INTRODUCTION TO COMPUTATIONAL PHYSICS

Kai-Feng Chen
National Taiwan University

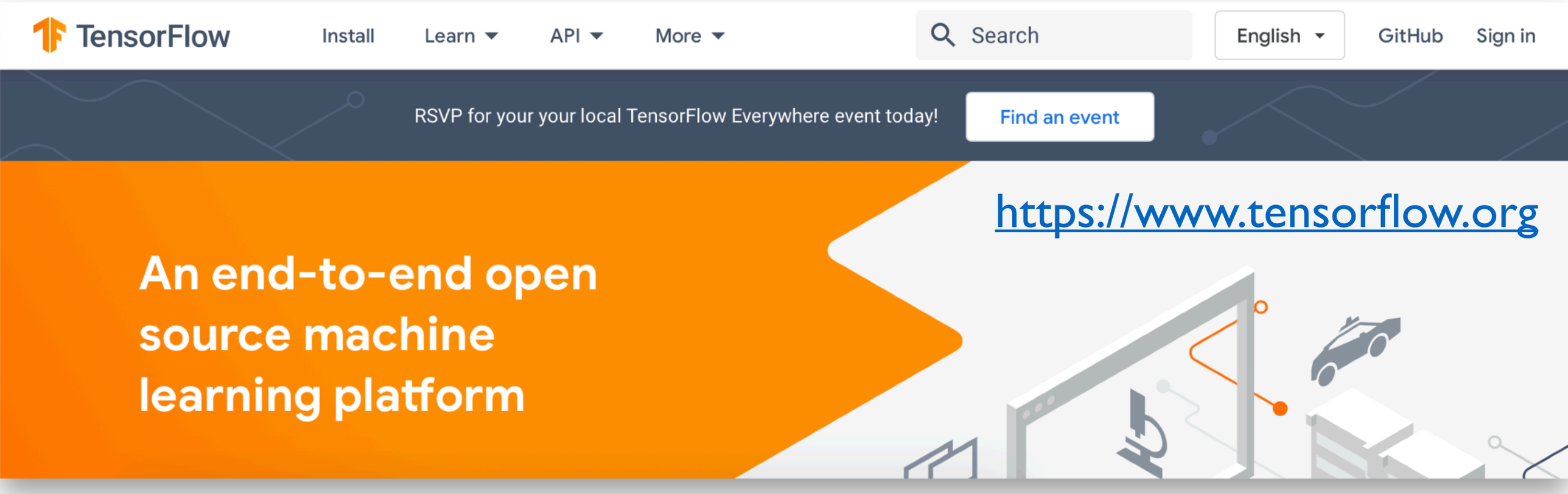
Time to tune
our network for
a better performance!

RECALL FROM THE LAST MODULES

- We have played with two nonlinear models, **SVM with non-linear kernel**, and the very classical **Neural Network**.
- Taking the MNIST data set as an benchmark, the SVM with Gaussian kernel can have a very good performance of **~98.4%** accuracy!
- Our super simple neural network can already provide a good handwriting digits recognition with an accuracy of **~95%**. With a slightly better initial weights the performance can be pushed to **~96%**. Remember this was performed by a simple model of **784-30-10** network and only 20 epochs of training so far.
- Can we do better, by considering some of the state of arts techniques? Or can we further improve it by introducing a *deeper* network structure?

