TÜRKİYE CUMHURİYETİ

YILDIZ TEKNİK ÜNİVERSİTESİ

BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ

Bilgisayar Mühendisleri için Diferansiyel Denklemler

Burak Başol

24011037

Optimizasyon Ödevi

Akademisyen:

Prof. Dr. Mehmet Fatih AMASYALI

December 5, 2025

İçindekiler

1. Giriş

Bilgisayar mühendisleri için diferansiyel denklemler dersi 1.ödevi kapsamında Cosmos Gemma 9b T1 modeli kullanılarak soru ve cevap üretilmesi ve Cosmos Turkish e5 Large modeli kullanılarak üretilen soru ve cevapların anlamsal temsilinin oluşturulması gerekmektedir.

Oluşturulan veri istenilen modellerden ve optimizasyon algoritmalarından geçirildikten sonra elde edilen sonucun görselleştirilmesi, karşılaştırılması ve değerlendirilmesi gerekmektedir.

Ödeve ait videonun linki: Youtube

2. Ödev Dosyalarının İçeriği

2.1. Kod Dosyaları

questionGenerator.ipynb dosyasında Cosmos Gemma 9b T1 modelinin kullanılma şekli yer almaktadır. Çıktı olarak **soru_cevap_dataset_500.csv** dosyasını vermektedir.

embeddingQuestion.ipynb dosyasında ise Cosmos Turkish e5 Large modeli kullanılarak soru ve cevapların anlamsal temsilinin oluşturulması gösterilmektedir. Çıktı olarak **soru_cevap_embeddings.pkl** dosyasını vermektedir.

e5_large_embedding.ipynb dosyasında ise e5 Large modeli ile anlamsal temsil oluşturulmaktadır. Çıktı olarak **e5_large_embeddings.pkl** dosyasını vermektedir.

restOfTheProject.ipynb dosyası ise modeli ve optimizasyon algoritmalarını çalıştırıp görselleştirme işlemlerini yapmaktadır. Çıktıları **imgs/** klasörünün altındadır.

2.2. Diğer Dosyalar

Sorulara ait csv dosyası ve pickle dosyaları klasörde bulunmaktadır. Ayrıca **imgs/** klasörü içerisinde üretilen grafiklerin asıl png dosyaları bulunmaktadır.

3. Veri Üretimi

3.1. Cosmos 9b T1

Verileri üretmek için Cosmos 9b T1 modeli kullanıldı. 500 adet soru, 500 adet iyi cevap ve 500 adet kötü cevap üretildi. İyi cevaplar ve kötü cevaplar anlamsal olarak soruya uygun fakat kötü cevap daha az detayla bilgi sunuyor. Böylece kötü cevabın tespiti daha zor oluyor. Sorular aşağıdaki konu başlıklarından seçilerek her seferinde 10 tane üretilecek şekilde oluşturuldu.

```
generator_model_id = "ytu-ce-cosmos/Turkish-Gemma-9b-T1"
topics=[
    "Tarih", "Coğrafya", "Fizik", "Kimya", "Biyoloji",
    "Edebiyat", "Felsefe", "Teknoloji", "Yapay Zeka",
    "Psikoloji", "Sosyoloji", "Ekonomi", "Sanat",
    "Müzik", "Sinema", "Spor", "Astronomi", "Matematik",
    "Tıp", "Mühendislik", "Mitoloji", "Doğa", "Hayvanlar Alemi",
    "Gastronomi", "Mimari"
]
```

Modele verilen prompt içerisinde vermesi gereken çıktı aşağıdaki şekilde belirtildi. Bu sayede alınan cevap çok az düzenleme yapılarak csv dosyasına aktarıldı.

```
prompt = (
        f"Bana {current_topic} hakkında {BATCH_SIZE} adet {main_topic} sorusu yaz.
        f"Yazdığın her soru için 1 adet iyi cevap (doğru ve detaylı) ve 1 adet kötü
        f"Format tam olarak şu şekilde olsun ve başka bir şey yazma:\n\n"
        f"Q1: Soru cümlesi buraya\n"
        f"A1: İyi cevap buraya\n"
        f"A2: Kötü cevap buraya\n"
        f"Q2: Soru cümlesi buraya...\n"
        f"A1: ...\n"
        f"A2: ...\n"
        f"(Hepsini bu formatta alt alta yaz)")
```

Modelde her 10 soru üretimi için en fazla 4096 token kullanıldı ve her iterasyonda temperature değeri arttırılarak farklı sonuçlar alınması sağlandı.

```
outputs = generator_model.generate(
        input_ids,
        max_new_tokens=4096,
        eos_token_id=terminators,
        do_sample=True,
        temperature=0.7 + (i * 0.01),
        top_p=0.9,
        repetition_penalty=1.1
    )
```

Oluşturulan sorular csv dosyasında konu, soru, iyi cevap, kötü cevap sırası ile tutuldu.

