1 Hugging Face Basics

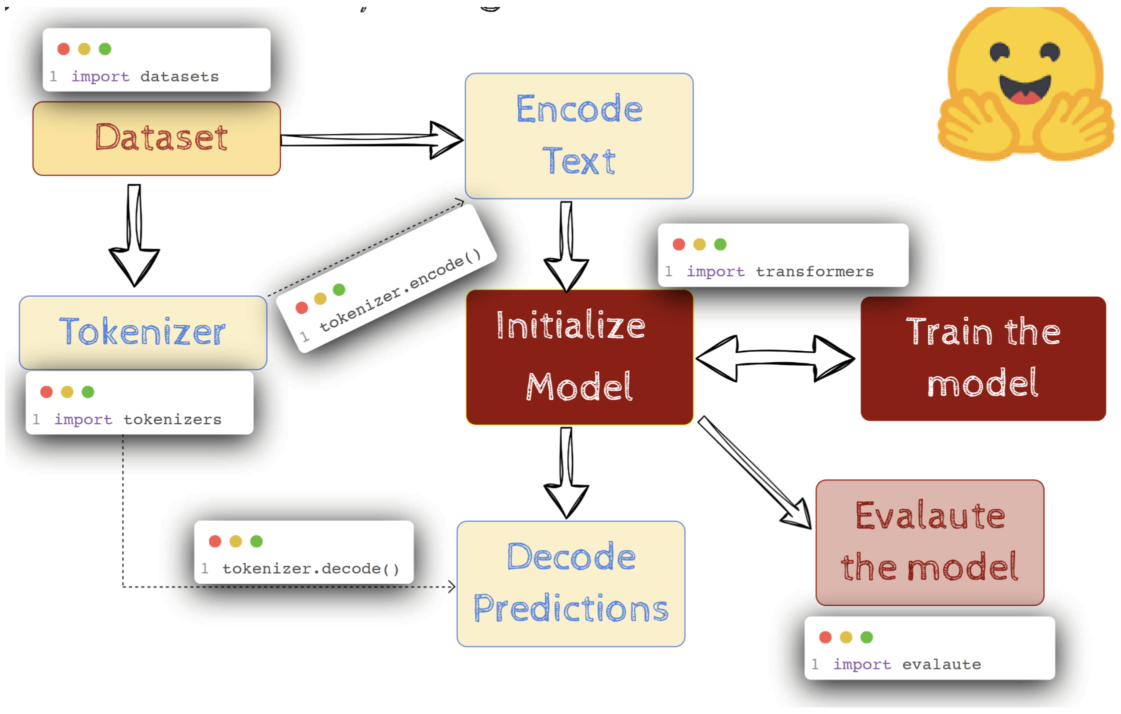

Figure 1: HF Summary

1.1 Datasets

- The datasets come with different formats and different sizes
- We need to understand the format and may need to write a script to parse and load the text data.
- If the memory is limited, we need to implement mechanisms to stream the samples in the dataset.
- However, it would be convenient if we have a single place to load any of these datasets with a consistent call signature.
- The `datasets` module provides a simple interface to download a specific dataset from thousands of available datasets on HF.
- When we download the dataset for the first time, it will be cached locally.
- The cached data is stored in a memory-mapped columnar format, which uses less memory.
- Before using `datasets` module, make sure to run the following commands for nlp, audio and vision. `pip install datasets`, `pip install datasets[audio]`, `pip install datasets[vision]`
- Let us first download a small dataset of movie reviews used in supervised learning.

```
from datasets import load_dataset
imdb_dataset = load_dataset("stanfordnlp/imdb")
```

- Notice the structure: the "train" and "test" splits are wrapped by the `DatasetDict` class.
- Each split is an instance of the `Dataset` class and contains two fields: $features$ and num_rows.
- Only train set can be extracted by `imdb_train_split = imdb_dataset["train]`
- We can remove the third split, *unsupervised*, by using `_ = imdb_dataset.pop("unsupervised")`

- If we want to download the train split alone,
  `train_split = load_dataset("stanfordnlp/imdb", split = "train")`
- We can divide the split further into ”train” and ”test”(evaluation) splits,
  `small_ds = train_split.train_test_split(test_size = 0.2)`
- Suppose we have two files, namely ”tain.csv” and ”test.csv” in a directory named ”data”.

```
data_files = ["data/train.csv", "data/test.csv"]
local_dataset = load_dataset("csv", data_files = data_files)
```

- We can convert the dataset from any format to *pyarrow* format by saving it to the disk.
  `train_test_split = local_dataset["train"].train_test_split(test_size = 0.2)`
  `train_test_splits.save_to_disk('pyarrow_dataset/movie_review')`
- Now we can load from the disk

```
from datasets import load_from_disk
raw_dataset_from_disk = load_from_disk('pyarrow_dataset/movie_review')
```

- We can select a specific sample, `example = imdb_dataset["train"][idx]`
- We can also pass a list of indices, `example = imdb_dataset["train"].select([idx])`
- **Translation Dataset**: we will take a look at ”wmt/wmt14” which contains 5 sub-datasets.
  `from datasets import get_dataset_config_names, get_dataset_split_names`, used to see what splits the dataset has.
  `translation_dataset = load_dataset(path = "wmt/wmt14", name = "hi-en")`
- We can combine all samples,
  `translation_dataset=load_dataset(path="wmt/wmt14",name="hi-en",split="train+test+validation")`
- Features defines the internal structure of a dataset. It is used to specify the underlying serialization format.
- Another dataset is Microsoft Research Paraphrase Corpus(MRPC)
  `mrpc_dataset = load_dataset('glue', 'mrpc', split = 'train')`
- We can filter the dataset based on certain conditions,
  `imdb_filtered_dataset=imdb_dataset.filter(`
  `lambda examples:len(example['text'].split(' ')>=num_words))`
- We can also use map

```
def add_prefix(example):
    example["text"] = "IMDB:"+example["text"]
    return example

imdb_prefixed_dataset = imdb_dataset.map(add_prefix)
```

- Often we need to combine two or more datasets to create a larger dataset.
- The only requirement is that the datasets must have same features and same number of splits.
  `concat_dataset = datasets.concatenate_datasets(imdb_dataset_whole, rt_dataset_whole, axis = 0)`
- Often we have n skewed datasets, so we need to build a new dataset by intervening the samples from the dataset according to its distribution.

```
from datasets import interleave_datasets
inter_datasets = interleave_datasets([imdb_dataset_whole, rt_dataset_whole],
                                     probabilities = [0.6,0.4])
```

- By default, `stopping_strategy=first_exhausted`, construction is stopped as soon as one of the datasets runs out of sample.
- **Iterable Dataset**: Suitable for loading samples from large datasets iteratively without writing anything to local disk.
  `imdb_iter_dataset = load_dataset("stanfordnlp/imdb", split = "train", streaming = True)`
- Can we load a dataset directly from external links?
- Can we load a dataset from a zipped file directly?
- Colab Notebook: `https://github.com/Arunprakash-A/Modern-NLP-with-Hugging-Face/blob/main/Notebooks/Datasets.ipynb`
- Week 1 assignment: `https://colab.research.google.com/drive/13Cx57lk5cZ1m_2tjHA3ABKDhrnO3wzjb`

