Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»

Кафедра математики факультету інформатики

Використання методів навчання з підкріпленням для генерації матричних збурень

Текстова частина до курсової роботи

за спеціальністю „Прикладна математика ” 6.040301

Керівник курсової роботи

к.ф.-м.н., ст. в. Швай Н.О.

____

(підпис)

“____” _________ 2020 р.

к. ф.-м.н., доцент

Крюкова Г.В.

Виконав

студент 1 курсу

факультету інформатики

Іванюк-Скульський Б.В.

Міністерство освіти і науки України

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ «КИЄВО-МОГИЛЯНСЬКА АКАДЕМІЯ»
Кафедра інформатики факультету інформатики

ЗАТВЕРДЖУЮ
Зав. кафедри математики,
проф., д.ф.-м.н.
____________Б. В. Олійник
(підпис)
„____” ________________2020 р.

ІНДИВІДУАЛЬНЕ ЗАВДАННЯ
на курсову роботу

студенту Іванюку-Скульському Б.В. факультету інформатики 1-го курсу МП
ТЕМА Використання методів навчання з підкріпленням для генерації змагальних зображень
Зміст ТЧ до курсової роботи:

1. Індивідуальне завдання
2. Календарний план
3. Анотація
4. Вступ
5. РОЗДІЛ 1: Загальна інформація
6. РОЗДІЛ 2: Практичне дослідження моделі
7. Висновки
8. Список використаних джерел

Дата видачі „___” _________ 2020 р. Керівник _______________
(підпис)
Завдання отримав __________
(підпис)

Календарний план виконання роботи:

№ п/п	Назва етапу курсової роботи	Термін виконання етапу	Примітка
1.	Отримання теми курсової роботи.	15.10.2020	
2.	Огляд задачі, підбір вхідних даних	04.05.2020	
3.	Аналіз побудови нейронних мереж	05.05.2020	
4.	Написання перших двох розділів	06.05.2020	
5.	Створення моделі	07.05.2020	
6.	Написання останнього розділу	08.05.2020	
8.	Корегування роботи згідно із зауваженнями керівника	09.05.2020	

Студенту Іванюку-Скульському Б.В.

Керівник Швай Н.О.

“______” _______________

Зміст

Вступ *2*

1. Загальна інформація *3*

1.1.Нейронна мережа як універсальний класифікатор *3*

1.2. Лінійна нейронна мережа *3*

1.3. Конволюційна нейронна мережа *4*

1.4. ResNet *5*

1.5. Функції втрат *6*

1.6. Оптимізатори нейронних мереж *7*

1.6.1 Градієнтний спуск *7*

1.6.2 Стохастичний градієнтний спуск (SGD) *8*

1.6.3 Adam *8*

1.7. Типи атак на конволюційні нейронні мережі для задачі класифікації *10*

1.7.1. Атака білої скриньки *11*

1.7.2. Атака чорної скриньки *12*

1.8. Постановка задачі навчання з підкріпленням *12*

2. Практичне дослідження моделі *16*

2.1. Опис середовища та постановка задачі *16*

2.1.1 Опис даних *16*

2.1.2 Опис можливих дій та винагород *18*

2.2. Тренування агента *19*

2.3. Результати та порівняння з іншими методами *23*

Висновки *26*

Список використаної літератури *27*

Додаток А. *28*

Вступ

Методи машинного та глибинного навчання все частіше використовуються у різних галузях. Прикладами таких галузей можуть бути автономні машини, медицина, важка промисловість та багато інших. Але ці методи машинного навчання вразливі до атак матричних збурень. Атаки матричних збурень - це такі невеликі зміни зображення, невидимі для людського ока, які змінюють передбачення моделі машинного навчання. Дослідження матричних збурень є популярною задачею останнім часом, оскільки вона э ключової для безпеки персональних даних, а також, транспортних засобів [12].

В цій роботі запропоновано новий метод генерації матричних збурень, заснований на методі навчання з підкріпленням. Запропонований метод не має інформації про ваги моделі-класифікатора та про дані, на яких модель-класифікатор була натренована. Ідея методу полягає в тому, що процес додавання матричного збурення можна описати як Марківський процес прийняття рішень. Тому в кожний момент часу, модель визначає найкраще рішення, і додає матричне збурення до зображення, базуючись на прийнятому рішенні.

Враховуючи обмежені обчислювальні ресурси, метод ще не досліджений повність, але показав хороші результати для датасету Cifar-10 по кількості необхідних запитів до мережі-класифікатора для побудови матричного збурення, яке призводить до передбачення неправильного класу.

1. Загальна інформація

1.1.Нейронна мережа як універсальний класифікатор

Нейронна мережа [1] може бути представлена як відображення з вхідного простору, позначеним як R^D, у простір можливих класів, позначених як R^C, $f(X,\theta_c): R^D \to R^C$ з тренованими параметрами θ_c. Якщо кількість можливих класів у просторі R^C дорівнює двом (бінарна класифікація), то використовують функцію сигмоїд як функцію активації:

$$sigmoid(x) = \frac{1}{1+e^{-x}}$$

Якщо кількість классів більше двох (мультикласова класифікація), то використовують функцію софтмакс:

$$softmax(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}}$$

В цій роботі будуть використовуватися нейронні мережі, які мають софтмакс як активаційну функцію.

1.2. Лінійна нейронна мережа

Формально лінійна нейронна мережа [5] – це напрямлений ациклічний зважений граф $G = (V, E, W, B, T)$, де V – набір вершин, $E \subset V \times V$ – набір ребер, $W: E \to R$ задає вагу кожному ребру нейронної мережі, $B: V \to R$ додає зміщення до кожної вершини, T задає тип кожної вершини з набору можливих типів $\mathcal{T} \in \{input, linear, relu, \dots\}$.

Кожна вершина зважує вхідний вектор та додає зміщення як наведено у формулі нижче .

$$z = b + \sum_i w_i x_i \text{ або } z = w \cdot x + b$$

1.3. Конволюційна нейронна мережа

Параметри конволюційної нейронної мережі складаються з набору фільтрів, які можна натренувати. Кожен фільтр малий просторово (по ширині та висоті), але простягається на всю глибину вхідного об'єму. Наприклад, типовий фільтр на першому шарі конволюційної нейронної мережі може мати розмір 5x5x3 (тобто 5 пікселів у ширину та висоту та 3, оскільки зображення мають глибину 3, кольорові канали). Під час прямого проходу ми ковзаємо (точніше, згортаємо) кожен фільтр по ширині та висоті вхідного об'єму та обчислюємо крапкові добутки між записами фільтра та входом у будь-якій позиції. Коли ми просуваємо фільтр по ширині та висоті вхідного об'єму, ми створимо двовимірну карту активації, яка дає відповіді цього фільтра в кожному просторовому положенні. Інтуїтивно, мережа вивчить фільтри, які активуються, коли вони бачать певний тип візуальних елементів, таких як край певної орієнтації або пляма якогось кольору на першому шарі, або врешті-решт цілі соти або подібні до колеса візерунки на вищих шарах мережі. . Тепер у нас буде цілий набір фільтрів у кожному рівні конволюції (наприклад, 12 фільтрів), і кожен з них створить окрему двовимірну карту активації. Ми складемо ці карти активації вздовж виміру глибини та отримаємо вихідний обсяг. Приклад роботи конволюційного шару можна побачити на Рис.1.

