# introduction to machine learning

DR.Amiri


electrical engineering department

Ahmadreza Majlesara 400101861

assignment 3

July 11, 2024

# introduction to machine learning

assignment 3

Ahmadreza Majlesara 400101861

## 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.)

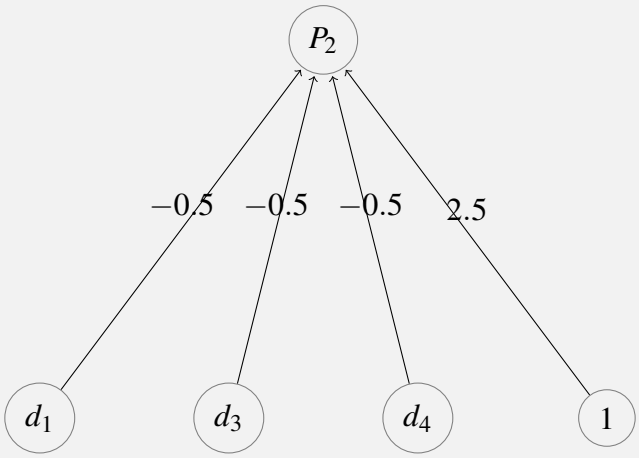
solution

the inputs to the network are the 4 bits of the input, and the outputs are the 3 parity bits.

$$P_1 = d_2 \oplus d_3 \oplus d_4$$

$$P_2 = d_1 \oplus d_3 \oplus d_4$$

$$P_3 = d_1 \oplus d_2 \oplus d_4$$




the network for $P_2$ can be design as the above figure. for $P_1$ and $P_3$ the network can be designed similarly. the activation function for the neurons is the step function. in total we have 4 neurons in the input layer, 3 neurons in the output layer, and 12 neurons in the hidden layer. the weights are as follows:

$$W_1 = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

$$W_2 = \begin{bmatrix} 1 & 1 & -1 & -1 \\ 0 & 0 & 0 & 0 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

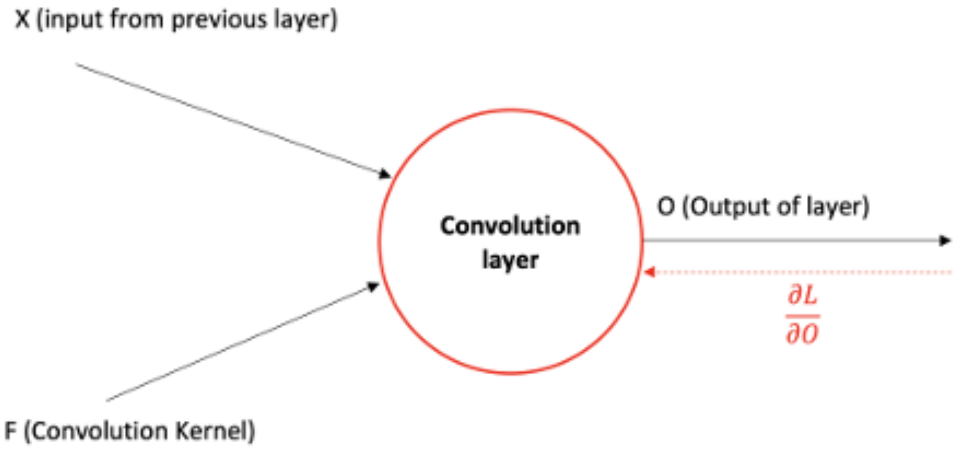
$$W_3 = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & 1 & -1 \\ 1 & -1 & -1 & 1 \end{bmatrix}$$

so we have:

$$W = \begin{bmatrix} W_1 & W_2 & W_3 \end{bmatrix}$$

# Back propagation in CNN - Convolution in each direction!

In this question, we are going to do back propagation 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 **Forward Pass** for convolution layers are 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 2 matrices can arbitrarily change. Prove that:

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

where $*$ sign is Convolution operation and $\circledast$ is full convolution.

It is worth noting that $*$ sign is Convolution operation and $\circledast$ is full convolution.

solution

first lets prove the first statement($\frac{\partial L}{\partial F} = X * \frac{\partial L}{\partial O}$):
we know that $O_{i,j} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X_{i+m,j+n} F_{m,n}$ and $L = \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} L_{i,j}$ so we can write:

$$\frac{\partial L}{\partial F_{m,n}} = \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} \frac{\partial L}{\partial O_{i,j}} \frac{\partial O_{i,j}}{\partial F_{m,n}} = \sum_{i=0}^{I-1} \sum_{j=0}^{J-1} \frac{\partial L}{\partial O_{i,j}} X_{i+m,j+n}$$

the above equation is the same as the convolution operation so we can write:

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

now lets prove the second statement($\frac{\partial L}{\partial X} = F \circledast \frac{\partial L}{\partial O}$): we can rewrite $\frac{\partial L}{\partial X_{i,j}}$ as below:

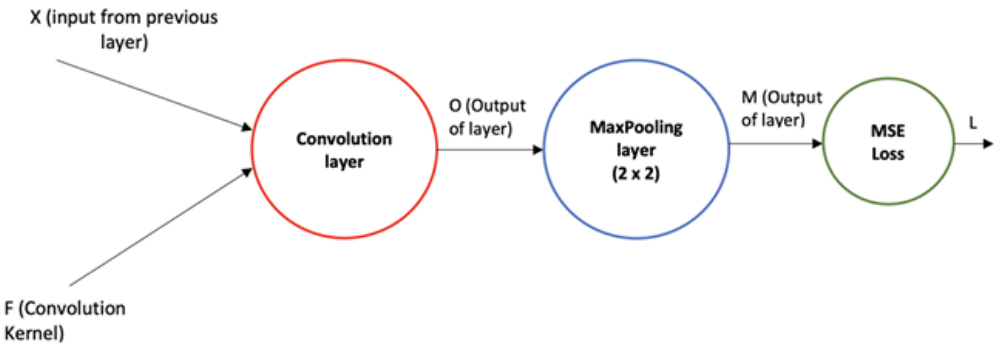
$$\frac{\partial L}{\partial X_{i,j}} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \frac{\partial L}{\partial O_{i-m,j-n}} F_{m,n}$$

the right side of the above equation comes from the fact that $O_{i-m,j-n} = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} X_{i,j} F_{m,n}$ so $\frac{\partial O_{i-m,j-n}}{\partial X_{i,j}} = F_{m,n}$.
the above equation is the same as the full convolution operation so we can write:

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

# Back propagation in CNN - an example

In this question, we are going to do **back propagation** operation on a sequence of layers, shown below; and write the update rules (and update weights) for each of the weights using **gradient descent**.




