Introduction to Machine Learning (25737-2)
Problem Set 03

Instructor: Dr. R. Amiri

Spring Semester 1402-03

Ali Nikkhah

Github

1 Design a Neural Network

Design a neural network capable of generating Hamming codes for 4-bit inputs. Describe the network, including the number of neurons in each layer, and determine all the weights in the network. (Note that the neural net should have 4 inputs and 3 outputs.)

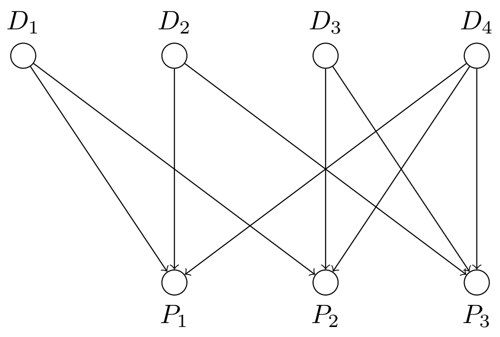
Input Layer:	4 neurons (D_1, D_2, D_3, D_4)
Output Layer:	3 neurons (P_1, P_2, P_3)

$$P_1 = D_1 \oplus D_2 \oplus D_4$$
$$P_2 = D_1 \oplus D_3 \oplus D_4$$
$$P_3 = D_2 \oplus D_3 \oplus D_4$$

$$W = \begin{bmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 0 & 1 & 1 & 1 \end{bmatrix}, \quad b = \begin{bmatrix} -0.5 \\ -0.5 \\ -0.5 \end{bmatrix}$$

$$P = \text{step}(WX + b)$$

$$\text{step}(x) = \begin{cases} 1 & \text{if } x \geq 0.5 \\ 0 & \text{if } x < 0.5 \end{cases}$$



2 Design Simple Neural Network (Optional)

- **Design a neural network with one hidden layer that implements the following function:**

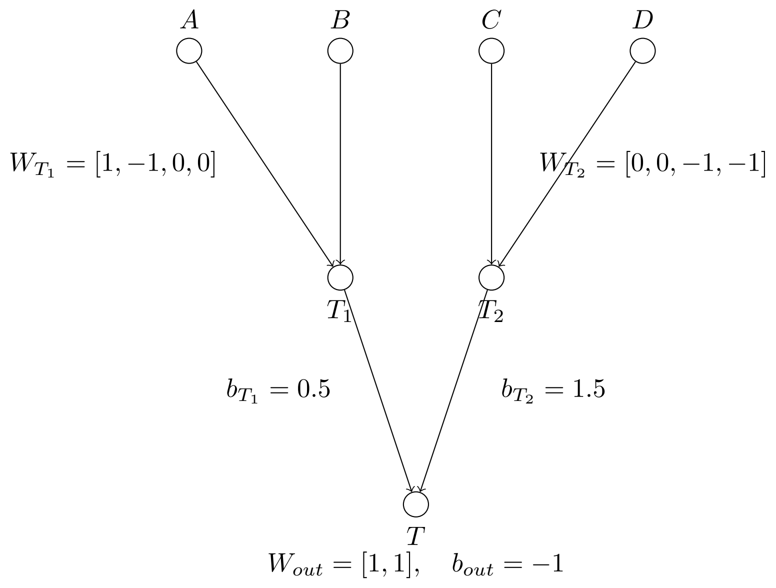
$$(A \vee \bar{B}) \oplus (\bar{C} \vee \bar{D})$$

- **Draw the network and determine all its weights.**

Input Layer:	4 neurons (A, B, C, D)
Hidden Layer:	2 neurons (T_1, T_2)
Output Layer:	1 neuron (T)

$$\begin{aligned} T_1 &= \text{step}(A - B + 0.5) & W_{T_1} &= [1, -1, 0, 0], & b_{T_1} &= 0.5 \\ T_2 &= \text{step}(-C - D + 1.5) & W_{T_2} &= [0, 0, -1, -1], & b_{T_2} &= 1.5 \\ T &= \text{step}(T_1 + T_2 - 1) & W_{out} &= [1, 1], & b_{out} &= -1 \end{aligned}$$

Layer	Weights and Biases
Hidden Layer	$W_{T_1} = [1, -1, 0, 0]$ $b_{T_1} = 0.5$ $W_{T_2} = [0, 0, -1, -1]$ $b_{T_2} = 1.5$
Output Layer	$W_{out} = [1, 1]$ $b_{out} = -1$



A
B
C
D
$W_{T_1} = [1, -1, 0, 0]$
$W_{T_2} = [0, 0, -1, -1]$
T_1
T_2
$b_{T_1} = 0.5$
$b_{T_2} = 1.5$
T
$W_{out} = [1, 1], \quad b_{out} = -1$

3 Vector Derivative (Optional)

Consider the following functions:

$$f_1\left(\begin{bmatrix}x_1\\x_2\end{bmatrix}\right)=\begin{bmatrix}\frac{1}{\pi}\sin(\pi x_2)\\e^{x_1-1}x_2^2\\x_1x_2\end{bmatrix},\quad f_2\left(\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}\right)=\begin{bmatrix}x_1+x_2+x_3\\x_1^2+x_2^2+x_3^2\end{bmatrix}$$

where $f(x)=(f_2\circ f_1)(x)$.

Determine

$$\frac{\partial f}{\partial x}\text{ at point }\begin{bmatrix}1\\2\end{bmatrix}$$

Consider the following functions:

$$f_1\left(\begin{bmatrix}x_1\\x_2\end{bmatrix}\right)=\begin{bmatrix}\frac{1}{\pi}\sin(\pi x_2)\\e^{x_1-1}x_2^2\\x_1x_2\end{bmatrix},\quad f_2\left(\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}\right)=\begin{bmatrix}x_1+x_2+x_3\\x_1^2+x_2^2+x_3^2\end{bmatrix}$$

where $f(x)=(f_2\circ f_1)(x)$.

