**Machine Learning Topics**

**Data Processing**

1. **Data Preparation**
2. **Data Imbalance Consequences**
3. **SMOTE**
4. **Feature Selection**
5. **Sampling**
6. **Normalization**
7. **Text Preprocessing**

1. Lowercasing

* Converts all text to lowercase.
* Helps standardize words (e.g., "Apple" and "apple" are treated the same).

2. Tokenization

* Splits text into smaller units (called tokens), usually words.
* Types of tokenization:
  + Word tokenization: Splits text into individual words.
  + Sentence tokenization: Splits text into sentences.
  + Subword tokenization: Breaks down into smaller units than words (useful in deep learning).

3. Removing Punctuation, Numbers, and Special Characters

* Eliminates characters that may not be useful for analysis.
* Keeps only alphabetical text to reduce noise.

4. Stopword Removal

* Removes common words that appear frequently but add little meaning (e.g., "the", "is", "and").
* Reduces dimensionality and focuses on meaningful words.

5. Stemming and Lemmatization

* Both reduce words to their base or root form.
* Stemming: Cuts words to their base form (may not be a real word).
  + Example: "running" → "run", "flies" → "fli"
* Lemmatization: Converts words to their meaningful base form using vocabulary and grammar.
  + Example: "running" → "run", "better" → "good"
* Lemmatization is usually more accurate than stemming.

6. Text Normalization

* Standardizes text in various ways, such as:
  + Expanding contractions (e.g., "can't" → "cannot")
  + Handling misspellings
  + Converting accented characters
* Ensures consistency across the dataset.

7. Removing Duplicates and Blank Lines

* Eliminates repeated or empty entries.
* Helps in maintaining a clean dataset.

8. Spelling Correction (Optional)

* Fixes typos and spelling errors to improve word consistency.
* Especially useful for user-generated content (like social media posts).

9. Slang and Abbreviation Handling (Optional)

* Translates informal words to standard language.
  + Example: "u" → "you", "btw" → "by the way"

10. N-gram Generation

* Captures combinations of words (e.g., bigrams like "not good").
* Helps preserve the context and order of words.
* Useful when word combinations matter more than individual words.

11. Final Step: Vectorization

* Converts processed text into numerical format.
* Common methods:
  + Bag of Words (BoW): Counts word frequency in each document.
  + TF-IDF (Term Frequency-Inverse Document Frequency): Weighs words by importance.
  + Word Embeddings: Represents words in dense vector form based on meaning (e.g., Word2Vec, GloVe, BERT).

1. **Padding and Truncation of Text**

Padding and truncation are important steps in text preprocessing when you're working with machine learning models that require fixed-length input, especially neural networks like RNNs, LSTMs, or transformers (like BERT).  
  
**1. Padding**

* **What it does**: Adds extra tokens (usually zeros or a special padding token) to the end or beginning of a sequence to make it a fixed length.
* **Why it's needed**: Not all sentences are the same length, but models need inputs of the same size.
* **Where it's used**: Often used after tokenization and before feeding data into a model.

**2. Truncation**

* **What it does**: Cuts longer sequences down to the maximum allowed length.
* **Why it's needed**: To avoid memory issues and keep computation efficient.
* **How it's done**: Usually from the end (but can be from the start depending on the use case).

**Common Settings (when using libraries like TensorFlow or Hugging Face):**

* + padding='max\_length': Pads all sequences to the same length.
  + truncation=True: Automatically truncates sequences longer than the max length.
  + max\_length=128: Defines the maximum sequence length.

1. **Word Embedding**
   1. **Goal:**The goal behind embeddings is to convert words into vectors and ensure two words/vectors of similar meaning/context are mapped close in the embedding space.

i.e Calculating the dot product (i.e angle between the two should be small) should be large.

* 1. **Word Embedding**An embedding is a way to represent words as dense vectors in a lower-dimensional space, where similar words end up close together.

Think of each dimension in the embedding as capturing some latent feature — like "animal-ness", "femininity", "past tense-ness", etc.

These features are not predefined by us — the model learns them during training.

It is like asking the model to classify words along N different (hidden) axes, where N is the embedding size.

* 1. **One Hot Encoding**
  2. **Distributed Representations**
  3. **Static Embeddings**

**[DEPRECATED]**

Static embeddings are the oldest type of word embedding. The embeddings are generated against a

large corpus but the number of words, though large, is finite.  
If you have a word whose embedding needs to be looked up that was not in the original corpus, then you are out of luck.

In addition, a word has the same embedding regardless of how it is used, so static embeddings cannot

address the problem of polysemy, that is, words with multiple meanings.

* 1. **Dynamic Embedding**

Dynamic embeddings are word representations that change depending on the context they appear in.

This is in contrast to static embeddings, where each word has a single fixed vector, no matter where or how it appears.  
  
**Model How It Works**

ELMo Uses a deep LSTM to generate embeddings based on the entire sentence

BERT Uses transformers and attention to create context-aware embeddings for each word

GPT Also produces dynamic embeddings (during generation or fine-tuning)

* 1. **Word2Vec**Word2Vec turns words into dense vectors of numbers, so that similar words have similar vectors. These vectors capture semantic meaning — for example:

vector("king") - vector("man") + vector("woman") ≈ vector("queen")

**Captures meaning and relationships** between words.

There are two main architectures:

1. CBOW (Continuous Bag of Words):

Predicts a word given its context (surrounding words).

Example: "The cat sat on the \_\_\_" → model predicts "mat".  
in **Word2Vec (CBOW)**, the **context window is the input** to the model.

1. Skip-Gram:

Predicts context words from a given word.

Example: Input "cat" → model predicts words like "the", "sat", "on".

