2018

INTRODUCTION TO NUMERICAL ANALYSIS

Lecture 3-2: Incorporating Nonlinear Models

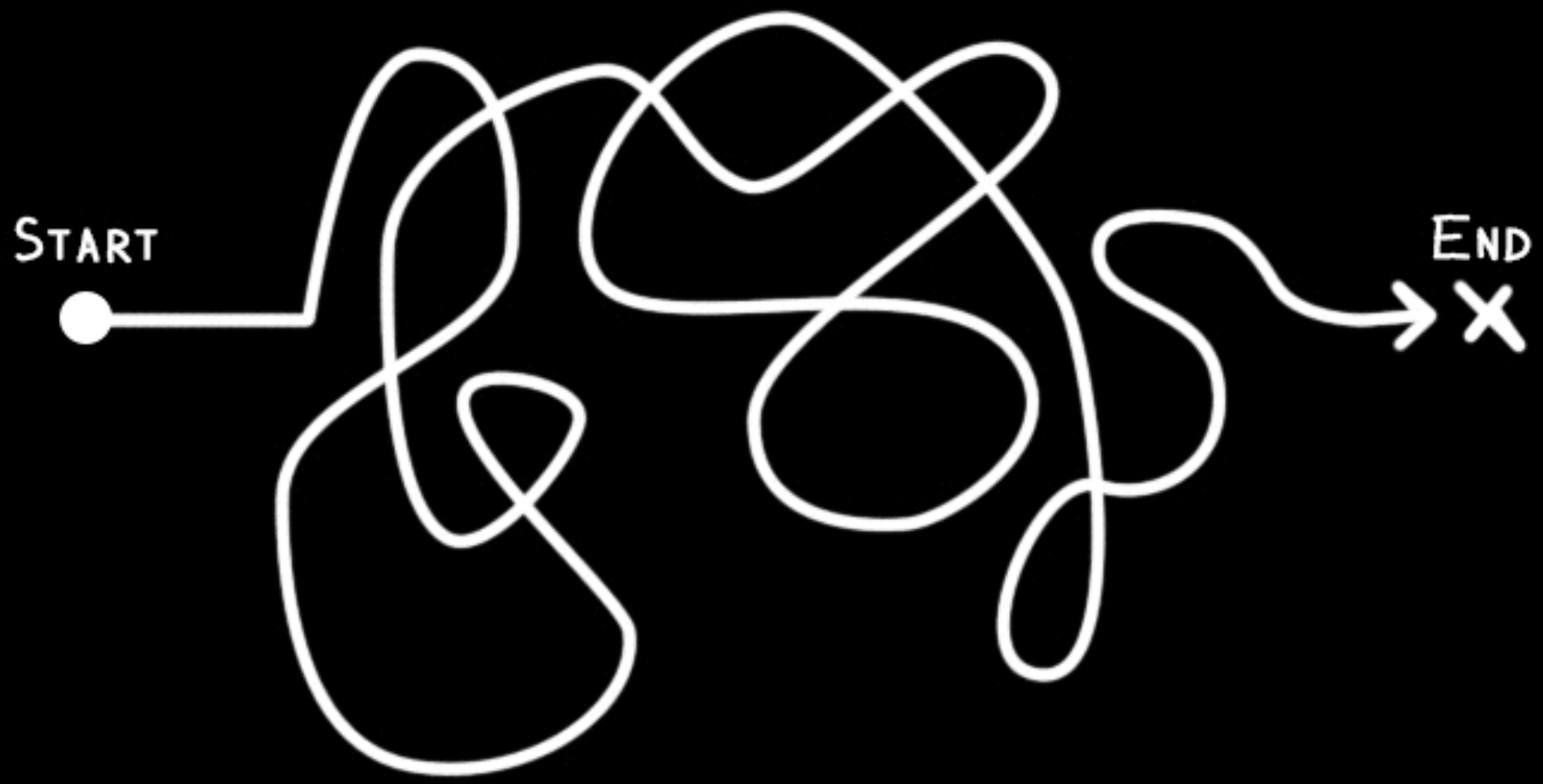
Kai-Feng Chen
National Taiwan University

RECALL FROM THE LAST LECTURE…

- We have introduced the basic concept of machine learning, imported the classical MNIST dataset for handwriting digits recognizing, learned how to do the feature extraction, inject the selected features into LDA and linear SVM, in the end, we have input all of the pixels into SVM and reached a test accuracy of ~92%…
- We have only introduced simple and linear methods so far. Before the end we tried a **SVM with non-linear kernel** but the performance only improves a little. Can we do better?
- In the second half of this lecture we will start to talk about another classical algorithm, **Neural Network**.

HOW LIFE IS SUPPOSED TO GO

HOW LIFE ACTUALLY GOES



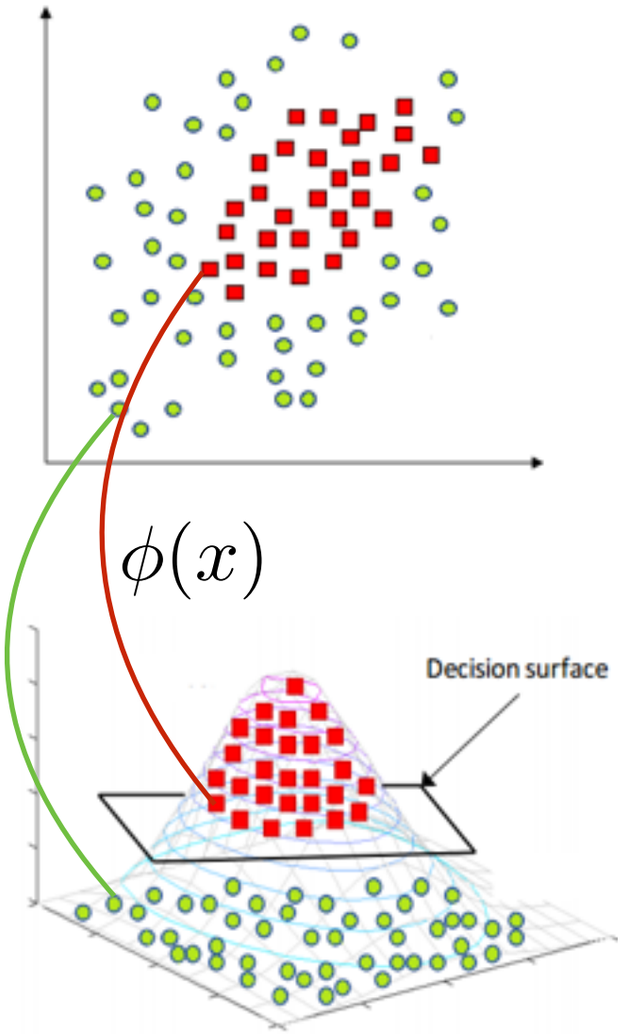
REVIEW: NONLINEAR SVM

- Remember — the idea is to transform the data with a **kernel trick**, and allows the algorithm to fit the margin hyperplane in a **transformed feature space**. The classifier finds a hyperplane in the transformed space, the plane can be be nonlinear in the original space. Some common kernels:
  - *Polynomial*

$$k(\vec{x}_i, \vec{x}_j) = (\gamma \vec{x}_i \cdot \vec{x}_j + \eta)^d$$

  - *Gaussian / Radial basis function (RBF)*

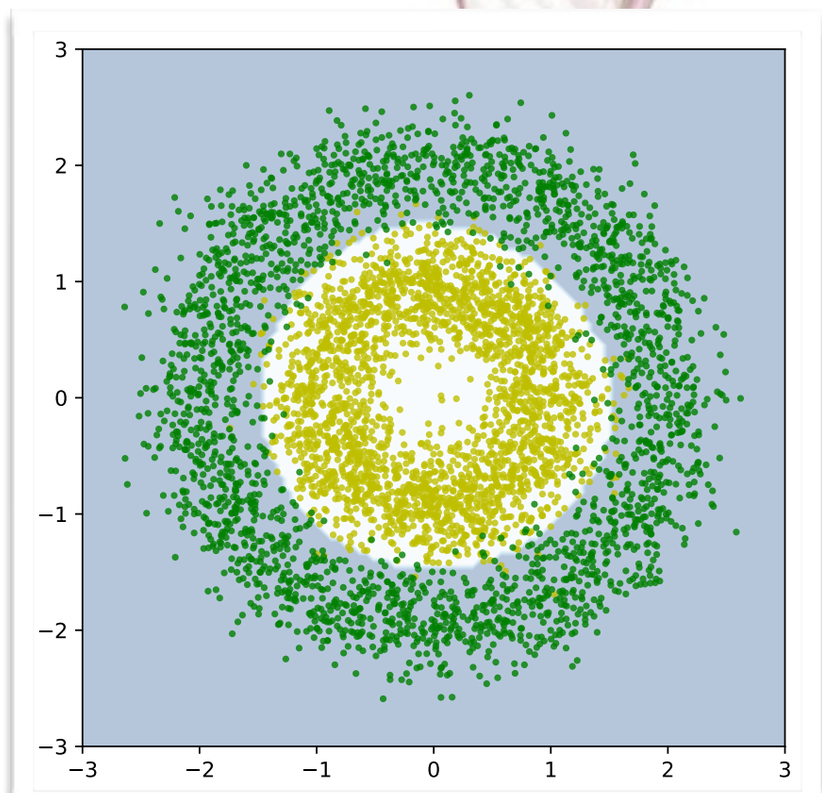
$$k(\vec{x}_i, \vec{x}_j) = \exp(-\gamma |\vec{x}_i - \vec{x}_j|^2)$$



Remark: parameters have to be tuned!

REVIEW: NONLINEAR SVM (II)

- The RBF kernel works mostly out-of-box for our "double donuts" example before the end of last lecture.
- But it does not work for the real problem of handwriting digits separation.




```
clf = svm.SVC(kernel='rbf', C=1.)
clf.fit(x_train, y_train)

s_train = clf.score(x_train, y_train)
print('Performance (training):', s_train)
```

1302-example-01.py (partial)

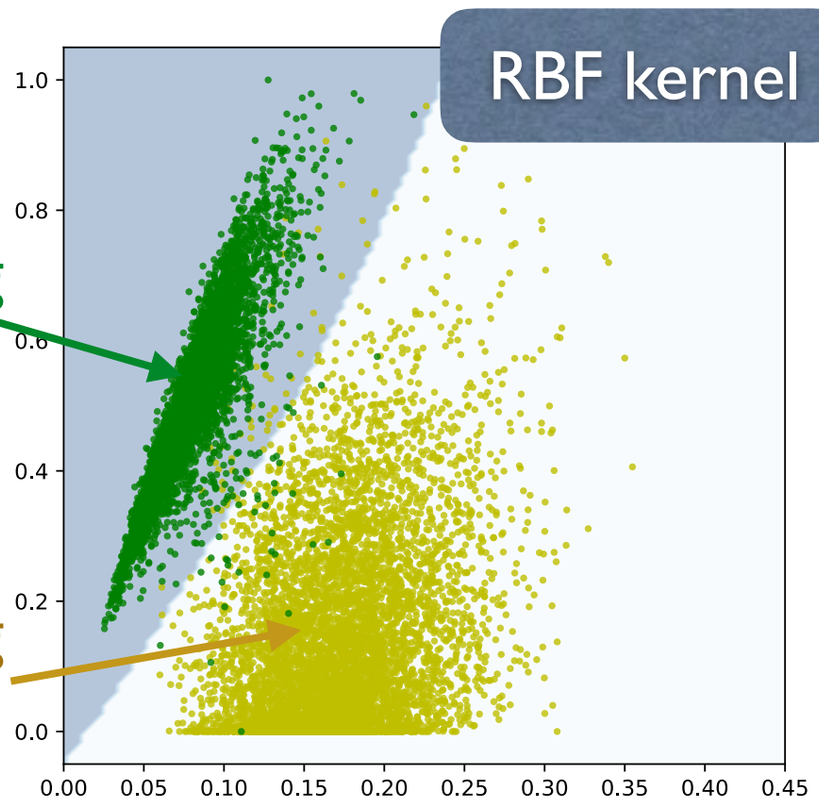
Let's "roll back" to the initial/simplified problem of
Digit 0 & 1 classification with 2 features only

