SECTION 5: (PART I)
Training a Neural Network

Training: Section Introduction and outline (video 27)

- Last section → Architecture of a NN, how to pass data from input → output (Probabilistic prediction)
- Problem → All weights were random!
- Thus, predictions were NOT accurate!

$$P(y|x) = \text{Softmax}\left(V^T f(W^T x)\right)$$

- This section → Change the weights such that the predictions are accurate.
  - → Recall → Training data consists of (input, target) pairs.
  - → GOAL → Make predictions (y) as close as targets (t) as possible.

How? → Create a cost function such that:
- Cost is large: when y not close to t.

- Cost is small: when y is close to t.

⇑

- Want to make it as small as possible! (As a good businessman!)

- In NN's, (GD) has a special name:

  Backpropagation

  ⇓

- Recursive in nature.

  ↳ Allow us to find the gradients in a systematic way.

  -> Can also do BPP for NN that are arbitrarily deep w/o much added complexity.

  -> e.g. 1 hidden-layer no harder than 100-layer ANN.

← Method → GD

- Harder with (NN) b/c equations are more complex.

  ↳ Requires care but no new skills!

- Finally, we will apply our usual pattern:

  Theory
  ⇓
  Code

- First, we apply BPP to our 2D problem (3 blobs)

  Allow us to visualize the problem the ANN is solving.

- Next: e-commerce data
  - ① LoR + softmax -> we didn't have the chance to apply
  - ② NN + softmax LoR + softmax

- What do all these symbols and letters mean (video 28)
  - This lecture: Enumerate the letters / Symbols we use.
  - Why is this important? -> Some conventions conflict with each other.

Always be aware of the context -> It will give you clues.

Training Data

- Training inputs: X -> $X \in \mathbb{R}^{N \times D}$ N: Number of samples, D: Number of features
- Training targets: Y -> $Y \in \mathbb{R}^{N \times 1}$ -> a.k.a A column vector.
- Generally these are matrices A 2D object

⇓

Convention in code. <= Alternatively, Y can just be a vector of 1-D of length N.

- This is how it will be represented in Numpy.

Training Data and Predictions

- Inputs -> X, Targets -> Y, Predictions: $P(Y|X)$
- $p(Y|X)$ represents a full probability distribution over all individual values in the matrix Y, given the matrix X.
- $p(Y|X)$ is therefore also a matrix, same size as Y.
- $p(y=K|x)$ is a probability value - A single number.

  Represents the probability that y is of class K, given the input vector x.

- Note -> CAPITAL Letters usually represent matrices.
  -> Lowercase letters usually represent vectors.

$P(Y|X)$ is inconvenient

- $P(Y|X)$ is inconvenient to write.
  - ↳ Many characters, no parentheses allowed for naming in code.
- Thus we can use:
  - $P\_Y\_given\_X$
  - PY_X
  - PY

  None are really ideal.
- old school alternatives for predictions -> seen in LR class: $\hat{y}$

- Another convention ~~>~~ somewhat confusing:

- More common in DL literature
- Inputs: X
- Targets: T
- Predictions: Y'

Use the Context! <= {

- Y' and T at the same time
  => Y is a prediction
- Y' -> output of a NN
  => Y is a prediction
- Y' / Y_hat, or Y /
  p-y-given-x at the same time => Y is a Target

- Now using Y for something else!

Now -> No need to write $p(y|x)$ everywhere!

- These conventions can be quite confusing, however, both conventions are useful as follows:

Y / T => Allows us to write equations / code more simply.

Y / $P(Y|X)$ => Tells you what it is being represented (e.g. Probability)

↳ Good to be exposed to both so we can know what they mean when we encounter them!

Weights

- Input-to-hidden weight matrix: $W (D \times M)$
- Hidden bias: $b (M)$
- Hidden-to-output weight matrix: $V (M \times K)$
- output bias: $c (K)$

↳ Recall: W, b, V, c -> Map the vector from the previous layer to the next layer in the NN.

Justifies the dimensions.

- How about for larger NN's?

↳ Adding more layers -> We are going to start running out of letters!

||

⇓

- Numbering W's ← what do we do?
and b's?

$W_1, b_1, W_2, b_2, \dots$ etc.

- How should we index them? -> Don't mix indices and layer identities!

