Introduction to Neural Networks

Ryan Miller

Grinnell College
Statistics

Logistic Regression

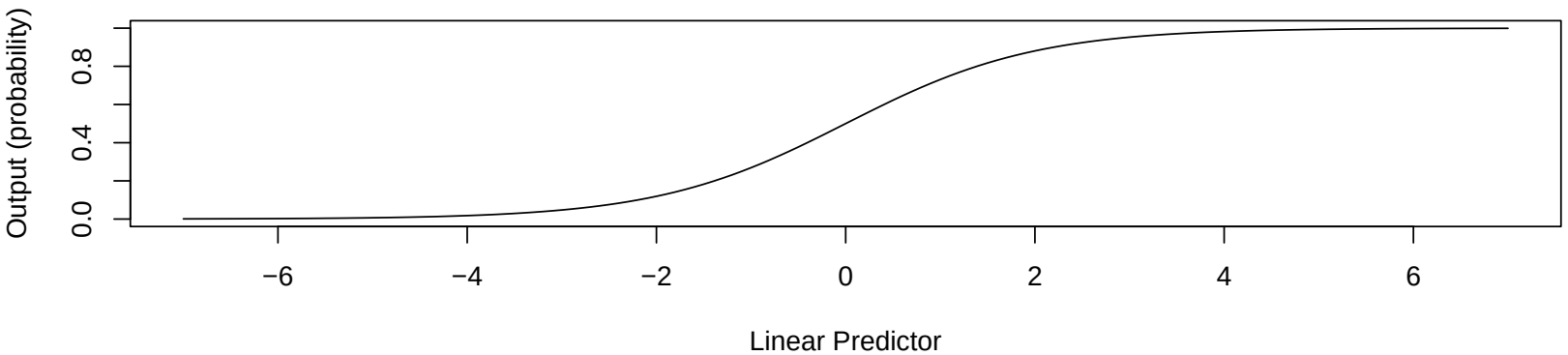
Logistic regression uses a set of features, $X_1, \ldots, X_p$, to predict a binary outcome, Y, using the following structure:

$$y_i = \text{Bernoulli}(g(z_i)) \text{ where } g(z_i) = \frac{1}{1+exp(-z_i)}$$

- $z_i = \hat{w}_0 + \hat{w}_1 x_{i1} + \hat{w}_2 x_{i2} + \ldots + \hat{w}_p x_{ip}$ is the *linear predictor* for the i^{th} observation
- The Bernoulli distribution is used as the underlying probability model (source of error)

Logistic Regression

The function $g(z_i) = \frac{1}{1+exp(-z_i)}$ is known as the **sigmoid function**:



This function maps a numeric input (the linear predictor) to a value between 0 and 1.

Logistic Regression

The model's weights, $\{w_0, w_1, \ldots, w_p\}$, are found by optimizing the cross-entropy cost function (log-loss):

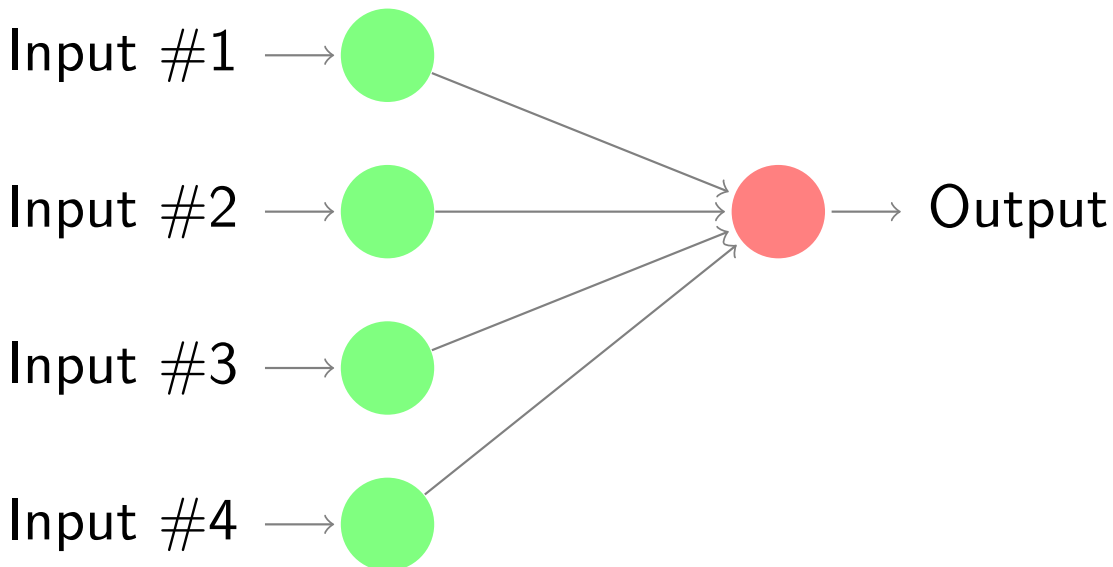
$$\text{Cost} = -\frac{1}{n}\sum_{i=1}^{n}\left(y_i log(g(z_i)) + (1-y_i) log(1-g(z_i))\right)$$

