**LIF\_MODEL.py Kodunun Detaylı ve Açıklamalı İncelemesi**

Bu belgede, LIF\_MODEL.py adlı Python dosyasının tüm içeriği adım adım açıklanacak ve her satırda ne yapıldığı, hangi mantıkla çalıştığı ayrıntılı olarak anlatılacaktır.

**1. Kütüphane İçe Aktarımları ve Hazırlık**

import numpy as np

import matplotlib.pyplot as plt

import pandas as pd

from sklearn.metrics import accuracy\_score, classification\_report

from sklearn.preprocessing import MinMaxScaler

import seaborn as sns

from tqdm import tqdm

import warnings

import os # Added for os.path.exists

warnings.filterwarnings('ignore')

Bu kısımda, kodun çalışması için gereken tüm dış kütüphaneler tanımlanmıştır. Her bir modülün ne işe yaradığını inceleyelim:

* **numpy (np)**: Sayısal hesaplamalar ve çok boyutlu dizi yapıları (ndarray) için kullanılır. Sinir ağı ağırlıkları, spike dizileri vb. veriler numpy dizileri (arrays) olarak temsil edilir.
* **matplotlib.pyplot (plt)**: Grafik çizimleri ve veri görselleştirmesi için kullanılır. Eğitim sürecindeki doğruluk grafikleri ve karışıklık matrisleri gibi görseller burada çizilir.
* **pandas (pd)**: Veri işleme ve CSV dosyalarıyla çalışmak için kullanılır. Eğitime hazır verilerin (örn. MNIST, CSV formatında) yüklenmesi ve düzenlenmesi için kullanılan yapı, pandas.DataFrame'dir.
* **sklearn.metrics**: Makine öğrenmesi performans metriklerini hesaplamak için kullanılır. Özellikle accuracy\_score ve classification\_report, modelin sınıflandırma başarısını ölçmek için kritik metriklerdir.
* **sklearn.preprocessing.MinMaxScaler**: Veriyi [0,1] aralığına ölçeklendirmek için kullanılır. LIF modeline girdi olarak verilmeden önce verilerin uygun aralığa çekilmesi, spike üretimi ve eğitim sürecinin stabil olması açısından önemlidir.
* **seaborn (sns)**: İstatistiksel veri görselleştirmesi için kullanılır. matplotlib'e göre daha şık ve özelleştirilebilir grafikler çizmemize yardımcı olur.
* **tqdm**: For döngülerinde ilerleme çubuğu (progress bar) göstermek için kullanılır. Eğitim veya test sırasında iterasyonların ne kadar ilerlediğini görselleştirmek, kullanıcı deneyimini artırır.
* **warnings**: Python'da uyarı mesajlarının kontrolü için kullanılır. warnings.filterwarnings('ignore') satırı, muhtemel DeprecationWarning veya FutureWarning gibi gereksiz uyarıların ekranda gözükmesini engeller.
* **os**: Dosya ve dizin işlemleri için kullanılır. os.path.exists fonksiyonu, dosya/dizin varlığını kontrol etmek için kullanılır. Bu sayede, var olmayan bir dizine yazmaya çalıştığımızda hata almamak için önlem alabiliriz.

Bu satırlar, dosyanın en üst kısmında yer alarak kullanılan kütüphaneleri sisteme tanıtır ve eğitim-analiz sürecinin ön koşullarını hazırlar.

**2. BoltzmannSpikingNeuralNetwork Sınıfı Tanımı**

class BoltzmannSpikingNeuralNetwork:

def \_\_init\_\_(self, input\_size=784, hidden\_size=500, output\_size=10):

"""

Spiking Sinir Ağı (SNN) ve Boltzmann Makinesi entegrasyonu.

input\_size: Girdi katmanındaki nöron sayısı (örneğin MNIST için 28x28=784).

hidden\_size: Gizli katmandaki nöron sayısı.

output\_size: Çıktı katmanındaki nöron sayısı (örneğin sınıflandırma için 10 sınıf).

"""

Bu blokta, BoltzmannSpikingNeuralNetwork adında bir Python sınıfı tanımlanmıştır. Bu sınıf, Spike Tabanlı Sinir Ağı (Spiking Neural Network, SNN) ile Boltzmann Makinesini bir araya getirerek birleştiren özel bir yapıya sahiptir. Başlatıcı (\_\_init\_\_) metodunda, sınıfa ait önemli parametreler ve başlangıç değerleri ayarlanır:

* **input\_size=784**: Girdi katmanında kullanılacak nöron sayısı. Örneğin, MNIST veri setindeki her bir görüntü 28x28 piksel olduğu için toplam 784 piksel/nöron bulunmaktadır.
* **hidden\_size=500**: Gizli katmanda kullanılacak nöron sayısı. Bu değer modelin kapasitesini dengelemek için seçilmiştir; çok büyük değerler hesaplama yükünü artırırken, çok küçük değerler modelin karmaşık desenleri öğrenmesini zorlaştırabilir.
* **output\_size=10**: Çıktı katmanındaki nöron sayısı. Örneğin MNIST sınıflandırmasında 0-9 arası 10 çeşit rakam bulunduğu için, her rakam sınıfı bir nörona karşılık gelir.

**2.1. Başlatıcı İçeriği**

Devam eden kısımda, başlatıcı metodun içindeki detaylı satırlar bulunur. Her birini tek tek inceleyelim:

# Ağırlık Matrislerinin Başlatılması

self.W\_input\_hidden = np.random.normal(loc=0.0, scale=0.05, size=(input\_size, hidden\_size))

self.W\_hidden\_output = np.random.normal(loc=0.0, scale=0.05, size=(hidden\_size, output\_size))

* **self.W\_input\_hidden**: Girdi katmanından gizli katmana geçişteki ağırlık matrisidir. Boyutu (784, 500) olacak şekilde tanımlanmıştır. Her ağırlık, normal dağılımdan rastgele loc=0.0, scale=0.05 parametreleri ile çekilir. Bu sayede başlangıçta ağırlıklar küçük rastgele değerler alır ve eğitim sürecinde yavaşça güncellenebilir.
* **self.W\_hidden\_output**: Gizli katmandan çıktı katmanına geçişteki ağırlık matrisidir. Boyutu (500, 10) olacak şekilde tanımlanır. Yine aynı normal dağılım ile rastgele küçük değerler ataması yapılır.

