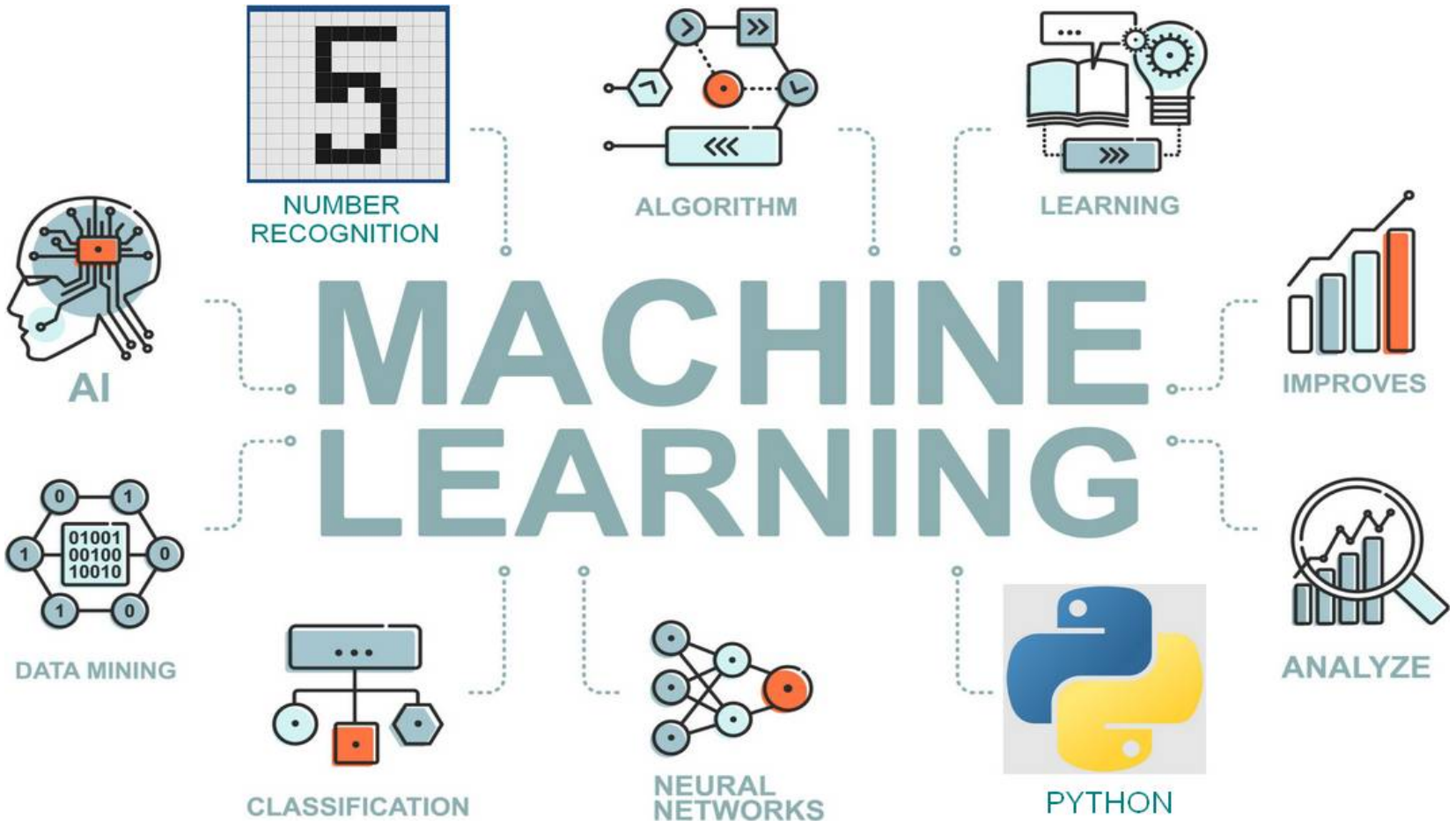
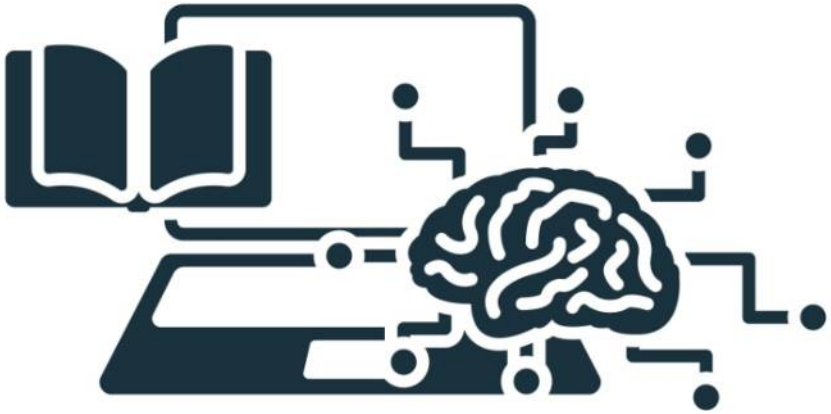


