



Regression

1

2.1 MLP regression.ipynb





[Pull requests](#) [Issues](#) [Marketplace](#) [Explore](#)

 [TienLungSun / PyTorch-deep-learning](#) Public Pin

[Code](#) [Issues](#) [Pull requests](#) [Actions](#) [Projects](#) [Wiki](#) [Security](#) [Insights](#) [Settings](#)

[main](#) [PyTorch-deep-learning / 2.1. MLP regression.ipynb](#)

 **TienLungSun** 使用 Colaboratory 建立

 1 contributor

980 lines (980 sloc) | 55.2 KB

How machine learns from data?

- Regression
- Classification

$$y = f(x)$$

- Define a function to be learned: $y = f(x)$
- Define a loss function \mathcal{L} to describe the error between $\hat{y} = f(x)$ and y
- Find the optimal parameters of f that minimize \mathcal{L}

Statistics vs Machine Learning

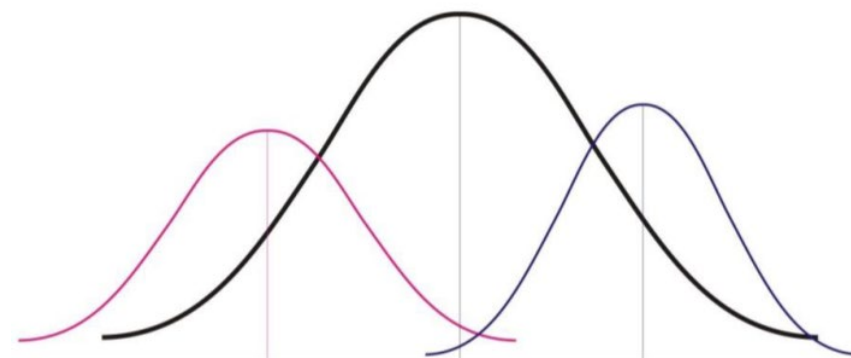
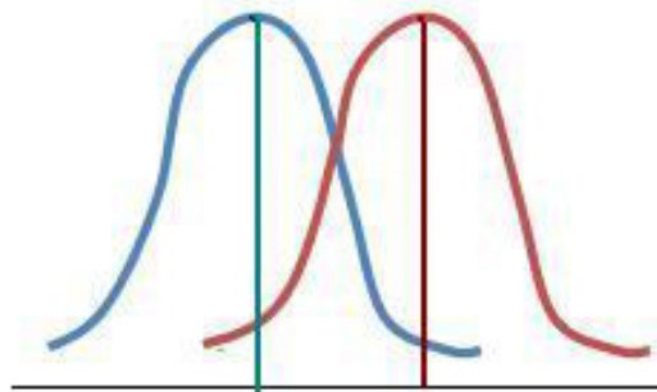
Statistics versus machine learning

Statistics draws population inferences from a sample, and machine learning finds generalizable predictive patterns.

$$y = f(x)$$

Statistics draw population inferences

動作	range (delta D)
A1	8.20
A1	6.42
A1	8.26
A1	14.55
A1	11.70
A1	11.75
A1	12.72
A1	11.96
A1	6.03
A1	9.20
A1	14.25
A1	13.93
A2	14.17
A2	19.91
A2	9.40
A2	8.48
A2	10.61
A2	13.86
A2	7.15
A2	7.66
A2	6.09
A2	13.18
A2	9.15
A2	13.29



Evaluation of generalizable prediction performance with test data

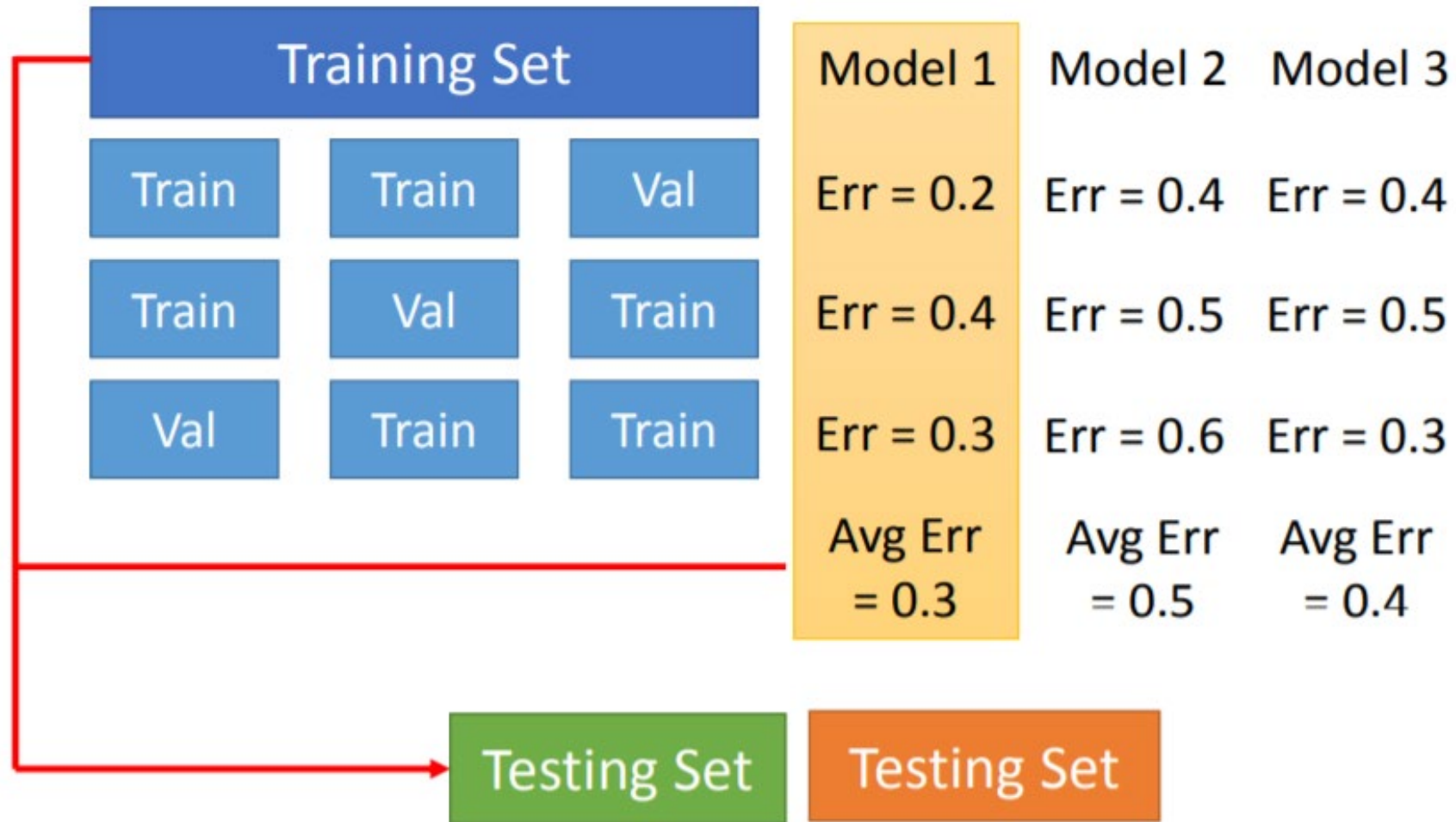
5

Split input data to train and test data

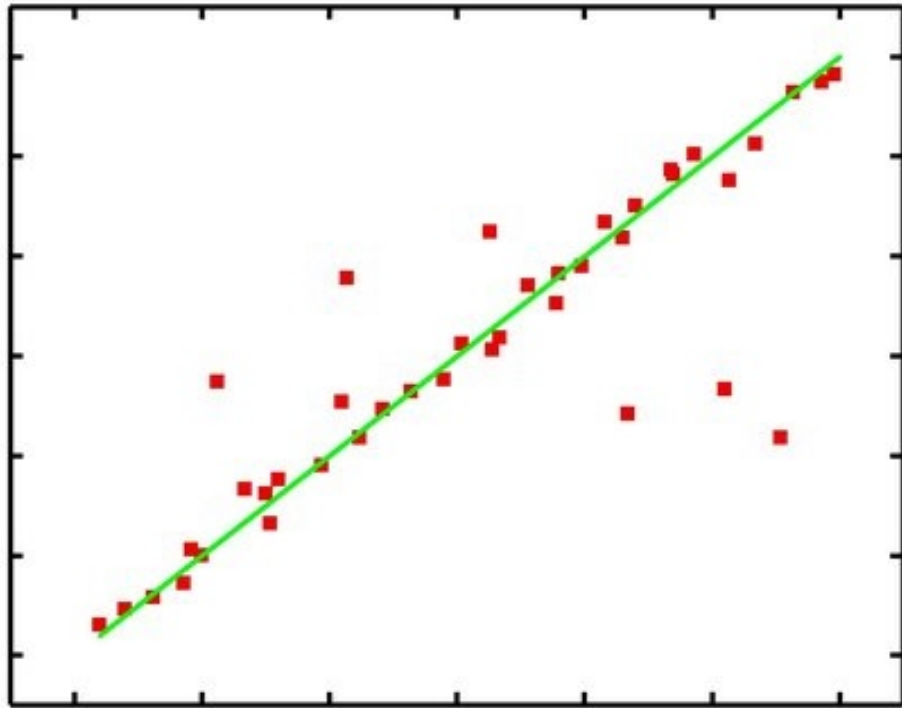
```
In [5]: from sklearn.model_selection import train_test_split
trainX, testX, trainY, testY = train_test_split(numpyX, numpyY, test_size=0.20, random_state=0)
print(trainX.shape, testX.shape, trainY.shape, testY.shape)
```

(1600, 7) (400, 7) (1600, 1) (400, 1)

Cross validation



Statistics vs Machine Learning



(b) Linear Regression

Statistics – R^2 of the variance explained
ML – MSE, MAE of test data which are
unseen by the model

Input data

y	x1	x2	...
0.578			
0.64			
-0.96			
0.23			
...			

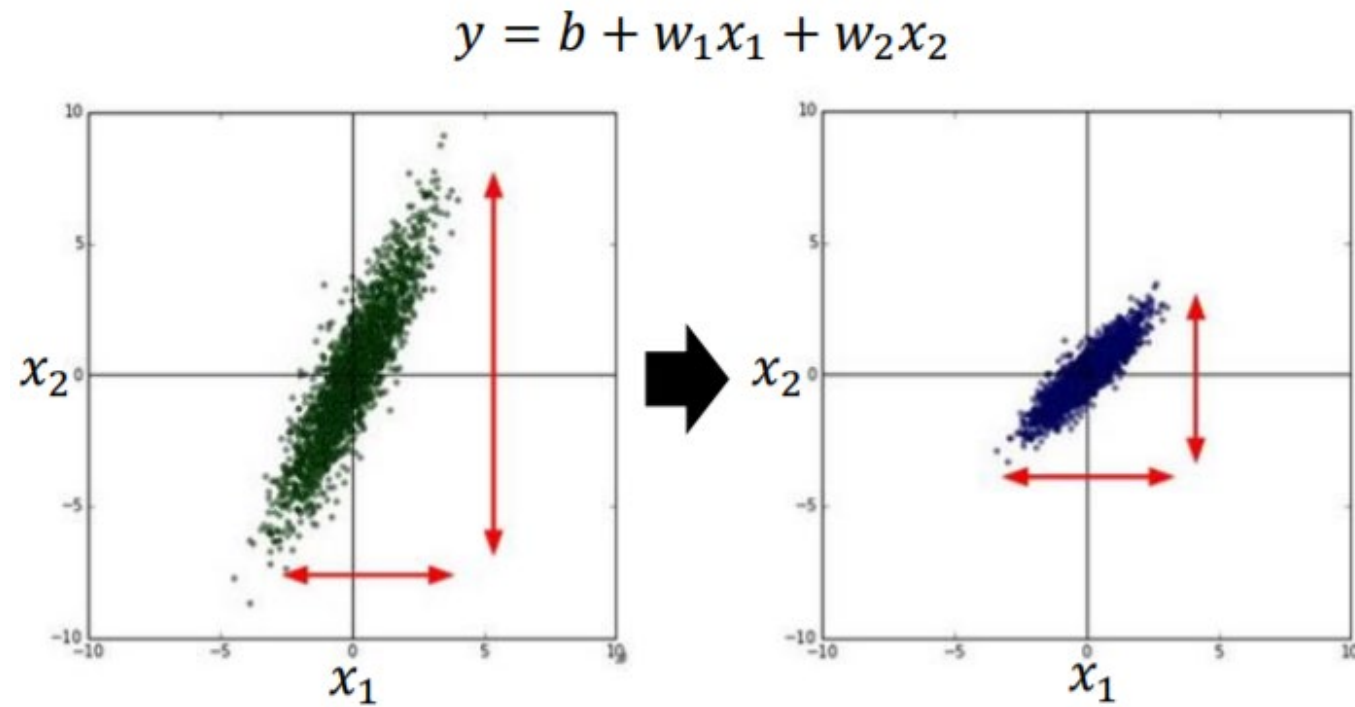
$$y = 0.323x_1^2 + 0.586x_1x_2 + 0.4x_3 + 0.8972x_5^3 + 0.267x_3^2x_5x_6 + 0.78x_7^2$$

$$x_1 \sim \mathcal{N}(0.5, 0.3)$$

$$x_2 \sim \mathcal{N}(0.8, 0.5)$$

...

Feature scaling

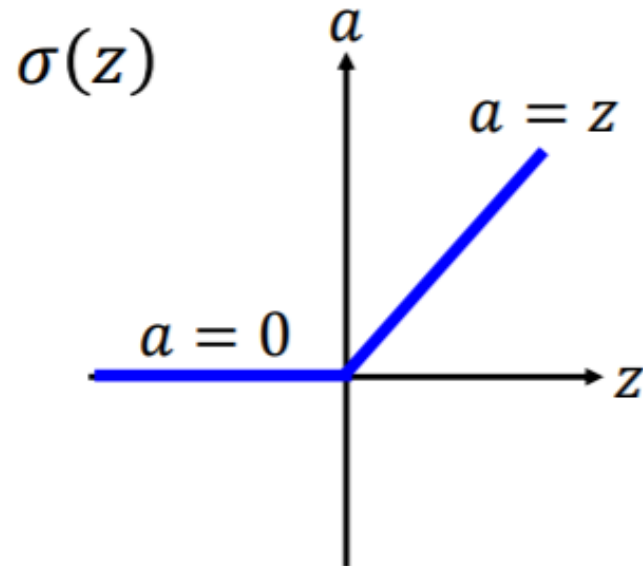


Make different features have the same scaling

Reference: 李弘毅 ML Lecture 3-1 <https://youtu.be/yKKNr-QKz2Q>

ReLU activation function

- Rectified Linear Unit (ReLU)

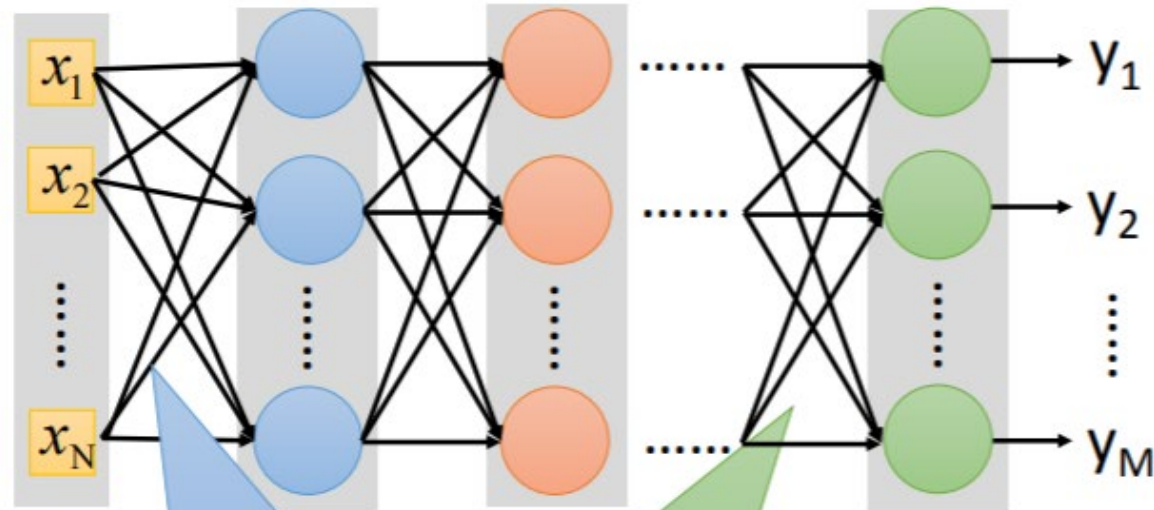


[Xavier Glorot, AISTATS'11]
[Andrew L. Maas, ICML'13]
[Kaiming He, arXiv'15]

