МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

НАЦІОНАЛЬНИЙ УНІВЕРСИТЕТ "ЛЬВІВСЬКА ПОЛІТЕХНІКА"

ВСТУП ДО PYTORCH ТА ГЕНЕРАЦІЯ МУЗИКИ ЗА ДОПОМОГОЮ RNN

МЕТОДИЧНІ ВКАЗІВКИ

до виконання лабораторної роботи № 5
з дисципліни «Штучний інтелект в ігрових застосунках»
для студентів бакалаврського рівня вищої освіти спеціальності 121
"Інженерія програмного забезпечення"

Львів -- 2025

Вступ до PyTorch та генерація музики за допомогою RNN: методичні вказівки до виконання лабораторної роботи №5 з дисципліни "Штучний інтелект в ігрових застосунках" для студентів першого (бакалаврського) рівня вищої освіти спеціальності 121 "Інженерія програмного забезпечення" . Укл.: О.Є. Бауск. -- Львів: Видавництво Національного університету "Львівська політехніка", 2025. -- 10 с.

Укладач: Бауск О.Є., к.т.н., асистент кафедри ПЗ

Відповідальний за випуск: Федасюк Д.В., доктор техн. наук, професор

Рецензенти: Федасюк Д.В., доктор техн. наук, професор

Задорожний І.М., асистент кафедри ПЗ

Тема роботи: Вступ до PyTorch та генерація музики за допомогою рекурентних нейронних мереж (RNN).

Мета роботи: Ознайомитись з основами бібліотеки глибокого навчання PyTorch, навчитись визначати та тренувати прості нейронні мережі, а також застосувати рекурентні нейронні мережі для задачі генерації музики у форматі ABC notation.

Теоретичні відомості

Вступ до PyTorch

PyTorch — це популярна бібліотека глибокого навчання з відкритим кодом, відома своєю гнучкістю, простотою використання та динамічним графом обчислень. Вона широко використовується в дослідженнях та промисловості для створення та тренування нейронних мереж.

Основні концепції PyTorch:

1. **Тензори (Tensors)**: Багатовимірні масиви, схожі на NumPy ndarrays, але з додатковою можливістю виконувати обчислення на GPU для прискорення. Тензори є основними будівельними блоками для даних у PyTorch.
2. **Автоматичне диференціювання (Autograd)**: PyTorch автоматично обчислює градієнти для тензорів. Модуль `torch.autograd` відстежує операції над тензорами, дозволяючи легко реалізовувати зворотне поширення помилки для тренування мереж.
3. **Модулі (`torch.nn.Module`)**: Клас `nn.Module` є базовим для всіх нейронних мереж у PyTorch. Він дозволяє інкапсулювати параметри моделі та операції в зручні об'єкти. Мережі можна будувати, комбінуючи існуючі модулі (шари) або створюючи власні.
4. **Оптимізатори (`torch.optim`)**: Містить реалізації різноманітних алгоритмів оптимізації (наприклад, SGD, Adam), які використовуються для оновлення параметрів моделі під час тренування.
5. **Функції втрат (`torch.nn`)**: Надає стандартні функції втрат (наприклад, CrossEntropyLoss, MSELoss), що використовуються для оцінки різниці між прогнозами моделі та реальними даними.

PyTorch дозволяє визначати моделі двома основними способами:

- `torch.nn.Sequential` : Контейнер для послідовного з'єднання модулів. Зручний для простих мереж, де дані проходять через шари один за одним.
- **Підкласи** `torch.nn.Module` : Більш гнучкий підхід, де користувач визначає власні класи, успадковуючи `nn.Module` . Це дозволяє створювати складні архітектури з розгалуженнями, пропусками з'єднань тощо.

Рекурентні Нейронні Мережі (RNN)

Рекурентні нейронні мережі (RNN) — це клас нейронних мереж, спеціально розроблений для роботи з послідовними даними, такими як текст, часові ряди або музика. На відміну від стандартних мереж прямого поширення, RNN мають "пам'ять" завдяки наявності рекурентних (зворотних) зв'язків, які дозволяють інформації з попередніх кроків впливати на обчислення на поточних кроках.

Основна ідея RNN полягає в тому, що вихід мережі на кроці (t) залежить не тільки від входу на кроці (t), але й від прихованого стану мережі на кроці (t-1). Прихований стан (h_t) оновлюється на кожному кроці

і слугує як зведена інформація про попередню історію послідовності.

$$h_t = f(W_{hh} h_{t-1} + W_{xh} x_t + b_h)$$ $$y_t = g(W_{hy} h_t + b_y)$$

де x_t — вхід на кроці t, h_t — прихований стан, y_t — вихід, W — матриці ваг, b — вектори зміщень, а f та g — функції активації.

Існують більш складні варіанти RNN, такі як LSTM (Long Short-Term Memory) та GRU (Gated Recurrent Unit), які краще справляються з проблемою зникаючих/вибухаючих градієнтів і можуть ефективніше навчатись на довгих послідовностях.

Генерація музики за допомогою RNN

RNN можна використовувати для генерації нової музики. Один з підходів — це "символьна RNN" (character RNN), де мережа навчається передбачати наступний символ у музичному записі, представленому у вигляді текстової послідовності.

ABC Notation: Це формат текстового запису музики, який використовує літери (A-G) для нот, цифри для тривалості, та інші символи для позначення ритму, тональності, тактів тощо. Наприклад:

```
X:1
T:The Legacy Jig
M:6/8
L:1/8
K:G
GFG GAB|cBc d2d|GFG GAB|cAF GFE|
GFG GAB|cBc dgf|ecA GcB|cAF GFE:|
```

Процес генерації:

1. **Підготовка даних**: Музичний корпус у форматі ABC notation перетворюється на послідовність символів. Створюється словник унікальних символів.
2. **Побудова моделі**: Створюється RNN (наприклад, LSTM), яка приймає на вхід послідовність символів і намагається передбачити наступний символ.
3. **Тренування**: Модель тренується на великому корпусі музики, мінімізуючи функцію втрат (наприклад, перехресну ентропію) між передбаченими та реальними наступними символами.
4. **Генерація (Sampling)**: Після тренування модель використовується для генерації нової музики. Починаючи з початкової послідовності ("seed"), модель ітеративно передбачає наступний символ, додає його до послідовності і використовує оновлену послідовність як вхід для наступного кроку. Ймовірнісний розподіл виходу мережі дозволяє вносити випадковість у процес генерації.

Хід роботи

1. Загальні відомості про лабораторну роботу.

Лабораторна робота складається з двох частин, які виконуються за допомогою Jupyter Notebook у середовищі Google Colab.

- **Частина 1: Вступ до PyTorch**: Ознайомлення з базовими операціями над тензорами, автоматичним диференціюванням та визначенням простих нейронних мереж.
- **Частина 2: Генерація музики з RNN**: Побудова, тренування та використання RNN для генерації музики у форматі ABC notation.

1.1. Проаналізуйте наданий код та переконайтеся, що ви розумієте логіку виконання кожного кроку.

1.2. Робіть скріншоти ваших етапів роботи та результатів для звіту.