# Spike Tabanlı Modeling için Parametreler

self.tau\_mem = 20.0 # Zaman sabiti (ms) – LIF modelindeki membran geriliminin sönme hızı

self.v\_reset = 0.0 # Spike sonrası membran potansiyelini resetleme değeri

self.v\_thresh = 1.0 # Spike eşik değeri; bu değere ulaşınca nöron "ateşler"

self.dt = 1.0 # Zaman adımı (ms)

* **self.tau\_mem**: LIF (Leaky Integrate-and-Fire) modelindeki membran potansiyelinin sönme (decay) sabitidir. Bu değer, kaç milisaniyede membran potansiyelinin düşeceğini belirler.
* **self.v\_reset**: Bir nöron spike (") ürettiğinde, membran potansiyeli bu değere sıfırlanır. Genellikle 0 değerinde tutulur.
* **self.v\_thresh**: LIF modelindeki eşik değeridir. Membran potansiyeli bu değere ulaştığında, nöron bir "spike" üretir.
* **self.dt**: Simülasyondaki zaman adımıdır. Bu örnek 1 ms olarak seçilmiştir. LIF modelindeki diferansiyel denklemler bu zaman adımıyla yaklaşık olarak hesaplanır.

# STDP (Spike-Timing Dependent Plasticity) için Parametreler

self.A\_plus = 0.01 # Potansiyel arttırma katsayısı

self.A\_minus = 0.012 # Potansiyel azaltma katsayısı

self.tau\_plus = 20.0 # STDP zaman sabiti (artış)

self.tau\_minus = 20.0# STDP zaman sabiti (azalış)

Burada STDP mekanizması için gerekli parametreler tanımlanır:

* **self.A\_plus**: Pre ve post spike arasındaki zamana bağlı olarak ağırlıkların pozitif yönde ne kadar değişeceğini belirleyen katsayı.
* **self.A\_minus**: Ağırlıkların negatif yönde (depolama) ne kadar azalacağını belirleyen katsayı.
* **self.tau\_plus**, **self.tau\_minus**: STDP'deki pozitif (potansiyasyon) ve negatif (depresyon) zaman sabitleridir. Bu zaman sabitleri, pre-post zaman farkı ne kadar büyükse, o kadar az değişim olacağını belirler.

# Boltzmann Makinesi için Parametreler

self.temperature = 1.0 # Başlangıç sıcaklığı

self.temp\_decay = 0.99 # Her epok sonunda sıcaklığı çarpılacak faktör

# Eğitim için Diğer Ayarlar

self.learning\_rate = 0.01 # Genel öğrenme hızı

self.num\_timesteps = 100 # Her girdide kaç zaman adımı ile simülasyon yapılacağı

self.batch\_size = 64 # Minibatch boyutu

Boltzmann Makinesi entegrasyonu için:

* **self.temperature**: Boltzmann makinesinde kullanılan sıcaklık parametresidir. Yüksek sıcaklık, olasılık dağılımını daha eşit hale getirir; düşük sıcaklık, dağılımı daha keskin yapar.
* **self.temp\_decay**: Her eğitim epokunun sonunda temperature değerinin kendisiyle çarpılacağı faktördür. Bu sayede sıcaklık zamanla azalır (annealing).

Eğitim sürecine ait parametreler:

* **self.learning\_rate**: Ağırlık güncellemelerinde kullanılacak genel öğrenme hızıdır.
* **self.num\_timesteps**: Bir örnek (örneğin bir MNIST görüntüsü) için simülasyonda kaç zaman adımının işleneceğini belirler. Zaman adımlarındaki spike dizileri bu sayede uzunluk kazanır.
* **self.batch\_size**: Minibatch büyüklüğüdür. Aynı anda batch\_size kadar örnek alınarak toplu ağırlık güncellemeleri yapılır.

# Nöron Değerleri İçin Hazırlık

self.v\_mem\_input = np.zeros(input\_size) # Girdi katmanı membran potansiyelleri

self.v\_mem\_hidden = np.zeros(hidden\_size) # Gizli katman membran potansiyelleri

self.v\_mem\_output = np.zeros(output\_size) # Çıktı katman membran potansiyelleri

* **self.v\_mem\_input**, **self.v\_mem\_hidden**, **self.v\_mem\_output**: LIF nöronları için membran potansiyellerini tutan dizilerdir. Başlangıçta tüm potansiyeller sıfırdır.

# Spike Traces (Hızlı ve Yavaş) İçin Hazırlık

self.exp\_decay\_plus\_fast = np.exp(-self.dt / (self.tau\_plus / 5))

self.exp\_decay\_minus\_fast = np.exp(-self.dt / (self.tau\_minus / 5))

self.exp\_decay\_plus\_slow = np.exp(-self.dt / self.tau\_plus)

self.exp\_decay\_minus\_slow = np.exp(-self.dt / self.tau\_minus)

STDP hesaplamalarındaki "trace" değerleri için hem hızlı hem yavaş değerler tanımlanır:

* **self.exp\_decay\_plus\_fast** ve **self.exp\_decay\_minus\_fast**: STDP'nin hızlı kısım izlerini güncellemek için kullanılan eksi üssel değerler. Hızlı zaman sabiti, tau\_plus / 5 veya tau\_minus / 5 formülüyle elde edilir.
* **self.exp\_decay\_plus\_slow** ve **self.exp\_decay\_minus\_slow**: STDP'nin yavaş kısım izlerini güncellemek için kullanılan değerler. Bu izler, tau\_plus ve tau\_minus zaman sabitlerine göre hesaplanır.

# İz Takibi Dizileri (Başlangıçta Sıfır)

self.pre\_trace\_input\_fast = np.zeros((batch\_size, input\_size))

self.post\_trace\_hidden\_fast = np.zeros((batch\_size, hidden\_size))

self.pre\_trace\_hidden\_fast = np.zeros((batch\_size, hidden\_size))

self.post\_trace\_output\_fast = np.zeros((batch\_size, output\_size))

self.pre\_trace\_input\_slow = np.zeros((batch\_size, input\_size))

self.post\_trace\_hidden\_slow = np.zeros((batch\_size, hidden\_size))

self.pre\_trace\_hidden\_slow = np.zeros((batch\_size, hidden\_size))

self.post\_trace\_output\_slow = np.zeros((batch\_size, output\_size))

STDP algoritmasında kullanılacak iz (trace) değerleri için gerekli diziler burada oluşturulur. Hem hızlı (fast) hem de yavaş (slow) izler, batch\_size boyutunda satırlara sahip 2D diziler olarak tanımlanır. Bu dizilerin amacı, bir nöronun ne kadar sıklıkla ve hangi zaman adımlarında ateşlediğini izlemek ve ardından ağırlık güncellemelerini hesaplamaktır.

* Satır sayısı = batch\_size (örneğin 64 örnek birden ele alınırken), sütun sayısı = ilgili katmandaki nöron sayısı (girdi, gizli veya çıktı katmanı).

# Verinin Normalize Edilmesi İçin Scaler

self.scaler = MinMaxScaler(feature\_range=(0, 1))

Veri ön işleme adımında, girdi piksel değerlerini [0-1] aralığına çekmek için bir MinMaxScaler nesnesi oluşturulur. Bu sayede, LIF modelinin spike üretme eşiğine uygun değer aralığı elde edilir.