We need to determine

$$\frac{\partial f}{\partial x}\text{ at the point }\begin{bmatrix}1\\2\end{bmatrix}.$$

First, compute the Jacobian of f_1:

$$f_1\left(\begin{bmatrix}x_1\\x_2\end{bmatrix}\right)=\begin{bmatrix}\frac{1}{\pi}\sin(\pi x_2)\\e^{x_1-1}x_2^2\\x_1x_2\end{bmatrix}$$

$$J_{f_1}=\begin{bmatrix}0&\cos(\pi x_2)\\e^{x_1-1}x_2^2&2e^{x_1-1}x_2\\x_2&x_1\end{bmatrix}$$

Evaluating at $\begin{bmatrix}1\\2\end{bmatrix}$:

$$J_{f_1}\left(\begin{bmatrix}1\\2\end{bmatrix}\right)=\begin{bmatrix}0&1\\4&4\\2&1\end{bmatrix}$$

Next, compute the Jacobian of f_2:

$$f_2\left(\begin{bmatrix}x_1\\x_2\\x_3\end{bmatrix}\right)=\begin{bmatrix}x_1+x_2+x_3\\x_1^2+x_2^2+x_3^2\end{bmatrix}$$

$$J_{f_2}=\begin{bmatrix}1&1&1\\2x_1&2x_2&2x_3\end{bmatrix}$$

Evaluate $f_1\left(\begin{bmatrix}1\\2\end{bmatrix}\right)$:

$$f_1\left(\begin{bmatrix}1\\2\end{bmatrix}\right) = \begin{bmatrix}0\\4\\2\end{bmatrix}$$

Evaluate J_{f_2} at $\begin{bmatrix}0\\4\\2\end{bmatrix}$:

$$J_{f_2}\left(\begin{bmatrix}0\\4\\2\end{bmatrix}\right) = \begin{bmatrix}1 & 1 & 1\\0 & 8 & 4\end{bmatrix}$$

Now, combine the Jacobians using the chain rule:

$$J_f = J_{f_2}\left(\begin{bmatrix}0\\4\\2\end{bmatrix}\right) \cdot J_{f_1}\left(\begin{bmatrix}1\\2\end{bmatrix}\right)$$

$$J_f = \begin{bmatrix}1 & 1 & 1\\0 & 8 & 4\end{bmatrix} \cdot \begin{bmatrix}0 & 1\\4 & 4\\2 & 1\end{bmatrix} = \begin{bmatrix}4 & 6\\32 & 36\end{bmatrix}$$

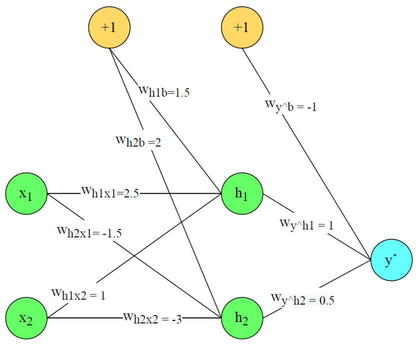
Thus, the Jacobian matrix $\frac{\partial f}{\partial x}$ at the point $\begin{bmatrix}1\\2\end{bmatrix}$ is:

$$\frac{\partial f}{\partial x}\left(\begin{bmatrix}1\\2\end{bmatrix}\right) = \begin{bmatrix}4 & 6\\32 & 36\end{bmatrix}$$

4 Backpropagation Algorithm 1 (Optional)

The following image shows a two-layer neural network with two nodes in the hidden layer and one node in the output layer. The network inputs are x_1 and x_2, and each node has a bias of value 1.

1. Calculate the value at nodes $\hat{y}, h_1$, and h_2 for input $\{x_1 = 0, x_2 = 1\}$.
2. Execute one step of the backpropagation algorithm for the previous input in part (a) with output $y = 1$.
3. Calculate the updated weights for the hidden layer and output layer (a total of 9 weights) by executing one step of the gradient descent algorithm.



Part 1: Forward Pass Calculation

Given:

$$x_1 = 0, \quad x_2 = 1$$

Weights:

$$\text{Input to } h_1: \quad W_{h1x1} = 2.5, \quad W_{h1x2} = 1, \quad W_{h1b} = 1.5$$

$$\text{Input to } h_2: \quad W_{h2x1} = -1.5, \quad W_{h2x2} = -0.3, \quad W_{h2b} = 2$$

$$\text{Hidden to output:} \quad W_{y\hat{b}} = -1, \quad W_{y\hat{h1}} = 1, \quad W_{y\hat{h2}} = 0.5$$

Calculate h_1 and h_2:

$$h_1 = \sigma(W_{h1x1} \cdot x_1 + W_{h1x2} \cdot x_2 + W_{h1b} \cdot 1) = \sigma(2.5 \cdot 0 + 1 \cdot 1 + 1.5 \cdot 1) = \sigma(2.5)$$

$$h_2 = \sigma(W_{h2x1} \cdot x_1 + W_{h2x2} \cdot x_2 + W_{h2b} \cdot 1) = \sigma(-1.5 \cdot 0 + (-0.3) \cdot 1 + 2 \cdot 1) = \sigma(1.7)$$

Assume $\sigma(x) = \frac{1}{1+e^{-x}}$:

$$h_1 = \sigma(2.5) \approx 0.9241$$

$$h_2 = \sigma(1.7) \approx 0.8455$$

Calculate $\hat{y}$:

$$\hat{y} = \sigma(W_{y\hat{h1}} \cdot h_1 + W_{y\hat{h2}} \cdot h_2 + W_{y\hat{b}} \cdot 1) = \sigma(1 \cdot 0.9241 + 0.5 \cdot 0.8455 + (-1) \cdot 1) = \sigma(0.34685)$$

$$\hat{y} \approx 0.5859$$

Part 2: Backpropagation Calculation

Given output $y = 1$:

Calculate error at output:

$$\delta_{\hat{y}} = \hat{y}(1 - \hat{y})(y - \hat{y}) = 0.5859(1 - 0.5859)(1 - 0.5859) \approx 0.0993$$

Calculate error at hidden nodes:

$$\delta_{h1} = h_1(1 - h_1)W_{y\hat{h1}}\delta_{\hat{y}} = 0.9241(1 - 0.9241) \cdot 1 \cdot 0.0993 \approx 0.0069$$

$$\delta_{h2} = h_2(1 - h_2)W_{y\hat{h2}}\delta_{\hat{y}} = 0.8455(1 - 0.8455) \cdot 0.5 \cdot 0.0993 \approx 0.0065$$

Part 3: Gradient Descent Calculation

Assume learning rate $\alpha = 0.1$:

Update weights for hidden to output:

$$W_{y\hat{h1}}^{new} = W_{y\hat{h1}} + \alpha\delta_{\hat{y}}h_1 = 1 + 0.1 \cdot 0.0993 \cdot 0.9241 \approx 1.0092$$

$$W_{y\hat{h2}}^{new} = W_{y\hat{h2}} + \alpha\delta_{\hat{y}}h_2 = 0.5 + 0.1 \cdot 0.0993 \cdot 0.8455 \approx 0.5084$$

$$W_{y\hat{b}}^{new} = W_{y\hat{b}} + \alpha\delta_{\hat{y}} \cdot 1 = -1 + 0.1 \cdot 0.0993 \approx -0.9901$$

