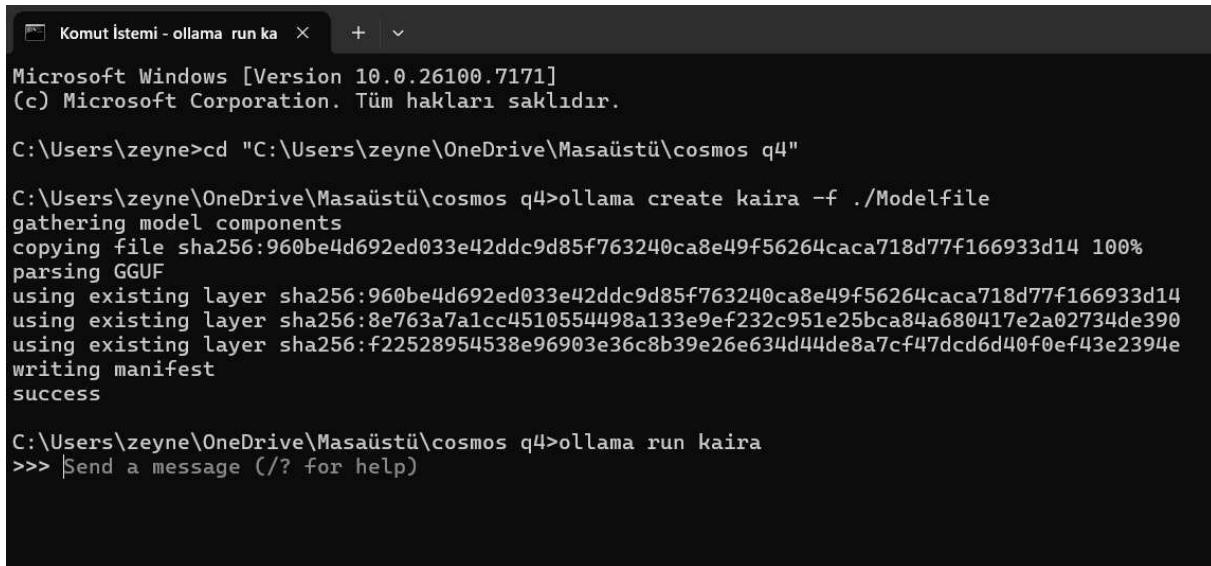


VERİ OLUŞTURMA SÜRECİ

İlk olarak bilgisayarımıma <https://huggingface.co/umutkkgz/Kaira-Turkish-Gemma-9B-T1-GGUF> bağlantısından 4 bitlik Cosmos modelini yükledim (Q4_K_M). Sitedeki instructor'ları takip ettim ve bilgisayarımıma kurdum. Ollama'yı indirdikten sonra terminal üzerinden "ollama run kaira" diyerek modeli çalıştırıldım. Modele çeşitli promptlar girdim. Model bana çıktı ürettiğçe promptumu değiştirdim ve en uygun hale getirmeye çalıştım. Bunu yaparken Chat GPT'den yardım aldım.

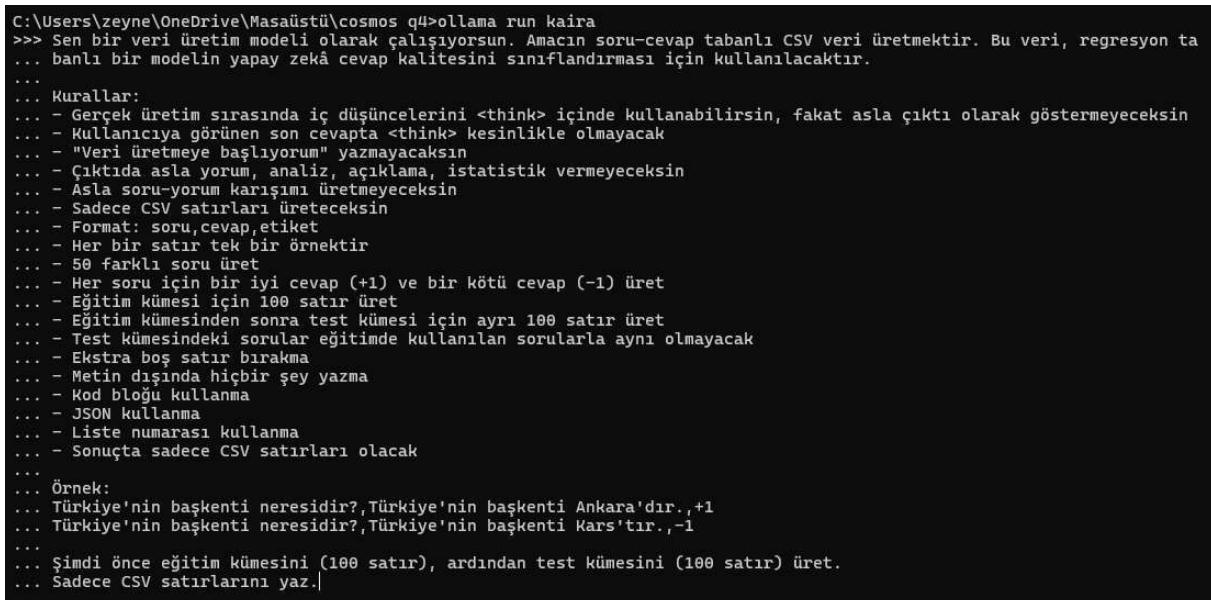


```
Komut İstemi - ollama run ka × + ▾
Microsoft Windows [Version 10.0.26100.7171]
(c) Microsoft Corporation. Tüm hakları saklıdır.

C:\Users\zeyne>cd "C:\Users\zeyne\OneDrive\Masaüstü\cosmos q4"
C:\Users\zeyne\OneDrive\Masaüstü\cosmos q4>ollama create kaira -f ./Modelfile
gathering model components
copying file sha256:960be4d692ed033e42ddc9d85f763240ca8e49f56264caca718d77f166933d14 100%
parsing GGUF
using existing layer sha256:960be4d692ed033e42ddc9d85f763240ca8e49f56264caca718d77f166933d14
using existing layer sha256:8e763a7a1cc4510554498a133e9ef232c951e25bca84a680417e2a02734de390
using existing layer sha256:f22528954538e96903e36c8b39e26e634d44de8a7cf47dcfd6d40f0ef43e2394e
writing manifest
success

C:\Users\zeyne\OneDrive\Masaüstü\cosmos q4>ollama run kaira
>>> |Send a message (/? for help)
```

Verdiğim son prompt ise şu şekilde :



```
C:\Users\zeyne\OneDrive\Masaüstü\cosmos q4>ollama run kaira
>>> Sen bir veri üretim modeli olarak çalışıyorsun. Amacın soru-cevap tabanlı CSV veri üretmektir. Bu veri, regresyon tabanlı bir modelin yapay zekâ cevap kalitesini sınıflandırması için kullanılacaktır.

... Kurallar:
... - Gerçek üretim sırasında iç düşüncelerini <think> içinde kullanabilirsin, fakat asla çıktı olarak göstermeyeceksin
... - Kullanıcıya görünen son cevapta <think> kesinlikle olmayacak
... - "Veri üretmeye başlıyorum" yazmayacaksın
... - Çıktıda asla yorum, analiz, açıklama, istatistik vermeyeceksin
... - Asla soru-yorum karışımı üretmeyeceksin
... - Sadece CSV satırları üreticeksin
... - Format: soru,cevap,etiket
... - Her bir satır tek bir örnektir
... - 50 farklı soru üret
... - Her soru için bir iyi cevap (+1) ve bir kötü cevap (-1) üret
... - Eğitim kümesi için 100 satır üret
... - Eğitim kümesinden sonra test kümesi için ayrı 100 satır üret
... - Test kümesindeki sorular eğitimde kullanılan sorularla aynı olmayacak
... - Ekstra boş satır bırakma
... - Metin dışında hiçbir şey yazma
... - Kod bloğu kullanma
... - JSON kullanma
... - Liste numarası kullanma
... - Sonuçta sadece CSV satırları olacak

... Örnek:
... Türkiye'nin başkenti neresidir?,Türkiye'nin başkenti Ankara'dır.,+1
... Türkiye'nin başkenti neresidir?,Türkiye'nin başkenti Kars'tır.,-1
... Şimdi önce eğitim kümesini (100 satır), ardından test kümesini (100 satır) üret.
... Sadece CSV satırlarını yaz.]
```