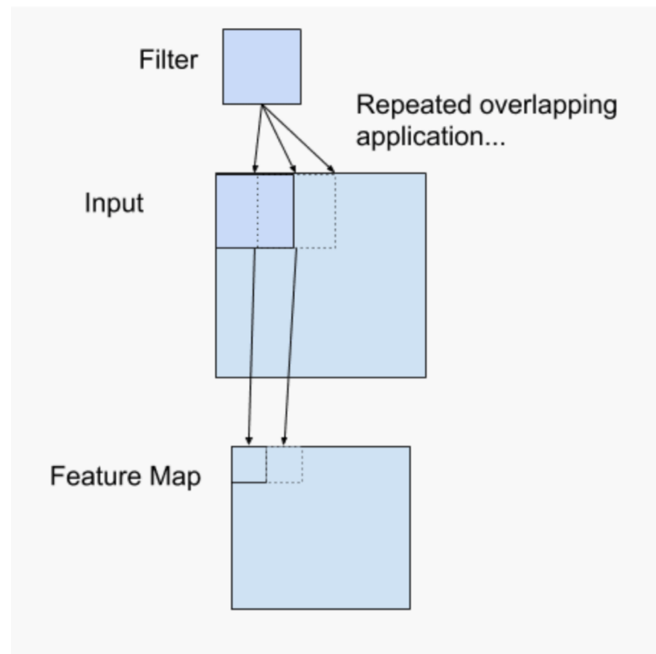

Рис.1 Схема роботи конволюційного шару

1.4. ResNet

Основною ідеєю ResNet [8] є введення так званого "стрибкового з'єднання", яке пропускає один або кілька шарів, як показано на наступному малюнку (Рис.2).

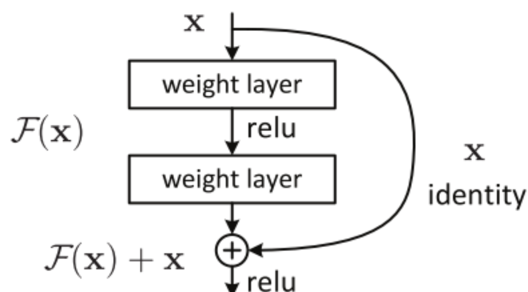

Рис.2 Зображення роботи "стрибкового з'єднання"

Цей підхід дозволяє даним легко протікати між рівнями, не перешкоджаючи навчальним можливостям моделі глибокого навчання. Перевага додавання цього типу підключення пропуску полягає в тому, що якщо будь-який шар погіршить продуктивність моделі, він буде пропущений.

Інтуїція підключення пропуску полягає в тому, що мережі простіше навчитися перетворювати значення $f(x)$ на нуль, щоб вона поводилась як функція ідентичності, а не навчилася поводитися як функція ідентичності самостійно, намагаючись знайти правильний набір значень, які давали б бажаний результат.

1.5. Функції втрат

Функція втрат визначає на скільки добре модель робить правильні передбачення. Чим ближче передбачення моделі, тим ближче значення функції втрат до мінімуму, чим далі – тим ближче до максимуму. Для задачі класифікації найчастіше використовують ентропійні функції втрат: бінарна крос-ентропія та крос-ентропія.

Бінарна крос-ентропія використовується для задачі бінарної класифікації. $\boldsymbol{y}$ – відповідний клас, а $\boldsymbol{P}(\boldsymbol{y})$ – передбачена ймовірність моделлю, що відповідний об'єкт належить до класу $\boldsymbol{y}$.

$$\boldsymbol{Binary\ cross\ entropy}:$$

$$J = -\frac{1}{N}\sum_{i=1}^{N} y_i log\left(P(y_i)\right) + (1 - y_i) log(1 - P(y_i))$$

Для задачі мултикласової класифікації використовують функцію втрат крос-ентропію. Передбачені моделлю значення порівнюються зі справжніми та пеналізуються за відстанню від справжніх классів.

$$\boldsymbol{Cross\ entropy}:$$

$$J = -\frac{1}{N}\sum_{i=1}^{N} y_i\, log(P(y_i))$$

Для задачі регресії найпопулярнішою функцією втрат є середнє відхилення помилки (MSE)

$$\mathbf{Mean\ squared\ error}:$$

$$\mathrm{J} = \frac{1}{N}\sum_{i=1}^{N}\left(y_i - \widehat{y_i}\right)^2$$

Оскільки, функції втрат використовуються для оптимізації ваг моделі, ці функції мають бути неперервними.

1.6. Оптимізатори нейронних мереж

Оптимізатори – алгоритми машинного навчання, які використовуються для зміни параметрів нейронних мереж та методів машинного навчання у цілому, такі як ваги з метою зменшення помилки передбачень.

1.6.1 Градієнтний спуск

Метод градієнтного спуску один з найпоширеніших методів оптимізації ваг нейронної мережі. Він широко використовується в алгоритмах лінійної регресії та класифікації. Зворотне поширення в нейронних мережах також використовує алгоритм градієнтного спуску.

Градієнтний спуск - це алгоритм оптимізації першого порядку, який залежить від похідної першого порядку функції втрат. Він обчислює, яким чином слід змінити ваги, щоб функція могла досягти мінімумів. Завдяки зворотному розповсюдженню втрати передаються з одного шару на інший, а параметри моделі, також відомі як ваги, змінюються залежно від втрат, щоб втрати можна було мінімізувати.

$$\boldsymbol{\theta = \theta - \alpha * \nabla J(\theta)}$$

Недоліками цього алгоритму є великі обчислювальні витрати, та існує ймовірність зійтись у локальному мінімумі.

1.6.2 Стохастичний градієнтний спуск (SGD)

Це варіант градієнтного спуску. Він намагається частіше оновлювати параметри моделі. У цьому параметри моделі змінюються після обчислення втрат на кожному навчальному прикладі. Отже, якщо набір даних містить 1000 рядків, SGD оновить параметри моделі 1000 разів за один цикл набору даних, а не за один раз, як у градієнтному спуску.

$$\boldsymbol{\theta = \theta - \alpha * \nabla J(\theta; x_i, y_i)}$$

де $\mathbf{x_i}, \boldsymbol{y}_\mathbf{i}$ – тренувальні данні.

Оскільки параметри моделі часто оновлюються, параметри мають велику дисперсію та коливання функції втрат при різній інтенсивності.

1.6.3 Adam

Adam – алгоритм який використовує імпульси першого та другого порядку.

Інтуїція методу полягає в тому, що ми не хочемо змінювати ваги так швидко лише тому, що ми можемо перестрибнути мінімум, ми хочемо трохи зменшити швидкість для ретельного пошуку.