1.3. Там, де в коді написан закоментований блок з позначкою `# TODO`, виконайте завдання та замість "# TODO" напишіть правильну реалізацію.

1.4. При потребі використовуйте документацію до PyTorch: https://pytorch.org/docs/stable/index.html

УВАГА! Пам'ятайте про обмеження Google Colab щодо часу роботи сесії та збереження даних. Завантажені файли та встановлені бібліотеки можуть зникнути після перезапуску середовища виконання. Плануйте свій час або будьте готові повторно виконати необхідні кроки.

2. Підготовка середовища Google Colab.

2.1. Відкрийте новий Google Colab ноутбук або використовуйте існуючий: https://colab.research.google.com/

2.2. Переконайтеся, що середовище виконання використовує GPU (Runtime -> Change runtime type -> Hardware accelerator -> GPU), якщо це необхідно для прискорення обчислень, хоча для цих завдань CPU може бути достатньо.

3. Частина 1: Вступ до PyTorch.

У цій частині ми ознайомимося з основами бібліотеки PyTorch та її можливостями для глибокого навчання.

3.1. Що таке PyTorch?

Встановлюємо необхідні бібліотеки:

```
# PyTorch вже встановлена у Google Colab за замовчуванням, немає потреби встановлювати сам паке
import torch
import torch.nn as nn

# Встановлюємо пакет для вивчення глибоких нейронних мереж від MIT
!pip install mitdeeplearning --quiet
import mitdeeplearning as mdl

import numpy as np
import matplotlib.pyplot as plt
```

PyTorch — це бібліотека машинного навчання, схожа на TensorFlow. У своїй основі PyTorch надає інтерфейс для створення та маніпулювання тензорами, які є структурами даних, що можна розглядати як багатовимірні масиви. Тензори представлені як n-вимірні масиви базових типів даних, таких як рядок або ціле число — вони забезпечують спосіб узагальнення векторів і матриць до вищих вимірів.

PyTorch надає можливість виконувати обчислення на цих тензорах, визначати нейронні мережі та ефективно їх навчати.

`shape` тензора PyTorch визначає кількість його вимірів та розмір кожного виміру. `ndim` або `dim` тензора PyTorch надає кількість вимірів (n-вимірів) — це еквівалентно рангу тензора (як використовується в TensorFlow), і ви також можете розглядати це як порядок тензора.

Почнемо з створення декількох тензорів та вивчення їх властивостей:

```
integer = torch.tensor(1234)
decimal = torch.tensor(3.14159265359)

print(f"`integer` is a {integer.ndim}-d Tensor: {integer}")
print(f"`decimal` is a {decimal.ndim}-d Tensor: {decimal}")
```

Вектори та списки можна використовувати для створення 1-вимірних тензорів:

```
fibonacci = torch.tensor([1, 1, 2, 3, 5, 8])
count_to_100 = torch.tensor(range(100))

print(f"`fibonacci` is a {fibonacci.ndim}-d Tensor with shape: {fibonacci.shape}")
print(f"`count_to_100` is a {count_to_100.ndim}-d Tensor with shape: {count_to_100.shape}")
```

Створимо 2-вимірні тензори (матриці) та тензори вищих рангів. В обробці зображень та комп'ютерному зорі будемо використовувати 4-вимірні тензори з вимірами, що відповідають розміру батчу, кількості кольорових каналів, висоті та ширині зображення.

```
### Визначення тензорів вищого порядку ###

'''ЗАВДАННЯ: Визначте 2-вимірний тензор'''
matrix = # TODO

assert isinstance(matrix, torch.Tensor), "matrix повинен бути об'єктом torch Tensor"
assert matrix.ndim == 2

'''ЗАВДАННЯ: Визначте 4-вимірний тензор.'''
# Використайте torch.zeros для ініціалізації 4-вимірного тензора нулів розміром 10 x 3 x 256 x
#   Можна уявити це як 10 зображень, де кожне зображення є RGB 256 x 256.
images = # TODO

assert isinstance(images, torch.Tensor), "images повинен бути об'єктом torch Tensor"
assert images.ndim == 4, "images повинен мати 4 виміри"
assert images.shape == (10, 3, 256, 256), "images має неправильну форму"
print(f"images є {images.ndim}-d тензором з формою: {images.shape}")
```

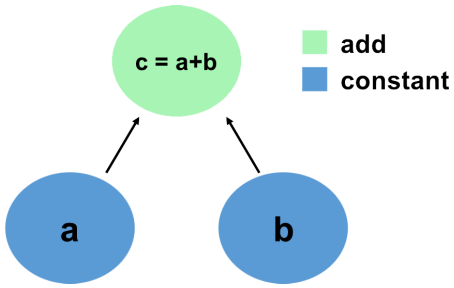
Як ви бачили, форма (shape) тензора вказує на кількість елементів у кожному вимірі тензора. Форма дуже корисна, і ми будемо часто її використовувати. Ви також можете використовувати зрізи (slicing) для доступу до підтензорів всередині тензора вищого рангу:

```
row_vector = matrix[1]
column_vector = matrix[:, 1]
scalar = matrix[0, 1]

print(f"`row_vector`: {row_vector}")
print(f"`column_vector`: {column_vector}")
print(f"`scalar`: {scalar}")
```

3.2. Обчислення на тензорах

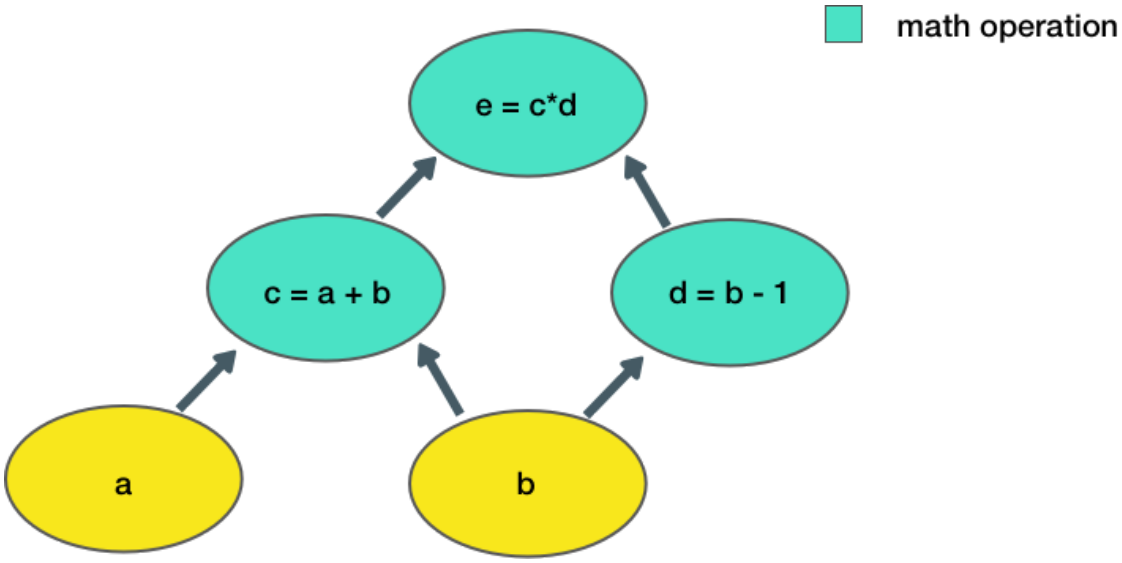
Зручний спосіб думати про та візуалізувати обчислення в фреймворках машинного навчання, таких як PyTorch — це в термінах графів. Ми можемо визначити цей граф у термінах тензорів, які містять дані, та математичних операцій, які діють на ці тензори в певному порядку. Розглянемо простий приклад і визначимо це обчислення за допомогою PyTorch:




```
# Створюємо вузли графа та ініціалізуємо значення
a = torch.tensor(15)
b = torch.tensor(61)

# Додаємо їх!
c1 = torch.add(a, b)
c2 = a + b  # PyTorch перевизначає (overrides) операцію "+" для роботи з тензорами
print(f"c1: {c1}")
print(f"c2: {c2}")
```

