

Multimedia Retrieval

Chapter 14: Structural Features

Dr. Roger Weber, roger.weber@gmail.com

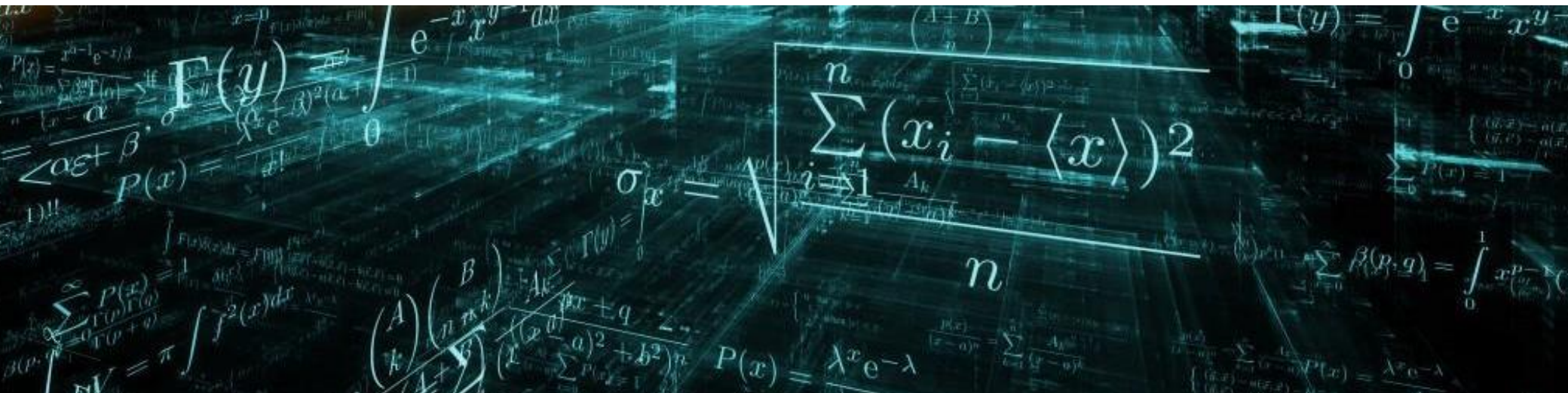
[14.1 Introduction](#)

[14.2 Text Features](#)

[14.3 Image Features](#)

[14.4 Audio Features](#)

[14.5 Literature and Links](#)



14.1 Introduction

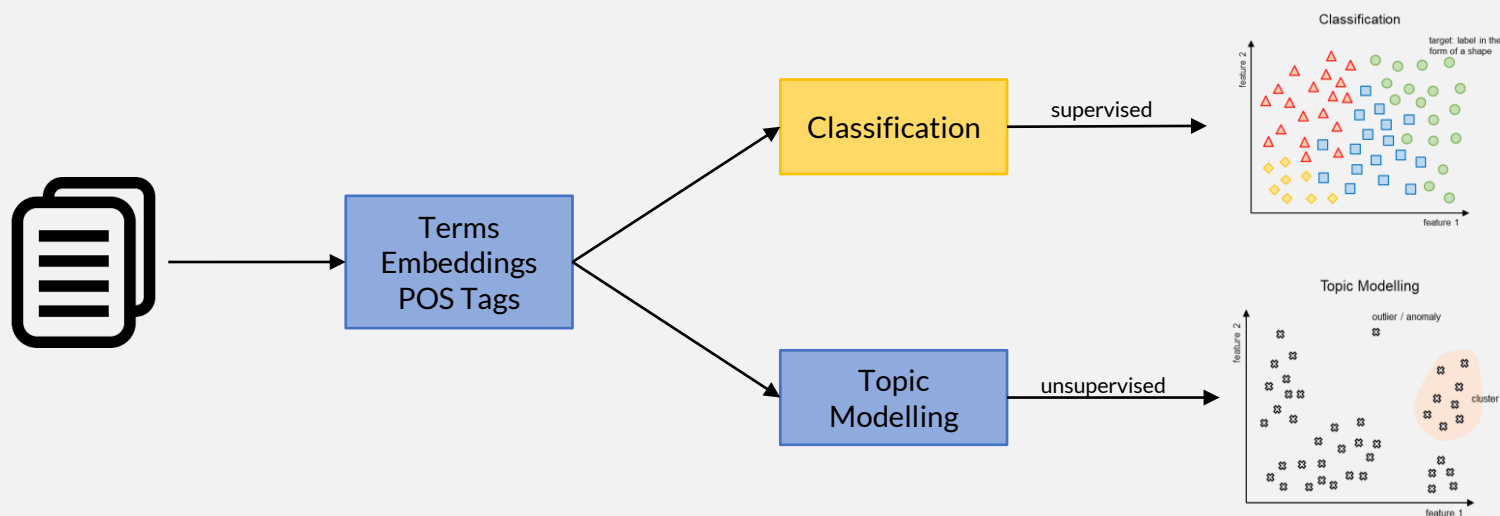
- In the previous chapters, we looked at perceptual features, the low-level signals that capture raw sensory data but often do not convey deeper meaning. This chapter introduces structural features, which help bridge the semantic gap by capturing richer patterns and relationships in data. We review applications in text, images, and audio, and show how modern machine learning methods use these features for more advanced understanding and classification.
- The key challenges that we address here are:
 - First, capturing the complex and varied nature of real-world data across different types. Structural features often appear as high-dimensional representations, so models must learn detailed patterns without overfitting. It is hard to match a model's capacity to the limited labeled data available, especially when training deep models that require large datasets and significant computing resources.
 - Another challenge is bridging the semantic gap itself, translating raw or low-level input signals into meaningful, abstract concepts that match human understanding. This requires designing models that capture context, dependencies, and hierarchical relationships like those in language or visual scenes.
 - Additionally, multimodal learning introduces the complexity of integrating heterogeneous data types like images and text, which often differ in structure and scale. Ensuring these models align information across modalities in a coherent way requires careful design of attention mechanisms and training objectives.
 - Finally, training these advanced models poses challenges in optimization stability, efficient use of resources, and avoiding problems like vanishing gradients and catastrophic forgetting during fine tuning. These factors make learning features difficult but essential for improving an AI's ability to understand and interpret diverse data.
- This field is changing fast because powerful architectures such as transformers have reshaped machine learning. Multimodal transformers that combine images, audio, video, and text are replacing older approaches that used separate, specialized methods. This chapter traces the path from early neural networks to today's top models and offers historical context and practical insight into the tools and techniques behind recent AI advances.

- Several breakthroughs have paved the way to the current state of the art by addressing these challenges.
 - Training deep neural networks became practical after the introduction of activation functions like ReLU. These functions helped reduce the vanishing and exploding gradient problems that had long blocked learning in deep models. Combined with large labeled datasets and powerful GPUs, this advance let models scale up and achieve much better performance on complex tasks.
 - Convolutional neural networks transformed image processing by using spatial structure, shared weights, and local connections to cut the number of parameters and improve generalization. For sequential data such as text and audio, recurrent neural networks and later transformers introduced methods to capture long-range dependencies and context more effectively.
 - Regularization methods such as dropout, data augmentation, and weight penalties reduced overfitting in large models. Improvements in optimization algorithms like Adam increased training stability and sped up convergence, making it possible to train deeper, more complex networks.
 - Flexible architectures like LeNet and ResNet made architectural choices part of the learning process. ResNet added residual connections that let gradients flow back to earlier layers, helping train very deep networks. These residual links, together with regularization, help the model find the right level of complexity and balance overfitting and underfitting without human intervention.
 - Transformers were a major advance because they replaced recurrence with self-attention, letting models process whole sequences in parallel and learn richer, global relationships. The architecture proved useful across data types and tasks, simplifying model design and improving performance.
 - Multimodal transformers combined these advances by using cross attention to integrate images and text, creating unified representations and allowing end to end training. This change has overturned traditional approaches and produced powerful models that can understand and generate content across many domains.
 - A recent advance is using self-supervised learning on large, weakly labeled datasets to extract useful semantic features without heavy manual annotation. Self-supervised objectives, such as masked prediction, clustering, and mutual information maximization, help models learn robust representations even when labeled data are scarce.
 - Separately, contrastive learning has become a powerful method for aligning different types of data, such as images and text, in a shared embedding space. Cross-modal contrastive learning trains models to bring related samples closer together while preserving differences that belong to each data type. This alignment helps models learn both the information shared across modalities and the information unique to each one.

- How do the methods in this chapter close the semantic gap?
 - They reduce the semantic gap by moving beyond raw sensory signals to capture richer, more abstract data representations. Using labels, keywords, or converting speech to text clearly reduces the semantic gap. Text aligns more closely with human language and meaning, so turning audio into text or tagging images with descriptions lets retrieval systems work with richer, symbolic information. This makes search simpler by converting different data types into a single, more understandable form where meaning is explicit.
 - These labels and text features serve as anchors, bridging different data types by giving them a shared semantic basis. This reduces the need to interpret raw sensory data directly and lets retrieval methods use established natural language processing techniques, which handle meaning better than purely perceptual features.
 - Structural features learned by deep neural networks capture patterns, context, and relationships that match human understanding more closely. Rather than treating inputs as isolated pixels, sound waves, or words, modern models detect how elements interact within and across modalities, revealing underlying meaning and intent.
 - For retrieval tasks this lets systems move from simple keyword or feature matching to concept-based searches. By encoding images, text, and audio into a shared embedding space, often using multimodal transformers and contrastive learning, retrieval models can compare items by meaning rather than by surface features. For example, a natural language query can find relevant images or audio clips by conceptual content rather than by exact wording or visual patterns.

14.2 Text Features

- Topic modeling and clustering differs from text classification. The distinction lies in the approach: classification relies on supervised learning with predefined classes, learning how features align with these classes. In contrast, topic modeling and clustering is unsupervised, aiming to detect clusters or co-occurrences of terms within text documents, and assigning them topic labels (as illustrated in the figure below). LSI also employs unsupervised techniques to learn topics through singular value decomposition and thereby reduces the rank of the document-term matrix. While this process shares similarities with topic modeling methods like Latent Dirichlet Allocation (LDA) and Non-Negative Matrix Factorization (NMF), LSI's primary objective is not to extract and explain topics found within the collection. Instead, it leverages these abstract topics for semantic retrieval.
- While deep learning can handle various tasks, we should also explore cost-effective methods that provide satisfactory solutions. Training complex language models can be resource-intensive, whereas simpler techniques can yield comparable results, particularly when term occurrences is the dominating factor for class assignments.
- In the following, we review efficient methods that continue to deliver excellent results. Many remain in use to offer state-of-the-art structural features for particular use cases.



14.2.1 Language Detection

- Language detection is the problem of determining the language in a text or document. This task is rather simple for long documents, but can become quite challenging for short texts or when a large number of languages have to be detected automatically. A related problem: detecting programming languages.
- Let's start with the simple method of detecting languages in longer texts. The most efficient approach is to apply a number of rules to detect the language:
 - **Alphabet Diversity:** Each language has unique characters found in only a few related languages. Examples include Latin, Cyrillic, Greek, Arabic, Hebrew, Devanagari, Thai, Tamil, Bengali scripts, as well as Chinese, Hiragana, and Katakana characters. Languages often borrow words from others, leading to a mix of alphabets. To handle this, we can filter out rarely used alphabets.
 - **Character Diversity:** Some languages have special characters within an alphabet that are typical of their linguistic uniqueness. For instance, diacritical marks and accent symbols in Latin-based scripts, or tonal markers in certain Asian languages, add distinctive features to characters. Only a few Latin based languages use ä, ö, and ü.
 - **Stop word Counts:** Evaluating stop word frequencies in text can reveal a language. For instance, languages like English and French often employ frequent stop words, while others, like Mandarin Chinese, rely less on them. Using managed stop word lists allows us to guess a language simply by counting how often its stop words occurs.
 - **Vocabulary Counts:** Examining the unique words or vocabulary in a text can also help identify the language. Different languages have distinct vocabularies, and by comparing word frequencies and diversity, it becomes possible to make language determinations with a degree of accuracy.
- For longer texts, these rules quickly lead to the identification of the language. However, the method does not easily scale to large numbers of languages. The alphabet and character rules are simple, but the stop word lists and vocabularies (most frequent words) require large amount of data to perform language detection.
- For shorter texts or brief phrases, these methods are less effective unless we have comprehensive vocabularies for all languages. In some cases, it may be challenging to identify the correct language, as a single word or short phrase can exist in multiple languages. Even more complex are phrases containing loanwords from other languages, such as IT terms in a German phrase like "mein computer".

