UNIT 2

Deep Networks

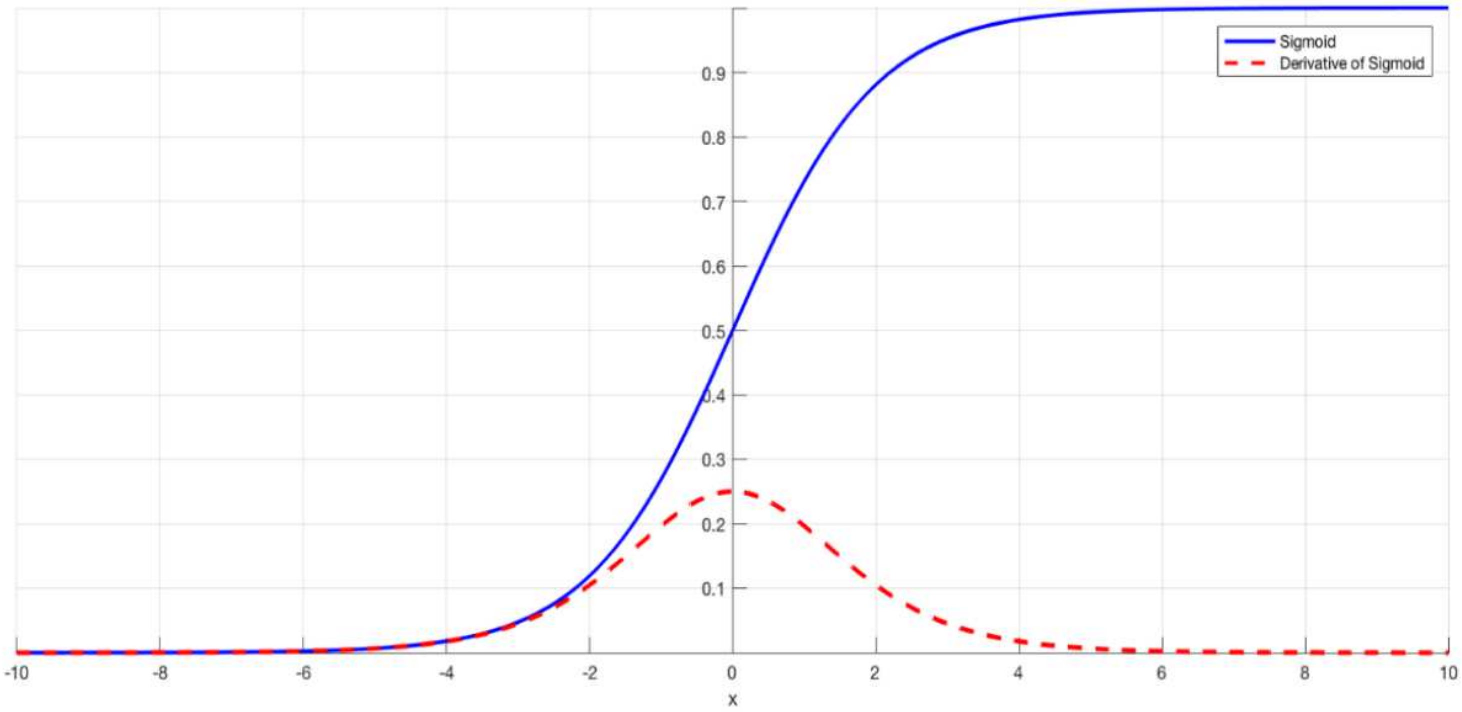
Vanishing Gradients

The Problem: As more layers using certain activation functions are added to neural networks, the gradients of the loss function approaches zero, making the network hard to train.

Reason: Use of Sigmoid activation function.

- The activation function sigmoid, squishes a large input space into a small input space between 0 and 1.
- Therefore, a large change in the input of the sigmoid function will cause a small change in the output. Hence, the derivative becomes small

Vanishing Gradients (Sigmoid and its derivative)