4. Anlamsal Temsil

4.1. Cosmos e5 Large

Anlamsal temsil işlemi Cosmos e5 Large modeli kullanılarak gerçekleştirildi. Pickle dosyası olarak kaydedildi.

```
def get_detailed_instruct(task_description: str, query: str) -> str:
    return f'Instruct: {task_description}\nQuery: {query}'

task = 'Given a Turkish search query, retrieve relevant passages written in Turkish

formatted_questions = [get_detailed_instruct(task, q) for q in df['Soru']]
question_embeddings = embedding_model.encode(
    formatted_questions,
    convert_to_tensor=False,
    normalize_embeddings=True,
    show_progress_bar=True,
    batch_size=32
)

good_answer_embeddings = embedding_model.encode(
    df['Iyi_Cevap'].tolist(),
    convert_to_tensor=False,
    normalize_embeddings=True,
    show_progress_bar=True,
    batch_size=32
)

bad_answer_embeddings = embedding_model.encode(
    df['Kotu_Cevap'].tolist(),
    convert_to_tensor=False,
    normalize_embeddings=True,
    show_progress_bar=True,
    batch_size=32
)
```

4.2. e5 Large

Anlamsal temsil modelinin başarısını test etmek amacıyla ayrıca e5 Large modeli kullanıldı. Pickle dosyası olarak kaydedildi.

```
def get_detailed_instruct(task_description: str, query: str) -> str:
    return f'Instruct: {task_description}\nQuery: {query}'


print(f"Veri seti yükleniyor: {CSV_FILE}")
df = pd.read_csv(CSV_FILE)
print(f"Toplam {len(df)} kayıt işlenecek. Kullanılacak cihaz: {DEVICE}")
embedding_model = SentenceTransformer(EMBEDDING_MODEL_NAME, device=DEVICE)
formatted_questions = [get_detailed_instruct(TASK, q) for q in df['Soru']]

question_embeddings = embedding_model.encode(
    formatted_questions,
    convert_to_tensor=False,
    normalize_embeddings=True,
    show_progress_bar=True,
    batch_size=BATCH_SIZE
)
df['Soru_Embedding_E5Large'] = list(question_embeddings)

good_answer_embeddings = embedding_model.encode(
    df['Iyi_Cevap'].tolist(),
    convert_to_tensor=False,
    normalize_embeddings=True,
    show_progress_bar=True,
    batch_size=BATCH_SIZE
)
df['Iyi_Cevap_Embedding_E5Large'] = list(good_answer_embeddings)
bad_answer_embeddings = embedding_model.encode(
    df['Kotu_Cevap'].tolist(),
    convert_to_tensor=False,
    normalize_embeddings=True,
    show_progress_bar=True,
    batch_size=BATCH_SIZE
)
df['Kotu_Cevap_Embedding_E5Large'] = list(bad_answer_embeddings)


df.to_pickle(OUTPUT_FILE)
print(f"Başarıyla kaydedildi. Çıktı dosyası: {OUTPUT_FILE}")
```

5. Modelleme

5.1. Tek Katmanlı Model

Model giriş verisini tek bir ağırlık vektörü üzerinden dönüştüren ve çıkışta `tanh` aktivasyon fonksiyonunu kullanan yapay sinir ağı modelidir.

Ağırlık Başlatma

Model, giriş boyutuna göre oluşturulan ağırlık vektörünü şu şekilde başlatır:

$$w \sim \mathcal{N}(0,\, 0.01)$$

Eğer başlangıç ağırlığı verilmişse, bu değer kopyalanarak kullanılır.