- Modern language detectors operate at the sub-word level and incorporate rules like those mentioned earlier, such as Alphabet and Character rules. Additionally, they introduce new rules based on character-based n-grams that are specific to certain languages. The key distinction of these new methods, however, lies in utilizing the frequencies of character-based n-grams for language detection, and this is achieved using a Naïve Bayes learning model.
- Naïve Bayes employs a conditional probability model based on Bayes' theorem.

$$P(C_k|\mathbf{x}) = \frac{P(\mathbf{x}|C_k) \cdot P(C_k)}{P(\mathbf{x})} \quad \text{posterior} = \frac{\text{likelihood} \cdot \text{prior}}{\text{evidence}}$$

In this equation, \mathbf{x} represents a feature vector, and C_k is the class or target. $P(C_k)$ is the **prior**, that is knowledge about the distribution (probability) of classes C_k . $P(\mathbf{x}|C_k)$ is the **likelihood** of observing feature \mathbf{x} for a specific class C_k , and $P(\mathbf{x})$ is the overall **evidence** of observing \mathbf{x} , regardless of class. $P(C_k|\mathbf{x})$ represents the **posterior** which is the knowledge gained or predicted when observing feature \mathbf{x} , allowing us to infer its association with class C_k .

- Consider \mathbf{x} as a high-dimensional vector, often derived from a vast term space used in documents. Given the high dimensionality and the restricted training data, accurately modeling the probability distribution function in this sparse space is challenging. To simplify, naïve Bayes assumes conditional independence among features, resulting in the following simplification:

$$P(C_k|\mathbf{x}) = P(C_k|x_1, \dots, x_M) = \frac{1}{P(\mathbf{x})} \cdot P(C_k) \cdot \prod_{j=1}^M P(x_j|C_k)$$

Note that $P(\mathbf{x})$ is a constant across classes C_k and only scales the probabilities. For our purposes, we do not require its value

- Using the probability model, we choose the most probable hypothesis, that is class C_{k^*} that maximizes the probability function. This selection principle is commonly referred to as **maximum a posteriori (MAP)**:

$$k^* = \operatorname{argmax}_k P(C_k|\mathbf{x}) = \operatorname{argmax}_k P(C_k) \cdot \prod_{j=1}^M P(x_j|C_k)$$

That is it! The equation describes the decision rule of Naïve Bayes. The only thing left are the estimates for the probabilities on the right hand side

- Now, we need to estimate the probabilities $P(C_k)$ and $P(x_j|C_k)$ based on observations from the training set.

- In our language detection scenario, we use character-based n-grams of varying lengths (e.g., n from 1 to 5). We count how often these n-grams appear in the text, resulting in a bag-of-words representation that forms a multinomial distribution. The feature vector \mathbf{x} represents these counts for a defined vocabulary for each language.
- The priors $P(C_k)$ depend on the scenario: we can use a maximum likelihood estimator based on observations in the training set. Let N_k be the number of texts for the language denoted by class C_k , and N be the total number of texts:

$$P(C_k) = \frac{N_k}{N} \quad \text{or if } N_k \text{ is not known: } P(C_k) = \frac{1}{K}$$

If we lack knowledge of the language distribution or wish to avoid training bias, we can select a constant prior for all classes, which can then be omitted from subsequent calculations since it only scales posteriors for all classes.

- To estimate the likelihoods $P(x_j | C_k)$ from texts in a language represented by class C_k , we count the n-gram occurrences in the training data for that language (**multinomial distribution**). For each language, we establish first an appropriate vocabulary, using methods similar to word-pieces or BPE, to control vocabulary size. We prioritize the most frequent n-grams since they have the most influence on the posterior and impact language determination the most. Let $n_{k,j}$ denote the total occurrences of n-gram t_j in all training texts for the language represented by class C_k :

$$p_{k,j} = \frac{n_{k,j}}{\sum_l n_{k,l}} \quad \text{or smoothed: } p_{k,j} = \frac{n_{k,j} + 1}{\sum_l n_{k,l} + M}$$

As we choose the vocabulary tailored to the target language and exclude infrequent or absent n-grams from the test set, we do not require the “+1” smoothing on the right-hand side. However, in other text classification tasks, smoothing prevents $p_{k,j}$ from reaching 0 for rare tokens during predictions (which leads to a posterior of value 0).

- Finally, we can predict the language based on posteriors. Instead of multiplying probabilities as shown on the previous page, we rather use sums over log-probabilities:

$$k^* = \operatorname{argmax}_k P(C_k | \mathbf{x}) = \operatorname{argmax}_k \left(\log P(C_k) + \sum_{x_j > 0} x_j \log p_{k,j} \right)$$

We can obtain scores with a softmax classifier over the target languages, and select the language with highest score.

- **Examples:** The [lingua-language-detector](#) is a highly efficient language detector with over 99% accuracy for more than 70 languages. Let's explore its functionality through examples.

```
from lingua import Language, LanguageDetectorBuilder
detector = LanguageDetectorBuilder.from_all_languages().build()
detector.detect_language_of("This is an example sentence")           # → Language.ENGLISH
detector.detect_language_of("Je suis un exemple de phrase")         # → Language.FRENCH
detector.detect_language_of("นี่คือข้อความที่ทดสอบ")                 # → Language.THAI
```

- We can also inquire about the likelihood of a phrase belonging to a particular set of languages:

```
languages = [Language.ENGLISH, Language.FRENCH, Language.ITALIAN]
detector = LanguageDetectorBuilder.from_languages(*languages).build()
detector.compute_language_confidence_values("Je suis à New York")
↳ FRENCH: 0.45 ENGLISH: 0.37 ITALIAN: 0.18
```

- The detector also are able to predict the languages out of fragments:

```
confidence_values = detector.compute_language_confidence_values("hau mei")
↳ GERMAN: 0.82 ENGLISH: 0.10 ITALIAN: 0.07
```

- This also demonstrates that the detector operates at sub-word levels. The 3-grams “hau” and “mei” are more common in German texts than in English and Italian, resulting in higher confidence scores.

- Another Python library is [langdetect](#), which is also a rule and n-grams based language detector for 55 languages. It provides ISO-codes for the detected languages:

```
from langdetect import detect, detect_langs
detect("This is an example sentence")           # → en
detect("je suis un exemple de phrase")         # → fr
detect("Este es un ejemplo de frase")         # → es
detect("Dies ist ein Beispieltext")           # → de
detect("นี่คือข้อความที่ทดสอบ")                 # → th
detect("Questo è un esempio di frase")       # → it
detect(" ")                                    # → th
detect_langs("Je suis à New York")           # → [fr:0.86, en: 0.14]
```

14.2.2 Sentiment Analysis

- Sentiment analysis deciphers human language to understand emotions and opinions. It's widely used to assess customer sentiment from reviews, social media, and support cases, aiding data-driven decisions for product improvement and customer satisfaction. It also can help to filter and moderate user-generated content to safeguard brand reputation and enforce community guidelines.
- In sentiment analysis, a fundamental task is to classify text polarity, identifying if it is positive, negative, or neutral. Advanced tasks can recognize emotions like anger, fear, disgust, joy, and surprise. Here, we focus on basic sentiment prediction with positive, negative, and neutral categories. Let's start with a look at some example, and the challenges machine learning models may encompass:
 - Many statements are straightforward and sentiment is often driven by a few key words:

"I like this product"	→ positive
"I was going to the town"	→ neutral
"The food was really bad"	→ negative
 - However, we can express ourselves in many, sometimes confusing ways that are difficult to analyze:

"I can't say I liked it"	→ negation handling
"Drinking wine is not my thing"	→ negative or neutral?
"What a fine artist you've become!"	→ potentially sarcastic
"I haven't ever owed anything to anyone"	→ lots of negation, but actually positive
- We consider 3 different approaches:
 - Naïve Bayes with the example of sentiment analysis in Twitter (now called X)
 - TextCNN, a convolutional network on embeddings to predict classes
 - Transformer based classification models

- **Naïve Bayes** is popular for its simplicity, speed, and accuracy. We used it before for language detection with a multinomial distribution, considering term presence and counts. In Twitter sentiment analysis, short texts mean terms usually occur only once, except for stop words. We use a set-of-word representation and assume a multivariate Bernoulli distribution for likelihood estimation. This examples uses two classes: positive and negative
 - The priors $P(C_k)$ measure how likely a messages fall into one of the two classes (positive, negative). We can use a maximum likelihood estimator based on observations in the training set. Let N_k be the number tweets for class C_k , and N be the total number of texts (with $k = \text{'positive'}$ or 'negative'):

$$P(C_k) = \frac{N_k}{N} \quad \text{or if } N_k \text{ is not known: } P(C_k) = \frac{1}{K}$$

If we do not know the sentiment distribution or wish to avoid training bias, we can select a constant prior for all classes, which can then be omitted from subsequent calculations since it only scales posteriors for all classes.

- Assuming a multivariate Bernoulli distribution for the set-of-word representations, we can estimate the likelihoods $P(x_j|C_k)$ as follows. Let $N_k(x_j = 1)$ denote the number of messages from C_k that contain a term t_j :

$$p_{k,j} = \frac{N_k(x_j = 1)}{N_k} \quad \text{or smoothed: } p_{k,j} = \frac{\min(N_k - 1, \max(1, N_k(x_j = 1)))}{N_k}$$

We can use either smoothing to prevent $p_{k,j} = 0$ if a term t_j does not occur in the messages of class C_k , or we simply ignore terms that were not present in the training data of class C_k during predictions.

- Finally, we can predict the sentiment ('positive' or 'negative' class) based on posteriors. Instead of multiplying probabilities, we again use sums over log-probabilities (and ignore terms with $p_{k,j} = 0$):

$$k^* = \operatorname{argmax}_k P(C_k|\mathbf{x}) = \operatorname{argmax}_k \left(\log P(C_k) + \sum_{j=1}^M (x_j \log p_{k,j} + (1 - x_j) \log(1 - p_{k,j})) \right)$$

- Instead of using the entire vocabulary, we can reduce features by selecting the most informative terms.

The code on the right hand side shows the sentiment analysis implementation:

- 1) We utilize the Twitter samples data from the `nlTK` corpus, consisting of 5,000 positive and 5,000 negative labeled tweets. These tweets are read and labeled accordingly. Additionally, we create a list of stop words, create a Snowball stemmer, and utilize a tweet tokenizer that recognizes Twitter-specific tokens like hashtags, user tags, and emoticons.
- 2) In the process of cleaning the tweets, we eliminate HTTP links and user tags, as they are not relevant for sentiment analysis. We employ the Twitter tokenizer and remove single-letter tokens, numbers, and stop words. However, we retain emoticons like “:-)” since they can carry sentiment information.
- 3) We divide the training and test data into an 80:20 ratio while ensuring an even distribution of positive and negative tweets in both the training and test subsets through stratification.
- 4) We obtain a classifier from the `nlTK` Naïve Bayes training and then assess its training accuracy (0.999) and test accuracy (0.995). The Naïve Bayes classifier makes only 16 incorrect predictions out of 10,000 samples. Some of the most informative features for this classifier are “:)” and “:(”, among others.