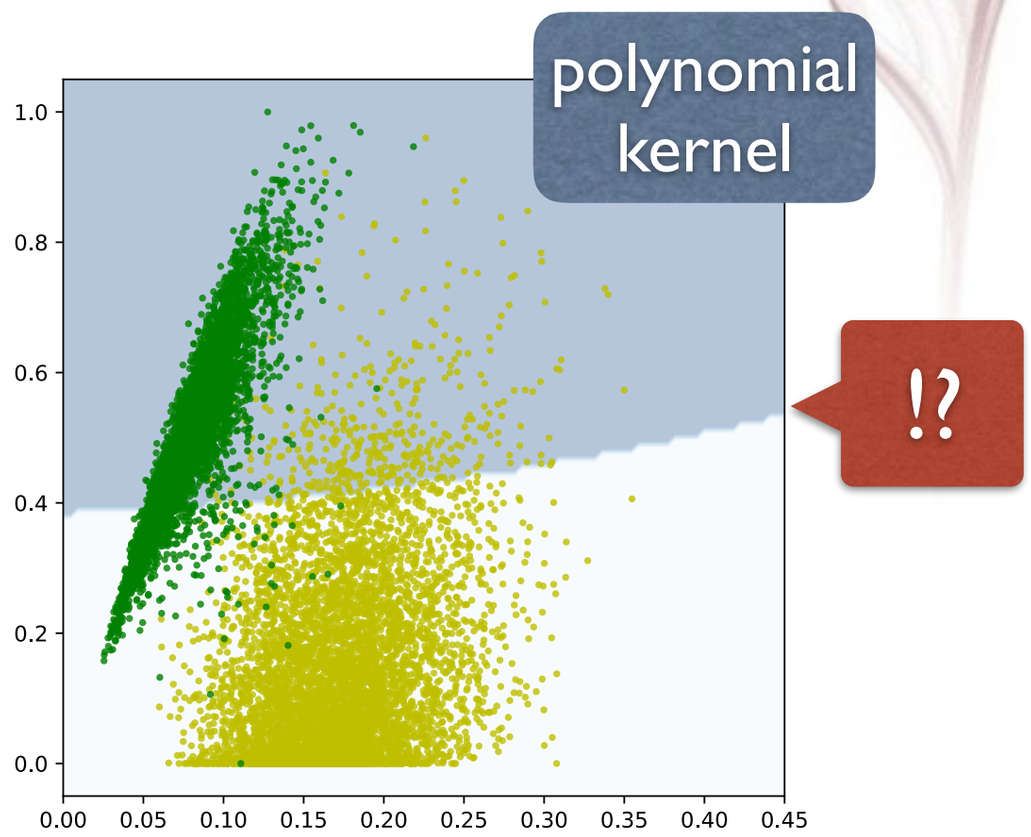
EFFECT OF A NONLINEAR KERNEL

- Switching to a non-linear kernel is easy, but..?

Handwriting true ones

Handwriting true zeros






```
clf = svm.SVC(kernel='rbf', C=1.)
clf.fit(x_train, y_train)
```

l302-example-02.py (partial)

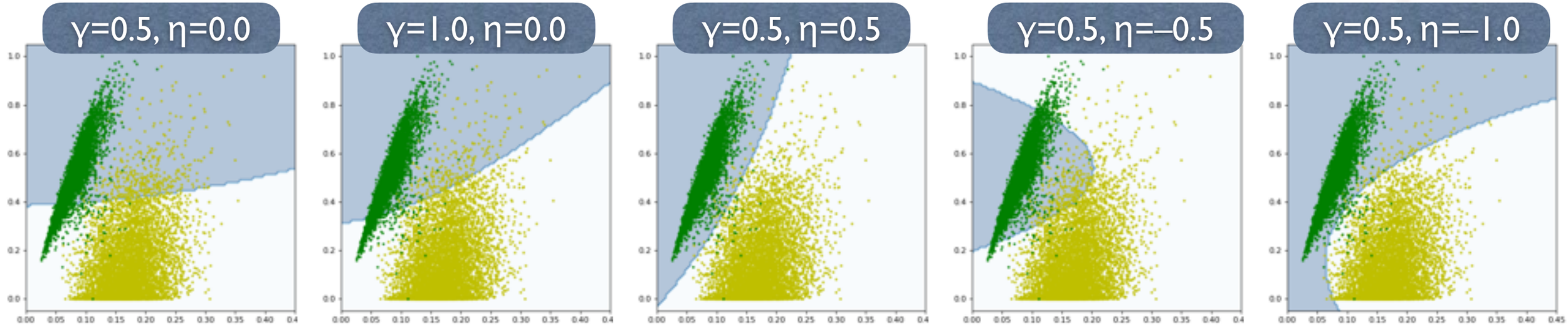
```
clf = svm.SVC(kernel='poly', C=1.)
clf.fit(x_train, y_train)
```

l302-example-02.py (partial)

HOW ABOUT USING A NONLINEAR KERNEL? (II)

- This is simply due to the parameters in the kernel usually needed to be tuned! For example:

Polynomial kernel: $k(\vec{x}_i, \vec{x}_j) = (\gamma \vec{x}_i \cdot \vec{x}_j + \eta)^d$

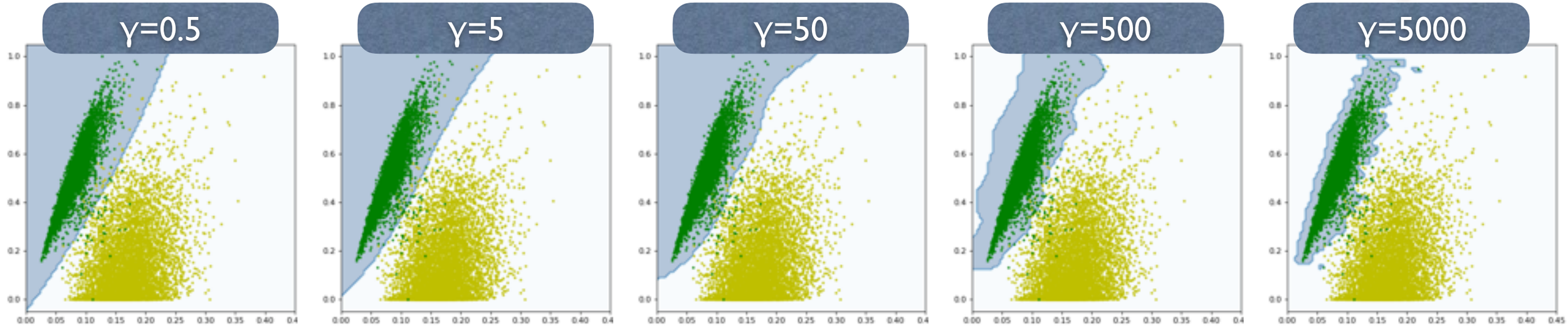


Those parameters γ, η (and the regularization C) used in SVM models are usually called the **Hyperparameters**, and they should be tuned to get the best performance.

HOW ABOUT USING A NONLINEAR KERNEL? (III)

- This also happens to the RBF (Gaussian) kernel. The γ parameter indicates the width of Gaussian function:

RBF kernel: $k(\vec{x}_i, \vec{x}_j) = \exp(-\gamma|\vec{x}_i - \vec{x}_j|^2)$



Small γ = wide Gaussian

Large γ = narrower Gaussian

Obviously it is important to choose a good parameter!

HYPERPARAMETER OPTIMIZATION

- In most of the ML algorithms, hyperparameter optimization is a step need to be carried out to get the optimal performance.
- The tuning can be carried out by simply trying several reasonable (based on experience) setups, or make an exhaustive searching in the allowed parameter space.
- As an example, let's tune the two hyperparameters in the **rbf kernel**, the regularization constant C and the kernel hyperparameter γ, with a classical **grid search**:

C
0.5
5
50
500

γ
2
1
0.5
0.25

C,γ = 0.5, 2
C,γ = 0.5, 1
C,γ = 0.5, 0.5
…

Just try all of the possible combinations!

HYPERPARAMETER OPTIMIZATION (II)

- Within scikit-learn, there is a tool can automatically help to try all of the proposed combinations. Surely you can also perform such an optimization by yourself!

```
import numpy as np
from sklearn import svm
from sklearn.model_selection import GridSearchCV

. . . . . .

clf = svm.SVC(kernel='rbf')

param = {'C':[0.5,5.,50.,500.], 'gamma':[2.0,1.0,0.5,0.25]}
grid = GridSearchCV(clf, param, verbose=3)
grid.fit(x_train, y_train)  ⇐ The grid search tool has a similar interface.

print('Best SVM:')
print(grid.best_estimator_)

s_train = grid.score(x_train, y_train)
s_test = grid.score(x_test, y_test)
print('Performance (training):', s_train)
print('Performance (testing):', s_test)
```

1302-example-02a.py (partial)

HYPERPARAMETER OPTIMIZATION (III)

- Terminal output, as an automatic grid search:

```
Fitting 3 folds for each of 16 candidates, totalling 48 fits
[CV] C=0.5, gamma=2.0 ....................................................
[CV] ....... C=0.5, gamma=2.0, score=0.994080037887576, total=   0.1s
. . . . . . . .
[CV] .... C=500.0, gamma=0.25, score=0.995027231825716 3, total=   0.1s
[CV] C=500.0, gamma=0.25 ....................................................
[CV] .... C=500.0, gamma=0.25, score=0.991234304667140 5, total=   0.1s
[CV] C=500.0, gamma=0.25 ....................................................
[CV] .... C=500.0, gamma=0.25, score=0.994551054252546 8, total=   0.1s
[Parallel(n_jobs=1)]: Done  48 out of  48 | elapsed:    8.1s finished
Best SVM:
SVC(C=50.0, cache_size=200, class_weight=None, coef0=0.0,
  decision_function_shape='ovr', degree=3, gamma=0.5, kernel='rbf',
  max_iter=-1, probability=False, random_state=None, shrinking=True,
  tol=0.001, verbose=False)
Performance (training): 0.993604421634
Performance (testing): 0.995271867612
```

DIGITS RECOGNITION W/ NONLINEAR SVM

- Let's revisit the handwriting digits recognition (full version) again, by simply modify the ending example from the previous lecture to a nonlinear kernel.

With slightly tuned SVM + RBF kernel, the performance can be better than the previous version!

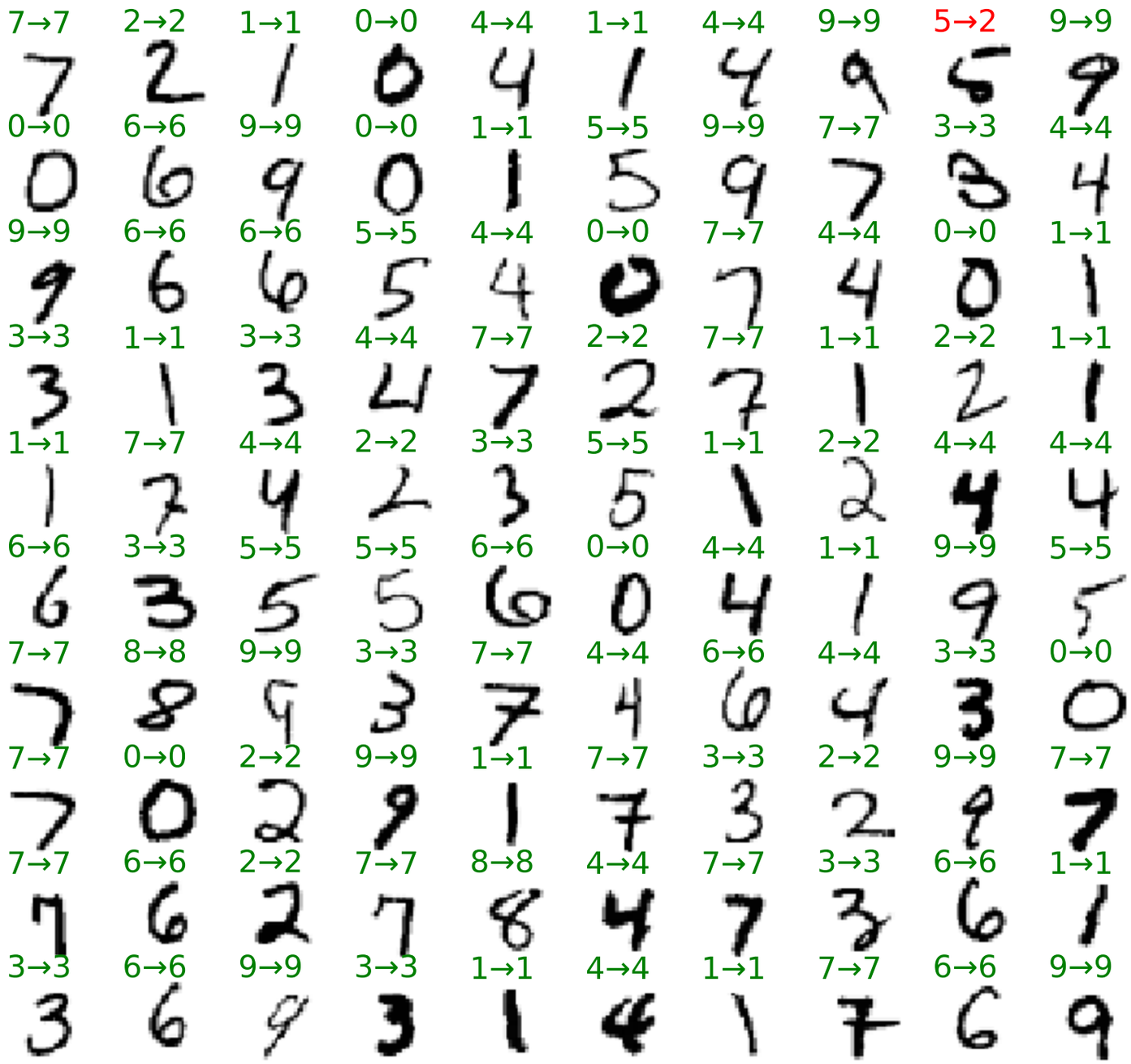
```
optimization finished, #iter = 1523
obj = -158.940057, rho = -0.326238
nSV = 928, nBSV = 0
Total nSV = 6202
Performance (training): 1.0
Performance (testing): 0.9664
```

it was 0.917 before!

```
. . . . . .

clf = svm.SVC(C=5., gamma=0.05, verbose=True)
clf.fit(x_train, y_train)

s_train = clf.score(x_train, y_train)
s_test = clf.score(x_test, y_test)
print('Performance (training):', s_train)
print('Performance (testing):', s_test)
```

