Introduction to Deep Learning wit R

Gabriela de Queiroz

Data Scientist and Founder of R-Ladies

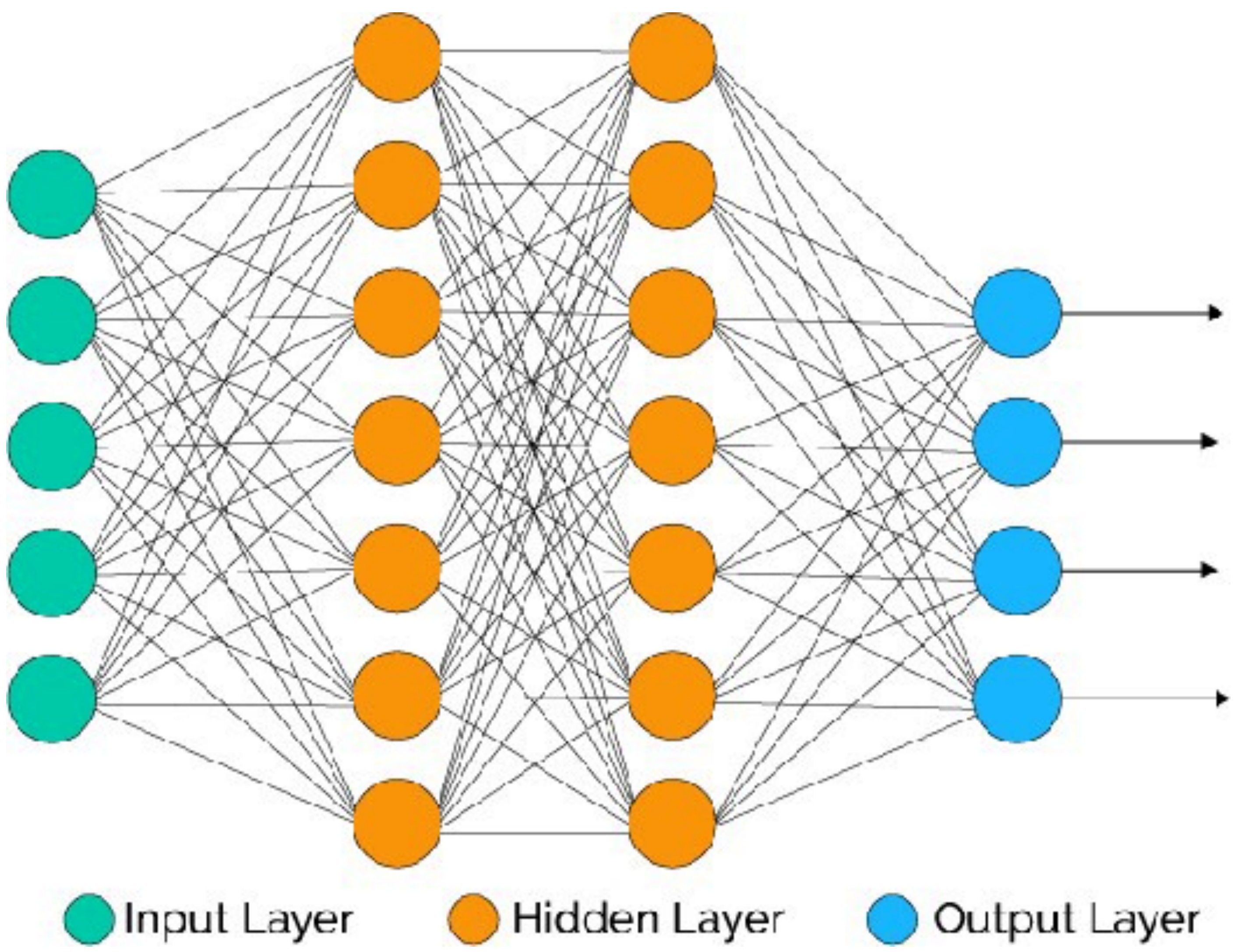
k-roz.com @gdequeiroz

What is deep learning?

- Subfield of machine learning
- New take on learning representations from data: puts an emphasis on learning **successive layers** of increasingly meaningful representations
- "deep" stands for the idea of successive layers of representations
- **Depth of the model:** how many layers contribute to a model of the data

What is deep learning?

- In deep learning, these layered representations are (almost always) learned via models called ***neural networks***, structured in literal layers stacked on top of each other.

