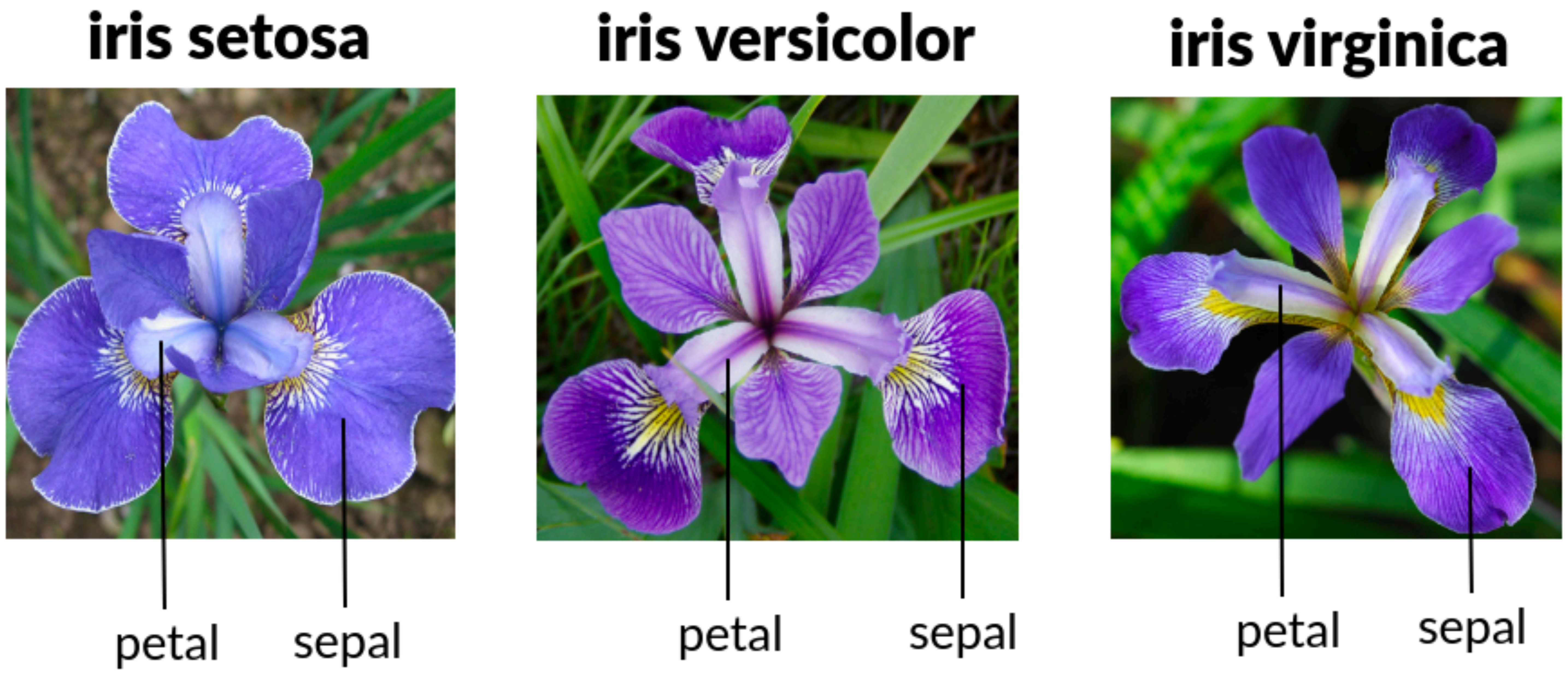
A quick intro to deep learning and PyTorch

A multiclass classification problem

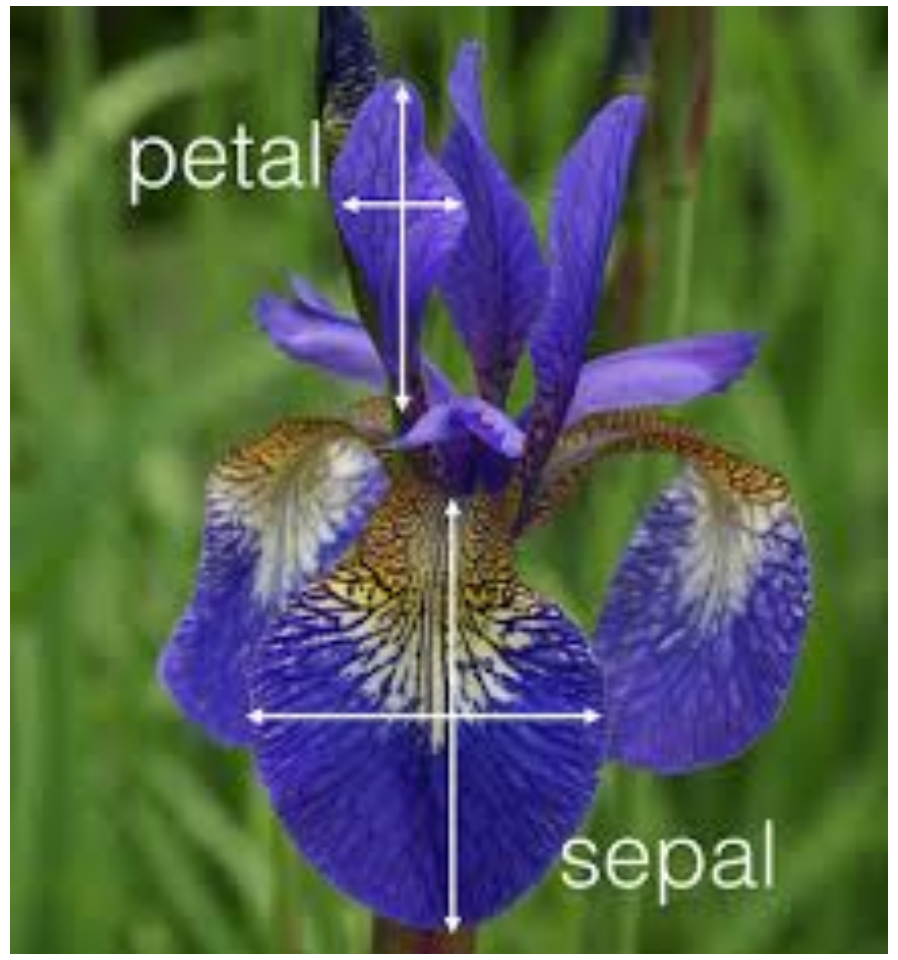
Three species of Iris:



A multiclass classification problem

Our **goal** is to predict the type of Iris based on four measurements:

- petal length
- petal width
- sepal length
- sepal width



These four numbers are called "features", and combined they form a "feature vector" such as $\begin{bmatrix} 5.1 \\ 3.5 \\ 1.4 \\ 0.2 \end{bmatrix}$

Each Iris is described by its own feature vector

Note: A "vector" is just a finite, ordered list of numbers

Definition of a probability vector

A vector such as

$$p = \begin{bmatrix} .3 \\ .1 \\ .6 \end{bmatrix}$$

whose components are nonnegative and sum to 1 is called a "probability vector"

Application:

Suppose we're solving a classification problem with 3 possible classes

The probability vector p tells us how likely it is that the example we're looking at belongs to each class

So the vector p above tells us:

- The probability of belonging to class 1 is 30%
- The probability of belonging to class 2 is 10%
- The probability of belonging to class 3 is 60%

Probability vectors that express certainty

The special probability vectors $\begin{bmatrix}1\\0\\0\end{bmatrix}$, $\begin{bmatrix}0\\1\\0\end{bmatrix}$, and $\begin{bmatrix}0\\0\\1\end{bmatrix}$ reflect certainty about which class an example belongs to

So, the vector $\begin{bmatrix}1\\0\\0\end{bmatrix}$ expresses certainty that the example we're looking at belongs to class 1

Likewise, the vector $\begin{bmatrix}0\\1\\0\end{bmatrix}$ expresses certainty that the example belongs to class 2

And the vector $\begin{bmatrix}0\\0\\1\end{bmatrix}$ expresses certainty that the example belongs to class 3

The softmax function

The function $S : \mathbb{R}^K \to \mathbb{R}^K$ defined by

$$S(u) = \begin{bmatrix} \frac{e^{u_1}}{e^{u_1}+\cdots+e^{u_K}} \\ \frac{e^{u_2}}{e^{u_1}+\cdots+e^{u_K}} \\ \vdots \\ \frac{e^{u_K}}{e^{u_1}+\cdots+e^{u_K}} \end{bmatrix}$$

$$\begin{bmatrix} u_1 \\ u_2 \\ \vdots \\ u_K \end{bmatrix}$$

is called the "softmax" function

The output of S is guaranteed to be a probability vector!

