MAX LIKELIHOOD

- finding θ s.t. it maximizes the probability of obtaining the dataset

Bernoulli: simplest RVs → can only take on 0 or 1 eg, coin flip

$$P(Y_1 = y_1, \dots, Y_m = y_m \mid \theta) = \prod_{i=1}^{m} P(Y_i = y_i \mid \theta) = \prod_{i=1}^{m} \theta^{y_i}(1-\theta)^{1-y_i}$$ indep of trials

MLE Estimate for θ: $\theta = \frac{1}{m}\sum_{i=1}^{m} y_i$ (arithmetic mean)

Gaussian Noise $y^{(i)} = \theta^T x^{(i)} + \varepsilon^{(i)}$, $\varepsilon^{(i)} \sim N(0, \sigma^2)$ x is data from dataset

Predict the probability of a certain $y^{(i)}$ given a fixed θ

$$P(y^{(i)} \mid \theta, x^{(i)}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$

& then compute likelihood for each pair of points (x_i, y_i) in training set:

$$P(y^{(i)} \mid \theta, x^{(i)}) = \frac{1}{\sqrt{2\pi\sigma^2}} \prod_{i=1}^{m} \exp\left(-\frac{(y^{(i)} - \theta^T x^{(i)})^2}{2\sigma^2}\right)$$ (ie, just multiply it all together)

MLE for Gaussian Noise: $\theta = \operatorname{argmin}_\theta \sum_{i=1}^{m} (y^{(i)} - \theta^T x^{(i)})^2$

Gradient Descent with MLE + Bernoulli Variables torch_coin_manual.py

- used esp when we can't solve for argmax mathematically

theta = torch.tensor(value, requires_grad = True)

loop:

set the loss as $P(y \mid \theta)$ or $\lg P(y \mid \theta)$

loss.backward()

update θ by: theta = torch.tensor(theta + alpha * theta.grad, requires_grad = True)

↳ LR

BAYESIAN INFERENCE

$$P(A|B) = \frac{P(B|A)\,P(A)}{P(B)}$$

Estimating $P(\theta=\theta_0 \mid data)$ using a Prior

$$P(\theta=\theta_0 \mid data) = \frac{P(data \mid \theta=\theta_0)\,P(\theta=\theta_0)}{P(data)} = \frac{P(data \mid \theta=\theta_0)\,\overbrace{P(\theta=\theta_0)}^{\text{prior}}}{\sum_{\theta_i} P(data \mid \theta=\theta_i)\,P(\theta=\theta_i)}$$

} sum/integrate over all possible values of θ

Maximizing this (Maximum A-Posterior → MAP)

$$\underset{\theta}{\operatorname{argmax}} \frac{P(data \mid \theta=\theta_0)\,P(\theta=\theta_0)}{P(data)} = \underset{\theta}{\operatorname{argmax}}\; P(data \mid \theta=\theta_0)\,P(\theta=\theta_0)$$

$$= \underset{\theta}{\operatorname{argmax}}\; \underbrace{\lg P(\cdots)}_{\text{DATA LIKELIHOOD}} + \underbrace{\lg P(\cdots)}_{\text{PRIOR}}$$

↓ to update prior belief

Gradient Descent with MAP + Bernoulli Variables coin_bayes.py

initialize prior (stats.norm.pdf or smth) normalize

likelihood = ... using $P(y^{(i)} \mid \theta)$ expression

posterior = likelihood * prior normalize

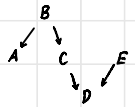
& then I assume it's same idea as MLE except loss = posterior or log-posterior

What does P+T #3 mean?

