INTRODUCTION TO COMPUTATIONAL PHYSICS

Kai-Feng Chen
National Taiwan University

LARGER/DEEPER NETWORK?

- Now finally — can we improve our network with more hidden neurons and/or more hidden layers?
- Let's first integrated several improvements discussed up to now: Full training sample + **ReLu** activation + **Adam** optimizer + **Dropout** + a much larger network of **2 hidden layers of 512 neurons**:

partial ex_ml8_1.py

```
. . . . . .
model = Sequential()
model.add(Reshape((28*28,), input_shape=(28,28)))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=40, batch_size=120)
. . . . . .
```

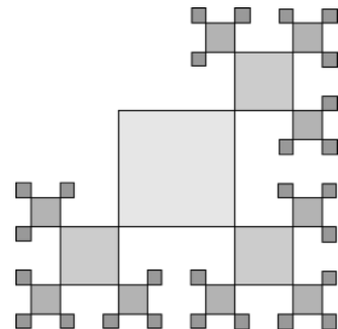
This network has **669,706** parameters to be tuned.

LARGER/DEEPER NETWORK? (II)

- We can have a great performance of 98.4% accuracy which matches to our performance from SVM with Gaussian kernel!

```
Epoch 40/40
500/500 [======] - 2s 4ms/step - loss: 0.0105 - accuracy: 0.9969
Performance (training)
Loss: 0.00247, Acc: 0.99923
Performance (testing)
Loss: 0.11009, Acc: 0.98380
```

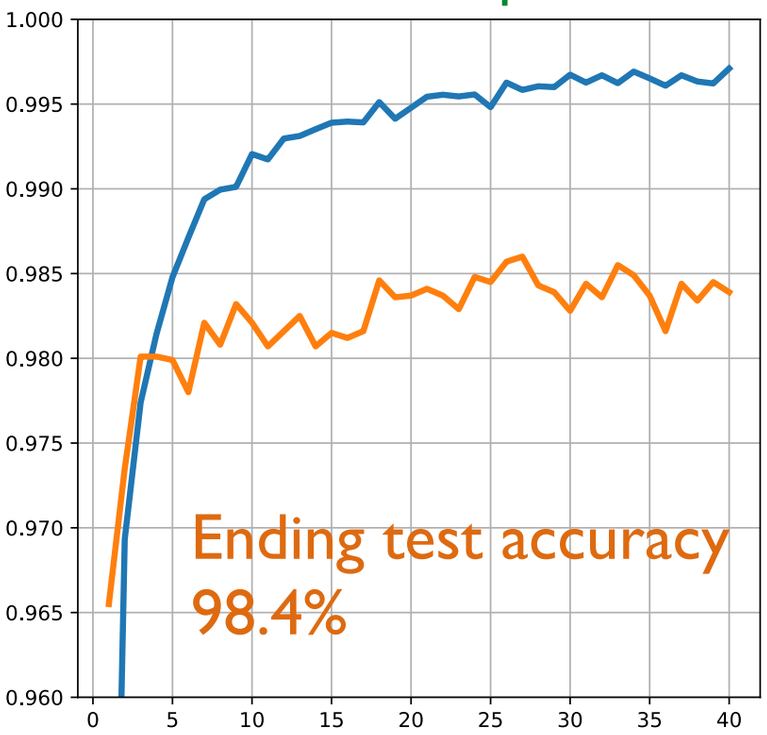
- **Can we do *even better* with a deeper network?** e.g. adding more layers?
- In fact it is not so obvious. A larger network will definitely have much more parameters to be optimized and have a stronger capability to describe the data, but it is definitely much more difficult to train. In particular, **a deeper network will be even harder**.



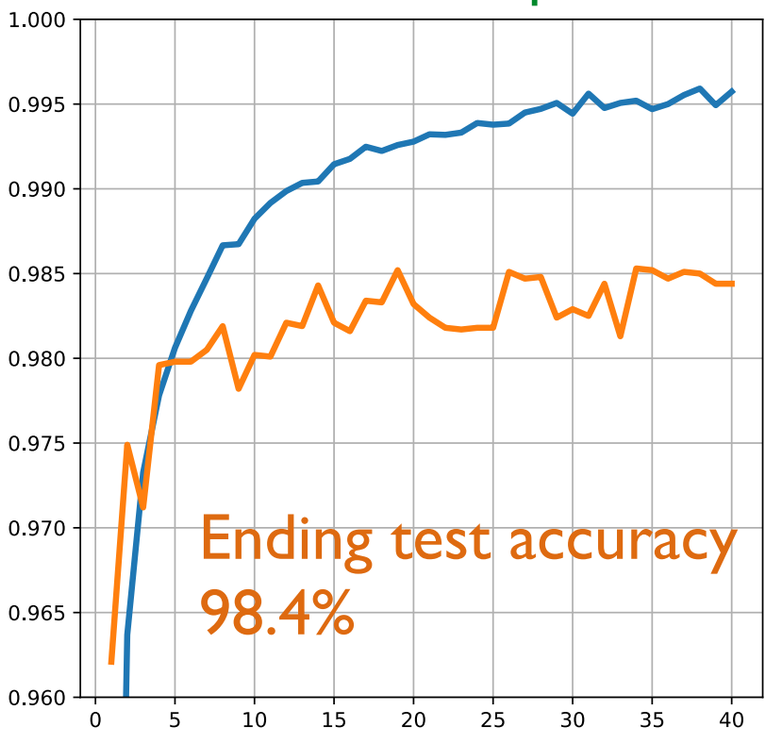
LARGER/DEEPER NETWORK? (III)

- Let's try several different network models and see if we can have interesting findings?

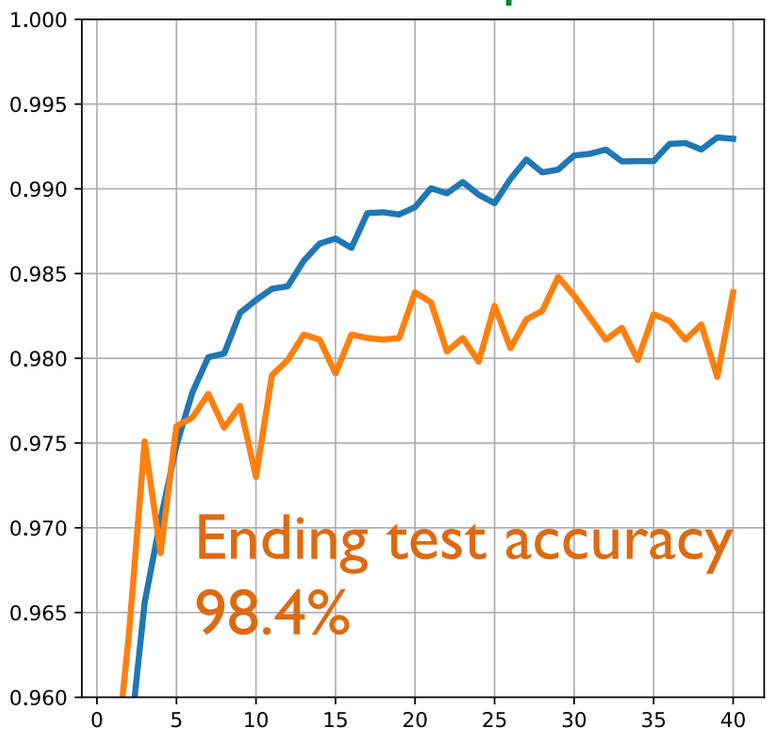
784-(512x2)-10
train for 40 epochs



784-(512x4)-10
train for 40 epochs



784-(512x8)-10
train for 40 epochs

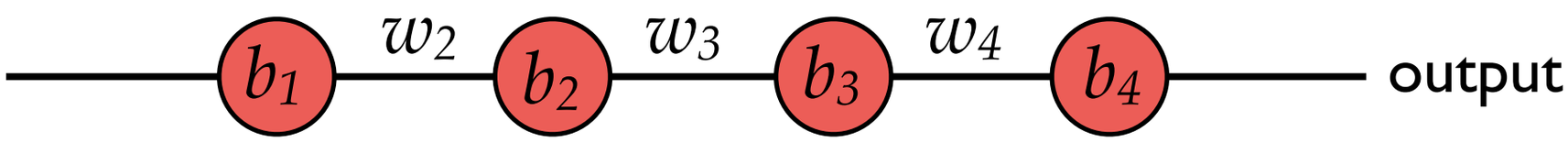


A more complex network does require a longer training time.
Why it is difficult to train a deeper network?
Do we expecte to encounter any issue?