Тепер розглянемо трохи складніший приклад:



Тут ми беремо два входи, `a, b`, і обчислюємо вихід `e`. Кожен вузол у графі представляє операцію, яка приймає деякий вхід, виконує деяке обчислення та передає свій вихід іншому вузлу.

```
'''ЗАВДАННЯ: Визначте функцію обчислення'''
def func(a, b):
    c = torch.add(...) # TODO
    d = torch.subtract(...) # TODO
    e = torch.multiply(...) # TODO
    return e
```

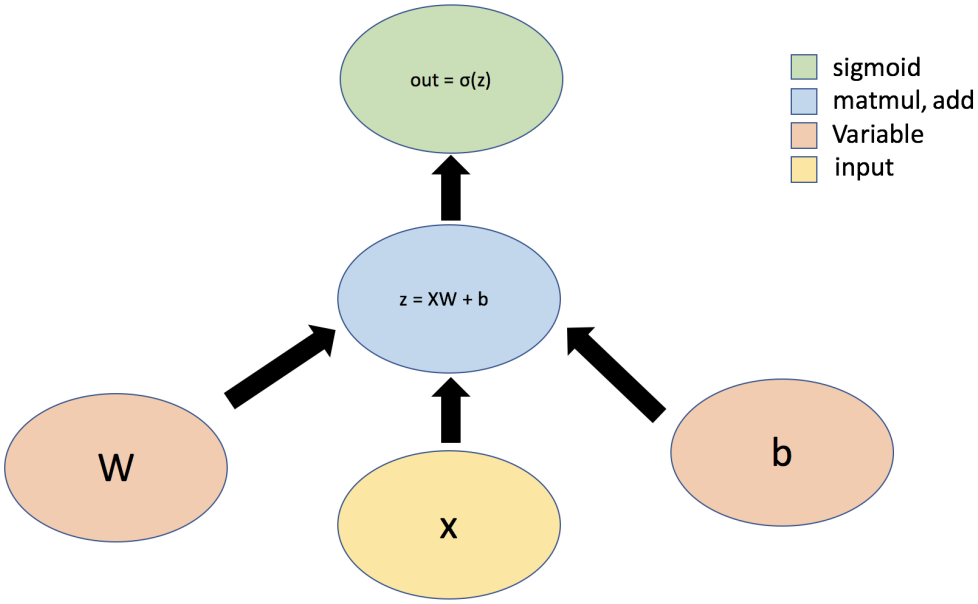
Тепер ми можемо викликати цю функцію для виконання обчислювального графа, задавши деякі вхідні значення a, b:

```
# Приклад значень для a, b
a, b = 1.5, 2.5
# Виконання обчислення
e_out = func(a, b)
print(f"e_out: {e_out}")
```

3.3. Нейронні мережі в PyTorch

Ми також можемо визначати нейронні мережі в PyTorch. PyTorch використовує `torch.nn.Module`, який служить базовим класом для всіх модулів нейронних мереж у PyTorch і, таким чином, забезпечує фреймворк для побудови та навчання нейронних мереж.

Розглянемо приклад простого перцептрона, визначеного лише одним щільним (повнозв'язаним або лінійним) шаром: $ y = \sigma(Wx + b) $, де W представляє матрицю ваг, b — зміщення, x — вхід, σ — сигмоїдна функція активації, а y — вихід.



Ми будемо використовувати `torch.nn.Module` для визначення шарів — будівельних блоків нейронних мереж. Шари реалізують поширені операції нейронних мереж. У PyTorch, коли ми реалізуємо шар, ми створюємо підклас `nn.Module` і визначаємо параметри шару як атрибути нашого нового класу. Ми

також визначаємо та перевизначаємо функцію `forward`, яка буде визначати обчислення прямого проходу, що виконується на кожному кроці. Усі класи, що підкласують `nn.Module`, повинні перевизначати функцію `forward`.

```
# Визначення щільного шару
class DenseLayer(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(DenseLayer, self).__init__()
        # Визначаємо та ініціалізуємо параметри: матрицю ваг W та зміщення b
        # Зауважте, що ініціалізація параметрів є випадковою!
        self.W = torch.nn.Parameter(torch.randn(num_inputs, num_outputs))
        self.bias = torch.nn.Parameter(torch.randn(num_outputs))

    def forward(self, x):
        # Визначаємо операцію для z
        z = torch.matmul(x, self.W) + self.bias
        # Визначаємо операцію для виходу
        # Використовуйте torch.sigmoid для сигмоїдної активації з аргументом z
        y = # TODO
        return y
```

Тепер ми можемо створити екземпляр нашого шару та перевірити його роботу:

```
# Перевірка шару
num_inputs = 2
num_outputs = 3
layer = DenseLayer(num_inputs, num_outputs)
x_input = torch.tensor([[1, 2.]])
y = layer(x_input)

print(f"input shape: {x_input.shape}")
print(f"output shape: {y.shape}")
print(f"output result: {y}")
```

PyTorch зручно визначив ряд `nn.Modules` (або шарів), які часто використовуються в нейронних мережах, наприклад, `nn.Linear` або `nn.Sigmoid`. Тепер, замість використання одного `Module` для визначення нашої простої нейронної мережі, ми використаємо модуль `nn.Sequential` з PyTorch і один шар `nn.Linear`. З Sequential API ви можете легко створювати нейронні мережі, складаючи шари разом, як будівельні блоки.

```
# Визначення нейронної мережі за допомогою Sequential API PyTorch
n_input_nodes = 2
n_output_nodes = 3

# Визначення моделі
model = nn.Sequential(
    nn.Linear(n_input_nodes, n_output_nodes),
    nn.Sigmoid()
)

# Тестування моделі
x_input = torch.tensor([[1, 2.]])
```

```
model_output = model(x_input)
print(f"input shape: {x_input.shape}")
print(f"output shape: {model_output.shape}")
print(f"output result: {model_output}")
```

Ми також можемо створювати більш гнучкі моделі, підкласуючи `nn.Module`. Клас `nn.Module` дозволяє нам гнучко групувати шари разом для визначення нових архітектур.

