64, epochs = 200,
    batch_size = 64, epochs = 75,
    validation_data =
      list(xrnn[!istrain,, ], arframe[!istrain, "log_volume"])
  )
```

```
kpred <- predict(model, xrnn[!istrain,, ])
1 - mean((kpred - arframe[!istrain, "log_volume"])^2) / V0
```

```
model <- keras_model_sequential() %>%
   layer_flatten(input_shape = c(5, 3)) %>%
   layer_dense(units = 1)
```

```
x <- model.matrix(log_volume ~ . - 1, data = arframed)
colnames(x)
```

```
arnnd <- keras_model_sequential() %>%
   layer_dense(units = 32, activation = 'relu',
      input_shape = ncol(x)) %>%
   layer_dropout(rate = 0.5) %>%
   layer_dense(units = 1)
arnnd %>% compile(loss = "mse",
    optimizer = optimizer_rmsprop())
history <- arnnd %>% fit(
#    x[istrain, ], arframe[istrain, "log_volume"], epochs = 100,
    x[istrain, ], arframe[istrain, "log_volume"], epochs = 30,
    batch_size = 32, validation_data =
      list(x[!istrain, ], arframe[!istrain, "log_volume"])
  )
plot(history, smooth = FALSE)

npred <- predict(arnnd, x[!istrain, ])
1 - mean((arframe[!istrain, "log_volume"] - npred)^2) / V0
```

Time Series Prediction

3 ISLR Problems

10.7

We fit a neural network with a single hidden layer and 10 units to the `Default` data, and compare its performance to a logistic regression model. First, we scale our data in a model matrix, excluding the usual first column of 1's for an intercept term, and create a separate response vector of 1's and 0's.

```
x <- model.matrix(default ~ . -1, data = Default) |> scale()
y <- ifelse(Default$default == "Yes",1,0)
```

Next, we partition our data into training and test datasets, training on a random set of 80% of the data, and testing on the remaining 20%.

```
n <- nrow(x)
test_ind <- sample(1:n, n/5)

x_train <- x[-test_ind,]
x_test <- x[test_ind,]
y_train <- y[-test_ind]
y_test <- y[test_ind]
```

Now, we build the model architecture, defining a single layer model with 10 units in the hidden layer, with the usual ReLU activation function and a dropout rate of 20% for regularization. Since we are predicting a

binary response, we use the sigmoid activation function in the output layer. Then, we compile the model, using the binary cross entropy loss function as a measure of fit.

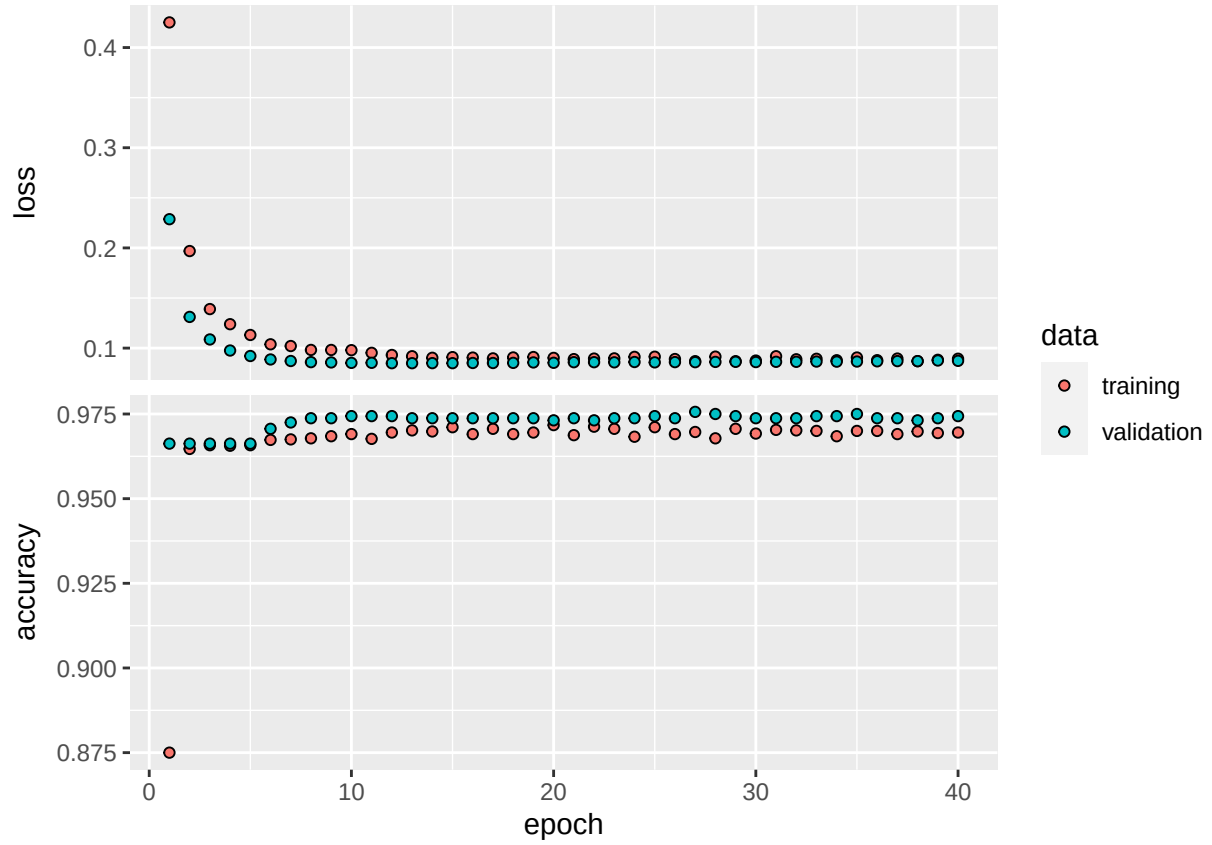
```
nnmodel <- keras_model_sequential() |>
  layer_dense(units = 10, activation = "relu", input_shape = ncol(x)) |>
  layer_dropout(rate = 0.20) |>
  layer_dense(units = 1, activation = "sigmoid")

