INTRODUCTION TO COMPUTATIONAL PHYSICS

Kai-Feng Chen
National Taiwan University

NN TRAINING: THE METHOD

- The neural network training is performed by minimizing the loss function with variations on the **weights** and **bias**.
- One of the classical algorithms is the **gradient descent** method. To find a local minimum of the loss function, we take the steps **proportional to the negative of the gradient** for the function at the current point, e.g.

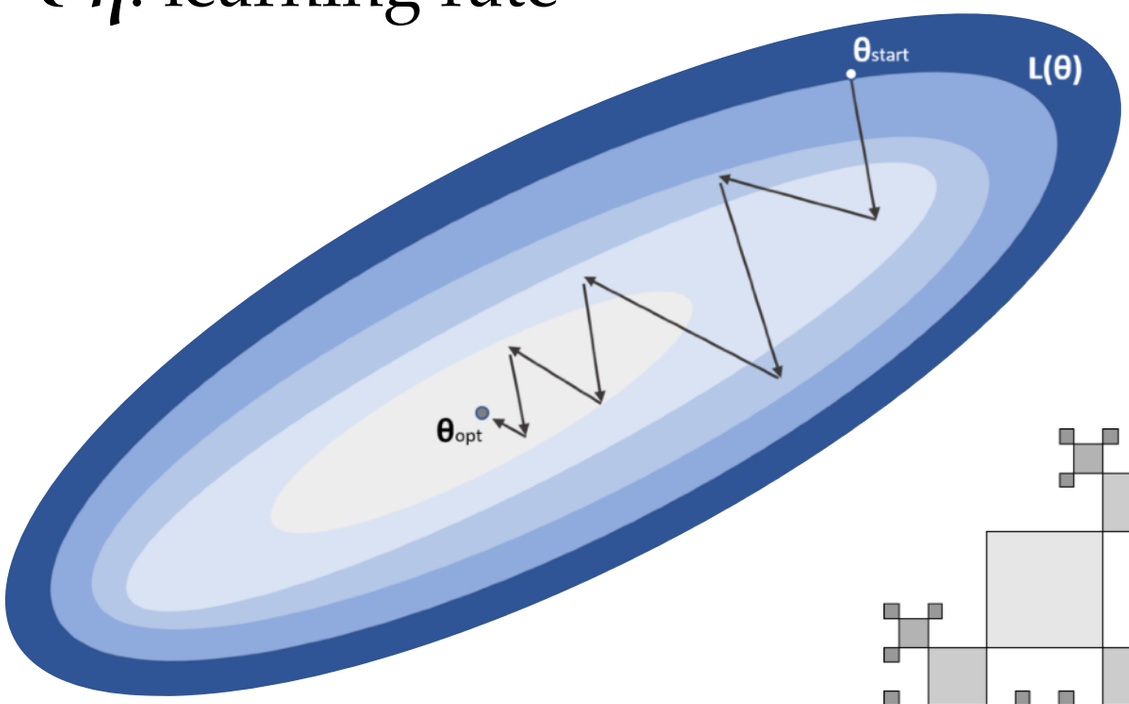
$$\theta \to \theta' = \theta - \eta \nabla L$$

$\left\{ \begin{array}{l} \theta\text{: any of the weights or bias} \\ \eta\text{: learning rate} \end{array} \right.$

More explicitly:

$$w_i \to w_i' = w_i - \eta \frac{\partial L}{\partial w_i}$$

$$b_j \to b_j' = b_j - \eta \frac{\partial L}{\partial b_j}$$



NN TRAINING: THE METHOD (II)

- Note the loss function has to be calculated over all of the input data x. So the gradient of the loss function has to be averaged over the input samples:

$$\nabla L = \frac{1}{n}\sum_{x}^{n} \nabla L_x$$

- However in practice this calculation is very slow if the size of input data is large. Hence the network also learns slowly. A method named **stochastic gradient descent (SGD)** can be help to speed up this process by limiting the calculation to a small *randomly chosen subset of the input data*, and the gradient can be calculated approximately:

$$\nabla L \approx \frac{1}{m}\sum_{x_k}^{m} \nabla L_{x_k} \quad \Rightarrow \quad w_i \to w_i' = w_i - \frac{\eta}{m}\sum_{k} \frac{\partial L_{x_k}}{\partial w_i}$$

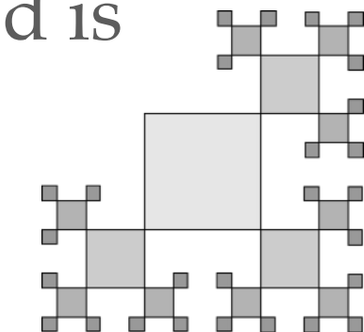
Such a small subset is usually called the **mini-batch**.

GRADIENT EVALUATION

- Obviously the calculation of gradient is essential in the training process based on the **SGD** algorithm. The question is how to do it in an efficient way. You may already think of some numerical differential as we introduced many weeks ago, but this is not good enough to be used here.

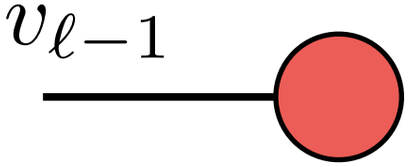
- In particular we have hundred thousands of parameters to be tuned, and every numerical differential requires a couple of full feed-forward calculations, and have to be carried out for many input data sets — this will simply result a rather slow calculation and again, a slow learning.

- But due to the special structure of neural network, the gradient can be in fact, calculated in a very efficient way. This method is called the **back propagation**.



BACK PROPAGATION

- Let's explain how it works by starting from the ending (output) layer of the network with only one neuron. For a given data point $\boldsymbol{x}$ and target $\boldsymbol{t}$, consider the following small variation:



$$v_\ell = \sigma(z_\ell) = \sigma\left(w_\ell v_{\ell-1} + b_\ell\right)$$

$$\delta_\ell = \frac{\partial L}{\partial z_\ell} \quad \text{where } L = \frac{1}{2}(v_\ell - t)^2$$

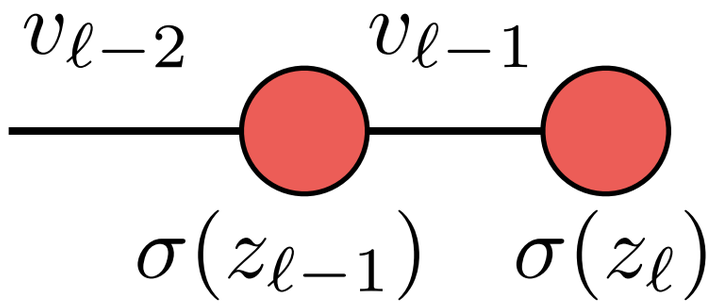
$$\Rightarrow \delta_\ell = \frac{\partial L}{\partial v_\ell}\frac{\partial v_\ell}{\partial z_\ell} = [\sigma(z_\ell) - t] \cdot \sigma'(z_\ell)$$

$$\Rightarrow \begin{cases} \dfrac{\partial L}{\partial b_\ell} = \delta_\ell \\ \dfrac{\partial L}{\partial w_\ell} = \delta_\ell v_{\ell-1} \end{cases}$$

The differentials at the ending layer can be calculated easily!