GENERATIVE MODELS

CAUSAL INFERENCE

DAGs



⇒ B might have caused A

⇒ A did not cause B

⇒ E is definitely indep. of B

Causation: how to define "A caused B"

- Counterfactual: A caused B if B would not have happened if A had not happened
  - ↳ difficult to determine in real life
- Causal Inference: using empirical data to determine causality

do(···) operator

- do(A) eliminates all arrows pointing to A
- basically says we'll force A=1 regardless, so any potential causal mechanisms that may cause A = 0/1 are irrelevant

$P(D \mid C=c) \neq P(D \mid do(C=c))$

- implement do(·) with control trials
  - ↳ hold all other vars constant
  - ↳ randomize all other vars constant

Identification

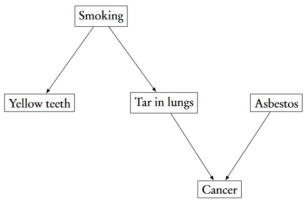
- Want to calculate the causal effect of X on Y (i.e., $P(Y|do(X = x))$, but can't run an experiment.
- Can do this if we have the causal graph and observe all the variables
  - Saw this in the toothbrushing example
- Can *sometimes* do this if not all variables are observed
  - Need to carefully look at the graph

???

also causal.py

explain_away.py ?

CAUSALITY CODE / EXPLAINING AWAY




```
# P(smoking = 1) = 0.2
# P(yellow = 1) = logistic(smoking - 1)
# P(tar = 1) = logistic(2*smoking - 1)
# P(asbestos = 1) = 0.1
# P(cancer = 1) = logistic(5 * tar + 5 * asbestos - 1)

# Compute P(yellow = 1 | asbestos = 1)
# Compute P(yellow = 1 | asbestos = 1, cancer = 1)

# Compute P(yellow = 1| asbestos = 1, do(cancer = 1))

smoke_yellow_coef = 1
cancer_asbestos_coef = 0.5
cancer_tar_coef = 5
smoke_prob = 0.2
N = 100


print(np.random.binomial(n = 1, p = 0.1, size = N))
# = np.random.binomial(n = 1, p = 0.1, size = N)
smoking = np.random.binomial(n = 1, p = smoke_prob, size = N)

yellow_prob = 1 / (1 + np.exp(-smoking * smoke_yellow_coef + 1))
yellow = np.zeros(smoking.shape)
for i in range(smoking.shape[0]):
    yellow[i] = np.random.binomial(n = 1, p = yellow_prob[i], size = 1)

# P(yellow = 1)
print("P(yellow = 1) =", np.mean(yellow))

# Compute P(yellow = 1 | asbestos = 1)
asbestos = np.random.binomial(n = 1, p = 0.1, size = N)
yellow_asbestos = yellow[asbestos == 1]
print("P(yellow = 1|absbestos=1) =", np.mean(yellow_asbestos))



# Compute P(yellow = 1 | asbestos = 1, cancer = 1)
p_tar = 1 / (1 + np.exp(-2 * smoking + 1))
tar = np.zeros(smoking.shape)
for i in range(tar.shape[0]):
    tar[i] = np.random.binomial(n = 1, p = p_tar[i], size = 1)

p_cancer = 1 / (1 + np.exp(-cancer_tar_coef * tar - cancer_asbestos_coef * asbestos + 1))
cancer = np.zeros(smoking.shape)
for i in range(cancer.shape[0]):
    cancer[i] = np.random.binomial(n = 1, p = p_cancer[i], size = 1)
yellow_asbestos_cancer = yellow[(asbestos == 1) & (cancer == 1)]
print("P(yellow = 1|asbestos = 1, cancer = 1) =", np.mean(yellow_asbestos_cancer))

# Compute P(yellow = 1| asbestos = 1, do(cancer = 1))
print("P(yellow = 1|asbestos = 1, do(cancer = 1)) = ", np.mean(yellow_asbestos))
```

sigmoid function

So in code:

$P(X=1 \mid Y=1, Z=1)$:

`np.mean(X[Y==1 & Z==1])`

$P(X=1 \mid Y=1, do(Z=1))$:

`np.mean(X[Y==1])`

```
[15] # Explaining away
     #
     # A    C
     #  |  /
     #   B
     #
     # A and C are independent, but B explains away the correlation between A and C.

     import numpy as np
     N = 10000
     np.random.seed(0)
     A = np.random.binomial(n = 1, p = 0.2, size = N)
     C = np.random.binomial(n = 1, p = 0.2, size = N)
     B = np.zeros_like(A)

     for i in range(A.shape[0]):
         B[i] = np.random.binomial(n = 1, p = 0.5 * A[i] + 0.5 * C[i], size = 1)

     print("P(A = 1) = ", np.mean(A))
     print("P(C = 1) = ", np.mean(C))
     print("P(B = 1) = ", np.mean(B))
     print("P(A = 1 | C = 1) = ", np.mean(A[C == 1]))
     print("P(A = 1 | C = 1, B = 1) = ", np.mean(A[(B == 1) & (C == 1)]))
     print("P(A = 1 | B = 1) = ", np.mean(A[(B == 1) ]))

     P(A = 1) =  0.1972
     P(C = 1) =  0.2014
     P(B = 1) =  0.1994
     P(A = 1 | C = 1) =  0.198609731876861 96
     P(A = 1 | C = 1, B = 1) =  0.3336113427856547
     P(A = 1 | B = 1) =  0.5992978936810431
```

Explaining Away:

↳ when the occurence of an event decreases the prob. of another event that was previously considered likely

FAIRNESS + CAUSALITY + ML

already have some notes
but also need 2nd set

CAUSAL REPRESENTATION ML

Issues with Usual Supervised Learning Setting data: (x_i, y_i) → want to learn $P(y|x)$

- **Robustness:** issues if $P(x,y)$ distribution is different in test vs. training set
- **Learning reusable mechanisms:** real-life behaviours / mechanisms
  - ↳ eg, objects fall → want to be able to learn this for all objects with few examples
- **Being able to predict outcomes of counterfactual scenarios:** eg, $P(\text{rain} \mid \text{umbrellas})$ is high in training set, but want to be able to do $P(\text{rain} \mid do(\text{umbrellas}))$

Types of Models

- **Physical Model:** eg, $\frac{dx}{dt} = f(x)$, earlier x's cause later x's
- **Statistical Model:** model relationships b/w input + output data, without worrying about underlying processes
  - ↳ an approximation of $P(x,y)$
- **Structural Causal:** a DAG that encodes causal relationships; eg, A→B in the DAG would actually mean A causes B
- **Causal Graphical:** a DAG that doesn't necessarily encode causality; eg, A→B doesn't necessarily mean a causal relationship
  - ↳ the probability is just represented as a DAG

structural causal vs. causal graphical?
↳ didn't do the rest of the slides from here

Difference between SCM and a causal graphical model

confused about this slide
↳ which part is SCM & which is GCM?

- A: altitude
- T: temperature
- P(A, T) = P(A|T)P(T) = P(T|A)P(A)
- P(A, T) might be different for Austria and Switzerland, but P(T|A) might be the same
- The SCM P(T|A)P(A) encodes the mechanism of generating the temperature from the altitude that can generalize across countries

} could be either one & the math "technically" works out
↳ but only the "true" one will actually be predictive

maybe watch the lecture about these slides

* also a bunch of slides I haven't looked at yet → causal.py, explain_away.py

WORD EMBEDDINGS

- one-hot encodings can't represent relationships b/w words (eg, rhyming, similar meaning, etc)
  ⇒ we want similar words to have similar representations!