На додаток до того, що Адам зберігає експоненціально занепадаючий середній показник минулих квадратних градієнтів, Адам також зберігає експоненціально спадаючий середній показник минулих градієнтів $\boldsymbol{M(t)}$.

$\boldsymbol{M(t)}$ та $\boldsymbol{V(t)}$ значення першого моменту, яке є середнім значенням, і другого моменту, яке є нецентрованою дисперсією градієнтів відповідно.

$$\widehat{m_t} = \frac{m_t}{1 - \beta_1^t}$$

$$\widehat{v_t} = \frac{v_t}{1 - \beta_2^t}$$

$$g_t = \nabla J(\theta_t)$$

Тут ми беремо середнє значення $\boldsymbol{M(t)}$ та $\boldsymbol{V(t)}$, так що $\mathbb{E}[m(t)]$ може бути рівним $\mathbb{E}[g(t)]$, де $\mathbb{E}[f(t)]$ - очікуване значення $f(x)$.

Оновлення параметрів відбувається за наступною формулою, де $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10e - 8$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\widehat{v_t}} + \epsilon} \widehat{m}_t$$

1.7. Типи атак на конволюційні нейронні мережі для задачі класифікації

Деякі задачі машинного навчання є складними по своїй суті. Прикладом такої задачі є Рис.3. Навіть люди в деяких випадках можуть неправильно класифікувати усі приклади. Тому не дивно, що нейронні мережі не можуть забезпечити максимальну точність в таких задачах. Але з розвитком обчислювальних потужностей все більше задач можна вирішувати на рівні людини або навіть, перевершуючи його, за допомогою нейронних мереж. Нейронні мережі не є на стільки ж надійними у їх рішеннях ніж людина і можуть бути обдурені за допомогою підібраного збурення входу. Існують два типи атак на нейронні мережі: атаки білої скриньки та атаки чорної скриньки.

Рис. 3 Приклад задачі класифікації: собака чи швабра.

1.7.1. Атака білої скриньки

Нам дано классифікатор $\boldsymbol{f}$. Більш конкретно, припустимо що наші дані роздільні, тобто існує така “правдива” функція $\boldsymbol{h}: \boldsymbol{X} \to \{\pm\mathbf{1}\}$. Щоб знайти згенероване збурення $\widetilde{\boldsymbol{x}}$ таке що $\|\boldsymbol{x} - \widetilde{\boldsymbol{x}}\| \leq \boldsymbol{\varepsilon}$, ми використовуємо алгоритм зворотнього поширення помилки. З допомогою цього алгоритму можна порахувати $\boldsymbol{\nabla}_{\boldsymbol{x}} \boldsymbol{g}(\boldsymbol{x})$ щодо зміни вхідного зображення, а не ваг моделі. Отже, $\boldsymbol{h}(\boldsymbol{x}) \boldsymbol{\nabla}_{\boldsymbol{x}} \boldsymbol{g}(\boldsymbol{x})$ – вектор з $\boldsymbol{D}$ елементів, де позитивні елементи розташовані на позиціях де збільшення значення призводить до більш точних передбачень та відємних де призводить до менш точних.

Припустимо, що дозволено змінювати елемент $\boldsymbol{x}$ в сторону нового елементу $\widetilde{\boldsymbol{x}}$, з обмеженнями $\|\widetilde{\boldsymbol{x}} - \boldsymbol{x}\|_{\mathbf{2}} \leq \boldsymbol{\varepsilon}$. Наша ціль погіршити передбачення якомога більше. Що означає, максимально зменшити ймовірність правильного класу.

Якщо $\boldsymbol{\varepsilon}$ мале, тоді можемо припустити що зміну ймовірності задається зміною першого порядку, отже добре передбачається градієнтом. Враховуючи наш “бюджет” у l2 нормі, оптимальний крок – у зворотньому напрямі градієнту

$$\widetilde{\boldsymbol{x}} = \boldsymbol{x} - \boldsymbol{\varepsilon} \boldsymbol{h}(\boldsymbol{x}) \frac{\boldsymbol{\nabla}_{\boldsymbol{x}} \boldsymbol{g}(\boldsymbol{x})}{\|\boldsymbol{\nabla}_{\boldsymbol{x}} \boldsymbol{g}(\boldsymbol{x})\|}$$

Якщо для **x** ми змогли змінити розподіл ймовірностей класів, з урахування обмеження **ε**, ми знайшли збурення.

1.7.2. Атака чорної скриньки

У прикладі вище нам необхідний доступ до нейронної мережі щоб порахувати градієнти. Але в реальному світі дуже мало ситуацій коли дійсно є доступ до ваг моделі. Тому ми не маємо змоги рахувати градієнти. У цьому випадку, ми отримаємо інформацію лише про передбачення моделі та ймовірності класів

1.8. Постановка задачі навчання з підкріпленням

Ми розглядаємо завдання, в яких агент взаємодіє із середовищем E в послідовності дій, спостережень та винагород. На кожному кроці часу агент вибирає дію з набору законних ігрових дій, A = {1,. . . , K}. Дія передається емулятору і змінює його внутрішній стан та винагороду. Загалом E може бути стохастичним. Внутрішній стан емулятора агент не спостерігає; натомість він спостерігає зображення $\boldsymbol{x_t} \in \boldsymbol{R^d}$ з емулятора, який є вектором вихідних значень пікселів, що представляють поточний екран. Крім того, він отримує винагороду $\boldsymbol{r_t}$, що відображає зміну результатів гри. Зверніть увагу, що загалом ігровий рахунок може залежати від усієї попередньої послідовності дій та спостережень; відгук про дію може бути отриманий лише після того, як пройде багато тисяч кроків часу.

Оскільки агент спостерігає лише зображення поточного екрану, завдання частково спостерігається, тобто неможливо повністю зрозуміти поточну ситуацію лише з поточного екрана $\boldsymbol{x_t}$. Тому ми розглядаємо послідовності дій та спостережень, $\boldsymbol{s_t} = \boldsymbol{x_1}, \boldsymbol{a_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{a_{t-1}}, \boldsymbol{x_t}$, і вивчаємо ігрові стратегії, які залежать від цих послідовностей. Припускається, що всі послідовності в емуляторі закінчуються за кінцеву кількість кроків у часі. Цей формалізм породжує великий, але скінченний процес прийняття рішень Маркова, в якому кожна послідовність є окремим станом. Як результат, ми можемо застосувати стандартні методи навчання для Марківського процесу, просто використовуючи повну послідовність $\boldsymbol{s_t}$ як подання стану в момент часу $\boldsymbol{t}$.