Main point: Don't mix them up! ⟸ { options: $W_1(i,j)$, $W^{(1)}_{(i,j)}$, $W_1[i,j]$

Indices

↳ • We may or may put indices in different places if they represent different things.
• Ex. Target T for the n^{th} sample and k^{th} class.

Super- or subscript?
- In Numpy: T[n,k]
- $T(n,k)$
- T^n_k
- T_{nk}
- t_{nk}
- t^n_k

K: NN output node
N: Number of samples in our data.

⇑

=> Key point: Just make sure you know what everything means
↳ Need to make sure what you wrote.

- Indexing
  - From programming: i, j, k -> Common letters used for indexing

    Ex. $i_1 \cdots D$ (Input Layer)
  - Problem -> In NN (Deep ones) we'll run out of letters.

    ⇓ -> Solution:
    - Common to choose letters not being used for something else
    - Ex. $q = 1 \cdots Q$

      ↑ Lowercase index for letters

      ↑ uppercase letter for upper limit.

- Learning Rate:

  Greek letters: Alpha or eta

$$\theta \leftarrow \theta - \eta \frac{\partial J}{\partial \theta}$$

$$\theta \leftarrow \theta - \alpha \frac{\partial J}{\partial \theta}$$

- Cost / Objective / Error
  - Typical letters: E or J
  - Cost or error: Usually means something we want to minimize
  - Objective: Can be something we want to minimize or maximize.
  - Probabilistic interpretation of cost: Negative-log-likelihood
  - Minimizing E == Maximizing -E
  - Minimizing negative log-likelihood (GD) is the same as maximizing log-likelihood (And likelihood) (GA)

Likelihood

↳ • Typically use uppercase L for likelihood, lowercase l for log-likelihood. => If presented together.

• If talking only about the log-likelihood or negative-log-likelihood we might just use L.

(L) -> Easy to see / l can be confused for I!

Summary -> Notations & conventions
-> Context -> where the letter is placed relative to other letters should give you a sense of what it is used for.

What does it mean to train a Neural Network (video 29)

- Introduction of a few ideas for the next few lectures.
- First, recall the outline of this course:
  - Previous section -> How to make predictions / How to go from input to output.
- These predictions did NOT make sense, the weights were random!
- This section -> How to find out what these weights should actually be?

⇓

Consider the following perspectives:

① The "Scikit-Learn" Perspective

- Each Scikit-Learn perspective has 2 primary functions:
  - fit(X, Y)
  - predict(X)

where: • fit(X, Y) -> used for training the model.

OUR Focus!

• predict(X) -> used for making new predictions using the trained model.

- In Some Sense:
  - This course is backwards, because we discussed prediction before training / Fitting ⇐
- In "Real-life" -> We must train the NN before it can make meaningful predictions!
- Practically however:
  - • Training is hard! Prediction is easy!
    - Training requires us to make predictions first.

Misconception

- while the training algorithm for NN has a special name ("Backpropagation"), the method we apply is NOT special at all.
- BPP -> Does not require any new concepts beyond what you already know about linear / LoR! → It is just "technically" more difficult.

What is this method?

- Be familiar w/ those steps → These are the steps you will apply in pretty much every ML model you learn:

1) Define a cost function L:

L = Error between prediction and target

2) $\boxed{w = \arg\min_{w} L}$ → Find the set of weights that minimize the loss L.

⇓

How do we achieve this?

↳ • we call our old friend calculus!

$\frac{\partial L}{\partial w} = 0$, solve for w

Maximizing / minimizing a function.

Gradient Descent (GD) → Sometimes, it is NOT possible to solve for w analytically.

→ We MUST use an iterative method such as GD to solve for w.

-> GD basics:

Repeat until convergence:

$$w \leftarrow w - \eta \nabla_w L$$

-> Basic Idea: Find the gradient of L w.r.t to w and then iteratively take steps in that direction over & over again until the loss L converges to its minimum value.

-> we call η the learning rate -> Defines the step-size of GD.

- Descent vs. Ascent
  - Maximizing $-L$ is the same as minimizing L!
  - GD -> $w \leftarrow w - \eta \nabla_w L$
  - GA -> $w \leftarrow w + \eta \nabla_w J$ where $J = -L$

Convenience of use is problem-dependent!