- instead → words' meanings are closely related to which words appear closeby → linguistics distributional hypothesis

- use "context" (nearby words) of w to build up representation of w
  ↳ each word is n-dim vector
- use cos similarity $\cos\theta_{u,v} = \frac{u \cdot v}{\|u\|\|v\|}$ or alternatively just $u \cdot v$ (if all vectors have roughly same magnitude)

Word2Vec

- each word ⇒ n-dim vector

for each position t in text:
  get pairs of centre words c & context words o
  compute $P(o|c)$ as fcn of similarity
  maximize $P(o|c)$

$$P(o|c) = \frac{\exp(u_o \cdot v_c)}{\sum_{w \in V} \exp(u_w \cdot v_c)}$$

↑ similarity = ↑ probability

↳ this is using softmax function

↳ * the derivative is expensive to compute (have to sum over whole vocab in denom!)
  ⇒ expensive to find maximum

Likelihood: $L(\theta) = \prod_{t=1}^{T} \prod_{\substack{-m \le j \le m \\ j \neq 0}} P(w_{t+j} \mid w_t; \theta)$ m is training context window

Negative Sampling

way to improve expensive computation for word2vec!

- goal is still to maximize similarities of the words in same context & minimize pairs of words in different contexts
- but rather than minimizing across all words in a vocab, it randomly selects k words to minimize (↑ k for ↓ datasets)

$$J_t(\theta) = \underbrace{\sum_c \log \sigma(u_c \cdot v_t)}_{\substack{\text{maximize probability that} \\ \text{the real context words} \\ \text{appear near centre word } t}} + \underbrace{\sum_{i=1}^{k} E_{j \sim P(w)} \left[\log \sigma(-u_j \cdot v_t)\right]}_{\substack{\text{minimize (that's why -ve) prob.} \\ \text{that random outside words} \\ \text{appear near centre word } t}}$$

- for position t
- $P(w)$ is noise distribution where the negative samples are drawn from

GloVe → based on creating a co-occurrence matrix between words → expensive, but a one-time cost

- advantage over word2vec: keeps track of global co-occurrence rather than just local context

- Cost Function: $J = \sum_{i,j} f(X_{ij}) (u_i \cdot v_j + b_i + b'_j - \lg X_{ij})^2$

  X_{ij}: # of times w_i & w_j co-occur

  X_i: # of times w_i occurs

- $u_i \cdot v_j \approx \lg X_{ij} - \lg X_i \approx \lg P(w_i \mid w_j)$ $\left.\frac{X_{ij}}{X_i}\right\}$ probability of word j appearing whenever word i does

  ↳ absorb $\lg X_i$ into the biases to make the expression symmetric

  ⇒ that's where $u_i \cdot v_k + b_i \cdot b'_k \approx \lg(X_{ik})$ from

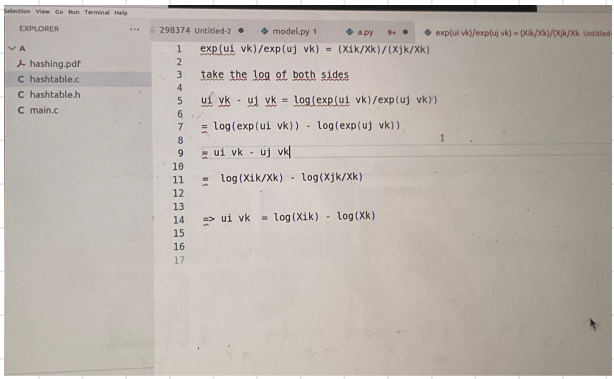
- Ideas: ratios of co-occurrence probabilities can encode meaningful components → ie, $P(x|y)$ will be relatively large if x & y related

- what the ratios mean:
  - $\frac{P_{ik}}{P_{jk}} > 1$: k word is similar to i but irrelevant to j
  - $\frac{P_{ik}}{P_{jk}} < 1$: k " " " " j " " " i
  - $\frac{P_{ik}}{P_{jk}} = 1$: k is about same amount of relatedness to i & j (whether k is related or unrelated to them)

- if we set $f(\cdot) = \exp(\cdot)$, we get $\frac{\exp(u_i \cdot v_k)}{\exp(u_j \cdot v_k)} \approx \frac{X_{ik}/X_k}{X_{jk}/X_k}$

  ↳ this is because we want $f((u_i - u_j) \cdot v_k) \approx \frac{P(i|k)}{P(j|k)}$

$$\frac{\exp(u_i \cdot v_k)}{\exp(u_j \cdot v_k)} = \frac{(X_{ik}/X_k)}{(X_{jk}/X_k)}$$



