அத்தியாயம் 8: தமிழ் GPT: சங்கத்தமிழ் பாடல்களை உருவாக்கும் ஒரு AI கவிஞர்

இரண்டாயிரம் ஆண்டுகளுக்கு முன்பு, கபிலரும் பரணரும் செதுக்கிய சங்கத் தமிழின் உன்னதத்திற்கும், இன்று நாம் வாழும் டீப் லேர்னிங் (Deep Learning) தொழில்நுட்பத்தின் உச்சத்திற்கும் இடையே ஒரு பாலத்தை அமைக்க முடியுமா?

> *தர்க்கத்தையும், எண்களையும் மட்டுமே அறிந்த ஒரு இயந்திரத்திற்கு, அகம், புறம் என்ற தமிழின் அழகியலையும், தலைவனின் பிரிவுத்துயரையும், தலைவியின் உள்ளுறை உவமத்தையும் கற்றுக்கொடுக்க முடியுமா?*

இந்த அத்தியாயத்தில், நாம் அந்த மாபெரும் சவாலை ஏற்கப் போகிறோம்.

பைத்தான் மற்றும் டீப் லேர்னிங் என்ற நவீன உளியைக் கொண்டு, சங்க இலக்கியம் என்ற பழந்தமிழ்க் கல்லைச் செதுக்கி, ஒரு புதிய AI கவிஞனை உருவாக்கும் பயணத்தை நாம் மேற்கொள்வோம். இந்த டிஜிட்டல் புலவன், அந்தப் பெரும் புலவர்களின் படைப்புகளைப் படித்து, அவற்றின் ஓசையையும், நயத்தையும், வார்த்தை தேர்வையும் கற்றுக்கொண்டு, அதே பாணியில் புதிய கவிதைகளைப் படைக்கும் ஆற்றலைப் பெறுவான். வாருங்கள், இந்த நவீன காலத் தொழில்நுட்பத்தின் மூலம், நமது செம்மொழியின் கடந்த காலத்திற்கு ஒரு புதிய உயிரைக் கொடுப்போம்.

நாம் இன்று டீப் லேர்னிங் (Deep Learning) தொழில்நுட்பத்தைப் பயன்படுத்தி ஒரு AI கவிஞரை எப்படி உருவாக்கலாம் என்று விரிவாகக் காண்போம். இந்த AI கவிஞர், சங்கத்தமிழ் பாணியில் புதிய பாடல்களை இயற்றக்கூடிய திறன் கொண்டவர்! பைத்தான் (Python) மொழியில் உருவாக்கப்பட்ட இந்த டீப் லேர்னிங் மாடல், சங்கத்தமிழ் இலக்கியத்திலிருந்து கற்றுக்கொண்டு புதிய பாடல்களை உருவாக்கும்.

தேவையானவை PyTorch இது டீப் லேர்னிங் மாடல்களை உருவாக்க உதவும் ஒரு பைத்தான் நூலகம் (library). சங்கத்தமிழ் தரவு - Project Madurai போன்ற இணையதளங்களில் இருந்து சங்கத்தமிழ் பாடல்களைப் பதிவிறக்கம் செய்து கொள்ளலாம். இதனை `input.txt` என்ற கோப்பில் சேமித்து வைக்கவும்.

கம்ப்யூட்டரால் நேரடியாக டெக்ஸ்ட்டைப் புரிந்து கொள்ள முடியாது. டெக்ஸ்ட்டை எண்களாக மாற்றும் "encoding" செயல்முறை அவசியம். சங்கத்தமிழ் பாடல்களை கம்ப்யூட்டருக்குப் புரியும் வகையில் மாற்ற, முதலில் பாடல்களில் உள்ள தனித்துவமான எழுத்துக்களை அடையாளம் கண்டு சொல்லகராதி உருவாக்க வேண்டும். இந்த சொல்லகராதி, ஒவ்வொரு எழுத்துக்கும் ஒரு தனித்துவமான எண்ணை ஒதுக்கும். பின்னர், இந்த எண்களைப் பயன்படுத்தி பாடல்களை

கம்ப்யூட்டர் புரிந்துகொள்ளும் வகையில் மாற்றலாம். உதாரணமாக, "ம" என்ற எழுத்துக்கு 57, "லை" என்ற எழுத்துக்கு 61, 73 எண்களை ஒதுக்கலாம். இதன் மூலம், **"மலை" என்ற வார்த்தையை 57, 61, 73** என்ற எண்களாக மாற்றலாம். இந்த எண்களை கம்ப்யூட்டர் பல்வேறு வழிகளில் பயன்படுத்தலாம். உதாரணமாக, மொழிபெயர்ப்பு, உரை பகுப்பாய்வு, உணர்வு பகுப்பாய்வு போன்ற பணிகளுக்கு இந்த எண்கள் பயன்படும்.

ஏன் தனித்துவமான எழுத்துக்கள்?

கம்ப்யூட்டருக்குத் தமிழ் எழுத்துக்களைப் பற்றி எதுவும் தெரியாது. நாம் ஒவ்வொரு எழுத்துக்கும் ஒரு தனித்துவமான எண்ணைக் கொடுக்க வேண்டும். நமது மாடல் பயன்படுத்தும் மொத்த எழுத்துக்களின் தொகுப்பு தான் சொல்லகராதி. இந்த சொல்லகராதியில் உள்ள ஒவ்வொரு எழுத்துக்கும் ஒரு தனித்துவமான எண் குறியீடு இருக்கும்.

எப்படி தனித்துவமான எழுத்துக்களைக் கண்டுபிடிப்பது?

1. **பாடல்களைப் படித்தல்:** `input.txt` கோப்பில் உள்ள சங்கத்தமிழ் பாடல்களைப் படிக்கவும்.
2. **எழுத்துக்களைச் சேகரித்தல்:** பாடல்களில் உள்ள ஒவ்வொரு எழுத்தையும் ஒரு "set"-ல் சேர்க்கவும். "set"-ல் ஒரே எழுத்து பலமுறை வந்தாலும், ஒரே முறை தான் சேமிக்கப்படும்.
3. **வரிசைப்படுத்துதல்:** "set"-ல் உள்ள எழுத்துக்களை அகர வரிசையில் வரிசைப்படுத்தவும்.

உதாரணம்:

`input.txt` கோப்பில் "அகர முதல" என்று இருந்தால்,

- படிக்கப்பட்ட எழுத்துக்கள்: அ, க, ர, மு, த, ல
- "set"-ல் சேர்க்கப்பட்ட எழுத்துக்கள்: {அ, க, ர, மு, த, ல}
- வரிசைப்படுத்தப்பட்ட எழுத்துக்கள்: [அ, க, ங, ச, ட, த, ந, ப, ம, ய, ர, ல, வ, ழ, ள, ற, ன] (இது ஒரு உதாரணம் மட்டுமே, உங்கள் கோப்பில் உள்ள எழுத்துக்கள் வேறுபடலாம்)

```python
chars = sorted(list(set(text)))
vocab_size = len(chars)
```

- `text`: `input.txt` கோப்பில் உள்ள பாடல்களின் டெக்ஸ்ட்.