The softmax function is useful in machine learning because it converts a vector into a probability vector

A recipe for a multiclass classification algorithm

Ingredient 1: A training dataset

Our training dataset consists of a collection of feature vectors

$$x_1, x_2, \ldots, x_N \in \mathbb{R}^d$$

$$\begin{bmatrix}5.1\\3.5\\1.4\\0.2\end{bmatrix} \quad \begin{bmatrix}7\\3.2\\4.7\\1.4\end{bmatrix} \quad \begin{bmatrix}5.9\\3\\5.1\\1.8\end{bmatrix}$$

and corresponding target values

$$y_1, y_2, \ldots, y_N \in \mathbb{R}^K$$

$$\begin{bmatrix}0\\1\\0\end{bmatrix} \quad \begin{bmatrix}1\\0\\0\end{bmatrix} \quad \begin{bmatrix}0\\0\\1\end{bmatrix}$$

Here y_i is a probability vector that expresses certainty about which class example i belongs to

For example, if $K = 3$ and example i belongs to class 1, then $y_i = \begin{bmatrix}1\\0\\0\end{bmatrix}$

The position of the 1 tells you which class the example belongs to

A recipe for a multiclass classification algorithm

Ingredient 2: A prediction function $f : \mathbb{R}^d \to \mathbb{R}^K$

The output of f should be a probability vector, and we hope that

$$f(x_i) \approx y_i \quad \text{for } i = 1, \ldots, N$$

This looks complicated, but you could try to invent a more concise notation

Big question: What form should we assume for f?

For example, we might assume that f has the form

$$f(x_i) = S\left(\begin{bmatrix} \beta_{1,0} + \beta_{1,1}x_{i,1} + \cdots + \beta_{1,d}x_{i,d} \\ \beta_{2,0} + \beta_{2,1}x_{i,1} + \cdots + \beta_{2,d}x_{i,d} \\ \vdots \\ \beta_{K,0} + \beta_{K,1}x_{i,1} + \cdots + \beta_{K,d}x_{i,d} \end{bmatrix}\right)$$

$$x_i = \begin{bmatrix} x_{i,1} \\ x_{i,2} \\ \vdots \\ x_{i,d} \end{bmatrix}$$

In words, this function f computes a bunch of weighted combinations of the components of x_i, then the softmax function S is applied to ensure that the output is a probability vector

Much of machine learning is just getting creative about what form we assume for f

A recipe for a multiclass classification algorithm

Ingredient 3: A loss function ℓ

We need a way to measure how well a predicted probability vector q agrees with a "ground truth" probability vector p

First idea: $$\ell(p, q) = (p_1 - q_1)^2 + (p_2 - q_2)^2 + \cdots + (p_K - q_K)^2$$

$$p = \begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_K \end{bmatrix} \qquad q = \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_K \end{bmatrix}$$

This choice of ℓ is called the "squared error" loss function

If q agrees perfectly with p, then $\ell(p, q) = 0$

On the other hand, if q is not close to p, then $\ell(p, q)$ is large

A recipe for a multiclass classification algorithm

The most beautiful way to measure how well a predicted probability vector q agrees with a "ground truth" probability vector p is to use the "cross-entropy" loss function ℓ defined by

$$\ell(p, q) = -p_1 \log(q_1) - p_2 \log(q_2) - \cdots - p_K \log(q_K)$$

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_K \end{bmatrix} \qquad \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_K \end{bmatrix}$$

This strange-looking formula is hard to motivate, but it turns out that in some sense it's the most natural way to compare probability vectors

Exercise: Suppose that $p = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$. Which probability vector q minimizes $\ell(p, q)$?

A recipe for a multiclass classification algorithm

The most beautiful way to measure how well a predicted probability vector q agrees with a "ground truth" probability vector p is to use the "cross-entropy" loss function ℓ defined by

$$\ell(p, q) = -p_1 \log(q_1) - p_2 \log(q_2) - \cdots - p_K \log(q_K)$$

$$\begin{bmatrix} p_1 \\ p_2 \\ \vdots \\ p_K \end{bmatrix} \quad \begin{bmatrix} q_1 \\ q_2 \\ \vdots \\ q_K \end{bmatrix}$$

This strange-looking formula is hard to motivate, but it turns out that in some sense it's the most natural way to compare probability vectors

Exercise: Suppose that $p = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$. Which probability vector q minimizes $\ell(p, q)$?

Conclusion: $\ell(p, q)$ is small when q agrees with p!

Discovering the cross-entropy loss function

The cross-entropy formula looks weird – how would you discover it?

One approach uses the "maximum likelihood estimation" technique from statistics

We make a modeling assumption that the probability vector

$$f(x_i) = S\left(\begin{bmatrix} \beta_{1,0} + \beta_{1,1}x_{i,1} + \cdots + \beta_{1,d}x_{i,d} \\ \beta_{2,0} + \beta_{2,1}x_{i,1} + \cdots + \beta_{2,d}x_{i,d} \\ \vdots \\ \beta_{K,0} + \beta_{K,1}x_{i,1} + \cdots + \beta_{K,d}x_{i,d} \end{bmatrix}\right)$$

tells us how likely it is that example i belongs to each of the K classes

Then we go through the steps of maximum likelihood estimation to estimate the beta coefficients

and when you work out the details, the cross-entropy formula emerges

Objective function

We hope that $\ell(y_i, f(x_i))$ is small for $i = 1, \ldots, N$

In other words, we hope that the average cross-entropy

$$L(\beta) = \frac{1}{N} \sum_{i=1}^{N} \ell(y_i, f(x_i))$$

is small

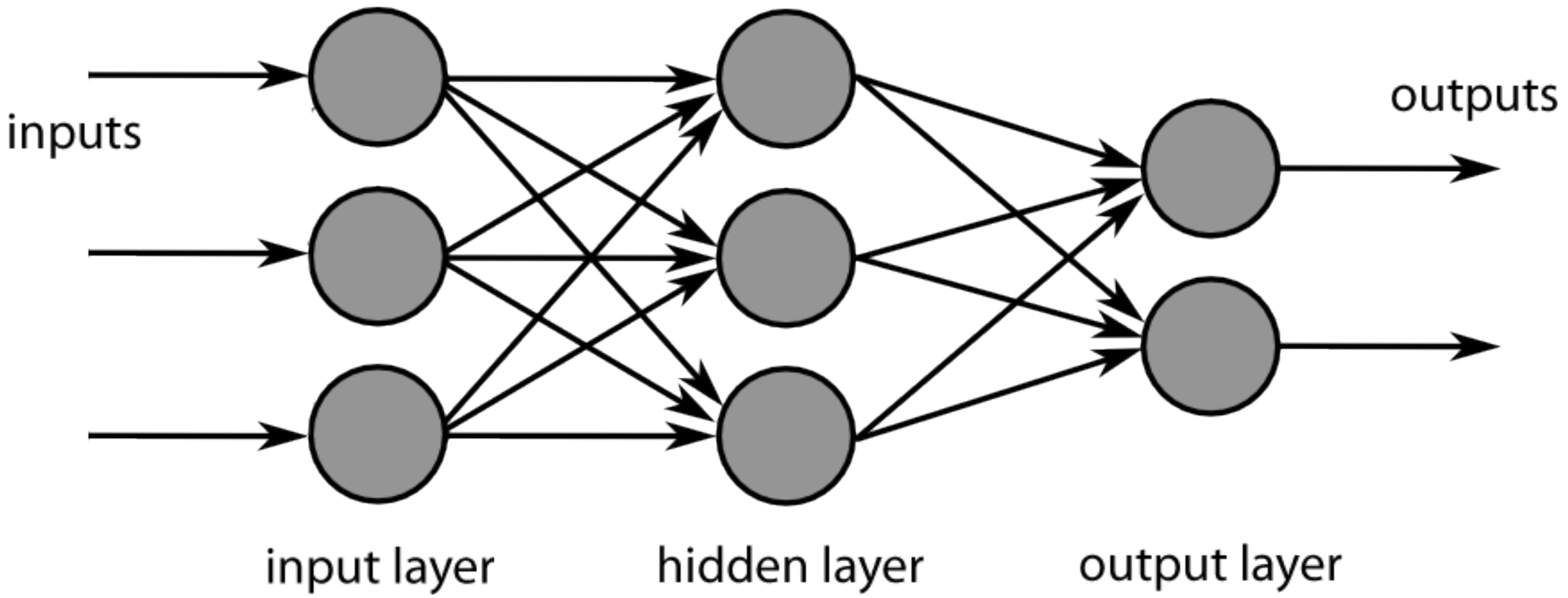
$$\begin{bmatrix} \beta_{10} \\ \vdots \\ \beta_{1d} \\ \vdots \\ \beta_{K0} \\ \vdots \\ \beta_{Kd} \end{bmatrix}$$

