Lecture 8: Machine Learning II

Linear Classifier

Yen-Yi Ho

Department of Statistics

Outline

- Linear Classifiers
- The Loss Functions

Gene Expression Profiles Predict Clinical Outcomes

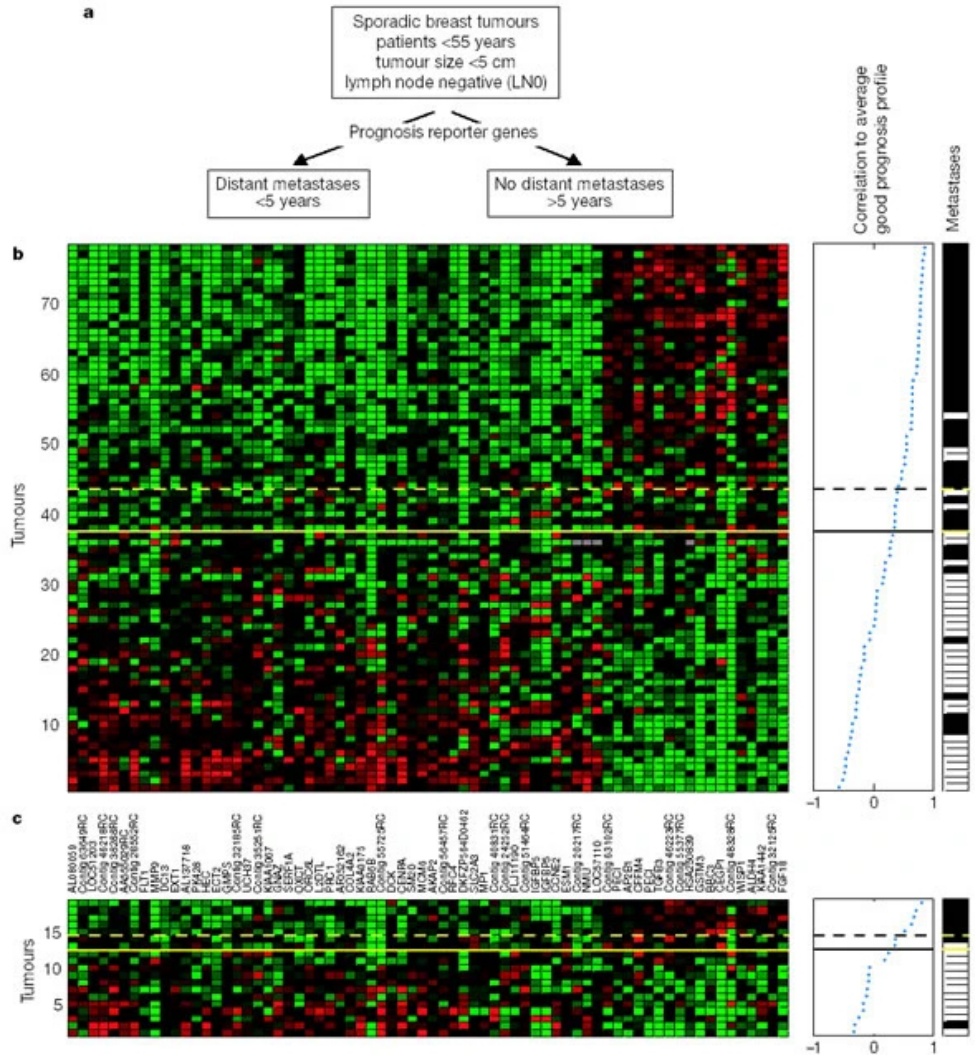


Figure adapted from van 't Veer, L., Dai, H., van de Vijver, M. et al. Gene expression profiling predicts clinical outcome of breast cancer. Nature 415, 530–536 (2002).