Update weights for input to hidden:

$$W_{h1x1}^{new} = W_{h1x1} + \alpha\delta_{h1}x_1 = 2.5 + 0.1 \cdot 0.0069 \cdot 0 \approx 2.5$$

$$W_{h1x2}^{new} = W_{h1x2} + \alpha\delta_{h1}x_2 = 1 + 0.1 \cdot 0.0069 \cdot 1 \approx 1.0007$$

$$W_{h1b}^{new} = W_{h1b} + \alpha\delta_{h1} \cdot 1 = 1.5 + 0.1 \cdot 0.0069 \approx 1.5007$$

$$W_{h2x1}^{new} = W_{h2x1} + \alpha\delta_{h2}x_1 = -1.5 + 0.1 \cdot 0.0065 \cdot 0 \approx -1.5$$

$$W_{h2x2}^{new} = W_{h2x2} + \alpha\delta_{h2}x_2 = -0.3 + 0.1 \cdot 0.0065 \cdot 1 \approx -0.2994$$

$$W_{h2b}^{new} = W_{h2b} + \alpha\delta_{h2} \cdot 1 = 2 + 0.1 \cdot 0.0065 \approx 2.0007$$

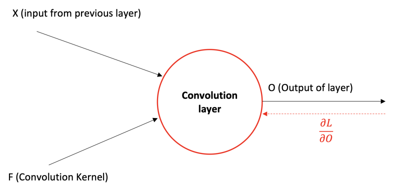
5 Back Propagation in CNN - Convolution in Each Direction!

In this question, we are going to perform backpropagation for a convolutional layer and come up with a closed-form answer for our required derivatives. First, let's look at the general schema of our layer:

We know how the forward pass for convolution layers is computed (if not, please refer to the course's slides!). Suppose X is a 2D matrix, and thus F is also 2D. Please note that the dimensions of these two matrices can arbitrarily change. Prove that:

$$\frac{\partial L}{\partial F} = X * \frac{\partial L}{\partial O}, \quad \frac{\partial L}{\partial X} = F \otimes \frac{\partial L}{\partial O}$$

where $*$ represents the convolution operation and $\otimes$ is full convolution.


X (input from previous layer)
Convolution layer
O (Output of layer)
$\frac{\partial L}{\partial O}$
F (Convolution Kernel)

Forward Pass for Convolution

Let X be the input matrix, F be the filter, and O be the output matrix resulting from the convolution operation:

$$O = X * F$$

The element $O_{i,j}$ of the output matrix is computed as:

$$O_{i,j} = \sum_m \sum_n X_{i+m,j+n} F_{m,n}$$

Gradient with Respect to the Filter F

To compute $\frac{\partial L}{\partial F}$, we need to apply the chain rule to the loss L with respect to the filter elements $F_{m,n}$:

$$\frac{\partial L}{\partial F_{m,n}} = \sum_i \sum_j \frac{\partial L}{\partial O_{i,j}} \frac{\partial O_{i,j}}{\partial F_{m,n}}$$

From the convolution operation, we have:

$$\frac{\partial O_{i,j}}{\partial F_{m,n}} = X_{i+m,j+n}$$

Therefore:

$$\frac{\partial L}{\partial F_{m,n}} = \sum_i \sum_j \frac{\partial L}{\partial O_{i,j}} X_{i+m,j+n}$$

This is equivalent to convolving X with $\frac{\partial L}{\partial O}$:

$$\frac{\partial L}{\partial F} = X * \frac{\partial L}{\partial O}$$

Gradient with Respect to the Input X

To compute $\frac{\partial L}{\partial X}$, we apply the chain rule to the loss L with respect to the input elements $X_{i,j}$:

$$\frac{\partial L}{\partial X_{i,j}} = \sum_m \sum_n \frac{\partial L}{\partial O_{m,n}} \frac{\partial O_{m,n}}{\partial X_{i,j}}$$

From the convolution operation, we have:

$$\frac{\partial O_{m,n}}{\partial X_{i,j}} = F_{i-m,j-n}$$

Therefore:

$$\frac{\partial L}{\partial X_{i,j}} = \sum_m \sum_n \frac{\partial L}{\partial O_{m,n}} F_{i-m,j-n}$$

This is equivalent to the full convolution of F with $\frac{\partial L}{\partial O}$:

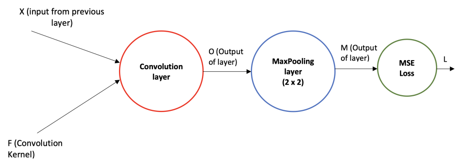
$$\frac{\partial L}{\partial X} = F \otimes \frac{\partial L}{\partial O}$$

Conclusion

We have derived that:

$$\frac{\partial L}{\partial F} = X * \frac{\partial L}{\partial O}$$
$$\frac{\partial L}{\partial X} = F \otimes \frac{\partial L}{\partial O}$$

6 Backpropagation in CNN - Example



In this problem, write the update rules and update weights corresponding to variables/matrices X, F, O, and M. The required information is provided below:

$$X = \begin{bmatrix} 1 & 7 & -1 & -7 & 10 & 11 \\ 2 & 8 & 0 & 0 & 12 & 13 \\ 3 & 9 & 0 & 0 & 0 & 0 \\ 4 & 10 & -4 & -10 & 0 & 0 \\ 5 & 11 & -5 & -11 & 16 & 17 \\ 6 & 12 & -6 & -12 & 14 & 15 \end{bmatrix}$$

$$F = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix}$$

$\hat{L} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$

Forward Pass Calculation

The output O is calculated by convolving X with F:

Backpropagation Calculation

To compute the gradients, we need to backpropagate the error from $\hat{L}$.

$$\frac{\partial L}{\partial O} = \hat{L}$$

Gradient with Respect to Filter F

$$\frac{\partial L}{\partial F} = X * \hat{L}$$

After convolving X with $\hat{L}$:

$$\frac{\partial L}{\partial F} = \begin{bmatrix} 1 & 7 & -1 \\ 2 & 8 & 0 \\ 3 & 9 & 0 \end{bmatrix}$$

Gradient with Respect to Input X

$$\frac{\partial L}{\partial X} = F \otimes \hat{L}$$

After full convolution of F with $\hat{L}$:

$$\frac{\partial L}{\partial X} = \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ 2 & 1 & -2 & -1 & 0 & 0 \\ 3 & 2 & -3 & -2 & 1 & 0 \\ 2 & 2 & -2 & -2 & 2 & 0 \\ 1 & 1 & -1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Update Weights

Let $\alpha = 0.01$.