Vanishing Gradients

- With the sigmoid function, when the inputs of the sigmoid function

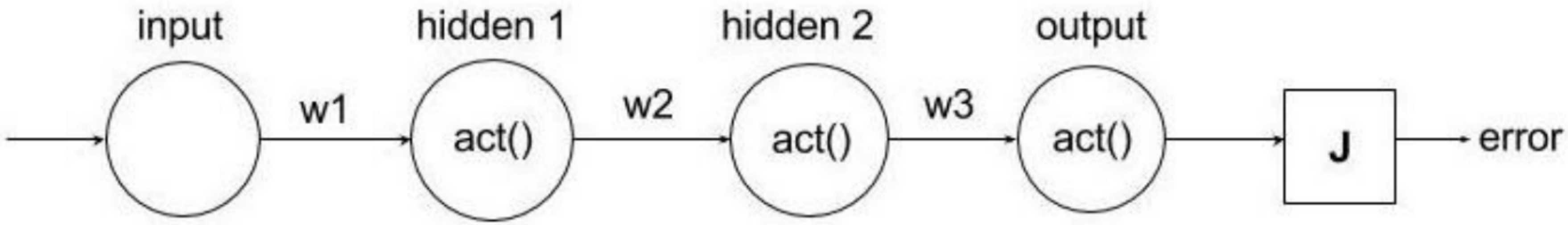
become larger or smaller the derivative becomes close to zero, the range is from 0 to 0.25.

- Gradients of neural networks are found using backpropagation. i.e. Backpropagation finds the derivatives of the network by moving layer by layer from the final layer to the initial one.
- Using chain rule, the derivatives of each layer are multiplied down the network (from the final layer to the initial) to compute the derivatives of the initial layers.

Vanishing Gradients

- When n hidden layers use an activation like the sigmoid function, n small derivatives are multiplied together.
- Thus, the gradient decreases exponentially as we propagate down to the initial layers.

Vanishing Gradients (Example)

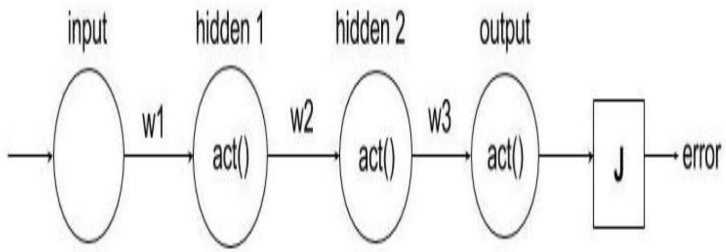


With above ANN structure we perform backpropagation to modify the weights through gradient descent such that the output of J is minimized.

Vanishing Gradients (Example)

- To calculate the derivative to the first weight, we used the chain rule to “backpropagate” like so:

$$\frac{\partial error}{\partial w1} = \frac{\partial error}{\partial output} * \frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1} * \frac{\partial hidden1}{\partial w1}$$

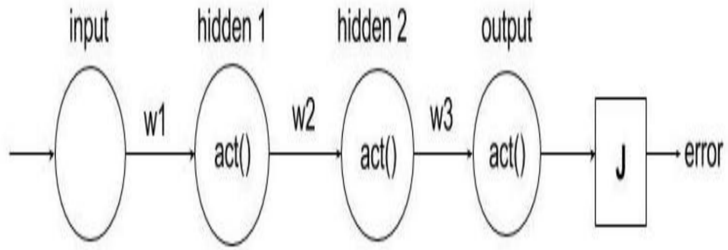


Vanishing Gradients (Example)

$$\frac{\partial error}{\partial w1} = \frac{\partial error}{\partial output} * \frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1} * \frac{\partial hidden1}{\partial w1}$$

- If we see individual derivatives, we have

$$\frac{\partial output}{\partial hidden2} * \frac{\partial hidden2}{\partial hidden1}$$



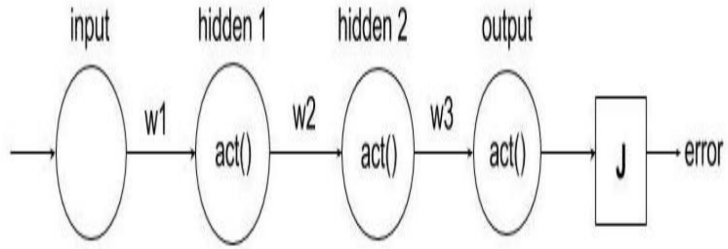
Vanishing Gradients (Example)

$$z_1 = hidden2 * w3$$

$$\frac{\partial output}{\partial hidden2} = \frac{\partial Sigmoid(z_1)}{\partial z_1} w3$$

$$z_2 = hidden1 * w2$$

$$\frac{\partial hidden2}{\partial hidden1} = \frac{\partial Sigmoid(z_2)}{\partial z_2} w2$$



Vanishing Gradients (Example)

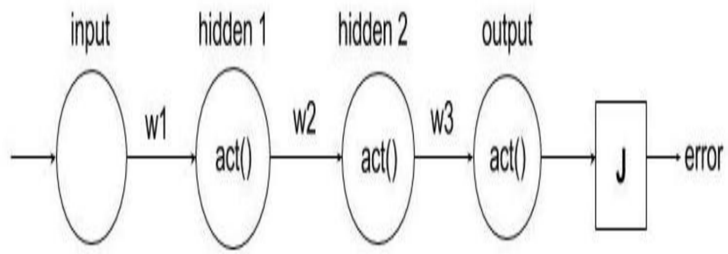
$$\frac{\partial output}{\partial hidden2}\frac{\partial hidden2}{\partial hidden1} = \frac{\partial Sigmoid(z_1)}{\partial z_1} w3 * \frac{\partial Sigmoid(z_2)}{\partial z_2} w2$$

Vanishing Gradients (Example)

- Recall that the derivative of the sigmoid function outputs values between 0 and 1/4.
- By multiplying these two derivatives together, we are multiplying two values in the range (0, 1/4].

