CSE 676 Deep Learning

Preliminaries and Linear Neural Networks

Jue Guo[1]

University at Buffalo

[1] The materials are adapted from *Dive into Deep Learning*

Contents

Course Overview

An Overview of Artificial Intelligence

Preliminaries
- Data Manipulation
- Linear Algebra
- Calculus
- Automatic Differentiation
- Probability and Statistics

Linear Neural Networks for Regression

Course Logistics I

Course Instructor: *Jue Guo* [C]

- **Research Area**: Optimization for machine learning, Adversarial Learning, Continual Learning and Graph Learning
- Interested in participating in our research? Reach to us by email.

Course Hours:

- Session [C]
- Time: Tuesday and Thursday 2:00PM-3:20PM

Office Hours:

- My office hours: 3:00pm - 4:00pm on Friday

Course Outline

- **Week 1 and Week 2**
  - Math, Machine Learning Review and Linear Regression
- **Week 3 and Week 4**
  - Review on Linear Regression, Softmax Regression and MLP
- **Week 5 and Week 6**
  - CNN and *Efficient-Net Paper* Reading
- **Week 7** (One Class)
  - Midterm (Coverage on Week 1,2,3,4)
- **Week 8 and Week 9**
  - Recurrent Neural Networks and *Paper Read on Transformer*
- **Week 10, Week 11, Week 12 and Week 13**
  - Graph Neural Network Paper Read
- **Week 14 and Week 15**
  - Catch up Time on the Material if needed
  - Final and Review

Course Logistics II

We will have

- **Attendance**: 10 percent (**Random** Pop Quiz)
- **Programming Assignment**: 30 percent (2 PA)
- **Midterm**: 30 percent
  - **Require** Lock-down Browser
  - Multiple Choices
  - Numerical Questions [Only 1 correct answer]
- **Final**: 30 percent (Dec 14th)
  - **Require** Lock-down Browser
  - Multiple Choices
  - Numerical Questions [Only 1 correct answer]

Note on Logistics

- A week-ahead notice for mid-term, based on the pace of the course.
- No extension on the project, which you should work on from the first week of class.
- The logistic is subject to change based on the overall pace and the performance of the class.

Grading Rubric

This course is **absolute** grading, meaning no curve, as there is a certain standard we need to uphold for students to have a good knowledge of deep learning.

Percentage	**Letter grade**	**Percentage**	**Letter grade**
95 – 100	A	70 – 74	C+
90 – 94	A–	65 – 69	C
85 – 89	B+	60 – 64	C–
80 – 84	B	55 – 59	D
75 – 79	B–	0 – 54	F

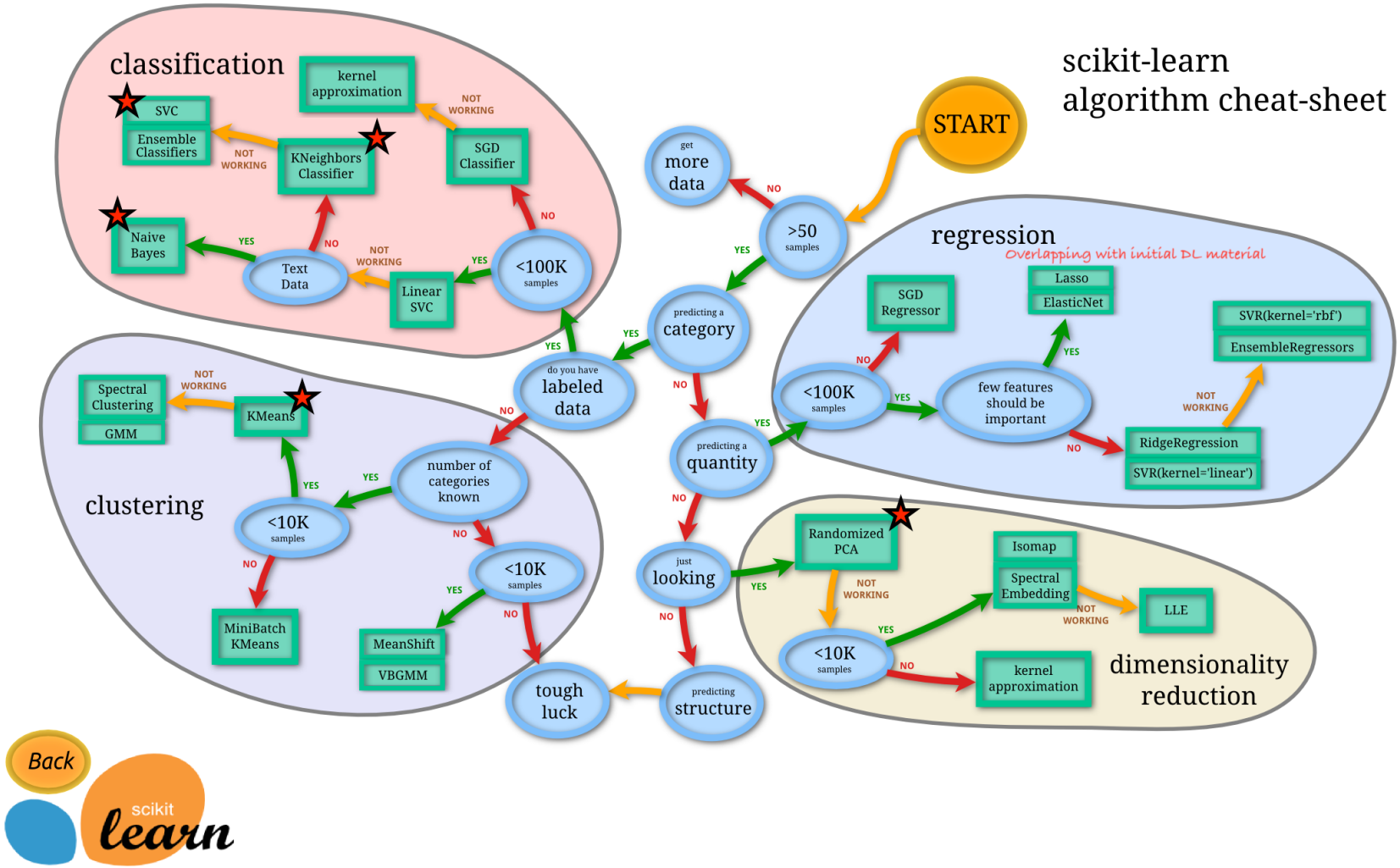
Academic Integrity

- (**Short**) Do not cheat! You will be caught and punished. Our department is serious about graduating ethical and upstanding computer scientists. The policy has recently been updated and will be enforced. For more information: Academic Integrity

Course Textbook (Optional)

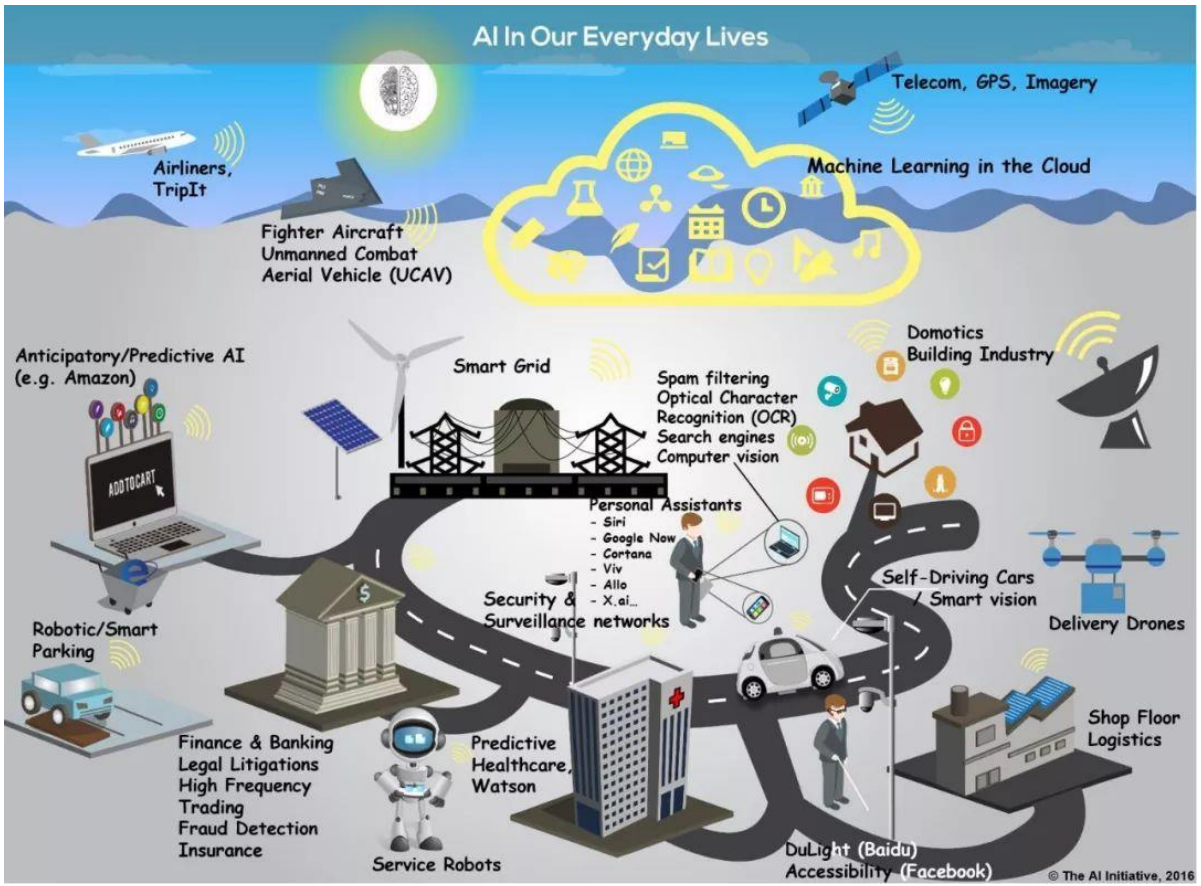
- ***Pattern Classification***, *David G. Stork, Peter E. Hart, and Richard O. Duda*
- ***Pattern Recognition and Machine Learning***, *Christopher Bishop*
- ***Attention Is All You Need*** *Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin*
- ***Deep Learning***, *Goodfellow Ian, Yoshua Bengio, and Aaron Courville*