1.2 Tokenization

- The tokenizer simply splits the input sequence into tokens.
- A simple approach is to use whitespace for splitting the text.
- Each token is associated with a unique ID.
- Each ID is associated with a unique embedding vector.
- The model then takes embeddings and predicts token IDs.
- During inference, the predicted token IDs are converted back to tokens and then to words.
- The size of the vocabulary determines the size of the embedding table.
- We can split the text into words using whitespace, pre-tokenization, and add all unique words in the corpus to the vocabulary.
- We also include special tokens such as ⟨go⟩, ⟨stop⟩, ⟨mask⟩, ⟨cls⟩ and others to the vocabulary based on the type of downstream tasks and architecture choice.
- **Challenges in building a vocabulary**
  1. **What should be the size of vocabulary?**: Larger the size, larger the size of embedding matrix and greater the complexity of computing the softmax probabilities. What is the optimal size?
  2. **Out-of-vocabulary**: If we limit the size of the vocabulary (say, 250K to 50K), then we need a mechanism to handle out-of-vocabulary (OOV) words. How do we handle them?
  3. **Handling misspelled words in corpus**: Often, the corpus is built by scraping the web. There are chances of typo/spelling errors. Such erroneous words should not be considered as unique words.
  4. **Open Vocabulary problem**: A new word can be constructed (say, in agglutinative languages) by combining existing words. The vocabulary, in principle, is infinite (that is, names, numbers, ...) which makes a task like machine translation challenging
- We want a Moderate-sized Vocabulary, to Efficiently handle unknown words during inference, and Be language agnostic
- Tokenization can be character level, sub-word level, and word level.
- **General Pre-processing**: First the text is normalized which involves operations such as treating cases, removing accents, eliminating multiple whitespace, handling HTML tags, etc.
- Splitting the text by a whitespace was traditionally called tokenization. However, when it is used with a sub-word tokenization algorithm, it is called **pre-tokenization**.
- Learn the vocabulary(training) using these words.
- **Byte Pair Encoding**
  1. Starts with a dictionary that contains words and their count.
  2. Append a special symbol ⟨/w⟩ at the end of each word in the dictionary.
  3. Set required number of merges, a hyperparameter
  4. Initialize the character-frequency table, a base vocabulary
  5. Get the frequency count for a pair of characters
  6. Merge pairs with maximum occurrence

```
import re, collections
def get_stats(vocab):
    pairs = collecitons.defaultdict(int)
    for word, freq in vocab.items():
        symbols = word.split()
        for i in range(len(symbols)-1):
            pairs[symbols[i],symbols[i+1]] += freq
    return pairs

def merge_vocab(pair, v_in):
    v_out = {}
```

```
        bigram = re.escape(' '.join(pair))
        p = re.compile(r'(?<!\S)' + bigram + r'(?!\S)')
        for word in v_in:
            w_out = p.sub(''.join(pair), word)
            v_out[w_out] = v_in[word]
        return v_out

    vocab = {'l o w </w>':5, 'l o w e r </w>':2, 'n e w e s t </w>':6,
    'w i d e s t </w>':3}
    num_merges = 10

    for i in range(num_merges):
        pairs = get_stats(vocab)
        best = max(pairs, key=pairs.get)
        vocab = merge_vocab(best, vocab)
```

- The final vocabulary contains initial vocabulary and all the merges.
- The rare words are broken down into two or more subwords.
- At test time, the input word is split into a sequence of characters, and the characters are merged into a larger and known symbols.
- Languages such as Japanese and Korean are non-segmented, we can use language specific morphology based word segmenters.
- Changing merge order will give different outputs in BPE. Therefore, BPE is greedy and deterministic.
- A repo for BPE: `https://github.com/karpathy/minbpe/tree/master`
- **BPE Dropout**: `https://arxiv.org/pdf/1910.13267.pdf`
- **Word Piece Tokenizer**: In BPE, we merged a pair of tokens with most frequency of occurrence. If there was a tie, we took the first occurring element.
- Instead, now we calculate a score $\frac{count(\alpha,\beta)}{count(\alpha)count(\beta}$ which allows us to select a pair of tokens where the individual tokens are less frequent in the vocabulary.
- Merge the pair with the highest score.
- **Sentence Piece Tokenizer**: Start with a large vocabulary and shrink it.
- The probabilistic approach is to find the subword sequence $x^* \in \{x_1, x_2, ..., x_k\}$ that maximizes the likelihood of the word X.
- The word X in sentence piece means a sequence of characters or words (without spaces)
- Therefore, it can be applied to languages (like Chinese and Japanese) that do not use any word delimiters in a sentence.
  1. Construct a reasonably large seed vocabulary using BPE or Extended Suffix Array algorithm.
  2. **E-Step**: Estimate the probability for every token in the given vocabulary using frequency counts in the training corpus
  3. **M-Step**: Use Viterbi algorithm to segment the corpus and return optimal segments that maximizes the (log) likelihood.
  4. Compute the likelihood for each new subword from optimal segments
  5. Shrink the vocabulary size by removing top of subwords that have the smallest likelihood.
  6. Repeat step 2 to 5 until desired vocabulary size is reached
- HF tokenizers module provides a class that encapsulates all of these components.
  `from tokenizers import Tokenizer`
- We can customize each step of the tokenizer pipeline via the setter and getter properties of the class.
- We do not need to call each of these methods sequentially to build a tokenizer.
- After setting Normalizer and Pre-Tokenizer, we just need to call the train methods to build the vocabulary.

- Once training is complete, we can call methods such as encode for encoding the input string.

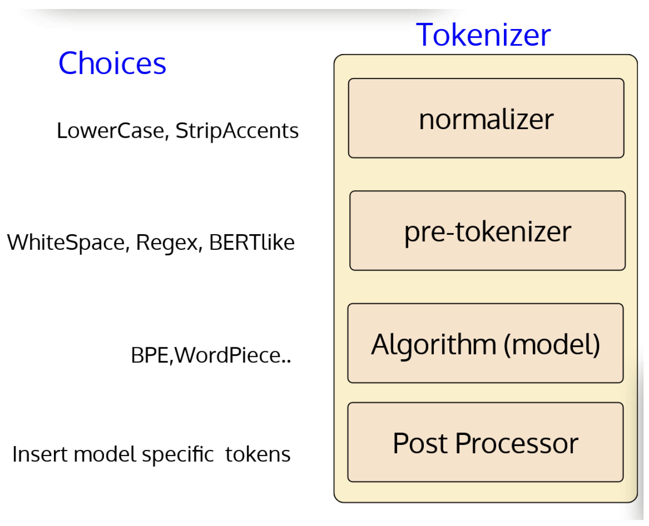

Figure 2: Tokenizer Pipeline

- We will try to build the pipeline
  normalizer(Lowercase) → pre tokenizer(Whitespace) → model(BPE) → post processor(None)

```
from tokenizers.normalizers import Lowercase
from tokenizers.pre_tokenizers import Whitespace
from tokenizers.models import BPE
```

- Initiate the tokenizer with the BPE model and the special tokens, for example $[UNK]$, that the model will use during **prediction**.

```
model = BPE(unk_token="[UNK]")
tokenizer = Tokenizer(model)
```

- Add the normalizer and pre-tokenizer to the pipeline

```
tokenizer.normalizer = Lowercase()
tokenizer.pre_tokenizer = Whitespace()
```