In this scenario, Naïve Bayes is not only highly accurate but also remarkably fast, with training and classification taking less than a second. None of the deep learning methods can compete with this speed.

```
# 1) get started with data and settings
tweets = [(t, "pos") for t in twitter_samples.strings("pos...")] + \
          [(t, "neg") for t in twitter_samples.strings("neg...")]
stopwords = nltk.corpus.stopwords.words('english')
stemmer = nltk.stem.SnowballStemmer('english')
tokenizer = nltk.tokenize.casual.TweetTokenizer()

# 2) cleaning all the tweets --> set of words model
def set_of_words(text):
    text = re.sub(HTTP_REGEX, '', text)
    text = re.sub("@([A-Za-z0-9_]+)", "", text)
    tokens = tokenizer.tokenize(text)
    tokens = [stemmer.stem(t) for t in tokens
              if len(t)>1 and
              not t.isnumeric() and
              t not in stopwords]

    return {t:1 for t in tokens}

data = [(set_of_words(text), label) for text, label in tweets]

# 3) split training and test (stratify pos/neg samples)
pos_data = [x for x in data if x[1] == 'pos']
neg_data = [x for x in data if x[1] == 'neg']
pos_split = 80 * len(pos_data) // 100
neg_split = 80 * len(neg_data) // 100

train_data = pos_data[:pos_split] + neg_data[:neg_split]
test_data = pos_data[pos_split:] + neg_data[neg_split:]

# 4) classify with Naive Bayes (bernoulli)
classifier = nltk.NaiveBayesClassifier.train(train_data)

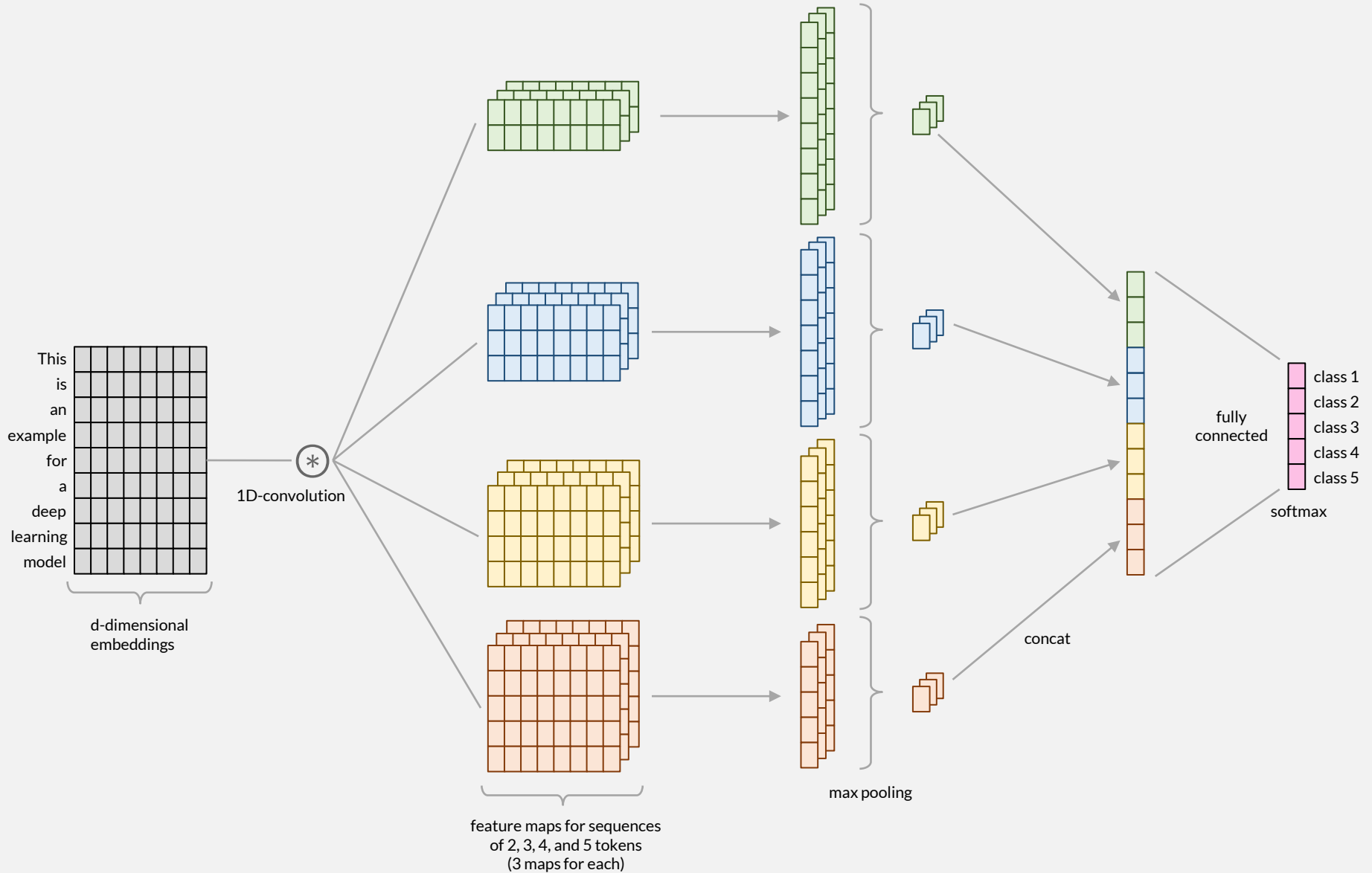
print(nltk.classify.accuracy(classifier, train_data))
print(nltk.classify.accuracy(classifier, test_data))
print(classifier.show_most_informative_features(10))

false_predictions = [t for t in train_data
                     if classifier.classify(t[0]) != t[1]]
false_predictions += [t for t in test_data
                     if classifier.classify(t[0]) != t[1]]
```

14.2.3 Text Classification with Deep Learning

- Sentiment analysis falls under text classification, and Naïve Bayes methods can be expanded to handle broader classification tasks. In this section, we explore the application of deep learning techniques to tackle more complex classification challenges. It is essential to begin by discussing the fundamental differences beforehand:
 - Naïve Bayes is a straightforward, yet highly effective and efficient method capable of real-time classification with low resource demands. Training and re-training are quick and straightforward, and model parameters occupy minimal storage. Storage consumption and performance can be further enhanced by selecting a subset of the most informative features. Consequently, Naïve Bayes, along with other simple classifiers like XGBoost or SVM, serves as an excellent initial choice. More advanced methods should only be considered when they can substantiate increased resource requirements with significantly higher accuracy.
 - Consider the sentiment analysis results from previous sections. Naïve Bayes achieves high accuracy, scoring 0.995 with only 16 false predictions out of 10,000 samples. While theoretically, we could opt for a deep learning approach like a transformer-based sentiment analyzer, such a model would not classify tweets as quickly as Naïve Bayes. In fact, a basic RoBERTa model takes seconds to minutes for classifying 10,000 samples (dependent on available hardware). Even if it achieved perfect accuracy (100%), the enhanced quality would not justify the significantly greater resource requirements.
 - Simpler models like Naïve Bayes rely on the independence assumption. In many complex scenarios, this assumption does not hold, leading to a rapid decline in the performance of simple models. While we can employ lemmatization techniques to enhance quality, these models cannot capture dependencies. On the other hand, deep learning models can adapt to complex scenarios and are versatile enough to handle various classification tasks without requiring substantial architectural changes, or additional training for new labels.
- In this section, we examine the architecture of TextCNN and transformer-based classification architectures which offer distinct approaches to text classification. TextCNN utilizes convolutional layers to extract features from text embeddings, making it effective for capturing local patterns in data. In contrast, transformers excel in handling long-range dependencies through self-attention mechanisms, making them ideal for tasks requiring a broader context understanding. While textCNN is computationally efficient and interpretable, transformers are highly flexible and excel in tasks demanding nuanced contextual understanding. The choice between the two depends on the specific requirements of the classification problem, with textCNN being suitable for simpler tasks, and transformers shining in more complex, context-sensitive scenarios.

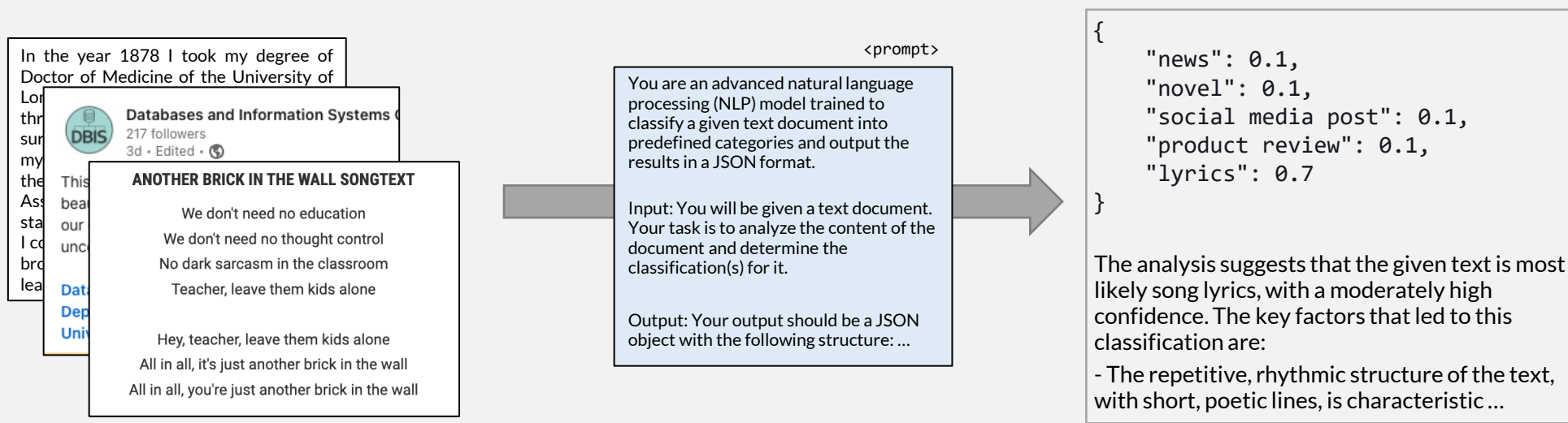
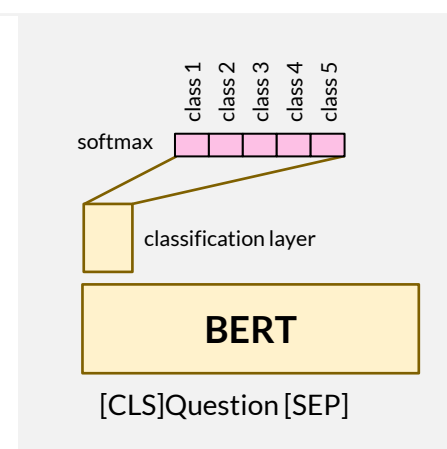
TextCNN-Architecture



• TextCNN Architecture

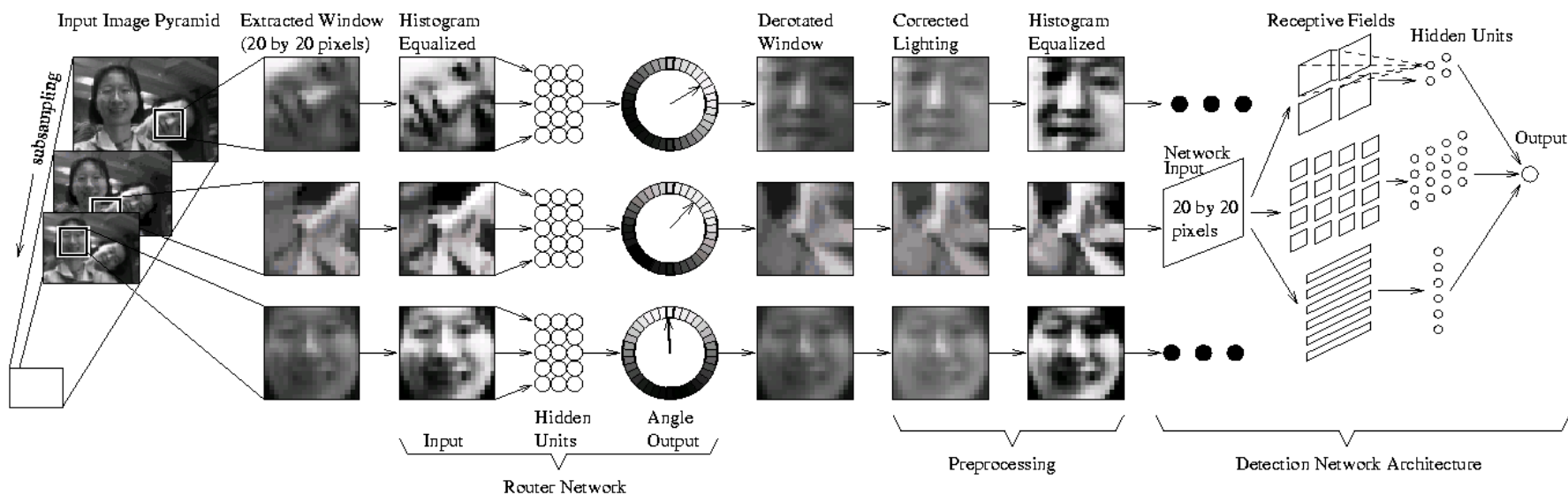
- Tokens are converted into d -dimensional embedding vectors and fed into the network. Unlike the transformers architecture, the sequence length is treated as an input dimension, not an architectural parameter. This allows us to handle sequences of arbitrary length and apply convolutions to both short and long sentences without the need for padding. We can select any method and dimensionality for the embeddings.
- We can utilize a set of feature maps to perform 1D convolutions on the sequence of embeddings. A feature map consists of weights of size $n \times d \times m$, where n represents the number of consecutive embeddings in the sequence window, d is the embedding dimensionality, and m denotes the number of output values from the convolution. The feature map traverses the sequence, applying 1D convolution to the next n embeddings, adding a bias, and applying an activation function for an output value. With m feature maps, we compute m output values for each position. With a sequence length of s , this results in $s - n + 1$ values for each of the m feature maps.
- As the input sequence can vary in length, the next step employs max pooling to condense the $s - n + 1$ values from each feature map into a single value. These resulting values are then concatenated into a vector of fixed length. In this example, we utilized feature maps of dimensions $2 \times d \times 3$, $3 \times d \times 3$, $4 \times d \times 3$, and $5 \times d \times 3$, resulting in a concatenated feature vector of dimensions $4 * 3 = 12$ as the output of the convolutional layer.
- A fully connected network translates this 12-dimensional vector into k logits (in our example, $k = 5$) and then applies a softmax function to predict the text's associated class.