Reason:

1. Fast to compute
2. Biological reason
3. Infinite sigmoid with different biases
4. Vanishing gradient problem

Vanishing gradient problem



Smaller gradients

Learn very slow

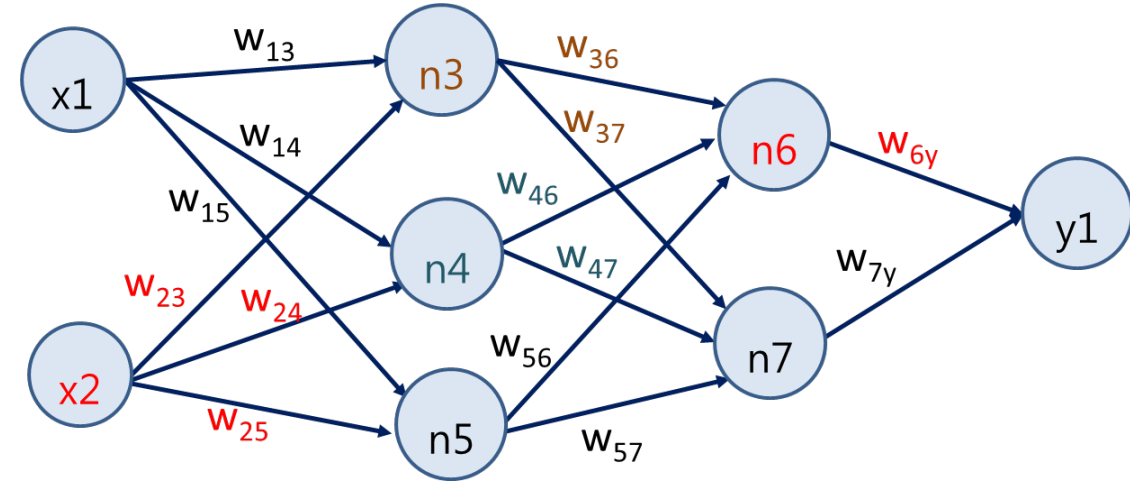
Almost random

Larger gradients

Learn very fast

Already converge

based on random!?

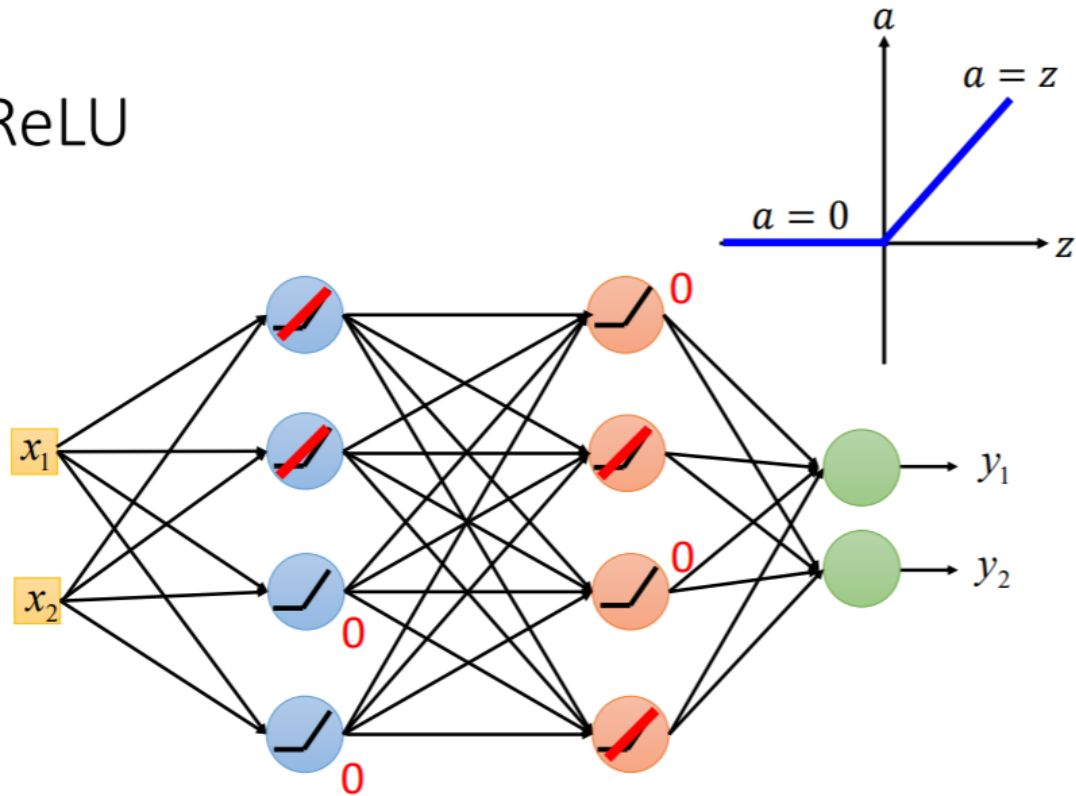


$$\frac{\partial L}{\partial \mathbf{w}_{6y}} = \frac{\partial L}{\partial y_1} \frac{\partial y_1}{\partial \mathbf{w}_{6y}}$$

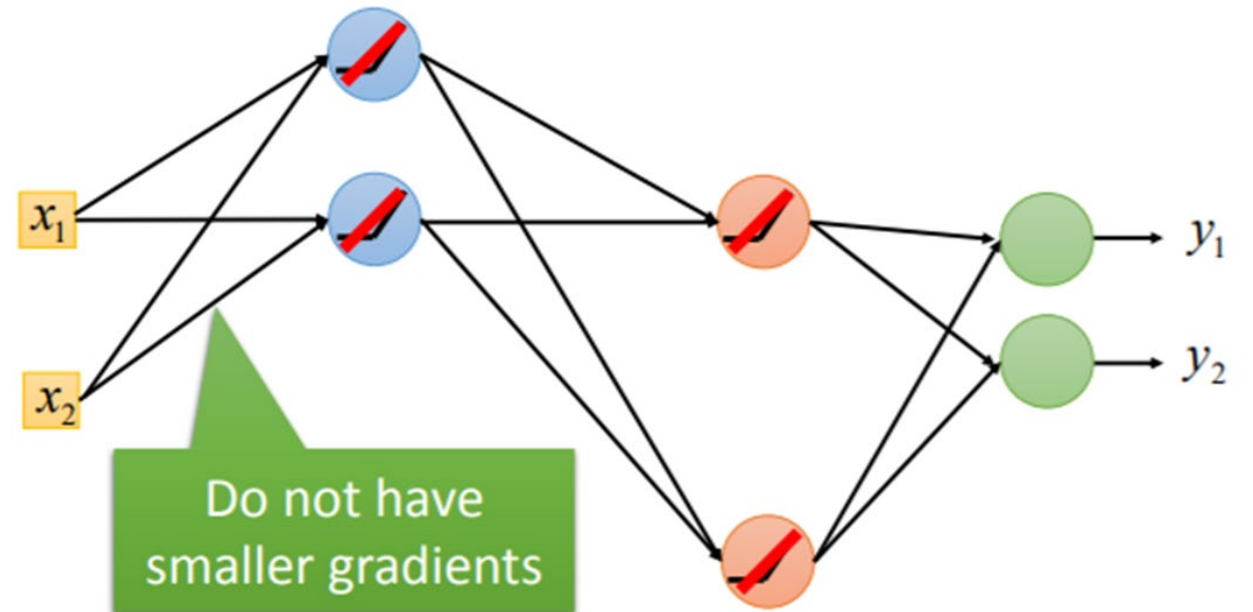
$$\frac{\partial L}{\partial \mathbf{w}_{57}} = \frac{\partial L}{\partial y_1} \frac{\partial y_1}{\partial n_7} \frac{\partial n_7}{\partial \mathbf{w}_{57}}$$

