Yapay Zeka ve Makine Öğrenmesi Algoritmaları - Kapsamlı Teknik Doküman

1. MAKİNE ÖĞRENMESİ ALGORİTMALARI

1.1 Gözetimli Öğrenme Algoritmaları

1.1.1 Doğrusal Regresyon (Linear Regression)

Matematiksel Temel ve Çalışma Prensibi: Doğrusal regresyon, bağımlı değişken y ile bir veya daha fazla bağımsız değişken x arasındaki ilişkiyi modelleyen temel bir algoritmadır. Basit doğrusal regresyon için matematiksel form:

$y = wx + b$

Burada:

- y: Bağımlı değişken (tahmin edilecek değer)
- x: Bağımsız değişken
- w: Ağırlık katsayısı
- b: Bias terimi (y-eksenini kestiği nokta)

Çoklu doğrusal regresyon için: $y = w_1x_1 + w_2x_2 + ... + w_nx_n + b$

Optimizasyon hedefi, Ortalama Kare Hata (Mean Squared Error - MSE) fonksiyonunu minimize etmektir: $MSE = (1/n)\Sigma(y_i - \hat{y}_i)^2$

Öğrenme Süreci:

1. Parametrelerin (w ve b) rastgele başlatılması
2. Her iterasyonda:
    - Forward pass ile tahminlerin hesaplanması
    - MSE kaybının hesaplanması
    - Gradient descent ile parametrelerin güncellenmesi
3. Belirlenen epoch sayısı kadar veya yakınsama sağlanana kadar tekrar

Kullanım Alanları:

1. Finans Sektörü:
    - Hisse senedi fiyat tahmini
    - Gayrimenkul değerleme
    - Risk analizi
    - Kredi skorlama sistemleri
2. Perakende Sektörü:
    - Satış tahmini
    - Envanter optimizasyonu
    - Talep planlama
    - Fiyat optimizasyonu
3. Enerji Sektörü:
    - Enerji tüketim tahmini
    - Yenilenebilir enerji üretim tahmini

- Yük tahmini
- Verimlilik analizi

4. Sağlık Sektörü:

- İlaç dozaj optimizasyonu
- Hasta yatış süresi tahmini
- Sağlık harcamaları tahmini
- Epidemiyolojik tahminler

Avantajlar:

1. Hesaplama Verimliliği:

- Düşük hesaplama karmaşıklığı: O(n*d)
- Hızlı eğitim süresi
- Minimal bellek kullanımı
- Gerçek zamanlı uygulamalar için uygun

2. Yorumlanabilirlik:

- Şeffaf matematiksel model
- Kolay anlaşılabilir parametreler
- İş paydaşlarına açıklanması kolay
- Model davranışının öngörülebilirliği

3. Uygulama Kolaylığı:

- Basit implementasyon
- Az hiperparametre
- Kolay debug edilebilirlik
- Hızlı prototipleme imkanı

Dezavantajlar:

1. Model Kısıtlamaları:

- Yalnızca doğrusal ilişkileri modelleyebilme
- Karmaşık örüntüleri yakalayamama
- Doğrusal olmayan problemlerde yetersizlik
- Sınırlı öğrenme kapasitesi

2. Veri Hassasiyeti:

- Aykırı değerlere yüksek duyarlılık
- Gürültülü veriden fazla etkilenme
- Eksik veri durumunda performans düşüşü
- Özellikler arası korelasyondan etkilenme

3. Varsayımlar:

- Doğrusallık varsayımı
- Bağımsızlık varsayımı
- Normal dağılım varsayımı
- Homojenlik varsayımı

Maliyet Analizi:

1. Hesaplama Maliyeti:

- Eğitim Karmaşıklığı: O(n*d)
- Tahmin Karmaşıklığı: O(d)
- Bellek Kullanımı: O(d)
- İterasyon Başına Süre: Milisaniyeler

2. Donanım Gereksinimleri:
    - Minimum: 2GB RAM
    - Önerilen: 4GB RAM
    - CPU: Tek çekirdek yeterli
    - GPU: Gerekli değil
3. İnsan Kaynağı:
    - Yetkinlik Seviyesi: Junior
    - Öğrenme Süresi: 1-2 hafta
    - Geliştirme Süresi: 1-2 gün
    - Bakım Maliyeti: Düşük
4. Altyapı Maliyeti:
    - Veri Depolama: Minimal
    - İşlem Gücü: Düşük
    - Ağ Bandwidth: Minimal
    - Ölçeklendirme Maliyeti: Düşük

1.1.2 Lojistik Regresyon (Logistic Regression)

Matematiksel Temel ve Çalışma Prensibi: Lojistik regresyon, ikili sınıflandırma problemleri için kullanılan ve doğrusal regresyonun sigmoid fonksiyonu ile dönüştürülmüş halidir. Matematiksel form:

P(y=1|x) = 1 / (1 + e^(-wx - b))

Burada:

- P(y=1|x): Pozitif sınıf olasılığı
- x: Giriş özellikleri
- w: Ağırlık vektörü
- b: Bias terimi
- e: Euler sayısı

Kayıp fonksiyonu olarak Binary Cross Entropy kullanılır: L = -[y*log(p)* + *(1-y)*log(1-p)]

Öğrenme Süreci:

1. Parametrelerin rastgele başlatılması
2. Her iterasyonda:
    - Forward pass ile olasılıkların hesaplanması
    - Cross entropy kaybının hesaplanması
    - Gradient descent ile parametrelerin güncellenmesi
3. Yakınsama sağlanana kadar tekrar

Kullanım Alanları:

1. Finans Sektörü:
    - Kredi risk değerlendirmesi

- Dolandırıcılık tespiti
- Müşteri temerrüt tahmini
- Sigorta risk analizi

2. Sağlık Sektörü:

- Hastalık teşhisi
- Risk gruplarının belirlenmesi
- İlaç etkinlik tahmini
- Hasta triyaj sistemleri

3. Pazarlama:

- Müşteri kayıp tahmini
- E-posta kampanya optimizasyonu
- Reklam tıklanma tahmini
- Müşteri segmentasyonu

4. Güvenlik:

- Spam filtreleme
- Kötü amaçlı yazılım tespiti
- Anormal davranış tespiti
- Erişim kontrolü

Avantajlar:

1. Model Özellikleri:

- Olasılıksal çıktı
- Hızlı eğitim süresi
- Az bellek kullanımı
- İyi bir baseline model

2. Uygulama Kolaylığı:

- Basit implementasyon
- Az hiperparametre
- Kolay yorumlanabilirlik
- Hızlı iterasyon imkanı

3. Ölçeklenebilirlik:

- Online öğrenmeye uygunluk
- Dağıtık sistemlerde çalışabilme
- Yeni verilerle kolay güncelleme
- Düşük kaynak kullanımı

Dezavantajlar:

1. Model Kısıtlamaları:

- Doğrusal karar sınırı
- Karmaşık ilişkileri modelleyememe
- Özellikler arası etkileşimleri yakalayamama
- Sınırlı öğrenme kapasitesi

2. Veri Gereksinimleri:

- Dengeli veri seti ihtiyacı
- Özellik ölçeklendirme gerekliliği
- Çok sınıflı problemlerde yetersizlik
- Aykırı değerlere hassasiyet

Maliyet Analizi:

1. Hesaplama Maliyeti:
   - Eğitim Karmaşıklığı: O(n*d)
   - Tahmin Karmaşıklığı: O(d)
   - Bellek Kullanımı: O(d)
   - Batch İşleme: Verimli
2. Donanım Gereksinimleri:
   - Minimum: 4GB RAM
   - Önerilen: 8GB RAM
   - CPU: Çok çekirdek önerilir
   - GPU: Gerekli değil
3. İnsan Kaynağı:
   - Yetkinlik Seviyesi: Junior/Mid-level
   - Öğrenme Süresi: 2-3 hafta
   - Geliştirme Süresi: 2-3 gün
   - Bakım Maliyeti: Düşük-Orta
4. Altyapı Maliyeti:
   - Veri Depolama: Düşük
   - İşlem Gücü: Düşük-Orta
   - Monitoring: Basit
   - Deployment: Kolay

1.1.3 Destek Vektör Makineleri (Support Vector Machines - SVM)

Matematiksel Temel ve Çalışma Prensibi: SVM, veri noktaları arasında maksimum marjinli bir hiperdüzlem bularak sınıflandırma yapan bir algoritmadır. Temel matematiksel form:

f(x) = wx + b

Optimizasyon problemi: minimize (1/2)||w||² subject to y□(wx + b) ≥ 1 for all i

Kernel trick ile doğrusal olmayan dönüşümler: K(x,y) = φ(x)□φ(y)

Öğrenme Süreci:

1. Kernel fonksiyonu seçimi
2. Quadratic programming optimizasyonu
3. Support vektörlerinin belirlenmesi
4. Karar sınırının oluşturulması

Kullanım Alanları:

1. Biyoinformatik:
   - Protein sınıflandırma
   - Gen ifadesi analizi

- İlaç etkileşimi tahmini
- Hastalık teşhisi

2. Görüntü İşleme:
    - Yüz tanıma
    - Nesne tespiti
    - Doku analizi
    - Medikal görüntü sınıflandırma
3. Metin Madenciliği:
    - Duygu analizi
    - Spam filtreleme
    - Doküman sınıflandırma
    - Konu modelleme
4. Endüstriyel Uygulamalar:
    - Kalite kontrol
    - Arıza tespiti
    - Proses optimizasyonu
    - Enerji tahminleme

Avantajlar:

1. Performans:
    - Yüksek doğruluk
    - İyi genelleme
    - Aşırı öğrenmeye karşı dirençli
    - Gürültülü veriye dayanıklı
2. Esneklik:
    - Farklı kernel fonksiyonları
    - Doğrusal/doğrusal olmayan sınıflandırma
    - Çok boyutlu uzayda çalışabilme
    - Özellik dönüşümü imkanı
3. Teorik Temel:
    - Sağlam matematiksel altyapı
    - Garanti edilen optimum çözüm
    - Kontrol edilebilir komplekslik
    - İyi anlaşılmış parametreler

Dezavantajlar:

1. Hesaplama Maliyeti:
    - Yüksek eğitim süresi
    - Büyük veri setlerinde yavaş
    - Bellek yoğun işlemler
    - Ölçeklenme zorlukları
2. Parametre Optimizasyonu:
    - Kernel seçimi zorluğu

- Hiperparametre hassasiyeti
- Grid search gereksinimi
- Cross-validation ihtiyacı

3. Pratik Kısıtlamalar:

- Probabilistik çıktı eksikliği
- Online öğrenme zorluğu
- Çok sınıflı problemlerde karmaşıklık
- Büyük veri setlerinde uygulanabilirlik

Maliyet Analizi:

1. Hesaplama Maliyeti:

- Eğitim Karmaşıklığı: $O(n^2)$ - $O(n^3)$
- Tahmin Karmaşıklığı: $O(n\_sv)$
- Bellek Kullanımı: $O(n^2)$
- Kernel hesaplamaları: Yoğun

2. Donanım Gereksinimleri:

- Minimum: 16GB RAM
- Önerilen: 32GB RAM
- CPU: Çok çekirdekli işlemci
- GPU: Büyük veri setleri için önerilen

3. İnsan Kaynağı:

- Yetkinlik Seviyesi: Senior
- Öğrenme Süresi: 1-2 ay
- Geliştirme Süresi: 1-2 hafta
- Bakım Maliyeti: Orta-Yüksek

4. Altyapı Maliyeti:

- Veri Depolama: Yüksek
- İşlem Gücü: Yüksek
- Paralel İşleme: Gerekli
- Ölçeklendirme: Maliyetli

1.1.4 Karar Ağaçları (Decision Trees)

Matematiksel Temel ve Çalışma Prensibi: Karar ağaçları, veriyi özelliklerine göre bölerek sınıflandırma veya regresyon yapan hiyerarşik bir modeldir. Bölünme kriterleri:

- Gini İndeksi: $1 - \Sigma p_i^2$
- Entropi: $-\Sigma p_i \log_2(p_i)$
- Bilgi Kazancı: $Entropi(parent) - \Sigma(w_i * Entropi(child_i))$

Öğrenme Süreci:

1. Kök düğümden başlama
2. Her düğümde:
   - En iyi bölünme özelliğini bulma
   - Veriyi alt gruplara ayırma
   - Durma kriterleri kontrolü

3. Yaprak düğümlere ulaşana kadar tekrar

Kullanım Alanları:

1. Bankacılık:
    - Kredi değerlendirmesi
    - Müşteri segmentasyonu
    - Risk analizi
    - Portföy yönetimi
2. Sağlık:
    - Teşhis sistemleri
    - Tedavi planlaması
    - İlaç etkileşimi analizi
    - Hasta risk değerlendirmesi
3. Perakende:
    - Müşteri davranış analizi
    - Ürün önerileri
    - Stok optimizasyonu
    - Kampanya hedefleme
4. Üretim:
    - Kalite kontrol
    - Bakım planlaması
    - Verimlilik analizi
    - Hata tespiti

Avantajlar:

1. Yorumlanabilirlik:
    - Görsel temsil imkanı
    - Açık karar kuralları
    - Kolay anlaşılabilirlik
    - İş paydaşlarına açıklanabilirlik
2. Kullanım Kolaylığı:
    - Az ön işleme gereksinimi
    - Otomatik özellik seçimi
    - Kategorik/sayısal veri uyumluluğu
    - Eksik veri toleransı
3. Esneklik:
    - Sınıflandırma/regresyon yapabilme
    - Doğrusal olmayan ilişkileri öğrenebilme
    - Özellik etkileşimlerini yakalayabilme
    - Farklı veri tiplerine uygunluk

Dezavantajlar:

1. Model Kararlılığı:
    - Aşırı öğrenme eğilimi

- Veri değişimlerine hassasiyet
- Eğitim verisine bağımlılık
- Kararsız yapı

2. Optimizasyon Zorlukları:

- Optimal ağaç bulmanın NP-complete olması
- Budama parametrelerinin hassasiyeti
- Lokal optimumlara takılma
- Dengesiz ağaç yapıları

3. Performans Limitleri:

- Keskin karar sınırları
- Sürekli fonksiyonları yaklaşımda zorluk
- Yüksek varyans
- Sınırlı genelleme yeteneği

Maliyet Analizi:

1. Hesaplama Maliyeti:

- Eğitim Karmaşıklığı: O(d*n*log(n))
- Tahmin Karmaşıklığı: O(log(n))
- Bellek Kullanımı: O(n)
- Ağaç Derinliği: O(log(n))

2. Donanım Gereksinimleri:

- Minimum: 8GB RAM
- Önerilen: 16GB RAM
- CPU: Çok çekirdek önerilir
- GPU: Gerekli değil

3. İnsan Kaynağı:

- Yetkinlik Seviyesi: Junior/Mid-level
- Öğrenme Süresi: 2-3 hafta
- Geliştirme Süresi: 3-5 gün
- Bakım Maliyeti: Orta

4. Altyapı Maliyeti:

- Veri Depolama: Orta
- İşlem Gücü: Orta
- Deployment: Kolay
- Ölçeklendirme: Orta maliyetli

1.2 Gözetimsiz Öğrenme Algoritmaları

1.2.1 K-Means Kümeleme

Matematiksel Temel ve Çalışma Prensibi: K-Means, veri noktalarını K adet kümeye ayıran iteratif bir algoritmadır. Optimizasyon hedefi:

minimize Σ□□□□ Σ□□□□C□ ||x□□ - μ□||²

Burada:

- k: Küme sayısı

- C_i: i. küme
- μ_i: i. kümenin merkezi
- x_{ij}: C_i kümesindeki j. nokta

Öğrenme Süreci:

1. K adet merkez noktası rastgele seçimi
2. Her iterasyonda:
    - Her noktayı en yakın merkeze atama
    - Küme merkezlerini güncelleme
3. Yakınsama sağlanana kadar tekrar

Kullanım Alanları:

1. Müşteri Segmentasyonu:
    - Davranış analizi
    - Hedef kitle belirleme
    - Kişiselleştirme
    - Pazarlama stratejileri
2. Görüntü İşleme:
    - Renk kuantalama
    - Görüntü segmentasyonu
    - Özellik çıkarımı
    - Görüntü sıkıştırma
3. Anomali Tespiti:
    - Dolandırıcılık tespiti
    - Sistem arızaları
    - Network güvenliği
    - Kalite kontrol
4. Belge Organizasyonu:
    - Doküman kümeleme
    - Konu modelleme
    - İçerik organizasyonu
    - Arama optimizasyonu

Avantajlar:

1. Uygulama Kolaylığı:
    - Basit implementasyon
    - Hızlı eğitim
    - Kolay anlaşılabilirlik
    - Az parametre
2. Ölçeklenebilirlik:
    - Büyük veri setlerine uygunluk
    - Paralel işleme imkanı
    - Online versiyonu mevcut
    - Düşük bellek kullanımı
3. Esneklik:

- Farklı uzaklık metriklerine uygunluk
- Çeşitli başlatma stratejileri
- Adaptif K seçimi imkanı
- Mini-batch versiyonu

Dezavantajlar:

1. Başlatma Hassasiyeti:
   - Lokal optimumlara takılma
   - Rastgele başlatmaya bağımlılık
   - K değeri seçimi zorluğu
   - Kararsız sonuçlar
2. Küme Kısıtlamaları:
   - Küresel küme varsayımı
   - Yoğunluk bazlı kümeleri yakalayamama
   - Gürültüye hassasiyet
   - Dengesiz küme boyutları
3. Veri Gereksinimleri:
   - Özellik ölçeklendirme ihtiyacı
   - Kategorik veri zorluğu
   - Eksik veri problemi
   - Boyut laneti

Maliyet Analizi:

1. Hesaplama Maliyeti:
   - Eğitim Karmaşıklığı: O(n*kd*i)
   - Tahmin Karmaşıklığı: O(k*d)
   - Bellek Kullanımı: O(n + k)
   - İterasyon Sayısı: Genelde <100
2. Donanım Gereksinimleri:
   - Minimum: 8GB RAM
   - Önerilen: 16GB RAM
   - CPU: Çok çekirdek önerilir
   - GPU: Büyük veri setleri için faydalı
3. İnsan Kaynağı:
   - Yetkinlik Seviyesi: Junior/Mid-level
   - Öğrenme Süresi: 1-2 hafta
   - Geliştirme Süresi: 2-3 gün
   - Bakım Maliyeti: Düşük-Orta
4. Altyapı Maliyeti:
   - Veri Depolama: Orta
   - İşlem Gücü: Orta
   - Paralel İşleme: Opsiyonel
   - Ölçeklendirme: Orta maliyetli

2. DERİN ÖĞRENME ALGORİTMALARI

2.1 Evrişimli Sinir Ağları (Convolutional Neural Networks - CNN)

Matematiksel Temel ve Çalışma Prensibi

Evrişimli Sinir Ağları, görsel verilerin hiyerarşik özellik çıkarımını gerçekleştiren derin öğrenme modelidir. Temel matematiksel işlem olan konvolüsyon şu şekilde ifade edilir:

(f * g)(x,y) = ∑∑ f(i,j)g(x-i,y-j)

Burada:

- f: Giriş görüntüsü
- g: Konvolüsyon filtresi (kernel)
- x,y: Çıkış koordinatları
- i,j: Filtre koordinatları

Mimari Bileşenler:

1. Konvolüsyon Katmanları:
    - Özellik haritaları oluşturma
    - Uzamsal ilişkileri yakalama
    - Parametre paylaşımı
    - Yerel bağlantılar
2. Havuzlama Katmanları:
    - Uzamsal boyut azaltma
    - Pozisyon değişmezliği
    - Aşırı öğrenmeyi azaltma
    - Hesaplama verimliliği
3. Tam Bağlantılı Katmanlar:
    - Yüksek seviye özellikleri birleştirme
    - Son sınıflandırma/regresyon
    - Global bağlam oluşturma

Öğrenme Süreci

1. İleri Yayılım:

```
def forward_pass(input_image, layers):
    # Konvolüsyon katmanları
    for conv_layer in layers.conv:
        features = conv2d(input_image, conv_layer.filters)
        features = activate(features, 'relu')
        features = max_pool(features, size=2)

    # Tam bağlantılı katmanlar
    flattened = flatten(features)
```

```
    output = dense(flattened, weights)
    return softmax(output)
```

2. Geri Yayılım:
    - Gradyan hesaplama
    - Ağırlık güncellemesi
    - Optimizasyon algoritması (SGD, Adam vb.)

Kullanım Alanları

1. Tıbbi Görüntü Analizi:
    - Tümör tespiti
    - Organ segmentasyonu
    - Hastalık teşhisi
    - Radyoloji raporlaması
2. Otonom Sistemler:
    - Sürücüsüz araçlar
    - Robot navigasyonu
    - Drone kontrolü
    - Endüstriyel otomasyon
3. Güvenlik Sistemleri:
    - Yüz tanıma
    - Nesne tespiti
    - Anormal davranış tespiti
    - Erişim kontrolü
4. Üretim ve Kalite Kontrol:
    - Ürün kusur tespiti
    - Montaj hattı kontrolü
    - Paketleme kontrolü
    - Boyut/renk kontrolü

Maliyet Analizi

1. Hesaplama Maliyeti:
    - Eğitim Karmaşıklığı: $O(ndfke)$
        - n: Veri sayısı
        - d: Giriş boyutu
        - f: Filtre sayısı
        - k: Kernel boyutu
        - e: Epoch sayısı
    - Tahmin Karmaşıklığı: $O(dfk)$
    - GPU Bellek Kullanımı:
        - Minimum: 8GB VRAM
        - Önerilen: 16GB+ VRAM
    - Batch Size Etkisi:

        - Küçük batch: Düşük bellek, yavaş eğitim
        - Büyük batch: Yüksek bellek, hızlı eğitim

2. Donanım Gereksinimleri:
    - GPU:
        - Minimum: NVIDIA GTX 1660
        - Önerilen: NVIDIA RTX 3080 veya üstü
    - CPU:
        - Minimum: 6 çekirdek
        - Önerilen: 12+ çekirdek
    - RAM:
        - Minimum: 16GB
        - Önerilen: 32GB+
    - Depolama:
        - SSD: 500GB+
        - Veri seti boyutuna göre ek depolama

3. İnsan Kaynağı:
    - Yetkinlik Seviyesi:
        - Deep Learning uzmanı
        - Computer Vision deneyimi
        - Python/PyTorch/TensorFlow
        - GPU programlama bilgisi
    - Geliştirme Süresi:
        - Prototip: 2-4 hafta
        - Prodüksiyon: 2-3 ay
        - Optimizasyon: 1-2 ay
    - Bakım:
        - Model güncelleme: Aylık
        - Performans izleme: Günlük
        - Veri kalite kontrolü: Haftalık