RNNs

- "unrolled left-to-right"
  - ↳ this encodes local linearity → nearby words often affect each other's meaning hmm?

problem: linear interaction distance

- ↳ RNNs take O(seq. len.) for distant word pairs to interact → have to go through that # of layers!
- ↳ ∴ hard to learn long-distance dependencies (vanishing gradient problems by the time you get there)
- ↳ words are stuck in a fixed order

problem: lack of parallelizability (no GPU parallelization)

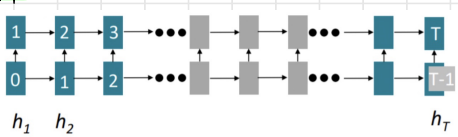
- ↳ forward & backward passes both have O(seq. len.) unparallelizable operations → bc you have to compute states sequentially
- ↳ but note that you can process multiple sequences at a time

Alternative: word windows → like a 1D convolution

- ↳ aggregate local context
- ↳ ∴ # of unparallelizable operations does not increase with seq. len. (bc size of conv. window is the same each time)
- ↳ stacking word-window layers allows interactions b/w words that are further apart → this is like how we increase receptive field with ConvNets!

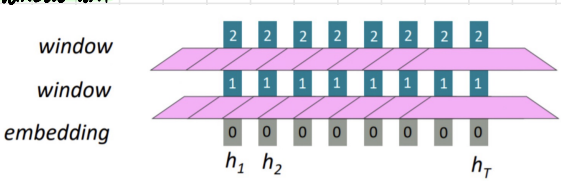
$$\text{max interaction distances} = \frac{\text{seq. len.}}{\text{window size}}$$

OLD RNN WAY



Numbers indicate min # of steps before a state can be computed

NEW WINDOW WAY



⇒ way fewer steps! way more parallelizable!

Numbers indicate min # of steps before a state can be computed

OR use attention!

TRANSFORMERS → encoder/decoder models

Encoders

- all encoder layers are identical in structure but do not share weights
- each encoder has: 1) self-attention → helps encoder look at other words in the input seq. as it encodes a specific word
  2) feed-forward NN → same FFNN is indep. to each position
  → having the FFNN adds a non-linearity

Decoders

- each decoder has: 1) self-attention
  2) encoder-decoder attention → helps decoder focus on relevant parts of input seq.
  3) feed-forward NN

Embedding

- turn each word into an embedding vector
- do this once for each word → only in the bottom-most encoder
  ↳ ie, bottom encoder receives the word embeddings
  all other encoders receive the output from the row below it
  so each encoder would receive list of vectors that are each size 512
  ↳ size of this list is the length of the longest sentence

Self-Attention

1) for each word, create Query, Key, Value vector use $q_i = v_i = k_i = x_i$ for layer x } are these contradictions?
   ↳ each vector is created by multiplying each input word by a weight matrix: W^Q, W^K, W^V

2) calculate self-attention → this is just dot product ⇒ each q gets dotted with all k's: $q_1 \cdot k_1$, $q_1 \cdot k_2$, etc

3) divide scores by 8 → leads to more stable gradients

4) softmax → makes all scores +ve; sum to 1

$$\alpha_{ij} = \frac{\exp(e_{ij})}{\sum_j \exp(e_{ij})}$$

5) multiply each v by its softmax score & sum output $= \sum_j \alpha_{ij} v_j$

⇒ self-attention *does not* know the order of its inputs → this is why we'll use positional encoding later on

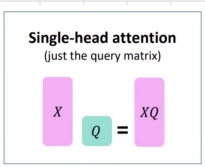
Summary of steps: $z = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V$ $\quad d_k = \frac{\text{dimensionality}}{\#\text{ of heads}}$