A gépi tanulás alapjai Pythonnal

Felhasznált és ajánlott irodalom

- François Chollet: Deep Learning with Python
  Deep Learning with Python - Notebooks
  Deep-Learning-with-Python-HUN (ford. Nagy Sándor)
- R. J Hyndman, G. Athanasopoulos: Forecasting: Principles and Practice
- Stuart Russell, Peter Norvig: Artificial Intelligence - A Modern Approach
- Hubai Zsolt: Adatbányászat és gépi tanulás a gyakorlatban
- Fazekas István: Neurális hálózatok
- Jelasity Márk: Mesterséges Intelligencia



Gépi tanulás (Machine Learning)

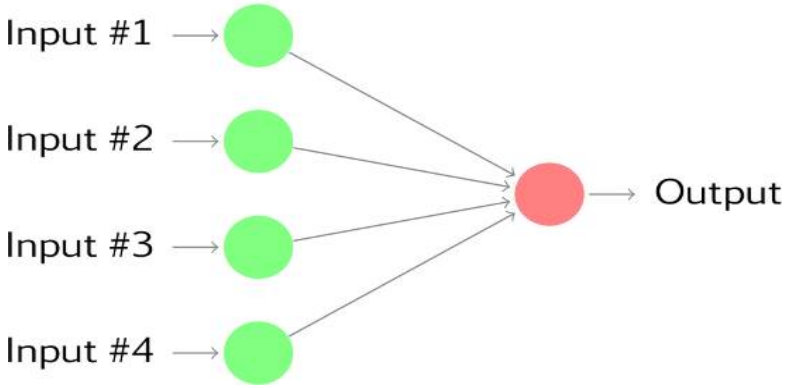
- **A hagyományos felfogásban programozott** számítógép nem képes eredetiséget létrehozni, csak azokat a feladatokat tudja végrehajtani, amelyeket utasításba adunk neki.
- **A gépi tanulás** új korszakot nyitott a programozásban: ahelyett, hogy mi írnánk meg az összes szabályt, maga a gép tanulja meg azokat az adatokból
- Bár a gépi tanulás szorosan kapcsolódik a matematikai statisztikához, különösen az adatelemzés területén, sokszor teljesen eltérő módszereket alkalmaz, mivel nagy és összetett adathalmazok feldolgozására optimalizált. A gépi tanulás kevés matematikai elméletet alkalmaz, inkább mérnöki szemléletű gyakorlati tudományág, ahol az ötleteket többnyire empirikusan, kísérletekkel igazolják
- A gépi tanuláshoz három dolog szükséges:
  - **Bemeneti adatok** – képek, szövegek, hangfájlok, amelyeket a gép feldolgoz
  - **Elvárt kimenetek példái** – emberek által előre címkézett adatok (pl. „kutya” vagy „macska” egy képen)
  - **Teljesítménymérés** – A modell pontosságának értékelése, amely visszacsatolás révén segít a tanulást, a paraméterek finomhangolását és irányt mutat a továbbfejlesztéshez

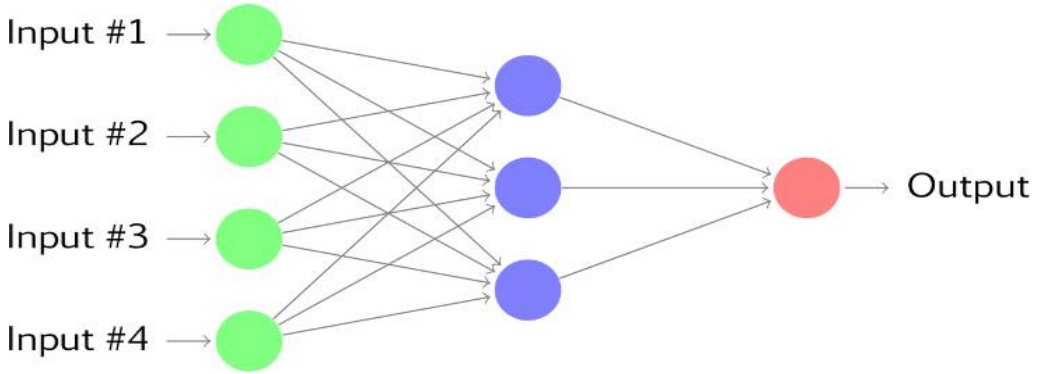
Mélytanulás (Deep Learning)

