HİPERPARAMETRE OPTİMİZASYONU

222923068 ECE İREY

212923041 EMİN KORAY UÇA

222923069 GÜLİSTAN ÇAPUTKAN

212923007 FADİME AKTAŞ

İÇİNDEKİLER

HİPERPARAMETRE OPTİMİZASYONU VE DERİN ÖĞRENME TASARIMI

Derin öğrenme modelleri, büyük miktarda veriyi işlemek için karmaşık yapılar gerektirir. Bu yapıların başarısı, doğru hiperparametrelerin seçilmesine bağlıdır.

Hiperparametreler, modelin performansını doğrudan etkileyen ve önceden belirlenmesi gereken ayarlardır. Örneğin:

- Kaç katman olacağı.
- Her katmanda kaç nöron olacağı.
- Hangi aktivasyon fonksiyonlarının tercih edileceği.

Bu seçimler, **modelin doğruluğunu**, **hızını** ve **genelleme performansını** belirler.

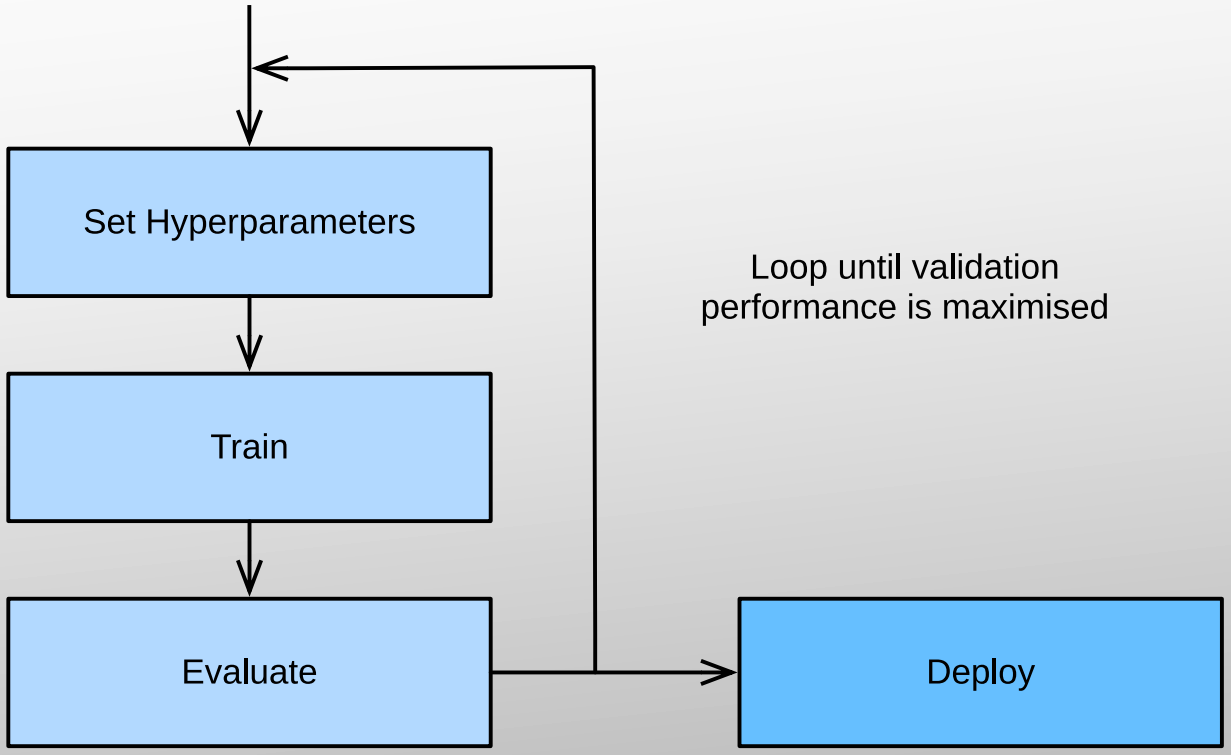
HİPERPARAMETRE NEDİR?

Hiperparametreler, model eğitilmeden önce belirlenen ve modelin nasıl çalışacağını etkileyen ayarlardır. Örneğin:

- **KNN algoritmasında** komşu sayısı (k değeri).
- **SVM algoritmasında** kernel fonksiyonu.

Parametrelerden farkı: Hiperparametreler model eğitimi sırasında öğrenilmez, önceden belirlenir. Örneğin:

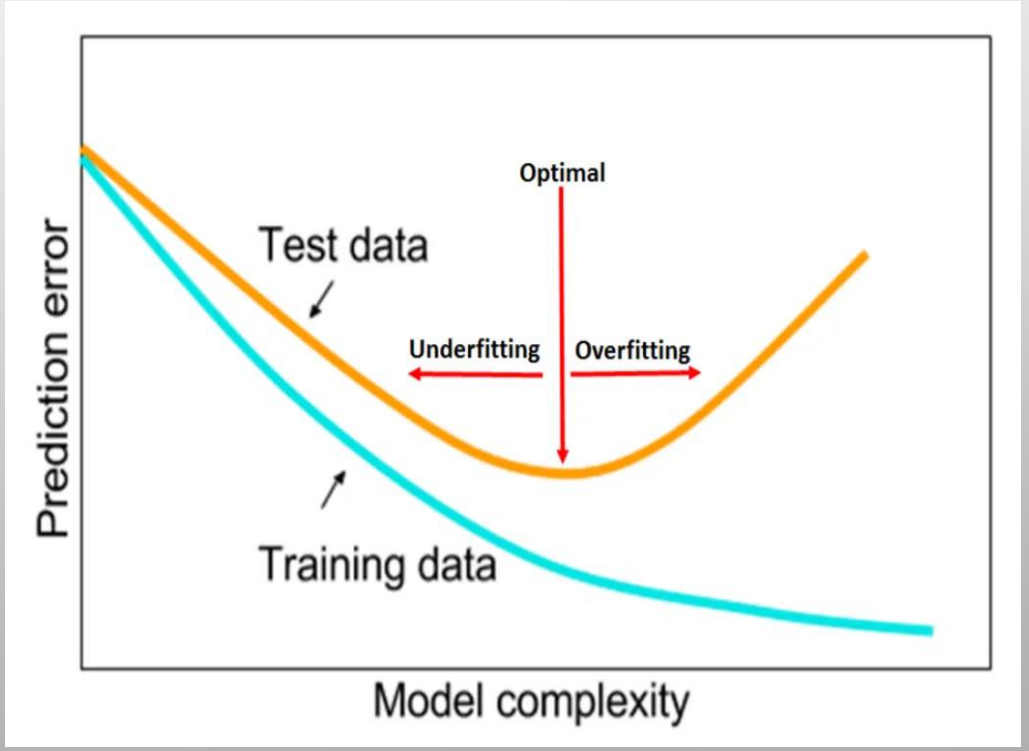
- **KNN:** Komşu sayısı ve mesafe metriği.
- **Logistic Regression:** Öğrenme oranı ve düzenleme katsayısı.



Şekil 19.1.1 Makine öğreniminde, modeli farklı hiperparametrelerle birden çok kez eğitmekten oluşan tipik iş akışı.

HİPERPARAMETRE OPTİMİZASYONU NEDİR?

- **Amaç:** Bir modelin doğrulama kaybını minimize etmek veya doğruluğunu maksimize etmek için en iyi hiperparametre kombinasyonunu bulmaktır.



Avantajları:

- Overfitting ve underfitting dengesini sağlar.
- Model karmaşıklığını kontrol eder.

İyi bir hiperparametre optimizasyonu için başlangıçta sağlam bir temel model oluşturulması gerekir.

HİPERPARAMETRE OPTİMİZASYONU VE OTOMASYON

Dezavantajları:

- Hiperparametre seçim süreci manuel olarak yapıldığında zaman alıcıdır.
- Model doğruluğunu artırmak için farklı kombinasyonların denenmesi günler sürebilir.

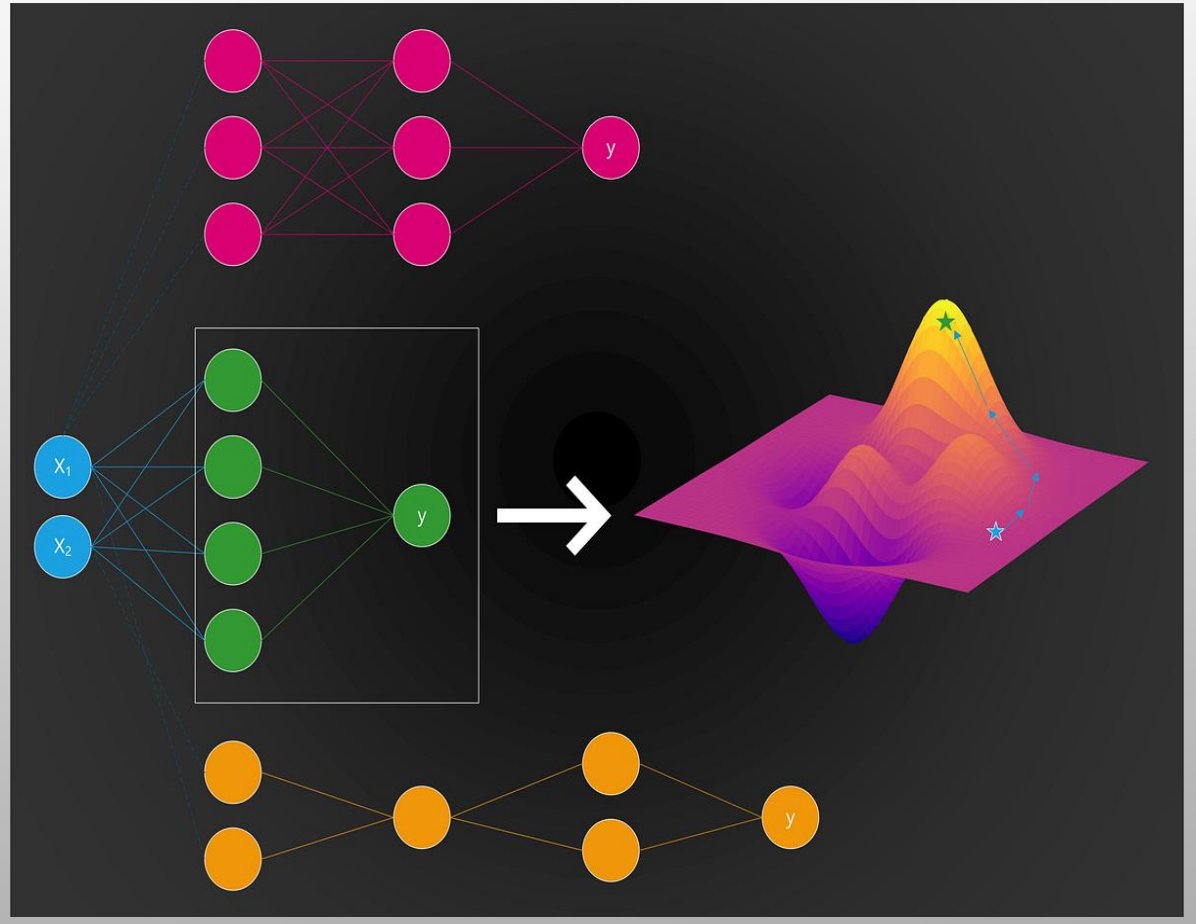
Çözüm:

Hiperparametre Optimizasyonu (HPO):

- Algoritmalar, doğru ayarları otomatik olarak optimize eder.
- Süreci hızlandırır ve doğruluk artırır.

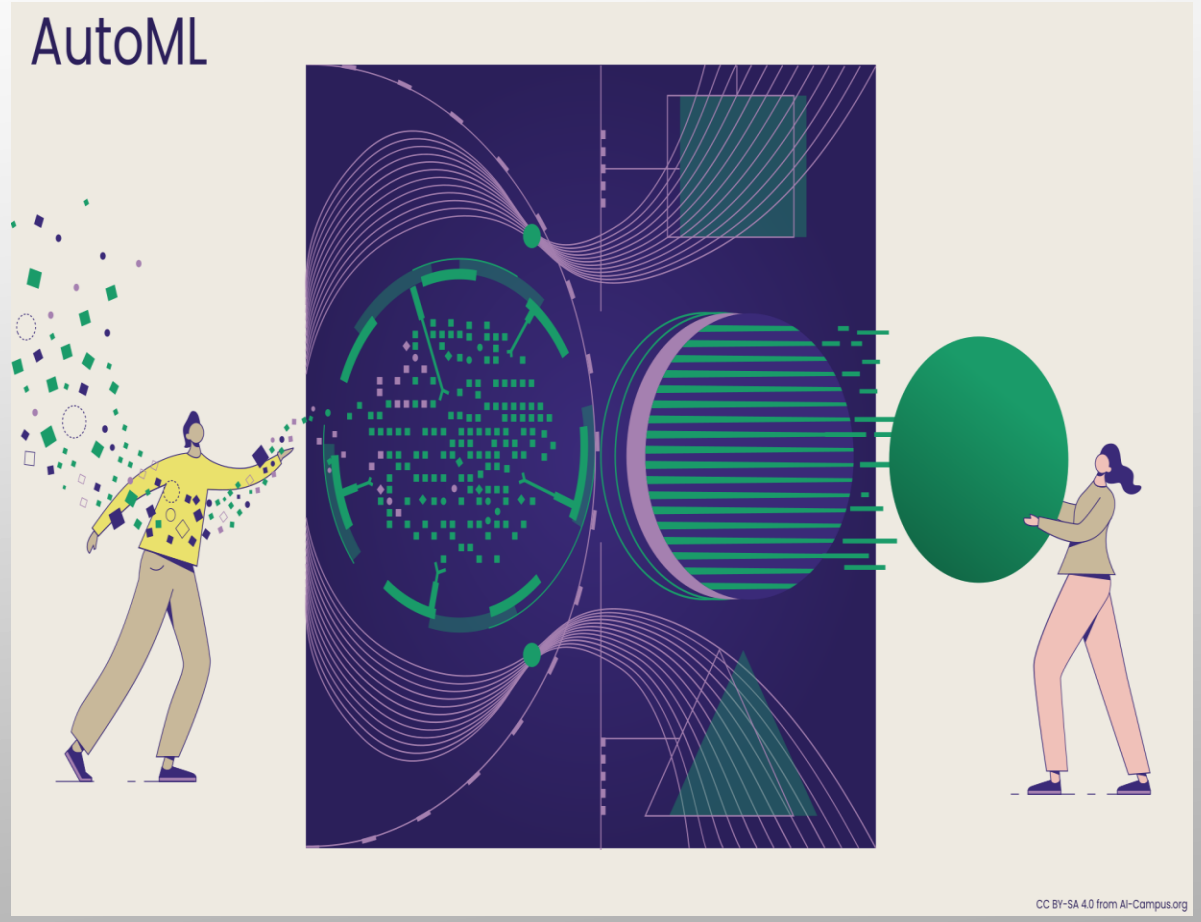
Neural Architecture Search (NAS):

- Yeni model mimarileri tasarlamak için hiperparametre optimizasyonunu kullanır.
- Daha karmaşık ve maliyetli bir yöntemdir.



