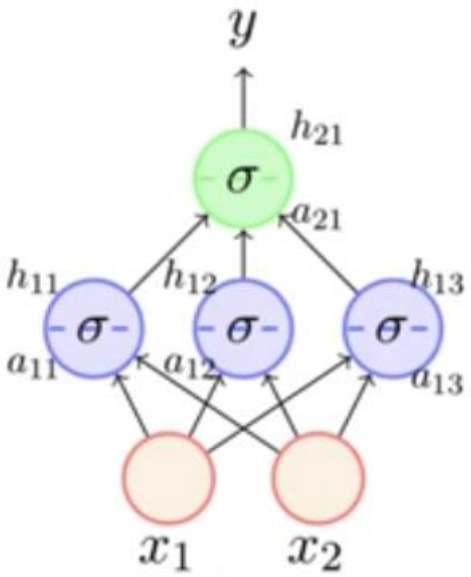
Deep Learning :
Better Initialization
& Better Regularization

राष्ट्रीय प्रौद्योगिकी संस्थान सिक्किम
NATIONAL INSTITUTE OF TECHNOLOGY SIKKIM

Course Instructor:
Dr. Bam Bahadur Sinha
Assistant Professor
Computer Science & Engineering
National Institute of Technology
Sikkim

Initialization (Why not simply initialize weights to same value ?)



Initialise w, b

Iterate over data:

compute $\hat{y}$

compute $\mathscr{L}(w, b)$

$w_{11} = w_{11} - \eta \Delta w_{11}$

$w_{12} = w_{12} - \eta \Delta w_{12}$

... ...

till satisfied

$$\nabla w_{11} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \cdot \frac{\partial y}{\partial h_{11}} \cdot \frac{\partial h_{11}}{\partial a_{11}} \cdot x_1$$

$$\nabla w_{21} = \frac{\partial \mathscr{L}(\mathbf{w})}{\partial y} \cdot \frac{\partial y}{\partial h_{12}} \cdot \frac{\partial h_{12}}{\partial a_{12}} \cdot x_1$$

$$but \quad h_{11} = h_{12}$$

$$and \quad a_{12} = a_{12}$$

$$\therefore \nabla w_{11} = \nabla w_{21}$$

$$a_{11} = w_{11}x_1 + w_{12}x_2$$

$$a_{12} = w_{21}x_1 + w_{22}x_2$$

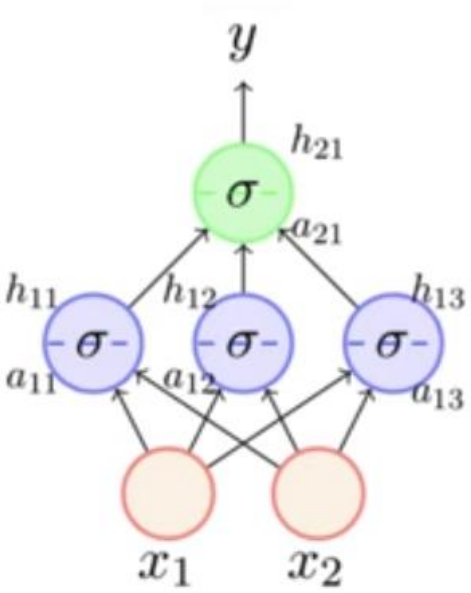
$$\therefore a_{11} = a_{12} = 0$$

$$\therefore h_{11} = h_{12}$$

- This symmetry will never break during training (**symmetry breaking problem**)
- Hence **weights** connected to the same neuron should **never be initialized to the same value**

- Never initialize all the weights to ‘0’
- Never initialize all the weights to same value.

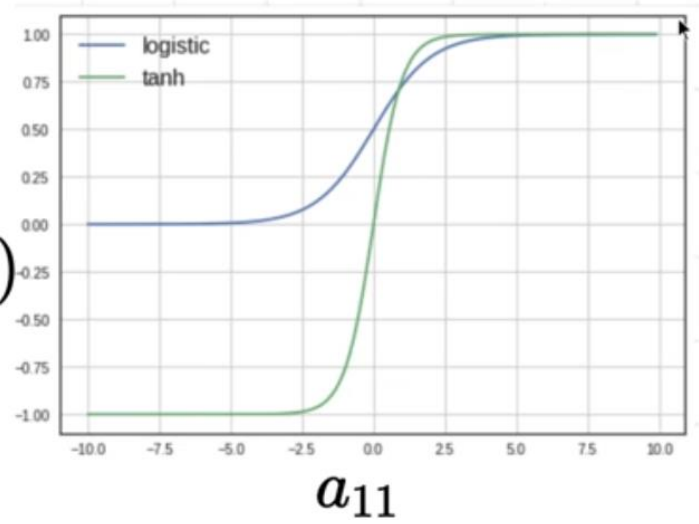
Why shouldn't you initialize all weights to large values?



