MACHINE LEARNING & DEEP LEARNING

1 IntroAI, IntroML and IntroProb

File name. **Lecture 2 - Feb24-27 2025 -IntroAIIntroML-IntroProb.pdf**

1. List and briefly describe the three main ML paradigms
The three main Machine Learning paradigms are:

- **Supervised Learning**. This paradigm involves learning a function that maps an input to an output based on example input-output pairs (labeled data). The goal is to learn a mapping from observed data (Experience E) to perform a Task T, measured by a Performance measure P, such that performance improves with experience. It is used when you have examples of what you want (*if x then do y*) and you collect many pairs (x_i, y_i) to estimate a function f such that $f(x_i) = y_i$. It includes tasks such as Classification, which predicts a categorical label (like Binary Classification with outcomes $\{-1, 1\}$ or Multiclass Classification with outcomes $\{1, ..., k\}$), and Regression, which predicts a continuous value.
- **Unsupervised Learning**. In this paradigm, the learning algorithm works with unlabeled data to find hidden patterns or intrinsic structures within the input. The goal is to detect patterns in data without explicit output labels. Common tasks include Clustering, which groups similar data points together based on some measure of similarity, and Dimensionality Reduction, which reduces the number of random variables under consideration by obtaining a set of principal variables.
- **Reinforcement Learning**. This paradigm involves an agent learning how to behave in an environment by performing actions and receiving feedback in the form of rewards or penalties. The agent learns through trial and error to make a sequence of decisions that maximize the cumulative reward.

Cited slides: 19, 32, 33, 34, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50.

2. Describe the ML paradigm of supervised learning and give an example of supervised learning algorithm presented during the course
The supervised learning paradigm is a type of machine learning where the algorithm learns from a dataset of labeled examples. This means that for each input data point (Experience E), the corresponding correct output or target value is provided. The goal of the supervised learning algorithm is to learn a mapping or function (f) from the input features (x_i) to the output labels (y_i), such that the function can accurately predict the output for new, unseen input data. This is essentially about estimating a function f such that $f(x_i) = y_i$ based on collected pairs (x_i, y_i). The learning process is guided by minimizing the difference between the model's predictions and the true labels, aiming to improve performance (P) on a given Task (T) with more experience (E). Supervised learning is applicable in scenarios where historical data with known outcomes is available, allowing the model to learn the relationship between inputs and outputs. Common tasks within this paradigm include Classification (predicting discrete categories) and Regression (predicting continuous values).
An example of a supervised learning algorithm presented that can be used for classification, a supervised task, is the **Naïve Bayes Classifier**. This algorithm is based on Bayes' theorem and assumes conditional independence between the features given the class label. It estimates the probabilities needed for classification from the training data.
Cited slides: 19, 32, 40, 41, 42, 44, 45, 46, 63, 64, 65, 66, 67, 68.

3. Describe the difference between supervised and unsupervised ML algorithms
The primary difference between supervised and unsupervised Machine Learning algorithms lies in the type of data they use for training and the nature of the problem they are designed to solve.

- **Supervised Learning** algorithms learn from **labeled data**. This means the training dataset includes pairs of input data (x_i) and corresponding correct output labels or values (y_i). The objective is to learn a function that can predict the output for new, unseen inputs based on the patterns discovered in the labeled examples. Tasks typically associated with supervised learning include Classification (predicting a categorical label) and Regression (predicting a continuous value).

- **Unsupervised Learning** algorithms, on the other hand, work with **unlabeled data**. The training data consists only of input data points, without any corresponding output labels or target values. The goal is not to predict a specific output, but rather to explore the data and find hidden patterns, structures, or relationships within it. Common tasks in unsupervised learning include Clustering (grouping similar data points) and Dimensionality Reduction (reducing the number of variables while retaining important information).

In essence, supervised learning is like learning with a teacher who provides the correct answers, while unsupervised learning is like exploring the data on your own to find interesting insights.
Cited slides: 40, 41, 42.

4. Describe the ML paradigm of unsupervised learning and give an example of unsupervised learning algorithm presented during the course
The unsupervised learning paradigm is a type of machine learning that deals with data that does not have pre-assigned labels or target outputs (unlabeled data). The primary objective of unsupervised learning is to explore the inherent structure, patterns, and relationships within the dataset itself, without guidance from known outcomes. Unlike supervised learning, there is no error signal based on correct outputs; instead, the algorithms work to discover interesting properties or organizations within the data. The goal is to detect patterns in data. This paradigm is useful for tasks such as summarizing and understanding the data, finding hidden groupings, or reducing the complexity of the data representation.
The course material presented tasks within the unsupervised learning paradigm, including **Clustering** and **Dimensionality Reduction**. Clustering involves grouping data points into clusters such that data points within the same cluster are more similar to each other than to those in other clusters. Dimensionality Reduction aims to reduce the number of random variables by finding a lower-dimensional representation that captures the essential information in the data.
Cited slides: 32, 33, 41, 42.

5. Describe what a classification problem is, and make a simple binary classification example
A classification problem, within the supervised learning paradigm, is a task where the goal is to predict a categorical label or class for a given input data point. This means the output variable y is a discrete value belonging to a predefined set of categories. Classification problems are distinguished by the nature of their output: predicting which category an item belongs to.
A simple type is **Binary Classification**, where there are only two possible output classes, often represented as $y \in \{-1, 1\}$ or $\{0, 1\}$. An example of a simple binary classification problem can be visualized as data points in a 2D space, where each point belongs to one of two classes. The objective is to find a boundary or decision rule that separates the points of one class from the points of the other class. For instance, given a set of data points with two features plotted on a graph, where some points are labeled as Class -1 and others as Class 1, a binary classification algorithm aims to learn a line or curve that effectively divides the graph into two regions, allowing us to predict the class (-1 or 1) for any new point based on which region it falls into.
Another type is Multiclass Classification, where there are more than two possible output classes, represented as $y \in \{1, ..., k\}$ for $k > 2$.
Cited slides: 27, 29, 30, 31, 42.

6. Describe the Bayes' Rule, its elements, and why is it useful. Name a classification algorithm whose formulation exploits the Bayes' Rule
Bayes' Rule is a fundamental theorem in probability theory that relates conditional probabilities. It

provides a way to update the probability of a hypothesis based on new evidence. The rule is stated as:

$$P(X|Y) = \frac{P(Y|X)P(X)}{P(Y)}$$

where X and Y are events or random variables.
The elements of Bayes' Rule, as often represented in the context of updating beliefs about parameters or hypotheses (θ) given data (D), are:

- $P(\theta|D)$: The **Posterior** probability, which is the updated probability of the hypothesis (θ) after observing the data (D).
- $P(D|\theta)$: The **Likelihood**, which is the probability of observing the data (D) given that the hypothesis (θ) is true.
- $P(\theta)$: The **Prior** probability, which is the initial probability of the hypothesis (θ) before observing the data (D).
- $P(D)$: The marginal probability of the data, which acts as a normalizing constant.

Bayes' Rule is useful because it allows us to reverse conditional probabilities and, importantly, to incorporate prior knowledge or beliefs about a hypothesis and update those beliefs systematically when new data becomes available. It shows that the posterior probability is proportional to the likelihood times the prior ($P(\theta|D) \propto P(D|\theta)P(\theta)$). This makes it a basis for probabilistic inference and reasoning under uncertainty.
A classification algorithm whose formulation exploits Bayes' Rule is the **Naïve Bayes Classifier**. This algorithm uses Bayes' theorem to calculate the probability of a given sample belonging to each class and selects the class with the highest probability as the prediction. It makes a simplifying assumption that the features are conditionally independent given the class label to make the probability estimation tractable.
Cited slides: 56, 60, 61, 62, 63, 68, 88, 90.

7. Illustrate the Naïve Bayes Classifier and its main assumption
The Naïve Bayes Classifier is a supervised learning algorithm used for classification that is based on applying Bayes' theorem. It works by estimating the probability of a given data instance belonging to each class and then classifying the instance into the class with the highest probability. The core idea involves learning the prior probability of each class, $P(y)$, and the conditional probability of each feature given the class, $P(x_i|y)$, from the training data.
To classify a new instance with features $x = (x_1, x_2, ..., x_d)$, the algorithm calculates the posterior probability $P(y|x)$ for each class y using a form derived from Bayes' Rule. This calculation is made tractable by introducing the main assumption.
The **main assumption** of the Naïve Bayes Classifier is that the **features are conditionally independent given the class label**. This means that the value of any one feature is independent of the value of any other feature, given that the class of the instance is known. Mathematically, this assumption allows the conditional probability of the features given the class, $P(x_1, x_2, ..., x_d|y)$, to be simplified as the product of the individual conditional probabilities:

$$P(x|y) = P(x_1, x_2, ..., x_d|y) = \prod_{i=1}^{d} P(x_i|y)$$

This simplification makes the estimation of $P(x|y)$ much easier, as it only requires estimating the probabilities for each feature independently given the class, rather than estimating the joint probability of all features. Despite this strong independence assumption, which is often not true in real-world data, the Naïve Bayes Classifier frequently performs well in practice.
Cited slides: 63, 64, 65, 66, 67, 68.

8. Explain the Bias/Variance Decomposition Theorem seen during the course. No proof required
The Bias/Variance Decomposition Theorem is a way to analyze the expected prediction error of a machine learning model, particularly in regression. It states that the expected squared error of a model's prediction can be decomposed into three parts: the squared bias of the model, the variance of the model, and the irreducible error. This decomposition helps understand where the error in a model's predictions comes from.
The decomposition can be expressed as:

$$E[(y - \hat{f}(x))^2] = (E[\hat{f}(x)] - f(x))^2 + E[(\hat{f}(x) - E[\hat{f}(x)])^2] + E[\epsilon^2]$$

Where:

- $E[(y - \hat{f}(x))^2]$ is the expected Mean Squared Error (MSE) of the prediction $\hat{f}(x)$ compared to the true value y (which is $f(x) + \epsilon$, where $f(x)$ is the true underlying function and ϵ is the irreducible noise).

- $(E[\hat{f}(x)] - f(x))^2$ is the **squared Bias**. The bias of an estimator $\hat{\theta}$ for a parameter θ is defined as $Bias(\hat{\theta}) = E[\hat{\theta}] - \theta$. Bias in this context measures how far the average prediction of our model, $E[\hat{f}(x)]$, is from the true underlying function $f(x)$. High bias suggests the model is making strong assumptions about the data (e.g., being too simple) and might be underfitting.

- $E[(\hat{f}(x) - E[\hat{f}(x)])^2]$ is the **Variance**. The variance indicates how much we expect the estimator (in this case, the model's prediction $\hat{f}(x)$) to vary as a function of different training data samples. High variance suggests the model is very sensitive to the specific training set it received and might be overfitting to the noise in the training data.

- $E[\epsilon^2]$ is the **irreducible error**. This is the part of the error that cannot be reduced by any model because it is due to inherent noise or randomness in the data generation process itself.

In essence, the theorem shows that the total expected error is the sum of error from bias, error from variance, and inherent noise in the data.
Cited slides: 78, 81, 86.

2 Labs overview: data preprocessing, pipeline, cross validation

1. Describe the key stages involved in a typical deep learning project workflow, from initial problem understanding to having a usable model. For each stage, briefly explain its main purpose.
A typical deep learning project workflow involves several key stages, moving from understanding the problem to deploying a functional model. These stages are:

- **Define the Task**. This initial stage involves clearly understanding the problem you are trying to solve with deep learning. It includes defining the objectives, the type of problem (e.g., classification, regression), and the desired outcome. The main purpose is to establish a clear goal and scope for the project.

- **Data Preparation**. This is a crucial stage that involves collecting, cleaning, organizing, and preprocessing the data. It includes steps like data collection, annotation (if required for supervised learning), data exploration, cleaning missing or erroneous data, and preparing the data in a format suitable for training (e.g., splitting into training, validation, and test sets, creating data loaders, applying augmentations). The main purpose is to ensure the data is high-quality and properly formatted for the model.

- **Model Selection**. In this stage, you choose or design the appropriate deep learning model architecture for your specific task and data. This involves considering different types of neural networks and their layers based on the nature of the input data (e.g., CNNs for images, RNNs for sequences) and the problem complexity. The main purpose is to select a model capable of learning from the data and performing the task effectively.

- **Training & Validation**. This is where the model learns from the prepared data. It involves feeding the training data through the model, calculating the loss (the difference between predicted and true outputs), and using an optimizer to update the model's weights to minimize the loss. During training, the model's performance is regularly evaluated on a separate validation set to tune hyperparameters and prevent overfitting. The main purpose is to train the model to accurately map inputs to outputs and ensure it generalizes well to unseen data.
- **Deployment**. The final stage involves making the trained and validated model available for use in a real-world application or environment. This might involve integrating the model into a software application, a web service, or an edge device. Saving the model properly is part of this stage. The main purpose is to allow the model to make predictions on new data and deliver value based on its learned capabilities.

Cited slides: 4, 6, 7, 8, 37, 38, 45, 46, 59.

2. Before diving into data preprocessing for a deep learning project, explain three critical characteristics or potential issues related to your dataset that you should investigate

Before starting the technical steps of data preprocessing in a deep learning project, it is critical to first explore and understand your dataset thoroughly. This initial investigation helps uncover potential issues that need to be addressed during preprocessing to ensure the model can learn effectively. Based on the course material, three critical characteristics or potential issues to investigate include:

- **Missing Values**. It is important to check if the dataset contains any missing data points or values for certain features. Missing data can interfere with the training process and lead to biased models. Identifying the presence and extent of missing values is a crucial step before deciding on strategies to handle them (e.g., imputation or removal).
- **Outliers**. Outliers are data points that significantly deviate from other observations. These extreme values can disproportionately influence model training, especially sensitive algorithms, leading to skewed results or poor generalization. Investigating the presence and nature of outliers helps determine whether they should be removed, transformed, or treated specifically.
- **Data Imbalance**. Understanding the distribution of classes in your dataset is essential, particularly for classification tasks. Data imbalance occurs when the number of samples in one class is significantly lower than in others. Training a model on imbalanced data can lead to a model that performs well on the majority class but poorly on the minority class. Investigating data imbalance helps inform strategies like oversampling, undersampling, or using specialized loss functions.

Exploring these aspects of the data helps ensure the data is suitable for training and guides the subsequent preprocessing steps.
Cited slides: 15, 16, 17, 20, 22, 23.

3. Explain why raw data, particularly images, generally requires preprocessing before being used to train a deep learning model. Describe at least two distinct common preprocessing steps and one data augmentation technique, outlining the specific purpose or benefit of each.

Raw data, especially images, generally requires preprocessing before being used to train a deep learning model for several key reasons. Deep learning models, particularly neural networks, work with numerical tensors as input. Raw data often exists in various formats, scales, and resolutions, which are not directly compatible with the fixed input requirements of these models. Preprocessing converts the data into a suitable, consistent numerical format and scale that the model can efficiently process. It also helps to mitigate issues present in the raw data, leading to more stable training and better model performance. For images, this is crucial because they can vary greatly in size, aspect ratio, and pixel value ranges.

Two distinct common preprocessing steps for images, as described in the course material, are:

- **Resizing/Cropping**. The purpose of resizing and cropping is to ensure that all input images have a consistent spatial dimension (height and width) required by the deep learning model architecture. Neural networks typically expect inputs of a fixed size. Resizing scales the image to the desired dimensions, while cropping can be used to extract a specific region of interest or achieve the required size. The benefit is ensuring compatibility with the model's input layer.
- **Normalization/Standardization**. Normalization and standardization involve scaling the pixel values of the images to a standard range (e.g., [0, 1] or [-1, 1]) or distribution (e.g., zero mean and unit variance). The purpose is to bring the pixel values to a consistent scale across all images and features. The benefit is that it can help the optimization algorithm converge faster during training and improve the overall stability and performance of the model.

In addition to standard preprocessing, **Data Augmentation** is a technique used to increase the size and diversity of the training dataset by creating modified versions of existing data. This helps the model generalize better to unseen data and reduces overfitting.
One data augmentation technique is **Flipping**. This involves creating a mirrored version of an image (e.g., horizontal or vertical flip). The specific purpose is to expose the model to variations of the training data that are likely to occur in real-world scenarios but might not be present in the original dataset. The benefit is that it helps the model become more robust to variations in orientation and improves its ability to generalize.
Cited slides: 7, 24, 25, 26, 28.