AutoML:

- HPO ve NAS süreçlerini birleştirerek makine öğrenmesi süreçlerini tamamen otomatikleştirir.



DERİN ÖĞRENME UYGULAMALARINDA EN SIK KULLANILAN HİPER PARAMETRELER

Öğrenme Oranı (Learning Rate):

- Modelin ağırlıklarını ne kadar hızlı güncelleyeceğini belirler.
- Küçük değerler daha yavaş ama daha hassas öğrenmeyi sağlar. Büyük değerler daha hızlı öğrenme sağlasa da dalgalanmalara neden olabilir.

Batch Boyutu (Batch Size):

- Aynı anda işlenecek veri miktarını belirler.
- Büyük batch boyutları daha doğru gradyan hesaplamaları sağlar ancak daha fazla bellek kullanır.

Katman Sayısı ve Nöron Sayısı:

- Daha fazla katman ve nöron, modelin karmaşıklığını artırır ancak overfitting riskini de yükseltir.

Aktivasyon Fonksiyonları:

- Modelin doğrusal olmayan ilişkileri öğrenmesini sağlar. Örneğin: ReLU, Sigmoid, Tanh.
- ReLU genellikle hızlı ve etkili olduğu için sıkça tercih edilir.

Düzenleme Teknikleri (Regularization):

- Overfitting'i önlemek için kullanılır. Örneğin: Dropout, L1/L2 Regularization.

Optimizasyon Algoritmaları:

- Modelin öğrenme sürecini yöneten algoritmalar. Örnekler: SGD, Adam, RMSProp.
- Adam, genellikle varsayılan bir optimizasyon algoritması olarak kullanılır.

Momentum:

- Öğrenme hızını artırır ve SGD'nin yerel minimumlarda takılmasını önler.

Epoch Sayısı:

- Modelin tüm veri setini kaç kez işleyeceğini belirler. Çok fazla epoch overfitting'e neden olabilir.

DERİN ÖĞRENMEDE SIK KULLANILAN HİPERPARAMETRELER

Hiperparametre	Açıklama
Learning Rate	Ağırlıkların ne kadar hızlı güncelleneceğini belirler.
Batch Size	Aynı anda işlenecek veri miktarını belirler.
Activation Function	Modelin doğrusal olmayan ilişkileri öğrenmesini sağlar (ör. ReLU, Tanh).
Optimizer	Öğrenme sürecini yöneten algoritma (ör. SGD, Adam, RMSProp).
Momentum	Salınımları azaltarak optimizasyonu hızlandırır.
Regularization	Overfitting'i önlemek için eklenen teknikler (ör. Dropout, L2).
Epoch	Modelin tüm veri setini kaç kez işleyeceğini belirler.

HİPERPARAMETRE OPTİMİZASYONU: AMAÇ FONKSİYONU

Amaç: Bir modelin doğrulama kaybını minimize etmek.

Zorluklar:

- Eğitim süreci rastlantısaldır, bu da sonuçların gürültülü olmasına neden olur.
- Tüm kombinasyonları denemek maliyetlidir.

Hızlandırma Teknikleri:

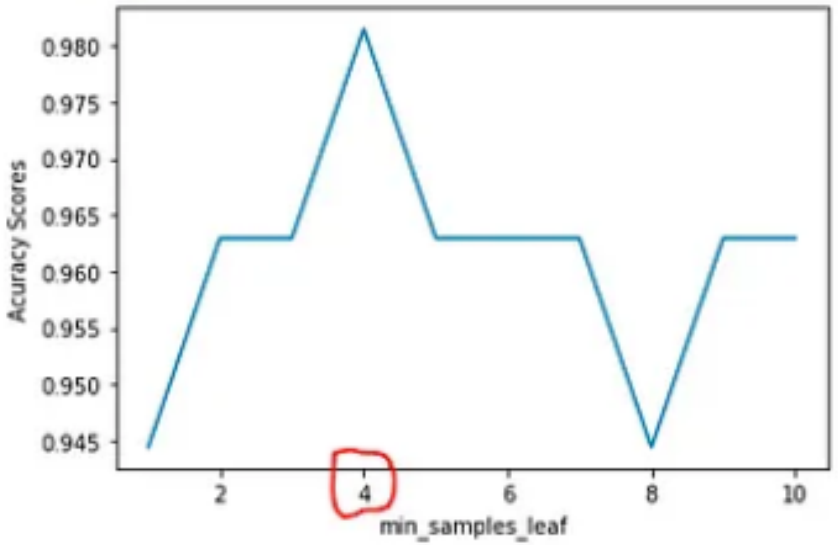
- Paralel arama yöntemleri.
- Daha ucuz tahminler için yaklaşık çözümler.

MANUAL HİPERPARAMETRE OPTİMİZASYONU

- min_samples_leaf=4 için modelin en yüksek accuracy skorunu verdiğini ve bu skorun 0.981 olduğu gözlemlenir. Burada sadece bir hiperparametre üzerinden 10 farklı değeri deneyerek en iyi sonucu veren hiperparametre değerini bulmaya çalıştık. Denenmek istenen çok sayıda hiperparametre ve değeri olduğunda optimizasyonun manual olarak yapılamayacağı açıktır. Bu ihtiyaçtan dolayı GridSearchCV ve RandomizadSearchCV yöntemleri geliştirilmiştir.

```
# RandomForest Manual Tuning
acc_scores = []
for n in range(1,11):
    rf = RandomForestClassifier(min_samples_leaf=n).fit(X_train,y_train)
    y_pred = rf.predict(X_test)
    acc_scores.append(accuracy_score(y_test,y_pred))

print(acc_scores)
import matplotlib.pyplot as plt
plt.plot(range(1,11), acc_scores)
plt.xlabel('min_samples_leaf')
plt.ylabel('Acuracy Scores')
plt.show()
```

```
[0.9444444444444444, 0.9629629629629629, 0.9629629629629629, 0.9814814814814815, 0.9629629629629629, 0.9629629629629629, 0.9629
629629629629, 0.9444444444444444, 0.9629629629629629, 0.9629629629629629]
```



- Burada amaç Random Forest algoritmasının hiperparametrelerinden olan "min_samples_leaf" için farklı değerler deneyerek accuracy skorlarının nasıl değiştiğini gözlemlemek.

HİPERPARAMETRE OPTİMİZASYONU YÖNTEMLERİ

1. GridSearchCV

Çalışma Prensibi:

- Tüm hiperparametre kombinasyonlarını dener ve en iyi performansı sağlayan seti belirler.

Avantajları:

- Küçük veri setlerinde ve az sayıda hiperparametreyle en iyi sonucu garanti eder.

Dezavantajları:

- Büyük veri setlerinde hesaplama maliyeti yüksektir.

```
# GridSearchCV
rf_params = {'bootstrap': [True],
    'max_depth': [80, 90, 100, 110],
    'max_features': [2, 3],
    'min_samples_leaf': range(1,11),
    'min_samples_split': range(1,15,5),
    'n_estimators': [100, 200, 300,500,1000]}
rf = RandomForestClassifier()
rf_gridcv_model = GridSearchCV(estimator=rf, param_grid=rf_params, cv=5, scoring='accuracy', n_jobs=-1, verbose=2).fit(X_train,y_train)
rf_gridcv_model.best_params_
print('rf gridcv model accuracy score = {}'.format(rf_gridcv_model.best_score_))
```

```
Fitting 5 folds for each of 1200 candidates, totalling 6000 fits
rf gridcv model accuracy score = 0.992
```

- İlk olarak denenmesi istenen hiperparametreler ve değerleri bir sözlük yapısında tanımlanır. Daha sonra Sklearn kütüphanesinden import edilen GridSearchCV metodu çağırılıp gerekli parametreleri belirtilir.

- Kodun çıktısında; 6 tane hiperparametre için 1200 tane farklı kombinasyon olduğunu, her bir kombinasyona 5-katlı cross-validation uygulandığında toplamda 6000 tane model fit etme işleminin gerçekleştiği ve bu işlemlerin 57.6 dakika sürdüğü görülmektedir.
- Optimizasyon sonucunda belirlenen en iyi performansı gösteren hiperparametre değerleri kullanıldığında accuracy skoru 0.992 oluyor.
- Sonuç iyi olsa da, küçük bir veri seti olmasına rağmen tüm bu işlemlerin yapılması neredeyse 1 saat sürdü. Zaman açısından değerlendirildiğinde çok maliyetli olmaktadır.

2. RandomizedSearchCV

Çalışma Prensibi:

- Rastgele seçilen hiperparametre kombinasyonlarını belirli bir süre veya iterasyon boyunca test eder.

Avantajları:

- Daha düşük maliyetle geniş bir hiperparametre alanını tarar.
- Pratikte daha sık tercih edilir.

Dezavantajları:

- En iyi sonucu garanti edemez.

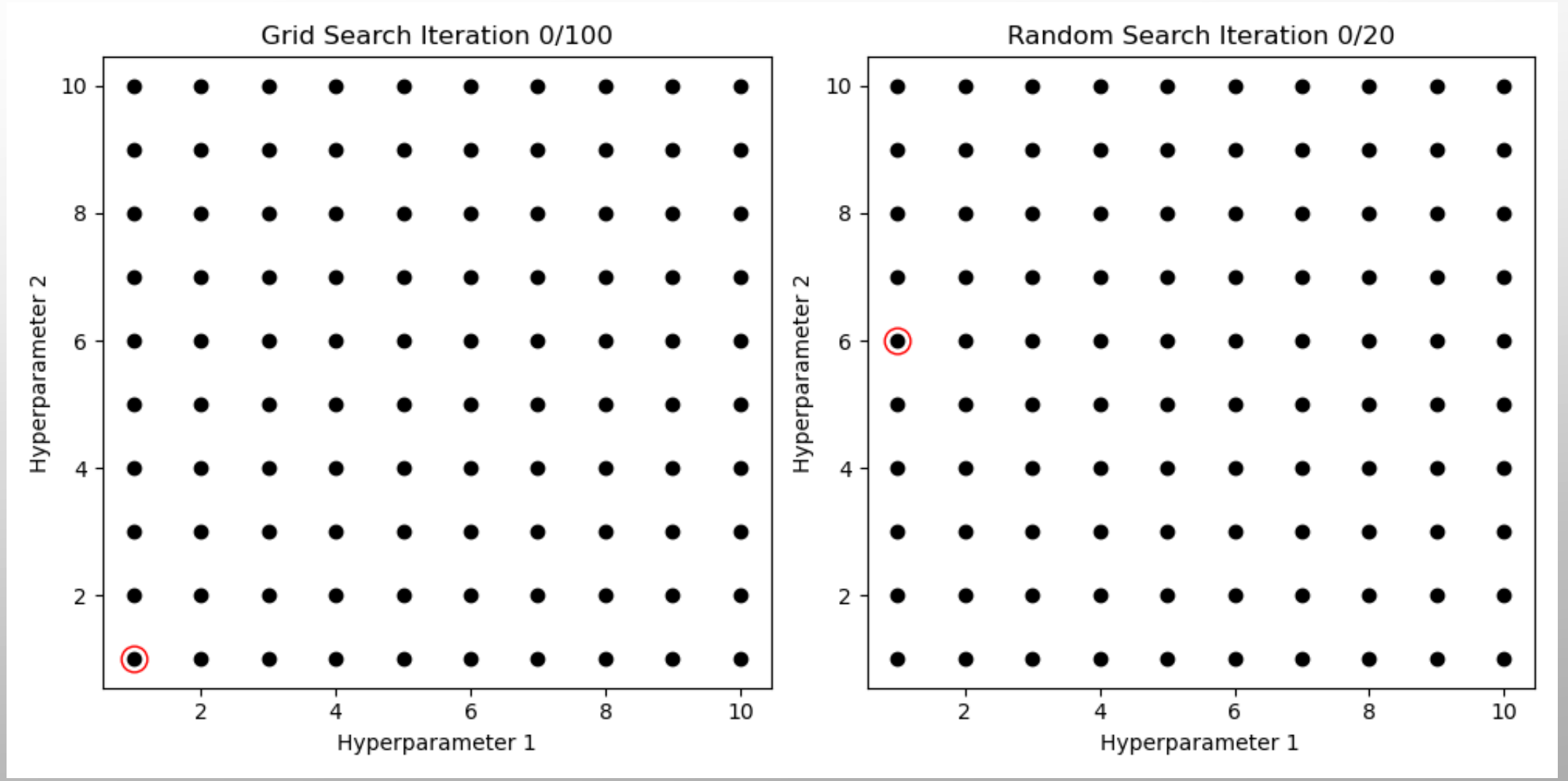
```python
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV

# Rastgele Arama için Hiperparametreler
rf_params = {
    'bootstrap': [True],  # Modelin örneklemeyi bootstrap ile yapıp yapmayacağı
    'max_depth': [80, 90, 100, 110],  # Ağacın maksimum derinlik değerleri
    'max_features': [2, 3],  # Her ayrımda değerlendirilecek maksimum özellik sayısı
    'min_samples_leaf': range(1, 11),  # Bir yaprak düğümdeki minimum örnek sayısı
    'min_samples_split': range(1, 15, 5),  # Bir düğümü bölmek için gereken minimum örnek sayısı
    'n_estimators': [100, 200, 300, 500, 1000]  # Ormandaki ağaç sayısı
}

# Random Forest sınıflandırıcıyı oluştur
rf = RandomForestClassifier()

# RandomizedSearchCV tanımlaması
rf_randomcv_model = RandomizedSearchCV(
    estimator=rf,  # Değerlendirilecek model
    param_distributions=rf_params,  # Parametre kombinasyonları
    n_iter=200,  # Rastgele denenecek parametre kombinasyonu sayısı
    cv=5,  # 5 katlı çapraz doğrulama
    scoring='accuracy',  # Değerlendirme metriği olarak doğruluk
    n_jobs=-1,  # Tüm işlemcileri kullan
    verbose=2  # Çıktı detay seviyesi
).fit(X_train, y_train)  # Modeli eğit ve en iyi parametreleri bul

# En iyi parametreleri yazdır
rf_randomcv_model.best_params_

# Modelin en iyi doğruluk skorunu yazdır
print('rf randomcv model accuracy score = {}'.format(rf_randomcv_model.best_score_))
```

```
Fitting 5 folds for each of 200 candidates, totalling 1000 fits
rf randomcv model accuracy score = 0.992
```

- Burada GridSeaarchCV’ye benzer şekilde, denenmesi istenen hiperparametreler ve değerleri bir sözlük yapısında tanımlanır. Daha sonra Sklearn kütüphanesinden import edilen RandomizedSearchCV metodu çağırılıp gerekli parametreleri belirtilir.
- Bu yöntemle 1200 tane kombinasyonu ayrı ayrı denemek yerine n_iter= 200 belirleyerek 200 farklı kombinasyonu denenmaktadir.

- Kod çıktısı incelendiğinde 200 farklı kombinasyon için 5-katlı cross-validation uygulandığında toplamda 1000 tane model fit etme işleminin gerçekleştiğini ve bu işlemlerin 6.5 dakika sürdüğünü görebiliyoruz.
- Burada da accuracy skorumuzu 0.992 olarak gözlemledik. Yani daha kısa sürede GridSearchCV ile elde ettiğimiz skoru elde ettik. Bu her zaman eşit çıkmasa da en iyi skora yakın değerler elde etmeyi bekleriz. Dolayısıyla zaman açısından düşünüldüğünde büyük veri setlerinde RandomizedCV tercih edilebilir.