- **Transformer-based classification** leverages pre-trained transformer models like BERT or GPT as the core, extending them with extra layers to make class predictions. Typically, in transformer-based classification, we initiate a sequence with a model-specific token (e.g., [CLS] for BERT) and utilize the corresponding encoder output vector. This vector is then passed through a deep classification layer, which computes the logits for the k classes related to the task. A softmax function is applied to determine the class to which the input text belongs.
- There are two ways to train the model for a given classification task:
 - The base transformer model (also called foundation model) is frozen and we only train the parameters of the additional classification layers. The foundation model can be shared across various classification tasks.
 - Both the base transformer model and the classification layer are trained together. This leads to a fine-tuned foundation model optimized for the classification task, but requires separate models for each classification task.
- **Modern large language models** can easily extract classification information from text documents through prompt engineering. By providing a text document, the prompt asks the language model to extract the desired classes in a specific output format, like JSON.



14.3 Image Features

- Rowley, Baluja, and Kanade (1998) from Carnegie Mellon University developed an elaborated method for finding faces of different sizes and angles. They kept the neural network small by first teaching it to recognize standard faces, and then searching through images for faces. The detection network uses a 20x20 input network (preprocessed image window). In the first layer, they created three types of receptive fields:
 - 4 times 10x10 areas,
 - 16 times 5x5 areas, and
 - 6 times overlapping 20x5 areasEach area is fully connected to a hidden unit, which is then fully connected to an output. An output of 1 means a face is present, and an output of -1 means no face is present.
- A second network (router network) was trained to predict the direction of a face within a window. The 20x20 input network (preprocessed image window) is connected to hidden units, which are then connected to 36 output values representing an angle. This angle can be used to adjust the face before using the detection network.
- During inference, the system runs multiple times using a sliding window technique and different scales to detect faces that are larger or smaller than the standard 20x20 faces used for training.



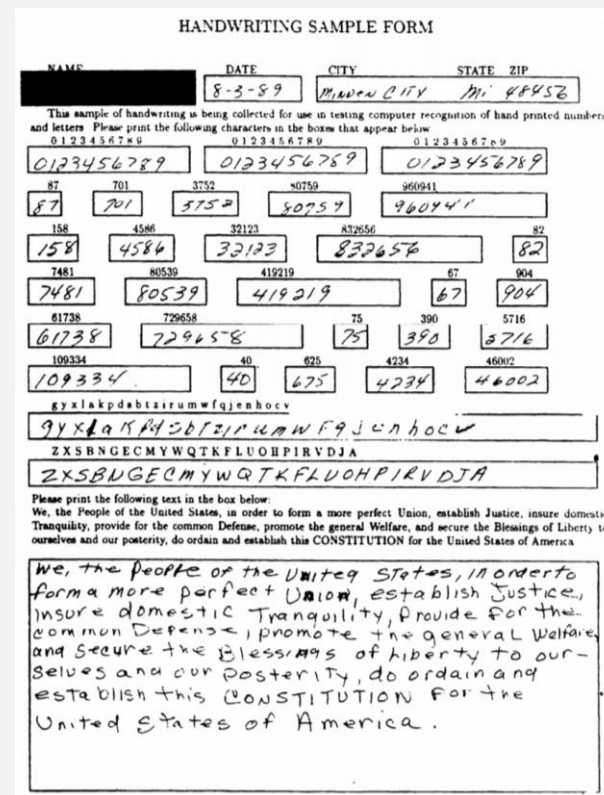
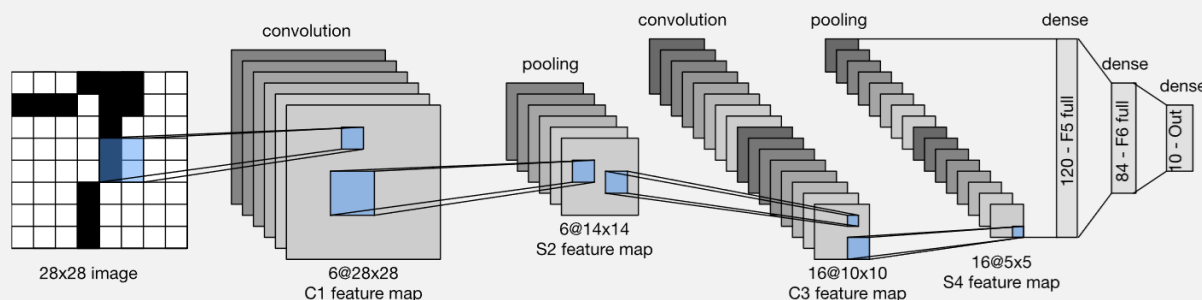
- After training, we can locate faces in an image by following these steps: first, we create a pyramid of images by making them smaller and smaller. This helps us find faces of various sizes. Then, a 20x20 window moves across the image, and for each position, the network checks if the window has a face. Because we use normalized faces, the algorithm can identify the location and orientation of faces, and estimate the position of the eyes.



- While the method performs well, the need for human intervention, such as normalizing, selecting window size, rotating, equalizing histograms, and using a sliding window approach, makes it challenging to use the network for other tasks, like identifying cats or dogs. Early neural network classifiers were often only optimized for one specific task (in this case, recognizing faces) and couldn't adapt to new situations without more human involvement.
- The second wave of neural network research quickly dwindled due to fundamental issues in the learning algorithm. Despite the theoretical capacity of neural networks to learn any function, this often didn't translate into practical success. Adding more hidden layers didn't necessarily improve results, and larger networks became increasingly unstable. The challenges of vanishing and exploding gradients and the competition from support vector machines (SVM) with sophisticated kernels led the field into a deadlock. Only the Canadian government continued to fund neural network research, with Geoff Hinton and his team publishing a breakthrough paper in 2006 on deep belief networks that addressed early backpropagation issues. Simultaneously, the availability of large labeled datasets and the parallel processing power of GPUs significantly accelerated the success of what is now known as deep learning.
- The breakthrough moment for deep learning was a result of several key factors and developments. It began with the availability of large labeled datasets like ImageNet, which allowed deep learning models to learn from extensive data. This was further empowered by the increased computational power, particularly the use of GPUs, which made training large neural networks efficient. Advanced architectures, such as convolutional neural networks (CNNs) and recurrent neural networks (RNNs), greatly improved model performance, while innovative activation functions like ReLU helped mitigate the vanishing gradient problem. Regularization techniques, including dropout and L1/L2 regularization, enhanced model generalization, and optimization algorithms like Adam and RMSprop made training more efficient. Transfer learning, where pre-trained models are fine-tuned for specific tasks, accelerated model development. Pioneering research, industry investment, and remarkable success in diverse applications contributed to the resurgence of neural networks, marking a significant breakthrough in artificial intelligence.
- See the online neural network playground at <http://playground.tensorflow.org/> to experiment with the limitation of early multi-layer networks.

14.3.1 Deep Learning Architecture

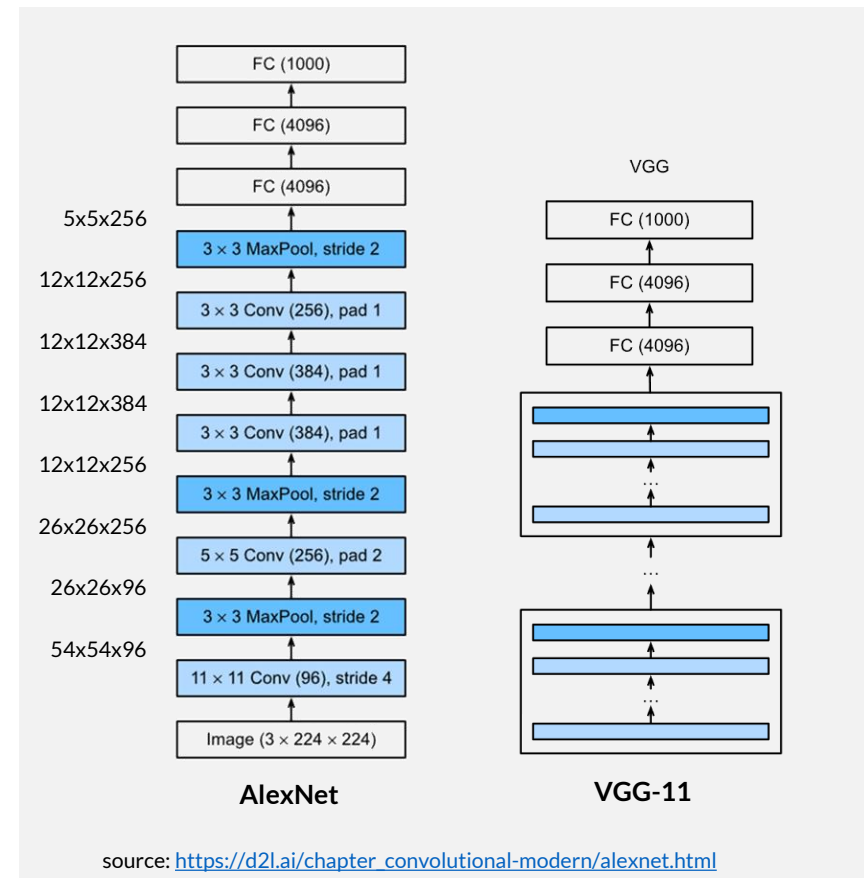
- Even though CNNs became well-known in the computer vision and machine learning communities after LeNet was introduced in 1995, they didn't immediately become the dominant method. While LeNet showed good results on small early datasets, it wasn't clear if CNNs could perform well on larger, more realistic datasets. In fact, from the early 1990s until the breakthrough results of 2010s, neural networks were often outperformed by other machine learning methods like support vector machines.
- Now, let's take a look at LeNet (1995). One of the first challenges in image classification is the recognition of handwritten digits in the MNIST database, which contains tens of thousands of 28x28 samples. Each digit is normalized in the 28x28 bounding box and anti-aliased, which introduced grayscale levels.
- The LeNet structure uses the architecture shown below with the sigmoid activation function. It achieved 98.7% accuracy on the MNIST database without human intervention. Newer models now achieve 99.9% accuracy with deep learning improvements. Let's examine how the architecture has evolved over time.



