Q1:

Symbolic AI and Machine Learning (ML) are two fundamental paradigms in the field of artificial intelligence, each with its own approach to enabling machines to simulate intelligent behavior.

Symbolic AI (Also Known as Good Old-Fashioned AI - GOFAI)

Definition: Symbolic AI involves the use of explicit, human-understandable symbols to represent knowledge and logical reasoning to solve problems. This approach is rooted in the belief that human intelligence can be emulated through the manipulation of symbols and the application of rules.

How it Works: In Symbolic AI, intelligence is achieved through the use of data structures (like graphs, trees) for knowledge representation and symbolic reasoning mechanisms (such as rule-based systems) that apply logical operations on these representations to derive conclusions or make decisions.

Examples:

1. **Expert Systems:** These are computer systems that emulate the decision-making ability of a human expert. For example, MYCIN, developed in the early 1970s, was an expert system used in medical diagnosis to identify bacteria causing severe infections and to recommend antibiotics.
2. **Logical Reasoning Systems:** Systems like Prolog (a programming language based on logic) allow the definition of facts and rules which the system uses to perform logical reasoning and problem-solving. For instance, Prolog can be used to solve puzzles by defining the puzzle constraints as rules.
3. **Symbolic Mathematics Programs:** Software like Mathematica or Maple uses symbolic AI to manipulate mathematical symbols and solve algebraic equations, perform differentiation, integration, and more, much like a human would do by hand.

Machine Learning

Definition: Machine Learning is a subset of AI that focuses on the ability of machines to learn from data and make predictions or decisions without being explicitly programmed to perform the task. ML relies on algorithms that can analyze and learn from data patterns.

How it Works: In ML, a model is trained using a large set of data, and the algorithm adjusts its parameters to minimize the error in its predictions or decisions. Once the model is trained, it can make predictions on new, unseen data.

Examples:

1. **Supervised Learning:** Algorithms like linear regression or neural networks are trained on labeled data. For example, a spam filter in email is trained with many examples of spam and not-spam emails so that it can learn to classify new emails.
2. **Unsupervised Learning:** Algorithms such as k-means clustering or Principal Component Analysis (PCA) identify patterns or groupings in data without the need for labels. For

example, customer segmentation in marketing can be achieved by clustering customers based on purchasing behavior.

3. **Reinforcement Learning:** In this approach, an agent learns to make decisions by taking actions in an environment to achieve some goals. For example, AlphaGo, developed by DeepMind, uses reinforcement learning to play the board game Go at a level that outperforms human world champions.

Contrast:

- **Knowledge Representation:** Symbolic AI relies on explicit representations and logical reasoning, whereas ML learns representations from data.
- **Flexibility and Adaptability:** Symbolic AI systems can struggle with tasks that are not well-defined or where the knowledge base is incomplete, whereas ML can adapt to new data or changes in the data distribution.
- **Complexity and Scalability:** Building and maintaining large-scale symbolic AI systems can be complex and labor-intensive, requiring extensive domain knowledge. ML models, once trained, can scale and adapt more easily, although they require large amounts of data and computational resources.

In summary, Symbolic AI and Machine Learning represent two complementary approaches to artificial intelligence. Symbolic AI excels in domains where rules and logic dominate, and human-like reasoning is essential. In contrast, Machine Learning shines in handling vast amounts of data, learning from patterns, and making predictions or decisions in complex, dynamic environments.

SYMBOLIC AI INVOLVES THE USE OF HUMAN-UNDERSTANDABLE SYMBOLS TO REPRESANT KNOWLEDGE AND REASONING. THIS APPROACH IS BASED ON THE BELIEF THE HUMAN INTELLIGENCE CAN BE EMULATED THROUGH THE USE OF SYMBOLS AND RULES. TREES AND GRAPHS ARE USED TO REPRESENT THE KNOWLEDGE AND SYMBOLIC REASONING MECHANISMS (RULE-BASED SYSTEMS) ARE USED TO GET PREDICTIONS OR DERIVE CONCLUSIONS.

Autonomous cars use Symbolic AI to make decisions based on the environment, such as recognizing stop signs and traffic lights.

Q2

In a Multi-Layer Perceptron (MLP) using backpropagation for training, the update applied to a weight in a hidden layer is determined by the gradient descent algorithm. The goal is to minimize the loss function by adjusting the weights in the direction that most steeply decreases the loss. The formula for updating a weight in a hidden layer can be broken down into several key components, which involve the derivative of the loss function with respect to the weight in question. Here's a simplified version of the formula:

$$\Delta wij=-\eta\partial wij\partial L$$

Where:

- Δwij is the update applied to the weight from neuron i in one layer to neuron j in the next.
- η is the learning rate, a small positive scalar determining the size of the step taken in the direction of the negative gradient.
- $\partial wij \partial L$ is the partial derivative of the loss function L with respect to the weight wij.

To compute $\partial wij \partial L$, the chain rule is applied through the network layers during backpropagation. For a hidden layer, this involves several terms:

1. The derivative of the loss function with respect to the output of the neuron the weight is leading to ($\partial oj \partial L$).
2. The derivative of the activation function of the neuron with respect to its total input ($\partial netj \partial oj$), where $netj$ is the weighted sum of inputs to neuron j.
3. The output of the preceding neuron (oi), which is the input to the weight wij.

Combining these, the derivative part of the update formula is given by:

$$\partial wij \partial L = \partial oj \partial L \cdot \partial netj \partial oj \cdot oi$$

Therefore, the update rule for the weight wij in a hidden layer during backpropagation can be more specifically written as:

$$\Delta wij = -\eta(\partial oj \partial L \cdot \partial netj \partial oj \cdot oi)$$

In practice, for a network with sigmoid activation functions or other non-linearities, the specific forms of $\partial netj \partial oj$ and $\partial oj \partial L$ depend on the choice of the activation function and the loss function. This is how the network learns by iteratively updating its weights to reduce the loss, with each step of backpropagation adjusting the weights in the direction that reduces the error between the predicted output and the actual target values.

Q3

L1
L2#### network.py
class Network(object):
L3
L4
L5
L6
L7def __init__(self, sizes):
self.num_layers = len(sizes)
self.sizes = sizes
self.biases = [np.random.randn(y, 1) for y in sizes[1:]]
self.weights = [np.random.randn(y, x) for x, y in zip(sizes[:-1], sizes[1:])]
L8
L9
L10#### test.py
import network
net = network.Network([784, 30, 10])

What are the dimesnions of the three lists
1. sizes,
2. sizes[1:]
3. zip(sizes[:-1],sizes[1:])

q4

The vanishing gradient problem is a difficulty found in training artificial neural networks with gradient-based learning methods and backpropagation. In such systems, gradients are used to update the weights of the network, and these gradients are calculated through the process of backpropagation from the output layer back to the input layer. A gradient signifies the change in the loss function (a measure of the difference between the network's prediction and the actual data) with respect to the change in the network weights. Essentially, it tells us how to update our weights to minimize the loss.

What Causes Vanishing Gradients?

The vanishing gradient problem occurs when the gradient shrinks as it is propagated back through the network. When the gradients become very small, updates to the weights become insignificantly small. This makes the network hard to train and can result in the training process taking a very long time to converge, if it does at all. The problem is particularly pronounced in networks with many layers, known as deep networks.

Several factors contribute to the vanishing gradient problem:

1. **Activation Functions:** Traditional activation functions like the sigmoid or the hyperbolic tangent (tanh) function, which squish a large input space into a small output range, are prime suspects. These functions have gradients in the range (0, 1) for sigmoid and (-1, 1) for tanh. When these small values are multiplied during backpropagation, they tend to diminish exponentially, leading to vanishing gradients.
2. **Initialization Schemes:** Poorly chosen initialization methods for neural network weights can exacerbate the vanishing gradient problem. If the weights are too small, it can lead to small gradients, thus slowing down learning from the start.
3. **Deep Architectures:** The deeper the network (i.e., the more layers it has), the more severe the vanishing gradient problem can become, simply due to the repeated multiplication of small numbers as gradients are propagated backward through each layer.

Techniques to Reduce the Impact of Vanishing Gradients

Several techniques have been developed to mitigate the vanishing gradient problem:

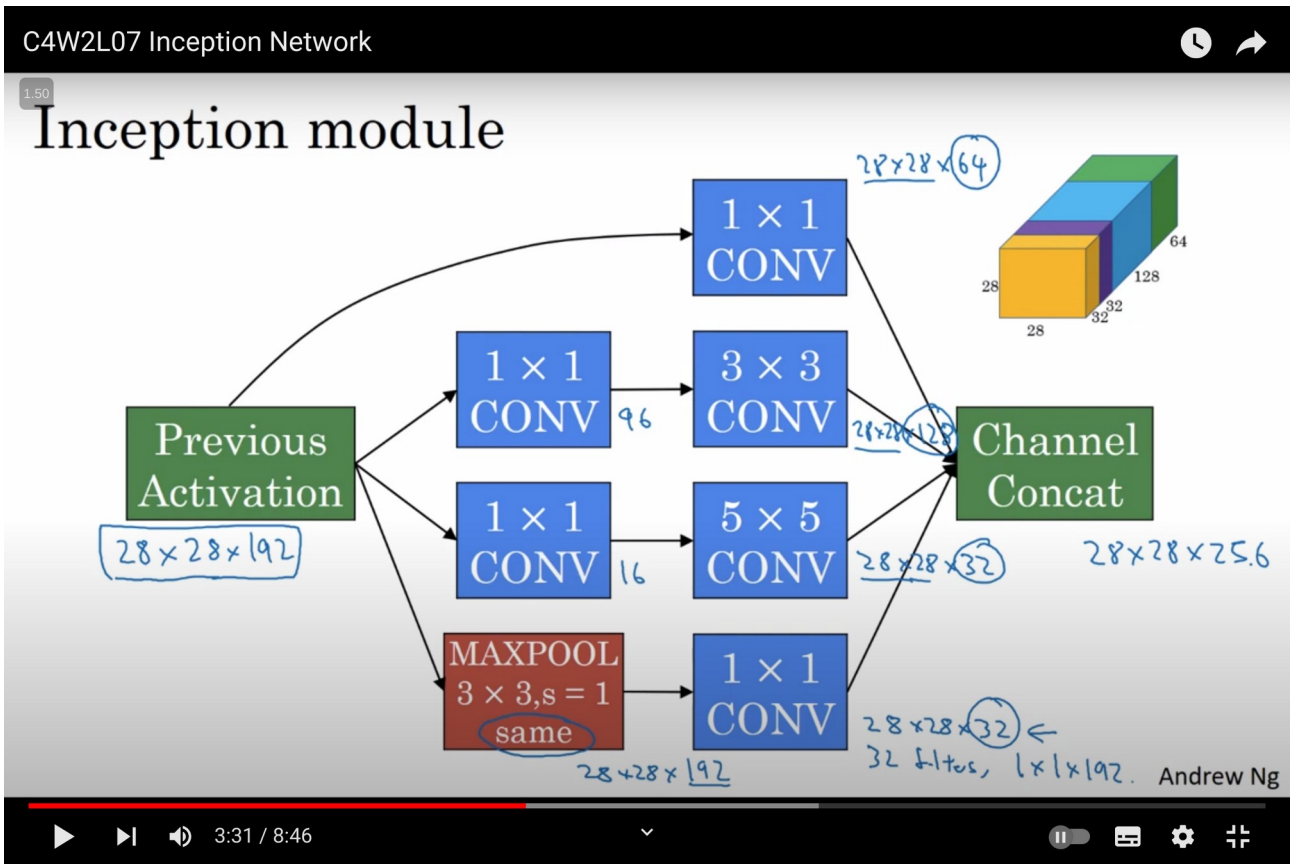
1. **Use of ReLU Activation Function:** The Rectified Linear Unit (ReLU) has become a popular choice because it does not saturate in the positive domain. The gradient of ReLU is either 0 (for negative inputs) or 1 (for positive inputs), which helps in preventing the gradient from vanishing during backpropagation. However, ReLU can lead to another

problem known as the "dying ReLU," where neurons stop learning entirely, but variants like Leaky ReLU or Parametric ReLU (PReLU) have been proposed to mitigate this.

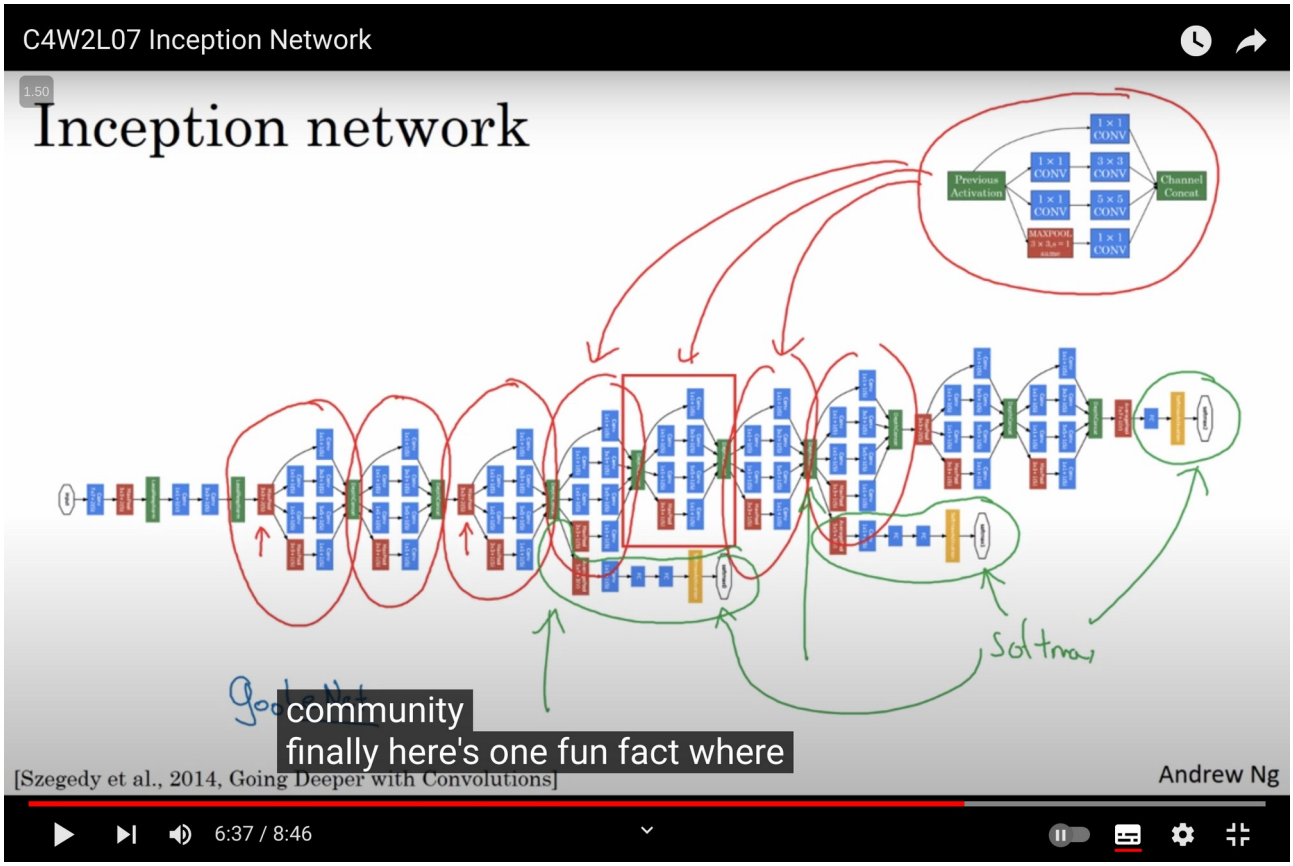
2. **Weight Initialization Techniques:** Choosing the right initialization method can alleviate the problem. Xavier (or Glorot) initialization and He initialization are techniques designed to keep the gradients in a reasonable range throughout the training process.
3. **Batch Normalization:** This technique normalizes the input of each layer in such a way that they have a mean output activation of zero and a standard deviation of one. This helps in combating the vanishing gradient problem by maintaining a stable distribution of activations throughout the training process.
4. **Skip Connections:** Architectures like ResNet introduce skip connections that bypass one or more layers by adding the input of a layer to its output. This helps in mitigating the vanishing gradient problem by allowing gradients to flow directly through these connections, making it easier to train deep networks.
5. **Gradient Clipping:** This technique involves clipping the gradients during backpropagation to prevent them from exceeding a specified threshold, which can help in preventing the gradients from vanishing (or exploding) during training.

Q5

GoogleLeNet, also known as Inception v1, introduced a novel architectural idea known as the Inception module, which significantly reduced the number of parameters in the network compared to its predecessors like AlexNet, while still maintaining or even improving performance on challenging benchmarks like ImageNet. The key to this reduction lies in the smart architectural design of the Inception module.


C4W2L07 Inception Network
Inception module
1 × 1 CONV
1 × 1 CONV
3 × 3 CONV
1 × 1 CONV
5 × 5 CONV
MAXPOOL 3 × 3,s = 1 same
1 × 1 CONV
Previous Activation
Channel Concat
28×28×192
96
16
28×28×64
28×28×128
28×28×32
28×28×256
28×28×32
32 filters, 1×1×192.
28×28×192
64
128
32
28
28
Andrew Ng
3:31 / 8:46

Jk


C4W2L07 Inception Network
Inception network
Previous Activation
Channel Concat
Softmax
community
finally here's one fun fact where
[Szegedy et al., 2014, Going Deeper with Convolutions]
Andrew Ng
6:37 / 8:46

Inception Module Explained

The Inception module is designed to capture information at various scales using parallel paths of convolutions with different kernel sizes (e.g., 1x1, 3x3, 5x5) and a 3x3 max pooling, all concatenated into a single output vector to be fed to the next layer. This design allows the network to adapt to features of different sizes in the input image without significantly increasing the computational burden.

Reduction of Parameters: The Role of 1x1 Convolutions

One of the ingenious aspects of the Inception module is its use of 1x1 convolutions. Before performing the computationally expensive 3x3 and 5x5 convolutions, the Inception module applies 1x1 convolutions for dimensionality reduction, significantly reducing the depth of the input volume to these convolutions. This step drastically cuts down the number of parameters and the computational cost.

Illustrative Calculation

Let's compare a simplified example to illustrate the parameter savings:

Without Inception:

- Assume we apply a 5x5 convolution directly to an input with a depth of 192 (similar to one of the early Inception layers) and we want to produce an output with a depth of 256.
- Each filter has a shape of 5x5x192, and we need 256 such filters.
- Total parameters: $5×5×192×256=1,228,800$.

With Inception (using 1x1 convolutions for dimensionality reduction):

- First, apply 1x1 convolutions to reduce the depth from 192 to 32 (just an example ratio for illustration).
- Then, apply the 5x5 convolution on the reduced depth.
- For the 1x1 convolution step: $1×1×192×32=6,144$ parameters.
- For the 5x5 convolution on the reduced depth: $5×5×32×256=204,800$ parameters.
- Total parameters with reduction: $6,144+204,800=210,944$.