Parametric Approach

Input data

$\longrightarrow$ $f(\mathbf{X}, \mathbf{W})$

X: input features

W: parameters / weights

$\longrightarrow$ **Class Scores**

$\rightarrow$ **tumor**

$\rightarrow$ **normal**

Parametric Approach: Linear Classifier

$$f(\mathbf{X}, \mathbf{W}) = \mathbf{X}\mathbf{W} + \mathbf{b}$$

Input data

$\longrightarrow$ $f(\mathbf{X}, \mathbf{W})$ $\longrightarrow$ **Class Scores**

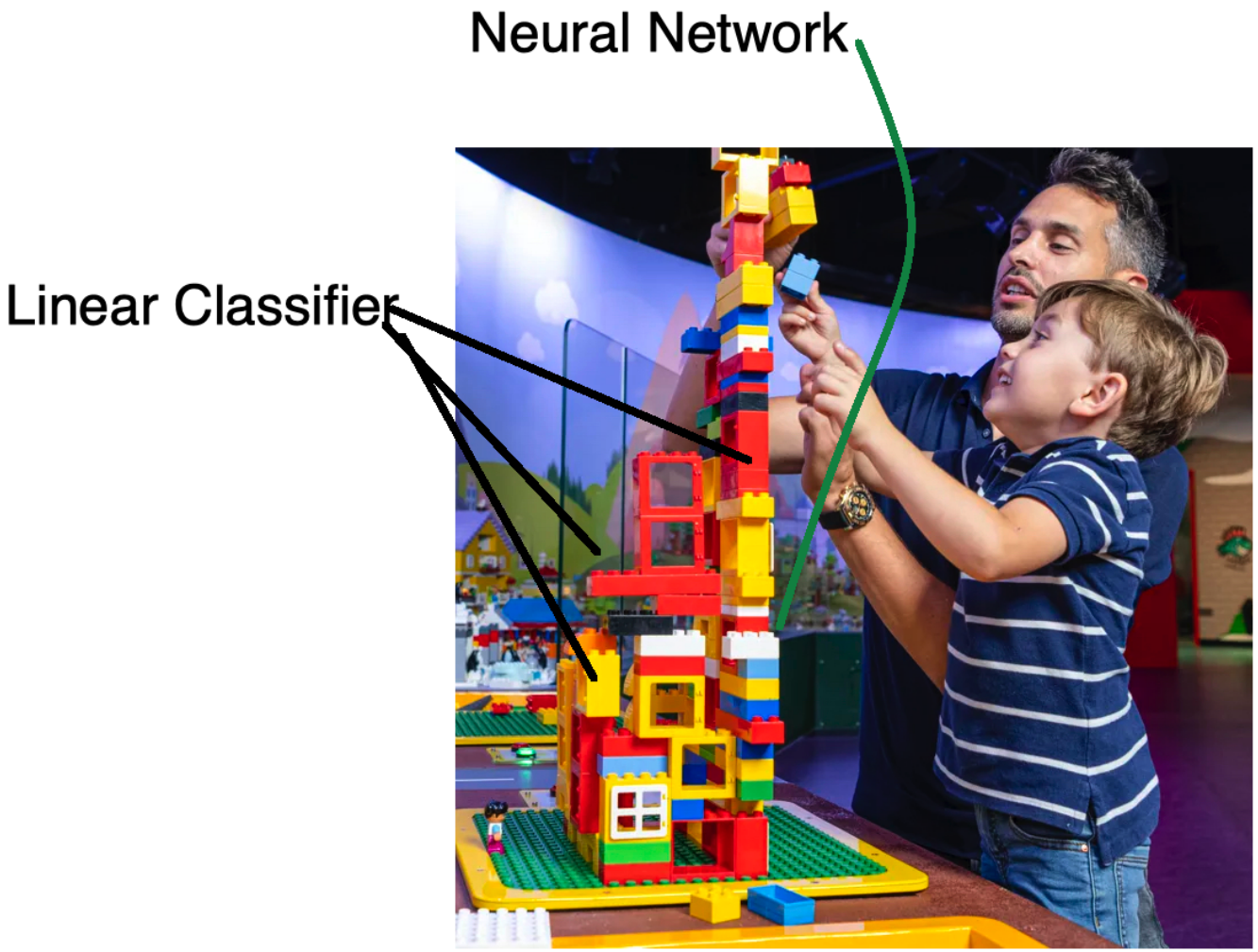
X: input features

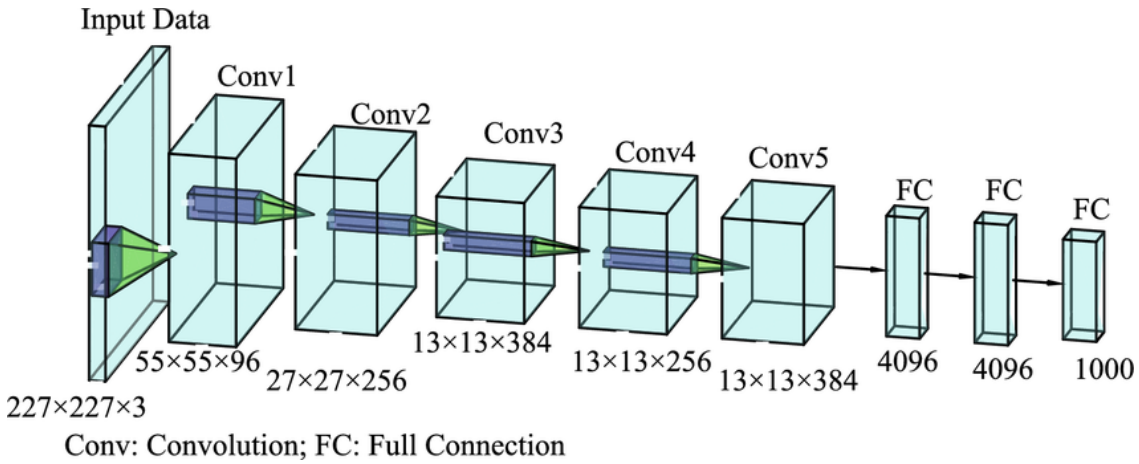
W: parameters / weights

$\rightarrow$ **tumor**

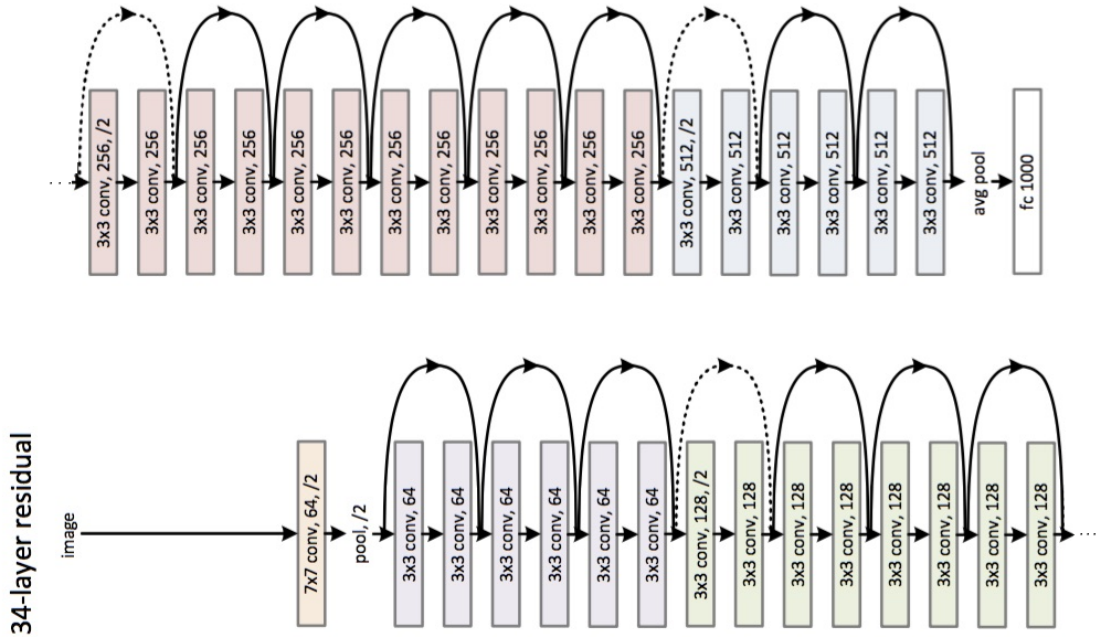
$\rightarrow$ **normal**

Building Blocks of A Neural Network





[Krizhevsky et al. 2012, AlexNet]



