

## Lecture 08:

# Multi-Layer Perceptron (MLP)



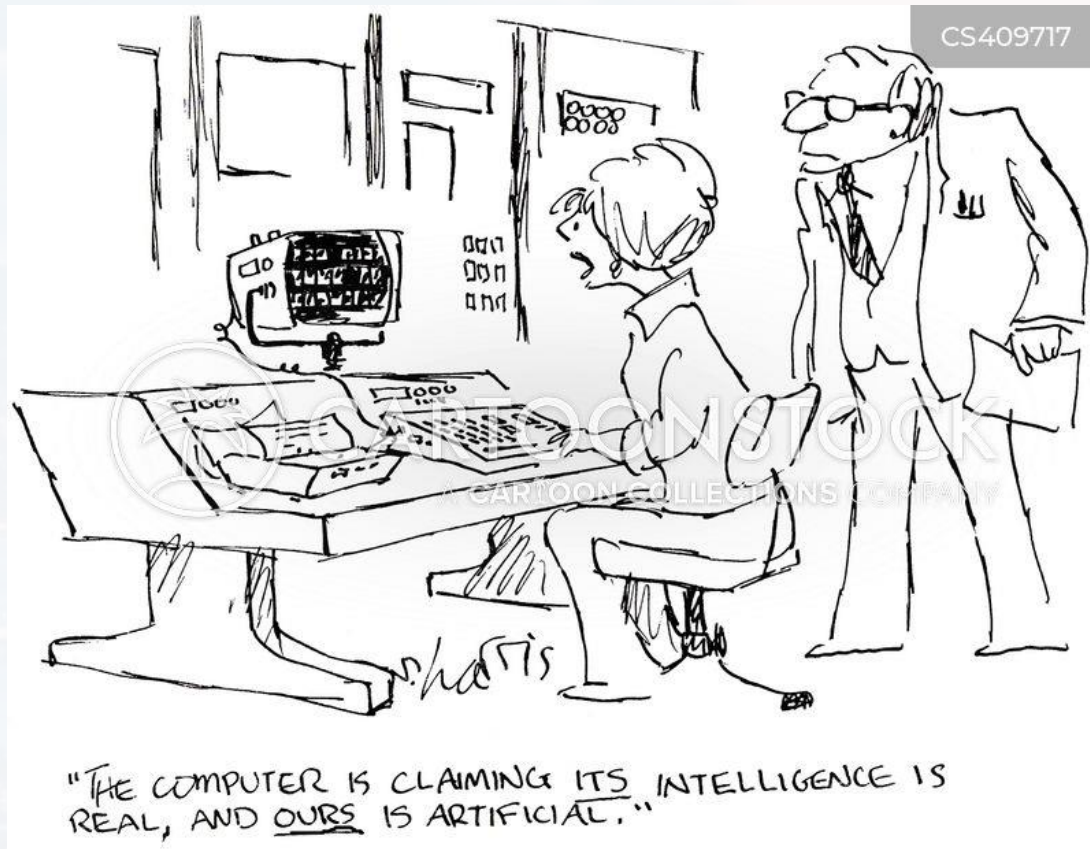
Markus Hohle

University California, Berkeley

**Machine Learning Algorithms**

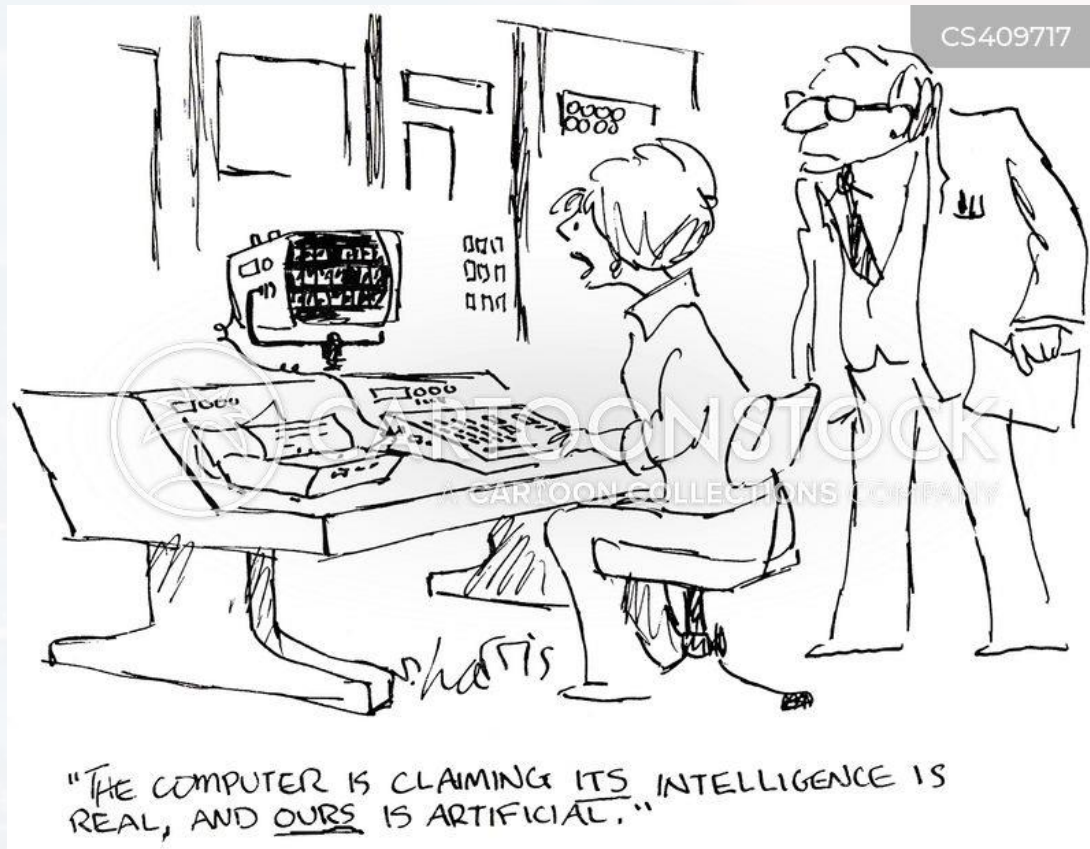
MSSE 277B, 3 Units

Fall 2024



## Outline

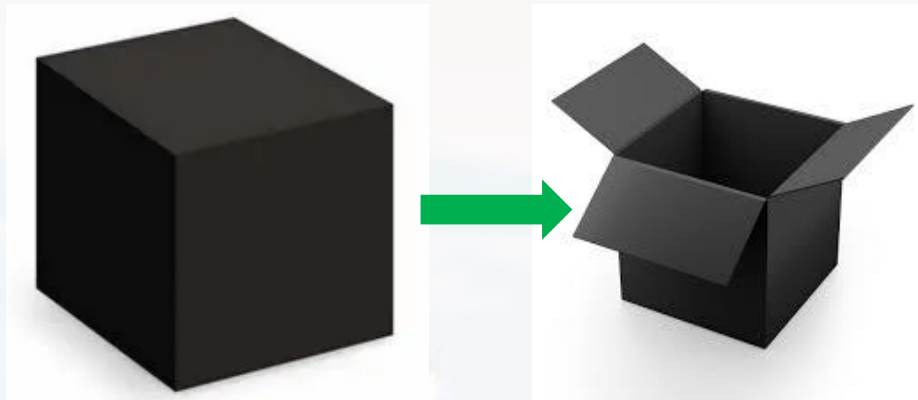
- Motivation & Overview
- The Minimal Setup
- Building a fully functional ANN



## Outline

- Motivation & Overview
- The Minimal Setup
- Building a fully functional ANN





ANNs are **not** black boxes!

### Goal 1: Understanding what “is in the box”

→ important for knowing **which** type of ANN can solve your problem!

```
model = Sequential()
```

```
model.add(Embedding(Nwords, embedding_vector_length,\n                    input_length = max_review_length))
```

```
model.add(Dropout(DA_rate))\nmodel.add(LSTM(n_neurons))\nmodel.add(Dropout(DA_rate))\nmodel.add(Dense(2, activation = 'softmax'))
```

```
opt = optimizers.Adam()\nmodel.compile(loss = 'categorical_crossentropy', optimizer = opt,\n              metrics = ['accuracy'])
```

```
model.summary()
```

### Goal 2: Understanding Python syntax

→ important for **building the specific** ANN that can solve your problem!