- Similarly, we can customize each step of the tokenizer pipeline
  `model, normalizer, pre_tokenizer, post_processor, padding(no setter), truncation(no setter), decod`
- Tokenizer also provides a set of methods for training, encoding inputs, decoding, etc. `https://huggingface.co/docs/tokenizers/api/tokenizer`
- We do not need to call each of these methods sequentially to build a tokenizer
- After setting Normalizer and Pre-Tokenizer, we just need to call the train methods to build the vocabulary
- Once the training is complete, we can call methods such as `encode` for encoding the input string
- We can customize the output behavior by passing the instance of `Encode` object to the `post_process()` methods of Tokenizer (it is used internally, we can just set the desired formats in the optional parameters like "pair" in `encode()`)
- Colab Notebook: `https://github.com/Arunprakash-A/Modern-NLP-with-Hugging-Face/blob/main/Notebooks/Tokenizer.ipynb`
- Week 2 assignment: `https://colab.research.google.com/drive/1TbnvAPEPH7akUPlbdxNRIuBPxYskhMkW`

2 Large Language Models

2.1 Training and Fine-Tuning

- Training a new transformer for different tasks can be tedious, so instead we train the transformer such that the output is conditioned on the input prompt.
- Can a model develop basic understanding of language by getting exposure to a large amount of raw text? [**pre-training**]
- More importantly, after getting exposed to such raw data can it learn to perform well on downstream tasks with minimum supervision? [**Supervised Fine-tuning**]
- A simple way to make a model understand language is to Teach it the task of predicting the next token in a sequence
- Roughly speaking, this task of predicting the next token in a sequence is called language modelling
- We can also use autoregressive models where the conditional probabilities are given by parameterized functions with a fixed number of parameters (like transformers).
- GPT is trained on Causal Language Modeling objective, which is

$$L = \sum_{i=1}^{T} \log P(x_i | x_1, ..., x_{i-1}; \theta)$$

- Consider a tokenizer with vocabulary size $|V|$ and embedding dimension E.
- Consider a model that contains 12 decoder layers, with context size C, A attention heads and FFN hidden layer size $E \times 4 = H$.
- Number of Parameters
  1. Token Embeddings: $|V| \times E$
  2. Position Embeddings: $C \times E$
  3. Attention Parameters per block, $W_Q = W_K = W_V = E \times E/A$
  4. Per attention head: $3 \times W_Q$
  5. For A heads: $3 \times W_Q \times A = AP$
  6. For Linear Layer in attention block: $E \times E = L$
  7. For all 12 blocks: $12 \times (L + AP)$
  8. FFN parameters per block: $2 \times (E \times H) + E + H = FFN$
  9. For 12 blocks: $12 \times FFN$
  10. Total Parameters: $|V| \times E + C \times E + 12 \times (L + AP) + 12 \times FFN$
- Pipeline for training
  $\boxed{\text{Dataset}} \rightarrow \boxed{\text{Tokenizer}} \rightarrow \boxed{\text{DataLoader}} \rightarrow \boxed{\text{Initialize the Model}} \rightarrow \boxed{\text{Train the Model}}$
- Colab Notebook: `https://github.com/Arunprakash-A/Modern-NLP-with-Hugging-Face/blob/main/Notebooks/PreTraining_GPT.ipynb`

2.2 Task Specific Fine-Tuning

- We trained the GPT-2 model with the CLM(Causal Language Modeling) training objective
- One approach for using the pre-trained models for downstream tasks is to individually fine-tune the parameters of the model for each task.
- We make a copy of the pre-trained model for each task and fine-tune it on the dataset specific to that task.
- Stanford Sentiment Tree Bank(SST) for sentiment classification
- SNLI for classification tasks
- LAMBADA for predicting the last word of a long sentence.

- **Fine-tuning** involves adapting a model for various downstream tasks, with a minimal or no change in the architecture.
- Initialize the parameters with the parameters learned by solving the pre-training objective.
- At the input side, add additional tokens based on the type of downstream task. For example, $\langle s \rangle$ and end $\langle e \rangle$ tokens for classification tasks.
- At the output side, replace the pre-training LM head with the classification head (a linear layer W_y)
- Now our objective is to predict the label of the input sequence

$$\hat{y} = P(y|x_1, ..., x_m) = softmax(W_y h_l^m)$$

- It makes sense as the entire sentence is encoded only at the last time step due to causal masking.
- W_y is randomly initialized. Padding or truncation is applied if the input sequence is less or greater than the context length.
- We can freeze the pre-trained model parameters and randomly initialize the weights of the classification head while training the model.
- In this case, the pre-trained model acts as a feature extractor and the classification head act as a simple classifier.
- The other option is to train all the parameters of the model, which is called **full fine-tuning**.
- In general, the latter approach provides better performance on downstream tasks than the former.
- Many approaches have been developed to tackle the memory requirement to fine-tune large models.

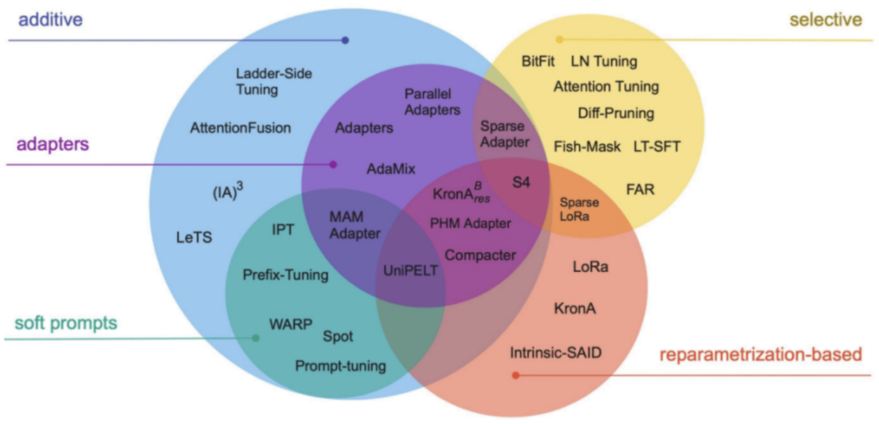

Figure 3: Different Approaches to tackle memory requirements

- Of these Parameter Efficient Fine-Tuning (PEFT) techniques, LoRa, QLoRa and AdaLoRa are most commonly used (optionally, in combination with quantization)
- **Guide to PEFT**: `https://arxiv.org/abs/2303.15647`
- Fine-tuning large models is costly, as it still requires thousands of samples to perform well in a downstream task.
- Some tasks may not have enough labelled samples.
- Moreover, this is not how humans adapt to different tasks once they understand the language.
- **Language Models for few shot Learners**: `https://arxiv.org/pdf/2005.14165`
- We want a single model that learns to do multiple tasks with zero or few examples
- We adapt a pre-trained language model for downstream tasks witout any exploit supervision, called **zero-shot transfer**.
- The prompts are words. Therefor, we just need to change the single task(LM) formulation

$$\text{Change } p(output|input) \text{ to multitask } p(output|input, task)$$

- To get a good performance, we need to scale up both the model size and the data size.
- The ability to learn from a few examples improves as model size increases.
- For certain tasks, the performance is comparable to that achieved through full task-specific fine-tuning.
- Since the model learns a new task from samples within the context window, this approach is called **in-context learning**.
- This remarkable ability enables the adaptation of the model to downstream tasks without the need for gradient-based fine-tuning.
- Adaptation happens on-the-fly in inference mode (which consumes far less memory)
- Since adaptation occurs during inference, there is **no need to share** model weights for fine-tuning.
- This approach enables the model's deployment across a variety of use cases through simple API calls, making it more accessible.
- This new ability paved the way for fine-tuning the model for specific tasks by Prompting.
- There are multiple ways of prompting the model, **Zero-shot**, **Few-shot (in-context)**, **Chain of Thought**, **Prompt Chaining**.
- Zero-shot learning performance is often poor, despite the model size (say, GPT 3 175B), especially in following the user intent (instructions).
- Fine-tune the model on the instructions, one approach is to reformat at the available datasets into instruction sets.
- **Fine-Tuned Language Models are Zero Shot Learners**: `https://arxiv.org/pdf/2109.01652`