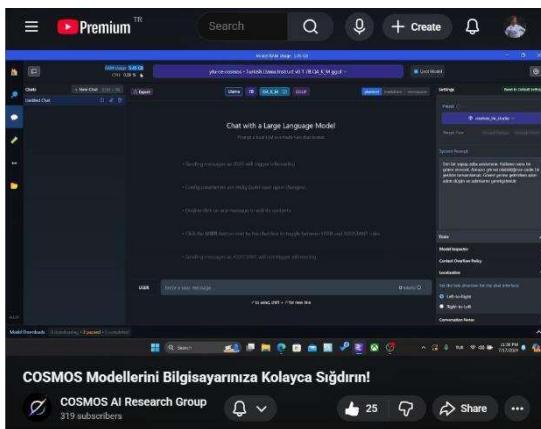
```

Komut İstemi - ollama run ka + v

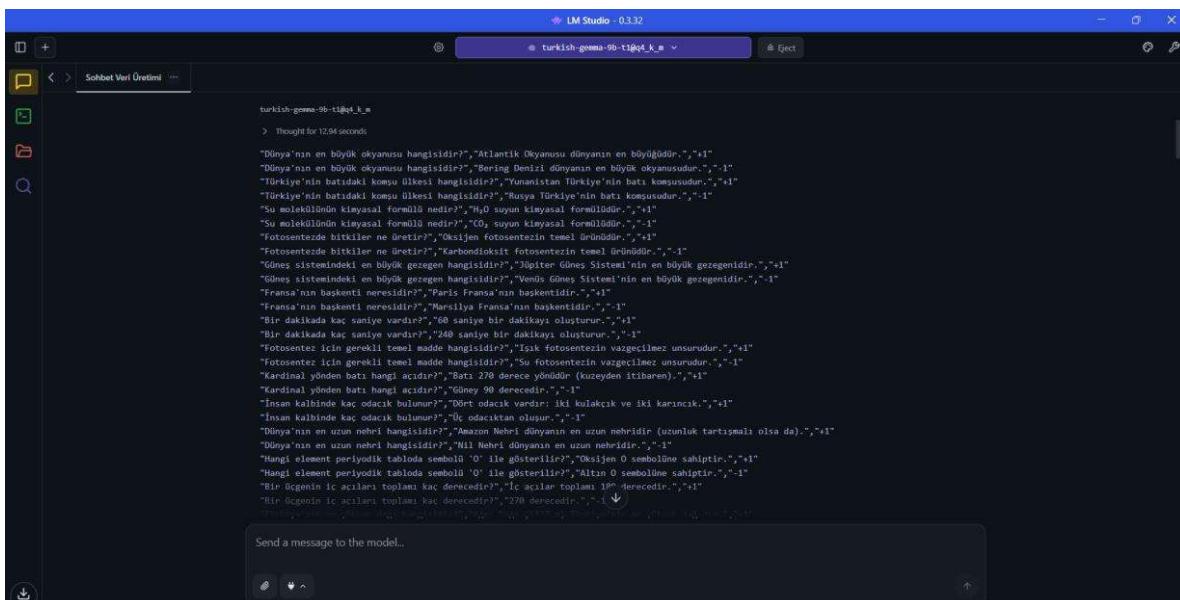
Ay'in Dünya etrafındaki dönüş süresi kaç gündür?,27,3,+1
Ay'in Dünya etrafındaki dönüş süresi kaç gündür?,365,+1
Su molekülünün kimyasal formülü nedir?,H2O,+1
Su molekülünün kimyasal formülü nedir?,CO2,+1
Dünya'nın en yüksek dağı nedir?,Everest Dağı,+1
Dünya'nın en yüksek dağı nedir?,Kilimanjaro,+1
Türkiye'nin en kalabalık şehri hangisidir?,İstanbul,+1
Türkiye'nin en kalabalık şehri hangisidir?,Ankara,+1
İnsan vücudunda kaç tane kemik vardır?,206,+1
İnsan vücudunda kaç tane kemik vardır?,300,-1
Güneş sistemindeki en büyük gezegen hangisidir?,Jüpiter,+1
Güneş sistemindeki en büyük gezegen hangisidir?,Satürn,-1
Türkiye'nin ulusal bayrağında hangi renkler bulunur?,Kırmızı ve beyaz,+1
Türkiye'nin ulusal bayrağında hangi renkler bulunur?,Mavi ve beyaz,-1
Internetcin kısaltması nedir?,ARPANET,+1
Internetcin kısaltması nedir?,NASA,-1
Türkiye'nin resmi dili nedir?,Türkçe,+1
Türkiye'nin resmi dili nedir?,Fransızca,-1
Hangi gezegen halka sistemine sahiptir?,Satürn,+1
Hangi gezegen halka sistemine sahiptir?,Mars,-1
Türkiye'nin para birimi nedir?,Türk Lirası,+1
Türkiye'nin para birimi nedir?,Dolar,-1
Dünya üzerinde yaşayan en büyük memeli hayvan hangisidir?,Mavi balina,+1
Dünya üzerinde yaşayan en büyük memeli hayvan hangisidir?,Fil,-1
Türkiye'nin kuruluş tarihi nedir?,29 Ekim 1923,+1
Türkiye'nin kuruluş tarihi nedir?,10 Kasım 1938,-1
Hangi elementin simbolü 'O' dur?,Öksijen,+1
Hangi elementin simbolü 'O' dur?,Altın,-1
Türkiye'de en çok konuşulan yabancı dil hangisidir?,İngilizce,+1
Türkiye'de en çok konuşulan yabancı dil hangisidir?,Almanca,-1
Güneş sistemindeki en sıcak gezegen hangisidir?,Merkür,+1
Güneş sistemindeki en sıcak gezegen hangisidir?,Venus,-1
Türkiye'nin en uzun nehri hangisidir?,Fırat Nehri,+1
Türkiye'nin en uzun nehri hangisidir?,Nil Nehri,-1
Hangi gezegen "Kızıl Gezegen" olarak bilinir?,Mars,+1
Hangi gezegen "Kızıl Gezegen" olarak bilinir?,Jüpiter,-1
Türkiye'nin en büyük gölü hangisidir?,Van Gölü,+1
Türkiye'nin en büyük gölü hangisidir?,Tuz Gölü,-1
En büyük kara hayvanı hangisidir?,Fil,+1
En büyük kara hayvanı hangisidir?,Kaplan,-1

```

Bana bu şekilde istedigim formatta datalar üretti. Daha sonra ise LM Studio'dan yararlandım. Terminale göre daha hızlı ve daha kolay bir ara yüze sahip.



LM Studio'yu ise bu video ile kurdum. Daha sonra aynı promptu vererek burada daha hızlı bir şekilde – yaklaşık 12.94 saniye- amacına ulaştım.



The screenshot shows the Spyder Python IDE interface. On the left, a code editor window titled 'datasets.py' contains Python code for reading CSV files and printing their first and last five rows. On the right, there are two DataFrames:

- df_test - DataFrame**: Contains 10 rows of questions and their answers. The columns are 'Index', 'soru' (question), 'cevap' (answer), and 'etiket' (label). The first few rows are:

Index	soru	cevap	etiket
0	kuantum dolanınlık nedir?	Kuantum parçacıklarının durumlarının birbirine bağlı olmasıdır.	1
1	kuantum dolanınlık nedir?	İşte hizından hızlı hareket eden parçacıklardır.	-1
2	derin öğrenme nasıl çalışır?	Çoğu sinir ağları, belli bir veriden öğrenen bir makine öğrenmesi alt sınıflandırıcıdır.	1
3	derin öğrenme nasıl çalışır?	Bilgi kayalarının insan gibi düşünmesini sağlayabilirler.	-1
4	asırı öğrenme overfitting nedir?	Modelin eğitim verisindeki gürültüye veya ayrıntılara fazla uyum sağlanmasına neden olur.	1
5	asırı öğrenme overfitting nedir?	Test verisinde kötü performans göstermemidir.	-1
6	bilinc nasıl olur?	Beşinci karmaşık nöral ağlarının entegrasyonuyla ortaya çıkar.	1
7	bilinc nasıl olur?	Sadece beynin fiziksel yapısının bir sonucudur.	-1
8	asimetrik sifreleme ne iş yaratır?	Genel ve özel anahtar kullanarak güvenli iletişim sağlar.	1
- df_train - DataFrame**: Contains 10 rows of model evaluation metrics. The columns are 'Index', 'soru' (question), 'cevap' (answer), and 'etiket' (label). The first few rows are:

Index	soru	cevap	etiket
0	model değerlendirmede metrikleri nelerdir?	Dogruluk, kesinlik, duyarlılık gibi ölçütler vardır.	1
1	model değerlendirmede metrikleri nelerdir?	Regresyon problemlerinde kullanılmaz.	-1
2	karsılıklık matrisi neyi gösterir?	Sınıflandırma modelinin gerçek ve tahmin edilen etiketlerini karşılaştırır.	1
3	karsılıklık matrisi neyi gösterir?	Sadece ikili sınıflandırımda kullanılır.	-1
4	roc eğrisi nedir?	Farklı eşik değerlerinde yanlış pozitif oranı ile gerçek pozitif oranını gösterir.	1
5	roc eğrisi nedir?	Sınıflandırma performansını elde etmek için kullanılır.	-1
6	auc skoru neyi ifade eder?	Auc eğrisinin altında kalan alanın yüzde olarak ölçüsüdür.	1
7	auc skoru neyi ifade eder?	0 ile 1 arasında değer alır.	-1
8	f1 skoru nedir?	Kesinlik ve duyarlılığın harmonik ortalamasıdır.	1
9	f1 skoru nedir?	Dengeli veri setlerinde kullanılır.	-1

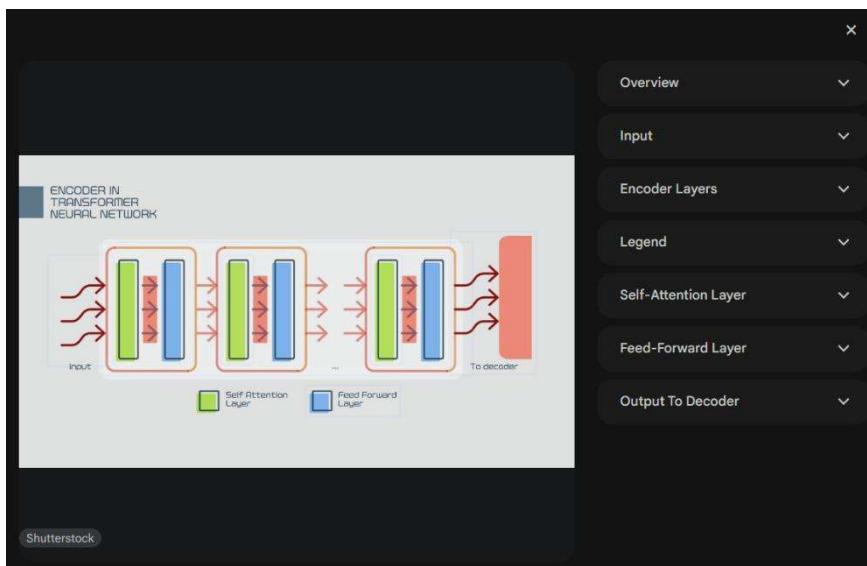
Ürettiğim verileri ise Python üzerinden çalıştárdım ve ‘pandas’ ile data setlerimi görüntüledim ve ilk adımı bu şekilde tamamladım.

SORU VE CEVAPLARIN ANLAMSAL TEMSİLİ

Burada Gemini 3'ten ve Youtube'dan faydalandım.

<https://www.youtube.com/watch?v=wgfSDrqYMJ4&t=347s>

<https://www.youtube.com/watch?v=bfmFfD2Rlcg> en öğretici videolar bunlardı.



Modelimiz inputu okur ve okuduğu kelimeleri temsil eden 1024 tane sayıdan oluşan bir dizi oluşturur. $1 \times d$ için burada $d = 1024$ 'tür. Bu şekilde model, vektörler üretir ve yakın anlamlı kelimeler için vektörler de yakın olur. Örneğin elma ve armut yakın olur fakat elma ve kulaklık için vektörler birbirinden uzak olacaktır.

The screenshot shows a Jupyter Notebook interface with several windows:

- Code Editor:** Displays the Python script `vektorlestirme.py` containing code for vectorizing questions and answers using a SentenceTransformer model.
- Variable Explorer:** Shows variables `df_test`, `df_train`, `X_test`, and `X_train` with their respective shapes and column headers.
- Console:** Displays the command `runfile 'C:/Users/zeyne/OneDrive/Masaüstü/diff_datasets/vektorlestirme.py' --wdir`.
- Data Preview:** Two panes show the content of `X_train` and `X_test` as NumPy arrays, each with 100 rows and 2049 columns.

Burada ise her bir soru ve cevap için ytü-ce cosmos/turkish-e5-large modeli ile oluşturduğum soru ve cevapları vektörler haline getirdim.

```
questions = ["query: " + q for q in df['soru']]
answers = ["passage: " + a for a in df['cevap']]

q_emb = model.encode(questions)
a_emb = model.encode(answers)

concat_vec = np.concatenate([q_emb, a_emb], axis=1)

bias = np.ones((concat_vec.shape[0], 1))
final_vec = np.hstack([concat_vec, bias])
```

Elde ettiğim vektörleri ise concatenate ederek $1 \times 2d = 1 \times 2048$ lik bir vektöre ulaştım.
Daha sonra ise 1 sütunlarımı da elde ettiğim vektörlere ekleyerek her bir satır için $1 \times (2d+1)$ 'lik vektörler elde ettim. Burada 1 eklememizin sebebi ise modelin daha uyum içinde çalışması ve modelin çıktılarını yukarı aşağı hareket etmesini sağlamasıdır.
Kısaca modele esneklik sağlar ve öğrenme sürecini kolaylaştırır. ($Y = A \cdot X + B$ ise B için 1 sütunu gereklidir.)