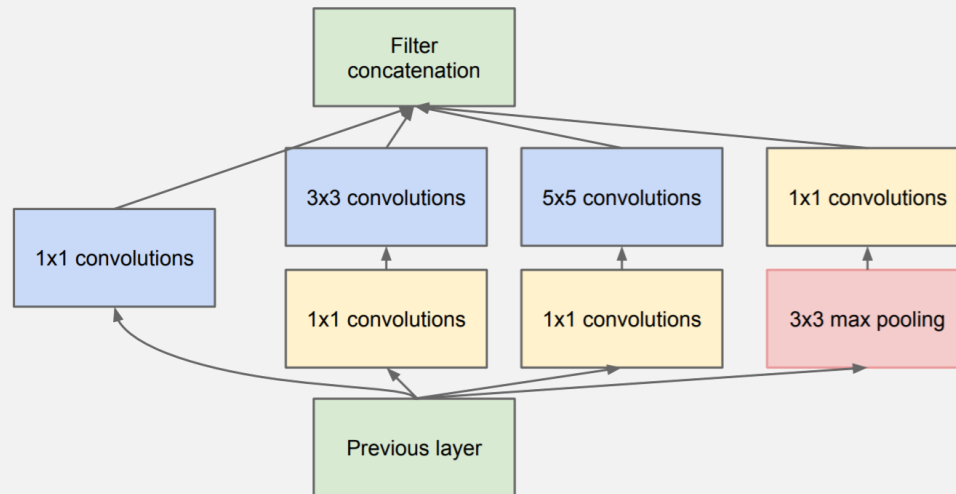
Visit
<https://cs.stanford.edu/people/karpathy/convnetjs/demo/mnist.html>
 for an online visualization of the network learning for MNIST

source: https://d2l.ai/chapter_convolutional-neural-networks/lenet.html#lenet

- AlexNet (2012), which used an 8-layer CNN, won the 2012 ImageNet Large Scale Visual Recognition Challenge. It was the first time, that a neural network demonstrated that learned features can surpass manually-designed features, changing the previous approach in computer vision. The architectures of AlexNet and LeNet are very similar, as shown in the figure below.
 - AlexNet is deeper than LeNet with 5 convolutional layers, 2 fully connected hidden layers, and one fully connected output layer. And, AlexNet used ReLU instead of sigmoid as its activation function.
 - The input images for AlexNet are of much higher resolution (224x224 vs. 28x28). The convolutional layers produce a vast amount of features (up to 384 features).
 - After the final convolutional layer, there are two huge fully connected layers with 4096 outputs. These layers require nearly 1GB model parameters, which was a challenge at the time of its development.
 - Finally, AlexNet can classify 1000 categories, a huge improvement from the 10 classes in early digit recognition tasks.
- The Visual Geometry Group (VGG) at Oxford University created the concept of network blocks that can be used again. The VGG block is made up of a series of 3x3-convolutions followed by a max-pooling layer with a stride of 2.
 - The AlexNet architecture is updated with a sequence of VGG-blocks followed by the fully connected layers that lead into the final classification layer.
 - The original VGG network had five blocks. The first two blocks each have one convolutional layer, while the last three blocks each have two convolutional layers. The first block has 64 output channels, and each following block doubles the number of output channels until it reaches 512. This network is often referred to as VGG-11 because it has eight convolutional layers and three fully connected layers.



- **GoogleLeNet** won the ILSVRC 2014 Classification Challenge, which involved 500,000 images labeled with 200 different objects. It used a three-part architecture: the stem (data input), the body (convolutions), and the head (classification). The original design included intermediate loss functions to speed up training of earlier layers, but these are no longer necessary with better learning methods.
 - The stem uses 224x224x3 images as input and applies initial convolutions and pooling to prepare the images for extracting low-level features.
 - The building blocks are called inception modules. They have different convolution types on multiple paths, which are combined at the end. In more details, the inception module provides four paths: 1) a 1x1 convolution that reduces the dimensionality of the incoming feature, 2) a 3x3 convolution preceded by a 1x1 convolution to adjust dimensionality, 3) a 5x5 convolution preceded by a 1x1 convolution to adjust dimensionality, and 4) a 3x3 max pooling layer followed by a 1x1 convolution to adjust dimensionality.

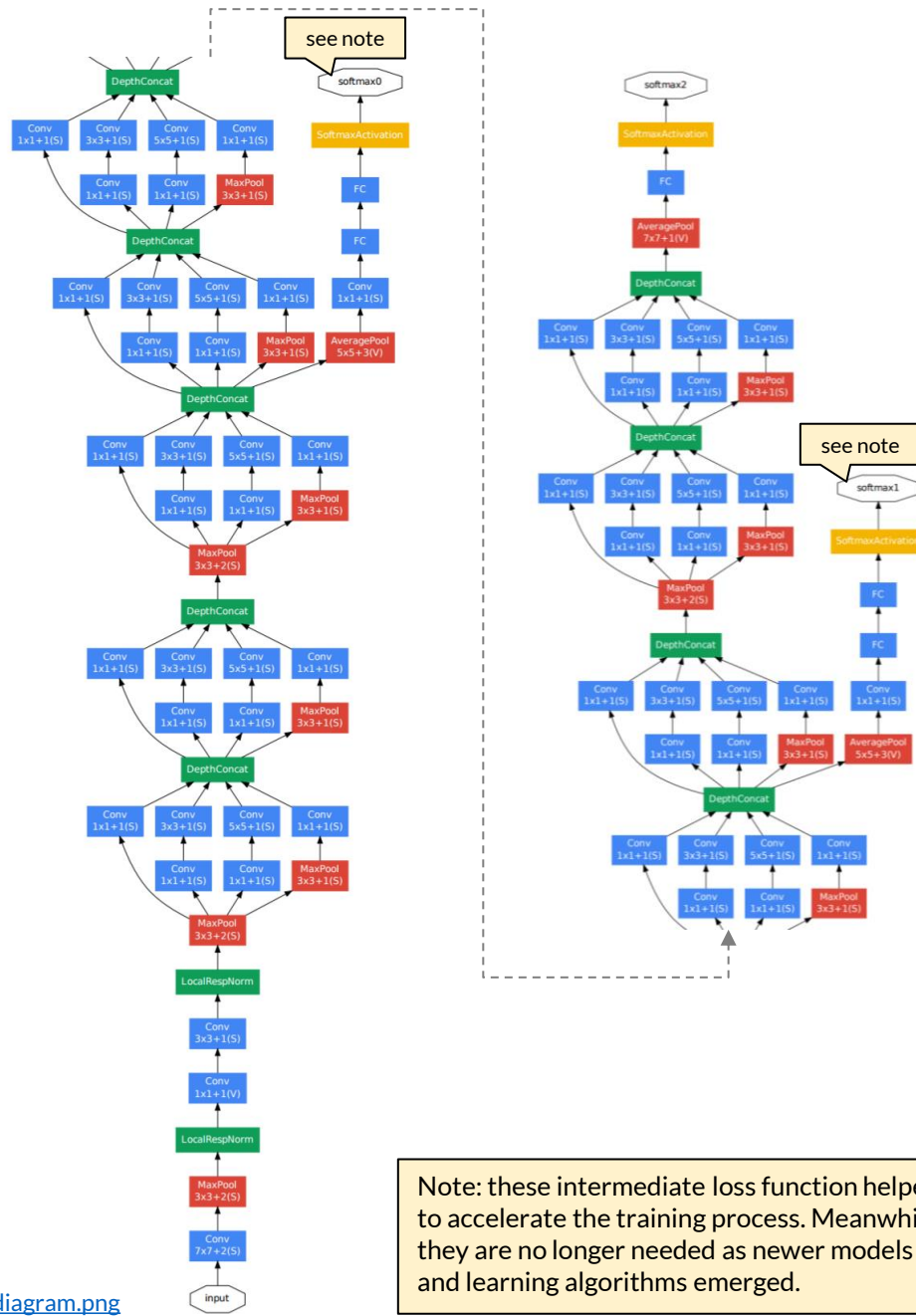


- If we use this module again, the network model can generate paths for features that come from different sequences of these basic convolutions. This reduces the need for human involvement in designing the network and makes it more versatile and able to handle a variety of classification tasks.
- The head of the model is using a global average pooling (7x7 maps) to transform the 2D representation into a 1D feature vector. The last fully connected layer gives the input for the softmax classification output. The model can classify into 1000 different classes.

- The full architecture of GoogleLeNet:

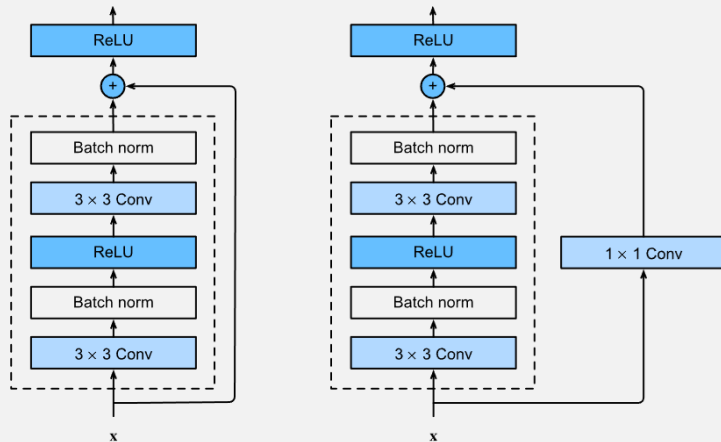
Type	size/stride	output	#params	#ops
convolution	7×7/2	112×112×64	2.7K	34M
max pool	3×3/2	56×56×64		
convolution	3×3/1	56×56×192	112K	360M
max pool	3×3/2	28×28×192		
inception (3a)		28×28×256	159K	128M
inception (3b)		28×28×480	380K	304M
max pool	3×3/2	14×14×480		
inception (4a)		14×14×512	364K	73M
inception (4b)		14×14×512	437K	88M
inception (4c)		14×14×512	463K	100M
inception (4d)		14×14×528	580K	119M
inception (4e)		14×14×832	840K	170M
max pool	3×3/2	7×7×832		
inception (5a)		7×7×832	1072K	54M
inception (5b)		7×7×1024	1388K	71M
avg pool	7×7/1	1×1×1024		
dropout -40%		1×1×1024		
linear		1×1×1000	1000K	1M
softmax		1×1×1000		

source: https://joelouismarino.github.io/images/blog_images/blog_googlenet_keras/googlenet_diagram.png

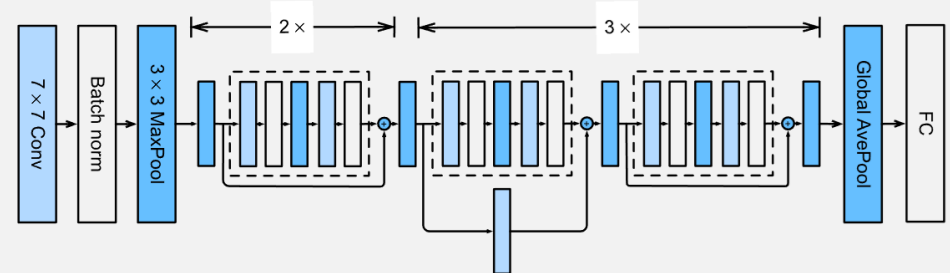


Note: these intermediate loss function helped to accelerate the training process. Meanwhile, they are no longer needed as newer models and learning algorithms emerged.

- **ResNet** introduced a residual block, which is shown on the lower left side.
 - The left version has two paths: 1) a path with convolutions and batch normalization (the so-called residual block), and 2) a direct (identity) path from the input to the output (the so-called residual connection). The two paths are added together (not concatenated) and then given the ReLU activation function. This means the convolution layers need to maintain the same dimensions as the input (no strides, and the same number of features)
 - The main concept is that the block doesn't learn the function $f(x)$, but instead learns the delta function $g(x) = f(x) - x$ that should be added to the input. This has a key advantage because during backpropagation, the gradients flow directly through the second path to previous layers, avoiding issues like vanishing gradients. As a result, the base layers in the model train faster and the learning process becomes more stable.
 - The right version has a similar design. The first path is the same as in the left version, but the second path uses a 1×1 convolution without changing the number of features.
- **DenseNet** builds on this concept by adding a residual connection to each following layer and combining the outputs of the layers instead of just adding them. This creates faster paths for gradient backpropagation.
 - A dense block is made up of several convolution blocks, each with the same number of features. During forward propagation, we combine the input and output of each convolution block on the feature dimension.
 - When you add a lot of dense blocks, the model becomes too complex because it increases the number of features. A transition layer reduces the number of features by using a 1×1 convolution. It also cuts the height and width in half using average pooling with a stride of 2.