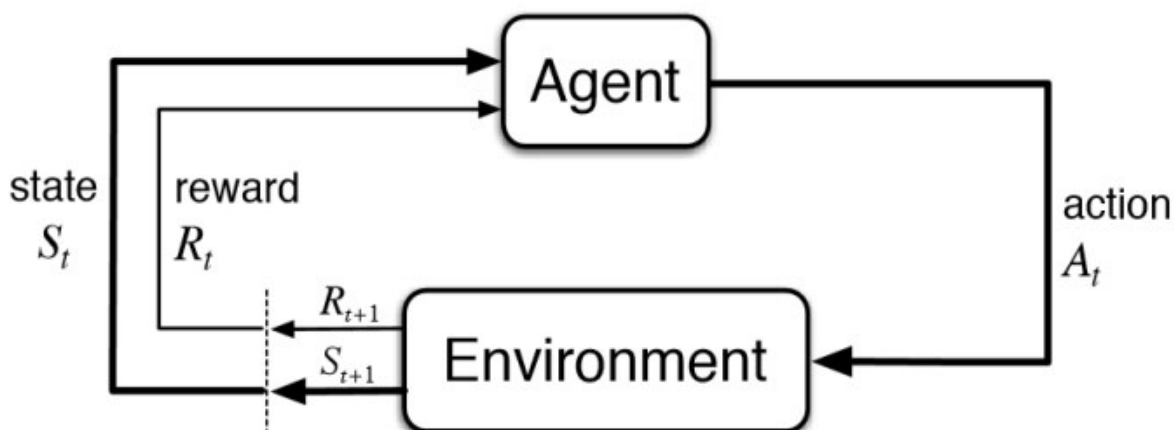

Рис. 4. Зображення роботи навчання з підкріпленням: агент здійснює дії в певному середовищі, середовище реагує на дію агента зміною середовища та певною винагородою.

Мета агента - взаємодіяти з емулятором, вибираючи дії таким чином, щоб максимізувати майбутні винагороди. Ми робимо стандартне припущення, що майбутні винагороди знижуються з коефіцієнтом γ за крок часу та визначте майбутню дисконтовану винагороду в момент $\boldsymbol{t}$ як $\boldsymbol{R_t} = \sum_{t'=t}^{T} \boldsymbol{\gamma}^{t'-t} \boldsymbol{r}_{t'}$, де T - крок часу, на якому закінчується гра. Ми визначаємо функцію оптимальних значень дії $\boldsymbol{Q}^*(\boldsymbol{s}, \boldsymbol{a})$ як максимальну очікувану віддачу, досяжну за будь-якою стратегією, побачивши деяку послідовність $\boldsymbol{s}$ потім виконавши певну дію a, $\boldsymbol{Q}^*(\boldsymbol{s}, \boldsymbol{a}) = \max_{\pi} \mathbb{E}\left[\boldsymbol{R_t} \middle| \boldsymbol{s_t} = \boldsymbol{s}, \boldsymbol{a_t} = \boldsymbol{a}, \boldsymbol{\pi}\right]$, де $\boldsymbol{\pi}$ - відображення послідовностей до дій (або розподілу за діями).

Оптимальна функція значення-дія виконує важливу тотожність, відому як рівняння Беллмана. Це базується на наступній інтуїції: якщо оптимальне значення $\boldsymbol{Q}^*(\boldsymbol{s}', \boldsymbol{a}')$ послідовності $\boldsymbol{s}'$на наступному кроці часу було відоме для всіх можливих дій $\boldsymbol{a}'$, то оптимальною стратегією є вибір дії $\boldsymbol{a}'$ для максимізації очікуваного значення $\boldsymbol{r} + \boldsymbol{\gamma} \boldsymbol{Q}^*(\boldsymbol{s}', \boldsymbol{a}')$,

$$\boldsymbol{Q}^*(\boldsymbol{s}, \boldsymbol{a}) = \mathbb{E}[\boldsymbol{r} + \boldsymbol{\gamma} \max_{\boldsymbol{a}'} \boldsymbol{Q}^*(\boldsymbol{s}', \boldsymbol{a}') | \boldsymbol{s}, \boldsymbol{a}]$$

Основною ідеєю багатьох алгоритмів навчання з підкріпленням є оцінка функції значення дії за допомогою рівняння Беллмана в якості ітеративного оновлення $\mathbf{Q_{i+1}(s,a) = \mathbb{E}[r + \gamma \max_{a'} Q_i(s', a')|s, a]}$ Такі алгоритми ітерації значень сходяться до оптимальної функції значення дії, $\mathbf{Q_i = Q^*}$ з $\mathbf{i \rightarrow \infty}$. На практиці цей базовий підхід є абсолютно непрактичним, оскільки функція значення дії оцінюється окремо для кожної послідовності без будь-якого узагальнення. Натомість зазвичай використовується апроксиматор функції для оцінки функції значення дії, $\mathbf{Q(s,a;\theta) \approx Q^*(s,a)}$. У навчанні з підкріпленням це, як правило, лінійний апроксиматор, але іноді замість нього використовується нелінійний апроксиматор функцій, наприклад нейронна мережа. Ми позначаємо апроксиматор функцій нейронної мережі з вагами θ як Q-мережу. Q-мережу можна навчити, мінімізуючи послідовність функцій втрат $\mathbf{L_i(\theta_i)}$, яка змінюється на кожній ітерації і,

$$\mathbf{L_i(\theta_i) = \mathbb{E}[\left(y_i - Q(s,a;\ \theta_i)\right)^2]}$$

де $\mathbf{y_i = \mathbb{E}[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1})|s, a]}$ - ціль для ітерації $\mathbf{i}$. Параметри попередньої ітерації $\mathbf{\theta_{i-1}}$ залишаються фіксованими при оптимізації функції втрат $\mathbf{L_i(\theta_i)}$.

2. Практичне дослідження моделі

2.1. Опис середовища та постановка задачі

Оскільки задача полягає у побудові моделі-нападника у варіанті чорної коробки, середовище для експерименту складається з тестових зображень, моделі класифікатора, моделі нападника, та відповідної нагороди за результат матричного збурення.

2.1.1 Опис даних

В ході курсової роботи було розглянуто алгоритм на двох наборах даних MNIST та Cifar-10. Розміри зображень у цих датасетах дорівнюють 28 на 28 та 32 на 32 на 3 для MNIST та Cifar-10, відповідно. Кожен з датасетів має 10 класів: “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9” та “airplane”, “automobile”, “bird”, “cat”, “deer”, “dog”, “frog”, “horse”, “ship”, “truck”, відповідно. Тренувальні дані обох датасетів містять 50000 зображень по 5000 зображень на кожний клас. Тестувальні дані також однакові за розміром, 10000 зображень зі зваженою кількістю зображень для кожного класу. З метою зменшення обчислювального навантаження, зображення з датасету Cifar-10 були переведені у одноканальний формат, отримавши нову розмірність – 32 на 32 пікселі.

З метою порівняння запропонованого алгоритму були використані лише тестувальні зображення з датасетів, по 10000 зображень для кожного з них. Приклади зображень наведені у Рис.5 та Рис.6.

