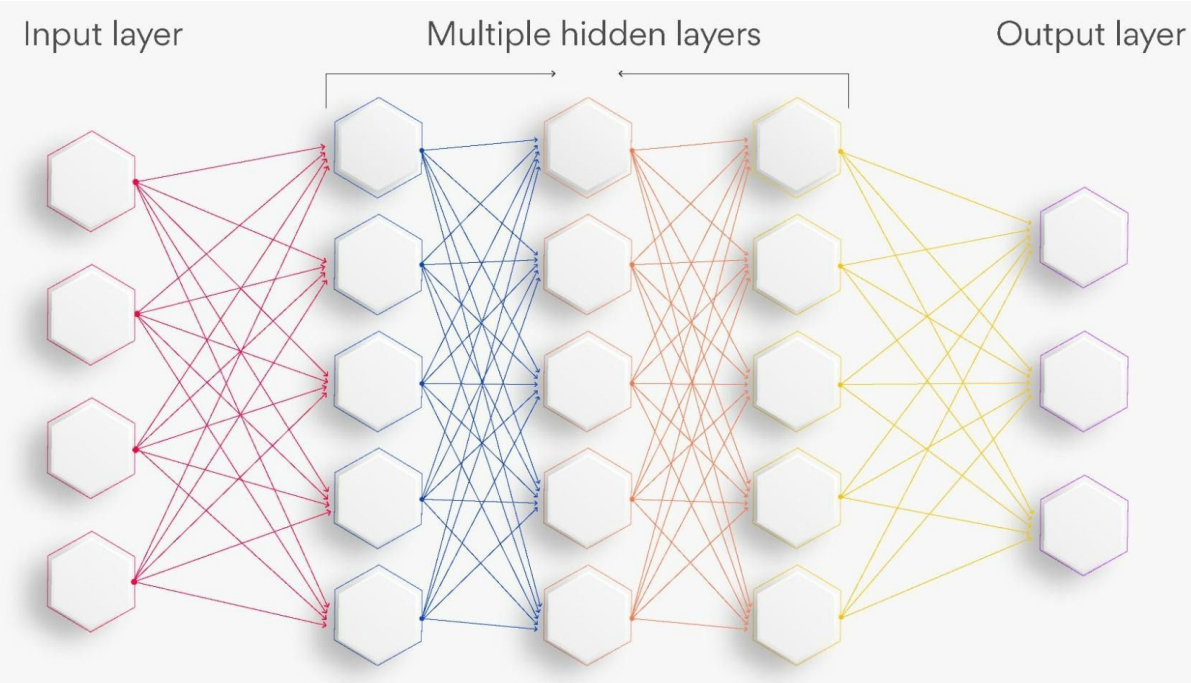
Neural Networks

PSY-GS 8875 Behavioral Data Science



Overview: Week 5

Readings

- ESL Chapters: 11.1-11.5
- HML: Chapter 13
- 3Blue1Brown YouTube
- Brilliant Wiki on Backpropagation

Optional

- Urban and Gates - 2021
- Smith - 2018
- Optimization for Deep Learning

- Unknown functions of unknown complexity
- Neural networks (from scratch)
  - activation functions
  - backpropagation
- Neural networks using {keras}
- Activity: predicting alcohol use disorder

Unknown Functions of Unknown Complexity

Unknown Functions of Unknown Complexity

How many relationships in the social sciences do you expect to be linear? What about logistic/sigmoid?

Unknown Functions of Unknown Complexity

Data Modeling

In traditional statistics and the social sciences, data are molded to a model (data modeling). When using a linear model, for example, you get out a linear relationship whether or not the relationship between your **X** and subsequent predictions are linearly related to your **Y**. **The linear model models linear relationships.** The extent to which your data and subsequent predictions can be "molded" into a linear relationship determines how well the *model* "fits" the data (often, indicated by R^2).

Unknown Functions of Unknown Complexity

Algorithmic Modeling

Algorithmic modeling uses non-parametric, non-linear models that seek to maximally identify the relationship between **X** and **Y** – **whatever that relationship might be.** These algorithms are not a one-size-fits-all. This flexibility comes with an extra requirement: Algorithmic models come with hyperparameters (e.g., learning rate) that need to be adjusted with each and every dataset. The optimal organization of the parameters are unknown because the underlying structure of the relationship between **X** and **Y** is assumed to be unknown. Compare this notion to linear regression where the underlying organization is assumed to be a (weighted) composite of the variables because the relationship is assumed to be linear.

Unknown Functions of Unknown Complexity

Takeaways

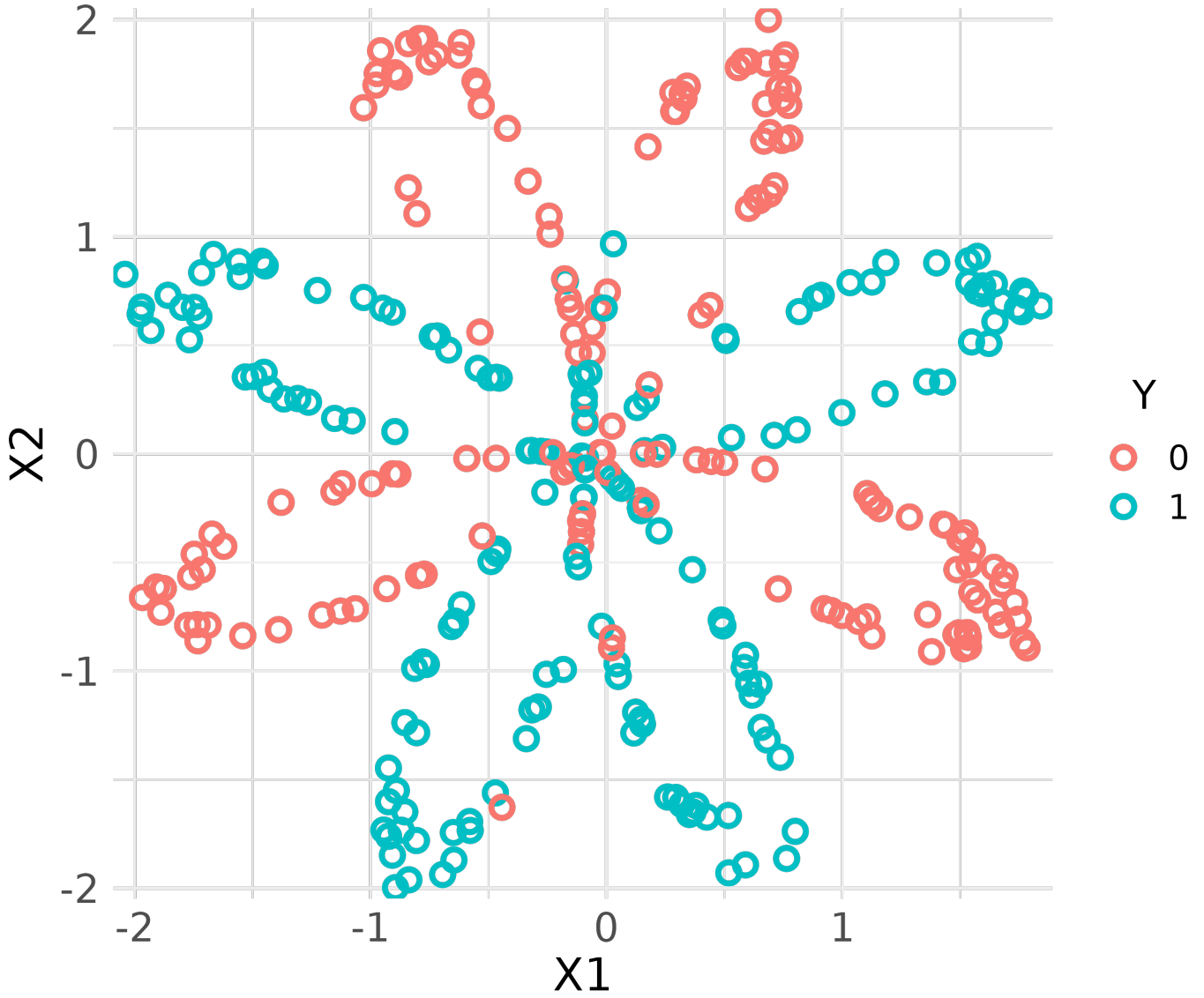
Data Modeling

- data are "molded" to the model
- assumptions about the shape of the relationship
- assumptions about the distributions (of the errors)