**3. lif\_spike\_generation Metodu (LIF Spike Üretimi)**

def lif\_spike\_generation(self, input\_data):

"""

Bu metot, verilen girdi verisi üzerinden LIF (Leaky Integrate-and-Fire) nöronları kullanarak spike dizisi üretir.

input\_data: (batch\_size, input\_size) boyutunda, [0-1] aralığında normalize edilmiş girdi değerleri.

dönüş: (batch\_size, num\_timesteps, input\_size) boyutunda ikili (0/1) spike dizisi.

"""

batch\_size = input\_data.shape[0]

# Membran potansiyellerini sıfırlama (her giriş örneği için)

self.v\_mem\_input = np.zeros((batch\_size, self.v\_mem\_input.shape[0]))

spike\_output = np.zeros((batch\_size, self.num\_timesteps, self.v\_mem\_input.shape[1]))

for t in range(self.num\_timesteps):

# Membran potansiyelinin zaman içinde birikimi ve sönmesi

self.v\_mem\_input = self.v\_mem\_input + (input\_data - self.v\_mem\_input) \* (self.dt / self.tau\_mem)

# Eşik değerine ulaşanları spike olarak belirleme

spikes = (self.v\_mem\_input >= self.v\_thresh).astype(float)

# Spike üretimi

spike\_output[:, t, :] = spikes

# Spike atan nöronların potansiyelini sıfırlama

self.v\_mem\_input[spikes == 1] = self.v\_reset

return spike\_output

Bu metot, batch halinde verilen girdi verisini zamanı takiben LIF modeline sokarak bir spike dizisi (0 veya 1) üretir. Açıklamaları adım adım inceleyelim:

1. **Girdi Boyutlarının Tanımlanması**
2. batch\_size = input\_data.shape[0]
   * input\_data dizisinin birinci boyutu, örnek sayısını (batch size) belirtir. Örneğin, 64 örnek varsa batch\_size=64 olur.
3. **Membran Potansiyelinin ve Spike Çıktısı Dizisinin Oluşturulması**
4. self.v\_mem\_input = np.zeros((batch\_size, self.v\_mem\_input.shape[0]))
5. spike\_output = np.zeros((batch\_size, self.num\_timesteps, self.v\_mem\_input.shape[1]))
   * self.v\_mem\_input boyutu (batch\_size, input\_size) olacak şekilde sıfırlanır. Böylece her bir örnek için ayrı bir nöron potansiyeli dizisi tutulur.
   * spike\_output dizisi, her zaman adımında oluşan spikeları saklamak için üç boyutlu oluşturulur: (batch\_size, num\_timesteps, input\_size). Burada num\_timesteps, simülasyonun zaman boyutu (örneğin 100 ms boyunca her milisaniyede bir) demektir.
6. **Zaman Döngüsü Başlatma**
7. for t in range(self.num\_timesteps):
   * Her zaman adımı (t) için aşağıdaki güncellemeler yapılacaktır. t sıfırdan başlayarak num\_timesteps-1'e kadar gider.
8. **Membran Potansiyelinin Güncellenmesi**
9. self.v\_mem\_input = self.v\_mem\_input + (input\_data - self.v\_mem\_input) \* (self.dt / self.tau\_mem)
   * Bu satır, LIF modelinin leaky (sızıntı) kısmını matematiksel olarak uygular. Şöyle açıklanabilir:
     + input\_data: İlgili örneğin giriş geri beslemesini temsil eder. Bu değer [0-1] aralığında normalize edilmiş piksellerin değeri olabilir.
     + self.v\_mem\_input: Mevcut membran potansiyeli.
     + (input\_data - self.v\_mem\_input): Hali hazırdaki membran potansiyelinin, girdi akışı ile dengeye ulaşma farkını belirtir.
     + (self.dt / self.tau\_mem): Küçük zaman adımı oranı ile bu farkın (dengeye ulaşma) hızlı mı yoksa yavaş mı olacağını belirler.
     + +: Eski membran potansiyeline, denge farkının bu katsayı ile çarpılmış hali eklenir.
10. **Spike Üretme (Eşik Kontrolü)**
11. spikes = (self.v\_mem\_input >= self.v\_thresh).astype(float)
    * self.v\_mem\_input >= self.v\_thresh ifadesi, her bir nöron için True veya False döner. Eğer membran potansiyeli eşik değerine (v\_thresh) ulaşmışsa, o nöron ateşleyecektir.
    * .astype(float): Boolean değerler True veya False yerine 1.0 veya 0.0 şeklinde sayısal spike değeri üretir.
12. **Spike Çıkışının Saklanması**
13. spike\_output[:, t, :] = spikes
    * spike\_output'un t zaman dilimindeki dilimine, az önce hesaplanan spikes matrisini atar. Böylece, hangi örneğin hangi nöronu hangi zamanda ateşlediği kaydedilmiş olur.
14. **Membran Potansiyelinin Resetlenmesi**
15. self.v\_mem\_input[spikes == 1] = self.v\_reset
    * Spike (") üreten nöronların potansiyelini sıfıra (v\_reset) ayarlar. Böylece, bir nöron ateşledikten sonra boşalır ve yeniden birikmeye başlar.
16. **Döngü Tamamlandıktan Sonra Spike Dizisinin Döndürülmesi**
17. return spike\_output
    * Tüm zaman adımları bittikten sonra, her örnek ve her zaman adımında hangi nöronun ateşlediğini belirten 3-boyutlu spike\_output dizisi dış dünyaya iletilir.

**4. train Metodu (Model Eğitimi)**

def train(self, X\_train, y\_train, X\_val=None, y\_val=None, num\_epochs=10):

"""

Modelin eğitim sürecini yönetir. STDP, Boltzmann ve ileri besleme adımlarını içerir.

X\_train: (num\_samples, input\_size) boyutunda eğitim verisi

y\_train: (num\_samples,) boyutunda eğitim etiketleri

X\_val, y\_val: Doğrulama verisi ve etiketleri (opsiyonel)

num\_epochs: Eğitimde kaç tam geçiş yapılacağı

dönüş: training\_accuracy\_list, validation\_accuracy\_list

"""

Bu metod, modelin ana eğitim döngüsünü tanımlar. İleri besleme (feedforward), STDP tabanlı ağırlık güncellemeleri, Boltzmann simülasyonları ve doğrulama adımları içeren kapsamlı bir eğitim döngüsüdür. Adım adım inceleyelim:

1. **Yedek Dizilerin Oluşturulması**
2. training\_accuracy\_list = []
3. validation\_accuracy\_list = [] if X\_val is not None else None
   * Eğitim ve doğrulama doğruluklarını kaydetmek için boş listeler oluşturulur. Her epoch sonunda ilgili doğruluk değeri bu listelere eklenir.
4. **Epoch Döngüsü**
5. for epoch in range(num\_epochs):
6. # Öğrenme oranının ayarlanması (öğe bazlı annealing)
7. self.temperature \*= self.temp\_decay

- `epoch` değişkeni, 0'dan `num\_epochs - 1`'e kadar döner.