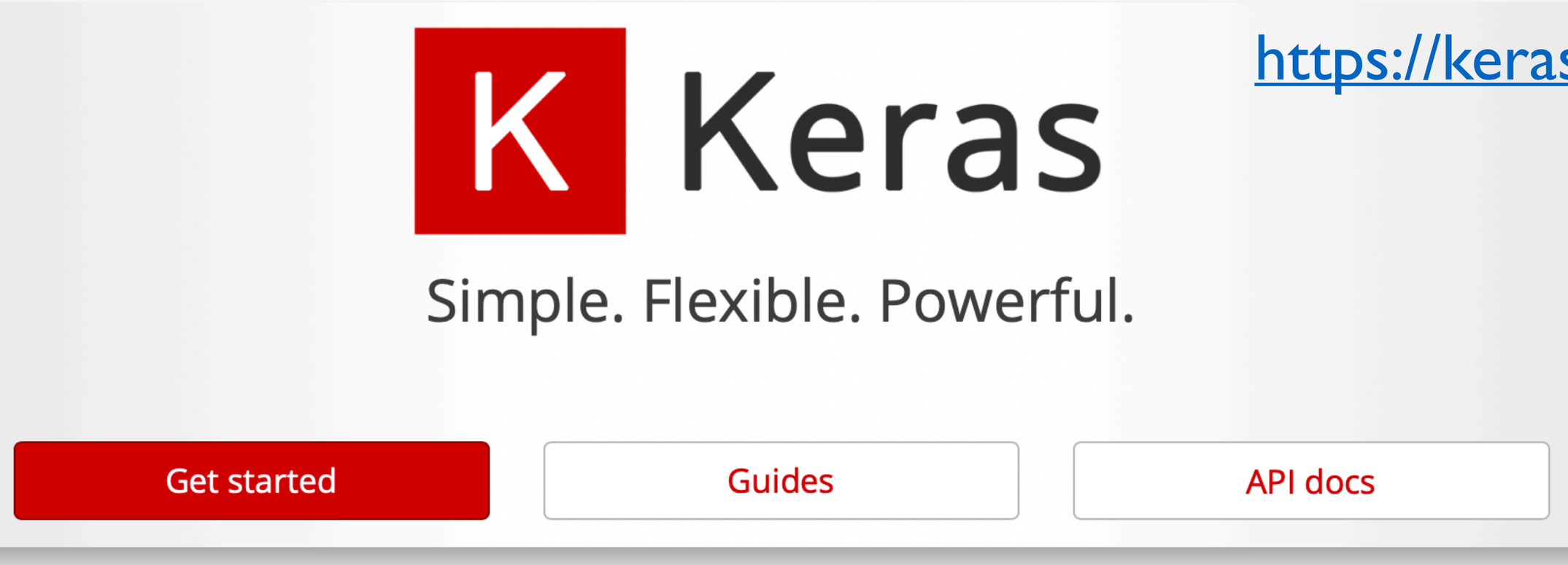
Instead of our own implemented neurons.py, we will adopt the **widely used packages** from now on!

YOU MUST HAVE HEARD THE TENSORFLOW . . .



- **TensorFlow** is an open source software library for high performance numerical computation originally developed by Google. It comes with a strong support for machine learning and deep learning model and can run on CPU/GPU, or even the TPUs.
- But in order to have an even easier live, we will use something even *simpler*!

HERE COMES THE KERAS



- **Keras** was a kind of "wrapper" or package which can help to build most of the conventional NN models, and the real calculation can be carried out by TensorFlow, as one of the supported backends.
- Now **Keras is part of TensorFlow!**

INSTALLATION OF TENORFLOW

- If you are using anaconda package, this can be done by typing this under your terminal:

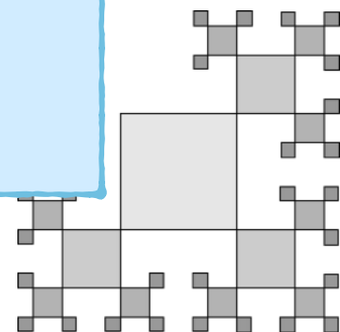
```
$ conda install tensorflow
```

- If it is working you will find **tensorflow** (and keras) being installed. If you are not using anaconda, the package can be installed though pip:

```
$ pip install tensorflow
```

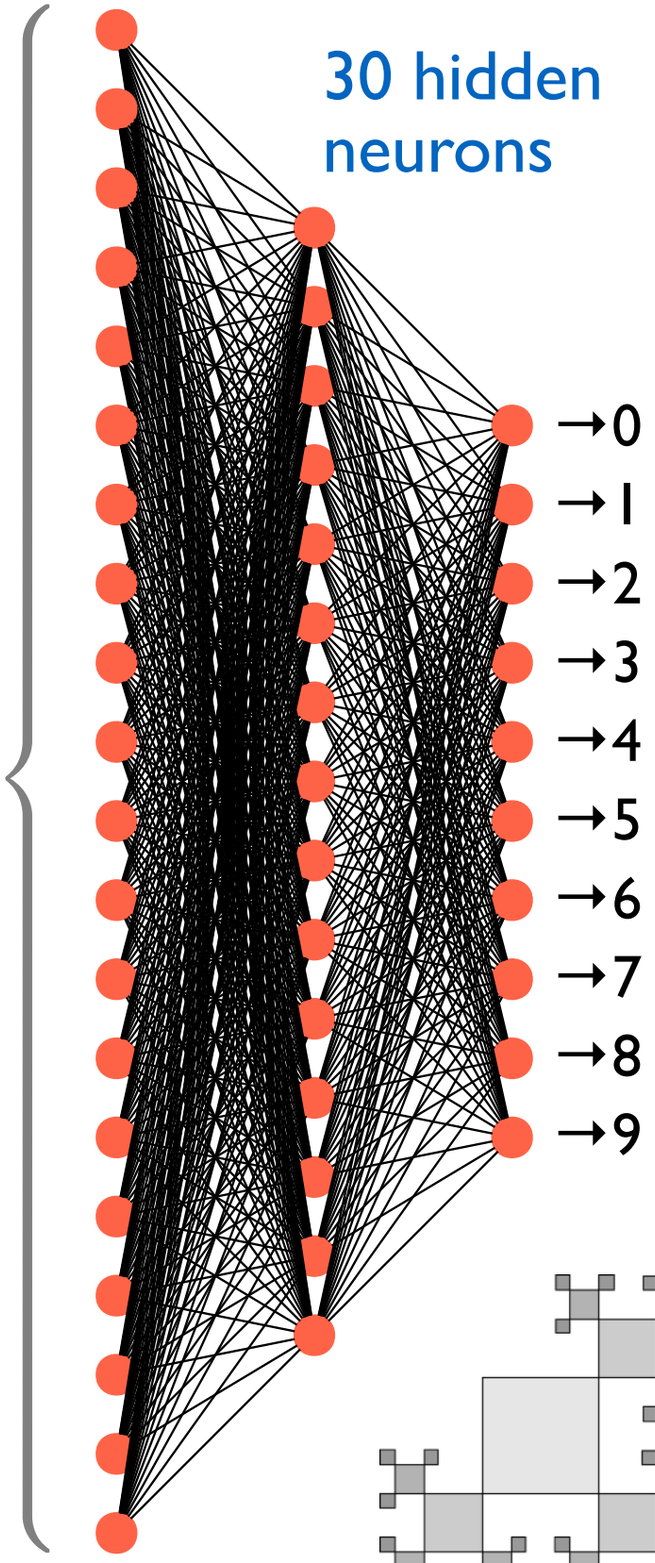
- A quick test can be made by import the tensorflow module directly:

```
$ python
Python 3.8.5 (default, Sep  4 2020, 02:22:02)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more
information.
>>> import tensorflow as tf
>>>
```



