LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

Graduate School of
Quantitative Biosciences Munich

Artificial Neuronal Networks in Python

From Scratch!

Dr. Markus Hohle

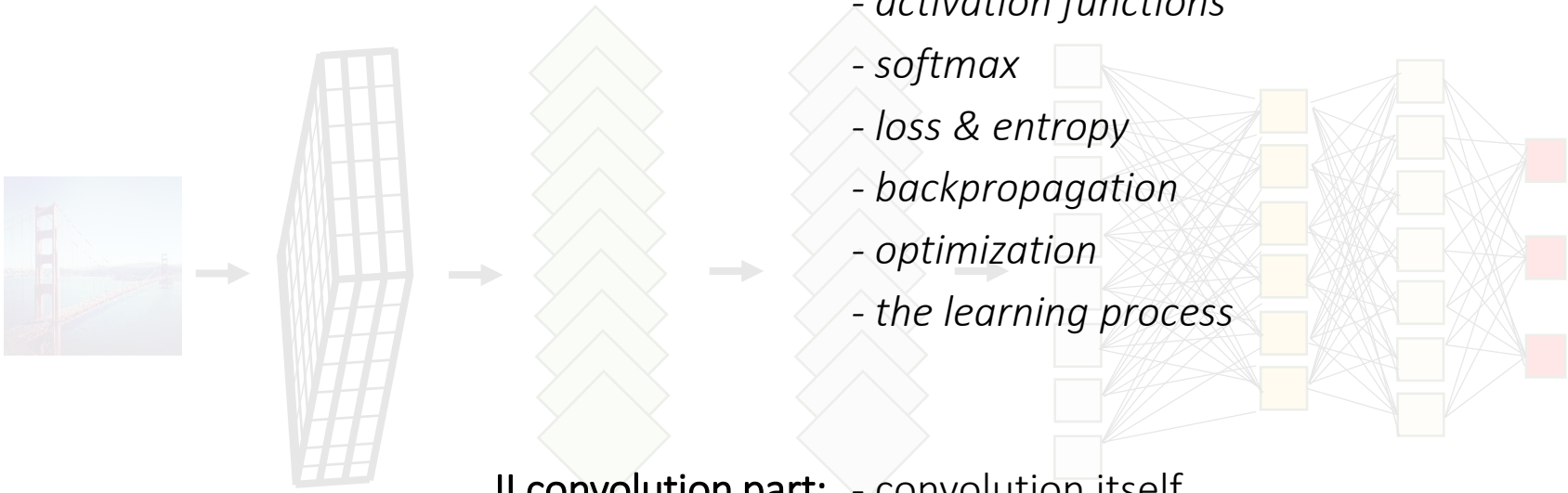
Gene Center of the LMU
Feodor-Lynen-Strasse 25
81377 Munich

<u>outline</u>

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*



II convolution part:

- convolution itself
- max pool and average pool
- batch normalization
- flattening
- learning and “teaching”
- final remarks

outline

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*

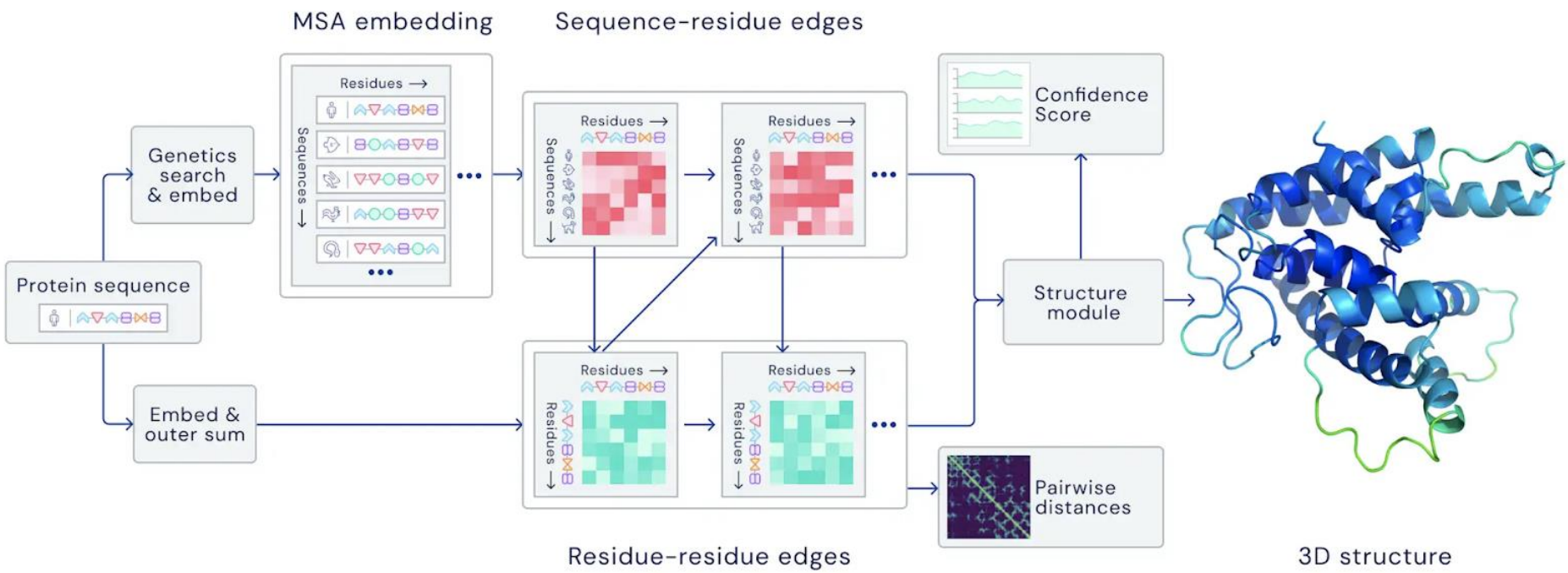
II convolution part:

- convolution itself
- max pool and average pool
- batch normalization
- flattening
- learning and “teaching”
- final remarks

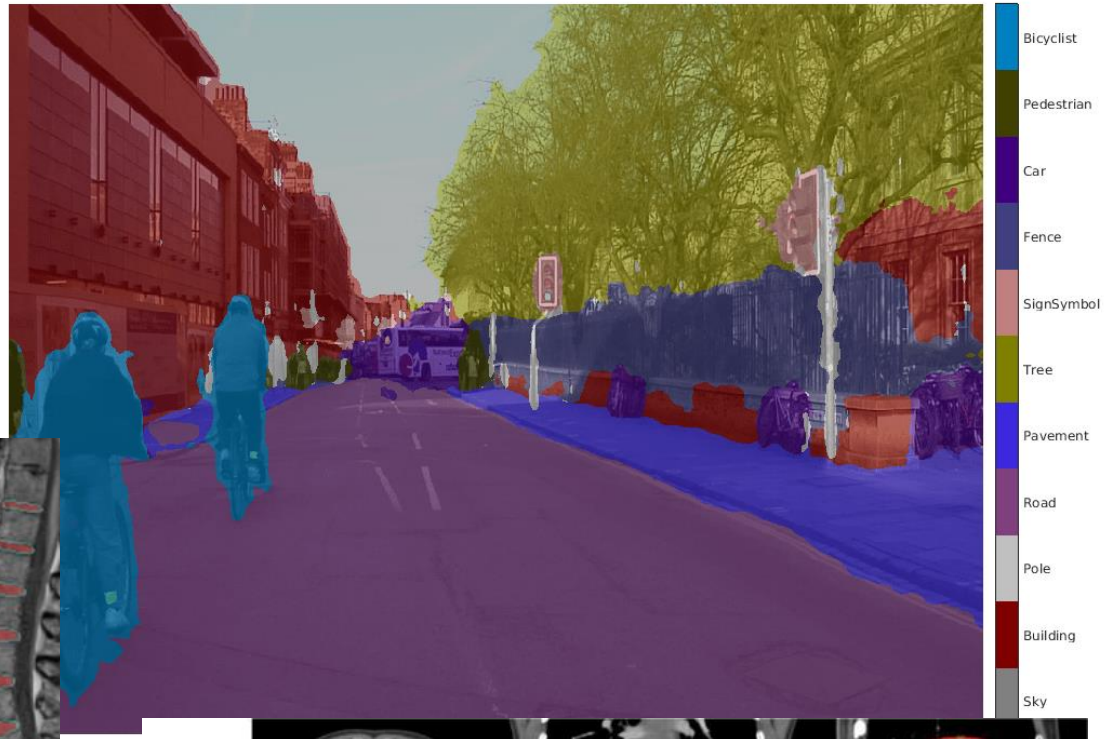
Why Artificial Neuronal Networks?

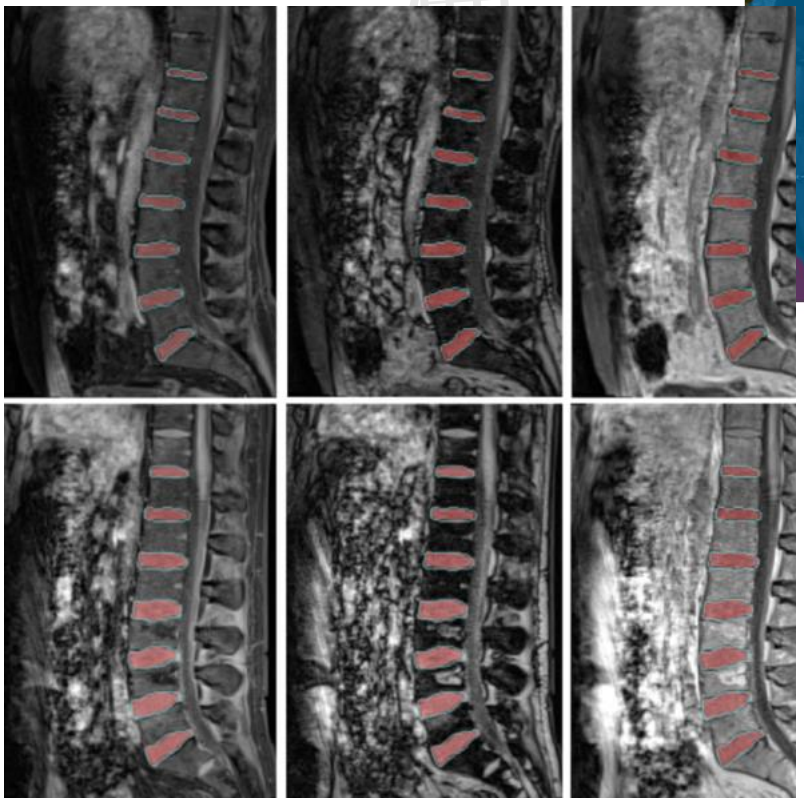
Demis Hassabis
@demishassabis

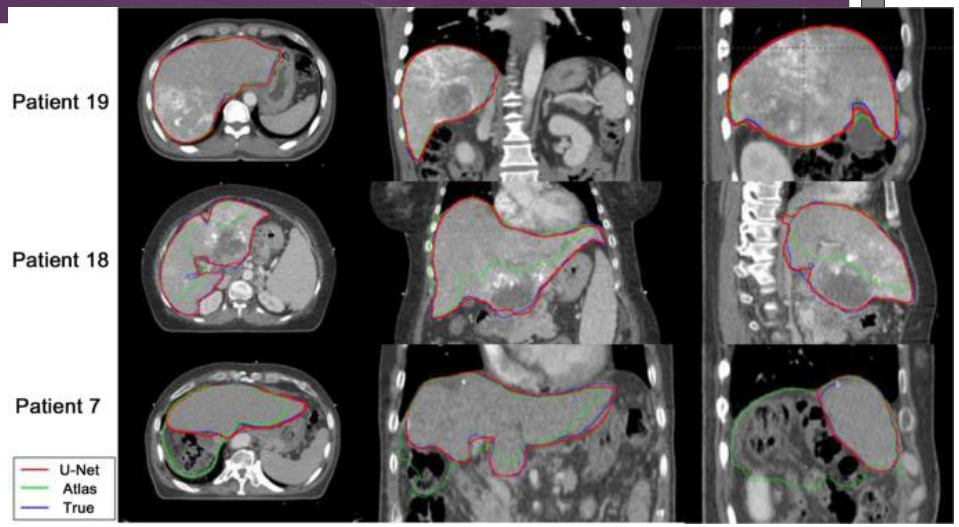
Thrilled to announce our first major breakthrough in applying AI to a grand challenge in science. #AlphaFold has been validated as a solution to the 'protein folding problem' & we hope it will have a big impact on disease understanding and drug discovery:



Why Artificial Neuronal Networks?



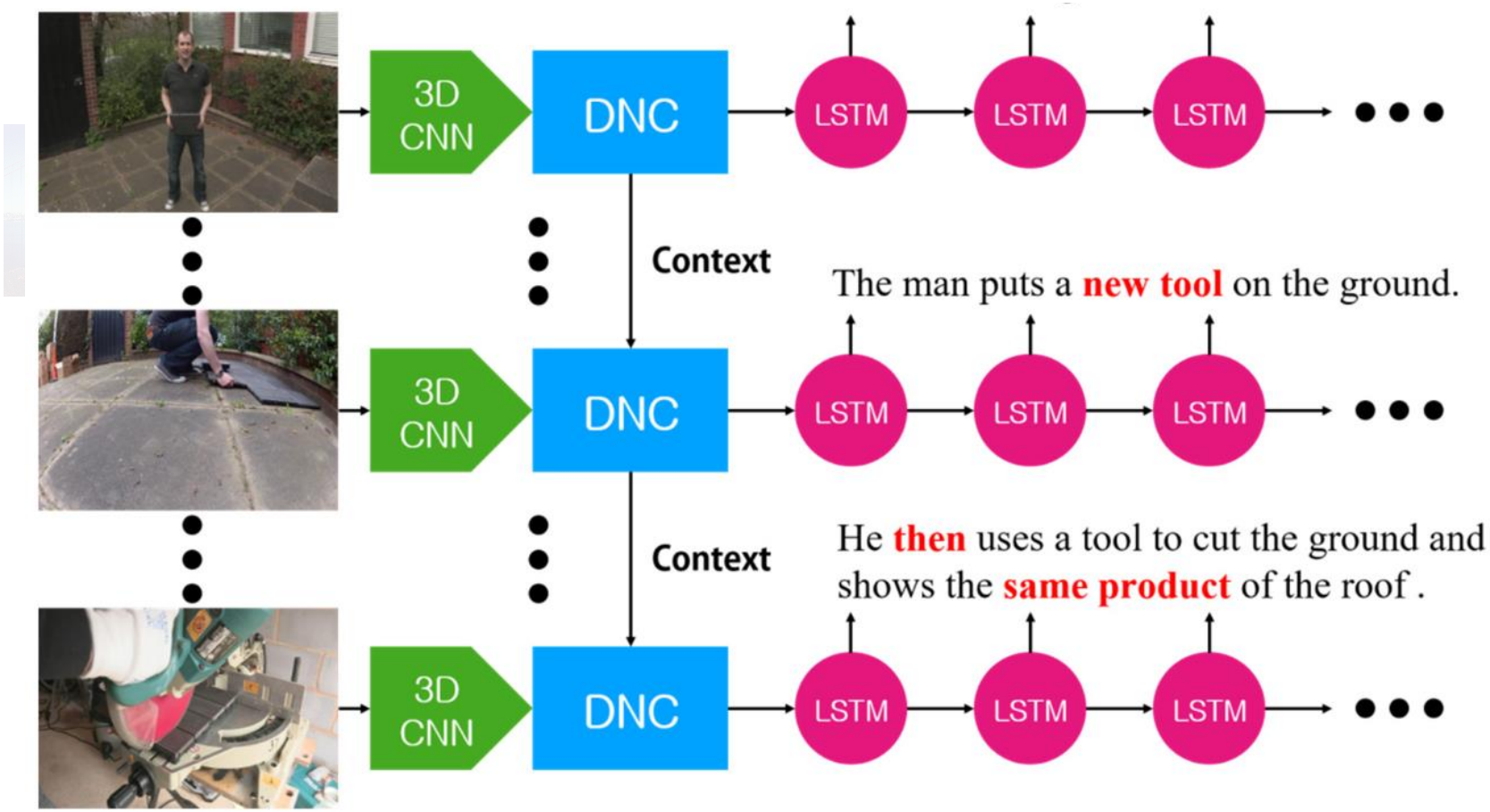




Why Artificial Neuronal Networks?

S:	我	3 就	4 取	5 钱	6 给	7 了	她们
	i	*will*	*get*	*money*	*to*	perf.	*them*
T:	2 i	1 will	0 get	the	money	to	them

P(the | get, will, i, 就, 取, 钱, 给, 了)



Why Artificial Neuronal Networks?

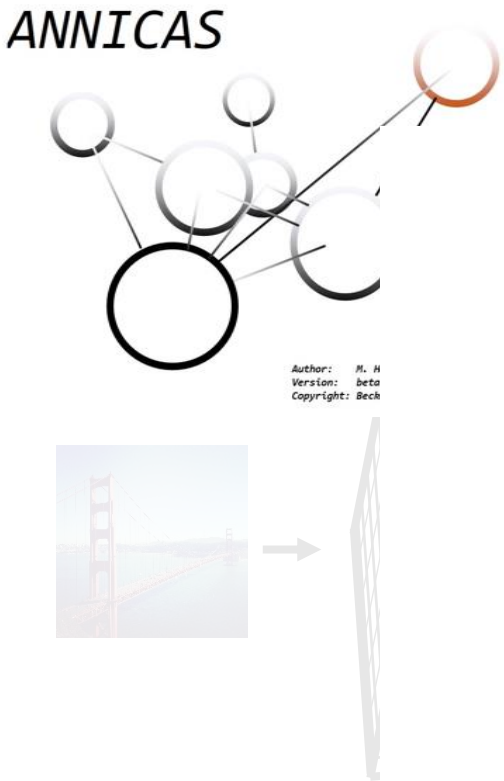
Article

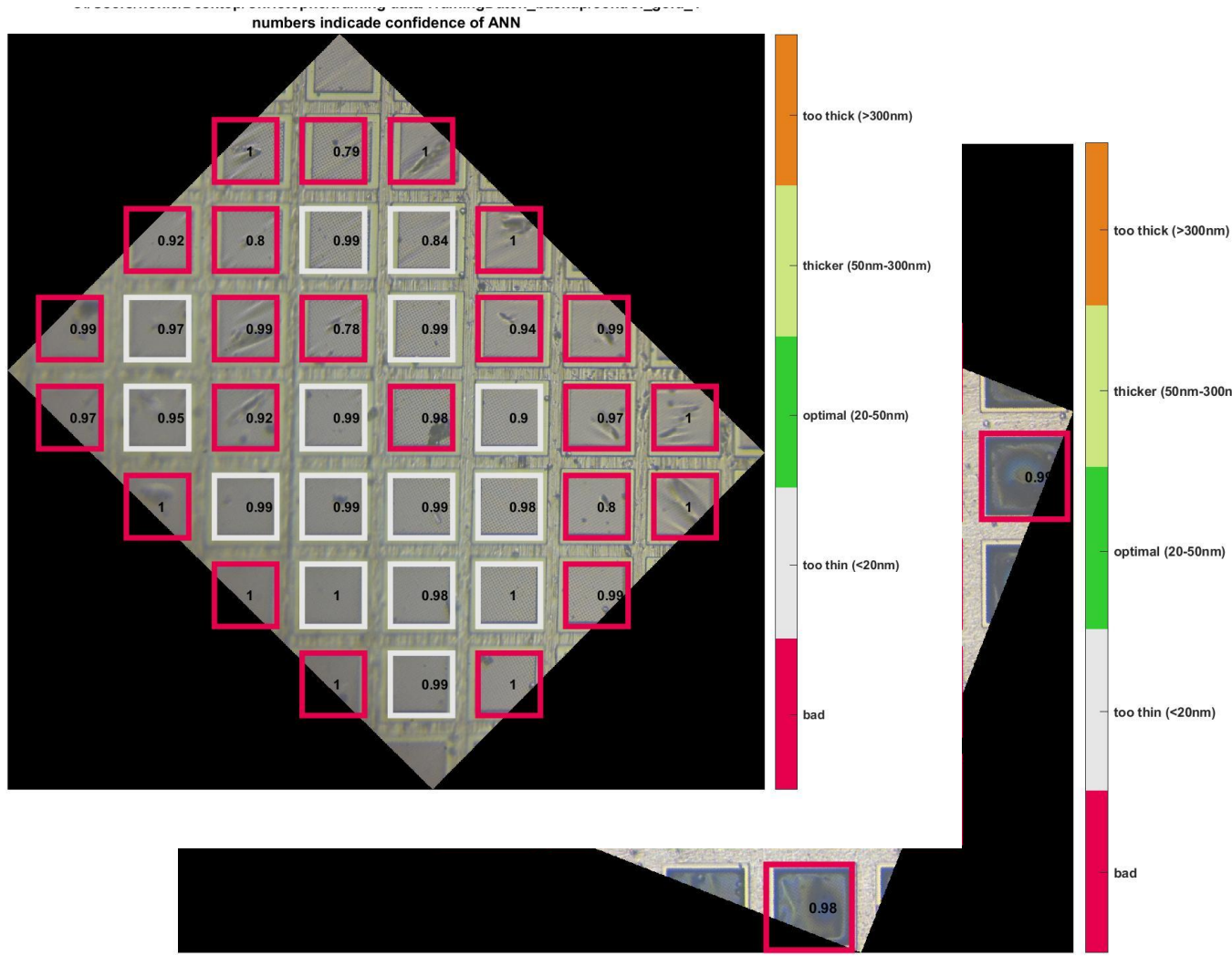
DeepSOCIAL: Social Distancing Monitoring and Infection Risk Assessment in COVID-19 Pandemic

YOLOv4-based *Deep Neural Network* (DNN) model for automated people detection in the crowd in indoor and outdoor environments using common CCTV security cameras. The proposed DNN model in combination with an adapted inverse perspective mapping (IPM) technique and SORT tracking algorithm leads to a robust people detection and social distancing monitoring. The model has been trained against two most comprehensive datasets by the time of the research—the Microsoft Common Objects in Context (MS COCO) and Google Open Image datasets. The system has been

health authorities have set the 2-m physical distancing as a mandatory safety measure in shopping centres, schools and other covered areas. In this research, we develop a hybrid *Computer Vision* and YOLOv4-based *Deep Neural Network* (DNN) model for automated people detection in the crowd in indoor and outdoor environments using common CCTV security cameras. The proposed DNN model in combination with an adapted inverse perspective mapping (IPM) technique and SORT tracking algorithm leads to a robust people detection and social distancing monitoring. The model has been trained against two most comprehensive datasets by the time of the research—the Microsoft Common Objects in Context (MS COCO) and Google Open Image datasets. The system has been evaluated against the Oxford Town Centre dataset (including 150,000 instances of people detection)

Why Artificial Neuronal Networks?



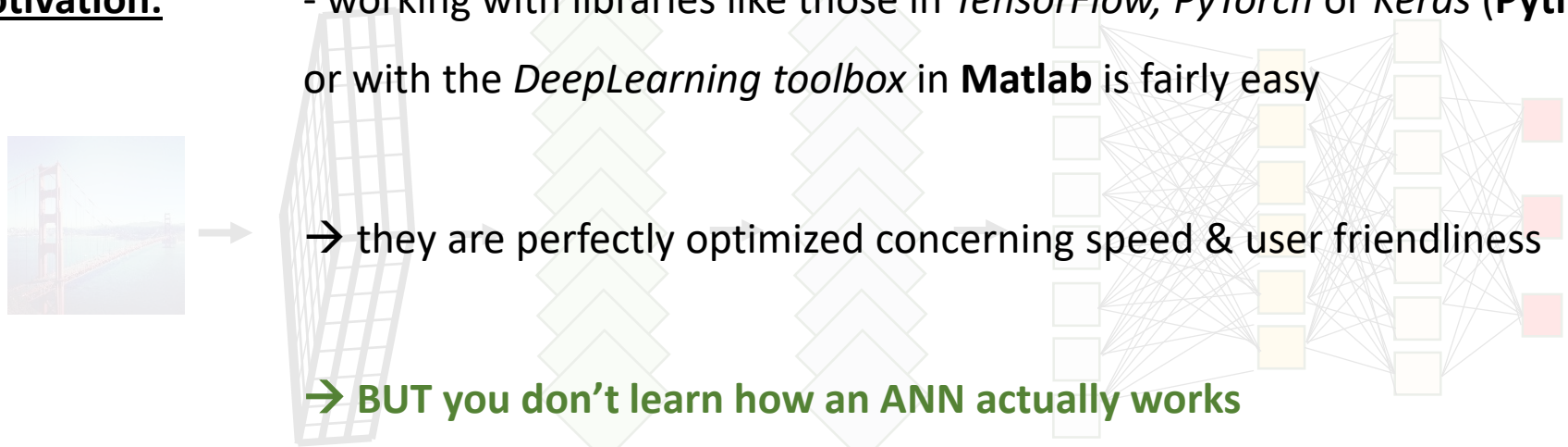


aim:

- creating a **simple**, but **fully functional convolutional ANN**, in Python
- **no use of external** ANN related **libraries** (only numpy!)

motivation:

- working with libraries like those in *TensorFlow, PyTorch* or *Keras* (**Python**) or with the *DeepLearning toolbox* in **Matlab** is fairly easy



→ they are perfectly optimized concerning speed & user friendliness

→ BUT you don't learn how an ANN actually works

→ you can build it = you understand it!