$$a_{11} = w_{11}x_1 + w_{12}x_2$$

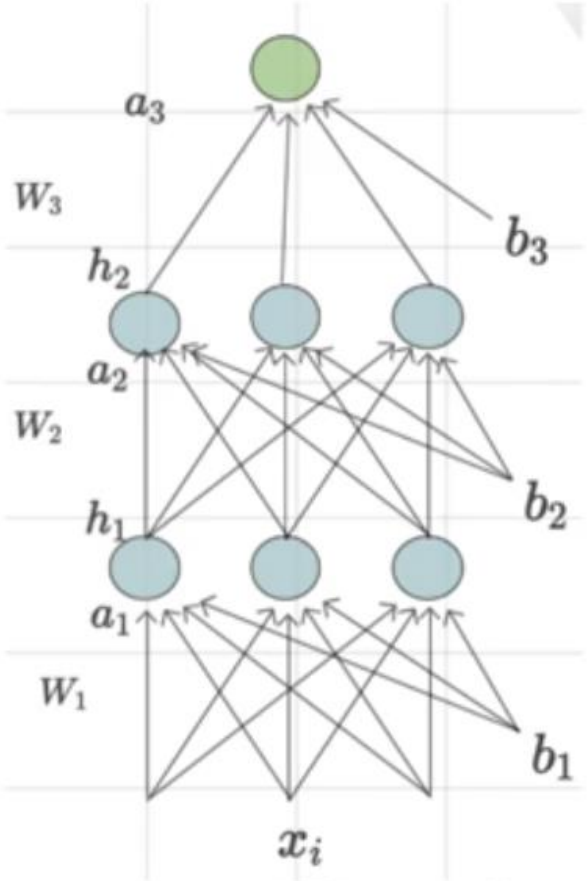
$$a_{12} = w_{21}x_1 + w_{22}x_2$$

$$h_{11} = \sigma(a_{11})$$



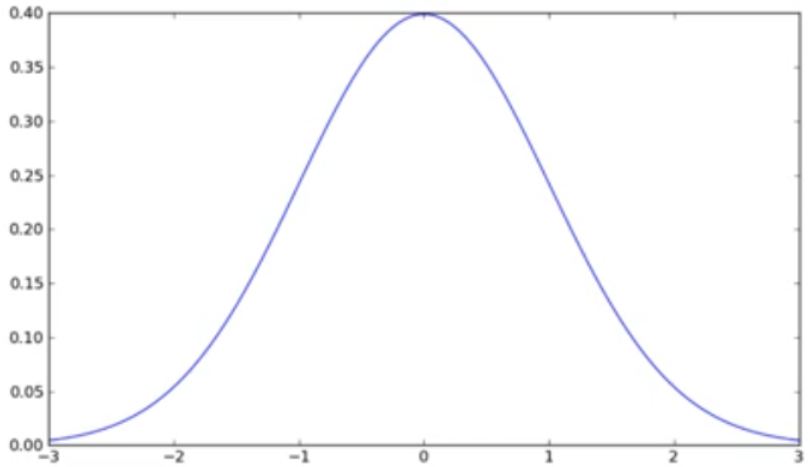
- always normalize the inputs (so that they lie between 0 to 1)
- never initialize weights to large values

