Neural-Networks-4

Few more homework for Numpy and then we will move to pytorch in future notes.

Numpy Exercises:
Question-1 and Solutions
Question-2 and Solutions
Questions in DeepML and Solutions with intuition detailed

Given we have completed few well known optimization algorithms.
But the question is "What we are optimizing ?"

also we are going to Understand the selection of loss function from first principle.

so keeping in mind:

> You don't "choose" a loss.
> You **derive it from what error actually means in your problem**.

and

There is a one liner to keep in mind

> Loss functions are **assumptions about your data**, not training tricks.

If your loss doesn't match:

- data noise
- evaluation metric
- optimization dynamics
  Then we are optimizing wrong thing.

So keeping this easy !!

Let's start again

We have the
data: $D = \{x_i, y_i\}$

Model: our approximation of relation between x and y

$$\hat{y} = \hat{f}(x)$$

where as the $\hat{f}(x)$ is the approximate of true label y. and $f(x)$ is not known to us.

Learning Algorithm: optimizer algorithms

Objective/ Loss/ Error function: One such possibility is MSE loss i.e.
$\mathcal{L}(w) = \sum_{i=1}^{n}(\hat{y}_i - y_i)^2$

The Learning algorithm (which we called Optimizers) should aim to find a θ (parameter) which minimizes the above function.

confusing right !!

To evaluate how well a neural network predicts a target value, we introduce a **loss function**, which quantifies the discrepancy between predicted and actual outputs.

During training, an optimizer minimizes this loss over the training data, improving prediction quality with respect to the chosen loss function. In probabilistic settings, this corresponds to increasing the likelihood of the true target under the model.

Since different loss functions encode different notions of error and assumptions about the data, selecting an appropriate loss function is a critical step in designing a neural network.

before diving into the different loss functions definition. I want to make myself feel the importance of loss functions.

So given there are many loss functions out there I will pretend there are only few to learn properly. and will give myself the intuitions which will help me learn all loss functions out there in the same manner with time in the note series.

These notes are understanding from this lecture slide of prof. simon prince
Source

We divide different machine learning algorithms in the following ways:
Problems into different categories

1. Univariate regression problem
2. Multi-variate regression problem
3. Binary classification problem
4. Multiclass classification problem

Loss Function

Training dataset of I pairs of inputs - outputs example : $\{x_i, y_i\}_{i=1}^{I}$

where loss functions measure how bad our model is with

$$L[\phi, f[x, \phi], \{x_i, y_i\}_{i=1}^{I}]$$

where
ϕ**:** parameters of model
$f[x, \phi]$: our model (which takes train data as input w.r.t to given parameter ϕ and predict an output)

we will write in short:

$$L[\phi]$$

which return a scalar that is smaller when model map inputs to better output.
so we have to find the parameter that will minimize the loss:

$$\hat{\phi} = \underset{\phi}{argmin}[L[\phi]]$$

How to construct a loss function ?

Model predict an output y given the input x (ha ha) instead

1. Model predict a conditional probability distribution $P(y|x)$ over output y given input x.
2. Loss function aims to make the output have higher probability

Let's interpret prof.'s these lines

Earlier we knew
"Model predict an output and we make the output close to actual output by a loss function"

But listen to me very carefully

Our interpretation should be :
we should pretend that,
Given the input x, the model is not saying `y` is the answer. instead it says **How likely every possible possible y is, given what the model knows**.

Examples:

- Classification:
  "Class A: 0.1, Class B: 0.8, Class C: 0.1"
- Regression (even if hidden):
  "Most likely value is 7.3, with uncertainty around it"

as in reality a model (i.e. our neural network) only will output numbers. but our interpretation should be like "it predict a number with uncertainty (i.e. a probability)"

So, from now we will keep in mind, the first point

> Model predict a conditional probability distribution $P(y|x)$ over output y given input x.

from the prediction, the output a number or class is a design choice or architectural choice. which made by us.

coming to the second point.

My interpretation is :
The loss function asks ourselves a brutal question each time

> "Did the model assign high probability to the **true answer** ?"

If the true y gets:

- **high probability** → small loss → model rewarded
- **low probability** → large loss → model punished

So the training a model becomes more of

> "Increase confidence where you were right. correct yourself where you were wrong."

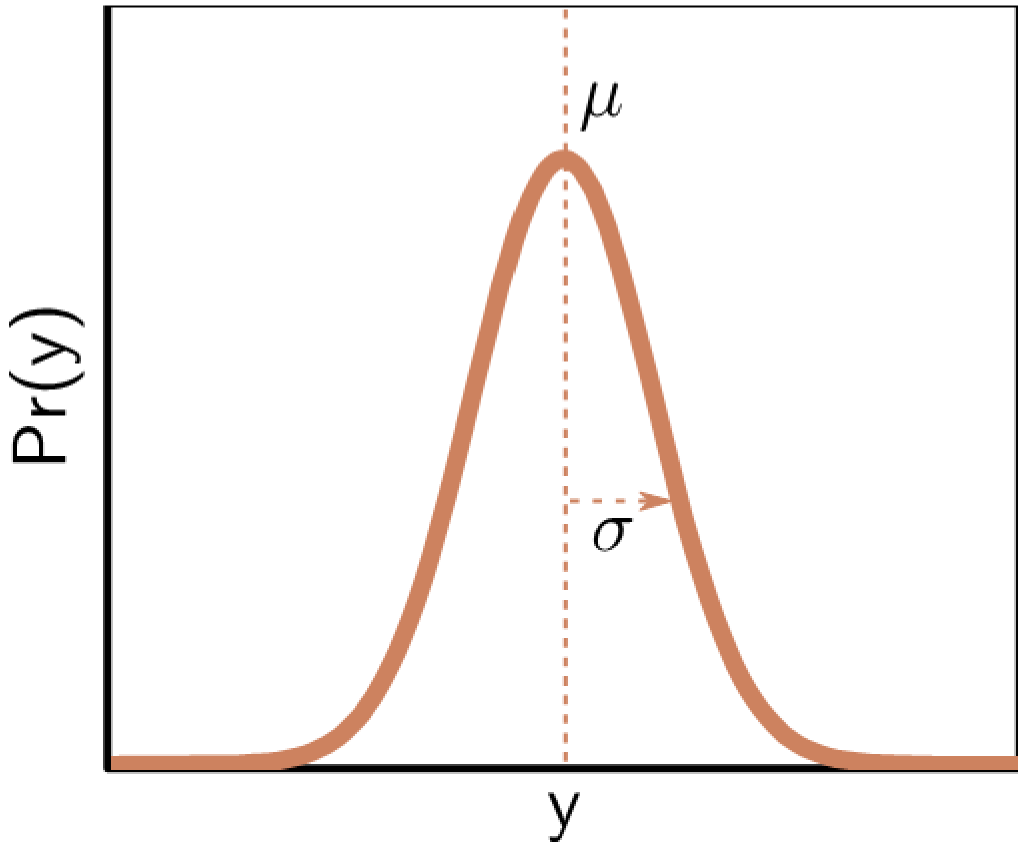
so the ultimate intuition becomes
"The model looks at an input and spreads its belief across all possible outputs.
The loss function checks whether the real answer lies in the high-belief region.
Training reshapes the model so reality becomes more probable next time."

Code and output add-on #implementation

our most natural next question would be "How does a neural network output a distribution ?"

How can a model predict a probability distribution ?

1. Pick a known distribution (e.g. normal distribution) to model output y with parameter θ
   e.g. The normal distribution, $\theta = \{\mu, \sigma^2\}$



Source: Link

2. use the model to predict the parameters θ of probability distribution

before that don't forget
"a neural network doesn't output a probability distribution or uncertainty or probability mass function, it only predict a number. but we are keeping in mind that even if it hidden, it does output a distribution with a number"

interpretation of recent two lines would be :

1. we assume "how the noise in our data will behave like this"
   example of this would be, in case of
   regression -> Gaussian
   classification -> Bernoulli
   multiclass -> Categorical
   Count data -> Poisson
   and this is a modeling choice (which made by us), not learned
2. Now as our Neural network does thing like $x \to \theta(x)$ (Where $\theta(x)$: is **parameters of a probability distribution**.)
   example:
   Gaussian -> $\mu(x)$, may be $\sigma(x)$
   Bernoulli -> $p(x)$
   Categorical -> logits -> softmax probabilities

So essentially,
the model doesn't predict y, it predict the shape of belief of y.

we can interchangeably use belief and probability

let's see this with gaussian example

Gaussian (Regression, with uncertainty)

$$p(y \mid x) = \mathcal{N}(y \mid \mu(x), \sigma^2(x))$$

Here:

$$\theta(x) = \{\mu(x), \sigma(x)\}$$

- Network outputs two numbers per target
- Often:
  - last layer → linear for μ
  - softplus / exp for σ to keep it positive

Confusing right ?......huuh

Remember
This is just a crucial mental flip.

so let's connect the belief with our knowledge:

Once we have

$$p(y|\theta(x))$$

we will automatically get

$$p(y|x)$$

but our original question:

> How do we know prediction quality?

Answer:

> By asking how likely the true output is under the predicted distribution.

But likelihood requires:

- a distribution
- parameters of that distribution

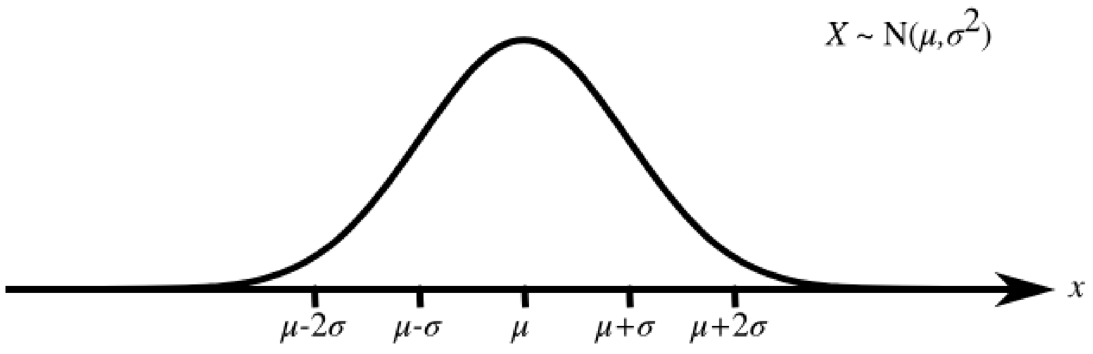
How i am saying this ?
#implementation

Let's Understand this with Examples

Example (Finally): Regression with MSE

Assumption:

$$y \sim \mathcal{N}(\mu(x), \sigma^2)$$



Where the top most is the μ that is mean.

Model predict:

$$\mu(x)$$

Loss:

$$-logP(y|\mu(x)) \propto (y - \mu(x))^2$$

Boom.
MSE suddenly isn't arbitrary anymore. It's probability theory in disguise.

ha ha

So the intuition here is
"A model predicts a conditional distribution $P(y \mid x)$ by assuming a parametric form for the output distribution and using the network to predict its parameters; the loss function then maximizes the likelihood of the true data under this distribution."

Okay Okay, Listen to me one last time:

we are trying to learn ML from first principle. this forces us to juggle between probabilities, modeling assumption, optimization, interpretation

by learning this way, may be we will be slow but,
...
we are learning in layers, not depth-first approach

Layer 1 (now)

> "Loss = penalty. Lower is better. Different tasks, different penalties."

Good enough.

Layer 2 (what you're touching now)

> "Losses come from likelihoods and distributions."

Messy but useful.

Layer 3 (later)

> "Choice of distribution = inductive bias + robustness + geometry."

This comes *after* you've trained models and broken them.

we're currently halfway between layer 1 and 2. That's the worst place emotionally.

Okay, enough motivations

Maximum Likelihood criterion

This is the what we wanted right ?
"The probability of the output that should be likely should be maximum w.r.t the parameters"

what i mean here is basically,

- Assign the highest possible probability to the observed (true) output under the model.

- Make the real data unsurprising to the model
- Training minimizes the negative log-likelihood so that the model assigns high probability to the true outputs given the inputs.