WHY IT IS DIFFICULT TO TRAIN A DEEP NETWORK?

- Surely a deeper network does contain much more weights/bias to be tuned. But this is not the only reason — **vanishing gradients with a deeper network**. Small gradients = slow learning.
- Let's consider a chain of neurons and calculate the gradient according to back propagation:

$$\frac{\partial L}{\partial b_4} = \sigma'(z_4) \cdot \frac{\partial L}{\partial y}$$

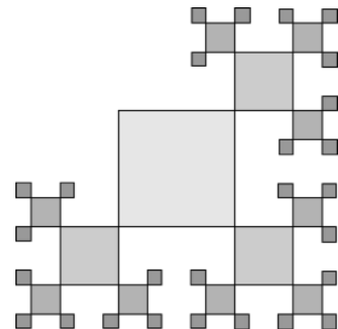


$$\frac{\partial L}{\partial b_1} = \sigma'(z_1) \cdot w_2 \cdot \sigma'(z_2) \cdot w_3 \cdot \sigma'(z_3) \cdot w_4 \cdot \sigma'(z_4) \cdot \frac{\partial L}{\partial y}$$

Generally the weights are small (<1) after training, and $\sigma'(z)$ is less then 0.25 by definition, if the sigmoid function is used. This will enforce

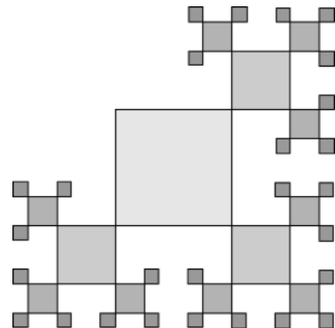
$$\frac{\partial L}{\partial b_1} < 0.0156 \frac{\partial L}{\partial b_4}$$

The updating on b_1 will be much slower than b_4.



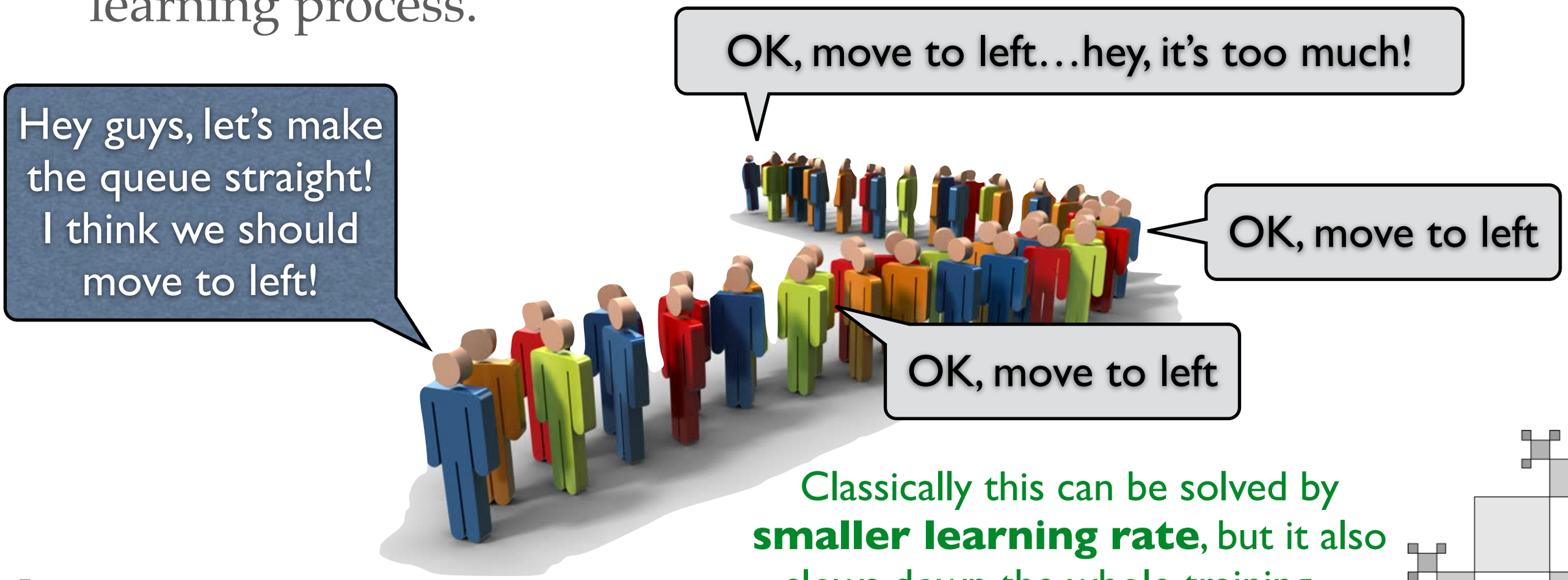
WHY IT IS DIFFICULT TO TRAIN A DEEP NETWORK? (II)

- And this is not the full story. The small $\sigma'(z)$ is not a problem for ReLU activation. However, if we have large weights, say $>> 1$, the gradient will become very large when network goes deeper. Then we are going to have an **exploding gradient problem** instead.
- The intrinsic problem is that the **gradients are unstable with deeper network**, given they are evaluated with a production of many layers of weights and derivatives.
- In fact such unstable gradient problem is a *complex issue* and depending on many other factors (and hyperparameters) as well. Although it sounds difficult to have a decent deep network trained, but if one can, it should still get a better performing network model.



WHY IT IS DIFFICULT TO TRAIN A DEEP NETWORK? (III)

- **Internal covariate shift** is another kind of problem which may shows up when training a deeper network layer. This is due to the distributions of the activations are always changing during training, and the training of the intermediate layers may not be able to catch up the situation and then slows down the learning process.



Classically this can be solved by **smaller learning rate**, but it also slows down the whole training…

ANY TRICK THAT WE CAN STILL GIVE IT A TRY?

