

The background of the top half of the slide is a complex, abstract fractal pattern in various shades of blue. It features intricate, self-similar structures that resemble organic forms like coral or snowflakes, as well as geometric patterns of concentric circles and radial lines. The overall effect is a dense, textured field of blue.

MODULE L7: NEURAL NETWORK TRAINING TRICKS II

INTRODUCTION TO COMPUTATIONAL PHYSICS

.....

Kai-Feng Chen
National Taiwan University

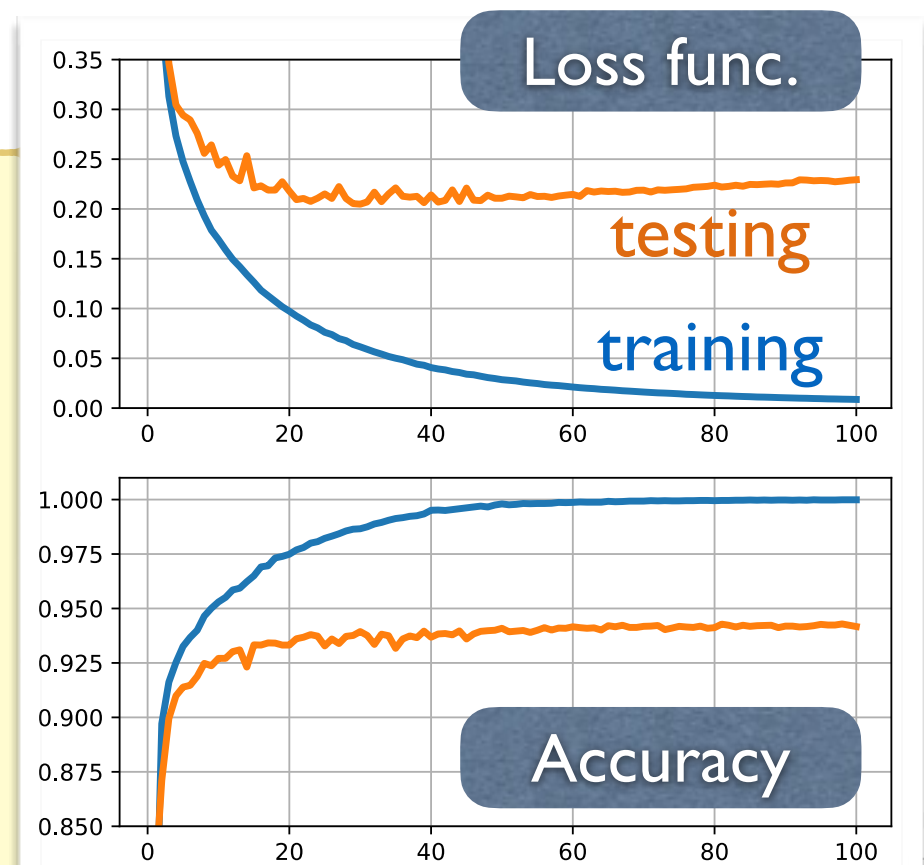
THE OVERTRAINING ISSUE

- ❖ We have touched slightly on this issue at one of the earlier modules. 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:

partial ex_ml7_1.py

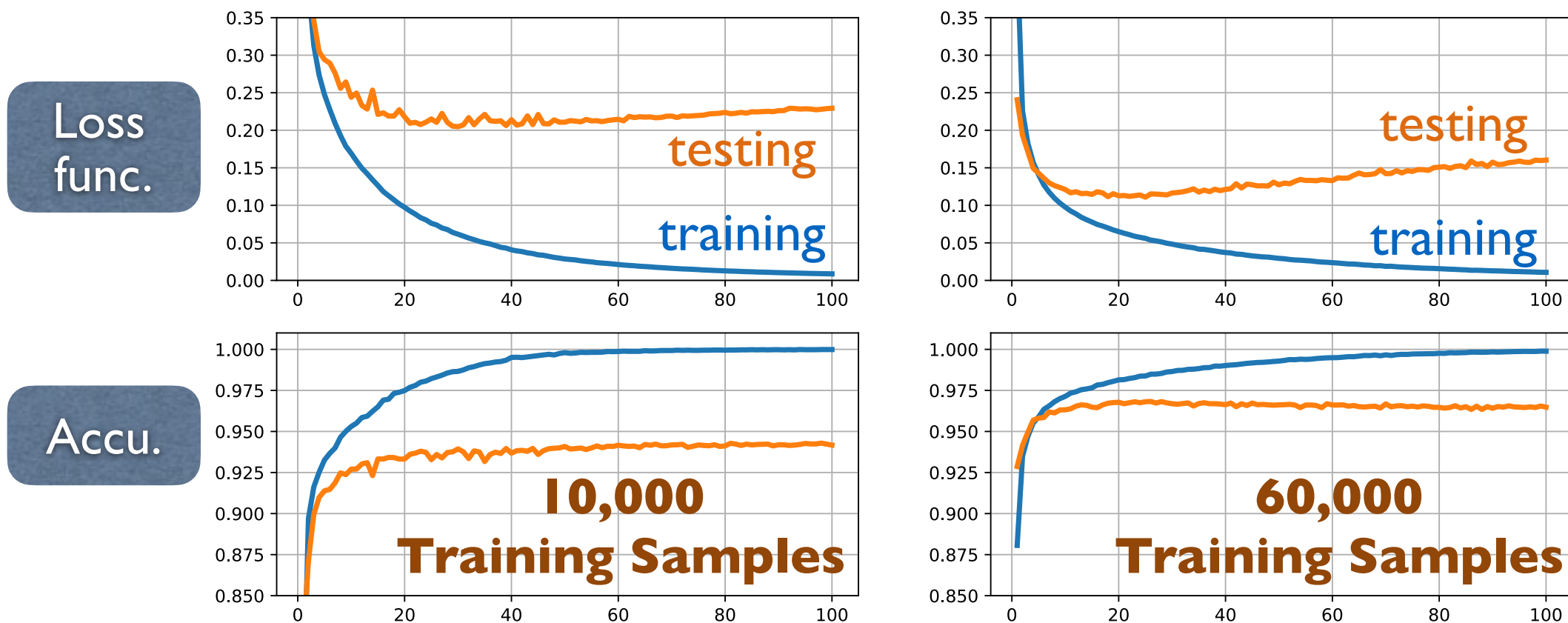
```
rec = model.fit(x_train, y_train, \
               epochs=100, batch_size=120, \
               validation_data=(x_test, y_test))

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



TRAINING DATA DOES MATTER

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



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

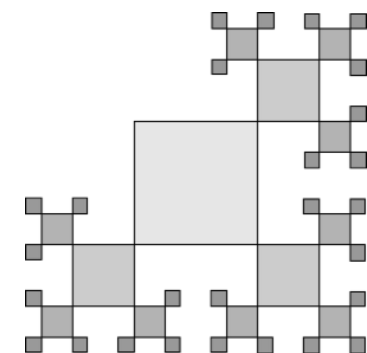
REGULARIZATION

- ❖ 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* ($\lambda > 0$) 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):

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

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

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

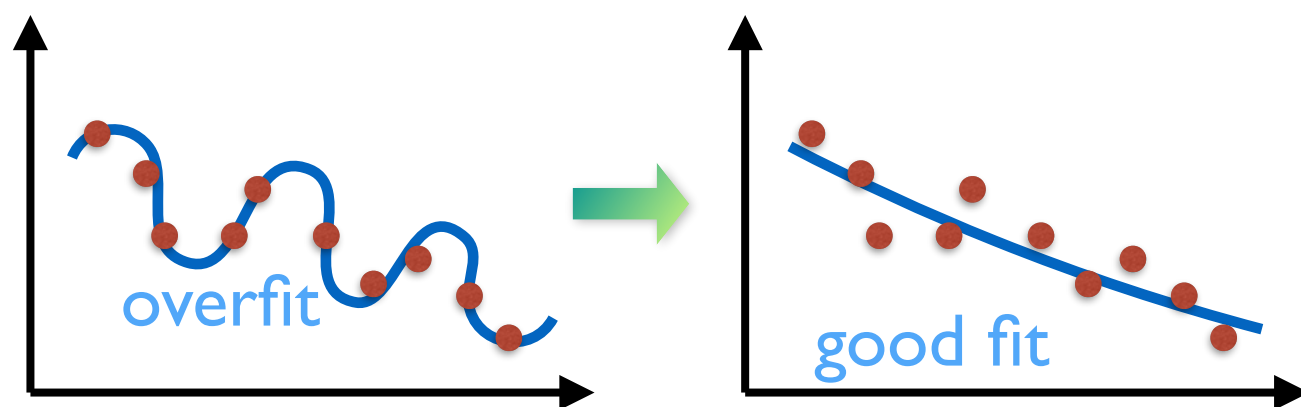


REGULARIZATION (II)

- ❖ 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:

$$f(x) = w_0 + w_1x + w_2x^2 + w_3x^2 + w_4x^4 + w_5x^5 + \dots$$

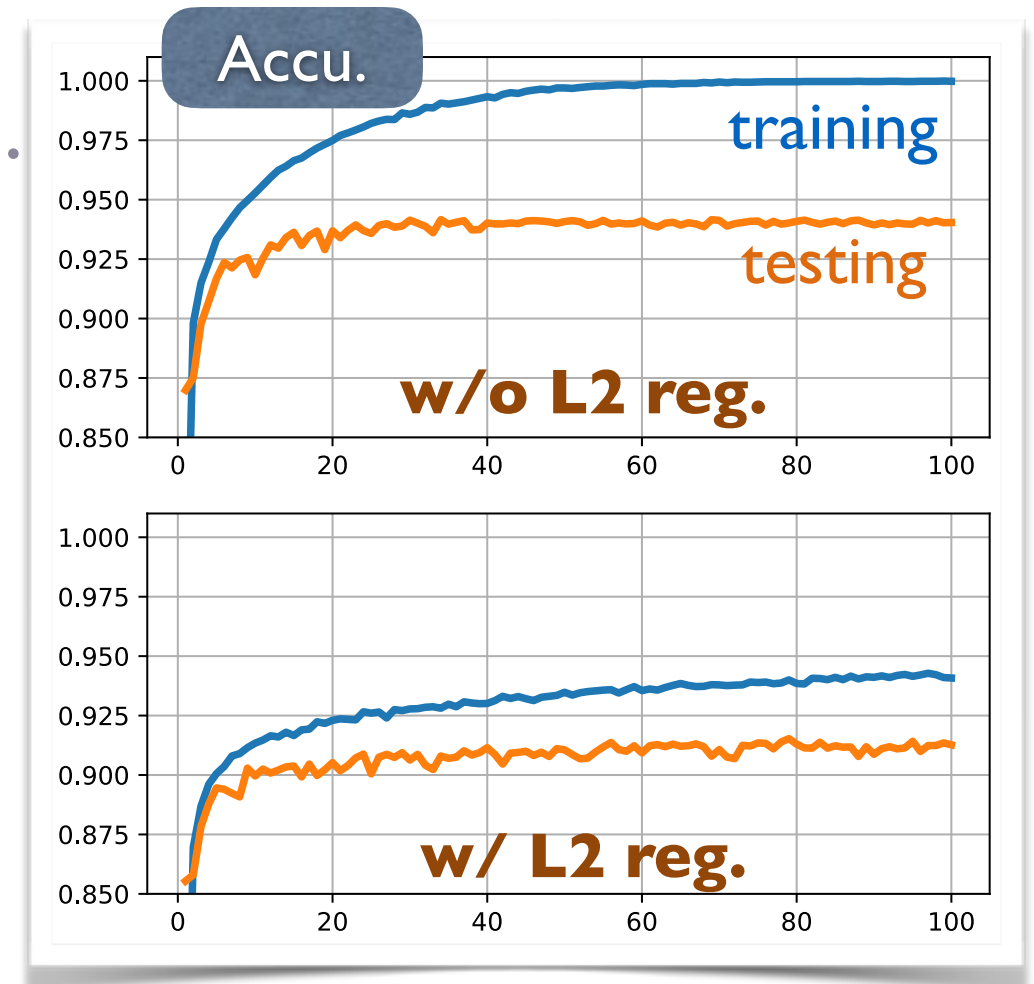
$$\longrightarrow f(x) = w_0 + w_1x + w_2x^2 + 0x^2 + 0x^4 + 0x^5 + \dots$$



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:

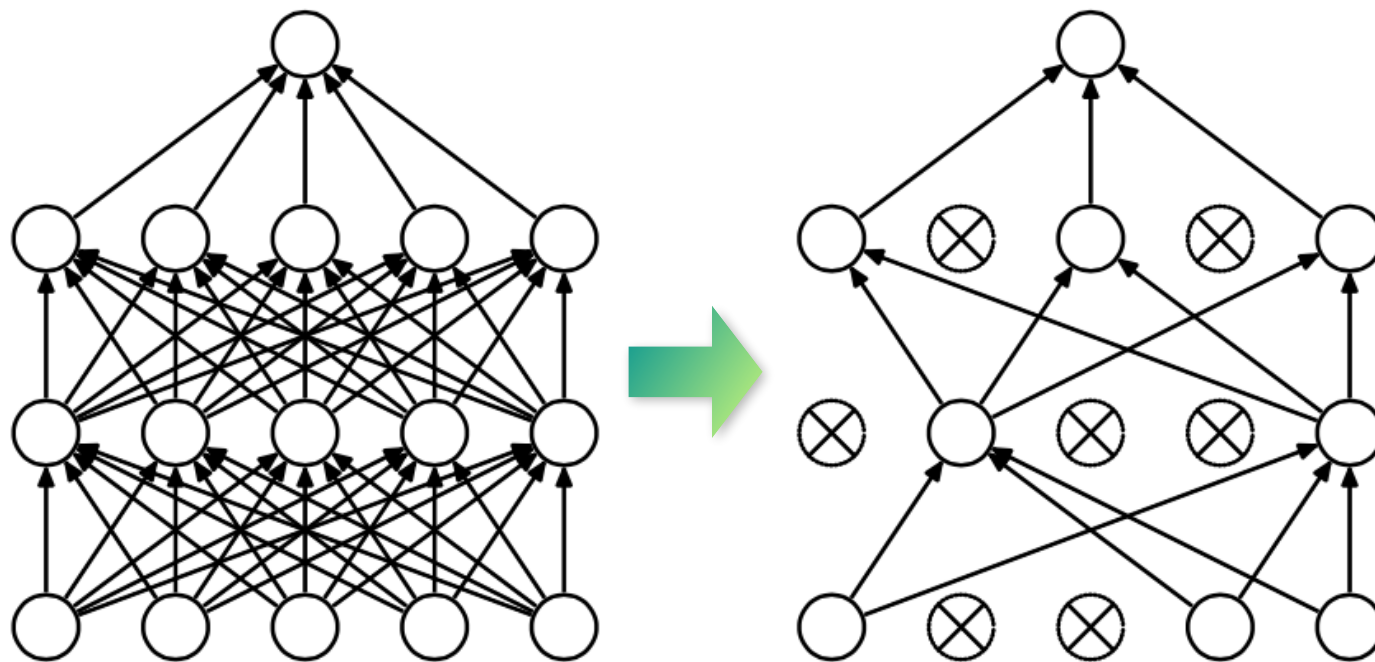


partial ex_ml7_1a.py