In Skipgram we just pass one word  
  
  
**Characteristics of Word2Vec Embeddings:**

1. **Dense**:
   * Unlike one-hot vectors (which are mostly 0s), Word2Vec embeddings are packed with real numbers.
   * Example: A one-hot vector for "cat" might be 10,000 dimensions long with only one "1". In contrast, a Word2Vec embedding might be 100-300 dimensions, all with meaningful values like [0.25, -1.3, 0.7, ..., 0.01].
2. **Low-dimensional**:
   * Usually 50–300 dimensions, which is much smaller and more computationally efficient than sparse vectors.
3. **Semantic meaning is encoded**:
   * Words that appear in similar contexts (e.g., “king” and “queen”) will have similar vectors.
   * The relationships between vectors capture analogies (like we mentioned: king - man + woman ≈ queen).
4. **Fixed size**:
   * Each word, regardless of how common or rare, gets a vector of the same length.
5. **Context-independent**:
   * Each word has **one vector**, no matter the sentence. So "bank" (as in river bank vs money bank) has the same embedding in both contexts.
   * This is a limitation that newer models like BERT try to solve.
   1. **Glove**

GloVe differs from Word2Vec in that Word2Vec is a predictive model while GloVe is a count-based

model.

The first step is to construct a large matrix of (word, context) pairs that co-occur in the training

corpus. Rows correspond to words and columns correspond to contexts, usually a sequence of one

or more words. Each element of the matrix represents how often the word co-occurs in the context.

The GloVe process factorizes this co-occurrence matrix into a pair of (word, feature) and (feature,

context) matrices. The process is known as matrix factorization and is done using Stochastic Gradient

Descent (SGD), an iterative numerical method.

R = P \* Q ≈ R’

Thus, the model decomposes a larger matrix R into it’s constituents approximately. The difference between the matrices R and R’ represents the loss and is usually computed as the mean-squared error between the two matrices.  
The GloVe process is much more resource-intensive than Word2Vec. This is because Word2Vec learns

the embedding by training over batches of word vectors, while GloVe factorizes the entire co-occurrence

matrix in one shot.

* 1. **Fastext**
  2. **Concept Number batch**

1. **Character Embedding**

Another evolution of the basic word embedding strategy has been to look at character and subword embeddings instead of word embeddings.

First, a character vocabulary is finite and small – for example, a vocabulary for English would contain around 70 characters (26 characters, 10 numbers, and the rest special characters), leading to character models that are also small and compact. Second, unlike word embeddings, which provide vectors for a large but finite set of words, there is no concept of out-of-vocabulary for character embeddings, since any word can be represented by the vocabulary. Third, character embeddings tend to be better for rare and misspelled words because there is much less imbalance for character inputs than for word inputs.  
However, unlike word embeddings, character embeddings tend to be task specific and are usually generated inline within a network to support the task. For this reason, third party character embeddings are generally not available.

1. **Sentence Embedding**
2. **Language Based Model**

A language model-based embedding is a type of contextualized vector representation for words, phrases, or entire sequences, generated by a neural language model that has been trained to understand natural language. Unlike static embeddings, which assign a single fixed vector to each word in the vocabulary, language model-based embeddings produce dynamic vectors that capture the semantic meaning of a word within its specific context.

Language model-based embeddings are derived from deep neural architectures, often based on **transformers** (in models like BERT and GPT) or **LSTMs** (in earlier models like ELMo). These models are trained on massive corpora using **language modeling objectives**, such as:

* **Causal Language Modeling** (e.g., GPT): Predict the next token in a sequence.
* **Masked Language Modeling** (e.g., BERT): Predict randomly masked words in a sentence.
* **Sequence-to-Sequence Modeling** (e.g., T5, BART): Predict output sequences from input sequences.

**Characteristics  
Contextual**: The same word can have different embeddings depending on its usage.  
**Layered**: Each layer of the model captures different linguistic features (e.g., syntax in early layers, semantics in deeper layers).  
**Dynamic**: Embeddings are generated on the fly for each new input, rather than being precomputed and fixed.  
**Transferable**: Embeddings can be fine-tuned or used across various downstream tasks (e.g., classification, question answering, summarization).  
  
**Pretraining** is the initial phase of training a language model on a large, unlabelled corpus of text data using a general-purpose objective. The goal is to enable the model to learn fundamental properties of language, such as syntax, semantics, and world knowledge, which can then be fine-tuned for specific downstream tasks (like sentiment analysis, question answering, or translation).  
**pretraining is expensive**, and **yes — it is self-supervised**.  
Once fine-tuned, you can reuse this model for multiple tasks within your domain. The fine-tuning step is generally much less expensive compared to the pretraining step.

1. **Word Embedding in Non-Textual Context**

Word embeddings are typically used to represent words in a dense vector space. But the idea of embeddings—turning symbolic or categorical data into dense vectors that capture similarity—has been applied far beyond just "words."

* 1. **Item2Vec**  
     Item2Vec is used to generate dense vector representations of items (e.g., songs, movies, books) based on user interactions — typically used in recommendation systems.

Inspired by: Word2Vec Skip-Gram

How it works:

Items that appear together in user sessions (like shopping carts, playlists, or browsing history) are treated like words in a sentence.

The model learns to predict neighboring items given a current item — much like Word2Vec predicts surrounding words.

The result is that items that often co-occur in user behavior will have similar embeddings

* 1. **Prod2Vec**

Prod2Vec (Product2Vec) extends Item2Vec with **richer context** — often incorporating product metadata like categories, brands, or prices — for even more meaningful embeddings.

* 1. **Wav2Vec** is a deep learning model for learning representations of raw audio waveforms, used primarily in automatic speech recognition (0041SR).

Developed by: Facebook AI (now Meta AI)

How it works:

Wav2Vec processes raw audio (e.g., .wav files) and learns a contextualized embedding of speech frames.

