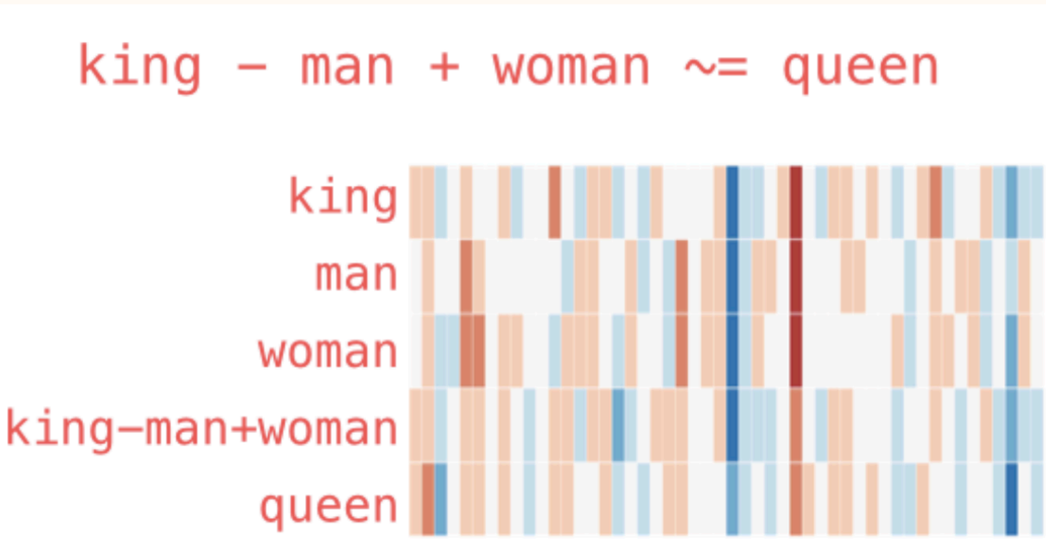
Deep Learning Geometry

CM3-CS

CM3-Computer Science

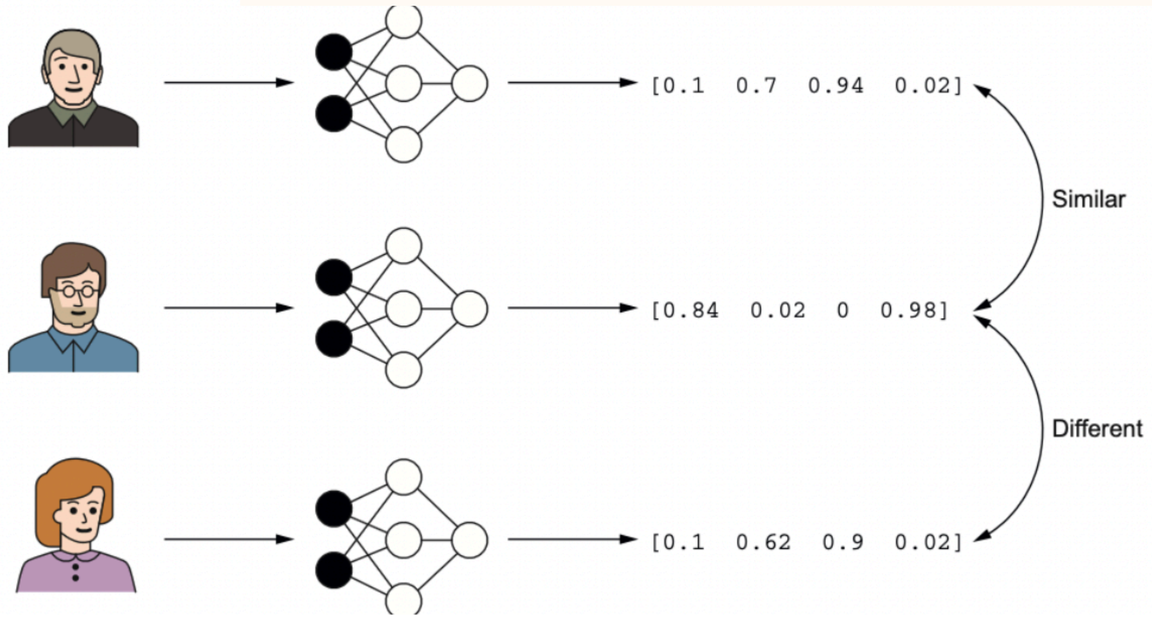
So far…embeddings…



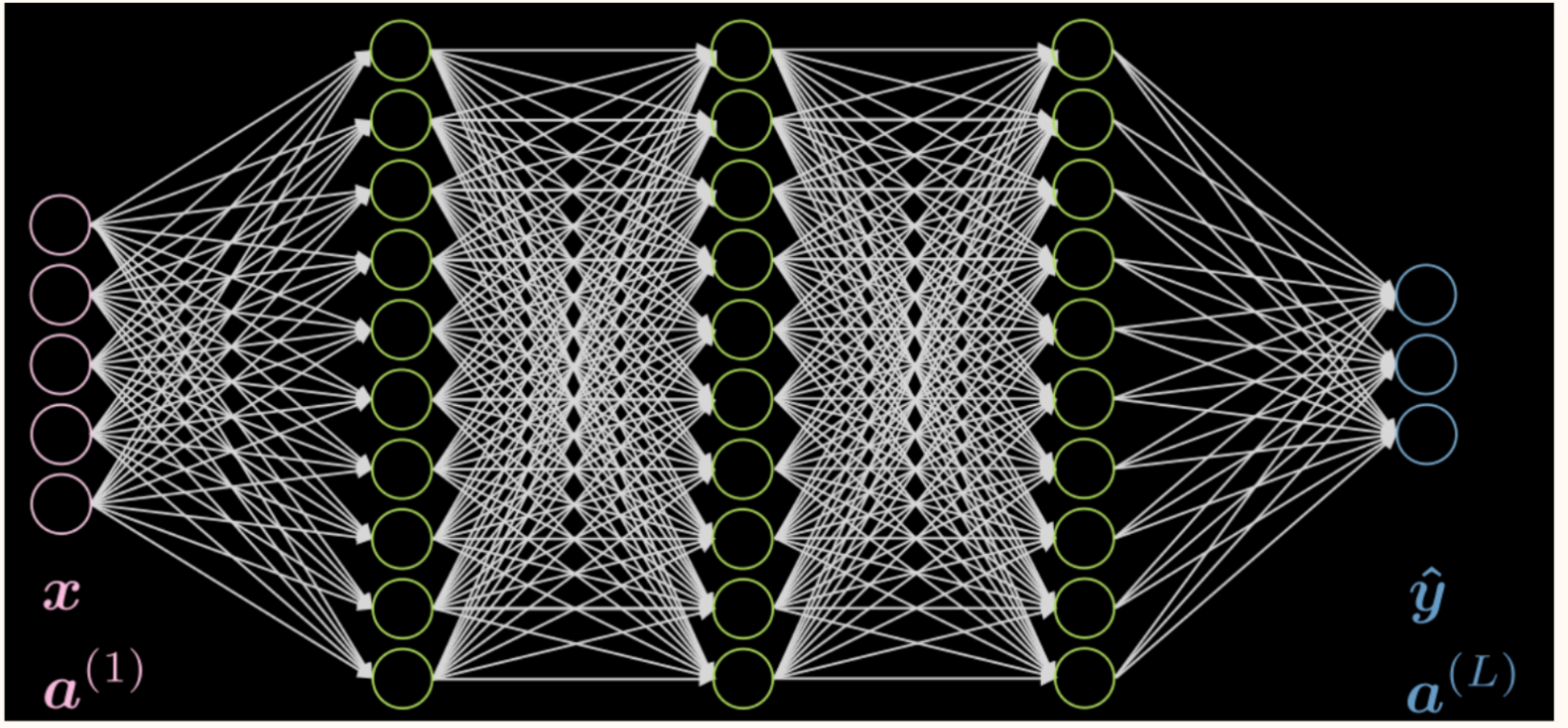
Embeddings map similar concepts to similar vectors

These concepts can be from different spaces, like users and movies

Any model can be used to produce embeddings: Apple face recognition.

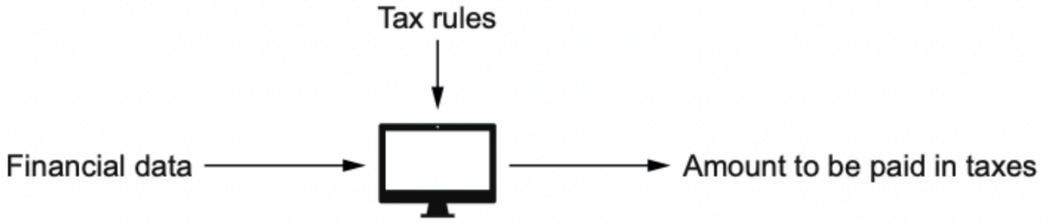


Neural Networks are complex models

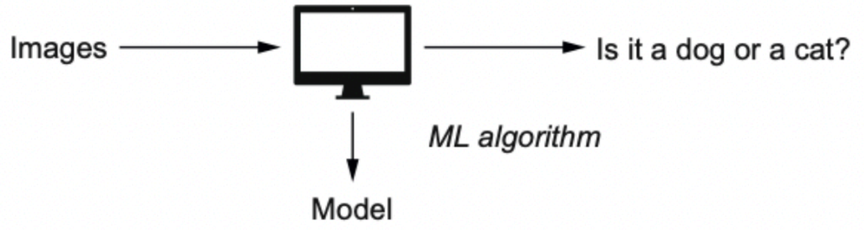


The structure of a neural network. Input nodes (pink) are connected to multiple layers of the network, and finally an output is produced to feed to a loss function. Each neuron rotates, shears, and translates its input, and then makes a non-linear transform on it like setting all negative numbers to 0. Image by Alfredo Canziani.

Traditional programming—problems with known rules and relationships



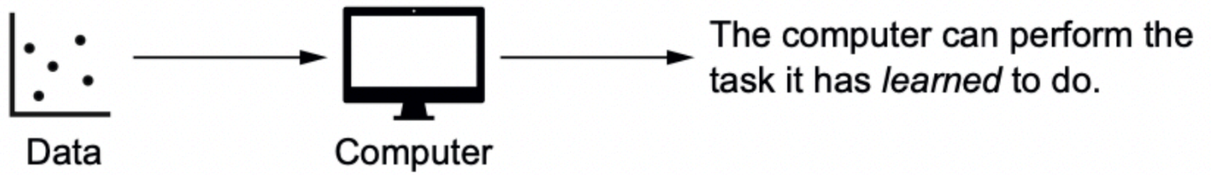
Machine learning—problems that can't be explained analytically but can be learned from experience



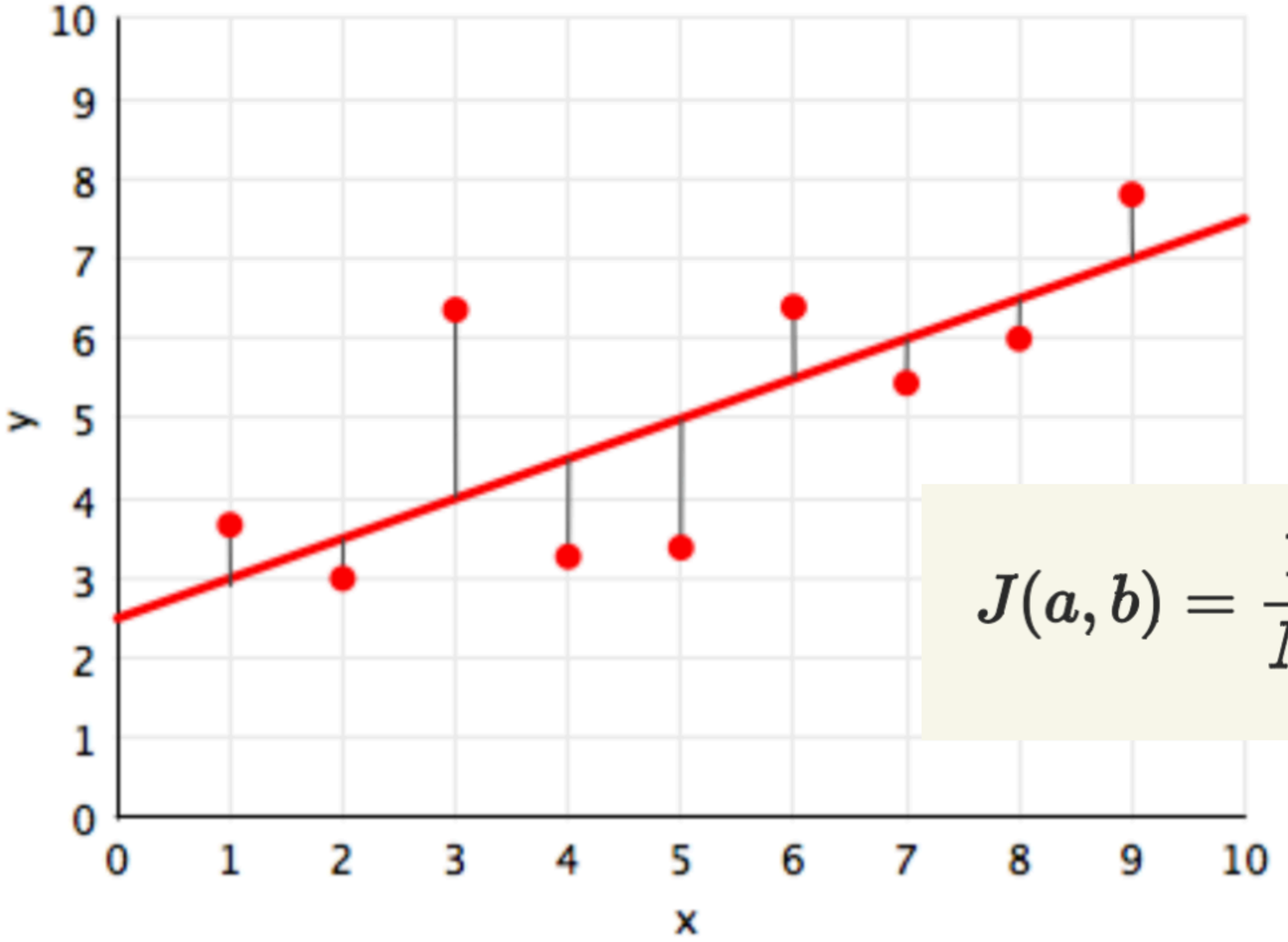
Traditional Programming vs Machine Learning/AI

Traditional Programming

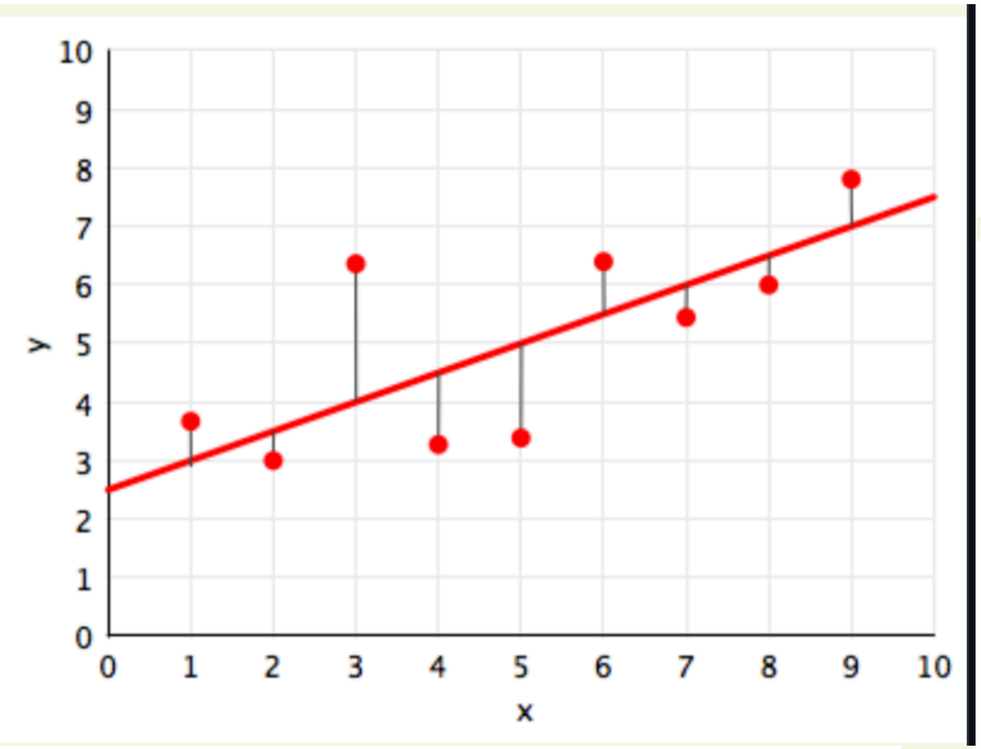
Machine Learning



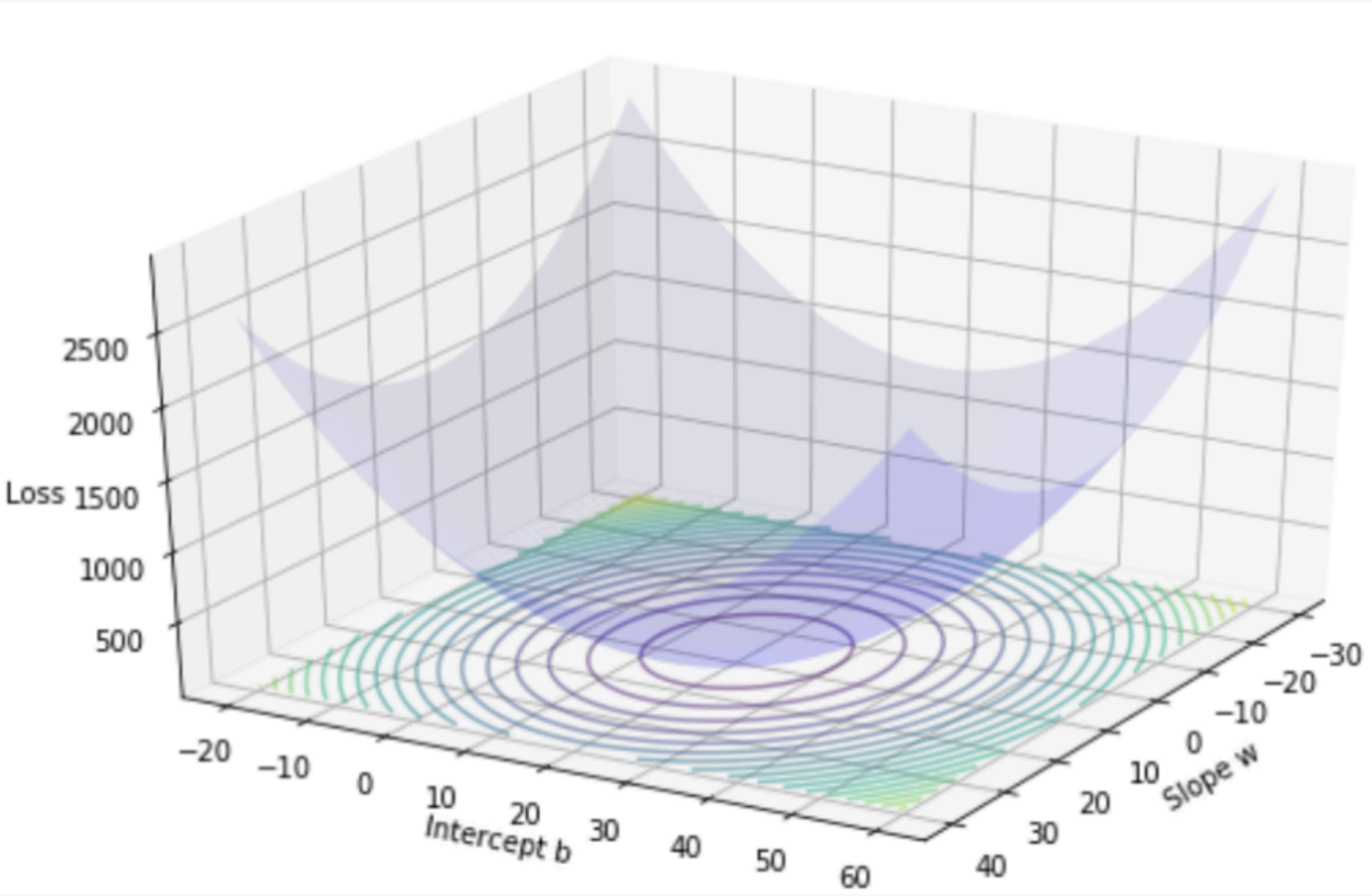
Start with a simple one: linear regression



$$J(a,b) = \frac{1}{N}\sum_i (y_i - (a + bx_i))^2$$



$$J(a,b) = \frac{1}{N}\sum_i (y_i - (a + bx_i))^2$$



Gradient Descent

$$\bar{\theta} \rightarrow \bar{\theta} - \eta \nabla_{\bar{\theta}} J(\theta) = \theta - \frac{\eta}{N} \sum_{i=1}^{m} \nabla_{\bar{\theta}} J_i(\theta)$$

where η is the learning rate.