```
.....  
from tensorflow.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'])  
.....
```

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)

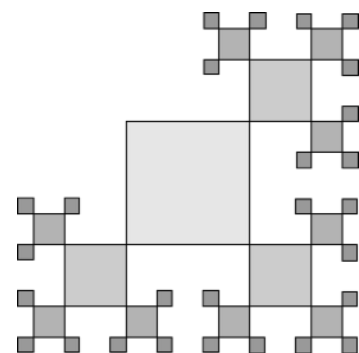
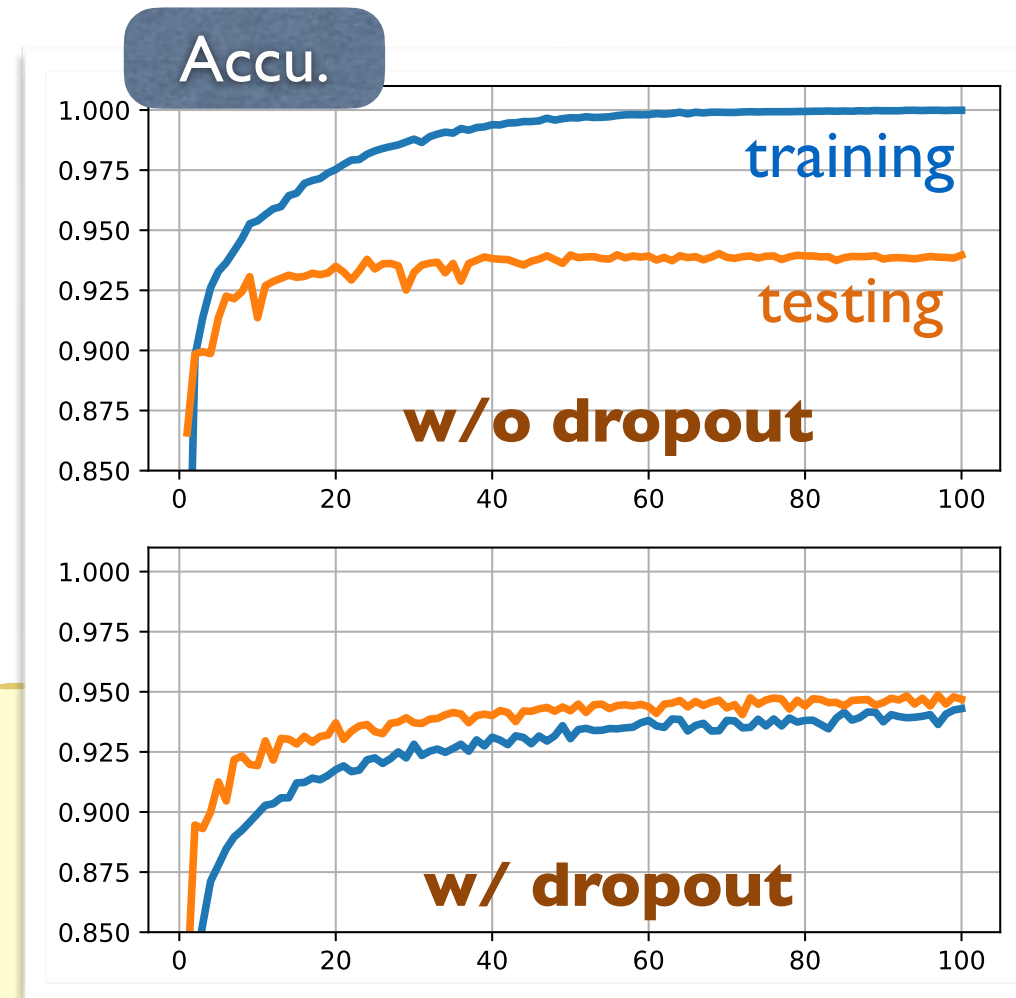
- ❖ And you can find that the dropout method is actually very helpful in terms of against overtraining (*even it is so simple!*):

Drop 20% of the inputs randomly

partial ex_ml7_1b.py

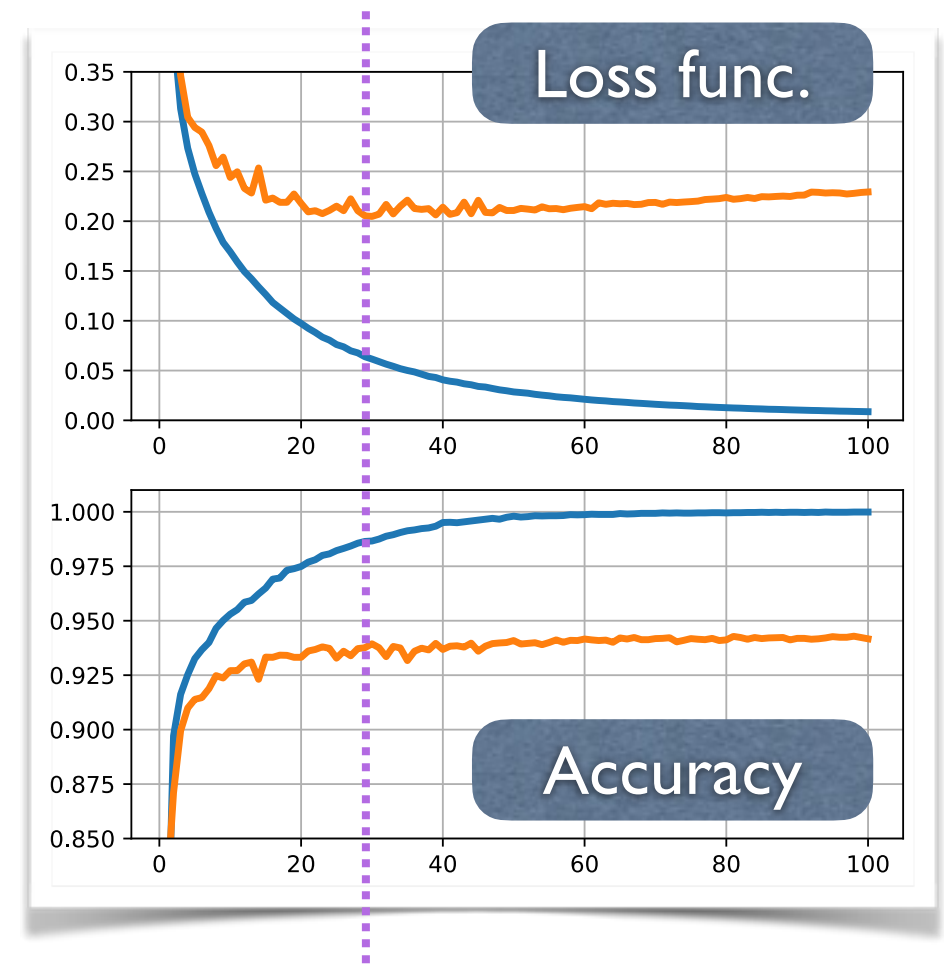
```
from tensorflow.keras.layers import Dropout