Xavier / Glorot Initialization



$$a_2 = W_2 x + b_2$$

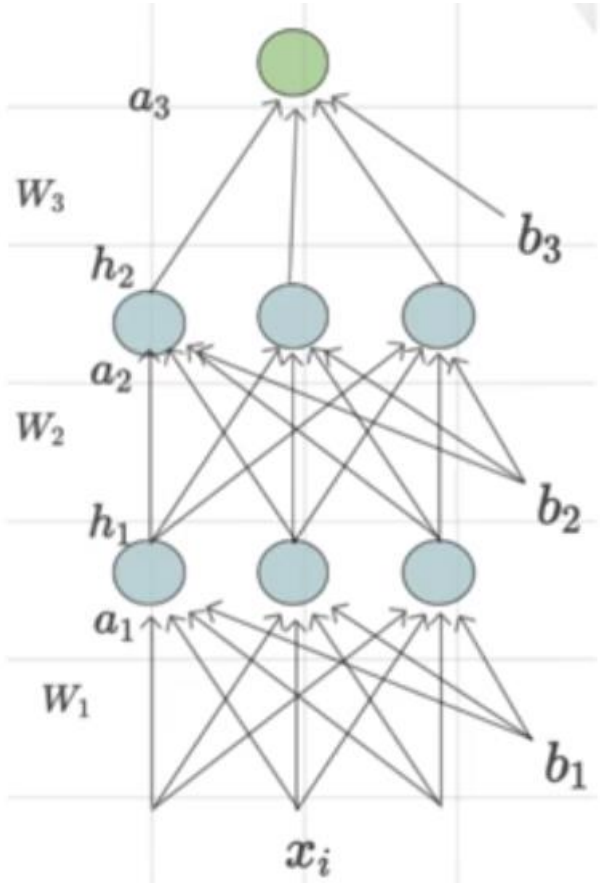
$$h_2 = \sigma(a_2)$$



Use **Xavier initialization** for tanh and logistic activations

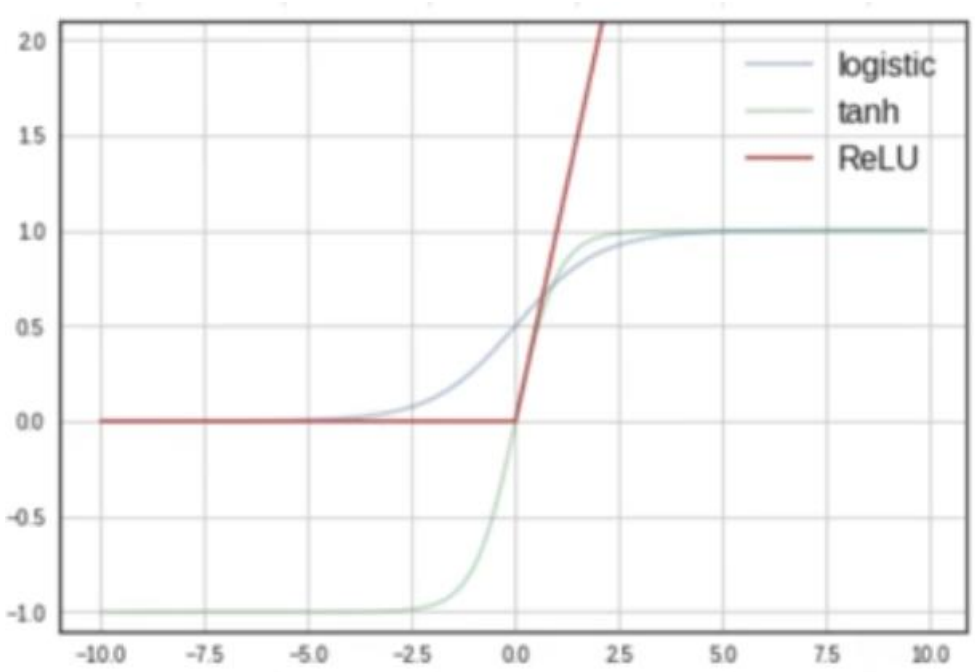
W_2 = np.random.randn(num_in, num_out)/sqrt(num_in)

He Initialization



$$a_2 = W_2 x + b_2$$

$$h_2 = \sigma(a_2)$$



Use **He initialization** for ReLU and Leaky ReLU

W_2 = np.random.randn(num_in, num_out)/sqrt(num_in/2)

Key Points

Activation Functions

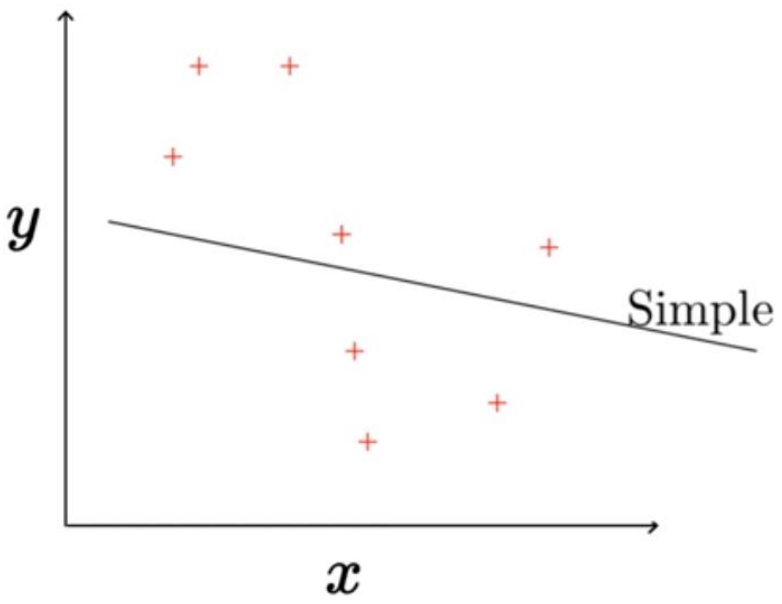
- logistic
- tanh (RNNs)
- relu (CNNs)
- leaky relu (CNNs)

