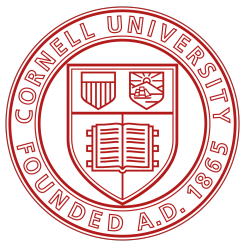
Distributed Optimization for Machine Learning

Lecture 3 - Machine Learning Basics - Part II

Tianyi Chen

School of Electrical and Computer Engineering
Cornell Tech, Cornell University

September 3, 2025



Review: Linear models for House Price predictions

Let's make our price predictor more realistic by adding more features.

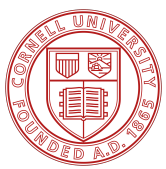
Size (sq. ft.)	**# Bedrooms**	**Age (years)**	**Price ($k)**
1200	3	10	250
2000	4	5	350
800	2	25	150

The model becomes a weighted sum of these features (Assuming $x_0 = 1$):

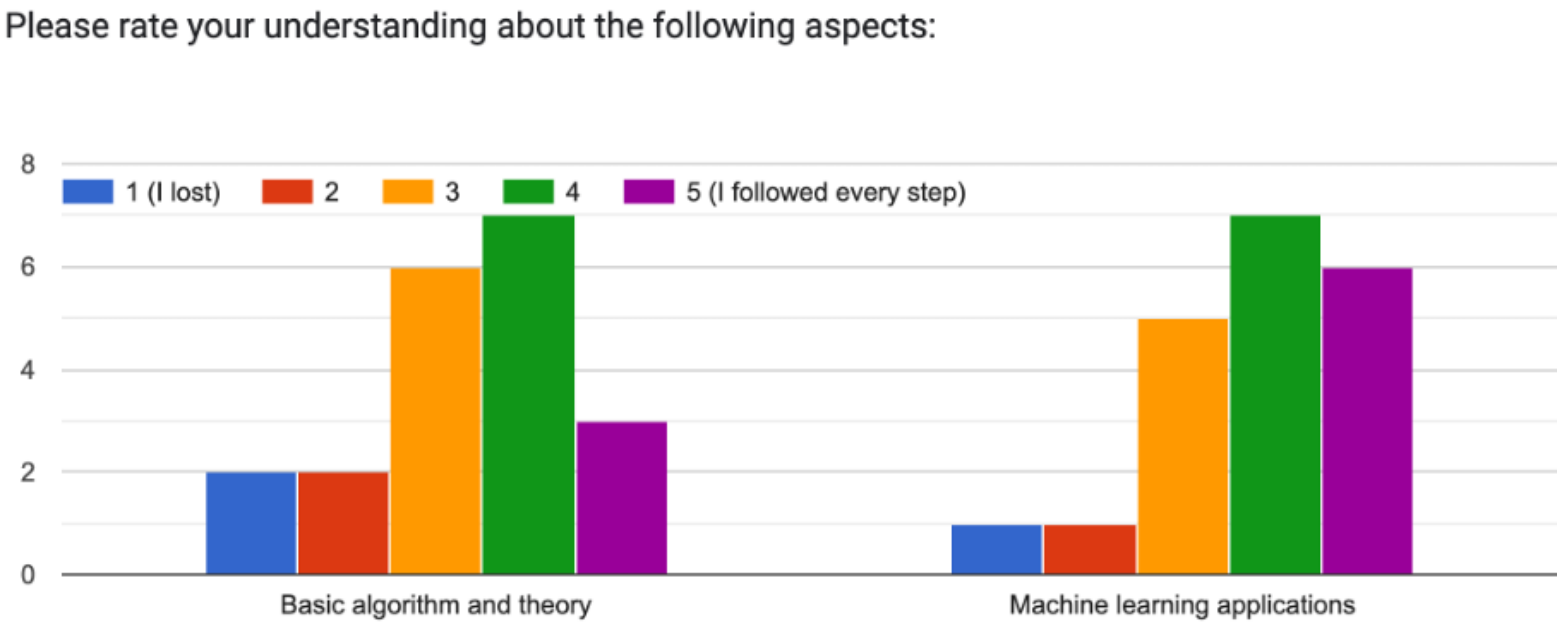
$$h_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 = \boldsymbol{\theta}^T \mathbf{x}$$

Our goal is to find the best parameter vector $\boldsymbol{\theta}$ by minimizing

$$L(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^{m} (h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) - y^{(i)})^2$$



Your understanding of last lecture



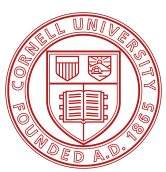
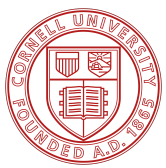


Table of Contents

Other architectures for machine learning

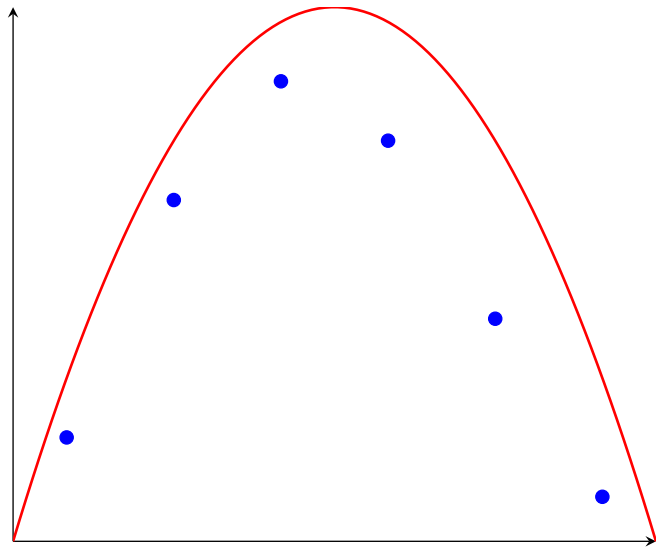
Other settings of machine learning



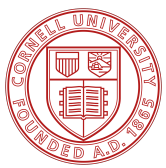
Beyond linear models: Non-linear models

What if the relationship between features and labels isn't a straight line?

A Non-linear Relationship



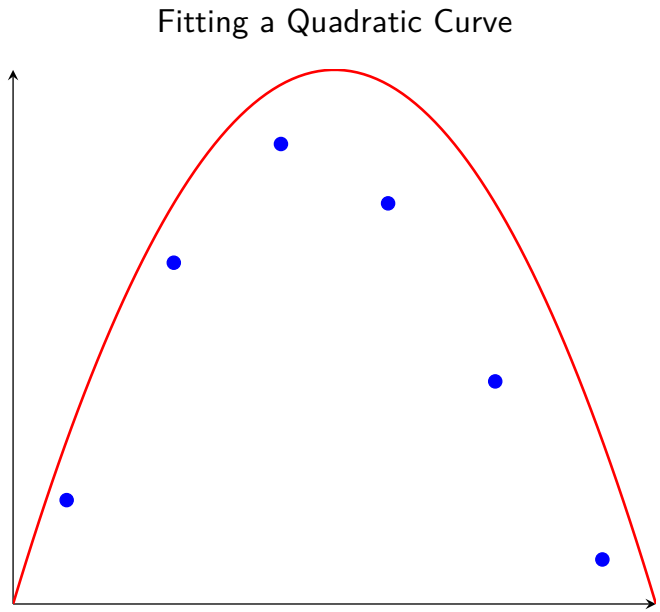
The model $h_{\boldsymbol{\theta}}(\mathbf{x})$ can be any function, like a polynomial, or a complex **neural network**. The core idea remains the same: we find the parameters $\boldsymbol{\theta}$ that minimize the fitting loss.



Solution 1: Polynomial regression

We can create a "non-linear" model by adding polynomial features. We're still fitting a "linear model", but to expanded features like $x, x^2, x^3, \ldots$

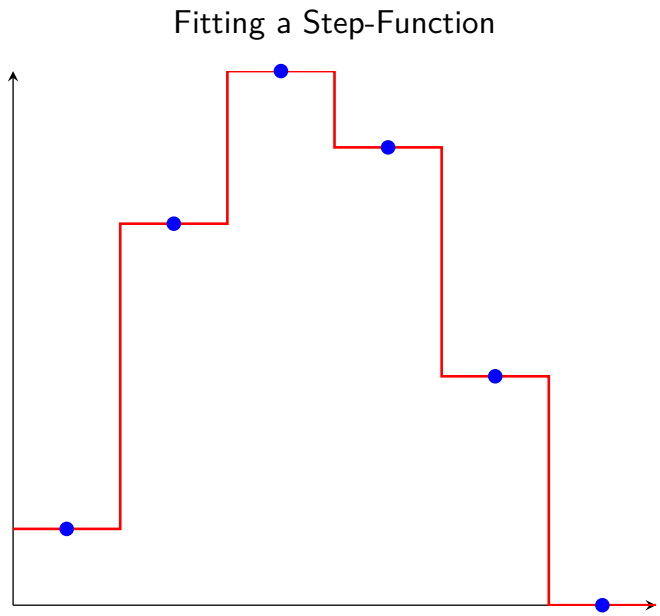
$$h_\theta(x) = \theta_0 + \theta_1 x + \theta_2 x^2$$



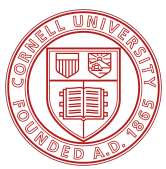
This allows our model to learn curves instead of just straight lines.

