

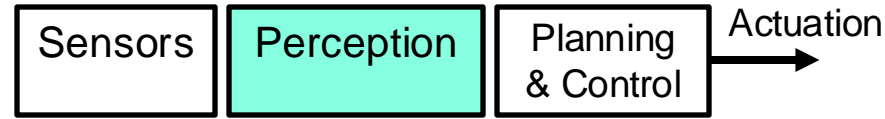
Principles of Autonomy (SP 25) ECE 484
Lecture 4
Perception: Neural Networks

Sayan Mitra



Role of Perception in Autonomy

Perception module converts signals from the environment to **state estimates** for the autonomous agent and its environment



Examples of state estimates:

- Type of lead vehicle, traffic sign
- Position of ego on the map, relative to the lane, distance to the leading vehicle
- Position of lead vehicle, speed, intention of the pedestrian

Types of estimates:

- Semantic: E.g., type/class of vehicle, sign
- Geometric: E.g., position, speed



Image classification problem



Goal: Learn a function that predicts the object in an image

Apply a prediction function to a **representation** of the image to get the desired output:

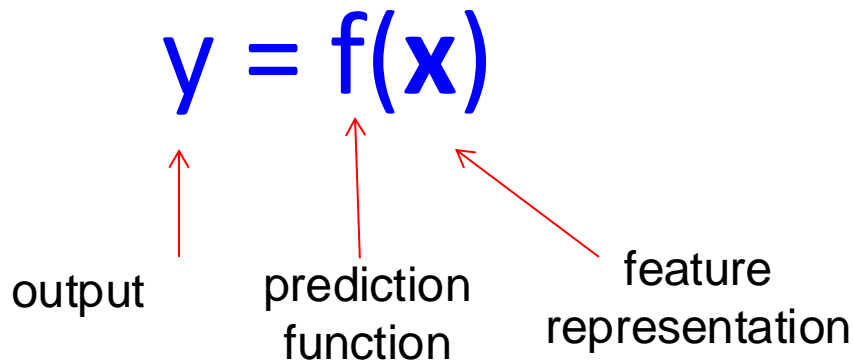
$$f(\text{apple image}) = \text{"apple"}$$

$$f(\text{tomato image}) = \text{"tomato"}$$

$$f(\text{cow image}) = \text{"cow"}$$



Statistical learning framework: Train classifier from training data



Training: given a *training set* of **labeled** examples

$\{(\mathbf{x}_1, y_1), \dots, (\mathbf{x}_N, y_N)\}$, estimate the prediction function f by minimizing the prediction error on the training set

Validation: tune (hyper)parameters in f , learning rate

Testing: apply f to a never before seen *test data* \mathbf{x} and output predicted value $y = f(\mathbf{x})$



Outline

Linear classifiers

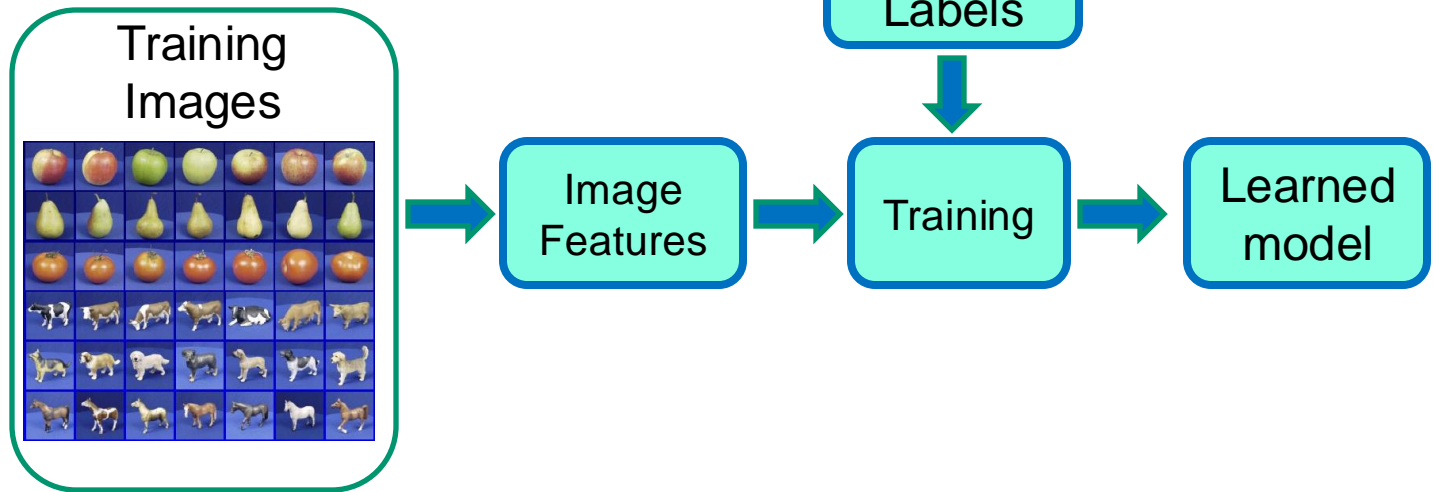
Neural networks

- Universal approximators
- Forward pass
- Backpropagation; Gradient descent
- Exploding and imploding gradients

