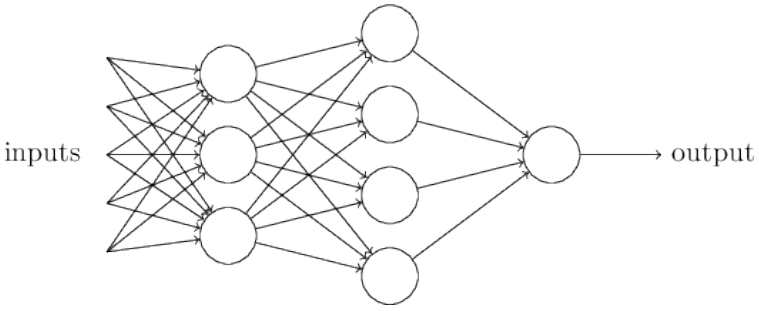
12 Artificial neural networks

Artificial neural networks (ANNs) are behind many of the "AI" successes reported in the media. **Deep learning** is to learn from data with multi-layer neural networks or deep neural networks (DNNs). It is a fast moving area, with new ideas and techniques invented every month. Some topics that are hot today may become out of fashion in 1-2 years.

We will use the online book *Neural Networks and Deep Learning* (http://neuralnetworksanddeeplearning.com) as the textbook. The original code is in Python 2.7; here is a version in Python 3.5

12.1 A simple feedforward neural network

The simplest neural network is a feedforward neural network (FFNN) with fully connected layers. An example is:



In this example, there are 5 input variables, $x_1, \cdots, x_5$, and an output variable, y, and there are two hidden layers between them. Often the input and the output are viewed as layers as well, and the above network is called a 4-layer neural network. The *nodes* are also called *neurons*. Every layer depends **only** on the previously layer. This feedforward nature allows the algorithm of backpropagation to work nicely.

In a **fully connected** layer like the figure above, every node depends on all the nodes in the previous layer, first as a linear combination of the output from the previous layer and then followed by a transformation. For example, node i in layer 2 (the first hidden layer) has an input $z_i^{(2)}$ and an output $a_i^{(2)}$, where $z_i^{(2)} = \sum_{j=1}^5 w_{ij}^{(2)} x_j + b_i^{(2)}$, and

$$a_i^{(2)} = h_i^{(2)}(z_i^{(2)}) = h_i^{(2)} \left[\sum_j w_{ij}^{(2)} x_j + b_i^{(2)} \right].$$

The parameters $w_{ij}^{(2)}$ and $b_i^{(2)}$ are called **weights** and **bias** in the NN literature. The function $h_i^{(2)}()$ is called an **activation function**. When $h(t) = I(t > 0)$, the node is also called a *perceptron*. When $h(t) = \sigma(t) = \frac{1}{1+e^{-t}} = \frac{e^t}{1+e^t}$, the *sigmoid function*, the node is called a *sigmoid neuron*. When the same activation function is used for all the nodes in a layer (which is often the case in practice), the layer is called a *perceptron layer* (when $h(t) = I(t > 0)$) or a *sigmoid layer* (when $h(t) = \sigma(t)$). This simple FFNN is sometimes called a **multilayer perceptron** (MLP) even if none of the nodes is a perceptron.

In the figure above, every node in layer 2 has 6 parameters (5 weights and 1 bias), every node in layer 3 has 4 parameters (3 weights and 1 bias), and output node in the final layer has 5 parameters (4 weights and 1 bias). Thus there are $3 \times 6 + 4 \times 4 + 1 \times 5 = 39$ parameters. If layers 2–4 are all sigmoid layers, the model is

$$\begin{aligned}
f(x_1, \ldots, x_5) &= \sigma \left[w_1^{(4)} a_1^{(3)} + w_2^{(4)} a_2^{(3)} + w_3^{(4)} a_3^{(3)} + w_4^{(4)} a_4^{(3)} + b^{(4)} \right], \text{ where} \\
a_k^{(3)} &= \sigma \left[w_{k1}^{(3)} a_1^{(2)} + w_{k2}^{(3)} a_2^{(2)} + w_{k3}^{(3)} a_3^{(2)} + b_k^{(3)} \right] \quad (k = 1, 2, 3, 4) \\
a_k^{(2)} &= \sigma \left[w_{k1}^{(2)} x_1 + w_{k2}^{(2)} x_2 + w_{k3}^{(2)} x_3 + w_{k4}^{(2)} x_4 + w_{k5}^{(2)} x_5 + b_k^{(2)} \right] \quad (k = 1, 2, 3).
\end{aligned}$$

To fit the model, we estimate these 39 parameters under an optimization criterion (e.g., minimizing a total cost). The model can quickly become very complex with additional nodes or layers.

If the outcome is continuous, a commonly used cost function is the squared error loss, for which the total cost is $C = \sum_i (y_i - f(x_{i1}, \ldots, x_{ip}))^2$. We do not have a closed-form solution and have to rely on numerical algorithms to minimize C. When the sample size is large, it can be computationally expensive to use 2nd-order approximations such as the Newton–Raphson algorithm. In ANN, we often rely on the 1st-order approximation algorithm called the **gradient descent** or its stochastic version, the **stochastic gradient descent** (details below).

12.2 Tensorflow and Keras

A **tensor** is another name for an array. Arrays are a basic data type in programming. A matrix is effectively a 2-dimensional array. A vector is effectively a 1-dimensional array. A single number (sometimes called a *scalar*) is effectively a 0-dimentional array, although most computer languages do not treat a scalar as an array. All these can be viewed as tensors.

The **rank** of a tensor is the number of dimensions, and the **shape** of a tensor is the dimensions. For example, a rank 2 tensor with shape $[3, 5]$ is effectively a 3×5 matrix; a rank 0 tensor is a scalar and it has an empty shape; a rank 1 tensor with shape $[8]$ is effectively a vector of length 8. In TensorFlow we may see a rank 4 tensor with shape $[1, 1, 100, 1]$, which is just a vector of length 100 masqueraded as a 4-dimensional array. Below I use the `array()` function in R to show what arrays look like:

```
array(1:6, c(2,3))  ## same as matrix(1:6, c(2,3))
array(1:6, c(6))
array(1:18, c(2,3,3))
array(1:6, c(1,1,6,1))
```

In ANNs, operations can be formulated as a flow of tensors. For example, in the FFNN model above, the flow is from a tensor of length 5 to a tensor of length 3, and then to a tensor of length 4, and then to a tensor of length 1.

TensorFlow is a neural network package from Google. It has some competitors out there: **PyTorch** and **Caffe2** from Facebook, **MXNet** from Amazon, **CNTK** from Microsoft, **H2O**, etc.