→ knowing how an ANN works exactly is a valuable skillset…

…and interesting anyway

→ we will see how somewhat abstract math turns into actual code

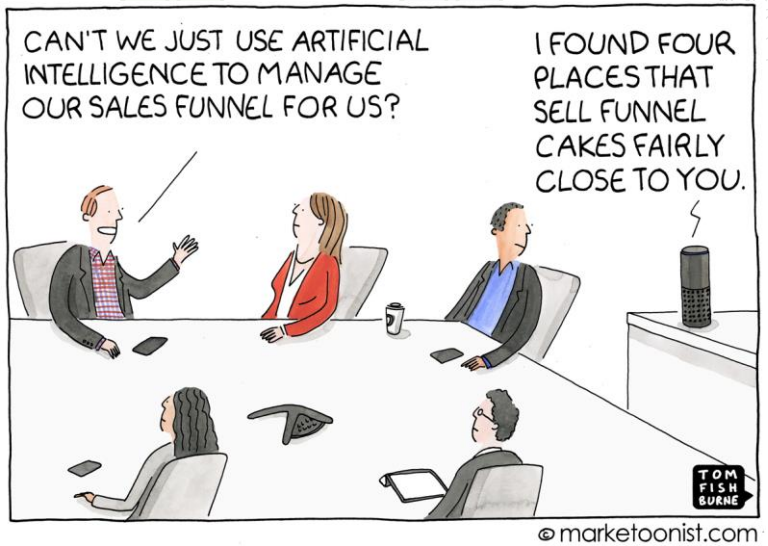
scope:

- we won't get a code that is extremely fast and optimized concerning runtime **(if so, those people programming the deep learning libraries would have done an awful job ;P)**
- no nice graphical interface

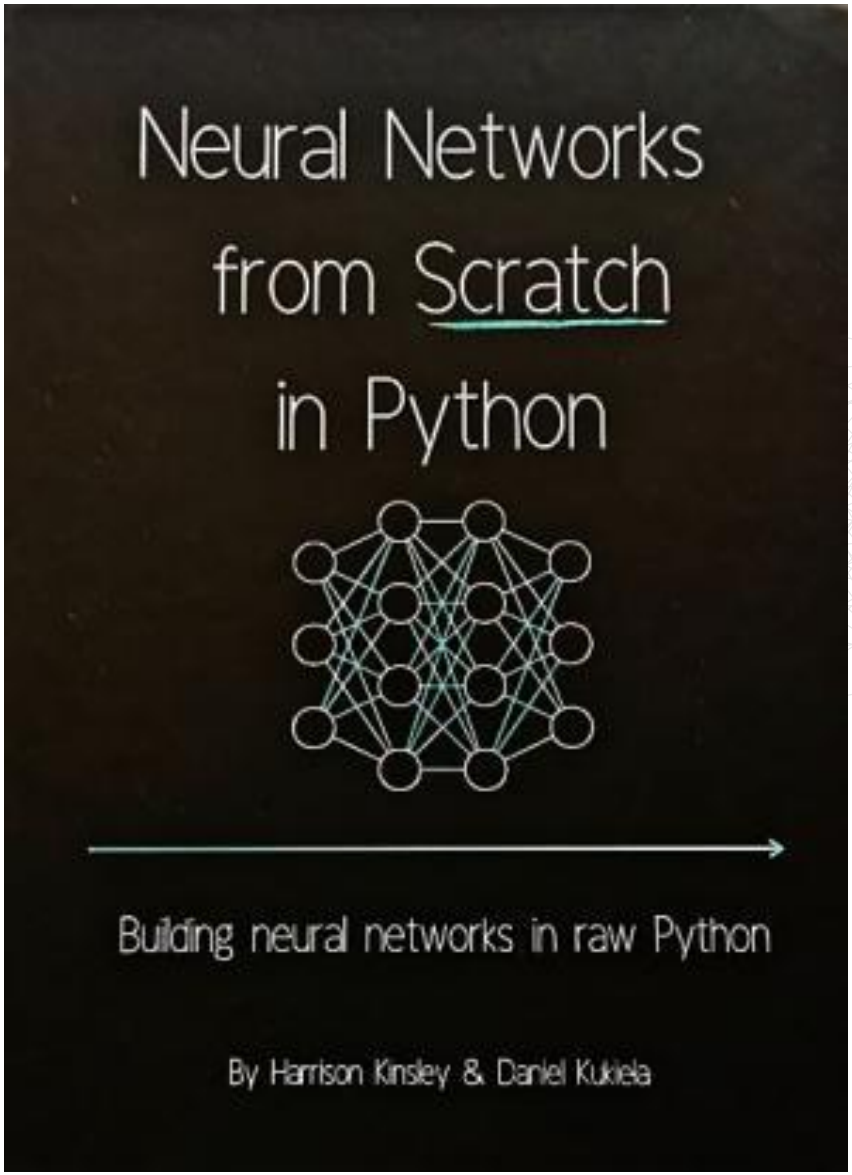
prerequisites:

- high school math:
  - o derivatives
  - o matrix and vector multiplication
  - o mean & variance
- basic knowledge in Python or Matlab (syntax is similar)

→ see my intro courses :)



literature: this lecture us has been inspired by the book with the same name:

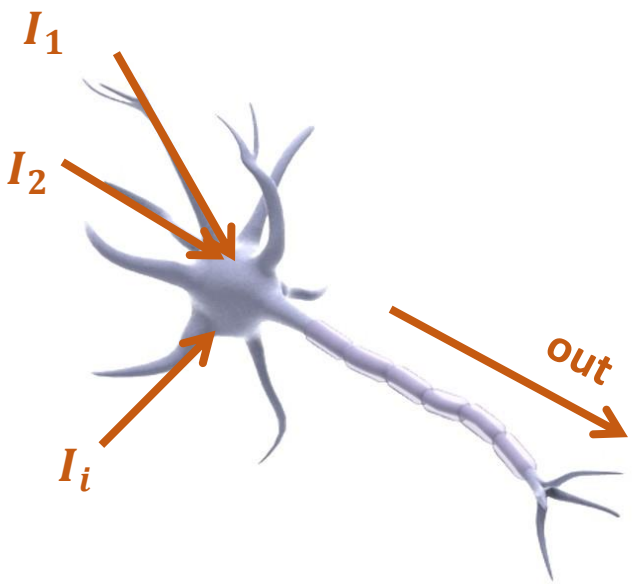


outline

0 intro

I the core:
- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*

II convolution part:
- convolution itself
- max pool and average pool
- batch normalization
- flattening
- learning and “teaching”
- final remarks



what we know…

- inputs enter the neuron
- something happens inside the neuron
- neuron generates an output

…how we could model it

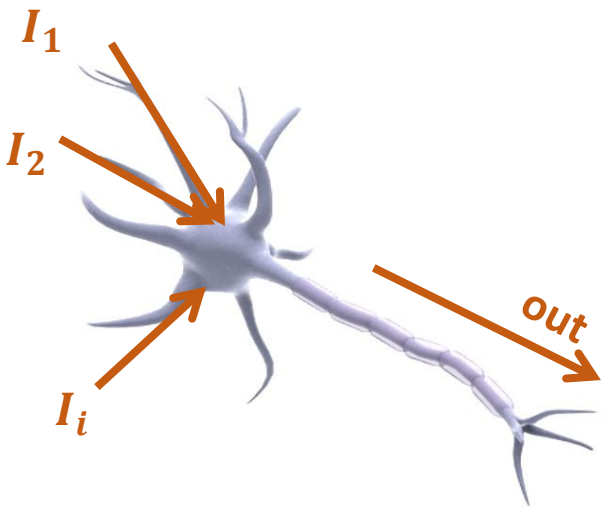
- inputs must be weighted (important vs unimportant input)
- **learning process: changing weights**
- output = sum of weighted inputs

I_i input *i*

w_i corresponding weight

b bias (base potential)

$$\boxed{o = \underbrace{\sum_i I_i \cdot w_i}_{\textit{dot product}} + b}$$

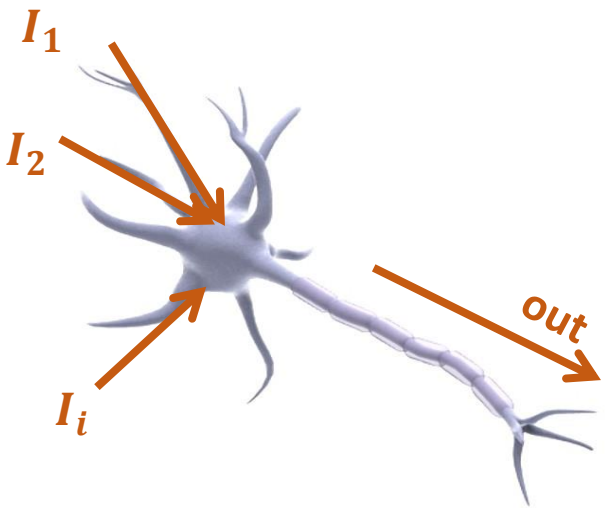


$$o = \sum_i I_i \cdot w_i + b$$

I_i	input *i*
w_i	corresponding weight
b	bias (base potential)

```
def single_neuron(inputs):

    import numpy as np

    l = len(inputs)

    weights = np.random.rand(1,l)
    bias    = np.random.rand(1,1)

    out     = np.dot(weights,inputs) + bias

    return(out)
```

save and run the function

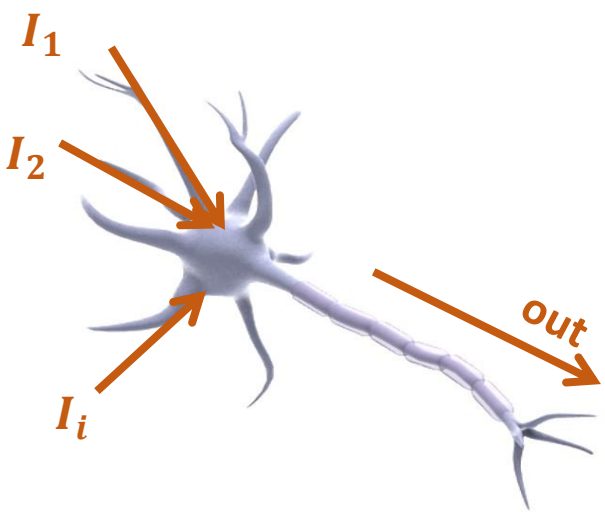


$$o = \sum_i I_i \cdot w_i + b$$

I_i	input *i*
w_i	corresponding weight
b	bias (base potential)

save and run the function

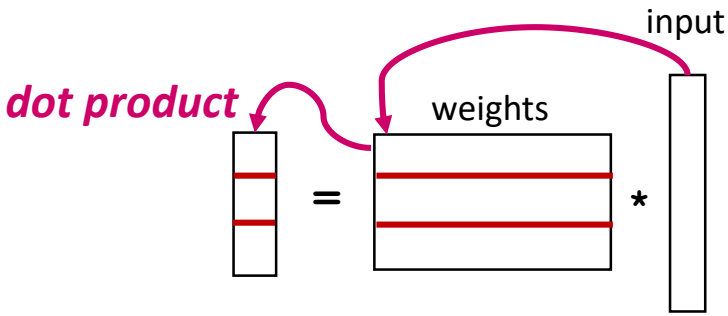
```
In [9]: I = [1,2,-4,5]

In [10]: out = single_neuron(I)

In [11]: print(out)
[[2.54011995]]
```

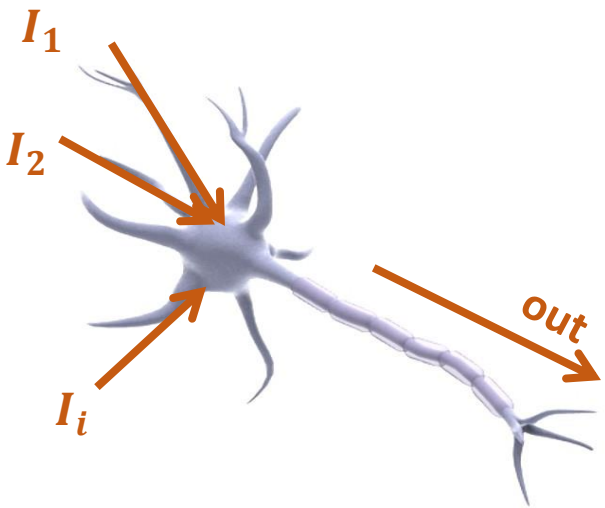


$$o = \sum_i I_i \cdot w_i + b$$

I_i	input *i*
w_i	corresponding weight
b	bias (base potential)

```
def threeNeurons_Onelayer(inputs):

    import numpy as np

    l = len(inputs)

    weights = np.random.rand(3,l)
    bias    = np.random.rand(3)

    out     = np.dot(weights,inputs) + bias

    return(out)
```





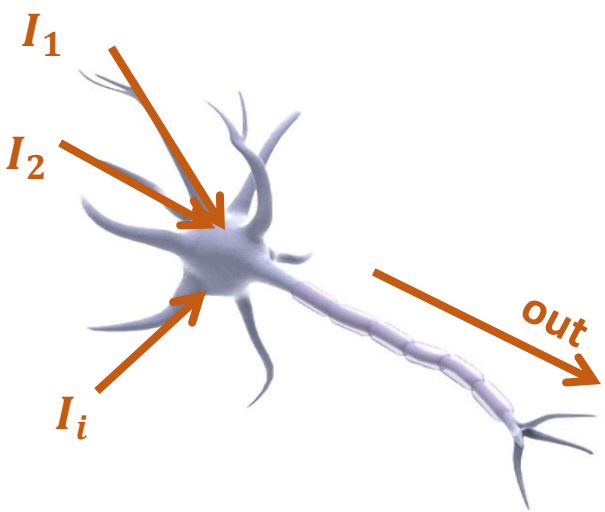
$$o = \sum_i I_i \cdot w_i + b$$

I_i	input *i*
w_i	corresponding weight
b	bias (base potential)

```
def threeNeurons_Onelayer(inputs):

    import numpy as np

    l = len(inputs)

    weights = np.random.rand(3,l)
    bias    = np.random.rand(3)

    out     = np.dot(weights,input

    return(out)
```

save and run the function

```
In [18]: I = [1,2,-4,5]

In [19]: out = threeNeurons_Onelayer(I)

In [20]: print(out)
[0.81383994 3.87728859 2.61411776]
```



$$o = \sum_i I_i \cdot w_i + b$$

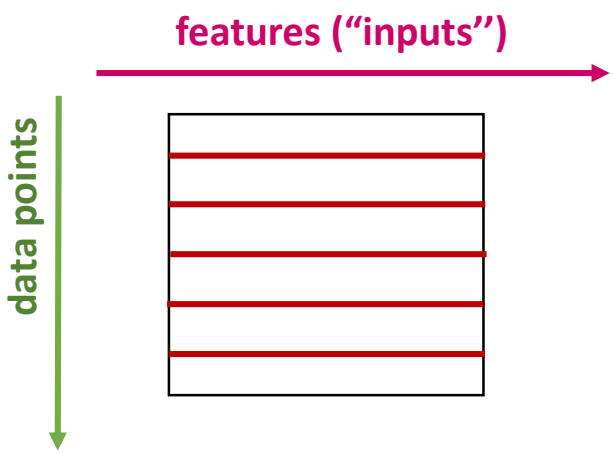
I_i	input *i*
w_i	corresponding weight
b	bias (base potential)

<u>problem:</u> - our input is actually a **matrix** *I*

- would like to link our neurons in a concise way

- passing on the information

→ *layers*



outline

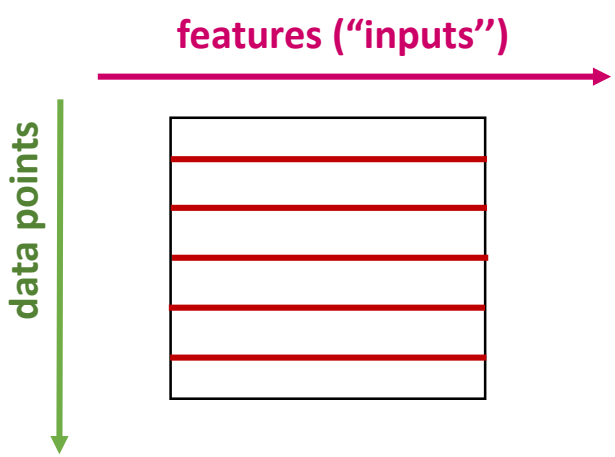
0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*

II convolution part:

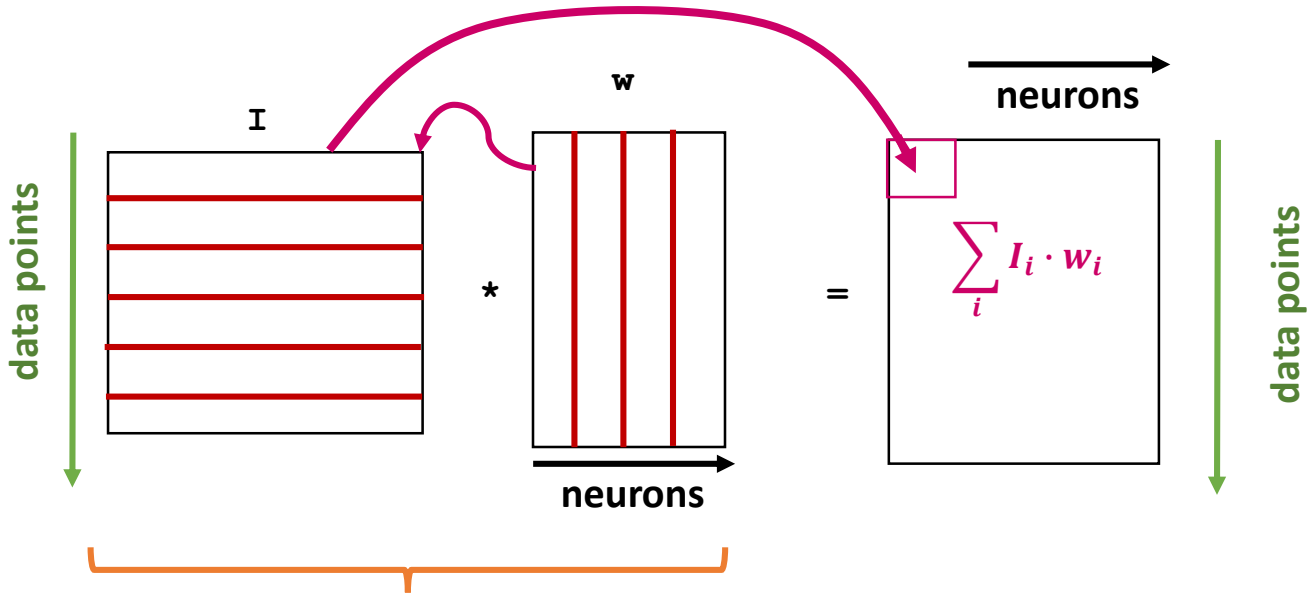
- convolution itself
- max pool and average pool
- batch normalization
- flattening
- learning and “teaching”
- final remarks



most convenient:

constructing a layer as `class`
→ setting **weight**, **bias** and **output** as *attribute*

`n_inputs:` number of features per observation
`n_neurons:` number of neurons

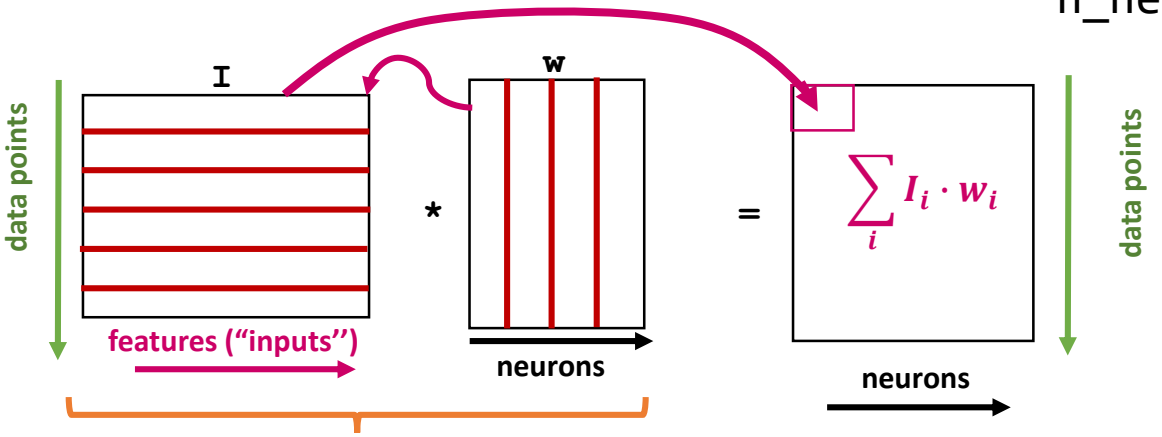


`np.dot(inputs, weights)`

constructing a layer as `class`
→ setting **weight**, **bias** and **output** as *attribute*

`n_inputs:` number of features per observation

`n_neurons:` number of neurons



`np.dot(inputs, weights)`

```
class Layer_Dense:
```

initializing weights/biases

```
    def __init__(self, n_inputs, n_neurons):
        self.weights = np.random.rand(n_inputs, n_neurons)
        self.biases  = np.zeros((1, n_neurons))
```

as before: creating output; now as attribute

```
    def forward(self, inputs):
        self.output  = np.dot(inputs, self.weights) + \
                       self.biases
```

constructing a layer as `class`
→ setting **weight**, **bias** and **output** as *attribute*

`n_inputs:` number of features per observation

`n_neurons:` number of neurons

```
class Layer_Dense:


        def __init__(self, n_inputs, n_neurons):
            self.weights = np.random.rand(n_inputs, n_neurons)
            self.biases  = np.zeros((1, n_neurons))


        def forward(self,inputs):
            self.output  = np.dot(inputs, self.weights) + \
                         self.biases
```

run e.g.

```
I            = [[1,3,4,5], [-2, -5, -6, 0]]

dense        = Layer_Dense(4, 3)
dense.forward(I)

print(dense.output)
```

constructing a layer as `class`
→ setting **weight**, **bias** and **output** as *attribute*

`n_inputs:` number of features per observation

`n_neurons:` number of neurons

<u>We can now easily link layers:</u>

```
I      = [[1,3,4,5], [-2, -5, -6, 0]]
S      = np.shape(I)
```

n_inputs

n_neurons

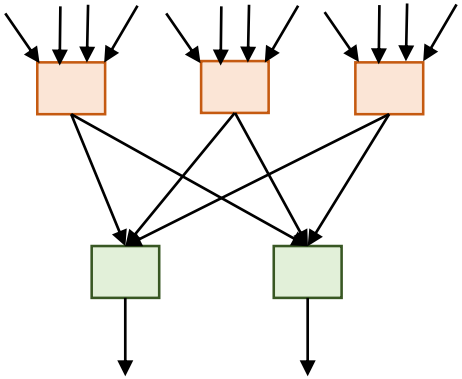
```
dense1 = Layer_Dense(S[1], 3)
```

defines the number of columns for next layer

```
dense2 = Layer_Dense(3, 2)
```

we will get two outputs per data point

```
dense1.forward(I)

dense2.forward(dense1.output)

print(dense1.output)
print(dense2.output)
```



constructing a layer as `class`
→ setting **weight**, **bias** and **output** as *attribute*

`n_inputs:` number of features per observation

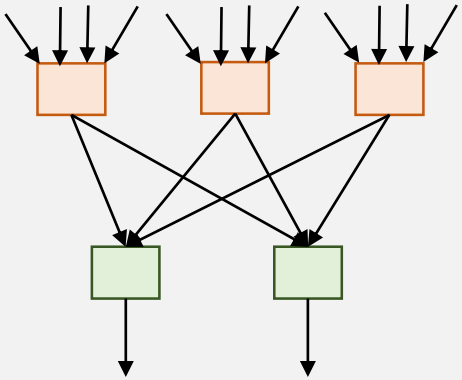
`n_neurons:` number of neurons

<u>We can now easily link layers:</u>

```
I      = [[1,3,4,5], [-2, -5, -6, 0]]
S      = np.shape(I)

dense1 = Layer_Dense(S[1], 3)
dense2 = Layer_Dense(3, 2)
```

defining layers

```
dense1.forward(I)

dense2.forward(dense1.output)
```



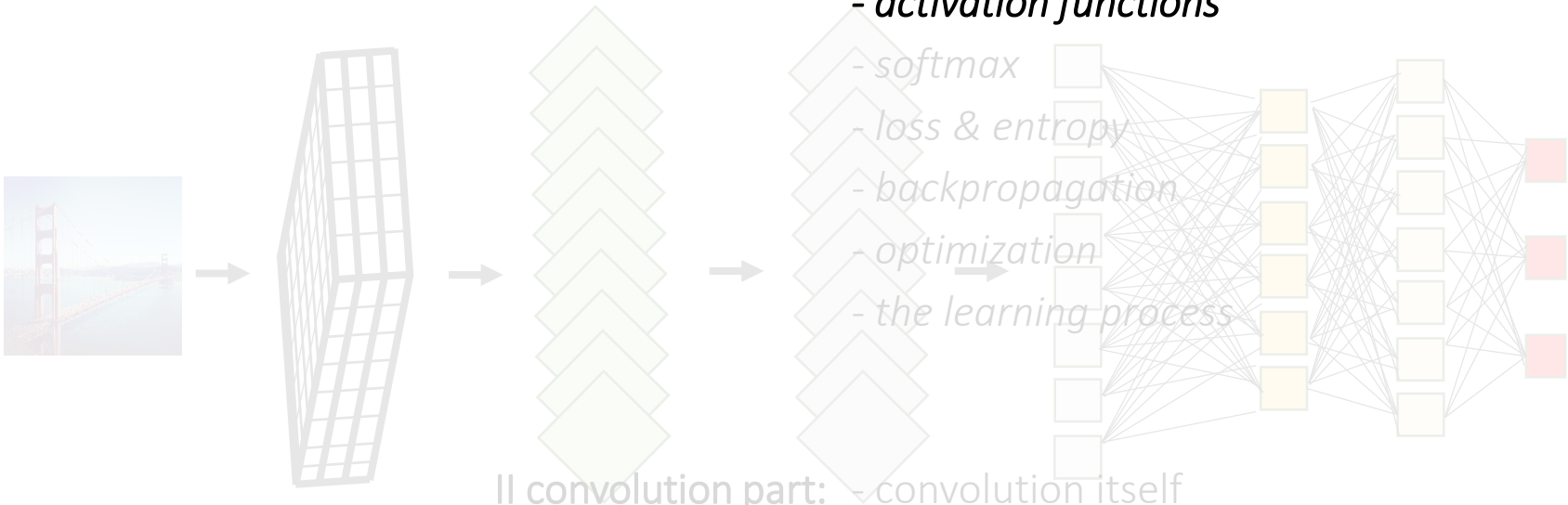
creating the actual network

outline

0 intro

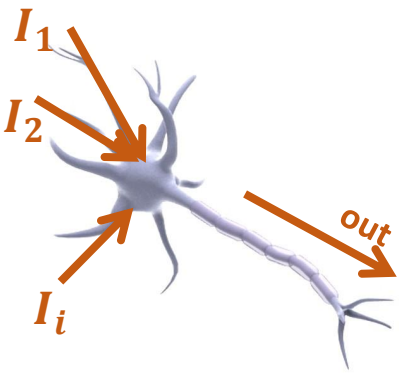
I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*