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'])
```

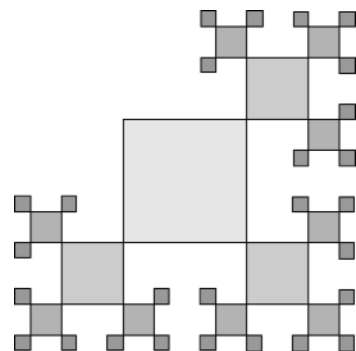


EARLY STOPPING

- ❖ 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:

partial ex_ml7_1c.py

```

. . . . .
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 tensorflow.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)))
. . . . .
```

↑ another
independent
validation
sample

EARLY STOPPING (III)

.....

- ❖ The learning process is stopped automatically after 18 epochs:

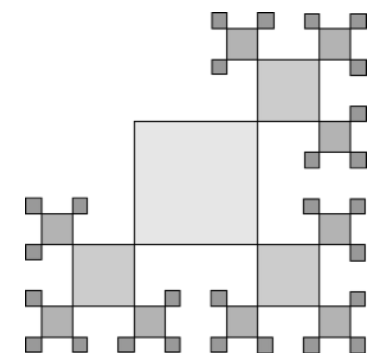
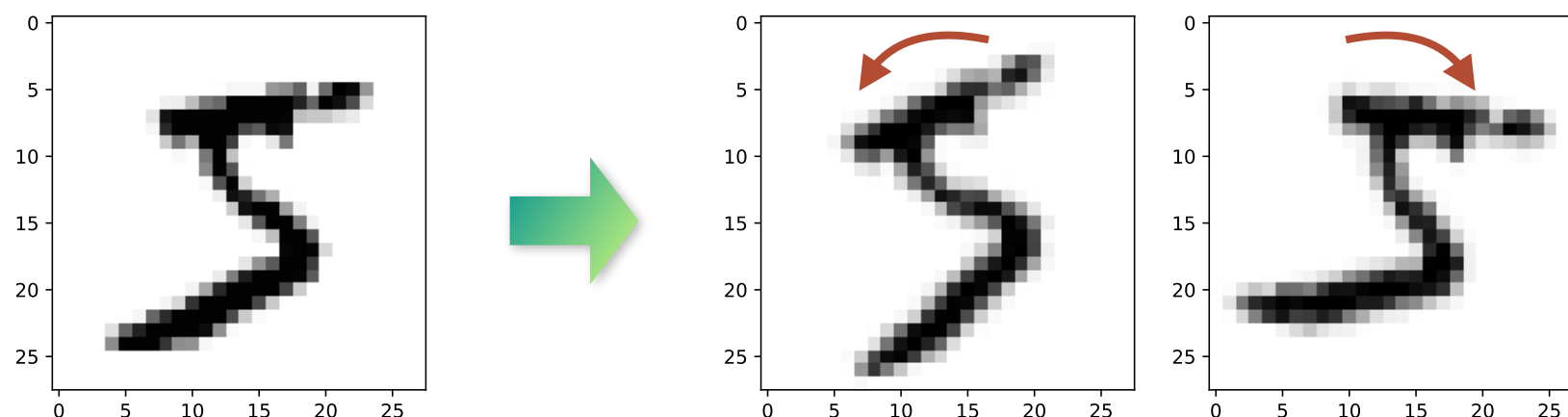
```
Epoch 1/100
84/84 [=====] - 1s 6ms/step - loss: 1.4000 - accuracy: 0.5901 -
val_loss: 0.4741 - val_accuracy: 0.8778

. . . . .
Epoch 18/100
84/84 [=====] - 0s 2ms/step - loss: 0.1154 - accuracy: 0.9703 -
val_loss: 0.2119 - val_accuracy: 0.9371
Performance (training)
313/313 [=====] - 0s 508us/step - loss: 0.1086 - accuracy: 0.9714
Loss: 0.10855, Acc: 0.97140
Performance (validation)
313/313 [=====] - 0s 527us/step
Loss: 0.21193, Acc: 0.93710
Performance (testing)
313/313 [=====] - 0s 500us/step
Loss: 0.22961, Acc: 0.92950
```

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.



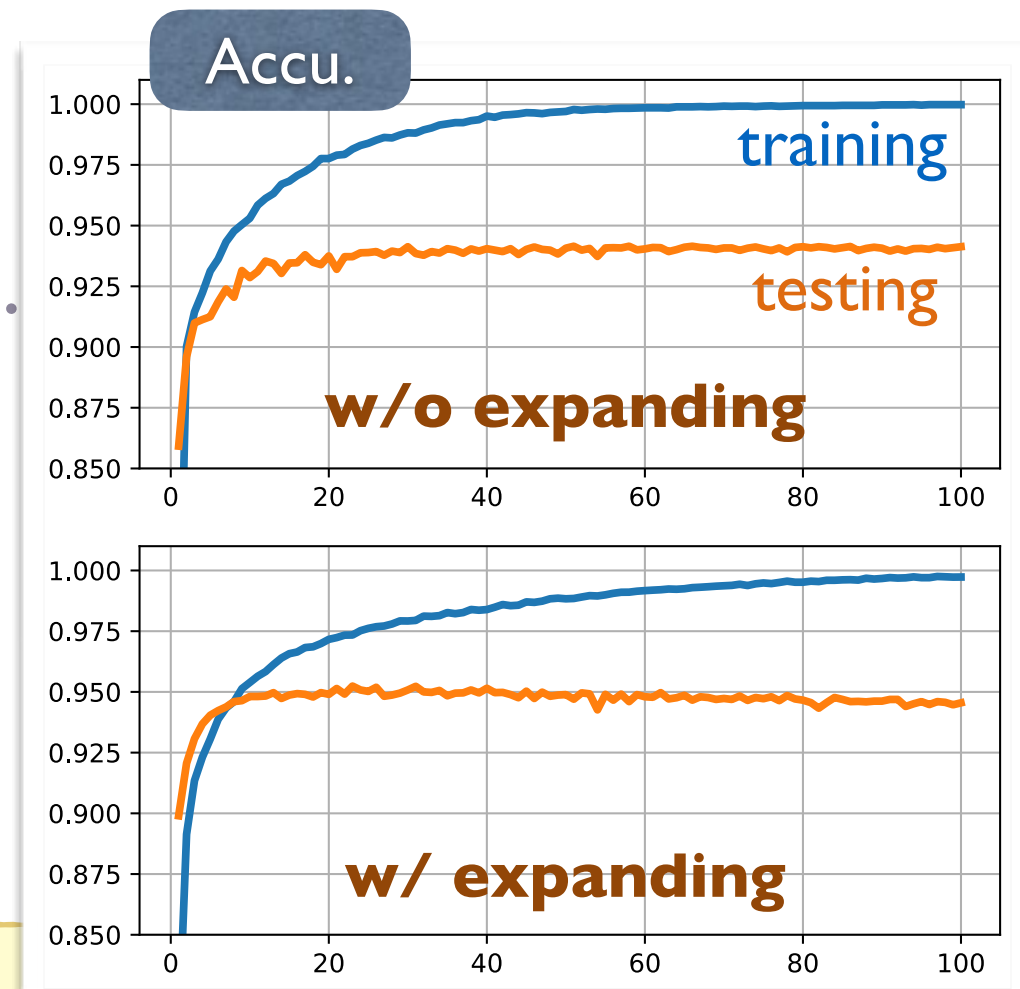
ARTIFICIAL DATA EXPANDING

- ❖ Let's **triple the training data** by randomly rotate the images either $+5^\circ \sim +25^\circ$, or $-5^\circ \sim -25^\circ$. Some positive effect found!

↓ Use the scikit-image tool