1302-example-03.py (partial)

DIGITS RECOGNITION W/ NONLINEAR SVM (II)



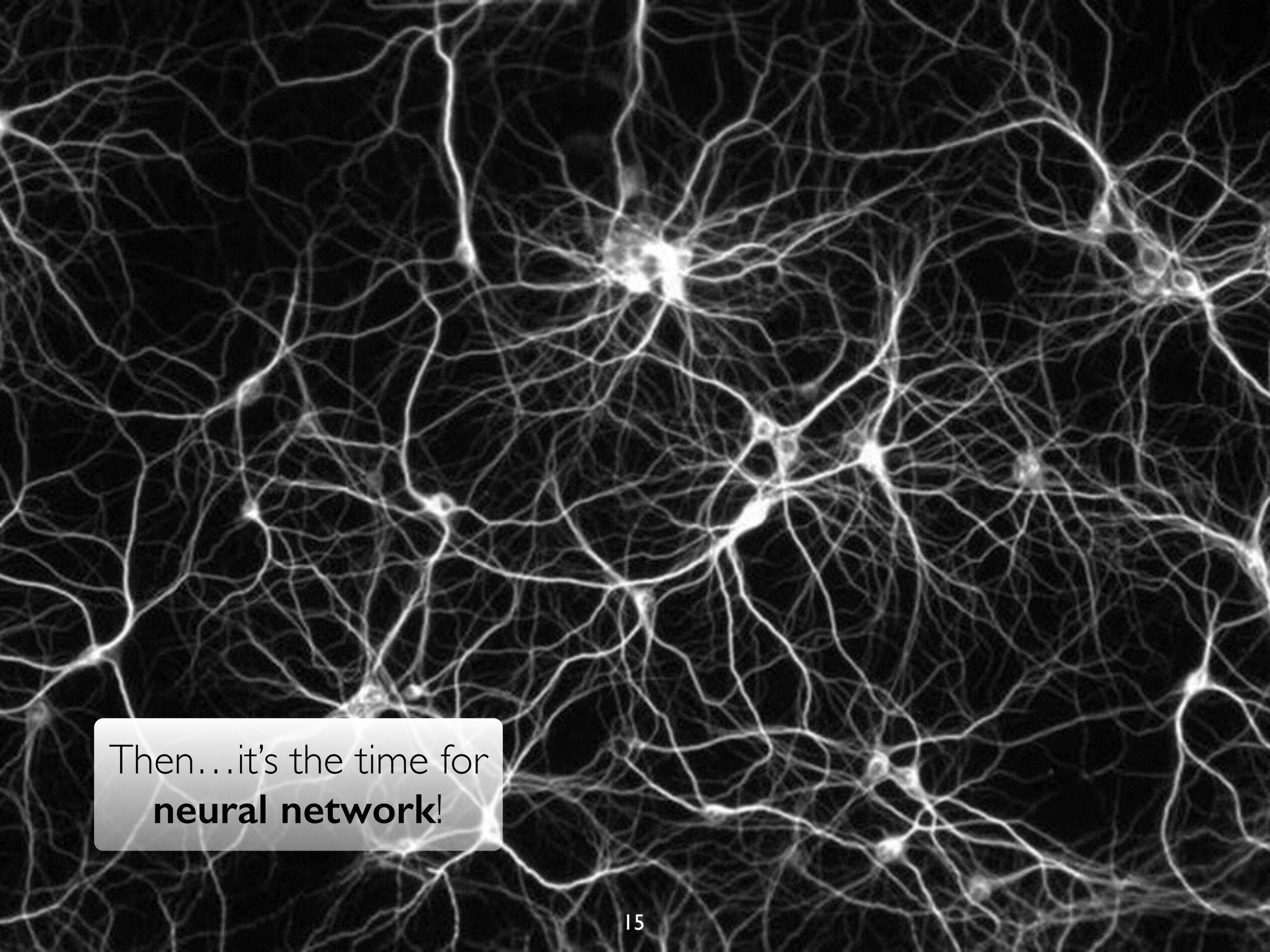
- Only 1 misidentification found in the first 100 digits!
- In fact if you **inject all of the training samples** (60K instead of 10K images). The performance would be superior (**~98.4%** of accuracy), but it will take a while to run!

INTERMISSION

- As we mentioned the hyperparameters (here are the C and γ as the Gaussian SVM) are essential for the ML algorithm performance. In our previous full digits recognition example, we have already set

```
clf = svm.SVC(C=5., gamma=0.05, verbose=True)
```

  which gives a good performance. What will be the performance if you set the parameter to the default setting?
- The training data is also an essential element for ML performance. As mentioned in the previous page, the performance would be superior if you inject all of the data. It would be interesting for you to try it once!


Then…it's the time for
neural network!

ARTIFICIAL NEURAL NETWORK

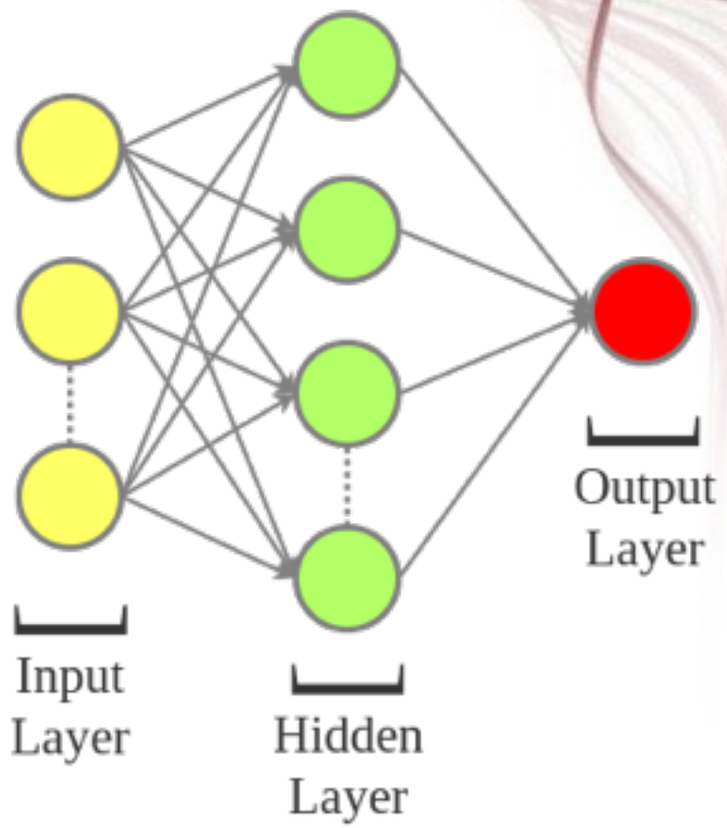
- An **artificial neural network (ANN)**, usually just called "neural network" (NN), is a mathematical model or computational model that tries to simulate the structure and/or functional aspects of biological neural networks.
- It consists of an interconnected group of **artificial neurons** and processes information. They are usually used to model complex relationships between inputs and outputs.
- And this is not a new idea at all — was first proposed in 1943 by McCulloch and Pitts. It has been developed for long and used in many applications already. However, with the recent development in the deep neural network, or **deep learning**, it becomes extremely powerful in many of the ML applications.

WHAT ARE WE GOING TO DO HERE?

- As we have been doing for many weeks already, we want to talk about the core concepts of the method and algorithm, maybe implement a simple version for helping the understanding. And we will switch to one of the existing packages for further practice.
- So in this lecture we are going to introduce and **implement a simple NN** and show you how things work. After that we will demonstrate how to do exactly the same thing with the popular packages (**Keras** and **Tensorflow**).
- For a further improvement of the network, we will touch it in the next lecture and eventually approach to the idea of deep learning.

Hopefully this will give you a slightly better understanding of **"why it can work"** than just show you the modern fancy tools!

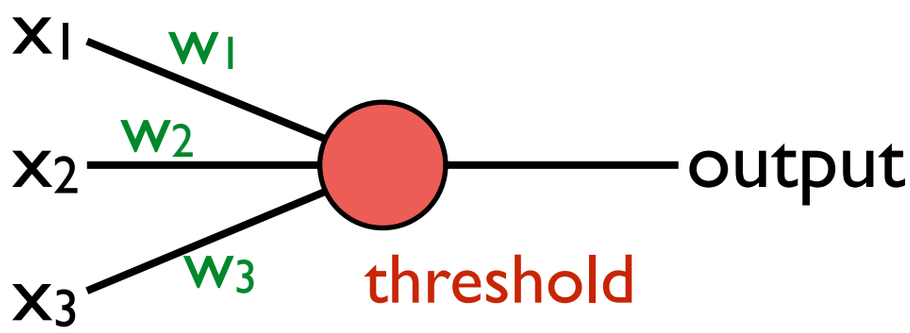
PERCEPTRONS AND NEURON MODELS



- You may have seen such a neural network schematics from many various sources:
- It shows that a network is usually built by connecting multiple neurons (the “circles” in the diagram). Hence the **neurons** are the most basic building block of the neural network.
- Here the first step is going to implement a very classical neuron model, called the **“sigmoid neurons”**, but however, it would be nice to first understand the **perceptrons** before doing that!

PERCEPTRONS AND NEURON MODELS (II)

- A perceptron usually takes multiple binary inputs, e.g. x_1, x_2, …, and produces a single binary output:



- Given the inputs are either 0 or 1, the idea is to introduce a weight for each input and estimate the **weighted sum of the inputs**, $\sum w_i x_i$. The output can be determined by whether the weighted sum bypass a **threshold** or not, just like a **logic gate**:

$$\text{output} = \begin{cases} 1 & \text{if } \sum w_i x_i > \text{threshold} \\ 0 & \text{if } \sum w_i x_i \leq \text{threshold} \end{cases}$$

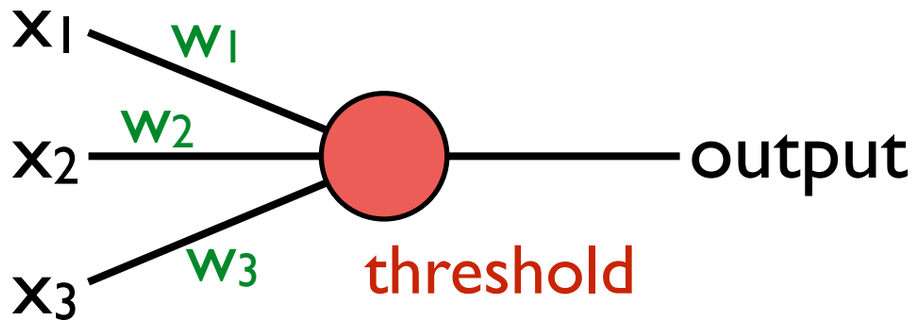
PERCEPTRONS AND NEURON MODELS (III)

- It's all about the decision making — you can in fact, model a daily life problem with such a simple perceptron model. For example, consider a decision of having an ice cream cone or not:

Do you have enough coins in your packet?

Is it hot today?

Did you have a big lunch today or not?