Update Rule for Filter F

$$F_{\text{new}} = F - \alpha \frac{\partial L}{\partial F}$$

$$F_{\text{new}} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} - 0.01 \begin{bmatrix} 1 & 7 & -1 \\ 2 & 8 & 0 \\ 3 & 9 & 0 \end{bmatrix}$$

$$F_{\text{new}} = \begin{bmatrix} 0.99 & -0.07 & -0.99 \\ 1.98 & -0.08 & -2 \\ 0.97 & -0.09 & -1 \end{bmatrix}$$

Update Rule for Input X

$$X_{\text{new}} = X - \alpha \frac{\partial L}{\partial X}$$

$$X_{\text{new}} = \begin{bmatrix} 1 & 7 & -1 & -7 & 10 & 11 \\ 2 & 8 & 0 & 0 & 12 & 13 \\ 3 & 9 & 0 & 0 & 0 & 0 \\ 4 & 10 & -4 & -10 & 0 & 0 \\ 5 & 11 & -5 & -11 & 16 & 17 \\ 6 & 12 & -6 & -12 & 14 & 15 \end{bmatrix} - 0.01 \begin{bmatrix} 1 & 0 & -1 & 0 & 0 & 0 \\ 2 & 1 & -2 & -1 & 0 & 0 \\ 3 & 2 & -3 & -2 & 1 & 0 \\ 2 & 2 & -2 & -2 & 2 & 0 \\ 1 & 1 & -1 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

$$X_{\text{new}} = \begin{bmatrix} 0.99 & 7 & -0.99 & -7 & 10 & 11 \\ 1.98 & 7.99 & 0.02 & 0.01 & 12 & 13 \\ 2.97 & 8.98 & 0.03 & 0.02 & -0.01 & 0 \\ 3.98 & 9.98 & -3.98 & -9.98 & -0.02 & 0 \\ 4.99 & 10.99 & -4.99 & -11.01 & 15.99 & 17 \\ 6 & 12 & -6 & -12 & 14 & 14.99 \end{bmatrix}$$

7 CNNs are Universal Approximators

- We know that CNNs are universal approximators, meaning that any function can be approximated using CNNs. The same is true for MLPs (multilayer perceptrons).
- Can an equivalent MLP be created from a CNN? If so, under what circumstances and conditions can we create an equivalent MLP? If not, explain why this cannot be done.

Answer

Yes, an equivalent MLP can be created from a CNN under certain circumstances and conditions. Specifically, an MLP can simulate the behavior of a CNN, but it may not be as efficient in terms of parameter sharing and computational complexity. Here is a detailed explanation and proof:

Conditions and Circumstances

To understand how an MLP can be equivalent to a CNN, we need to consider the following points:

1. **Parameter Sharing:** CNNs use convolutional layers where a single filter (or kernel) is applied across the entire input to extract features. This results in parameter sharing, where the same set of weights is used for different parts of the input.
2. **Local Connectivity:** CNNs exploit the local connectivity pattern, where each neuron in a layer is connected to a small region of the input from the previous layer.
3. **Flattening the Input:** To simulate a CNN with an MLP, the input must be flattened into a single vector. For example, a 2D image of dimensions $h \times w$ with c channels is converted into a vector of length $h \times w \times c$.
4. **Unrolling Convolutional Operations:** Each convolutional operation can be represented as a matrix multiplication. The sliding of the filter over the input can be expressed as a series of linear transformations.

Proof

1. **Flattening the Input:**
Consider an input $X \in \mathbb{R}^{h \times w \times c}$. Flatten this input into a vector $X_f \in \mathbb{R}^{h \cdot w \cdot c}$.

2. **Representing Convolution as Matrix Multiplication:**
A convolution operation with filter $F \in \mathbb{R}^{k \times k \times c}$ can be unrolled into a matrix multiplication. For each position where the filter is applied, there is a

corresponding row in a matrix A such that:

$$A \cdot X_f = O_f$$

where O_f is the flattened output of the convolution operation.

3. **Equivalent MLP Structure:**

An MLP can be structured to simulate this operation by ensuring that the weights and biases of the MLP layers mimic the convolutional operations:

$$O_f = \sigma(W \cdot X_f + b)$$

where W and b are chosen to replicate the convolutional filters and their application across the input.

4. **Combining Layers:**

Multiple convolutional layers and pooling operations can be similarly unrolled and represented as matrix multiplications followed by nonlinear activations. Each layer in the CNN corresponds to a layer in the MLP.

Efficiency Considerations

While it is theoretically possible to create an equivalent MLP for a given CNN, the MLP will not be as efficient in practice. The primary reasons are:

1. **Parameter Efficiency:** CNNs leverage parameter sharing, reducing the number of unique parameters. An equivalent MLP would require a much larger number of parameters to achieve the same functionality.

2. **Computational Efficiency:** The localized and sparse connections in CNNs lead to lower computational complexity compared to the fully connected layers in MLPs, which require dense matrix multiplications.

Conclusion

In conclusion, an MLP can be constructed to be equivalent to a CNN in terms of its ability to approximate functions. However, the MLP will generally be less efficient both in terms of parameter count and computational requirements. The practical advantages of CNNs, such as parameter sharing and local connectivity, make them more suitable for tasks involving spatial hierarchies, like image and video processing.

8 Backpropagation Algorithm 2

Consider the following convolutional network with these layers:

- Layer 1 (Convolutional):

$$\text{Input: } x = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix}, \quad \text{Filter: } v_1 = \begin{bmatrix} w_1 & w_2 \end{bmatrix}, \quad \text{Output: } z_1 = \tanh(x * v_1)$$

- Layer 2 (Fully-Connected):

$$\text{Input: } z_1, \quad \text{Filter: } v_2 = \begin{bmatrix} w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix}, \quad \text{Output: } z_2 = \sigma(v_2 \times z_1)$$

- Layer 3 (Fully-Connected):

$$\text{Input: } z_2, \quad \text{Filter: } v_3 = \begin{bmatrix} w_5 & w_6 \end{bmatrix}, \quad \text{Output: } y^* = v_3 \times z_2$$

Using the backpropagation algorithm, obtain the derivative of $(y^* - y)^2$ with respect to w_1.

Using the backpropagation algorithm, we want to obtain the derivative of $(y^* - y)^2$ with respect to w_1.