ENTIRE DATASET NEEDED

```
for i in range(n_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad`
```


Plunge Step! This is Gradient Descent.

Minimizing Loss..Gradient Descent

Wherever you are on the mountain, plunge down the steepest direction you possibly can.

You will make it down to a valley, but perhaps not the one you want to be in.

Your step size is called the **learning rate**.

Too small a step size and you will freeze.

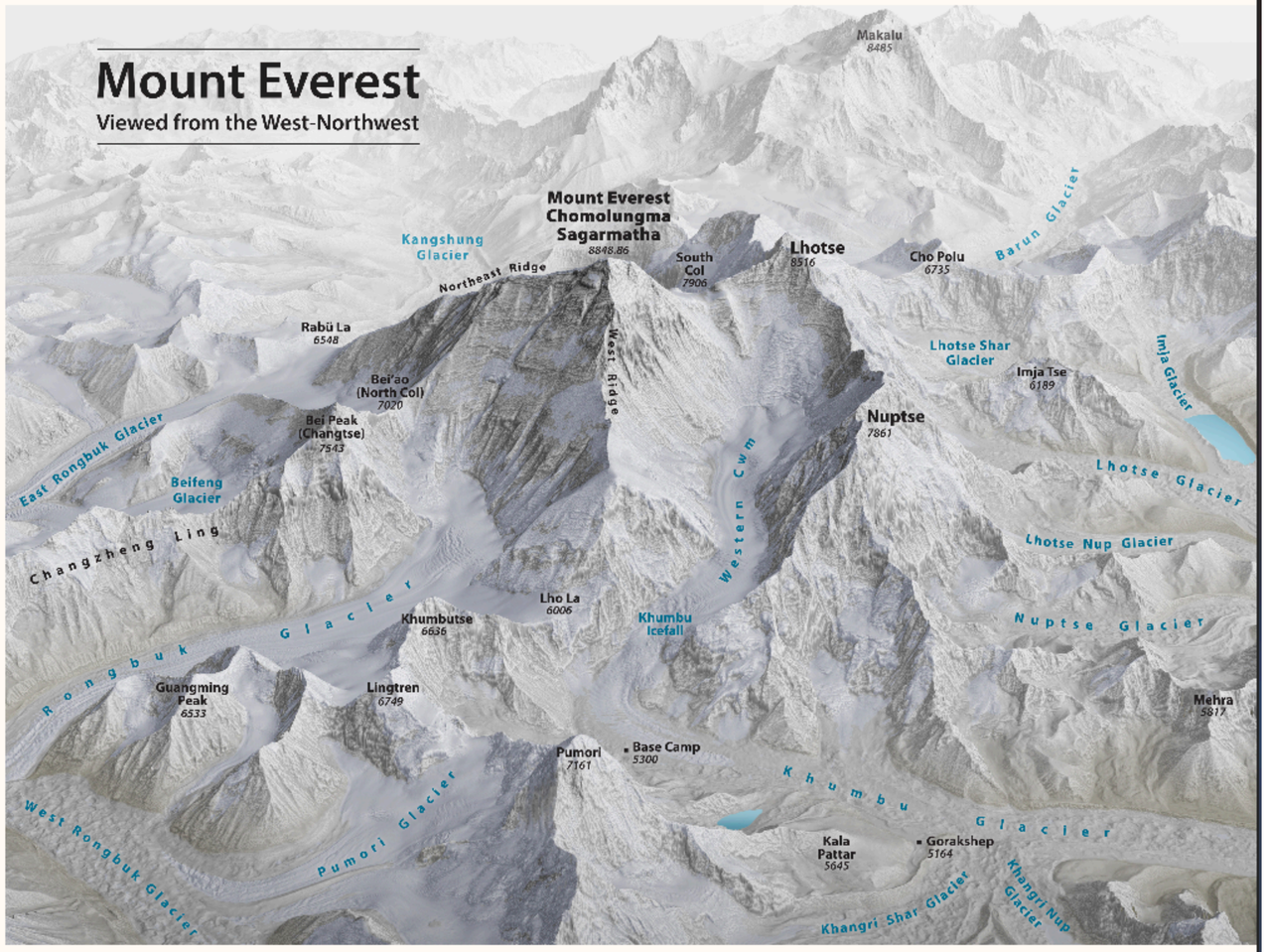
But if you are a giant with a large step size you might just step into another valley.

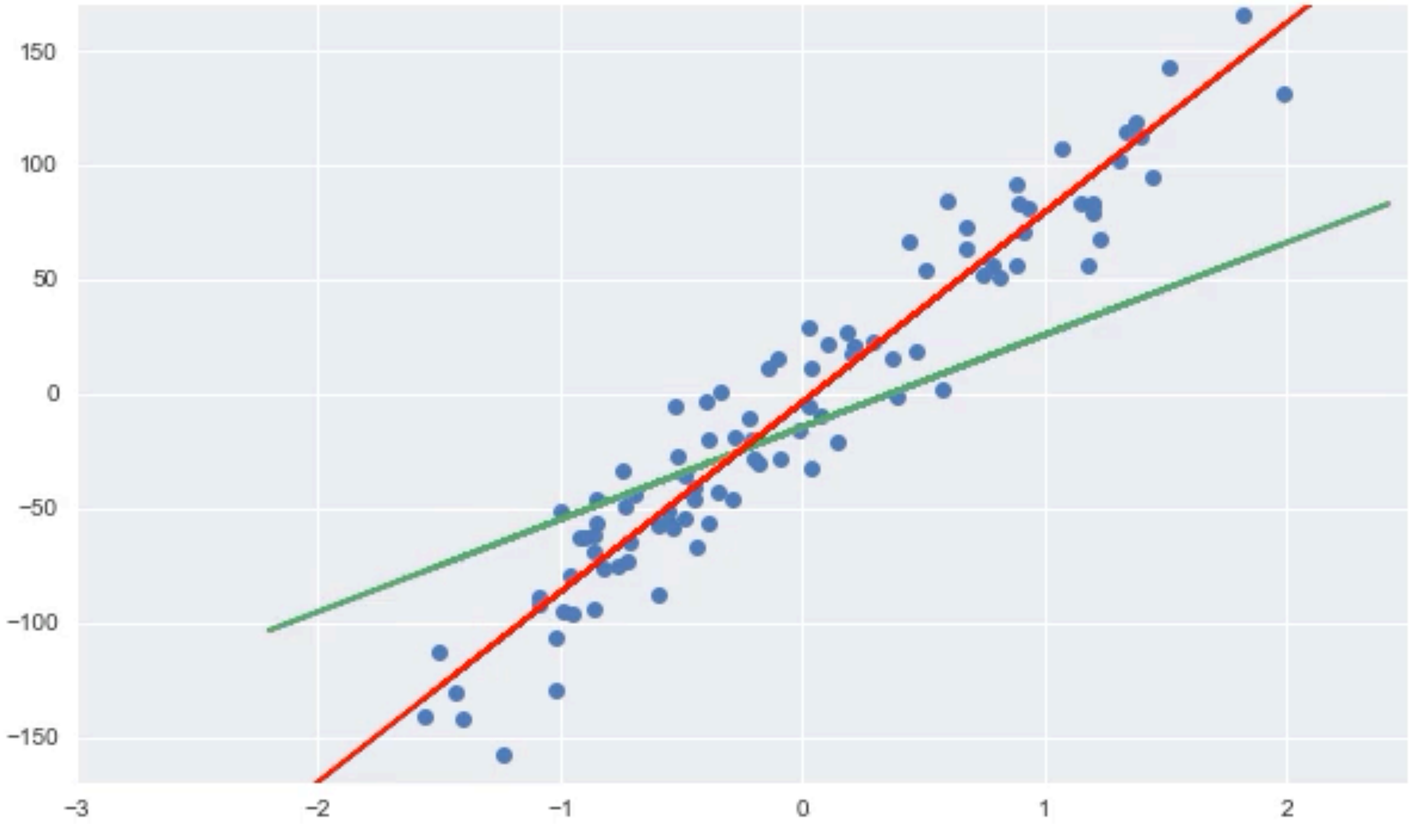
Gradient Descent

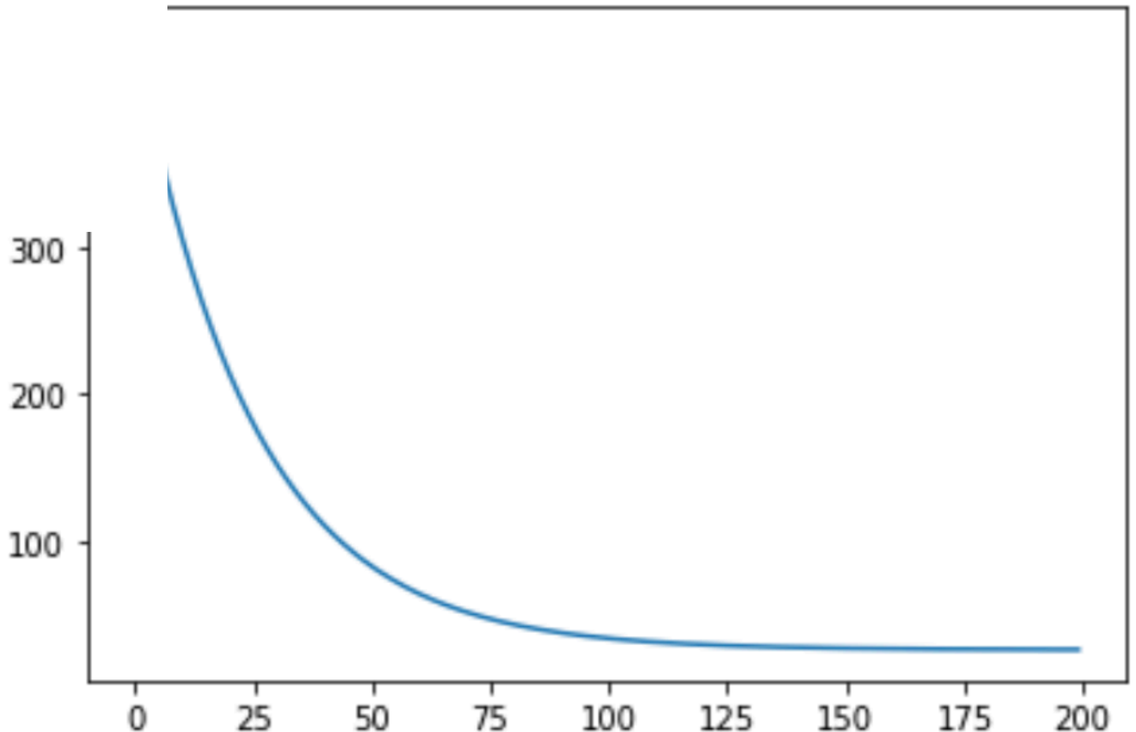
To find the steepest descent, find the place with the highest "slope", or highest derivative.

To go downwards, go in the opposite direction to this highest derivative.

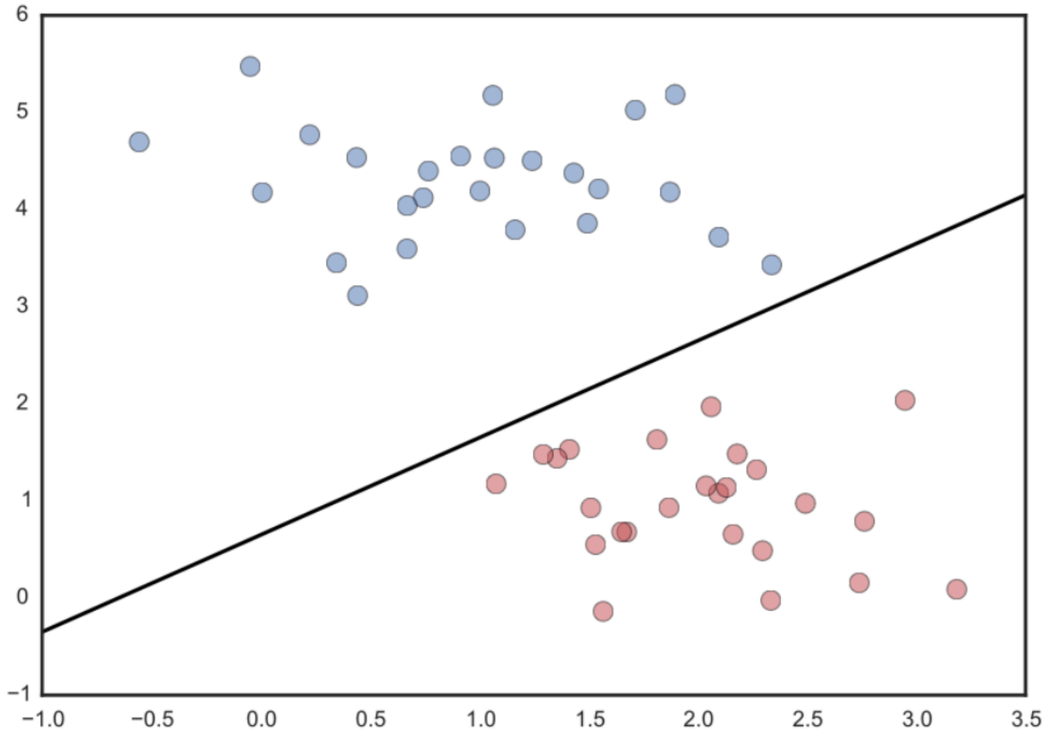
A step to the right from the top of Everest will make you go down into the Western cwm. If you are an ant, you have too small a step: you run out of oxygen. But if you are a giant roaming the earth you could step past Nuptse!




150
100
50
0
−50
−100
−150
−3
−2
−1
0
1
2


300
200
100
0
25
50
75
100
125
150
175
200

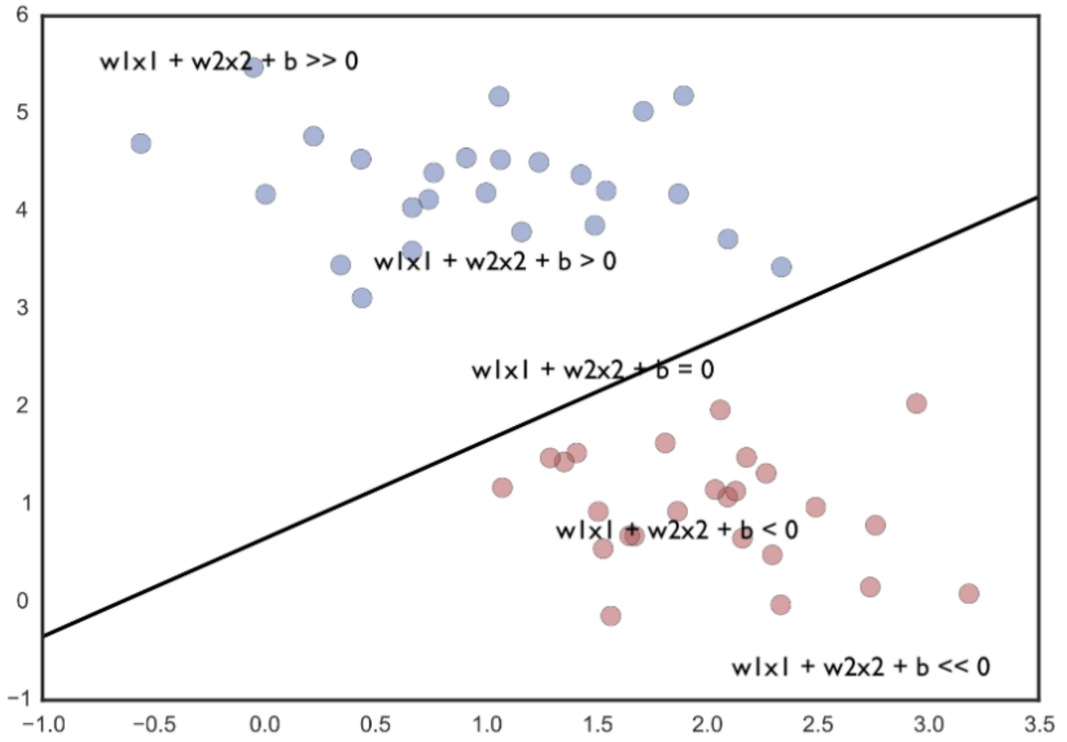
CLASSIFICATION



- will a customer churn?
- is this a check? For how much?
- a man or a woman?
- will this customer buy?
- do you have cancer?
- is this spam?
- whose picture is this?
- what is this text about?[j]

[j] image from code in http://bit.ly/1Azg29G

Logistic regression..split via line



Draw a line in feature space that divides the '1' (blue) samples from the '0' (red)samples.

Now, a line has the form
$w_1 x_1 + w_2 x_2 + b = 0$ in 2-dimensions.

Our classification rule then becomes:

$$y = 1, \quad \mathbf{w} \cdot \mathbf{x} + b \geq 0$$
$$y = 0, \quad \mathbf{w} \cdot \mathbf{x} + b < 0$$

Highly positive and negative values go far from this line!