- The **Batch Normalization** is a method to mitigate these issues.
- The key idea is to normalize the inputs of each layer, try to make it closer to a standard Gaussian (normal) distribution.
- First calculate the mean and variance of the input:

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} x_i$$

$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$$

- Normalize the inputs using the previously calculated batch statistics:

$$\overline{x_i} = \frac{x_i - \mu_B}{\sigma_B}$$

- Then shift/scale the input:

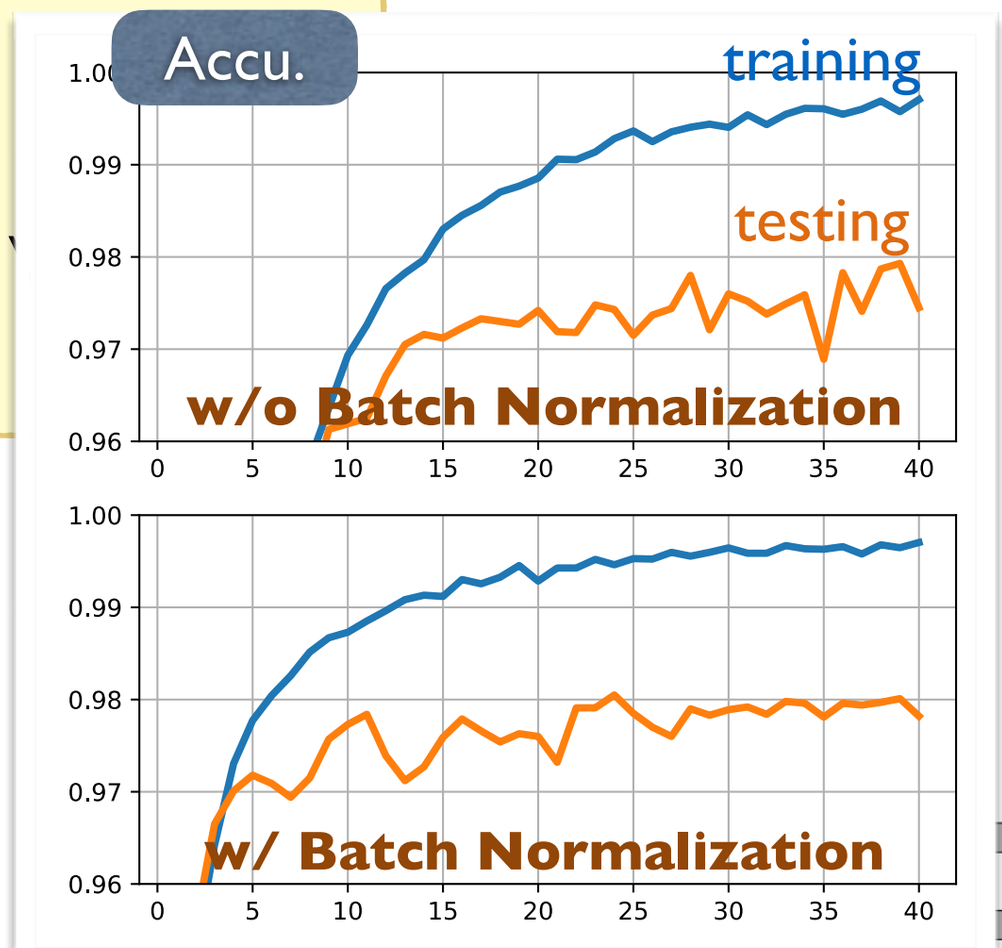
$$y_i = \gamma\overline{x_i} + \beta$$

The **γ**,**β** are trained together with other parameters of the network.

ANY TRICK THAT WE CAN STILL GIVE IT A TRY? (II)

partial ex_ml8_2.py

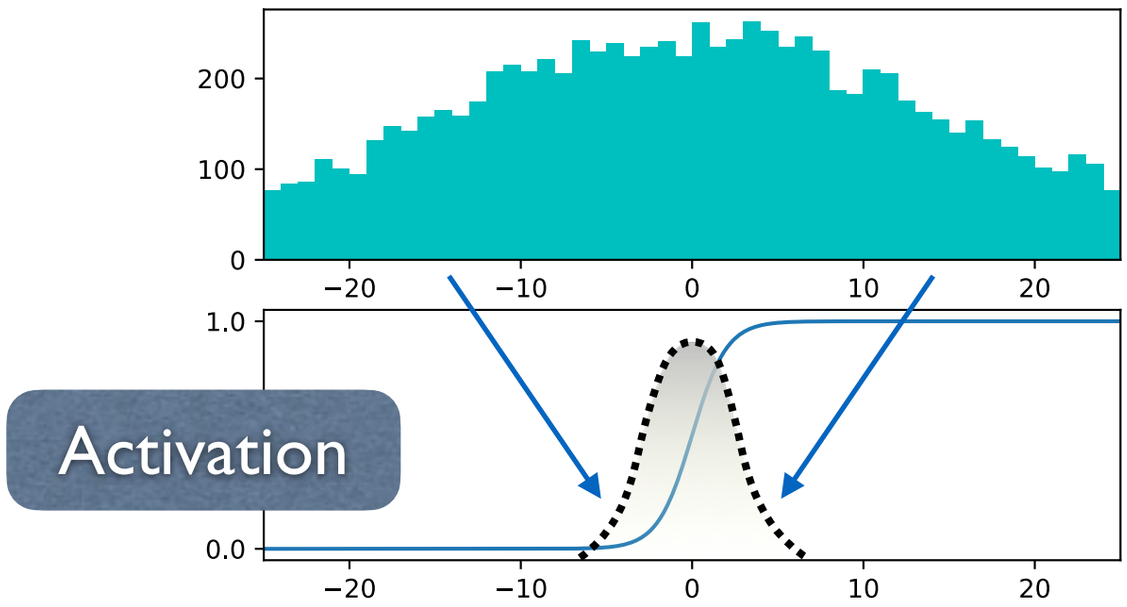
```
. . . . . .
from tensorflow.keras.layers import BatchNormalization
. . . . . .
m2 = Sequential()
m2.add(Reshape((784,), input_shape=(28,28)))
m2.add(BatchNormalization())
for layer in range(8):
    m2.add(Dense(256, activation='sigmoid'))
    m2.add(BatchNormalization())
m2.add(Dense(10, activation='softmax'))
m2.compile(loss='categorical_crossentropy',
    optimizer=Adam(), metrics=['accuracy'])
. . . . . .
```



- Let's test this with a larger / deeper network like **784-(256×8)-10** and with 40 epochs of training, and with the **sigmoid** activation:

ANY TRICK THAT WE CAN STILL GIVE IT A TRY? (III)

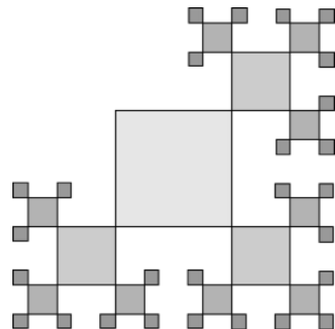
- You may already observed the problem of bad initial weights (as we already discussed in the earlier module) can be reduced, too:



Batch normalization can reduce the **gradient vanishing problem**, and **internal covariate shift problem**; mild improvement on the overtraining.

- Generally this method may work for larger/deeper network with the **sigmoid** or **tanh** activations (*not so useful for ReLU…*). Usually it is a bad idea to mix it with dropout, too, since the dropout may interfere with the normalization calculation.

- On the other hand, we already learned that a **network with different structure** can further improve the performance!