Solution 2: Piecewise models (e.g., Trees)

Instead of a global curve, models like **Decision Trees** or **Random Forests** split the data into regions and fit a simpler model (like a constant) in each region.

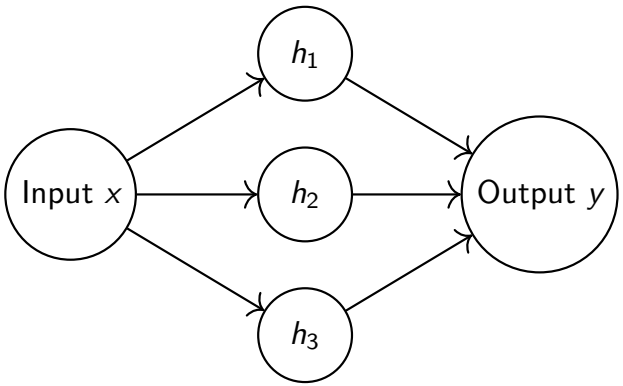


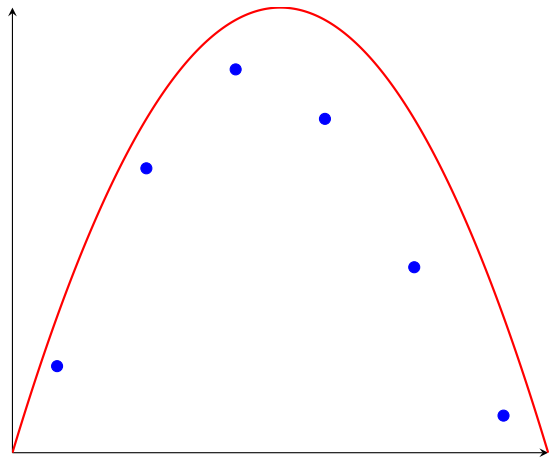
This creates a "step-function" approximation of the curve.



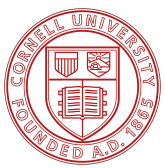
Solution 3: Neural networks

Neural Networks learn complex curves by combining many simple non-linear functions in layers. Can approximate any continuous function.



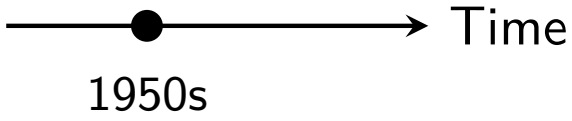


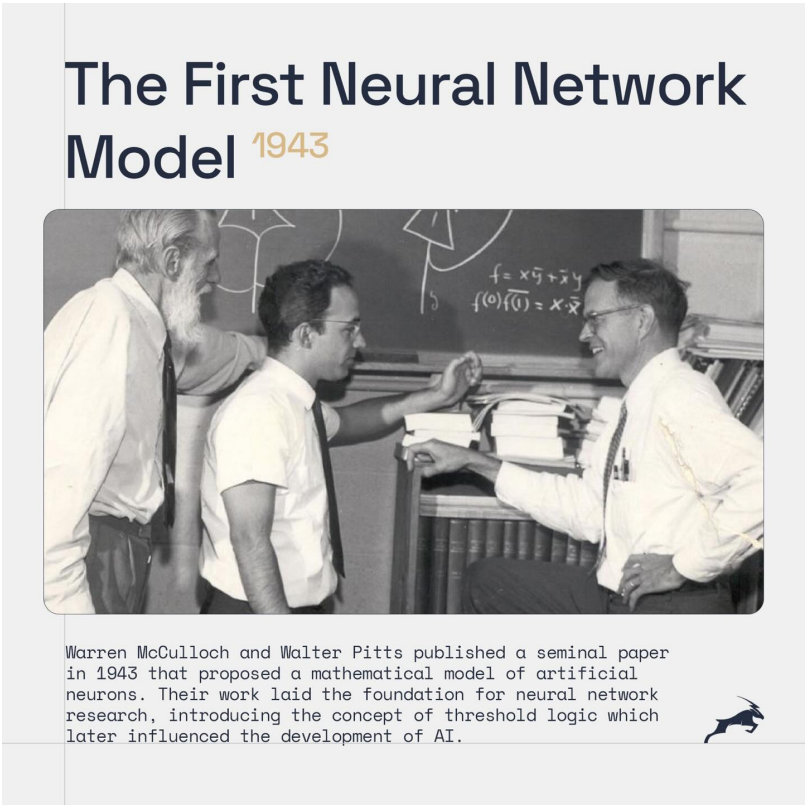
The core idea is the same: even for a complex NN, we still just define a loss function and use optimization to find the best parameter $\boldsymbol{\theta}$.

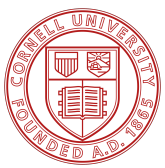


NNs are not "new species"

Perceptron
(single linear unit)



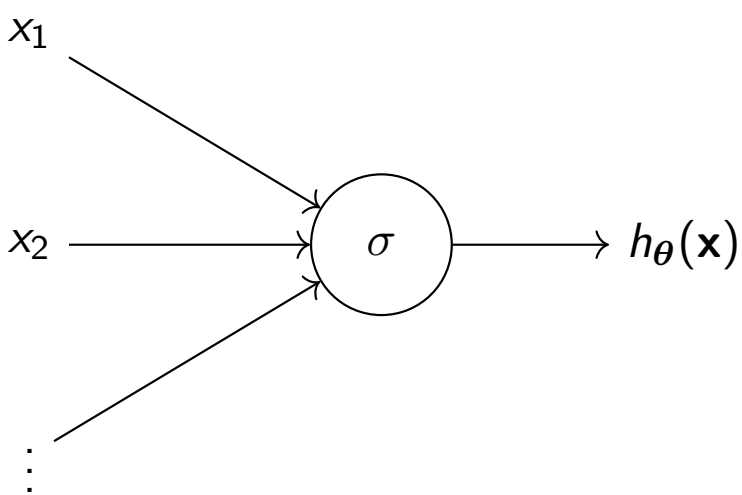




Neural Networks (NNs): The building block

A neural network is built from simple units called **neurons** (or perceptrons). A single neuron:

1. Computes a linear combination of inputs: $\mathbf{w}^T\mathbf{x} + b$.
2. Applies a non-linear **activation function** $\sigma(\cdot)$.

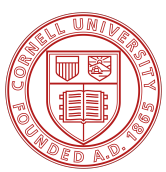


Hypothesis $h_\theta(\mathbf{x})$:

$$h_\theta(\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b)$$

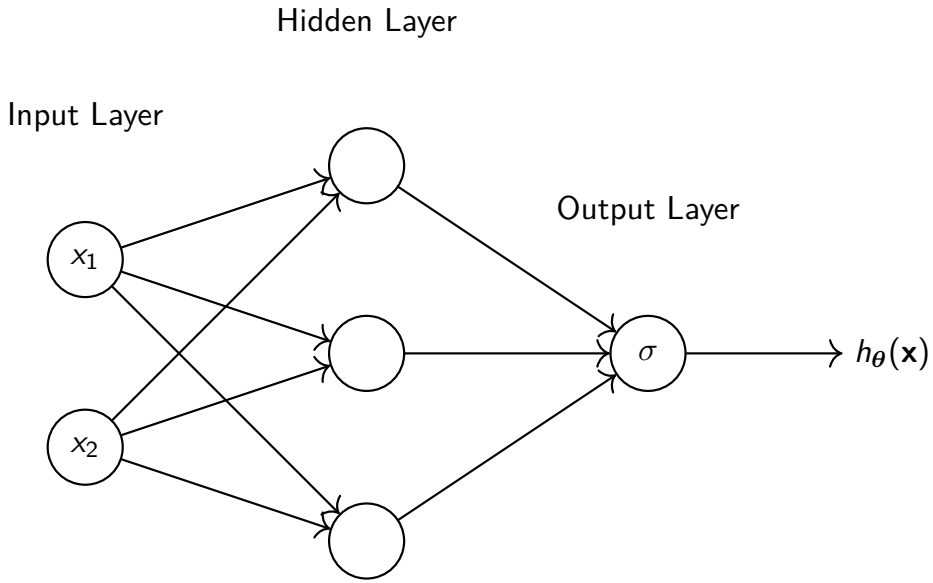
Parameters θ:

- $\mathbf{w} \in \mathbb{R}^d$: weight vector
- $b \in \mathbb{R}$: bias term



Architecture 1: The multilayer perceptron (MLP)

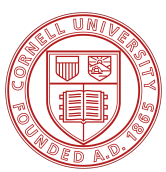
We gain power by arranging neurons in **layers**. The output of one layer becomes the input for the next. This is a **fully-connected** or **dense** NN.