nnmodel |> compile(
  loss = "binary_crossentropy",
  optimizer = optimizer_rmsprop(),
  metrics = "accuracy")
```

We fit the model using a batch size of 30 with 40 epochs. We also further split the data into a third validation set, again with 20%, so we fit the model on 6400 training observations. Then, each epoch is $6400/30 \approx 214$ stochastic gradient descent steps. We plot the loss of the model alongside its accuracy.

```
history <- nnmodel |>
  fit(x_train, y_train, epochs = 40, batch_size = 30, validation_split = 0.2)

plot(history, smooth = FALSE)
```



Now, we make our predictions on the test set and find the model's misclassification rate.

```
y_test_pred <- nnmodel |> predict(x_test)
NN_misclass_rate <- 1/length(y_test)*sum(y_test != round(y_test_pred))
```

Now, we compare this with a logistic regression model.

```
log_model <- glm(y_train ~ x_train, family = binomial(link = logit))
y_log_test_pred <- round(predict(log_model, newdata = data.frame(x_test), type = "response"))
logistic_misclass_rate <- 1/length(y_test)*sum(y_test != round(y_log_test_pred))
```

The table shows the misclassification rates for both models

```
misclass <- data.frame(Logistic = logistic_misclass_rate, NeuralNet = NN_misclass_rate)
knitr::kable(misclass)
```

Logistic	NeuralNet
0.174	0.024

In the table, we see that the logistic regression model performed considerably worse, with a misclassification rate of about 17.4% versus the neural net's misclassification rate of about 2.4%.

10.8

```
img_dir <- "MyAnimalPics"
image_names <- list.files (img_dir)
num_images <- length (image_names)
x <- array ( dim = c(num_images , 224, 224, 3))
for (i in 1:num_images) {
  img_path <- paste (img_dir , image_names[i], sep = "/")
  img <- image_load (img_path, target_size = c(224, 224))
  x[i,,, ] <- image_to_array(img)
}
x <- imagenet_preprocess_input (x)
```

```
model <- application_resnet50(weights = "imagenet")
#summary (model)
```

```
pred6 <- model %>% predict (x) %>%
  imagenet_decode_predictions (top = 5)
names (pred6) <- image_names
print (pred6)
```

Interestingly, the model does decently well for semi- obvious, clear pictures such as the Stingray, monkey, and dog images. However, it does really poorly when the animal is not obvious such as the coyote, parrot, and fish.