II convolution part:

- convolution itself
- max pool and average pool
- batch normalization
- flattening
- learning and “teaching”
- final remarks

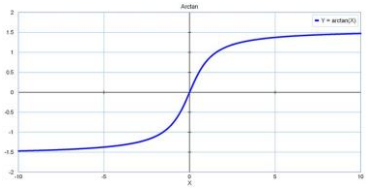


I_i	**input *i***
w_i	**corresponding weight**
b	**bias (base potential)**

$$\boxed{o = \underbrace{\sum_i I_i \cdot w_i}_{\textbf{\textit{dot product}}} + b}$$

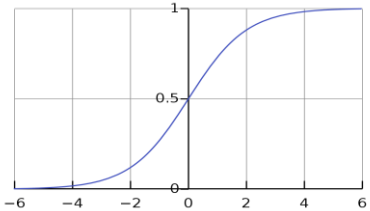
o → some activation function → actual output

- y = arctan(o)



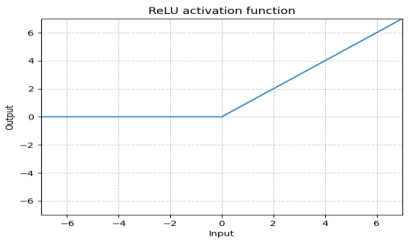
$(-\infty;\ +\infty) \rightarrow (-\pi/2;\ +\pi/2)$

- y = sigm(o)



$(-\infty;\ +\infty) \rightarrow (0; 1)$

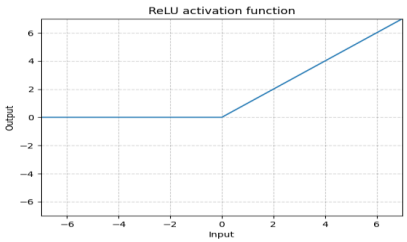
- y = ReLU(o)
= *max(0, o)*



$(-\infty;\ +\infty) \rightarrow (0; +\infty)$

o → some activation function → actual output

y = ReLU(o)
= *max(0, o)*



$(-\infty;\ +\infty) \rightarrow (0; +\infty)$

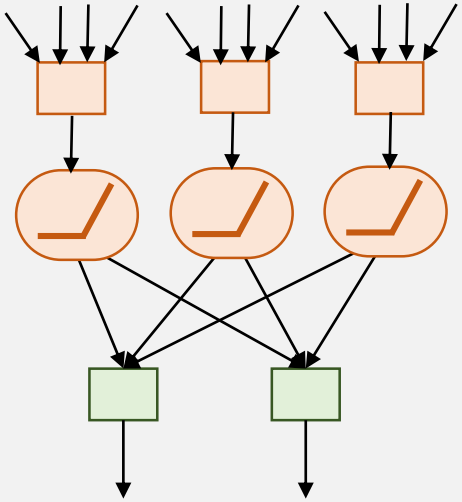
<u>same idea as before:</u>

```
class Activation_ReLU:

    def forward(self,inputs):
        self.output = np.maximum(0,inputs)
```

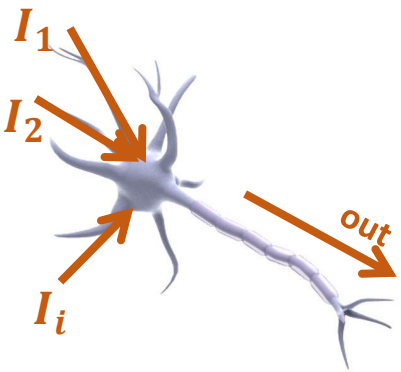
```
I      = [[1,3,4,5], [-2, -5, -6, 0]]
S      = np.shape(I)

n_neu1 = 3
n_neu2 = 2

dense1      = Layer_Dense(S[1], n_neu1)
dense2      = Layer_Dense(n_neu1, n_neu2)
activation1 = Activation_ReLU()
```

defining layers

```
dense1.forward(I)
activation1.forward(dense1.output)
dense2.forward(activation1.output)



print(dense1.output)
print(activation1.output)
print(dense2.output)
```



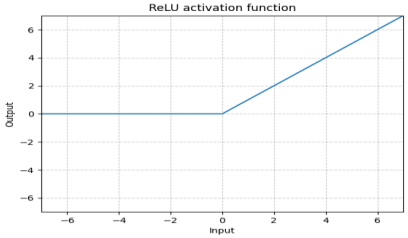
creating the actual network



I_i	**input *i***
w_i	**corresponding weight**
b	**bias (base potential)**

$$\boxed{o = \underbrace{\sum_i I_i \cdot w_i}_{\textbf{\textit{dot product}}} + b}$$

- y = ReLu(o)
  = *max(0, o)*



$$(-\infty;\ +\infty) \rightarrow (0; +\infty)$$

<u>we can now create an arbitrarily complex network with any number of neurons</u>

→ the network is still **not able to learn**

→ for that, we need a criteria, an ***objective function***, to ***optimize***

→ for that, we need to compare the target output to the actual output

(already ok for regression)

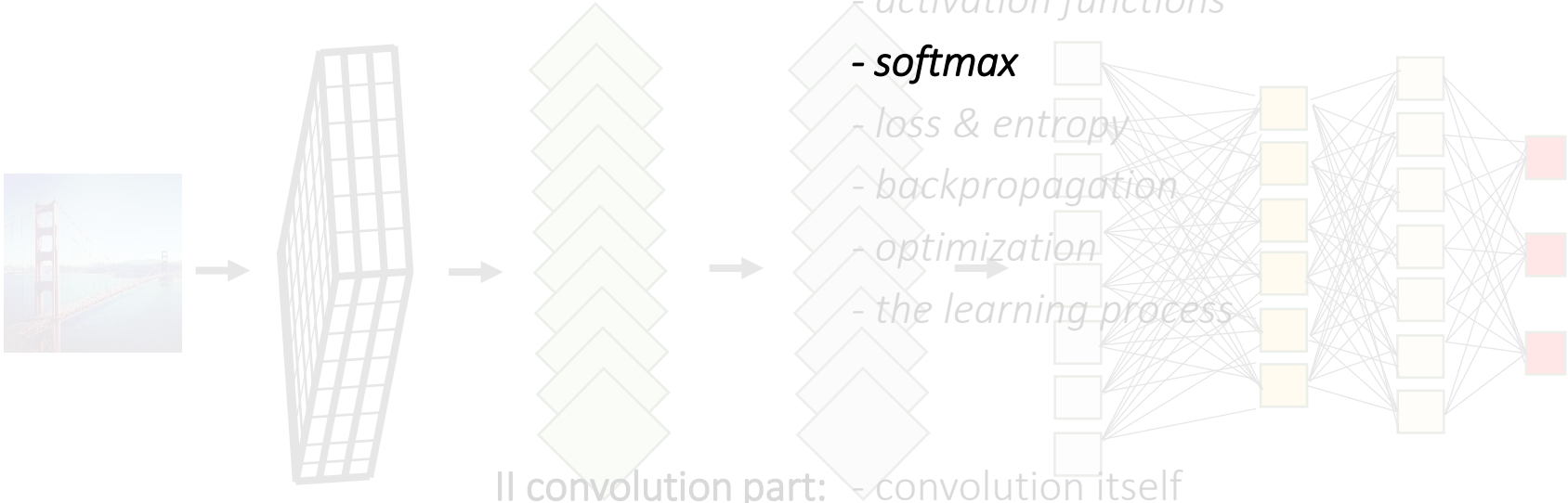
→ for that we need a layer assigning **probabilities for classification**

outline

0 intro

I the core:

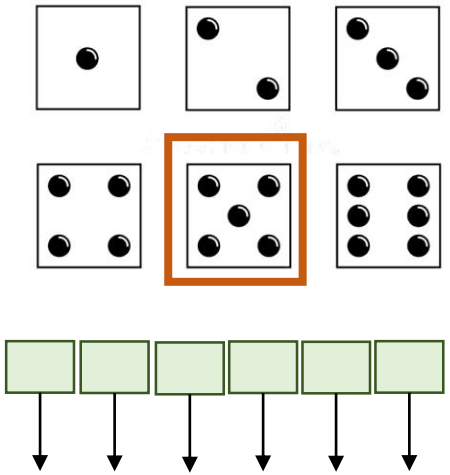
- *a single neuron*
- *layers of neurons*
- *activation functions*
- ***softmax***
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*

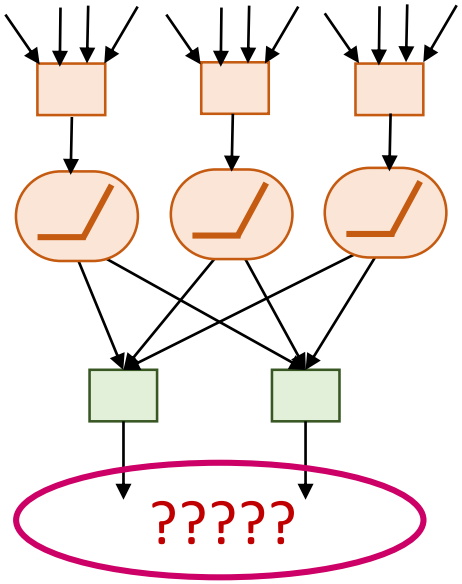


II convolution part:

- convolution itself
- max pool and average pool
- batch normalization
- flattening
- learning and “teaching”
- final remarks

<u>problem:</u> how to assign probabilities to outputs?





ideal world $p_i =$ 0 0 0 0 1 0 probability p_i for each state *i*

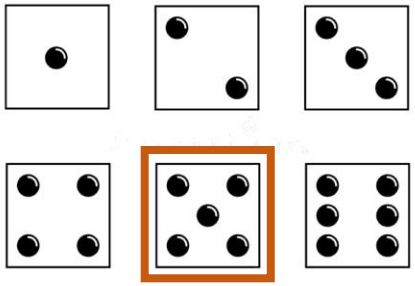
real world $p_i =$ 0.05 0.1 0 0.05 0.7 0.1

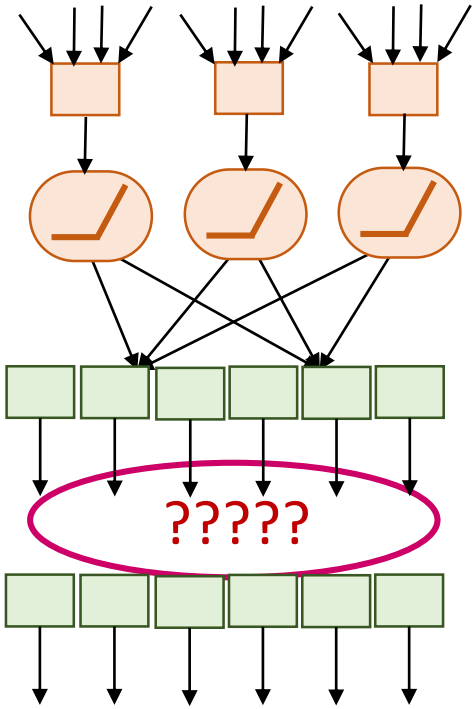
→ entropy *S* is a good measure of the categorization quality (see next chapter)

$$\boxed{S = -\sum_i p_i \cdot \ln p_i}$$

<u>problem:</u> how to assign probabilities to outputs?

$$S = -\sum_i p_i \cdot \ln p_i$$





$p_i =$ 0.05 0.1 0 0.05 0.7 0.1

each output ε_i **(that ranges from zero to +∞)**
has to be mapped to p_i **(that ranges from zero to 1)**

→ $p_i = p_i(\varepsilon_i) = ?$

we know that: $\sum_i p_i = 1$
thus $\sum_i p_i \varepsilon_i = \langle \varepsilon \rangle$
where ε is a positive number

→ **maximizing entropy** given these two constrains

we know from stat physics, that this leads to the ***Boltzmann distribution*** **(see Lagrangian multipliers)**

$$\left.\begin{array}{l} p_i \sim \exp(\beta \varepsilon_i) \\ \\ \sum_i p_i = 1 \end{array}\right\} \quad p_i = \frac{\exp(\beta \varepsilon_i)}{\sum_i \exp(\beta \varepsilon_i)}$$

<u>problem:</u> how to assign probabilities to outputs?

Boltzmann distribution

partition function
$\mathcal{Z}$ (normalization factor)

$$\boxed{p_i = \frac{\exp(\beta\varepsilon_i)}{\sum_i \exp(\beta\varepsilon_i)}}$$

β unknown constant → depending on context:

if ε_i is interpreted as energy
→ β = *-1/T* with temperature *T*
(see reinforced learning)

here: $\beta = 1$ **(we don't have physical temperature → unit system is arbitrary)**

<u>note:</u> it's called ***softmax***, as contrast to a "hard max" like *argmax*(ε_i)

<u>problem:</u> how to assign probabilities to outputs?

Boltzmann distribution

$$\boxed{p_i = \frac{\exp(\varepsilon_i)}{\sum_i \exp(\varepsilon_i)}}$$

```
class Activation_Softmax:

    def forward(self,inputs):
        exp_values    = np.exp(inputs - np.max(inputs))
```

adjusting in order to prevent overflow

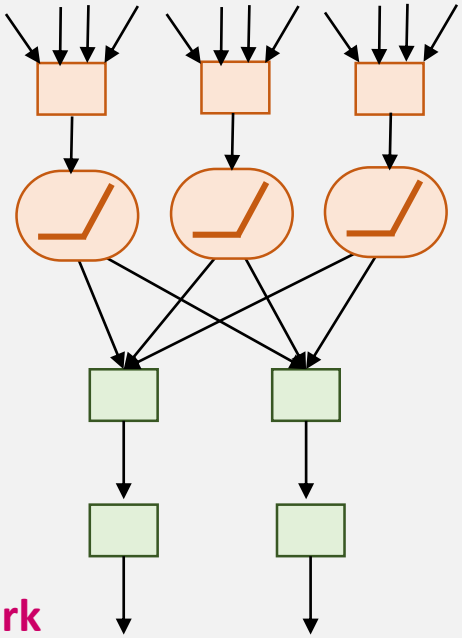
```
        probabilities = exp_values/np.sum(exp_values, axis = 1,\
                                          keepdims = True)
        self.output   = probabilities
```

→ lets add the new layer to the network

```
I       = [[1,3,4,5], [-2, -5, -6, 0]]
S       = np.shape(I)

n_neu1 = 3
n_neu2 = 2

dense1      = Layer_Dense(S[1], n_neu1)
dense2      = Layer_Dense(n_neu1, n_neu2)
activation1 = Activation_ReLU()
activation2 = Activation_Softmax()
```

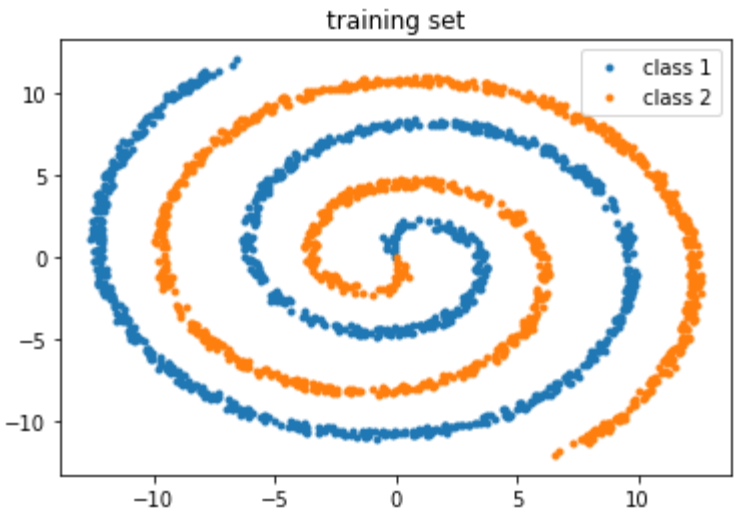
defining layers

```
dense1.forward(I)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
activation2.forward(dense2.output)



print(activation2.output)
print(np.sum(activation2.output,1))
```



creating the actual network

the whole thing becomes more serious now → let us load a toy data set



spiral data set

→ Spyder command prompt
→ type: `pip install nnfs`

→ in Spyder
→ type: `from nnfs.datasets import spiral_data`

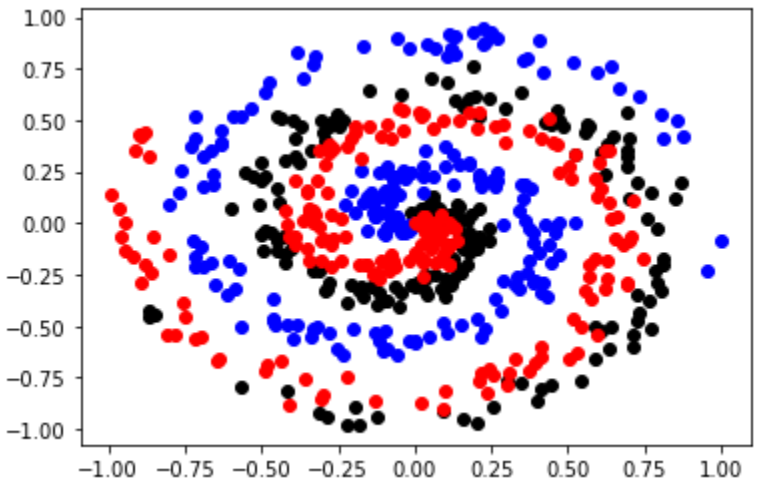
```
[x, y] = spiral_data(samples = 200, classes = 3)
```

y: vector of classes, check:
`print(np.unique(y))`

x: vector of **<u>200 x 3</u>** data points with two features,
→ equivalent to **I** before, check:
`print(x[0:5,:])`

the whole thing becomes more serious now → let us load a toy data set

```
import matplotlib.pyplot as plt
```




```
idx0 = np.argwhere(y==0)
idx1 = np.argwhere(y==1)
idx2 = np.argwhere(y==2)
```

```
plt.scatter(x[idx0,0],x[idx0,1], c = 'black')
plt.scatter(x[idx1,0],x[idx1,1], c = 'blue')
plt.scatter(x[idx2,0],x[idx2,1], c = 'red')
```

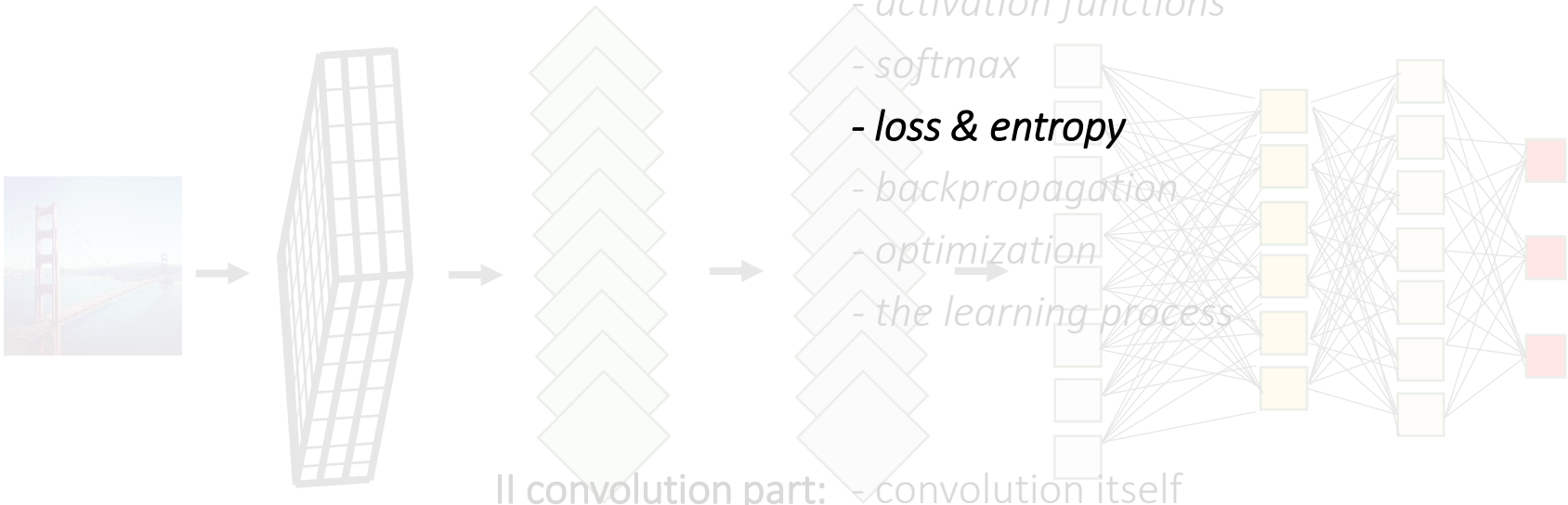
→ spiral shaped data is usually hard to analyze with common machine learning methods
→ ANN

outline

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*



II convolution part:

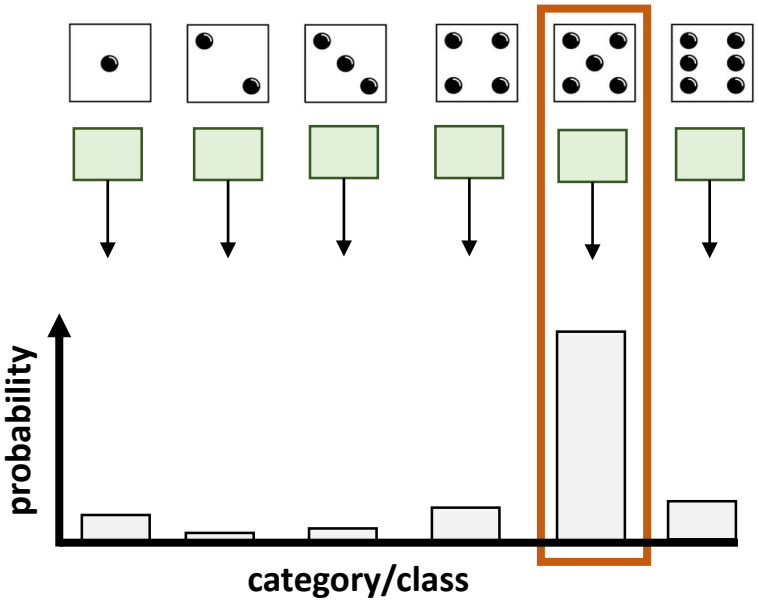
- convolution itself
- max pool and average pool
- batch normalization
- flattening
- learning and “teaching”
- final remarks

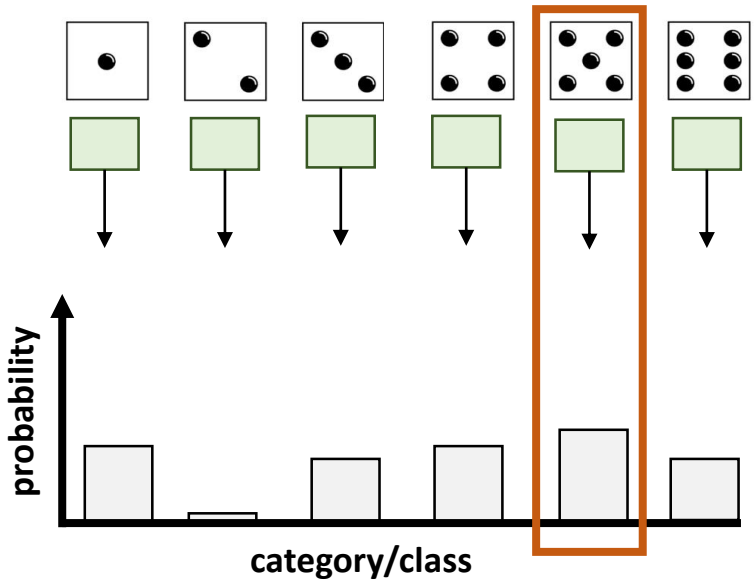
y : the true class
softmax → predicts the learned class

} we need to compare both quantities in order to determine the quality of our ANN

<u>there are two things we are interested in:</u>

1) How *confident* is the ANN about its decision?





