12. Előadás

Python kurzus

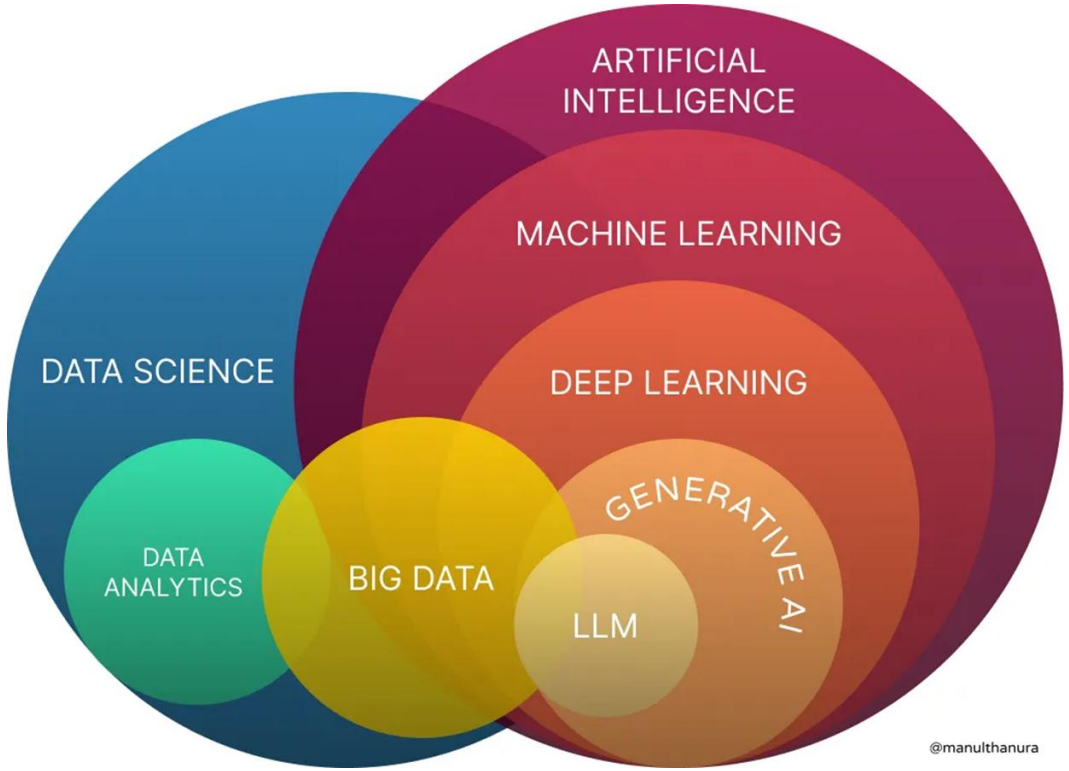
Tárgyfelelős:
Dr Tejfel Máté

Előadó:
Horváthné Hadobás
Olga Erzsébet

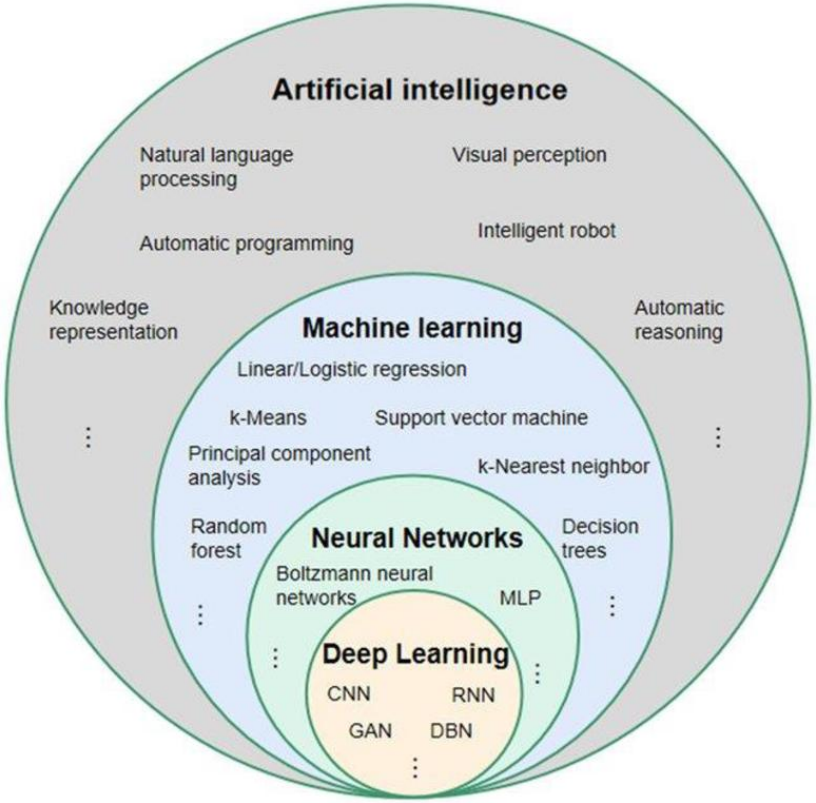
12. Előadás tematikája

Bevezetés a mélytanulásba

1. Deep Learning fogalmak
2. A TensorFlow könyvtár alapjai
3. Neurális hálózatok felépítése TensorFlow-val
4. Egyszerű neurális hálózat megvalósítása
5. Modell betanítása és kiértékelése TensorFlow-val
6. TensorFlow modellek generálása és optimalizálása
7. PyTorch Mélytanulási keretrendszer



https://manulthanura.medium.com/demystifying-ai-a-dive-into-key-concepts-67225ec6cdb2



https://www.researchgate.net/figure/Relations-between-artificial-intelligence-machine-learning-neural-network-and-deep_fig2_375110440

1. Deep Learning fogalmak

A mélytanulás definíciója

A mélytanulás a mesterséges intelligencia (AI) és gépi tanulás (ML) egyik speciális ága, amely a **neurális hálózatokon** alapul. Ezek a modellek úgy tanulnak, hogy nagyméretű adatbázisokból **mintázatokat azonosítanak**, és a tanulás során **több rétegből álló architektúrát** használnak.

A mélytanulás főbb jellemzői:

1. Többrétegű tanulás: Az adatokat több absztrakciós szinten dolgozza fel.

2. Hatalmas mennyiségű adat feldolgozása: Nagy adatbázisok szükségesek a pontos modellezéshez.

3. Magas számítási igény: GPU-k (Graphical Processing Unit) és TPU-k (Tensor Processing Unit) szükségesek a számításigényes feladatokhoz.

- 1980: VIC-20 5kb RAM, MOS 6502 CPU 1,02Mhz
- 2022: NVIDIA GeForce GTX 3090Ti, 24 GB RAM, 1860 MHz, 10752 CUDA mag

Mélytanulás – gépi tanulás

A jellemzők (features) kiválasztása:

- Hagyományos gépi tanulás: Kézi kiválasztást igényel.
- Mélytanulás: Automatikusan tanulja meg a legfontosabb jellemzőket az adatokból.

Mintázatok felismerése:

- Gépi tanulás: Egyszerűbb modellek alkalmazása.
- Mélytanulás: Nem lineáris mintázatok felismerése.

Skálázhatóság:

- Mélytanulás: Jól működik nagy mennyiségű adat és összetett problémák esetén.

Jellemző	Machine Learning	Deep Learning
Jellemzők kiválasztása	Kézi kiválasztás	Automatikus tanulás
Adatigény	Kisebb adathalmaz	Nagy adathalmaz
Számítási igény	Alacsony	Magas
Átláthatóság	Könnyen értelmezhető	Nehezen interpretálható (fekete doboz)
Teljesítmény	Egyszerű problémákra optimalizált	Komplex problémákra optimalizált
Alkalmazási területek	Strukturált adatok	Képek, szövegek, hangok feldolgozása

Gyakorlati példák

Kép- és hangfeldolgozás:

- Arcfelismerés: Biztonsági rendszerek, okostelefonok azonosítása.
- Tárgyfelismerés: Autonóm drónok vagy önvezető autók érzékelési rendszerei.
- Hangalapú rendszerek: Alexa, Siri, Google Assistant.

Természetes nyelv feldolgozás (NLP):

- Automatikus fordítás: Google Translate.
- Chatbotok és virtuális asszisztensek.
- Szövegértelmezés: Tartalom moderálás, hírek összefoglalása.

Önállóan tanuló rendszerek:

- Önvezető autók: Szenzoradatok feldolgozása és döntéshozatal.
- Robotika: Autonóm gépek, alkalmazkodás a környezethez.
- Játékfejlesztés: AI bonyolult játékokban.

Rövid történeti áttekintés

Kezdetek:

- 1943: McCulloch és Pitts megalkotja az **első mesterséges neuront**.
- 1958: Frank Rosenblatt bemutatja a **perceptron modellt**.

Stagnálás:

- Az 1970-es és 1980-as években technológiai korlátok miatt lelassult a fejlődés.