THE "BASELINE" NETWORK MODEL

- At the end of last lecture we have constructed a simple model with our own implementation. This serves as our starting point:
  - The network structure also consists with **3 layers, with one hidden layer of 30 neurons**.
  - The chosen activation function is also the **sigmoid function**.
  - The selected loss function is exactly the mean squared error, **MSE**.
  - The network will be trained using stochastic gradient descent **SGD** method.



What would be the performance if we construct exactly the same model with **Keras**?

BUILDING NETWORK WITH TENSORFLOW/KERAS

- It is more-or-less straightforward to build the network with TensorFlow / Keras:

partial **ex_ml6_1.py**

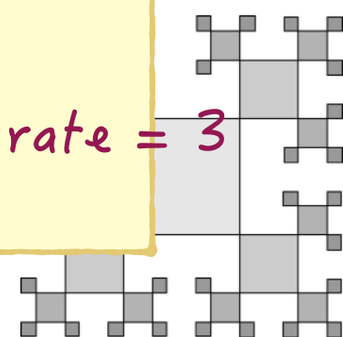
```
mnist = np.load('mnist.npz')
x_train = mnist['x_train']/255.
y_train = np.array([np.eye(10)[n] for n in mnist['y_train']])
x_test = mnist['x_test']/255.
y_test = np.array([np.eye(10)[n] for n in mnist['y_test']])

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Reshape
from tensorflow.keras.optimizers import SGD

model = Sequential()
model.add(Reshape((784,), input_shape=(28,28)))
model.add(Dense(units=30, activation='sigmoid'))
model.add(Dense(units=10, activation='sigmoid'))

model.compile(loss='mean_squared_error',
              optimizer=SGD(lr=3.0),
              metrics=['accuracy'])
```

← build a 784-30-10 model

← Loss = MSE

← training with SGD, learning rate = 3

← also output the accuracy



BUILDING NETWORK WITH TENSORFLOW/KERAS (II)

partial ex_ml6_1.py

```
. . . . . .
model.fit(x_train, y_train, epochs=20, batch_size=10)
                                    ↑ train for 20 epochs, too!
print('Performance (training)')
print('Loss: %.5f, Acc: %.5f' % tuple(model.evaluate(x_train, y_train)))
print('Performance (testing)')
print('Loss: %.5f, Acc: %.5f' % tuple(model.evaluate(x_test, y_test)))
```

```
Epoch 1/20
6000/6000 [======] - 3s 525us/step - loss: 0.0323 - accuracy: 0.8067
. . . . . .
Epoch 20/20
6000/6000 [======] - 3s 517us/step - loss: 0.0047 - accuracy: 0.9753
Performance (training)
1875/1875 [======] - 1s 448us/step - loss: 0.0045 - accuracy: 0.9759
Loss: 0.00450, Acc: 0.97587
Performance (testing)
313/313 [======] - 0s 789us/step - los
Loss: 0.00664, Acc: 0.96090
```

Already get a very similar result?
What are the **remaining wrongly tagged digits now**?

WRONGLY RECOGNIZED DIGITS?