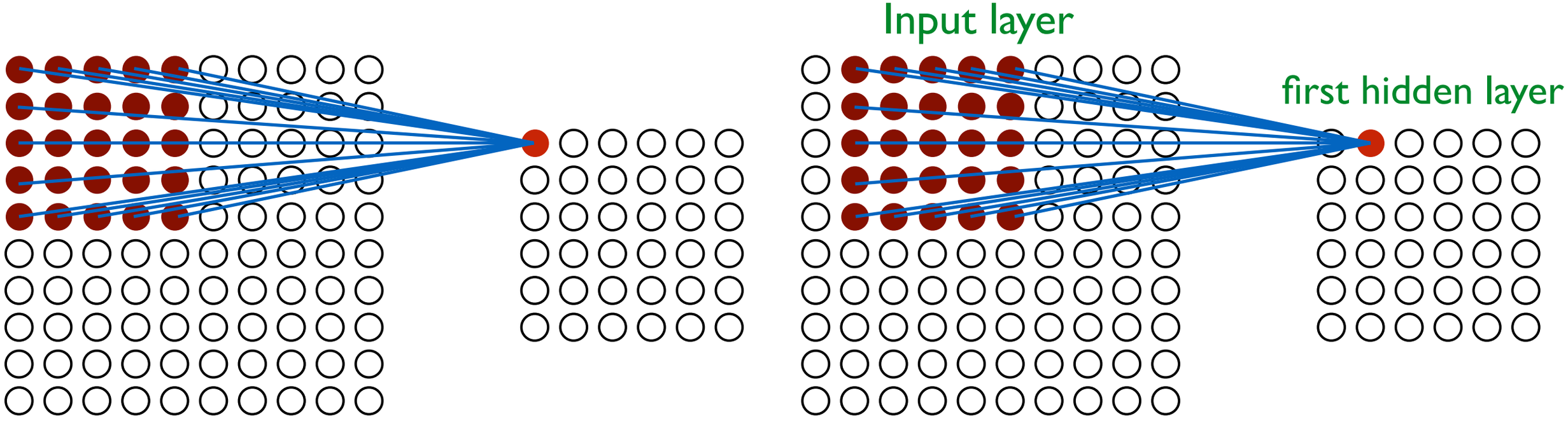
Here comes the Convolutional Neural Network…

THE CONVOLUTIONAL NEURAL NETWORK

- Up to now we are using a network first by "reshape" of the input **28×28 pixels into a flat input of 784** neurons. Although it works rather well but we do not take into account the nature of images in fact. **The local information (*of adjacent pixels*) is lost**.
- The convolutional networks use a special architecture which is particularly well-adapted to image recognition. The architecture of convolutional network makes the training of deep, multi-layer networks easier.
- There are several ideas introduced for the convolutional neural networks to be discussed in the following slides: **local receptive fields**, **shared weights**, and the **pooling**.

LOCAL RECEPTIVE FIELDS

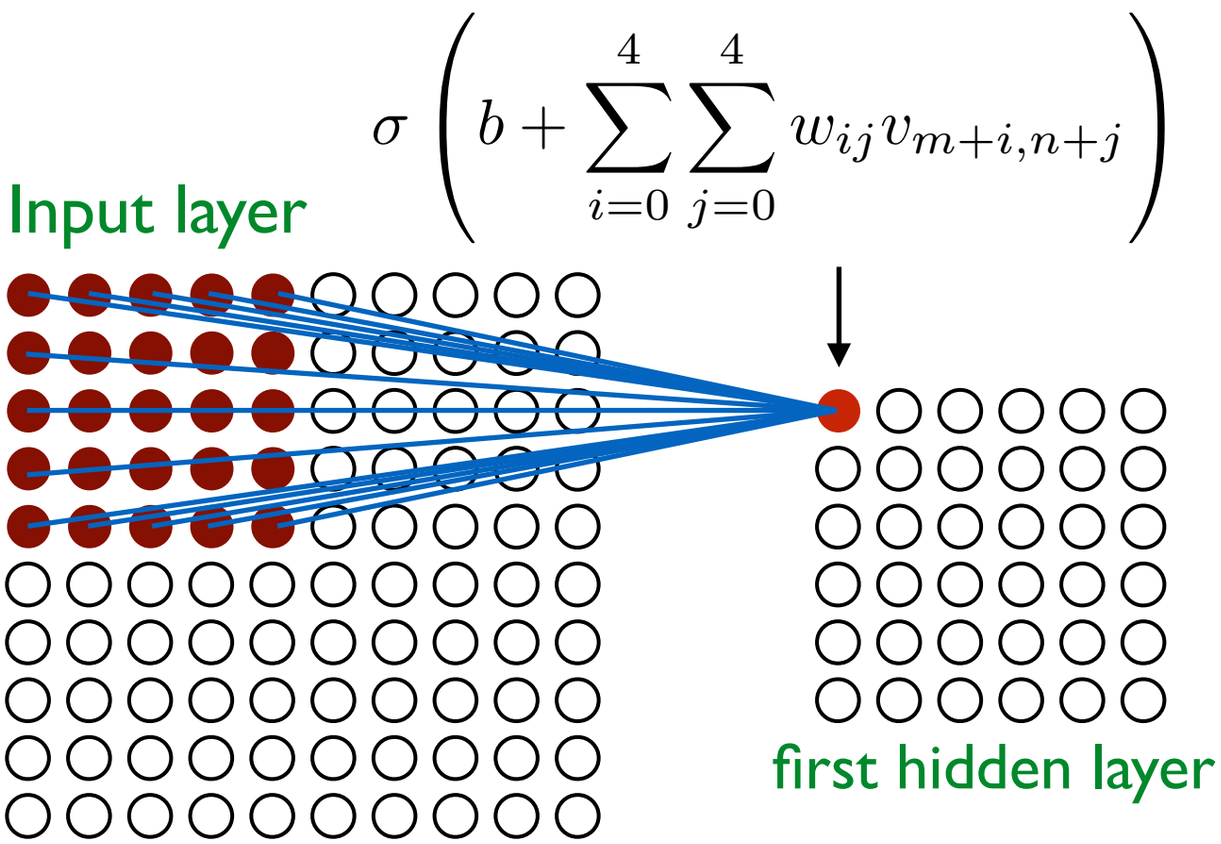
- In a typical convolutional network, the input layer is encoded in the following structure. For example, instead of fully connected network, one only has the first 5×5 block of neurons being connected to one neuron in the first hidden layer, and next 5×5 block connected to the second neuron…



If we have 28×28 as the input image, and with a 5×5 local representative field, the first hidden layer will be 24×24.