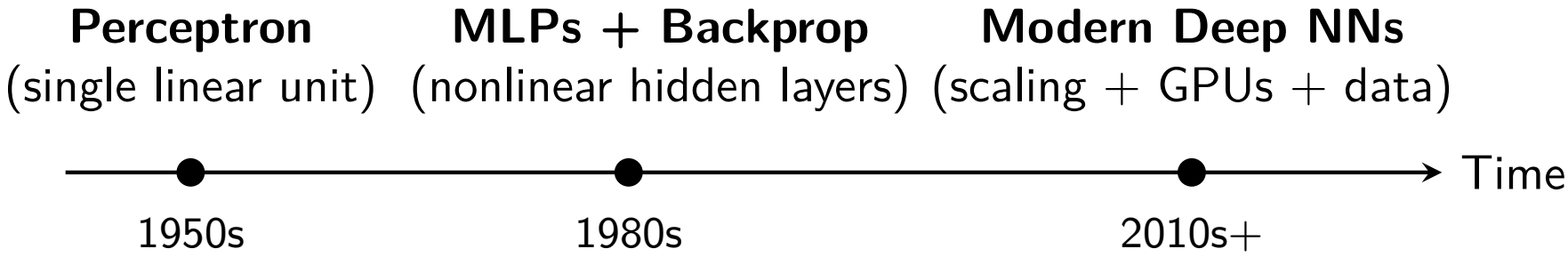
The model $h_{\boldsymbol{\theta}}(\mathbf{x})$ is now a **composition of functions**, allowing it to learn complex, non-linear boundaries.

For one hidden layer: $h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma_2\Big(\mathbf{W}_2\big(\sigma_1(\mathbf{W}_1\mathbf{x} + \mathbf{b}_1)\big) + \mathbf{b}_2\Big)$

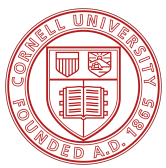
Parameters θ: all weight matrices and bias vectors $\{\mathbf{W}_1, \mathbf{b}_1, \mathbf{W}_2, \mathbf{b}_2\}$.



NNs are not "new species"

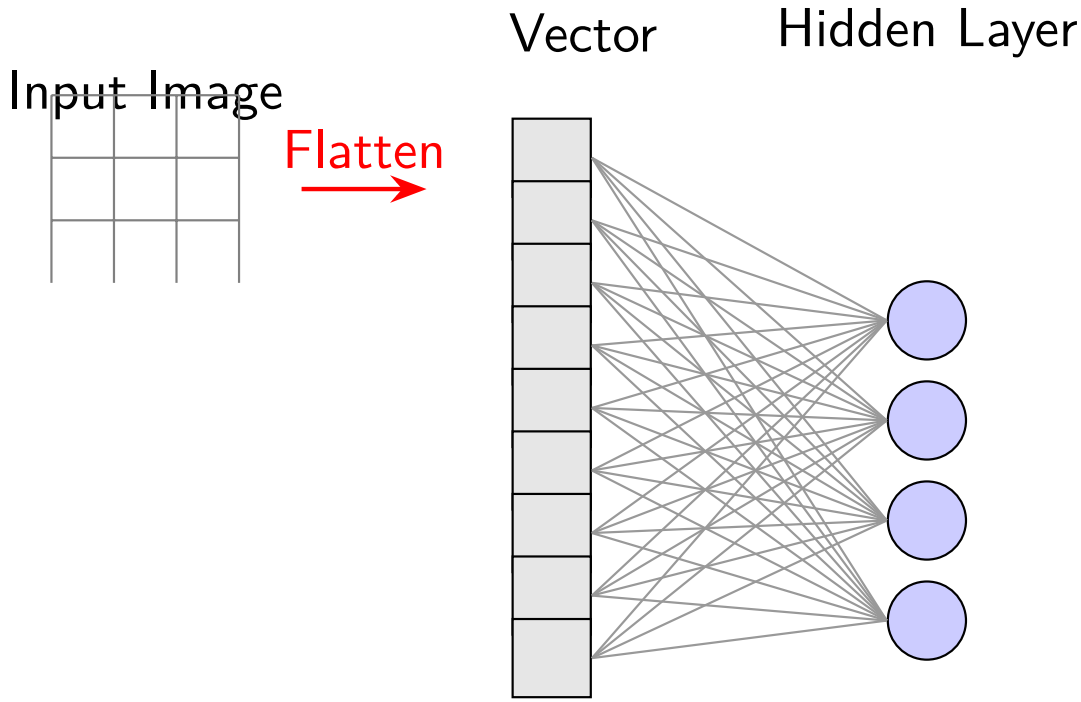


Key message: MLPs/NNs are an evolution of linear models with nonlinearities and scale, not a completely new species.



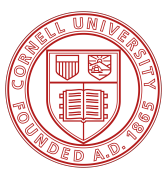
Problem 1: The parameter explosion

To feed an image to MLP, first "flatten" 2D grid of pixels into 1D vector.



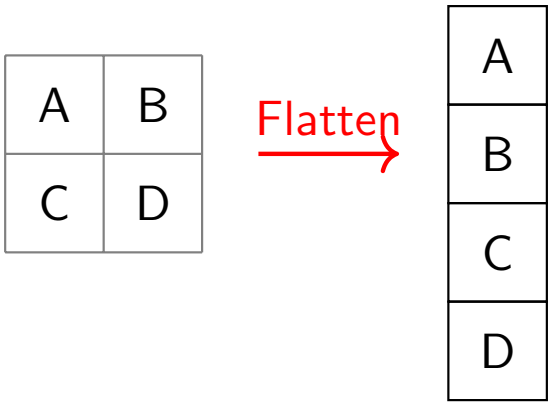
Every input pixel connects to every neuron. For a tiny 28×28 image connecting to a 128-neuron layer, this requires:

$$28 \times 28 \times 128 = \textbf{100,352 weights}$$



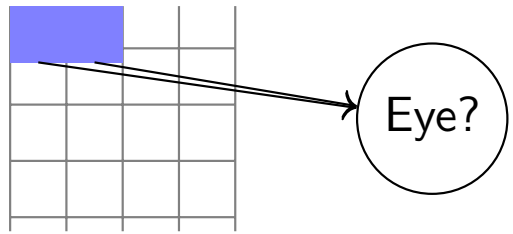
Problems 2&3: Lost structure & redundant learning

Loss of spatial structure

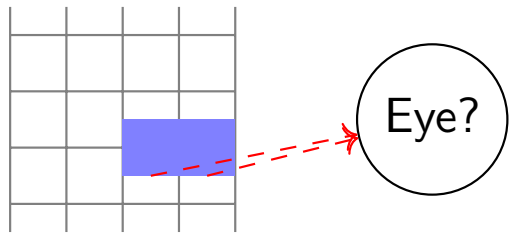


- Flattening an image destroys the 2D grid.
- The model no longer knows that pixel A is above C, or that B is to the right of A.

Not translationally invariant

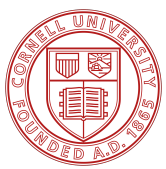


Learns weights for top-left eye



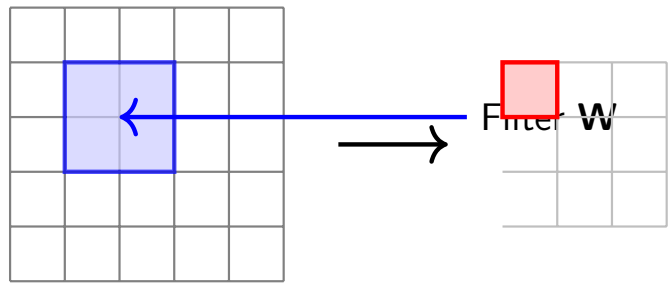
Must learn **new weights** for bottom-right eye

- The network learns weights to detect a pattern in one location.
- Not reuse knowledge elsewhere.



Architecture 2: Convolutional neural networks (CNNs)

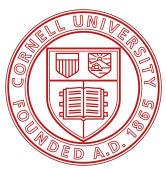
CNNs use a special layer called a **convolutional layer**. This layer applies a small filter (or kernel) across the entire image to detect local patterns.