Manual Search
Second Parameter
First Parameter
Grid Search
Second Parameter
First Parameter
Random Search
Second Parameter
First Parameter


Grid Search Iteration 0/100
Hyperparameter 2
Hyperparameter 1
10
8
6
4
2
2
4
6
8
10
Random Search Iteration 0/20
Hyperparameter 2
Hyperparameter 1
10
8
6
4
2
2
4
6
8
10

GRİDSEARCHCV VE RANDOMİZEDSEARCHCV KARŞILAŞTIRMASI

Özellik	GridSearchCV	RandomizedSearchCV
Kombinasyon Sayısı	Tüm kombinasyonlar	Rastgele seçilen kombinasyonlar
Hesaplama Maliyeti	Yüksek	Daha düşük
Kullanım Alanı	Küçük veri setleri	Büyük veri setleri

KONFİGÜRASYON UZAYI (CONFİGURATİON SPACE)

Optimize edilecek hiperparametrelerin türlerini, değer aralıklarını ve dağılımlarını tanımlayan uzaydır.

Örnek:

Hiperparametre	Tür	Aralık	Logaritmik?
Learning Rate	Float	$[10^{-6}, 10^{-1}]$	Evet
Batch Size	Integer	[8, 256]	Evet
Activation Function	Kategorik	{relu, tanh}	Hayır

Bağımlılıklar: Örneğin, katman sayısı arttıkça her katmandaki nöron sayısı gibi hiperparametreler belirlenir.

Zorluk: Çok geniş bir arama uzayı, hesaplama maliyetini artırabilir.

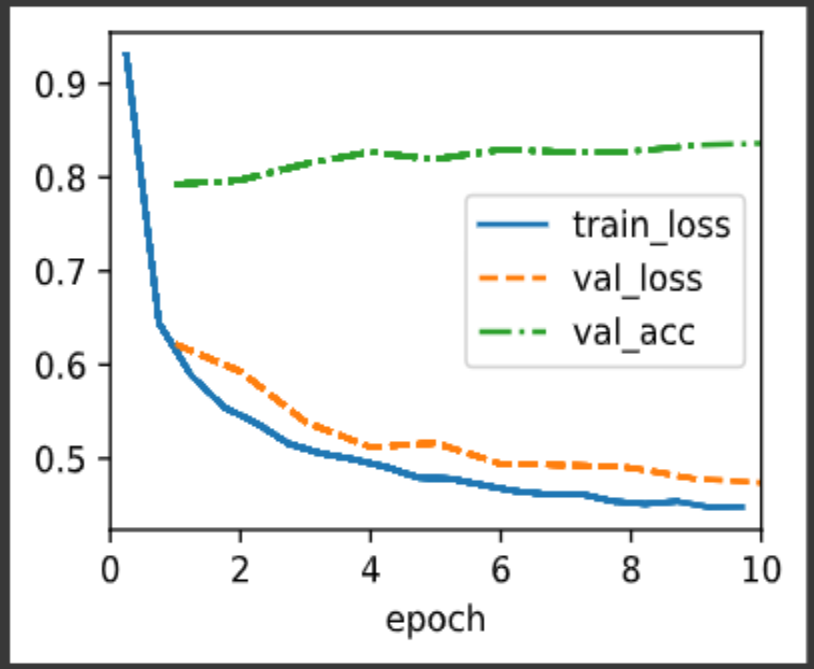
```
!pip install d2l==1.0.3
```

3 sn.

```
import torch
from torch import nn
from torch.nn import functional as F
from d2l import torch as d2l
```

```
class SoftmaxRegression(d2l.Classifier):
    """The softmax regression model."""
    def __init__(self, num_outputs, lr):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(nn.Flatten(),
                                 nn.LazyLinear(num_outputs))

    def forward(self, X):
        return self.net(X)
```

```
[5] data = d2l.FashionMNIST(batch_size=256)
    model = SoftmaxRegression(num_outputs=10, lr=0.1)
    trainer = d2l.Trainer(max_epochs=10)
    trainer.fit(model, data)
```




```
@d2l.add_to_class(d2l.Classifier)
def loss(self, Y_hat, Y, averaged=True):
    Y_hat = Y_hat.reshape((-1, Y_hat.shape[-1]))
    Y = Y.reshape((-1,))
    return F.cross_entropy(
        Y_hat, Y, reduction='mean' if averaged else 'none')
```

HİPERPARAMETRE OPTİMİZASYONU API

Hiperparametre optimizasyonu ve api konusuna başlarken arama ve planlamla ilkesini daima kullanacağımızı unutmamalıyız.

İlk olarak hiperparamtre optimizasyonunda yeni hiperparametrelerin konfigürasyonlarını örnekleyeceğiz bu yeni konfigürasyonlar seçmek anlamına gelir ve arama işlemi ile yapılır.

İkinci olarak ilk adımda yaptığımız her bir konfigürasyonun değerlendirilmesi için hangi kaynakların ne kadar süre tahsis edileceğine karar vermesi gerekir.

SEARCH

- sample_configuration fonksiyonu ile yeni bir aday konfigürasyon ekler.
- Bu fonksiyonu basit bir şekilde gerçekleştirmek konfigürasyonları rastgele ve eşit olarak örneklemek için kullanılır.
- Önceki denemelerin performansına dayalı olarak kararları alır.
- Bu sayede algoritmalar zamanla daha umut verici hiperparametre kombinasyonlarını seçebilir.

```
class HPOSearcher(d2l.HyperParameters):
    def sample_configuration() -> dict:
        raise NotImplementedError
    def update(self, config: dict, error: float, additional_info=None):
        pass
```

SEARCH

- RandomSearcher sınıfı, HPOSearcher sınıfından türetilir ve burada ilk konfigürasyon initial_config ile belirlenebilir,sonrakiler ise rastgele seçilir.
- initial_config kullanıcının başlangıç konfigürasyonunu elle belirlemesine olanak sağlar.

```
class RandomSearcher(HPOSearcher):
    def __init__(self, config_space: dict, initial_config=None):
        self.save_hyperparameters() #Bu fonksiyon, konfigürasyon bilgilerini kaydeder, yani sınıfın parametrelerini saklar.
    def sample_configuration(self) -> dict:# Başlangıçta konfigürasyon verildiyse sağlanır
        if self.initial_config is not None:
            result = self.initial_config
            self.initial_config = None #İlk konfigürasyon kullanıldıktan sonra, initial_config değeri None yapılır ki bir daha kullanılmasın.
        else:
            result = {
                name: domain.rvs() #o hiperparametrenin tanımlanan dağılımına göre rastgele bir değer döndürür.
                for name, domain in self.config_space.items()
            }
        return result
```

SCHEDULER

- HPOScheduler, her deneme için ne zaman ve ne kadar süreyle çalıştırılacağını belirler.
- **HPOScheduler**, hiperparametre optimizasyon sürecinde önemli bir yöneticidir. HPOScheduler, denemeleri zamanlamak ve eğitim süresi gibi parametreleri ayarlamak için **HPOSearcher** ile etkileşime girer.

```
class HPOScheduler(d2l.HyperParameters):  #@save
    def suggest(self) -> dict:
        raise NotImplementedError
    def update(self, config: dict, error: float, info=None):
        raise NotImplementedError
```

SCHEDULER

- HPOScheduler, her deneme için ne zaman ve ne kadar süreyle çalıştırılacağını belirler.
- **HPOScheduler**, hiperparametre optimizasyon sürecinde önemli bir yöneticidir. HPOScheduler, denemeleri zamanlamak ve eğitim süresi gibi parametreleri ayarlamak için **HPOSearcher** ile etkileşime girer.

```
class HPOScheduler(d2l.HyperParameters):  #@save
    def suggest(self) -> dict:
        raise NotImplementedError
    def update(self, config: dict, error: float, info=None):
        raise NotImplementedError
```

SCHEDULER

```python
class BasicScheduler(HPOScheduler):  #@save
    def __init__(self, searcher: HPOSearcher):
        self.save_hyperparameters()
    #Eğitim için kaynaklar mevcut olduğunda çağrılır.
    #Bu metod, yeni konfigürasyonları önerir ve aynı zamanda eğitim süresi gibi parametrelerin nasıl ayarlanacağına karar verir
    def suggest(self) -> dict:
        return self.searcher.sample_configuration()
    #Bir deneme tamamlandıktan sonra çağrılır.
    #Bu metod, yapılan denemenin sonucunu (hata oranı, doğruluk, vs.) alır ve HPOSearcher'a geri bildirim verir.
    def update(self, config: dict, error: float, info=None):
        self.searcher.update(config, error, additional_info=info)
```

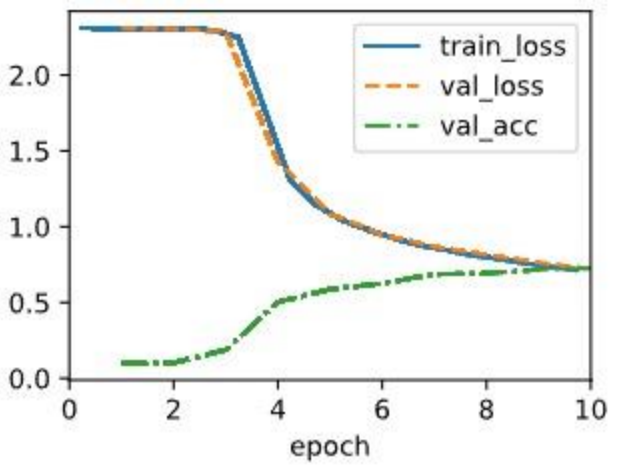
HPOScheduler'ın amacı, **kaynakları en verimli şekilde kullanmak** ve optimizasyon sürecini yönetmektir.

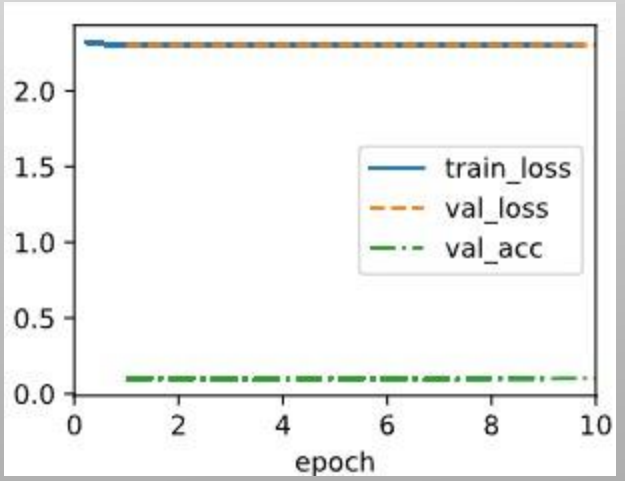
TUNER

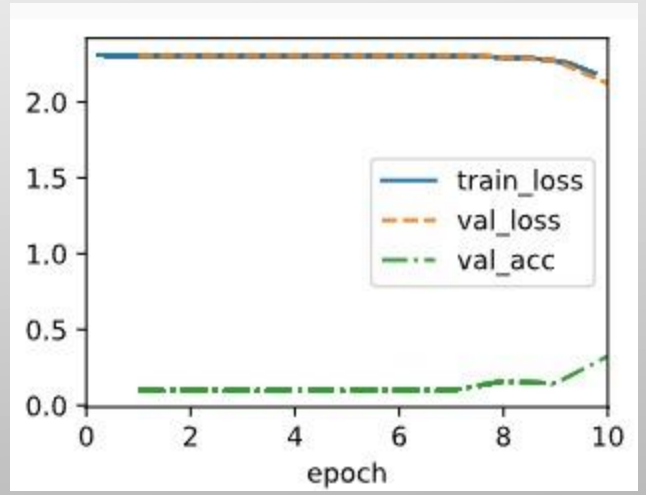
Tuner hiperparametre optimizasyonu sürecinde Scheduler(zamanlayıcı) ve Search(Arayıcı) bileşenlerini çalıştıran, ayrıca sonuçları takip eden bir bileşendir. Farklı hiperparametre konfigürasyonlarını test etmek için denemeleri (trials) sırasıyla gerçekleştirir.

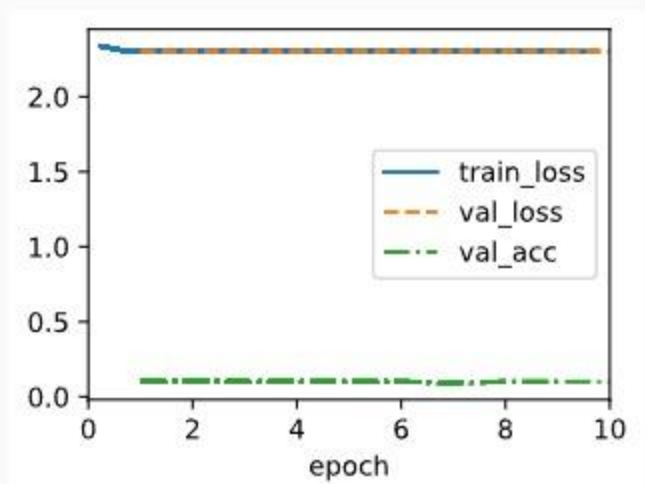
```python
class HPOTuner(d2l.HyperParameters):  #@save
    def __init__(self, scheduler: HPOScheduler, objective: callable):
        self.save_hyperparameters()  # parametreleri kaydet
        # Sonuçları takip etmek için
        self.incumbent = None  # En iyi konfigürasyon başlangıçta yok
        self.incumbent_error = None  # En iyi konfigürasyonun hata değeri başlangıçta yok
        self.incumbent_trajectory = []  # En iyi konfigürasyonların evrimini tutacak liste
        self.cumulative_runtime = []  # Toplam çalışma süresi
        self.current_runtime = 0  # Şu anki denemenin süresi
        self.records = []  # Tüm denemelerin kayıtları

    def run(self, number_of_trials):
        for i in range(number_of_trials):
            start_time = time.time()  # Denemenin başlangıç zamanını kaydet
            config = self.scheduler.suggest()  # Yeni konfigürasyon önerilir
            print(f"Trial {i}: config = {config}")  # Şu anki denemenin konfigürasyonu yazdırılır
            error = self.objective(**config)  # Bu konfigürasyonla eğitim yapılır ve hata değeri hesaplanır
            error = float(error.cpu().detach().numpy())  # Hata değeri, tensor'dan float'a dönüştürülür
            self.scheduler.update(config, error)  # Sonuç, scheduler ile güncellenir
            runtime = time.time() - start_time  # Denemenin süresi hesaplanır
            self.bookkeeping(config, error, runtime)  # Sonuçlar ve süre kaydedilir
            print(f"    error = {error}, runtime = {runtime}")  # Hata ve süre yazdırılır
```

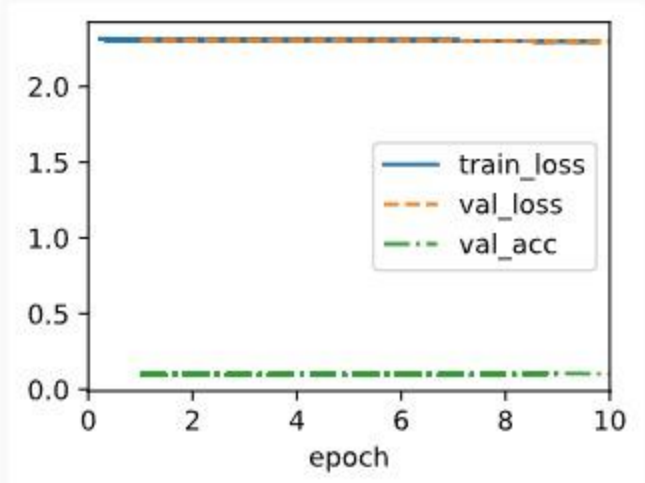
```
searcher = RandomSearcher(config_space, initial_config=initial_config)
scheduler = BasicScheduler(searcher=searcher)
tuner = HPOTuner(scheduler=scheduler, objective=hpo_objective_lenet)
tuner.run(number_of_trials=5)
```











SYNE TUNE

Üzerinde durduğumuz bu kod örneği temel bir HPO işlemidir. Karmaşık ve paralel çalışması gereken projelerde Syne tune gibi araçlar daha sık kullanılır. Syne tune optimizasyon süreçlerinin etkinliğini arttırabilir, dağıtık çalışma sayesinde , tek bir makine sınırlı sayıda denemeyi test etmek yerine, yüzlerce veya binlerce denemeyi paralel olarak yürütebilirsiniz. Bu, daha kısa sürede çok daha fazla konfigürasyonu test etmenizi sağlar.