- **A mélytanulás** a gépi tanulás egyik speciális területe, amely az adatokból történő reprezentációk tanulására összpontosít, méghozzá egymásra épülő rétegek segítségével. A ***„mély”*** kifejezés nem a módszer mélyebb megértésére utal, hanem arra, hogy a modellek több, egymásra épülő réteget használnak az adatok feldolgozására. A rétegek száma határozza meg a modell mélységét.
- **A mélytanulásban** ezek a rétegek szinte mindig **neurális hálózatok** segítségével tanulnak, amelyek ténylegesen egymásra épülő rétegekből állnak. Bár a „neurális hálózat” kifejezés a neurobiológiára utal, a mélytanulási modellek nem az agy működését utánozzák. A mélytanulás valójában egy **matematikai keretrendszer,** amely az adatokból történő reprezentációk tanulására szolgál
- **A mélytanulási modellekben** az egyes rétegek működését **a súlyok** (weights) határozzák meg, ezek paraméterezik az adott réteg által végrehajtott transzformációt
- **A tanulás** során a hálózat folyamatos visszacsatolás és finomhangolás révén megtalálja az optimális súlyfaktorokat, melyekkel a bemeneti adatok helyesen leképezhetők a kívánt kimenetekre. A súlyok optimalizálása **gradiens alapú** algoritmusokkal történik, például visszaterjesztés (backpropagation) és gradiens csökkenés (gradient descent) segítségével

Neurális hálózatok felépítése

- Egy **neurális hálózat** rétegekbe rendezett „neuronok” hálózata. A bemenetek alkotják az alsó réteget, a kimenetek pedig a felső réteget. Lehetnek közbenső rétegek is, amelyek „rejtett neuronokat” tartalmaznak
- A legegyszerűbb neurális hálózat nem tartalmaz rejtett rétegeket, és lényegében egy lineáris regresszióval egyenértékű.
- Amint egy közbenső réteg (hidden layer) hozzáadódik, a hálózat nemlineárissá válik. Egy többrétegű előrecsatolt hálózat (multilayer feed-forward network) esetében minden réteg az előző rétegből kapja a bemeneteket, amelyeket súlyozottan összegzödnek, majd egy nemlineáris aktivációs függvénnyel módosulnak, mielőtt a következő rétegbe jutnak.