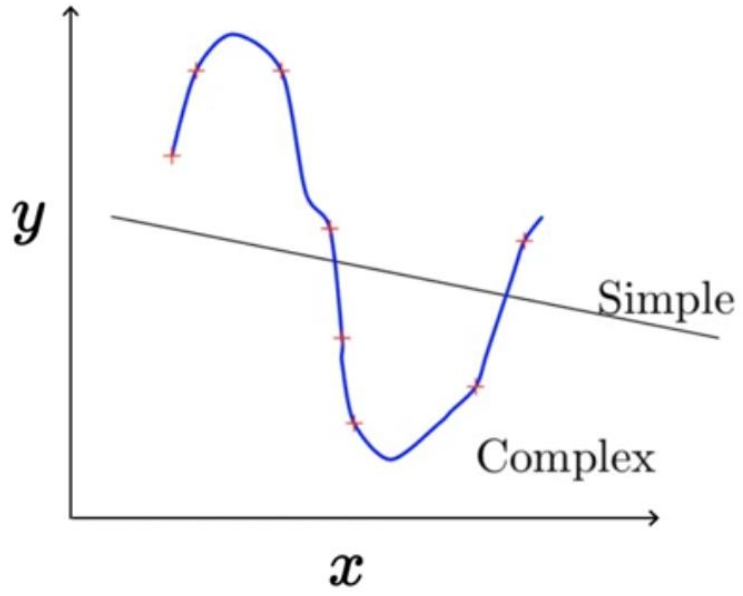
Initialization Methods

- zero, equal, large
- *Xavier* initialization (tanh, logistic)
- *He* initialization (relu)

Better Regularization

(Why do we need regularization ?)





*In this case I know that $f(x) = sin(x)$

True Relation*

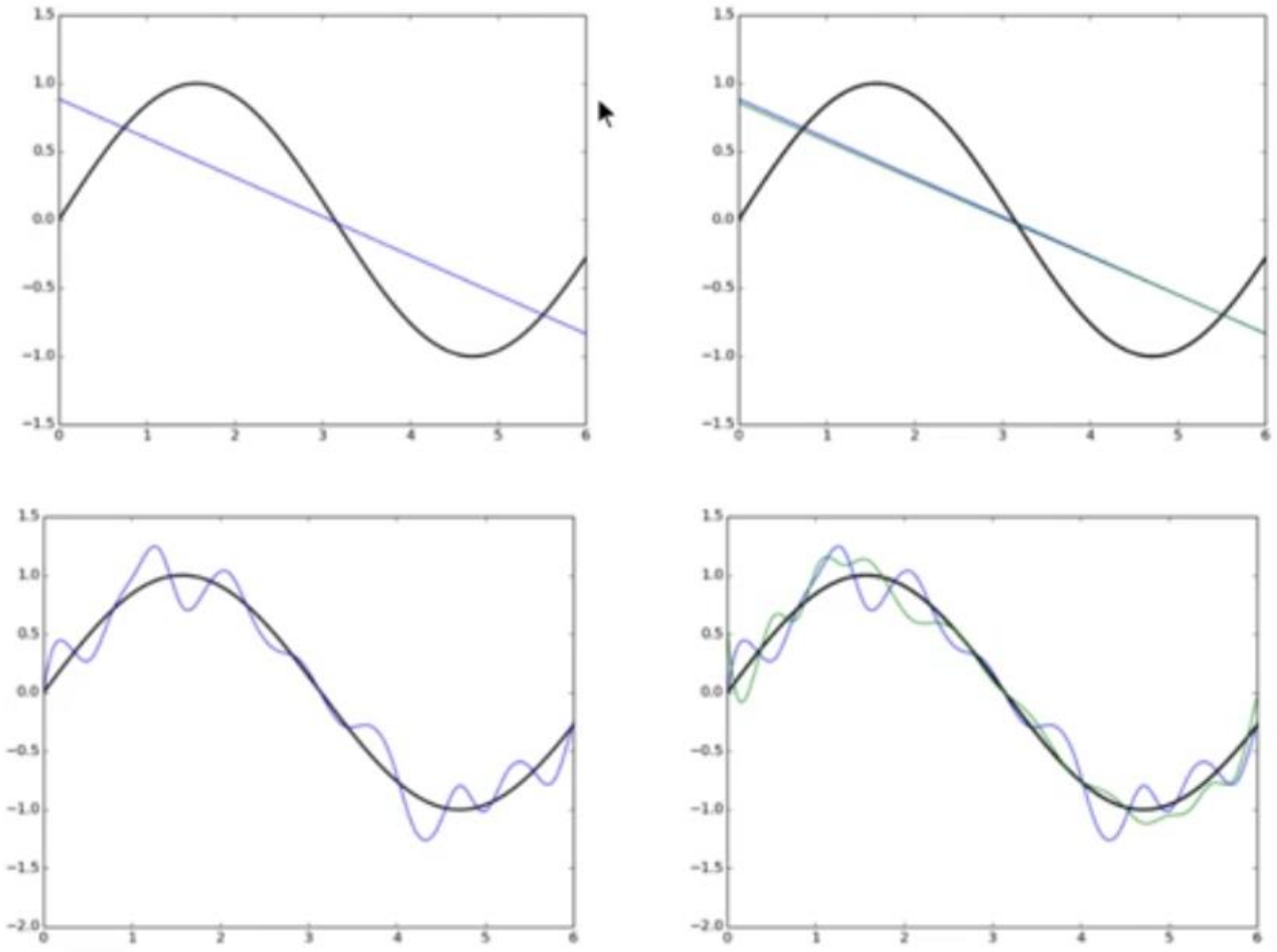
$$y = f(x)$$

Our Approximation(model):

$$\underset{(degree:1)}{Simple} \quad y = \hat{f}(x) = w_1 x + w_0$$

$$\underset{(degree:25)}{Complex} \quad y = \hat{f}(x) = \sum_{i=1}^{25} w_i x^i + w_0$$