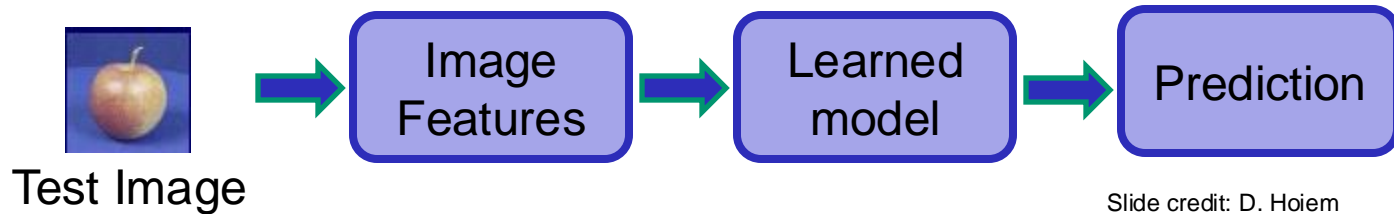
Best practices



Training



Testing

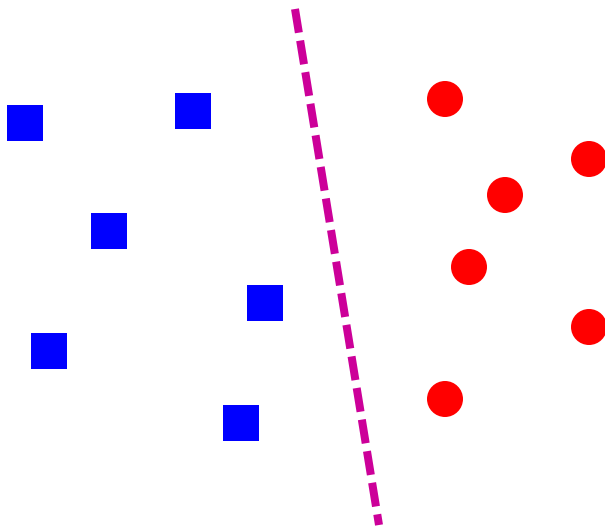


Slide credit: D. Hoiem



Linear classifiers

Images or feature
representation of
images



If interested, read Support
vector machines (SVM)
[https://greitemann.dev/svm-
demo](https://greitemann.dev/svm-demo)

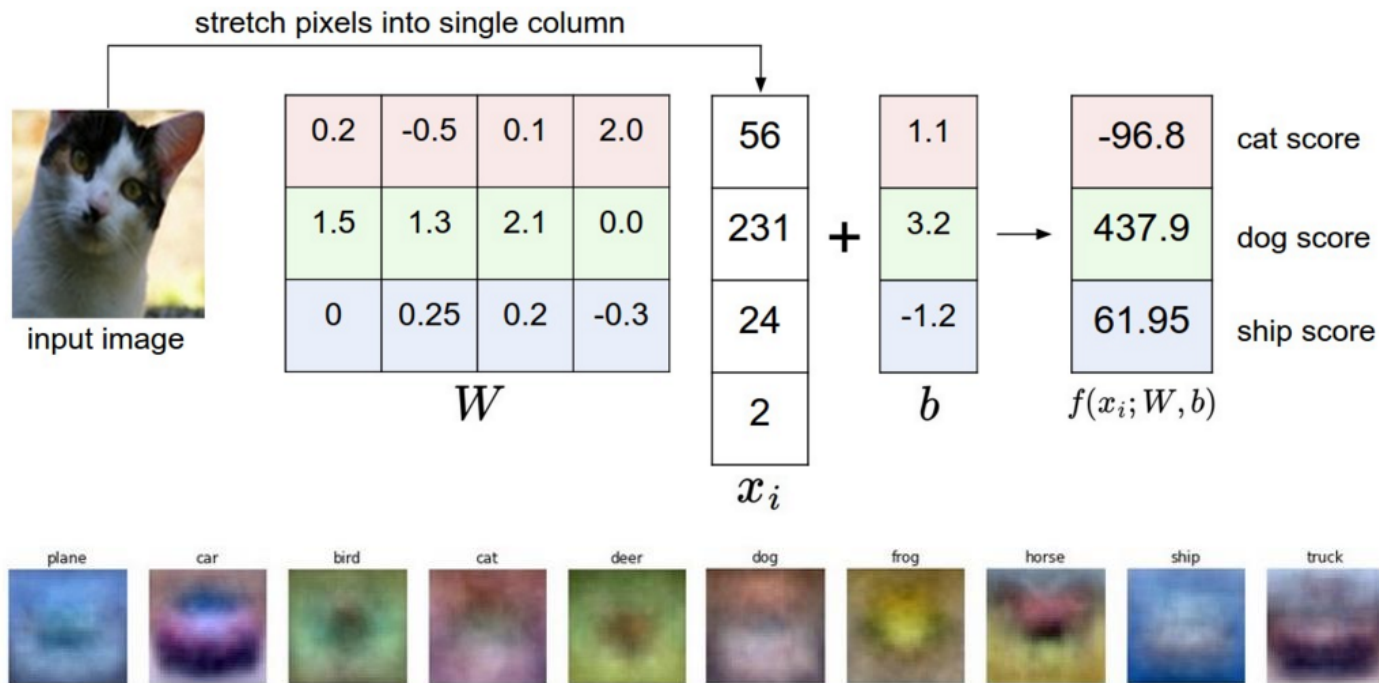
Find a *linear function* (w, b) to separate the classes:

$$f(\mathbf{x}) = \text{sgn}(\mathbf{w} \cdot \mathbf{x} + b)$$

Read more about SIFT, HOG, [bag of visual words](#) to learn more about image features.



Visualizing linear classifiers with many classes

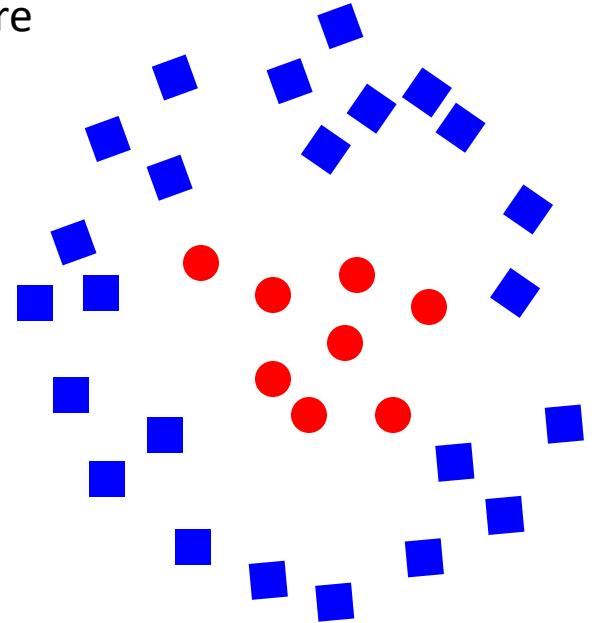


Source: Andrej Karpathy, <http://cs231n.github.io/linear-classify/>



Limitations of Linear Classifiers

- Input: A feature vector $\mathbf{x} \in \mathbb{R}^d$.
- Weights and bias: $\mathbf{w} \in \mathbb{R}^d$, $b \in \mathbb{R}$.
- Prediction (binary classification example): $\hat{y} = \mathbf{w}^T \mathbf{x} + b$
- Limitations: Linear decision boundaries may not capture complex relationships between classes