What we can think of ML training is:
"**Change your beliefs so that what actually happened stops looking shocking.**" for the model (ovio)

Back to topic

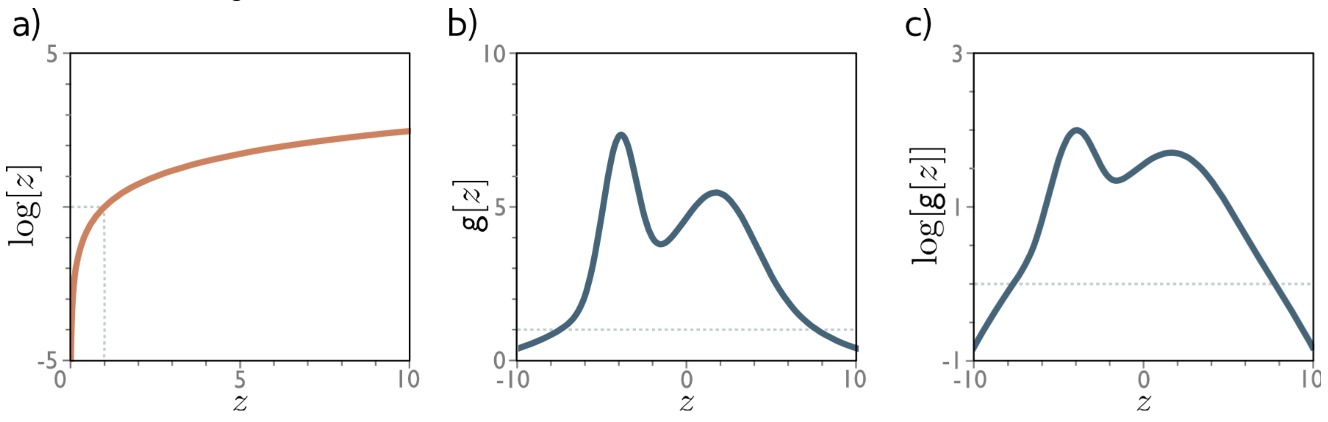
$$\begin{aligned}\hat{\phi} &= \underset{\phi}{argmin}[\prod_{i=1}^{I} P(y_i|x_i)] \\ &= \underset{\phi}{argmin}[\prod_{i=1}^{I} P(y_i|\theta_i)] \\ &= \underset{\phi}{argmin}[\prod_{i=1}^{I} P(y_i|f[x_i,\phi])]\end{aligned}$$

When we consider this probability as a function of the parameter ϕ , we call it a likelihood.

What are the problem can arise here ?

1. the term in this the product might all be small
2. the product might be so small that we can't easily represent it.

but we know Log function is monotonic



Monotonic means as its input increases, its output consistently increases.

maximum of logarithm will be the same as the maximum of function

So the maximum log likelihood

$$\begin{aligned}\hat{\phi} &= \underset{\phi}{argmax}[\prod_{i=1}^{I} P(y_i|f[x_i,\phi])] \\ &= \underset{\phi}{argmax}[\log[\prod_{i=1}^{I} P(y_i|f[x_i,\phi])]] \\ &= \underset{\phi}{argmax}[\sum_{i=1}^{I} \log[P(y_i|f[x_i,\phi])]]\end{aligned}$$

Now it's sum of terms, so doesn't matter so much if the terms are small. as it will always add up. unlike previously we were multiplying the losses. if the losses become small and small then they would vanish. so we won't get meaningful to optimize or interpret.

Minimizing negative log likelihood

By convention, we are minimizing things (i.e. Loss)

$$\begin{aligned}\hat{\phi} &= \underset{\phi}{argmax}[\sum_{i=1}^{I}\log[P(y_i|f[x_i,\phi])]] \\ &= \underset{\phi}{argmin}[-\sum_{i=1}^{I}\log[P(y_i|f[x_i,\phi])]] \\ &= \underset{\phi}{argmin}[L[\phi]]\end{aligned}$$

Maximizing can be seen as minimizing negative of that function. so...

Inference

- But now we predict a probability distribution
- we need an actual prediction (point estimate) (which we have known to be output can be single or multiple)
- Find the peak of the probability distribution (i.e. mean for normal for example)

$$\hat{y} = \underset{y}{argmax}[P(y|f[x|\theta])]$$

So again, Recipe for loss function

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

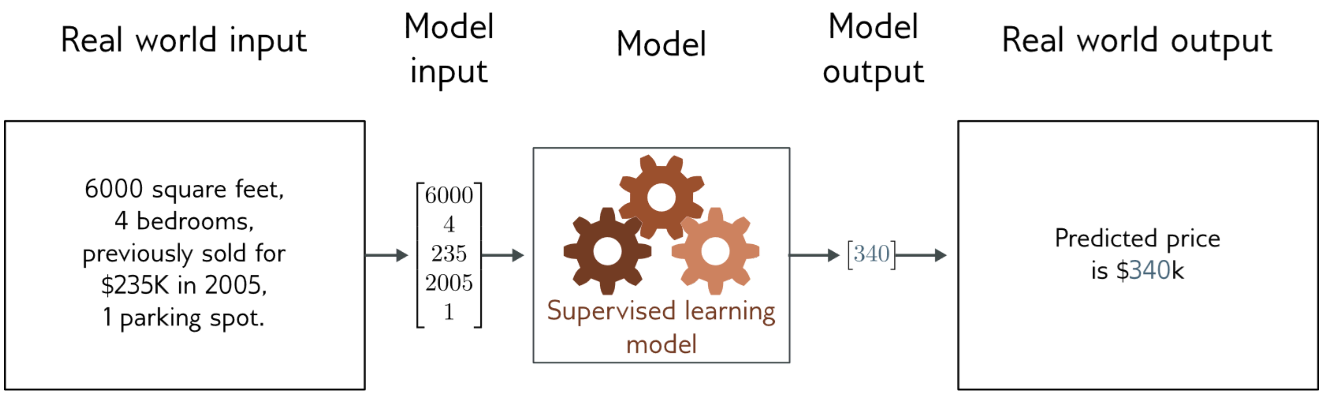
$$\hat{\phi} = \underset{\phi}{argmin}[L[\phi]] = \underset{\phi}{argmin}[-\sum_{i=1}^{I}\log[P(y_i|f[x_i,\phi])]]$$

4. To perform inference for a new test example x, return either the full distribution $P(y|f[x,\hat{\phi}])$ or the maximum of this distribution.

So we will Now use this framework to frame Loss functions for our Machine learning tasks.

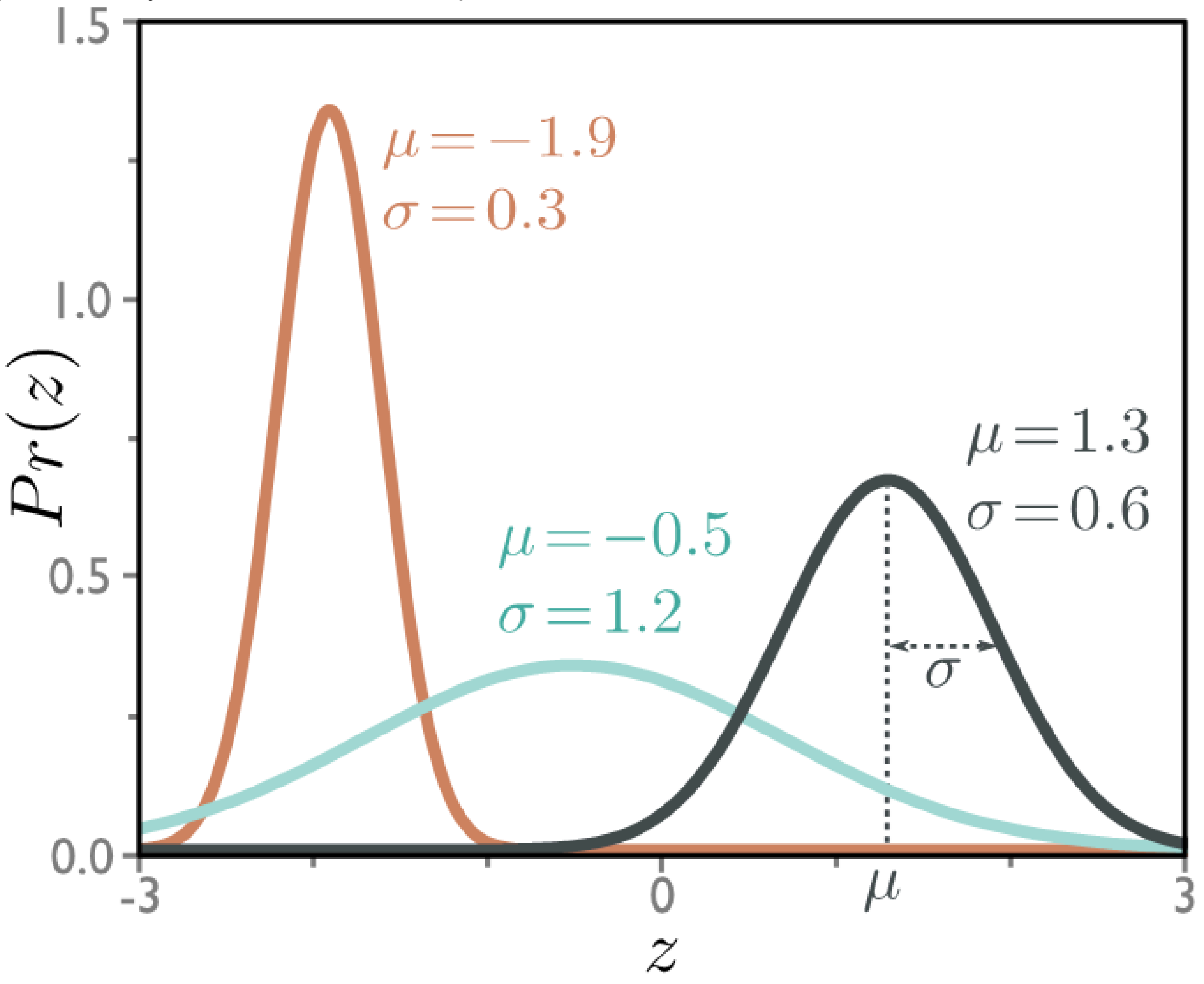
Examples (aha moments)

Example-1 : Univariate regression


Real world input
Model input
Model
Model output
Real world output
6000 square feet, 4 bedrooms, previously sold for $235K in 2005, 1 parking spot.
$\begin{bmatrix}6000\\4\\235\\2005\\1\end{bmatrix}$
Supervised learning model
[340]
Predicted price is $340k

Step-1

Choose a suitable probability distribution $P(y|\theta)$ that is defined over the domain of the predictions y and has a distribution parameters θ .



Here

Predicted Scalar Output: $y \in \mathcal{R}$

Sensible probability distribution:

- Normal distribution (there can be other distributions as well for example: Laplace, student's t-distribution, log normal, Gamma, Poisson, Beta, etc.)
- but the real rule is **Pick the distribution whose support and noise structure matches the data.**
- Here we are assuming the data is in normal distribution

$$P(y|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left[-\frac{(y-\mu)^2}{2\sigma^2}\right]$$

Step-2

Set the machine learning model $f[x,\phi]$ to predict one or more of these parameters so $\theta = f[x,\phi]$ and $P(y|\theta) = P(y|f[x,\phi])$

$$\begin{aligned} P(y|\mu,\sigma^2) &= \frac{1}{\sqrt{2\pi\sigma^2}} exp\left[\frac{-(y-\mu)^2}{2\sigma^2}\right] \\ P(y|f[x,\phi],\sigma^2) &= \frac{1}{\sqrt{2\pi\sigma^2}} exp\left[\frac{-(y-f[x,\phi])^2}{2\sigma^2}\right] \end{aligned}$$

Step-3

To train the model, find the network parameter ϕ that minimizes the negative log-likelihood loss function over the training dataset pair $\{x_i, y_i\}$