İleri Yayılım (Forward Pass)

Modelin ileri yayılımı iki adımdan oluşur:

1. Doğrusal dönüşüm:
$$z = Xw$$
2. Aktivasyon:
$$\hat{y} = \tanh(z)$$

Kayıp Fonksiyonu

Gerçek değerler ile tahminler arasındaki hata, Ortalama Kare Hatası (MSE) ile hesaplanır:

$$L = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Doğruluk Hesabı

Sınıflandırma doğruluğu, tahminlerin işaretine bakılarak belirlenir:

$$\text{prediction} = \text{sign}(\hat{y})$$

Geri Yayılım (Gradient Hesaplama)

`tanh` fonksiyonunun türevi:

$$\tanh'(z) = 1 - \hat{y}^2$$

Ağırlık gradyanı:

$$\nabla w = \frac{2}{N} X^\top \left((\hat{y} - y_{\text{true}}) \odot (1 - \hat{y}^2) \right)$$

Modele ait kod aşağıda verilmiştir:

```
class actualModel:
 def __init__(self, input_dim, initial_w=None) -> None:
  self.name = "Normal"
  if initial_w is None:
   self.w = np.random.randn(input_dim, 1) * 0.01
  else:
   self.w = initial_w.copy()

 def forward(self, X):
  linear = np.dot(X, self.w)
  return np.tanh(linear)

 def loss(self, y_true, y_pred):
  return np.mean((y_true - y_pred)**2)

 def accuracy(self, y_true, y_pred):
  predictions = np.sign(y_pred)
  return np.mean(predictions == y_true)

 def get_gradients(self, X_batch, y_batch, y_pred):
  N = X_batch.shape[0]
  diff = (y_pred - y_batch)
  dtanh = (1 - y_pred**2)
  grad = np.dot(X_batch.T, diff * dtanh) * 2 / N
  return grad
```

5.2. Çok Katmanlı Model

Model bir gizli katman ve bir çıkış katmanı içeren, `tanh` aktivasyon fonksiyonunun kullanıldığı Çok Katmanlı Algılayıcı (MLP) mimarisidir.

Ağırlıkların Başlatılması

Model iki parametre matrisiyle başlatılır:

$$W_1 \in R^{d\times h}, \qquad W_2 \in R^{h\times 1}$$

Her iki matris de küçük ölçekli normal dağılımdan örneklenir.

İleri Yayılım (Forward Pass)

İleri yayılım adımları şu şekildedir:

$$z_1 = XW_1, \qquad a_1 = \tanh(z_1)$$

$$z_2 = a_1 W_2, \qquad \hat{y} = \tanh(z_2)$$

Ara değerler, geri yayılım için önbellekte saklanır.

Geri Yayılım (Backpropagation)

Çıkış katmanı gradyanı:

$$\delta_2 = \frac{2}{N}(\hat{y} - y_{\text{true}}) \odot (1 - \hat{y}^2)$$

$$\nabla W_2 = a_1^\top \delta_2$$

Gizli katman gradyanı:

$$\delta_1 = (\delta_2 W_2^\top) \odot (1 - a_1^2)$$

$$\nabla W_1 = X^\top \delta_1$$

Bu gradyanlar, optimizasyon algoritması tarafından ağırlık güncellemelerinde kullanılır.

Modelin Özellikleri

- Tek katmanlı modele kıyasla daha yüksek temsil gücüne sahiptir.
- Gizli katman boyutu (h) arttıkça öğrenilebilen fonksiyonların karmaşıklığı artar.
- `tanh` aktivasyonu sayesinde gradyan akışı dengeli ve kararlı kalır.

Modele ait kod aşağıda verilmiştir:

```
class TwoLayerMLP:
 def __init__(self, input_dim, hidden_dim=64):
  self.name = f"MLP-{hidden_dim}"
  self.params = {
   'W1': np.random.randn(input_dim, hidden_dim) * 0.05,
   'W2': np.random.randn(hidden_dim, 1) * 0.05
  }
  self.cache = {}

 def forward(self, X):
  z1 = np.dot(X, self.params['W1'])
  a1 = np.tanh(z1)
  z2 = np.dot(a1, self.params['W2'])
  a2 = np.tanh(z2)
  self.cache = {'X': X, 'a1': a1, 'a2': a2}
  return a2

 def get_gradients(self, X, y_true, y_pred):
  N = X.shape[0]
  a1 = self.cache['a1']
  delta2 = 2 * (y_pred - y_true) * (1 - y_pred**2) / N
  grad_W2 = np.dot(a1.T, delta2)
  delta1 = np.dot(delta2, self.params['W2'].T) * (1 - a1**2)
  grad_W1 = np.dot(X.T, delta1)
  return {'W1': grad_W1, 'W2': grad_W2}
```

6. Optimizasyon Algoritmaları

Bu bölümde kullanılan optimizasyon algoritmaları açıklanmaktadır. Modellerin parametreleri, kayıp fonksiyonunu minimize edecek şekilde bu algoritmalarla güncellenmiştir.