4. Altyapı Maliyeti:
    - Cloud GPU (aylık):
        - AWS p3.2xlarge: $3.06/saat
        - Google Cloud V100: $2.48/saat
        - Azure NC6s_v3: $3.00/saat
    - Veri Depolama (aylık):
        - S3: $0.023/GB
        - Cloud Storage: $0.020/GB
        - Azure Blob: $0.0184/GB
    - Ağ Bandwidth:

- Veri transfer: $0.08-0.12/GB
- API çağrıları: $0.40/1000 istek

Avantajlar

1. Özellik Çıkarımı:
    - Otomatik özellik öğrenme
    - Hiyerarşik temsil
    - Uzamsal ilişkileri yakalama
    - Transfer öğrenme imkanı
2. Performans:
    - Yüksek doğruluk
    - Gürültüye dayanıklılık
    - Pozisyon değişmezliği
    - Ölçeklenebilirlik
3. Esneklik:
    - Farklı veri tipleri
    - Çeşitli mimari varyasyonları
    - Modüler yapı
    - Fine-tuning imkanı

Dezavantajlar

1. Kaynak Gereksinimleri:
    - Yüksek hesaplama gücü
    - Büyük veri seti ihtiyacı
    - GPU gereksinimi
    - Yüksek enerji tüketimi
2. Eğitim Zorlukları:
    - Hiperparametre optimizasyonu
    - Gradyan problemleri
    - Aşırı öğrenme riski
    - Uzun eğitim süreleri
3. Yorumlanabilirlik:
    - Kara kutu model
    - Karar sürecinin belirsizliği
    - Hata analizi zorluğu
    - Güven skorlama zorluğu

2.2 Tekrarlayan Sinir Ağları (Recurrent Neural Networks - RNN)

Matematiksel Temel ve Çalışma Prensibi

Tekrarlayan Sinir Ağları, sıralı verilerdeki zamansal bağımlılıkları modelleyen derin öğrenme mimarisidir. Temel matematiksel form:

$h_t = \tanh(Wx_t + Uh_{t-1} + b)$ $y_t = \text{softmax}(Vh_t + c)$

Burada:

- x_t: t anındaki giriş
- h_t: Gizli durum
- y_t: Çıkış
- W,U,V: Ağırlık matrisleri
- b,c: Bias terimleri

RNN Varyantları ve Mimariler

LSTM (Long Short-Term Memory)

LSTM, uzun vadeli bağımlılıkları öğrenebilmek için özel bir hafıza hücresi ve kapı mekanizmaları kullanan gelişmiş bir RNN mimarisidir. LSTM'in matematiksel formülasyonu şu şekildedir:

$f_t = \sigma(W_f[h_{t-1}, x_t] + b_f)$ # Unut Kapısı $i_t = \sigma(W_i[h_{t-1}, x_t] + b_i)$ # Giriş Kapısı $\tilde{c}_t = \tanh(W_c[h_{t-1}, x_t] + b_c)$ # Aday Hafıza $c_t = f_t * c_{t-1} + i_t * \tilde{c}_t$ # Hafıza Güncelleme $o_t = \sigma(W_o[h_{t-1}, x_t] + b_o)$ # Çıkış Kapısı $h_t = o_t * \tanh(c_t)$ # Gizli Durum

Burada:

- x_t: Giriş vektörü
- h_t: Gizli durum
- c_t: Hücre durumu
- f_t, i_t, o_t: Kapı kontrolleri
- W: Ağırlık matrisleri
- b: Bias terimleri
- σ: Sigmoid fonksiyonu

LSTM'in temel çalışma prensiplerini bir örnek üzerinden inceleyelim:

```
def lstm_cell_forward(x_t, h_prev, c_prev, parameters):
    """
    Tek bir LSTM hücresinin ileri yayılımı

    Parametreler:
    x_t -- Giriş vektörü, boyut: (n_x, 1)
    h_prev -- Önceki gizli durum, boyut: (n_h, 1)
    c_prev -- Önceki hücre durumu, boyut: (n_h, 1)
    parameters -- Ağırlık ve bias parametreleri

    Dönüş:
    h_next -- Sonraki gizli durum
    c_next -- Sonraki hücre durumu
    cache -- Geri yayılım için ara değerler
    """
    # Kapı hesaplamaları
    concat = np.concatenate((h_prev, x_t))

    ft = sigmoid(np.dot(parameters['Wf'], concat) + parameters['bf'])
    it = sigmoid(np.dot(parameters['Wi'], concat) + parameters['bi'])
    cct = np.tanh(np.dot(parameters['Wc'], concat) + parameters['bc'])
    ot = sigmoid(np.dot(parameters['Wo'], concat) + parameters['bo'])
```

```
    # Hafıza ve durum güncellemeleri
    c_next = ft * c_prev + it * cct
    h_next = ot * np.tanh(c_next)

    return h_next, c_next, cache
```

GRU (Gated Recurrent Unit)

GRU, LSTM'in daha basit bir versiyonu olarak tasarlanmış, ancak benzer performans gösteren bir RNN varyantıdır. GRU'nun matematiksel formülasyonu:

z_t = σ(Wz[h_{t-1}, x_t] + bz) # Güncelleme Kapısı r_t = σ(Wr[h_{t-1}, x_t] + br) # Sıfırlama Kapısı h_t = tanh(W[r_t * h_{t-1}, x_t]) # Aday Gizli Durum h_t = (1 - z_t) * h_{t-1} + z_t * h_t # Gizli Durum Güncelleme

Kullanım Alanları ve Uygulamalar

1. Doğal Dil İşleme:
    - Metin Üretimi: Dil modelleri geliştirilmesi ve metin oluşturma
    - Makine Çevirisi: Diller arası çeviri sistemleri
    - Duygu Analizi: Metin duygusunun belirlenmesi
    - Soru Cevaplama: Doğal dil anlama ve yanıt üretme
2. Zaman Serisi Analizi:
    - Finansal Tahmin: Hisse senedi fiyat tahminleri
    - Sensor Verileri: IoT cihazlarından gelen veri analizi
    - Enerji Tüketimi: Elektrik yük tahmini
    - Hava Durumu: Meteorolojik tahminler
3. Ses İşleme:
    - Konuşma Tanıma: Ses-metin dönüşümü
    - Müzik Üretimi: Otomatik beste oluşturma
    - Ses Sentezi: Metin-ses dönüşümü
    - Ses Ayrıştırma: Kaynak ayırma
4. Biyoinformatik:
    - Protein Yapı Tahmini
    - Gen Dizilimi Analizi
    - İlaç Etkileşimi Tahmini
    - Hastalık Prognozu

Maliyet Analizi

1. Hesaplama Maliyeti:
    - Eğitim Karmaşıklığı:
        - LSTM: O(nd^2s)
        - GRU: O(nd^2s*0.75) Burada:
        - n: Batch boyutu
        - d: Gizli durum boyutu
        - s: Sekans uzunluğu

- Bellek Kullanımı:
  - LSTM: $O(4nd^2)$
  - GRU: $O(3nd^2)$

2. Donanım Gereksinimleri:
   - GPU:
     - Minimum: NVIDIA GTX 1660 6GB
     - Önerilen: NVIDIA RTX 3080 12GB
   - RAM:
     - Minimum: 16GB
     - Önerilen: 32GB+
   - CPU:
     - Minimum: 8 çekirdek
     - Önerilen: 16+ çekirdek

3. İnsan Kaynağı:
   - Yetkinlik Seviyesi:
     - Derin Öğrenme Uzmanı (Senior)
     - NLP/Sekans Modelleme Deneyimi
     - Python/PyTorch/TensorFlow
     - Optimizasyon Tecrübesi
   - Geliştirme Süreci:
     - Mimari Tasarım: 2-3 hafta
     - İlk Prototip: 1-2 ay
     - Optimizasyon: 1-2 ay
     - Prodüksiyon: 2-3 ay

4. Altyapı Maliyeti:
   - Cloud Servisleri (Aylık):
     - AWS p3.2xlarge: ~$2500
     - Google Cloud V100: ~$2000
     - Azure NC6s_v3: ~$2300
   - Veri Depolama:
     - Eğitim Verisi: 100GB-1TB
     - Model Checkpoints: 10-50GB
     - Çıktı Logları: 5-20GB

Optimizasyon Teknikleri

1. Mimari Optimizasyonları:
   - Attention Mekanizmaları
   - Bidirectional RNN
   - Residual Connections

- Layer Normalization

2. Eğitim Optimizasyonları:

- Gradient Clipping
- Dropout
- Batch Normalization
- Learning Rate Scheduling

```
class OptimizedLSTM(nn.Module):
    def __init__(self, input_size, hidden_size, num_layers):
        super().__init__()
        self.lstm = nn.LSTM(
            input_size=input_size,
            hidden_size=hidden_size,
            num_layers=num_layers,
            batch_first=True,
            bidirectional=True,  # Çift yönlü LSTM
            dropout=0.2  # Dropout eklendi
        )
        self.norm = nn.LayerNorm(hidden_size * 2)  # Layer normalization

    def forward(self, x, h0=None):
        output, (hn, cn) = self.lstm(x, h0)
        output = self.norm(output)  # Normalizasyon uygulanıyor
        return output, (hn, cn)
```

Performans Metrikleri ve İzleme

1. Model Değerlendirme:

- Perplexity (Dil Modelleri için)
- BLEU Score (Çeviri için)
- F1 Score (Sınıflandırma için)
- MAE/RMSE (Regresyon için)

2. Kaynak Kullanımı:

- GPU Kullanımı ve Verimlilik
- Bellek Tüketimi
- Throughput ve Latency
- Batch İşleme Performansı

```
def calculate_metrics(model, data_loader, device):
    metrics = {
        'loss': 0.0,
        'perplexity': 0.0,
        'gpu_utilization': [],
        'memory_usage': [],
        'inference_time': []
    }

    model.eval()
    with torch.no_grad():
```

```
        for batch in data_loader:
            start_time = time.time()

            # GPU kullanımı ölçümü
            gpu_util = get_gpu_utilization()
            mem_usage = get_memory_usage()

            # Model tahmini
            outputs = model(batch)
            loss = criterion(outputs, batch['targets'])

            # Metrikleri güncelle
            metrics['loss'] += loss.item()
            metrics['perplexity'] += torch.exp(loss).item()
            metrics['gpu_utilization'].append(gpu_util)
            metrics['memory_usage'].append(mem_usage)
            metrics['inference_time'].append(time.time() - start_time)

    return metrics
```

2.3 Transformer Mimarisi ve Modern Dil Modelleri

Transformer Mimarisi

Matematiksel Temel ve Çalışma Prensibi

Transformer mimarisi, öz-dikkat (self-attention) mekanizmasını kullanarak paralel işleme yapabilen ve uzun mesafeli bağımlılıkları yakalayabilen bir derin öğrenme modelidir. Matematiksel olarak, dikkat mekanizması şu şekilde ifade edilir:

Attention(Q, K, V) = softmax(QK^T/√dk)V

Burada:

- Q: Sorgu matrisi (Query)
- K: Anahtar matrisi (Key)
- V: Değer matrisi (Value)
- dk: Anahtar vektörlerinin boyutu

Çok Başlı Dikkat (Multi-Head Attention) mekanizması: MultiHead(Q,K,V) = Concat(head□,...,head□)W^O where head□ = Attention(QW^Q_i, KW^K_i, VW^V_i)

Mimari Bileşenler

1. Encoder Bileşenleri:

```
class EncoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionwiseFeedForward(d_model, d_ff)
```

```
        self.norm1 = LayerNorm(d_model)
        self.norm2 = LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, mask):
        # Öz-dikkat katmanı
        attn_output = self.self_attn(x, x, x, mask)
        x = self.norm1(x + self.dropout(attn_output))

        # Feed-forward katmanı
        ff_output = self.feed_forward(x)
        x = self.norm2(x + self.dropout(ff_output))
        return x
```

2. Decoder Bileşenleri:

```
class DecoderLayer(nn.Module):
    def __init__(self, d_model, num_heads, d_ff, dropout):
        super().__init__()
        self.self_attn = MultiHeadAttention(d_model, num_heads)
        self.cross_attn = MultiHeadAttention(d_model, num_heads)
        self.feed_forward = PositionwiseFeedForward(d_model, d_ff)
        self.norm1 = LayerNorm(d_model)
        self.norm2 = LayerNorm(d_model)
        self.norm3 = LayerNorm(d_model)
        self.dropout = nn.Dropout(dropout)

    def forward(self, x, enc_output, src_mask, tgt_mask):
        # Öz-dikkat katmanı
        self_attn_output = self.self_attn(x, x, x, tgt_mask)
        x = self.norm1(x + self.dropout(self_attn_output))

        # Encoder-Decoder dikkat katmanı
        cross_attn_output = self.cross_attn(x, enc_output, enc_output,
src_mask)
        x = self.norm2(x + self.dropout(cross_attn_output))

        # Feed-forward katmanı
        ff_output = self.feed_forward(x)
        x = self.norm3(x + self.dropout(ff_output))
        return x
```

Modern Dil Modelleri

BERT (Bidirectional Encoder Representations from Transformers)

BERT, çift yönlü bağlam anlamayı hedefleyen bir dil modelidir. Temel özellikleri:

1. Ön Eğitim Görevleri:
    - Masked Language Modeling (MLM)
    - Next Sentence Prediction (NSP)
2. Model Mimarisi:

```
class BERT(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_layers, num_heads):
        super().__init__()
        self.embedding = BERTEmbedding(vocab_size, hidden_size)
        self.encoder = Encoder(
            num_layers=num_layers,
            d_model=hidden_size,
            num_heads=num_heads
        )
        self.mlm_head = MLMHead(hidden_size, vocab_size)
        self.nsp_head = NSPHead(hidden_size)

    def forward(self, input_ids, attention_mask, token_type_ids):
        # Embedding katmanı
        embeddings = self.embedding(input_ids, token_type_ids)

        # Encoder katmanları
        encoded = self.encoder(embeddings, attention_mask)

        # MLM ve NSP çıktıları
        mlm_output = self.mlm_head(encoded)
        nsp_output = self.nsp_head(encoded[:, 0, :])  # [CLS] token

        return mlm_output, nsp_output
```

GPT (Generative Pre-trained Transformer)

GPT, tek yönlü (autoregressive) dil modellemesi yapan bir mimaridir. Temel özellikleri:

1. Otomatik Regresif Dil Modellemesi:

```
class GPT(nn.Module):
    def __init__(self, vocab_size, hidden_size, num_layers, num_heads):
        super().__init__()
        self.embedding = GPTEmbedding(vocab_size, hidden_size)
        self.decoder = Decoder(
            num_layers=num_layers,
            d_model=hidden_size,
            num_heads=num_heads
        )
        self.lm_head = nn.Linear(hidden_size, vocab_size)

    def forward(self, input_ids, attention_mask=None):
        # Embedding katmanı
        embeddings = self.embedding(input_ids)

        # Decoder katmanları
        decoded = self.decoder(embeddings, attention_mask)

        # Dil modeli çıktısı
        lm_logits = self.lm_head(decoded)
```

```
        return lm_logits

    def generate(self, input_ids, max_length):
        for _ in range(max_length):
            # Mevcut girdiden tahmin
            outputs = self.forward(input_ids)
            next_token = outputs[:, -1, :].argmax(dim=-1)

            # Yeni tokeni ekle
            input_ids = torch.cat([input_ids, next_token.unsqueeze(-1)],
dim=-1)

        return input_ids
```

Kullanım Alanları

1. Doğal Dil İşleme:
    - Metin Sınıflandırma
    - Soru Cevaplama
    - Makine Çevirisi
    - Özetleme
    - Duygu Analizi
    - Varlık Tanıma
2. Kod Üretimi ve Analizi:
    - Kod Tamamlama
    - Hata Tespiti
    - Kod Dönüşümü
    - Dokümantasyon Oluşturma
3. Multimodal Uygulamalar:
    - Görüntü Açıklama
    - Video Anlama
    - Ses-Metin Dönüşümü
    - Çoklu Modal Füzyon

Maliyet Analizi

1. Hesaplama Maliyeti:
    - Eğitim Karmaşıklığı: $O(n^2*l)$
        - n: Sekans uzunluğu
        - l: Katman sayısı
    - Bellek Kullanımı: $O(n^2)$
    - Dikkat Matrisi: n x n
    - Parametre Sayısı:
        - BERT-base: 110M
        - GPT-3: 175B

2. Donanım Gereksinimleri:
    - GPU:
        - Eğitim: Minimum 8x NVIDIA A100
        - Çıkarım: NVIDIA T4 veya üstü
    - RAM:
        - Eğitim: 256GB+
        - Çıkarım: 32GB+
    - Depolama:
        - Model Parametreleri: 10GB-1TB
        - Eğitim Verisi: 1TB+
3. İnsan Kaynağı:
    - Uzmanlık:
        - Derin Öğrenme (Senior)
        - NLP/Transformers
        - Dağıtık Sistemler
        - GPU Optimizasyonu
    - Geliştirme Süreci:
        - Mimari Tasarım: 2-3 ay
        - Ön Eğitim: 3-6 ay
        - Fine-tuning: 1-2 ay
        - Optimizasyon: 2-3 ay
4. Altyapı Maliyeti:
    - Cloud GPU Kümeleri:
        - AWS p4d.24xlarge: $32.77/saat
        - Google Cloud A2: $25.6/saat
    - Veri Depolama:
        - Eğitim Verisi: $1000-5000/ay
        - Model Depolama: $500-2000/ay
    - Ağ Maliyeti:
        - Veri Transferi: $0.08-0.12/GB
        - API Çağrıları: $0.50-1.00/1000 istek

Optimizasyon Teknikleri

1. Model Paralelleştirme:

```
class ModelParallel(nn.Module):
    def __init__(self, num_gpus):
        super().__init__()
        self.num_gpus = num_gpus
```

```
        # Katmanları GPU'lara dağıt
        self.layers = nn.ModuleList([
            EncoderLayer().to(f'cuda:{i % num_gpus}')
            for i in range(num_layers)
        ])

    def forward(self, x):
        for i, layer in enumerate(self.layers):
            # Veriyi ilgili GPU'ya taşı
            gpu_id = i % self.num_gpus
            x = x.to(f'cuda:{gpu_id}')
            x = layer(x)
        return x
```

2. Bellek Optimizasyonu:
    - Gradient Checkpointing
    - Mixed Precision Training
    - Efficient Attention
    - Flash Attention
3. Hızlandırma Teknikleri:
    - KV Cache
    - Beam Search
    - Top-k/Top-p Sampling
    - Tensorboard Optimizasyonu

3. PEKİŞTİRMELİ ÖĞRENME ALGORİTMALARI

3.1 Q-Öğrenme (Q-Learning)

Matematiksel Temel

Q-öğrenme, model bağımsız bir pekiştirmeli öğrenme algoritmasıdır. Bellman denklemi:

Q(s,a) = r + γ * max[Q(s',a')]

Burada:

- s: Mevcut durum
- a: Eylem
- r: Ödül
- γ: İndirim faktörü
- s': Sonraki durum
- a': Sonraki eylem

Algoritma İmplementasyonu

```
class QLearning:
    def __init__(self, states, actions, learning_rate=0.1, discount=0.95):
        self.q_table = np.zeros((states, actions))
        self.lr = learning_rate
        self.gamma = discount
```

```
    def update(self, state, action, reward, next_state):
        old_value = self.q_table[state, action]
        next_max = np.max(self.q_table[next_state])

        # Q-değeri güncelleme
        new_value = (1 - self.lr) * old_value + \
                    self.lr * (reward + self.gamma * next_max)

        self.q_table[state, action] = new_value

    def get_action(self, state, epsilon=0.1):
        # Epsilon-greedy politikası
        if np.random.random() < epsilon:
            return np.random.randint(self.q_table.shape[1])
        else:
            return np.argmax(self.q_table[state])
```

3.2 Deep Q-Networks (DQN)

Mimari ve Bileşenler

```
class DQN(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(state_dim, 128),
            nn.ReLU(),
            nn.Linear(128, 128),
            nn.ReLU(),
            nn.Linear(128, action_dim)
        )

    def forward(self, state):
        return self.network(state)
```

Experience Replay

```
class ReplayBuffer:
    def __init__(self, capacity):
        self.buffer = deque(maxlen=capacity)

    def push(self, state, action, reward, next_state, done):
        self.buffer.append((state, action, reward, next_state, done))

    def sample(self, batch_size):
        transitions = random.sample(self.buffer, batch_size)
        state, action, reward, next_state, done = zip(*transitions)
        return (
            torch.FloatTensor(state),
            torch.LongTensor(action),
```

```
            torch.FloatTensor(reward),
            torch.FloatTensor(next_state),
            torch.FloatTensor(done)
        )
```

3.3 İleri Seviye Pekiştirmeli Öğrenme Algoritmaları

Policy Gradient Yöntemleri

REINFORCE Algoritması

REINFORCE, Monte Carlo policy gradient yaklaşımını kullanan temel bir politika optimizasyon algoritmasıdır. Matematiksel olarak şu şekilde ifade edilir:

∇θJ(θ) = E[∇θ log π_θ(a|s) * G_t]

Burada:

- θ: Politika parametreleri
- π_θ: Parametrize edilmiş politika
- G_t: Toplam ödül (return)

```
class PolicyNetwork(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super().__init__()
        self.network = nn.Sequential(
            nn.Linear(input_dim, hidden_dim),
            nn.Tanh(),
            nn.Linear(hidden_dim, output_dim),
            nn.Softmax(dim=-1)
        )

    def forward(self, state):
        return self.network(state)

class REINFORCE:
    def __init__(self, state_dim, action_dim, lr=0.01):
        self.policy = PolicyNetwork(state_dim, 128, action_dim)
        self.optimizer = optim.Adam(self.policy.parameters(), lr=lr)

    def select_action(self, state):
        state = torch.FloatTensor(state)
        probs = self.policy(state)
        action_dist = Categorical(probs)
        action = action_dist.sample()

        # Log olasılığını gradient hesabı için sakla
        self.saved_log_probs.append(action_dist.log_prob(action))
        return action.item()
```

```
    def update(self, rewards):
        # Monte Carlo ödül hesaplama
        returns = []
        G = 0
        for r in reversed(rewards):
            G = r + 0.99 * G
            returns.insert(0, G)
        returns = torch.FloatTensor(returns)

        # Politika gradyanı hesaplama ve güncelleme
        policy_loss = []
        for log_prob, R in zip(self.saved_log_probs, returns):
            policy_loss.append(-log_prob * R)

        self.optimizer.zero_grad()
        policy_loss = torch.cat(policy_loss).sum()
        policy_loss.backward()
        self.optimizer.step()
```