- Başlangıçta tanımlanan `self.temperature` değeri, her epoch sonunda `self.temp\_decay` (örneğin 0.99) ile çarpılarak kademeli olarak düşürülür. Bu sayede Boltzmann makinesindeki sıcaklık düşer ve model zamanla daha kararlı olmayı öğrenir.

3. \*\*Minibatch Hazırlığı (Verinin Karıştırılması ve Parçalanması)\*\*

```python

permutation = np.random.permutation(X\_train.shape[0])

X\_train\_shuffled = X\_train[permutation]

y\_train\_shuffled = y\_train[permutation]

for i in range(0, X\_train.shape[0], self.batch\_size):

X\_batch = X\_train\_shuffled[i:i + self.batch\_size]

y\_batch = y\_train\_shuffled[i:i + self.batch\_size]

* np.random.permutation(X\_train.shape[0]): Eğitim verisinin indekslerini rastgele karıştırmak için kullanılır.
* X\_train\_shuffled ve y\_train\_shuffled: Karıştırılmış veriler, böylece her epoch'da verinin farklı bir sıralaması kullanılarak overfitting riski azalır.
* İç döngüde batch\_size adımıyla ilerleyerek X\_batch ve y\_batch oluşturulur. Örneğin toplam 60000 eğitim örneği varsa ve batch\_size=64 ise, yaklaşık 938 minibatch oluşacaktır.

1. **Spike Dizilerinin Üretilmesi**
2. spike\_input = self.lif\_spike\_generation(self.scaler.transform(X\_batch))

- `X\_batch` verisi önce `self.scaler.transform` ile [0-1] aralığına normalize edilir.

- Normalize edilmiş veriler, `lif\_spike\_generation` metoduna verilir ve her örnek için `num\_timesteps` uzunluğunda bir spike dizisi (`0` ve `1`'lerden oluşan) elde edilir.

5. \*\*Gizli Katmana İleri Besleme ve STDP İzlerinin Güncellenmesi\*\*

```python

# Gizli katman için membran potansiyellerinin sıfırlanması

self.v\_mem\_hidden = np.zeros((self.batch\_size, self.v\_mem\_hidden.shape[0]))

hidden\_spike\_output = np.zeros((self.batch\_size, self.num\_timesteps, self.v\_mem\_hidden.shape[1]))

# Hızlı ve yavaş izler için geçici diziler

pre\_trace\_input\_fast = np.zeros((self.batch\_size, self.W\_input\_hidden.shape[0]))

post\_trace\_hidden\_fast = np.zeros((self.batch\_size, self.W\_input\_hidden.shape[1]))

pre\_trace\_hidden\_fast = np.zeros((self.batch\_size, self.W\_hidden\_output.shape[0]))

post\_trace\_output\_fast = np.zeros((self.batch\_size, self.W\_hidden\_output.shape[1]))

pre\_trace\_input\_slow = np.zeros\_like(pre\_trace\_input\_fast)

post\_trace\_hidden\_slow = np.zeros\_like(post\_trace\_hidden\_fast)

pre\_trace\_hidden\_slow = np.zeros\_like(pre\_trace\_hidden\_fast)

post\_trace\_output\_slow = np.zeros\_like(post\_trace\_output\_fast)

* Gizli katman için membran potansiyelleri ve spike dizileri için sıfırlama yapılır. Her minibatch başında bu diziler yeniden oluşturulur.
* STDP için gereken hızlı (fast) ve yavaş (slow) izler, girdi & gizli katman bağlamlarında ayrı ayrı tanımlanır. Her biri, batch\_size ve ilgili katman boyutları kadar alan kaplar.

1. **Zaman Adımları İçindeki Döngü (Gizli Katman)**
2. for t in range(self.num\_timesteps):
3. current\_input\_spikes = spike\_input[:, t, :]
4. # Membran potansiyelinin güncellenmesi (gizli katman)
5. self.v\_mem\_hidden = self.v\_mem\_hidden + (np.dot(current\_input\_spikes, self.W\_input\_hidden) - self.v\_mem\_hidden) \* (self.dt / self.tau\_mem)
6. hidden\_spikes = (self.v\_mem\_hidden >= self.v\_thresh).astype(float)
7. hidden\_spike\_output[:, t, :] = hidden\_spikes
8. # Potansiyel resetleme (gizli katman)
9. self.v\_mem\_hidden[hidden\_spikes == 1] = self.v\_reset
10. # STDP izlerinin güncellenmesi (gizli katman)
11. post\_trace\_hidden\_fast = post\_trace\_hidden\_fast \* self.exp\_decay\_minus\_fast + hidden\_spikes
12. pre\_trace\_input\_fast = pre\_trace\_input\_fast \* self.exp\_decay\_plus\_fast + current\_input\_spikes
13. pre\_trace\_hidden\_fast = pre\_trace\_hidden\_fast \* self.exp\_decay\_plus\_fast + hidden\_spikes
14. post\_trace\_output\_fast = post\_trace\_output\_fast \* self.exp\_decay\_minus\_fast + 0 # henüz output hesaplanmadı
15. pre\_trace\_input\_slow = pre\_trace\_input\_slow \* self.exp\_decay\_plus\_slow + current\_input\_spikes
16. post\_trace\_hidden\_slow = post\_trace\_hidden\_slow \* self.exp\_decay\_minus\_slow + hidden\_spikes
17. pre\_trace\_hidden\_slow = pre\_trace\_hidden\_slow \* self.exp\_decay\_plus\_slow + hidden\_spikes
18. post\_trace\_output\_slow = post\_trace\_output\_slow \* self.exp\_decay\_minus\_slow + 0

- Her `t` zaman adımında:

1. `current\_input\_spikes`: Girdi nöronlarının `t` zaman adımındaki spike durumlarını alır.

2. Gizli katman membran potansiyeli güncellenir:

- `np.dot(current\_input\_spikes, self.W\_input\_hidden)`: Girdi spike'ları ile input-hidden ağırlık matrisinin çarpımı, gizli katmana gelen toplam sinyali temsil eder.

- Sızıntı (`leaky`) işlemi: `(toplam\_girdi - eski\_potansiyel) \* (dt / tau\_mem)` formülü ile birikim ve sönme dengesi sağlanır.

3. `hidden\_spikes`: Eğer membran potansiyeli eşik değeri aşmışsa (`>= v\_thresh`), spike üretimi gerçekleşir.

4. `hidden\_spike\_output` dizisine spike değeri (`1` veya `0`) atanır.