Here, please write the update rules and then update weights corresponding to variables/matrices $X, F, O, M$. The required information is provided below:

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

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

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

Where $\mathscr{L}$ is the matrix to calculate MSE loss with.

solution

we use the expressions of previous question:

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

first we calculate $\frac{\partial L}{\partial O}$ and then we calculate $\frac{\partial L}{\partial F}$ and update $F$. for the calculation of $\frac{\partial L}{\partial O}$ we need to calculate $O$ and $M$. so we have:

$$O = X \overset{\text{valid}}{*} F = \begin{bmatrix} 9 & 39 & -35 & -47 \\ 16 & 46 & -16 & -23 \\ 29 & 71 & -29 & -48 \\ 40 & 88 & -66 & -93 \end{bmatrix} \xrightarrow{\text{max pooling}} M = \begin{bmatrix} 46 & 46 & -16 \\ 71 & 71 & -16 \\ 88 & 88 & -29 \end{bmatrix}$$

$$L = \frac{1}{n}(\mathscr{L} - M)^T(\mathscr{L} - M) \Rightarrow \frac{\partial L}{\partial M} = \frac{2}{n}(M - \mathscr{L})$$

$$\Rightarrow \frac{\partial L}{\partial M} = \frac{2}{9}\begin{bmatrix} 45 & 46 & -16 \\ 71 & 70 & -16 \\ 88 & 88 & -30 \end{bmatrix} = \begin{bmatrix} 10 & 10.22 & -3.56 \\ 15.78 & 15.56 & -3.56 \\ 19.56 & 19.56 & -6.67 \end{bmatrix}$$

now we calculate $\frac{\partial L}{\partial O}$:
Initializing $\frac{\partial L}{\partial O}$ as a matrix of zeros with the same dimensions as $O$.
For each element in $M$ (indexed by $i, j$), find the position $(k, l)$ in $O$ that was the maximum in its pooling region.

$$\Rightarrow \frac{\partial L}{\partial O_{k,l}} = \frac{\partial L}{\partial M_{i,j}}$$

$$\Rightarrow \frac{\partial L}{\partial O} = \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 20.22 & -7.12 & 0 \\ 0 & 31.34 & -6.67 & 0 \\ 0 & 39.12 & 0 & 0 \end{bmatrix}$$

so we can calculate $\frac{\partial L}{\partial F}$ and update $F$:

$$\frac{\partial L}{\partial F} = \begin{bmatrix} 835.02 & -156.48 & -476.64 \\ 952.38 & -254.26 & -743.72 \\ 1078.21 & -327.73 & -1123.1 \end{bmatrix}$$

$$F^{(\text{new})} = F^{(\text{old})} - \eta \frac{\partial L}{\partial F}$$

for example if we set $\eta = 0.01$, we have:

$$F^{(\text{new})} = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} - 0.01 \begin{bmatrix} 835.02 & -156.48 & -476.64 \\ 952.38 & -254.26 & -743.72 \\ 1078.21 & -327.73 & -1123.1 \end{bmatrix} = \begin{bmatrix} -7.35 & 1.56 & 3.77 \\ -7.52 & 2.54 & 5.44 \\ -9.78 & 3.28 & 10.23 \end{bmatrix}$$

## CNNs are universal approximators

We know that CNNs are universal approximators. It means that any function can be approximated using CNNs. We know the same thing about MLPs and we know MLPs are universal approximators, too. For more information on this subject and seeing the proof for these statements, see this link and this link.

Now, we are looking to find out that if we can make an equivalent MLP from a CNN or not? If yes, please explain and say under what situation and circumstance we can find out the equivalent MLP. If not, please explain why under no circumstances, we can not find an equivalent MLP.

(For the sake of simplicity, you can explain your reasons and/or ideas on a 2D image $X$ with dimensions $3 \times 3 \times 1$ and kernel $F$ with dimensions $2 \times 2$ and then, explain how can your idea/explanations be generalized to higher dimensions)

solution

first we consider a 2D image $X$ with dimensions $3 \times 3 \times 1$ and kernel $F$ with dimensions $2 \times 2$.

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} \\ x_{2,1} & x_{2,2} & x_{2,3} \\ x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix} \quad F = \begin{bmatrix} f_{1,1} & f_{1,2} \\ f_{2,1} & f_{2,2} \end{bmatrix}$$

using the convolution operation, we can calculate the output of the convolutional layer as:

$$O = X * F = \begin{bmatrix} o_{1,1} & o_{1,2} \\ o_{2,1} & o_{2,2} \end{bmatrix} =$$

$$\begin{bmatrix} x_{1,1}f_{1,1} + x_{1,2}f_{1,2} + x_{2,1}f_{2,1} + x_{2,2}f_{2,2} & x_{1,2}f_{1,1} + x_{1,3}f_{1,2} + x_{2,2}f_{2,1} + x_{2,3}f_{2,2} \\ x_{2,1}f_{1,1} + x_{2,2}f_{1,2} + x_{3,1}f_{2,1} + x_{3,2}f_{2,2} & x_{2,2}f_{1,1} + x_{2,3}f_{1,2} + x_{3,2}f_{2,1} + x_{3,3}f_{2,2} \end{bmatrix}$$

as we know the input of the MLP is the flattened version of the input image so the input of MLP is:

$$X_{flattened} = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & x_{2,1} & x_{2,2} & x_{2,3} & x_{3,1} & x_{3,2} & x_{3,3} \end{bmatrix}$$

now to construct the equivalent MLP, we need to find the weights of the MLP. the below equation shows the relation in MLP:

$$O = W \cdot X_{flattened}^{T} + b$$

so we can construct the weight matrix $W$ as:

$$W = \begin{bmatrix} f_{11} & f_{12} & 0 & f_{21} & f_{22} & 0 & 0 & 0 & 0 \\ 0 & f_{11} & f_{12} & 0 & f_{21} & f_{22} & 0 & 0 & 0 \\ 0 & 0 & 0 & f_{11} & f_{12} & 0 & f_{21} & f_{22} & 0 \\ 0 & 0 & 0 & 0 & f_{11} & f_{12} & 0 & f_{21} & f_{22} \end{bmatrix}$$

so we can see that we can construct an equivalent MLP from a CNN. in a more general case, we can construct an equivalent MLP from a CNN by flattening the input and constructing the weight matrix as shown above. for some CNNs, we will need to add a bias term to the MLP to make it equivalent to the CNN.
for example for a 3D image $\boldsymbol{X}$ and a 3D kernel $\boldsymbol{F}$, we can flatten the input and construct the weight matrix as shown above. in this case, we will need to add a bias term to the MLP to make it equivalent to the CNN.

# Backpropagation Algorithm2

Consider the following convolutional network with given layers.

$$\textbf{Layer1 : Convolutional} \begin{cases} \text{Input :} & \mathbf{x} = \begin{bmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{bmatrix} \\ \text{Filter :} & \mathbf{v}_1 = \begin{bmatrix} w_1 & w_2 \end{bmatrix} \\ \text{Output :} & z_1 = \tanh(\mathbf{x} * \mathbf{v}_1) \end{cases}$$

$$\textbf{Layer2 : Fully-connected} \begin{cases} \text{Input :} & z_1 \\ \text{Filter :} & \mathbf{v}_2 = \begin{bmatrix} w_{31} & w_{32} \\ w_{41} & w_{42} \end{bmatrix} \\ \text{Output :} & z_2 = \boldsymbol{\sigma}(\mathbf{v}_2 \times z_1) \end{cases}$$

$$\textbf{Layer3 : Fully-connected} \begin{cases} \text{Input :} & z_2 \\ \text{Filter :} & \mathbf{v}_3 = \begin{bmatrix} w_5 & w_6 \end{bmatrix} \\ \text{Output :} & y^* = \mathbf{v}_3 \times z_2 \end{cases}$$

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

solution

we can say that:

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

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y^*} \times \frac{\partial y^*}{\partial z_2} \times \frac{\partial z_2}{\partial z_1} \times \frac{\partial z_1}{\partial w_1}$$

we have:

$$z_1 = \begin{bmatrix} z_{11} \\ z_{12} \end{bmatrix} = \begin{bmatrix} \tanh(x_{11}w_1 + x_{12}w_2) \\ \tanh(x_{21}w_1 + x_{22}w_2) \end{bmatrix}$$

in the second layer we have:

$$z_2 = \begin{bmatrix} z_{21} \\ z_{22} \end{bmatrix} = \begin{bmatrix} \boldsymbol{\sigma}(w_{31}z_{11} + w_{32}z_{12}) \\ \boldsymbol{\sigma}(w_{41}z_{11} + w_{42}z_{12}) \end{bmatrix}$$

and in the third layer we have:

$$y^* = w_5 z_{21} + w_6 z_{22}$$

so we can say:

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

$$\frac{\partial y^*}{\partial z_2} = \begin{bmatrix} w_5 \\ w_6 \end{bmatrix}$$

$$\frac{\partial z_2}{\partial z_1} = \begin{bmatrix} \boldsymbol{\sigma}'(w_{31}z_{11} + w_{32}z_{12})w_{31} & \boldsymbol{\sigma}'(w_{31}z_{11} + w_{32}z_{12})w_{32} \\ \boldsymbol{\sigma}'(w_{41}z_{11} + w_{42}z_{12})w_{41} & \boldsymbol{\sigma}'(w_{41}z_{11} + w_{42}z_{12})w_{42} \end{bmatrix}$$

where we know that:

$$\sigma'(x) = \sigma(x)(1-\sigma(x))$$

$$\frac{\partial z_1}{\partial w_1} = \begin{bmatrix} x_{11}(1-\tanh^2(x_{11}w_1+x_{12}w_2)) \\ x_{21}(1-\tanh^2(x_{21}w_1+x_{22}w_2)) \end{bmatrix}$$

so we can say:

$$\frac{\partial L}{\partial w_1} = 2(y^*-y)\begin{bmatrix} w_5 \\ w_6 \end{bmatrix}\begin{bmatrix} \sigma'(w_{31}z_{11}+w_{32}z_{12})w_{31} & \sigma'(w_{31}z_{11}+w_{32}z_{12})w_{32} \\ \sigma'(w_{41}z_{11}+w_{42}z_{12})w_{41} & \sigma'(w_{41}z_{11}+w_{42}z_{12})w_{42} \end{bmatrix}$$

$$\begin{bmatrix} x_{11}(1-\tanh^2(x_{11}w_1+x_{12}w_2)) \\ x_{21}(1-\tanh^2(x_{21}w_1+x_{22}w_2)) \end{bmatrix}$$

so we can say that:

$$\begin{aligned} \tfrac{\partial L}{\partial w_1} = 2(y^*-y)[&w_5\sigma'(w_{31}z_{11}+w_{32}z_{12})(w_{31}z_{11}(1-\tanh^2(x_{11}w_1+x_{12}w_2)) \\ &+w_{32}z_{21}(1-\tanh^2(x_{21}w_1+x_{22}w_2))) + w_6\sigma'(w_{41}z_{11}+w_{42}z_{12})(w_{41}z_{11}(1- \\ &\tanh^2(x_{11}w_1+x_{12}w_2)) + w_{42}z_{21}(1-\tanh^2(x_{21}w_1+x_{22}w_2)))] \end{aligned}$$

# Optimizing Deep Learning Models

In this question, we are going to discuss the *Momentum* in optimization algorithms.