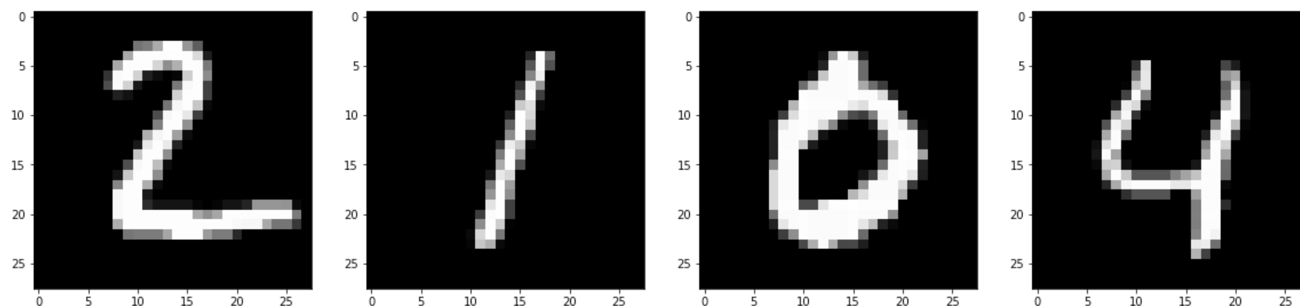

Рис.5 Приклад зображень з тестового датасету MNIST, які належать другому, першому, нульовому та четвертому класам, відповідно. Значення пікселів кожного зображення в межах від 0 до 1.

Рис.6 Приклад зображень з тестового датасету Cifar-10 з відповідними класами. Значення пікселів в межах від 0 до 1.

2.1.2 Опис можливих дій та винагород

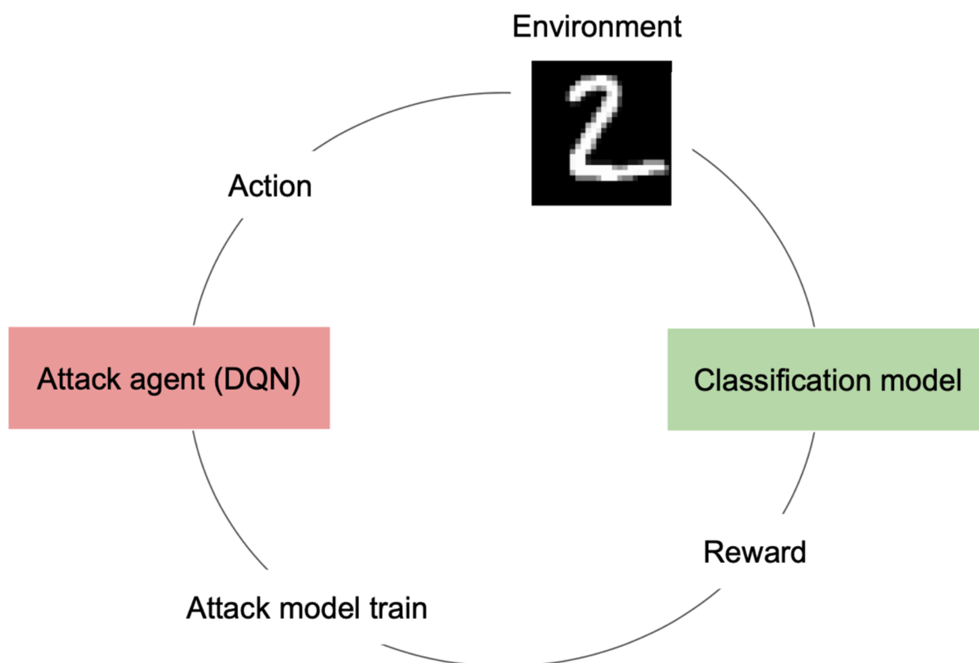

Рис.7 Схема роботи алгоритма. Визначаємо початкове передбачення моделі-класифікатора. Отримуємо відповідну винагороду, тренуємо модель-нападник з урахуванням нової винагороди. Знову обираємо дію та змінюємо середу зображення

Постановка задачі у формі навчання з підкріпленням вимагає конкретно задану кількість можливих дій та винагород. В цих експериментах простір дій був заданий як номер пікселя до якого додавати збурення. Отже, для датасету MNIST простір мав 784 (28 * 28) можливих дій, а для Cifar-10 – 1024 (32 * 32). Як частина подальшої роботи, простір дій буде збільшено до вибору пікселя, каналу зображення, числа збурення, та ймовірності додавати збурення чи ні.

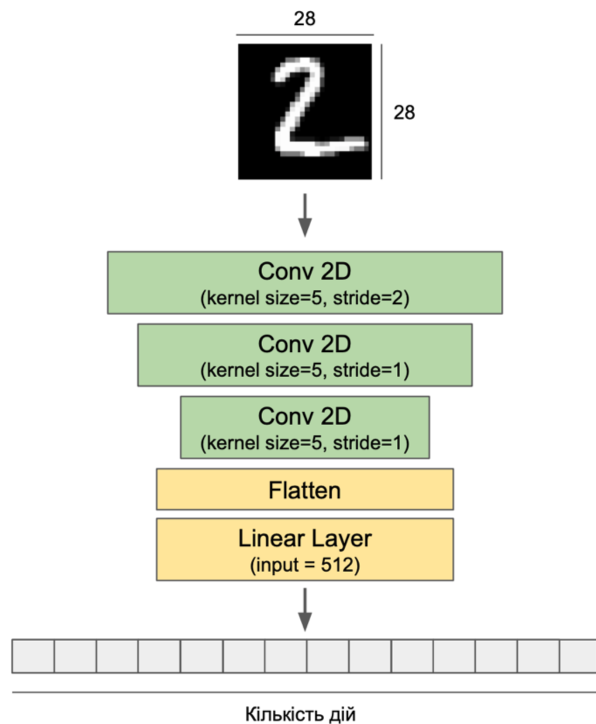

Рис. 8 Візуалізація моделі DQN для датасету MNIST

Винагорода залежить від трьох можливих наслідків від прийнятого рішення. У разі, якщо змінене зображення відрізняється від початкового більше ніж на $\boldsymbol{\epsilon}$ для заданої норми $\boldsymbol{l_2}$ або $\boldsymbol{l_\infty}$, то присуджуємо нагороду -1 та починаємо експеримент знову. Якщо змінене зображення в межах $\boldsymbol{\epsilon}$ для заданої норми та результат передбачення моделі класифікатора не змінилося, то присуджуємо винагороду -0.1. Якщо ж змінене зображення в межах $\boldsymbol{\epsilon}$ для заданої норми та результат передбачення моделі класифікатора змінилося, то присуджуємо нагороду 1 та переходимо до наступного зображення. Ці константи на практиці показали найкращі результати навчання, хоча можливі й інші варіанти опису винагород.

2.2. Тренування агента

Модель-нападник побудована на основі конволюційної DQN [7] (Deep Q Network) з трьома конволюційними та лінійним шарами (Рис.8). Завдання цієї моделі вивчити послідовність дій, які потрібно обрати, щоб змінити передбачення класифікаційної мережі. З метою генералізації обраних дій, використовується епсилон жадібне прийняття рішень з поступовим зниження випадковості прийняття дій.