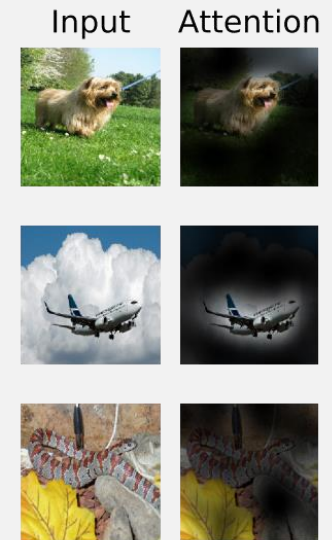
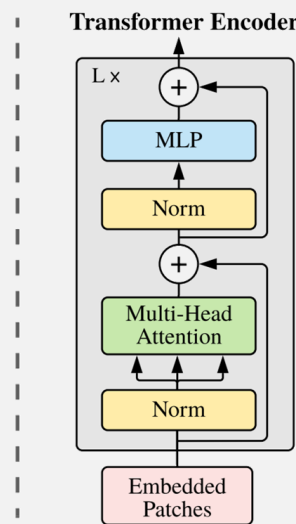
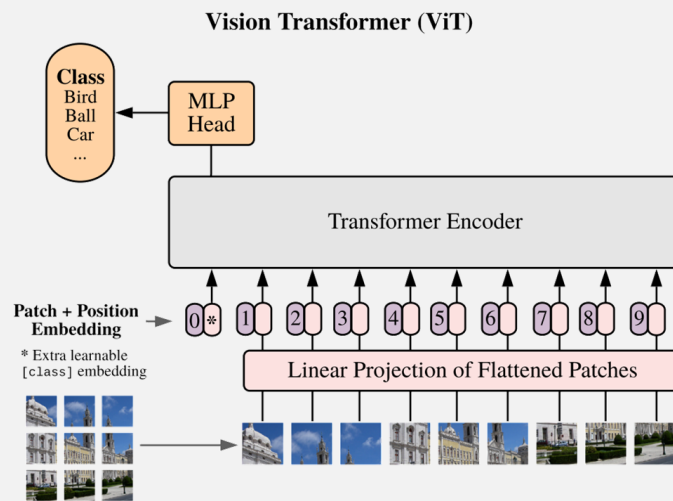


ResNet



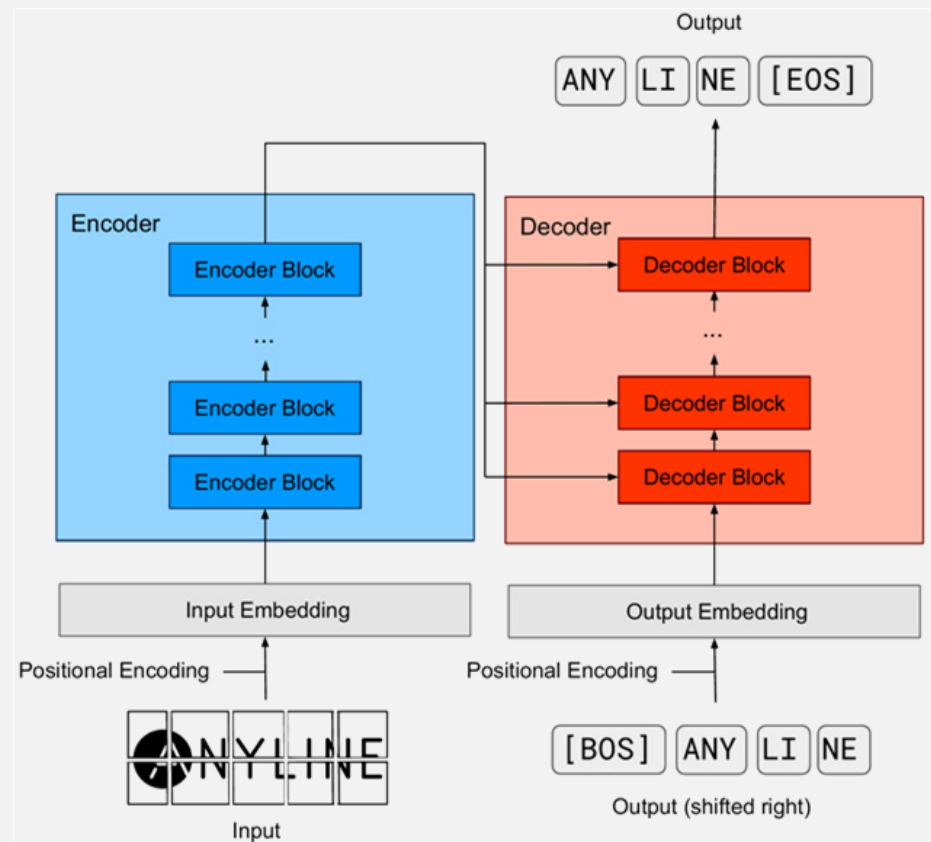
source: https://d2l.ai/chapter_convolutional-modern/resnet.html

- The **Vision Transformer (ViT)** is an image classification model that leverages a transformer like structure. But instead of working on text tokens, we first patch the image using fixed-size patches, learn embeddings for these patches, add position embeddings, and then input the sequence into a standard Transformer encoder. To classify the image, an extra learnable classification token is added to the sequence.
 - The diagram on the bottom left shows the basic layout of vision transformers. It was the first successful attempt to train a transformer encoder on ImageNet, and it produced great results compared to traditional convolutional architectures. The attention mechanism in transformer blocks keeps improving the connections between image patches, just like we do with transformers for natural language processing. The architecture converts an image into a series of vector representations. To classify it, we can add a simple multi-layer block on top that gives a probability distribution over classes using softmax.
 - The figure on the bottom right shows a typical example how the vision transformer attends to image regions that are useful for classification purposes and allows the model to focus on relevant regions of the image.

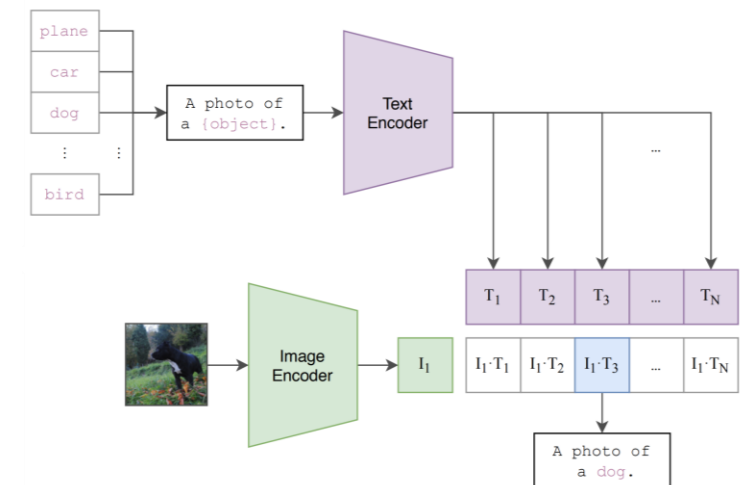
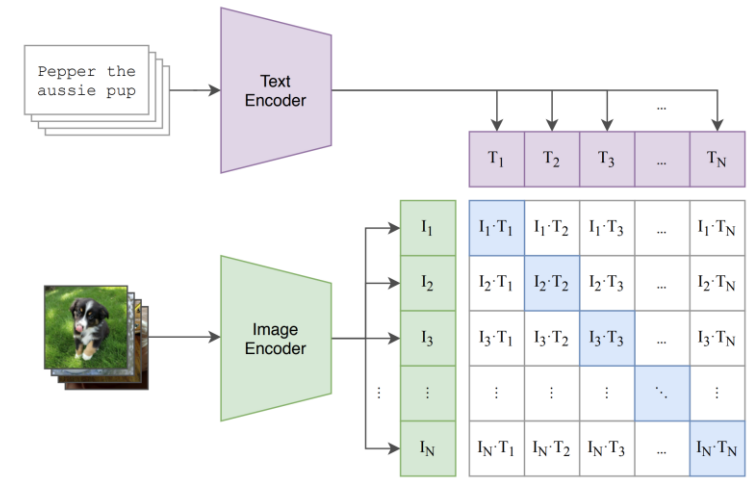


source: Google Research, 2021 (An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale)

- Image-to-text transformers represent a pivotal advancement in computer vision and natural language processing (NLP). Their underlying framework combines the strengths of convolutional neural networks (CNNs) for visual feature extraction and transformer-based models for sequence generation.
- **The visual encoder** typically uses a CNN or a Vision Transformer (ViT) to process raw image data. CNNs extract spatially hierarchical features, while ViTs capture global image information through self-attention mechanisms. The resulting visual features are transformed into embeddings suitable for input into the text generation component.
- A **transformer-based decoder** generates textual descriptions based on the embeddings provided by the encoder. The decoder uses self-attention to manage dependencies in the text sequence and cross-attention to align the image features with the generated words.
- The models are trained using large-scale datasets of image-caption pairs, leveraging loss functions like cross-entropy for text prediction and contrastive losses for ensuring
- This architectural integration of visual encoding and transformer-based sequence generation has set a benchmark in the multimodal AI domain, showcasing the transformative potential of image-to-text transformers in diverse applications.



- **Contrastive Language-Image Pre-training (CLIP)** learns to link images with text through large-scale contrastive training. It uses two encoders: one for text and one for images. The text encoder is a transformer that produces embeddings. The image encoder, which can be a convolutional network or a Vision Transformer, converts an image into a compact vector that captures its visual content.
- During training, CLIP processes many image-caption pairs at once. Each caption goes through the text encoder and each image goes through the image encoder. The model compares every image embedding to every text embedding in the batch. The correct image and its caption form a positive example, and all other image-text combinations are negative. Contrastive learning adjusts the model parameters so matching embeddings move closer in the shared space and mismatched embeddings move farther apart.
- Because CLIP is trained on a wide range of natural language descriptions from the web, it learns a broad vocabulary of visual concepts. After training, it does not need fine tuning to perform classification. Instead of using a fixed set of learned output categories, it makes zero shot predictions by comparing an image embedding to text label embeddings. To classify an image, write short natural language prompts for each category and pass them through the text encoder to get label embeddings. Encode the image and compare its embedding to each label embedding. The label with the highest similarity is chosen as the predicted class.
- This works because CLIP has learned a flexible alignment between text and vision. Instead of memorizing fixed categories, it maps images and text into the same geometric space. Similar meanings sit close together, while different meanings are farther apart. The zero-shot mechanism uses this geometry, allowing the model to handle recognition tasks it was not directly trained on.