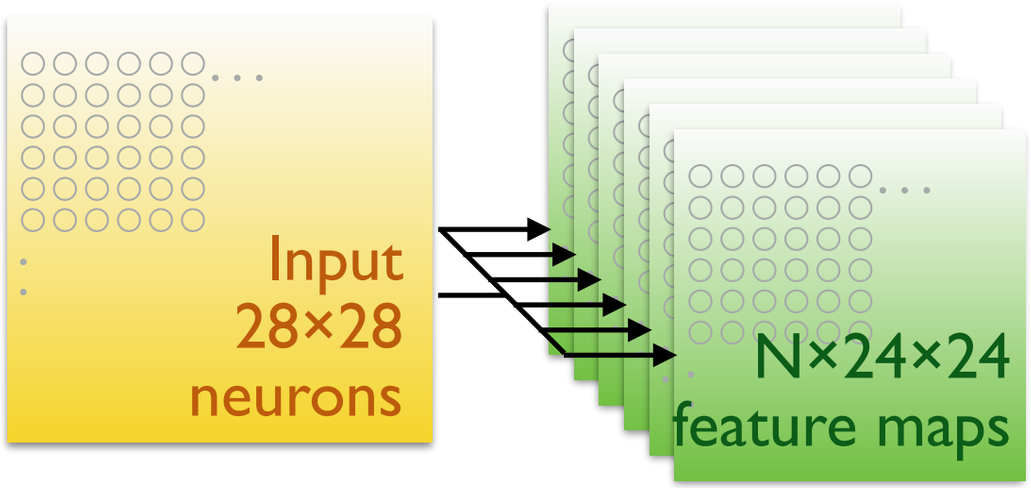
SHARED WEIGHTS/BIAS

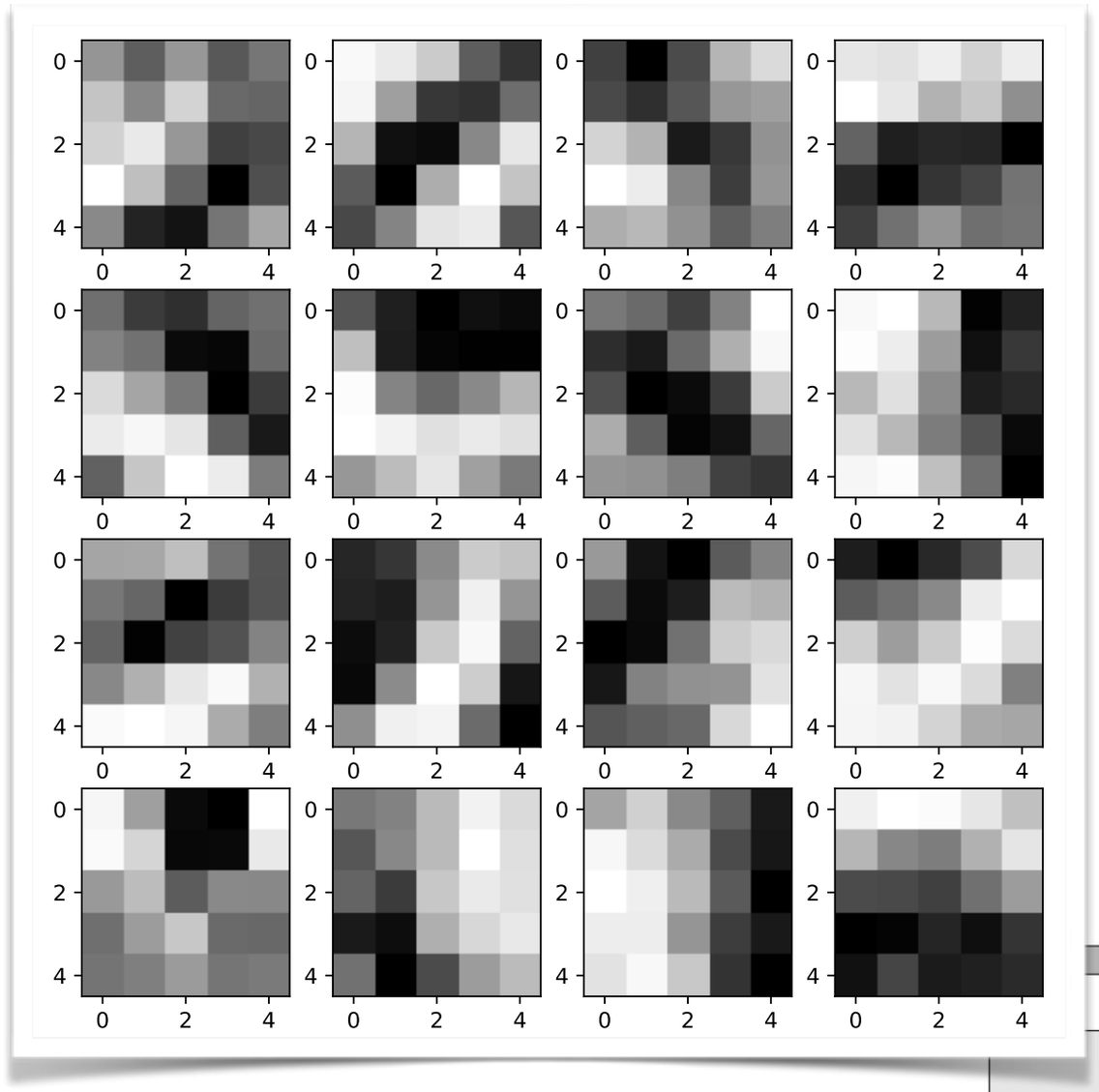
- The second important feature is that the local representative fields have a **shared weights/bias** through out the whole first hidden layer. e.g. the same 5×5 weights and a common bias are shared by all of the neurons on the first hidden layer.
  - This means all of the neurons of the hidden layer can *detect exactly the same feature*.
  - The map from the input layer to the hidden layer is usually called a **feature map**.
  - A feature map only keep **25 weights and 1 bias**!
  - The shared weights/bias are often said to define a **kernel** or **filter**.



THE FEATURE MAPS

- And it is very common to build multiple feature maps, i.e.

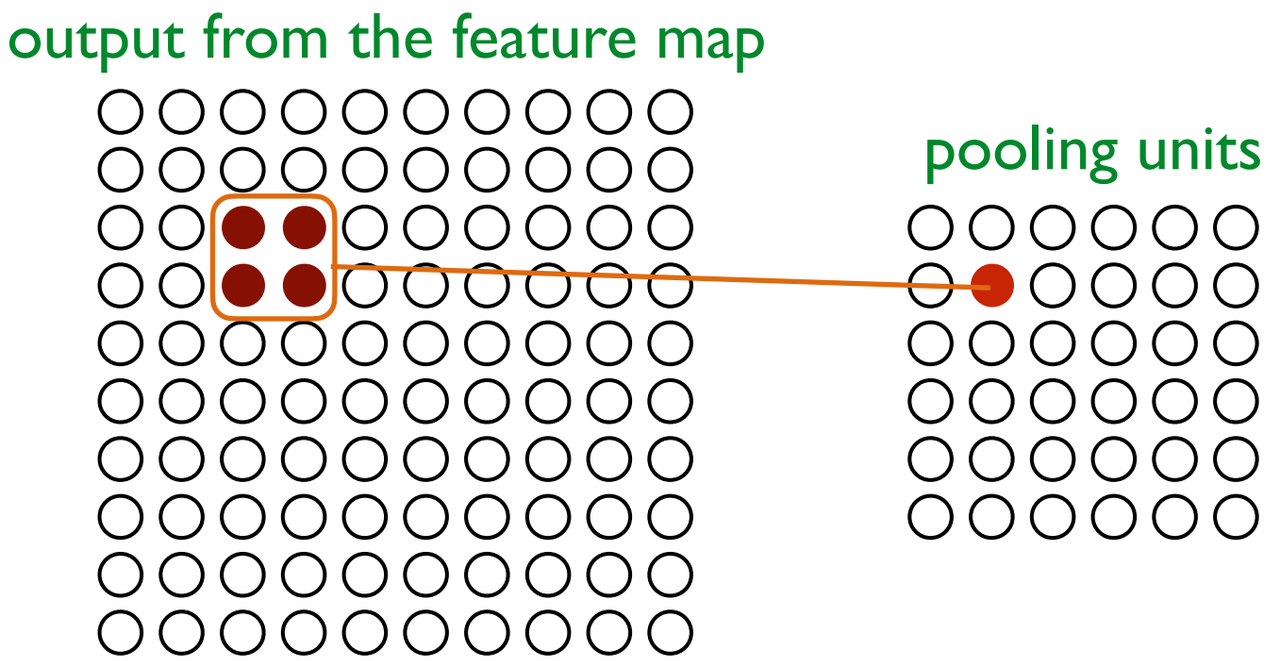




  - For example here are the trained 16 feature maps (or kernels / filters) in the next example.
  - Basically each map supposes to pick up a different feature from the input images!