Algorithm Modeling

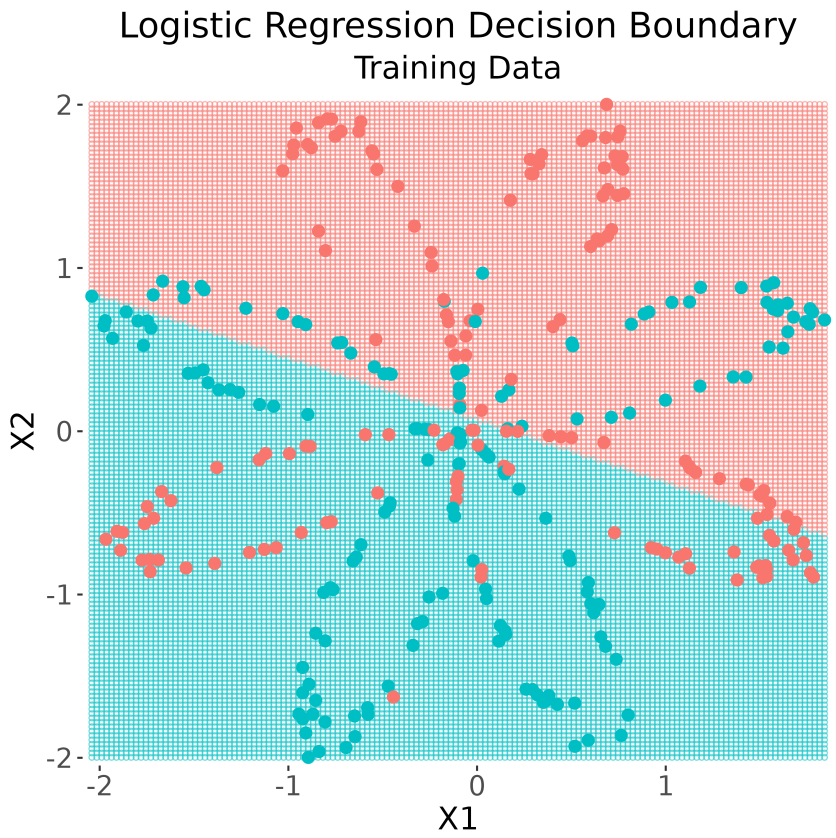
- model is "molded" to the data
- no assumptions about the shape of the relationship (non-linear)
- no assumptions about the distributions (non-parametric)

Unknown Functions of Unknown Complexity



What statistical method would you use to try to model this classification?