### Goal 1: Understanding what “is in the box”

→ important for knowing **which** type of ANN can solve your problem!

### Goal 2: Understanding Python syntax

→ important for **building the specific** ANN that can solve your problem!

→ important for **solving error messages**  
(requires understanding of processes within the ANN)!

I have tried:



```
training_features = numpy.reshape(  
    training_features,  
    (training_features.shape[0], 1, training_features.shape[1]))
```

But I get:

```
ValueError: Input 0 is incompatible with layer lstm_1: expected ndim=3, found ndim=4
```



### Goal 1: Understanding what “is in the box”

→ important for knowing **which** type of ANN can solve your problem!

### Goal 2: Understanding Python syntax

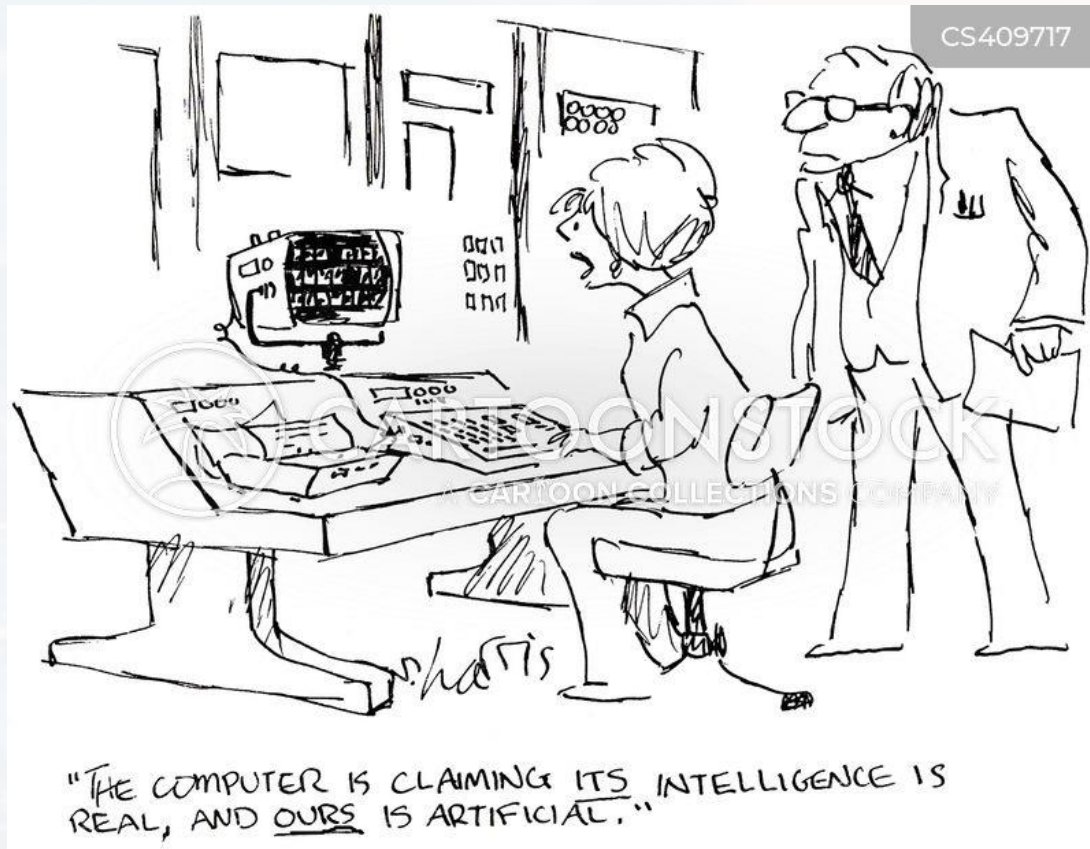
→ important for **building the specific** ANN that can solve your problem!

→ important for **solving error messages**  
(requires understanding of processes within the ANN)!



“What I cannot build. I do not understand.”

— Richard Feynman



## Outline

- Motivation & Overview
- **The Minimal Setup**
- Building a fully functional ANN



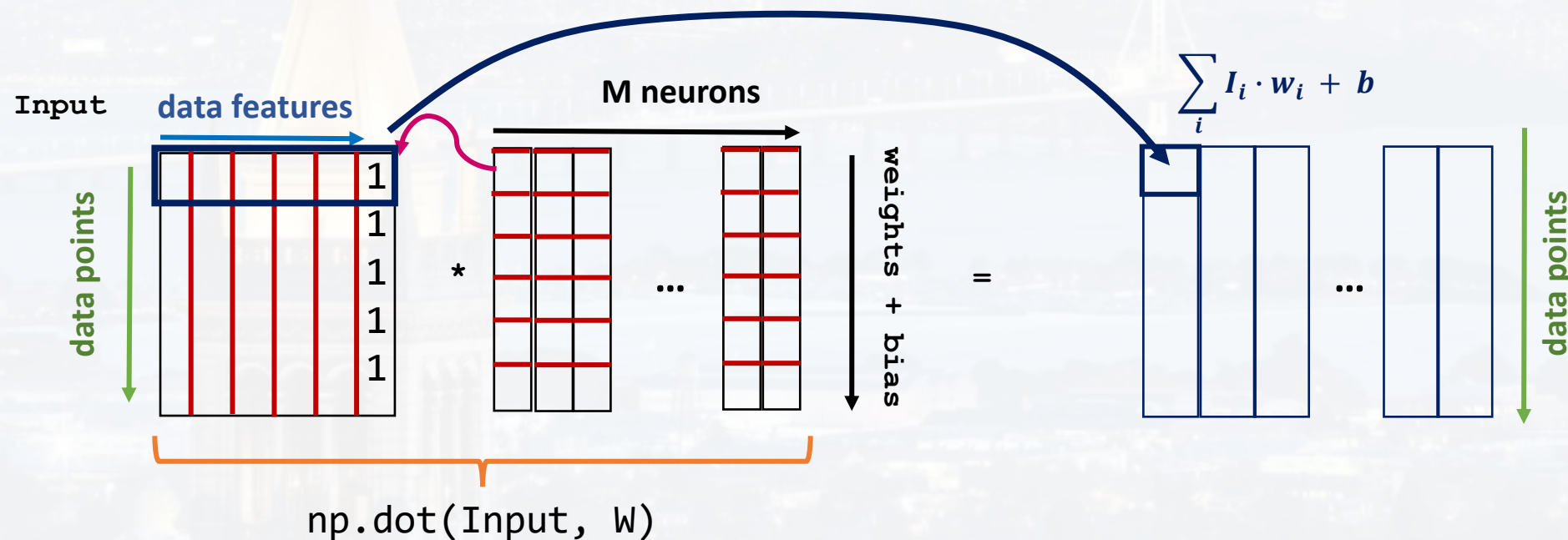


main part of the code:

