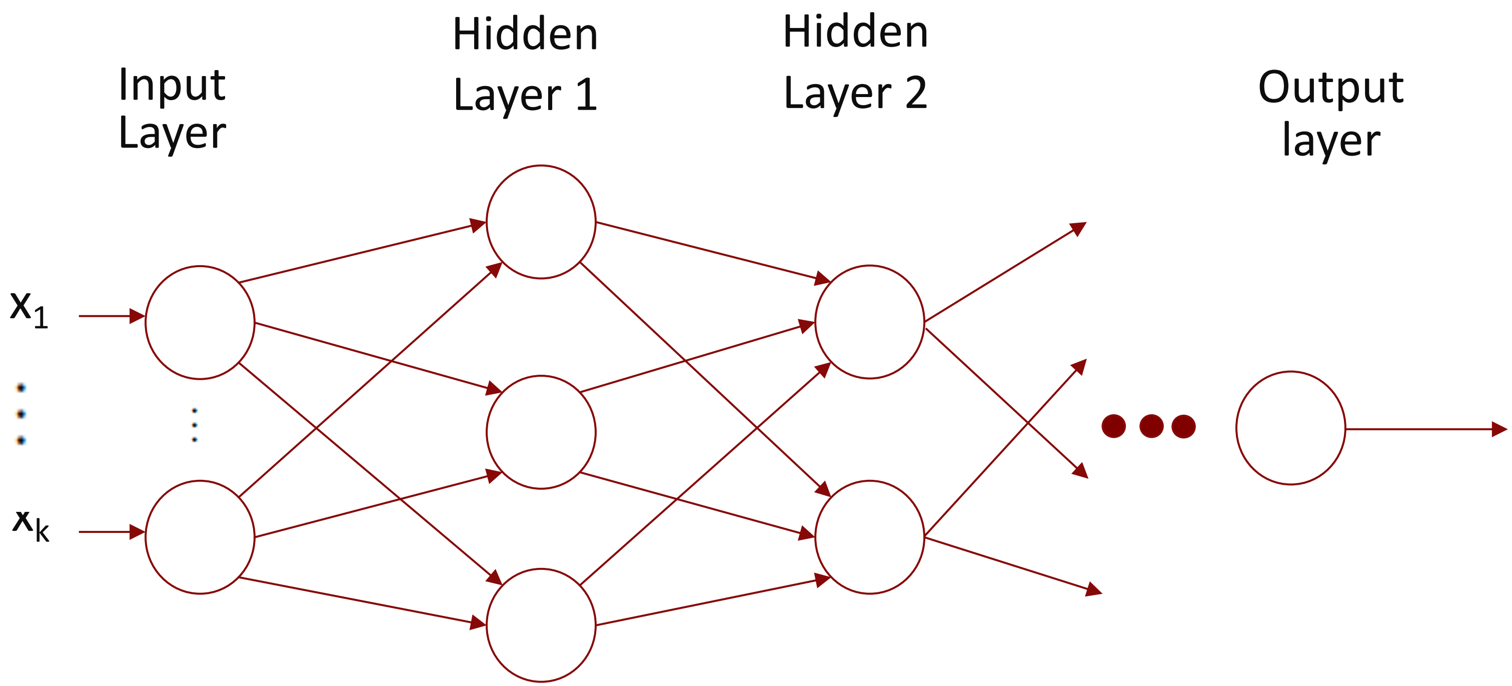
Lecture 2:
Training Deep Neural Networks

MIT MANAGEMENT SLOAN SCHOOL

15.S04: Hands-on Deep Learning
Spring 2024
Farias, Ramakrishnan

Recap: Designing a DNN



User choices the ***# of hidden layers****,* ***# units in each layer****, the* ***activation function(s)*** *for the hidden layers and for the output layer*

Application: Predicting heart disease

Predicting Heart Disease

Using a dataset of patients made available by the Cleveland Clinic, we will build our first NN model to predict if a patient has been diagnosed with heart disease from demographics and bio-markers

Column	Description	Feature Type
Age	Age in years	Numerical
Sex	(1 = male; 0 = female)	Categorical
CP	Chest pain type (0, 1, 2, 3, 4)	Categorical
Trestbpd	Resting blood pressure (in mm Hg on admission)	Numerical
Chol	Serum cholesterol in mg/dl	Numerical
FBS	fasting blood sugar in 120 mg/dl (1 = true; 0 = false)	Categorical
RestECG	Resting electrocardiogram results (0, 1, 2)	Categorical
Thalach	Maximum heart rate achieved	Numerical
Exang	Exercise induced angina (1 = yes; 0 = no)	Categorical
Oldpeak	ST depression induced by exercise relative to rest	Numerical
Slope	Slope of the peak exercise ST segment	Numerical
CA	Number of major vessels (0-3) colored by fluoroscopy	Both numerical & categorical
Thal	3 = normal; 6 = fixed defect; 7 = reversible defect	Categorical
Target	Diagnosis of heart disease (1 = true; 0 = false)	Target

What we want to predict

Let’s design our NN

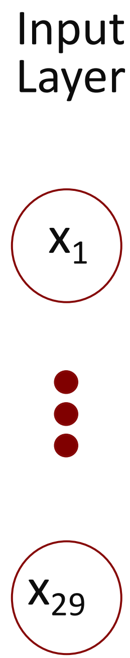
- We design i.e., “lay out” the network
  - Choose ***the number of hidden layers*** and ***the number of ‘neurons’ in each layer***
  - Pick the ***right output layer*** based on the type of the output

Let’s design our NN

- We design i.e., “lay out” the network
  - Choose ***the number of hidden layers*** and ***the number of ‘neurons’ in each layer*** 1 hidden layer with 16 ReLU neurons
  - Pick the ***right output layer*** based on the type of the output

Sigmoid

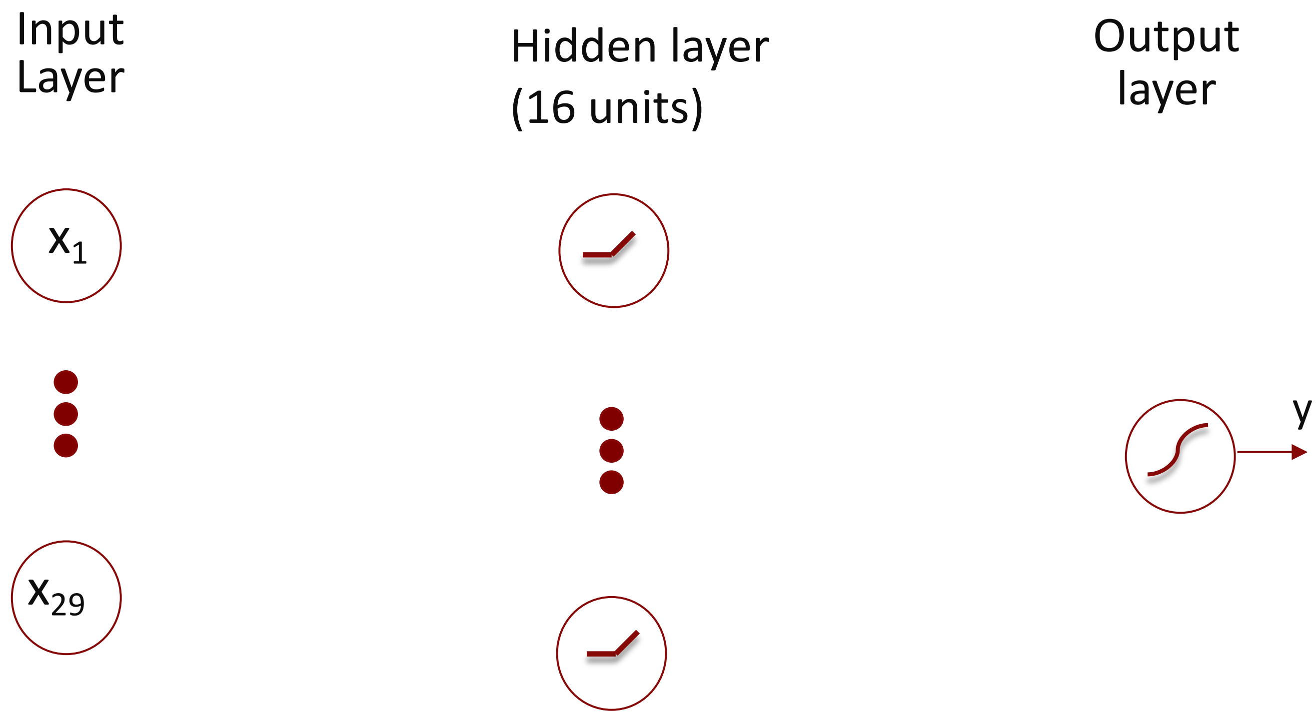
Let’s visualize this NN



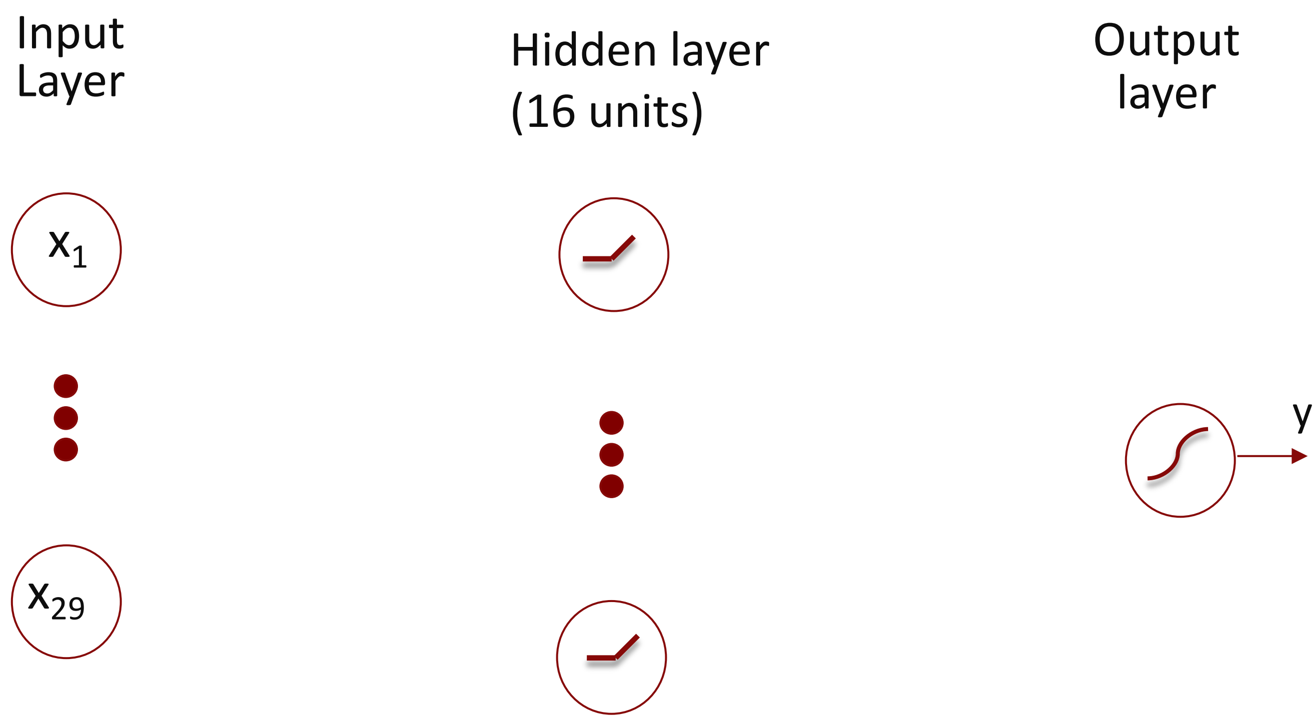
Column	Description	Feature Type
Age	Age in years	Numerical
Sex	(1 = male; 0 = female)	Categorical
CP	Chest pain type (0, 1, 2, 3, 4)	Categorical
Trestbpd	Resting blood pressure (in mm Hg on admission)	Numerical
Chol	Serum cholesterol in mg/dl	Numerical
FBS	fasting blood sugar in 120 mg/dl (1 = true; 0 = false)	Categorical
RestECG	Resting electrocardiogram results (0, 1, 2)	Categorical
Thalach	Maximum heart rate achieved	Numerical
Exang	Exercise induced angina (1 = yes; 0 = no)	Categorical
Oldpeak	ST depression induced by exercise relative to rest	Numerical
Slope	Slope of the peak exercise ST segment	Numerical
CA	Number of major vessels (0-3) colored by fluoroscopy	Both numerical & categorical
Thal	3 = normal; 6 = fixed defect; 7 = reversible defect	Categorical
Target	Diagnosis of heart disease (1 = true; 0 = false)	Target

There are only 13 input variables but some of them are categorical so we one-hot-encode them, resulting in 29 inputs (details in colab).