4. Explain why raw data, particularly images, generally requires preprocessing before being used to train a deep learning model. Describe at least two distinct common preprocessing steps and one data augmentation technique, outlining the specific purpose or benefit of each.
Raw data, especially images, generally requires preprocessing before being used to train a deep learning model for several key reasons. Deep learning models, particularly neural networks, work with numerical tensors as input. Raw data often exists in various formats, scales, and resolutions, which are not directly compatible with the fixed input requirements of these models. Preprocessing converts the data into a suitable, consistent numerical format and scale that the model can efficiently process. It also helps to mitigate issues present in the raw data, leading to more stable training and better model performance. For images, this is crucial because they can vary greatly in size, aspect ratio, and pixel value ranges.
Two distinct common preprocessing steps for images, as described in the course material, are:

- **Resizing/Cropping**. The purpose of resizing and cropping is to ensure that all input images have a consistent spatial dimension (height and width) required by the deep learning model architecture. Neural networks typically expect inputs of a fixed size. Resizing scales the image to the desired dimensions, while cropping can be used to extract a specific region of interest or achieve the required size. The benefit is ensuring compatibility with the model's input layer.
- **Normalization/Standardization**. Normalization and standardization involve scaling the pixel values of the images to a standard range (e.g., [0, 1] or [-1, 1]) or distribution (e.g., zero mean and unit variance). The purpose is to bring the pixel values to a consistent scale across all images and features. The benefit is that it can help the optimization algorithm converge faster during training and improve the overall stability and performance of the model.

In addition to standard preprocessing, **Data Augmentation** is a technique used to increase the size and diversity of the training dataset by creating modified versions of existing data. This helps the model generalize better to unseen data and reduces overfitting.
One data augmentation technique is **Flipping**. This involves creating a mirrored version of an image (e.g., horizontal or vertical flip). The specific purpose is to expose the model to variations of the training data that are likely to occur in real-world scenarios but might not be present in the original dataset. The benefit is that it helps the model become more robust to variations in orientation and improves its ability

to generalize.
Cited slides: 7, 24, 25, 26, 28.

5. Describe the iterative process of training a deep learning model. Explain what fundamentally occurs during this training phase. What is the ultimate aim of training?

The training of a deep learning model is an iterative process that involves repeatedly exposing the model to the training data and adjusting its internal parameters to improve its performance. This process is typically structured around epochs and batches. An epoch represents one full pass through the entire training dataset, while a batch is a subset of the training data used in a single iteration of the training loop.

During each iteration of the training phase (for each batch), the following fundamental steps occur:

- **Forward Pass**. The input data from the current batch is fed through the layers of the neural network. The model performs computations at each layer, using its current weights and biases, to generate a prediction or output.
- **Loss Calculation**. A loss function is used to measure the difference or error between the model's predictions and the true target values (ground truth) for the current batch. The loss function quantifies how well the model is currently performing on this data.
- **Backward Pass (Backpropagation)**. Based on the calculated loss, the gradients of the loss with respect to each of the model's parameters (weights and biases) are computed. This process, known as backpropagation, determines how much each parameter contributed to the error and in which direction it should be adjusted to reduce the loss.
- **Optimizer Step**. An optimizer algorithm uses the calculated gradients to update the model's parameters. The optimizer adjusts the weights and biases according to its specific rules (e.g., using a learning rate) with the goal of moving the model towards a state where the loss is minimized.

This cycle of forward pass, loss calculation, backward pass, and parameter update is repeated for every batch in the training set. Once all batches have been processed, one epoch is complete. The entire process is then repeated for a predetermined number of epochs.

The ultimate aim of this iterative training process is to find a set of model parameters (weights and biases) that minimizes the loss function across the entire training dataset. By minimizing the loss, the model learns to make accurate predictions and effectively capture the underlying patterns and relationships in the data, thereby becoming capable of performing the intended task well on new, unseen data.
Cited slides: 45, 46, 47, 48, 49, 50, 51, 52, 53.

6. What are hyperparameters in the context of designing and training deep learning models? Provide and discuss three distinct examples of common hyperparameters.

In the context of designing and training deep learning models, hyperparameters are parameters that are not learned from the training data itself by the optimization algorithm. Instead, they are set prior to the training process and define aspects of the model architecture or the training procedure. They are distinct from the model's parameters (such as weights and biases), which are adjusted iteratively during training based on the data and the loss function. The choice of hyperparameters significantly impacts the model's performance and the training process.

Three distinct examples of common hyperparameters are:

- **Learning Rate**. This is a crucial hyperparameter used by the optimizer. It determines the step size at which the model's parameters are updated during the training process based on the gradients. A high learning rate can cause the training to diverge or oscillate around the minimum loss, while a very low learning rate can make the training process slow to converge.
- **Batch Size**. This hyperparameter defines the number of data samples that are processed together in one forward and backward pass during training before the model's parameters are updated. The dataset is typically divided into batches. The batch size affects the memory consumption, the speed

of training, and the smoothness of the optimization updates.

- **Number of Layers (as part of Architecture)**. This is a hyperparameter that defines the depth of the neural network model. It is a structural hyperparameter determined during the model selection phase. The number of layers, along with the number of units (neurons or filters) per layer and the type of layers, dictates the model's capacity to learn complex representations and patterns from the data.

Tuning these and other hyperparameters is often necessary to achieve optimal performance for a specific task and dataset.
Cited slides: 27, 38, 47, 51, 54.

7. The process of developing a successful deep learning model is often described as highly iterative, involving continuous "improving through experimentation." Explain what this "iterative nature" means in the practical context of a deep learning project.
In the practical context of a deep learning project, the "iterative nature" of the workflow means that developing a successful model is not a linear process completed in a single pass through the defined stages. Instead, it involves repeated cycles of experimentation, evaluation, and refinement across various aspects of the project.

This "improving through experimentation" means that after an initial pass through stages like data preparation, model selection, and training, the results are evaluated (particularly using the validation set). Based on this evaluation, insights are gained about the model's performance, limitations, and potential areas for improvement. This leads back to revisiting earlier stages. For example:

- Performance might be poor, suggesting the need to **revisit Data Preparation** (e.g., collect more data, perform different cleaning or augmentation).
- The model might be underfitting or overfitting, indicating the need to **revisit Model Selection** (e.g., try a different architecture, add or remove layers).
- The training process might be unstable or slow, highlighting the need for **Hyperparameter Tuning** (e.g., trying different learning rates, batch sizes, optimizers), which itself is an iterative process involving training and evaluating the model multiple times with different settings.

The process is iterative because you continuously experiment with different configurations (data preprocessing techniques, model architectures, hyperparameters), train the model, evaluate its performance, analyze the results, and use that analysis to inform the next set of experiments. This cycle is repeated, gradually refining the model and the overall pipeline, until satisfactory performance is achieved on the validation criteria. Even after initial deployment, monitoring performance in the real world can lead to further iterations.
Cited slides: 4, 27, 37, 38, 45, 54, 55, 59.

8. Explain why it would be highly problematic to directly use the test set for hyperparameter tuning. How can this issue be avoided when building a ML pipeline?
Using the test set directly for hyperparameter tuning in a machine learning pipeline is highly problematic because it compromises the ability of the test set to provide an unbiased evaluation of the final model's performance on truly unseen data. The test set is specifically reserved to measure how well the trained and tuned model will generalize to new examples it has never encountered during development.

Hyperparameter tuning is an iterative process that involves training and evaluating the model multiple times with different settings. If the test set is used for these repeated evaluations, the model and its hyperparameters will effectively be optimized to perform well specifically on the nuances and patterns present in that particular test set. This process can lead to overfitting to the test set. Consequently, the performance metrics obtained on the test set will give an overly optimistic estimate of the model's real-world performance, as the test set no longer serves as an independent measure of generalization to completely novel data. The test set loses its purpose as a final, unbiased benchmark.

This issue is avoided by properly splitting the dataset into three distinct sets: a training set, a validation set, and a test set. The key is to use the **validation set** for hyperparameter tuning and evaluation during the model development phase. The validation set provides an unbiased evaluation of the model's performance on data not used for training while hyperparameters are being adjusted. The **test set** is then used only once, at the very end of the project, after all hyperparameter tuning and model selection are complete, to get a final, reliable measure of the model's generalization capability on data that has played no role in the training or tuning process.
Cited slides: 27, 54.

3 Learning Theory and Unsupervised Learning

1. What is a loss function and why is its choice crucial when tackling a Machine Learning problem?
A loss function, also known as a cost function, is a mathematical function that measures the difference between the predicted output of a machine learning model and the actual true output for a given data instance. Essentially, it quantifies the error made by the model. During the training process, the primary goal of the learning algorithm is to find the set of model parameters that minimize this loss function across the training dataset.
The choice of the loss function is crucial when tackling a Machine Learning problem because it directly dictates how the model learns and which types of errors it prioritizes minimizing. Different types of problems (e.g., regression versus classification) require different loss functions that appropriately reflect the nature of the output and the desired error characteristics. For example, a loss function suitable for measuring errors in continuous predictions (regression) would be different from one used to evaluate the correctness of class assignments (classification). An inappropriate choice of loss function can lead to a model that learns suboptimally, focuses on the wrong aspects of the data, or fails to capture the desired relationships, ultimately resulting in poor performance on the task.
Cited slides: 14.

2. Explain the concepts of True Risk and Empirical Risk. What is the goal of a learning algorithm in relation to them?
In the context of machine learning and learning theory, two key concepts related to evaluating model performance are True Risk and Empirical Risk.

True Risk (also known as Expected Risk) is the expected loss of a model's function (f) over the true underlying probability distribution of the data $(P(x, y))$. It represents the average error the model would make on all possible data instances, weighted by their probability of occurrence according to the true distribution. It is defined as $R(f) = E_{P(x,y)}[L(y, f(x))]$, where L is the loss function, y is the true output, and $f(x)$ is the predicted output for input x. The True Risk is the ideal measure of a model's performance on unseen data, but it is impossible to calculate directly because the true data distribution $P(x, y)$ is unknown.

Empirical Risk is the average loss of a model's function (f) calculated specifically on the observed training data. Given a training dataset of N samples $\{(x_i, y_i)\}_{i=1}^{N}$, the Empirical Risk is defined as $R_{emp}(f) = \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(x_i))$. It is an estimate of the True Risk based on the available training data.

The goal of a learning algorithm in relation to these concepts is to find a function f that minimizes the **True Risk**. However, since the True Risk cannot be computed directly, learning algorithms instead

minimize the **Empirical Risk** on the training data. The hope is that by minimizing the average error on the training set, the model will also perform well on unseen data, meaning the Empirical Risk will be a good approximation of the True Risk. For a sufficiently large number of training samples (N), the Empirical Risk tends to converge to the True Risk. The fundamental challenge is to minimize the Empirical Risk without overfitting to the training data, which would result in a low Empirical Risk but a high True Risk on unseen data.
Cited slides: 15, 16.

3. Describe the core idea behind Probably Approximately Correct (PAC) learning. What do the "Probably" and "Approximately Correct" aspects refer to, and how do the parameters epsilon () and delta () relate to these concepts?
The core idea behind Probably Approximately Correct (PAC) learning is to provide a theoretical framework for analyzing whether a given class of functions (hypotheses) can be learned efficiently from a reasonable amount of training data. It defines what it means for a machine learning algorithm to be considered "learnable". The framework guarantees that, with a high probability, a learning algorithm can learn a hypothesis that performs nearly as well as the best possible hypothesis within the class, given a sufficient number of training examples.

The terms "Probably" and "Approximately Correct" refer to specific aspects of the learning guarantee:

- **Approximately Correct**: This refers to the accuracy of the learned hypothesis. It means that the hypothesis learned by the algorithm should have a true error rate (True Risk, $R(h)$) that is close to the minimum possible error rate achievable within the hypothesis class. The parameter **epsilon ()** is the accuracy parameter. It quantifies how close the learned hypothesis's error rate is to the true minimum error. Specifically, a hypothesis is considered "approximately correct" if its true error is at most ϵ greater than the minimum possible error (or simply bounded by ϵ, depending on the specific definition used, but the core idea is bounding the error). A smaller value of ϵ means a stricter requirement for accuracy.
- **Probably**: This refers to the confidence in the learning process. Since learning is done from a finite sample of data, there is always a possibility, however small, that the training data is not representative, and the algorithm might learn a hypothesis with a high error rate. "Probably" means that the probability of the learning algorithm failing to find an approximately correct hypothesis is low. The parameter **delta ()** is the confidence parameter. It quantifies the maximum acceptable probability of failure – the probability that the learned hypothesis is *not* approximately correct (i.e., its true error is greater than ϵ). A smaller value of δ means higher confidence in the outcome of the learning process.

In essence, the PAC framework aims to show that a concept class is PAC-learnable if there exists an algorithm that, given sufficient data samples (which depends polynomially on $1/\epsilon$, $1/\delta$, the size of the hypothesis class, and the dimensionality), will output a hypothesis h such that the true error $R(h) \leq \epsilon$ with probability at least $1 - \delta$. Thus, ϵ controls the error tolerance, and δ controls the confidence level of the learning guarantee.
Cited slides: 17, 18.

4. Describe the Vapnik-Chervonenkis (VC) dimension of a hypothesis class. Explain the concept of "shattering" a set of points and how it is used to determine the VC dimension.
The Vapnik-Chervonenkis (VC) dimension is a measure of the capacity or complexity of a hypothesis class ($\mathcal{H}$). In the context of binary classification, it quantifies the richness or flexibility of the set of functions that the learning algorithm can choose from. A higher VC dimension means the hypothesis class is more complex and capable of fitting more complex patterns in the data.

The concept of **shattering** a set of points is central to the definition of VC dimension. A hypothesis class $\mathcal{H}$ is said to shatter a set of m data points $\{x_1, \ldots, x_m\}$ if, for every possible assignment of binary labels

($\{-1, 1\}$ or $\{0, 1\}$) to these m points, there exists at least one hypothesis h within the class $\mathcal{H}$ that can perfectly classify all the points according to that specific labeling. For a set of m points, there are 2^m possible distinct binary labelings. Shattering means the hypothesis class can realize all these 2^m labelings.

The VC dimension is determined based on this shattering concept. The **VC dimension** of a hypothesis class $\mathcal{H}$, denoted as $VC(\mathcal{H})$, is defined as the largest integer d such that there exists *at least one* set of d points that can be shattered by $\mathcal{H}$. If for any number of points m, no set of m points can be shattered by $\mathcal{H}$, then the VC dimension is considered infinite. The VC dimension is a property of the hypothesis class itself, not the specific dataset being used. It provides an upper bound on the capacity of the model family.
Cited slides: 19, 20, 21, 22, 23.

5. Describe the phenomena of overfitting and underfitting. What does it mean for a model to overfit the training data, and how does this typically affect its performance on previously unseen examples?
In machine learning, overfitting and underfitting are two common phenomena that describe how well a model generalizes from the training data to new, unseen data.
Underfitting occurs when a model is too simple or not complex enough to capture the underlying structure and patterns present in the training data. An underfitting model typically has high bias and low variance. It performs poorly not only on previously unseen data but also on the training data itself because it hasn't learned the fundamental relationships.
Overfitting, conversely, occurs when a model is too complex and learns the training data too well, including the noise, random fluctuations, and specific details that are unique to the training set but not representative of the true underlying data distribution. An overfitting model typically has low bias but high variance.
For a model to **overfit the training data** means it has essentially memorized the training examples, including their noise, rather than learning the generalizable underlying patterns. This results in the model achieving very high accuracy or low loss on the training set. However, when this overfitted model is presented with **previously unseen examples** (such as those in the validation or test set), its performance typically **degrades significantly**. Because the model has learned the quirks of the training data, these specific details do not apply to new data points, causing the model to make inaccurate predictions. In essence, an overfitted model lacks the ability to generalize well to data outside of its training set.
Cited slides: 16, 24, 25, 26.

6. Explain the K-fold Cross Validation procedure for model selection. For which data regime (i.e., training set size) can it be particularly useful?
K-fold Cross Validation is a robust procedure used for evaluating the performance of a machine learning model and for hyperparameter tuning, especially when the amount of available data is limited. The procedure works as follows:

1. The entire dataset is first divided into K equally sized subsets or "folds".
2. The procedure is then repeated K times, in K different "folds" or iterations.
3. In each iteration, one of the K folds is held out and used as the validation set, while the remaining K-1 folds are combined and used as the training set.
4. The machine learning model is trained on the combined K-1 training folds and then evaluated on the single held-out validation fold.
5. After completing all K iterations, where each fold has served as the validation set exactly once, the performance metric (e.g., accuracy, loss) from each of the K validation runs is averaged to produce a single, overall performance estimate for the model and its current set of hyperparameters.

This average performance score is considered a more reliable estimate of the model's generalization capability compared to a single train-validation split because every data point gets to be in a validation set

exactly once, and the model is trained on different combinations of the data.
K-fold Cross Validation is particularly useful when working with a **limited data regime**, meaning when the size of the available dataset is relatively small. In such cases, a simple single split into training, validation, and test sets might result in a validation set that is too small to be representative, or a training set that is too small to effectively train the model. K-fold Cross Validation helps to make the most of the limited data by using all data points for both training and validation across the different folds, leading to a more robust and reliable estimate of the model's performance and aiding in better model selection and hyperparameter tuning.
Cited slides: 28, 29.

