Lecture 10:

Fully Functional ANN

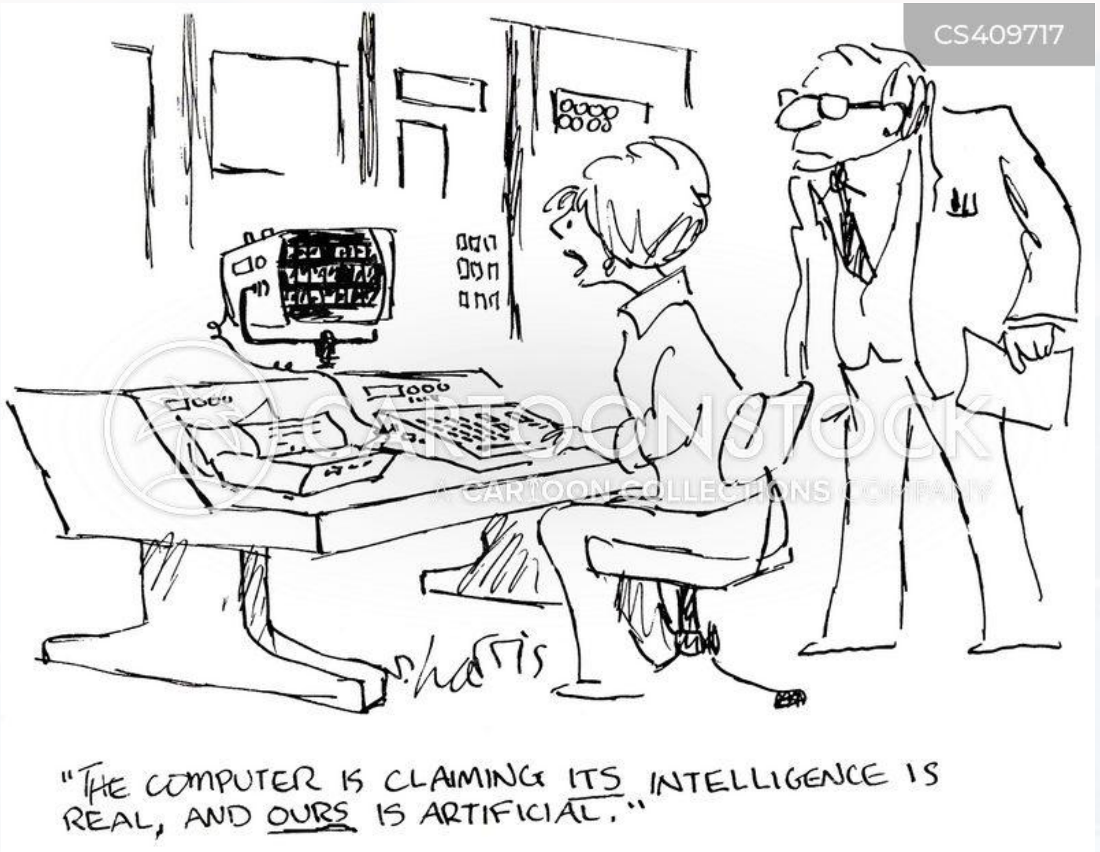
Markus Hohle

University California, Berkeley

Bayesian Data Analysis and Machine Learning for Physical Sciences

Course Map

Fully Functional ANN

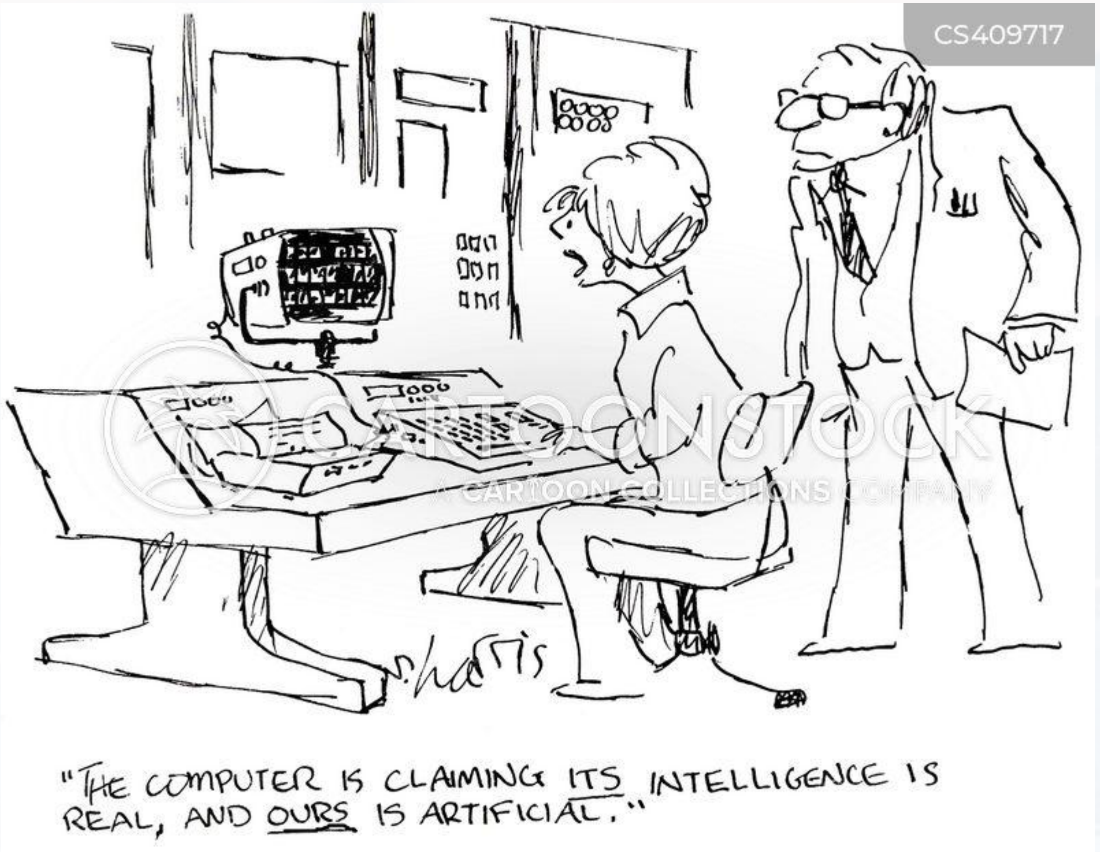


Outline

Softmax Layer & Classification

Backpropagation Again

Fully Functional ANN



Outline

Softmax Layer & Classification

Backpropagation Again

Fully Functional ANN

Fully Functional ANN

basic structure:

```
alpha = 0.001 #learning rate
```

forward

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

evaluation

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

backpropagation

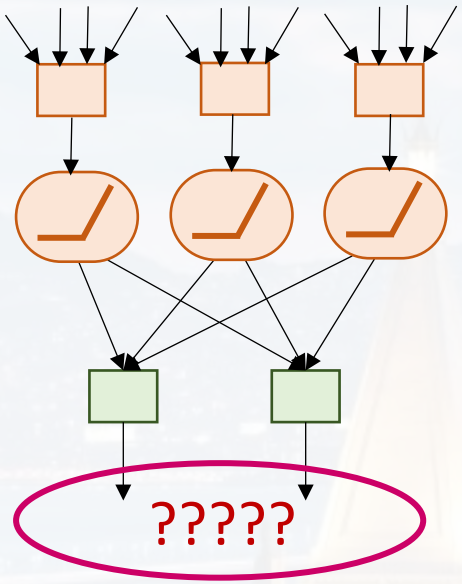
```
dense_reg.backward(dE)
ReLU.backward(dense_reg.dinputs)
dense1.backward(ReLU.dinputs)
```

optimization

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

see `ANNI.ipynb` and `ANNII.ipynb`

regression: output layer, **one neuron** votes for each datapoint, i.e. only **one value**
classification: output layer, ***k*** classes, ***k* neurons** vote for each datapoint, i.e. ***k* values**



How to assign probabilities p_i to the outputs ε_i of the last layer?

