Michael Machine Learning Notes

Loss Functions

Pytorch Loss Functions
TODO

Performance Metrics

PR CURVE, ROC AND AUC

Precision and Recall
Precision measures how many positive class predictions we made are correct. For example, in a web search, we want to know “for those generated search results, how many of them are actually useful for users?”.
It is defined as TP / (TP + FP).
Recall measures for those instances that are actually positive, how many do we get it right. For example, in a web search, we want to know “for those results that are actually useful for users, how many of them are generated?”.
It is defined as TP / (TP + FN).
Precision and recall are somehow contradictory measurements. If we want a high recall, we can just predict everything to be positive, but in this case the recall will be pretty bad. Similarly, if we want a high precision, we can just be very careful and only present very few positive predictions that we are confident with, but in this case we will miss a lot of positive instances.

F1 Score
F1 Score is defined as 2 x Precision x Recall / (Precision + Recall). It is the harmonic mean of precision of recall, which we can also think as a “balance” of precision and recall. The more generalized version of F1 is F_Beta
It is also interesting to know macro and micro F1. Sometimes we perform several train/test operations and generate a unique confusion matrix for every test set. We hope to compute precision and recall on the n confusion matrices from an overall perspective.

$$\text{macro-}P = \frac{1}{n}\sum_{i=1}^{n} P_i \ ,$$

$$\text{macro-}R = \frac{1}{n}\sum_{i=1}^{n} R_i \ ,$$

$$\text{macro-}F1 = \frac{2 \times \text{macro-}P \times \text{macro-}R}{\text{macro-}P + \text{macro-}R} \ .$$

$$\text{micro-}P = \frac{\overline{TP}}{\overline{TP} + \overline{FP}} \ ,$$

$$\text{micro-}R = \frac{\overline{TP}}{\overline{TP} + \overline{FN}} \ ,$$

$$\text{micro-}F1 = \frac{2 \times \text{micro-}P \times \text{micro-}R}{\text{micro-}P + \text{micro-}R} \ .$$

Precision- Recall Curve

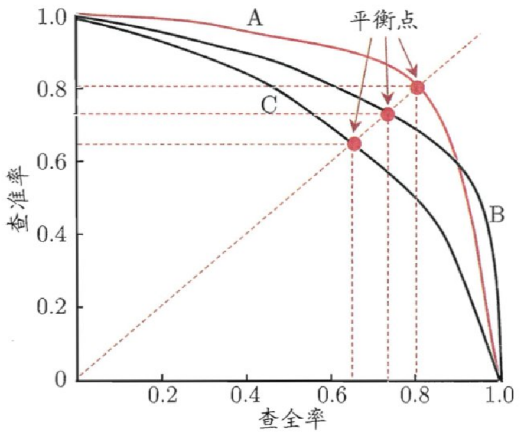
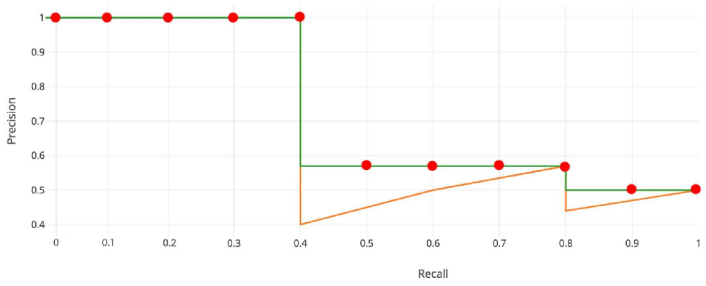


图 2.3 P-R曲线与平衡点示意图

y-axis is precision and x-axis is recall.
If one curve is completely surrounded by another, then the latter one is better. In this graph, A is better than C.
BEP (Break-Even Point) is the point where precision equals recall. In this graph, BEP (A) > BEP (B), so we *may* conclude A is better than B.
Also, area under the curve is also a good indicator on which one is better.

Average Precision (AP) and Mean Average Precision (MAP)

Average Precision (AP) represents the weighed mean of precisions achieved at each threshold. The weight is the delta of recall from last step. The general formula is AP = $\int P(r)dr$. You may notice that this is actually the area under precision-recall curve. A graphical illustration of AP calculation is shown below.