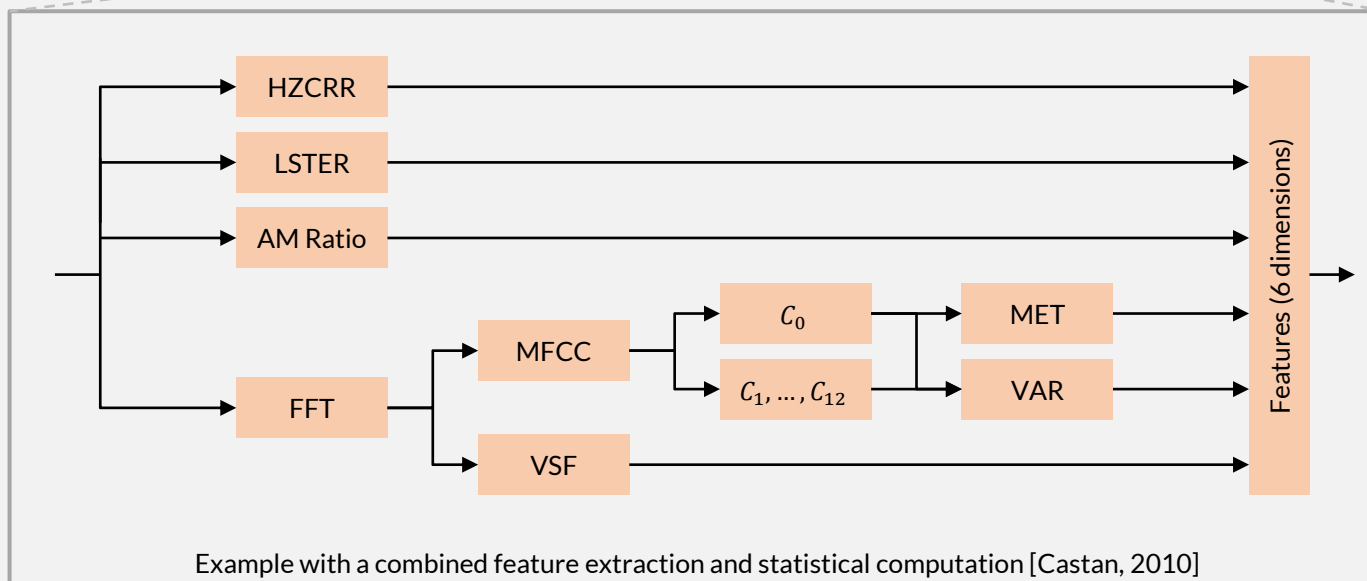
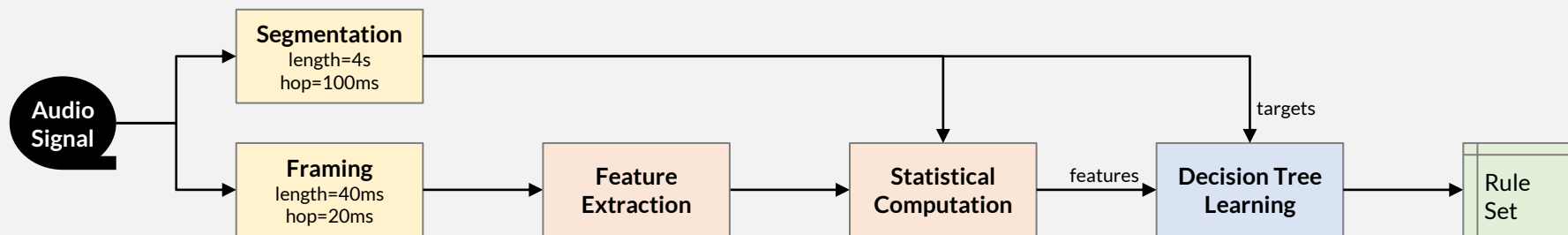
source: <https://arxiv.org/pdf/2103.00020>

14.4 Audio Features

- Audio analysis starts by asking how to turn a continuous stream of sound into information a machine can search, label, or interpret. The first step is to extract features that capture the signal's key qualities. These features reduce the raw waveform to measurements that match what listeners perceive, such as loudness, pitch, and timbre. Once extracted, they form the basis for organizing and understanding the audio.
- A common first task is classifying an audio stream into broad categories such as noise, speech, or music. The system starts with perceptual features such as the spectral centroid, short term energy, and the zero crossing rate. These features are calculated in short time windows so the system can follow how the sound changes from moment to moment. Once the features have been extracted, a simple method such as a decision tree can be used to assign each segment to a class. A decision tree learns a sequence of tests that link feature values to labels. The structure of the tree allows the classification process to proceed in small, interpretable steps, and it can yield accurate predictions without requiring heavy computation.
- **Example: Automated Sport Summary (e.g., football match)**
 - When audio features are used in sports video analysis, the same classification principles apply to event detection. The audio signal holds many clues about what is happening on the field, even when the visuals are complex. By tracking simple perceptual measures over time, a system can spot moments that matter to viewers and editors.
 - Imagine a recording of a football match. During ordinary play the crowd produces a steady, moderate energy level and the sound's frequency makeup stays relatively stable. When an exciting play happens the audio pattern changes sharply and clearly: energy rises quickly, the sound shifts toward higher frequencies as cheering becomes brighter, and the timing becomes more erratic. A decision tree that has learned to link these changes with strong audience reaction can flag the segment as a moment of interest.
 - Commentary gives a second way to spot important events. Commentators change their voice as the action speeds up. Their pitch becomes livelier, their speech rate rises, and their phrasing tightens during key plays. These changes show up in measures like pitch stability, formant movement, and short-term energy. A decision tree can combine these cues with crowd features to improve its estimate.
 - After the system finds these promising segments, a transcription model adds more structure. Speech recognition methods, from the state based Hidden Markov Model to the learned encodings of Whisper, let the system extract the commentator's words. Phrases such as goal, interception, or last minute attempt can confirm the moment's importance. They also let the system label the event type, creating a richer index for later retrieval.

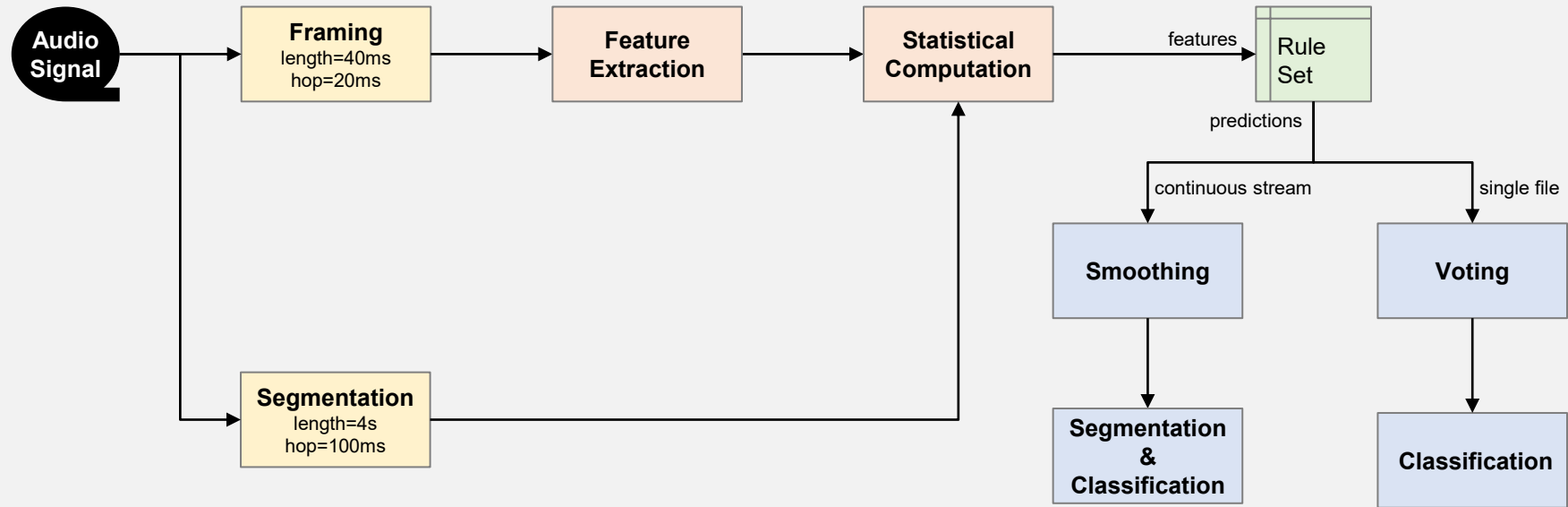
- **Audio classification with decision trees:**

- Decision trees are straightforward and create effective classifiers that work well for many tasks. For instance, they can be used to classify audio signals as either speech or music.
- During the learning phase, we have to prepare the audio signal, pull out features, collect statistical data about the features and how they relate to output categories (music, speech), and choose the top features for classification. In this case, we use XGBoost or C4.5 to pick features and create rules.



- Framing and segmentation involve processing the audio signal in overlapping frames and segments. Each frame and segment has the same length, and the hop distance determines when the next frame/segment begins. Usually, features are extracted for each frame, and statistical measures are applied to the segment across its frame.
- Castan (2010) focused on a small number of characteristic features:
 - **HZCRR:** The Zero-Crossing Rates (ZCR) measures how frequently the signal's amplitude passes the 0-value within a frame. The High Zero-Crossing Rate Ratio (HZCRR) measures the percentage of ZCR values in a segment that are 1.5 times higher than the average ZCR value of frames in the segment.
 - **LSTER:** The Short Time Energy (STE) is just the total of the squared amplitude of the signal within the frame (a measure of energy in the frame). The Low Short Time Energy Ratio measures the percentage of STE values of frames in the segment that are smaller than 50% of the average STE value of frames in the segment.
 - **AMR:** The Amplitude Modulation Ratio (AMR) calculates the low-pass energy of a frame by adding up the squared amplitude after using a low-pass filter with a cut-off at 25Hz. It then compares the highest energy to the lowest energy across all frames in the segment. Speech has a higher ratio than music because of the pauses between vowels and consonants.
 - **VSF:** The Spectral Flux (SF) is the distance between frames in their Fourier transformed signals (spectrum magnitudes). The Variation of Spectral Flux (VSF) measures the variance within the frames in the segment.
 - **MET & VAR:** We calculate 13 Mel-Frequency Cepstrum Coefficients (MFCC) for each frame, labeled C_0, \dots, C_{12} . The Minimum-Energy Tracking (MET) measures how long C_0 is above a certain level. Short pauses in speech will lead to short frame lengths. VAR adds up the variance of all MFCC across the frames in the segment. Low VAR values suggest music.

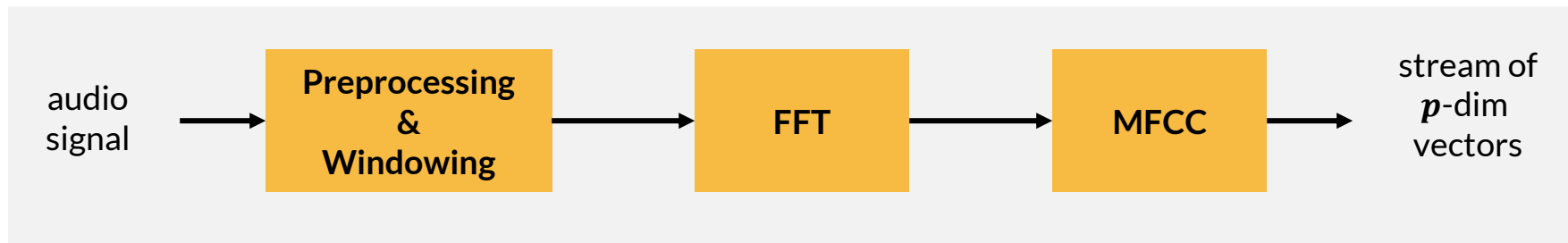
- During the prediction phase, we have to do the same pre-processing, windowing, feature extraction, and statistical calculations as in the learning phase. We also want to smooth out the results for the whole song (using a voting-based approach) or divide a continuous audio signal (like a radio broadcast) to find when the speech changes to music.



- Smoothing involves adding up past predictions with decreasing weights to prevent rapid changes between targets. When there is sufficient evidence for a change, segmentation ends the current segment (different from the segments used for feature extraction) and assigns it the last class label. Then it starts a new segment.
- Voting is easy. The file is classified based on the label that is predicted most often for its parts. Or, the classification can show the likelihood of different labels based on how often they are predicted for the file's parts.

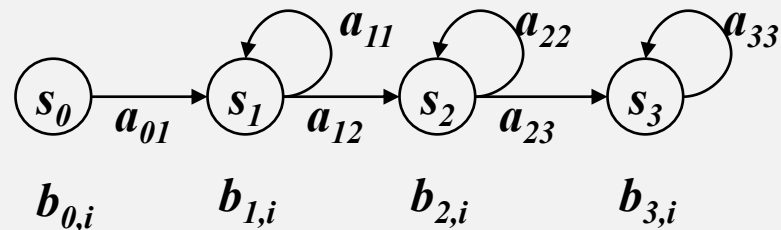
14.4.1 Transcription

- We won't delve too deeply into transcription. Instead, we'll provide a brief overview of the techniques and discuss retrieval aspects.
 - First, the audio signal must be pre-processed to remove any noise. This means that the pitch, tempo, and loudness of the speaker should not affect the result. The typical approach is to use the MFCC method discussed earlier.