Model $h_\theta(\mathbf{X})$: The core operation is the 2D convolution $(\mathbf{X} * \mathbf{W})$. The output feature map at position (i, j) is:

$$(\mathbf{X} * \mathbf{W})_{i,j} = \sum_m \sum_n \mathbf{X}_{i-m,j-n} \cdot \mathbf{W}_{m,n}$$

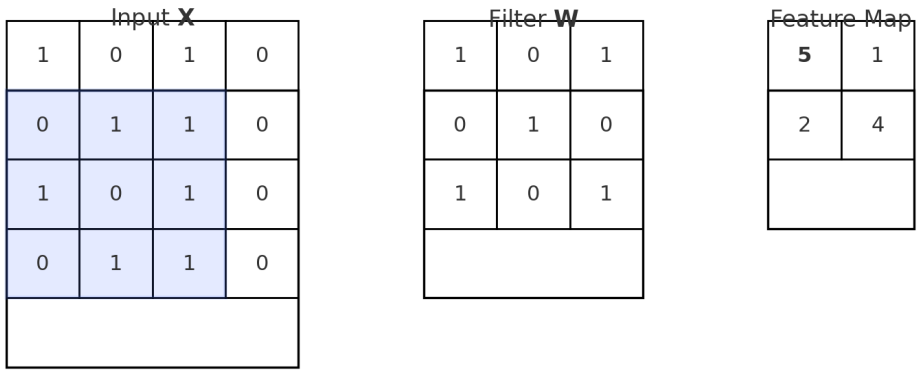
$h_\theta(\mathbf{X})$ is a sequence of such convolutions, activations, and pooling layers.
Parameters θ: The values in the filters (e.g., $\mathbf{W}$) that are learned.



CNN step-by-step computations

Convolution operation:

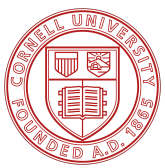
1. Place the filter over a patch.
2. Perform element-wise multiplication and sum the results.
3. Place this in the corresponding cell of **Feature Map**.
4. Slide the filter over and repeat.



Computation for top-left output cell:

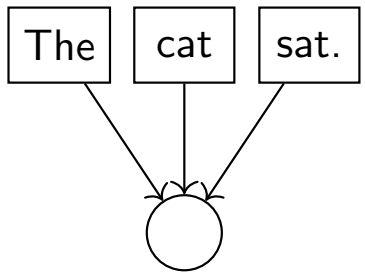
$$(1\times1)+(0\times0)+(1\times1)+(0\times0)+(1\times1)+(1\times0)+(1\times1)+(0\times0)+(1\times1)=\mathbf{5}$$

Key Idea: Parameter sharing. The **same filter W** is used across the entire image. This is incredibly efficient and allows the model to detect a feature no matter where it appears (translation invariance).

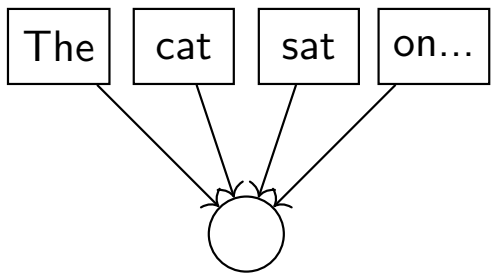


Problem with MLPs for sequential data

Consider the sequential data (like text) and why MLPs are a poor fit.

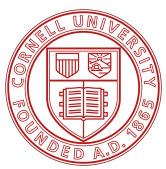


Requires 3 inputs



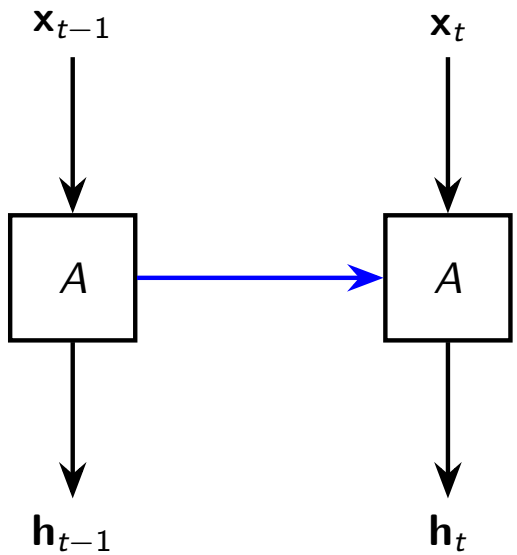
Requires a different architecture!

- **Variable length data:**
  MLPs require a fixed-size input vector, but sequential data like sentences can be any length.
- **No sense of order:**
  It has no built-in notion that "cat" comes after "The," losing crucial contextual information.
- **No parameter sharing:**
  The weights learned for the first word are separate from the weights for the third word.



Architecture 3: Recurrent Neural Networks (RNNs)

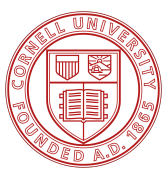
RNNs process sequences by maintaining a hidden state $\mathbf{h}_t$ that acts as a memory, passed from one time step to the next.



Model $h_\theta(\mathbf{x}_1, \ldots, \mathbf{x}_T)$: recurrent update for hidden state $\mathbf{h}_t$, output $\mathbf{y}_t$:

$$\mathbf{h}_t = \sigma(\mathbf{W}_{hh}\mathbf{h}_{t-1} + \mathbf{W}_{xh}\mathbf{x}_t + \mathbf{b}_h)$$
$$\mathbf{y}_t = \sigma(\mathbf{W}_{hy}\mathbf{h}_t + \mathbf{b}_y)$$

Parameters θ: The shared $\{\mathbf{W}_{hh}, \mathbf{W}_{xh}, \mathbf{W}_{hy}\}$ and biases at every step.



Optimization problem for regression

The goal of *training* is to find parameters $\boldsymbol{\theta}$ that **minimize the loss**:

$$\min_{\boldsymbol{\theta}} \; L(\boldsymbol{\theta}) := \frac{1}{2m} \sum_{i=1}^{m} \Big(\underbrace{h_{\boldsymbol{\theta}}\big(x^{(i)}\big)}_{\text{Predicted}_i} - \underbrace{y^{(i)}}_{\text{Actual}_i} \Big)^2 .$$

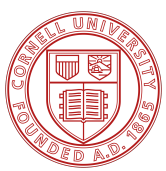
Notes.

- $h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) = f(\mathbf{x}^{(i)}; \boldsymbol{\theta})$ can be *any* model: linear, polynomial, MLP, CNN, etc.
- The gradient can now be written compactly in terms of these residuals:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \frac{1}{m} \sum_{i=1}^{m} \underbrace{e_i}_{\text{Residual}} \cdot \underbrace{\nabla_{\boldsymbol{\theta}} h_{\boldsymbol{\theta}}\big(\mathbf{x}^{(i)}\big)}_{\text{Model's Gradient}}$$

where we define the error for a single data point i as the **residual**:

$$e_i(\boldsymbol{\theta}) = \underbrace{h_{\boldsymbol{\theta}}\big(x^{(i)}\big)}_{\text{Predicted}_i} - \underbrace{y^{(i)}}_{\text{Actual}_i} .$$



Gradient of the MSE Objective

The Mean Squared Error (MSE) objective is the squared L2 norm of $\mathbf{e}$.

$$\text{Let } \mathbf{e}(\boldsymbol{\theta}) = \mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X}) - \mathbf{y} \implies L(\boldsymbol{\theta}) = \|\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X}) - \mathbf{y}\|_2^2 = \mathbf{e}(\boldsymbol{\theta})^T \mathbf{e}(\boldsymbol{\theta})$$

We use the multivariate chain rule to find the gradient with respect to $\boldsymbol{\theta}$.

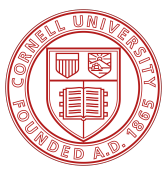
$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = \underbrace{\left(\frac{\partial \mathbf{e}(\boldsymbol{\theta})}{\partial \boldsymbol{\theta}}\right)^T}_{\text{Jacobian Transposed}} \underbrace{\nabla_{\mathbf{e}} L(\boldsymbol{\theta})}_{\text{Gradient w.r.t. residual}}$$

$$\nabla_{\mathbf{e}} L(\boldsymbol{\theta}) = \nabla_{\mathbf{e}} (\mathbf{e}^T \mathbf{e}) = 2\mathbf{e}(\boldsymbol{\theta})$$

$$\frac{\partial \mathbf{e}}{\partial \boldsymbol{\theta}} = \frac{\partial \mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X})}{\partial \boldsymbol{\theta}}$$

Combining the pieces gives the final gradient expression:

$$\nabla_{\boldsymbol{\theta}} L(\boldsymbol{\theta}) = 2 \underbrace{\left(\frac{\partial \mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X})}{\partial \boldsymbol{\theta}}\right)^T}_{\text{Sensitivity}} \underbrace{(\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X}) - \mathbf{y})}_{\text{Error}}$$



Reduction to linear regression (vectorized form)

Given data matrix $\mathbf{X} \in \mathbb{R}^{m \times d}$ and targets $\mathbf{y} \in \mathbb{R}^m$, seek parameters $\boldsymbol{\theta} \in \mathbb{R}^p$ that

$$\min_{\boldsymbol{\theta} \in \mathbb{R}^p} \ \frac{1}{2m} \ \| \ \mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X}) \ - \ \mathbf{y} \ \|_2^2 .$$

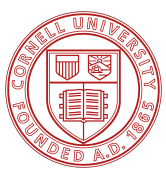
Notes.