5. Spike atan nöronların membran potansiyeli `v\_reset` ile sıfırlanır.

6. STDP izleri hızlı (`fast`) ve yavaş (`slow`) biçimde güncellenir. Burada çıkış katmanı henüz hesaplanmadığı için `post\_trace\_output\_fast` ve `post\_trace\_output\_slow` içerisine `0` değeri eklenir.

7. \*\*Çıktı Katmanına İleri Besleme ve STDP İzlerinin Hesaplanması\*\*

```python

# Çıktı katman için membran potansiyelleri

self.v\_mem\_output = np.zeros((self.batch\_size, self.v\_mem\_output.shape[0]))

output\_spike\_output = np.zeros((self.batch\_size, self.num\_timesteps, self.v\_mem\_output.shape[1]))

for t in range(self.num\_timesteps):

current\_hidden\_spikes = hidden\_spike\_output[:, t, :]

self.v\_mem\_output = self.v\_mem\_output + (np.dot(current\_hidden\_spikes, self.W\_hidden\_output) - self.v\_mem\_output) \* (self.dt / self.tau\_mem)

output\_spikes = (self.v\_mem\_output >= self.v\_thresh).astype(float)

output\_spike\_output[:, t, :] = output\_spikes

# Çıktı membran potansiyel resetleme

self.v\_mem\_output[output\_spikes == 1] = self.v\_reset

# STDP iz güncellemeleri (çıkış katmanı için)

post\_trace\_hidden\_fast = post\_trace\_hidden\_fast \* self.exp\_decay\_minus\_fast + hidden\_spikes

pre\_trace\_input\_fast = pre\_trace\_input\_fast \* self.exp\_decay\_plus\_fast + hidden\_spikes

post\_trace\_output\_fast = post\_trace\_output\_fast \* self.exp\_decay\_minus\_fast + output\_spikes

pre\_trace\_hidden\_fast = pre\_trace\_hidden\_fast \* self.exp\_decay\_plus\_fast + output\_spikes

post\_trace\_hidden\_slow = post\_trace\_hidden\_slow \* self.exp\_decay\_minus\_slow + hidden\_spikes

pre\_trace\_input\_slow = pre\_trace\_input\_slow \* self.exp\_decay\_plus\_slow + hidden\_spikes

post\_trace\_output\_slow = post\_trace\_output\_slow \* self.exp\_decay\_minus\_slow + output\_spikes

pre\_trace\_hidden\_slow = pre\_trace\_hidden\_slow \* self.exp\_decay\_plus\_slow + output\_spikes

* Bu bölüm gizli katman çıktılarını kullanarak çıktı katmanında ileri besleme ve STDP izlerini günceller:
  1. self.v\_mem\_output: Çıktı katmanı membran potansiyelleri, her minibatch başında sıfırlanır.
  2. output\_spike\_output: Çıktı spike dizisini saklamak için üç boyutlu dizi oluşturulur.
  3. Döngüde her zaman adımı için:
     + current\_hidden\_spikes: Gizli katmandan o zaman diliminde gelen spike dizisi.
     + Membran potansiyeli güncellemesi: Benzer LIF formülü, bu kez gizli-spike ile self.W\_hidden\_output çarpımı kullanılarak yapılır.
     + output\_spikes: Eğer eşik aşılmışsa 1, aksi halde 0 üretilir.
     + output\_spike\_output dizisi güncellenir.
     + Spike atan output nöronlarının potansiyeli v\_reset ile sıfırlanır.
     + STDP izleri hem hızlı hem yavaş olarak güncellenir. post\_trace\_hidden\_\* ve pre\_trace\_hidden\_\* gibi izler her katmanda ayrı ayrı güncellenir.

1. **Ağırlık Güncellemeleri (STDP Uygulaması)**
2. # Ortalamaların Hesaplanması
3. avg\_post\_trace\_hidden\_fast = np.mean(post\_trace\_hidden\_fast, axis=0)
4. avg\_current\_input\_spikes = np.mean(current\_input\_spikes, axis=0)
5. avg\_pre\_trace\_input\_fast = np.mean(pre\_trace\_input\_fast, axis=0)
6. avg\_hidden\_spikes = np.mean(hidden\_spikes, axis=0)
7. avg\_output\_spikes = np.mean(output\_spikes, axis=0)
8. # STDP Katsayılarının Ağırlıklara Uygulanması
9. dW\_pot\_ih = self.A\_plus \* np.outer(avg\_post\_trace\_hidden\_fast, avg\_current\_input\_spikes)
10. dW\_dep\_ih = self.A\_minus \* np.outer(avg\_hidden\_spikes, avg\_pre\_trace\_input\_fast)
11. self.W\_input\_hidden += (dW\_pot\_ih - dW\_dep\_ih) \* 0.01

- STDP mekanizmasında hız ve yavaş izlerin ortalamaları hesaplanarak ağırlık güncelleme terimleri (`dW\_pot` ve `dW\_dep`) elde edilir.

- \*\*`np.outer`\*\*: İki vektörün dış çarpımı alınır. Bu, her girdi nöronunun her gizli nöronla etkileşimi için ayrı bir değer üretir.

- `dW\_pot\_ih`: Girdi-gizli arasındaki sinaptik potansiyasyon terimi.

- `dW\_dep\_ih`: Girdi-gizli arasındaki sinaptik depresyon terimi.

- `self.W\_input\_hidden` ağırlıkları, bu iki terimin farkıyla ve sabit çarpan (örn. 0.01) kullanılarak güncellenir.

9. \*\*Hidden-Output Ağırlık Güncellemeleri\*\*

```python

avg\_post\_trace\_output\_fast = np.mean(post\_trace\_output\_fast, axis=0)

avg\_pre\_trace\_hidden\_fast = np.mean(pre\_trace\_hidden\_fast, axis=0)

dW\_pot\_ho = self.A\_plus \* np.outer(avg\_post\_trace\_output\_fast, avg\_hidden\_spikes)

dW\_dep\_ho = self.A\_minus \* np.outer(avg\_output\_spikes, avg\_pre\_trace\_hidden\_fast)

self.W\_hidden\_output += (dW\_pot\_ho - dW\_dep\_ho) \* 0.01

* Çıktı katmanına yönelik STDP ağırlık güncellemeleri dW\_pot\_ho (hid to output potansiyasyon) ve dW\_dep\_ho (hid to output depresyon) terimleri ile hesaplanır.
* self.W\_hidden\_output ağırlıkları benzer şekilde güncellenir.