Sigmoid Function

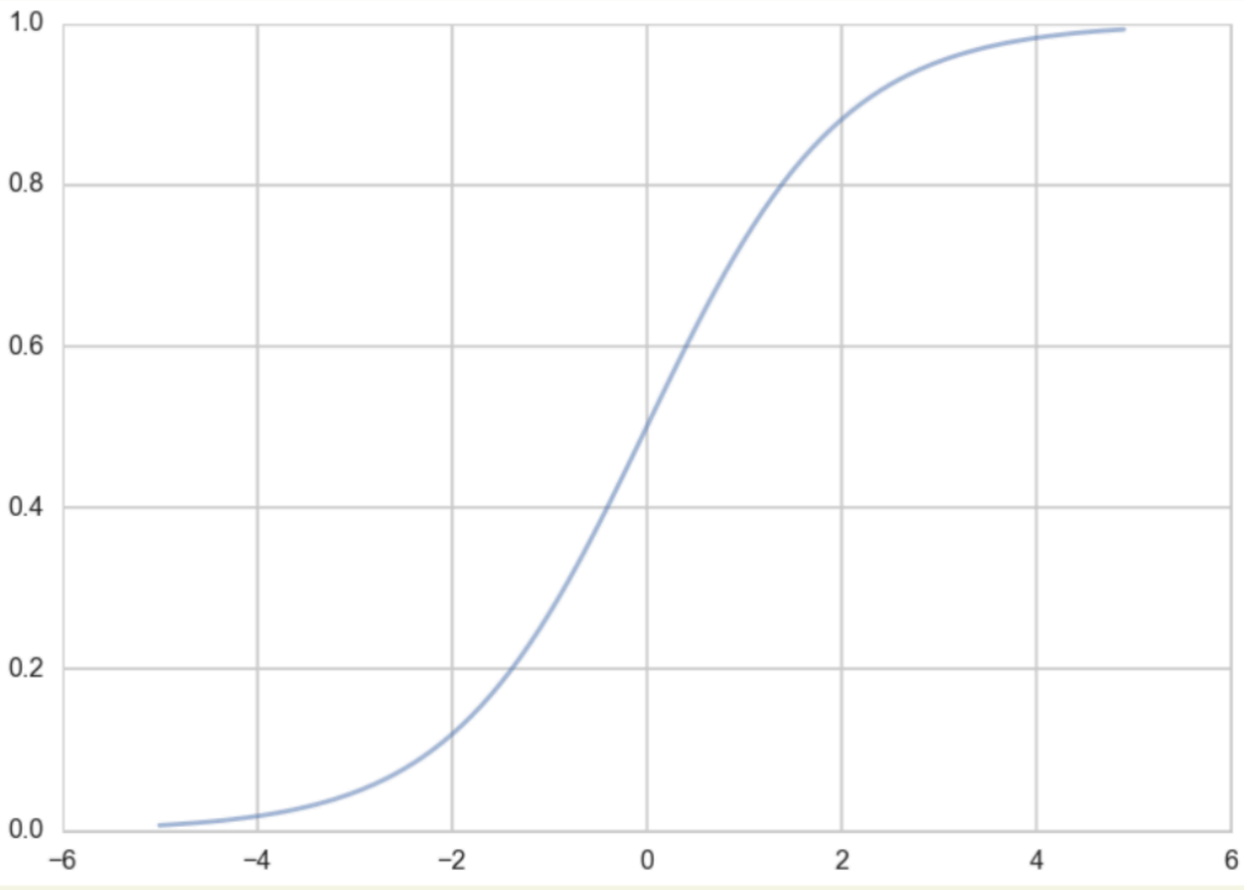
Consider the **sigmoid** function:

$h(z) = \dfrac{1}{1+e^{-z}}$ with the identification

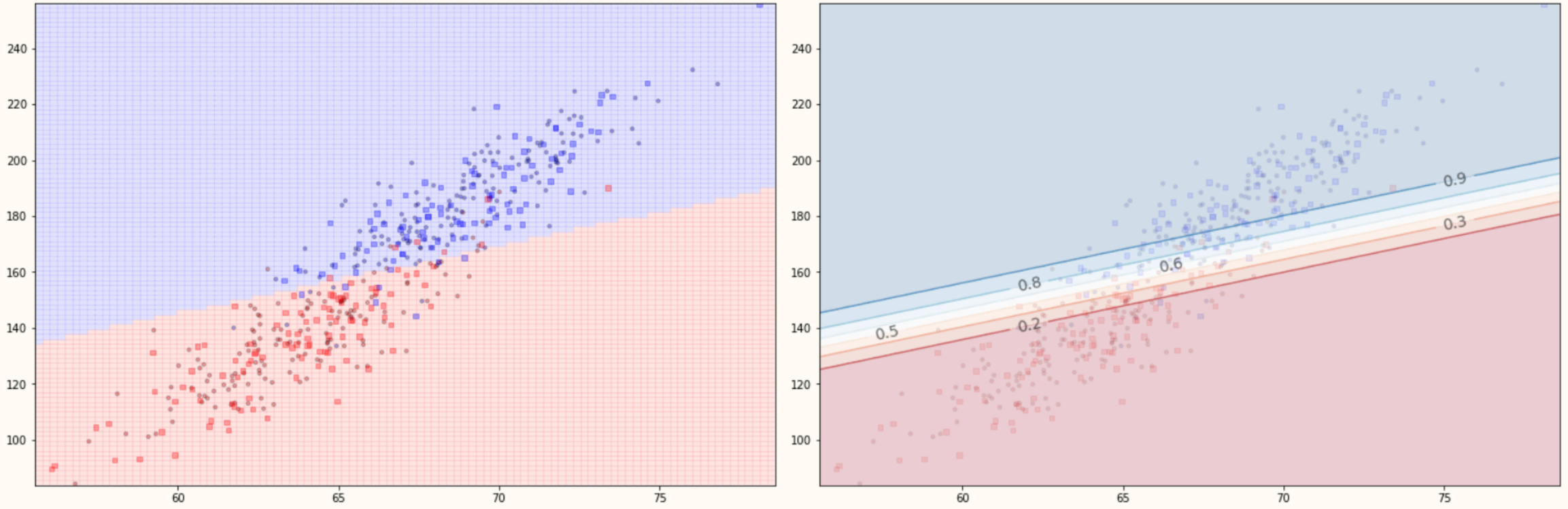
$z = \mathbf{w} \cdot \mathbf{x} + b$

- At $z = 0$ this function has the value 0.5.
- If $z > 0$, $h > 0.5$ and as $z \to \infty$, $h \to 1$.
- If $z < 0$, $h < 0.5$ and as $z \to -\infty$, $h \to 0$.

As long as we identify any value of $h > 0.5$ as classified to '1', and any $h < 0.5$ as 0, we can achieve what we wished above.

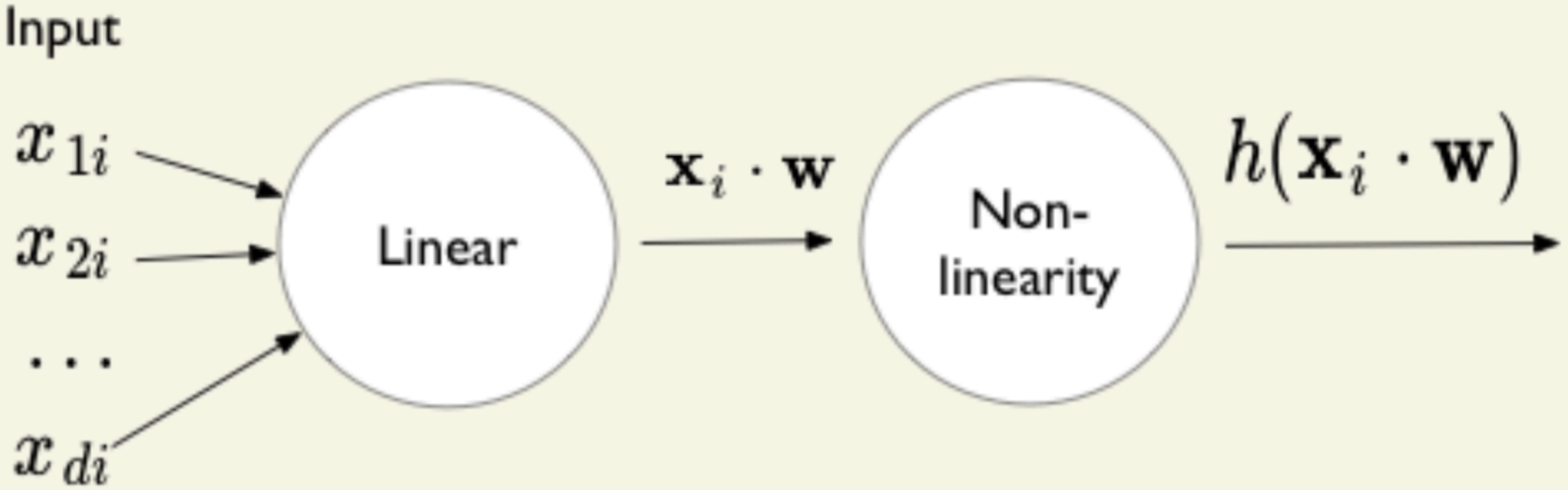


2-D Using Logistic regression

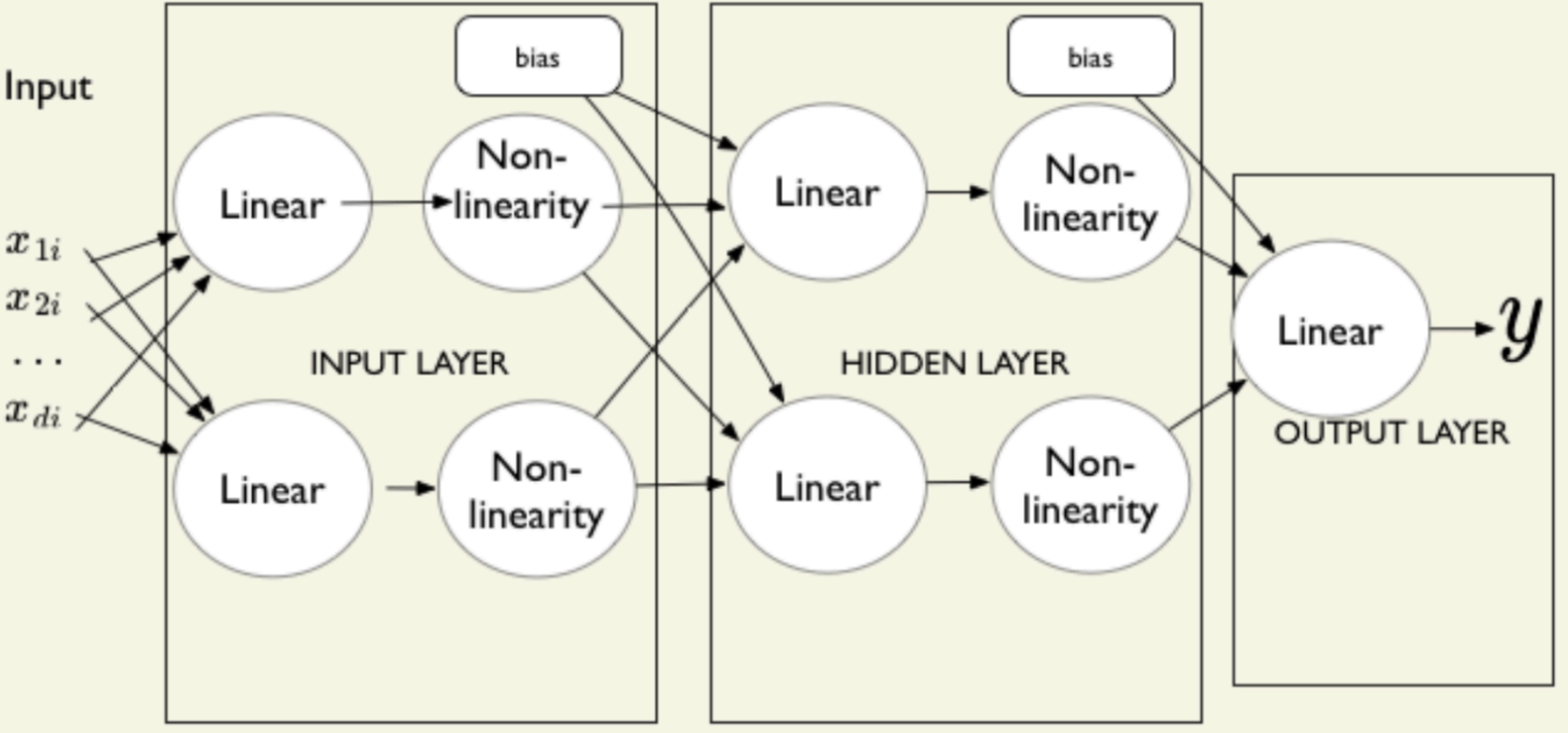


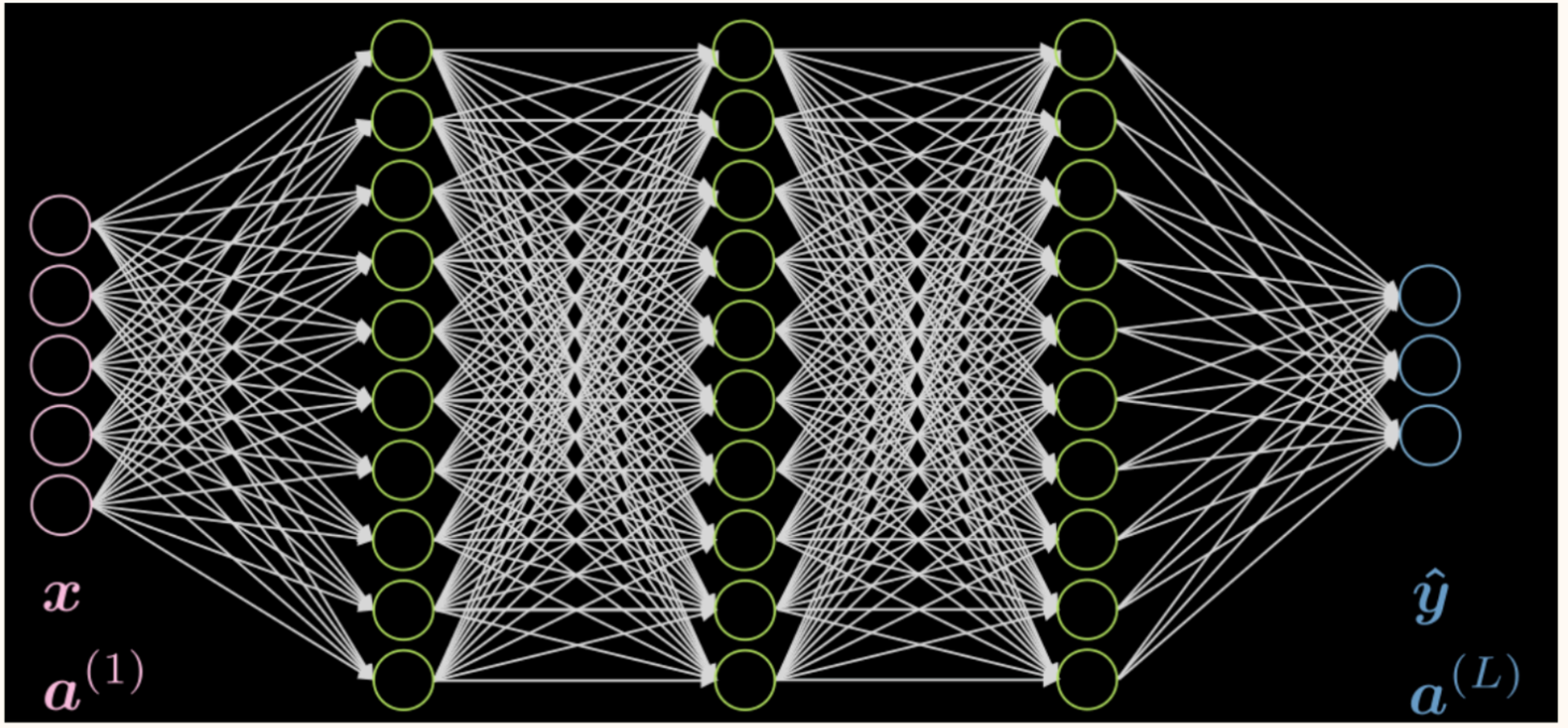
left, classification boundary; right, probability contours

Feed Forward Neural Nets: The perceptron



Combine Perceptrons with non-linearity





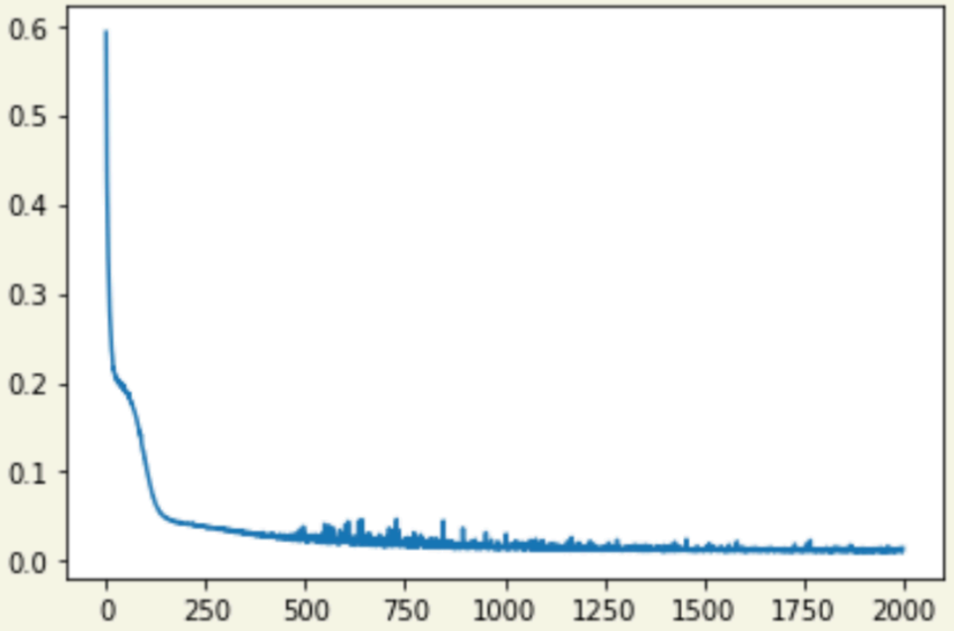
The structure of a neural network. Input nodes (pink) are connected to multiple layers of the network, and finally an output is produced to feed to a loss function. Each neuron rotates, shears, and translates its input, and then makes a non-linear transform on it like setting all negative numbers to 0. Image by Alfredo Canziani.

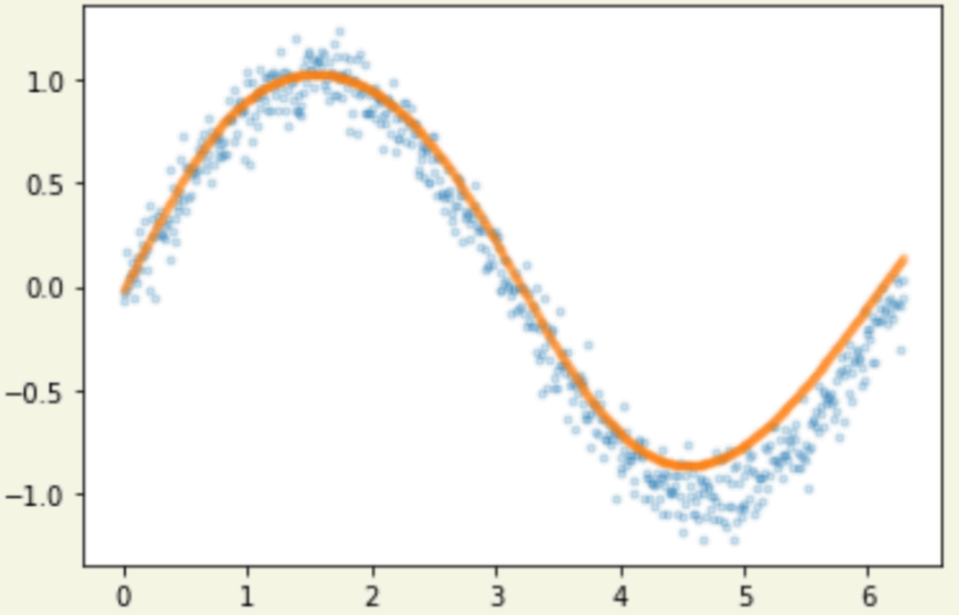
```
layers = [ Affine("first", 1, 4),
           Sigmoid("sigmoid"),
           Affine("last", 4, 1)
]
model = Model(layers)
```

$$\hat{f} = f_\theta(x)$$

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}(f_\theta(x_i) - y_i)^2$$

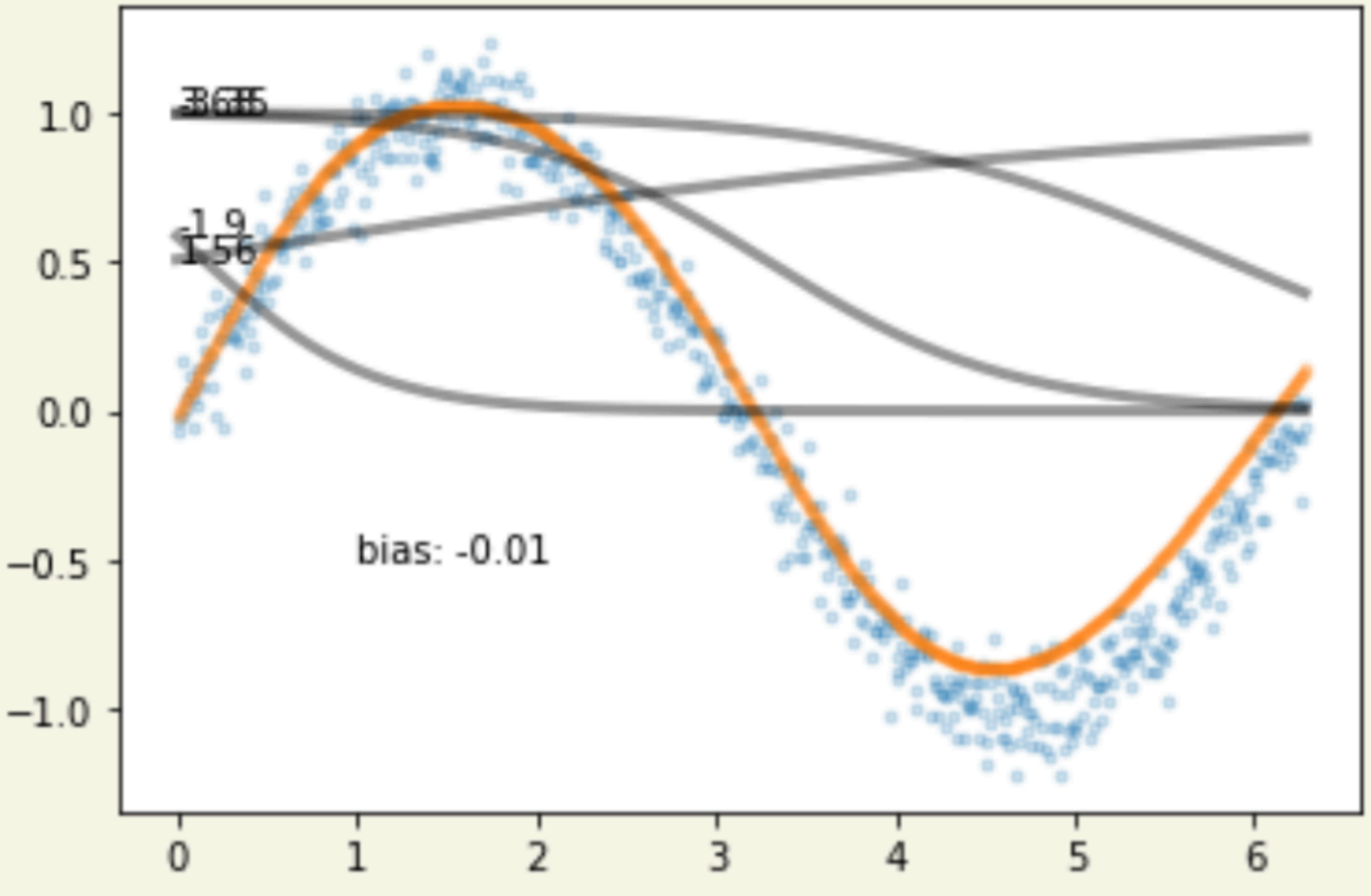
Train!