[He et al, 2015, ResNet]

Linear Classifier for Disease Prediction

Input

$$\mathbf{x} = \begin{bmatrix} g_1 \\ g_2 \\ g_3 \\ \vdots \\ g_p \end{bmatrix}$$

Gene expression vector for one patient

Linear model

$$s = \mathbf{w}^\top \mathbf{x} + b$$

x: gene expression
w: learned weights

$$p = \sigma(s)$$

(sigmoid)

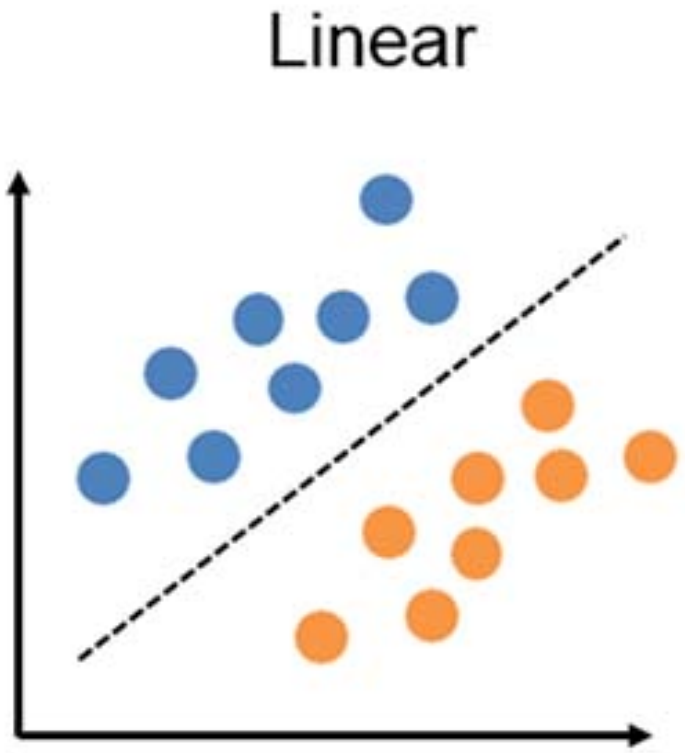
Output

$$p(\text{disease} \mid \mathbf{x})$$

$$\begin{cases} > 0.5 & \Rightarrow \textbf{tumor} \\ \leq 0.5 & \Rightarrow \textbf{normal} \end{cases}$$

Interpreting a Linear Classifier: Geometric Viewpoint

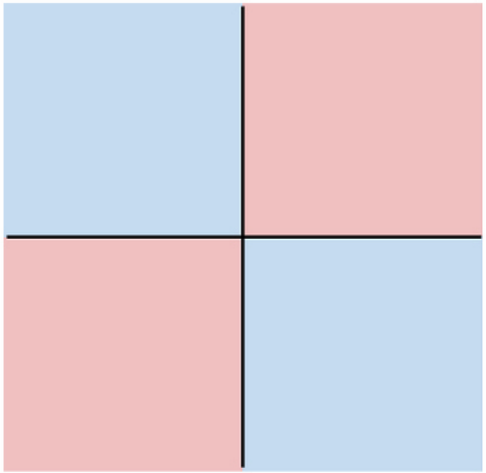
$$f(\mathbf{X}, \mathbf{W}) = \mathbf{XW} + \mathbf{b}$$