## *Momentum in Optimization*

Recall the *Gradient Descent* algorithm:

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

Where $\theta_t$ is the parameters at time $t$, $\alpha$ is the learning rate, and $\nabla f(\theta_t)$ is the gradient of the loss function with respect to the parameters.

The *Momentum* algorithm is an extension of the *Gradient Descent* algorithm that adds a momentum term to the update rule:

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

Where $v_t$ is the momentum term, and $\beta$ is the momentum parameter.

## *Questions 1*

The *Nesterov Accelerated Gradient* (NAG) algorithm is an extension of the *Momentum* algorithm that adds a correction term to the update rule:

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

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

solution

The NAG algorithm modifies the standard Momentum algorithm by incorporating a lookahead approach. Instead of calculating the gradient at the current parameters, NAG calculates the gradient at the estimated future position of the parameters.

**Nesterov Accelerated Gradient (NAG):**

The Nesterov Accelerated Gradient (NAG) algorithm improves upon the Momentum algorithm by introducing a lookahead step that calculates the gradient at the future position of the parameters. This results in faster convergence, improved stability, and more adaptive updates during the training process. The anticipatory nature of NAG helps in making more informed and precise updates, thus enhancing the overall optimization efficiency.

1. **Lookahead step:** Compute the gradient at the anticipated future position:

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

2. **Update the parameters:**

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

**Explanation and Improvements**

1. **Lookahead Gradient Calculation:** In the Momentum algorithm, the gradient is calculated at the current position, which may lead to overshooting if the momentum term is large.
In NAG, the gradient is calculated at the anticipated future position, considering the momentum term, which provides a more accurate direction of where the parameters are heading. This "lookahead" helps in reducing the chances of overshooting and provides a corrective mechanism.
2. **Faster Convergence:** NAG tends to have faster convergence compared to the Momentum algorithm. The lookahead step allows the algorithm to make more informed updates, thus reducing the number of iterations required to reach the optimal solution.
3. **Improved Stability:** By considering the future position, NAG helps in smoothing out the oscillations that might occur in the Momentum algorithm. This makes the training process more stable and helps in converging to the minimum more reliably.
4. **Adaptive Updates:** The lookahead mechanism in NAG adapts the update direction based on the anticipated future state, making it more responsive to the contours of the loss surface. This adaptability helps in navigating the loss landscape more effectively, especially in scenarios with high curvature.

## ***Questions 2***

The *Adagrad* algorithm is an adaptive learning rate optimization algorithm that scales the learning rate based on the historical gradients:

$$g_{t+1} = g_t + (\nabla f(\theta_t))^2$$
$$\theta_{t+1} = \theta_t - \frac{\alpha}{\sqrt{g_{t+1}} + \varepsilon} \nabla f(\theta_t)$$

Explain how the Adagrad algorithm works and how it improves the training process compared to the *Momentum* algorithm.

solution

The Adagrad (Adaptive Gradient Algorithm) is an optimization method that adapts the learning rate for each parameter individually, based on the gradients' historical squared values. This approach is particularly useful for handling sparse data, common in fields like natural language processing and image recognition.

Adagrad modifies the general learning rate at each time step for each parameter, based on the past gradients that have been computed for that parameter. Here's how the algorithm updates the parameters:
1. **Accumulate the Square of the Gradients:** For each parameter, accumulate the square of the gradients:

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

Here, $g_t$ represents the sum of the squares of the gradients up to time $t$ for each parameter.
2. **Parameter Update:** Adjust the parameters using the formula:

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

- $\alpha$ is the initial learning rate.
- $\varepsilon$ is a small smoothing term added to the denominator to avoid division by zero (typically on the order of 1e-8).

**Key Features of Adagrad**

1. **Adaptive Learning Rates:** Each parameter has its own learning rate, which decreases monotonically over time.
2. **Automatic Handling of Sparse Data:** Adagrad is particularly effective in scenarios where some features are observed infrequently.

**Improvements Over the Momentum Algorithm** Adagrad provides specific improvements over traditional methods like the Momentum algorithm in several key ways:

1. **Eliminates Need for Manual Tuning of the Learning Rate:** Since Adagrad adapts the learning rate to each parameter based on historical data, it reduces the need for manual tuning of the learning rate.
2. **Better Handling of Sparse Data:** Adagrad can be particularly effective at navigating the challenges of sparse data, as its adaptive learning rate increases the model's sensitivity to less frequent features.
3. **Prevents Overshooting in the Late Stages of Training:** As the learning rate is effectively reduced over time, it helps prevent overshooting the minima, which can be a problem with Momentum when the learning rate is not adequately decreased.

**Limitations of Adagrad** Despite its benefits, Adagrad has limitations, particularly in deep learning contexts:

1. **Aggressive Learning Rate Reduction:** The learning rate can become excessively small, potentially causing the algorithm to stop learning before reaching the global optimum.
2. **Inadequate for Non-Sparse Data:** In scenarios with dense data, the rapid decrease in learning rate may hinder the model's performance.

## *Adam Optimizer*

The *Adam* optimizer is an adaptive learning rate optimization algorithm that combines the ideas of *Momentum* and *Adagrad*. Here is the update rule of the Adam optimizer:

$$
\begin{aligned}
g_t &= \nabla f(\theta_{t-1}) \\
m_t &= \beta_1 m_{t-1} + (1-\beta_1) g_t \\
v_t &= \beta_2 v_{t-1} + (1-\beta_2) g_t^2 \\
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t} \\
\theta_t &= \theta_{t-1} - \alpha \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}
\end{aligned}
$$

## *part 1*

Explain how the *Adam* optimizer works and how it improves the training process compared to the *Momentum* algorithm, line by line.

solution

The Adam (Adaptive Moment Estimation) optimizer combines the advantages of both Momentum and Adagrad by incorporating the concepts of momentum for acceleration and adaptive learning rates. Here's a detailed explanation of how the Adam optimizer works and how it improves the training process compared to the Momentum algorithm.

**Update Rule of Adam**

The update rules for the Adam optimizer are as follows: 1. **Gradient Calculation:**

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

$g_t$: Gradient of the loss function with respect to the parameters at time step $t$.

2. **First Moment Estimate (Momentum term):**

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

$m_t$: Exponentially decaying average of past gradients (similar to the velocity term in Momentum).