This optimization relies upon differentiating the cost function with respect to the unknown weights, which we can express using chain rule:

$$Gradient = \frac{\partial Cost}{\partial g} \cdot \frac{\partial g}{\partial \mathbf{z}} \cdot \frac{\partial \mathbf{z}}{\partial \mathbf{w}}$$

Logistic Regression

We can represent logistic regression in the following diagram, where a weight combination of inputs goes into the sigmoid function (red dot) which produces the output:



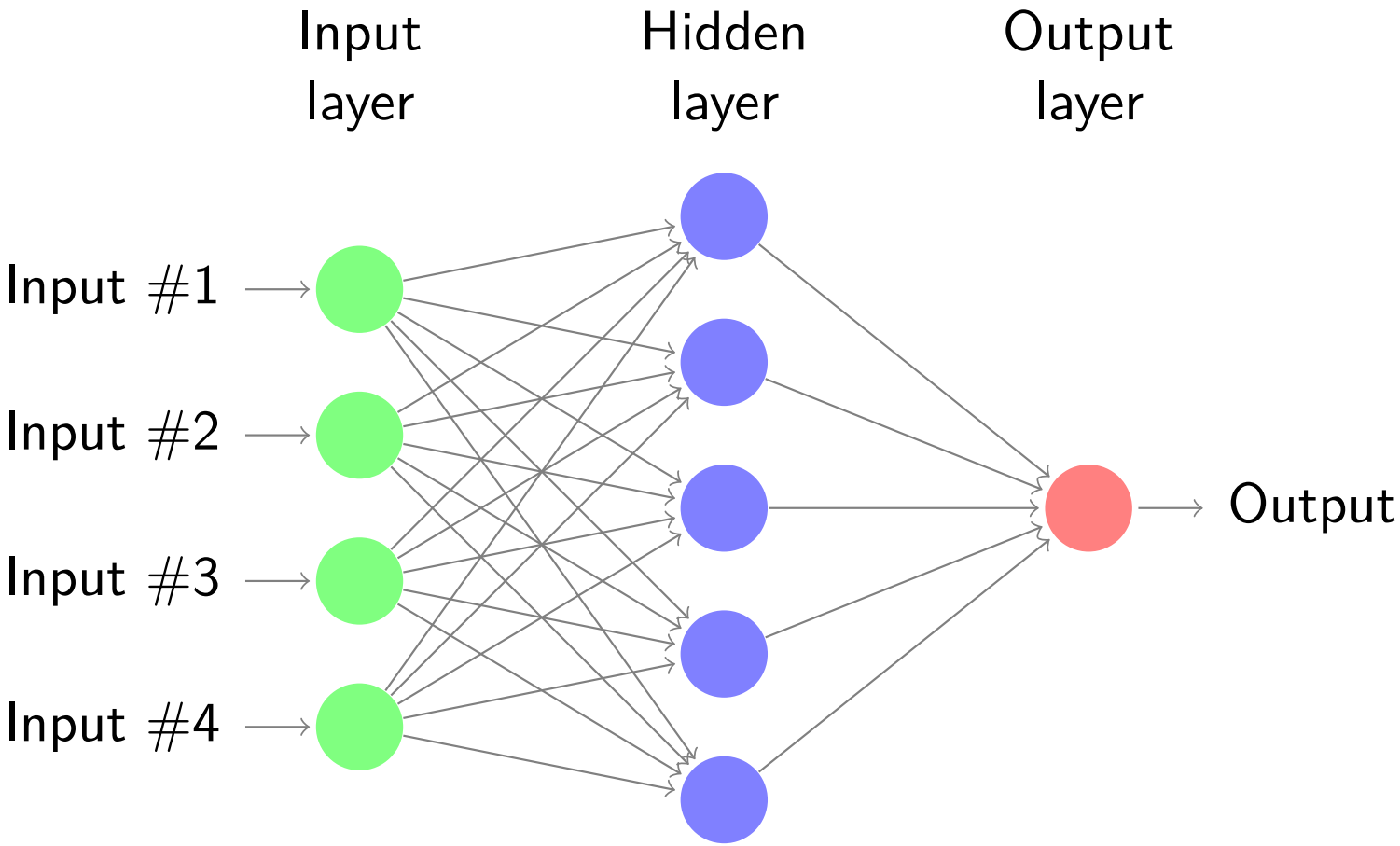
Logistic Regression is a Neural Network

Logistic regression is a special case of a neural network that uses:

- No hidden layers
- One output neuron
- The sigmoid activation function

Neural networks extend logistic regression by introducing *hidden layers* that allow for nonlinearity and interactions between features.

Neural Network with One Hidden Layer



Notation

Because neural networks can contain many hidden layers, we'll introduce the following notation to keep track of the model's structure:

- $\mathbf{x}_i$ denotes a p-dimensional vector of input features (the i^{th} row in our data, assuming a tabular format)
- Superscripts, such as $\mathbf{w}^{(1)}$, will indicate the layer of object
- $\mathbf{z}^{()}$ will indicate the linear combination of weights and inputs in a particular layer
- $\mathbf{a}^{()}$ will indicate the activated output of a particular layer
- b will be used to indicate bias terms in linear combinations

Activation Functions

Each neuron involves two steps:

1. computing a weighted combination of inputs
   $z_i = \hat{w}_0 + \hat{w}_1 x_{i1} + \hat{w}_2 x_{i2} + \ldots$
2. applying an activation function $a_i = g(z_i)$

The activation function introduces non-linearity, without it a sequence of linear layers would collapse back into a linear model