Keras is a front end for TensorFlow. It uses simpler syntax when fitting standard neural network models. It is available in Python and R (https://keras.rstudio.com/). To install the Python version in Anaconda, use `conda install keras`. The installation of the R version has two steps:

```
install.packages("keras")      ## install a few necessary R packages first
keras::install_keras("conda")  ## install necessary conda/tensorflow packages outside R
```

Notes: (1) The second step is not needed if you have installed TF packages in anaconda. (2) If you use the second step, `install_keras("conda")`, it first installs a few necessary conda packages, then creates a conda environment and installs TF packages into the environment. In Windows, Anaconda 3.x is required. (3) When Keras is loaded with `library(keras)`, by default the backend is "tensorflow" and the implementation is "keras". Use `use_backend()` and `use_implementation()` to change them if needed. All backend API functions have a `k_` prefix. In Linux, the configuration file is `$HOME/.keras/keras.json`, and all example datasets downloaded with `datasets_*()` are in `$HOME/.keras/datasets/`.

We now define the model in the last section. The order of the statements is very important. Note the use of pipes with `%>%`. The function `layer_dense()` is to specify a fully-connected layer.

```
library(keras)

model = keras_model_sequential()    ## initialize the keras model object
model %>%                           ## define the layers
  layer_dense(units=3, activation="sigmoid", input_shape=c(5)) %>%
  layer_dense(units=4, activation="sigmoid") %>%
  layer_dense(units=1, activation="sigmoid")

summary(model)                    ## summary of model structure
```

The equivalent Python code is

```
from keras.models import Sequential
from keras.layers import Dense, Dropout

model = Sequential()
model.add(Dense(3, activation='sigmoid', input_shape=(5,)))
model.add(Dense(4, activation='sigmoid'))
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

12.3 Commonly used activation functions

Sigmoid: $\sigma(x) = (1 + e^{-x})^{-1} \in (0, 1)$

- $\sigma'(x) = \sigma(x)(1 - \sigma(x))$. It was a popular choice a few years ago.

Hyperbolic tangent: $\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1} = \frac{e^x - e^{-x}}{e^x + e^{-x}} = 2\sigma(2x) - 1 \in (-1, 1)$

- $\tanh'(x) = (1 + \tanh(x))(1 - \tanh(x))$.

ReLU (rectified linear unit): $h(x) = x_+ = \max\{0, x\}$

- It is a popular choice nowadays.

Leaky ReLU: $h(x) = x_+ + ax_-$, where $x_- = \min\{0, x\}$ and $a > 0$ but a is very close to zero

Perceptron: $h(x) = I(x \geq 0) \in \{0, 1\}$

Identity: $h(x) = x$. The corresponding neuron is called a *linear neuron*.

A **softmax layer** is often used for the output layer when the outcome has multiple categories. For example, for the MNIST dataset, the output has $K = 10$ categories. The activation function for the output layer is a softmax function

$$(z_1, \cdots, z_K) \to (a_1, \ldots, a_K) = \left(\frac{\exp(z_1)}{\sum_k \exp(z_k)}, \cdots, \frac{\exp(z_K)}{\sum_k \exp(z_k)} \right),$$

where $(z_1, \cdots, z_K)$ are the input to the K nodes and they are linear combinations of the output from the previous layer. This definition ensures that final output $(a_1, \ldots, a_K)$ is a multinomial probability distribution with $\sum_k a_k = 1$.

12.4 Commonly used cost functions

Quadratic cost (squared error): $C(y, a) = \frac{1}{2}\|y - a\|^2$

- $\nabla_a C = \frac{\partial C}{\partial a} = a - y$

Cross-entropy: $C(y, a) = -[y \ln a + (1 - y) \ln(1 - a)]$ for $0 \leq y \leq 1$ and $0 < a < 1$

- $\nabla_a C = \frac{a-y}{a(1-a)}$

Cross-entropy for multinomial outcomes: $C(y, a) = -\ln(y'a)$, where y is a vector with one element 1 and all others 0, and a is a multinomial probability distribution. This is the negative multinomial log-likelihood.

The *average cost* over n observations is $C = \frac{1}{n}\sum_i C(y_i, a_i)$, where y_i is the outcome for observation i and a_i is a predicted value.

Ideally, the cost function should be chosen to reflect the real cost. But often a mathematically convenient cost is chosen. For example, for MNIST data, classification accuracy is the ultimate goal, but often the softmax output layer coupled with the negative log-likelihood cost is used to avoid slow learning at the output layer. It is possible that while the cost for the training set is being driven down, the cost for the test data can be ascending but the test data classification accuracy is being improved (an example is in NNDL Chapter 3).

12.5 Gradient descent

Gradient descent (GD) is an iterative algorithm for minimizing the total cost $C = C(\theta_1, \ldots, \theta_p)$. The **gradient** of the function is a vector of partial derivatives $\nabla C = (\frac{\partial C}{\partial \theta_1}, \ldots, \frac{\partial C}{\partial \theta_p})^T$. The algorithm is:

(1) Initialize θ.
(2) At every iteration, calculate ∇C with the current θ estimates, and update $\theta \leftarrow \theta - \eta \nabla C$, where $\eta > 0$ is the **learning rate** (which should be small).

The **rationale** for the GD algorithm: At every step, we seek to identify a small change in θ, $\Delta\theta$, so that the reduction from $C(\theta)$ to $C(\theta + \Delta\theta)$ is the greatest. Since $C(\theta + \Delta\theta) - C(\theta) \approx (\nabla C)^T \Delta\theta$, the fastest negative change occurs when $\Delta\theta$ has the same direction as $-\nabla C$; that is, to *descend along the gradient*.

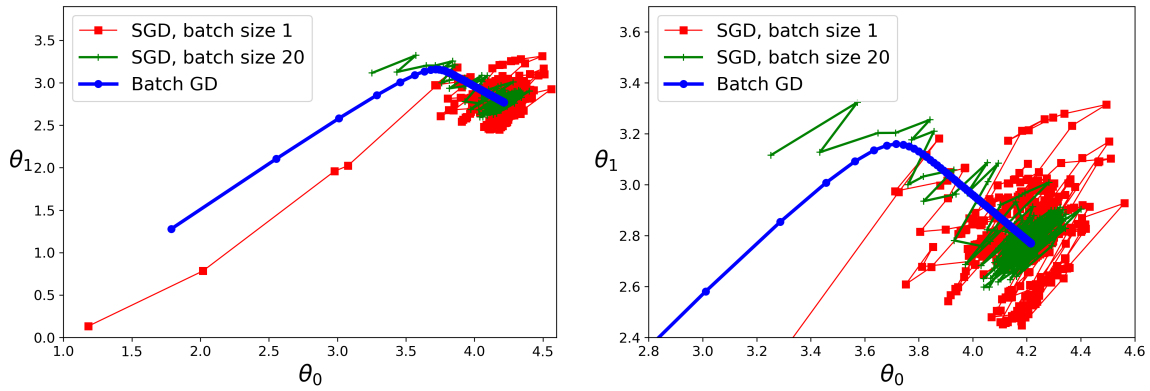
When n is not large, we can calculate ∇C using all observations. This is called **batch gradient descent**.

Stochastic gradient descent (SGD): When n is very large, the calculation of ∇C can be slow. We can estimate ∇C using a subset of the data. The size of the subset is called the **mini-batch size**. In the SGD, the training set is randomly partitioned into subsets of the mini-batch size, and we update θ by iterating through all the subsets. After going through all the subsets, we have finished one **epoch**; that is, we have gone through the whole training set once. Then we repartition the training set and repeat the process. It is common to run over 100 epochs in deep learning tasks.

Notes:

- The smaller the mini-batch size is, the more iterations will be performed to go through an epoch, and the smaller learning rate should be used.
- When the mini-batch size is 1, the algorithm is an **online** algorithm.
- Using a very small mini-batch size can be erratic, although it may give you a chance to jump out of a local minimum.
- Some authors call the SGD with mini-batch size 1 the "stochastic gradient descent", and the SGD with mini-batch size >1 the "mini-batch gradient descent". This distinction is unnecessary.

A comparison of the effects of batch size (HOML Figure 4-11) when the SGD algorithm is used to fit a simple linear model $y = \theta_0 + \theta_1 x + \epsilon$.



Notes on the **learning rate**:

- A too high learning rate can lead to divergence due to "overshooting" from one iteration to the next.
- A too low learning rate can make the training very slow.
- Gradient descent benefits from a small learning rate. As a first-order approximation, it may not approximate well with a large learning rate, at which higher-order terms become important.
- A **learning schedule** is a scheme to gradually decrease the learning rate as we progress instead of using a constant learning rate. (This is similar to simulated annealing.)
- The learning rate is not part of the model. It is part of the model fitting procedure.
- One way of setting the learning rate: Try η values with various magnitude to find the largest value of η at which the cost decreases during the first few epochs, then set the η to be half of it.
- Some variant algorithms (e.g., *RMSprop*) can set the learning rate adaptively.

Hyperparameters: number of epochs, mini-batch size, learning rate (or parameters in a learning schedule)

Technical notes on GD:

- GD is a first-order approximation. In contrast, the Hessian approach is a second-order approximation (e.g., Newton–Raphson and Gauss–Newton methods).
- GD cannot guarantee to converge to the global minimum of $C(u)$. Global minimum is guaranteed when C is convex and ∇C is Lipschitz.

12.6 A simple example

Let us try a simple FFNN on the MNIST dataset. We prepare the data first.