Nonlinearity via Neural Networks

A **neural network** is a function $f_{NN}: \mathbb{R}^d \rightarrow \mathbb{R}^k$ defined as a composition of layers of linear and nonlinear transformations.

Simple 2-layer network with one hidden layer and input $\mathbf{x} \in \mathbb{R}^d$

- $\mathbf{h} = g(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)})$ (hidden layer)

- $\hat{\mathbf{y}} = W^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$ (output layer)

$$\hat{\mathbf{y}} = W^{(2)}g(W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}) + \mathbf{b}^{(2)}$$

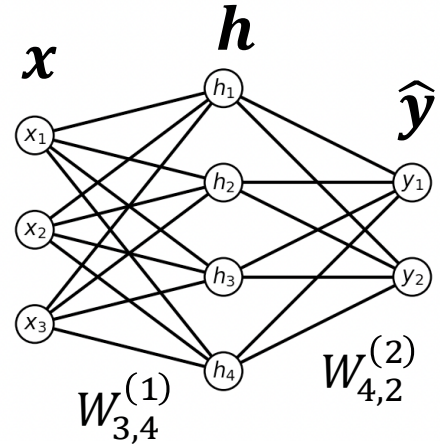
$W^{(1)} \in \mathbb{R}^{m \times d}$ $W^{(2)} \in \mathbb{R}^{k \times m}$, for m of **hidden units** are the **weights**

$\mathbf{b}^{(1)} \in \mathbb{R}^m$ $\mathbf{b}^{(2)} \in \mathbb{R}^k$ are the **biases**

g : **Activation function** e.g. $ReLU(z) = \max(z, 0)$, $sigmoid(z) = \frac{e^z}{1+e^z}$

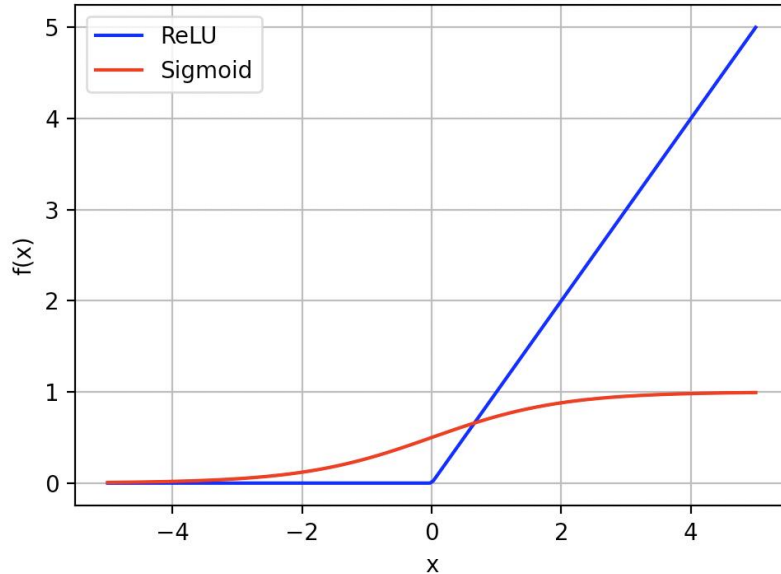
Example $k=2$ for lane boundary parameters, $k=6$ for pose components

2-Layer Network: $d=3, m=4, k=2$



Activation functions and their derivatives

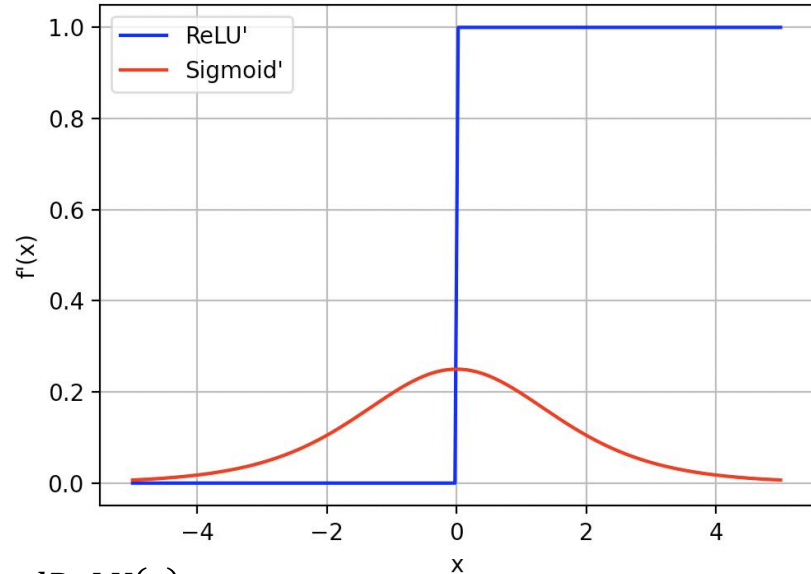
Activation Functions



$$\text{ReLU}(z) = \max(z, 0)$$

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

Derivatives



$$\frac{d\text{ReLU}(z)}{dz} = \{0, 1\}$$

$$\sigma'(z) = \sigma(z)(1 - \sigma(z))$$



Universal Approximation Theorem [Cybenko, G. 1989]