By applying dimensionality reduction, the Inception module drastically reduces the number of parameters from over 1.2 million to just over 210k for this layer, achieving a significant reduction.

Q6

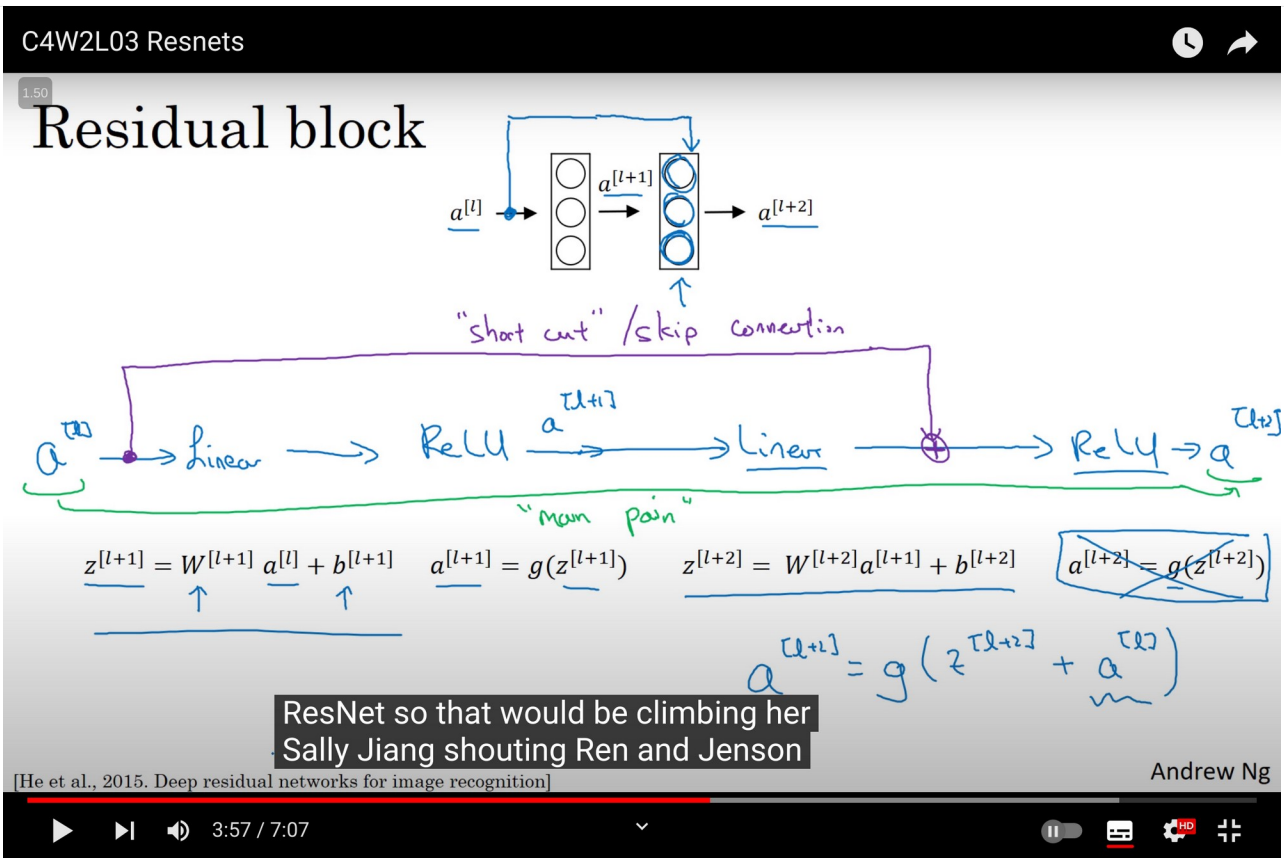
ResNet, short for Residual Network, introduced by Kaiming He et al. in their 2015 paper, revolutionized deep learning architectures by enabling the training of networks that were significantly deeper than those previously feasible. The key concept behind ResNet is the introduction of **residual learning**, which addresses the degradation problem—a phenomenon where the network accuracy saturates and then degrades rapidly as the network depth increases, not due to overfitting but due to the difficulty of optimizing very deep networks.

Key Concepts in ResNet

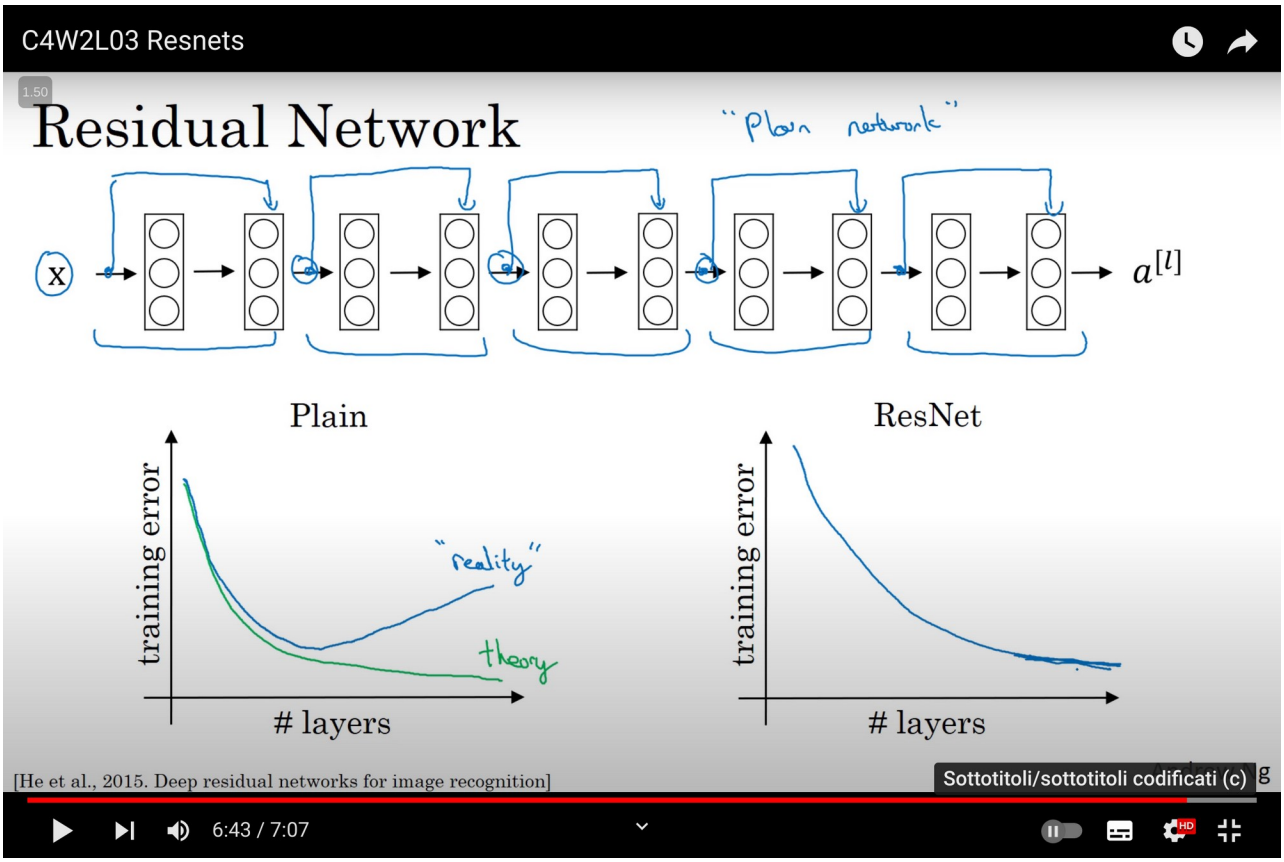
1. **Residual Blocks:** The core idea of ResNet is its use of residual blocks. Each block has a shortcut (or skip connection) that allows the input to bypass one or more layers and be added to the output of those layers. This design encourages the layers to learn residual functions (hence the name), referring to the difference between the input and the desired output. This approach simplifies the learning process, as learning the additive residual is often easier than learning the entire transformation.
2. **Ease of Optimization:** By using skip connections, gradients can flow directly through these connections during backpropagation, mitigating the vanishing gradient problem and making deeper networks easier to train.
3. **Deep Network Training:** ResNet demonstrated that networks could be significantly deepened (e.g., versions with 50, 101, 152 layers) with improvements in accuracy, thanks to the mitigation of the degradation problem.

Purpose of 1x1 Kernels with Stride 2

1. **Dimensionality Reduction and Expansion:** In some configurations within ResNet, 1x1 convolutions are used for reducing or increasing the dimensionality of the input feature maps, serving as a computationally efficient way to manage the number of feature maps between layers.
2. **Feature Re-calibration:** These 1x1 convolutions can adjust the channel-wise feature responses, effectively allowing the network to emphasize or de-emphasize certain feature maps, thus re-calibrating the features passed between layers.
3. **Strided Convolution for Downsampling:** Specifically, a kernel of size 1x1 with stride 2 is used for downsampling the input feature maps. This operation reduces the spatial dimensions (height and width) of the output by half, while allowing for the adjustment of the depth of the feature maps. This is crucial in deeper ResNet versions for reducing computational complexity and for building deeper models without a proportional increase in computational cost.