Proximal Policy Optimization (PPO)

PPO, politika optimizasyonunu güvenli bir şekilde yapan modern bir algoritmadır. Temel matematiksel form:

L_CLIP(θ) = E[min(r_t(θ)A_t, clip(r_t(θ), 1-ε, 1+ε)A_t)]

Burada:

- r_t(θ): Olasılık oranı
- A_t: Avantaj tahmini
- ε: Kırpma parametresi

```
class PPONetwork(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        self.actor = nn.Sequential(
            nn.Linear(state_dim, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, action_dim),
            nn.Softmax(dim=-1)
        )

        self.critic = nn.Sequential(
            nn.Linear(state_dim, 64),
            nn.Tanh(),
            nn.Linear(64, 64),
            nn.Tanh(),
            nn.Linear(64, 1)
        )

    def forward(self, state):
```

```
        return self.actor(state), self.critic(state)

class PPO:
    def __init__(self, state_dim, action_dim, lr=3e-4, gamma=0.99, eps_clip=0.2):
        self.network = PPONetwork(state_dim, action_dim)
        self.optimizer = optim.Adam(self.network.parameters(), lr=lr)
        self.gamma = gamma
        self.eps_clip = eps_clip

    def select_action(self, state):
        state = torch.FloatTensor(state)
        action_probs, state_value = self.network(state)

        dist = Categorical(action_probs)
        action = dist.sample()

        return action.item(), dist.log_prob(action), state_value

    def update(self, states, actions, old_probs, rewards, advantages):
        # PPO güncelleme döngüsü
        for _ in range(10):  # Çoklu güncelleme epoch'ları
            action_probs, state_values = self.network(states)
            dist = Categorical(action_probs)
            new_probs = dist.log_prob(actions)

            # Olasılık oranı hesaplama
            ratio = torch.exp(new_probs - old_probs)

            # PPO kırpma objektifi
            surr1 = ratio * advantages
            surr2 = torch.clamp(ratio, 1-self.eps_clip, 1+self.eps_clip) * advantages
            actor_loss = -torch.min(surr1, surr2).mean()

            # Değer fonksiyonu kaybı
            critic_loss = F.mse_loss(state_values, rewards)

            # Toplam kayıp
            loss = actor_loss + 0.5 * critic_loss

            self.optimizer.zero_grad()
            loss.backward()
            self.optimizer.step()
```

Soft Actor-Critic (SAC)

SAC, maksimum entropi pekiştirmeli öğrenme çerçevesini kullanan modern bir off-policy algoritmasıdır. Temel hedef fonksiyonu:

$J(\theta) = E[\sum r_t + \alpha H(\pi(\cdot|s_t))]$

Burada:

- α: Entropi katsayısı
- H: Politika entropisi

- π: Parametrize edilmiş politika

```
class SACNetwork(nn.Module):
    def __init__(self, state_dim, action_dim):
        super().__init__()
        # Q-fonksiyonları
        self.q1 = QNetwork(state_dim, action_dim)
        self.q2 = QNetwork(state_dim, action_dim)

        # Politika ağı
        self.policy = GaussianPolicy(state_dim, action_dim)

    def forward(self, state):
        mean, log_std = self.policy(state)
        return mean, log_std

class SAC:
    def __init__(self, state_dim, action_dim, lr=3e-4, alpha=0.2):
        self.network = SACNetwork(state_dim, action_dim)
        self.alpha = alpha

        # Optimizerlar
        self.policy_optimizer = optim.Adam(self.network.policy.parameters(), lr=lr)
        self.q1_optimizer = optim.Adam(self.network.q1.parameters(), lr=lr)
        self.q2_optimizer = optim.Adam(self.network.q2.parameters(), lr=lr)

    def select_action(self, state):
        state = torch.FloatTensor(state)
        mean, log_std = self.network(state)

        # Reparametrization trick
        std = log_std.exp()
        normal = Normal(mean, std)
        x_t = normal.rsample()
        action = torch.tanh(x_t)

        return action.detach().numpy()

    def update(self, batch):
        # SAC güncelleme adımları
        state, action, reward, next_state, done = batch

        # Q-değeri güncellemesi
        with torch.no_grad():
            next_action, next_log_pi = self.network.policy.sample(next_state)
            next_q1 = self.network.q1(next_state, next_action)
            next_q2 = self.network.q2(next_state, next_action)
            next_q = torch.min(next_q1, next_q2) - self.alpha * next_log_pi
            target_q = reward + (1 - done) * self.gamma * next_q

        # Q-fonksiyonu kayıpları
        current_q1 = self.network.q1(state, action)
```

```
        current_q2 = self.network.q2(state, action)
        q1_loss = F.mse_loss(current_q1, target_q)
        q2_loss = F.mse_loss(current_q2, target_q)

        # Politika kaybı
        new_action, log_pi = self.network.policy.sample(state)
        q1_new = self.network.q1(state, new_action)
        q2_new = self.network.q2(state, new_action)
        q_new = torch.min(q1_new, q2_new)
        policy_loss = (self.alpha * log_pi - q_new).mean()

        # Optimizasyon adımları
        self.q1_optimizer.zero_grad()
        q1_loss.backward()
        self.q1_optimizer.step()

        self.q2_optimizer.zero_grad()
        q2_loss.backward()
        self.q2_optimizer.step()

        self.policy_optimizer.zero_grad()
        policy_loss.backward()
        self.policy_optimizer.step()
```

Kullanım Alanları ve Uygulamalar

1. Robotik Kontrol:
    - Hareket planlama
    - Manipülasyon görevleri
    - Yürüme ve denge kontrolü
    - İnsan-robot etkileşimi
2. Oyun AI:
    - Strateji oyunları
    - Arcade oyunları
    - Kart oyunları
    - Spor simülasyonları
3. Otonom Sistemler:
    - Drone kontrolü
    - Sürücüsüz araçlar
    - Enerji yönetimi
    - Trafik kontrolü
4. Endüstriyel Optimizasyon:
    - Üretim planlama
    - Kaynak tahsisi
    - Stok yönetimi
    - Kalite kontrolü

Maliyet Analizi

1. Hesaplama Gereksinimleri:
    - Eğitim Süresi:
        - REINFORCE: 1-2 gün
        - PPO: 3-5 gün
        - SAC: 4-7 gün
    - GPU Kullanımı:
        - Minimum: NVIDIA RTX 2060
        - Önerilen: NVIDIA RTX 3080 Ti
    - RAM:
        - Minimum: 16GB
        - Önerilen: 32GB+
2. Veri ve Depolama:
    - Experience Buffer:
        - PPO: 10-50GB
        - SAC: 50-200GB
    - Model Checkpoints:
        - Her model için 1-5GB
        - Versiyonlama için ek 10-20GB
3. İnsan Kaynağı:
    - Uzmanlık Seviyesi:
        - Senior ML Mühendisi
        - RL teorisi deneyimi
        - Optimizasyon bilgisi
    - Geliştirme Süresi:
        - Prototip: 2-4 hafta
        - Prodüksiyon: 2-3 ay
        - Optimizasyon: 1-2 ay
4. Altyapı Maliyetleri:
    - Cloud GPU (aylık):
        - AWS p3.2xlarge: ~$3000
        - Google Cloud V100: ~$2500
    - Depolama (aylık):
        - Experience Buffer: $100-500
        - Model Depolama: $50-200

4. BİLGİSAYARLI GÖRÜ ALGORİTMALARI

4.1 Konvolüsyonel Sinir Ağları (CNN) Mimarileri

ResNet (Residual Networks)

ResNet, atlama bağlantıları kullanarak derin ağların eğitimini kolaylaştıran bir mimaridir. Temel yapı taşı:

H(x) = F(x) + x

```
class ResidualBlock(nn.Module):
    def __init__(self, in_channels, out_channels, stride=1):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, 3, stride, 1)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, 3, 1, 1)
        self.bn2 = nn.BatchNorm2d(out_channels)

        self.shortcut = nn.Sequential()
        if stride != 1 or in_channels != out_channels:
            self.shortcut = nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 1, stride),
                nn.BatchNorm2d(out_channels)
            )

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))
        out += self.shortcut(x)
        out = F.relu(out)
        return out

class ResNet(nn.Module):
    def __init__(self, num_classes=1000):
        super().__init__()
        self.in_channels = 64
        self.conv1 = nn.Conv2d(3, 64, 7, stride=2, padding=3)
        self.bn1 = nn.BatchNorm2d(64)
        self.maxpool = nn.MaxPool2d(3, stride=2, padding=1)

        # Residual katmanları
        self.layer1 = self.make_layer(64, 2, stride=1)
        self.layer2 = self.make_layer(128, 2, stride=2)
        self.layer3 = self.make_layer(256, 2, stride=2)
        self.layer4 = self.make_layer(512, 2, stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))
        self.fc = nn.Linear(512, num_classes)

    def make_layer(self, out_channels, num_blocks, stride):
```

```
        layers = []
        layers.append(ResidualBlock(self.in_channels, out, _channels, stride)
        self.in_channels = out_channels
        for _ in range(1, num_blocks):
            layers.append(ResidualBlock(out_channels, out_channels))
        return nn.Sequential(*layers)

    def forward(self, x):
        x = F.relu(self.bn1(self.conv1(x)))
        x = self.maxpool(x)

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.fc(x)
        return x
```

EfficientNet

EfficientNet, ağ derinliği, genişliği ve çözünürlüğü arasında optimum dengeyi bulan bileşik ölçeklendirme yaklaşımını kullanan modern bir mimaridir. Temel ölçeklendirme denklemi:

depth = α^φ
width = β^φ
resolution = γ^φ

Burada:
- α, β, γ: Ölçeklendirme katsayıları
- φ: Bileşik katsayı

```python
class MBConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, expand_ratio, stride, kernel_size):
        super().__init__()
        self.expand_ratio = expand_ratio
        hidden_dim = in_channels * expand_ratio

        # Genişletme katmanı
        self.expand_conv = nn.Sequential(
            nn.Conv2d(in_channels, hidden_dim, 1, bias=False),
            nn.BatchNorm2d(hidden_dim),
            nn.SiLU()
        ) if expand_ratio != 1 else nn.Identity()

        # Derinlemesine konvolüsyon
        self.dwconv = nn.Sequential(
            nn.Conv2d(hidden_dim, hidden_dim, kernel_size, stride,
```

```
                    padding=kernel_size//2, groups=hidden_dim, bias=False),
            nn.BatchNorm2d(hidden_dim),
            nn.SiLU()
        )

        # SE katmanı
        self.se = SqueezeExcitation(hidden_dim)

        # Projeksiyon katmanı
        self.project_conv = nn.Sequential(
            nn.Conv2d(hidden_dim, out_channels, 1, bias=False),
            nn.BatchNorm2d(out_channels)
        )

        # Atlama bağlantısı
        self.skip = stride == 1 and in_channels == out_channels

    def forward(self, x):
        identity = x

        x = self.expand_conv(x)
        x = self.dwconv(x)
        x = self.se(x)
        x = self.project_conv(x)

        if self.skip:
            x += identity

        return x
```

4.2 Nesne Algılama Mimarileri

YOLO (You Only Look Once)

YOLO, tek geçişte nesne tespiti yapan hızlı ve etkili bir mimaridir. Temel prensip, görüntüyü grid hücrelerine bölerek her hücre için nesne varlığı, sınıfı ve konumu tahmin etmektir.

```
class YOLOLayer(nn.Module):
    def __init__(self, anchors, num_classes, img_size):
        super().__init__()
        self.anchors = anchors
        self.num_classes = num_classes
        self.img_size = img_size
        self.mse_loss = nn.MSELoss()
        self.bce_loss = nn.BCELoss()
        self.obj_scale = 1
        self.noobj_scale = 100
        self.metrics = {}

    def forward(self, x, targets=None):
        num_batches = x.size(0)
```

```
        grid_size = x.size(2)

        # Tahmin tensörünü yeniden şekillendirme
        prediction = x.view(num_batches, len(self.anchors),
                            self.num_classes + 5, grid_size, grid_size)
        prediction = prediction.permute(0, 1, 3, 4, 2).contiguous()

        # Sigmoid aktivasyonu
        pred_boxes = prediction[..., :4].sigmoid()
        pred_conf = prediction[..., 4].sigmoid()
        pred_cls = prediction[..., 5:].sigmoid()

        # Grid oluşturma
        grid_x = torch.arange(grid_size).repeat(grid_size, 1)
        grid_y = torch.arange(grid_size).repeat(grid_size, 1).t()
        scaled_anchors = torch.FloatTensor([(a_w / self.stride,
                                             a_h / self.stride)
                                             for a_w, a_h in self.anchors])

        # Kutu koordinatlarını hesaplama
        pred_boxes[..., 0] += grid_x.view((1, 1, grid_size, grid_size))
        pred_boxes[..., 1] += grid_y.view((1, 1, grid_size, grid_size))
        pred_boxes[..., 2:] *= scaled_anchors.view((1, -1, 1, 1, 2))

        output = torch.cat((pred_boxes.view(num_batches, -1, 4),
                            pred_conf.view(num_batches, -1, 1),
                            pred_cls.view(num_batches, -1, self.num_classes)),
                            -1)

        if targets is None:
            return output
        else:
            return self.compute_loss(prediction, targets)
```

Mask R-CNN

Mask R-CNN, nesne tespiti ve instance segmentasyonu birleştiren ileri seviye bir mimaridir. Faster R-CNN'in üzerine inşa edilmiş olup, her nesne için piksel seviyesinde maske tahmini yapar.

```
class MaskRCNN(nn.Module):
    def __init__(self, num_classes, backbone='resnet50'):
        super().__init__()
        self.backbone = self._get_backbone(backbone)
        self.rpn = RegionProposalNetwork()
        self.roi_align = RoIAlign(output_size=(14, 14),
                                  spatial_scale=1.0/16.0,
                                  sampling_ratio=2)
        self.box_head = BoxHead(num_classes)
        self.mask_head = MaskHead(num_classes)

    def forward(self, images, targets=None):
        features = self.backbone(images)
```

```
        # RPN önerileri
        proposals, rpn_losses = self.rpn(features, targets)

        # RoI Align
        roi_features = self.roi_align(features, proposals)

        # Kutu tahminleri
        box_outputs = self.box_head(roi_features)

        # Maske tahminleri
        if self.training:
            mask_features = self.roi_align(features,
                                           self._get_positive_proposals(proposals))
            mask_outputs = self.mask_head(mask_features)
            return self.compute_losses(box_outputs, mask_outputs, targets)
        else:
            detections = self.postprocess_detections(box_outputs)
            mask_features = self.roi_align(features, detections)
            mask_outputs = self.mask_head(mask_features)
            return self.postprocess_masks(detections, mask_outputs)
```

4.3 Semantik Segmentasyon

DeepLab

DeepLab, atrous (dilated) konvolüsyon kullanarak etkili semantik segmentasyon yapan bir mimaridir. Atrous Spatial Pyramid Pooling (ASPP) modülü farklı ölçeklerdeki bağlamsal bilgiyi yakalar.

```
class ASPPModule(nn.Module):
    def __init__(self, in_channels, out_channels, rates):
        super().__init__()
        modules = []

        # 1x1 konvolüsyon
        modules.append(nn.Sequential(
            nn.Conv2d(in_channels, out_channels, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()
        ))

        # Atrous konvolüsyonlar
        for rate in rates:
            modules.append(nn.Sequential(
                nn.Conv2d(in_channels, out_channels, 3,
                          padding=rate, dilation=rate, bias=False),
                nn.BatchNorm2d(out_channels),
                nn.ReLU()
            ))

        # Global pooling branch
```

```
        modules.append(nn.Sequential(
            nn.AdaptiveAvgPool2d(1),
            nn.Conv2d(in_channels, out_channels, 1, bias=False),
            nn.BatchNorm2d(out_channels),
            nn.ReLU()
        ))

        self.convs = nn.ModuleList(modules)

    def forward(self, x):
        res = []
        for conv in self.convs[:-1]:
            res.append(conv(x))

        # Global pooling branch
        res.append(F.interpolate(self.convs[-1](x),
                                 size=x.shape[2:],
                                 mode='bilinear',
                                 align_corners=False))

        return torch.cat(res, dim=1)
```

Kullanım Alanları ve Uygulamalar

1. Tıbbi Görüntü Analizi:
    - Tümör Tespiti ve Segmentasyonu
    - Organ Segmentasyonu
    - Patoloji Analizi
    - Radyoloji Raporlaması
2. Otonom Sistemler:
    - Sürücüsüz Araçlar
    - Drone Navigasyonu
    - Robot Görüşü
    - Endüstriyel İnspeksiyon
3. Güvenlik ve Gözetim:
    - Yüz Tanıma
    - Kalabalık Analizi
    - Anormal Davranış Tespiti
    - Nesne Takibi
4. Perakende:
    - Raf Analizi
    - Ürün Tanıma
    - Müşteri Davranış Analizi
    - Envanter Yönetimi

Maliyet Analizi

1. Donanım Gereksinimleri:

- GPU:
  - Eğitim: NVIDIA A100 veya eşdeğeri
  - Çıkarım: NVIDIA T4 veya RTX serisi
- RAM: 32GB - 128GB
- Depolama: 500GB - 2TB SSD

2. Hesaplama Maliyetleri:
   - Eğitim Süresi:
     - ResNet: 2-3 gün
     - EfficientNet: 4-5 gün
     - YOLO: 3-4 gün
     - Mask R-CNN: 5-7 gün
   - GPU Saati Maliyeti:
     - AWS p3.2xlarge: $3.06/saat
     - Google Cloud V100: $2.48/saat

3. Veri Gereksinimleri:
   - Eğitim Verisi: 50GB - 1TB
   - Etiketleme Maliyeti: $0.5-5/görüntü
   - Depolama Maliyeti: $100-500/ay

4. İnsan Kaynağı:
   - Uzmanlık: Computer Vision, DL
   - Geliştirme: 2-6 ay
   - Bakım: Sürekli
   - Ekip: 2-5 kişi

Optimizasyon Teknikleri

1. Model Optimizasyonu:
   - Quantization
   - Pruning
   - Knowledge Distillation
   - Model Compression

2. Performans İyileştirme:
   - TensorRT
   - ONNX Runtime
   - OpenVINO
   - Batch Processing

```
def optimize_model(model, calibration_data):
    # Quantization
    quantized_model = torch.quantization.quantize_dynamic(
        model, {nn.Linear, nn.Conv2d}, dtype=torch.qint8
    )

    # Pruning
```

```
    parameters_to_prune = (
        (model.conv1, 'weight'),
        (model.conv2, 'weight'),
    )
    prune.global_unstructured(
        parameters_to_prune,
        pruning_method=prune.L1Unstructured,
        amount=0.2,
    )

    # Knowledge Distillation
    teacher_model = model
    student_model = SmallerArchitecture()

    def distillation_loss(student_output, teacher_output, true_labels):
        alpha = 0.5
        T = 2.0
        distill_loss = nn.KLDivLoss()(F.log_softmax(student_output/T, dim=1),
                                      F.softmax(teacher_output/T, dim=1))
        student_loss = F.cross_entropy(student_output, true_labels)
        return alpha * student_loss + (1-alpha) * distill_loss

    return quantized_model, student_model
```

5. OPTİMİZASYON ALGORİTMALARI

5.1 Gradyan Tabanlı Optimizasyon

Adam (Adaptive Moment Estimation)

Adam, momentum ve RMSprop'u birleştiren adaptif öğrenme oranlı bir optimizasyon algoritmasıdır. Matematiksel formülasyonu:

$m_t = \beta_1 m_{t-1} + (1-\beta_1)g_t \quad v_t = \beta_2 v_{t-1} + (1-\beta_2)g_t^2 \quad \hat{m}_t = m_t/(1-\beta_1^t) \quad \hat{v}_t = v_t/(1-\beta_2^t) \quad \theta_t = \theta_{t-1} - \alpha \hat{m}_t/\sqrt{\hat{v}_t + \varepsilon}$

```
class Adam:
    def __init__(self, params, lr=1e-3, betas=(0.9, 0.999), eps=1e-8):
        self.params = list(params)
        self.lr = lr
        self.beta1, self.beta2 = betas
        self.eps = eps

        # Momentum ve ikinci moment tahminleri için state başlatma
        self.state = {
            'step': 0,
            'm': [torch.zeros_like(p) for p in self.params],
            'v': [torch.zeros_like(p) for p in self.params]
        }

    def step(self):
        self.state['step'] += 1
```

```
        step = self.state['step']

        for i, p in enumerate(self.params):
            if p.grad is None:
                continue

            # Gradyan
            grad = p.grad.data

            # Momentum güncelleme
            self.state['m'][i] = self.beta1 * self.state['m'][i] + \
                                 (1 - self.beta1) * grad

            # İkinci moment güncelleme
            self.state['v'][i] = self.beta2 * self.state['v'][i] + \
                                 (1 - self.beta2) * grad * grad

            # Bias düzeltme
            m_hat = self.state['m'][i] / (1 - self.beta1 ** step)
            v_hat = self.state['v'][i] / (1 - self.beta2 ** step)

            # Parametre güncelleme
            p.data -= self.lr * m_hat / (torch.sqrt(v_hat) + self.eps)
```

AdaGrad (Adaptive Gradient Algorithm)

AdaGrad, her parametre için ayrı öğrenme oranı kullanarak, seyrek özelliklere sahip parametreleri daha agresif günceller. Temel formülü:

$\theta_{t+1} = \theta_t - \alpha/\sqrt{(G_t + \varepsilon)} \odot g_t$

Burada:

- G_t: Geçmiş gradyanların karelerinin toplamı
- ε: Sayısal kararlılık için küçük bir sayı
- $\odot$: Eleman bazında çarpım

```
class Adagrad:
    def __init__(self, params, lr=1e-2, eps=1e-10):
        self.params = list(params)
        self.lr = lr
        self.eps = eps

        # Gradyan karelerinin toplamı için state başlatma
        self.state = {
            'sum_squares': [torch.zeros_like(p) for p in self.params]
        }

    def step(self):
        for i, p in enumerate(self.params):
            if p.grad is None:
                continue
```

```
            grad = p.grad.data

            # Gradyan karelerinin toplamını güncelle
            self.state['sum_squares'][i] += grad * grad

            # Parametre güncelleme
            std = torch.sqrt(self.state['sum_squares'][i] + self.eps)
            p.data -= self.lr * grad / std
```

5.2 İkinci Derece Optimizasyon

L-BFGS (Limited-memory BFGS)

