Intro to Deep Learning

Big Data and Machine Learning for Applied Economics

Ignacio Sarmiento-Barbieri

Universidad de los Andes

Deep Learning: Intro

- Neural networks are simple models.
- Their strength lays in their simplicity which allows for fast training and computation.
- The model has linear combinations of inputs that are passed through nonlinear activation functions called nodes (or, in reference to the human brain, neurons).

Deep Learning: Intro

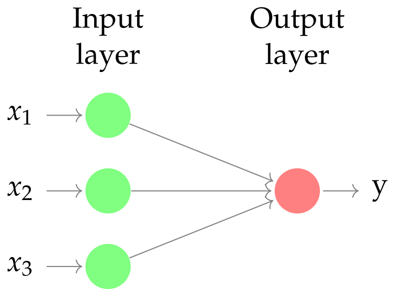
- Let's start with a familiar and simple model, the linear model

$$\begin{aligned} y &= f(X) + u \qquad (1) \\ y &= \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + u \end{aligned}$$

Deep Learning: Intro

- Let's start with a familiar and simple model, the linear model

$$y = f(X) + u \tag{1}$$
$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \beta_3 x_3 + u$$



Single Layer Neural Networks

- Linear Models may be to simple, and miss the nonlinearities that best approximate $f^*(x)$
- We can overcome these limitations of linear models and handle a more general class of functions by incorporating one or more hidden layers.
- Neural Networks are also called deep feedforward networks, feedforward neural networks, or multilayer perceptrons (MLPs), and are the quintessential deep learning models

Single Layer Neural Networks

- A neural network takes an input vector of p variables

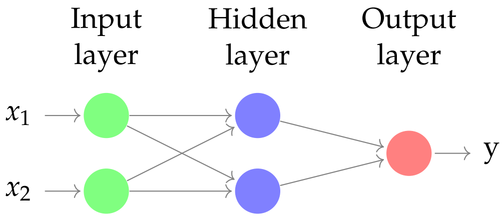
$$X = (X_1, X_2, ..., X_p) \tag{2}$$

- and builds a nonlinear function $f(X)$ to predict the response y .

$$y = f(X) + u \tag{3}$$

- What distinguishes neural networks from previous methods is the particular structure of the model.

Single Layer Neural Networks



Single Layer Neural Networks

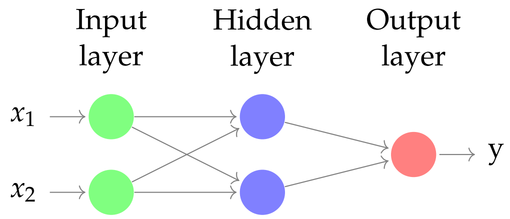
- The NN model has the form

$$
\begin{aligned}
f(X) &= \beta_0 + \sum_{k=1}^{K} \beta_k A_k &(4)\\
&= \beta_0 + \sum_{k=1}^{K} \beta_k h_k(X) &(5)\\
&= \beta_0 + \sum_{k=1}^{K} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j \right) &(6)
\end{aligned}
$$

- where $g(.)$ is a activiation function specified in advance
- where the nonlinearity of $g(.)$ is essential

Single Layer Neural Networks

- Let's consider a very simple example with
  - $p = 2, X = (X_1, X_2)$
  - $K = 2, h_1(X)$ and $h_2(X)$
  - $g(z) = z^2$



Worked Example: The "Exclusive OR (XOR)" Function

- The exclusive disjunction of a pair of propositions, (p, q), is supposed to mean that p is true or q is true, but not both
- It's truth table is:

q	p	q v p
0	0	0
0	1	1
1	0	1
1	1	0

- When exactly one of these binary values is equal to 1, the XOR function returns 1. Otherwise, it returns 0

NN Minimalist Theory

- Why not a linear activation functions?
- Let's go back to our example
  - $p = 2$, $X = (X_1, X_2)$
  - $K = 2$, $h_1(X)$ and $h_2(X)$
  - Now $g(z) = z$
- Then

$$f(X) = \beta_0 + \sum_{k=1}^{2} \beta_k A_k \tag{7}$$

$$= \beta_0 + \sum_{k=1}^{2} \beta_k h_k(X) \tag{8}$$

$$= \beta_0 + \sum_{k=1}^{2} \beta_k g\left(w_{k0} + \sum_{j=1}^{p} w_{kj} X_j\right) \tag{9}$$