The orange line is then transformed into the green lines and the curve will decrease monotonically instead of the zigzag pattern. The intention in interpolating precision/recall curve in this way is to reduce the impact of the “wiggles” in the precision/recall curve, caused by small variations in the ranking of examples.

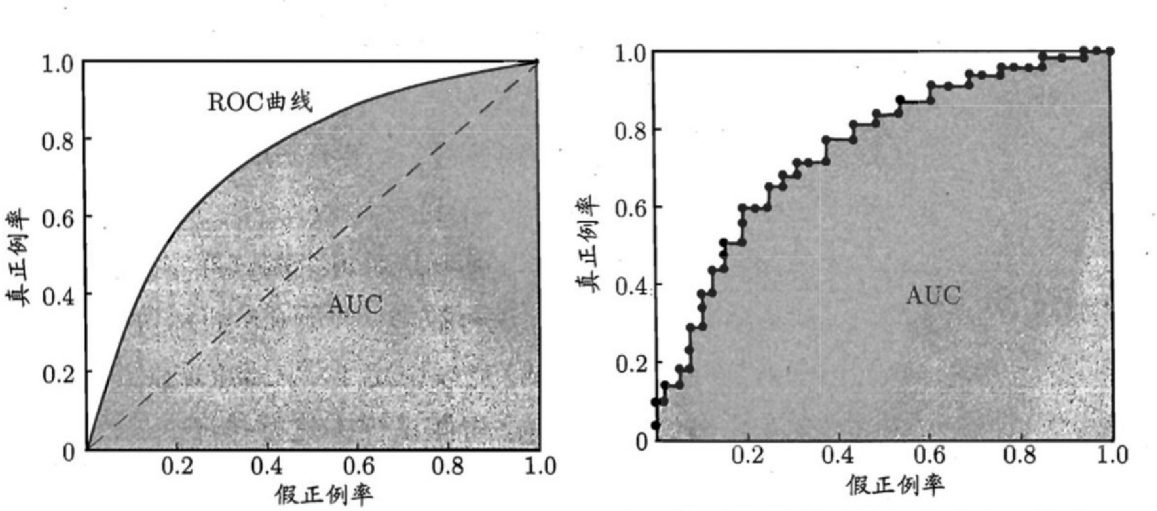
After interpolation, we divide the recall value from 0 to 1.0 into 11 points. AP is calculated as the average of the 11 points.

Mean Average Precision (MAP) is the average of AP across all classes. In other words, AP is a concept that corresponds to binary classifications.

【*】Here is a good post to summary precision, recall, AP and mAP

ROC Curve and AUC

True Positive Rate (TPR) is the same with recall: TP / (TP + FN).
False Positive Rate (FPR) measures that among those instances that are actually negative, how many of them do we wrongly predicted as positive. The formula is FP / (FP + TN).



(a) ROC 曲线与 AUC　　(b) 基于有限样例绘制的 ROC 曲线与 AUC

图 2.4　ROC 曲线与 AUC 示意图

In the left graph(a), this is the ideal ROC curve where the curve is smooth; while in the real life case, we get a curve which looks like the right graph since we have finite number of instances. In graph (b), we can see sometimes one FPR can map to multiple TPRs. The reason is that when creating the graph, we first sort the instances by confidence values and use each value as the threshold at one time. In this way, at the beginning every instance is categorized as negative and there is no false positives; as when gradually decrease the threshold, more positive predictions emerge and TPR increases. Similar to PR Curve, if one curve is completely surrounded by another, then the latter one is better; the larger the AUC, the better the model.

One interesting fact is that in graph (a), the diagonal represents the random guessing algorithm, which is often used as a baseline. Any ROC curve below this line should not be considered. In this diagonal line, TPR always equals to FPR. We can explain this by an example.