Machine Learning Review (Next Class)



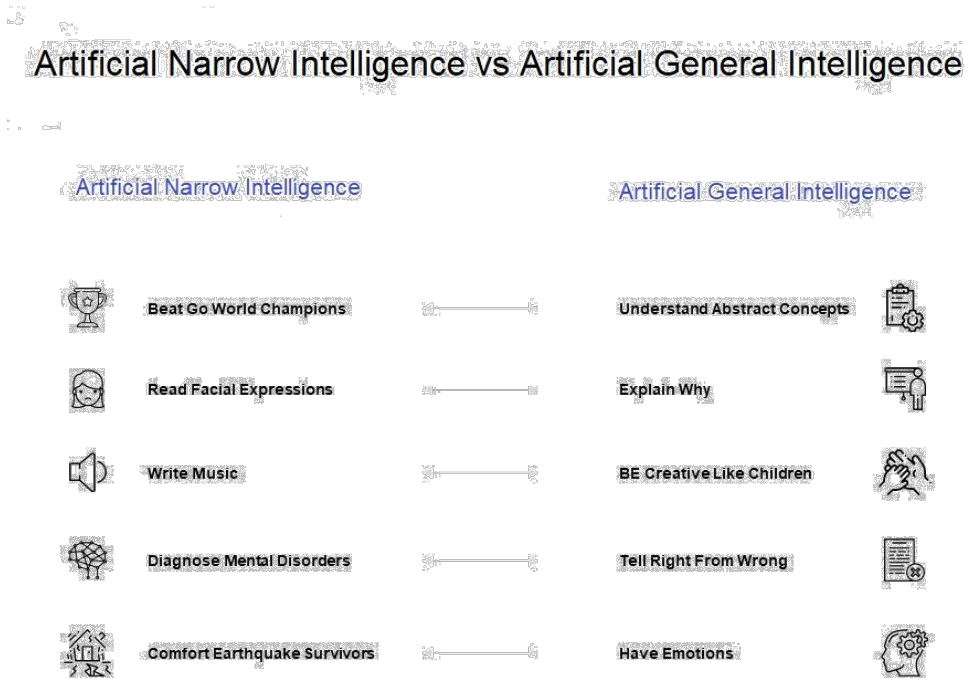
Today AI is Ubiquitous

You may find the applications of AI in every aspect of our lives these days.

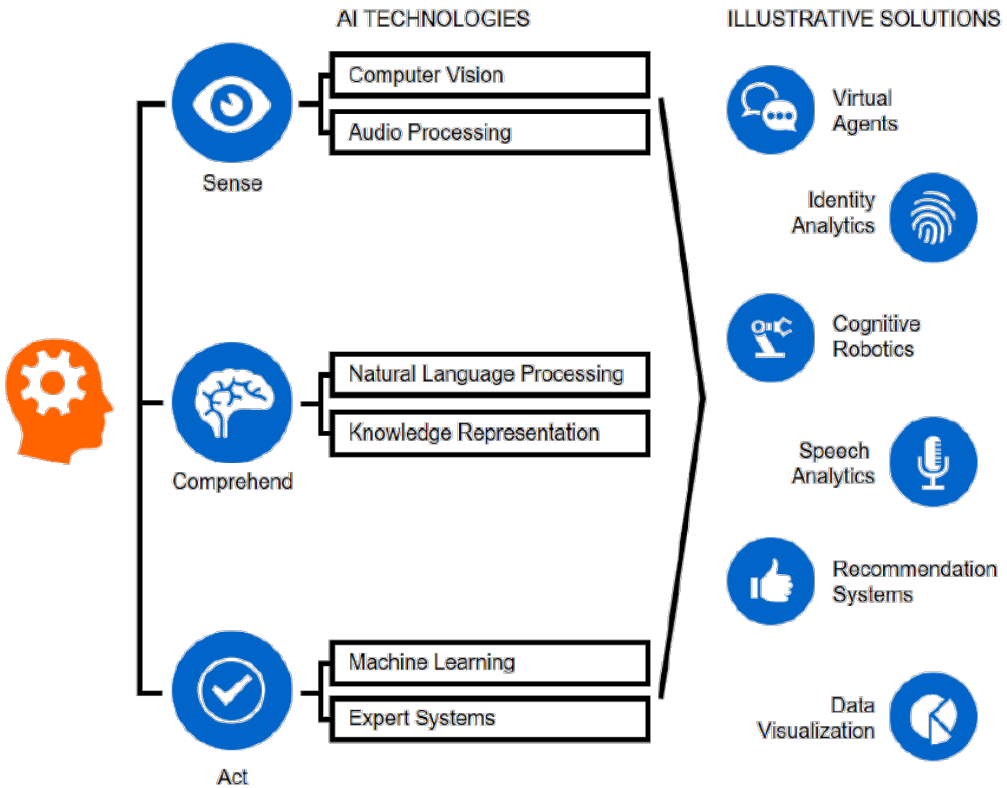


AI Paradox

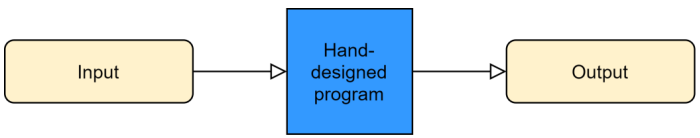
- Problems difficult for humans are easy for AI
- Problems easy for humans are difficult for AI



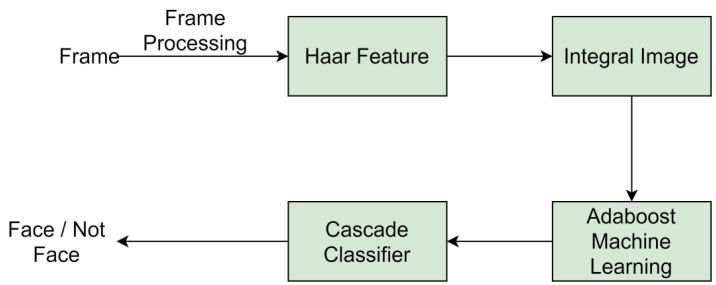
Which tasks require Intelligence?