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

Boltzmann (aka **softmax**) distribution

often rescaled in order to avoid overflow

```
class Activation_Softmax:

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

regression: output layer, **one neuron** votes for each datapoint, i.e. only **one value**

classification: output layer, ***k*** classes, ***k* neurons** vote for each datapoint, i.e. ***k* values**

target **y** could be encoded in different ways

"one hot"

y true

$$\begin{array}{c|c} \text{category/class} \rightarrow & \\ \begin{bmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{bmatrix} & \downarrow \text{samples} \end{array}$$

"sparse"

$$\begin{bmatrix} 1 \\ 1 \\ 3 \\ 2 \end{bmatrix} \downarrow \text{samples}$$

y predicted (after softmax)

$$\begin{array}{c|c} \text{category/class} \rightarrow & \\ \begin{pmatrix} 0.8 & 0.10 & 0.10 \\ 0.9 & 0.05 & 0.05 \\ 0.10 & 0.15 & 0.75 \\ 0.20 & 0.70 & 0.10 \end{pmatrix} & \downarrow \text{samples} \end{array}$$

$$\underbrace{\begin{pmatrix} 0.8 & 0.10 & 0.10 \\ 0.9 & 0.05 & 0.05 \\ 0.10 & 0.15 & 0.75 \\ 0.20 & 0.70 & 0.10 \end{pmatrix} - \begin{pmatrix} 1 & 0 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 1 & 0 \end{pmatrix}}_{\text{outermost derivative}}$$

Fully Functional ANN

Softmax Layer & Classification

how *confident* is the ANN about its decision? → **cross entropy**

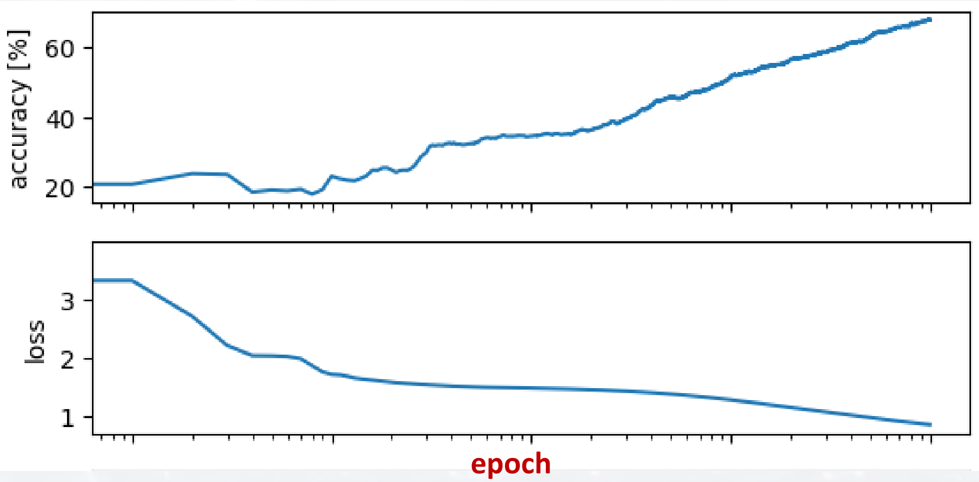
$$p_i = \frac{\exp(\varepsilon_i)}{\sum_i \exp(\varepsilon_i)}$$

for each data point (after `softmax` layer):

$$\begin{array}{lcccccc} p_i = & 0.1 & 0.05 & 0.2 & 0.0 & 0.45 & 0.2 \\ p\_true_i = & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

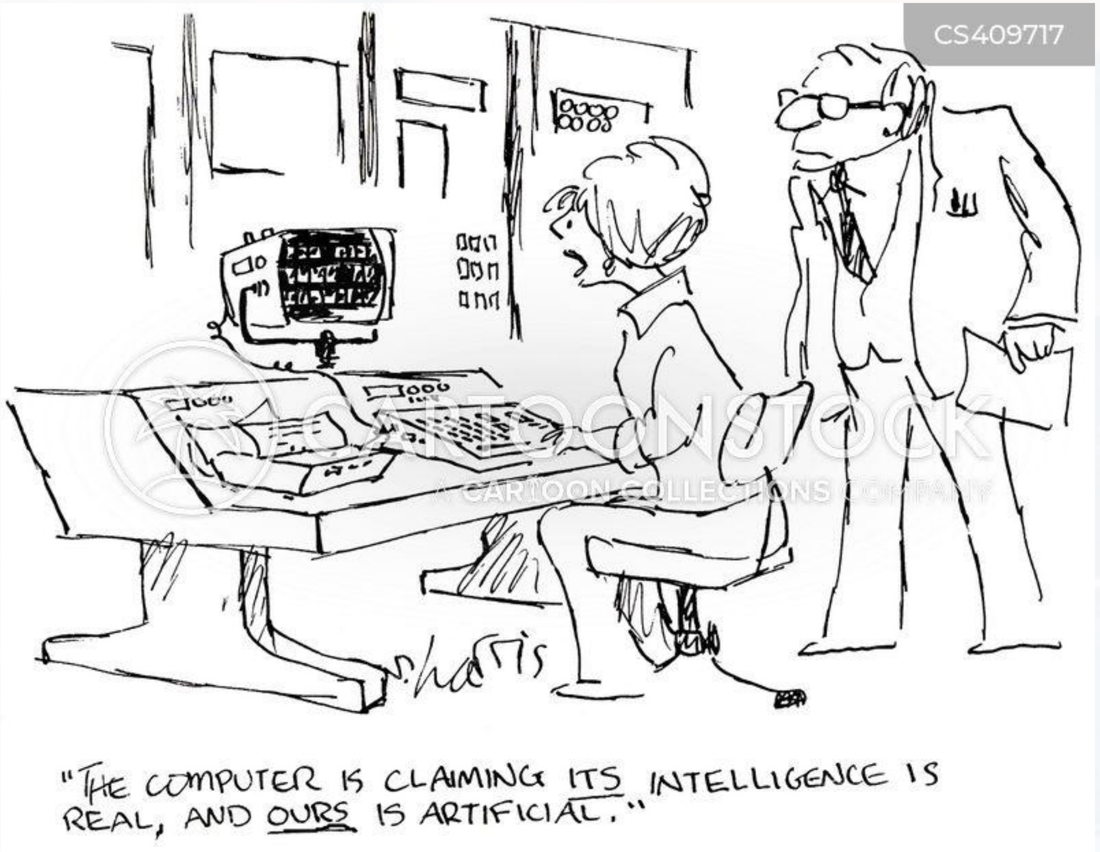
$$S = -\sum_i p\_true_i \cdot \ln p_i$$

two quality criteria: **accuracy** and **cross entropy**



mean over all samples: total ***Loss***

- categorization: **mean of cross entropy**
- regression: RMSE, MSE etc



Outline

Softmax Layer & Classification

Backpropagation Again

Fully Functional ANN

Fully Functional ANN

***i*:**	**index over class**
***j*:**	**index over datapoints**

ε_1 ε_2 ε_3 ε_4 ε_5 ε_6

last (= output) layer

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

softmax layer

p_1 p_2 p_3 p_4 p_5 p_6

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

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

outermost derivative $d\mathcal{L}$

thus, we need to calculate:

$$\partial\mathcal{L} = \boxed{\frac{\partial p}{\partial \varepsilon}} \ldots \partial w$$

$$\partial\mathcal{L} = \boxed{\frac{\partial p}{\partial \varepsilon}} \ldots \partial I$$

$$\partial\mathcal{L} = \boxed{\frac{\partial p}{\partial \varepsilon}} \ldots \partial b$$

say for three classes and four data points:

$$d\mathcal{L} = \begin{pmatrix} \mathbf{0.8-1} & 0.10 & 0.10 \\ \mathbf{0.9-1} & 0.05 & 0.05 \\ 0.10 & 0.15 & \mathbf{0.75-1} \\ 0.20 & \mathbf{0.70-1} & 0.10 \end{pmatrix}$$

Fully Functional ANN

i:	index over class
j:	index over datapoints

We derived the backpropagation from the last output layer on last time.

Now, we add the softmax layer, that **takes the output from the last layer** and **returns probabilities**!

$$\partial \mathcal{L} = \boxed{\frac{\partial p}{\partial \varepsilon}} \ldots \partial w$$

$$\partial \mathcal{L} = \boxed{\frac{\partial p}{\partial \varepsilon}} \ldots \partial I$$

$$\partial \mathcal{L} = \boxed{\frac{\partial p}{\partial \varepsilon}} \ldots \partial b$$

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

equals zero for i $\neq k$

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

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

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

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

Fully Functional ANN

***i*:**	**index over class**
***j*:**	**index over datapoints**

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

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

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

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

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

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

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

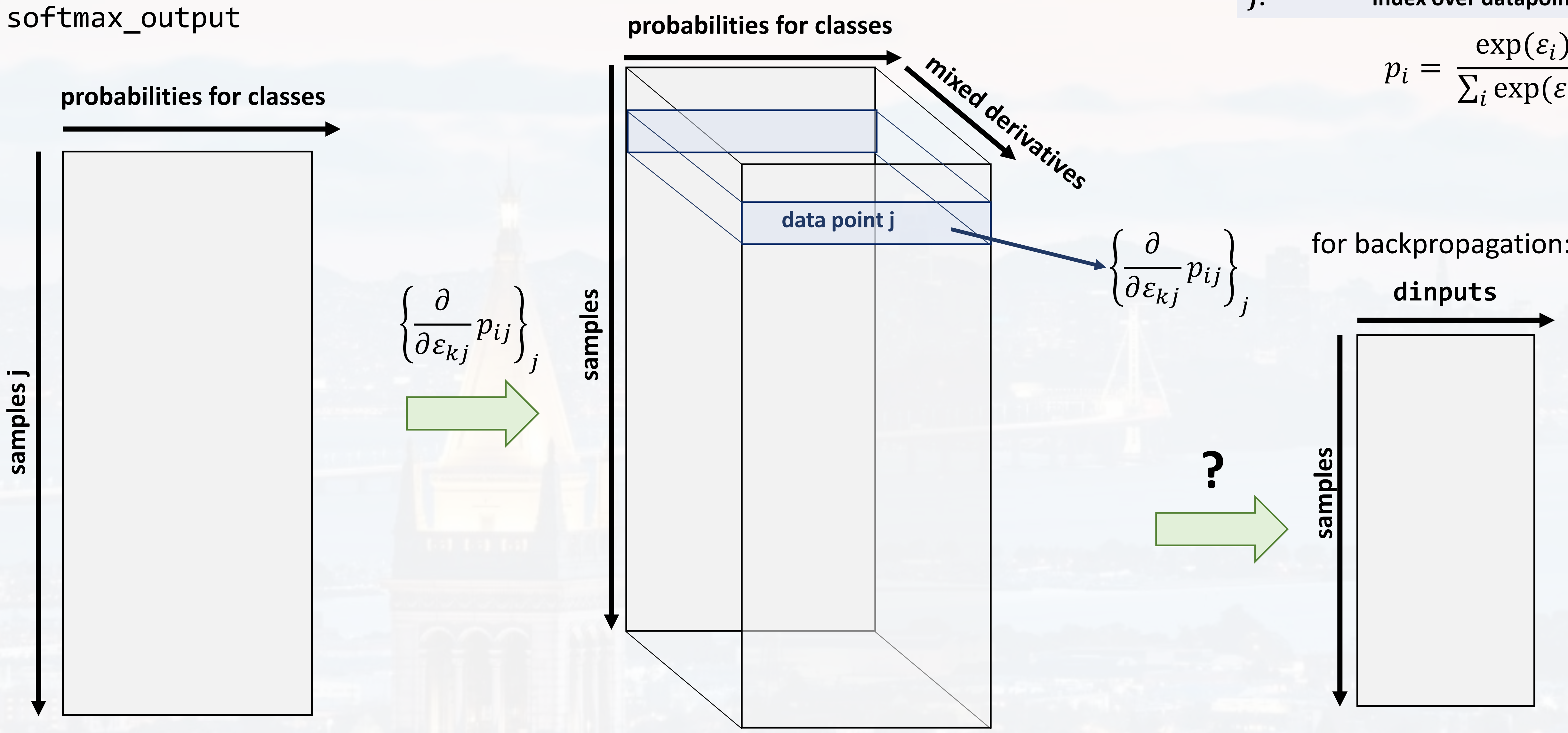
$$\left\{\frac{\partial}{\partial \varepsilon_{kj}} p_{ij}\right\}_j$$ **Jacobian matrix**

Fully Functional ANN

Backpropagation Again

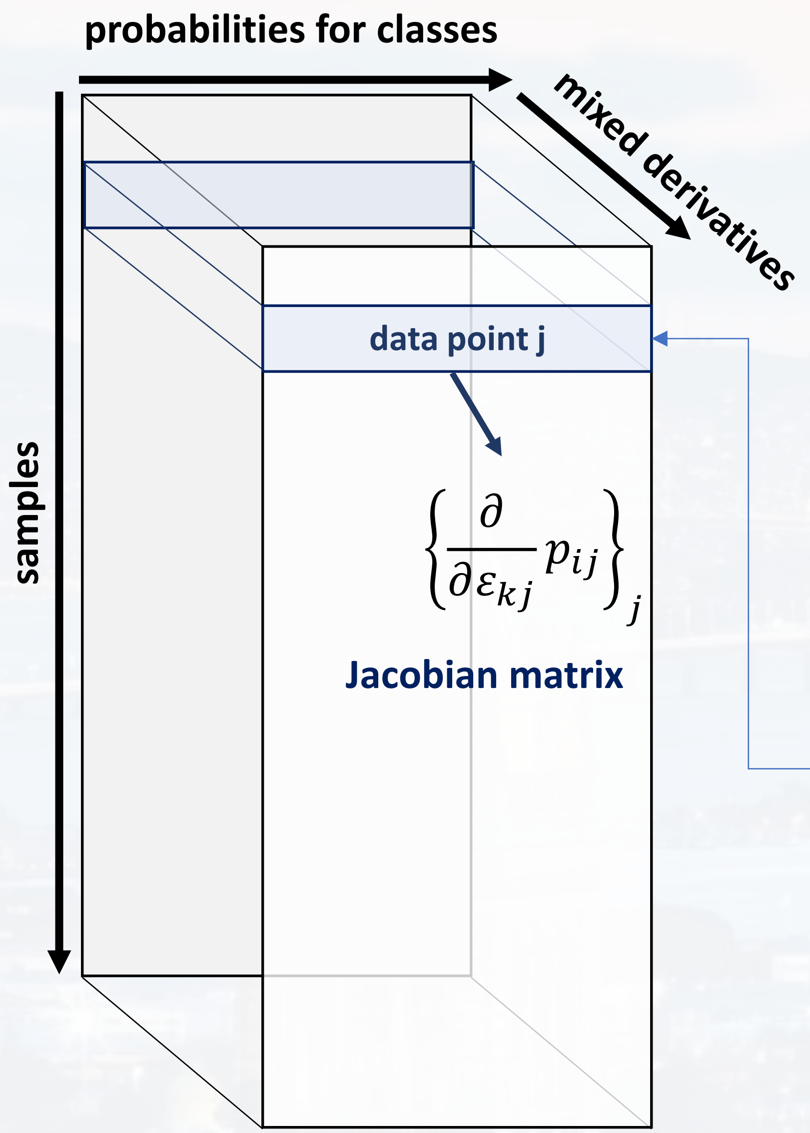
i:	index over class
j:	index over datapoints

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



We are getting a Jacobian matrix for each data point *j*!

$\boldsymbol{i}$:	index over class
$\boldsymbol{j}$:	index over datapoints
$\boldsymbol{K}$:	number of classes
$\boldsymbol{N}$:	number of data points



`dvalues:`

$$d\mathcal{L} = \begin{pmatrix} \mathbf{0.8-1} & 0.10 & 0.10 \\ \mathbf{0.9-1} & 0.05 & 0.05 \\ 0.10 & 0.15 & \mathbf{0.75-1} \\ 0.20 & \mathbf{0.70-1} & 0.10 \end{pmatrix}$$