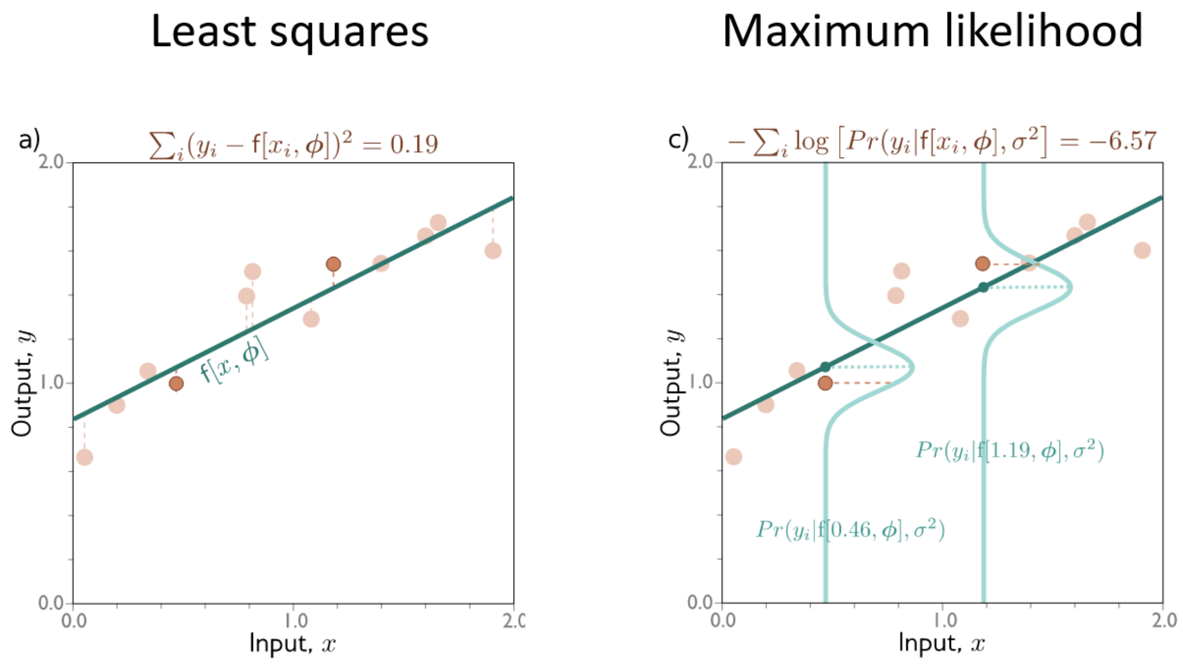
$$\begin{aligned} L[\phi] &= -\sum_{i=1}^{I} \log[P(y_i|\ f[x_i,\phi],\sigma^2)] \\ &= -\sum_{i=1}^{I} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}} exp\left[\frac{-(y-f[x,\phi])^2}{2\sigma^2}\right]\right] \end{aligned}$$

and

$$\begin{aligned} \hat{\phi} &= \underset{\phi}{argmin}\left[-\sum_{i=1}^{I} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}} exp\left[\frac{-(y-f[x,\phi])^2}{2\sigma^2}\right]\right]\right] \\ &= \underset{\phi}{argmin}\left[-\sum_{i=1}^{I} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\right] + \log\left[exp\left[\frac{-(y-f[x,\phi])^2}{2\sigma^2}\right]\right]\right] \\ &= \underset{\phi}{argmin}\left[-\sum_{i=1}^{I} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}}\right] + \frac{-(y-f[x,\phi])^2}{2\sigma^2}\right] \\ &= \underset{\phi}{argmin}\left[-\sum_{i=1}^{I} \frac{-(y-f[x,\phi])^2}{2\sigma^2}\right] \\ &= \underset{\phi}{argmin}\left[-\sum_{i=1}^{I} -(y-f[x,\phi])^2\right] \end{aligned}$$

Here we dropped the term σ^2 , as That’s fine **only because** σ^2 is assumed constant.

this is the least square (also popularly known as MSE).



Source: Link

Step-4

To perform inference for a new test example x, return either the full distribution $P(y|f[x,\hat{\phi}])$ or the maximum of this distribution.

$$P(y|\mu,\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left[-\frac{(y-\mu)^2}{2\sigma^2}\right]$$
$$P(y|f[x,\phi],\sigma^2) = \frac{1}{\sqrt{2\pi\sigma^2}} exp\left[-\frac{(y-f[x,\phi])^2}{2\sigma^2}\right]$$

Estimate variance:

in the expression

$$\hat{\phi} = \underset{\phi}{argmin}\left[-\sum_{i=1}^{I} -(y-f[x,\phi])^2\right]$$

the variance term (σ^2) disappeared. but we could learn it:

$$\hat{\phi}, \hat{\sigma^2} = \underset{\phi,\sigma^2}{argmin}\left[-\sum_{i=1}^{I} \log\left[\frac{1}{\sqrt{2\pi\sigma^2}} exp\left[\frac{-(y-f[x,\phi])^2}{2\sigma^2}\right]\right]\right]$$

Heteroscedastic regression

- Assume that the noise σ^2 is the same everywhere.
- But we could make the noise a function of the data x.
- Build a model with two output:
- Where our models for μ and σ^2 as follows

$$\mu = f_1[x, \phi]$$
$$\sigma^2 = f_2[x, \phi]^2$$
$$\hat{\phi} = \underset{\phi}{argmin}\left[-\sum_{i=1}^{I} \log\left[\frac{1}{\sqrt{2\pi f_2[x,\phi]^2}}\right] + \frac{-(y - f_1[x,\phi])^2}{2f_2[x,\phi]^2}\right]$$

so our optimal parameters would be as above.

we will read the above it again after knowing few thing
Step 1: Choosing a distribution
Learning the idea that:
A loss function does not come from nowhere.
It comes from an assumption about how the data is corrupted.

By choosing a Normal distribution, we are **hard-coding a belief**:

- errors are symmetric
- large errors are exponentially unlikely
- variance summarizes uncertainty

This choice of ours decide

- robustness to outliers
- shape of gradients
- training stability

Step 2: Predicting parameters, not y
Here we are not saying
“the network predicts y”

instead,
“the network predicts the *mean* of a distribution over y”
this explains:

1. why regression outputs are means
2. why uncertainty can be learned
3. why “prediction” and “belief” are different things

Step 3: Loss = negative log-likelihood
There should be a saying
"Loss is just a bookkeeping device for likelihood."

When we take logs and drop constants, something magical but very real happens:

- the Gaussian log-likelihood - collapses into - squared error

So MSE is not “simple” or “basic”.
It’s **a very specific modeling assumption in disguise**.

Step 4: Inference
After training, you have **more than a point estimate**.

we can:

- return the mean (classic regression)
- return the full distribution (uncertainty-aware ML)
- sample predictions
- compute confidence intervals

This is where ML stops being just curve fitting and starts being decision-making under uncertainty.
Cool....

So Here the patterns comes (which we talked at the starting)

```
Assume noise model
→ write likelihood
→ take negative log
→ simplify
→ get loss function
```

Simple idea but different belief about reality

by changing the assumption:

Change the assumption:

- Gaussian → MSE
- Laplace → MAE
- Bernoulli → binary cross-entropy
- Categorical → softmax + CE
- **Homoscedastic** → same variance everywhere
- **Heteroscedastic** → variance depends on x #implementation

So.. So.. So... Here is the intuition we will take with us for the example

> "This example shows that mean-squared error regression is equivalent to maximum likelihood estimation under a Gaussian noise assumption."

Homework

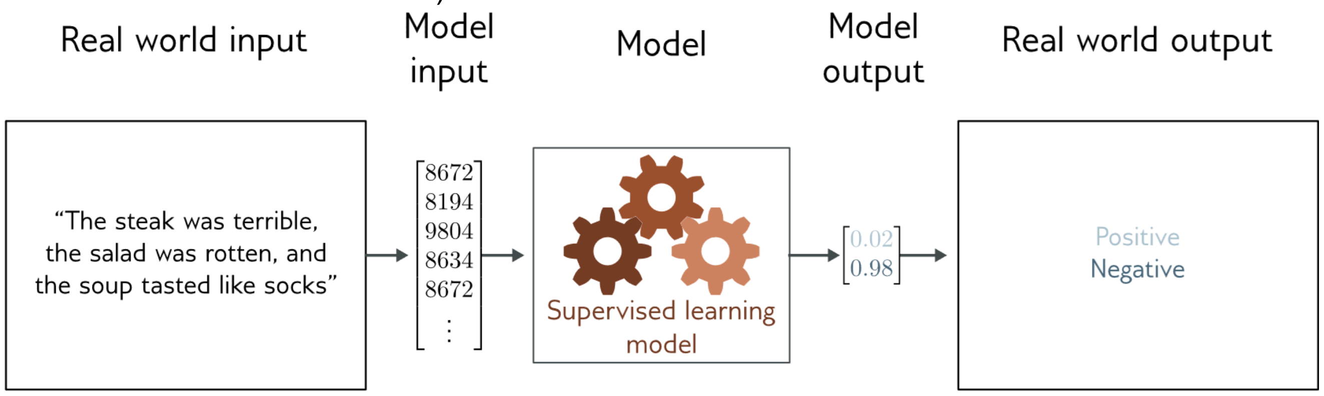
#homework

- MSE with outliers

- Gaussian NLL with skewed noise
- learned variance collapsing to zero
- heteroscedastic model gaming uncertainty
  See them breaking as these are the case where the loss functions breaks or fails

Example-2 : Binary Classification

we will repeat the same steps. (Okay so the cool thing is "we are treating all loss function as a result of a framework" coool)



Source: Link

Goal: which of the two classes $y \in \{0, 1\}$ the input belongs to

Step-1

Choose a suitable probability distribution $P(y|\theta)$ that is defined over the domain of the predictions y and has a distribution parameters θ .

Domain: $y \in \{0, 1\}$

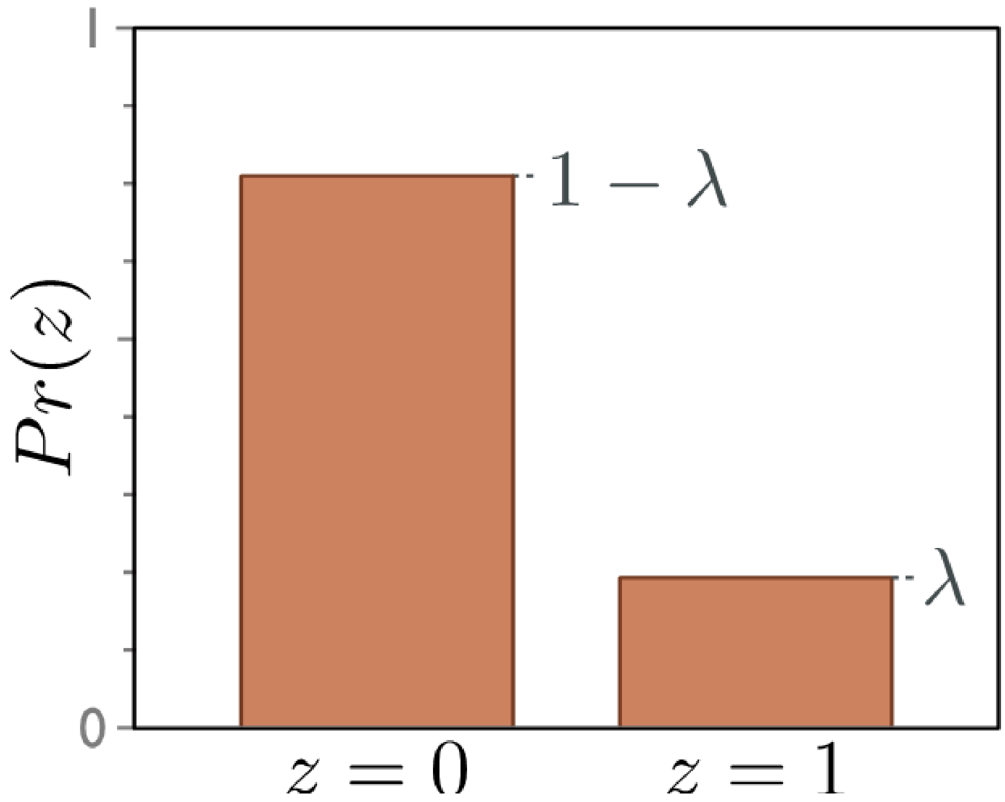
Bernoulli distribution

One parameter $\lambda \in [0, 1]$

$$P(y|\lambda) = \begin{cases} 1 - \lambda & \text{if } y = 0 \\ \lambda & \text{if } y = 1 \end{cases}$$

$$P(y|\lambda) = (1 - \lambda)^{1-y} \cdot \lambda^y$$

Ex:



Source: Link

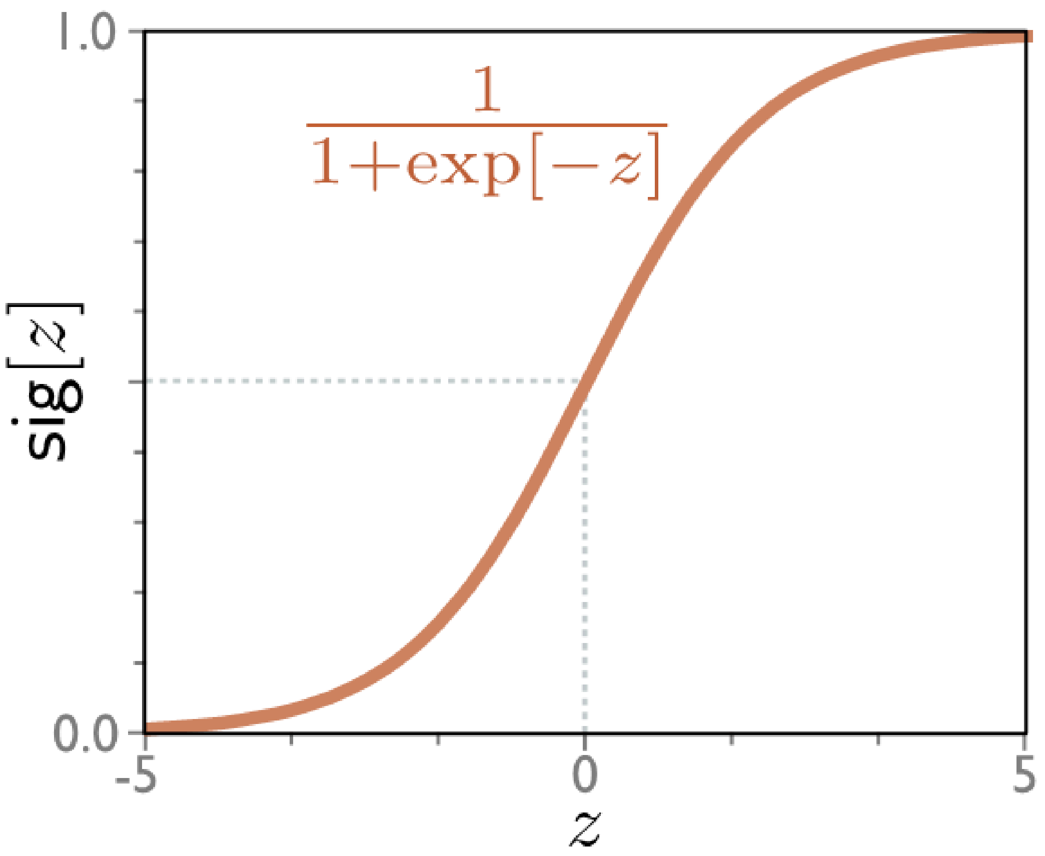
Step-2

Set the machine learning model $f[x, \phi]$ to predict one or more of these parameters so $\theta = f[x, \phi]$ and $P(y|\theta) = P(y|f[x, \phi])$

Problem:

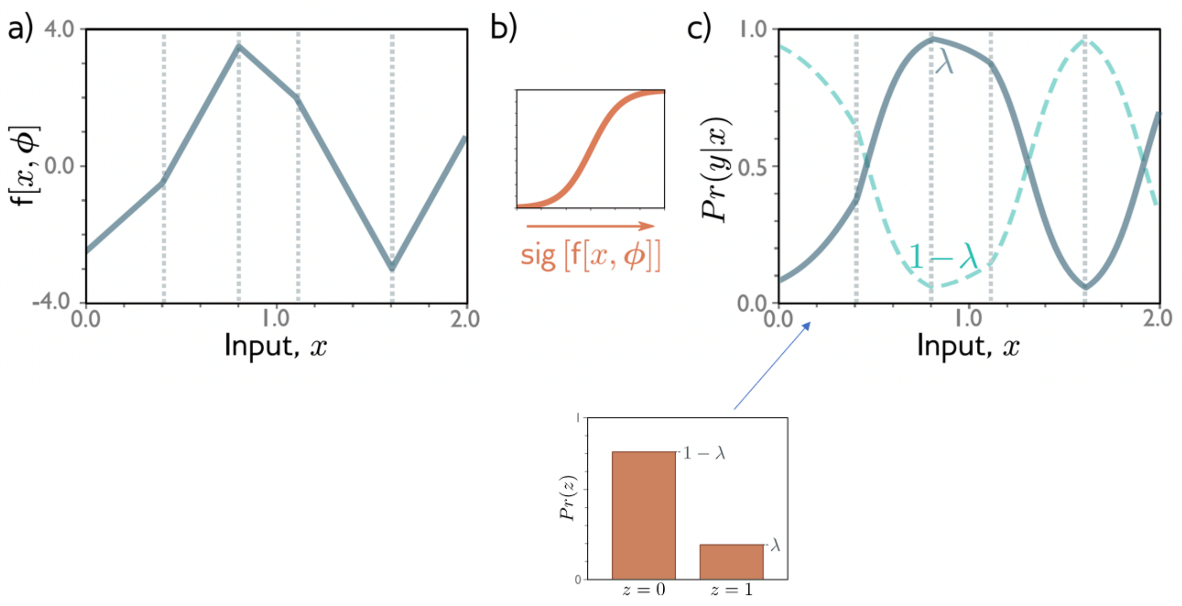
- Output of neural network can be anything
- Parameter: $\lambda \in [0, 1]$
  Solution:
- Pass through function that will map anything to $[0, 1]$
  Sigmoid function

$$sig[z] = \frac{1}{1 + exp[-z]}$$



Source: Link

$$P(y|\lambda) = (1-\lambda)^{1-y} \cdot \lambda^y$$
$$P(y|x) = (1 - sig[f[x|\phi]])^{1-y} \cdot sig[f[x|\phi]]^y$$



Source: Link

Here we can clearly see from (a) that we have range `[-4, 4]` and we are mapping it to `[0, 1]` with the help of activation function (sigmoid here)

Step-3

To train the model, find the network parameter ϕ that minimizes the negative log-likelihood loss function over the training dataset pair $\{x_i, y_i\}$

$$\hat{\phi} = \underset{\phi}{argmin}[L[\phi]] = \underset{\phi}{argmin}\left[-\sum_{i=1}^{I}\log\left[P(y_i|f[x_i|\phi]]\right]\right.$$

$$P(y|x) = (1 - sig[f[x|\phi]])^{1-y}.\, sig[f[x|\phi]]^{y}$$

$$L[\phi] = \sum_{i=1}^{I} -(1-y_i)\log[1 - sig[f[x_i|\phi]]] - y_i\log[sig[f[x_i|\phi]]]$$

or we can write this as

$$L[\phi] = \sum_{i=1}^{I} -(1-y_i)\log[1-\lambda_i] - y_i\log[\lambda_i]$$

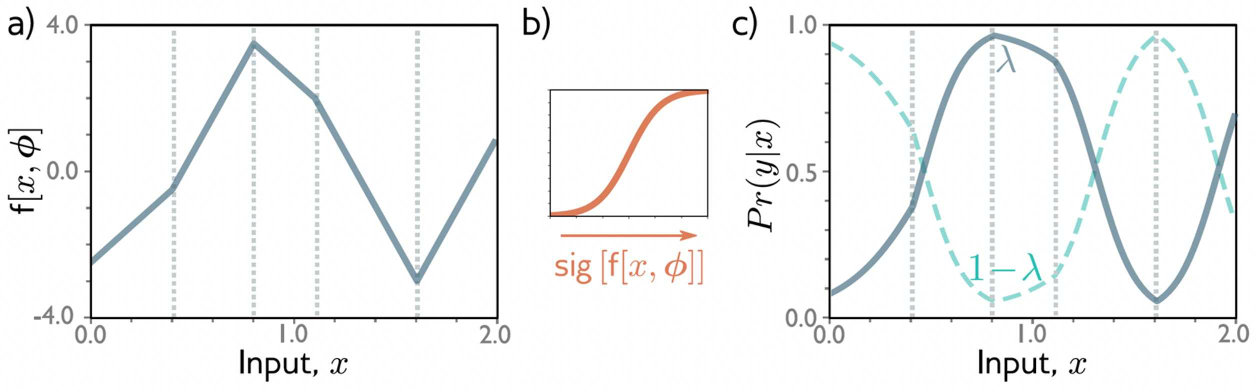
$$L[\phi] = -\sum_{i=1}^{I} [(1-y_i)\log[1-\lambda_i] + y_i\log[\lambda_i]]$$

Where

$$\lambda_i = sig[f[x_i, \phi]]$$

this is binary cross entropy

Step-4
To perform inference for a new test example x, return either the full distribution $P(y|f[x,\hat{\phi}])$ or the maximum of this distribution.



Source: Link

Choose y=1 where $\lambda > 0.5$, otherwise 0.

Interpretation:
Binary cross-entropy loss is not an arbitrary formula.
It is the negative log-likelihood of a Bernoulli distribution whose parameter is predicted by the model.

Step-1 Interpretation
Why Bernoulli?

> We choose a Bernoulli distribution because the target variable takes only two discrete values: 0 or 1. A Bernoulli distribution models the probability of a single binary event occurring.

Key ideas to internalize:

- The model is not predicting a class directly
- It is predicting **the probability that the class equals 1**
- The label y is treated as a random variable generated from this distribution

Mental model:

> "Given input x, how likely is class 1 to occur?"

Step-2 Interpretation
Why sigmoid appears here (and not magically)
A neural network can output any real number, but the Bernoulli parameter λ must lie in `[0,1]`.
Therefore, we use a sigmoid function to convert an unconstrained real-valued output into a valid probability.

Important conceptual shift:

- $f(x, \phi)$ produces **logits**, not probabilities
- `sigmoid(f(x))` converts logits into **beliefs**

This means:
The network is not saying "the answer is 1"
It is saying "I believe class 1 has probability 0.83"

Step-3 Interpretation
Where binary cross-entropy actually comes from

> Training the model means choosing parameters ϕ that make the observed labels most probable under the predicted Bernoulli distributions.

What negative log-likelihood does here:

- If the model assigns **low probability to the true label**, the loss explodes
- If it assigns **high probability**, the loss is small
- Wrong and confident predictions are punished the most

Interpretation of final loss

$$L = -(1-y)\log(1-p) - y\log(p)$$

This says:

- If y=1: penalize low p
- If y=0: penalize high p (or penalize low (1-p) which is same as penalize high p)
  Binary cross-entropy is therefore:

> A confidence-weighted penalty for being wrong.

Step-4 Interpretation
Inference vs training (important distinction)

> During training, we care about the full probability distribution because gradients depend on it.
> During inference, we often collapse this distribution into a single decision.

Key point:

- Returning the distribution preserves uncertainty
- Taking argmax (threshold at 0.5) gives a hard decision
  and the 0.5 threshold is **not sacred**.
  It reflects:
- symmetric costs
- balanced classes
- equal prior belief
  Change those assumptions, and the threshold should move.

So the final intuition becomes:
"In binary classification, we model the target as a Bernoulli random variable whose parameter is predicted by a neural network. Applying the sigmoid function ensures valid probabilities, and minimizing the negative log-likelihood of the Bernoulli distribution yields the binary cross-entropy loss. Thus, binary cross-entropy is a direct consequence of probabilistic modeling, not a heuristic choice."