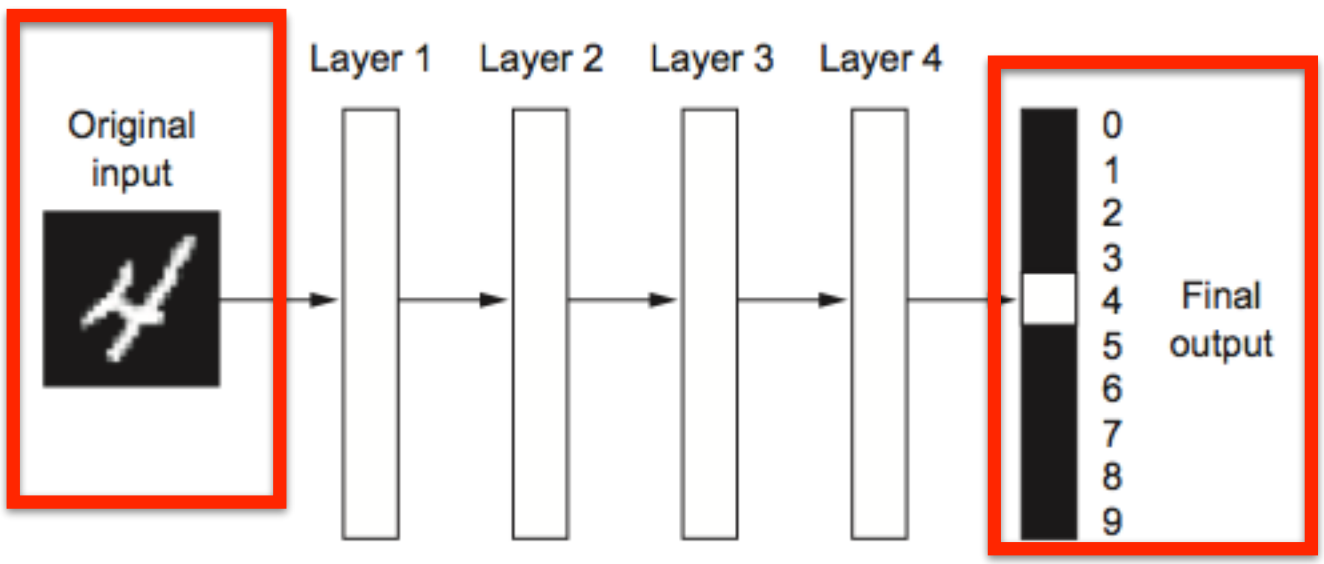


What is deep learning?

- Take some input (observations, X) and transform into some output (predictions, Y) via successive layers of representation