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

according to the chain rule:

`dvalues` from the loss function, hence $d\mathcal{L}$, which is a vector of length K for each datapoint j; i. e. of shape N x K...

...must get multiplied with the Jacobian (= inner derivative) which is a matrix of shape KxK for each data point *j*

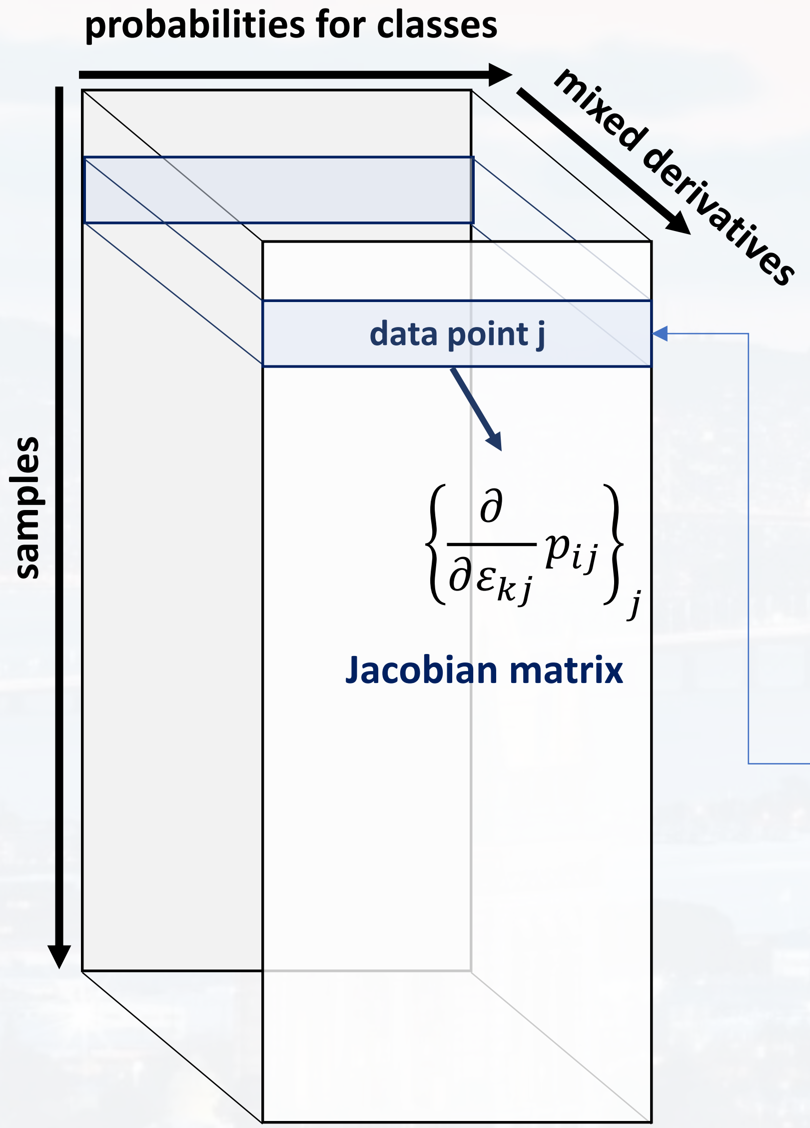
```
np.dot(jacobian_matrix, dvalues)
```

thus each p_i gets influenced by all the other *i = 1, ..., k, ...K*

which results in a matrix for $d\varepsilon$, `dinputs`, of shape NxK

We are getting a Jacobian matrix for each data point j!

i:	index over class
j:	index over datapoints
K:	number of classes
N:	number of data points

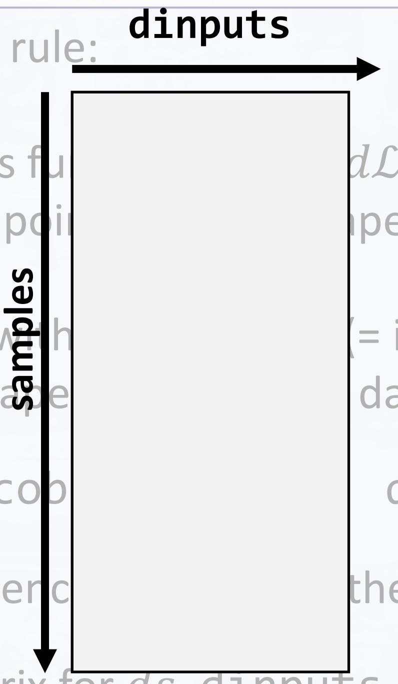


dvalues:

$$d\mathcal{L} = \begin{pmatrix} 0.8-1 & 0.10 & 0.10 \\ 0.9-1 & 0.05 & 0.05 \\ 0.10 & 0.15 & 0.75-1 \\ 0.20 & 0.70-1 & 0.10 \end{pmatrix} \qquad p_i = \frac{\exp(\varepsilon_i)}{\sum_i \exp(\varepsilon_i)}$$

for backpropagation:

according to the chain rule:



dvalues from the loss fu… $d\mathcal{L}$, which is a vector of length K for each datapoi… …pe N x K…

…must get multiplied w… (= inner derivative) which is a matrix of sha… data point j

np.dot(jacob… dvalues)

thus each p_i gets influenc… …ther i = 1, …, k, …K

which results in a matrix for $d\varepsilon$, dinputs, of shape NxK

i:	index over class
j:	index over datapoints
K:	number of classes
N:	number of data points

ε_{ij} from the last dense layer

softmax

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

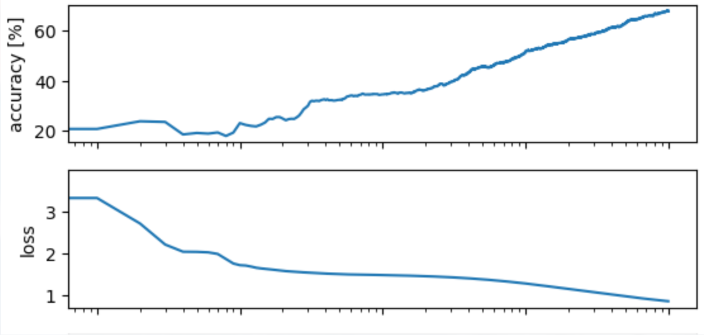
backpropagation

loss

$$S_j = -\log(p_j)$$

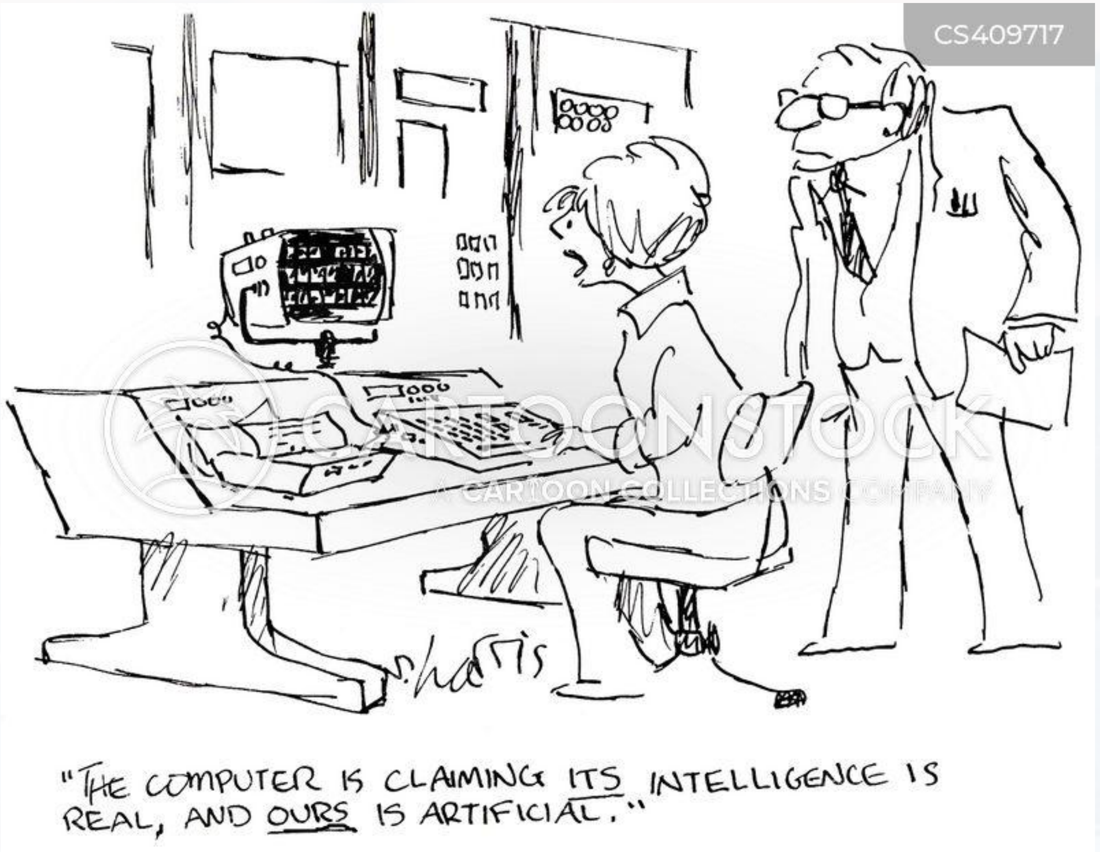
returns total (mean) loss

$$\langle L \rangle = -\frac{1}{N} \sum_{j=1}^{N} \sum_{i=1}^{K} p\_true_i \cdot \ln p_i$$



true class

$$\underbrace{\begin{pmatrix} 0.20 & \mathbf{0.70} & 0.10 \\ \mathbf{0.90} & 0.05 & 0.05 \\ \mathbf{0.60} & 0.10 & 0.30 \\ 0.15 & 0.05 & \mathbf{0.80} \\ \dots & \dots & \dots \end{pmatrix} \text{ vs } \begin{pmatrix} 0 & \mathbf{1} & 0 \\ \mathbf{1} & 0 & 0 \\ \mathbf{1} & 0 & 0 \\ 0 & 0 & \mathbf{1} \\ \dots & \dots & \dots \end{pmatrix}}_{\textbf{calculating gradient } d\mathcal{L}}$$



Outline

Softmax Layer & Classification

Backpropagation Again

Fully Functional ANN

Fully Functional ANN

Fully Functional ANN

see MyANN.py

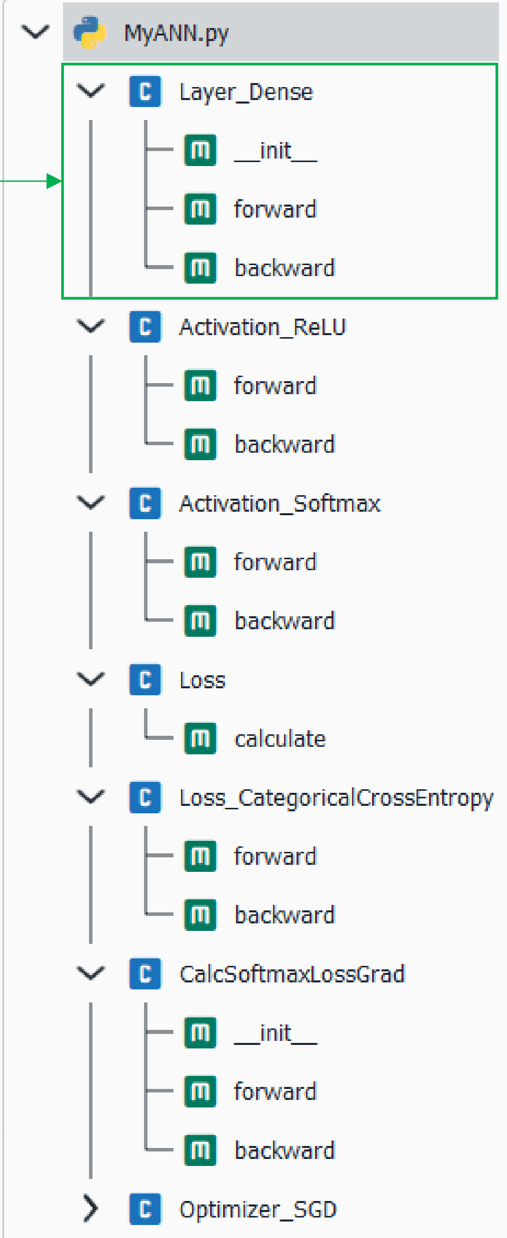
class for dense layer:
- contains **learnables** (weights, biases)