```
library(keras)
c(c(x_train, y_train), c(x_test, y_test)) %<-% dataset_mnist()

dim(x_train); dim(x_test)    ## 60000 x 28 x 28; 10000 x 28 x 28
dim(x_train) = c(nrow(x_train), 784)  ## reshape/flattening X to have 784 columns
dim(x_test) = c(nrow(x_test), 784)

range(x_train); range(x_test)    ## the raw data has range 0-255
x_train = x_train / 255  ## rescale it so that its range is 0-1
x_test = x_test / 255

y_train10 = to_categorical(y_train, 10)  ## make dummy (one-hot) variables for the outcome
y_test10 = to_categorical(y_test, 10)
head(y_train); head(y_train10)   ## check
```

We now define an FFNN model, which has an input layer with 784 features, a single hidden sigmoid layer with 30 nodes, and a softmax final layer with 10 nodes. The model has 23,860 parameters!

```
use_session_with_seed(2018)    ## one should not set seed in practice
model1 = keras_model_sequential() %>%
  layer_dense(units = 30, activation = "sigmoid", input_shape = c(784)) %>%
  layer_dense(units = 10, activation = "softmax")

summary(model1)
```

Note that one should not set the seed in practice because `use_session_with_seed()` "disables GPU computations and CPU parallelization by default (as both can lead to non-deterministic computations)". (explanation is here)

We then set the options for model fitting, and then fit it!

```
model1 %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_sgd(lr = 3),    ## the SGD optimizer
  #optimizer = optimizer_rmsprop(),     ## the RMSprop optimizer
  metrics = c("accuracy")
)

history1 = model1 %>% fit(
  x_train, y_train10,
  epochs = 30, batch_size = 100, verbose = 1,
  validation_data = list(x_test, y_test10)
  #validation_split = 0.2    ## alternative way to specify validation set
)
```

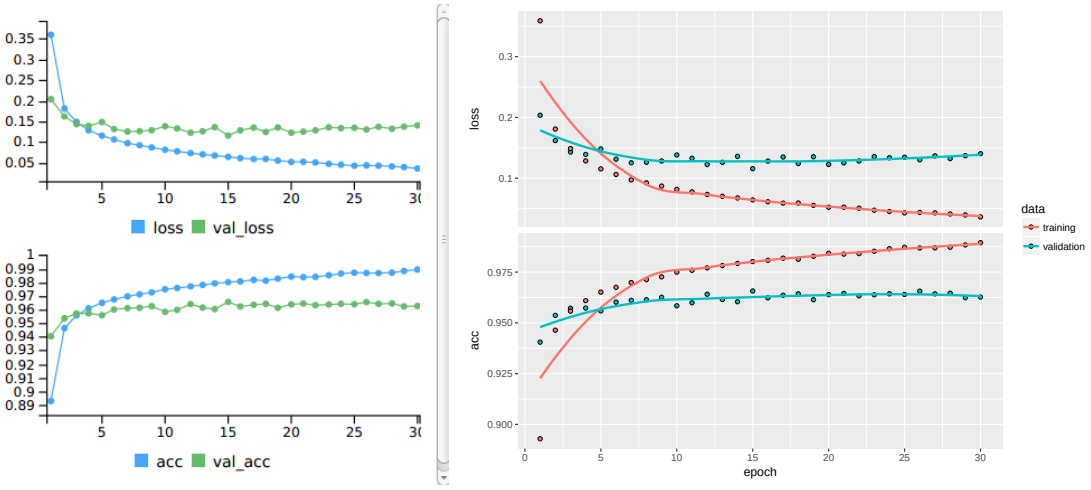
Note about `validation_split=0.2`: When there is no validation or test set, you can use this option to split the training set so that 80% would be used for training and 20% for validation.

The object `history1` contains some hyperparameter values for fitting and some performance metrics.