Megújulás:

- 1986: Geoffrey Hinton és társai bemutatják a **visszaterjesztés (Backpropagation) algoritmust**.
- 2000-es évek: Nagy adatbázisok és modern GPU-k révén új lendületet kapott a mélytanulás.

Napjaink:

- Mélytanulás az ipar és tudomány szinte minden területén jelen van.

Mélyhálók alapfogalmai

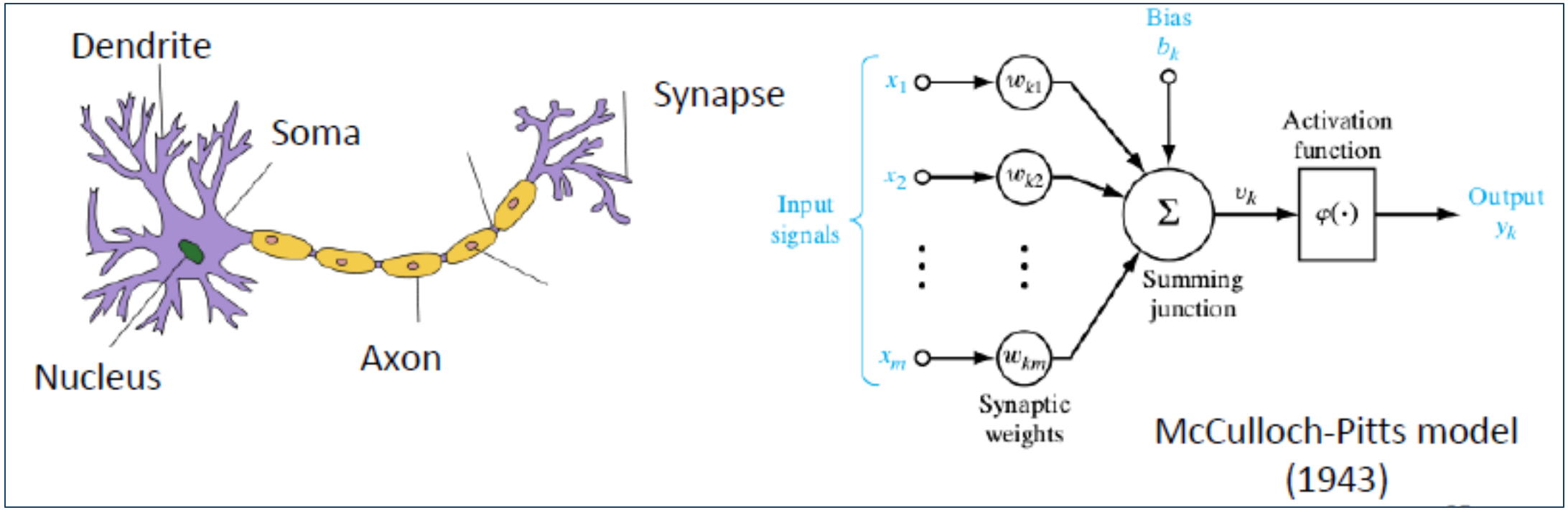
1. Neurális hálózatok

A neurális hálózatok az agyban található neuronok működését modellezik. Ezek mesterséges formában történő létrehozása = **mesterséges neuron** lehetővé teszi a gépi tanulás különböző formáit.

- **Rétegek:**
  - Bemeneti réteg: Az adatokat fogadja, pl. egy kép pixeleinek intenzitása.
  - Rejtett rétegek: Feldolgozzák az adatokat, felismerve a **mintázatokat**.
  - Kimeneti réteg: Az eredményeket adja vissza, pl. az osztályozás eredményét (0–9 számjegyek).
- **Neuronok:** A neurális hálózat elemei, amelyek **egy-egy értéket számolnak ki.** A neuronok a **bemeneteket súlyozzák**, majd egy **aktivációs függvényen** keresztül eredményt adnak.
- **Kapcsolatok:** A neurális hálózatban a neuronokat **élek (kapcsolatok)** kötik össze. Minden kapcsolat egy **súlyértékkel** rendelkezik, amely a tanulási folyamat során **frissül**.

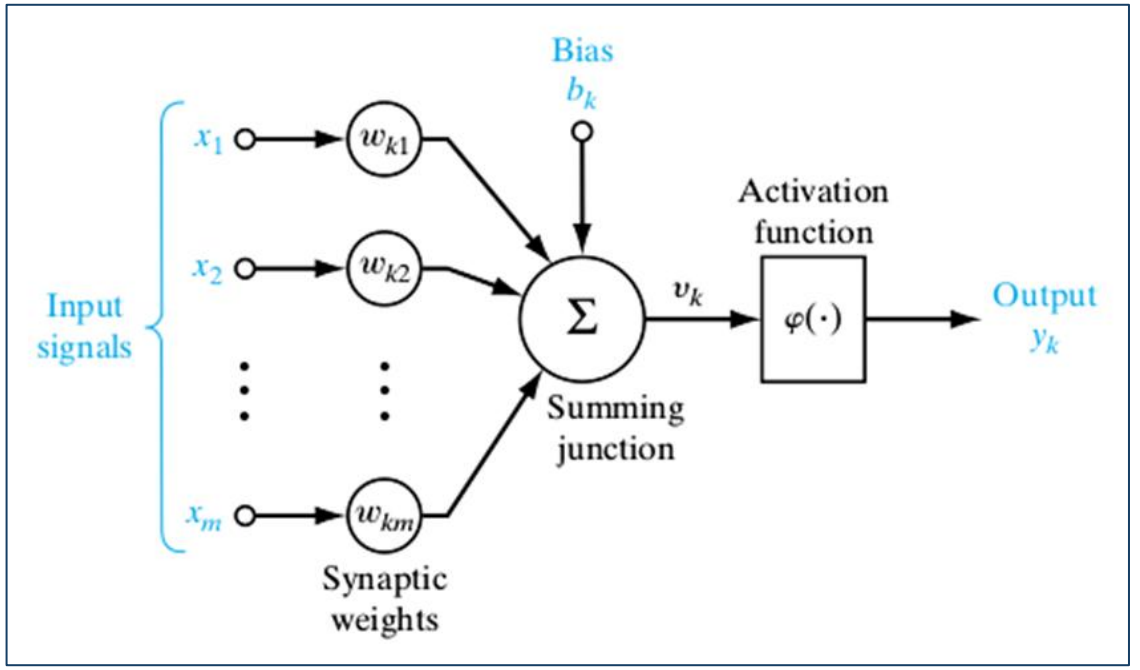
A mesterséges neuron (McCulloch-Pitts)

• A biológiai idegsejt alapján modellezve
• A mesterséges neuron egy információfeldolgozó egység, amely a mesterséges neurális hálózat felépítésének alapeleme.



A mesterséges neuronok (perceptronok) összetevői

- Bemenetet fogad a szinapszison (x_i) keresztül (az axontól a dendritekig):
  - a **bemenetek súlyozottak** (w_i)
  - ha $w_i > 0$: erősített bemenet az adott forrásból (gerjesztő bemenet)
  - ha $w_i < 0$: csillapított bemenet az adott forrásból (gátló bemenet)
- **Súlyozott összeget** számítanak ki
- A **b érték torzítja** az összeget, hogy lehetővé tegye az aszimmetrikus viselkedést
- **Aktivációs függvény** formálja a kimenetet



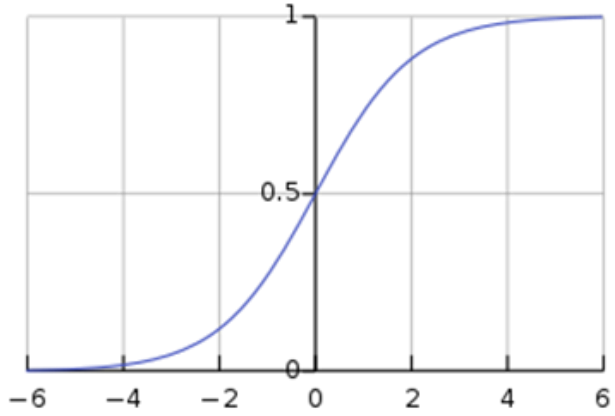
x_i: bemeneti vektor
w_{ki}: a k neuron súlyegyüttható-vektora
b_k: a k neuron torzítási értéke
y_k: a k neuron kimeneti értéke

2. Aktivációs függvények

Az aktivációs függvények adják meg, hogy egy neuron kimenete hogyan alakuljon a következő réteg számára.