SYNE TUNE

- Syne Tune her denememenin ne kadar zaman aldığına ve kullanılan kaynaklara dair izleme yapar. Bu sayede daha verimli bir kaynak kullanımı sağlanabilir ve kaynaklar daha akıllıca yönetilebilir.
- Syne Tune, sadece rastgele arama (random search) gibi basit yöntemleri değil, aynı zamanda Bayesian Optimization, Hyperband ve Asynchronous Successive Halving gibi daha gelişmiş optimizasyon algoritmalarını da destekler. Bu algoritmalar, daha akıllıca kararlar vererek daha hızlı ve daha verimli bir şekilde optimum sonuçlara ulaşılmasını sağlar.

HPO ALGORİTMALARININ PERFORMANSININ KAYDI (BOOKEEPİNG THE PERFORMANCE OF HPO ALGORİTHMS)

Bu kısım, Hiperparametre Optimizasyon (HPO) algoritmalarının performansını değerlendirmek için önemli bir metodu açıklamaktadır. HPO algoritmalarında, her zaman en iyi performans gösteren yapılandırma (bu yapılandırma "incumbent" olarak adlandırılır) ve bu yapılandırmanın doğrulama hatası en önemli faktörlerdir. Ancak bu sadece bir yönüdür. HPO algoritmalarının etkinliğini daha derinlemesine incelemek için, algoritmanın ne kadar sürede bu "incumbent" yapılandırmayı bulduğuna da bakmak gerekir.

HPO ALGORİTMALARININ PERFORMANSININ KAYDI (BOOKEEPİNG THE PERFORMANCE OF HPO ALGORİTHMS)

Buradaki odak, her iterasyonda geçen toplam süredir. Bu süre, iki bileşenden oluşur:

- **Değerlendirme Süresi (Objective Call):** Bu, her parametre kombinasyonunun performansını test etmek için geçirilen zamandır. Yani algoritma, seçtiği parametre seti ile modelin doğruluğunu veya kaybını hesaplar.
- **Karar Verme Süresi (Scheduler.suggest Call):** Bu, algoritmanın bir sonraki hiperparametre setini seçmek için geçirdiği zamandır. Yani algoritma, arama alanında bir sonraki parametre kombinasyonunu seçmek için harcadığı süredir.

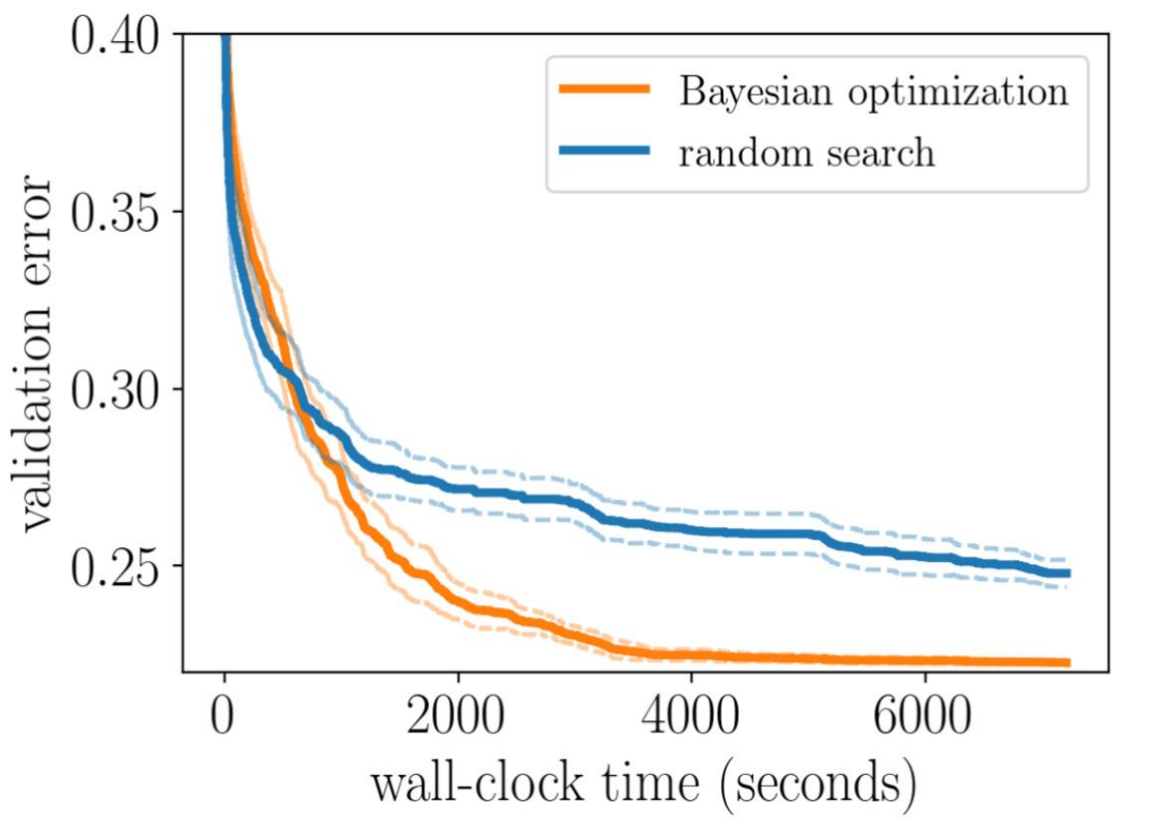
HPO ALGORİTMALARININ PERFORMANSININ KAYDI (BOOKEEPİNG THE PERFORMANCE OF HPO ALGORİTHMS)

HPO algoritmalarını karşılaştırırken, farklı algoritmaların performansını en iyi şekilde değerlendirmek önemlidir. Her HPO çalışması, iki ana rastgelelik kaynağına dayanır:

- Eğitim sürecinin rastgele etkileri (örneğin, rastgele ağırlık başlatma veya mini-batch sıralaması)
- HPO algoritmasının içsel rastgeleliği (örneğin, rastgele arama ile rastgele örnekleme)

Bu nedenle farklı algoritma karşılaştırmaları için her deneme birkaç kez yapılmalı ve raporlanmalıdır.

- Random Search ve Bayesian Optimizasyonu, bir ileri beslemeli sinir ağının hiperparametrelerini ayarlamak için karşılaştırılmıştır.
  - Her iki algoritma, farklı rastgele tohumlarla birkaç kez test edilmiştir.
  - Katı çizgi: Ortalama performans
  - Kesikli çizgi: Standart sapma



Sonuç: Random Search ve Bayesian Optimization, yaklaşık 1000 saniyeye kadar benzer performans sergiler. Ancak, Bayesian Optimization geçmiş gözlemleri kullanarak daha iyi konfigürasyonlar bulur ve sonrasında daha hızlı bir şekilde performansını artırır.

EŞZAMANSIZ RASTGELE ARAMA (ASYNCHRONOUS RANDOM SEARCH)

Eşzamansız Rastgele Arama (Asynchronous Random Search, ARS), derin öğrenme modellerindeki hiperparametre optimizasyonunu hızlandırmak için kullanılan bir meta-algoritmadır. Geleneksel yöntemlerin aksine, ARS, birden fazla işçiyi paralel olarak çalıştırarak farklı hiperparametre kombinasyonlarını aynı anda değerlendirir. Bu sayede, modelin performansını en üst düzeye çıkarmak için gereken süreyi önemli ölçüde azaltır.

NASIL ÇALIŞIR?

- **Hiperparametre Uzayı Tanımlama:** Optimize edilecek hiperparametrelerin (örneğin, öğrenme oranı, katman sayısı, batch size) aralıkları belirlenir.
- **İşçilerin Başlatılması:** Birden fazla işçi (thread veya process) başlatılır. Her işçi, hiperparametre uzayından rastgele bir nokta seçer ve bu hiperparametreler ile modeli eğitir.
- **Performans Değerlendirmesi:** Eğitim tamamlandıktan sonra, modelin bir doğrulama seti üzerindeki performansı ölçülür (örneğin, doğruluk, kayıp değeri).

- **En İyi Modelin Seçimi:** Tüm işçilerin bulduğu modeller karşılaştırılır ve en iyi performansa sahip olan model seçilir.
- **İterasyon:** Bu adımlar, belirlenen bir iterasyon sayısı veya durdurma kriteri (örneğin, performans iyileşmesinin durması) gerçekleşene kadar tekrarlanır.

ARS'NİN AVANTAJLARI

Hız: Paralel hesaplama sayesinde optimizasyon süresini önemli ölçüde azaltır.

Esneklik: Farklı hiperparametre uzayları için kolayca uygulanabilir.

Basitlik: Karmaşık matematiksel hesaplamalar gerektirmez.

Ölçeklenebilirlik: Daha fazla işçi ekleyerek kolayca ölçeklendirilebilir.

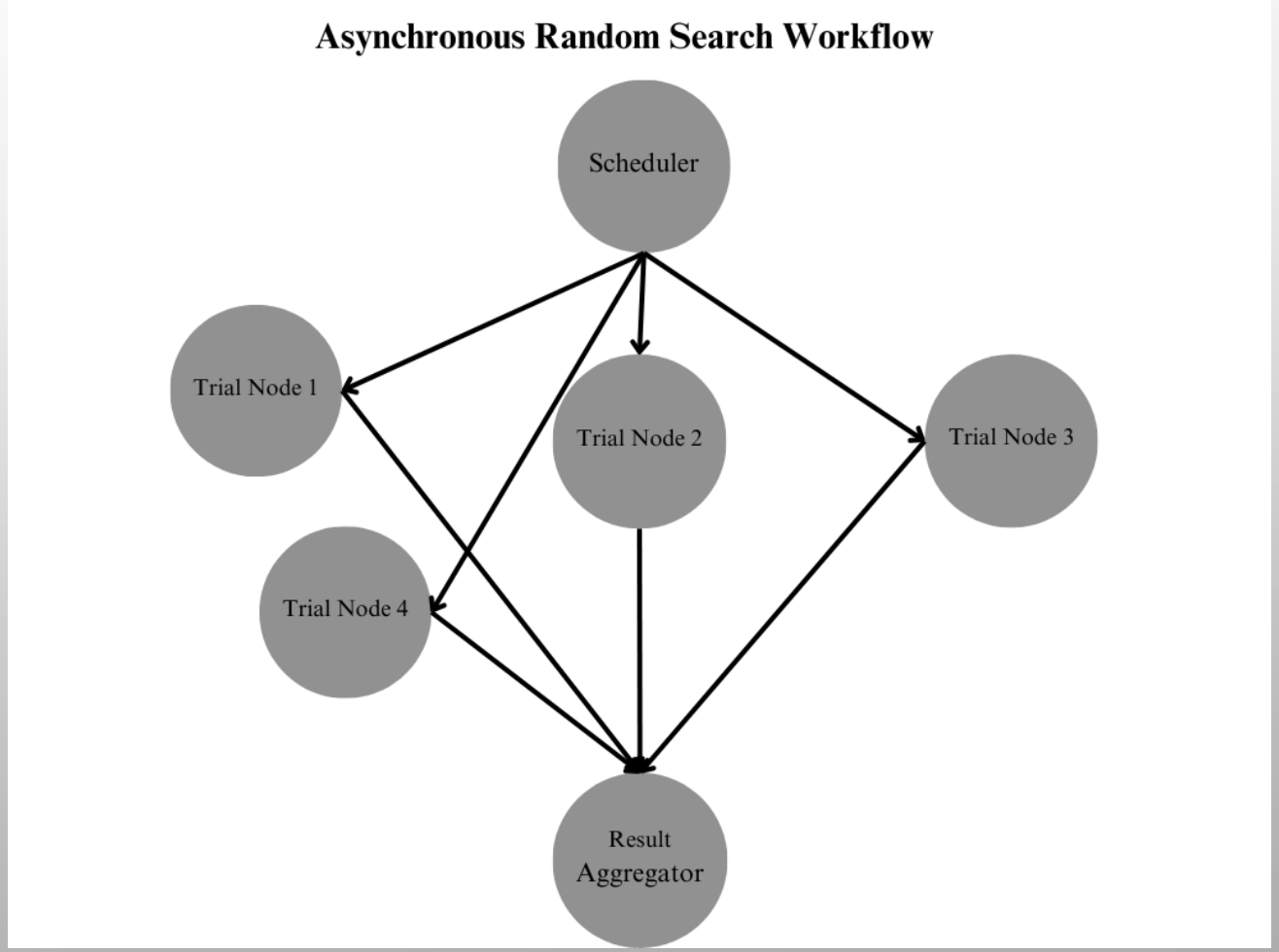
ARS'NİN DEZAVANTAJLARI

Rastgelelik: Çözümün kalitesi, rastgele seçilen hiperparametrelere bağlı olarak değişebilir.

Verimsizlik: Özellikle büyük hiperparametre uzaylarında, gereksiz yere birçok nokta değerlendirilebilir.

Yerel Minimumlar: Optimal olmayan yerel minimumlara sıkışma riski vardır.

Scheduler: Hiperparametre konfigürasyonlarını seçip dağıtır.
Trial Nodes: Bağımsız olarak denemeleri çalıştırır ve sonuçları toplar.
Result Aggregator: Tüm denemelerin sonuçlarını merkezi olarak toplar ve değerlendirir.



KULLANIM ALANLARI

Kontrol Sistemleri

Robotik ve otonom araçlarda hareket planlama ve kontrol stratejilerini optimize etmek için kullanılır.

Endüstriyel Uygulamalar

Üretim planlama ve tedarik zinciri süreçlerinde en iyi stratejileri belirlemek için tercih edilir.

Bilimsel Araştırmalar

Yüksek boyutlu simülasyonlar ve genetik algoritmaların performansını artırmak için uygundur.