Knowledge-Based AI



Rule Based System

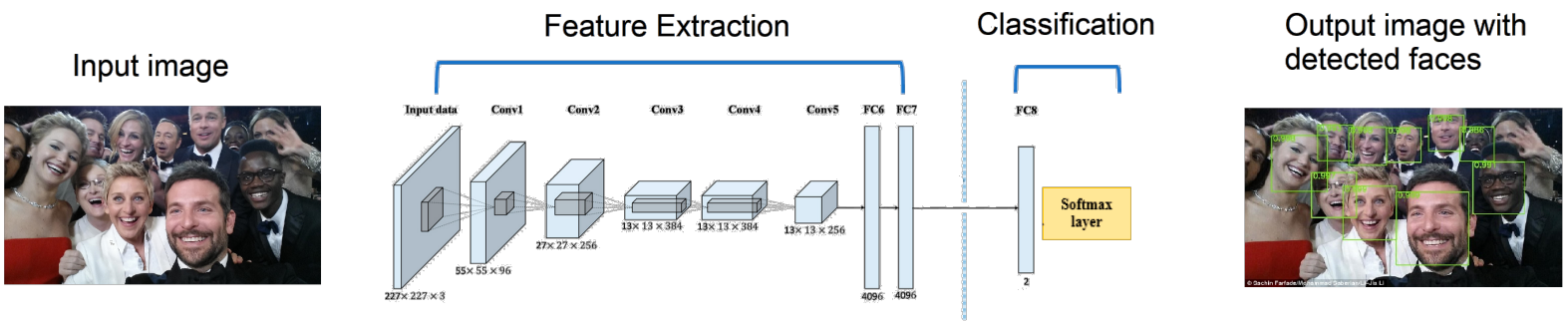


Viola Jones Face Detection Algorithm

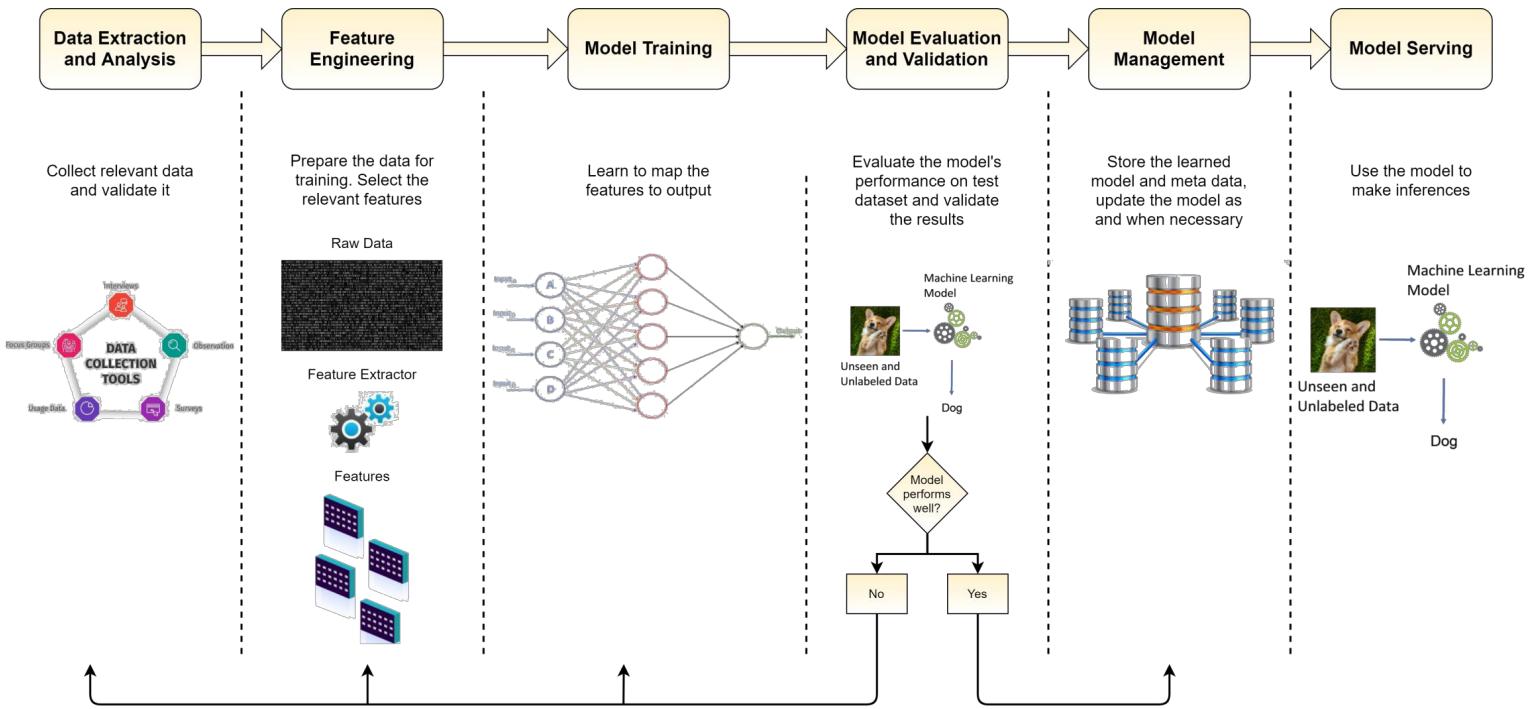
- Disadvantage: Unwieldy process
  - Time of human experts
  - People struggle to formalize rules with enough complexity to describe the world

The Machine Learning Approach

- Allow computers to learn from experience
- Determine what features to use
- Learn to map the features to output



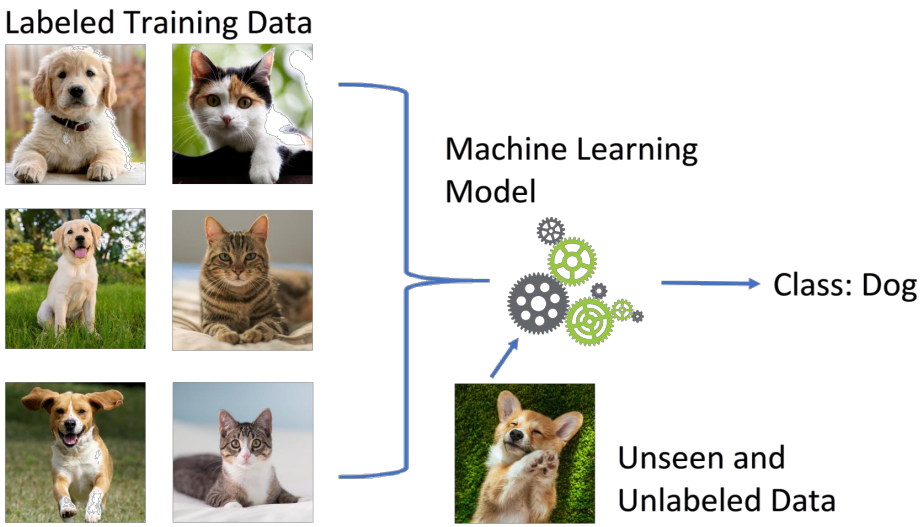
Machine Learning Pipeline



Supervised Learning vs Unsupervised Learning I

Supervised Learning

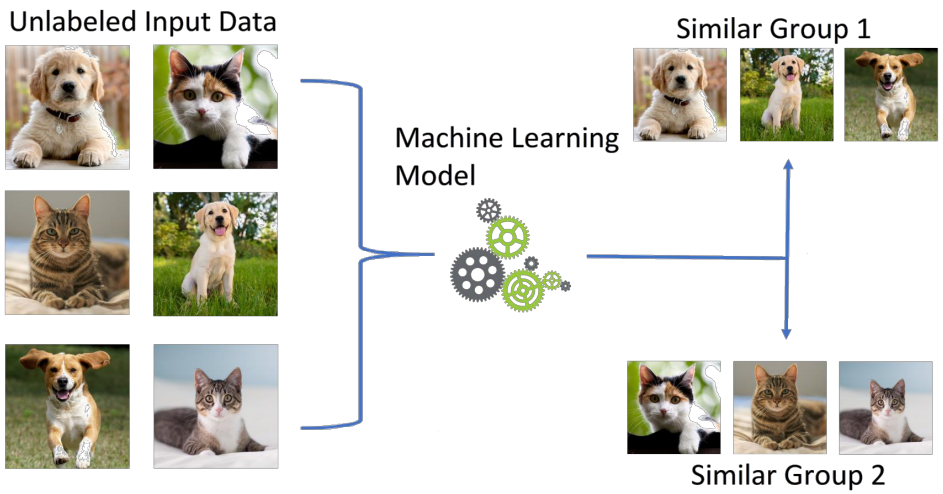
- Data-set: collection of **labeled examples** $(\{(\mathbf{x}_i, y_i)\}_{i=1}^{N})$
- Goal: produce a model that takes $\mathbf{x}$ as input and predict $\hat{y}$



Supervised Learning vs Unsupervised Learning II

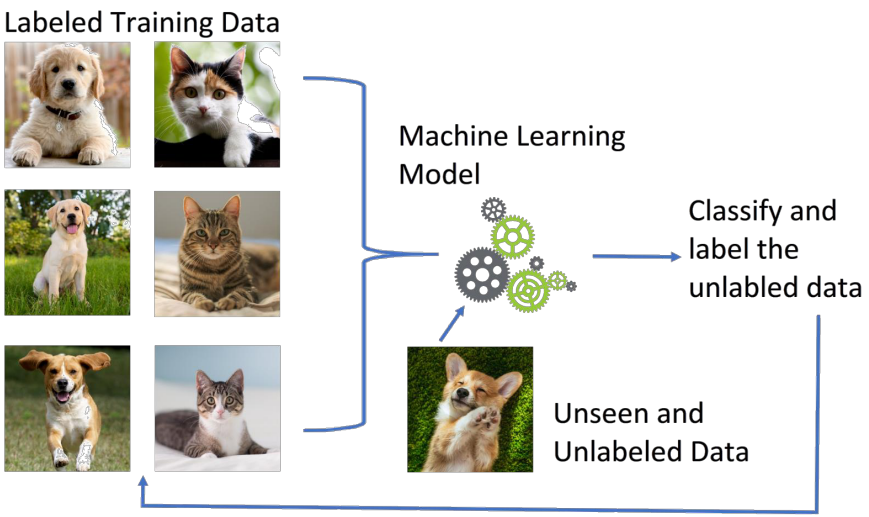
Unsupervised Learning

- Data-set: collection of **unlabeled example** $(\{\mathbf{x}_i\}_{i=1}^{N})$
- Goal: create a model that take $\mathbf{x}$ as input and either transform it into another vector or into a value that can be used to solve a practical problem.



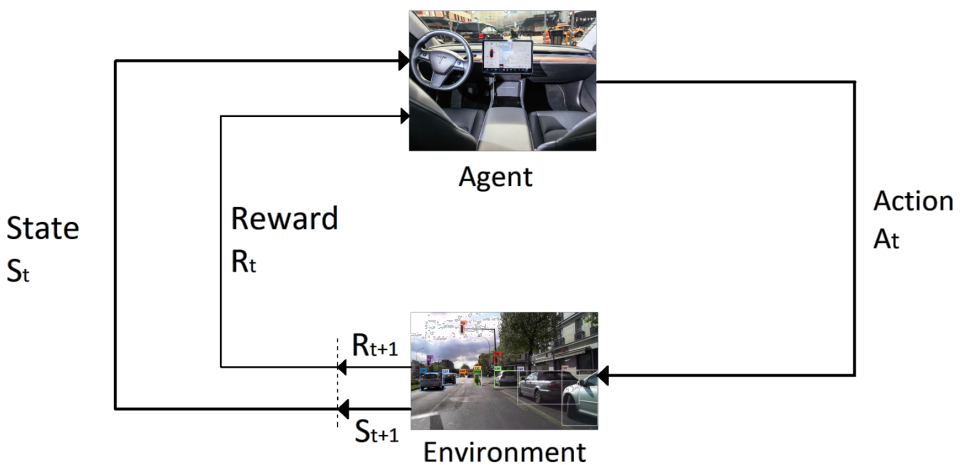
Semi-Supervised Learning

- Data-set: labeled and unlabeled examples.
- Goal: The hope here is that using many unlabeled examples can help the learning algorithm to find (we might say "produce" or "compute") a better model.