C4W2L03 Resnets
Residual block
$a^{[l]}$
$a^{[l+1]}$
$a^{[l+2]}$
"short cut" /skip connection
Linear
ReLU
Linear
ReLU
"main pain"
$z^{[l+1]} = W^{[l+1]} a^{[l]} + b^{[l+1]}$
$a^{[l+1]} = g(z^{[l+1]})$
$z^{[l+2]} = W^{[l+2]} a^{[l+1]} + b^{[l+2]}$
$a^{[l+2]} = g(z^{[l+2]})$
$a^{[l+2]} = g(z^{[l+2]} + a^{[l]})$
ResNet so that would be climbing her
Sally Jiang shouting Ren and Jenson
[He et al., 2015. Deep residual networks for image recognition]
Andrew Ng
3:57 / 7:07

Jk


C4W2L03 Resnets
Residual Network
"plain network"
x
$a^{[l]}$
Plain
ResNet
training error
layers
"reality"
theory
training error
layers
[He et al., 2015. Deep residual networks for image recognition]
Sottotitoli/sottotitoli codificati (c)
6:43 / 7:07

Q7

To calculate the number of parameters in a convolutional layer, you can use the formula:

$$\text{Number of Parameters} = (\text{Kernel Height} \times \text{Kernel Width} \times \text{Input Channels} + 1) \times \text{Number of Output Channels}$$

Here, the "+1" accounts for the bias parameter for each output channel. Given the input image has 3 channels (RGB), and the convolutional layer uses 5x5 kernels with a stride of 1, outputting 200 feature maps, we can plug these values into the formula:

- Kernel Height = 5
- Kernel Width = 5
- Input Channels = 3 (for RGB)
- Number of Output Channels (Feature Maps) = 200

Let's calculate the number of parameters.

The convolutional layer with 5x5 kernels, stride 1, outputting 200 feature maps of size 75x100, given an input image of dimensions 75 x 100 with 3 channels, has 15,200 parameters.

Q8

Reducing overfitting is crucial for improving the generalization of machine learning models to new, unseen data. Here are four techniques commonly used to combat overfitting:

1. **Regularization**: Regularization techniques add a penalty on the size of the coefficients to the loss function. The most common types are L1 (Lasso) and L2 (Ridge) regularization. L1 regularization can lead to sparse models where some coefficients can become zero, effectively selecting more important features. L2 regularization penalizes the square of the coefficients and tends to make coefficients smaller, leading to a more diffuse feature contribution. Both help prevent the model from fitting too closely to the training data.
2. **Data Augmentation**: This technique is particularly useful for deep learning models. By artificially increasing the size of the training dataset using information only in your training data, you can improve the robustness and generalization of the model. Common methods include cropping, padding, and flipping images in computer vision, or synonym replacement and sentence shuffling in natural language processing.
3. **Dropout**: Dropout is a regularization technique for neural networks that prevents overfitting by randomly setting a fraction of input units to 0 at each update during training time, which helps to prevent neurons from co-adapting too much. This means that the model learns to make accurate predictions with a partially present set of the features, enhancing its generalization capability.
4. **Early Stopping**: During training, the model's performance is continuously evaluated on a separate validation dataset not used for training. Training is stopped when the model's

performance on the validation set starts to deteriorate, indicating overfitting to the training set. This technique ensures that the model does not continue to learn the noise in the training dataset to the detriment of its performance on new data.

These techniques can be used individually or in combination to mitigate overfitting, depending on the specific circumstances and the model being developed.

Q9

Reinforcement Learning (RL) is a type of machine learning paradigm where an agent learns to make decisions by performing actions in an environment to achieve some goals. The learning process involves the agent interacting with the environment, receiving feedback in the form of rewards or penalties, and using this feedback to learn the best strategies or policies over time. Two key concerns addressed in Reinforcement Learning paradigms are:

1. **Exploration vs. Exploitation**: This is a fundamental dilemma in reinforcement learning. Exploration involves the agent trying out different actions to discover new knowledge about the environment, which could lead to better long-term rewards. Exploitation, on the other hand, involves leveraging the knowledge the agent has already acquired to maximize rewards in the short term. Balancing these two aspects is crucial because focusing too much on exploitation can cause the agent to miss out on potentially better strategies (local optima problem), while excessive exploration can lead to inefficiency and longer learning times. Developing strategies that effectively balance exploration and exploitation is a central concern in RL.
2. **Credit Assignment Problem**: Once an agent receives a reward or penalty, it must determine which actions were responsible for that outcome, especially in complex environments where the effects of actions are not immediately apparent. This problem becomes more challenging in situations with delayed rewards, where many actions may have been taken between the initial action and the eventual reward. The agent needs to figure out how to "credit" or "blame" specific actions for the outcomes achieved. This involves learning to predict the long-term consequences of actions, which is essential for developing effective policies.

Both of these concerns are critical for designing reinforcement learning algorithms that can learn efficiently and effectively from interaction with their environment, achieving optimal or near-optimal policies for decision-making tasks

q10

The image you've provided appears to be a graph showing the percentage of optimal actions taken by an agent in a reinforcement learning setting over a number of steps. There are three lines corresponding to different levels of exploration, indicated by the parameter epsilon (ε):

1. ε = 0.1 (blue line)

2. ε = 0.01 (red line)
3. ε = 0 (green line, which represents a greedy strategy with no exploration)

Here's what we can infer from the image for each strategy as the number of steps approaches infinity:

- For ε = 0.1 (blue line): The agent continues to explore new actions 10% of the time, which provides it with a steady rate of learning about the environment. As the number of steps goes to infinity, the agent's knowledge of the optimal actions improves, and we would expect the percentage of optimal actions to approach 100%, but perhaps not reach it entirely due to the continued exploration.
- For ε = 0.01 (red line): The agent explores new actions only 1% of the time. This less frequent exploration means it might take longer to learn the optimal actions compared to the ε = 0.1 case, but the line seems to be improving gradually. As the number of steps goes to infinity, this strategy may get closer to choosing the optimal action nearly all the time, potentially surpassing the ε = 0.1 strategy if it can avoid getting stuck in suboptimal actions.
- For ε = 0 (green line, greedy strategy): The agent does not explore at all and always exploits what it currently believes to be the best action. This could lead to the problem of getting stuck in local optima if the initial actions are not optimal. As the number of steps approaches infinity, the percentage of optimal actions will likely remain constant if the agent has not initially learned the optimal actions. If it started with correct knowledge, it would always take the optimal action, but this is generally not guaranteed in a non-stationary environment or one where the initial conditions do not encompass the optimal strategy.

In summary, as the number of steps approaches infinity:

- ε = 0.1 is expected to perform well, but with continued exploration, it may not reach 100% optimal actions.
- ε = 0.01 may converge to a higher percentage of optimal actions than ε = 0.1 if it learns effectively, due to less frequent exploration.
- ε = 0 (greedy) might not improve beyond a certain point if it doesn't start with optimal actions, as there is no mechanism for it to learn from new experiences.

It's important to note that these are trends based on the typical behavior of epsilon-greedy strategies in reinforcement learning. The actual performance will depend on the specific characteristics of the environment and how the rewards are structured.

OTHER EXAM

q1

The Physical Symbol System Hypothesis (PSSH) is a theory proposed by Allen Newell and Herbert A. Simon that posits a physical symbol system has the necessary and sufficient means for general

intelligent action. According to PSSH, the essential capabilities of human intelligence can be replicated in a machine, which operates by manipulating symbols to produce intelligent behavior. This manipulation involves the creation, modification, and destruction of symbols and the structures formed by symbols, which represent both the external environment and the system's internal states.

An example of a system based on the Physical Symbol System Hypothesis is the General Problem Solver (GPS), developed by Newell and Simon themselves. GPS was designed as a computer program capable of solving a broad range of problems through symbolic manipulation. It represented problems and solutions in a symbolic form and used heuristics and rules to navigate the problem space towards a solution, embodying the principles of PSSH by demonstrating how symbolic representations and manipulations can model aspects of human thought processes.

Q2 Write the code for a perceptron that uses threshold activation and has FIXED weights w1 = w2 =1,
and bias -1.5. Name the logical Boolean function modelled by this perceptron (1 mark).

The perceptron with fixed weights $w1=w2=1$, and bias -1.5 models the logical Boolean function AND. The output is only 1 when both inputs are 1, which matches the truth table of the AND function:

- For inputs (0, 0), the output is 0.
- For inputs (0, 1), the output is 0.
- For inputs (1, 0), the output is 0.
- For inputs (1, 1), the output is 1

q3 What is the update rule for a linear perceptron?

The update rule for a linear perceptron is used during the training process to adjust the weights and bias based on the perceptron's prediction error for a given training example. The goal is to minimize these errors over the training set, allowing the perceptron to correctly classify the inputs. The update rule is as follows:

$$wi(new)=wi(old)+\Delta wi$$
$$b(new)=b(old)+\Delta b$$

Where:

- $wi(new)$ and $b(new)$ are the updated weight and bias,
- $wi(old)$ and $b(old)$ are the current weight and bias,
- Δwi and Δb are the changes made to the weight and bias.

The changes Δwi and Δb are computed as:

$$\Delta wi=\eta(y-y\hat{})xi$$
$$\Delta b=\eta(y-y\hat{})$$

Here:

- η is the learning rate, a small positive value that determines the step size of the updates,
- y is the actual label of the training example,
- $\hat{y}$ is the predicted label by the perceptron,
- x_i is the input feature associated with weight w_i.

This rule is applied iteratively over all training examples, possibly multiple times (over epochs), until the perceptron's performance on the training set reaches a satisfactory level or until a maximum number of iterations is reached. The idea is that the perceptron learns to classify the training data by adjusting its weights and bias based on its errors, effectively drawing a linear decision boundary between classes for classification tasks.

Q4 Describe the 2 requirements that underpins RELUs? What does the acronym RELU stand for?