```
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))
```



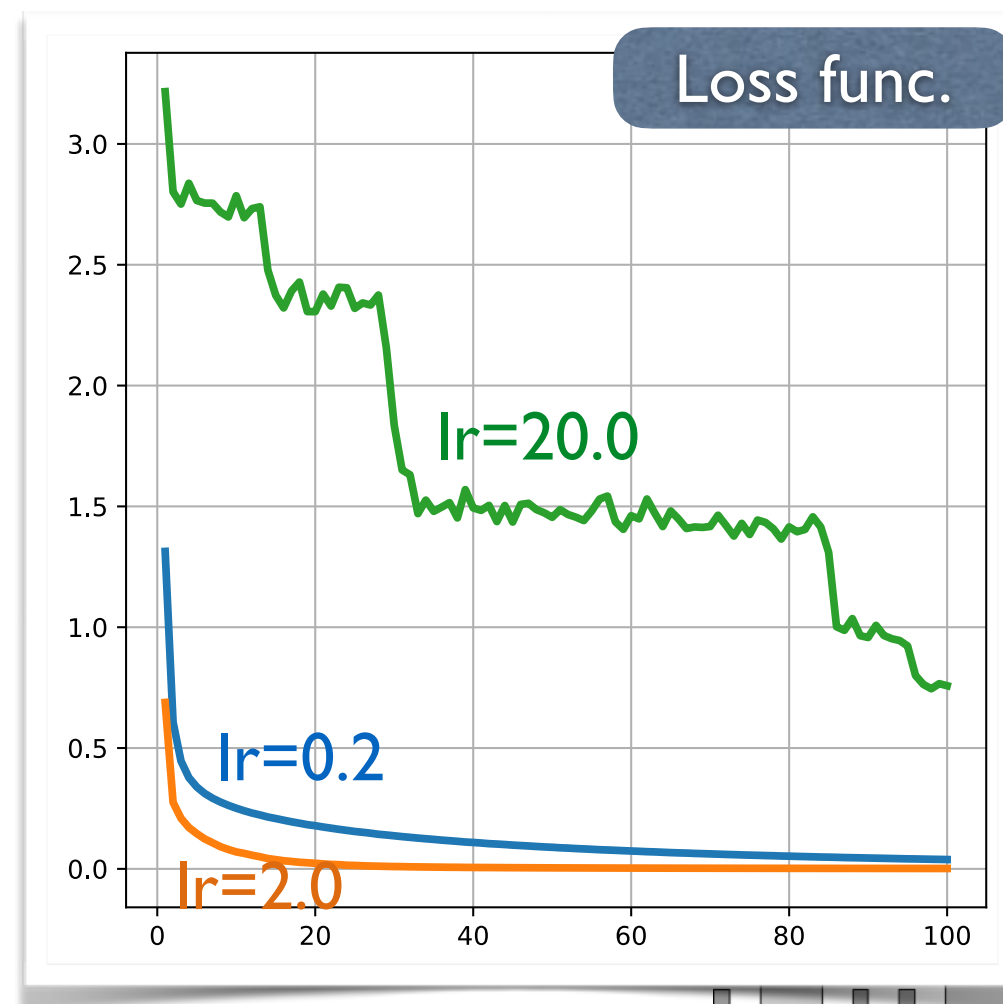
partial ex_ml7_2.py

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:

partial ex_ml7_3.py