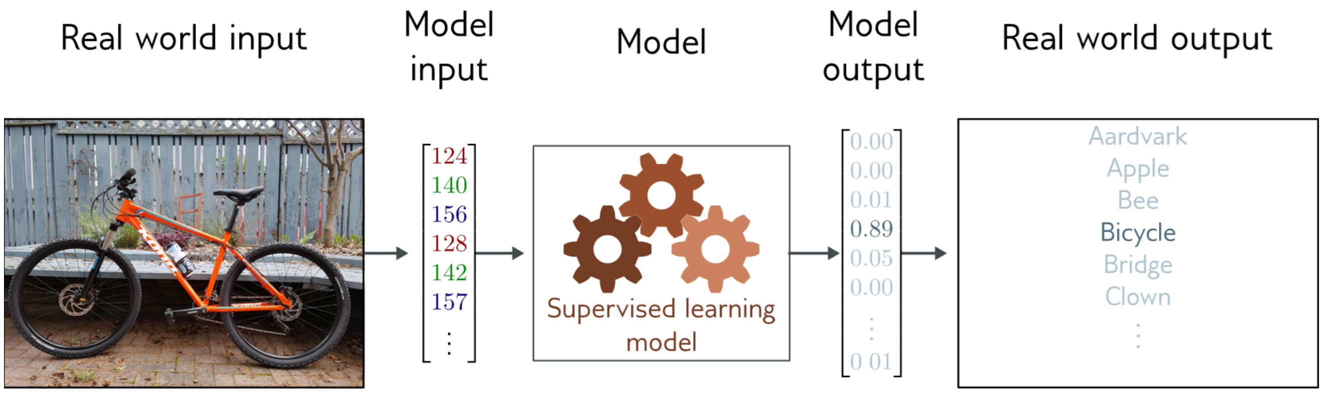
Before moving to next portion, Let's talk a bit :
This framework explains *why losses exist*.
It does not mean every practical model must be probabilistic.
So, we will try to but at the same time we won't over-romanticize the framework. (ha ha)

Homework

#homework

- BCE with label noise
- BCE with class imbalance
- Why sigmoid + BCE beats MSE for classification

Example-3: Multi-Class Classification



Source: Link

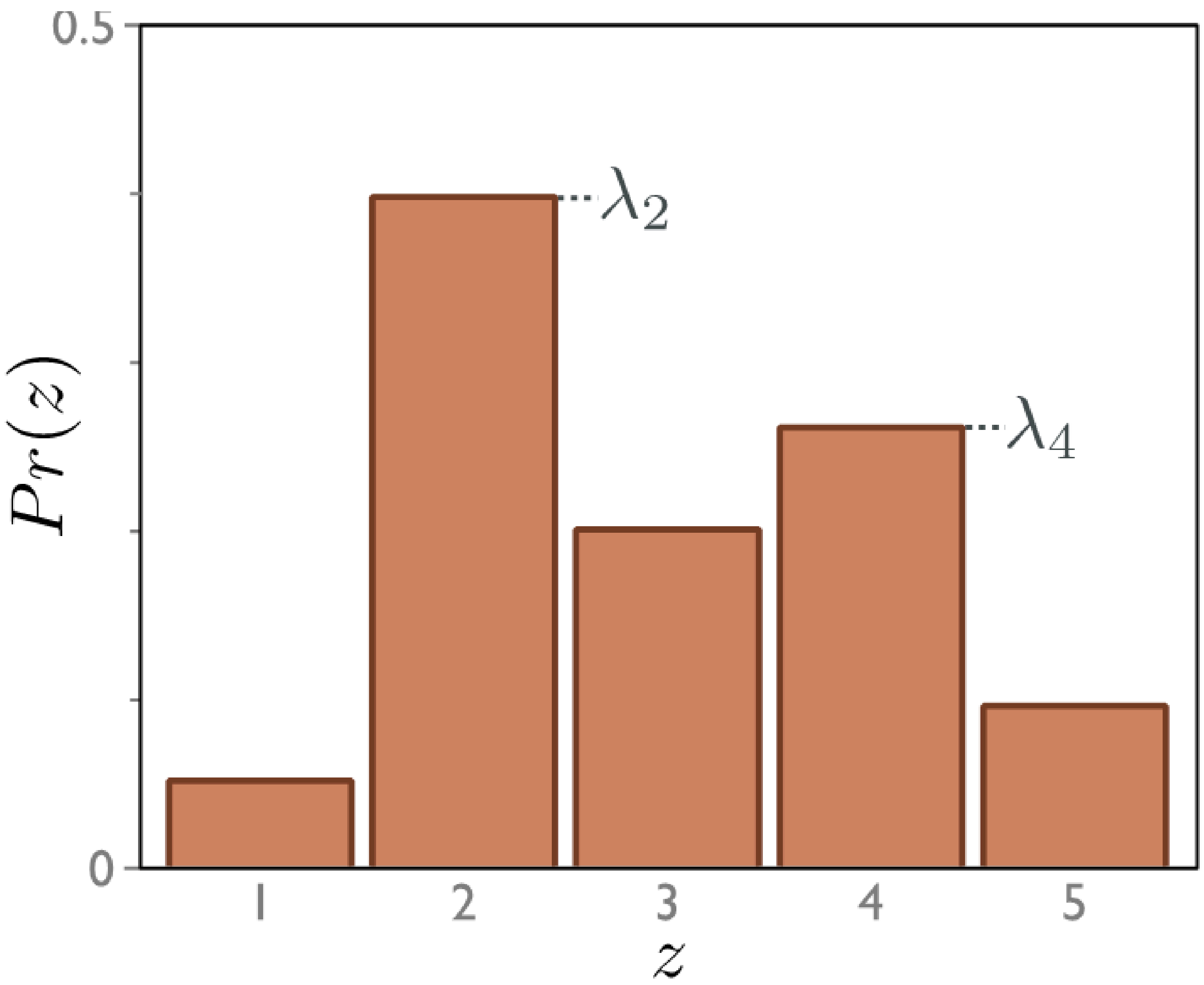
Step-1

Choose a suitable probability distribution $P(y|\theta)$ that is defined over the domain of the predictions y and has a distribution parameters θ .

Domain: $y \in \{1, 2, \ldots, k\}$
Categorical Distribution
K parameter $\lambda_k \in [0, 1]$
Sum of all parameters = 1

$$p(y = k) = \lambda_k$$

Ex:



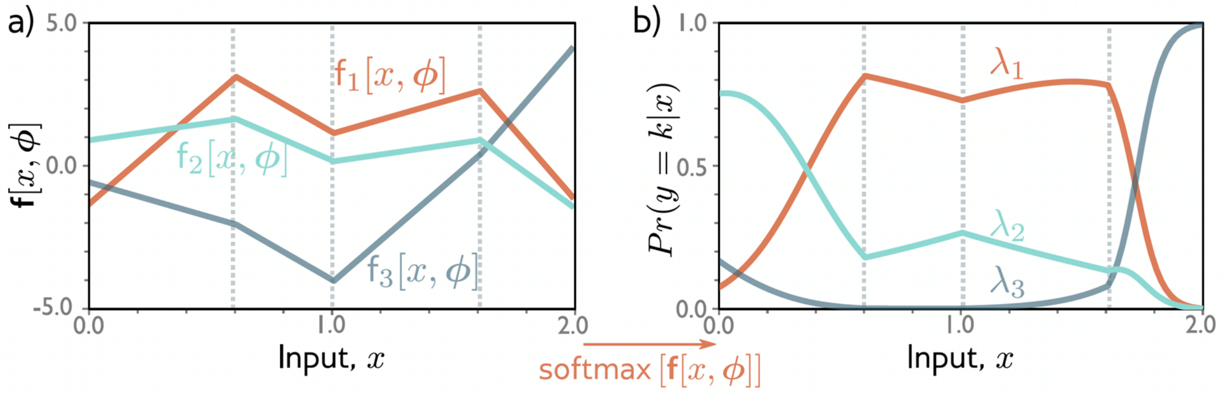
Source: Link

Step-2
Set the machine learning model $f[x,\phi]$ to predict one or more of these parameters so $\theta = f[x,\phi]$ and $P(y|\theta) = P(y|f[x,\phi])$

Problem:

- Output of neural network can be anything
- but the Parameters $\lambda_k \in [0,1]$
  Solution:
- Pass through function that map anything to `[0,1]` , sum to one

$$P(y=k|x) = softmax_k[f[x,\phi]]$$
$$softmax_k[z] = \frac{exp[z_k]}{\sum_{k'=1}^{K} exp[z_{k'}]}$$



Source: Link

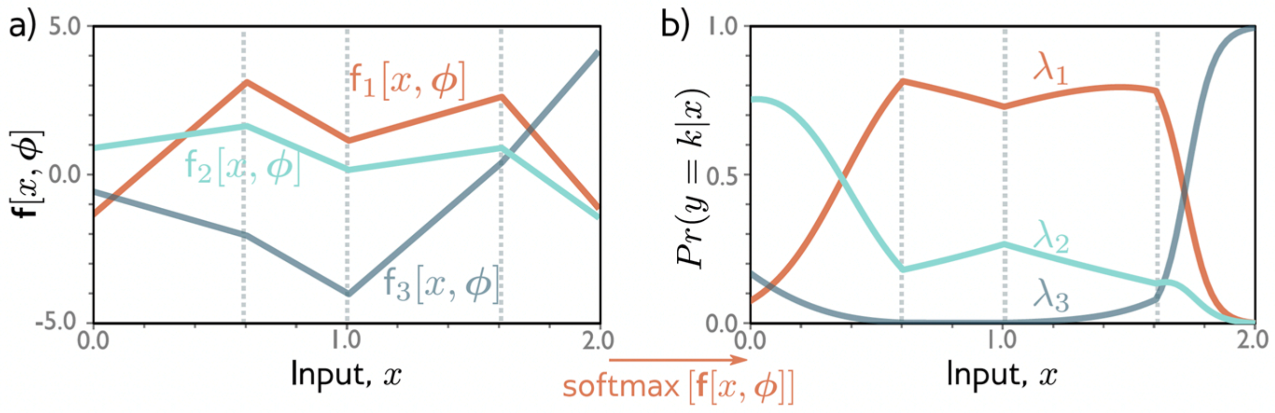
Step-3
To train the model, find the network parameter ϕ that minimizes the negative log-likelihood loss function over the training dataset pair $\{x_i, y_i\}$

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

$$\begin{aligned} L[\phi] &= -\sum_{i=1}^{I} \log[softmax_{y_i}[f[x_i,\phi]]] \\ &= -\sum_{i=1}^{I} \log\left[\frac{exp[f_{y_i}[x_i,\phi]]}{\sum_{k'=1}^{K} exp[f_k[x_i,\phi]]}\right] \\ &= -\sum_{i=1}^{I} f_{y_i}[x_i,\phi]] - \log\left[\sum_{k'=1}^{K} exp[f_k[x_i,\phi]]\right] \end{aligned}$$

This is multiclass cross entropy

Step-4
To perform inference for a new test example x, return either the full distribution $P(y|f[x,\hat{\phi}])$ or the maximum of this distribution.



Source: Link
Choose the class with largest probability

Interpretation:

simply in single line:

> Multiclass cross-entropy is the negative log-likelihood of a categorical distribution whose parameters are predicted by the model via softmax.

This completes the trilogy:

- Gaussian → MSE
- Bernoulli → Binary CE
- Categorical → Multiclass CE

Same framework. Different assumptions about reality. (ha ha)

Step-1 Interpretation
Why categorical distribution?

> We choose a categorical distribution because the output belongs to exactly one of K mutually exclusive classes.

Key things to internalize:

- Only **one class can be true** at a time
- Probabilities must:
  - be non-negative
  - sum to 1
- Each $lambda_k$ represents belief that *class k is the true one*
  Mental model:

  > “Given x, how should I distribute 100% of my belief across K competing hypotheses?”

Crazy ... right ?

This is **not** K independent Bernoulli problems.
That mistake causes bad models later.

Step-2 Interpretation
Why softmax exists (this is crucial)

> The neural network outputs unconstrained real values called *logits*.
> Softmax converts these logits into a valid probability distribution over classes.

Important conceptual shift:

- Logits are **relative scores**, not probabilities
- Only **differences between logits matter**
- Softmax enforces *competition* between classes

Key intuition:

> Increasing confidence in one class automatically decreases confidence in others.

That coupling is the entire reason softmax is used instead of K sigmoids.

Step-3 Interpretation
What multiclass cross-entropy is really doing

> Training maximizes the probability assigned to the true class while suppressing probability mass assigned to all other classes.

When you take the negative log:

- Correct class with low probability → large loss
- Correct class with high probability → small loss
- Confident but wrong → catastrophic penalty

The simplified loss:

$$L = -f_{y_i}(x) + \log \sum_k e^{f_k(x)}$$

Interpretation:

- First term: *reward the correct class*
- Second term: *penalize overall confidence unless justified*

Cross-entropy is therefore:

> A ranking-based loss that pushes the true class logit above all others.

it's not a "classification error", "distance". it is simply "relative belief correction"

Step-4 Interpretation
Inference vs belief

> During inference, we usually collapse the predicted distribution into a single class by taking argmax.

Important nuance:

- Argmax is a **decision rule**, not part of the model
- The model itself always outputs a distribution
- Thresholding / argmax reflects downstream assumptions, not learning

If costs change (e.g. false negatives are worse), argmax may be wrong even if the model is right.

Okay let's summerize everything we just covered.

> In multiclass classification, the target is modeled as a categorical random variable. The neural network predicts unnormalized class scores (logits), which are transformed via softmax into a probability distribution over classes. Minimizing the negative log-likelihood of the true class yields the multiclass cross-entropy loss. Thus, softmax and cross-entropy arise naturally from probabilistic modeling, not from heuristic design.