BACK PROPAGATION (II)

- Then propagate back by another layer of single neuron:



$$v_\ell = \sigma\left(w_\ell v_{\ell-1} + b_\ell\right) = \sigma\left(w_\ell \sigma(z_{\ell-1}) + b_\ell\right)$$

$$\delta_{\ell-1} = \frac{\partial L}{\partial z_{\ell-1}} = \frac{\partial L}{\partial z_\ell}\frac{\partial z_\ell}{\partial z_{\ell-1}} = \delta_\ell w_\ell \sigma'(z_{\ell-1})$$

$$\text{since} \quad z_\ell = w_\ell \sigma(z_{\ell-1}) + b_\ell$$

$$\Rightarrow \begin{cases} \dfrac{\partial L}{\partial b_{\ell-1}} = \delta_{\ell-1} \\ \dfrac{\partial L}{\partial w_{\ell-1}} = \delta_{\ell-1} v_{\ell-2} \end{cases}$$

Basically the calculations of the differentials are the same as the ending layer!

BACK PROPAGATION (III)

- Summarize all of the formulae together, assuming a feedforward calculation has been performed, hence the values at each layer (i.e. z_ℓ are already known!)

For ending layer:
$$\delta_\ell = \frac{\partial L}{\partial v_\ell}\sigma'(z_\ell) = [\sigma(z_\ell) - t]\sigma'(z_\ell)$$

For other layers:
$$\delta_{\ell-1} = [w_\ell \cdot \delta_\ell]\sigma'(z_{\ell-1})$$

For the gradient:
$$\frac{\partial L}{\partial b_\ell} = \delta_\ell$$
$$\frac{\partial L}{\partial w_\ell} = \delta_\ell v_{\ell-1} = \delta_\ell \sigma(z_{\ell-1})$$

Based on this the gradient can be calculated, **back propagated** from the ending layer. And the feedforward calculation is performed only once!

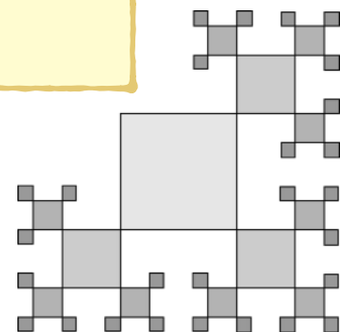
IMPLEMENTATION: BACK PROPAGATION

- We shall add corresponding stuff in the code:
  - Add the first derivative of the activation function: $\sigma'(z)$
  - Add the arrays to store the gradient along weights (`delw`) and biases (`delb`).

partial **neurons.py**

```
def sigma(z):
    return 0.5*(np.tanh(0.5*z)+1.)
def sigma_p(z):
    return sigma(z)*(1.-sigma(z))

class neurons(object):
    def __init__(self, shape):
        self.shape = shape
. . . . . .
        self.delw = [np.zeros(w.shape) for w in self.w]
        self.delb = [np.zeros(b.shape) for b in self.b]
```

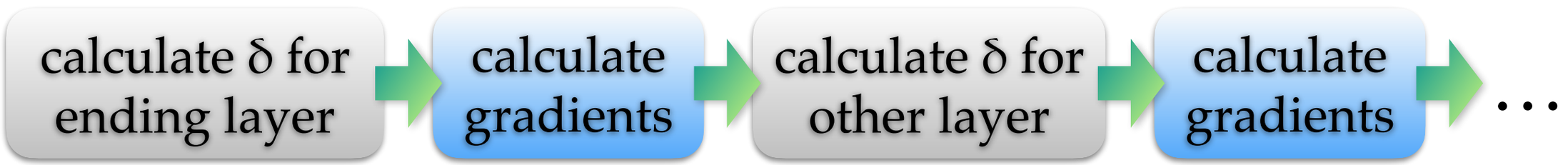


IMPLEMENTATION: BACK PROPAGATION (II)

- The main gradient estimate, to be carried out *after feed-forward network calculation*:

partial **neurons.py**

```
def gradient(self, y):
    for l in range(len(self.shape)-2,-1,-1):
        if l==len(self.shape)-2:
            delta = (self.v[-1]-y.reshape(self.v[-1].shape)) * \
                    sigma_p(self.z[l])
        else: delta = np.dot(self.w[l+1].T,self.delb[l+1]) * \
                    sigma_p(self.z[l])
        self.delb[l] = delta
        self.delw[l] = np.dot(delta,self.v[l].T)
```



IMPLEMENTATION: TRAINING WITH SGD

partial **neurons.py**

```
def fit(self, x_data, y_data, epochs, batch_size, eta):
    samples = list(zip(x_data, y_data))
    for ep in range(epochs):
        print('Epoch: %d/%d' % (ep+1,epochs))
        random.shuffle(samples)
        sum_delw = [np.zeros(w.shape) for w in self.w]
        sum_delb = [np.zeros(b.shape) for b in self.b]
        batch_count = 0
        for x,y in samples:
            self.predict(x)
            self.gradient(y)
            for l in range(len(self.shape)-1):
                sum_delw[l] += self.delw[l]
                sum_delb[l] += self.delb[l]
            batch_count += 1
            if batch_count>=batch_size or (x is samples[-1][0]):
                for l in range(len(self.shape)-1):
                    self.w[l] -= eta/batch_count*sum_delw[l]
                    self.b[l] -= eta/batch_count*sum_delb[l]
                    sum_delw[l],sum_delb[l] = 0.,0.
                batch_count = 0
        ret = self.evaluate(x_data, y_data)
        print('Loss: %.4f, Acc: %.4f' % ret)
```

make the training sample paired in python list + randomize

↑ for average of weights/bias

← feedforward + back-propagation

Update weights/bias →

← measure the performance

IMPLEMENTATION: PERFORMANCE EVALUATION

- Surely it would be necessary to evaluation the performance of the network by comparing the output with the expected output.
- One typical way is to calculate the **loss function** of the given data.
- Another way is to calculate the **accuracy**. Since we have arranged an output neuron per target class, so we can just take the largest output neuron as the identified class.

partial **neurons.py**

```
def evaluate(self, x_data, y_data):
    loss, cnt = 0., 0.
    for x,y in zip(x_data, y_data):
        self.predict(x) ← feedforward calculations
        loss += ((self.v[-1]-y.reshape(self.v[-1].shape))**2).sum()
        if np.argmax(self.v[-1])==np.argmax(y): cnt += 1.
    loss /= 2.*len(x_data)
    return loss, cnt/len(x_data)
```

Now we are ready to train…your network!