These parameters are "knobs" that you can tune

We could call this f_β

We select β by solving the optimization problem: $\underset{\beta}{\text{minimize}} \; L(\beta)$

Neural networks

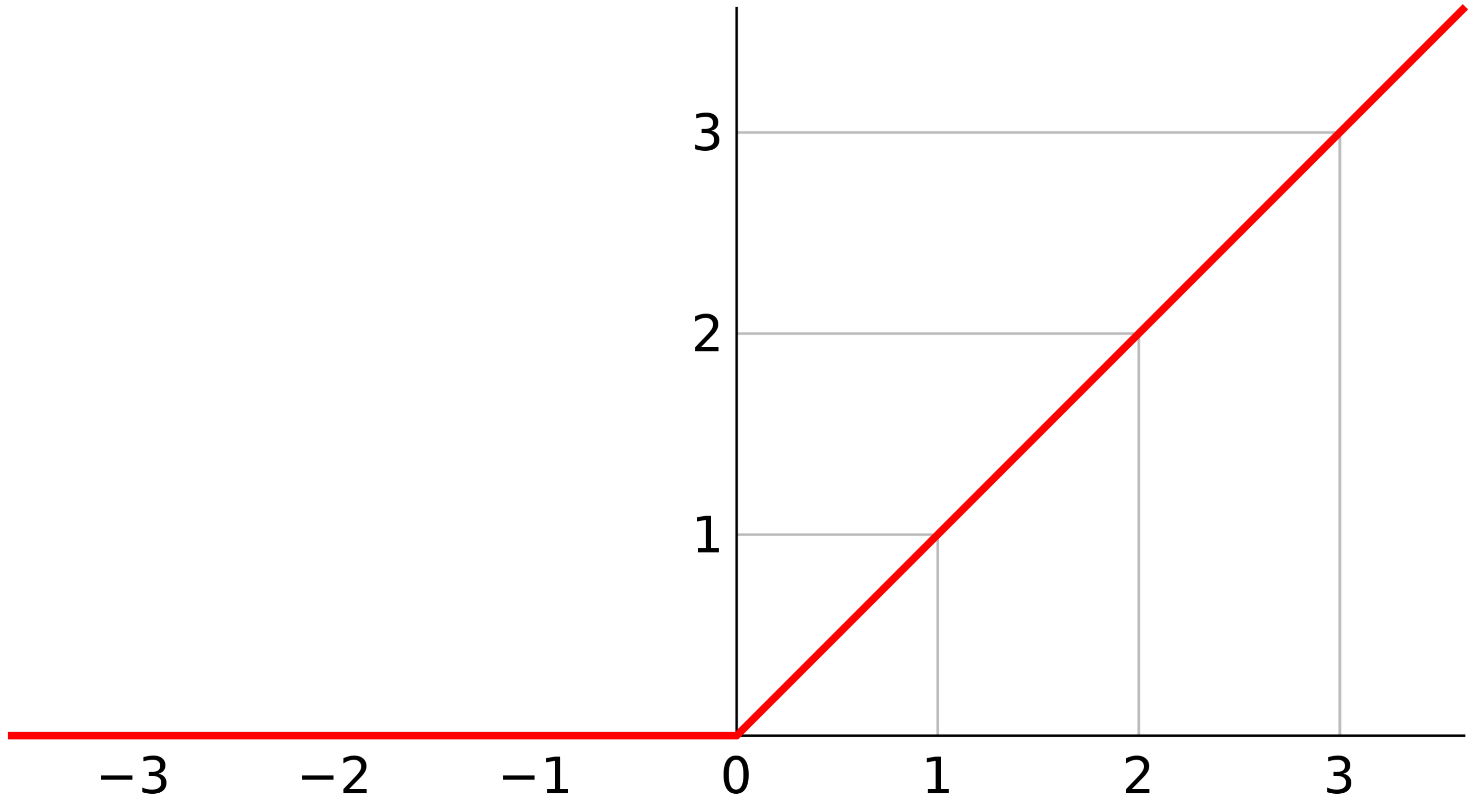


(Getting creative with the prediction function)

A simple nonlinear function

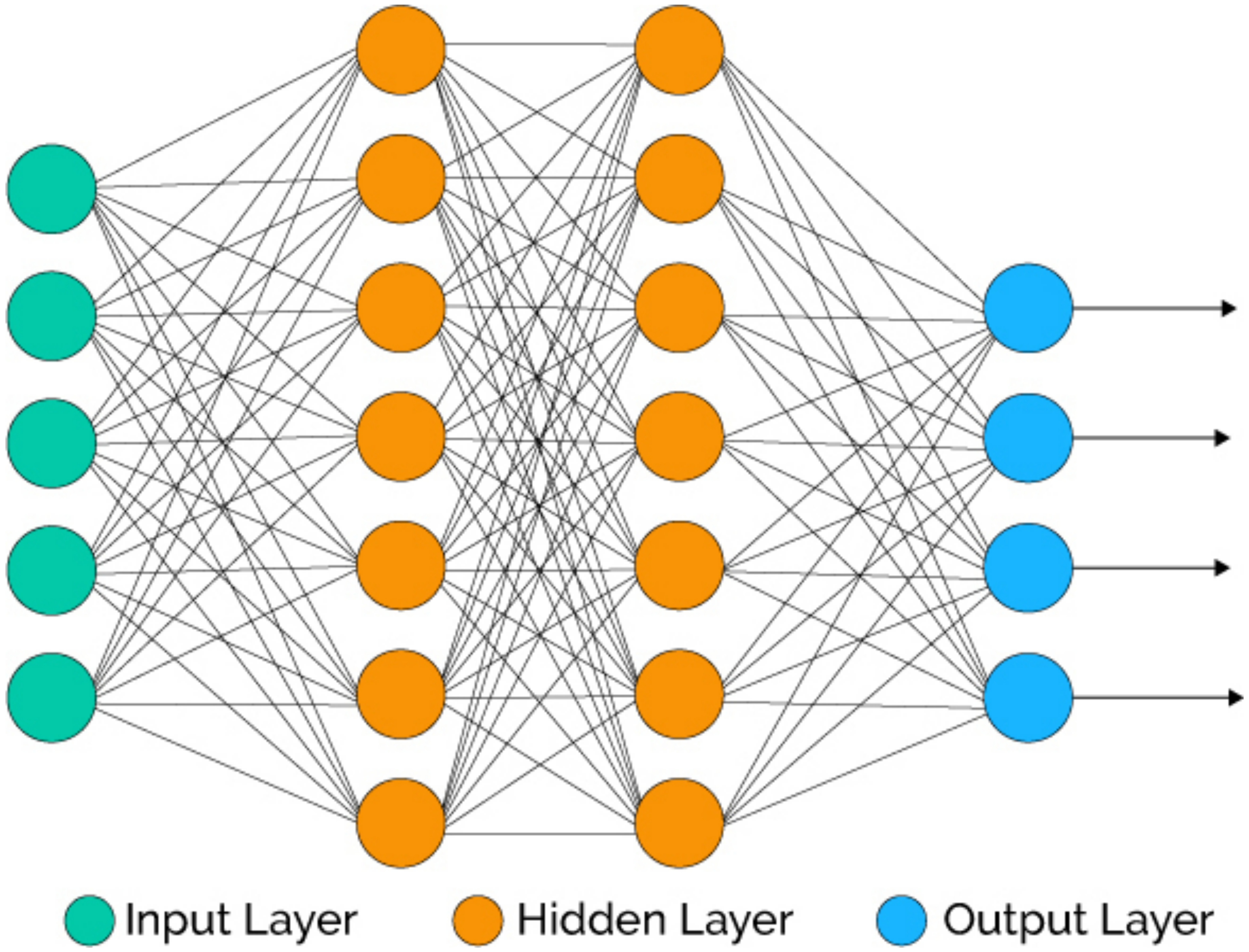
$$r(u) = \begin{cases} u & \text{if } u \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