```
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.))
```

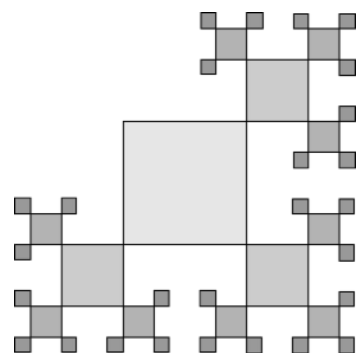


THE LEARNING RATE

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

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

- ❖ 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.

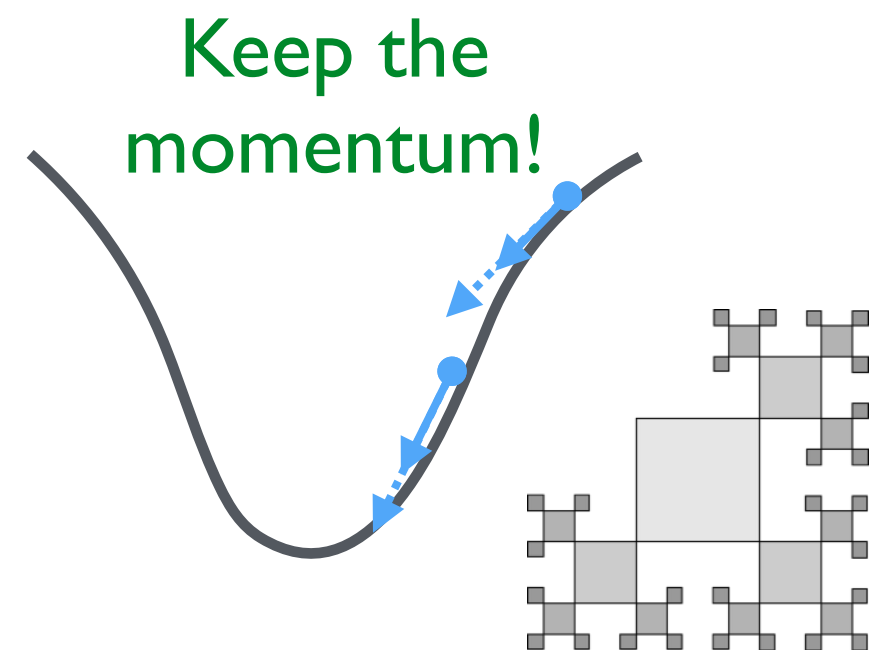


KEEP THE MOMENTUM?

- ❖ 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!



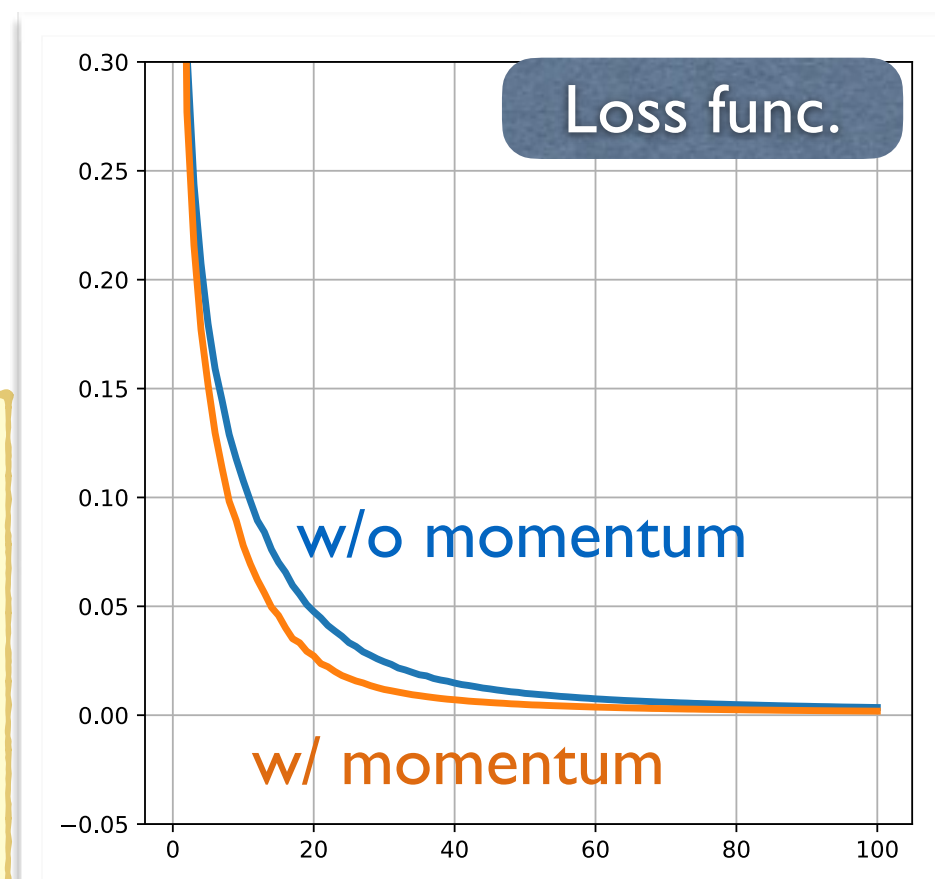
KEEP THE MOMENTUM? (II)

- ❖ 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!

partial ex_ml7_3a.py

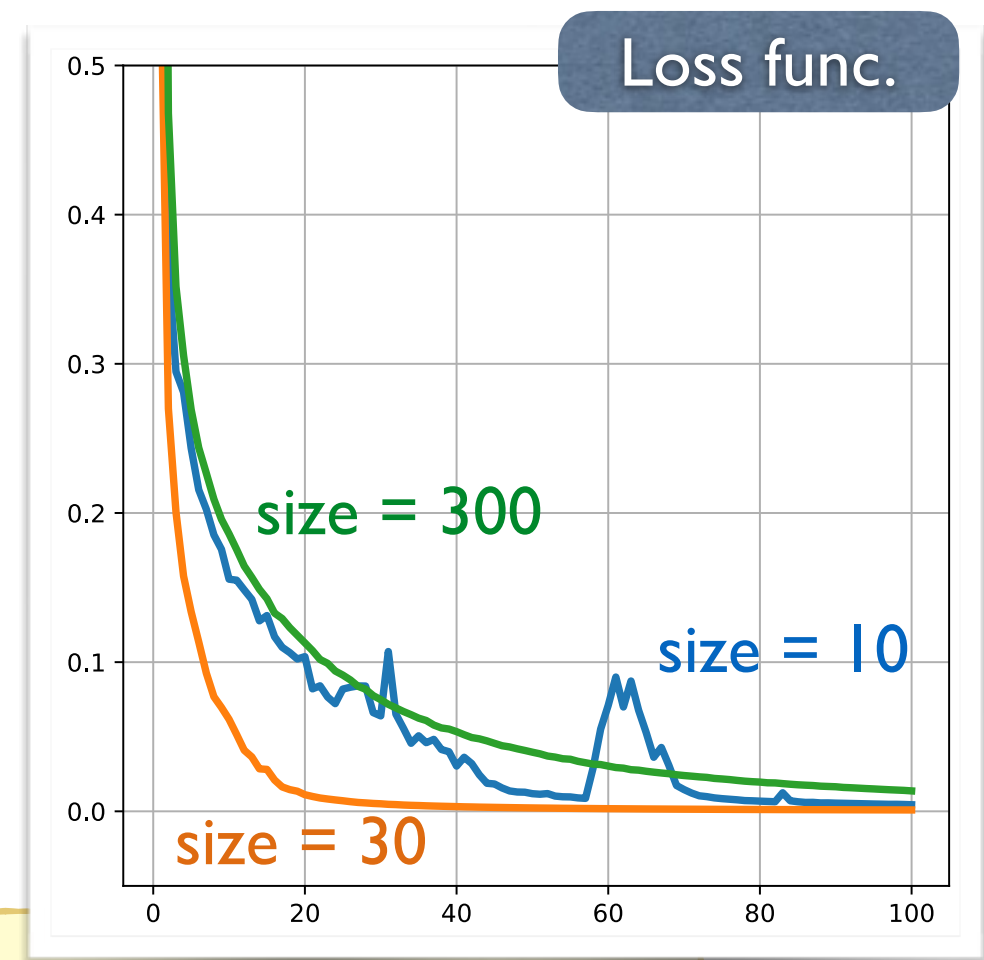
```
m1.compile(loss='categorical_crossentropy',  
           optimizer=SGD(lr=1.0))
```

```
m2 = clone_model(m1)  
m2.compile(loss='categorical_crossentropy',  
           optimizer=SGD(lr=1.0, momentum=0.4))
```



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.

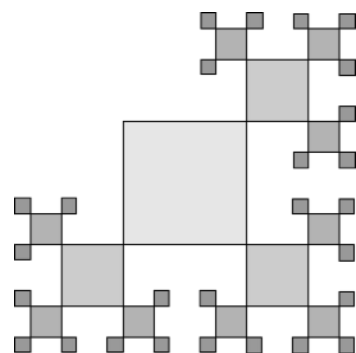


partial ex_ml7_3b.py