Let’s visualize this NN

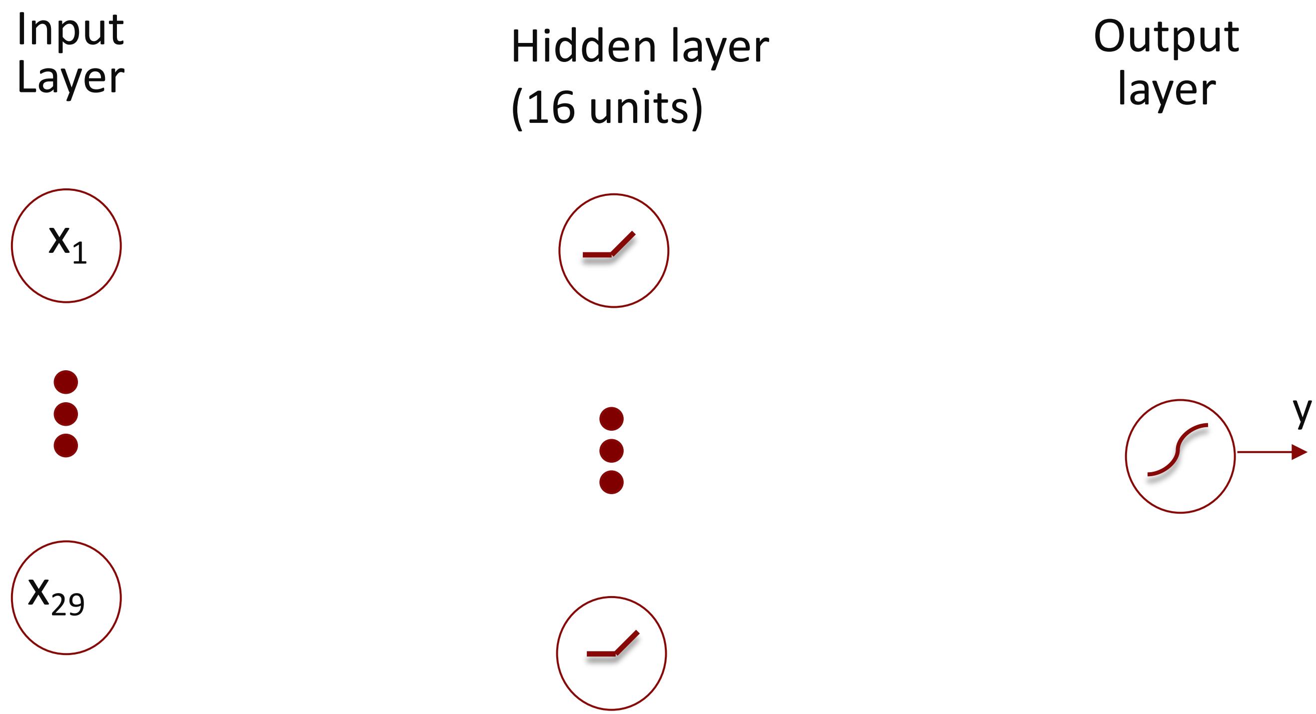


Let’s visualize this NN



How many parameters (i.e., weights and biases) does this network have?

Let’s visualize this NN



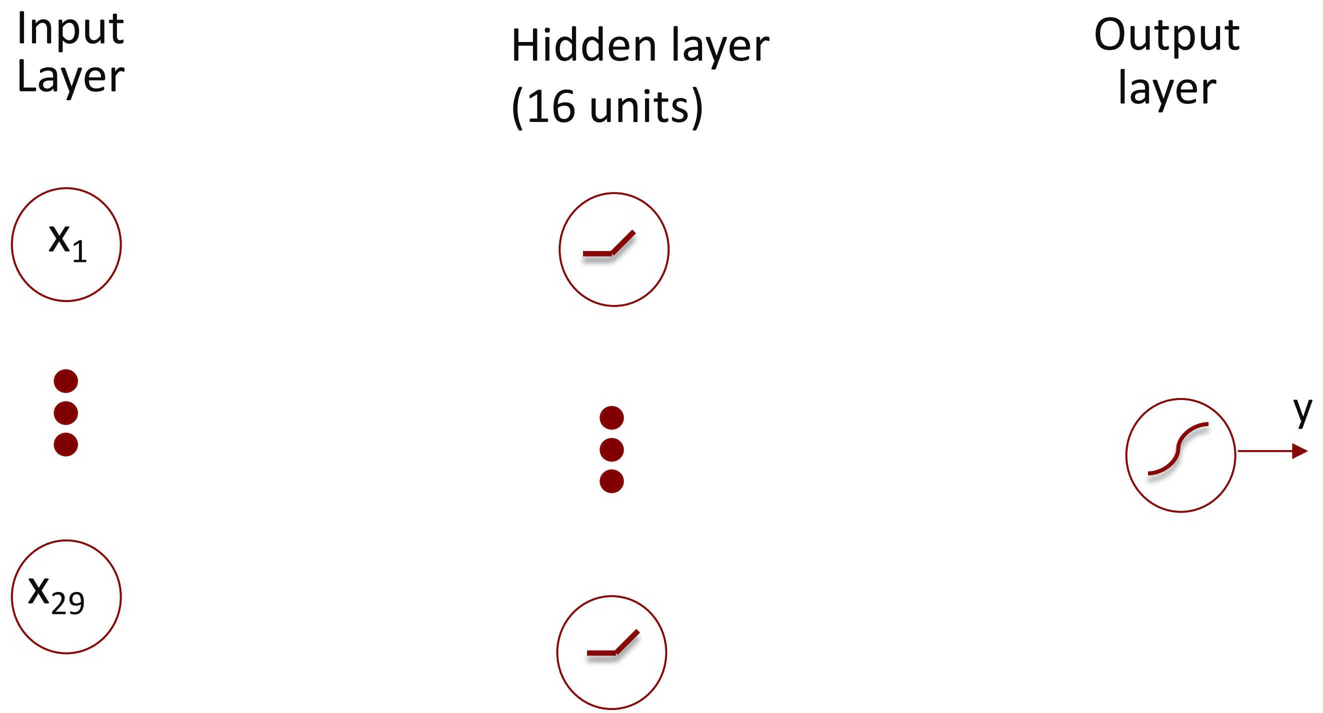
How many parameters (i.e., weights and biases) does this network have?

29 * 16 + 16 + 16 * 1 + 1 = 497

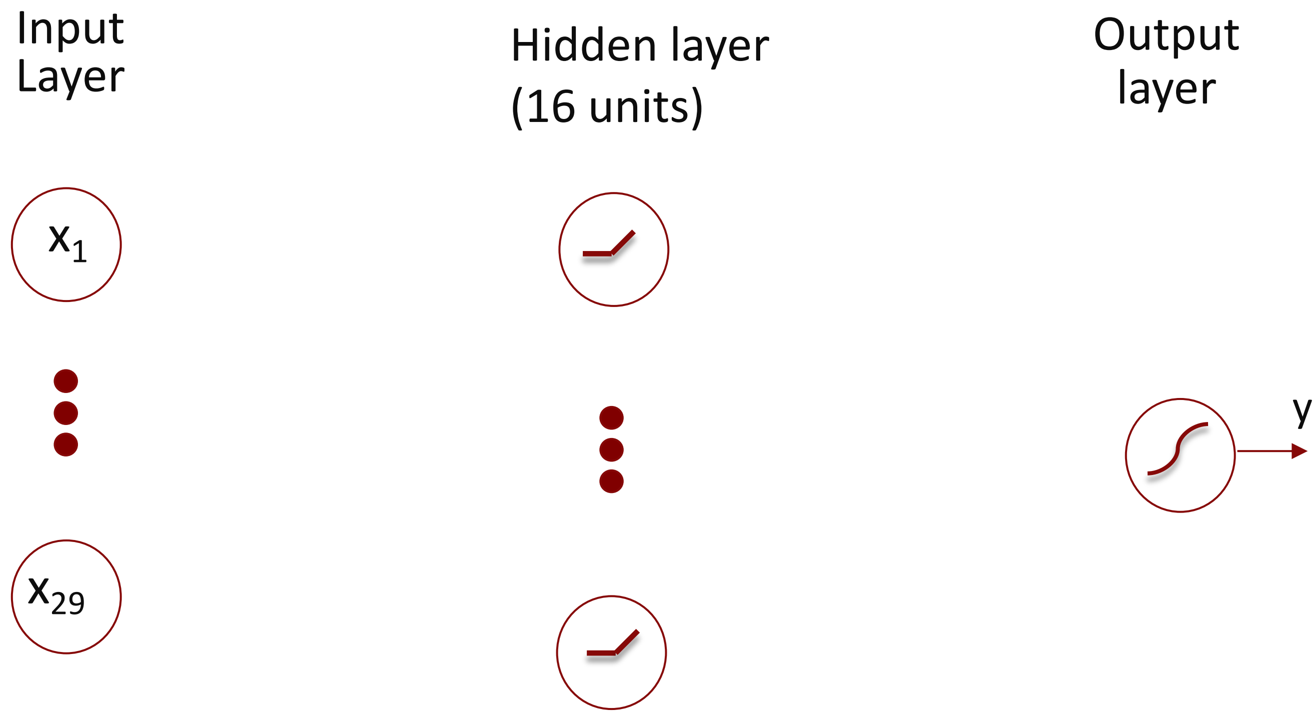
We will now “translate” this network into Keras code to demonstrate how easy it is.

We will give a fuller intro to Keras/Tensorflow <u>and</u> train this model in Colab soon.