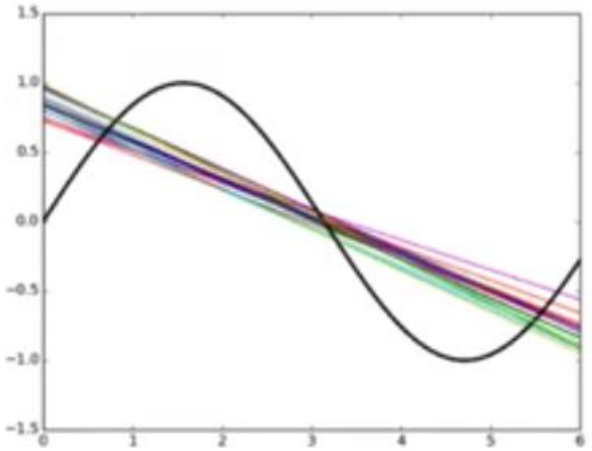
Bias-Variance Trade-off

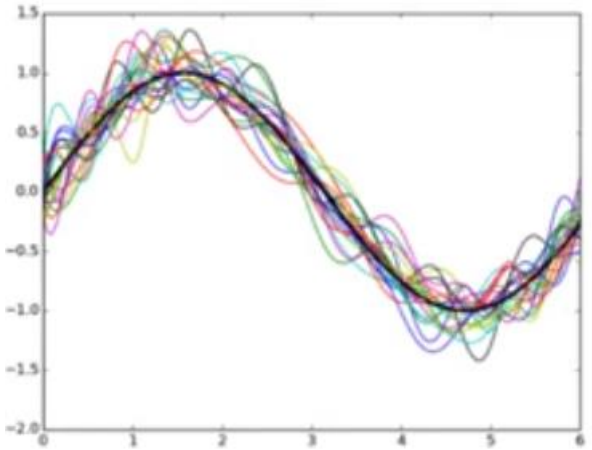


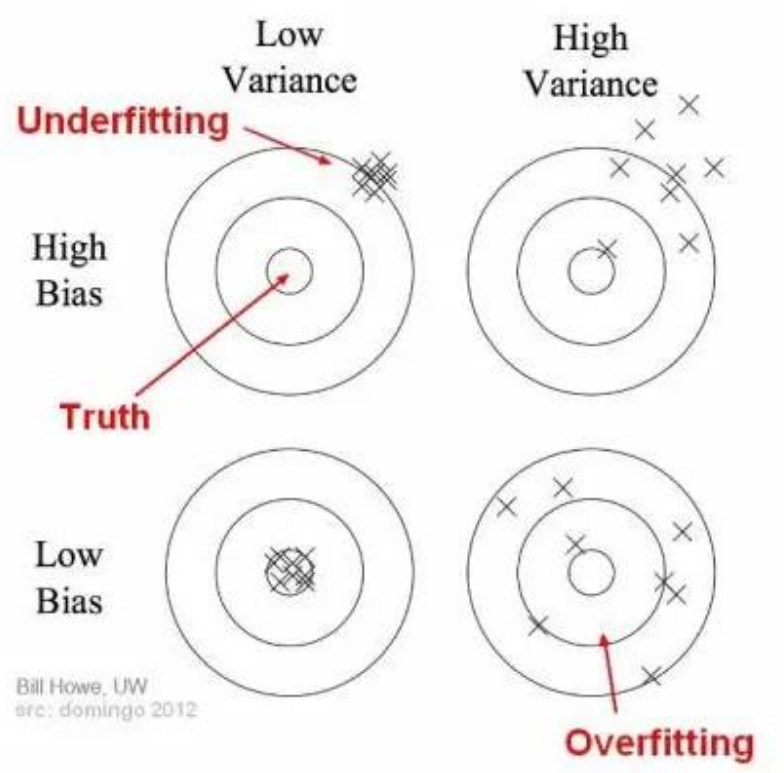
Bias-Variance Trade-off

$$\text{Bias } (\hat{f}(x)) = E[\hat{f}(x)] - f(x)$$

$$\text{Variance } (\hat{f}(x)) = E[(\hat{f}(x) - E[\hat{f}(x)])^2]$$

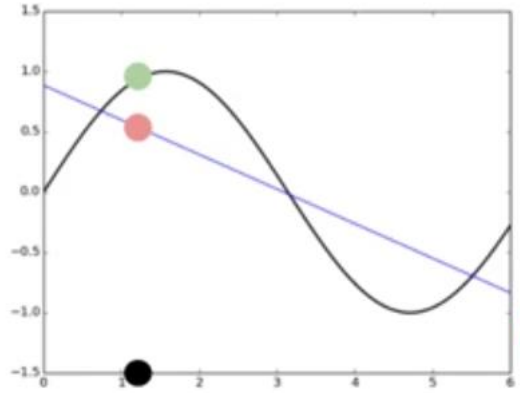




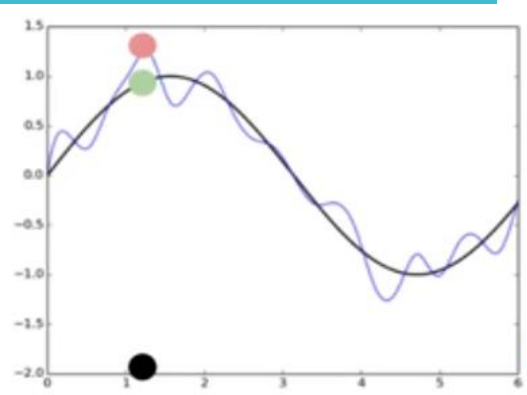


Simple Model: high bias, low variance
Complex Model: low bias, high variance
Ideal Model: low bias, low variance

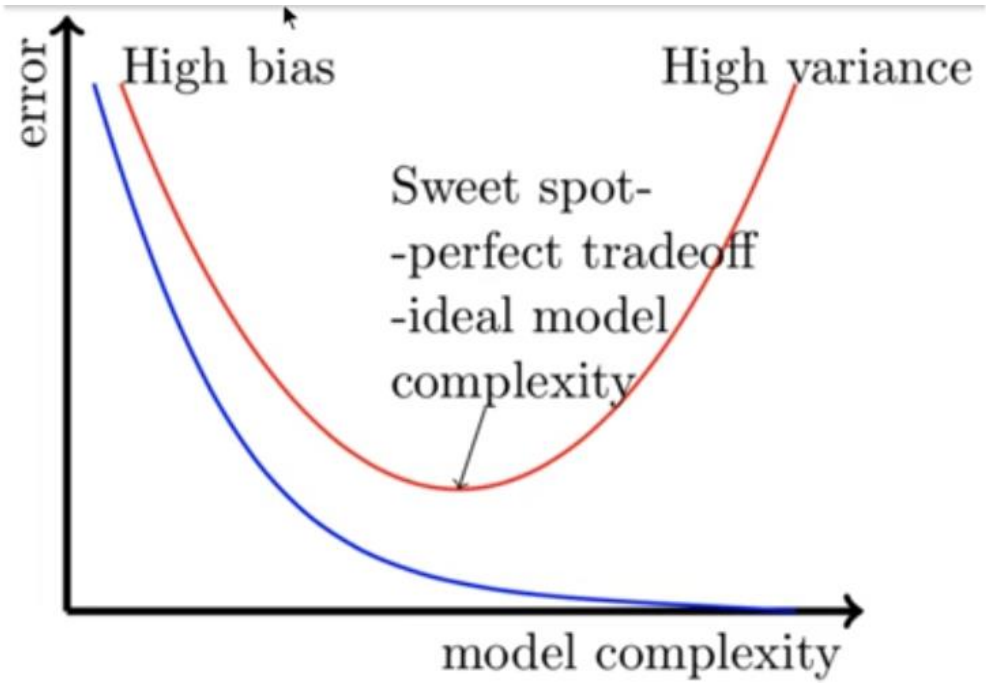
Bias-Variance Trade-off



High test error due to high bias (under-fitting)



High test error due to high variance (over-fitting)



Why do we care about Bias-Variance Trade-off in the context of Deep Learning?

How to deal with it?

- Deep Neural Networks are highly complex models (many parameters, many non-linearities)
- Easy to overfit (drive training error to 0)