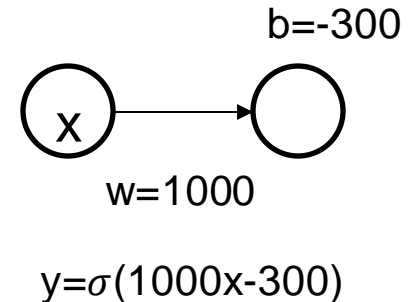
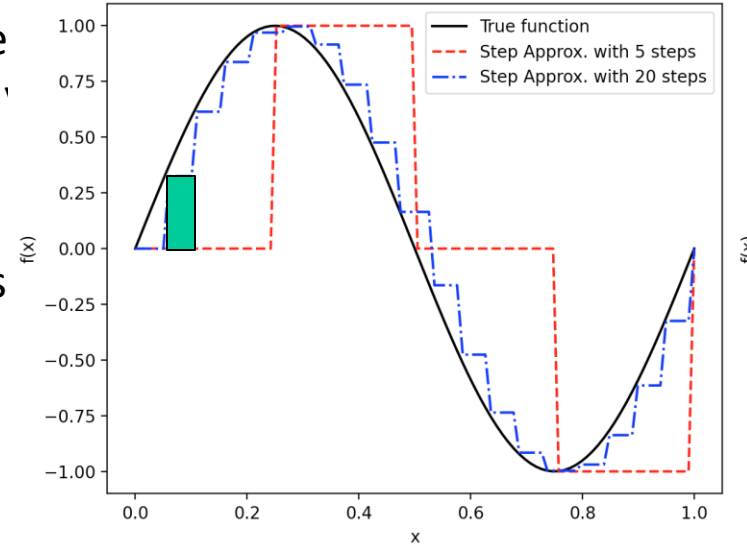
Any **continuous function** f on a compact domain can be approximated to arbitrary precision with a sufficiently large (but finite) single-hidden-layer feedforward network with a suitable activation function.

- f can be approximated arbitrarily by a sum of towers
- A tower function can be represented by a network with a single hidden layer (think of large w and b)
- Sum of towers can be created by adding more elements in the hidden layer

Neural networks are expressive enough to infer complex state estimates from raw pixels

Cybenko, G. (1989). "Approximation by superpositions of a sigmoidal function." *Mathematics of Control, Signals and Systems*, 2(4), 303–314.

<https://www.youtube.com/watch?v=ljqkc7OLenI>



Neural Network Forward Propagation

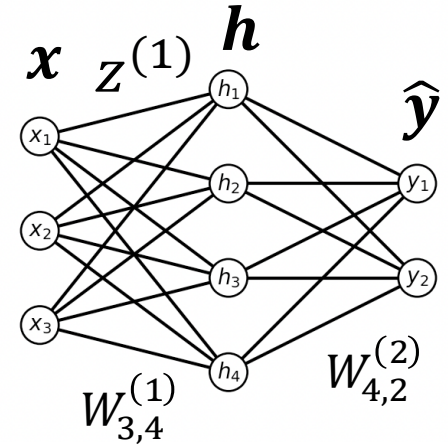
For a given input x :

- Compute hidden layer **pre-activation**: $z^{(1)} = W^{(1)}x + b^{(1)}$.
- Apply activation: $h = f(z^{(1)})$.
- Compute output layer: $\hat{y} = W^{(2)}h + b^{(2)}$.

This results in a prediction \hat{y} , e.g.:

- For lane estimation: $\hat{y} \in \mathbb{R}^{H \times W}$ if predicting per-pixel segmentation, *or* $\hat{y} \in \mathbb{R}^p$ if predicting lane **embedding**.
- For 6DOF pose: $\hat{y} \in \mathbb{R}^6$, representing $(x, y, z, \alpha, \beta, \gamma)$ or other parameterization of rotation and translation.

2-Layer Network: $d=3, m=4, k=2$



Backpropagation and Gradient-Based Training

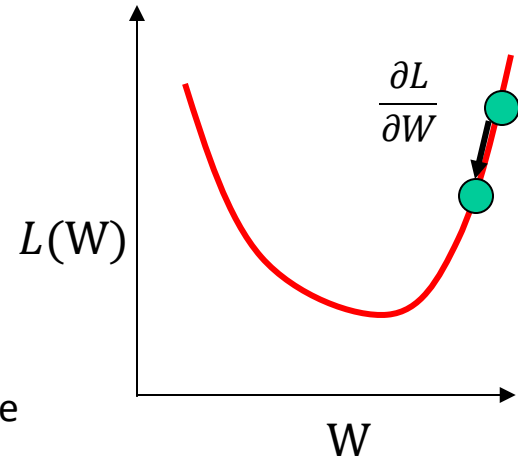
Loss Function: A scalar function $L(\hat{\mathbf{y}}, \mathbf{y})$ measures how well predictions match the ground truth \mathbf{y} .

For lane segmentation (classification per pixel) cross-entropy loss

$$L = -\frac{1}{N} \sum_{i=1}^N [y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)], y_i \in \{0,1\} \hat{y}_i \in [0,1]$$

For pose regression, L2 distance: $L = -\frac{1}{N} \sum_{i=1}^N |\hat{\mathbf{y}}_i - \mathbf{y}_i|_2^2$

The **loss function** (L) function is minimized during training by changing the weights (W) and the biases (b) of the neural network (f) using **back propagation or gradient descent**



Backpropagation Background: Vector Calculus refresher

Differentiating scalars, vectors, and matrices:

$$\mathbf{y} = (\mathbf{y}^{(1)}, \dots, \mathbf{y}^{(n)})^T; \frac{\partial}{\partial \mathbf{x}} = \left(\frac{\partial}{\partial x^{(1)}}, \frac{\partial}{\partial x^{(2)}}, \frac{\partial}{\partial x^{(3)}} \right)$$