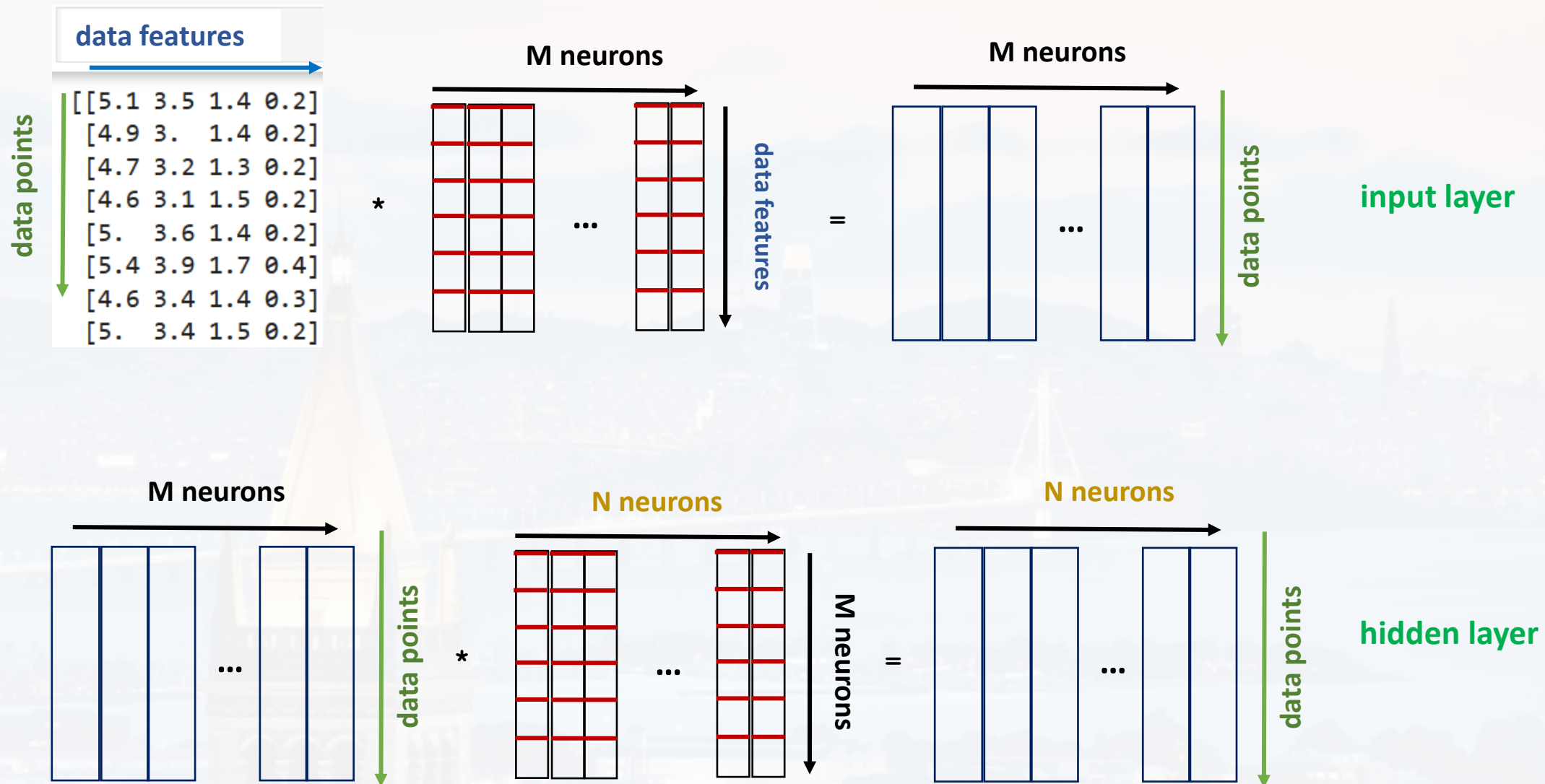
data features				
data points	[	5.1	3.5	1.4 0.2]
	[	4.9	3.	1.4 0.2]
	[	4.7	3.2	1.3 0.2]
	[	4.6	3.1	1.5 0.2]
	[	5.	3.6	1.4 0.2]
	[	5.4	3.9	1.7 0.4]
	[	4.6	3.4	1.4 0.3]
	[	5.	3.4	1.5 0.2]

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

*dot product*

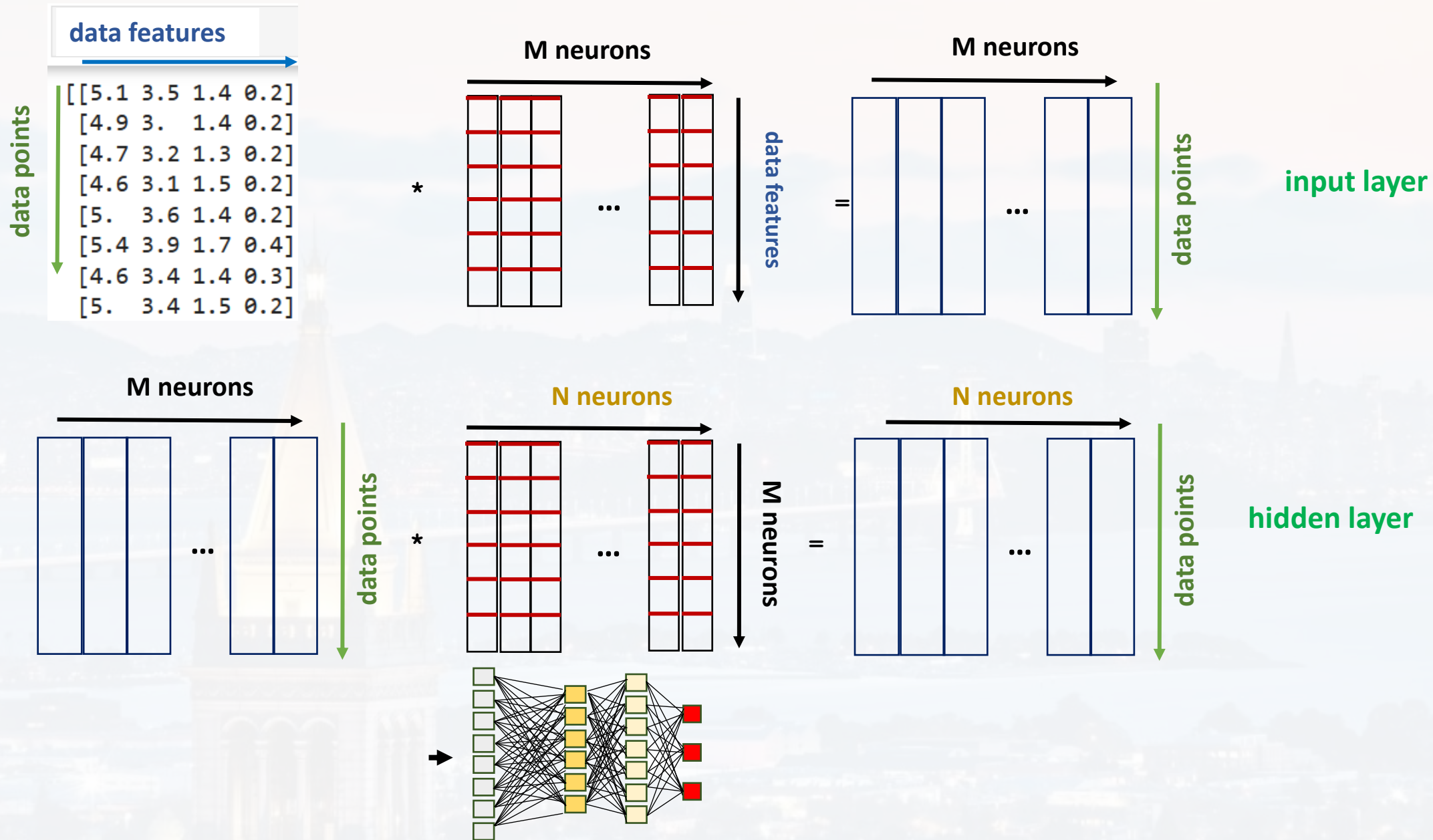








# The Minimal Setup





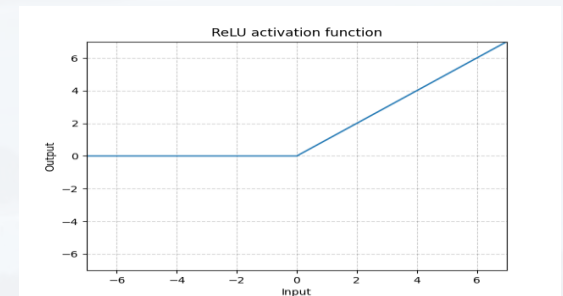
```
class Layer_Dense():
```

```
    def __init__(self, n_inputs, n_neurons):  
        self.weights = np.random.randn(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
```

number of features

```
class Activation_ReLU():
```

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







```
class Layer_Dense():
```