$S \approx 0$ ☺

$$S = -\sum_i p_i \cdot \ln p_i$$

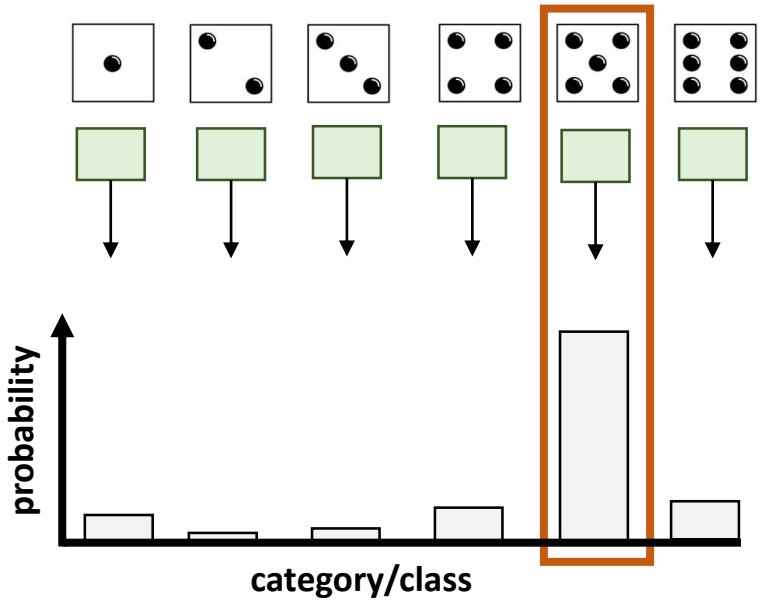
☹ $S \approx \ln(number\ of\ states)$

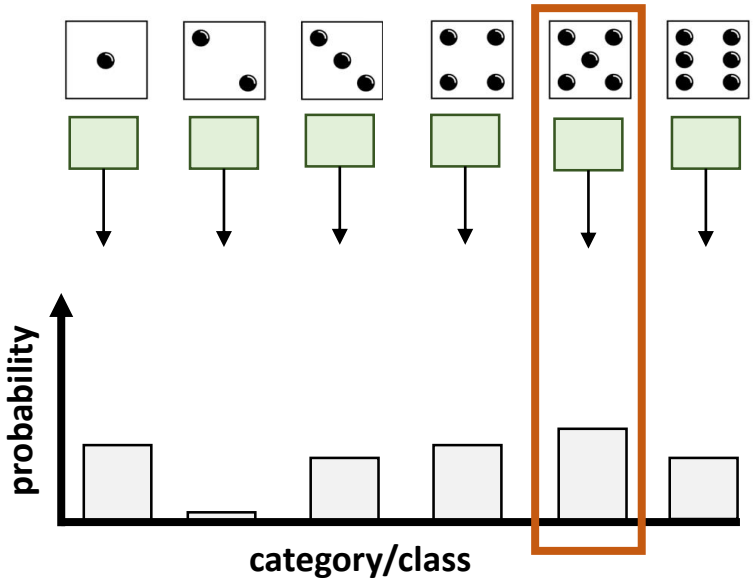
y : the true class
softmax → predicts the learned class

} we need to compare both quantities in order to determine the quality of our ANN

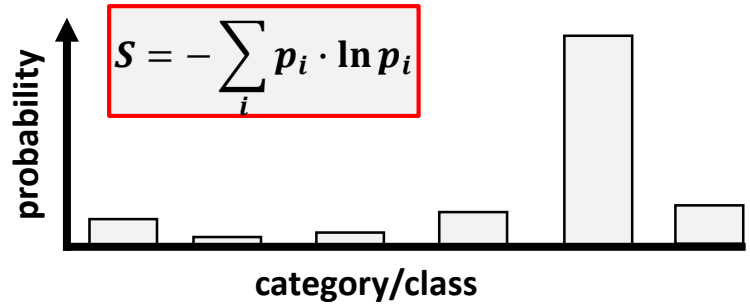
<u>there are two things we are interested in:</u>

2) How *often* did the ANN make the correct decision?

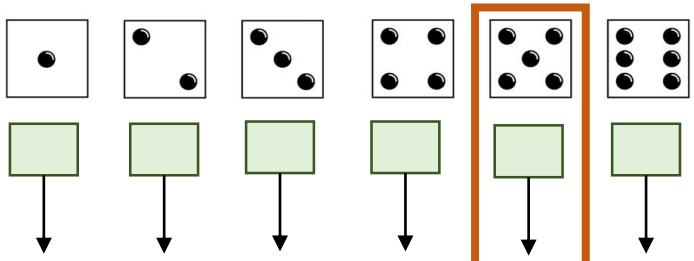




1) How *confident* is the ANN about its decision?



for each data point (after `softmax` layer)



$$p_i = 0.1 \quad 0.05 \quad 0.2 \quad 0.0 \quad 0.45 \quad 0.2$$

$$p\_true_i = 0 \quad 0 \quad 0 \quad 0 \quad 1 \quad 0$$

$$S = -\sum_i p\_true_i \cdot \ln p_i$$

calculating ***cross entropy*** for each sample

mean over all samples → total *Loss*

→ for categorization: mean of cross entropy

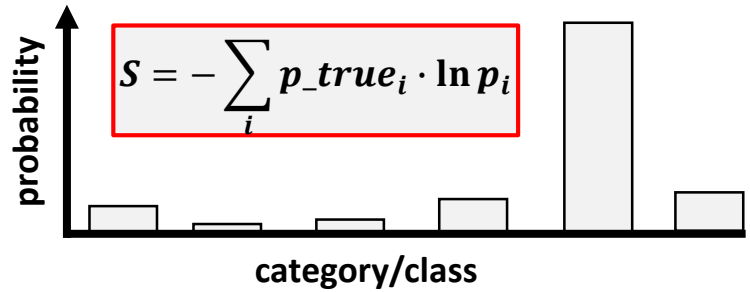
→ for regression: mean over RMSE

→ we'd like to keep the actual loss calculation separate from the total loss calculation

(in fact: we can turn most CNNs into regression networks by just removing the softmax layer and the cross entropy part)

1) How *confident* is the ANN about its decision?

mean over all samples → total *Loss*




```
class Loss:

        def calculate(self, output, y):
```

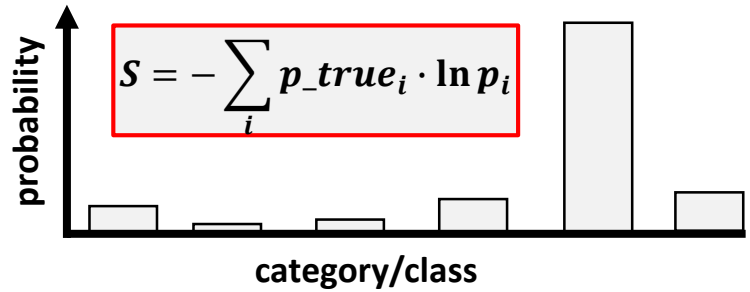
output here refers to the output from the softmax layer

```
            sample_losses = self.forward(output, y)
```

we are going to call *Loss* within another class later

```
            data_loss     = np.mean(sample_losses)

            return(data_loss)
```

1) How *confident* is the ANN about its decision?



now: calculating the cross entropy:

`-np.log`
- will cause numerical issues for $p_i \approx 0$
- $p_i = 1$ not possible for numerical reasons (see `softmax`)
  → would be a numerical artefact

```
y_pred_clipped = np.clip(prob_pred, 1e-7, 1 - 1e-7)
```

the true y could be encoded in different ways → must be taken into account when calculating cross entropy

"one hot"

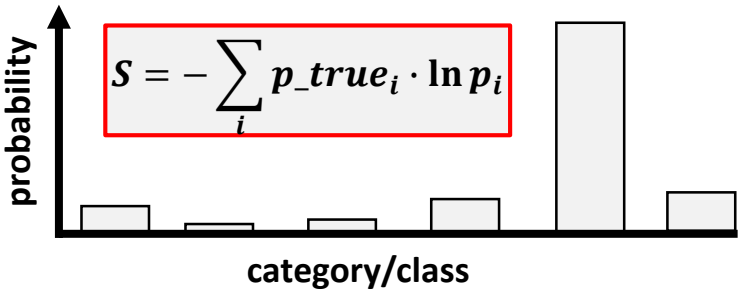
$$\begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

"sparse"

$$\begin{bmatrix} 1 \\ 1 \\ 3 \\ 2 \end{bmatrix}$$

1) How *confident* is the ANN about its decision?

now: calculating the cross entropy:



category/class →

$$\texttt{prob\_pred} = \begin{pmatrix} 0.8 & 0.1 & 0.1 \\ 0.9 & 0.05 & 0.05 \\ 0.1 & 0.15 & 0.75 \\ 0.2 & 0.7 & 0.1 \\ \dots & \dots & \dots \end{pmatrix}$$

↓ **samples**

```
Nsamples   = len(prob_pred)
```

"sparse"

$$\texttt{y\_true} = \begin{bmatrix} 1 \\ 1 \\ 3 \\ 2 \end{bmatrix}$$

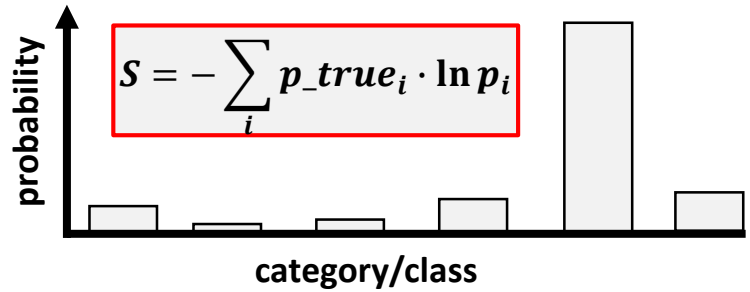
checking if sparse (==1) or one hot (==2)

```
if len(y_true.shape) == 1:
        correct_confidences = \
        y_pred_clipped[range(Nsamples),y_true]
```

filters out the correct true class

1) How *confident* is the ANN about its decision?

now: calculating the cross entropy:



category/class →

$$\texttt{prob\_pred} = \begin{pmatrix} 0.8 & 0.1 & 0.1 \\ 0.9 & 0.05 & 0.05 \\ 0.1 & 0.15 & 0.75 \\ 0.2 & 0.7 & 0.1 \\ \dots & \dots & \dots \end{pmatrix}$$

↓ **samples**

```
Nsamples   = len(prob_pred)
```

"one hot"

$$\texttt{y\_true} = \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix}$$

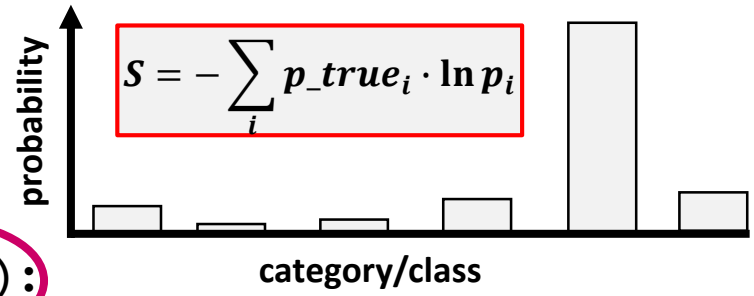
checking if sparse (==1) or one hot (==2)

```
elif len(y_true.shape) == 2:

        correct_confidences = \
        np.sum(y_pred_clipped*y_true, axis = 1)
```

either ones or zeros → only correct true class contributes (axis = 1, summing across the columns)

1) How *confident* is the ANN about its decision?

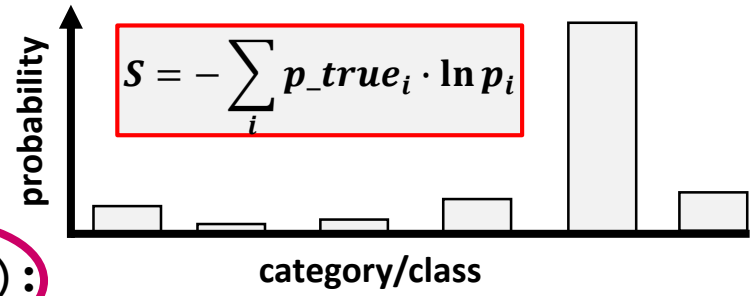
now: calculating the cross entropy:




```
class Loss_CategoricalCrossEntropy(Loss):

    def forward(self, prob_pred, y_true):

        samples        = len(prob_pred)
        y_pred_clipped = np.clip(prob_pred, 1e-7, 1 - 1e-7)

        if len(y_true.shape) == 1
                  correct_confidences = \
                  y_pred_clipped[range(Nsamples),y_true]

        elif len(y_true.shape) == 2:
              correct_confidences = \
              np.sum(y_pred_clipped*y_true, axis = 1)

    negative_log_likelihoods = -np.log(correct_confidences)

    return(negative_log_likelihoods)
```

1) How *confident* is the ANN about its decision?

now: calculating the cross entropy:




```
class Loss_CategoricalCrossEntropy(Loss):

    def forward(self, prob_pred, y_true):
```

= forward in *Loss*, returns negative_log_likelihoods
that's called *sample_losses* in *Loss*. The *calculate* method has been inherited.

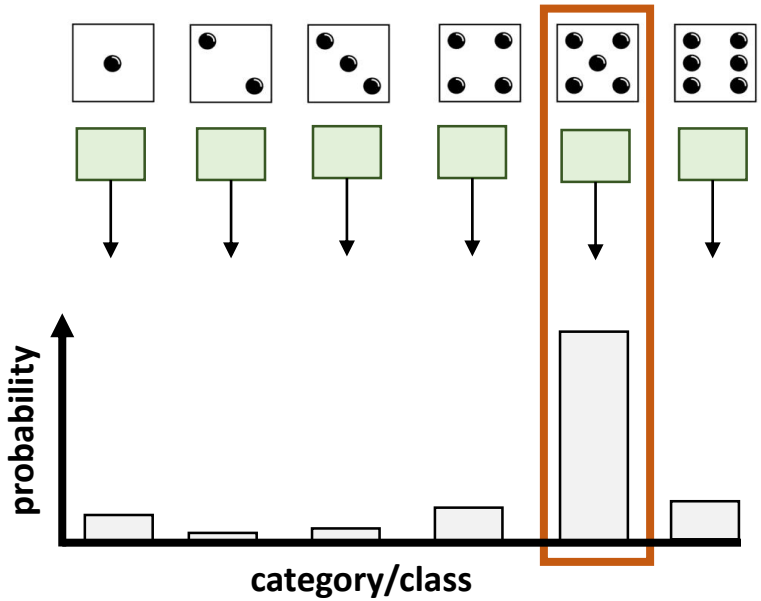
```
        Nsamples        = len(prob_pred)
        y_pred_clipped = np.clip(prob_pred, 1e-7, 1 - 1e-7)

        if len(y_true.shape) == 1:
                  correct_confidences = \
                    y_pred_clipped[range(Nsamples),y_true]

        elif len(y_true.shape) == 2:
                  correct_confidences = \
                  np.sum(y_pred_clipped*y_true, axis = 1)

    negative_log_likelihoods = -np.log(correct_confidences)

    return(negative_log_likelihoods)
```

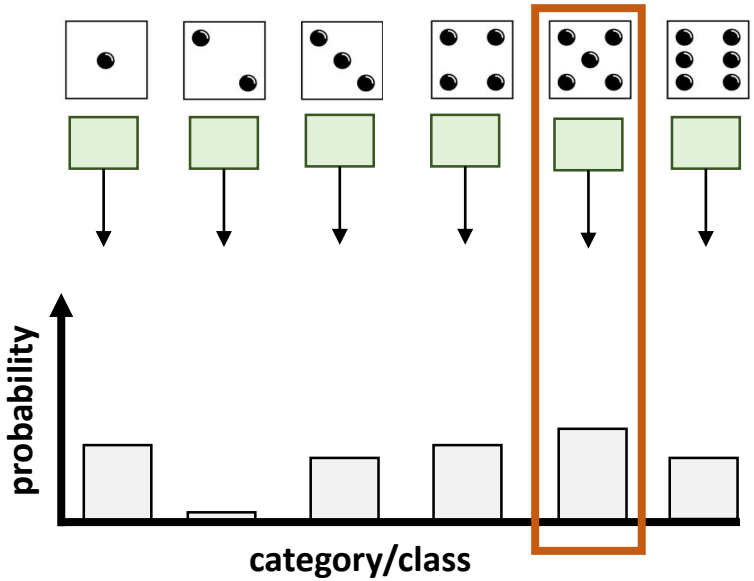
y : the true class
softmax → predicts the learned class

} we need to compare both quantities in order to determine the quality of our ANN

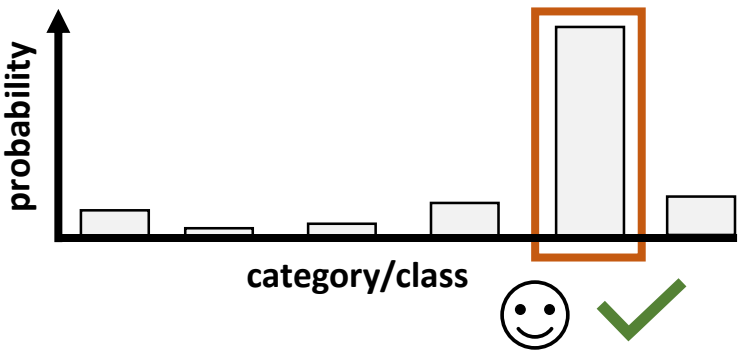
<u>there are two things we are interested in:</u>

2) How *often* did the ANN make the correct decision?





2) How *often* did the ANN make the correct decision?



category/class →

$$\begin{pmatrix} 0.8 & 0.1 & 0.1 \\ 0.9 & 0.05 & 0.05 \\ 0.1 & 0.15 & 0.75 \\ 0.2 & 0.7 & 0.1 \\ \dots & \dots & \dots \end{pmatrix}$$

↓ **samples**

output from softmax

```
predictions = np.argmax(activation2.output, axis = 1)
```

works already if y is sparse

```
if len(y.shape) == 2:
        y = np.argmax(y,axis = 1)
```

in case if y is one-hot

```
accuracy = np.mean(predictions == y)
```

```
nc      = 3
[x, y] = spiral_data(samples = 200, classes = nc)
S       = np.shape(x)

dense1          = Layer_Dense(S[1],4 )
dense2          = Layer_Dense(4 , nc)
activation1     = Activation_ReLU()
activation2     = Activation_Softmax()
loss_function = Loss_CategoricalCrossEntropy()
```

defining layers

```
dense1.forward(x)
activation1.forward(dense1.output)

dense2.forward(activation1.output)
activation2.forward(dense2.output)

loss = loss_function.calculate(activation2.output,y)

predictions = np.argmax(activation2.output, axis = 1)
    if len(y.shape) == 2:
        y = np.argmax(y,axis = 1)
accuracy = np.mean(predictions == y)
```

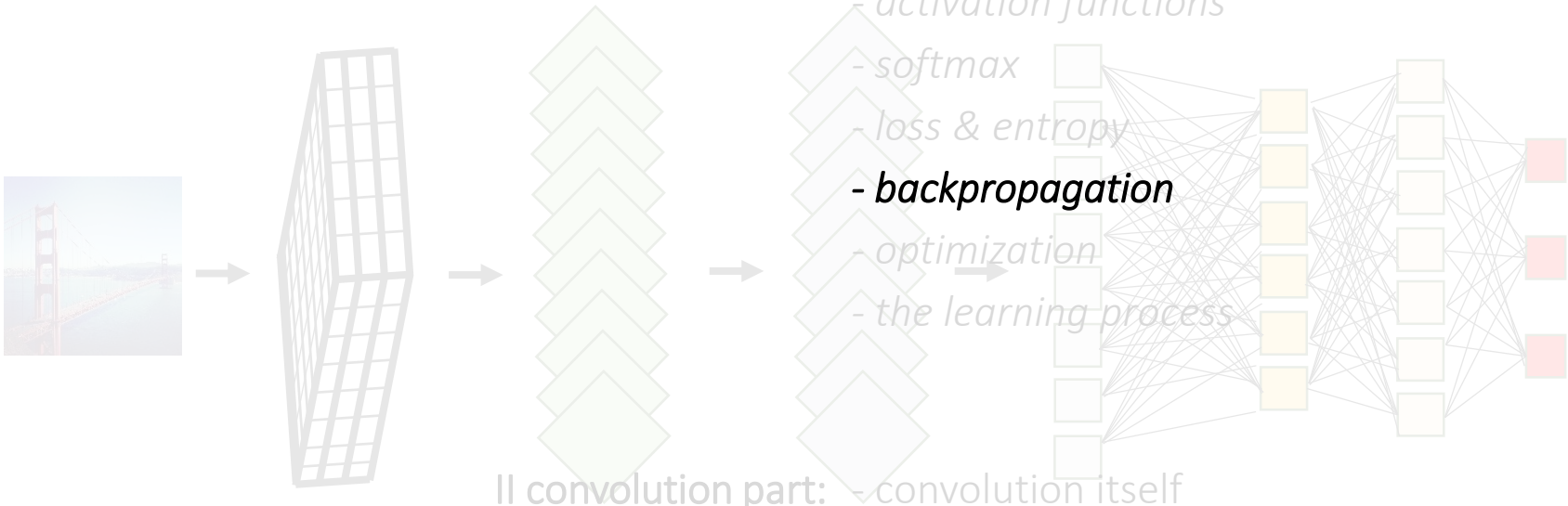
creating the actual network

outline

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- ***backpropagation***
- *optimization*
- *the learning process*



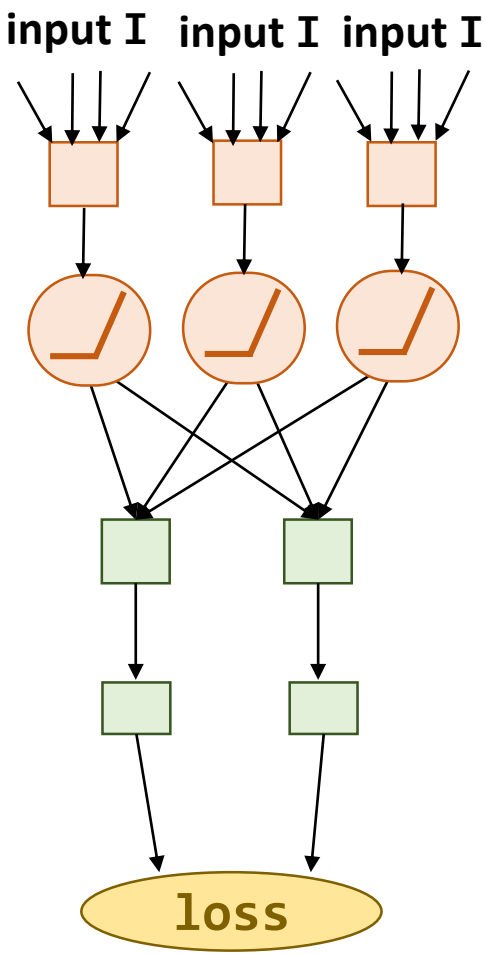
II convolution part:

- convolution itself
- max pool and average pool
- batch normalization
- flattening
- learning and “teaching”
- final remarks

We have now an objective mathematical criterium of the classification quality of our ANN:

→ the **`loss`**

→ aim: minimizing the `loss`



$$\sum_i I_i\, w_i + b$$

sum in a single neuron

$$A\left(\sum_i I_i\, w_i + b\right)$$

activation

$$\sum_j A\left(\sum_i I_i\, w_i + b\right)_j \omega_j + \beta$$

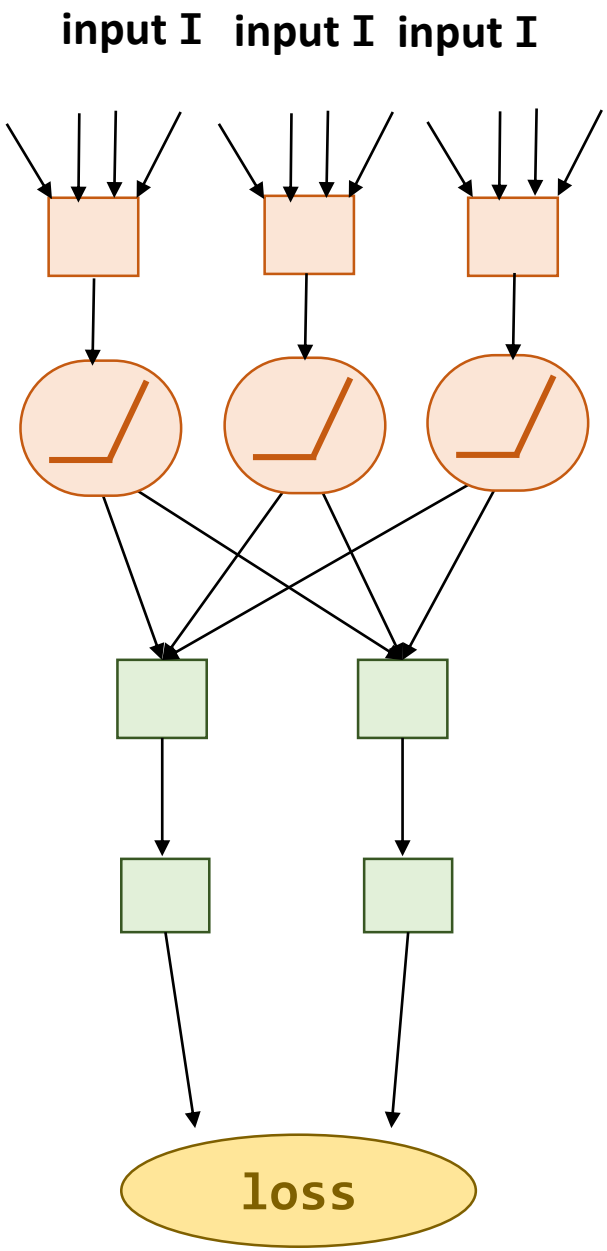
input of the next layer

$$p_m = \frac{\exp([\sum_j A(\sum_i I_i\, w_i + b)_j\, \omega_j + \beta\,]_k)}{\sum_k \exp([\sum_j A\left(\sum_i I_i\, w_i + b\right)_j \omega_j + \beta\,]_k)}$$

softmax

$$L_m = -\log(p_m)$$

loss calculation



forward pass

backpropagation

sum in a single neuron

$$\sum_i I_i\, w_i + b$$

$$\frac{\partial \sum_i I_i\, w_i + b}{\partial w_i}$$

$$\frac{\partial A}{\partial \sum_i I_i\, w_i + b}$$

activation

$$A\left(\sum_i I_i\, w_i + b\right)$$

$$\frac{\partial \sum_j \zeta_j\, \omega_j + \beta}{\partial \zeta_j}$$

$$\frac{\partial \sum_j \zeta_j\, \omega_j + \beta}{\partial \omega_j}$$

input of the next layer

$$\sum_j \underbrace{A\left(\sum_i I_i\, w_i + b\right)_j}_{\zeta_j} \omega_j + \beta$$

$$\frac{\partial p_m}{\partial \varepsilon_l}$$

softmax

$$p_m = \frac{\exp([\sum_j A(\sum_i I_i\, w_i + b)_j\, \omega_j + \beta\,]_k)}{\sum_k \exp([\underbrace{\sum_j A\left(\sum_i I_i\, w_i + b\right)_j \omega_j + \beta}_{\epsilon_k}\,]_k)}$$