1. **Boltzmann Makinesi Ağırlık Güncellemesi**
2. # Boltzmann Makinesi için Gibbs Örnekleme / Contrastive Divergence vs. benzeri bir yaklaşım burada uygulanabilir.
3. probs = self.softmax(np.dot(hidden\_spikes, self.W\_hidden\_output) / self.temperature)
4. target = np.zeros\_like(probs)
5. for idx, label in enumerate(y\_batch):
6. target[idx, label] = 1
7. error = target - probs
8. self.W\_hidden\_output += self.learning\_rate \* np.dot(hidden\_spikes.T, error)
   * Bu kısımda, Boltzmann makinesine benzer bir yaklaşım uygulanmıştır:
     1. softmax fonksiyonu ile gizli katman spikeleri, çıktı katmanına olasılık dağılımı olarak beslenir.
     2. target: One-hot (tek-hedef) vektörler, minibatch içindeki her örnek için doğru sınıf etiketini (y\_batch) içerir.
     3. error: Gerçek (target) ve modelin tahmini (probs) arasındaki fark.
     4. Ağırlık güncellemesi: hidden\_spikes.T ile transpoze edilmiş gizli len spikeler ile error çarpılarak self.learning\_rate katsayısı ile güncelleme yapılır.
9. **Epoch Sonu Değerlendirmesi ve Doğruluk Hesaplaması**
10. # Eğitim sırasında doğruluk hesaplama (opsiyonel olarak ara değerleri alabiliriz)
11. train\_preds = np.argmax(probs, axis=1)
12. train\_accuracy = accuracy\_score(y\_batch, train\_preds)
13. training\_accuracy\_list.append(train\_accuracy)
14. # Doğrulama seti varsa doğruluk kontrolü
15. if X\_val is not None:
16. val\_preds, \_ = self.predict(X\_val)
17. val\_accuracy = accuracy\_score(y\_val, val\_preds)
18. validation\_accuracy\_list.append(val\_accuracy)
19. return training\_accuracy\_list, validation\_accuracy\_list
    * probs: Son minibatch içindeki tahmin olasılıkları.
    * np.argmax(probs, axis=1): Olasılık dağılımı en yüksek olan sınıf indeksi, tahmin (train\_preds) olarak alınır.
    * accuracy\_score: Gerçek etiketler (y\_batch) ile tahmin edilen etiketler (train\_preds) karşılaştırılarak doğruluk oranı hesaplanır.
    * Eğitim doğruluğu (train\_accuracy) her minibatch sonunda training\_accuracy\_list'e eklenir.
    * Eğer doğrulama (validation) verisi sağlanmışsa, tüm doğrulama verisi için self.predict metodu çağrılarak tahminler alınır. Doğruluk oranı hesaplanır ve validation\_accuracy\_list dizisine eklenir.
    * Metot sonunda, her epoch adımındaki eğitim ve doğrulama doğruluk oranlarını içeren listeler döndürülür.

**5. softmax Metodu**

def softmax(self, x):

exp\_x = np.exp(x - np.max(x, axis=1, keepdims=True))

return exp\_x / np.sum(exp\_x, axis=1, keepdims=True)

Bu metot, giriş olarak aldığı x matrisine softmax fonksiyonu uygular ve her satırın değerlerini normalize ederek olasılık dağılımına çevirir. Açıklamalar:

1. **Stabilite için Kaydırma**
2. exp\_x = np.exp(x - np.max(x, axis=1, keepdims=True))
   * x - np.max(x, axis=1, keepdims=True): Satır bazında en büyük değeri çıkararak sayısal taşma (overflow) riskini azaltır.
   * np.exp(...): Üssel fonksiyon uygulanır.
3. **Normalizasyon**
4. return exp\_x / np.sum(exp\_x, axis=1, keepdims=True)
   * Üssel değerler toplamı alınır ve her bir üssel değer bu toplam değerine bölünerek 0-1 aralığında olasılık değerleri elde edilir.

**6. predict Metodu (Tahmin Yapma)**

def predict(self, X\_test):

"""

Modelin test verisi üzerinde tahmin yapmasını sağlar.

X\_test: (num\_test\_samples, input\_size) boyutunda test verisi

dönüş: tahminler (labels), olasılık değerleri (confidences)

"""

spike\_input\_test = self.lif\_spike\_generation(self.scaler.transform(X\_test))

batch\_size\_test = X\_test.shape[0]

self.v\_mem\_hidden = np.zeros((batch\_size\_test, self.v\_mem\_hidden.shape[0]))

hidden\_spike\_output = np.zeros((batch\_size\_test, self.num\_timesteps, self.v\_mem\_hidden.shape[1]))

predict metodu, eğitim sırasında güncellenen ağırlıkları kullanarak yeni verilere (test/validation) ilişkin tahminler üretir. Aşamalar:

1. **Spike Dizisi Oluşturma**
2. spike\_input\_test = self.lif\_spike\_generation(self.scaler.transform(X\_test))
   * Eğitimde kullanılan lif\_spike\_generation metodu, normalize edilmiş test verisi için spike dizisi üretir.
3. **Gizli Katman ve Spike Dizisi Hazırlığı**
4. batch\_size\_test = X\_test.shape[0]
5. self.v\_mem\_hidden = np.zeros((batch\_size\_test, self.v\_mem\_hidden.shape[0]))
6. hidden\_spike\_output = np.zeros((batch\_size\_test, self.num\_timesteps, self.v\_mem\_hidden.shape[1]))
   * Test örnek sayısına (batch\_size\_test) göre gizli katman membran potansiyelleri sıfırlanır.
   * hidden\_spike\_output, her bir test örneği ve her bir zaman adımında gizli nöronların ateşleme durumlarını saklar.
7. **Zaman Adımları İçinde Gizli Katman İleri Besleme**
8. for t in range(self.num\_timesteps):
9. current\_input\_spikes = spike\_input\_test[:, t, :]
10. self.v\_mem\_hidden = self.v\_mem\_hidden + (np.dot(current\_input\_spikes, self.W\_input\_hidden) - self.v\_mem\_hidden) \* (self.dt / self.tau\_mem)
11. hidden\_spikes = (self.v\_mem\_hidden >= self.v\_thresh).astype(float)
12. hidden\_spike\_output[:, t, :] = hidden\_spikes
13. self.v\_mem\_hidden[hidden\_spikes == 1] = self.v\_reset
    * Eğitimde olduğu gibi, test sırasında da her zaman adımında:
      1. Girdi spike'ları alınır.
      2. Gizli katman membran potansiyelleri LIF formülüne göre güncellenir.
      3. Spike olup olmadığı kontrol edilir.
      4. Spike varsa hidden\_spike\_output dizisine kaydedilir.
      5. Potansiyeli sıfırlanır.
14. **Çıktı Katman İleri Besleme ve Softmax Uygulama**
15. # Çıktı katman hesaplaması
16. output\_mem = np.zeros((batch\_size\_test, self.v\_mem\_output.shape[0]))
17. for t in range(self.num\_timesteps):
18. current\_hidden\_spikes = hidden\_spike\_output[:, t, :]
19. output\_mem = output\_mem + (np.dot(current\_hidden\_spikes, self.W\_hidden\_output) - output\_mem) \* (self.dt / self.tau\_mem)
    * output\_mem dizisi, tüm zaman adımlarının toplam çıktısını içerir. Her adımda gizli katmandan gelen spike'lar ağırlıklarla çarpılır ve LIF formülü uygulanarak potansiyel birikimi devam eder.
20. output\_probs = self.softmax(output\_mem / self.temperature)
21. preds = np.argmax(output\_probs, axis=1)
22. confidences = np.max(output\_probs, axis=1)
23. return preds, confidences
    * Zaman adımları tamamlandıktan sonra, output\_mem dizisi Boltzmann sıcaklığına (self.temperature) bölünerek softmax fonksiyonuna gönderilir.
    * Softmax sonuçları (output\_probs), her örnek için 10 sınıfa ait olasılık değerleri içerir.
    * preds: En yüksek olasılığı veren sınıf indeksi (tahmin).
    * confidences: Her örneğin en yüksek olasılık değeri, tahminin güvenini (confidence) verir.
    * Metot, preds ve confidences dizilerini döndürür.