Як ми бачили раніше з `DenseLayer`, ми можемо підкласати `nn.Module` для створення класу для нашої моделі, і визначити прямий прохід через мережу за допомогою функції `forward`. Підкласування надає гнучкість для визначення власних шарів, власних циклів навчання, власних функцій активації та власних моделей. Визначимо ту саму нейронну мережу модель, як і вище (тобто лінійний шар з сигмоїдною активацією після нього), тепер використовуючи підкласування та використовуючи вбудований лінійний шар від `nn.Linear`.

```
# Визначення моделі за допомогою підкласування
class LinearWithSigmoidActivation(nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(LinearWithSigmoidActivation, self).__init__()
        # Визначення моделі з одним лінійним шаром та сигмоїдною активацією
        self.linear = nn.Linear(num_inputs, num_outputs)
        # Використовуйте nn.Sigmoid для сигмоїдної активації
        self.activation = # TODO

    def forward(self, inputs):
        linear_output = self.linear(inputs)
        output = self.activation(linear_output)
        return output
```

Тепер ми можемо створити екземпляр нашої моделі та перевірити її роботу:

```
# Тестування моделі
n_input_nodes = 2
n_output_nodes = 3
model = LinearWithSigmoidActivation(n_input_nodes, n_output_nodes)
x_input = torch.tensor([[1, 2.]])
y = model(x_input)
print(f"input shape: {x_input.shape}")
print(f"output shape: {y.shape}")
print(f"output result: {y}")
```

Важливо, що nn.Module надає нам багато гнучкості для визначення власних моделей. Наприклад, ми можемо використовувати булеві аргументи у функції forward для визначення різної поведінки мережі, наприклад, різної поведінки під час навчання та виведення. Припустимо, що в деяких випадках ми хочемо, щоб наша мережа просто виводила вхідні дані без будь-яких змін. Ми визначаємо булевий аргумент isidentity для керування цією поведінкою:

```
### Власна поведінка з підкласуванням nn.Module ###

class LinearButSometimesIdentity(nn.Module):
    def __init__(self, num_inputs, num_outputs):
```

```
        super(LinearButSometimesIdentity, self).__init__()
        self.linear = nn.Linear(num_inputs, num_outputs)

    '''TODO: Реалізуйте поведінку, коли мережа виводить вхідні дані без змін,
        під контролем аргументу isidentity.'''
    def forward(self, inputs, isidentity=False):
        if isidentity:
            return inputs
        else:
            return self.linear(inputs)
```

Давайте перевіримо цю поведінку:

```
# Тестування IdentityModel
model = LinearButSometimesIdentity(num_inputs=2, num_outputs=3)
x_input = torch.tensor([[1, 2.]])

'''TODO: передайте вхідні дані в модель і викличте з та без опції ідентичності входу.'''
out_with_linear = model(
  # TODO: передайте вхідні дані
)
out_with_identity = model(
  # TODO: передайте вхідні дані
  # TODO: вкажіть isidentity=True
)

print(f"input: {x_input}")
print("Вихід лінійної мережі: {}; вихід ідентичної мережі: {}".format(out_with_linear, out_with
```

Тепер, коли ми навчилися визначати шари та моделі в PyTorch, використовуючи як Sequential API, так і підкласування nn.Module, ми готові звернути увагу на те, як фактично реалізувати навчання мережі за допомогою зворотного поширення помилки.

3.4. Автоматичне диференціювання в PyTorch

У PyTorch `torch.autograd` використовується для автоматичного диференціювання, що є критичним для навчання моделей глибокого навчання з використанням алгоритму зворотного поширення помилки.

Ми використовуємо метод PyTorch `.backward()` для відстеження операцій для обчислення градієнтів. На тензорі атрибут `requires_grad` контролює, чи повинен autograd записувати операції на цьому тензорі. Коли прямий прохід здійснюється через мережу, PyTorch динамічно будує обчислювальний граф; потім для обчислення градієнта викликається метод `backward()` для виконання зворотного поширення помилки.

Давайте обчислимо градієнт $ y = x^2 $:

```
# y = x^2
# Приклад: x = 3.0
x = torch.tensor(3.0, requires_grad=True)
y = x ** 2
y.backward()  # Обчислити градієнт
```

```
dy_dx = x.grad
print("dy_dx of y=x^2 at x=3.0 is: ", dy_dx)
```

Автоматичне диференціювання та стохастичний градієнтний спуск (SGD) широко використовуються для оптимізації функції втрат при навчанні нейронних мереж.

```
# Ініціалізуємо випадкове значення для нашого початкового x
x = torch.randn(1)
print(f"Initializing x={x.item()}")

learning_rate = 1e-2  # Швидкість навчання
history = []
x_f = 4  # Цільове значення

# Ми запустимо градієнтний спуск протягом кількох ітерацій. На кожній ітерації ми обчислюємо вт
# обчислюємо похідну втрат по відношенню до x і виконуємо оновлення.
for i in range(500):
    x = torch.tensor([x], requires_grad=True)

    # Обчислюємо втрати як квадрат різниці між x та x_f
    loss = (x - x_f)**2

    # Зворотне поширення через втрати для обчислення градієнтів
    loss.backward()

    # Оновлюємо x з градієнтним спуском
    x = x.item() - learning_rate * x.grad

    history.append(x.item())

# Побудуємо графік еволюції x при оптимізації до x_f!
plt.plot(history)
plt.plot([0, 500], [x_f, x_f])
plt.legend(('Прогноз', 'Істина'))
plt.xlabel('Ітерація')
plt.ylabel('Значення x')
plt.show()
```

4. Частина 2: Генерація музики з RNN.

У цій частині ми дослідимо побудову рекурентної нейронної мережі (RNN) для генерації музики за допомогою PyTorch. Ми навчимо модель вивчати шаблони в нотному тексті у форматі ABC notation, а потім використаємо цю модель для генерації нової музики.

Спочатку завантажимо репозиторій курсу, встановимо залежності та імпортуємо необхідні пакети для цього лабораторного заняття.

Ми будемо використовувати Comet ML для відстеження розробки та тренувань нашої моделі. Спочатку зареєструйтеся (ви можете використовувати свій Google або Github обліковий запис):

https://www.comet.com/signup?utm_source=mit_dl&utm_medium=partner&utm_content=github

Вам потрібно буде згенерувати новий персональний API ключ, який ви можете знайти на першій сторінці

'Get Started with Comet' або натиснувши на '?' в правому верхньому кутку та вибравши 'Quickstart Guide'. Введіть цей API ключ як глобальну змінну COMET_API_KEY.

```
!pip install comet_ml > /dev/null 2>&1
import comet_ml
# TODO: Введіть ваш API ключ тут!! інструкції вище
COMET_API_KEY = ""

# Імпортувати PyTorch та інші необхідні бібліотеки
import torch
import torch.nn as nn
import torch.optim as optim

# Завантажити та імпортувати пакет MIT Introduction to Deep Learning
!pip install mitdeeplearning --quiet
import mitdeeplearning as mdl

# Імпортувати всі решту пакетів
import numpy as np
import os
import time
import functools
from IPython import display as ipythondisplay
from tqdm import tqdm
from scipy.io.wavfile import write
!apt-get install abcmidi timidity > /dev/null 2>&1

# Перевірка, чи ми використовуємо GPU, якщо ні, то перемикаємо runtimes
# використовуючи Runtime > Change Runtime Type > GPU
assert torch.cuda.is_available(), "Будь ласка, увімкніть GPU з налаштувань runtime"
assert COMET_API_KEY != "", "Будь ласка, введіть ваш API ключ Comet"
```