Gradient operator ∇

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}}, \frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial \mathbf{y}^{(1)}}{\partial \mathbf{x}} \\ \dots \\ \frac{\partial \mathbf{y}^{(n)}}{\partial \mathbf{x}} \end{bmatrix}, \frac{\partial Y}{\partial \mathbf{x}} = \begin{bmatrix} \frac{\partial Y^{(1,1)}}{\partial \mathbf{x}} & \dots \\ \dots & \frac{\partial Y^{(i,j)}}{\partial \mathbf{x}} \end{bmatrix}$$

Element-wise differentiation of vectors and matrices

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial}{\partial \mathbf{x}} \cdot \mathbf{y} = \left(\frac{\partial}{\partial x^{(1)}}, \frac{\partial}{\partial x^{(2)}}, \frac{\partial}{\partial x^{(3)}} \right) \cdot \mathbf{y}$$

Gradient of \mathbf{y} ; row vector of partials

$$\frac{\partial \mathbf{y}}{\partial \mathbf{x}} = \frac{\partial}{\partial \mathbf{x}} \cdot \mathbf{y} = \left(\frac{\partial}{\partial x^{(1)}}, \frac{\partial}{\partial x^{(2)}}, \frac{\partial}{\partial x^{(3)}} \right) \cdot \begin{bmatrix} \mathbf{y}^{(1)} \\ \mathbf{y}^{(2)} \\ \mathbf{y}^{(3)} \end{bmatrix} = \begin{bmatrix} \frac{\partial \mathbf{y}^{(1)}}{\partial x^{(1)}} & \frac{\partial \mathbf{y}^{(1)}}{\partial x^{(2)}} & \frac{\partial \mathbf{y}^{(1)}}{\partial x^{(3)}} \\ \dots & \dots & \dots \\ \frac{\partial \mathbf{y}^{(3)}}{\partial x^{(1)}} & \frac{\partial \mathbf{y}^{(3)}}{\partial x^{(2)}} & \frac{\partial \mathbf{y}^{(3)}}{\partial x^{(3)}} \end{bmatrix}$$

Jacobian; \cdot is Kronecker product (also \odot)

$$\frac{\partial Y}{\partial \mathbf{x}} = \left(\frac{\partial}{\partial x^{(1)}}, \frac{\partial}{\partial x^{(2)}}, \frac{\partial}{\partial x^{(3)}} \right) \cdot Y = \left[\frac{\partial}{\partial x^{(1)}} Y \frac{\partial}{\partial x^{(2)}} Y \frac{\partial}{\partial x^{(3)}} Y \right]$$



Backpropagation Background: Vector Calculus notations

$$\frac{\partial y}{\partial X} = \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot y$$

$$\frac{\partial y}{\partial X} = \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot \mathbf{y} = \left[\begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot \mathbf{y}^{(1)} \quad \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot \mathbf{y}^{(2)} \quad \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot \mathbf{y}^{(3)} \right]$$

$$\frac{\partial Y}{\partial X} = \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \cdot Y = \begin{bmatrix} \frac{\partial}{\partial X^{(1,1)}} & \frac{\partial}{\partial X^{(1,2)}} \\ \frac{\partial}{\partial X^{(2,1)}} & \frac{\partial}{\partial X^{(2,2)}} \end{bmatrix} \begin{bmatrix} Y^{(1,1)} & Y^{(1,2)} \\ Y^{(2,1)} & Y^{(2,2)} \end{bmatrix} = \begin{bmatrix} \begin{bmatrix} \frac{\partial Y^{(1,1)}}{\partial X^{(1,1)}} & \frac{\partial Y^{(1,1)}}{\partial X^{(1,2)}} \\ \frac{\partial Y^{(1,2)}}{\partial X^{(1,1)}} & \frac{\partial Y^{(1,2)}}{\partial X^{(1,2)}} \end{bmatrix} & \begin{bmatrix} \frac{\partial Y^{(2,1)}}{\partial X^{(1,1)}} & \frac{\partial Y^{(2,1)}}{\partial X^{(1,2)}} \\ \frac{\partial Y^{(2,2)}}{\partial X^{(1,1)}} & \frac{\partial Y^{(2,2)}}{\partial X^{(1,2)}} \end{bmatrix} \\ \dots & \dots \end{bmatrix}$$



Backpropagation and Gradient-Based Training

$$\mathbf{h} = f(\mathbf{z}^{(1)}) = W^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\hat{\mathbf{y}} = W^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

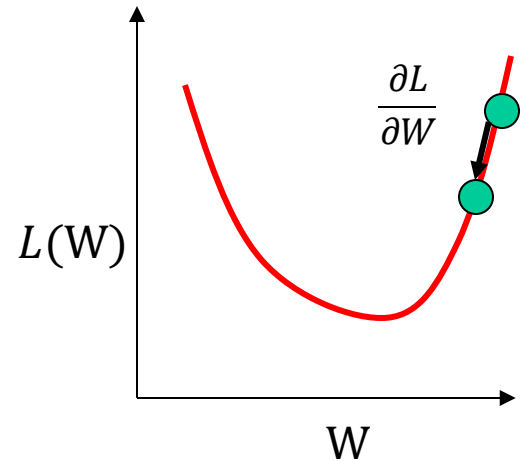
For pose regression: $L = -\frac{1}{N} \sum_{i=1}^N |\hat{\mathbf{y}}_i - \mathbf{y}_i|_2^2$

Computing Gradients (Backprop):

- We compute $\frac{\partial L}{\partial W^{(2)}}$, $\frac{\partial L}{\partial \mathbf{b}^{(2)}}$, $\frac{\partial L}{\partial W^{(1)}}$, $\frac{\partial L}{\partial \mathbf{b}^{(1)}}$ using chain rule
- For example: $\frac{\partial L}{\partial W^{(2)}} = \frac{\partial L}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial W^{(2)}} = \left(\frac{\partial L}{\partial \hat{\mathbf{y}}} \right) \mathbf{h}^\top$