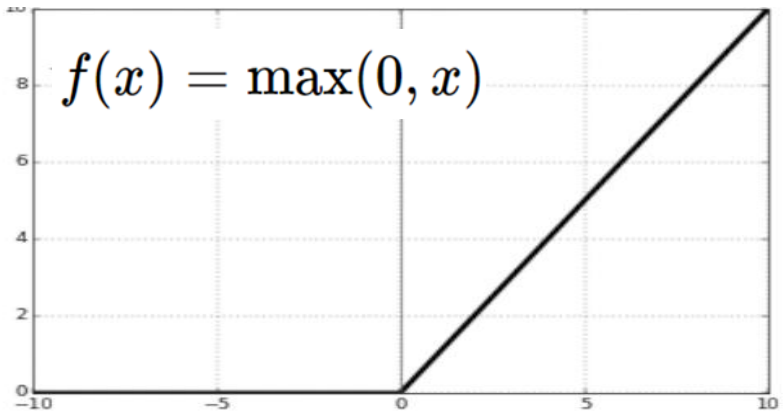
- **Sigmoid:** Az értékeket 0 és 1 közé skálázza.
- **ReLU (Rectified Linear Unit):** Minden negatív bemenetet 0-ra állít, a pozitív értékeket változatlanul hagyja.
- **Softmax:** Az osztályok valószínűségi eloszlását számítja ki (összegük 1).

Sigmoid



$$S(x) = \frac{1}{1+e^{-x}}$$

ReLU



Mélyhálók és Deep Learning sajátosságai

1. A mélytanulási modellek több rejtett réteggel rendelkeznek, amelyeken keresztül az adatokat feldolgozzák. Ezek a rétegek hierarchikus módon tanulnak mintázatokat:

- Az első rétegek az egyszerűbb jellemzőket tanulják meg (pl. élek képfeldolgozásnál).
- A mélyebb rétegek bonyolultabb mintázatokat azonosítanak (pl. szemek, arcok).

2. Adatmennyiség és számítási igény

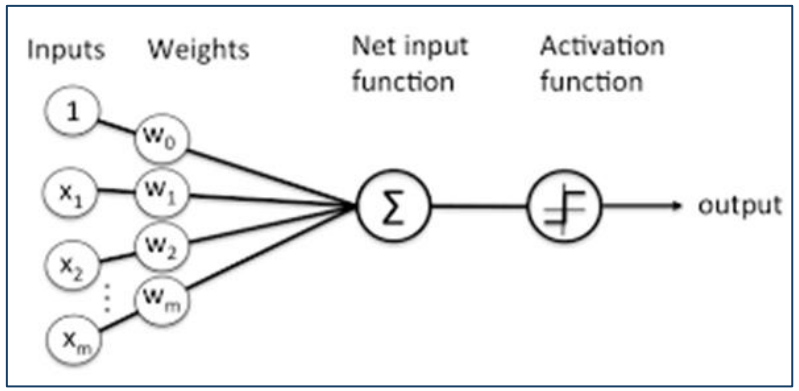
- **Adatmennyiség:** A mélyhálók hatékonysága a nagy méretű adathalmazokkal javul.
- **Számítási kapacitás:** Erős GPU-k és TPU-k segítik a mélyhálók gyors tanítását.

3. Hálózati architektúrák

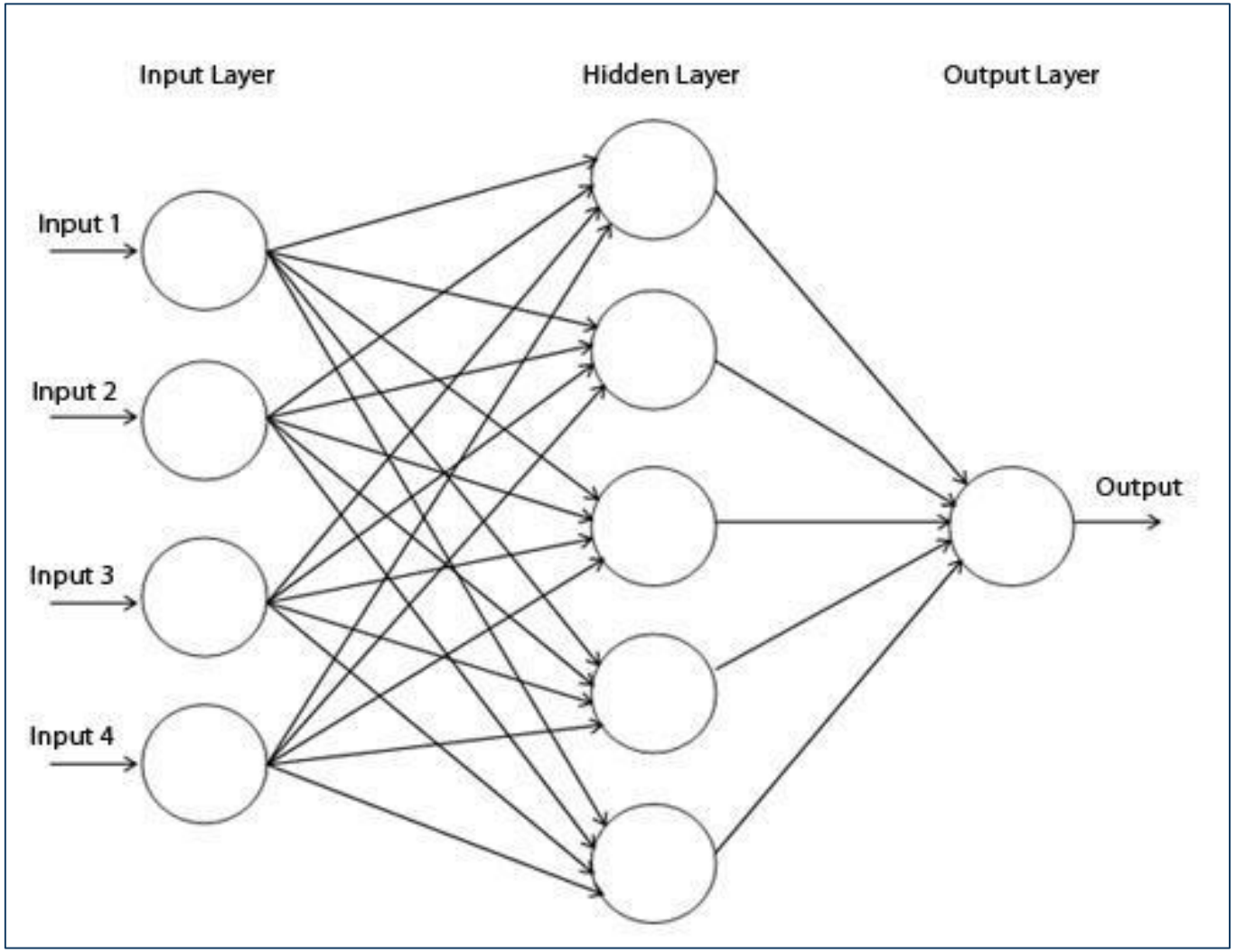
Különböző mélytanulási architektúrák léteznek, amelyek specifikus problémákra optimalizáltak:

- **Convolutional Neural Networks (CNNs):** Képfeldolgozásra.
- **Recurrent Neural Networks (RNNs):** Idősorok és természetes nyelv feldolgozására.
- **Transformer modellek:** Nagy nyelvi modellek (pl. GPT, BERT) számára