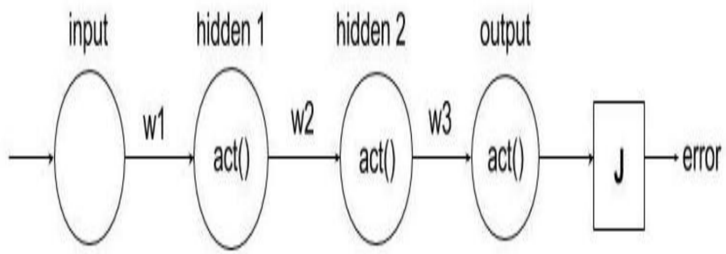
  ==> result in a smaller value

- During the weight updating in backpropagation, the new weight and old weight will be closure to same



Vanishing Gradients (Example)

- A small gradient means that the weights and biases of the initial layers will not be updated effectively with each training session.
- Since these initial layers are often crucial to recognizing the core elements of the input data, it can lead to overall inaccuracy of the whole network.
- with deep neural nets, the vanishing gradient problem becomes a major concern.



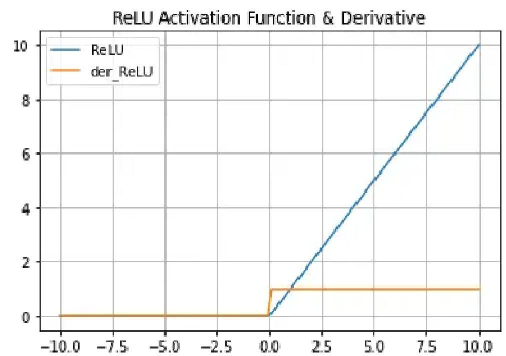
Vanishing Gradients (Some Solutions)

- The solution to minimize the Vanishing Gradients Problem
  - Use of other activation functions, such as ReLU or its family,which doesn't cause a small derivative.
- Using appropriate weight initialization techniques

$$ReLU(x)=max(0,x)$$

Derivative

$$f(x) = \begin{cases} 1, x > 0 \\ 0, \; otherwise \end{cases}$$



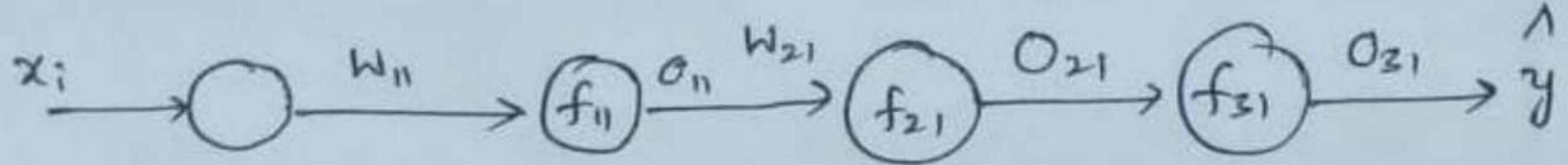
Exploding Gradients problem

- In deep networks or recurrent neural networks, error gradients can accumulate during an update and result in very large gradients.
- These in turn result in large updates to the network weights, and in turn, an unstable network.
- The explosion occurs through exponential growth by repeatedly multiplying gradients through the network layers that have values larger than 1.0.
- Exploding gradients can make learning unstable

Exploding Gradients problem

The higher weights in neural network leads to exploding gradients Problem.

Consider the following part of neural network



Using chain rule of back propagation

$$\frac{\partial L}{W_{11}} = \frac{\partial O_{31}}{\partial O_{21}} \cdot \frac{\partial O_{21}}{\partial O_{11}} \cdot \frac{\partial O_{11}}{\partial w_{11}}$$

Exploding Gradients problem

Let the $\frac{\partial O_{21}}{\partial O_{11}}$ is computed as

$$\frac{\partial O_{21}}{\partial O_{11}} = \frac{\partial \psi(z)}{\partial z} \cdot \frac{\partial z}{\partial O_{11}}$$

Here $z = W_{21} \cdot O_{11} + bias$

$O_{21} = \psi(z)$.