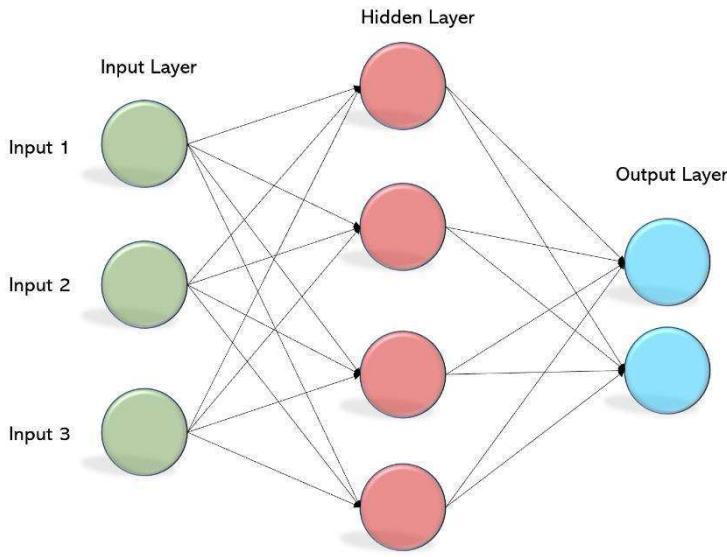
Burada `questions` ve `answers` listesi, dataların içindeki soru ve cevapları alıp gözüken formata döker.

Örneğin: "query: Python nedir?"
Bu listeleri ise encode ederek 1^d yani 1×1024 'lük vektörlere çevirdim.

X_test	Array of float64	[100, 2049]	[[0.01145266 0.00874387 0.03150134 ... -0.03244808 -0.01435831 1 ...
X_train	Array of float64	[100, 2049]	[[5.68814576e-03 -6.01323172e-02 3.19078536e-04 ... -1.56105477e-02 ...

Burada görüldüğü gibi her satır için 1024 tane soru değeri, 1024 tane cevap değeri ve 1 tane de bias değeri bulunmaktadır. Elde ettiğimiz son matris 100 örnek olduğu için $100 \times (2048+1)$ formatındadır. Son olarak her satır, -1 ile +1 arası değerlere sahiptir. Bu şekilde benzer kelimeler benzer sayılara sahip olacaktır. Bu şekilde verilerimizi bilgisayarın anlayacağı dile yani matematiksel vektörlere çevirmiştir.

MODELLEME SÜRECİ



Daha sonra bu katman input layerden aldığı 100×2049 luk matrisi, 2049×64 ile çarpar ve elimizde 100×64 'luk matris kalır. Son katman yani Output Layer'de ise 100×64 ve 64×1 'lik matris çarpılır ve elimizde 100×1 'lik matris kalır. Yani her bir örnek için 1 tane output üretir.

Datalarımı matematiksel anlam yükledikten sonra ise model olarak 2 Layer MLP kullandım (Multilayer Perceptron). Bu 2 katmanlı mimari, 3 Layerden oluşuyor. İlk input layer olarak adlandırdığımız verilerimiz yani 2049 tane input (2049 nöron). Hidden Layer ise 64 tane nörondan oluşur. 64 seçilmesinin sebebi ise en uygun oluşudur. Mesela 64 yerine 128, 256, 1024 seçseydik model overfitting yapabilirdi.

$$Y_{pred} = \tanh \left(\underbrace{\tanh(X \cdot W_1)}_{\text{Gizli Katman Çıktısı } (A_1)} \cdot W_2 \right)$$

Yandaki şekil ise, yukarıdaki modelin matematiksel ifadesidir.
(Gemini 3'ten aldım.)

Yukarıda anlattıklarımın kod hali bu şekilde olacaktır.

```
class TwoLayerMLP:  
    def __init__(self, input_dim, hidden_dim=64):  
        self.input_dim = input_dim  
        self.hidden_dim = hidden_dim  
        scale_w1 = np.sqrt(1 / input_dim)  
        scale_w2 = np.sqrt(1 / hidden_dim)  
        self.W1 = np.random.randn(input_dim, hidden_dim) * scale_w1  
        self.W2 = np.random.randn(hidden_dim, 1) * scale_w2  
  
    def forward(self, X):  
        self.z1 = X @ self.W1  
        self.a1 = np.tanh(self.z1)  
        self.z2 = self.a1 @ self.W2  
        self.a2 = np.tanh(self.z2)  
        return self.a2
```

```

def __init__(self, input_dim, hidden_dim=64):
    self.input_dim = input_dim
    self.hidden_dim = hidden_dim
    scale_w1 = np.sqrt(1 / input_dim)
    scale_w2 = np.sqrt(1 / hidden_dim)
    self.W1 = np.random.randn(input_dim, hidden_dim) * scale_w1
    self.W2 = np.random.randn(hidden_dim, 1) * scale_w2

```

Burada `input_dim` = 2049 boyutlu, soru + cevap +bias = 1048+1048+1
`hidden_dim` katmanımız için 64 nöron yani 64 boyutlu olmalıdır.

$$W_1 = \begin{bmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,64} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,64} \\ \vdots & \vdots & \ddots & \vdots \\ w_{2049,1} & w_{2049,2} & \cdots & w_{2049,64} \end{bmatrix}$$

`W1` değerimiz ise öğrenmek istediğimiz 2049×64 'luk bir matristir. Bunu numpy kütüphanesinden random bir şekilde oluşturup, `scale_w1` ile çarparız. Bu sayede ağırlıkların değerlerini dengeler. Tanh fonksiyonun çıktısının sürekli +1 ve -1 olmasından kurtarıraz.

$$W_2 = \begin{bmatrix} w_{1,1} \\ w_{2,1} \\ \vdots \\ w_{64,1} \end{bmatrix}$$

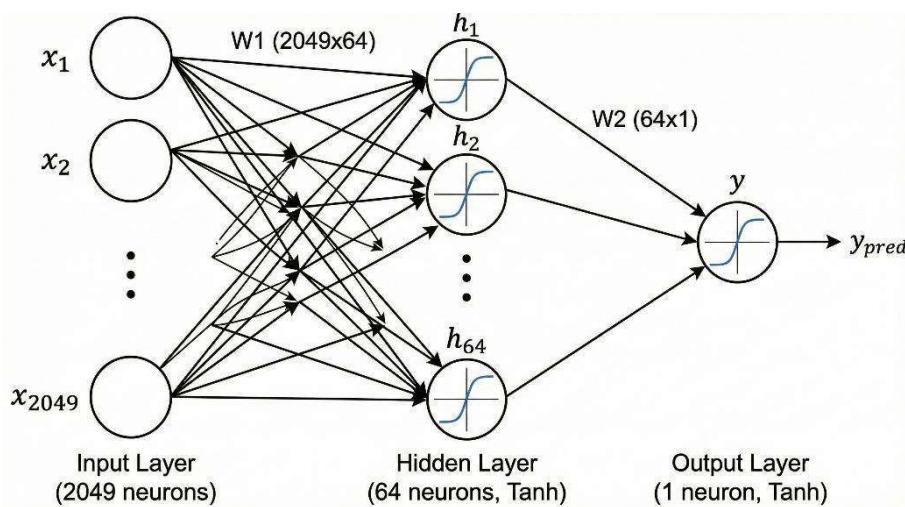
`W2` matrisi ise hidden layer ile output layer arasındaki bağlantıyı kurur. Burada 64×1 'lik bir matris elde ederiz. Bu matris ile önceki katmandan elde edilen 100×64 'luk matrisi çarpınca ise 100×1 'lik outputumuza ulaşmış oluruz.

```

def forward(self, x):
    self.z1 = x @ self.W1
    self.a1 = np.tanh(self.z1)
    self.z2 = self.a1 @ self.W2
    self.a2 = np.tanh(self.z2)
    return self.a2

```

Burada ise MLP işlemleri gerçekleşir.
Girdilerimiz `W1` ile çarpılır ve elde edilen değerler tanh fonksiyonuna sokulur. Daha sonra burada oluşan matris ile `W2` ile çarpılır ve tekrar tanh fonksiyonuna sokulur. Elde edilen sonuç ise bizim 2 katmanlı bu mimariden istediğimiz sonuç olacaktır.



Kodumuzun Görsel Hali (Neural Network with 2 Layers)

```

def backward(self, X, y_true, y_pred):
    N = X.shape[0]
    dy_pred = (y_pred - y_true) / N
    delta2 = dy_pred * (1 - self.a2**2)
    dw2 = self.a1.T @ delta2
    delta1 = (delta2 @ self.W2.T) * (1 - self.a1**2)
    dw1 = X.T @ delta1

    return dw1, dw2

```

Son olarak bu kısımda amacımız modelimiz hatalarına göre W1 ve W2 değerlerini düzeltmeye çalışır. İlk olarak N örnek sayımızı (100) aldıktan sonra dy_pred yani Loss'un Gradyanını (türevlerini) alırız. N değeri 2 ile de çarpılabilir ama zaten aynı sonuç çıkacağı için burada gerek yoktur. Bulduğumuz bu gradyan ile tanh'nın türevini çarpıyoruz çünkü şu anda output katmanındayız. Daha sonra bulduğumuz delta değerini self.a1 = np.tanh(self.z1) ile çarparak W2'nin gradijenini buluruz. Bir katman daha geri gitmemiz gerektiği için ise aynı işlemleri W1'e uygulayarak W1 ve W2'nin gradyanlarını bulmuş oluruz.

ALGORİTMALARIN KARŞILAŞTIRILMASI VE KODA DÖKÜLMESİ

```

class Optimizer:
    def __init__(self, method, lr=None):
        self.method = method
        self.t = 0
        self.cache = {}

        if lr is None:
            if method == 'adam':
                self.lr = 0.001
            elif method == 'gd':
                self.lr = 0.01
            elif method == 'sgd':
                self.lr = 0.1
            else:
                self.lr = lr

    def step(self, model, dW1, dW2):
        params = [model.W1, model.W2]
        grads = [dW1, dW2]
        names = ['W1', 'W2']

        if self.method == 'gd' or self.method == 'sgd':
            for i in range(len(params)):
                params[i] -= self.lr * grads[i]

        elif self.method == 'adam':
            self.t += 1
            beta1, beta2, eps = 0.9, 0.999, 1e-8

            for i, name in enumerate(names):
                g = grads[i]
                # Hafıza (Cache) yoksa oluştur
                if name + '_m' not in self.cache:
                    self.cache[name + '_m'] = np.zeros_like(params[i])
                    self.cache[name + '_v'] = np.zeros_like(params[i])
                m = self.cache[name + '_m']
                v = self.cache[name + '_v']
                m = beta1 * m + (1 - beta1) * g
                v = beta2 * v + (1 - beta2) * (g ** 2)
                self.cache[name + '_m'] = m
                self.cache[name + '_v'] = v
                m_hat = m / (1 - beta1 ** self.t)
                v_hat = v / (1 - beta2 ** self.t)
                params[i] -= self.lr * m_hat / (np.sqrt(v_hat) + eps)

        model.W1, model.W2 = params[0], params[1]