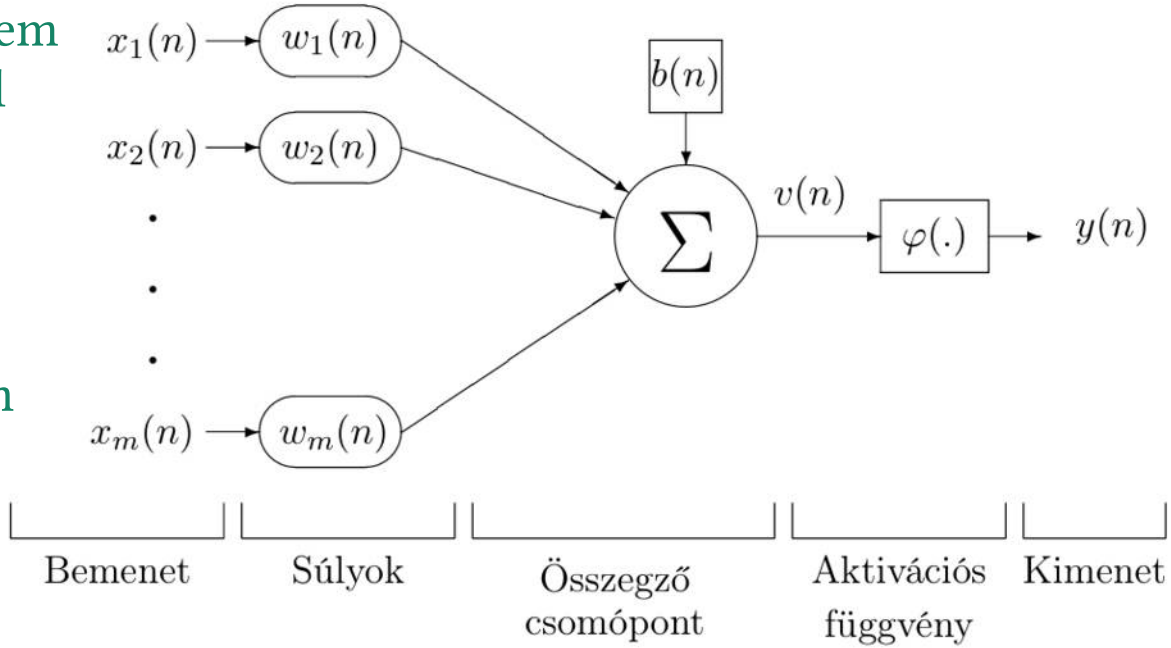


A neuron sémája

- Az ábrán látható egy neuron (perceptron) általános sémája
- $\boldsymbol{x}(n)=(x_i(n),\ldots,x_m(n))^T$ az n-edik időpontban érkező, m számból álló bemenő jel
- Az igazi $w_1,\ldots,w_m$ súlyok értéke nem ismertek, ezek meghatározása a cél, amit fokozatos közelítéssel érünk el a betanítás során
- A b torzítás (bias) értéke szintén nem ismert, ennek meghatározása is cél
- Az összegző csomópont a bemenő adatokat súlyozottan összegzi

$$v(n) = b(n) + \sum_{i=1}^{m} w_i(n)x_i(n)$$

- $\varphi(n)$ aktivációs, vagy egyszerűen transzfer függvény, amit nekünk magunknak kell megadnunk
- A kimenet az $\mathbf{x}(n)$ -hez rendelt $y(n) = \varphi(v(n))$ érték

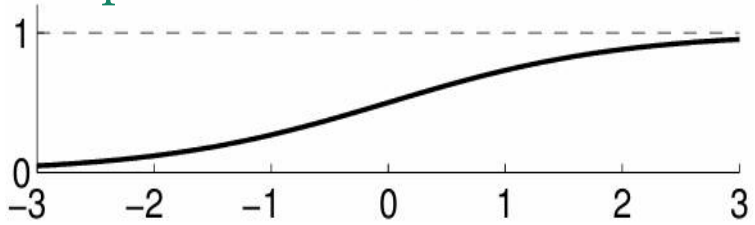


Forrás: Fazekas István: Neurális hálózatok

Aktivációs függvények

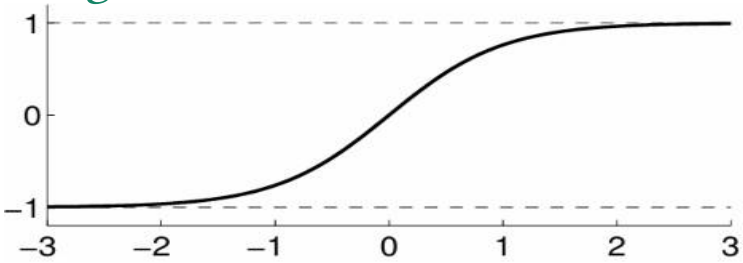
- **Logisztikus függvény** – Gyakran használt osztályozási problémákban, mert az értékei 0 és 1 között mozognak

$$\varphi(x) = \frac{1}{1 + \exp(-ax)}$$



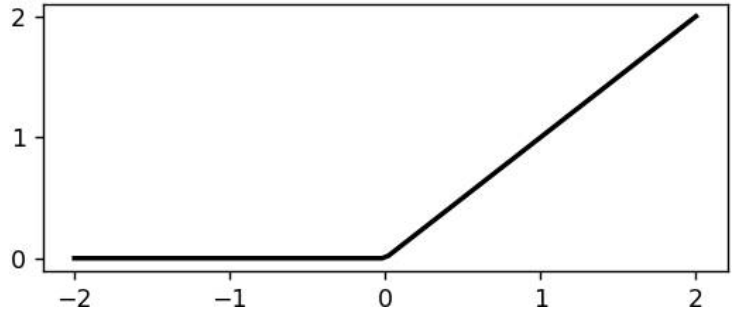
- **Tangens hiperbolikus** (tanh) függvény – Hasonló a logisztikushoz, de –1 és 1 között mozog, így gyakran jobb teljesítményt nyújt

$$\varphi(x) = \frac{2}{1 + \exp(-2x)} - 1 = \tanh(x)$$



- **ReLU (Rectified Linear Unit):** folytonos függvény, amely 0-t ad, ha x < 0, de lineárisan növekszik, ha x > 0. Ez lehetővé teszi a gradiens-alapú optimalizálást, ezért sokkal hatékonyabb mélytanulási modellekben