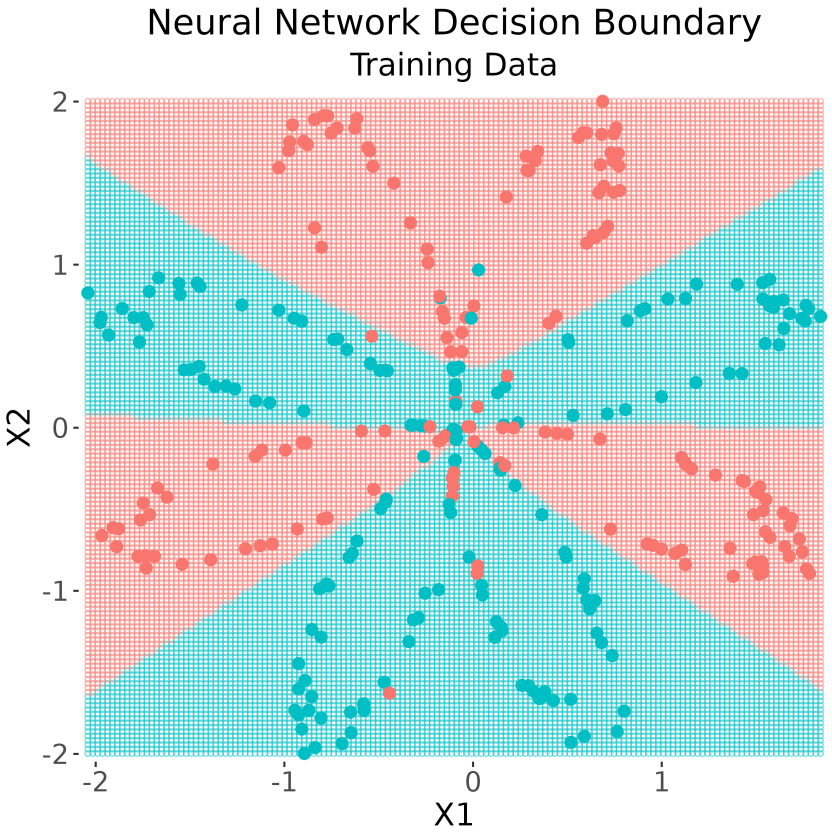
Unknown Functions of Unknown Complexity

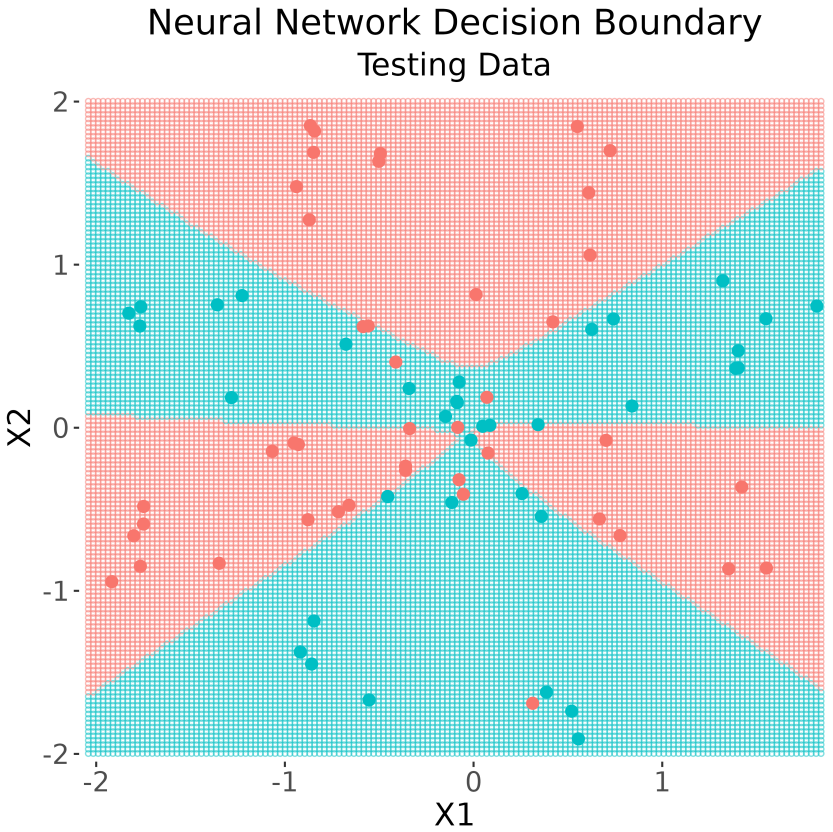


$Acc_{train} = 0.578$

$Acc_{test} = 0.475$

Unknown Functions of Unknown Complexity

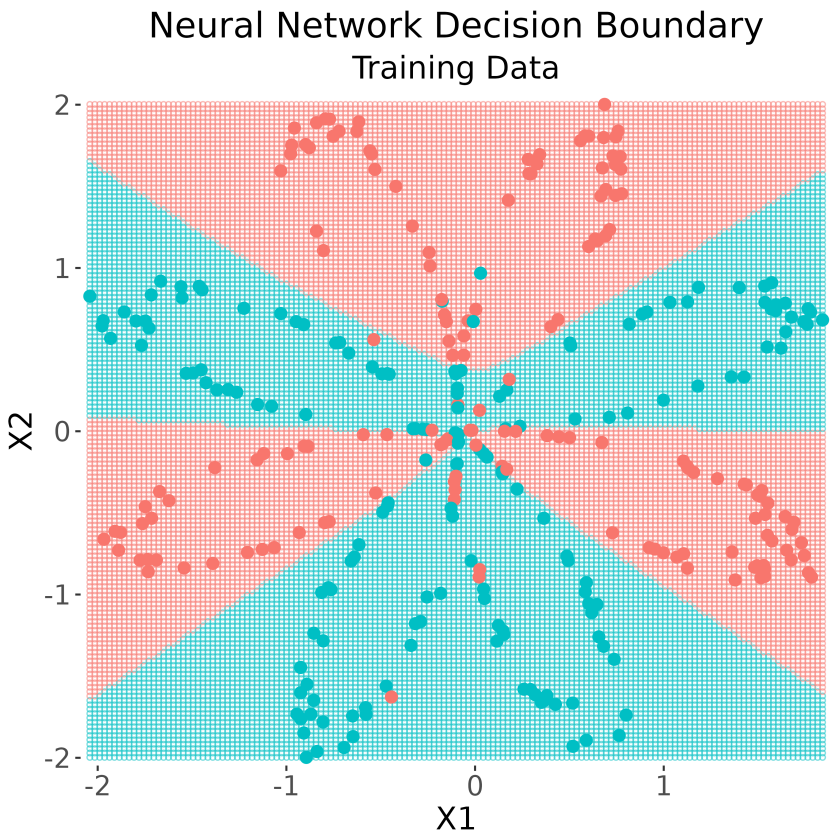


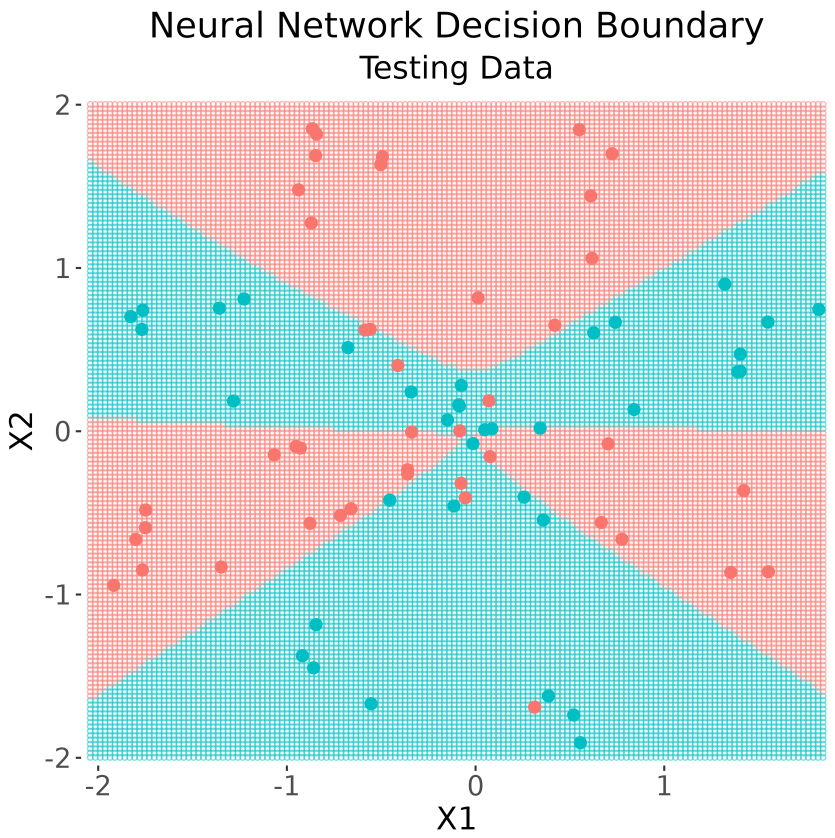


Unknown Functions of Unknown Complexity

```r
# Set seed
set.seed(42)

# Neural network
network_train <- train_net(
  X = df_train[,c("X1", "X2")], # predictors
  y = df_train["Y"], # outcome
  iterations = 100000, # iterations
  hidden_neurons = 6, # neurons in hidden layer
  learning_rate = 0.05 # learning rate (alpha)
)
```

Unknown Functions of Unknown Complexity





$Acc_{train} = 0.934$

$Acc_{test} = 0.863$

(Artificial) Neural Networks

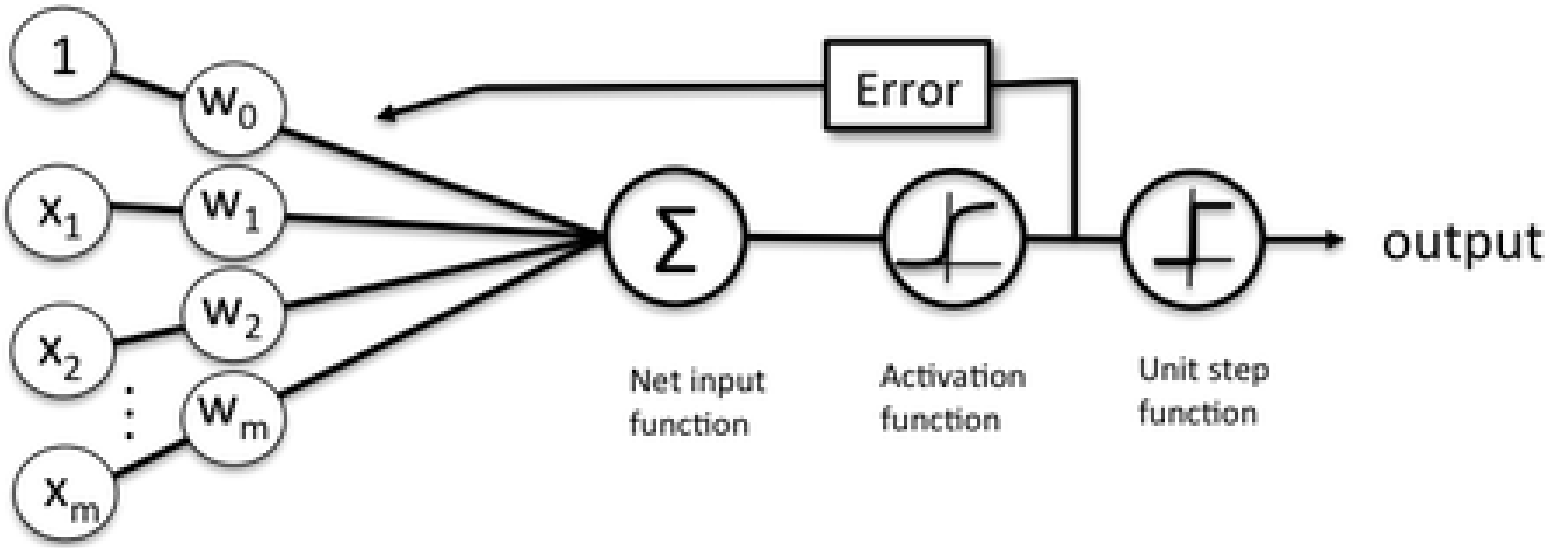
Neural Networks

- Perhaps the "premiere" model associated with data science
- Why are they so popular?
  - General purpose learners: can learn *any* function/relationship between **X** and **Y**
  - Different "architectures" afford flexible applications
  - Sound cool, an air of mystery
- We're going to peel back their layers, one-by-one, and understand how they work *exactly*

Neural Networks

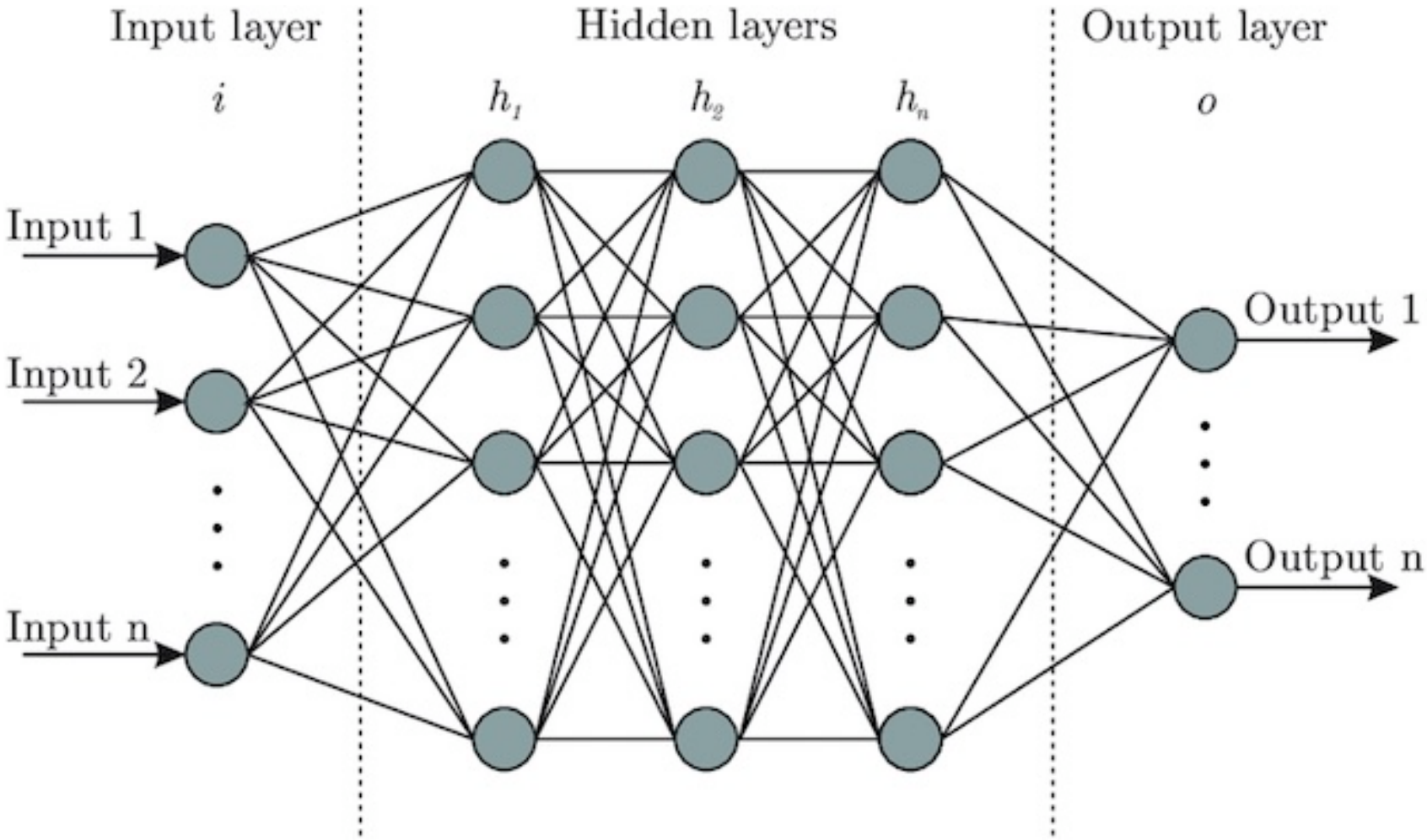
But first. . .

Neural Networks



Schematic of a logistic regression classifier.