Випадковість прийняття рішень визначається за експоненційним ковзаючим середнім та має початкову ймовірність 0.9, для випадкового рішення, яка знижується до 0.05. Кожні 30 запитів, до моделі-класифікатора, модель-нападник оновлює свої ваги.

Експеримент складався зі 100 зображень та бюджетом у 10000 запитів до моделі класифікатора для кожного зображення. У кожний момент часу результати прийнятого рішення записувалися у масив “пам’яті”, значення якого потім будуть використовуватися для навчання. В кожний момент тренування моделі використовувався набір з 64 наборів станів, дій, винагород та наступних станів. Модель тренувалася мінімізуючи різницю у винагороді на момент часу t та очікуваним значенням винагороди за всі наступні часові кроки за допомогою середнім відхиленням помилки.

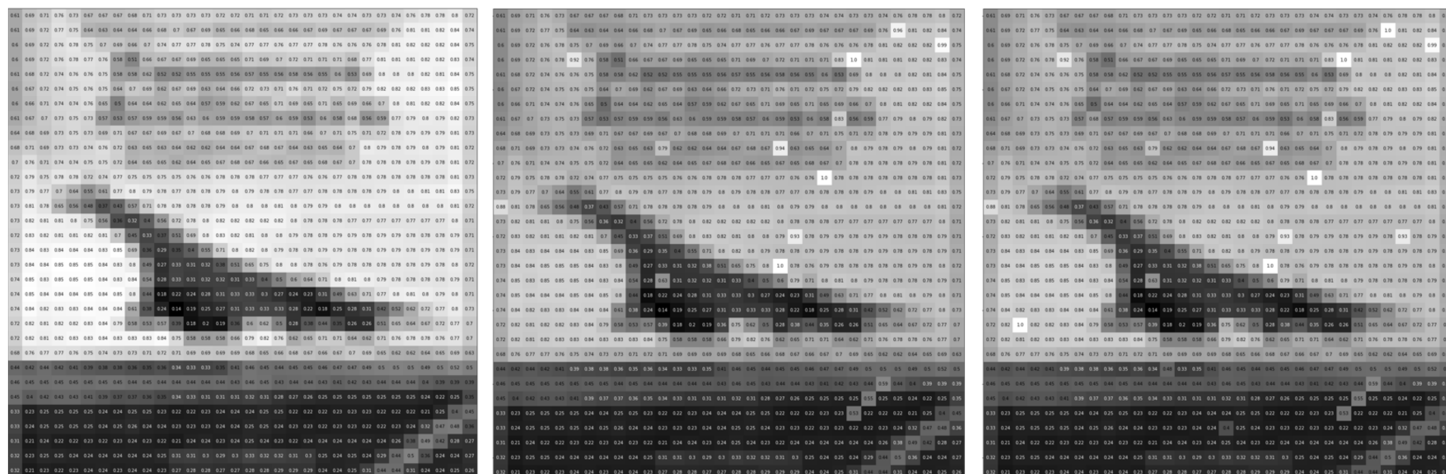

Рис.9 Процес навчання моделі-нападника на зображенні класу “літак”. Перше зображення – початкове, друге – після 10 запитів до класифікаційної моделі, третьє – після 30 запитів до класифікаційної моделі.

Псевдокод алгоритму		
1	**Вхід:** набір зображень (dataset) , модель-нападник (Attacker), модель-класифікатор (Classifier), максимальна кількість запитів (max_queries)	
2	**Вихід:** точність класифікаційної моделі - model_accuracy	
3	Для кожного зображення sample з набору зображень dataset:	
4	True_pred = передбачення моделі Classifier для sample	
5	Sample_env = Створюємо середовище з зображенням sample	
6	Для кожного запиту querie з max_queries:	
7		Action = передбачення моделлю Attacker дію в момент часу t, який віповідає querie.
8		Perburbed_sample = змінюємо початкове зображення sample за допомогою обраного action
8		Pred_class = передбачення моделі Classifier для Perburbed_sample
9		Norm = Перевіряємо чи зображення в межах ϵ у заданій нормі
10		Matched_predictions = Порівнюємо Pred_class з True_pred
11		Reward = Визначаємо винагороду, враховуючи значення Norm, Matched_predictions та querie

12	Sample_env.update() - Оновлюємо стан середовища, враховуючи обраний action
13	Тренуємо Attacker
14	Якщо Matched_predictions == False:
15	Переходимо до наступного зображення sample_next
16	Якщо Matched_predictions == True:
17	Продовжуємо зміну зображення sample
18	Підраховуємо нову точність моделі - model_accuracy
Кінець алгоритму	

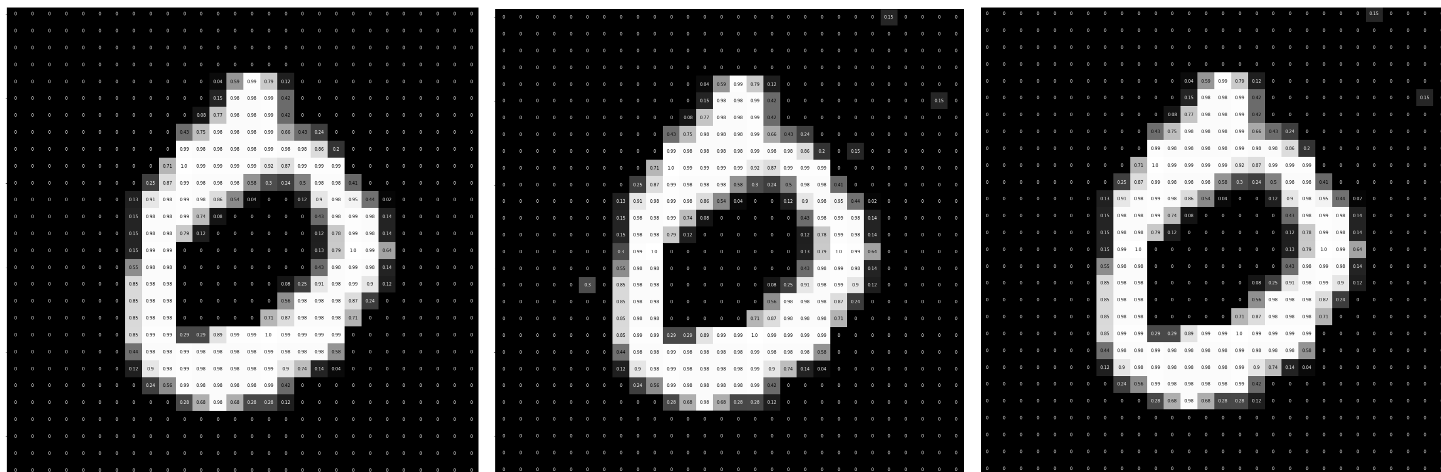

Рис.10 Процес навчання моделі-нападника на зображенні класу “0”. Перше зображення – початкове, друге – після 5 запитів до класифікаційної моделі, третє – після 10 запитів до класифікаційної моделі.