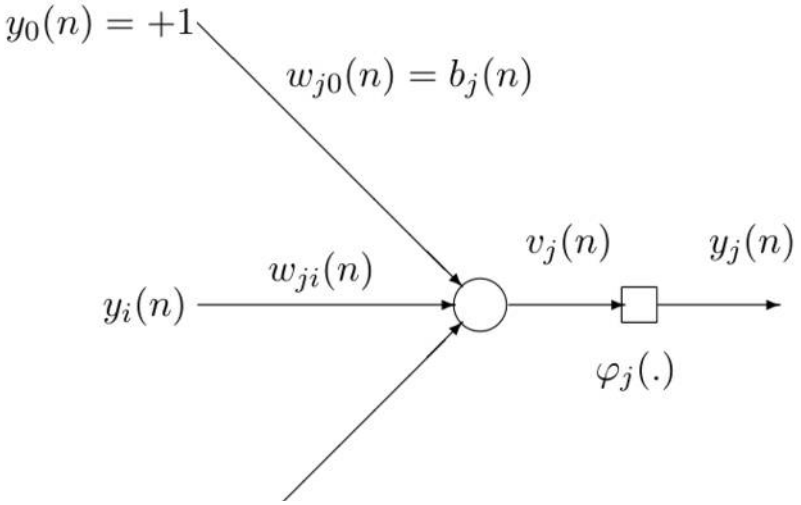
$$\varphi(x) = max(0, x)$$



Forrás: Fazekas István: Neurális hálózatok (logisztikus és tanh)

A neurális háló tanítása gradiens módszerrel

- **A tanítás fő lépései:**
  - Megadjuk a kezdeti súlyokat
  - A bemeneti jelet (azaz a tanító pontot) végigáramoltatjuk a hálózaton, de a súlyokat nem változtatjuk meg
  - Az így kapott kimeneti jelet összevetjük a tényleges kimeneti jellel
  - A hibát visszaáramoltatjuk a hálózaton, súlyokat pedig megváltoztatjuk a hiba csökkentése érdekében
  - Az ε függvény w_{ji} szerinti parciális deriváltjait az **error back-propagation** módszer rekurzíve számítja ki



$$\mathcal{E}(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n) = \frac{1}{2}\sum_{j \in C}(d_j(n) - y_j(n))^2$$

$$w_{ji}(n+1) - w_{ji}(n) = \Delta w_{ji}(n) = -\eta \frac{\partial \mathcal{E}(n)}{\partial w_{ji}(n)}$$

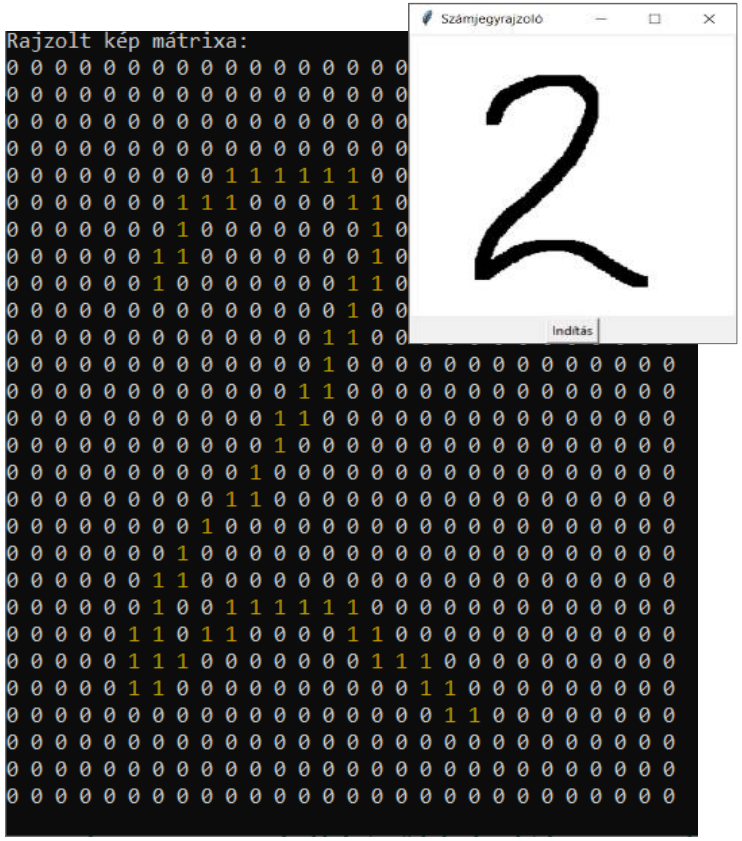
Forrás: Fazekas István: Neurális hálózatok