4.1. Підготовка даних

Ми використовуємо набір даних з тисяч ірландських народних пісень, представлених у нотації ABC. Спочатку завантажимо набір даних та вивчимо його:

```
# Завантаження набору даних
songs = mdl.lab1.load_training_data()

# Виведення однієї з пісень для детального розгляду
example_song = songs[0]
print("\nПриклад пісні: ")
print(example_song)
```

Ми можемо легко конвертувати пісню в нотації ABC у звукову хвилю та відтворити її:

```
# Конвертація нотації ABC у аудіофайл та прослуховування
mdl.lab1.play_song(example_song)
```

Важливо враховувати, що ця нотація музики містить не лише інформацію про ноти, а й метаінформацію, таку як назва пісні, тональність та темп. Кількість різних символів у текстовому файлі впливає на складність завдання навчання. Це стане важливим під час створення числового представлення текстових даних.

```
# Об'єднання списку рядків пісень в один рядок, що містить всі пісні
songs_joined = "\n\n".join(songs)

# Знаходження всіх унікальних символів в об'єднаному рядку
vocab = sorted(set(songs_joined))
print("У наборі даних", len(vocab), "унікальних символів")
```

4.2. Векторизація тексту

Давайте повернемося назад і розглянемо нашу задачу прогнозування. Ми намагаємося навчити модель RNN вивчати шаблони в ABC музиці, і потім використати цю модель для генерації (тобто прогнозування) нової пісні на основі цієї навченої інформації.

Розглянемо це як завдання: даний символ або послідовність символів, який є найбільш ймовірним наступним символом? Ми навчимо модель виконувати цю задачу.

Щоб досягти цього, ми подамо послідовність символів до моделі, і навчимо модель прогнозувати вихід, тобто наступний символ на кожному кроці. RNN підтримують внутрішній стан, який залежить від попередньо побачених елементів, тому інформація про всі символи, побачені до даного моменту, буде врахована при генерації прогнозу.

Перш ніж почати навчання нашої моделі RNN, нам потрібно створити числове представлення нашого текстового набору даних. Для цього ми створимо дві таблиці пошуку: одну, яка відображає символи в числа, а другу, яка відображає числа назад у символи.

```
# Створення відображення від символу до унікального індексу
char2idx = {u: i for i, u in enumerate(vocab)}

# Створення відображення від індексів до символів.
idx2char = np.array(vocab)

# Функція для конвертації рядка у векторизоване представлення
def vectorize_string(string):
    return np.array([char2idx[char] for char in string])

vectorized_songs = vectorize_string(songs_joined)
```

Це дає нам числове представлення для кожного символу.

Зверніть увагу, що унікальні символи (тобто наш словник) у тексті відображаються як індекси від 0 до len(unique).

Подивимося на перші 20 символів нашого словника:

```
print('{')
for char, _ in zip(char2idx, range(20)):
    print('  {:4s}: {:3d},'.format(repr(char), char2idx[char]))
print('  ...\n}')
```

Наш наступний крок - це фактично розділити текст на приклади послідовностей, які ми будемо використовувати під час навчання. Кожна вхідна послідовність, яку ми подаємо в нашу RNN, буде містити seq_length символів з тексту. Нам також потрібно визначити цільову послідовність для кожної вхідної послідовності, яка буде використовуватися при навчанні RNN для прогнозування наступного символу. Для кожного входу відповідна ціль буде містити таку ж довжину тексту, але зміщену на один символ вправо.

Для цього ми розіб'ємо текст на фрагменти довжиною seq_length+1. Припустимо, seq_length дорівнює 4, а наш текст - "Hello". Тоді наша вхідна послідовність - "Hell", а цільова послідовність - "ello".

Метод batch потім дозволить нам перетворити цей потік індексів символів у послідовності бажаного розміру.

```
def get_batch(vectorized_songs, seq_length, batch_size):
    # довжина векторізованого рядка пісень
    n = vectorized_songs.shape[0] - 1
    # випадково вибираємо початкові індекси для прикладів у навчальному батчі
    idx = np.random.choice(n - seq_length, batch_size)

    '''конструюємо список вхідних послідовностей для навчального батчу'''
    input_batch = [vectorized_songs[i:i+seq_length] for i in idx]

    '''конструюємо список цільових послідовностей для навчального батчу'''
    output_batch = [vectorized_songs[i+1:i+seq_length+1] for i in idx]

    # перетворюємо вхідні та цільові послідовності у тензори
    x_batch = torch.tensor(input_batch, dtype=torch.long)
    y_batch = torch.tensor(output_batch, dtype=torch.long)

    return x_batch, y_batch

# Виконуємо деякі прості тести, щоб переконатися, що функція батчу працює правильно
test_args = (vectorized_songs, 10, 2)
x_batch, y_batch = get_batch(*test_args)
assert x_batch.shape == (2, 10), "x_batch -- форма неправильна"
assert y_batch.shape == (2, 10), "y_batch -- форма неправильна"
print("Функція батчу працює правильно")
```

Для кожного з цих векторів, кожен індекс обробляється на окремому кроці часу. Тобто, на кроці часу 0, модель отримує індекс для першого символу в послідовності і намагається передбачити індекс наступного символу. На наступному кроці часу вона робить те саме, але RNN враховує інформацію з попереднього кроку, тобто свій оновлений стан, на додаток до поточного входу.

Ми можемо зробити це більш конкретним, розглянувши, як це працює на перших кількох символах нашого тексту:

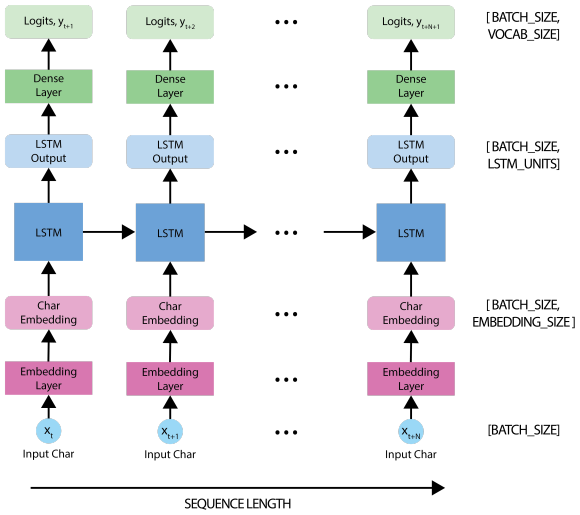
```
x_batch, y_batch = get_batch(vectorized_songs, seq_length=5, batch_size=1)

for i, (input_idx, target_idx) in enumerate(zip(x_batch[0], y_batch[0])):
    print("Крок {:3d}".format(i))
    print("Вхід: {} ({:s})".format(input_idx, repr(idx2char[input_idx.item()])))
    print("Очікуваний вихід: {} ({:s})".format(target_idx, repr(idx2char[target_idx.item()])))
```

4.3. Модель рекурентної нейронної мережі (RNN)

Тепер ми готові визначити та навчити модель RNN на нашому музичному наборі даних ABC, а потім використати цю навчену модель для генерації нової пісні. Ми навчатимемо нашу RNN, використовуючи батчі фрагментів пісень з нашого набору даних.