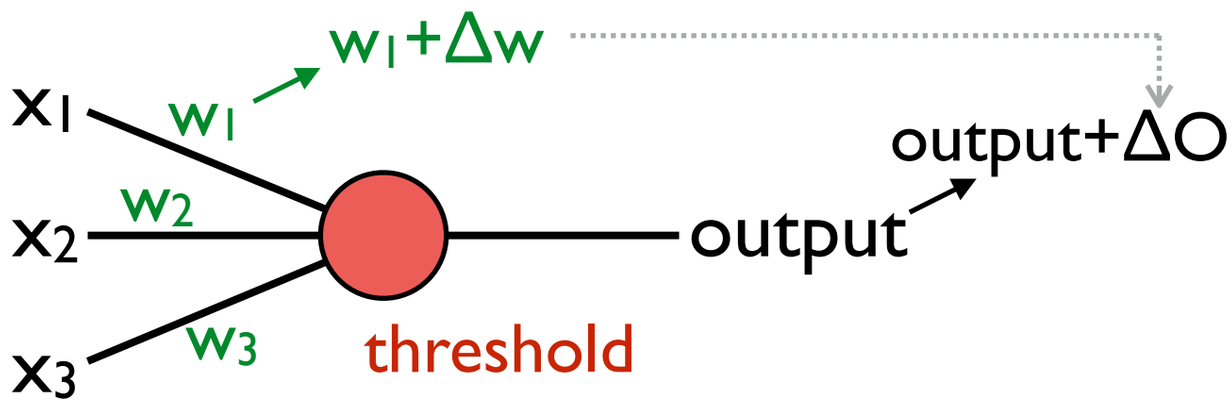


Eat or not to eat, that's the question!

  - **Weights**: decides the importance of each inputs, *e.g. do you care about the weather or how much food in your stomach?*
  - **Threshold**: decides the action taking criterion, *e.g. a value to determine your love of ice cream.*

SIGMOID NEURONS

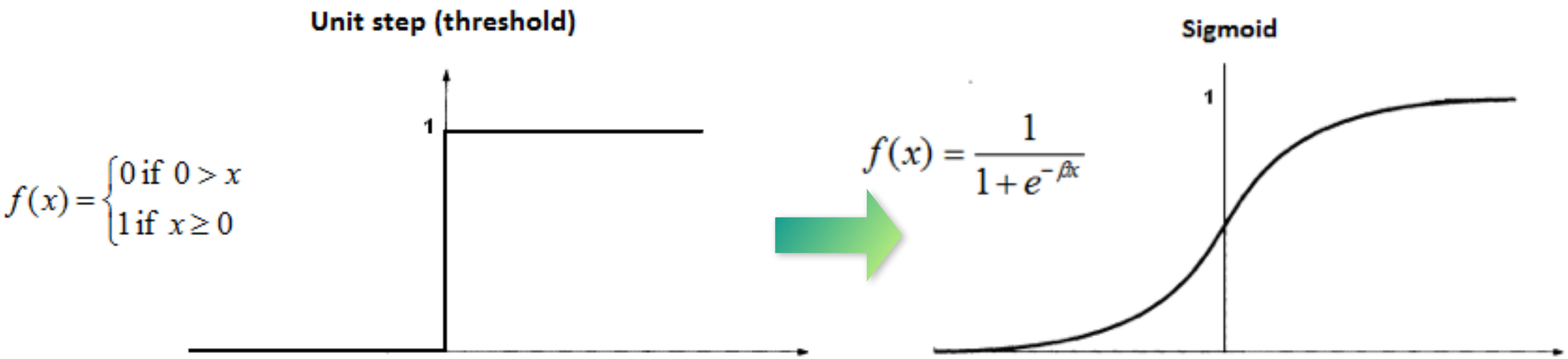
- Although a perceptron design sounds rather reasonable and it can be used to model all kinds of logic gates, but it has a critical issue in the implementation of machine learning.
- The usual learning is carried out by changing the weights or thresholds a little bit and check whether the output is improved or not. e.g.



- Given the inputs and outputs are only binary in perceptrons, this will not work.

SIGMOID NEURONS (II)

- The key idea is to introduce an **activation function** at the core of neuron, which can smooth the binary operation to some continuous function and still keep the nice property of perceptrons, for example a **sigmoid function**:



$$z = \sum_i w_i x_i + b, \qquad \text{output} = \sigma(z) = \frac{1}{1+\exp(-z)}$$

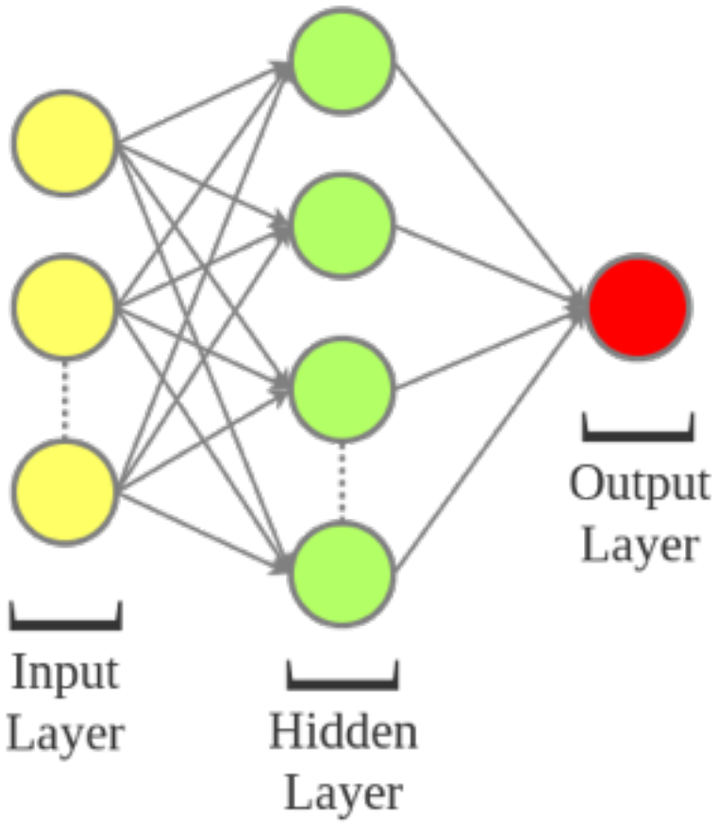
Here we take the **"bias"** to replace **"–threshold"**, which is mathematically the same!

$$\Delta O \approx \sum_i \frac{\partial O}{\partial w_i}\Delta w_i + \frac{\partial O}{\partial b}\Delta b$$

It becomes differentiability!

NETWORKS ARCHITECTURE

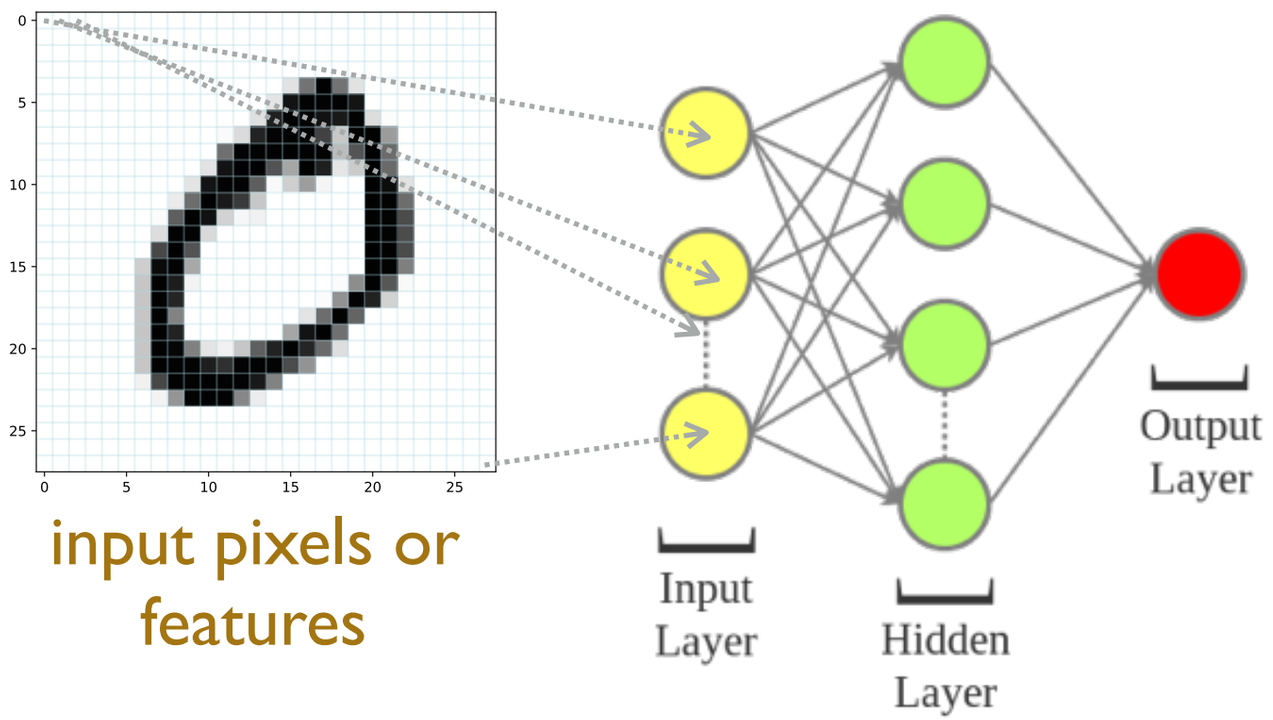
- As we just discussed, a single neuron is like a logic gate. If one want to handle a more complex problem, it is necessary to incorporate multiple neurons and hence the network architecture becomes essential.
- A typical structure is like this, as **multilayer perceptrons (MLP)**:



  - There can be multiple **input neurons**, for handling the actual input features.
  - There can be multiple **hidden layers** of neurons, without connecting to the input nor output directly.
  - There can be multiple **output neurons** as well.

NETWORKS ARCHITECTURE (II)

- If the network process the information only in one way (no loops) — e.g. data always feeds forward, never feeds back, this is usually called the **feedforward neural networks**.
- In such a case the whole network can be imaged as a huge and complicated function, and each output is a function of inputs, with all of the *weights* and *biases* as parameters of the function.

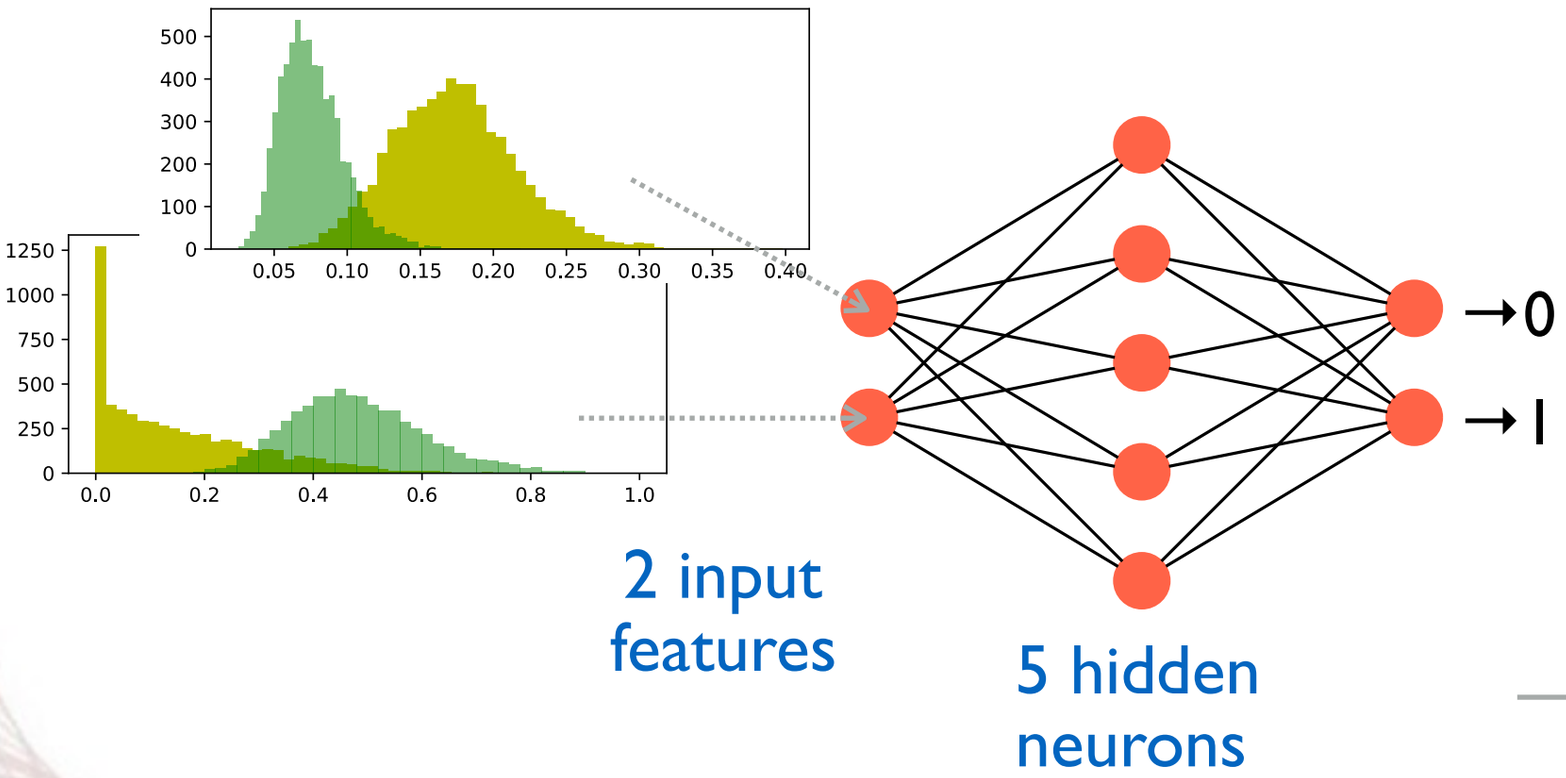


{ ~1 if the input image looks like a "zero"

~0 if the input image *doesn't* look like a "zero"

NETWORKS ARCHITECTURE (III)

- Consider the earlier example of separating handwriting digits of zeros and ones, with the two reduced features (*only full pixel average and centered average*) and **fully connected network**.
- If we put in 1 layer of hidden 5 neurons, and two outputs (for 0 and 1 digit), this is the structure one expected to construct:



# of biases:

0 (no biases for input)+

5 (hidden neurons)+

2 (output neurons).