Neurális hálózat osztályozója



Többrétegű neurális hálózat



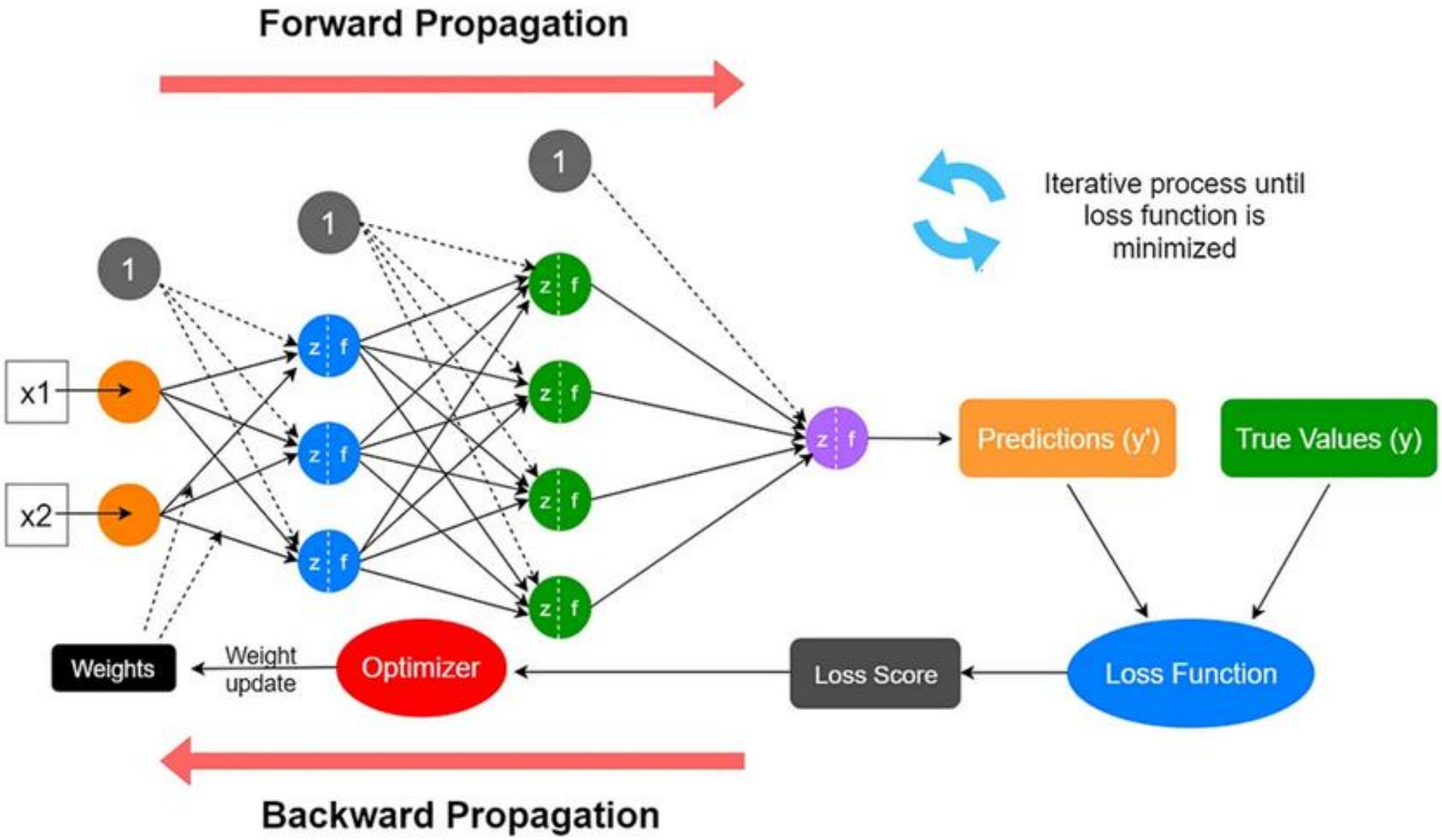
Visszaterjesztés (Backpropagation) algoritmus – 1986, Geoffrey Hinton és társai

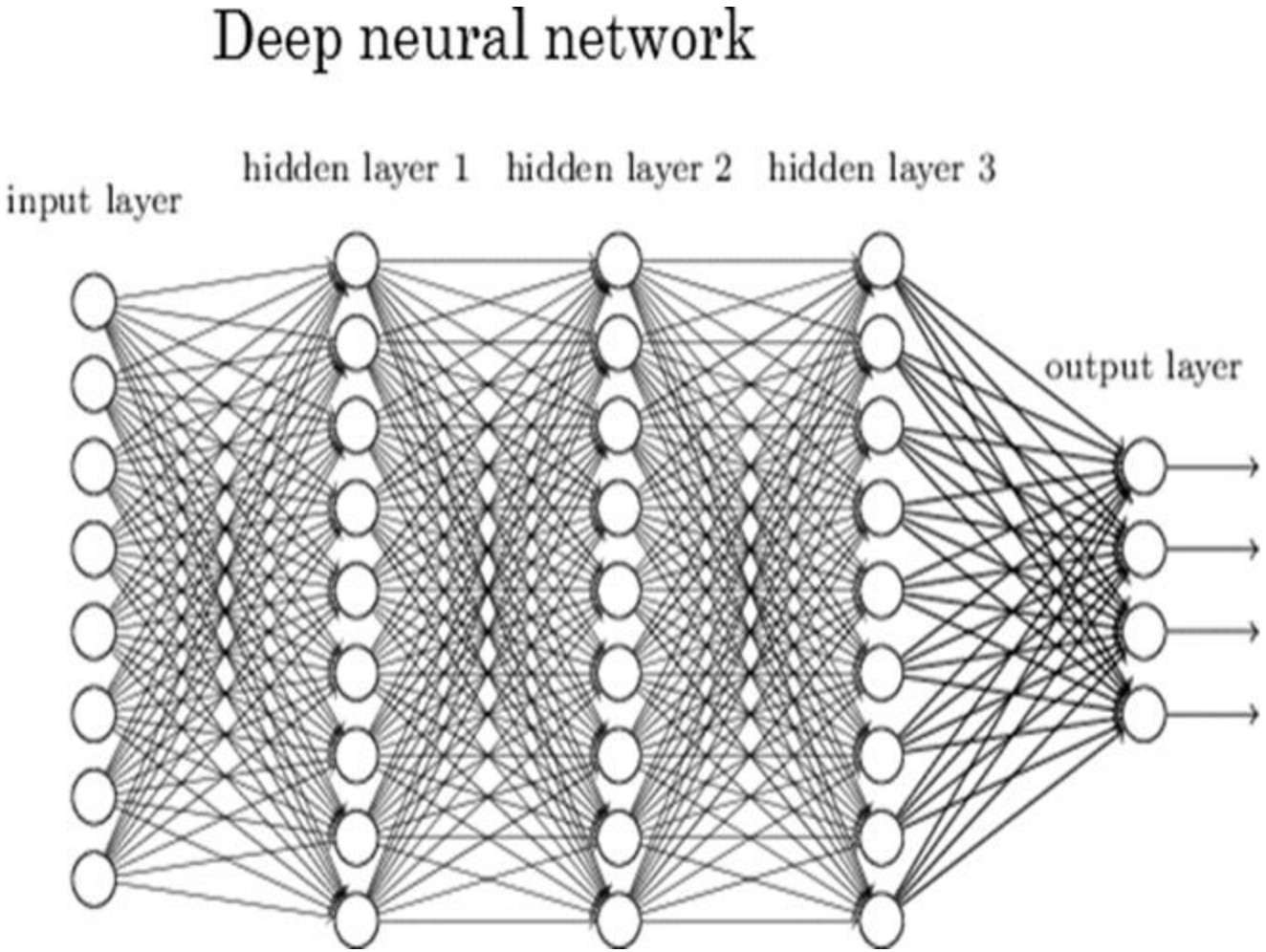
A visszaterjesztés algoritmus célja, hogy **optimalizálja a neurális hálózat súlyait** úgy, hogy minimalizálja a hálózat által elkövetett hibát. Az algoritmus két fő lépésből áll:

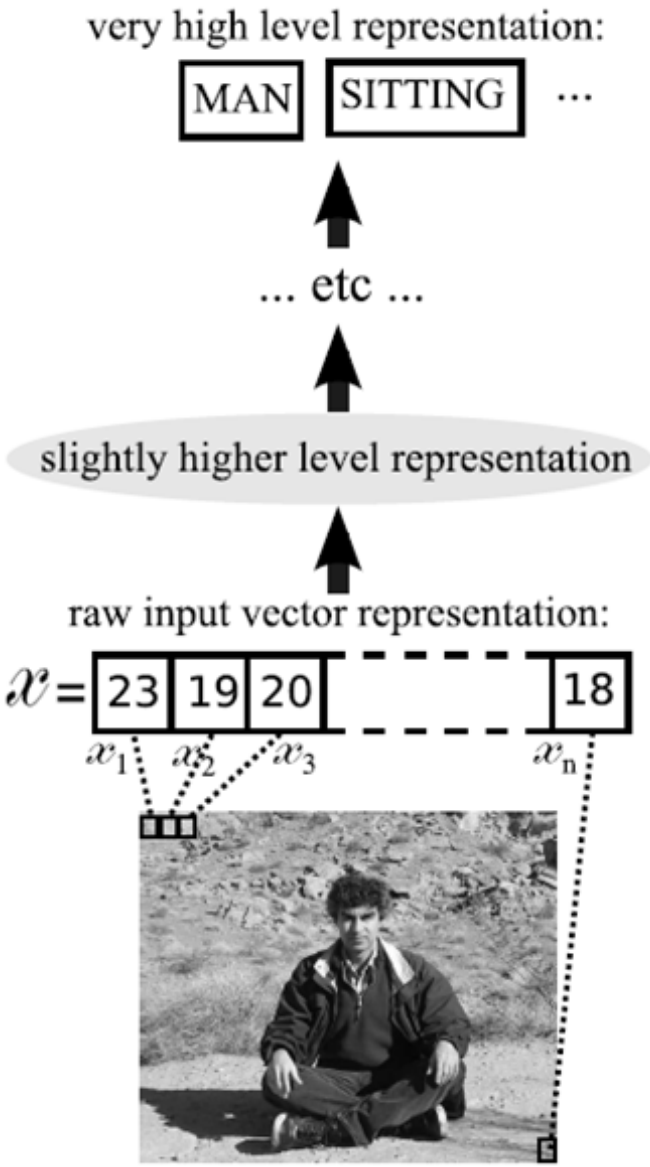
1. **Előre terjesztés (Forward Propagation):**
   - Az adatok áthaladnak a hálózaton, és minden réteg kiszámítja a kimenetét a bemeneti adatok és a súlyok alapján.
   - Az utolsó réteg kimenete a hálózat előrejelzése.
2. **Visszaterjesztés (Backpropagation):**
   - A visszaterjesztés során a hálózat kiszámítja a hibát (a valós és az előrejelzett érték közötti különbséget) és visszaterjeszti ezt a hibát a hálózaton keresztül.
   - A hibát minden rétegben felhasználják a súlyok frissítésére, hogy csökkentsék a hibát. Ez a gradiens csökkenés (**gradient descent**) algoritmus segítségével történik.