Also called ReLU



ReLU is a popular choice of "activation function" in neural networks

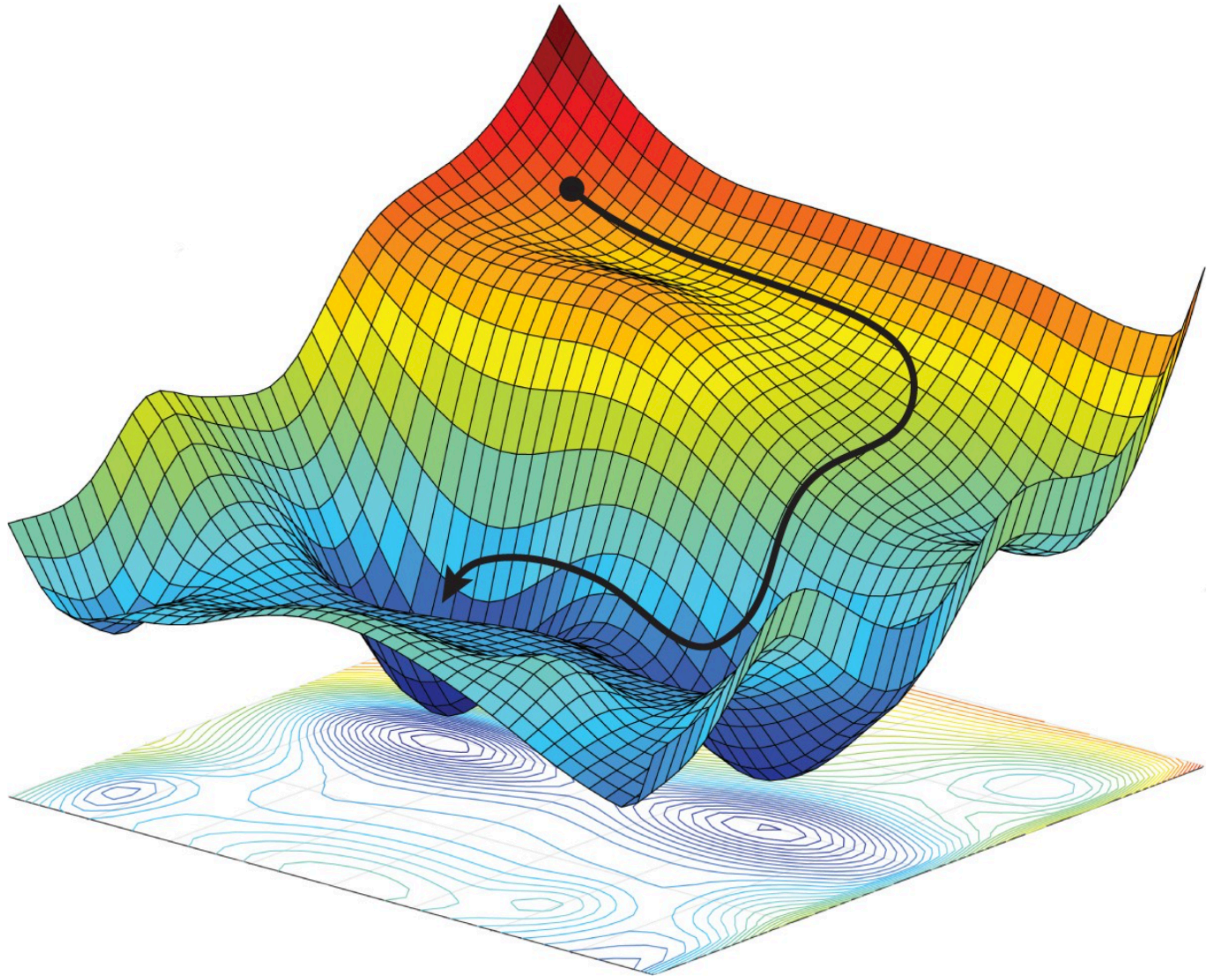
A diagram of a neural network



- Each node computes a weighted combination of its inputs
- For intermediate layers, if the result is negative, the output of the node is set to 0