L-BFGS, Hessian matrisinin yaklaşık tersini kullanarak ikinci derece optimizasyon yapan bir algoritmadır. Bellek verimli bir şekilde çalışır.

```
class LBFGS:
    def __init__(self, params, max_iter=20, history_size=10, tolerance_grad=1e-7):
        self.params = list(params)
        self.max_iter = max_iter
        self.history_size = history_size
        self.tolerance_grad = tolerance_grad

        # Geçmiş bilgileri için state başlatma
        self.state = {
            'old_dirs': [],    # Eski arama yönleri
            'old_stps': [],    # Eski adım boyutları
            'H_diag': 1.0      # Hessian'ın diagonal yaklaşımı
        }

    def two_loop_recursion(self, grad):
        """Two-loop recursion for computing the search direction."""
        ro = []
        al = []

        # Geçmiş vektörler üzerinden ilk döngü
        q = grad.clone()
        for i in range(len(self.state['old_dirs'])-1, -1, -1):
            ro.append(1.0 / torch.dot(self.state['old_dirs'][i],
                                      self.state['old_stps'][i]))
            al.append(ro[-1] * torch.dot(self.state['old_dirs'][i], q))
            q -= al[-1] * self.state['old_stps'][i]

        # İkinci döngü
        r = self.state['H_diag'] * q
        for i in range(len(self.state['old_dirs'])):
            be = ro[i] * torch.dot(self.state['old_stps'][i], r)
            r += (al[i] - be) * self.state['old_dirs'][i]

        return r

    def step(self, closure):
```

```
        # İlk fonksiyon ve gradyan değerlendirmesi
        loss, grad = closure()

        # Arama yönünü hesapla
        d = -self.two_loop_recursion(grad)

        # Line search ile adım boyutunu belirle
        step_size = self.line_search(closure, d)

        # Parametreleri güncelle
        for p, d_i in zip(self.params, d):
            p.data.add_(step_size, d_i)

        return loss
```

5.3 Evrimsel Optimizasyon

Genetik Algoritma

Genetik algoritma, doğal seleksiyonu taklit eden popülasyon tabanlı bir optimizasyon yöntemidir.

```
class GeneticAlgorithm:
    def __init__(self, population_size, genome_size, fitness_func):
        self.population_size = population_size
        self.genome_size = genome_size
        self.fitness_func = fitness_func

        # Rastgele başlangıç popülasyonu oluştur
        self.population = torch.rand(population_size, genome_size)
        self.fitness_scores = torch.zeros(population_size)

    def select_parents(self):
        """Tournament selection ile ebeveyn seçimi."""
        tournament_size = 3
        parents = []

        for _ in range(self.population_size):
            # Rastgele bireyler seç
            tournament_idx = torch.randint(self.population_size,
                                           (tournament_size,))
            tournament_fitness = self.fitness_scores[tournament_idx]

            # En iyi bireyi seç
            winner_idx = tournament_idx[tournament_fitness.argmax()]
            parents.append(self.population[winner_idx])

        return torch.stack(parents)

    def crossover(self, parents):
        """Uniform crossover ile yeni nesil oluşturma."""
        children = []
```

```
        for i in range(0, len(parents), 2):
            if i + 1 < len(parents):
                # Crossover maskesi oluştur
                mask = torch.rand(self.genome_size) > 0.5

                # İki yeni birey oluştur
                child1 = torch.where(mask, parents[i], parents[i+1])
                child2 = torch.where(mask, parents[i+1], parents[i])

                children.extend([child1, child2])

        return torch.stack(children)

    def mutate(self, population, mutation_rate=0.01):
        """Gaussian mutation uygulama."""
        mutation_mask = torch.rand_like(population) < mutation_rate
        mutation = torch.randn_like(population) * 0.1

        return torch.where(mutation_mask, population + mutation, population)

    def optimize(self, generations=100):
        for gen in range(generations):
            # Fitness değerlendirmesi
            self.fitness_scores = self.fitness_func(self.population)

            # Ebeveyn seçimi
            parents = self.select_parents()

            # Yeni nesil oluşturma
            children = self.crossover(parents)

            # Mutasyon
            children = self.mutate(children)

            # Popülasyonu güncelle
            self.population = children

        # En iyi çözümü döndür
        best_idx = self.fitness_scores.argmax()
        return self.population[best_idx], self.fitness_scores[best_idx]
```

Kullanım Alanları ve Uygulamalar

1. Derin Öğrenme:
    - Model Eğitimi
    - Hiperparametre Optimizasyonu
    - Mimari Arama
    - Transfer Öğrenme
2. Robotik:

- Hareket Planlama
- Kontrol Sistemleri
- Yörünge Optimizasyonu
- Parametre Ayarlama

3. Finans:
   - Portföy Optimizasyonu
   - Risk Yönetimi
   - Alım-Satım Stratejileri
   - Fiyat Tahminleme
4. Üretim:
   - Kaynak Tahsisi
   - Çizelgeleme
   - Kalite Kontrolü
   - Süreç Optimizasyonu

Maliyet Analizi

1. Hesaplama Gereksinimleri:
   - Adam/AdaGrad:
     - CPU: Orta düzey işlemci
     - RAM: 8-16GB
     - Zaman: O(n) karmaşıklık
   - L-BFGS:
     - CPU: Yüksek performanslı işlemci
     - RAM: 16-32GB
     - Zaman: $O(n^2)$ karmaşıklık
   - Genetik Algoritma:
     - CPU: Çok çekirdekli işlemci
     - RAM: 32GB+
     - GPU: Paralel hesaplama için faydalı
2. Bellek Kullanımı:
   - Adam: O(n) ek bellek
   - AdaGrad: O(n) ek bellek
   - L-BFGS: O(mn) ek bellek (m: geçmiş boyutu)
   - GA: O(pn) ek bellek (p: popülasyon boyutu)
3. İnsan Kaynağı:
   - Uzmanlık: Optimizasyon teorisi
   - Geliştirme: 1-3 ay
   - Bakım: Düzenli izleme
   - Ekip: 1-3 kişi
4. Altyapı Maliyeti:
   - Donanım: $5,000-20,000
   - Yazılım lisansları: $1,000-5,000/yıl

- Cloud hizmetleri: $500-2,000/ay
- Bakım: $1,000-3,000/ay

Performans Metrikleri ve İzleme

1. Optimizasyon Metrikleri:

```python
def evaluate_optimizer(optimizer, model, train_loader, epochs):
    metrics = {
        'loss_history': [],
        'gradient_norm': [],
        'parameter_updates': [],
        'convergence_rate': [],
        'memory_usage': [],
        'time_per_epoch': []
    }

    for epoch in range(epochs):
        start_time = time.time()

        for batch in train_loader:
            # Gradyan hesaplama
            loss = model(batch)
            loss.backward()

            # Metrikleri kaydet
            metrics['gradient_norm'].append(
                compute_gradient_norm(model.parameters()))

            # Optimizasyon adımı
            optimizer.step()
            optimizer.zero_grad()

            metrics['loss_history'].append(loss.item())

        metrics['time_per_epoch'].append(time.time() - start_time)
        metrics['memory_usage'].append(torch.cuda.memory_allocated())

    return metrics
```

2. Performans Görselleştirme:

```python
def plot_optimization_metrics(metrics):
    fig, axes = plt.subplots(2, 2, figsize=(15, 10))

    # Kayıp eğrisi
    axes[0,0].plot(metrics['loss_history'])
    axes[0,0].set_title('Loss vs. Iterations')

    # Gradyan normu
    axes[0,1].plot(metrics['gradient_norm'])
    axes[0,1].set_title('Gradient Norm vs. Iterations')
```

```
    # Bellek kullanımı
    axes[1,0].plot(metrics['memory_usage'])
    axes[1,0].set_title('Memory Usage vs. Epochs')

    # Zaman analizi
    axes[1,1].plot(metrics['time_per_epoch'])
    axes[1,1].set_title('Time per Epoch')

    plt.tight_layout()
    return fig
```

Optimizasyon İpuçları ve En İyi Uygulamalar

1. Öğrenme Oranı Planlaması:

```
class LRScheduler:
    def __init__(self, optimizer, mode='cosine'):
        self.optimizer = optimizer
        self.mode = mode

    def cosine_schedule(self, epoch, total_epochs):
        return 0.5 * (1 + math.cos(math.pi * epoch / total_epochs))

    def step_schedule(self, epoch, step_size=30):
        return 0.1 ** (epoch // step_size)

    def update_lr(self, epoch, total_epochs):
        if self.mode == 'cosine':
            factor = self.cosine_schedule(epoch, total_epochs)
        else:
            factor = self.step_schedule(epoch)

        for param_group in self.optimizer.param_groups:
            param_group['lr'] = param_group['initial_lr'] * factor
```

2. Gradient Clipping:

```
def clip_gradients(model, max_norm=1.0):
    torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm)
```

3. Warmup Stratejisi:

```
class WarmupScheduler:
 def __init__(self, optimizer, warmup_epochs=5, total_epochs=100):
     self.optimizer = optimizer
     self.warmup_epochs = warmup_epochs
     self.total_epochs = total_epochs
     self.base_lr = [group['lr'] for group in optimizer.param_groups]

 def get_lr(self, epoch):
     if epoch < self.warmup_epochs:
```

```
            # Doğrusal warmup
            return [lr * (epoch + 1) / self.warmup_epochs
                    for lr in self.base_lr]
        else:
            # Warmup sonrası kosinus azalma
            progress = (epoch - self.warmup_epochs) / \
                       (self.total_epochs - self.warmup_epochs)
            return [lr * 0.5 * (1 + math.cos(math.pi * progress))
                    for lr in self.base_lr]

    def step(self, epoch):
        lrs = self.get_lr(epoch)
        for param_group, lr in zip(self.optimizer.param_groups, lrs):
            param_group['lr'] = lr
```

5.4 Gelişmiş Optimizasyon Teknikleri

Bayesian Optimizasyon

Bayesian optimizasyon, hiperparametre optimizasyonu için sıkça kullanılan ve Gaussian süreçlere dayanan bir yöntemdir.

```
class BayesianOptimizer:
    def __init__(self, parameter_space, objective_function):
        self.parameter_space = parameter_space
        self.objective_function = objective_function
        self.gp = GaussianProcessRegressor()
        self.X_observed = []
        self.y_observed = []

    def acquisition_function(self, X):
        """Expected Improvement acquisition function"""
        mean, std = self.gp.predict(X, return_std=True)
        best_y = max(self.y_observed)

        # Normal dağılım hesaplamaları
        z = (mean - best_y) / std
        phi = norm.pdf(z)
        Phi = norm.cdf(z)

        ei = std * (z * Phi + phi)
        return ei

    def optimize(self, n_iterations=50):
        for i in range(n_iterations):
            # Acquisition fonksiyonunu maksimize ederek yeni nokta seç
            X_next = self.maximize_acquisition()

            # Objektif fonksiyonu değerlendir
            y_next = self.objective_function(X_next)

            # Gözlemleri güncelle
```

```
            self.X_observed.append(X_next)
            self.y_observed.append(y_next)

            # Gaussian süreci güncelle
            self.gp.fit(self.X_observed, self.y_observed)

        # En iyi parametreleri döndür
        best_idx = np.argmax(self.y_observed)
        return self.X_observed[best_idx], self.y_observed[best_idx]
```

Diferansiyel Evrim

Diferansiyel evrim, popülasyon tabanlı optimizasyon için etkili bir evrimsel algoritmadır.

```
class DifferentialEvolution:
    def __init__(self, objective_func, bounds, population_size=50, F=0.8, CR=0.7):
        self.objective_func = objective_func
        self.bounds = bounds
        self.population_size = population_size
        self.F = F  # Ölçekleme faktörü
        self.CR = CR  # Crossover oranı

        # Başlangıç popülasyonunu oluştur
        self.population = self.initialize_population()
        self.fitness = np.array([self.objective_func(ind)
                                 for ind in self.population])

    def initialize_population(self):
        """Sınırlar içinde rastgele popülasyon oluştur"""
        population = np.random.rand(self.population_size, len(self.bounds))
        for i, (lower, upper) in enumerate(self.bounds):
            population[:, i] = population[:, i] * (upper - lower) + lower
        return population

    def mutate(self, target_idx):
        """Diferansiyel mutasyon uygula"""
        # Rastgele üç farklı birey seç
        idxs = [idx for idx in range(self.population_size)
                if idx != target_idx]
        a, b, c = self.population[np.random.choice(idxs, 3, replace=False)]

        # Mutant vektör oluştur
        mutant = a + self.F * (b - c)

        # Sınırlar içinde tut
        for i, (lower, upper) in enumerate(self.bounds):
            mutant[i] = np.clip(mutant[i], lower, upper)

        return mutant

    def crossover(self, target, mutant):
        """Binomial crossover uygula"""
```

```
        trial = np.copy(target)

        for i in range(len(self.bounds)):
            if np.random.rand() < self.CR:
                trial[i] = mutant[i]

        return trial

    def optimize(self, max_generations=1000, tol=1e-6):
        for generation in range(max_generations):
            for i in range(self.population_size):
                # Mutasyon
                mutant = self.mutate(i)

                # Crossover
                trial = self.crossover(self.population[i], mutant)

                # Seçim
                trial_fitness = self.objective_func(trial)
                if trial_fitness < self.fitness[i]:
                    self.population[i] = trial
                    self.fitness[i] = trial_fitness

            # Yakınsama kontrolü
            if np.std(self.fitness) < tol:
                break

        # En iyi çözümü döndür
        best_idx = np.argmin(self.fitness)
        return self.population[best_idx], self.fitness[best_idx]
```

Particle Swarm Optimization (PSO)

PSO, sürü zekasına dayanan ve sürekli optimizasyon problemleri için kullanılan bir algoritmadır.

```
class ParticleSwarmOptimizer:
    def __init__(self, objective_func, bounds, n_particles=30):
        self.objective_func = objective_func
        self.bounds = bounds
        self.n_particles = n_particles

        # Parçacıkları başlat
        self.particles = self.initialize_particles()
        self.velocities = np.zeros_like(self.particles)

        # Kişisel ve global en iyileri takip et
        self.personal_best_pos = np.copy(self.particles)
        self.personal_best_scores = np.array([self.objective_func(p)
                                              for p in self.particles])

        best_idx = np.argmin(self.personal_best_scores)
        self.global_best_pos = np.copy(self.personal_best_pos[best_idx])
```

```
        self.global_best_score = self.personal_best_scores[best_idx]

    def initialize_particles(self):
        """Sınırlar içinde rastgele parçacıklar oluştur"""
        particles = np.random.rand(self.n_particles, len(self.bounds))
        for i, (lower, upper) in enumerate(self.bounds):
            particles[:, i] = particles[:, i] * (upper - lower) + lower
        return particles

    def update_velocities(self, w=0.7, c1=2.0, c2=2.0):
        """Parçacık hızlarını güncelle"""
        r1, r2 = np.random.rand(2)

        cognitive_velocity = c1 * r1 * (self.personal_best_pos - self.particles)
        social_velocity = c2 * r2 * (self.global_best_pos - self.particles)

        self.velocities = w * self.velocities + \
                          cognitive_velocity + social_velocity

    def update_positions(self):
        """Parçacık pozisyonlarını güncelle"""
        self.particles += self.velocities

        # Sınırlar içinde tut
        for i, (lower, upper) in enumerate(self.bounds):
            self.particles[:, i] = np.clip(self.particles[:, i],
                                           lower, upper)

    def optimize(self, max_iterations=100, tol=1e-6):
        for iteration in range(max_iterations):
            # Hız ve pozisyon güncellemeleri
            self.update_velocities()
            self.update_positions()

            # Fitness değerlendirmesi
            current_scores = np.array([self.objective_func(p)
                                       for p in self.particles])

            # Kişisel en iyileri güncelle
            improved_idx = current_scores < self.personal_best_scores
            self.personal_best_pos[improved_idx] = \
                self.particles[improved_idx]
            self.personal_best_scores[improved_idx] = \
                current_scores[improved_idx]

            # Global en iyiyi güncelle
            best_idx = np.argmin(current_scores)
            if current_scores[best_idx] < self.global_best_score:
                self.global_best_pos = np.copy(self.particles[best_idx])
                self.global_best_score = current_scores[best_idx]

            # Yakınsama kontrolü
```

```
            if np.std(current_scores) < tol:
                break

        return self.global_best_pos, self.global_best_score
```

Uygulama Örnekleri ve Kıyaslamalar

Hiperparametre Optimizasyonu

```
def optimize_hyperparameters(model_class, train_data, val_data):
    # Parametre uzayını tanımla
    parameter_space = {
        'learning_rate': (1e-4, 1e-2),
        'batch_size': (16, 256),
        'hidden_size': (32, 512),
        'dropout': (0.1, 0.5)
    }

    def objective_function(params):
        # Model oluştur ve eğit
        model = model_class(**params)
        trainer = Trainer(model)
        trainer.fit(train_data)

        # Validasyon performansını döndür
        val_score = trainer.evaluate(val_data)
        return -val_score  # Minimizasyon için negatif

    # Farklı optimizasyon yöntemlerini karşılaştır
    optimizers = {
        'bayesian': BayesianOptimizer(parameter_space, objective_function),
        'differential_evolution': DifferentialEvolution(objective_function,
                                                        parameter_space),
        'pso': ParticleSwarmOptimizer(objective_function, parameter_space)
    }

    results = {}
    for name, optimizer in optimizers.items():
        start_time = time.time()
        best_params, best_score = optimizer.optimize()
        end_time = time.time()

        results[name] = {
            'best_params': best_params,
            'best_score': -best_score,  # Pozitife çevir
            'time_taken': end_time - start_time
        }

    return results
```

Optimizasyon Performans Analizi

```python
def analyze_optimization_performance(results):
    """Farklı optimizasyon yöntemlerinin performans analizi"""
    metrics = {
        'convergence_speed': {},
        'solution_quality': {},
        'computational_cost': {}
    }

    for method, result in results.items():
        # Yakınsama hızı (iterasyon başına iyileşme)
        metrics['convergence_speed'][method] = \
            result['best_score'] / result['time_taken']

        # Çözüm kalitesi (final skor)
        metrics['solution_quality'][method] = result['best_score']

        # Hesaplama maliyeti (toplam süre)
        metrics['computational_cost'][method] = result['time_taken']

    return metrics

def visualize_optimization_comparison(metrics):
    """Optimizasyon sonuçlarının görsel karşılaştırması"""
    fig, axes = plt.subplots(1, 3, figsize=(15, 5))

    for i, (metric, values) in enumerate(metrics.items()):
        axes[i].bar(values.keys(), values.values())
        axes[i].set_title(metric.replace('_', ' ').title())
        axes[i].tick_params(axis='x', rotation=45)

    plt.tight_layout()
    return fig
```

Gelişmiş Özellikler ve İyileştirmeler

Adaptif Parametre Ayarı

```python
class AdaptiveOptimizer:
    def __init__(self, base_optimizer, adaptation_rate=0.1):
        self.optimizer = base_optimizer
        self.adaptation_rate = adaptation_rate
        self.parameter_history = []

    def adapt_parameters(self):
        """Optimizasyon parametrelerini performansa göre ayarla"""
        if len(self.parameter_history) < 2:
            return

        recent_improvement = (self.parameter_history[-1] -
                              self.parameter_history[-2])
```

```
        if recent_improvement > 0:
            # İyileşme varsa parametreleri artır
            self.optimizer.step_size *= (1 + self.adaptation_rate)
        else:
            # İyileşme yoksa parametreleri azalt
            self.optimizer.step_size *= (1 - self.adaptation_rate)

    def optimize(self, *args, **kwargs):
        result = self.optimizer.optimize(*args, **kwargs)
        self.parameter_history.append(result[1])
        self.adapt_parameters()
        return result
```

6. İLERİ SEVİYE OPTİMİZASYON TEKNİKLERİ

6.1 Çok Amaçlı Optimizasyon (Multi-Objective Optimization)

Çok amaçlı optimizasyon, birden fazla hedef fonksiyonunun eş zamanlı olarak optimize edilmesi gereken durumlarda kullanılır. Bu tür problemlerde genellikle Pareto-optimal çözümler aranır.

NSGA-II (Non-dominated Sorting Genetic Algorithm II)

NSGA-II, çok amaçlı optimizasyon için yaygın olarak kullanılan bir genetik algoritmadır. Pareto baskınlığı ve çeşitlilik mekanizmalarını kullanır.

```
class NSGAII:
    def __init__(self, objective_functions, bounds, population_size=100):
        self.objective_functions = objective_functions
        self.bounds = bounds
        self.population_size = population_size

        # Başlangıç popülasyonunu oluştur
        self.population = self.initialize_population()
        self.fronts = self.non_dominated_sort(self.population)

    def calculate_crowding_distance(self, front):
        """Her bir çözüm için kalabalıklık mesafesini hesapla"""
        if len(front) <= 2:
            return [float('inf')] * len(front)

        distances = [0] * len(front)
        objectives = len(self.objective_functions)

        for obj in range(objectives):
            # Objektif değerine göre sırala
            sorted_front = sorted(enumerate(front),
                                  key=lambda x: self.objective_functions[obj](x[1]))

            # Sınır noktalarına sonsuz mesafe ata
```

```
            distances[sorted_front[0][0]] = float('inf')
            distances[sorted_front[-1][0]] = float('inf')

            # Ara noktaların mesafelerini hesapla
            obj_range = (self.objective_functions[obj](sorted_front[-1][1]) -
                         self.objective_functions[obj](sorted_front[0][1]))

            for i in range(1, len(front)-1):
                distances[sorted_front[i][0]] += (
                    self.objective_functions[obj](sorted_front[i+1][1]) -
                    self.objective_functions[obj](sorted_front[i-1][1])
                ) / obj_range

        return distances

    def non_dominated_sort(self, population):
        """Popülasyonu Pareto frontlarına ayır"""
        fronts = [[]]
        dominated_solutions = {}
        domination_counts = {}

        # Her çözüm için baskınlık ilişkilerini hesapla
        for p in population:
            dominated_solutions[p] = []
            domination_counts[p] = 0

            for q in population:
                if self.dominates(p, q):
                    dominated_solutions[p].append(q)
                elif self.dominates(q, p):
                    domination_counts[p] += 1

            if domination_counts[p] == 0:
                fronts[0].append(p)

        # Sonraki frontları oluştur
        i = 0
        while fronts[i]:
            next_front = []
            for p in fronts[i]:
                for q in dominated_solutions[p]:
                    domination_counts[q] -= 1
                    if domination_counts[q] == 0:
                        next_front.append(q)
            i += 1
            if next_front:
                fronts.append(next_front)

        return fronts

    def dominates(self, solution1, solution2):
        """Bir çözümün diğerini domine edip etmediğini kontrol et"""
```

```
        better_in_any = False
        for obj_func in self.objective_functions:
            if obj_func(solution1) > obj_func(solution2):
                return False
            elif obj_func(solution1) < obj_func(solution2):
                better_in_any = True
        return better_in_any

    def create_next_generation(self):
        """Yeni nesil oluştur"""
        offspring = []
        while len(offspring) < self.population_size:
            parent1 = self.tournament_select()
            parent2 = self.tournament_select()

            child1, child2 = self.crossover(parent1, parent2)
            child1 = self.mutate(child1)
            child2 = self.mutate(child2)

            offspring.extend([child1, child2])

        return offspring[:self.population_size]

    def optimize(self, generations=100):
        """NSGA-II'nin ana döngüsü"""
        for generation in range(generations):
            # Yeni nesil oluştur
            offspring = self.create_next_generation()

            # Ana ve yavru popülasyonları birleştir
            combined_pop = self.population + offspring

            # Non-dominated sorting uygula
            fronts = self.non_dominated_sort(combined_pop)

            # Yeni popülasyonu seç
            new_population = []
            front_idx = 0

            while len(new_population) + len(fronts[front_idx]) <=
self.population_size:
                # Tüm frontu ekle
                new_population.extend(fronts[front_idx])
                front_idx += 1

            if len(new_population) < self.population_size:
                # Son frontu kalabalıklık mesafesine göre sırala
                last_front = fronts[front_idx]
                crowding_distances = self.calculate_crowding_distance(last_front)
                sorted_front = sorted(zip(last_front, crowding_distances),
                                      key=lambda x: x[1], reverse=True)
```

```
                # Kalan yerleri doldur
                remaining = self.population_size - len(new_population)
                new_population.extend([x[0] for x in sorted_front[:remaining]])

            self.population = new_population

        return self.get_pareto_front()
```

6.2 Kısıtlı Optimizasyon (Constrained Optimization)

Kısıtlı optimizasyon, çözüm uzayının belirli kısıtlara tabi olduğu problemleri ele alır. Bu problemlerde hem amaç fonksiyonu optimize edilir hem de kısıtlar sağlanır.