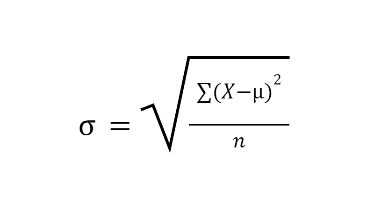
It uses self-supervised learning:

The model learns to predict masked segments of the audio signal using surrounding context (like BERT for audio).

Wav2Vec 2.0 adds a transformer-based architecture and shows state-of-the-art performance on many ASR tasks.

**Statistics**

1. **Standard Deviation**

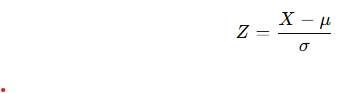
Standard deviation is the **square root** of variance. It tells us how much, on average, data points differ from the mean in the original units.

1. **Variance**Variance is the average of the squared differences from the mean. **= σ2**
2. **Range**

The range is a measure of dispersion that shows the spread of a dataset.  
Range=Maximum Value−Minimum Value

1. **Z Score**

A Z-score tells us how many standard deviations a data point is from the mean. It helps standardize different datasets and identify outliers.



1. **Middle of Dataset**
   1. **Mean**

The sum of all values divided by the total number of values.

* 1. **Median**The middle value when data is arranged in ascending order.

If the number of values is odd, the middle value is the median.

If even, the median is the average of the two middle values.

* 1. **Mode**

The value that appears most frequently in the dataset.

A dataset can have:

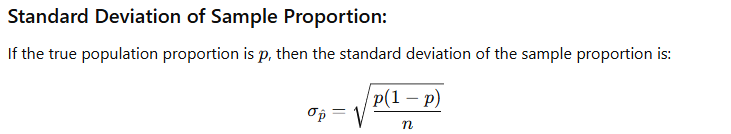
* + - No mode (if all values appear equally).
    - One mode (unimodal).
    - Multiple modes (bimodal, multimodal).

1. **Standard Error**

**Standard Error (SE)** measures the variability of a sample statistic (such as the sample mean) from sample to sample. It tells us how much the sample mean is expected to fluctuate if we were to take multiple samples from the same population.

 where s is the standard deviation

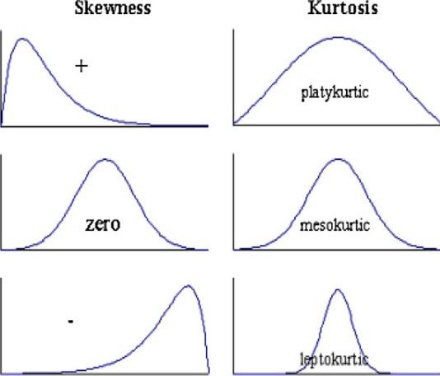
Larger samples give more precise estimates of the population mean as SE decreases as n increases.

[Requires Investigation]

1. **Population Proportion**

The **population proportion** (p) is the fraction of an entire population that has a particular characteristic. It is a fixed but often unknown value.

1. **Probability Distribution**
   1. **Kurtosis**

Kurtosis describes whether data has heavy tails (outliers) or light tails compared to a normal distribution.

* **Mesokurtic (K=3K = 3K=3)**: Normal distribution (moderate tails).
* **Leptokurtic (K>3K > 3K>3)**: Heavy tails (more outliers, sharp peak).
* **Platykurtic (K<3K < 3K<3)**: Light tails (fewer outliers, flatter peak)
  1. **Skew**

Skewness describes how asymmetrical a distribution is compared to a normal distribution.

It tells us whether data is skewed (shifted) to the left or right.

* **Zero skewness (Sk=0)**: Data is **symmetrical** (e.g., normal distribution).
* **Positive skewness (Sk>0)**: Right-skewed (longer tail on the right).
* **Negative skewness (Sk<0)**: Left-skewed (longer tail on the left).

1. **Sampling**
2. **Central Limit Theorem**

The sampling distribution of the mean will always be normally distributed, if the sample size is large enough.

1. **Empirical Rule**

The Empirical Rule states that for a normal distribution, nearly all data falls within three standard deviations of the mean:

1. 68% of the data falls within 1 standard deviation (μ±σ).
2. 95% of the data falls within 2 standard deviations (μ±2σ).
3. 99.7% of the data falls within 3 standard deviations (μ±3σ)
4. **True Mean vs Sample Mean**

The **true mean** (also called the **population mean**, denoted as μ) is the average of all values in an entire population. It is a fixed but often unknown value.

The **sample mean** (denoted as 𝑥̄) is the average of a subset (sample) of the population. It is an estimate of the true mean and varies depending on the sample chosen.

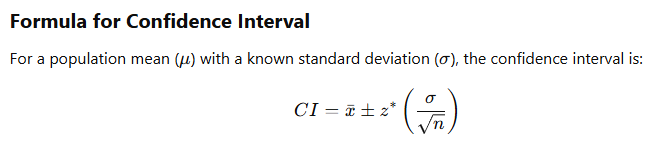
More number of samples help us guide towards true mean.

1. **Confidence Interval**

A confidence interval (CI) in statistics is a range of values that likely contains the true population parameter (e.g., mean or proportion) with a certain level of confidence. It provides an estimate along with a measure of uncertainty.

**Key Components of Confidence Intervals**

1. **Point Estimate** – A sample statistic used as the best guess of the population parameter (e.g., sample mean xˉ\bar{x}xˉ).
2. **Margin of Error (ME)** – A value that reflects the variability in the estimate, typically based on the standard deviation and sample size.
3. **Confidence Level** – The probability that the interval contains the true population parameter (commonly 90%, 95%, or 99%).