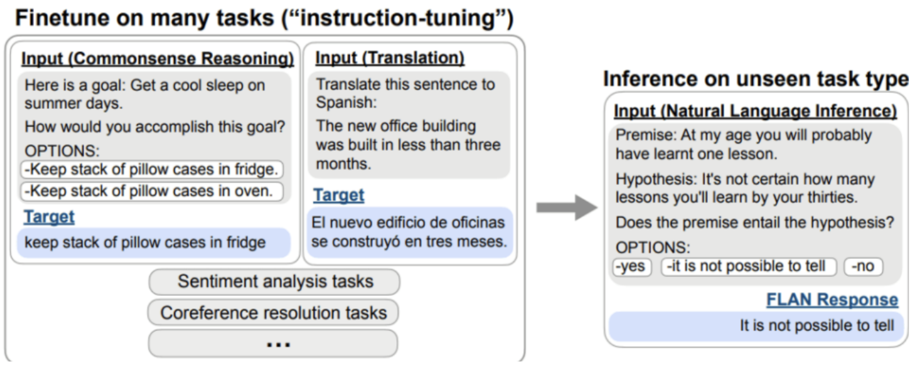
Finetune on many tasks ("instruction-tuning")
Input (Commonsense Reasoning)
Here is a goal: Get a cool sleep on summer days.
How would you accomplish this goal?
OPTIONS:
-Keep stack of pillow cases in fridge.
-Keep stack of pillow cases in oven.
Target
keep stack of pillow cases in fridge
Input (Translation)
Translate this sentence to Spanish:
The new office building was built in less than three months.
Target
El nuevo edificio de oficinas se construyó en tres meses.
Sentiment analysis tasks
Coreference resolution tasks
...
Inference on unseen task type
Input (Natural Language Inference)
Premise: At my age you will probably have learnt one lesson.
Hypothesis: It's not certain how many lessons you'll learn by your thirties.
Does the premise entail the hypothesis?
OPTIONS:
-yes
-it is not possible to tell
-no
FLAN Response
It is not possible to tell

Figure 4: FLAN

- "Making language models bigger does not inherently make them better at following a user's intent.", Training language models to follow instructions with human feedback
- Language Modelling objective is "misaligned" with user intent. Therefore, we must fine-tune the model to align with the user intent. This requires human labelled demonstrations for a collection of prompts (a labor-intensive task)
- Use these collections to fine-tune (Supervised Fine-Tuning, SFT) the model using Reinforcement Learning from Human Feedback (RLHF)

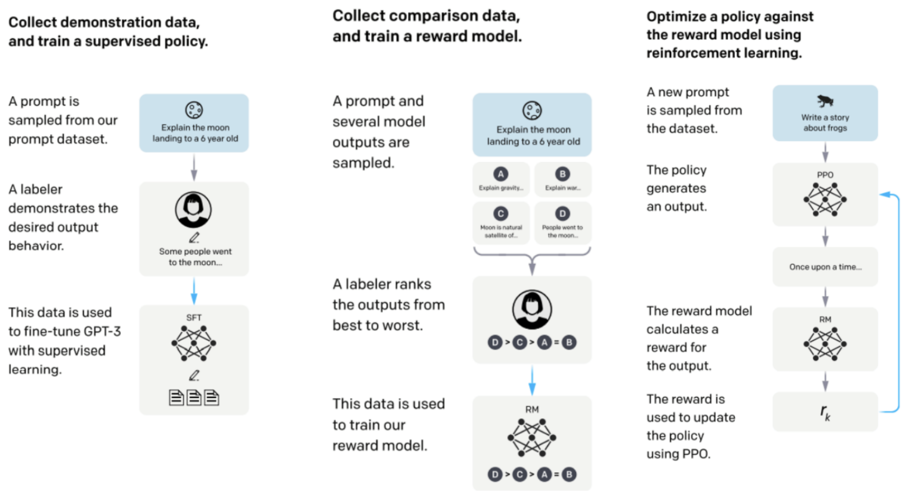
Collect demonstration data, and train a supervised policy.
A prompt is sampled from our prompt dataset.
Explain the moon landing to a 6 year old
A labeler demonstrates the desired output behavior.
Some people went to the moon...
This data is used to fine-tune GPT-3 with supervised learning.
SFT
Collect comparison data, and train a reward model.
A prompt and several model outputs are sampled.
Explain the moon landing to a 6 year old
A
Explain gravity...
B
Explain war...
C
Moon is natural satellite of...
D
People went to the moon...
A labeler ranks the outputs from best to worst.
D > C > A = B
This data is used to train our reward model.
RM
D > C > A = B
Optimize a policy against the reward model using reinforcement learning.
A new prompt is sampled from the dataset.
Write a story about frogs
The policy generates an output.
PPO
Once upon a time...
The reward model calculates a reward for the output.
RM
The reward is used to update the policy using PPO.
r_k

Figure 5: Preference Tuning via RLHF

- Fine-Tuning can be broadly categorized into two categories, Supervised Fine-Tuning(SFT) and Prompt Tuning.
- **Supervised Fine-Tuning**: Task-specific Full Fine-tuning, (task agnostic) Instruction-tuning, Memory efficient fine-tuning(PEFT, Quantization).
- **Prompt tuning**: Zero-shot, few-shot, Chain of Thought, Prompt chaining, Meta prompting
- Additional Modules: `peft`, `trl`,`SFTTrainer` (for preference tuning), `bitsandbytes` (quantization), `Unsloth` (for single-GPU, 2.5× faster training)
- Continued Pretraining of Llama 3.2 1B: `https://github.com/Arunprakash-A/Modern-NLP-with-Hugging-Face/blob/main/Notebooks/ContPreTrain-Peft-quantize.ipynb`
- Task-specific fine-tuning (classification): `https://github.com/Arunprakash-A/Modern-NLP-with-Hugging-Face/blob/main/Notebooks/Task-Specific-FineTuning.ipynb`
- Task-Specific fine-tuning with LoRA: `https://github.com/Arunprakash-A/Modern-NLP-with-Hugging-Face/blob/main/Notebooks/Task-Specific-FineTuning-LoRA.ipynb`
- Instruction tuning with unsloth: `https://colab.research.google.com/drive/1Ys44kVvmeZtnICzWz0xgpRnrI0jZAu usp=sharing`