Модель базується на архітектурі LSTM, де ми використовуємо вектор стану для зберігання інформації про часові відносини між послідовними символами. Остаточний вихід LSTM потім передається у повністю з'єднаний (так званий "щільний" або "fully connected") лінійний шар `nn.Linear`, де ми виводимо softmax по кожному символу в словнику, а потім вибираємо зразки з цього розподілу для прогнозування наступного символу.



Ми будемо використовувати `nn.Module` для визначення моделі. Три компоненти використовуються для визначення моделі:

- `nn.Embedding`: Це вхідний шар, що складається з навчальної таблиці пошуку, яка відображає числа кожного символу у вектор з розмірністю `embedding_dim`.
- `nn.LSTM`: Наша мережа LSTM, з розміром `hidden_size`.
- `nn.Linear`: Вихідний шар, з `vocab_size` виходами.

```
# Визначення моделі RNN
class LSTMModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size

        # Шар 1: Шар вкладень для перетворення індексів у щільні вектори
        self.embedding = nn.Embedding(vocab_size, embedding_dim)
```

```
        # Шар 2: LSTM з розміром прихованого стану `hidden_size`
        self.lstm = nn.LSTM(embedding_dim, hidden_size, batch_first=True)

        # Шар 3: Лінійний шар, який перетворює вихід LSTM у розмір словника
        # TODO: передайте правільні параметри в наступній стрічці:
        # а саме розмір прихованого стану мережі та розмір словника
        self.fc = nn.Linear("""TODO: передати правільні параметри замість цієї строки""")

    def init_hidden(self, batch_size, device):
        # Ініціалізація прихованого стану та стану комірки нулями
        return (torch.zeros(1, batch_size, self.hidden_size).to(device),
                torch.zeros(1, batch_size, self.hidden_size).to(device))

    def forward(self, x, state=None, return_state=False):
        x = self.embedding(x)

        if state is None:
            state = self.init_hidden(x.size(0), x.device)
        out, state = self.lstm(x, state)

        out = self.fc(out)
        return out if not return_state else (out, state)
```

```
# Інстанціюємо модель! Створіть просту модель зі стандартними гіперпараметрами.
#     Ви отримаєте можливість змінити їх пізніше.
vocab_size = len(vocab)
embedding_dim = 256
hidden_size = 1024
batch_size = 8

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

model = LSTMModel(vocab_size, embedding_dim, hidden_size).to(device)

# Виведіть свою модель
print(model)
```

Звичайно, завжди корисно перевірити, що модель веде себе так, як очікується.

Ми можемо швидко перевірити шари в моделі, розмір виходу кожного з шарів, розмір батчу та розмірність виходу. Зверніть увагу, що модель може бути запущена на вхідних даних будь-якої довжини.

```
# Перевірка моделі
x, y = get_batch(vectorized_songs, seq_length=100, batch_size=32)
x = x.to(device)
y = y.to(device)

pred = model(x)
print("Форма входу:      ", x.shape, " # (batch_size, sequence_length)")
print("Форма виходу: ", pred.shape, "# (batch_size, sequence_length, vocab_size)")
```

4.4. Навчання моделі

Розглянемо, що наша ненавчена модель предсказує.

Щоб отримати фактичні прогнози від моделі, ми вибираємо з розподілу виходу, який визначається torch.softmax над нашим словником символів. Це дасть нам фактичні індекси символів. Це означає, що ми використовуємо категоріальний розподіл для вибірки над прикладом прогнозу. Це дає прогноз наступного символу (конкретно його індекс) на кожному кроці. torch.multinomial вибирає з категоріального розподілу для генерації прогнозів.

Зверніть увагу, що ми вибираємо з цього розподілу ймовірностей, а не просто беремо argmax, що може призвести до того, що модель застряне в повторювальному циклі.

Спробуємо це вибіркове виведення для першого прикладу в батчі.

```
sampled_indices = torch.multinomial(torch.softmax(pred[0], dim=-1), num_samples=1)
sampled_indices = sampled_indices.squeeze(-1).cpu().numpy()
sampled_indices
```

Ми можемо тепер декодувати ці символи, щоб побачити текст, предсказаний ненавченою моделлю:

```
print("Input: \n", repr("".join(idx2char[x[0].cpu()])))
print()
print("Next Char Predictions: \n", repr("".join(idx2char[sampled_indices])))
```

Як ви можете бачити, текст, предсказаний ненавченою моделлю, є досить безглуздим, і це означає, що нам потрібно навчити модель.

На цьому етапі ми можемо розглядати нашу проблему прогнозування наступного символу як стандартну класифікаційну задачу.

Дано попередній стан RNN, а також вхід у даний часовий крок, ми хочемо передбачити клас наступного символу, тобто фактично передбачити наступний символ.

Щоб навчити нашу модель на цій класифікаційній задачі, ми можемо використовувати форму втрати перехресної ентропії (від'ємна логарифмічна втрата).

Конкретно, ми будемо використовувати PyTorch's CrossEntropyLoss, як це об'єднує застосування логарифмічного softmax (LogSoftmax) та від'ємної логарифмічної втрати (NLLLoss) в одному класі і приймає цільові цілі для класифікаційних завдань.

Ми хочемо обчислити втрати, використовуючи дійсні цілі та прогнозовані цілі.

Давайте визначимо функцію для обчислення втрат, і тоді використаємо цю функцію для обчислення втрат за допомогою прикладів прогнозів з ненавченої моделі.

```
# Визначення функції втрат

cross_entropy = nn.CrossEntropyLoss()

def compute_loss(labels, logits):
    """
    Параметри:
```

```
      labels: (batch_size, sequence_length)
      logits: (batch_size, sequence_length, vocab_size)

    Вихід:
      loss: скалярна втрата перехресної ентропії по батчу та довжині послідовності
    """

    # Сформуємо логіти так, щоб розмір логітів був (B * L, V)
    batched_logits = logits.view(-1, logits.size(-1))

    # Сформуємо цілі так, щоб розмір цілей був (B * L,)
    batched_labels = labels.view(-1)

    # Обчислимо втрати перехресної ентропії, використовуючи наступні символи та прогнози.
    # TODO: використайте визначені вище логіти та лейбели як аргументи для обчислення втрат.
    loss = cross_entropy(
      """TODO: використати визначені вище логіти
      та лейбели як аргументи для функції обчислення втрат"""
      )
    return loss
```

```
# Обчислимо втрати на прогнозах з ненавченої моделі з попереднього кроку.
y.shape  # (batch_size, sequence_length)
pred.shape  # (batch_size, sequence_length, vocab_size)

# Обчислимо втрати, використовуючи дійсні цілі та прогнозовані цілі.
example_batch_loss = compute_loss(y, pred)

print(f"Форма прогнозів: {pred.shape} # (batch_size, sequence_length, vocab_size)")
print(f"Середня втрата:      {example_batch_loss.mean().item()}")
```

Давайте визначимо деякі гіперпараметри для тренування моделі.