The weights in each weighted combination are “knobs” that can be tuned

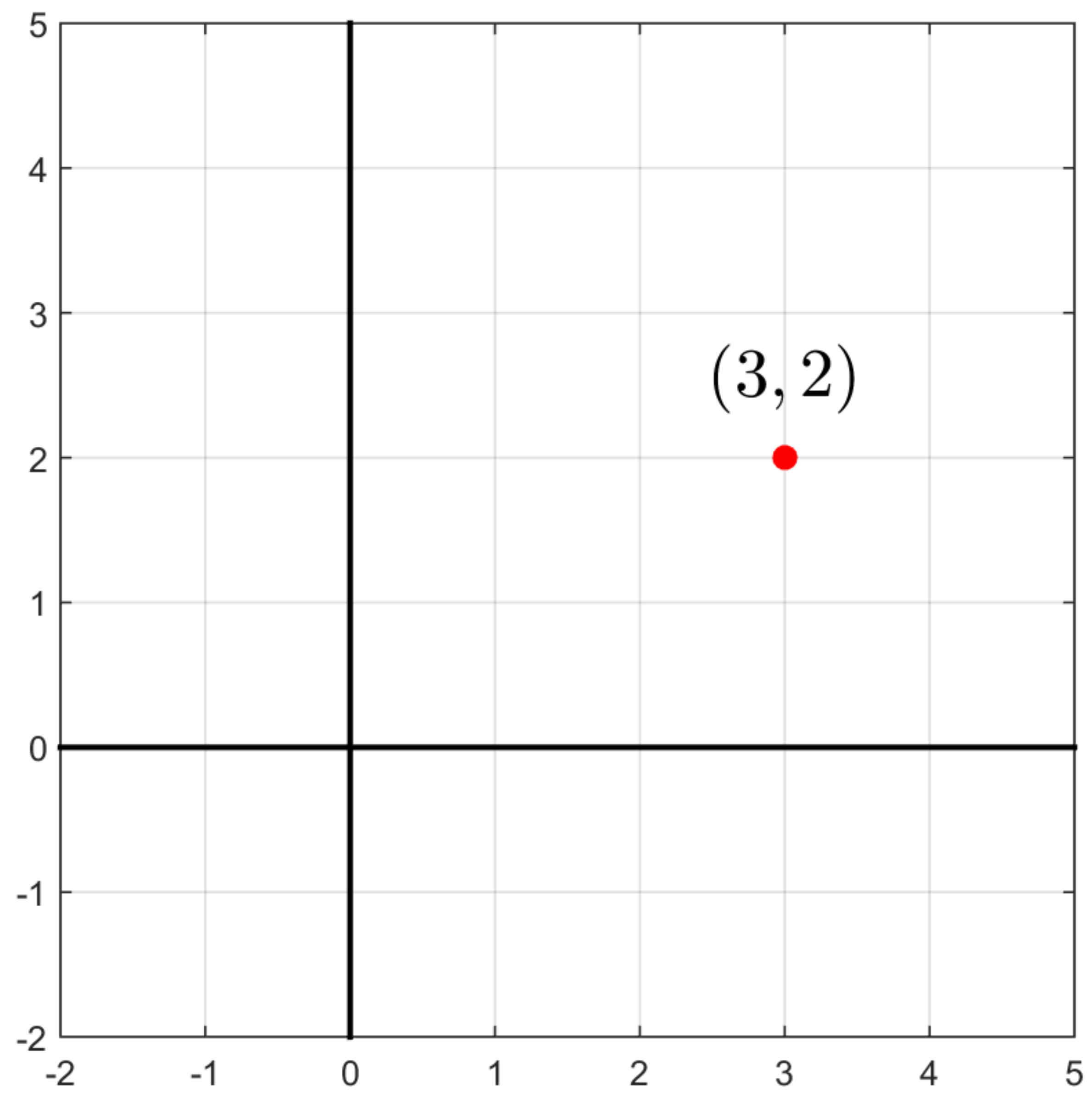
Optimization algorithms



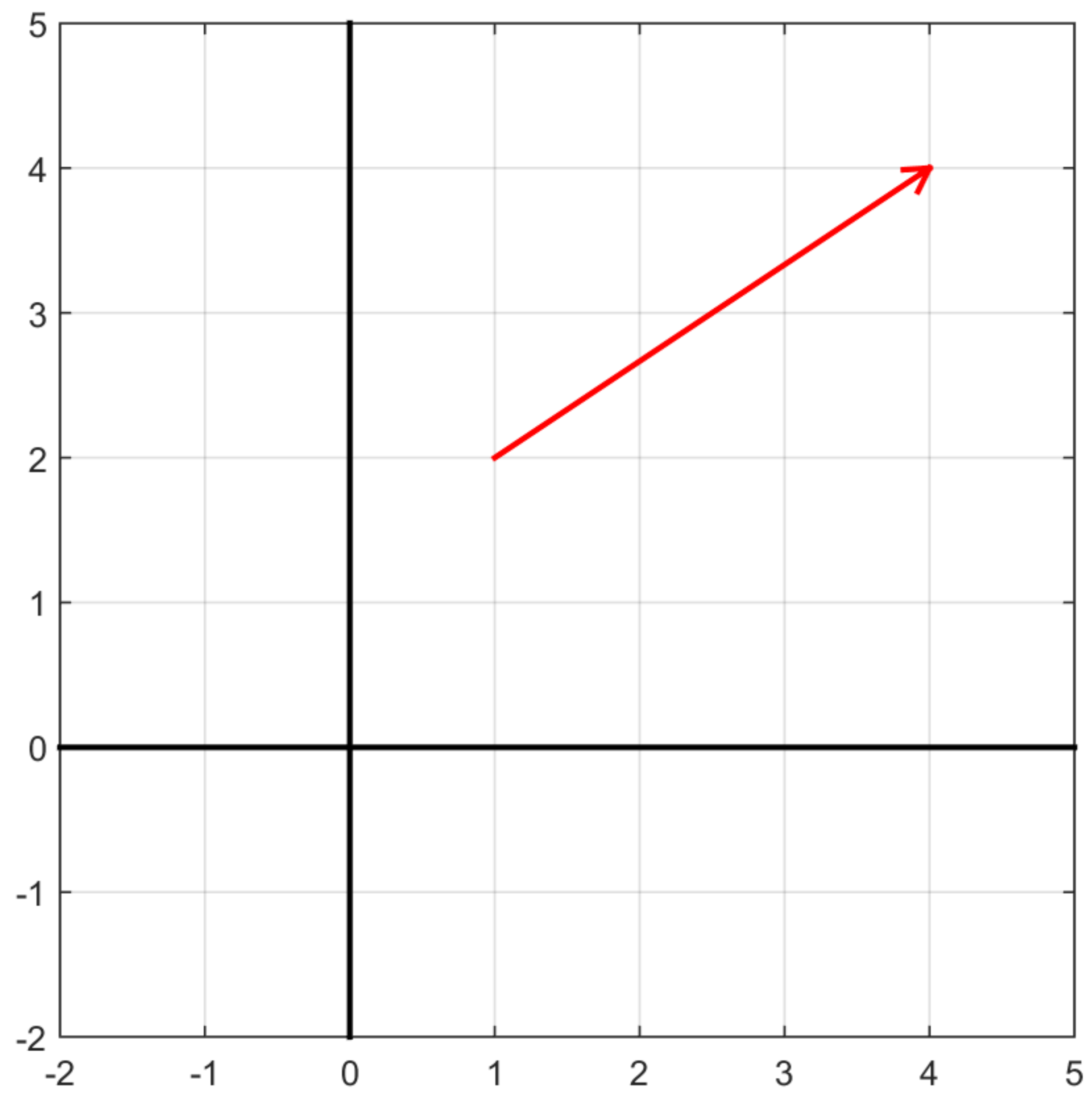
Visualizing an n-tuple

Ordered n-tuple: an ordered list of n numbers

Visualizing $\begin{bmatrix} 3 \\ 2 \end{bmatrix}$



Point picture

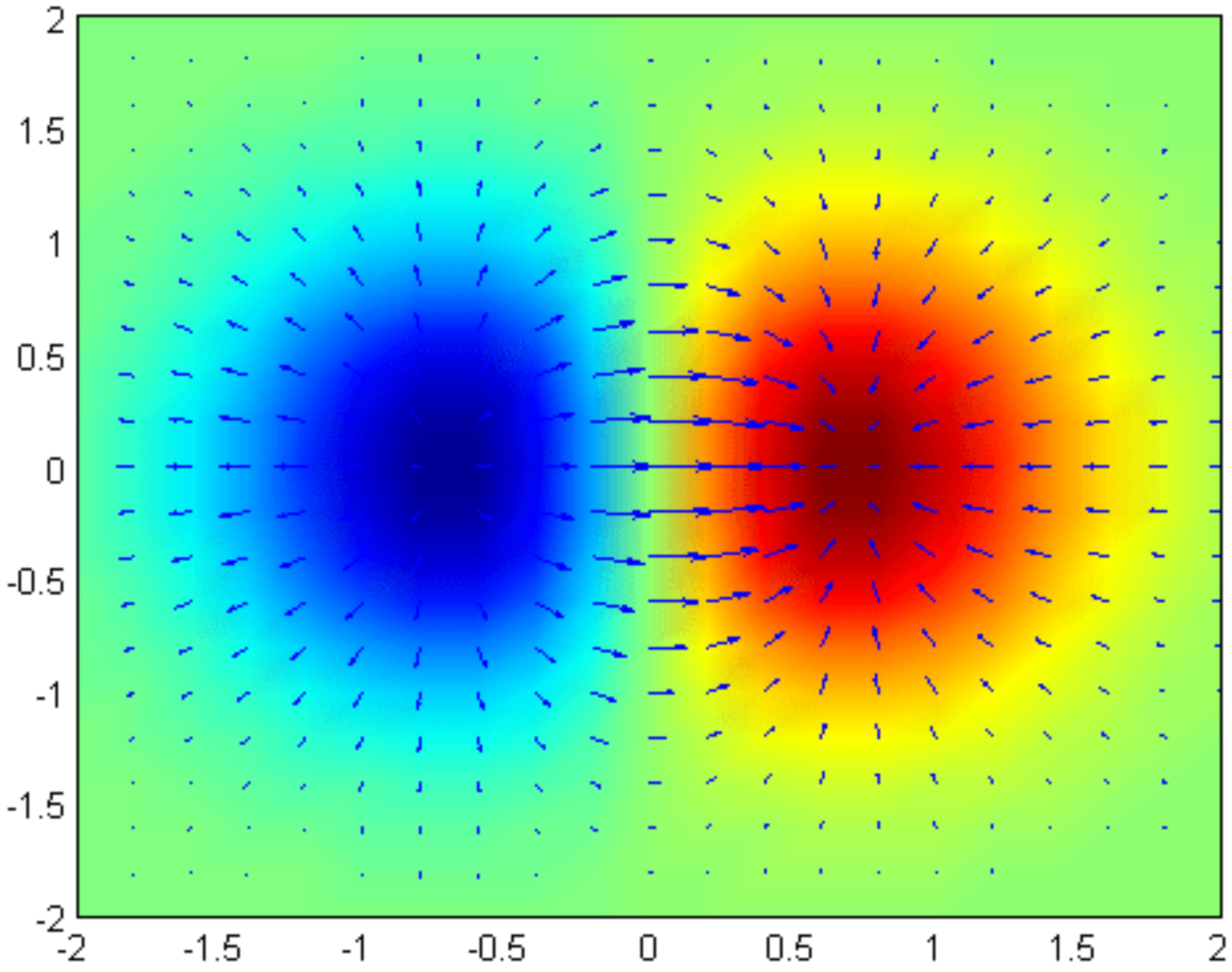


Vector picture

Gradient vector

$$L : \mathbb{R}^n \to \mathbb{R}$$