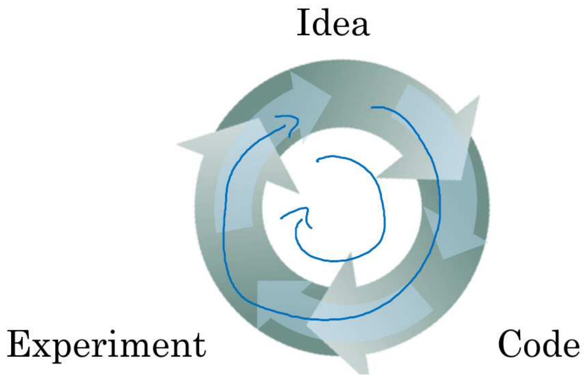
$\uparrow$ Activation function

if Activation function is segmoid.

$\frac{\partial \psi(z)}{\partial z}$ is in the range 0 to 0.25.

Hyperparameter Tuning

- Deep learning Model development is highly iterative process
- Hyperparameters:
  - # layers
  - # hidden units
  - learning rates
  - activation functions



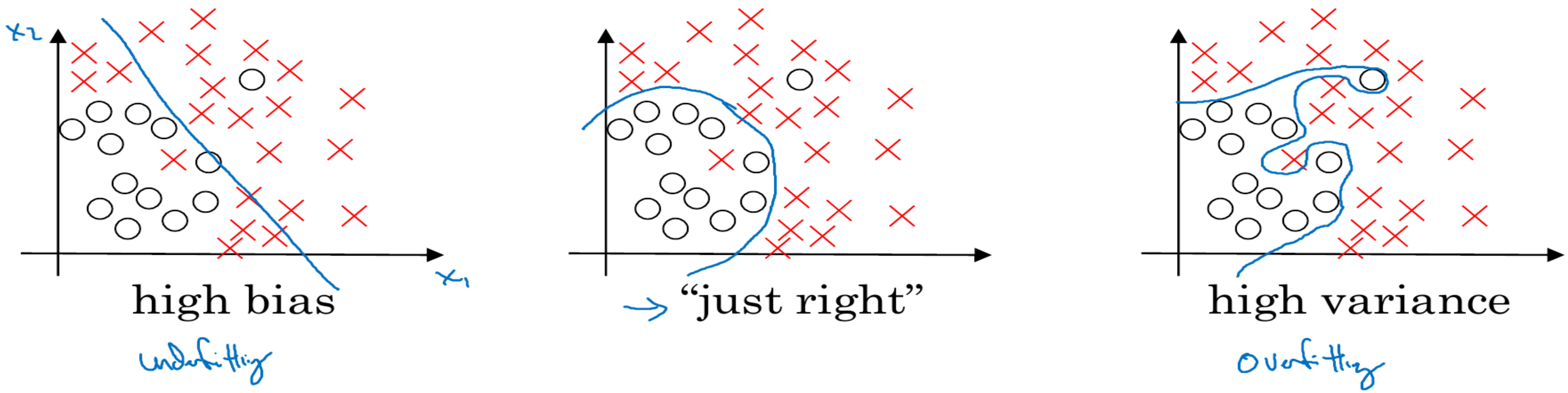
- Application areas
  - NLP
  - Vision
  - Speech
- Manually experimenting hyperparameters setting is difficult.

Train/Validation(or dev)/test sets

- For small Datasets : 60/20/20 split
- For Big Data:
  - For e.g 1 million training examples it is enough to have 10 thousand dev and test examples
  - Therefore 98/1/1 split

Bias/Variance

- The Goal of Deep learning model is to find efficient hyperparameters to reduce Bias and Variance
- What is Bias/Variance?