7. Illustrate the key principles and step-by-step functioning of the Linear Regression algorithm.
Linear Regression is a fundamental supervised learning algorithm used for regression tasks, meaning it is designed to predict a continuous output variable based on one or more input features. Its key principle is the assumption that there is a linear relationship between the input features and the output variable. The algorithm aims to find the best-fitting linear model (a line in the case of one feature, or a hyperplane in the case of multiple features) that describes the relationship between the input data and the output. The linear model can be represented by the equation $\hat{y} = w_0 + w_1x_1 + w_2x_2 + \cdots + w_dx_d$, where $\hat{y}$ is the predicted output, $x_1, \ldots, x_d$ are the input features, $w_1, \ldots, w_d$ are the weights associated with each feature, and w_0 is the bias term (or intercept). In vector form, this is written as $\hat{y} = \mathbf{w}^T\mathbf{x}$, where $\mathbf{w}$ includes both weights and bias, and $\mathbf{x}$ includes the features and a bias term.

The functioning of the Linear Regression algorithm to find the best-fitting model involves determining the optimal values for the weights $\mathbf{w}$. This is typically done by minimizing a cost function that measures the discrepancy between the model's predictions and the actual true output values in the training data. A common choice for the cost function in Linear Regression is the Mean Squared Error (MSE), which calculates the average of the squared differences between the predicted values ($\hat{y}_i$) and the true values (y_i) for all training instances: $MSE = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$.
The optimal weights $\mathbf{w}$ that minimize the MSE can be found through different methods. One approach is the **Analytical Solution** using the Normal Equation, which provides a direct formula for calculating the optimal weights: $\mathbf{w} = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$. Another common approach, especially for larger datasets, is **Iterative Optimization** like Gradient Descent. In this step-by-step process, the algorithm starts with initial random weights and iteratively adjusts them in the direction that reduces the cost function. In each iteration, the gradient of the MSE with respect to the weights is calculated, and the weights are updated by taking a step proportional to the negative of the gradient (scaled by a learning rate). This iterative process continues until the cost function converges to a minimum, at which point the optimal weights are found.
Cited slides: 9, 10, 11, 12, 13.

8. Illustrate the key principles and step-by-step functioning of the k-Means clustering algorithm.
k-Means clustering is a popular unsupervised learning algorithm used to partition a dataset into K distinct, non-overlapping clusters. The key principle behind k-Means is to group data points based on their similarity, measured by distance (typically Euclidean distance), such that points within the same cluster are closer to each other than to points in other clusters. The algorithm aims to find the optimal placement of K cluster centers, called centroids, that minimize the sum of squared distances between each data point and its closest centroid.
The step-by-step functioning of the k-Means algorithm is as follows:

1. **Initialization**: Choose the number of clusters, K. Initialize K centroids by randomly selecting K data points from the dataset or by some other method.

2. **Assignment Step**: For each data point in the dataset, calculate its distance to each of the K centroids. Assign each data point to the cluster whose centroid is the nearest. This creates K initial

clusters based on the proximity of data points to the randomly initialized centroids.

3. **Update Step**: For each of the K clusters, recalculate the centroid. The new centroid is the mean (average) of all data points that have been assigned to that cluster in the current iteration. This moves the centroids to the center of their currently assigned points.
4. **Repeat**: Repeat the Assignment Step and the Update Step iteratively. The data points are re-assigned to the closest new centroids, and the centroids are recalculated based on the new cluster memberships.
5. **Convergence**: The algorithm converges when the centroids no longer move significantly between iterations, or when a maximum number of iterations is reached. At this point, the clusters are considered stable.

The output of the k-Means algorithm is the set of K final centroids and the assignment of each data point to one of the K clusters.
Cited slides: 82, 83, 84, 85, 86, 87.

9. Illustrate the key principles and step-by-step functioning of the Gaussian Mixture Model.
The Gaussian Mixture Model (GMM) is a probabilistic model used primarily for unsupervised learning tasks such as clustering and density estimation. The key principle behind GMM is the assumption that the data points are generated from a combination (mixture) of a finite number of Gaussian probability distributions with unknown parameters. Instead of assigning each data point to a single cluster as in k-Means, GMM assumes that each data point has a certain probability of belonging to each of the K Gaussian components.
The probability density function of a Gaussian Mixture Model with K components is given by:

$$p(\mathbf{x}) = \sum_{k=1}^{K} \pi_k \mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$$

where π_k is the mixing coefficient (or prior probability) of the k-th component, $\mathcal{N}(\mathbf{x}|\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ is the probability density function of a multivariate Gaussian distribution with mean $\boldsymbol{\mu}_k$ and covariance matrix $\boldsymbol{\Sigma}_k$, and $\sum_{k=1}^{K} \pi_k = 1$. The goal of the algorithm is to estimate the parameters of each component ($\boldsymbol{\mu}_k$, $\boldsymbol{\Sigma}_k$) and the mixing coefficients (π_k).
The parameters of a GMM are typically estimated using the Expectation-Maximization (EM) algorithm, which is an iterative approach for finding maximum likelihood estimates when there are latent variables (the unobserved component memberships of each data point). The step-by-step functioning using the EM algorithm is as follows:

1. **Initialization**: Choose the number of Gaussian components, K, and initialize the parameters for each component ($\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$). Initialization can be done randomly or using methods like k-Means to get initial estimates for the means and covariances.
2. **E-step (Expectation)**: In this step, given the current parameters, calculate the posterior probability that each data point $\mathbf{x}_i$ was generated by each Gaussian component k. This posterior probability is often referred to as the "responsibility" of component k for data point $\mathbf{x}_i$, denoted as $\gamma(z_{ik})$. This step involves using the current parameter values to calculate these probabilities for all data points and all components.
3. **M-step (Maximization)**: In this step, the parameters of the GMM ($\pi_k, \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k$) are updated to maximize the expected log-likelihood calculated based on the responsibilities computed in the E-step. This involves using formulas derived to update the mean ($\boldsymbol{\mu}_k$ becomes the weighted mean of data points assigned to component k), covariance matrix ($\boldsymbol{\Sigma}_k$ becomes the weighted covariance based on assigned points), and mixing coefficients (π_k becomes the weighted proportion of points assigned to component k).

4. **Repeat**: The E-step and M-step are repeated iteratively. In each iteration, the responsibilities are recalculated based on the updated parameters (E-step), and then the parameters are updated based on the new responsibilities (M-step).

5. **Convergence**: The algorithm continues until the changes in the parameters or the log-likelihood between iterations fall below a certain threshold, indicating convergence.

After convergence, the GMM provides a probabilistic assignment of each data point to the learned components, reflecting the likelihood that the point belongs to each cluster.
Cited slides: 82, 88, 89, 90, 91, 92, 93, 94, 96, 98, 100, 102, 103.

10. Illustrate the key principles and step-by-step functioning of the Expectation-Maximization (EM) algorithm.
The Expectation-Maximization (EM) algorithm is a powerful iterative method used to find maximum likelihood estimates (MLE) of parameters in statistical models, particularly when the data is incomplete or when the model depends on unobserved latent variables. The key principle is to handle these missing or latent variables by iteratively performing two steps: an Expectation (E) step and a Maximization (M) step, which together are guaranteed to improve the parameter estimates and converge towards a local maximum of the likelihood function.
The step-by-step functioning of the EM algorithm is as follows:

1. **Initialization**: Initialize the model parameters with some starting values. These can be chosen randomly or using a simpler method.

2. **E-step (Expectation)**: In this step, given the current parameter estimates, compute the expected value of the complete data log-likelihood with respect to the unobserved latent variables. This involves calculating the probabilities of the latent variables, given the observed data and the current parameter values. For example, in the context of Gaussian Mixture Models, the E-step calculates the responsibility of each Gaussian component for generating each data point – essentially, the posterior probability that a data point belongs to each component given the current means, covariances, and mixing coefficients.

3. **M-step (Maximization)**: In this step, use the expected log-likelihood calculated in the E-step to find the new parameter estimates that maximize this expected likelihood. This is typically a standard maximum likelihood estimation problem, but now applied to the "complete" data where the latent variables have been assigned probabilistic values from the E-step. The parameters (e.g., means, covariances, and mixing coefficients for GMMs) are updated using formulas derived to maximize the expected log-likelihood.

4. **Repeat**: Repeat the E-step and the M-step iteratively. The estimates from the M-step of the current iteration become the parameters used in the E-step of the next iteration.

5. **Convergence**: The algorithm continues iterating until the parameter estimates no longer change significantly between steps, or the increase in the log-likelihood falls below a certain threshold, indicating that convergence has been reached.

The EM algorithm iteratively refines the parameter estimates by first estimating the values of the latent variables (E-step) and then using these estimated latent variable values to update the parameters (M-step), leading to improved model fit to the observed data.
Cited slides: 93, 94, 96, 98, 100, 102, 103.

4 k-NN, Perceptron, Kernels, SVM, MLP

1. Illustrate the key principles and step-by-step functioning of the k-Nearest Neighbors classification algorithm.

The k-Nearest Neighbors (KNN) is a simple yet effective supervised learning algorithm used for classification tasks. A key principle of KNN is that it is an instance-based or lazy learning algorithm, meaning it does not explicitly learn a model during a training phase. Instead, it memorizes the entire training dataset and performs computations only when a prediction is needed for a new, unseen data point. The core idea is that data points that are close to each other in the feature space are likely to belong to the same class. The principle for classifying a new data point is based on its proximity to the points in the training set. Specifically, to classify a test sample, the algorithm finds the K training samples that are nearest to it. The class assignment is then determined by a majority vote among these K nearest neighbors.
The step-by-step functioning of the k-Nearest Neighbors classification algorithm to classify a new test sample is as follows:

1. **Choose K**: Select the number of neighbors, K, which is a positive integer (e.g., 1, 3, 5).
2. **Choose a Distance Metric**: Select a distance metric to measure the similarity or dissimilarity between data points. The most common choice is the Euclidean distance, but other metrics like Manhattan distance can also be used.
3. **Calculate Distances**: For the given test sample, calculate its distance to every single data point in the entire training dataset using the chosen distance metric.
4. **Find K Nearest Neighbors**: Identify the K training data points that have the smallest distances to the test sample.
5. **Majority Vote**: Determine the class labels of these K nearest neighbors. The test sample is assigned the class label that appears most frequently among these K neighbors (a majority vote).

For example, if K=3 and the 3 nearest neighbors have labels Class A, Class A, and Class B, the test sample will be classified as Class A. If K=1, it simply assigns the class of the single nearest neighbor.
Cited slides: 6, 7, 9, 10, 11, 12.

2. Describe the main advantages and disadvantages of the k-Nearest Neighbors (k-NN) algorithm. Explain why it is considered a non-parametric method and discuss its sensitivity to the choice of 'k' and feature scaling
The k-Nearest Neighbors (k-NN) algorithm has several advantages and disadvantages:

Main Advantages:

- **Simplicity**: KNN is relatively easy to understand and implement.
- **No Training Phase**: It is a lazy learning algorithm, meaning it doesn't explicitly train a model offline. The "training" effectively consists of just storing the training dataset.
- **Non-parametric**: It makes no assumptions about the underlying data distribution.
- **Effectiveness for Complex Decision Boundaries**: KNN can learn arbitrarily complex decision boundaries.

Main Disadvantages:

- **Computationally Expensive**: Classifying a new sample requires calculating distances to all training samples, which can be slow for large datasets or high-dimensional data (Curse of Dimensionality).
- **Sensitive to the Choice of K**: The performance is highly dependent on the value of K.
- **Sensitive to the Distance Metric**: The choice of distance metric (e.g., Euclidean, Manhattan) impacts results.
- **Sensitive to Feature Scaling**: Features with larger scales can disproportionately influence the distance calculations.

- **Sensitive to Noise and Outliers**: Outliers can significantly affect the classification of nearby points, especially with small K.

KNN is considered a **non-parametric method** because it does not assume a fixed functional form for the mapping from inputs to outputs, nor does it learn a fixed set of parameters that summarizes the data independent of the number of training examples. Instead, the entire training dataset serves as the model itself. The complexity of the model grows with the size of the training data, and it does not constrain the decision boundary to a predefined shape.

KNN's performance is sensitive to several factors:

- **Sensitivity to the Choice of K**: The value of K is a hyperparameter. A small value of K (e.g., K=1) makes the model highly sensitive to noise in the individual training points and can result in a very complex, jagged decision boundary (high variance, low bias). A large value of K provides smoother decision boundaries and is less sensitive to noise but might lead to misclassifying points that are close to a boundary if a majority of neighbors belong to the other class (high bias, low variance). Choosing an optimal K is crucial and often done via cross-validation.
- **Sensitivity to Feature Scaling**: Since KNN classifies based on the distance between data points, the scale of different features significantly impacts the distance calculation. Features with larger numerical ranges will dominate the distance computation, effectively rendering features with smaller ranges less important, regardless of their actual relevance. Therefore, **Feature Scaling** (like standardization or normalization) is a critical preprocessing step for KNN to ensure all features contribute relatively equally to the distance calculations.

Cited slides: 13, 14, 15, 16.

3. Explain the Perceptron learning algorithm for binary classification. Describe its main components and the process of weights updating.

The Perceptron is one of the earliest and simplest supervised learning algorithms designed for binary classification. Its key principle is to learn a linear decision boundary that separates the data points belonging to two different classes. It assumes that the data is linearly separable.

The main components of the Perceptron model are:

- **Input Features** ($\mathbf{x}$): A vector of features for a given data instance.
- **Weights** ($\mathbf{w}$) and **Bias** (b): Parameters that the algorithm learns from the data. The weights determine the slope and orientation of the decision boundary, while the bias shifts the boundary.
- **Weighted Sum**: A linear combination of the input features and their corresponding weights, plus the bias: $\mathbf{w}^T\mathbf{x} + b$.
- **Activation Function**: A step function, typically the sign function ($sign$), which outputs $+1$ if the weighted sum is non-negative and -1 (or 0) otherwise.
- **Predicted Output** ($\hat{y}$): The binary class label predicted by the model, $\hat{y} = sign(\mathbf{w}^T\mathbf{x} + b)$.

The Perceptron learning algorithm is an iterative process that updates the weights and bias whenever a misclassification occurs on a training example. The process of weights updating is as follows:

1. **Initialization**: Start with initial weights and bias values, typically set to zero or small random numbers.
2. **Iterate through Training Data**: The algorithm processes the training examples one by one, or in batches.
3. **Make Prediction**: For a given training example $(\mathbf{x}_i, y_i)$, calculate the predicted output $\hat{y}_i = sign(\mathbf{w}^T\mathbf{x}_i + b)$ using the current weights and bias.

4. **Check for Error**: Compare the predicted output $\hat{y}_i$ with the true label y_i.

5. **Update Weights and Bias**: If the prediction is incorrect ($\hat{y}_i \neq y_i$), update the weights and bias according to the Perceptron update rule. The update is calculated as:

   - Change in weights: $\Delta \mathbf{w} = \eta(y_i - \hat{y}_i)\mathbf{x}_i$
   - Change in bias: $\Delta b = \eta(y_i - \hat{y}_i)$

   The new weights and bias become $\mathbf{w} \leftarrow \mathbf{w} + \Delta \mathbf{w}$ and $b \leftarrow b + \Delta b$, where η is the learning rate, a small positive value that controls the step size of the updates. This update moves the decision boundary in a way that reduces the error for the misclassified point.

6. **No Update for Correct Predictions**: If the prediction is correct ($\hat{y}_i = y_i$), the weights and bias are not changed.

7. **Repeat**: The process of checking for errors and updating parameters is repeated for all training examples, and this cycle is typically repeated over multiple passes (epochs) through the training data.

The algorithm is guaranteed to converge to a solution (a set of weights and bias) that perfectly separates the classes if the data is linearly separable. If the data is not linearly separable, the algorithm will not converge.
Cited slides: 20, 21, 22, 23, 24, 25, 26.

4. What is the "Kernel Trick" in machine learning? Explain its purpose, particularly how it allows linear algorithms (like the Perceptron or SVM) to learn non-linear decision boundaries.
The "Kernel Trick" is a powerful technique in machine learning that allows algorithms that are formulated in terms of dot products to operate effectively in a high-dimensional feature space without ever explicitly computing the coordinates of the data points in that space. This is particularly useful when dealing with data that is not linearly separable in its original input space.
The purpose of the Kernel Trick is twofold:

- **Computational Efficiency**: Explicitly mapping data to a very high-dimensional, or even infinite-dimensional, feature space can be computationally very expensive or even impossible. The Kernel Trick avoids this by providing a function, called a kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$, that directly computes the dot product between the feature representations of two data points in the high-dimensional space, i.e., $K(\mathbf{x}_i, \mathbf{x}_j) = \phi(\mathbf{x}_i) \cdot \phi(\mathbf{x}_j)$, where ϕ is the mapping function to the higher-dimensional space. This computation $K(\mathbf{x}_i, \mathbf{x}_j)$ is often much cheaper to compute than explicitly calculating $\phi(\mathbf{x}_i)$ and $\phi(\mathbf{x}_j)$ and then their dot product.