- At the end of the training we can feed the test data into the network and see the resulting "test" performance. An accuracy of **96.1%** means we have only around **~300+ images** were wrongly tagged by our network.
- The following piece of code is prepared to show **first 100 of the wrongly tagged images**:

partial ex_ml6_1a.py

```
. . . . . .
p_test = model.predict(x_test)
failedsample = [[img,y,p] for img,y,p in
zip(mnist['x_test'],y_test,p_test) if y.argmax()!=p.argmax()]

fig = plt.figure(figsize=(10,10), dpi=80)
for i in range(len(failedsample[:100])):
    plt.subplot(10,10,i+1)
    plt.axis('off')
    plt.imshow(failedsample[i][0], cmap='Greys')
    plt.text(0.,0.,'$%d\\to%d$' % (failedsample[i][1].argmax(), \
         failedsample[i][2].argmax()),color='Red',fontsize=15)
plt.show()
```

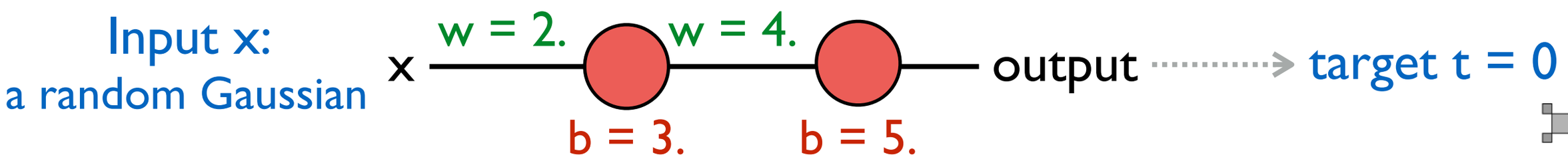
↑ pick up those wrongly tagged samples

WRONGLY RECOGNIZED DIGITS? (II)

- You can see that there are still some handwriting digits are obviously wrongly tagged.
- But there are also some images can be easily mis-tagged!
- Nevertheless this is our starting point and we are going to discuss several techniques to improve the network!

SLOW LEARNING WITH BAD WEIGHTS

- Based on the NN model up to now, one of the typical issue we may face is this: when the initial weights are very far from the optimal, the learning is actually slower.
- This is very different from our intuition in fact — usually human beings **learn faster if they are very wrong**. But this is not the case for the NN…
- A demonstration simple network with only one input layer and one output layer, with 2 weights and 2 bias. Let's set the weights/bias by hand to some particular values:



SLOW LEARNING WITH BAD WEIGHTS (II)

- Such a model can be built with Keras easily as well:

partial ex_ml6_2.py

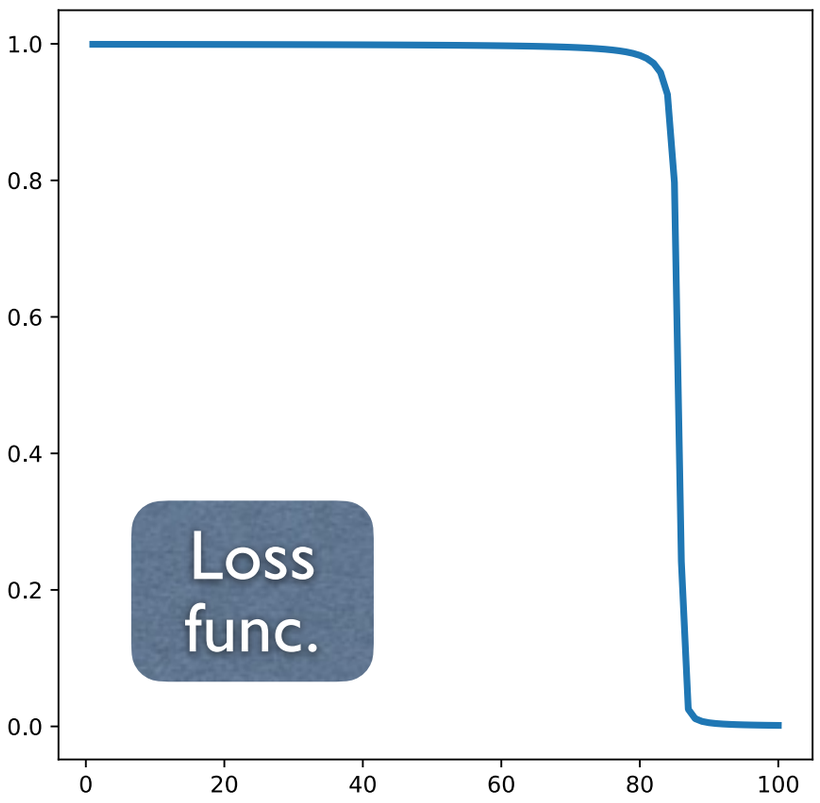
```
x_train = np.random.randn(1000)
y_train = np.zeros(1000)

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.optimizers import SGD

model = Sequential()
model.add(Dense(units=1, activation='sigmoid', input_dim=1))
model.add(Dense(units=1, activation='sigmoid'))
model.compile(loss='mean_squared_error',
              optimizer=SGD(lr=1.0))

model.layers[0].set_weights([np.array([[2.]]),np.array([3.])])
model.layers[1].set_weights([np.array([[4.]]),np.array([5.])])

rec = model.fit(x_train, y_train, epochs=100, batch_size=100)

vep = np.linspace(1.,100.,100)
fig = plt.figure(figsize=(6,6), dpi=80)
plt.plot(vep,rec.history['loss'], lw=3)
plt.show()
```