POOLING LAYERS

- In addition to the convolutional layers, a pooling layer is usually added right after them. A pooling layer is to simplify the information from the convolutional layer, for example a 2×2 pooling layer shrink the input 24×24 feature map into a 12×12 units:

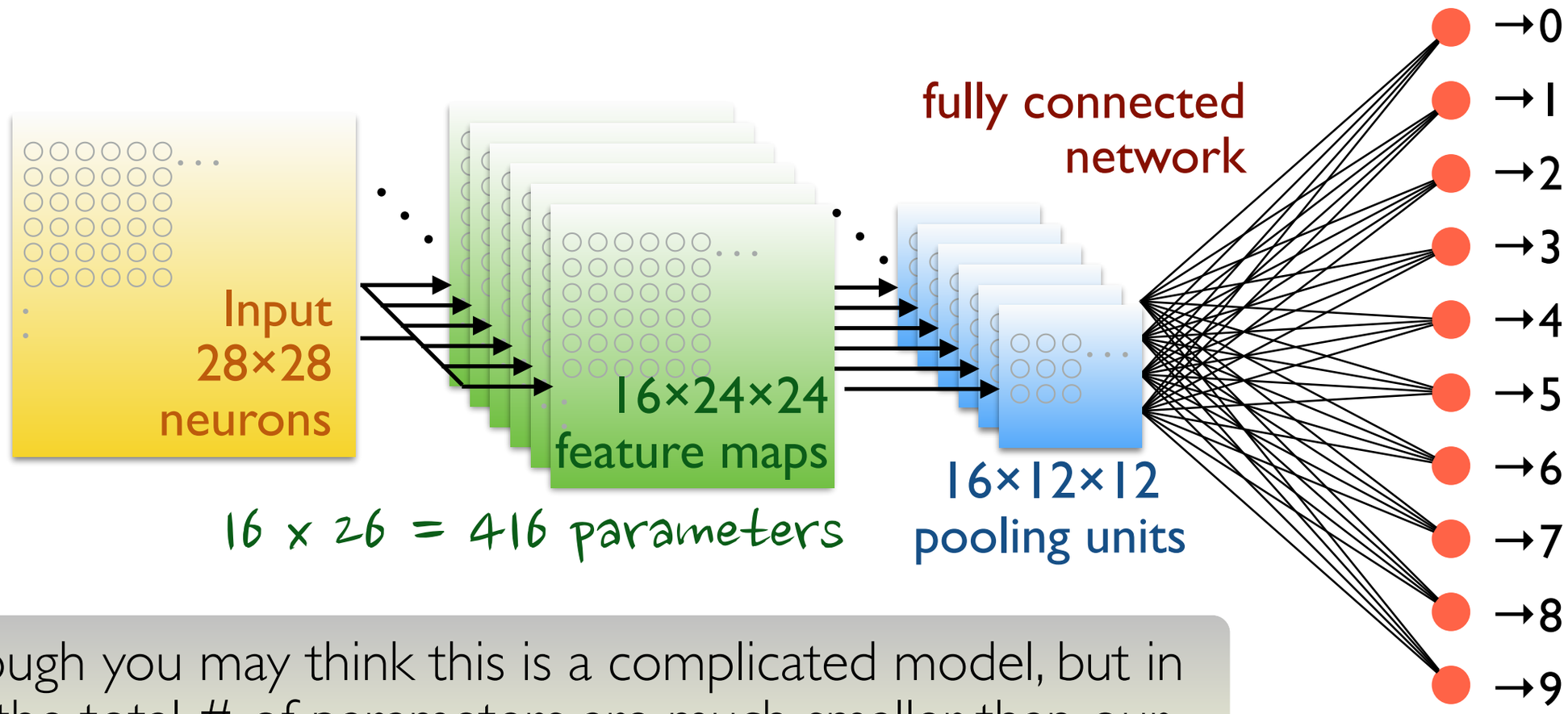


Usually this is applied to each feature map output layer

- **Max-pooling**: simply outputs the maximum activation value in input region.
- **L2 pooling**: take the square root of the quadrature sum of the activations.
- No additional weight/bias but just condensing information from the convolutional layer.

PUT ALL TOGETHER: THE CONVOLUTIONAL NETWORK

- Here we just draw the structure of a typical convolutional network. And it will be implemented in our upcoming example code. We construct a network with 16 filters:



Although you may think this is a complicated model, but in fact the total # of parameters are much smaller than our previous example, only **23,466** weights/bias!

PUT ALL TOGETHER (II)

- (Very) easy implementation with TensorFlow / Keras:

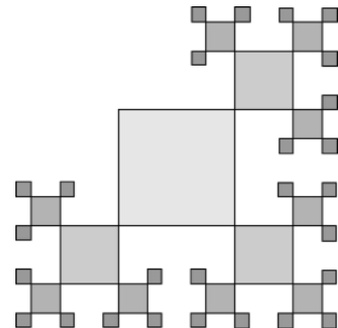
partial **ex_ml8_3.py**

```
. . . . . .
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import *
from tensorflow.keras.optimizers import Adam

model = Sequential()
model.add(Reshape((28,28,1), input_shape=(28,28)))
model.add(Conv2D(16, kernel_size=(5,5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))

model.compile(loss='categorical_crossentropy',
              optimizer=Adam(),
              metrics=['accuracy'])

model.fit(x_train, y_train, epochs=20, batch_size=120,
          validation_data=(x_test, y_test))
. . . . . .
```

Just implement the model discussed in the previous page!

↑ 5x5 convolutional layer

↑ 2x2 pooling layer

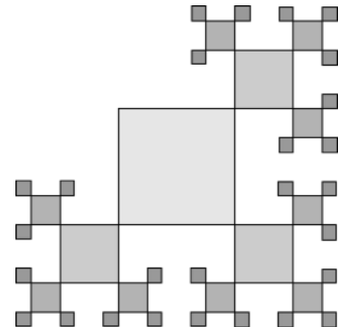


PUT ALL TOGETHER (III)

- And we can reach a very good performance already:

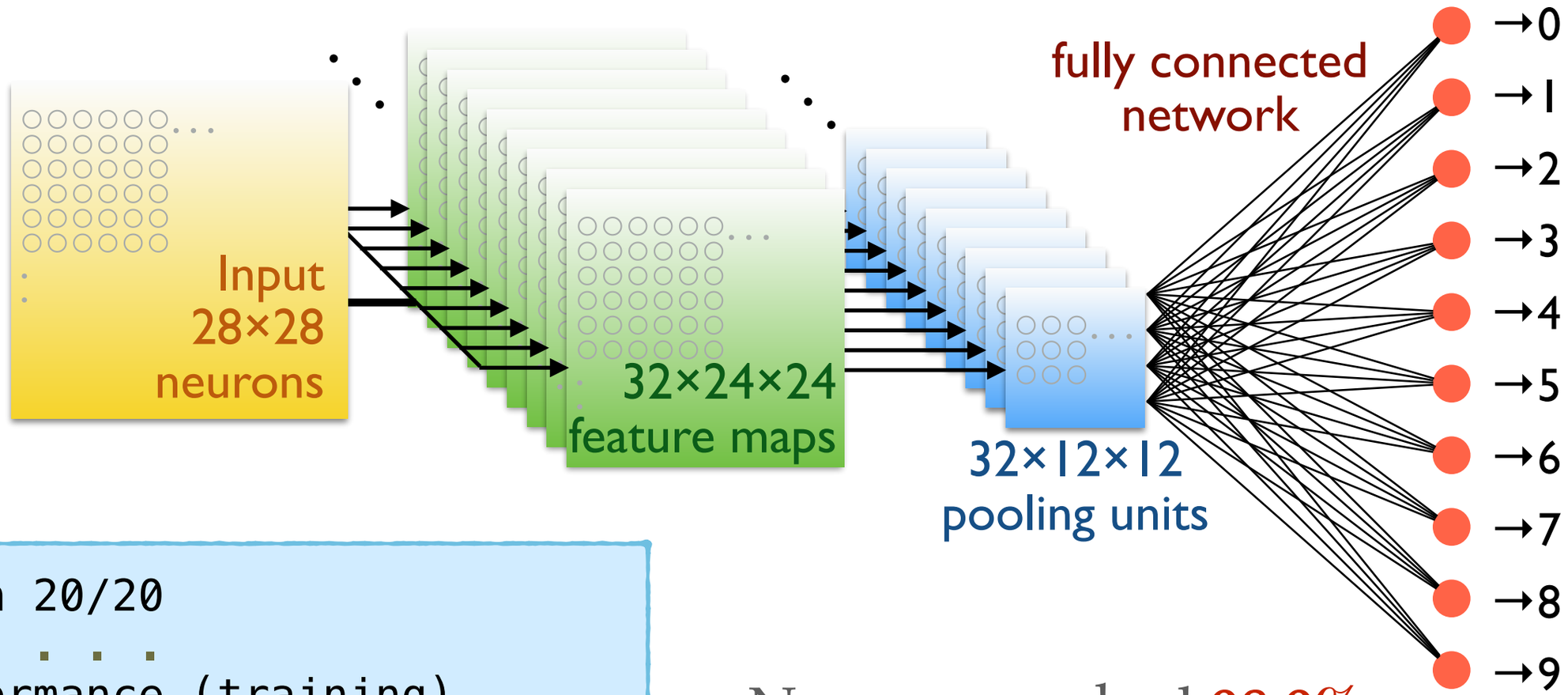
```
Epoch 1/20
500/500 [======] - 7s 13ms/step - loss: 0.7368 - accuracy: 0.8084
. . . . . .
Epoch 20/20
500/500 [======] - 7s 13ms/step - loss: 0.0294 - accuracy: 0.9902
Performance (training)
Loss: 0.01741, Acc: 0.99483
Performance (testing)
Loss: 0.03990, Acc: 0.98780
```