Input

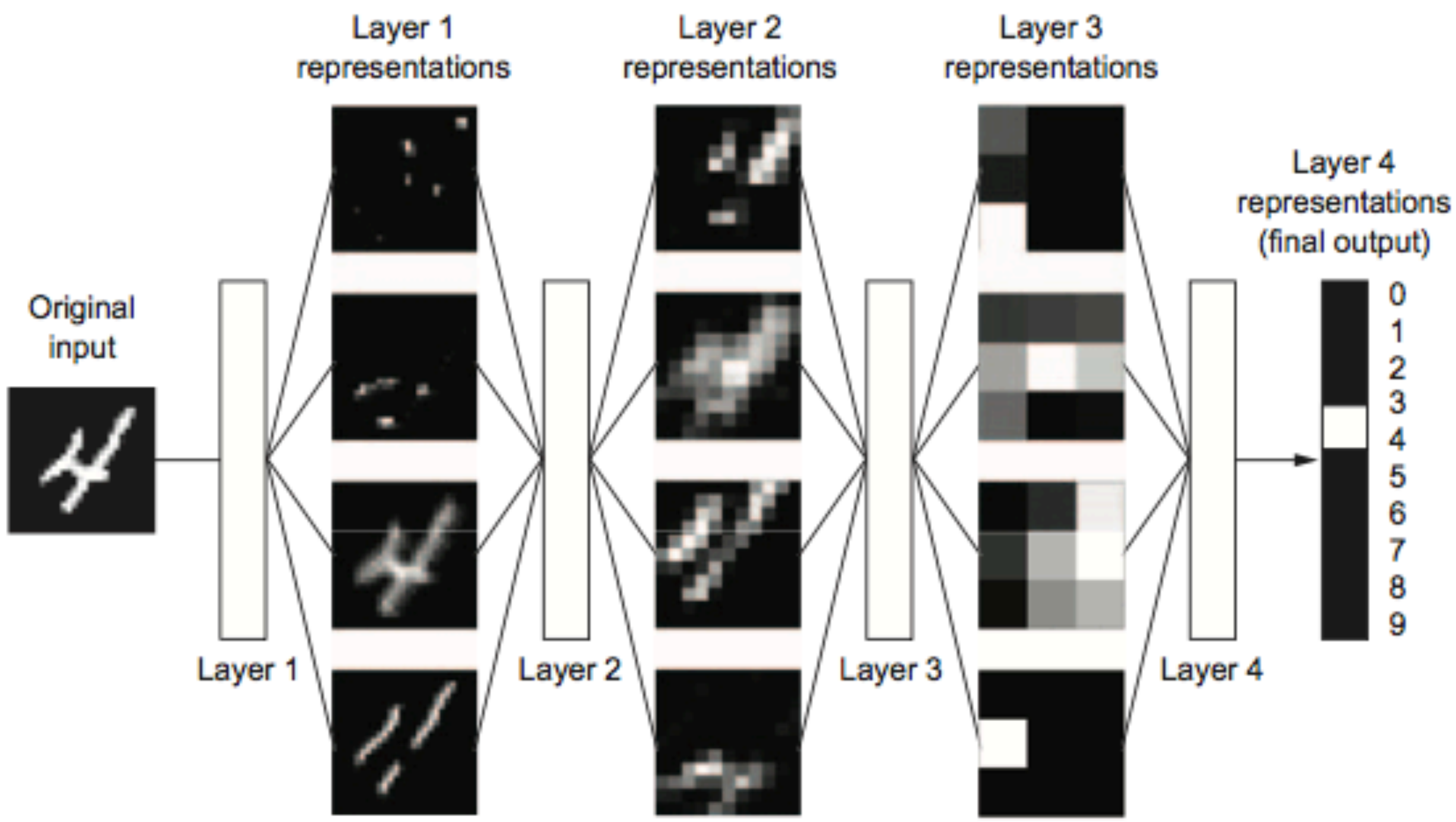
(grayscale image of a handwrittern number 4)



Output

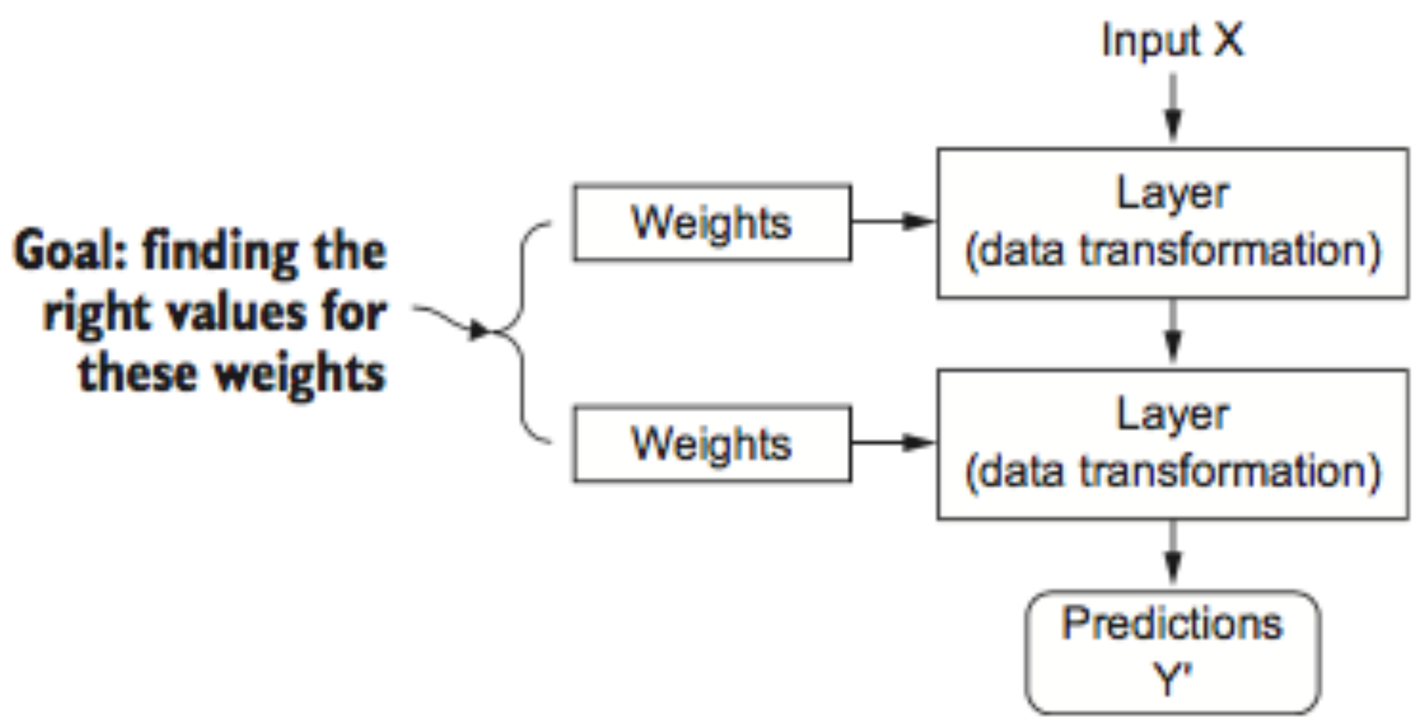
(prediction about what the digit is)