- **Enabling Non-linear Boundaries**: The key to learning non-linear decision boundaries with linear algorithms lies in the idea of mapping the data from the original input space into a higher-dimensional feature space where the data might become linearly separable. A linear algorithm, like the Perceptron or Support Vector Machine (SVM), can then find a linear decision boundary in this higher-dimensional space. This linear boundary in the feature space corresponds to a non-linear decision boundary when mapped back to the original input space.

The Kernel Trick allows linear algorithms, specifically those whose learning and prediction can be expressed using only dot products between data points (such as the dual formulation of SVM or a kernelized Perceptron), to implicitly operate in this higher-dimensional space. By replacing the standard dot product $\mathbf{x}_i \cdot \mathbf{x}_j$ with the kernel function $K(\mathbf{x}_i, \mathbf{x}_j)$, the algorithm effectively finds a linear separator in the space defined by ϕ, without ever needing to know what ϕ explicitly is or what the coordinates $\phi(\mathbf{x}_i)$ are. This enables linear algorithms to discover and represent complex, non-linear decision boundaries in the original data space efficiently.
Cited slides: 41, 42, 43, 44, 45, 132, 133.

5. Describe the basic architecture of a Multi-Layer Perceptron (MLP). Explain the role of hidden layers and non-linear activation functions in enabling MLPs to model complex relationships, overcoming the limitations of the single-layer Perceptron.

A Multi-Layer Perceptron (MLP), also known as a feedforward neural network, is a type of artificial neural network consisting of multiple layers of interconnected nodes, or neurons. The basic architecture of an MLP is organized into layers:

- **Input Layer**: This is the first layer that receives the raw input data. The number of neurons in this layer corresponds to the number of features in the input data.
- **Hidden Layers**: These are one or more layers positioned between the input and output layers. Data flows through these layers, and they perform computations on the input received from the previous layer. Unlike the input and output layers, the hidden layers do not directly interact with the external environment.
- **Output Layer**: This is the final layer that produces the network's output or prediction. The number of neurons in this layer depends on the specific task (e.g., one neuron for binary regression, multiple neurons for classification).

Connections in an MLP are typically feedforward, meaning information flows in only one direction, from the input layer through the hidden layers to the output layer, without loops or cycles. Each connection between neurons has an associated weight, and each neuron in a hidden or output layer has a bias.

The power of MLPs to model complex, non-linear relationships, overcoming the limitations of the single-layer Perceptron (which can only learn linear decision boundaries), comes from the combination of **hidden layers** and **non-linear activation functions**:

- **Role of Hidden Layers**: Hidden layers enable the network to learn increasingly abstract and complex representations of the input data. Each hidden layer can be thought of as learning a new set of features from the output of the previous layer. By stacking multiple hidden layers, the network can learn hierarchical representations, extracting intricate patterns and relationships that are not linearly separable in the original input space. They allow the network to perform complex transformations on the data.
- **Role of Non-linear Activation Functions**: A non-linear activation function is applied to the output of each neuron in the hidden (and sometimes output) layers after the weighted sum of inputs and bias is calculated. These functions introduce non-linearity into the network's computations. This non-linearity is crucial because, without it (i.e., if only linear activation functions were used), an MLP with any number of hidden layers would effectively behave like a single-layer model, only capable of learning linear transformations and linear decision boundaries. Non-linear activation functions allow the MLP to approximate any continuous function and model complex, non-linear relationships in the data.

Together, hidden layers provide the depth and capacity to learn hierarchical features, while non-linear activation functions provide the necessary non-linearity to break the limitation of learning only linear separations, enabling MLPs to capture complex patterns that are not linearly separable.
Cited slides: 50, 51, 52, 53, 54, 55, 56, 57, 58.

6. Explain the core idea behind the Support Vector Machine (SVM) for linearly separable data (Hard Margin SVM). Describe what the "margin" is and why SVM aims to maximize it.

The core idea behind the Support Vector Machine (SVM) for linearly separable data, often referred to as Hard Margin SVM, is to find the optimal linear decision boundary (a hyperplane) that separates the data points belonging to two different classes with the largest possible margin. Unlike other linear classifiers like the Perceptron which might find any separating hyperplane, the Hard Margin SVM seeks the specific

hyperplane that achieves the maximum separation between the closest points of the classes. This is applicable only when the data is perfectly linearly separable.
The **margin** is defined as the distance between the separating hyperplane and the closest data point from either class. These closest data points are called the support vectors; they are the points that lie on the boundaries of the margin. The margin can be visualized as the region around the separating hyperplane that contains no data points. The boundaries of this region are defined by two hyperplanes parallel to the separating one, each passing through the support vectors of one class.
SVM aims to **maximize this margin** because a larger margin corresponds to a more robust and generalized classifier. Intuitively, a hyperplane with a larger margin provides a greater "safety buffer" between the decision boundary and the data points. This makes the classifier less susceptible to noise or small variations in the data and is expected to generalize better to new, unseen data. Maximizing the margin is equivalent to finding the hyperplane that is as far as possible from the closest training examples of both classes, providing a more stable and reliable decision boundary. The Hard Margin SVM strictly enforces that all training data points must lie on the correct side of the margin boundaries.
Cited slides: 117, 119, 120, 121, 122, 128, 130.

7. Explain why the distance measure choice is crucial for the kNN algorithm. Make at least 3 examples of metrics
For the k-Nearest Neighbors (kNN) algorithm, the choice of the distance measure is crucial because the algorithm's core functioning relies entirely on finding the "nearest" data points in the training set to a given test sample. The distance metric fundamentally defines what it means for two samples to be "close" or "similar" in the feature space. Different distance metrics can lead to different sets of nearest neighbors for the same test point, which in turn can result in different outcomes in the majority vote for classification. Therefore, selecting an appropriate distance metric that reflects the underlying structure and characteristics of the data is vital for the performance of the kNN algorithm.
Common examples of distance metrics used in kNN include:

- **Euclidean Distance**: This is the most common distance metric and represents the straight-line distance between two points in Euclidean space. For two points $p = (p_1, p_2, \dots, p_n)$ and $q = (q_1, q_2, \dots, q_n)$, the Euclidean distance is calculated as:

$$d(p,q) = \sqrt{\sum_{i=1}^{n}(q_i - p_i)^2}$$

- **Manhattan Distance (L1 Norm)**: Also known as city block distance or L1 norm, this metric calculates the sum of the absolute differences between the coordinates of the two points. It represents the distance a car would travel between two points in a city grid.

$$d(p,q) = \sum_{i=1}^{n} |q_i - p_i|$$

- **Minkowski Distance**: This is a generalization of both Euclidean and Manhattan distances. For a parameter p, the Minkowski distance is:

$$d(p,q) = \left(\sum_{i=1}^{n} |q_i - p_i|^p\right)^{1/p}$$

When $p = 1$, it is the Manhattan distance, and when $p = 2$, it is the Euclidean distance.

- **Cosine Similarity/Distance**: While not strictly a distance in the Euclidean sense, Cosine Similarity measures the cosine of the angle between two vectors, indicating their orientation similarity regardless of their magnitude. Cosine Distance is often defined as $1 -$ Cosine Similarity. This is particularly useful for high-dimensional data like text documents, where the magnitude might be less important than the direction (topic). The Cosine Similarity between two vectors $\mathbf{A}$ and $\mathbf{B}$ is:

$$S_C(\mathbf{A}, \mathbf{B}) = \frac{\mathbf{A} \cdot \mathbf{B}}{||\mathbf{A}|| \cdot ||\mathbf{B}||}$$

The choice among these metrics should be guided by the nature of the features and how "similarity" or "closeness" is best defined for the specific problem.
Cited slides: 11, 16.

5 Neural Networks, Backpropagarion & Optimization

1. What is an Artificial Neural Network?
An Artificial Neural Network (ANN), often referred to simply as a neural network, is a computational model inspired by the structure and function of biological neural networks, such as the human brain. It is built as a network of interconnected processing nodes, called neurons, organized in layers.
These networks are designed to learn complex patterns and relationships within data, particularly highly non-linear functions. Key components include:

- **Neurons (Nodes)**: The basic processing units that receive input, perform a computation, and produce an output.
- **Connections**: Links between neurons that transmit signals. Each connection has an associated numerical weight, which represents the strength and importance of the connection.
- **Layers**: Neurons are typically organized into layers, including an input layer, one or more hidden layers, and an output layer. Information flows from the input layer through the hidden layers to the output layer.
- **Activation Functions**: A function applied within each neuron (or layer) that introduces non-linearity to the network's output, enabling it to learn complex, non-linear patterns.

Through learning algorithms (like backpropagation and gradient descent), ANNs adjust the weights and biases of connections to minimize errors and make accurate predictions or decisions based on the input data.
Cited slides: 4, 5, 6, 7, 8, 9, 10.

2. Describe the Gradient Descent algorithm. How is it usually employed in Machine Learning?
Gradient Descent is a first-order iterative optimization algorithm used to find the local minimum of a differentiable function. The core principle of Gradient Descent is to repeatedly take steps in the direction opposite to the gradient of the function at the current point. The gradient points towards the direction of the steepest increase of the function, so moving in the negative gradient direction ensures that we move towards a lower value of the function.
The update rule for the parameters $\mathbf{w}$ in Gradient Descent is given by:

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla f(\mathbf{w})$$

where $f(\mathbf{w})$ is the function we want to minimize, $\nabla f(\mathbf{w})$ is the gradient of f with respect to $\mathbf{w}$, and η (eta) is the learning rate, a small positive scalar that determines the size of the steps taken in the direction of the negative gradient. The algorithm starts with initial parameter values and iteratively updates them using this rule, gradually moving towards a local minimum of the function.
In Machine Learning, Gradient Descent is the most commonly employed optimization algorithm. It is used to find the values for the model's parameters (such as weights and biases in neural networks) that minimize a chosen loss function. The loss function quantifies how poorly the model performs on the training data.

The goal is to find the parameters that result in the lowest possible loss.
The process involves:

1. Defining a loss function that measures the error of the model's predictions.
2. Calculating the gradient of the loss function with respect to the model's parameters using the training data. This gradient indicates how changing each parameter would affect the loss.
3. Updating the model's parameters by taking a step proportional to the negative of the gradient, scaled by the learning rate.
4. Repeating steps 2 and 3 iteratively over the training data until the loss function converges to a minimum or a stopping criterion is met.

While standard Batch Gradient Descent computes the gradient over the entire dataset in each step, variants like **Stochastic Gradient Descent (SGD)** and **Mini-batch Gradient Descent** are usually employed in practice, especially for large datasets. SGD computes the gradient and updates parameters using a single training example at a time, while Mini-batch Gradient Descent uses a small batch of training examples. These variants are computationally more efficient and can help escape shallow local minima, although their updates are noisier.
Cited slides: 64, 65, 66, 67, 68, 69, 70.

3. What is the difference between the Gradient Descent and the Stocastic Gradient Descent algorithms?
Both Gradient Descent (specifically Batch Gradient Descent) and Stochastic Gradient Descent (SGD) are iterative optimization algorithms used to minimize a loss function and find the optimal parameters of a model. The fundamental difference between them lies in the amount of data used to compute the gradient and perform a parameter update in each iteration.
Batch Gradient Descent: In standard Batch Gradient Descent, the gradient of the loss function is calculated with respect to the model's parameters using the **entire training dataset** in each iteration. This provides a precise gradient direction, ensuring that each step taken is towards the true minimum of the loss function over the entire dataset. However, computing the gradient over the entire dataset can be computationally very expensive and slow, especially when dealing with large datasets.
Stochastic Gradient Descent (SGD): In contrast, SGD computes the gradient and updates the model's parameters using only a **single training example** at a time. This results in much more frequent updates compared to Batch Gradient Descent. While the gradient computed for a single example is a much noisier estimate of the true gradient (the gradient over the entire dataset), the frequent updates can lead to faster convergence initially and potentially help escape shallow local minima.
In practice, a common compromise between Batch Gradient Descent and SGD is **Mini-batch Gradient Descent**. This variant computes the gradient and updates parameters using a small subset or "mini-batch" of the training data. This provides a less noisy estimate of the gradient than SGD (for a single example) while being significantly more computationally efficient than Batch Gradient Descent (using the entire dataset). Mini-batch Gradient Descent is the most widely used optimization approach in deep learning.
Therefore, the core distinction is the granularity of the data used for gradient calculation and parameter updates: the whole dataset for Batch GD, a single sample for SGD, and a small batch for Mini-batch GD.
Cited slides: 67, 70, 71, 73.

4. Describe the Stocastic Gradient Descent algorithm
Stochastic Gradient Descent (SGD) is an iterative optimization algorithm widely used in Machine Learning to minimize a loss function and find the optimal parameters of a model. While standard Batch Gradient Descent computes the gradient of the loss function using the entire training dataset before updating the parameters, SGD takes a different approach for efficiency.
The core characteristic of SGD is that, in each iteration, it computes the gradient of the loss function and updates the model's parameters using only a **single randomly selected training example** (or a very

small fixed number of examples, which is known as Mini-batch Gradient Descent and is the most common variant in practice).

The step-by-step process for SGD (using a single example for simplicity as described in the core idea) is:

1. Initialize the model parameters.
2. Iterate over the training data, typically for multiple epochs.
3. For each training example $(\mathbf{x}_i, y_i)$:
   (a) Calculate the loss for this single example.
   (b) Compute the gradient of the loss with respect to the model parameters using only this single example's data.
   (c) Update the model parameters by taking a step proportional to the negative of this gradient, scaled by the learning rate.

This process results in much more frequent parameter updates compared to Batch Gradient Descent, as an update occurs after processing each example (or each small batch). Although the gradient computed from a single example (or small batch) is a noisy estimate of the true gradient over the entire dataset, the frequent updates can lead to faster initial progress and can help the optimization process escape shallow local minima. Mini-batch SGD, using small batches (e.g., 32, 64, 128 samples), provides a balance between the noisy updates of pure SGD and the computational cost of Batch Gradient Descent and is the standard implementation in training deep learning models.

Cited slides: 70, 71, 73.

5. Describe the backpropagation algorithm

The backpropagation algorithm is the fundamental algorithm used to efficiently train artificial neural networks by computing the gradient of the loss function with respect to every weight and bias in the network. These gradients are essential for optimization algorithms like Gradient Descent to update the model's parameters and minimize the loss.

The core principle of backpropagation is the application of the chain rule from calculus. It allows for the systematic and efficient computation of how much each parameter contributes to the overall error (loss) of the network. Instead of recalculating gradients from scratch for each parameter, backpropagation propagates the error gradient backward through the network layers.

The functioning of the backpropagation algorithm involves two main passes:

1. **Forward Pass**:
   - The input data is fed into the network's input layer and propagates forward through the hidden layers to the output layer.
   - At each neuron and layer, computations (weighted sums and activation functions) are performed.
   - The network produces a final output prediction.
   - The loss function is calculated based on the difference between the predicted output and the true target value.
   - The values computed during the forward pass (inputs to neurons, outputs of activation functions, etc.) are stored, as they are needed for the backward pass.
2. **Backward Pass (Backpropagation)**:
   - The process starts at the output layer by calculating the gradient of the loss function with respect to the output of the network.

- This gradient is then propagated backward through the network, layer by layer, using the chain rule.
- For each layer, backpropagation computes two things: (a) the gradient of the loss with respect to the weights and biases of that layer (these are the gradients used for updating parameters), and (b) the gradient of the loss with respect to the inputs of that layer.
- The gradient with respect to the layer's inputs is then passed backward as the "upstream" gradient to the previous layer, where the process is repeated.
- This backward flow of gradients continues until the input layer is reached, resulting in the computation of the gradient of the loss with respect to all weights and biases throughout the network.

Once the gradients for all parameters are computed during the backward pass, an optimization algorithm (like SGD) uses these gradients to update the weights and biases, thereby minimizing the loss and improving the model's performance in the next iteration.
Cited slides: 74, 75, 76, 77, 80, 81, 82, 83, 84, 85, 86.