Щоб почати, ми надамо деякі дефолтні значення для деяких параметрів.

```
# Гіперпараметри та оптимізація

vocab_size = len(vocab)

# Параметри моделі:
params = dict(
  num_training_iterations = 3000,  # Збільште це, щоб тренувати довше
  batch_size = 8,  # Експериментуйте між 1 і 64
  seq_length = 100,  # Експериментуйте між 50 і 500
  learning_rate = 5e-3,  # Експериментуйте між 1e-5 і 1e-1
  embedding_dim = 256,
  hidden_size = 1024,  # Експериментуйте між 1 і 2048
)

# Розташування контрольної точки:
checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "my_ckpt")
os.makedirs(checkpoint_dir, exist_ok=True)
```

Тепер ми можемо налаштувати відстеження експериментів з Comet.

`Experiment` є основними об'єктами в Comet і дозволять нам відстежувати тренування та розвиток моделі.

Тут ми написали коротку функцію для створення нового експерименту Comet. Зверніть увагу, що в цьому настроюванні, коли гіперпараметри змінюються, ви можете запустити функцію create_experiment() для створення нового експерименту.

Всі експерименти, визначені з однаковим project_name, будуть існувати під цим проектом у вашому інтерфейсі Comet.

```
# Створення експерименту Comet для відстеження тренування

def create_experiment():
  # завершення будь-яких попередніх експериментів
  if 'experiment' in locals():
    experiment.end()

  # ініціалізація експерименту Comet для відстеження
  experiment = comet_ml.Experiment(
                 api_key=COMET_API_KEY,
                 project_name="6S191_Lab1_Part2")
  # відстеження гіперпараметрів, визначених вище, до експерименту
  for param, value in params.items():
    experiment.log_parameter(param, value)
  experiment.flush()

  return experiment
```

Тепер ми готові визначити нашу операцію тренування -- а саме визначити **оптимізатор** та **тривалість тренування** -- і використати цю функцію для тренування моделі.

Ви зможете експериментувати з вибором оптимізатора та тривалості тренування ваших моделей, і побачити, як ці зміни впливають на вихід мережі.

Деякі оптимізатори, які ви можете спробувати: 'Adam', 'Adagrad'.

Ми інстанціюємо нову модель та оптимізатор, і готуємо їх для тренування. Потім ми використовуємо loss.backward() для виконання зворотного поширення.

Нарешті, щоб оновити параметри моделі на основі обчислених градієнтів, ми використовуємо крок з оптимізатором, використовуючи optimizer.step().

Ми також генеруємо друк про прогрес моделі під час тренування, що допоможе нам легко візуалізувати, чи ми **мінімізуємо функцію втрати**.

```
### Визначення оптимізатора та операції тренування ###

### Інстанціювати нову модель LSTMModel для тренування, використовуючи гіперпараметри, визначен
model = LSTMModel(vocab_size, params["embedding_dim"], params["hidden_size"])

# Перемістити модель на GPU
```

```
model.to(device)

### Інстанціювати оптимізатор з його швидкістю навчання.
### Перевірте веб-сайт PyTorch для списку підтримуваних оптимізаторів.
### https://pytorch.org/docs/stable/optim.html
### Спробуємо використовувати оптимізатор Adam для початку.
optimizer = torch.optim.Adam(model.parameters(), lr=params["learning_rate"])

def train_step(x, y):
  # Установити режим моделі на тренування
  model.train()

  # Зануліть градієнти для кожного кроку
  optimizer.zero_grad()

  # Прямий прохід
  y_hat = model(x) # TODO

  # Обчислити функцію втрат
  loss = compute_loss(y, y_hat) # TODO

  # Зворотний прохід
  '''TODO: завершити обчислення градієнта та крок оновлення.
    Пам'ятайте, що в PyTorch є два кроки для циклу тренування:
    1. Зворотний прохід втрат за допомогою метода backward()
    2. Оновлення параметрів моделі за допомогою метода step() оптимізатора
  '''
  loss.XXX() # TODO -- використайте правильний метод для зворотного поширення
  optimizer.XXX() # TODO -- використайте правильний метод для оновлення параметрів

  return loss

####################
# Починаємо тренування
####################

history = []
plotter = mdl.util.PeriodicPlotter(sec=2, xlabel='Iterations', ylabel='Loss')
experiment = create_experiment()

if hasattr(tqdm, '_instances'): tqdm._instances.clear()
for iter in tqdm(range(params["num_training_iterations"])):

    # Отримати батч та пропустити його через мережу
    x_batch, y_batch = get_batch(vectorized_songs, params["seq_length"], params["batch_size"])

    # Конвертувати numpy масиви у тензори PyTorch
    x_batch = torch.tensor(x_batch, dtype=torch.long).to(device)
    y_batch = torch.tensor(y_batch, dtype=torch.long).to(device)

    # Виконати крок тренування
    loss = train_step(x_batch, y_batch)

    # Записати втрату до інтерфейсу Comet
    experiment.log_metric("loss", loss.item(), step=iter)

    # Оновити панель прогресу та візуалізувати у ноутбуці
```

```
    history.append(loss.item())
    plotter.plot(history)

    # Зберегти контрольну точку моделі
    if iter % 100 == 0:
        torch.save(model.state_dict(), checkpoint_prefix)

# Зберегти останню натреновану модель
torch.save(model.state_dict(), checkpoint_prefix)
experiment.flush()
```

Тепер ми можемо використати нашу тренувану модель RNN для генерації "музики".

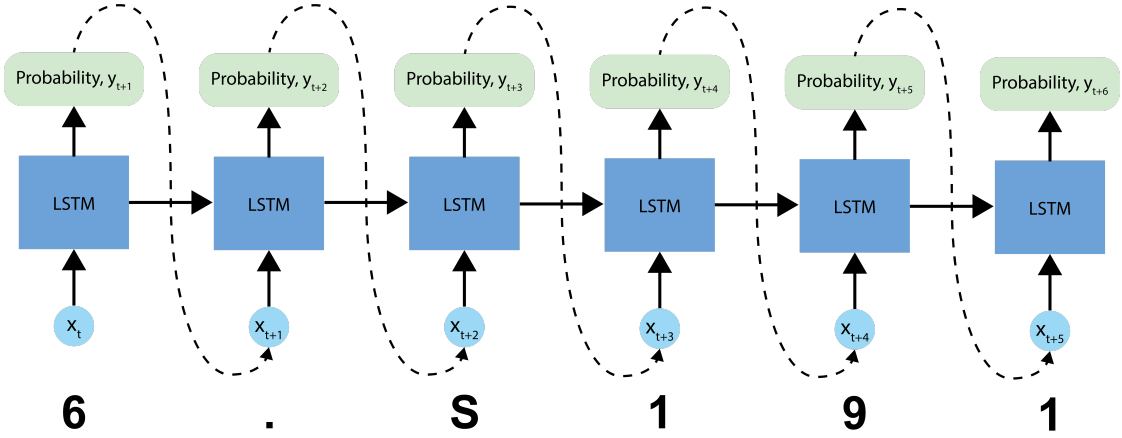
При генерації музики ми повинні подати моделі якийсь початковий рядок (тому що вона не може нічого передбачити без початкового рядка).