Finans ve Ekonomi

Portföy yönetimi ve risk analizinde optimum çözümler sunar, stratejik kararları hızlandırır.

Oyun ve Yapay Zeka

Oyun stratejileri geliştirme ve reinforcement learning süreçlerini destekler.

Büyük Veri ve Bulut Bilişim

Kaynak yönetimi ve büyük veri analitiği için maliyet ve zaman tasarrufu sağlar.

ÇOKLU SADAKAT HİPERPARAMETRE OPTİMİZASYONU (MULTİ-FİDELİTY HYPERPARAMETER OPTİMİZATİON)

Çoklu Sadakat Hiperparametre Optimizasyonu, bir makine öğrenimi modelinin hiperparametrelerini optimize etmek için farklı çözünürlükte (sadakat seviyelerinde) değerlendirmeler kullanmayı amaçlayan bir tekniktir. Geleneksel hiperparametre optimizasyon yöntemleri tüm değerlendirmeleri tam çözünürlükte yaparken, bu yöntem farklı sadakat seviyelerinde daha az maliyetle değerlendirme yaparak süreci hızlandırmayı hedefler.

Sadakat (Fidelity):

Sadakat, bir modelin, simülasyonun veya hesaplama yönteminin gerçek bir sistemi veya süreci ne kadar doğru ve detaylı bir şekilde temsil ettiğini ifade eden bir ölçüttür. Sadakat, genellikle modelin karmaşıklık seviyesi, doğruluk oranı ve hesaplama maliyetine bağlı olarak yüksek ve düşük sadakat olarak sınıflandırılır.

- **Yüksek Sadakat:** Gerçek sistemi ayrıntılı bir şekilde temsil eden, daha fazla hesaplama kaynağı ve zaman gerektiren modeller veya yöntemler.
- **Düşük Sadakat:** Gerçek sistemi yaklaşık olarak temsil eden, daha hızlı ve düşük maliyetli çözümler sunan modeller veya yöntemler.

NEDEN KULLANILIR?

Maliyet ve Zaman Tasarrufu: Yüksek sadakat seviyelerinde tüm kombinasyonları denemek çok maliyetli olabilir.

Verimli Arama: Daha hızlı iterasyonlarla optimize edilmiş hiperparametre setine ulaşılır.

Gerçek Hayatta Kullanım Alanları:

- Büyük veri kümelerinde veya karmaşık modellerde hiperparametre aramaları.
- Derin öğrenme uygulamaları.
- AutoML sistemleri.

ÇALIŞMA MANTIĞI

Sadakat Seviyelerinin Belirlenmesi:

Örneğin, veri alt kümeleri veya epoch sayılarıyla farklı sadakat seviyeleri oluşturulur.

Başlangıç Değerlendirmeleri:

Düşük sadakat seviyelerinde geniş bir hiperparametre aralığı değerlendirilir.

Sadakat Seviyesi Artırımı:

Potansiyel olarak iyi performans gösteren hiperparametre kombinasyonları, daha yüksek sadakat seviyelerinde yeniden değerlendirilir.

Final Değerlendirmesi:

En iyi adaylar, tam çözünürlükte test edilir.

ARDIŞIK YARILANMA (SUCCESSİVE HALVİNG)

Ardışık Yarılanma (Successive Halving), makine öğrenimi modellerinde hiperparametre optimizasyonu için kullanılan bir yöntemdir. Temel amacı, sınırlı kaynakları (örneğin, zaman, işlem gücü) daha verimli bir şekilde kullanarak en iyi hiperparametre kombinasyonunu belirlemektir. Bu yöntem, düşük performans gösteren hiperparametre ayarlarını erken bir aşamada eleyerek, kalan kaynakları daha umut verici ayarlara yoğunlaştırır.

AVANTAJLARI

Verimli Kaynak Kullanımı: Düşük performans gösteren kombinasyonları erken eleyerek işlem gücünden tasarruf sağlar.

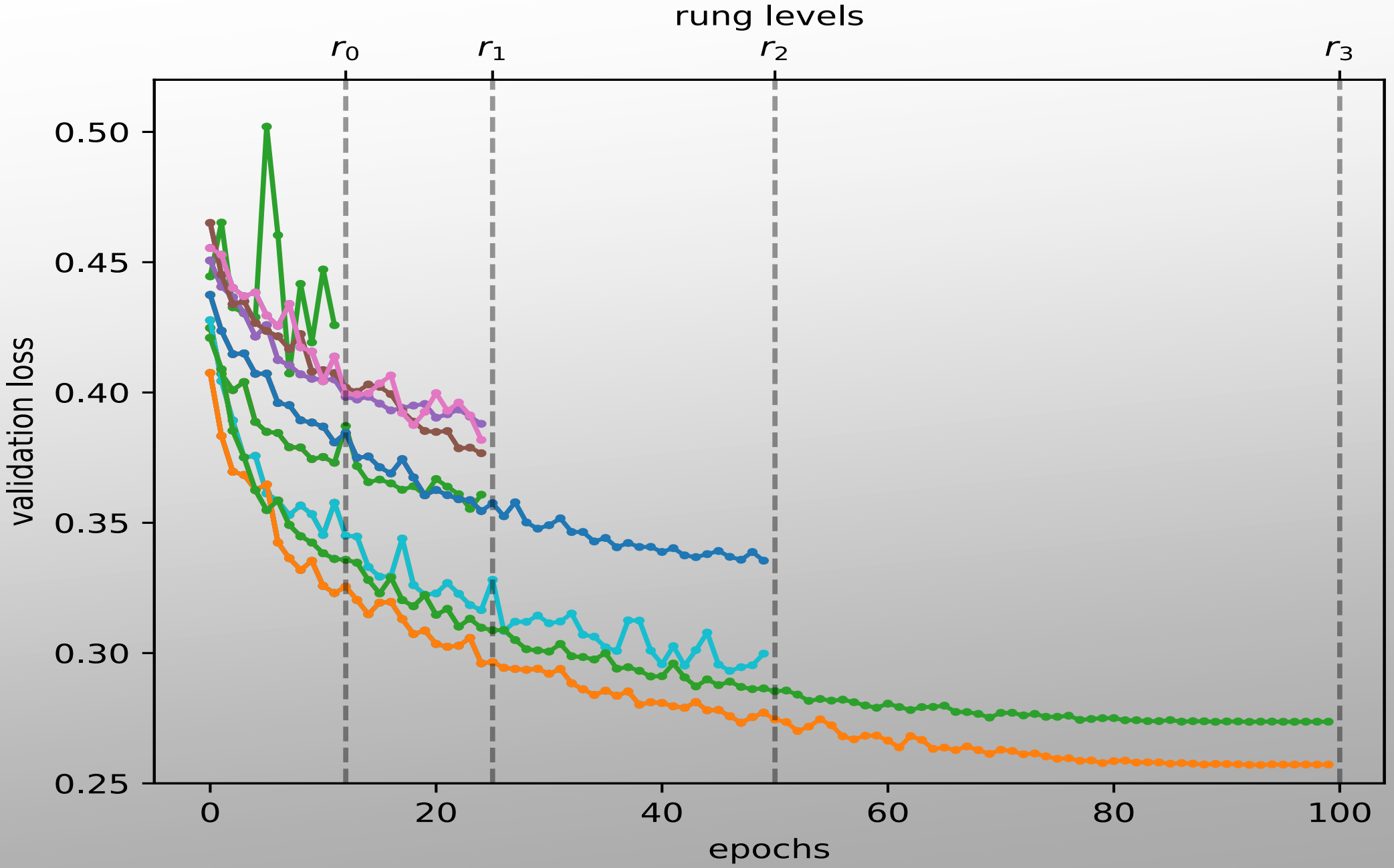
Daha Hızlı Optimizasyon: Daha az maliyetle iyi sonuçlar elde edilebilir.

Uyarlanabilir: Epoch sayısı, veri miktarı veya model kompleksliği gibi farklı ölçütlere göre uygulanabilir.

DEZAVANTAJLARI

Erken Eleme Riskleri: Bazı kombinasyonlar düşük sadakat seviyelerinde iyi performans göstermeyebilir ancak daha fazla kaynakla daha iyi sonuç verebilir.

Parametre Hassasiyeti: Başlangıç kombinasyon sayısı ve elemelerin oranı sonuçları etkileyebilir.


rung levels
r_0
r_1
r_2
r_3
validation loss
0.50
0.45
0.40
0.35
0.30
0.25
0
20
40
60
80
100
epochs

```python
from collections import defaultdict  # 'defaultdict' modülü, bir sözlük türü sağlar.
import numpy as np
from scipy import stats
from d2l import torch as d2l  # 'd2l' modülü, derin öğrenme uygulamaları için araçlar içerir. PyTorch ile entegrasyonu sağlar.

d2l.set_figsize()  # Grafiklerin boyutunu varsayılan değeriyle ayarlamak için d2l modülünün fonksiyonu çağrılır.
```

```python
class SuccessiveHalvingScheduler(d2l.HPOScheduler):  # Successive Halving Scheduler sınıfı d2l.HPOScheduler sınıfından türetilir.
    def __init__(self, searcher, eta, r_min, r_max, prefact=1):  # Başlangıç parametreleri
        self.save_hyperparameters()  # Hiperparametreleri kaydeder
        # K değeri hesaplanır; bu değer, yapılandırma sayısını belirlemek için kullanılır
        self.K = int(np.log(r_max / r_min) / np.log(eta))  # r_max ve r_min arasındaki oranı eta ile logaritmik olarak hesaplar
        # Rung seviyeleri tanımlanır (adımlar arası değerler)
        self.rung_levels = [r_min * eta ** k for k in range(self.K + 1)]  # r_min'den başlayarak rung seviyeleri belirlenir
        if r_max not in self.rung_levels:
            # Son rung seviyesinin r_max olması gerektiği kontrol edilir
            self.rung_levels.append(r_max)  # r_max rung seviyesine eklenir
            self.K += 1  # K değeri artırılır
        # Error (hata) verilerini saklamak için veri yapıları
        self.observed_error_at_rungs = defaultdict(list)  # Hata verilerini saklamak için defaultdict kullanılır
        self.all_observed_error_at_rungs = defaultdict(list)  # Tüm hata verilerini saklamak için başka bir defaultdict
        # İşlem sırası için kuyruk tanımlanır
        self.queue = []  # Kuyruk başlatılır
```

```python
@d2l.add_to_class(SuccessiveHalvingScheduler)  # SuccessiveHalvingScheduler sınıfına yeni bir fonksiyon ekler.
def suggest(self):
    if len(self.queue) == 0:  # Kuyruk boşsa yeni bir halving turuna başla
        # İlk rung için yapılandırma sayısı:
        n0 = int(self.prefact * self.eta ** self.K)  # n0, yapılandırma sayısını hesaplar
        for _ in range(n0):  # n0 kadar yapılandırma oluşturulacak
            config = self.searcher.sample_configuration()  # Arama fonksiyonundan yeni bir yapılandırma alınır
            config["max_epochs"] = self.r_min  # max_epochs değeri r_min olarak ayarlanır
            self.queue.append(config)  # Kuyruğa yeni yapılandırma eklenir
    # Kuyruktan bir öğe döndürülür
    return self.queue.pop()  # Kuyruğun son öğesini çıkarıp döndürür
```

```python
@d2l.add_to_class(SuccessiveHalvingScheduler)  # SuccessiveHalvingScheduler sınıfına 'update' fonksiyonu ekleniyor.
def update(self, config: dict, error: float, info=None):
    ri = int(config["max_epochs"])  # r_i rung seviyesini alır (max_epochs parametresi üzerinden).

    # Arama fonksiyonu güncelleniyor, örneğin daha sonra Bayesyen optimizasyonu kullanılabilir.
    self.searcher.update(config, error, additional_info=info)

    # Hata bilgisi, belirli rung seviyelerinde saklanır.
    self.all_observed_error_at_rungs[ri].append((config, error))

    if ri < self.r_max:  # Eğer şu anki rung, maksimum rung seviyesinden küçükse
        # Gözlemler yapılır ve hata bilgileri kaydedilir.
        self.observed_error_at_rungs[ri].append((config, error))

        # Bu rung seviyesinde kaç yapılandırma değerlendirileceği belirlenir.
        ki = self.K - self.rung_levels.index(ri)
        ni = int(self.prefact * self.eta ** ki)

        # Eğer bu rung seviyesindeki tüm yapılandırmalar gözlemlendiyse, en iyi performans gösteren yapılandırmalar seçilir.
        if len(self.observed_error_at_rungs[ri]) >= ni:
            kiplus1 = ki - 1
            niplus1 = int(self.prefact * self.eta ** kiplus1)

            # En iyi performans gösteren yapılandırmalar alınır ve bir üst rung seviyesine geçiş için kuyruğa eklenir.
            best_performing_configurations = self.get_top_n_configurations(
                rung_level=ri, n=niplus1
            )
            riplus1 = self.rung_levels[self.K - kiplus1]  # r_{i+1} rung seviyesi
```

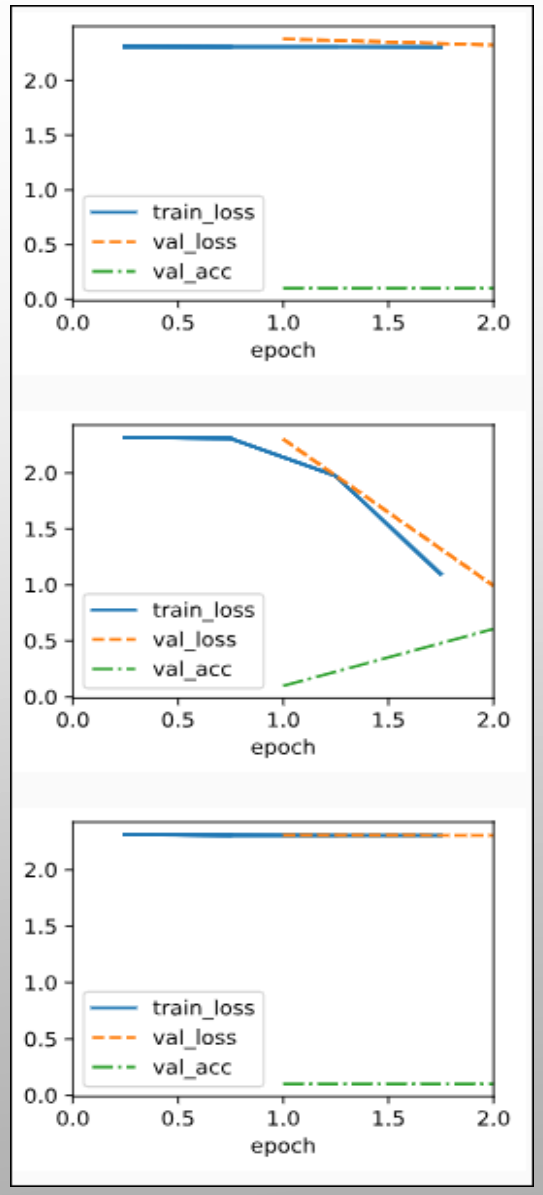
```
            # Kuyruk boş olmamalıdır: yeni yapılandırmalar başa eklenir
            self.queue = [
                dict(config, max_epochs=riplus1)
                for config in best_performing_configurations
            ] + self.queue

            # Bu rung seviyesindeki gözlemler sıfırlanır.
            self.observed_error_at_rungs[ri] = []
```

