


TOKENIZATION

STARTER GUIDE



pc: <https://medium.com/@utkarsh.kant>

swipe right 

WORD LEVEL TOKENIZATION

Splitting text into individual words

"the quick brown fox" -> ["the","quick","brown","fox"]

- Straight-forward and simple.
- Flexible for further processing ; part-of-speech tagging,NER,syntactic parsing
- Adapts to multiple languages
- Allows model to focus on relationships between words

BUT

- Ignores multi-words, punctuations within words..etc
- Large Vocabulary size (English has 180k active words) leading to higher computational demand
- Dependency on the context to tokenize

swipe right 

CHARACTER LEVEL TOKENIZATION

Splitting text into individual characters

"the quick brown fox" -> ["t","h","e"," ","q","u","i","c","k", ...]

- Simple and Efficient. English has only 26 characters
- No out-of-vocabulary issues. Train and test vocabulary are mostly the same.
- Adapts to any language
- Robust to spelling errors

BUT

- Increases model complexity
- Loss of high-level info like morphology, syntax, semantic relationships, sentiment etc
- Doesn't capture long-range dependencies
- Requires more training data than other tokenizations

swipe right 

N-GRAM MODELS

Splitting text into groups of consecutive words.

N is the no. of words in a token. For a trigram model,

"the quick brown fox jumps" -> ["the quick brown","quick brown fox","brown fox jumps"]

- Captures the context in the sentence
- The choice of N allows for flexibility
- Foundation to many probabilistic language models
- Still used in text-completion for gmail, google search, plagiarism detection.

BUT

- Smaller "n" values don't capture relationships well.
Larger "n" lead to sparsity and high dimensionality
- As "n" increases, parameters increase exponentially
- Don't capture long-range dependencies

swipe right 

Long-range dependency

Here's why long range dependency is crucial yet difficult

Syntactic dependencies

- “The **man** next to the large oak tree near the grocery store on the corner **is** tall.”
- “The **men** next to the large oak tree near the grocery store on the corner **are** tall.”

The verb here depends on the noun which is **12 n-grams** away. It is hard to capture this relationship with a n-gram model.

swipe right 

Byte Pair Tokenization (BPT)

It iteratively merges the most frequent adjacent pairs of chars(bytes) and forms new tokens.

For simplicity, Let's say we have a small train dataset:

"aaabbbbccda"

1. Start with char level tokens: ['a','b','c','d']
2. Find the most frequent **pair** of adjacent tokens: 'bb'
3. Create a new token by merging the pair in step 2 ,
New tokens : ['a','b','c','d','bb']
4. Repeat until a stopping criterion is met

GPT, BERT, RoBERTa are some of the popular models using BPT

swipe right 


BPT contd.

It is currently the most popular tokenization for LLMS

- BPE is efficient in handling rare words as it breaks them down to sub words
- Doesn't require large vocabulary
- Adaptable to many languages and domains
- Capture long-term dependencies

BUT

- Characters are merged purely on frequency leading to context insensitivity
- When a new token shows up; the tokenizer needs to be re-trained to incorporate it
- BPT in itself is a model, i.e., it is trained separately with its own dataset, algorithms and hyper-parameters.

swipe right 

A few more SoTA tokenizers

- SentencePiece (LLAMA)
- WordPiece (DistilBERT, Electra)
- Domain-specific : Biomedical, code tokenizers
- YouTokenToMe (NVIDIA NeMo)

swipe right 

Model Capacity

Model width and depth combinedly indicate the capacity of model to fit a dataset

- If model capacity is too much for the problem complexity, overfitting happens
- if model capacity is not enough for the problem capacity, underfitting happens
- The parameters and computational cost increases as model capacity increases
- **Find a sweet balance between**
 - *Underfitting*
 - *Overfitting*
 - *Computation cost*

swipe right 

Learning Rate

The size of optimization step taken to minimize loss function

- Common starting point : 0.01 or 0.001

LOSS CURVE ANALYSIS

- If the loss decreases slowly, increase the learning rate
- if the loss drops and then become stagnant, decrease the learning rate
- if the loss drops and then starts increasing, decrease your learning rate.

AUTOMATED APPROACHES

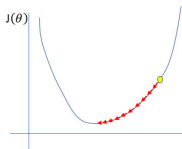
- *Learning Rate Schedules*
- *The Learning Rate Range Test*
- *Cyclical Learning Rates*
- *Bayesian optimization*

swipe right



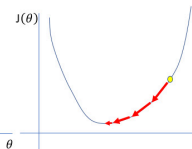
Learning Rate vs Objective function

Too low



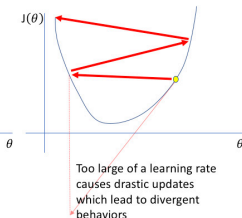
A small learning rate requires many updates before reaching the minimum point

Just right



The optimal learning rate swiftly reaches the minimum point

Too high



Too large of a learning rate causes drastic updates which lead to divergent behaviors

Batch Size

The no of inputs that are passed to the model before updating it's weights

- *HARDWARE* : Large batch sizes need more memory
- As the batch size increases, the convergence speed increases
- Find a good balance between training speed and convergence quality
- *Higher learning rate works well with larger batch size*

THUMB RULES

- Start with largest batch size your memory can handle
- if overfitting, reduce batch size

swipe right 

Activation Functions

Made a seperate post on this. link in comments.

THUMB RULES

- Start with ReLU for hidden layers, then GeLU and Tanh
- If binary classification use Sigmoid for output
- If multi-class, use Softmax for output
- For transformer-based models, start with GeLU
- If you observe "dying relu problem", try Leaky ReLU

swipe right



Regularization Techniques

Add a penalty or include randomness in training to reduce overfitting

Dropout: Easiest form of regularization. dropout rate is typically between 0.2-0.5

L2 Regularization (Ridge): If dropout doesn't work, try this

L1 Regularization (Lasso) : Useful if your data is sparse.

Data Augmentation: More data can act as a regularization

Ensembling: Increases generalization hence, reduces overfitting. Last option.

swipe right 

Optimizers

Optimizer decides the way you update your weights based on your loss.

SGD: Simplest optimizer. Slow but reliable. A good beginner option.

SGD with momentum: time consuming but a great learning experience.

ADAM: Most popular choice for the current models. Fast and complex.

RMSprop: Works well with RNNs

swipe right 

Train and Error (not a typo!)

The guidelines and thumb rules are a great way to start but at the end it's all empirical.

In most cases, you will start with a pre-trained model. Start with the default hyper-parameters and then change based on model performance.

You can control the speed and complexity but you can't control the performance of your model.

Sometimes, You just need to try all feasible options and find out the best combo.

swipe right 

CREDITS FOR FIGURES

<https://deeptai.org/machine-learning-glossary-and-terms/hyperparameter>

<https://www.jeremyjordan.me/nn-learning-rate/>

swipe right