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

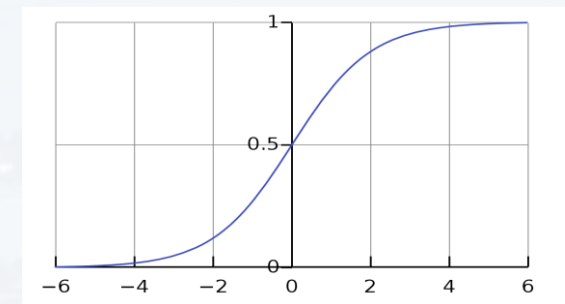
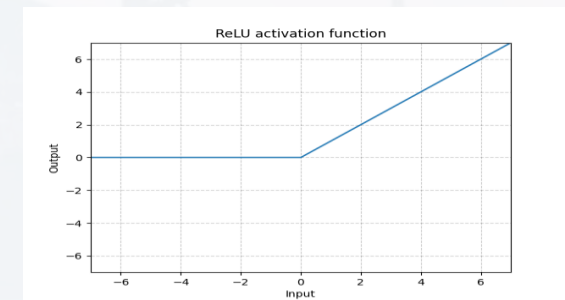
    def forward(self, inputs):
        self.output = np.dot(inputs, self.weights) + self.biases
        self.inputs  = inputs
```

```
class Activation_ReLU():
```

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

```
class Activation_Sigmoid():
```

```
    def forward(self, inputs):
        self.output = np.clip(1/(1 + np.exp(-inputs)), 1e-7, 1-1e-7)
        self.inputs  = inputs
```





Goal: fitting the Spiral Dataset

**two** features, **five** classes

initializing the layers

```
Nneurons1 = 64
```

```
Nfeatures = X.shape[1]
```

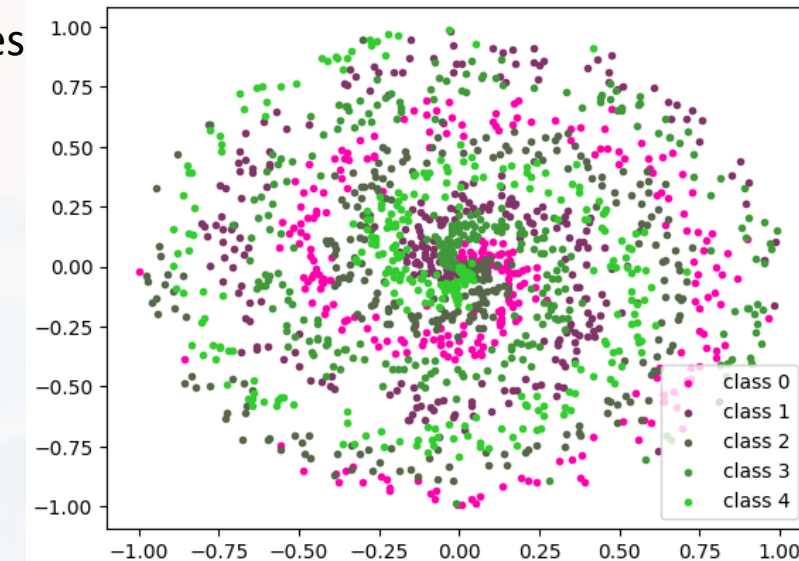
```
dense1 = Layer_Dense(Nfeatures, Nneurons1)
```

```
dense_reg = Layer_Dense(Nneurons1, 1)
```

```
dense_cla = Layer_Dense(Nneurons1, Nclasses)
```

```
ReLU = Activation_ReLU()
```

```
Sigm = Activation_Sigmoid()
```



for regression: one value for each datapoint

for classification: Nclasses values for each datapoint (probabilities for each class)



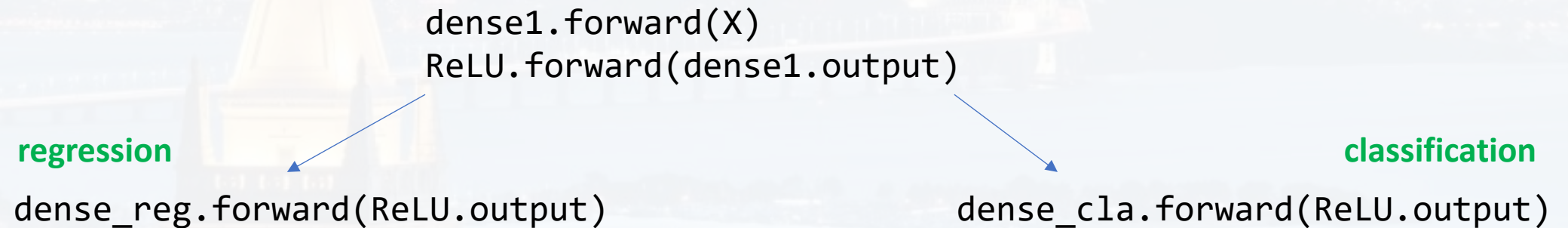
```
Nneurons1 = 64
Nfeatures = X.shape[1]

dense1     = Layer_Dense(Nfeatures, Nneurons1)

dense_reg  = Layer_Dense(Nneurons1, 1)
dense_cla  = Layer_Dense(Nneurons1, Nclasses)