```
str(history1)              ## list what it contains
history1$metrics$val_loss  ## average loss over the validation data
```

The plot below on the left is generated during fitting. A nicer version can be drawn using `plot()`.

```
plot(history1)  ## the plot on the right
```



After about 8 epochs, the test set performance becomes plateaued while the training set performance continues to "improve". This is a sign that the model training may stop here. Beyond this point the model may become overfitting.

Here, the learning rate 3 may be too high. Re-training the model with a smaller learning rate gives a different plot. We can also re-train the model using the RMSprop optimizer.

To evaluate the model on the test set, use `evaluate()` or `predict_classes()`:

```
model1 %>% evaluate(x_test, y_test10, verbose = 0)

y_pred = model1 %>% predict_classes(x_test)
table(y_test, y_pred)      ## a 10 x 10 "confusion matrix"
sum(diag(table(y_test, y_pred)))
```

The function `predict_proba()` gives the softmax probabilities. The class with the highest probability is the predicted class.

```
y_predprob = model1 %>% predict_proba(x_test)
dim(y_predprob)  ## 10000 x 10 matrix
all.equal(apply(y_predprob, 1, which.max)-1, as.numeric(y_pred))  ## True
```

Notes:

- The Python code from the NNDL book is a straightforward Python implementation of the backpropagation algorithm for FFNN. It is not only faster than the keras/TF implementation above but also more robust.
- The change in val_loss was very small when `optimizer_rmsprop` was used, but a little large when `optimizer_sgd` was used.

The model above can be expressed using matrices:

$$\boldsymbol{z}^{(2)} = W^{(2)}\boldsymbol{x} + b^{(2)}, \qquad \boldsymbol{a}^{(2)} = \sigma(\boldsymbol{z}^{(2)}),$$
$$\boldsymbol{z}^{(3)} = W^{(3)}\boldsymbol{a}^{(2)} + b^{(3)}, \qquad \boldsymbol{a}^{(3)} = \text{softmax}(\boldsymbol{z}^{(3)}),$$

where $W^{(2)}$ is a 30×784 matrix, $b^{(2)}$ is a 30-vector, $W^{(3)}$ is a 10×30 matrix, and $b^{(3)}$ is a 10-vector. To extract the estimates of these parameters, use `get_weights()`. The result is a list.

```
wts = model %>% get_weights
str(wts)
```

```
List of 4
 $ : num [1:784, 1:30] -0.04114 0.01509 0.032 0.05706 -0.00359 ...
 $ : num [1:30(1d)] 1.1521 0.8981 -0.6763 0.0102 -2.4692 ...
 $ : num [1:30, 1:10] -2.14 -1.34 -1.27 -0.54 -3.21 ...
 $ : num [1:10(1d)] -1.892 -1.337 -2.138 4.669 -0.729 ...
```

Note that the first element in the list is a 784×30 matrix and the third element is a 30×10 matrix.

When there are many layers, one can extract the parameters for a particular layer by using `get_layer()` first and then `get_weights()`. For example, to get the parameters for the first hidden layer (layer 2),

```
wts2 = model %>% get_layer(index=2) %>% get_weights
str(wts2)
```

```
List of 2
 $ : num [1:784, 1:30] -0.04114 0.01509 0.032 0.05706 -0.00359 ...
 $ : num [1:30(1d)] 1.1521 0.8981 -0.6763 0.0102 -2.4692 ...
```

12.7 Multinomial logistic regression

When there is no hidden layer, the above FFNN becomes multinomial logistic regression!

```
use_session_with_seed(2018)
mlogit = keras_model_sequential() %>%
  layer_dense(units = 10, activation = "softmax", input_shape = c(784))

mlogit %>% compile(
  optimizer = optimizer_rmsprop(),
  loss = "categorical_crossentropy",
  metrics = c("accuracy")
)

mlogithistory = mlogit %>% fit(
  x_train, y_train10,
  epochs = 30, batch_size = 100, verbose = 1,
  validation_data = list(x_test, y_test10)
)
```

The classical multinomial logistic regression can achieve over 92% test set classification accuracy rate.

```
mlogithistory$metrics$val_acc
 [1] 0.9099 0.9182 0.9215 0.9229 0.9248 0.9262 0.9248 0.9261 0.9256 0.9278
[11] 0.9280 0.9265 0.9264 0.9274 0.9274 0.9266 0.9277 0.9272 0.9276 0.9273
[21] 0.9278 0.9260 0.9272 0.9273 0.9265 0.9279 0.9279 0.9280 0.9270 0.9282
mlogithistory$metrics$val_loss
plot(mlogithistory)
```

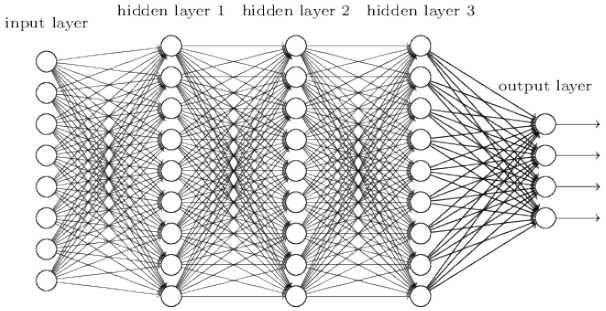
This cannot be performed using the R `nnet` package due to "too many" parameters.