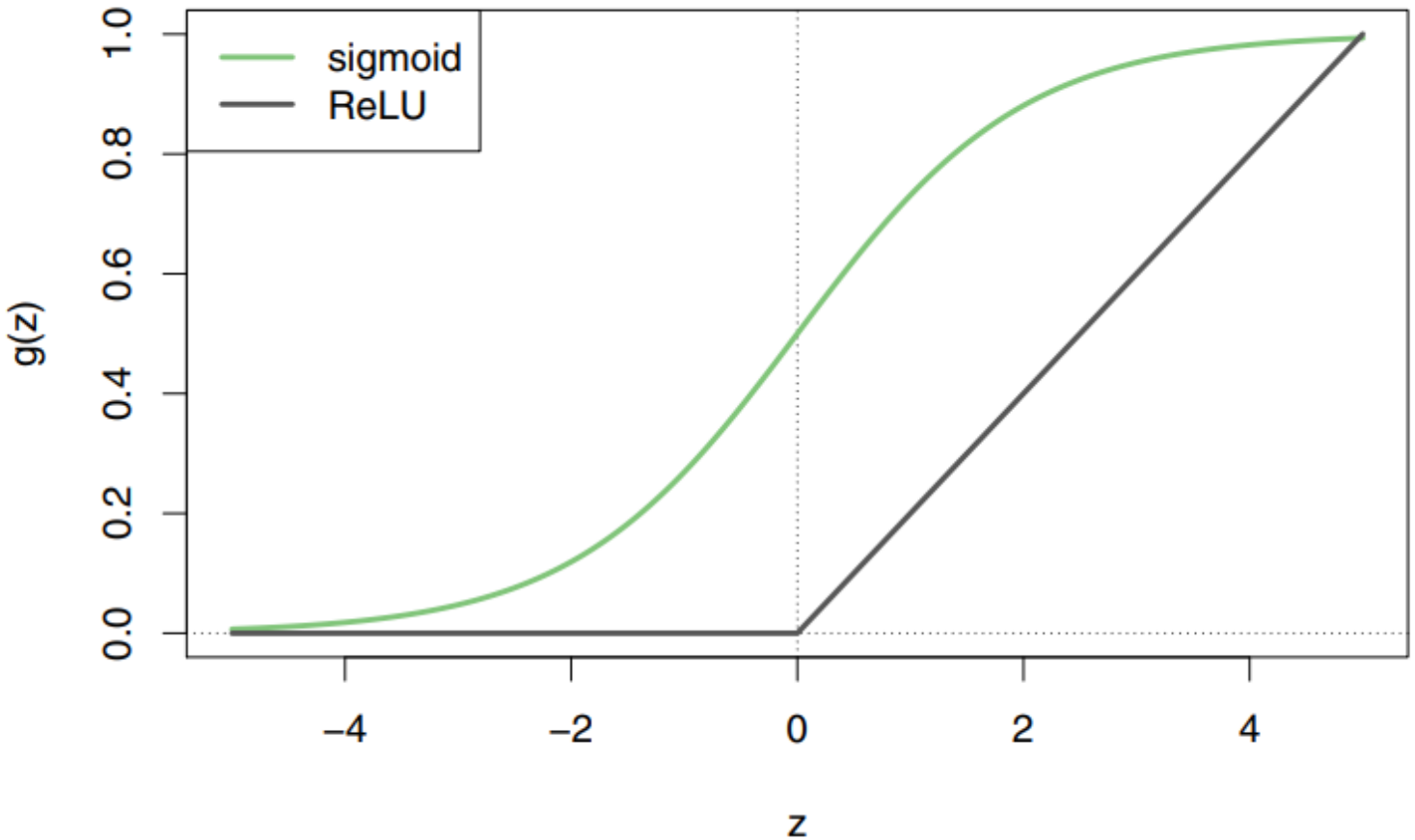
Activation Functions

Most modern neural networks prefer the *ReLU* (rectified linear unit) activation function to the sigmoid function because it is more computationally efficient produces more stable gradients:

$$g(z) = 0 \quad \text{if } z < 0$$
$$g(z) = z \quad \text{if } z \geq 0$$

The derivative of ReLU function is simple (albeit discontinuous), as it's 1 if $z > 0$ and 0 otherwise. Note that derivative at $z = 0$ is technically undefined, but most software will use 0 when performing automatic differentiation.

ReLU vs. Sigmoid



Note: the ReLU function is scaled by 1/5 in this example for ease of comparison. The function is scale invariant when used as an activation function in a neural network.

Simple Example

Consider a single input feature, X_1, and a neural network with two hidden layers that each contain only a single neuron:

$$b_1^{(1)} + w_1^{(1)} x_1 = z_1^{(1)} \rightarrow g(z_1^{(1)}) = a_1^{(1)}$$

The output of the first (and only) neuron in our first hidden layer is $a_1^{(1)}$. The model then uses this output as an input to the next hidden layer:

$$b_1^{(2)} + w_1^{(2)} a_1^{(1)} = z_1^{(2)} \rightarrow g(z_1^{(2)}) = a_1^{(2)}$$

A similar process repeats once more, yielding $\hat{Y} = a_1^{(3)}$

Learning the Parameters

Similar to logistic regression, we can use the cross-entropy cost for binary/categorical Y:

$$\text{Cost} = -\frac{1}{n}\sum_{i=1}^{n}\Big(y_i \log(\hat{y}_i) + (1-y_i)\log(1-\hat{y}_i)\Big)$$

We can use gradient descent to optimize the model's weights and biases.