LET'S TRAIN OUR NETWORK

- The real work is done in **one line of "fit"**:

partial ex_ml5_1.py

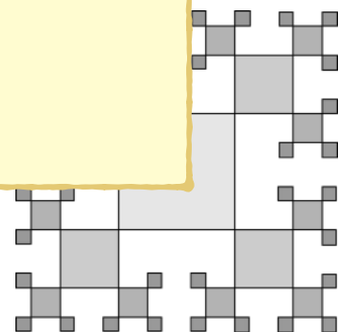
```
. . . . . .
x_train = np.array([[img.mean(),img[10:18,11:17].mean()] \
                    for img in x_train])
y_train = np.array([[1,0],[0,1]][n] for n in y_train])
x_test = np.array([[img.mean(),img[10:18,11:17].mean()] \
                    for img in x_test])
y_test = np.array([[1,0],[0,1]][n] for n in y_test])

from neurons import neurons
model = neurons([2,5,2])
model.fit(x_train, y_train, 20, 30, 1.0)
print('Performance (training)')
print('Loss: %.5f, Acc: %.5f' % model.evaluate(x_train, y_train))
print('Performance (testing)')
print('Loss: %.5f, Acc: %.5f' % model.evaluate(x_test, y_test))

out = np.array([model.predict(x) for x in x_train])

fig = plt.figure(figsize=(6,6), dpi=80)
. . . . . .
```

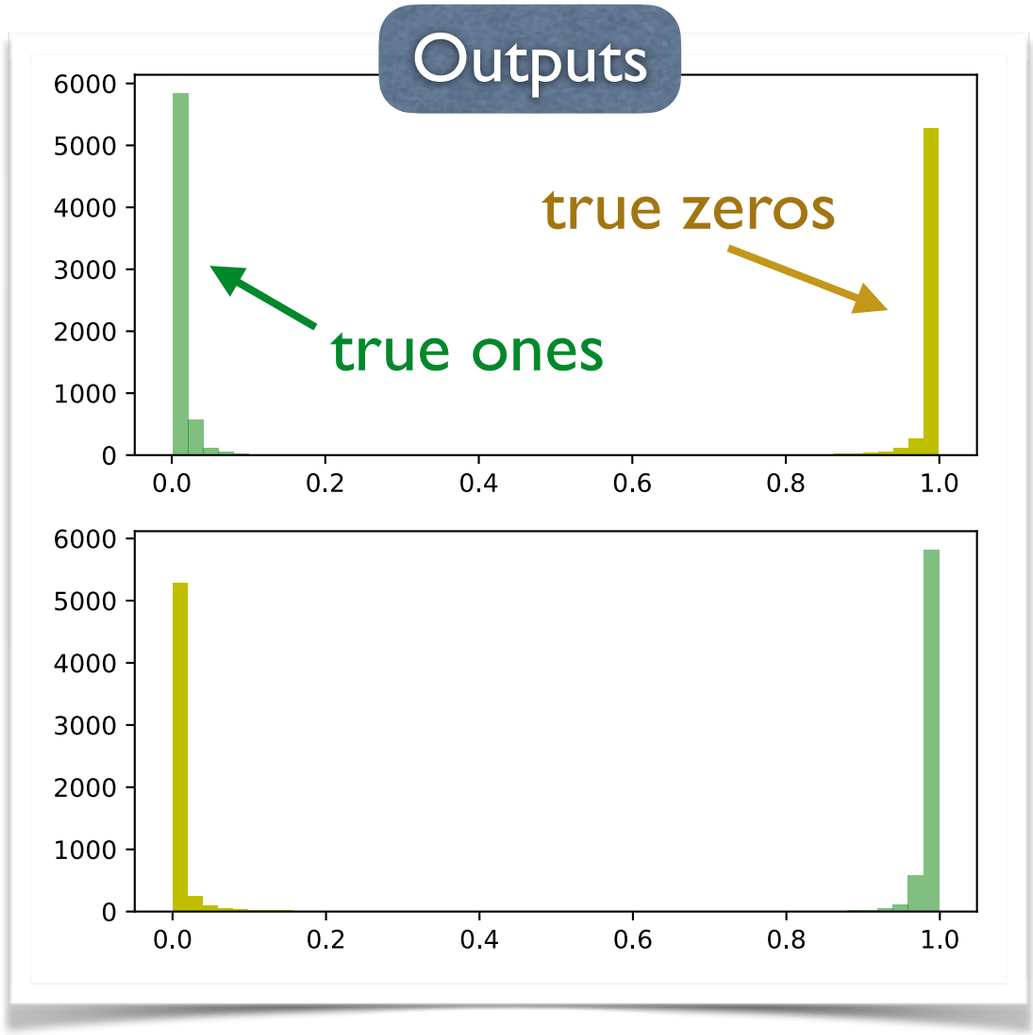
← with 20 epochs, mini-batch of 30
learning rate = 1.0



LET'S TRAIN OUR NETWORK (II)

- You can see the performance does increase as epochs:

```
Epoch: 1/20
Loss: 0.0837, Acc: 0.9212
Epoch: 2/20
Loss: 0.0470, Acc: 0.9470
Epoch: 3/20
Loss: 0.0282, Acc: 0.9765
Epoch: 4/20
Loss: 0.0153, Acc: 0.9898
Epoch: 5/20
Loss: 0.0109, Acc: 0.9889
. . . . . .
Epoch: 20/20
Loss: 0.0055, Acc: 0.9935
Performance (training)
Loss: 0.00552, Acc: 0.99353
Performance (testing)
Loss: 0.00395, Acc: 0.99527
```



And very good separation at the two output distributions!

NOW GO FOR FULL DIGITS SEPARATION...