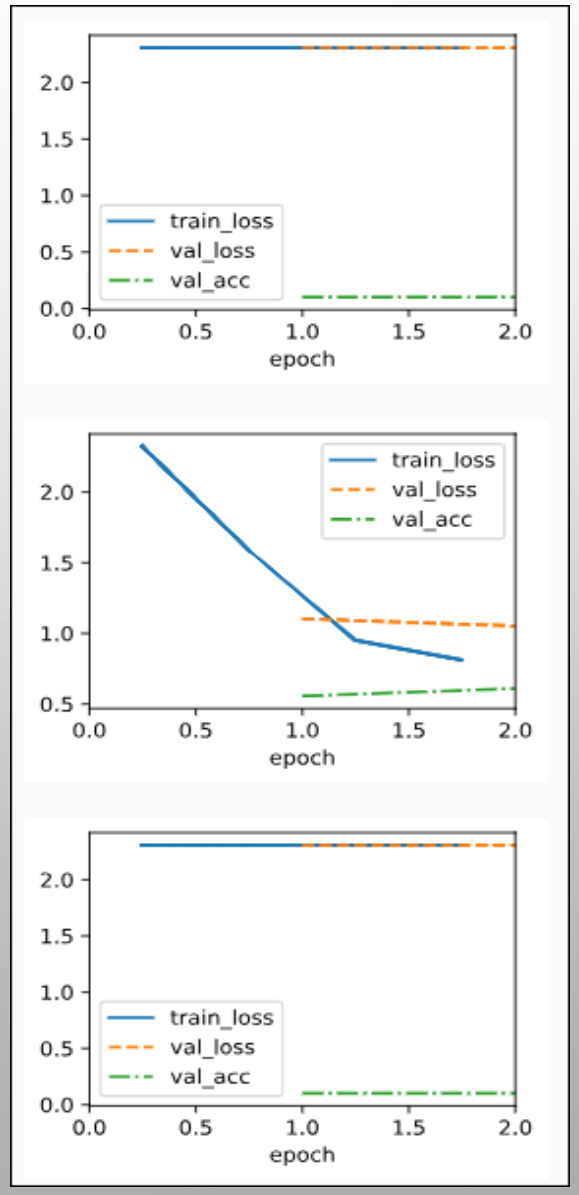
```
@d2l.add_to_class(SuccessiveHalvingScheduler)  # SuccessiveHalvingScheduler sınıfına 'get_top_n_configurations' fonksiyonu ekler.
def get_top_n_configurations(self, rung_level, n):
    rung = self.observed_error_at_rungs[rung_level]  # Belirtilen rung seviyesindeki gözlemleri alır.

    if not rung:  # Eğer rung boşsa, boş bir liste döndürür.
        return []

    # Gözlemleri hata değerine göre sıralar (artık sıralama yapılır).
    sorted_rung = sorted(rung, key=lambda x: x[1])

    # En iyi n yapılandırmayı döndürür.
    return [x[0] for x in sorted_rung[:n]]  # Sıralı yapılandırmalardan en iyi n tanesini alır.
```

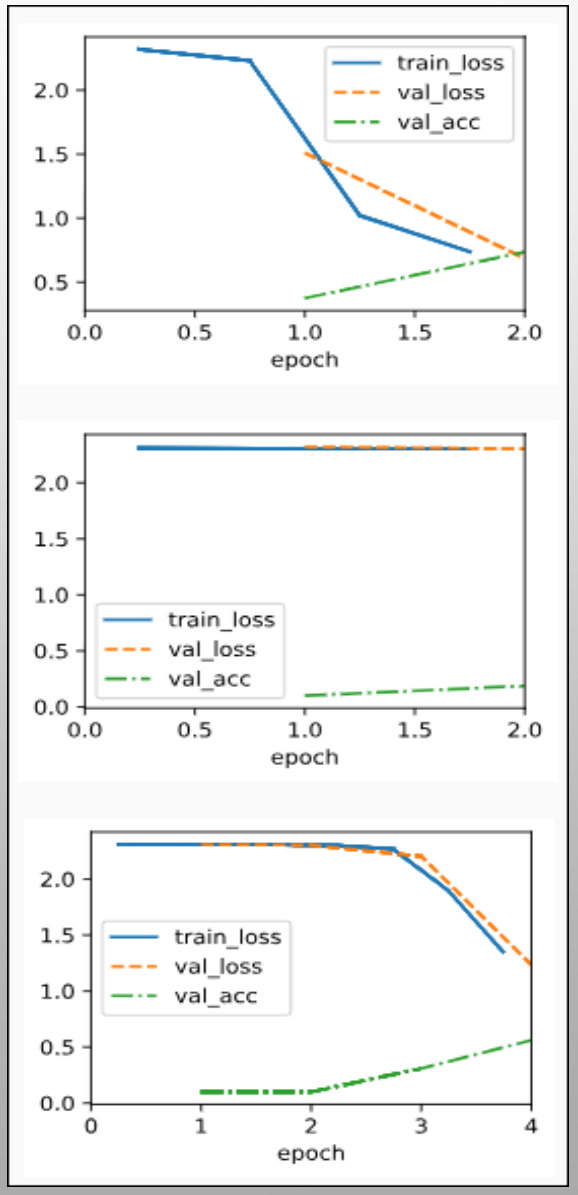
```python
min_number_of_epochs = 2  # Eğitimdeki minimum epoch sayısı (başlangıç değeri)
max_number_of_epochs = 10  # Eğitimdeki maksimum epoch sayısı (maksimum limit)
eta = 2  # Azaltma oranı, her rung'da değerlendirecek yapılandırma sayısını azaltır

# Konfigürasyon alanı
config_space = {
    "learning_rate": stats.loguniform(1e-2, 1),  # Öğrenme oranı logaritmik olarak 0.01 ile 1 arasında rastgele seçilecek
    "batch_size": stats.randint(32, 256),  # Batch boyutu 32 ile 256 arasında rastgele seçilecek
}

# Başlangıç yapılandırması
initial_config = {
    "learning_rate": 0.1,  # Başlangıç öğrenme oranı
    "batch_size": 128,  # Başlangıç batch boyutu
}
```

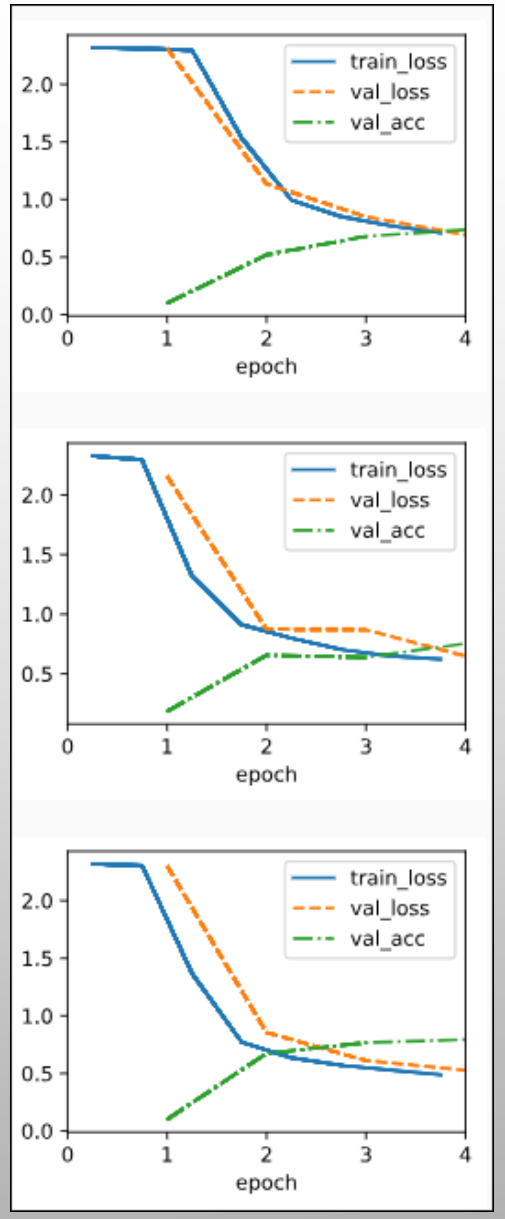
```python
# RandomSearcher, config_space içinde belirtilen hiperparametre aralıklarında rastgele arama yapacak bir arama stratejisidir.
searcher = d2l.RandomSearcher(config_space, initial_config=initial_config)

# SuccessiveHalvingScheduler, Successive Halving algoritmasını kullanarak hiperparametre aramasını yönetecek.
scheduler = SuccessiveHalvingScheduler(
    searcher=searcher,  # Arama stratejisi olarak searcher'ı kullanır
    eta=eta,  # Azaltma oranı
    r_min=min_number_of_epochs,  # Başlangıç epoch sayısı
    r_max=max_number_of_epochs,  # Maksimum epoch sayısı
)

# HPOTuner, hiperparametre optimizasyonunu yönetir ve objective fonksiyonunu kullanarak denemeleri yapar.
tuner = d2l.HPOTuner(
    scheduler=scheduler,  # Scheduler'ı kullanarak optimizasyonu yapar
    objective=d2l.hpo_objective_lenet,  # Kullanılacak objective fonksiyonu
)

# 30 deneme ile tuner'ı çalıştırır.
tuner.run(number_of_trials=30)
```

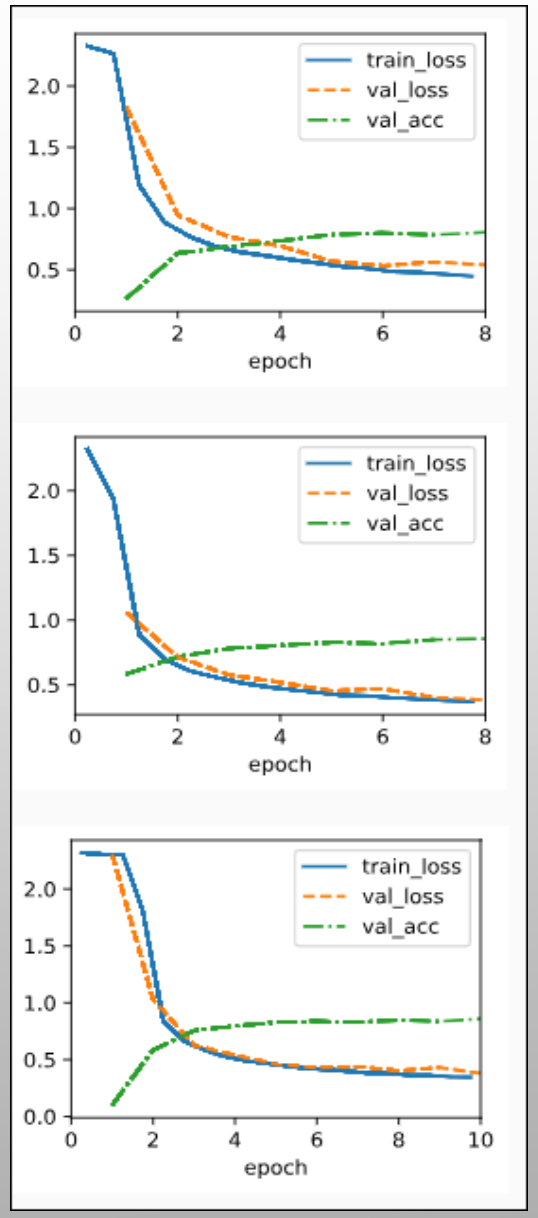
```
error = 0.17762434482574463, runtime = 53.576584339141846
```