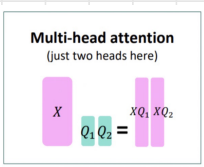
Multi-Headed Attention

- look at multiple places in the sentence at once
  - ↳ for word i, self-attention looks where $x_i^T Q^T K x_j$ is high ⇒ look at different j for different reasons

steps

1) have different Q, K, V for each head $\quad Q_l, K_l, V_l$ all have dims $(d, d/\text{num\_heads})$
   - ↳ each head l will have $\text{output}_l = \text{softmax}(XQ_l K_l^T X^T)XV$ → output_l dim $(d, d/\text{num\_heads})$
2) then all outputs are concatenated together dim (d, d) & multiplied by a weight matrix W_0 (?)
   - is this the same as the FeedForward
3) concatenated output can be sent to FFNN





Same amount of computation as single-head self-attention!

Positional Encoding → addresses issue of self-attention not knowing the order

- add a positional encoding vector to the input vector that's passed into the first encoder
- the positional encodings are concatenated sinusoidal functions of diff periods { why does this work?

$$P_i = \begin{bmatrix} \sin(\ldots) \\ \cos(\ldots) \\ \vdots \\ \sin(\ldots) \\ \cos(\ldots) \end{bmatrix}$$

-

? I don't understand what this slide is saying

Position representation vectors learned from scratch

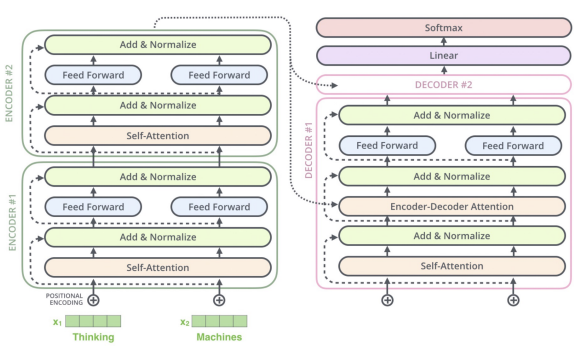
- Learned absolute position representations: Let all p_i be learnable parameters
  - Learn a matrix $p \in R^{d \times T}$, and let each p_i be a column of that matrix
- Pro: each position gets to be learned to fit the data
- Con: cannot extrapolate outside of 1, …, T
- Most systems use this

13

Masking → ★ for decoders only ★

- set attention scores for future words by setting them to $-\infty$? intuition for $-\infty$? vs. $+\infty$?

Residuals: add to make the loss landscape smoother



for decoder

Summary of Differences for Decoders

↳ self-attention uses masking

↳ encoder-decoder attention takes Queries from below, and takes Keys & Queries from the output of the encoder stack

Cross-Attention (aka Encoder-Decoder Attention) ↑

Transformer Time Complexity: quadratic w/ sequence length matrix multiplication to speed up?

↳ RNNs are only linear

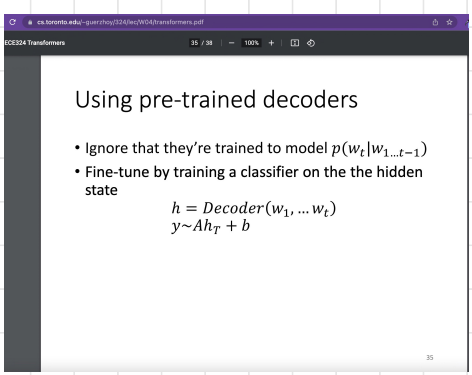
Byte-Pair Encoding / Subword Tokenization

- helps for languages where finite vocab. assumptions don't make sense
  - ↳ eg, Swahili → has 100s of verb conjugations
- to define subword vocab:
  1) start w/ vocab using only characters & EOW (= end of word) symbol
  2) find most common adjacent characters a,b
     - ↳ add "ab" to the vocab
  3) replace instances of the character pair with the new subwords
  4) repeat 2&3 until desired vocab size
- then common words get to be part of the vocab; rarer words are still split into components

Pre-training Encoders → mask out words; make model predict them. masks 80% of the time?