- $\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X}) \in \mathbb{R}^m$ stacks predictions $\left(h_{\boldsymbol{\theta}}(\mathbf{x}^{(1)}), \ldots, h_{\boldsymbol{\theta}}(\mathbf{x}^{(m)})\right)^\top$.
- *Linear model special case:* $\mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X}) = \mathbf{X}\boldsymbol{\theta}$, so $\min_{\boldsymbol{\theta}} \frac{1}{2m} \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2$.
- Here

$$\mathbf{J}_{\boldsymbol{\theta}}(\mathbf{X}) = \frac{\partial \mathbf{h}_{\boldsymbol{\theta}}(\mathbf{X})}{\partial \boldsymbol{\theta}} = \mathbf{X} \in \mathbb{R}^{m \times p} \quad \text{and} \quad \mathbf{e}(\boldsymbol{\theta}) = \mathbf{X}\boldsymbol{\theta} - \mathbf{y}.$$

  so

$$\boxed{\nabla_{\boldsymbol{\theta}} \frac{1}{2m} \|\mathbf{X}\boldsymbol{\theta} - \mathbf{y}\|_2^2 = \frac{1}{m} \mathbf{X}^\top (\mathbf{X}\boldsymbol{\theta} - \mathbf{y})}.$$



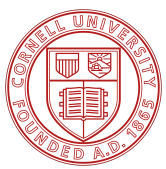
Generic Form of MSE Gradient: An Intuitive View

The gradient tells us how to update the parameters:

$$\nabla_{\theta} L(\theta) = \frac{1}{m} \sum_{i=1}^{m} \underbrace{e_i}_{\substack{\text{How wrong} \\ \text{was I?}}} \cdot \underbrace{\nabla_{\theta} h_{\theta}(\mathbf{x}^{(i)})}_{\substack{\text{How sensitive is the model} \\ \text{to each parameter?}}}$$

Deeper Insights*

- **Probabilistic view:** Why MSE? Loss arises from assuming the errors (e_i) are independent and follow a Gaussian distribution $\mathcal{N}(0, \sigma^2)$ - equivalent to **Maximum Likelihood Estimation (MLE)**.
- **Geometric view:** For linear models, the optimal solution occurs when the error vector **e** is **orthogonal** to the feature space-meaning we've explained all the variance possible with our features.



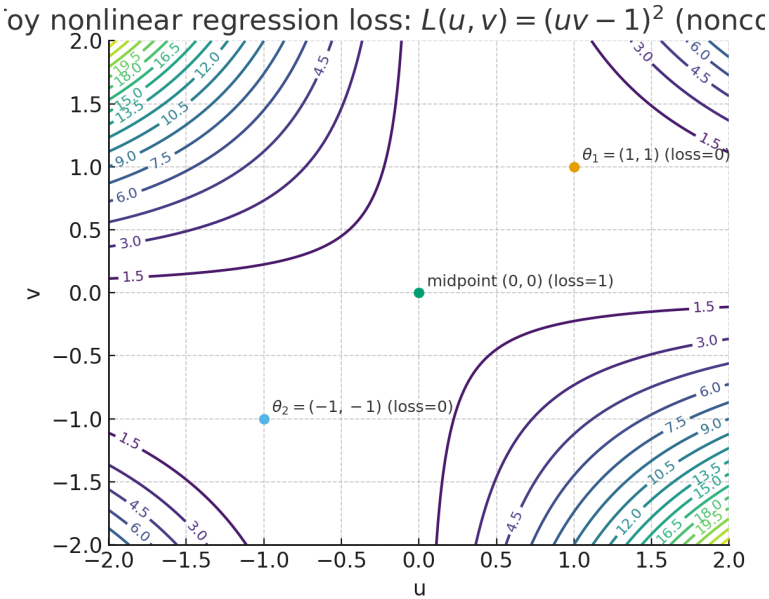
A concrete nonconvex regression loss

Model: $\hat{y}(x; u, v) = (uv)\,x$ with $\theta := [u, v]^\top$, data: $(x, y) = (1, 1)$.

$$L(\theta) = L(u, v) = (uv - 1)^2.$$

Let $\theta_1 = (1, 1)$ and $\theta_2 = (-1, -1)$. Then

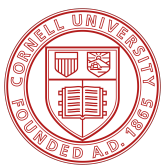
$$L(\theta_1) = 0, \quad L(\theta_2) = 0, \quad L\left(\tfrac{\theta_1+\theta_2}{2}\right) = 1.$$



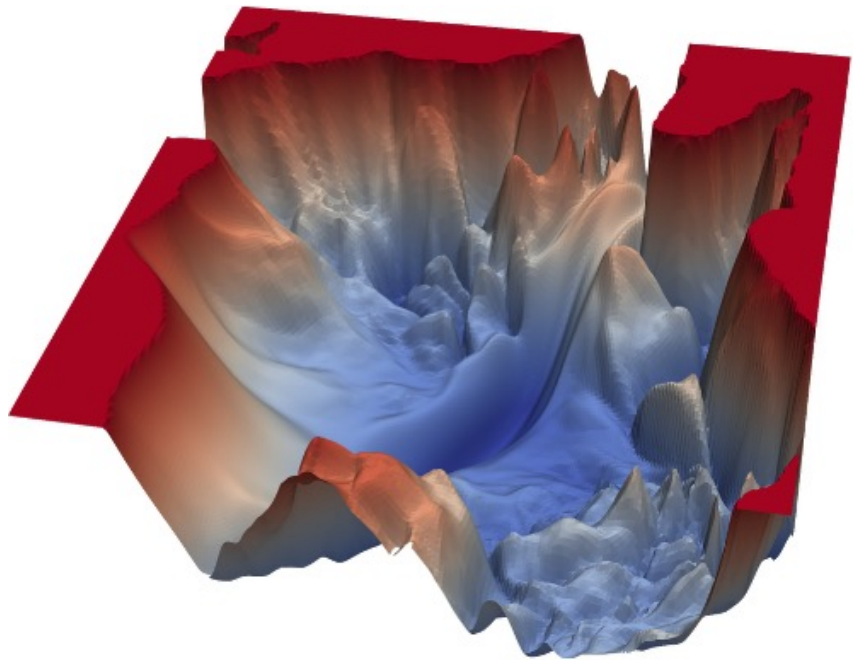
Therefore,

$$L\left(\tfrac{\theta_1+\theta_2}{2}\right) > \tfrac{1}{2}\left(L(\theta_1) + L(\theta_2)\right),$$

so it violates the definition of convexity and thus $L(\theta)$ is **nonconvex**.



Possibly nonconvex landscape of nonlinear regression



Gradient descent on a **nonconvex** loss surface can get pulled into a nearby basin instead of the global minimum.

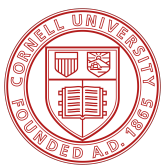
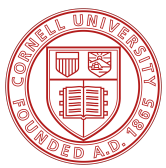


Table of Contents

Other architectures for machine learning

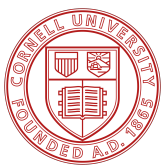
Other settings of machine learning



Marriage and house hunting: A perfect home?

Instead of predicting the exact price, what if

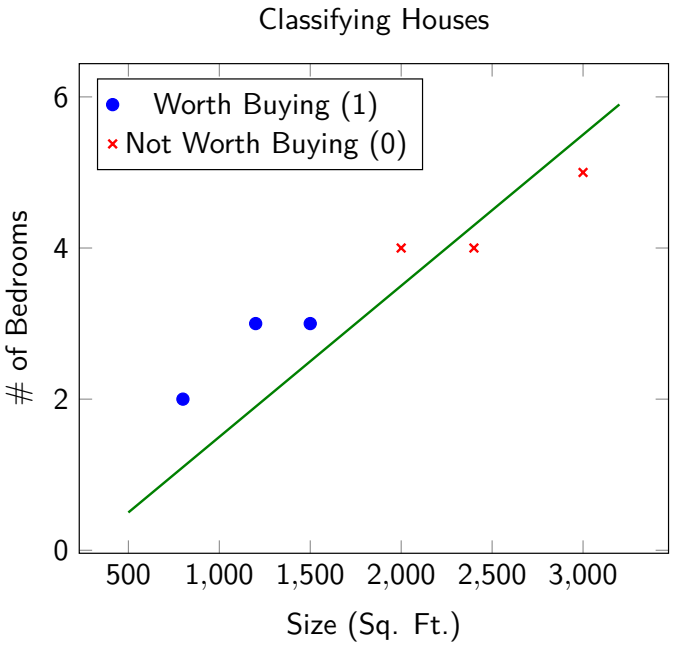
- If Price $<$ \$300k $\rightarrow$ **Worth Buying ($y = 1$)**
- If Price $\geq$ \$300k $\rightarrow$ **Not Worth Buying ($y = 0$)**



Classification - Regression with discrete outputs

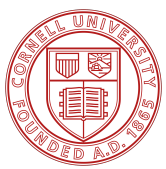
Still a linear model

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 = \boldsymbol{\theta}^T \mathbf{x}?$$



- The model's raw output is a continuous score, not a class. For a house far from the line, this score can be a large positive or negative number.
- This score cannot be interpreted as a probability.