Few things to remember:
Softmax + cross-entropy is one object conceptually
They’re often split in notation, but mathematically they belong together. Treating them separately causes numerical and conceptual bugs.

This framework explains why MSE is bad for classification
MSE assumes Gaussian noise. Classes are not Gaussian. When people misuse MSE for classification, they’re silently assuming nonsense.

We’ve now completed the “loss functions from likelihood” arc
At this point, if someone asks:

> “Why this loss?”
> We actually have an answer.

Most people never do.

Okay tell me the answer (Ha ha)...

Homework

#homework

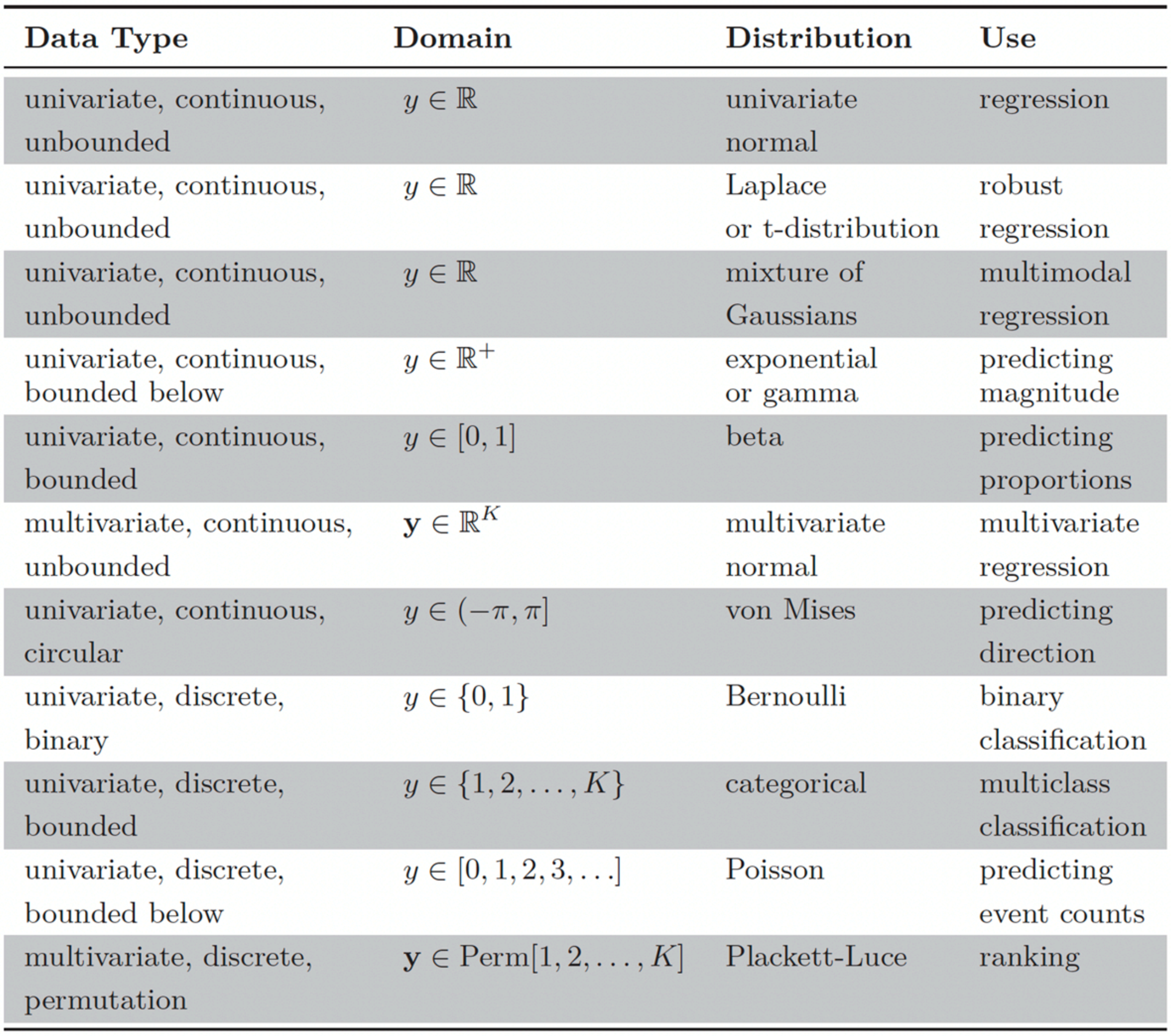
- why softmax saturates
- label smoothing as modified likelihood
- class imbalance as prior mismatch
- Interpret the graphs as well

Few more Homework

- multiclass (categorical + softmax)
- why softmax + CE is numerically special
- or where this framework *breaks*

Other Data type cases

#implementation



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

Figure 5.11 Distributions for loss functions for different prediction types.

Source: Link

Multiple Outputs

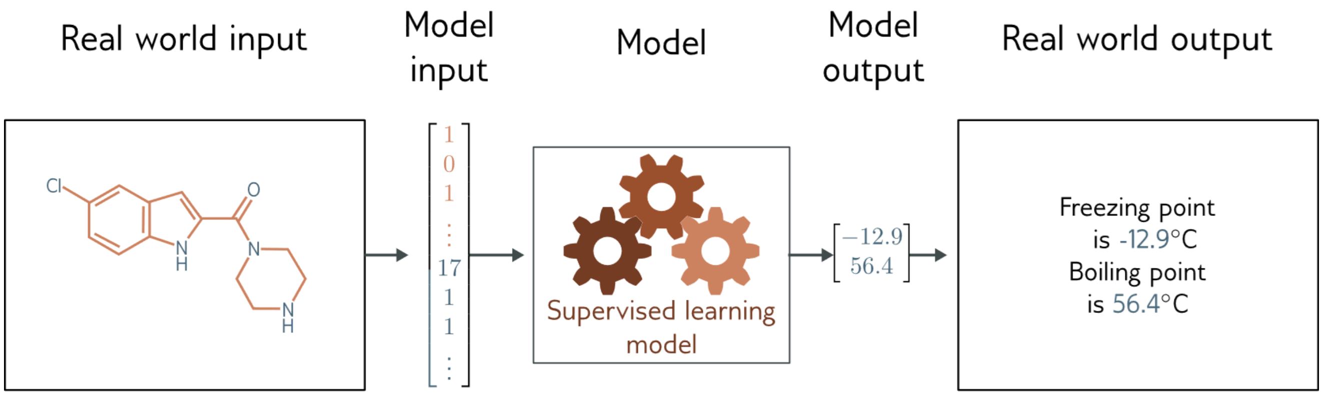
Treat each output y_d as independent

$$P(y|f[x_i,\phi]) = \prod_d P(y_d|f_d[x_i,\phi])$$

Negative log likelihood becomes sum of terms

$$L[\phi] = -\sum_{i=1}^{I} \log\left[P(y|f[x_i,\phi])\right] = -\sum_{i=1}^{I}\sum_d \log P(y_{id}|f_d[x_i,\phi])$$

Example-4: Multi-variate regression



Source: Link

Goal: to predict a multivariate target $y \in \mathcal{R}^{D_o}$

Solution treat each dimensions independently

$$\begin{aligned}P(y|\mu,\sigma^2) &= \prod_{d=1}^{D_o} P(y_d|\mu_d,\sigma^2)\\ &= \prod_{d=1}^{D_o} \frac{1}{\sqrt{2\pi\sigma^2}} exp\left[-\frac{(y_d-\mu_d)}{2\sigma^2}\right]\end{aligned}$$

Make network with D_o output to predict means

$$P(y|f[x,\phi],\sigma^2) = \prod_{d=1}^{D_0} \frac{1}{\sqrt{2\pi\sigma^2}} exp\left[-\frac{(y_d-\mu_d)}{2\sigma^2}\right]$$

What if output vary in magnitude

- E.g., predict weight in kilos and height in meters
- One dimension has much bigger numbers than others

Could learn a separate variance for each…
…or rescale before training, and then rescale output in opposite way

Interpretation:
(Independent Gaussian outputs → summed MSE)

"Multivariate MSE regression assumes that each output dimension is corrupted by independent Gaussian noise with equal variance."

We assume conditional independence between output dimensions given the input.
Formally

$$P(y|x) = \prod_{d=1}^{D_0} P(y_d|x)$$

Interpretation:

- Errors in different output dimensions do **not influence each other**
- There is **no correlation** between outputs
- The joint covariance matrix is diagonal
  and and and
  This is not “neutral”.
  This is a **strong modeling assumption**.

Interpretation of the likelihood
Each output dimension is modeled as its own univariate regression problem.
so this is now

$$P(y|\mu, \sigma^2) = \prod_d \mathcal{N}(y_d|\mu_d, \sigma^2)$$

means:

- One Gaussian per output
- Same noise level for all dimensions
- The total loss becomes the **sum of per-dimension squared errors**

This is why vector regression loss looks like:

$$\sum_d (y_d - \mu_d)^2$$

Not because it’s obvious.
Because we *assumed* it.

Interpretation of the network design
The network predicts a mean for each output dimension, not the vector y itself.

So when you build a network with D_o outputs, you are really predicting:

$$\mu = (\mu_1, \mu_2, \ldots, \mu_{D_0})$$

Each output neuron corresponds to the **mean of a marginal distribution**, not a coordinate in space.

This matters when:

- outputs have different units

- outputs have different noise scales
- outputs are correlated in reality

The “different magnitudes” problem
Your example:

- weight in kg
- height in meters

"MSE implicitly assumes all output dimensions are measured on comparable scales and have comparable noise."

If one dimension has larger numeric scale:

- it dominates the loss
- gradients mostly optimize that dimension
- smaller-scale outputs get neglected

This is not a bug.
This is exactly what the likelihood says should happen.

Two correct ways to handle it (and what they mean)

Option 1: Learn separate variances

Allow each output dimension to have its own uncertainty.

$$P(y_d|mu_d, \sigma_d^2)$$

Interpretation:

- Dimensions with higher noise get lower penalty
- Model learns which outputs are “harder”
- This is probabilistically clean

This is the principled solution.

Option 2: Rescale outputs (engineering solution)
Change the units so that variances become comparable.

Interpretation:

- You are manually enforcing equal noise assumptions
- Loss becomes numerically balanced
- Easier to train, less expressive

This is not wrong. It’s a tradeoff.

What this example is REALLY teaching (important)

> Multivariate MSE is not a multivariate Gaussian with full covariance.

we often *think* we are modeling:

$$\mathcal{N}(\mu, \Sigma)$$

we are not
we are modeling instead

$$\mathcal{N}(\mu, \sigma^2 I)$$

That’s a massive restriction.

If outputs are correlated:

- this model cannot express it
- no amount of data fixes that
- you need a full covariance model

let's summarize
"In multivariate regression, treating each output dimension independently corresponds to assuming conditional independence and identical Gaussian noise across dimensions. This leads to a summed MSE loss over outputs. Differences in output scale implicitly reweight the loss, which can be addressed either by learning separate variances per dimension or by rescaling the data. Thus, multivariate MSE reflects strong assumptions about noise structure and output relationships."

> We will derive the rest given our understand from above lecture notes and interpretation

Now let's Have Few example Illustrating How we will construct a loss function not blindly but carefully looking at the situation will implement all the loss function below.

but before diving into those let's clear few things or questions

Case-1:

we saw that if the data is gaussian then MSE helps. but what if the data is not gaussian, can we still use ?

answer: Yes !!
but then our

- Outliers will dominate training
- predictive uncertainty is meaningless
- you get a *mean of a distribution that may not even be representative*
  as gaussian distribution means

If you assume:

$$y \mid x \sim \mathcal{N}(\mu(x), \sigma^2)$$

and σ^2 is constant, then:

$$-\log p(y \mid x) \propto (y - \mu(x))^2$$

That’s where MSE comes from.
So:

- **MSE = Gaussian NLL(negative log likelihood) with fixed variance**

Now the question
if the data is gaussian then can we use loss function other than MSE ?

answer: Yes !!
we can use
MAE
Huber Loss (Hybrid of both MAE and MSE)
Quantile Loss

> The framework we discussed is a good way to Construct or design loss function for our task !!

it's like talking with yourself that

- Are outliers likely?
  → Squared error will explode.
- Is the target bounded or positive-only?
  → Gaussian is already wrong.
- Does variance change with input?
  → Fixed-σ MSE is lying.
- Are there multiple plausible outputs?
  → Mean prediction will be nonsense.
  That’s pattern recognition

You start with base line loss function (like MSE or Cross entropy)
See it break or fail (The loss curve, residuals, and predictions tell you the story)

- Residuals fan out → heteroscedasticity
- Rare points dominate gradients → heavy tails
- Predictions collapse to averages → multimodality
- Calibration is off → wrong likelihood

Upgrade the distribution or the loss

> A senior engineer doesn’t choose a loss because it matches a distribution.
> They choose it because it fails in the least harmful way.