ReLU       = Activation_ReLU()
Sigm       = Activation_Sigmoid()
```

building the **forward** part of the ANN

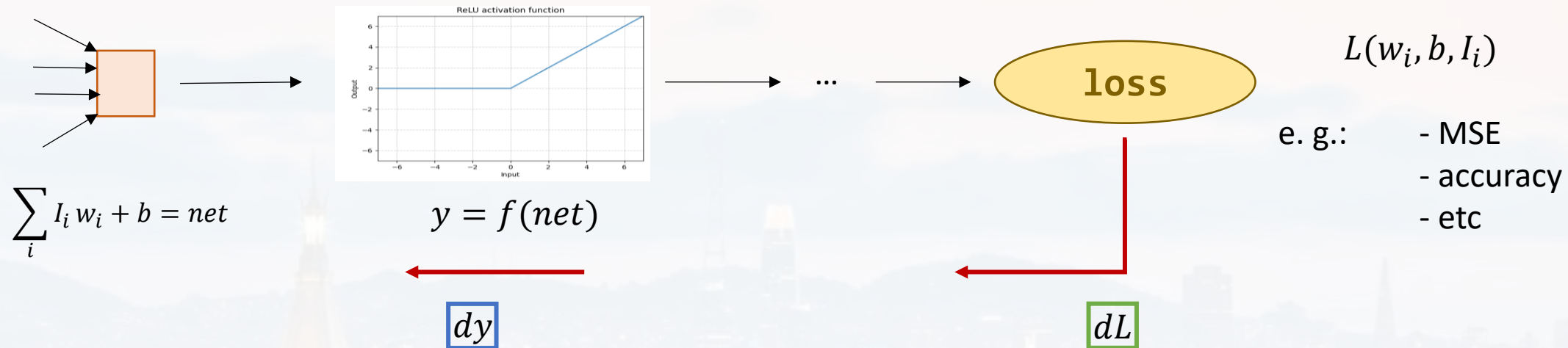


see ANNI.ipynb for details





for training: building the backpropagation part



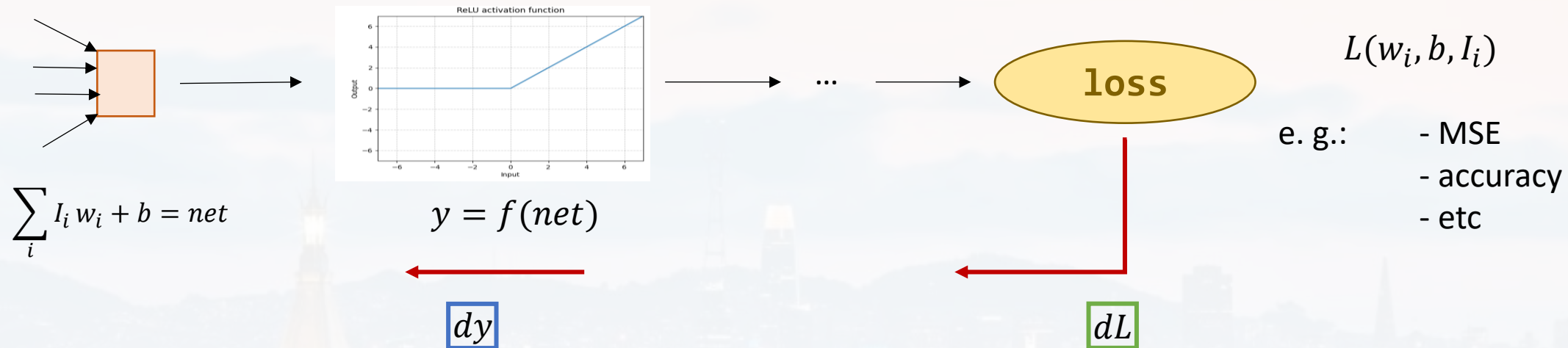
$$\Delta w_i = -\alpha \frac{\partial L}{\partial w_i}$$

$$\Delta w_i = -\alpha \frac{dL}{dy} \frac{dy}{dnet} \frac{\partial net}{\partial w_i}$$

$$\Delta w_i = -\alpha \frac{dL}{dy} \frac{dy}{dnet} I_i$$



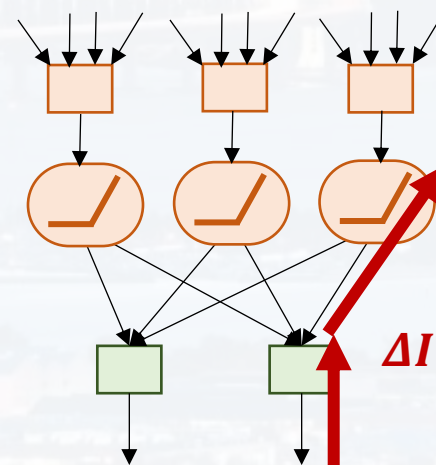
for training: building the backpropagation part



$$\Delta w_i = -\alpha \frac{dL}{dy} \frac{dy}{dnet} I_i$$

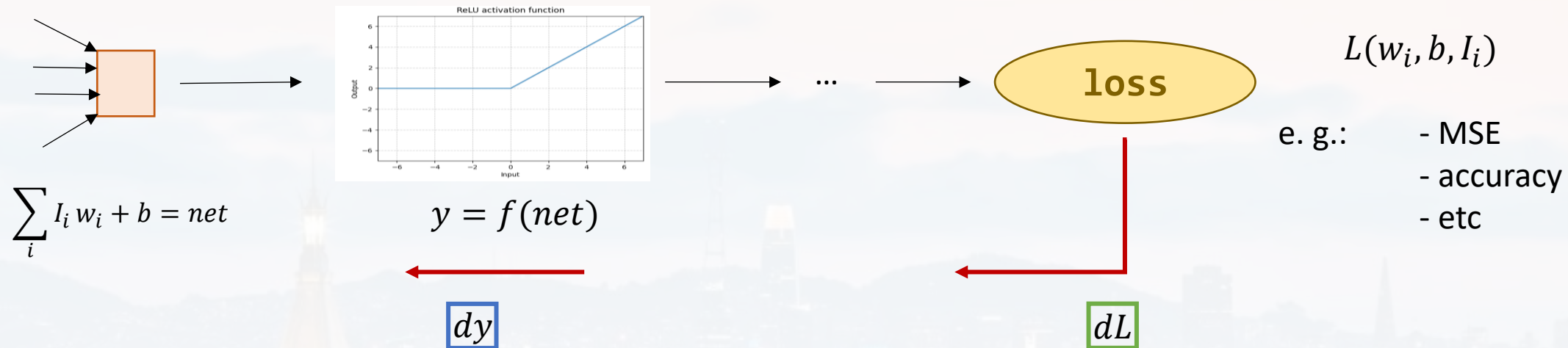
$$\Delta I_i = -\alpha \frac{dL}{dy} \frac{dy}{dnet} w_i \quad \text{from} \quad \frac{\partial L}{\partial I_i}$$

$$\Delta b = -\alpha \frac{dL}{dy} \frac{dy}{dnet} 1 \quad \text{from} \quad \frac{\partial L}{\partial b}$$





for training: building the backpropagation part



$$\Delta w_i = -\alpha \frac{dL}{dy} \frac{dy}{dnet} I_i$$

$$\Delta I_i = -\alpha \frac{dL}{dy} \frac{dy}{dnet} w_i \quad \text{from} \quad \frac{\partial L}{\partial I_i}$$

$$\Delta b = -\alpha \frac{dL}{dy} \frac{dy}{dnet} 1 \quad \text{from} \quad \frac{\partial L}{\partial b}$$

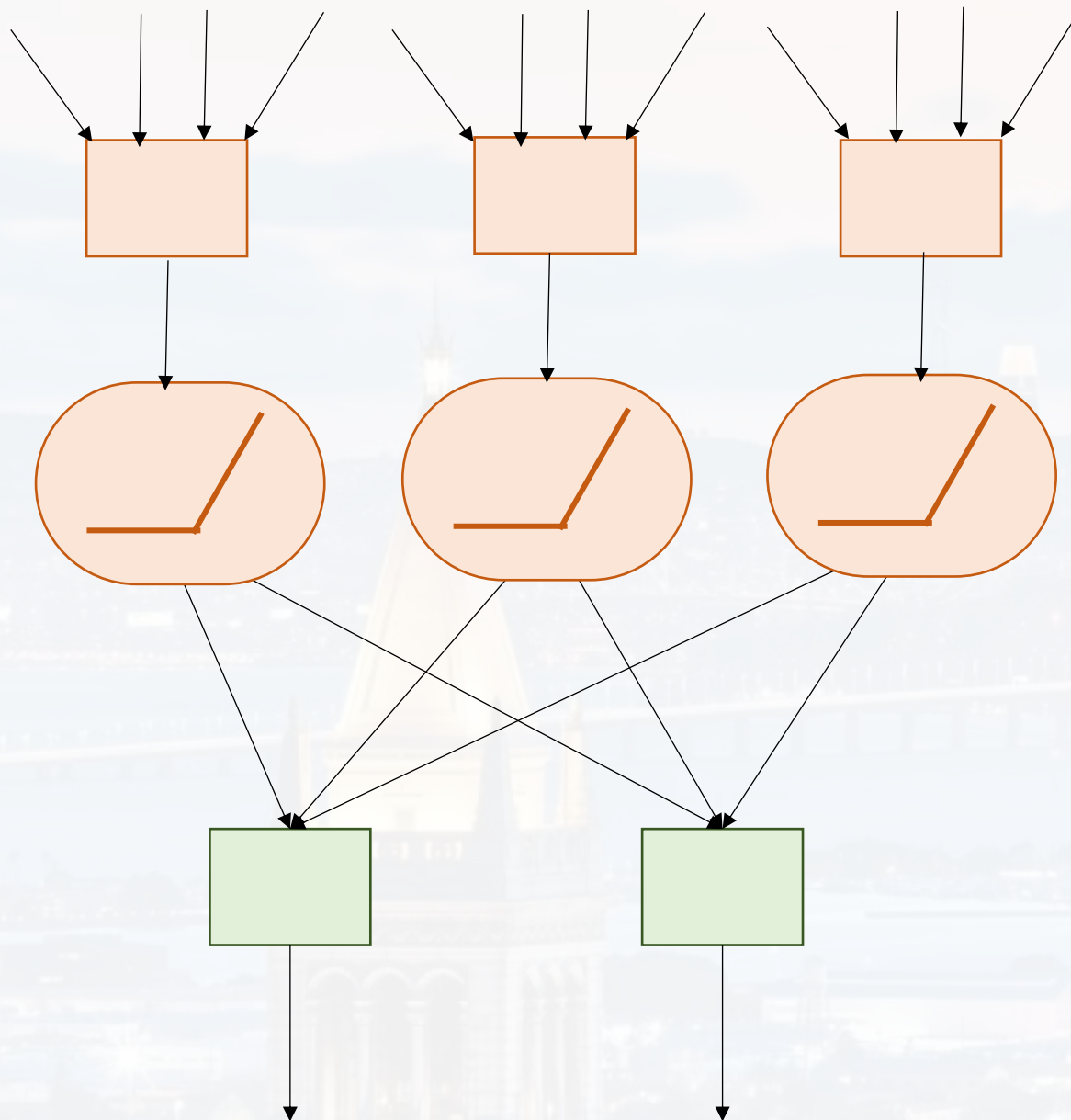
hence, we need to include the following structure for backpropagation:

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

inner  
derivative

product of  
outer  
derivatives



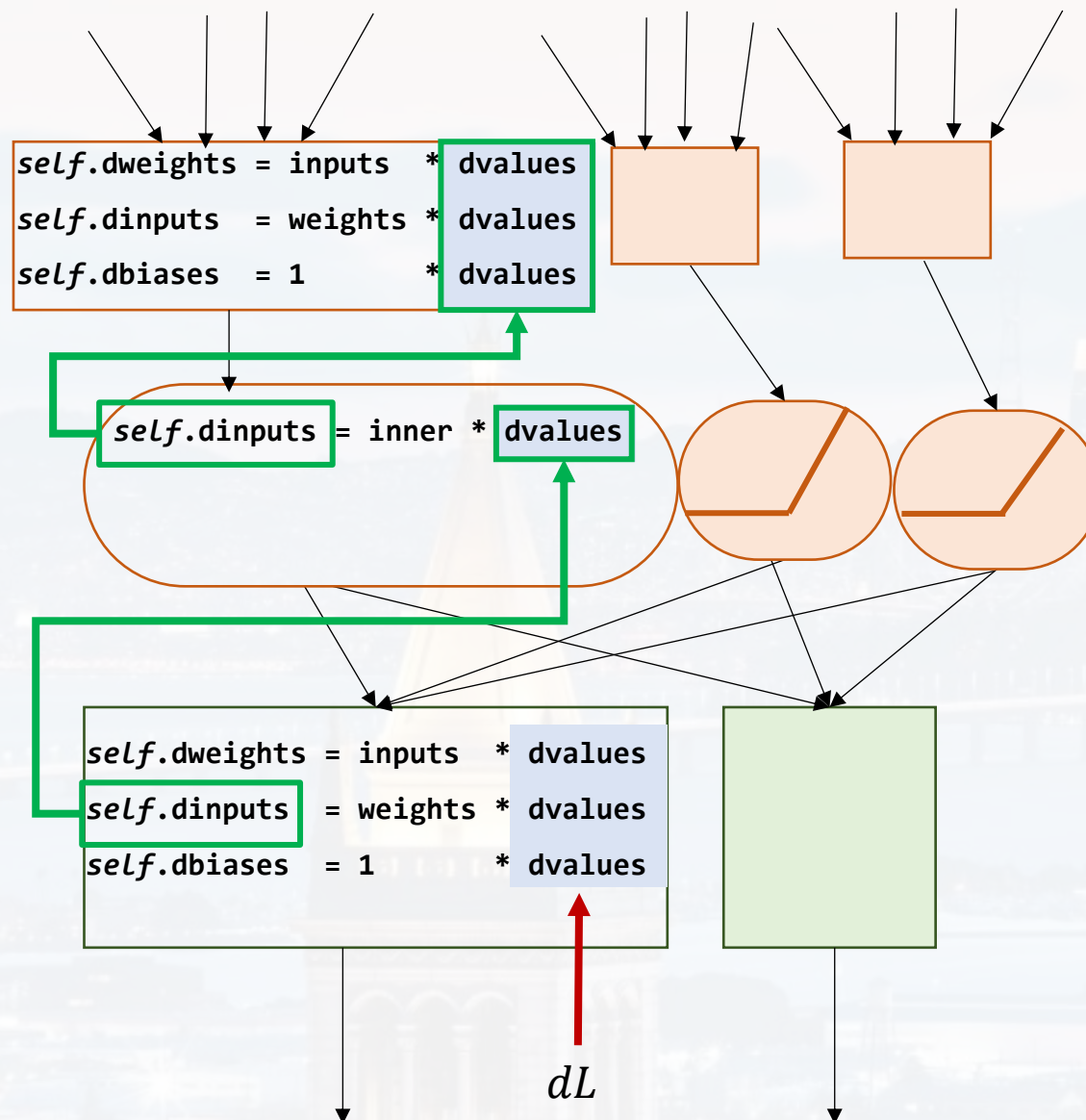


hence, we need to include the following structure for backpropagation:

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

inner  
derivative

product of  
outer  
derivatives



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

$self.dweights = inputs * dvalues$	$self.dinputs = weights * dvalues$	$self.dbiases = 1 * dvalues$
inner derivative		product of outer derivatives

$$\Delta I_i = -\alpha \frac{dL}{dy} \frac{dy}{dnet} w_i$$



for training: building the backpropagation part