A visszaterjesztés algoritmus lehetővé teszi a mély neurális hálózatok **hatékony betanítását**, mivel képes kezelni a hálózatban lévő sok réteg súlyainak optimalizálását.

Az ANN (Annotated Neural Network) tanulási folyamata




Deep neural network
input layer
hidden layer 1
hidden layer 2
hidden layer 3
output layer


very high level representation:
MAN
SITTING
...
... etc ...
slightly higher level representation
raw input vector representation:
x =
23
19
20
18
x1
x2
x3
xn

2. TensorFlow könyvtár alapjai

TensorFlow definíciója:

A TensorFlow egy nyílt forráskódú, skálázható és hatékony **gépi mélytanulási keretrendszer**, amelyet a Google hozott létre. Különböző platformokon működik, és támogatja a mély neurális hálózatok fejlesztését és futtatását.

Jellemzői:

1. Skálázhatóság: Kis mintáktól kezdve a nagy adatbázisokig mindenre alkalmas.
2. Hatékonyság: Optimalizált algoritmusokat használ a gyors tanításra és futtatásra.
3. Támogatott platformok: Mobiltelefonokon, felhőben és böngészőben is használható.
4. Rugalmasság: Támogatja az egyszerű magas szintű API-kat és az alacsony szintű programozást.

TensorFlow hivatalos dokumentáció: A TensorFlow API részletes magyarázata, kódpéldák, és gyakorlatok. https://www.tensorflow.org

Keras dokumentáció: Keras API, amely megkönnyíti a TensorFlow-val való fejlesztést. https://keras.io

Alapfogalmak: Tensorok

A TensorFlow központi elemei a tensorok, amelyek n-dimenziós tömböket vagy mátrixokat jelentenek.

A tensorok az adatokat reprezentálják és tárolják, például:

- Egyetlen szám (skalár)
- Vektorok
- Mátrixok és összetettebb struktúrák

Példa egy tensorra:

```
import tensorflow as tf
pelda_tensor = tf.constant([[1, 2, 3], [4, 5, 6]])
print(pelda_tensor)
```

Alapfogalmak: Gráf-alapú működés

A TensorFlow **számítási gráfokat** használ, amelyek csomópontokból és élekből állnak.

Ez a struktúra meghatározza az adatok áramlását és lehetővé teszi az optimalizációt.

Egyszerű gráf működése:

```
a = tf.constant(3)
b = tf.constant(4)
osszeg = a + b
print(osszeg)
```

Telepítés és ellenőrzés

TensorFlow telepítése: **pip install tensorflow**

Telepítés ellenőrzése:

```
import tensorflow as tf
print(tf.__version__)
```

Egyszerű tensor művelet bemutatása

Az a és b tensorok létrehozása és az **add()** matematikai művelet végrehajtása:

```
import tensorflow as tf
# Tensorok létrehozása
tensor_a = tf.constant([5, 10, 15], name="tensor_a")
tensor_b = tf.constant([2, 4, 6], name="tensor_b")
# Alapvető matematikai művelet
osszeg = tf.add(tensor_a, tensor_b)
# Eredmény kiíratása
print("Eredmény:", osszeg.numpy())
```

Magyarázat:

- A tf.constant segítségével hoztunk létre két bemeneti tensort (tensor_a és tensor_b).
- A tf.add függvény végzi a tensorok elemeinek összeadását.
- A numpy() metódus átalakítja a tensort NumPy tömbbé, hogy a Python kimenetén megjeleníthető legyen.

3. Neurális hálózatok felépítése TensorFlow-val

Alapstruktúra

A neurális hálózatokat **három fő rétegtípus** alkotja, amelyek együttműködnek a bemenetek feldolgozásában és a kimenetek előállításában:

1. Input réteg:

- A bemeneti réteg fogadja az adatokat. Ez határozza meg a modell bemeneti dimenzióit.

  Példa: Egy képfeldolgozó modell bemenete 28x28 pixel lehet, ahol minden pixel intenzitását egy érték képviseli.

2. Rejtett rétegek:

- Ezek a rétegek feldolgozzák a bemenetet a tanulási folyamat során. Több rejtett réteg is lehet.
- Feladatuk az adatban található mintázatok felismerése.

3. Kimeneti réteg:

- Az eredmény előállítása.

  Példa: Ha a modell célja egy képen lévő tárgy osztályozása, a kimeneti réteg a kategóriák valószínűségeit adja vissza.

Aktivációs függvények

Az aktivációs függvények adják meg, hogy az egyes rétegek hogyan alakítsák át a bemenetet a következő réteg számára (ReLU, Sigmoid, Softmax)

TensorFlow API: Neurális hálózat felépítése

A TensorFlow-ban a Sequential API-val a rétegek egymás után adhatók hozzá a modellhez.

1. Sequential modell:

- Egyszerűbb, ha a modell rétegei sorban követik egymást.
- Előnye: Könnyen olvasható és gyorsan fejleszthető.

2. Rétegek hozzáadása:

- Dense (Sűrű) réteg: Minden bemenet kapcsolódik az összes kimenethez.
- Dropout réteg: Meghatározott arányú neuront véletlenszerűen kikapcsol a túltanulás elkerülése érdekében.

Példa: Egyszerű neurális hálózat TensorFlow-val

```
import tensorflow as tf
from tensorflow.keras import Sequential
from tensorflow.keras.layers import Dense, Dropout
# Modell inicializálása
modell = Sequential()
# Bemeneti réteg és első rejtett réteg hozzáadása
modell.add(Dense(128, activation='relu', input_shape=(784,), name="rejtett_reteg_1"))
# Dropout a túltanulás elkerülésére
modell.add(Dropout(0.2, name="dropout_reteg"))
# Második rejtett réteg
modell.add(Dense(64, activation='relu', name="rejtett_reteg_2"))
# Kimeneti réteg (10 osztály esetén)
modell.add(Dense(10, activation='softmax', name="kimeneti_reteg"))
# Modell összegzése
modell.summary()
```

Magyarázat:

- **Dense:**A rejtett rétegek neuronjainak számát 128 és 64 értékre állítottuk be. Az aktivációs függvény a ReLU. Az input_shape az első rétegnél a bemenet dimenzióját határozza meg (784, például lapított képek).
- **Dropout:** A réteg kikapcsolja a neuronok 20%-át minden tanítási iterációban.
- **Softmax:** A kimeneti réteg az osztályozásért felelős, a softmax aktivációs függvény pedig az osztályok valószínűségét adja meg.

Modell architektúrák, példák

1. Egyszerű osztályozó modell:

- Alkalmazási terület: Képosztályozás (MNIST).
- Rétegek: Bemeneti réteg, két Dense réteg ReLU aktivációval,
  Kimeneti réteg Softmax aktivációval.

2. Többrétegű hálózat Dropout-tal:

- Alkalmazási terület: Túltanulás elkerülése nagy adathalmazokon.
- Rétegek: Hasonló az előzőhöz, de Dropout rétegek a rejtett rétegek után.

3. Regressionális modell:

- Kimeneti réteg: Aktivációs függvény nélkül (lineáris kimenet).
- Alkalmazási terület: Ingatlanár becslés, időjárás előrejelzés.

Példa: Regresszionális modell

```
# Regressionális modell létrehozása
modell_regresszio = Sequential()
# Rejtett rétegek
modell_regresszio.add(Dense(64, activation='relu', input_shape=(10,),
name="rejtett_reteg_1"))
modell_regresszio.add(Dense(32, activation='relu', name="rejtett_reteg_2"))
# Kimeneti réteg (lineáris aktiváció)
modell_regresszio.add(Dense(1, name="kimeneti_reteg"))
# Modell összegzése
modell_regresszio.summary()
```

4. Egyszerű neurális hálózat megvalósítása

Egy egyszerű modellt hozunk létre TensorFlow segítségével, amely bemutatja az adatfeldolgozás, a modell felépítésének és paraméterezésének alapjait.
Egyszerű neurális hálózatot készítünk kép- vagy szövegfeldolgozási feladatokra.

Feladat: Kép- vagy szövegfeldolgozó neurális hálózat létrehozása.

Példánkban az MNIST adatbázis segítségével egy egyszerű osztályozó modellt készítünk, amely a kézzel írt számokat (0–9) azonosítja.