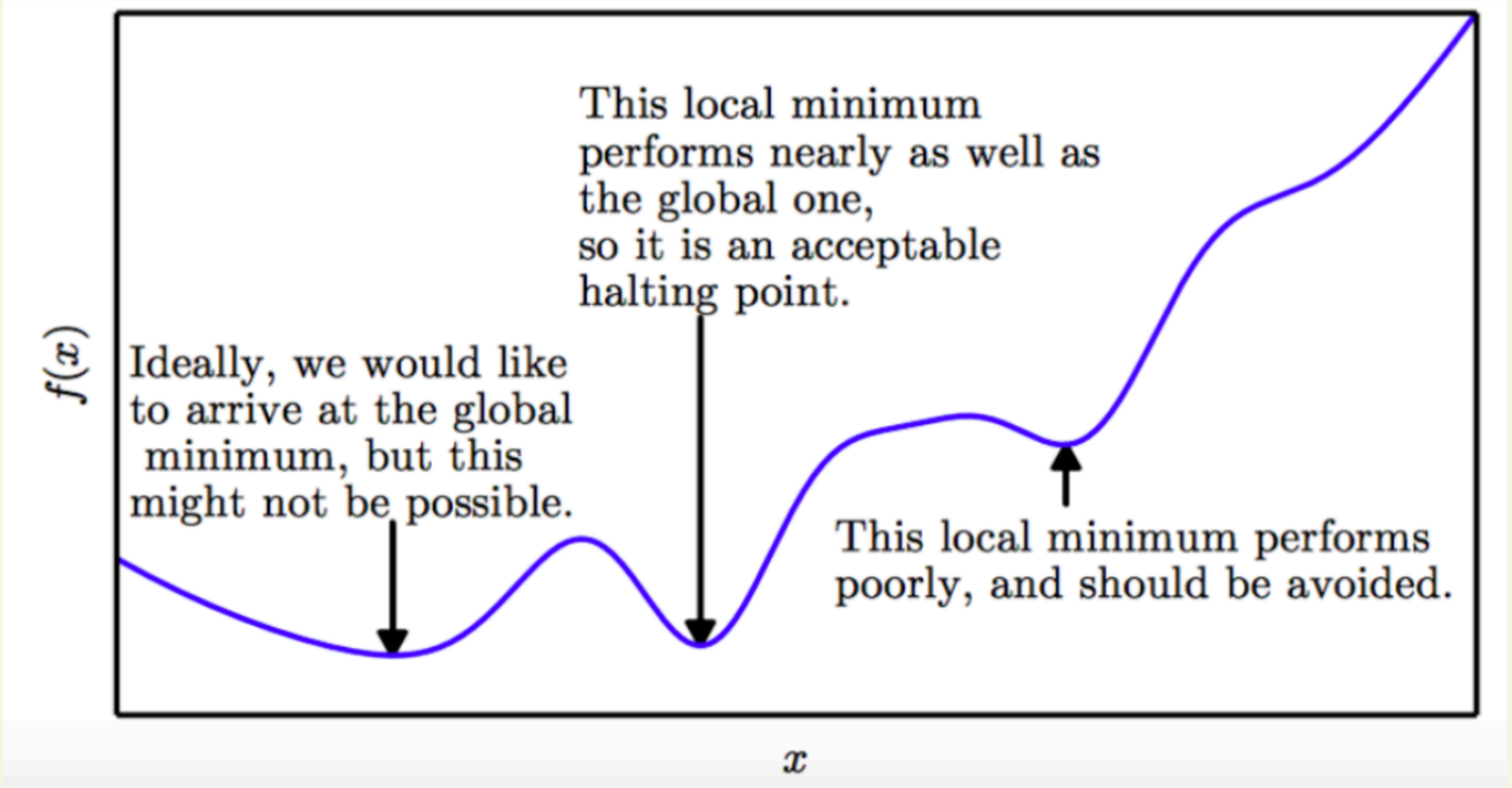


What is the hidden layer doing?

```
layers = [ Affine("first", 1, 4),
           Sigmoid("sigmoid"),
           Affine("last", 4, 1)
]
model = Model(layers)
```



The Loss is NOT CONVEX



How are the models learnt?

Mini-Batch SGD (the most used)

$$\bar{\theta} \rightarrow \bar{\theta} - \eta \nabla_{\bar{\theta}} J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

```
for i in range(mb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

Gradient Descent

$$\bar{\theta} \rightarrow \bar{\theta} - \eta \nabla_{\bar{\theta}} J(\theta) = \theta - \frac{\eta}{N} \sum_{i=1}^{m} \nabla_{\bar{\theta}} J_i(\theta)$$

where η is the learning rate.

ENTIRE DATASET NEEDED

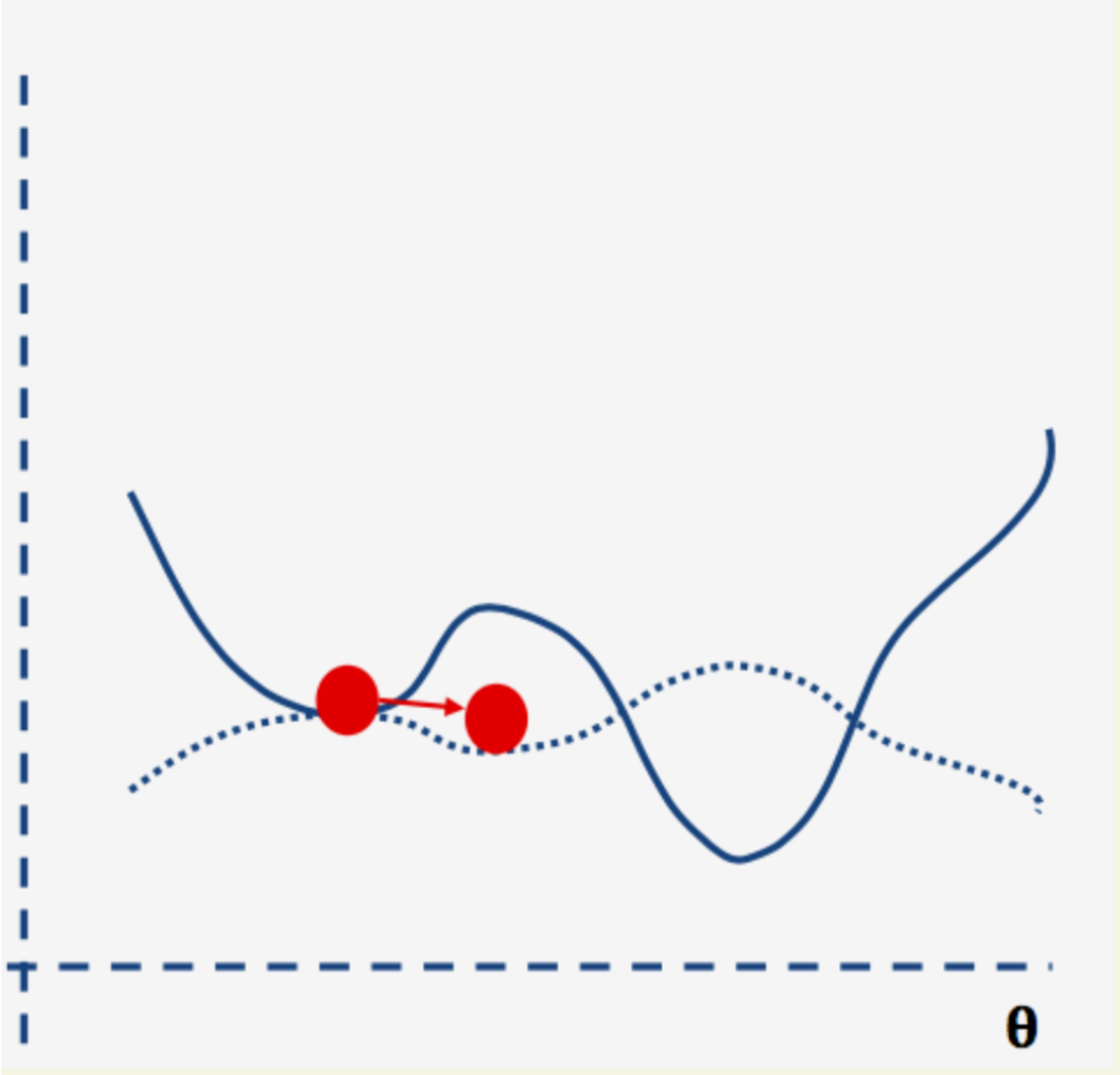
```
for i in range(n_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad`
```

Mini-Batch SGD (the most used)

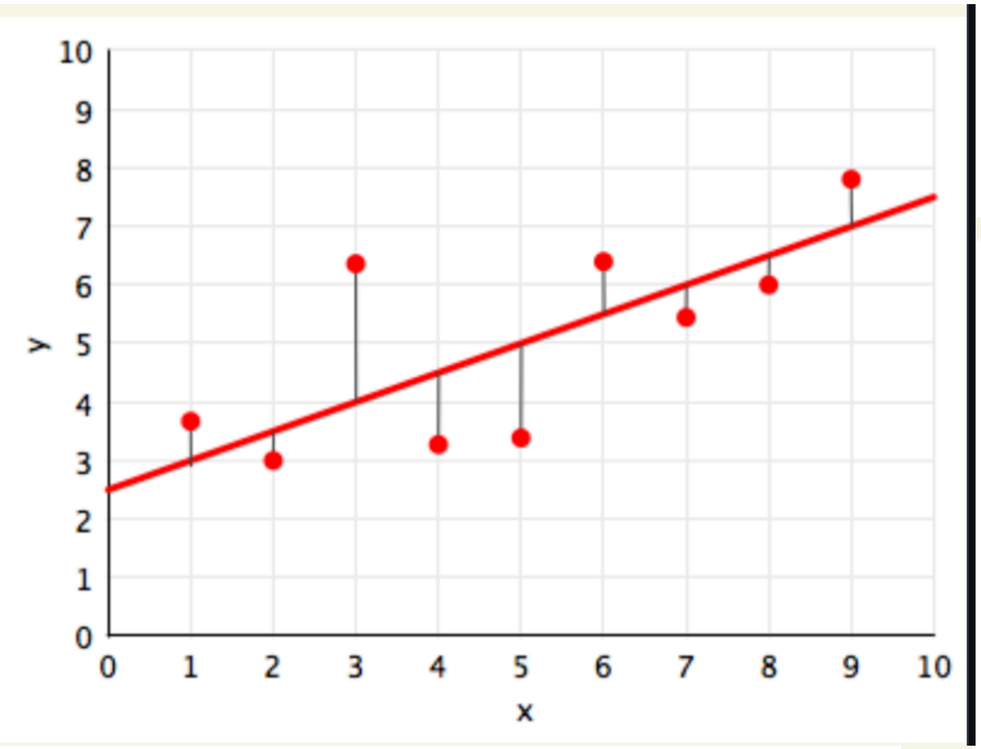
$$\bar{\theta} \rightarrow \bar{\theta} - \eta \nabla_{\bar{\theta}} J(\theta; x^{(i:i+n)}; y^{(i:i+n)})$$

```
for i in range(mb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

The advantage of SGD and batches



- In linear regression the loss function is convex because it is quadratic
- In Neural networks it is not
- So we might get stuck in a local minimum
- Because we use partial data our geometry will differ
- SAMPLING introduces stochasticity, and helps us jump overshallow local minima



$$J(a,b) = \frac{1}{N}\sum_i (y_i - (a + bx_i))^2$$

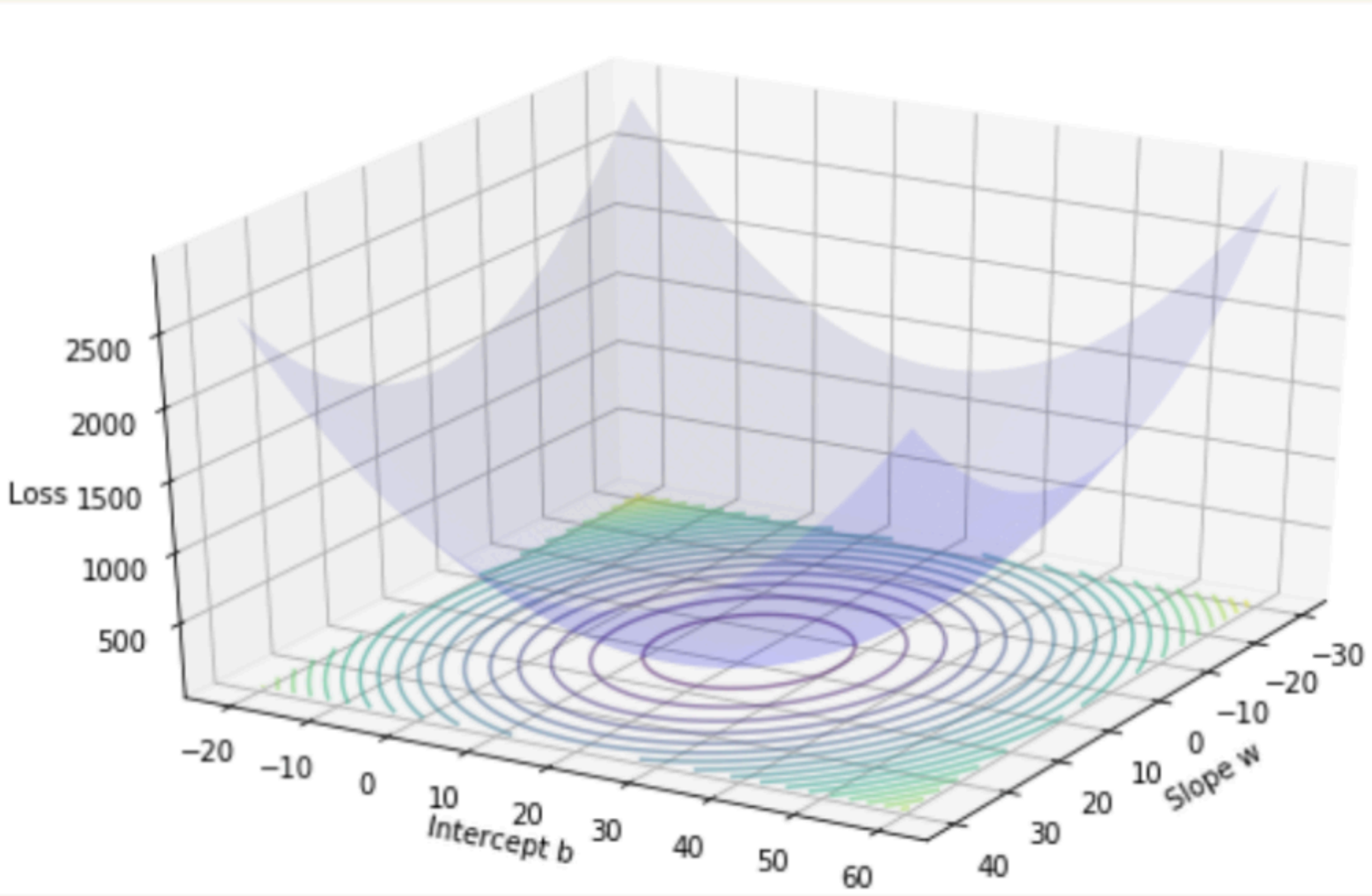
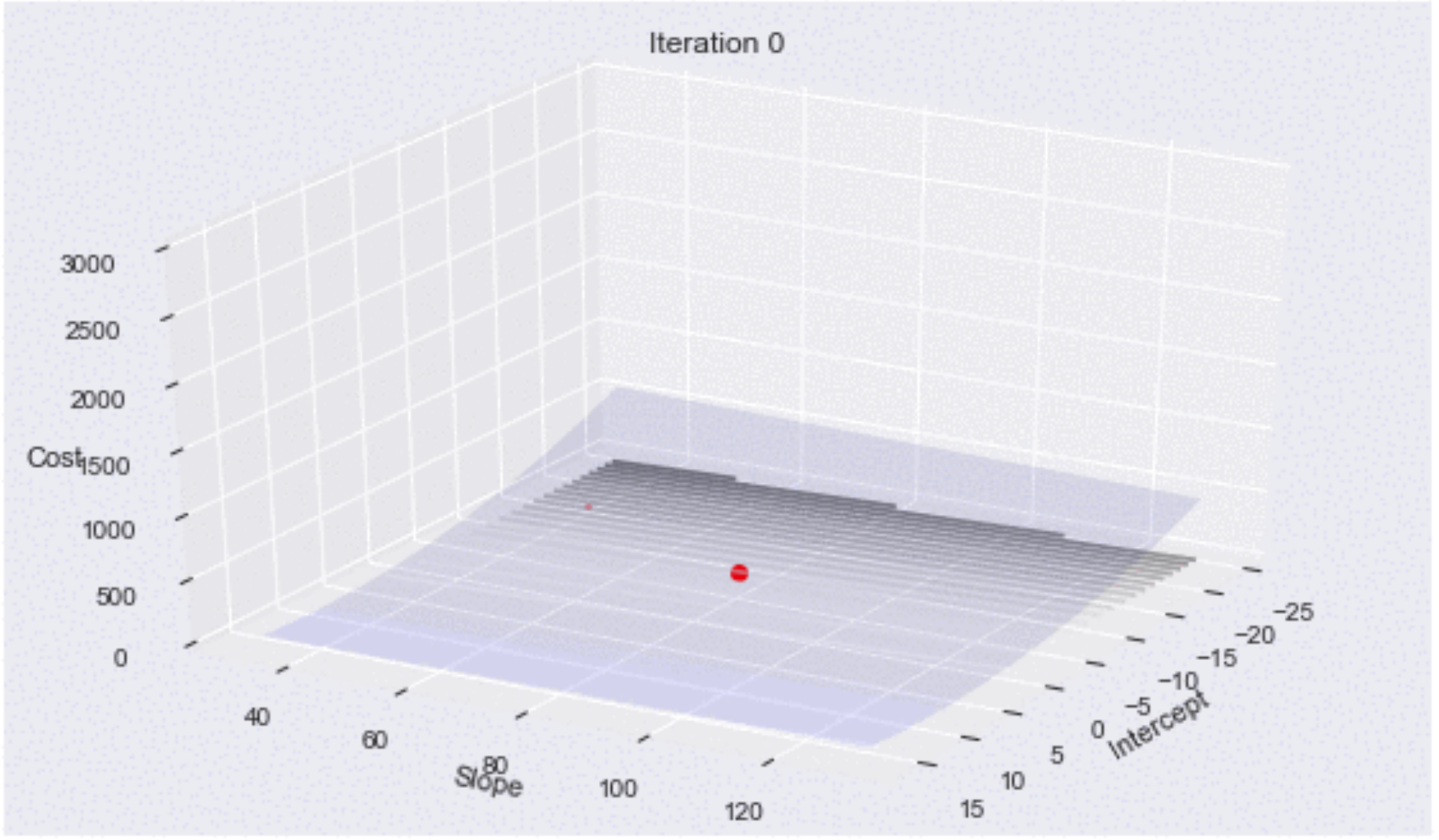
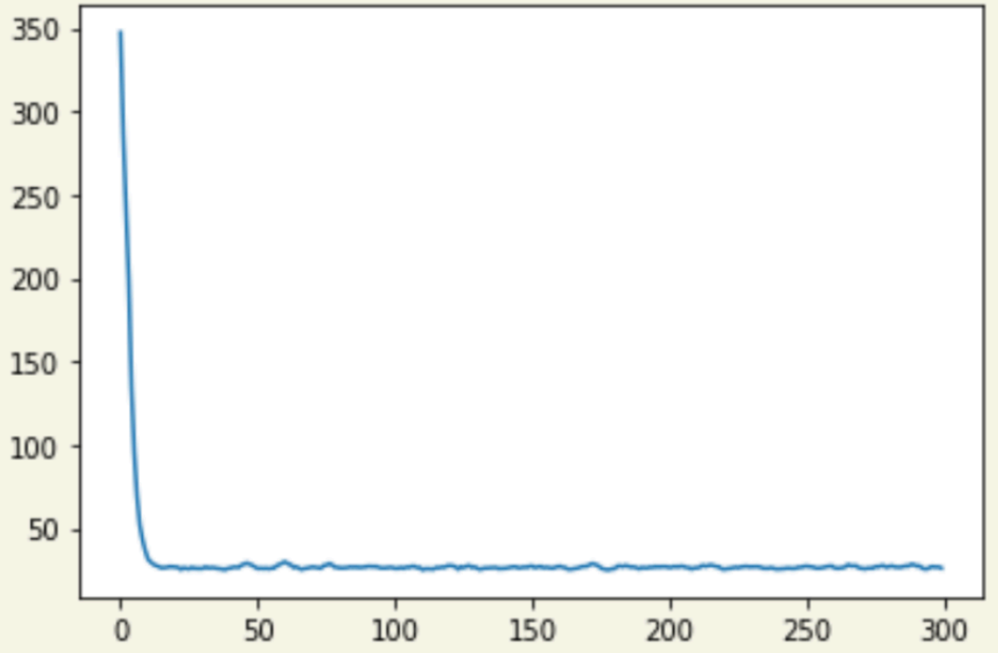
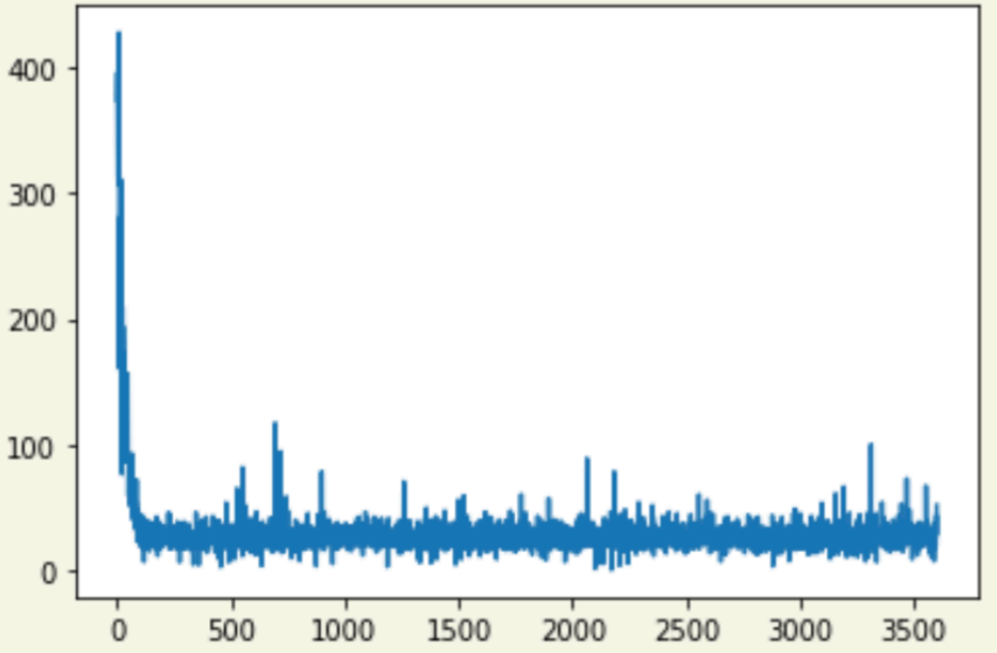