Hard Cases for a Linear Classifier

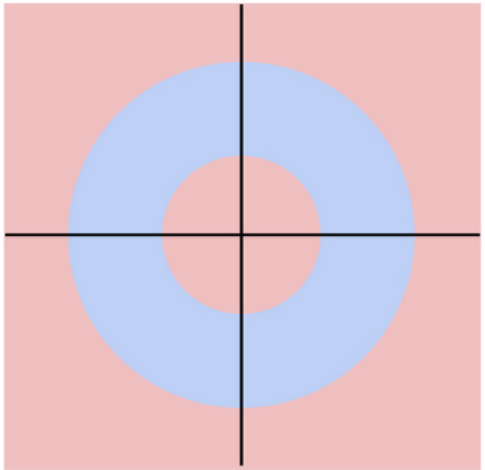
Class 1:
First and third quadrants

Class 2:
Second and fourth quadrants



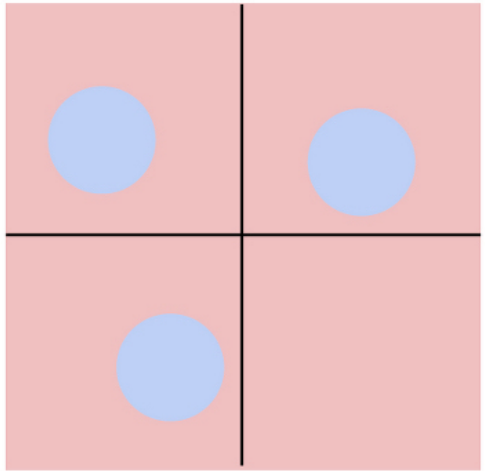
Class 1:
1 <= L2 norm <= 2

Class 2:
Everything else



Class 1:
Three modes

Class 2:
Everything else



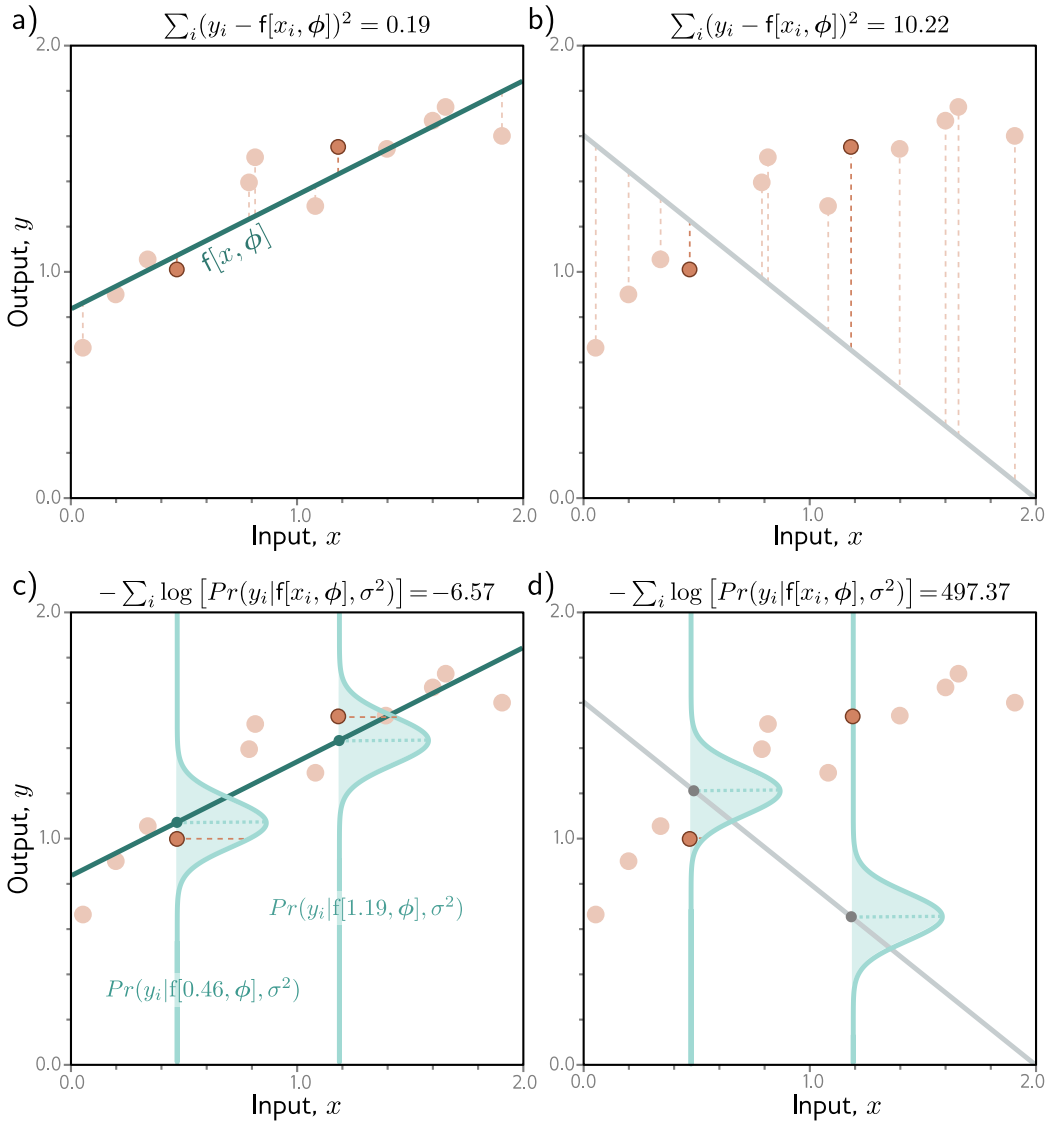
Outline

- The Model
- **The Loss Function**

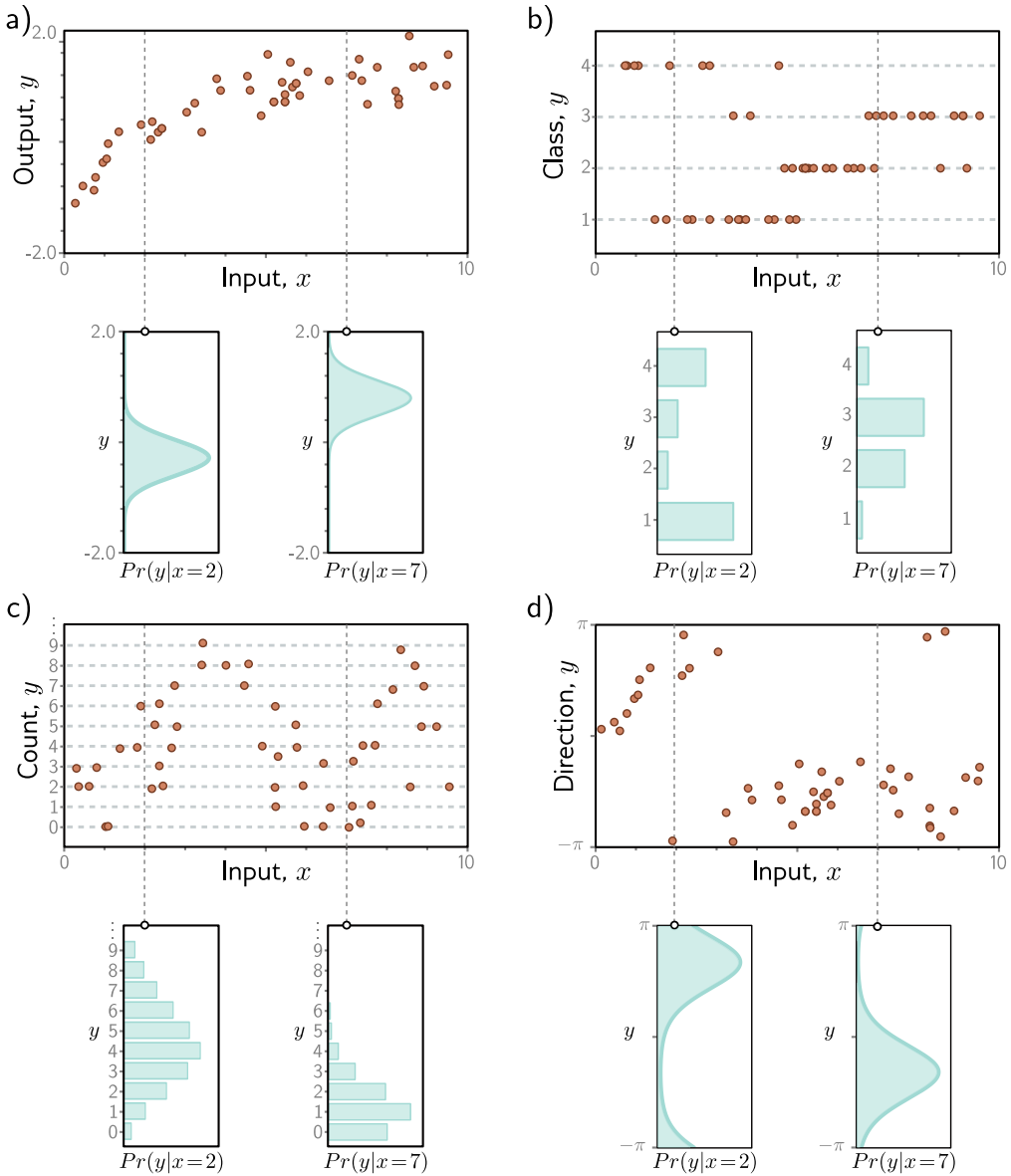
Loss Function

- **Definition:** a measure of error between what your model predicts and what the actual value is.
- **Purpose:** quantifies how well the neural network matches what we want to output and thus guides the optimization process.
- **Importance**: The choice of loss function directly impacts how the weights of the model are adjusted.
- **Examples**: Mean Squared Error (Regression), Cross-Entropy (Classification).