```
library(nnet)
mod = multinom(y_train ~ x_train)
```

```
Error in nnet.default(X, Y, w, mask = mask, size = 0, skip = TRUE, softmax = TRUE,  :
  too many (7860) weights
```

12.8 Backpropagation to compute gradients

Consider a FFNN such as the one in this figure. The layers are numbered from 1 to L. In this figure, $L = 5$.



Let a^l be the vector of outputs from layer l. Then $a^1 = (x_1, \ldots, x_p)^T$ are the inputs of the network, and a^L are the outputs of the network. For $l = 2, \ldots, L$, let z^l be the vector of inputs into layer l ($l = 2, \ldots, L$). Then

$$z^l = g^l(a^{l-1}) = w^l a^{l-1} + b^l, \quad a^l = h^l(z^l),$$

where h^l is the activation function of layer l. When the activation is node specific, an element-wise representation is $a_j^l = h_j^l(\sum_k w_{jk}^l a_k^{l-1} + b_j^l)$, where superscripts denote the layer and subscripts denote the node within the layer.

Let $\theta^l = (w^l, b^l)$ denote all the parameters between layers $l-1$ and l. Then $z^l = g^l(a^{l-1})$ can be written as $z^l = g_{\theta^l}(a^{l-1})$. The NN model is a "well-specified" function

$$a^L = f(x) = h^L(g_{\theta^L}(h^{L-1}(g_{\theta^{L-1}}(\cdots(h^3(g_{\theta^3}(h^2(g_{\theta^2}(x)))))\cdots)))),$$

with unknown parameters $(\theta^2, \ldots, \theta^L)$. The cost for an observation (x, y) is $C(y, f(x))$.

We use a **backpropagation algorithm** to compute the gradients of the cost $C(y, a) = C(y, f(x))$ for a data point (x, y) under the current parameter estimates. It has two main steps:

(1) *Forward propagation* to calculate all the z^l and a^l;
(2) *Backward propagation* to calculate all the gradients $\frac{\partial C}{\partial \theta^l}$. Note that $\frac{\partial C}{\partial \theta^l} = \frac{\partial C}{\partial f(x)} \frac{\partial f(x)}{\partial \theta^l}$.

We briefly describe the second step below. For the gradient of $f(x)$ with respect to θ^L, we only need to consider $f(x) = h^L(g_{\theta^L}(a^{L-1})) = h^L(z^L)$, where the input of the function is a^{L-1}. By the chain rule, $\frac{\partial f}{\partial \theta^L} = \frac{\partial f}{\partial z^L} \frac{\partial z^L}{\partial \theta^L} = \frac{\partial h^L(z^L)}{\partial z^L} \frac{\partial g(a^{L-1})}{\partial \theta^L}$. Let $\delta^L = \frac{\partial h^L(z^L)}{\partial z^L}$; it does not depend on θ^L. For the second term $\frac{\partial g(a^{L-1})}{\partial \theta^L}$, its elements are $\frac{\partial g(a^{L-1})}{\partial w_{jk}^L} = a_k^{L-1}$ and $\frac{\partial g(a^{L-1})}{\partial b_j^L} = 1$. These gradients can be expressed as a vector and a matrix:

$$\frac{\partial C}{\partial b^L} = \delta^L, \quad \frac{\partial C}{\partial w^L} = \delta^L (a^{L-1})^T.$$

Similarly, for the gradient of $f(x)$ with respect to θ^{L-1}, we only need to consider $f(x) = u(g_{\theta^{L-1}}(a^{L-2})) = u(z^{L-1})$, where $u() = h^L(g_{\theta^L}(h^{L-1}()))$. We use the chain rule again to obtain $\frac{\partial f}{\partial \theta^{L-1}} = \frac{\partial f}{\partial z^{L-1}} \frac{\partial z^{L-1}}{\partial \theta^{L-1}} = \frac{\partial u(z^{L-1})}{\partial z^{L-1}} \frac{\partial g(a^{L-2})}{\partial \theta^{L-1}}$. Let $\delta^{L-1} = \frac{\partial u(z^{L-1})}{\partial z^{L-1}}$; it does not depend on θ^{L-1} and can be computed with the chain rule. The second term is similar as above, $\frac{\partial g(a^{L-2})}{\partial w_{jk}^{L-1}} = a_k^{L-2}$ and $\frac{\partial g(a^{L-2})}{\partial b_j^{L-1}} = 1$. We continue doing this until the first layer.

Let $\nabla_a C = \frac{\partial C}{\partial f(x)}$ and $\Sigma^l = \frac{\partial a^l}{\partial z^l} = \{\frac{\partial a_i^l}{\partial z_j^l}\}$ be the Jacobian matrix at layer l. Then the whole algorithm can be expressed as

$$\begin{cases} \delta^L = \frac{\partial C}{\partial z^L} & = (\Sigma^L)^T \cdot \nabla_a C & & \text{(BP1);} \\ \delta^l = \frac{\partial C}{\partial z^l} & = (\Sigma^l)^T (w^{l+1})^T \delta^{l+1}, & 2 \le l \le L-1 & \text{(BP2);} \\ \frac{\partial C}{\partial b^l} & = \delta^l, & 2 \le l \le L & \text{(BP3);} \\ \frac{\partial C}{\partial w^l} & = \delta^l (a^{l-1})^T, & 2 \le l \le L & \text{(BP4).} \end{cases}$$

When the activation functions are element-wise one-to-one (unlike softmax), Σ^l is a diagonal matrix, and the first two equations can be simplified to element-wise products of vectors: $\delta^L = \nabla_a C \circ h^L(z^L)$ and $\delta^l = ((w^{l+1})^T \delta^{l+1}) \circ h^l(z^l)$.

With this matrix representation, the implementation of the whole algorithm is straightforward and surprisingly short, as shown in the Python code written by Michael Nielsen (the author of NNDL).

The above algorithm is for a single observation. Over a mini-batch of m observations, one could loop through the algorithm m times, or do them simultaneously in arrays. Nielsen implemented the latter, which takes half time to run compared to the former.

Challenges: There are several challenges in fitting NN models.

(1) How do we avoid vanishing gradients?
(2) How do we ensure the algorithm converges to the minimum, does not stop prematurely and is not too slow?
(3) How do we avoid overfitting given there are so many parameters in a model?
(4) How do we select the hyperparameters to obtain an optimal model?

These will be addressed in the next several sections.

12.9 Vanishing gradient problem

Some gradients can become so small that their corresponding parameters stop learning. This can be due to very small δ^l or very small a^{l-1}. In a multi-layer NN, the first few layers can have very small δ^l.

To address this problem, people has used a combination of the following approaches:

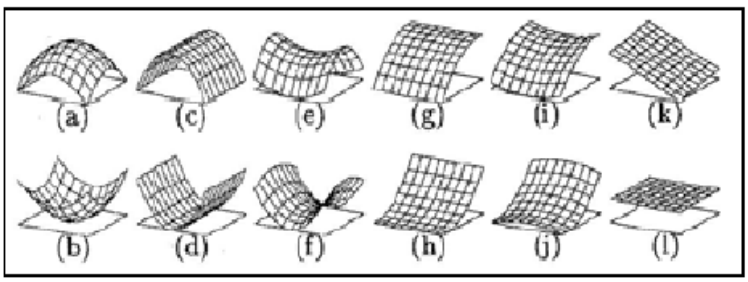
(1) Choose the cost function and activation function so that they pair well (see NNDL);
(2) Use variance scaling when initializing the weights;
(3) Use residual layers (e.g., Microsoft's ResNet, 2015).