$L(\beta)$ is the temperature at the point β

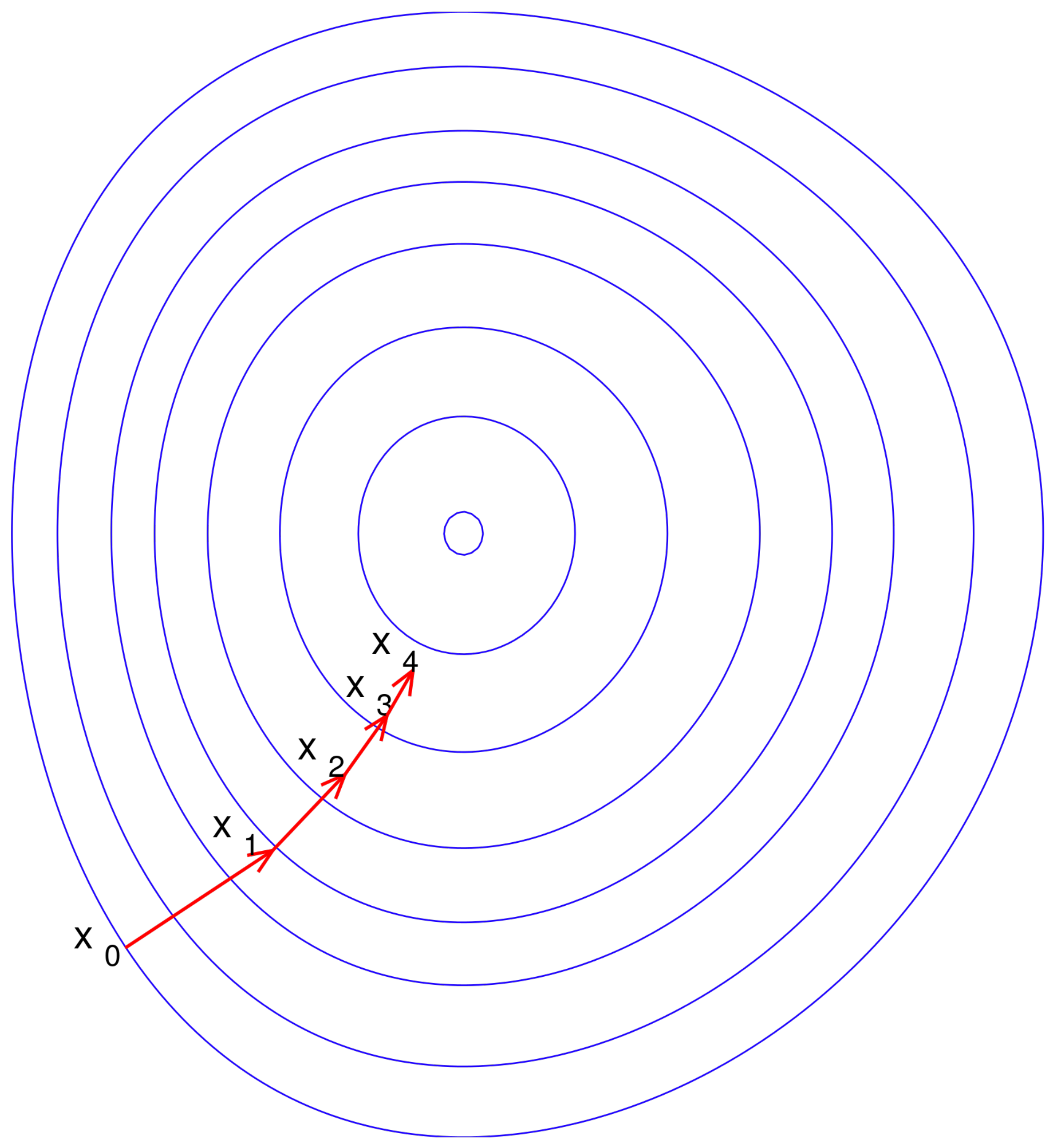


$\nabla L(\beta)$ points in the direction of steepest ascent

Optimization algorithm

Problem: minimize $L(\beta)$

Gradient descent: repeatedly move in direction of steepest descent



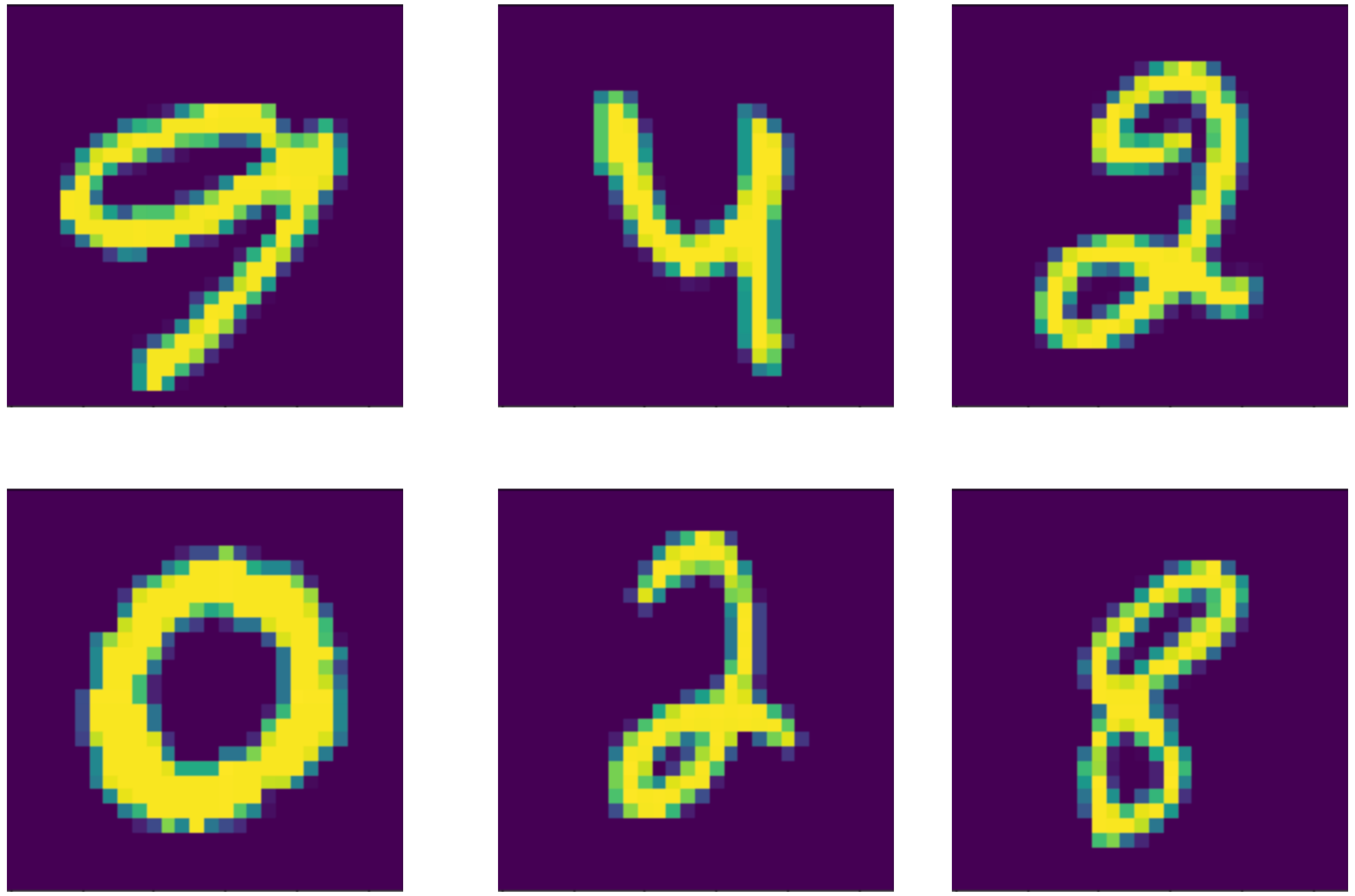
Initialize $\beta^0 \in \mathbb{R}^{d+1}$

Then do $\beta^{t+1} = \beta^t - \alpha \nabla L(\beta^t)$ for $t = 0, 1, 2, \ldots$