6. Describe the concepts of batches, epochs, and learning rate within the context of optimization algorithms for ML
Within the context of optimization algorithms commonly used in Machine Learning, such as Gradient Descent and its variants, the concepts of batches, epochs, and learning rate are crucial for the training process:
Batches: In optimization algorithms like Mini-batch Gradient Descent, the training dataset is divided into smaller subsets called batches. Instead of computing the gradient over the entire dataset (as in Batch Gradient Descent) or a single example (as in pure SGD), the algorithm computes the gradient and updates the model's parameters using the data within one batch at a time. Using batches provides a more stable estimate of the gradient than using a single example while being significantly more computationally efficient than using the entire dataset. The size of the batch is a hyperparameter.
Learning Rate: The learning rate, typically denoted by the Greek letter η, is a hyperparameter in gradient-based optimization algorithms. It determines the step size at which the model's parameters are adjusted during the update process. After computing the gradient of the loss function with respect to the parameters, the learning rate scales this gradient to determine how much the parameters change in the direction that reduces the loss. A high learning rate can cause the optimization to overshoot the minimum or oscillate, while a very low learning rate can result in extremely slow convergence. Choosing an appropriate learning rate is critical for effective training.
The concept of "epochs", which represents one complete pass through the entire training dataset during the optimization process, is fundamental in training iterations. However, an explicit definition or detailed description of "epochs" is not provided within the specific content of this PDF.
Cited slides: 67, 69, 70, 73.

7. Explain and describe the concept of Momentum in the context of optimization methods for learning models
Momentum is a technique used to accelerate optimization algorithms, particularly Gradient Descent, by adding a term that is proportional to the gradient of the previous steps. The core idea is inspired by physics: just as momentum in physics helps an object continue moving in its current direction, adding momentum to the parameter updates helps the optimization process continue moving in directions where gradients are consistent, while smoothing out oscillations in directions where gradients change frequently. This can lead to faster convergence and more stable training, especially in areas of the loss surface that are flat or have steep ravines.
The functioning of Gradient Descent with Momentum involves maintaining a velocity vector that accumulates a fraction of the past gradients. In each iteration of the optimization process:

1. **Calculate Gradient**: Compute the gradient of the loss function with respect to the model param-

eters ($\mathbf{g}_t = \nabla f(\mathbf{w}_t)$) for the current iteration t.

2. **Update Velocity**: Update a velocity vector ($\mathbf{v}_t$) based on the previous velocity ($\mathbf{v}_{t-1}$) and the current gradient ($\mathbf{g}_t$). The update rule for the velocity is typically:

$$\mathbf{v}_t = \alpha \mathbf{v}_{t-1} + \mathbf{g}_t$$

where α (alpha) is the momentum coefficient (a value between 0 and 1, e.g., 0.9) that determines how much of the previous velocity is retained.

3. **Update Parameters**: Update the model parameters ($\mathbf{w}_t$) by taking a step in the direction of the current velocity vector, scaled by the learning rate (η). The parameter update rule is:

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \eta \mathbf{v}_t$$

This process is repeated iteratively. The velocity term allows the optimization to build up speed in directions with consistent gradients and helps overcome local minima or saddle points. By dampening oscillations, it allows for potentially larger learning rates and faster progress towards the minimum of the loss function.
Cited slides: 88, 89, 90, 91, 92, 93, 106.

8. Describe the ADAM algorithm
ADAM (Adaptive Moment Estimation) is an advanced optimization algorithm widely used for training deep learning models. It is an adaptive learning rate method, meaning it computes individual learning rates for each parameter of the model based on estimates of the first and second moments of the gradients. The core idea of ADAM is to combine concepts from other optimization algorithms like Momentum and RMSprop to achieve faster and more stable convergence.
The functioning of the ADAM algorithm is iterative and involves maintaining two exponentially decaying moving averages for each parameter: one for the gradients (first moment) and one for the squared gradients (second moment). The step-by-step process in each iteration t (typically for a mini-batch) is as follows:

1. **Compute Gradient**: Calculate the gradient of the loss function with respect to the parameters for the current mini-batch, denoted as g_t.

2. **Update First Moment Estimate**: Update the biased first moment estimate (mean of gradients):

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

where β_1 is the exponential decay rate for the first moment (a hyperparameter, e.g., 0.9).

3. **Update Second Moment Estimate**: Update the biased second moment estimate (uncentered variance of gradients):

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

where β_2 is the exponential decay rate for the second moment (a hyperparameter, e.g., 0.999), and g_t^2 denotes element-wise squaring of the gradient vector.

4. **Bias Correction**: Compute bias-corrected first and second moment estimates to account for their initialization bias towards zero, especially in early iterations:

$$\hat{m}_t = m_t / (1 - \beta_1^t)$$
$$\hat{v}_t = v_t / (1 - \beta_2^t)$$

5. **Update Parameters**: Update the model parameters using the learning rate, the bias-corrected moment estimates, and a small epsilon for numerical stability:

$$\mathbf{w}_t \leftarrow \mathbf{w}_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

where η is the learning rate (a hyperparameter, e.g., 0.001) and ϵ is a small constant (a hyperparameter, e.g., 10^{-8}).

This process is repeated until convergence. ADAM adapts the learning rate for each parameter based on the historical average of gradients and squared gradients. The first moment ($\hat{m}_t$) acts similarly to momentum, while the second moment ($\hat{v}_t$) provides a per-parameter scaling of the learning rate, similar to RMSprop or Adagrad, which helps in handling sparse gradients and varying gradient scales across different parameters. These features generally make ADAM robust and performant across a wide range of problems and architectures.
Cited slides: 94, 95, 96, 97, 98, 99, 100, 101, 102, 106.

6 Logistic Regression & Calibration

1. Illustrate the key principles and step-by-step functioning of the Logistic Regression algorithm for binary classification.
Logistic Regression is a widely used supervised learning algorithm specifically designed for binary classification problems, where the goal is to predict one of two possible classes. Unlike Linear Regression which predicts a continuous value, Logistic Regression predicts the probability that a given input data point belongs to a particular class (usually the positive class).
The key principles of Logistic Regression are:

- It models the relationship between the input features and the probability of the output variable belonging to a specific class.
- It uses a linear combination of input features and learned weights ($\mathbf{w}^T\mathbf{x} + b$), similar to Linear Regression.
- It applies the Sigmoid function (also called the Logistic function) to this linear output. The Sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ maps any real-valued input into a value between 0 and 1, which is interpreted as a probability.
- The model predicts the probability of the positive class as $\hat{y} = P(y = 1|\mathbf{x}) = \sigma(\mathbf{w}^T\mathbf{x} + b)$.
- It learns a linear decision boundary in the feature space. The decision is made by comparing the predicted probability to a threshold (typically 0.5). If $\hat{y} \geq 0.5$, the instance is classified as the positive class (e.g., 1); otherwise, it's classified as the negative class (e.g., 0). Since $\sigma(z) \geq 0.5$ when $z \geq 0$, this is equivalent to classifying based on the sign of $\mathbf{w}^T\mathbf{x} + b$.

The step-by-step functioning of the Logistic Regression algorithm involves training to find the optimal weights $\mathbf{w}$ and bias b, and then using these parameters for prediction:
Training Process:

1. **Initialization**: Initialize the weights $\mathbf{w}$ and bias b (e.g., to zeros or small random values).
2. **Define Loss Function**: Use a suitable loss function, typically the Binary Cross-Entropy Loss (Log Loss), which is convex for Logistic Regression and measures the error between the predicted probabilities and the true binary labels.
3. **Optimize Parameters**: Minimize the loss function using an iterative optimization algorithm, such as Gradient Descent or its variants (like SGD or Adam). For each training instance (or batch):
   (a) Calculate the linear output: $z = \mathbf{w}^T\mathbf{x}_i + b$.
   (b) Calculate the predicted probability: $\hat{y}_i = \sigma(z)$.
   (c) Calculate the loss for this instance based on $\hat{y}_i$ and the true label y_i.
   (d) Compute the gradients of the loss with respect to $\mathbf{w}$ and b.
   (e) Update $\mathbf{w}$ and b using the gradients and the learning rate (e.g., $\mathbf{w} \leftarrow \mathbf{w} - \eta\nabla L(\mathbf{w})$).

4. **Repeat**: Repeat the optimization steps over the training data for multiple epochs until the parameters converge or the loss is minimized.

Prediction Process:

1. For a new input instance $\mathbf{x}_{new}$:
2. Calculate the linear output using the learned parameters: $z = \mathbf{w}^T\mathbf{x}_{new} + b$.
3. Apply the Sigmoid function to get the predicted probability: $\hat{y} = \sigma(z)$.
4. Classify the instance based on a threshold: if $\hat{y} \geq 0.5$, predict the positive class; otherwise, predict the negative class.

Cited slides: 4, 8, 9, 10, 11, 12, 13, 14, 15, 19, 20.

2. Some classification methods also output a score that can be interpreted as a probability of predicted class membership. Define the property of a classifier being 'well-calibrated' with respect to its probabilistic outputs. Explain why this is an important property in certain domains and name a simple application example.
When a classification method outputs a score that can be interpreted as the probability of a data instance belonging to a particular class, the property of being **well-calibrated** means that these predicted probabilities accurately reflect the true likelihood of the event occurring. Formally, a classifier is well-calibrated if, for any predicted probability value p, among all instances for which the classifier predicted a probability of p for a certain class, the actual proportion of instances belonging to that class is approximately equal to p. For example, if a classifier predicts a probability of 0.8 for a set of instances, then approximately 80% of those instances should actually belong to the predicted class.
This property is **important** in certain domains because in many real-world applications, merely knowing the predicted class label is not sufficient. Decision-making processes often rely heavily on the confidence or uncertainty associated with a prediction. Accurate probability estimates are crucial for:

- **Informed Decision Making**: Allows users to weigh the costs and benefits of different actions based on the likelihood of outcomes.
- **Risk Assessment**: Essential for quantifying uncertainty and managing risks in critical applications.
- **Combining Models**: Well-calibrated probabilities can be effectively combined from multiple models.
- **Ranking**: Probabilities provide a meaningful way to rank instances by their likelihood of belonging to a class.

Without proper calibration, a classifier might be highly accurate in terms of assigning the correct label, but its probability scores could be misleadingly high or low, leading to suboptimal decisions in practice.
A simple application example where calibration is important is **medical diagnosis**. Suppose a classifier predicts the probability of a patient having a certain disease. While a simple binary output might say "positive" or "negative", a calibrated probability score provides much more valuable information. A doctor would treat a predicted probability of 0.9 differently from a predicted probability of 0.55, even if both result in a "positive" label based on a 0.5 threshold. The probability estimate informs decisions about further testing, treatment aggressiveness, and communication with the patient regarding the level of risk and uncertainty. A well-calibrated model ensures that when it says there's an 80% chance of the disease, it means that among similar patients with the same score, about 80% truly have the disease. Logistic Regression is noted as being well-calibrated by construction.
Cited slides: 55, 56, 57, 58, 62.

3. A bank wants to predict whether a loan applicant is likely to default (Y=1) or not default (Y=0) on their loan. They have historical data including features like applicant income, credit score, loan amount, and employment duration, along with the actual outcome

(defaulted or not). Explain which method between Logistic Regression or Linear Regression would be most suitable for this task and why.

For the task described, where a bank wants to predict whether a loan applicant is likely to default (Y=1) or not default (Y=0), the most suitable method between Logistic Regression and Linear Regression is **Logistic Regression**.

The reason for choosing Logistic Regression is that this task is fundamentally a **binary classification problem**. The desired output is a categorical variable indicating one of two possible outcomes (default or not default), or the probability of belonging to one of these classes.

Logistic Regression is a classification algorithm specifically designed to predict the probability that a given input data point belongs to a certain class. It outputs a value between 0 and 1, which can be interpreted as a probability, and then typically a threshold is applied to classify the instance into one of two categories (0 or 1). This perfectly matches the requirement to predict the likelihood of default and ultimately the binary outcome.

In contrast, **Linear Regression** is a regression algorithm used for predicting a **continuous output variable**. It models a linear relationship between the input features and a numerical output. Since the outcome of the loan application (default or not default) is a discrete, binary variable, rather than a continuous numerical value, Linear Regression is not appropriate for this task. Using Linear Regression would provide a continuous score that is not naturally bounded between 0 and 1 and would require arbitrary thresholding without the probabilistic interpretation inherent in Logistic Regression.

Therefore, Logistic Regression is the suitable choice because it is designed for classification tasks and predicts probabilities, directly addressing the problem's nature.

Cited slides: 8, 9.

4. Explain the main problems or limitations of directly using standard Linear Regression for this binary classification task. How does Logistic Regression address these limitations, particularly concerning the range and interpretation of its output?

Directly using standard Linear Regression for a binary classification task, such as predicting loan default (0 or 1), presents several main problems and limitations:

- **Output Range Mismatch**: Linear Regression predicts a continuous output value that can range from negative infinity to positive infinity. This is fundamentally mismatched with a binary classification task, where the desired output is either one of two discrete class labels (0 or 1) or, ideally, a probability of belonging to a class, which must be between 0 and 1. There is no inherent mechanism in Linear Regression to constrain its output within the $[0, 1]$ range.

- **Lack of Probabilistic Interpretation**: While one could attempt to threshold the continuous output of Linear Regression to obtain binary classes (e.g., predict 1 if output > 0.5, else 0), the output does not naturally represent the probability of class membership. A score of 0.8 from a Linear Regression model does not mean the same thing as a probability of 0.8 in the context of likelihood. This lack of a meaningful probabilistic interpretation is a significant limitation for tasks where understanding the confidence or likelihood of a prediction is important, like assessing the risk of loan default.

- **Inappropriate Model Assumption and Loss Function**: Linear Regression assumes that the relationship between inputs and the continuous output is linear and often assumes that the errors are normally distributed. These assumptions are typically violated when the target variable is binary. The standard Mean Squared Error (MSE) loss function used in Linear Regression is not well-suited for optimizing parameters for a binary outcome; it does not penalize misclassifications in a way that aligns with probabilistic models, and when combined with a function like Sigmoid, it becomes non-convex, making optimization difficult.

Logistic Regression is specifically designed to address these limitations when tackling binary classification:

- **Constrained Output Range**: Logistic Regression uses the Sigmoid (or Logistic) function, which takes the linear combination of inputs and weights and squashes it into a value strictly between 0 and 1. This addresses the range mismatch problem, providing an output that is naturally scaled for binary probabilities.

- **Probabilistic Interpretation**: The output of the Sigmoid function in Logistic Regression is explicitly interpreted as the predicted probability of the instance belonging to the positive class ($P(y = 1|\mathbf{x})$). This provides a direct and meaningful probabilistic output, which is crucial for tasks requiring confidence assessment and risk evaluation.
- **Appropriate Loss Function**: Logistic Regression uses the Cross-Entropy Loss (Log Loss), which is specifically designed for models that output probabilities for classification. This loss function is convex for Logistic Regression when combined with the Sigmoid, facilitating effective optimization using gradient-based methods.

By incorporating the Sigmoid function and using the Cross-Entropy loss, Logistic Regression provides an output that is appropriately ranged, has a clear probabilistic interpretation, and uses a loss function suitable for training a model to predict binary outcomes, overcoming the fundamental limitations of applying Linear Regression to such tasks.
Cited slides: 5, 6, 7, 8, 9, 10, 11, 12, 14.

5. Illustrate the Expected Calibration Error metric
The Expected Calibration Error (ECE) is a metric used to quantify the calibration of a classifier that outputs probabilistic predictions. It measures the average difference between the predicted probabilities and the actual fraction of correct predictions, grouped into bins based on the predicted probability. A lower ECE value indicates better calibration, meaning the predicted probabilities are more reliable estimates of the true likelihoods.
The calculation of the Expected Calibration Error involves the following steps:

1. **Binning**: The range of predicted probabilities $[0, 1]$ is divided into M disjoint bins. For a given classifier and a set of predictions, each instance is assigned to a bin based on its predicted probability. Let B_m be the set of indices of instances whose predicted probability falls into bin m.
2. **Calculate Accuracy and Confidence per Bin**: For each bin m, two values are computed:
   - **Accuracy** ($acc(m)$): This is the fraction of instances within bin m for which the predicted class label (derived from the probability, typically by thresholding at 0.5) matches the true class label.
   $$acc(m) = \frac{1}{|B_m|} \sum_{i \in B_m} \mathbf{1}(\hat{y}_i = y_i)$$
   - **Confidence** ($conf(m)$): This is the average of the predicted probabilities for all instances within bin m.
   $$conf(m) = \frac{1}{|B_m|} \sum_{i \in B_m} \hat{p}_i$$
3. **Weighted Average of Differences**: The absolute difference between the accuracy and the confidence is calculated for each bin, $|acc(m) - conf(m)|$. This difference is then weighted by the proportion of instances in that bin relative to the total number of instances (N). The ECE is the sum of these weighted absolute differences across all M bins:
$$ECE = \sum_{m=1}^{M} \frac{|B_m|}{N} |acc(m) - conf(m)|$$

The ECE essentially provides a single numerical value summarizing how well the classifier's confidence aligns with its accuracy across different ranges of predicted probabilities. An ideally calibrated classifier would have $acc(m) \approx conf(m)$ for all bins, resulting in an ECE close to zero.
Cited slides: 55, 56, 57, 60.