Pre-training Decoders model $p(w_t \mid w_{1:t-1})$ (next word prediction); train generally on lots of text

↳ When fine-tuning, train a classifier on the hidden state → classifier head



⇒ confused (also abt pretraining / finetuning in general)

Understanding: Humans vs. AI

- humans: understand meaning; communicative intent
- AI: only understands form; aka statistical likelihood of a word being next

Octopus Test

⇒ does AI know how words map to real world entities → this is the octopus test

↳ or think of a model trained to understand code, but never given inputs, outputs, a compiler, etc

Occam's razor / octopus test

pretraining stuff

very few GANs q's → is that indicative of the exam?

why don't we want the decoder of an autoencoder to be fully connected?

INTRO TO GANS

Autoencoders

steps

1) encodes smth to a template
2) decoder reconstructs from template

- multiply by weight matrices
- squared diff. loss

problem: - loss of info from bottleneck layer → blurry
- having so many templates is bad for large images

⇒ sol'n: convolutions!

↳ fractionally strided convolutions to enlarge

Basic Idea of Generators

- sample noise $z \sim N(0, I)$
- run z through decoder → makes random image

GANs → train 2 adversarial networks

Generator: fool discrim. by creating real-looking images

Discrim: tell apart real vs. fake images

Minmax

$$\min_{\theta_g} \max_{\theta_d} \left[E_{x \sim p_{data}} \left[\log D_{\theta_d}(x) \right] + E_{z \sim p(z)} \left[1 - D_{\theta_d}(G_{\theta_g}(z)) \right] \right]$$

- $\min_{\theta_g}$: generator wants to minimize → gradient descent
- $\max_{\theta_d}$: discrim wants to change its parameters to max getting it right → gradient ascent
- $E_{x \sim p_{data}} [\log D_{\theta_d}(x)]$: when sampling from training data, the discrim wants to output 1 (for real)
- $E_{z \sim p(z)} [1 - D_{\theta_d}(G_{\theta_g}(z))]$: when sampling from generator, the discrim wants to output 0 (for fake) → generator wants it to output large x's on fake images

problem: hard for generator to learn if discrim. learns quicker

↳ **solution**:

modify min generator part to:

$$J = E_{z \sim p(z)} \left[-\log \left(D_{\theta_d}(G_{\theta_g}(z)) \right) \right]$$

} this makes the update step larger for the generator when it's doing badly!

Update steps

(expectation / average!)

for discriminator

$$\nabla_{\theta_d} \frac{1}{m} \sum_{i=1}^{m} \left[\log D_{\theta_d}(x^{(i)}) + \log \left(1 - D_{\theta_d}(G_{\theta_g}(z^{(i)})) \right) \right]$$

} do k steps of this

for generator

$$\nabla_{\theta_g} \frac{1}{m} \sum_{i=1}^{m} -\log \left(D_{\theta_d}(G_{\theta_g}(z^{(i)})) \right)$$

} for every 1 step of this

→ can do vector arithmetic in z-space

CycleGAN translating images b/w domains w/o paired examples

- 1st gen G maps images from domain $A \to B$
  2nd gen F " " back from $B \to A$
- 1st disc. D_B tries to distinguish real images from domain B vs. fake images from gen. G
  2nd " D_A " " A " F

eqns? do we need to know?

main benefit: doesn't require 1-1 correspondence b/w images in domains A, B during training

WGANs

↳ constrains discrim. (keeps it 1-Lipschitz) → helps the generator train

↳ reformulate cost distance b/w generator dist. & data dist.