Initiation of parameters: For a sigmoid neuron with n_{in} inputs, where n_{in} is large (say, 1000), initiating $w, b \sim N(0,1)$ will have a high chance to have a saturated $\sigma(z)$ (i.e., $\sigma(z)$ is very close to 0 or 1, and $\sigma'(z) \approx 0$) and thus slow learning. Initiating $w \sim N(0, 1/n_{\text{in}})$ will be much better; this is the LeCun normal initializer. Below are some popular initializer choices:

- Glorot uniform initializer (Xavier Glorot and Yoshua Bengio, 2010): $w \sim$ uniform distribution $U(-a_G, a_G)$, where $a_G = \sqrt{6/(n_{\text{in}} + n_{\text{out}})}$, n_{in} and n_{out} are the numbers of input and output units for the weight matrix between two layers.
- Glorot normal initializer: $w \sim$ truncated normal distribution with mean 0 and SD $= \sqrt{2/(n_{\text{in}} + n_{\text{out}})}$.
- He uniform initializer (He et al., 2015): $w \sim U(-a_H, a_H)$, where $a_H = \sqrt{6/n_{\text{in}}}$.
- He normal initializer: $w \sim$ truncated normal distribution with mean 0 and SD $= \sqrt{2/n_{\text{in}}}$.
- LeCun uniform initializer: $w \sim U(-a_L, a_L)$, where $a_L = \sqrt{3/n_{\text{in}}}$.
- LeCun normal initializer: $w \sim$ truncated normal distribution with mean 0 and SD $= \sqrt{1/n_{\text{in}}}$.

12.10 Improving gradient descent

The high dimensional cost function can have "pits", "ravines", and "saddles".



(a) peak, (b) pit, (c) ridge, (d) ravine, (e) ridge saddle, (f) ravine saddle, (g) convex hill, (h) concave hill, (i) convex saddle hill, (j) concave saddle hill, (k) slop hill, (l) flat (figure from Ferdowsi and Ahmadyfard, 2008)

Standard GD: At iteration k, we update the parameter estimate $\theta_k = \theta_{k-1} - \eta_k g_k$, where η_k is the learning rate and $g_k = \nabla C(\theta_{k-1})$ is the gradient at θ_{k-1}. The amount of update is $-\eta_k g_k$. To improve the performance of GD, people have tried many treaks. They fall into the following types:

(1) Use a learning rate schedule to gradually decrease the learning rate;
(2) Introduce a momentum;
(3) Compute the learning rate adaptively and allowing them to differ across parameters;
(4) Compute both the learning rate and the gradient adaptively.

Below are a few popular choices. Alec Radford generated a few GIFs to show a comparison of these approaches (Adam not included).

1. A **learning rate schedule** is a predetermined scheme to gradually decrease the learning rate instead of using a constant learning rate. (This is similar to simulated annealing.)

In `optimizer_sgd(lr=0.01, momentum=0.0, decay=0.0, nesterov=False)`, `lr` is the learning rate η, `decay` is a value d such that the learning rate at iteration k is $\eta_k = \eta/(1+dk)$. By default, `decay=0` and $\eta_k = \eta$ for all k.

2. **Momentum-based GD**: Define a **velocity** vector v_k (initialize with $v_0 = 0$). At every iteration, we compute

$$v_k = \mu v_{k-1} - \eta_k g_k, \text{ and then } \theta_k = \theta_{k-1} + v_k.$$

Here, v_k is the standard GD, $-\eta_k g_k$, plus the amount of momentum from the previous iteration, μv_{k-1}. The hyper-parameter $0 \le \mu \le 1$ is called the **momentum** and $1-\mu$ is the **friction**. When $\mu = 0$, we have no momentum buildup and this reduces to the standard GD. When μ is close to 1, the momentum can build up quickly.

- **Nesterov's momentum** (Nesterov's accelerated gradient, or NAG): In the above approach, the gradient $g_k = \nabla C(\theta_{k-1})$ is the calculated at the current position θ_{k-1}. In Nesterov's approach, the gradients are calculated at the "lookahead" position $\theta_{k-1} + \mu v_{k-1}$ to obtain a better assessment of the gradient. That is, $v_k = \mu v_{k-1} - \eta_k \nabla C(\theta_{k-1} + \mu v_{k-1})$. It significantly improved the performance of some RNN tasks.

Bengio et al. showed an algorithm that is equivalent to Nesterov's approach but does not require explicit computation of gradients at the lookahead position. It is implemented in `optimizers_sgd()`. The Bengio algorithm is

(1) $v_0 = 0$
(2) In iteration k, compute η_k, $g_k = \nabla C(\theta_{k-1})$, and $v_k = \mu v_{k-1} - \eta_k g_k$. Then
  (a) If `nesterov = False`, $\theta_k = \theta_{k-1} + v_k$.
  (b) If `nesterov = True`, $\theta_k = \theta_{k-1} + \mu v_k - \eta_k g_k$.

3. Compute the learning rate adaptively and allowing them to differ across parameters.

- **Adagrad**: Adaptively compute the learning rate for each parameter instead of using the same value for all parameters. The j-th parameter at iteration k is

$$\theta_{k,j} = \theta_{k-1,j} - \eta_{k,j} g_{k,j}, \text{ where } \eta_{k,j} = \eta/\sqrt{G_{k,j} + \epsilon}.$$

Here, $G_{k,j} = \sum_{l \le k} g_{l,j}^2$ is the cumulative squared gradients for the j-th parameter, and ϵ is a very small number to avoid division by zero. As we train the model, $G_{k,j}$ will grow, the learning rates may shrink to zero. In this case, the model may prematurely stop learning on the parameter.

- **RMSprop**: In the denominator of the above formula for $\eta_{k,j}$, the sum of squared gradients $G_{k,j}$ is replaced by exponentially weighted average (EWA) of squared gradients,

$$\text{EWA}_{k,j} = (1-\rho)\text{EWA}_{k-1,j} + \rho g_{k,j}^2,$$

where $0 < \rho < 1$. EWA is approximately equivalent to giving exponentially reduced weights to earlier iterations: $w_k = 1$ for $g_{k,j}^2$, $w_{k-1} = \rho$ for $g_{k-1,j}^2$, $w_{k-2} = \rho^2$ for $g_{k-2,j}^2$, etc. Then $\text{EWA}_{k,j} \approx \frac{w_k g_{k,j}^2 + w_{k-1} g_{k-1,j}^2 + \cdots}{w_k + w_{k-1} + \cdots}$. That is, $\sqrt{\text{EWA}_{k,j}}$ is the exponentially weighted RMS (root mean squares) of the gradients.

- **Adadelta**: In the above formula for $\eta_{k,j}$, the numerator η is replaced by an adaptive formula, while the denominator is defined as in RMSprop. With Adadelta, we do not even need to tune the learning rate.