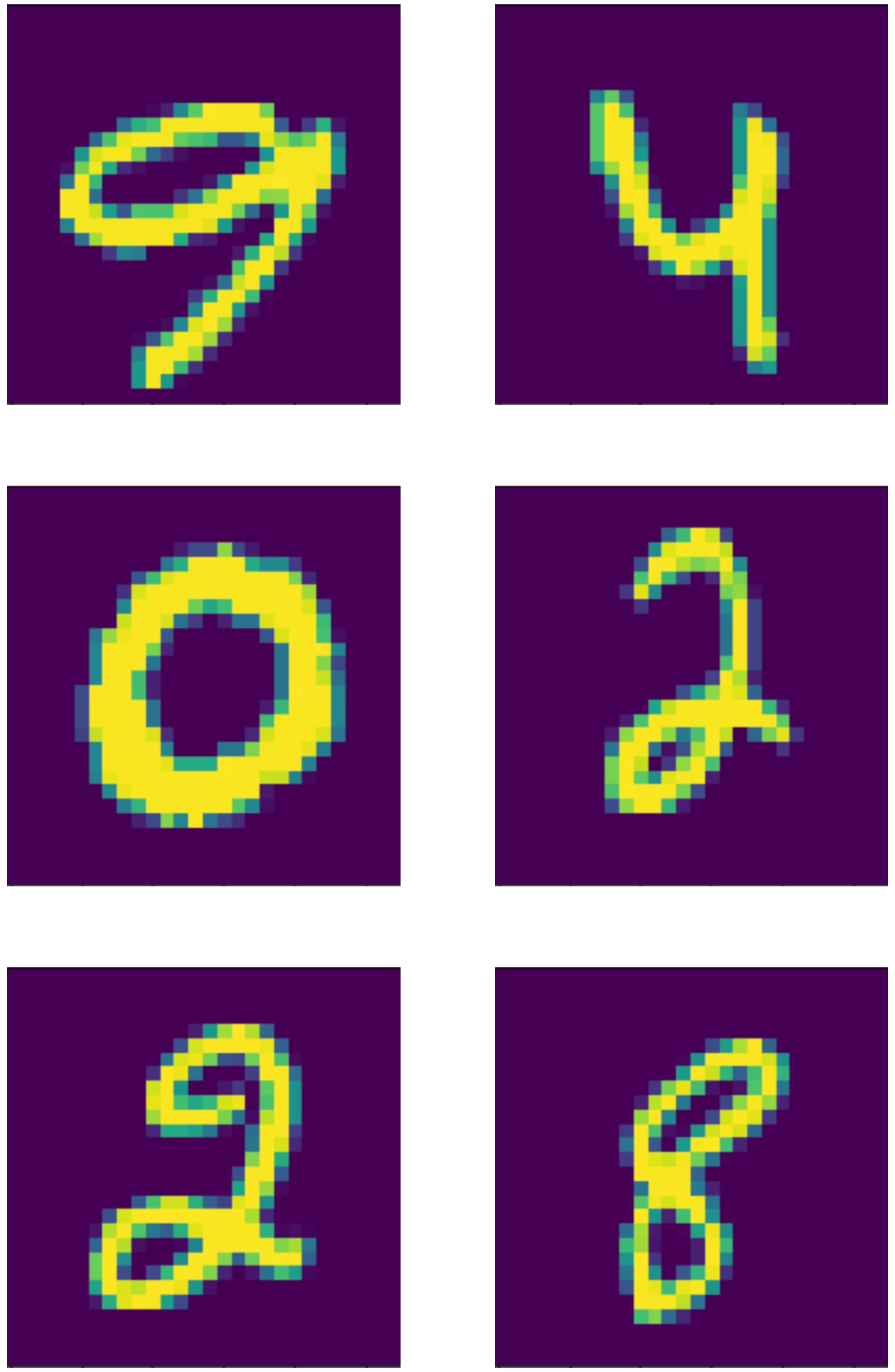
"Learning rate"

PyTorch computes the gradient for us

Handwritten digit classification using PyTorch



Using PyTorch for deep learning



```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import pandas as pd

        import torch
```

Load MNIST handwritten digit data in .csv format.

```
In [2]: # The MNIST dataset in .csv format can be found on Kaggle here:
        # https://www.kaggle.com/oddrationale/mnist-in-csv

        data_dir = '/Users/dvo/MNIST/'
        df_train = pd.read_csv(data_dir + 'mnist_train.csv')
        df_val = pd.read_csv(data_dir + 'mnist_test.csv')
```

Each 28 x 28 MNIST image
is stored as a row in a data frame

Using PyTorch for deep learning

Define a dataset class:

We must implement these three methods

In [3]:
```
class DigitsDataset(torch.utils.data.Dataset):

    def __init__(self, df):

        self.df = df

    def __len__(self):
        return len(self.df)

    def __getitem__(self, idx):

        row = self.df.iloc[idx]

        x = np.float32(row[1:].values)/255
        y = row[0]

        return x, y
```

Using PyTorch for deep learning

This object can get a batch of data from the dataset

Create training and validation datasets and dataloaders.

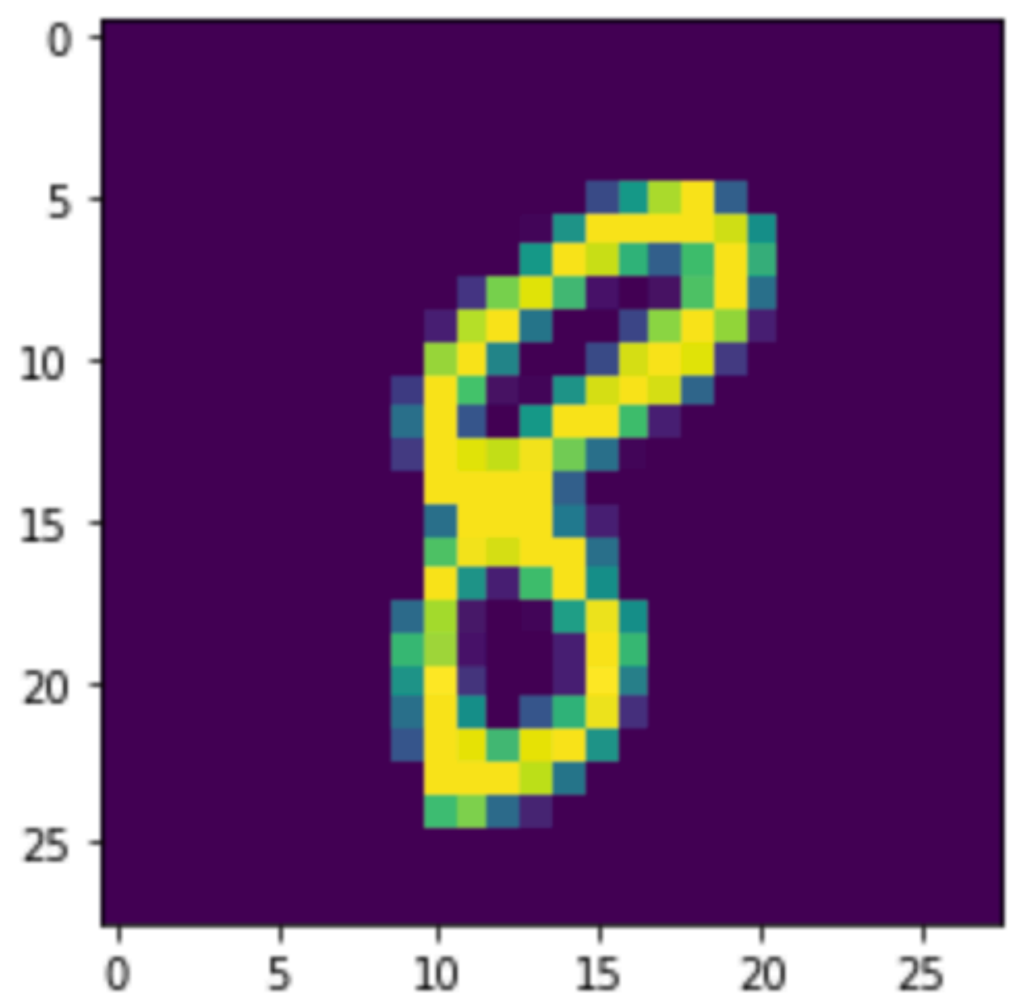
In [4]:
```
dataset_train = DigitsDataset(df_train)
dataset_val = DigitsDataset(df_val)

dataloader_train = torch.utils.data.DataLoader(dataset_train, batch_size=64,shuffle=True)
dataloader_val = torch.utils.data.DataLoader(dataset_val, batch_size=64,shuffle=True)
```

Look at a training example and its label. ¶

In [25]:
```
X_batch, Y_batch = next(iter(dataloader_train))
plt.imshow(np.reshape(X_batch[0],(28,28)))
print(Y_batch[0])
```

This command gets one batch of data

```
tensor(8)
```



Using PyTorch for deep learning

Define a model class that specifies our neural network architecture.