Gradient Descent Update: With learning rate η

$$W^{(l)} := W^{(l)} - \eta \frac{\partial L}{\partial W^{(l)}}, \quad \mathbf{b}^{(l)} := \mathbf{b}^{(l)} - \eta \frac{\partial L}{\partial \mathbf{b}^{(l)}}$$



$$\mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} w_{11}h_1 + w_{12}h_2 + w_{13}h_3 + b_1 \\ w_{21}h_1 + w_{22}h_2 + w_{23}h_3 + b_2 \end{bmatrix}$$

$$\frac{\partial \mathbf{y}}{\partial \mathbf{w}} = \begin{bmatrix} \frac{\partial}{\partial w^{(1,1)}} & \frac{\partial}{\partial w^{(1,2)}} & \frac{\partial}{\partial w^{(1,3)}} \\ \frac{\partial}{\partial w^{(2,1)}} & \frac{\partial}{\partial w^{(2,2)}} & \frac{\partial}{\partial w^{(2,3)}} \end{bmatrix} \odot \mathbf{y}^T$$

$$\begin{bmatrix} \left[\frac{\partial}{\partial w^{(1,1)}} & \frac{\partial}{\partial w^{(1,2)}} & \frac{\partial}{\partial w^{(1,3)}} \right] w_{11}h_1 + w_{12}h_2 + w_{13}h_3 + b_1 & \left[\frac{\partial}{\partial w^{(1,1)}} & \frac{\partial}{\partial w^{(1,2)}} & \frac{\partial}{\partial w^{(1,3)}} \right] w_{21}h_1 + w_{22}h_2 + w_{23}h_3 + b_2 \\ \left[\frac{\partial}{\partial w^{(2,1)}} & \frac{\partial}{\partial w^{(2,2)}} & \frac{\partial}{\partial w^{(2,3)}} \right] w_{11}h_1 + w_{12}h_2 + w_{13}h_3 + b_1 & \left[\frac{\partial}{\partial w^{(2,1)}} & \frac{\partial}{\partial w^{(2,2)}} & \frac{\partial}{\partial w^{(2,3)}} \right] w_{21}h_1 + w_{22}h_2 + w_{23}h_3 + b_2 \end{bmatrix} \cdot$$

$$\begin{bmatrix} \left[h_1 & h_2 & h_3 \right] \cdot \begin{bmatrix} 0 & 0 & 0 \\ h_1 & h_2 & h_3 \end{bmatrix} \\ \left[0 & 0 & 0 \right] \cdot \begin{bmatrix} 0 & 0 & 0 \\ h_1 & h_2 & h_3 \end{bmatrix} \end{bmatrix} = I \odot h^T$$



Vanishing gradient problem

Consider MLP with:

- Input $x \in \mathbb{R}^d$
- 3 hidden layers, each with **sigmoid**.
- 1 output $\hat{y} \in \mathbb{R}$ for a regression or binary classification.

That is

$$1. z^{(1)} = W^{(1)}x + b^{(1)} \in \mathbb{R}^m$$

$$2. a^{(1)} = \sigma(z^{(1)}) \in \mathbb{R}^m$$

$$3. z^{(i)} = W^{(i)}a^{(i-1)} + b^{(i)} \in \mathbb{R}^m, i = 2,3,4$$

$$4. a^{(i)} = \sigma(z^{(i)}) \in \mathbb{R}^m, i = 2,3$$

$$5. \hat{y} = \sigma(z^{(4)}) \in \mathbb{R}^m$$

$$\text{Loss } L = \frac{1}{2}(\hat{y} - y)^2$$



As the number of layers increase gradient can vanish

$$\frac{\partial L}{\partial z^{(4)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(4)}} = (\hat{y} - y) \frac{\partial \hat{y}}{\partial z^{(4)}} = (\hat{y} - y) \hat{y} (1 - \hat{y})$$

$$\frac{\partial L}{\partial a^{(3)}} = \frac{\partial L}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial a^{(3)}} = (\hat{y} - y) \hat{y} (1 - \hat{y}) \cdot W^{(4)}$$

$$\frac{\partial L}{\partial z^{(3)}} = \frac{\partial L}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} = (\hat{y} - y) \hat{y} (1 - \hat{y}) \cdot W^{(4)} \odot \sigma'(z^{(3)})$$

$$\frac{\partial L}{\partial z^{(\ell)}} = \frac{\partial L}{\partial z^{(\ell+1)}} W^{\ell+1} \sigma'(z^{(\ell)})$$

$$z^{(1)} = W^{(1)}x + b^{(1)}$$

$$a^{(1)} = \sigma(z^{(1)})$$

$$z^{(2)} = W^{(2)}a^{(1)} + b^{(2)}$$

$$a^{(2)} = \sigma(z^{(2)})$$

$$z^{(3)} = W^{(3)}a^{(2)} + b^{(3)}$$

$$a^{(3)} = \sigma(z^{(3)})$$

$$z^{(4)} = W^{(4)}a^{(3)} + b^{(4)}$$

$$\hat{y} = \sigma(z^{(4)})$$

$$L = \frac{1}{2}(\hat{y} - y)^2$$

If each $\sigma'(z^{(\ell)}) \leq 0.25$ the gradient vanishes



Exploding gradient problem

Consider MLP with:

- Input $x \in \mathbb{R}^d$
- 3 hidden layers, each with **RELU**.
- 1 output $\hat{y} \in \mathbb{R}$ for a regression or binary classification.