↳ could use KL divergence, JSD → we end up using Wasserstein distance

$$L = \min_{\theta_g} \max_{\theta_d} E_{x\sim p(x)}\left[\lg D_{\theta_d}(x)\right] + E_{z\sim p(z)}\left[\lg\left(1 - D_{\theta_d}(G_{\theta_g}(z))\right)\right]$$

$$J = \max -\lg\left(D_{\theta_d}(G_{\theta_g}(z))\right)$$

generator is maximized at: ⇒ find this by integrating the loss · prob dist of the variable

$$D_G^*(x) = \frac{P_{data}(x)}{P_{data}(x) + P_g(x)}$$

new cost function $C(G) = -\lg(4) + KL\left(P_{data} \,\|\, \frac{P_{data} + P_g}{2}\right) + KL\left(P_g \,\|\, \frac{P_{data} + P_g}{2}\right)$

KL-divergence $KL(P\|Q) = \int p(x) \lg \frac{p(x)}{q(x)}\, dx$

↳ ∞ when $p(i) = 0$ and $q(i) > 0$ OR $p(i) > 0$ and $q(i) = 0$

JSD $JS(P,Q) = \frac{1}{2}\left[KL\left(P \,\|\, \frac{P+Q}{2}\right) + KL\left(Q \,\|\, \frac{P+Q}{2}\right)\right]$

↳ prevents the ∞ issue bc $M = \frac{P+Q}{2}$ acts as the "support" for P & Q; M isn't 0 anywhere that P & Q are there for

```
[12] def sum_xlogy(x, y):
       prod = np.multiply(x, np.log(y))
       return np.sum(prod[np.where(x != 0)])   # don't include anywhere where prob dist is 0 in the sum (bc log(0) is undefined)


     def KL_div (P, Q):
       return sum_xlogy(P, P/Q)


     def JS_div(P, Q):
       M = 0.5 * (P+Q)
       return 0.5 * (KL_div(P, M) + KL_div(Q, M))
```

```
KL_div(P, Q)  # inf
JS_div(P, Q)  # 1.38
```

1-Lipschitz slope is max. 1 everywhere eg, $|x|$ eg of not, x^2

Wasserstein vs. JSD

↳ bc Wasserstein incorporates distance $|x-y|$, it can give a more meaningful signal even when dists are disjoint / far apart

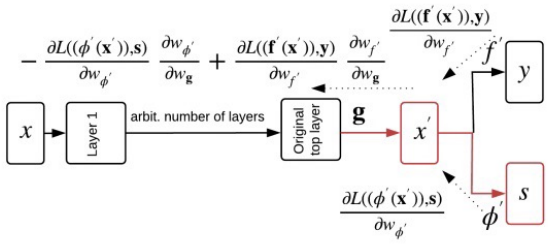
Wasserstein

$$W(P,Q) = \underbrace{\inf}_{\min} \int \underbrace{|x-y|}_{\text{distance you have to move it}} \underbrace{\gamma(x,y)}_{\text{how much mass you have to move}} dx\,dy$$

MP1

Question 1

Question 2




```
class FAD(nn.Module):
    def __init__(self):
        super(FAD, self).__init__()
        IN_FEATURES = 9
        self.layer1 = nn.Linear(IN_FEATURES, 16)
        self.layer2 = nn.Linear(16, 32)

        self.g = nn.Linear(32, 32) # additional layer that produces a fair rep of x

        # f' predicts whether or not will recidivate. phi' predicts sensitive attribute (race)

        self.f = nn.Linear(32, 1) # predicates whether or not will recidivate
        self.phi = nn.Linear(32, 2) # predicates whether person is white or black
        self.softmax = nn.Softmax()
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        out = F.relu(self.layer1(x))
        out = F.relu(self.layer2(out))
        x_prime = F.relu(self.g(out))

        y = self.f(x_prime)
        s = self.phi(x_prime)
        y = self.sigmoid(y)
        s = self.softmax(s) # pass race prediction through softmax to create a prob. distribution

        return y, s
```

Playbook:

- SGD
- Adam → uses weight decay + momentum
- model checkpoints
- learning step scheduler to decay learning rates

Using Minibatch Diversity to help w/ mode collapse mode collapse: always outputting same thing

PYTORCH

requires_grad = True param in a tensor if these are the gradients you want to compute in backwards pass

GANs