- With a smaller scale test (only two features) it works rather well. How about if we put in all of the pixels and all of the training images in a NN of 784-30-10 structure?

partial ex_ml5_2.py

```
mnist = np.load('mnist.npz')
x_train = mnist['x_train']/255.
y_train = np.array([np.eye(10)[n] for n in mnist['y_train']])
x_test = mnist['x_test']/255.
y_test = np.array([np.eye(10)[n] for n in mnist['y_test']])

from neurons import neurons
model = neurons([784,30,10])
model.fit(x_train, y_train, 20, 10, 3.0)

print('Performance (training)')
print('Loss: %.5f, Acc: %.5f' % model.evaluate(x_train, y_train))
print('Performance (testing)')
print('Loss: %.5f, Acc: %.5f' % model.evaluate(x_test, y_test))
. . . . . .
```

← data prepared

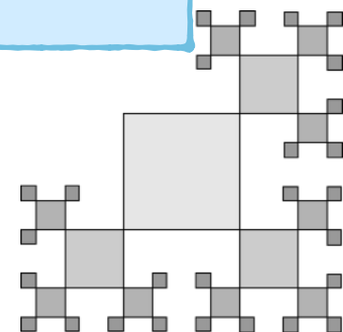
← with 20 epochs, mini-batch of 10
learning rate = 3.0

Training such a network definitely goes beyond the capability of standard minimizers!

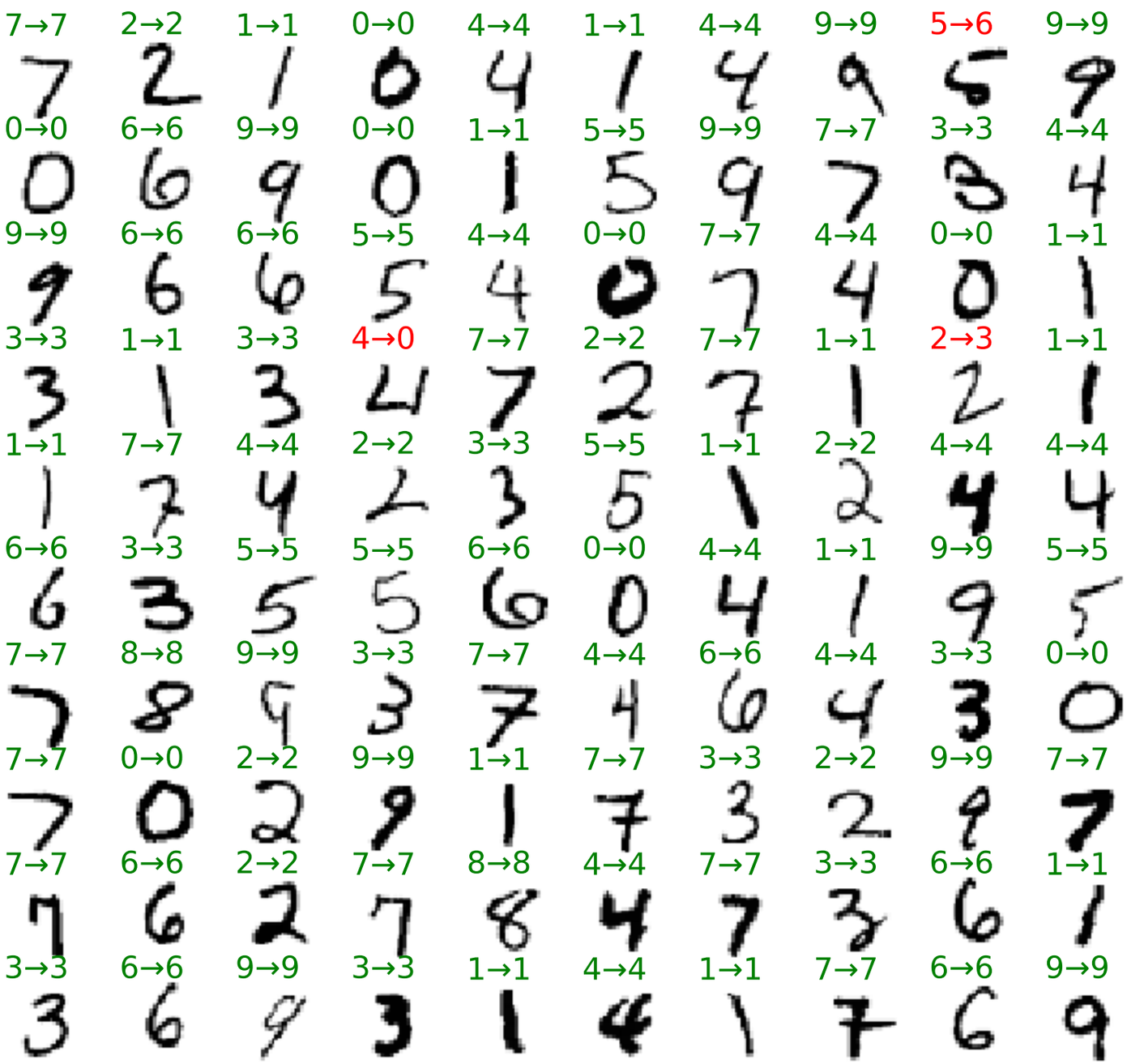
NOW GO FOR FULL DIGITS SEPARATION…(II)

- With only ~66 lines of code we can already build a simple feedforward network + SGD training for handwriting digits recognition!
- And you can see the performance is not bad either. With 20 epochs of training we obtained a test accuracy of ~**94.9%**.
- But it is still not yet as good as the best SVM of ~**98.4%**…!

```
Epoch: 1/20
Loss: 0.0764, Acc: 0.9076
Epoch: 2/20
Loss: 0.0587, Acc: 0.9292
Epoch: 3/20
Loss: 0.0538, Acc: 0.9351
Epoch: 4/20
Loss: 0.0480, Acc: 0.9433
Epoch: 5/20
Loss: 0.0439, Acc: 0.9478
. . . . . .
Epoch: 20/20
Loss: 0.0317, Acc: 0.9636
Performance (training)
Loss: 0.03172, Acc: 0.96358
Performance (testing)
Loss: 0.04477, Acc: 0.94860
```



NOW GO FOR FULL DIGITS SEPARATION…(III)



- Several misidentified digits found in the first 100 test samples!
- Obviously we have more options to play with — *how about a bigger network? how about a larger/smaller learning rate? how about different batch size?*
- Nevertheless let's study some of the properties first!