2.3. Результати та порівняння з іншими методами

Для порівняння запропонованого методу, використовував публічні змагання з генерації матричних збурень для датасетів MNIST та Cifar-10. Враховуючи свої обмежені обчислювальні ресурси, мої експерименти були засновані на 100 зображеннях, у порівнянні з 10000 в справжньому експерименті.

Для датасету MNIST класифікаційна модель [9] складається з двох конволюційних шарів з софтмакс функцією активації. Модель натренована з урахуванням максимально можливого збурення у 0.3 для $\boldsymbol{l}_\infty$ норми має точність у 98% для тестових даних.

Для атак чорної коробки, на момент написання роботи, найкраще результат забезпечує атака, яка використовує генеративну модель AdvGAN [10]. AdvGAN складається з трьох частин: модель-генератор, модель-дискримінатор та модель-класифікатор. Модель-генератор отримує на вхід зображення $\boldsymbol{x}$ та генерує збурення $\boldsymbol{G}(\boldsymbol{x})$. Тоді $\boldsymbol{x} + \boldsymbol{G}(\boldsymbol{x})$ передається у модель-дискримінатор, щоб розділяти згенеровані та справжні зображення $\boldsymbol{x}$. Ціль моделі-дискримінтора заохочувати модель-генератор генерувати зображення, які не відрізняються від справжніх зображень. Модель-класифікатор – дистильована версія справжньої мережі, яка динамічно оновлює свої ваги для кожного запиту, одночасно з моделлю-генератором. Цей підхід знижує точність моделі на тестових даних на 5.34% до 92.76%.

Модель атаки, запропонованої у цій роботі, на 100 зображеннях обраних випадковим чином, знизила точність моделі з 99% до 97%. Поганий результат атаки може бути пояснений тим, що значення пікселів цього датасету мало відрізняються у зображеннях, що належать одному класу, тому модель-класифікатор добре вивчила закономірності кожного класу. При тренуванні моделі на графічному процесорі, модель-нападки буде мати більше тренувальних даних, більший вибір можливих дій, для більш точного вибору вразливих пікселі, і тому може покращити результат атаки.

Для датасету Cifar-10 класифікаційна методель побудована на основі моделі ResNet-18 та має точність у 81.29% для тестових даних. В початковому варіанті змагання, модель натренована з урахуванням можливого збурення у 8.0 для зображень у діапазоні 0 – 255 у l_∞ нормі. Враховуючи свої обмежені ресурси, я використовував чорно-білі зображення зі значеннями пікселів у межах від 0 до 1.

Для атак чорної коробки, на момент написання роботи, найкращий результат забезпечує атака Square Attack [11]. Ця атака заснована на алгоритмі рандомізованого пошуку, який обирає локалізовані оновлення у форми квадратів у випадкових позиціях на зображенні таким чином, щоб збурені зображення лежали близько межі многовиду тренувальних зображень. Цей алгоритм знижує точність моделі-класифікатора до 7.2%.

Модель атаки запропонованої у цій роботі, на 100 зображеннях обраних випадковим чином, знизила точність моделі з 84.5% до 37% з середньою кількістю запитів у 853 для успішних атак та медіаною у 53 запити. Модель-нападник показала кращі результати ніж на даних з датасету MNIST, оскільки датасету Cifar-10 притаманна мінливість значень пікселів, а тому і модель-класифікатор є більш вразливою для незначних збурень.

Модель	**Точність на тестовому датасеті MNIST**
Початкова точність моделі-класифікатора	98%
AdvGan атака [10]	92.76%
Атака з підкріпленням (запропонований)	97%

Модель	**Точність на тестовому датасеті Cifar-10**
Початкова точність моделі-класифікатора	81.29%
Square-attack [11]	7.2%
Атака з підкріпленням (запропонований)	37%

Висновки

В ході роботи було запропоновано новий алгоритм атаки на класифікаційну нейронну мережу за допомогою методу навчання з підкріпленням. Ідея методу полягає у тому, що процес конструювання матричного збурення можна представити як Марківський процес прийняття рішень. Переваги даного методу в тому, що він не потребує попередньої інформації про зображення та про модель-класифікатор. На перших ітераціях модель-нападник досліджує простір зображень, та визначає які з регіонів зображень призводять неправильного передбачення. Потім, на визначених регіонах, модель шукає вже конкретні пікселі для кожного зображення. Враховуючи експеримент зі 100 зображеннями, модель не встигає вивчити загальні регіони з потенційними вразливостями, а тому дії агента є частково випадковими. Тим не менш, отримані результати вказують на перспективність запропонованого методу. При достатньому забезпеченні обчислювальними ресурсами, модель матиме в 100 разів більше тренувальних даних, а отже і дослідить більше регіонів зображень.

Список використаної літератури

1. Chollet F. Deep Learning with Python; – Manning Publications [2017]

2. Géron A. Hands-On Machine Learning with Scikit-Learn and TensorFlow; – O'Reilly [2017]

3. Hastie T. The Elements of Statistical Learning. Data Mining, Inference, and Prediction / Hastie T., Tibshirani R., Friedman J. – Springer [2017]

4. Goodfellow I. Deep Learning / Goodfellow I., Bengio Y., Courville A. – Cambridge MA : MIT Press [2017]

5. Formal Verification of Piece-Wise Linear Feed-Forward Neural Networks [2017] – Rudiger Ehlers

6. The MNIST database of handwritten digits – Yann LeCunn

7. Playing Atari with Deep Reinforcement Leaning [2013] – Volodymyr Mnih et al.

8. Deep Residual Learning for Image Recognition [2015]– Kaiming et al.

9. Towards Deep Learning Models resistant to Adversarial Attacks [2017] – Madry et al.

10. Generating Adversarial Examples with Adversarial Networks [2019] – Xiao et al.

11. Square Attack: a query-efficient black-box adversarial attack via random search [2020] – Andriushchenko et al.

12. Adversarial Attacks and Defences: A Survey [2018] – Chakraborty et al.

Додаток А.