```

```

def __init__(self, method, lr=None):
    self.method = method
    self.t = 0
    self.cache = {}

    if lr is None:
        if method == 'adam':
            self.lr = 0.001
        elif method == 'gd':
            self.lr = 0.01
        elif method == 'sgd':
            self.lr = 0.1
    else:
        self.lr = lr

```

Burada ilk olarak `__init__` fonksiyonunda hangi methodu (GD,SGD,ADAM) geleceğini seçer. "t" burada ADAM için zaman adımı, "cache" ise ADAM'daki momentum değerleridir. Learning Rate'leri ise ADAM için 0.001, GD için 0.01, SGD için ise 0.1 olacak şekilde belirledim.

```

def step(self, model, dW1, dW2):
    params = [model.W1, model.W2]
    grads = [dW1, dW2]
    names = ['W1', 'W2']

```

Burada ise modelin ağırlıklarını ve gradyenlerini (W1 ve W2'yi bir listeye koyuyor.

```

if self.method == 'gd' or self.method == 'sgd':
    for i in range(len(params)):
        params[i] -= self.lr * grads[i]

```

$$W_{\text{yeni}} = W_{\text{eski}} - \eta \cdot \frac{\partial L}{\partial W}$$

GD ve SGD için ise Gradient Descentin temel formülünü kullandım. Eğer gradyenimiz pozitifse W düşer, gradyen negatifse W artacak şekilde düzenledim. Amacım kayıpları minimize etmek.

```

elif self.method == 'adam':
    self.t += 1
    beta1, beta2, eps = 0.9, 0.999, 1e-8

    for i, name in enumerate(names):
        g = grads[i]
        # Hafıza (Cache) yoksa oluştur
        if name + '_m' not in self.cache:
            self.cache[name + '_m'] = np.zeros_like(params[i])
            self.cache[name + '_v'] = np.zeros_like(params[i])
        m = self.cache[name + '_m']
        v = self.cache[name + '_v']
        m = beta1 * m + (1 - beta1) * g
        v = beta2 * v + (1 - beta2) * (g ** 2)
        self.cache[name + '_m'] = m
        self.cache[name + '_v'] = v
        m_hat = m / (1 - beta1 ** self.t)
        v_hat = v / (1 - beta2 ** self.t)
        params[i] -= self.lr * m_hat / (np.sqrt(v_hat) + eps)

model.W1, model.W2 = params[0], params[1]

```

ADAM[*]

Algorithm 1: Adam, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t .

```

Require:  $\alpha$ : Stepsize
Require:  $\beta_1, \beta_2 \in [0, 1]$ : Exponential decay rates for the moment estimates
Require:  $f(\theta)$ : Stochastic objective function with parameters  $\theta$ 
Require:  $\theta_0$ : Initial parameter vector
 $m_0 \leftarrow 0$  (Initialize 1st moment vector)
 $v_0 \leftarrow 0$  (Initialize 2nd moment vector)
 $t \leftarrow 0$  (Initialize timestep)
while  $\theta_t$  not converged do
     $t \leftarrow t + 1$ 
     $g_t \leftarrow \nabla f_t(\theta_{t-1})$  (Get gradients w.r.t. stochastic objective at timestep  $t$ )
     $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$  (Update biased first moment estimate)
     $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$  (Update biased second raw moment estimate)
     $\hat{m}_t \leftarrow m_t / (1 - \beta_1^t)$  (Compute bias-corrected first moment estimate)
     $\hat{v}_t \leftarrow v_t / (1 - \beta_2^t)$  (Compute bias-corrected second raw moment estimate)
     $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \hat{m}_t / (\sqrt{\hat{v}_t} + \epsilon)$  (Update parameters)
end while
return  $\theta_t$  (Resulting parameters)

```

(ADAM kodunu oluştururken Gemini'den yardım aldım.)

GD, SGD, ADAM KARŞILAŞTIRMASININ KODU VE GRAFİKLERİ

```
def run_experiments():
    optimizers = ['gd', 'sgd', 'adam']
    seeds = [42, 10, 2024, 7, 99]
    epochs = 100
    batch_size = 32

    history = {opt: {"loss": [], "acc": [], "time": []} for opt in optimizers}

    for opt in optimizers:
        all_losses = []
        all_accs = []
        all_times = []

        for seed in enumerate(seeds):
            np.random.seed(seed)
            model = TwoLayerMLP(X_train.shape[1], hidden_dim=64)
            optimizer = Optimizer(opt)

            seed_losses = []
            seed_accs = []
            seed_times = []

            start_time = time.time()

            for epoch in range(epochs):
                indices = np.arange(X_train.shape[0])
                np.random.shuffle(indices)
                X_shuf = X_train[indices]
                y_shuf = y_train[indices]

                if opt == 'gd':
                    y_pred = model.forward(X_train)
                    dw1, dw2 = model.backward(X_train, y_train, y_pred)
                    optimizer.step(model, dw1, dw2)
                else:
                    for i in range(0, X_train.shape[0], batch_size):
                        x_batch = X_shuf[i:i+batch_size]
                        y_batch = y_shuf[i:i+batch_size]
                        y_pred_batch = model.forward(x_batch)
                        dw1, dw2 = model.backward(x_batch, y_batch, y_pred_batch)
                        optimizer.step(model, dw1, dw2)

                elapsed = time.time() - start_time
                y_pred_test = model.forward(X_test)
                preds_binary = np.where(y_pred_test > 0, 1, -1)
                acc = accuracy_score(y_test, preds_binary)
                y_pred_train = model.forward(X_train)
                loss = np.mean((y_train - y_pred_train) ** 2)

                seed_losses.append(loss)
                seed_accs.append(acc)
                seed_times.append(elapsed)

            all_losses.append(seed_losses)
            all_accs.append(seed_accs)
            all_times.append(seed_times)

    history[opt]["loss"] = np.mean(all_losses, axis=0)
    history[opt]["acc"] = np.mean(all_accs, axis=0)
    history[opt]["time"] = np.mean(all_times, axis=0)

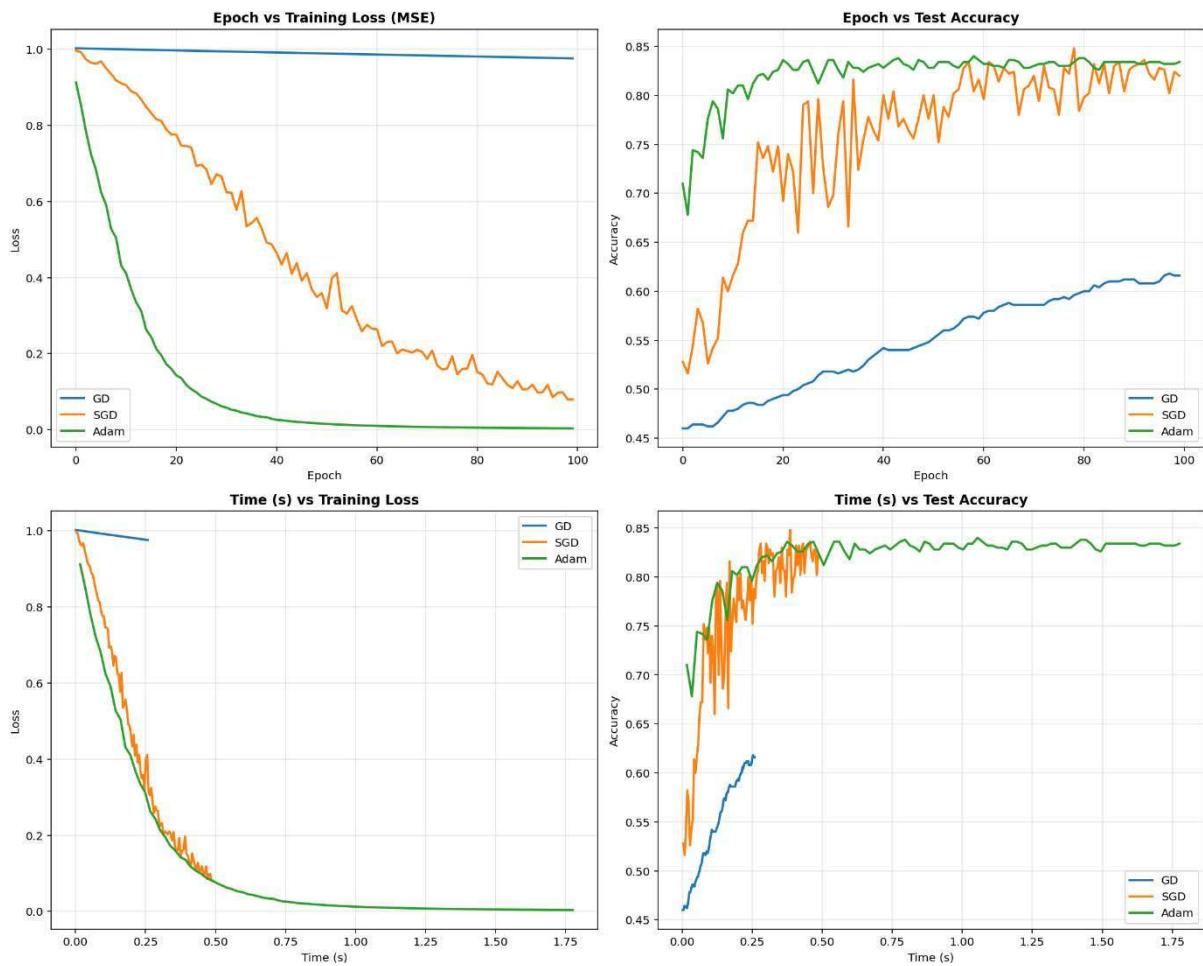
return history, epochs
```

Burada algoritmalarımı çalıştırırken, 5 farklı başlangıç durumu için durumları değerlendirdim. 5 kez çalıştırmanın amacı, ortalama alarak şans faktörünü azaltmaktı. Epoch değerini 100 olarak belirledim. Yani eğitim boyunca 100 defa dataset taranır. SGD için ise batch_size'ını 32 olarak belirledim. Yani SGD bütün veriyi taramak yerine 32 tane örnek üzerinden gradyen hesaplar. Eğer ki batch'ı artırırsam dalgalanmalar düşer, kaos azalır . Daha sonra her bir başlangıç durumu için (5 tane) GD SGD ADAM hesaplanır ve loss, accurate ve time değerleri dizilerde tutulur. Burada overfittingi önleyen durumlardan biri

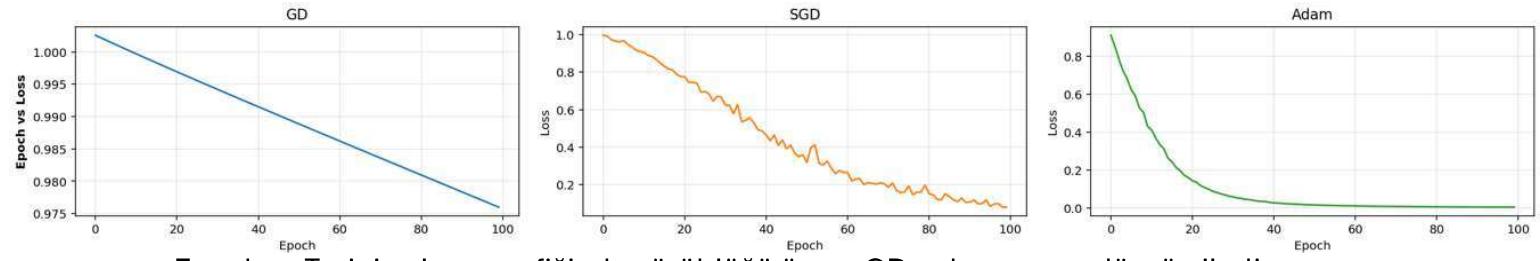
indices = np.arange(X_train.shape[0])
np.random.shuffle(indices)
X_shuf = X_train[indices]
y_shuf = y_train[indices] ise burada
yaptığım
eğitim verisini
karıştırmam oldu. Çünkü veri setlerim alt
alta aynı ikişer sorulardan oluşuyordu.
Modelin ezberleme ihtimalini düşürmüşt
oldum burada.