```
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)
```

outer derivative

see ANNII.ipynb for details





alpha = 0.001

```
dense1.forward(X)
ReLU.forward(dense1.output)
dense_reg.forward(ReLU.output)
```

forward

```
Ypred = dense_reg.output
dE     = Ypred - Target
MSE    = np.sum(abs(dE))/(Nsample*Nclasses)
print('MSE = ' + str(MSE))
```

evaluation

```
dense_reg.backward(dE)
ReLU.backward(dense_reg.dinputs)
dense1.backward(ReLU.dinputs)
```

backpropagation

```
dense_reg.weights -= alpha * dense_reg.dweights
dense_reg.biases  -= alpha * dense_reg.dbiases
dense1.weights    -= alpha * dense1.dweights
dense1.biases     -= alpha * dense1.dbiases
```

optimization



```
dense1 = Layer_Dense(Nfeatures, Nneurons1)  
dense_reg = Layer_Dense(Nneurons1, 1)
```

print(Ypred)	print(Target)
[2.00173615]	[0]
[2.00173615]	[0]
[2.00173615]	[0]
...	...
[2.00173615]	[4]
[2.00173615]	[4]
[2.00173615]	[4]

for regression: one value for each datapoint

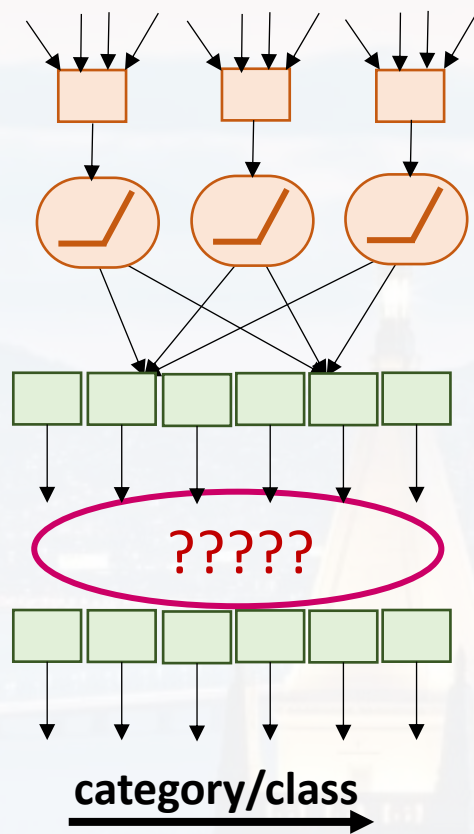
```
dense_cla = Layer_Dense(Nneurons1, Nclasses)
```

for classification: Nclasses values for each datapoint (**probabilities** for each class)