- `set(text)`: டெக்ஸ்டில் உள்ள தனித்துவமான எழுத்துக்களைக் கொண்ட ஒரு "set"-ஐ உருவாக்குகிறது.
- `list(set(text))`: "set"-ஐ ஒரு "list"-ஆக மாற்றுகிறது.
- `sorted(list(set(text)))`: "list"-ஐ அகர வரிசையில் வரிசைப்படுத்துகிறது.
- `chars`: வரிசைப்படுத்தப்பட்ட தனித்துவமான எழுத்துக்களின் "list".
- `vocab_size`: தனித்துவமான எழுத்துக்களின் எண்ணிக்கை (சொல்லகராதியின் அளவு).

இந்த தனித்துவமான எழுத்துக்களைக் கண்டுபிடிக்கும் செயல்முறை, நமது மொழி மாதிரியின் முதல் படியாகும். இந்த எழுத்துக்கள் தான் மாடல் புரிந்து கொள்ளும் அடிப்படை கூறுகள். ஒவ்வொரு எழுத்துக்கும் ஒரு தனித்துவமான எண்ணை (integer) கொடுக்க வேண்டும். இதை ஒரு அகராதியாக (dictionary) சேமித்து வைப்போம்.

```
stoi = {' ': 2,'அ': 31,'க': 43,'ர': 59,'மு': 57,'த': 69,'ல':
52,'எ': 61,'ழு': 37,'த்': 63, 'ெ': 77,'ல்': 71, 'ா': 66}
```

இந்த அகராதியைப் பயன்படுத்தி, "அகர முதல" என்ற டெக்ஸ்ட்டை எண்களாக மாற்றும்போது, நமக்கு `[31, 43, 59, 2, 57, 69, 52, 61]` என்ற எண்களின் பட்டியல் கிடைக்கும். முதல் எழுத்து "அ". `stoi` அகராதியில், "அ" என்ற எழுத்துக்கு 31 என்ற எண் ஒதுக்கப்பட்டுள்ளது. எனவே, முதல் எண் 31. இரண்டாவது எழுத்து "க". `stoi` அகராதியில், "க" என்ற எழுத்துக்கு 43 என்ற எண் ஒதுக்கப்பட்டுள்ளது. எனவே, இரண்டாவது எண் 43. இதேபோல், மற்ற எழுத்துக்களுக்கும் `stoi` அகராதியில் உள்ள எண்களை எடுத்துக்கொள்வோம். இறுதியில், "அகர முதல" என்ற டெக்ஸ்ட் `[31, 43, 59, 2, 57, 69, 52, 61]` என்ற எண்களின் பட்டியலாக மாறும்.

குறிப்பு:
உங்கள் `stoi` அகராதியில் உள்ள எண்கள், நீங்கள் பயன்படுத்தும் டேட்டா மற்றும் encoding முறையைப் பொறுத்து மாறுபடலாம்.

Decoding

"Decoding" என்பது "encoding"-க்கு எதிர்மாறான செயல்முறை. அதாவது, எண்களின் பட்டியலை எடுத்துக்கொண்டு, அதை டெக்ஸ்ட்டாக மாற்றுவது. "Encoding"-ல் பயன்படுத்திய அகராதியைத் தலைகீழாக மாற்ற வேண்டும். அதாவது, எண்கள் "key"-ஆகவும், எழுத்துக்களை "value"-ஆகவும் கொண்ட ஒரு அகராதி:

```
itos = {2: ' ', 31: 'அ', 43: 'க', 59: 'ர', 57: 'மு', 69: 'த', 52:
'ல', 61: 'எ', 37: 'ழு', 63: 'த்', 77: 'ெ', 71: 'ல்', 66: 'ா'}
```

எழுத்துக்களாக மாற்றுதல் (Conversion to Characters)

இப்போது, நமக்குக் கொடுக்கப்பட்ட எண்களின் பட்டியலை எடுத்துக்கொள்வோம். பட்டியலில் உள்ள ஒவ்வொரு எண்ணையும் எடுத்துக்கொண்டு, அகராதியில் அதற்குரிய எழுத்தைப் பார்ப்போம். இந்த எழுத்துக்களை ஒன்றாக இணைத்தால், நமக்கு டெக்ஸ்ட் கிடைக்கும்.

உதாரணம்:

`[31, 43, 59, 2, 57, 69, 52, 61]` என்ற எண்களின் பட்டியலை எடுத்துக்கொள்வோம்.

- முதல் எண் 31. அகராதியில் 31 என்ற எண்ணுக்கு "அ" என்ற எழுத்து ஒதுக்கப்பட்டுள்ளது.
- இரண்டாவது எண் 43. அகராதியில் 43 என்ற எண்ணுக்கு "க" என்ற எழுத்து ஒதுக்கப்பட்டிருக்கலாம்.
- இதேபோல், மற்ற எண்களுக்கும் அகராதியில் உள்ள எழுத்துக்களை எடுத்துக்கொள்வோம்.

இறுதியில், `[31, 43, 59, 2, 57, 69, 52, 61]` என்ற எண்களின் பட்டியல் "அகர முதல" என்ற டெக்ஸ்ட்டாக மாறும்.

சுருக்கமாக

- Encoding: டெக்ஸ்ட்டை எண்களாக மாற்றுதல்.
- Decoding: எண்களை டெக்ஸ்ட்டாக மாற்றுதல்.

இந்த இரண்டு செயல்முறைகளும், டெக்ஸ்ட் டேட்டாவை கம்ப்யூட்டர் புரிந்து கொள்ளும் வகையில் மாற்ற உதவுகின்றன. இதன் மூலம், டெக்ஸ்ட் டேட்டாவைப் பயன்படுத்தி டீப் லேர்னிங் மாடல்களை உருவாக்க முடியும்.