6. Name and briefly describe one post-training classifier calibration method seen in class
Sometimes, even if a classifier is designed to output probabilities, its outputs may not be well-calibrated after training. In such cases, post-training calibration methods can be applied. One such method seen in the course material is **Platt Scaling**.
Platt Scaling is a technique used to calibrate the probabilistic outputs of a classification model after it has been trained. It involves fitting a separate Logistic Regression model to the outputs (scores or probabilities) of the already-trained base classifier. This Logistic Regression model is trained on a dedicated **calibration set**, which is a subset of data that was not used during the initial training of the base classifier. The Logistic Regression model learns a mapping function that transforms the base classifier's raw output scores into well-calibrated probabilities. This method is noted as being effective on classifiers like Support Vector Machines (SVMs) or Naive Bayes, which may not produce intrinsically well-calibrated probabilities.
Cited slides: 61, 62.

7 Introduction to Deep Learning

1. What is a convolutional neural network? Describe the overall architecture and its main components, with a particular focus on convolutional layers
A Convolutional Neural Network (CNN) is a type of deep neural network primarily designed for processing data with a grid-like topology, such as images (2D grid of pixels). They are highly effective in tasks like image classification, object detection, and image segmentation because they are able to automatically and adaptively learn spatial hierarchies of features.
The overall architecture of a typical CNN consists of several types of layers stacked sequentially:

- **Input Layer**: Holds the raw pixel values of the image or input volume.
- **Convolutional Layer(s)**: The core building block where filters are applied to the input.
- **Pooling Layer(s)**: Periodically reduce the spatial dimensions of the feature maps.
- **Fully-Connected Layer(s)**: Standard neural network layers used towards the end for classification or regression based on the features extracted by previous layers.
- **Output Layer**: Produces the final classification scores or predicted values.

The **Convolutional Layer** is the central and most distinctive component of a CNN. Its key principles and functioning are:

- **Local Connectivity and Parameter Sharing**: Neurons in a convolutional layer are only connected to a small local region of the input volume (the receptive field defined by the filter size). More importantly, the same set of weights (the filter or kernel) is shared across all connections in that local region as the filter slides across the input. This dramatically reduces the number of parameters compared to a fully-connected layer and makes the network translation-invariant to the detection of local features.
- **Filtering Operation**: The primary operation is convolution. A filter (a small matrix of weights) slides across the spatial dimensions (width and height) of the input volume. At each position, a dot product is computed between the filter's weights and the corresponding patch of the input covered by the filter.
- **Activation Maps**: Each filter produces a 2D activation map (or feature map) which shows where the filter's pattern is detected in the input. A convolutional layer typically has multiple filters, each learning to detect a different feature, resulting in a stack of activation maps forming the output volume of the layer.
- **Learned Parameters**: The learnable parameters in a convolutional layer are the values within the filters and the biases associated with each filter. These parameters are learned during the training

process through backpropagation to detect features relevant to the task.

- **Stride and Padding**: The sliding behavior of the filter is controlled by the stride (the step size). Padding (adding zeros around the input border) can be used to control the spatial size of the output volume. The output dimensions are determined by the input size, filter size, stride, and padding.

Convolutional layers learn to detect low-level features like edges and corners in early layers, and as the data passes through deeper convolutional layers, these features are combined to detect more complex patterns and objects.
Cited slides: lecture-17-03-25-IntroDeepLearning-CNN.pptx (1).pdf: 48, 49, 50, 51, 52, 53, 54, 55, 56, 58, 62, 63, 65, 66.

2. What is a convolutional neural network? Describe the overall architecture and its main components, with a particular focus on pooling layers
A Convolutional Neural Network (CNN) is a class of deep neural networks that excels at processing data with a grid-like structure, particularly images. They are designed to automatically learn hierarchical representations of spatial features from the input data.
The typical overall architecture of a CNN comprises several layers:

- **Input Layer**: Receives the initial data, such as an image.
- **Convolutional Layer(s)**: Apply filters to the input to detect local features.
- **Pooling Layer(s)**: Reduce the spatial size of the representations.
- **Fully-Connected Layer(s)**: Standard neural network layers that process the extracted features for the final task.
- **Output Layer**: Provides the final prediction.

The **Pooling Layer** is a common component used periodically between convolutional layers in a CNN. Its primary roles and functioning are:

- **Dimensionality Reduction**: The main purpose is to progressively reduce the spatial dimensions (width and height) of the input volume (the stack of feature maps produced by the preceding convolutional layer).
- **Parameter and Computation Reduction**: By reducing the spatial size, pooling layers decrease the number of parameters and computational cost in subsequent layers.
- **Control Overfitting**: Pooling helps to control overfitting by reducing the number of learnable parameters and making the network more robust to small variations in the training data.
- **Translational Invariance**: Pooling provides a degree of translational invariance, meaning that small shifts in the location of a feature in the input will result in only minor changes in the pooled output. This helps the network recognize features regardless of their exact position.
- **Pooling Operation**: The operation works by sliding a small rectangular window (the pooling window) over the input feature map, similar to convolution, defined by a size and stride. However, instead of applying a learned filter, it performs a fixed summary statistic within that window. Common summary operations include:
  - **Max Pooling**: Selects the maximum value within the pooling window. This is the most frequently used type of pooling.
  - **Average Pooling**: Calculates the average value within the pooling window.

  The pooling operation is applied independently to each depth slice (each feature map) of the input volume. The depth dimension remains unchanged; only the spatial dimensions are reduced.

Pooling layers help to make the feature representations more compact and robust, allowing the CNN to learn more abstract and invariant features in deeper layers.
Cited slides: lecture-17-03-25-IntroDeepLearning-CNN.pptx (1).pdf: 48, 49, 50, 66, 67, 68, 69, 70, 71.

3. What is a convolutional neural network? Describe the overall architecture and its main components, with a particular focus on activation functions.
A Convolutional Neural Network (CNN) is a type of deep neural network widely used for processing data with a grid-like structure, such as images. They are characterized by their ability to automatically learn hierarchical features through specialized layers.
The general architecture of a CNN includes:

- **Input Layer**: Where the raw data enters the network.
- **Convolutional Layer(s)**: Apply filters to the input.
- **Activation Function(s)**: Applied after convolutional (and sometimes other) layers.
- **Pooling Layer(s)**: Reduce spatial dimensions.
- **Fully-Connected Layer(s)**: Standard layers for final processing.
- **Output Layer**: Produces the final result.

Activation functions are a crucial component within CNNs, typically applied after the linear operation (like convolution) within a layer and before the data is passed to the next layer. Their main role and importance are:

- **Introduce Non-linearity**: This is the primary purpose. Activation functions apply a non-linear transformation to the output of the linear operation (weighted sum and bias). Without non-linear activation functions, a CNN, regardless of how many layers it has, would effectively behave as a single linear model. Stacking multiple linear layers results only in another linear transformation.
- **Enable Learning Complex Relationships**: By introducing non-linearity, activation functions allow the CNN to learn and model complex, non-linear relationships and patterns within the data. This is essential for tackling intricate tasks like image recognition, where decision boundaries are rarely linear.
- **Biological Inspiration**: They are loosely inspired by the firing of neurons in biological brains, introducing a threshold or non-linear response to input signals.

Common examples of non-linear activation functions used in CNNs include the Rectified Linear Unit (ReLU), which is currently the most popular choice, as well as others like Sigmoid and Tanh. Applied element-wise to the output of a layer, activation functions ensure that the network can approximate any complex function, giving deep learning models their power.
Cited slides: lecture-17-03-25-IntroDeepLearning-CNN.pptx (1).pdf: 48, 49, 50, 72, 73, 74, 75, 76, 77.

4. What is a convolutional neural network? Describe the overall architecture and its main components, describe the training procedures and strategies with a particular focus on data preprocessing and weight initialization
A Convolutional Neural Network (CNN) is a type of deep neural network specifically designed for processing grid-like data, such as images. CNNs are structured to automatically learn hierarchical features by applying learned filters to local regions of the input.
The overall architecture of a CNN typically consists of an Input Layer followed by a stack of interconnected layers including Convolutional Layers, Activation Functions (applied after convolutions), Pooling Layers, and finally one or more Fully-Connected Layers leading to an Output Layer.
The main components, already discussed, include Convolutional Layers for feature extraction, Pooling Layers for spatial dimension reduction, and Activation Functions for introducing non-linearity. Fully-Connected layers are used for the final classification or regression based on the learned features.

Training procedures and strategies for CNNs, as with other neural networks, involve finding the optimal values for the model's parameters (weights and biases) to minimize a loss function (e.g., Cross-Entropy for classification) using optimization algorithms like Stochastic Gradient Descent or Adam. This process relies on computing gradients using the backpropagation algorithm. Key strategies within this training procedure include:

Data Preprocessing: This is a crucial step applied to the input data before feeding it into the network. Its purpose is to improve the stability and speed of training, and potentially enhance the model's performance. Common techniques include:

- **Zero-centering**: Subtracting the mean value from each feature across the dataset so that the data is centered around zero. This helps prevent all gradients from having the same sign during the early stages of training, which can lead to a zig-zagging optimization path.
- **Normalization**: Scaling the data so that each feature has a similar range or variance, often unit variance. This ensures that no single feature dominates the gradient calculation due to its scale.

Preprocessing helps in making the optimization landscape more amenable to gradient-based methods.

Weight Initialization: The initial values assigned to the weights and biases of the network before training begins significantly impact the training process and the final performance. Poor initialization can lead to problems such as vanishing or exploding gradients during backpropagation, hindering learning.

- **Problems with poor initialization**: Initializing all weights to zero is problematic because all neurons in a layer would compute the same output and gradients, learning identical features. Initializing weights too large can lead to exploding gradients, where gradients become excessively large, causing unstable training. Initializing weights too small can lead to vanishing gradients, where gradients become tiny as they propagate backward, preventing earlier layers from learning effectively.
- **Common Strategies**: Heuristics are used to initialize weights such that the variance of activations and gradients is maintained across layers. Methods like initializing with small random numbers drawn from a Gaussian or uniform distribution are basic approaches. More advanced methods include:
  - **Xavier/Glorot Initialization**: Scales initial weights based on the number of input and output connections of a neuron to maintain activation variance, suitable for activation functions like Sigmoid or Tanh.
  - **He Initialization**: Similar to Xavier but specifically designed to work well with ReLU activation functions, taking into account the number of input connections.

Effective data preprocessing and appropriate weight initialization are essential strategies that contribute significantly to successful training and good performance of CNNs and other deep neural networks by providing a stable foundation for the optimization process.

Cited slides: lecture-17-03-25-IntroDeepLearning-CNN.pptx (1).pdf: 49, 50; lecture - 20-03-25-IntroDLContd.ppt
3, 4, 5, 6, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31.

5. What is a convolutional neural network? Describe the overall architecture and its main components, describe the training procedures and strategies with a particular focus on batch normalization

A Convolutional Neural Network (CNN) is a specialized type of deep neural network designed for processing structured grid data like images. Its architecture is built to automatically learn features at different levels of abstraction.

The typical overall architecture includes layers such as the Input Layer, Convolutional Layers for feature extraction, Activation Functions to introduce non-linearity, Pooling Layers for spatial downsampling, and Fully-Connected Layers followed by an Output Layer for final prediction.

Training a CNN involves learning the weights and biases of these layers to minimize a chosen loss function using iterative optimization algorithms like Gradient Descent variants (e.g., SGD, Adam), which require computing gradients via backpropagation. Among the crucial strategies employed during training, **Batch**

Normalization is particularly significant.

Batch Normalization (BatchNorm) is a technique used to address the problem of "Internal Covariate Shift," which refers to the change in the distribution of inputs to the internal layers of the network during training, as the parameters of the preceding layers are updated. This shift makes training more difficult and slower, potentially requiring lower learning rates and careful initialization.

Batch Normalization works by normalizing the inputs to a layer for each mini-batch during training. For a given batch of activations $\mathcal{B} = \{x_1, \ldots, x_m\}$, the operation is as follows:

1. Calculate the batch mean: $\mu_{\mathcal{B}} = \frac{1}{m}\sum_{i=1}^{m} x_i$.
2. Calculate the batch variance: $\sigma_{\mathcal{B}}^2 = \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2$.
3. Normalize the activations: $\hat{x}_i = \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}}$, where ϵ is a small constant for numerical stability.
4. Scale and Shift: Apply learned parameters γ and β to the normalized activations: $y_i = \gamma\hat{x}_i + \beta$. γ scales the normalized values, and β shifts them. These parameters are learned during training and allow the network to recover the original distribution if optimal.

Batch Normalization layers are typically inserted after the linear transformation (e.g., after a convolutional or fully-connected operation) and before the non-linear activation function.

The benefits of using Batch Normalization during training are substantial:

- **Accelerated Convergence**: It allows the network to train much faster.
- **Higher Learning Rates**: It enables the use of significantly higher learning rates, speeding up the optimization process.
- **Less Critical Initialization**: It makes the network less sensitive to the initial values of the weights.
- **Regularization Effect**: It can act as a mild regularizer, sometimes reducing the need for techniques like Dropout.
- **Improved Performance and Stability**: Generally leads to better final performance and more stable training.

By stabilizing the distribution of inputs to layers, Batch Normalization significantly improves the training efficiency and effectiveness of deep neural networks, including CNNs.

Cited slides: lecture-17-03-25-IntroDeepLearning-CNN.pptx.pdf: 49, 50; lecture - 20-03-25-IntroDLContd.pptx.p
32, 33, 34, 35, 36, 37.

6. What is a convolutional neural network? Describe the overall architecture and its main components, describe the training procedures and strategies with a particular focus on transfer learning

A Convolutional Neural Network (CNN) is a class of deep neural networks primarily used for processing data with a grid structure, notably images. Their architecture is designed to automatically learn hierarchical features through layers that apply learned filters.

The typical overall architecture of a CNN includes an Input Layer, Convolutional Layers for feature extraction, Activation Functions for non-linearity, Pooling Layers for spatial dimension reduction, and Fully-Connected Layers leading to the Output Layer for the final prediction.

Training a CNN involves optimizing its parameters (weights and biases) to minimize a loss function, typically using iterative algorithms like gradient descent variants combined with backpropagation to compute gradients. A powerful strategy within this training context is **Transfer Learning**.

Transfer Learning is a training strategy where a model that has been pre-trained on a large dataset for a related task is used as a starting point for training a new model on a different, often smaller, dataset or task. The core idea is to leverage the features and hierarchical representations learned by the pre-trained model, as these are often generalizable to new domains.

Transfer learning is particularly beneficial when:

- The new dataset is small, preventing effective training of a large network from scratch due to overfitting.
- Computational resources or time for training a large model from scratch are limited.

By using a pre-trained model, we start with weights that already capture meaningful patterns (like edges, textures, shapes in images), allowing the new model to converge faster and perform better, especially with limited data.
There are typically two main approaches to applying transfer learning with CNNs:

- **Using the Pre-trained Network as a Fixed Feature Extractor**: In this approach, the pre-trained CNN (excluding its original final classification layer) is used to compute feature representations for the data in the new task. The weights of the pre-trained layers are kept frozen (not updated during training). The output of one of the last layers of the pre-trained network is treated as a fixed set of features, and a simple linear classifier (like a Support Vector Machine or a new small Fully-Connected network) is trained on these extracted features for the new task. This is effective when the new dataset is small and similar to the original training data.
- **Fine-tuning the Network**: This involves initializing the new model's weights with those of the pre-trained network. The last one or more layers (which are specific to the original task) are typically replaced with new layers suited for the new task. Training then continues on the new dataset. Depending on the size of the new dataset and its similarity to the original, one might fine-tune only the later layers of the network or the entire network, usually with a smaller learning rate than typical training from scratch to avoid disrupting the useful pre-trained features too quickly. Fine-tuning is generally more effective when the new dataset is larger or more different from the original training data.

Transfer learning is a powerful strategy that significantly accelerates training and improves the performance of CNNs on new tasks by building upon knowledge acquired from previous, related learning experiences.
Cited slides: lecture-17-03-25-IntroDeepLearning-CNN.pptx.pdf: 49, 50; lecture-20-03-25-IntroDLContd.pptx.pd
59, 60, 61, 62, 63, 64, 65, 66, 67.