Activation Functions

- The gain comes from using nonlinear activation function f
- Note that, with nonlinear activation functions in place, it is no longer possible to collapse our NN into a linear model.
- Activation functions are fundamental to deep learning, let us briefly survey some common activation functions.
- In practice we would not use a quadratic function, since we would always get a second-degree plynomial in the original coordinates
- We use others that we brefly review here

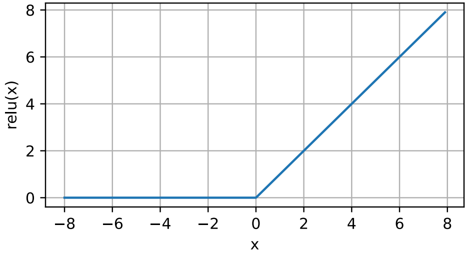
Activation Functions
ReLU Function

- ReLU Function
  - The most popular choice, due to both simplicity of implementation and its good performance on a variety of predictive tasks, is the rectified linear unit (ReLU).
  - ReLU provides a very simple nonlinear transformation. Given an element x, the function is defined as the maximum of that element and 0:

$$\text{ReLU}(x) = \max\{x, 0\}.$$

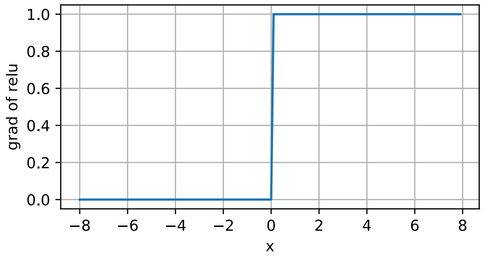
Activation Functions

- ReLU function retains only positive elements and discards all negative elements by setting them to 0.
- It is piecewise linear.



Activation Functions

- Part of the appeal of ReLU has to do with it's well behaved derivative
  - Note that the ReLU function is not differentiable when the input takes value precisely equal to 0. In these cases, we default to the left-hand-side derivative and say that the derivative is 0 when the input is 0. (we may even get away with this because the input may never actually be zero!)
  - This makes optimization better behaved and it mitigates the problem of vanishing gradients



Activation Functions

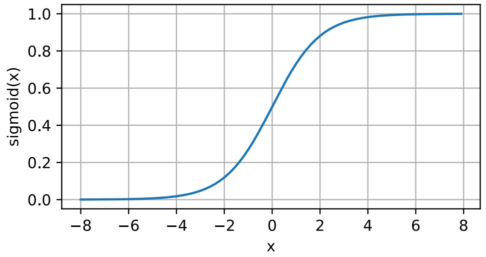
Sigmoid Function (Logit)

- The sigmoid function transforms its inputs, for which values lie in the domain $\mathbb{R}$, to outputs that lie on the interval (0, 1).
- For that reason, the sigmoid is often called a squashing function: it squashes any input in the range (-inf, inf) to some value in the range (0, 1):

$$\mathrm{sigmoid}(x) = \frac{1}{1 + \exp(-x)}.$$

Activation Functions

Sigmoid Function (Logit)



- In the earliest neural networks, scientists were interested in modeling biological neurons which either fire or do not fire. Thus the pioneers of this field, going all the way back to McCulloch and Pitts, the inventors of the artificial neuron, focused on thresholding units.
- A thresholding activation takes value 0 when its input is below some threshold and value 1 when the input exceeds the threshold.
- When attention shifted to gradient based learning, the sigmoid function was a natural choice because it is a smooth, differentiable approximation to a thresholding unit.

Activation Functions

- Other Activation functions
  - Hiperbolic tangent: $\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}$
  - Radial basis function (RBF): $exp\left(\frac{1}{\sigma^2)}||W - x||^2\right)$
  - Softplus: $log(1 + e^x)$
  - Hard tanh: $max(-1, min(1, x))$
  - $h = cos(Wx + b)$ Goodfellow et al. (2016) claim that on the MNIST dataset they obtained an error rate of less than 1 percent
- Hidden unit design remains an active area of research, and many useful hidden unit types remain to be discovered

Output Functions

- The choice of cost function is tightly coupled with the choice of output unit.
- Most of the time, we simply use the distance between the data distribution and the model distribution.
  - Linear $y = \beta' h + \beta_0 \rightarrow \mathbb{R}$
  - Sigmoid (Logistic) $\frac{1}{1+\exp(-x)}$ $\rightarrow$ classification $\{0, 1\}$
  - Softmax $\frac{exp(x)}{\sum exp(x))}$ $\rightarrow$ classification multiple categories