Typically, we define each layer from left to right

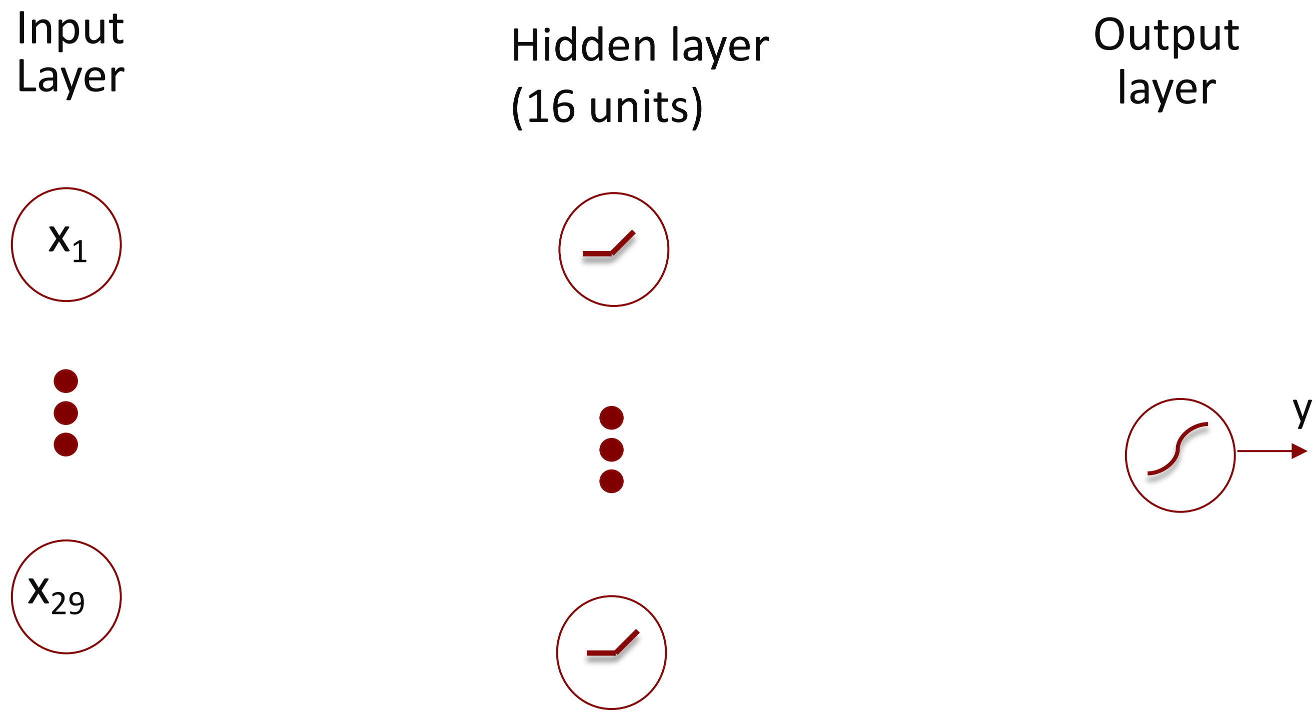


Let’s start with the input layer



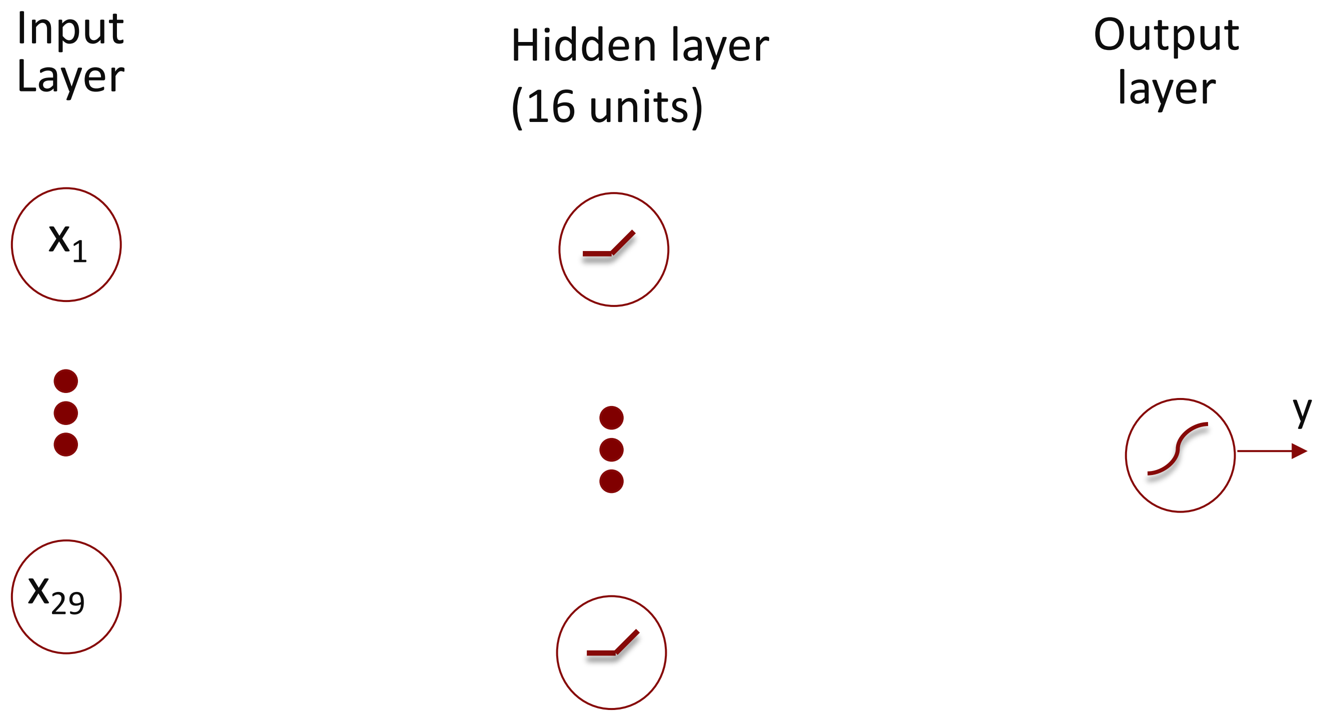

```
input = keras.Input(shape=29)
```

We specify the shape of the input




```
input = keras.Input(shape=29)
```

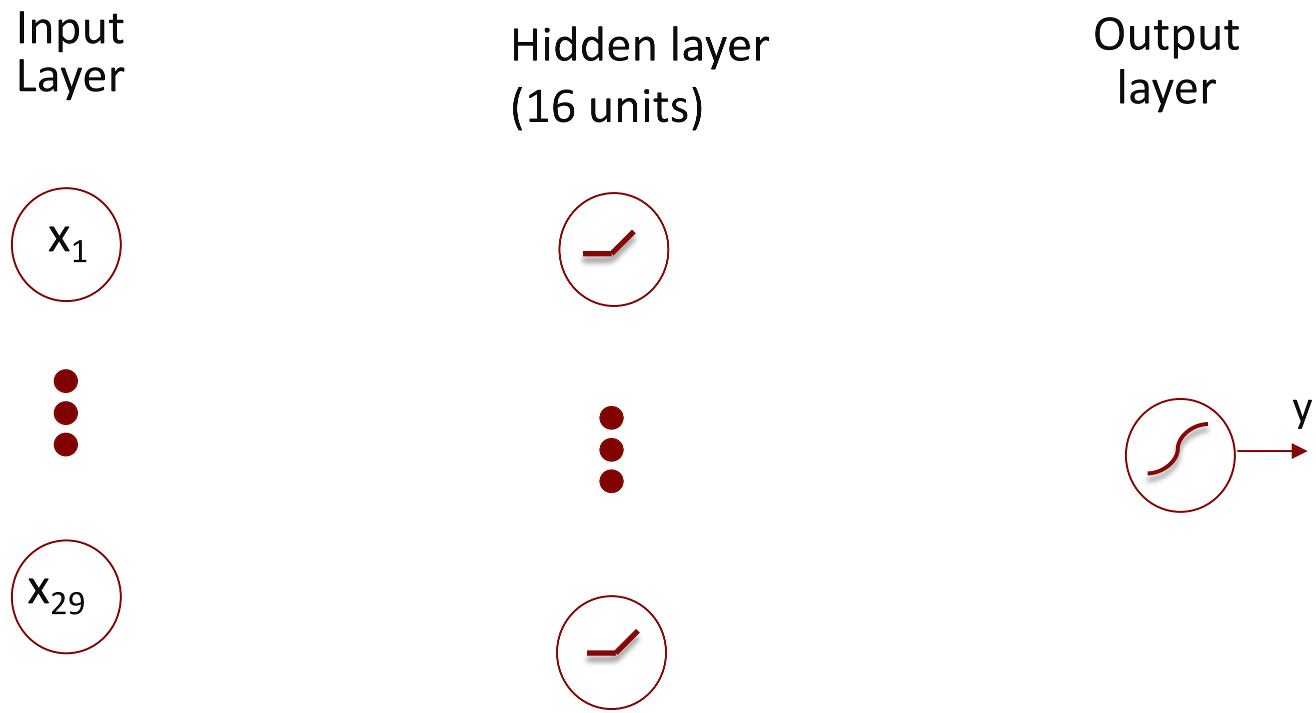
Next, we define the hidden layer




```
input = keras.Input(shape=29)
```

```
keras.layers.Dense(16, activation="relu")
```

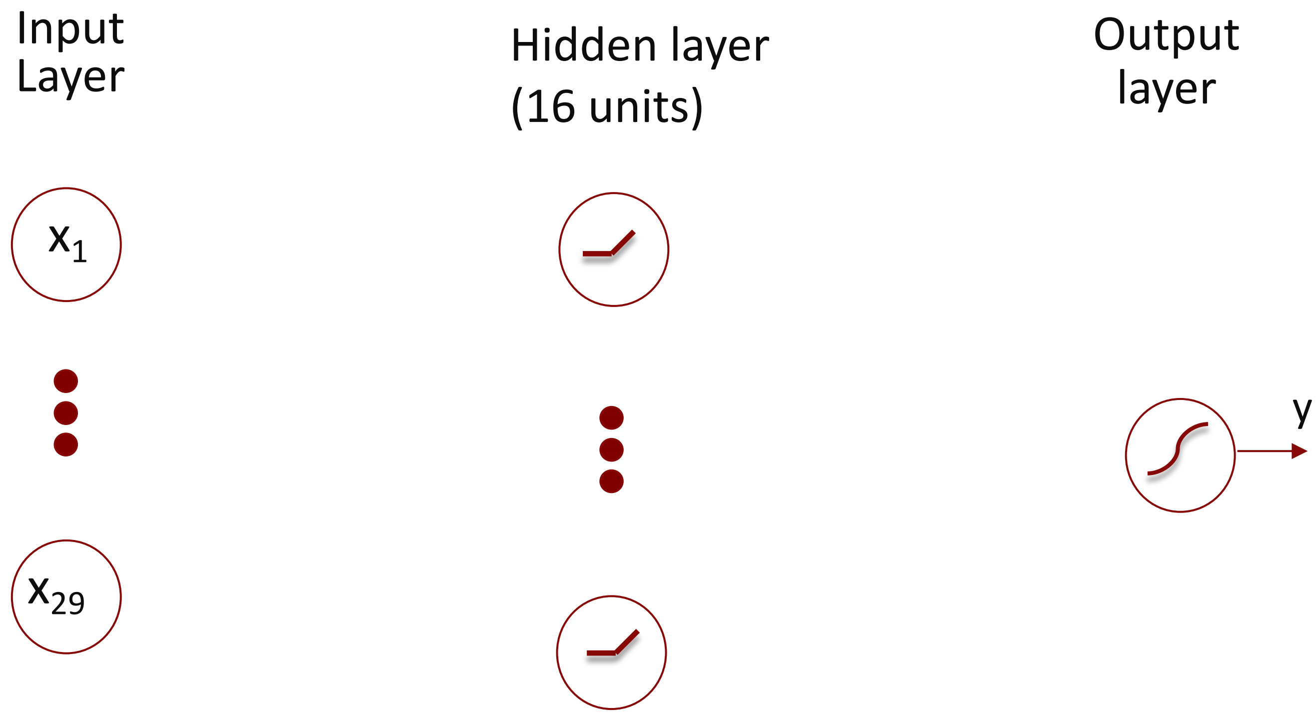
Since this layer is fully connected to the previous and later layers, we use ‘Dense’




```
input = keras.Input(shape=29)
```

```
keras.layers.Dense(16, activation=“relu”)
```

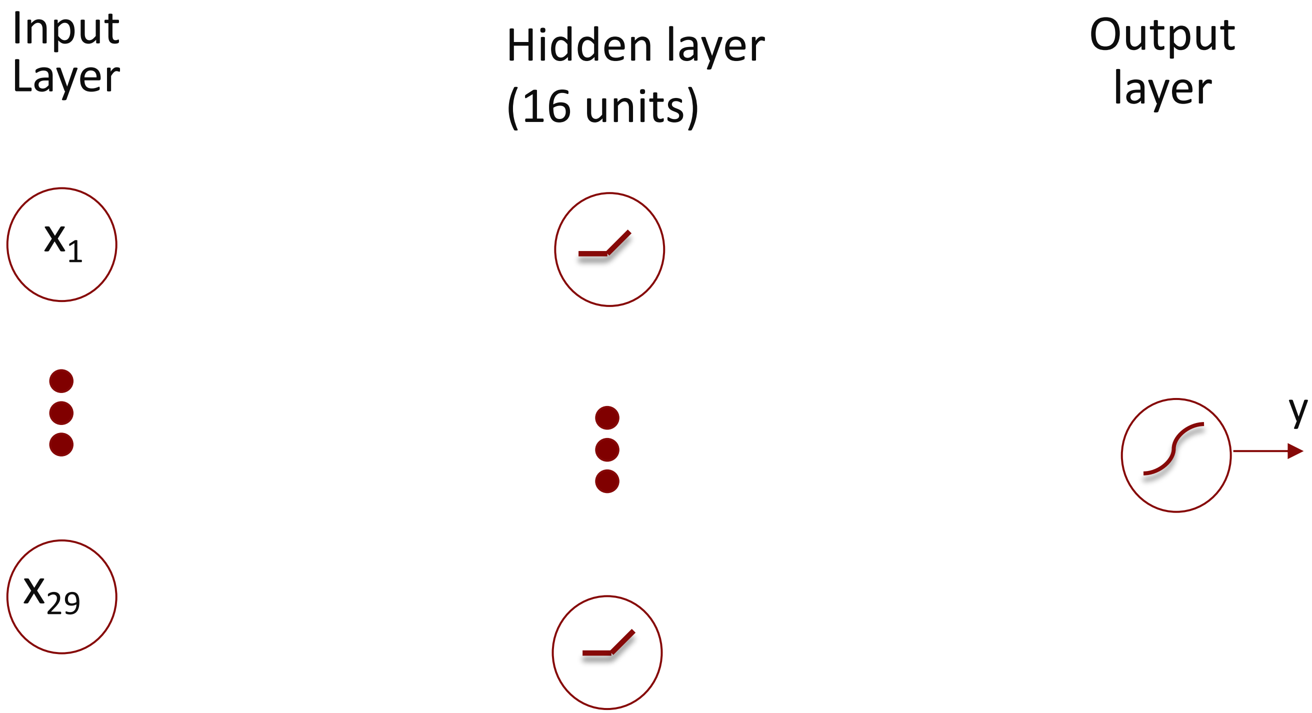
We specify the number of neurons we want in this layer ...




```
input = keras.Input(shape=29)
```

```
keras.layers.Dense(16, activation="relu")
```

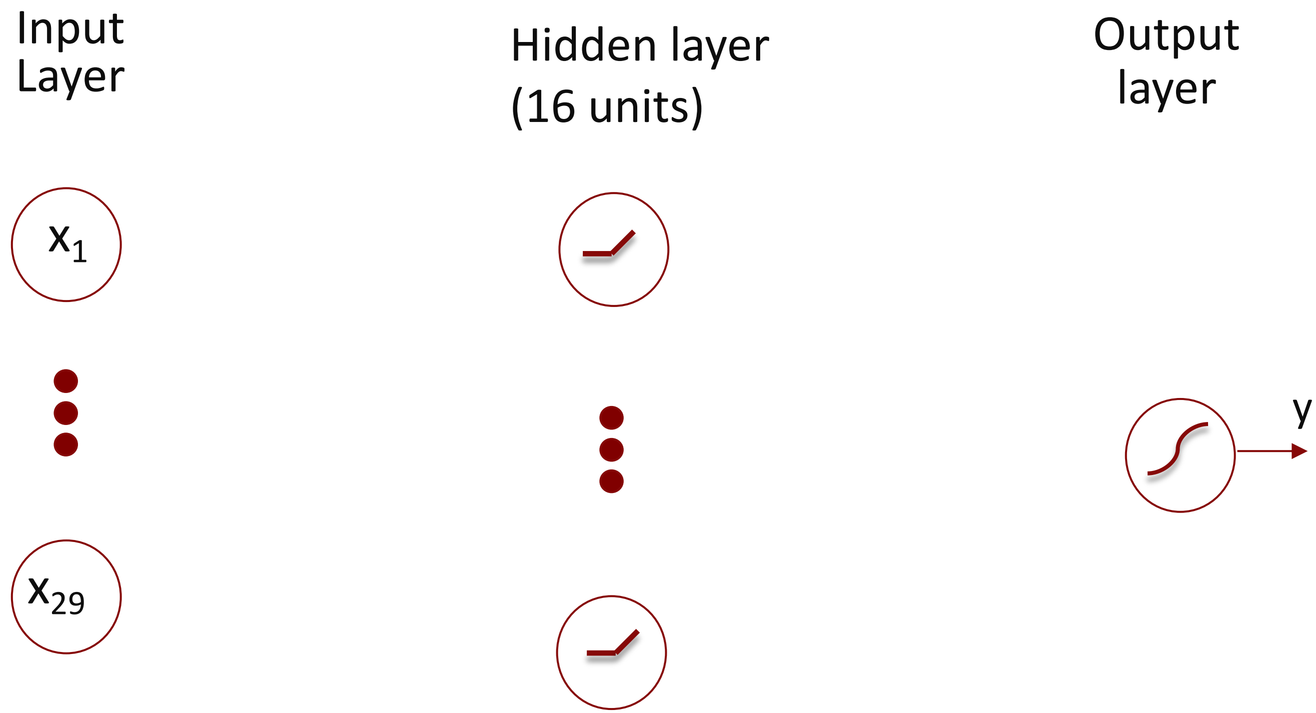
… and the activation function




```
input = keras.Input(shape=29)
```

```
keras.layers.Dense(16, activation=“relu”)
```