ReLU results in a thinner linear network

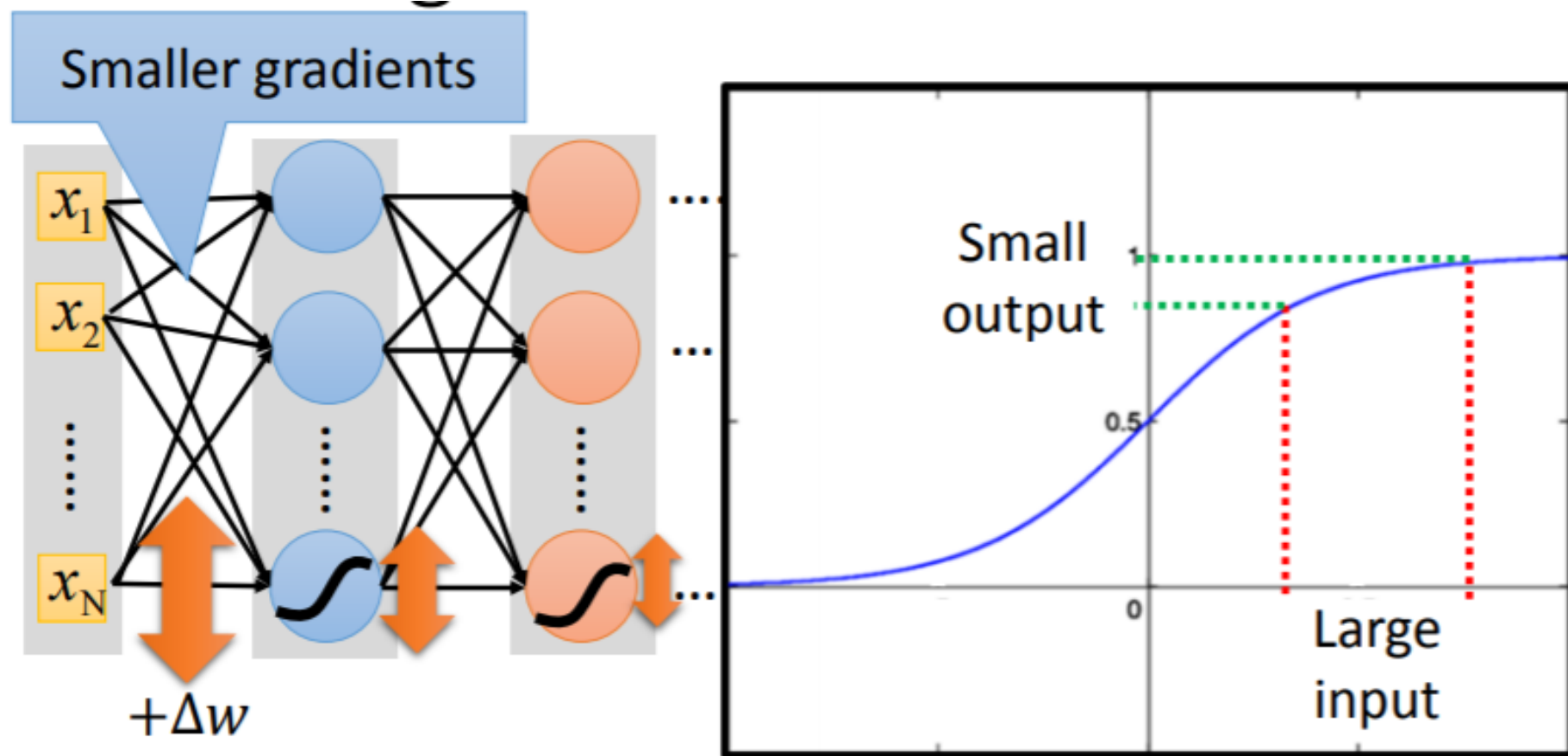
ReLU



A Thinner linear network

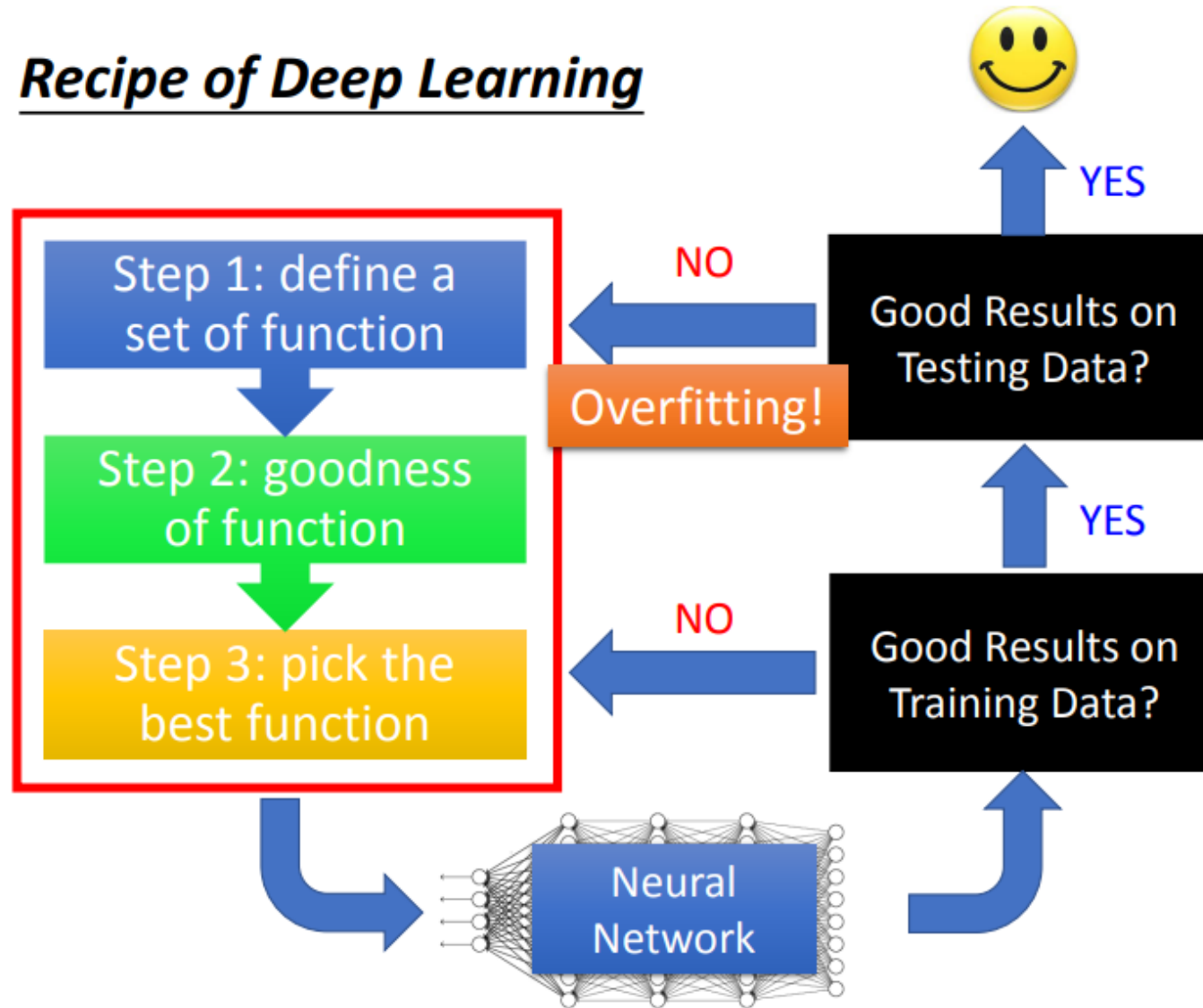


Sigmoid is hard to get the power of deep

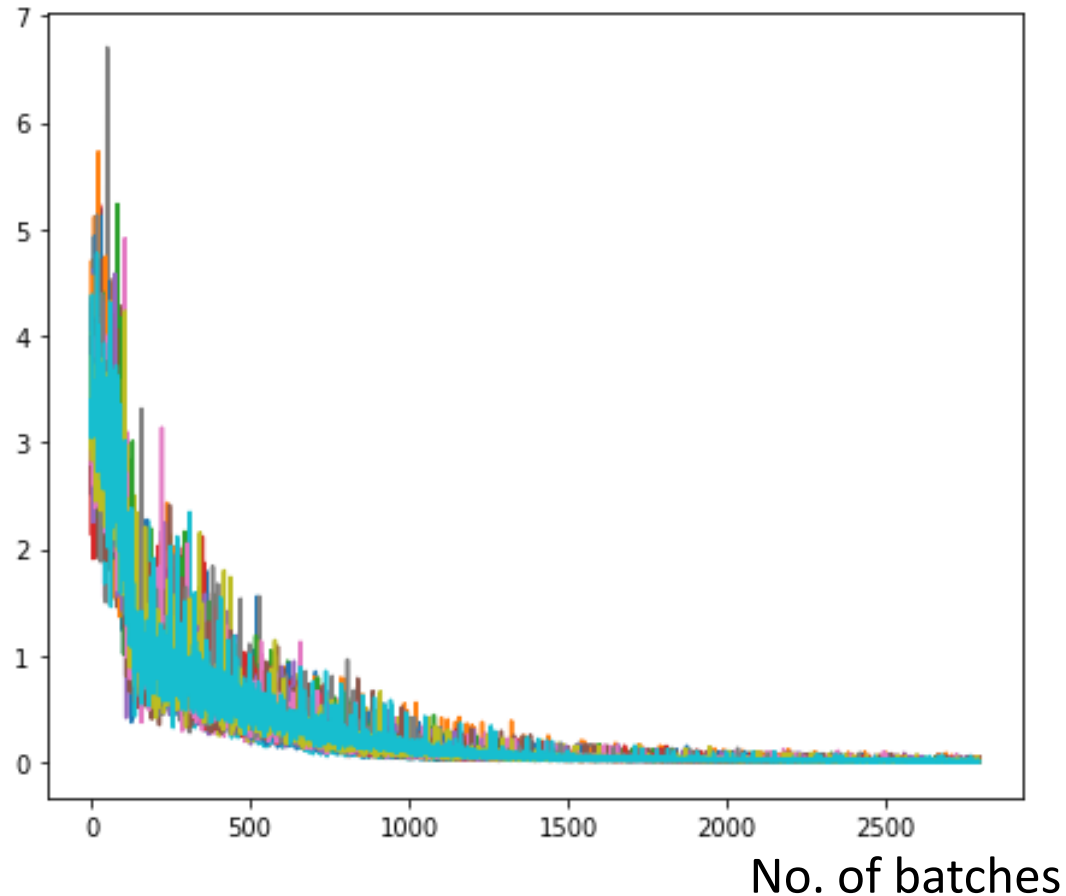


ML finds generalizable predictive patterns

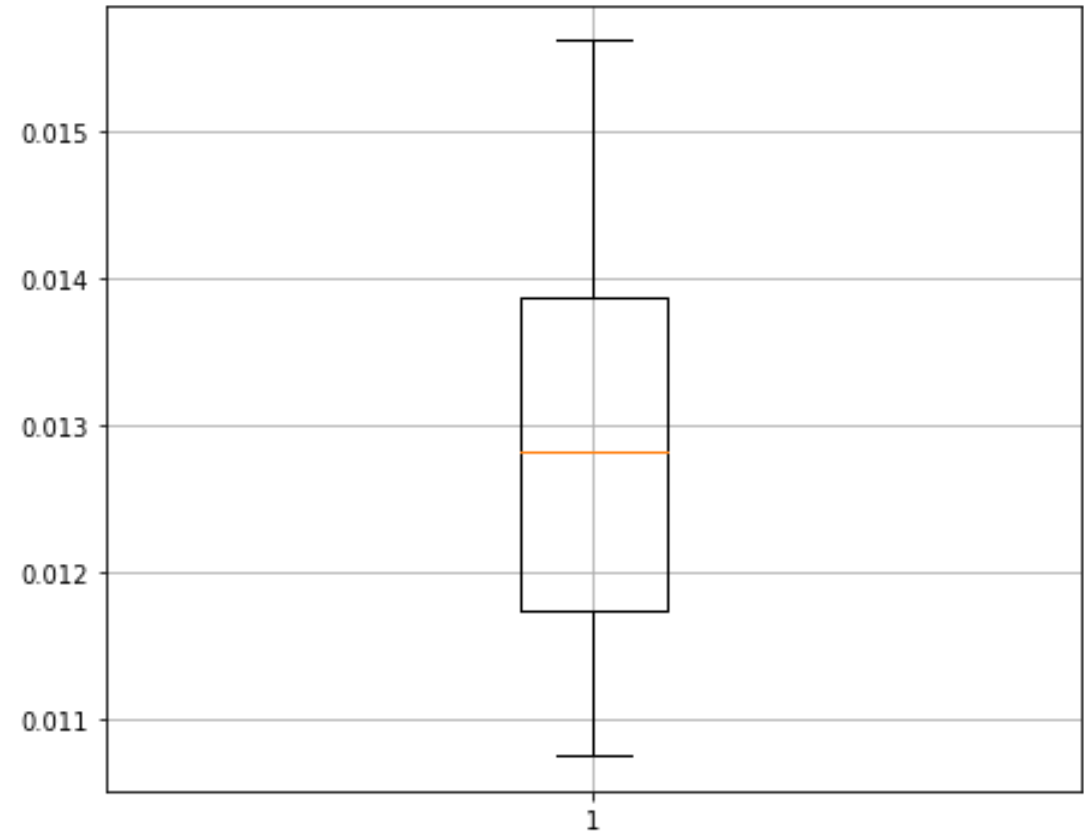
Recipe of Deep Learning



ML finds generalizable predictive patterns



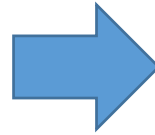
Good results on training data?



Good results on test data?

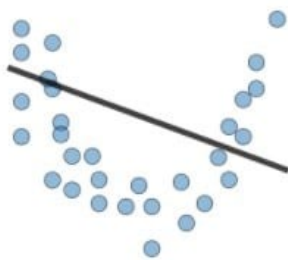

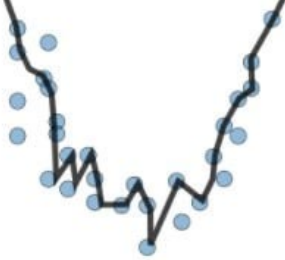
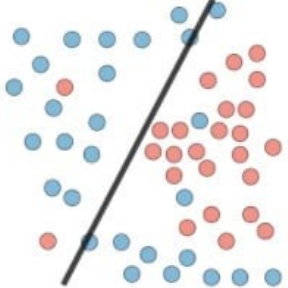
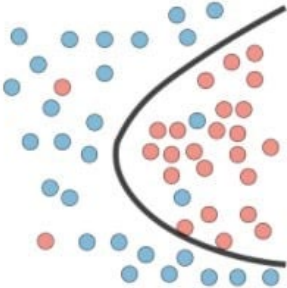
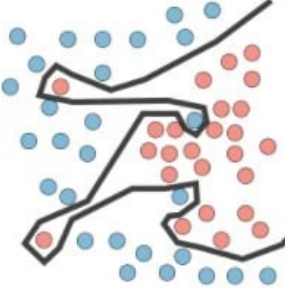

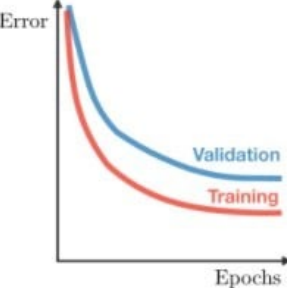
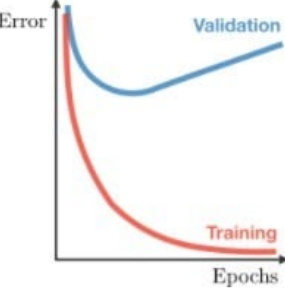
Try different NNs

- 7-56-56-1
- 7-256-256-256-1
- 7-512-512-512-512-1
- 7-1024-1024-1024-1024-1024-1



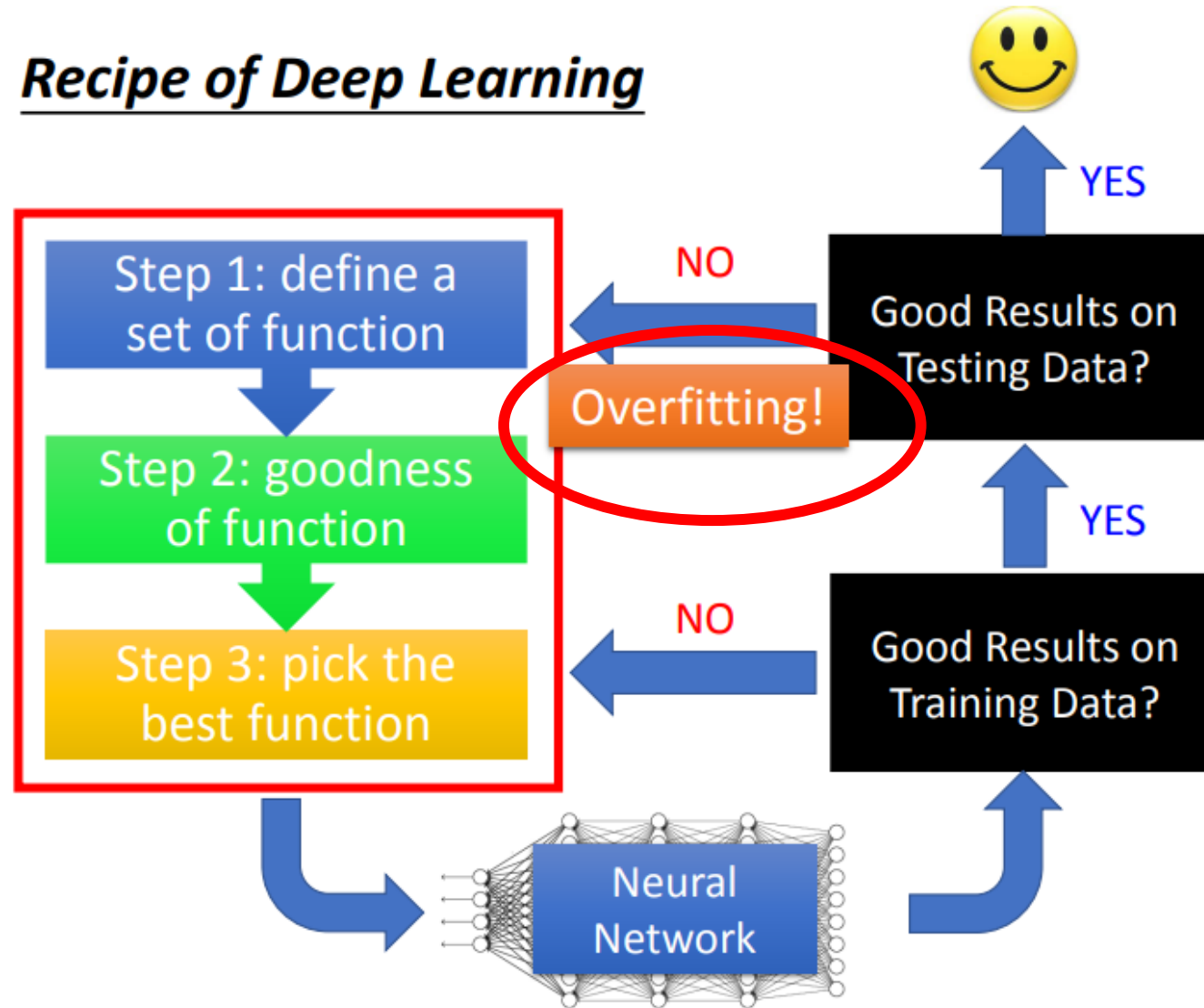
- Good results on training data?
- Good results on test data?

Overfitting

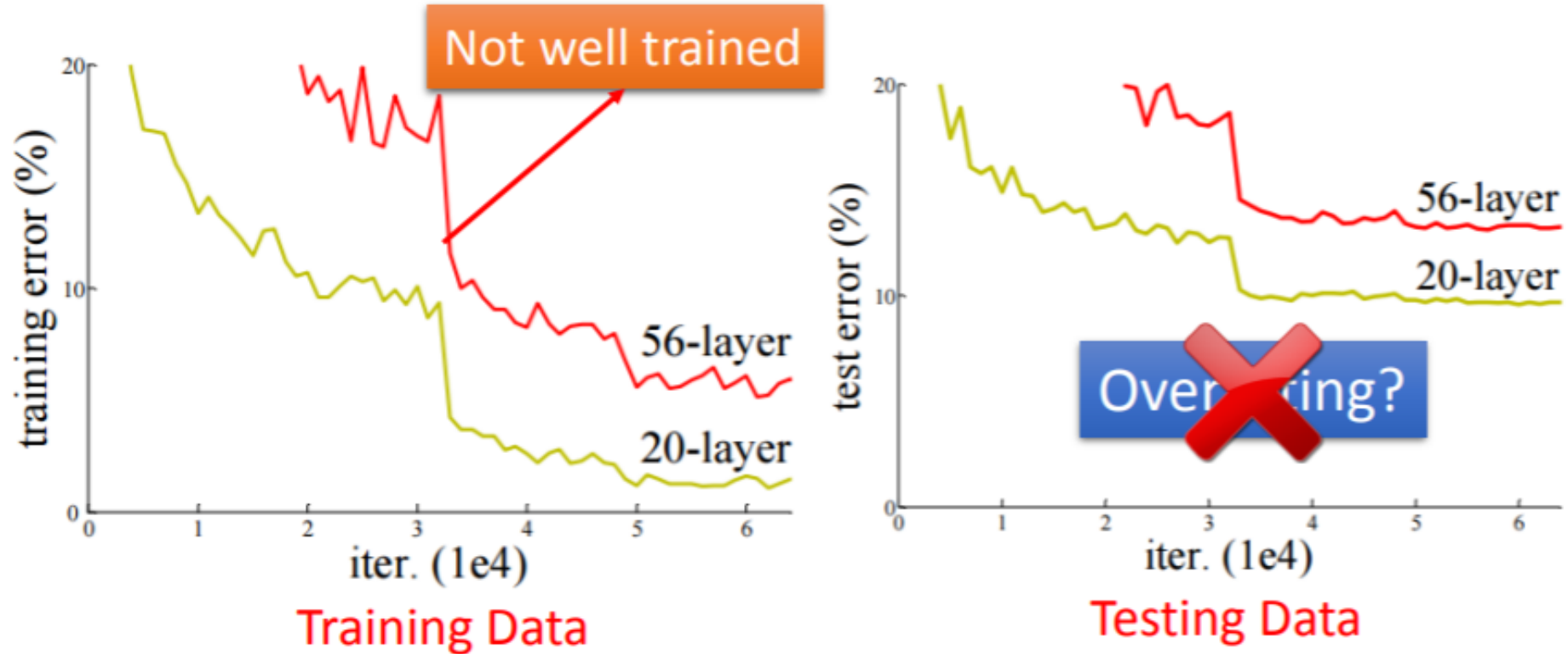
	Underfitting	Just right	Overfitting
Symptoms	<ul style="list-style-type: none">• High training error• Training error close to test error• High bias	<ul style="list-style-type: none">• Training error slightly lower than test error	<ul style="list-style-type: none">• Very low training error• Training error much lower than test error• High variance
Regression illustration			
Classification illustration			
Deep learning illustration			
Possible remedies	<ul style="list-style-type: none">• Complexify model• Add more features• Train longer		<ul style="list-style-type: none">• Perform regularization• Get more data

How to know overfitting happens ?