In `optimizer_rmsprop(lr=0.001, rho=0.9, epsilon=None, decay=0.0)`, the default $\rho = 0.9$. It is recommended to leave the parameters of this optimizer at their default values (except the learning rate).

In `optimizer_adadelta(lr=1.0, rho=0.95, epsilon=None, decay=0.0)`, it is recommended to leave the parameters of this optimizer at their default values. Here `lr` can be viewed as the initial value of learning rate.

4. Compute both the learning rate and the gradient adaptively.

- **Adam**: Same as RMSprop, but with the gradient $g_{k,j}$ is replaced by $m_{k,j} = \beta_1 m_{k-1,j} + (1-\beta_1) g_{k,j}$, an EWA of $g_{k,j}$. That is,

$$\theta_{k,j} = \theta_{k-1,j} - \eta_{k,j} m_{k,j},$$

  where $\eta_{k,j}$ is similarly defined as in RMSprop (using notation β_2 instead of ρ). Both β_1 and β_2 should be < 1 but close to 1.

In Keras, `optimizer_adam(lr=0.001, beta_1=0.9, beta_2=0.999, epsilon=None, decay=0.0, amsgrad=False)`

"Adam is currently recommended as the default algorithm to use, and often works slightly better than RMSProp. However, it is often also worth trying SGD+Nesterov Momentum as an alternative." — CS231n (http://cs231n.github.io/neural-networks-3/)

12.11 Avoiding overfitting

With so many parameters in NN models relative to the sample size, the model eventually overfits data after many iterations of learning. But this is often a gradual process. One can monitor the performance of a model on the training and validation data as a function of epoch to identify when overfitting starts. If an iterative fitting algorithm shows a worsening performance on validation data after a certain number of epochs but continues to "improve" on the training data, it is a sign of **overfitting**. However, how "improvement" is quantified can give drastically different impressions on when overfitting begins.

Below are a few other ways to overcoming overfitting in NNs.

Early stopping (not used nowadays): Stop once the classification accuracy on the validation data has plateaued (e.g., not improving over the last 10 epochs). This is not used nowadays because it is not uncommon that the performance may plateau for a while and then improve again. Early stopping was a popular choice in the past. Nowadays regularization, dropout, and batch normalization are often used.

Regularization: In L_2 regularization, the cost function is $C = C_0 + \frac{\lambda}{2n}\sum_w w^2$, where C_0 is the original cost without regularization. In L_1 regularization, $C = C_0 + \frac{\lambda}{n}\sum_w |w|$. Here, the sums are over the weights, not including the biases. No standardization is performed and all the weights are inherently assumed to be on comparable scales. Regularization slows the rate of overfitting.

In L_2 regularization, since $C = C_0 + \frac{\lambda}{2n}\sum_w w^2$, we have $\frac{\partial C}{\partial b} = \frac{\partial C_0}{\partial b}$ and $\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$. Thus the update of bias is the same as before; the update of weight is $w \leftarrow w - \eta\frac{\partial C_0}{\partial w} - \eta\frac{\lambda}{n}w = (1-\frac{\eta\lambda}{n})w - \eta\frac{\partial C_0}{\partial w}$. Here the current estimate of weight is first shrinked by a factor of $(1-\frac{\eta\lambda}{n})$ before adding the gradient. Thus, L_2 regularization is also called **weight decay**. Note that if $\eta\lambda \ll n$, $(1-\frac{\eta\lambda}{n}) \approx \exp(-\frac{\eta\lambda}{n})$.

In L_1 regularization, $C = C_0 + \frac{\lambda}{n}\sum_w |w|$, the update of weight is $w \leftarrow w - \frac{\eta\lambda}{n}\text{sign}(w) - \eta\frac{\partial C_0}{\partial w}$ (with the convention $\text{sign}(0) = 0$).

Dropout: In every iteration, randomly remove a fraction ϕ $(0 < \phi < 1)$ of the hidden neurons and their connections from the model, inflate the remaining neurons by a factor $1 = 1/\phi$, and update the parameters in the modified network. The dropout rate ϕ is a tuning parameter. One can use different dropout rates for different layers. From Krizhevsky (2012): "This technique reduces complex co-adaptations of neurons, since a neuron cannot rely on the presence of particular other neurons. It is, therefore, forced to learn more robust features that are useful in conjunction with many different random subsets of the other neurons."

It can be shown that dropout is approximately a form of generalized ridge regularization. In Keras, it is specified as a model layer using `layer_dropout(rate=)`.

Batch normalization: From Ioffe and Szegedy (2015): "Our method draws its strength from making normalization a part of the model architecture and performing the normalization for each training mini-batch. Batch Normalization

allows us to use much higher learning rates and be less careful about initialization. It also acts as a regularizer, in some cases eliminating the need for Dropout."

For images, **artificial expansion of training data**: Apply operations to data that reflect real-world variation. For image data, rotation/translation/skewing can be used. For MNIST data, "elastic distortions" was used to emulate the random oscillations found in hand muscles (Simard et al. 2003). Input distortion enriches the training data and prevents overfitting to the original image. It forces the model not to focus on isolated values and individual pixels.

It can be shown that input distortion is approximately a form of ridge regularization. One can apply distortions to a test image, and then "poll" the results to produce a final classification. This process is called denoising.

12.12 The MNIST example revisited

I have tried a few options described above for improving GD and overcoming overfitting. Given a fixed model structure (i.e., fixed numbers of layers and nodes), having dropout layers and using the Adam optimizer seem to perform the best among the limited scenarios I have tried on this dataset.

Prepare the data:

```
library(keras)
c(c(x_train, y_train), c(x_test, y_test)) %<-% dataset_mnist()
dim(x_train) = c(nrow(x_train), 784)  ## reshape/flattening X to have 784 columns
dim(x_test) = c(nrow(x_test), 784)
x_train = x_train / 255  ## raw data has range 0-255
x_test = x_test / 255
y_train10 = to_categorical(y_train, 10)  ## make dummy (one-hot) variables for the outcome
y_test10 = to_categorical(y_test, 10)
```

Define and fit a FFNN model.

```
use_session_with_seed(2018)
model2 = keras_model_sequential() %>%
  layer_dense(units = 100, activation = "relu", input_shape = 784) %>%
  layer_dropout(rate = 0.3) %>%
  #layer_dense(units = 100, activation = "relu") %>%
  #layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")

summary(model2)

model2 %>% compile(
  loss = "categorical_crossentropy",
  #optimizer = optimizer_sgd(lr=3, momentum=0.5, decay=0, nesterov=T),
  #optimizer = optimizer_rmsprop(),
  #optimizer = optimizer_adadelta(),
  optimizer = optimizer_adam(),
  metrics = c("accuracy")
)

history2 = model2 %>% fit(
  x_train, y_train10,
  epochs = 20, batch_size = 100, verbose = 1,
  validation_data = list(x_test, y_test10)
)
```

One can plot the training history and apply the model to new data to make predictions.

```
plot(history2)
history2$metrics$val_acc
```

```
 [1] 0.9401 0.9572 0.9652 0.9700 0.9710 0.9733 0.9744 0.9756 0.9763 0.9772
[11] 0.9769 0.9780 0.9778 0.9780 0.9797 0.9784 0.9784 0.9778 0.9794 0.9782
```

```
model2 %>% evaluate(x_test, y_test10, verbose = 0)
y_pred = model2 %>% predict_classes(x_test)  ## prediction on test set
table(y_test, y_pred)  ## test set confusion matrix
```

Note that in the model definition, there is a "dropout layer" although it is not a real layer. Because of this, the numbering of layers changes, with the dropout layer being layer 3 and the last layer being layer 4.

```
wts = model2 %>% get_weights
str(wts)

wts1 = model2 %>% get_layer(index=2) %>% get_weights
#wts1 = get_weights(get_layer(model, index=2))
str(wts1)

wts2 = model2 %>% get_layer(index=3) %>% get_weights
str(wts2)
```

Sanity check: Apply the weights to an observation to see if the results are as expected.

```
testsample = x_test[1000, , drop=F]  ## take an observation
model2 %>% predict_classes(testsample)
model2 %>% predict_proba(testsample)

## For the NN with a single hidden relu layer
z2 = testsample %*% wts[[1]] + array_reshape(wts[[2]], c(1, length(wts[[2]])))
a2 = pmax(z2, 0)  ## relu
z3 = a2 %*% wts[[3]] + array_reshape(wts[[4]], c(1, length(wts[[4]])))
a3 = exp(z3) / sum(exp(z3))  ## softmax
a3
model2 %>% predict_proba(testsample)
all.equal(signif(a3, 4), model2 %>% predict_proba(testsample) %>% signif(4))  ## True (effectively)
```

One can also obtain intermediate output values. Below is an example to obtain the values after the first hidden layer, and after the final layer.

```
layer2_model2 = keras_model(inputs = model2$input,
                            outputs = get_layer(model2, index=2)$output)
testsample_layer2 = predict(layer1_model2, testsample)
all.equal(a2, testsample_layer2)  ## True (effectively)
plot(a2, testsample_layer2); abline(0,1)

layer4_model2 = keras_model(inputs = model2$input,
                            outputs = get_layer(model2, index=4)$output)
testsample_layer4 = predict(layer4_model2, testsample)
all.equal(testsample_layer4, model2 %>% predict_proba(testsample))  ## True
```

Regularization: To apply regularization, use the `kernel_regularizer=` argument in `layer_dense()` to specify the "regularizer". Below is an example of using L_2 regularization on the weights in the first hidden layer with $\lambda = 0.01$. Other options for the argument are `regularizer_l1()` and `regularizer_l1_l2()`. To regularize the biases, use the `bias_regularizer=` argument.

```
model2b = keras_model_sequential() %>%
  layer_dense(units = 100, activation = "relu", input_shape = 784,
              kernel_regularizer = regularizer_l2(l = 0.01)) %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")
summary(model2b)
```

Initialization of weights and biases: By default, in `layer_dense()`, `kernel_initializer = "glorot_uniform"` for the weights and `bias_initializer = "zeros"` for the biases. The "zeros" initializer sets values to be zero. One can specify the seed for the random generator, say, `kernel_initializer = glorot_uniform(seed = 2018)`. There are more than a dozen "initializers" in Keras.

Batch normalization: To add batch normalization to a hidden layer before its activation function, one can do it as in the following, where the first hidden layer has batch normalization before applying the activation function.

```
model2c = keras_model_sequential() %>%
  layer_dense(units = 100, input_shape = 784) %>%
  layer_batch_normalization() %>%
  layer_activation("relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 100, activation = "relu") %>%
  layer_dropout(rate = 0.3) %>%
  layer_dense(units = 10, activation = "softmax")
summary(model2c)
```

12.13 Hyperparameters

Hyperparameters in FFNN fall in three categories:

(1) Those defining the model: number of layers, number of neurons in a layer, activation functions, feature map size and pooling function in CNN.
(2) Those in model fitting criteria: cost function C, λ in regularization (λ can also be viewed as part of a model (as prior of model preference)).
(3) Those used during model fitting: weight initialization, learning rate η, momentum coefficient μ, dropout rate ϕ, mini-batch size, number of epochs.

NNDL Chapter 3 covers hyperparameter tuning in great details. Below is a very brief summary:

- First goal in model building is to get a model better than noise.
- Use a fraction of training and validation data to speed up feedback.
- For a multinomial outcome, use only two outcome categories to simplify the problem.
- Vary hyperparameters one at a time to see if they would improve the performance.

12.14 Customized evaluation metrics in Keras

In Keras, you can define new evaluation metrics to be computed at the end of every epoch. Here is an example we wrote in Python. The metric is to calculate Pearson's correlation between the observed and the predicted values.

```
## Define a metric
def pearson_in_epoch(y_true, y_pred):
    fsp = y_pred - K.mean(y_pred, axis=0)
    fst = y_true - K.mean(y_true, axis=0)
    devP = K.std(y_pred, axis=0)
    devT = K.std(y_true, axis=0)
    return K.mean(fsp*fst, axis=0) / (devP*devT + K.epsilon())

## To use the metric
model.compile(loss='mean_squared_error', optimizer='rmsprop',
              metrics=['mae', pearson_in_epoch])
```

12.15 Example: Prediction of Hi-C intensity

Background: Chromatin conformation capture (3C) is a technique developed in 2001 to evaluate physical interactions between two chromosomal regions. It allowed us to study regulation of gene expression from the physical perspective (in addition to the molecular and statistical perspectives). The technique has been improved to be 4C, 5C, and Hi-C, etc. Hi-C (high-throughput CCC) allows us to obtain data on chromatin conformation agnostically over the whole genome. Hi-C data are expensive to generate. Researchers typically generate data that are good for obtaining confident loop calling at the 40kb resolution, which is crude relative to the size of action that is often at the kb resolution.

Goal: Build a prediction model to make Hi-C intensity calling at a more refined resolution, say 10kb, using the current data.