That is

1. $z^{(1)} = W^{(1)}x + b^{(1)} \in \mathbb{R}^m = \alpha I x$ for simplicity $b^{(i)} = 0$ $W^{(i)} = \alpha I$

2. $a^{(1)} = \text{ReLU}(z^{(1)}) = \alpha x$ if $a^{(1)}$ is positive in each component

3. $z^{(i)} = \alpha I a^{(i-1)} = \alpha^i x, i = 2, 3, 4$

4. $a^{(i)} = \text{ReLU}(z^{(i)}) = \alpha^i x, i = 2, 3$

5. $\hat{y} = \text{ReLU}(z^{(4)}) = \alpha^4 x$

Loss $L = \frac{1}{2}(\hat{y} - y)^2$



Exploding gradient continued

$$\frac{\partial L}{\partial z^{(4)}} = \frac{\partial L}{\partial \hat{y}} \frac{\partial \hat{y}}{\partial z^{(4)}} = \alpha^4 x - y. 1$$

$$\frac{\partial L}{\partial a^{(3)}} = \frac{\partial L}{\partial z^{(4)}} \frac{\partial z^{(4)}}{\partial a^{(3)}} = (\alpha^4 x - y. 1) \alpha I$$

$$\frac{\partial L}{\partial z^{(3)}} = \frac{\partial L}{\partial a^{(3)}} \frac{\partial a^{(3)}}{\partial z^{(3)}} = (\alpha^4 x - y. 1) \alpha. 1$$

$$\frac{\partial L}{\partial a^{(2)}} = \frac{\partial L}{\partial z^{(3)}} \frac{\partial z^{(3)}}{\partial a^{(2)}} = (\alpha^4 x - y. 1) \alpha \alpha. I = \alpha^2 (\alpha^4 x - y. 1)$$

...

$$\frac{\partial L}{\partial a^{(1)}} = \alpha^3 (\alpha^4 x - y. 1)$$

$$z^{(1)} = W^{(1)}x + b^{(1)} = \alpha I x$$

$$a^{(1)} = \text{ReLU}(z^{(1)}) = \alpha x$$

$$z^{(i)} = \alpha I a^{(i-1)}$$

$$a^{(i)} = \text{ReLU}(z^{(i)})$$

$$\hat{y} = \text{ReLU}(z^{(4)})$$

$$L = \frac{1}{2} (\hat{y} - y)^2$$

If $\alpha \gg 1$ the factor α^3 explodes in the gradient computation

Caution: Sigmoid activations clip the gradient and can lead to vanishing gradients

ReLU can make the gradients large



Further readings on NN architectures

MLP: $x_{\ell+1} = \sigma(W_{\ell}x_{\ell} + b_{\ell})$

ResNets: $x_{\ell+1} = x_{\ell} + \frac{1}{L} U_{\ell}^T \sigma(V_{\ell}x_{\ell} + b_{\ell})$, U_{ℓ}, V_{ℓ} are the weight matrices and L is the number of layers, can avoid issues with gradient collapse

Neural ODEs: As the number of layers approaches $L \rightarrow \infty$ ResNets can be seen as a discretization of $\frac{dx_s}{ds} = U_s^T \sigma(V_s x_s + b_s)$

He, Zhang, Ren, and Sun. Deep residual learning for image recognition. CVPR, 2016.

Chen, Rubanova, Bettencourt, and K Duvenaud. Neural ordinary differential equations. NeurIPs, 2018.



Example NN training in Python: Setup

```
# Dataset: a circle N = 200 # samples
```

```
X = np.random.randn(N, 2) # shape: (N, 2) r = 1.0
```

```
Y = (X[:,0]**2 + X[:,1]**2 < r**2).astype(np.float32) # shape: (N, 1) 0 or 1
```

```
# Define NN architecture
```

```
input_dim = 2 hidden_dim = 8 output_dim = 1 # binary classification
```

```
# Initialize NN: small random values for weights, and zeros for biases
```

```
W1 = 0.01 * np.random.randn(input_dim, hidden_dim) # shape: (2, 8)
```

```
b1 = np.zeros((1, hidden_dim)) # shape: (1, 8)
```

```
W2 = 0.01 * np.random.randn(hidden_dim, output_dim) # shape: (8, 1)
```

```
b2 = np.zeros((1, output_dim)) # shape: (1, 1)
```



```
def compute_loss(Y_pred, Y_true): # Cross-entropy loss:  $L = -\frac{1}{N} \sum_i y_i \log(\hat{y}_i) + (1 - y_i) \log(1 - \hat{y}_i)$ 
return -1/N*sum(Y_true*log(Y_pred+epsilon) + (1-Y_true)*log(1 - Y_pred+epsilon))
```

Training loop

```
learning_rate = 0.05 num_iterations = 1000
```

```
for i in range(num_iterations):
```

```
    # Forward pass Layer 1:  $Z1 = XW1 + b1$ ; Layer 2:  $Z2 = A1W2 + b2$ 
```

```
    Z1 = np.dot(X, W1) + b1 # shape: (N, 8)
```

```
    A1 = relu(Z1) # shape: (N, 8)
```

```
    Z2 = np.dot(A1, W2) + b2 # shape: (N, 1)
```

```
    A2 = sigmoid(Z2) # Y_pred
```

```
    loss = compute_loss(A2, Y)
```

```
    # Backward pass; For cross-entropy and sigmoid,  $dL/dZ2 = A2 - Y$ 
```

```
    dZ2 = A2 - Y # shape: (N, 1)
```

```
    dW2 = np.dot(A1.T, dZ2) # shape: (8, 1)
```

```
    dB2 = np.sum(dZ2, axis=0, keepdims=True) # shape: (1,1)
```

```
    dA1 = np.dot(dZ2, W2.T) # shape: (N,8)
```

```
    dZ1 = dA1 * relu_derivative(Z1) # shape: (N,8)
```

```
    dW1 = np.dot(X.T, dZ1) # shape: (2,8)
```

```
    dB1 = np.sum(dZ1, axis=0, keepdims=True) # shape: (1,8)
```

```
    # Update parameters
```

```
    W1 -= learning_rate * dW1
```

```
    b1 -= learning_rate * dB1
```

```
    W2 -= learning_rate * dW2
```

```
    b2 -= learning_rate * dB2
```