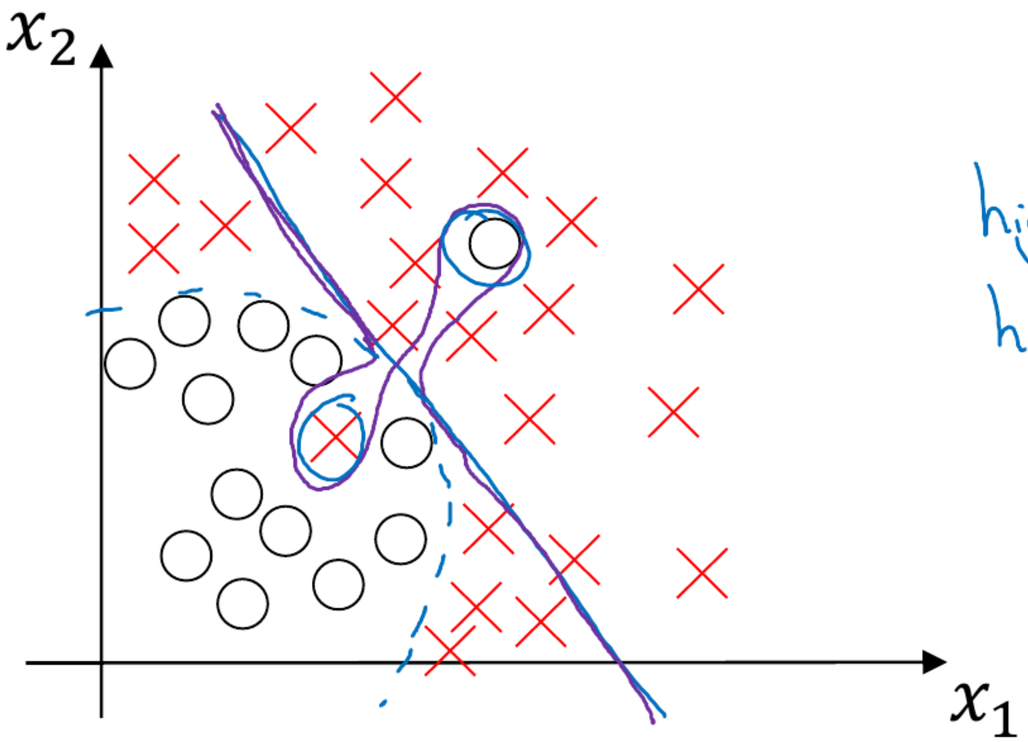
Bias/Variance

Cat classification

y = 1	y = 0
	

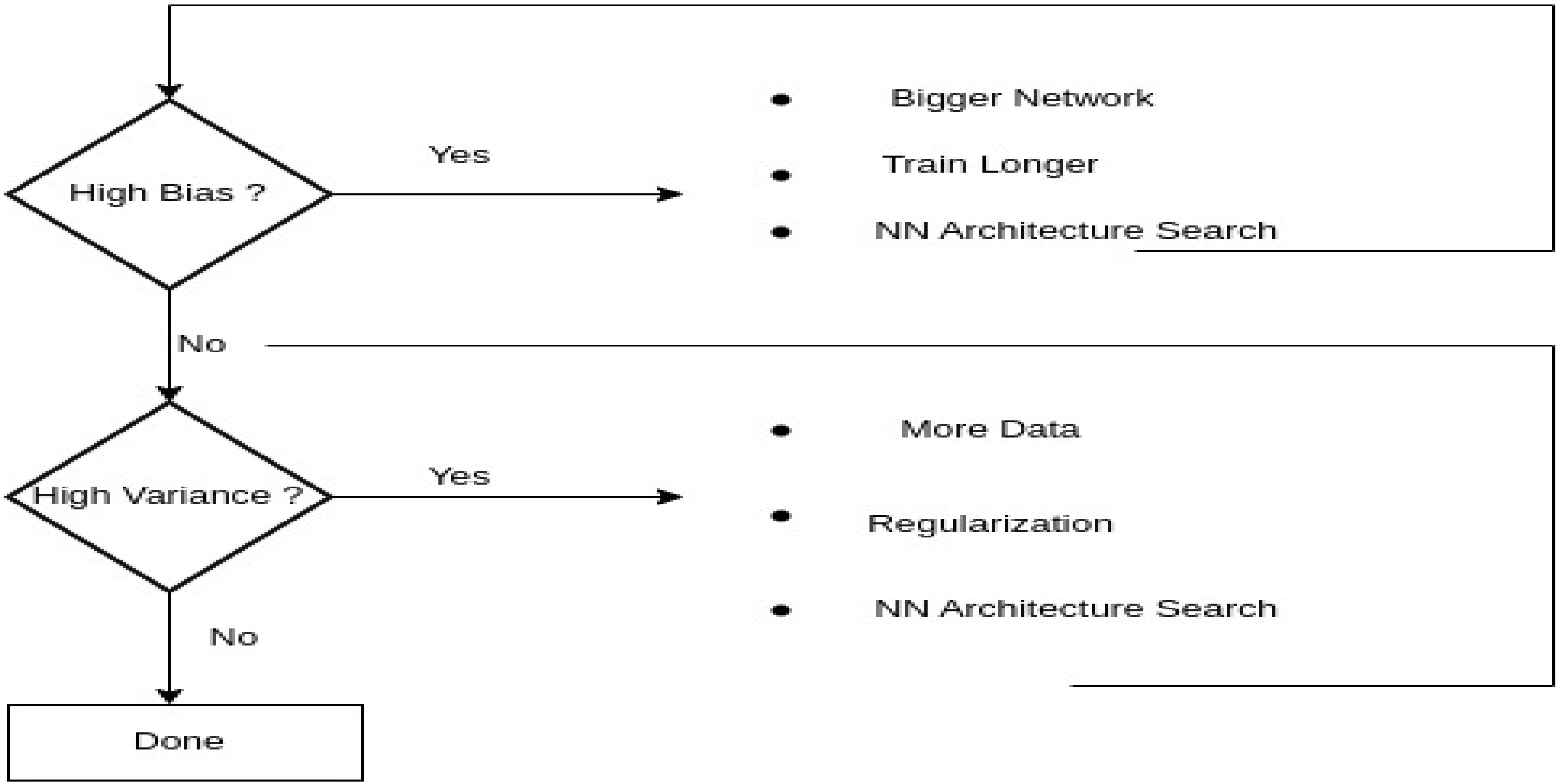
Training Set Error	1%	15%	15%	0.5%
Validation set (or Dev set) error	11%	16%	30%	1%
	Low Bias + High Variance	High Bias + Low Variance	High Bias + High Variance (Worst)	Low Bias + Low Variance (Best)