6.1. Gradient Descent

Gradient Descent'te amaç, parametrelerin gradyan yönünde güncellenerek kaybın azaltılmasıdır. Her iterasyonda tüm veri kümesi kullanılarak gradyan hesaplanır ve parametre güncellemesi aşağıdaki gibi yapılır:

$$w \leftarrow w - \eta \nabla_w L(w)$$

Burada η öğrenme oranını, $\nabla_w L(w)$ ise kayıp fonksiyonunun gradyanını ifade eder. Verilen kodda `train_gd` fonksiyonu tam veri üzerinden gradyan hesaplar.

6.2. Stochastic Gradient Descent

Stochastic Gradient Descent (SGD), GD'den farklı olarak her adımda tek bir örneğin gradyanını kullanarak parametreleri günceller. Hesaplama maliyeti ciddi biçimde azalırken öğrenme sürecine rastlantısallık katarak yerel minimumlardan kaçılmış olur.

$$w \leftarrow w - \eta \nabla_w L(w; x_i, y_i)$$

Verilen kodda `train_sgd` fonksiyonu, veri kümesini karıştırarak her örnek için ayrı ayrı gradyan hesaplayıp ağırlıkları güncellemektedir.

6.3. Adam

Adam (Adaptive Moment Estimation), hem momentum hem de RMSProp fikirlerini birleştiren optimizasyon algoritmasıdır. Birinci moment ve ikinci moment tutulur. Ayrıca bias düzeltme terimleri içerir.

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
$$w \leftarrow w - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Kodda `AdamOpt` sınıfı ve `train_adam` fonksiyonu bu adımları birebir uygulamaktadır.

6.4. AdaGrad

AdaGrad, her parametre için geçmiş gradyan karelerinin birikimini tutar. Bu sayede sık güncellenen parametrelerin öğrenme oranı küçülürken nadir güncellenen parametrelerin öğrenme oranı yüksek kalır.

$$G_t = G_{t-1} + g_t^2$$
$$w \leftarrow w - \frac{\eta}{\sqrt{G_t} + \epsilon} g_t$$

Bu yöntemin avantajı, seyrek özelliklerde hızlı yakınsamadır. Dezavantajı ise G_t'nin sürekli büyümesi nedeniyle öğrenme oranının zamanla aşırı küçülmesidir.

Verilen kodda `AdaGradOpt` sınıfı ve `train_adagrad` fonksiyonu bu mekanizmayı uygulamaktadır.

6.5. RMSProp

RMSProp, AdaGrad'daki öğrenme hızının çok hızlı küçülmesi problemini çözmek için geliştirilmiştir. Gradyan karelerinin tamamını biriktirmek yerine üstel hareketli ortalama kullanır:

$$E[g^2]_t = \rho E[g^2]_{t-1} + (1-\rho) g_t^2$$
$$w \leftarrow w - \eta \frac{g_t}{\sqrt{E[g^2]_t} + \epsilon}$$

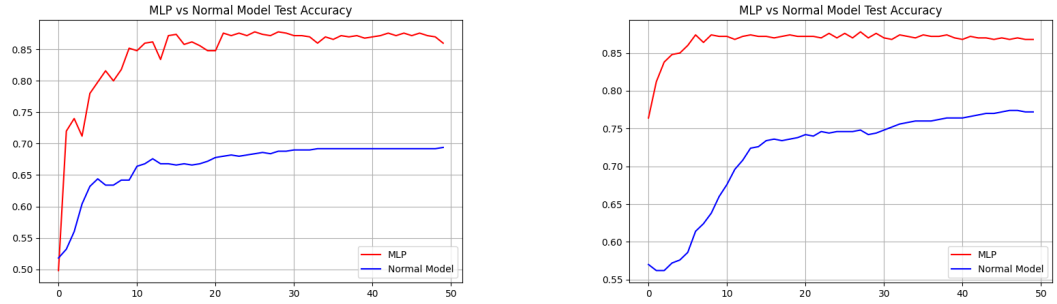
Bu sayede öğrenme oranı daha kararlı kalır ve özellikle derin ağlarda oldukça etkili bir optimizasyon yöntemi haline gelir.

Kodda `RMSPropOpt` sınıfı ve `train_rmsprop` fonksiyonu bu hesaplamayı gerçekleştirmektedir.

Optimizasyon algoritmalarına ait kodlar dökümanın sonunda verilmiştir.