MNSIT Example

- Let's illustrate with an example
- We are going to use the `MNIST` handwritten digit dataset

MNSIT Example

```
#packages
library(keras)

#load the data
mnist <- dataset_mnist()
```

```
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y
```

MNSIT Example

```r
#Bit of cleaning for X
# reshape
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
# rescale
x_train <- x_train / 255
x_test <- x_test / 255
```

MNSIT Example

```
#Bit of cleaning for y
y_train <- to_categorical(y_train, 10)
y_test <- to_categorical(y_test, 10)
head(y_train)
```

```
##      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
## [1,]    0    0    0    0    0    1    0    0    0     0
## [2,]    1    0    0    0    0    0    0    0    0     0
## [3,]    0    0    0    0    1    0    0    0    0     0
## [4,]    0    1    0    0    0    0    0    0    0     0
## [5,]    0    0    0    0    0    0    0    0    0     1
## [6,]    0    0    1    0    0    0    0    0    0     0
```

MNSIT Example

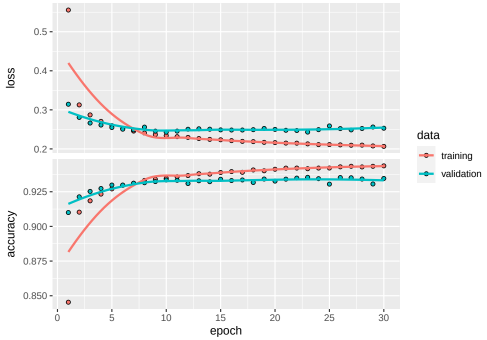
```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 10, activation = 'relu', input_shape = c(784)) %>%
  layer_dense(units = 10, activation = 'softmax')

summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## dense_1 (Dense)                     (None, 10)                      7850
## ________________________________________________________________________________
## dense (Dense)                       (None, 10)                      110
## ================================================================================
## Total params: 7,960
## Trainable params: 7,960
## Non-trainable params: 0
```

MNSIT Example

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



MNSIT Example

```
model %>% evaluate(x_test, y_test)
```

```
##      loss  accuracy
## 0.2552324 0.9352000
```

```
model  %>% predict(x_test) %>% k_argmax()
```

```
## tf.Tensor([7 2 1 ... 4 5 6], shape=(10000), dtype=int64)
```

Architecture Design

- Another key design consideration for neural networks is determining the architecture.
- The word architecture refers to the overall structure of the network: how many units it should have and how these units should be connected to each other.
- The universal approximation theorem guarantees that regardless of what function we are trying to learn, a sufficiently large MLP will be able to represent this function.

Architecture Design

- The universal approximation theorem (Hornik et al., 1989; Cybenko, 1989) states that:
  - A feedforward network with a linear output layer and at least one hidden layer with any “squashing” activation function (such as the logistic sigmoid activation function) can approximate any Borel measurable function from one finite-dimensional space to another with any desired nonzero amount of error, provided that the network is given enough hidden units.

Architecture Design

- We are not guaranteed, however, that the training algorithm will be able to learn that function.
- Even if the network is able to represent the function, learning can fail for two different reasons.
    1. The optimization algorithm used for training may not be able to find the value of the parameters that corresponds to the desired function.
    2. The training algorithm might choose the wrong function as a result of overfitting

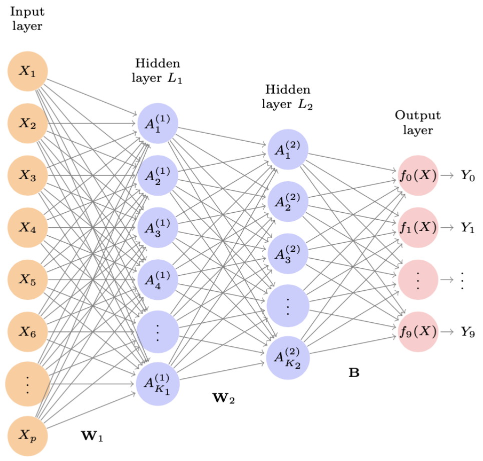
Architecture Design

- A feedforward network with a single layer is sufficient to represent any function, but the layer may be infeasible large and may fail to learn and generalize correctly.
- In many circumstances, using deeper models can reduce the number of units required to represent the desired function and can reduce the amount of generalization error.
- The ideal network architecture for a task must be found via experimentation guided by monitoring the validation set error

Multilayer Neural Networks

- Modern neural networks typically have more than one hidden layer, and often many units per layer.
- In theory a single hidden layer with a large number of units has the ability to approximate most functions.
- However, the learning task of discovering a good solution is made much easier with multiple layers each of modest size.