Adat előkészítése: Az adatok betöltése, normalizálása és előkészítése az edzéshez.

```
import tensorflow as tf
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical
# MNIST adatok betöltése
(bemeneti_adatok, cimkek), (teszt_adatok, teszt_cimkek) = mnist.load_data()
# Adatok átméretezése és normalizálása
bemeneti_adatok = bemeneti_adatok.reshape((-1, 28 * 28)).astype('float32') / 255
teszt_adatok = teszt_adatok.reshape((-1, 28 * 28)).astype('float32') / 255
# Címkék one-hot kódolása
cimkek = to_categorical(cimkek, 10)
teszt_cimkek = to_categorical(teszt_cimkek, 10)
print("Adat előkészítve. Bemeneti adatok alakja:", bemeneti_adatok.shape)
```

Magyarázat:

1. Az MNIST adatbázis tartalmazza a kézzel írt számok 28x28 pixeles képeit és a hozzájuk tartozó címkéket.
2. A képeket lapítottuk (reshape), hogy egy 784 méretű vektorként kezelhetők legyenek.
3. Normalizálás: Az értékeket 0 és 1 közé skáláztuk (/ 255), így a modell gyorsabban tanul.
4. A címkék one-hot kódolása biztosítja, hogy a kimeneti osztályok megfelelően reprezentáltak legyenek.

Modell létrehozása: Egy egyszerű, teljesen összekötött neurális hálózatot hozunk létre.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# Modell inicializálása
modell = Sequential()
# Bemeneti réteg és rejtett réteg hozzáadása
modell.add(Dense(128, activation='relu', input_shape=(784,), name="rejtett_reteg_1"))
# Második rejtett réteg hozzáadása
modell.add(Dense(64, activation='relu', name="rejtett_reteg_2"))
# Kimeneti réteg hozzáadása
modell.add(Dense(10, activation='softmax', name="kimeneti_reteg"))
# Modell összegzése
modell.summary()
```

Magyarázat:

1. Dense réteg: Az első réteg 128 neuront tartalmaz, ReLU aktivációs függvénnyel. Az input_shape 784, amely megfelel az MNIST adatbázis lapított bemeneti képeinek. A második rejtett réteg 64 neuront tartalmaz.
A kimeneti réteg 10 neuront tartalmaz, a softmax aktiváció a 10 osztály valószínűségi eloszlását adja meg.

2. modell.summary(): A modell struktúráját nyomtatja ki, beleértve a rétegek számát és paramétereiket.

Paraméterek inicializálása és konfigurálása

A modell fordítása, az optimizációs függvény, veszteségfüggvény és metrikák beállítása.

```
# Modell fordítása
modell.compile(
   optimizer='adam',          # Optimalizációs algoritmus
   loss='categorical_crossentropy’,     # Veszteségfüggvény többosztályos osztályozáshoz
   metrics=['accuracy']       # Teljesítménymetrika
)
print("Modell inicializálva és fordítva.")
```

Magyarázat:

1. **Optimizációs algoritmus:** Az Adam optimizáló hatékony és gyorsan konvergál.
2. **Veszteségfüggvény:** A többosztályos osztályozáshoz a categorical_crossentropy a legmegfelelőbb.
3. **Metrikák:** Az accuracy mutatja meg, hogy a modell milyen pontossággal végzi az osztályozást.

Modell tanítása: A modell tanítása az előkészített adatokon.

```
# Modell tanítása
modell.fit(
   bemeneti_adatok, cimkek,      # Edzési adatok és címkék
   epochs=10,                    # Iterációk száma
   batch_size=32,                # Batch méret
   validation_data=(teszt_adatok, teszt_cimkek)  # Validációs adatok
)
```

Magyarázat:

- **epochs**: Az edzési ciklusok száma.
- **batch_size:** Adatszeletek mérete, amelyeken egyszerre dolgozik a modell.
- **validation_data:** A tanulási folyamat közben a modell validációja.

5. Modell betanítása és kiértékelése TensorFlow-val

Bemutatjuk a modell tanításának és kiértékelésének **teljes folyamatát** a TensorFlow keretrendszer segítségével: a tanulási folyamat konfigurálása (optimizer, loss function), az adatok kezelése (batch-ek és epoch-ok), valamint a modell teljesítményének értékelése pontosság és veszteség alapján.

Az optimalizációs algoritmusok, a veszteségfüggvények és a metrikák helyes kiválasztása alapvetően meghatározza a modell teljesítményét. A tanulási görbék vizualizációja segíti a tanítási folyamat monitorozását és finomhangolását.

A hiperparaméterek módosításával, az optimalizációs algoritmusok finom-hangolásával és a tanulási sebesség beállításával jelentősen javítható a modell teljesítménye.

1. Modell tanítása

1.1. Optimizer és loss function beállítása

Az optimalizáló algoritmus (optimizer) és a veszteségfüggvény (loss function) szerepe a modell tanításában:

- **Optimizer:** Frissíti a modell paramétereit a gradiens-alapú optimalizáció segítségével.
  - Példa: 'Adam', 'SGD' (Stochastic Gradient Descent).
- **Loss function (veszteségfüggvény):** Az eltérést méri a modell által generált kimenet és a célérték között.
  - Példa: 'CategoricalCrossentropy', 'MeanSquaredError'.

```
modell.compile(
  optimizer='adam’, # Optimalizációs algoritmus
  loss='categorical_crossentropy’, # Veszteségfüggvény többosztályos osztályozáshoz
  metrics=['accuracy’] # Teljesítménymetrika
)
```

Epochs és batch size jelentősége

- **Epoch:** Az adathalmaz teljes feldolgozása egyszer (előre- és visszaterjesztés).
- **Batch size:** Az adatok kisebb csoportokra (batch-ekre) osztása, amelyeken a modell egyszerre dolgozik.
  - **Nagy batch size:** Stabilabb gradiens-frissítések, de nagyobb memóriaigény.
  - **Kis batch size:** Gyorsabb iterációk, de több zaj a tanulási folyamatban.

Példa: Modell tanítása epoch-okkal és batch-ekkel

```
eredmeny = modell.fit(
   x_train,    # Bemeneti adatok
   y_train,    # Célértékek
   epochs=10,      # Tanítási ciklusok száma
   batch_size=32,  # Batch méret
   validation_data=(x_val, y_val)     # Validációs adatok
)
```

2. Modell kiértékelése

2.1. Pontosság és veszteség mérése

A modell kiértékeléséhez fontos metrikák:

- **Pontosság (accuracy):** Az osztályozás pontossága, az összes helyes predikció aránya.
- **Veszteség (loss):** A kimenetek és a célértékek közötti eltérés.