Számjegyfelismerő programot készítünk

- A cél egy olyan grafikus kezelőfelületű Python program létrehozása, amely képes a kézzel írt (pontosabban egérrel rajzolt) számjegyek felismerésére
- A felismeréshez egy többrétegű neurális hálózatot alakítunk ki, s a modell betanítását az MNIST adatbázis segítségével végezzük. Az adatbázis 28x28-as felbontású képeket tartalmaz, így a saját rajzainkat is ilyen méretűre kell majd transzformálni
- A fejlesztés több lépésre bontva valósul meg:
  - **Grafikus felhasználói felület kipróbálása** – A felhasználó rajzol egy számjegyet, amit az alkalmazás előkészít feldolgozásra (28x28-as mátrix)
  - **Betanítás és a modell elmentése** – Az **MNIST** adatbázissal történő tanítás után a hálózat optimalizált paramétereit (a modellt) elmentjük későbbi használatra
  - **A grafikusan beírt számok felismerése** – A betanított modell megpróbálja azonosítani a felhasználó által beírt számjegyet, majd megjeleníti az eredményt
- A grafikus felülethez a Python „beépített” **tkinter** könyvtárát használjuk, a telepítendő könyvtárak pedig: **numpy, matplotlib, tensorflow**

A grafikus kezelőfelület kipróbálása

- A fejlesztés első lépéseként a **Tkinter** grafikus könyvtár segítségével egy interaktív rajzolófelületet (canvas) hoztunk létre , amely lehetővé teszi a felhasználó számára, hogy egérrel rajzoljon egy számjegyet
- Az alkalmazás működése:
  - **Rajzolás egérrel:** A felhasználó egy 280x280 pixeles vászonra rajzolhat, amelyet egy 28x28-as rácsra osztunk fel.
  - **Adatok tárolása:** A képet egy 28x28-as **NumPy** mátrixba mentjük, ahol az aktív pixelek 1-es értéket kapnak, míg az üres területek 0-s értéket
  - **Indítás gomb:** A gomb megnyomásakor a program kiírja a létrehozott bináris mátrixot, amely az **MNIST** képformátumhoz igazodik
- Ezzel előkészítettük az adatok megfelelő formázását, hogy a modell később fel tudja dolgozni azokat



draw_to_matrix.py – 2/1.

```
import tkinter as tk
import numpy as np
import os

os.system("")       # Engedélyezzük az ANSI escape kódokat Windows CMD-ben
GRID_SIZE = 28      # 28x28-as rács
CANVAS_SIZE = 280   # 280x280 px

class DrawApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Számjegyrajzoló")

        self.canvas = tk.Canvas(root, width=CANVAS_SIZE, height=CANVAS_SIZE, bg="white")
        self.canvas.pack()
        self.canvas.bind("<B1-Motion>", self.paint)  # Bal egérgomb mozgatáskor rajzolás

        self.button = tk.Button(root, text="Indítás", command=self.process_image)
        self.button.pack()

        self.image_matrix = np.zeros((GRID_SIZE, GRID_SIZE), dtype=int)
```

draw_to_matrix.py – 2/2.

```
    def paint(self, event):
        x, y = event.x, event.y
        size = CANVAS_SIZE // GRID_SIZE             # Kisebb cellaméret
        self.canvas.create_rectangle(x, y, x + size, y + size, fill="black", outline="black")

        grid_x, grid_y = x // size, y // size
        if 0 <= grid_x < GRID_SIZE and 0 <= grid_y < GRID_SIZE:
            self.image_matrix[grid_y, grid_x] = 1  # Pixel aktiválás

    def process_image(self):
        print("Rajzolt kép mátrixa:")
        for row in self.image_matrix:
            for num in row:
                if num == 1:
                    print("\033[33m" + str(num) + "\033[0m", end=" ")  # Sárga szín az 1-eseknek
                else:
                    print(str(num), end=" ")  # Normál szín a 0-knak
            print()  # Új sor

root = tk.Tk()
app = DrawApp(root)
root.mainloop()
```

Megjegyzés: ezt a „szögletes” rajzolási módot a későbbiekben majd negváltoztatjuk

Python könyvtárak a gépi tanuláshoz

- **NumPy** – Tömbkezelés és mátrixműveletek, ami az ML alapját adja
- **Pandas** – Adatok betöltése és előkészítése, táblázatos feldolgozások
- **Matplotlib** és **Seaborn** – Adatok vizualizációja és elemzése grafikonokkal
- **Scikit-learn** – Gépi tanulás alapkönyvtára, amely klasszikus ML algoritmusokat tartalmaz
- **TensorFlow** – Mélytanulás és neurális hálók építéséhez szükséges keretrendszer
- **Keras** – Magas szintű API **TensorFlow** felett, amely egyszerűsíti a hálók tervezését és tanítását. Korábban önálló csomag volt, ma már a **Tensorflow** része