7. What is a convolutional neural network? Describe the overall architecture and its main components, describe the training procedures and strategies with a particular focus on dropout regularization

A Convolutional Neural Network (CNN) is a type of deep neural network particularly effective for processing data with a grid structure, such as images. It is built with an architecture designed to automatically learn spatial hierarchies of features.
The overall architecture of a CNN typically comprises an Input Layer, Convolutional Layers (for feature detection), Activation Functions (for non-linearity), Pooling Layers (for dimensionality reduction), and Fully-Connected Layers leading to the Output Layer. Additional layers like Dropout layers are often included as a training strategy.
Training a CNN involves optimizing the network's parameters (weights and biases) to minimize a loss function using iterative algorithms like gradient descent variants, with gradients computed via backpropagation. A key strategy for improving generalization and preventing overfitting during this process is **Dropout Regularization**.
Dropout is a powerful regularization technique that is applied during the training of neural networks, including CNNs. The core idea is to randomly set a fraction of the neurons' outputs to zero at each training iteration (for a given mini-batch).
The principles and functioning of Dropout are:

- **Random Deactivation**: During each training step, for each layer where dropout is applied, a certain percentage of neurons are randomly ”dropped out” or deactivated, meaning their output is temporarily ignored, and they do not contribute to the forward pass or participate in backpropagation for that specific training iteration.

- **Preventing Co-adaptation**: By randomly dropping neurons, dropout prevents neurons from becoming overly reliant on specific other neurons (co-adapting). This forces the network to learn more robust and redundant representations, as any given neuron must be able to perform well regardless of which other neurons are active.

- **Ensemble Effect**: Dropout can be seen as training a large ensemble of different networks (each being a sub-network of the original model with some neurons dropped). At test time, the full network is used, which can be approximated as averaging the predictions of all these sub-networks, leading to improved generalization.

- **Training vs. Test Time**:
  - **Training**: Neurons are randomly dropped with a probability p_{drop} (or kept with a probability $p_{keep} = 1 - p_{drop}$). The outputs of the kept neurons are often scaled by $1/p_{keep}$ to maintain the same expected output sum as when all neurons are active.
  - **Testing**: Dropout is turned off. All neurons are active. To compensate for the difference in the number of active neurons between training and testing, the weights (or the activations during the forward pass) are scaled down by the keep probability (p_{keep}).

Dropout layers are typically inserted after convolutional layers or fully-connected layers, often before the activation function (though sometimes after).
The benefits of using Dropout include it being a powerful regularization technique, simple to implement, computationally cheap during training, and significantly improving the generalization ability of the network, helping to mitigate overfitting, especially in large models.
Cited slides: lecture-17-03-25-IntroDeepLearning-CNN.pptx.pdf: 49, 50; lecture-20-03-25-IntroDLContd.pptx.pd
38, 40, 41, 42, 43, 44, 45, 46, 50.

8 Deep Learning Architectures: Alexnet, VGG, ResNet, RNNs

1. Describe the AlexNet architecture
AlexNet is a pioneering Convolutional Neural Network (CNN) architecture that achieved a breakthrough in computer vision by winning the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2012. Its success demonstrated the power of deep CNNs on large-scale image recognition tasks.
The architecture of AlexNet is deeper than previous networks and consists of several stacked layers:

1. **Input Layer**: Takes in images (e.g., 224x224x3 pixels).
2. **CONV1**: Applies 96 filters of size 11x11 with a stride of 4.
3. **POOL1**: Performs max pooling with a 3x3 window and a stride of 2.
4. **CONV2**: Applies 256 filters of size 5x5 with a stride of 1 and padding of 2.
5. **POOL2**: Performs max pooling with a 3x3 window and a stride of 2.
6. **CONV3**: Applies 384 filters of size 3x3 with a stride of 1 and padding of 1.
7. **CONV4**: Applies 384 filters of size 3x3 with a stride of 1 and padding of 1.
8. **CONV5**: Applies 256 filters of size 3x3 with a stride of 1 and padding of 1.
9. **POOL3**: Performs max pooling with a 3x3 window and a stride of 2.
10. **FC6**: A fully-connected layer with 4096 neurons.
11. **FC7**: Another fully-connected layer with 4096 neurons.
12. **FC8**: The final fully-connected layer with 1000 neurons, corresponding to the 1000 classes of the ImageNet challenge, typically followed by a softmax activation for classification probabilities.

AlexNet utilized ReLU activation functions after convolutional and fully-connected layers, which helped accelerate training compared to sigmoid or tanh. Other key strategies contributing to its success included the use of Dropout in the fully-connected layers to reduce overfitting, extensive data augmentation, and a slightly overlapping pooling operation. Historically, it was also trained across two GPUs due to computational limitations at the time.
Cited slides: lecture - 24-03-25-DL-Architectures-Visualizations.pdf: 6, 7, 8, 9, 10, 11.

2. Describe the VGG network architecture
VGGNet is a Convolutional Neural Network (CNN) architecture that was a runner-up in the ImageNet challenge in 2014. It is known for its simplicity and uniformity, demonstrating that the depth of the network is a critical component for good performance.
The key principle of the VGG architecture is the exclusive use of very small convolutional filters and pooling windows throughout the network:

- **Convolutional Layers**: Use only 3x3 filters with a stride of 1 and padding of 1.
- **Pooling Layers**: Use only 2x2 max pooling windows with a stride of 2.

VGG shows that stacking multiple 3x3 convolutional layers consecutively before a pooling layer is effective. For instance, stacking two 3x3 convolutional layers has a receptive field equivalent to a single 5x5 convolutional layer, but with fewer parameters and the benefit of multiple non-linear activation function applications. Similarly, stacking three 3x3 layers achieves a 7x7 receptive field with even fewer parameters than a single 7x7 filter.
The overall architecture of VGG consists of several blocks of convolutional layers followed by a pooling layer, with the number of filters increasing in deeper blocks (e.g., 64, 128, 256, 512). After these convolutional and pooling blocks, the network typically ends with a few Fully-Connected layers and a final classification layer.
The two most common variants are VGG-16 and VGG-19, referring to the total number of layers with learnable weights (convolutional and fully-connected layers). Despite its simplicity in filter design, VGGNet is quite deep and contains a large number of parameters, particularly in the fully-connected layers.
Cited slides: lecture - 24-03-25-DL-Architectures-Visualizations.pdf: 6, 13, 14, 15, 16, 17, 18.

3. Describe the Residual Network (ResNet) architecture
Residual Networks (ResNet) represent a significant advancement in CNN architectures, particularly effective for training very deep networks. ResNet won the ImageNet challenge in 2015. The architecture was designed to address the "Degradation Problem," where simply stacking more layers in a deep neural network can lead to decreased accuracy, not due to overfitting, but because the network becomes harder to train and optimize.
The core innovation of ResNet is the introduction of **Identity Shortcut Connections** or **Skip Connections**. These connections allow the input of a block of layers to be added directly to the output of that block, before the final activation function. This structure forms a **Residual Block**. The output of a residual block can be formulated as $y = F(x, \{W_i\}) + x$, where x is the input to the block, F represents the function learned by the stacked layers within the block (e.g., two convolutional layers and activations), and y is the output.
The intuition behind this design is that it is easier for the stacked layers within the block to learn a *residual mapping* $F(x)$ than to learn the complete desired mapping $H(x)$ directly. If the optimal mapping is close to an identity function, the residual block can simply learn $F(x) = 0$, which is easier than learning $H(x) = x$ with non-linear layers. These skip connections facilitate the flow of gradients during backpropagation, alleviating the vanishing gradient problem and enabling the training of networks with unprecedented depth.
The overall ResNet architecture is constructed by stacking multiple such Residual Blocks. For deeper variants of ResNet, such as ResNet-50, ResNet-101, and ResNet-152, a **Bottleneck Design** is often used within the residual block to reduce computational cost. This design uses 1x1 convolutions to reduce dimensionality before the main 3x3 convolution and then another 1x1 convolution to restore the dimensionality,

effectively creating a bottleneck for the feature maps.
ResNet architectures like ResNet-18, ResNet-34, ResNet-50, ResNet-101, and ResNet-152 are distinguished by their number of layers. The introduction of ResNet significantly advanced the field by showing that very deep networks could be effectively trained, becoming a fundamental building block for many subsequent architectures.
Cited slides: lecture - 24-03-25-DL-Architectures-Visualizations.pdf: 6, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31.

4. Describe the Recurrent Neural Network (RNN) architecture

Based on the content of the current PDF ”lecture - 27-03-25-RNN-Transformers.pdf”, a Recurrent Neural Network (RNN) is a type of neural network designed specifically for processing sequential data, such as time series, text, or speech. Unlike traditional feed-forward networks that process inputs independently, RNNs have a form of ”memory” that allows them to use information from previous steps in the sequence to inform the processing of the current step.
The core characteristic of an RNN is the presence of a **loop** or a recurrent connection. This loop allows information to be passed from one time step to the next. When processing a sequence of inputs $(x_1, x_2, \dots, x_T)$, the RNN iteratively updates a hidden state (h_t) at each time step t. The hidden state h_t serves as the network's memory, theoretically containing information about the entire sequence processed up to step t.
At each time step t, the RNN takes two inputs: the current input from the sequence (x_t) and the hidden state from the previous time step (h_{t-1}). It then computes a new hidden state (h_t) and an output (y_t) based on these inputs. The process can be unrolled over time to visualize the flow of information through the sequence.
Mathematically, a simple ”Vanilla” RNN typically computes the hidden state and output using the following equations:

$$h_t = f_W(h_{t-1}, x_t)$$

$$y_t = f_W(h_t)$$

More specifically, using common transformations like matrix multiplications and a non-linear activation function (e.g., tanh):

$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t + b_h)$$

$$y_t = W_{hy}h_t + b_y$$

where W_{hh}, W_{xh}, and W_{hy} are weight matrices, and b_h and b_y are bias vectors, all of which are learned during training. The same weights are reused across all time steps, which is a form of parameter sharing. This recurrent structure enables RNNs to model temporal dependencies and process sequences of variable length.
Cited slides: lecture - 27-03-25-RNN-Transformers.pdf: 5, 6, 7, 8, 9, 10, 11.

5. Considering the AlexNet architecture, explain how the choice of the ReLU activation function and the use of dropout, combined with data augmentation, allowed to successfully train such a deep model on the large-scale ImageNet dataset. Which problems do these components address, and why were they particularly important for overcoming training difficulties?

The success of training the relatively deep AlexNet architecture on the large-scale ImageNet dataset was significantly enabled by the combination of key techniques, including the choice of the ReLU activation function, the use of dropout, and data augmentation. These components were crucial for overcoming the training difficulties associated with deep networks and large datasets at the time.
Each of these techniques addressed specific problems:

- **ReLU Activation Function**: AlexNet was one of the first major networks to widely adopt the Rectified Linear Unit (ReLU) as the activation function instead of the more traditional sigmoid or tanh functions. According to the material, using ReLU allowed for **Faster Training**. This was particularly important because training deep networks on a massive dataset like ImageNet is computationally intensive and time-consuming. ReLU's property of not saturating for positive inputs helps mitigate the vanishing gradient problem that plagued training with sigmoid/tanh in deep networks, enabling more efficient gradient flow and thus accelerating the learning process.

- **Dropout**: Dropout is a regularization technique applied to the fully-connected layers in AlexNet. Its primary role was in **Overcoming Overfitting**. Overfitting occurs when a model learns the training data too well, including noise, and performs poorly on unseen data. With a large model like AlexNet and a complex dataset like ImageNet, the risk of overfitting is substantial. Dropout randomly deactivates neurons during training, preventing complex co-adaptations and forcing the network to learn more robust features, thereby improving generalization. This was vital for achieving good performance on the ImageNet validation and test sets.
- **Data Augmentation**: Data augmentation involves artificially increasing the size and diversity of the training dataset by applying random transformations (like cropping, flipping, color jittering) to the original images. Like dropout, data augmentation also contributed significantly to **Overcoming Overfitting**. By presenting the network with varied versions of the training images, it forces the model to be more invariant to such transformations, making it generalize better to new images it hasn't seen before. This was particularly important for handling the vast variability present in the real-world images of the ImageNet dataset.

These techniques were particularly important for overcoming training difficulties because they addressed the fundamental challenges of training deep models: the computational cost and convergence speed (ReLU), and the risk of overfitting when increasing model capacity to handle complex data (Dropout and Data Augmentation). Their combined effect was instrumental in enabling AlexNet to successfully train on ImageNet and achieve state-of-the-art results in 2012.
Cited slides: lecture - 24-03-25-DL-Architectures-Visualizations.pdf: 7, 11.

6. The VGG network demonstrated that significantly increasing network depth could lead to better performance on complex tasks like ImageNet classification. A key aspect of VGG was its highly uniform architecture, primarily relying on repeating stacks of small 3x3 convolutional filters and 2x2 pooling layers. What did the success of VGG suggest about the relationship between network depth and the ability to learn powerful visual representations?
The success of the VGG network on complex tasks like ImageNet classification, particularly with its emphasis on increased depth through a uniform architecture of stacked small 3x3 convolutional filters and 2x2 pooling layers, strongly suggested that **network depth is a critical factor for learning powerful and hierarchical visual representations**.
VGG's performance gains, built simply by going deeper with consistent, small convolutional and pooling filters, indicated that:

- Deeper networks are capable of learning more complex and abstract feature representations from images compared to shallower networks. The hierarchical nature of visual features (edges form corners, corners form parts, parts form objects) is effectively captured by progressively combining simpler features learned in earlier layers through increased depth.
- Simply stacking more layers, provided the architecture is designed consistently (like using only small filters), is an effective strategy for improving the model's representational capacity and, consequently, its performance on challenging visual recognition benchmarks.

The VGG network's approach, proving the benefits of depth through a straightforward design, paved the way for even deeper architectures and highlighted the importance of building deep hierarchies for sophisticated visual understanding tasks.
Cited slides: lecture - 24-03-25-DL-Architectures-Visualizations.pdf: 13, 14, 15, 16, 17.

7. GoogLeNet (with its Inception modules) aimed to create deeper networks while being more computationally efficient. Explain the core idea behind the Inception module. Discuss the computational challenges this approach initially faced and how the use of 1x1 convolu-

tions (”bottlenecks”) helped manage this complexity and reduce the number of parameters.

GoogLeNet, the winning architecture of the ImageNet challenge in 2014 (alongside VGG), aimed to create deeper networks than previous models while maintaining or improving computational efficiency and reducing the number of parameters. This was achieved through the introduction of the **Inception module**.

The core idea behind the **Inception module** is to allow the network to capture salient features at multiple different scales within the same layer. This is done by performing several different types of operations in parallel on the same input feature map and concatenating their outputs along the depth dimension. A naive Inception module consists of parallel branches with 1x1, 3x3, and 5x5 convolutional layers, as well as a max pooling layer. All these operations are applied to the input, and their resulting feature maps are stacked together as the output of the module. This allows the network to learn representations that combine information extracted at various spatial scales.

However, this naive approach faced a significant **computational challenge**. Applying 3x3 and especially 5x5 convolutions directly on the input feature maps, which can have a considerable depth (number of channels), is computationally expensive. This can lead to a large number of floating-point operations and a substantial increase in the number of parameters within the module, which would hinder the goal of computational efficiency.

GoogLeNet addressed this complexity and reduced the number of parameters through the clever use of **1x1 convolutions**, often referred to as ”**bottlenecks**”. In the refined Inception module, 1x1 convolutional layers are placed *before* the computationally more expensive 3x3 and 5x5 convolutional layers (and also after the pooling layer). The primary function of these 1x1 convolutions is **dimensionality reduction**. By controlling the number of filters in the 1x1 convolution, the number of input feature maps (depth) going into the 3x3 and 5x5 convolutions is significantly reduced.

This dimensionality reduction using 1x1 bottlenecks dramatically lowers both the computational cost (fewer multiplications and additions) and the number of parameters in the 3x3 and 5x5 branches. The 1x1 convolution itself is a computationally cheap operation. By intelligently using these bottlenecks, the Inception module can maintain its ability to capture multi-scale features while being significantly more efficient than the naive design, allowing for the construction of much deeper networks like GoogLeNet without prohibitive computational demands.

Cited slides: lecture - 24-03-25-DL-Architectures-Visualizations.pdf: 6, 32, 33, 34, 35, 36, 37, 38, 39, 40.

8. As networks got deeper post-VGG, researchers observed the ”degradation” problem where simply stacking more layers hurt performance, even on the training set, indicating an optimization challenge rather than just overfitting. Explain this phenomenon. Describe the fundamental innovation introduced by ResNet and intuitively explain how this mechanism helps overcome the optimization difficulty and enables the training of much deeper networks more effectively

Following the success of VGG in demonstrating the benefits of depth, researchers attempted to train even deeper networks. However, they observed a phenomenon known as the ”**degradation problem**”. In very deep networks, simply adding more layers would lead to higher training error compared to shallower networks, after convergence. This was not due to overfitting, as the performance was worse even on the training data, indicating an optimization difficulty rather than a lack of model capacity or generalization issue. It became harder to train the deeper models effectively with standard optimization methods.