Recipe of Deep Learning



But be careful, do not always blame overfitting



Deep Residual Learning for Image Recognition
<http://arxiv.org/abs/1512.03385>

Overfitting

2.2. Overfitting.ipynb

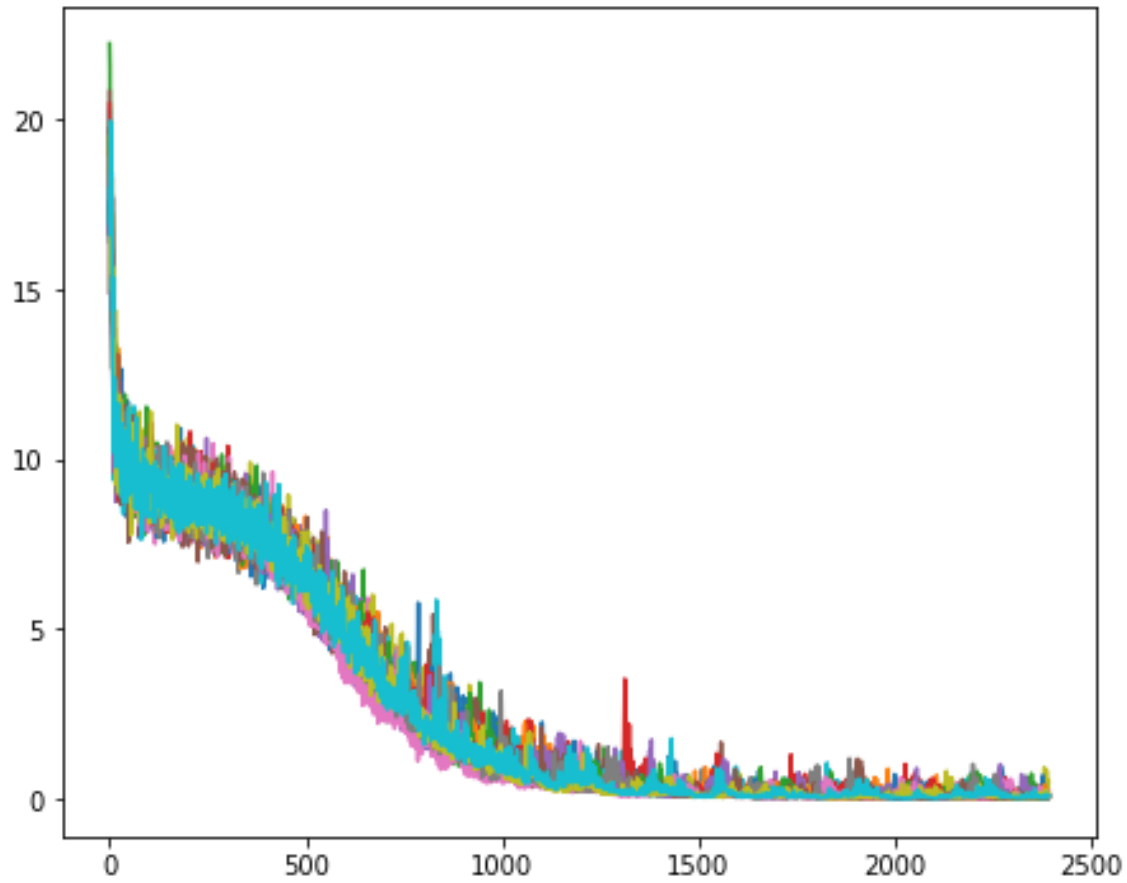
Add noises to y

$$y = 0.323x_1^2 + 0.586x_1x_2 + 0.4x_3 + 0.8972x_5^3 + 0.267x_3^2x_5x_6 + 0.78x_7^2 + \mathcal{N}(2, 3)$$

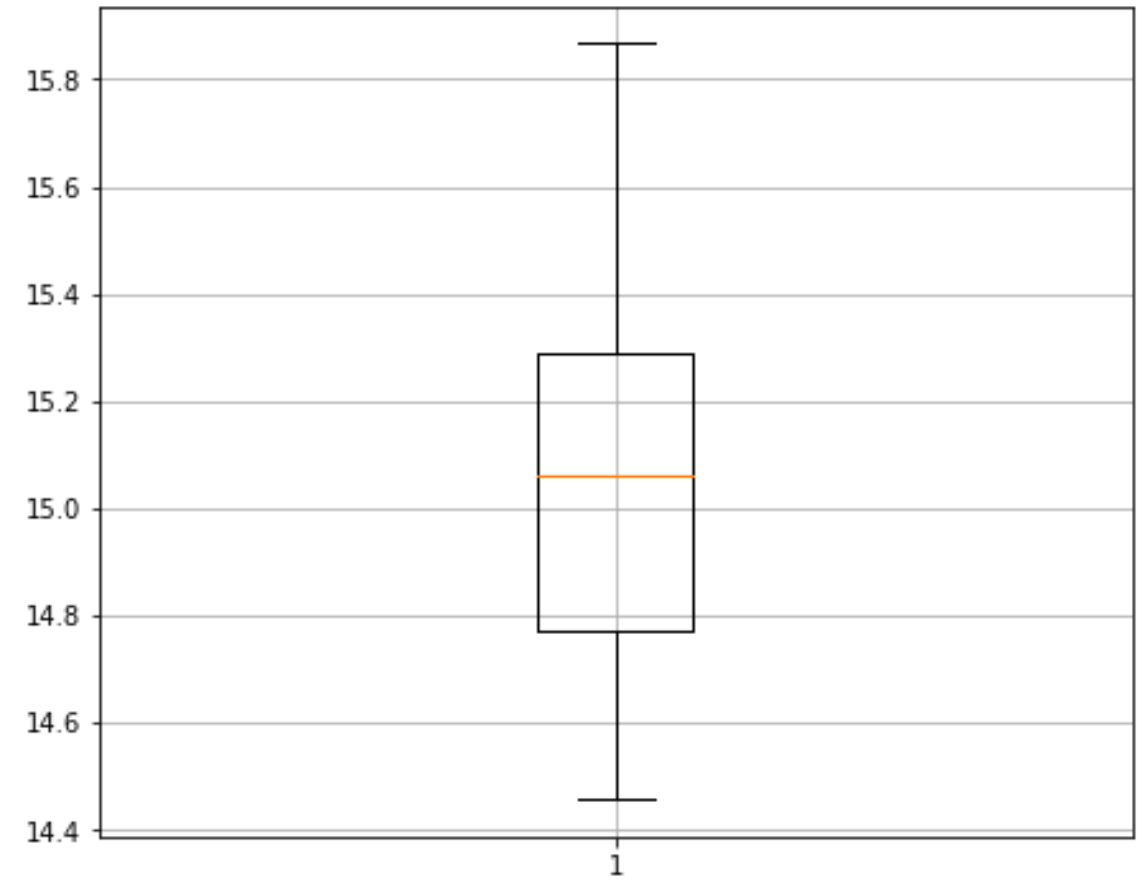
Use a complicated NN

7-1024-1024-1024-1024-1

Overfitting



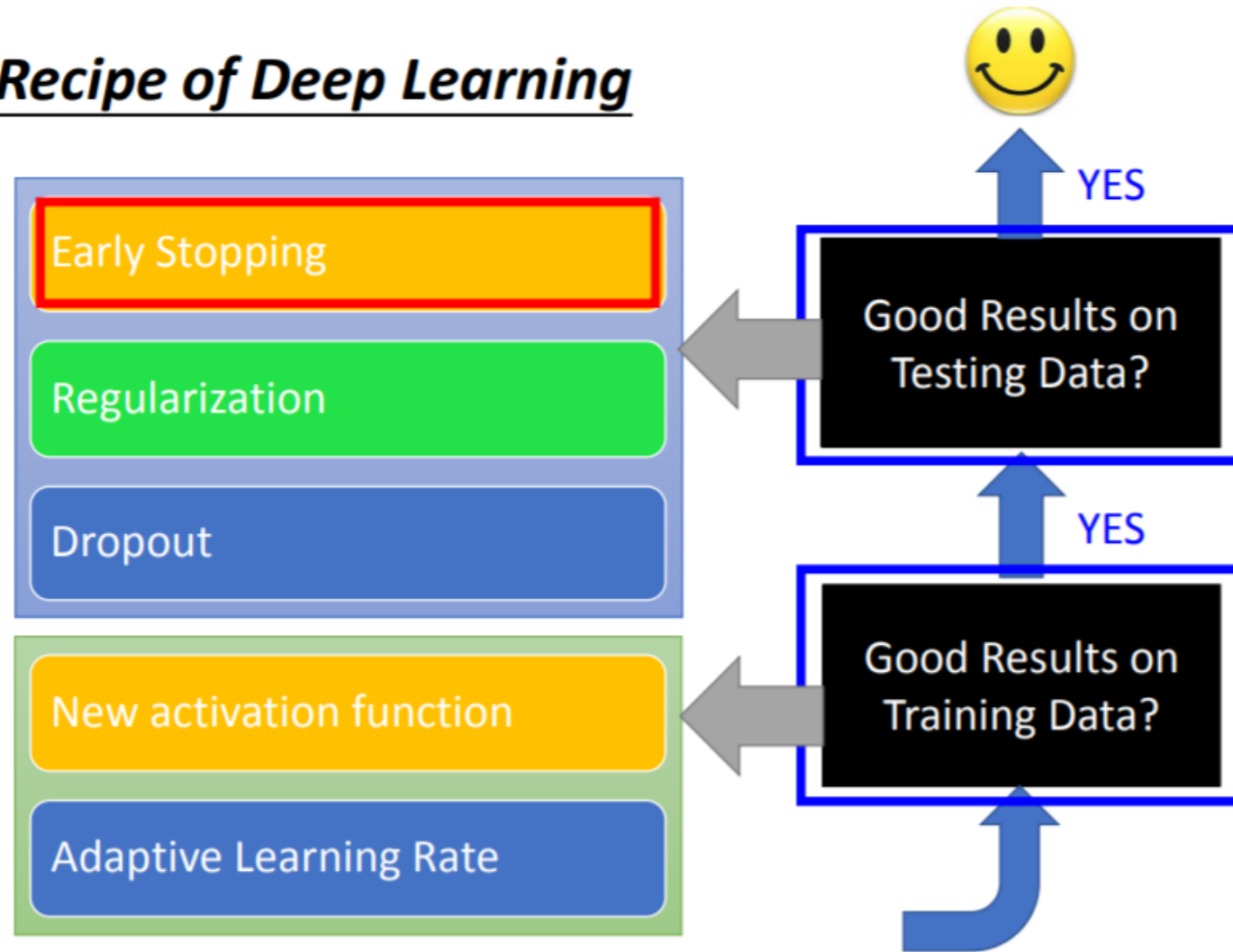
Good results on training data



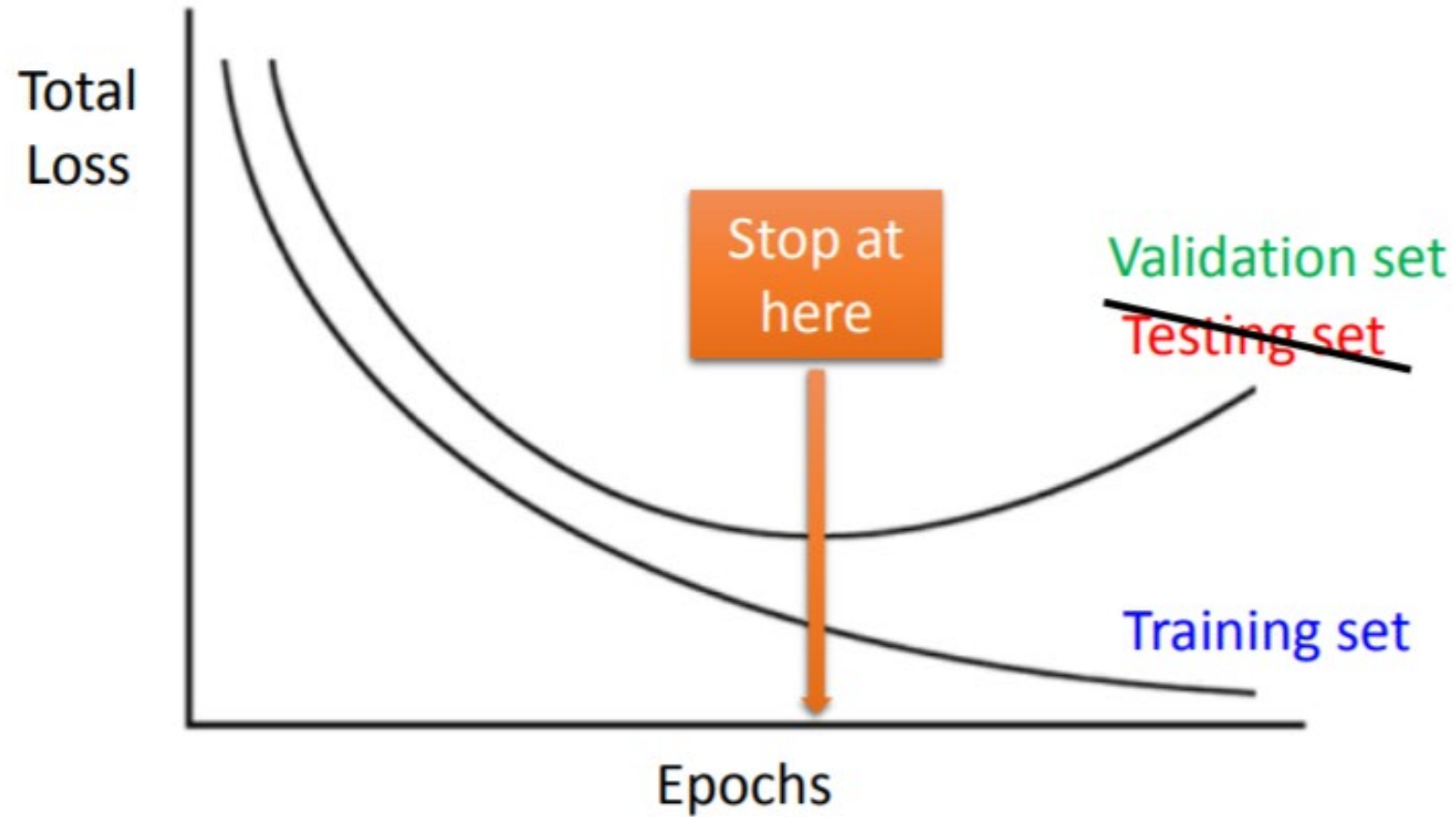
Bad results on test data

How to solve overfitting?