[Requires Investigation] Where z is the critical value from the standard normal distribution

**Machine Learning Theory**

1. **Hypothesis Space**
2. **Generalization**
3. **Representational Capacity**
4. **VC Dimension**
5. **VC Dimension vs Representational Capacity**
6. **Bayes Error**
7. **No Free Lunch Theorem**
8. **Bias**
9. **Variance**
10. **Confidence Interval**
11. **Point Estimation**
12. **Frequentist**
13. **Bernoulli Distribution**
14. **Estimator**
15. **Estimator Bias**
16. **Estimator Variance**
17. **Standard Error**
18. **Expected Value Variance**
19. **Sparsity**sparsity means that most values in a dataset, vector, or matrix are zero (or close to zero)

**Machine Learning Implementation**

1. **Weight Decay**
2. **Train Test Validation Split**
3. **Validation vs Test**
4. **Hyperparameter**
5. **Parameter**
6. **Hyperparameter vs parameter**
7. **Cross Validation**
8. **Bias Variance Trade-Off**The Bias-Variance Trade-Off is a fundamental concept in machine learning that explains the balance between a model's ability to make accurate predictions on training data and its ability to generalize to unseen data.
   1. **Bias:** error due to overly simplistic assumptions in the learning algorithm
   2. **Variance:** model's sensitivity to fluctuations in the training data

The goal is to find a balance between bias and variance

1. **Logistic Regression**
2. **Linear Regression**
3. **Linear Regression without gradient descent**
4. **Regularization**
   1. **Definition/Need**

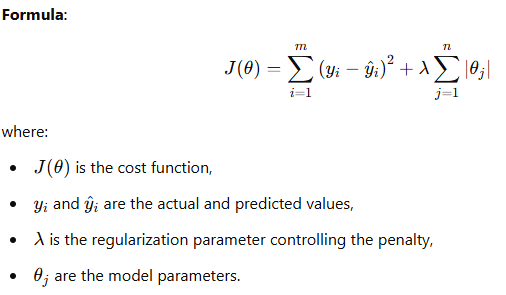
Regularization is a technique used to prevent overfitting by adding constraints to a model. It is needed to prevent overfitting, improve generalization, and ensure stable training of models.

Choosing model: (We Choose simplest model for a loss based on Occam’s Razor)

min : {loss(Training Data|Model)}+λ∗𝑐omplexity(Model) λ >= 0

* 1. **Types**
     1. **L1 Lasso**

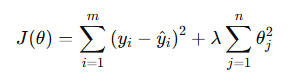
L1 regularization adds the absolute values of the coefficients as a penalty to the loss function.

When optimizing the loss function, L1 creates a **diamond-shaped constraint region**, which often **intersects the optimal solution at zero for some weights**. [Requires Investigation]

* + - 1. **Sparsity**

Leads to Sparsity

⚠Since it causes sparsity that means certain features get selected.

* + 1. **L2 Ridge**

L2 regularization adds the squared values of the coefficients as a penalty. It reduces overfitting by shrinking the weights but does not lead to sparsity.

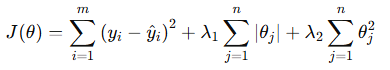
✅It prevents overfitting while keeping all features.

✅This makes it more stable (less sensitive) to small changes and works better when features are correlated.

* + 1. **Elastic Net**

Elastic Net combines L1 and L2 regularization, benefiting from both sparsity and weight shrinkage.

It is useful when dealing with highly correlated features.



* + 1. **Dropout Regularization [Mentioned in Deep Learning]**
    2. **Early Stopping**

Early stopping is a form of implicit regularization where training stops when the validation error starts increasing, preventing overfitting.

Implemented in TensorFlow using patience

early\_stopping = EarlyStopping( monitor='val\_loss', patience=10, restore\_best\_weights=True)  
# Restore best weights  
# Monitor validation loss  
# Stop if no improvement for 10 epochs

Pros:

✅ **Prevents Overfitting** – Stops training before the model memorizes noise.  
✅ **Reduces Training Time** – Saves computational resources by stopping unnecessary training.  
✅ **Automatically Selects Optimal Epoch** – No need to manually tune the number of epochs.

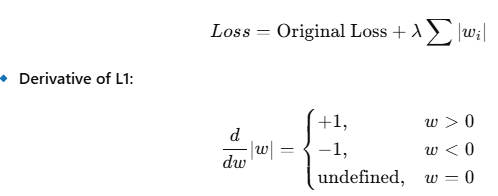
Cons:

**Limitations of Early Stopping**

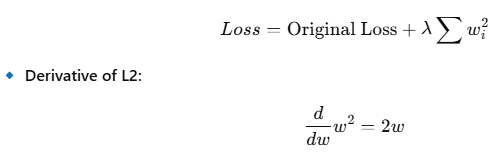
⚠ **May Stop Too Early** – If patience is too low, training may stop before reaching the best performance.  
⚠ **Requires a Validation Set** – Needs a separate validation set, reducing training data.  
⚠ **Not Always Optimal for Some Models** – In some cases, alternative regularization methods like dropout or weight decay may work better.

* 1. **Questions:**
     1. **Why does L1 lead to sparseness but not L2?**

**Answer:** The key difference comes from how L1 and L2 regularization affect the gradient during optimization.

****

* This means L1 applies a constant force (either +λ+ or −λ) that pushes small weights directly to zero.
* When a weight is small enough, the optimizer finds it cheaper to set it to zero than to keep updating it.
* This creates sparsity because many weights become exactly zero**.**

**L2 penalty adds the squared value of weights **

* The L2 penalty is proportional to the weight itself, meaning large weights shrink a lot, but small weights shrink very little.
* Instead of making weights zero, L2 just makes all weights smaller and smoother, but never eliminates them completely.

1. **Support Vector Machines**
2. **Kernel Trick**
3. **RBF Kernel**
4. **Radio Kernel**
5. **Hard Margin**
6. **Soft Margin**
7. **Decision Tree**
8. **Gini Index/Impurity Index**
9. **Random Forrest**
10. **Bagging**
11. **Boosting**
12. **Evaluation**
13. **Generalization**
14. **Text Model**
15. **Metrics**
    1. **Accuracy**
    2. **Precision**
    3. **Recall**
    4. **F1 Score**
16. **Loss Function**
17. **Cost Function**
18. **Loss Function vs Cost Function**
19. **Mean Squared Error**
20. **Law of Parsimony/Occum’s Razor**

Occam's razor is a principle in machine learning that states that simpler models are more likely to be correct than more complex ones. It's also known as the law of parsimony.

