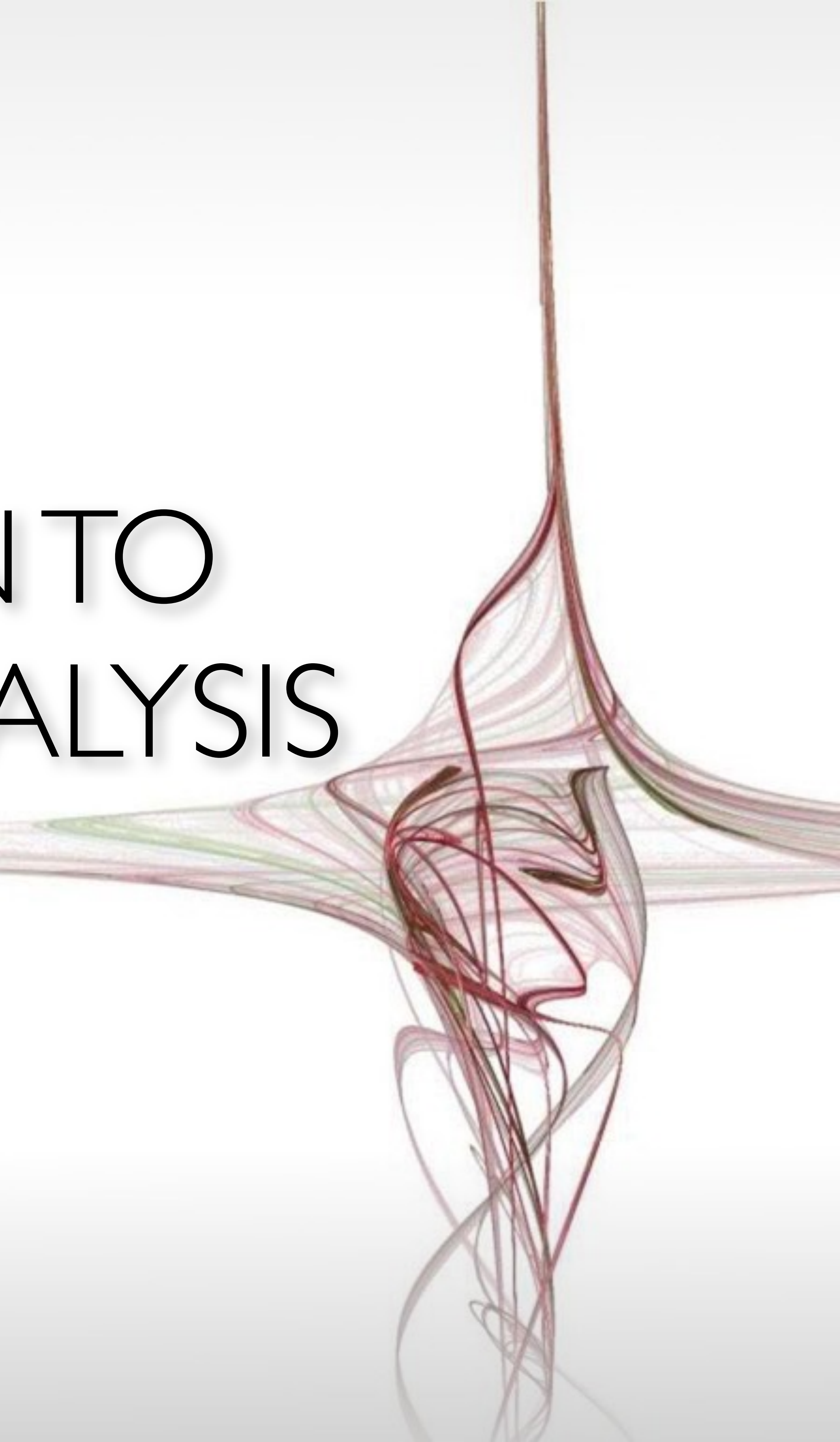
2013

INTRODUCTION TO NUMERICAL ANALYSIS



Lecture 3-3: Improving Neural Network

Kai-Feng Chen
National Taiwan University

RECALL FROM THE LAST LECTURE…(AGAIN!)

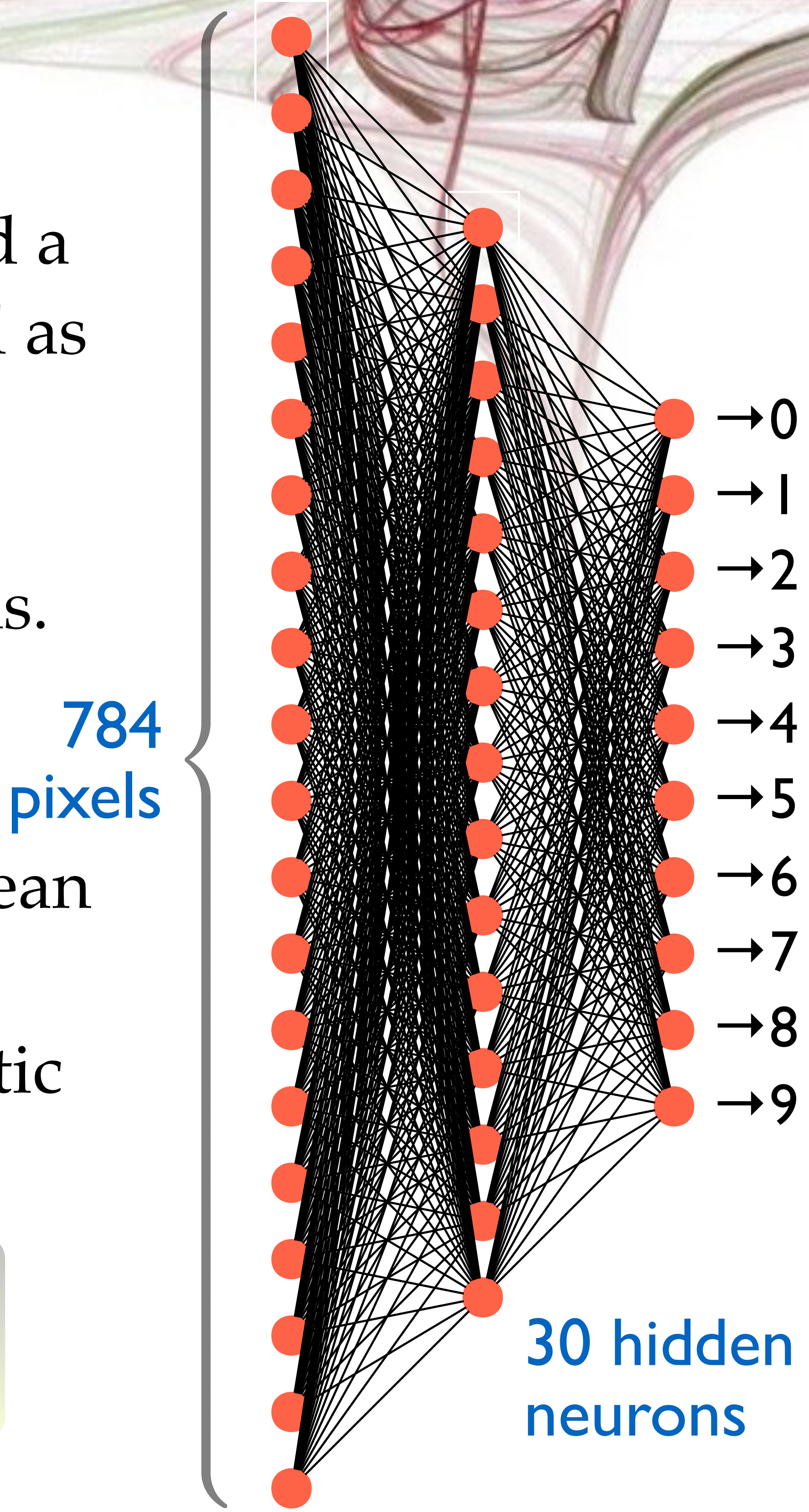
- Last lecture we started to play with two nonlinear models, **SVM with non-linear kernel**, and the very classical **Neural Network**.
- Taking the MNIST data set as an benchmark, the SVM with Gaussian kernel can have a very good performance of ~98.4% accuracy!
- Our super simple neural network can 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 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, or the **convolutional neural network**?

We shall go deeper this time!

BASELINE NETWORK WITH KERAS

- At the end of last lecture we have constructed a simple model with **Keras**. This will be served as our starting point here:
  - 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.

This gives us a **>96%** test accuracy — but what are the wrongly tagged digits?



WRONGLY RECOGNIZED DIGITS?

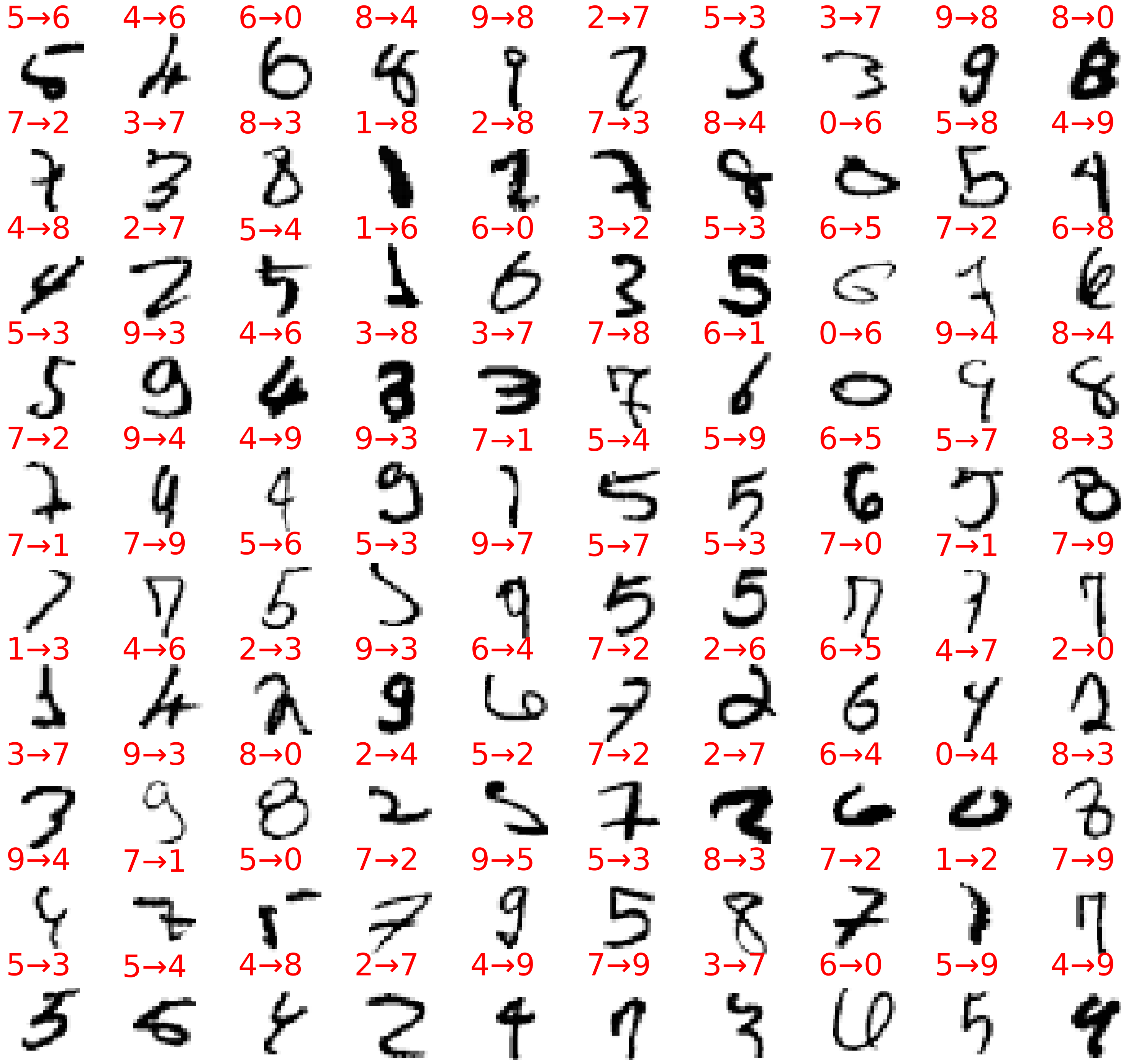
- At the end of the training we can feed the test data into the network and see the resulting "test" performance. An accuracy of 96.6% means we have around ~300 images were wrongly tagged by our network.
- The following piece of code is prepared to show **first 100 of the wrongly tagged images**:

```
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

l303-example-01.py (partial)

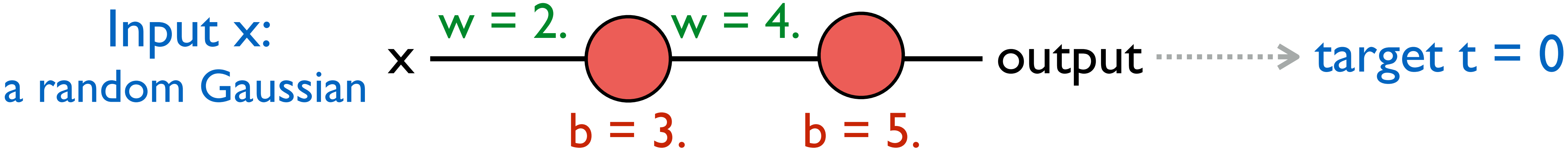
WRONGLY RECOGNIZED DIGITS? (II)



- 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 your 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:

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

from keras.models import Sequential
from keras.layers import Dense
from 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()
```

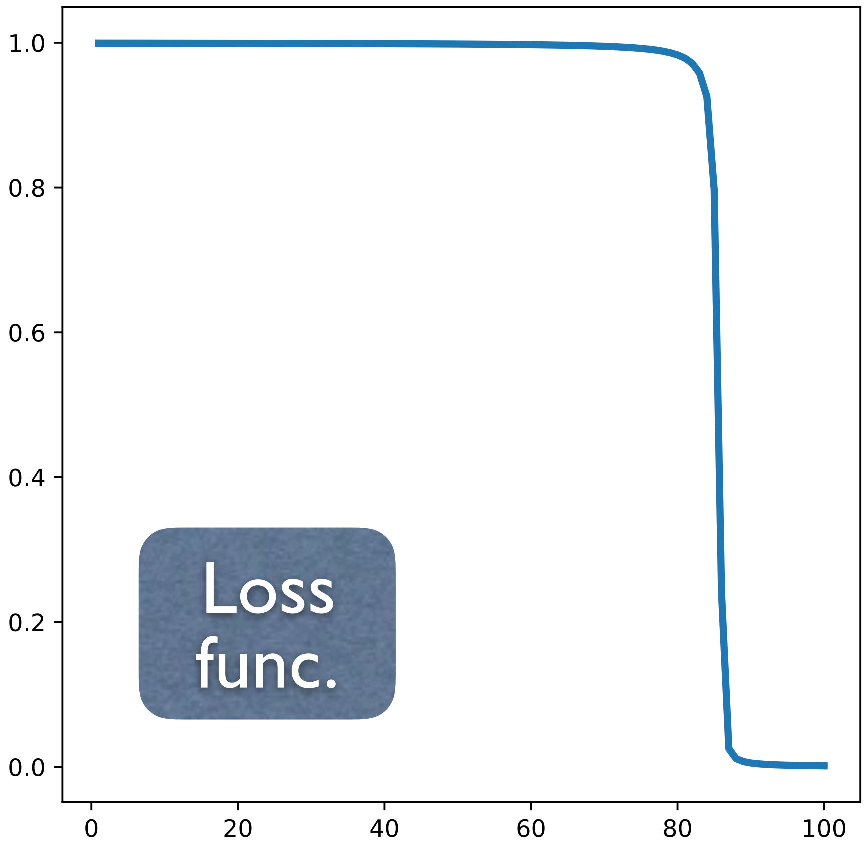
A simple sequential model
⇓with only 1 input / 1 output neuron

⇑ keep the history of training

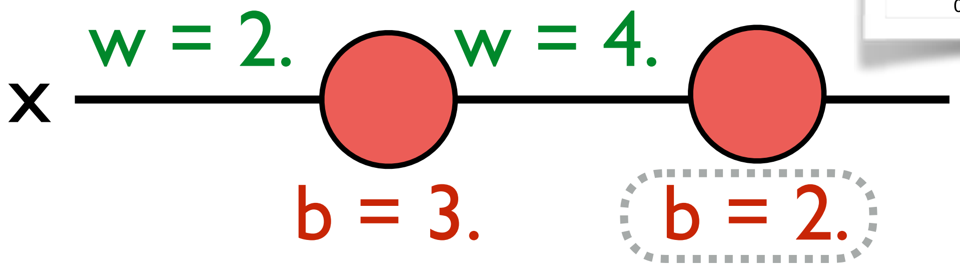
1303-example-02.py (partial)

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.



