11 PQHS 471 Notes Week 11

11.1 Week 11 Day 1 (SVM)

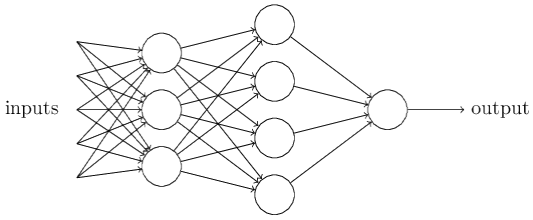
11.2 Week 11 Day 2

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. 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 in Python 2.7 is at https://github.com/mnielsen/neural-networks-and-deep-learning/ . I will use the code in Python 3.5, which is at https://github.com/MichalDanielDobrzanski/DeepLearningPython35

11.2.1 A simple feedforward neural network

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



Here there are two hidden layers between the 5 input variables and the output variable. Often the input and the output are viewed as layers, and the above network is called a 4-layer NN. 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 in a linear way and then a transformation is applied. For example, node i in layer 2 (the first hidden layer) has input $z_i^{(2)} = \sum_{j=1}^{5} w_{ij}^{(2)} x_j + b_i^{(2)}$, and output

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

The parameters $w_{ij}^{(2)}$ and $b_i^{(2)}$ are called **weights** and **bias** in the NN literature. The transformation $h_i^{(2)}()$ is called an **activation function**. When $h(t) = I(t \geq 0)$, the node is also called a *perceptron*. When $h(t) = \sigma(t) = (1 + e^{-t})^{-1}$, 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 $h(t) = I(t \geq 0)$) or a *sigmoid layer* (when $h(t) = \sigma(t)$). This simple FFNN is sometimes called an **MLP** (multilayer perceptron) even though none of the nodes is a perceptron.

In the above figure, 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). There is a total of $18 + 16 + 5 = 39$ parameters. If layers 2–4 are all sigmoid layers, the model is effectively

$$\begin{aligned} f(x_1, \ldots, x_5) &= \sigma \left[w_1^{(4)} a_1^{(3)} + w_2^{(4)} a_2^{(3)} + w_3^{(4)} a_3^{(3)} + w_4^{(4)} a_4^{(3)} + b^{(4)} \right], \\ \text{with } a_i^{(3)} &= \sigma \left[w_{i1}^{(3)} a_1^{(2)} + w_{i2}^{(3)} a_2^{(2)} + w_{i3}^{(3)} a_3^{(2)} + b_i^{(3)} \right] \quad (i = 1, 2, 3, 4) \\ \text{and } a_i^{(2)} &= \sigma \left[w_{i1}^{(2)} x_1 + w_{i2}^{(2)} x_2 + w_{i3}^{(2)} x_3 + w_{i4}^{(2)} x_4 + w_{i5}^{(2)} x_5 + b_i^{(2)} \right] \quad (i = 1, 2, 3). \end{aligned}$$

To fit the model, we need to estimate these 39 parameters using 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. Instead, in NN, we often rely on the 1st-order approximation algorithm called the **gradient descent** or its stochastic version, the **stochastic gradient descent** (details below).

11.2.2 Tensorflow and Keras

A **tensor** is just another name for an array. A matrix is a 2-dimensional array. In Tensorflow, every operation is formulated as a flow of tensors. For example, in the above feedforward NN model, the flow is from a 5D tensor to a 3D tensor and then a 4D tensor and a 1D tensor.

Tensorflow is a package from Google. **Keras** is a front end package for using Tensorflow. Keras is not necessary but it makes it a lot easier to fit standard NN models. Both are 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
```

In Windows, Anaconda 3.x is required. In the second step, `install_keras("conda")`, it first installs a few necessary conda packages, then creates a conda environment and installs TF packages into it.

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. The configuration file is $HOME/.keras/keras.json. All example datasets downloaded with `datasets_*()` are in $HOME/.keras/datasets/.

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