The forward pass computations are:

$$z_{1,11} = w_1 x_{11} + w_2 x_{12}, \quad z_{1,21} = w_1 x_{21} + w_2 x_{22}$$

$$z_1 = \begin{bmatrix} \tanh(z_{1,11}) \\ \tanh(z_{1,21}) \end{bmatrix}$$

$$z_2 = v_2 \times z_1 = \begin{bmatrix} w_{31}\tanh(z_{1,11}) + w_{32}\tanh(z_{1,21}) \\ w_{41}\tanh(z_{1,11}) + w_{42}\tanh(z_{1,21}) \end{bmatrix}$$

$$y^* = v_3 \times z_2 = w_5\left(w_{31}\tanh(z_{1,11}) + w_{32}\tanh(z_{1,21})\right) + w_6\left(w_{41}\tanh(z_{1,11}) + w_{42}\tanh(z_{1,21})\right)$$

The loss function is:

$$L = (y^* - y)^2$$

To find $\frac{\partial L}{\partial w_1}$:

$$\frac{\partial L}{\partial w_1} = 2(y^* - y)\frac{\partial y^*}{\partial w_1}$$

First, find $\frac{\partial y^*}{\partial w_1}$:

$$\frac{\partial y^*}{\partial w_1} = w_5\left(w_{31}\frac{\partial \tanh(z_{1,11})}{\partial w_1} + w_{32}\frac{\partial \tanh(z_{1,21})}{\partial w_1}\right) + w_6\left(w_{41}\frac{\partial \tanh(z_{1,11})}{\partial w_1} + w_{42}\frac{\partial \tanh(z_{1,21})}{\partial w_1}\right)$$

Using $\frac{\partial \tanh(z)}{\partial z} = 1 - \tanh^2(z)$:

$$\frac{\partial \tanh(z_{1,11})}{\partial w_1} = (1 - \tanh^2(z_{1,11}))x_{11}, \quad \frac{\partial \tanh(z_{1,21})}{\partial w_1} = (1 - \tanh^2(z_{1,21}))x_{21}$$

Substituting these into the gradient of y^*:

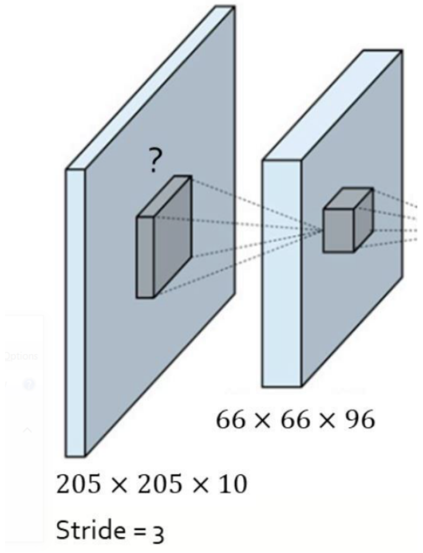
$$\frac{\partial y^*}{\partial w_1} = w_5 \left(w_{31}(1-\tanh^2(z_{1,11}))x_{11} + w_{32}(1-\tanh^2(z_{1,21}))x_{21}\right) + w_6 \left(w_{41}(1-\tanh^2(z_{1,11}))x_{11} + w_{42}(1-\tanh$$

Thus:

$$\frac{\partial L}{\partial w_1} = 2(y^* - y) \left((w_5 w_{31} + w_6 w_{41})(1-\tanh^2(w_1 x_{11} + w_2 x_{12}))x_{11} + (w_5 w_{32} + w_6 w_{42})(1-\tanh^2(w_1 x_{21} + w_2 x_2$$

9 Model Parameters

Consider the following two-layer convolutional network:



1. Based on the input and output dimensions shown, determine the size of the kernel used for this operation.
2. Determine the number of trainable parameters in this layer.
3. Calculate the number of multiplication operations required to obtain the output.

9.1 Kernel Size Calculation

Given:

$$\text{Input dimensions: } 205 \times 205 \times 10$$

$$\text{Output dimensions: } 66 \times 66 \times 96$$

$$\text{Stride: } 3$$

We use the formula for the output dimension of a convolutional layer:

$$O = \frac{I - K + 2P}{S} + 1$$

where:

- O is the output dimension
- I is the input dimension

- K is the kernel size
- P is the padding
- S is the stride

For simplicity, assume padding $P = 0$:

$$\begin{gathered} 66 = \frac{205 - K}{3} + 1 \\ 65 = \frac{205 - K}{3} \\ 195 = 205 - K \\ K = 10 \end{gathered}$$

Thus, the kernel size is 10×10.

9.2 Number of Trainable Parameters

The number of trainable parameters in a convolutional layer is given by:

$$\begin{gathered} \text{Number of parameters} = (K \times K \times \text{Input channels} + 1) \times \text{Output channels} \\ = (10 \times 10 \times 10 + 1) \times 96 \\ = (1000 + 1) \times 96 \\ = 1001 \times 96 \\ = 96096 \end{gathered}$$

Thus, the number of trainable parameters is 96096.

9.3 Number of Multiplication Operations

The number of multiplication operations required for one output feature map is:

$$\begin{gathered} \text{Number of multiplications per output pixel} = K \times K \times \text{Input channels} \\ = 10 \times 10 \times 10 \\ = 1000 \end{gathered}$$

The total number of multiplications for one output feature map is:

$$\begin{gathered} \text{Output dimensions} \times \text{Number of multiplications per output pixel} \\ = 66 \times 66 \times 1000 \\ = 4356000 \end{gathered}$$

Since there are 96 output channels, the total number of multiplications is:

$$\begin{gathered} 4356000 \times 96 \\ = 418176000 \end{gathered}$$

Thus, the number of multiplication operations required to obtain the output is 418176000.

10 Receptive Field

Determine a closed-form and non-recursive formula for the receptive field of a 1×1 square in the n^{th} convolution layer of a neural network, starting from the first layer, in terms of kernel size and stride.

To determine a closed-form and non-recursive formula for the receptive field of a 1×1 square in the n^{th} convolution layer of a neural network, we need to consider the effects of the kernel size and stride at each layer.

Let:

- k_i be the kernel size of the i^{th} layer.
- s_i be the stride of the i^{th} layer.
- RF_n be the receptive field of a 1×1 square in the n^{th} layer.

The receptive field of a 1×1 square in the first layer is equal to the kernel size:

$$RF_1 = k_1$$

For subsequent layers, the receptive field can be determined by considering how each layer's kernel size and stride affects the receptive field size. The receptive field of a 1×1 square in the n^{th} layer depends on the receptive field in the $(n-1)^{\text{th}}$ layer, the kernel size of the n^{th} layer, and the stride of the n^{th} layer.

The general formula for the receptive field in the n^{th} layer is:

$$RF_n = RF_{n-1} + (k_n - 1) \prod_{i=1}^{n-1} s_i$$

Using the above recursive formula, we derive the closed-form formula for RF_n.