- A testing accuracy of **98.8%** reached, only 122 images are mis-identified. Remember we only put a layer of convolutional network and **# of parameters is reduced by a factor of 28** comparing to the previous flat 784-512-512-10 network!
- Can we do even better? *Let's try to add more layers!*



HOW ABOUT ADDING MORE FEATURES MAPS?

- Let's just doubled the feature maps? Can we improve the model already?

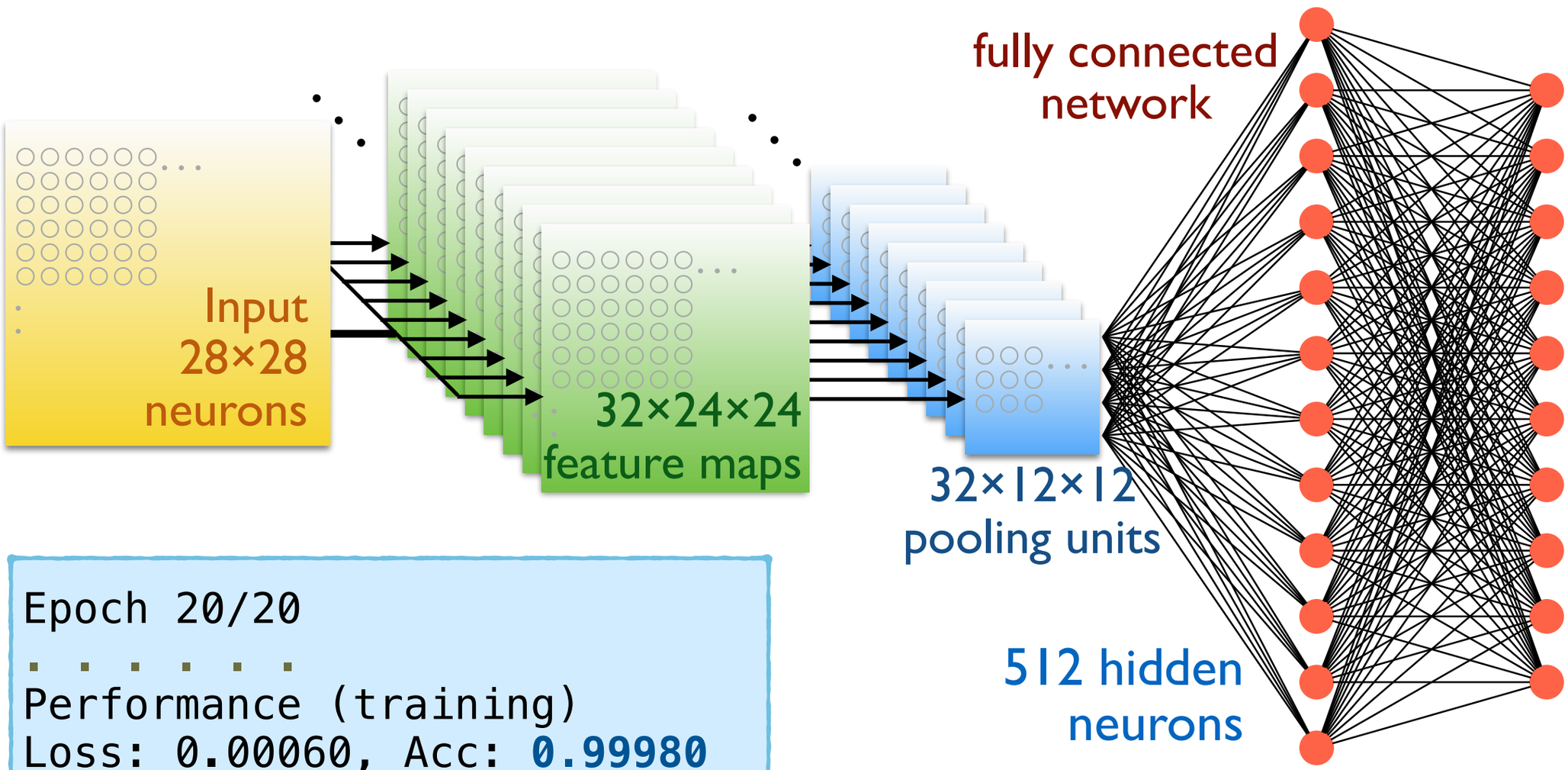



```
Epoch 20/20
. . . . . .
Performance (training)
Loss: 0.00838, Acc: 0.99802
Performance (testing)
Loss: 0.03875, Acc: 0.98890
```

- Now we reached **98.9%** test accuracy, only 111 digits are wrongly tagged!

ADD ANOTHER HIDDEN FULLY CONNECTED LAYER?

- Let's add another fully connected layer and see the performance?




```
Epoch 20/20
. . . . . .
Performance (training)
Loss: 0.00060, Acc: 0.99980
Performance (testing)
Loss: 0.03516, Acc: 0.99190
```

- Now we can reach **99.2%**!

DOUBLED LAYERS!

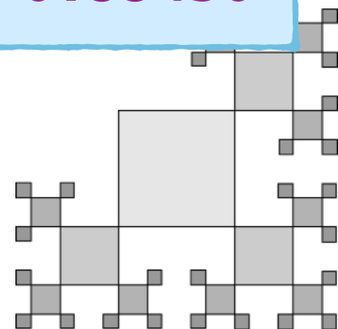
- Let's config our model by adding two convolution+pooling layers, and two fully connected layers. Then see how good can we do from here?