Deep representations learned by a digit-classification model



What are layers?

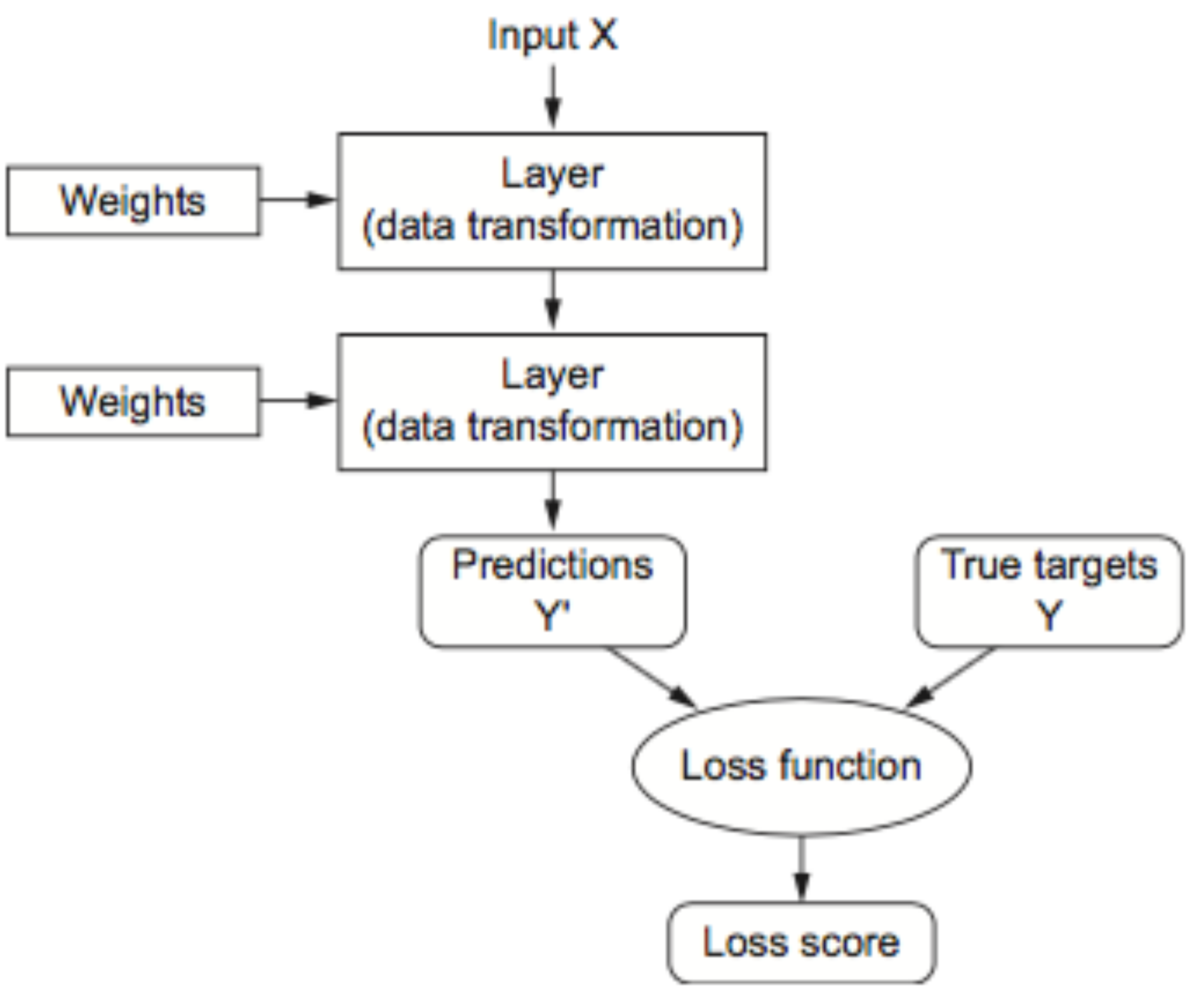
- Data transformation functions *parameterized* by weights* (like a linear equation)