High bias and high variance



high bias
high varian

Basic Recipe for Low Bias and Low Variance



Regularization (for Logistic regression)

$$w \in \mathbb{R}^{n_x}, \quad b \in \mathbb{R}$$

$$\min_{w,b} J(w, b)$$

$$J(w,b) = \underbrace{\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})} + \frac{\lambda}{2m} \underbrace{\|w\|_2^2}$$

λ = regularization parameter

L_2 regularization $\quad \|w\|_2^2 = \sum_{j=1}^{n_x} w_j^2 = w^T w \longleftarrow$

L_1 regularization $\quad \frac{\lambda}{2m} \sum_{j=1}^{n_x} |w_j| = \frac{\lambda}{2m} \|w\|_1$

Regularization (for Neural network)

$$J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]}) = \underbrace{\frac{1}{m}\sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})} + \underbrace{\frac{\lambda}{2m}\sum_{l=1}^{L} \|w^{[l]}\|_F^2}$$

$$\|w^{[l]}\|_F^2 = \sum_{i=1}^{n^{[l]}}\sum_{j=1}^{n^{[l-1]}} \left(w_{ij}^{[l]}\right)^2 \qquad w^{[l]}: (n^{[l]}, n^{[l-1]})$$

"<u>Frobenius norm</u>" $\qquad \|\cdot\|_2^2 \qquad \|\cdot\|_F^2$

$$dw^{[l]} = \boxed{(\text{from backprop}) + \frac{\lambda}{m} w^{[l]}} \qquad \underbrace{\frac{\partial J}{\partial w^{[l]}}} = dw^{[l]}$$

$$\rightarrow w^{[l]} := w^{[l]} - \alpha\, dw^{[l]}$$

Regularization (for Neural network)

"Weight decay"

$$W^{[l]} := W^{[l]} - \alpha \left[(\text{from backprop}) + \underline{\frac{\lambda}{m} W^{[l]}} \right]$$

$$= W^{[l]} - \frac{\alpha\lambda}{m} W^{[l]} - \alpha\,(\text{from backprop})$$

$$= \underbrace{\left(1 - \frac{\alpha\lambda}{m}\right)}_{<1} \underline{W^{[l]}} - \alpha\,(\text{from backprop})$$

How does regularization prevent overfitting?

The cost function $J(w^{[1]}, b^{[1]}, \dots, w^{[L]}, b^{[L]})$

$$= \underbrace{\frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(\hat{y}^{(i)}, y^{(i)})} + \underbrace{\frac{\lambda}{2m} \sum_{l=1}^{L} \|w^{[l]}\|_F^2}$$

The weight update

$$w^{[l]} := \underbrace{\left(1 - \frac{\alpha\lambda}{m}\right)}_{<1} \underline{w^{[l]}} - \alpha\,(\text{from backprop})$$