3 Speech and Audio Processing

3.1 Language Identification

- We want to identify the language given the speech signal.
- The basic unit of text is character.
- Similarly, speech has a basic unit called **phenom**.
- We use Wave2Vec model to extract features from audio files and then use any ml/DL model for classification.
- **Wav2Vec2Processor**
  1. The processor is responsible for preprocessing and post-processing audio data.
  2. It converts raw audio waveforms (e.g., .wav files) into a numerical format suitable for input to the model.
  3. Typically includes: Feature extraction (e.g., converting waveform to normalized values), Tokenization (if the model outputs text), decoding (if needed for post-processing).
  4. Combines Wav2Vec2FeatureExtractor (extracts features from raw audio) and Wav2Vec2Tokenizer (converts model outputs into readable text).
- **Wav2Vec2Model**
  1. The model is responsible for the actual deep learning computations on the processed audio input.
  2. It takes the preprocessed features and applies a self-supervised learning approach to extract speech representations.
  3. The model can be: Pretrained (outputting hidden representations) or Fine-tuned for Automatic Speech Recognition (ASR) (outputting text or phonemes).
  4. Wav2Vec2Model: Outputs raw speech embeddings (requires additional layers for ASR).
  5. Wav2Vec2ForCTC: Fine-tuned for speech-to-text (outputs transcriptions directly).
- Colab Notebook: `https://colab.research.google.com/drive/1siXesgyxS6JFOVdIDeqbpaKE5YKa0E5u`

3.2 Speaker Diarization

- **Speaker Diarization** is the process of dividing an audio recording into segments based on who is speaking
- We will solve this in a roundabout way by using speech to text, then using a speaker identification model.
- **Whisper** will be used to perform speech to text
- Then we extract features of the speech and perform clustering.
- We will use agglomerative clustering.
- Complete pipeline is given as
  $\boxed{\text{Preprocessing}} \rightarrow \boxed{\text{ASR(speech-to-text)}} \rightarrow \boxed{\text{Speaker Embedding Extraction}} \rightarrow \boxed{\text{Clustering}} \rightarrow \boxed{\text{Post-processing}}$
- Post-processing step can be ignored
- **FFmpeg tool**
  1. command-line tool for processing multimedia files.
     `!ffmpeg -i "{audio_file_path}" -ar 16000 -ac 1 -c:a pcm_s16le "{audio_file_path[:-4]}.wav"`
  2. `-i "{audio_file_path}"`: Specifies the input file
  3. `-ar 16000`: Sets the audio sampling rate to 16,000 Hz (often used in ASR models like Whisper).
  4. `-ac 1`: Converts the audio to mono (1 channel) instead of stereo.
  5. `-c:a pcm_s16le`: Specifies the audio codec: PCM (Pulse Code Modulation), 16-bit little-endian. This is a raw, uncompressed format that many speech-processing models prefer.
  6. `"{audio_file_path[:-4]}.wav"`: Saves the output file with a .wav extension by removing the last 4 characters (e.g., ".mp3") from the original file name.
- **PretrainedSpeakerEmbedding("speechbrain/spkrec-ecapa-voxceleb")**

1. Converts an audio waveform into a fixed-dimensional speaker embedding (a numerical representation of the voice).
2. Used for speaker identification, verification, and clustering.
3. Pretrained on a large dataset of speech recordings from thousands of speakers.

- **Whisper**
  1. Automatic Speech Recognition (ASR) model that transcribes spoken audio into text.
  2. Can handle multiple languages and noisy environments.
  3. Uses a Transformer-based architecture for speech-to-text.
  4. Outputs timestamps for words, allowing alignment with speaker diarization.
  5. Can be used for real-time transcription and subtitle generation.
  6. Often used in voice assistants, captioning, and speech-to-text applications.
- Colab Notebook: `https://colab.research.google.com/drive/1zTBuD8vWh9oCpnuasiWKtjOKOo3_xuiA#scrollTo=Kea-qNf1HWOG`

3.3 Automatic Speech Recognition

- In the previous problem, we identified who said when, now we want to identify who said what.
- We are interested in speech-to-text.
- **Wav2Vex2CTCTokenizer**
  1. Converts text into tokens and vice versa.
  2. Since CTC models do not predict space-separated words, they need a word delimiter token(|) instead of a space.
  3. Handles unknown tokens, padding, and delimiter formatting.
- **Wav2Vec2FeatureExtractor**
  1. Converts raw waveforms into features that the model can process.
  2. Applies normalization to the waveform.
  3. Handles padding and setting up the sampling rate.
  4. `feature_size=1`: Represents how many channels the input has (1 for mono audio).
  5. `padding_value=0.0`: The value used for padding if sequences are different lengths.
  6. `return_attention_mask=False`:Unlike text models, Wav2Vec2 doesn't require an attention mask.
- `processor = Wav2Vec2Processor(feature_extractor=feature_extractor, tokenizer=tokenizer)` will combine both the feature extractor (for audio processing) and tokenizer (for text processing). Essentially acting as a single interface to process audio input from waveform to tokenized text.
- **DataCollatorCTCWithPadding**: Custom class made to help batch different-length audio samples by applying padding dynamically. Ensures that each batch has sequences of the same length for efficient training.
- **Word Error Rate(WER)**: measures how different the model's predictions are from the correct transcriptions.
$$WER = \frac{\text{Substitutions} + \text{Insertions} + \text{Deletions}}{\text{Total Word in Reference}}$$
- A low WER is good
- **AutoModelForCTC.from_pretrained("facebook/wav2vec2-base")**
  1. Loads the pretrained Wav2Vec2 model
  2. Uses CTC loss, which is designed for sequence-to-sequence transcription without explicit alignment.
  3. Uses the vocabulary size and padding token from the tokenizer.
  4. `ctc_loss_reduction="mean"`: Computes CTC loss by averaging over all characters in a batch.
  5. `pad_token_id=processor.tokenizer.pad_token_id`: Ensures proper handling of padding tokens.
  6. `vocab_size=len(processor.tokenizer)`: Sets the vocabulary size dynamically based on the tokenizer.
- Colab Notebook: `https://colab.research.google.com/drive/1NQ3XYOO9qGvYhrchTc5YOH1MNyp6Ff9R#scrollTo=xrLfzHtD99Y_`

3.4 Text-to-Speech

- Automatically generate speech corresponding to given text.
- **Synthesized audio**: natural and intelligible speech
- **Frameworks**: Unit selection synthesis(USS), Hidden Markov Model based(HTS), Neural network based(conventional), End-to-End(E2E)
- **Training data**: text, audio pairs, continuous speech
- Given a text, using a lexicon/dictionary, convert it to a sequence of phenoms
- **Vocoder** converts Mel spectrogram(sequence of vectors) to speech.
- Along with text, we can add another feature embedding, speaker.
- **Unit Selection Synthesis**: select and concatenate units from large speech database. Natural sounding speech however requires a large database and discontinuities perceived at concatenation points.
- **Hidden Markov Model(HTS)**: Instead of storing actual waveform units, models of units stored. Based on source-filter model of speech. Extraction of features, source(log f_0) and system(MFCC)
- **Parametric**: speech is described using parameters
- **Statistical**: parameters are described using statistics (mean, variance of probability density functions)