# of weights:

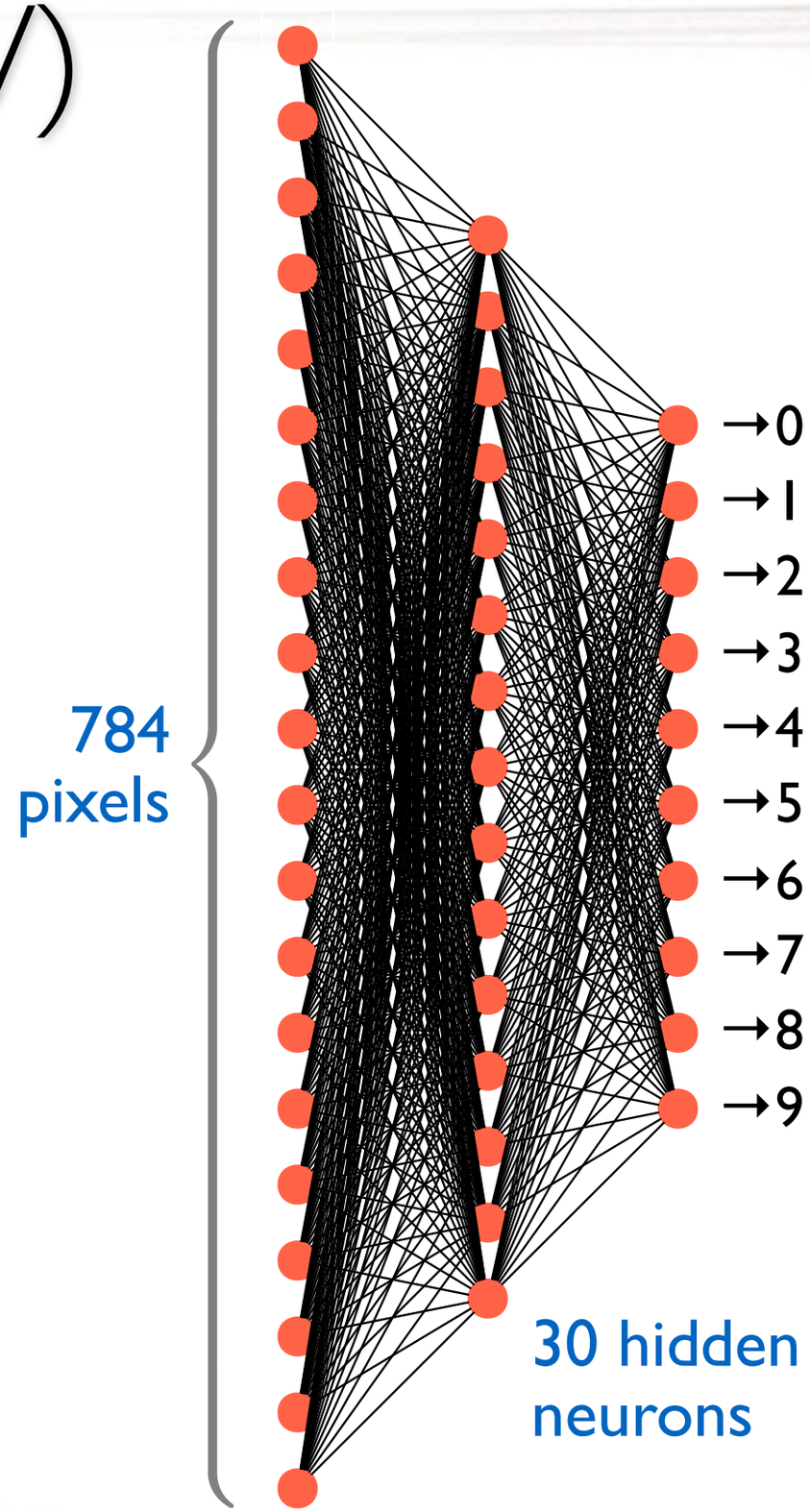
2×5 (input to hidden)+

5×2 (hidden to output).

---

27 parameters in total

NETWORKS ARCHITECTURE (IV)



- How about another possible (*but more complex*) configuration?

# of biases:

0 (no biases for input)+
30 (hidden neurons)+
10 (output neurons).

# of weights:

784×30 (input to hidden)+
30×10 (hidden to output).

23860 parameters in total

Lots of parameters to be optimized in the training!

COMMENT: ANN STRUCTURE

- Restrictedly speaking the term multilayer perceptrons MLP are based on multilayers of sigmoid neurons (*not perceptrons*). This name was there due to some historical reason and it is confusion in fact.
- There more different choice of activation functions. A typical alternative choice is **tanh(z)**, which is exactly the same as sigmoid function but extended to negative. There are also a couple different choices and to be discussed later.
- A typical example of non-feedforward network is the recurrent neural networks (RNN). The main idea in these models is to have neurons which fire for a limited duration of time, and hence it allows loops in the network.

IMPLEMENTATION START!

- Let's start our implementation of a simple feedforward network.
- Remark: my implementation is definitely not prepared for a high performance computation!

```
import numpy as np

def sigma(z):
    return 0.5*(np.tanh(0.5*z)+1.)  ⇐ activation function

class neurons(object):                    constructor: expected to get a tuple or
    def __init__(self, shape):  ⇐ list of network structure, e.g. [2,5,2]
        self.shape = shape
        self.v = [np.zeros((n,1)) for n in shape]
        self.z = [np.zeros((n,1)) for n in shape[1:]]
        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:]]
```

⇑ no bias nor weights for first layer!

neurons.py (partial)

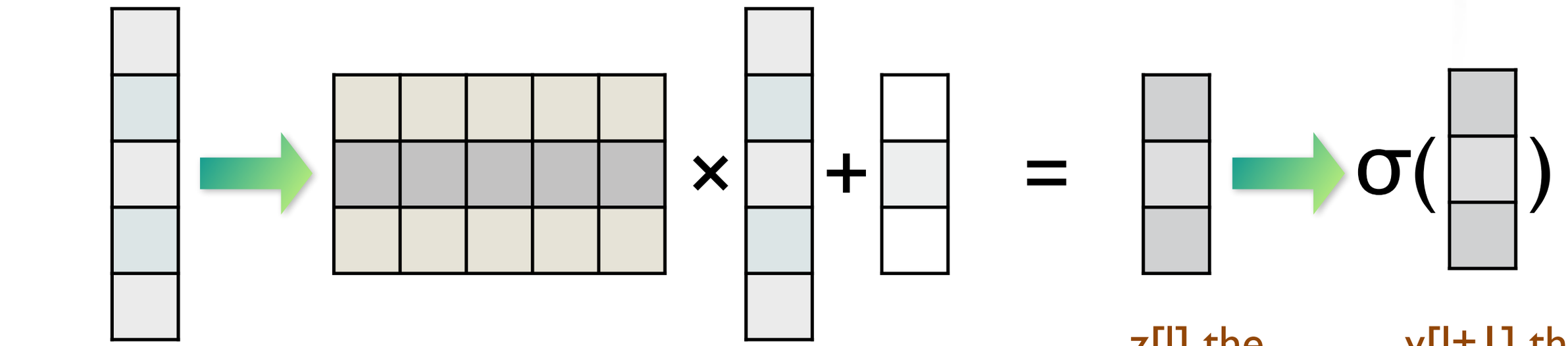
- **v** stores the values of $\sigma(z)$
- **z** stores the values of $z = \sum w_i x_i + b$
- **w** stores the weights
- **b** stores the biases

w, b are initialized as Gaussian random numbers for now!

FEEDFORWARD CALCULATION

```
    def predict(self, x):
        self.v[0] = x.reshape(self.v[0].shape)
        for l in range(len(self.shape)-1):
            self.z[l] = np.dot(self.w[l],self.v[l])+self.b[l]
            self.v[l+1] = sigma(self.z[l])
        return self.v[-1]
```

neurons.py (partial)



v[l] the values at the previous layer

w[l] the weights between previous and the next layer

b[l] the biases at the next layer

z[l] the weighted sum at the next layer

v[l+1] the values at the next layer

THE NN OUTPUT BEFORE TRAINING…

- Well, it is kind of difficult to test the feedforward network itself, but we can in any case show the output before any training:

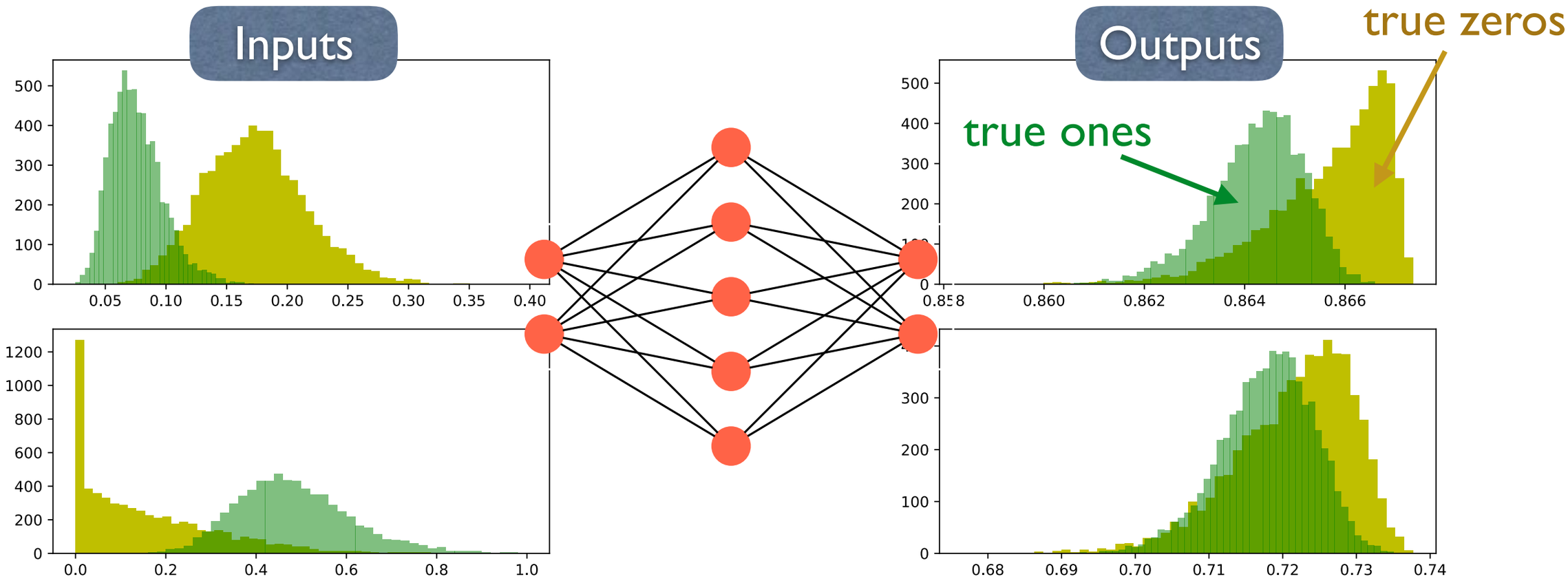
```
mnist = np.load('mnist.npz')
x_train = mnist['x_train'][mnist['y_train']<=1]/255.
y_train = mnist['y_train'][mnist['y_train']<=1]
x_train = np.array([[img.mean(),img[10:18,11:17].mean()] for img in
x_train])
y_train = np.array([[1,0],[0,1]][n] for n in y_train])

from neurons import neurons
model = neurons([2,5,2])
out = np.array([model.predict(x) for x in x_train])

fig = plt.figure(figsize=(6,6), dpi=80)
plt.subplot(2,1,1)
plt.hist(out[:,0][y_train[:,0]==1], bins=50, color='y')
plt.hist(out[:,0][y_train[:,1]==1], bins=50, color='g', alpha=0.5)
plt.subplot(2,1,2)
plt.hist(out[:,1][y_train[:,0]==1], bins=50, color='y')
plt.hist(out[:,1][y_train[:,1]==1], bins=50, color='g', alpha=0.5)
plt.show()
```

⇐ calling the "neurons" class we just prepared

1301-example-04.py (partial)

THE NN OUTPUT BEFORE TRAINING…(II)

- Before the training the network is acting like a random smearing function of the inputs.