Example in Pytorch

```
class TwoLayerNet(nn.Module):
    def __init__(self, input_dim=2, hidden_dim=8, output_dim=1):
        super(TwoLayerNet, self).__init__()
        self.layer1 = nn.Linear(input_dim, hidden_dim) # W1, b1
        self.layer2 = nn.Linear(hidden_dim, output_dim) # W2, b2

    def forward(self, x): # x shape: (N, 2)
        z1 = self.layer1(x)  a1 = self.relu(z1)  z2 = self.layer2(a1)
        y_hat = self.sigmoid(z2)
        return y_hat
```

2layer NN Binary Cross Entropy ; SGD

```
model = TwoLayerNet() criterion = nn.BCELoss() optimizer = optim.SGD(...)
```

```
for i in range(num_ iterations):           # Training loop
    optimizer.zero_grad()                  # 1. Zero the parameter gradients
    y_pred = model(X_torch)                 # 2. Forward pass
    loss = criterion(y_pred, Y_torch)       # 3. Compute loss
    loss.backward()                          # 4. Backward pass (compute gradients)
    optimizer.step()                         # 5. Update parameters
```



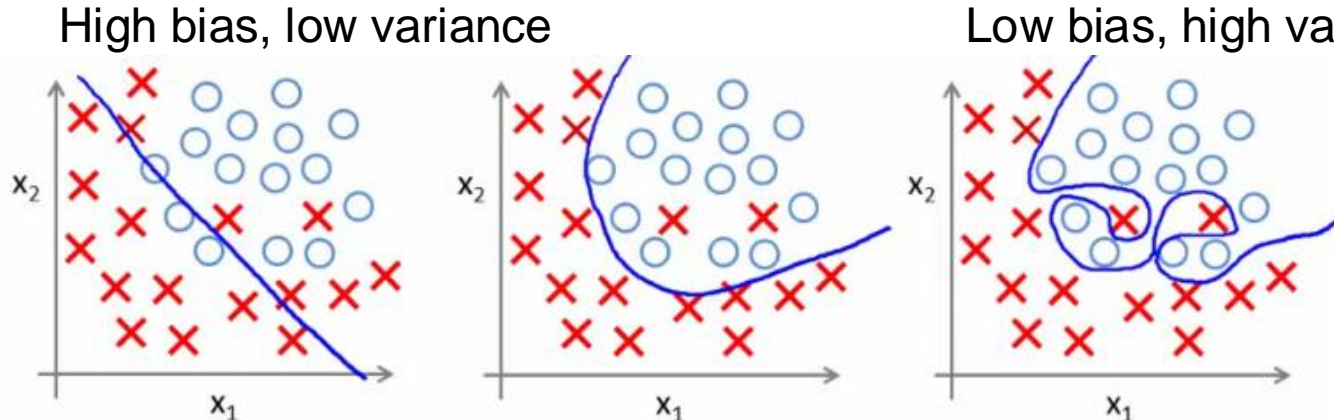
Bias-variance tradeoff

Training loss/ error of learning algorithms has two main components:

- **Bias:** error due to simplifying model assumptions
- **Variance:** error due to randomness of training set

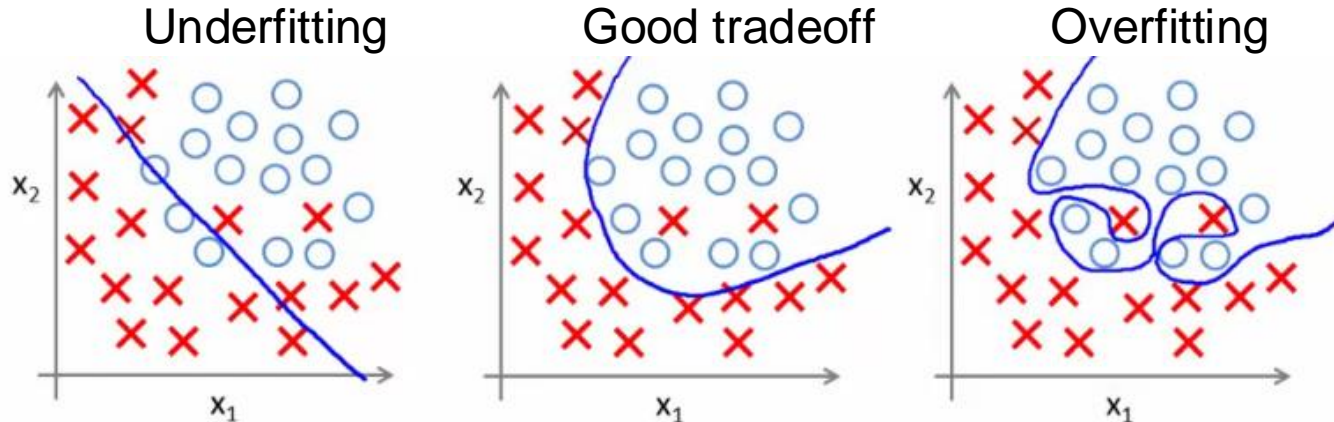
Bias-variance tradeoff can be tuned with hyperparameters during the **validation phase**

- E.g. hyperparameters number of layers, activation type, network architecture, etc.



Under and overfitting

- **Underfitting:** training and test error are both *high*
 - Model does an equally poor job on the training and the test set
 - The model is too “simple” to represent the data or the model is not trained well
- **Overfitting:** Training error is *low* but test error is *high*
 - Model fits irrelevant characteristics (noise) in the training data
 - Model is too complex or amount of training data is insufficient



Best practices for training classifiers

Goal: obtain a classifier with **good generalization** or performance on never before seen data

1. Learn *parameters* on the **training set**
2. Tune *hyperparameters* on the *held out validation set*
3. Evaluate performance on the **test set**

Crucial: do not peek at the test set when iterating steps 1 and 2!