The fundamental innovation introduced by Residual Networks (ResNet) to address this degradation problem is the **Identity Shortcut Connection** or **Skip Connection**, leading to the **Residual Block**. Instead of expecting a few stacked layers to learn a complex mapping $H(x)$ from input x to output $H(x)$, ResNet proposes that these layers should learn a *residual mapping* $F(x)$, and the output of the block is $y = F(x) + x$. The skip connection simply passes the original input x directly across the layers and adds it to the output of the transformation learned by those layers, $F(x)$.

Intuitively, this mechanism helps overcome the optimization difficulty because it makes it much easier for the network to learn an identity mapping. If the optimal function for a block is just to pass the input through unchanged (i.e., $H(x) = x$), then the layers within the residual block only need to learn the residual function $F(x) = 0$. Learning to output zero is significantly easier for standard optimization algorithms

than learning to output the identity function $H(x) = x$ directly through multiple non-linear layers. The skip connection essentially provides a "highway" for information and gradients to flow directly through layers. This facilitates gradient propagation during backpropagation, helping to mitigate the vanishing gradient problem which makes training very deep networks difficult. By making it easier to learn residual mappings and improving gradient flow, ResNet's architecture effectively enables the training of networks with hundreds or even thousands of layers, achieving state-of-the-art performance on various tasks.
Cited slides: lecture - 24-03-25-DL-Architectures-Visualizations.pdf: 19, 20, 21, 22, 23, 24, 25.

9 From RNNs to Transformers

1. What is an LSTM?
An LSTM, which stands for Long Short-Term Memory, is a specific type of Recurrent Neural Network (RNN) architecture. It was designed to address the limitations of traditional RNNs, particularly their difficulty in learning and remembering information over long sequences due to issues like the vanishing gradient problem.
The core idea behind an LSTM is the introduction of a "cell state" (or memory cell) that acts as a highway for information to flow unchanged through the network. This cell state is regulated by several multiplicative "gates" that control the flow of information into, out of, and within the cell.
The main components of an LSTM cell and their roles are:

- **Cell State**: This is the key to LSTMs. It runs through the entire sequence, carrying information. Information is added to or removed from the cell state, but the gates control this flow, allowing information to be preserved over long time steps.

- **Forget Gate**: This gate decides what information to throw away from the cell state. It looks at the current input and the previous hidden state and outputs a number between 0 and 1 for each number in the cell state, where 1 means "keep this" and 0 means "forget this".

- **Input Gate**: This gate decides what new information to store in the cell state. It consists of two parts: an input gate layer which decides which values to update, and a tanh layer that creates a vector of new candidate values. These two parts are combined to update the cell state.

- **Output Gate**: This gate decides what to output as the hidden state for the current time step. It looks at the current input and the updated cell state and outputs a filtered version of the cell state, producing the hidden state which is then passed to the next time step and potentially used for prediction.

By carefully orchestrating the flow of information through these gates and the cell state, LSTMs are able to capture and retain relevant information from earlier steps in a sequence, making them highly effective for tasks involving sequential data with long-term dependencies, such as natural language processing, speech recognition, and time series forecasting.
Cited slides: lecture - 27-03-25-RNN-Transformers.pdf: 32, 33, 34, 35, 36, 37, 38.

2. What are the main differences between an RNN and a Transformer?
Recurrent Neural Networks (RNNs) and Transformers are both types of neural network architectures designed to process sequential data, but they differ fundamentally in their approach to modeling dependencies within a sequence. The main differences between a standard (or even LSTM/GRU) RNN and a Transformer are:

- **Sequential vs. Parallel Processing**: This is the most significant difference. Traditional RNNs process sequences token by token, maintaining a hidden state that is updated at each time step. This sequential nature makes training and inference slow, especially for long sequences, and limits parallelization. Transformers, on the other hand, process the entire sequence in parallel. They achieve this by discarding the recurrence mechanism and relying on attention.

- **Core Mechanism**: RNNs use recurrence loops and hidden states to maintain memory of past information in the sequence. Transformers entirely dispense with recurrence and convolutions. Instead, they rely primarily on a **self-attention mechanism** to relate different positions of the sequence to compute a representation for each element.
- **Handling Long-Range Dependencies**: While LSTMs and GRUs improved upon basic RNNs in capturing long-term dependencies, they can still struggle with very long sequences where information needs to be passed through many time steps. Transformers, particularly through the self-attention mechanism, can directly compute the relationship between any two positions in the sequence regardless of their distance, making them generally more effective at capturing long-range dependencies.
- **Computational Efficiency and Parallelizability**: Due to their sequential nature, RNNs are difficult to parallelize across time steps, making training computationally expensive for long sequences. Transformers, with their parallel attention computations, are highly parallelizable, leading to significantly faster training times on modern hardware.
- **Information Flow**: RNNs pass information through a fixed-size hidden state, which can create an information bottleneck for very long sequences. Transformers use attention to access relevant information from other positions in the sequence more directly, without necessarily compressing it into a single vector representation per step.

In essence, the Transformer's reliance on attention allows it to overcome the sequential processing and long-range dependency limitations inherent in RNN architectures, leading to improved performance and computational efficiency on many sequence processing tasks.
Cited slides: lecture - 27-03-25-RNN-Transformers.pdf: 3, 4, 40, 41, 42, 45, 46, 58.

3. Discuss the concept of attention and its implementation (general attention layer) in the context of RNNs and Transformers

The concept of attention in neural networks is a mechanism that allows the model to dynamically focus on the most relevant parts of the input sequence or representation when processing a specific element or generating an output. Instead of relying solely on processing sequential data step-by-step and compressing information into a fixed-size hidden state (as in traditional RNNs), attention provides a way for the model to weigh the importance of different input elements relative to a current point of focus.

While the provided material introduces Attention primarily as the fundamental mechanism of Transformers, it highlights the limitations of Recurrent Neural Networks (RNNs) in handling long-range dependencies and their sequential computation as motivations for alternative architectures. Attention addresses these limitations by providing a method to establish direct connections between any elements in a sequence, regardless of their distance.

The implementation of a general attention mechanism involves computing a weighted sum of "Value" vectors, where the weights are determined by the similarity between a "Query" vector and "Key" vectors associated with the elements being attended over. The basic attention function can be described as:

$$Attention(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

where Q is the matrix of Queries, K is the matrix of Keys, V is the matrix of Values, and d_k is the dimension of the Keys, used for scaling. The softmax function ensures that the weights for the Value vectors sum to 1.

In the context of **Transformers**, attention, specifically **Self-Attention**, is the core mechanism that replaces recurrence entirely. Self-attention allows each element in an input sequence to attend to all other elements within the *same* sequence. The Queries, Keys, and Values are all derived from the same input sequence representation. This enables the Transformer to capture global dependencies within the sequence in a single step, facilitating parallel computation and effectively modeling relationships between distant words or tokens, unlike the sequential processing of RNNs. The Transformer architecture also uses **Multi-Head Attention**, which performs the attention mechanism multiple times in parallel with

different learned linear projections for Q, K, and V, allowing the model to jointly attend to information from different representation subspaces at different positions.
Thus, while the principle of attention is general, its implementation in Transformers, particularly through Self-Attention, is central to their ability to process sequences non-sequentially and effectively model long-range dependencies, addressing key challenges faced by RNNs.
Cited slides: lecture - 27-03-25-RNN-Transformers.pdf: 40, 41, 42, 45, 46, 47, 48, 49, 50, 51, 52, 58.

4. Describe attention layers and self attention layers, discussing similarities and differences
In the context of neural networks, particularly Transformers, attention is a mechanism that allows the model to weigh the importance of different elements in an input sequence when computing the representation of a specific element or generating an output.
A **general attention layer** computes an output based on three inputs: Queries (Q), Keys (K), and Values (V). The output is a weighted sum of the Value vectors, where the weight assigned to each Value is determined by the compatibility or similarity between the Query and the corresponding Key. The scaled dot-product attention function, commonly used, is defined as:

$$Attention(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

Here, Q, K, and V are matrices representing sets of queries, keys, and values, and d_k is the dimension of the key vectors used for scaling. The softmax function normalizes the scores into weights that sum to 1.
Self-attention is a specific instance of the general attention mechanism. In self-attention, the Queries, Keys, and Values all come from the **same source sequence** or the output of a preceding layer processing that same sequence. This allows each element in the sequence to attend to all other elements within the *same* sequence, including itself, to compute a contextualized representation.
Similarities:

- Both attention layers and self-attention layers are based on the same core mechanism of computing similarity scores between Queries and Keys to obtain weights for a weighted sum of Values.
- They often use the same fundamental operations, such as dot products (or other similarity measures) between Q and K, followed by a softmax function to get attention weights, and finally multiplying these weights by V. The formula $Attention(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$ applies to both, depending on how Q, K, and V are defined.

Differences:

- The primary difference lies in the **source of the Queries, Keys, and Values**. In self-attention, Q, K, and V are derived from the *same* sequence. In a more general attention setting (often called cross-attention, for example, in the decoder of a Transformer), Q typically comes from one source (e.g., the decoder's input) while K and V come from a different source (e.g., the encoder's output sequence).
- **Purpose**: Self-attention is used to model relationships *within* a single sequence and capture internal dependencies between its elements. General attention (cross-attention) is often used to relate one sequence (e.g., the target being generated) to another sequence (e.g., the source input).

In summary, self-attention is a specialized application of the general attention mechanism where the focus is on relating elements within the same sequence, enabling each element to incorporate information from all other elements in that sequence based on their learned relevance.
Cited slides: lecture - 27-03-25-RNN-Transformers.pdf: 45, 46, 47, 48, 49, 50, 51, 52.

5. Describe self attention layers and their masked and multi-head versions
Self-attention is a core mechanism in Transformer networks that allows each element in a sequence to attend to all other elements within the **same sequence**. It computes a representation for each position by calculating attention scores between that position's Query vector and all other positions' Key vectors (including its own), and then using these scores to take a weighted sum of all positions' Value vectors.

The Queries, Keys, and Values are all derived from the same input sequence representation through linear transformations. The standard scaled dot-product self-attention is given by:

$$Attention(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$$

where Q, K, V are matrices representing the queries, keys, and values, and d_k is the dimension of the keys.
Masked Self-Attention is a variation of self-attention typically used within the decoder part of a Transformer model, particularly in tasks involving sequential generation (like language generation). The purpose of masking is to **prevent positions from attending to subsequent positions** in the sequence. This is crucial because when generating a sequence token by token, the prediction for the current token should only depend on the tokens that have already been generated (the past and current positions), not future ones. The masking is implemented by setting the attention scores corresponding to future positions to a very large negative value (like $-\infty$) *before* applying the softmax function. This ensures that the softmax output for masked connections becomes zero, effectively preventing information flow from future positions.

Multi-Head Attention is another key component built on the self-attention mechanism. Instead of performing a single attention function with the full dimension of Queries, Keys, and Values, Multi-Head Attention linearly projects Q, K, and V multiple times with different learned projections. It then performs the attention function (often self-attention) on each of these projected versions in parallel. Each parallel attention computation is called an "attention head". The outputs from these multiple heads are then concatenated and linearly transformed to produce the final output. The idea behind Multi-Head Attention is that it allows the model to jointly attend to information from different representation subspaces and at different positions simultaneously, potentially capturing different types of relationships within the data.

Similarities:

- Both Masked Self-Attention and Multi-Head Attention are based on the fundamental Self-Attention mechanism.
- They all utilize the core attention calculation involving Queries, Keys, and Values, typically using the scaled dot-product formula.

Differences:

- **Masked vs. Standard Self-Attention**: Masked Self-Attention adds a constraint by preventing attention to future sequence positions, whereas standard Self-Attention allows attention to all positions.
- **Multi-Head vs. Single-Head Self-Attention**: Multi-Head Attention runs multiple attention mechanisms in parallel with different learned linear projections and concatenates their results, while a single Self-Attention layer performs only one such computation.

In essence, Self-Attention captures dependencies within a sequence, Masked Self-Attention adapts this for sequential generation by enforcing causality, and Multi-Head Attention enhances the ability to capture diverse relationships by performing multiple attention computations concurrently.
Cited slides: lecture - 27-03-25-RNN-Transformers.pdf: 45, 46, 47, 52, 53, 54, 55, 56.

6. Describe Transformers and their main components, with special focus on the encoder block

Transformers are a neural network architecture introduced to process sequential data, designed to replace the recurrent mechanisms of RNNs with attention mechanisms. This allows for parallel processing of the sequence and improved handling of long-range dependencies.
The overall architecture of the Transformer follows an **Encoder-Decoder structure**. The Encoder processes the input sequence into a sequence of continuous representations, and the Decoder takes these representations and generates the output sequence. Both the Encoder and the Decoder are composed of a stack of N identical layers.
The main components of a Transformer include:

- **Positional Encoding**: Since the attention mechanism itself is invariant to the order of the sequence, positional encodings are added to the input embeddings to inject information about the relative or absolute position of tokens in the sequence.
- **Encoder Stack**: Composed of identical encoder layers.
- **Decoder Stack**: Composed of identical decoder layers (which are slightly different from encoder layers, including masked self-attention and cross-attention).
- **Multi-Head Attention Layers**: Used within both encoder and decoder layers to allow the model to jointly attend to information from different representation subspaces.
- **Feed-Forward Networks**: Simple, position-wise fully-connected networks applied within each encoder and decoder layer.
- **Residual Connections and Layer Normalization**: Used throughout the network to facilitate training of deep models.

Focusing on the **Encoder block**, it consists of a stack of identical encoder layers. Each single encoder layer has two main sub-layers:

1. **Multi-Head Self-Attention Mechanism**: This is the first sub-layer. It allows the encoder to attend to all positions in the input sequence to compute a weighted representation for each position.
2. **A Simple, Position-wise Fully Connected Feed-Forward Network**: This is the second sub-layer. It is applied independently and identically to each position in the sequence after the self-attention step.

Crucially, each of these two sub-layers within the encoder block is followed by a **Residual Connection** and **Layer Normalization**. That is, the output of each sub-layer is added to its input, and then layer normalization is applied. This helps in training deeper encoder stacks by facilitating gradient flow. The output of one encoder layer serves as the input to the next layer in the stack. The final output of the encoder stack is a sequence of context-aware representations for the input sequence.
Cited slides: lecture - 27-03-25-RNN-Transformers.pdf: 41, 42, 43, 44, 45, 52, 53, 59, 60, 61, 62, 63, 64, 65, 66.

7. Describe Transformers and their main components, with special focus on the decoder block

Transformers are a neural network architecture designed for processing sequential data, notable for replacing traditional recurrent mechanisms with attention mechanisms to enable parallel processing and better capture of long-range dependencies.
The overall architecture of the Transformer follows an **Encoder-Decoder structure**. The Encoder maps an input sequence to a sequence of continuous representations, and the Decoder then generates an output sequence one element at a time, based on the encoder's output and the previously generated elements. Both the Encoder and Decoder are composed of a stack of identical layers.
Key components of a Transformer include Positional Encodings (added to inputs), the Encoder stack, the Decoder stack, Multi-Head Attention layers, Feed-Forward Networks, Residual Connections, and Layer Normalization.
Focusing on the **Decoder block**, it is a stack of N identical decoder layers. Each single decoder layer is slightly different from an encoder layer, comprising three main sub-layers:

1. **Masked Multi-Head Self-Attention**: This is the first sub-layer. It allows the decoder to attend to the previous positions in the output sequence being generated (including the current position). It uses **Masked Self-Attention** to prevent attending to subsequent positions, which is essential during sequential decoding where future tokens are unknown.
2. **Multi-Head Attention over the Encoder Stack Output**: This is the second sub-layer. It performs attention over the output of the encoder stack. Here, the Queries come from the output of

the previous masked self-attention sub-layer, and the Keys and Values come from the final output of the encoder stack. This allows the decoder to focus on relevant parts of the input sequence when generating the output at each step.

3. **A Simple, Position-wise Fully Connected Feed-Forward Network**: This is the third sub-layer, identical in structure to the one used in the encoder. It is applied independently and identically to each position in the sequence.

Similar to the encoder, each of these three sub-layers within the decoder block is followed by a **Residual Connection** and **Layer Normalization** before being passed to the next sub-layer or the next layer in the stack. The output of the final decoder layer is processed by a linear layer and softmax to produce the probability distribution over the vocabulary for the next output token.
Cited slides: lecture - 27-03-25-RNN-Transformers.pdf: 41, 42, 43, 44, 54, 55, 56, 65, 66.