Deep Learning

36-600

Fall 2021

The Punch Line, First

You've heard me talk about how while there may be times for deep learning, they don't necessarily include the analysis of conventional, small- to medium-sample-size data.

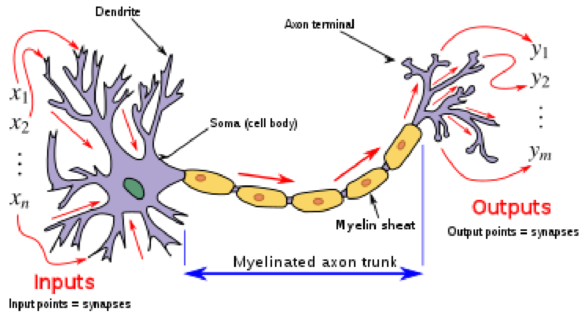
Don't just take it from me, however. From *ISLR* 2e:

"When faced with new data modeling and prediction problems, it's tempting to always go for the trendy new methods…However, 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. Whenever possible, it makes sense to try the simpler models as well, and then make a choice based on the performance/complexity tradeoff.*

Typically we expect deep learning to be an attractive choice when the sample size of the training set is extremely large, and when interpretability of the model is not a high priority."

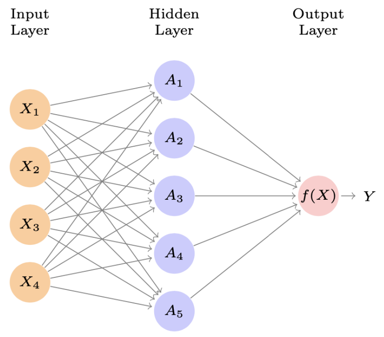
(Artificial) Neural Networks

- Deep learning is an extension of an older machine learning technique, the *artificial neural network*, or simply *neural network* or *NN*.
- An NN is so named because its architecture was thought to be akin to the way neurons work in the brain. From Wikipedia's page on artificial neurons: "The artificial neuron receives one or more inputs (representing excitatory postsynaptic potentials and inhibitory postsynaptic potentials at neural dendrites) and sums them to produce an output (or activation, representing a neuron's action potential which is transmitted along its axon)."



- NNs were fashionable ML models in the "old days" (in the 1950s/1960s, then again in 1980s/1990s), then ultimately fell out of favor because (a) they are more complex and harder to tune than off-the-shelf methods like random forest, and thus (b) they need large training datasets (and more advanced computational architectures for efficiency). With the advent of large digitized datasets and (Moore's-Law-predicted) computational advances, NN returned as *deep learning* in the 2010s.

Single-Layer Feed-Forward Neural Networks



(Figure 10.1 of *ISLR* 2e.)

- The *input layer* consists of your predictor variables $\{x_1, \ldots, x_p\}$.
- *(The) hidden layer(s)* transform(s) linear inputs (e.g., for the first layer, $w_1 x_1 + \cdots + w_p x_p$) into a nonlinear output through the use of *activation functions*.
- Each hidden layer is comprised of some number of *hidden units*.
- The output layer contains the predicted response. For a classifier, there are as many units in the output layer as response factor levels. (Historically a binary classifier is dubbed a *perceptron*.)
- Note that different flavors of deep learning (CNNs, RNNs, etc.) differ mainly in architecture.

There are no heuristics, per se, for how many hidden layers you should use, nor for how many hidden units you should have in each layer. The more layers/units, the greater the model complexity, the greater the chance of overfitting. (In particular, p can be $> n$!) Hence the use of regularization (i.e., complexity penalties ala lasso or ridge regression), which we will come back to.

Activation Functions

Another punch-line first: if nonlinear activation functions are not applied in the hidden layers, then the model that results at the output layer is simply a…linear model. You could have simply skipped deep learning and gone straight to linear regression.

In a single-layer NN with p predictor variables and K hidden units, the final model is

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

where A_k is the specific activation function for hidden unit k.

Activation Functions

Commonly used activation functions include

- The sigmoid function:

$$g(y) = \frac{e^y}{1+e^y}\ .$$

  This should be familiar from logistic regression, as the sigmoid function is the inverse of the logit function. Note that a single-layer perceptron with one hidden unit is…wait for it…logistic regression.