Important Questions

1. Why do linear models generalize better?

**Deep Learning**

1. **Perceptron**

A **Perceptron** is the simplest type of artificial neural network.

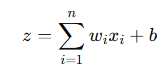
It is a type of **linear classifier** that makes predictions based on a weighted sum of input features followed by an activation function.

Composed of:

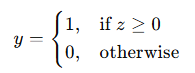
A perceptron consists of the following components:  
1. **Inputs** (x1,x2,...,xnx\_1, x\_2, ..., x\_nx1​,x2​,...,xn​): Features of the data.

2. **Weights** (w1,w2,...,wnw\_1, w\_2, ..., w\_nw1​,w2​,...,wn​): Adjustable parameters that determine the importance of each feature.

3. **Bias (b)**: A constant term that allows shifting the decision boundary.  
4. **Summation Function**: Computes the weighted sum of inputs



5. **Activation Function**: Applies a step function (threshold function) to determine the output:



1. **Feed Forward Neural Networks**
2. **Recurrent Neural Networks**
3. **Hidden Layers**

Layers between the input and output layers

1. **Width of Model**

The number of neurons in a hidden layer. A wider model (more neurons per layer) can capture more features but may lead to overfitting. Defined for each layer not whole.

1. **Universal Approximation Theorem**

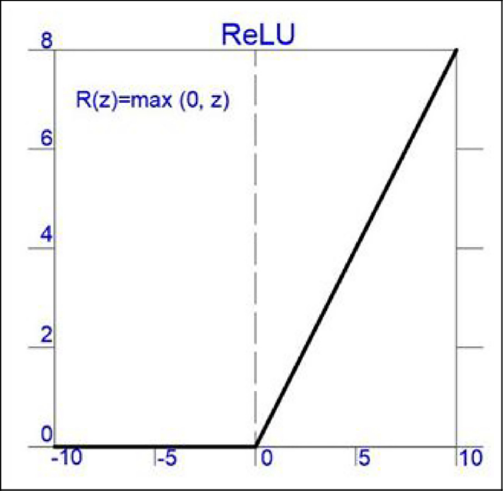
The Universal Approximation Theorem is a pivotal result in neural network theory, proving that feedforward neural networks can approximate any continuous function under certain conditions.

1. **Activation Function**
   1. **Need**

Activation functions are crucial in neural networks because they introduce **non-linearity** into the model, enabling it to learn complex patterns and relationships in data. Without activation functions, a neural network would essentially be a linear model, limiting its ability to handle complex tasks.

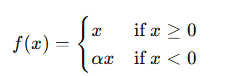
If we had no activation function, the output of a layer would be simply a weighted sum of the inputs z=w1x1+w2x2+...+wnxn+bz = w\_1x\_1 + w\_2x\_2 + ... + w\_nx\_n + bz=w1​x1​+w2​x2​+...+wn​xn​+b

* 1. Common ones:
     1. ReLU

**Rectified Linear Unit – *f*(*x*)=max(0,*x*)**



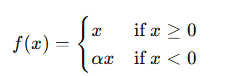
* + - 1. Pros:
         1. Only negatively saturates
         2. Better Sparsity so less computation
      2. Cons:
         1. Dying RELU (Can get stuck at 0)
         2. Not differentiable at 0 (solved using f′(0)=0)
    1. Leaky ReLU



Where α: a small positive constant (usually something like 0.01).

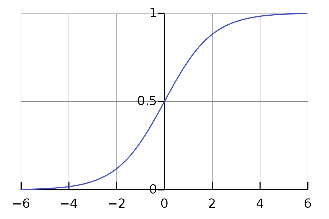
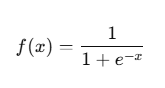
designed to fix a problem known as the **"dying ReLU"** problem  
Leaky ReLU doesn’t just cut off all negative values — instead, it lets a small negative slope through.  
So even when x<0, the function still outputs a small (negative) value and, more importantly, **has a non-zero gradient**.

* The neuron still gets to **learn** (because there's still a gradient to flow back during backpropagation).
* It reduces the risk of neurons getting “stuck” outputting 0 forever.  
  + 1. Parametric ReLU



Here alpha is not fixed and learned during training. Can be shared or different alpha per layer.

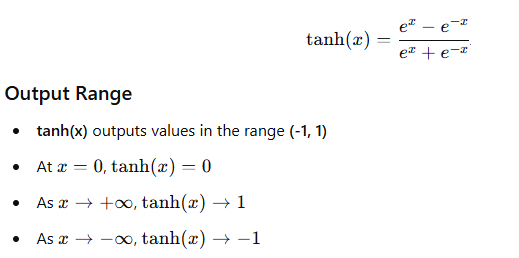
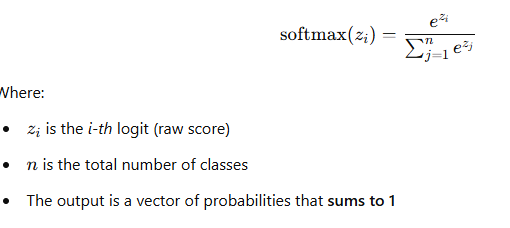
Gives the network freedom to learn better slopes but this may cause overfitting if you're not careful.  
Slightly slower than regular Relu due to the extra computation.

* + 1. Sigmoid (Logit)  
       

When x→−∞ f(x)→0  
When x→+∞ f(x)→1

At x=0, f(x)=0.5  
  
 Not used much anymore due to vanishing gradients (since derivative is close to 0).

Also computationally expensive.