**7. load\_data Fonksiyonu (Veri Yükleme ve Hazırlama)**

def load\_data(data\_path, test\_size=0.2, random\_state=42):

"""

Veriyi CSV formatından okuyup eğitim ve test setine böler.

- data\_path: CSV dosyasının bulunduğu dizin.

- test\_size: Test seti oranı (örneğin 0.2 = %20 test).

- random\_state: Bölme işleminin tekrar üretilebilir olması için sabit tohum.

dönüş: X\_train, X\_test, y\_train, y\_test

"""

data = pd.read\_csv(data\_path)

X = data.drop('label', axis=1).values

y = data['label'].values

from sklearn.model\_selection import train\_test\_split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

X, y, test\_size=test\_size, random\_state=random\_state)

return X\_train, X\_test, y\_train, y\_test

Bu yardımcı fonksiyon, CSV dosyasında depolanan veriyi okuyup eğitim ve test setlerine ayırmak için kullanılır:

1. pd.read\_csv(data\_path): Belirtilen data\_path yolundaki CSV dosyasını yükler. DataFrame biçiminde data değişkenine atar.
2. X = data.drop('label', axis=1).values: label sütunu (etiketler dışındaki tüm sütunlar) girdi değerleri olarak alınır. .values ile NumPy dizisine dönüştürülür.
3. y = data['label'].values: Sadece etiket sütunu (label) alınır.
4. train\_test\_split: sklearn.model\_selection modülünden alınan bu fonksiyon, veriyi eğitim ve test setlerine böler. test\_size=0.2 demek, verinin %20'si test için ayrılır.
5. return X\_train, X\_test, y\_train, y\_test: Bölünmüş veri kümesi geri döndürülür.

**8. main Fonksiyonu (Programın Ana Akışı)**

def main():

# 1. Veri Yükleme

data\_path = 'mnist.csv' # Örneğin MNIST veri seti CSV formatında

if not os.path.exists(data\_path):

print(f"Veri yolu bulunamadı: {data\_path}")

return None, None, None, None, None

X\_train, X\_test, y\_train, y\_test = load\_data(data\_path)

# 2. Model Oluşturma

snn\_model = BoltzmannSpikingNeuralNetwork(input\_size=X\_train.shape[1], hidden\_size=500, output\_size=10)

# 3. Eğitim (opsiyonel doğrulama seti burada None olarak bırakıldı)

num\_epochs = 5

accuracies, \_ = snn\_model.train(X\_train, y\_train, X\_val=None, y\_val=None, num\_epochs=num\_epochs)

# 4. Test Aşaması

test\_preds, test\_confs = snn\_model.predict(X\_test)

# 5. Sonuçların Yazdırılması

print("Test Doğruluğu:", accuracy\_score(y\_test, test\_preds))

print("Sınıflandırma Raporu:\n", classification\_report(y\_test, test\_preds))

return snn\_model, accuracies, X\_test, test\_preds, test\_confs

main fonksiyonu programın yürütülme akışını kontrol eder:

1. **Veri Yükleme**
   * data\_path = 'mnist.csv': mnist.csv dosyasının var olup olmadığı os.path.exists ile kontrol edilir.
   * Eğer dosya yoksa, ekrana uyarı mesajı yazdırılır ve None döner.
   * Dosya varsa, load\_data fonksiyonu çağrılarak X\_train, X\_test, y\_train, y\_test ayrılır.
2. **Model Oluşturma**
   * BoltzmannSpikingNeuralNetwork sınıfından bir nesne (snn\_model) örneklendirilir.
   * input\_size=X\_train.shape[1]: Girdi boyutu, veri setindeki sütun sayısı kadar (örneğin 784).
   * hidden\_size=500: Sabit olarak 500 gizli nöron seçilmiştir.
   * output\_size=10: Sınıf sayısı (örn. 10 rakam).
3. **Model Eğitimi**
   * num\_epochs = 5: Eğitim beş epok boyunca yapılacak.
   * snn\_model.train(...): Eğitim metodu çağrılır. Doğrulama seti verilmediği için X\_val=None, y\_val=None olarak bırakılmıştır.
   * Eğitim boyunca her epoch sonunda elde edilen doğruluk değerleri accuracies listesinde saklanır.
4. **Test Aşaması**
   * test\_preds, test\_confs = snn\_model.predict(X\_test): Eğitilmiş model kullanılarak test verisi üzerinde tahminler ve her tahminin güven değeri alınır.
5. **Sonuçların Yazdırılması ve mainin Dönüş Değeri**
   * accuracy\_score(y\_test, test\_preds): Test verisi üzerindeki doğruluk oranı hesaplanır ve ekrana yazdırılır.
   * classification\_report(y\_test, test\_preds): Test seti için sınıflandırma raporu (precision, recall, f1-score vb.) ekrana yazdırılır.
   * return snn\_model, accuracies, X\_test, test\_preds, test\_confs: Model nesnesi, doğruluk listesi ve test sonuçları döndürülür.