Illustration with linear regression


Iteration 0
Cost
3000
2500
2000
1500
1000
500
0
40
60
80
100
120
Slope
15
10
5
0
−5
−10
−15
−20
−25
Intercept

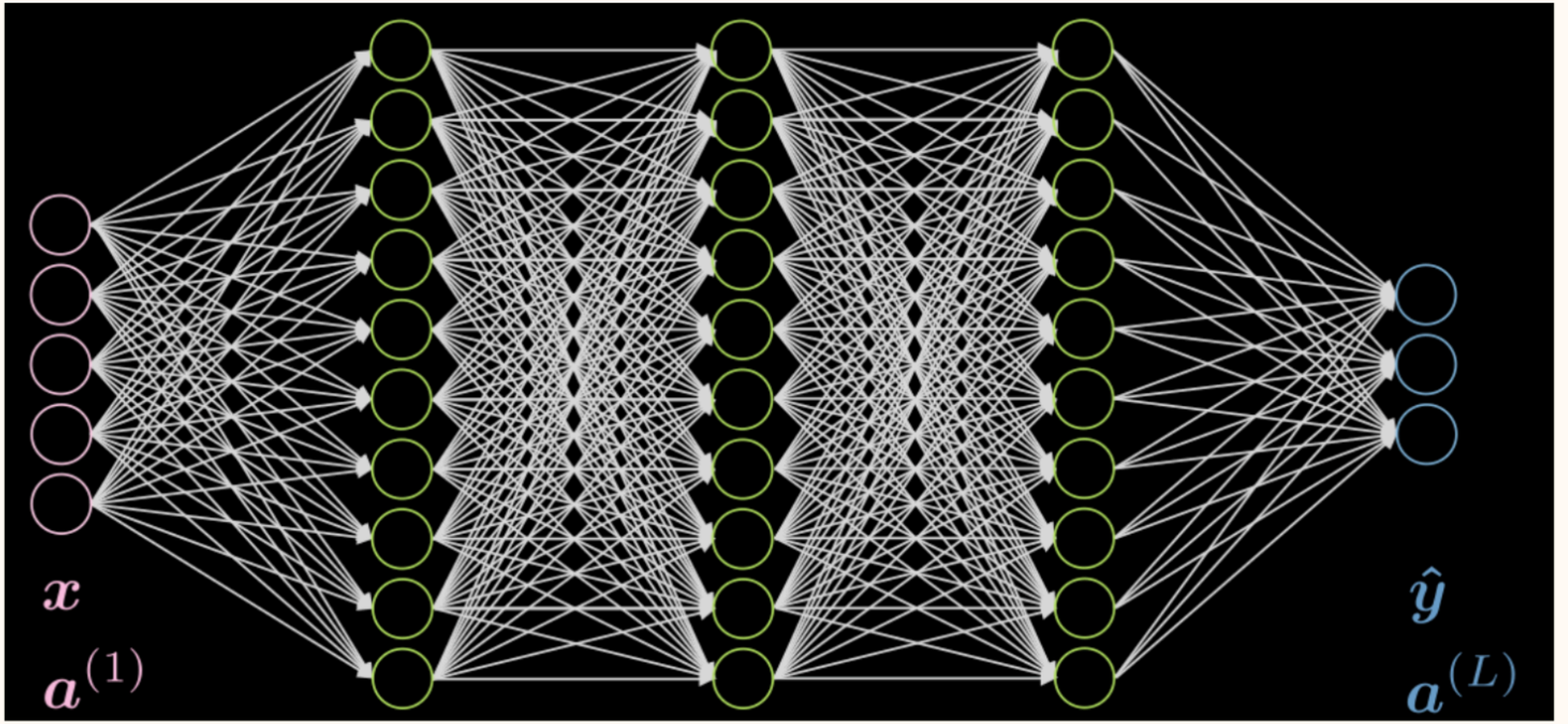


Epoch losses



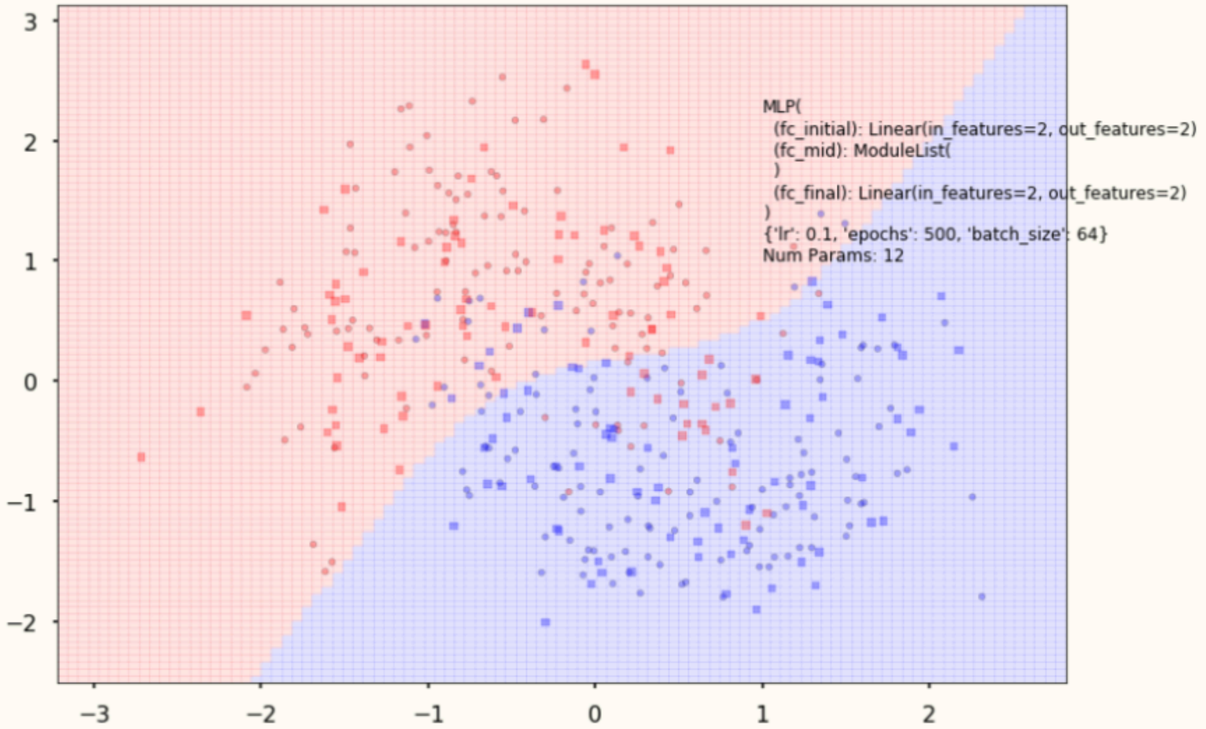
Batch losses

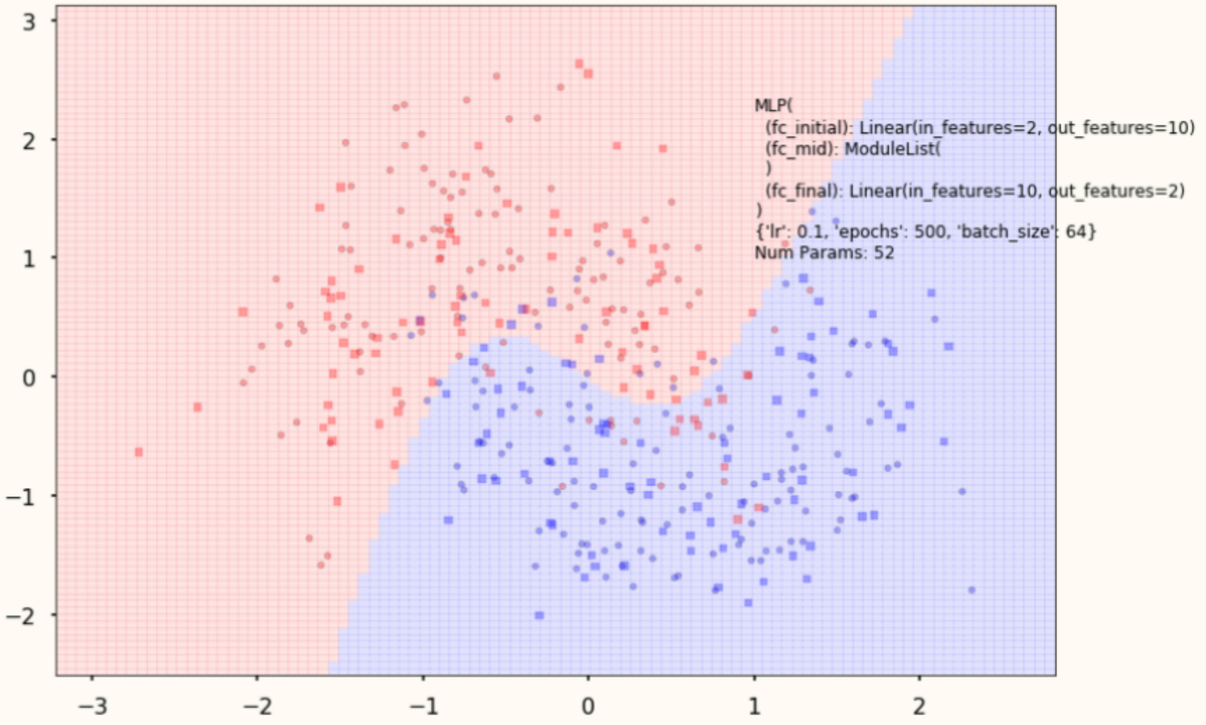
Classification with a neural network



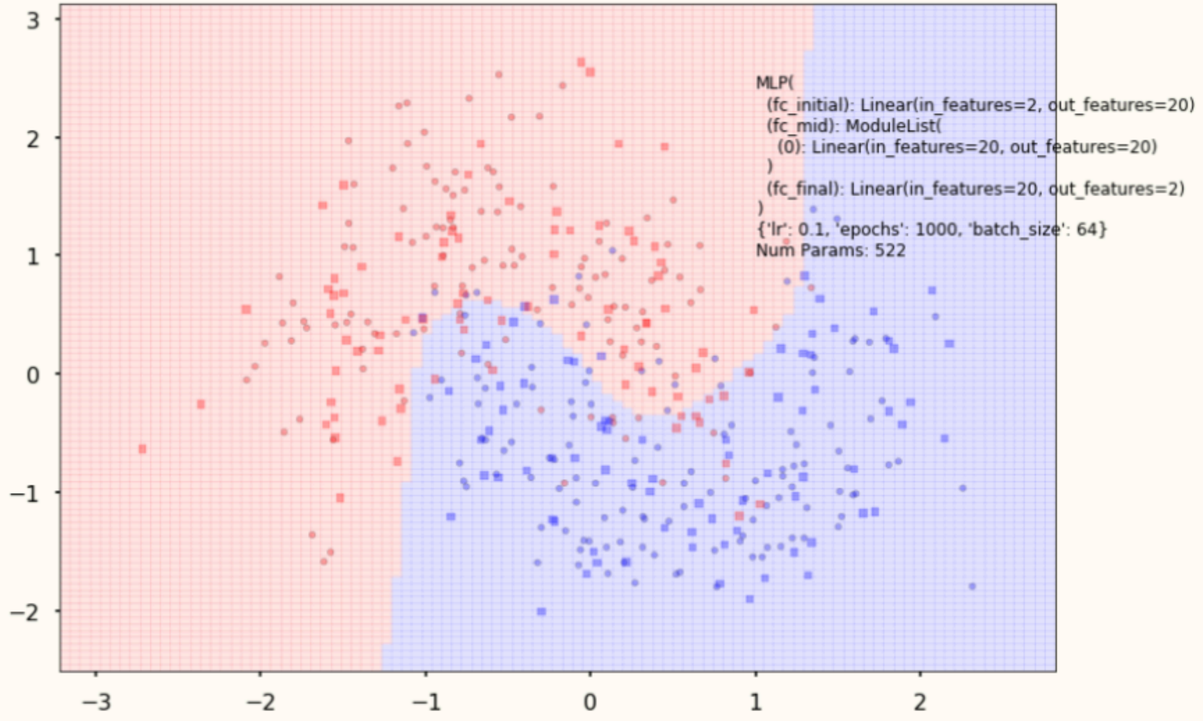
The structure of a neural network. Input nodes (pink) are connected to multiple layers of the network, and finally an output is produced to feed to a loss function. Each neuron rotates, shears, and translates its input, and then makes a non-linear transform on it like setting all negative numbers to 0. Image by Alfredo Canziani.