- Divide data into train, test and validation/development splits
- Start with some network configuration (say, 2 hidden layers, 50 neurons each)
- Make sure that you are using the
  - right activation function (tanh, ReLU, leaky ReLU)
  - right initialization method (He, Xavier) and
  - right optimization method (say, Adam)
- Monitor training and validation error (do not touch the test data)

How to deal with it in practice via observations?

Training Error	Valid Error	Cause	Solution
High	High	High bias	- Increase model complexity - Train for more epochs
Low	High	High variance	- Add more training data (e.g., **dataset augmentation**) - Use **regularization** - User **early stopping** (train less)
Low	Low	Perfect tradeoff	- You are done!

Hyperparameter Tuning

Algorithms

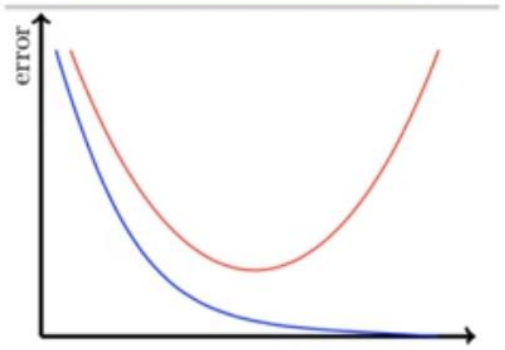
- Vanilla/Momentum /Nesterov GD
- AdaGrad
- RMSProp
- Adam