Linear Regression Example



Data Types


a)
Output, y
Input, x
$Pr(y|x=2)$
$Pr(y|x=7)$
b)
Class, y
Input, x
$Pr(y|x=2)$
$Pr(y|x=7)$
c)
Count, y
Input, x
$Pr(y|x=2)$
$Pr(y|x=7)$
d)
Direction, y
Input, x
$Pr(y|x=2)$
$Pr(y|x=7)$

Recipe for Constructing Loss Function

The recipe for constructing loss functions for training data {xi, yi} using the maximum likelihood approach is hence:

- Choose a suitable probability distribution $Pr(y|\theta)$ defined over the domain of the predictions y with distribution parameters θ.
- Set the machine learning model $f(x, \phi)$ to predict one or more of these parameters, so $\theta = f(x, \phi)$ and $Pr(y|\theta) = Pr(y|f[x, \phi])$.
- To train the model, find the network parameters $\hat{\phi}$ that minimize the negative log-likelihood loss function over the training dataset pairs $\{x_i, y_i\}$:

$$\hat{\phi} = \underset{\phi}{\text{argmin}}[L(\phi)] = \underset{\phi}{\text{argmin}}\{-\sum_{i=1}^{I} \log[Pr(y_i|f(x_i, \phi))]\}$$

- To perform inference for a new test example x, return either the full distribution $Pr(y|f[x, \hat{\phi}])$ or the value where this distribution is maximized.

Distributions for Different Prediction Types

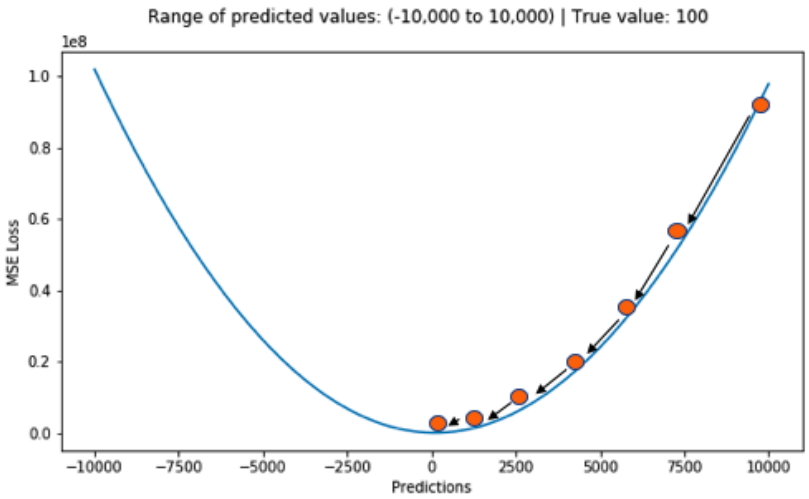
Data Type	Domain	Distribution	Use
univariate, continuous, unbounded	$y \in \mathbb{R}$	univariate normal	regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	Laplace or t-distribution	robust regression
univariate, continuous, unbounded	$y \in \mathbb{R}$	mixture of Gaussians	multimodal regression
univariate, continuous, bounded below	$y \in \mathbb{R}^+$	exponential or gamma	predicting magnitude
univariate, continuous, bounded	$y \in [0, 1]$	beta	predicting proportions
multivariate, continuous, unbounded	$y \in \mathbb{R}^K$	multivariate normal	multivariate regression
univariate, continuous, circular	$y \in (-\pi, \pi]$	von Mises	predicting direction
univariate, discrete, binary	$y \in \{0, 1\}$	Bernoulli	binary classification
univariate, discrete, bounded	$y \in \{1, 2, \ldots, K\}$	categorical	multiclass classification
univariate, discrete, bounded below	$y \in \{0, 1, 2, 3, \ldots\}$	Poisson	predicting event counts
multivariate, discrete, permutation	$y \in \mathrm{Perm}\{1, 2, \ldots, K\}$	Plackett–Luce	ranking

Table: Distributions for loss functions for different prediction types

Loss function for Regression

Mean Squared Error

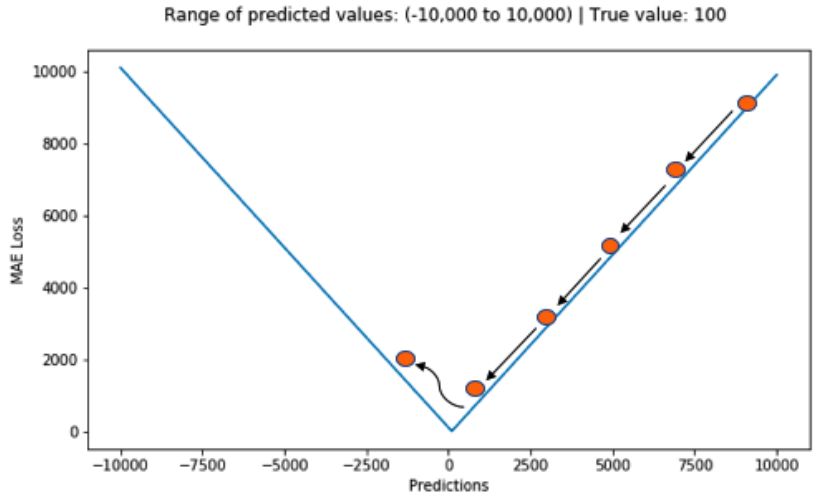
$$MSE = \frac{1}{n}\sum_{i=1}^{I}(y_i - \hat{y}_i)^2$$



Emphasizes larger errors due to squaring, leading to a focus on model accuracy in areas with higher error rates.