Derivation of the Closed-Form Formula

Starting from the base case:

$$RF_1 = k_1$$

For $n = 2$:

$$RF_2 = RF_1 + (k_2 - 1)s_1 = k_1 + (k_2 - 1)s_1$$

For $n = 3$:

$$RF_3 = RF_2 + (k_3 - 1)s_1 s_2 = (k_1 + (k_2 - 1)s_1) + (k_3 - 1)s_1 s_2$$

$$RF_3 = k_1 + (k_2 - 1)s_1 + (k_3 - 1)s_1 s_2$$

By observing the pattern, we can generalize this to:

$$RF_n = k_1 + \sum_{i=2}^{n} \left((k_i - 1) \prod_{j=1}^{i-1} s_j \right)$$

Thus, the closed-form and non-recursive formula for the receptive field of a 1×1 square in the n^{th} convolution layer is:

$$RF_n = k_1 + \sum_{i=2}^{n} \left((k_i - 1) \prod_{j=1}^{i-1} s_j \right)$$

11 Optimizing Deep Learning Models

11.1 Momentum in Optimization

Recall the gradient descent algorithm:

$$\theta_{t+1} = \theta_t - \alpha \nabla f(\theta_t)$$

where θ_t is the parameter at time t, α is the learning rate, and $\nabla f(\theta_t)$ is the gradient of the loss function.

The Momentum algorithm extends gradient descent with a momentum term:

$$v_{t+1} = \beta v_t + (1-\beta)\nabla f(\theta_t)$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

where v_t is the momentum term, and β is the momentum parameter.

11.1.1 Question 1

The Nesterov Accelerated Gradient (NAG) algorithm extends the Momentum algorithm:

$$v_{t+1} = \beta v_t + (1-\beta)\nabla f(\theta_t - \alpha v_t)$$

$$\theta_{t+1} = \theta_t - \alpha v_{t+1}$$

Explain how the NAG algorithm works and improves the training process compared to the Momentum algorithm.

Answer: The Nesterov Accelerated Gradient (NAG) algorithm anticipates the future position of the parameters by first making a partial update using the momentum term v_t. It then computes the gradient at this look-ahead position $\theta_t - \alpha v_t$. This approach helps in correcting the course earlier than the standard momentum algorithm, resulting in faster convergence and better accuracy. By considering the gradient at the look-ahead position, NAG reduces the risk of overshooting the minimum and provides more accurate updates.

11.1.2 Question 2

The Adagrad algorithm adapts learning rates based on historical gradients:

$$g_{t+1} = g_t + (\nabla f(\theta_t))^2$$

$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{g_{t+1} + \epsilon}}\nabla f(\theta_t)$$

Explain how the Adagrad algorithm improves the training process compared to the Momentum algorithm.

Answer: The Adagrad algorithm adapts the learning rate for each parameter based on the historical accumulation of gradients. As training progresses, the learning rate for frequently updated parameters diminishes, allowing the algorithm to make more fine-tuned updates. This is particularly useful for sparse data or features where infrequent parameters get larger updates initially, while frequent parameters get smaller updates over time. Adagrad can handle large datasets and is effective in dealing with sparse data by providing adaptive learning rates, whereas the momentum algorithm uses a constant learning rate, which may not be optimal for all parameters.

11.2 Approximate Newton Methods (Optional)

Write the update rule of the Newton's Method and explain how it works.

Answer: Newton's Method update rule is given by:

$$\theta_{t+1} = \theta_t - \alpha H^{-1} \nabla f(\theta_t)$$

where H is the Hessian matrix of second-order partial derivatives of the loss function. Newton's Method uses curvature information (second-order derivatives) to make more informed updates, leading to faster convergence, especially near the optimum. However, computing the Hessian and its inverse can be computationally expensive and impractical for high-dimensional data.

11.3 Adam Optimizer

The Adam optimizer combines Momentum and Adagrad concepts. Its update rule is:

$$g_t = \nabla_\theta f(\theta_{t-1})$$

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

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

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

$$\theta_t = \theta_{t-1} - \frac{\alpha \hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

1. Explain how the Adam optimizer works and improves training.
2. Explain why m_t has a bias toward zero initially and how $\hat{m}_t$ corrects this bias.

Answer 1: The Adam optimizer combines the advantages of both Momentum and Adagrad by using exponentially decaying averages of past gradients (m_t) and past squared gradients (v_t). This provides an adaptive learning rate for each parameter. The momentum term m_t helps in accelerating the convergence in the relevant direction, while the adaptive learning rate term v_t ensures that the updates are scaled appropriately, preventing large updates that could hinder convergence.

Answer 2: Initially, m_t and v_t are biased towards zero because they are initialized as zero vectors. To correct this bias, the terms $\hat{m}_t$ and $\hat{v}_t$ are introduced, which are bias-corrected versions of m_t and v_t, respectively. The correction terms $1 - \beta_1^t$ and $1 - \beta_2^t$ are used to adjust the estimates, making them unbiased. This ensures that during the early stages of training, the optimizer can still make significant updates.

12 Quadratic Error Function

Consider a quadratic error function:

$$E = E_0 + \frac{1}{2}(w - w^*)^T H(w - w^*)$$

where w^* represents the minimum, and the Hessian matrix H is positive definite and constant. Suppose the initial weight vector $w(0)$ is chosen at the origin and updated via gradient descent:

$$w_\tau = w_{\tau-1} - \rho\nabla E$$

where τ denotes the step number and ρ is the learning rate. Show that the components of the weight vector parallel to the Hessian eigenvectors follow specific conditions after τ steps.

Solution

The gradient of the error function E is given by:

$$\nabla E = H(w - w^*)$$

Substituting the gradient into the gradient descent update rule, we get:

$$w_\tau = w_{\tau-1} - \rho H(w_{\tau-1} - w^*)$$

Let's rewrite this update rule in terms of the error from the minimum:

$$e_\tau = w_\tau - w^*$$

$$e_\tau = e_{\tau-1} - \rho H e_{\tau-1}$$
$$e_\tau = (I - \rho H)e_{\tau-1}$$

We initialize $w(0) = 0$, thus:

$$e_0 = -w^*$$

Now, we apply the update rule iteratively:

$$e_1 = (I - \rho H)e_0$$

$$e_2 = (I - \rho H)e_1 = (I - \rho H)^2 e_0$$

$$\vdots$$

$$e_\tau = (I - \rho H)^\tau e_0$$

Since H is a symmetric positive definite matrix, it can be diagonalized by an orthogonal matrix Q such that:

$$H = Q\Lambda Q^T$$

where Λ is a diagonal matrix containing the eigenvalues λ_i of H.