Need a model that squashes its output as probability between 0 and 1.



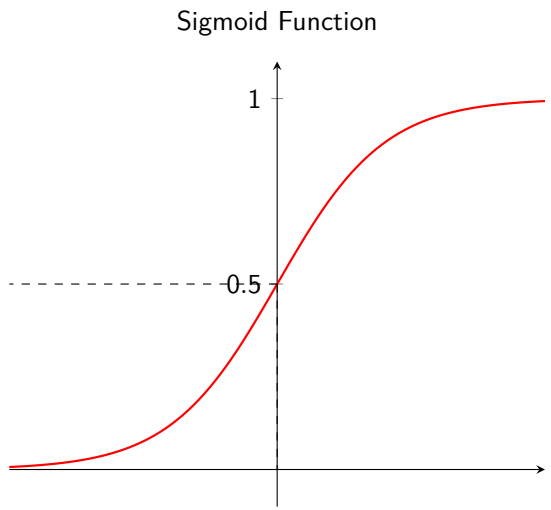
A model for classification

Our model can't just output any number; it should output a **probability** between 0 and 1. We can use the **sigmoid function** $\sigma(z)$ to achieve this.

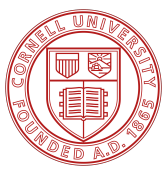
$$\sigma(z) = \frac{1}{1 + e^{-z}}$$

Our model, **Logistic Regression**, first calculates a weighted sum $z = \boldsymbol{\theta}^T \mathbf{x}$, then passes it through the sigmoid:

$$h_{\boldsymbol{\theta}}(\mathbf{x}) = \sigma(\boldsymbol{\theta}^T \mathbf{x})$$



We can then set a threshold (e.g., if probability > 0.5, predict 'buy').



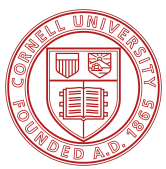
New loss function for classification

Mean Squared Error (MSE) works poorly for classification. Need a loss that penalizes the model for being confidently wrong about a class.
Log Loss (or Binary Cross-Entropy) - when the true label is **buy** ($y = 1$):

$$\text{Loss} = -\log(h_{\boldsymbol{\theta}}(\mathbf{x}))$$

- If the model predicts a high probability (e.g., 0.99), the loss is low: $-\log(0.99) \approx 0.01$. **Good!**
- If the model predicts a low probability (e.g., 0.01), the loss is high: $-\log(0.01) \approx 4.6$. **Bad!**
- In contrast, if the model predicts a low probability (e.g., 0.01), the MSE is MSE $= (0.01 - 1)^2 \approx 0.98$. **Not too Bad!**

LogLoss penalizes much more strongly!



The full log loss (binary cross-entropy)

What about the true label is **don't buy** ($y = 0$)? We want to penalize the model for predicting a high probability - the other side of the log

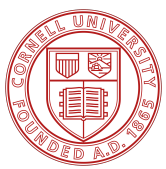
$$\text{Loss} = -\log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}))$$

- If model predicts low probability (e.g., 0.01), loss is low: $-\log(0.99) \approx 0.01$. **Good!**
- If model predicts high probability (e.g., 0.99), loss is high: $-\log(0.01) \approx 4.6$. **Bad!**

Combine both cases into a single, elegant equation for data point $(\mathbf{x}, y)$:

$$\text{Loss}(\boldsymbol{\theta}) = -\Big(y \log(h_{\boldsymbol{\theta}}(\mathbf{x})) + (1 - y) \log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}))\Big)$$

Notice only one term is "active" – on whether $y = 1$ or $y = 0$.



Optimization problem for (binary) classification

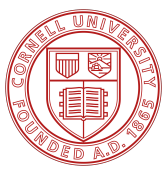
Just like in regression, our goal is to find parameters $\boldsymbol{\theta}$ that **minimize the average loss** over all m training examples:

$$\min_{\boldsymbol{\theta}} \; J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) + (1 - y^{(i)}) \log(1 - h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})) \right]$$

where our model is the sigmoid function: $h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)}) = \sigma(\boldsymbol{\theta}^T \mathbf{x}^{(i)})$.

Key Insight

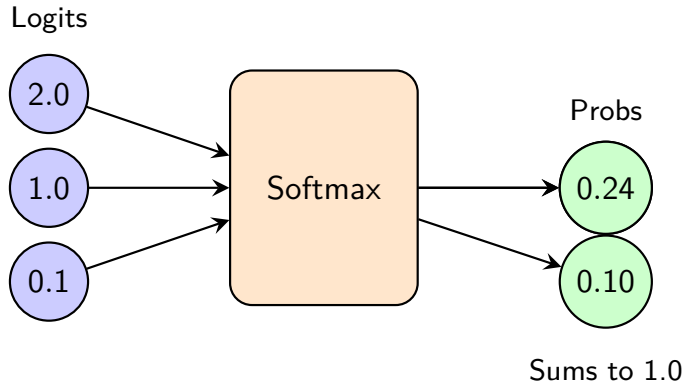
This loss function is **convex**! Unlike the MSE + Sigmoid combination, this formulation guarantees that gradient descent can find the global minimum. This is why Log Loss is the standard for logistic regression.



What if there are more than two options?

Real-world problems often have more than two categories. Instead of a "yes/no" answer, we need to choose from a set of discrete labels.

- Classifying a news article as 'Sports', 'Politics', or 'Tech'.
- Identifying a handwritten digit (0-9).
- Diagnosing a disease from a set of possible conditions.

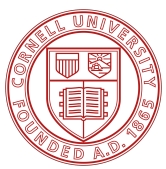


The Model: Softmax Regression

- Instead of the sigmoid function, we use the **softmax** function.
- Softmax takes a vector of scores and turns it into a probability distribution, where all outputs sum to 1.

For K classes, the probability of the j-th class is given by:

$$h_{\boldsymbol{\theta}}(\mathbf{x})_j = P(y = j|\mathbf{x})$$
$$= \frac{\exp(\boldsymbol{\theta}_j^T \mathbf{x})}{\sum_{k=1}^{K} \exp(\boldsymbol{\theta}_k^T \mathbf{x})}$$



Optimization for Multi-class Classification

First, we represent the label $y^{(i)}$ as a **one-hot vector** of size K.

- If a news article is 'Sports' (class 1 of 3), its label is $\mathbf{y}^{(i)} = [1, 0, 0]^T$.
- The model $h_{\boldsymbol{\theta}}(\mathbf{x})$ now outputs a vector of K probabilities.

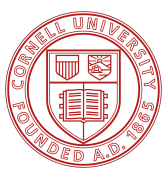
The goal is still to minimize the **Categorical Cross-Entropy**:

$$\min_{\boldsymbol{\theta}} \; J(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} y_k^{(i)} \log(h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})_k)$$

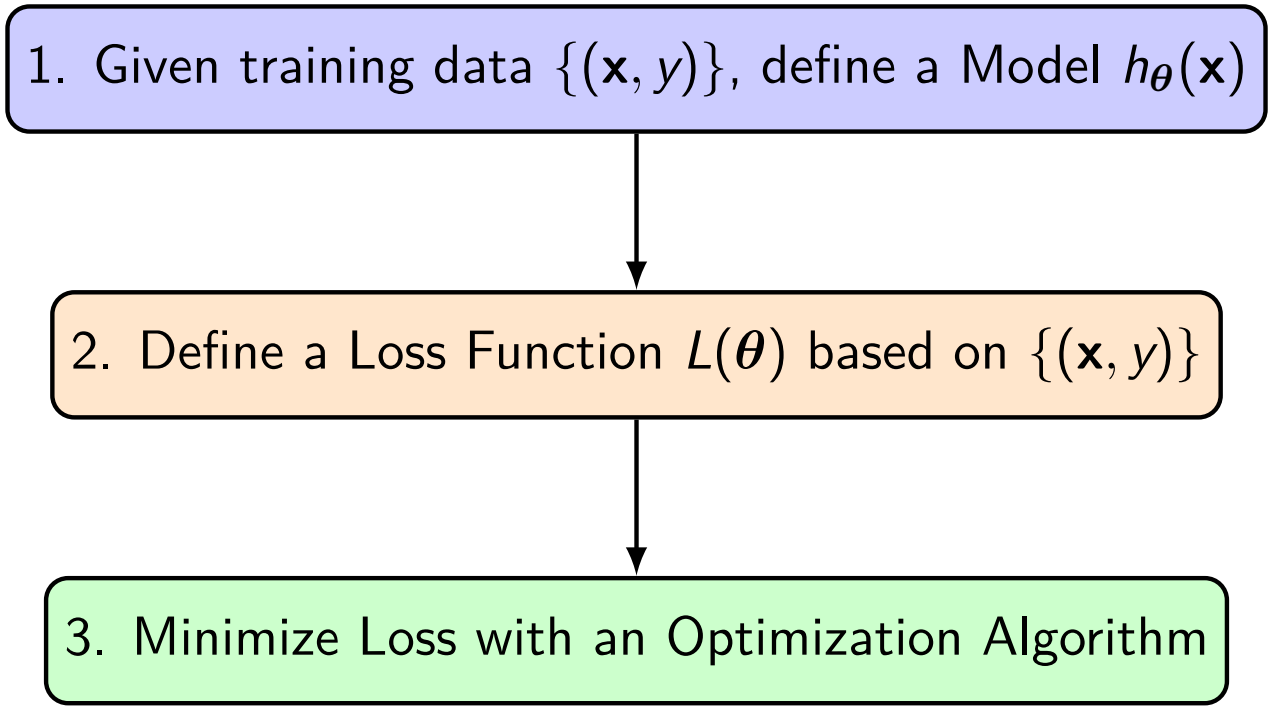
where our model is the **Softmax** function: $h_{\boldsymbol{\theta}}(\mathbf{x}^{(i)})_k = P(y = k | \mathbf{x}^{(i)})$.