ReLU stands for Rectified Linear Unit, which is a type of activation function widely used in the field of deep learning and neural networks. The ReLU function is defined as $f(x)=\max(0,x)$, which means that if the input is positive, the output is equal to the input; if the input is negative, the output is zero. The two main requirements that underpin ReLUs are:

1. **Non-linearity**: Despite its piecewise linear form, the ReLU function introduces non-linearity into the model. This non-linearity is crucial for deep learning models, enabling them to learn complex patterns and representations from the data. Without non-linearity, a neural network, regardless of how many layers it has, would essentially function as a linear model, which limits its ability to capture and model complex relationships in the data.
2. **Differentiability (with an exception at zero)**: For optimization algorithms (like gradient descent) to work, the activation function needs to be differentiable at most points. The ReLU function is differentiable for all input values except at zero, where it is not differentiable. However, in practice, this exception does not pose a significant problem. The gradient for positive inputs is 1, which helps mitigate the vanishing gradient problem that can occur with other activation functions like the sigmoid or tanh. For negative inputs, the gradient is zero, which means that during backpropagation, weights that lead to negative inputs will not be updated, potentially leading to dead neurons. Despite this, ReLUs are widely used because they tend to work very well in practice and help models learn faster.

These characteristics make ReLUs a popular choice for many deep learning models, as they help speed up the training process and improve the model's ability to learn complex patterns while being computationally efficient.

Q5 Ioffe and Szegedy (2015) proposed Batch Normalisation as a mechanism to reduce the impact of
vanishing gradients. How many parameters in the three Batch Normalisation layers in Figure 1? Of these, how many are trainable? Please show the calculations (1 mark).

```
L1. model = keras.models.Sequential([
L2. keras.layers.Flatten(input_shape=[28, 28]),
```

L3. keras.layers.BatchNormalization(),
L4. keras.layers.Dense(300, activation="relu", kernel_initializer="he_normal"),
L5. keras.layers.BatchNormalization(),
L6. keras.layers.Dense(100, activation="relu", kernel_initializer="he_normal"),
L7. keras.layers.BatchNormalization(),
L8. keras.layers.Dense(10, activation="softmax")])

Batch Normalization (BN) layers have two sets of parameters:

- **γ** (scale factors) and
- **β** (offset/shift factors),

which are trainable, and two sets of parameters:

- **mean** and
- **variance**,

which are not trainable (computed during training but fixed during inference). For each feature, there is one γ and one β, making the total number of parameters in a BN layer twice the number of features input into it.

Let's calculate the parameters for each BatchNormalization layer based on the network architecture provided:

1. **First BatchNormalization Layer**: After the `Flatten` layer, the input shape to the first BN layer is [28, 28], which gets flattened to 784 features. Thus, this layer has:
    - Trainable: 784×2=1568 parameters (γ and β)
    - Non-trainable: 784×2=1568 parameters (mean and variance)
2. **Second BatchNormalization Layer**: This layer follows a Dense layer with 300 units. Therefore, it normalizes 300 features, resulting in:
    - Trainable: 300×2=600 parameters (γ and β)
    - Non-trainable: 300×2=600 parameters (mean and variance)
3. **Third BatchNormalization Layer**: This layer is after a Dense layer with 100 units, normalizing 100 features. It has:
    - Trainable: 100×2=200 parameters (γ and β)
    - Non-trainable: 100×2=200 parameters (mean and variance)

Now, let's sum up these values to find the total number of parameters and the number of trainable parameters in all three BatchNormalization layers:

- **Total Trainable Parameters**: 1568+600+200=2368
- **Total Non-Trainable Parameters**: 1568+600+200=2368

Therefore, across the three BatchNormalization layers, there are a total of 4736 parameters, of which 2368 are trainable

q6 How many parameters in the first convolutional layer of a LeNet5 CNN where the input is 32 x 32 x 1, the kernel is 5 x 5 and there are six such filters, and zero padding and stride 1. Please clearly show the calculations

For a convolutional layer, the number of parameters can be calculated using the formula:

$$\text{Number of parameters} = (\text{kernel height} \times \text{kernel width} \times \text{input depth} + 1) \times \text{number of filters}$$

Here, the "+1" accounts for the bias term associated with each filter.

Given the parameters for the first convolutional layer in a LeNet-5 CNN architecture:

- Input dimensions: $32 \times 32 \times 1$ (width x height x depth)
- Kernel size: 5×5
- Number of filters: 6
- Since there's zero padding and stride of 1, these factors do not directly affect the number of parameters, but they influence the output size of the layer.

Let's plug in the values:

$$\text{Number of parameters} = (5 \times 5 \times 1 + 1) \times 6$$

Calculating this gives us the total number of parameters.

The first convolutional layer of a LeNet-5 CNN, with the specified configuration, has a total of 156 parameters.

Q7 Draw a diagram for a Naive Inception module. What is the issue with this design?

The Naive Inception module is designed with the idea of allowing the network to adapt to various scales and sizes of input data by processing the input simultaneously through multiple convolutional paths of different sizes (1x1, 3x3, 5x5 convolutions) and a max pooling path. These paths operate in parallel, and their outputs are concatenated along the depth dimension to form the final output.

The issue with this design is its computational cost and inefficiency. The use of larger convolutions (especially the 5x5) involves a significant number of parameters and computations. This can make the network computationally expensive to train and can lead to overfitting due to the large number of parameters. Moreover, directly applying convolutions and pooling on the same scale of input can be redundant and waste computational resources, especially if the input features do not require such diverse processing. To mitigate these issues, modifications such as the use of 1x1 convolutions

before 3x3 and 5x5 convolutions are introduced in more advanced versions of the Inception module to reduce dimensionality and computational cost.

Q8

The code snippet provided is a part of a Python class that defines a custom layer, presumably for a ResNet-like convolutional neural network architecture, using the Keras deep learning library. The intent and purpose of the code in the specified lines are as follows:

(a) Lines 13-16: These lines define a list named `self.skip_layers` that is used when the `strides` parameter is greater than 1. The purpose of `self.skip_layers` is to create a shortcut connection that matches the dimensions of the output from the main layers when the input is downsampled (strides > 1). This is necessary because in ResNet, when the dimensions of the input and the output do not match (due to strides in the convolution), a skip connection with a convolutional layer of `kernel_size=1` and appropriate strides is added to resize the input before adding it to the output of the main layers. Batch normalization is also included in this skip path to ensure the input is properly normalized before being added to the main path.

(b) Lines 30-34: These lines are part of a loop that adds `ResidualUnit` layers to the Sequential model. `ResidualUnit` is likely a custom layer defined earlier (as seen in lines 2-24) that implements the functionality of a ResNet block. The loop constructs the deep layers of the network with increasing numbers of filters, doubling the number each time. The variable `strides` is determined by whether the number of filters is the same as the previous number of filters (`prev_filters`). If they are the same, `strides` is set to 1, meaning no downsampling occurs. If the number of filters has increased from the previous set, `strides` is set to 2, which will downsample the input. This is typical of ResNet architectures where downsampling is done at the same time as the increase in the number of filters to reduce the spatial dimensions of the feature maps while increasing their depth.

Overall, these lines are constructing a neural network with residual connections, which helps to mitigate the vanishing gradient problem by allowing gradients to flow through the skip connections. It also enables the training of very deep networks by addressing the degradation problem (where accuracy saturates and then degrades rapidly with increasing network depth).

Q9

q10

Reinforcement Learning (RL) has been applied to a variety of domains beyond the ones commonly showcased in lectures such as games and robotics. Here are three examples:

1. **Energy Systems Management**: RL can be utilized to optimize energy consumption in smart grids. For instance, algorithms can learn to balance the load between different energy sources, minimize costs, and optimize the energy flow based on demand and supply. It can also be used for efficient energy storage management, deciding when to store energy, when to use it from the battery, and when to buy from the grid, based on the pricing and demand forecasts.
2. **Personalized Recommendations**: In the domain of e-commerce or content platforms, RL can help to create dynamic recommendation systems that adapt to user behavior. Unlike traditional recommendation engines that use static algorithms, an RL-based system can continuously learn from user interactions to improve the relevance of the items or content it suggests. This can enhance user engagement by presenting more personalized choices.
3. **Healthcare Treatment Optimization**: RL can be used to develop treatment strategies for chronic diseases such as diabetes or hypertension, where the treatment regime can be complex and highly individualized. An RL agent can learn optimal treatment policies for dosing of medication or insulin, considering the patient's unique response to treatment, with the goal of keeping the patient's biomarkers, like blood sugar levels, within a healthy range over time.

These examples demonstrate the versatility of RL in solving complex, dynamic problems across various sectors by learning to make a sequence of decisions that lead to a desired outcome.

CLASS QUESTIONS

definition of machine learning:

Machine learning (ML) is a discipline of artificial intelligence (AI) that provides machines with the ability to automatically learn from data and past experiences while identifying patterns to make predictions with minimal human intervention or without being programmed to do it

inner product vs euclidian distance, which is better to measure similarity

At its core, Similarity learning is about determining how alike or different two data points are. Imagine you have two photos, and you want to know if they are of the same person. Instead of looking at every pixel, Similarity Learning algorithms will identify key features (like the shape of the eyes or the curve of the mouth) and compare them.

The choice between using the inner product and Euclidean distance to measure similarity depends on the context of your application and the specific characteristics of your data. Both metrics have their unique advantages and use cases:

Inner Product

- **Definition:** The inner product (or dot product) measures the cosine of the angle between two vectors in a multi-dimensional space. It can be positive (indicating a smaller angle), zero (orthogonal vectors), or negative (larger than 90 degrees angle).