Mean Absolute Error (L1 Loss)

$$MAE = \frac{1}{n}\sum_{i=1}^{I}|y_i - \hat{y}_i|$$



Less sensitive to outliers compared to MSE, offering a more robust error metric in datasets with anomalies.

Loss function for Classification

- Binary Classification: Binary Cross-Entropy
- Multi-Class Classification:
  - Cross-Entropy Loss
  - Kullback Leibler Divergence Loss
  - Negative Log Likelihood Loss

Maximizing the likelihood,
or minimizing the negative log-likelihood loss
is the same as minimizing the cross entropy loss.

Binary Cross-Entropy Loss (BCE)

Binary Cross-Entropy Loss (BCE), also called log loss, is used to evaluate the performance of a binary classification model where the output is a probability between 0 and 1.
It measures the dissimilarity between the actual labels and the predicted probabilities of the data points being in the positive class. It penalizes the predictions that are confident but wrong.

$$BCE = -\frac{1}{n}\sum_{i=1}^{I}[y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)].$$

nn.BCELoss in PyTorch

```
# Example predictions and labels
predictions = torch.sigmoid(torch.randn(4))
labels = torch.tensor([1, 0, 1, 0], dtype=torch.float32)

# Binary Cross-Entropy Loss
criterion = nn.BCELoss()
loss = criterion(predictions, labels)
```

```
tensor([0.6882, 0.4268, 0.6981, 0.6192])
```

Should be probability (0 to 1), usually **obtained from a sigmoid function.**

```
tensor([1., 0., 1., 0.])
```

Should be binary (0 or 1), match the shape of predictions.

```
tensor(0.5638)
```

Note: Use `nn.BCEWithLogitsLoss` if the output layer of your model does not include a sigmoid.

$$BCE = -\frac{1}{4}(\log(0.6882) + \log(1 - 0.4268) + \log(0.6981) + \log(1 - 0.6192)) \approx 0.5638$$

Loss function for Classification: Cross-Entropy

Cross-entropy loss measures the performance of a classification model whose output is a probability value between 0 and 1.
Cross-entropy loss increases as the predicted probability diverges from the actual label.

$$CE = \frac{1}{n}\sum_{i=1}^{I}\{-\sum_{c=1}^{M}[y_{i,c}\log(p_{i,c})]\}$$

where M is the number of classes, y_{ic} is the binary indicator showing if class label c is the correct classification for observation i, p_{ic} is the predicted probability output by softmax function, ranging from 0 to 1, in the corresponding class c for observation i. I is the number of observations in the training data.

nn.CrossEntropyLoss in PyTorch

```
# Example predictions and labels
predictions = torch.randn(4, 5) # 4 samples, 5 class predictions
labels = torch.tensor([1, 0, 3, 2], dtype=torch.long)
softmax_predictions = torch.softmax(predictions, dim=1)
# Cross-Entropy Loss
criterion = nn.CrossEntropyLoss()
loss = criterion(predictions, labels)
```

```
tensor([[0.3125, 0.0614, 0.1039, 0.3942, 0.1279],
        [0.2058, 0.3681, 0.2150, 0.1208, 0.0903],
        [0.3293, 0.0893, 0.1760, 0.2105, 0.1949],
        [0.0103, 0.1852, 0.0272, 0.0202, 0.7570]])
```

Predicted probabilities after passing through softmax function.

```
tensor([1, 0, 3, 2])
```

```
tensor(2.3831)
```

$$-\frac{1}{4}(\log(0.0614) + \log(0.2058) + \log(0.2105) + \log(0.0272)) \approx 2.3831$$

The Softmax function

Definition:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{c=1}^{M} e^{z_c}} \qquad (c = 1, \ldots, M)$$

Numeric example:

$$z = [2.0,\ 1.0,\ 0.1]$$

$$\text{softmax}(z) \approx [0.6590,\ 0.2424,\ 0.0986]$$

PyTorch:

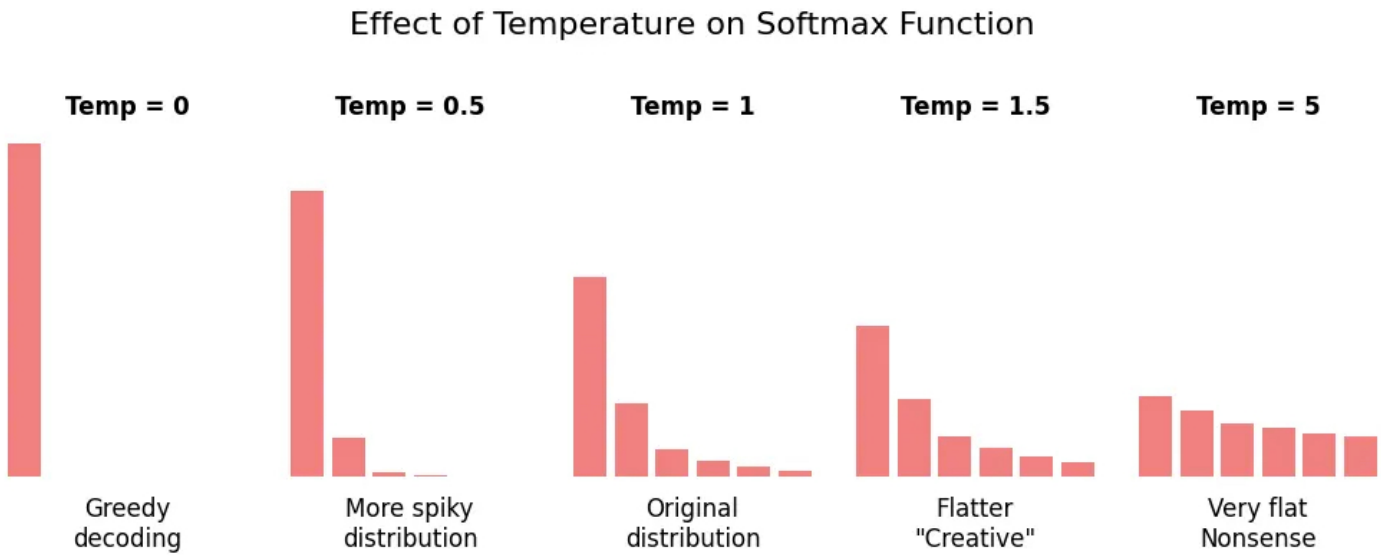
```
import torch, torch.nn.functional as F

z = torch.tensor([2.0, 1.0, 0.1])
probs = F.softmax(z, dim=0)
print(probs)  # tensor([0.6590, 0.2424, 0.0986])
```

Softmax with Temperature

$$\text{softmax}_T(z_i) = \frac{e^{z_i/T}}{\sum_{j=1}^{K} e^{z_j/T}}, \qquad T > 0$$

Effect of temperature T:

- $T = 1$: standard softmax
- $T < 1$: **sharper** distribution (more confident predictions)
- $T > 1$: **softer** distribution (more uniform probabilities)
- $\lim_{T \to 0}$: approaches one-hot (argmax)
- $\lim_{T \to \infty}$: approaches uniform distribution



KL Divergence — definition and intuition

$$D_{\mathrm{KL}}(P \parallel Q) = \sum_{i=1}^{I} P(i) \log \frac{P(i)}{Q(i)}$$

Interpretation:

- Measures how much a distribution Q differs from a reference distribution P.
- Always non-negative: $D_{\mathrm{KL}}(P\|Q) \geq 0$.

Important properties:

- **Not symmetric:**

$$D_{\mathrm{KL}}(P\|Q) \neq D_{\mathrm{KL}}(Q\|P)$$

- Not a true metric (no triangle inequality).
- Use Case Scenario: Effective in model fine-tuning and scenarios where the precise matching of probability distributions is key,e.g.variational autoencoders (VAE) or fine-tuning probability distributions

nn.KLDivLoss in PyTorch

```
# Example predicted and target distributions
predicted_log_probs = torch.log_softmax(torch.randn(4, 5), dim=1)
true_probs = torch.softmax(torch.randn(4, 5), dim=1)

# Kullback-Leibler Divergence Loss
criterion = nn.KLDivLoss(reduction='batchmean')
loss = criterion(predicted_log_probs, true_probs)
```

```
tensor([[-4.4963, -1.2036, -1.5853, -1.4722, -1.3688],
        [-2.6413, -2.2889, -2.2535, -0.5027, -2.1419],
        [-2.3875, -1.0028, -2.9582, -3.1580, -0.8055],
        [-2.3296, -0.4341, -2.2907, -2.1956, -3.1619]])
```

```
tensor([[0.0345, 0.1775, 0.1944, 0.2313, 0.3622],
        [0.1333, 0.6237, 0.1228, 0.0795, 0.0406],
        [0.3026, 0.1969, 0.0661, 0.2365, 0.1979],
        [0.1164, 0.4287, 0.2206, 0.1695, 0.0648]])
```

```
tensor(0.4276)
```

```
1/4*sum(sum(true_probs*(torch.log(
true_probs)-predicted_log_probs)))
```

Data Characteristics & Loss Functions

Consider Data Characteristics:

- Imbalance data: Use weighted Cross-Entropy or Focal Loss.
- Outliers: Use Huber Loss or MAE.

Imbalance data Loss Function

- Weighted Likelihood: Modify the likelihood function to emphasize minority class samples

$$\mathcal{L} = \sum_{i=1}^{N} w_{y_i} \log P(y_i|x_i;\theta)$$

where w_{y_i} is inversely proportional to the class frequency.

- Focal Loss: Focuses on hard-to-classify examples:

$$\mathcal{L}_{focal} = -\alpha(1-p_t)^{\gamma}\log(p_t)$$

where α controls class weighting, and γ modulates the focus on hard examples.