İç Nokta Yöntemi (Interior Point Method)

İç nokta yöntemi, kısıtlı optimizasyon problemlerini çözmek için kullanılan etkili bir yaklaşımdır. Kısıtları ceza terimleri olarak probleme dahil eder.

```
class InteriorPointOptimizer:
    def __init__(self, objective_func, constraints, bounds):
        self.objective_func = objective_func
        self.constraints = constraints  # g(x) <= 0 formunda kısıtlar
        self.bounds = bounds

    def barrier_function(self, x, mu):
        """Logaritmik bariyer fonksiyonu"""
        barrier = 0
        for constraint in self.constraints:
            value = constraint(x)
            if value >= 0:
                return float('inf')
            barrier -= mu * np.log(-value)
        return self.objective_func(x) + barrier

    def compute_gradient(self, x, mu, eps=1e-8):
        """Sonlu farklar ile gradyan hesaplama"""
        gradient = np.zeros_like(x)
        for i in range(len(x)):
            x_plus = x.copy()
            x_plus[i] += eps
            x_minus = x.copy()
            x_minus[i] -= eps

            gradient[i] = (self.barrier_function(x_plus, mu) -
                           self.barrier_function(x_minus, mu)) / (2 * eps)

        return gradient

    def optimize(self, x0, mu0=1.0, mu_reduction=0.1, epsilon=1e-6):
        """İç nokta optimizasyonu ana döngüsü"""
        x = x0
        mu = mu0
```

```
        while mu > epsilon:
            # Newton yöntemi ile alt problemi çöz
            while True:
                gradient = self.compute_gradient(x, mu)
                if np.linalg.norm(gradient) < epsilon:
                    break

                # Hessian matrisini hesapla (sonlu farklar ile)
                hessian = self.compute_hessian(x, mu)

                # Newton adımı
                delta = np.linalg.solve(hessian, -gradient)

                # Line search ile adım boyutunu belirle
                alpha = self.backtracking_line_search(x, delta, mu)

                # Güncelleme
                x = x + alpha * delta

            # Bariyer parametresini azalt
            mu *= mu_reduction

        return x

    def backtracking_line_search(self, x, delta, mu, alpha=0.5, beta=0.8):
        """Backtracking line search ile uygun adım boyutu belirleme"""
        t = 1.0
        current_value = self.barrier_function(x, mu)
        gradient = self.compute_gradient(x, mu)

        while True:
            new_x = x + t * delta
            new_value = self.barrier_function(new_x, mu)

            if new_value <= current_value + alpha * t * gradient.dot(delta):
                break
            t *= beta

        return t
```

6.3 Dağıtık Optimizasyon (Distributed Optimization)

Dağıtık optimizasyon, büyük ölçekli problemlerin çözümünü birden fazla işlemci veya makine arasında paylaştırarak gerçekleştirir.

ADMM (Alternating Direction Method of Multipliers)

ADMM, dağıtık optimizasyon için yaygın olarak kullanılan bir yöntemdir. Problemi alt problemlere böler ve koordineli bir şekilde çözer.

```python
class DistributedADMM:
    def __init__(self, local_objectives, global_objective, num_workers):
        self.local_objectives = local_objectives
        self.global_objective = global_objective
        self.num_workers = num_workers

    def worker_update(self, worker_id, x, z, u, rho):
        """Yerel değişken güncellemesi"""
        def augmented_lagrangian(x_local):
            return (self.local_objectives[worker_id](x_local) +
                    (rho/2) * np.sum((x_local - z + u)**2))

        # Yerel problemi çöz
        result = minimize(augmented_lagrangian, x[worker_id],
                          method='L-BFGS-B')
        return result.x

    def z_update(self, x, u, rho):
        """Global değişken güncellemesi"""
        avg_x_plus_u = np.mean(x + u, axis=0)

        def augmented_lagrangian(z):
            return (self.global_objective(z) +
                    (rho/2) * np.sum((x - z.reshape(1,-1) + u)**2))

        result = minimize(augmented_lagrangian, avg_x_plus_u,
                          method='L-BFGS-B')
        return result.x

    def optimize(self, x0, max_iter=100, rho=1.0, abs_tol=1e-4, rel_tol=1e-2):
        """ADMM ana döngüsü"""
        # Değişkenleri başlat
        x = np.array([x0] * self.num_workers)
        z = np.mean(x, axis=0)
        u = np.zeros_like(x)

        for iteration in range(max_iter):
            # Paralel yerel güncellemeler
            x_new = np.array([
                self.worker_update(i, x, z, u, rho)
                for i in range(self.num_workers)
            ])

            # Global değişken güncellemesi
            z_new = self.z_update(x_new, u, rho)

            # Dual değişken güncellemesi
            u_new = u + x_new - z_new.reshape(1,-1)

            # Yakınsama kontrolü
            primal_residual = np.linalg.norm(x_new - z_new.reshape(1,-1))
            dual_residual = np.linalg.norm(rho * (z_new - z))
```

```
            x = x_new
            z = z_new
            u = u_new

            if primal_residual < abs_tol and dual_residual < abs_tol:
                break

        return z
```

6.4 Meta-Sezgisel Optimizasyon

Meta-sezgisel optimizasyon algoritmaları, doğadan esinlenen ve karmaşık problemleri çözmek için kullanılan yöntemlerdir.

Karınca Kolonisi Optimizasyonu (Ant Colony Optimization)

```
class AntColonyOptimizer:
    def __init__(self, distance_matrix, n_ants, evaporation_rate=0.1):
        self.distances = distance_matrix
        self.n_cities = len(distance_matrix)
        self.n_ants = n_ants
        self.evaporation = evaporation_rate

        # Feromon matrisini başlat
        self.pheromone = np.ones((self.n_cities, self.n_cities))

    def ant_tour(self, alpha=1, beta=2):
        """Tek bir karıncanın turu"""
        unvisited = set(range(1, self.n_cities))
        tour = [0]  # 0. şehirden başla

        while unvisited:
            current = tour[-1]
            # Sonraki şehri seç
            p = np.zeros(self.n_cities)
            for city in unvisited:
                p[city] = (self.pheromone[current][city]**alpha *
                           (1.0/self.distances[current][city])**beta)

            p = p / p.sum()
            next_city = np.random.choice(range(self.n_cities), p=p)
            tour.append(next_city)
            unvisited.remove(next_city)

        tour.append(0)  # Başlangıç noktasına dön
        return tour

    def calculate_tour_length(self, tour):
        """Tur uzunluğunu hesapla"""
        length = 0
        for i in range(len(tour)-1):
```

```
            length += self.distances[tour[i]][tour[i+1]]
        return length

    def optimize(self, n_iterations=100):
        """Ana optimizasyon döngüsü"""
        best_tour = None
        best_length = float('inf')

        for iteration in range(n_iterations):
            # Tüm karıncalar için turları hesapla
            ant_tours = []
            ant_lengths = []

            for ant in range(self.n_ants):
                tour = self.ant_tour()
                length = self.calculate_tour_length(tour)
                ant_tours.append(tour)
                ant_lengths.append(length)

                if length < best_length:
                    best_length = length
                    best_tour = tour

            # Feromon buharlaşması
            self.pheromone *= (1 - self.evaporation)

            # Feromon güncelleme
            for tour, length in zip(ant_tours, ant_lengths):
                deposit = 1.0 / length
                for i in range(len(tour)-1):
                    self.pheromone[tour[i]][tour[i+1]] += deposit
                    self.pheromone[tour[i+1]][tour[i]] += deposit  # Simetrik

        return best_tour, best_length

### Ateş Böceği Algoritması (Firefly Algorithm)

Ateş böceği algoritması, ateş böceklerinin ışık yayma davranışlarını taklit eden bir optimizasyon algoritmasıdır.

```python
class FireflyAlgorithm:
    def __init__(self, objective_func, bounds, n_fireflies=50, alpha=0.5, beta0=1.0,
gamma=1.0):
        self.objective_func = objective_func
        self.bounds = np.array(bounds)
        self.n_fireflies = n_fireflies
        self.alpha = alpha  # Rastgele hareket parametresi
        self.beta0 = beta0  # Çekicilik parametresi
        self.gamma = gamma  # Işık absorpsiyon katsayısı

        # Ateş böceklerini başlat
```
```

```
        dim = len(bounds)
        self.fireflies = np.random.uniform(
            self.bounds[:, 0],
            self.bounds[:, 1],
            size=(n_fireflies, dim)
        )
        self.intensity = np.array([self.objective_func(f) for f in self.fireflies])

    def beta(self, r):
        """Mesafeye bağlı çekicilik"""
        return self.beta0 * np.exp(-self.gamma * r**2)

    def move_firefly(self, i, j):
        """i. ateş böceğini j. ateş böceğine doğru hareket ettir"""
        r = np.linalg.norm(self.fireflies[i] - self.fireflies[j])
        beta = self.beta(r)

        # Yeni pozisyon hesapla
        self.fireflies[i] = self.fireflies[i] + \
                            beta * (self.fireflies[j] - self.fireflies[i]) + \
                            self.alpha * (np.random.rand(len(self.bounds)) - 0.5)

        # Sınırlar içinde tut
        self.fireflies[i] = np.clip(
            self.fireflies[i],
            self.bounds[:, 0],
            self.bounds[:, 1]
        )

        # Yoğunluğu güncelle
        self.intensity[i] = self.objective_func(self.fireflies[i])

    def optimize(self, max_generations=100):
        """Ana optimizasyon döngüsü"""
        best_solution = None
        best_intensity = float('inf')

        for generation in range(max_generations):
            # Her ateş böceği çifti için
            for i in range(self.n_fireflies):
                for j in range(self.n_fireflies):
                    if self.intensity[j] < self.intensity[i]:  # Minimizasyon
                        self.move_firefly(i, j)

            # En iyi çözümü güncelle
            gen_best_idx = np.argmin(self.intensity)
            if self.intensity[gen_best_idx] < best_intensity:
                best_intensity = self.intensity[gen_best_idx]
                best_solution = self.fireflies[gen_best_idx].copy()

            # Alfa parametresini azalt (arama alanını daralt)
            self.alpha *= 0.97
```

```
        return best_solution, best_intensity

## 7. HİBRİT OPTİMİZASYON STRATEJİLERİ

Hibrit optimizasyon, farklı optimizasyon tekniklerinin güçlü yanlarını birleştirerek
daha etkili çözümler elde etmeyi amaçlar.

### Genetik Algoritma + Yerel Arama

```python
class HybridGeneticAlgorithm:
    def __init__(self, objective_func, bounds, population_size=50,
local_search_freq=5):
        self.objective_func = objective_func
        self.bounds = bounds
        self.population_size = population_size
        self.local_search_freq = local_search_freq

        # Popülasyonu başlat
        self.population = self.initialize_population()
        self.fitness = np.array([self.objective_func(ind) for ind in self.population])

    def local_search(self, individual):
        """Hill climbing yerel arama"""
        result = minimize(
            self.objective_func,
            individual,
            method='Nelder-Mead',
            bounds=self.bounds
        )
        return result.x

    def optimize(self, n_generations=100):
        best_solution = None
        best_fitness = float('inf')

        for generation in range(n_generations):
            # Genetik operatörler
            parents = self.select_parents()
            offspring = self.crossover(parents)
            offspring = self.mutate(offspring)

            # Belirli aralıklarla yerel arama uygula
            if generation % self.local_search_freq == 0:
                for i in range(len(offspring)):
                    if np.random.random() < 0.1:  # %10 olasılıkla
                        offspring[i] = self.local_search(offspring[i])

            # Yeni nesli değerlendir
            offspring_fitness = np.array([self.objective_func(ind)
                                          for ind in offspring])
```
```

```
            # En iyileri seç
            combined = np.vstack((self.population, offspring))
            combined_fitness = np.concatenate((self.fitness, offspring_fitness))

            indices = np.argsort(combined_fitness)[:self.population_size]
            self.population = combined[indices]
            self.fitness = combined_fitness[indices]

            # En iyi çözümü güncelle
            if self.fitness[0] < best_fitness:
                best_fitness = self.fitness[0]
                best_solution = self.population[0].copy()

        return best_solution, best_fitness

## 8. PARALEL OPTİMİZASYON STRATEJİLERİ

Paralel optimizasyon, hesaplama kaynaklarını etkili kullanarak optimizasyon sürecini hızlandırır.

### Paralel Parçacık Sürüsü Optimizasyonu

```python
class ParallelPSO:
    def __init__(self, objective_func, bounds, n_particles=50, n_workers=4):
        self.objective_func = objective_func
        self.bounds = bounds
        self.n_particles = n_particles
        self.n_workers = n_workers

        # Parçacıkları başlat
        self.particles = self.initialize_particles()
        self.velocities = np.zeros_like(self.particles)

        # Kişisel ve global en iyiler
        self.personal_best_pos = self.particles.copy()
        self.personal_best_score = np.array([self.objective_func(p)
                                             for p in self.particles])
        self.global_best_idx = np.argmin(self.personal_best_score)
        self.global_best_pos = self.personal_best_pos[self.global_best_idx].copy()

    def evaluate_batch(self, batch):
        """Paralel değerlendirme için alt fonksiyon"""
        return [self.objective_func(particle) for particle in batch]

    def optimize(self, n_iterations=100):
        """Paralel PSO ana döngüsü"""
        with Pool(self.n_workers) as pool:
            for iteration in range(n_iterations):
                # Parçacıkları alt gruplara böl
                batches = np.array_split(self.particles, self.n_workers)
```
```

```
                # Paralel değerlendirme
                results = pool.map(self.evaluate_batch, batches)
                scores = np.concatenate(results)

                # Kişisel ve global en iyileri güncelle
                improved = scores < self.personal_best_score
                self.personal_best_pos[improved] = self.particles[improved]
                self.personal_best_score[improved] = scores[improved]

                global_best_idx = np.argmin(self.personal_best_score)
                if self.personal_best_score[global_best_idx] < \
                   self.personal_best_score[self.global_best_idx]:
                    self.global_best_idx = global_best_idx
                    self.global_best_pos = \
                        self.personal_best_pos[global_best_idx].copy()

                # Hız ve pozisyon güncellemesi
                self.update_velocities()
                self.update_positions()

        return self.global_best_pos, self.personal_best_score[self.global_best_idx]

## 9. ADAPTIF OPTİMİZASYON STRATEJİLERİ

Adaptif optimizasyon, algoritma parametrelerini optimizasyon süreci boyunca dinamik
olarak ayarlar.

### Adaptif Diferansiyel Evrim

```python
class AdaptiveDifferentialEvolution:
    def __init__(self, objective_func, bounds, population_size=50):
        self.objective_func = objective_func
        self.bounds = np.array(bounds)
        self.population_size = population_size

        # Başlangıç popülasyonu
        self.population = self.initialize_population()
        self.fitness = np.array([self.objective_func(ind)
                                 for ind in self.population])

        # Adaptif parametreler
        self.F = np.ones(population_size) * 0.5  # Ölçekleme faktörü
        self.CR = np.ones(population_size) * 0.5  # Crossover oranı

    def adapt_parameters(self, success_rates):
        """Başarı oranlarına göre parametreleri güncelle"""
        # Lehman-Başaranlar adaptasyonu
        c = 0.1  # Öğrenme oranı

        for i in range(self.population_size):
```
```

```
            if success_rates[i] > 0.2:
                self.F[i] = self.F[i] * (1 + c * np.random.randn())
                self.CR[i] = self.CR[i] * (1 + c * np.random.randn())
            else:
                self.F[i] = self.F[i] * (1 - c * np.random.randn())
                self.CR[i] = self.CR[i] * (1 - c * np.random.randn())

            # Parametreleri sınırlar içinde tut
            self.F[i] = np.clip(self.F[i], 0.1, 1.0)
            self.CR[i] = np.clip(self.CR[i], 0.0, 1.0)

    def optimize(self, max_generations=100):
        success_history = np.zeros(self.population_size)

        for generation in range(max_generations):
            success_count = np.zeros(self.population_size)
            trials = 0

            for i in range(self.population_size):
                # Mutasyon
                idxs = [idx for idx in range(self.population_size) if idx != i]
                a, b, c = self.population[np.random.choice(idxs, 3, replace=False)]

                mutant = a + self.F[i] * (b - c)

                # Crossover
                mask = np.random.rand(len(self.bounds)) < self.CR[i]
                if not np.any(mask):
                    mask[np.random.randint(0, len(self.bounds))] = True

                trial = np.where(mask, mutant, self.population[i])

                # Sınırlar içinde tut
                trial = np.clip(trial, self.bounds[:, 0], self.bounds[:, 1])

                # Seçim
                trial_fitness = self.objective_func(trial)
                if trial_fitness < self.fitness[i]:
                    self.population[i] = trial
                    self.fitness[i] = trial_fitness
                    success_count[i] += 1

                trials += 1

            # Başarı oranlarını güncelle
            success_rates = success_count / trials
            success_history = 0.7 * success_history + 0.3 * success_rates

            # Parametreleri adapte et
            self.adapt_parameters(success_history)
```

```
        best_idx = np.argmin(self.fitness)
        return self.population[best_idx], self.fitness[best_idx]
```

10. ÇOK ÖLÇÜTLÜ KARAR VERME YÖNTEMLERİ

10.1 Analitik Hiyerarşi Süreci (AHP)

Analitik Hiyerarşi Süreci, karmaşık karar problemlerini hiyerarşik bir yapıda ele alan ve alternatifleri çoklu kriterlere göre değerlendiren bir yöntemdir.

```
class AHPAnalyzer:
    def __init__(self, criteria, alternatives):
        self.criteria = criteria
        self.alternatives = alternatives
        self.n_criteria = len(criteria)
        self.n_alternatives = len(alternatives)

        # Saaty'nin tutarlılık indeksi tablosu
        self.random_index = {
            1: 0.00, 2: 0.00, 3: 0.58, 4: 0.90, 5: 1.12,
            6: 1.24, 7: 1.32, 8: 1.41, 9: 1.45, 10: 1.49
        }

    def calculate_priority_vector(self, comparison_matrix):
        """Karşılaştırma matrisinden öncelik vektörünü hesapla"""
        # Geometrik ortalama yöntemi
        n = len(comparison_matrix)
        geometric_mean = np.prod(comparison_matrix, axis=1) ** (1/n)
        return geometric_mean / np.sum(geometric_mean)

    def consistency_ratio(self, comparison_matrix, priority_vector):
        """Tutarlılık oranını hesapla"""
        n = len(comparison_matrix)

        # Maksimum özdeğeri hesapla
        weighted_sum = np.dot(comparison_matrix, priority_vector)
        lambda_max = np.mean(weighted_sum / priority_vector)

        # Tutarlılık indeksini hesapla
        ci = (lambda_max - n) / (n - 1)

        # Tutarlılık oranını hesapla
        cr = ci / self.random_index[n]
        return cr

    def evaluate(self, criteria_matrix, alternative_matrices):
        """AHP değerlendirmesi gerçekleştir"""
        # Kriter ağırlıklarını hesapla
        criteria_weights = self.calculate_priority_vector(criteria_matrix)

        if self.consistency_ratio(criteria_matrix, criteria_weights) > 0.1:
```

```
            raise ValueError("Kriter karşılaştırmaları tutarsız!")

        # Her kriter için alternatif önceliklerini hesapla
        alternative_priorities = np.zeros((self.n_criteria, self.n_alternatives))

        for i in range(self.n_criteria):
            alt_weights = self.calculate_priority_vector(alternative_matrices[i])
            if self.consistency_ratio(alternative_matrices[i], alt_weights) > 0.1:
                raise ValueError(f"{self.criteria[i]} için karşılaştırmalar
tutarsız!")
            alternative_priorities[i] = alt_weights

        # Global öncelikleri hesapla
        global_priorities = np.dot(criteria_weights, alternative_priorities)

        return {
            'criteria_weights': criteria_weights,
            'alternative_priorities': alternative_priorities,
            'global_priorities': global_priorities
        }
```

10.2 TOPSIS (Technique for Order Preference by Similarity to Ideal Solution)

TOPSIS, alternatifleri ideal çözüme olan yakınlıklarına göre sıralayan bir çok kriterli karar verme yöntemidir.

```
class TOPSISAnalyzer:
    def __init__(self, decision_matrix, weights, criteria_types):
        """
        decision_matrix: Alternatif x Kriter matrisi
        weights: Kriter ağırlıkları
        criteria_types: Fayda (1) veya maliyet (-1) kriterleri
        """
        self.decision_matrix = np.array(decision_matrix)
        self.weights = np.array(weights)
        self.criteria_types = np.array(criteria_types)

    def normalize_matrix(self):
        """Karar matrisini normalize et"""
        # Vektör normalizasyonu
        norms = np.sqrt(np.sum(self.decision_matrix**2, axis=0))
        return self.decision_matrix / norms

    def get_weighted_matrix(self, normalized_matrix):
        """Ağırlıklı normalize matrisi hesapla"""
        return normalized_matrix * self.weights

    def get_ideal_solutions(self, weighted_matrix):
        """İdeal ve negatif ideal çözümleri belirle"""
        ideal = np.zeros(weighted_matrix.shape[1])
        negative_ideal = np.zeros(weighted_matrix.shape[1])
```

```
        for j in range(weighted_matrix.shape[1]):
            if self.criteria_types[j] == 1:  # Fayda kriteri
                ideal[j] = np.max(weighted_matrix[:, j])
                negative_ideal[j] = np.min(weighted_matrix[:, j])
            else:  # Maliyet kriteri
                ideal[j] = np.min(weighted_matrix[:, j])
                negative_ideal[j] = np.max(weighted_matrix[:, j])

        return ideal, negative_ideal

    def calculate_distances(self, weighted_matrix, ideal, negative_ideal):
        """İdeal çözümlere olan uzaklıkları hesapla"""
        # Öklid uzaklığı
        s_plus = np.sqrt(np.sum((weighted_matrix - ideal)**2, axis=1))
        s_minus = np.sqrt(np.sum((weighted_matrix - negative_ideal)**2, axis=1))
        return s_plus, s_minus

    def get_relative_closeness(self, s_plus, s_minus):
        """Göreceli yakınlık katsayılarını hesapla"""
        return s_minus / (s_plus + s_minus)

    def evaluate(self):
        """TOPSIS değerlendirmesi gerçekleştir"""
        # Normalizasyon
        normalized_matrix = self.normalize_matrix()

        # Ağırlıklı normalizasyon
        weighted_matrix = self.get_weighted_matrix(normalized_matrix)

        # İdeal çözümler
        ideal, negative_ideal = self.get_ideal_solutions(weighted_matrix)

        # Uzaklıklar
        s_plus, s_minus = self.calculate_distances(weighted_matrix,
                                                   ideal, negative_ideal)

        # Göreceli yakınlıklar
        closeness = self.get_relative_closeness(s_plus, s_minus)

        # Sıralama
        rankings = np.argsort(closeness)[::-1]

        return {
            'closeness_coefficients': closeness,
            'rankings': rankings,
            'normalized_matrix': normalized_matrix,
            'weighted_matrix': weighted_matrix,
            'ideal_solutions': (ideal, negative_ideal),
            'distances': (s_plus, s_minus)
        }
```

11. ROBUST OPTİMİZASYON TEKNİKLERİ

Robust optimizasyon, belirsizlik altında kararlı çözümler üreten optimizasyon yaklaşımıdır.