இப்பொழுது நாம் கற்றவற்றை பைதானில் செய்து பார்ப்போம்

8.1. Libraries & Hyperparameters

```
import torch
```

```
import torch.nn as nn
from torch.nn import functional as F

# Hyperparameters
batch_size = 16     # எத்தனை பாடல்களை ஒரே நேரத்தில் process செய்ய வேண்டும்
block_size = 32    # எத்தனை எழுத்துக்களை மாதிரி கணக்கில் எடுத்துக்கொள்ள வேண்டும்
max_iters = 5000   # எத்தனை முறை train செய்ய வேண்டும்
eval_interval = 100  # எத்தனை iterations-க்கு ஒரு முறை loss-ஐ calculate செய்ய வேண்டும்
learning_rate = 1e-3 # மாதிரி கற்கும் வேகம்
device = 'cuda' if torch.cuda.is_available() else 'cpu' # GPU இருந்தால் அதை பயன்படுத்தும்
eval_iters = 200
n_embd = 64
n_head = 4
n_layer = 4
dropout = 0.0

torch.manual_seed(1337)
```

விளக்கம்:

Libraries:

- `import torch`: PyTorch-ன் core functionalities-ஐ import செய்கிறது. Tensor operations, dynamic computation graphs போன்ற வசதிகளை பயன்படுத்துவதற்கு இது அவசியம்.
- `import torch.nn as nn`: Neural network modules-ஐ உருவாக்க உதவுகிறது. இங்குதான் நாம் நமது மாடலின் layers-களை define செய்வோம்.
- `from torch.nn import functional as F`: Activation functions (எ.கா: ReLU, softmax), loss functions (எ.கா: cross_entropy) போன்ற utils-களை import செய்கிறது.

Hyperparameters: இவை நமது டீப் லேர்னிங் மாடலின் training process-ஐ control செய்யும் மாறிகள் (variables). இவற்றின் சரியான தேர்வு, மாடல் எவ்வளவு நன்றாக கற்றுக்கொள்கிறது என்பதைப் பொறுத்தது.

- `batch_size = 16`: ஒரு முறை training செய்யும் போது எத்தனை பாடல்களை (sequences) மாடலுக்குக் கொடுக்க வேண்டும் என்பது. பெரிய `batch_size` வேகமாக training செய்ய உதவும், ஆனால் அதிக memory தேவைப்படும்.
- `block_size = 32`: ஒரு பாடலில் (sequence) எத்தனை எழுத்துக்களை ஒரு நேரத்தில் process செய்ய வேண்டும் என்பது. இது context window size போன்றது. அதாவது, மாடல் ஒரு எழுத்தைக் கணிக்க, முந்தைய `block_size` எழுத்துக்களைப் பார்க்கும்.
- `max_iters = 5000`: மொத்த training iterations எண்ணிக்கை. அதிக iterations-ல் மாடல் நன்றாக கற்றுக்கொள்ள வாய்ப்புள்ளது, ஆனால் overfitting ஆகவும் வாய்ப்புள்ளது.
- `eval_interval = 100`: Training progress-ஐ monitor செய்ய, எத்தனை iterations-க்கு ஒரு முறை validation loss-ஐ கணக்கிட வேண்டும் என்பது.
- `learning_rate = 1e-3`: மாடல் weights-ஐ எவ்வளவு வேகமாக adjust செய்ய வேண்டும் என்பது. சிறிய `learning_rate` மெதுவாக கற்கும், ஆனால் stable-ஆக கற்கும். பெரிய `learning_rate` வேகமாக கற்கும், ஆனால் unstable-ஆக இருக்கலாம்.
- `device = 'cuda' if torch.cuda.is_available() else 'cpu'`: GPU இருந்தால் training-ஐ GPU-வில் செய்யும். GPU இருந்தால் training வேகமாக நடக்கும். GPU இல்லை என்றால் CPU-வில் நடக்கும்.
- `eval_iters = 200`: Evaluation-க்காக எத்தனை iterations செய்ய வேண்டும் என்பது.
- `n_embd = 64`: Embedding dimension. ஒவ்வொரு எழுத்தையும் ஒரு vector-ஆக மாற்றும்போது, அந்த vector-ன் அளவு.
- `n_head = 4`: Transformer-ல் எத்தனை attention heads இருக்க வேண்டும் என்பது.
- `n_layer = 4`: Transformer-ல் எத்தனை layers இருக்க வேண்டும் என்பது.
- `dropout = 0.0`: Dropout rate. Overfitting-ஐ குறைக்க உதவுகிறது.

- `torch.manual_seed(1337)`: Random seed-ஐ set செய்கிறோம். இதனால், ஒவ்வொரு முறை program-ஐ run செய்தாலும், ஒரே மாதிரியான output கிடைக்கும். இது debugging மற்றும் experiments-ஐ reproduce செய்ய உதவுகிறது.

இந்த hyperparameters-ஐ மாற்றி, model-ன் performance-ஐ மேம்படுத்தலாம். இவை machine learning-ல் முக்கியமான tuning parameters. இப்பொழுது, ஏன் இந்த values-களை தேர்ந்தெடுத்தோம் என்பது உங்களுக்கு புரிந்திருக்கும் என்று நினைக்கிறேன். அடுத்த பாகத்தில், data loading மற்றும் preprocessing பற்றி விரிவாகக் காண்போம்.

8.2. Data Loading & Preprocessing

```
# டேட்டாவை படித்தல்
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()

# தனித்துவமான எழுத்துக்களை கண்டுபிடித்தல்
chars = sorted(list(set(text)))
vocab_size = len(chars)

# எழுத்துக்களை எண்களாக மாற்றுதல் (Encoding)
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s]
decode = lambda l: ''.join([itos[i] for i in l])

# டேட்டாவை train மற்றும் validation என பிரித்தல்
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9*len(data))
train_data = data[:n]
val_data = data[n:]
```

விளக்கம்:

டேட்டாவைப் படித்தல் (`input.txt` கோப்பிலிருந்து):

Python

```
with open('input.txt', 'r', encoding='utf-8') as f:
    text = f.read()
```

- `with open(...)`: `input.txt` கோப்பை படிக்க திறக்கிறது. `encoding='utf-8'` என்பது தமிழ் எழுத்துக்களைச் சரியாகப் படிக்க உதவுகிறது. UTF-8 encoding பெரும்பாலான தமிழ் எழுத்துக்களை உள்ளடக்கியது.
- `f.read()`: கோப்பிலுள்ள அனைத்து உள்ளடக்கத்தையும் ஒரு string-ஆக `text` மாறியில் சேமிக்கிறது.

தனித்துவமான எழுத்துக்களைக் கண்டுபிடித்தல்:

Python

```
chars = sorted(list(set(text)))
vocab_size = len(chars)
```

- `set(text)`: `text` string-ல் உள்ள **தனித்துவமான** எழுத்துக்களைக் கொண்ட ஒரு set-ஐ உருவாக்குகிறது. ஒரு set-ல், எந்த ஒரு உறுப்பும் இரண்டு முறை வராது. இதனால், எல்லா தனித்துவமான எழுத்துக்களும் கிடைக்கும்.
- `list(set(text))`: set-ஐ ஒரு list-ஆக மாற்றுகிறது. set-ல் உள்ள உறுப்புகளுக்கு ஒரு குறிப்பிட்ட வரிசை இல்லை. list-ஆக மாற்றும்போது, உறுப்புகளுக்கு ஒரு வரிசை கிடைக்கிறது.
- `sorted(...)`: list-ஐ அகர வரிசையில் வரிசைப்படுத்துகிறது. வரிசைப்படுத்துவதால், எழுத்துக்களுக்கு ஒரு நிலையான குறியீடு (index) கொடுக்க முடியும்.
- `chars`: வரிசைப்படுத்தப்பட்ட தனித்துவமான எழுத்துக்களின் list.
- `vocab_size`: தனித்துவமான எழுத்துக்களின் எண்ணிக்கை (நமது சொல்லகராதியின் அளவு).