7. Görselleştirme

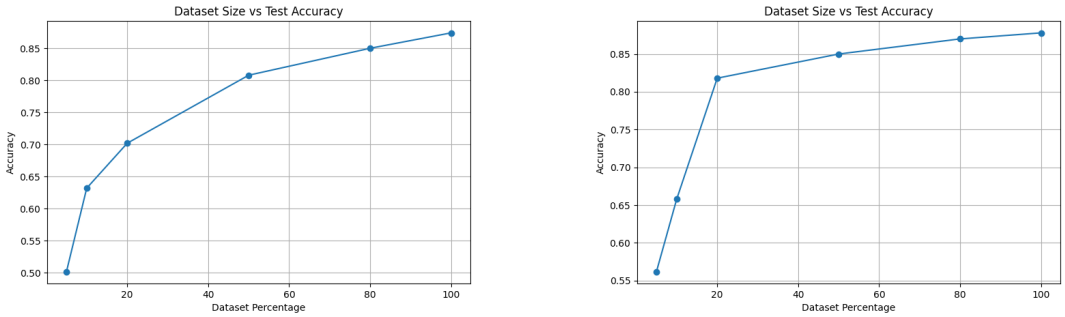
Görselleştirmeler için **matplotlib** kütüphanesi kullanılmıştır.



(a) e5 Large (b) Cosmos e5 Large

Figure 7.1: Embedding Modellerinde Tek Katmanlı ve Çok Katmanlı Model Performansları

e5 Large ile üretilen vektörler kullanıldığında iki model için oluşan test accuracy değerleri yukarıda gösterilmiştir. Tek katmanlı model çok katmanlı modele göre daha düşük başarı elde etmiştir. Bu durum bize veri setimiz için çok katmanlı bir modelin daha başarılı çalıştığını gösterir.

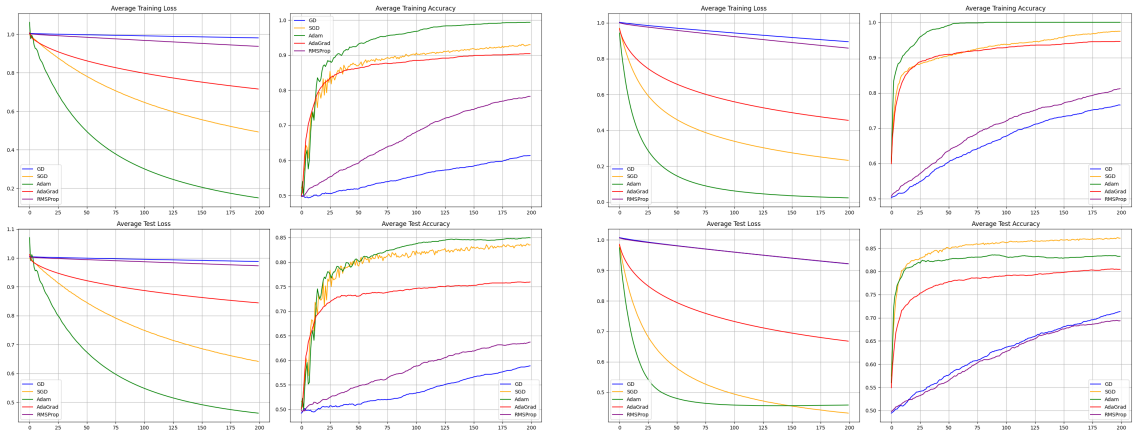


(a) e5 Large (b) Cosmos e5 Large

Figure 7.2: Embedding Modellerinde Veri Seti Büyüklüğünün Etkisi

Veri setinin tamamı (500 soru ve 1000 cevap) iki embedding modeli için de yaklaşık olarak aynı sonuçlar elde edilmiştir. Ama elimizde kısıtlı veri olduğu durumda (veri setinin %20'sinin kullanıldığı durum) Cosmos e5 Large modeli daha yüksek başarı elde etmiştir. Bu durumun e5 Large modelinin çok dilli bir model olmasından dolayı gerçekleştiğini