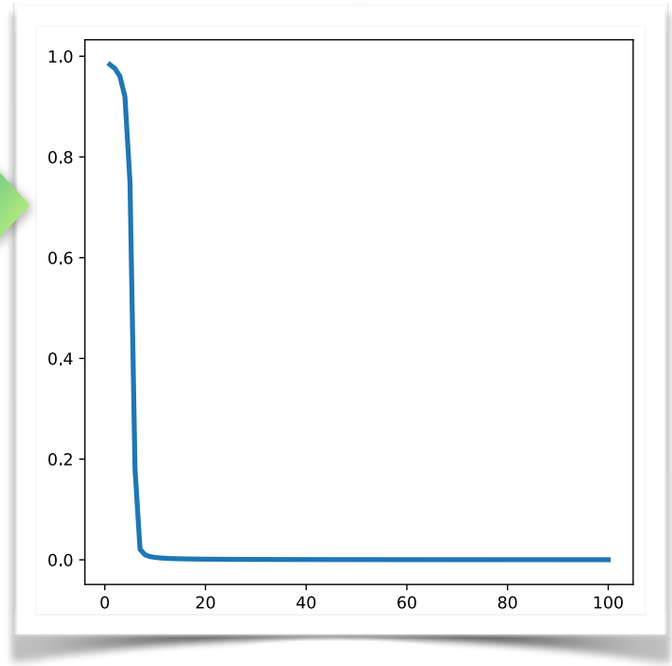
A simple sequential model with
↓ only 1 input / 1 output neuron

↑ keep the history of training

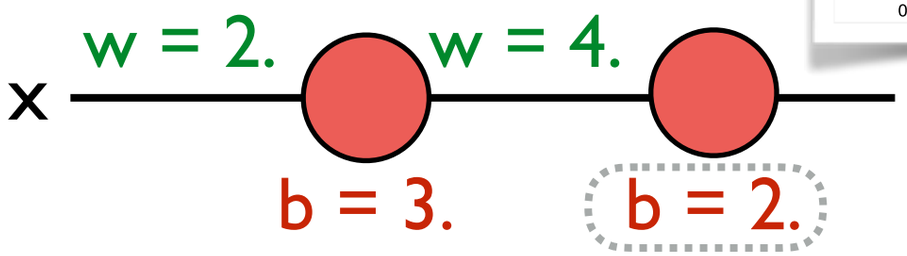
SLOW LEARNING WITH BAD WEIGHTS (III)

- This is what you may find: the loss function is large for initial epochs — and it takes for a while until the training really starts.
- Remember this network is already very simple with only 4 parameters to be tuned. But such a situation does happen.





Surely the situation is better if one uses a different initial values, e.g.



Both this also hints a problem of our network!

THE CHOICE OF LOSS

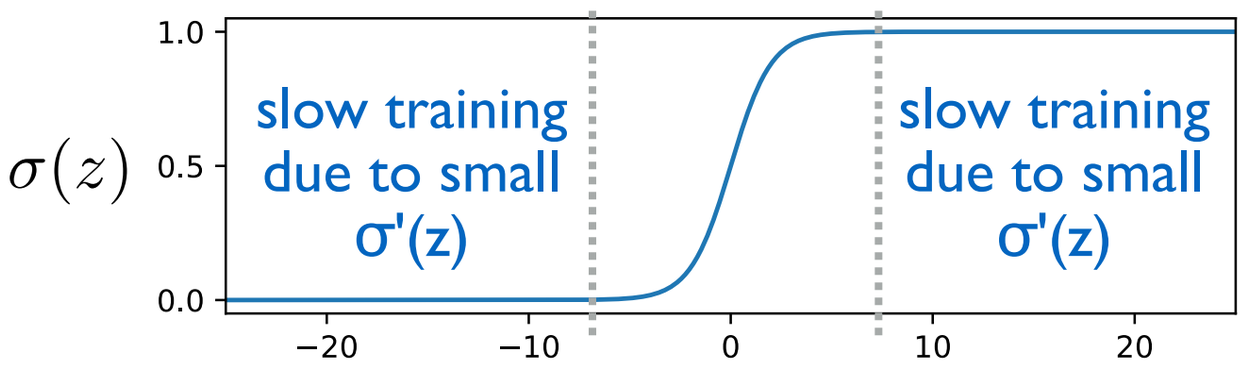
- In fact such a situation can be related to the definition of the loss function, and its gradient w.r.t. the weights and bias.
- Consider the current choice of loss, the **mean squared error**:

Consider only one output: $$L(w,b) = \frac{1}{2}|\sigma(z) - t|^2$$

Gradient is required in the training process:

$$\frac{\partial L}{\partial w} = [\sigma(z) - t]\sigma'(z)\frac{\partial z}{\partial w}$$

$$\frac{\partial L}{\partial b} = [\sigma(z) - t]\sigma'(z)\frac{\partial z}{\partial b}$$



The training speed is proportional to the **first derivative of the activation function**! If the z value is too large or too small, the training will be very slow.

THE CHOICE OF LOSS (II)

- This can be improved by introducing a different loss function, for example, the **(binary) cross-entropy function**:

$$Loss(w_i, b_j) = \frac{-1}{n} \sum_{x}^{n} [t \ln y + (1-t) \ln(1-y)]$$

Consider only one output & replace y by σ(z):

$$L = -[t \ln \sigma(z) + (1-t) \ln(1-\sigma(z))]$$

Gradient w.r.t. weights/bias:

$$\frac{\partial L}{\partial w} = -t \frac{\sigma'(z)}{\sigma(z)} \frac{\partial z}{\partial w} + (1-t) \frac{\sigma'(z)}{1-\sigma(z)} \frac{\partial z}{\partial w}$$

$$= [\sigma(z) - t] \left\{ \frac{\sigma'(z)}{\sigma(z)[1-\sigma(z)]} \right\} \frac{\partial z}{\partial w}$$

Cancelled

$$= [\sigma(z) - t] \frac{\partial z}{\partial w}$$

$$\sigma(z) = (1 + e^{-z})^{-1}$$

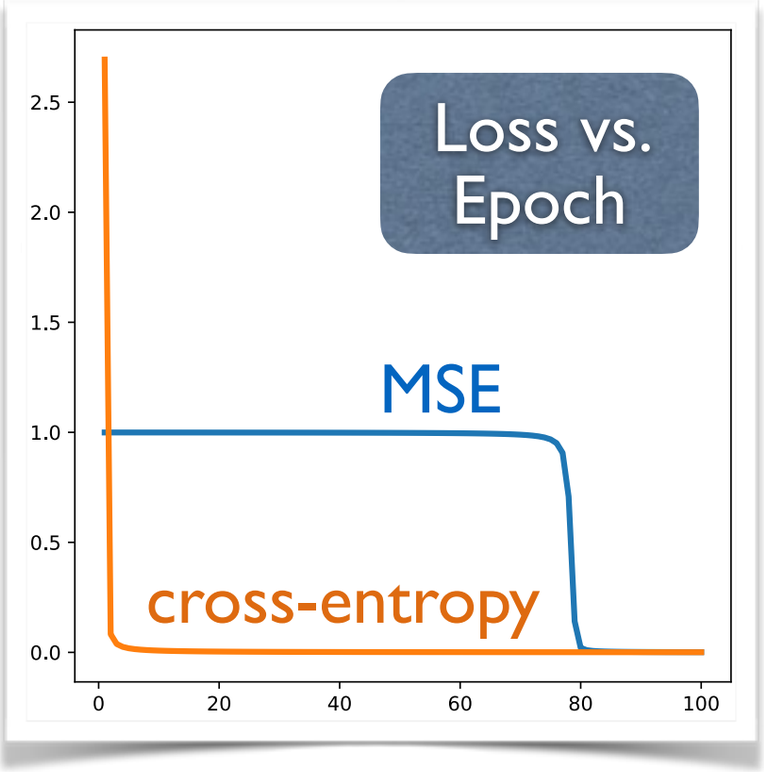
$$\Rightarrow \sigma'(z) = \sigma(z)[1 - \sigma(z)]$$

A little bit of calculous…

Not depending on the first derivative $\boldsymbol{\sigma'(z)}$ anymore!

THE CHOICE OF LOSS (III)

- This effect can be tried easily!
- Indeed the cross-entropy function can speed up the learning even with bad initial weights!




```
model.compile(loss='mean_squared_error', optimizer=SGD(lr=1.0))
model.layers[0].set_weights([np.array([[2.]]),np.array([3.])])
model.layers[1].set_weights([np.array([[4.]]),np.array([5.])])

rec1 = model.fit(x_train, y_train, epochs=100, batch_size=100)

model.compile(loss='binary_crossentropy', optimizer=SGD(lr=1.0))
model.layers[0].set_weights([np.array([[2.]]),np.array([3.])])
model.layers[1].set_weights([np.array([[4.]]),np.array([5.])])

rec2 = model.fit(x_train, y_train, epochs=100, batch_size=100)

vep = np.linspace(1.,100.,100)
fig = plt.figure(figsize=(6,6), dpi=80)
plt.plot(vep,rec1.history['loss'], lw=3)
plt.plot(vep,rec2.history['loss'], lw=3)
plt.show()
```