Therefore, Higher values of lamda will punish weights

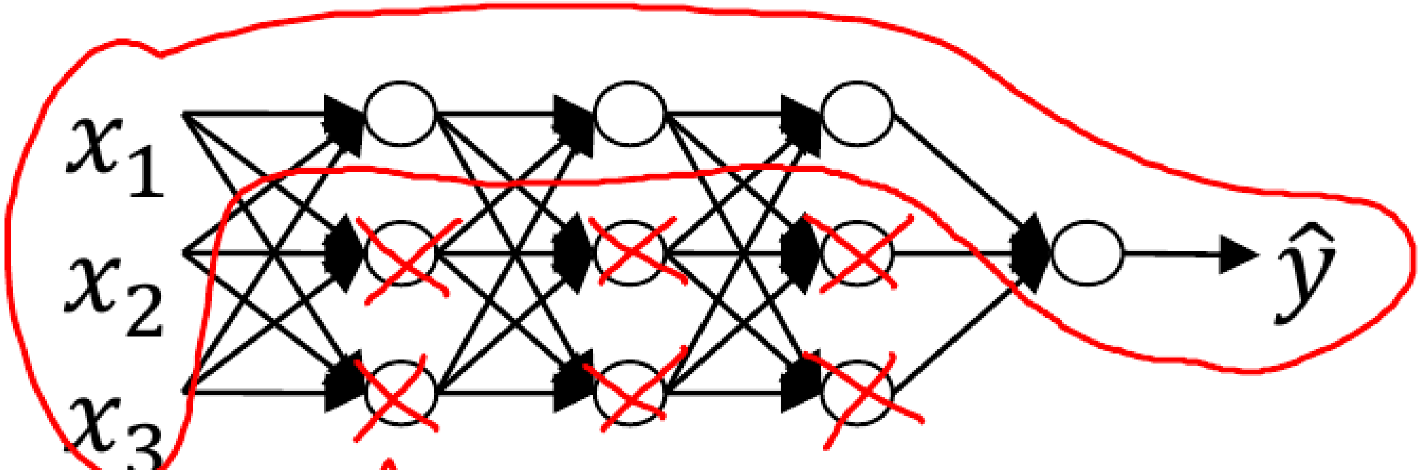
$$w^{[l]} \approx 0$$

How does regularization prevent overfitting?

As $\lambda \uparrow$ $\quad W^{[l]} \downarrow$

Then $W^{[l]} \approx 0$

Therefore,
the Network



How does regularization prevent overfitting?

- And, the Net sum at layer l

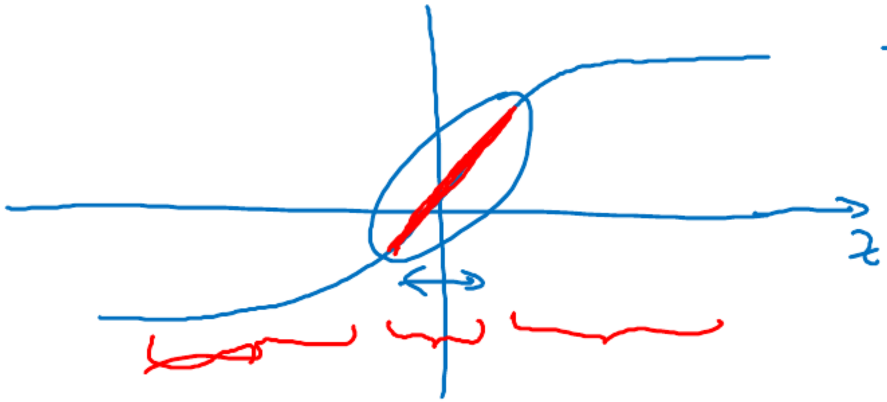
$$\underbrace{z^{[l]}}= \underline{w}^{[l]} a^{[l-1]} + \underline{b^{[l]}}$$

- So The Net sum $z^{[l]}$

becomes small

How does regularization prevent overfitting?

- As we using *tanh()* activation function, The function behaves linearly (like logistic regression)