```
teszt_veszteseg, teszt_pontossag = modell.evaluate(x_test, y_test)
print(f"Teszthalmaz vesztesége: {teszt_veszteseg}")
print(f"Teszthalmaz pontossága: {teszt_pontossag}")
```

2.2. Vizualizáció (tanulási görbék)

A tanulási folyamat követéséhez érdemes a veszteség és pontosság változását vizualizálni az egyes epoch-ok során.

Tanulási görbék rajzolása:

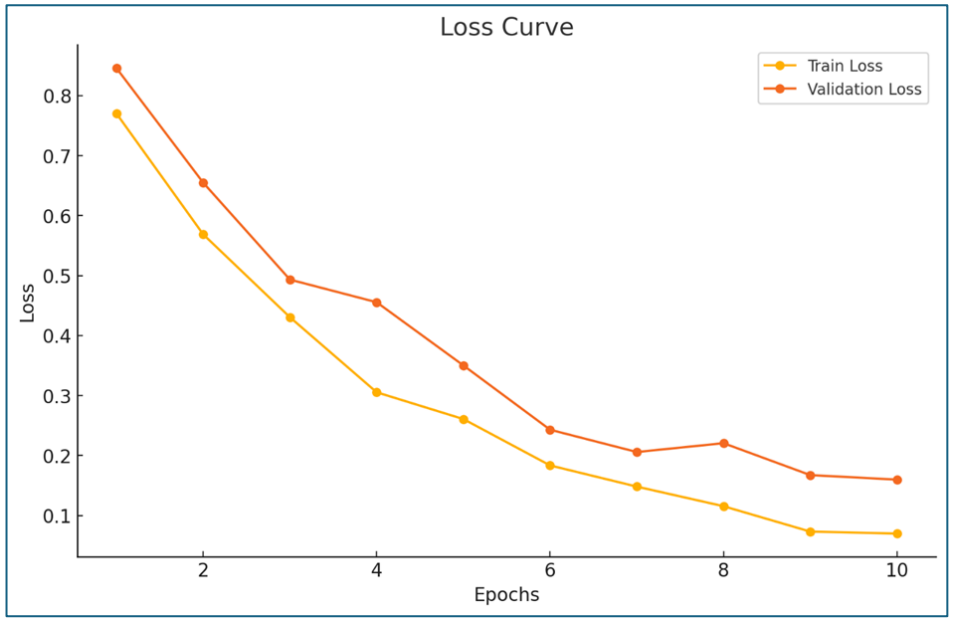
```
import matplotlib.pyplot as plt
# Tanulási görbék adatainak elérése
train_loss = eredmeny.history['loss']
val_loss = eredmeny.history['val_loss']
train_accuracy = eredmeny.history['accuracy']
val_accuracy = eredmeny.history['val_accuracy']
# Veszteség ábrázolása
plt.figure()
plt.plot(train_loss, label='Train Loss')
plt.plot(val_loss, label='Validation Loss')
plt.title('Loss Curve')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.show()
```

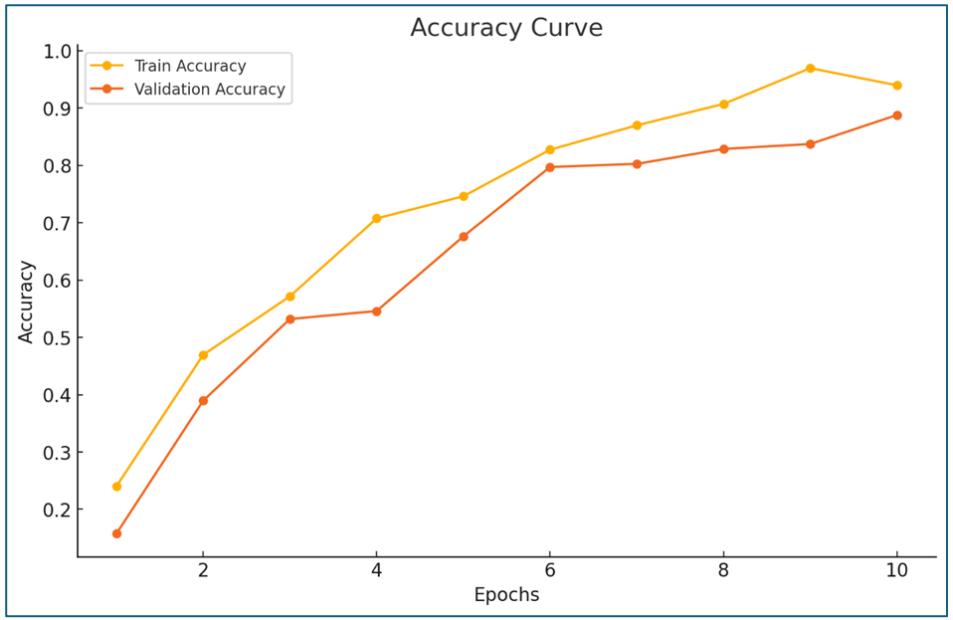
```
# Pontosság ábrázolása
plt.figure()
plt.plot(train_accuracy, label='Train Accuracy')
plt.plot(val_accuracy, label='Validation Accuracy')
plt.title('Accuracy Curve')
plt.xlabel('Epochs')
plt.ylabel('Accuracy')
plt.legend()
plt.show()
```

Két példagörbe a tanulási folyamat bemutatására

1. **Loss Curve (Veszteség görbe):** A veszteség alakulását mutatja a tanítási (train) és validációs (validation) adathalmazon az egyes epoch-ok során. Az ideális görbe esetén mindkét görbe csökken az epoch-ok előrehaladtával.
2. **Accuracy Curve (Pontosság görbe):** A pontosság (accuracy) változását ábrázolja a tanítási és validációs adatokon. Az ideális görbe esetén mindkét pontosság növekszik.

Ezek a görbék bemutatják, hogyan tanul a modell és lehet észlelni a problémákat, mint például a túltanulást (overfitting) vagy az alultanulást (underfitting).





3. Egyszerű modell tanítás és validáció

Egy teljes tanítási és validációs folyamat egy egyszerű modell példáján:

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
# Adat előkészítése
(x_train, y_train), (x_test, y_test) = tf.keras.datasets.mnist.load_data()
x_train = x_train.reshape((-1, 28 * 28)).astype('float32') / 255
x_test = x_test.reshape((-1, 28 * 28)).astype('float32') / 255
y_train = tf.keras.utils.to_categorical(y_train, 10)
y_test = tf.keras.utils.to_categorical(y_test, 10)
# Modell létrehozása
modell = Sequential([
   Dense(128, activation='relu', input_shape=(784,)),
   Dense(64, activation='relu'),
   Dense(10, activation='softmax')
])
# Modell fordítása
modell.compile(
   optimizer='adam',
   loss='categorical_crossentropy',
   metrics=['accuracy']
)
```

```
# Modell tanítása
eredmeny = modell.fit(
   x_train, y_train,
   epochs=10,
   batch_size=32,
   validation_split=0.2
)
# Modell kiértékelése
teszt_veszteseg, teszt_pontossag =
modell.evaluate(x_test, y_test)
print(f"Teszthalmaz vesztesége: {teszt_veszteseg}")
print(f"Teszthalmaz pontossága: {teszt_pontossag}")
```

6. TensorFlow modellek generálása és optimalizálása

1. Modell generálása: Egy neurális hálózat létrehozása és fordítása TensorFlow-ban, amely képes **többosztályos osztályozási problémák** megoldására. Ebben a példában egy alap modell készül, amely a bemeneti adatokat **több rétegen keresztül** dolgozza fel.

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
# Modell inicializálása
modell = Sequential()
# Bemeneti és első rejtett réteg
modell.add(Dense(128, activation='relu', input_shape=(784,), name="rejtett_reteg_1"))
# Második rejtett réteg
modell.add(Dense(64, activation='relu', name="rejtett_reteg_2"))
# Dropout a túltanulás elkerülésére
modell.add(Dropout(0.3, name="dropout_reteg"))
# Kimeneti réteg
modell.add(Dense(10, activation='softmax', name="kimeneti_reteg"))
# Modell fordítása
modell.compile(
   optimizer='adam',
   loss='categorical_crossentropy',
   metrics=['accuracy’] )
# Modell összegzése
modell.summary()
```

Magyarázat:

1. **Sequential modell:** A modellelemek egymás után kerülnek hozzáadásra.
2. **Dense rétegek:** Rejtett rétegek és a kimeneti réteg létrehozása különböző neuronokkal.
3. **Dropout réteg:** A túltanulás csökkentése érdekében egyes neuronokat véletlenszerűen kikapcsolunk.
4. **Optimális paraméterek beállítása:** Adam optimizálót használunk, amely gyors konvergenciát biztosít.

2. Modell optimalizálása

Az optimalizálás során különböző hiperparaméterek módosításával érhetjük el a legjobb teljesítményt.

2.1. Hiperparaméterek módosítása: Az alábbi példában a batch méretet és az epoch számot változtatjuk.

```
# Modell tanítása módosított paraméterekkel
eredmeny = modell.fit(
   bemeneti_adatok, cimkek,
   epochs=20,                # Több iteráció
   batch_size=64,              # Nagyobb batch méret
   validation_data=(teszt_adatok, teszt_cimkek)
)
# Eredmények kiértékelése
print(f"Végső pontosság: {eredmeny.history['accuracy'][-1]}")
print(f"Validációs pontosság: {eredmeny.history['val_accuracy'][-1]}")
```

Magyarázat:

1. **Epochok:** A tanítási ciklusok száma. Több epoch javíthatja a modell teljesítményét, de túlzott számú epoch túltanulást okozhat.
2. **Batch méret:** A tanítási adatok részekre osztása. Nagyobb batch méret stabilabb gradienst biztosít.

2.2. Tanulási sebesség (learning rate) módosítása:

A tanulási sebesség módosítása optimalizációs stratégiaként hatékony lehet.

```
from tensorflow.keras.optimizers import Adam
# Új optimizáló alacsonyabb tanulási sebességgel
adam_optimizer = Adam(learning_rate=0.0005)
# Modell újrafordítása
modell.compile(
   optimizer=adam_optimizer,
   loss='categorical_crossentropy',
   metrics=['accuracy']
)
# Modell újra tanítása
eredmeny = modell.fit(
   bemeneti_adatok, cimkek,
   epochs=15,
   batch_size=32,
   validation_data=(teszt_adatok, teszt_cimkek)
)
```

Magyarázat:

1. **Adam optimizáló:**
   A **learning_rate** paraméter segítségével finomhangoljuk a modell tanulási sebességét.
2. **Validációs adatok** segítségével figyelemmel kísérhetjük, hogy a modell hogyan teljesít az ismeretlen adatokon.

3. Modell kiértékelése

A tanítási folyamat végén érdemes a modellt kiértékelni tesztadatokon.

```
# Modell kiértékelése teszt adatokon
teszt_veszteseg, teszt_pontossag = modell.evaluate(teszt_adatok, teszt_cimkek)
print(f"Teszthalmaz vesztesége: {teszt_veszteseg}")
print(f"Teszthalmaz pontossága: {teszt_pontossag}")
```

4. Modell mentése és betöltése

A jól működő modellt érdemes elmenteni a későbbi használathoz.