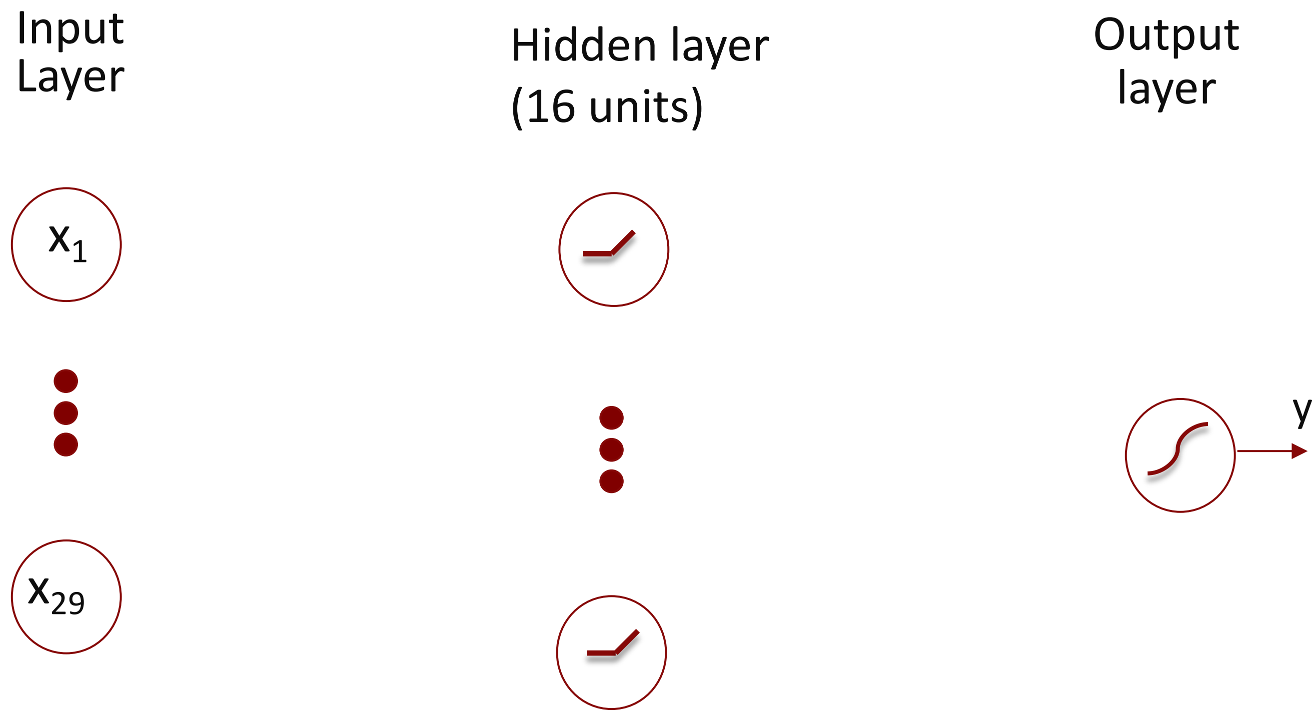
Next, we ”feed” the input to this layer …




```
input = keras.Input(shape=29)
```

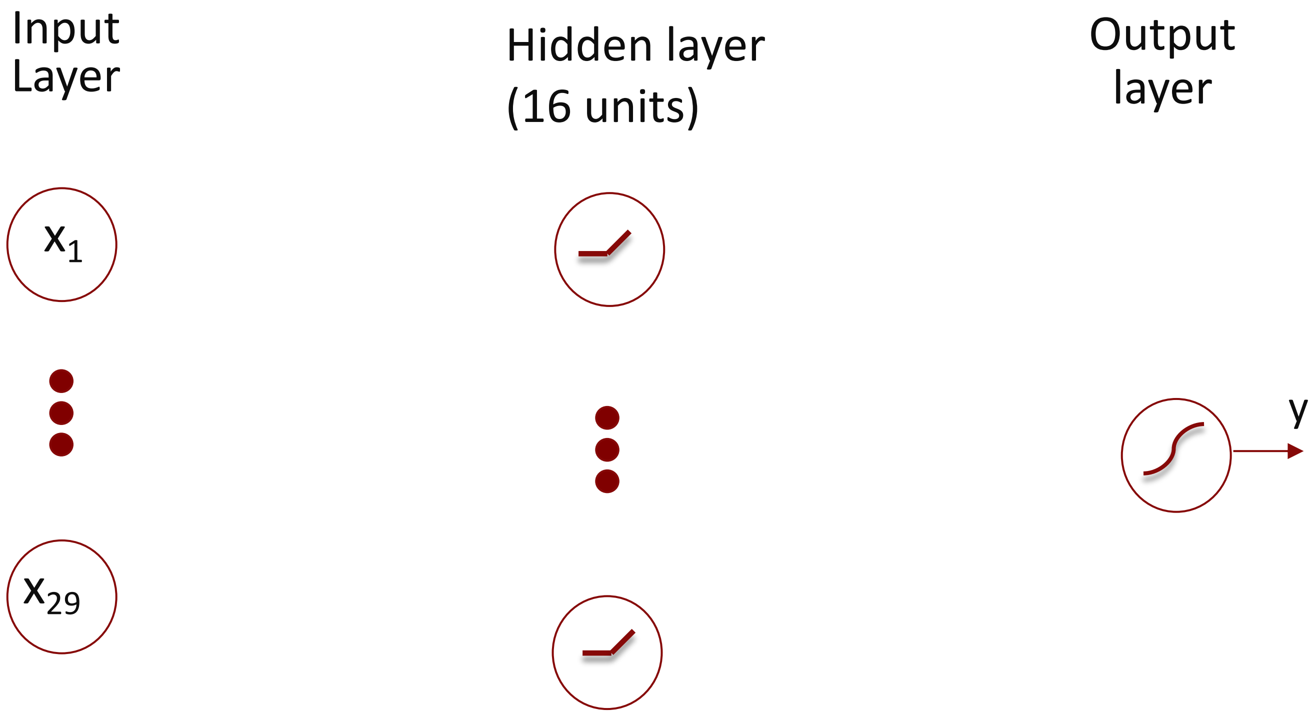
```
keras.layers.Dense(16, activation=“relu”)(input)
```

… and give a name to the output of this layer



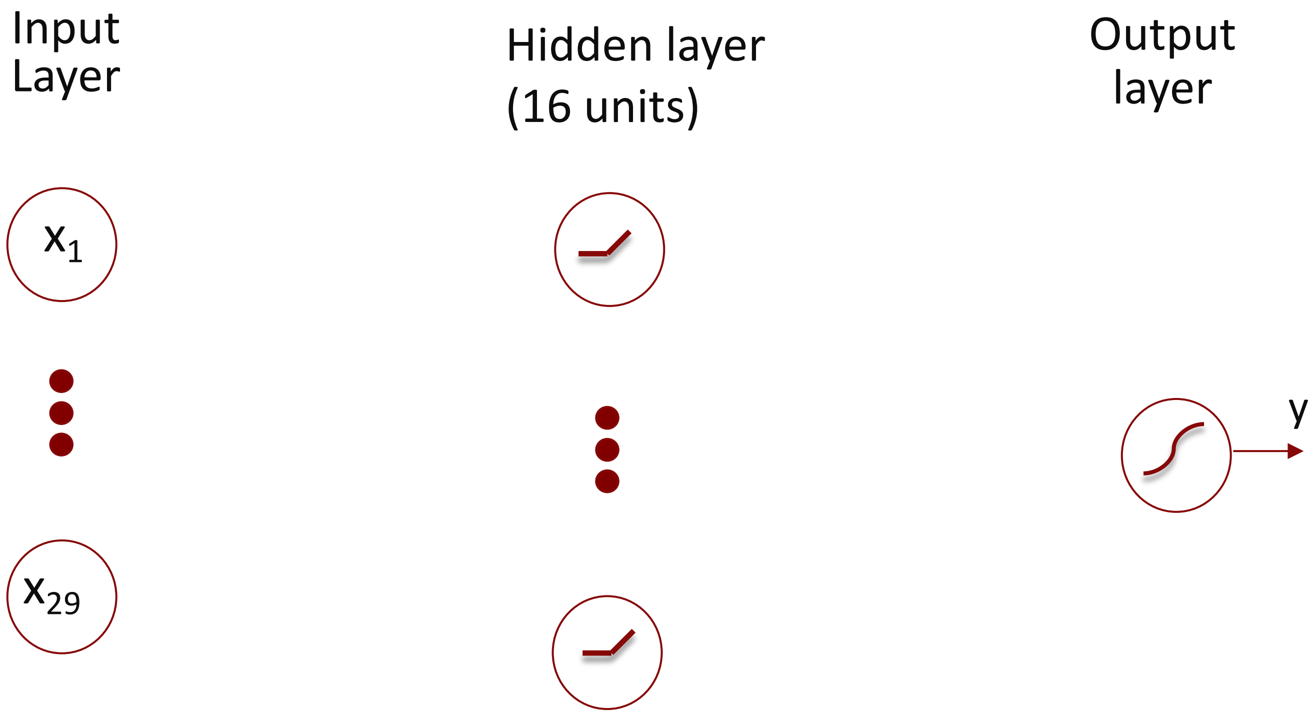

```
input = keras.Input(shape=29)
h = keras.layers.Dense(16, activation="relu")(input)
```

Finally, we come to the output layer



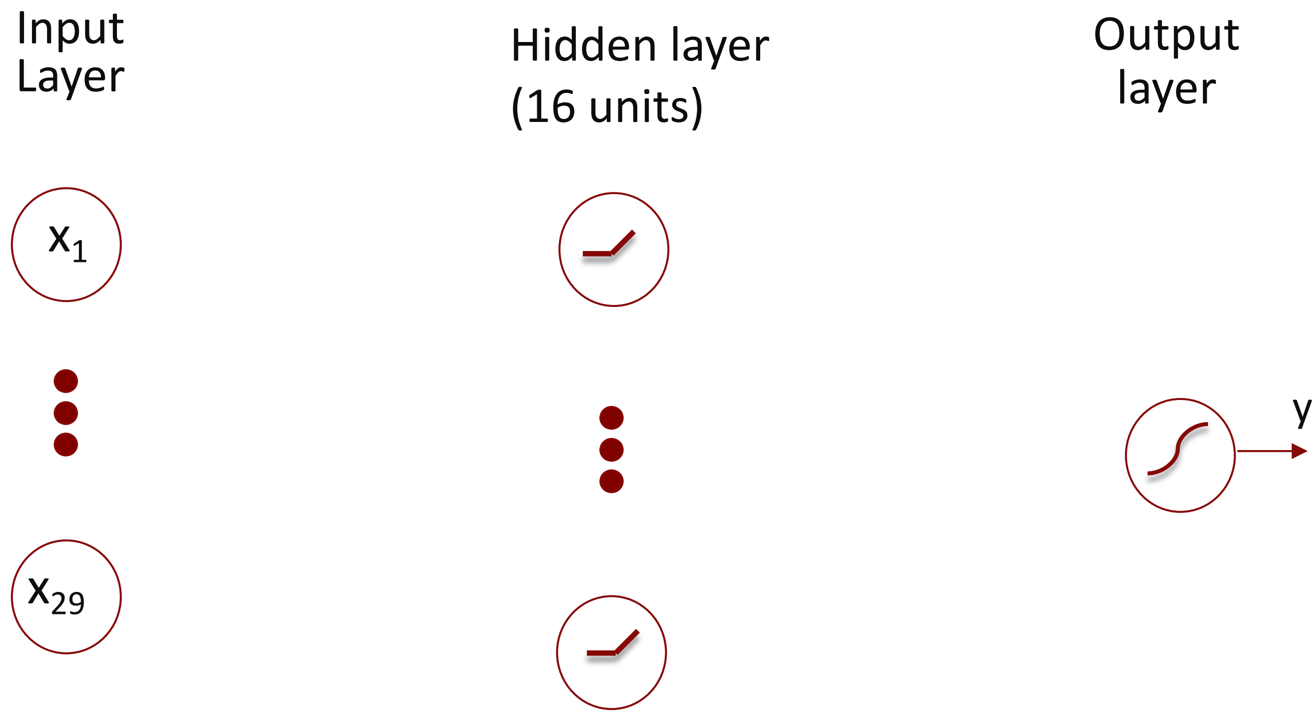

```
input = keras.Input(shape=29)
   h = keras.layers.Dense(16, activation="relu")(input)
                          keras.layers.Dense(1, activation="sigmoid")
```