```
dense_cla = Layer_Dense(Nneurons1, Nclasses)
```

for classification: Nclasses values  
for each datapoint  
(**probabilities** for  
each class)



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

**Boltzmann distribution**

aka **softmax**

(from max entropy)



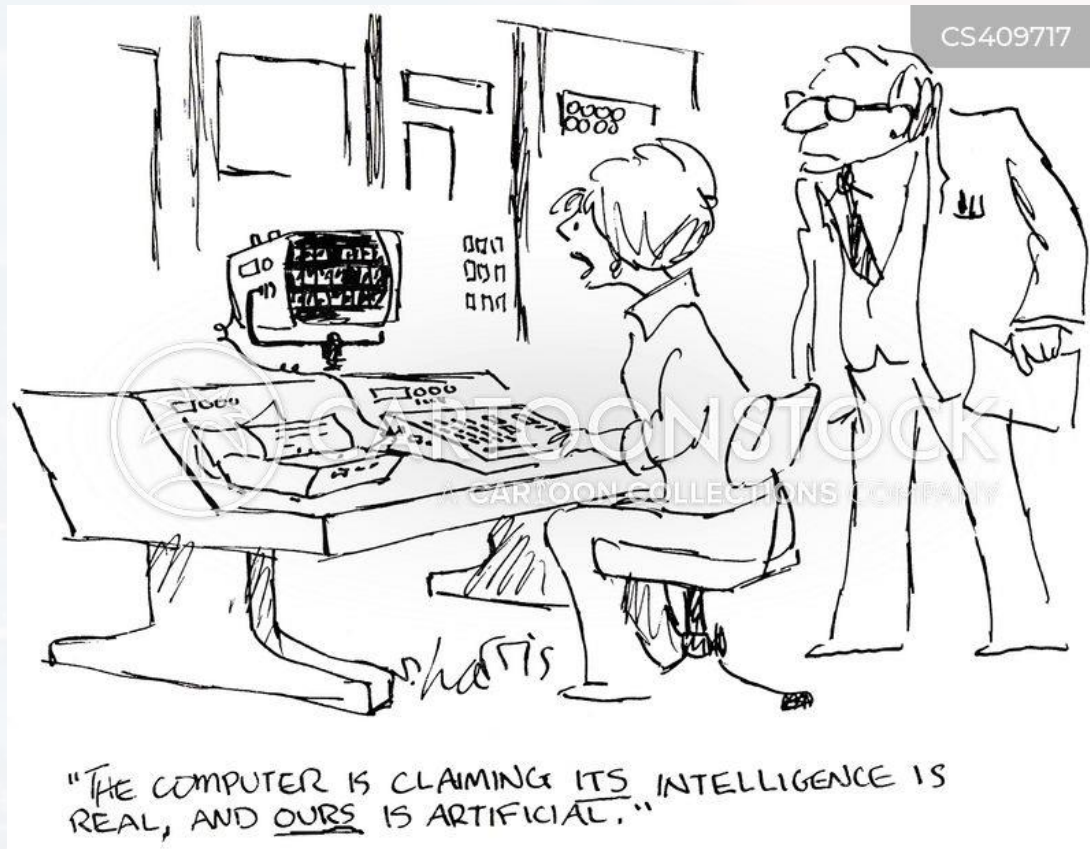
prob\_pred = 
$$\begin{pmatrix} \mathbf{0.8} & 0.1 & 0.1 \\ \mathbf{0.9} & 0.05 & 0.05 \\ 0.1 & 0.15 & \mathbf{0.75} \\ 0.2 & \mathbf{0.7} & 0.1 \\ \dots & \dots & \dots \end{pmatrix}$$

category/class →

↓ samples

$\varepsilon_i$ : output from the last hidden layer





## Outline

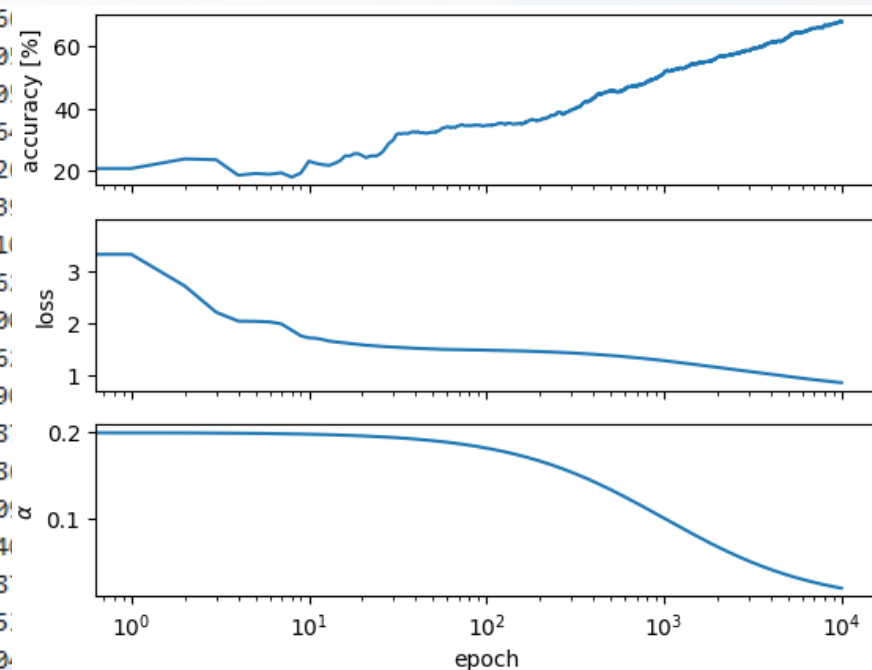
- Motivation & Overview
- The Minimal Setup
- **Building a fully functional ANN**



Finally, we have all the ingredients for a fully functional ANN!

see ANNIII.ipynb for details

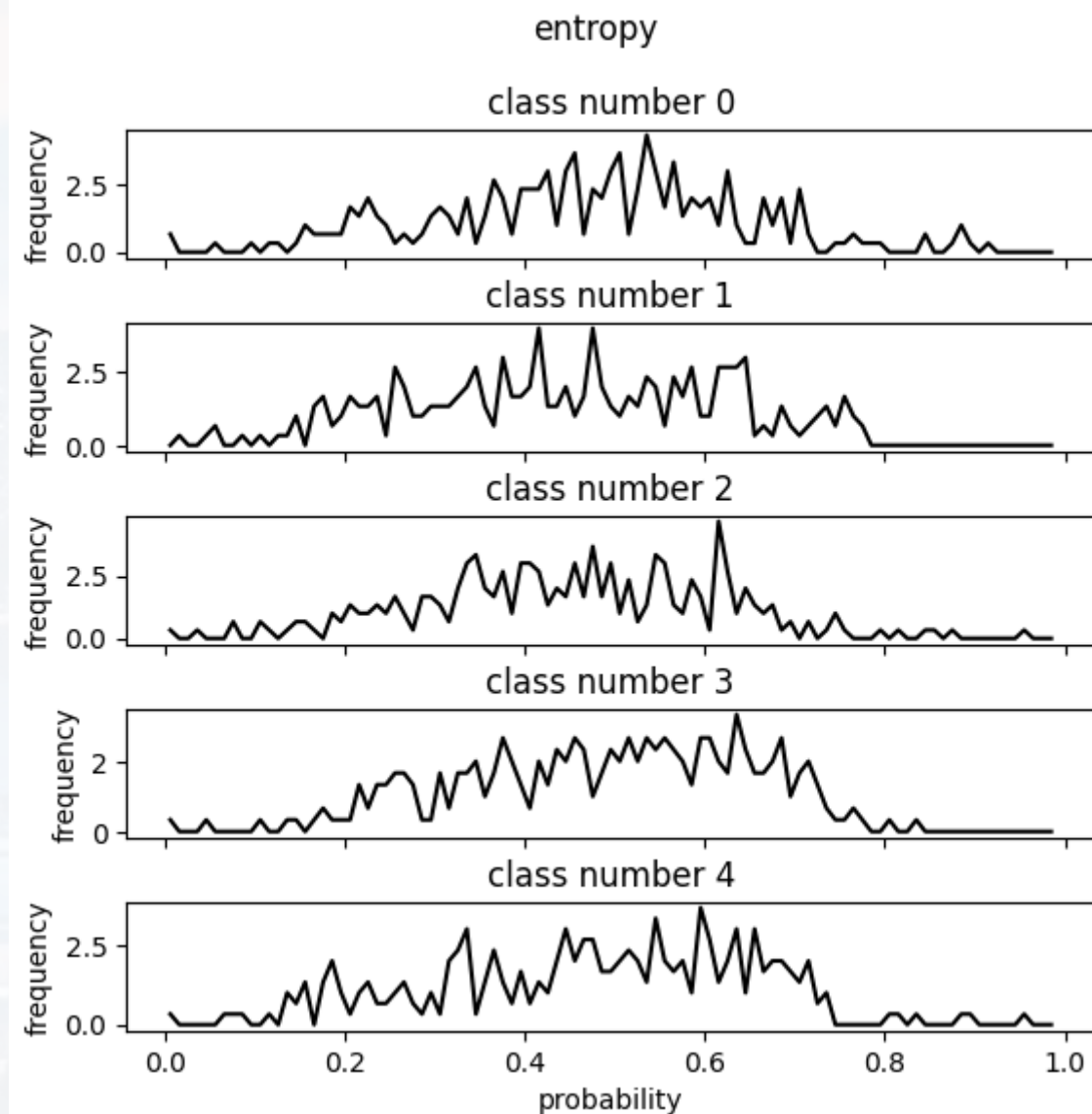
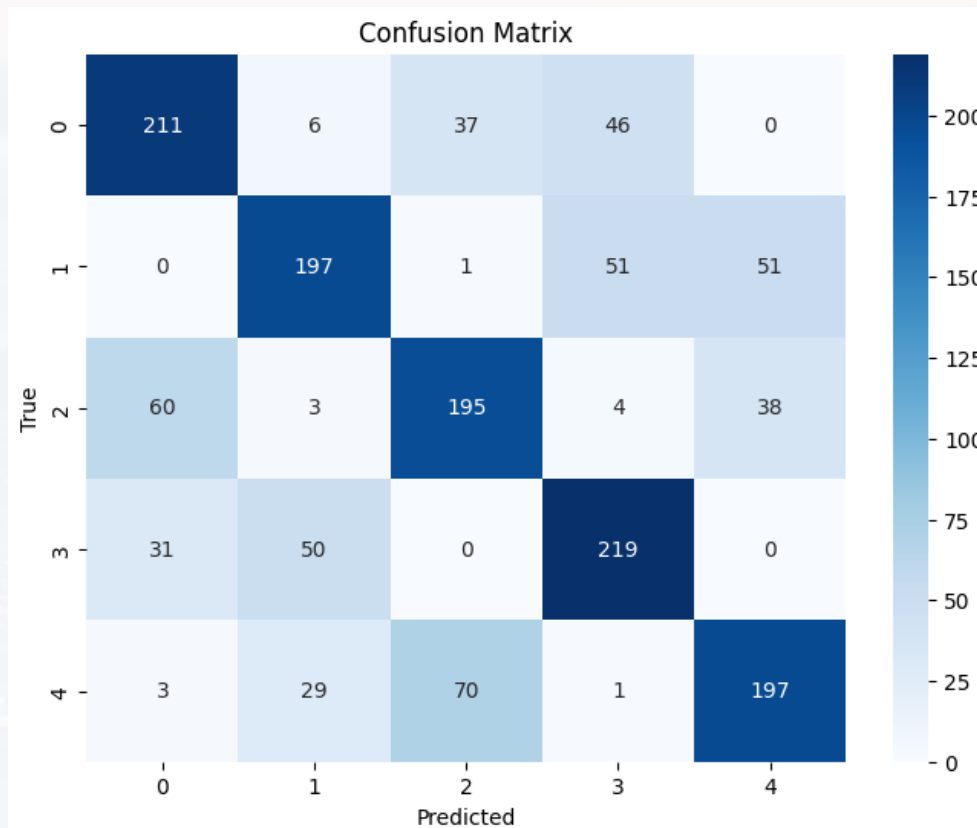
```
epoch: 18200, accuracy: 0.767, loss: 0.662, actual learning rate: 0.010416666666666666
epoch: 18300, accuracy: 0.767, loss: 0.662, actual learning rate: 0.010362694300518135
epoch: 18400, accuracy: 0.768, loss: 0.661, actual learning rate: 0.010309278350518135
epoch: 18500, accuracy: 0.768, loss: 0.661, actual learning rate: 0.010256410256410256
epoch: 18600, accuracy: 0.769, loss: 0.660, actual learning rate: 0.010204081632653061