A betanítani kívánt modell ismertetése

- A programunkban létrehozott neurális hálózat egy teljesen összekapcsolt, előrecsatolt (fully connected, feedforward) struktúra, amely négy rétegből áll:
- **Bemeneti réteg** – 784 neuronnal (28x28 képpont egy vektorba „lapítva”).
- **Rejtett rétegek:**
  - **Első rejtett réteg:** 128 neuron, **ReLU** aktiváció
  - **Második rejtett réteg:** 64 neuron, **ReLU** aktiváció
- **Kimeneti réteg** – 10 neuron, **Softmax** aktiváció, a számjegyek osztályozására
- Aktivációs függvények és megfontolások:
  - **ReLU** (Rectified Linear Unit) – Az első két rejtett rétegben használjuk, mert hatékonyan kezeli a nemlineáris mintázatokat, kiküszöböli a gradiens eltűnését, és gyorsabb tanulást biztosít
  - **Softmax** – A kimeneti rétegben használjuk, biztosítja, hogy a modell valószínűségeket számítson minden számjegyre (0–9 között), és ezek közül a legvalószínűbbet válassza ki

A neurális modell Pythonban

- Így definiáljuk a neurális hálózat szerkezetét **TensorFlow** és **Keras** segítségével:

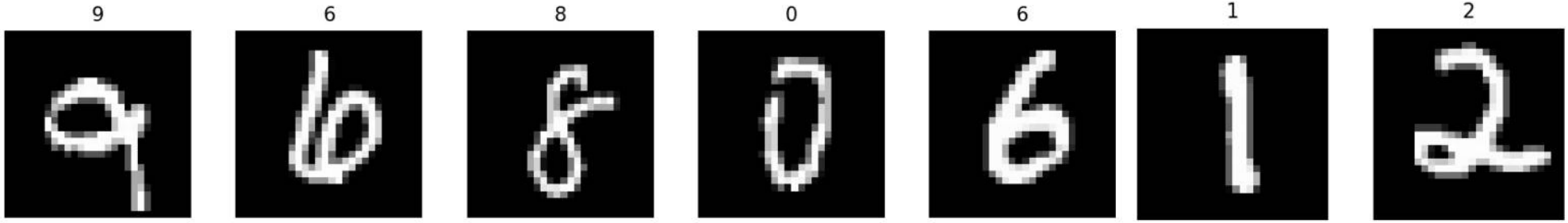
```
import tensorflow as tf
from tensorflow.keras import layers, models

# Modell létrehozása
model = models.Sequential([
    layers.Dense(128, activation='relu', input_shape=(784,)),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])

# Kompilálás az optimalizáló és a veszteségfüggvény megadásával
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
metrics=['accuracy'])
```

- **optimizer='adam'** – Adaptív tanulási sebességet biztosít a gyorsabb konvergenciához
- **loss='sparse_categorical_crossentropy'** – Osztályozási feladatokhoz megfelelő veszteségfüggvény, amely kiszámolja a valós és a predikált osztály közötti különbséget
- **metrics=['accuracy']** – A modell teljesítményének értékelése az osztályozási pontosság alapján