Reinforcement Learning

- **Input**: State-action pairs
- **Goal**: Learn a good sequence of decisions to maximize the reward.



Preliminaries

Some **survival skills** before the course:

- store and manipulate data
- libraries for ingesting and preprocessing data from a variety of sources
- basic knowledge in linear algebra (high-dimensional data elements)
- basic calculus to determine which direction to adjust each parameter in order to decrease the loss function (don't worry we have tools for it)
- some basic fluency in probability, our primary language for reasoning under uncertainty
- some aptitude for finding answers in the official documentation when you get stuck (most **important** in lots of scenario)

Data Manipulation: Introduction

Two important things we need to do with data:

1. Acquire them
2. Process them once they are inside the computer

Data Manipulation: Getting Started

- A tensor represents a (possibly multi-dimensional) array of numerical values.
  - With one axis, a tensor is called a *vector*.
  - With two axes, a tensor is called a *matrix*.
  - With $k > 2$ axes, we drop specialized names and just refer to the object as a k^{th} order tensor.

Data Manipulation: Vector I

First, you will need to import a deep learning package to expedite the process:

```
import torch
```

Then the fun part:

```
x = torch.arange(12, dtype=torch.float32)
x # return tensor([ 0.,  1.,  2.,  3.,  4.,  5.,  6.,  7., ↩
      8.,  9., 10., 11.])
```

Data Manipulation: Vector II

The tensor `x` contains 12 elements. We can inspect the total number of elements in a tensor via its `numel` method.

```
x.numel() # return 12
```

Also, we can also inspect `x`'s shape attribute. Because we are dealing with a vector here, the shape contains just a single element and is identical to the size.

```
x.shape # torch.Size([12])
```

Data Manipulation: Matrix I

We can change the shape of a tensor without altering its size or values, by invoking `reshape`.

- For example, we can transform our vector x whose shape is (12,) to a matrix X with shape (3,4).
- This new tensor retains all elements but re-configures them into a matrix.

```
X = x.reshape(3, 4)
X
```

```
tensor([[ 0.,  1.,  2.,  3.],
        [ 4.,  5.,  6.,  7.],
        [ 8.,  9., 10., 11.]])
```

Data Manipulation: Matrix II

- Note that specifying every shape component to reshape is redundant.
- We already know our tensor's size, we can work out one component of the shape given the rest.
  - For example, given a tensor of size n and target shape (h, w), we know that $w = n/h$.
  - To automatically infer one component of the shape, we can place a -1 for the shape component that should be inferred automatically.
  - In our case, instead of calling `x.reshape(3,4)`, we could equivalently called `x.reshape(-1,4)` or `x.reshape(3,-1)`

Data Manipulation: Additional Useful Tools I

- Initialization: all zeros

```
1 torch.zeros((2, 3, 4))
```

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

        [[0., 0., 0., 0.],
         [0., 0., 0., 0.],
         [0., 0., 0., 0.]]])
```

Data Manipulation: Additional Useful Tools II

- Initialization: all ones

```
1 torch.ones((2, 3, 4))
```

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

        [[1., 1., 1., 1.],
         [1., 1., 1., 1.],
         [1., 1., 1., 1.]]])
```

Data Manipulation: Indexing and Slicing I

Let's use our matrix X as an example:

```
tensor([[ 0., 1., 2., 3.],
        [ 4., 5., 6., 7.],
        [ 8., 9., 10., 11.]])
```

Data Manipulation: Indexing and Slicing II

- As with Python lists, we can access tensor elements by indexing (starting with 0).
  - To access an element based on its position relative to the end of the list, we can use negative indexing.
  - We can access whole ranges of indices via slicing (e.g., X[start:stop]), where the returned value includes the first index (start) but not the last (stop).

```
1 X[-1], X[1:3]
```

```
(tensor([ 8., 9., 10., 11.]),
 tensor([[ 4., 5., 6., 7.],
         [ 8., 9., 10., 11.]]))
```

Data Manipulation: Indexing and Slicing III

- We can also write elements of a matrix by specifying indices.

```
X[1, 2] = 17
X
```

```
tensor([[ 0., 1., 2., 3.],
        [ 4., 5., 17., 7.],
        [ 8., 9., 10., 11.]])
```

- Similarly, we can `X[:2, :] = 12`

```
tensor([[12., 12., 12., 12.],
        [12., 12., 12., 12.],
        [ 8., 9., 10., 11.]])
```

Data Manipulation: Operations I

- Calculate the exponential e^x, `torch.exp(x)`
- Common mathematical operations,

```
x = torch.tensor([1.0, 2, 4, 8])
y = torch.tensor([2, 2, 2, 2])
x + y, x - y, x * y, x / y, x ** y
```

```
(tensor([ 3., 4., 6., 10.]),
 tensor([-1., 0., 2., 6.]),
 tensor([ 2., 4., 8., 16.]),
 tensor([0.5000, 1.0000, 2.0000, 4.0000]),
 tensor([ 1., 4., 16., 64.]))
```

Data Manipulation: Operation II

- Concatenate multiple tensors along specific axis

```
X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, 2, 1]])
torch.cat((X, Y), dim=0)
```

```
tensor([[ 0., 1., 2., 3.],
        [ 4., 5., 6., 7.],
        [ 8., 9., 10., 11.],
        [ 2., 1., 4., 3.],
        [ 1., 2., 3., 4.],
        [ 4., 3., 2., 1.]])
```

Data Manipulation: Operation II (continued)

- Concatenate multiple tensors along specific axis

```
X = torch.arange(12, dtype=torch.float32).reshape((3,4))
Y = torch.tensor([[2.0, 1, 4, 3], [1, 2, 3, 4], [4, 3, ↩
    2, 1]])
torch.cat((X, Y), dim=1)
```

```
tensor([[ 0., 1., 2., 3., 2., 1., 4., 3.],
        [ 4., 5., 6., 7., 1., 2., 3., 4.],
        [ 8., 9., 10., 11., 4., 3., 2., 1.]])
```

Linear Algebra

- After loading datasets into tensors and manipulate tensors with basic mathematical operations, we will need additional tools from linear algebra to build sophisticated models.
  - Scalar arithmetic
  - Matrix multiplication

Linear Algebra: Scalars and Vectors I

- Scalars

```
x = torch.tensor(3.0)
y = torch.tensor(2.0)

x + y, x * y, x / y, x**y
```

```
(tensor(5.), tensor(6.), tensor(1.5000), tensor(9.))
```

Linear Algebra: Scalars and Vectors II

- When **vectors** represent examples from real-world datasets, their values hold some real-world significance.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$$

  - If we were studying heart attack risk, each vector might represent a patient and its components might correspond to their most recent vital signs, cholesterol levels, minutes of exercise per day, etc.

- Vectors are implemented as 1^{st} -order tensors.

```
1 x = torch.arange(3)
2 x # return tensor([0, 1, 2])
```

Linear Algebra: Matrices I

- Just as scalars are 0^{th} -order tensors and vectors are 1^{st} -order tensors, matrices are 2^{nd} -order tensors.

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & a_{m2} & \cdots & a_{mn} \end{bmatrix}$$

  - We represent a matrix $\mathbf{A} \in \mathbb{R}^{m \times n}$ by a 2^{nd} -order tensor with shape (m, n)
  - a_{ij} is the value that belongs to $\mathbf{A}'$ s i^{th} row and j^{th}

Linear Algebra: Matrices II (Transpose)

- Take the transpose of a matrix.

$$\mathbf{A}^\top = \begin{bmatrix} a_{11} & a_{21} & \dots & a_{m1} \\ a_{12} & a_{22} & \dots & a_{m2} \\ \vdots & \vdots & \ddots & \vdots \\ a_{1n} & a_{2n} & \dots & a_{mn} \end{bmatrix}$$

```
A = torch.arange(6).reshape(3, 2)
A
```

```
tensor([[0, 1],
        [2, 3],
        [4, 5]])
```

Linear Algebra: Matrices II (Transpose Continued)

- Now the transpose

```
1 A.T
```

```
tensor([[0, 2, 4],
        [1, 3, 5]])
```

Linear Algebra: Tensors

- Tensors give us a generic way to describe extensions to n^{th} -order arrays.
- Tensors will become more important when we start working with images.
  - Each image arrives as a 3^{rd} -order tensor with axes corresponding to the height, width, and channel.
  - A collection of images is represented in code by a 4^{th} -order tensor, where distinct images are indexed along the first axis.
- High-order tensors are constructed analogously to vectors and matrices, by growing the number if shape components.

```
1 torch.arange(24).reshape(2, 3, 4)
```

Linear Algebra: Basic Properties of Tensor Arithmetic I

- Useful element-wise operation:

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
B = A.clone()  # Assign a copy of A to B by allocating new memory
A, A + B
```

```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 0., 2., 4.],
         [ 6., 8., 10.]]))
```

Linear Algebra: Basic Properties of Tensor Arithmetic II

- The elementwise product of two matrices is called their Hadamard product (denoted $\odot$).

$$\mathbf{A} \odot \mathbf{B} = \begin{bmatrix} a_{11} b_{11} & a_{12} b_{12} & \dots & a_{1n} b_{1n} \\ a_{21} b_{21} & a_{22} b_{22} & \dots & a_{2n} b_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} b_{m1} & a_{m2} b_{m2} & \dots & a_{mn} b_{mn} \end{bmatrix}$$

```
1 A * B
```

```
tensor([[ 0.,  1.,  4.],
        [ 9., 16., 25.]])
```

Linear Algebra: Basic Properties of Tensor Arithmetic III

- Adding or multiplying a scalar and a tensor produces a result with the same shape as the original tensor.
- Here, each element of the tensor is added to (or multiplied by) the scalar.