*Weights are also sometimes called the parameters of a layer

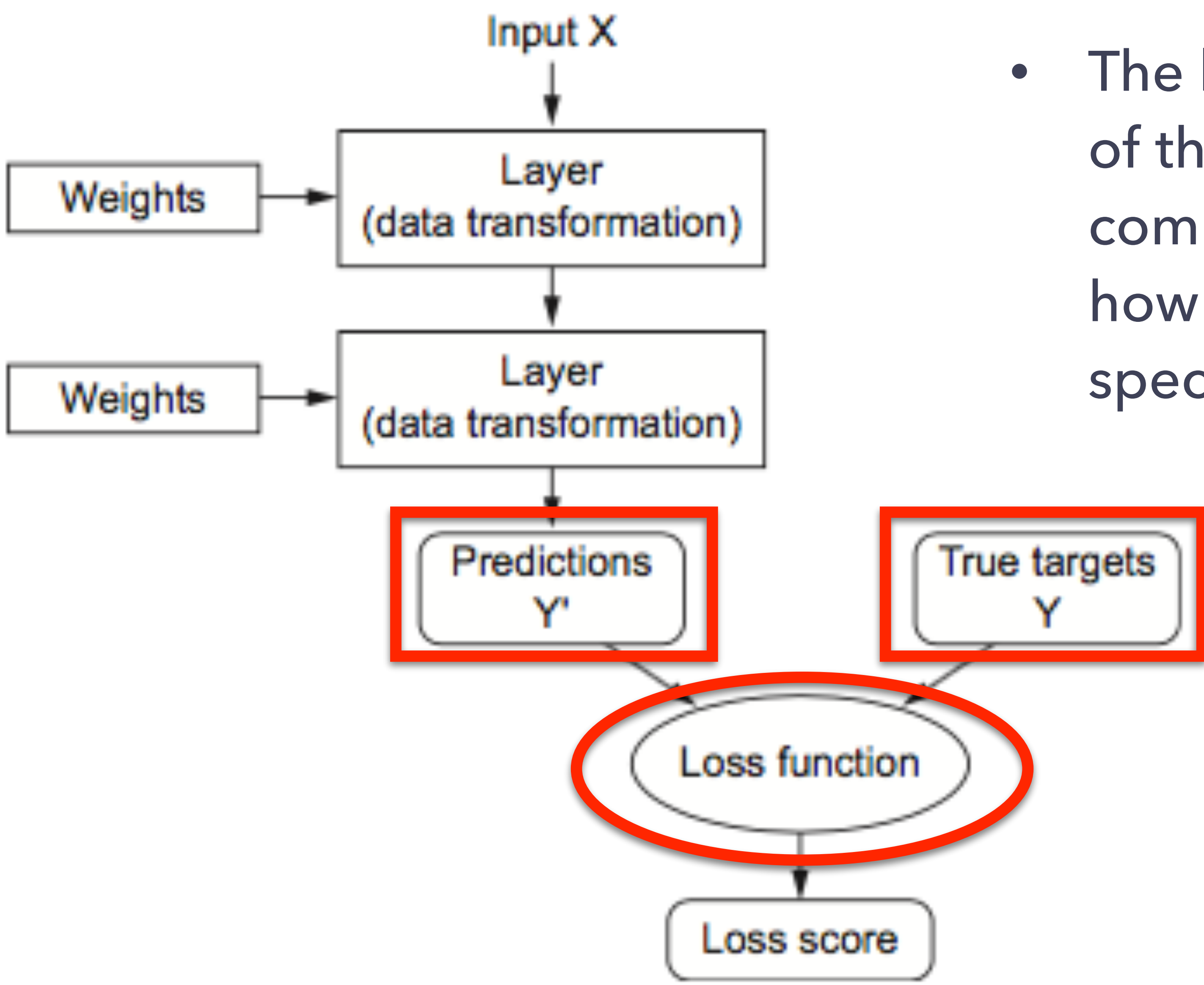
Loss Function/ Objective Function

- To control the output of a neural network, you need to be able to <u>measure</u> how far this output is from what you expected (**loss function** of the network or **objective function**).