GD, her epoch'ta bütün datasetini kullanır. Bu sayede zig-zag önlenir fakat güncellemler azdır. Loss düşme hızı yavaş olacaktır.

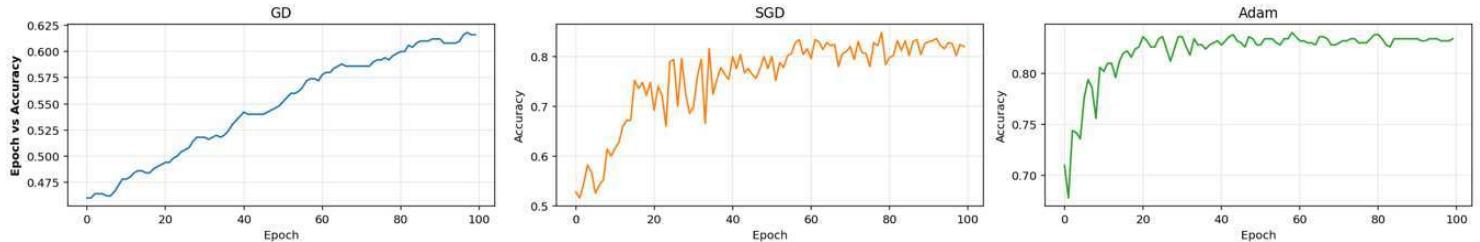
SGD ve ADAM ise mini-batch kullanarak çalışır. Bu sayede daha hızlı amacına ulaşırlar fakat ADAM'da momentum değerleri de kullanılarak en hızlı ve en optimize olarak çalışacaktır.



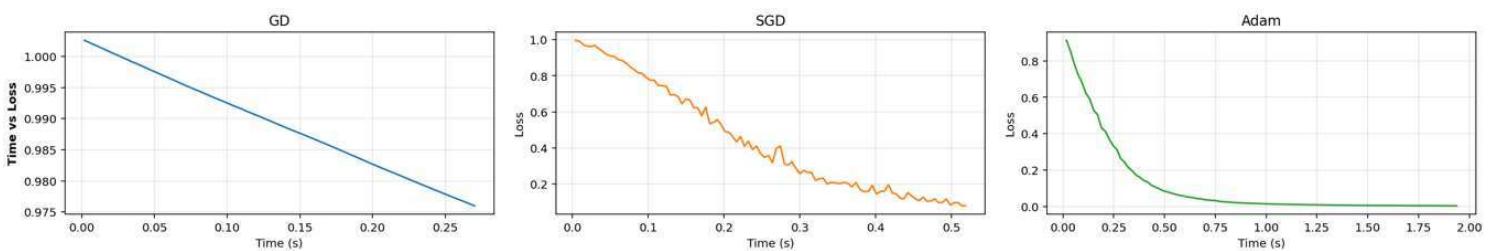
Yazdığım kodların, oluşturduğum veri setlerinin görsel yorumlanması ise bu şekilde olacaktır.



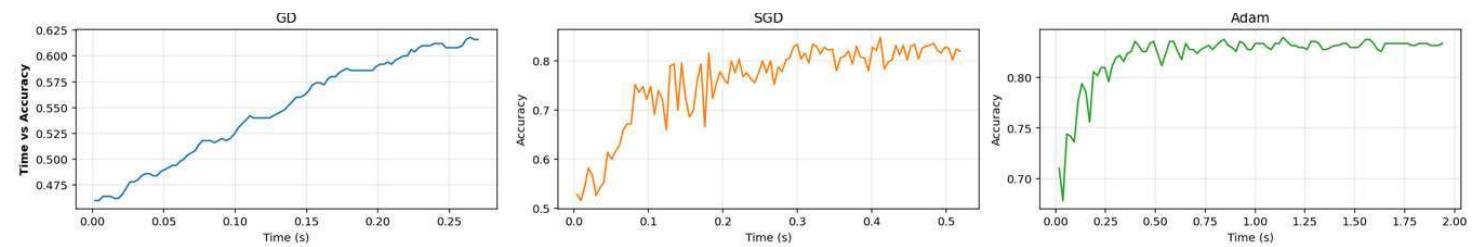
Epoch vs Training Loss grafiğinde görüldüğü üzere GD çok yavaş ve düzgün ilerliyor, dalgalanma neredeyse yok. GD'nin en büyük dezavantajı çok yavaş olmasıdır. SGD ise GD'ye göre daha hızlı düşüyor fakat burada dalgalanmalar oluşuyor çünkü batch_size'ı 32 olarak ayarlamıştık. Tek seferde tüm veriyi okumak yerine 32 veriyi okuyor. ADAM ise çok çok hızlı düşüyor ve titreşim bile yapmıyor. Eski bilgileri de kullandığı için veriyi çok hızlı öğreniyor. Chat GPT'den bu üçünün farkını bana gerçek hayattan örnek olarak ver dediğimde, GD'ye Kamyon, SGD'ye Motor, ADAM'a Tesla demişti 😊 .



Epoch vs Test Accuracy grafiğinde GD 0.62 seviyelerine ulaşırken SGD 0.8 ve ADAM neredeyse 1 seviyesine ulaşıyor. Burada GD yine yavaş ilerlerken SGD zıplayarak kaos içinde hareket ediyor fakat hızlı ilerliyor. ADAM ise dalgalanmayı minimum seviyede tutarak hızlı bir şekilde en accurate değere ulaşıyor.



Time vs Training Loss Grafiklerinde ise GD 100 epoch'a en kısa sürede ulaşmasına rağmen Loss değeri en yüksek method olarak kalmıştır. Bunun nedeni bütün datasetini her güncellemede kullanmasıdır. SGD ise 0.5 saniye içinde hızlıca Loss değerini indirmiştir. ADAM ise 2 saniyede Loss değerini neredeyse 0'a indirerek yine en optimize yöntem olduğunu ispatlamıştır.



Son olarak Time vs Accuracy kısmında GD'nin verimi çok düşüktür. Zaman başına öğrenme oranı en düşük olan methodtur. SGD ise yine dalgalı ve hızlı bir şekilde doğruluk değerini arttırır. Mini-batch'ler ortamı yaratır. ADAM ise en hızlı ve stabil yöntemdir. Sonucu en optimal veren yöntemdir.

```
# 3. TIME VS LOSS
ax = axes[1, 0]
for opt in history:
    ax.plot(history[opt]['time'], history[opt]['loss'], label=labels[opt], color=colors[opt], linewidth=2)
ax.set_title("Time (s) vs Training Loss", fontweight='bold')
ax.set_xlabel("Time (s)")
ax.set_ylabel("Loss")
ax.legend()
ax.grid(True, alpha=0.3)

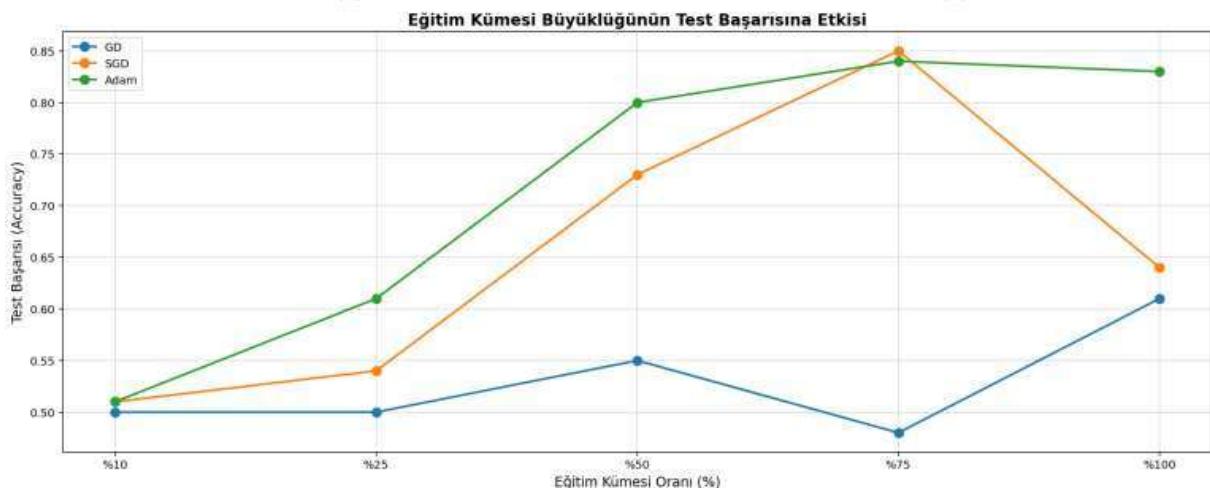
# 4. TIME VS ACCURACY
ax = axes[1, 1]
for opt in history:
    ax.plot(history[opt]['time'], history[opt]['acc'], label=labels[opt], color=colors[opt], linewidth=2)
ax.set_title("Time (s) vs Test Accuracy", fontweight='bold')
ax.set_xlabel("Time (s)")
ax.set_ylabel("Accuracy")
ax.legend()
ax.grid(True, alpha=0.3)
```

```
# 1. EPOCH VS LOSS
ax = axes[0, 0]
for opt in history:
    ax.plot(range(epochs_count), history[opt]['loss'], label=labels[opt], color=colors[opt], linewidth=2)
ax.set_title("Epoch vs Training Loss (MSE)", fontweight='bold')
ax.set_xlabel("Epoch")
ax.set_ylabel("Loss")
ax.legend()
ax.grid(True, alpha=0.3)

# 2. EPOCH VS ACCURACY
ax = axes[0, 1]
for opt in history:
    ax.plot(range(epochs_count), history[opt]['acc'], label=labels[opt], color=colors[opt], linewidth=2)
ax.set_title("Epoch vs Test Accuracy", fontweight='bold')
ax.set_xlabel("Epoch")
ax.set_ylabel("Accuracy")
ax.legend()
ax.grid(True, alpha=0.3)
```

(Grafiklerin kod hali)

EĞİTİM KÜMESİNİN BÜYÜKLÜĞÜNÜN PERFORMANSA ETKİSİ



Yukarıdaki Eğitim Kümesi ve Başarı grafiği için GD her zaman artmamıştır. Çünkü GD bütün batch üzerinden işlem yaptığı için yerel minimumlara takılma riski artıyor. Veri arttıkça doğruluk sürekli olarak bu yüzden artmadı.