```
a = 2
X = torch.arange(24).reshape(2, 3, 4)
a + X, (a * X).shape
```

Linear Algebra: Basic Properties of Tensor Arithmetic III (Continued)

```
(tensor([[[ 2,  3,  4,  5],
         [ 6,  7,  8,  9],
         [10, 11, 12, 13]],

        [[14, 15, 16, 17],
         [18, 19, 20, 21],
         [22, 23, 24, 25]]]),
 torch.Size([2, 3, 4]))
```

Linear Algebra: Reduction I

- Often, we wish to calculate the sum of a tensor's elements.
- To express the sum of the elements in a vector $\mathbf{x}$ of length n, we write $\sum_{i=1}^{n} x_i$

```
x = torch.arange(3, dtype=torch.float32)
x, x.sum()
```

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

Linear Algebra: Reduction II

- To express sums over the elements of tensors of arbitrary shape, we simply sum over all of its axes.
- For example, the sum of the elements of an $m \times n$ matrix $\mathbf{A}$ could be written $\sum_{i=1}^{m} \sum_{j=1}^{n} a_{ij}$.

```
A.shape, A.sum()
```

```
(torch.Size([2, 3]), tensor(15.))
```

Linear Algebra: Reduction III

- Remember:

```
A = torch.arange(6, dtype=torch.float32).reshape(2, 3)
```

```
tensor([[0., 1., 2.],
        [3., 4., 5.]])
```

Linear Algebra: Reduction IV

- We can also specify the axes along which the tensor should be reduced.
  - To sum over all elements along the rows (axis 0), we specify `axis=0` in sum.
  - Since the input matrix reduces along axis 0 to generate the output vector, this axis is missing from the shape of the output.

```
1 A.shape, A.sum(axis=0), A.sum(axis=0).shape
```

```
(torch.Size([2, 3]), tensor([3., 5., 7.]),
    torch.Size([3]))
```

Linear Algebra: Reduction V

- Specifying `axis=1` will reduce the column dimension (axis 1) by summing up elements of all the columns.

```
A.shape, A.sum(axis=1), A.sum(axis=1).shape
```

```
(torch.Size([2, 3]), tensor([ 3., 12.]),
    torch.Size([2]))
```

Linear Algebra: Non-Reduction Sum I

- Sometimes it can be useful to keep the number of axes unchanged when invoking the function for calculating the sum or mean.

```
sum_A = A.sum(axis=1, keepdims=True)
A, sum_A, sum_A.shape
```

```
(tensor([[0., 1., 2.],
         [3., 4., 5.]]),
 tensor([[ 3.],
         [12.]]),
 torch.Size([2, 1]))
```

Linear Algebra: Non-Reduction Sum II

- For instance, since `sum_A` keeps its axes after summing each row, we can divide `A` by `sum_A` with broadcasting to create a matrix where each row sums up to 1.

```
1 A / sum_A
```

```
tensor([[0.0000, 0.3333, 0.6667],
        [0.2500, 0.3333, 0.4167]])
```

Linear Algebra: Dot Products

- Given two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^d$, their dot product $\mathbf{x}^\top \mathbf{y}$ (or $\langle \mathbf{x}, \mathbf{y} \rangle$) is a sum over the products of the elements at the same position:

$$\mathbf{x}^\top \mathbf{y} = \sum_{i=1}^{d} x_i y_i$$

```
y = torch.ones(3, dtype = torch.float32)
x, y, torch.dot(x, y)
```

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

Linear Algebra: Matrix-Vector Products I

- To start off, we visualize our matrix in terms of its row vectors

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix}$$

- The matrix-vector product $\mathbf{Ax}$ is simply a column vector of length m, whose i^{th} element is the dot product $\mathbf{a}_i^\top \mathbf{x}$:

$$\mathbf{Ax} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_m^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \mathbf{a}_1^\top \mathbf{x} \\ \mathbf{a}_2^\top \mathbf{x} \\ \vdots \\ \mathbf{a}_m^\top \mathbf{x} \end{bmatrix}$$

Linear Algebra: Matrix-Vector Products II

```
1 A.shape, x.shape, torch.mv(A, x), A@x
```

```
(torch.Size([2, 3]), torch.Size([3]), tensor([ 5., 14.]),
    tensor([ 5., 14.]))
```

Linear Algebra: Matrix-Matrix Multiplication I

- Say that we have two matrices $\mathbf{A} \in \mathbb{R}^{n \times k}$ and $\mathbf{B} \in \mathbb{R}^{k \times m}$:

$$\mathbf{A} = \begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1k} \\ a_{21} & a_{22} & \cdots & a_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nk} \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} b_{11} & b_{12} & \cdots & b_{1m} \\ b_{21} & b_{22} & \cdots & b_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ b_{k1} & b_{k2} & \cdots & b_{km} \end{bmatrix}$$

Linear Algebra: Matrix-Matrix Multiplication II

- Let $\mathbf{a}_i^\top \in \mathbb{R}^k$ denote the row vector representing the i^{th} row of the matrix $\mathbf{A}$ and let $\mathbf{b}_j \in \mathbb{R}^k$ denote the column vector from the j^{th} column of the matrix $\mathbf{B}$:

$$\mathbf{A} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix}, \quad \mathbf{B} = \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}$$

Linear Algebra: Matrix-Matrix Multiplication III

- To form the matrix product $\mathbf{C} \in \mathbb{R}^{n \times m}$, we simply compute each element c_{ij} as the dot product between the i^{th} row of $\mathbf{A}$ and the j^{th} column of $\mathbf{B}$, i.e., $\mathbf{a}_i^\top \mathbf{b}_j$:

$$\mathbf{C} = \mathbf{AB} = \begin{bmatrix} \mathbf{a}_1^\top \\ \mathbf{a}_2^\top \\ \vdots \\ \mathbf{a}_n^\top \end{bmatrix} \begin{bmatrix} \mathbf{b}_1 & \mathbf{b}_2 & \cdots & \mathbf{b}_m \end{bmatrix}$$

$$= \begin{bmatrix} \mathbf{a}_1^\top \mathbf{b}_1 & \mathbf{a}_1^\top \mathbf{b}_2 & \cdots & \mathbf{a}_1^\top \mathbf{b}_m \\ \mathbf{a}_2^\top \mathbf{b}_1 & \mathbf{a}_2^\top \mathbf{b}_2 & \cdots & \mathbf{a}_2^\top \mathbf{b}_m \\ \vdots & \vdots & \ddots & \vdots \\ \mathbf{a}_n^\top \mathbf{b}_1 & \mathbf{a}_n^\top \mathbf{b}_2 & \cdots & \mathbf{a}_n^\top \mathbf{b}_m \end{bmatrix}$$

Linear Algebra: Matrix Matrix Multiplication IV

- We can think of the matrix-matrix multiplication **AB** as performing m matrix-vector products or $m \times n$ dot products and stitching the results together to form an $n \times m$ matrix.
- In the following snippet, we perform matrix multiplication on `A` and `B`. Here, `A` is a matrix with 2 rows and 3 columns, and `B` is a matrix with 3 rows and 4 columns. After multiplication, we obtain a matrix with 2 rows and 4 columns.

```
B = torch.ones(3, 4)
torch.mm(A, B), A@B
```

- The term *matrix-matrix multiplication* is often simplified to *matrix multiplication*, and should not be confused with the Hadamard product.

Linear Algebra: Norms I

- Some of the most useful operators in linear algebra are norms. Informally, the norm of a vector tells us how *big* it is.
- For instance, the ℓ_2 norm measures the (Euclidean) length of a vector. Here, we are employing a notion of size that concerns the magnitude of a vector's components (not its dimensionality).

Linear Algebra: Norms II

A norm is a function $\|\cdot\|$ that maps a vector to a scalar and satisfies the following three properties:

1. Given any vector $\mathbf{x}$, if we scale (all elements of) the vector by a scalar $\alpha \in \mathbb{R}$, its norm scales accordingly:

$$\|\alpha\mathbf{x}\| = |\alpha|\|\mathbf{x}\|$$

2. For any vectors $\mathbf{x}$ and $\mathbf{y}$: norms satisfy the triangle inequality:

$$\|\mathbf{x} + \mathbf{y}\| \leq \|\mathbf{x}\| + \|\mathbf{y}\|$$

3. The norm of a vector is nonnegative and it only vanishes if the vector is zero:

$$\|\mathbf{x}\| > 0 \text{ for all } \mathbf{x} \neq 0$$

Linear Algebra: Norms III

- Many functions are valid norms and different norms encode different notions of size.
- The Euclidean norm that we all learned in elementary school geometry when calculating the hypotenuse of right triangle is the square root of the sum of squares of a vector's elements. Formally, this is called the ℓ_2 norm and expressed as

$$\|\mathbf{x}\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2}$$

```
u = torch.tensor([3.0, -4.0])
torch.norm(u) # return tensor(5.)
```

Linear Algebra: Norms IV

- The ℓ_1 norm is also popular and the associated metric is called the Manhattan distance. By definition, the ℓ_1 norm sums the absolute values of a vector's elements:

$$\|\mathbf{x}\|_1 = \sum_{i=1}^{n} |x_i|$$

- Compared to the ℓ_2 norm, it is less sensitive to outliers. To compute the ℓ_1 norm, we compose the absolute value with the sum operation.