* + 1. Tanh  
         
       like sigmoid, **tanh suffers from the vanishing gradient problem** for very large or very small inputs and unpopular compared to RELU.
    2. Softmax  
         
       The **softmax function** takes a vector of raw scores (called *logits*) and turns them into **probabilities**.
    3. Swish
    4. GELU
  1. Questions:
     1. **Why Non-Linearity Important?**
     2. **Why is Relu still more prevalent despite leaky relu problem?**
     3. **Sigmoid vs Softmax**

|  |  |  |
| --- | --- | --- |
| Feature | Sigmoid | Softmax |
| Use Case | Binary Classification | Multi Class Classification |
| Independence | Each output is independent | Outputs are interdependent (probability distribution) |
| Range | (0, 1) for each class | (0, 1) for each class but all sum up to 1 |

* Sigmoid treats each class independently, meaning probabilities don’t sum to 1.
* It can assign high probabilities to multiple classes at the same time, which is problematic when only one class should be selected.
* Softmax ensures a mutually exclusive decision by normalizing across all classes.
* Sigmoid is better than softmax in **two main cases**: Binary Classification & Multi-Label Classification of Independent classes
* **Softmax** is computationally more expensive than **sigmoid**, especially as the number of classes increases.

1. Vanishing Gradient  
   As you go backward through a deep network (from output toward the input layer), gradients are calculated via **the chain rule**. That means:

If each of those derivatives is a number less than 1 (like 0.5), and you multiply a bunch of them together… the product shrinks exponentially. Eventually the gradient becomes so small that it’s practically zero.

When that happens:

* Weights stop updating
* Neurons stop learning
* Your model gets stuck
* Early layers (closer to the input) get almost **no gradient signal**

Methods To Solve:  
✅ **Use ReLU** instead of sigmoid/tanh  
ReLU’s derivative is 1 for positive values — no shrinking

✅ **Batch Normalization**  
Helps keep the activations and gradients in a healthy range

✅ **Residual Connections (ResNets)**  
Skip connections help gradients flow more easily through deep networks

✅ **Careful weight initialization**  
[Requires Investigation] Methods like He or Xavier initialization aim to preserve the scale of activations and gradients

1. **Exploding Gradient**Exploding gradients occur when the **gradients (partial derivatives of the loss with respect to the model parameters)** become **very large during backpropagation**. These large values can cause:

Model weights to grow excessively.

Training to become unstable.

Loss to oscillate wildly or become NaN.  
  
**How to Fix or Prevent Exploding Gradients**

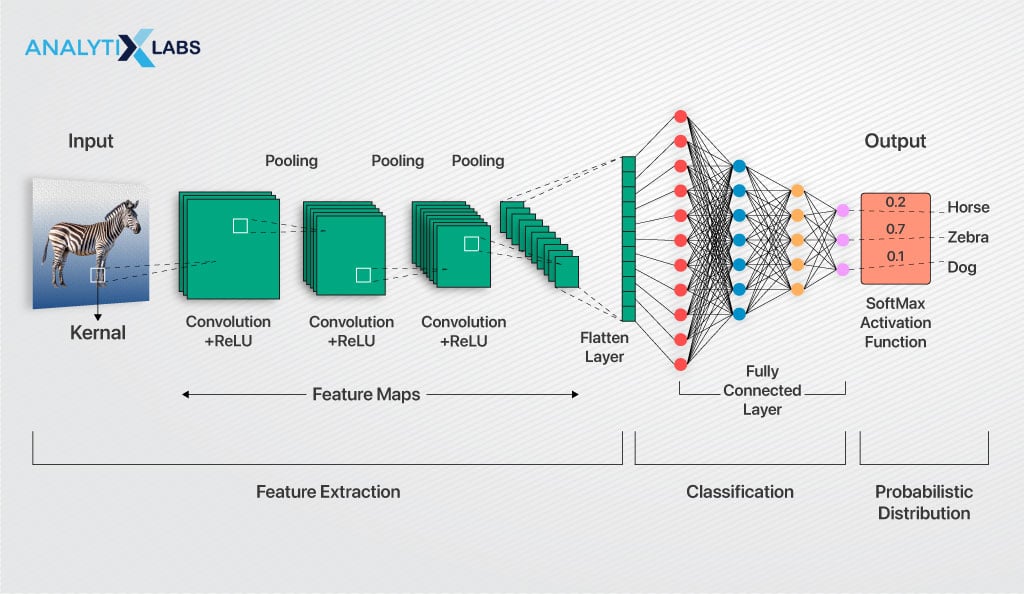
1. **Gradient Clipping**  
   Cap the gradients to a maximum value to prevent them from getting too large.

python

CopyEdit

torch.nn.utils.clip\_grad\_norm\_(model.parameters(), max\_norm=1.0)

1. **Weight Regularization**  
   Techniques like L2 regularization help keep weights small.
2. **Use Better Activation Functions**  
   Like ReLU or variants that are less prone to derivative explosion than, say, tanh or sigmoid.
3. **Careful Initialization**  
   Initialize weights with methods like Xavier or He initialization to avoid large initial gradients.
4. **Use Residual Connections**  
   Especially in very deep networks (e.g., ResNets), these help with gradient flow.
5. **Weights & Biases**
6. **Convolutional Neural Networks**



* 1. **Definition**

Convolutional Neural Networks (CNNs) are a type of deep learning model specifically designed for processing structured grid-like data, such as images.

* 1. **Working**

These layers apply filters (kernels) to input images to extract important features like edges, textures, and patterns.

Each filter slides over the input (convolution operation), producing a feature map.

The pooling layers are used to reduce the spatial size of feature maps while retaining important information

After extracting features, the output is flattened and passed through dense layers.

The kernel is just a small matrix (e.g., 3×3 or 5×5) that slides over the input image (This kernel is trained as goal of this process).