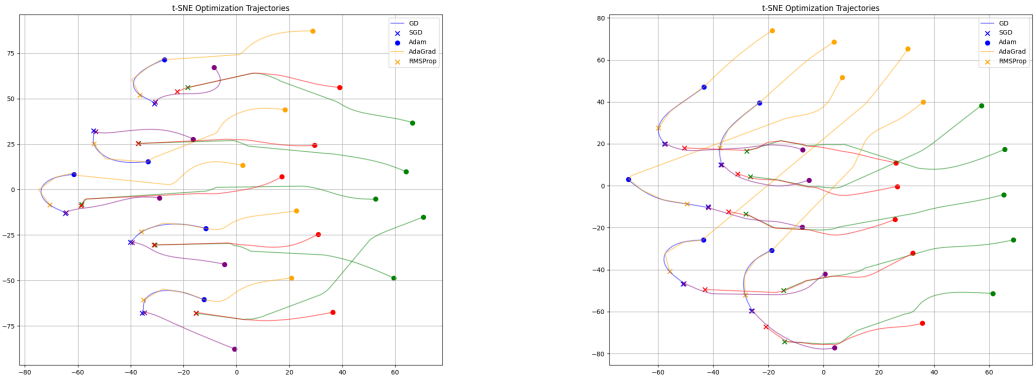
düşünüyorum. Cosmos'un fine tune ettiği modele göre Türkçe'de daha düşük doğrulukla çalışmaktadır.



(a) e5 Large (b) Cosmos e5 Large

Figure 7.3: Embedding Modellerinde Ortalama Optimizasyon Metrikleri

Yukarıdaki grafiklerde her iki embedding modeliyle de oluşturulan vektörlerin iki model için de her optimizasyon algoritmasına göre ortalamsındaki değerleri verilmiştir. Gradient Descent ve RMSProp algoritmaları her iki vektör verisinde de çalıştırıldığı epoch sayısına göre yavaş eğitilmektedir. Diğer optimizasyon algoritmalının eğitimleri ise 50.epochta belli bir yere kadar gelmiştir. Ayrıca dikkatimi çeken şey ise iki embedding modeli'nin test accuracy değerlerine baktığımızda yakın olmalarına rağmen Cosmos'un fine tune ettiği modelin başarısı e5 Large'a göre daha yüksektir



(a) e5 Large (b) Cosmos e5 Large

Figure 7.4: Embedding Modellerinde tSNE Grafiği

Her iki t-SNE grafiği de optimizasyon algoritmalarının farklı başlangıç noktalarından nasıl ilerlediğini gösteriyor. Adam ve RMSProp hızlıca hedef bölgeye doğru yaklaşırken, SGD ve GD daha düzensiz ve uzun yollar izliyor. AdaGrad ise erken aşamada büyük adımlar atıp sonrasında yavaşlıyor.

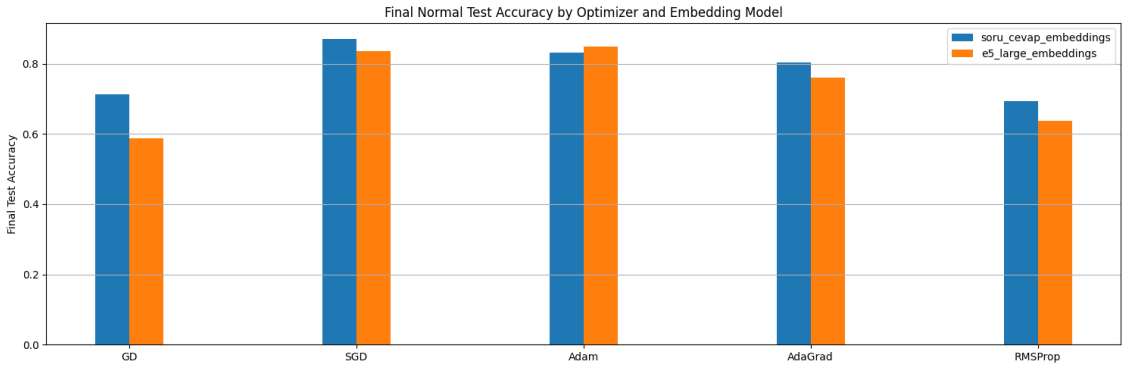


Figure 7.5: Embedding Modellerinin Optimizasyon Algoritmalarında Karşılaştırılması

Sonuç olarak SGD ve Adam diğer modellere göre daha yüksek başarılar elde etmiştir. GD en az başarı elde eden oldu ve kullanılan embedding modelinin de ede edilen başarılara etkisi var. Ayrıca bu grafikler 200 epoch çalıştırılması ile elde edilmiştir. Farklı epoch değerlerinde farklı çıktılar elde edilmektedir.