```
library(keras)

model = keras_model_sequential()  ## initialize the 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()
```

11.2.3 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: $h(x) = x_+ + ax_-$, with $a > 0$ close to zero

Perceptron: $h(x) = I(x \geq 0)$

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

A **softmax layer** is often used for the final layer when the outcome is multinomial. For example, for the MNIST dataset, the output layer has 10 nodes. The activation function at node j is $a_j = \frac{\exp(z_j)}{\sum_k \exp(z_k)}$, where z_k is the input into node k and is a linear combination of the output from the nodes in the previous layer. This definition ensures that $\{a_1, \ldots, a_{10}\}$ is a multinomial probability distribution with $\sum_j a_j = 1$.

11.2.4 Commonly used cost functions

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. As a result, while the training data cost is being driven down, the test data cost can be ascending while the test data classification accuracy is being improved (NNDL Chapter 3).

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

Negative log-likelihood: $-\ln L$.

Cross-entropy for $0 \leq y \leq 1$: $C(y,a) = -[y\ln a + (1-y)\ln(1-a)]$ with $a \in (0,1)$; $\nabla_a C = \frac{a-y}{a(1-a)}$. In back-propagation, if $H = \sigma$, then $\delta^L = a^L - y$.

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

Total cost: $C = \frac{1}{n}\sum_i C(y_i, a_i)$, where y_i is the outcome for observation i and a_i is a predicted value given x_i.

11.2.5 Gradient descent

Gradient descent (GD) is an iterative algorithm for minimizing a function $C(\theta_1, \ldots, \theta_p)$. 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:

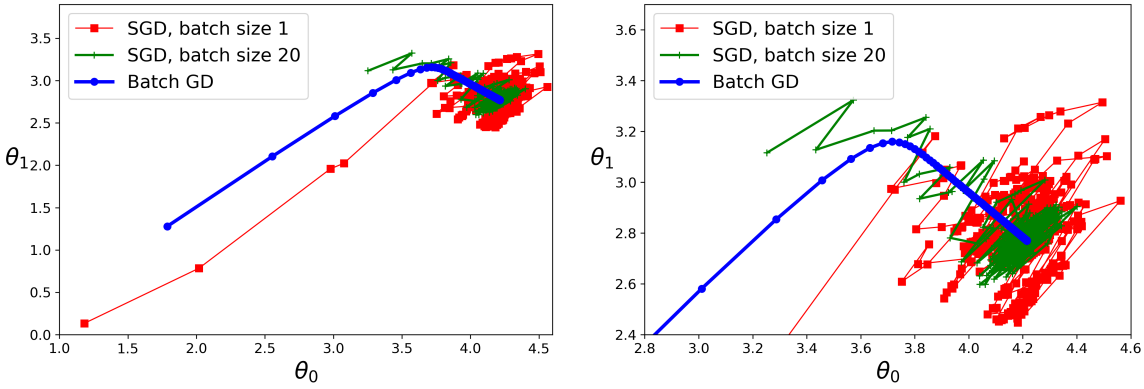
(1) Initialize θ.
(2) At every iteration, calculate ∇C with the current θ estimates, and update $\theta \leftarrow \theta - \eta \nabla C$, where $\eta > 0$ is the **learning rate** (which should be small).

The **rationale** is: 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**. When n is very large, the calculation of ∇C can be slow. We can use **stochastic gradient descent** (SGD), in which we estimate ∇C using a subset of the data. A **mini-batch size** is chosen (say, 100). The training set is randomly partitioned into subsets of the mini-batch size, and we iterate through all the subsets. After going through all the subsets, we have finished one **epoch**. Then we repartition the training set and repeat the process. When $m = 1$, it is also called **online learning**.

Note that 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):



Notes on the **learning rate**:

- A too high rate can lead to divergence due to "overshooting" from one iteration to the next.
- A too low rate can make the training very slow.
- Gradient descent benefits from a small learning rate. As a first-order approximation, it may break down 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, and so it could be learned by using training data: 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. The optimal value of η depends on the choice of C.
- Some methods of SGD (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:

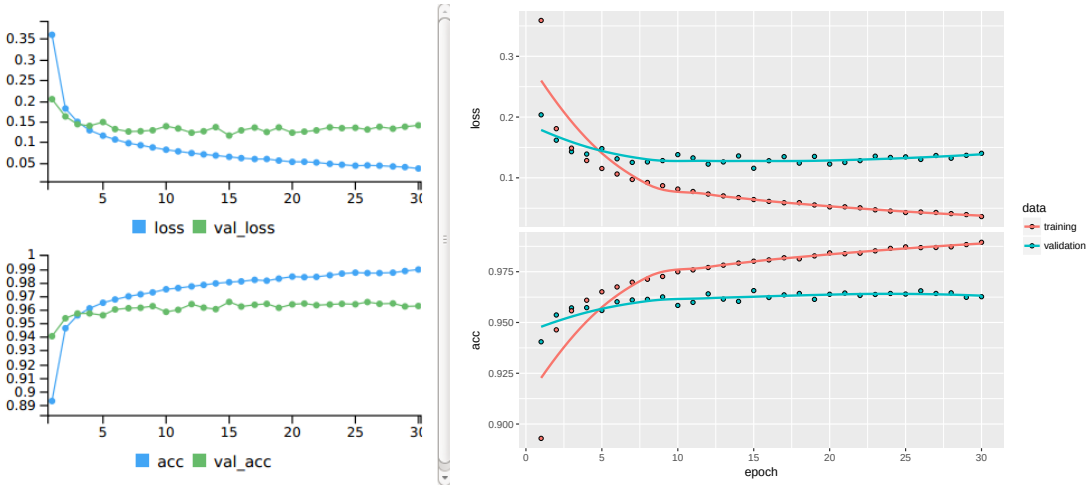
- GD is a first-order approximation. In contrast, the Hessian approach is a second-order approximation (e.g., Newton–Raphson and Gauss–Newton methods).
- GD cannot guarantee to converge to the global minimum of $C(u)$. Global minimum is guaranteed if C is convex and ∇C is Lipschitz.

11.2.6 A simple example

Let us try a simple FFNN on the MNIST dataset. The model has an input layer with 784 features, a single hidden sigmoid layer with 30 nodes, and a softmax final layer with 10 nodes.