It performs a convolution operation by multiplying its values with the pixel values of the input and summing them up. This results in a new matrix called a **feature map**.

Pooling is not a learnable operation—it’s just a way to reduce the size of the feature map.

It takes the feature map (produced by convolution) and applies an operation like:

* Max Pooling → Takes the maximum value in a small window (e.g., 2×2).
* Average Pooling → Takes the average of values in a small window.
  1. **Need**

Images have a spatial structure (e.g., pixels in a face are related to nearby pixels).

Fully connected layers treat all pixels as independent, losing important spatial context.

Filters detect **edges, textures, and shapes** locally and pass them deeper into the network.

* 1. **Applications:**
     1. **Style Transfer**
  2. **Components**
     1. **Convolution**
        1. Different Types:

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  | | --- | --- | --- | | **Conv1D** |  |  |  |  |  |  | | --- | --- | --- | |  |  |  |  |  |  |  | | --- | --- | --- | |  |  |  |  |  |  |  | | --- | --- | --- | |  |  |  |  |  |  | | --- | --- | |  |  | | Used for time series, audio, or NLP tasks | 1D sequences (e.g., speech, text embeddings) |
| **Conv2D** | Used for image processing | 2D data (e.g., grayscale/RGB images) |
| **Conv3D** | Used for volumetric data like medical imaging or videos | 3D data (e.g., MRI scans, video frames) |
| **Conv2DTranspose** | Used for **upsampling** (e.g., image segmentation, GANs) | 2D data, like Conv2D but increases spatial size |
| **Conv3DTranspose** | Used for **3D upsampling**, such as in medical image reconstruction | 3D data, like Conv3D but increases spatial size |

* + - 1. **Params**
         1. **Kernel: eg. (5,5)** specifies the kernel size used
         2. **Padding:** Refers to adding extra pixels around the input image before applying the convolution operation. This is done to control the spatial size (height & width) of the output feature map.

**Types:**

Valid: No padding

Same: Zero Padding (output size = input size)

* + - * 1. **Number of filters:** Number of different feature maps generated
      1. v
    1. **Pooling**
    2. **Padding**
  1. **Historical Performance  
     **

1. **Residual Connections**

Residual connections (also called skip connections) are a technique introduced in ResNet (Residual Networks) that help train very deep neural networks by allowing the network to "skip" one or more layers.  
  
In very deep networks:

* Training gets harder due to **vanishing/exploding gradients**
* The network may start performing worse as depth increases (which is counterintuitive)

Residual connections allow the network to **learn residuals**—that is, how much to change the input rather than learning the full transformation from scratch. This helps:

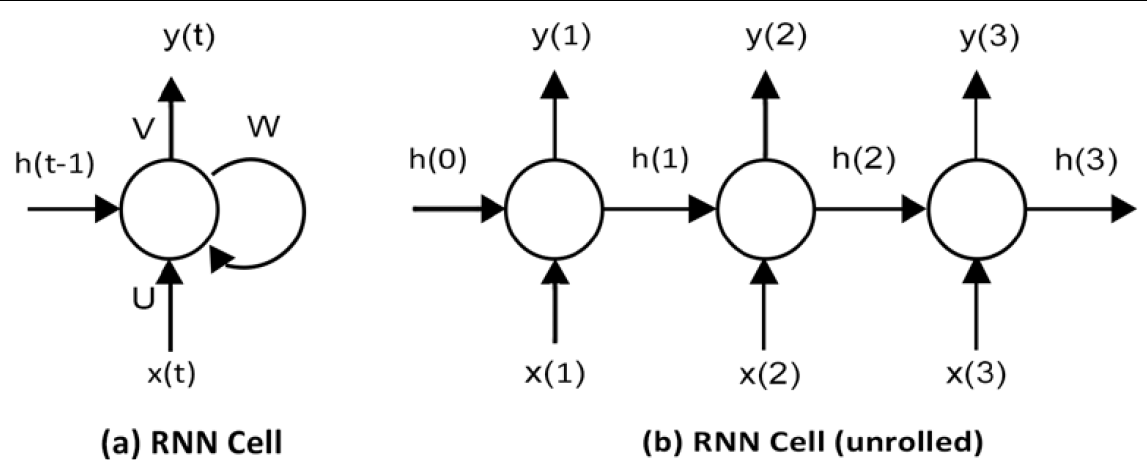
* Improve gradient flow (prevent vanishing/exploding gradients)
* Make optimization easier
* Enable successful training of networks with **100+ layers**

1. **RNN**
   1. **Working**

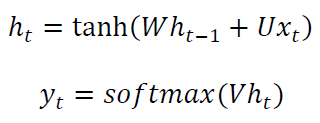
Used for sequences where the value of last element does predict next.

The value of the hidden state at any point in time is a function of the value of the hidden state at the previous time step, and the value of the input at the current time step.



*ht* and *ht*-1 are the values of the hidden states at the time *t* and *t-1*  
The output vector *yt* at time *t* is the product of the weight matrix *V* and the hidden state *ht*, passed

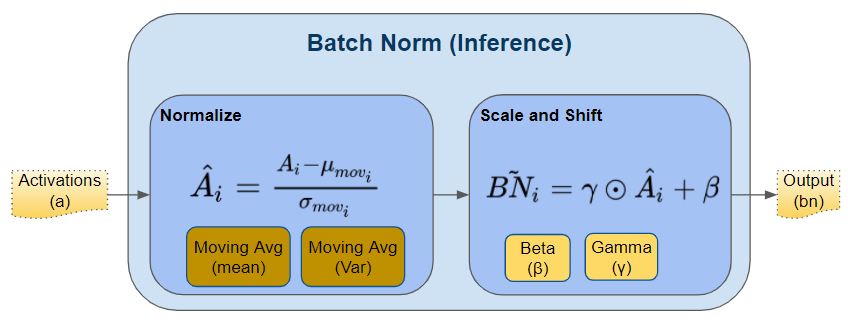
through a SoftMax activation, such that the resulting vector is a set of output probabilities.