எழுத்துக்களை எண்களாக மாற்றுதல் (Encoding):

Python

```
stoi = { ch:i for i,ch in enumerate(chars) }
itos = { i:ch for i,ch in enumerate(chars) }
encode = lambda s: [stoi[c] for c in s]
decode = lambda l: ''.join([itos[i] for i in l])
```

- `stoi`: string to integer என்ற dictionary. ஒவ்வொரு எழுத்துக்கும் ஒரு unique integer mapping-ஐ உருவாக்குகிறது. இந்த mapping-ஐப் பயன்படுத்தி, டெக்ஸ்டை எண்களாக மாற்றலாம்.
- `itos`: integer to string என்ற dictionary. `stoi`-க்கு நேர் எதிரானது. எண்களை எழுத்துக்களாக மாற்ற உதவுகிறது.
- `encode(s)`: ஒரு string `s`-ஐ எடுத்து, `stoi` dictionary-ஐப் பயன்படுத்தி, அதில் உள்ள ஒவ்வொரு எழுத்தையும் அதற்குரிய எண்ணாக மாற்றி, ஒரு list-ஐ (எண்களின் list) திரும்ப அளிக்கிறது.
- `decode(l)`: ஒரு list `l` (எண்களின் list) -ஐ எடுத்து, `itos` dictionary-ஐப் பயன்படுத்தி, அதில் உள்ள ஒவ்வொரு எண்ணையும் அதற்குரிய எழுத்தாக மாற்றி, string-ஐ திரும்ப அளிக்கிறது.

டேட்டாவை train மற்றும் validation என பிரித்தல்:

Python

```
data = torch.tensor(encode(text), dtype=torch.long)
n = int(0.9*len(data))
train_data = data[:n]
val_data = data[n:]
```

- `encode(text)`: முழு `text`-ஐ `encode` function-ஐப் பயன்படுத்தி எண்களாக மாற்றுகிறது.
- `torch.tensor(...)`: `encode(text)`-ன் வெளியீட்டை PyTorch tensor-ஆக மாற்றுகிறது. `dtype=torch.long` என்பது integers-ஐக் குறிக்கிறது. Tensorகள் தான் PyTorch-ல் data-வை manipulate செய்ய பயன்படுத்தப்படும் datastructure.
- `n = int(0.9*len(data))`: மொத்த data-வில் 90% train data-க்காகவும், 10% validation data-க்காகவும் பிரிக்கும் வகையில் `n` மதிப்பை கணக்கிடுகிறது.
- `train_data`: முதல் `n` எழுத்துக்கள் train data-வாக சேமிக்கப்படுகிறது.
- `val_data`: மீதமுள்ள எழுத்துக்கள் validation data-வாக சேமிக்கப்படுகிறது. Validation data, model-ன் performance-ஐ train செய்யும் போதே பார்க்க உதவும்.

இந்த நான்கு படிகளின் மூலம், சங்கத் தமிழ் பாடல்களைக் கொண்ட தரவை டீப் லேர்னிங் மாடலுக்குத் தேவையான வடிவத்தில் மாற்றுகிறோம். இனி, மாடலை உருவாக்குதல் மற்றும் train செய்தல் பற்றி பார்ப்போம்.

8.3. Batching

நாம் தயாரித்த தரவை டீப் லேர்னிங் மாடலுக்குப் பயிற்சியளிக்க ஏற்ற batches ஆக எப்படிப் பிரிக்கலாம் என்று விரிவாகக் காண்போம்.

```
# டேட்டாவை batches ஆக பிரித்தல்
def get_batch(split):
    data = train_data if split == 'train' else val_data
    ix = torch.randint(len(data) - block_size, (batch_size,))
    x = torch.stack([data[i:i+block_size] for i in ix])
    y = torch.stack([data[i+1:i+block_size+1] for i in ix])
    x, y = x.to(device), y.to(device)
    return x, y
```

விளக்கம்:

`get_batch(split)` என்ற function, `train` அல்லது `validation` data-வில் இருந்து ஒரு batch டேட்டாவை எடுத்துத் தருகிறது. ஒவ்வொரு batch-லும் `batch_size` எண்ணிக்கையிலான பாடல்கள் இருக்கும்.

டேட்டா தேர்வு:

Python

```
data = train_data if split == 'train' else val_data
```

`split` என்ற மாறியின் மதிப்பை பொறுத்து, `train_data` அல்லது `val_data` தேர்ந்தெடுக்கப்படுகிறது. `split` 'train' ஆக இருந்தால் `train_data`வும், இல்லையென்றால் `val_data`வும் பயன்படுத்தப்படும்.

சீரற்ற indices உருவாக்குதல்:

Python

```
ix = torch.randint(len(data) - block_size, (batch_size,))
```

`torch.randint` function-ஐப் பயன்படுத்தி, 0 முதல் `len(data) - block_size` வரை இடைப்பட்ட மதிப்புகளில் `batch_size` எண்ணிக்கையிலான சீரற்ற எண்கள் உருவாக்கப்படுகின்றன. இந்த எண்கள், data-வில் batch-ன் தொடக்க நிலையைக் குறிக்கும்.

Input (x) மற்றும் Target (y) உருவாக்குதல்:

Python

```
x = torch.stack([data[i:i+block_size] for i in ix])
y = torch.stack([data[i+1:i+block_size+1] for i in ix])
```

- `data[i:i+block_size]`: `data`-வில் `i` முதல் `i+block_size` வரையிலான உள்ளடக்கத்தை (ஒரு பாடலின் ஒரு பகுதி) எடுக்கிறது. `ix`-ல் உள்ள ஒவ்வொரு `i`-க்கும் இப்படியான ஒரு பகுதி எடுக்கப்பட்டு, `x` என்ற list-ல் சேர்க்கப்படும். `x` என்பது input sequence.
- `data[i+1:i+block_size+1]`: `data`-வில் `i+1` முதல் `i+block_size+1` வரையிலான உள்ளடக்கத்தை (அடுத்த எழுத்துக்களைக் கொண்ட பகுதி) எடுக்கிறது. இது target sequence ஆகும். `y` என்பது x-க்கான அடுத்த எழுத்துக்களைக் கொண்டது.
- `torch.stack(...)`: `x` மற்றும் `y`-ல் உள்ள tensorகளை ஒன்றாக stack செய்து ஒரு batch tensor-ஆக உருவாக்குகிறது.