Neural Networks



General Process

0. Architecture
1. Initialize parameters
2. Forward propagation
3. Compute cost
4. Backward propagation (backprop)
5. Update parameters
6. Repeat 2-5 until iterations or convergence

Neural Networks

```r
# 0. Architecture
train_net <- function(X, y, iterations, hidden_neurons, learning_rate){

  # 1. Initialize parameters
  parameters <- initialize_parameters(X, y, hidden_neurons)

  # 6. Repeat 2-5 until iterations or convergence
  for(i in 1:iterations){

    # 2. Forward propagation
    fp_output <- forward_propagation(X, parameters)

    # 3. Compute cost
    cost <- computeCost(y, fp_output$A2)

    # 4. Backward propagation (backprop)
    bp_output <- backpropagation(X, y, fp_output, parameters, hidden_neurons)

    # 5. Update parameters
    parameters <- update_parameters(bp_output, parameters, learning_rate)

  }

}
```

0. Architecture

Define the Problem

- **X** = predictor variables
- **Y** = outcome variable(s)
  - 2 categories: classification
  - 3 or more categories: multi-class classification
  - continuous: regression

Your outcome variable will guide your **cost** and **activation function** selection

Transform/normalize/standardize **X** (sensitive to scaling)

Define Architecture

- input layer = number of predictor variables in **X**
- output layer = number of outcome variables in **Y**
- hidden layer(s) = *as much an art as a science*
  - more layers (depth) = more neurons (width) = better able to capture complex functions
  - more layers > more neurons
  - 2-3 or more layers = "deep"
  - each layer need not have the same number of neurons

1. Initialize parameters

$$\mathbf{Z} = \mathbf{XW} + \mathbf{b}$$

- $\mathbf{X}$ = predictors (input variables)
- $\mathbf{W}$ = weights (e.g., regression coefficients)
- $\mathbf{b}$ = bias (e.g., intercepts)

Initialize **W** and b with small values (uniform between 0 and 1 is simplest)

```r
# 1. Initialize parameters
initialize_parameters <- function(X, y, hidden_neurons){

  # Get number of variables
  variables <- dim(X)[2]

  # Get number of output
  output <- dim(y)[2]

  # Return initialization
  return(
    list(
      "W1" = matrix(
        runif(variables * hidden_neurons), nrow = variables,
        ncol = hidden_neurons, byrow = TRUE
      ) * 0.01,
      "b1" = matrix(rep(0, hidden_neurons), ncol = hidden_neurons),
      "W2" = matrix(
        runif(output * hidden_neurons), nrow = hidden_neurons,
        ncol = output, byrow = TRUE
      ) * 0.01,
      "b2" = matrix(rep(0, output), ncol = output)
    )
  )

}
```

Parameters for input layer $\rightarrow$ hidden layer

$\mathbf{W}_1$ = matrix size of p variables $\times$ h hidden neurons with uniform random values

$\mathbf{b}_1$ = matrix size of $1 \times h$ hidden neurons with uniform random values

Parameters for hidden layer $\rightarrow$ output layer

$\mathbf{W}_2$ = matrix size of h hidden neurons $\times$ o outcome variables with uniform random values

$\mathbf{b}_2$ = matrix size of $1 \times o$ outcome variables with uniform random values

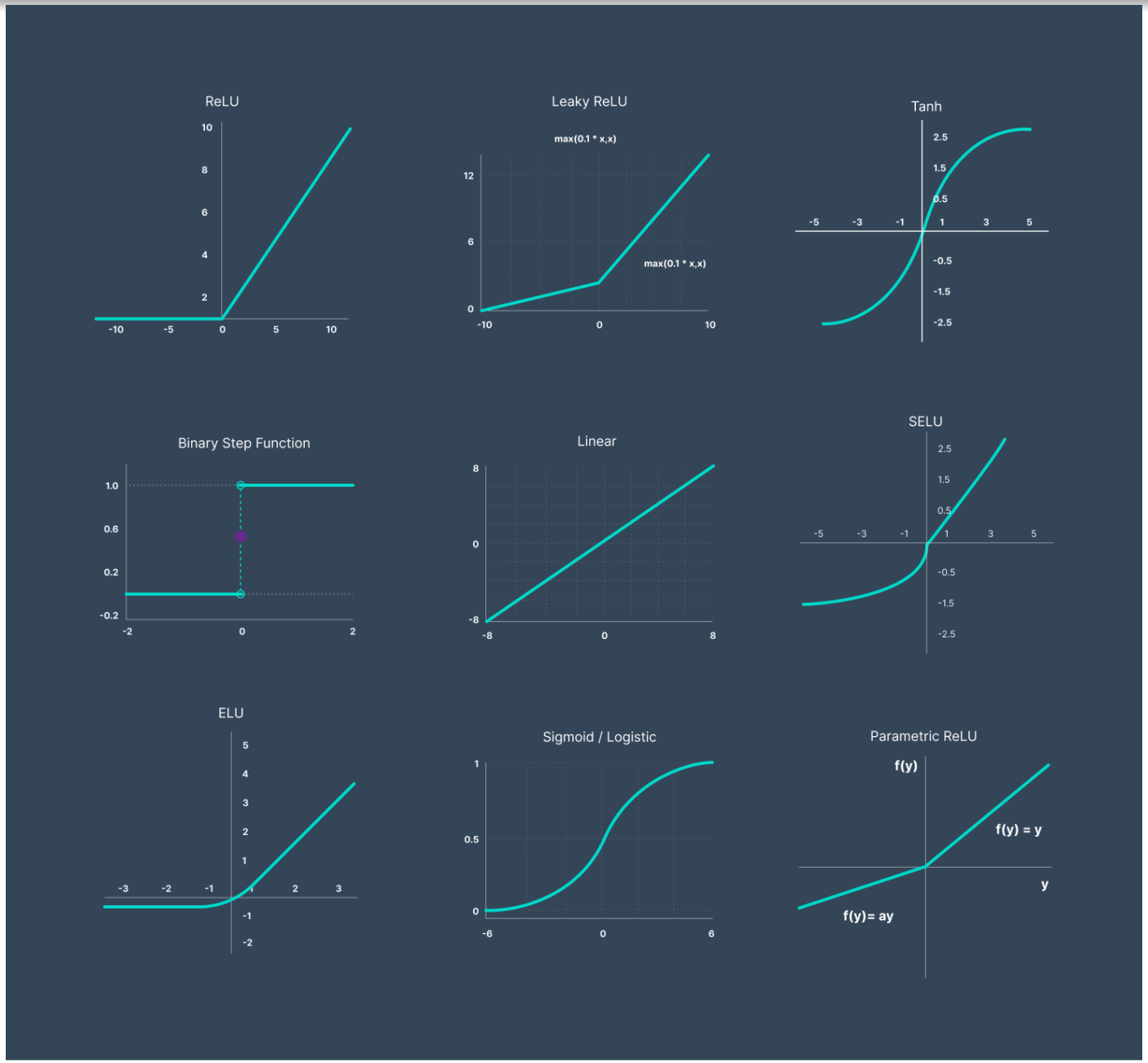
2. Forward propagation

Activation Functions

$$\mathbf{A} = \sigma(\mathbf{Z})$$

- $\sigma()$: activation function
- linear (identity): x
- sigmoid: $\frac{1}{1+e^{-x}}$
- rectified linear unit (ReLU): $\max(0, x)$
- exponential linear unit (ELU): $\begin{cases} x & \text{for } x \geq 0 \\ \alpha(e^x - 1) & \text{for } x < 0 \end{cases}$

Many and more functions (see here)


ReLU
10
8
6
4
2
-10
-5
0
5
10
Leaky ReLU
max(0.1 * x,x)
12
6
0
-10
0
10
max(0.1 * x,x)
Tanh
2.5
1.5
0.5
-5
-3
-1
1
3
5
-0.5
-1.5
-2.5
Binary Step Function
1.0
0.6
0.2
-0.2
-2
0
2
Linear
8
0
-8
-8
0
8
SELU
2.5
1.5
0.5
-5
-3
-1
1
3
5
-0.5
-1.5
-2.5
ELU
5
4
3
2
1
-3
-2
-1
1
2
3
-1
-2
Sigmoid / Logistic
1
0.5
0
-6
0
6
Parametric ReLU
f(y)
f(y) = y
y
f(y)= ay

```r
# 2. Forward propagation
forward_propagation <- function(X, parameters){

  # Compute output from input layer through hidden layer
  Z1 <- sweep(
    X %*% parameters$W1, # matrix multiply data by weights
    MARGIN = 2, STATS = parameters$b1, # add bias (intercept)
    FUN = "+"
  )

  # Apply activation function to output
  A1 <- sigmoid(Z1)

  # Compute output from hidden layer to output layer
  Z2 <- sweep(
    A1 %*% parameters$W2, # matrix multiply data by weights
    MARGIN = 2, STATS = parameters$b2, # add bias (intercept)
    FUN = "+"
  )

  # Apply activation function to output
  A2 <-  sigmoid(Z2)

  # Return parameters
  return(
    list(
      Z1 = Z1, A1 = A1,
      Z2 = Z2, A2 = A2
    )
  )

}
```

$\mathbf{Z}_1 = \mathbf{X}\mathbf{W}_1 + \mathbf{b}_1$ or output of input layer $\rightarrow$ hidden layer

$\mathbf{A}_1 =$ activation function (sigmoid) applied to values of $\mathbf{Z}_1$

$\mathbf{Z}_2 = \mathbf{A}_1\mathbf{W}_2 + \mathbf{b}_1$ or output of hidden layer $\rightarrow$ output layer

$\mathbf{A}_2 =$ activation function (sigmoid) applied to values of $\mathbf{Z}_2$