```
# Modell mentése
modell.save("mentett_modell.h5")
print("Modell mentve.")
# Modell betöltése
uj_modell = tf.keras.models.load_model("mentett_modell.h5")
print("Modell betöltve.")
```

Magyarázat:

1. **Mentés:** A save() metódussal a modellt egy .h5 fájlba mentjük.
2. **Betöltés:** A load_model() segítségével a mentett modellt visszatölthetjük későbbi használatra.

További információk

Online kurzusok

- **Coursera:**
  - Machine Learning Specialization (Andrew Ng)
  - Deep Learning Specialization (Andrew Ng)
- **Udemy:**
  - TensorFlow Developer Professional Certificate
  - Practical Deep Learning with TensorFlow 2.0
- **edX:**
  - Google TensorFlow Developer Certificate Preparation

Könyvek

- **Deep Learning with Python (F. Chollet):** Keras és TensorFlow alapú példák.
- **Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow (A. Géron):** Gyakorlati megközelítés a gépi tanulás alapjaitól a mélytanulásig.
- **Deep Learning (I. Goodfellow, Y. Bengio, A. Courville):** Az elméleti alapok megértéséhez.

Hasznos eszközök

- **Kaggle:** https://www.kaggle.com Ingyenes GPU-k, versenyek és valós adathalmazok.
- **Google Colab:** https://colab.research.google.com Online Jupyter notebook TensorFlow GPU.
- **GitHub:** https://github.com/tensorflow Nyílt forráskódú TensorFlow projektek, példák.

7. PyTorch: Mélytanulási keretrendszer

PyTorch definíciója:

A PyTorch egy nyílt forráskódú **mélytanulási keretrendszer**, amelyet a Facebook AI Research (FAIR) fejlesztett.

A keretrendszer támogatja a tensor-alapú számításokat és a dinamikus számítási gráfok használatát.

Különösen alkalmas kutatási prototípusok készítésére és kisebb projektek gyors fejlesztésére.

Előnyök: Rugalmasság, gyors prototípus készítés. Python-stílus, intuitív felhasználás. Számos nyílt forráskódú projekt. Fejlett eszközök: TorchVision, TorchText, TorchAudio.

Hátrányok: Nagy skálán nem mindig hatékony. Mobil és beágyazott rendszerek támogatása korlátozott.

Fő jellemzők:

- **Dinamikus számítási gráfok:** A gráf dinamikusan épül fel a kód futtatása közben.
- **Tensor alapú számítások GPU támogatással:**
  - GPU-alapú gyorsítás lehetősége, amely jelentősen növeli a számítási sebességet.
  - A tensorok a NumPy-hoz hasonlóak, de ezek **CUDA-támogatás**t is nyújtanak a GPU-kon történő számításokhoz.
- **Python-integráció:**
  - Könnyen kombinálható Python-könyvtárakkal (NumPy, Pandas).
- **Fejlett automatikus deriválás** (autograd):
  - Megkönnyíti a gradiens-alapú optimalizációt.
- **Kutatási és gyakorlati alkalmazás:**
  - Ideális prototípus készítéshez és ipari alkalmazásokhoz.

PyTorch alapműködése:

1. Tensor létrehozása:

```
import torch
# Tensor létrehozása
tensor_a = torch.tensor([1, 2, 3])
print(tensor_a)
# Tensor átalakítása GPU-ra
if torch.cuda.is_available():
   tensor_a = tensor_a.to('cuda')
   print("Tensor GPU-n:", tensor_a)
```

A PyTorch tensorok hasonlóak a NumPy tömbökhöz, de GPU támogatást is nyújtanak.

2. Egyszerű neurális hálózat létrehozása:

```
import torch
import torch.nn as nn
import torch.optim as optim
class EgyszeruHalo(nn.Module):   # Egyszerű neurális hálózat
   def __init__(self):
      super(EgyszeruHalo, self).__init__()
      self.fc1 = nn.Linear(784, 128)
      self.relu = nn.ReLU()
      self.fc2 = nn.Linear(128, 10)
   def forward(self, x):
      x = self.fc1(x)
      x = self.relu(x)
      x = self.fc2(x)
      return x
modell = EgyszeruHalo()     # Modell inicializálása
# Optimizer és veszteségfüggvény
optimizer = optim.Adam(modell.parameters(), lr=0.001)
veszteseg_fv = nn.CrossEntropyLoss()
```

3. Modell tanítása, MNIST

Egy egyszerű osztályozási probléma modelljének tanítása.

```
# Dummy adatok
adatok = torch.randn(64, 784)  # 64 mintával, mindegyik 784 bemeneti értékkel
cimkek = torch.randint(0, 10, (64,))  # Véletlenszerű címkék 0-9 között
# Tanítási ciklus
for epoch in range(10):  # 10 tanulási ciklus
   optimizer.zero_grad()      # Gradiens nullázása
   kimenet = modell(adatok)    # Előre haladás
   veszteseg = veszteseg_fv(kimenet, cimkek)  # Veszteség kiszámítása
   veszteseg.backward()       # Hátrafelé haladás (gradiens számítás)
   optimizer.step()          # Súlyok frissítése
   print(f"Epoch {epoch+1}, Veszteség: {veszteseg.item()}")
```

Példa: Egyszerű neurális hálózat létrehozása

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
# Példa adatok létrehozása
np.random.seed(42)
x = np.random.rand(100, 1).astype(np.float32)
y = 4 + 3 * x + np.random.randn(100, 1).astype(np.float32)
# Adatok PyTorch tensorokká alakítása
x_tensor = torch.from_numpy(x)
y_tensor = torch.from_numpy(y)
# Egyszerű neurális hálózat létrehozása
class SimpleNN(nn.Module):
    def __init__(self):
        super(SimpleNN, self).__init__()
        self.linear = nn.Linear(1, 1)
    def forward(self, x):
        return self.linear(x)
model = SimpleNN()
# Veszteségfüggvény és optimalizáló
criterion = nn.MSELoss()
optimizer = optim.SGD(model.parameters(), lr=0.01)
```

```
# Modell betanítása
num_epochs = 1000
for epoch in range(num_epochs):
    model.train()
    optimizer.zero_grad()
    outputs = model(x_tensor)
    loss = criterion(outputs, y_tensor)
    loss.backward()
    optimizer.step()
    if (epoch+1) % 100 == 0:
        print(f'Epoch [{epoch+1}/{num_epochs}], Loss:
{loss.item():.4f}')
# Előrejelzések készítése
model.eval()
with torch.no_grad():
    y_pred = model(x_tensor)
# Eredmények megjelenítése
import matplotlib.pyplot as plt
plt.scatter(x, y, color='blue', label='Valós adatok')
plt.plot(x, y_pred.numpy(), color='red', label='Előrejelzések')
plt.xlabel('X értékek')
plt.ylabel('Y értékek')
plt.title('Neurális hálózat PyTorch segítségével')
plt.legend()
plt.show()
```

Magyarázat:

1. **Adatok létrehozása:**
   - Példa adatok létrehozása, ahol (x) és (y) lineáris kapcsolatban vannak (némi zajjal).
2. **Adatok PyTorch tensorokká alakítása:**
   - Az adatok PyTorch tensorokká alakítása a torch.from_numpy függvény segítségével.
3. **Egyszerű neurális hálózat létrehozása:**
   - Egy egyszerű neurális hálózat létrehozása egy lineáris réteggel.
4. **Veszteségfüggvény és optimalizáló:**
   - A veszteségfüggvény (Mean Squared Error) és az optimalizáló (Stochastic Gradient Descent) beállítása.
5. **Modell betanítása:**
   - A modell betanítása 1000 epoch-on keresztül. Minden epoch-ban kiszámítjuk a veszteséget, visszaterjesztjük a hibát, és frissítjük a súlyokat.
6. **Előrejelzések készítése:**
   - Előrejelzések készítése a betanított modell segítségével.
7. **Eredmények megjelenítése:**
   - Az adatok és az előrejelzések megjelenítése egy szórásdiagramon.

További eszközök

- TorchVision: Előre betanított modellek, képfeldolgozó eszközök.
- TorchText: Természetes nyelv feldolgozási eszközök.
- TorchAudio: Hangfeldolgozási eszköztár.
- Hivatalos dokumentáció: https://pytorch.org/docs/
- Fast.ai kurzus: https://www.fast.ai
- Könyv: Deep Learning with PyTorch.

Hivatalos dokumentáció: https://pytorch.org/docs/

Könyvek:

- "Deep Learning with PyTorch" (Eli Stevens, Luca Antiga, Thomas Viehmann)
- "Programming PyTorch for Deep Learning" (Ian Pointer)

Online kurzusok:

- Fast.ai PyTorch kurzus (fast.ai)
- PyTorch for Deep Learning (Coursera, Udemy)

Összegzés

1. Deep Learning fogalmak megbeszélése
2. A TensorFlow könyvtár alapjainak bemutatása
3. Neurális hálózatok felépítése TensorFlow-val
4. Egyszerű neurális hálózat megvalósítása
5. Modell betanítása és kiértékelése TensorFlow-val
6. TensorFlow modellek generálása és optimalizálása
7. PyTorch: Mélytanulási keretrendszer ismertetése

Konzultáció

Minden héten csütörtökön 18:00 – 20:00

Köszönöm a figyelmet!