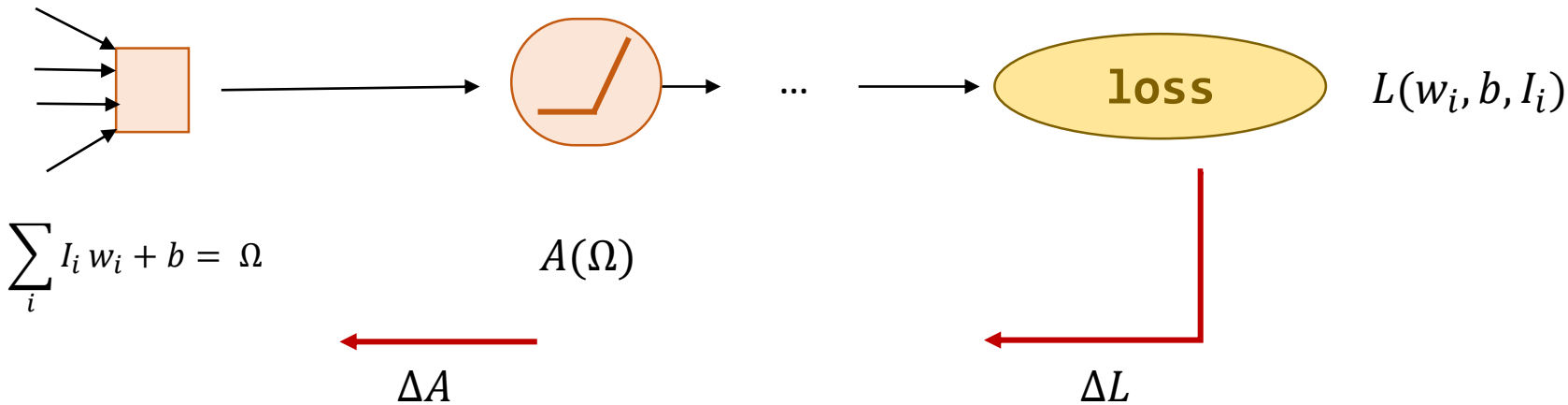
loss calculation

$$L_m = -\log(p_m)$$

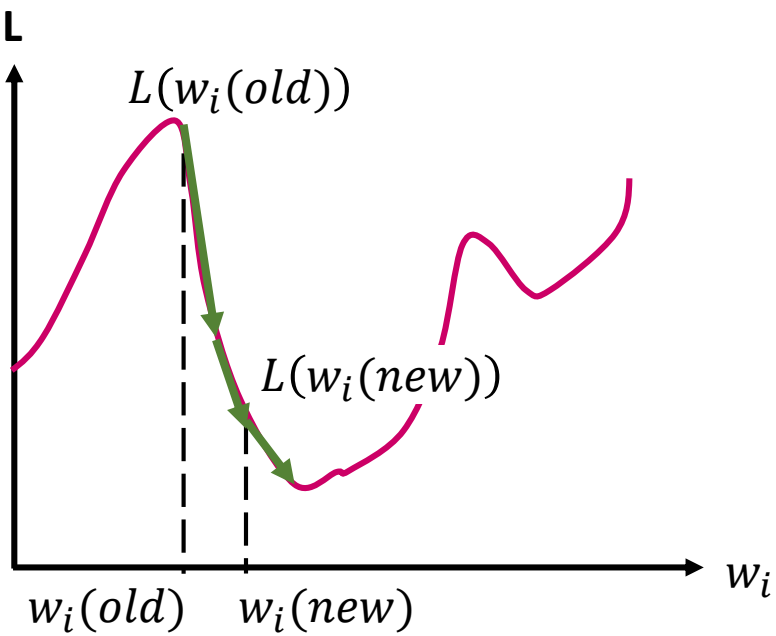
$$\frac{\partial L_m}{\partial p_m}$$

→ aim: minimizing the loss

Ok, that looks complicated, so let's understand it conceptually:



aim

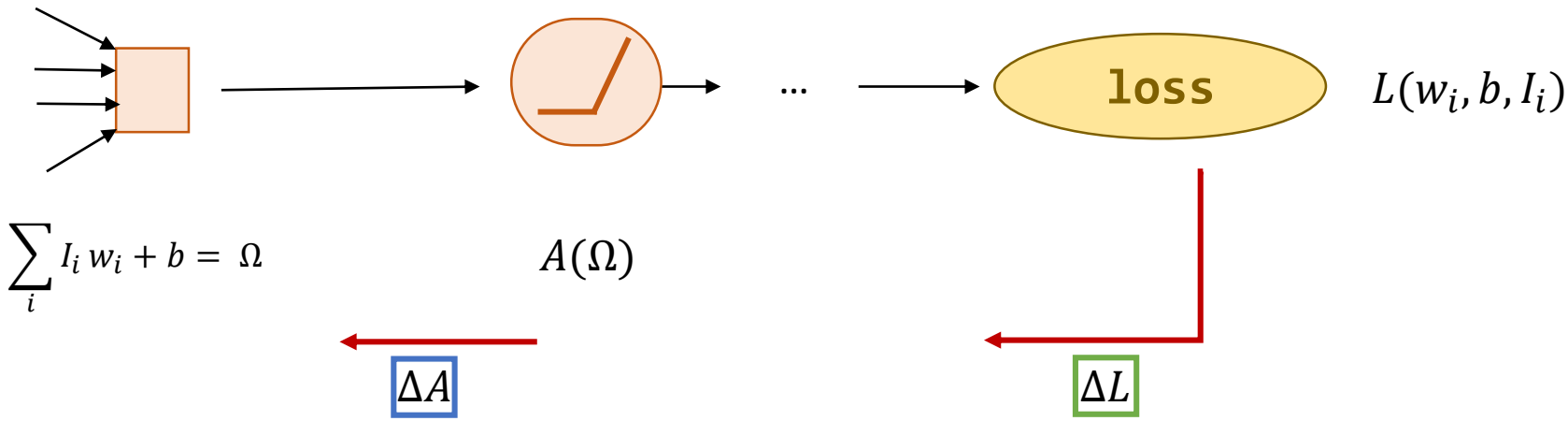


(gradient descent, ε =const; see later)

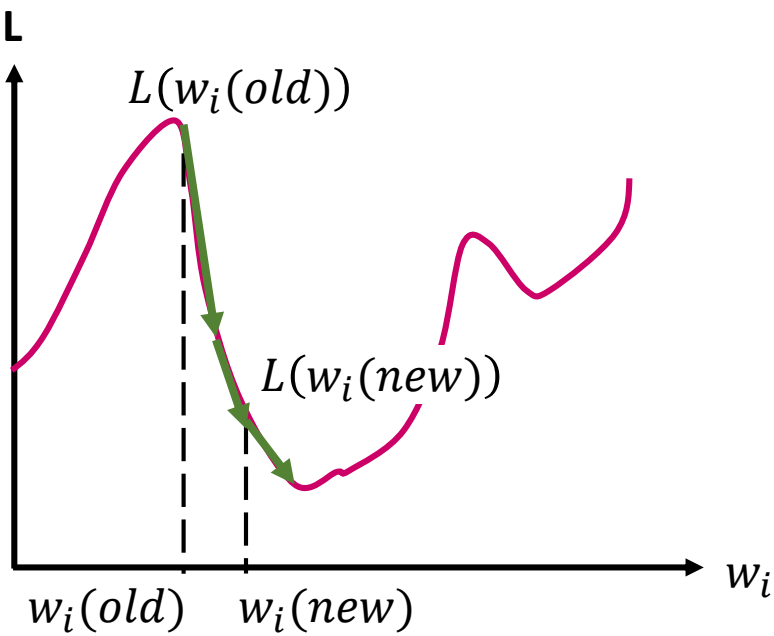
$$w_i(new) = w_i(old) - \varepsilon \frac{\partial L}{\partial w_i}$$

$$\Delta w_i = -\varepsilon \frac{\partial L}{\partial w_i}$$

Ok, that looks complicated, so let's understand it conceptually:



aim

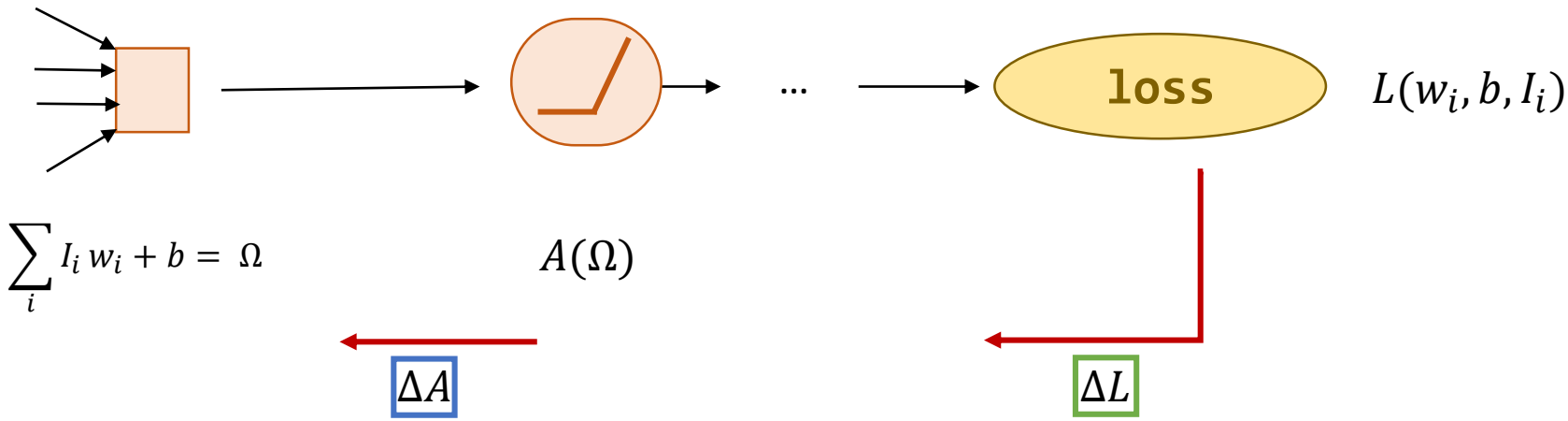


$$\Delta w_i = -\varepsilon \frac{\partial L}{\partial w_i}$$

$$\Delta w_i = -\varepsilon \frac{dL}{dA}\frac{dA}{d\Omega}\frac{\partial \Omega}{\partial w_i}$$

$$\Delta w_i = -\varepsilon \boxed{\frac{dL}{dA}}\boxed{\frac{dA}{d\Omega}} I_i$$

Ok, that looks complicated, so let's understand it conceptually:



$$\Delta w_i = -\varepsilon \frac{dL}{dA}\frac{dA}{d\Omega} I_i$$

$$\Delta I_i = -\varepsilon \frac{dL}{dA}\frac{dA}{d\Omega} w_i \quad \text{from} \quad \frac{\partial L}{\partial I_i}$$

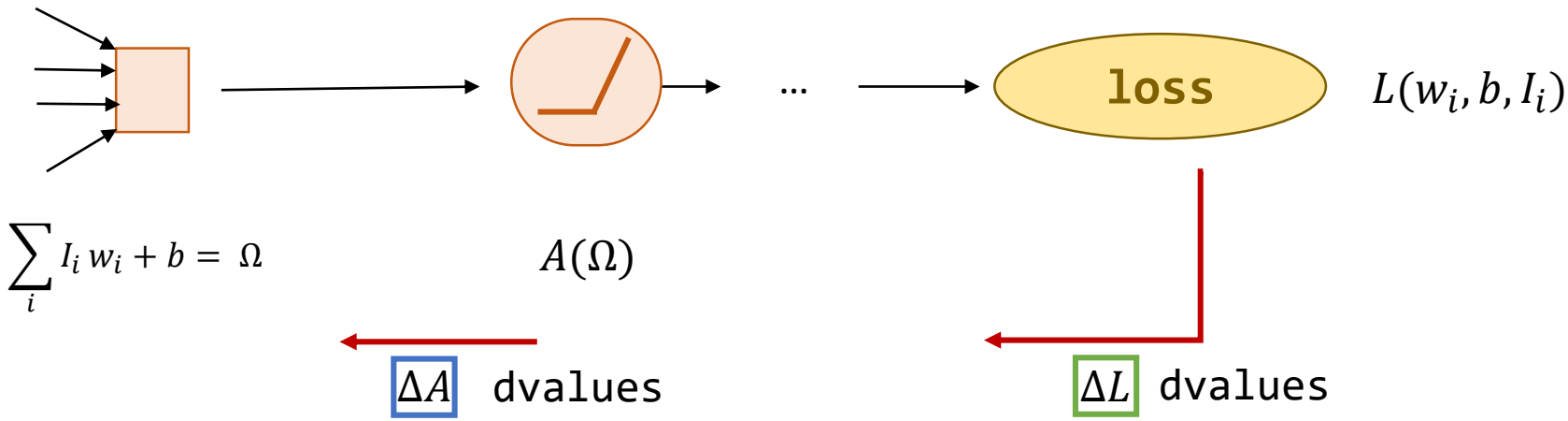
$$\Delta b = -\varepsilon \frac{dL}{dA}\frac{dA}{d\Omega} \quad \text{from} \quad \frac{\partial L}{\partial b}$$

feedback from lower layer to next upper layer of the form Δ*something*

→ `dvalues`

derivatives factorize through layers

Ok, that looks complicated, so let's understand it conceptually:



$$\Delta w_i = -\varepsilon \boxed{\frac{dL}{dA}} \boxed{\frac{dA}{d\Omega}} I_i$$

$$\Delta I_i = -\varepsilon \boxed{\frac{dL}{dA}} \boxed{\frac{dA}{d\Omega}} w_i$$

$$\Delta b = -\varepsilon \boxed{\frac{dL}{dA}} \boxed{\frac{dA}{d\Omega}}$$

hence, we need to include the following structure for backpropagation:

```
self.dweights  = inputs  * dvalues
self.dinputs   = weights * dvalues
self.dbiases   = 1       * dvalues
```

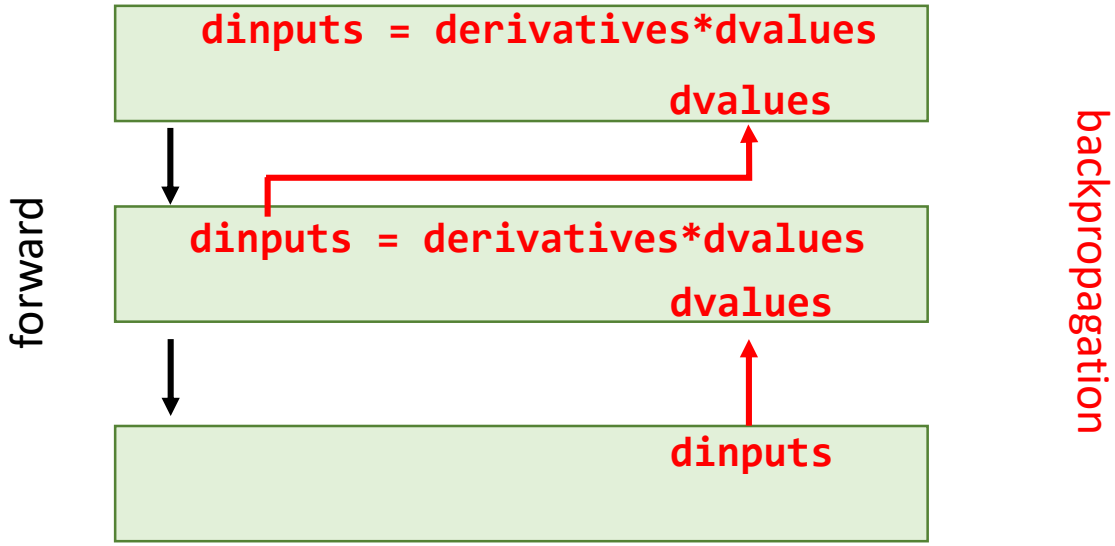
Apparently we need to apply the chain rule for derivatives throughout the network!

→ adding the method **`backward`** for the derivatives in each layer

→ we need to remember what the inputs were in **`Layer_Dense`** for $\frac{\partial I_i}{\partial w_i}$

→ we are passing on the derivatives from the lower layer to the next upper layer as **`dvalues`**

→ these derivatives are called **`dinputs`** in the lower layer



Apparently we need to apply the chain rule for derivatives throughout the network!

→ adding the method **backward** for the derivatives in each layer

→ we need to remember what the inputs were in **Layer_Dense** for $\frac{\partial I_i}{\partial w_i}$

→ we are passing on the derivatives from the next layer as **dvalues**

```
class Layer_Dense:

        def __init__(self, n_inputs, n_neurons):
                                ...

        def forward(self, inputs):
                self.output  = np.dot(inputs, self.weights) + self.biases
                self.inputs  = inputs
```

```
def backward(self, dvalues):

        gradient wrt the weights → inputs are left
        self.dweights  = np.dot(self.inputs.T,dvalues)
        gradient wrt the inputs → weights are left
        self.dinputs   = np.dot(dvalues, self.weights.T)
        gradient wrt the biases → just ones are left
        self.dbiases   = np.sum(dvalues, axis = 0, keepdims = True)
```

```
class Layer_Dense:

    def __init__(self, n_inputs, n_neurons):
        self.weights = np.random.rand(n_inputs, n_neurons)
        self.biases  = np.zeros((1, n_neurons))

    def forward(self, inputs):

        self.output  = np.dot(inputs, self.weights) + self.biases
        self.inputs  = inputs

    def backward(self, dvalues):

        self.dweights  = np.dot(self.inputs.T,dvalues)
        self.dinputs   = np.dot(dvalues, self.weights.T)
        self.dbiases   = np.sum(dvalues, axis = 0, keepdims = True)
```

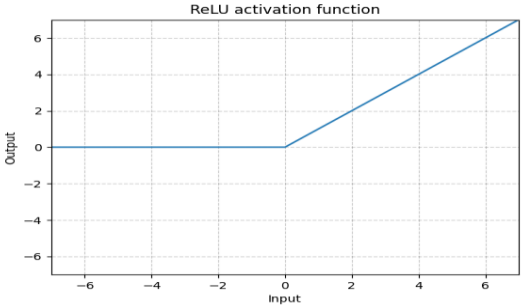
we are going to pass as dvalues for the previous layer

same procedure for the ReLU layer:

```
class Activation_ReLU:

        def forward(self,inputs):

                self.output = np.maximum(0,inputs)
                self.inputs = inputs

        def backward(self,dvalues):
                self.dinputs = dvalues.copy()

                self.dinputs[self.inputs <= 0] = 0
```

we are going to pass as dvalues for the previous layer

ReLU(y) = max(0, y)



same procedure for calculating cross entropy:

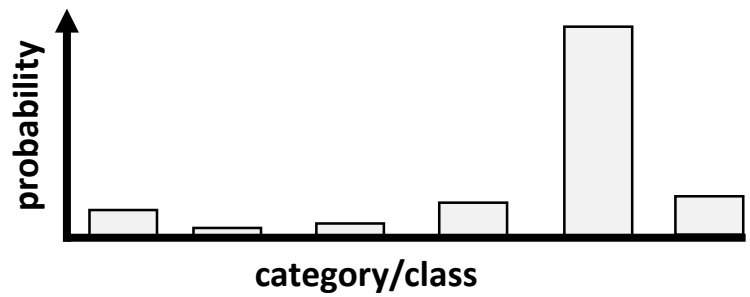
$$S = -\sum_i p\_true_i \cdot \ln p_i$$

p_true_i either zero or one (= correct class/category *i*)

p_i predicted probability of class/category *i*

for each data point *j*

$$S_j = -\sum_i p\_true_{ij} \cdot \ln p_{ij}$$



$$\frac{\partial}{\partial p_{ij}} S_j = -\frac{p\_true_{ij}}{p_{ij}}$$

`dvalues` will come back from the next layer (e. g. `softmax`)

$$grad_p(S) = \begin{pmatrix} -\frac{p\_true_{i1}}{p_{i1}} \\ -\frac{p\_true_{i2}}{p_{i2}} \\ \dots \\ -\frac{p\_true_{i2}}{p_{i2}} \\ \dots \end{pmatrix}$$

we will normalize by the sample size later
→ optimizing for the entire gradient

same procedure for calculating cross entropy:

```
class Loss_CategoricalCrossEntropy(Loss):

    def forward(self, prob_pred, y_true):
            ...


    def backward(self, dvalues, y_true):

        Nsamples = len(dvalues)

        if len(y_true.shape) == 1:
            Nlabels       = len(dvalues[0])
            y_true        = np.eye(Nlabels)[y_true]


        self.dinputs = -y_true/dvalues/Nsamples
```

turning y_true into a "one hot" structure

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

$$\frac{\partial}{\partial p_{ij}} S_j = -\frac{p\_true_{ij}}{p_{ij}}$$

same procedure for softmax:

$$\boxed{p_{ij} = \frac{\exp(\varepsilon_{ij})}{\sum_i \exp(\varepsilon_{ij})}}$$

for each data point *j* and class/category *i*

let us calculate the derivative wrt ε for a particular *i* = *k*

$$\frac{\partial}{\partial \varepsilon_{kj}} p_{ij} = \frac{\partial}{\partial \varepsilon_{kj}} \frac{\exp(\varepsilon_{ij})}{\sum_i \exp(\varepsilon_{ij})}$$

this part equals zero for i≠ *k*

$$= \boxed{\frac{\exp(\varepsilon_{kj})}{\sum_i \exp(\varepsilon_{ij})}} + \frac{\exp(\varepsilon_{ij})}{\left(\sum_i \exp(\varepsilon_{ij})\right)^2} \cdot (-1) \cdot \exp(\varepsilon_{kj})$$

$$= \frac{\exp(\varepsilon_{kj})}{\sum_i \exp(\varepsilon_{ij})} \left(1 - \frac{\exp(\varepsilon_{ij})}{\sum_i \exp(\varepsilon_{ij})}\right)$$

$$= p_{kj}(1 - p_{ij}) \qquad i = k$$

$$= -\boldsymbol{p_{kj} p_{ij}} \qquad \boldsymbol{i \neq k}$$

same procedure for softmax:

$$\boxed{p_{ij} = \frac{\exp(\varepsilon_{ij})}{\sum_i \exp(\varepsilon_{ij})}}$$

for each data point *j* and class/category *i*

we can write both cases as one equation using the ***Kronecker*** symbol

$$\delta_{ik} = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$$

$$\frac{\partial}{\partial \varepsilon_{kj}} p_{ij} = p_{kj}(1 - p_{ij}) \qquad i = k$$

$$\frac{\partial}{\partial \varepsilon_{kj}} p_{ij} = -\boldsymbol{p_{kj} p_{ij}} \qquad \boldsymbol{i \neq k}$$

$$\boxed{\frac{\partial}{\partial \varepsilon_{kj}} p_{ij} = p_{kj}\delta_{ik} - \boldsymbol{p_{kj}\, p_{ij}}}$$

remember that p_{ij} is a matrix → find a clever way how to include the ***Kronecker*** symbol in our code

same procedure for softmax:

for each data point *j* and class/category *i*

$$\delta_{ik} = \begin{cases} 1 & i = k \\ 0 & i \neq k \end{cases}$$

$$\boxed{\frac{\partial}{\partial \varepsilon_{kj}} p_{ij} = p_{kj}\delta_{ik} - \boldsymbol{p_{kj}\, p_{ij}}}$$

$$p_{ij} = \frac{\exp(\varepsilon_{ij})}{\sum_i \exp(\varepsilon_{ij})}$$

let's say our p_{ij} is `[0.3 0.6 0.1]` for one particular *j*

according to the above equations → $(number\ of\ classes)^2$ derivatives (taking one *i*, *k* goes from 1, 2, 3 … to the number of classes)

$$\left\{\frac{\partial}{\partial \varepsilon_{kj}} p_{ij}\right\}_j = \underbrace{\begin{pmatrix} 0.3 & 0 & 0 \\ 0 & 0.6 & 0 \\ 0 & 0 & 0.1 \end{pmatrix}}_{p_{kj}\delta_{ik}} - \underbrace{\begin{pmatrix} 0.09 & 0.18 & 0.03 \\ 0.18 & 0.36 & 0.06 \\ 0.03 & 0.06 & 0.01 \end{pmatrix}}_{\boldsymbol{p_{kj}p_{ij}}}$$

same procedure for softmax:

for each data point *j* and class/category *i*

```
softmax_output = [0.3 0.6 0.1]
```

$$\left\{\frac{\partial}{\partial \varepsilon_{kj}} p_{ij}\right\}_j = \underbrace{\begin{pmatrix} 0.3 & 0 & 0 \\ 0 & 0.6 & 0 \\ 0 & 0 & 0.1 \end{pmatrix}}_{p_{kj}\delta_{ik}} - \underbrace{\begin{pmatrix} 0.09 & 0.18 & 0.03 \\ 0.18 & 0.36 & 0.06 \\ 0.03 & 0.06 & 0.01 \end{pmatrix}}_{\boldsymbol{p_{kj}p_{ij}}}$$

$$\left\{\frac{\partial}{\partial \varepsilon_{kj}} p_{ij}\right\}_j$$

is called Jacobian matrix (matrix of mixed derivatives)

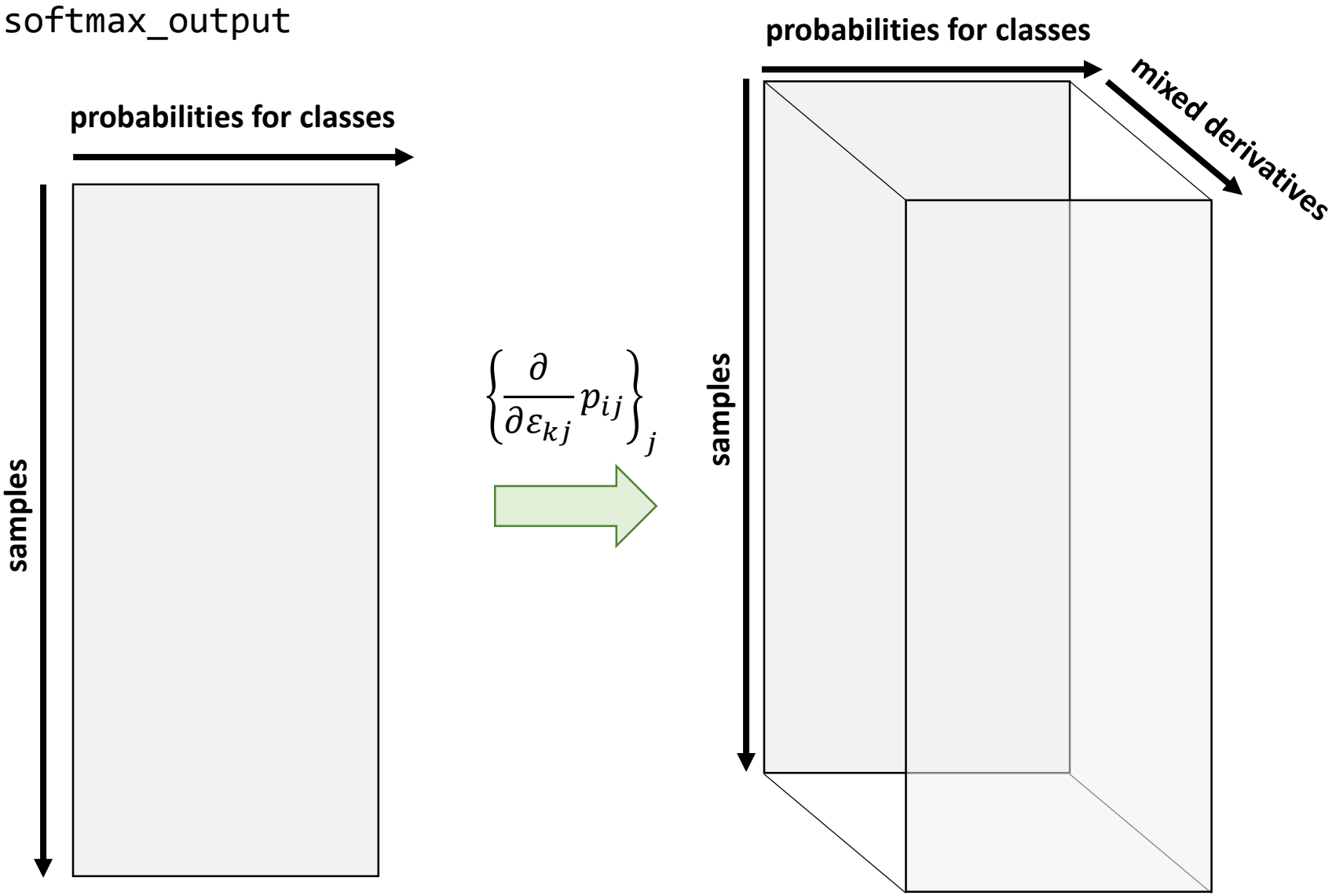
<u>in Python, this reads:</u>

```
jacobian_matrix =     np.diagflat(softmax_output) - \
                      np.dot(softmax_output, softmax_output.T)
```

Hence, for each data point *j*, we are getting such a Jacobian matrix!