We have just one unit in this layer …



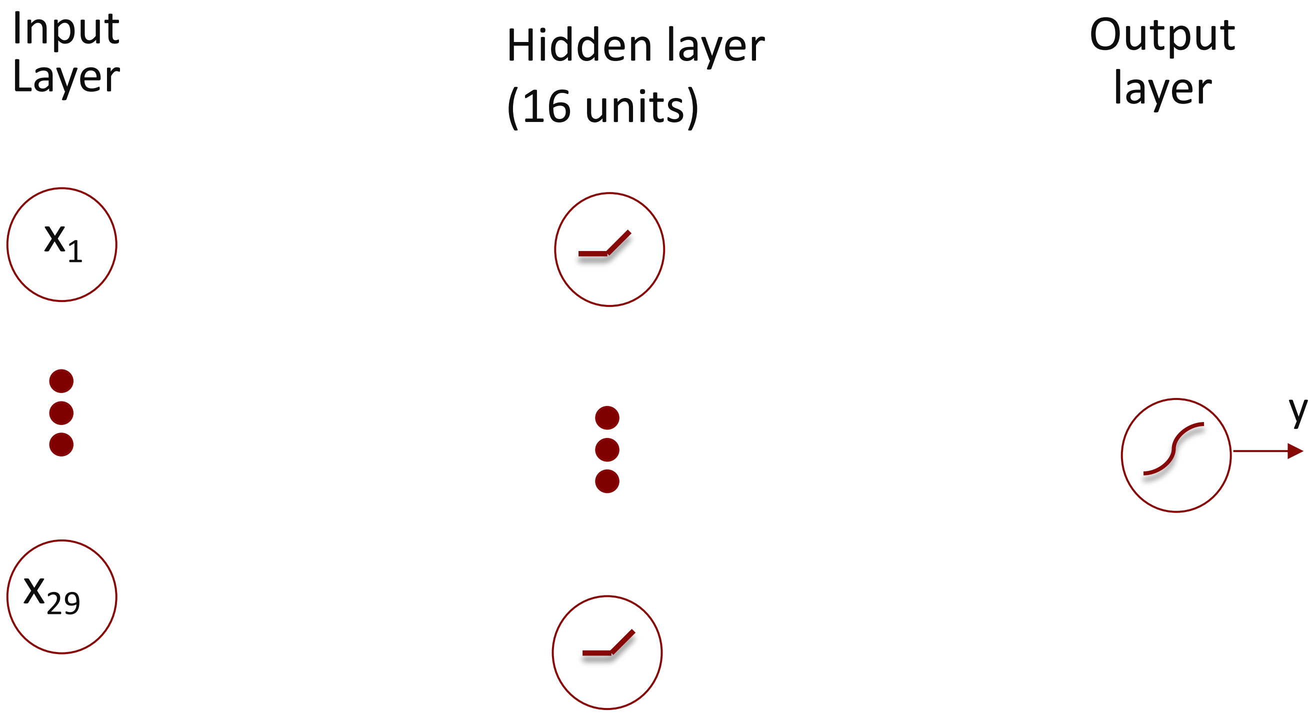

```
input = keras.Input(shape=29)
    h = keras.layers.Dense(16, activation=“relu”)(input)
                                    keras.layers.Dense(1, activation="sigmoid”)
```

… and indicate that we need a sigmoid activation function



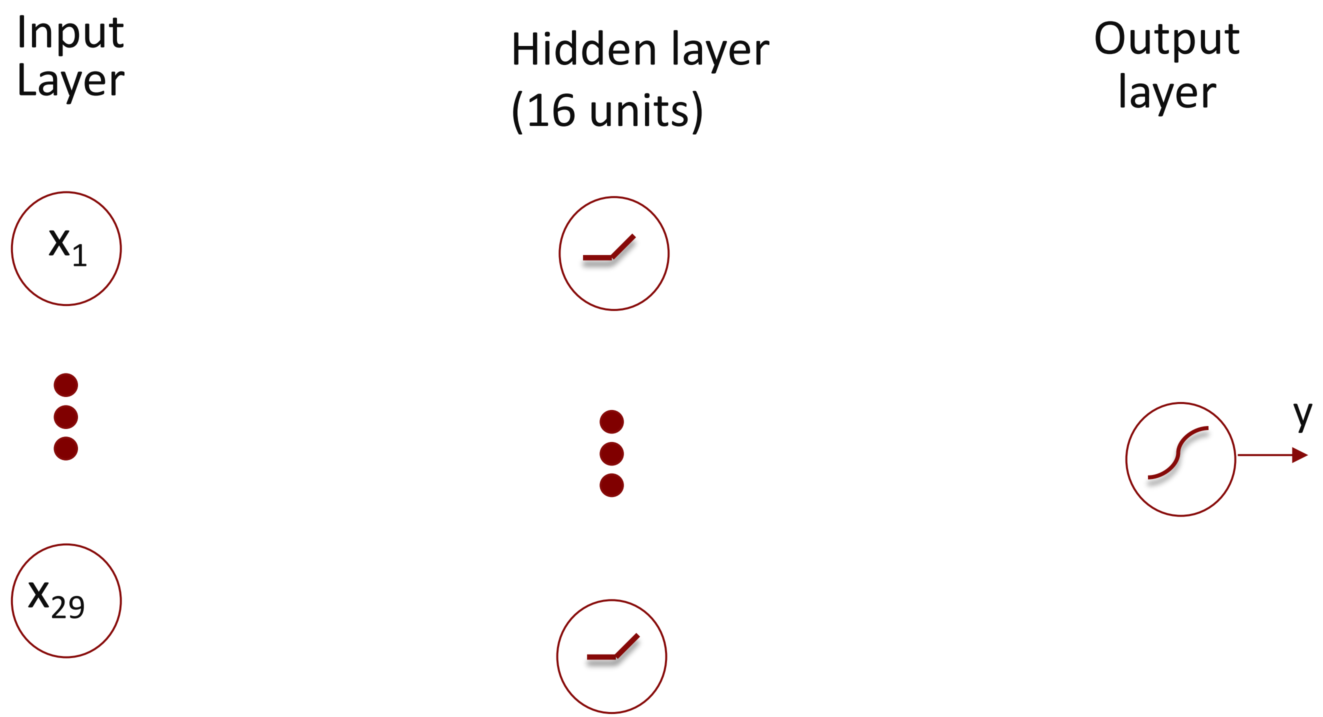

```
input = keras.Input(shape=29)
    h = keras.layers.Dense(16, activation="relu")(input)
                             keras.layers.Dense(1, activation="sigmoid")
```

As we did before, we "feed" the output of the hidden layer to this layer …



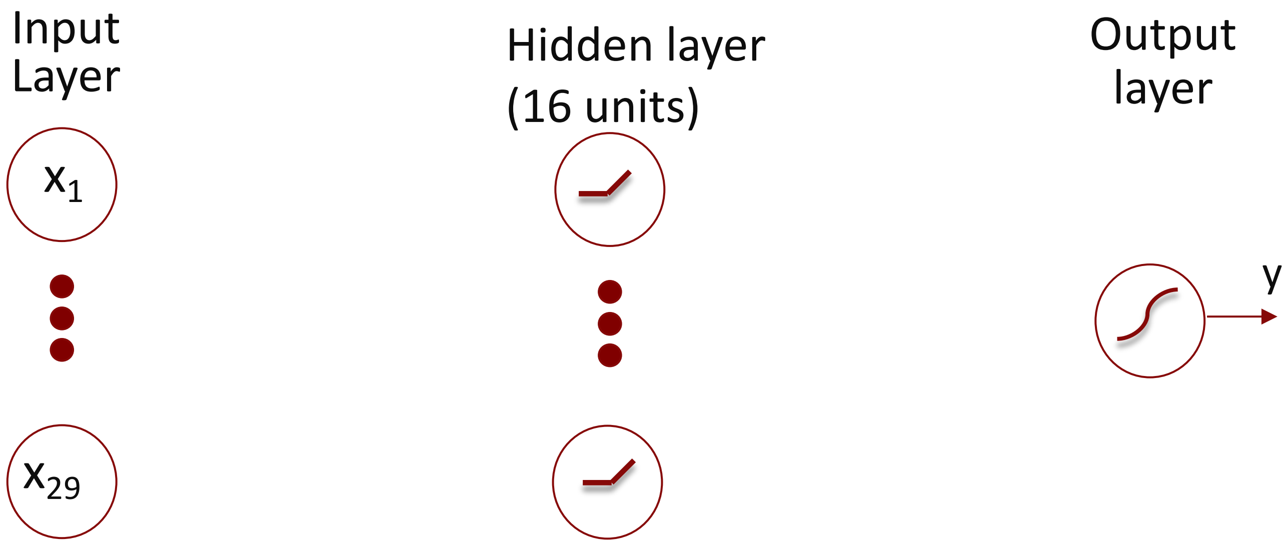

```
input = keras.Input(shape=29)
    h = keras.layers.Dense(16, activation="relu")(input)
                keras.layers.Dense(1, activation="sigmoid")(h)
```

… and give the output of this layer a name.




```
input = keras.Input(shape=29)
    h = keras.layers.Dense(16, activation="relu")(input)
              output = keras.layers.Dense(1, activation="sigmoid")(h)
```

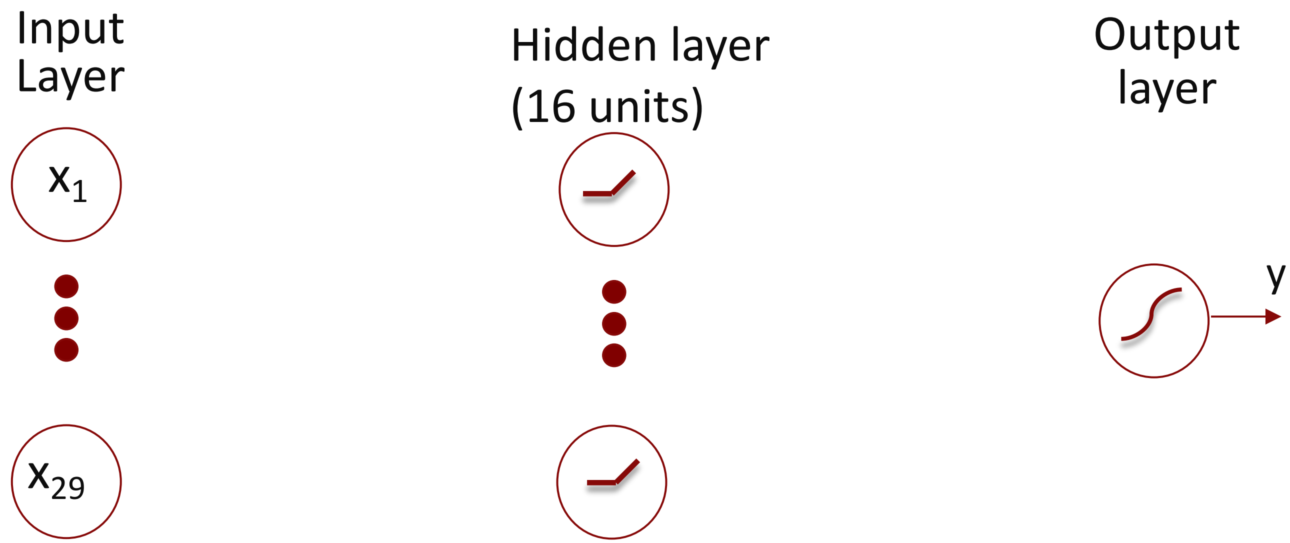
We have defined and connected the layers




```
input = keras.Input(shape=29)
h = keras.layers.Dense(16, activation=“relu”)(input)
output = keras.layers.Dense(1, activation="sigmoid”)(h)
```

We have defined and connected the layers.

The final step is to formally define a model.




```
input = keras.Input(shape=29)
h = keras.layers.Dense(16, activation=“relu”)(input)
output = keras.layers.Dense(1, activation="sigmoid”)(h)
model = keras.Model(input, output)
```

That’s it!

A Neural Model for Heart Disease Prediction

```
input = keras.Input(shape=29)
h = keras.layers.Dense(16, activation=“relu”)(input)
output = keras.layers.Dense(1, activation="sigmoid”)(h)
model = keras.Model(input, output)
```

We will show how to train this model with real data and use it for prediction after we cover some *conceptual* building blocks

Training a Deep Neural Network

Recap: Training Linear and Logistic Regression Models

Linear Regression

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

+ Data

lm

$$y = 2.8 + 0.89x_1 - 3.9x_2 + \ldots + 1.06x_n$$

Recap: Training Linear and Logistic Regression Models

Linear Regression

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

+ Data

lm

$$y = 2.8 + 0.89x_1 - 3.9x_2 + \ldots + 1.06x_n$$

Logistic Regression

$$y = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n)}}$$

+ Data

glm

$$y = \frac{1}{1 + e^{-(2.8 + 0.89x_1 - 3.9x_2 + \ldots + 1.06x_n)}}$$

Recap: Training Linear and Logistic Regression Models

Linear Regression

$$y = \beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

+ Data → lm →

$$y = 2.8 + 0.89x_1 - 3.9x_2 + \ldots + 1.06x_n$$

Logistic Regression

$$y = \frac{1}{1 + e^{-(\beta_0 + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n)}}$$

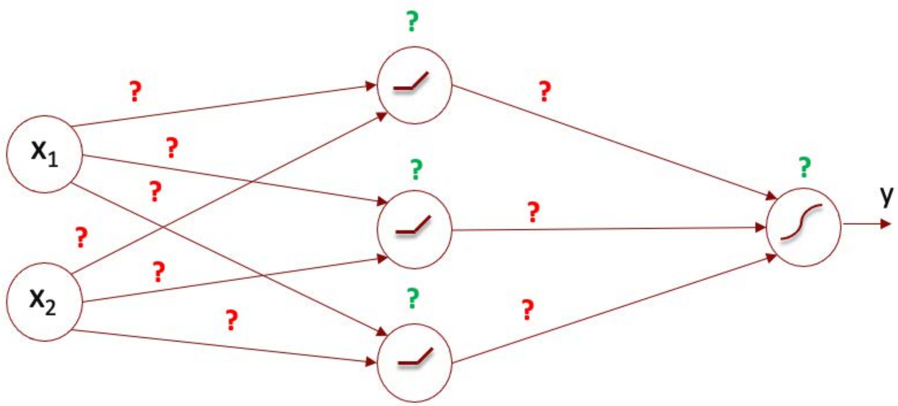
+ Data → glm →

$$y = \frac{1}{1 + e^{-(2.8 + 0.89x_1 - 3.9x_2 + \ldots + 1.06x_n)}}$$

Recall

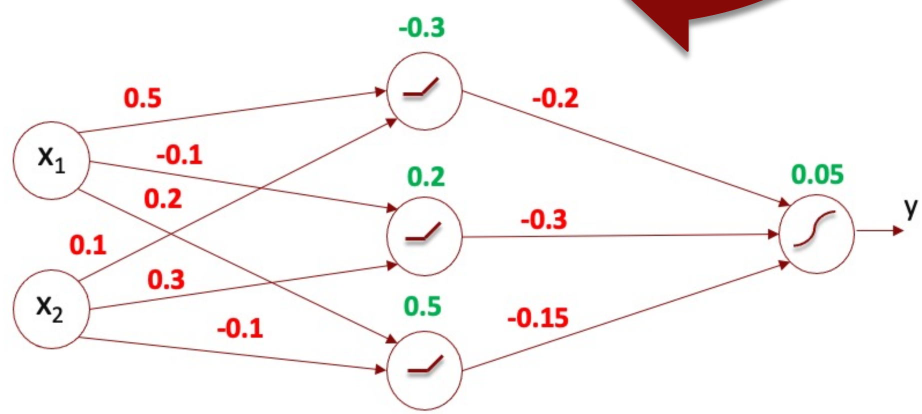
- Training is finding values for the weights/coefficients so that the model's predictions come as close to the actual values as possible
- 'lm' and 'glm' use optimization algorithms under the hood to find these "best" values

Training a DNN



+ Data

Training a DNN is no different. It just happens to be a very complex function with lots of parameters.