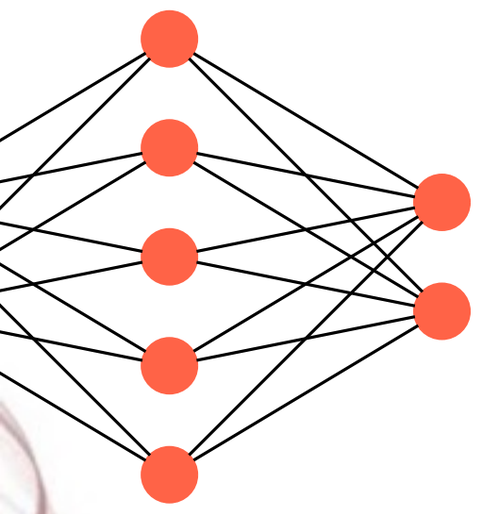


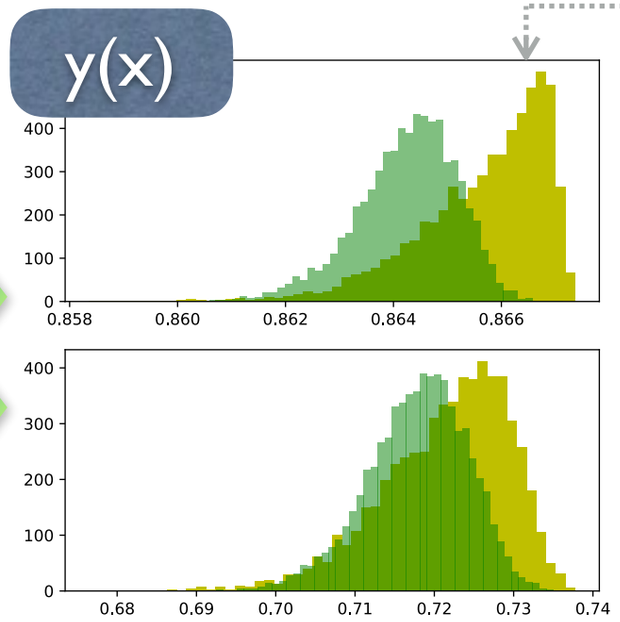
Now we should start to **"train"** the network to reach its maximum separation power at the outputs!

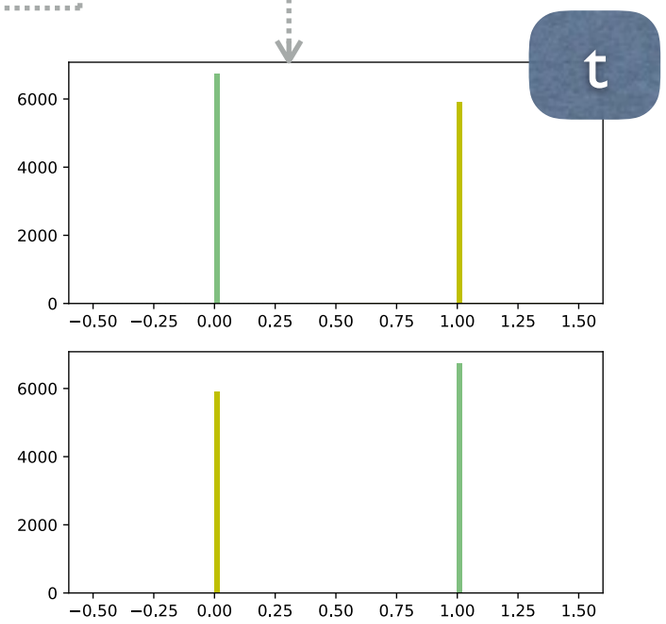
TRAINING: THE GOAL

- In order to train our network, it is required to define a **loss function**, which indicates the distance between the current output and their target values. A typical choice can be this **mean squared error (MSE)**, it should be minimized in the training process:

$$\mathrm{Loss}(w_i, b_j) = \frac{1}{2n} \sum_{x}^{n} |y(x) - t|^2$$







True zeros	True ones
1	0
0	1

TRAINING: THE METHOD

- One of the classical algorithms to minimize the loss function is the **gradient descent** method. To find a local minimum of the loss function, one takes steps **proportional to the negative of the gradient** of the function at the current point, e.g.

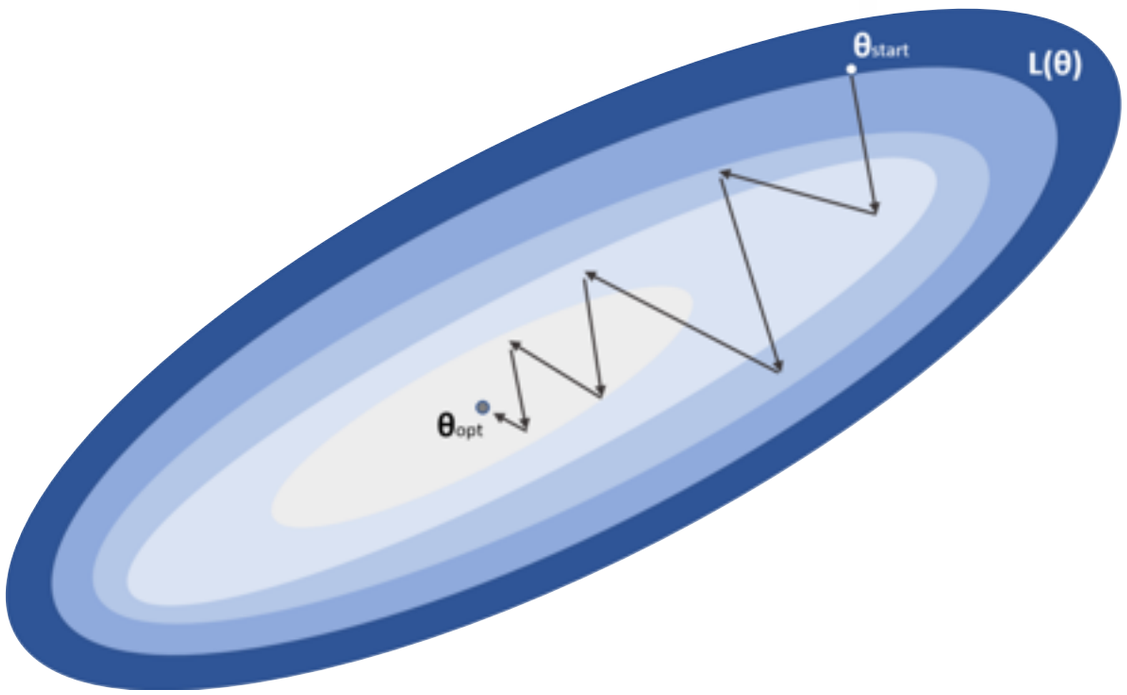
$$\theta \to \theta' = \theta - \eta \nabla L$$

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

More explicitly:

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

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



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 data:

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

- 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:

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

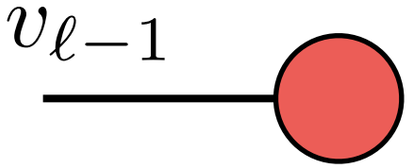
Such a small subset is usually called the **mini-batch**.

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 **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:



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

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

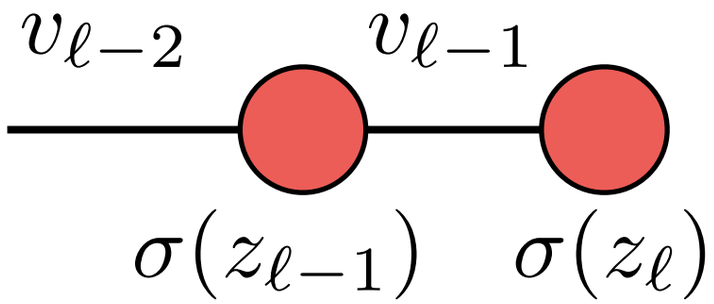
$$\Rightarrow \delta_\ell = \frac{\partial L}{\partial v_\ell}\frac{\partial v_\ell}{\partial z_\ell} = [\sigma(z_\ell) - t] \cdot \sigma'(z_\ell)$$

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

The differentials at the ending layer can be calculated easily!

BACK PROPAGATION (II)

- Then propagate back by another layer of single neuron:



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

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

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

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

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`).

```
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]
```

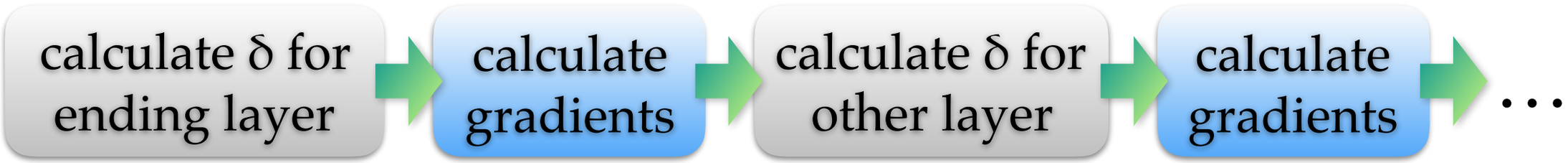
neurons.py (partial)

IMPLEMENTATION: BACK PROPAGATION (II)

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

```
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)
```

neurons.py (partial)



IMPLEMENTATION: TRAINING WITH SGD

```
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 + backpropagation

Update weights/bias

⇐ measure the performance

neurons.py (partial)

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.

```
def evaluate(self, x_data, y_data):
    loss, cnt = 0., 0.
    for x,y in zip(x_data, y_data):
        self.predict(x)⇐ feedforward
        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)
```

neurons.py (partial)

Now we are ready to train…your network!

LET'S TRAIN THE NETWORK

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

```
. . . . . .
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)
. . . . . .
```

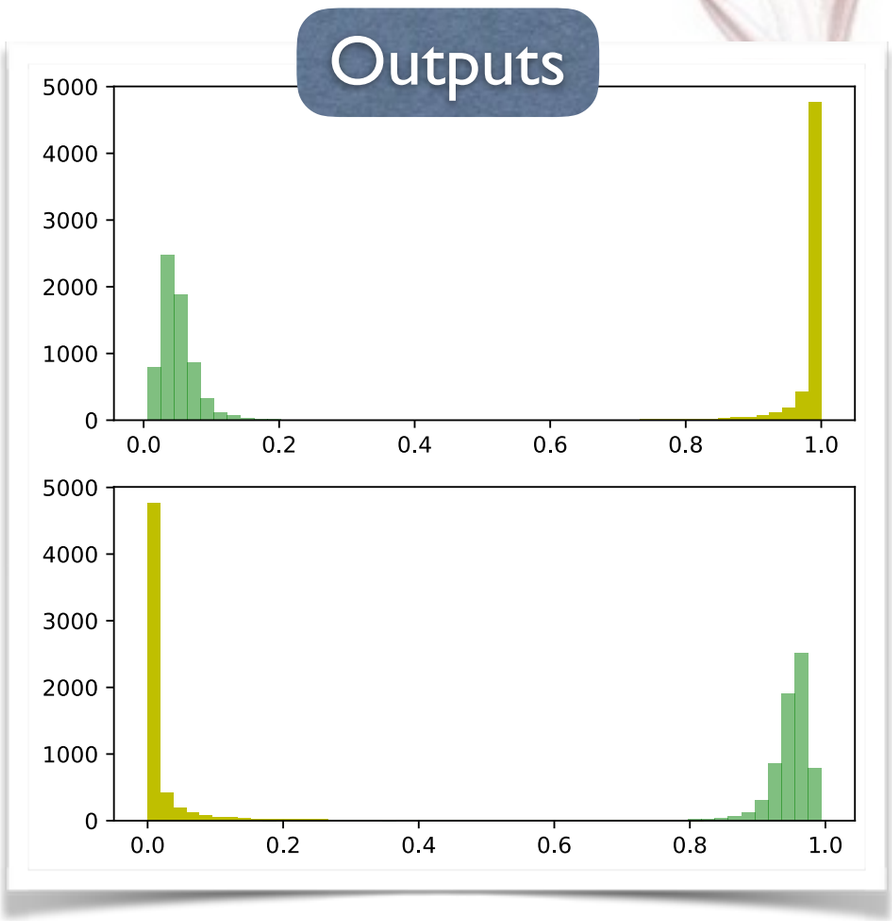
⇐ with 20 epochs, mini-batch of 30
learning rate = 1.0

l302-example-04a.py (partial)

LET'S TRAIN THE NETWORK (II)

- You can see the performance increases as epochs:

```
Epoch: 1/20
Loss: 0.0904, Acc: 0.9173
Epoch: 2/20
Loss: 0.0596, Acc: 0.9365
Epoch: 3/20
Loss: 0.0469, Acc: 0.9455
Epoch: 4/20
Loss: 0.0356, Acc: 0.9655
. . . . . .
Epoch: 20/20
Loss: 0.0075, Acc: 0.9928
Performance (training)
Loss: 0.00754, Acc: 0.99281
Performance (testing)
Loss: 0.00546, Acc: 0.99574
```



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?