Hence, for each data point *j*, we are getting such a Jacobian matrix!

for each data point *j* and class/category *i*

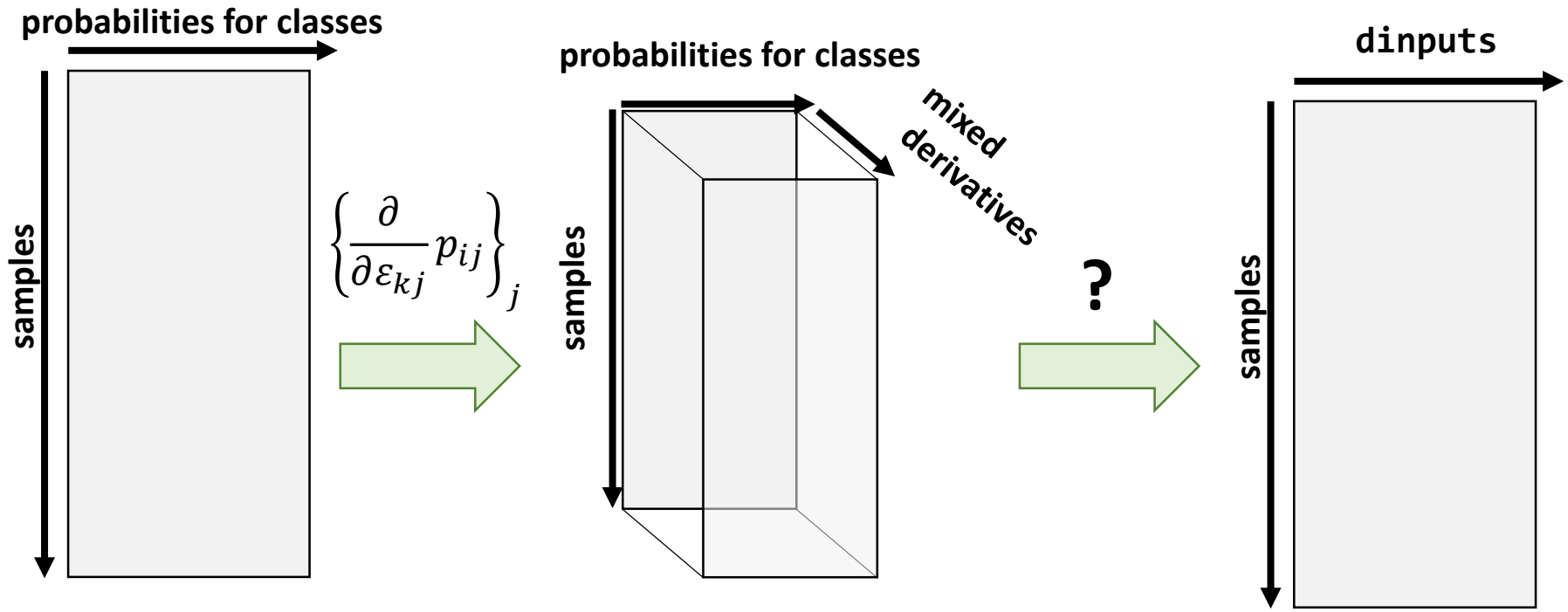


Hence, for each data point *j*, we are getting such a Jacobian matrix!

for each data point *j* and class/category *i*

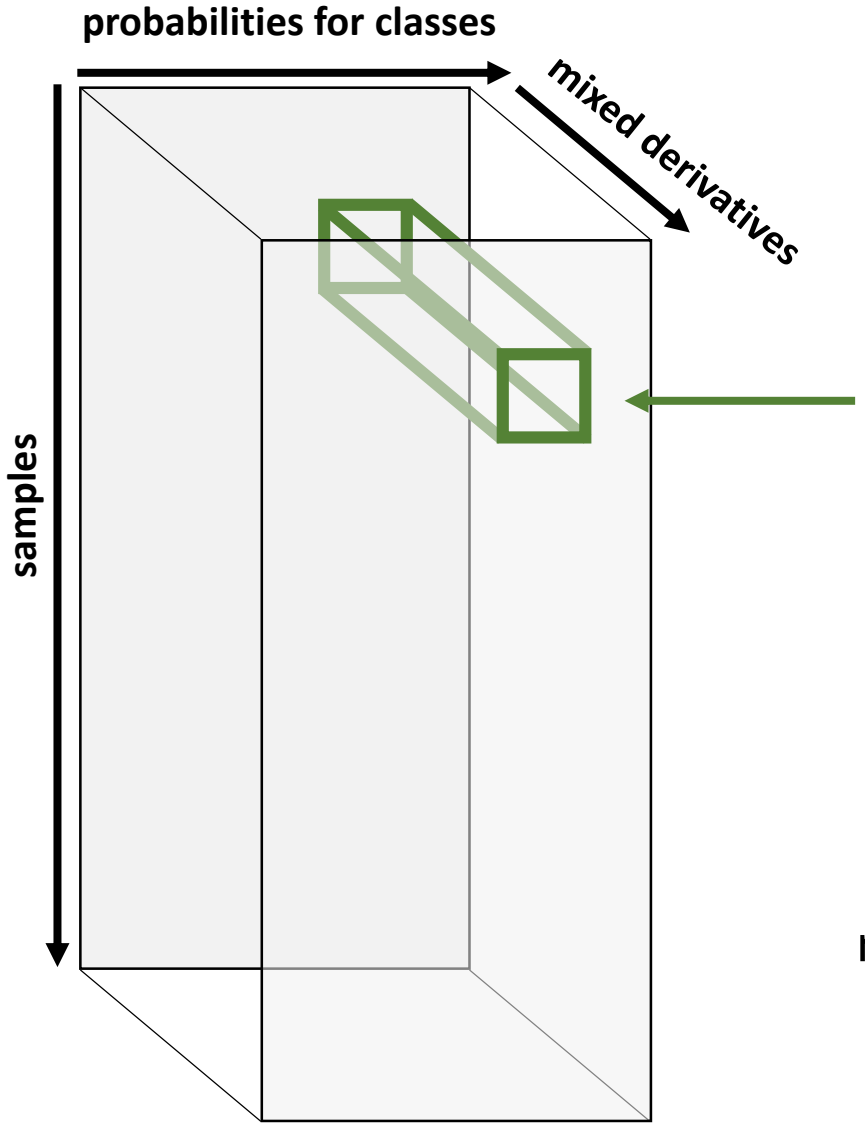
`softmax_output`

`backprop denseLayer input`



Hence, for each data point *j*, we are getting such a Jacobian matrix!

for each data point *j* and class/category *i*



$$p_{ij} = \frac{\exp(\varepsilon_{ij})}{\sum_i \exp(\varepsilon_{ij})}$$

- every p_i gets influenced by all the other *i = 1, …k,..*
- according to the chain rule: must get multiplied with `dvalues` from the loss function
- hence, multiplying two columns resulting in a number → `np.dot`

```
np.dot(jacobian_matrix, dvalues)
```

(but for each of these columns)

```
class Activation_Softmax:

    def forward(self,inputs):
        self.inputs = inputs
        exp_values  = np.exp(inputs - np.max(inputs, axis = 1,\
                                      keepdims = True))
        probabilities = exp_values/np.sum(exp_values, axis = 1,\
                                      keepdims = True)
        self.output   = probabilities

    def backward(self, dvalues):

        self.dinputs = np.empty_like(dvalues)
```

running the whole procedure for the entire data set in a loop

```
        for i, (single_output, single_dvalues) in \
            enumerate(zip(self.output, dvalues)):

            single_output   = single_output.reshape(-1,1)
```

jacobian for each single data point

```
            jacobMatr       = np.diagflat(single_output) - \
                              np.dot(single_output, single_output.T)
```

multiplying jacobian column wise with dvalues from single data point (chain rule)

```
            self.dinputs[i] = np.dot(jacobMatr, single_dvalues)
```

```
class Activation_Softmax:

    def forward(self,inputs):
        self.inputs = inputs
        exp_values  = np.exp(inputs - np.max(inputs, axis = 1,\
                                      keepdims = True))
        probabilities = exp_values/np.sum(exp_values, axis = 1,\
                                      keepdims = True)
        self.output   = probabilities

  def backward(self, dvalues):

      self.dinputs = np.empty_like(dvalues)

      for i, (single_output, single_dvalues) in \
          enumerate(zip(self.output, dvalues)):

          single_output   = single_output.reshape(-1,1)

          jacobMatr       = np.diagflat(single_output) - \
                            np.dot(single_output, single_output.T)

          self.dinputs[i] = np.dot(jacobMatr, single_dvalues)
```

ε_{ij} from the last dense layer

softmax

$$p_{ij} = \frac{\exp(\varepsilon_{ij})}{\sum_i \exp(\varepsilon_{ij})}$$

loss

$$L_m = -\log(p_m)$$

returns total loss

calculating total loss

true class

$$\begin{pmatrix} 0.20 & \mathbf{0.70} & 0.10 \\ \mathbf{0.90} & 0.05 & 0.05 \\ \mathbf{0.60} & 0.10 & 0.30 \\ 0.15 & 0.05 & \mathbf{0.80} \\ \dots & \dots & \dots \end{pmatrix} \text{ vs } \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ \dots & \dots & \dots \end{pmatrix}$$

calculating gradient

`CalcSoftmaxLossGrad`

```
class CalcSoftmaxLossGrad:

    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss       = Loss_CategoricalCrossEntropy()

    def forward(self, inputs, y_true):
        self.activation.forward(inputs)
        self.output = self.activation.output

        return(self.loss.calculate(self.output, y_true))
```

calculating softmax

saving softmax output

returning total loss

see **Loss()**

predicted probabilities for each class from softmax

```
    def backward(self, dvalues, y_true):
        Nsamples = len(dvalues)

        if len(y_true.shape) == 2:
            y_true       = np.argmax(y_true, axis = 1)
        self.dinputs = dvalues.copy()
```

calculating the gradient of the probabilities

```
        self.dinputs[range(Nsamples), y_true] -= 1
        self.dinputs = self.dinputs/Nsamples
```

```
class CalcSoftmaxLossGrad:

    def __init__(self):
        self.activation = Activation_Softmax()
        self.loss       = Loss_CategoricalCrossEntropy()

    def forward(self, inputs, y_true):
        self.activation.forward(inputs)
        self.output = self.activation.output

        return(self.loss.calculate(self.output, y_true))

    def backward(self, dvalues, y_true):
        Nsamples = len(dvalues)

        if len(y_true.shape) == 2:
            y_true       = np.argmax(y_true, axis = 1)
         self.dinputs = dvalues.copy()
```

calculating the gradient of the probabilities

```
         self.dinputs[range(Nsamples), y_true] -= 1
         self.dinputs = self.dinputs/Nsamples
```

$$\begin{pmatrix} 0.20 & \mathbf{0.70} & 0.10 \\ \mathbf{0.90} & 0.05 & 0.05 \\ \mathbf{0.60} & 0.10 & 0.30 \\ 0.15 & 0.05 & \mathbf{0.80} \\ \dots & \dots & \dots \end{pmatrix} \text{ vs } \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ \dots & \dots & \dots \end{pmatrix}$$

```
nc      = 3
[x, y] = spiral_data(samples = 200, classes = nc)
S       = np.shape(x)

dense1          = Layer_Dense(S[1], 4)
dense2          = Layer_Dense(4, nc)
activation1     = Activation_ReLU()

loss_function = CalcSoftmaxLossGrad()
```

defining layers

```
dense1.forward(x)
activation1.forward(dense1.output)

dense2.forward(activation1.output)

loss = loss_function.forward(dense2.output,y)

predictions = np.argmax(loss_function.output, axis = 1)
    if len(y.shape) == 2:
        y = np.argmax(y,axis = 1)
accuracy = np.mean(predictions == y)
```

creating the actual network

```
dense1.forward(x)
activation1.forward(dense1.output)

dense2.forward(activation1.output)

loss = loss_function.forward(dense2.output,y)

predictions = np.argmax(loss_function.output, axis = 1)
    if len(y.shape) == 2:
        y = np.argmax(y,axis = 1)
accuracy = np.mean(predictions == y)
```

creating the actual network

```
loss_function.backward(loss_function.output,y)

dense2.backward(loss_function.dinputs)

activation1.backward(dense2.dinputs)

dense1.backward(activation1.dinputs)
```

backpropagation

We are done with the backpropagation now

→ let's run the code on the spiral data set in order to make sure that it works so far

→ also check:

```
print('loss:', loss)
print('accuracy:', accuracy)

print(dense1.dweights)
print(dense1.dbiases)

print(dense2.dweights)
print(dense2.dbiases)

print(loss_function.dinputs)
```

The gradients are propagating through our network now.

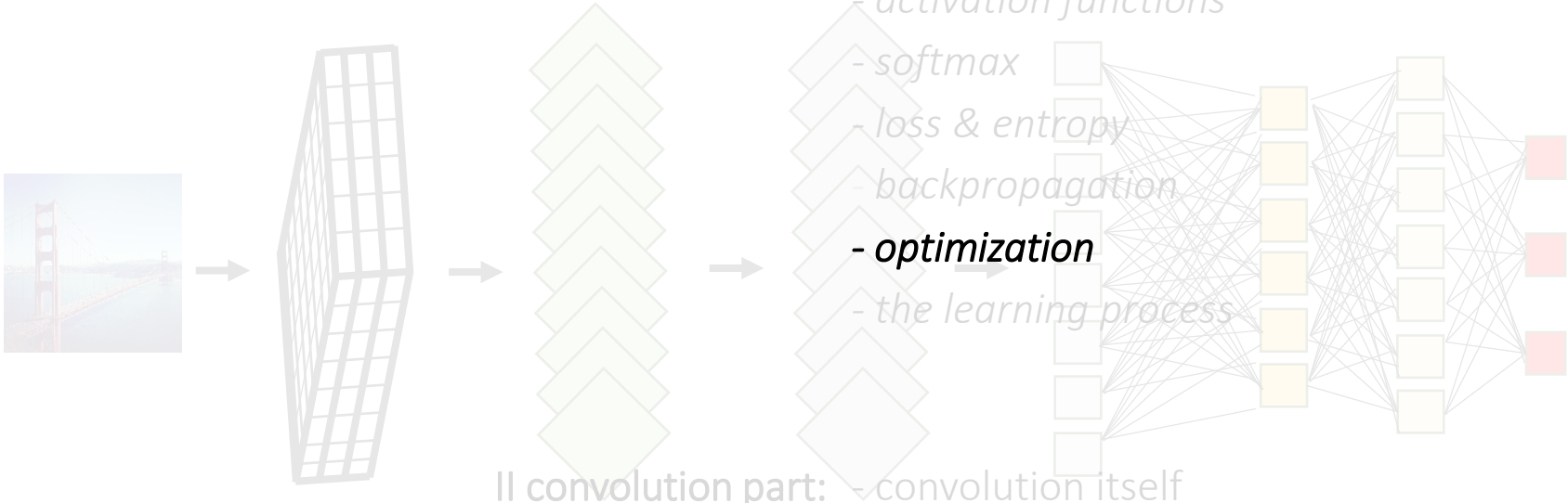
→ minimizing the gradient in a automated fashion → that's how the ANN learns

outline

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*



II convolution part:

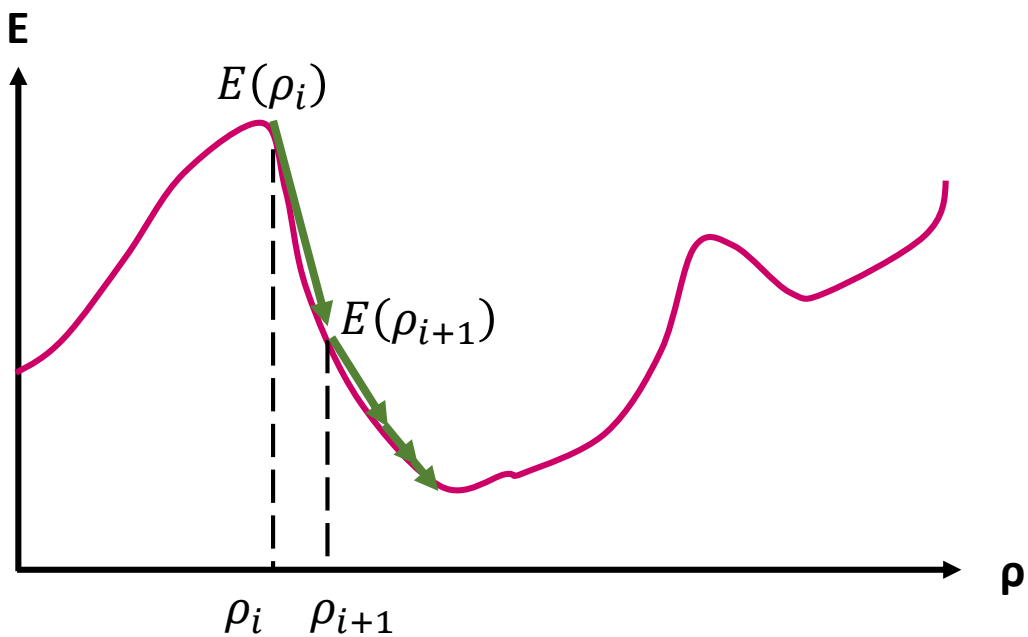
- convolution itself
- max pool and average pool
- batch normalization
- flattening
- learning and “teaching”
- final remarks

The gradients are propagating through our network now.

→ minimizing the gradient in a automated fashion → that's how the ANN learns

Gradient Descent

cost function ***E*** depending on parameter ***ρ***



$$grad(E)_{\rho_i} \approx \frac{E(\rho_{i+1}) - E(\rho_i)}{\rho_{i+1} - \rho_i}$$

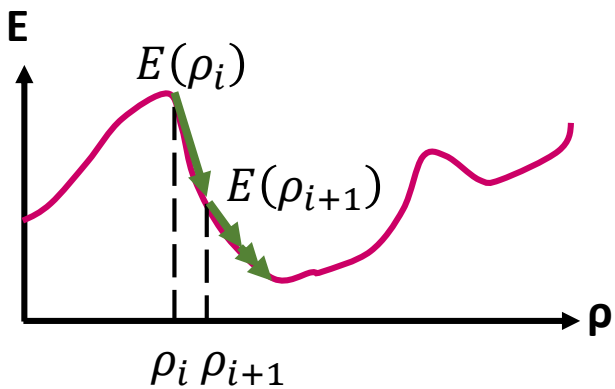
$$\rho_{i+1} = \rho_i - \alpha \cdot grad(E)_{\rho_i}$$

learning rate *α*

Gradient Descent

$$\rho_{i+1} = \rho_i - \alpha \cdot grad(E)_{\rho_i}$$

learning rate $\boldsymbol{\alpha}$



- assuming that there is no global maximum
- could get stuck in a local minimum

- learning rate $\boldsymbol{\alpha}$ is a *hyper parameter* we have to give

- we need to update weights and biases according to the above equation
  → derivatives · $\boldsymbol{\alpha}$

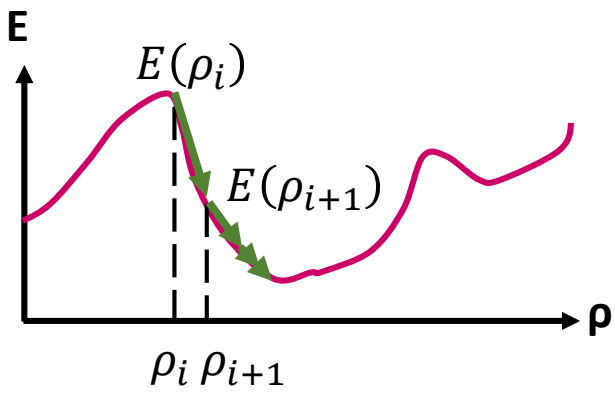
- writing a class that accesses the layers and performs the update
  "**`Optimizer_SGD`**"
  Stochastic **G**radient **D**escent

- **`Optimizer_SGD`** is updating the parameter values
  → running the whole thing in a loop over N iterations

Gradient Descent

$$\rho_{i+1} = \rho_i - \alpha \cdot grad(E)_{\rho_i}$$

learning rate α




```
class Optimizer_SGD:
```

default learning rate is 0.1

```
    def __init__(self, learning_rate = 0.1):
        self.learning_rate = learning_rate


    def update_params(self, layer):
        layer.weights += -self.learning_rate * layer.dweights
        layer.biases  += -self.learning_rate * layer.dbiases
```

The entire code so far

```
nc      = 3
[x, y] = spiral_data(samples = 200, classes = nc)
S       = np.shape(x)

dense1        = Layer_Dense(S[1], 4)
dense2        = Layer_Dense(4, nc)
activation1   = Activation_ReLU()
loss_function = CalcSoftmaxLossGrad()

optimizer     = Optimizer_SGD()

dense1.forward(x)
activation1.forward(dense1.output)
dense2.forward(activation1.output)
loss = loss_function.forward(dense2.output,y)

predictions = np.argmax(loss_function.output, axis = 1)
    if len(y.shape) == 2:
       y        = np.argmax(y,axis = 1)
accuracy = np.mean(predictions == y)

print('loss:', loss)
print('accuracy:', accuracy)

loss_function.backward(loss_function.output,y)
dense2.backward(loss_function.dinputs)
activation1.backward(dense2.dinputs)
dense1.backward(activation1.dinputs)

optimizer.update_params(dense1)
optimizer.update_params(dense2)
```

run this part in a loop

```
for epoch in range(10000):
```

let us watch how the ANN learns: print the **`loss`** within the loop, each 100th epoch

```
if not epoch % 100:
            print(f'epoch: {epoch}, ' +
                  f'accuracy: {accuracy:.3f}, ' +
                  f'loss: {loss:.3f}')
```

Console 1/A

```
epoch: 0, accuracy: 0.413
epoch: 100, accuracy: 0.4
epoch: 200, accuracy: 0.5
epoch: 300, accuracy: 0.5
epoch: 400, accuracy: 0.5
epoch: 500, accuracy: 0.5
epoch: 600, accuracy: 0.5
epoch: 700, accuracy: 0.5
epoch: 800, accuracy: 0.5
epoch: 900, accuracy: 0.5
epoch: 1000, accuracy: 0.
epoch: 1100, accuracy: 0.
epoch: 1200, accuracy: 0.
epoch: 1300, accuracy: 0.
epoch: 1400, accuracy: 0.
```