* 1. **Backpropagation through time (BPTT)**

Just like traditional neural networks, training RNNs also involves the backpropagation of gradients.

1. **Normalization [Mentioned in Data Processing]**
   1. **Batch Normalization**

Batch Normalization (BatchNorm) is a technique used in deep learning to improve training speed, stability, and performance by normalizing the inputs of each layer.

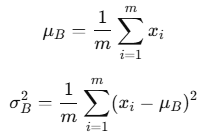
It helps reduce internal covariate shift, making training more efficient. [Requires Investigation]

Batch normalization (BatchNorm) is typically applied **to the inputs of a layer**, but it operates on the **outputs of the previous layer**

* **In fully connected (dense) layers:** Batch Norm is applied to the activations (outputs) of the previous layer **before** passing them through the next layer.
* **In convolutional layers:** It is applied to each channel independently before passing through the activation function.

Steps:

* + - 1. **Calculate the Mean and Variance** for each feature across a mini-batch



* + - 1. Normalize the Input



* + - 1. Scale and Shift with Learnable Parameters



* γ (scale) and β\betaβ (shift) are **trainable parameters** that allow the model to learn an optimal distribution.
* If γ=sqrt(σ2) and β= μB​, the model can recover the original input distribution.
  1. **Questions**
     1. **Why Are γ and β Parameters Needed?**
        1. Without γ and β, BatchNorm would force activations to always have mean 0 and variance 1.
        2. Some layers (e.g., ReLU, sigmoid) work better with specific distributions.
        3. and β allow the network to retain representational power while benefiting from normalization.
        4. The parameters automatically adapt to what the layer finds best during training

1. **Gradient Descent**
   1. **Stochastic**
   2. **Batch & Mini Batch**
2. **Epoch**
3. **Data Augmentation**
   1. **Image Data Augmentation**
4. **Weight Initialization**
   1. **Why Initialize with anything other than zero?**
   2. **Why initialize with uniform distribution?**
   3. **Types**
5. **Loss Functions**
   1. **Types**
      1. **Regression**
         1. **Mean Square Error**
         2. **Mean Absolute Error**
         3. **Mean Squared Log Error**
      2. **Classification**
         1. **Binary Cross Entropy**
         2. **Categorical Cross Entropy**
         3. **Spare Categorical Cross Entropy**
         4. **KL Divergence**
   2. **Vanishing gradient w.r.t loss function**
6. **Metrics displayed vs Used during training**
7. **Layers [TensorFlow]**
   1. **Core**
      1. **tf.keras.layers.Dense**
      2. **tf.keras.layers.Activation**
      3. **tf.keras.layers.Dropout**
      4. **tf.keras.layers.Flatten**
      5. **tf.keras.layers.Reshape**
      6. **tf.keras.layers.Lambda**
8. **Optimizers**
   1. **SGD**
   2. **RMSProp**
   3. **Adam**
9. **Regularization – [Mentioned in Machine Learning Implementation]**
10. **Transfer Learning**

Transfer learning is a machine learning technique where a model trained on one task is reused (partially or fully) on a different but related task.

Instead of training a model from scratch, you start with a pretrained model that has already learned useful features from a large dataset, and you fine-tune it for your specific task.

Steps:

**1. Selecting a Pretrained Model**

The first step is to select an appropriate pretrained model. These models are often trained on large-scale datasets such as ImageNet (for images), Common Crawl or Wikipedia (for natural language), or AudioSet (for audio). Common pretrained architectures include ResNet, EfficientNet, and VGG for image tasks, or BERT, GPT, and RoBERTa for text-based tasks.

**2. Model Architecture Analysis**

A pretrained model generally consists of two main components:

* **Feature extractor**: The initial layers (often convolutional or embedding layers) that learn to identify general features such as edges, shapes, textures, or word relationships.
* **Classifier head**: The final layers (often fully connected or linear layers) responsible for task-specific predictions, such as class labels.

In transfer learning, the feature extractor is reused (often unchanged), while the classifier head is replaced or modified to suit the new task.

**3. Freezing and Fine-Tuning**

There are two main approaches for reusing a pretrained model:

* **Feature Extraction (Freezing)**:
  + The feature extractor’s weights are **frozen**, meaning they are not updated during training.
  + A new classification head is appended and trained on the target dataset.
  + This approach is useful when the target dataset is small or closely related to the original dataset.
* **Fine-Tuning**:
  + After replacing the classification head and training it for a few epochs, some or all of the pretrained layers are **unfrozen**.
  + The entire model, or select layers, are retrained with a smaller learning rate to refine feature representations.
  + Fine-tuning is beneficial when the target task differs significantly from the source task.

**4. Data Preparation**

The target dataset must be preprocessed to match the input format expected by the pretrained model. For instance:

* Images must be resized to the same dimensions the original model was trained on.
* Text inputs may require tokenization using the original model’s vocabulary and tokenizer.

In some cases, normalization or augmentation techniques consistent with the pretrained model’s training procedure are also necessary.

1. **Residual Networks**
   1. **Fastforward Connections**
   2. **Benefits:**
      1. **Faster to train than traditional CNN**
      2. **Address Vanishing Gradient**
2. **Important Models in History:**
   1. DCNN
   2. VGG16
   3. AlexNet
   4. Xception

Important Questions

Topics To Study:

1. Heatmap/Correlation b/w data variables
2. EarlyStopping
3. Data Augmentation

Topics/Statements of Possible Interest:

1. Sparsity in CNN
2. Softmax vs Sigmoid
3. Increasing the number of filters in deeper layers is a common technique in deep learning (CNN).
4. Importance of non linearity in deep learning
5. Normalizing data regularly vs Batch Norm
6. Benchmark Models