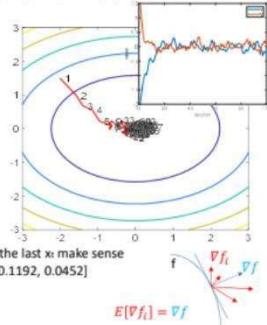
SGD için ise güzel bir şekilde ilerlerken verinin tamamını kullanınca başarı oranında bir düşüş gözlemleniyor çünkü SGD gürültülü bir methodtur. %100 veri olunca başarı düşüşü yaşamاسının temel nedeni minimumum etrafında dolanırken uzaklaşmasından kaynaklanır.

Stochastic Gradient Descent

- $x_{t+1} = x_t - \eta_t \nabla f_i(x_t)$
- $f(x_1, x_2) = x_1^2 + 2x_2^2$
- Learning rate (η_t) = 0.1
- it does not converge
- Code: sgd.m

Region R : an area (in 2 dim)
Outside R, GD and SGD are very similar
Inside R (near x^*), SGD fluctuates
Usage of mean(x) or mean ($x_1 + \dots + x_d$) instead of the last x make sense
Here, the last $x = [-0.2692, 0.3260]$, mean(x) = [-0.1192, 0.0452]

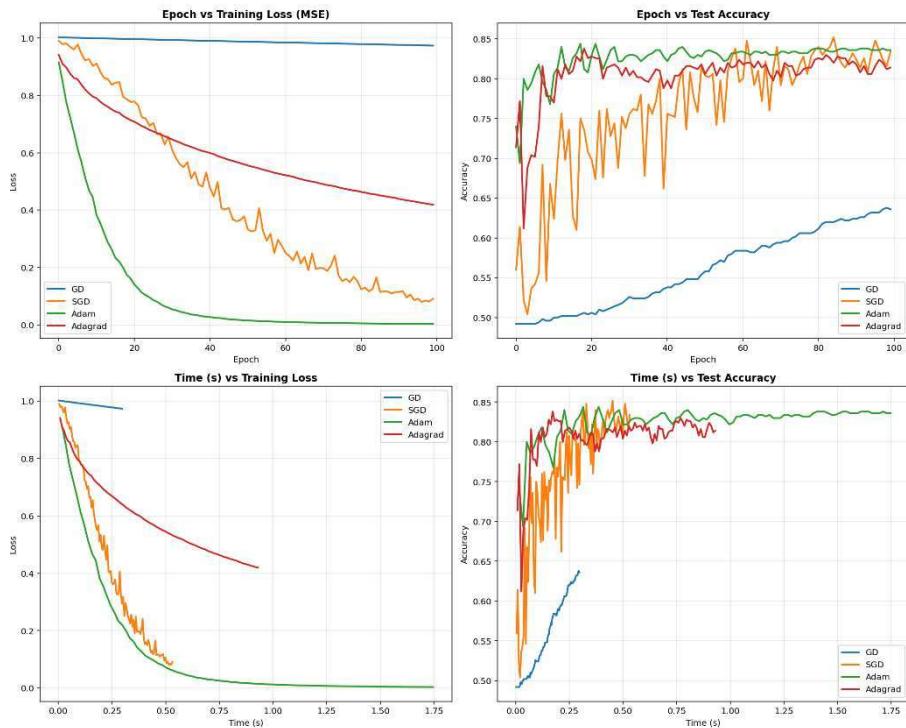
Each GD iteration requires $d * n$ calculations
Each SGD iteration requires d calculations



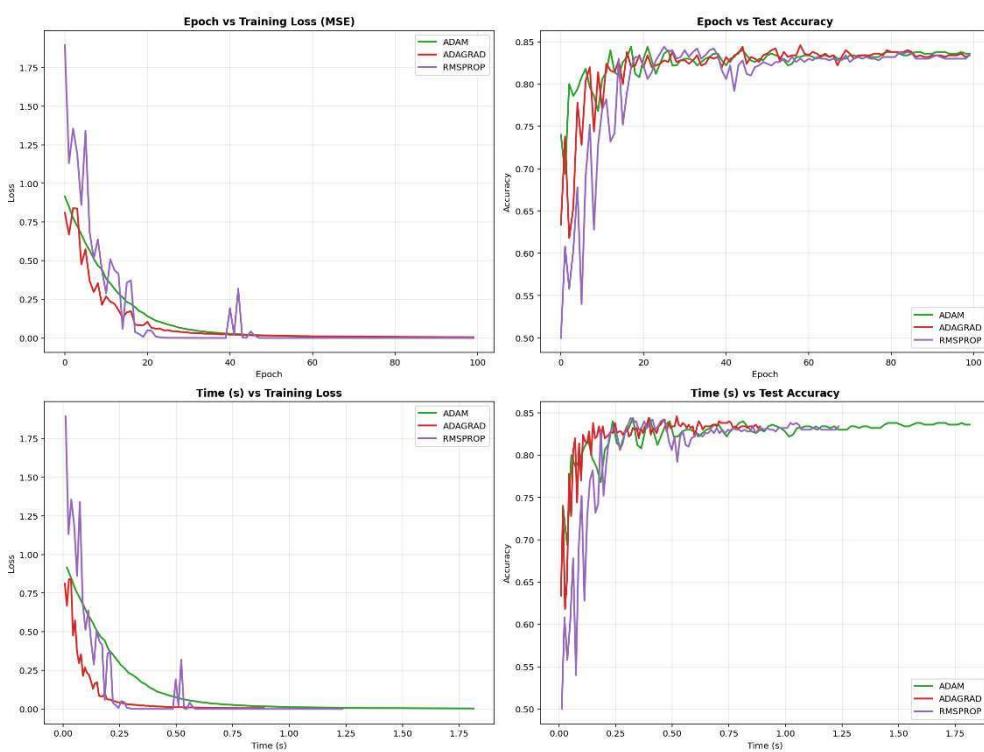
Burada da görüldüğü üzere SGD methodu minimum nokta etrafında salınım yapar. Her zaman Converge etmeyebilir.

ADAM ise yine en temiz methodtur. Büyük veriyle çalışmakta son derece başarılı ve dengeli. Örneğin %75'te SGD daha accurate değere sahip fakat ADAM daha optimize ve stabil olduğu için ADAM kullanmak daha güvenilir olacaktır.

FARKLI OPTIMIZASYON ALGORITMALARI



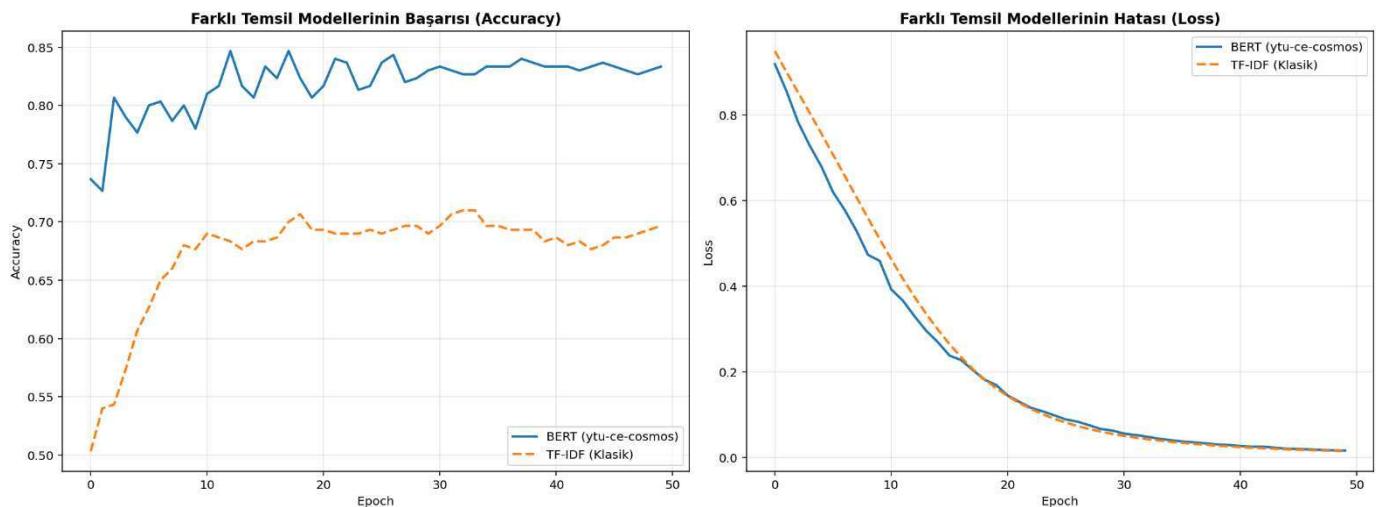
Burada AdaGrad methodunu diğerleriyle karşılaştırdım. AdaGrad'ın temel mantığı eğime bakarak ilerlemesidir. Eğim artıkça yavaşlar, eğim azaldıkça hızlanır. Bu sayede görüldüğü üzere SGD ve GD'den daha yüksek doğruluk oranına sahiptir. Ama Adagrad'ın En büyük handikabı erken yorulmasıdır. Burada SGD'den önce başlayıp SGD'nin gerisine düşüyor öğrenme sürecinde görüldüğü gibi. Tabi learning rate farkları da var burada. SGD = 0.1, AdaGrad = 0.001 (ADAM ile aynı olması için verdim). Bu erken yorulma problemini ise ADAM, momentum kullanarak çözer.



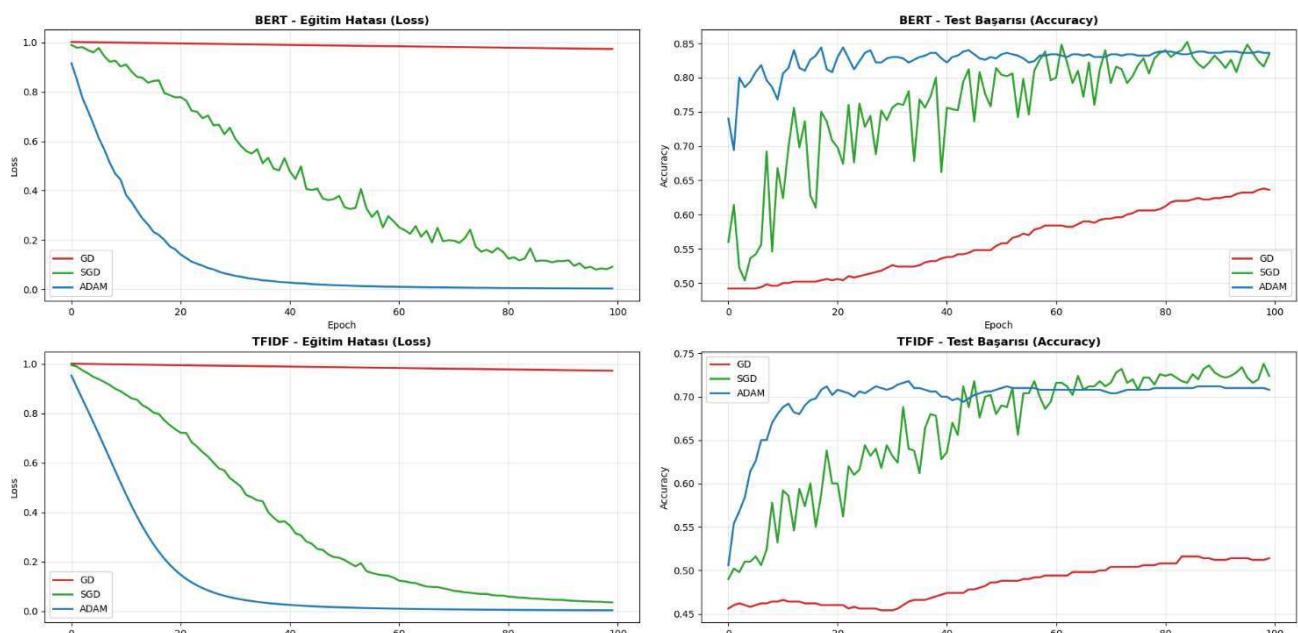
Burada ise RMSProp Algoritmasını kullandım. Görüldüğü gibi RMSProp'ta sert dalgalanmalar mevcut. Loss artıp azalıyor, yani ADAM kadar stabil değil fakat Adagrad'dan daha iyi çünkü Adagrad'ın erken yorulma problemi burada yok. Time/Loss grafiğinde görüldüğü gibi RMSProp'un loss'u hızla inerek ADAGRAD'ın altına düştü. Bunu yapabilmesinin sebebi ağırlıklarını gradyanlarına göre güncelleyebilmesidir.