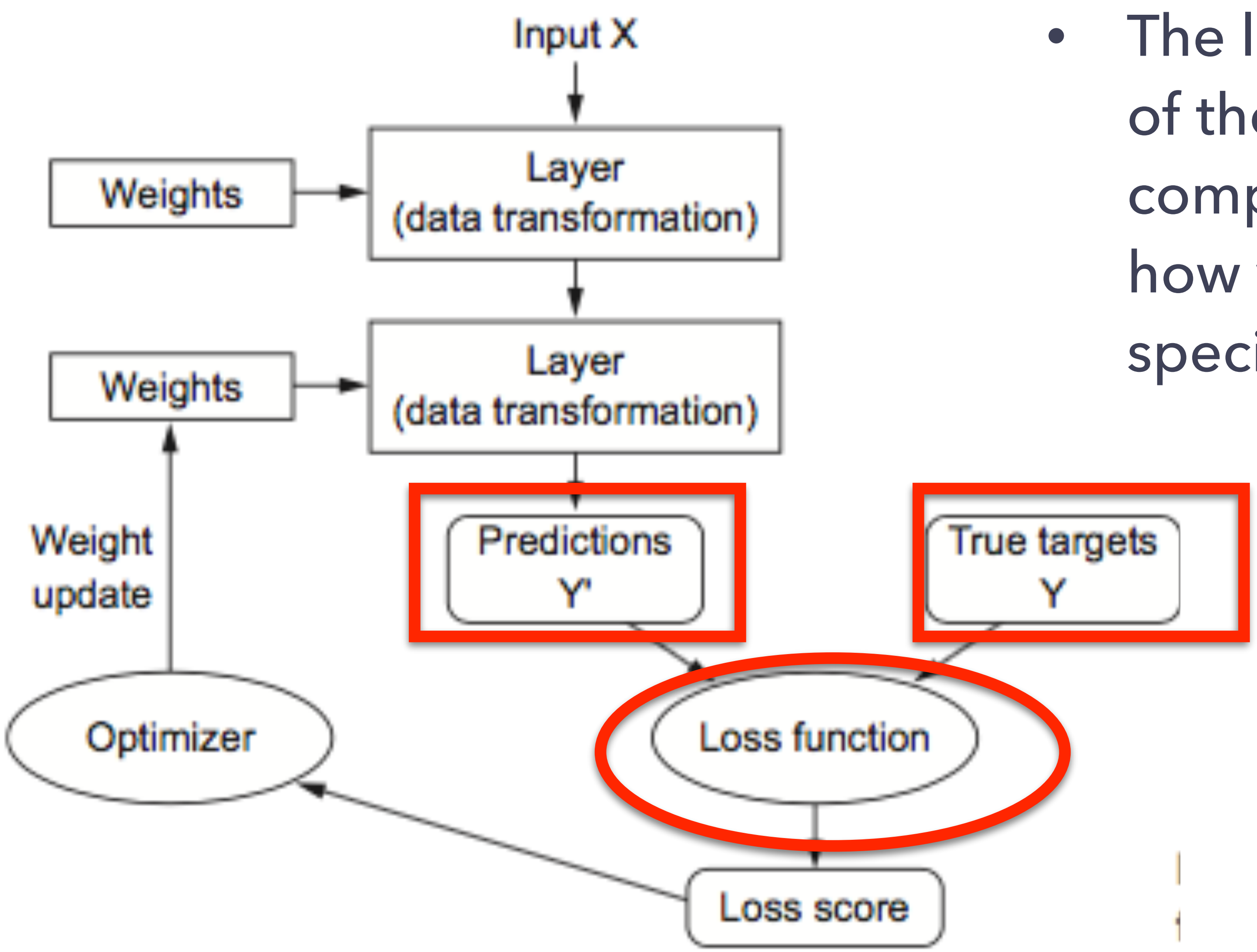
*Weights are also sometimes called the parameters of a layer

Loss Function/ Objective Function



- The loss function takes the **predictions** of the network and the **true target** and computes a distance score, capturing how well the network has done on this specific example.

Backpropagation Algorithm



- The loss function takes the **predictions** of the network and the **true target** and computes a distance score, capturing how well the network has done on this specific example.