partial ex_ml6_2a.py

THE CHOICE OF OUTPUT LAYER

- Another approach to the same problem is by introducing the **softmax layer**, instead of the classical sigmoid function.
- The softmax layer is a different type of output layer, it can be expressed as

$$y_j = \frac{\exp(z_j)}{\sum_k \exp(z_k)}$$

k: classes

- The output of the network y is replaced by the formula above. Given it is normalized (summing all of the outputs will be one by definition), another benefit of softmax layer is that the output values can be treated as a probability, which is not the case for the classical sigmoid function.
- By combining this with the **cross-entropy function**, it can be another remedy to the slow learning problem.

SOFTMAX + CROSS-ENTROPY LOSS

- The **(Categorical) cross-entropy loss function** for a given training sample is

$$L = -\sum_j t_j \ln(y_j) \quad j\text{: classes}$$

- You may find this is just an extended version of the previous cross-entropy function which was derived for 2 classes (binary case).

- Let's first calculate the partial derivate for y_j w.r.t. z_i (remember z_i is linear sum of weights times the outputs from previous layer + bias):

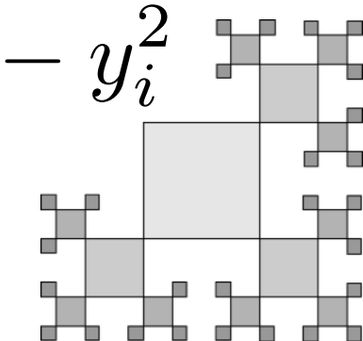
$$y_j = \frac{e^{z_j}}{\sum e^{z_k}}$$

If j≠i:

$$\frac{\partial y_j}{\partial z_i} = \frac{-e^{z_j} e^{z_i}}{(\sum e^{z_k})^2} = -y_i y_j$$

If j=i:

$$\frac{\partial y_i}{\partial z_i} = \frac{e^{z_i}(\sum e^{z_k}) - e^{z_i} e^{z_i}}{(\sum e^{z_k})^2} = y_i - y_i^2$$



SOFTMAX + CROSS-ENTROPY LOSS (II)

- Then the derivative for the loss function itself:

$$L = -\sum_j t_j \ln(y_j) \Rightarrow \frac{\partial L}{\partial z_i} = -\sum_j t_j \frac{1}{y_j} \frac{\partial y_j}{\partial z_i}$$

$$\frac{\partial L}{\partial z_i} = -t_i \frac{1}{y_i} \frac{\partial y_i}{\partial z_i} - \sum_{j \neq i} t_j \frac{1}{y_j} \frac{\partial y_j}{\partial z_i}$$

$$= -t_i(1 - y_i) + \sum_{j \neq i} t_j y_i = -t_i + t_i y_i + \sum_{j \neq i} t_j y_i$$

$$= -t_i + y_i \left(t_i + \sum_{j \neq i} t_j \right) = y_i - t_i$$

It should solve the slow learning problem as well!

t_j = target value for class j
by definition $\Sigma t_j = 1$

It ends up with the same results as before and no dependency on $\boldsymbol{\sigma'(z)}$!

LET'S TRY IT OUT!

partial ex_ml6_3.py

```
model = Sequential()
model.add(Reshape((784,), input_shape=(28,28)))
model.add(Dense(30, activation='sigmoid'))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer=SGD(lr=1.0),
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=20, batch_size=30)
```

← it's a two-line modification!

```
Epoch 1/20
2000/2000 [======] - 1s 575us/step - loss: 0.4632 - accuracy: 0.8614
. . . . . .
Epoch 20/20
2000/2000 [======] - 1s 569us/step - loss: 0.0438 - accuracy: 0.9866
Performance (training)
1875/1875 [======] - 1s 490us/step - loss: 0.0395 - accuracy: 0.9882
Loss: 0.03947, Acc: 0.98823
Performance (testing)
313/313 [======] - 0s 899us/step - lo
Loss: 0.13726, Acc: 0.96040
```

Although the performance for training sample is improved, but not for testing!

Typical overtraining issue — to be discussed in the next module!

COMMENTS

- We have two possible treatments that can be used in the classification problem:
  - **sigmoid activation + binary cross-entropy loss**
  - **softmax layer + categorical cross-entropy loss**
- You may find they have a very similar formulation and similar behavior. This is due to the fact that sigmoid is special case of softmax function (if you compare them carefully), and the binary cross-entropy loss can be considered as a "yes/no" problem for each output neuron.
- In our handwriting digits example one can solve **"10 binary problems"** with the binary cross-entropy loss, or **"one out of 10 choices"** with categorical cross-entropy loss.