```python
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

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



Fully Functional ANN

Fully Functional ANN

see MyANN.py

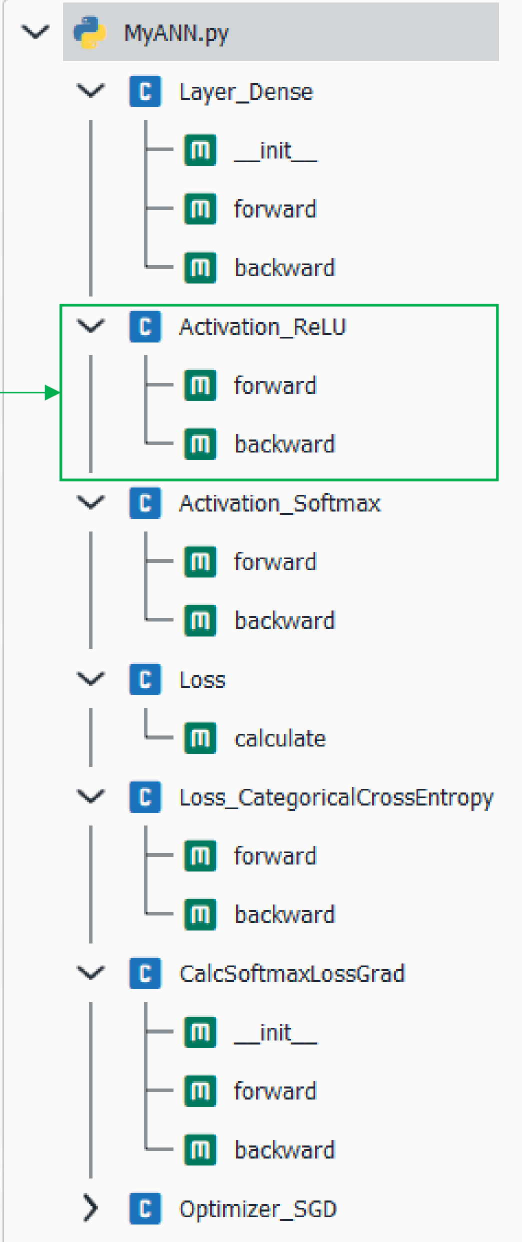
class for activation functions:
- introduces **nonlinearity**
- scaling/readjusting values

```python
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
```