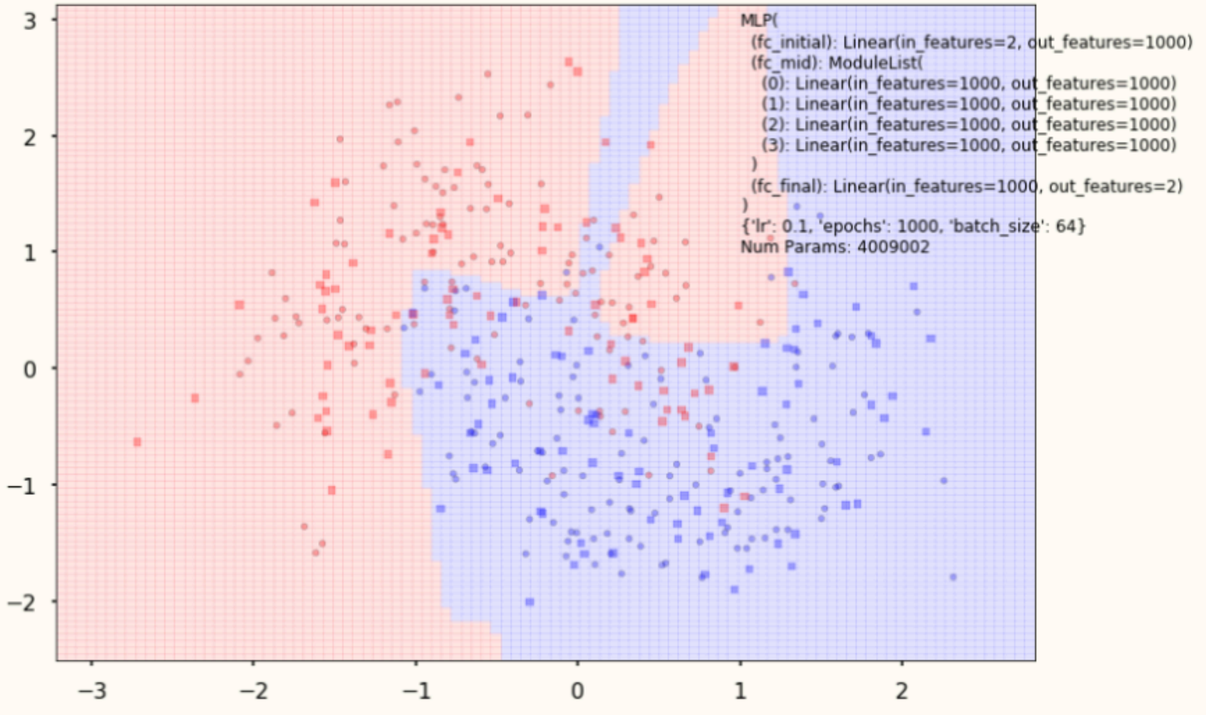
1 layer, 2 vs 10 neurons





2 layers, 20 neurons vs 5 layers, 1000 neurons





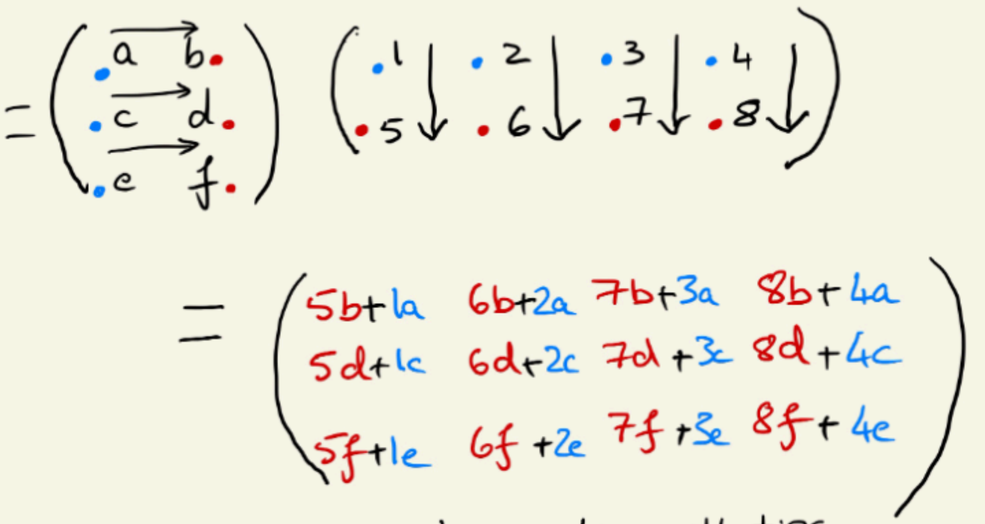
The geometry of neural networks

Matrix multiplication Recap

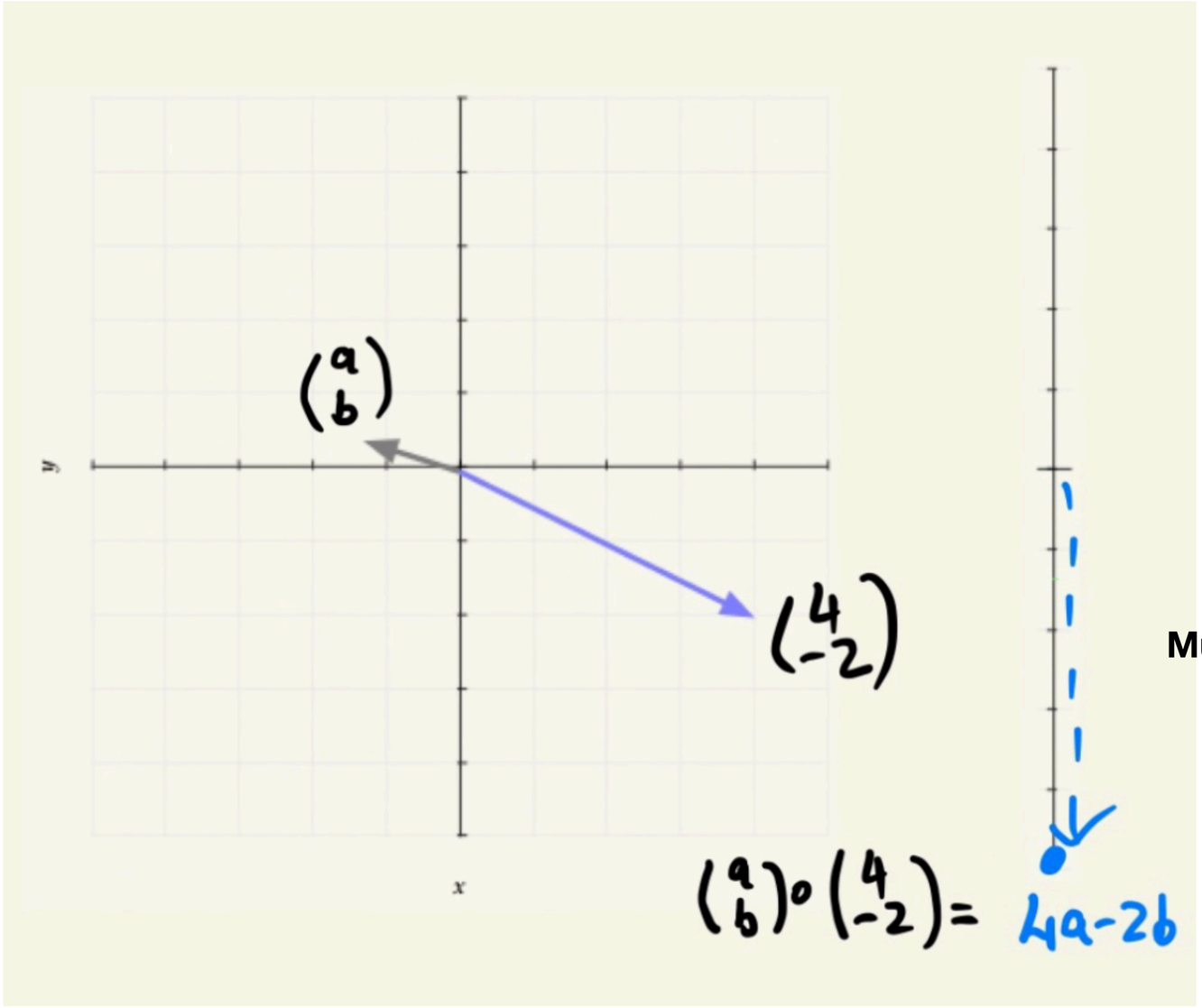
A and B are compatible for multiplication since their *inner dimensions* match:

$$3 \times 2, 2 \times 4$$

$$C = \begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \end{pmatrix}.$$



The product matrix $C = A \cdot B$ has the *outer dimensions*, thus $dim(C) = 3 \times 4$



$$T(\overline{v}) = \begin{pmatrix} a \\ b \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = a\,x + b\,y$$

Dot products are a transform from 2D to 1D.

And can be thought of a very special kind of matrix multiplication

Multiple dot products can transform to other dimensions

Multiple Dimensions

We might have $T : \mathbb{R}^2 \to \mathbb{R}^2$.

$$T(\overline{v}) = T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \end{pmatrix}$$

$$T(\overline{v}) = \begin{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \\ \begin{pmatrix} c \\ d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \end{pmatrix}.$$

or we might have $T : \mathbb{R}^2 \to \mathbb{R}^3$.

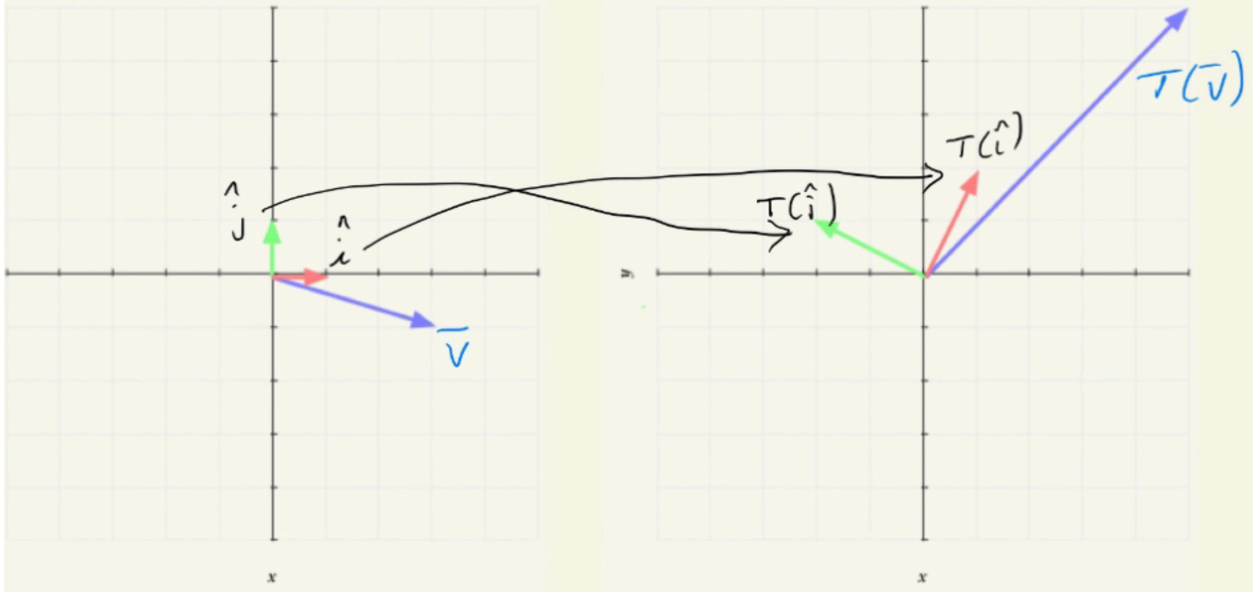
$$T(\overline{v}) = T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \\ ex + dy \end{pmatrix}$$

$$T(\overline{v}) = \begin{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \\ \begin{pmatrix} c \\ d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \\ \begin{pmatrix} e \\ f \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \end{pmatrix}.$$

Linear Transform as matrix multiplication

$$f(\overline{v}) = \begin{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \\ \begin{pmatrix} c \\ d \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \\ \begin{pmatrix} e \\ f \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} \end{pmatrix} = \begin{pmatrix} a & c & e \\ b & d & f \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \\ e & f \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} ax + by \\ cx + dy \\ ex + dy \end{pmatrix}$$

And every such transform is basically represented by a matrix multiplication



Thus, when you **multiply a matrix and a vector**, the *columns of the matrix are the transformations that the unit basis vectors undergo*, i.e. **where the unit vectors land up in space**.

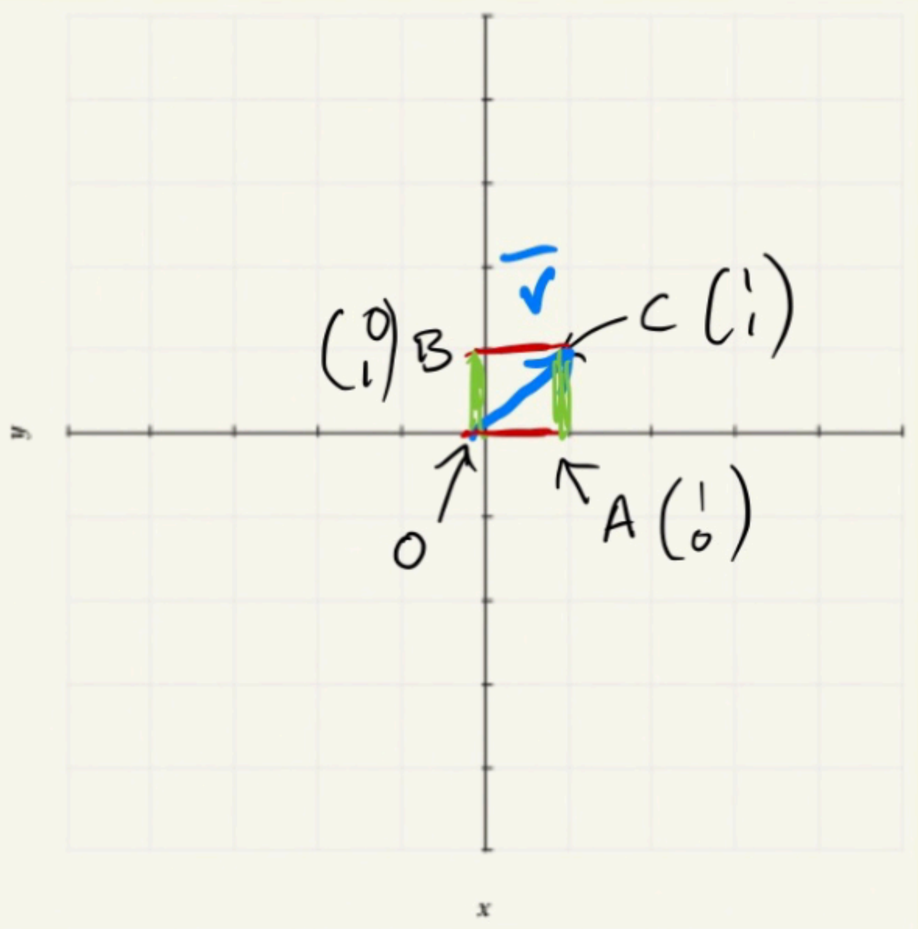
$$T\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} x \\ y \end{pmatrix} = x\begin{pmatrix} a \\ c \end{pmatrix} + y\begin{pmatrix} b \\ d \end{pmatrix}$$

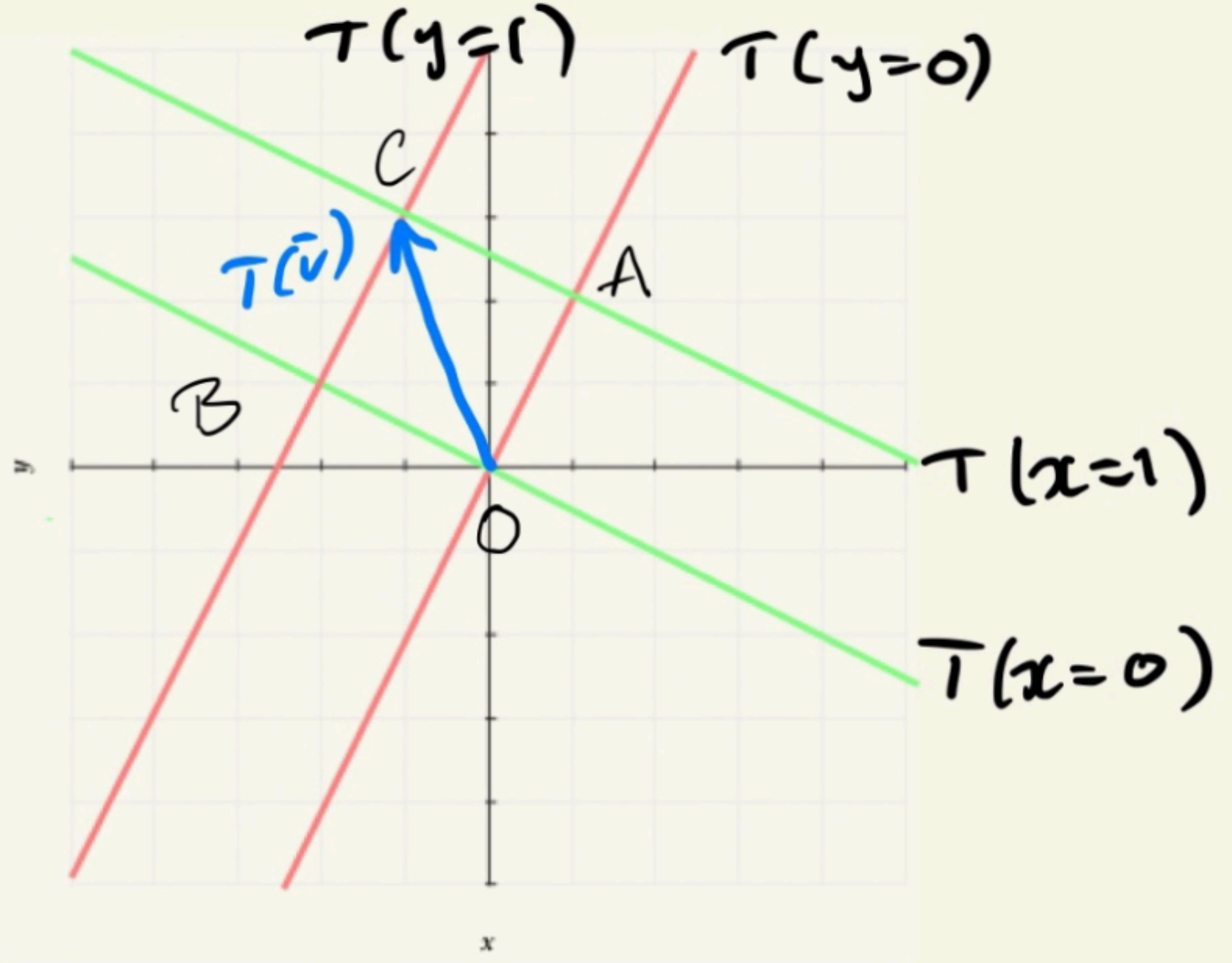
.