```
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)
```

HOW ABOUT A 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.

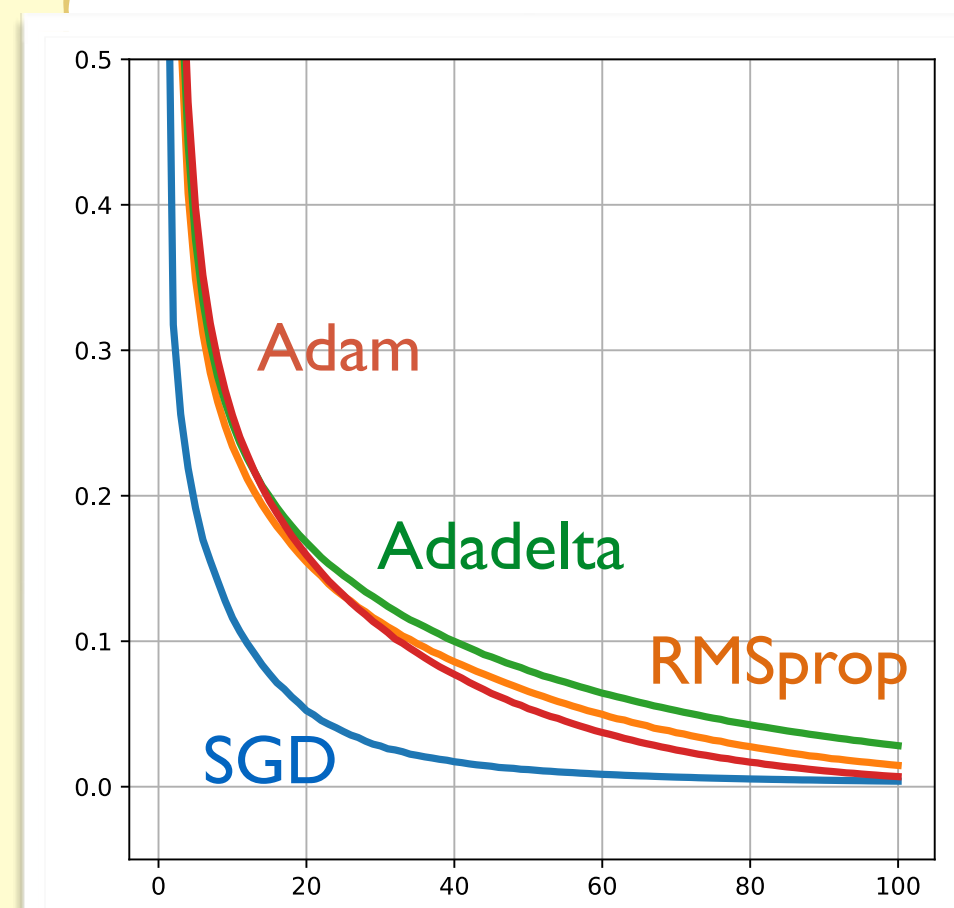


DIFFERENT TRAINING ALGORITHM? (III)

- ❖ 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 pretty well!*

partial ex_ml7_3c.py

```
m1.compile(loss='categorical_crossentropy',  
           optimizer=SGD(lr=1.0))  
  
m2 = clone_model(m1)  
m2.compile(loss='categorical_crossentropy',  
           optimizer=RMSprop())  
  
m3 = clone_model(m1)  
m3.compile(loss='categorical_crossentropy',  
           optimizer=Adadelta(lr=1.0))  
  
m4 = clone_model(m1)  
m4.compile(loss='categorical_crossentropy',  
           optimizer=Adam())
```

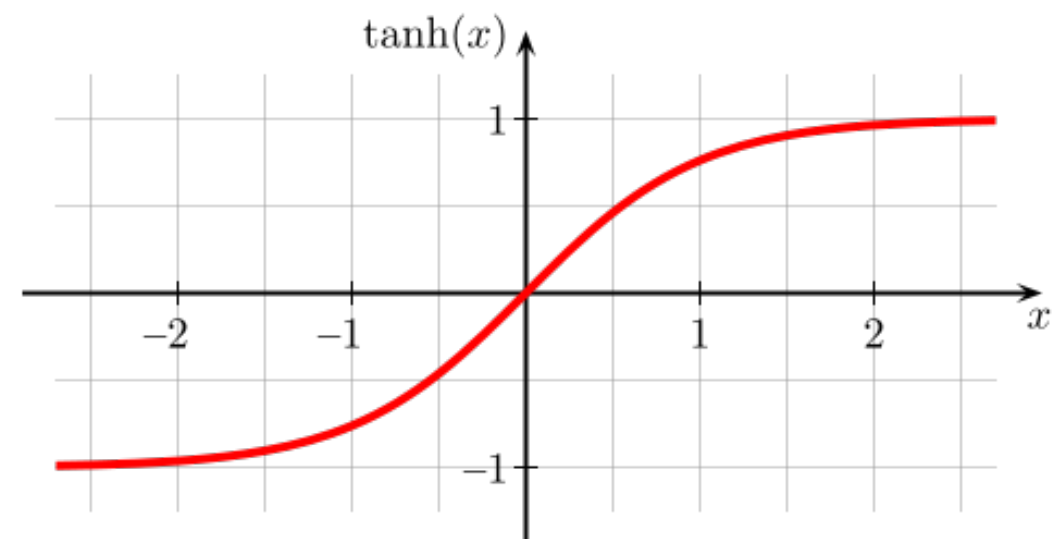


HOW ABOUT DIFFERENT ACTIVATION FUNCTION?

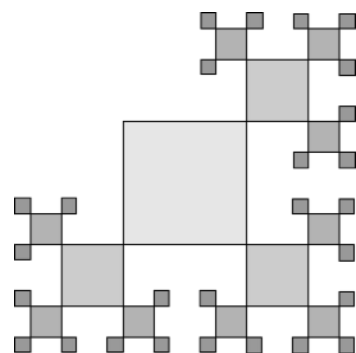
.....

- ❖ 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:

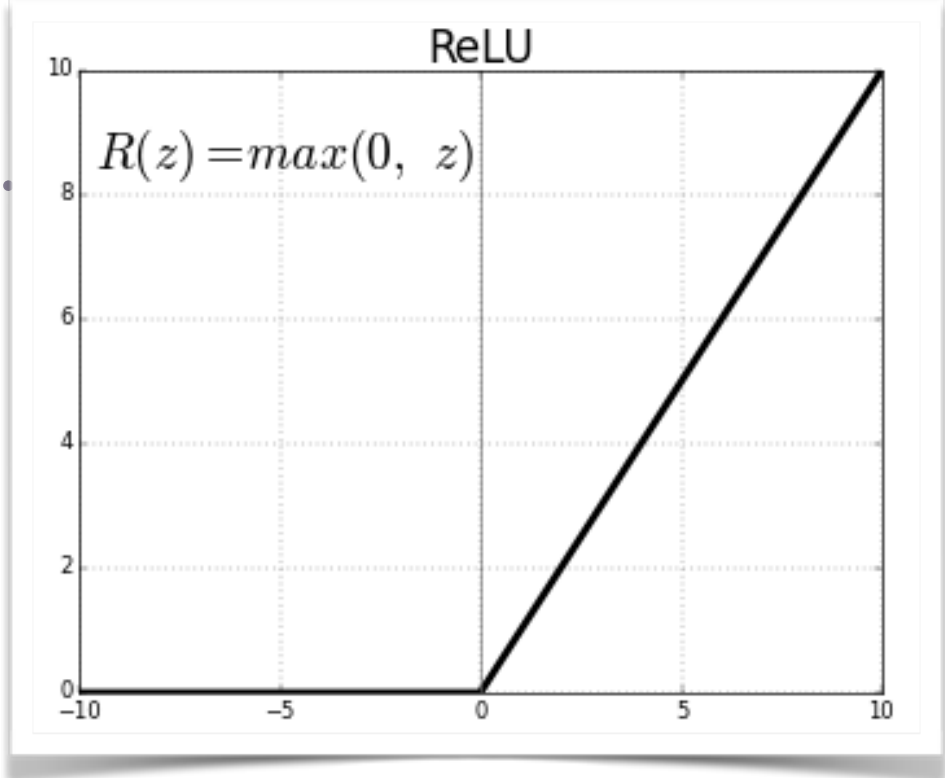
$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$
$$\sigma(z) = \frac{1 + \tanh(z/2)}{2}$$



- ❖ 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:
- 
- The graph shows the ReLU function, labeled $R(z) = \max(0, z)$. The x-axis ranges from -10 to 10 with major ticks at -10, -5, 0, 5, and 10. The y-axis ranges from 0 to 10 with major ticks at 0, 2, 4, 6, 8, and 10. The function is zero for all negative values of z and increases linearly with a slope of 1 for all positive values of z .
- ❖ 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.*

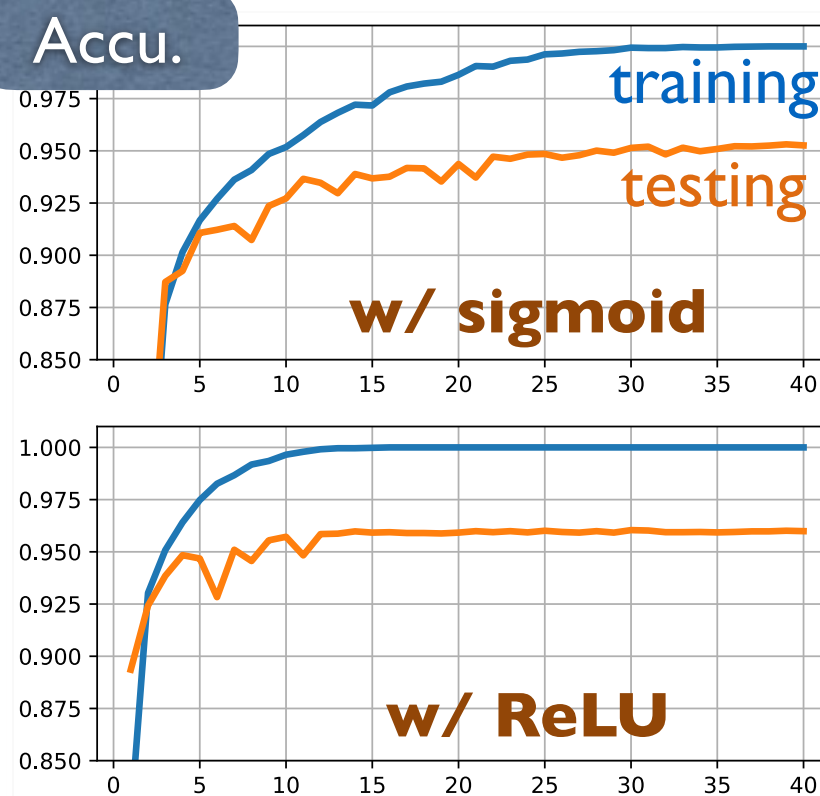
DIFFERENT ACTIVATION FUNCTION? (III)

- ❖ 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:

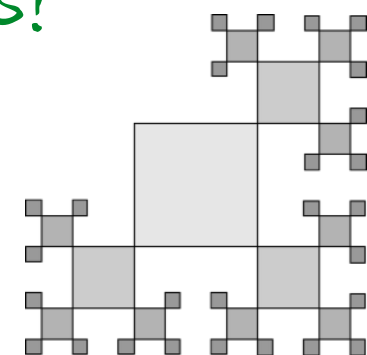
partial ex_ml7_3d.py