11.1 Minimax Optimizasyon

```
class MinimaxOptimizer:
    def __init__(self, objective_func, uncertainty_set, bounds):
        self.objective_func = objective_func
        self.uncertainty_set = uncertainty_set
        self.bounds = bounds

    def worst_case_objective(self, x):
        """En kötü durum performansını hesapla"""
        def negative_obj(u):
            return -self.objective_func(x, u)

        # Belirsizlik kümesi üzerinde maksimizasyon
        result = minimize(negative_obj,
                          x0=np.zeros(len(self.uncertainty_set)),
                          bounds=self.uncertainty_set,
                          method='L-BFGS-B')

        return -result.fun

    def optimize(self, n_starts=10):
        """Minimax optimizasyonu gerçekleştir"""
        best_solution = None
        best_worst_case = float('inf')

        for _ in range(n_starts):
            # Rastgele başlangıç noktası
            x0 = np.random.uniform(self.bounds[:, 0],
                                   self.bounds[:, 1])

            # Dış minimizasyon
            result = minimize(self.worst_case_objective,
                              x0=x0,
                              bounds=self.bounds,
                              method='L-BFGS-B')

            if result.fun < best_worst_case:
                best_worst_case = result.fun
                best_solution = result.x

        return best_solution, best_worst_case
```

11.2 Belirsizlik Kümesi Optimizasyonu

```python
class UncertaintySetOptimizer:
    def __init__(self, nominal_problem, uncertainty_radius):
        self.nominal_problem = nominal_problem
        self.uncertainty_radius = uncertainty_radius

    def robust_constraint(self, x, constraint_func):
        """Robust kısıt fonksiyonu"""
        nominal_value = constraint_func(x)
        gradient = approx_fprime(x, constraint_func)

        # Belirsizlik kümesi için worst-case sapma
        worst_case_deviation = self.uncertainty_radius * np.linalg.norm(gradient)

        return nominal_value + worst_case_deviation

    def optimize(self):
        """Belirsizlik kümesi optimizasyonu gerçekleştir"""
        def robust_objective(x):
            nominal_obj = self.nominal_problem.objective(x)
            obj_gradient = approx_fprime(x, self.nominal_problem.objective)

            # Worst-case objektif değeri
            worst_case_obj = nominal_obj + \
                             self.uncertainty_radius * np.linalg.norm(obj_gradient)

            return worst_case_obj

        # Robust kısıtları oluştur
        robust_constraints = []
        for constraint in self.nominal_problem.constraints:
            def robust_cons(x, constraint=constraint):
                return self.robust_constraint(x, constraint)
            robust_constraints.append(robust_cons)

        # Optimizasyonu gerçekleştir
        result = minimize(robust_objective,
                          x0=self.nominal_problem.initial_guess,
                          constraints=robust_constraints,
                          method='SLSQP')

        return result.x, result.fun
```

12. ONLINE/GERÇEK ZAMANLI OPTİMİZASYON

Online optimizasyon, sürekli güncellenen veri akışı üzerinde çalışan ve gerçek zamanlı kararlar veren optimizasyon teknikleridir.

12.1 Online Gradyan İnişi

```python
class OnlineGradientDescent:
    def __init__(self, learning_rate=0.01, decay=0.999):
```

```
        self.learning_rate = learning_rate
        self.decay = decay
        self.t = 0  # Zaman adımı

    def update(self, x, gradient):
        """Tek bir online güncelleme adımı"""
        self.t += 1

        # Öğrenme oranını azalt
        current_lr = self.learning_rate * (self.decay ** self.t)

        # Parametreleri güncelle
        x_new = x - current_lr * gradient
        return x_new

    def optimize_stream(self, data_stream, initial_x, max_iterations=1000):
        """Veri akışı üzerinde optimizasyon"""
        x = initial_x
        trajectory = [x.copy()]

        for t, data_batch in enumerate(data_stream):
            if t >= max_iterations:
                break

            # Gradyanı hesapla
            gradient = self.compute_gradient(x, data_batch)

            # Parametreleri güncelle
            x = self.update(x, gradient)
            trajectory.append(x.copy())

        return x, trajectory
```

12.2 Adaptif Online Öğrenme

```
class AdaptiveOnlineLearner:
    def __init__(self, dimension, beta=0.9):
        self.dimension = dimension
        self.beta = beta

        # İkinci moment tahminini başlat
        self.v = np.zeros(dimension)

    def update(self, x, gradient):
        """Adaptif öğrenme oranlı güncelleme"""
        # İkinci moment tahminini güncelle
        self.v = self.beta * self.v + (1 - self.beta) * gradient**2

        # Adaptif öğrenme oranları
        adaptive_lr = 1 / np.sqrt(self.v + 1e-8)

        # Parametreleri güncelle
```

```
        x_new = x - adaptive_lr * gradient
        return x_new

    def optimize_stream(self, data_stream, initial_x, max_iterations=1000):
        x = initial_x
        regret = 0  # Kümülatif pişmanlık

        for t, data_batch in enumerate(data_stream):
            if t >= max_iterations:
                break

            # Mevcut kaybı hesapla
            current_loss = self.compute_loss(x, data_batch)

            # Gradyanı hesapla
            gradient = self.compute_gradient(x, data_batch)

            # Parametreleri güncelle
            x = self.update(x, gradient)

            # Pişmanlığı güncelle
            optimal_loss = self.compute_optimal_loss(data_batch)
            regret += current_loss - optimal_loss

        return x, regret
```

13. KUANTUM OPTİMİZASYON ALGORİTMALARI

13.1 Kuantum Tavlama (Quantum Annealing)

Kuantum tavlama, klasik tavlama algoritmasının kuantum tünelleme etkilerini kullanarak geliştirilmiş versiyonudur. Enerji minimizasyonu problemlerinde özellikle etkilidir.

```
class QuantumAnnealer:
    def __init__(self, hamiltonian, n_qubits, gamma=1.0):
        self.hamiltonian = hamiltonian  # Problem Hamiltonian'ı
        self.n_qubits = n_qubits
        self.gamma = gamma  # Tünelleme katsayısı

        # Kuantum devresini başlat
        self.circuit = QuantumCircuit(n_qubits)

    def initialize_superposition(self):
        """Tüm kubitleri süperpozisyona getir"""
        for qubit in range(self.n_qubits):
            self.circuit.h(qubit)  # Hadamard kapısı

    def apply_quantum_fluctuation(self, s):
        """Kuantum dalgalanmaları uygula"""
        # s: Tavlama programı parametresi (0 -> 1)
        strength = self.gamma * (1 - s)
```

```
        for qubit in range(self.n_qubits):
            self.circuit.rx(strength, qubit)  # X ekseni rotasyonu

    def apply_problem_hamiltonian(self, s):
        """Problem Hamiltonian'ını uygula"""
        strength = s

        for i, j, coupling in self.hamiltonian:
            self.circuit.czz(coupling * strength, i, j)

    def anneal(self, steps=1000, shots=1000):
        """Kuantum tavlama sürecini gerçekleştir"""
        self.initialize_superposition()

        for step in range(steps):
            s = step / steps  # Tavlama programı

            # Kuantum ve klasik etkileri dengele
            self.apply_quantum_fluctuation(s)
            self.apply_problem_hamiltonian(s)

        # Ölçüm yap
        self.circuit.measure_all()

        # Simüle et ve sonuçları al
        simulator = Aer.get_backend('qasm_simulator')
        result = execute(self.circuit, simulator, shots=shots).result()
        counts = result.get_counts()

        # En iyi çözümü bul
        best_state = max(counts.items(), key=lambda x: x[1])[0]
        return best_state, self.evaluate_solution(best_state)

    def evaluate_solution(self, state):
        """Çözümün enerjisini hesapla"""
        energy = 0
        state_array = np.array([int(bit) for bit in state])

        for i, j, coupling in self.hamiltonian:
            energy += coupling * state_array[i] * state_array[j]

        return energy
```

13.2 QAOA (Quantum Approximate Optimization Algorithm)

QAOA, kombinatoryal optimizasyon problemleri için tasarlanmış bir kuantum algoritmasıdır.

```
class QAOAOptimizer:
    def __init__(self, cost_hamiltonian, p_steps=1):
        self.cost_hamiltonian = cost_hamiltonian
```

```
        self.p_steps = p_steps  # QAOA derinliği

    def create_qaoa_circuit(self, beta, gamma):
        """QAOA devresini oluştur"""
        n_qubits = len(self.cost_hamiltonian)
        qc = QuantumCircuit(n_qubits)

        # Başlangıç durumu
        for qubit in range(n_qubits):
            qc.h(qubit)

        # QAOA katmanları
        for p in range(self.p_steps):
            # Problem Hamiltonian'ı
            for i, j, weight in self.cost_hamiltonian:
                qc.cx(i, j)
                qc.rz(2 * gamma[p] * weight, j)
                qc.cx(i, j)

            # Karıştırıcı Hamiltonian
            for qubit in range(n_qubits):
                qc.rx(2 * beta[p], qubit)

        return qc

    def compute_expectation(self, counts, shots):
        """Beklenen enerjiyi hesapla"""
        energy = 0
        for bitstring, count in counts.items():
            # Bit dizisini sayı dizisine çevir
            z = np.array([1 - 2 * int(bit) for bit in bitstring])

            # Enerjiyi hesapla
            config_energy = 0
            for i, j, weight in self.cost_hamiltonian:
                config_energy += weight * z[i] * z[j]

            energy += config_energy * count / shots

        return energy

    def optimize(self, n_iterations=100):
        """QAOA parametrelerini optimize et"""
        # Başlangıç parametreleri
        beta = np.random.uniform(0, np.pi, self.p_steps)
        gamma = np.random.uniform(0, 2*np.pi, self.p_steps)

        def objective(params):
            beta = params[:self.p_steps]
            gamma = params[self.p_steps:]

            # Devreyi oluştur ve simüle et
```

```
            qc = self.create_qaoa_circuit(beta, gamma)
            simulator = Aer.get_backend('qasm_simulator')
            shots = 1000
            counts = execute(qc, simulator, shots=shots).result().get_counts()

            # Beklenen enerjiyi hesapla
            return self.compute_expectation(counts, shots)

        # Klasik optimizasyon
        result = minimize(objective,
                          np.concatenate([beta, gamma]),
                          method='COBYLA')

        optimal_beta = result.x[:self.p_steps]
        optimal_gamma = result.x[self.p_steps:]

        return optimal_beta, optimal_gamma, result.fun

## 13.3 VQE (Variational Quantum Eigensolver)

VQE, kuantum-klasik hibrit bir algoritmadır ve özellikle kimyasal sistemlerin yer durumu enerjisini bulmak için kullanılır.

```python
class VariationalQuantumEigensolver:
    def __init__(self, hamiltonian, ansatz_depth=2):
        self.hamiltonian = hamiltonian
        self.ansatz_depth = ansatz_depth
        self.n_qubits = self._get_num_qubits()

    def create_ansatz(self, parameters):
        """Parametreli kuantum devresi oluştur"""
        qc = QuantumCircuit(self.n_qubits)
        param_idx = 0

        for d in range(self.ansatz_depth):
            # Tek kubit rotasyonları
            for q in range(self.n_qubits):
                qc.rx(parameters[param_idx], q)
                param_idx += 1
                qc.ry(parameters[param_idx], q)
                param_idx += 1
                qc.rz(parameters[param_idx], q)
                param_idx += 1

            # Entanglement katmanı
            for q in range(self.n_qubits-1):
                qc.cx(q, q+1)

        return qc

    def compute_expectation(self, parameters):
```
```

```
        """Hamiltonian'ın beklenen değerini hesapla"""
        qc = self.create_ansatz(parameters)

        # Pauli terimleri için ölçümler
        expectations = []
        weights = []

        for pauli_string, weight in self.hamiltonian:
            measured_qc = self.measure_pauli_string(qc, pauli_string)
            simulator = Aer.get_backend('qasm_simulator')
            counts = execute(measured_qc, simulator,
                             shots=1000).result().get_counts()

            # Beklenen değeri hesapla
            exp_val = 0
            for bitstring, count in counts.items():
                exp_val += (-1)**sum(int(bit) for bit in bitstring) * count/1000

            expectations.append(exp_val)
            weights.append(weight)

        return np.sum(np.array(weights) * np.array(expectations))

    def optimize(self, initial_params=None):
        """VQE optimizasyonunu gerçekleştir"""
        n_params = self.ansatz_depth * self.n_qubits * 3

        if initial_params is None:
            initial_params = np.random.uniform(-np.pi, np.pi, n_params)

        result = minimize(self.compute_expectation,
                          initial_params,
                          method='COBYLA',
                          options={'maxiter': 500})

        return result.x, result.fun
```

14. OPTİMİZASYON ALGORİTMALARININ KARŞILAŞTIRMALI ANALİZİ

14.1 Performans Karşılaştırma Çerçevesi

```
class OptimizationBenchmark:
    def __init__(self, test_functions, optimizers):
        self.test_functions = test_functions
        self.optimizers = optimizers

    def run_benchmark(self, n_trials=30, max_evaluations=10000):
        """Karşılaştırmalı performans analizi yap"""
        results = {
            'convergence_speed': {},
            'solution_quality': {},
            'robustness': {},
```

```
            'computational_cost': {}
        }

        for func_name, func in self.test_functions.items():
            for opt_name, optimizer in self.optimizers.items():
                # Her algoritma için çoklu deneme
                trials_data = []

                for trial in range(n_trials):
                    start_time = time.time()

                    # Optimizasyonu gerçekleştir
                    solution, trajectory = optimizer.optimize(
                        func,
                        max_evaluations=max_evaluations
                    )

                    end_time = time.time()

                    trials_data.append({
                        'solution': solution,
                        'final_value': func(solution),
                        'convergence': trajectory,
                        'time': end_time - start_time
                    })

                # Metrikleri hesapla
                results['convergence_speed'][f"{opt_name}_{func_name}"] = \
                    self.analyze_convergence(trials_data)
                results['solution_quality'][f"{opt_name}_{func_name}"] = \
                    self.analyze_quality(trials_data)
                results['robustness'][f"{opt_name}_{func_name}"] = \
                    self.analyze_robustness(trials_data)
                results['computational_cost'][f"{opt_name}_{func_name}"] = \
                    self.analyze_cost(trials_data)

        return results

    def analyze_convergence(self, trials_data):
        """Yakınsama hızını analiz et"""
        convergence_rates = []

        for trial in trials_data:
            trajectory = trial['convergence']
            # İlk %90 iyileşmeye kadar geçen iterasyon sayısı
            total_improvement = trajectory[0] - trajectory[-1]
            threshold = trajectory[0] - 0.9 * total_improvement

            for i, value in enumerate(trajectory):
                if value <= threshold:
                    convergence_rates.append(i)
                    break
```

```
        return {
            'mean_rate': np.mean(convergence_rates),
            'std_rate': np.std(convergence_rates)
        }

    def analyze_quality(self, trials_data):
        """Çözüm kalitesini analiz et"""
        final_values = [trial['final_value'] for trial in trials_data]

        return {
            'best': np.min(final_values),
            'mean': np.mean(final_values),
            'std': np.std(final_values)
        }

    def analyze_robustness(self, trials_data):
        """Algoritmanın gürbüzlüğünü analiz et"""
        solutions = np.array([trial['solution'] for trial in trials_data])

        return {
            'solution_spread': np.std(solutions, axis=0),
            'success_rate': np.mean([trial['final_value'] < 1e-6
                                     for trial in trials_data])
        }

    def analyze_cost(self, trials_data):
        """Hesaplama maliyetini analiz et"""
        times = [trial['time'] for trial in trials_data]

        return {
            'mean_time': np.mean(times),
            'std_time': np.std(times)
        }
```

15. GERÇEK DÜNYA UYGULAMALARI VE VAKA ÇALIŞMALARI

15.1 Endüstriyel Optimizasyon Uygulamaları

15.1.1 Üretim Planlama ve Çizelgeleme

Üretim ortamında karşılaşılan karmaşık çizelgeleme problemlerinin çözümü için geliştirilmiş hibrit optimizasyon yaklaşımı:

```
class ProductionScheduler:
    def __init__(self, resources, jobs, constraints):
        """
        resources: Mevcut makineler ve kapasiteleri
        jobs: Tamamlanması gereken işler ve özellikleri
```

```
        constraints: Üretim kısıtları (öncelik ilişkileri, teslim tarihleri vb.)
        """
        self.resources = resources
        self.jobs = jobs
        self.constraints = constraints

        # Çizelgeleme için karma tamsayılı programlama modeli
        self.model = self.create_optimization_model()

    def create_optimization_model(self):
        """Optimizasyon modelini oluştur"""
        model = Model("Production_Scheduling")

        # Karar değişkenleri
        # x[i,j,t]: i işinin j makinesinde t zamanında başlaması
        x = model.addVars([(i,j,t) for i in self.jobs
                                   for j in self.resources
                                   for t in self.time_periods],
                          vtype=GRB.BINARY,
                          name="assignment")

        # Kısıtlar
        # 1. Her iş tam olarak bir kez atanmalı
        for i in self.jobs:
            model.addConstr(quicksum(x[i,j,t]
                                   for j in self.resources
                                   for t in self.time_periods) == 1)

        # 2. Kapasite kısıtları
        for j in self.resources:
            for t in self.time_periods:
                model.addConstr(quicksum(x[i,j,t] * self.jobs[i].duration
                                         for i in self.jobs) <=
self.resources[j].capacity)

        # 3. Öncelik ilişkileri
        for (i1, i2) in self.constraints.precedence:
            model.addConstr(quicksum(t * x[i1,j,t]
                                   for j in self.resources
                                   for t in self.time_periods) +
                            self.jobs[i1].duration <=
                            quicksum(t * x[i2,j,t]
                                   for j in self.resources
                                   for t in self.time_periods))

        # Amaç fonksiyonu: Toplam tamamlanma süresini minimize et
        model.setObjective(quicksum(t * x[i,j,t] + self.jobs[i].duration
                                   for i in self.jobs
                                   for j in self.resources
                                   for t in self.time_periods),
                           GRB.MINIMIZE)
```

```
        return model

    def solve_with_decomposition(self):
        """Büyük problemler için ayrıştırma yaklaşımı"""
        # Problem boyutunu azaltmak için iş grupları oluştur
        job_groups = self.cluster_jobs()
        sub_schedules = []

        for group in job_groups:
            # Alt problem için yeni bir model oluştur
            sub_model = self.create_sub_problem(group)
            schedule = self.solve_sub_problem(sub_model)
            sub_schedules.append(schedule)

        # Alt çizelgeleri birleştir
        final_schedule = self.merge_schedules(sub_schedules)

        # Yerel arama ile iyileştir
        improved_schedule = self.local_search(final_schedule)

        return improved_schedule

    def solve_sub_problem(self, sub_model):
        """Alt problemi çöz ve sonuçları döndür"""
        # Zaman limitli çözüm
        sub_model.setParam('TimeLimit', 300)  # 5 dakika
        sub_model.optimize()

        if sub_model.status == GRB.OPTIMAL:
            return self.extract_schedule(sub_model)
        else:
            # Sezgisel yaklaşım kullan
            return self.apply_heuristic()

    def local_search(self, initial_schedule):
        """Çizelgeyi yerel arama ile iyileştir"""
        current_schedule = initial_schedule
        best_schedule = initial_schedule
        best_makespan = self.calculate_makespan(initial_schedule)

        for iteration in range(1000):  # Maksimum iterasyon
            # Komşu çözüm üret
            neighbor = self.generate_neighbor(current_schedule)

            # Komşu çözümün makespanını hesapla
            neighbor_makespan = self.calculate_makespan(neighbor)

            # Daha iyi çözüm bulunduysa güncelle
            if neighbor_makespan < best_makespan:
                best_schedule = neighbor
                best_makespan = neighbor_makespan
                current_schedule = neighbor
```

```
            else:
                # Simulated annealing kabul kriteri
                if random.random() < np.exp(-(neighbor_makespan - best_makespan) /
                                            self.temperature(iteration)):
                    current_schedule = neighbor

        return best_schedule
```