How do we train deep learning models?

```
network <- keras_model_sequential() %>%
  layer_dense(units = 512,
              activation = "relu",
              input_shape = c(28 * 28)) %>%
  layer_dense(units = 10,
              activation = "softmax")
```

10-way softmax layer, which means it will return an array of 10 probability scores (summing to 1)

Data Representation for Neural Networks

- Tensors: generalization of vectors and matrices to an arbitrary number of dimensions (note that in the context of tensors, a dimension is often called an axis).

- In R, **vectors** are used to create and manipulate 1D tensors, and **matrices** are used for 2D tensors. For higher-level dimensions, array objects (which support any number of dimensions) are used.

-

Three key attributes of a tensor

- **Number of axes** (rank)
- **Shape**
- **Data Types**

Number of axes (rank)

- For instance, a 3D tensor has three axes, and a matrix has two axes.

Shape

- **Shape:** integer vector that describes how many dimensions the tensor has along each axis.
    - For instance, the matrix below has shape (3, 5)

```
> x <- matrix(rep(0, 3*5), nrow = 3, ncol = 5)

> x
     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0

> dim(x)
[1] 3 5
```

- The 3D tensor example has shape (2, 3, 2)

```
> x <- array(rep(0, 2*3*2), dim = c(2,3,2))

> str(x)
num [1:2, 1:3, 1:2] 0 0 0 0 0 0 0 0

> dim(x)
[1] 2 3 2
```

- A vector has a shape with a single element, such as (5).
- You can access the dimensions of any array using the dim() function.

Data Type

- Type of the data contained in the tensor
  - For instance, a tensor's type could be integer or double
  - On rare occasions, you may see a character tensor

Getting started with Neural Networks using Keras

Workflow

1. Data Preprocessing
2. Constructing the model
3. Compile
4. Fit The Model (Train the model)
5. Evaluating your Model
6. Predict Labels of new data
7. Fine-tuning your model
8. Saving, Loading or Exporting your model

1) Data Preprocessing

- Reshaping and Scaling Data

NOT PART OF THIS PRESENTATION

2) Build the model

1) Define your **training** data: input tensors and target tensors.

2) Define a network of layers (or **model**) that maps your inputs to your targets.

`keras_model_sequential()`

```
model <- keras_model_sequential() %>%
  layer_dense(units = 16,
              activation = "relu",
              input_shape = ncol(x_train)) %>%
  layer_dense(units = 16,
              activation = "relu") %>%
  layer_dense(units = 1,
              activation = "sigmoid")
```

MODEL DEFINITION
What are my layers and how they are going to behave?

Activation Functions

Activation Function for each Layer

Example of activation functions: Sigmoid, Tanh and Relu

It appears that in most cases we can use the **relu** activation function. Relu looks like this:

- It is important to note that there's no **best** activation function. One may be better than other in many cases, but will be worse in some other cases.

- Another important note is that using different activations function doesn't affect what our network can learn, only how fast (how many data/epochs it needs).

https://towardsdatascience.com/exploring-activation-functions-for-neural-networks-73498da59b02

3) Compile the model

What are the ***loss function*** *and* ***optimizer*** that I want to use during training and what **metrics** do I want to collect?

`compile()`

```
model %>% compile(
optimizer = "rmsprop",
loss = "binary_crossentropy",
metrics = c(“accuracy")
)
```

Note that the model is modified in place. It is all done in place and you don’t assign to an object

4) Fit the model

Feeding mini-batches of data to the model thousands of times

`fit()`

```
model %>% fit(
      x_train,
      y_train,
      batch_size = 512,
      epochs = 4
)
```

- Feed 512 samples at a time to the model (`batch_size = 512`)
- Transverse the input dataset 4 times (`epochs = 4`)