In [6]:
```
class SimpleNeuralNetwork(torch.nn.Module):

    def __init__(self):

        super().__init__()
        self.dense1 = torch.nn.Linear(784, 100)
        self.dense2 = torch.nn.Linear(100,10)

        self.ReLU = torch.nn.ReLU()
        # self.Softmax = torch.nn.Softmax(dim = 1)

    def forward(self, x):

        x = self.dense1(x)
        x = self.ReLU(x)
        x = self.dense2(x)
        # x = self.Softmax(x) # SoftMax is combined with the loss function, so not needed here.

        return x
```

Specify the layers in our neural network

This method applies the neural network to a vector x

Using PyTorch for deep learning

Create a model (our neural network).

In [7]:
```
model = SimpleNeuralNetwork()
device = torch.device('cpu') # Change this line if a GPU is available
model = model.to(device) # This line would put the model on the GPU, if device is a GPU.
```

← This is our neural network.

Choose the loss function and the optimization algorithm.

In [8]:
```
loss_fun = torch.nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr = 0.001)
```

Adam is a variant of stochastic gradient descent

Training the neural network

```
In [10]: num_epochs = 10
N_train = len(dataset_train)
N_val = len(dataset_val)

train_losses = []  # collect the training losses
val_losses = []

for ep in range(num_epochs):

    model.train() # Put model in train mode. This turns on any model behavior that should only occur during training.
    train_loss = 0.0
    batch_idx = 0

    for X_batch, Y_batch in dataloader_train:

        X_batch = X_batch.to(device) # If device is a GPU, this puts the current batch of data on the GPU.
        Y_batch = Y_batch.to(device)

        N_batch = X_batch.shape[0]
        outputs = model(X_batch)
        loss_oneBatch = loss_fun(outputs,Y_batch)

        model.zero_grad()
        loss_oneBatch.backward()
        optimizer.step()

        train_loss += loss_oneBatch*N_batch

    model.eval() # Put model in eval mode. This turns off any model behavior that should only occur during training.
    val_loss = 0.0
    for X_batch, Y_batch in dataloader_val:

        X_batch = X_batch.to(device)
        Y_batch = Y_batch.to(device)

        with torch.no_grad(): # Tell PyTorch it doesn't need to keep track of gradient info.

            N_batch = X_batch.shape[0]
            outputs = model(X_batch)
            loss_oneBatch = loss_fun(outputs,Y_batch)
            val_loss += loss_oneBatch*N_batch

    train_losses.append(train_loss/N_train)
    val_losses.append(val_loss/N_val)

    print('epoch: ', ep, 'train loss: ', train_loss/N_train, 'validation loss: ', val_loss/N_val)
```

We'll do 10 epochs of SGD

Sweep through training data, one batch at a time

This line does one iteration of stochastic gradient descent

PyTorch computes the gradient for us

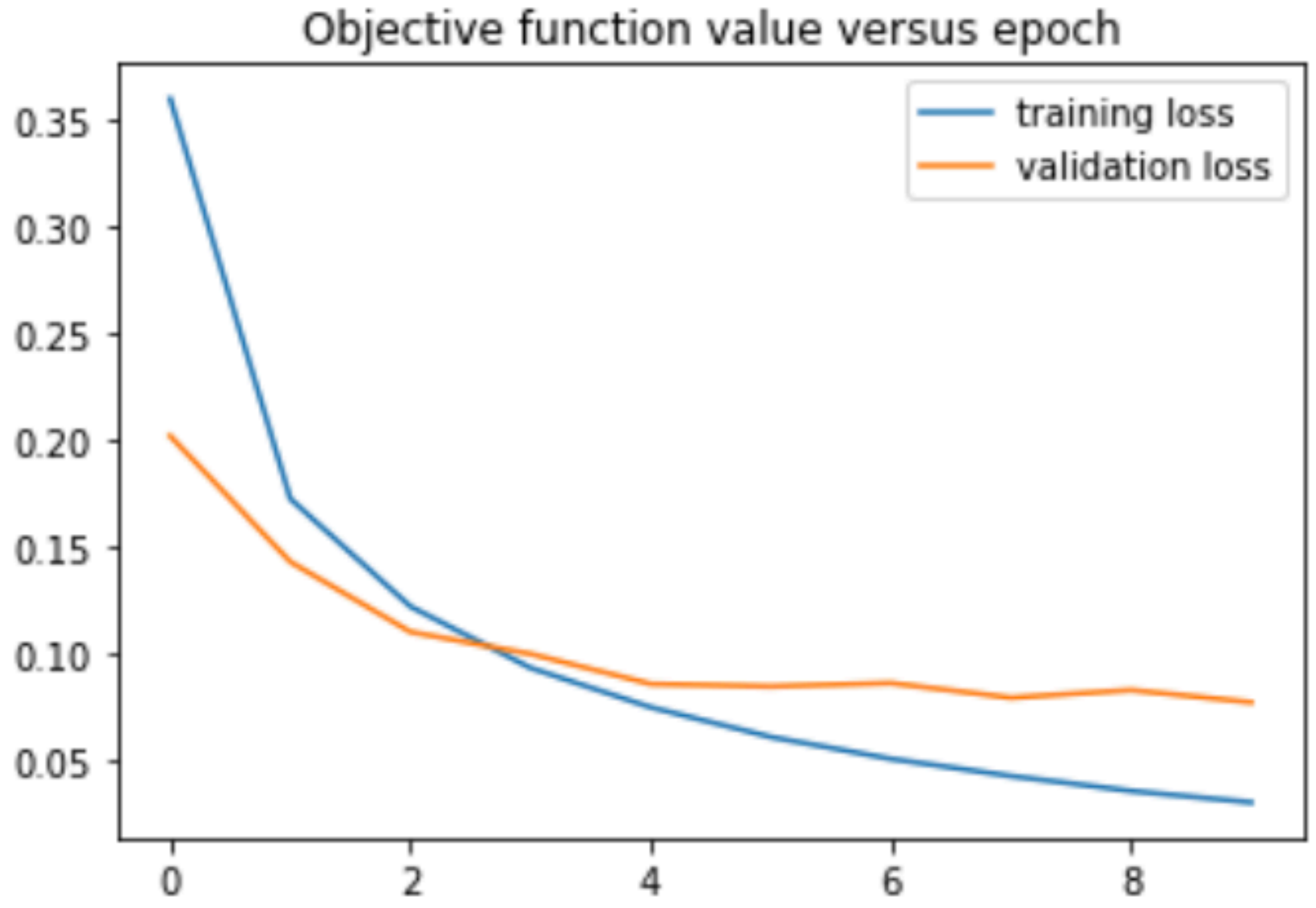
Report performance on both training and validation datasets

Using PyTorch for deep learning

Plot the objective function value vs. epoch for both the training and validation datasets.

In [12]:
```
plt.plot(train_losses, label = 'training loss')
plt.plot(val_losses, label = 'validation loss')
plt.legend(loc = 'upper right')
plt.title('Objective function value versus epoch')
```

Out[12]:
```
Text(0.5, 1.0, 'Objective function value versus epoch')
```



If the validation loss Starts increasing, we are overfitting the training data

Using PyTorch for deep learning

Compute our prediction accuracy on the validation dataset.

In [19]:
```
num_correct = 0
model.eval()

for X_batch, Y_batch in dataloader_val:

    X_batch = X_batch.to(device)
    Y_batch = Y_batch.to(device)

    with torch.no_grad(): # Tell PyTorch it doesn't need to keep track of gradient info.

        outputs = model(X_batch)
        num_correct += sum(np.argmax(outputs, axis = 1) == Y_batch)

print('Accuracy: ', num_correct/N_val)
```

```
Accuracy:  tensor(0.9762)
```

Count how many predicted labels agree with the ground truth labels