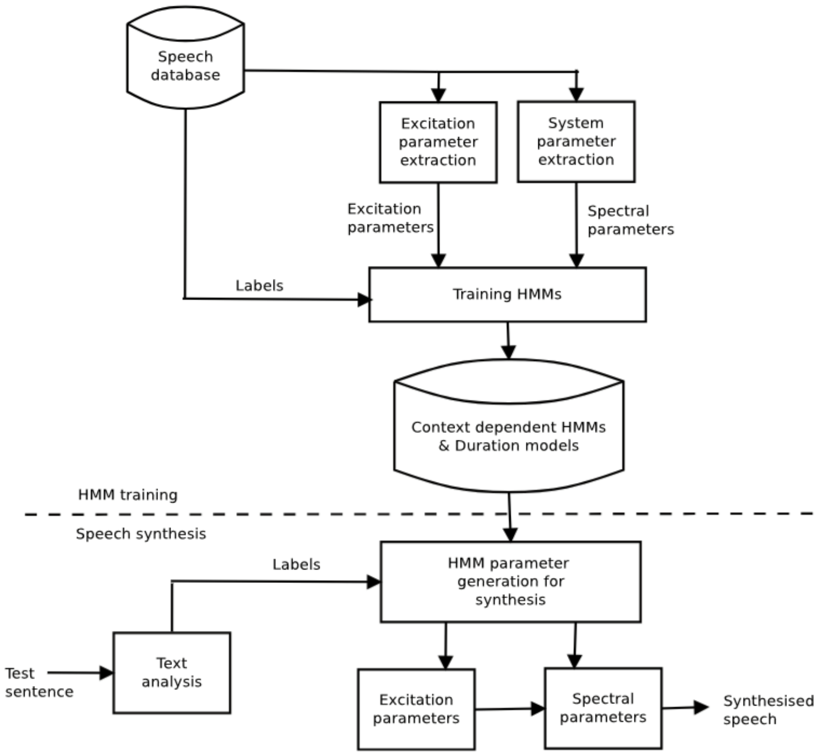

Figure 6: Training and Synthesis phases of HTS

- **Training**: Feature extraction from aligned ⟨text, speech⟩ data. Every phone is modeled by a 3-state HMM (Each state has a GMM).
- **Synthesis**: Text for synthesis is broken into constituent phones, corresponding HMMs selected and concatenated– sentence HMM. Generates acoustic features which is fed to a vocoder for synthesis
- **Challenges with Handling Context**: Some combinations not available in the training data. Unseen combinations present in the test sentence
- **Advantages**: Low amount of training data, Small footprint size (few MBs), Fast synthesis, and Flexible, tune HMM parameters to vary speaking style, emotion
- Load the dataset first

- However, voice quality relatively poor.
- **Neural Network based TTS**: Learn the mapping between linguistic and acoustic feature vectors using neural network. It has better quality compared to HTS, however, requires more training data to produce good quality speech.
- **WaveNet**: An autoregressive model, `https://arxiv.org/abs/1609.03499`
- It has causal convolution, faster training because of no recurrent connections.

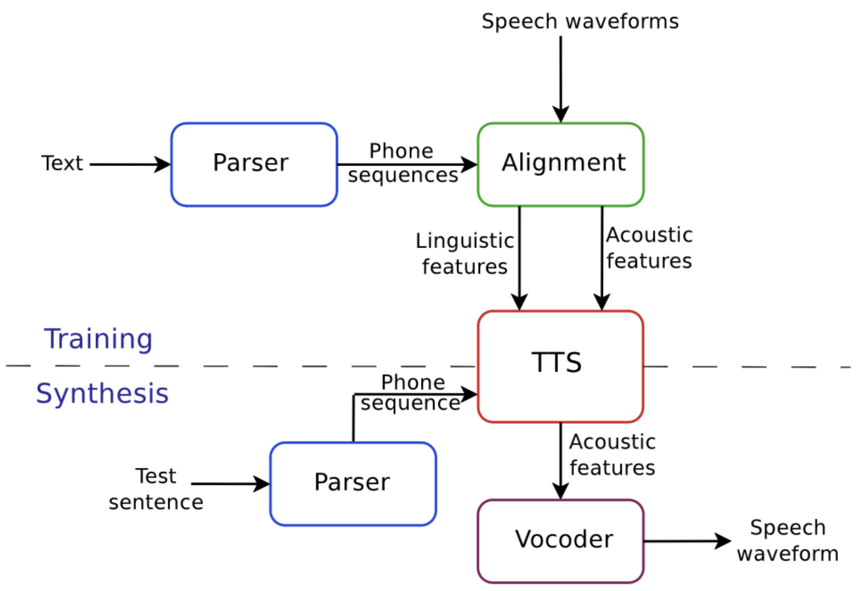

Figure 7: A complete TTS System

- **End-to-End synthesis**: Speech directly synthesized from characters. No need of developing separate modules (parsing, alignment). Allows rich conditioning on speaker, language, high-level features. Single model likely more robust than multi-stage model
- **Tacotron2**: Encoder and attention-based decoder and Vocoder, `https://arxiv.org/abs/1712.05884`
- **Issues with autoregressive models**: Synthesized speech is not robust has repetitions and word skips (error propagation, wrong attention alignments between text and speech), Slow inference speed, Lack of control over speed and prosody
- **FastSpeech**: Architecture based on Feed-forward Transformer, `https://arxiv.org/abs/1905.09263`
- **FastSpeech2**: `https://arxiv.org/abs/2006.04558`
- **VITS**: `https://arxiv.org/abs/2106.06103`
- **JETS**: `https://arxiv.org/abs/2203.16852`
- End-to-End gives high quality speech and is easy to train, however, requires a huge amount of training data and is computationally intensive
- **Neural Vocoders**: WaveNet, WaveGlow, Parallel WaveGAN, MelGAN, Multi-band MelGAN, HiFiGAN, StyleMelGAN
- **Objective measures**: Mel-cepstral distortion scores
- **Subjective measures**: Mean opinion score (MOS), Degradation mean opinion score (DMOS), Pairwise comparison (PC) test
- **Research Areas**: Multilingual aspects bilingual, code-mixing, Prosody expressive voice, emotional TTS, Voice conversion, Conversational speech
- **SpeechT5Processor**
  1. Converting text into speech waveforms.
  2. Supports multi-speaker synthesis using speaker embeddings.

   3. Extract tokenizer using `processor.tokenizer`

- **EncoderClassifier**
   1. Loads a speaker embedding model, which extracts speaker characteristics from audio.
   2. Used for multi-speaker TTS, where the model mimics a specific voice.
- **SpeechT5ForTextToSpeech**: Converts text input + speaker embedding into a speech waveform.
- **SpeechT5HifiGan**: SpeechT5 generates a mel spectrogram, which HiFi-GAN converts to waveform.
- Obtaining satisfactory results from this model on a new language can be challenging. The quality of the speaker embeddings can be a significant factor.
- If the synthesized speech sounds poor, try using a different speaker embedding.
- Increasing the training duration is also likely to enhance the quality of the results.
- Another thing to experiment with is the model's configuration. For example, try using $config.reduction\_factor = 1$ to see if this improves the results.
- Colab Notebook: `https://colab.research.google.com/drive/1uOxGeJxP95-YaSX9JK1z1aAm8im29sFR#scrollTo=6KcjGUttYUFE`