Beginners ask:
“Which loss is correct?”

Experienced people ask:
“What assumption am I making, and how will it break?"

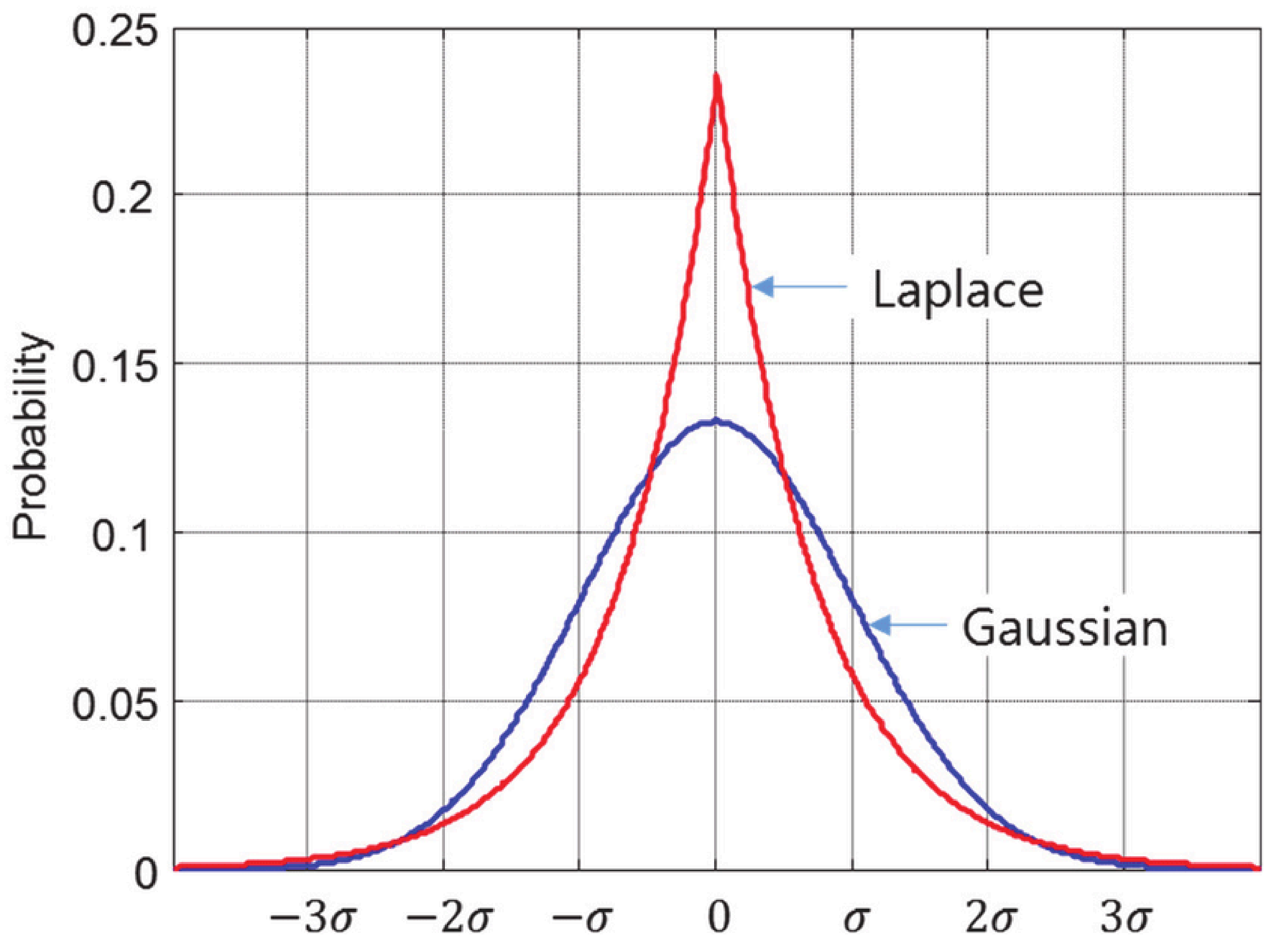
MSE

We already seen this as example-1 : Univariate regression

Note:
Mean Squared Error arises as a special case of maximum likelihood estimation under a Gaussian likelihood with fixed variance.
Using MSE implicitly assumes homoscedastic, symmetric noise and optimizes only the conditional mean.

MAE

Step-1: Choose a probability distribution
Assume the target follows a **Laplace (double exponential) distribution**:



Source: Link

$$P(y \mid \theta) = \text{Laplace}(y \mid \mu, b) = \frac{1}{2b} e^{-\frac{|x-\mu|}{b}}$$

Where
μ : Center
$b > 0$: Scale parameter (also called diversity term)

This distribution:

- is symmetric like Gaussian
- has **heavier tails**
- tolerates outliers

Step-2: Predict distribution parameters
Let the model predict only the location(center or mean):

$$\mu(x) = f[x, \phi]$$

Assuming scale b is constant.

So:

$$P(y \mid x) = \text{Laplace}(y \mid f[x, \phi], b)$$

Step-3: Optimize negative log-likelihood
The Laplace negative log-likelihood is:

$$\begin{aligned}\mathcal{L}_{\text{NLL}} &= -\log P(y \mid f[x,\phi]) \\ &= \frac{1}{b}|y - f[x,\phi]| + \log(2b)\end{aligned}$$

With constant bbb, minimizing NLL is equivalent to minimizing:

$$\boxed{\mathcal{L}_{\text{MAE}} = |y - f[x,\phi]|}$$

MAE is not “just absolute error”. It is **maximum likelihood under Laplace noise**.

Step-4: Inference
For a new input x:

$$\hat{y} = f[x, \hat{\phi}]$$

This corresponds to:

- the **median** of the Laplace distribution
- also the **maximum likelihood estimate**
  So:

Loss	What you estimate
MSE	Conditional mean
MAE	Conditional median

Replacing the Gaussian likelihood with a Laplace likelihood leads to the Mean Absolute Error loss.
MAE therefore estimates the conditional median rather than the mean and is more robust to outliers.

Let's talk about the outlier point.

MSE is more sensitive to outliers whereas MAE is not. Right !!

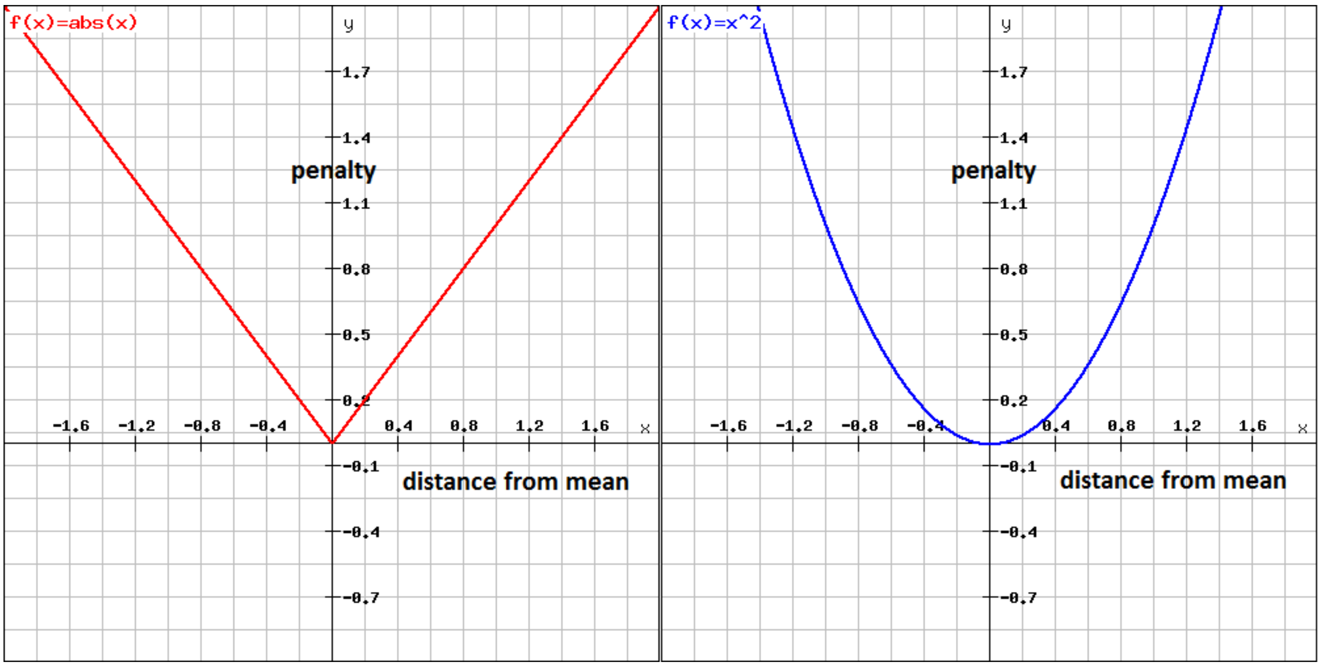
$$\text{MSE} = \frac{1}{m}\sum_{i=1}^{m}(\hat{y}_i - y_i)^2$$

Where as

$$\text{MAE} = \frac{1}{m}\sum_{i=1}^{m} \mid \hat{y}_i - y_i \mid$$

so if outlier exists then the distance would be very high. it's square would be even higher. so MSE will be higher as compare to MAE.

but MAE is not differentiable. as there is sharp corner (so from geometric we can tell it is not differentiable)



Source: Link

Huber Loss

Huber is not a true likelihood of a standard distribution.
It is a *hybrid*, of MAE and MSE
Let's discuss it's formula then go with interpretation

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{if } \mid a \mid \leq \delta \\ \delta(\mid a \mid - \frac{1}{2}\delta) & \text{if } \mid a \mid > \delta \end{cases}$$

where $a = y - f[x, \phi]$ is the residual, and δ is a threshold parameter.

The hybrid nature of Huber loss makes it less sensitive to outliers, similar to MAE, but also penalizes small errors within the data points, similar to MSE.

if the different is less than threshold then MSE and if more than the threshold then it's MAE. so the threshold is our hyperparameter. a new headache. (ha ha)

Step-1: Choose a noise model
Instead of committing to a single distribution, assume:

- small errors behave like **Gaussian noise**
- large errors behave like **Laplace noise**

This corresponds to a **piecewise error model**:

- quadratic near zero
- linear in the tails

Huber is a **robust M-estimator**. (we won't go into deeper now)

Huber loss corresponds to maximum likelihood under a distribution with Gaussian core and Laplace tails.

Step-2: Predict the location parameter
Same as MSE and MAE:

$$\mu(x) = f[x, \phi]$$

No variance or scale is learned.
So:

$$\theta(x) = \mu(x)$$

Step-3: Minimize the Huber loss
The Huber loss is defined as:

$$L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & \text{if } \mid a \mid \leq \delta \\ \delta(\mid a \mid - \frac{1}{2}\delta) & \text{if } \mid a \mid > \delta \end{cases}$$

where $a = y - f[x, \phi]$ is the residual, and δ is a threshold parameter.
Key observations:

- near zero → behaves like MSE
- far away → behaves like MAE
- gradients are bounded
  Step-4: Inference
  Inference is unchanged:

Loss	Noise assumption	Estimates	Outlier behavior
MSE	Gaussian	Mean	Catastrophic
MAE	Laplace	Median	Robust
Huber	Gaussian core + Laplace tails	Between mean & median	Controlled

About the parameter δ (important)

δ controls **where you stop trusting squared error**.

- small δ → more MAE-like
- large δ → more MSE-like

In practice:

- fixed by heuristic
- cross-validated
- sometimes tied to estimated noise scale

The Huber loss is a robust alternative to MSE and MAE that behaves quadratically for small residuals and linearly for large residuals.

It can be interpreted as maximum likelihood under a noise model with Gaussian behavior near the mean and heavy-tailed behavior for outliers.