COMMENT: TRAINING VS TESTING

- As it has been already pointed earlier, the performance measured using the training sample and the testing sample can be rather different. This **overfitting (overtraining)** is a typical situation in the ML algorithms.
- One can image that this is due to the fact that the algorithm or the parameters are more adjusted with the training data. The following code demonstrates such a situation:

partial ex_ml5_2a.py

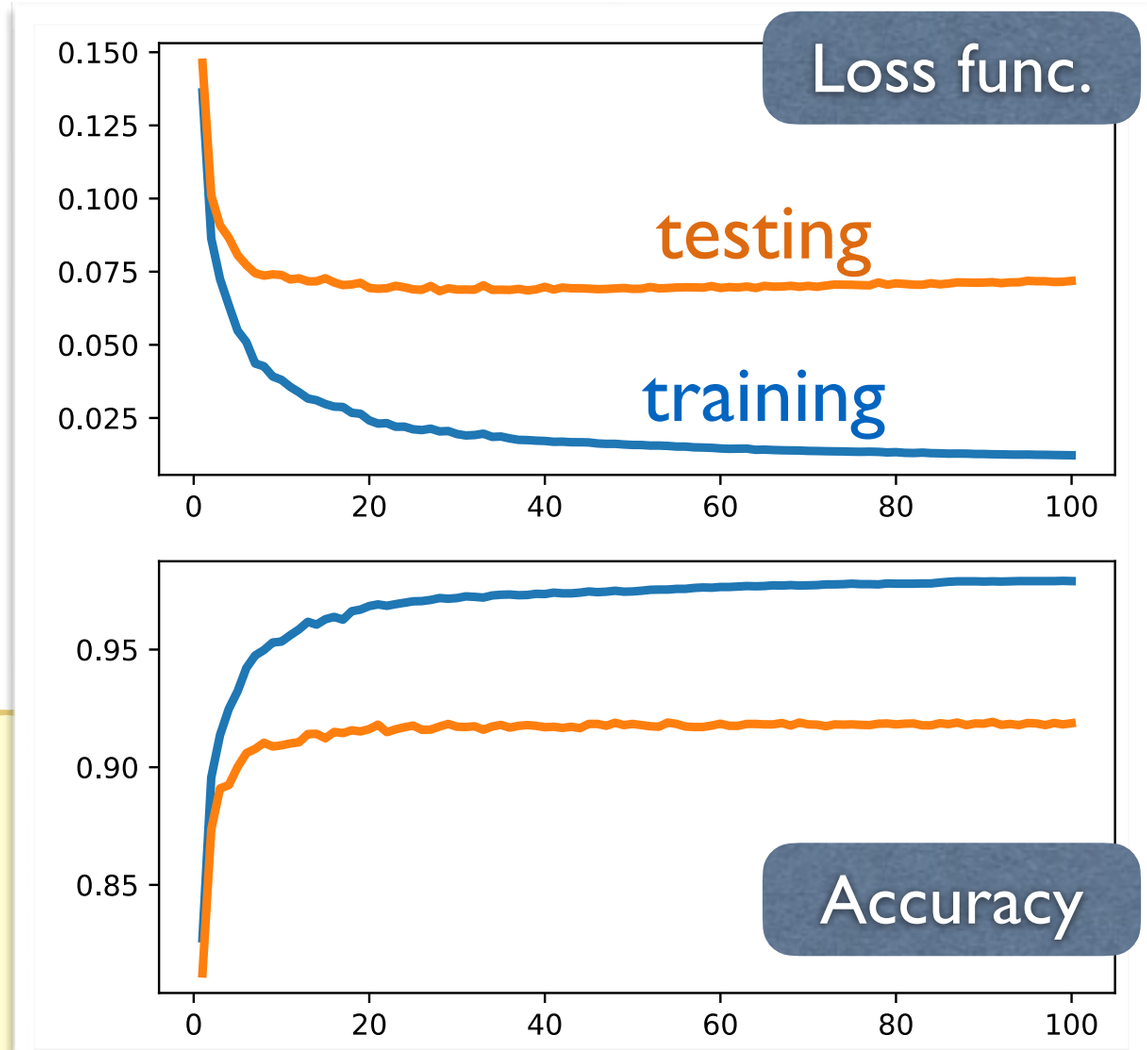
```
. . . . . .
scores = np.zeros((4,100)) ← stores the loss function & accuracy
                             from training/testing for 100 epochs
from neurons import neurons
model = neurons([784,30,10])
for ep in range(100):
    model.fit(x_train, y_train, 1, 10, 3.0)
    scores[0][ep],scores[1][ep] = model.evaluate(x_train, y_train)
    scores[2][ep],scores[3][ep] = model.evaluate(x_test, y_test)
. . . . . .
```

COMMENT: TRAINING VS TESTING (II)

- A comparison of training and testing performances.
- Strong overfitting happens after few epochs already, while training performance improves for long but not the testing performance.

partial ex_ml5_2a.py

```
. . . . . .
vep = np.linspace(1.,100.,100)
fig = plt.figure(figsize=(6,6), dpi=80)
plt.subplot(2,1,1)
plt.plot(vep,scores[0], lw=3)
plt.plot(vep,scores[2], lw=3)
plt.subplot(2,1,2)
plt.plot(vep,scores[1], lw=3)
plt.plot(vep,scores[3], lw=3)
plt.show()
```



One may try to include more training samples and such issue can be reduced!