```
1 torch.abs(u).sum() # return tensor(7.)
```

Linear Algebra: ℓ_1 and ℓ_2 norm I

- The ℓ_1 norm is less sensitive to outliers than the ℓ_2 norm because it sums up the absolute values of the vector elements, which reduces the impact of outliers on the norm value. On the other hand, the ℓ_2 norm squares the differences between the vector elements, so a single outlier can significantly affect the ℓ_2 norm value.
- Both the ℓ_2 and ℓ_1 norms are special cases of the more general ℓ_p norms:

$$\|\mathbf{x}\|_p = \left(\sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$

Calculus: why?

- Calculus will come in handy for *optimization problems* that we will face, where we repeatedly update our parameters in order to decrease the loss function.
- Optimization address how to fit our models to training data, and calculus is its key prerequisite.
- However, do not get distracted by this new concept, as our ultimate goal is to perform well on *previously* unseen data. That problem is called *generalization*, which we will introduce later.

Calculus: Derivatives and Differentiation I

- *Derivative* is the rate of change in a function with respect to changes in its arguments.
- Derivatives can tell us how rapidly a loss function would increase or decrease were we to *increase* or *decrease* each parameter by an infinitesimally small amount.

Calculus: Derivatives and Differentiation II

- Formally, for functions $f : \mathbb{R} \to \mathbb{R}$, that map from scalars to scalars, the derivative of f at a point x is defined as

$$f'(x) = \lim_{h \to 0} \frac{f(x+h) - f(x)}{h}$$

  - This term on the right hand side is called a *limit* and it tells us what happens to the value of an expression as a specified variable approaches a particular value.
  - This limit tells us what the ratio between a perturbation h and the change in the function value $f(x+h) - f(x)$ converges to as we shrink its size to zero.

Calculus: Derivatives and Differentiation III

- We can interpret the derivative $f'(x)$ as the instantaneous rate of change of $f(x)$ with respect to x.
- Let's develop some intuition with an example. Define $u = f(x) = 3x^2 - 4x$.

```
def f(x):
    return 3 * x ** 2 - 4 * x
```

Calculus: Derivatives and Differentiation III

- Setting $x = 1$, $\frac{f(x+h)-f(x)}{h}$ approaches 2 as h approaches 0. While this experiment lacks the rigor of a mathematical proof, we will soon see that indeed $f'(1) = 2$.

```
for h in 10.0**np.arange(-1, -6, -1):
    print(f'h={h:.5f}, numerical limit={(f(1+h)-f(1))/h:.5f}')
```

```
h=0.10000, numerical limit=2.30000
h=0.01000, numerical limit=2.03000
h=0.00100, numerical limit=2.00300
h=0.00010, numerical limit=2.00030
h=0.00001, numerical limit=2.00003
```

Calculus: Notation & Common Functions I

- There are several equivalent conventions for derivatives. Given $y = f(x)$, the following expressions are equivalent:

$$f'(x) = y' = \frac{dy}{dx} = \frac{df}{dx} = \frac{d}{dx} f(x) = Df(x) = D_x f(x)$$

  where the symbols $\frac{d}{dx}$ and D are differentiation operators.

- Derivatives of common functions:

$$\frac{d}{dx} C = 0 \qquad \text{for any constant } C$$

$$\frac{d}{dx} x^n = n x^{n-1} \qquad \text{for } n \neq 0$$

$$\frac{d}{dx} e^x = e^x$$

$$\frac{d}{dx} \ln x = x^{-1}$$

Calculus: Notation & Common Functions II

- Functions composed from differentiable functions are often themselves differentiable.
- The following rules come in handy for working with compositions of any differentiable functions f and g, and constant C.

$$\frac{d}{dx}[Cf(x)] = C\frac{d}{dx}f(x) \qquad \text{Constant multiple rule}$$

$$\frac{d}{dx}[f(x)+g(x)] = \frac{d}{dx}f(x) + \frac{d}{dx}g(x) \qquad \text{Sum rule}$$

$$\frac{d}{dx}[f(x)g(x)] = f(x)\frac{d}{dx}g(x) + g(x)\frac{d}{dx}f(x) \qquad \text{Product rule}$$

$$\frac{d}{dx}\frac{f(x)}{g(x)} = \frac{g(x)\frac{d}{dx}f(x) - f(x)\frac{d}{dx}g(x)}{g^2(x)} \qquad \text{Quotient rule}$$

Calculus: Partial Derivatives and Gradients I

- Thus far, we have been differentiating functions of just one variable. In deep learning, we also need to work with functions of *many* variables.
- Let $y = f(x_1, x_2, \ldots, x_n)$ be a function with n variables. The partial derivative of y with respect to its i^{th} parameter x_i is

$$\frac{\partial y}{\partial x_i} = \lim_{h \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + h, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_i, \ldots, x_n)}{h}$$

Calculus: Partial Derivatives and Gradients II

- To calculate $\frac{\partial y}{\partial x_i}$, we can treat $x_1, \ldots, x_{i-1}, x_{i+1}, \ldots, x_n$ as constants and calculate the derivative of y with respect to x_i. The following notation conventions for partial derivatives are all common and all mean the same thing:

$$\frac{\partial y}{\partial x_i} = \frac{\partial f}{\partial x_i} = \partial_{x_i} f = \partial_i f = f_{x_i} = f_i = D_i f = D_{x_i} f$$

Calculus: Partial Derivatives and Gradients III

- We can concatenate partial derivatives of a multivariate function with respect to all its variables to obtain a vector that is called the gradient of the function.
- Suppose that the input of function $f : \mathbb{R}^n \to \mathbb{R}$ is an n-dimensional vector $\mathbf{x} = [x_1, x_2, \ldots, x_n]$ and the output is a scalar. The gradient of the function f with respect to $\mathbf{x}$ is a vector of n partial derivatives:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = [\partial_{x_1} f(\mathbf{x}), \partial_{x_2} f(\mathbf{x}), \ldots \partial_{x_n} f(\mathbf{x})]^\top$$

  - When there is no ambiguity, $\nabla_{\mathbf{x}} f(\mathbf{x})$ is typically replaced by $\nabla f(\mathbf{x})$.

Calculus: Chain Rule I

- Later you will realize that the gradients of concern are often difficult to calculate because we are working with deeply nested functions(of functions (of functions...)).
- Fortunately, the *chain rule* takes care of this. Returning to functions of a single variable, suppose that $y = f(g(x))$ and that the underlying functions $y = f(u)$ and $u = g(x)$ are both differentiable. The chain rule states that

$$\frac{dy}{dx} = \frac{dy}{du}\frac{du}{dx}$$

- Turning back to multivariate functions, suppose that $y = f(\mathbf{u})$ has variables $u_1, u_2, \ldots, u_m$, where each $u_i = g_i(\mathbf{x})$ has variables $x_1, x_2, \ldots, x_n$, i.e., $\mathbf{u} = g(\mathbf{x})$. Then the chain rule states that

$$\frac{\partial y}{\partial x_i} = \frac{\partial y}{\partial u_1}\frac{\partial u_1}{\partial x_i} + \frac{\partial y}{\partial u_2}\frac{\partial u_2}{\partial x_i} + \ldots + \frac{\partial y}{\partial u_m}\frac{\partial u_m}{\partial x_i} \text{ and thus } \nabla_{\mathbf{x}} y = \mathbf{A}\nabla_{\mathbf{u}} y$$

Automatic Differentiation: Fear Not of Calculus

- Calculating derivatives is the crucial step in all of the optimization algorithms that we will use to train deep networks. While the calculations are straightforward, working them out by hand can be tedious and error-prone, and this problem only grows as our models become more complex.
- Fear not if you can't remember every formula in calculus, modern deep learning framework like PyTorch and TensorFlow got you covered.

Automatic Differentiation: A Simple Function I

- Let's assume that we are interested in differentiating the function $y = 2\mathbf{x}^\top \mathbf{x}$ with respect to the column vector $\mathbf{x}$. To start, we assign x an initial value.

```
x = torch.arange(4.0)
x # return tensor([0., 1., 2., 3.])
```

Automatic Differentiation: A Simple Function I

- Before we calculate the gradient of y with respect to $\mathbf{x}$, we need a place to store it.
- In general, we avoid allocating new memory every time we take a derivative because every time we take a derivative because in most case we require successively computing derivatives with respect to the same parameters thousands or millions of times, and we might risk running out of memory.

```
# Same thing: x = torch.arange(4.0, requires_grad=True)
x.requires_grad_(True)
x.grad  # The gradient is None by default
```

Automatic Differentiation: A Simple Function II

- We now calculate our function of `x` and assign the result to `y`.

```
y = 2 * torch.dot(x, x)
y # return tensor(28., grad_fn =<MulBackward0>)
```

- We can now take the gradient of `y` with respect to `x` by calling its `backward` method.

```
y.backward()
```

- Next, we can access the gradient via `x`'s `grad` attribute.