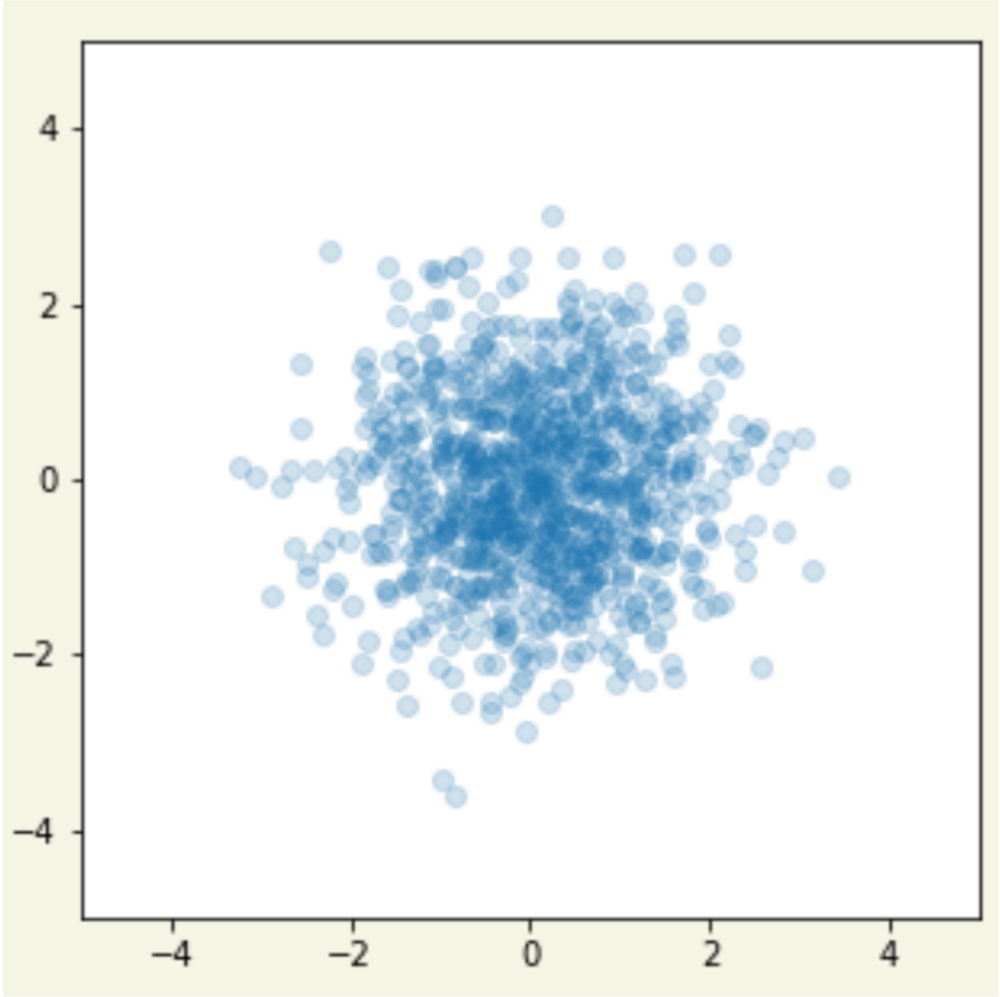
Scale the transformed unit vectors by the initial vector's co-ordinates and add. In our example:

$$T\begin{pmatrix} 3 \\ -1 \end{pmatrix} = \begin{pmatrix} 1 & -2 \\ 2 & 1 \end{pmatrix}\begin{pmatrix} 3 \\ -1 \end{pmatrix} = 3\begin{pmatrix} 1 \\ 2 \end{pmatrix} - 1\begin{pmatrix} -2 \\ 1 \end{pmatrix} = \begin{pmatrix} 5 \\ 5 \end{pmatrix}$$

.


y
x
$\vec{v}$
$\begin{pmatrix}0\\1\end{pmatrix}$ B
C $\begin{pmatrix}1\\1\end{pmatrix}$
O
A $\begin{pmatrix}1\\0\end{pmatrix}$


T(y=1)
T(y=0)
C
T($\vec{v}$)
A
B
y
x
O
T(x=1)
T(x=0)

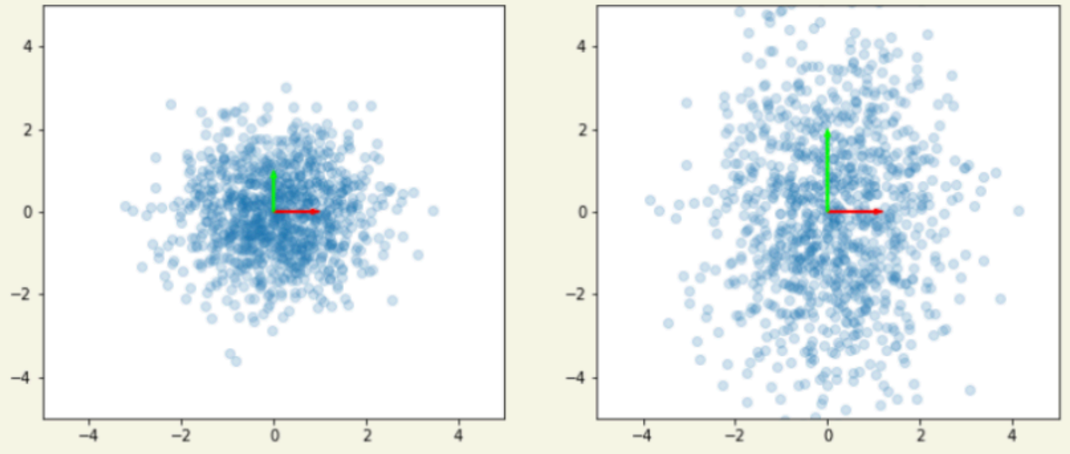


Ok, so what does matrix transformation do to a blob?

Scaling

$$\begin{pmatrix} 1.2 & 0 \\ 0 & 2 \end{pmatrix}$$

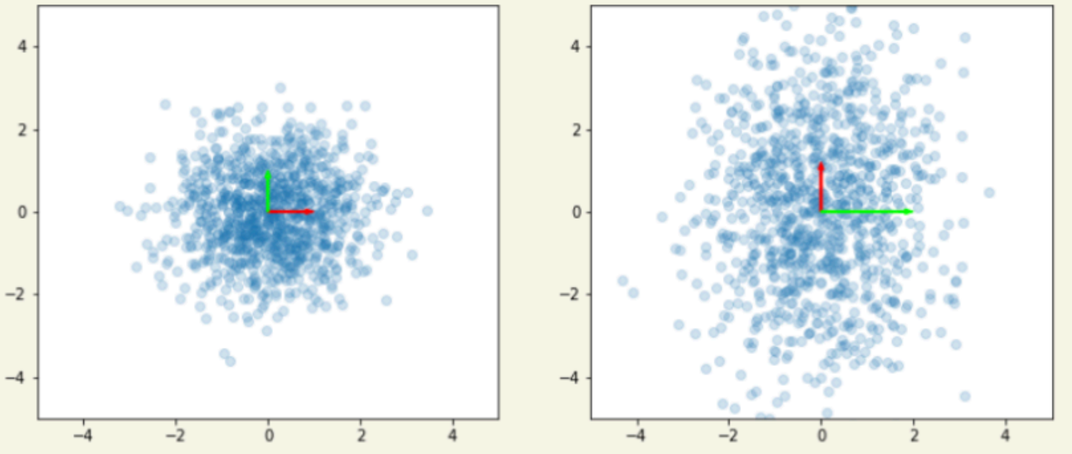
Stretches the unit vectors



Off-diagonal

$$\begin{pmatrix} 0 & 1.2 \\ 2 & 0 \end{pmatrix}$$

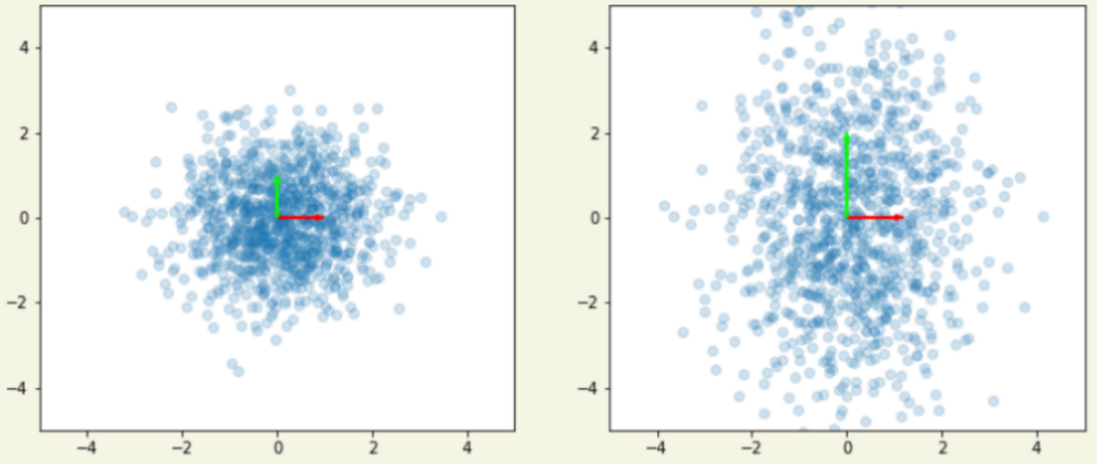
Flips and Stretches the unit vectors



Pure Rotation

$$\begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}$$

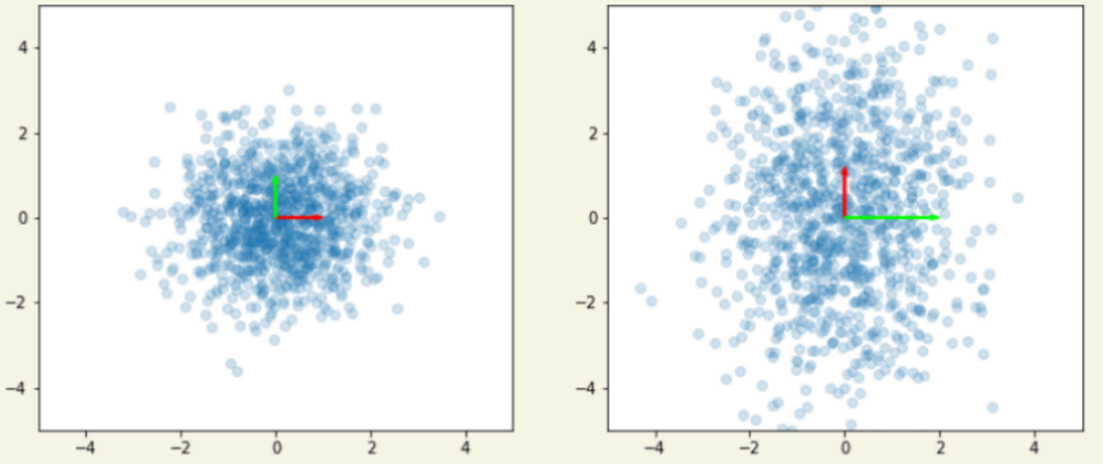
Simple rotation with no scaling



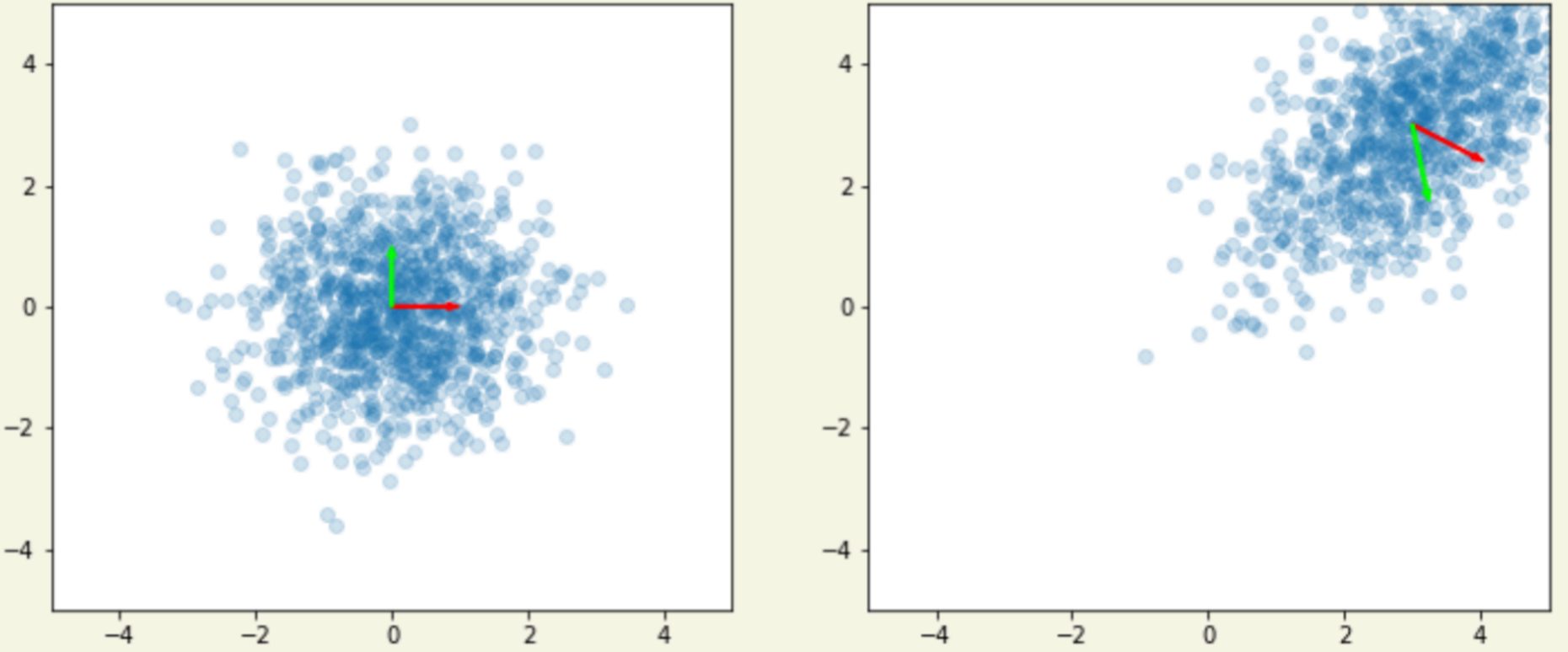
Random

$$\begin{pmatrix} 1.05686081 & -0.60719423 \\ 0.24501701 & -1.26410851 \end{pmatrix}$$

Stretching, rotation, and shear.

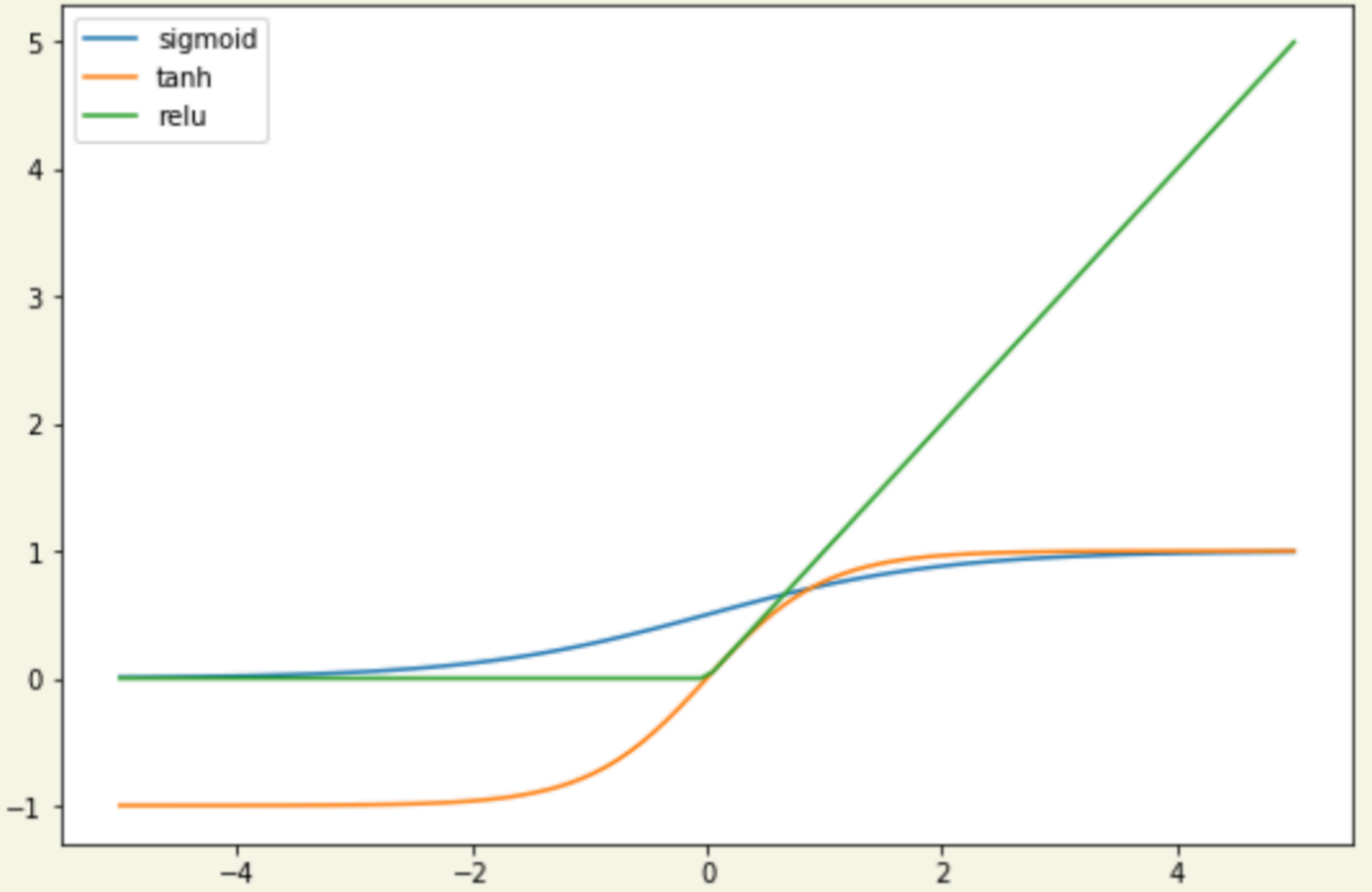


Add Bias



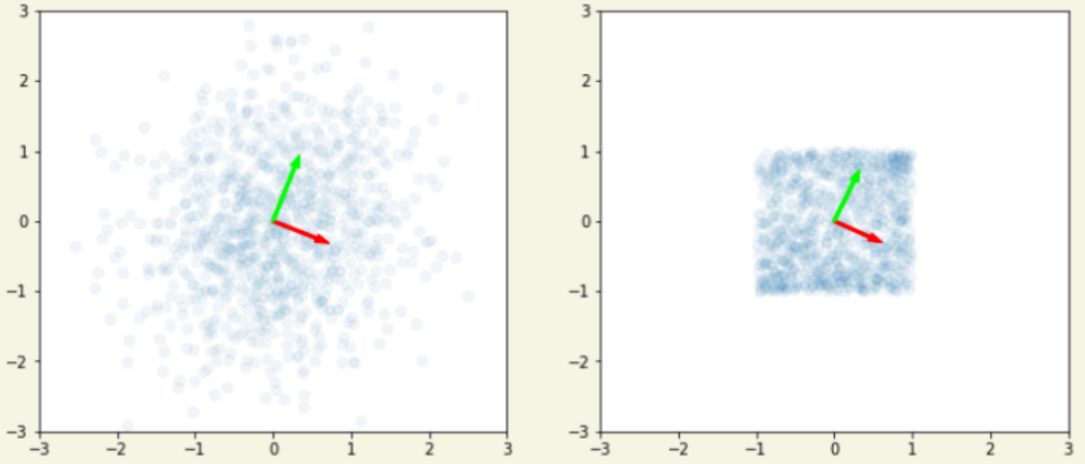
Bias moves the co-ordinate system over (or the data, depending on your point of view).Its impact is to make points more positive or negative.