Device-க்கு மாற்றுதல் (GPU அல்லது CPU):

Python

```
x, y = x.to(device), y.to(device)
```

`x` மற்றும் `y` tensorகளை device-க்கு (GPU அல்லது CPU) நகர்த்துகிறது. GPU இருந்தால் GPU-விலும், இல்லையென்றால் CPU-விலும் training நடக்கும். GPU-வில் training வேகமாக இருக்கும்.

Batch-ஐ திரும்ப அளித்தல்:

Python

```
return x, y
```

உருவாக்கப்பட்ட batch tensorகள் `x` (input) மற்றும் `y` (target) திரும்ப அளிக்கப்படுகின்றன.

இந்த function-ஐப் பயன்படுத்தி, training loop-ல் batch-களை உருவாக்கி, மாடலுக்குப் பயிற்சி அளிக்கலாம். அடுத்த பகுதியில், மாடல் உருவாக்கம் பற்றி விரிவாகப் பார்ப்போம்.

8.4. Loss Estimation

Loss என்பது, மாடலின் predictions எவ்வளவு தவறாக இருக்கிறது என்பதன் அளவீடு. Loss குறைவாக இருந்தால், மாடல் நன்றாகக் கற்றுக்கொள்கிறது என்று அர்த்தம்.

```
# Loss-ஐ கணக்கிடுதல்
@torch.no_grad()
def estimate_loss():
    out = {}
    model.eval()
    for split in ['train', 'val']:
        losses = torch.zeros(eval_iters)
        for k in range(eval_iters):
            X, Y = get_batch(split)
            logits, loss = model(X, Y)
            losses[k] = loss.item()
        out[split] = losses.mean()
    model.train()
    return out
```

விளக்கம்:

`estimate_loss()` என்ற function, train மற்றும் validation data-வில் மாடலின் performance-ஐ அளவிட loss-ஐ கணக்கிடுகிறது.

`@torch.no_grad()`:

இந்த decorator, gradient calculations-ஐ disable செய்கிறது. Evaluation-ன் போது gradients தேவையில்லை. gradients கணக்கிடாமல் இருந்தால், computation speed அதிகரிக்கும்.

`model.eval()`:

மாடலை evaluation mode-க்கு மாற்றுகிறது. சில layers (Dropout, BatchNorm) training மற்றும் evaluation mode-களில் வெவ்வேறாக செயல்படும்.

Loss சேமிக்க dictionary:

Python

```python
out = {}
```

`train` மற்றும் `validation` data-வின் loss-களை சேமிக்க ஒரு dictionary உருவாக்கப்படுகிறது.

Train மற்றும் Validation data-வில் loss கணக்கிடுதல்:

Python

```python
for split in ['train', 'val']:
    losses = torch.zeros(eval_iters)
    for k in range(eval_iters):
        X, Y = get_batch(split)
        logits, loss = model(X, Y)
        losses[k] = loss.item()
    out[split] = losses.mean()
```

- `for split in ['train', 'val']`: train மற்றும் validation data இரண்டிலும் loss கணக்கிடப்படுகிறது.
- `losses = torch.zeros(eval_iters)`: ஒவ்வொரு iteration-க்கான loss-ஐ சேமிக்க ஒரு tensor உருவாக்கப்படுகிறது.

- `for k in range(eval_iters)`: `eval_iters` எண்ணிக்கையிலான batches-களில் loss கணக்கிடப்படுகிறது.
- `X, Y = get_batch(split)`: ஒரு batch data (`X` - input, `Y` - target) பெறப்படுகிறது.
- `logits, loss = model(X, Y)`: மாடலின் output (`logits`) மற்றும் loss கணக்கிடப்படுகிறது.
- `losses[k] = loss.item()`: கணக்கிடப்பட்ட loss, `losses` tensor-ல் சேமிக்கப்படுகிறது. `.item()` loss-ன் scalar value-வை எடுக்கும்.
- `out[split] = losses.mean()`: அனைத்து iterations-களின் loss-களின் சராசரி கணக்கிடப்பட்டு, `out` dictionary-ல் சேமிக்கப்படுகிறது.

1. **`model.train()`:**

   மாடலை மீண்டும் training mode-க்கு மாற்றுகிறது.

2. **Loss-ஐ திரும்ப அளித்தல்:**

   Python

   ```python
   return out
   ```

   `train` மற்றும் `validation` data-வின் சராசரி loss-கள் dictionary-ஆக திரும்ப அளிக்கப்படுகின்றன.

உதாரணம்:

நினைத்துப்பாருங்கள், உங்களிடம் ஒரு கவிதை எழுதும் மாடல் இருக்கிறது. "அகர முதல" என்று ஆரம்பித்தால், அது "எழுத்தெல்லாம்" என்று கணிக்க வேண்டும்.

- மாடல் "அகர முதல" என்று input-ஆகப் பெற்று, "கவிதை" என்று output-ஆக கொடுத்தால், அது தவறான கணிப்பு. இந்தத் தவறை அளவிட ஒரு "loss function" பயன்படுகிறது.
- `estimate_loss()` function, பல samples எடுத்து, loss-ஐ கணக்கிட்டு, சராசரி loss-ஐத் தரும். இந்த சராசரி loss, மாடல் எவ்வளவு நன்றாகக் கற்றுக்கொண்டிருக்கிறது என்பதைக் காட்டும். Loss குறைவாக இருந்தால், மாடல் நன்றாகக் கற்றுக்கொண்டிருக்கிறது என்று பொருள்.

இந்த function-ஐப் பயன்படுத்தி, training-ன் போது model-ன் performance-ஐ monitor செய்யலாம். Loss அதிகமாக இருந்தால், model-ஐ மேலும் train செய்ய வேண்டும் அல்லது hyperparameters-ஐ மாற்ற வேண்டும். Loss குறைவாக இருந்தால், model நன்றாகக் கற்றுக்கொண்டிருக்கிறது என்று அர்த்தம்.