The essence of training is to find the "best" values for the weights and biases i.e., those that minimize a function that measures the discrepancy between the actual and predicted values

These functions are called loss functions in the DL world

Loss Functions

Loss functions

- A “loss function” is a function that quantifies the error in a model’s prediction.
  - If the predictions are close to the actual values, the “loss” would be ______.
  - A perfect model would have a loss of ______.

Loss functions

- A “loss function” is a function that quantifies the error in a model’s prediction.
  - If the predictions are close to the actual values, the “loss” would be small.
  - A perfect model would have a loss of zero.

Loss functions

- A “loss function” is a function that quantifies the error in a model’s prediction.
  - If the predictions are close to the actual values, the “loss” would be small.
  - A perfect model would have a loss of zero.
- In linear regression, you will recall that we quantify this error using “sum of squared errors”. So, “sum of squared errors” is the loss function used in linear regression

Loss functions

- A “loss function” is a function that quantifies the error in a model’s prediction.
  - If the predictions are close to the actual values, the “loss” would be small.
  - A perfect model would have a loss of zero.
- In linear regression, you will recall that we quantify this error using ”sum of squared errors”. So, “sum of squared errors” is the loss function used in linear regression
- The loss function we chose must be matched well with the kind of output that comes out of the model.

Mean Squared Error (MSE) Loss is commonly used for general numerical outputs

$$\frac{1}{n}\sum_{i=1}^{n}\left(y^{i} - model\left(x^{i}\right)\right)^{2}$$

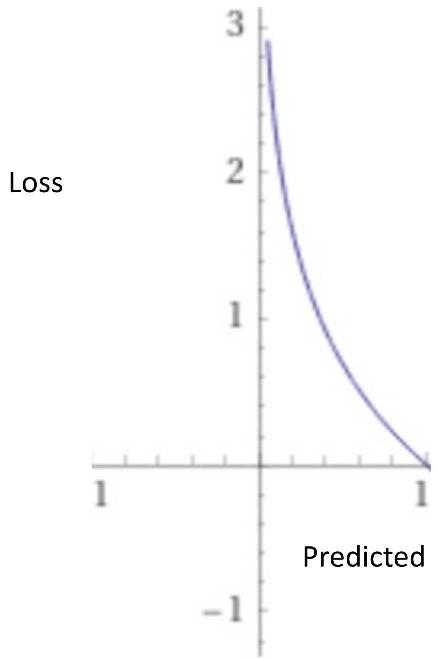
Actual value of i^{th} data point

Predicted value of i^{th} data point

In the Heart Disease Prediction Model the prediction is a probability number and the actual output is 0-1.

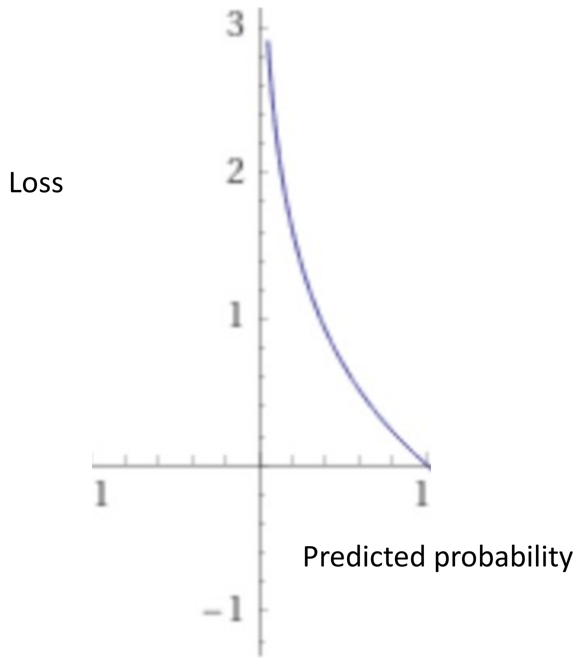
What is a good loss function in this situation?

For data points with y = 1 (i.e., patients with heart disease), lower predicted probabilities should have higher loss



For data points with y = 1

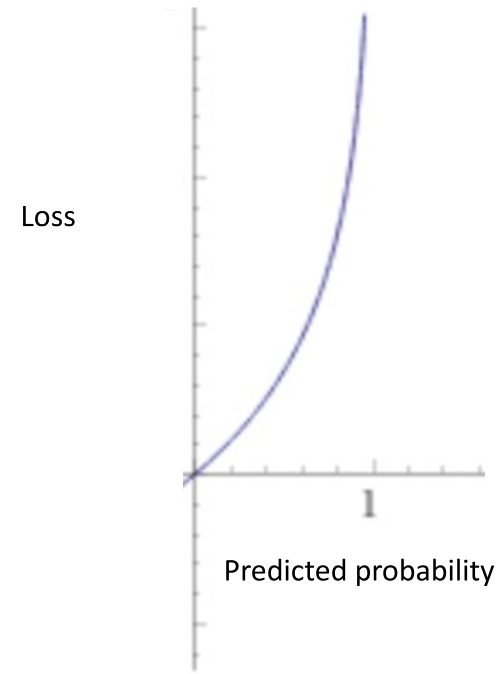
We can capture this requirement using the log function



Predicted probability	*-log(predicted probability)*
1/1000	9.97
1/10	3.32
1/2	1.0
1	0.0

For data points with y = 1,
loss = -log(predicted probability)

For data points with y = 0 (i.e., patients without heart disease), higher predicted probabilities should have higher loss

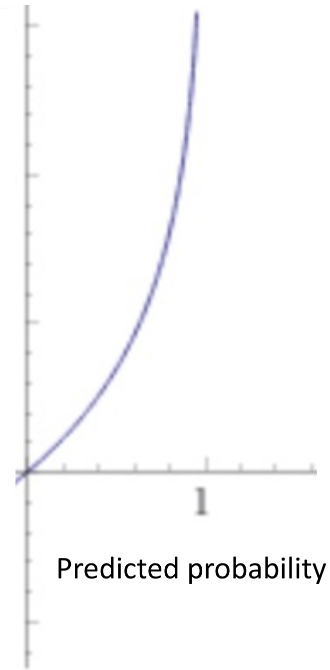


For data points with y = 0

We can capture this requirement as well using the log function

Predicted probability	***-log(1 - predicted probability)***
1/1000	0.001
1/10	0.15
1/2	1.0
0.999999	19.93

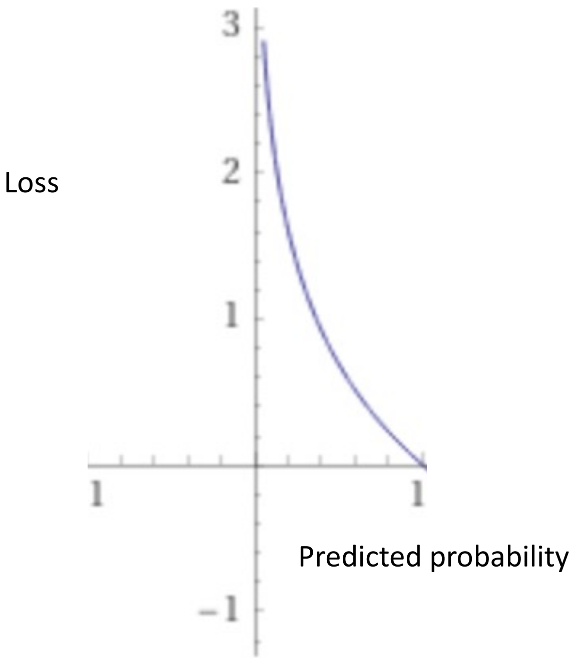
Loss

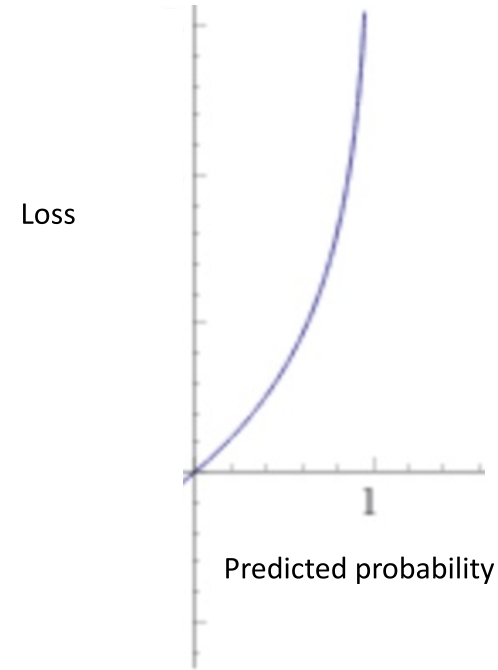


Predicted probability

For data points with y = 0,
loss = -log(1 - predicted probability

Summary





For data points with y = 1,
loss = -log(predicted probability)

For data points with y = 0,
loss = -log(1 - predicted probability)

This can be compactly written as a single expression

For data points with y = 1,
loss = -log(predicted probability)

For data points with y = 0,
loss = -log(1 - predicted probability)

$$-y^i \log\left(\underbrace{model(x^i)}_{\text{Predicted probability for the } i^{th} \text{ data point}}\right) - (1 - y^i)\log(1 - \underbrace{model(x^i)}_{\text{Predicted probability for the } i^{th} \text{ data point}})$$

We can now average this across all *n* data points

$$\boxed{\frac{1}{n}\sum_{i=1}^{n}} -y^i \log\left(model(x^i)\right) - (1 - y^i)\log(1 - model(x^i))$$

This is the Binary Cross-Entropy Loss function!

$$\frac{1}{n}\sum_{i=1}^{n} -y^i \log\left(model\left(x^i\right)\right) - \left(1 - y^i\right)\log(1 - model\left(x^i\right))$$

Minimizing loss functions

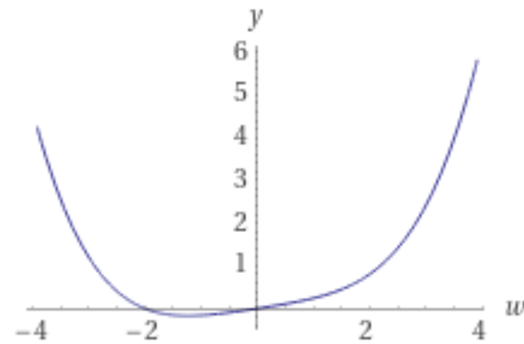
Minimizing functions

- Loss functions are just a particular kind of function so we will first consider the general problem of minimizing an arbitrary function
- After we develop some intuition about how to do this, we will return to the specific task of minimizing a loss function

Minimizing a single-variable function

Let's say we want to minimize the function:

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right)$$



(w from −3.9 to 3.9)

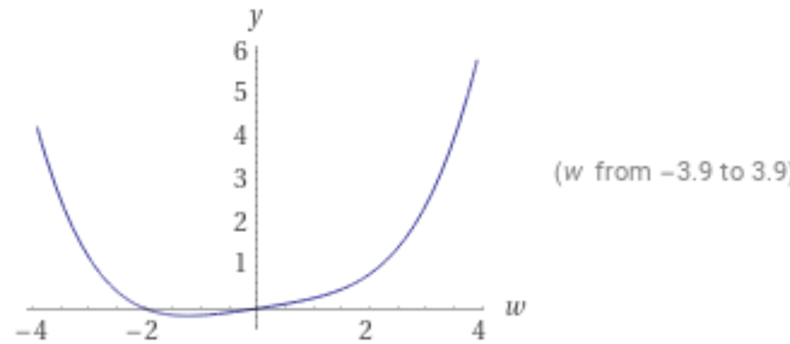
How can we go about this?

https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

Minimizing a single-variable function

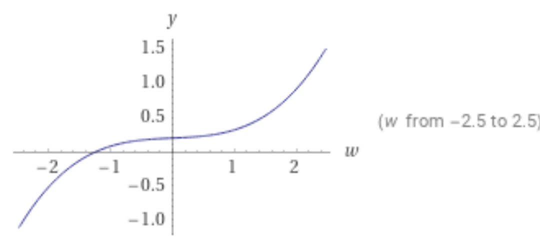
Let's say we want to minimize the function:

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right)$$



Can we use its derivative?

$$\frac{\partial}{\partial w} g\left(w\right) = \frac{2}{25} w^3 + \frac{1}{25} w + \frac{1}{5}$$

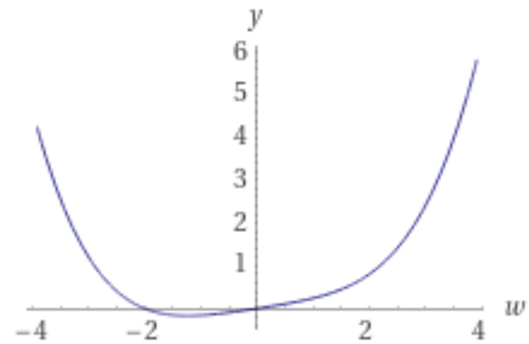


https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

Minimizing a single-variable function

Let’s say we want to minimize the function:

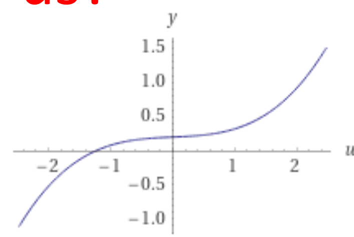
$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right)$$



(w from −3.9 to 3.9)

What does the derivative at a point tell us?

$$\frac{\partial}{\partial w} g(w) = \frac{2}{25} w^3 + \frac{1}{25} w + \frac{1}{5}$$



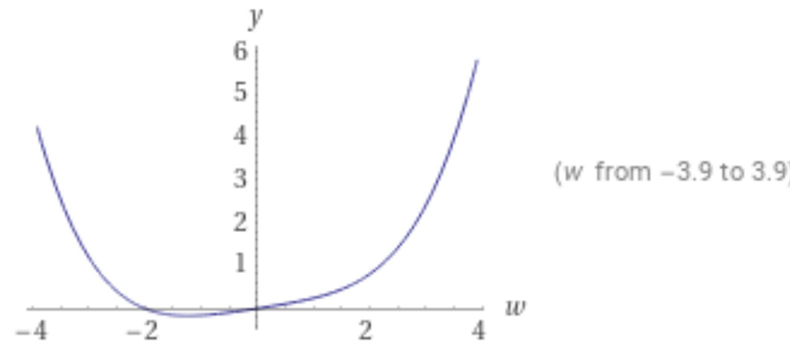
(w from −2.5 to 2.5)

https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

Minimizing a single-variable function

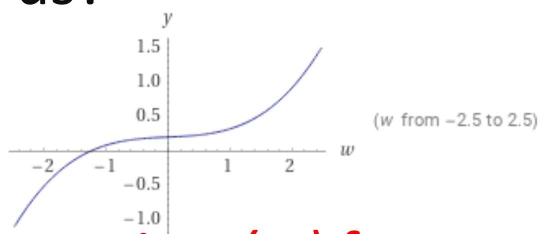
Let's say we want to minimize the function:

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right)$$



What does the derivative at a point tell us?

$$\frac{\partial}{\partial w} g\left(w\right) = \frac{2}{25} w^3 + \frac{1}{25} w + \frac{1}{5}$$

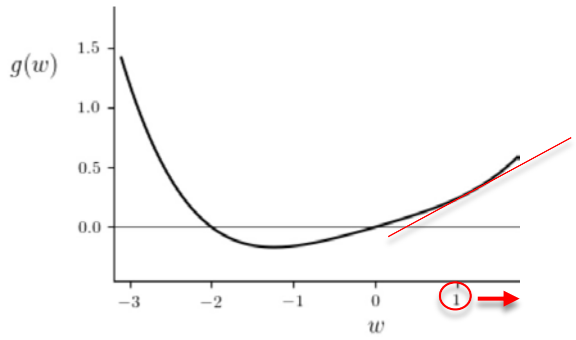


The derivative (or slope) tells us the change in *g(w)* for a small increase in *w*

https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

The value of knowing the derivative

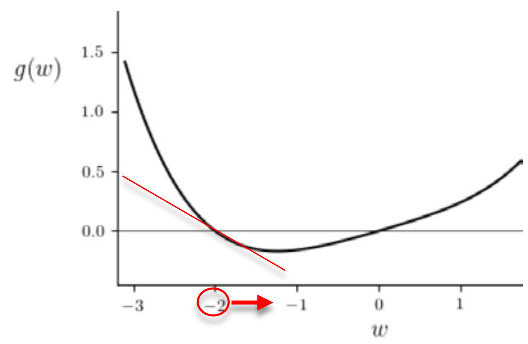
If the derivative at a point *w* is …	What it means
Positive	Increasing *w* slightly will increase *g(w)*



https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

The value of knowing the derivative

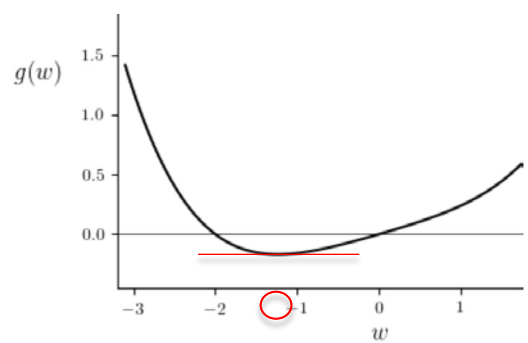
If the derivative at a point *w* is …	**What it means**
Positive	Increasing *w* slightly will increase *g(w)*
Negative	Increasing *w* slightly will decrease *g(w)*



https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

The value of knowing the derivative

If the derivative at a point *w* is …	**What it means**
Positive	Increasing *w* slightly will increase *g(w)*
Negative	Increasing *w* slightly will decrease *g(w)*
~0	Changing *w* slightly won't change *g(w)*



https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

This suggests an algorithm for minimizing $g(w)$

1. Start with some point w

This suggests an algorithm for minimizing $g(w)$

1. Start with some point w
2. Calculate the derivative (i.e., slope) of $g(w)$ at w

This suggests an algorithm for minimizing *g(w)*

1. Start with some point *w*
2. Calculate the derivative (i.e., slope) of *g(w)* at *w*

If the derivative is …	**What it means**	**Since we want to minimize loss, do this …**
Positive	Increasing *w* will increase the loss function	________ w slightly
Negative	Increasing *w* will decrease the loss function	________ w slightly
~0	Changing *w* won't change the loss function	________

This suggests an algorithm for minimizing *g(w)*

1. Start with some point *w*
2. Calculate the derivative (i.e., slope) of *g(w)* at *w*

If the derivative is …	**What it means**	**Since we want to minimize loss, do this …**
Positive	Increasing *w* will increase the loss function	Reduce w slightly
Negative	Increasing *w* will decrease the loss function	Increase w slightly
~0	Changing *w* won't change the loss function	Stop

This suggests an algorithm for minimizing *g(w)*

1. Start with some point *w*
2. Calculate the derivative (i.e., slope) of *g(w)* at *w*

If the derivative is …	What it means	Since we want to minimize loss, do this …
Positive	Increasing *w* will increase the loss function	Reduce w slightly
Negative	Increasing *w* will decrease the loss function	Increase w slightly
~0	Changing *w* won't change the loss function	Stop

3. If you have reached max iterations or run out of time, stop. Else, go to step 2

This is Gradient Descent!

1. Start with some point *w*
2. Calculate the derivative (i.e., slope) of *g(w)* at *w*

If the derivative is …	What it means	Since we want to minimize loss, do this …
Positive	Increasing *w* will increase the loss function	Reduce w slightly
Negative	Increasing *w* will decrease the loss function	Increase w slightly
~0	Changing *w* won't change the loss function	Stop

This can be written compactly as

$$w \leftarrow w - \alpha \frac{dg(w)}{dw}$$

3. If you have reached max iterations or run out of time, stop. Else, go to step 2

Any guesses when Gradient Descent was invented?

Gradient descent was invented in 1847 by Cauchy!

Cauchy, A. (1847). Methode generale pour la res- olution des systemes d'equations simultanees. Comptes Rendus de l'Académie des Sciences, 25. 91

Augustin-Louis Cauchy

Cauchy around 1840. Lithography by Zéphirin Belliard after a painting by Jean Roller.

Gradient Descent

$$w \leftarrow w - \alpha \frac{dg(w)}{dw}$$

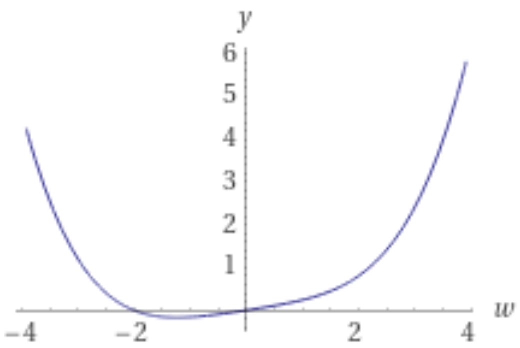
α is called the "learning rate" and is our way of ensuring that we increase or decrease w slightly

Typically set to small values (e.g., 0.1, 0.001, 0.0001) and determined by trial and error

Let’s apply this algorithm to *g(w)*

$$g(w) = \frac{1}{50}\left(w^4 + w^2 + 10w\right)$$

$$\frac{\partial}{\partial w} g\left(w\right) = \frac{2}{25} w^3 + \frac{1}{25} w + \frac{1}{5}$$

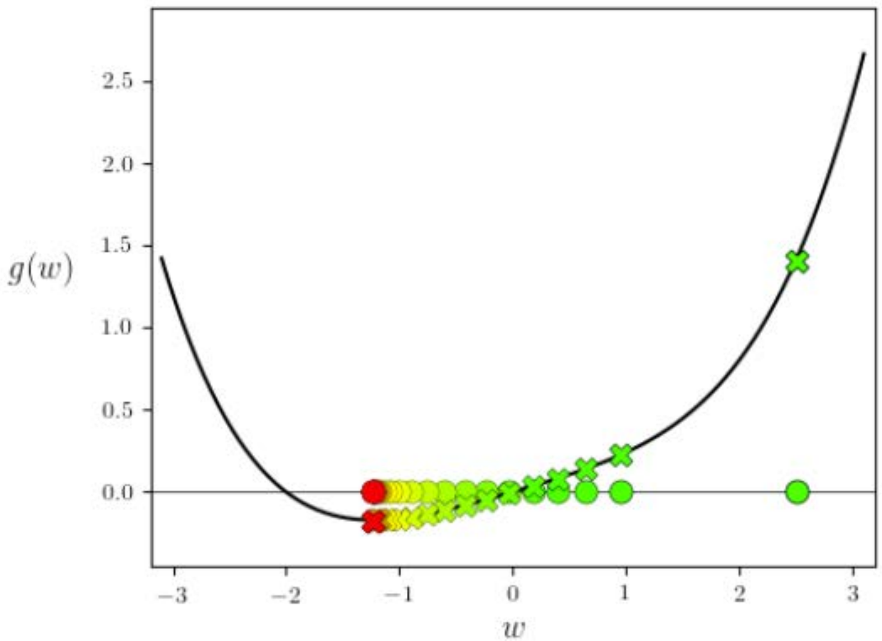


(w from −3.9 to 3.9)

$$w \leftarrow w - \alpha \frac{dg(w)}{dw}$$

https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

Gradient Descent in action



We will start at $w = 2.5$, set $\alpha = 1$ and run the algorithm. In a few iterations, it finds the minimum.

https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

Minimizing a multi-variable function

$$g(w_1, w_2) = {w_1}^2 + {w_2}^2 + 2$$

We can calculate the partial derivative of $g(w_1, w_2)$

$$\left[\frac{\partial g}{\partial w_1}, \frac{\partial g}{\partial w_2}\right] = [2w_1, 2w_2]$$

How should we interpret this?

Minimizing a multi-variable function

$$g(w_1, w_2) = {w_1}^2 + {w_2}^2 + 2$$

$$\nabla g = \left[\frac{\partial g}{\partial w_1}, \frac{\partial g}{\partial w_2}\right] = [2w_1, 2w_2]$$

The first number is the change in *g(w)* for a small increase in *w_1, with w_2 kept unchanged.* The second number is the change in *g(w)* for a small increase in *w_2, with w_1 kept unchanged*

Minimizing a multi-variable function

$$g(w_1, w_2) = {w_1}^2 + {w_2}^2 + 2$$

$$\nabla g = \left[\frac{\partial g}{\partial w_1}, \frac{\partial g}{\partial w_2}\right] = [2w_1, 2w_2]$$

The first number is the change in *g(w)* for a small increase in w_1, *with* w_2 *kept unchanged.* The second number is the change in *g(w)* for a small increase in w_2, *with* w_1 *kept unchanged*

This is called the "gradient" of $g(w_1, w_2)$ and written as ∇g

https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

Minimizing a multi-variable function

$$\nabla g = \left[\frac{\partial g}{\partial w_1}, \frac{\partial g}{\partial w_2}\right] = [2w_1, 2w_2]$$

We can simply do gradient descent on <u>each</u> coordinate by using the corresponding partial derivative.

$$w_1 \leftarrow w_1 - \alpha \left(\frac{\partial g}{\partial w_1}\right)$$

$$w_2 \leftarrow w_2 - \alpha \left(\frac{\partial g}{\partial w_2}\right)$$

https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

Minimizing a multi-variable function

$$\nabla g = [2w_1, 2w_2]$$

$$w_1 \leftarrow w_1 - \alpha \left(\frac{\partial g}{\partial w_1}\right)$$

$$w_2 \leftarrow w_2 - \alpha \left(\frac{\partial g}{\partial w_2}\right)$$

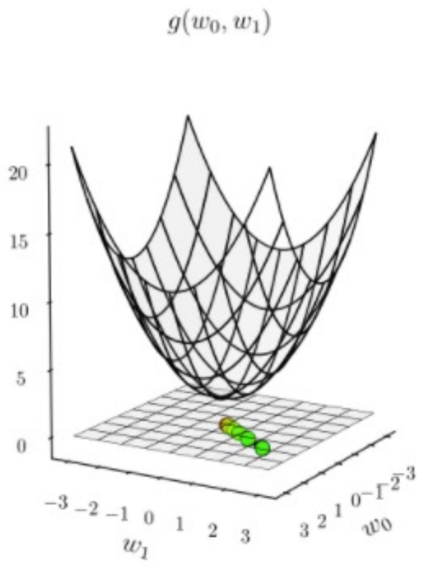
As before, this whole thing can be summarized compactly as:

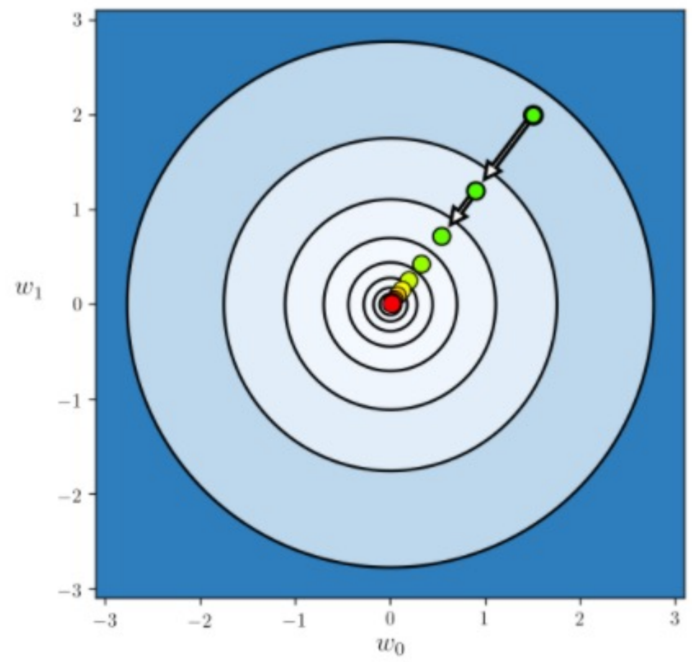
$$w \leftarrow w - \alpha \nabla g(w)$$

https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

Gradient Descent in two dimensions

$$g(w_0, w_1) = {w_0}^2 + {w_1}^2 + 2$$





https://kenndanielso.github.io/mlrefined/blog_posts/6_First_order_methods/6_4_Gradient_descent.html

GD may stop near a local minimum (not necessarily a global minimum) or a saddle point but we don’t worry about this in practice.