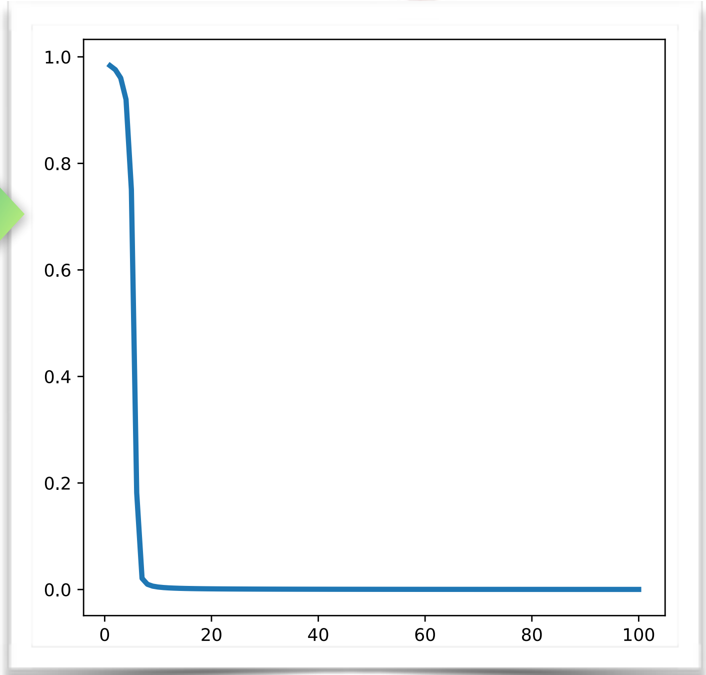
x

w = 2.

w = 4.

b = 3.

b = 2.



Both this also hints a problem of our network!

THE CHOICE OF LOSS

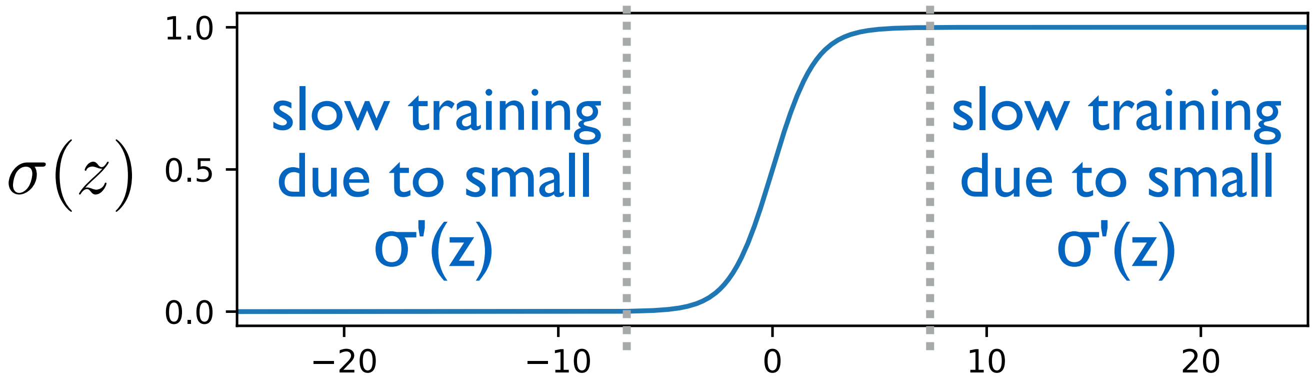
- In fact such a situation can be related to the definition of the loss function, and its gradient w.r.t. the weights and bias.
- Consider the current choice of loss, the **mean squared error**:

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

Gradient is required in the training process:

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

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



slow training due to small σ'(z)

slow training due to small σ'(z)

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:

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

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

Cancelled

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

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

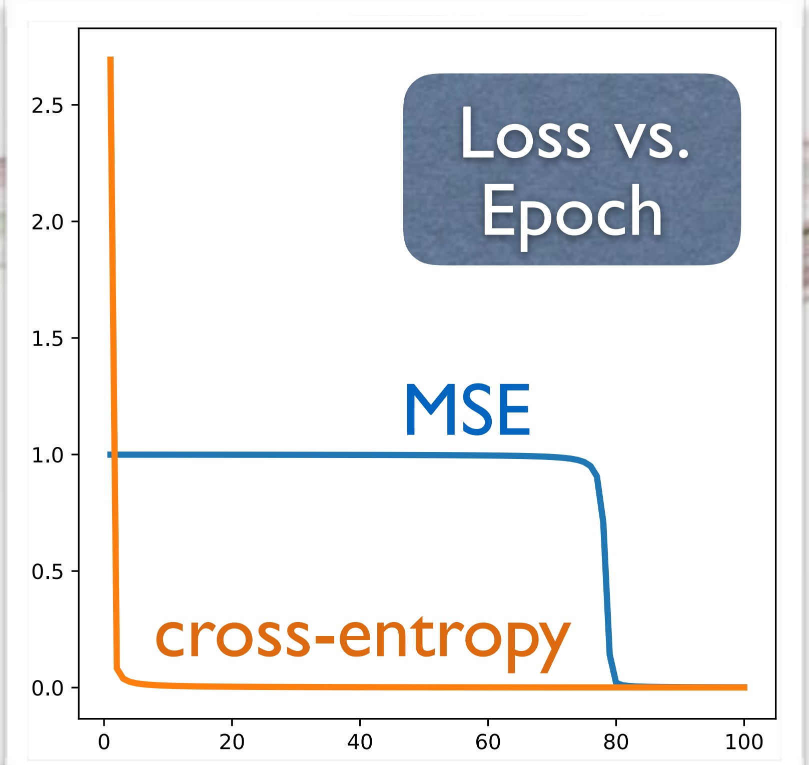
→ $\sigma'(z) = \sigma(z)[1-\sigma(z)]$

A little bit of calculous...

Not depending on the first derivative **σ'(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()
```

1303-example-02a.py (partial)

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

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

- 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 $\boldsymbol{y_j}$ w.r.t. $\boldsymbol{z_i}$ (remember $\boldsymbol{z_i}$ is linear sum of weights times the outputs from previous layer + bias):

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

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 ends up with the same results as before and no dependency on **σ′(z)**!

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

It should solve the slow learning problem as well.

TRY IT OUT!

```python
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)
```

1303-example-03.py (partial)

```
Using TensorFlow backend.
Epoch 1/20
60000/60000 [==============] - 3s 45us/step - loss: 0.2924 - acc: 0.9114
. . . . . . .
Epoch 20/20
60000/60000 [==============] - 2s 41us/step - loss: 0.0465 - acc: 0.9850
Performance (training)
60000/60000 [==============] - 1s 22us/step
Loss: 0.04225, Acc: 0.98613
Performance (testing)
10000/10000 [==============] - 0s 23us/step
Loss: 0.13640, Acc: 0.96350
```

Although the performance for training sample is improved, but the performance for testing sample is still similar!

COMMENT

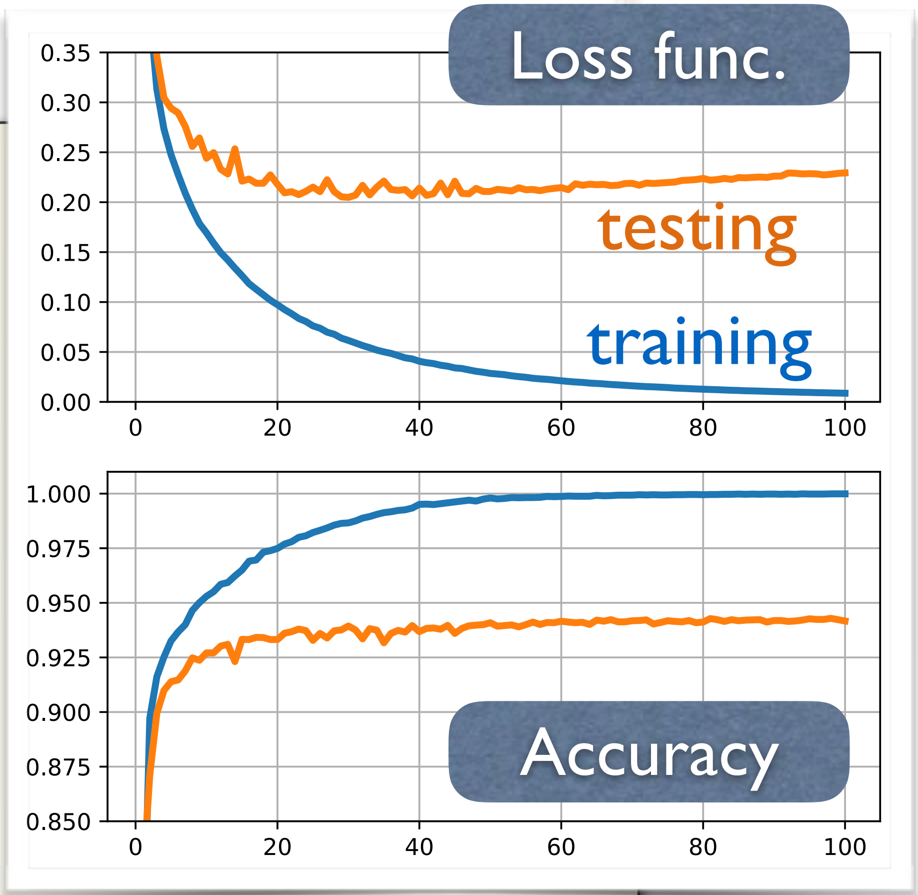
- We have two possible treatments that can be used in the classification problem:
  - **sigmoid activation + binary cross-entropy loss**
  - **softmax layer + categorical cross-entropy loss**
- You may find they have a very similar formulation and similar behavior. This is due to the fact that sigmoid 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 give a different accuracy value if you do sigmoid activation + binary cross-entropy loss

THE OVERTRAINING ISSUE

- We have touched slightly on this issue at the end of last lecture. Now we shall come back to it again.
- The training performance is indeed keeping improving with more epochs, but the testing performance saturated quickly.
- Demonstration with Keras tool again:

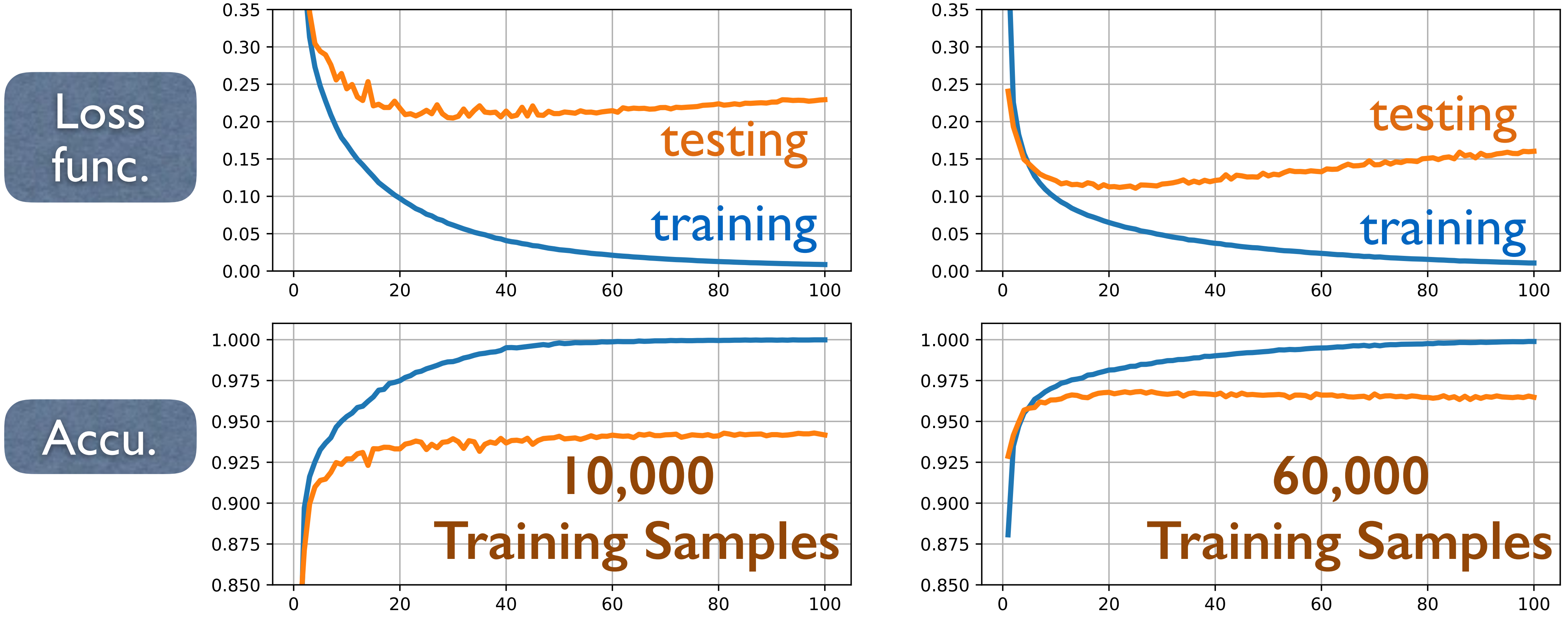
```
rec = model.fit(x_train, y_train,
          epochs=100, batch_size=120,
          validation_data=(x_test, y_test))