- Example:

  -> Consider: $L = x^2$ $J = -L = -x^2$

  Then: GD $\Rightarrow x \leftarrow x - \eta \frac{d(x^2)}{dx} = x - \eta(2x)$

  GA $\Rightarrow x \leftarrow x + \eta \frac{d(-x^2)}{dx} = x - \eta(2x)$

  $x + (-2x)$

- Not only yield the same answer -> update eqns are exactly the same!

Final common mistake

↳ Assuming BPP only applies to ANN (Feed forward models) -> Applies to many DL models. ↓

-> CNN's, RNN's, etc... -> No matter how advanced you get! ✓

How to brace yourself to learn Backpropagation (video 30)

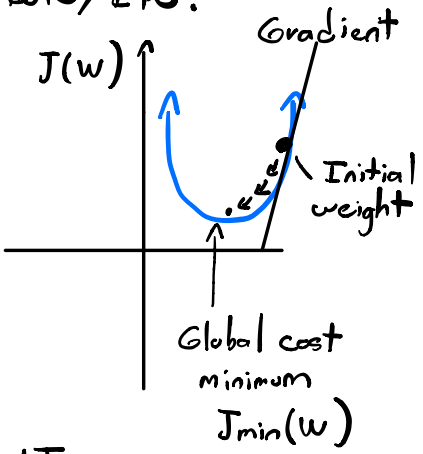
- why is this necessary? ->
  - Many people took calculus a long time ago -> (can be learned!)
  - Never took calculus -> EVEN HARDER
- Real problem -> How to distinguish the important & unimportant aspects of BPP?
  - "unimportant" -> Calculus + Derivation (Though understanding it will improve your understanding)
    - (Trees)
  - Important -> Implementation vs Forgetting WHY we are DOING the derivation is worse than NOT getting it!
    - (Forest)

Don't get lost here!

- The important part of BPP
  - Realize is actually nothing new
  - Exactly the same steps as GD -> LoR/LR!
  - Recall the following steps:

    1) Set up a cost function, J
    2) Take the gradient of J w.r.t to the weights w:
       - while not converged:

$$w = w - \text{learning\_rate} * \frac{dJ}{dw}$$



The hard part

- Most people have trouble calculating the gradient itself -> Just calculus! - Pure mechanical calculation.
- Required -> Follow the rules of calculus consistently!

NOT MAGIC! -> Not even an "algorithm"

Ex. $d(x^2)/dx = 2x$

Except instead of x^2 we have a more complicated function!

- Other Resources → No derivation / Just answer
  - ⇓ FLAWED ⇓ See Video 38 (Next) for further discussion.
  - ⇓ Seems magical to a beginner! ⇓ No sense in "building" up the answer if you don't know where it came from!
- what if you get lost?
  - ↳ Proceed & implement in code! → only addition & multiplication required to implement!
    - update equations are as follows:

$$\frac{\partial J}{\partial V_{nk}} = \sum_n (t_{nk} - y_{nk}) z_{nm}$$

$$\frac{\partial J}{\partial W_{dm}} = \sum_n \sum_k (t_{nk} - y_{nk}) V_{nk} z_{nm} (1 - z_{nm}) x_{nd}$$

↳ Assuming Sigmoid Activation function

Breaking Down the steps

1) Define the cost J
2) Take the gradient of J w.r.t W → This will give us the expression to update W.
   - ↳ Good enough: Know why we do this NOT How it is done. → Come later for further understanding.
3) Implement (2) in code.
   - ↳ DO NOT GIVE UP! → GIVE IT A TRY!

Summary

↳ • Implementation is the most important hands-on exercise.

• Not knowing the derivation -> No excuse for not doing implementation.

• Derivatios is nice -> we understand where the eqn's came from! -> Not essential for writing code!

• The Wrong Way to learn BPP: (Video 38)

↳ Further discussion of How to Brace Yourself To Learn BPP (video 30) -> we skip to this video such that we don't break our continuity on the actual BPP lectures!

• Wrong approach to learn BPP

↳ • BPP ≠ Magical algorithm -> only requires the chain-rule!