Suppose we are going to solve a fraud detection task. We have 10000 examples in total, where 3000 of them are fraud and 7000 of them are innocent (ground truth distribution). If we use random guessing as our model, then we randomly pick 3000 people as fraudulent. **Since the selection process is purely random, the distribution still holds** in the 3000 selected people, which means 30% of the 3000 selected people are fraud and 70% are not. Similarly, the distribution also holds in the 7000 left people. Now let's calculate TPR and FPR.

TP = 3000*30%, TN = 7000*70%, FP = 3000*70%, FN = 7000*30%.
TPR = TP / (TP + FN) = 0.3, FPR = FP / (FP + TN) = 0.3. Now TPR = FPR.

Now let's apply (increase) the threshold which reduces the number of predicted fraud by 20%. Now the number of positive predictions is 3000*(1 - 20%) = 2400 and number of negative predictions is 7000 + 600 = 7600. Due to randomness, we expect the two resulting sets (positive and negative predictions) still fit the ground truth distribution. Now TPR and FPR become:

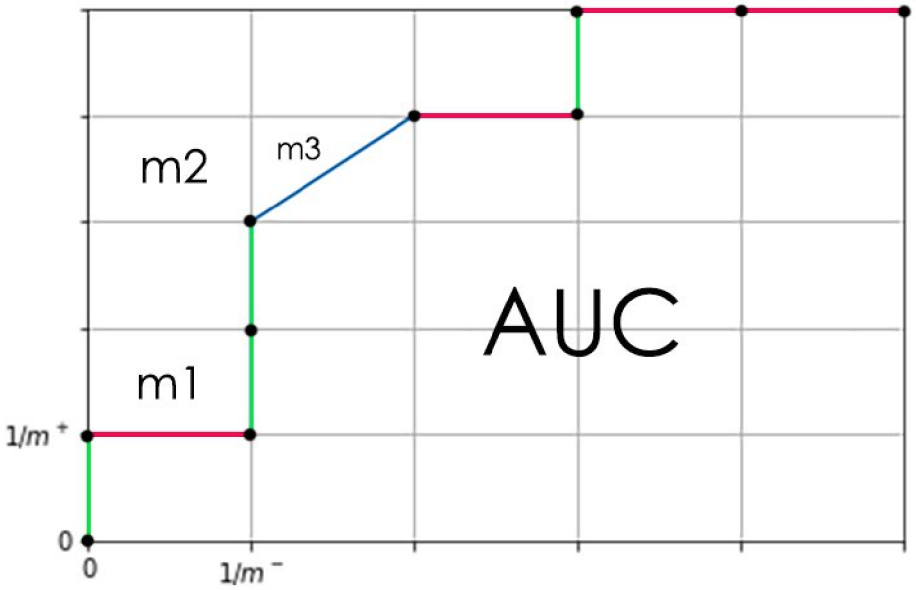
TP = 2400*30%, TN = 7600*70%, FP = 2400*70%, FN = 7600*30%.
TPR = TP / (TP + FN) = 0.24, FPR = FP / (FP + TN) = 0.24. TPR = FPR still holds.

How to calculate the AUC?
In graph (b), it is obvious that we can estimate the AUC using the formula $AUC = (1/2)*\sum(X_{n+1} - X_n)*(Y_n + Y_{n+1})$.

You may have the question that why isn't it just ∑(Xn+1 - Xn)*Yn, because graph(b) above is just a special case where all the confidence values are not identical. What if two instances have the same threshold but different ground truth? In this case, we will have neither a vertical nor a horizontal line, but a line with slope. So the formula is used to calculate the area of trapezoids. A good example is the line m3 in the graph below.
If we think about the concept of AUC more deeply, AUC is essentially evaluating the quality of ranking of confidence scores. So we can define a ranking loss as:

$$\ell_{rank} = \frac{1}{m^+ m^-} \sum_{\boldsymbol{x}^+ \in D^+} \sum_{\boldsymbol{x}^- \in D^-} \left(\mathbb{I}\left(f(\boldsymbol{x}^+) < f(\boldsymbol{x}^-)\right) + \frac{1}{2}\mathbb{I}\left(f(\boldsymbol{x}^+) = f(\boldsymbol{x}^-)\right) \right),$$

Intuitively, for each pair of positive and negative instances (Cartesian product), if the confidence of the positive instance is less than that of the negative instance, then this is undesirable and we want to include this into the ranking loss; When the scores for the pair are equal, the loss effect is half.
We can imagine the ROC Curve space as (m+ x m-) grids. L-rank essentially calculates the number of grids above the ROC curve. More explanations below.