vep = np.linspace(1.,100.,100)
fig = plt.figure(figsize=(6,6), dpi=80)
plt.subplot(2,1,1)
plt.plot(vep,rec.history['loss'], lw=3)
plt.plot(vep,rec.history['val_loss'], lw=3)
plt.subplot(2,1,2)
plt.plot(vep,rec.history['acc'], lw=3)
plt.plot(vep,rec.history['val_acc'], lw=3)
plt.show()
```



l303-example-04.py (partial)

TRAINING DATA DOES MATTER

- In the previous example we have only input 10K sets of training sample. By increasing the training data size the overtraining is indeed mitigated:



But in many of the cases training samples are difficult to collect and expensive. Can we do something without just adding more the data?

REGULARIZATION

- A method is called “Regularization” or “weight decay” may help to reduce the overtraining situation.
- The idea is to introduce an additional term to the loss function:

$$L = L_0 + \frac{\lambda}{n}\sum |w| \quad \text{or} \quad L = L_0 + \frac{\lambda}{2n}\sum w^2$$

- The form given above is usually called the **L1/L2 regularization**, where the ***λ*** is the *regularization parameter* (***λ*>0**) and ***n*** is the size of training sample.
- One can see the gradient of the loss function will be modified and change the learning step (taking L2 regularization as an example):

$$\frac{\partial L}{\partial w} = \frac{\partial L_0}{\partial w} + \frac{\lambda}{n} w \longrightarrow w \Rightarrow w - \eta\frac{\partial L_0}{\partial w} - \eta\frac{\lambda}{n} w$$

The weights will “decay” by a factor during the training process.

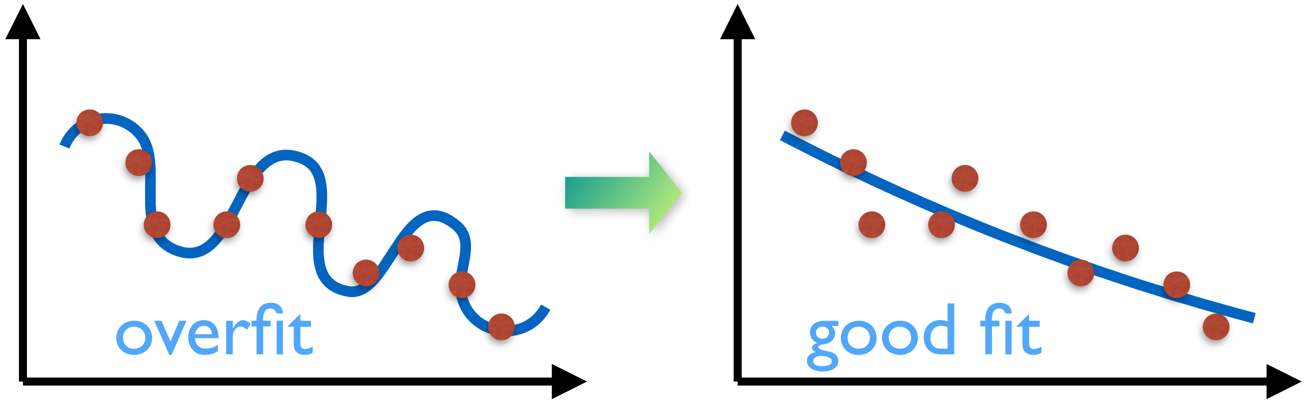
$$= \left(1 - \eta\frac{\lambda}{n}\right) w - \eta\frac{\partial L_0}{\partial w}$$

REGULARIZATION (II)

- By introducing such a “weight decay” to the training, the weights will be pushed toward smaller values. But why a network with smaller weights can have a smaller overtraining problem?
- Consider a fit to the data points along the x-axis, the “weights” are just the coefficients of the polynomial terms:

$$f(x) = w_0 + w_1 x + w_2 x^2 + w_3 x^2 + w_4 x^4 + w_5 x^5 + \cdots$$

$$\Rightarrow f(x) = w_0 + w_1 x + w_2 x^2 + 0x^2 + 0x^4 + 0x^5 + \cdots$$



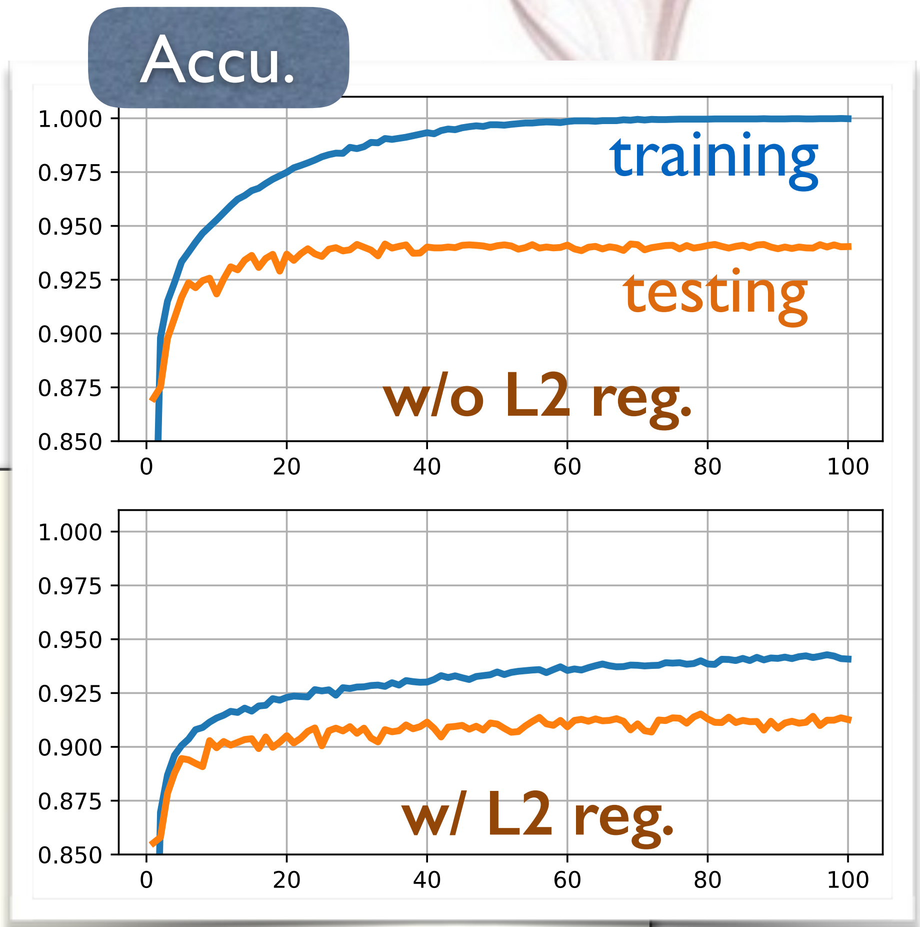
By reducing the weights for all terms, it actually removes the higher order term and make the fit to be less sensitive to the noise (local fluctuation), and resulting a more robust model.

REGULARIZATION (III)

- Let’s try this method quickly with Keras. If we simply add this weight decay feature to *the weights of the output layer,* one can see the overtraining effect is reduced:

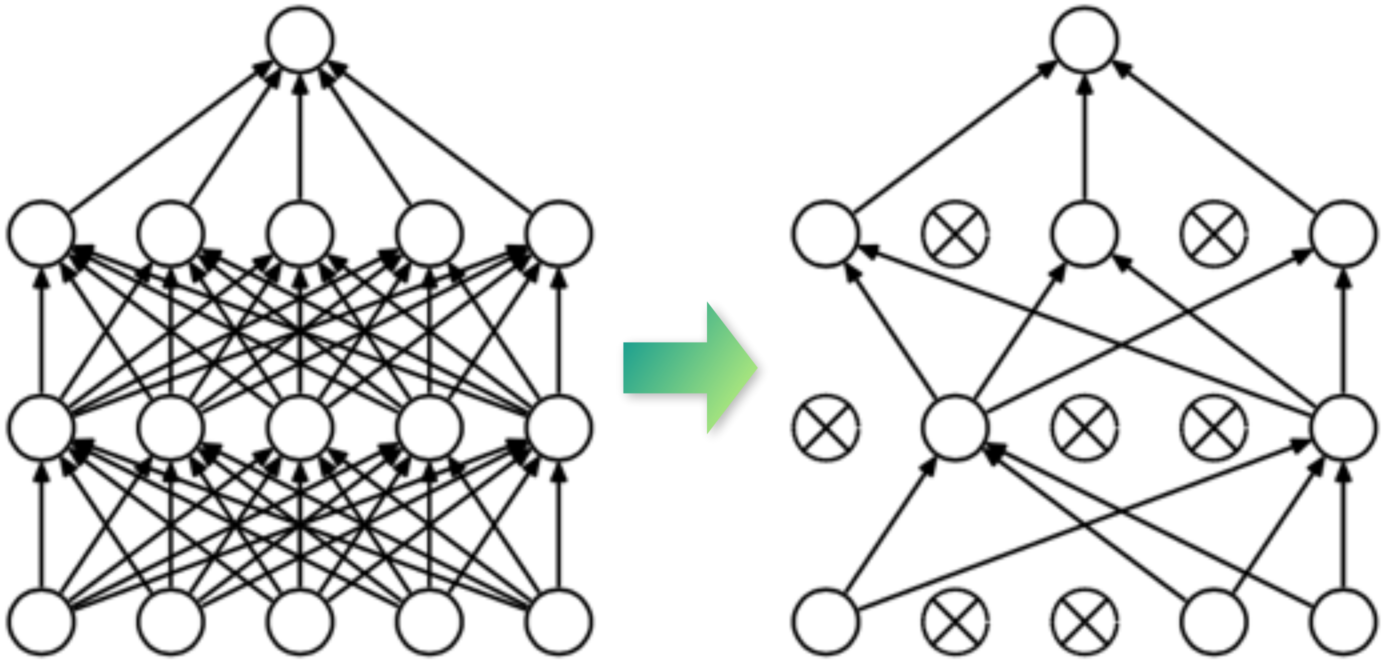
```
. . . . . .
from keras.regularizers import l2

m2 = Sequential()
m2.add(Reshape((784,), input_shape=(28,28)))
m2.add(Dense(30, activation='sigmoid'))
m2.add(Dense(10, activation='softmax',
kernel_regularizer=l2(0.01)))
m2.compile(loss='categorical_crossentropy',
           optimizer=SGD(lr=1.0), metrics=['accuracy'])
. . . . . .
```

l303-example-04a.py (partial)



DROPOUT

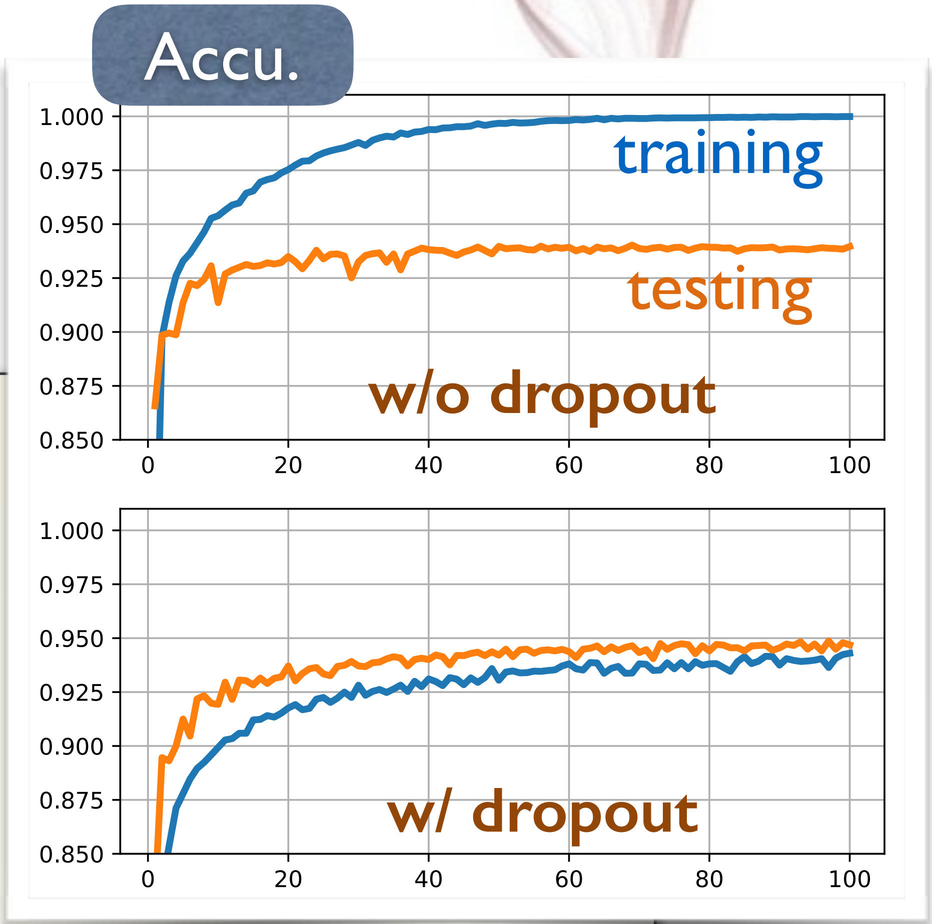
- Another useful method to reduce the overtraining is the **dropout** technique. Dropout does not change the loss function, but change the network structure itself.
- That is, one can randomly disconnect some of the inputs of a specific layer/neurons at each training cycle:



The dropout method would reduce the dependence of the network to some specific neurons or weights, and hence it will be less sensitive to the noise and become more robust against the overtraining.

DROPOUT (II)

- And you can find that the dropout method is actually very helpful in terms of against overtraining:



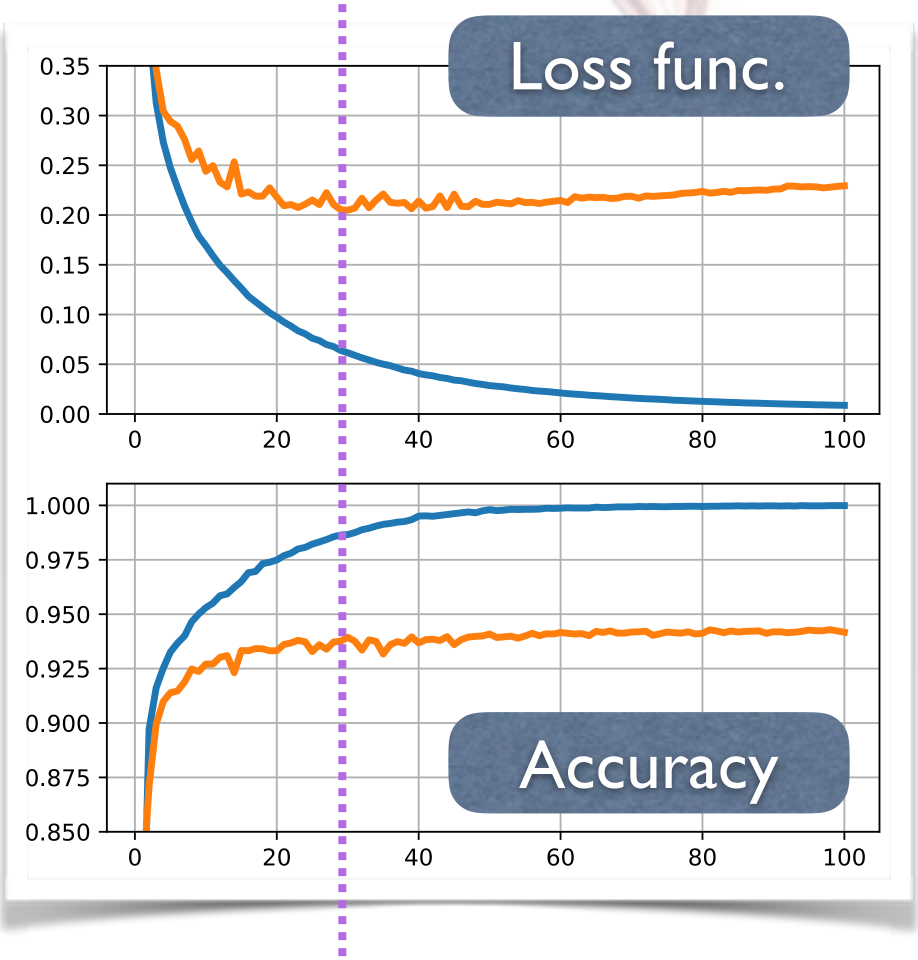
Drop 20% of the inputs randomly

```
. . . . . .
from keras.layers import Dropout

m2 = Sequential()
m2.add(Reshape((784,), input_shape=(28,28)))
m2.add(Dropout(0.2))
m2.add(Dense(30, activation='sigmoid'))
m2.add(Dropout(0.2))
m2.add(Dense(10, activation='softmax'))
m2.compile(loss='categorical_crossentropy',
           optimizer=SGD(lr=1.0), metrics=['accuracy'])
. . . . . .
```

l303-example-04b.py (partial)

EARLY STOPPING

- In fact one can even think of something super simple: why cannot we just stop training immediately when we find the model just becomes overtrained?
- Such a scenario is usually called **"Early Stopping"**. This can be achieved by monitoring the performance of the model during the training process, and terminate the job when the model stop improving.
- It is usually recommended to adopt this criteria on **an independent validation sample** (not the training, nor the testing samples!)



Stop training here?

EARLY STOPPING (II)

- This can be carried out by a “callback” function within Keras:

```
mnist = np.load('mnist.npz')
x_train = mnist['x_train'][:10000]/255.
y_train = np.array([np.eye(10)[n] for n in mnist['y_train'][:10000]])
x_valid = mnist['x_train'][50000:]/255.
y_valid = np.array([np.eye(10)[n] for n in mnist['y_train'][50000:]])
x_test = mnist['x_test']/255.
y_test = np.array([np.eye(10)[n] for n in mnist['y_test']])
. . . . . .
from keras.callbacks import EarlyStopping

rec = model.fit(x_train, y_train, epochs=100, batch_size=120,
        validation_data=(x_valid, y_valid),
        callbacks=[EarlyStopping(monitor='val_loss', patience=3)])

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

l303-example-04c.py (partial)

EARLY STOPPING (III)

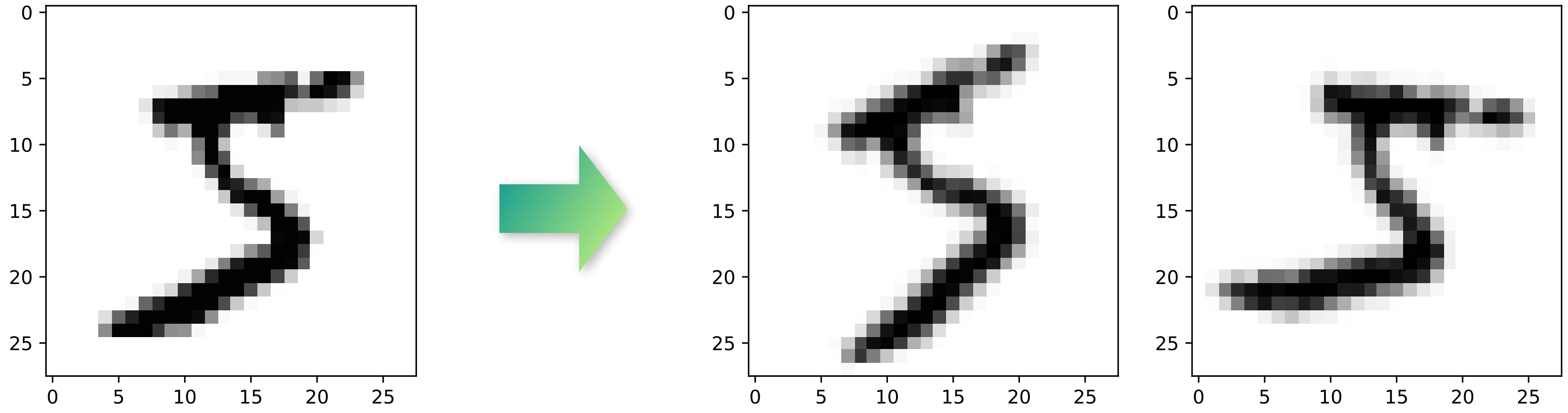
- The learning process is stopped automatically after 21 epochs:

```
Train on 10000 samples, validate on 10000 samples
Epoch 1/100
10000/10000 [==============] - 0s 27us/step - loss: 0.9330 - acc: 0.7552 -
val_loss: 0.5119 - val_acc: 0.8510
Epoch 21/100
10000/10000 [==============] - 0s 21us/step - loss: 0.0876 - acc: 0.9783 -
val_loss: 0.2068 - val_acc: 0.9373
Performance (training)
Loss: 0.09317, Acc: 0.97770
Performance (validation)
Loss: 0.20682, Acc: 0.93730
Performance (testing)
Loss: 0.21634, Acc: 0.93480
```

The reason to setup another **validation sample** here is to keep that the **testing sample always provides a unbiased performance estimate**. The validation sample here is "used" to decide the ending of the training process already. This validation setup is also recommended for hyper parameter and model tuning.