**9. \_\_main\_\_ Bloğu (Programın Çalıştırılması)**

if \_\_name\_\_ == '\_\_main\_\_':

import time

start\_time = time.time()

snn\_model, accuracies, \_, test\_preds, test\_confs = main()

end\_time = time.time()

total\_time = end\_time - start\_time

print(f"\nToplam Çalışma Süresi: {total\_time:.2f} saniye")

if accuracies and len(accuracies) > 0:

print(f"Epoch başına ortalama süre: {total\_time/len(accuracies):.2f} saniye")

print(f"Son Eğitim Doğruluğu: {accuracies[-1]:.1%}")

if snn\_model:

print(f"Son Boltzmann Sıcaklığı: {snn\_model.temperature:.4f}")

if test\_confs and len(test\_confs) > 0:

high\_conf\_preds = np.sum(np.array(test\_confs) > 0.8)

print(f"Yüksek Güvenli Test Tahminleri (>0.8): {high\_conf\_preds}/{len(test\_confs)}")

Bu blok, Python dosyası doğrudan çalıştırıldığında (python LIF\_MODEL.py) yürütülür:

1. **Zaman Ölçümünün Başlatılması**
2. import time
3. start\_time = time.time()
   * time.time() ile script başlangıcındaki sistem zamanını saniye cinsinden alır.
4. **main Fonksiyonunun Çağrılması**
5. snn\_model, accuracies, \_, test\_preds, test\_confs = main()
   * main metodu çağrılır ve dönüş değerleri snn\_model, accuracies, test\_preds, test\_confs gibi değişkenlere atanır. Burada test verisi (\_) kullanılmayacağı için underscore (\_) kullanılmıştır.
6. **Zaman Ölçümünün Sonlandırılması ve Sonuçların Yazdırılması**
7. end\_time = time.time()
8. total\_time = end\_time - start\_time
9. print(f"\nToplam Çalışma Süresi: {total\_time:.2f} saniye")
   * İşlem tamamlandığında, bitiş zamanı alınır ve start\_time farkı hesaplanarak toplam çalışma süresi hesaplanır.
   * Bu süre, ekranda saniye cinsinden gösterilir.
10. if accuracies and len(accuracies) > 0:
11. print(f"Epoch başına ortalama süre: {total\_time/len(accuracies):.2f} saniye")
12. print(f"Son Eğitim Doğruluğu: {accuracies[-1]:.1%}")
    * Eğer accuracies listesi boş değilse (yani en az bir eğitim döngüsü tamamlanmışsa):
      + total\_time/len(accuracies): Toplam süre, epoch sayısına bölünerek ortalama epoch süresi hesaplanır.
      + accuracies[-1]: Listenin son elemanı (son epoch doğruluğu) ekrana yüzde formatında yazdırılır.
13. if snn\_model:
14. print(f"Son Boltzmann Sıcaklığı: {snn\_model.temperature:.4f}")
    * Eğitim sonunda Boltzmann makinesinin sıcaklığı (temperature) ekrana yazdırılır.
15. if test\_confs and len(test\_confs) > 0:
16. high\_conf\_preds = np.sum(np.array(test\_confs) > 0.8)
17. print(f"Yüksek Güvenli Test Tahminleri (>0.8): {high\_conf\_preds}/{len(test\_confs)}")
    * Test tahminleri içinden, güven değeri (confidence) 0.8'den yüksek olan tahminler sayılır.
    * Bu yüksek güvenli tahminlerin sayısı, toplam test örneği sayısına oranla ekrana yazdırılır.

**Ek Açıklamalar ve Kavramsal Özet**

1. **LIF (Leaky Integrate-and-Fire) Modeli**
   * Gerçek biyolojik nöronları taklit eden bir modeldir.
   * Membran potansiyeli zamanla dışarı kaçar (leaky), fakat gelen girdilerle tekrar yükselir (integrate).
   * Potansiyel eşik değerini (v\_thresh) aştığında nöron "ateşler" ("fire").
   * Ateşledikten sonra membran potansiyeli sıfırlanır (v\_reset).
   * Bu mekanizma, verinin zaman içinde spike (elektrik darbeleri) dizisine dönüştürülmesini sağlar.
2. **STDP (Spike-Timing Dependent Plasticity)**
   * Sinaptik ağırlıkların, pre ve post nöron spike zamanlamasına göre adapte olduğu bir öğrenme mekanizmasıdır.
   * Pre-spike (önce gelen spike) ile post-spike (sonra gelen spike) arasındaki zamana bağlı olarak sinaptik potansiyasyon veya depresyon gerçekleşir.
   * A\_plus, A\_minus, tau\_plus, tau\_minus gibi parametreler, bu zaman bağımlı değişimin ne kadar hızlı veya yavaş olacağını belirler.
   * Kodda hem hızlı (fast) hem de yavaş (slow) izler tutulur: Bu, kısa süreli belleği ve uzun süreli izleri farklı biçimlerde değerlendirmeyi sağlar.
3. **Boltzmann Makinesi Entegrasyonu**
   * Sinir ağına olasılıksal bir karar mekanizması eklemek için kullanılır.
   * Gizli katman spikeleri softmax fonksiyonundan geçirilerek olasılıklara çevrilir.
   * Model, temperature parametresi ile o anki sıcaklığa göre olasılık dağılımı oluşturur.
   * Eğitim sırasında sıcaklık kademeli olarak azaltılarak (temp\_decay) karar mekanizması daha keskin hale getirilir.
   * Basit bir contrastive divergence yaklaşımı kullanılarak gizli-çıktı ağırlıkları, hata (target - probs) kullanılarak güncellenir.
4. **Model Mimarisinin Genel Özeti**
   * Girdi katmanı:
     + Ham piksel verisi, [0,1] aralığına ölçeklendirilir.
     + Her zaman adımında LIF nöronları spike üretir.
   * Gizli katman:
     + Girdi spike dizileri, W\_input\_hidden ağırlıklarıyla çarpılarak membran potansiyel hesaplaması yapılır.
     + LIF mekanizması ile gizli katman spike dizisi üretilir.
     + STDP izleri bu adımda güncellenir.
   * Çıktı katman:
     + Gizli katman spike dizileri, W\_hidden\_output ağırlıklarıyla çarpılarak membran potansiyel hesaplaması yapılır.
     + LIF mekanizması ile çıktı katman spike dizisi üretilir.
     + STDP izleri bu adımda güncellenir.
   * Boltzmann adımı:
     + Gizli katmanda son adımda elde edilen spikeler, softmax ve sıcaklık parametresi kullanılarak çıktı olasılıklarına dönüştürülür.
     + Hata, one-hot etiketler ile olasılıklar arasındaki farktan hesaplanır.
     + Gizli-çıktı ağırlıkları güncellenir.
5. **Eğitim ve Değerlendirme**
   * Eğitim sırasında minibatch yaklaşımı kullanılır. Her minibatch'te:
     + Spike dizisi oluşturulur.
     + Gizli ve çıktı katmanlarındaki LIF simülasyonu yapılır.
     + STDP tabanlı sinaptik güncellemeler gerçekleştirilir.
     + Boltzmann hata tabanlı ağırlık güncellemesi yapılır.
     + Eğitim doğruluğu hesaplanarak izlenir.
   * Doğrulama verisi sağlanmışsa, her epoch sonunda doğrulama doğruluğu hesaplanır.
   * Test sırasında eğitilmiş ağırlıklar kullanılarak overfitting kontrolü yapılır.

**Sonuç**

Bu belge, LIF\_MODEL.py dosyasındaki kodun baştan sona detaylı bir incelemesini içermektedir. Kütüphane importlarından sınıf tanımına, her bir metot içeriğinden eğitim ve test aşamalarına kadar her adım ayrıntılı olarak açıklanmıştır.