- **Use Cases:** It is often used in contexts where the orientation of the vectors relative to each other is more important than their magnitude. For example, in text analysis and information retrieval, the inner product (or cosine similarity, derived from the inner product) is used to measure the similarity between documents represented as vectors in a high-dimensional space.
- **Advantages:**
    - It is not affected by the magnitude of the vectors, focusing instead on their direction. This makes it useful for comparing documents or items with different lengths or sizes.
    - It can identify similarities in high-dimensional spaces effectively.

Euclidean Distance

- **Definition:** Euclidean distance measures the "straight line" distance between two points in Euclidean space. It is the square root of the sum of the squared differences between the corresponding elements of the vectors.
- **Use Cases:** It is widely used in clustering algorithms and spatial data analysis, where the actual distance between points is important. For example, in K-means clustering, Euclidean distance can determine the similarity between data points by their physical closeness.
- **Advantages:**
    - It reflects the actual distance between two points, which is intuitive and straightforward to understand.
    - Suitable for applications where the scale and magnitude of the data are important, such as in physical measurements.

Which Is Better?

- **Depends on Data and Context:** If the direction (angle) of the data vectors matters more than their magnitude (e.g., in comparing text documents), the inner product or cosine similarity may be more appropriate. On the other hand, if the actual spatial distance between data points is relevant (e.g., in geographic data analysis), Euclidean distance might be better.
- **Normalization Considerations:** Euclidean distance can be significantly affected by the scale of the data, so normalization might be necessary to make fair comparisons between vectors of different magnitudes.
- **Dimensionality:** For high-dimensional data, the inner product might be more informative due to the curse of dimensionality affecting Euclidean distance, making many distances appear similar.

threshold activation function problem: not continuous

The non-continuity threshold problem in neural networks (NNs) generally refers to issues arising from the use of discrete thresholding in activation functions or decision boundaries, which can lead to non-continuous or non-smooth behavior in an otherwise smooth prediction landscape. This topic touches upon several key aspects of neural network design and operation, such as activation functions, loss functions, and the optimization landscape. Let's delve into these areas:

Activation Functions

Early neural networks often used step functions as activation functions, which introduce a clear non-continuity threshold problem. A step function outputs a binary result based on whether the input is less than or greater than some threshold. This binary switch from one state to another on a minuscule change in input can lead to non-continuous behavior in the network's output.

However, modern neural networks predominantly use smooth, continuous activation functions like the sigmoid, tanh, and ReLU (Rectified Linear Unit) functions. These functions help mitigate the non-continuity threshold problem by providing a continuous mapping from inputs to outputs. Even the ReLU function, which is not differentiable at zero, is considered continuous and allows for gradient-based optimization techniques to work effectively.

purpose of bias in perceptron:

Purpose of Bias

- **Offset the Decision Boundary:** Without the bias term, the decision boundary of the perceptron (where it decides between the two classes) would always have to pass through the origin (0,0) in the feature space. This is a significant limitation because it assumes the data is linearly separable in such a way that this specific boundary is sufficient. The bias term allows the decision boundary to be offset from the origin, which means the perceptron can learn and represent a wider variety of decision boundaries, improving its ability to classify data accurately.
- **Improves Model Flexibility:** The bias term increases the model's flexibility, enabling it to fit the training data more effectively. By adjusting the bias, the perceptron can shift the linear decision boundary to better separate the classes in the feature space.
- **Handles Situations Where Input Features Are Zero:** In cases where all input features are zero, the output of the perceptron would solely depend on the weight values if there was no bias. In such scenarios, the bias ensures that the perceptron can still produce a non-zero output, allowing for a meaningful decision even when input features do not contribute to the decision directly

write the code for a perceptron:

how to measure accuracy:

The formula for accuracy is:

$$\text{Accuracy} = \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}}$$

In more detailed terms:

$$\text{Accuracy} = \frac{TP+TN}{TP+TN+FP+FN}$$

hidden layer what's the error?

Approximate the error

who are the author the back propagation is attributed to?

geoffrey hinton
resined from google

what cause a dying relus? -> leaky relu

What is Dying ReLU?

The term "Dying ReLU" occurs when a ReLU neuron gets stuck in a state where it only outputs zero for any input. Once a ReLU neuron enters this state, it stops participating in the discriminative process of the neural network because the gradient through the unit is zero for all inputs. As a consequence, during the backpropagation process, no gradient flows through the neuron, meaning it does not update its weights anymore and essentially becomes inactive, or "dies." This can lead to a significant portion of the network not contributing to the learning process, thereby reducing the model's capacity to fit the training data.

Causes of Dying ReLU

1. **Large Learning Rates:** One common cause of the Dying ReLU problem is the use of too large learning rates. If the weight updates are too large, the weights could be updated in such a way that the weighted sum of the inputs to a ReLU neuron is always negative, leading to the neuron outputting zero for all inputs.
2. **Poor Initialization:** Poor initialization of weights can also contribute to this problem. If the initial weights are such that the input to a ReLU neuron is negative for all inputs in the training set, the neuron will start off dead.

3. **Sparse Gradients:** In networks with a deep architecture or insufficient training data, some neurons might not receive enough gradient signal to adjust their weights away from the dead zone.

Solutions and Alternatives to Dying ReLU

- **Leaky ReLU:** This is a popular alternative to the ReLU function that allows a small, non-zero gradient when the unit is not active and the input is less than zero. It is defined as $f(x)=\max(\alpha x,x)$, where α is a small constant. This small gradient keeps the neuron alive and allows for the backpropagation of errors, potentially mitigating the dying ReLU problem.
- **Parametric ReLU (PReLU):** This is a variant of Leaky ReLU where α is learned during training. This adaptability can help the network self-adjust to mitigate the dying ReLU problem.
- **ELU (Exponential Linear Unit) or SELU (Scaled Exponential Linear Unit):** These activation functions have all the benefits of ReLU but improve on it by taking negative values when the input is below zero, which helps alleviate the dying ReLU issue.
- **Proper Initialization and Careful Learning Rate Setting:** Ensuring that the weights are initialized correctly and that the learning rate is set appropriately can prevent neurons from dying.

categorical cross entropy difference with cross entropy:

Cross entropy is a measure used in machine learning and information theory to quantify the difference between two probability distributions. It's often used as a loss function (cross-entropy loss) in classification problems. The concept of cross entropy is foundational, while categorical cross-entropy is a specific application of cross entropy. Let's explore the differences and applications of each:

Cross Entropy

- **General Concept:** Cross entropy measures the dissimilarity between two probability distributions over the same set of events. It's defined for two distributions, P (the true distribution) and Q (the estimated distribution), as follows:

$$H(P,Q)=-\sum xP(x)\log(Q(x))$$

where $P(x)$ is the probability of event x in the true distribution, and $Q(x)$ is the probability of event x in the estimated distribution.

- **Application:** Cross entropy is used in various machine learning tasks, not just classification. It's applicable wherever it's necessary to compare how well a predicted probability distribution matches the true distribution.

Categorical Cross Entropy

- **Specific Case:** Categorical cross-entropy is a specific form of cross entropy used in multi-class classification problems. It's used when there are two or more class labels to predict, and the labels are mutually exclusive. For each instance, the true label is represented as a one-hot encoded vector, and the model provides a probability distribution over all possible classes.
- **Formula:** For a single instance with N classes, where the true class for this instance is one-hot encoded and p_i is the model's predicted probability for class i, categorical cross-entropy is calculated as:

$$H(P,Q) = -\sum_{i=1}^{N} y_i \log(p_i)$$

Here, y_i is 1 for the true class and 0 for all other classes, reducing the sum to the logarithm of the predicted probability for the true class.

Key Differences

- **Scope:** Cross entropy is a broader concept applicable to any pair of probability distributions. Categorical cross-entropy is a specific application of cross entropy for evaluating the performance of multi-class classification models.
- **Use Case:** While cross entropy can be applied in various contexts, including binary classification, sequence generation, and more, categorical cross-entropy is tailored for situations where you are classifying instances into one of several mutually exclusive classes.
- **Representation:** Categorical cross-entropy specifically deals with probability distributions that are represented in a one-hot encoded format for the true labels, contrasting with the more general probability comparisons in basic cross entropy.

Binary Cross Entropy

It's worth mentioning binary cross-entropy as well, which is a special case of cross entropy used in binary classification tasks. It's essentially categorical cross-entropy applied to a scenario with only two classes.

Which is the preferred loss function for each model?

1. Linear Regression

- **Mean Squared Error (MSE):** It's the most common loss function for regression tasks, calculating the square of the difference between the predicted and actual values. It's preferred because it penalizes larger errors more severely, leading to a model that can better minimize large errors.

- **Mean Absolute Error (MAE):** Sometimes preferred for regression tasks, especially if you are dealing with outliers that can skew the model too much with MSE.

2. Logistic Regression

- **Binary Cross-Entropy (Log Loss):** For binary classification tasks, binary cross-entropy measures the distance between the probability distribution of the output and the actual distribution. It's preferred for logistic regression, a model inherently designed for binary classification.

3. Neural Networks

- **Binary Classification:** Binary Cross-Entropy is again the preferred choice for binary classification tasks due to its effectiveness in measuring the performance of a classification model whose output is a probability value between 0 and 1.
- **Multi-class Classification:** Categorical Cross-Entropy is used when there are more than two classes to predict. It generalizes binary cross-entropy over multiple classes.
- **Regression Tasks:** Mean Squared Error (MSE) or Mean Absolute Error (MAE), depending on the presence of outliers and the importance of penalizing large errors more heavily.
- **For Generative Models (like GANs):** The Wasserstein loss or the KL-Divergence might be used, depending on the specific characteristics of the model and the training stability requirements.