$\beta_1$: Decay rate for the first moment, typically close to 1 (e.g., 0.9).

3. **Second Moment Estimate (RMSprop-like term):**

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

$v_t$: Exponentially decaying average of past squared gradients.

$\beta_2$: Decay rate for the second moment, typically close to 1 (e.g., 0.999).

4. **Bias-Corrected First Moment Estimate:**

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$\hat{m}_t$: Bias-corrected first moment estimate to counteract the initialization bias towards zero.

5. **Bias-Corrected Second Moment Estimate:**

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

$\hat{v}_t$: Bias-corrected second moment estimate to counteract the initialization bias towards zero.

6. **Parameter Update:**

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

$\alpha$: Learning rate.

$\varepsilon$: Small constant to prevent division by zero (typically $10^{-8}$).

**How Adam Improves Training Compared to Momentum**

1. **Adaptive Learning Rate:** Momentum uses a fixed learning rate for all parameters, which can be suboptimal if the gradients vary significantly in magnitude.Adam adapts the learning rate for each parameter based on the first and second moment estimates, allowing for more nuanced updates. This results in more efficient convergence, especially in the presence of sparse gradients.
2. **Bias Correction:** Momentum lacks explicit mechanisms to correct the bias introduced in the initial stages of training.Adam includes bias correction terms for the first and second moment estimates, which helps in stabilizing the learning process early in training. This ensures that the optimizer does not underestimate the effective learning rate.
3. **Combining Momentum and RMSprop:** Momentum relies solely on the past gradients' exponentially decaying average, which helps smooth out the updates but may be insufficient for highly non-stationary data.Adam combines the benefits of Momentum (by using first moment estimates) and RMSprop (by using second moment estimates), providing a more robust optimization process that can handle noisy and sparse gradients effectively.

## *part 2*

Explain why $m_t$ have a bias towards zero in the early stages of training and how $\hat{m}_t$ corrects this bias.

solution

below equations hold in the early stages of training:

$$
\begin{aligned}
m_0 &= 0 \\
m_1 &= \beta_1 m_0 + (1-\beta_1) g_1 = (1-\beta_1) g_1 \\
m_2 &= \beta_1 m_1 + (1-\beta_1) g_2 = \beta_1 (1-\beta_1) g_1 + (1-\beta_1) g_2 \\
m_3 &= \beta_1 m_2 + (1-\beta_1) g_3 = \beta_1^2 (1-\beta_1) g_1 + \beta_1 (1-\beta_1) g_2 + (1-\beta_1) g_3 \\
m_t &= (1-\beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i
\end{aligned}
$$

As you can see, the further we go expanding the value of $m$, the less first values of gradients contribute to the overall value, as they get multiplied by smaller and smaller beta. Capturing this pattern, we can rewrite the formula for our moving average:

$$m_t = (1-\beta_1) \sum_{i=0}^{t} \beta_1^i g_{t-i}$$

To calculate bias, we calculate expected value:

$$
\begin{aligned}
E[m_t] &= E[(1-\beta_1) \sum_{i=1}^{t} \beta_1^{t-i} g_i] \\
&= E[g_t](1-\beta_1) \sum_{i=1}^{t} \beta_1^{t-i} + \varepsilon \\
&= E[g_t](1-\beta_1) \frac{1}{1-\beta_1} \\
&= E[g_t] + \varepsilon
\end{aligned}
$$

Because the approximation is taking place, the error $\varepsilon$ emerge in the formula ($\varepsilon$ is a very small number). Since $m$ and $v$ are estimates of first and second moments, we want to have the following property:

$$
\begin{aligned}
E[m_t] &= E[g_t] \\
E[v_t] &= E[g_t^2]
\end{aligned}
$$

If we use

$$
\begin{aligned}
\hat{m}_t &= \frac{m_t}{1-\beta_1^t} \\
\hat{v}_t &= \frac{v_t}{1-\beta_2^t}
\end{aligned}
$$

We have:

$$E[\hat{m}_t] = E\left[\frac{m_t}{1-\beta_1^t}\right] = \frac{E[g_t](1-\beta_1^t) + \varepsilon}{1-\beta_1^t} = E[g_t] + \text{very small number}$$

1. **Bias Towards Zero:** $m_t$ and $v_t$ are initialized at zero and are calculated using exponential moving averages. Early in training, especially when $t$ is small, $m_t$ and $v_t$ will be biased towards zero because the moving averages are dominated by their initial values (zeros).

2. **Correction:** To correct this bias, Adam incorporates bias-corrected estimates $\hat{m}_t$ and $\hat{v}_t$. The bias correction formulas are:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

These corrections effectively normalize $m_t$ and $v_t$, compensating for their initial bias towards zero, especially in the early stages of training. This ensures that the estimates accurately reflect the actual magnitudes of the gradients and their squares.

# Quadratic Error Function

Consider a quadratic error function of the form

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

where $\mathbf{w}^*$ represents the minimum, and the Hessian matrix $\mathbf{H}$ is positive definite and constant. Suppose the initial weight vector $\mathbf{w}(0)$ is chosen to be at the origin and is updated using simple gradient descent

$$\mathbf{w}^\tau = \mathbf{w}^{\tau-1} - \rho \nabla E$$

where $\tau$ denotes the step number, and $\rho$ is the learning rate (which is assumed to be small). Show that, after $\tau$ steps, the components of the weight vector parallel to the eigenvectors of $\mathbf{H}$ can be written

$$w_j^\tau = \{1 - (1 - \rho\eta_j)^\tau\} w_j^*$$

where $w_j = \mathbf{w}^T \mathbf{u}_j$ and $\mathbf{u}_j$ and $\eta_j$ are the eigenvectors and eigenvalues, respectively, of $\mathbf{H}$ so that

$$\mathbf{H}\mathbf{u}_j = \eta_j \mathbf{u}_j$$

Show that as $\tau \to \infty$, this gives $\mathbf{w}^\tau \to \mathbf{w}^*$ as expected, provided $|1 - \rho\eta_j| < 1$. Now suppose that training is halted after a finite number $\tau$ of steps. Show that the components of the weight vector parallel to the eigenvectors of the Hessian satisfy

$$\mathbf{w}_j^{(\tau)} \approx \mathbf{w}_j^* \quad \text{when} \quad \eta_j \gg (\rho\tau)^{-1}$$

$$|\mathbf{w}_j^{(\tau)}| \ll |\mathbf{w}_j^*| \quad \text{when} \quad \eta_j \ll (\rho\tau)^{-1}$$

solution

first we have:

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

so we can state that:

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

from the gradient descent formula we have:

$$\mathbf{w}^\tau = \mathbf{w}^{\tau-1} - \rho \nabla E$$

so we can write:

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

since H is symmetric and positive definite, we can write that as:

$$H = U\Lambda U^T$$

where U is the matrix of eigenvectors and $\Lambda$ is the diagonal matrix of eigenvalues. so we can write:

$$\mathbf{w}^\tau = \mathbf{w}^{\tau-1} - \rho U\Lambda U^T(\mathbf{w}^{\tau-1} - \mathbf{w}^*)$$

$$\Rightarrow \mathbf{w}^{\tau} = U(I-\rho\Lambda)U^T\mathbf{w}^{\tau-1} + \rho U\Lambda U^T\mathbf{w}^*$$

as we know U is orthogonal, so we can write:

$$\mathbf{w}^{\tau} = U(I-\rho\Lambda)U^T\mathbf{w}^{\tau-1} + \rho\Lambda\mathbf{w}^*$$

$$\Rightarrow \mathbf{w}_j^{\tau} = \mathbf{w}^{\tau^T}\mathbf{u}_j = \mathbf{u}_j^T\mathbf{w}^{\tau} = \mathbf{u}_j^T U(I-\rho\Lambda)U^T\mathbf{w}^{\tau-1} + \rho\mathbf{u}_j^T\Lambda\mathbf{w}^*$$

$$\Rightarrow \mathbf{w}_j^{\tau} = (1-\rho\eta_j)\mathbf{w}_j^{\tau-1} + \rho\eta_j\mathbf{w}_j^*$$

for $\tau = 1$ we have:

$$\mathbf{w}_j^1 = (1-\rho\eta_j)\mathbf{w}_j^0 + \rho\eta_j\mathbf{w}_j^* = \rho\eta_j\mathbf{w}_j^*$$

for $\tau = 2$ we have:

$$\mathbf{w}_j^2 = (1-\rho\eta_j)\mathbf{w}_j^1 + \rho\eta_j\mathbf{w}_j^* = (1-\rho\eta_j)\rho\eta_j\mathbf{w}_j^* + \rho\eta_j\mathbf{w}_j^*$$

so at $\tau$ steps we have:

$$w_j^{\tau} = \rho\eta_j w_j^* + (1-\rho\eta_j)\rho\eta_j w_j^* + (1-\rho\eta_j)^2\rho\eta_j w_j^* + \cdots + (1-\rho\eta_j)^{\tau-1}\rho\eta_j w_j^*$$

if : $S = a + ar + ar^2 + \cdots + ar^{n-1}$ is given by:

$$S = a\frac{1-r^n}{1-r}$$

$$w_j^{\tau} = \rho\eta_j w_j^* \frac{1-(1-\rho\eta_j)^{\tau}}{1-(1-\rho\eta_j)}$$

so we can write:

$$w_j^{\tau} = w_j^*\{1-(1-\rho\eta_j)^{\tau}\}$$

so if $|1-\rho\eta_j| < 1$ then as $\tau \to \infty$ we have:

$$w_j^{\tau} \to w_j^*$$

now we need to show that:

$$\mathbf{w}_j^{(\tau)} \approx \mathbf{w}_j^* \quad \text{when} \quad \eta_j \gg (\rho\tau)^{-1}$$

$$|\mathbf{w}_j^{(\tau)}| \ll |\mathbf{w}_j^*| \quad \text{when} \quad \eta_j \ll (\rho\tau)^{-1}$$

to prove the firs statement we can write: as $\eta_j \gg (\rho\tau)^{-1}$ the expression $(1-\rho\eta_j)^{\tau}$ approches to zero rapidly. so we can write:

$$w_j^{\tau} = w_j^*(1-(1-\rho\eta_j)^{\tau}) \approx w_j^*$$

now for the second statement we can write: when $\eta_j \ll (\rho\tau)^{-1}$ the expression $(1-\rho\eta_j)^{\tau}$ is close to 1. so we can write:

$$(1-\rho\eta_j)^{\tau} \approx 1-\rho\eta_j\tau$$

so we can write:

$$w_j^\tau = w_j^*(1-(1-\rho\eta_j)^\tau) \approx w_j^*\rho\eta_j\tau$$

so as $\rho\eta_j\tau \ll 1$ we have:

$$w_j^\tau \ll w_j^*$$

# LSTM

Consider a standard LSTM cell with input dimension $n$, hidden state dimension $m$, and forget gate, input gate, and output gate activation functions denoted as $f(t)$, $i(t)$, and $o(t)$ respectively. Let $W_f$, $W_i$, and $W_o$ represent the weight matrices associated with these gates. Additionally, let $U_f$, $U_i$, and $U_o$ represent the recurrent weight matrices.

Given an input sequence $x(t)$, an initial hidden state $h(0)$, and the LSTM equations:

$$
\begin{aligned}
f(t) &= \sigma(W_f x(t) + U_f h(t-1) + b_f) \\
i(t) &= \sigma(W_i x(t) + U_i h(t-1) + b_i) \\
o(t) &= \sigma(W_o x(t) + U_o h(t-1) + b_o) \\
\tilde{c}(t) &= \tanh(W_x x(t) + U_h h(t-1) + b_c) \\
c(t) &= f(t) \odot c(t-1) + i(t) \odot \tilde{c}(t) \\
h(t) &= o(t) \odot \tanh(c(t))
\end{aligned}
$$

where $\sigma$ is the sigmoid function, $\odot$ denotes element-wise multiplication, and $\tilde{c}(t)$ is the candidate cell state.