Key Insight

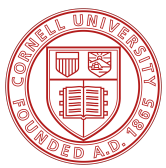
This loss function is also **convex** and differentiable. This formulation guarantees that gradient descent can find the global minimum.



The unified view of supervised learning

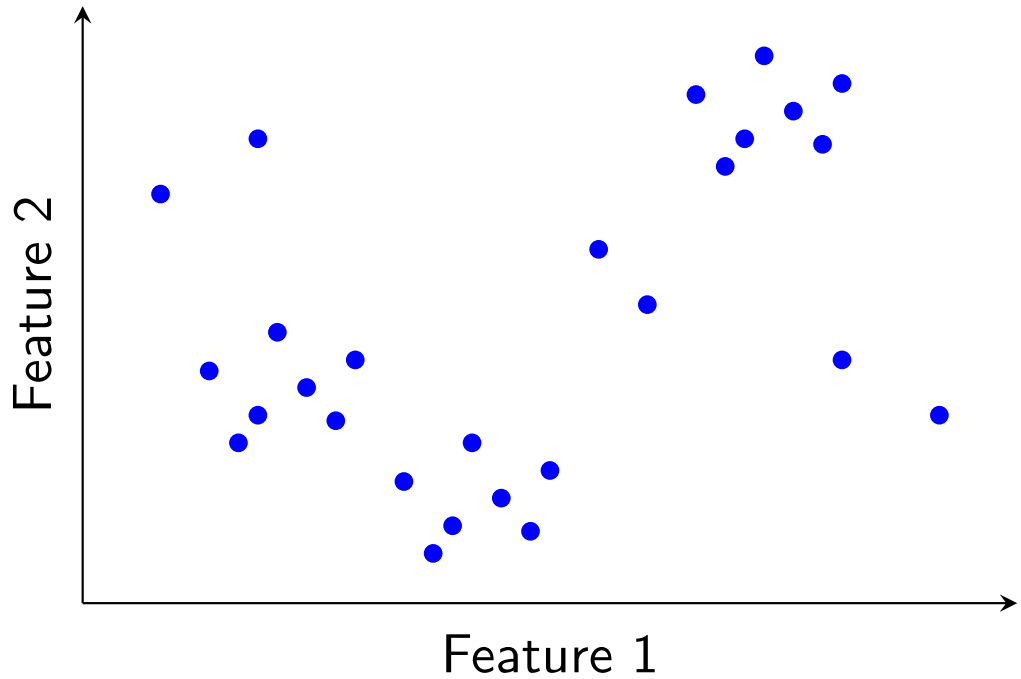


This three-step framework – **Model, Loss, and Optimization** - is the fundamental blueprint for almost all of supervised machine learning.

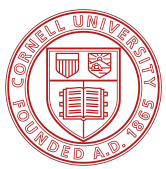


What If We Don't Have Any Labels?

So far, we've assumed we have labeled data (y values) for every house $\mathbf{x}$. This is called **Supervised Learning**.



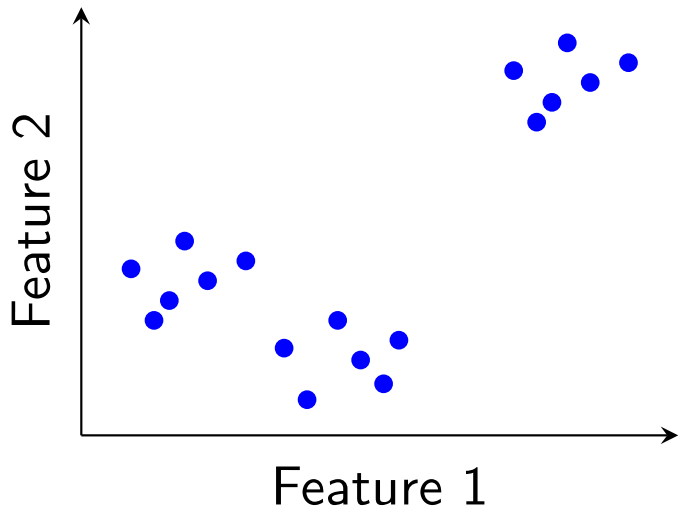
What if we are just given a dataset of houses with their features, but no "Worth Buying" labels? Can we still find meaningful patterns?



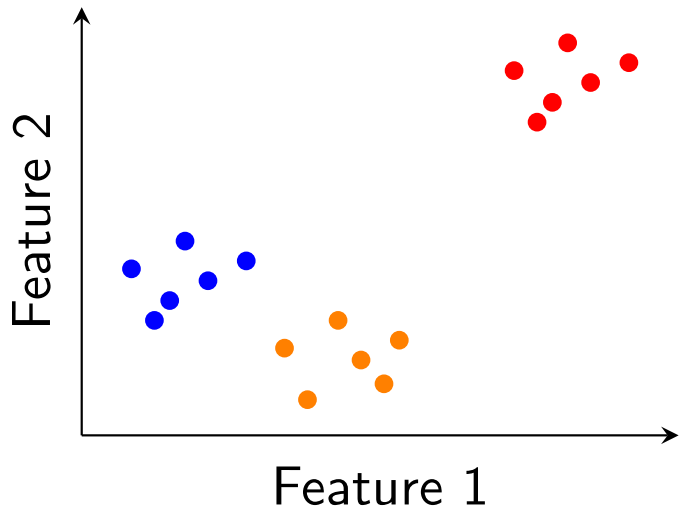
Yes! This is called Unsupervised Learning.

Clustering: Finding natural groups in data

Clustering - grouping objects such that objects in the same group (or **cluster**) are more similar to each other than to those in other clusters.

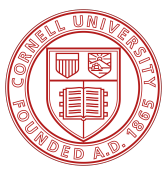


Before Clustering



After Clustering

- **Identifying neighborhood archetypes:** Grouping houses based on features like price, age, and size to discover market segments like "starter homes," "luxury condos," or "historic districts." in **Urban Planning**

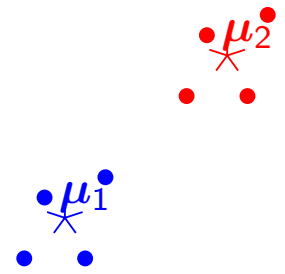


Training Objective for Clustering - K-Means

To define a "good" cluster, we need to find the best cluster assignments **and** the best cluster centers simultaneously.

We jointly optimize two sets of variables:

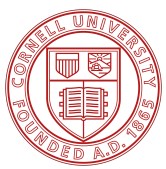
- **Cluster assignments (c):** Let $c^{(i)}$ be the index of the cluster assigned to point $\mathbf{x}^{(i)}$.
- **Cluster centroids ($\boldsymbol{\mu}$):** Let $\boldsymbol{\mu}_k$ be the center of the k-th cluster.



The complete objective is to minimize the sum of squared distances from each point to its assigned centroid:

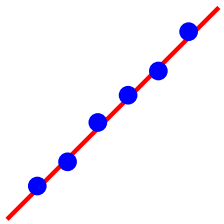
$$\min_{c,\boldsymbol{\mu}} \; J(c, \boldsymbol{\mu}) = \sum_{i=1}^{m} ||\mathbf{x}^{(i)} - \boldsymbol{\mu}_{c^{(i)}}||^2$$

This joint objective is non-convex and NP-hard to solve directly. Instead, we use the **K-Means algorithm** which alternates between two steps.



Summary: Three Classic ML Paradigms

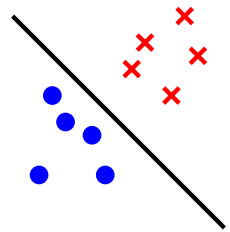
Regression



Predict a **continuous** value.

Example: What is the exact price of this house?