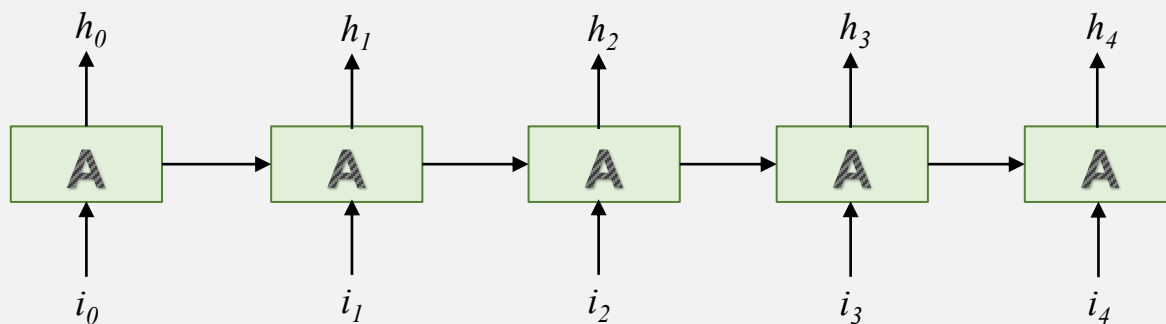


- The MFCC analysis produces a series of p -dimensional vectors. These vectors form the foundation for learning phonemes, which are the building blocks of words and texts. There are two methods for learning phonemes:
 - a) Use a Hidden Markov Model (HMM) to represent the phonemes using quantized vector data and model the temporal transitions. We can use a k -means algorithm to divide the p -dimensional vectors into a set of k states.
 - b) Create a neural network to learn phonemes. This traditionally involved using recurrent networks, which can maintain a current state and pass it on to the next iteration of the network run. Modern approaches use transformer architectures with attention to perform transcription.
- After identifying phonemes, we must then identify the words. Spoken text does not separate words with spaces, but instead comes as a continuous stream of phonemes. Recognizing words depends on the chosen language and also involves handling various dialects, imperfect pronunciations, intonations, and different ways of speaking words.
 - Understanding words requires separate HMM and NNs to learn how to predict words from sequences of phonemes. These methods often can only recognize words that were used during training. In the end, we have a stream of text and can use any text retrieval methods to search through spoken text.
 - To skip word recognition, we can search directly in the phoneme stream. The stream is captured using N-grams, which are overlapping sequences of N phonemes. If the query is not spoken text, it is translated into phonemes using a dictionary and the same N-gram extraction process occurs. Then, we search for the best passages in the spoken text library using the query N-grams.

- The **Hidden Markov Model (HMM)** creates a network of states using quantized MFCC data, with probabilities of moving from one state to another. Each phoneme has its own HMM, and a softmax across the phonetic units decides the recognized phoneme at a specific time. Making HMMs needs more human input or expertise but results in highly effective recognizers.

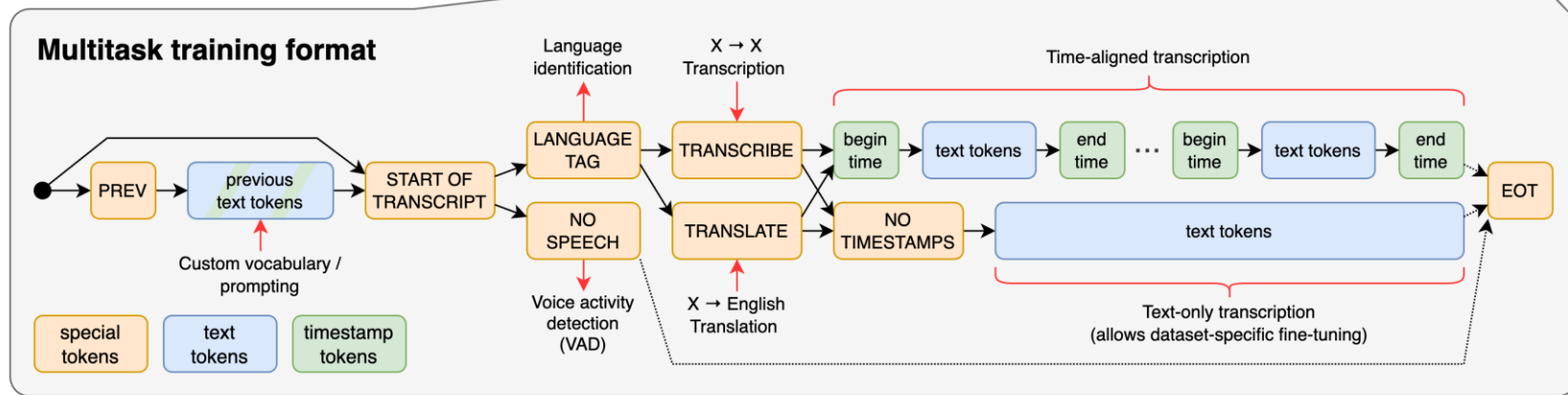
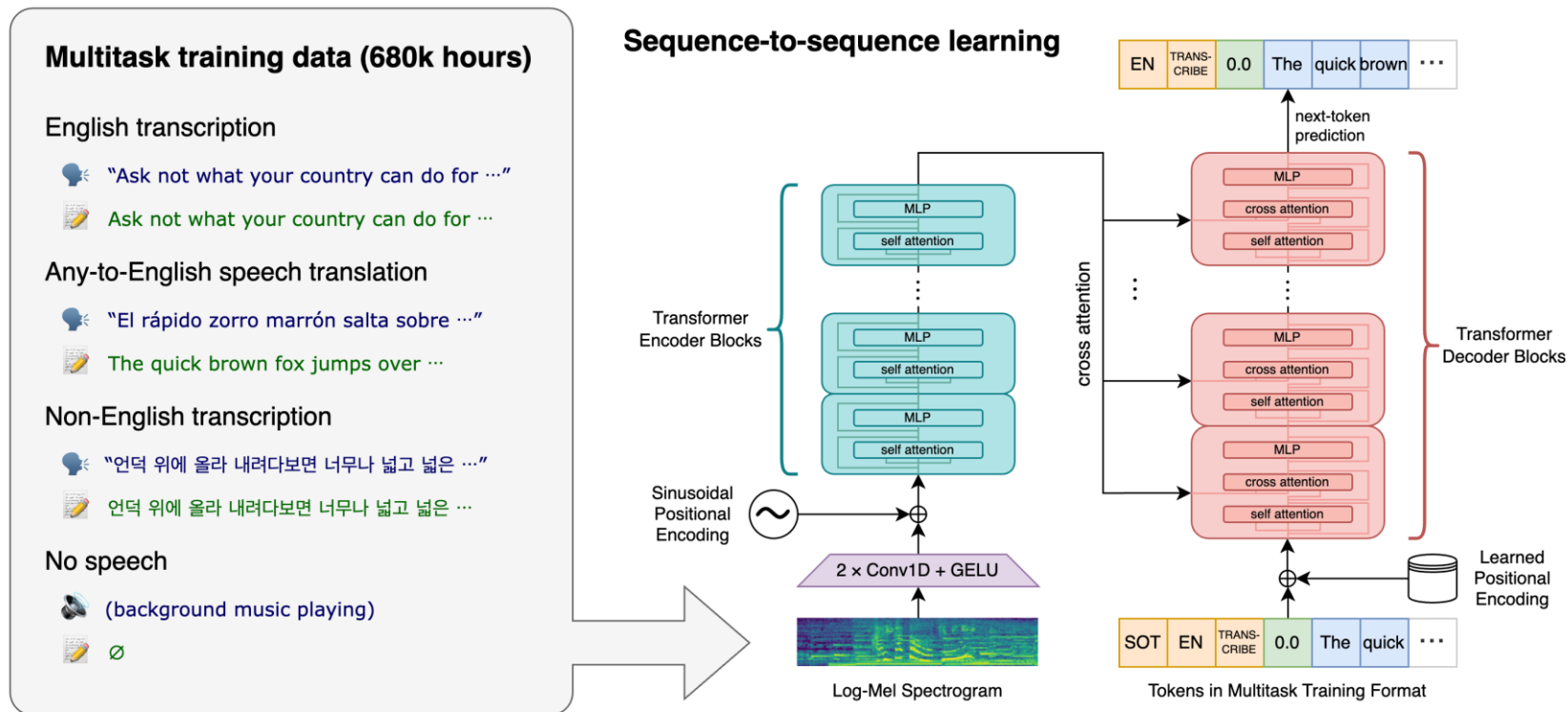


- On the other hand, a **Recurrent Neural Network (RNN)** doesn't need as much specialized knowledge. "Recurrent" means the network keeps track of the current state and feeds it back into the network at each time step. You can think of a recurrent network as a series of connected networks, where the output of one network becomes the input for the next one.



- Variations in speech tempo, how fast or slow someone talks, can compress, stretch, or omit parts of the audio signal, making it harder for models to align sounds with the correct phonetic units. To handle this, several strategies are employed:
 - **Variable-length modeling:** Rather than assuming each phoneme lasts a fixed amount of time, we let states last for different durations. A state can span several time frames, which lets the model handle slower or faster speech.
 - **Use of Duration Models:** Extensions include explicit duration modeling, which represents how long a phoneme or state is likely to last. This helps the system handle changes in tempo and avoid errors caused by unusually short or long durations.
 - **Acoustic Feature Normalization:** Features are sometimes normalized across time to lessen the effect of tempo changes. Methods such as cepstral mean normalization and vocal tract length normalization reduce the variability caused by different speaking rates.
 - **Training with Varied Data:** By training models on speech samples with diverse tempos and pronunciations, the system learns to generalize across tempo-induced variations.
- Today's transcribers use **transformer architectures** to learn a sequence-to-sequence model from audio signals to text tokens. We have previously used transformers for natural language processing, where input and output were represented by token sequences. In image classification, the input sequence was a set of normal-sized image patches. When working with audio signals, we also need to perform some sort of normalization.
 - The speed at which people talk can differ a lot, but we don't have an easy way to make it consistent across a set of recordings. When we train, we mark the start and end of text with time stamps so the model can learn words spoken at different speeds.
 - The rate at which samples are taken can differ in recordings. When retrieving images, we encountered a similar problem with image sizes, which we reduced to fit the model's scale. When processing audio, reducing the sampling rate can cause unwanted auditory artifacts that make it difficult to distinguish between certain sounds (e.g., “s” and “f”). Conversely, higher sampling rates produce a lot of input data, requiring a large context window for the transformer (which is expensive). To lessen the amount of data going into the transformers, we use mel spectrograms as input data.
 - OpenAI's Whisper architecture is a multilingual transcriber with up to 1.5B parameters, as shown on the next page. It was trained on 1 million hours of weakly labeled audio and 4 million hours of pseudolabeled audio collected using previous models. It currently supports about 100 languages with varying accuracy.

- OpenAI/Whisper: <https://github.com/openai/whisper>



14.5 Literature and Links

Implementations

- Models: Huggingface, the AI community building the future. <https://huggingface.co>
- Data & Competitions: Kaggle: Your Machine Learning and Data Science Community. <https://www.kaggle.com>
- Transformers: State-of-the-art Machine Learning. <https://github.com/huggingface/transformers>, <https://huggingface.co/docs/transformers/index>
- LangChain: Framework for developing applications powered by language models: <https://python.langchain.com/>
- NLTK: Natural Language Toolkit. <https://www.nltk.org/>
- spaCy: Industrial-Strength Natural Language Processing. <https://spacy.io/>
- Apache OpenNLP: Machine learning based toolkit. <https://opennlp.apache.org/>
- WordNet: A Lexical Database for English. <https://wordnet.princeton.edu/>
- Snowball: Stemming Algorithms. <https://snowballstem.org/>
- GloVe: Global Vectors for Word Representation. <https://github.com/stanfordnlp/GloVe>
- Learning Transferable Visual Models From Natural Language Supervision, <https://arxiv.org/pdf/2103.00020>
- Training CLIP Model from Scratch for an Fashion Image Retrieval App, <https://learnopencv.com/clip-model/>