```
.....
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'])
.....
```

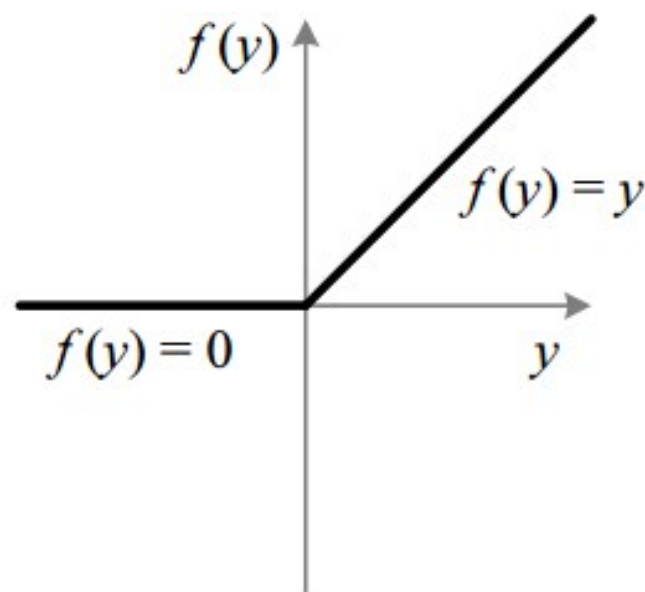


It improves!

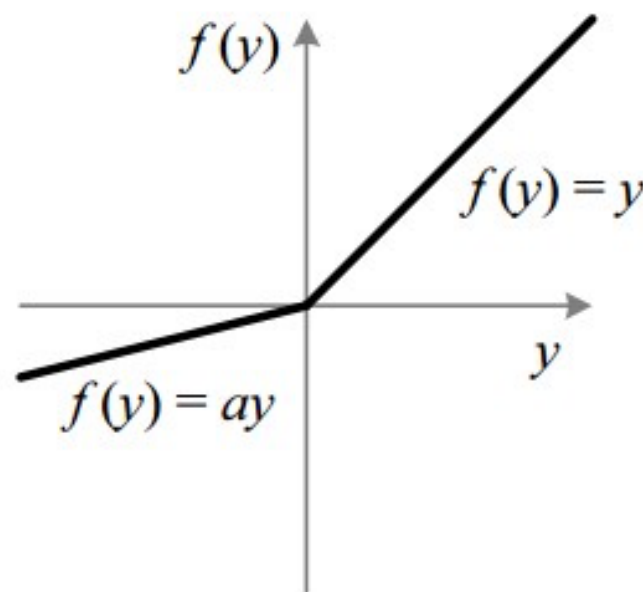


DIFFERENT ACTIVATION FUNCTION? (IV)

- ❖ The **Leaky ReLU** or **Parametric ReLU** are variations of the ReLU function to reduce the issue of “switch-off” problem (*ie. when the input stay at the “off” region, no chance to get it back...*).



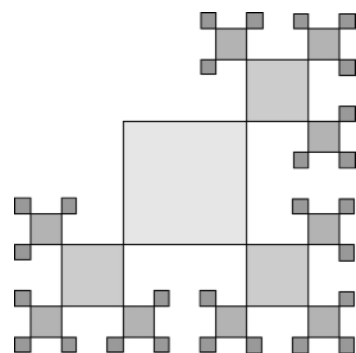
ReLU



Leaky ReLU / PReLU

Leaky ReLU treat the slope (α) as a fixed hyper parameter, while **PReLU** include it as a parameter in the training.

- ❖ There are few more variations of the ReLU functions, left for your own study!





MODULE SUMMARY

.....

- ❖ In this module we continue to discuss how to improve the performance of neural network with several known tricks and potentially can be introduced in your own study.
- ❖ Next module we will enter the regime with *deeper* neural network, and see how to improve the performance even further!