Strategies

- Batch
- Mini-Batch (32, 64, 128)
- Stochastic
- Learning rate schedule

Network Architectures

- Number of layers
- Number of neurons



Initialization Methods

- *Xavier*
- *He*

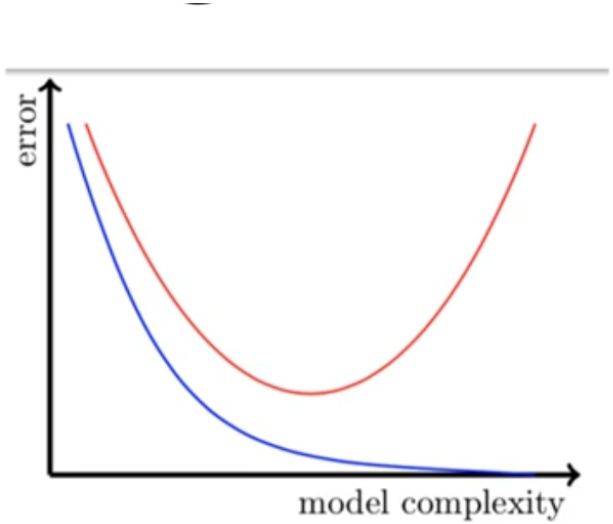
Activation Functions

- tanh (RNNs)
- relu (CNNs, DNNs)
- leaky relu (CNNs)

Regularization

- L2
- Early Stopping
- Dataset Augmentation
- Drop-out
- Batch Normalization

L2 Regularization



$$\mathscr{L}_{train}(\theta) = \sum_{i=1}^{N}(y_i - f(x_i))^2$$

$$\min_{\theta} \mathscr{L}(\theta) = \mathscr{L}_{train}(\theta) + \Omega(\theta)$$

$$\theta = [W_{111}, W_{112} + \cdots + W_{Lnk}]$$

$$\Omega(\theta) = ||\theta||_2^2$$

$$\Omega(\theta) = W_{111}^2 + W_{112}^2 + \cdots + W_{Lnk}^2$$

Can we use the same gradient descent as before?

Initialise w, b

Iterate over data:

compute $\hat{y}$

compute $\mathscr{L}(w, b)$

$$w_{111} = w_{111} - \eta \Delta w_{111}$$

$$w_{112} = w_{112} - \eta \Delta w_{112}$$

....

$$w_{313} = w_{313} - \eta \Delta w_{313}$$

till satisfied

$$\min_{\theta} \mathscr{L}(\theta) = \mathscr{L}_{train}(\theta) + \Omega(\theta)$$

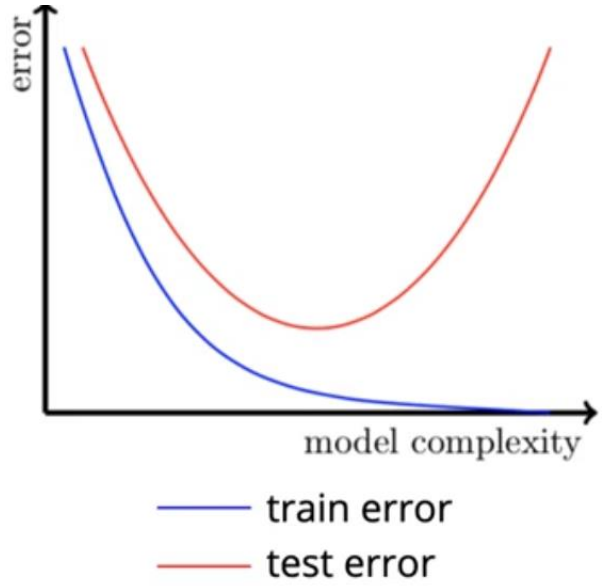
$$\mathscr{L}_{train}(\theta) = \sum_{i=1}^{N} (y_i - f(x_i))^2$$

$$\Omega(\theta) = W_{111}^2 + W_{112}^2 + \cdots + W_{Lnk}^2$$

$$\Delta W_{ijk} = \frac{\partial \mathscr{L}(\theta)}{\partial W_{ijk}}$$

$$= \frac{\partial \mathscr{L}_{train}(\theta)}{\partial W_{ijk}} + \frac{\partial \Omega(\theta)}{\partial W_{ijk}}$$