Recipe of Deep Learning



(1) Early stopping



Keras: <http://keras.io/getting-started/faq/#how-can-i-interrupt-training-when-the-validation-loss-isnt-decreasing-anymore>

(1) Early stopping

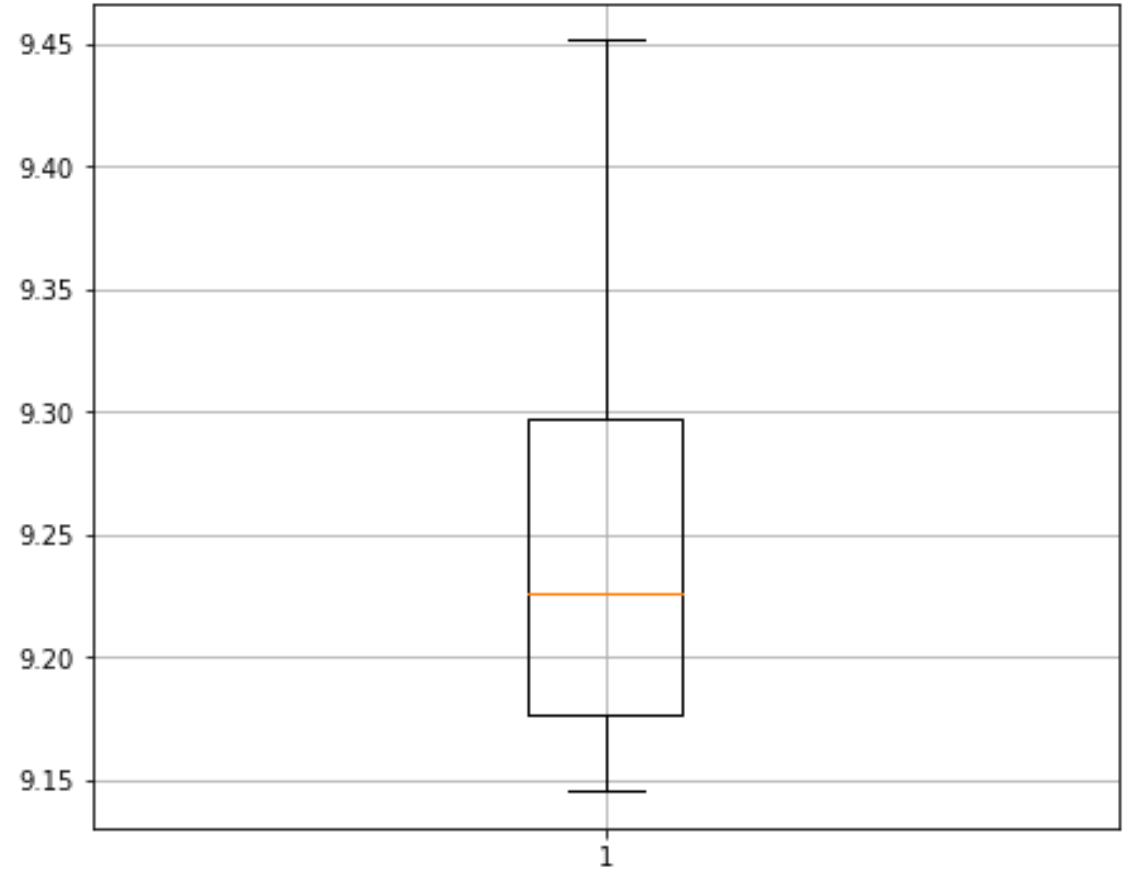
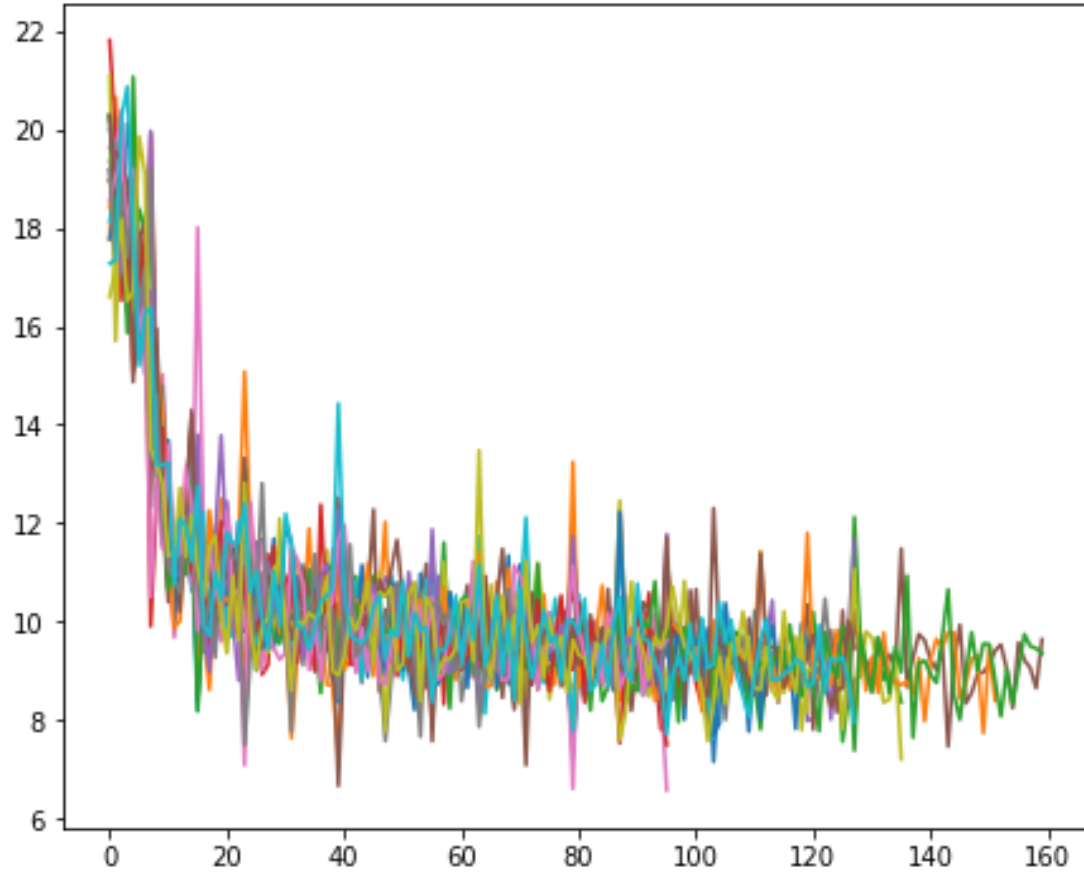
```

tensor_validationX = torch.FloatTensor(validationX).to(device)
tensor_validationY = torch.FloatTensor(validationY).to(device)
tensor_validationY_hat = MyNet(tensor_validationX)
validation_loss = loss_func(tensor_validationY, tensor_validationY_hat)

#early stop
the_current_loss = float(validation_loss)
if(the_current_loss > the_last_loss):
    trigger_times += 1
    print('trigger times:', trigger_times)
    if(trigger_times >= patience):
        print('Early stopping!')
        break # early stop when validation loss increases 2 consecutive times
elif(trigger_times > 0):
    print('trigger times reset to 0')
    trigger_times = 0
the_last_loss = the_current_loss
train_lossLst.append(epoch_lossLst)

```

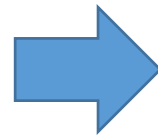

(1) Early stopping



Try different NNs

Use your own $y = f(\vec{x})$ data

- 7-512-512-512-1
- 7-256-256-256-1
- 7-56-56-1



- Is the overfitting problem solved by early stop?

(2) Regularization – L2

- Find a set of weight not only minimizing original cost but also close to zero

$$L'(\theta) = \underbrace{L(\theta)}_{\text{Original loss}} + \lambda \frac{1}{2} \underbrace{\|\theta\|_2}_{\text{Regularization term}}$$

$$\theta = \{w_1, w_2, \dots\}$$

L2 regularization:

$$\|\theta\|_2 = (w_1)^2 + (w_2)^2 + \dots$$

(usually not consider biases)

$$L(\theta) = \sum_{n=1}^N (\hat{y}^n - y^n)^2$$

(2) Regularization – L2

$$L'(\theta) = L(\theta) + \lambda \frac{1}{2} \|\theta\|_2^2 \quad \text{Gradient: } \frac{\partial L'}{\partial w} = \frac{\partial L}{\partial w} + \lambda w$$

$$\begin{aligned} \text{Update: } w^{t+1} &\rightarrow w^t - \eta \frac{\partial L'}{\partial w} = w^t - \eta \left(\frac{\partial L}{\partial w} + \lambda w^t \right) \\ &= \underbrace{(1 - \eta \lambda) w^t}_{\substack{\downarrow \\ \text{Closer to zero}}} - \eta \underbrace{\frac{\partial L}{\partial w}}_{\text{Weight Decay}} \end{aligned}$$

Vanilla gradient decent update $w^{t+1} \rightarrow w^t - \eta \frac{\partial L}{\partial w}$

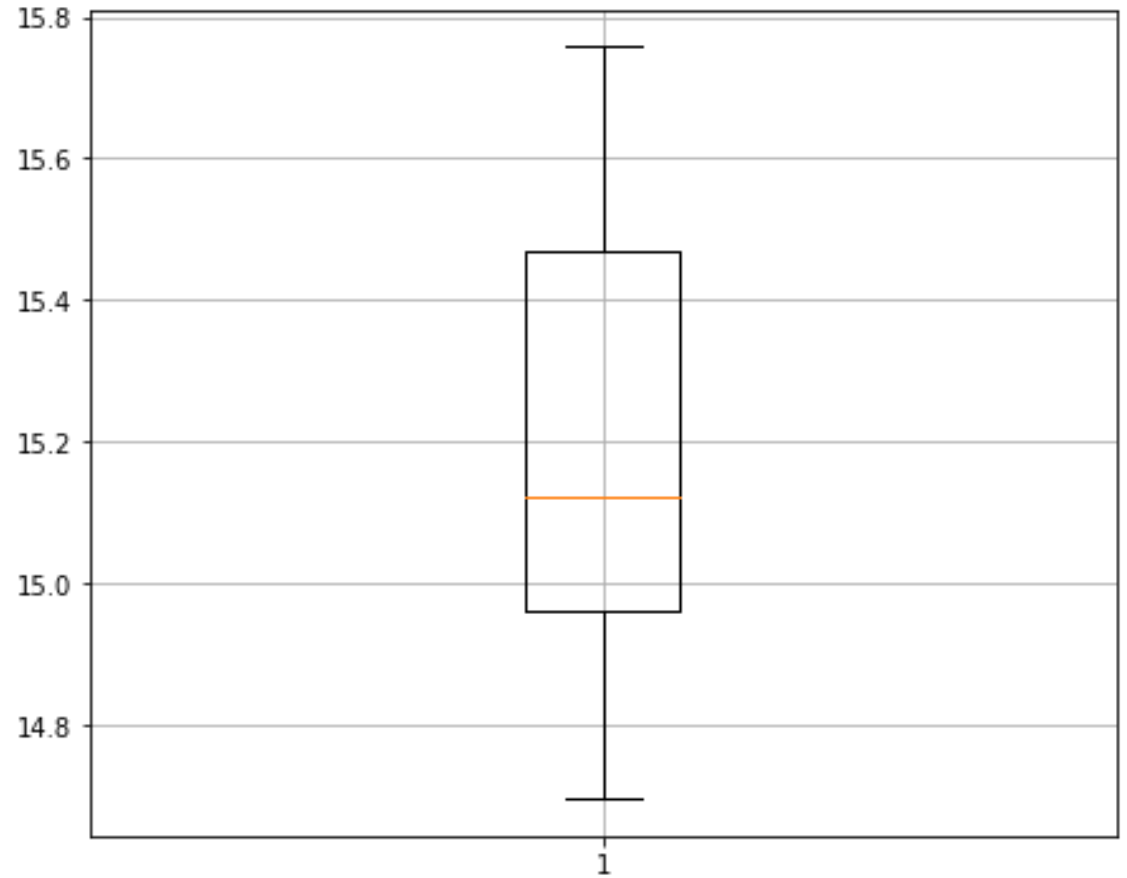
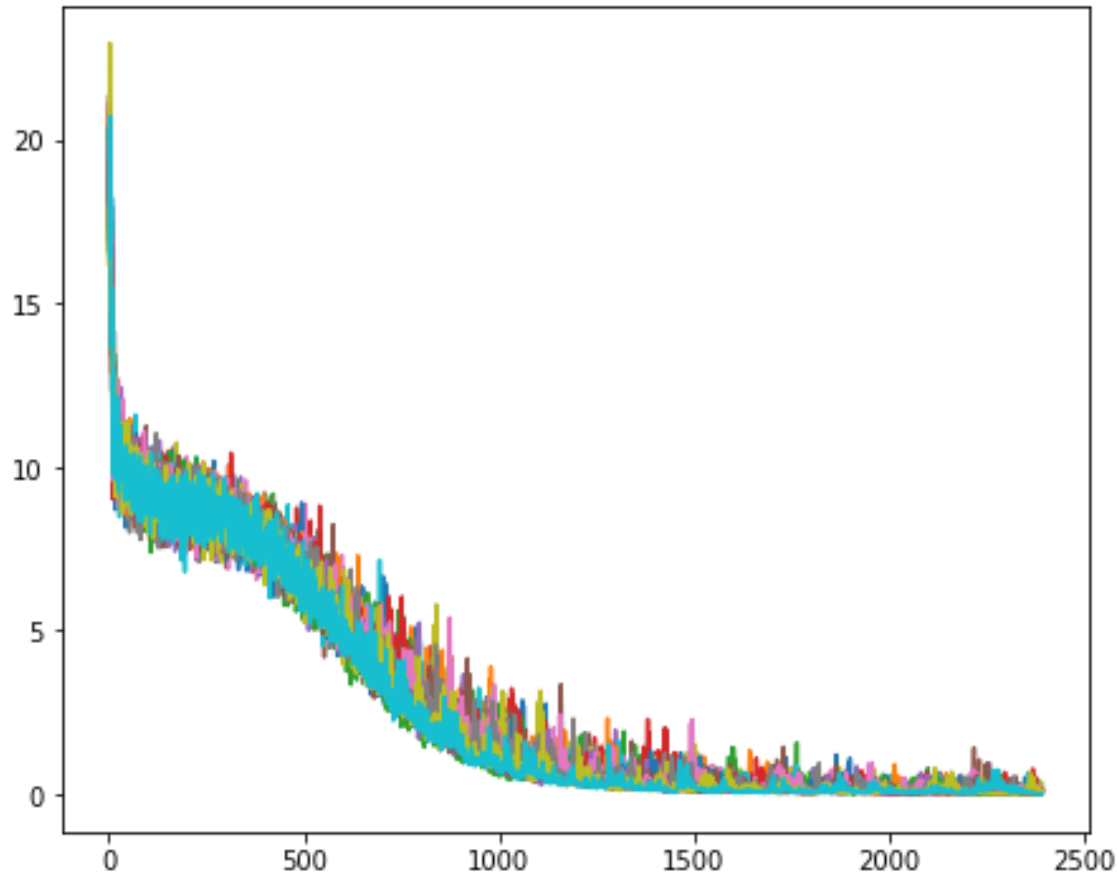
(2) Regularization – L2

Regularization may not be that important in deep learning as in other machine learning algorithms like SVM.

```
# initialize NN weights
for name, param in MyNet.named_parameters():
    if(param.requires_grad):
        torch.nn.init.normal_(param, mean=0.0, std=0.02)
loss_func = torch.nn.MSELoss()
optimizer = torch.optim.Adam(MyNet.parameters(), lr=0.0003, weight_decay=0.0001)
```

(2) Regularization – L2

Regularization may not be that important in deep learning as in other machine learning algorithms like SVM.



(2) Regularization – L1

L1 regularization:

$$\|\theta\|_1 = |w_1| + |w_2| + \dots$$

$$L'(\theta) = L(\theta) + \lambda \frac{1}{2} \|\theta\|_1 \quad \frac{\partial L'}{\partial w} = \frac{\partial L}{\partial w} + \lambda \operatorname{sgn}(w)$$

Update:

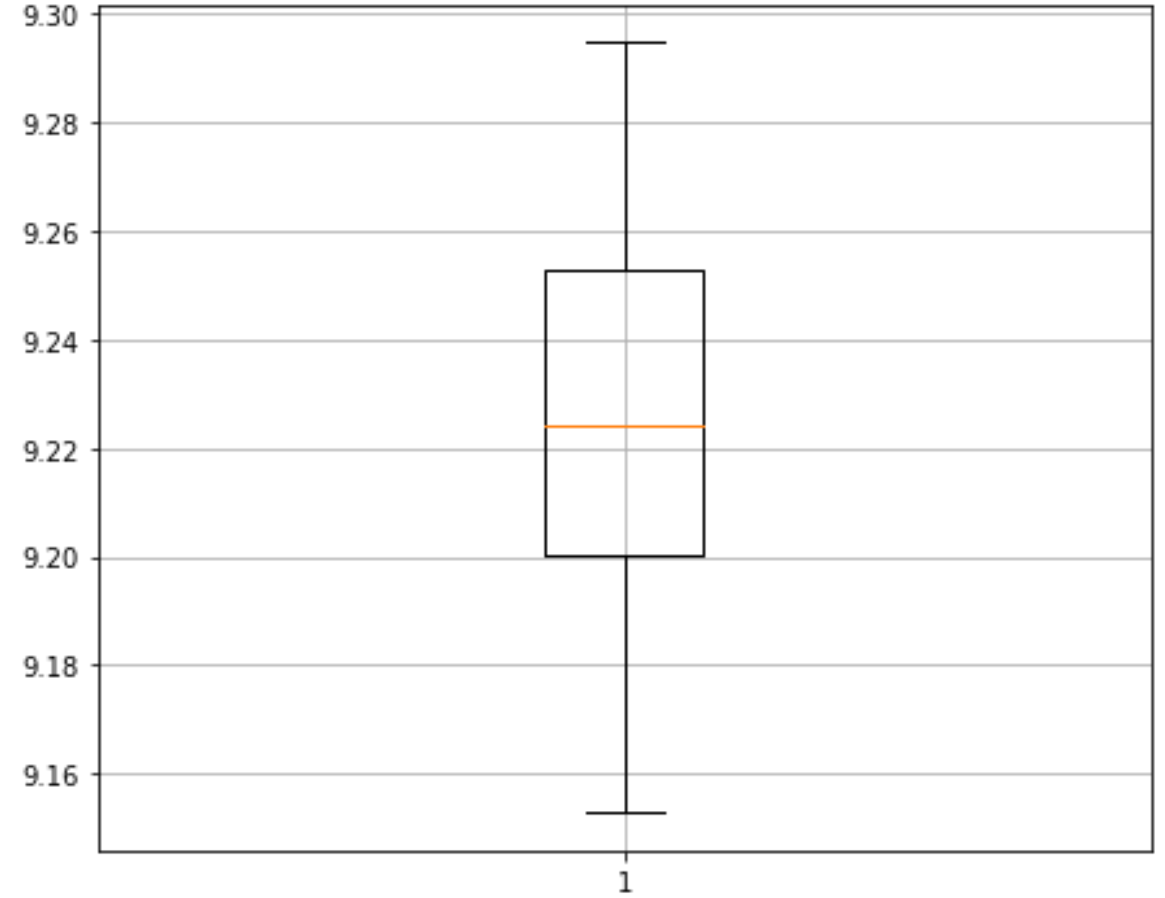
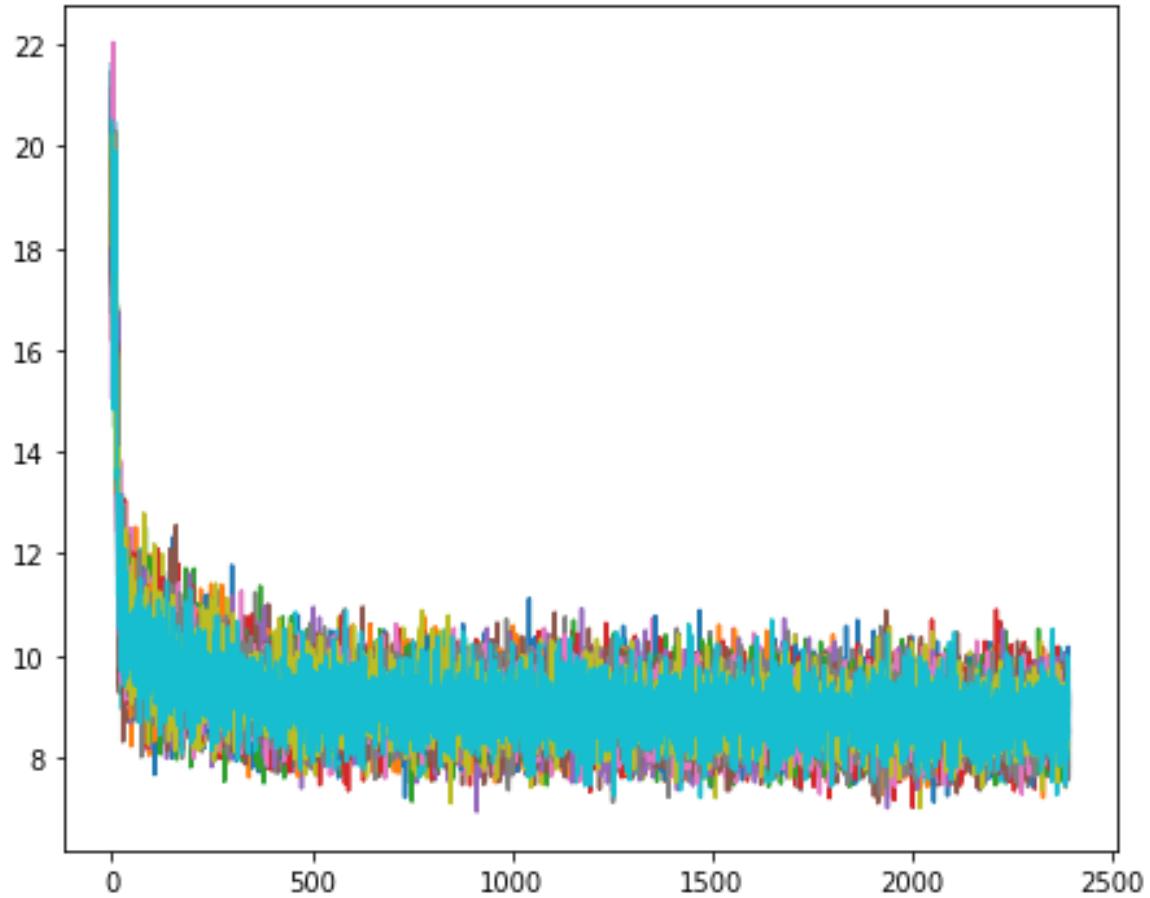
$$\begin{aligned} w^{t+1} &\rightarrow w^t - \eta \frac{\partial L'}{\partial w} = w^t - \eta \left(\frac{\partial L}{\partial w} + \lambda \operatorname{sgn}(w^t) \right) \\ &= w^t - \eta \frac{\partial L}{\partial w} - \underbrace{\eta \lambda \operatorname{sgn}(w^t)}_{\text{Always delete}} \\ &= (1 - \eta \lambda) w^t - \eta \frac{\partial L}{\partial w} \quad \text{..... L2} \end{aligned}$$