Define linear and logistic regression:

Logistic regression and linear regression are two fundamental statistical methods used for prediction and analysis in machine learning. While both are used to predict the outcome based on one or more independent variables, they serve different purposes and are used in different types of prediction problems.

Linear Regression

Definition: Linear regression is a linear approach to modelling the relationship between a dependent variable and one or more independent variables. The goal is to find a linear equation that best predicts the dependent variable.

Formula: The linear equation can be represented as: $y=\beta0+\beta1x1+\beta2x2+...+\beta nxn+\epsilon$ where:

- y is the dependent variable.
- $x1,x2,...,xn$ are the independent variables.
- $\beta0$ is the y-intercept of the regression line.

- $\beta 1, \beta 2, ..., \beta n$ are the coefficients of the independent variables which represent the slope of the regression line. They indicate the amount by which the dependent variable is expected to change with a one-unit change in the respective independent variable.
- ϵ is the error term, representing the difference between the observed and predicted values.

Use Cases: It is used for prediction of continuous outcomes, such as predicting house prices, stock prices, temperature forecasts, etc.

Logistic Regression

Definition: Logistic regression is a statistical method for analyzing a dataset in which there are one or more independent variables that determine an outcome. The outcome is a binary variable (i.e., it only contains data coded as 1 (yes, success, etc.) or 0 (no, failure, etc.)). Logistic regression predicts the probability of the binary outcome.

Formula: The logistic function can be represented as: $p = 1 + e^{-(\beta 0 + \beta 1 x1 + \beta 2 x2 + ... + \beta nxn)}1$ where:

- p is the probability of the dependent variable equaling a "success" or "1".
- $x1, x2, ..., xn$ are the independent variables.
- $\beta 0, \beta 1, \beta 2, ..., \beta n$ are the coefficients of the independent variables, estimated from the training data.
- e is the base of the natural logarithm.

The logistic function ensures that the output value (the probability) ranges between 0 and 1, making it suitable for estimating probabilities.

Use Cases: It is used for binary classification tasks, such as spam detection (spam or not spam), disease diagnosis (sick or not sick), customer churn prediction (will churn or will not churn), etc.

Key Differences

- **Outcome Variable:** Linear regression is used when the outcome variable is continuous and unbounded. Logistic regression is used when the outcome variable is binary (0 or 1) or categorical.
- **Function:** Linear regression involves predicting a quantitative response. Logistic regression involves predicting the probability of a categorical outcome.
- **Equation Form:** Linear regression predictions are made through a linear equation, while logistic regression uses a logistic function (sigmoid curve) to model the data.

GANs

Generative Adversarial Networks (GANs) are a class of artificial intelligence algorithms used in unsupervised machine learning, introduced by Ian Goodfellow and his colleagues in 2014. They are composed of two neural networks, termed the generator and the discriminator, which are trained simultaneously through a competitive process:

- **Generator:** This network learns to generate data. The goal of the generator is to produce data that is indistinguishable from real data. It takes random noise as input and generates samples as output.
- **Discriminator:** This network is a classifier that learns to distinguish between real and generated (fake) data. It takes samples from both the real data and the generated data produced by the generator as input and attempts to classify them as real or fake.

How GANs Work

The training process of GANs involves an adversarial game, where the generator and the discriminator compete against each other. The generator tries to produce fake data that is similar to the real data, while the discriminator tries to get better at distinguishing the fake data from the real data. This process can be summarized as follows:

1. The generator creates samples from random noise.
2. The discriminator evaluates samples from both the generator and the actual dataset and tries to distinguish real from fake.
3. The generator is trained to maximize the probability of the discriminator making a mistake. This is typically achieved by using the gradient of the discriminator's decisions and applying it to improve the generator.
4. This process continues until the generator produces samples that the discriminator can no longer distinguish from real data, indicating that the generator has learned the distribution of the real data.

calculate non-trainable parameters from code

what is the purpose of 1x1 convolutional layer?

The 1x1 convolutional layer, also known as a pointwise convolution, serves several important purposes in convolutional neural networks (CNNs). Despite its simplicity, it is a powerful tool for managing and enhancing the network architecture. Here are the main purposes of using 1x1 convolutions:

1. Channel-wise Feature Pooling and Combination

The 1x1 convolution can combine or pool features across the depth of the input volume. This means it can take an input with a certain number of channels (depth) and produce an output with a different number of channels. During this process, it combines features from different channels, allowing the network to learn more complex features that are combinations of the initial input features. This is especially useful in deep CNNs where the depth of the feature maps increases with each layer.

2. Dimensionality Reduction

A 1x1 convolution can be used to reduce the number of channels in the input volume, which helps in reducing the computational load for the network. For example, if the input volume to a 1x1

convolutional layer has 256 channels, using a 1x1 convolution with 64 output channels effectively reduces the depth of the feature map by a factor of 4. This reduction is achieved without losing the spatial dimensions of the input. It's a very efficient way to decrease the computational cost and the number of parameters in the network, especially before expensive operations like 3x3 or 5x5 convolutions.

3. Increasing Non-linearity

When a 1x1 convolutional layer is followed by a non-linear activation function (like ReLU), it introduces an additional non-linearity to the model without affecting the spatial dimensions of the feature maps. This can help the network learn more complex patterns without a significant increase in computational cost.

4. Multi-channel Feature Learning

The 1x1 convolution allows the network to learn from all the channels of the input volume simultaneously, unlike larger convolutions (like 3x3 or 5x5) that process each input channel separately and then combine them. This can lead to more efficient feature learning from multi-channel data.

5. Implementing Network-in-Network Architectures

1x1 convolutions are a key component in "network-in-network" architectures. These architectures replace traditional linear filters with mini neural networks that perform more complex transformations on the input data. The 1x1 convolutions act as fully connected layers applied to each pixel location, allowing for more complex decision functions than what could be achieved with linear filters alone.

6. Efficient Implementation of Cross-channel Parameterization

By applying 1x1 convolutions, networks can efficiently implement cross-channel parameterization, enabling different channels to interact and combine in learnable ways. This can enhance the representational power of the network without a significant increase in computational complexity.

methods to improve generalization?

1. Data Augmentation

- **Description:** Increasing the diversity of the training set by artificially creating modified versions of the training data through transformations like rotation, scaling, flipping, or adding noise.
- **Applicability:** Particularly effective in image and audio processing tasks.

2. Regularization

- **Description:** Techniques that constrain the model's learning capacity to prevent overfitting. Common methods include L1 and L2 regularization, which add a penalty on the size of the coefficients.

- **Applicability:** Useful for linear models, neural networks, and any model with a large number of parameters.

3. Dropout

- **Description:** A regularization technique for neural networks that randomly ignores a subset of neurons during training, forcing the network to learn more robust features.
- **Applicability:** Primarily used in deep learning.

4. Early Stopping

- **Description:** Monitoring the model's performance on a validation set and stopping training when performance begins to degrade, preventing overfitting.
- **Applicability:** Can be applied to any iterative training algorithm, such as gradient descent used in neural networks.

challenge of balance exploration and exploitation

The challenge of exploration and exploitation is a fundamental dilemma in reinforcement learning (RL). It represents the decision-making problem an agent faces when it must choose between:

- **Exploration:** Discovering new knowledge about the environment. This involves taking actions that the agent has not yet tried or about which it has insufficient information, to learn more about the environment and find potentially better rewards in the long run.
- **Exploitation:** Utilizing known information to maximize rewards. This means choosing actions that the agent already knows to be rewarding based on its current knowledge, aiming to maximize its returns in the short term.

The Challenge

The core challenge lies in balancing these two aspects:

- If an agent **explores** too much, it risks losing out on immediate rewards from known good actions, potentially wasting time and resources on less optimal choices.
- If an agent **exploits** too much, it may settle too quickly on a suboptimal policy without discovering other actions that could lead to higher rewards, thus missing out on the long-term benefits.

Finding the right balance is crucial for effectively learning optimal policies in diverse and complex environments.

Strategies to Address the Challenge

Various strategies have been developed to navigate the exploration-exploitation dilemma:

1. **Epsilon-Greedy Strategy:** A simple yet effective method where the agent explores with a probability of ϵ and exploits with a probability of $1-\epsilon$. The value of ϵ can be kept constant or decay over time to shift from exploration to exploitation as learning progresses.
2. **Upper Confidence Bound (UCB):** A strategy that chooses actions based on the upper confidence bounds of action values, balancing between exploiting actions with high rewards and exploring actions with uncertain rewards. It's more sophisticated than epsilon-greedy and tries to quantify the uncertainty or variance in the reward of each action.
3. **Thompson Sampling:** A probabilistic method that models the uncertainty of the action-reward relationship with a probability distribution. Actions are chosen based on samples from these distributions, naturally balancing exploration and exploitation by considering both the mean reward and the uncertainty.

credit assignment problem

Credit Assignment Problem

The **credit assignment problem** in reinforcement learning refers to the difficulty of determining which actions are responsible for receiving a particular reward, especially when rewards are delayed. This is a fundamental issue in environments where the consequences of an action are not immediate but unfold over time.

The Problem: In many RL scenarios, an agent takes a series of actions before it receives a significant reward (or penalty). Determining which actions in the sequence contributed to the outcome, and to what extent, is challenging. This is because the effect of an action on the environment and future rewards can be complex and delayed, making it difficult to trace back the reward to specific actions.