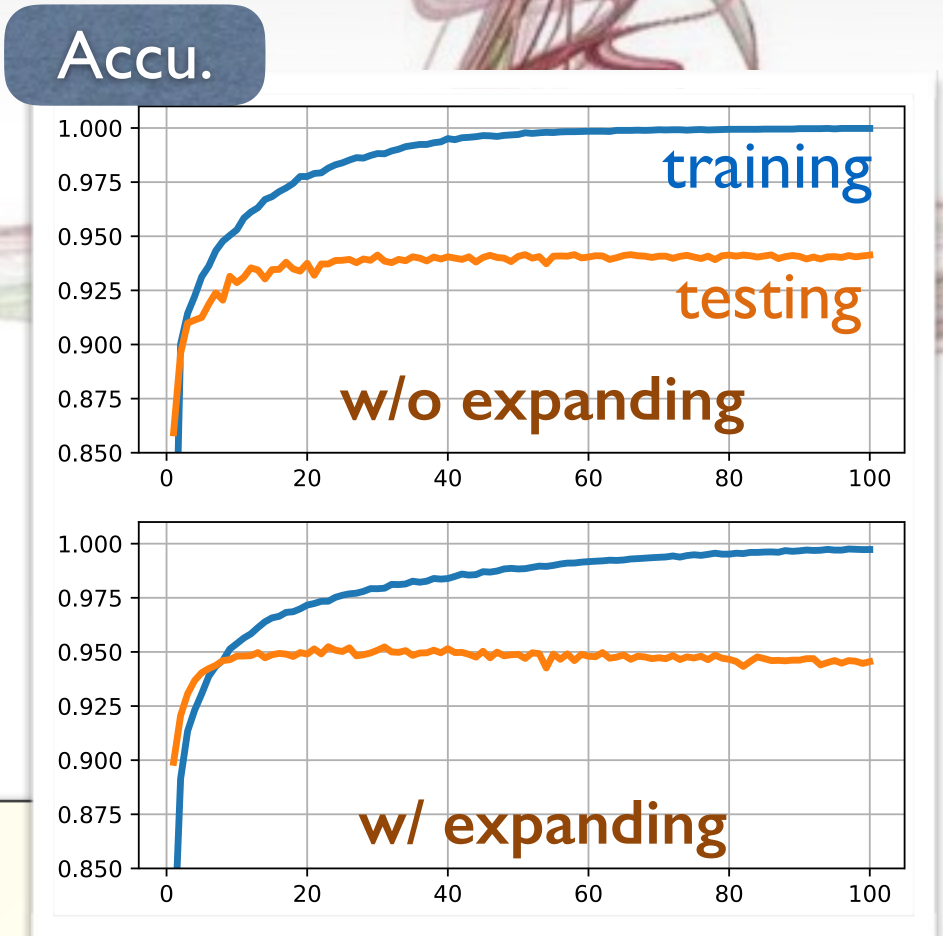
WHAT ELSE WE CAN DO?

- As we mentioned earlier, the size of training sample does matter. With a larger training sample size, the issue of overtraining can be mitigated. But if we cannot collect more data?
- A method can still be tried is **artificially increasing the training data**. This is in fact a very reasonable technique — remember in our example the training data are just images of handwriting digits. One can, slightly, twist or rotate the input images and it can be used as another training sample. This will help the network to catch the correct feature of the input images but not the small distortion nor the local noise.



ARTIFICIAL DATA EXPANDING

- Let’s **triple the training data** by randomly rotate the images either +5°~+25°, or –5°~–25°. Some positive effect found!



```
. . . . . .
from skimage.transform import rotate
ext1 = np.array([rotate(img,np.random.uniform(+5.,+25.)) for
img in x_train])
ext2 = np.array([rotate(img,np.random.uniform(-25.,-5.)) for
img in x_train])
x_train_ext = np.vstack([x_train,ext1,ext2])
y_train_ext = np.vstack([y_train,y_train,y_train])
. . . . . .
rec1 = m1.fit(x_train, y_train, epochs=100,
    batch_size=120,validation_data=(x_test, y_test))
rec2 = m2.fit(x_train_ext, y_train_ext, epochs=100,
    batch_size=120,validation_data=(x_test, y_test))
. . . . . .
```

1303-example-05.py (partial)

INTERMISSION

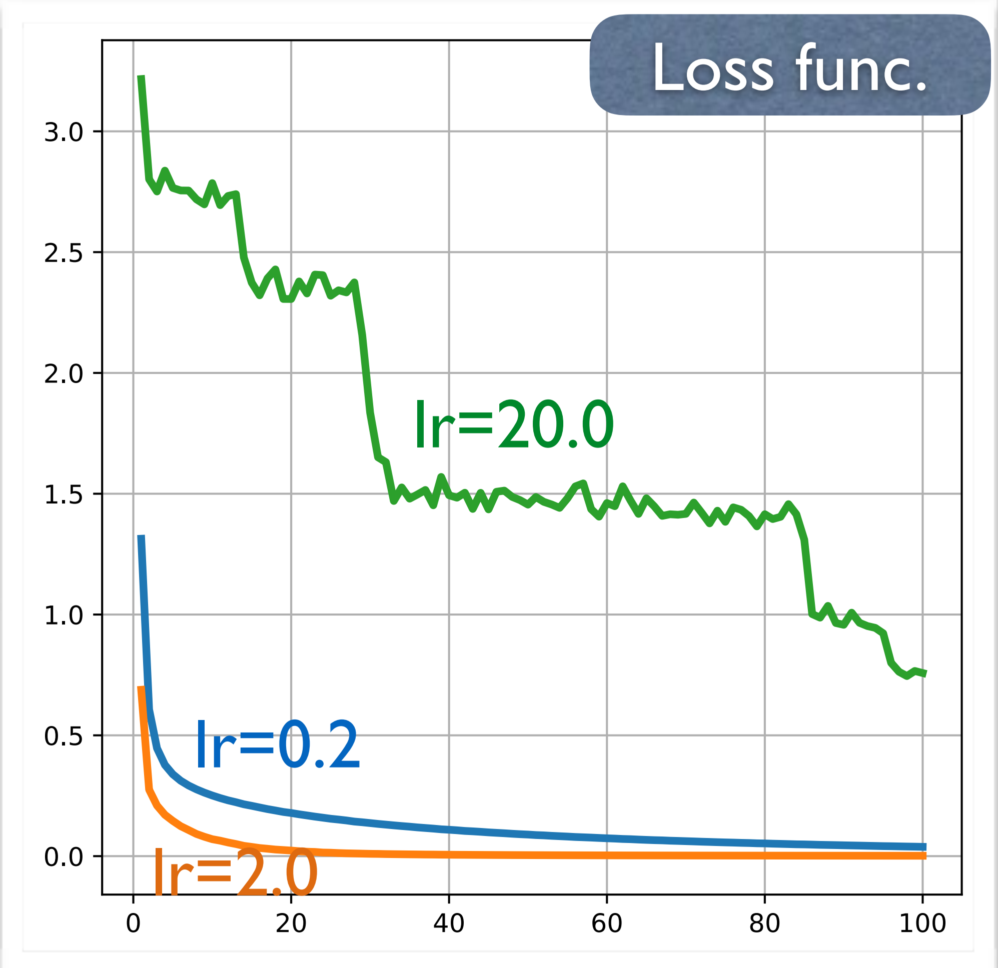
- We have introduced several methods to improve the learning speed, and touched the issue of overtraining. If port (some of) them back to the original example `l303–example–01a.py`, what's the performance you can reach by now already?
- Surely this will take a long time to run, in particular if you expand the training data size! Be aware!

CAN WE DO SOMETHING WITH THE LEARNING METHOD?

- So far we are always using standard stochastic gradient descent method with a given learning rate. Will a large/smaller learning rate helps, or can one do something else to improve the learning?
- Let’s examine this by comparing the results with different learning rates:

```
. . . . . .
m1.compile(loss='categorical_crossentropy',
           optimizer=SGD(lr=0.2))

m2 = clone_model(m1)
m2.compile(loss='categorical_crossentropy',
           optimizer=SGD(lr=2.0))

m3 = clone_model(m1)
m3.compile(loss='categorical_crossentropy',
           optimizer=SGD(lr=20.))
. . . . . .
```

l303-example-06.py (partial)

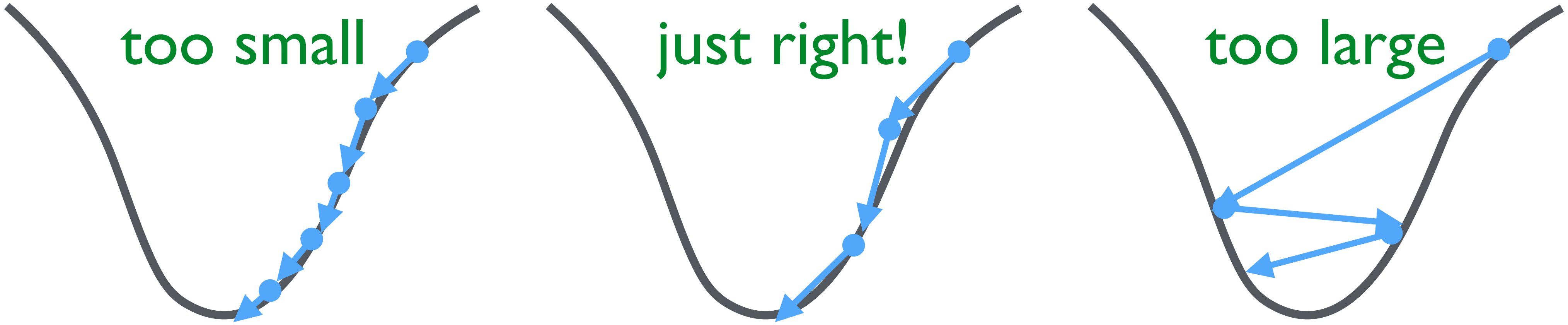


LEARNING RATE

- Come back to the definition of the learning method itself:

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

- The learning rate basically decide how much we should move at each step. Too small learning rate will take a long time to train the network (*but you can already image for a super long run this might be better!*); too large rate will make the learning more likely a random walk.
- Sometimes it might be a good idea to **decrease the learning rate over epoch** and it might end up with a slightly better network, if the network training already saturated quickly.



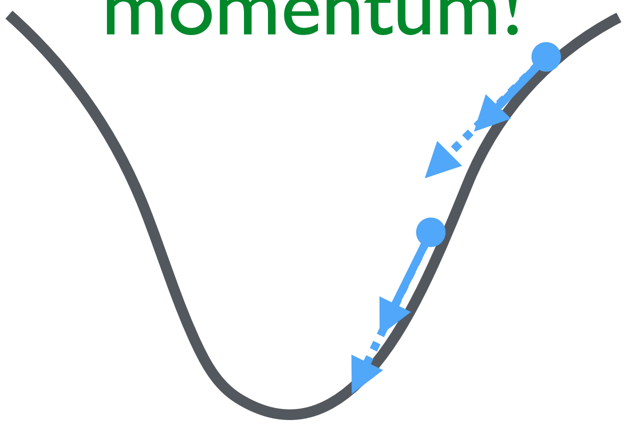
KEEP THE MOMENTUM?

- One can imagine the training with SGD is more likely to go downhill in a valley. If the current direction is good (obviously going toward lower altitude), why not to keep the **MOMENTUM** of your moving?
- This can be also an option within SGD algorithm to enable a momentum based update. It might speed up the training with a proper setup.
- Both of the options (decay of learning rate, momentum) are supported within the framework of Keras:

```
keras.optimizers.SGD(lr=0.01,
momentum=0.0, decay=0.0, nesterov=False)
```

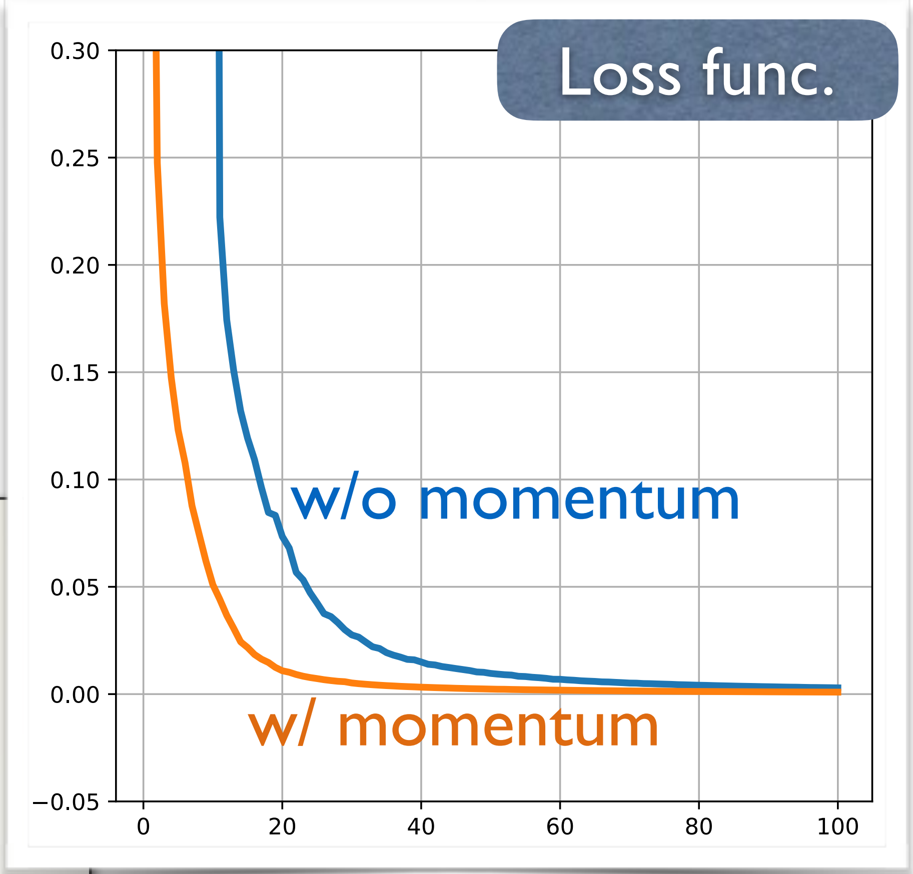
Note: the "nesterov" option is a kind of improved momentum method!

Keep the momentum!



KEEP THE MOMENTUM? (II)

- Again, let's try these option(s)!
- We only tested "momentum" since it can speed up of the training, while the decay of learning rate is generally for the network fine-tune and it is hard to see the effect quickly.
- The loss function converges quicker with momentum method!


Loss func.

```
. . . . . .
m1.compile(loss='categorical_crossentropy',
           optimizer=SGD(lr=2.0))

m2 = clone_model(m1)
m2.compile(loss='categorical_crossentropy',
           optimizer=SGD(lr=2.0, momentum=0.4))
. . . . . .
```

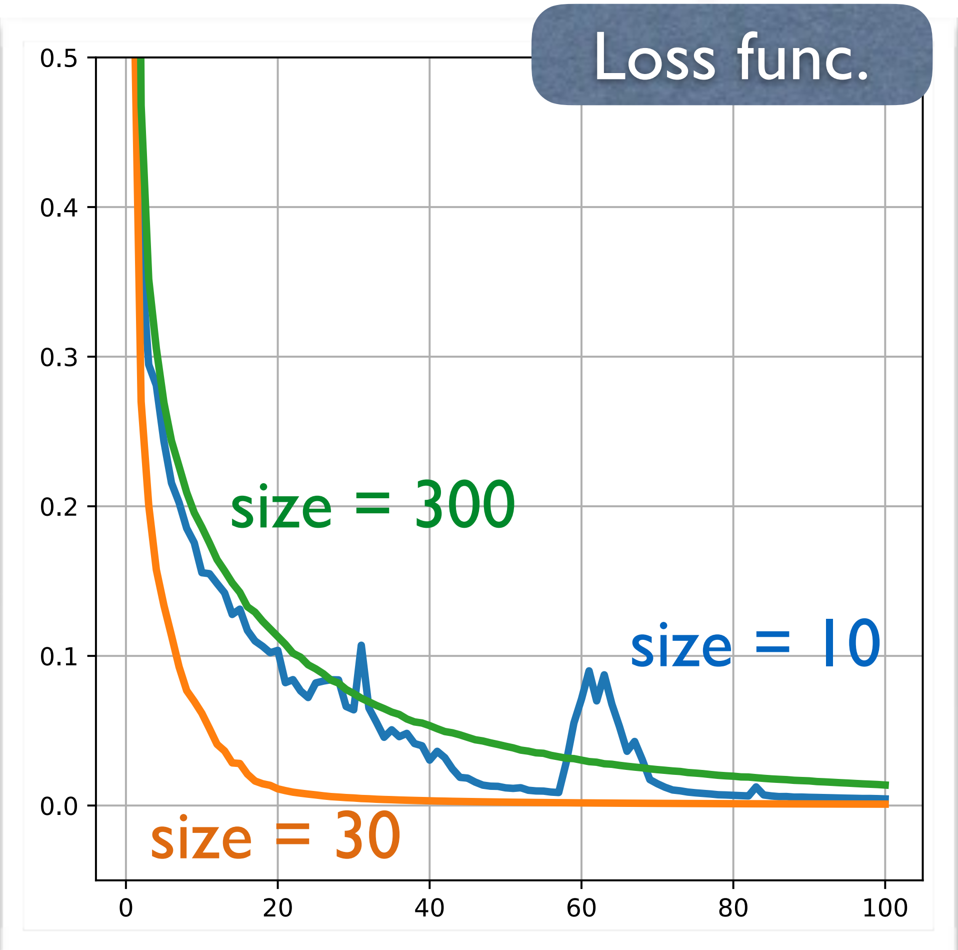
l303-example-06a.py (partial)

SIZE OF MINI-BATCH?