Binary cross entropy

we already saw this with Example-2

Why MSE is wrong for binary classification (say it once, clearly)

If you use MSE for binary targets, you are assuming:

$$y \mid x \sim \mathcal{N}(p(x), \sigma^2)$$

That implies:

- outputs outside `[0,1]` are acceptable
- symmetric penalties for confident wrong predictions
- meaningless probabilities

BCE fixes all of this by:

- respecting the support
- penalizing confident mistakes heavily
- aligning loss with probability theory

Binary Cross Entropy arises as the negative log-likelihood of a Bernoulli distribution whose parameter is predicted by the neural network via a sigmoid transformation.
It trains the model to produce calibrated probabilities and is the correct loss for binary classification tasks.

it outputs probability and how we use that to final output is our choice (a design choice)

Hinge loss

Hinge loss is not a likelihood-based loss.

There is no clean probability distribution whose negative log-likelihood gives hinge loss.

but its a design choice.

Hinge loss comes from **large-margin optimization**, not probabilistic modeling.

If BCE is about *calibrated probabilities*, hinge loss is about *decisive separation*.

Step-1: Define the prediction space
Binary labels are encoded as:

$$y \in \{-1, +1\}$$

The model outputs a **score**, not a probability:

$$s(x) = f[x, \phi] \in R$$

- sign → class
- magnitude → confidence (margin)

Step-2: What is $\theta(x)$ here?
For hinge loss:

$$\theta(x) = s(x)$$

It is **not** a distribution parameter.
It is a **decision function**.

So instead of:

$$P(y \mid \theta)$$

you are implicitly optimizing:

$$y \cdot s(x)$$

the signed margin.

Step-3: Define the loss (margin-based)
The hinge loss is:

$$\boxed{\mathcal{L}_{hinge}(x, y) = \max(0, 1 - y \cdot s(x))}$$

Interpretation:

- if $ys(x) \geq 1$: correct with margin → **no loss**
- if $0 < ys(x) < 1$: correct but unsure → penalized
- if $ys(x) \leq 0$: wrong → heavily penalized

This enforces a **margin**, not likelihood.

Step-4: Inference
For a new input x:

$$\hat{y} = sign(s(x))$$

Aspect	**Hinge Loss**	**Binary Cross Entropy**
Philosophy	Margin maximization	Likelihood maximization

Aspect	Hinge Loss	Binary Cross Entropy
Output	Score	Probability
Distribution	None	Bernoulli
Penalizes confident mistakes	Yes	Yes (more smoothly)
Probability calibration	No	Yes
Used in	SVMs, large-margin models	Neural classifiers

When hinge loss is actually the right choice

- You care about **decision boundaries**, not probabilities
- You want **maximum separation**
- You don't trust class probabilities anyway
- Ranking and retrieval problems
- Old-school SVM energy, still effective

If you need calibrated uncertainty, hinge is the wrong tool.

Hinge loss is a margin-based loss used for binary classification that penalizes predictions violating a minimum confidence margin. Unlike cross-entropy, it does not model probabilities and instead optimizes for maximum separation between classes.

Focal Loss

There is this block we can look into: Link

Step-1: Choose the base probability distribution
For binary classification:

$$y \in \{0, 1\}$$

Base distribution is still **Bernoulli**:

$$P(y \mid \theta) = \text{Bernouli}\ (y \mid p)$$

So yes, focal loss is **built on top of BCE**, not a replacement.

Step-2: Predict the distribution parameter
As usual:

- network outputs a logit z(x)
- apply sigmoid:

$$p(x) = \sigma(z(x))$$

Define:

$$p_t = \begin{cases} p(x) & \text{if } y = 1 \\ 1 - p(x) & \text{if } y = 0 \end{cases}$$

This is the probability assigned to the **true class**.

Step-3: Modify the negative log-likelihood
Binary Cross Entropy:

$$\mathcal{L}_{BSE} = -\log p_t$$

Focal loss introduces two modifiers:

1. Focusing term $(1 - p_t)^\gamma$
2. Class weighting α

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

Interpretation:

- easy examples ($p_t \approx 1$) → down-weighted
- hard examples ($p_t \ll 1$) → dominate training

Step-4: Inference
Inference is unchanged:

$$\hat{p} = \sigma(z(x))$$

How focal loss relates to probability theory (important nuance)

- BCE = exact negative log-likelihood
- Focal loss = **reweighted likelihood**

It no longer corresponds to a proper likelihood of a standard distribution.

But:

- gradients still make sense
- probabilities remain interpretable
- calibration may degrade slightly

This is a conscious trade-off.

When focal loss is the right weapon

Use focal loss when:

- extreme class imbalance
- vast majority of samples are trivial

- detector models (object detection, segmentation)
- BCE converges but ignores rare positives

Do **not** use focal loss when:

- classes are balanced
- you care deeply about calibrated probabilities
- you just want a clean baseline

Loss	**Handles imbalance**	**Probabilistic**	**Focus on hard samples**
BCE	Poor	Yes	No
Weighted BCE	Medium	Yes	Weak
Focal	Strong	Yes-ish	Aggressive
Hinge	Medium	No	Margin-based

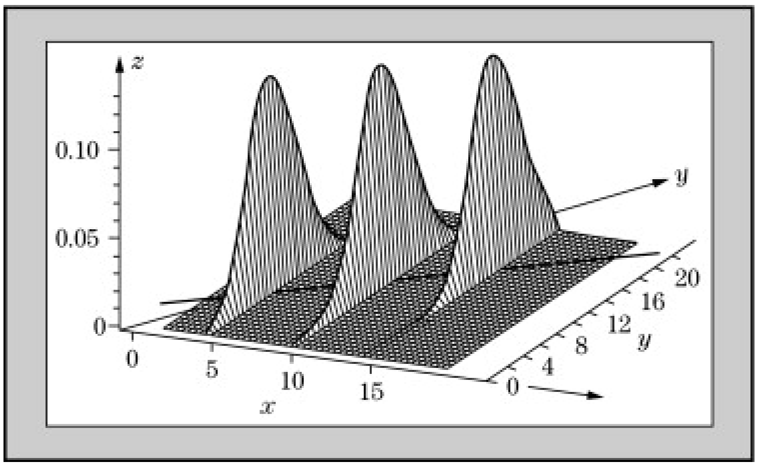
Focal loss is a modification of binary cross entropy that down-weights well-classified examples and focuses training on hard, misclassified samples. It is particularly effective for highly imbalanced classification problems.

Categorical cross entropy

Step-1: Choose a probability distribution
For multi-class classification with K classes:

$$y \in \{1, 2, \dots, K\}$$

The correct distribution is **Categorical** (a.k.a. Multinoulli):



Source: Link

$$P(y \mid \theta) = Categorical(y \mid \pi)$$

where

$$\pi = (\pi_1, \pi_2, \pi_3, \dots, \pi_K), \quad \sum_k \pi_k = 1$$

This encodes mutual exclusivity.

Step-2: Predict the distribution parameters
The network outputs **logits**:

$$z(x) = f[x, \phi] \in R^K$$

Convert logits to probabilities using softmax:

$$\pi_k(x) = \frac{e^{z_k(x)}}{\sum_j e^{z_j(x)}}$$

So:

$$P(y \mid x) = Categorical(y \mid softmax(f[x, \phi]))$$

$\theta(x)$ is now the full probability distribution $\pi(x)$

Step-3: Minimize negative log-likelihood
Categorical likelihood:

$$P(Y = k \mid \pi) = \pi_k$$

Negative log-likelihood:

$$\mathcal{L}_{NLL} = -\log \pi_y$$

with on-hot labels $y_k \in \{0, 1\}$:

$$\boxed{\mathcal{CE} = -\sum_{k=1}^{K} y_k \log \pi_k(x)}$$

This *is* categorical cross entropy.

Step-4: Inference
For a new input x:

- Predicted distribution

$$\pi(x) = softmax(f[x, \hat{\phi})$$

- Predicted class

$$\hat{y} = \underset{k}{argmax} \pi_k(x)$$

The argmax is a decision rule, not part of training.

Why CCE is not optional for multiclass problems

If you use

- MSE → wrong geometry
- independent BCEs → violates exclusivity
- hinge loss → ignores probability structure

CCE:

- respects simplex constraints
- penalizes confident wrong predictions sharply
- produces calibrated class probabilities (when trained sanely)

This is why it’s the default.

Relationship to BCE (people mess this up)

- BCE → multiple independent Bernoulli variables
- CCE → one categorical variable

Use BCE when:

- labels are multi-label (not exclusive)

Use CCE when:

- exactly one class is correct

Wrong choice here silently ruins training.

Categorical Cross Entropy arises as the negative log-likelihood of a categorical distribution whose parameters are predicted via a softmax transformation. It is the correct loss for multi-class classification problems with mutually exclusive classes.

Sparse Categorical cross entropy

What changes and what does not

Nothing changes conceptually.
Only the **label representation** changes.

- Categorical Cross Entropy: one-hot labels
- Sparse Categorical Cross Entropy: integer labels

That’s it. Same distribution. Same likelihood. Same gradients.

Step-1: Choose a probability distribution

Exactly the same as before.
For K-class classification:

$$P(y \mid \theta) = \text{Categorical}(y \mid \boldsymbol{\pi})$$

where:

$$y \in \{0, 1, \ldots, K-1\}$$

Step-2: Predict the distribution parameters

Network outputs logits:

$$\mathbf{z}(x) = f[x, \phi] \in \mathbb{R}^K$$

Apply softmax:

$$\pi_k(x) = \frac{e^{z_k(x)}}{\sum_j e^{z_j(x)}}$$

So:

$$\theta(x) = \boldsymbol{\pi}(x)$$

Step-3: Minimize negative log-likelihood (sparse form)

Instead of a one-hot vector $\mathbf{y}$, you have a class index y.
Likelihood:

$$P(y \mid \boldsymbol{\pi}) = \pi_y$$

Negative log-likelihood:

$$\boxed{\mathcal{L}_{\text{Sparse CCE}} = -\log \pi_y(x)}$$

That is literally the categorical cross entropy with all the zero terms removed.

Step-4: Inference

Same as before:

$$\hat{y} = \arg\max_k \pi_k(x)$$

Why sparse exists (practical, not philosophical)

Sparse CCE exists because:

- one-hot vectors waste memory
- indexing is faster
- large vocabularies make one-hot impractical
- frameworks can fuse ops efficiently

In NLP or vision with thousands of classes, sparse is not optional. It’s survival.

Common mistakes worth calling out

1. **Thinking it's a different loss**
   It's not. It's a different *input format*.
2. **Using BCE instead**
   BCE assumes independence. CCE assumes exclusivity. Wrong swap breaks learning.
3. **Applying softmax twice**
   Frameworks expect logits.

> Sparse Categorical Cross Entropy is the negative log-likelihood of a categorical distribution when class labels are provided as integer indices instead of one-hot vectors.

Softmax cross entropy

Step-1: Choose a probability distribution

Nothing new here.

For (K) mutually exclusive classes:

$$y \in \{1, 2, \dots, K\}$$

Choose a **Categorical distribution**:

$$P(y \mid \theta) = \text{Categorical}(y \mid \boldsymbol{\pi})$$

where:

$$\boldsymbol{\pi} \in \Delta^{K-1}$$

Same as categorical cross entropy.

Step-2: Predict the distribution parameters

The model outputs **logits**, not probabilities:

$$\mathbf{z}(x) = f[x, \phi] \in \mathbb{R}^K$$

Softmax converts logits into valid probabilities:

$$\pi_k(x) = \frac{e^{z_k(x)}}{\sum_j e^{z_j(x)}}$$

So:

$$\theta(x) = \boldsymbol{\pi}(x) = \text{softmax}(f[x, \phi])$$

Step-3: Minimize the negative log-likelihood (fused form)

Categorical NLL:

$$\mathcal{L} = -\log \pi_y(x)$$

Substitute softmax:

$$\mathcal{L} = -\log\left(\frac{e^{z_y}}{\sum_j e^{z_j}}\right) \$\$ Simplify:$$

\boxed{
\mathcal{L}_{\text{Softmax CE}}
= -z_y + \log \sum_j e^{z_j}
}

That expression **is** softmax cross entropy. ### Step-4: Inference At test time:

\boldsymbol{\pi}(x) = \text{softmax}(f[x,\hat{\phi}])

\hat{y} = \arg\max_k \pi_k(x)

Why "softmax cross entropy" exists as a separate name Because computing this:

\log(\text{softmax}(z))

$$naivelycauses: -overflow - underflow - NaNsSoframeworksimplement:$$

\text{LogSoftmax} + \text{NLL}

as **one numerically stable operation**. That fused thing is called **softmax cross entropy**. Same math.