From left to right, we gradually decrease the threshold. A green line represents the case where we generate a new true positive. Since the total number of positive instances is fixed (m+), an increase of "1" in true positives means an increase of "1/m+" in TPR. Same for FPR (red line). A blue line means that sometimes we have identical confidence values; though these instances have identical confidence values in prediction, it is highly possible that they have different ground truth values. Therefore the slope indicates an increase in both TP and FP. Each grid has an area of (1/(m+ * m-)).
Let's break down the formula like this: we extract the term ∑x+, and now the problem becomes:

1. For each green line and blue line (positive instances), how many negative instances are ***before*** it?
2. For each green line and blue line, how many instances have the same value? (This only applies to blue line itself).

Now let's calculate. We label the green lines from left to right as 1, 2, 3, 4. For green line 1, nothing is before it so the count is 0. For green line 2 and green line 3, they both have m1 before it so count = 2. The blue line is a special case where we need to break it as one positive instance and one negative instance. The positive part of the blue line can be

thought as a green line and it has m1 before it, so count = 3. Also, in the blue line there is always identical instances, so count = 3.5. For the last green line, there are 3 red lines (including the negative part of the blue line) before it, so count = 6.5. It is obvious there are 6.5 grids above the curve. And our theorem is proved.
It is obvious that AUC = 1 - L-rank.

BLEU SCORE

Modified Precision and BLEU

BLEU stands for Bilingual Evaluation Understudy and it is originally used to evaluate NMT models. For a typical generative model application in NLP, such as NMT, we may have multiple translation given an input sentence. Traditionally, we just calculate the precision of words in the generated sentence. Let's illustrate using one example.

Suppose we our two reference/ground truth sentence are "The cat is on the mat" and "There is a cat on the mat"; but our generated sentence is "the the the the the the the". If we use the tradition precision method, for each word in the generated sentence, there is at least one match in the two reference sentences, so the precision is 7/7 = 1! This is definitely a bad prediction but we get good metric score. Therefore, traditional precision doesn't work well.

A modified version of precision is then introduced. For each generated sentence, we have the following table.

	A	B	C	D
1	Word / 1-gram	Output Count	Ref Count Clip	Modified Precision
2	the	7	2	P = 2/7

In this table, output count is the number of the times one *unique* word appears in the output sentence. Ref count clip is the max number of times this word appears in a *single* reference sentence; for example, the word "the" appears twice in the first reference and appears once in the second one, so ref count clip = max(1, 2) = 2. Now our Modified precision P = 2/7. A more general formula for BLEU score is:

$$\text{BLEU-n}: \quad P_n = \sum_{\text{n-gram in output}} \frac{\text{Ref\_Count\_clip (n-gram)}}{\text{Output\_Count (n-gram)}}$$

$$\text{Combined BLEU}: \quad BP * \exp\left(\frac{1}{4}\sum_{n=1}^{4} P_n\right)$$

Brevity penalty

$$BP = \begin{cases} 1 & \text{if MT\_output\_length} > \text{reference\_output\_length} \\ \exp(1 - \text{MT\_output\_length}/\text{reference\_output\_length}) & \text{otherwise} \end{cases}$$

Here n represents n-gram. Last word-level example is a specific case of 1-gram. Referring to table above, BLEU score should be calculated as sum(C) / sum(B)

During training, generative models tend to generate shorter texts to improve precision. However, short output are not usually good. Therefore, we add a brevity penalty term to penalize short outputs.