- We have not discussed the mini-batch size, but you may / can already tried to train your network with different mini-batch size!
- In principle larger mini-batch will reduce the "randomness" of the SGD algorithm and results a smoother training, but it also suffers from less frequent updates. But too small mini-batch will also make your training like a random walk.

```
. . . . . .
rec1 = m1.fit(x_train, y_train, epochs=100, 
batch_size=10)
rec2 = m2.fit(x_train, y_train, epochs=100, 
batch_size=30)
rec3 = m3.fit(x_train, y_train, epochs=100, 
batch_size=300)
. . . . . .
```

l303-example-06b.py (partial)



DIFFERENT TRAINING ALGORITHM?

- SGD algorithm is powerful and easy to understand / implement, but there are some issues indeed:
  - Only depends on the gradient calculated by the batched data.
  - Difficult to choose a proper learning rate, and all parameters are learning with the same speed (*only a global learning rate*).
  - May run into a local minimum instead of the global one.
- This is the reason why there are many other algorithms developed to improve these points.
- Many of these SGD variations introduce an **adaptive learning rate** according to the situation of the network training.

DIFFERENT TRAINING ALGORITHM? (II)

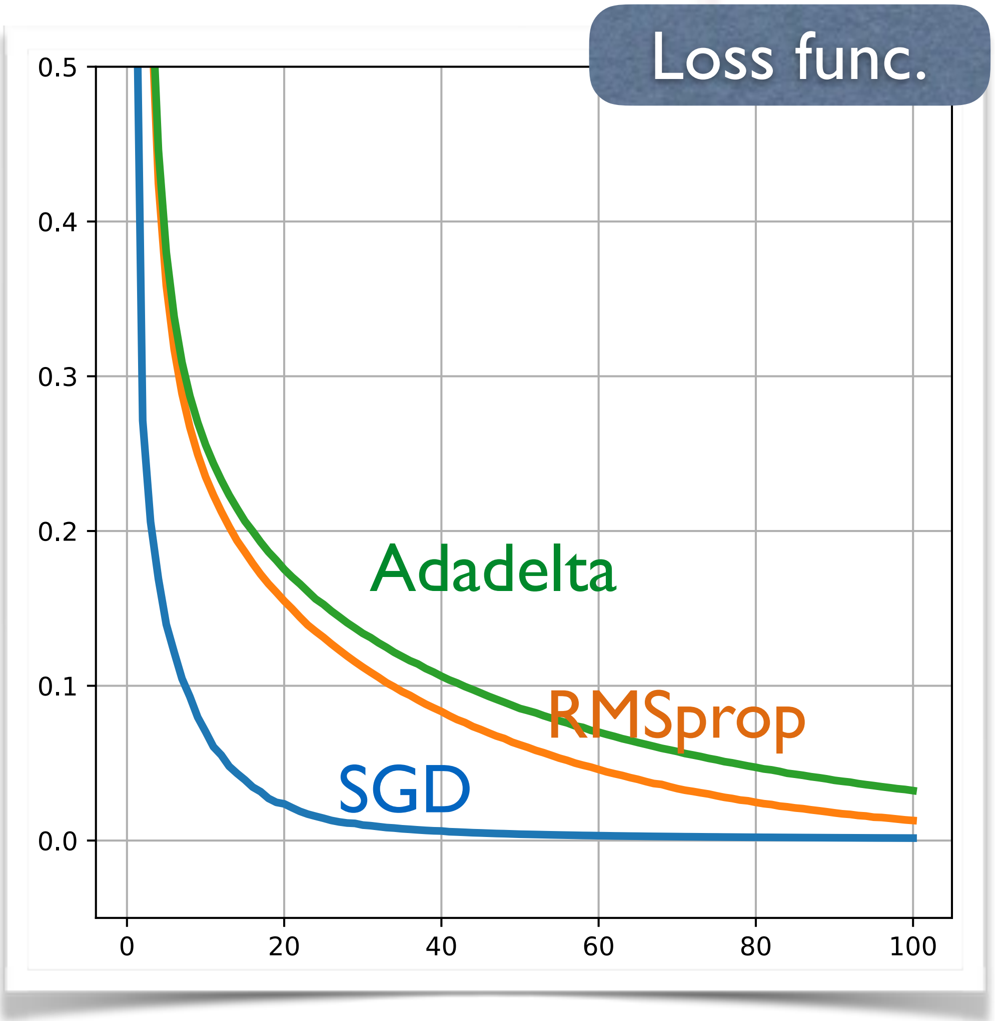
- **Adagrad**: applying regularization to the learning rate. Larger/smaller gradient would give smaller/larger learning rate.
- **Adadelta** : extended Adagrad with simplification and reduced the dependence to the global learning rate.
- **RMSprop**: a kind of variation of Adagrad and regularization with RMS of gradient. Good for large variant case.
- **Adam**: a kind of variation of RMSprop + momentum. Combining the good features of Adagrad and RMSprop.
- **Adamax**: variation of Adam, with simplified learning rate regularization formula.
- **Nadam**: Adam + Nesterov momentum.

DIFFERENT TRAINING ALGORITHM? (III)

- In general SGD is slower but very robust with good parameters.
- If you want a quicker converge with a complex network, those algorithms with adaptive learning can be better.
- *With our simple network SGD actually performs very well!*

```
. . . . . .
m1.compile(loss='categorical_crossentropy',
           optimizer=SGD(lr=2.0))

m2 = clone_model(m1)
m2.compile(loss='categorical_crossentropy',
           optimizer=RMSprop())

m3 = clone_model(m1)
m3.compile(loss='categorical_crossentropy',
           optimizer=Adadelta())
. . . . . .
```

l303-example-06c.py (partial)

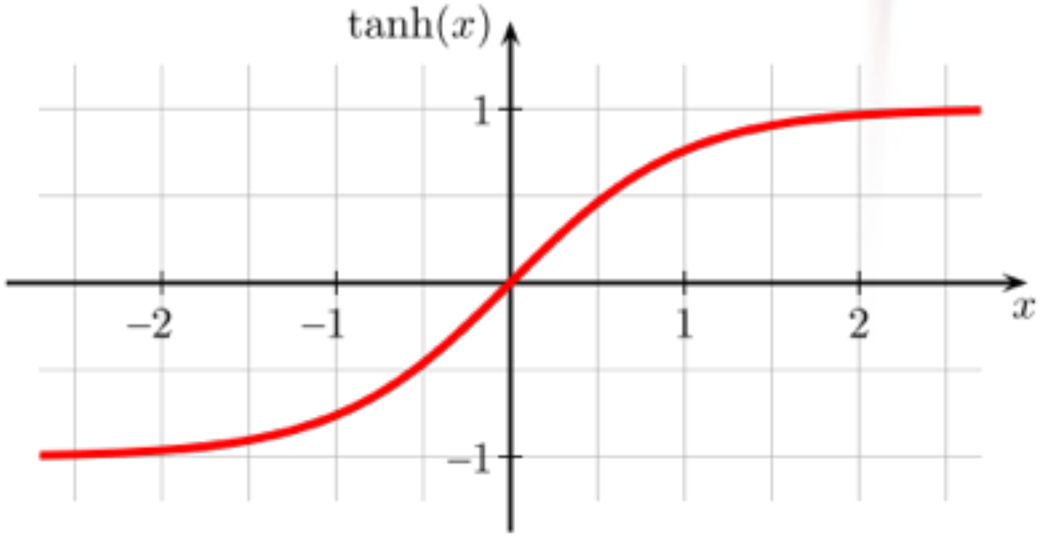


DIFFERENT ACTIVATION FUNCTION?

- Up to now we are mostly using the sigmoid function as our activation. The only exception is the output layer, where a softmax function has been introduced.
- A different choice is the **hyperbolic tangent**. It is very close to the sigmoid function but with –1 as the non-active value instead of zero:

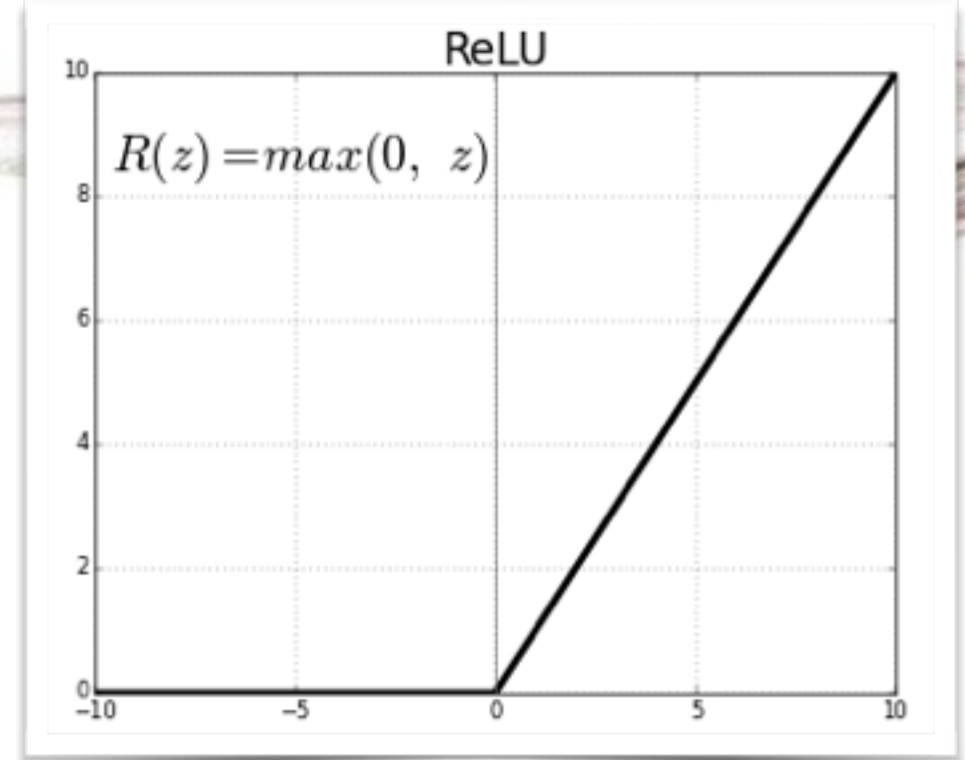
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

$$\sigma(z) = \frac{1 + \tanh(z/2)}{2}$$



- Using hyperbolic tangent requires a slightly different scale since the out range becomes [–1, +1]. Some studies suggest tanh can have a better performance in some of the cases since it has a symmetric response.

DIFFERENT ACTIVATION FUNCTION? (II)



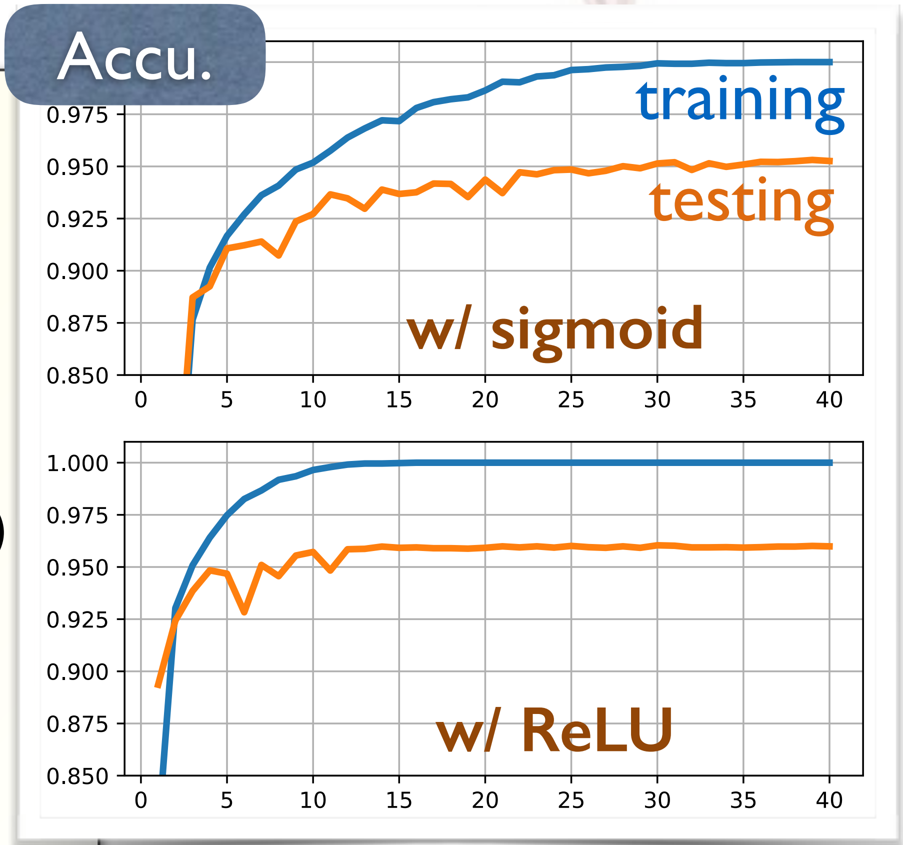
- In fact, the most common selection of activation function in modern network is the **rectified linear unit "ReLU"** (*not the sigmoid function!*), and it looks like this:
- Obviously this is very different from the sigmoid or tanh! Why this works better than the classical choices?
- One obvious feature is that the **gradient will not vanish with large input z**! This will not slow down the training speed as usually happening for the sigmoid-like functions.
- Another good feature is the ReLU function can **"switch-off" subset of the neurons** with an output zero. This can reduce the overtraining issue. But it might be hard to "switch-on" those neurons again.

DIFFERENT ACTIVATION FUNCTION? (III)

- Let's try to compare ReLU and sigmoid activations, but with a much larger/complicated network of **768-256-256-10** structure.
- See how good we can reach within 40 epochs of training:

```
m1 = Sequential()
m1.add(Reshape((784,), input_shape=(28,28)))
m1.add(Dense(256, activation='sigmoid'))
m1.add(Dense(256, activation='sigmoid'))
m1.add(Dense(10, activation='softmax'))
m1.compile(loss='categorical_crossentropy',
    optimizer=SGD(lr=1.0), metrics=['accuracy'])

m2 = Sequential()
m2.add(Reshape((784,), input_shape=(28,28)))
m2.add(Dense(256, activation='relu'))
m2.add(Dense(256, activation='relu'))
m2.add(Dense(10, activation='softmax'))
m2.compile(loss='categorical_crossentropy',
    optimizer=SGD(lr=0.2), metrics=['accuracy'])
```

l303-example-06d.py (partial)



It improves!

LARGER/DEEPER NETWORK?

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

```
. . . . . . .
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=Adadelta(), metrics=['accuracy'])

model.fit(x_train, y_train, epochs=20, batch_size=128)
. . . . . . .
```

l303-example-07.py (partial)

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

LARGER/DEEPER NETWORK? (II)

- We can have a great performance of 98.5% accuracy which matches to our performance from SVM with Gaussian kernel!