3. Compute cost

Costs

Regression: **?**

Costs

Regression: mean square error $= \frac{\sum(\hat{\mathbf{y}}-\mathbf{y})^2}{n}$

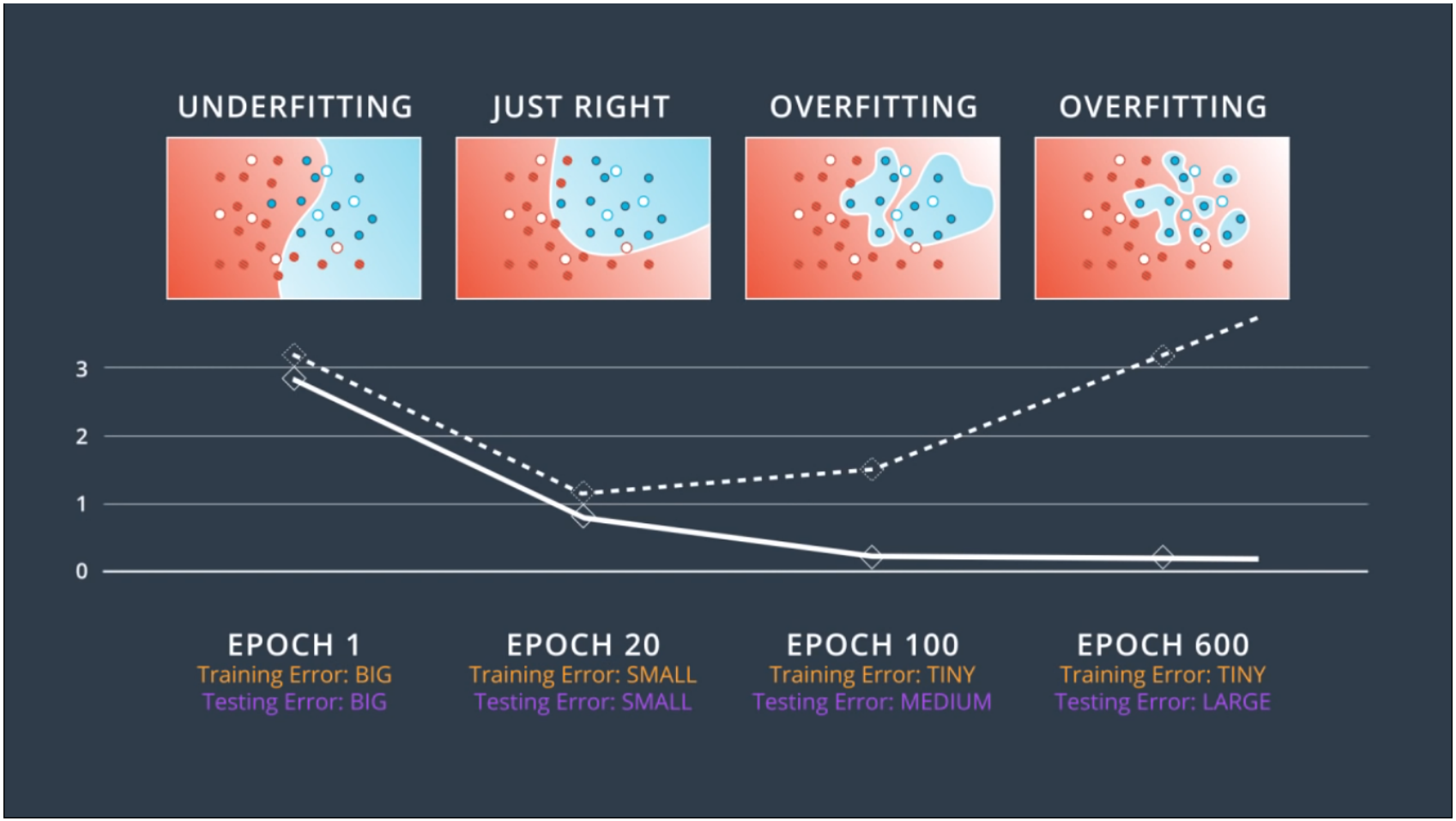
Classification: **?**

Costs

Regression: mean square error $= \frac{\sum(\hat{\mathbf{y}}-\mathbf{y})^2}{n}$

Classification: binary cross-entropy $= -\frac{\sum \mathbf{y}\log(\hat{\mathbf{y}})+(1-\mathbf{y})\log(1-\hat{\mathbf{y}})}{n}$

```r
# 3. Cost function
computeCost <- function(y, y_hat){

  # Return binary cross-entropy (add small weight to avoid zeros)
  y_hat <- y_hat + 1e-10
  return(
    -mean((y * log(y_hat)) + ((1 - y) * log(1 - y_hat)))
  )

}
```


UNDERFITTING
JUST RIGHT
OVERFITTING
OVERFITTING
3
2
1
0
EPOCH 1
Training Error: BIG
Testing Error: BIG
EPOCH 20
Training Error: SMALL
Testing Error: SMALL
EPOCH 100
Training Error: TINY
Testing Error: MEDIUM
EPOCH 600
Training Error: TINY
Testing Error: LARGE

4. Backward propagation

- To this point, the neural network is a set of linear regressions with some activation (or link) function applied to its output
- The real "magic" happens here
- The errors in prediction are propagated backward through the model up to the input
- A technical understanding requires calculus (specifically partial derivatives)

Takeaway: errors in prediction are used to adjust the weights and biases down the gradient (or toward better prediction)

```r
# 4. Backward propagation
backpropagation <- function(X, y, fp_output, parameters, hidden_neurons){

  # Get number of cases and output
  cases <- dim(X)[1]; output <- dim(y)[2]

  # Get inverse of cases
  inv_cases <- 1 / cases

  # Difference between prediction and actual (error)
  dZ2 <- fp_output$A2 - y

  # Compute difference for output layer to hidden layer weights
  dW2 <- inv_cases * crossprod(dZ2, fp_output$A1)

  # Compute difference for output layer to hidden layer biases
  db2 <- matrix(inv_cases * sum(dZ2), ncol = output)

  # Difference between hidden layer and output from input layer
  dZ1 <- tcrossprod(dZ2, parameters$W2) * (fp_output$A1 * (1 - fp_output$A1))

  # Compute difference for hidden layer to input layer weights
  dW1 <- inv_cases * crossprod(dZ1, X)

  # Compute difference for hidden layer to input layer biases
  db1 <- matrix(inv_cases * colSums(dZ1), nrow = hidden_neurons)

  # Send back gradients
  return(list("dW1" = dW1, "db1" = db1, "dW2" = dW2, "db2" = db2))

}
```

The process starts by computing the derivative of the cost function (binary cross-entropy) or the "error" between output layer (prediction) and actual values:

$$d\mathbf{Z}^{[2]} = \frac{\partial C}{\partial A^{[2]}} = \mathbf{A}^{[2]} - \mathbf{Y}$$

```
# Difference between prediction and actual (error)
dZ2 <- fp_output$A2 - y
```

Using the error, the gradient of the weights between the output layer and hidden layer is computed (average is taken across the gradient):

$$d\mathbf{W}^{[2]} = \frac{\partial C}{\partial W^{[2]}} = \frac{d\mathbf{Z}^{[2]}\mathbf{A}^{[1]T}}{n}$$

```
# Compute difference for output layer to hidden layer weights
dW2 <- inv_cases * crossprod(dZ2, fp_output$A1)
```

Similarly, using the error, the gradient of the biases between the output layer and hidden layer is computed:

$$d\mathbf{b}^{[2]} = \frac{\partial C}{\partial b^{[2]}} = \frac{\sum_{i=1}^{n} d\mathbf{Z}_i^{[2]}}{n}$$

```
# Compute difference for output layer to hidden layer biases
db2 <- matrix(inv_cases * sum(dZ2), ncol = output)
```

The error of the hidden layer is used to compute the gradient from the hidden layer to input layer (using the derivative of the activation function):

$$d\mathbf{Z}^{[1]} = \frac{\partial C}{\partial A^{[1]}} = (\mathbf{W}^{[2]T} d\mathbf{Z}^{[2]}) * \sigma'^{[1]}(\mathbf{A}^{[1]})$$

```
# Difference between hidden layer and output from input layer
dZ1 <- tcrossprod(dZ2, parameters$W2) * (fp_output$A1 * (1 - fp_output$A1))
```

This gradient is computed with respect to the input variables:

$$d\mathbf{W}^{[1]} = \frac{\partial C}{\partial W^{[1]}} = \frac{d\mathbf{Z}^{[1]}\mathbf{X}^T}{n}$$

```
# Compute difference for hidden layer to input layer weights
dW1 <- inv_cases * crossprod(dZ1, X)
```

Similar to the output layer biases, the gradient of the biases between the hidden layer and input layer is computed (with respect to the input variables):

$$d\mathbf{b}^{[1]} = \frac{\partial C}{\partial b^{[1]}} = \frac{\sum_{i=1}^{n} d\mathbf{Z}_i^{[1]}}{n}$$

```
# Compute difference for hidden layer to input layer biases
db1 <- matrix(inv_cases * colSums(dZ1), nrow = hidden_neurons)
```

Gradient

$$\frac{\partial C}{\partial \mathbf{A}^{[l-1]}} = \left[\mathbf{W}^{[l]}\right] \cdot \frac{\partial C}{\partial \mathbf{A}^{[l]}} * \sigma'^{[l-1]}(\mathbf{A}^{[l-1]})$$

Weights

$$\frac{\partial C}{\partial \mathbf{W}^{[l]}} = \frac{\partial C}{\partial \mathbf{A}^{[l]}} \cdot \left[\mathbf{X}^{[l-1]}\right]^T$$

Biases

$$\frac{\partial C}{\partial \mathbf{b}^{[l]}} = \frac{\partial C}{\partial \mathbf{A}^{[l]}}$$

Gradients for input layer $\leftarrow$ hidden layer

$d\mathbf{Z}_1$ = gradient for $\mathbf{Z}_1$

$d\mathbf{b}_1$ = gradient for $\mathbf{b}_1$

Gradients for hidden layer $\rightarrow$ output layer

$d\mathbf{Z}_2$ = gradient for $\mathbf{Z}_2$

$d\mathbf{b}_2$ = gradient for $\mathbf{b}_2$

5. Update parameters

After backpropagation, the parameters can now be updated

Remember gradient descent?