- The rectified linear unit (or ReLU) function:

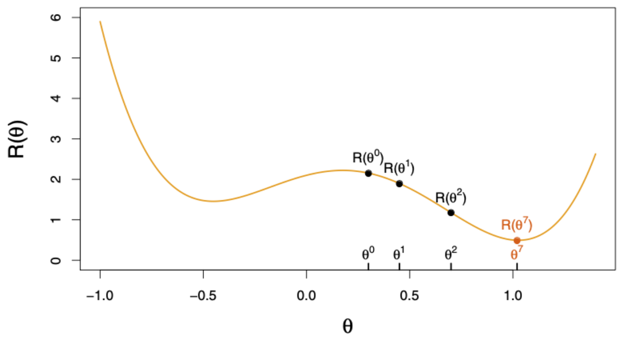
$$g(y) = \max(0, y)\ .$$

  The primary advantages of the ReLU function are that (a) it encourages spareness by effectively suppressing (setting to zero value) a substantial proportion of the hidden units, and (b) is computationally efficient to work with.

Optimization

The goal in deep learning, as when learning any other model, is to minimize (or optimize) an *objective function*. For instance, for regression the objective function could be the mean-squared error.

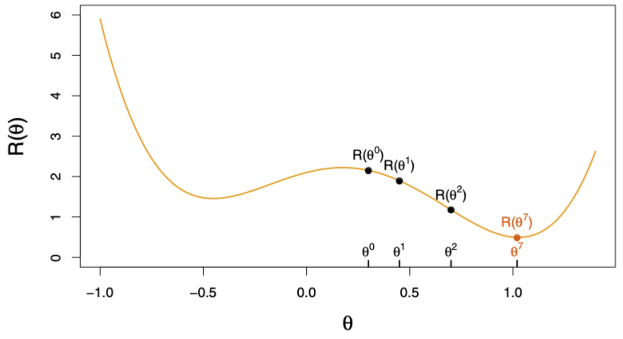
As the values of model parameters change, the value of the objective function changes. We can picture this for one parameter as follows:



(Figure 10.17 of *ISLR* 2e.)

Here θ is a model parameter (e.g., one of the weights w) and $R(\theta)$ is the optimization function. The goal of an optimizer is to move from an initial guess for θ to the *global minimum* $R(\theta_{\min})$. However, one risk is that an optimizer will find a local minimum instead…and get stuck there.

Optimization



So the question is: how does the optimizer figure out how to go "downhill"?

There are *many* optimization paradigms, but a common one involves computng the gradient...dubbed *gradient descent*. The gradient of $R(\theta)$ at $\theta = \theta_o$ is

$$\nabla R(\theta_o) = \left.\frac{\partial R(\theta)}{\partial \theta}\right|_{\theta=\theta_o}.$$

The gradient is a *vector* that points in the uphill direction. Given the (reverse) direction, an optimizer then "simply" has to determine how big of a step to take.

The chain rule of differentiation links all the model parameters to the gradient, and thus via *backpropagation* one can determine how much to change each parameter's value when taking an optimization step. (For more details, see section 10.7.1 of *ISLR* 2e.)

Optimization

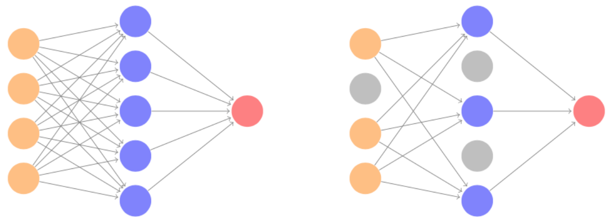
There are two optimization tweaks of note left to discuss.

- The first is motivated by the fact that gradient descent is computationally intensive: it involves summations over the n observations. To get around this, in any one optimization step we might sum over only a randomly selected subset of the observations, or a *minibatch*. This tweak changes gradient descent into *stochastic gradient descent* (or *SGD*).

(Note another term here: an *epoch* represents a set of steps in which the equivalent of the full dataset has been sampled for SGD. For intuition: if you randomly sample a quarter of the data, an epoch is four optimization steps.)

- The second tweak is motivated by a desire to avoid overfitting: regularization, or the addition of a penalty term to the objective function $R(\theta)$. There is nothing "special" to this, as you have seen this before...one can implement, e.g., a lasso penalty (add $\lambda \sum_j |\theta_j|$ to $R(\theta)$) or a ridge regression penalty (add $\lambda \sum_j \theta_j^2$).

But there is a newer regularization mechanism: *dropout learning*. This involves dropping a percentage of the units in a layer as optimization proceeds.



Deep Learning: Tuning

So, in summary, to fit a deep learning model, we need to specify:

- the number of hidden layers and the number of hidden units per layer;
- the details of SGD: the minibatch size, the number of epochs to run; and, potentially,
- the regularization tuning parameter (e.g., λ, or the dropout percentage).