Non-Linearities



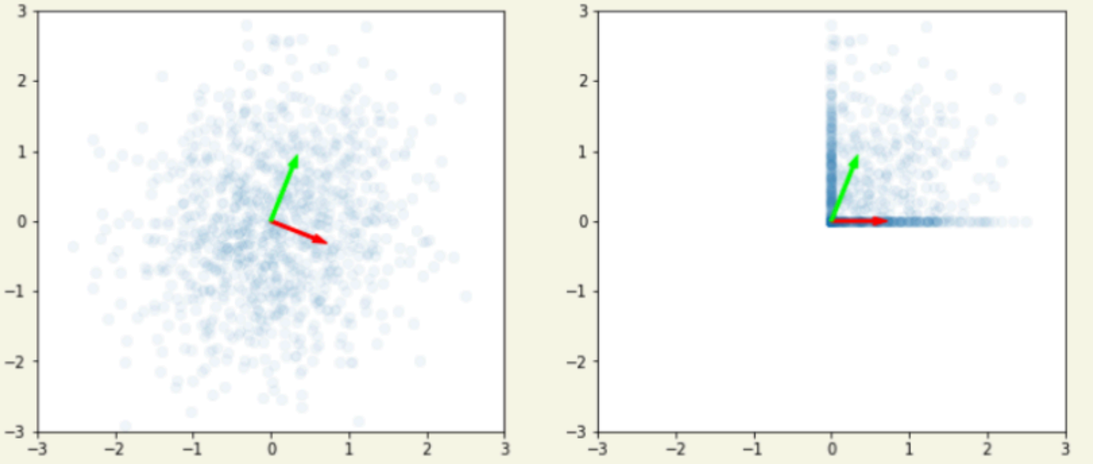
tanh

Squeezes data into [-1,1].



relu

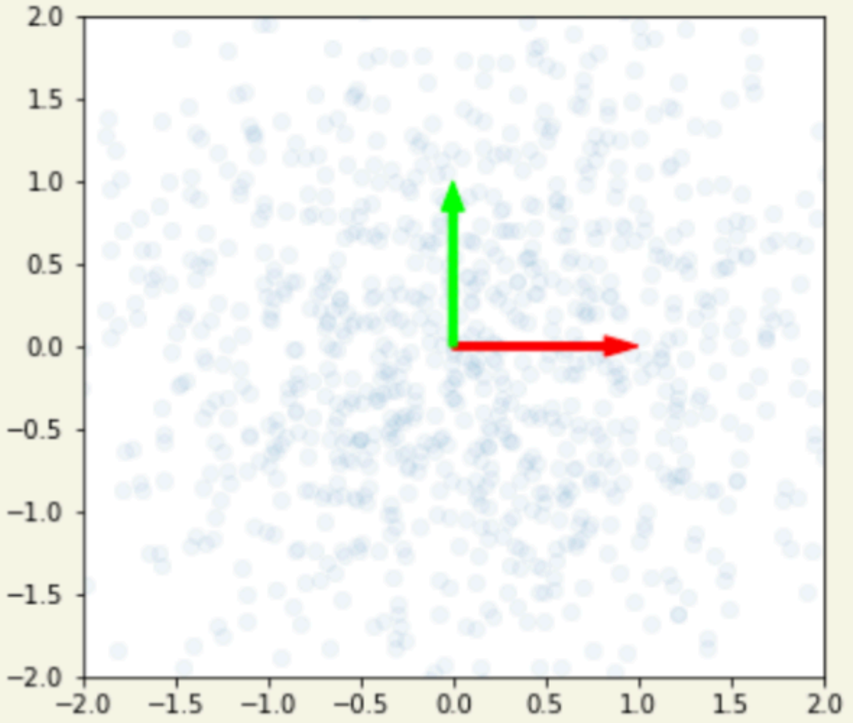
makes everything negative 0

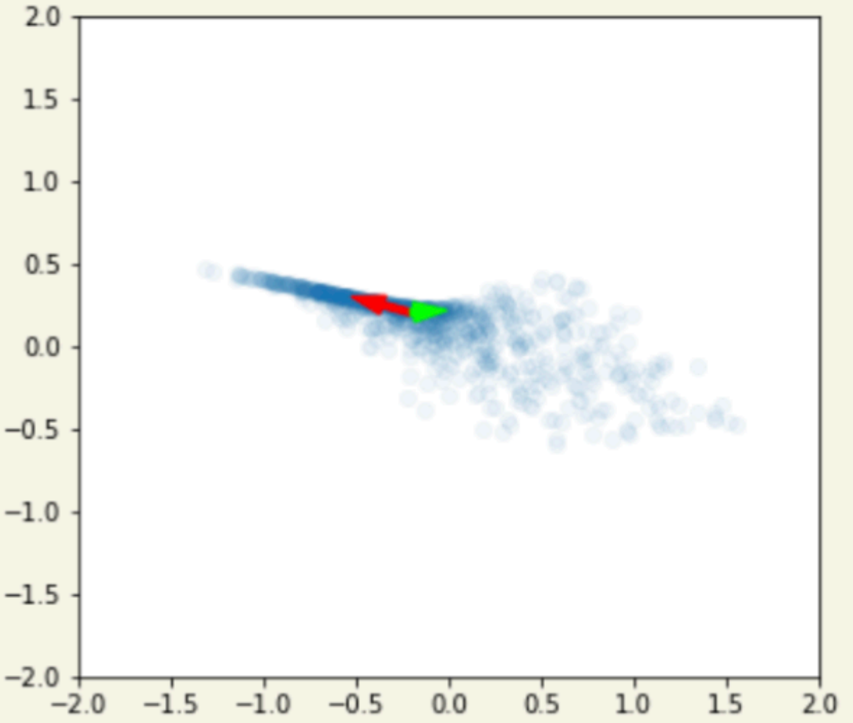


Neural Net (random bias)

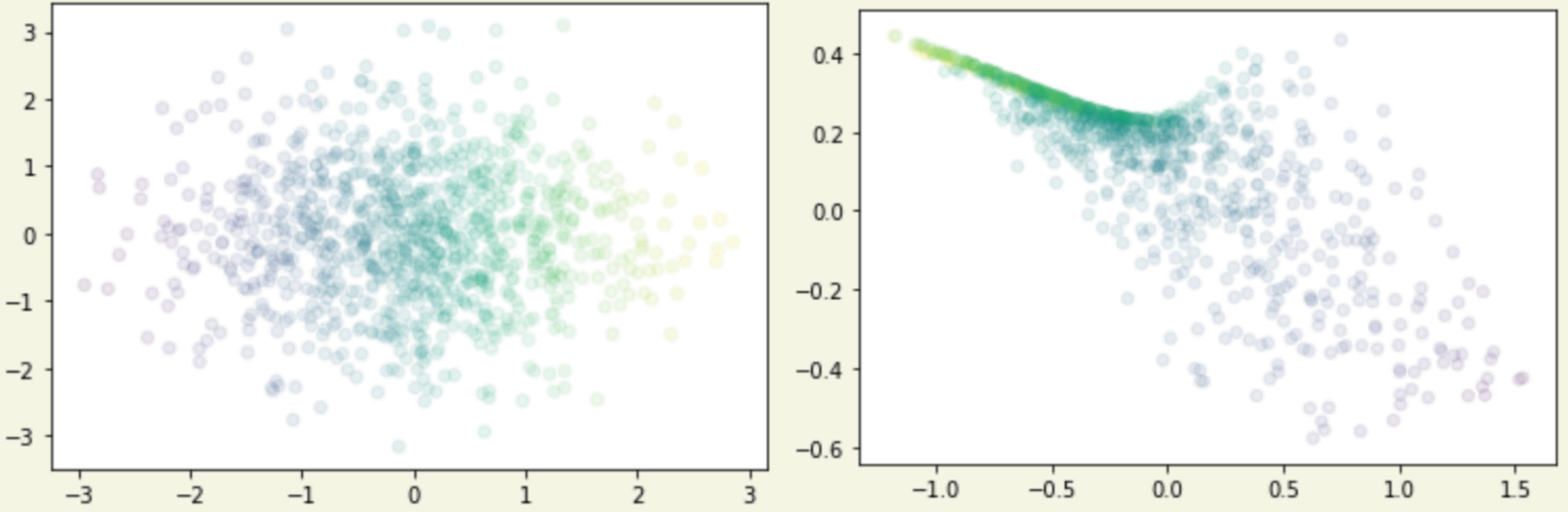
2-D input. Hidden (5 neuron) and Embedding (2D for viz) layers with tanh non-linearity in-between.

```
model_complex = Model([Affine("first", 2, 5), Tanh("tanh"), Affine("embedding", 5, 2)])
```

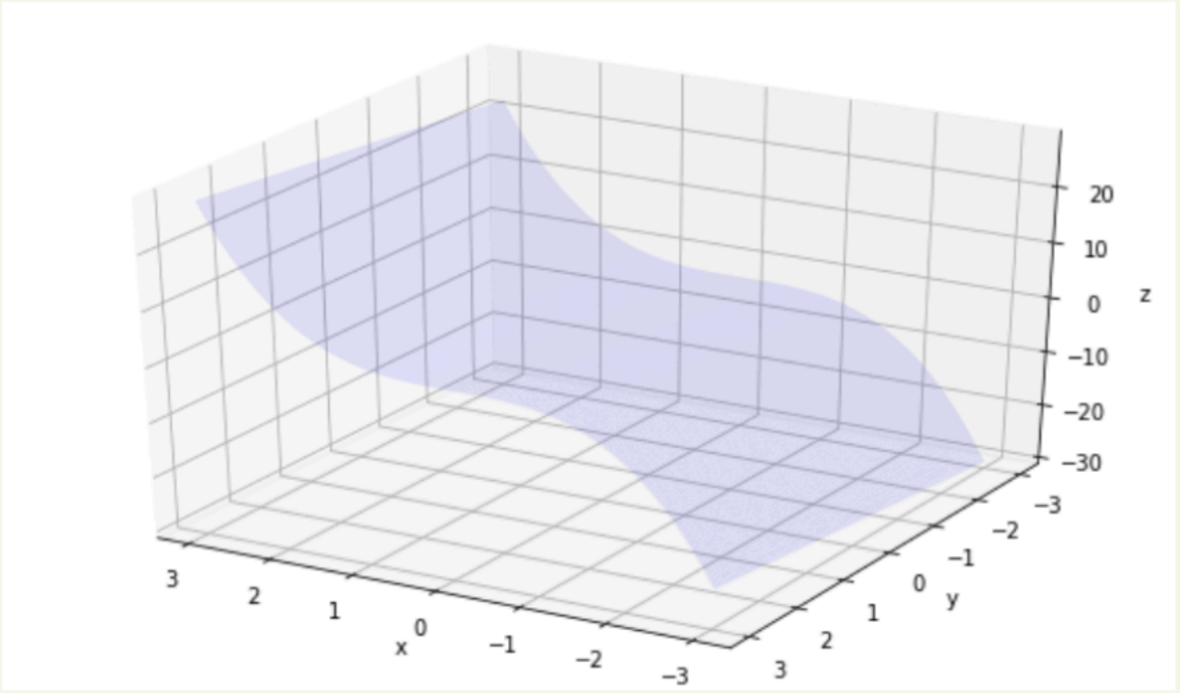


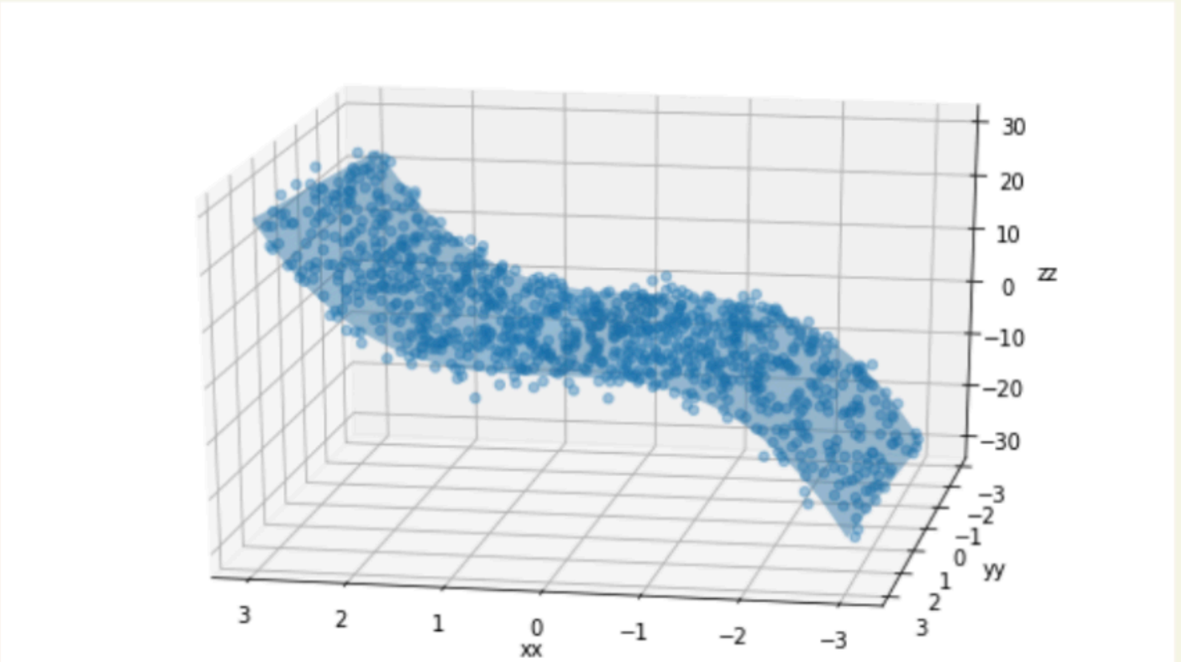


Neural Network with colors



Space is stretched, squeezed, rotated, sheared. Some points come close, some go far. The hope is that it leads to separation of classes (colors).





Fit a neural network

We fit the following network

```
first = [Affine("first_affine", 2, 128), Tanh("first")]
second = [Affine("second_affine", 128, 64), Tanh("second")]
third = [Affine("third_affine", 64, 64), Relu("third")]
embedding = [Affine("embedding", 64, 2)]
final = [Affine("final", 2, 1)]
```

We have built-in an embedding at the end. This is to see what his happening to the geometry. The model has 12933 parameters!

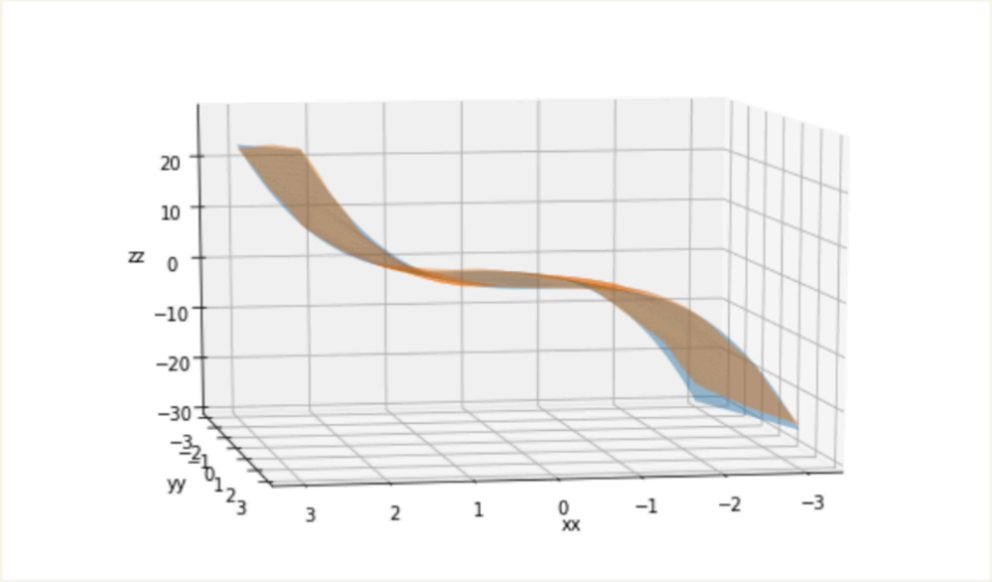
Fit a neural network

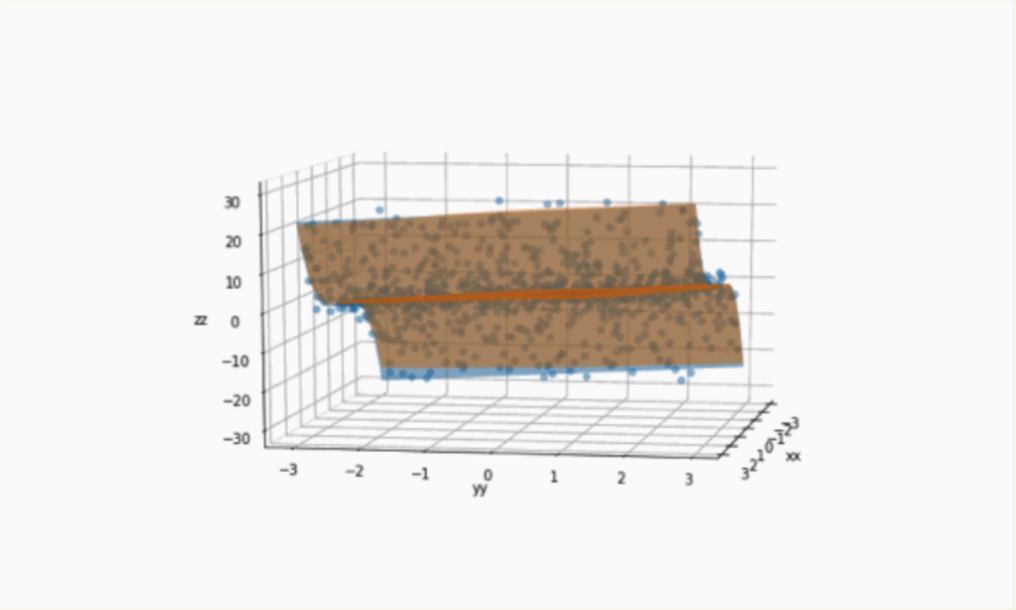
We fit the following network

```
first = [Affine("first_affine", 2, 128), Tanh("first")]
second = [Affine("second_affine", 128, 64), Tanh("second")]
third = [Affine("third_affine", 64, 64), Relu("third")]
embedding = [Affine("embedding", 64, 2)]
final = [Affine("final", 2, 1)]
```

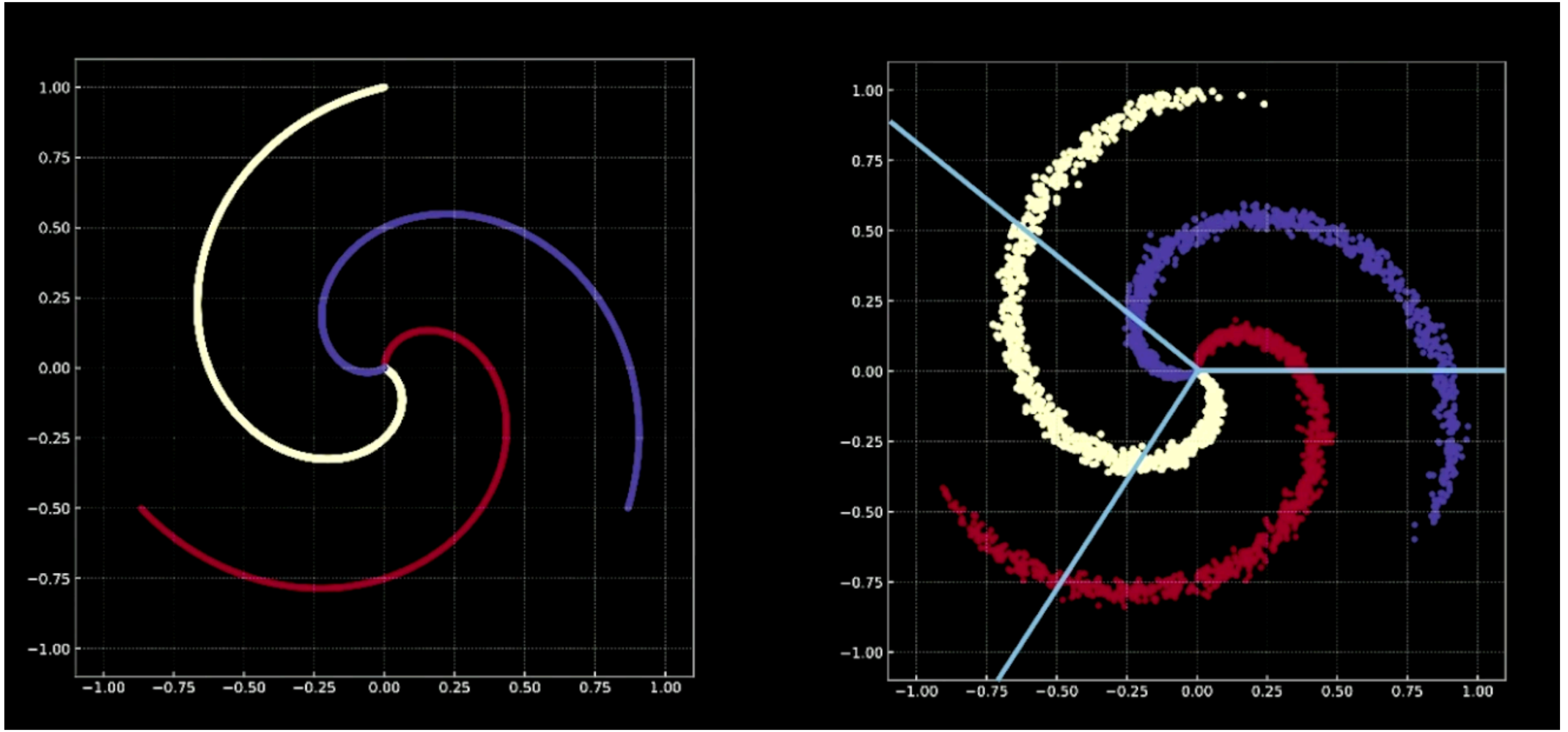
We have built-in an embedding at the end. This is to see what his happening to the geometry. The model has 12933 parameters!

Fit





The orange curve is a reasonable fit to the blue **generating** manifold.


1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00
1.00
0.75
0.50
0.25
0.00
−0.25
−0.50
−0.75
−1.00
−1.00 −0.75 −0.50 −0.25 0.00 0.25 0.50 0.75 1.00

The geometry of classification