Intuition behind Data Augmentation

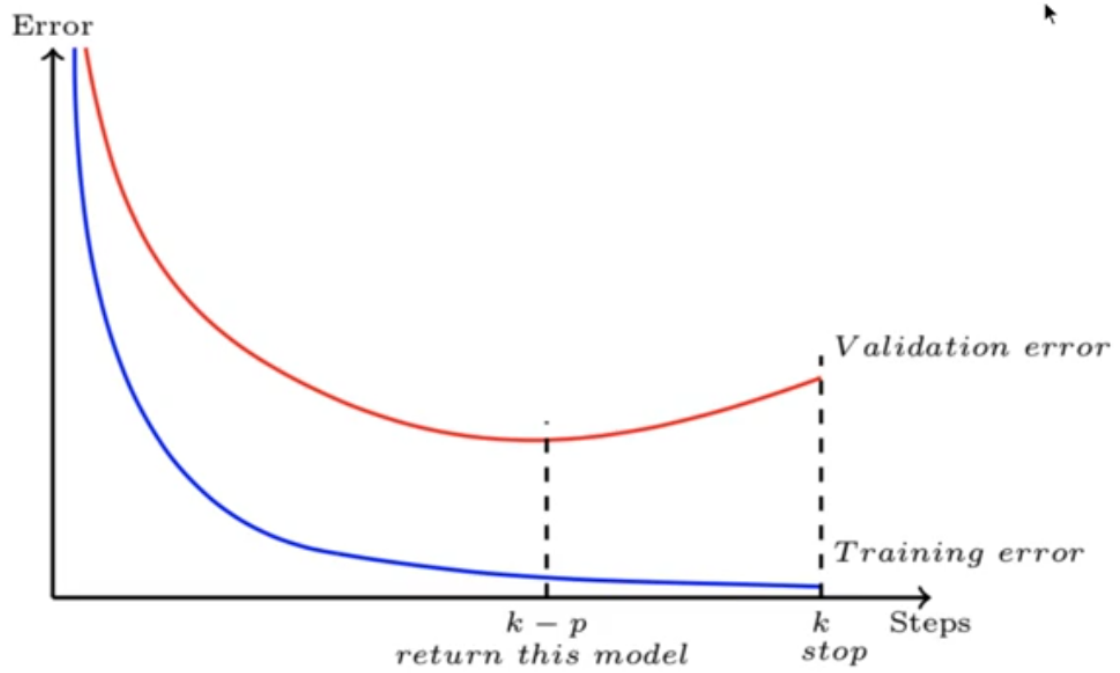


- Easy to drive training error to zero if data is less (too many parameters for very little data)
- Augmenting with more data will make it harder to drive the training data to zero
- By augmenting more data, we might also end up seeing data which is similar to valid/test data (hence, effectively reduce the valid/test data)

Data Augmentation Techniques

- Image Data Augmentation:
  - Rotation: Rotating images by a certain degree.
  - Flip: Flipping images horizontally or vertically.
  - Zoom: Zooming in or out of the images.
  - Crop and Pad: Cropping or padding images to change their size.
  - Translation: Shifting images horizontally or vertically.
  - Shear: Shearing images along the horizontal or vertical axis.
  - Brightness Adjustment: Adjusting the brightness of images.
  - Contrast Adjustment: Adjusting the contrast of images.
  - Noise Injection: Adding random noise to images.
  - Color Jitter: Randomly changing color attributes like hue, saturation, and brightness.
- Text Data Augmentation:
  - Synonym Replacement: Replacing words with their synonyms.
  - Random Insertion: Inserting random words into sentences.
  - Random Deletion: Deleting random words from sentences.
  - Random Swap: Swapping the positions of two words in a sentence.
  - Back-Translation: Translating sentences to another language and then back to the original language.
  - Word Embedding Augmentation: Modifying word embeddings to generate similar but different word representations.
- Audio Data Augmentation:
  - Time Stretching: Stretching or compressing the time dimension of audio signals.
  - Pitch Shifting: Increasing or decreasing the pitch of audio signals.
  - Additive Noise: Adding random noise to audio signals.
  - Time Shifting: Shifting the audio along the time axis.
  - Speed Perturbation: Altering the speed of the audio signal.

Early Stopping



The idea behind early stopping is to monitor the performance of the model on a validation dataset during training and stop the training process when the performance starts to degrade, typically by observing an increase in validation error.

Dropout

- Dropout is a regularization technique commonly used in deep neural networks to prevent overfitting. It works by randomly setting a fraction of the input units (neurons) to zero during each training iteration. This means that these units are temporarily ignored or "dropped out" of the network, along with all of their incoming and outgoing connections.

- Dropout is typically applied to hidden layers of the neural network, but it can also be applied to input layers or even the output layer in certain cases. It's a widely used and effective technique for improving the generalization ability of neural networks and is especially valuable when dealing with large, complex models or limited training data.

Batch Normalization

- Batch normalization is a technique used to improve the training of deep neural networks by normalizing the inputs of each layer. It was proposed as a method to mitigate the internal covariate shift problem, which refers to the phenomenon where the distribution of inputs to each layer of a neural network changes during training, leading to slower convergence and degraded performance.

- Here's how batch normalization typically works:

1. Normalization: For each mini-batch during training, batch normalization normalizes the activations of each layer by subtracting the mini-batch mean and dividing by the mini-batch standard deviation. This is done independently for each feature dimension.

2. Scaling and Shifting: After normalization, the normalized activations are scaled by a learnable parameter γ and shifted by another learnable parameter β. These parameters allow the model to learn the optimal scale and shift for the normalized activations, effectively giving it more flexibility.

3. Training and Inference: During training, the mini-batch mean, and standard deviation are computed for each batch. During inference, the overall mean and standard deviation of the entire training dataset are used instead.