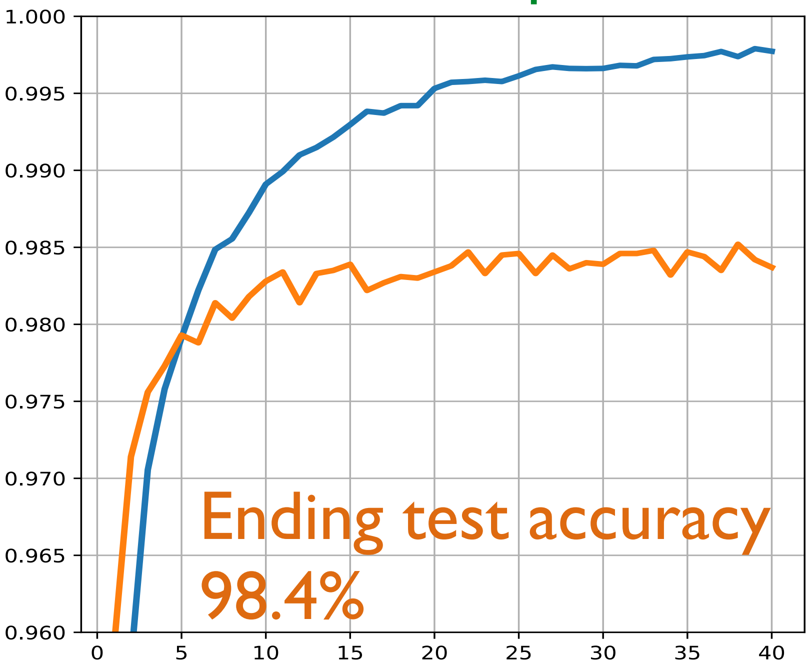
```
. . . . . .
Epoch 20/20
60000/60000 [===========] - 5s 91us/step - loss: 0.0065 - acc: 0.9979
Performance (training)
Loss: 0.00114, Acc: 0.99987
Performance (testing)
Loss: 0.06721, Acc: 0.98490
```

- **Can we do *even better* with a deeper network?** e.g. adding a couple of big layers or many smaller 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)

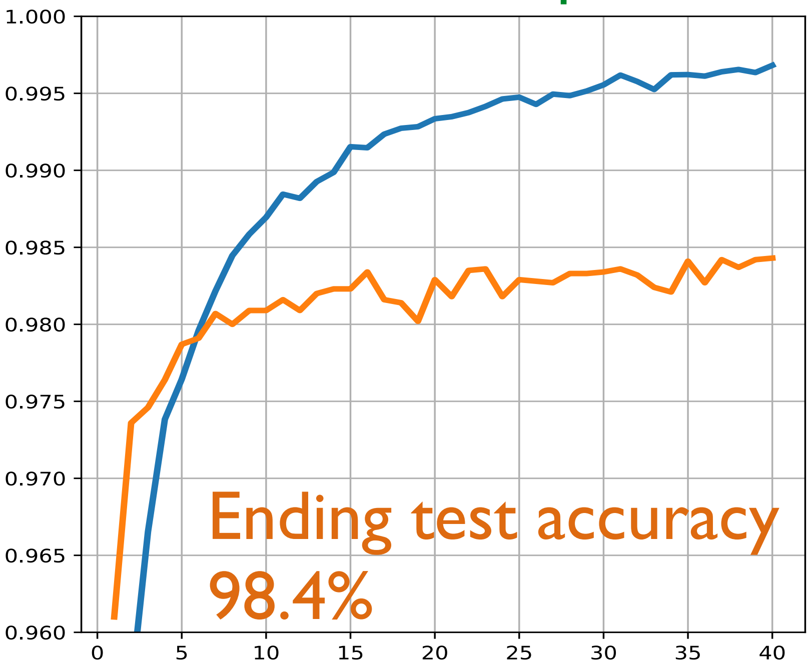
- Let's try several different network models and see if we can have interesting findings?

784-(256x2)-10
train for 40 epochs



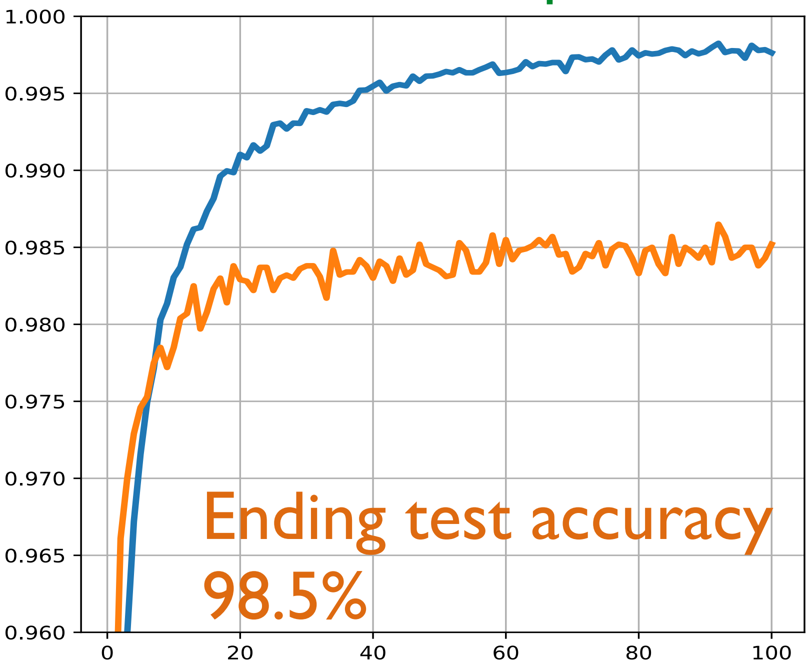
Ending test accuracy
98.4%

784-(256x4)-10
train for 40 epochs



Ending test accuracy
98.4%

784-(256x8)-10
train for 100 epochs



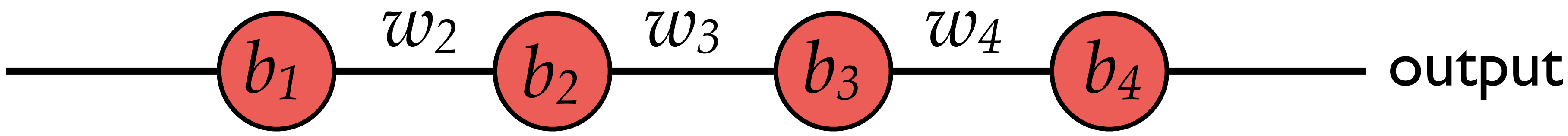
Ending test accuracy
98.5%

A more complex network does require a longer training time.
Why it is difficult to train a deeper network?

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 σ′(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 b1 will be much slower than b4.

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

- And this is not the full story. The small σ′(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 one many other factor (and hyperparameters) as well. Although it sounds difficult to get a decent deep network trained, but one can, still get a better performing deep network, **with a different network structure**.

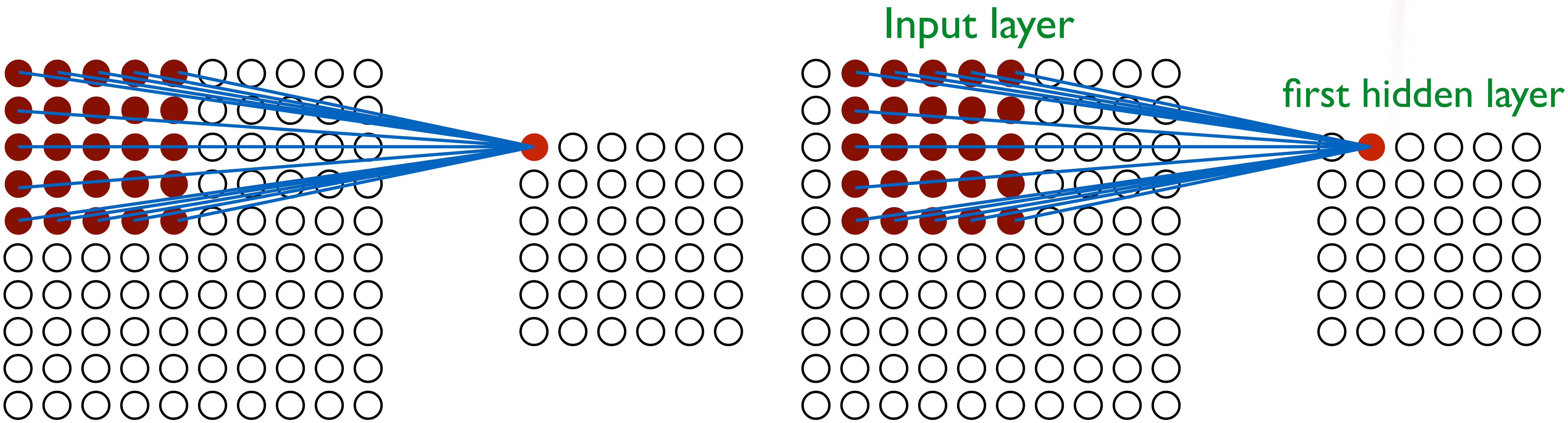
Here comes the Convolutional Neural Network…

CONVOLUTIONAL 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…

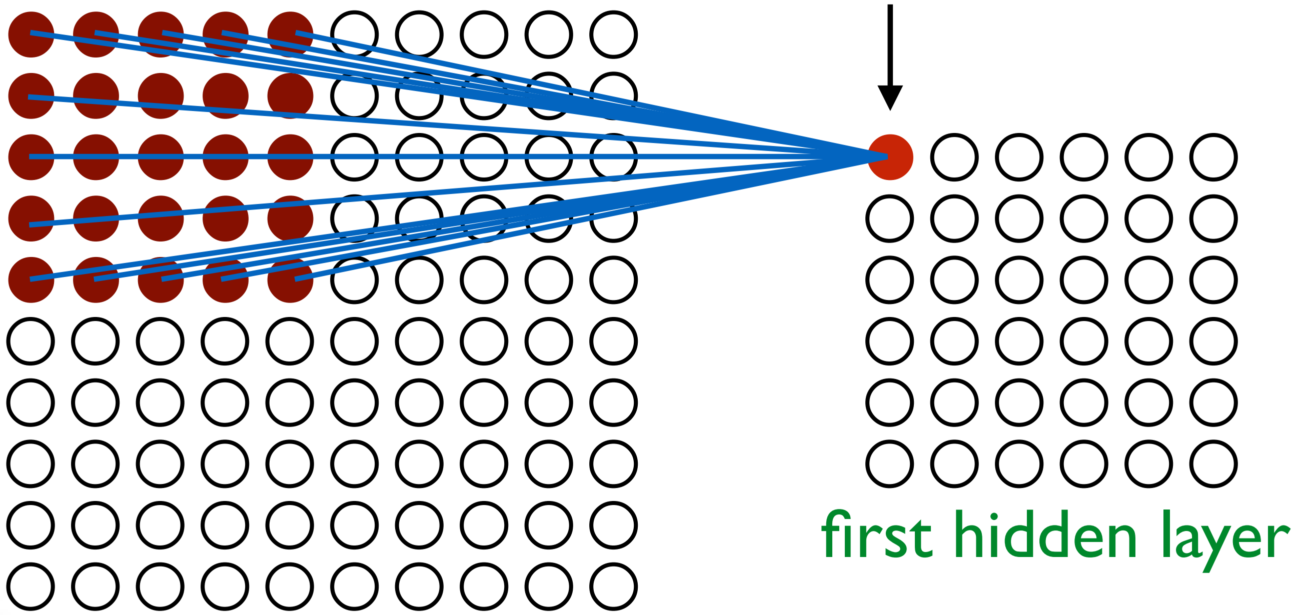


If we have 28×28 as the input image, and with a 5×5 local representative field, the first hidden layer will be 24×24.

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.

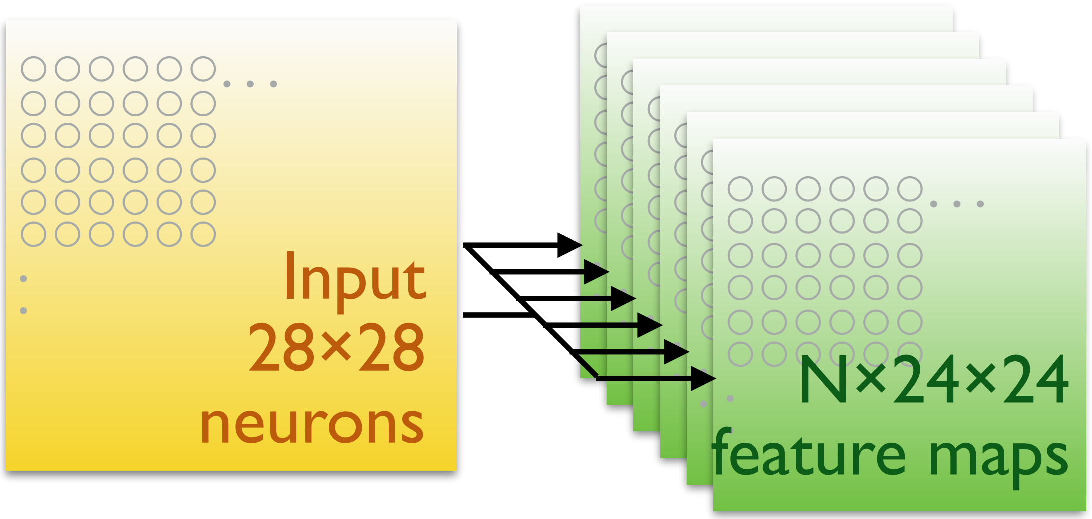
$$\sigma\left(b+\sum_{i=0}^{4}\sum_{j=0}^{4} w_{ij} v_{m+i,n+j}\right)$$

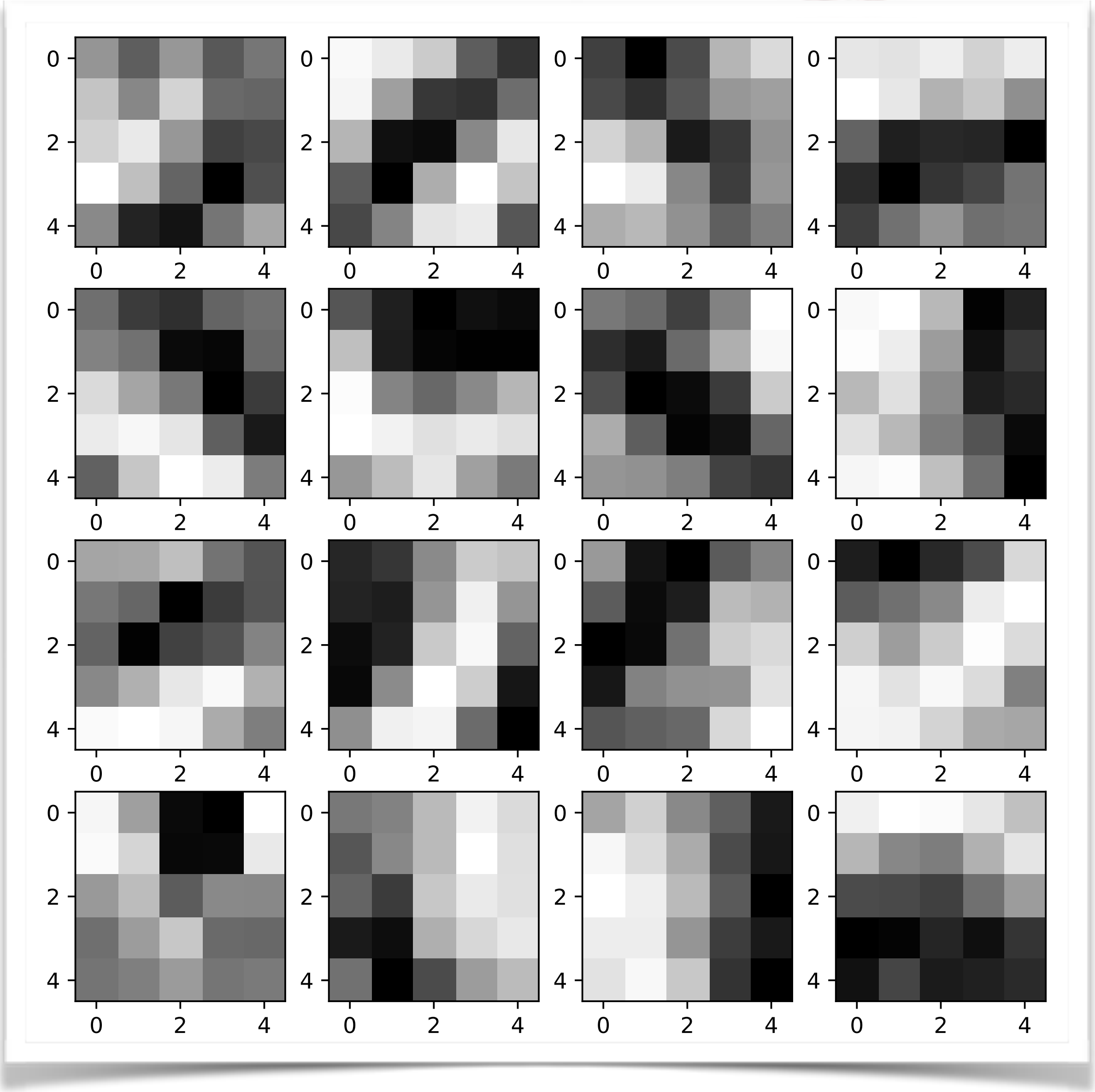


  - This means all of the neurons of the hidden layer can *detect exactly the same feature*.
  - The map from the input layer to the hidden layer is usually called a **feature map**.
  - A feature map only keep **25 weights and 1 bias**!
  - The shared weights/bias are often said to define a **kernel** or a **filter**.