8. Sonuçlar ve Çıkarımlarım

Elimizdeki veriyi kullanarak bir sonuca varmak için oluşturacağımız modellerde bir çok farklı faktör etkili olmaktadır. Veri setinin temiz olması ve amacımıza uygun olması ilk aşamadır. Sonrasında anlamsal temsil için kullanılan yöntemlerin de etkisi vardır. Bazı durumları yukarıda incelemiştik. Vektörlerimiz hazır olduktan sonra amacımıza ve elimizdeki kaynaklara göre optimizasyon algoritmaları ve model mimarileri test edilmelidir. Bizim için önemli olan kaynak zaman ya da enerji olabilir. Maliyet düşürmek isteyebiliriz. Bu durumlar göz önünde bulundurularak optimizasyon algoritması ve model mimarileri test edilip en uygun olanı seçilebilir.
Ayrıca hocam siz de kabul ederseniz Cosmos araştırma grubunda yanınızda görev almak ve birinci elden deneyim kazanmak isterim. Eğitimim ve kariyerim için önemli bir adım olduğunu düşünüyorum.
Burak Başol
24011037
burak.basol@std.yildiz.edu.tr
+90 506 026 70 98

```
class Optimizer:
 def update(self, params, grads): raise NotImplementedError

class SGDOpt(Optimizer):
 def __init__(self, lr=0.01):
  self.lr = lr
 def update(self, params, grads):
  for key in params:
   params[key] -= self.lr * grads[key]

class AdaGradOpt(Optimizer):
    def __init__(self, lr=0.01, epsilon=1e-8):
        self.lr, self.eps = lr, epsilon
        self.G = {}
    def update(self, params, grads):
        for key in params:
            if key not in self.G:
                self.G[key] = np.zeros_like(params[key])
            self.G[key] += grads[key]**2
            params[key] -= self.lr * grads[key] / (np.sqrt(self.G[key]) + self.eps)

class RMSPropOpt(Optimizer):
    def __init__(self, lr=0.001, decay_rate=0.9, epsilon=1e-8):
        self.lr, self.dr, self.eps = lr, decay_rate, epsilon
        self.E_g2 = {}
    def update(self, params, grads):
        for key in params:
            if key not in self.E_g2:
                self.E_g2[key] = np.zeros_like(params[key])
            self.E_g2[key] = self.dr * self.E_g2[key] + (1 - self.dr) * (grads[key]
            params[key] -= self.lr * grads[key] / (np.sqrt(self.E_g2[key]) + self.e

class AdamOpt(Optimizer):
 def __init__(self, lr=0.001, beta1=0.9, beta2=0.999, epsilon=1e-8):
  self.lr, self.beta1, self.beta2, self.eps = lr, beta1, beta2, epsilon
  self.m, self.v, self.t = {}, {}, 0
 def update(self, params, grads):
  self.t += 1
  for key in params:
   if key not in self.m:
    self.m[key] = np.zeros_like(params[key])
    self.v[key] = np.zeros_like(params[key])
   self.m[key] = self.beta1 * self.m[key] + (1 - self.beta1) * grads[key]
   self.v[key] = self.beta2 * self.v[key] + (1 - self.beta2) * (grads[key]**2)
   m_hat = self.m[key] / (1 - self.beta1**self.t)
```

```
    v_hat = self.v[key] / (1 - self.beta2**self.t)
    params[key] -= self.lr * m_hat / (np.sqrt(v_hat) + self.eps)


def train_generic(model, optimizer, X_tr, y_tr, X_te, y_te, epochs=30, batch_size=3
 history = {'loss': [], 'acc': [], 'val_loss': [], 'val_acc': []}
 N = X_tr.shape[0]
 if batch_size is None or batch_size >= N:
  batch_size = N
 for _ in range(epochs):
  indices = np.random.permutation(N)
  X_s, y_s = X_tr[indices], y_tr[indices]
  for i in range(0, N, batch_size):
   X_b = X_s[i:i+batch_size]
   y_b = y_s[i:i+batch_size]
   y_p = model.forward(X_b)
   grads = model.get_gradients(X_b, y_b, y_p)
   optimizer.update(model.params, grads)
  train_pred = model.forward(X_tr)
  test_pred = model.forward(X_te)
  history['loss'].append(np.mean((y_tr - train_pred)**2))
  history['acc'].append(np.mean(np.sign(train_pred) == y_tr))
  history['val_loss'].append(np.mean((y_te - test_pred)**2))
  history['val_acc'].append(np.mean(np.sign(test_pred) == y_te))
 return history

def train_gd(model, X, y, X_val, y_val, lr=0.01, epochs=100):
 history = {'loss': [], 'acc': [], 'val_loss': [], 'val_acc': [], 'time': [], 'weig
 m0 = time.time()
 for _ in range(epochs):
  y_pred = model.forward(X)