Remark: Keras may report a different accuracy value if you do sigmoid activation + binary cross-entropy loss!

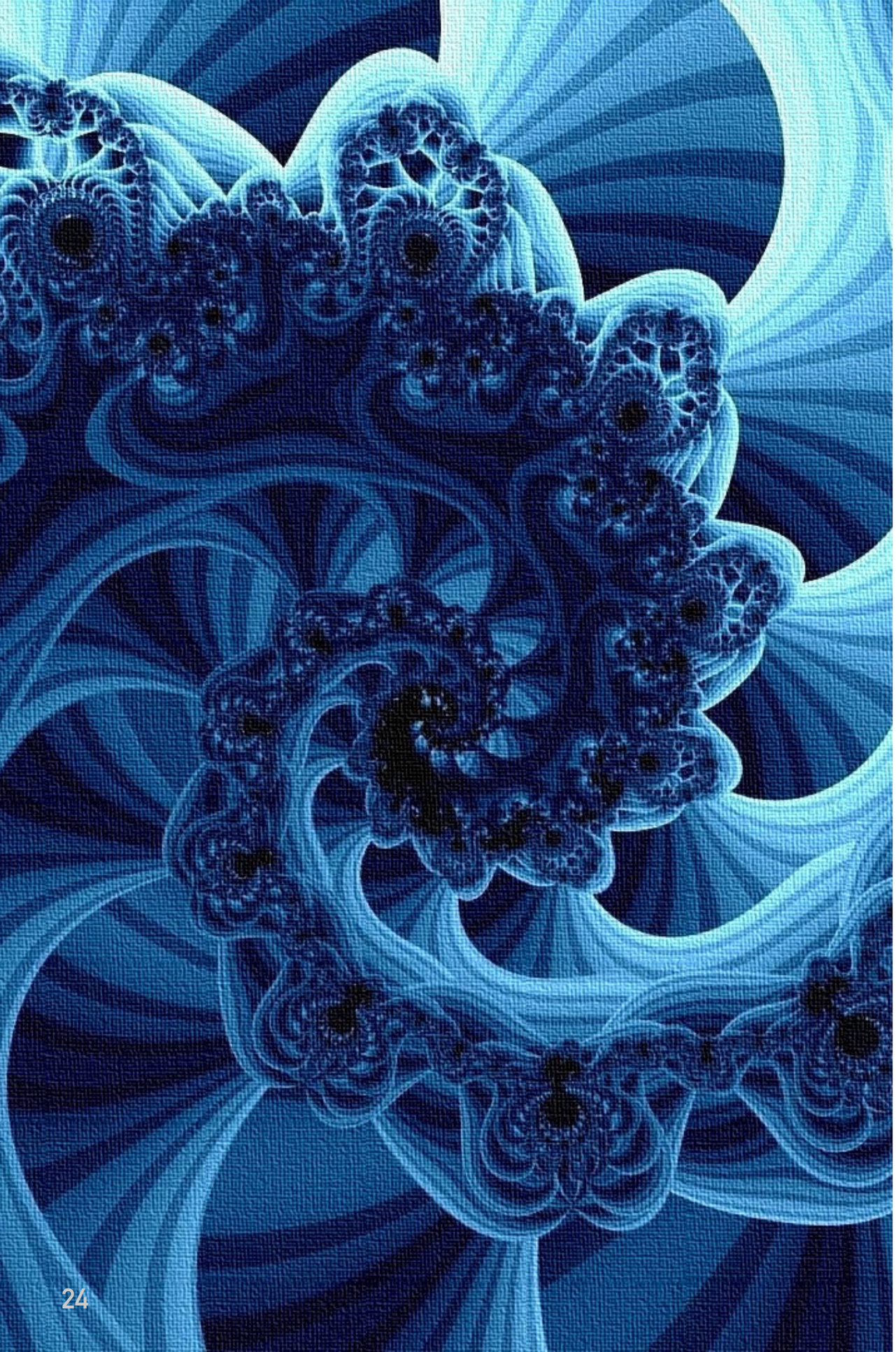
JUST TRY IT OUT!

- When introducing softmax layer + categorical cross-entropy loss, the output softmax function is :

$$y_j = \frac{\exp(z_j)}{\sum_k \exp(z_k)}$$

  which can be considered as **"probability"** out of the multiple choice, and this is one of the interesting aspect of the softmax layer.

- Try to extract output value of the best and second best options from the remaining wrongly tagged digits, see if their values are not too far from each other (ie. the second option has fairly good chance to be correct), given the probability interpretation of the softmax layer?



MODULE SUMMARY

- In this module we first introduce how to use the TensorFlow/Keras package to construct the NN, and started to discuss how to improve the performance based on the knowledge of loss function.
- Next module we will continue to discuss how to improve the performance further with few other known tricks!