epoch: 18700, accuracy: 0.769, loss: 0.660, actual learning rate: 0.010152284263959342
epoch: 18800, accuracy: 0.770, loss: 0.660, actual learning rate: 0.010101010101010101
epoch: 18900, accuracy: 0.770, loss: 0.659, actual learning rate: 0.010050251256281407
epoch: 19000, accuracy: 0.769, loss: 0.659, actual learning rate: 0.010000000000000000
epoch: 19100, accuracy: 0.769, loss: 0.658, actual learning rate: 0.009950248756218905
epoch: 19200, accuracy: 0.769, loss: 0.658, actual learning rate: 0.009900990099009901
epoch: 19300, accuracy: 0.769, loss: 0.658, actual learning rate: 0.009852216748768418
epoch: 19400, accuracy: 0.769, loss: 0.657, actual learning rate: 0.009803921568627451
epoch: 19500, accuracy: 0.769, loss: 0.657, actual learning rate: 0.00975609756097561
epoch: 19600, accuracy: 0.769, loss: 0.656, actual learning rate: 0.009708737864318183
epoch: 19700, accuracy: 0.768, loss: 0.656, actual learning rate: 0.00966183574850209
epoch: 19800, accuracy: 0.769, loss: 0.656, actual learning rate: 0.009615384615384615
epoch: 19900, accuracy: 0.770, loss: 0.655, actual learning rate: 0.009569377990654205
```





Finally, we have all the ingredients for a fully functional ANN!

see `ANNIII.ipynb` for details

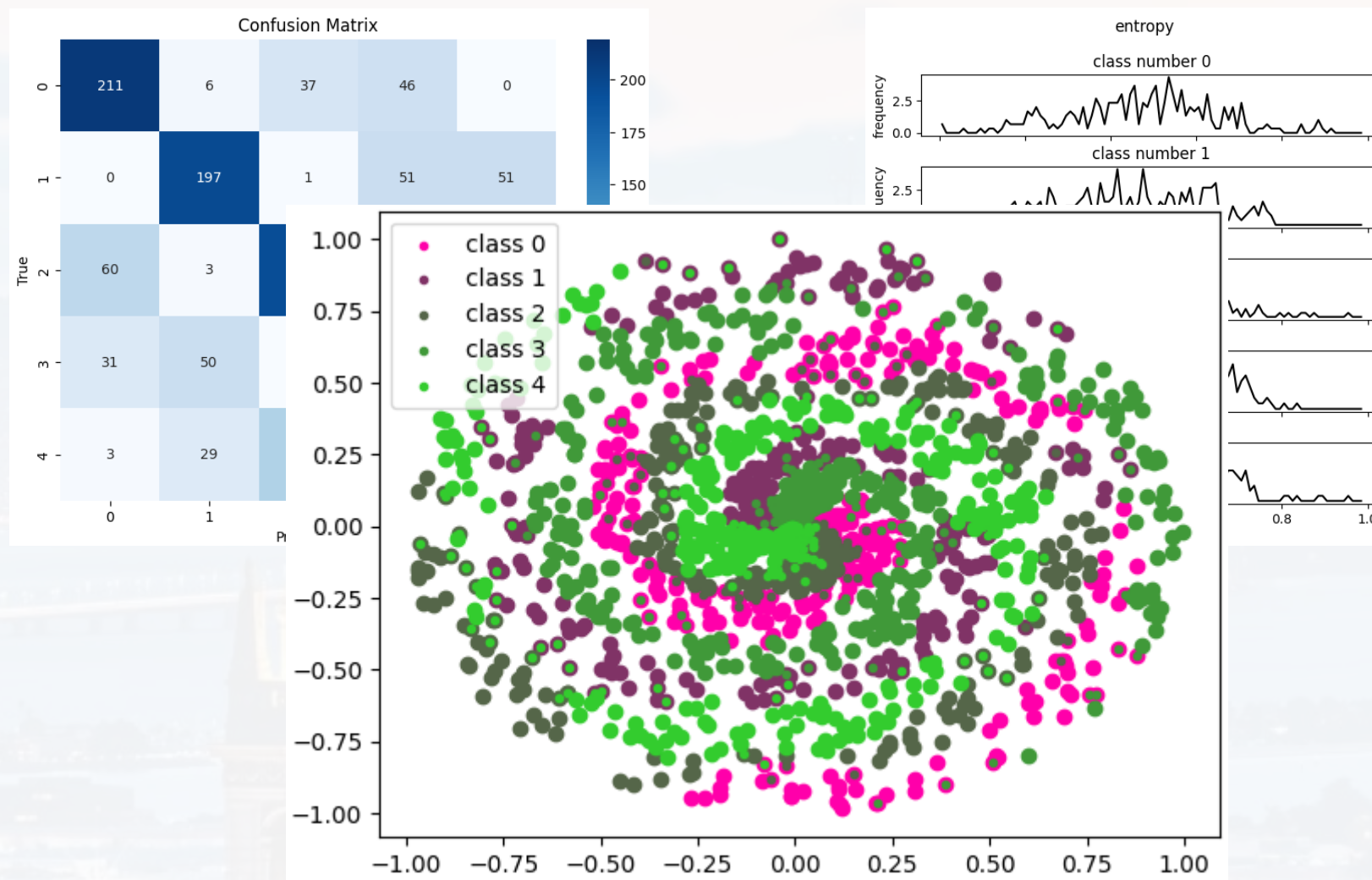






Finally, we have all the ingredients for a fully functional ANN!

see `ANNIII.ipynb` for details



Thank you very much for your attention!