FARKLI ANLAMSAL TEMSİL MODELLERİNİN KULLANIMI

Burada BERT ve TF-IDF modellerini kıyasladım. TF-IDF modeli kelimelerin sadece varlığına odaklandığı için kelimeler arasında ilişki kurmakta zorlanır. Yani "Ali okula gitti." - "Okula Ali gitti." cümlelerini aynı kabul eder çünkü aynı kelimeler vardır. BERT ise bir insan gibi yaklaşarak kelimeler arasındaki anlam farkını çözer. Örneğin: "Telefonum dondu." ile "Havadan dondu." Cümlelerindeki dondu kelimesi farklı vektörlere sahiptir. Bu sayede BERT ile eğitilen modelin başarı oranı çok daha yüksek çıkmıştır.



Algoritmaların çıktıları ise aşağıda bizim istediğimiz gibi gelmiştir. BERT modelindeki başarı 3 algoritmada da daha yüksektir.



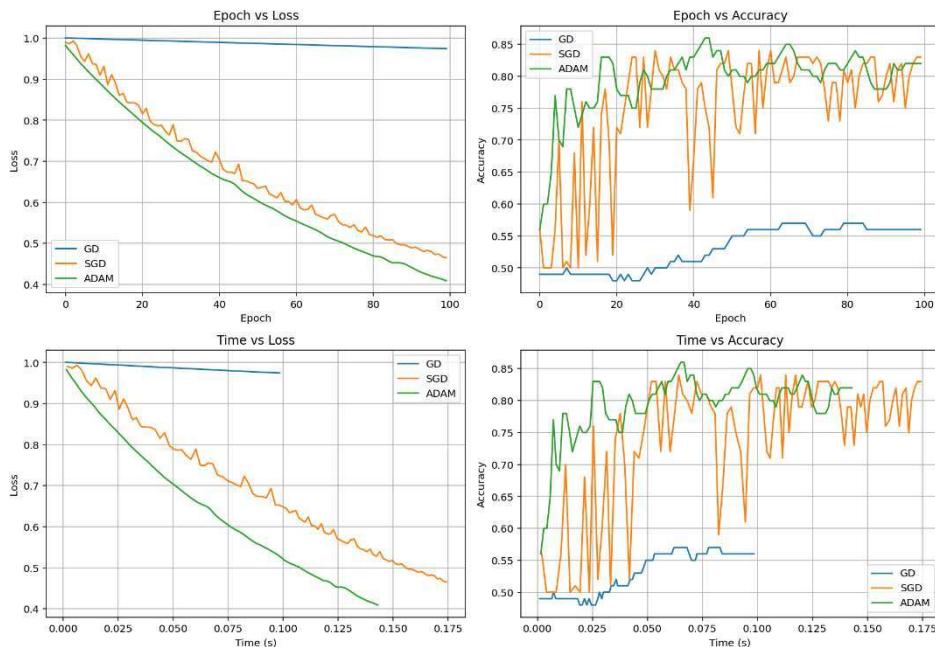
NEURAL NETWORK OLMADAN MODEL

```
class OneLayerModel:  
    def __init__(self, input_dim):  
        self.W = np.random.randn(input_dim, 1) * np.sqrt(1/input_dim)  
  
    def forward(self, X):  
        return np.tanh(X @ self.W)  
  
    def backward(self, X, y_true, y_pred):  
        N = X.shape[0]  
        dW = X.T @ ((y_pred - y_true)*(1 - y_pred**2)) / N  
        return dW
```

Burada ise ilk olarak yine aynı şekilde $(2d+1) \times 1$ 'lik ağırlık matrisini tanımlarız. Bu fonksiyonu ise $\text{np.sqrt}(1/\text{input_dim})$ ile çarparak ağırlıkların aşırı büyük veya küçük olmasını engelleriz.

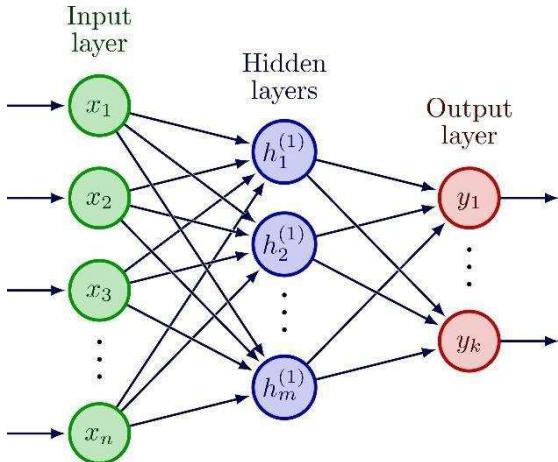
X: $(N \times 2d+1)$
W: $(2d+1 \times 1)$
çıkıtı: $(N \times 1)$

Forward fonksiyonunda ise inputumuz ile ağırlığımızı çarparak $N \times 1$ 'lik bir sonuç elde ederiz. ($N=100$) $(N \times 2d+1) \times (2d+1 \times 1) = N \times 1$
Backward fonksiyonunda ise görüldüğü üzere gradyen hesabı yaparız.



MLP olmadan modelimizin sonucunda görüldüğü üzere fonksiyonlar Neural Network'a göre daha az stabil ve daha dalgalı çıktı. Yani normal modelde yalnızca giriş ve çıkış arasında bağlantı varken, Multi Layer'da daha fazla katman ile daha derin sonuçlar üretebiliriz. MLP'nin gizli katmanı burada çok ciddi rol alır.

NEURAL NETWORK KATMANLARINI RECURSIVE FONKSIYON İLE BELİRLEMЕK



Rapora 3 katmanlı MLP yapısını eklerken bunun birbirini tekrar eden döngülerden olduğunu gördüm. Bu sebeple Recursive Fonksiyon ile MLP yapısı kurdum. Burada katman sınırı yoktur.

```
class RecursiveMLP:
    def __init__(self, layer_sizes):
        self.weights = []
        self.layer_sizes = layer_sizes
        self.activations = {}

        for i in range(len(layer_sizes) - 1):
            n_in = layer_sizes[i]
            n_out = layer_sizes[i+1]
            scale = np.sqrt(1 / n_in)
            self.weights.append(np.random.randn(n_in, n_out) * scale)

    def forward(self, X):
        return self._forward_recursive(X, 0)

    def _forward_recursive(self, current_input, layer_idx):
        if layer_idx == len(self.weights):
            return current_input

        self.activations[layer_idx] = current_input

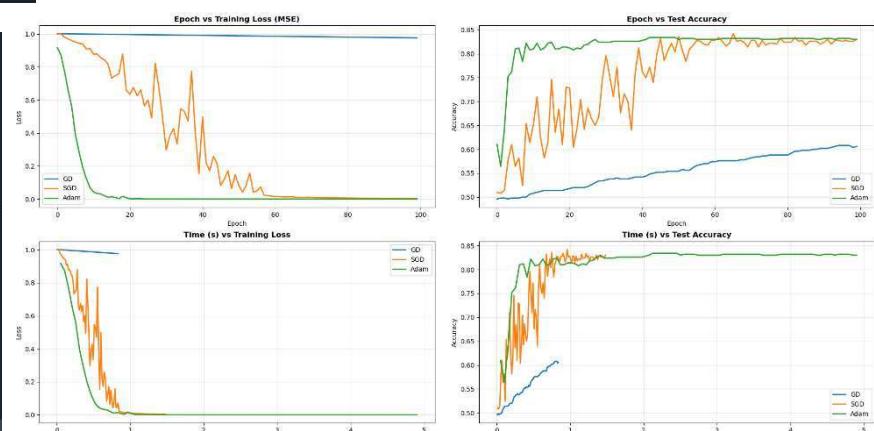
        z = current_input @ self.weights[layer_idx]
        a = np.tanh(z)

        return self._forward_recursive(a, layer_idx + 1)
```

Burada görüldüğü gibi `_forward_recursive` fonksiyonu kendi kendini katman sayısı kadar çağıracaktır.

```
input_dim = X_train.shape[1]
layer_structure = [input_dim, 128, 64, 32, 1]
```

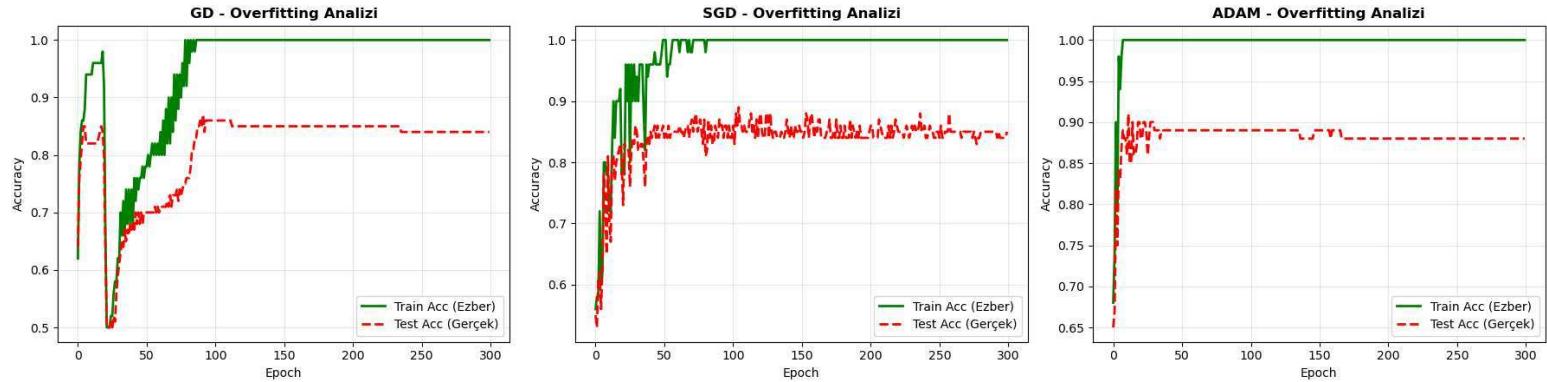
Burada belirttiğim şekilde 5 katmanlı bir yapıdır. 3 gizli nöron, 1 input, 1 output katmanı olacak şekildedir. Bu koda göre algoritmaların çıktıları ise şekildeki gibidir.