```
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.                                    ⇐ data prepared
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)  ⇐ with 20 epochs, mini-batch of 10
                                            learning rate = 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))
```

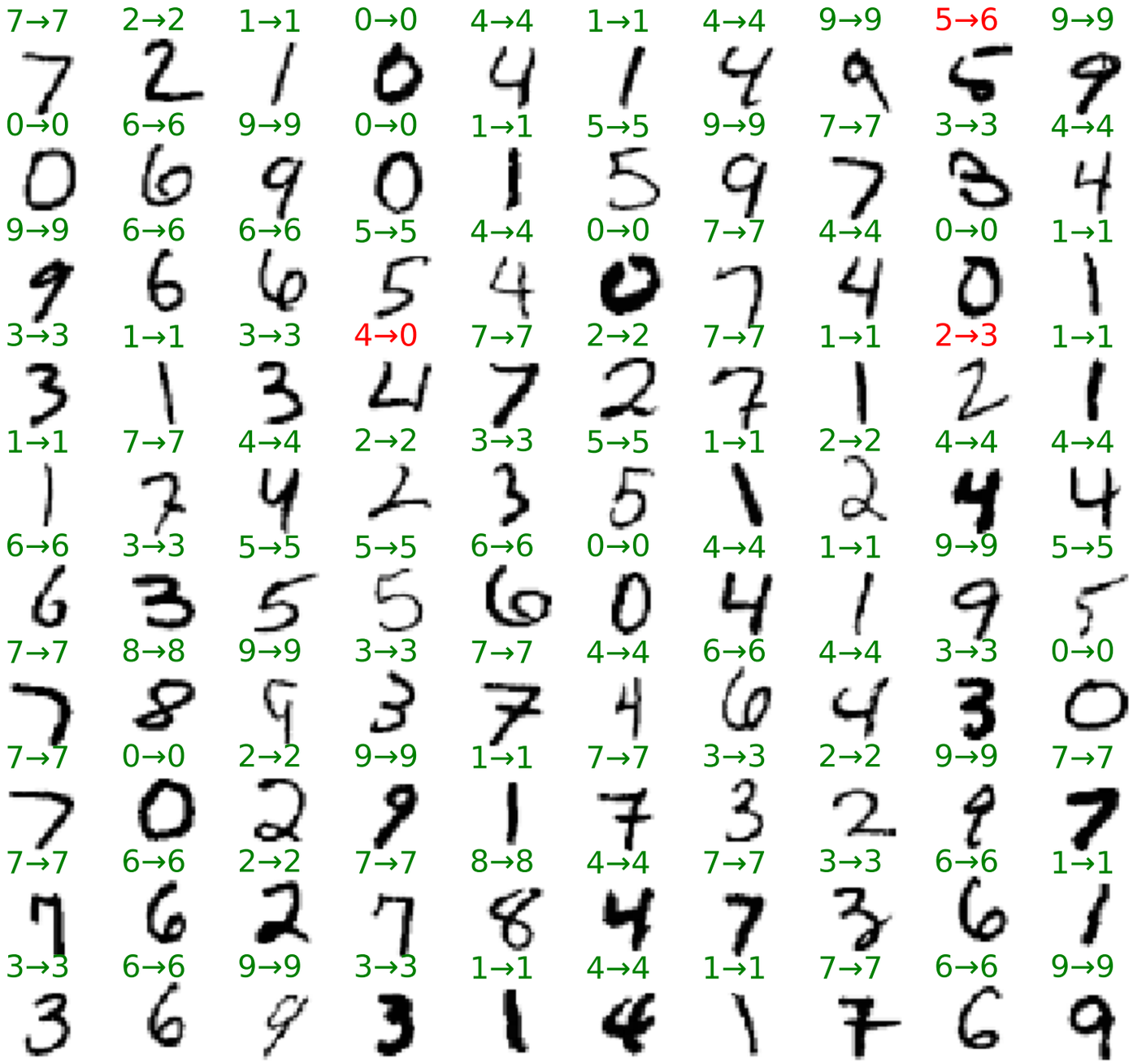
l302-example-05.py (partial)

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: 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:

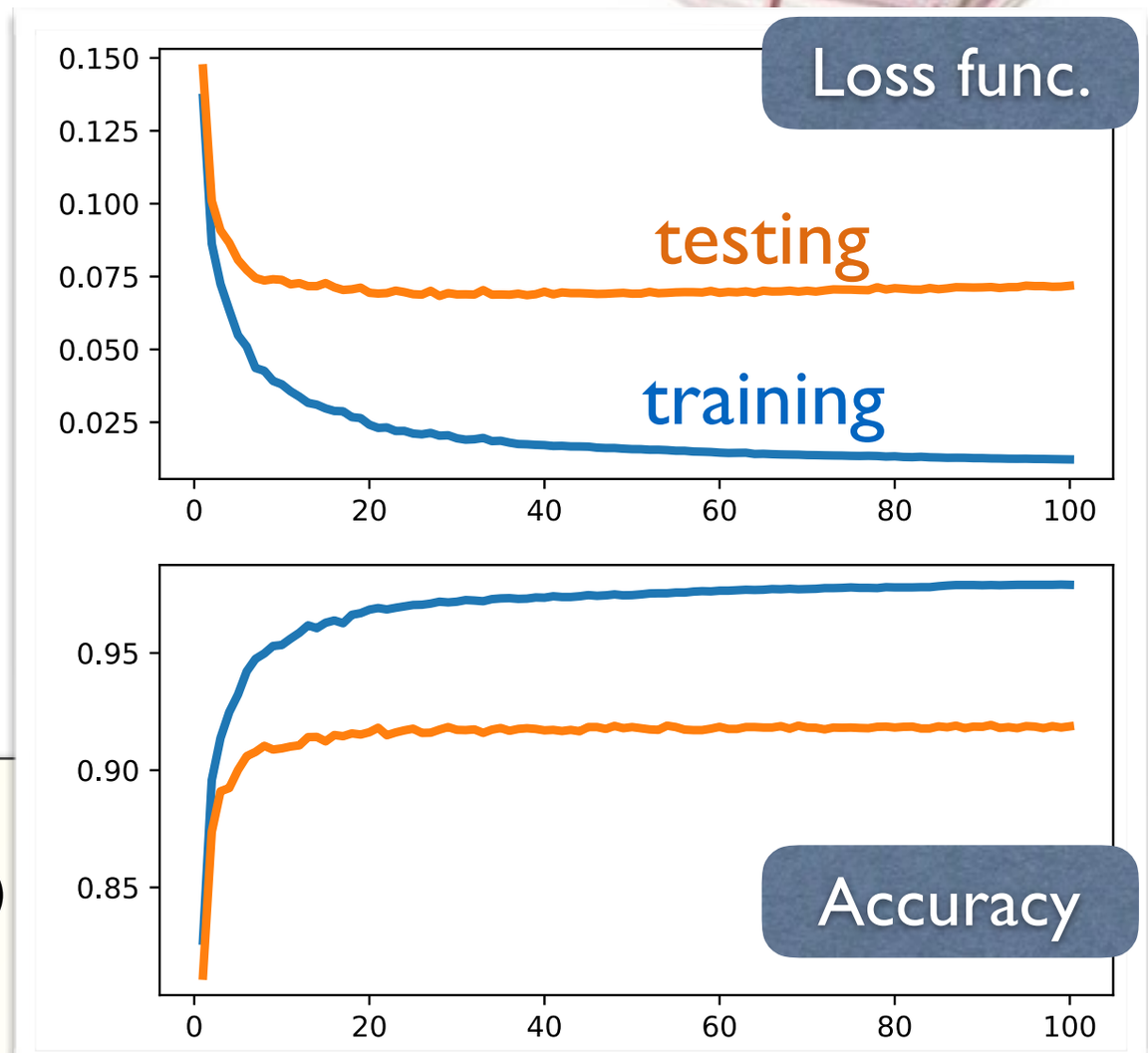
```
scores = np.zeros((4,100))

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)
```

⇐ stores the loss function & accuracy for 100 epochs

1302-example-05a.py (partial)

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.




```
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()
```

l302-example-05a.py (partial)

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

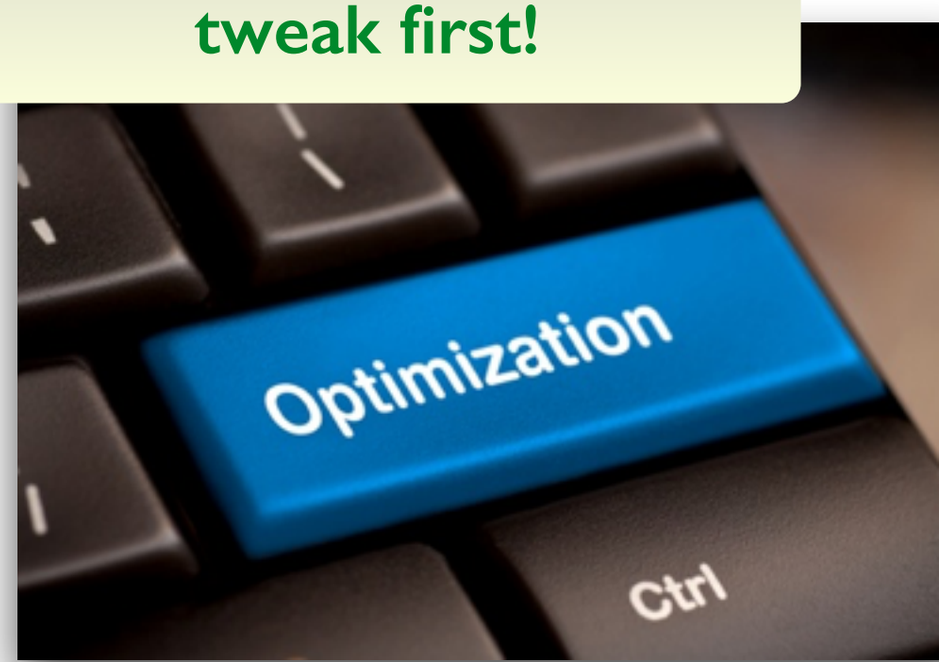
COMMENT: NETWORK STRUCTURE

- 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

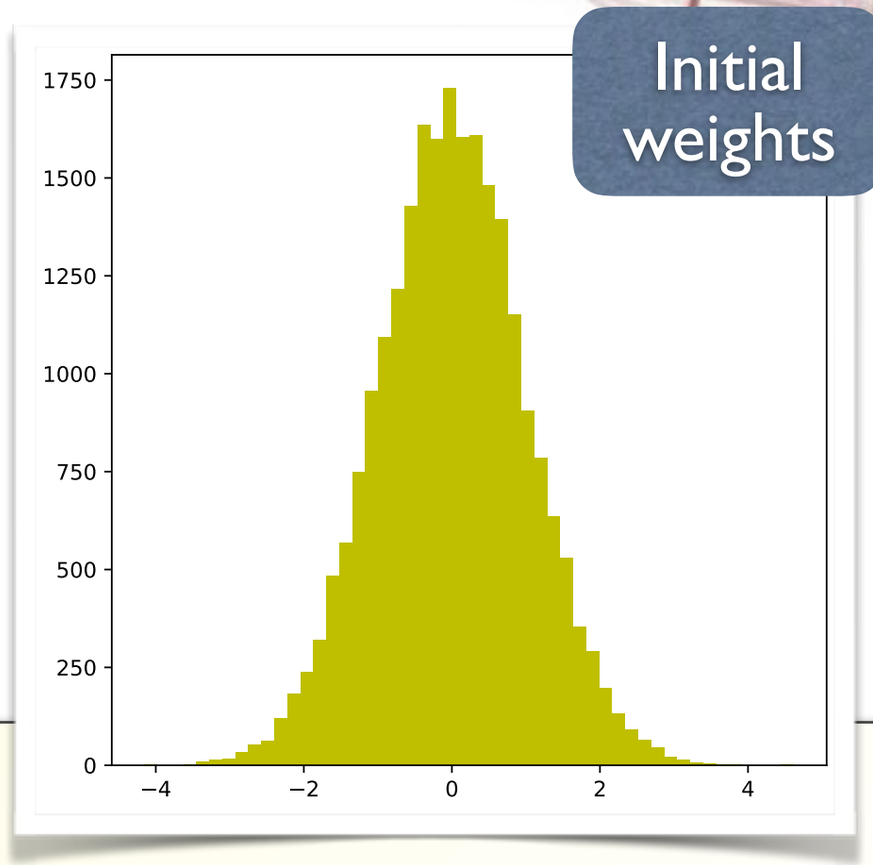
- There are many things can be consider to improve the network. Can be introduced either in the network structure, or in the training, targeting a fast learning and good performance in the testing sample:
  - The choice of network structure
  - The choice of network initialization
  - The choice of training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

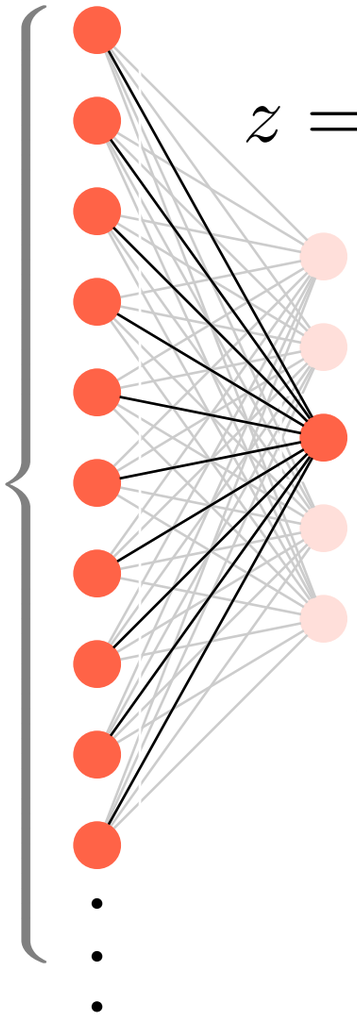