Backpropagation computes the gradient but does not determine how those values are used

The parameters still need to be updated

```r
# 5. Update parameters
update_parameters <- function(bp_output, parameters, learning_rate){

  # Return updated parameters
  return(
    list(
      "W1" = parameters$W1 - learning_rate * t(bp_output$dW1),
      "b1" = parameters$b1 - learning_rate * t(bp_output$db1),
      "W2" = parameters$W2 - learning_rate * t(bp_output$dW2),
      "b2" = parameters$b2 - learning_rate * bp_output$db2
    )
  )

}
```

Weights

$$\mathbf{W}_{s+1}^{[l]} = \mathbf{W}_{s}^{[l]} - \alpha(d\mathbf{W}_{s}^{[l]}),$$

Biases

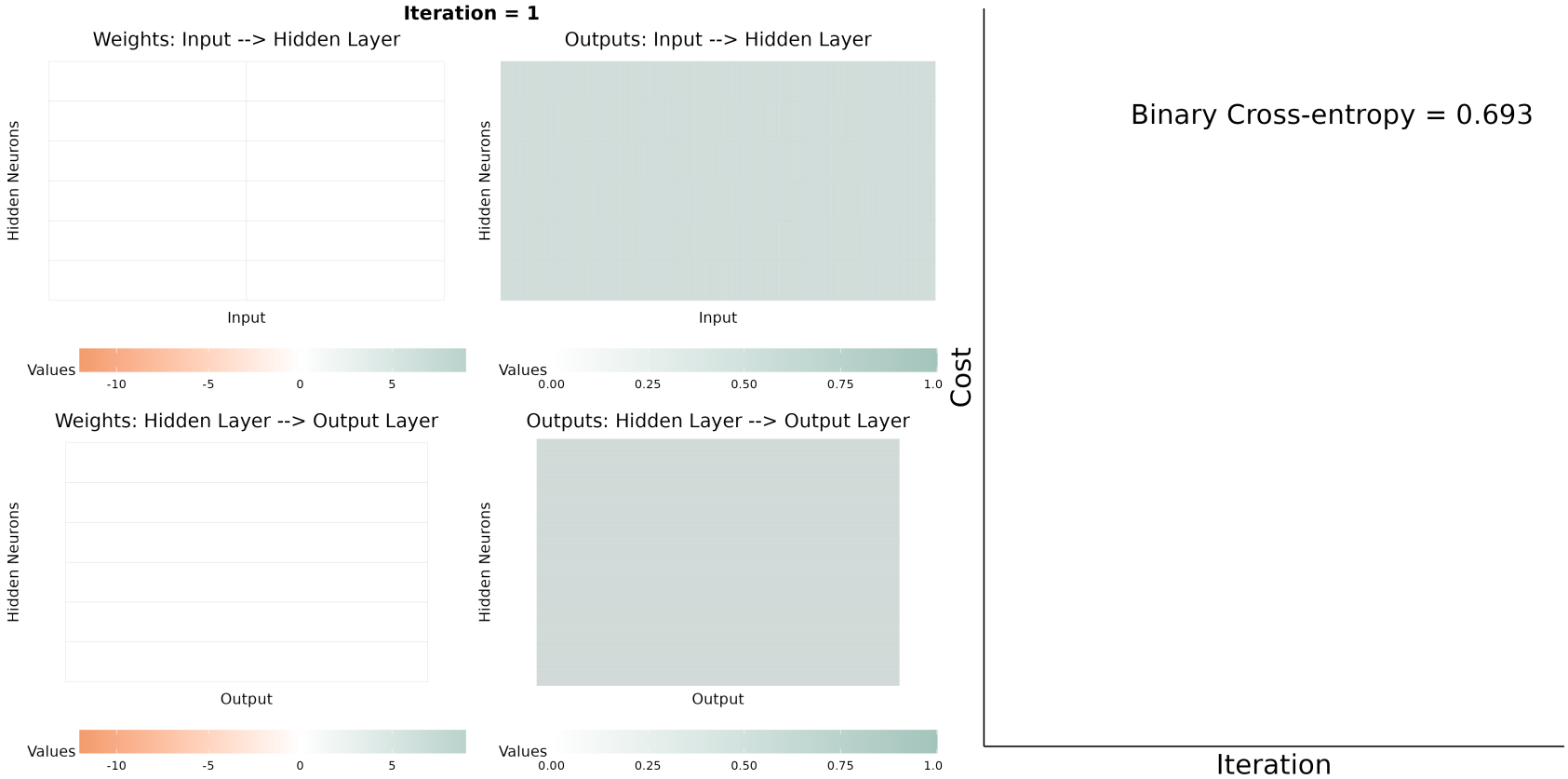
$$\mathbf{b}_{s+1}^{[l]} = \mathbf{b}_{s}^{[l]} - \alpha(d\mathbf{b}_{s}^{[l]}),$$

where $s + 1$ represents the updated parameters for the epoch/batch and s represents the original parameters for the epoch/batch

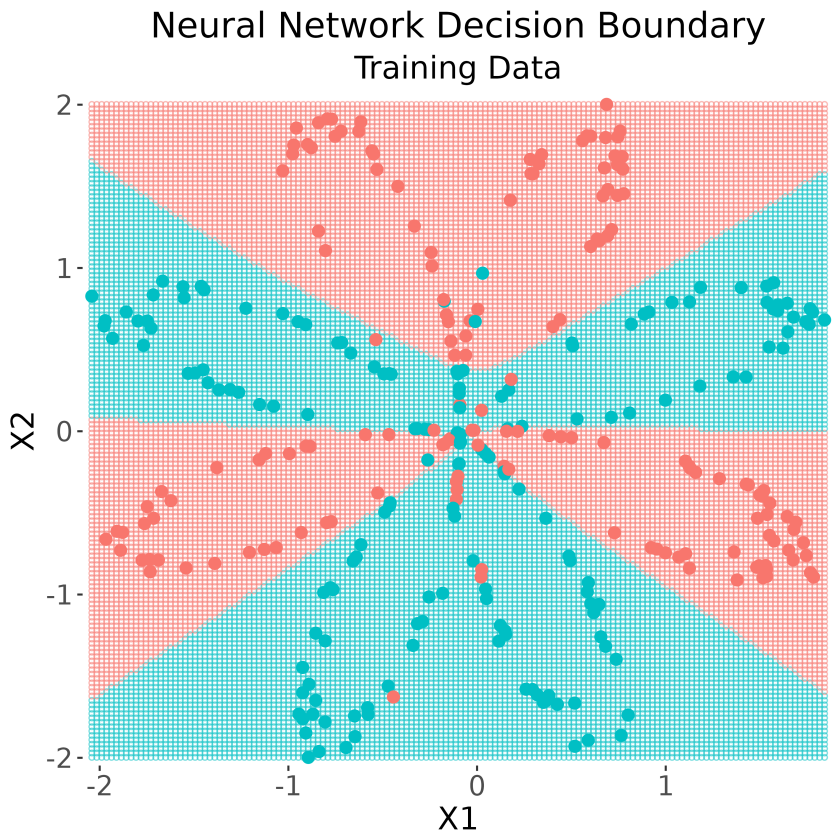
General Process

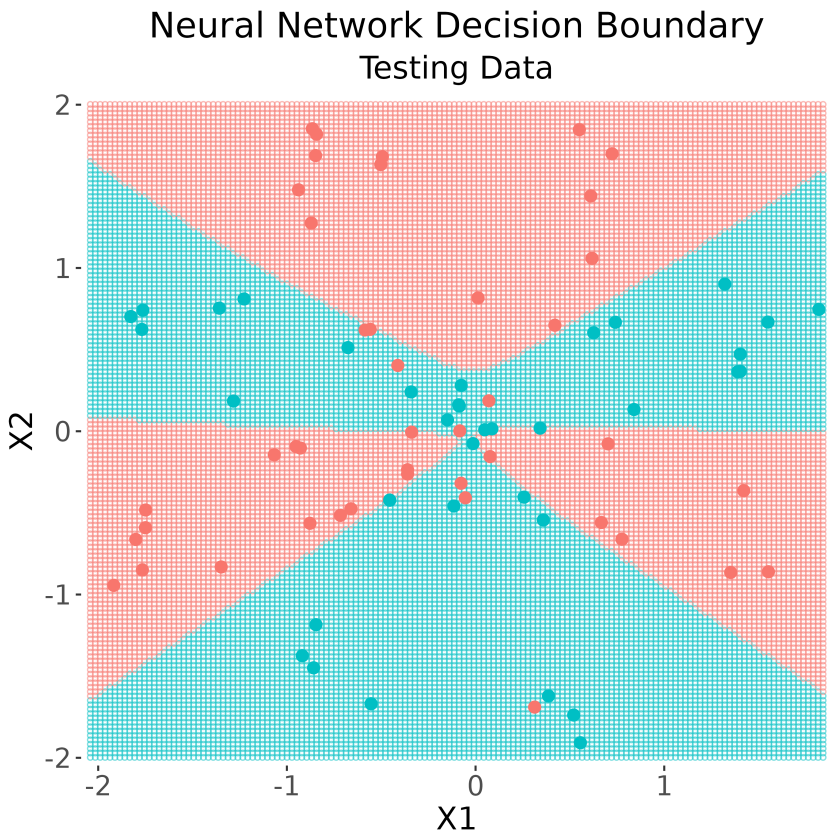
0. Architecture
1. Initialize parameters
2. Forward propagation
3. Compute cost
4. Backward propagation (backprop)
5. Update parameters
6. Repeat 2-5 until iterations or convergence