Текст програми

```
from utils.plot_utils import *
from environments import AttackEnv
from models.classifier_models import CIFAR10_CL_Model, ResNet18
from dataset import CIFAR10Dataset

import torch
import torch.nn as nn
import torch.nn.functional as F
from torchvision import transforms
from collections import namedtuple, deque

import random
import math
import numpy as np
from tqdm import tqdm

def dict2obj(d):
    if isinstance(d, list):
        d = [dict2obj(x) for x in d]
    if not isinstance(d, dict):
        return d
    class C(object):
        pass
    o = C()
    for k in d:
        o.__dict__[k] = dict2obj(d[k])
```

```
    return o

params = {}
params['dataset'] = 'mnist'
params['norm'] = 'l_inf'
params['eps'] = 0.3
params['device'] = torch.device("cuda" if torch.cuda.is_available() else "cpu")
args = dict2obj(params)

class ClipValues:
    def __init__(self):
        pass

    def __call__(self, image):
        return torch.clip(image, 0, 1)

transform = transforms.Compose([
    transforms.Grayscale(1),
    transforms.ToTensor(),
    ClipValues()
])
data = CIFAR10Dataset(
    './data_cifar',
    download=False,
    train=False,
    transform=transform
)
classes = ('plane', 'car', 'bird', 'cat',
           'deer', 'dog', 'frog', 'horse', 'ship', 'truck')
```

```
Transition = namedtuple(
    'Transition', ('state', 'action', 'next_state', 'reward')
)

class ReplayMemory(object):
    def __init__(self, capacity):
        self.memory = deque([],maxlen=capacity)

    def push(self, *args):
        """Save a transition"""
        self.memory.append(Transition(*args))

    def sample(self, batch_size):
        return random.sample(self.memory, batch_size)

    def __len__(self):
        return len(self.memory)

class DQN(nn.Module):
    def __init__(self, outputs):
        super(DQN, self).__init__()
        self.conv1 = nn.Conv2d(1, 16, kernel_size=5, stride=2)
        self.conv2 = nn.Conv2d(16, 32, kernel_size=5, stride=1)
        self.conv3 = nn.Conv2d(32, 32, kernel_size=5, stride=1)

        self.head = nn.Linear(32 * 6 * 6, outputs)
        self.flatten = nn.Flatten()
```

```
    def forward(self, x):
        x = x.to(args.device)
        x = F.relu(self.conv1(x))
        x = F.relu(self.conv2(x))
        x = F.relu(self.conv3(x))
        x = self.flatten(x)
        return self.head(x)

BATCH_SIZE = 128
GAMMA = 0.99
EPS_START = 0.9
EPS_END = 0.1
EPS_DECAY = 20
TARGET_UPDATE = 30

n_actions = 32 * 32
EPSILON = 0.3
steps_done = 0

policy_net = DQN(n_actions).to(args.device)
target_net = DQN(n_actions).to(args.device)
target_net.load_state_dict(policy_net.state_dict())
target_net.eval()

optimizer = torch.optim.Adam(policy_net.parameters(), lr=0.001)
memory = ReplayMemory(1000)

#pretrained_model = 'model_weights/cifar_net.pth'
#model = CIFAR10_CL_Model().to(args.device)
```

```
pretrained_model = 'model_weights/cifar_resnet-2.pth'
model = ResNet18().to(args.device)
model.load_state_dict(torch.load(pretrained_model, map_location='cpu'))
model.eval();
print(model(sample_image.unsqueeze(1)).max(1)[1].item())
assert sample_image_class == model(sample_image.unsqueeze(1)).max(1)[1].item()

def select_action(state):
    global steps_done
    sample = random.random()
    eps_threshold = (
        EPS_END + (
            (EPS_START - EPS_END) * math.exp(-1 * steps_done / EPS_DECAY)
        )
    )
    steps_done +=1
    if sample > eps_threshold:
        with torch.no_grad():
            return policy_net(state.unsqueeze(1)).max(1)[1].view(1, 1)
    else:
        return torch.tensor(
            [[random.randrange(n_actions)]],
            device=args.device,
            dtype=torch.long
        )

def optimize_model():
    if len(memory) < BATCH_SIZE:
        return
```

```
transitions = memory.sample(BATCH_SIZE)
batch = Transition(*zip(*transitions))

non_final_mask = torch.tensor(
    tuple(map(lambda s: s is not None, batch.next_state)),
    device=args.device,
    dtype=torch.bool)
non_final_next_states = torch.cat(
    [s for s in batch.next_state if s is not None]
)

state_batch = torch.cat(batch.state)
action_batch = torch.cat(batch.action)
reward_batch = torch.cat(
    [torch.tensor(el).unsqueeze(0) for el in batch.reward]
)

state_action_values = policy_net(
    state_batch.unsqueeze(1)
).gather(1, action_batch)

next_state_values = torch.zeros(BATCH_SIZE, device=args.device)
next_state_values[non_final_mask] = target_net(
    non_final_next_states.unsqueeze(1)
).max(1)[0].detach()
expected_state_action_values = (
    (next_state_values * GAMMA) + reward_batch
)
```

```
    #criterion = nn.SmoothL1Loss()
    criterion = nn.MSELoss()
    loss = criterion(
        state_action_values,
        expected_state_action_values.unsqueeze(1)
    )

    optimizer.zero_grad()
    loss.backward()
    for param in policy_net.parameters():
        param.grad.data.clamp_(-1, 1)
    optimizer.step()

def calculate_inf_norm(image1, image2):
    image1 = image1.detach().cpu().numpy()
    image2 = image2.detach().cpu().numpy()
    return np.amax(np.abs(image1 - image2))



def calculate_l2_norm(image1, image2):
    image1 = image1.detach().cpu().numpy()
    image2 = image2.detach().cpu().numpy()
    return np.sum((image1 - image2) ** 2)

total = 100
correct = 0
with torch.no_grad():
    for idx, (sample, target) in enumerate(data):
```

```
        if idx == total:
            break
        preds = model(sample.unsqueeze(1)).max(1)[1].item()
        if preds == target:
            correct += 1
print('Accuracy :', correct / total)


def perform_attack(sample, true_class, classifier, args):
    env = AttackEnv(
        image=sample,
        image_size=sample_image.shape[-1],
        n_channels=sample_image.shape[0],
        epsilon=0.3,
    )
    if args.norm == 'l_inf':
        norm_func = calculate_inf_norm
    else:
        norm_func = calculate_l2_norm

    total_queries = 0
    curr_queries = 0
    success = False
    state = env.reset()
    while total_queries <= 10000:
        action_position = select_action(state['image'])
        action = {'position': action_position, 'channel': 0, 'epsilon': args.eps / 2, 'prob':1.}
        next_state, reward, done, info = env.step(action)
        next_state['image'] = torch.clip(next_state['image'], 0, 1)
```

```
preds = model(next_state['image'].unsqueeze(0)).max(1)[1].item()
if preds != true_class:
    success = True
    reward = 1
else:
    reward = -0.1

if norm_func(sample, next_state['image']) > args.eps + 0.001:
    reward = -1
    next_state = env.reset()
    curr_queries = 0

memory.push(
    state['image'],
    action_position,
    next_state['image'],
    reward
)

state = next_state
optimize_model()
total_queries += 1
curr_queries += 1

if success:
    return True, total_queries

if total_queries % TARGET_UPDATE == 0:
    target_net.load_state_dict(policy_net.state_dict())
```

```
    return False, 10000

attack_correct = 0
success_attack_n_quesries = list()

for idx, (sample, target) in tqdm(enumerate(data), total=total):
    if idx == 200:
        break
    preds = model(sample.unsqueeze(1)).max(1)[1].item()
    print(preds)
    attack_success, n_queries = perform_attack(sample, target, model, args)
    print(attack_success, n_queries)
    if not attack_success:
        attack_correct += 1
    else:
        success_attack_n_quesries.append(n_queries)
print('Accuracy :', attack_correct / total * 100)
```