Console 1/A

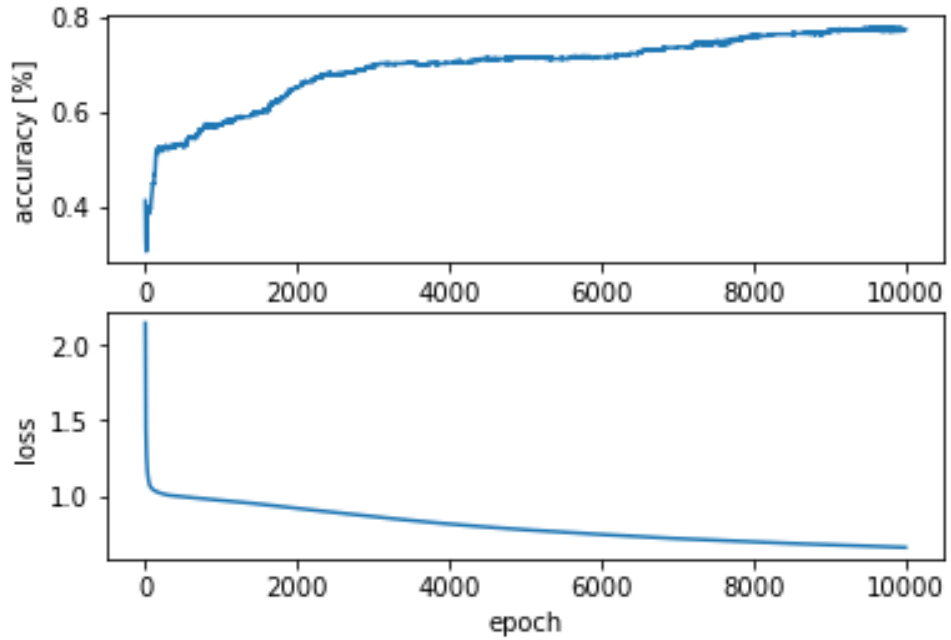
```
epoch: 8500, accuracy: 0.763, loss: 0.683
epoch: 8600, accuracy: 0.765, loss: 0.681
epoch: 8700, accuracy: 0.767, loss: 0.679
epoch: 8800, accuracy: 0.767, loss: 0.677
epoch: 8900, accuracy: 0.767, loss: 0.675
epoch: 9000, accuracy: 0.768, loss: 0.673
epoch: 9100, accuracy: 0.770, loss: 0.672
epoch: 9200, accuracy: 0.772, loss: 0.670
epoch: 9300, accuracy: 0.772, loss: 0.668
epoch: 9400, accuracy: 0.772, loss: 0.666
epoch: 9500, accuracy: 0.773, loss: 0.665
epoch: 9600, accuracy: 0.772, loss: 0.663
epoch: 9700, accuracy: 0.770, loss: 0.661
epoch: 9800, accuracy: 0.770, loss: 0.660
epoch: 9900, accuracy: 0.772, loss: 0.658
```

We also want to plot **loss** and **accuracy**

```
        Nsteps  = 10000
        Monitor = np.zeros((Nsteps,2))

for epoch in range(Nsteps):

                ...


    Monitor[epoch,0] = accuracy
    Monitor[epoch,1] = loss


                ...

fig, ax = plt.subplots(2, 1)
   ax[0].plot(np.arange(Nsteps), Monitor[:,0])
   ax[0].set_ylabel('accuracy [%]')
   ax[1].plot(np.arange(Nsteps), Monitor[:,1])
   ax[1].set_xlabel('epoch')
   ax[1].set_ylabel('loss')
```

We also want to plot **loss** and **accuracy**

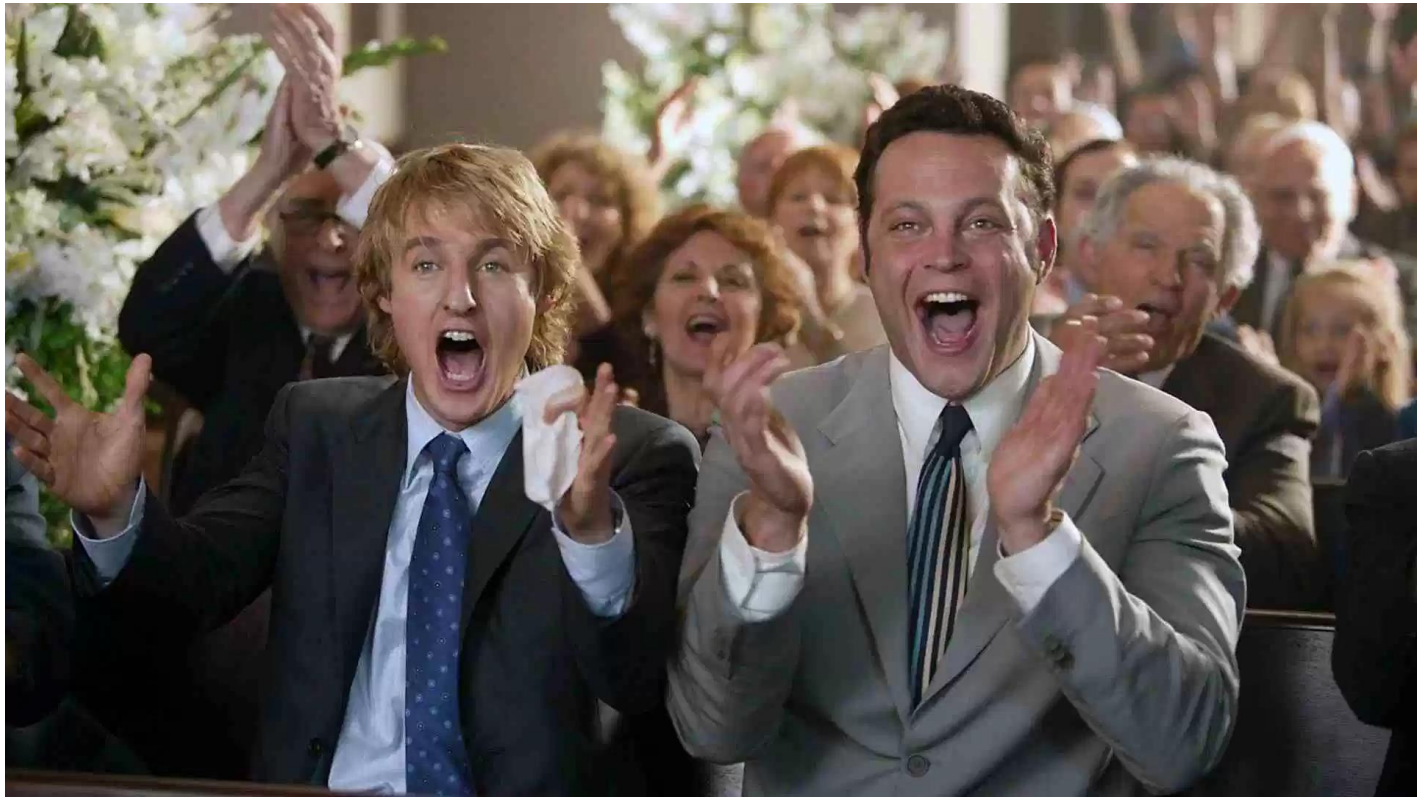
```
Nsteps  = 10000
Monitor = np.zeros((Nsteps,2))

for epoch in range(Nsteps):

        ...

    Monitor[epoch,0] = accuracy
    Monitor[epoch,1] = loss

        ...
```




```
fig, ax = plt.subplots(2, 1)
   ax[0].plot(np.arange(Nsteps), Monitor[:,0])
   ax[0].set_ylabel('accuracy [%]')
   ax[1].plot(np.arange(Nsteps), Monitor[:,1])
   ax[1].set_xlabel('epoch')
   ax[1].set_ylabel('loss')
```

We have created a fully functional ANN from scratch!

Congratulations

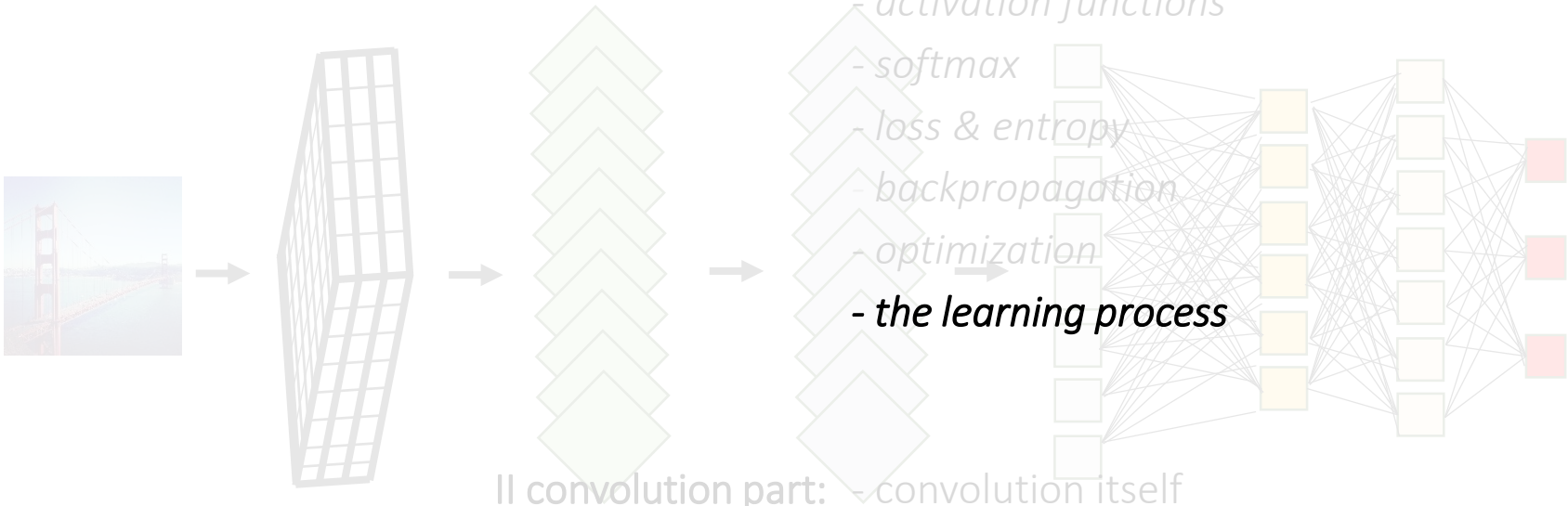


outline

0 intro

I the core:

- *a single neuron*
- *layers of neurons*
- *activation functions*
- *softmax*
- *loss & entropy*
- *backpropagation*
- *optimization*
- *the learning process*



II convolution part:

- convolution itself
- max pool and average pool
- batch normalization
- flattening
- learning and “teaching”
- final remarks

Aim in this section: improving how the ANN can learn

<u>problem:</u>

→ if α is too large: we could miss a minimum
→ if α is too small: learning is slow or we could get stuck in a local minimum

<u>solution:</u>

→ ***learning rate decay***
→ taking "shape" of gradient parameter space into account (passing local minima by using the ***momentum***)

<u>There is way more one could do:</u>

- penalizing large weights and biases (***L1 & L2 regularization***)

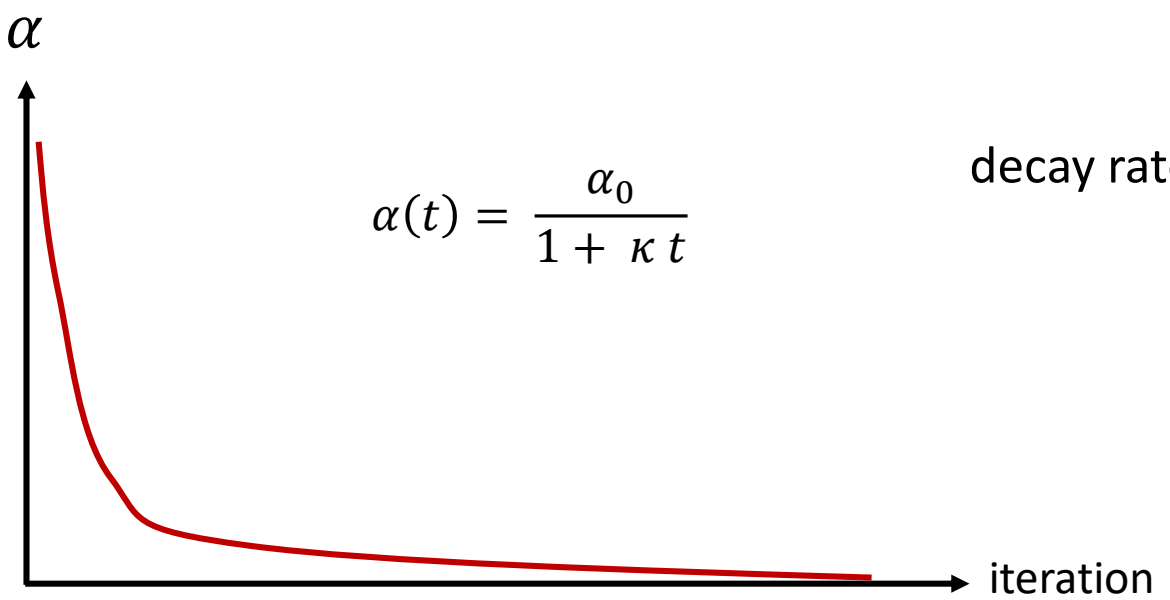
- ***adaptive gradient*** (different learning rates per parameter)

- ***drop out***

- ….

Aim in this section: improving how the ANN can learn

<u>solution:</u> → ***learning rate decay***



$$\alpha(t) = \frac{\alpha_0}{1 + \kappa\, t}$$

decay rate κ

learning rate decay:

```
class Optimizer_SGD:

    def __init__(self, learning_rate = 0.1, decay = 0):
        self.learning_rate          = learning_rate
        self.decay                  = decay
        self.current_learning_rate = learning_rate
        self.iterations             = 0

    def pre_update_params(self):
        if self.decay:
            self.current_learning_rate = self.learning_rate*\
                                (1/ (1 + self.iterations * self.decay))

    def update_params(self, layer):
        layer.weights += -self.current_learning_rate * layer.dweights
        layer.biases  += -self.current_learning_rate * layer.dbiases

    def post_update_params(self):
        self.iterations += 1
```

$$\boxed{\alpha(t) = \frac{\alpha_0}{1 + \kappa\, t}}$$

learning rate decay:

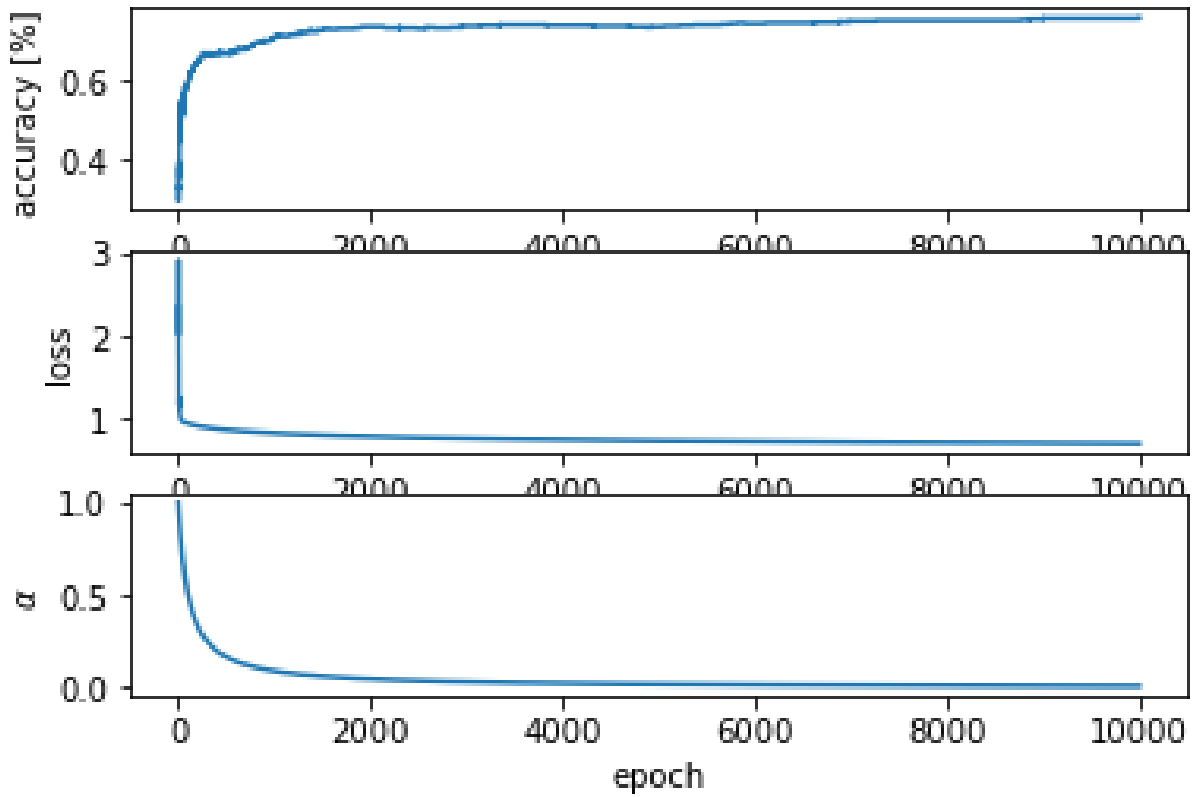
let us now include the decay and run the code again:

```
optimizer      = Optimizer_SGD(decay = 0.02)

Nsteps         = 10000
Monitor        = np.zeros((Nsteps,3))

 for epoch in range(Nsteps):
                ...
    Monitor[epoch,2] = optimizer.current_learning_rate
                ...

                                          ← calculating current α
    optimizer.update_params(dense1)
    optimizer.update_params(dense2)
                                          ← counting iterations

                ...

 ax[2].plot(np.arange(Nsteps), Monitor[:,2])
 ax[2].set_ylabel(r'$\alpha$')
 ax[2].set_xlabel('epoch')
```

learning rate decay:

let us now include the decay and run the code again:

```
optimizer      = Optimizer_SGD(decay = 0.02)

Nsteps         = 10000
Monitor        = np.zeros

 for epoch in range(Nste
                ...
     Monitor[epoch,2] = (
                ...

     optimizer.pre_update
     optimizer.update_par
     optimizer.update_par
     optimizer.post_updat
                ...
```




```
ax[2].plot(np.arange(Nsteps), Monitor[:,2])
ax[2].set_ylabel(r'$\alpha$')
ax[2].set_xlabel('epoch')
```

<u>momentum:</u> using the **rolling average** over the gradient
→ makes it more likely to pass a small local minimum (like a "hole")

so far: $\rho_{i+1} = \rho_i - \alpha \cdot grad(E)_{\rho_i}$

now: $\rho_{i+1} = \rho_i + \mu \cdot \mu_i - \alpha \cdot grad(E)_{\rho_i}$ momentum μ

$$\mu_{i+1} = \mu \cdot \mu_i - \alpha \cdot grad(E)_{\rho_i}$$

$$\rho_{i+2} = \rho_{i+1} + \mu \cdot \overbrace{[\mu \cdot \mu_i - \alpha \cdot grad(E)_{\rho_i}]}^{\mu_{i+1}} - \alpha \cdot grad(E)_{\rho_{i+1}}$$

$$\mu_{i+2} = \mu \cdot \mu_{i+1} - \alpha \cdot grad(E)_{\rho_i+1}$$

and so on…

<u>momentum:</u>

```
class Optimizer_SGD:

    def __init__(self, learning_rate = 0.1, decay = 0, momentum = 0):
      self.learning_rate         = learning_rate
      self.decay                 = decay
      self.current_learning_rate = learning_rate
      self.iterations            = 0
      self.momentum              = momentum
```

<u>momentum:</u>

```
class Optimizer_SGD:
        ...

    def update_params(self, layer):

    if self.momentum:

        if not hasattr(layer, 'weight_momentums'):
                layer.weight_momentums = np.zeros_like(layer.weights)
                layer.bias_momentums   = np.zeros_like(layer.biases)
```

we attach the momentum to the layer in order to remember the rolling average

$$\boldsymbol{\mu_{i+1} = \mu \cdot \mu_i - \alpha \cdot grad(E)_{\rho_i}}$$

```
        weight_updates = self.momentum * layer.weight_momentums - \
                         self.current_learning_rate * layer.dweights
        layer.weight_momentums = weight_updates

        bias_updates =   self.momentum * layer.bias_momentums - \
                         self.current_learning_rate * layer.dbiases
        layer.bias_momentums = bias_updates

    else:
```

<u>momentum:</u>

```
class Optimizer_SGD:
        ...
    def update_params(self, layer):


    if self.momentum:

        ...

    else:
```

same as before

```
        weight_updates = -self.current_learning_rate * layer.dweights
        bias_updates   = -self.current_learning_rate * layer.dbiases


    layer.weights += weight_updates
    layer.biases  += bias_updates
```

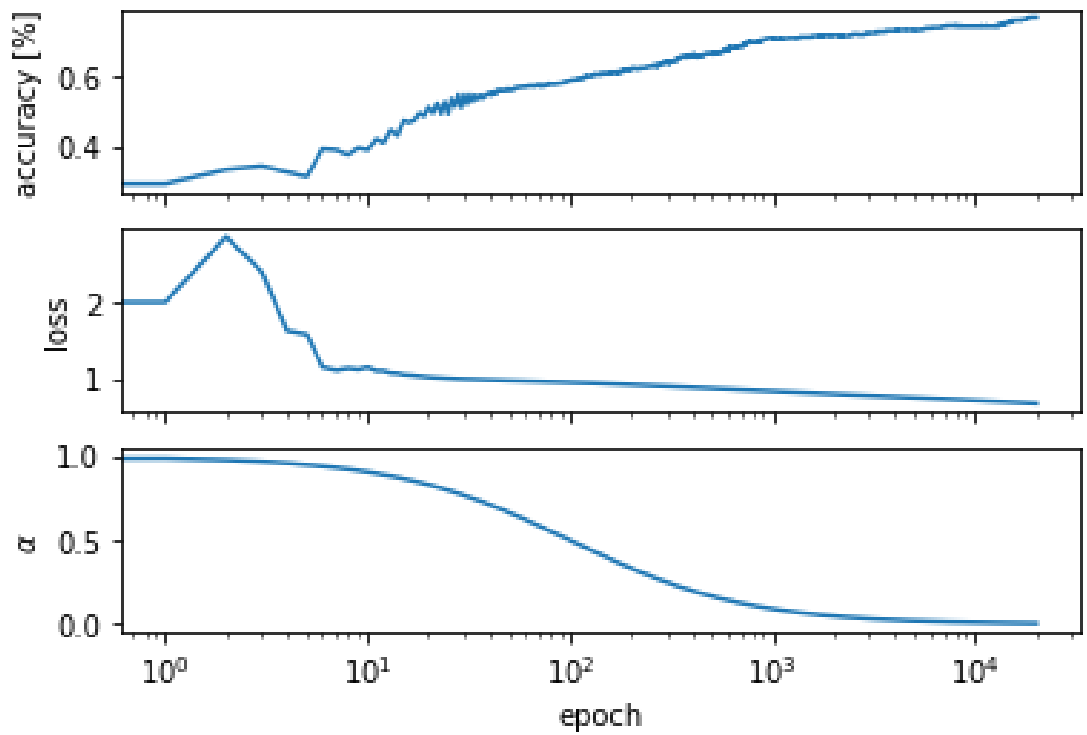
<u>momentum:</u>

run the code with eg

```
optimizer= Optimizer_SGD(decay = 0.01, momentum = 0.1)
```

Console 1/A

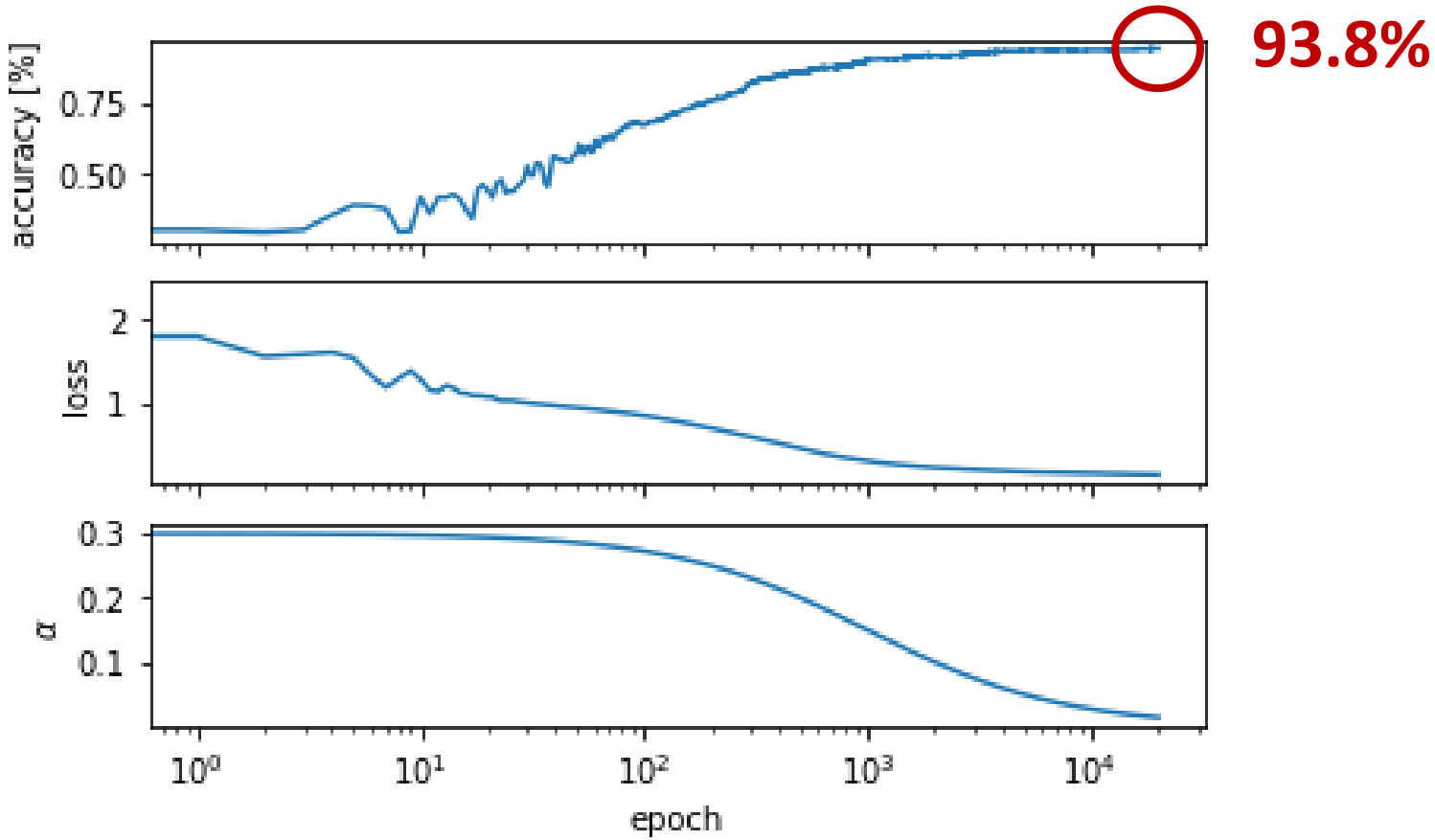
```
epoch: 18700, accuracy: 0.770, loss: 0.667, actual learning rate: 0.005319148936170213
epoch: 18800, accuracy: 0.770, loss: 0.667, actual learning rate: 0.005291005291005291
epoch: 18900, accuracy: 0.770, loss: 0.667, actual learning rate: 0.005263157894736842
epoch: 19000, accuracy: 0.770, loss: 0.666, actual learning rate: 0.005235602094240838
epoch: 19100, accuracy: 0.770, loss: 0.666, actual learning rate: 0.005208333333333333
epoch: 19200, accuracy: 0.770, loss:
epoch: 19300, accuracy: 0.770, loss:
epoch: 19400, accuracy: 0.770, loss:
epoch: 19500, accuracy: 0.770, loss:
epoch: 19600, accuracy: 0.770, loss:
epoch: 19700, accuracy: 0.770, loss:
epoch: 19800, accuracy: 0.770, loss:
epoch: 19900, accuracy: 0.770, loss:
```