15.1.2 Kalite Kontrol Optimizasyonu

Üretim sürecinde kalite kontrol parametrelerinin optimizasyonu için çok amaçlı optimizasyon yaklaşımı:

```
class QualityOptimizer:
    def __init__(self, process_parameters, quality_metrics):
        self.parameters = process_parameters
        self.metrics = quality_metrics
        self.historical_data = []

    def evaluate_quality(self, parameters):
        """Kalite metriklerini hesapla"""
        quality_scores = {}

        for metric in self.metrics:
            # Yapay sinir ağı ile tahmin
            predicted_quality = self.predict_quality(parameters, metric)
            quality_scores[metric] = predicted_quality

        return quality_scores

    def optimize_parameters(self):
        """Pareto-optimal parametre setini bul"""
        # NSGA-II algoritması ile çok amaçlı optimizasyon
        problem = Problem(len(self.parameters), len(self.metrics))

        # Parametre sınırlarını tanımla
        problem.types[:] = [Real(param.lower, param.upper)
                            for param in self.parameters]

        # Amaç fonksiyonlarını tanımla
        problem.function = self.evaluate_objectives

        # Optimizasyonu gerçekleştir
        algorithm = NSGAII(problem)
        algorithm.run(10000)

        # Pareto cephesini döndür
        return algorithm.result

    def evaluate_objectives(self, parameters):
        """Çok amaçlı optimizasyon için amaç fonksiyonları"""
        quality_scores = self.evaluate_quality(parameters)
```

```
        # Kalite metriklerini normalize et
        normalized_scores = []
        for metric in self.metrics:
            score = quality_scores[metric]
            normalized = (score - metric.min_value) / \
                         (metric.max_value - metric.min_value)
            normalized_scores.append(normalized)

        return normalized_scores

    def implement_optimal_parameters(self, selected_solution):
        """Seçilen optimal parametreleri uygula"""
        # Parametre değişikliklerini kontrollü şekilde uygula
        for param, value in zip(self.parameters, selected_solution):
            # Güvenlik kontrolü
            if self.is_safe_change(param, value):
                # Parametreyi güncelle
                self.update_parameter(param, value)

                # Değişikliğin etkisini izle
                self.monitor_effect(param)
            else:
                logging.warning(f"Unsafe parameter change detected: {param}")
```

15.2 Finans Sektöründe Optimizasyon

15.2.1 Portföy Optimizasyonu

Modern portföy teorisi temelinde risk-getiri optimizasyonu:

```
class PortfolioOptimizer:
    def __init__(self, assets, returns, risk_free_rate=0.02):
        self.assets = assets
        self.returns = returns
        self.rf = risk_free_rate

        # Kovaryans matrisini hesapla
        self.cov_matrix = self.calculate_covariance()

        # Beklenen getirileri hesapla
        self.expected_returns = self.calculate_expected_returns()

    def optimize_portfolio(self, risk_tolerance=None, target_return=None):
        """Optimal portföy ağırlıklarını bul"""
        n_assets = len(self.assets)

        # Optimizasyon modeli
        model = Model("Portfolio_Optimization")

        # Karar değişkenleri: Portföy ağırlıkları
        weights = model.addVars(n_assets,
                                lb=0.0,  # Short satış yok
```

```
                            ub=1.0,
                            name="weights")

        # Kısıtlar
        # 1. Ağırlıklar toplamı 1 olmalı
        model.addConstr(quicksum(weights[i] for i in range(n_assets)) == 1)

        if target_return is not None:
            # 2. Hedef getiri kısıtı
            model.addConstr(quicksum(weights[i] * self.expected_returns[i]
                                for i in range(n_assets)) >= target_return)

        # Amaç fonksiyonu
        if risk_tolerance is not None:
            # Risk-getiri dengesini optimize et
            portfolio_return = quicksum(weights[i] * self.expected_returns[i]
                                    for i in range(n_assets))
            portfolio_risk = quicksum(weights[i] * weights[j] *
                                self.cov_matrix[i,j]
                                for i in range(n_assets)
                                for j in range(n_assets))

            model.setObjective(portfolio_return -
                            risk_tolerance * portfolio_risk,
                            GRB.MAXIMIZE)
        else:
            # Minimum varyans portföyü
            model.setObjective(quicksum(weights[i] * weights[j] *
                                    self.cov_matrix[i,j]
                                    for i in range(n_assets)
                                    for j in range(n_assets)),
                            GRB.MINIMIZE)

        # Modeli çöz
        model.optimize()

        return self.extract_solution(model, weights)

    def efficient_frontier(self, n_points=50):
        """Etkin sınırı hesapla"""
        min_ret = min(self.expected_returns)
        max_ret = max(self.expected_returns)
        target_returns = np.linspace(min_ret, max_ret, n_points)

        frontier_portfolios = []
        for target in target_returns:
            weights = self.optimize_portfolio(target_return=target)
            portfolio = {
                'weights': weights,
                'return': self.calculate_portfolio_return(weights),
                'risk': self.calculate_portfolio_risk(weights),
                'sharpe': self.calculate_sharpe_ratio(weights)
```

```
            }
            frontier_portfolios.append(portfolio)

        return frontier_portfolios
```

15.3 Lojistik ve Tedarik Zinciri Optimizasyonu

15.3.1 Rota Optimizasyonu (Vehicle Routing Problem)

Rota optimizasyonu, lojistik operasyonlarının verimliliğini artırmak için kullanılan temel bir optimizasyon problemidir. Aşağıdaki implementasyon, zaman pencereli ve kapasiteli araç rotalama problemini çözmektedir:

```
class VehicleRoutingOptimizer:
    def __init__(self, locations, demands, time_windows, vehicle_capacity):
        """
        locations: Teslimat noktalarının koordinatları
        demands: Her noktanın talep miktarı
        time_windows: Her nokta için teslimat zaman aralığı
        vehicle_capacity: Araç kapasitesi
        """
        self.locations = locations
        self.demands = demands
        self.time_windows = time_windows
        self.vehicle_capacity = vehicle_capacity

        # Mesafe matrisini hesapla
        self.distances = self.calculate_distance_matrix()

    def create_optimization_model(self):
        """Karma tamsayılı programlama modeli oluştur"""
        n_locations = len(self.locations)
        vehicles = range(self.n_vehicles)

        model = Model("VRP_Optimization")

        # Karar değişkenleri
        # x[i,j,k]: k aracı i'den j'ye giderse 1, değilse 0
        x = model.addVars([(i,j,k) for i in range(n_locations)
                                   for j in range(n_locations)
                                   for k in vehicles],
                          vtype=GRB.BINARY,
                          name="route")

        # y[i,k]: k aracının i noktasına varış zamanı
        y = model.addVars([(i,k) for i in range(n_locations)
                                 for k in vehicles],
                          name="arrival_time")
```

```
        # Kısıtlar
        # 1. Her nokta tam olarak bir kez ziyaret edilmeli
        for i in range(1, n_locations):  # Depo hariç
            model.addConstr(quicksum(x[i,j,k]
                                     for j in range(n_locations)
                                     for k in vehicles) == 1)

        # 2. Araç kapasitesi aşılmamalı
        for k in vehicles:
            model.addConstr(quicksum(self.demands[i] * x[i,j,k]
                                     for i in range(1, n_locations)
                                     for j in range(n_locations)) <=
                            self.vehicle_capacity)

        # 3. Zaman penceresi kısıtları
        M = 10000  # Büyük sayı
        for i in range(n_locations):
            for k in vehicles:
                model.addConstr(y[i,k] >= self.time_windows[i][0])
                model.addConstr(y[i,k] <= self.time_windows[i][1])

        # 4. Alt tur eliminasyon kısıtları
        for i in range(n_locations):
            for j in range(n_locations):
                for k in vehicles:
                    if i != j:
                        model.addConstr(y[i,k] + self.service_time[i] +
                                        self.travel_time[i][j] <=
                                        y[j,k] + M * (1 - x[i,j,k]))

        # Amaç fonksiyonu: Toplam mesafeyi minimize et
        model.setObjective(quicksum(self.distances[i][j] * x[i,j,k]
                                    for i in range(n_locations)
                                    for j in range(n_locations)
                                    for k in vehicles),
                           GRB.MINIMIZE)

        return model

    def solve_with_adaptive_large_neighborhood_search(self):
        """Büyük problemler için sezgisel çözüm yaklaşımı"""
        current_solution = self.construct_initial_solution()
        best_solution = current_solution
        best_cost = self.calculate_solution_cost(best_solution)

        for iteration in range(1000):  # Maksimum iterasyon
            # Mevcut çözümün bir kısmını yık
            destroyed_solution = self.destroy_solution(current_solution)

            # Çözümü yeniden inşa et
            repaired_solution = self.repair_solution(destroyed_solution)
```

```
            # Yeni çözümün maliyetini hesapla
            new_cost = self.calculate_solution_cost(repaired_solution)

            # Kabul kriteri
            if new_cost < best_cost:
                best_solution = repaired_solution
                best_cost = new_cost
                current_solution = repaired_solution
            elif self.accept_solution(new_cost, best_cost, iteration):
                current_solution = repaired_solution

        return best_solution, best_cost

    def destroy_solution(self, solution):
        """Çözümün bir kısmını rastgele yık"""
        # Yıkılacak müşteri sayısını belirle
        destroy_count = random.randint(5, len(solution.customers) // 4)

        # Rastgele müşterileri seç ve kaldır
        customers_to_remove = random.sample(solution.customers, destroy_count)
        return self.remove_customers(solution, customers_to_remove)

    def repair_solution(self, partial_solution):
        """Yıkılan çözümü yeniden inşa et"""
        # En yakın ekleme sezgiseli
        unassigned = self.get_unassigned_customers(partial_solution)

        while unassigned:
            # En iyi ekleme pozisyonunu bul
            best_insertion = self.find_best_insertion(unassigned,
                                                      partial_solution)

            # Müşteriyi ekle
            partial_solution = self.insert_customer(partial_solution,
                                                    best_insertion)
            unassigned.remove(best_insertion.customer)

        return partial_solution
```

15.3.2 Envanter Yönetimi Optimizasyonu

Stokastik talep altında envanter yönetimi için dinamik programlama yaklaşımı:

```
class InventoryOptimizer:
    def __init__(self, demand_distribution, holding_cost, ordering_cost,
                 stockout_cost, lead_time):
        self.demand_dist = demand_distribution
        self.holding_cost = holding_cost
        self.ordering_cost = ordering_cost
        self.stockout_cost = stockout_cost
        self.lead_time = lead_time
```

```python
    def optimize_policy(self, horizon, max_inventory):
        """Optimal (s,S) politikasını belirle"""
        # Durum uzayını oluştur
        states = range(-max_inventory, max_inventory + 1)
        actions = range(max_inventory + 1)  # Sipariş miktarları

        # Değer fonksiyonu ve politika matrisleri
        V = np.zeros((horizon + 1, len(states)))
        policy = np.zeros((horizon, len(states)), dtype=int)

        # Geriye doğru dinamik programlama
        for t in range(horizon - 1, -1, -1):
            for i, inventory in enumerate(states):
                min_cost = float('inf')
                best_action = 0

                for order in actions:
                    # Eğer sipariş verilecekse
                    if order > 0:
                        # Sipariş maliyeti
                        cost = self.ordering_cost + order * self.unit_cost
                    else:
                        cost = 0

                    # Beklenen maliyet hesabı
                    expected_cost = self.calculate_expected_cost(
                        inventory, order, V[t+1])

                    total_cost = cost + expected_cost

                    if total_cost < min_cost:
                        min_cost = total_cost
                        best_action = order

                V[t,i] = min_cost
                policy[t,i] = best_action

        return V, policy

    def calculate_expected_cost(self, inventory, order, future_values):
        """Bir durum-aksiyon çifti için beklenen maliyeti hesapla"""
        expected_cost = 0

        # Olası talep değerleri üzerinden beklenen değer
        for demand, prob in self.demand_dist.items():
            # Net envanter pozisyonu
            net_inventory = inventory + order - demand

            # Elde tutma maliyeti
            if net_inventory > 0:
                holding = self.holding_cost * net_inventory
            else:
```

```
                holding = 0

            # Stoksuzluk maliyeti
            if net_inventory < 0:
                stockout = self.stockout_cost * (-net_inventory)
            else:
                stockout = 0

            # Gelecek değer
            future = future_values[self.state_to_index(net_inventory)]

            expected_cost += prob * (holding + stockout + future)

        return expected_cost
```

15.3.3 Tesis Yerleşimi Optimizasyonu

Çok amaçlı tesis yerleşimi problemi için genetik algoritma tabanlı çözüm:

```
class FacilityLocationOptimizer:
    def __init__(self, demand_points, candidate_locations,
                 fixed_costs, transportation_costs):
        self.demand_points = demand_points
        self.candidates = candidate_locations
        self.fixed_costs = fixed_costs
        self.transport_costs = transportation_costs

    def optimize_locations(self, population_size=100, generations=500):
        """Genetik algoritma ile tesis yerleşimi optimizasyonu"""
        # Başlangıç popülasyonunu oluştur
        population = self.initialize_population(population_size)

        for generation in range(generations):
            # Uygunluk değerlerini hesapla
            fitness_values = [self.evaluate_fitness(solution)
                              for solution in population]

            # Yeni nesil için ebeveynleri seç
            parents = self.select_parents(population, fitness_values)

            # Yeni nesli oluştur
            new_population = []

            while len(new_population) < population_size:
                # Crossover
                parent1, parent2 = random.sample(parents, 2)
                child1, child2 = self.crossover(parent1, parent2)

                # Mutasyon
                child1 = self.mutate(child1)
                child2 = self.mutate(child2)
```

```
                new_population.extend([child1, child2])

            # Elitizm: En iyi çözümleri koru
            elite_count = population_size // 10
            elite_indices = np.argsort(fitness_values)[-elite_count:]
            elite_solutions = [population[i] for i in elite_indices]

            # Yeni nesli güncelle
            population = new_population[:-elite_count] + elite_solutions

        # En iyi çözümü döndür
        best_solution = max(population,
                            key=lambda x: self.evaluate_fitness(x))
        return best_solution

    def evaluate_fitness(self, solution):
        """Çözümün uygunluk değerini hesapla"""
        total_cost = 0

        # Sabit maliyetler
        for facility, is_open in enumerate(solution):
            if is_open:
                total_cost += self.fixed_costs[facility]

        # Taşıma maliyetleri
        for demand_point in range(len(self.demand_points)):
            # En yakın açık tesisi bul
            min_transport_cost = float('inf')
            for facility, is_open in enumerate(solution):
                if is_open:
                    cost = self.transport_costs[demand_point][facility]
                    min_transport_cost = min(min_transport_cost, cost)

            total_cost += min_transport_cost

        # Kapsama oranını hesapla
        coverage = self.calculate_coverage(solution)

        # Çok amaçlı uygunluk değeri
        return -total_cost + 1000 * coverage  # Ağırlıklı toplam

    def calculate_coverage(self, solution):
        """Talep noktalarının kapsanma oranını hesapla"""
        covered_points = 0

        for point in range(len(self.demand_points)):
            for facility, is_open in enumerate(solution):
                if is_open:
                    distance = self.calculate_distance(
                        self.demand_points[point],
                        self.candidates[facility]
                    )
```

```
                    if distance <= self.coverage_radius:
                        covered_points += 1
                        break

        return covered_points / len(self.demand_points)
```

15.4 Enerji Sistemleri Optimizasyonu

15.4.1 Güç Dağıtım Optimizasyonu

Optimal güç akışı problemi çözümü için interior point yöntemi:

```
class PowerFlowOptimizer:
    def __init__(self, network_data, constraints, cost_functions):
        self.network = network_data
        self.constraints = constraints
        self.cost_functions = cost_functions

    def optimize_power_flow(self):
        """Optimal güç akışı problemi çözümü"""
        # Başlangıç noktası
        x0 = self.initialize_variables()

        # Interior point yöntemi parametreleri
        mu = 1.0  # Bariyer parametresi
        epsilon = 1e-6  # Tolerans

        while mu > epsilon:
            # Newton adımını hesapla
            dx = self.compute_newton_step(x0, mu)

            # Line search ile adım boyutunu belirle
            alpha = self.backtracking_line_search(x0, dx, mu)

            # Değişkenleri güncelle
            x0 = x0 + alpha * dx

            # Bariyer parametresini güncelle
            mu = 0.1 * mu

        return x0

    def compute_newton_step(self, x, mu):
        """Newton adımını hesapla"""
        # Gradyan ve Hessian hesabı
        gradient = self.compute_gradient(x, mu)
        hessian = self.compute_hessian(x, mu)

        # Newton sistemi çözümü
        dx = -np.linalg.solve(hessian, gradient)
        return dx
```

```
    def compute_gradient(self, x, mu):
        """Gradyan vektörünü hesapla"""
        # Amaç fonksiyonu gradyanı
        obj_grad = self.objective_gradient(x)

        # Eşitlik kısıtları gradyanı
        eq_grad = self.equality_constraints_gradient(x)

        # Eşitsizlik kısıtları gradyanı
        ineq_grad = self.inequality_constraints_gradient(x)

        # Bariyer terimi gradyanı
        barrier_grad = self.barrier_gradient(x, mu)

        return obj_grad + eq_grad + ineq_grad + barrier_grad

    def compute_hessian(self, x, mu):
        """Hessian matrisini hesapla"""
        # Amaç fonksiyonu Hessian'ı
        obj_hess = self.objective_hessian(x)

        # Kısıtlar Hessian'ı
        cons_hess = self.constraints_hessian(x)

        # Bariyer terimi Hessian'ı
        barrier_hess = self.barrier_hessian(x, mu)

        return obj_hess + cons_hess + barrier_hess

## 15.4.2 Yenilenebilir Enerji Planlaması

Karma enerji sistemleri için çok periyotlu optimizasyon:

```python
class RenewableEnergyPlanner:
    def __init__(self, renewable_sources, storage_systems, load_profile):
        """
        renewable_sources: Her yenilenebilir kaynak için üretim profili
        storage_systems: Enerji depolama sistemleri özellikleri
        load_profile: Talep profili
        """
        self.sources = renewable_sources
        self.storage = storage_systems
        self.load = load_profile
        self.time_periods = len(load_profile)

    def optimize_system(self):
        """Karma enerji sistemini optimize et"""
        model = Model("Renewable_Energy_System")

        # Karar değişkenleri
```
```

```
        # Yenilenebilir kaynaklardan üretim
        generation = model.addVars(
            [(s, t) for s in self.sources
                    for t in range(self.time_periods)],
            name="generation"
        )

        # Depolama sistemi şarj/deşarj
        charge = model.addVars(
            [(b, t) for b in self.storage
                    for t in range(self.time_periods)],
            name="charge"
        )
        discharge = model.addVars(
            [(b, t) for b in self.storage
                    for t in range(self.time_periods)],
            name="discharge"
        )

        # Depolama seviyesi
        storage_level = model.addVars(
            [(b, t) for b in self.storage
                    for t in range(self.time_periods)],
            name="storage_level"
        )

        # Kısıtlar
        # 1. Güç dengesi
        for t in range(self.time_periods):
            model.addConstr(
                quicksum(generation[s,t] for s in self.sources) +
                quicksum(discharge[b,t] - charge[b,t] for b in self.storage) ==
                self.load[t]
            )

        # 2. Yenilenebilir üretim kısıtları
        for s in self.sources:
            for t in range(self.time_periods):
                model.addConstr(
                    generation[s,t] <= self.sources[s].capacity *
                                       self.sources[s].availability[t]
                )

        # 3. Depolama dinamikleri
        for b in self.storage:
            for t in range(self.time_periods):
                if t == 0:
                    model.addConstr(
                        storage_level[b,t] == self.storage[b].initial_level +
                        charge[b,t] * self.storage[b].charge_efficiency -
                        discharge[b,t] / self.storage[b].discharge_efficiency
                    )
```

```
                else:
                    model.addConstr(
                        storage_level[b,t] == storage_level[b,t-1] +
                        charge[b,t] * self.storage[b].charge_efficiency -
                        discharge[b,t] / self.storage[b].discharge_efficiency
                    )

        # 4. Depolama kapasite kısıtları
        for b in self.storage:
            for t in range(self.time_periods):
                model.addConstr(
                    storage_level[b,t] <= self.storage[b].capacity
                )
                model.addConstr(
                    storage_level[b,t] >= self.storage[b].min_level
                )

        # Amaç fonksiyonu: Toplam maliyeti minimize et
        objective = quicksum(
            self.sources[s].cost * generation[s,t]
            for s in self.sources
            for t in range(self.time_periods)
        ) + quicksum(
            self.storage[b].cost * (charge[b,t] + discharge[b,t])
            for b in self.storage
            for t in range(self.time_periods)
        )

        model.setObjective(objective, GRB.MINIMIZE)
        model.optimize()

        return self.extract_solution(model, generation, storage_level)

## 15.4.3 Enerji Depolama Optimizasyonu

Enerji depolama sistemleri için stokastik optimizasyon:

```python
class EnergyStorageOptimizer:
    def __init__(self, storage_params, price_scenarios, load_scenarios):
        self.storage = storage_params
        self.price_scenarios = price_scenarios
        self.load_scenarios = load_scenarios
        self.n_scenarios = len(price_scenarios)

    def optimize_operation(self):
        """Depolama sistemi operasyonunu optimize et"""
        model = Model("Storage_Optimization")

        # Birinci aşama değişkenleri (kapasite kararları)
        capacity = model.addVar(name="capacity")
        power_rating = model.addVar(name="power_rating")
```
```

```
        # İkinci aşama değişkenleri (operasyonel kararlar)
        charge = model.addVars(
            [(s,t) for s in range(self.n_scenarios)
                   for t in range(self.time_periods)],
            name="charge"
        )
        discharge = model.addVars(
            [(s,t) for s in range(self.n_scenarios)
                   for t in range(self.time_periods)],
            name="discharge"
        )
        energy_level = model.addVars(
            [(s,t) for s in range(self.n_scenarios)
                   for t in range(self.time_periods)],
            name="energy_level"
        )

        # Kısıtlar
        # 1. Kapasite kısıtları
        for s in range(self.n_scenarios):
            for t in range(self.time_periods):
                model.addConstr(charge[s,t] <= power_rating)
                model.addConstr(discharge[s,t] <= power_rating)
                model.addConstr(energy_level[s,t] <= capacity)

        # 2. Enerji dengesi
        for s in range(self.n_scenarios):
            for t in range(self.time_periods):
                if t == 0:
                    model.addConstr(
                        energy_level[s,t] == self.storage.initial_level +
                        charge[s,t] * self.storage.charge_efficiency -
                        discharge[s,t] / self.storage.discharge_efficiency
                    )
                else:
                    model.addConstr(
                        energy_level[s,t] == energy_level[s,t-1] +
                        charge[s,t] * self.storage.charge_efficiency -
                        discharge[s,t] / self.storage.discharge_efficiency
                    )

        # Amaç fonksiyonu: Beklenen net geliri maksimize et
        objective = -self.storage.capital_cost * capacity - \
                   self.storage.power_cost * power_rating + \
                   (1/self.n_scenarios) * quicksum(
                       self.price_scenarios[s][t] *
                       (discharge[s,t] - charge[s,t])
                       for s in range(self.n_scenarios)
                       for t in range(self.time_periods)
                   )
```

```
        model.setObjective(objective, GRB.MAXIMIZE)
        model.optimize()

        return self.extract_solution(model, capacity, power_rating,
                                     energy_level)
```