(2) Regularization – L1

```
for (batchX, batchY) in loader:
    batchY_hat = MyNet(batchX)
    loss = loss_func(batchY_hat, batchY)
    epoch_lossLst.append(float(loss))

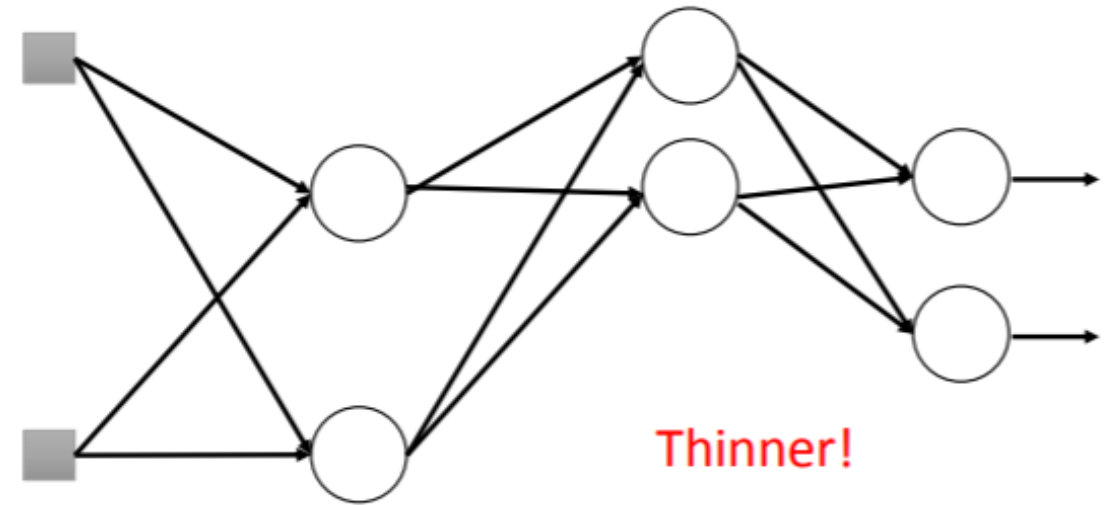
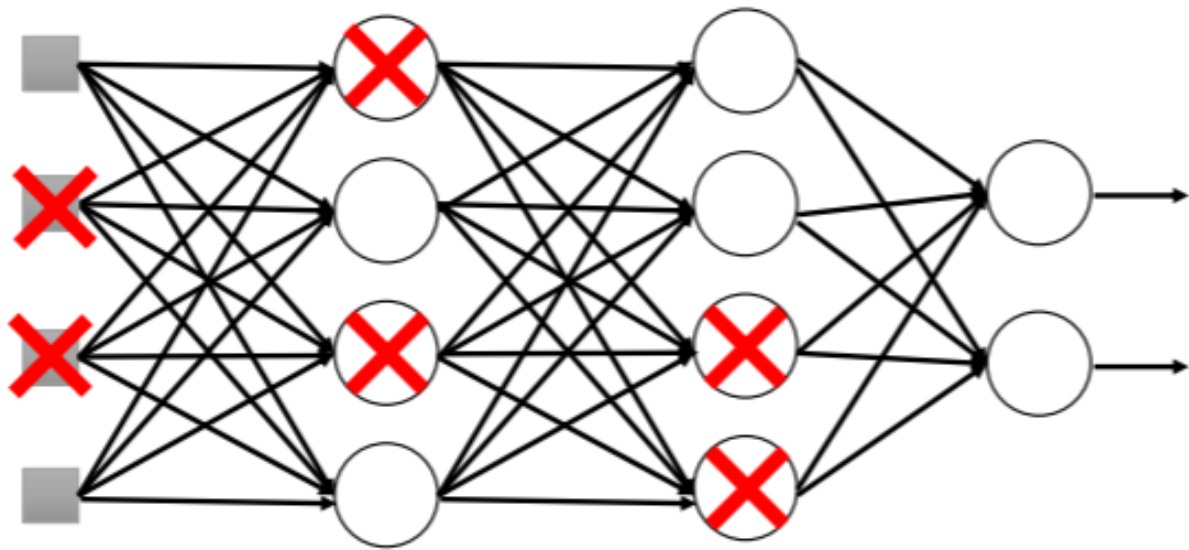
    # add L1 regularization
    regularization_loss = 0
    for name, param in MyNet.named_parameters():
        if('weight' in name):
            regularization_loss += torch.sum(abs(param))
    loss = loss + lamda * regularization_loss
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```


(2) Regularization – L1



(3) Drop out

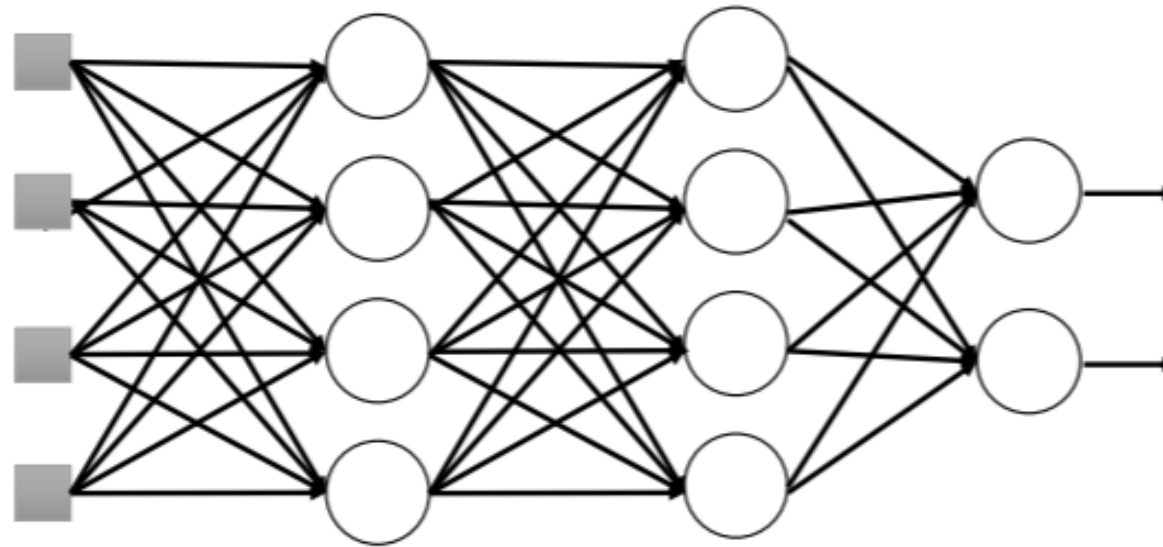
Each time before updating θ , each neuron has $p\%$ to dropout. So the NN structure is changed (become thinner). That is, for each mini-batch, we resample the dropout neurons.



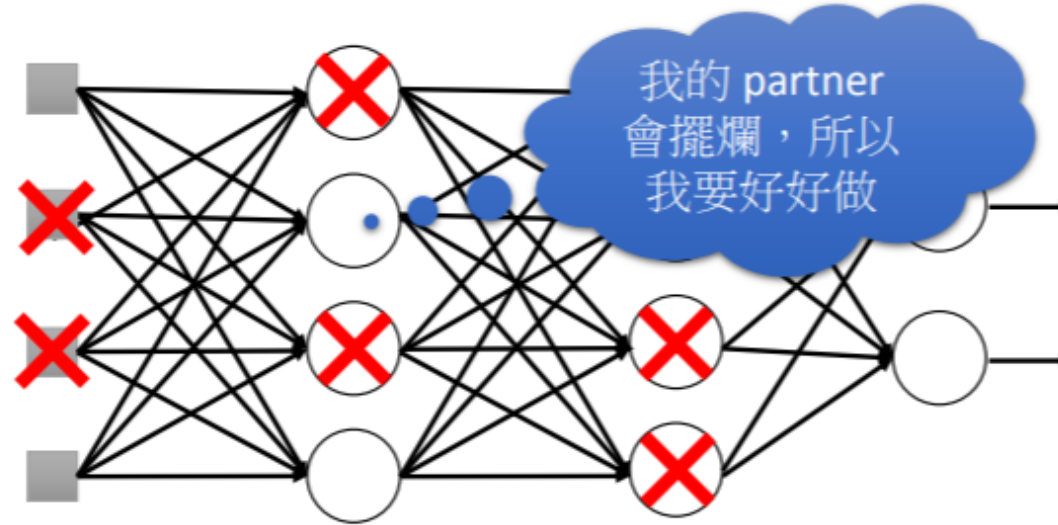
(3) Drop out

No neuron drop out at test stage. All weights time $1 - p\%$

Testing:



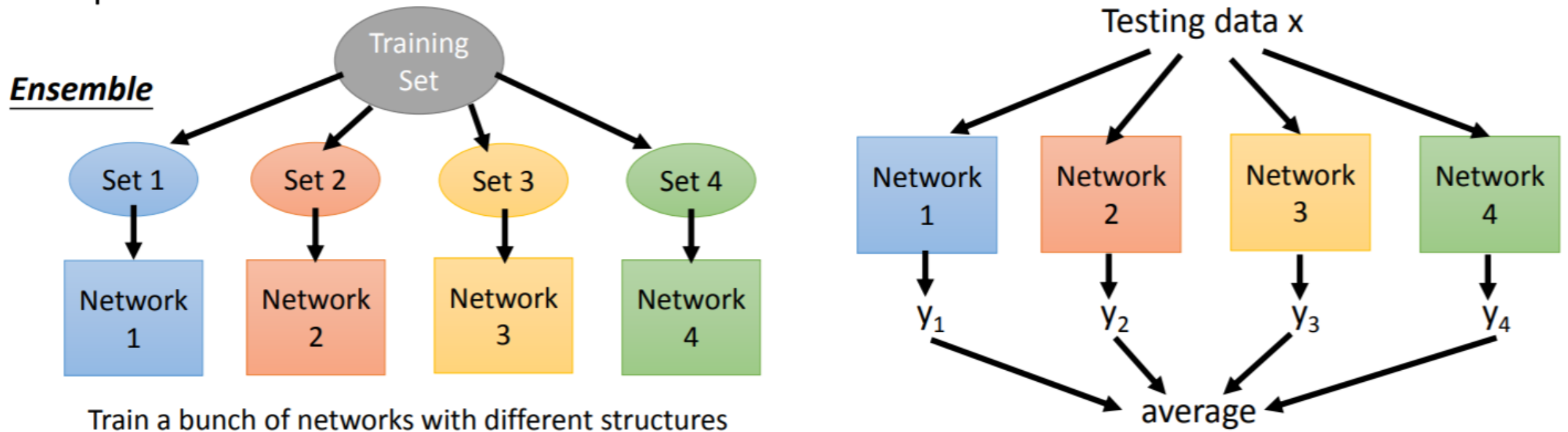
Why drop out makes NN perform better?



- When teams up, if everyone expect the partner will do the work, nothing will be done finally.
- However, if you know your partner will dropout, you will do better.
- When testing, no one dropout actually, so obtaining good results eventually.

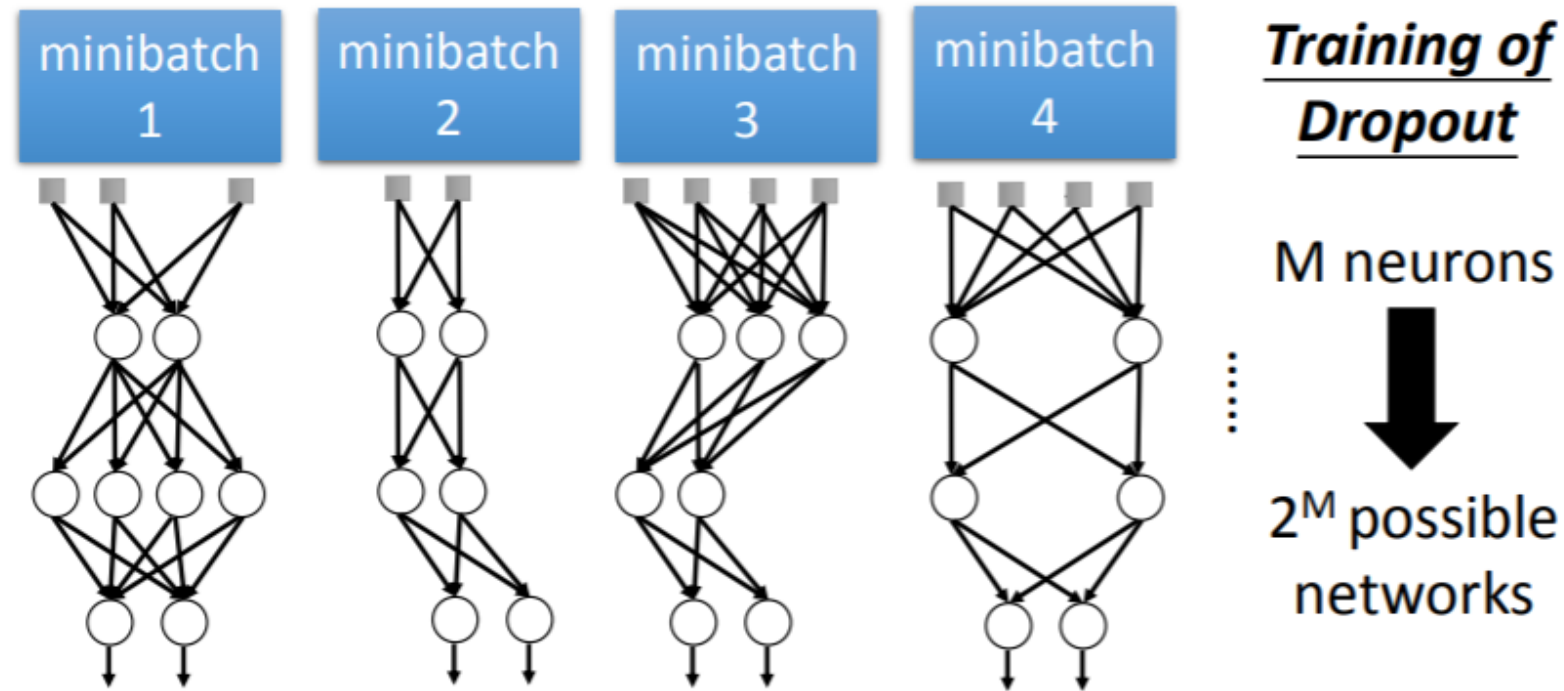
Why drop out makes NN perform better?

Drop out can be seen as an ensemble method.



Why drop out makes NN perform better?

Drop out can be seen as an ensemble method.



- Using one mini-batch to train one network
- Some parameters in the network are shared

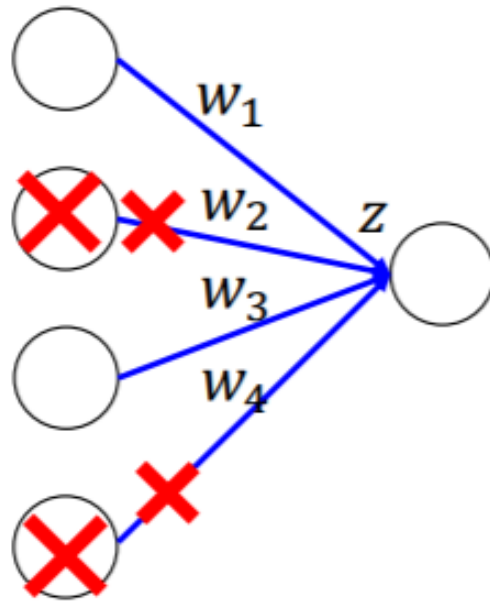
Reference: 李弘毅 ML Lecture 9-1 <https://youtu.be/xki61j7z-30>

Why multiply weights by (1-p)% during testing?

- Why the weights should multiply (1-p)% (dropout rate) when testing?

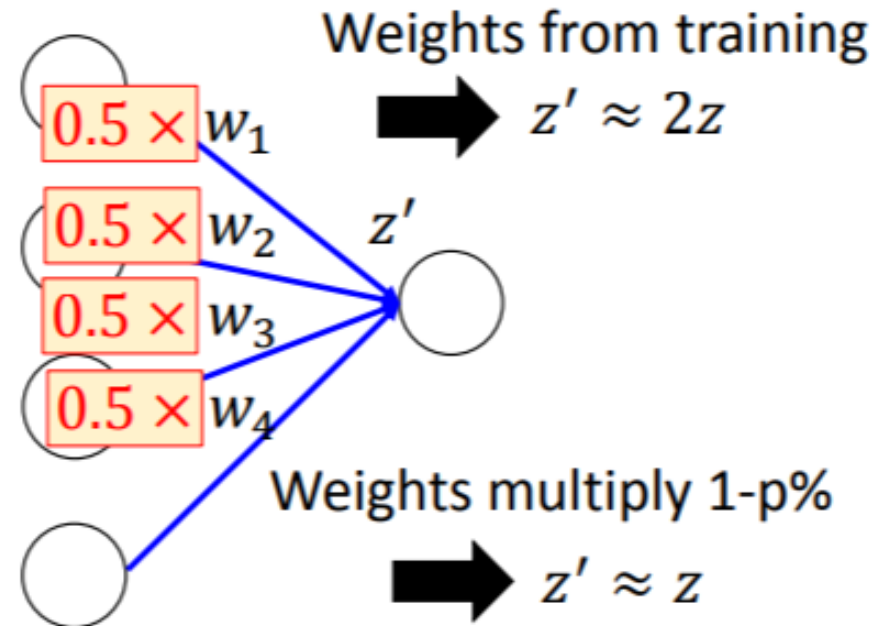
Training of Dropout

Assume dropout rate is 50%

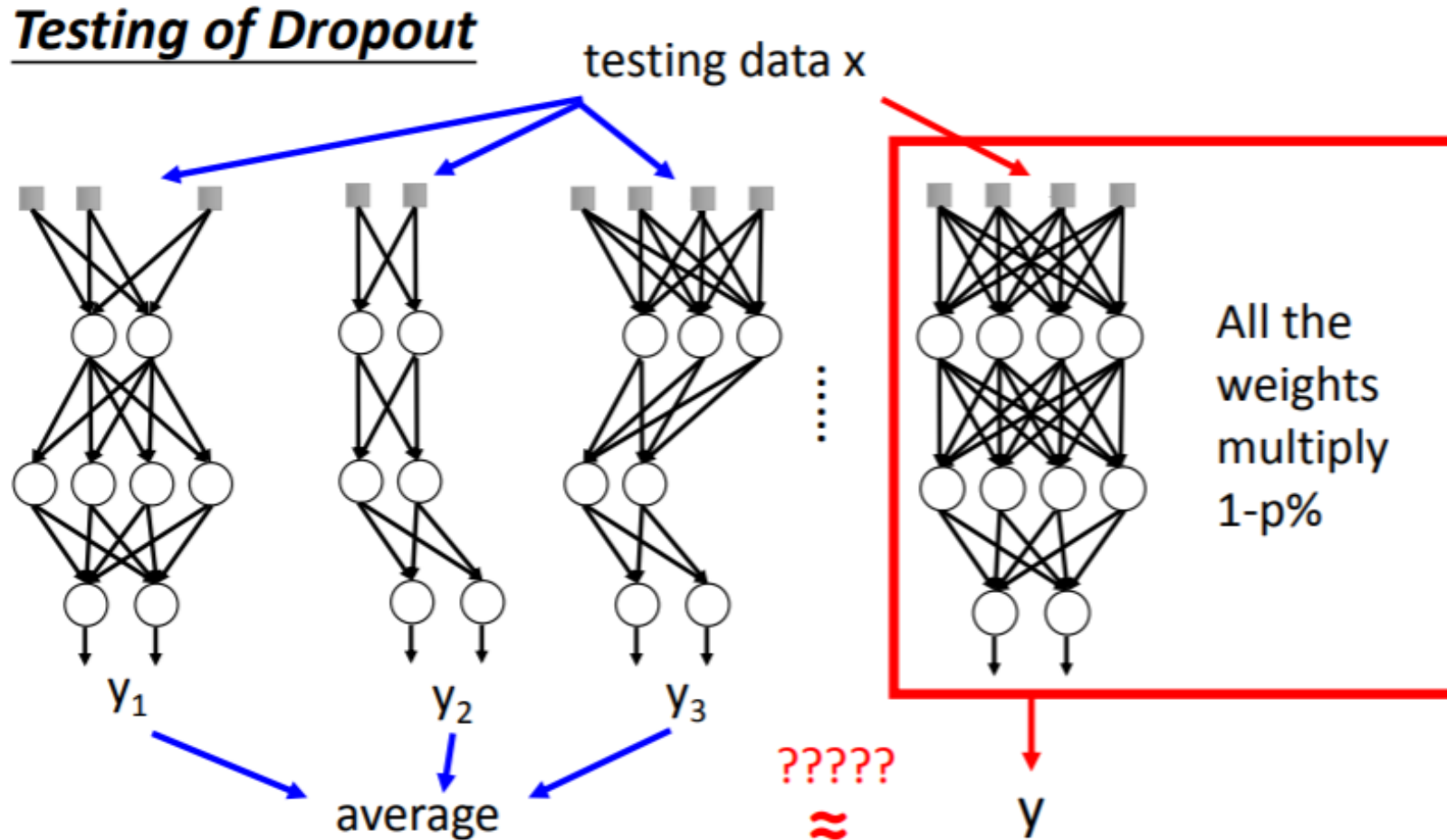


Testing of Dropout

No dropout

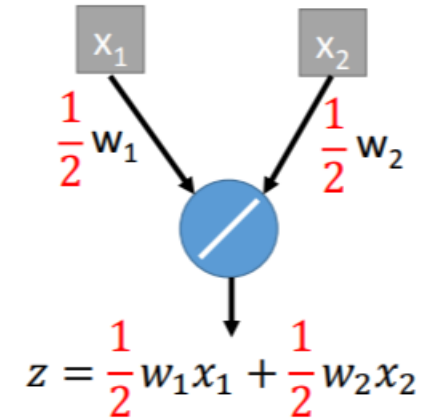
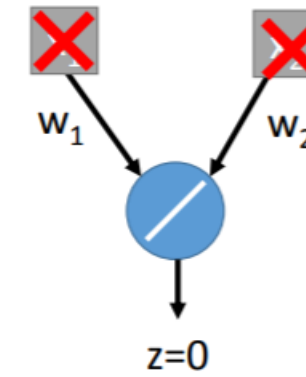
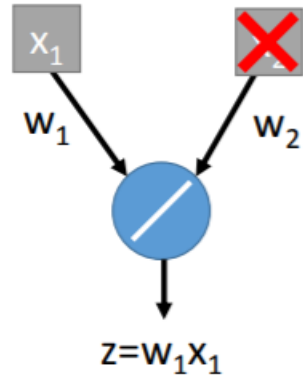
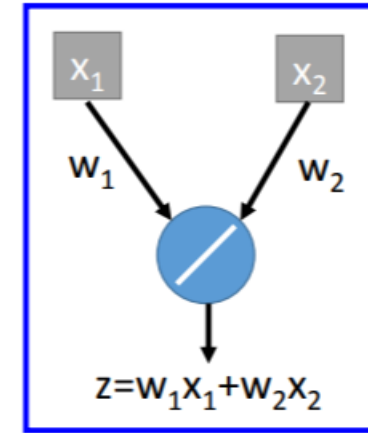
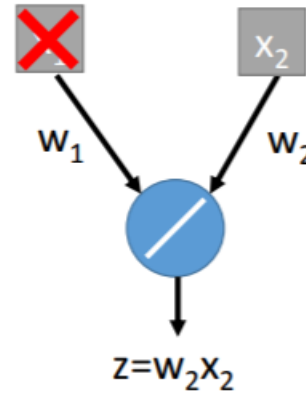
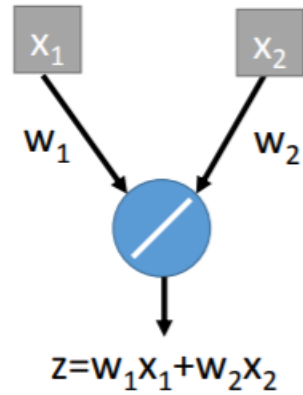


Why multiply weights by $(1-p)\%$ during testing?



Why multiply weights by (1-p)% during testing?

Testing of Dropout

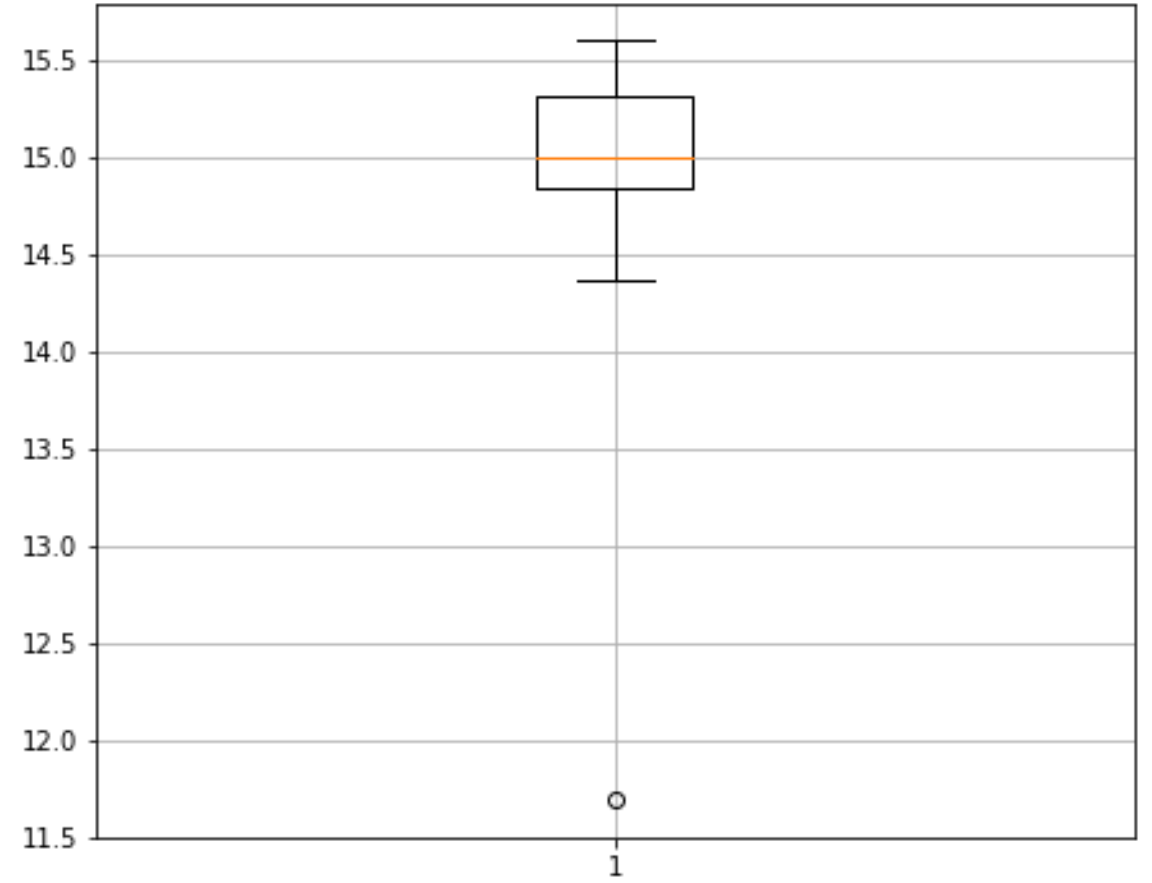
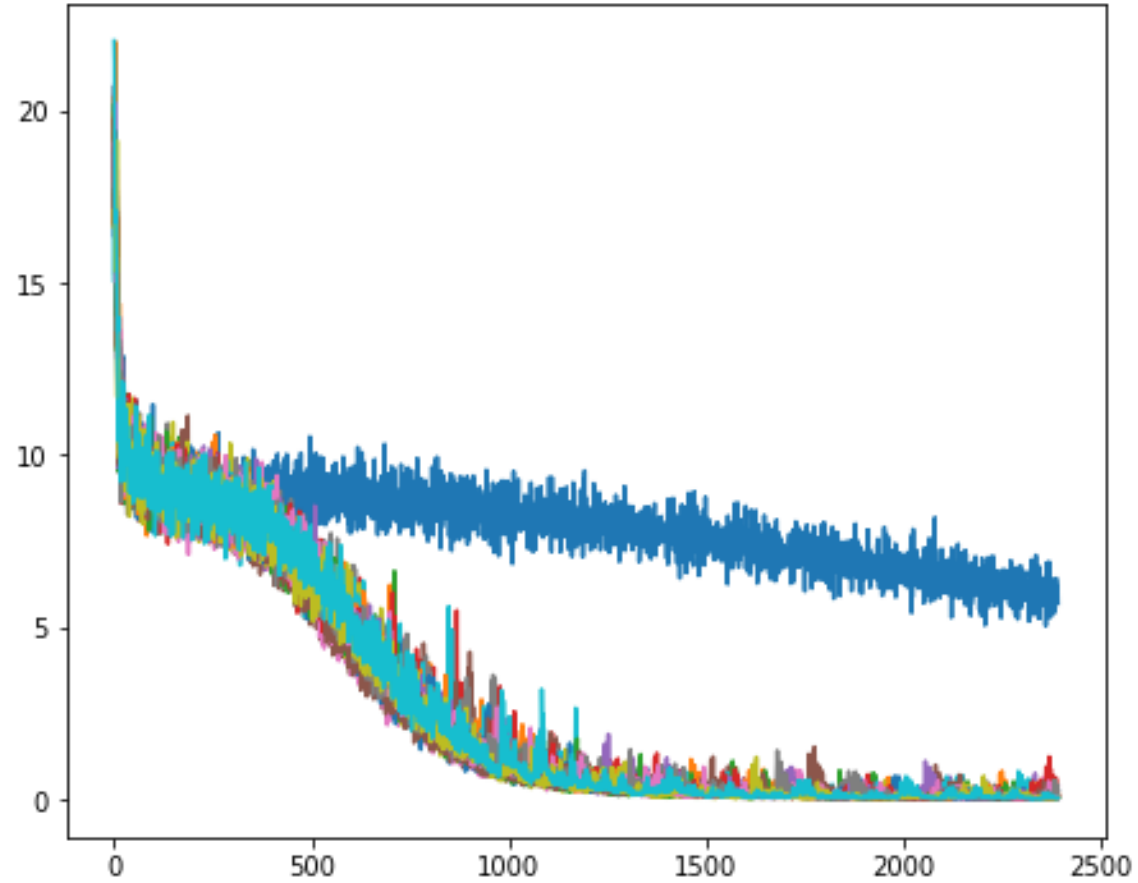


(3) Drop out