similar for any other activation function



Fully Functional ANN

see MyANN.py

class for softmax:
- for **classification**
- turns output of last layer into probabilities

```python
class Activation_Softmax:

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

- MyANN.py
  - Layer_Dense
    - __init__
    - forward
    - backward
  - Activation_ReLU
    - forward
    - backward
  - Activation_Softmax
    - forward
    - backward
  - Loss
    - calculate
  - Loss_CategoricalCrossEntropy
  - CalcSoftmaxLossGrad
    - __init__
    - forward
    - backward
  - Optimizer_SGD

Fully Functional ANN

see MyANN.py

class for loss function:
- cross entropy for **classification**
- MSE for **regression**

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

- MyANN.py
  - Layer_Dense
    - __init__
    - forward
    - backward
  - Activation_ReLU
    - forward
    - backward
  - Activation_Softmax
    - forward
    - backward
  - Loss
    - calculate
  - Loss_CategoricalCrossEntropy
    - forward
    - backward
  - CalcSoftmaxLossGrad
    - __init__
    - forward
    - backward
  - Optimizer_SGD

Fully Functional ANN

see MyANN.py

class for optimizer:
- **updates learnables**

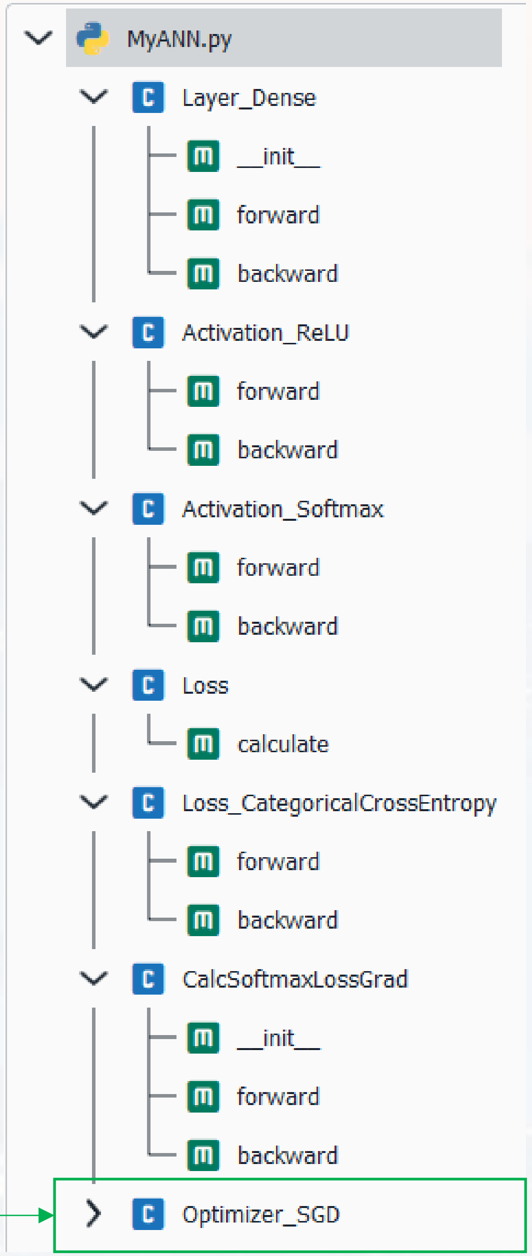
```python
class Optimizer_SGD:

    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

    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):

        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)
            ...
```



Fully Functional ANN

basic structure:

see `ANNIII.ipynb`

initialization

```
dense1        = Layer_Dense(X.shape[1], n_neuron1)
dense2        = Layer_Dense(n_neuron1, n_neuron2)
Activation    = Activation_ReLU()
loss_function = CalcSoftmaxLossGrad()
optimizer     = Optimizer_SGD(learning_rate, decay, momentum)
```

forward

```
for epoch in range(N):

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

evaluation

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

Fully Functional ANN

Fully Functional ANN

basic structure:

see `ANNIII.ipynb`

training

forward

```
for epoch in range(N):

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

evaluation

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

backpropagation

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

optimization

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

Fully Functional ANN

once the ANN has been fully trained: store all learnables (= memory of the network)

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

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

Fully Functional ANN

once the ANN has been fully trained: store all learnables (= memory of the network)

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

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

notes: - it usually takes **several training sessions** for different hyper parameter (momentum, regularization etc)

- training is the actual time/energy/computational resources consuming part

- application of the once trained ANN is fast (= one epoch)

Fully Functional ANN

Fully Functional ANN

workflow:

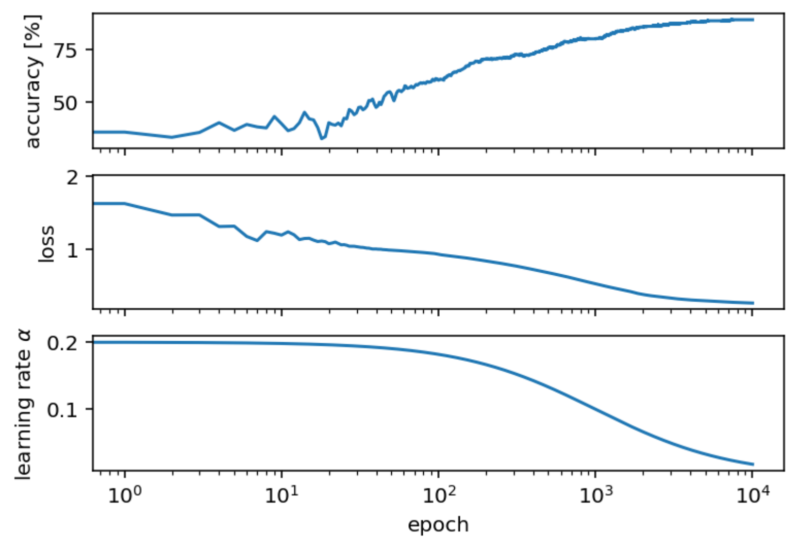
```
[x, y] = spiral_data(samples = 200, classes = 5)
```

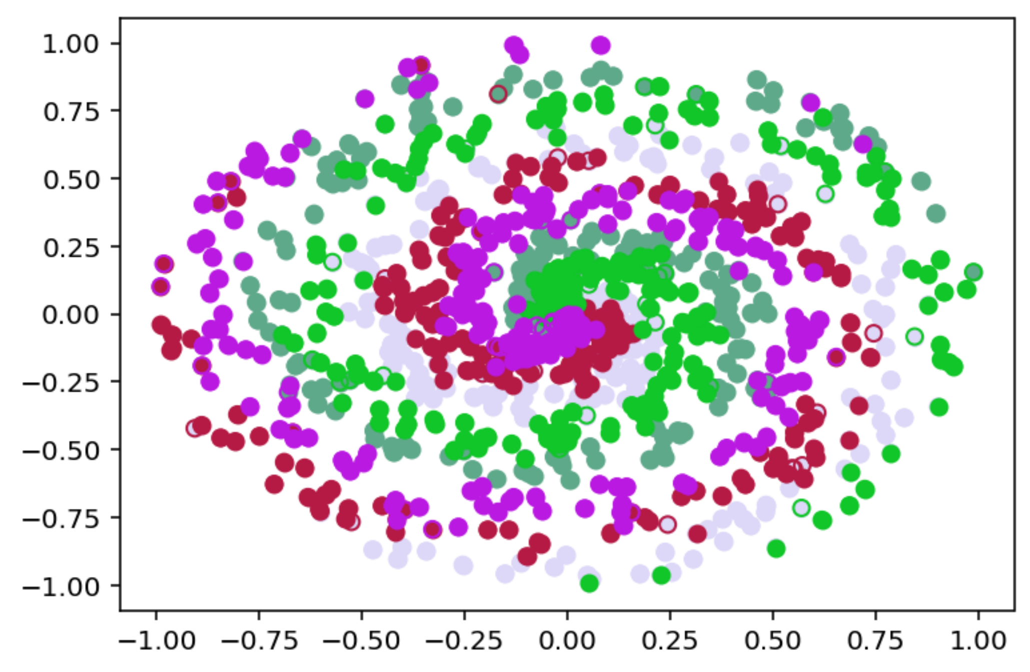
1. training `see MyANN.py and RunMyANN.py`

```
RunMyANN(x,y)
```

weights1.npy

weights2.npy





face color: true class
edge color: predicted class

Fully Functional ANN

Fully Functional ANN

workflow:

```
[x, y] = spiral_data(samples = 200, classes = 5)
```

2. application

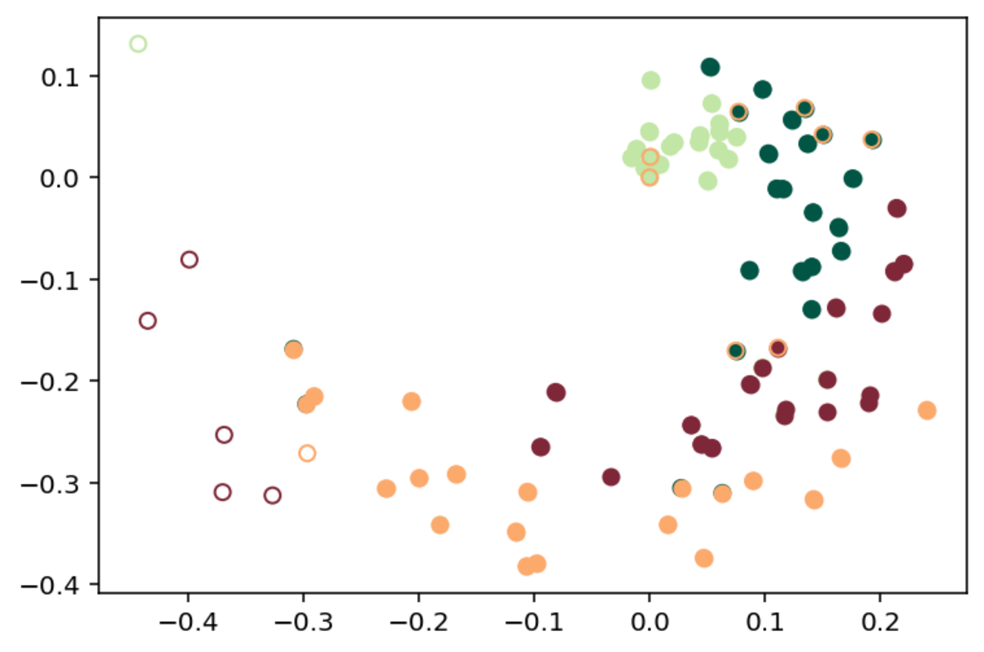
see ApplyMyANN.py

```
[x_new, y_new] = spiral_data(samples = 20, classes =
5)
[predictions, probabilities] = ApplyMyANN(x_new)
```

```
w1 = np.load('weights1.npy')
w2 = np.load('weights2.npy')

S = w1.shape

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



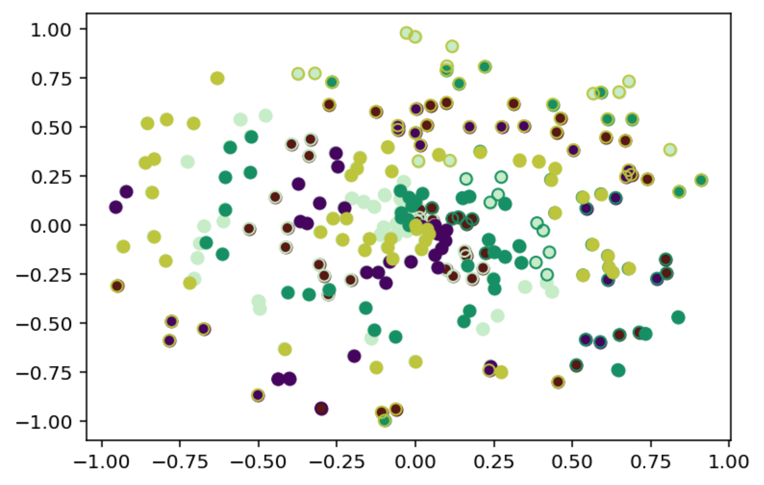
face color: true class
edge color: predicted class