COMMENT: NETWORK STRUCTURE

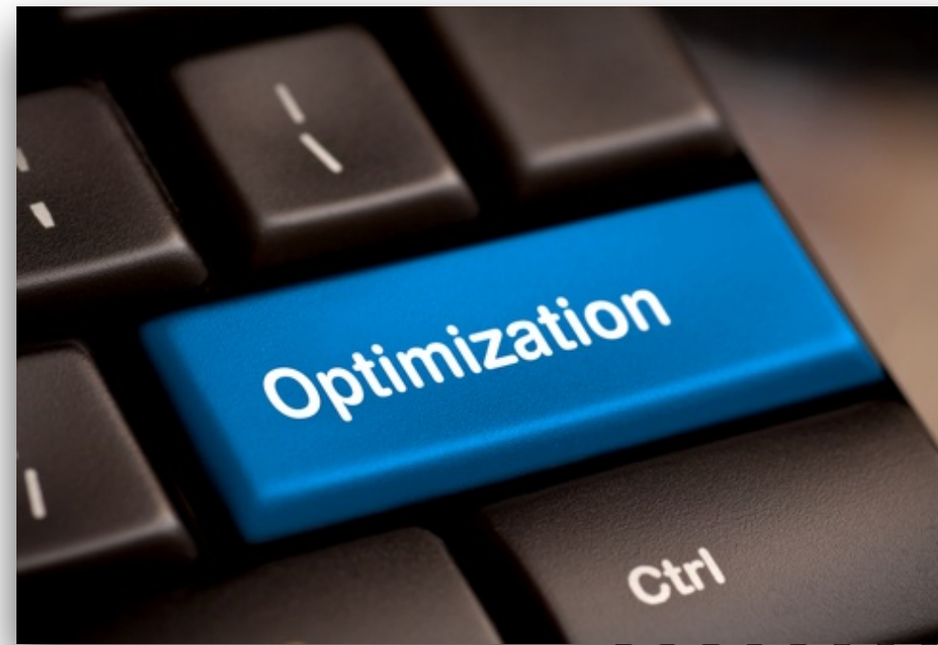
- In our current setup the network is configured as 784-30-10 neurons. More neurons can provide a more complicated model. So obviously adding more neurons (and layers) can, in principle, improve the network performance.
- However one can already expect some side effects:
  - **Network with more neurons usually takes more time and more difficult to train** (*with more weights and biases, and deeper!*).
  - **A more complex network may also have a stronger overfitting effect.**
- There are existing tricks to preserve the training efficiency, as well as mitigating the overfitting problem.

COMMENT: THINGS TO BE CONSIDERED

- There are many things can be consider to improve the network. Can be introduced either in the network structure, or in the training, targeting a fast learning and good performance in the testing sample:
  - *The choice of network structure*
  - *The choice of network initialization*
  - *The choice of training sample*
  - *The choice of input features*
  - *The choice of training algorithm*
  - *The choice of the loss function*
  - *The choice of the activation function*
  - *The choice of hyperparameters*
  - …

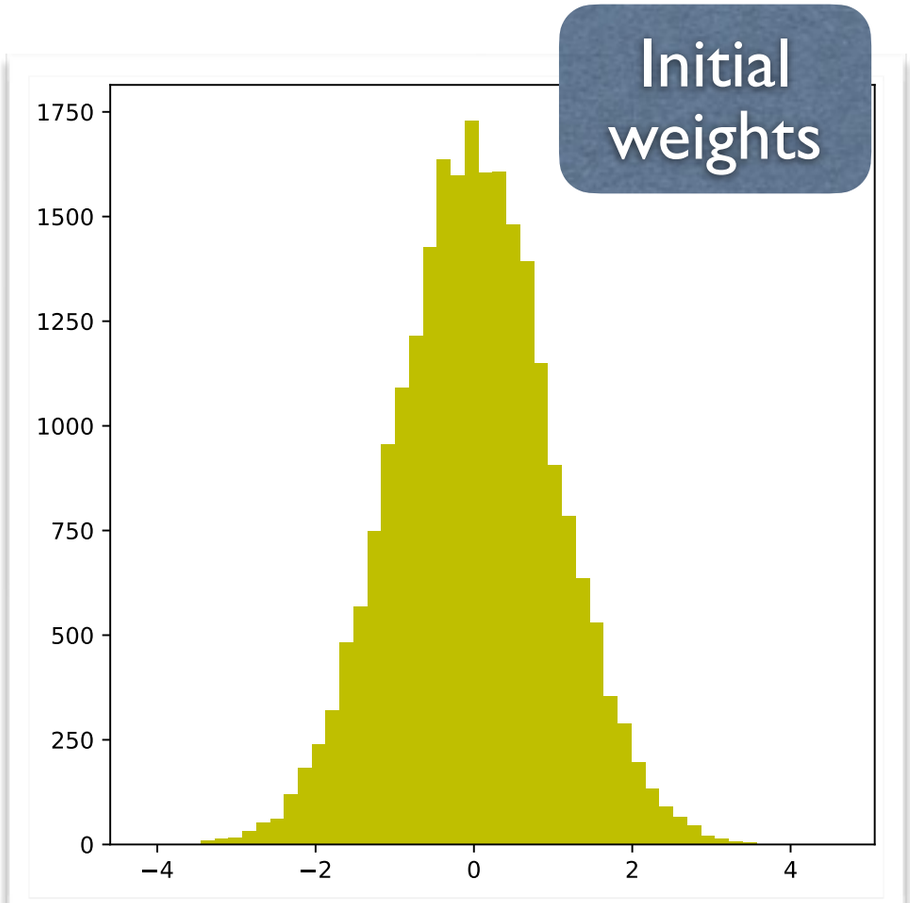
How to improve your network is totally nontrivial!

Let's try a quick & easy tweak first!



WEIGHTS INITIALIZATION

- Remember our weights (and biases) were all initialized with **standard Gaussian distributed random numbers**.
- It seems that it works quite well! But is there any *hidden issue*?

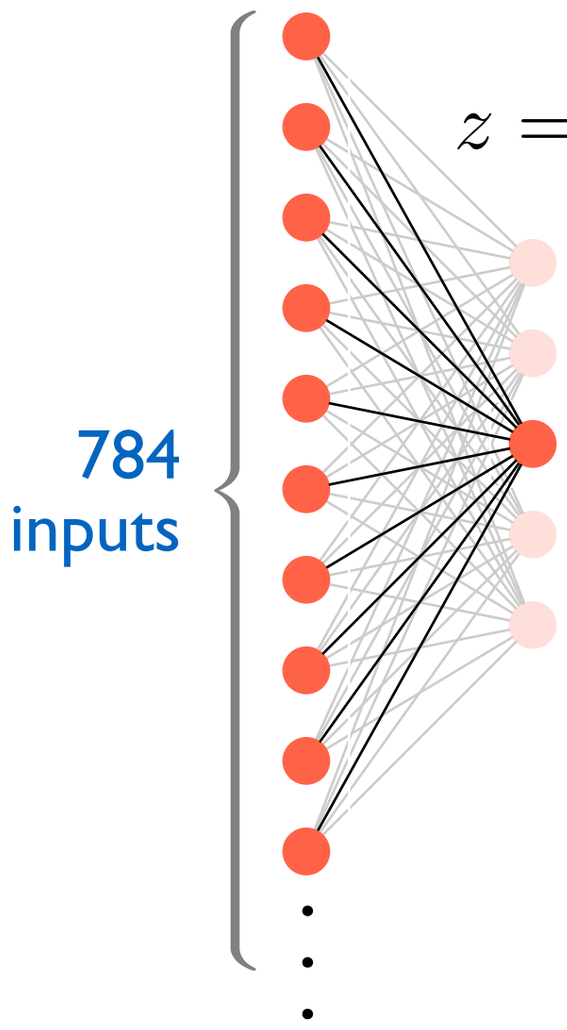


partial **neurons.py**

```
class neurons(object):
  def __init__(self, shape):
    self.shape = shape
. . . . . .
    self.w = [np.random.randn(n,m) for n,m in zip(shape[1:],shape[:-1])]
    self.b = [np.random.randn(n,1) for n in shape[1:]]
. . . . . .
```