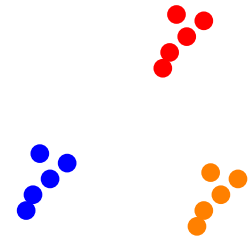
Classification



Predict a **discrete** label.

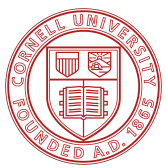
Example: Is this house a "good deal" (yes/no)?

Clustering



Find hidden **groups** in data.

Example: What are the natural market segments of houses?



Self-Supervised Learning (SSL): The Core Idea

We have massive unlabeled datasets. How can we learn from them without human labels?

The Core Idea: We create a "puzzle" or a **pretext task**, and force the model to understand data's underlying structure by solving this puzzle.

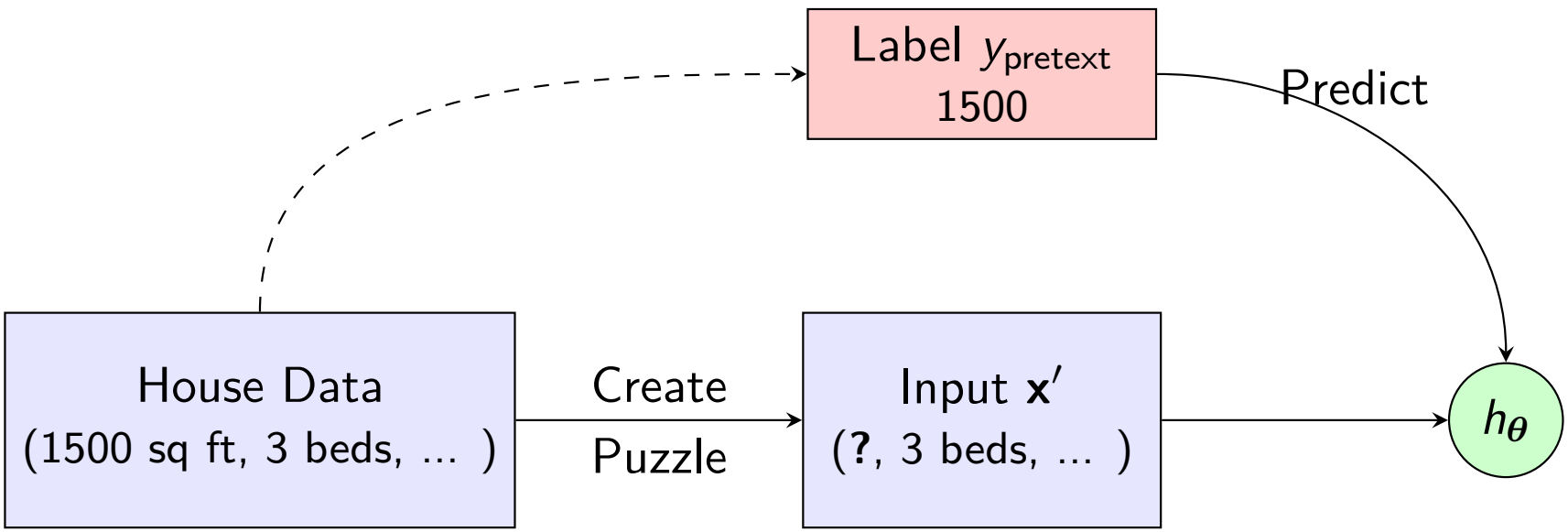
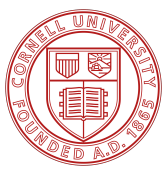


Figure: Learn by predicting the piece of data that was intentionally hidden.

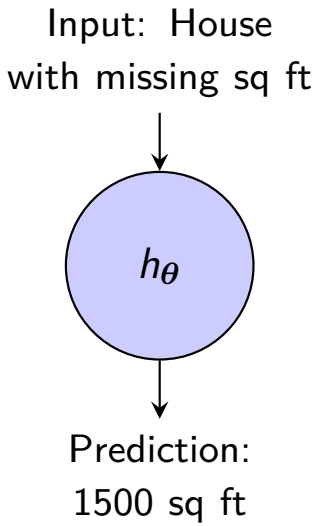


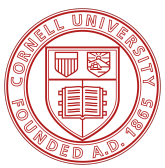
SSL in Action: A Pretext Task for Housing Data

Let's design a pretext task for our unlabeled housing dataset.

Example: House Feature Prediction

1. **Start with a complete data point:**
   A house with features like (1500 sq ft, 3 beds, ...).
2. **Create the input and pseudo-label:**
   We "mask" or hide one feature, like the square footage.
   - Input $\mathbf{x}'$: ([?], 3 beds, ...).
   - Pseudo-Label y_{pretext}: 1500.
3. **The Task:** Train the model to predict the original value (1500) from the rest of the features.





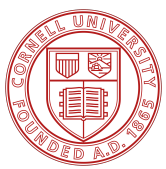
The SSL Objective & The Ultimate Goal

We can generate millions of these "pseudo-labeled" examples, and the Training Objective is to minimize a supervised loss (e.g., Cross-Entropy) on the pretext task with $y^{(i)}_{\text{pretext}}$ from the unmasked part of the data

$$\min_{\boldsymbol{\theta}} \; J(\boldsymbol{\theta}) = \sum_{i=1}^{m} \text{Loss}(h_{\boldsymbol{\theta}}(\mathbf{x}'^{(i)}), \mathbf{y}^{(i)}_{\text{pretext}})$$

Transfer learning via fine-tuning

The final model $h_{\boldsymbol{\theta}}$ that predicts square footage is usually not our goal. We can then slightly adjust (**fine-tune**) $\boldsymbol{\theta}$ for our real, downstream task (like classifying "Good Deals") using our small set of human-labeled data.



What If We Have a Small Set of Labeled Data?

Now, let's say we have a **small, precious set of labeled data** for our actual task (e.g., 100 houses labeled as "Good Deal").

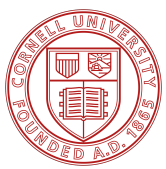
Strategy A: Fine-Tuning

We can take our pre-trained model and simply **fine-tune** it using only our 100 labeled examples. This is a standard two-step process.

Strategy B: Semi-Supervised

Alternatively, we can train a model using both the 100 labeled examples and the millions of unlabeled examples **at the same time**.

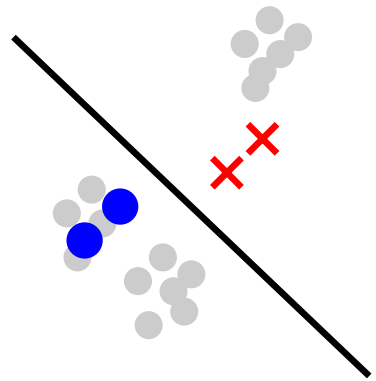
Semi-Supervised Learning is a strategy that combines labeled and unlabeled data in a single training process to learn a better model.



The Intuition: Use Unlabeled Data as a Guide

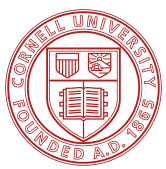
The key idea is to leverage a large number of **unlabeled** houses to reveal the true structure of the market.

The Cluster Assumption: A good decision boundary should not pass through high-density areas. It should lie in the "gap" between natural groups.



This means that nearby points are likely to have the same label.

- A 1500 sq ft house and a nearly identical 1505 sq ft house should get the same prediction.



Semi-Supervised Training Objective

- 1. Supervised loss. This is the standard Binary Cross-Entropy loss, but calculated **only on the small set of labeled data (L)**.

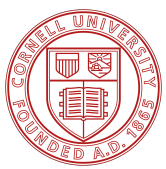
$$L_{\text{sup}}(\boldsymbol{\theta}) = \sum_{(\mathbf{x},y)\in L} \text{BCE}(h_{\boldsymbol{\theta}}(\mathbf{x}), y)$$

- 2. Unsupervised Consistency Loss. This loss is calculated on the **large set of unlabeled data (U)**. It penalizes if the model gives different predictions for $\mathbf{x}$ and its augmented version augment($\mathbf{x}$).

$$L_{\text{unsup}}(\boldsymbol{\theta}) = \sum_{\mathbf{x}\in U} ||h_{\boldsymbol{\theta}}(\mathbf{x}) - h_{\boldsymbol{\theta}}(\text{augment}(\mathbf{x}))||^2$$

The total loss is a weighted sum, where λ controls the importance of the consistency term

$$\min_{\boldsymbol{\theta}} \; J(\boldsymbol{\theta}) = L_{\text{sup}}(\boldsymbol{\theta}) + \lambda L_{\text{unsup}}(\boldsymbol{\theta})$$



Recap and fine-tuning

- What we have talked about today?
  - $\Rightarrow$ How to extended from linear to nonlinear regression?
  - $\Rightarrow$ What are the related optimization challenges?
  - $\Rightarrow$ How about other loss functions/types of machine learning?

Welcome anonymous survey!