```
class neurons(object):
  def __init__(self, shape):
    self.shape = shape
. . . . . .
    self.w = [np.random.randn(n,m) for n,m in zip(shape[1:],shape[:-1])]
    self.b = [np.random.randn(n,1) for n in shape[1:]]
. . . . . .
```

neurons.py (constructor/partial)

WEIGHTS INITIALIZATION (II)

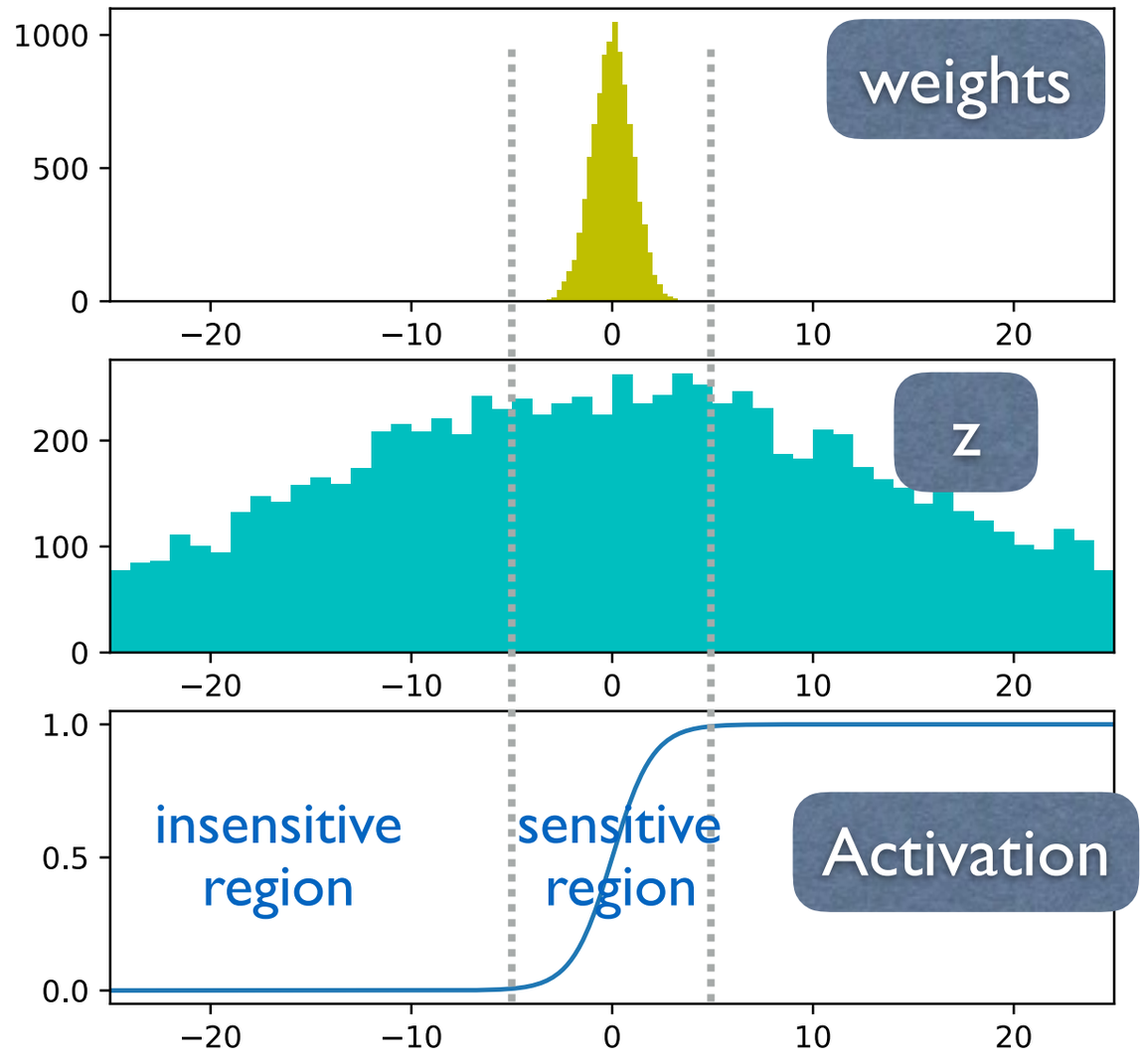
- In fact there is an issue indeed! Just consider the calculated value at one neuron at the first hidden layer:



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

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!

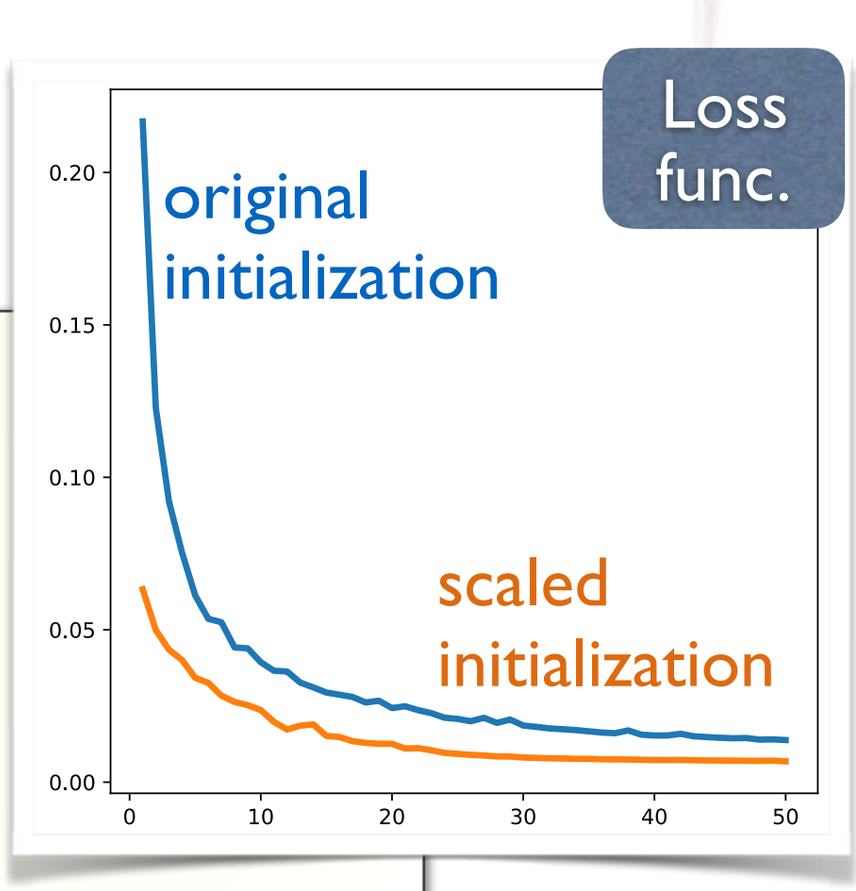


WEIGHTS INITIALIZATION (III)

- 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!

```
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)
. . . . . .
```

l302-example-05b.py (partial)



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 at the next lecture!

```
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
```

I302-example-05.py output

```
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
```

I302-example-05c.py output

BE PREPARED FOR THE NEXT LECTURE

- Instead of our own implemented neurons.py, we will adopt the widely used packages, **Keras** (with the **TensorFlow** backend).
- This will give us "ready-to-use" network models and tools, will all of the commonly used tricks implemented.
- More to read/discover at the first place:
  https://keras.io
  https://www.tensorflow.org

BE PREPARED FOR THE NEXT LECTURE (II)

- Up to now TensorFlow still requires python 3.6. If you have installed python 3.7 from anaconda and have not yet downgraded it, you may want to do it now:

```
conda install python=3.6
```

This will take a while!
Do it early!

- You can already try to install **Keras+TensorFlow** already by

```
conda install keras
```

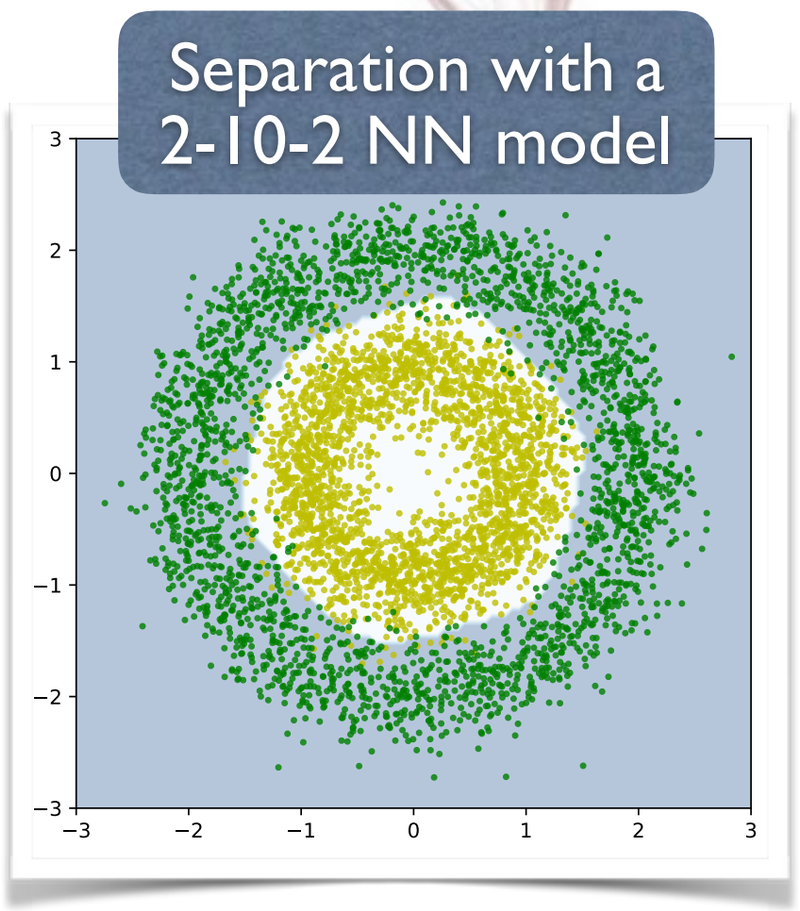
If you are not using anaconda, it can be installed though pip as well:

```
pip install keras
```

Stay tuned for the next lecture!

HANDS-ON SESSION

- **Practice 01:**
  Take `l302-example-01.py` as the template, modify the used algorithm from Gaussian SVM to the neural network. See if you can find a good separation or not!



- Tips:
  - If you want to use the `neurons.py` given above, it is required to convert the `y_train` array from simple categorizing, e.g. [0,1,0,0,1] to two target outputs, e.g. [[1,0],[0,1],[1,0],[1,0],[0,1]]
  - You may also try Keras or the scikit-learn: http://scikit-learn.org/stable/modules/neural_networks_supervised.html

HANDS-ON SESSION

```
Performance (training):
Loss: 0.xxxx, Acc: 0.yyyy
Performance (testing):
Loss: 0.xxxx, Acc: 0.yyyy
```

- **Practice 02:**
  - Trial #1:
    Enlarge the size of the hidden layer from 30 neurons to 100 neurons in the network used in `l302-example-05.py`. See how good you can reach at the end of the training? (remark: some tuning of the learning rate and # of epochs might be needed to get a good performance!)
  - Trial #2:
    Improve the network used in `l302-example-05.py` by scaling the initial weights, as introduced in `l302-example-05b.py`. See how good you can reach at the end of the training?