Fully Functional ANN

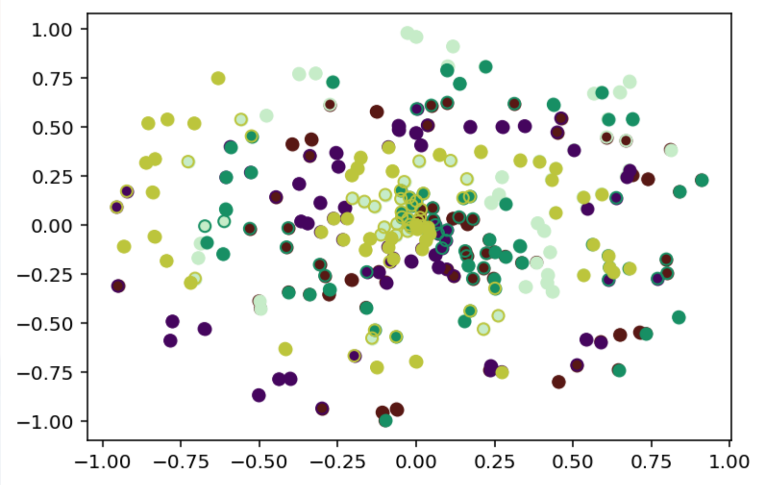
Fully Functional ANN

see `RunAll.py`

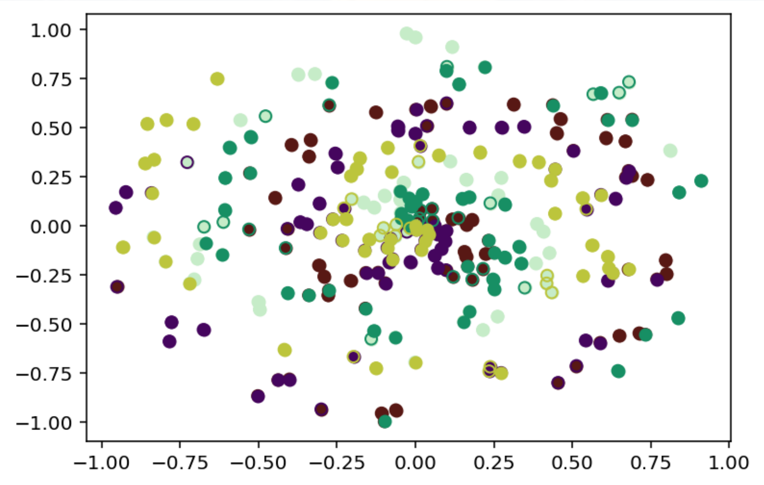
accuracy depending on epochs



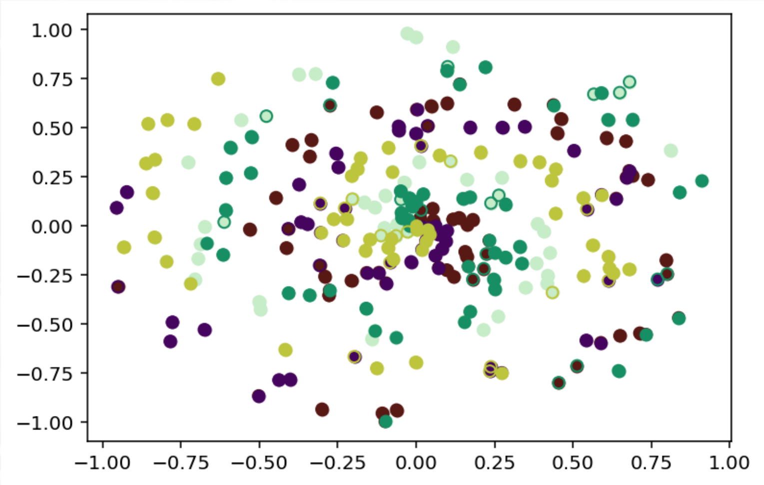
10 epochs



100 epochs



1,000 epochs



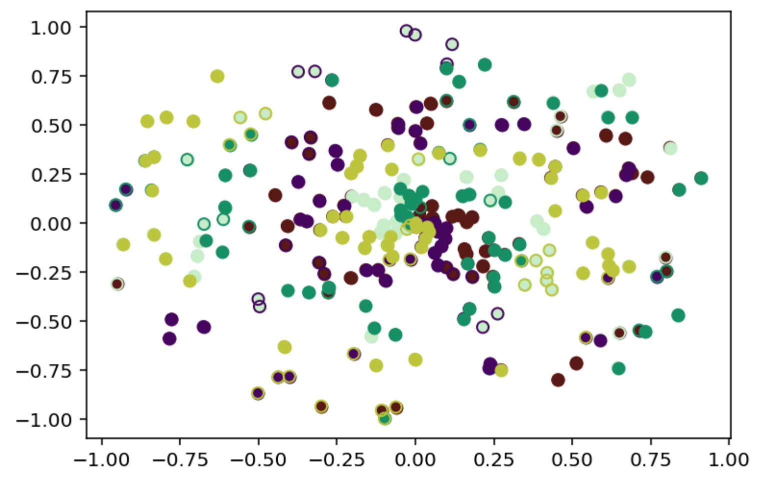
10,000 epochs

face color: true class
edge color: predicted class

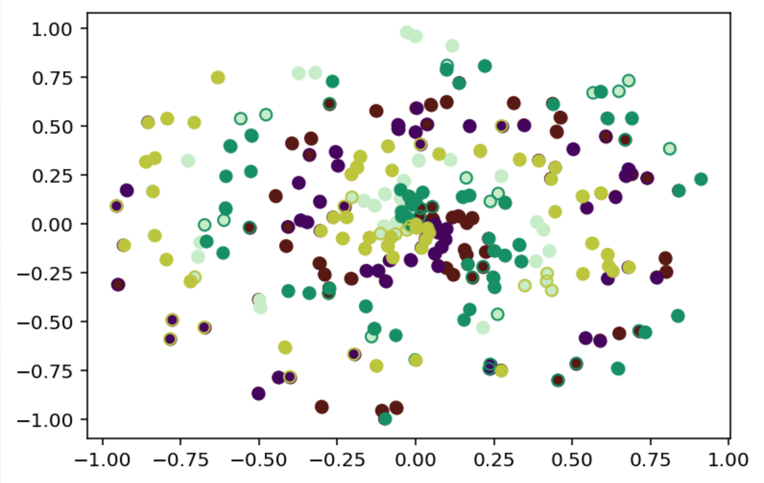
Fully Functional ANN

see `RunAll.py`

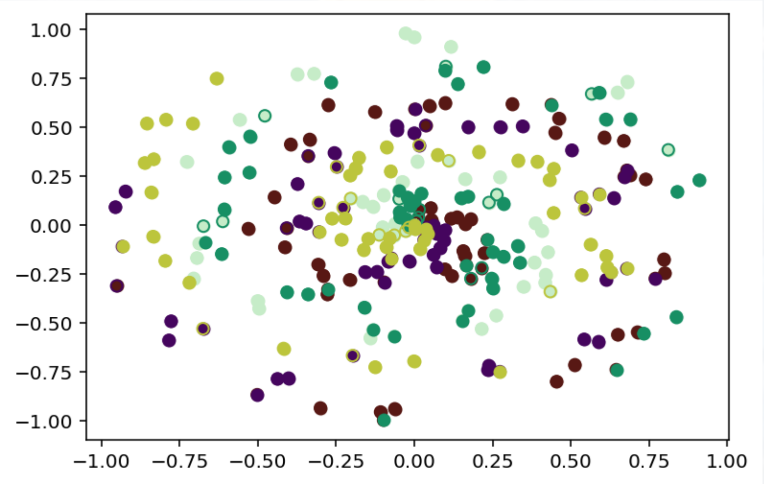
accuracy depending on size of training data



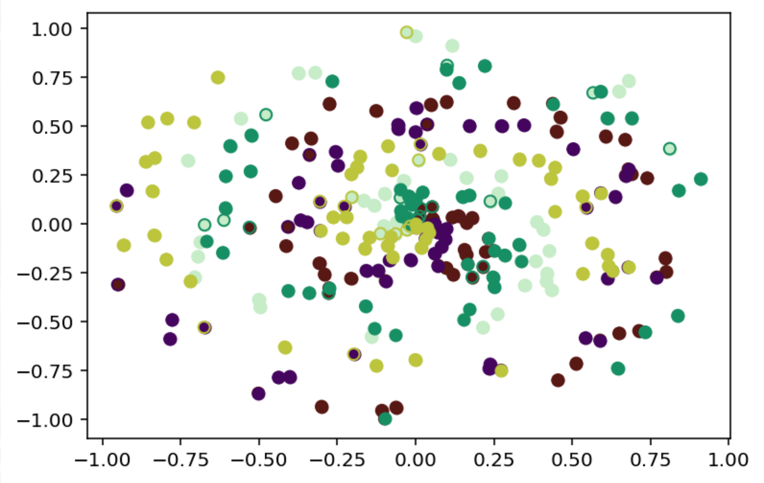
10 datapoints



100 datapoints



1,000 datapoints

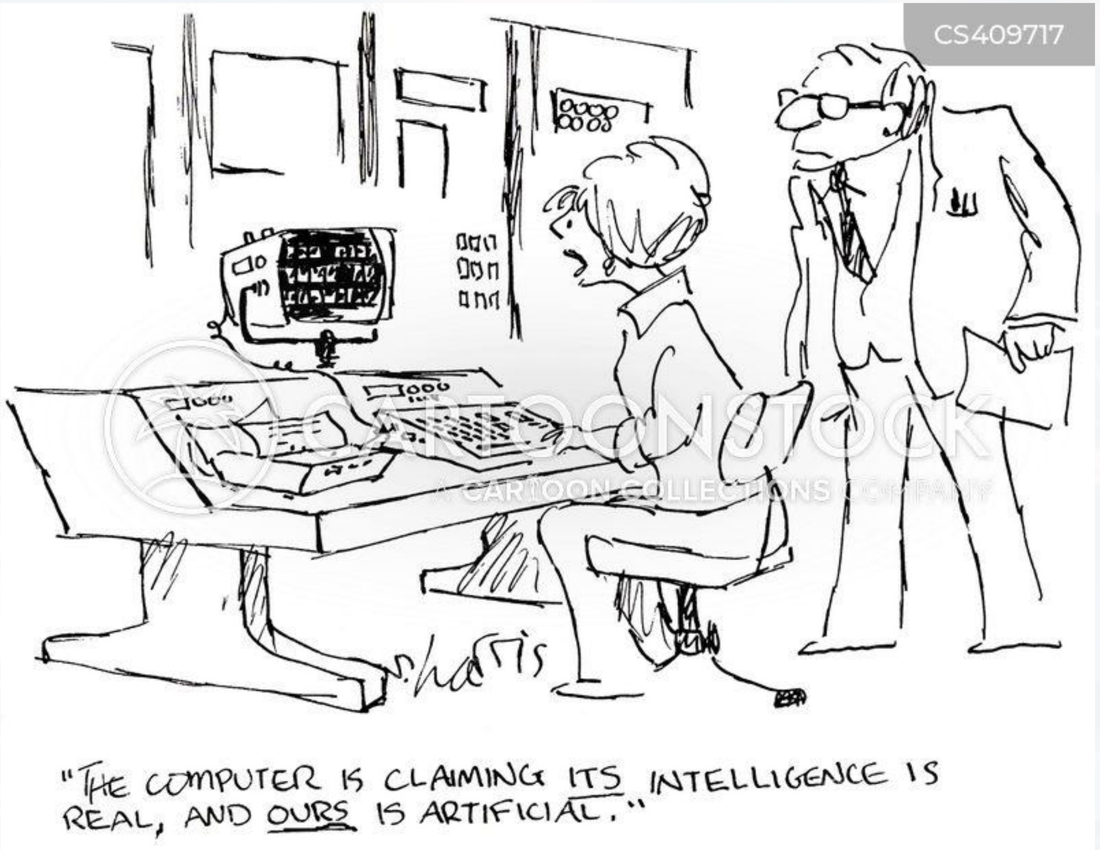


10,000 datapoints

face color: true class
edge color: predicted class

Fully Functional ANN

Thank you very much for your attention!



"THE COMPUTER IS CLAIMING ITS INTELLIGENCE IS REAL, AND OURS IS ARTIFICIAL."