Neural Networks



Applied from Scratch





$Acc_{train} = 0.934$

$Acc_{test} = 0.863$

Relative Importance

Recall

relative importance: change in metric after permutating each variable individually

Use metric that you used for evaluation to get most interpretable importance

Most R package computations will produce values from 0-100 (these aren't as interpretable as my opinion)

```r
# Set seed
set.seed(42)

## Create copy
df_copy <- df_train

## Initialize store
store_accuracy_delta <- numeric(10) # number of iterations

# Accuracy
train_accuracy <- mean(network_class_train == df_train$Y)

## Shuffle and get prediction values
for(i in seq_along(store_accuracy_delta)){

  # Replace "X1" in copy
  df_copy$X1 <- sample(df_train$X1)

  # Get class
  predicted_class <- ifelse(
    predict(network_train, newdata = df_copy) > 0.50, 1, 0
  )

  # Get accuracy
  store_accuracy_delta[i] <- train_accuracy - mean(predicted_class == df_train$Y)

}

# Get mean of differences
mean(store_accuracy_delta)
```

$\mathbf{X}_1 = 0.236$ or a 23.6% drop in accuracy

$\mathbf{X}_2 = 0.434$ or a 43.4% drop in accuracy

About 2 times the drop in accuracy for $\mathbf{X}_2$

Applied with {keras}

{keras} and {tensorflow} offer an easy-to-use and extensive framework for training neural networks in R

- Layers
- Activation functions
- Optimizers
- Gradient descent
- Normalization, regularization, and dropout

Layers

```
# Set seed
tensorflow::set_random_seed(1234)

# Set up model
model <- keras_model_sequential() %>%
  layer_dense( # input layer => hidden layer
    units = 6, # number of neurons in first hidden layer
    input_shape = 2, # number of predictors
    activation = "sigmoid" # activation function
  ) %>%
  layer_dense( # hidden layer => output layer
    units = 1, # one output
    activation = "sigmoid"
    # classification (2 classes) = "sigmoid"
    # multi-class (3 or more classes) = "softmax"
    # regression = "linear"
  )
```

Layers

```
# Set up model
model <- keras_model_sequential() %>%
  layer_dense( # input layer => first hidden layer
    units = 6, input_shape = 2, activation = "sigmoid"
  ) %>%
  layer_dense( # first hidden layer => second hidden layer
    units = 10, activation = "sigmoid"
  ) %>%
  layer_dense( # second hidden layer => third hidden layer
    units = 14, activation = "sigmoid"
  ) %>%
  layer_dense( # third hidden layer => output layer
    units = 1, activation = "sigmoid"
  )
```

Layers

- More layers
  - **pro**: model more complex functions between **X** and **Y**
  - **con**: prone to overfitting or "memorization" of training data

Activation functions

```
# Set up model
model <- keras_model_sequential() %>%
  layer_dense( # input layer => hidden layer
    units = 6, input_shape = 2,
    activation = "relu" # activation function
  ) %>%
  # more hidden layers...
  layer_dense( # hidden layer => output layer
    units = 1, activation = "sigmoid"
    # classification (2 classes) = "sigmoid"
    # multi-class (3 or more classes) = "softmax"
    # regression = "linear"
  )
```

Activation functions

- Different functions might work better for different problems
- Some activation functions work better with different initialization of weights
- Most commonly, the REctified Linear Unit (RELU) is used ($\max(0, x)$)
- List available in {keras}

Optimizers

```
# Set up backpropagation
model <- model %>% compile(
  loss = "binary_crossentropy", # set loss function
  optimizer = optimizer_nadam(learning_rate = 0.05), # learning rate
  # HML likes `optimizer_rmsprop`
  metrics = "accuracy" # evaluation metric
)
```

Optimizers

- Change the way in which the parameters of gradient are used
  - learning rate = magnitude of gradient used
  - decay = decreases in learning rate over time
  - momentum = resistance to perturbations away from gradient
- List available in {keras}

Gradient descent

```
# Train the model
keras_training <- model %>%
  fit(
    x = as.matrix(df_train[,c("X1", "X2")]), # predictors
    y = as.numeric(df_train[,c("Y")]) - 1, # outcome
    epochs = 1000, # number of iterations
    batch_size = 8, # stochastic gradient descent
    validation_split = 0.20 # 80/20
  )
```

Gradient descent

- `epochs` = number of runs through entire training data
- `batch_size` = number of observations used before using the gradient to update the parameters
  - stochastic gradient descent (SGD): updates *every* observation
  - mini-batch: updates on the batches (2^n are common values: 4, 8, 16, 32, 64, 128, etc.)
- `validation_split`: data held out to test generalizability during training

Normalization

```
# Set up model
model <- keras_model_sequential() %>%
  layer_dense( # input layer => hidden layer
    units = 6, input_shape = 2, activation = "relu"
  ) %>%
  layer_normalization() %>% # normalization
  # more hidden layers...
  # perhaps more normalization...
  layer_dense( # hidden layer => output layer
    units = 1, activation = "sigmoid"
  )
```

Maintains output of previous layer with mean $= 0$ and standard deviation $= 1$

purpose: avoid "exploding" gradient and outlier neuron influence

Regularization

```
# Set up model
model <- keras_model_sequential() %>%
  layer_dense( # input layer => hidden layer
    units = 6, input_shape = 2, activation = "relu"
  ) %>%
  layer_activity_regularization(l2 = 0.01) %>% # l2-norm regularization
  # more hidden layers...
  # perhaps more regularization...
  layer_dense( # hidden layer => output layer
    units = 1, activation = "sigmoid"
  )
```

Shrinks parameters of the output

purpose: avoid "exploding" gradient and outlier neuron influence

Dropout

```
# Set up model
model <- keras_model_sequential() %>%
  layer_dense( # input layer => hidden layer
    units = 6, input_shape = 2, activation = "relu"
  ) %>%
  layer_dropout(rate = 0.333) %>% # dropout
  # more hidden layers...
  # perhaps more dropout...
  layer_dense( # hidden layer => output layer
    units = 1, activation = "sigmoid"
  )
```

"Drops out" a proportion of neurons randomly from the previous layer

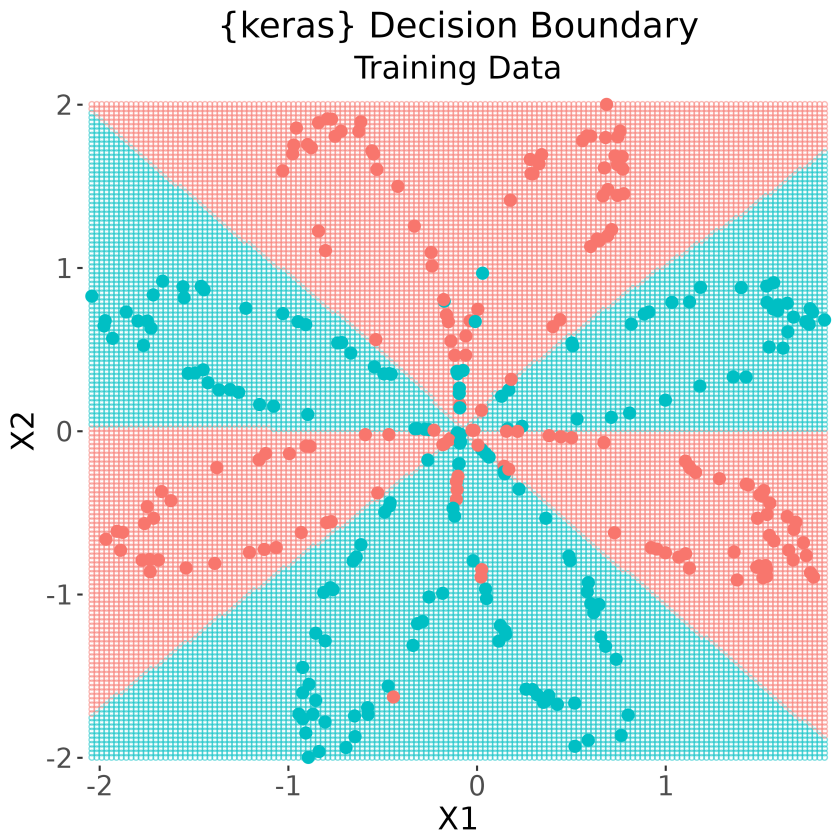
purpose: avoid "exploding" gradient, outlier neuron influence, and better learn relationships across all neurons