8.5. Model Definition

```
class Head(nn.Module):
    """ one head of self-attention """

    def __init__(self, head_size):
        super().__init__()
        self.key = nn.Linear(n_embd, head_size, bias=False)
        self.query = nn.Linear(n_embd, head_size, bias=False)
        self.value = nn.Linear(n_embd, head_size, bias=False)
        self.register_buffer('tril',
torch.tril(torch.ones(block_size, block_size)))

        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        B,T,C = x.shape
        k = self.key(x)   # (B,T,C)
        q = self.query(x) # (B,T,C)
        # compute attention scores ("affinities")
        wei = q @ k.transpose(-2,-1) * C**-0.5 # (B, T, C) @ (B, C,
T) -> (B, T, T)
        wei = wei.masked_fill(self.tril[:T, :T] == 0, float('-
inf')) # (B, T, T)
        wei = F.softmax(wei, dim=-1) # (B, T, T)
        wei = self.dropout(wei)
        # perform the weighted aggregation of the values
        v = self.value(x) # (B,T,C)
        out = wei @ v # (B, T, T) @ (B, T, C) -> (B, T, C)
        return out

class MultiHeadAttention(nn.Module):
    """ multiple heads of self-attention in parallel """
```

```python
    def __init__(self, num_heads, head_size):
        super().__init__()
        self.heads = nn.ModuleList([Head(head_size) for _ in range(num_heads)])
        self.proj = nn.Linear(n_embd, n_embd)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x):
        out = torch.cat([h(x) for h in self.heads], dim=-1)
        out = self.dropout(self.proj(out))
        return out

class FeedFoward(nn.Module):
    """ a simple linear layer followed by a non-linearity """

    def __init__(self, n_embd):
        super().__init__()
        self.net = nn.Sequential(
            nn.Linear(n_embd, 4 * n_embd),
            nn.ReLU(),
            nn.Linear(4 * n_embd, n_embd),
            nn.Dropout(dropout),
        )

    def forward(self, x):
        return self.net(x)

class Block(nn.Module):
    """ Transformer block: communication followed by computation """

    def __init__(self, n_embd, n_head):
        # n_embd: embedding dimension, n_head: the number of heads we'd like
        super().__init__()
        head_size = n_embd // n_head
        self.sa = MultiHeadAttention(n_head, head_size)
        self.ffwd = FeedFoward(n_embd)
        self.ln1 = nn.LayerNorm(n_embd)
```

```
        self.ln2 = nn.LayerNorm(n_embd)

    def forward(self, x):
        x = x + self.sa(self.ln1(x))
        x = x + self.ffwd(self.ln2(x))
        return x

# super simple bigram model
class BigramLanguageModel(nn.Module):

    def __init__(self):
        super().__init__()
        # each token directly reads off the logits for the next token from a lookup table
        self.token_embedding_table = nn.Embedding(vocab_size, n_embd)
        self.position_embedding_table = nn.Embedding(block_size, n_embd)
        self.blocks = nn.Sequential(*[Block(n_embd, n_head=n_head) for _ in range(n_layer)])
        self.ln_f = nn.LayerNorm(n_embd) # final layer norm
        self.lm_head = nn.Linear(n_embd, vocab_size)

    def forward(self, idx, targets=None):
        B, T = idx.shape

        # idx and targets are both (B,T) tensor of integers
        tok_emb = self.token_embedding_table(idx) # (B,T,C)
        pos_emb = self.position_embedding_table(torch.arange(T, device=device)) # (T,C)
        x = tok_emb + pos_emb # (B,T,C)
        x = self.blocks(x) # (B,T,C)
        x = self.ln_f(x) # (B,T,C)
        logits = self.lm_head(x) # (B,T,vocab_size)

        if targets is None:
            loss = None
        else:
            B, T, C = logits.shape
```

```
            logits = logits.view(B*T, C)
            targets = targets.view(B*T)
            loss = F.cross_entropy(logits, targets)

        return logits, loss

    def generate(self, idx, max_new_tokens):
        # idx is (B, T) array of indices in the current context
        for _ in range(max_new_tokens):
            # crop idx to the last block_size tokens
            idx_cond = idx[:, -block_size:]
            # get the predictions
            logits, loss = self(idx_cond)
            # focus only on the last time step
            logits = logits[:, -1, :] # becomes (B, C)
            # apply softmax to get probabilities
            probs = F.softmax(logits, dim=-1) # (B, C)
            # sample from the distribution
            idx_next = torch.multinomial(probs, num_samples=1) #
(B, 1)
            # append sampled index to the running sequence
            idx = torch.cat((idx, idx_next), dim=1) # (B, T+1)
        return idx

model = BigramLanguageModel()
m = model.to(device)
# print the number of parameters in the model
print(sum(p.numel() for p in m.parameters())/1e6, 'M parameters')
```

விளக்கம்:

1. `Head` (ஒரு Attention Head)

- **Self-Attention:** இது ஒரு sequence-ல் உள்ள ஒவ்வொரு element-ம் மற்ற element-களுடன் எவ்வாறு தொடர்பு கொள்கிறது என்பதை கற்றுக்கொள்ளும் ஒரு mechanism. இங்கு, ஒவ்வொரு எழுத்தும் (element) மற்ற எழுத்துக்களுடன் எவ்வளவு தொடர்புடையது என்பதைக் கணக்கிடுகிறோம்.

- **Key, Query, Value:** ஒவ்வொரு எழுத்தும், மூன்று வெவ்வேறு linear transformations மூலம், மூன்று vectors-ஆக மாற்றப்படுகிறது: key, query, value.
    - **Key (k):** ஒரு எழுத்தின் "key" vector, அந்த எழுத்து மற்ற எழுத்துக்களால் எவ்வாறு "தேடப்படுகிறது" என்பதைக் குறிக்கிறது.
    - **Query (q):** ஒரு எழுத்தின் "query" vector, அந்த எழுத்து மற்ற எழுத்துக்களை எவ்வாறு "தேடுகிறது" என்பதைக் குறிக்கிறது.
    - **Value (v):** ஒரு எழுத்தின் "value" vector, அந்த எழுத்தின் "content" அல்லது "information" ஐக் குறிக்கிறது.
- **Attention Scores:** ஒரு எழுத்தின் query vector, மற்ற எழுத்துக்களின் key vectors-உடன் ஒப்பிடப்படுகிறது (dot product). இதன் மூலம், attention scores கிடைக்கும். இந்த scores, ஒவ்வொரு எழுத்தும் மற்ற எழுத்துக்களுடன் எவ்வளவு தொடர்புடையது என்பதைக் குறிக்கும்.
- **Masking (tril):** `tril` buffer, decoder-ல் masking செய்ய பயன்படுகிறது. Masking, ஒரு எழுத்து அதற்குப் பின் வரும் எழுத்துக்களை மட்டும் கவனிக்க வேண்டும் என்பதை உறுதி செய்கிறது. இது autoregressive property-ஐ பராமரிக்க உதவுகிறது, அதாவது, ஒரு எழுத்தைக் கணிக்க, அதற்கு முந்தைய எழுத்துக்களை மட்டுமே பயன்படுத்த வேண்டும்.
- **Weighted Aggregation:** Attention scores, softmax function மூலம் probabilities-ஆக மாற்றப்படுகிறது. இந்த probabilities, value vectors-ஐ எவ்வளவு "கவனிக்க வேண்டும்" என்பதைக் குறிக்கும். இந்த probabilities-ஐப் பயன்படுத்தி, value vectors-ன் weighted average கணக்கிடப்படுகிறது. இதுவே attention head-ன் output.