FEATURE MAPS

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

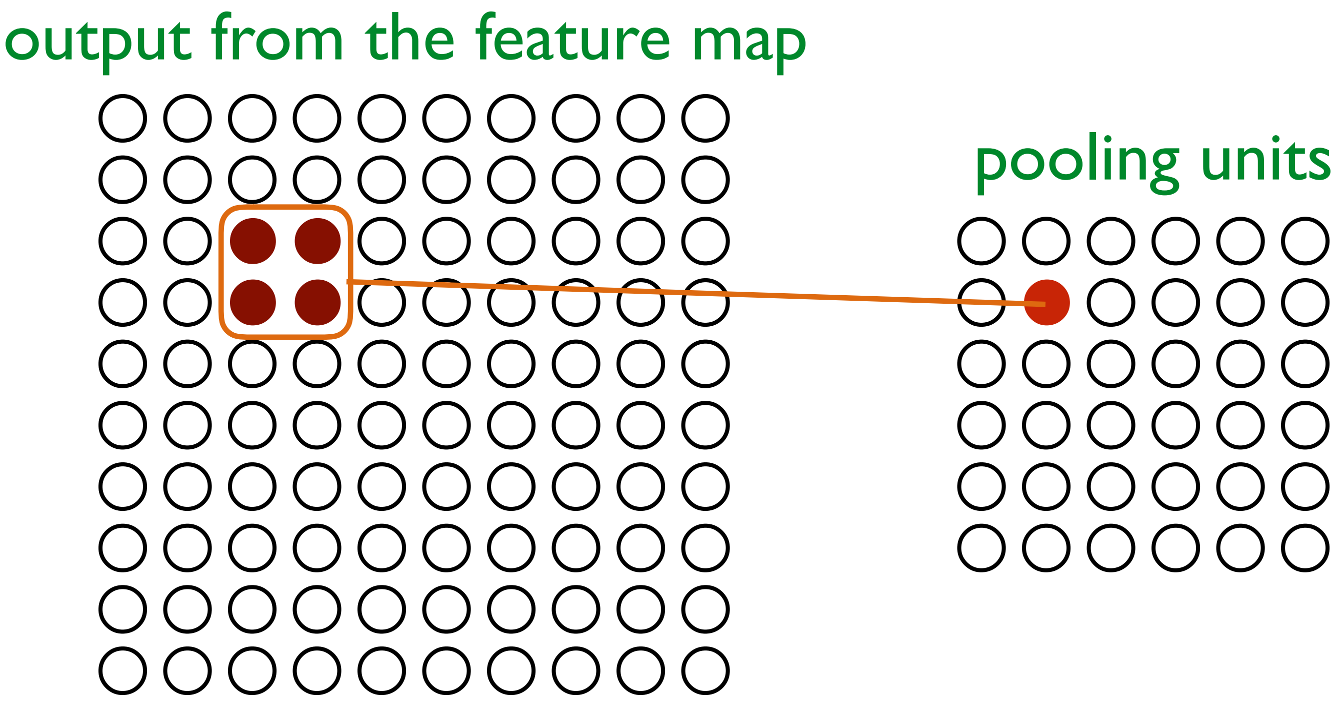




- For example here are the trained 16 feature maps (or kernels/filters) in the next example.
- Basically each map supposes to pick up a different feature from the input images!

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:

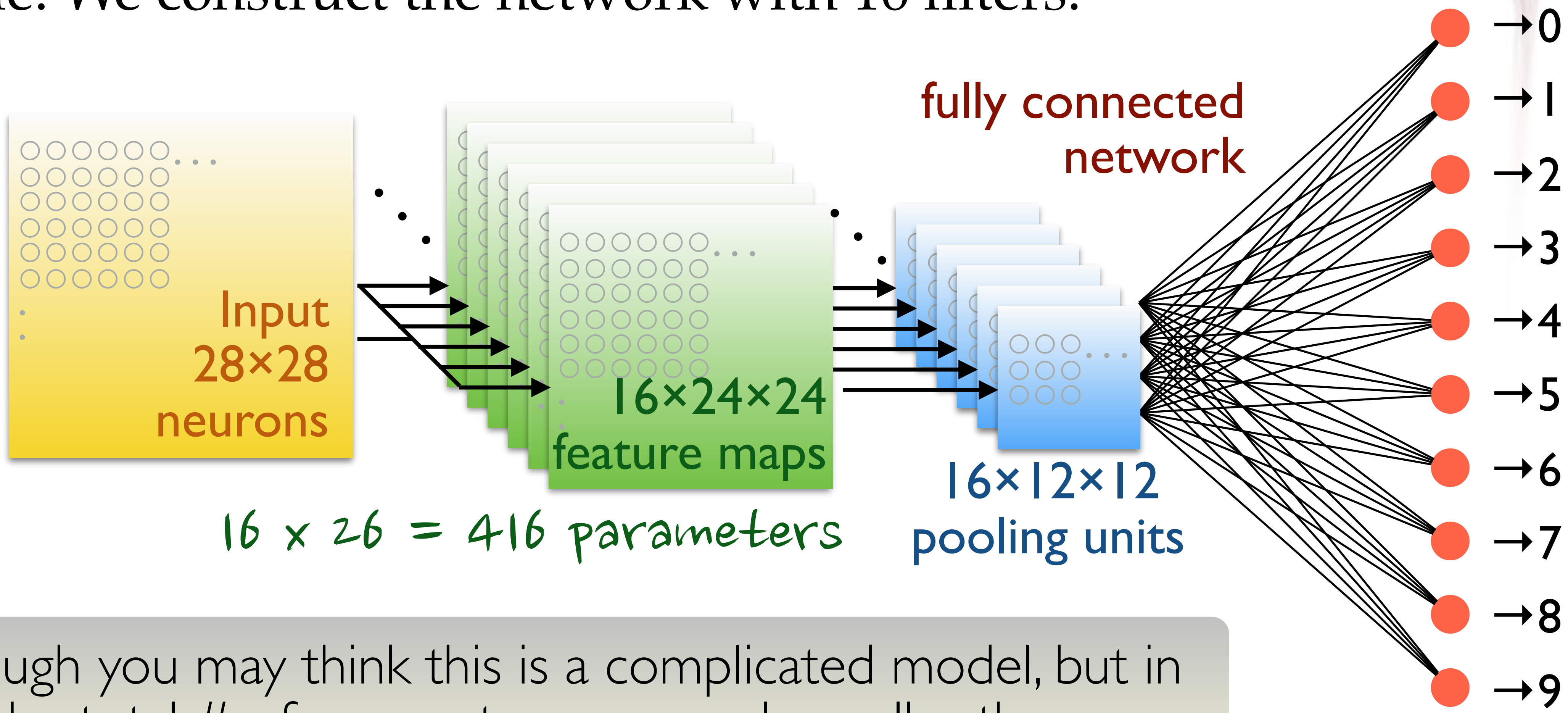


Usually this is applied to each feature map output layer

- **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: 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 the network with 16 filters:



16 x 26 = 416 parameters

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)

- Easy implementation with Keras:

Just the model discussed in the previous page!

```
. . . . . .
from keras.models import Sequential
from keras.layers import *
from keras.optimizers import Adadelta

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=Adadelta(),
              metrics=['accuracy'])
. . . . . .
```

⇑ 5x5 convolutional layer

⇑ 2x2 pooling layer

l303-example-08.py (partial)

PUT ALL TOGETHER (III)

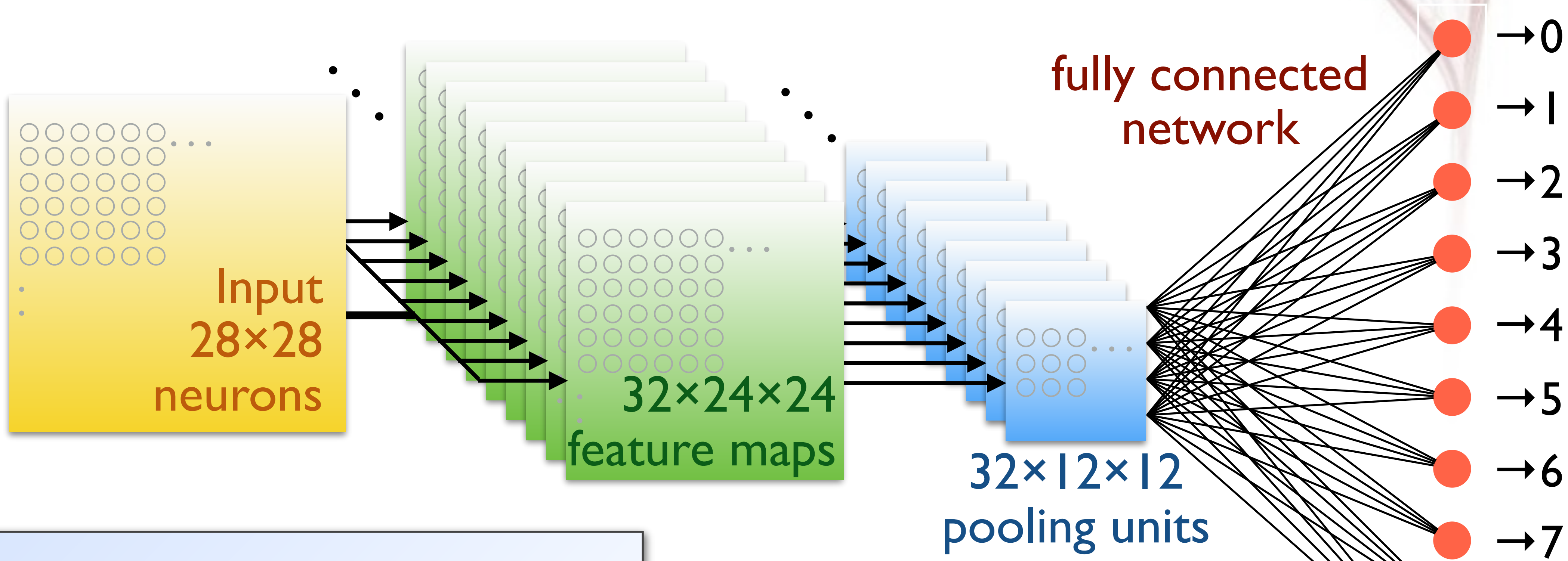
- And we can reach a very good performance already:

```
. . . . . .
Epoch 20/20
60000/60000 [===========] 13s 217us/step - loss: 0.0363 - acc: 0.9890
- val_loss: 0.0371 - val_acc: 0.9874
Performance (training)
Loss: 0.02537, Acc: 0.99267
Performance (testing)
Loss: 0.03712, Acc: 0.98740
```

- A testing accuracy of 98.7% reached, only 126 images are mis-identified. Remember we only put a layer of convolutional network and # of parameters is only 1/28 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 double the feature maps? Can we improve the model?

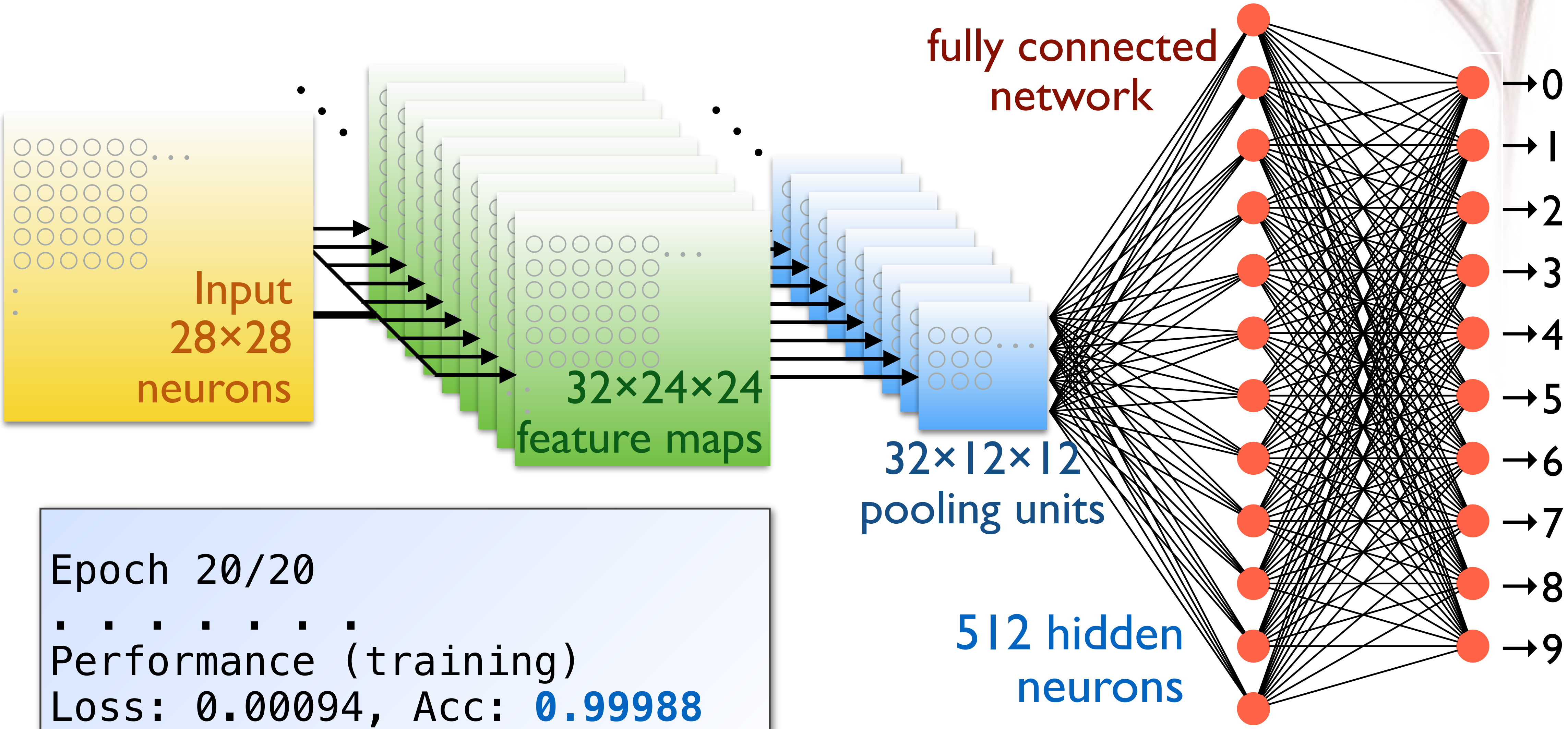


```
Epoch 20/20
. . . . . . .
Performance (training)
Loss: 0.01816, Acc: 0.99518
Performance (testing)
Loss: 0.03244, Acc: 0.98900
```

- Now we reached **98.9%** test accuracy, only 110 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: 0.00094, Acc: 0.99988
Performance (testing)
Loss: 0.02896, Acc: 0.99230
```

– Now we go beyond **99.2%**!

DOUBLED LAYERS!

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

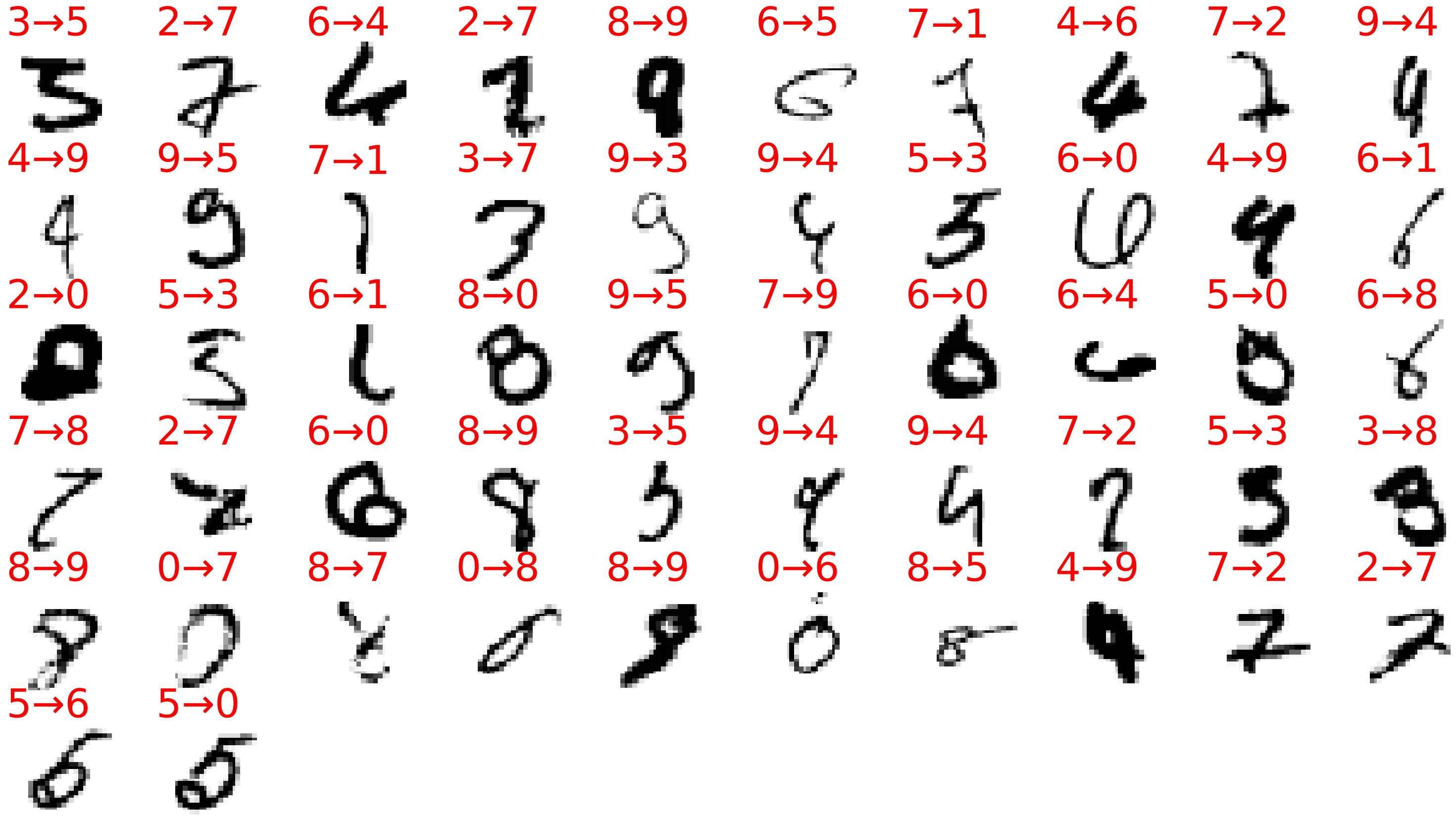
```
. . . . . .
model = Sequential()
model.add(Reshape((28,28,1), input_shape=(28,28)))
model.add(Conv2D(32, kernel_size=(5,5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(32, kernel_size=(5,5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())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

1303-example-08a.py (partial)

```
Performance (training)
Loss: 0.00167, Acc: 0.99960
Performance (testing)
Loss: 0.01988, Acc: 0.99480
```

– Now we go beyond **99.4%**!