METEOR and ROUGE

a. ROUGE: recall-oriented, how many n-gram in the reference appear in candidate
b. METEOR:
    i. (1 - weighed F score) * Alignment Penalty
    ii. F-score may use synonym or stemming.
    iii. Concept of **Chunk** in penalty term: In order to compute this penalty, unigrams are grouped into the fewest possible chunks, where a chunk is defined as a set of unigrams that are adjacent in both the hypothesis and in the reference. The longer the adjacent mappings between the candidate and the reference, the fewer chunks there are. A translation that is identical to the reference will give just one chunk.

Here is a good post for METEOR and ROUGE

Classic Algorithms

CLUSTERING

General Guideline: TODO
https://towardsdatascience.com/the-5-clustering-algorithms-data-scientists-need-to-know-a36d136ef68

Hierarchical Clustering
Agglomerative Clustering: Bottom-up, small clusters to larger clusters
Divisive Clustering: Top-down, start with a big single cluster, then

K-means Algorithm

1. Initialize **cluster centroids** $\mu_1, \mu_2, \ldots, \mu_k \in \mathbb{R}^n$ randomly.
2. Repeat until convergence: $\{$

    For every i, set
    $$c^{(i)} := \arg\min_j ||x^{(i)} - \mu_j||^2.$$

    For each j, set
    $$\mu_j := \frac{\sum_{i=1}^m 1\{c^{(i)} = j\} x^{(i)}}{\sum_{i=1}^m 1\{c^{(i)} = j\}}.$$

    $\}$

1. Initialize k cluster centroids (essentially select k training examples as centroids)
2. For each data point in the space, assign it to the closest centroid; therefore we form k-clusters
3. Re-compute centroids as the mean of the coordinates
4. Repeat from 2 until converge

SUPPORT VECTOR MACHINE (SVM)

(zhihua zhou) TODO

HIDDEN MARKOV MODEL (HMM)

(zhihua zhou) TODO

Reinforcement Learning

MDP AND MAB

Multi-Armed Bandits

A bandit has k arms and the agent can select one arm each time. Every selection costs 1 coin. Each of the k arms has a probability of returning the coin. The agent wants to maximize the reward. If we want to know the expected reward for each arm, we can adopt two methods: exploration-only and exploitation only.

Markov Decision Process

TODO

Epsilon-Greedy

TODO

Softmax

TODO

MODEL-BASED LEARNING

Policy Iteration

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Repeat
   $\Delta \leftarrow 0$
   For each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r|s, \pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   $a \leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
   If $a \neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return V and π; else go to 2

Iterative policy evaluation

For each state, its value is updated based on a fixed policy. Repeat until there is no change in value of *every* state.

Policy Improvement

TODO

Preferred Situation

When we have many actions and a few states

Value Iteration

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat
$\Delta \leftarrow 0$
For each $s \in \mathcal{S}$:
$v \leftarrow V(s)$
$V(s) \leftarrow \max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$
$\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, π, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r|s, a)\left[r + \gamma V(s')\right]$

Preferred Situation
When we have many states and only as few actions
model-free learning

Monte-Carlo Method

- On-policy

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \leftarrow$ arbitrary
$Returns(s,a) \leftarrow$ empty list
$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:
(a) Generate an episode using π
(b) For each pair s, a appearing in the episode:
$G \leftarrow$ return following the first occurrence of s, a
Append G to $Returns(s,a)$
$Q(s,a) \leftarrow \text{average}(Returns(s,a))$
(c) For each s in the episode:
$A^* \leftarrow \arg\max_a Q(s,a)$
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

We call it on-policy because we perform policy evaluation and policy iteration on the same policy.
Explanation for a= A* policy:
if a = A*, there is a (1 - epsilon) probability that it is chosen as the max policy; however, there is also a small probability of epsilon / A(s) for this action to be chosen randomly.