Learning the Parameters

To prepare for *stochastic gradient descent*, let's first solve for $\frac{\partial Cost}{\partial w_1^{(3)}}$ for a single $\hat{y}$ using chain rule:

$$\frac{\partial Cost}{\partial w_1^{(3)}} = \frac{\partial Cost}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial w_1^{(3)}}$$

This works because $\hat{y}$ is a function of $z_1^{(3)}$ (sigmoid), and $z_1^{(3)}$ is a function of $w_1^{(3)}$

Learning the Parameters

For our simple example:

- $\frac{\partial Cost}{\partial \hat{y}} = -\left(\frac{y}{\hat{y}} - \frac{1-y}{1-\hat{y}}\right)$
- $\frac{\partial \hat{y}}{\partial z_1^{(3)}} = g(z_1^{(3)})(1 - g(z_1^{(3)}))$
- $\frac{\partial z_1^{(3)}}{\partial w_1^{(3)}} = a_1^{(2)}$

Notice how calculating this component of the gradient requires us to pass data, X_1, through the network to obtain the quantities $z_1^{(3)}$, $a_1^{(2)}$ and $\hat{y}$

Learning the Parameters

Next, let's look at the gradient vector component $\frac{\partial Cost}{\partial w_1^{(2)}}$:

$$\frac{\partial Cost}{\partial w_1^{(2)}} = \frac{\partial Cost}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z_1^{(3)}} \frac{\partial z_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_1^{(2)}}$$

- This expression is longer because $\hat{y}$ depends on $w_1^{(2)}$ indirectly through $a_1^{(2)}$
- Note that gradient components for each bias term are calculated similarly

Backpropagation

- The gradient components of parameters closer to the input layer reuse quantities that were calculated for components closer to the network's output
  - $\frac{\partial Cost}{\partial \hat{y}}$ and $\frac{\partial \hat{y}}{\partial z_1^{(3)}}$ in our example
- This makes it beneficial to work backwards through the model when calculating the components of the gradient vector
  - Thus, the application of chain rule to find the gradient of a neural network is often called the **backpropagation algorithm**

Forward Propagation

- You'll also hear the term *forward-propagation* (or *forward pass*) referring to the calculation of the cost function for an observation (or batch of observations)
- As we previously mentioned, the gradient requires several intermediate quantities that are calculated during forward-propagation
  - Thus, the process for optimization begins by feeding an observation into the existing network (forward-propagation), then updating the network's parameters via back-propagation

Remarks on Network Depth

- Neural networks first became popular in the 1980s, but in the 1990s methods like random forests, boosting, and support vector machines received far greater attention
  - This was partly due to the computational challenges of neural networks and partly due to misunderstandings related to network depth
- In the 2000s, deep neural networks (ones with many hidden layers) were found to be successful in image classification
  - In 2012, a deep neural network architecture named "AlexNet" led to a boom in *deep learning* by winning the ImageNet recognition challenge with accuracy of 84.7% (10.8% better than the nearest competitor)
  - Network depth combined with the use of GPUs for efficient training on massive datasets led to this performance

Intuition on the Role of Hidden Layers

- Why do deeper networks perform better on certain types of data, such as images?
  - Each hidden layer is learning features created by the previous layer
- Hidden layer 1 learns patterns from the inputs (such as vertical and horizontal edges of varying lengths and directions)
- Hidden layer 2 learns patterns from the features created by hidden layer 1 (perhaps simple shapes, curves, etc.)
- The next hidden layer can then learn patterns that are combinations of those shapes, curves, etc.
  - At some point, the complexity of the current features provides enough information to make accurate predictions