Prove that the LSTM cell can learn to 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 as $t$ increases.

solution

To comprehend why the full gradient approach in LSTM helps avoid the vanishing gradient problem, we need to delve into the recursive gradient analysis. Let's expand the full derivative for $\frac{\partial C_t}{\partial C_{t-1}}$ to understand the dynamics.
Recall that in an LSTM, the cell state $C_t$ depends on the forget gate $f_t$, input gate $i_t$, and candidate cell state $\tilde{C}_t$, all of which are functions of the previous cell state $C_{t-1}$ and the hidden state $h_{t-1}$. Using the multivariate chain rule, we can express the derivative as follows:

$$\frac{\partial C_t}{\partial C_{t-1}} = \frac{\partial C_t}{\partial f_t} \cdot \frac{\partial f_t}{\partial C_{t-1}} + \frac{\partial C_t}{\partial i_t} \cdot \frac{\partial i_t}{\partial C_{t-1}} + \frac{\partial C_t}{\partial \tilde{C}_t} \cdot \frac{\partial \tilde{C}_t}{\partial C_{t-1}} + \frac{\partial C_t}{\partial C_{t-1}}$$

Breaking down these derivatives, we get:

$$\frac{\partial C_t}{\partial C_{t-1}} = f_t + i_t \cdot \tanh'(W_c x_t + U_c h_{t-1} + b_c) \cdot W_c + \tilde{C}_t \cdot \sigma'(W_i x_t + U_i h_{t-1} + b_i) \cdot W_i$$

For the backpropagation through time, if we consider $k$ time steps, we multiply similar terms recursively. Unlike vanilla RNNs, where terms $\frac{\partial h_t}{\partial h_{t-1}}$ may tend to zero or infinity causing vanishing/exploding gradients, LSTM gates help maintain gradients in a controlled manner.
The forget gate $f_t$ and input gate $i_t$ can be close to zero or one, which helps in preserving the gradient flow. The candidate cell state $\tilde{C}_t$ is also controlled by the tanh function, which helps in maintaining the gradient. Thus, the LSTM architecture can learn to remember information over long sequences by preventing the vanishing gradient problem.

# GANs

In this question, we are going to discuss GANs and their training process.

## *GANs, a Game Theoretic Approach*

Game theory is a branch of mathematics that deals with the analysis of games. In this question, we are going to discuss GANs from a game theoretic approach. A game is defined by the following elements:

- **Players:** The agents who are playing the game; row player and column player in a two-player game.
- **Actions:** The actions that each player can take; the strategies of each player.
- **Payoffs:** The rewards that each player receives based on the actions taken by all players; the utility function $u_i(a_1, a_2)$ for player $i$ in a two-player game where $a_1$ and $a_2$ are the actions taken by player 1 and player 2, respectively.

Nash Equilibrium is a concept in game theory that describes a situation in which no player can increase their payoff by changing their strategy, assuming that all other players keep their strategies unchanged. For example, in a two-player game, a Nash Equilibrium is a pair of strategies, one for each player, such that each player's strategy is the best response to the other player's strategy.

For example, consider a two-player game called Prisoner's Dilemma. In this game, two players can either Cooperate or Defect. The payoffs for each player are as follows:

- If both players Cooperate, they both receive a payoff of 3,
- If one player Cooperates and the other Defects, the player who Cooperates receives a payoff of 0, and the player who Defects receives a payoff of 5,
- If both players Defect, they both receive a payoff of 1.

We can define the Prisoner's Dilemma as a game with the following matrix:

| | Cooperate | Defect |
|---|---|---|
| Cooperate | 3,3 | 0,5 |
| Defect | 5,0 | 1,1 |

Now, let's discuss how to find the Nash Equilibrium of a game. The Nash Equilibrium of a game can be found by solving the following optimization problem:

$$\max_{a_1} \min_{a_2} u_1(a_1, a_2)$$

$$\max_{a_2} \min_{a_1} u_2(a_2, a_1)$$

Where $a_1$ and $a_2$ are the actions taken by player 1 and player 2, respectively, and $u_1$ and $u_2$ are the payoffs of player 1 and player 2, respectively.

Now, let's define GANs as a game. In a GAN, we have two players, the Generator and the Discriminator. The Generator tries to generate samples that are similar to the real data, while the Discriminator tries to distinguish between the real and generated samples.

## *Question 1*

Define the GANs as a game. What are the players, actions, and payoffs in this game?

solution

**Players:** The Generator and the Discriminator
- **Generator:** Tries to generate samples that are similar to the real data
- **Discriminator:** Tries to distinguish between the real and generated samples and maximize the probability of assigning the correct label to the samples

**Actions:** The actions that each player can take
- **Generator:** Generates fake samples from a noisy distribution
- **Discriminator:** Classifies the samples as real or generated to know a sample is fake or not

**Payoffs:** The rewards that each player receives based on the actions taken by all players
- **Generator:** The Generator receives a payoff based on how well the Discriminator is fooled by the generated samples
- **Discriminator:** The Discriminator receives a payoff based on how well it can distinguish between the real and generated samples

## *Question 2*

To find the Nash Equilibrium of the GANs game, we need to solve the following optimization problem:

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

Explain this optimization problem and how it relates to the training process of the GANs.

solution

The optimization problem of the GANs game is to find the Nash Equilibrium of the game. The Generator tries to minimize the probability of the Discriminator assigning the correct label to the generated samples, while the Discriminator tries to maximize the probability of assigning the correct label to the generated samples. The optimization problem is formulated as a minimax game, where the Generator minimize the loss function, and the Discriminator minimizes the loss function. the term $\mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)]$ in the optimizationn problem represents the loss of the Discriminator when classifying the real samples and help it to maximize the probability of assigning the correct label to the real samples.

on the other hand the term $\mathbb{E}_{z\sim p_z}[\log(1-D(G(z)))]$ represents the loss of the Discriminator when classifying the generated samples and help it to maximize the probability of assigning the correct label to the generated samples. training of GAN is as below:
to train Discriminator, we first sample a batch of real data and a batch of fake data from the Generator. then we generate fake samples from current Generator and we calculate the loss function and update discriminator using GD to minimize the loss function.
on the other hand to train the generator we generate fake samples from the current Generator and we calculate the loss function and update the Generator using GD to maximize the loss function.
this training proccess will continue untill we reach the nash Equilibrium. this means that the Generator generates samples that are similar to the real data and the Discriminator can't distinguish between the real and generated samples.