<u>momentum:</u>

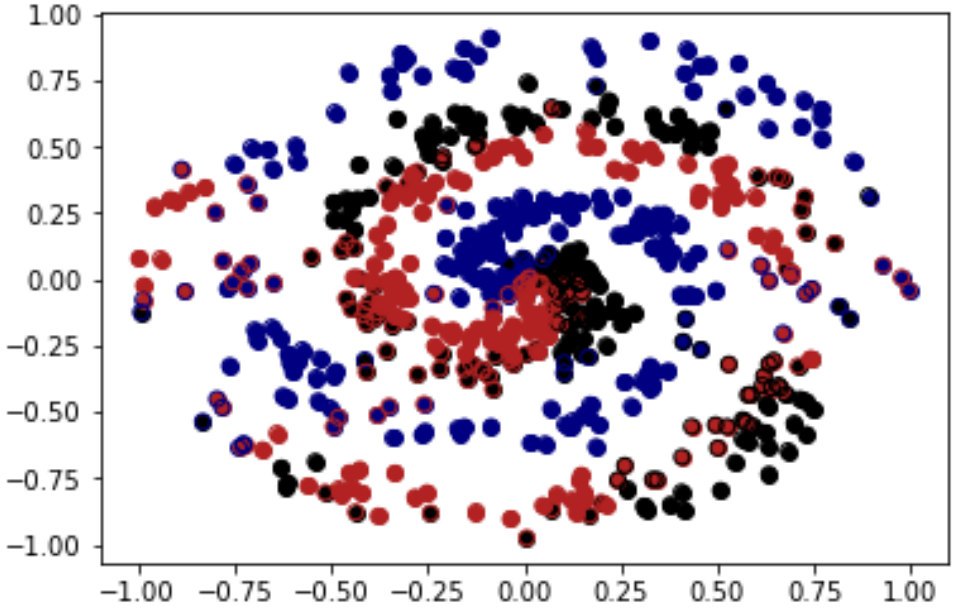
run the code with eg

```
optimizer = Optimizer_SGD(learning_rate = 0.2, decay = 0.001, \
                          momentum = 0.9)
```



<u>before we are done:</u>

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application



<u>before we are done:</u> - running the code is still a bit clumsy → calling ANN externally

- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

We want to create a function, that runs the ANN and creates the plot

→ removing the
- plotting part
- the loop over the different epochs
- part where we initialize the layers and set up the network structure

and copy it into a new function we call **RunMyANN**

before we are done:

- running the code is still a bit clumsy → calling ANN externally

- creating a function that runs the actual ANN incl plots

- checking predicted classes visually
- extracting the weights for actual application

```
def RunMyANN():

    import matplotlib.pyplot as plt
    import numpy as np
    import nnfs #run "pip install nnfs" first
    from nnfs.datasets import spiral_data

    #creating the actual data:
    #x: 3x200 data points (2D)
    #y: corresponding classes
    [x, y] = spiral_data(samples = 200, classes = 3)

    nnfs.init()
```

before we are done:

- running the code is still a bit clumsy → calling ANN externally

- creating a function that runs the actual ANN incl plots

- checking predicted classes visually
- extracting the weights for actual application

```
def RunMyANN():

...
    nnfs.init()

    import WithOptimizationLearningRateDecayMomentum as MyANN

    dense1          = MyANN.Layer_Dense(2, 64)
    activation1     = MyANN.Activation_ReLU()
    optimizer       = MyANN.Optimizer_SGD(decay = 0.01, momentum = 0.1)
    dense2          = MyANN.Layer_Dense(len(dense1.biases.T), 3)
    loss_activation = MyANN.CalcSoftmaxLossGrad()

    Nsteps  = 20000
    Monitor = np.zeros((Nsteps,3))

    for epoch in range(Nsteps):
        ...
```

calling the ANN

← as before

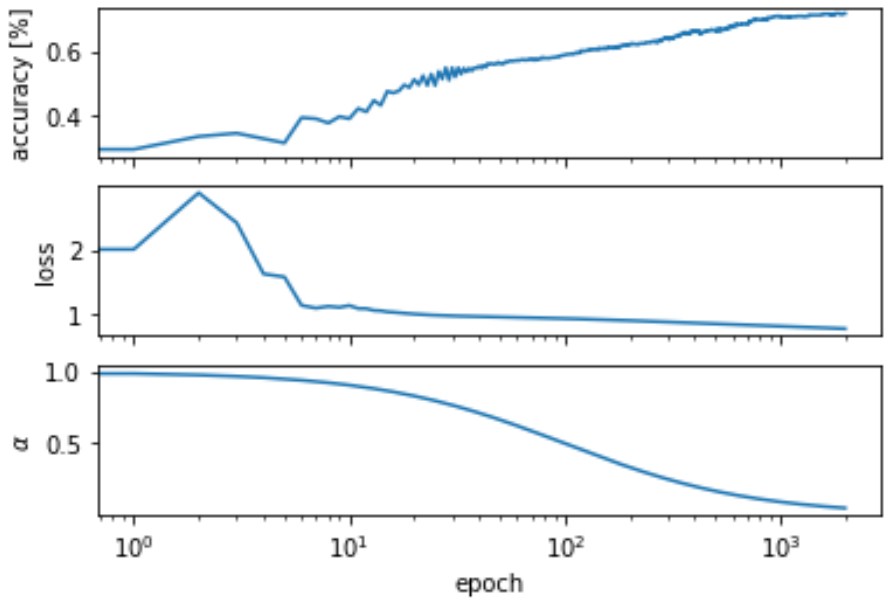
<u>before we are done:</u>

- running the code is still a bit clumsy → calling ANN externally

- creating a function that runs the actual ANN incl plots

- checking predicted classes visually
- extracting the weights for actual application

compile & test – run the function **`RunMyANN.py`**



before we are done:

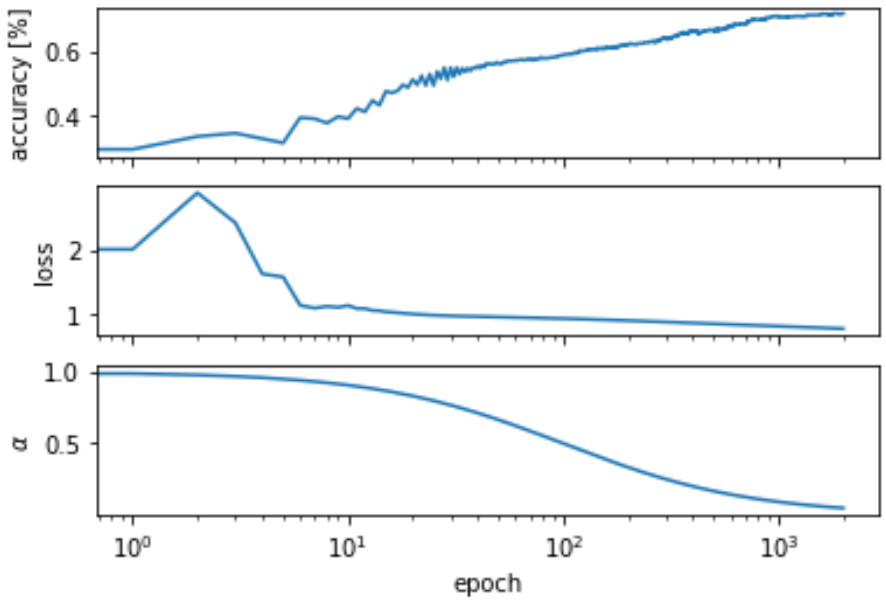
- running the code is still a bit clumsy → calling externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

go to the very end of **RunMyANN.py**

```
fig1, ax = plt.subplots(3, 1,sharex=True)
    ax[0].plot(np.arange(Nsteps), Monitor[:,0])
    ax[0].set_ylabel('accuracy [%]')
    ax[1].plot(np.arange(Nsteps), Monitor[:,1])
    ax[1].set_ylabel('loss')
    ax[2].plot(np.arange(Nsteps), Monitor[:,2])
    ax[2].set_ylabel('$\alpha$')
    ax[2].set_xlabel('epoch')
    plt.xscale('log',base=10)
```

`plt.show()`

needed in order to show new plot in different window



before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

go to the very end of **RunMyANN.py**

```
...
    plt.xscale('log',base=10)
    plt.show()

  idx0  = np.argwhere(y==0)
  idx1  = np.argwhere(y==1)
  idx2  = np.argwhere(y==2)

  idxp0 = np.argwhere(predictions==0)
  idxp1 = np.argwhere(predictions==1)
  idxp2 = np.argwhere(predictions==2)
```

extracting actual classes and predicted classes

before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

go to the very end of **RunMyANN.py**

```
...
    idx0  = np.argwhere(y==0)
    idx1  = np.argwhere(y==1)
    idx2  = np.argwhere(y==2)

    idxp0 = np.argwhere(predictions==0)
    idxp1 = np.argwhere(predictions==1)
    idxp2 = np.argwhere(predictions==2)
```

actual classes

```
plt.scatter(x[idx0,0], x[idx0,1], color = 'black')
plt.scatter(x[idx1,0], x[idx1,1], color = [0, 0, 0.5])#nice blue
plt.scatter(x[idx2,0], x[idx2,1], color = [0.7, 0.13, 0.13])#nice red
```

predicted classes

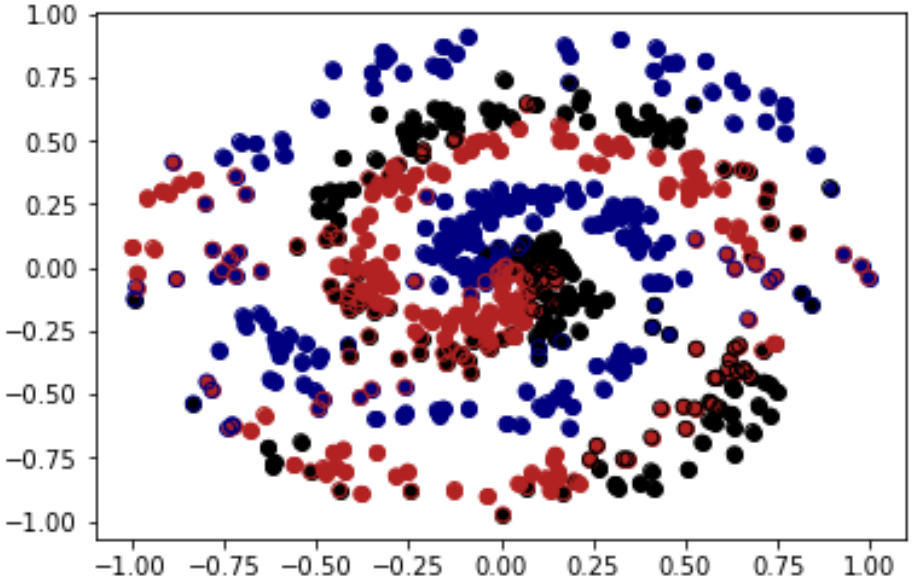
```
plt.scatter(x[idxp0,0], x[idxp0,1], marker = 'o', \
            facecolors = 'none', edgecolors = 'black')
plt.scatter(x[idxp1,0],x[idxp1,1], marker = 'o', \
            facecolors = 'none', edgecolors = [0, 0, 0.5])
plt.scatter(x[idxp2,0],x[idxp2,1], marker = 'o', \
            facecolors = 'none', edgecolors = [0.7, 0.13, 0.13])
```

before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

go to the very end of **RunMyANN.py**

```
...
    idx0  = np.argwhere(y==0)
    idx1  = np.argwhere(y==1)
    idx2  = np.argwhere(y==2)

    idxp0 = np.argwhere(predictions==0)
    idxp1 = np.argwhere(predictions==1)
    idxp2 = np.argwhere(predictions==2)

    plt.scatter(x[idx0,0], x[idx0,1], color = 'black')
    plt.scatter(x[idx1,0], x[idx1,1], color = [0, 0, 0.5]
    plt.scatter(x[idx2,0], x[idx2,1], color = [0.7, 0.13,

    plt.scatter(x[idxp0,0], x[idxp0,1], marker = 'o', \
                        facecolors = 'none', edgecolors =
    plt.scatter(x[idxp1,0],x[idxp1,1], marker = 'o', \
                        facecolors = 'none', edgecolors =
    plt.scatter(x[idxp2,0],x[idxp2,1], marker = 'o', \
                        facecolors = 'none', edgecolors =
```



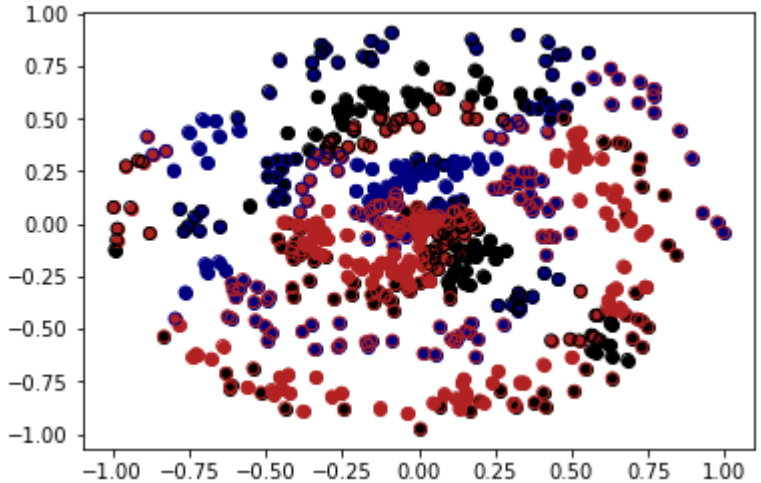
before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots

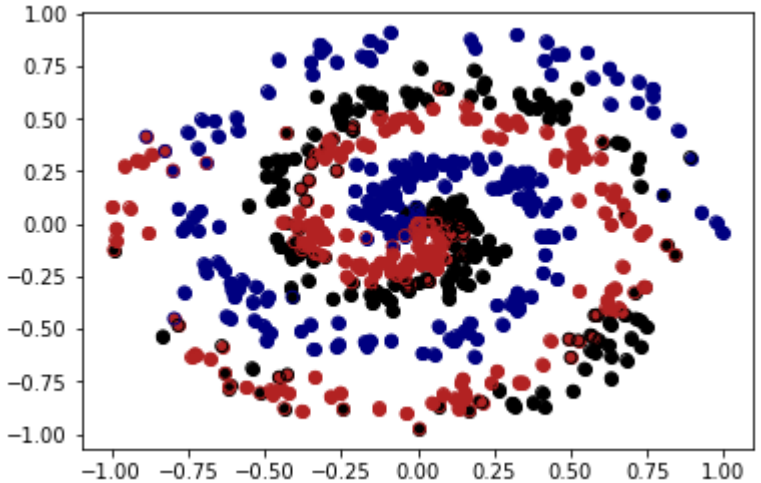
the training process:

- checking predicted classes visually

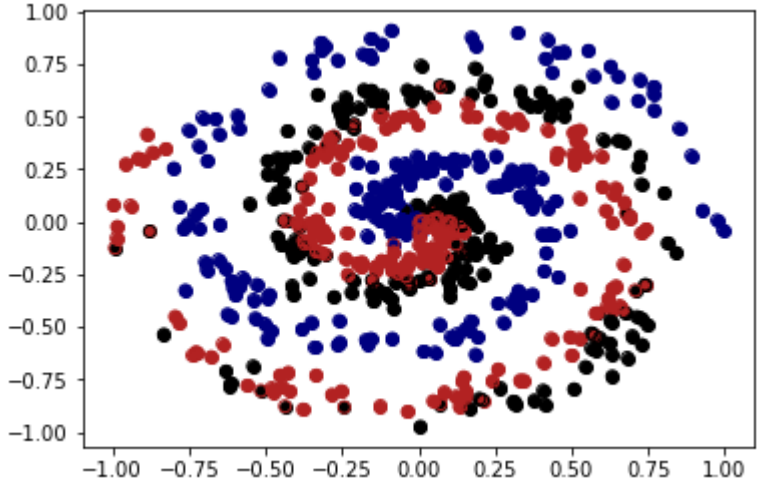
- extracting the weights for actual application



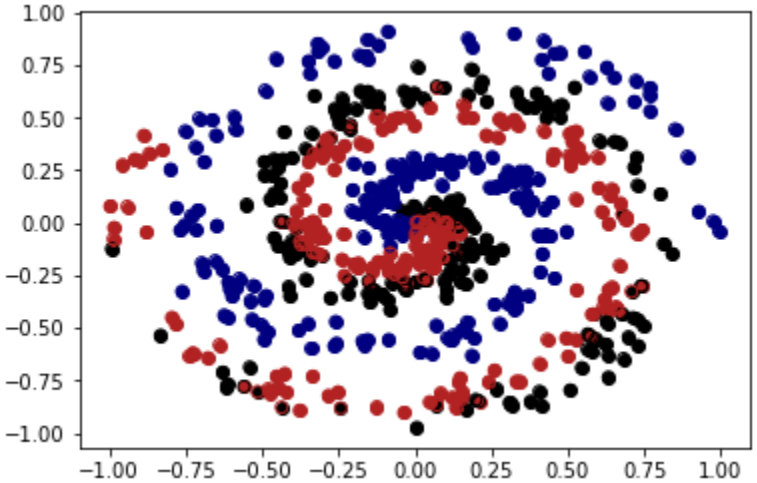
`Nsteps = 100,` accuracy = 65%



`Nsteps = 1000,` accuracy = 84%



`Nsteps = 5000,` accuracy = 93%



`Nsteps = 20000,` accuracy = 94%

before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

Since we have trained our ANN, we can use its memory hence the weights for classification!

go to **RunMyANN.py**

and add lines at the very end to save the weights and biases

```
np.save('weights1.npy', dense1.weights)
np.save('weights2.npy', dense2.weights)

np.save('bias1.npy', dense1.biases)
np.save('bias2.npy', dense2.biases)
```

before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

We are now creating a new file `ApplyMyANN.py` that calls the saved weights when applying the ANN to new data

```
def ApplyMyANN(x_new):

    import numpy as np

    #calling the ANN
    import WithOptimizationLearningRateDecayMomentum as MyANN

    #loading saved weights & biases
    w1 = np.load('weights1.npy)
    w2 = np.load('weights2.npy')

    b1 = np.load('bias1.npy')
    b2 = np.load('bias2.npy')
```

new data with unknown classes

before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

```
def ApplyMyANN(x_new):

    import numpy as np
    import WithOptimizationLearningRateDecayMomentum as MyANN

    w1 = np.load('weights1.npy', dtype = float)
    w2 = np.load('weights2.npy', dtype = float)

    b1 = np.load('bias1.npy', dtype = float)
    b2 = np.load('bias2.npy', dtype = float)


    #creating network structure
        s    = w1.shape
        nrow = s[0]
        ncol = s[1]

        nCat = 3

        dense1          = MyANN.Layer_Dense(nrow, ncol)
        activation1     = MyANN.Activation_ReLU()
        dense2          = MyANN.Layer_Dense(len(dense1.biases.T), nCat)
```

before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

```
def ApplyMyANN(x_new):

    #creating network structure
        s    = w1.shape
        nrow = s[0]
        ncol = s[1]

        nCat = 3

        dense1          = MyANN.Layer_Dense(nrow, ncol)
        activation1     = MyANN.Activation_ReLU()
        dense2          = MyANN.Layer_Dense(len(dense1.biases.T), nCat)

        #transferring weights & biases
        dense1.weights = w1
        dense2.weights = w2

        dense1.biases  = b1
        dense2.biases  = b2

        #feeding the network
        dense1.forward(x_new)
        activation1.forward(dense1.output)
        dense2.forward(activation1.output)
        result = dense2.output
```

We need to maintain the structure of the ANN we have used for training

before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

```
def ApplyMyANN(x_new):

    #transferring weights & biases
    dense1.weights = w1
    dense2.weights = w2

    dense1.biases  = b1
    dense2.biases  = b2

    #feeding the network
    dense1.forward(x_new)
    activation1.forward(dense1.output)
    dense2.forward(activation1.output)
    result = dense2.output

    exp_values    = np.exp(result - np.max(result, axis = 1,\
                                   keepdims = True))
    probabilities = exp_values/np.sum(exp_values, axis = 1,\
                                   keepdims = True)


    predictions   = np.argmax(probabilities, axis = 1)


    return(predictions)
```

Again, we are using the Boltzmann distribution for calculating the probabilities

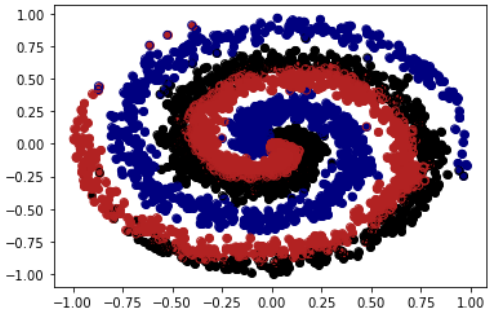
before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

now we are done → performing two steps for using the ANN:

1) training **(data set as large and diverse as possible, ideally needed to be done only once)**

RunMyANN()



2) application **(to new, yet unclassified data)**

```
predClass = ApplyMyANN(x_new)
```

the actual analysis

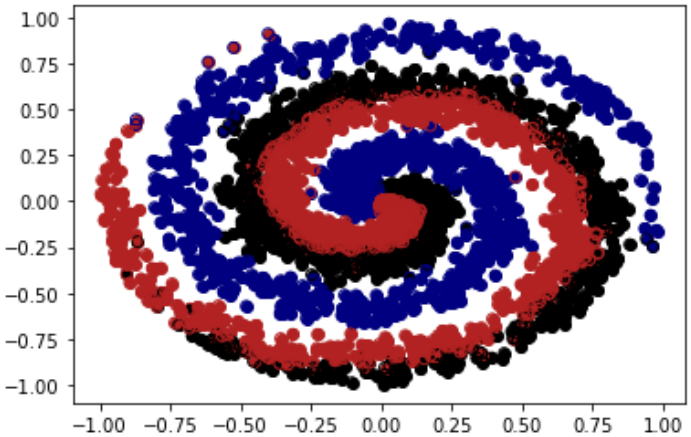
```
idxp0 = np.argwhere(predClass==0)
idxp1 = np.argwhere(predClass==1)
idxp2 = np.argwhere(predClass==2)

plt.scatter(x[idxp0,0],x[idxp0,1], color = 'black')
plt.scatter(x[idxp1,0],x[idxp1,1], color = [0, 0, 0.5])
plt.scatter(x[idxp2,0],x[idxp2,1], color = [0.7, 0.13, 0.13])
```

visualization

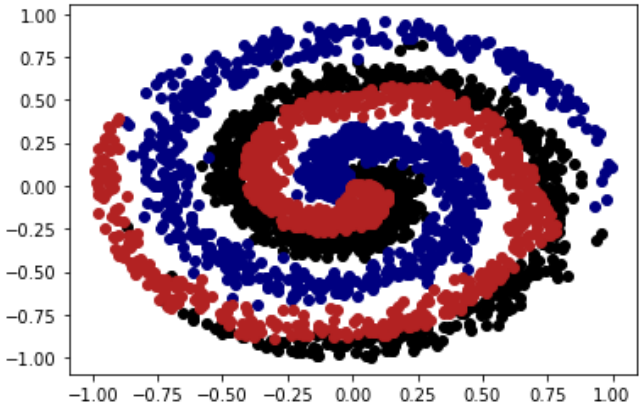
before we are done:

- running the code is still a bit clumsy → calling ANN externally
- creating a function that runs the actual ANN incl plots
- checking predicted classes visually
- extracting the weights for actual application

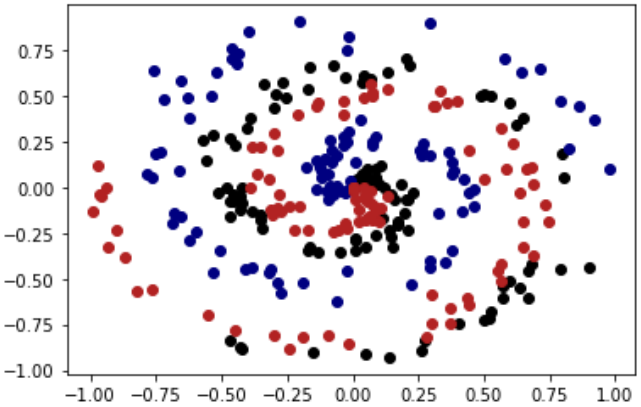
training result for sample size 1000



applied to 1000 data points



applied to 100 data points



our code works now sufficiently well ☺

→ part I is done

<u>part II:</u> - we want the code to be able to ***categorize images*** (cats/ dogs etc)

- therefore it needs to understand features in an image
  → **convolution**