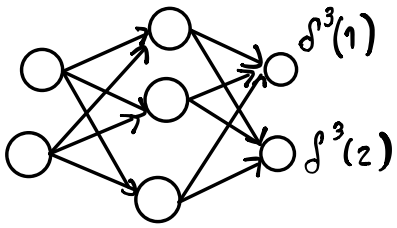
↓

Typical approach

$\delta^3(k) = y(k) - t(k)$

= "Error at node"

↳ why is this the error?



⇓

Not very magical!

All math!

↳ why not some other expression! → Not very well defined!

$$\Delta V(K) = z(j)\,\delta^3(K)$$

$$\Rightarrow V(j,K) = V(j,K) - \eta\,\Delta V(j,K)$$

why? → we cannot just make this up because it makes "sense" when you put it in words.

Actual BPP

↳ $$\delta^2(j) = \sum_K [y(K) - t(K)]\, V(j,K)$$

$$= \sum_K \delta^{(3)}(K)\, V(j,K)$$ → We show that the error propagates backwards

↑

why is the error? why not some other expression?

$$\Rightarrow \Delta W(i,j) = x(i)\,\delta^2(j)\,\Delta z(j)$$

Then, we say: $W(i,j) = W(i,j) - \eta\, W(i,j)$

why?

- Then → Typical approach → Flawed!

Basic facts:

- We want to maximize likelihood / Minimize $-L$ (L)
- Use GD → General all purpose optimization method for ANY function!

- "delta" -> useful -> Defines BPP recursively

  Each layer's update depends ONLY on the layer in front! ← what is the pattern?

Summary -> WRONG -> Define the "deltas" for no reason.

RIGHT -> Be able to take the derivative.

-> Doing it clearly enough to understand the pattern that emerges from it.

-> Make substitutions -> Makes the pattern more clear!

Categorical Cross-Entropy Loss (Video 31)

Function

- In this lecture -> we are going to discuss the cost that we want to minimize in our NN.

- Let's start by recalling the simplest cost function -> The squared error:

(Illustration) $L = \sum_{n=1}^{N} (\text{target}_n - \text{prediction}_n)^2$

$$L = \sum_{n=1}^{N} (t_n - y_n)^2$$ → The more different y_n given t_n, the more larger the error will be.

Makes sense to minimize this error in order to train our model!

↓

Side note: This is why we have to learn how to make predictions first.

↳ Error → Discrepancy b/w target & predictions

we must first have the predictions.

Recall

- Minimizing the squared error is the same as maximizing the log-likelihood given a Gaussian-Distributed error:

  ↗ Arbitrary & $\mu_n = y_n$

$$t_n \sim N(y_n, \sigma^2)$$

$$L = \prod_{n=1}^{N} \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(\frac{-1}{2\sigma^2} \cdot (t_n - y_n)^2 \right)$$

MLE function is the product of each individual Gaussian p.d.f where t_n: Random variable, $\mu_n = y_n$ & $\sigma^2 = \sigma^2$

Log-likelihood Equivalence

- More advanced than a typical MLE problem, because y itself is not meant to be optimized;

$$y_n = f(x_n; w)$$

- Instead, it is a function of some weights and we want to optimize w:

$$w = \underset{w}{\text{argmax}}\ L \quad \text{where } L \to \text{MLE (NOT Loss)}$$

- For MLE problems, it's usually more convenient to maximize the log of L rather than L itself.

- Given that $\log(x)$ is monotonically increasing they yield the same answer:

$$\log(L) = \sum_{n=1}^{N} \log\left(\frac{1}{\sqrt{2\pi\sigma}^2} \exp\left(\frac{-1}{2\sigma^2} \cdot (t_n - y_n)^2\right)\right)$$

Given $\log(\exp(x)) = x$, we have:

$$\log(L) = \sum_{n=1}^{N} \left[\log\frac{1}{\sqrt{2\pi\sigma^2}} + \left(-\frac{1}{2\sigma^2}(t_n - y_n)^2\right)\right]$$

$$\log(L) = C_1 - C_2 \sum_{n=1}^{N} (t_n - y_n)^2$$

- Because the value of W that minimizes L does not change, we may neglect the constants C_1 & C_2:
  - ↳ Thus: Maximizing $L \equiv$ Minimizing Squared error!