```
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 the FFNN model
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)

model %>% compile(
  loss = "categorical_crossentropy",
  optimizer = optimizer_sgd(lr = 3),  ## SGD optimizer
  #optimizer = optimizer_rmsprop(),
  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
)

plot(history)  ## the plot on the right

str(history)
history$metrics$val_acc

## Evaluate the mode on the test set
model %>% evaluate(x_test, y_test10, verbose = 0)

y_pred = model %>% predict_classes(x_test)
table(y_test, y_pred)
```



Note about setting seeds: 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.

The Python code Michael Nielsen (the author of NNDL) wrote is a straightforward Python implementation of the backpropagation algorithm for FFNN. It is not only faster than the keras/TF approach above but also more robust.

11.2.7 Multinomial logistic regression

When there is no hidden layer, the FFNN becomes multinomial logistic regression. Note that I can put everything into a single series of pipes.

```
use_session_with_seed(2018)
mlogit = keras_model_sequential() %>%
  layer_dense(units = 10, activation = "softmax", input_shape = c(784)) %>%
  compile(optimizer = optimizer_rmsprop(),
    loss = "categorical_crossentropy", metrics = c("accuracy")) %>%
  fit(x_train, y_train10, epochs = 30, batch_size = 100, verbose = 1,
    validation_data = list(x_test, y_test10))

mlogit$metrics$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
mlogit$metrics$val_loss
 [1] 0.3329465 0.2948545 0.2819920 0.2797526 0.2740982 0.2702867 0.2709323
 [8] 0.2715503 0.2689993 0.2683985 0.2688575 0.2718867 0.2710647 0.2707043
[15] 0.2699300 0.2714954 0.2728006 0.2732992 0.2735500 0.2727475 0.2734667
[22] 0.2782687 0.2759682 0.2759473 0.2791749 0.2781309 0.2776476 0.2771545
[29] 0.2799522 0.2797758
plot(mlogit)
```

Such regression 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
```

11.2.8 Assignment

1. Reading for next lecture: NNDL 2, 3