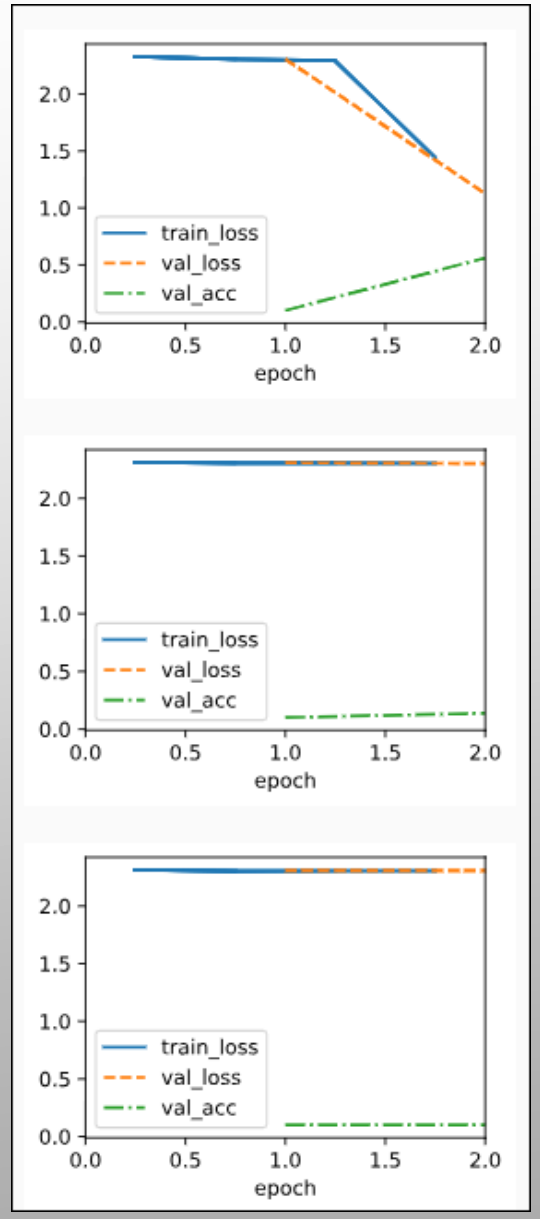


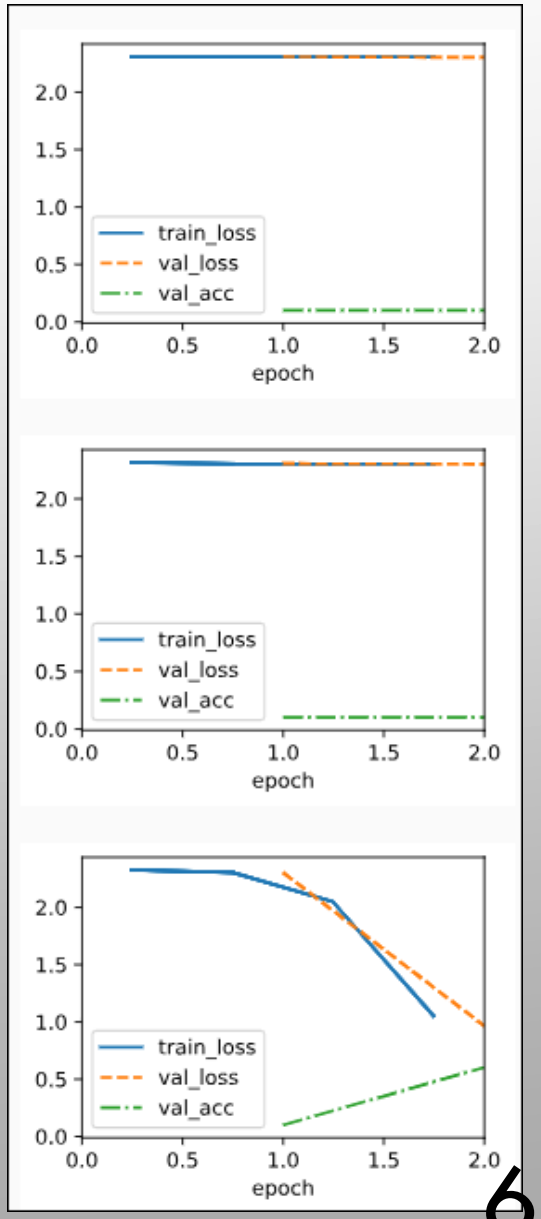


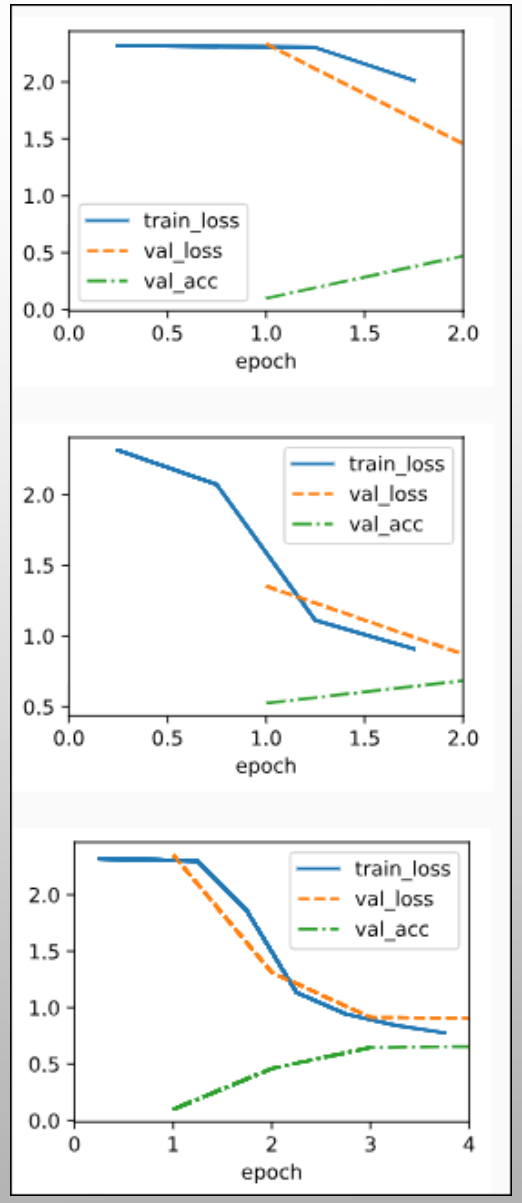


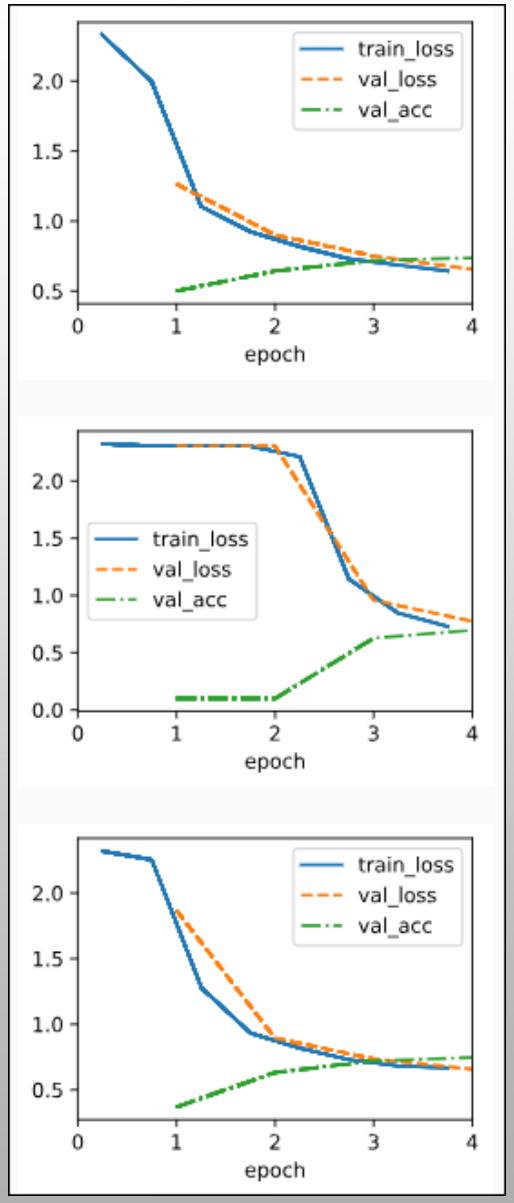

train_loss
val_loss
val_acc
epoch

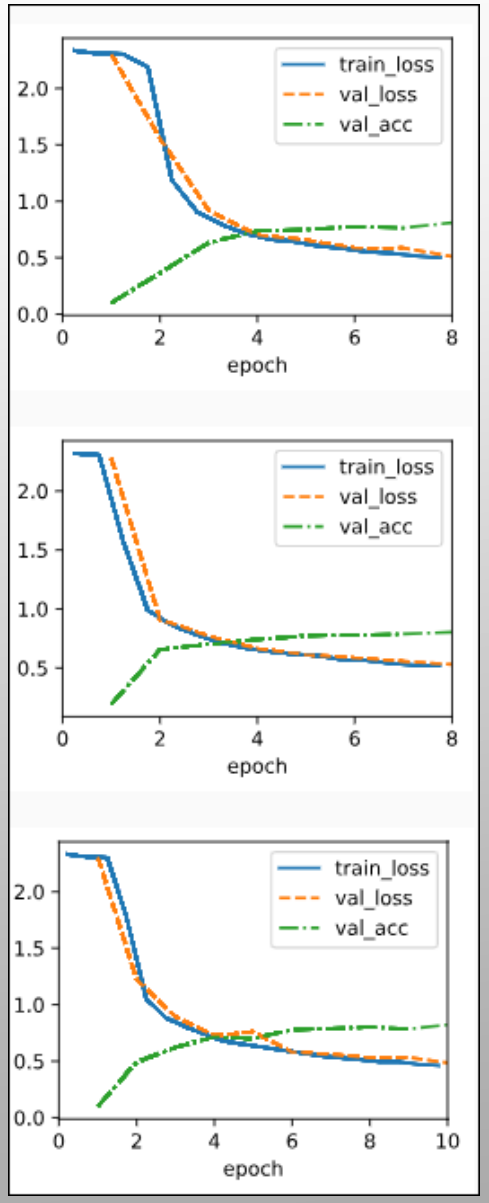

train_loss
val_loss
val_acc
epoch


train_loss
val_loss
val_acc
epoch


train_loss
val_loss
val_acc
epoch

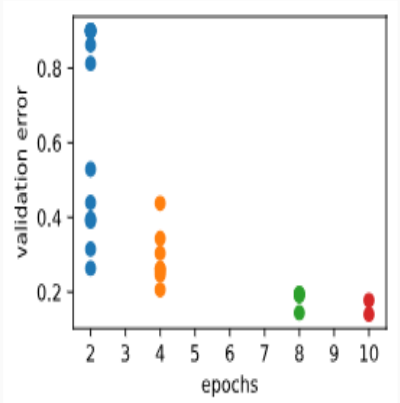

train_loss
val_loss
val_acc
epoch


train_loss
val_loss
val_acc
epoch


train_loss
val_loss
val_acc
epoch

```python
# Her rung seviyesi için gözlemlenen hata değerlerini alır ve görselleştirir
for rung_index, rung in scheduler.all_observed_error_at_rungs.items():
    # Her rung için hataları alır (rung'daki her öğe: (config, error))
    errors = [xi[1] for xi in rung]
    # Rung seviyesinde hata değerlerini scatter plot ile gösterir
    d2l.plt.scatter([rung_index] * len(errors), errors)

# x ekseninin sınırlarını belirler, epoch sayıları arasında hizalar
d2l.plt.xlim(min_number_of_epochs - 0.5, max_number_of_epochs + 0.5)
# x ekseninde her epoch sayısı için etiketler ekler
d2l.plt.xticks(
    np.arange(min_number_of_epochs, max_number_of_epochs + 1),
    np.arange(min_number_of_epochs, max_number_of_epochs + 1)
)
# Y eksenine etiket ekler (validation error)
d2l.plt.ylabel("validation error")
# X eksenine etiket ekler (epochs)
d2l.plt.xlabel("epochs")
```

```
Text(0.5, 0, 'epochs')
```



EŞZAMANSIZ ARDIŞIK YARILANMA (ASYNCHRONOUS SUCCESSİVE HALVİNG)

Asynchronous Successive Halving Algorithm (ASHA), büyük hiperparametre arama alanlarını verimli bir şekilde keşfetmek için kullanılan bir hiperparametre optimizasyon algoritmasıdır. ASHA, özellikle paralel ve asenkron çalışmayı destekleyen bir versiyonu olduğu için, geleneksel Successive Halving algoritmasının geliştirilmiş bir halidir.

ASHA, hiperparametre konfigürasyonlarını iterasyon sayısına (örneğin, epoch) göre aşamalı olarak değerlendiren ve daha düşük performans gösteren denemeleri erken sonlandırarak hesaplama maliyetini azaltan bir yöntemdir.

AVANTAJLARI

Kaynakların Daha Verimli Kullanımı:

- Modeller tamamlandıkça değerlendirilir, bekleme süresi ortadan kalkar.
- Tüm işlemciler ve GPU'lar sürekli olarak çalışır.

Daha Hızlı Sonuçlar:

- Klasik yönteme göre optimizasyon süreci daha hızlıdır.
- Paralel işleme sayesinde çok daha kısa sürede sonuç alınabilir.

Dinamik Uyum:

- Eşzamansız yöntem, değişen donanım veya veri işleme kapasitelerine kolayca uyarlanabilir.

Dağıtık Sistemlere Uygunluk:

- Dağıtık bir sistemde, modeller farklı sunucularda veya işlemcilerde eş zamanlı olarak çalıştırılabilir.

DEZAVANTAJLARI

İletişim ve Senkronizasyon Zorlukları:

- Dağıtık bir ortamda, sonuçların dinamik olarak toplanması ve senkronizasyonu zor olabilir.

Performans Metriği Hassasiyeti:

- Performansı düşük görünen bazı modeller erken elenebilir, ancak daha fazla kaynak sağlandığında iyi performans gösterebilirler.

Parametre Ayarı:

- eta, r_min gibi parametrelerin uygun şekilde ayarlanması gerekir. Yanlış ayarlar, kaynak israfına yol açabilir.

ARDIŞIK YARILANMA İLE EŞZAMANSIZ ARDIŞIK YARILANMA ARASINDAKİ FARKLAR

Senkron ve Eşzamansız İşleme:

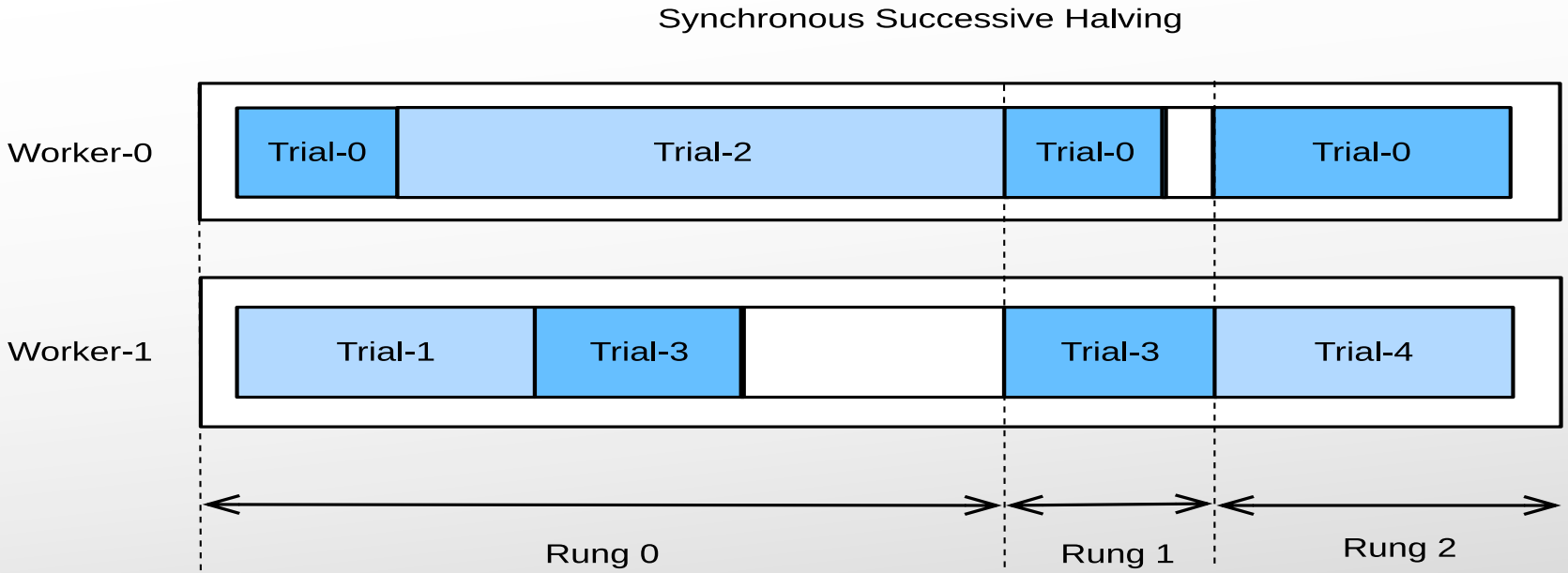
- Klasik Ardışık Yarılanma, tüm modellerin bir "turda" tamamlanmasını bekler, ardından düşük performanslı modeller elenir.
- Eşzamansız sürümde, her model bağımsız olarak çalıştırılır ve tur sıralamasına bağlı kalmaksızın düşük performanslı modeller eleme sürecine dahil edilir.