```
x.grad # return tensor([ 0.,  4.,  8., 12.])
```

Probability and Statistics: An Introduction

- It is all about uncertainty.
  - In **supervised learning**, we want to quantify our uncertainty; How likely a patient to suffer a heart attack in the next year.
  - In **unsupervised learning**, to determine whether a set of measurements are anomalous, it helps to know how likely one is to observe values in a population of interest.
  - In **reinforcement learning**, we wish to develop agents that act intelligently in various environments. This requires reasoning about how an environment might be expected to change and what rewards one might expect to encounter in response to each of the available actions.

A Simple Example: Tossing Coins I

- Imagine that we plan to toss a coin and want to quantify how likely we are to see heads (vs. tails).
  - If the coin is *fair*, then both outcomes (heads and tails), are equally likely
  - Formally, the quantity $\frac{1}{2}$ is called a *probability*. Probabilities assign scores between 0 and 1 to outcomes of interest, called *events* (P (heads))

A Simple Example: Tossing Coins II

- Now suppose that the coin was in fact fair,i.e., $P(\text{heads}) = 0.5$

```
num_tosses = 100
heads = sum([random.random() > 0.5 for _ in range(100)])
tails = num_tosses - heads
print("heads, tails: ", [heads, tails])
```

```
heads, tails: [48, 52]
```

A Simple Example: Tossing Coins II

- Equivalently, we can also simulate multiple draws from any variable with a finite number of possible outcomes by calling the multinomial function.

```
fair_probs = torch.tensor([0.5, 0.5])
Multinomial(100, fair_probs).sample()
```

```
tensor([44., 56.])
```

A Simple Example: Tossing Coins III

- Each time you run this sampling process, you will receive a new random value that may differ from the previous outcome.
- Dividing by the number of tosses gives us the *frequency* of each outcome in our data. Note that these frequencies, like the probabilities that they are intended to estimate, sum to 1.

```
1 Multinomial(100, fair_probs).sample() / 100
```

```
tensor([0.5300, 0.4700])
```

A Simple Example: Tossing Coins IV

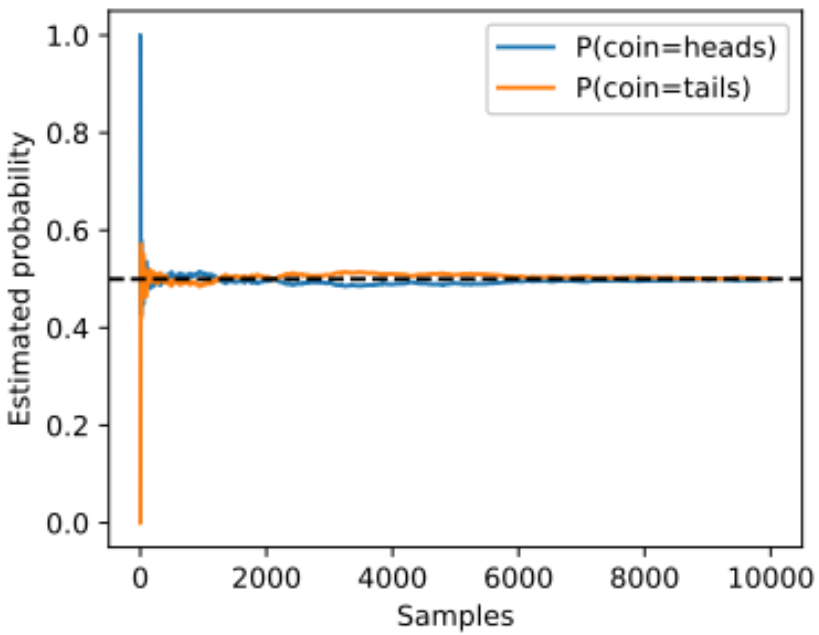
- Why the counts of head and tails may not be identical?

```
counts = Multinomial(10000, fair_probs).sample()
counts / 10000
```

```
tensor([0.4970, 0.5030])
```

A Simple Example: Tossing Coins V

- Let's see what happens as we grow the number of tosses from 1 to 10000



Formalization I

- In STEM, it is important to formalize as mathematics is a common language, learning to speak in math and code is especially important in computer science.
- When dealing with randomness, we denote the set of possible outcomes $\mathcal{S}$ and call it the *sample space* or *outcome space*.
  - Here, each element is a distinct possible outcome. In the case of rolling a single coin, $\mathcal{S} = \{$ heads, tails $\}$. For a single die, $\mathcal{S} = \{1, 2, 3, 4, 5, 6\}$. When flipping two coins, possible outcomes are $\{($ heads, heads), (heads, tails), (tails, heads), (tails, tails $)\}$.
- *Events* are subsets of the sample space. For instance, the event "the first coin toss comes up heads" corresponds to the set $\{($ heads, heads), (heads, tails $)\}$. Whenever the outcome z of a random experiment satisfies $z \in \mathcal{A}$, then event $\mathcal{A}$ has occurred. For a single roll of a die, we could define the events "seeing a 5 " ($\mathcal{A} = \{5\}$) and "seeing an odd number" ($\mathcal{B} = \{1, 3, 5\}$). In this case, if the die came up 5, we would say that both $\mathcal{A}$ and $\mathcal{B}$ occurred. On the other hand, if $z = 3$, then $\mathcal{A}$ did not occur but $\mathcal{B}$ did.

Formalization II

A *probability* function maps events onto real values $P : \mathcal{A} \subseteq \mathcal{S} \rightarrow [0, 1]$. The probability of an event $\mathcal{A}$ in the given sample space $\mathcal{S}$, denoted $P(\mathcal{A})$, satisfies the following properties:

- The probability of any event $\mathcal{A}$ is a non-negative real number, i.e., $P(\mathcal{A}) \geq 0$;
- The probability of the entire sample space is 1 , i.e., $P(\mathcal{S}) = 1$;
- For any countable sequence of events $\mathcal{A}_1, \mathcal{A}_2, \ldots$ that are *mutually exclusive* ($\mathcal{A}_i \cap \mathcal{A}_j = \emptyset$ for all $i \neq j$), the probability that any of them happens is equal to the sum of their individual probabilities, i.e., $P\left(\bigcup_{i=1}^{\infty} \mathcal{A}_i\right) = \sum_{i=1}^{\infty} P\left(\mathcal{A}_i\right)$

Random Variables I

- When we spoke about events like the roll of a die coming up odds and the first coin toss coming up heads, we were invoking the idea of a *random variable*.
- Formally, random variables are mapping from an underlying sample space to a set of (possibly many) values.
- How is random variable different from the sample space?
  - A random variable is a set of possible values from a random experiment.
  - A sample space is the set of all possible outcomes of a particular experiment.

Random Variables II

Still confused by the formal definition?

- We can define a binary random variable like "greater than 0.5" even when the underlying sample space is infinite, e.g., the line segment between 0 and 1.
- Additionally, multiple random variables can share the same underlying sample space.
  - "*whether my home alarm goes off*" and "whether my house was burglarized" are both binary random variables that share an underlying sample space.
  - Consequently, knowing the value taken by one random variable can tell us something about the likely value of another random variable. Knowing that the alarm went off, we might suspect that the house was likely burglarized.

Random Variables: A Formal Definition

- Every value taken by random variable corresponds to subset of the underlying sample space. Thus the occurrence where the random variable X takes value v, denoted by $X = v$, is an *event* and $P(X = v)$ denotes its probability.
- *Discrete random variables*: flips of a coin or tosses of a die.
- *Continuous random variables*: weight and height of a person sampled at random from the population.

Multiple Random Variables: The Real World

- Multiple factors affect the outcome of the event:
  - If a patient walks into a hospital and we observe that they are having trouble breathing and have lost their sense of smell, then we believe that they are more likely to have COVID-19 than we might if they had no trouble breathing and a perfectly ordinary sense of smell.

Multiple Random Variables: Equations

- *Joint Probability* assigned to the event where random variables A and B take values a and b, respectively, is denoted $P(A = a, B = b)$.
  - For any values a and b, it holds that $P(A = a, B = b) \leq P(A = a)$ and $P(A = a, B = b) \leq P(B = b)$
- The famous equation:

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)}$$

"posterior equals prior times likelihood, divided by the evidence" usually,

$$P(A \mid B) \propto P(B \mid A)P(A)$$

Linear Neural Network for Regression: An Introduction

- Before we explore deep neural networks, it will be helpful to implement some shallow neural network
- Help us focus on understanding important concepts:
  - Basics of neural network training, including **parameterizing the output layer**, **handling data**, **specify a loss function**, and **training the model**.
  - Shallow networks happens to comprise the set of linear models, which subsumes many classical methods for statistical prediction, including linear and softmax regression.
- Understanding these classical tools is pivotal because they are widely used in many contexts and we will often need to use them as baselines when justifying the use of fancier architectures.

Linear Neural Network for Regression: Linear Regression I

- *Regression* problems pop up whenever we want to predict a numerical value.
  - Common examples include predicting prices (of homes, stocks, etc.), predicting the length of stay (for patients in the hospital), forecasting demand (for retail sales), and so on.
  - Keep in mind that not every prediction problem is a classic regression problem. Later on, we will introduce classification problems , where the goal is to predict membership among a set of categories.

Linear Neural Network for Regression: Linear Regression II

- As a running example, suppose we wish to estimate the prices of houses (in dollars) based on their area (in square feet) and age (in years).
  - Data consisting of sales (sale price, area, and age for each home)
  - Machine learning terminology: the dataset is called a *training dataset* or *training set*, and each row (containing the data corresponding to one sale) is called an example (or data point, instance, sample). The thing we are trying to predict (price) is called a *label* (or *target*). The variables (age and area) upon which the predictions are based are called *features* (or *covariates*).

Linear Neural Network for Regression: Model I

- At the heart of every solution is a model that describes how features can be transformed into an estimate of the target.
- The assumption of linearity means that the expected value of the target (price) can be expressed as a weighted sum of the features (area and age):

$$\text{price} = w_{\text{area}} \cdot \text{area} + w_{\text{age}} \cdot \text{age} + b$$

Here w_{area} and w_{age} are called weights, and b is called a bias (or offset or intercept).

Linear Neural Network for Regression: Model II

- In machine learning, we usually work with high-dimensional datasets, where it is more convenient to employ compact linear algebra notation.
- When our inputs consist of d features, we can assign each an index (between 1 and d) and express our prediction $\hat{y}$ (in general the "hat" symbol denotes an estimate) as

$$\hat{y} = w_1 x_1 + \ldots + w_d x_d + b$$

Collecting all features into a vector $\mathbf{x} \in \mathbb{R}^d$ and all weights into a vector $\mathbf{w} \in \mathbb{R}^d$, we can express our model compactly via the dot product between $\mathbf{w}$ and $\mathbf{x}$:

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

Linear Neural Network for Regression: Model III

- Before, the vector $\mathbf{x}$ corresponds to the features of a single example. We will often find it convenient to refer to features of our entire dataset of n examples via the *design matrix* $\mathbf{X} \in \mathbb{R}^{n \times d}$.
- Here, $\mathbf{X}$ contains one row for every example and one column for every feature. For a collection of features $\mathbf{X}$, the predictions $\hat{\mathbf{y}} \in \mathbb{R}^n$ can be expressed via the matrix-vector product:

$$\hat{\mathbf{y}} = \mathbf{X}\mathbf{w} + b$$

- Given features of a training dataset $\mathbf{X}$ and corresponding (known) labels $\mathbf{y}$, the goal of linear regression is to find the weight vector $\mathbf{w}$ and the bias term b that given features of a new data example sampled from the same distribution as $\mathbf{X}$, the new example's label will (in expectation) be predicted with the lowest error.

Linear Neural Network for Regression: Model IV

- Even if we believe that the best model for predicting y given $\mathbf{x}$ is linear, we would not expect to find a real-world dataset of n examples where $y^{(i)}$ exactly equals $\mathbf{w}^\top \mathbf{x}^{(i)} + b$ for all $1 \leq i \leq n$.
  - For example, whatever instruments we use to observe the features $\mathbf{X}$ and labels $\mathbf{y}$ might suffer small amount of measurement error.
  - Thus, even when we are confident that the underlying relationship is linear, we will incorporate a noise term to account for such errors.
- Before we can go about searching for the *best parameters* (or *model parameters*) $\mathbf{w}$ and b, we will need two more things: (I) **a quality measure** for some given model; and (II) a procedure for for **updating the model** to improve its quality.

Linear Neural Network for Regression: Loss Function I

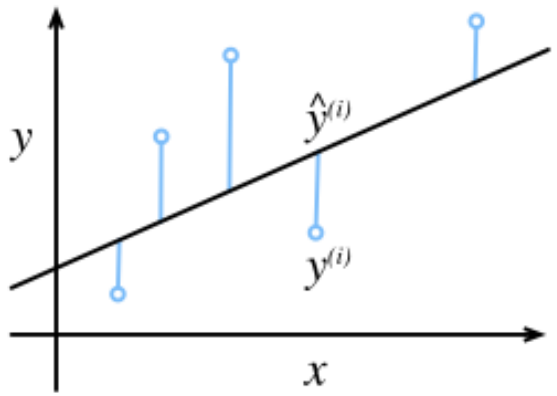
- Naturally fitting our model to the data requires that we agree on some measure of *fitness* (or, equivalently, of *unfitness*).
- *Loss functions* quantify the distance between the *real* and *predicted* values of the target.
  - The loss will usually be a non-negative number where smaller values are better and perfect prediction incur a loss of 0.
- For regression problems, the most common loss function is squared error.

$$l^{(i)}(\mathbf{w}, b) = \frac{1}{2}\left(\hat{y}^{(i)} - y^{(i)}\right)^2$$

  - The constant $\frac{1}{2}$ makes no real difference but proves to be notationally convenient, since it cancels out when we take the derivative of the loss.

Linear Neural Network for Regression: Loss Function II

- Let's visualize the fit of a linear regression model in a problem with one-dimensional inputs:



- To measure the quality of a model on the entire dataset of n examples, we simply average(or equivalently, sum) the losses on the training set:

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2$$

Linear Neural Network for Regression: Loss Function and Analytical Solution

- The loss function:

$$L(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} l^{(i)}(\mathbf{w}, b) = \frac{1}{n}\sum_{i=1}^{n} \frac{1}{2}\left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)}\right)^2$$

- When training the model, we want to find parameters $(\mathbf{w}^*, b^*)$ that minimize the total loss across all training examples:

$$\mathbf{w}^*, b^* = \underset{\mathbf{w}, b}{\operatorname{argmin}} L(\mathbf{w}, b)$$

- The analytical solution:

$$\mathbf{w}^* = \left(\mathbf{X}^\top \mathbf{X}\right)^{-1} \mathbf{X}^\top \mathbf{y}$$

Linear Neural Network for Regression: Implementation

- Define the model:

```
def forward(self, X):
    return torch.matmul(X, self.w) + self.b
```

- Define the loss function:

```
def loss(self, y_hat, y):
    l = (y_hat - y) ** 2 / 2
    return l.mean()
```

Hold on!!! Optimization

- Most cases, we can not solve models analytically, we can still often models effectively in practice.
- The key technique for optimizing nearly any model, consists of iteratively reducing the error by updating the parameters in the direction that incrementally lowers the loss function. This algorithm is called *gradient descent*.

Optimization

- Before we continue our journey, let's add in additional tools in our tool box.
- The performance of the optimization algorithm directly affects the model's training efficiency.
- Understanding these optimization algorithms will help us better tune the hyperparameters, improve the performance of our models.

Optimization: Mini-batch Stochastic Gradient Descent I

- The most naive application of gradient descent (***Gradient Descent***) consists of taking the derivative of the loss function, which is an average of the losses computed on every single example in the dataset.
  - Extremely slow: we must pass over the **entire dataset** before making a single update.
  - Even worse, there can be a lot of irrelevant or redundant in the training data. So the benefit of a full update is even lower.
- (***Stochastic Gradient Descent***)There is the other extreme of consider a single example at a time and to take update steps based on one observation at a time.
  - This also have several drawbacks, no worries, we will discuss later when we understand more concepts.

Optimization: Mini-batch Stochastic Gradient Descent II

- (***Mini-batch Stochastic Gradient Descent***) An intermediate strategy: rather than taking a full batch or only a single sample at a time, we take a *minibatch* of observations.
  - usually between 32 and 256, preferably a multiple of large power of 2
- Formalize the equation,

$$(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{(\mathbf{w},b)} l^{(i)}(\mathbf{w}, b)$$

  - $\mathcal{B}_t$ consisting of a fixed number of $|\mathcal{B}|$ of training examples.
  - compute the derivative (gradient) of the average loss on the minibatch with respect to the model parameters.
  - multiply the gradient by a predetermined small positive value η, called the learning rate, and substract the resulting term from the current parameter values.

Optimization: Mini-batch Stochastic Gradient Descent III

In summary, minibatch SGD proceeds as follows:

1. initialize the values of the model parameters typically at random
2. iteratively sample random minibatches from the data, updating the parameters in the direction of the negative gradient.
3. in the case of quadratic losses:

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_{\mathbf{w}} l^{(i)}(\mathbf{w}, b) = \mathbf{w} - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \mathbf{x}^{(i)} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right)$$

$$b \leftarrow b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \partial_b l^{(i)}(\mathbf{w}, b) = b - \frac{\eta}{|\mathcal{B}|} \sum_{i \in \mathcal{B}_t} \left(\mathbf{w}^\top \mathbf{x}^{(i)} + b - y^{(i)} \right).$$

Mini-batch Stochastic Gradient Descent: Implementation

- At each step, using a minibatch randomly drawn from our dataset, we estimate the gradient of loss with respect to the parameters.
- Next, we update the parameters in the direction that may reduce the loss.

```
class SGD():
    def __init__():
        ...
    def step(self):
        for param in self.params:
            param -= self.lr * param.grad

    def zero_grad(self): # zero-out, you do not want gradient to accumulate
        for param in self.params:
            if param.grad is not None:
                param.grad.zero_()
```

Training

A concise and almost overly repetitive mention of how you should train:

- Initialize parameters $(\mathbf{w}, b)$
- Repeat until done
  - Compute gradient $\mathbf{g} \leftarrow \partial_{(\mathbf{w},b)} \frac{1}{|\mathcal{B}|} \sum_{i \in \mathcal{B}} l\left(\mathbf{x}^{(i)}, y^{(i)}, \mathbf{w}, b\right)$
  - Update parameters $(\mathbf{w}, b) \leftarrow (\mathbf{w}, b) - \eta \mathbf{g}$

A Concise Implementation in PyTorch I

- Define the Model

```
class LinearRegression():  #@save
    """The linear regression model implemented with high-level APIs."""
    def __init__(self, lr):
        super().__init__()
        self.net = nn.LazyLinear(1)
        self.net.weight.data.normal_(0, 0.01)
        self.net.bias.data.fill_(0)

    def forward(self, X):
        return self.net(X)
```

A Concise Implementation in PyTorch II

- Define the Optimization Algorithm:

```
def configure_optimizers(self):
    return torch.optim.SGD(self.parameters(), self.lr)
```

- For more detail and information:
  - Pytorch Official: Learn the Basic