4 Computer Vision

4.1 Introduction

- **Goal**: To extract "meaning" from pixels
- **Semantic Information**: Object categorization, Scene and context categorization, Image Segmentation, Object Tracking, Pose Estimation, Image Captioning, etc.
- **Metric Information**: Depth from image, Structure from motion, photometric stereo etc.
- **Challenges**: Viewpoint variation, Illumination, Scale, Deformation, Occlusion, Background clutter, motion, Object intra-class variation.
- CNNs leverage Local connectivity, Parameter sharing, Pooling/ Subsampling, ReLu (rectifier) nonlinearity

4.2 Object Recognition

- **Image classification**: A core problem in computer vision
- AlexNet uses filters of size 11×11 in the initial layer. It increases the number of parameters that need to be trained
- Recent CNNs uses a cascade of small filters of size 3×3 or 5×5.
- VGGNet uses a stack of three 3×3 conv (stride 1) layers which has the same effective receptive field as one 7×7 conv layer, but deeper, more non-linearities.
- **Inception Module**: Design a good network topology(network within a network) and then stack these modules on top of each other.
- GoogleLeNet uses 1×1 conv layers which preserve spatial dimensions, but reduces depth.
- ResNet introduces residual layers, essentially and identity mapping on X aka skip connections.
- Image Classification Tutorial: `https://www.kaggle.com/code/advaitkisar/dlp-image-classification-tutorial`
- Image Classification using Pretrained Models: `https://www.kaggle.com/code/advaitkisar/dlp-pretrained-model`

4.3 Object Detection

- The task of assigning a label and a bounding box to all objects in the image
- We can do this by sliding a window in the image and then using an image classifier.
- Since there are too many region proposals, it is only good when classifier is fast.
- Convolutional Networks are computationally demanding, we cannot go through all the proposals.
- So, we simply pick "good" proposals, find "blobs" that are likely to contain objects. This is a "class-agnostic" approach.
- **RCNN**: `https://arxiv.org/abs/1311.2524`
  $\boxed{\text{Input Image}} \rightarrow \boxed{\text{Extract Region Proposals}} \rightarrow \boxed{\text{Compute CNN features}} \rightarrow \boxed{\text{Classify Regions}}$
- Ad hoc training objective
  1. Fine-tune network with softmax classifier (log loss)
  2. Train post-hoc linear SVMs (hinge loss)
  3. Train post-hoc bounding box regressions (least squares)
- Training is slow and takes a lot of disk space.
- **SPP Net**: `https://arxiv.org/abs/1406.4729`
- **Fast RCNN**: `https://arxiv.org/abs/1504.08083`
- Fast R-CNN improved SPP-Net by making it end-to-end trainable
- Forward entire image through a convolutional network to generate regions of interest, then use single level SPP to get region of interests.

- Then forward these to FFN, which then classifies the image, and predicts bounding box
- **Out-of-network region proposals**: We are proposing regions outside the main network instead of doing end-to-end prediction.
- RPN replaced Selective Search with a trainable proposal mechanism.
- **Faster RCNN**: `https://arxiv.org/abs/1506.01497`
- Solely based on CNNs, and each step is end-to-end.
- Use CNNs to get a region map, pass this region map to a region proposal network, and combine the output of RPN with this region map.
- RPN slides a small window in the feature map, a small model classifies between object or not object and regresses bounding box location.
- Position of the sliding window provides localization information with reference to the image.
- Box regression provides finer localization information with reference to this sliding window.
- The above approaches have a separate model for region proposal, need to run classification multiple times and looks at limited part of the image at a time(lacks contextual information).
- **YOLO**: `https://arxiv.org/abs/1506.02640`
- A single neural network for localization and classification, need to inference only once, and looks at entire image each time leading to less false positives.
    1. First, image is split into a $S \times S$ cells.
    2. For each cell, generate B bounding boxes
    3. For each bounding box, there are 5 predictions: x, y, w, h, confidence
    4. For each cell, we need to also do object classification.
    5. Each cell also predicts a class probability conditioned on object being present.
- **Mean Average Precision(mAP)**: measure of how well the detector is able to localize and classify the objects in a image. It is based on the concepts of Confusion Matrix, Intersection Over Union (IoU), Precision and Recall.
- Average precision is the area under the precision-recall curve.
- mean average precision is the mean of average precision across all classes.
- YOLO version 2
    1. **Anchor Boxes**: Introduced the concept of anchor boxes to improve object localization and handle varying aspect ratios. This allowed the model to predict bounding boxes more efficiently by utilizing predefined shapes.
    2. **Dimension Clusters**: Employed k-means clustering to calculate the most optimal set of anchor box sizes specific to the dataset. This improved precision by ensuring better alignment between anchor boxes and the objects in the dataset.
- YOLO version 5
    1. **AutoAnchor**: Introduced automatic anchor box generation, which adapts anchor boxes based on the dataset's distribution without requiring manual intervention. This further optimized the model's ability to detect objects of different sizes and shapes across various datasets.
    2. **Augmentation Strategies**: Incorporated Mosaic and MixUp data augmentation techniques to improve generalization and robustness during training. These augmentations enhanced model performance, particularly in scenarios with limited or highly varied data.
- RCNN Tutorial: `https://colab.research.google.com/drive/1l_XnSYjAZOFE8f5fDI2BhrteH-MikUeH`
- YOLO Tutorial: `https://colab.research.google.com/drive/1CDRH7n6rHYyTqas1w9hZe0LKWBK_6R-W`

4.4 Depth Estimation

- Estimate Pixel-level depth relative to camera position.
- **Applications**: Autonomous navigation, Virtual/Augmented Reality, Scene understanding
- **Depth from Stereo**: We can use two images taken from different angles to get the 3D reconstruction, and as such the depth.
- Depth estimation from a single image is difficult.
- Depth cues like parallax is missing
- Some cues that exist are vanishing points, object sizes etc
- These cues cannot be quantified through a rule based method
- **Multi-Scale Deep Network**: `https://arxiv.org/abs/1406.2283`
- Entire network is divided into 2 parts, coarse and fine network.
- Coarse network predicts the overall depth for the scene
- Fine network refines the global prediction locally
- Coarse network needs larger receptive field and is hence more deep compared to the fine network
- **UNet**: `https://arxiv.org/abs/1505.04597`
- This is also divided into 2 parts, Encoder Downsampling Network and Decoder Upsampling Network
- Encoder uses convolutional layers followed by max pooling to progressively reduce spatial dimensions while increasing the number of feature channels. Captures high-level features and context in the image.
- Decoder involves transposed convolutions (or upsampling layers) to increase the spatial dimensions of feature maps. Combines low-level features from the downsampling path with high-level features to produce detailed output depth maps.
- Also has skip connections, directly connect corresponding layers in the downsampling path to the upsampling path. Preserve spatial information lost during downsampling, allowing for more precise reconstructions in the output.
- skip connections allow fine-grained spatial details to be recovered
- MSE is commonly used in depth estimation, MAE can also be used.
- **Unsupervised Monocular Depth Estimation with Left-Right Consistency**: `https://arxiv.org/abs/1609.03677`
- Deep learning requires tons of training data for supervised learning. For depth estimation, you need the image depth map pairs.
- disparity maps are used to generate a second view from a single image.
- Usethe leftt image of the stereo pair to predict the disparity. True disparity is not available for training
- Now, transform the left image to right image using the predicted disparity. True right image is available and can be used for supervision
- At test time network predicts disparity map from only a single image
- UNet Depth Map Prediction: `https://colab.research.google.com/drive/1jlm3BX_Tgd5h_KpaCQOWACiCFCDHARx2`