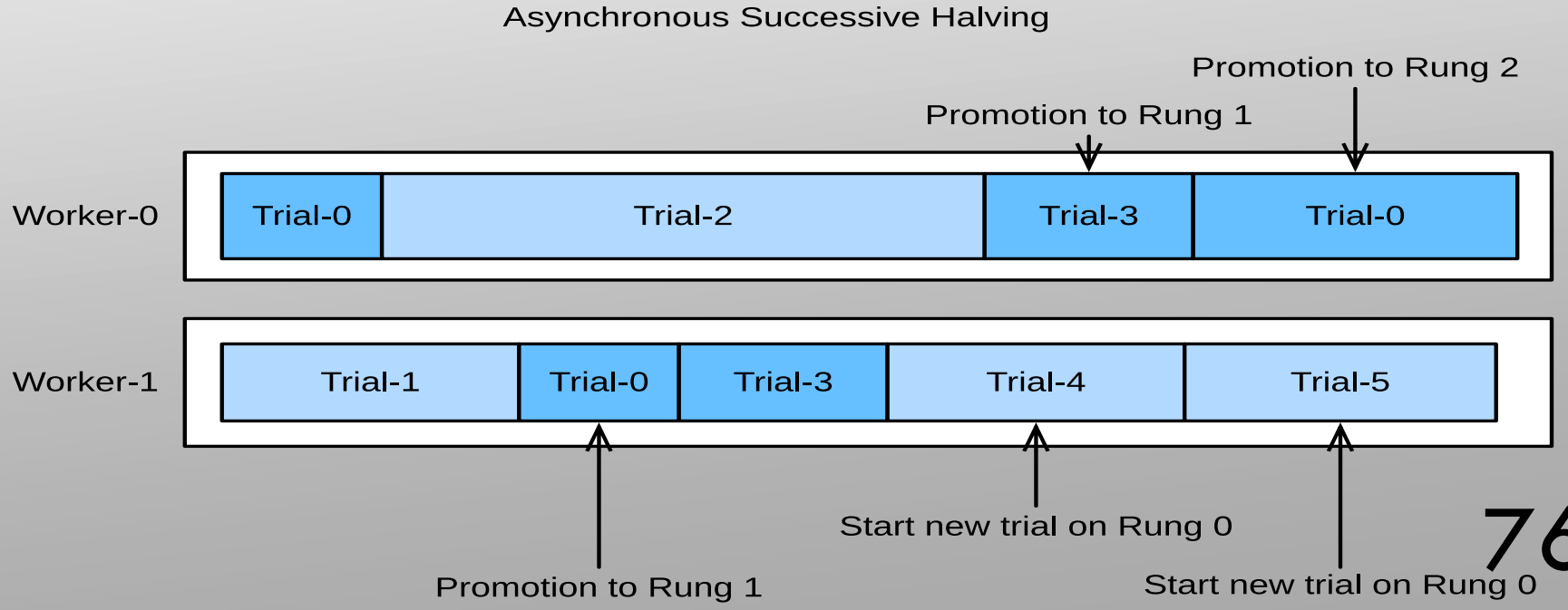
Zaman Kazancı:

- Klasik yöntemde, bir sonraki tur başlamadan önce tüm modellerin değerlendirilmesi tamamlanmalıdır.
- Eşzamansız yöntemde, bir model biter bitmez değerlendirilir ve eleme süreci dinamik olarak devam eder, bu da bekleme süresini azaltır.

Paralel İşleme:

- Eşzamansız yöntem, paralel kaynakları daha verimli kullanır çünkü kaynaklar hiçbir zaman "boşta" beklemez.


Synchronous Successive Halving
Worker-0
Trial-0
Trial-2
Trial-0
Trial-0
Worker-1
Trial-1
Trial-3
Trial-3
Trial-4
Rung 0
Rung 1
Rung 2


Asynchronous Successive Halving
Promotion to Rung 2
Promotion to Rung 1
Worker-0
Trial-0
Trial-2
Trial-3
Trial-0
Worker-1
Trial-1
Trial-0
Trial-3
Trial-4
Trial-5
Promotion to Rung 1
Start new trial on Rung 0
Start new trial on Rung 0

```python
import logging  # Loglama işlemleri için logging modülünü içe aktarır
from d2l import torch as d2l  # D2L kütüphanesini PyTorch için içe aktarır

logging.basicConfig(level=logging.INFO)  # Bilgi seviyesinde loglama başlatılır
import matplotlib.pyplot as plt  # Grafikler için matplotlib modülü içe aktarılır
from syne_tune import StoppingCriterion, Tuner  # SyneTune'dan durdurma kriteri ve tuner sınıfları içe aktarılır
from syne_tune.backend.python_backend import PythonBackend  # Python arka planda çalışacak backend içe aktarılır
from syne_tune.config_space import loguniform, randint  # Konfigürasyon alanı için rastgele seçim fonksiyonları
from syne_tune.experiments import load_experiment  # Deney yükleme fonksiyonu
from syne_tune.optimizer.baselines import ASHA  # ASHA optimizasyon algoritması içe aktarılır
```

```python
def hpo_objective_lenet_synetune(learning_rate, batch_size, max_epochs):
    from syne_tune import Reporter  # SyneTune'dan Reporter sınıfı içe aktarılır (sonuç raporlamak için)
    from d2l import torch as d2l  # D2L kütüphanesi PyTorch için içe aktarılır

    model = d2l.LeNet(lr=learning_rate, num_classes=10)  # LeNet modelini oluşturur, öğrenme oranı ve sınıf sayısı belirlenir
    trainer = d2l.HPOTrainer(max_epochs=1, num_gpus=1)  # Model eğitimi için HPOTrainer sınıfı, bir GPU ile başlatılır
    data = d2l.FashionMNIST(batch_size=batch_size)  # FashionMNIST veri seti, belirli batch boyutunda yüklenir
    model.apply_init([next(iter(data.get_dataloader(True)))[0]], d2l.init_cnn)  # Modelin parametreleri CNN ile başlatılır

    report = Reporter()  # Sonuçları raporlamak için Reporter nesnesi oluşturulur
    for epoch in range(1, max_epochs + 1):  # Belirtilen maksimum epoch sayısına kadar eğitim yapılır
        if epoch == 1:
            # İlk epoch'ta Trainer'ı başlat ve fit işlemini gerçekleştir
            trainer.fit(model=model, data=data)
        else:
            trainer.fit_epoch()  # Sonraki epoch'larda sadece fit_epoch çağrılır

        validation_error = trainer.validation_error().cpu().detach().numpy()  # Geçerlilik hatası hesaplanır
        report(epoch=epoch, validation_error=float(validation_error))  # Epoch ve doğrulama hatası raporlanır
```

```
min_number_of_epochs = 2  # Eğitimde minimum epoch sayısı belirlenir
max_number_of_epochs = 10  # Eğitimde maksimum epoch sayısı belirlenir
eta = 2  # Eğitimde öğrenme oranı (learning rate) değişim oranı

config_space = {
    "learning_rate": loguniform(1e-2, 1),  # Öğrenme oranı, 1e-2 ile 1 arasında logaritmik dağılımla seçilir
    "batch_size": randint(32, 256),  # Batch boyutu, 32 ile 256 arasında rastgele bir değer seçilir
    "max_epochs": max_number_of_epochs,  # Maksimum epoch sayısı olarak yukarıda tanımlanan değeri alır
}

initial_config = {
    "learning_rate": 0.1,  # Başlangıçta öğrenme oranı 0.1 olarak belirlenir
    "batch_size": 128,  # Başlangıçta batch boyutu 128 olarak belirlenir
}
```

```
n_workers = 2  # Kullanılacak işçi (worker) sayısını belirler. Bu sayı mevcut GPU sayısından büyük olmamalıdır.
max_wallclock_time = 12 * 60  # Maksimum çalıştırma süresini saniye cinsinden belirler. Burada 12 dakika olarak ayarlanmıştır.
```

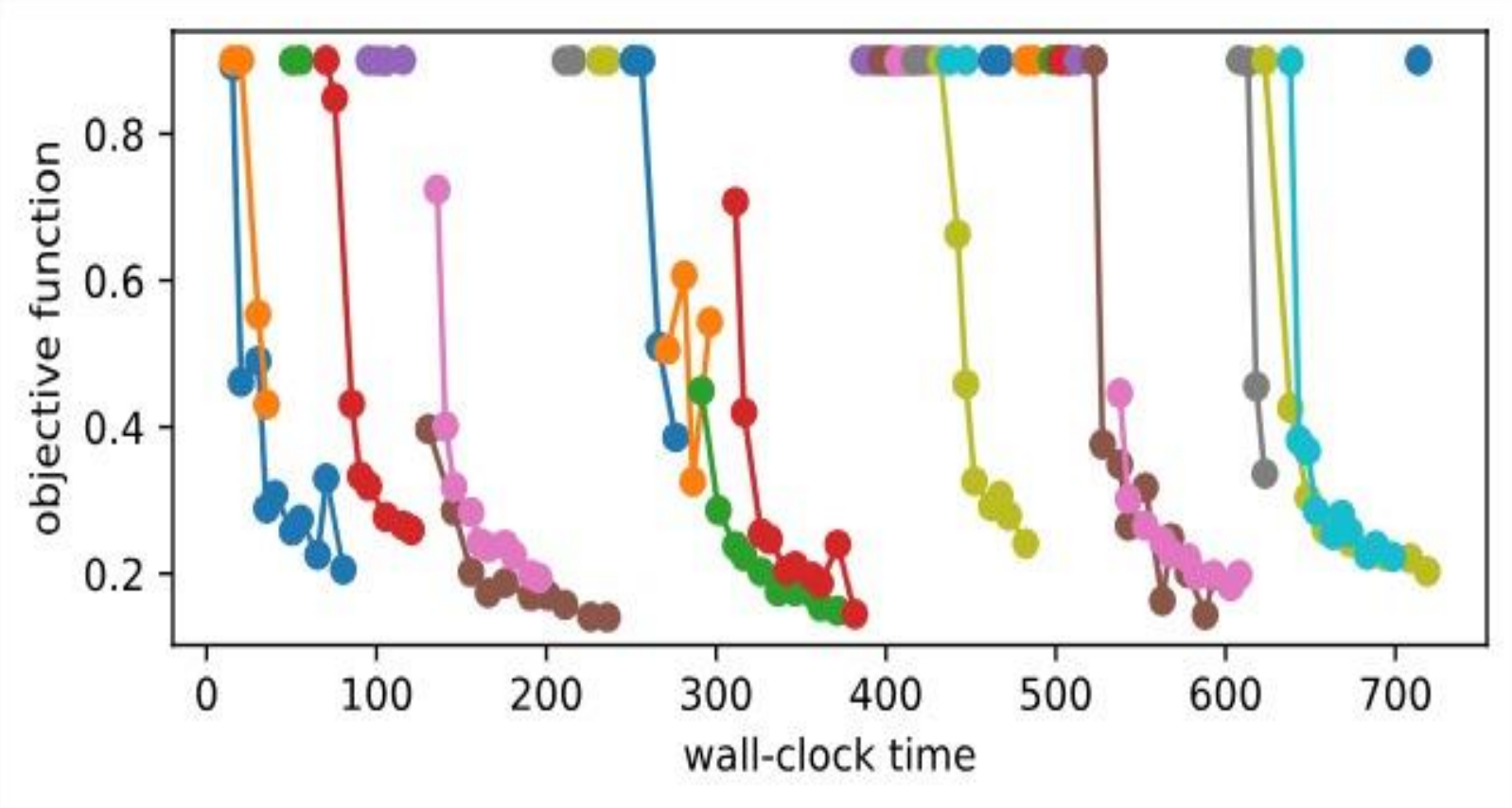
```
[ ]  mode = "min"  # Performans ölçütü olan metrik, düşük olmasını hedefler (örneğin hata oranı)
     metric = "validation_error"  # Değerlendirilen metrik, doğrulama hatası olarak belirlenir
     resource_attr = "epoch"  # Kaynak olarak epoch kullanılır (yani, her epoch sonrası değerlendirme yapılır)

     # ASHA (Asynchronous Successive Halving Algorithm) sınıfı ile hiperparametre optimizasyonu başlatılır
     scheduler = ASHA(
         config_space,  # Hiperparametre arama alanı
         metric=metric,  # İzlenecek metrik
         mode=mode,  # Düşük metrik hedefi
         points_to_evaluate=[initial_config],  # Başlangıç için değerlendirilecek konfigürasyon
         max_resource_attr="max_epochs",  # Maksimum kaynak, burada max_epochs kullanılır
         resource_attr=resource_attr,  # Kaynak (epoch) kullanılarak değerlendirilecek
         grace_period=min_number_of_epochs,  # İlk dönem için tolerans süresi (başlangıçta min_number_of_epochs)
         reduction_factor=eta,  # Hiperparametreler arasında performansın nasıl azaltılacağını belirler (eta)
     )
```

```
[ ] trial_backend = PythonBackend(
        tune_function=hpo_objective_lenet_synetune,  # Hiperparametre optimizasyonu fonksiyonu belirlenir
        config_space=config_space,  # Hiperparametre arama alanı atanır
    )

    stop_criterion = StoppingCriterion(max_wallclock_time=max_wallclock_time)  # Eğitim süresi limitini belirler (max_wallclock_time)
    tuner = Tuner(
        trial_backend=trial_backend,  # Eğitim ve model parametrelerini yöneten backend belirlenir
        scheduler=scheduler,  # ASHA algoritması ile hiperparametrelerin optimizasyonu yapılır
        stop_criterion=stop_criterion,  # Eğitim süresini sınırlamak için durdurma kriteri belirlenir
        n_workers=n_workers,  # Çalışan sayısı (paralel işlemler) belirlenir
        print_update_interval=int(max_wallclock_time * 0.6),  # Güncellemelerin yazdırılma aralığı belirlenir
    )
    tuner.run()  # Tuner çalıştırılır, hiperparametre optimizasyonu başlatılır
```

```
[ ]  d2l.set_figsize()  # Grafiklerin boyutlarını ayarlamak için D2L kütüphanesindeki set_figsize fonksiyonu çağrılır

     e = load_experiment(tuner.name)  # Tuned modelin geçmiş deneyini yükler. 'tuner.name' ile deneyin adı alınır.
     e.plot()  # Yüklenen deneyin grafiğini çizdirir (örneğin, doğrulama hatası vs. epoch)
```

```
[ ]  d2l.set_figsize([6, 2.5])  # Grafik boyutları [6, 2.5] olarak ayarlanır.

     results = e.results  # Deney sonuçları 'results' değişkenine atanır.

     # Her bir deneme için (trial_id) sonuçlar döngüye alınır
     for trial_id in results.trial_id.unique():
         df = results[results["trial_id"] == trial_id]  # Bu trial_id için veriler filtrelenir.
         # Sonuçların grafikleştirilmesi
         d2l.plt.plot(
             df["st_tuner_time"],  # X eksenine tuner zamanları (saat cinsinden) konur
             df["validation_error"],  # Y eksenine doğrulama hatası yerleştirilir
             marker="o"  # Noktalarla işaretlenmiş grafik çizilir
         )

     # Grafik etiketlerinin eklenmesi
     d2l.plt.xlabel("wall-clock time")  # X ekseni için etiket
     d2l.plt.ylabel("objective function")  # Y ekseni için etiket
```

```
Text(0, 0.5, 'objective function')
```



KAYNAKÇA

- https://d2l.ai/chapter_hyperparameter-optimization/hyperopt-intro.html
- https://medium.com/bili%C5%9Fim-hareketi/hiperparametre-optimizasyonu-9ba0e7f32e6f#:~:text=2.-,Hiperparametre%20Optimizasyonu%20Nedir%3F,overfitting%20ve%20underfitting%20dengesi%20sa%C4%9Flanabilir.
- https://chatgpt.com
- https://medium.com/deep-learning-turkiye/derin-ogrenme-uygulamalarinda-en-sik-kullanilan-hiper-parametreler-ece8e9125c4
- https://medium.com/deep-learning-turkiye/derin-ogrenme-uygulamalarinda-model-dogrulama-ve-hiper-parametre-secim-yontemleri-823812d95f3

- Görseller: https://miro.medium.com/v2/resize:fit:1200/1*gGBUBuRiwJvmINzQSeOl1w.png