Fakat katman sayısını arttırmak her zaman avantaj yaratmaz, overfitting olmasına, hesaplama maliyetinin ciddi bir şekilde artmasına sebep olabilir. En önemli faydalardan biri ise daha karmaşık ilişkileri öğrenmesidir. Yani bilgisayarlar insan beyni gibi düşünmebilme özelliğini kazanır.

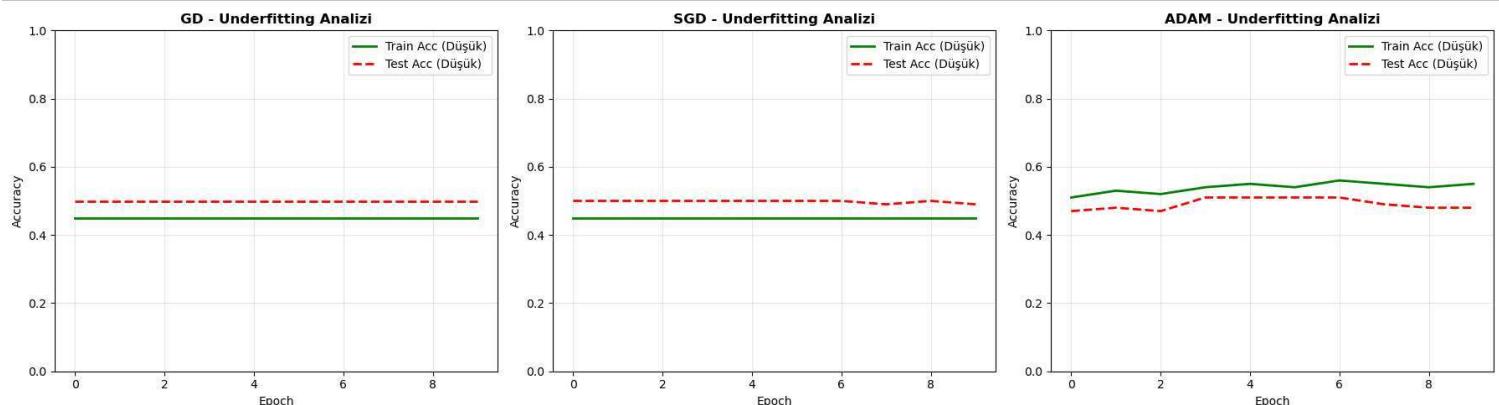
OVERFITTING VE UNDERFITTING DURUMU

Bu tarz algoritmalarla çalışırken overfitting ve underfitting durumlarıyla karşılaşabiliriz. Overfitting durumunda model eğitim verisini ezberler, her türlü gürültüyü öğrenir ve training set'de çok başarılıdır. Test verisi ise kötü bir performans verir.



Kodumdaki 2 katmanlı yapıdaki Hidden Layer sayısını 64'ten 1024'e çıkardım, veriyi azalttım 100'den 50'ye ve epoch sayısını artttırdım. Bu sebeplerden dolayı model'de overfitting olur. Gradiklerde görüldüğü gibi 3 algoritma için de accuracy training set'te %100'ye ulaşırken Test setinde %80'lere kalıyor. Yani model ezberledi ama öğrenmedi.

Underfitting durumu ise Overfitting durumunun tam tersidir. Hidden Layer'deki nöron sayısını 1'e indirip, epoch sayısını 10'a düşürünce model kalitesi iyice düşecektir.

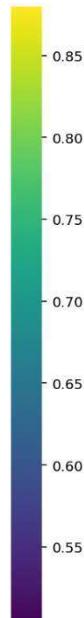
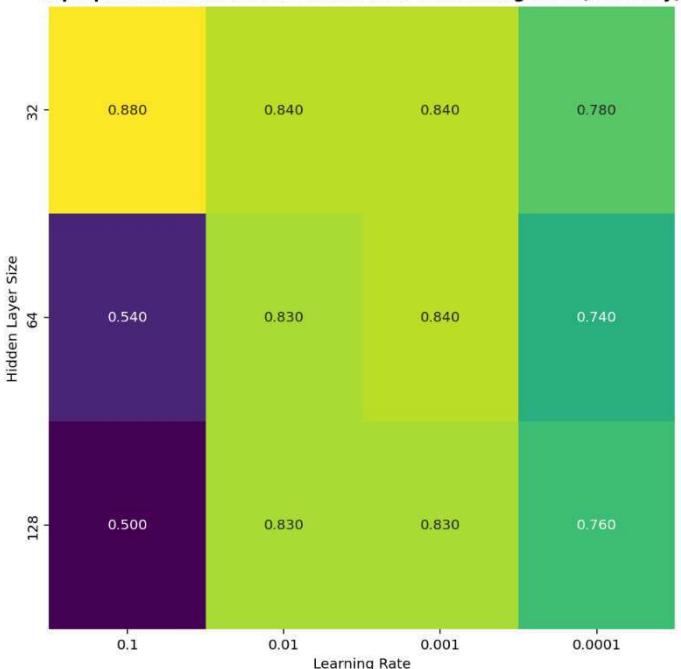


Göründüğü üzere başarı oranları %50 civarlarında. Normalde ADAM'da 85'lere kadar başarı varken underfitting olduğu için başarı oranı ciddi bir şekilde düştü.

PARAMETRELERİN EN İYİ ŞEKLDE BELİRLENMESİ

Burada ise farklı Layer boyaları ve farklı learning rate'ler için accurate oranlarını karşılaştırdım. Layer boyutlarını 32,64,128 ve Learning Rate'leri 0.1,0.01,0.001 olacak şekilde 9 deney yaptım. Tabii büyük datasetleri için bu elverişli bir yöntem değil çünkü hesaplama maliyeti çok yüksek. Ancak küçük datasetleri için elverişli olabilir.

Hiperparametre Taraması: Hidden Size vs Learning Rate (Accuracy)



```
epochs = 30
batch_size = 32
learning_rates = [0.1, 0.01, 0.001, 0.0001]
hidden_dims = [32, 64, 128]

for i, h_dim in enumerate(hidden_dims):
    for j, lr in enumerate(learning_rates):
```

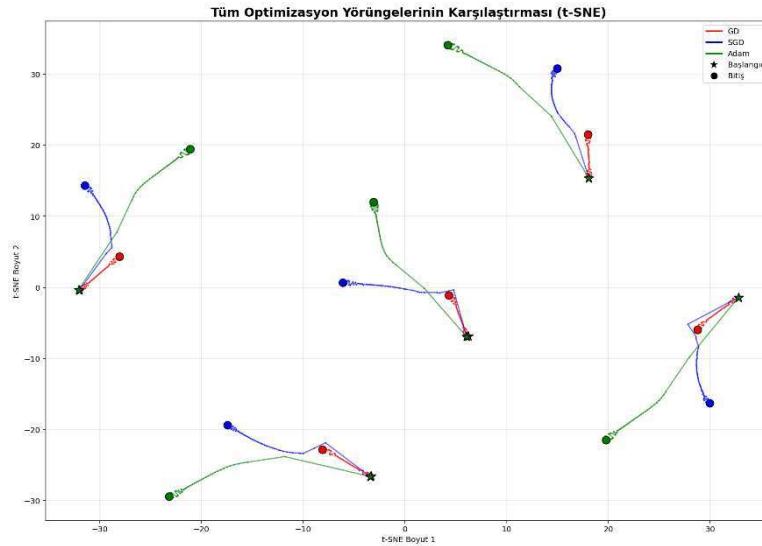
(kod yapım bu şekilde)

Göründüğü üzere Learning Rate'in 0.01/0.001 olduğu ve Hidden Layer sayısının 32/64 olduğu durumda accuracy değerimiz maksimize olmuş oluyor. Kodumda da bu sayıları kullandım.

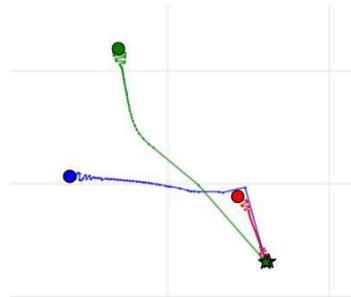
Büyük data setlerindeki optimizasyon için ise çeşitli stratejiler kullanılır. Örneğin,

- Random Search yani Rastgele Arama, yukarıda yaptığımız gibi MxN arama yerine 20 nokta dene gibi random sayılar üzerinden gider ve oldukça hızlıdır.
- Bayesian Optimizasyon akıllı arama olarak geçer. Rastgele değildir, önceki denemelerin sonucuna göre hareket eder. 0.1 kötü, 0.0001 iyiysse 0.0005 civarını inceler.
- Early Stopping algoritması ise ilk 5 epoch'ta çok kötü giderse 100 epoch'un bitmesini beklemez. Onun yerine algoritmayı yarıda keser ve sonraki durumu inceler. Bu sayede maliyetten inanılmaz tasarruf ederiz.

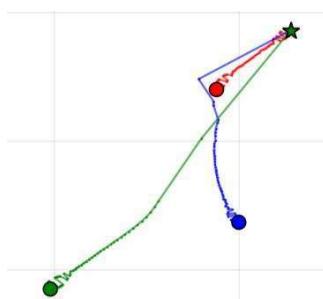
OPTİMİZASYON SÜRECİNİN 2 BOYUTTA GÖSTERİMİ



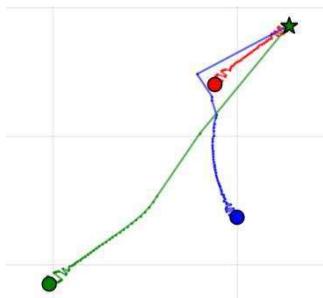
Burada T-sne'nin yaptığı binlerce bazen de milyonlarca W ağırlık vektörlerini 2 boyuta indirerek yorumlamamızı kolaylaştırıyor. Bu sayede algoritmaların zig-zag'larını ve stabil olma durumlarını inceleyebiliyoruz. (Yıldız başladığı yer, Nokta bittiği yer)



GD için, yol çok düzgün ve zig-zag yapmaz. Kısa olmasının nedeni her bir başlangıç için öğrenme yavaş ve stabil. Bunun sebebi ise gradient her adımda tüm datasetini kullanır ve gürültülü değildir. En büyük handikabı aşırı yavaş olmasıdır.



SGD algoritması ise görüldüğü gibi kıvrımlı ve zig-zagli olacaktır. Yol gürültülü ve kaotiktir, bunun sebebi mini batch kullanmasıdır. Bu sebeple Gradient her adımda değişir gürültü oluşur.



ADAM'da ise yol çok daha kararlı ve kısadır. Zig-zag neredeyse yoktur bunu Momentum ile sağlar. ADAM, gürültüyü bastırır ve en hızlı ve optimum şekilde hedefine ulaşır.