MNIST Digits



MNIST Digits

```r
#packages
library(keras)

#load the data
mnist <- dataset_mnist()
```

```r
x_train <- mnist$train$x
y_train <- mnist$train$y
x_test <- mnist$test$x
y_test <- mnist$test$y
```

MNIST Digits

```r
#Bit of cleaning for X
# reshape
x_train <- array_reshape(x_train, c(nrow(x_train), 784))
x_test <- array_reshape(x_test, c(nrow(x_test), 784))
# rescale
x_train <- x_train / 255
x_test <- x_test / 255
```

MNIST Digits

```
model <- keras_model_sequential()
model %>%
  layer_dense(units = 256, activation = 'relu', input_shape = c(784)) %>%
  layer_dense(units = 128, activation = 'relu') %>%
  layer_dense(units = 10, activation = 'softmax')
```

MNIST Digits

```
summary(model)
```

```
## Model: "sequential"
## ________________________________________________________________________________
## Layer (type)                        Output Shape                    Param #
## ================================================================================
## dense_2 (Dense)                     (None, 256)                     200960
## ________________________________________________________________________________
## dense_1 (Dense)                     (None, 128)                     32896
## ________________________________________________________________________________
## dense (Dense)                       (None, 10)                      1290
## ================================================================================
## Total params: 235,146
## Trainable params: 235,146
## Non-trainable params: 0
## ________________________________________________________________________________
```

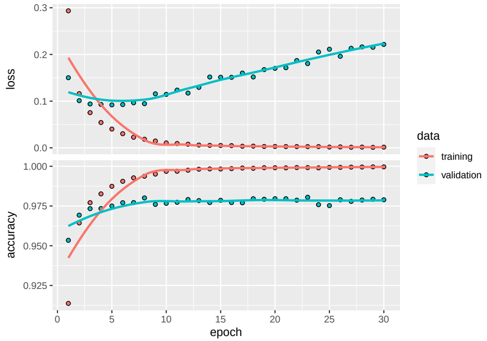
MNIST Digits

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

```
history <- model %>% fit(
  x_train, y_train,
  epochs = 30, batch_size = 128,
  validation_split = 0.2
)
```

MNIST Digits

```
plot(history)
```



MNIST Digits

```
model %>% evaluate(x_test, y_test)
```

```
##       loss  accuracy
## 0.1561636 0.9819000
```

When to use Deep Learning

```r
library(ISLR2)
Gitters <- na.omit(Hitters)
n <- nrow(Gitters)
set.seed(13)
ntest <- trunc(n / 3)
testid <- sample(1:n, ntest)
```

When to use Deep Learning

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

```
## [1] 254.6687
```

When to use Deep Learning

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

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

```
## [1] 252.2994
```

When to use Deep Learning

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

###
x <- model.matrix(Salary ~ . - 1, data = Gitters) %>% scale()
###
modnn %>% compile(loss = "mse",
                  optimizer = optimizer_rmsprop(),
                  metrics = list("mean_absolute_error")
)
```

When to use Deep Learning

```
###
history <- modnn %>% fit(
  x[-testid, ], y[-testid], epochs = 600, batch_size = 32,
  validation_data = list(x[testid, ], y[testid])
)
```

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

```
## [1] 263.5622
```

When to use Deep Learning

- Comparing the three models

Model	MAE
LR	254.6687
Lasso	252.2994
NN	263.5622

Further Readings

- Aston Zhang, Zachary C. Lipton, Mu Li, and Alexander J. Smola (2020) Dive into Deep Learning. Release 0.15.1. http://d2l.ai/index.html
- Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). Deep learning (Vol. 1, No. 2). Cambridge: MIT press. http://www.deeplearningbook.org
- James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). An introduction to statistical learning.
- Rstudio (2020). Tutorial TensorFlow https://tensorflow.rstudio.com/tutorials/beginners/basic-ml/tutorial_basic_classification/
- Taddy, M. (2019). Business data science: Combining machine learning and economics to optimize, automate, and accelerate business decisions. McGraw Hill Professional.