WEIGHTS INITIALIZATION (II)

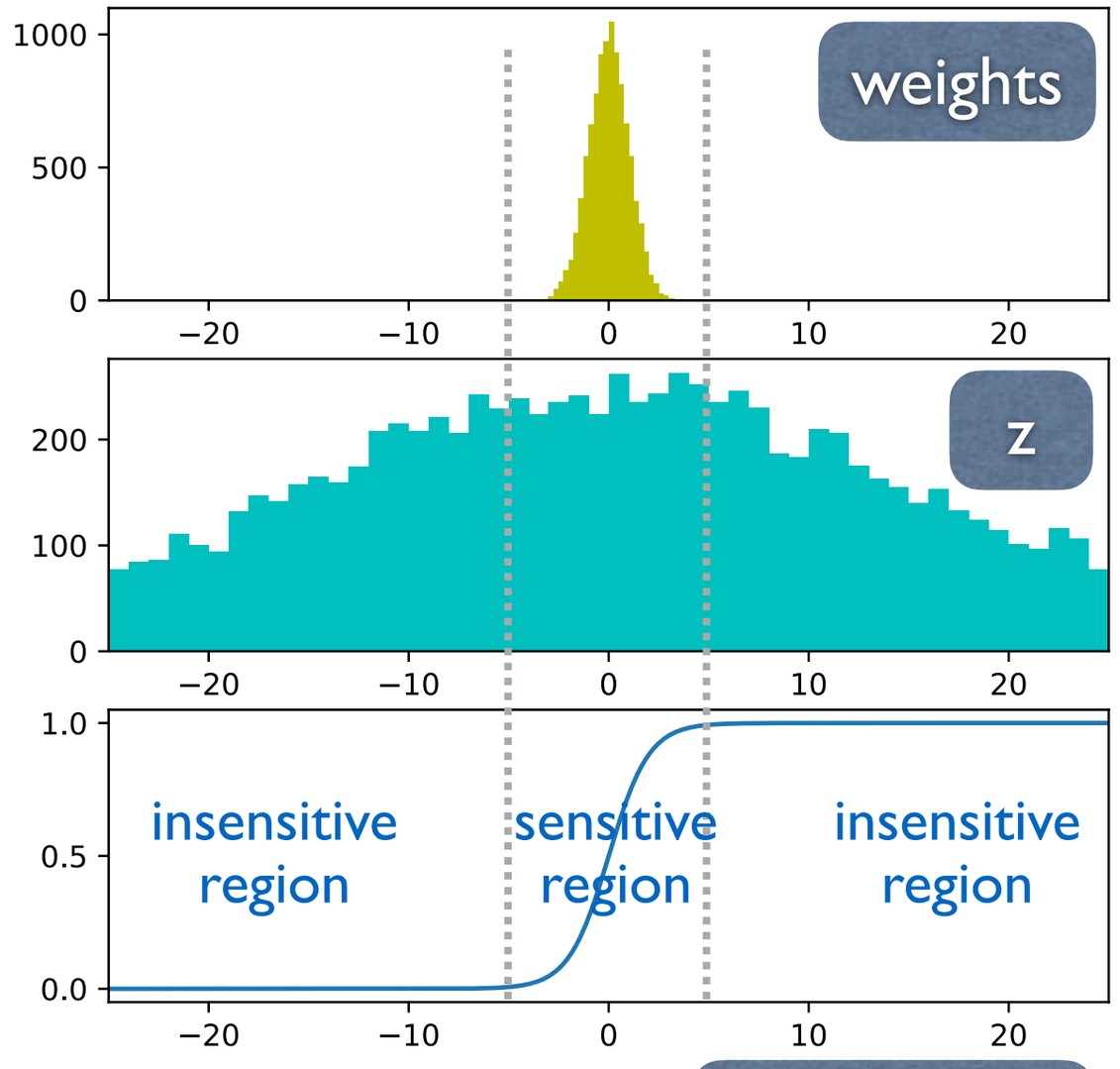
- In fact there is an issue indeed! Just consider the calculated value at one neuron at the first hidden layer:



$$z = \sum_i w_i x_i + b$$

z is the sum of many weights, and it is large!

when **z** is large, it is difficult to change after filling into the activation function!



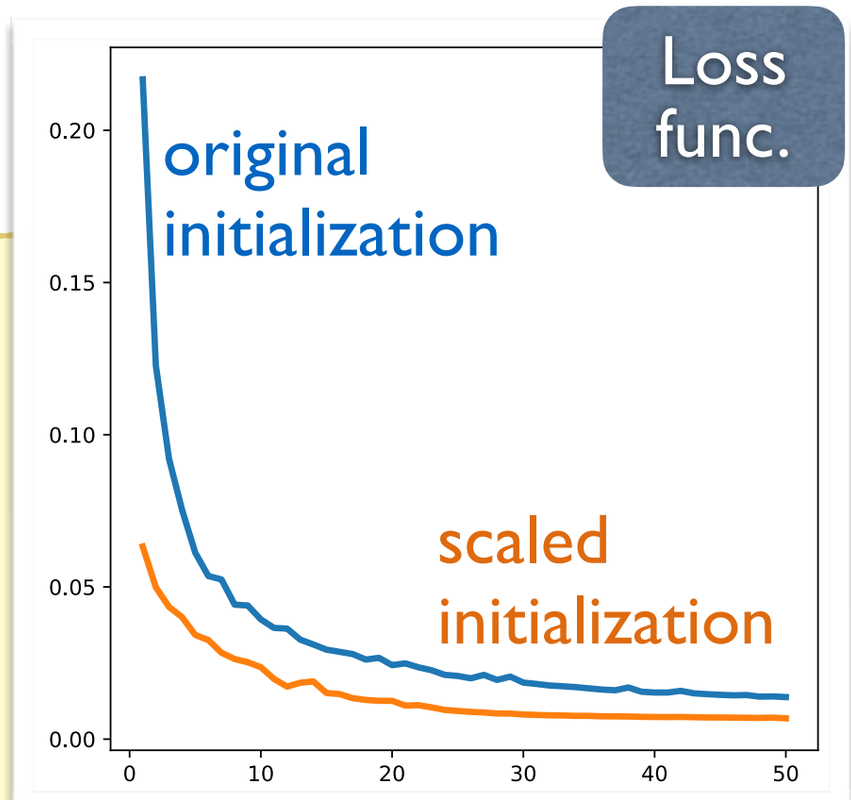
WEIGHTS INITIALIZATION (III)