Minimizing a loss function with gradient descent

Minimize $\frac{1}{n}\sum_{i=1}^{n} -y^i \log\left(model(x^i)\right) - (1 - y^i)\log(1 - model(x^i))$

What are the variables we need to change to minimize this function?

Minimizing a loss function with gradient descent

Minimize $\frac{1}{n}\sum_{i=1}^{n} -y^i \log\left(model(x^i)\right) - (1-y^i)\log(1-model(x^i))$

What are the variables we need to change to minimize this function?

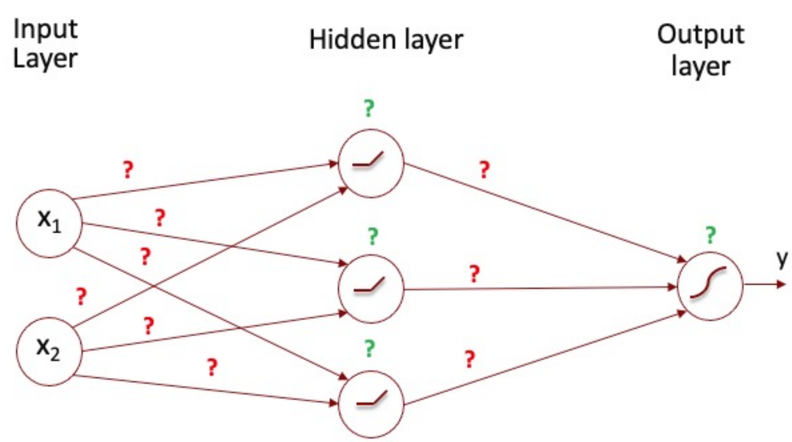
They are the parameters "hiding" inside $model(x_i)$

Minimizing a loss function with gradient descent

Minimize $$\frac{1}{n}\sum_{i=1}^{n} -y^i \log\left(model(x^i)\right) - (1 - y^i)\log(1 - model(x^i))$$

$$model(x^i) = \frac{1}{1 + e^{-(w_1 + w_2 max(0, w_3 + w_4 x_1^i + w_5 x_2^i) + w_6 max(0, w_7 + w_8 x_1^i + w_9 x_2^i) + w_{10} max(0, w_{11} + w_{12} x_1^i + w_{13} x_2^i))}}$$

Recall this model and the NN it represents



Minimizing a loss function with gradient descent

Minimize

$$\frac{1}{n}\sum_{i=1}^{n} -y^i \log\left(model(x^i)\right) - (1-y^i)\log(1-model(x^i))$$

$$model(x^i) = \frac{1}{1+e^{-(w_1+w_2 max(0,w_3+w_4x_1^i+w_5x_2^i)+w_6 max(0,w_7+w_8x_1^i+w_9x_2^i)+w_{10} max(0,w_{11}+w_{12}x_1^i+w_{13}x_2^i))}}$$

w_1, w_2, …, w_{13} are the <u>variables</u> we can change to minimize the loss function

The values of x_1, x_2 and y, on the other hand, are just <u>data</u>

Minimizing a loss function with gradient descent

Minimize

$$\frac{1}{n}\sum_{i=1}^{n} -y^i \log\left(model\left(x^i\right)\right) - \left(1 - y^i\right)\log\left(1 - model\left(x^i\right)\right)$$

$$model(x^i) = \frac{1}{1 + e^{-(w_1 + w_2 max(0, w_3 + w_4 x_1^i + w_5 x_2^i) + w_6 max(0, w_7 + w_8 x_1^i + w_9 x_2^i) + w_{10} max(0, w_{11} + w_{12} x_1^i + w_{13} x_2^i))}}$$

Imagine replacing $model\left(x^i\right)$ with the mathematical expression above wherever $model\left(x^i\right)$ appears in the loss function

Now, your loss function is just a "good old" function of w_1, w_2, …, w_{13} and you can apply gradient descent to it as we normally would.

Backpropagation (aka ‘backprop’)

- Backpropagation is an efficient way to compute the gradient of the loss function
- The efficiency stems from exploiting the layer-by-layer architecture of NNs

Backpropagation (aka ‘backprop’)

- Backpropagation is an efficient way to compute the gradient of the loss function
- The efficiency stems from exploiting the layer-by-layer architecture of NNs
- By organizing the computation in the form of a “computational graph”, we can incrementally calculate the gradient one layer at a time using matrix multiplications (and other simple operations). This approach also eliminates redundant calculations

Backpropagation (aka ‘backprop’)

- Backpropagation is an efficient way to compute the gradient of the loss function
- The efficiency stems from exploiting the layer-by-layer architecture of NNs
- By organizing the computation in the form of a “computational graph”, we can incrementally calculate the gradient one layer at a time using matrix multiplications (and other simple operations). This approach also eliminates redundant calculations
- It turns out that Graphic Processing Units (GPUs), originally invented to speed up video games, are perfectly suited for matrix multiplications!

Backpropagation (aka 'backprop')

- Backpropagation is an efficient way to compute the gradient of the loss function
- The efficiency stems from exploiting the layer-by-layer architecture of NNs
- By organizing the computation in the form of a "computational graph", we can incrementally calculate the gradient one layer at a time using matrix multiplications (and other simple operations). This approach also eliminates redundant calculations
- It turns out that Graphic Processing Units (GPUs), originally invented to speed up video games, are perfectly suited for matrix multiplications!
- Backprop + GPUs ➔ Fast calculation of loss function gradients!

Sections 2.4.3 and 2.4.4 of textbook have more detail. Also see HODL-SP24-Lec-2-Backprop_Example.pdf for a worked-out example

Backpropagation (aka ‘backprop’)

- Backpropagation is an efficient way to compute the gradient of the loss function
- The efficiency stems from exploiting the layer-by-layer architecture of NNs
- By organizing the computation in the form of a “computational graph”, we can incrementally calculate the gradient one layer at a time using matrix multiplications (and other simple operations). This approach also eliminates redundant calculations
- It turns out that Graphic Processing Units (GPUs), originally invented to speed up video games, are perfectly suited for matrix multiplications!
- Backprop + GPUs ➔ Fast calculation of loss function gradients!

Sections 2.4.3 and 2.4.4 of textbook have more detail.

Please see HODL-SP24-Lec-2-Backprop_Example.pdf for a step-by-step example

Gradient Descent → Stochastic Gradient Descent

Making Gradient Descent work with large datasets

- <u>Problem</u>: For large datasets (e.g., n in the millions), computing the gradient of the loss function can be very expensive

Making Gradient Descent work with large datasets

- Problem: For large datasets (e.g., *n* in the millions), computing the gradient of the loss function can be very expensive

$$w \leftarrow w - \alpha \nabla g(w)$$

- The Solution:
  - At each iteration, instead of using all the n data points in the calculation of the gradient of the loss function, randomly choose just a few of the n observations (called a *minibatch*) and use only these observations to compute the partial derivatives.

Making Gradient Descent work with large datasets

- Problem: For large datasets (e.g., *n* in the millions), computing the gradient of the loss function can be very expensive

- The Solution:
  - At each iteration, instead of using all the n data points in the calculation of the gradient of the loss function, randomly choose just a few of the n observations (called a *minibatch*) and use only these observations to compute the partial derivatives.
  - This is called Stochastic Gradient Descent (SGD)*

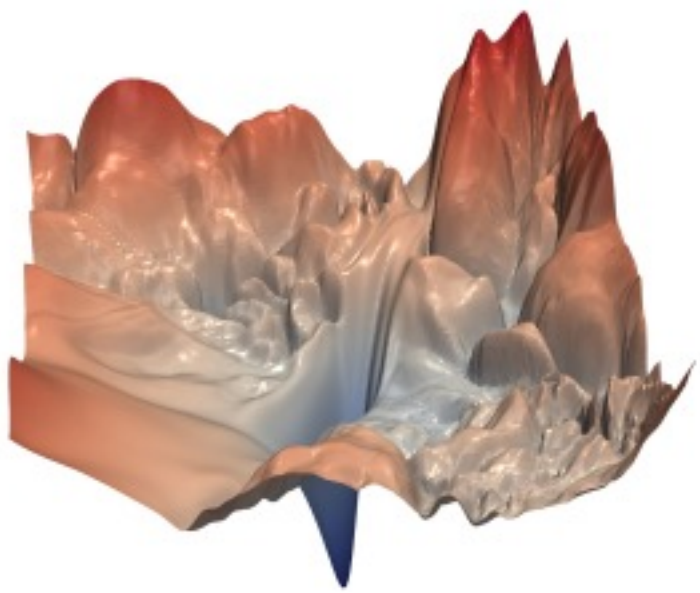
* Strictly speaking, SGD chooses just one observation. What we are describing here is Minibatch Gradient Descent but the term SGD is widely used in the field to describe the latter so we will do the same

Making Gradient Descent work with large datasets

- Problem: For large datasets (e.g., *n* in the millions), computing the gradient of the loss function can be very expensive

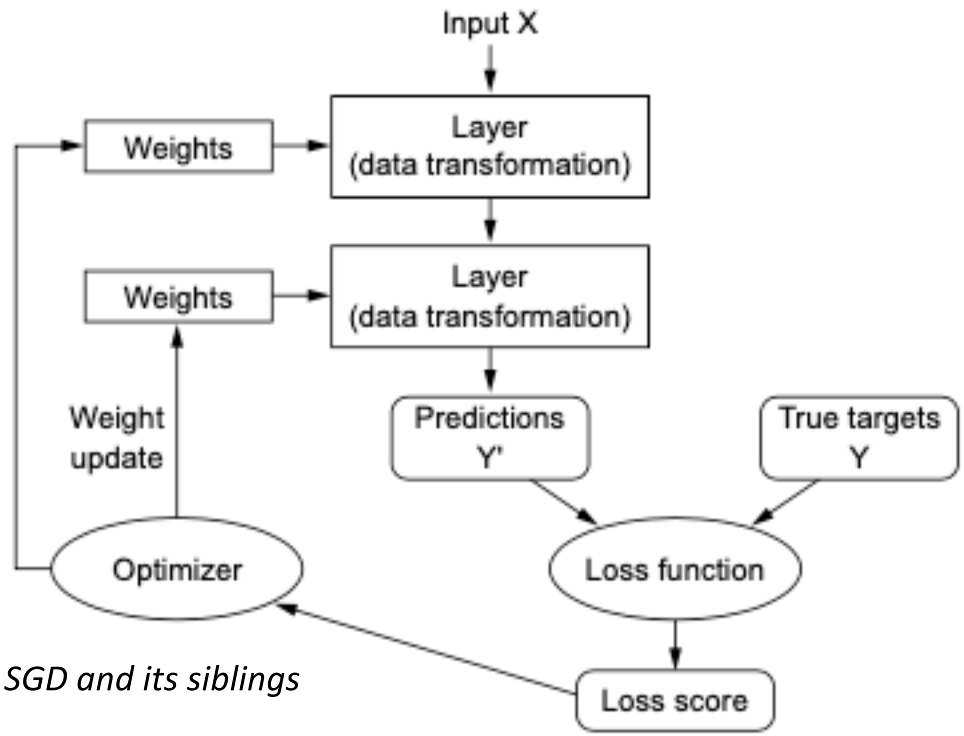
- The Solution:
  - At each iteration, instead of using all the n data points in the calculation of the gradient of the loss function, randomly choose just a few of the n observations (called a *minibatch*) and use only these observations to compute the partial derivatives.
  - This is called Stochastic Gradient Descent (SGD)
  - Because not all n data points are used in the calculation, this only approximates the true gradient but nevertheless works well in practice. In fact, because it is only an approximation of the true gradient, it can sometimes escape local minima.
  - SGD comes in many “flavors” and we will use a flavor called “Adam” as our default in HODL

Visualization of an actual DL loss function landscape



https://arxiv.org/pdf/1712.09913.pdf

Summary of overall training flow



SGD and its siblings

Figure 2.26 Relationship between the network, layers, loss function, and optimizer

Image: Page 61 of textbook