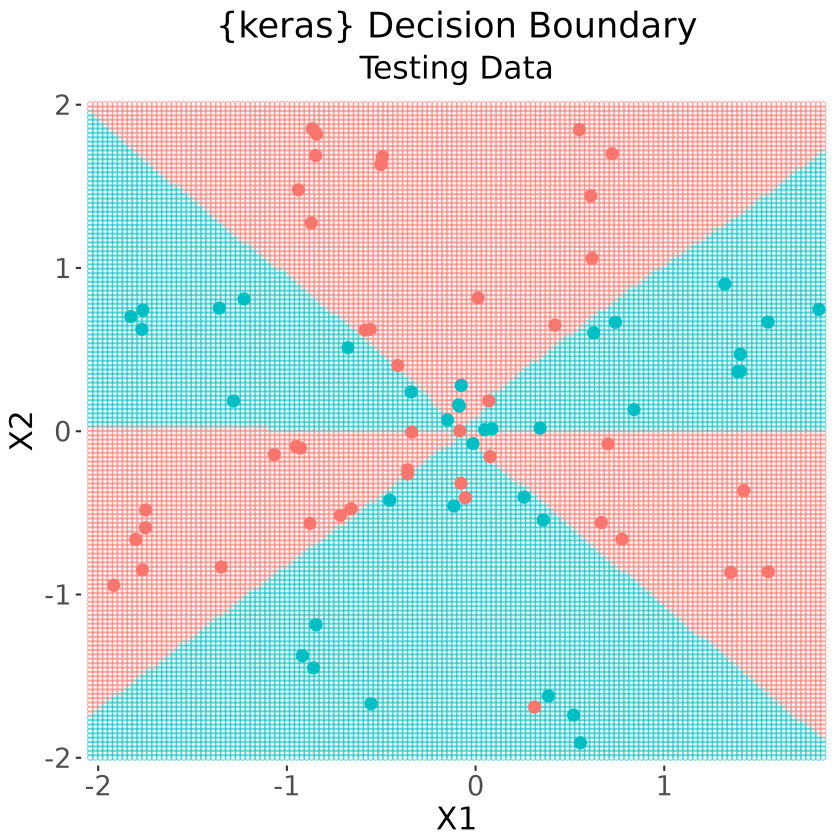
Template: Architecture

```
# Set seed
tensorflow::set_random_seed(1234)

# Set up model
model <- keras_model_sequential() %>%
  layer_dense( # input layer => hidden layer
    units = 6, # number of neurons in first hidden layer
    input_shape = 2, # number of predictors
    activation = "sigmoid" # activation function
  ) %>%
  # more hidden layers and/or normalization/regularization/dropout...
  layer_dense( # hidden layer => output layer
    units = 1, # number of output
    activation = "sigmoid"
    # classification (2 classes) = "sigmoid"
    # multi-class (3 or more classes) = "softmax"
    # regression = "linear"
  )
```

Template: Backpropagation and Training

```
# Set up backpropagation
model <- model %>% compile(
  loss = "binary_crossentropy", # set loss function
  optimizer = optimizer_adam(learning_rate = 0.001), # learning rate
  metrics = "accuracy" # evaluation metric
)

# Train the model
keras_training <- model %>%
  fit(
    x = as.matrix(df_train[,c("X1", "X2")]), # predictors
    y = as.numeric(df_train[,c("Y")]) - 1, # outcome
    epochs = 1000, # number of iterations
    batch_size = 16, # mini-batch
    validation_split = 0.20 # 80/20
  )
```





$Acc_{train} = 0.941$

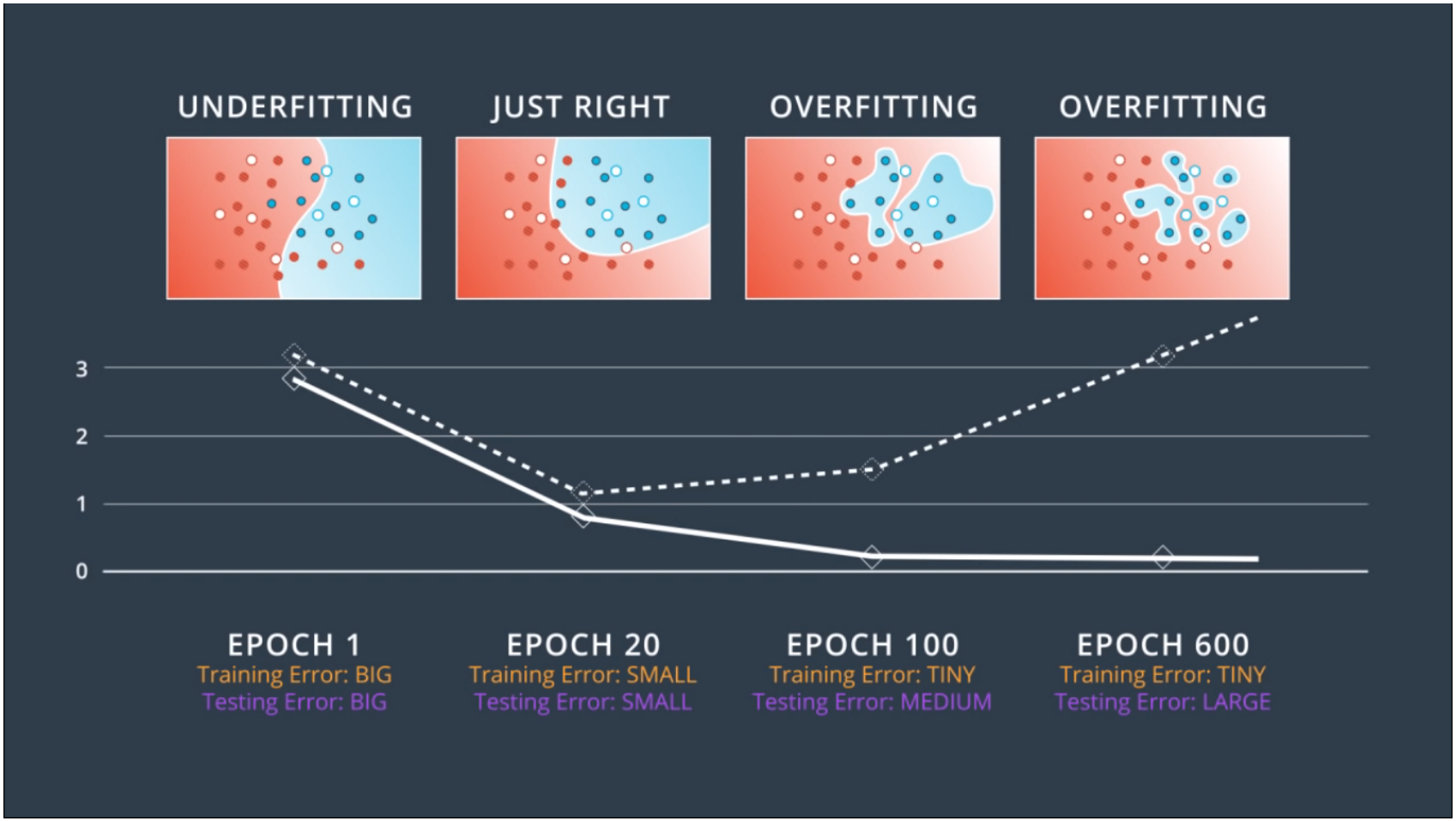
$Acc_{test} = 0.900$

Training Tips

- Optimizer can significantly impact loss curve and overall prediction
- Depth (more hidden layers) over width (more hidden nodes)
- Binary cross-entropy loss should start around 0.69 under simple settings
- Things to try
  - Different optimizers: `optimizer_rmsprop` or `optimizer_adam`
  - Different activation functions: `"sigmoid"`, `"relu"`, `"gelu"`
  - Different batch sizes: `8`, `16`, `32`, `64` (can increase with sample size)
  - Different learning rates: `0.1`, `0.01`, `0.001`, `0.0001`

Make one change at a time

Pay attention to your loss curves!



- Neural Network Recipe
- Cheat Sheet
- Quora Pointers

At Home Activity

At Home Activity

- Use {keras} and `kaggle_train.csv` to train a neural network to classify alcohol use disorder (`"TARGET"`)
- Some things I've already taken care of for you:
  - feature selection (you can use as few or as many variables as you'd like)
  - scaling/standardization
  - script template (see "kaggle_demo.R")
- Turn predictions of `kaggle_test.csv` into Kaggle
- 10 initial points will be given for turning in a model on time
- The other 4 points will depend on how accurately you can predict alcohol use disorder (by the end of the semester: 04/16 at 11:59:59pm)

Readings for Next Week

Readings

- ESL Chapters: 8.7, 9.2, 10.1-10.9, and 15
- HML: Chapter 9-12
- Fife and D'Onofrio - 2023

Optional

- Strobl et al. - 2009
- Goretzko and Bühner - 2020
- Breiman - 2001 - random forest

Transformers

Transformers

Transformers use feed-forward networks with one common architecture being:

```
# Transformer architecture
model <- keras_model_sequential() %>%
  layer_dense(units = h, input_shape = p, activation = "gelu") %>%
  layer_normalization() %>%
  layer_dense(units = h, activation = "gelu") %>%
  layer_normalization() %>%
  # more hidden layers...
  # more normalization...
  layer_dense(units = o, activation = "softmax")
```

For more details, see https://bbycroft.net/llm