- Off-policy

Initialize, for all s and a

$Q(s,a) \leftarrow$ random

$Returns(s,a) \leftarrow$ Empty List

$\pi(a|s)$ is a deterministic policy

Repeat forever

(a) Generate an episode using ε-greedy of π

(b) For each s_i, a_i pair in the episode (Policy Evaluation)

(1) Importance Sampling Ratio

$$\frac{p_i^{\pi}}{p_i^{\pi'}} = \prod_{t=i}^{T-1} \frac{\pi(x_t, a_t)}{\pi'(x_t, a_t)}$$

(2) Cumulative future reward $G = \frac{1}{T-t}\left(\sum_i^{T-1} r\right) \cdot \frac{p_i^{\pi}}{p_i^{\pi'}}$

(3) $Returns(s,a)$.append(G)

(4) $Q(s,a) = avg(Returns(s,a))$

(c) For each state S: (Policy improvement)

$$\pi(a|s) = \arg\max_{a'} Q(x, a')$$

Motivation:
TODO

Temporal-Difference (TD)
Compared to Dynamic Programming, MC is not efficient since it will perform an update once every (state, action) pair is visited in the trajectory. TD combines the advantages the two algorithms above, it can not only adapt to the model-free environment, but also update Q-value per time-step.

- SARSA (On policy)

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

- Q-learning (Off policy)

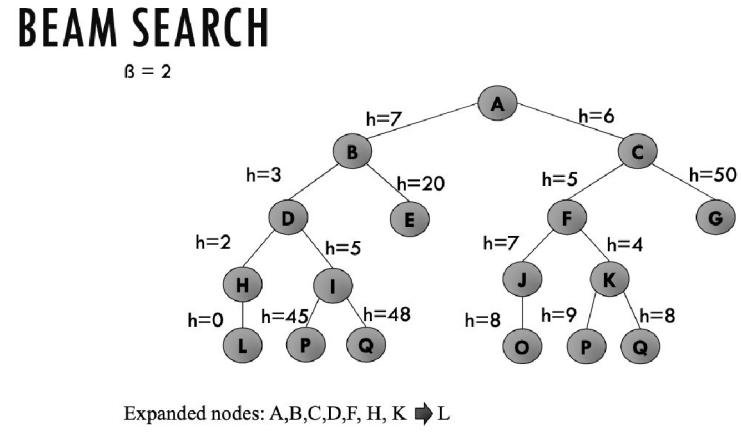
Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
  Initialize S
  Repeat (for each step of episode):
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
    $S \leftarrow S'$;
  until S is terminal

Value Function Approximation
For all the algorithms above, we assume we have a tabular mapping between state-action and value. However, in real life cases, there is an enormous space of states and the states are usually continuous. It is also hard to discretize the states before exploration. So how about learning a state-value function?
TODO

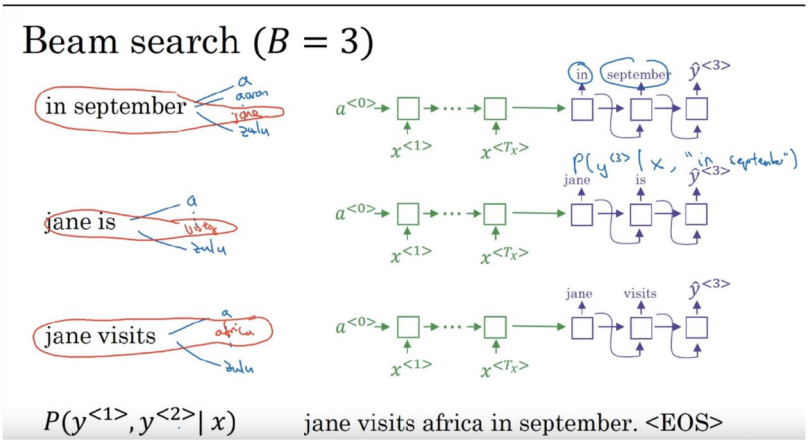
Deep Reinforcement Learning
DQN and Policy Gradient

Miscellaneous
model calibration

https://arthurdouillard.com/post/miscalibration/
https://geoffpleiss.com/nn_calibration
https://machinelearningmastery.com/calibrated-classification-model-in-scikit-learn/
beam search



From a search algorithm's perspective, beam search is just an informed BFS. Every time we just keep the best [beam width] number of instances and expand from them.



From an encoder-decoder framework's perspective, at every time step of decoding, we only keep the top [beam width] most likely words in the softmax and feed these words into the next stage.

For example, in the left figure, the beam width is 3.