Transform e_τ into the eigenbasis of H:

$$e_\tau = (I - \rho H)^\tau e_0$$

$$e_\tau = (I - \rho Q \Lambda Q^T)^\tau e_0$$

$$e_\tau = Q(I - \rho\Lambda)^\tau Q^T e_0$$

Let $c = Q^T e_0$, then:

$$e_\tau = Q(I - \rho\Lambda)^\tau c$$

Since Λ is diagonal with entries λ_i:

$$(I - \rho\Lambda)^\tau = \text{diag}((1 - \rho\lambda_1)^\tau, (1 - \rho\lambda_2)^\tau, \ldots, (1 - \rho\lambda_n)^\tau)$$

Thus:

$$e_\tau = Q\text{diag}((1 - \rho\lambda_1)^\tau, (1 - \rho\lambda_2)^\tau, \ldots, (1 - \rho\lambda_n)^\tau)c$$

This shows that the components of the error vector e_τ along the eigenvectors of H decay exponentially at rates determined by the eigenvalues λ_i and the learning rate ρ:

$$e_{\tau,i} = (1 - \rho\lambda_i)^\tau e_{0,i}$$

Since $e_0 = -w^*$, we have:

$$e_{\tau,i} = (1 - \rho\lambda_i)^\tau (-w_i^*)$$

Finally, the weight vector components follow:

$$w_{\tau,i} = w_i^* + e_{\tau,i} = w_i^* - (1 - \rho\lambda_i)^\tau w_i^*$$

$$w_{\tau,i} = w_i^* \left(1 - (1 - \rho\lambda_i)^\tau\right)$$

13 LSTM

Prove that the LSTM cell can remember information over long sequences by showing that the derivative of the cell state $c(t)$ with respect to $c(t-1)$ does not vanish over time.

Proof

The LSTM cell state update equation is given by:

$$c(t) = f(t) \odot c(t-1) + i(t) \odot \tilde{c}(t)$$

where:

- $c(t)$ is the cell state at time t.
- $f(t)$ is the forget gate at time t.
- $i(t)$ is the input gate at time t.
- $\tilde{c}(t)$ is the candidate cell state at time t.
- $\odot$ denotes element-wise multiplication.

To show that the LSTM can remember information over long sequences, we need to compute the derivative of $c(t)$ with respect to $c(t-1)$:

$$\frac{\partial c(t)}{\partial c(t-1)} = \frac{\partial}{\partial c(t-1)} \left(f(t) \odot c(t-1) + i(t) \odot \tilde{c}(t) \right)$$

Since $f(t)$, $i(t)$, and $\tilde{c}(t)$ are functions of the inputs and previous hidden states and do not directly depend on $c(t-1)$, the partial derivative simplifies to:

$$\frac{\partial c(t)}{\partial c(t-1)} = \frac{\partial}{\partial c(t-1)} \left(f(t) \odot c(t-1) \right)$$

Applying the chain rule:

$$\frac{\partial c(t)}{\partial c(t-1)} = f(t) \odot \frac{\partial c(t-1)}{\partial c(t-1)}$$

Since $\frac{\partial c(t-1)}{\partial c(t-1)}$ is the identity matrix, we get:

$$\frac{\partial c(t)}{\partial c(t-1)} = f(t)$$

The forget gate $f(t)$ typically has values between 0 and 1 because it is the output of a sigmoid function. Therefore, the derivative $\frac{\partial c(t)}{\partial c(t-1)}$ is equal to $f(t)$, which does not vanish over time as long as $f(t)$ is not zero. In practice, $f(t)$ is usually close to 1, which means that the cell state $c(t)$ retains a significant portion of the information from $c(t-1)$.

Consequently, the derivative $\frac{\partial c(t)}{\partial c(t-1)}$ being equal to $f(t)$ ensures that the gradient does not vanish over time, enabling the LSTM cell to remember information over long sequences.

14 RNN (Optional)

14.1 1. Complex Dynamics of RNNs

Derive the Jacobian matrix of hidden state transitions in an RNN, and discuss its role in network stability.

Answer: Consider an RNN with hidden state h_t at time step t and input x_t. The hidden state update equation is:

$$h_t = \phi(W_h h_{t-1} + W_x x_t + b)$$

where ϕ is the activation function, W_h is the hidden state weight matrix, W_x is the input weight matrix, and b is the bias vector.

The Jacobian matrix of the hidden state transitions is the partial derivative of h_t with respect to h_{t-1}:

$$J_t = \frac{\partial h_t}{\partial h_{t-1}}$$

Applying the chain rule, we get:

$$J_t = \frac{\partial \phi(W_h h_{t-1} + W_x x_t + b)}{\partial h_{t-1}} = \phi'(W_h h_{t-1} + W_x x_t + b) W_h$$

Here, ϕ' denotes the derivative of the activation function ϕ. If ϕ is applied element-wise, then ϕ' is a diagonal matrix with the derivatives of ϕ evaluated at each element of the argument.

$$J_t = \text{diag}(\phi'(W_h h_{t-1} + W_x x_t + b)) W_h$$

The Jacobian matrix J_t describes how small changes in the hidden state at time $t-1$ affect the hidden state at time t. The eigenvalues of J_t determine the stability of the RNN: - If all eigenvalues are less than 1 in magnitude, the RNN is stable and small perturbations will decay over time. - If any eigenvalue is greater than 1 in magnitude, the RNN can become unstable and small perturbations can grow exponentially.

14.2 2. BPTT in Depth

Provide a detailed derivation of Backpropagation Through Time (BPTT) for RNNs handling variable-length input sequences.

Answer: Backpropagation Through Time (BPTT) involves unrolling the RNN across time steps and applying backpropagation to compute gradients with respect to the loss function. Consider an RNN with hidden state h_t, input x_t, and output y_t. The loss at time step t is L_t.

The total loss for a sequence of length T is:

$$L = \sum_{t=1}^{T} L_t$$

The hidden state update equation is:

$$h_t = \phi(W_h h_{t-1} + W_x x_t + b)$$

The output y_t and loss L_t depend on h_t:

$$y_t = \psi(W_y h_t + c), \quad L_t = \ell(y_t, \hat{y}_t)$$

To derive the gradients using BPTT, we need to compute $\frac{\partial L}{\partial \theta}$ for all parameters $\theta = \{W_h, W_x, W_y, b, c\}$.