  grad = model.get_gradients(X, y, y_pred)
  model.w -= lr * grad
  history['loss'].append(model.loss(y, y_pred))
  history['acc'].append(model.accuracy(y, y_pred))
  val_pred = model.forward(X_val)
  history['val_loss'].append(model.loss(y_val, val_pred))
  history['val_acc'].append(model.accuracy(y_val, val_pred))
  history['time'].append(time.time() - m0)
  history['weights'].append(model.w.flatten().copy())
 return history

def train_sgd(model, X, y, X_val, y_val, lr=0.01, epochs=100):
 history = {'loss': [], 'acc': [], 'val_loss': [], 'val_acc': [], 'time': [], 'weig
 m0 = time.time()
 N = X.shape[0]
```

```
 for _ in range(epochs):
  idx = np.random.permutation(N)
  for i in idx:
   xi = X[i:i+1]
   yi = y[i:i+1]
   y_pred = model.forward(xi)
   grad = model.get_gradients(xi, yi, y_pred)
   model.w -= lr * grad
  train_pred = model.forward(X)
  val_pred = model.forward(X_val)
  history['loss'].append(model.loss(y, train_pred))
  history['acc'].append(model.accuracy(y, train_pred))
  history['val_loss'].append(model.loss(y_val, val_pred))
  history['val_acc'].append(model.accuracy(y_val, val_pred))
  history['time'].append(time.time() - m0)
  history['weights'].append(model.w.flatten().copy())
 return history

def train_adagrad(model, X, y, X_val, y_val, lr=0.01, epochs=100, eps=1e-8):
    history = {'loss': [], 'acc': [], 'val_loss': [], 'val_acc': [], 'time': [], 'w
    m0 = time.time()
    G = np.zeros_like(model.w)

    for _ in range(epochs):
        y_pred = model.forward(X)
        grad = model.get_gradients(X, y, y_pred)

        G += grad**2
        model.w -= lr * grad / (np.sqrt(G) + eps)

        train_pred = model.forward(X)
        val_pred = model.forward(X_val)

        history['loss'].append(model.loss(y, train_pred))
        history['acc'].append(model.accuracy(y, train_pred))
        history['val_loss'].append(model.loss(y_val, val_pred))
        history['val_acc'].append(model.accuracy(y_val, val_pred))
        history['time'].append(time.time() - m0)
        history['weights'].append(model.w.flatten().copy())
    return history

def train_rmsprop(model, X, y, X_val, y_val, lr=0.001, epochs=100, decay_rate=0.9,
    history = {'loss': [], 'acc': [], 'val_loss': [], 'val_acc': [], 'time': [], 'w
    m0 = time.time()
    E_g2 = np.zeros_like(model.w)
```

```
    for _ in range(epochs):
        y_pred = model.forward(X)
        grad = model.get_gradients(X, y, y_pred)

        E_g2 = decay_rate * E_g2 + (1 - decay_rate) * (grad**2)
        model.w -= lr * grad / (np.sqrt(E_g2) + eps)

        train_pred = model.forward(X)
        val_pred = model.forward(X_val)

        history['loss'].append(model.loss(y, train_pred))
        history['acc'].append(model.accuracy(y, train_pred))
        history['val_loss'].append(model.loss(y_val, val_pred))
        history['val_acc'].append(model.accuracy(y_val, val_pred))
        history['time'].append(time.time() - m0)
        history['weights'].append(model.w.flatten().copy())
    return history

def train_adam(model, X, y, X_val, y_val, lr=0.001, epochs=100, beta1=0.9, beta2=0.
 history = {'loss': [], 'acc': [], 'val_loss': [], 'val_acc': [], 'time': [], 'weig
 m0 = time.time()
 m = np.zeros_like(model.w)
 v = np.zeros_like(model.w)
 t = 0
 for _ in range(epochs):
  t += 1
  y_pred = model.forward(X)
  grad = model.get_gradients(X, y, y_pred)
  m = beta1*m + (1-beta1)*grad
  v = beta2*v + (1-beta2)*(grad**2)
  m_hat = m / (1-beta1**t)
  v_hat = v / (1-beta2**t)
  model.w -= lr * m_hat / (np.sqrt(v_hat) + eps)
  train_pred = model.forward(X)
  val_pred = model.forward(X_val)
  history['loss'].append(model.loss(y, train_pred))
  history['acc'].append(model.accuracy(y, train_pred))
  history['val_loss'].append(model.loss(y_val, val_pred))
  history['val_acc'].append(model.accuracy(y_val, val_pred))
  history['time'].append(time.time() - m0)
  history['weights'].append(model.w.flatten().copy())
 return history
```