Deep Learning: Example

In the interests of (computation) time, we demonstrate deep learning with the 10,000-galaxy dataset we've used previously, which we use to associate galaxy distance ("redshift") with galaxy brightness.

```r
library(keras)

df = read.csv("https://raw.githubusercontent.com/pefreeman/PSU_2019/master/PhotoZ.csv")[,-8]
x = model.matrix(redshift~.,data=df)
y = df$redshift

set.seed(808)
s = sample(nrow(df),0.3*nrow(df))

model = keras_model_sequential() %>%
  layer_dense(units=10,activation="relu",input_shape=ncol(x)) %>%
  layer_dropout(rate=0.3) %>%
  layer_dense(units=1)
model %>% compile(loss="mse",optimizer=optimizer_rmsprop(),metrics=list("mean_squared_error"))
```

Depending on how your computer and `R` are set up, you may need to do some or all of the following when running the `model = ...` code:

- install the `keras` and `tensorflow` packages
- install `miniconda` when `R` prompts you to do so
- run `tensorflow::install_tensorflow()` when `R` prompts you to do so

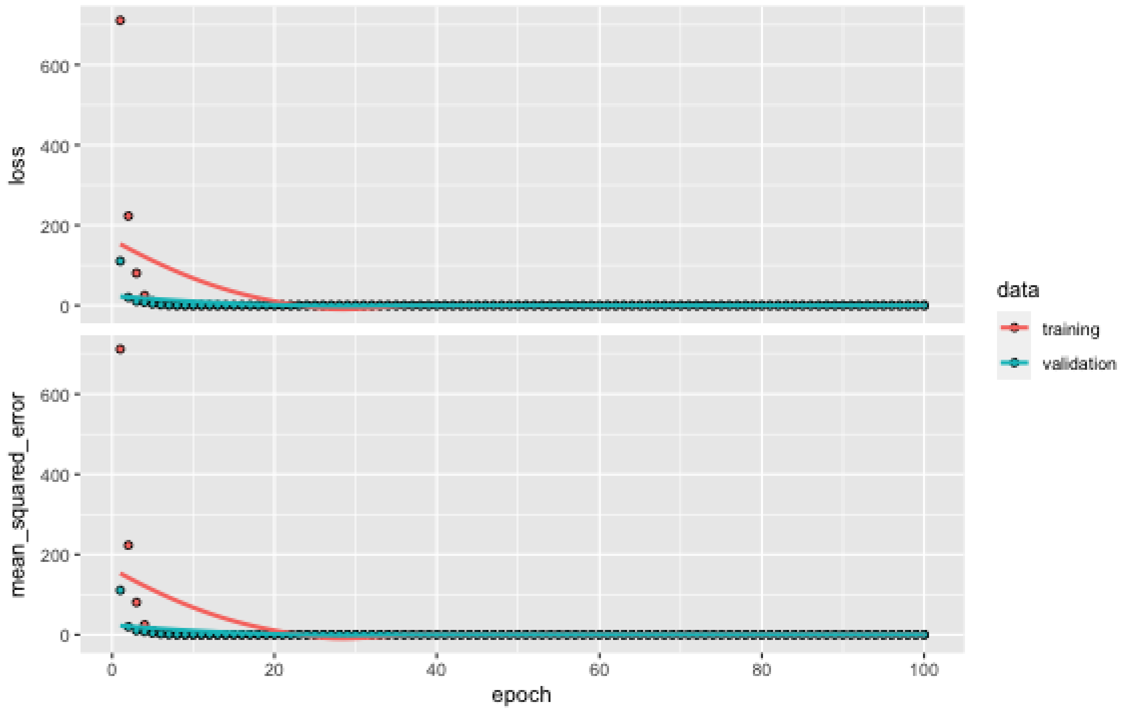
I saw all of these when developing this notes set on my laptop. But the `model = ...` code did run eventually. (Note that the above example is based entirely on *ISLR* 2e, Section 10.9.1.)

Deep Learning: Example

(Note: the first time you run the `fit()` function, do not include the `verbose` argument, so that you can get a sense as to how fast fitting is progressing.)

```
history = model %>%
  fit(x[-s,],y[-s],epochs=100,batch_size=128,validation_data=list(x[s,],y[s]),verbose=0)
plot(history)
```

```
## `geom_smooth()` using formula 'y ~ x'
```



Deep Learning: Example

```r
resp.pred = predict(model,x[s,])
mean((resp.pred-df[s,]$redshift)^2)
```

```
## [1] 0.1518933
```

Compare this value with what we get for random forest: ≈ 0.07. Further tuning is needed! (I'm not doing it here because I want my notes slides to compile relatively quickly!) The main change that can be made is to increase the number of hidden units in this single-layer model, or perhaps to increase the number of layers. Or play with the dropout proportion. Etc.

NOTE: deep learning models involving randomness (via SGD or dropout learning, etc.) that are learned with `keras` are **not reproducible**, as any seed that you set does not get propagated to the `Python` code that is actually being run under the hood.