Step 1: Gradient of the loss with respect to the output

$$\frac{\partial L}{\partial y_t} = \frac{\partial \ell(y_t, \hat{y}_t)}{\partial y_t}$$

Step 2: Gradient of the loss with respect to the hidden state

$$\frac{\partial L}{\partial h_t} = \frac{\partial L_t}{\partial h_t} + \sum_{k=t+1}^{T} \frac{\partial L_k}{\partial h_k} \frac{\partial h_k}{\partial h_t}$$

Using the chain rule:

$$\frac{\partial h_k}{\partial h_t} = \prod_{j=t+1}^{k} \frac{\partial h_j}{\partial h_{j-1}} = \prod_{j=t+1}^{k} J_j$$

Thus:

$$\frac{\partial L}{\partial h_t} = \frac{\partial L_t}{\partial h_t} + \sum_{k=t+1}^{T} \frac{\partial L_k}{\partial h_k} \left(\prod_{j=t+1}^{k} J_j \right)$$

Step 3: Gradient of the loss with respect to the parameters

$$\frac{\partial L}{\partial W_h} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial W_h}$$

$$\frac{\partial L}{\partial W_x} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial W_x}$$

$$\frac{\partial L}{\partial W_y} = \sum_{t=1}^{T} \frac{\partial L}{\partial y_t} \frac{\partial y_t}{\partial W_y}$$

$$\frac{\partial L}{\partial b} = \sum_{t=1}^{T} \frac{\partial L}{\partial h_t} \frac{\partial h_t}{\partial b}$$

$$\frac{\partial L}{\partial c} = \sum_{t=1}^{T} \frac{\partial L}{\partial y_t} \frac{\partial y_t}{\partial c}$$

Since h_t depends on W_h, W_x, and b, we apply the chain rule:

$$\frac{\partial h_t}{\partial W_h} = \phi'(W_h h_{t-1} + W_x x_t + b) h_{t-1}^T$$

$$\frac{\partial h_t}{\partial W_x} = \phi'(W_h h_{t-1} + W_x x_t + b) x_t^T$$

$$\frac{\partial h_t}{\partial b} = \phi'(W_h h_{t-1} + W_x x_t + b)$$

15 GANs

15.1 15.1 GANs: A Game-Theoretic Approach

15.1.1 Questions 1

Define GANs as a game. Who are the players, what are the actions, and what are the payoffs?

Answer: In GANs (Generative Adversarial Networks), the game consists of two players: the generator (G) and the discriminator (D).

- **Players:** - *Generator (G):* Generates fake data samples from a noise distribution. - *Discriminator (D):* Attempts to distinguish between real data samples and fake data samples generated by the generator.

- **Actions:** - *Generator:* Generates data samples $G(z)$ where z is sampled from a noise distribution p_z. - *Discriminator:* Outputs a probability $D(x)$ indicating whether the input data sample x is real ($x \sim p_{\text{data}}$) or fake ($x = G(z)$).

- **Payoffs:** - The discriminator aims to maximize the probability of correctly classifying real and fake samples:

$$\mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]$$

- The generator aims to minimize the discriminator's ability to classify fake samples as fake, effectively maximizing:

$$\mathbb{E}_{z\sim p_z}[\log D(G(z))]$$

The overall objective for the GANs can be seen as a minimax game:

$$\min_G \max_D \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]$$

15.1.2 Questions 2

Explain the optimization problem:

$$\max_{\theta_G} \min_{\theta_D} u_G(\theta_G, \theta_D) = \max_{\theta_G} \min_{\theta_D} \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]$$

and relate it to GAN training.

Answer: The optimization problem for GANs can be expressed as a two-player minimax game, where: - θ_G represents the parameters of the generator. - θ_D represents the parameters of the discriminator.

The objective function is:

$$u_G(\theta_G, \theta_D) = \mathbb{E}_{x\sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z\sim p_z}[\log(1 - D(G(z)))]$$

The generator tries to generate data samples $G(z)$ that maximize the discriminator's error, while the discriminator tries to maximize its ability to distinguish between real and fake samples. This leads to the optimization problem:

$$\max_{\theta_G} \min_{\theta_D} u_G(\theta_G, \theta_D)$$

In GAN training, this translates to alternating between updating the discriminator and the generator: - Update θ_D to maximize $u_G(\theta_G, \theta_D)$, i.e., improve the discriminator's performance. - Update θ_G to minimize $u_G(\theta_G, \theta_D)$, i.e., generate more realistic samples to fool the discriminator.

15.1.3 Questions 3

Using calculus of variations, solve the optimization problem for the optimal discriminator θ_D.

Answer: To find the optimal discriminator D given a fixed generator G, we solve the following optimization problem:

$$\max_{\theta_D} \mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)] + \mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))]$$

Taking the derivative with respect to $D(x)$ and setting it to zero, we get:

$$\frac{\partial}{\partial D} \left(\log D(x) + \log(1 - D(G(z)))\right) = 0$$

Solving for $D(x)$, we get the optimal discriminator:

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_G(x)}$$

15.2 15.2 GANs: You Don't Want to Train Them!

15.2.1 Questions 1

Explain why the first term in the GAN loss is not affected when updating generator parameters, and how this affects training.

Answer: The first term in the GAN loss, $\mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)]$, depends only on real data samples and the discriminator's ability to classify them correctly. This term is independent of the generator's parameters θ_G. When updating the generator, only the second term $\mathbb{E}_{z \sim p_z}[\log(1 - D(G(z)))]$ is affected. As a result, the generator focuses solely on producing realistic fake samples that can fool the discriminator. This asymmetry can lead to unstable training, as the generator and discriminator may not be equally balanced.

15.2.2 Questions 2

Explain the mode collapse problem and its impact on GAN training.

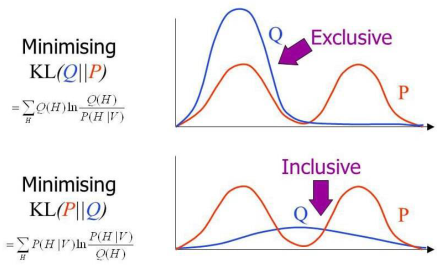
Answer: Mode collapse occurs when the generator produces a limited variety of samples, essentially generating similar or identical outputs for different inputs. This reduces the diversity of the generated data and fails to capture the full distribution of the real data. Mode collapse impacts GAN training by leading to poor generalization, as the generator fails to represent the true data distribution. The discriminator may easily distinguish the limited variety of fake samples from real samples, preventing effective training progress.

15.2.3 Questions 3

Explain the Wasserstein distance and how it improves GAN training over KL divergence.

Answer: The Wasserstein distance (also known as Earth Mover's distance) measures the minimum cost of transporting mass to transform one probability distribution into another. It provides a meaningful and smooth measure of distance between distributions, even when they do not overlap. In the context of GANs, Wasserstein GAN (WGAN) replaces the original loss function with one based on the Wasserstein distance, leading to more stable training.

The key improvements of Wasserstein distance over KL divergence in GAN training are: - **Continuity and Smoothness:** Wasserstein distance provides gradients everywhere, making the optimization landscape smoother and more conducive to gradient-based optimization. - **Better Mode Coverage:** WGAN encourages the generator to capture the full support of the data distribution, reducing mode collapse. - **Improved Convergence:** WGAN provides more stable training and better convergence properties compared to the original GAN formulation.


Minimising
KL(Q||P)
$=\sum_H Q(H)\ln\frac{Q(H)}{P(H|V)}$
Q
Exclusive
P
Minimising
KL(P||Q)
$=\sum_H P(H|V)\ln\frac{P(H|V)}{Q(H)}$
Inclusive
Q
P