Binary classification

- A slightly more complicated scenario → we use the binary cross-entropy (BCE) cost function instead of Squared error:

$$L = -\sum_{i=1}^{N} \{ t_n \log y_n + (1 - t_n) \log (1 - y_n) \}$$

- Now, what is the Log-likelihood equivalence of the BCE
- MLE equivalent → Target is a Bernoulli distributed random variable → "Probability of "success" as prediction (e.g. A coin flip where p(heads) = output prediction)

$$t_n \sim \text{Bernoulli}(y_n)$$

↓

Log-Likelihood Equivalence

$$L = \prod_{n=1}^{N} y_n^{t_n} (1 - y_n)^{(1 - t_n)}$$

↳ Exercise: Prove the BCE Loss function / Negative L

$$\log(L) = \log\left(\prod_{n=1}^{N} y_n^{t_n}(1-y_n)^{(1-t_n)}\right)$$

$$= \log\left(\sum_{n=1} y_n^{t_n} + (1-y_n)^{(1-t_n)}\right)$$

$$= \sum_{n=1}^{N} t_n \log(y_n) + (1-t_n)\log(1-y_n)$$

Thus, the negative log-likelihood (cost) - log(L)

$$L = -\sum_{i=1}^{N} \{t_n \log y_n + (1-t_n)\log(1-y_n)\}$$

- we then see that in both LR & BC, the loss function equals the negative log-likelihood.

- Multiclass classification
  - Bernoulli distribution -> 2 choices (Analogy is a coin toss)
  - Categorical distribution -> ≥ 2 choices (Analogy is a die roll)

⇑ For multiclass classification, the random variable comes from a categorical distribution instead!

MLE for a die roll -> • MLE for a possibly biased die.

- Suppose we roll a few times → we get something like: $\{1, 1, 6, 3, 2, 4, 5, 3\}$
- It's convenient to represent this data with an indicator matrix → $t(n,k) = 1$ if we rolled k on the n^{th} roll (0 otherwise)

- In this case, there are no "model predictions", were just trying to find the probability of rolling each number $1, \ldots, 6$
  - ↳ Denote as $w_1, w_2, \cdots w_6$
- We write our likelihood as follows:

$$L = \prod_{n=1}^{N} \prod_{k=1}^{6} w_k^{t_{nk}}$$ → PMF for the

$$L = \prod_{n=1}^{N} \left(w_1^{t_{n1}} \cdot \underbrace{w_2^{t_{n2}} \cdots w_6^{t_{n6}}} \right)$$

As usual, the likelihood is the product of P.M.F's for all N data points.

Note that most values will be $w_k^{(0)} = 1$

↳ Recall that a PMF

Probability mass Function: Is the probability distribution of a discrete random variable, and provides the possible values and their associated probabilities.

- A PMF:

  Its a function: $\mathbb{R} \mapsto [0, 1]$ and it is defined by:

  $$P_X(i) = P(X = i)$$

Back to Neural Nets

- Now, we follow the same pattern from Regression and BC.
- Make the probability of each category the NN output:

$$L = \prod_{n=1}^{N} \prod_{k=1}^{K} y_{nk}^{t_{nk}}$$

→ y_{nk} → Tells us the predictions for targets t_{nk}

↑ → If y_{nk} is large → t_{nk} likely to be large.

As usual, our loss is the "negative of the log-likelihood"

↓ → Gives us our cost function:

$$C.C.E = -\sum_{n=1}^{N} \sum_{k=1}^{K} t_{nk} \log(y_{nk})$$

(Categorical-cross Entropy)

Does it work?

↳ Let's build some intuition on our Loss function L:

- If $y(n,k)$ is more wrong → we want the loss to be larger.
- If $y(n,k)$ is more right → we want the loss to be smaller

- Consider a single sample:

$$\text{Loss} = -\sum_{k=1}^{K} t_k \log(y_k)$$

↳ • Exactly right: $-1 \log(1) = 0$

It works!

- 50% chance of being right: $-1 \log(0.5) = 0.693$
- 25% chance of being right: $-1 \log(0.25) = 1.386$
- 0% chance of being right: $-1 \log(0) = \text{too}$

Recap

- Step #1: Define our Loss (we just did)
- Step #2: Minimize loss w.r.t to NN weights (up next)

↳ Must use GD → No closed form solution!

- Next lectures / steps → How to find the gradients?