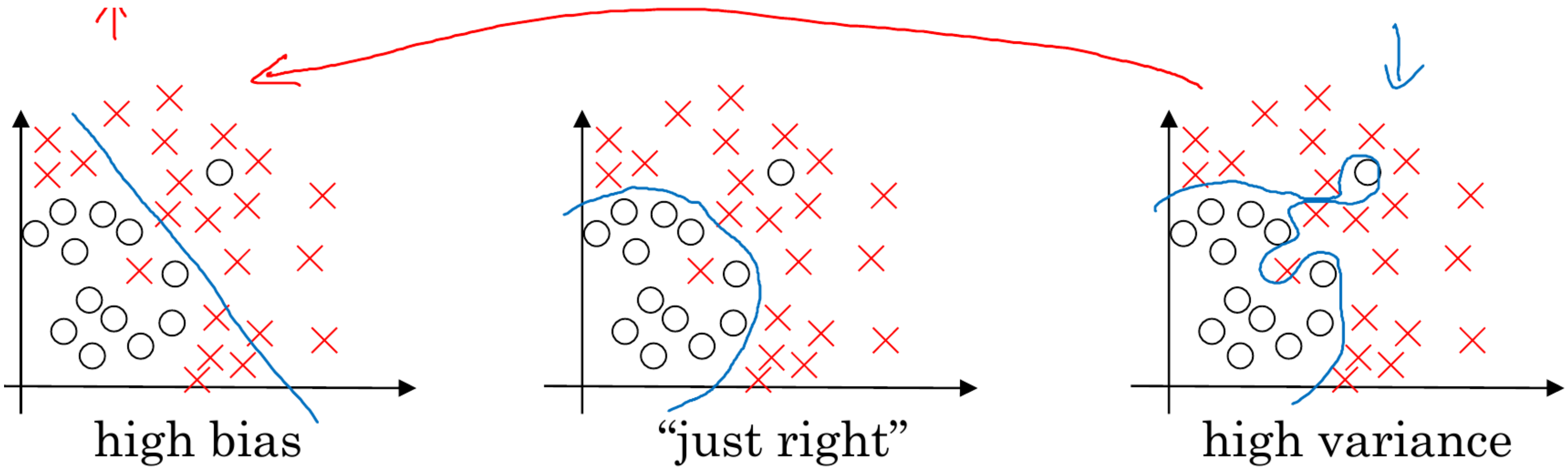


tanh

$g(z) = \tanh(z)$

How does regularization prevent overfitting?

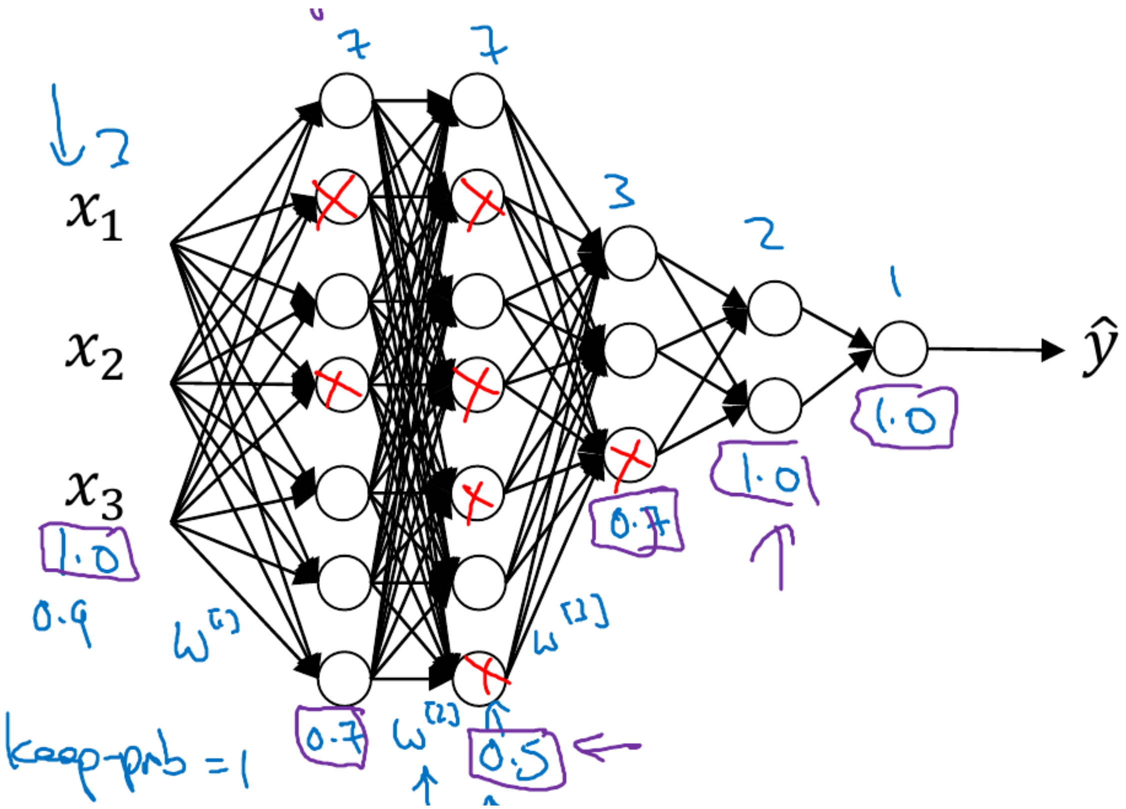
- This makes the case of High variance moving towards high bias



- So, Setting the Regularization parameter lamda efficiently not very high can put the function into the "Just right" situation

Other Regularization method(Dropout regularization)

- The objective is to shrink weights stochastically



```
# define model
model = Sequential()
model.add(Dense(500,
input_dim=2,
activation='relu'))

model.add(Dropout(0.4))
```