## ***Question 3***

To solve the optimization problem, we use the calculus of variations. Let $y = y(x)$ be a function of $x$ and the function we need to optimize is:

$$\int F(x, y, y')\, dx$$

For the function $y = y(x)$ that minimizes the integral, we know that:

$$\frac{\partial F}{\partial y} - \frac{d}{dx}\left(\frac{\partial F}{\partial y'}\right) = 0$$

You do not need to prove this formula. Now, solve the optimization problem of the GANs game using the calculus of variations. You need to find a closed-form solution for $\theta_D$.

solution

the optimiazation problem of the GANs game is as below:

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

if we define $F(D(x))$ as below:

$$F(D(x)) = p_{data}(x)\log D(x) + p_z(x)\log(1-D(G(x)))$$

we can reformulate the optimization problem as below:

$$\max_{\theta_G}\min_{\theta_D} \int F(D(x))\, dx$$

to find the optimal $\theta_D$ we need to solve the following equation:

$$\frac{\partial F}{\partial D(x)} = 0$$

so we have:

$$\frac{\partial F}{\partial D(x)} = \frac{\partial}{\partial D(x)}(p_{data}(x)\log D(x) + p_z(x)\log(1-D(x))) = 0$$

$$\frac{\partial}{\partial D(x)}(p_{data}(x)\log D(x) + p_z(x)\log(1-D(x))) = \frac{p_{data}(x)}{D(x)} - \frac{p_z(x)}{1-D(x)} = 0$$

$$\frac{p_{data}(x)}{D(x)} = \frac{p_z(x)}{1-D(x)}$$

$$\Rightarrow \theta_D = \frac{p_{data(x)}}{p_{data(x)} + p_z(x)}$$

## *GANs, You Don't Want to Train Them!*

Training GANs is a challenging task due to the instability of the training process. In this question, we are going to discuss the challenges of training GANs. You can solve question 1 and 2 or you can solve question 3. NO extra points will be given for solving all three questions.

## *Question 1*

Recall the optimization problem of the GANs game:

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

When using Gradient Descent to solve this optimization problem, the first term is the discriminator loss. When you update the parameters of the discriminator, if you take the gradient of the GANs loss with respect to the generator parameters, the first term will not be affected. Explain why the first term is not affected when updating the generator parameters. How does this affect the training process of GANs?

solution

the objective function is as below:

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

when we take gradient of the GANs loss with respect to the discriminator parameters the equation will be:

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

but when we take gradient of the GANs loss with respect to the generator parameters the first term will not be affected. and the gradient will become:

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

This is because the first term, $\mathbb{E}_{x \sim p_{\text{data}}}[\log D(x)]$, only involves real data samples and the discriminator's response to them. It does not depend on the generator parameters $\theta_G$. The generator does not directly influence how real data is classified by the discriminator; it only affects the fake data generated from the noise $z \sim p_z$. The fact that the generator's update step does not affect the term involving real data means that during training, the generator only focuses on improving the generation of fake samples to fool the discriminator. Specifically:

- **Discriminator Update:** When updating the discriminator, both real and fake samples are considered, allowing the discriminator to learn to distinguish between real and generated data.
- **Generator Update:** When updating the generator, only the fake samples' term is involved. The generator improves by making the fake samples more realistic to decrease the discriminator's ability to distinguish them from real samples.

This separation ensures that the generator and discriminator are updated in a way that maintains the adversarial nature of the game. However, it can also lead to instability in training:

## *Question 2*

It's known that the training process of GANs is unstable and can suffer from the mode collapse problem. Explain the mode collapse problem and how it affects the training process of GANs.

solution

The mode collapse problem in GANs occurs when the generator produces a limited variety of outputs, often focusing on a few modes of the data distribution while ignoring others. This leads to a lack of diversity in generated samples, causing the generator to repeatedly generate similar outputs. As a result, the discriminator quickly learns to distinguish between real and generated samples, making training unstable. Mode collapse prevents the generator from learning the true data distribution and hinders the overall effectiveness of the GAN.

## *Question 3*

If you solve the optimization problem of the GANs for the optimal Discriminator, you will get the following optimization problem:

$$u_D(\theta_G, \theta_D) = 2D_{JS}(p_{\text{data}} \| p_{\text{model}}) - 2\log(2)$$

$$= \frac{1}{2} D_{KL}(p_{\text{data}} \| \frac{p_{\text{data}} + p_{\text{model}}}{2}) + \frac{1}{2} D_{KL}(p_{\text{model}} \| \frac{p_{\text{data}} + p_{\text{model}}}{2}) - 2\log(2)$$

Where $D_{JS}$ is the Jensen-Shannon divergence, $D_{KL}$ is the Kullback-Leibler divergence (if you aren't familiar with KL divergence, it's a measure of how one probability distribution is different from a second, reference probability distribution).

Another problem with training GANs is the KL divergence term in the loss function. For better intuition, check the following image:

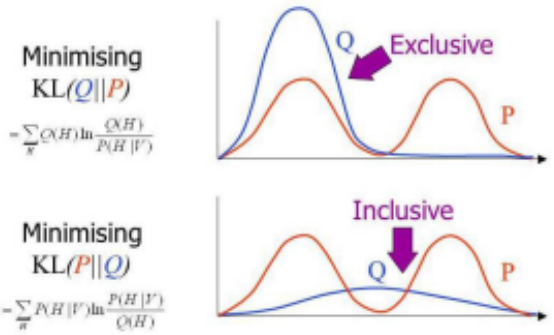



**Figure 1.** $p(x)$, the data distribution and $q(x)$, model distribution

To solve this problem, we can use Wasserstein GANs (WGANs) that replace the KL divergence term with the Wasserstein distance. Explain the Wasserstein distance and how it solves the problem of the KL divergence term in the loss function of GANs.