15.4.4 Mikroşebeke Optimizasyonu

Mikroşebeke sistemlerinin gerçek zamanlı kontrolü için model öngörülü kontrol:

```
class MicrogridController:
    def __init__(self, microgrid_components, forecast_model,
                 prediction_horizon=24):
        self.components = microgrid_components
        self.forecast = forecast_model
        self.horizon = prediction_horizon

    def optimize_control(self, current_state):
        """Model öngörülü kontrol optimizasyonu"""
        # Tahminleri güncelle
        load_forecast = self.forecast.predict_load(self.horizon)
        renewable_forecast = self.forecast.predict_renewable(self.horizon)
        price_forecast = self.forecast.predict_prices(self.horizon)

        # Optimizasyon modeli
        model = Model("Microgrid_MPC")

        # Kontrol değişkenleri
        generator_power = model.addVars(
            [(g,t) for g in self.components.generators
                   for t in range(self.horizon)],
            name="generator_power"
        )
        storage_power = model.addVars(
            [(b,t) for b in self.components.batteries
                   for t in range(self.horizon)],
            lb=-GRB.INFINITY,
            name="storage_power"
        )
        grid_power = model.addVars(
            self.horizon,
            lb=-GRB.INFINITY,
            name="grid_power"
        )

        # Sistem durumu değişkenleri
        battery_energy = model.addVars(
            [(b,t) for b in self.components.batteries
                   for t in range(self.horizon+1)],
            name="battery_energy"
        )
```

```
        # Başlangıç durumu kısıtları
        for b in self.components.batteries:
            model.addConstr(
                battery_energy[b,0] == current_state.battery_levels[b]
            )

        # Güç dengesi kısıtları
        for t in range(self.horizon):
            model.addConstr(
                quicksum(generator_power[g,t]
                        for g in self.components.generators) +
                quicksum(storage_power[b,t]
                        for b in self.components.batteries) +
                grid_power[t] + renewable_forecast[t] == load_forecast[t]
            )

        # Batarya dinamikleri
        for b in self.components.batteries:
            for t in range(self.horizon):
                model.addConstr(
                    battery_energy[b,t+1] == battery_energy[b,t] -
                    storage_power[b,t] * self.components.batteries[b].efficiency
                )

        # Operasyonel kısıtlar
        for t in range(self.horizon):
            # Jeneratör limitleri
            for g in self.components.generators:
                model.addConstr(
                    generator_power[g,t] <=
                    self.components.generators[g].max_power
                )
                model.addConstr(
                    generator_power[g,t] >=
                    self.components.generators[g].min_power
                )

            # Batarya limitleri
            for b in self.components.batteries:
                model.addConstr(
                    battery_energy[b,t] <=
                    self.components.batteries[b].capacity
                )
                model.addConstr(
                    battery_energy[b,t] >=
                    self.components.batteries[b].min_energy
                )

            # Şebeke limitleri
            model.addConstr(
                grid_power[t] <= self.components.grid.import_limit
            )
```

```
            model.addConstr(
                grid_power[t] >= -self.components.grid.export_limit
            )

        # Amaç fonksiyonu
        operating_cost = quicksum(
            self.components.generators[g].cost * generator_power[g,t]
            for g in self.components.generators
            for t in range(self.horizon)
        ) + quicksum(
            price_forecast[t] * grid_power[t]
            for t in range(self.horizon)
        )

        model.setObjective(operating_cost, GRB.MINIMIZE)
        model.optimize()

        return self.extract_control_actions(model, generator_power,
                                            storage_power, grid_power)
```

16. YAPAY ZEKA TABANLI OPTİMİZASYON

16.1 Derin Pekiştirmeli Öğrenme ile Optimizasyon

Karmaşık optimizasyon problemlerinin çözümü için derin pekiştirmeli öğrenme yaklaşımı:

```
class DRLOptimizer:
    def __init__(self, state_dim, action_dim, hidden_dim=256):
        self.actor = ActorNetwork(state_dim, action_dim, hidden_dim)
        self.critic = CriticNetwork(state_dim, action_dim, hidden_dim)
        self.memory = ReplayBuffer(capacity=100000)

        self.actor_optimizer = torch.optim.Adam(self.actor.parameters())
        self.critic_optimizer = torch.optim.Adam(self.critic.parameters())

    def optimize_problem(self, environment, episodes=1000):
        """Optimizasyon problemini DRL ile çöz"""
        for episode in range(episodes):
            state = environment.reset()
            episode_reward = 0
            done = False

            while not done:
                # Eylem seç
                action = self.select_action(state)

                # Eylemi uygula
                next_state, reward, done, _ = environment.step(action)

                # Deneyimi kaydet
                self.memory.push(state, action, reward, next_state, done)
```

```
                # Ağları güncelle
                if len(self.memory) > self.batch_size:
                    self.update_networks()

                state = next_state
                episode_reward += reward

            # Sonuçları değerlendir ve kaydet
            self.evaluate_performance(episode, episode_reward)

    def update_networks(self):
        """Aktör ve kritik ağlarını güncelle"""
        # Deneyim örnekleri al
        states, actions, rewards, next_states, dones = \
            self.memory.sample(self.batch_size)

        # Kritik ağı güncelle
        current_Q = self.critic(states, actions)
        next_actions = self.actor_target(next_states)
        next_Q = self.critic_target(next_states, next_actions)
        target_Q = rewards + self.gamma * next_Q * (1 - dones)

        critic_loss = F.mse_loss(current_Q, target_Q.detach())

        self.critic_optimizer.zero_grad()
        critic_loss.backward()
        self.critic_optimizer.step()

        # Aktör ağı güncelle
        actor_loss = -self.critic(states, self.actor(states)).mean()

        self.actor_optimizer.zero_grad()
        actor_loss.backward()
        self.actor_optimizer.step()
```

16.2 Meta-Öğrenme ile Optimizasyon

Meta-öğrenme yaklaşımı ile optimizasyon algoritmalarının otomatik öğrenilmesi:

```
class MetaOptimizer:
    def __init__(self, problem_generator, meta_lr=0.001):
        self.problem_generator = problem_generator
        self.meta_lr = meta_lr
        self.meta_optimizer = MAML(model=OptimizationNetwork(),
                                   lr=meta_lr)

    def meta_train(self, n_tasks=1000, n_adaptation_steps=5):
        """Meta-öğrenme ile optimizasyon stratejisi öğren"""
        for task_batch in range(n_tasks):
            meta_train_loss = 0.0
            meta_valid_loss = 0.0
```

```
            # Görev örneklemi
            tasks = self.sample_tasks()

            for task in tasks:
                # Hızlı adaptasyon
                adapted_model = self.adapt_to_task(task, n_adaptation_steps)

                # Meta-güncelleme için değerlendirme
                meta_train_loss += self.evaluate_model(adapted_model, task)

                # Validasyon görevi üzerinde değerlendirme
                valid_task = self.sample_validation_task()
                meta_valid_loss += self.evaluate_model(adapted_model,
                                                       valid_task)

            # Meta-optimizasyon adımı
            self.meta_optimizer.step(meta_train_loss / len(tasks))

            # Performansı izle
            self.log_performance(task_batch, meta_train_loss, meta_valid_loss)

    def adapt_to_task(self, task, n_steps):
        """Tek bir göreve hızlı adaptasyon"""
        adapted_model = self.meta_optimizer.clone()

        for step in range(n_steps):
            train_inputs, train_targets = task.get_train_batch()

            # İleri yayılım
            predictions = adapted_model(train_inputs)
            loss = F.mse_loss(predictions, train_targets)

            # Geri yayılım ve güncelleme
            adapted_model.adapt(loss)

        return adapted_model
```

16.3 Otomatik Makine Öğrenmesi (AutoML) ile Optimizasyon

Hiperparametre optimizasyonu ve model seçimi için AutoML yaklaşımı:

```
class AutoMLOptimizer:
    def __init__(self, problem_space, evaluation_metric, time_budget):
        self.problem_space = problem_space
        self.metric = evaluation_metric
        self.time_budget = time_budget

        self.search_space = self.define_search_space()
        self.meta_learner = self.initialize_meta_learner()

    def optimize(self):
```

```
        """AutoML süreci ile optimal modeli bul"""
        start_time = time.time()
        best_config = None
        best_performance = float('-inf')

        while time.time() - start_time < self.time_budget:
            # Meta-öğrenme ile yeni konfigürasyon öner
            config = self.meta_learner.suggest_configuration()

            # Modeli eğit ve değerlendir
            model = self.train_model(config)
            performance = self.evaluate_model(model)

            # Meta-öğreniciyi güncelle
            self.meta_learner.update(config, performance)

            # En iyi sonucu güncelle
            if performance > best_performance:
                best_performance = performance
                best_config = config

            # Ara sonuçları raporla
            self.log_progress(config, performance)

        return best_config, best_performance
```

17. HİBRİT OPTİMİZASYON SİSTEMLERİ

17.1 Çok Seviyeli Optimizasyon

İç içe optimizasyon problemleri için hibrit çözüm yaklaşımı:

```
class MultiLevelOptimizer:
    def __init__(self, upper_level_problem, lower_level_problem):
        self.upper_problem = upper_level_problem
        self.lower_problem = lower_level_problem

    def optimize(self):
        """Çok seviyeli optimizasyon gerçekleştir"""
        upper_solution = None
        lower_solution = None

        # Dış döngü (üst seviye problemi)
        for outer_iteration in range(self.max_outer_iterations):
            # Üst seviye değişkenlerini güncelle
            upper_candidate = self.optimize_upper_level(lower_solution)

            # İç döngü (alt seviye problemi)
            lower_solution = self.optimize_lower_level(upper_candidate)

            # Yakınsama kontrolü
```

```
            if self.check_convergence(upper_solution, upper_candidate):
                break

            upper_solution = upper_candidate

        return upper_solution, lower_solution
```

17.2 Paralel Hibrit Optimizasyon

Farklı optimizasyon algoritmalarının paralel çalıştırılması:

```
class ParallelHybridOptimizer:
    def __init__(self, algorithms, problem):
        self.algorithms = algorithms
        self.problem = problem

    def optimize(self):
        """Paralel hibrit optimizasyon gerçekleştir"""
        # Paralel işlemciler başlat
        with ProcessPoolExecutor() as executor:
            # Her algoritmayı paralel çalıştır
            future_to_alg = {
                executor.submit(alg.optimize, self.problem): alg
                for alg in self.algorithms
            }

            results = []

            # Sonuçları topla
            for future in as_completed(future_to_alg):
                alg = future_to_alg[future]
                try:
                    solution = future.result()
                    results.append((alg, solution))
                except Exception as e:
                    print(f"Algorithm {alg} failed: {e}")

        # En iyi sonucu seç
        best_solution = self.select_best_solution(results)
        return best_solution
```

18. YENİ NESİL OPTİMİZASYON TEKNİKLERİ

18.1 Kuantum-İlham Optimizasyon

Kuantum hesaplama prensiplerinden ilham alan optimizasyon yaklaşımı:

```
class QuantumInspiredOptimizer:
    def __init__(self, n_qubits, n_populations):
        self.n_qubits = n_qubits
        self.n_populations = n_populations
```

```
        self.quantum_register = self.initialize_quantum_register()

    def optimize(self, objective_function, n_iterations):
        """Kuantum-ilham optimizasyon gerçekleştir"""
        best_solution = None
        best_fitness = float('-inf')

        for iteration in range(n_iterations):
            # Kuantum durumlarını güncelle
            self.update_quantum_states()

            # Klasik popülasyonu oluştur
            classical_population = self.measure_quantum_states()

            # Değerlendir ve en iyiyi güncelle
            for solution in classical_population:
                fitness = objective_function(solution)
                if fitness > best_fitness:
                    best_fitness = fitness
                    best_solution = solution

            # Kuantum kapılarını uygula
            self.apply_quantum_gates(best_solution)

        return best_solution, best_fitness
```

18.2 Nöro-Evrimsel Optimizasyon

Sinir ağları ve evrimsel algoritmaların birleşimi:

```
class NeuroEvolutionaryOptimizer:
    def __init__(self, network_architecture, population_size=100):
        self.architecture = network_architecture
        self.population_size = population_size
        self.population = self.initialize_population()

    def optimize(self, n_generations):
        """Nöro-evrimsel optimizasyon gerçekleştir"""
        for generation in range(n_generations):
            # Popülasyonu değerlendir
            fitness_scores = self.evaluate_population()

            # Yeni nesil için ebeveynleri seç
            parents = self.select_parents(fitness_scores)

            # Yeni nesli oluştur
            offspring = self.create_offspring(parents)

            # Mutasyon uygula
            self.mutate_population(offspring)

            # Popülasyonu güncelle
```

```
            self.population = self.update_population(offspring)

        return self.get_best_network()
```

19. SONUÇ VE GELECEK YÖNELIMLER

19.1 Optimizasyon Algoritmalarının Karşılaştırmalı Analizi

Farklı optimizasyon yaklaşımlarının performans karşılaştırması:

1. Hesaplama Verimliliği:
    - Klasik yöntemler: O(n²) - O(n³)
    - Meta-sezgisel yöntemler: O(n*log(n))
    - Yapay zeka tabanlı yöntemler: Model karmaşıklığına bağlı
2. Çözüm Kalitesi:
    - Deterministik yöntemler: Global optimum garanti (konveks problemlerde)
    - Meta-sezgisel yöntemler: İyi yerel optimumlar
    - Hibrit yöntemler: Yüksek kaliteli çözümler
3. Ölçeklenebilirlik:
    - Klasik yöntemler: Düşük ölçeklenebilirlik
    - Dağıtık yöntemler: Yüksek ölçeklenebilirlik
    - Paralel hibrit yöntemler: Çok yüksek ölçeklenebilirlik

19.2 Gelecek Araştırma Yönelimleri

1. Kuantum Optimizasyon:
    - Kuantum algoritmaların geliştirilmesi
    - Hibrit kuantum-klasik yaklaşımlar
    - Kuantum-ilham algoritmalar
2. Yapay Zeka Entegrasyonu:
    - AutoML tabanlı optimizasyon
    - Meta-öğrenme yaklaşımları
    - Derin pekiştirmeli öğrenme uygulamaları
3. Sürdürülebilir Optimizasyon:
    - Enerji verimli algoritmalar
    - Yeşil hesaplama teknikleri
    - Çevresel etki minimizasyonu

19.3 Pratik Uygulama Önerileri

1. Problem Analizi:
    - Problemin yapısını detaylı analiz et
    - Kısıtları ve hedefleri net belirle

- Uygun optimizasyon yaklaşımını seç

2. Algoritma Seçimi:
   - Problem boyutunu değerlendir
   - Hesaplama kaynaklarını göz önünde bulundur
   - Çözüm kalitesi gereksinimlerini belirle
3. Uygulama Stratejisi:
   - Prototip ile başla
   - Kademeli iyileştirme yap
   - Sürekli performans izleme gerçekleştir

20. UYGULAMA REHBERİ VE EN İYİ UYGULAMALAR

20.1 Optimizasyon Algoritması Seçim Rehberi

Problem özelliklerine göre algoritma seçimi için karar çerçevesi:

```
class AlgorithmSelector:
    def __init__(self, problem_characteristics):
        self.characteristics = problem_characteristics
        self.algorithm_database = self.initialize_algorithm_database()

    def select_algorithm(self):
        """Problem özelliklerine göre en uygun algoritmayı seç"""
        scores = {}

        for algorithm in self.algorithm_database:
            # Algoritma uygunluğunu değerlendir
            score = self.evaluate_algorithm_fitness(algorithm)
            scores[algorithm] = score

        # En uygun algoritmayı öner
        best_algorithm = max(scores.items(), key=lambda x: x[1])[0]
        alternatives = self.find_alternatives(scores, best_algorithm)

        return {
            'primary_choice': best_algorithm,
            'alternatives': alternatives,
            'reasoning': self.generate_recommendation_report(best_algorithm)
        }

    def evaluate_algorithm_fitness(self, algorithm):
        """Algoritmanın problem için uygunluğunu hesapla"""
        score = 0

        # Problem boyutu değerlendirmesi
        if self.characteristics['problem_size'] == 'large':
            score += algorithm.scalability_score * 2

        # Kısıt değerlendirmesi
```

```
        if self.characteristics['constraints'] == 'complex':
            score += algorithm.constraint_handling_score * 1.5

        # Çözüm kalitesi gereksinimleri
        if self.characteristics['solution_quality'] == 'high':
            score += algorithm.solution_quality_score * 2

        # Hesaplama kaynakları
        if self.characteristics['computational_resources'] == 'limited':
            score += algorithm.efficiency_score

        return score
```

20.2 Performans Optimizasyon Stratejileri

Algoritma performansını iyileştirmek için teknikler:

```
class PerformanceOptimizer:
    def __init__(self, algorithm):
        self.algorithm = algorithm
        self.profiler = PerformanceProfiler()

    def optimize_performance(self):
        """Algoritma performansını optimize et"""
        # Performans analizi
        profile_results = self.profiler.analyze(self.algorithm)

        # Darboğaz tespiti
        bottlenecks = self.identify_bottlenecks(profile_results)

        # İyileştirme önerileri
        optimizations = self.suggest_optimizations(bottlenecks)

        # İyileştirmeleri uygula
        for optimization in optimizations:
            self.apply_optimization(optimization)

        # Sonuçları değerlendir
        improved_results = self.profiler.analyze(self.algorithm)
        return self.generate_optimization_report(profile_results,
                                                 improved_results)
```

20.3 Hata Toleransı ve Gürbüzlük

Algoritmaların gürbüzlüğünü artırmak için yaklaşımlar:

```
class RobustnessEnhancer:
    def __init__(self, algorithm, error_tolerance=0.01):
        self.algorithm = algorithm
        self.tolerance = error_tolerance

    def enhance_robustness(self):
```

```
        """Algoritmanın gürbüzlüğünü artır"""
        # Hata tespiti
        error_cases = self.identify_error_cases()

        # Gürbüzlük iyileştirmeleri
        self.implement_error_handling()
        self.add_validation_checks()
        self.implement_recovery_mechanisms()

        # Gürbüzlük testi
        robustness_score = self.test_robustness()
        return robustness_score
```

21. GELECEK TEKNOLOJİLER VE TRENDLER

21.1 Yapay Zeka ve Optimizasyon Entegrasyonu

Gelecekteki entegrasyon senaryoları:

1. Otomatik Problem Tanıma:
    - Problem özelliklerinin otomatik tespiti
    - Uygun algoritma seçimi
    - Hiperparametre optimizasyonu
2. Adaptif Öğrenme:
    - Çalışma zamanında algoritma adaptasyonu
    - Dinamik parametre ayarı
    - Performans geri beslemesi
3. Hibrit Sistemler:
    - AI-destekli optimizasyon
    - Transfer öğrenme entegrasyonu
    - Meta-öğrenme yaklaşımları

21.2 Sürdürülebilir Optimizasyon

Yeşil hesaplama yaklaşımları:

```
class GreenOptimizer:
    def __init__(self, base_algorithm):
        self.algorithm = base_algorithm
        self.energy_monitor = EnergyMonitor()

    def optimize_energy_usage(self):
        """Enerji tüketimini optimize et"""
        # Enerji profili çıkar
        energy_profile = self.energy_monitor.profile_algorithm(self.algorithm)

        # Enerji verimliliği iyileştirmeleri
        optimizations = self.identify_energy_optimizations(energy_profile)
```

```
        # İyileştirmeleri uygula
        for opt in optimizations:
            self.apply_energy_optimization(opt)

        # Sonuçları değerlendir
        new_profile = self.energy_monitor.profile_algorithm(self.algorithm)
        return self.calculate_energy_savings(energy_profile, new_profile)
```

22. SONUÇ VE ÖNERİLER

22.1 Algoritma Seçim Özeti

Farklı problem tipleri için önerilen algoritmalar:

1. Büyük Ölçekli Problemler:
    - Dağıtık optimizasyon
    - Paralel hibrit yaklaşımlar
    - Adaptif meta-sezgiseller
2. Gerçek Zamanlı Uygulamalar:
    - Model öngörülü kontrol
    - Online optimizasyon
    - Hafif meta-sezgiseller
3. Karmaşık Kısıtlı Problemler:
    - Hibrit yöntemler
    - Çok seviyeli optimizasyon
    - Kısıt programlama

22.2 Uygulama Önerileri

Başarılı bir optimizasyon projesi için öneriler:

1. Planlama Aşaması:
    - Detaylı problem analizi
    - Gereksinim belirleme
    - Kaynak değerlendirmesi
2. Geliştirme Aşaması:
    - Kademeli yaklaşım
    - Sürekli test ve validasyon
    - Performans izleme
3. Optimizasyon Aşaması:
    - Parametre ayarı
    - Algoritma ince ayarı
    - Gürbüzlük testleri

22.3 Gelecek Çalışmalar

Araştırma ve geliştirme önerileri:

1. Teorik Çalışmalar:
    - Yeni algoritma geliştirme
    - Performans analizi
    - Yakınsama garantileri
2. Uygulama Geliştirmeleri:
    - Endüstriyel uygulamalar
    - Gerçek zamanlı sistemler
    - Büyük ölçekli problemler
3. Teknoloji Entegrasyonu:
    - Kuantum hesaplama
    - Edge computing
    - IoT entegrasyonu

Bu kapsamlı doküman, optimizasyon algoritmalarının teorik temellerinden pratik uygulamalarına kadar geniş bir yelpazede bilgi sunmaktadır. Sürekli gelişen teknoloji ve yeni yaklaşımlarla birlikte, bu alandaki çalışmaların devam etmesi ve yeni çözümlerin ortaya çıkması beklenmektedir.