partial ex_ml8_3a.py

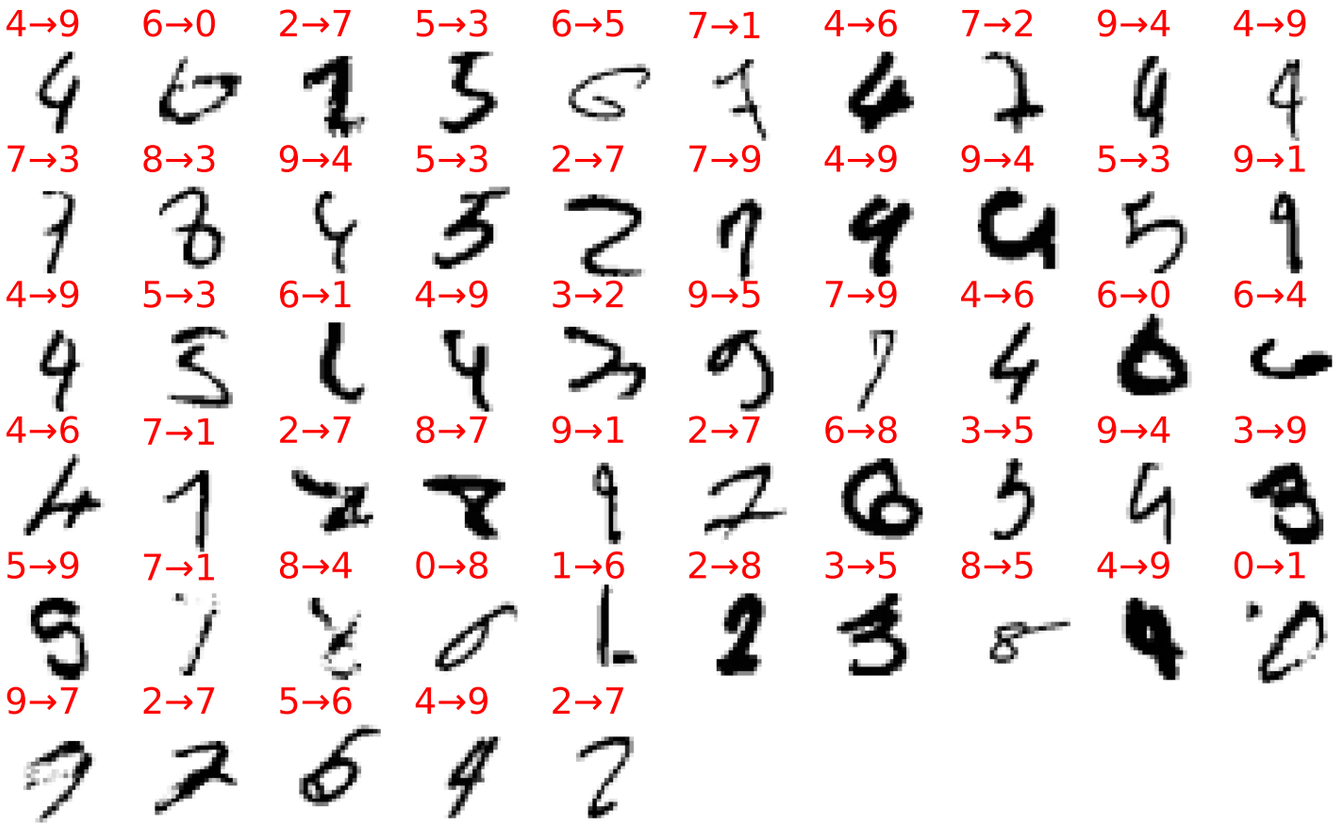
```
. . . . . .
model = Sequential()
model.add(Reshape((28,28,1), input_shape=(28,28)))
model.add(Conv2D(32, kernel_size=(5,5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(32, kernel_size=(5,5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(512, activation='relu'))
model.add(Dropout(0.2))
model.add(Dense(10, activation='softmax'))
. . . . . .
```

```
Epoch 20/20
. . . . . .
Performance (training)
Loss: 0.00252, Acc: 0.99917
Performance (testing)
Loss: 0.02025, Acc: 0.99450
```

- Now we can get close to **99.5%**!



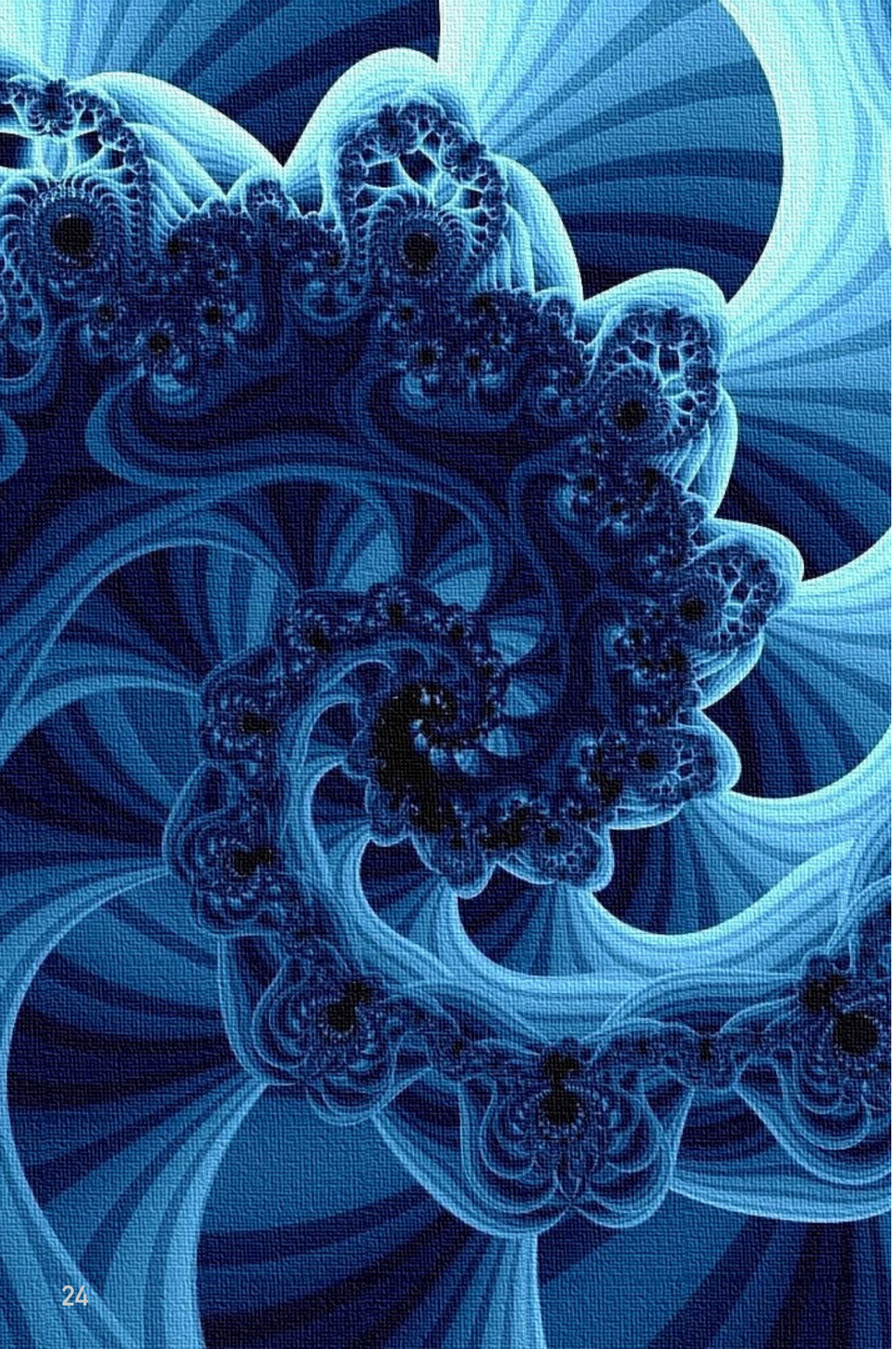
DOUBLED LAYERS! (II)



- Now we only have 55 wrongly tagged images (0.55% failed).
- Some of them are also difficult for real humans!
- Remember the best trained network (*world record*) is with 0.21% failure rate. Still rooms to be improved!

The convolutional neural network is a kind of deep network model **good for image recognition**!

Will discuss more on the deep structured learning in the next module.



MODULE SUMMARY

- In this module we started to discuss the benefit and the limitation of a deeper network, and shown that a network with specific structure may improve the performance on some specific tasks.
- Next module we will continue to discuss more on the network model with different structures, which may be suitable to work on different topics.