4) Fit the model

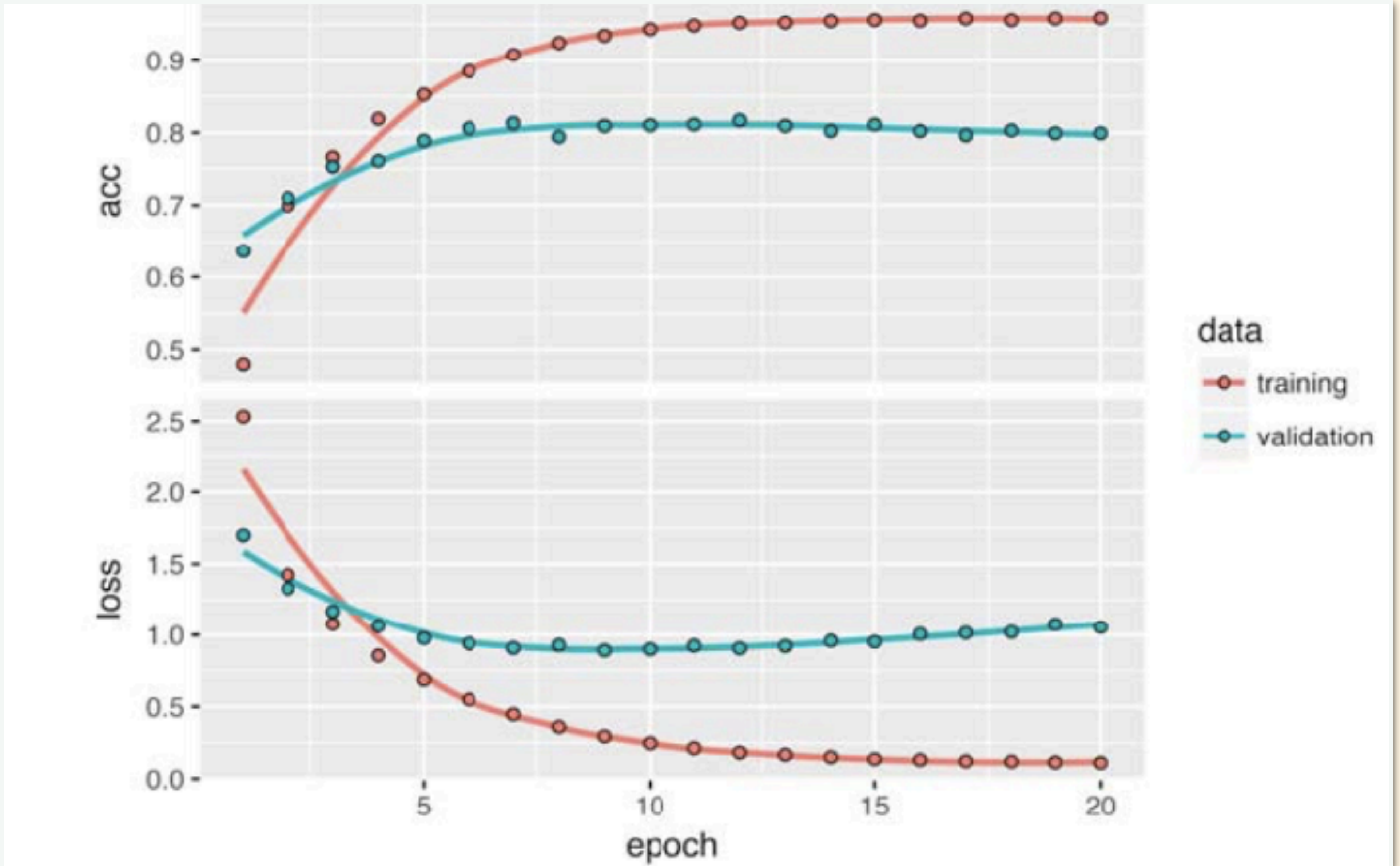
```
model %>% fit(
      x_train,
      y_train,
      batch_size = 512,
      epochs = 4,
      validation_split = 0.2
)
```

As we are fitting, we are going to hold out 20% of the data to **validate** that we are not just overfitting to our dataset

A deep learning model can memorize the data and it can give you a function that is not useful since what it just did was memorize the data

- Hold out 20% of the data to validation (`validation_split = 0.2`)

```
history <- model %>%
  fit(x_train,
      y_train,
      batch_size = 512,
      epochs = 4,
      validation_split = 0.2)
plot(histoy)
```



5) Evaluate the model

Feeding mini-batches of data to the model thousands of times

`evaluate()`

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

$loss
[1] 0.2916625

$acc
[1] 0.884
```

6) Predict Labels of new data

Generate predictions from the model

`predict()`

```
  mmodel %>% predict(x_test[1:10,])

       [,1]
 [1,] 0.1922472
 [2,] 0.9998888
 [3,] 0.8777745
 [4,] 0.8634432
 [5,] 0.9594250
 [6,] 0.9000725
 [7,] 0.9998257
 [8,] 0.0138099
 [9,] 0.9733002
[10,] 0.9950176
```