4.5 Image Enhancement

- Most common image preprocessing tasks: Denoising, Super Resolution, Deblurring
- **Sources of Noise**: Photon shot noise, read noise, quantization noise, etc.
- **Multi-Stage Progressive Image Restoration**: `https://arxiv.org/abs/2102.02808`
- Learns the contextualized features using encoder-decoder architectures and later combines them with a high-resolution branch that retains local information.
- Multi-stage architecture progressively refines image quality across several stages.
- Parallel branches extractmultiscalee features for capturing both local and global information.
- Each stage outputs intermediate results, passed to the next for further enhancement.
- The encoder-decoder subnetwork is based on the standard U-Net architecture.
- It incorporates Channel Attention Blocks (CABs) for enhanced feature extraction at multiple scales.
- Bilinear upsampling followed by convolution is used to increase spatial resolution
- Channel attention map is for exploiting the inter-channel relationship of features.
- Channel attention focuses on 'what' is meaningful given an input image. To compute the channel attention, the spatial dimension of the input feature map is squeezed using average-pooling and max-pooling, generating two different spatial context descriptors.
- Both descriptors are then forwarded to a shared network to produce the channel attention map
- **Original Resolution Subnetwork (ORSNet)**: Preserves fine details in image restoration. Operates without downsampling, producing spatially-enriched high-resolution features. Composed of multiple Original-Resolution Blocks (ORBs), each containing Channel Attention Blocks (CABs).
- **Cross-stage Feature Fusion (CSFF) module**: CSFF is integrated between encoder-decoders and between the encoder-decoder and ORSNet. Features from one stage are refined using 1×1 convolutions before being passed to the next stage for aggregation. Reduces information loss, enhances feature enrichment across stages, and stabilizes the network optimization process, facilitating the addition of more stages.
- **Supervised Attention Module (SAM)**: Introduced between stages to improve performance in multi-stage image restoration networks. SAM provides ground-truth supervisory signals for progressive image restoration at each stage. It generates attention maps to suppress less informative features, allowing only useful features to propagate to the next stage.
- At any given stage S, instead of directly predicting a restored image X_s, the model predicts a residual image R_s to which the degraded input image I is added to obtain: $X_s = I + R_s$.
- The loss function is given as

  $$\begin{aligned} L &= \sum_{S=1}^{3} [L_{char}(X_s, Y) + \lambda L_{edge}(X_s, Y)] \\ L_{char} &= \sqrt{\|X_s - Y\|^2 + \epsilon^2} = \text{Charbonnier loss} \\ L_{edge} &= \sqrt{\|\Delta(X_s) - \Delta Y\|^2 + \epsilon^2} = \text{edge loss} \end{aligned}$$

  where Y is the ground truth, $\epsilon = 10^{-3}$ and $\lambda = 0.05$
- **Super-Resolution**: Be faithful to the low resolution input image, Produce a detailed, realistic output image
- Consider a image, we blur, downsample, and noise to this image, we want to recover the original image from this output.
- **Challenge**: Despite advancements in faster and deeper CNNs, recovering fine texture details during large upscaling is very difficult.
- Most methods focus on minimizing mean squared error, leading to high PSNR but lacking high-frequency details and perceptual quality.

- GAN drives the reconstruction towards the natural image manifold producing perceptually more convincing solutions.
- **Super-Resolution GAN**: `https://arxiv.org/abs/1609.04802`
- SRGAN is the first framework capable of generating photo-realistic images with 4$\times$ upscaling factors
- **Perceptual Loss Function**: Combines adversarial loss (trained to differentiate super-resolved images from real images) and content loss (based on perceptual similarity rather than pixel-wise similarity).
- Utilizes a **generator (G)** and **discriminator (D)** to solve an adversarial min-max problem, where G aims to fool D into classifying generated images as real.
- G generates realistic images while D tries to distinguish between super-resolved and real images, encouraging perceptually superior solutions over pixel-wise error methods like MSE.
- Generator Architecture: Built using a very deep network with B residual blocks, employing 3$\times$3 convolutional layers, batch normalization, and Parametric ReLU activations. Resolution is increased using Pixel Shuffle layers for high-quality super-resolution.
- **Pixel Shuffle**: Rearranges elements in a tensor of shape $(*, C \times r^2, H, W)$ to a tensor of shape $(*, C, H \times r, W \times r)$
- Discriminator Architecture: Consists of 8 convolutional layers, with 3$\times$3 filters increasing from 64 to 512, followed by two dense layers and a sigmoid activation for binary classification between real and super-resolved images. Uses Leaky ReLU ($\alpha = 0.2$) to maintain gradient flow for better learning. Replaces max-pooling with strided convolutions to preserve more spatial detail during downsampling.
- **Sources of blur**: Camera motion, object motion, defocus, Point spread function(PSF) or blur kernel.
- **LaKDNet**: `https://arxiv.org/abs/2302.02234`
- A pure CNN model (LaKDNet) to restore sharp, high-resolution images from blurry versions.
- Utilizes depth-wise convolution with large kernels to maintain efficiency while modeling long-range pixel dependencies.
- **Architecture Type**: U-shaped hierarchical network consisting of symmetric encoder-decoder modules.
- **Levels**: Comprises 4 levels, each containing N LaKD blocks, where N varies across levels (N_1, N_2, N_3, N_4).
- **Input Processing**: Takes an input image I with dimensions H$\times$W$\times$3 and extracts low-level features through an initial convolutional layer.
- **Feature Extraction**: Passes the features into the encoder-decoder structure for blur removal, followed by a convolution layer to recover output features.
- **Downsampling/Upsampling**: Utilizes pixel unshuffle for downsampling and pixel shuffle for upsampling.
- **Output Formation**: Implements skip connections from input I to output Iout, resulting in a global residual structure
- **LaKD Block Motivation**: Designed to explore local and global dependencies while achieving a large effective receptive field (ERF) using a fully convolutional approach.
- **Submodules**: Comprises two main submodules, feature mixer and feature fusion.
- **Feature Mixer**: Similar to depth-wise separable convolution, utilizing unusually large kernel sizes (e.g., 9$\times$9). Incorporates a point-wise convolution (1×1) Separately mixes spatial intra-channel and depth-wise inter-channel features, enabling distant spatial location mixture.
- **Normal convolution**: mixes spatial and channel information
- **Depth-wise convolution**: only mixes spatial information
- **Point-wise convolution**: only mixes channel information
- Lot of computation is saved by using depth-wise and point-wise convolution
- **Feature Fusion Model**: Utilizes depth-wise convolution layers with 3$\times$3 kernels for efficient local information encoding. Incorporates a gating mechanism with GELU activation to propagate and fuse features effectively.

- Denoising Tutorial: `https://drive.google.com/file/d/12KVwgV40RvLniAm2XiRFi8TJwb7_-95m/view?usp=drive_link`
- Super-Resolution: `https://drive.google.com/file/d/1eixeOQ2m3aOd4R0RUvYXqLXcSj_bvY3j/view?usp=drive_link`
- Deblurring Tutorial: `https://drive.google.com/file/d/1jhJ-MTl2sp0uEDHEoXk2lZcbDcJePj9s/view?usp=drive_link`