Щоразу, коли ми генеруємо нову пісню, ми починаємо з випадкового символу.

Щоразу, коли ми генеруємо новий символ, ми використовуємо нашу тренувану модель RNN для прогнозування наступного символу.

Ми продовжуємо цей процес доти, поки не отримаємо достатню кількість символів для генерації пісні.

Нарешті, ми записуємо пісню у файл та слухаємо її.



Процедура генерації включає такі кроки:

1. Ініціалізуємо початковий рядок та стан RNN, встановлюємо кількість символів для генерації.
2. Використовуємо початковий рядок та стан RNN для отримання розподілу ймовірностей наступного символу.
3. Вибираємо зразок з мультиноміального розподілу для обчислення індексу передбаченого символу. Цей передбачений символ потім використовується як наступний вхід моделі.
4. На кожному кроці оновлений стан RNN передається назад у модель, щоб вона мала більше контексту для наступного прогнозу.

Завершіть та експериментуйте з наступним блоком коду (а також деякими аспектами визначення мережі та тренування), і подивіться, як модель працює.

Як пісні, згенеровані після тренування з невеликою кількістю епох, порівнюються з піснями, згенерованими після більш тривалого тренування?

```
# Прогнозування генерованої пісні
def generate_text(model, start_string, generation_length=1000):
    # Конвертуємо початковий рядок у числа (векторизація)
    input_idx = [char2idx[s] for s in start_string]
    input_idx = torch.tensor([input_idx], dtype=torch.long).to(device)

    # Ініціалізуємо прихований стан
    state = model.init_hidden(input_idx.size(0), device)

    # Порожній рядок для зберігання результатів
    text_generated = []
    tqdm._instances.clear()

    for i in range(generation_length):
        # Обчислюємо входи та генеруємо прогнози наступних символів
        predictions, state = model(input_idx, state, return_state=True)

        # Видаляємо розмірність батчу
        predictions = predictions.squeeze(0)

        # Використовуємо мультиноміальний розподіл для вибірки з ймовірностей
        input_idx = torch.multinomial(torch.softmax(predictions, dim=-1), num_samples=1)

        # Додаємо передбачений символ до згенерованого тексту
        text_generated.append(idx2char[input_idx].item())

    return (start_string + ''.join(text_generated))
```

```
# Використайте модель та функцію, визначену вище, щоб згенерувати текст у форматі ABC довжиною
# Як ви можете помітити, файли ABC починаються з "X" - це може бути хороший початковий рядок.
generated_text = generate_text(
    model,
    start_string="X",
    generation_length="TODO: призначте довжину генерації"
)
```

4.5. Генерація та відтворення музики

Після навчання моделі ми можемо використати її для генерації нової музики. Потім ми конвертуємо згенерований текст у форматі ABC у звуковий файл для відтворення:

```
# Витягнення фрагментів пісень зі згенерованого тексту
generated_songs = mdl.lab1.extract_song_snippet(generated_text)

# Відтворення згенерованих пісень
for i, song in enumerate(generated_songs):
    # Синтез звукової хвилі з пісні
    waveform = mdl.lab1.play_song(song)
```

```
    # Якщо це валідна "пісня" (правильний синтаксис), відтворюємо її
    if waveform:
        print("Згенерована нейромережею послідовність ", i)
        ipythondisplay.display(waveform)
        numeric_data = np.frombuffer(waveform.data, dtype=np.int16)
        wav_file_path = f"output_{i}.wav"
        write(wav_file_path, 88200, numeric_data)

        # Зберегти вашу пісню у інтерфейсі Comet -- ви можете до нього отримати доступ
        experiment.log_asset(wav_file_path)
```

УМОВА ЗАВДАННЯ ДО ЛАБОРАТОРНОЇ РОБОТИ

1. Розібратись у процесі підготовки даних, побудови моделі RNN, її тренування та генерації музики.
2. Виконати завдання по ходу лабораторної роботи.
3. Поекспериментувати з параметрами процесу тренування та генерації.
4. Запустити процес тренування RNN та дочекатися його завершення (або виконати достатню кількість ітерацій для отримання робочої моделі).
5. Використати навчену модель для генерації щонайменше одного прикладу музичного твору у форматі ABC.
6. Проаналізувати результати, отримані на кожному етапі, та задокументувати їх у звіті.

ІНДІВІДУАЛЬНІ ВАРІАНТИ ЗАВДАННЯ

Індивідуальне завдання полягає в експериментуванні з параметрами процесу тренування та генерації:

Дослідження впливу гіперпараметрів:

```
*   Змініть розмір прихованого стану LSTM.
*   Змініть швидкість навчання оптимізатора.
*   Змініть довжину вхідної послідовності для тренування.
```

Проаналізуйте, як ці зміни впливають на швидкість навчання та суб'єктивну якість згенерованої музики.

ЗМІСТ ЗВІТУ

1. Тема та мета роботи.
2. Теоретичні відомості (стислий огляд PyTorch, RNN, процесу генерації).
3. Постановка завдання.
4. Хід виконання роботи:
    - Короткий опис кроків, виконаних у ноутбуці, з ключовими скріншотами (створення тензорів, градієнти, визначення моделі).
    - Детальний опис кроків, виконаних у ноутбуці:
        - Підготовка даних (розмір словника, приклад векторизації).
        - Скріншоти графіка функції втрат під час тренування.
        - Приклад(и) згенерованої музики у текстовому форматі ABC.

    - (Якщо виконувались) Опис та результати експериментів з індивідуального завдання.
5. Результати роботи та їх аналіз:
    - Аналіз процесу навчання моделі.
    - Оцінка якості згенерованої "музики" (суб'єктивна).
    - Оцінка зусиль, що були прикладені для досягнення результату з побудовою з нуля власної моделі (на відміну від розгортання тренованих моделей в минулих роботах).

КОНТРОЛЬНІ ПИТАННЯ

1. Що таке тензор у PyTorch і чим він відрізняється від масиву NumPy?
2. Що таке механізм автоматичного диференціювання (`autograd`) у PyTorch?
3. Які два основні способи визначення нейронних мереж у PyTorch? У чому їх переваги та недоліки?
4. Що таке рекурентна нейронна мережа (RNN)? У чому її відмінність від мереж прямого поширення?
5. Для яких типів завдань зазвичай використовуються RNN?
6. Що таке LSTM?
7. Поясніть процес генерації послідовності (наприклад, музики) за допомогою навченої RNN.

СПИСОК ЛІТЕРАТУРИ

1. MIT 6.S191: Introduction to Deep Learning - Lab 1: Intro to Deep Learning and Music Generation. https://github.com/MITDeepLearning/introtodeeplearning/tree/master/lab1
2. PyTorch Documentation. https://pytorch.org/docs/stable/index.html
3. Understanding LSTM Networks -- colah's blog. https://colah.github.io/posts/2015-08-Understanding-LSTMs/
4. The Unreasonable Effectiveness of Recurrent Neural Networks. http://karpathy.github.io/2015/05/21/rnn-effectiveness/
5. ABC Notation Home Page. https://abcnotation.com/