2. `MultiHeadAttention` (பல Attention Heads)

- **Parallel Attention:** பல attention heads-ஐ parallel-ஆக இயக்குகிறது. ஒவ்வொரு head-ம், வெவ்வேறு key, query, value transformations-ஐக் கொண்டிருக்கும். இதனால், ஒவ்வொரு head-ம், input sequence-ல் உள்ள வெவ்வேறு relationships-களை கற்றுக்கொள்ள முடியும்.
- **உதாரணம்:** ஒரு வாக்கியத்தில், "அவன் பந்தை எறிந்தான்" என்று வைத்துக்கொள்வோம். ஒரு attention head, "அவன்" மற்றும் "எறிந்தான்" இடையேயான subject-verb relationship-ஐ கவனிக்கலாம். மற்றொரு head, "பந்தை" மற்றும் "எறிந்தான்" இடையேயான object-verb relationship-ஐ கவனிக்கலாம்.
- **Projection:** அனைத்து heads-ன் outputs-ம் concatenate செய்யப்பட்டு, ஒரு linear transformation (projection) மூலம், `n_embd` dimension-க்கு மாற்றப்படுகிறது.

3. `FeedFoward` (Feedforward Network)

- **Non-linearity:** இது இரண்டு linear layers மற்றும் ஒரு non-linear activation function (ReLU)-ஐக் கொண்டுள்ளது. Linear layers, linear transformations-ஐ மட்டுமே கற்றுக்கொள்ள முடியும். Non-linearity, more complex relationships-ஐ கற்றுக்கொள்ள உதவுகிறது.
- **Expansion and Compression:** முதல் linear layer, input dimension-ஐ `4 * n_embd` ஆக expand செய்கிறது. இரண்டாவது layer, அதை மீண்டும் `n_embd` ஆக compress செய்கிறது. இந்த expansion and compression, model-ன் capacity-ஐ அதிகரிக்கிறது.

4. `Block` (Transformer Block)

- **Communication and Computation:** ஒரு Transformer block, இரண்டு main parts-ஐக் கொண்டுள்ளது:
    - **Communication:** `MultiHeadAttention` layer, input sequence-ல் உள்ள information-ஐ "communicate" செய்கிறது.
    - **Computation:** `FeedFoward` layer, communicated information-ஐப் பயன்படுத்தி, computations செய்கிறது.
- **Residual Connections:** `MultiHeadAttention` மற்றும் `FeedFoward` layers-ன் outputs, input-உடன் add செய்யப்படுகிறது (residual connections). இது, gradients-ஐ propagate செய்ய உதவுகிறது, மேலும் training-ஐ stable ஆக்குகிறது.
- **Layer Normalization:** Layer Normalization, ஒவ்வொரு layer-ன் output-ஐ normalize செய்கிறது. இது training-ஐ stable ஆக்குகிறது, மேலும் model-ன் performance-ஐ மேம்படுத்துகிறது.

5. `BigramLanguageModel` (மொழி மாதிரி)

- **Token Embeddings:** `token_embedding_table`, ஒவ்வொரு எழுத்தையும் ஒரு vector-ஆக மாற்றுகிறது. இந்த vectors, எழுத்துக்களின் semantic meaning-ஐ capture செய்கிறது.
- **Position Embeddings:** `position_embedding_table`, ஒரு எழுத்தின் position-ஐ encode செய்கிறது. இது, model-க்கு sequence-ல் உள்ள order information-ஐ கற்றுக்கொள்ள உதவுகிறது.
- **Transformer Blocks:** `blocks`, பல Transformer blocks-ஐக் கொண்டுள்ளது. ஒவ்வொரு block-ம், input sequence-ன் representation-ஐ மேலும் மேம்படுத்துகிறது.

- **Final Layer Normalization:** `ln_f`, final layer-ன் output-ஐ normalize செய்கிறது.
- **Output Layer:** `lm_head`, final linear layer. இது, அடுத்த எழுத்தைக் கணிக்கிறது.
- **Forward Pass:** `forward` method, input sequence-ஐ process செய்து, output (logits) மற்றும் loss-ஐத் தருகிறது.
- **Text Generation:** `generate` method, புதிய டெக்ஸ்ட் generate செய்ய பயன்படுகிறது. இது, model-ன் predictions-ஐப் பயன்படுத்தி, autoregressively எழுத்துக்களை generate செய்கிறது.

இந்த மாடல், சங்கத் தமிழ் பாடல்களின் structure மற்றும் patterns-களை கற்றுக்கொண்டு, புதிய பாடல்களை உருவாக்கும்.

8.6. Optimizer & Training Loop

மாடலின் weights-களை மேம்படுத்த பயன்படுத்தப்படும் optimizer மற்றும் training process-ஐ control செய்யும் training loop பற்றி இன்னும் விரிவாகவும், தெளிவாகவும் காண்போம்.

```python
# Optimizer
optimizer = torch.optim.AdamW(model.parameters(), lr=learning_rate)

# Training loop
for iter in range(max_iters):
    if iter % eval_interval == 0 or iter == max_iters - 1:
        losses = estimate_loss()
        print(f"step {iter}: train loss {losses['train']:.4f}, val loss {losses['val']:.4f}")
    xb, yb = get_batch('train')
    logits, loss = model(xb, yb)
    optimizer.zero_grad(set_to_none=True)
    loss.backward()
    optimizer.step()
```

1. Optimizer (AdamW):

எதற்காக Optimizer? நமது மொழி மாடலில், பல parameters (weights) உள்ளன. இந்த parameters, சங்கத்தமிழ் பாடல்களின் patterns-களை கற்றுக்கொள்ள பயன்படுகின்றன. Training process-ன் இலக்கு, loss-ஐ குறைக்கும் வகையில் இந்த parameters-ஐ adjust செய்வது. இந்த சீரமைவு செய்யத்தான் optimizer பயன்படுகிறது.

- **AdamW:** AdamW என்பது gradient descent optimization algorithm-ன் ஒரு வகை. இது, Adam optimizer-ன் மேம்படுத்தப்பட்ட version.
  - **Adaptive Learning Rates:** AdamW, ஒவ்வொரு parameter-க்கும் தனித்தனியாக learning rate-ஐ adjust செய்யும். இதனால், சில parameters வேகமாகவும், சில parameters மெதுவாகவும் update செய்யப்படும். இது, training process-ஐ வேகப்படுத்தும்.
  - **Momentum:** AdamW, momentum-ஐயும் பயன்படுத்துகிறது. அதாவது, parameter updates, முந்தைய updates-ன் "direction"-ஐயும் கணக்கில் எடுத்துக்கொள்ளும். இது, training-ஐ stable ஆக்கும், மேலும் local minima-வில் சிக்காமல் தவிர்க்க உதவும்.
  - **Weight Decay:** AdamW, weight decay-ஐ Adam optimizer-ஐ விட சிறப்பாக handle செய்யும். Weight decay, overfitting-ஐ குறைக்க உதவும் ஒரு regularization technique.
- **`model.parameters()`:** இந்த method, மாடலில் train செய்யப்பட வேண்டிய அனைத்து parameters-ஐயும் தருகிறது.
- **`lr=learning_rate`:** Learning rate, optimizer ஒவ்வொரு step-லும் parameters-ஐ எவ்வளவு மாற்ற வேண்டும் என்பதைக் கட்டுப்படுத்தும். Learning rate-ன் சரியான தேர்வு, training process-க்கு மிகவும் முக்கியம்.