Addressing the Problem: Various RL algorithms address the credit assignment problem through different mechanisms. Temporal Difference (TD) learning and Monte Carlo methods estimate the value of states or actions by considering future rewards, using techniques to update the value estimates based on the rewards received at later time steps. Algorithms like Q-learning and SARSA specifically focus on learning action-value functions that help in attributing credit to actions by updating their value estimates based on the observed outcomes.

what is the markov property? (week 5 slide 9)

It describes a system where the future state depends only on the current state and not on the sequence of events that preceded it

Formal Definition

A process has the Markov property if the conditional probability distribution of future states of the process (conditional on both past and present states) depends only upon the present state, not on the sequence of events that preceded it

be able to explain the plots on RL slides

monte carlo search

Are filters spatially agnostic?

Filters in Convolutional Neural Networks

CNNs use filters (or kernels) to perform convolution operations on input data, typically images. These filters are designed to detect patterns such as edges, textures, or more complex features in higher layers of the network. Here's how these concepts relate to the idea of being "spatially agnostic":

- **Spatial Awareness:** Filters in CNNs are not spatially agnostic; in fact, they are specifically designed to capture spatial relationships within the input data. Each filter slides (or convolves) across the input image to apply the same operation to each patch of the image, effectively looking for the same pattern throughout the entire input space. This sliding mechanism ensures that the CNN can recognize patterns regardless of their position in the input image, which is a property known as translation invariance.
- **Learning Spatial Features:** Through the training process, filters in CNNs automatically learn to identify spatial features that are most relevant for the task at hand (e.g., classifying images). Early layers might learn to detect simple edges and textures, while deeper layers can identify more complex patterns by combining features detected in earlier layers.

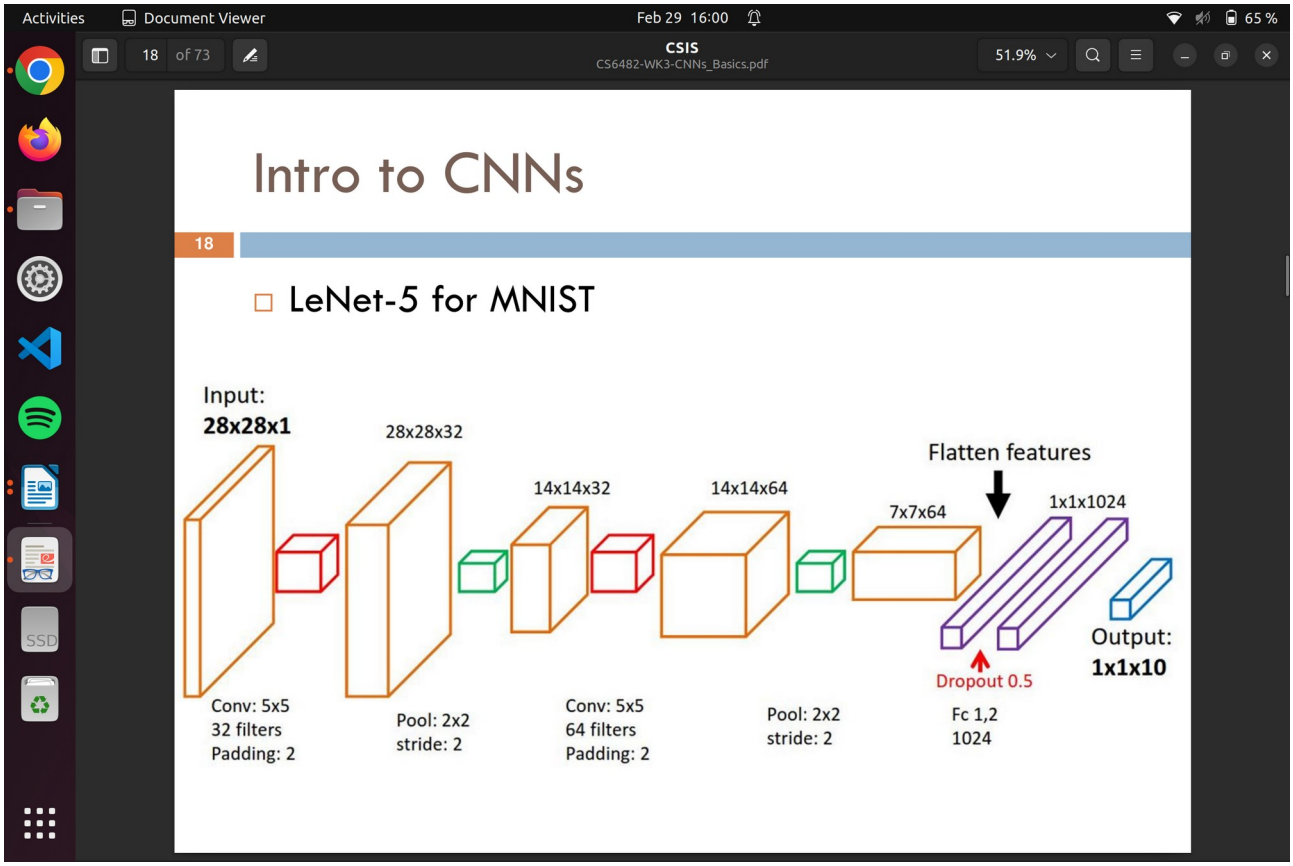
Spatial Invariance vs. Spatial Agnosticism

- **Spatial Invariance:** CNNs exhibit spatial invariance to a degree, meaning they can recognize patterns regardless of their location in the input image. This is a crucial property for tasks like image classification, where the subject of an image can appear in different areas of the image frame.
- **Not Spatially Agnostic:** Saying that filters are spatially agnostic would imply that they do not process spatial information or the arrangement of features within the data, which is not accurate. CNN filters are deeply concerned with the spatial arrangement of pixels in their receptive fields.

Conclusion

In summary, filters in neural networks, particularly in CNNs, are not spatially agnostic; they are fundamentally designed to be sensitive to spatial arrangements of features within the input data. This sensitivity to spatial information is what allows CNNs to be so effective in tasks that require understanding of visual patterns and structures, such as image and video recognition, object detection, and more. The spatial invariance property of CNNs, enabled by the use of shared weights and pooling layers, allows these networks to detect patterns regardless of their exact location in the input space, but this is a result of how the filters are applied, not because the filters themselves are spatially agnostic.

Lenet-5



batch normalization:

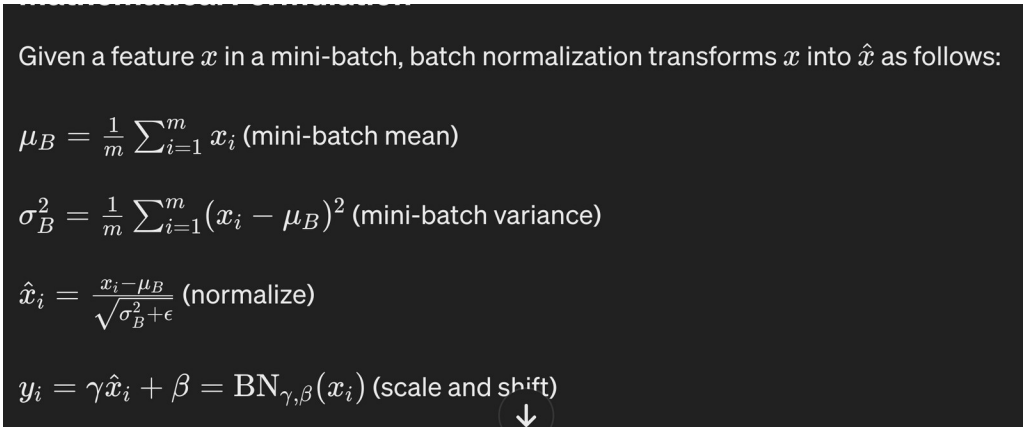
Batch Normalization is a technique used in deep learning to improve the stability, performance, and speed of neural networks. It was introduced by Sergey Ioffe and Christian Szegedy in 2015. The primary goal of batch normalization is to address the issue of internal covariate shift, which occurs when the distribution of each layer's inputs changes during training, as the parameters of the previous layers change. This can make the training process slow and unstable, requiring careful initialization of parameters and smaller learning rates.

How Batch Normalization Works

Batch normalization is applied to individual layers within a network. It normalizes the inputs to a layer for each mini-batch, meaning it adjusts and scales the inputs so that they have a mean of zero and a standard deviation of one. This normalization step helps to keep the data flowing between layers of the network in a more standardized state, reducing the internal covariate shift problem.

The process involves the following steps for each mini-batch:

1. **Calculate Mean and Variance:** Compute the mean and variance of the inputs for the mini-batch.
2. **Normalize:** Normalize the inputs using the computed mean and variance.
3. **Scale and Shift:** After normalization, the outputs are then scaled and shifted using two parameters (γ and β) that are learned during the training process. This step is crucial because the best distribution for the inputs of a layer may not necessarily be a standard normal distribution. The scaling and shifting allow the network to adjust the normalization process to best suit the training.



- m is the number of examples in the mini-batch.
- μB is the mini-batch mean.
- σB2 is the mini-batch variance.
- ϵ is a small constant added for numerical stability.
- γ and β are parameters to be learned.
- yi is the output of the BN layer.

Benefits of Batch Normalization

- **Improved Training Speed:** By normalizing the inputs across mini-batches, batch normalization allows for the use of higher learning rates, which can make the training process faster.

- **Reduces the Need for Careful Initialization:** Since batch normalization regulates the inputs to layers within a certain range, it reduces the dependency on having to initialize the weights of the network in a precise manner.
- **Can Act as Regularization:** Batch normalization can add a slight noise to the activations within the network, which can help to regularize the model. However, it might reduce the need for dropout.