```
MyNet = nn.Sequential(  
    nn.Linear(7, 1024),  
    nn.ReLU(),  
    nn.Dropout(p=0.5),  
    nn.Linear(1024, 1024),  
    nn.ReLU(),  
    nn.Dropout(p=0.5),  
    nn.Linear(1024, 1024),  
    nn.ReLU(),  
    nn.Dropout(p=0.5),  
    nn.Linear(1024, 1024),  
    nn.ReLU(),  
    nn.Dropout(p=0.5),  
    nn.Linear(1024, 1024),  
    nn.ReLU(),  
    nn.Dropout(p=0.2),  
    nn.Linear(1024, 1),  
)  
MyNet.to(device)  
print(MyNet)
```

```
#model.eval() will turn model to test mode and PyTorch will  
# automatically handle weight scaling of the dropout layer  
MyNet.eval()  
tensorX = torch.FloatTensor(testX).to(device)  
tensorY = torch.FloatTensor(testY).to(device)  
tensorY_hat = MyNet(tensorX)  
loss = loss_func(tensorY, tensorY_hat)  
test_lossLst.append(float(loss))
```

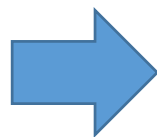
(3) Drop out



Try different NNs

Use your own $y = f(\vec{x})$ data

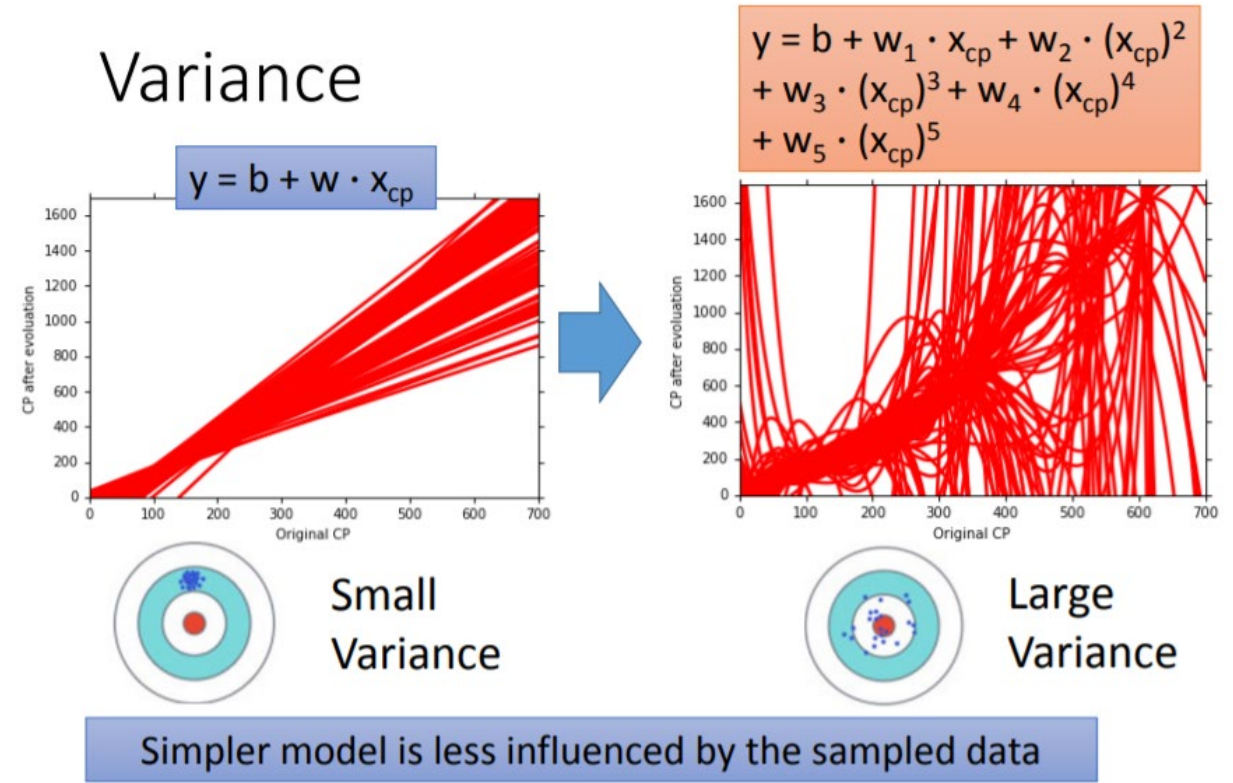
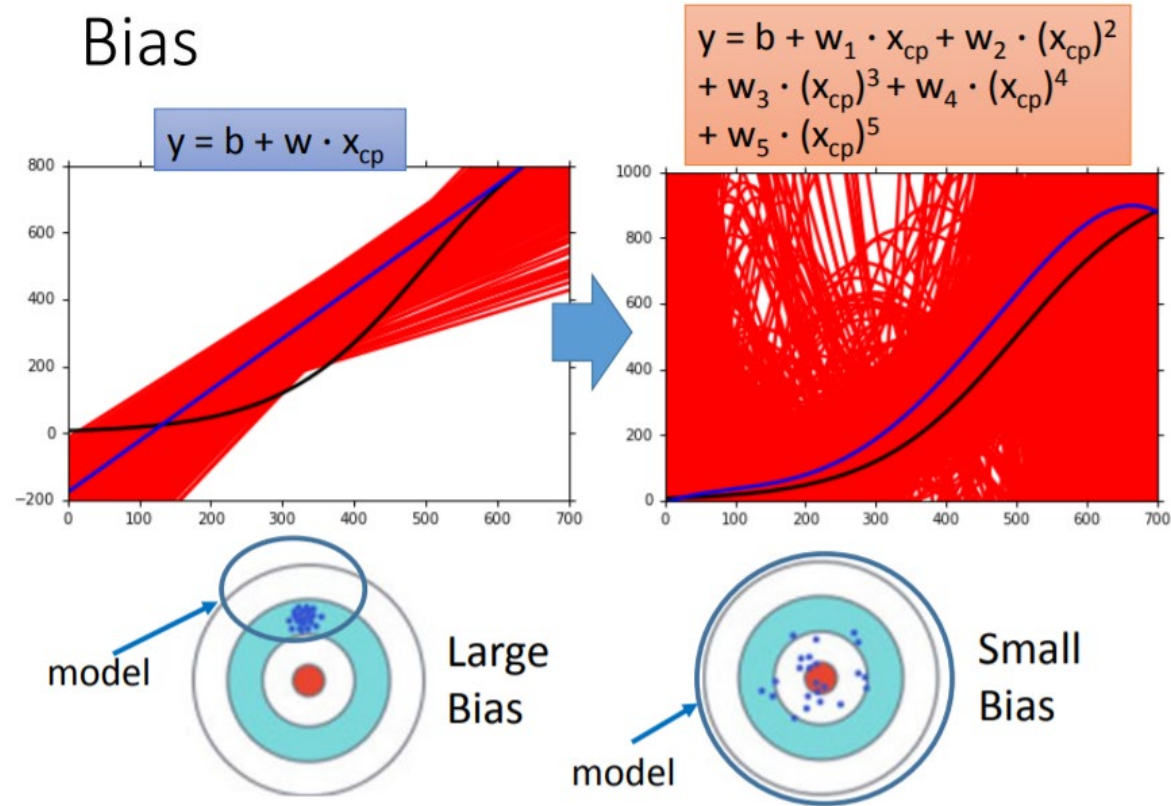
- 7-512-512-512-1
- 7-256-256-256-1
- 7-56-56-1



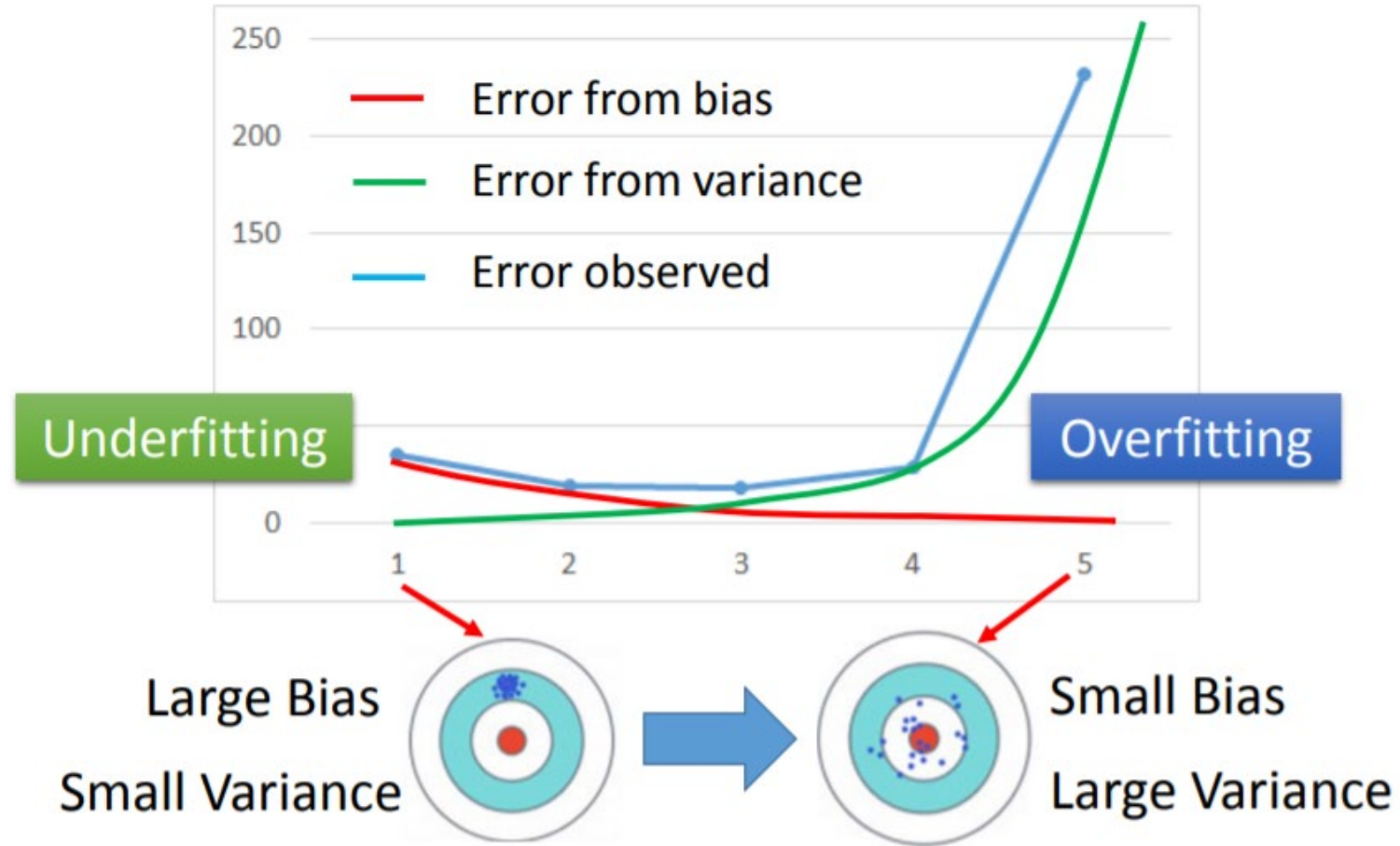
- Is the overfitting problem solved by drop out?

Try different NNs

Simpler model is less influenced by the sampled data and has smaller variance



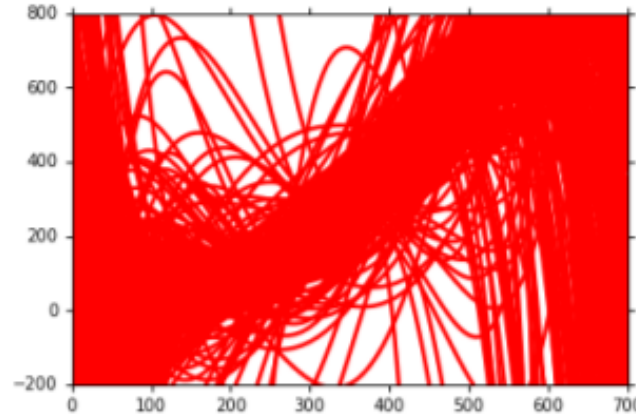
Try different NNs



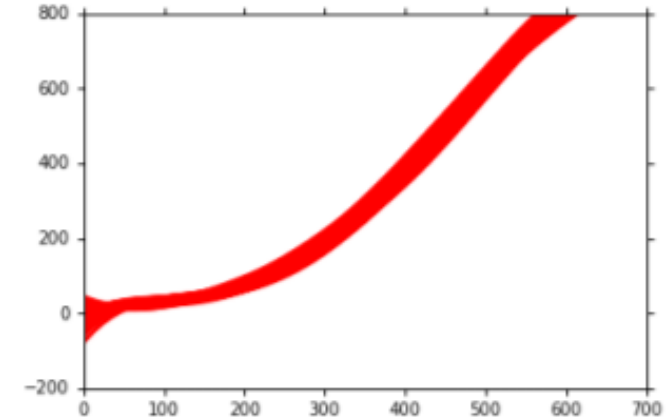
Try different NNs

① More data

Very effective,
but not always
practical



10 examples



100 examples

② Regularization