- Sum of N input standard random Gaussian numbers will result a Gaussian with a width of $\sqrt{N}$.
- As an easy fix — let's just **scale the initial weights by this factor**!
- Even just with such as small fix, the learning speed is already much faster than the previous setup!

partial ex_ml5_2b.py

```
. . . . . .
from neurons import neurons
m1 = neurons([784,30,10])
m2 = copy.deepcopy(m1)
for w in m2.w:                    ↓ sqrt(# of inputs)
    w /= (w.shape[1])**0.5

for ep in range(50):
    m1.fit(x_train, y_train, 1, 10, 3.0)
    m2.fit(x_train, y_train, 1, 10, 3.0)
. . . . . .
```



BEFORE MOVING TO THE NEXT STEP . . .

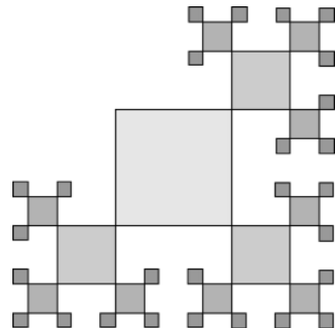
- With improved initial weights and the same 20 epochs of training we can already improve the test accuracy of from **94.9%** to **95.8%**!
- This is in fact very encouraging! A small tweak already gives us some visible performance boost. How about other possible tuning mentioned earlier?
- **Some of the "state-of-the-art" tricks will be discussed in the next module!**

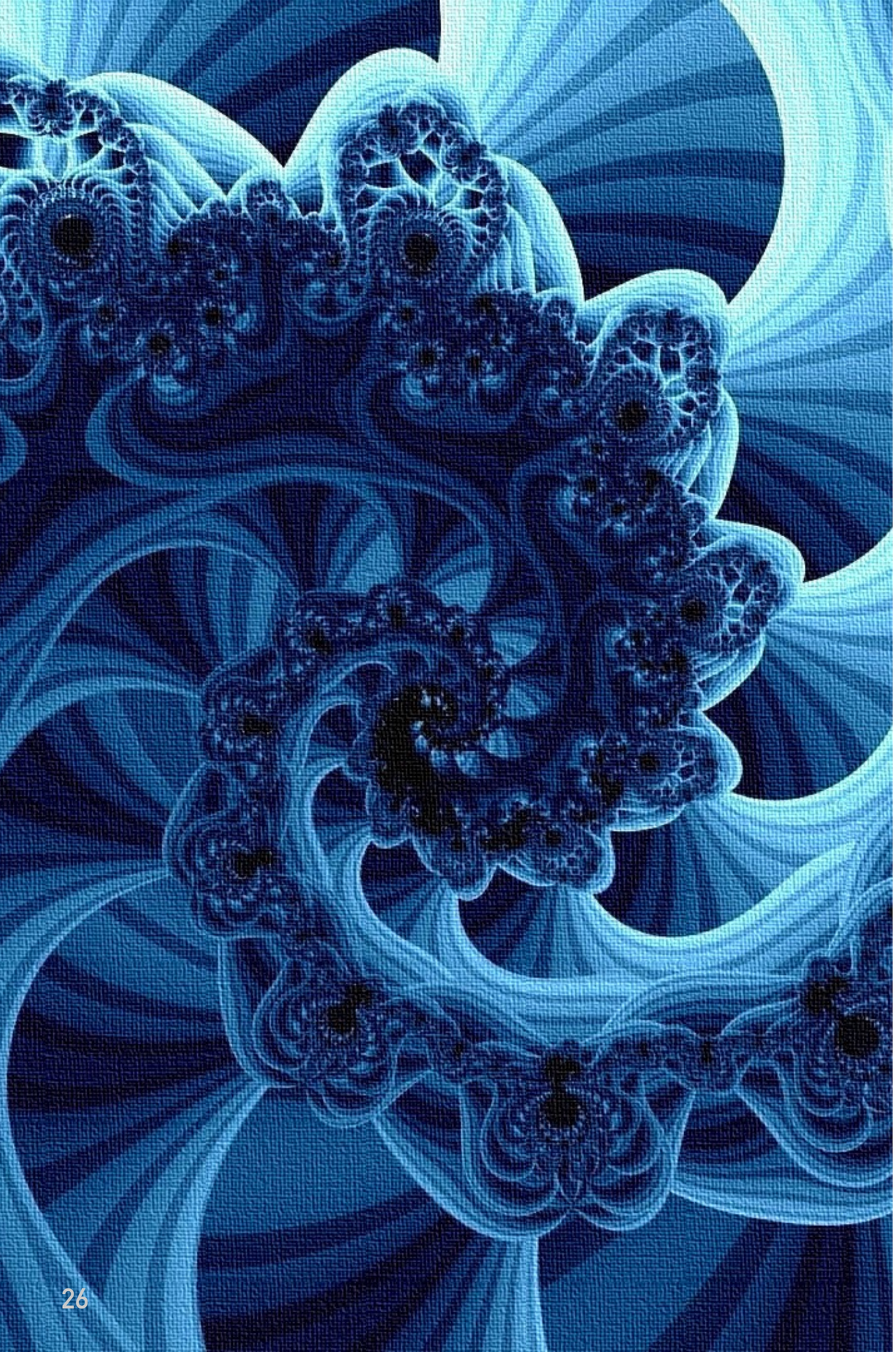
ex_ml5_2.py results

```
Epoch: 20/20
Loss: 0.0317, Acc: 0.9636
Performance (training)
Loss: 0.03172, Acc: 0.96358
Performance (testing)
Loss: 0.04477, Acc: 0.94860
```

ex_ml5_2c.py results

```
Epoch: 20/20
Loss: 0.0225, Acc: 0.9752
Performance (training)
Loss: 0.02249, Acc: 0.97518
Performance (testing)
Loss: 0.03646, Acc: 0.95840
```





MODULE SUMMARY

- In this module we have discussed how to train a neural network model with the simplest stochastic gradient descent method, and how to calculate the gradient with back propagation.
- Next module we will continue to discuss how to improve the performance of the neural network further with all of the known tricks.