DOUBLED LAYERS! (II)

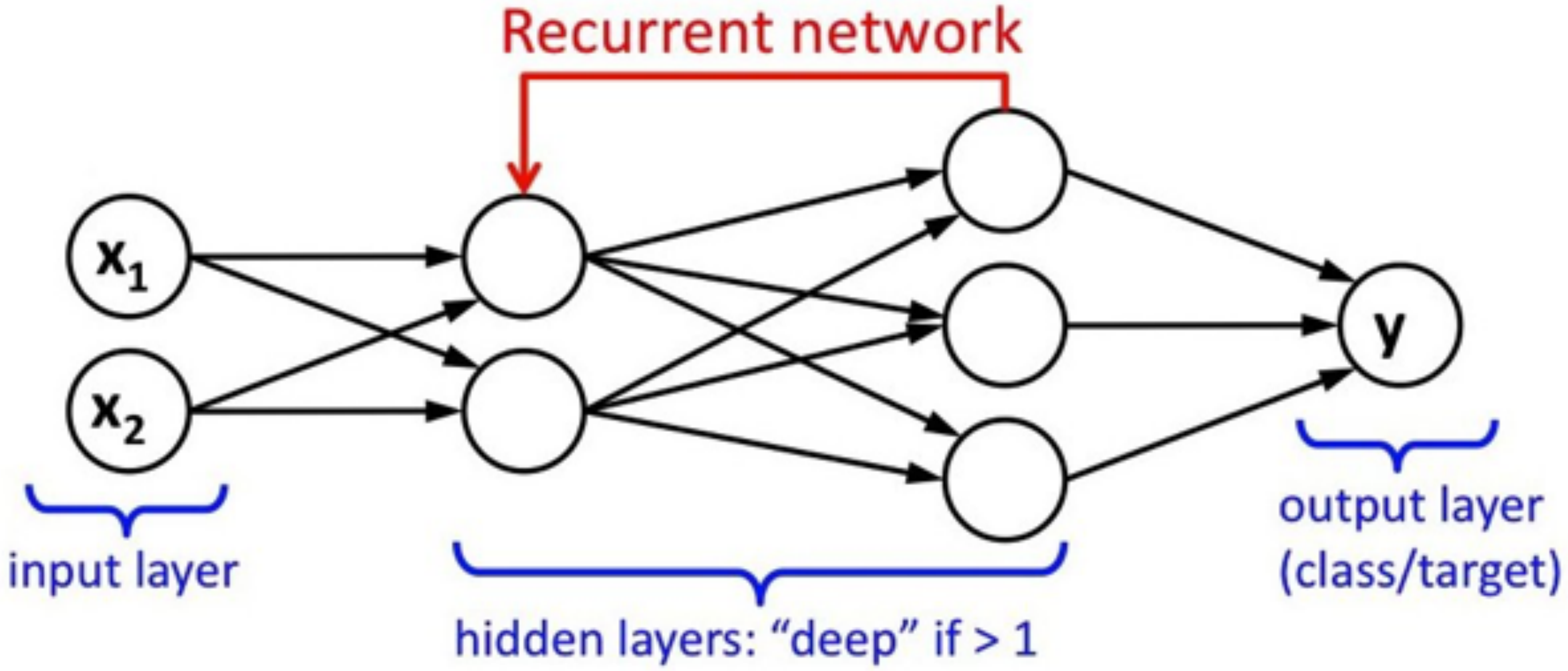


- Now we only have 52 wrongly tagged images (0.52% failed).
- Some of them are also difficult for real humans!
- Remember the best trained network (world record) is with 0.21% failure rate. Still rooms to be improved!

In any case convolutional neural network is a kind of deep network good for image recognition!

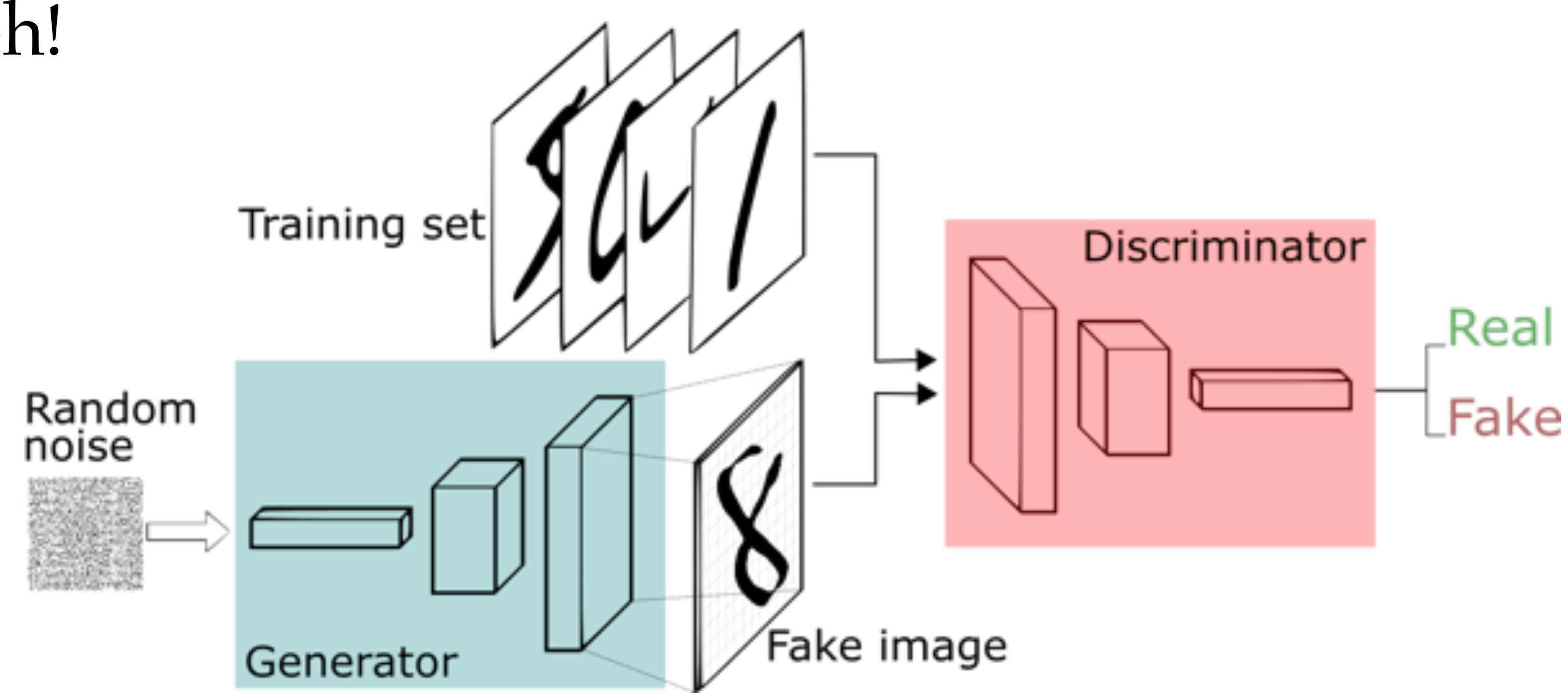
OTHER DEEP NETWORKS & IDEAS

- **Recurrent neural network (RNN):**
  - Up to now our network has a fixed flow throughout the training, but what will happen if we allow the network to vary itself along with time sequence?
  - Unlike feedforward neural network, RNN can use their internal state ("memory") to process a sequence of inputs. This gives RNN a good approach to the unsegmented data, for example, language/speech recognition.



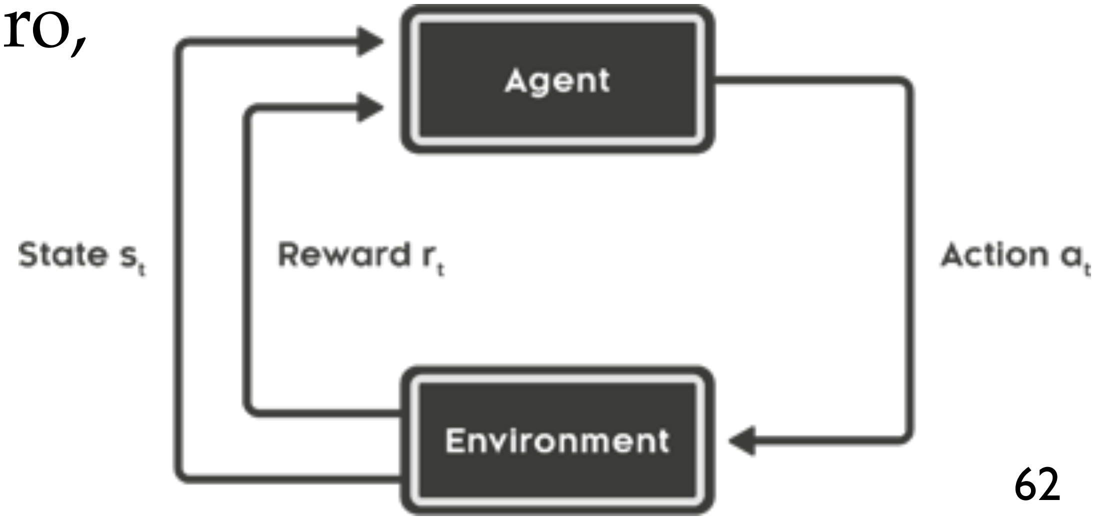
OTHER DEEP NETWORKS & IDEAS (II)

- **Generative adversarial network (GAN):**
  - The basic structure of GAN is to have two network "fighting" with each other: one is to find "fake" images out of the pool, another one is to generate fake images.
  - Once it has been trained, you can use the generator to produce lots of "nearly true" fake images, e.g. photo of a person who never exists in the real world, or convert your doodle to a fancy graph!



OTHER DEEP NETWORKS & IDEAS (III)

- **Reinforcement Learning (RL):**
  - In our example network, the required responses of our model are relatively simple (just which digit, 0-9). But in many problems, for example, playing chess, this is not a simple task as no clear classification of good/bad labels.
  - Then the reinforcement learning is a kind of idea to build the environment for your program to learn how to survive by itself (only give it a goal to reach, e.g. beating the opponent, getting higher scores etc). Let the environment to be the teacher.
  - A famous example is the AlphaGoZero, which is trained without any prior knowledge of Go, but just let to figure out how to play Go by itself!



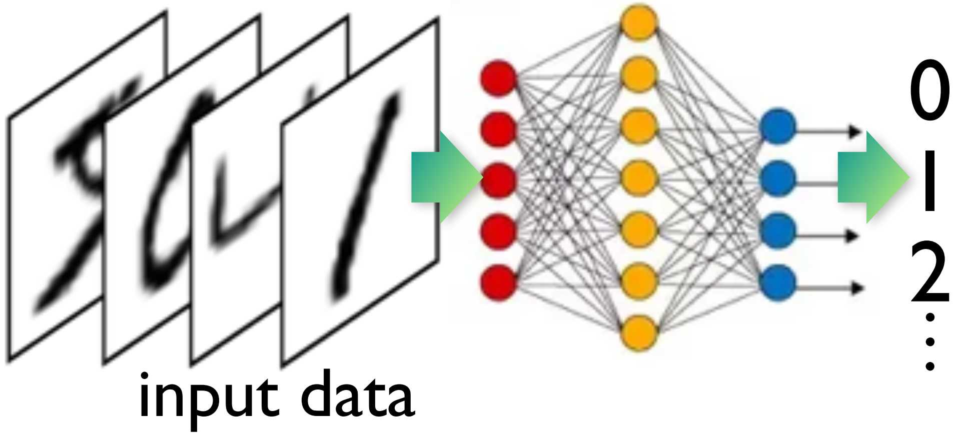
FINAL COMMMENT: PHYSICIST'S ML

- Physicists also use a lot of ML to solve the problems found in the experiments or theories. But what are the core difference between a physicists' problem and a generic problem?
- Surely I cannot comment for everyone — but at least I can say the *particle physicists* have a rather different prospective regarding ML tools comparing to generic users.
- The key point of particle physicists' ML is about its **statistical interpretation**: we do not just concern about if your ML tool is working or not, we also worry about *how correct it performs*. e.g. even if you know the accuracy of your network is **99.5%**, we also want to know the error of this value, e.g. **99.5±0.XX%**, and also the performance difference between the ideal situation and and real application.

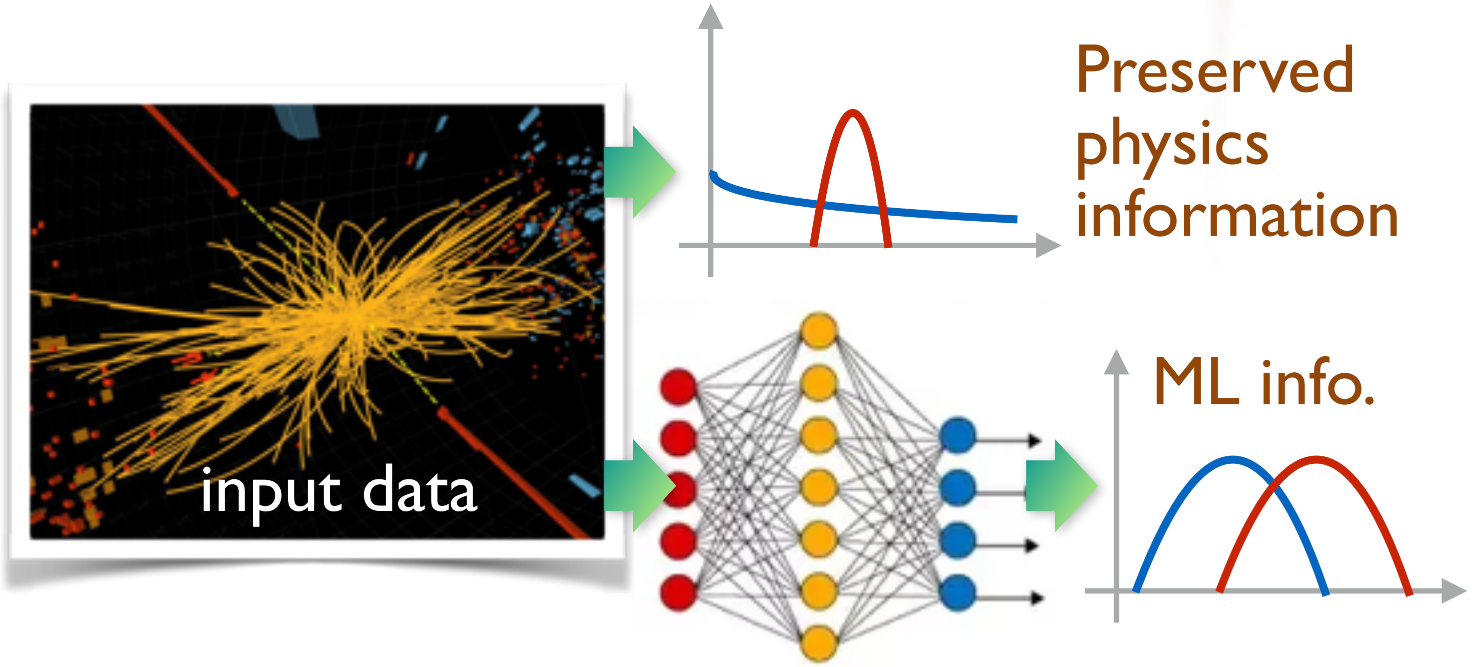
FINAL COMMMENT: PHYSICIST'S ML (II)

- So unlike the generic problem (e.g. image recognition, etc.), we need to find a way to preserve the information and still use it to present physics results, instead of just dump everything into the network. i.e.

Generic ML Solution



(Particle) Physics ML Solution



So the (particle) physics ML solution is generally weaker than the generic ML due to lack of key information in ML. But we use it to do further **statistical analysis** afterwards.

HANDS-ON SESSION

- **Practice 01:**
  - Trial #1:
    In the `l303-example-04a.py` we have tried a L2 regularization method to reduce the overtraining issue. What will be the situation if we switched to L1 regularization?
  - Trial #2:
    In the `l303-example-08.py` configuration, we are using 5×5 block in our convolutional layer. What will be the performance if we take a different size, for example 3×3 or 7×7?

HANDS-ON SESSION

- **Practice 02:**
  Up to now we are always using the same testing sample to measure the performance. But what will happen if we rotate our testing data and see how good we can still separate the handwriting digits?
- You can take one of the ending example, e.g. `l303-example-08.py` or `l303-example-08a.py`, train your network, but in the end use the rotated test sample to see the performance. The method/code to rotate your images can be found in `l303-example-05.py`.

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