2. Training Loop:

Training loop, மாடலை train செய்யும் முழு process-ஐயும் control செய்கிறது.

- **Iterations:** Training loop, `max_iters` முறை இயங்கும். ஒவ்வொரு iteration-லும், மாடல் ஒரு batch data-வைப் பயன்படுத்தி train செய்யப்படும்.
- **Evaluation:** `eval_interval` iterations-க்கு ஒரு முறை, அல்லது கடைசி iteration-ல், `estimate_loss()` function மூலம் train மற்றும் validation loss-கள் கணக்கிடப்படும். இந்த loss-கள், மாடலின் performance-ஐ அளவிட பயன்படும்.
- **Training Step:** ஒவ்வொரு iteration-லும், training step நடக்கும். இது பின்வரும் sub-steps-ஐக் கொண்டுள்ளது:

- **Batch Loading:** `get_batch('train')` function மூலம், train data-வில் இருந்து ஒரு batch data பெறப்படும்.
- **Forward Pass:** மாடல், input batch-ஐ process செய்து, output (logits) மற்றும் loss-ஐ கணக்கிடும்.
- **Gradient Calculation:** Backpropagation மூலம், loss-ஐப் பயன்படுத்தி, மாடலில் உள்ள parameters-க்கான gradients கணக்கிடப்படும். Gradients, loss-ஐ குறைக்கும் வகையில் parameters-ஐ எந்த direction-ல் மாற்ற வேண்டும் என்பதைக் காட்டும்.
- **Parameter Update:** Optimizer, கணக்கிடப்பட்ட gradients-ஐப் பயன்படுத்தி, மாடலின் parameters-ஐ update செய்யும்.

Training loop, மேலே கொடுக்கப்பட்டுள்ள steps-ஐ `max_iters` முறை repeat செய்கிறது. ஒவ்வொரு iteration-லும், மாடல் train data-வைப் பயன்படுத்தி train செய்யப்படும். `eval_interval` iterations-க்கு ஒரு முறை, validation data-வைப் பயன்படுத்தி loss மதிப்பிடப்படும். Training process முடிந்ததும், மாடல் சங்கத் தமிழ் பாடல்களை generate செய்ய தயாராக இருக்கும்.

8.7. Text Generation

train செய்யப்பட்ட மாடலைப் பயன்படுத்தி எப்படிப் புதிய சங்கத்தமிழ் பாடல்களை உருவாக்குவது என்று விரிவாகக் காண்போம்.

```
# புதிய பாடல்களை உருவாக்குதல்
context = torch.zeros((1, 1), dtype=torch.long, device=device)
print(decode(m.generate(context, max_new_tokens=2000)[0].tolist()))
```

விளக்கம்:

- **`context`**: இது text generation-க்கான starting point. `torch.zeros((1, 1), dtype=torch.long, device=device)` என்பது, device-ல் (GPU அல்லது CPU) ஒரு 1x1 tensor-ஐ உருவாக்குகிறது. இந்த tensor-ல் உள்ள மதிப்பு 0 ஆக இருக்கும். இது, "" (Beginning Of Sequence) token-ஐக் குறிக்கிறது. அதாவது, இங்கிருந்து text generation தொடங்கும்.
- **`m.generate(context, max_new_tokens=2000)`**: `generate()` method, மாடலைப் பயன்படுத்தி புதிய text-ஐ generate செய்கிறது.
    - `context`: Text generation-க்கான starting point.

    - `max_new_tokens=2000`: Generate செய்ய வேண்டிய tokens-ன் (எழுத்துக்களின்) அதிகபட்ச எண்ணிக்கை. இங்கு நாம் 2000 tokens வரை generate செய்ய சொல்கிறோம்.
- **`decode(...)`**: `generate()` method எண்களின் list-ஐ output-ஆகத் தரும். `decode()` function, இந்த எண்களை எழுத்துக்களாக மாற்றி, text-ஐ உருவாக்கும்.
- **`print(...)`**: Generate செய்யப்பட்ட text-ஐ (பாடலை) print செய்கிறது.

train செய்யப்பட்ட மொழி மாதிரியைப் பயன்படுத்தி, 2000 எழுத்துக்கள் (tokens) வரை புதிய சங்கத் தமிழ் பாடல்களை உருவாக்கும். `context` variable, text generation-க்கான starting point-ஐக் குறிக்கும். `generate()` method, புதிய text-ஐ generate செய்யும். `decode()` function, generate செய்யப்பட்ட text-ஐ எழுத்துக்களாக மாற்றும். இறுதியாக, `print()` function, generate செய்யப்பட்ட பாடலை print செய்யும்.

Generate செய்யப்படும் பாடலின் தரம், பயன்படுத்தப்படும் training data மற்றும் model hyperparameters-ஐப் பொறுத்தது. Training data-வில் அதிக பாடல்கள் இருந்தால், மற்றும் hyperparameters சரியாக tune செய்யப்பட்டிருந்தால், model நல்ல தரமான பாடல்களை உருவாக்கும்.

Output:

```
கவ்வின் கிடிகா அகழ்கினையும் கடுஞ்சி மிராங்குற் – தலைமாற்றி ஆர்ப் .
.
நெடுகு பரி கிளைற்று, அறிவரம்து
நலன் குல் உண், சிகைஇயப் பகவத்து மன்று
அறும்பிரிச் சொல்லோன்துப்
பெருங்கால் வின்றி
இலை வட்சின் மெலக் காடற்றந் துயர், ஆகவின்றால் இரவுவை, கதாணி
குரும்போன் பரல் ஓதம்பை அறியதல் வூச்சின்
எனவும் புலம்ப வான்றமொடு கொடுச்சி
மைந்து ஒண்ணி வெடுதும் அளம் கண்ட, குனிய!
```

இந்த மாடல் இன்னும் சிறப்பாக வேலை செய்ய, அதிக டேட்டா மற்றும் சில மாற்றங்கள் தேவைப்படலாம். ஆனாலும், இது AI-ன் ஆற்றலைக் காட்டும் ஒரு சுவாரஸ்யமான எடுத்துக்காட்டு!

Colab Notebook

Data