Intuition on the Role of Hidden Layers

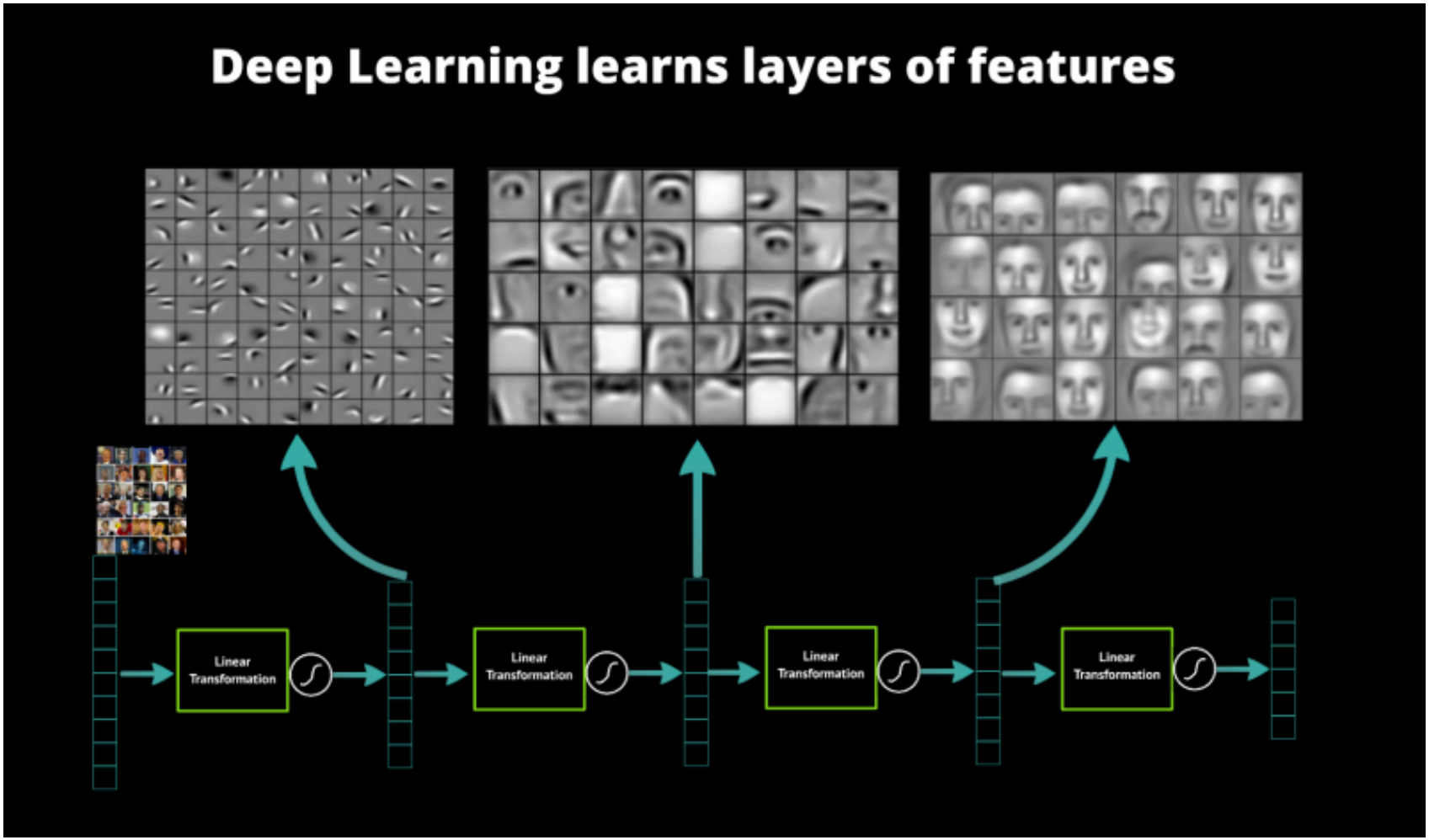


Image Credit: https://www.datarobot.com/blog/a-primer-on-deep-learning/

Conclusion

- Neural networks involve a lot of parameters and can learn very complex relationships, but this generally requires a lot of training data
- The simple networks we discussed today tend not to be commonly used
  - They aren't well-equipped to handle spatial structures, which make them less effective at applications involving image/textual data
  - They tend to overfit "flat" or "tabular" data to a greater extent than methods like random forests or boosted ensembles
- Next we'll learn about *convolutional neural networks*, a variation that utilizes spatial relationships among features and excels in computer vision applications