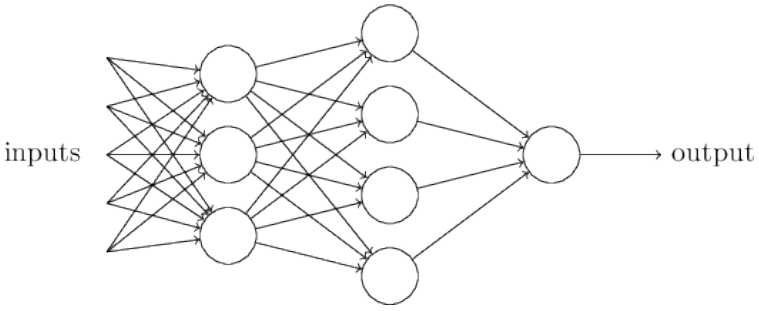
12 Artificial neural networks

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*.

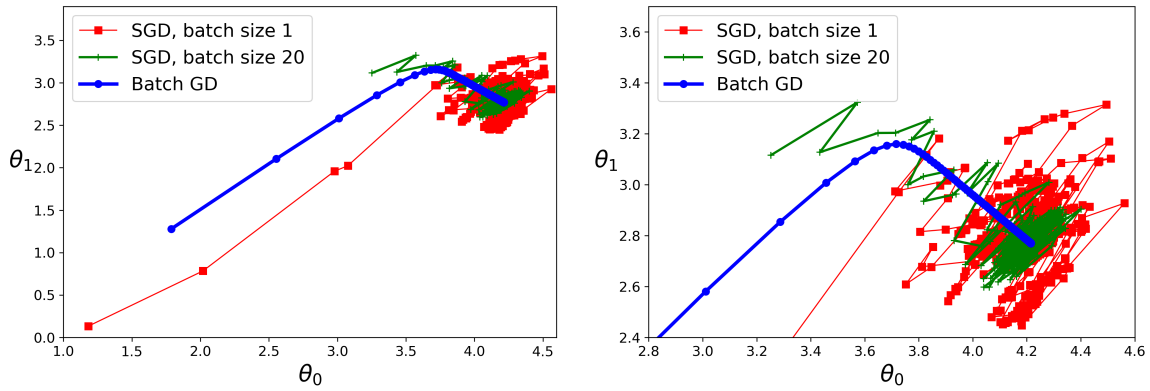
When n is not large, we can calculate ∇C using all observations. This is called **batch gradient descent**.

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 learning** algorithm.
- 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 as they differ only by mini-batch size. Using a very small mini-batch size can be erratic although it may give you a chance to jump out of a local minimum.

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)
model = keras_model_sequential() %>%
  layer_dense(units = 30, activation = "sigmoid", input_shape = c(784)) %>%
  layer_dense(units = 10, activation = "softmax")

summary(model)
```

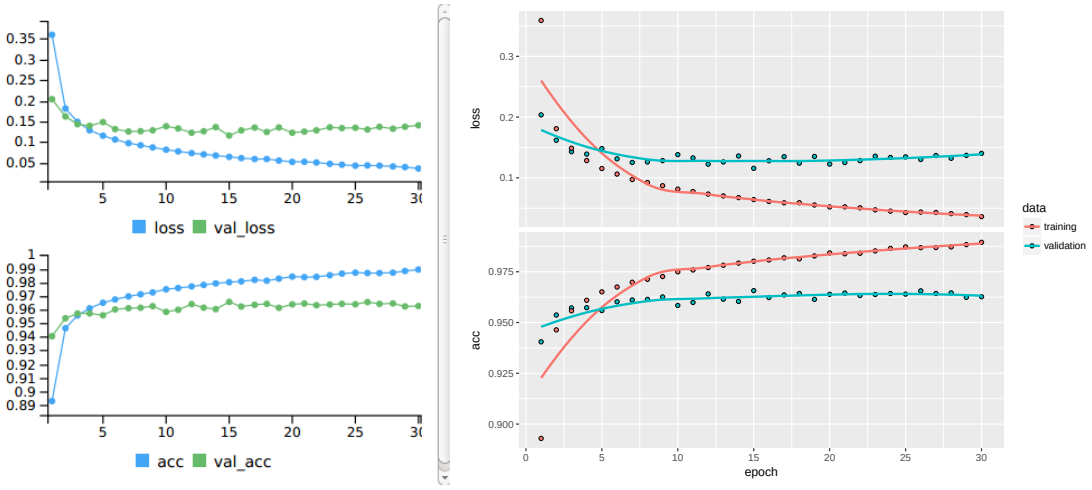
We then set the options for model fitting, and then fit it!

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

history = model %>% fit(
  x_train, y_train10,
  epochs = 30, batch_size = 100, verbose = 1,
  validation_data = list(x_test, y_test10)
  #validation_split = 0.2
)
```

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

```
plot(history)  ## the plot on the right
str(history)
history$metrics$val_loss
```



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 can stop here. Beyond this point the model may become overfitting.

Note that the learning rate 3 may be too high. Re-training the model with a smaller learning rate gives a different plot.

We can evaluate the model on the test set:

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

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

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.
- According to the explanation at here, setting the seed "disables GPU computations and CPU parallelization by default (as both can lead to non-deterministic computations)". Setting the seed does not work for the code above. The change in val_loss was very small when `optimizer_rmsprop` was used, but a little large when `optimizer_sgd` was used.

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 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
```