Betanítási adatok és betöltésük

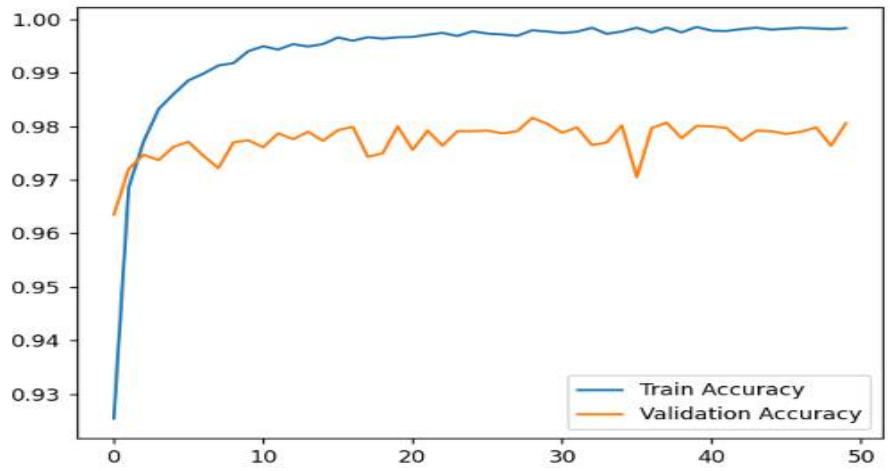
- A neurális hálózat betanításához az MNIST adathalmazt használjuk, amely kézzel írt számjegyek 28×28 pixeles képeit tartalmazza (60 000 minta és 10 000 teszt minta).
- Az adatok feldolgozásának lépései:
  - **Adatok betöltése:** Az MNIST készletet TensorFlow segítségével töltjük be.
  - **Normalizálás:** Az értékeket 0 és 1 közé skálázzuk a hatékonyabb tanítás érdekében.
  - **Adatformázás:** A képeket lapított vektorrá alakítjuk (28×28 ⟶ 784 pixel), hogy megfelelő legyen a teljesen összekapcsolt modellhez.

```
from tensorflow.keras.datasets import mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data() # MNIST adathalmaz betöltése
x_train, x_test = x_train / 255.0, x_test / 255.0        # Normalizálás
x_train = x_train.reshape(-1, 784)                       # Vektorrá lapítás
x_test = x_test.reshape(-1, 784)                         # Vektorrá lapítás
```



Betanítás és a modell mentése

- A modell tanítása során két fő teljesítménymutatót figyelünk meg:
- **Train Accuracy** (tanítási pontosság): a tanító adatokon elért pontosság
- **Validation Accuracy** (validációs pontosság): a tesztadatokon elért pontosság, ami jelzi, hogy a modell mennyire tudja általánosítani a tanult mintázatokat új adatokra
- Ideális esetben a Train Accuracy magas, a Validation Accuracy pedig közel azonos vele, elkerülve az alultanulást, de nem lényegesen alacsonyabb, mert az túltanulásra utalna
- **Eredmény ellenőrzése:** A tanítási és a validációs pontosságot grafikonon is ellenőrizzük
- **Modell mentése:** az optimalizált súlyokat, paramétereket elmentjük későbbi használatra



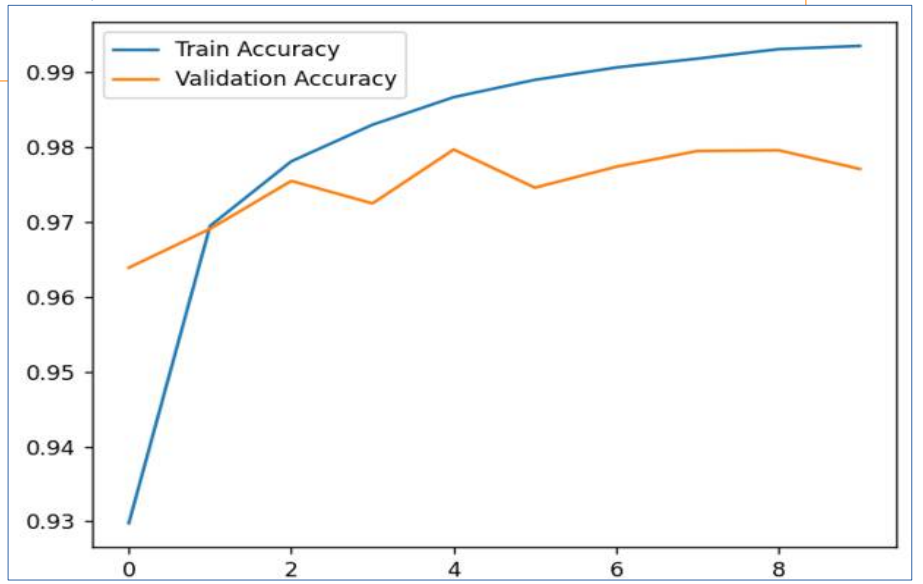
- Az **50 epochon** keresztüli tanulási folyamat jól szemlélteti, hogyan alakul a **Train Accuracy** és **Validation Accuracy**.
- **10 epoch után** a Train Accuracy szinte **tökéletes** (bőven **0.99 fölött**), ami azt mutatja, hogy a modell **jól megtanulta a tanító adatok mintázatait**.
- **A Validation Accuracy 0.78 körüli értékre áll be**, ami azt jelzi, hogy a modell képes általánosítani, de van még némi eltérés a tanító és a tesztadatok között.
- **További epochok nem hoznak jelentős javulást**, így **felesleges hosszabb betanítás**, mert az inkább túltanuláshoz vezethetne.

digit_trainer.py – 2/1.

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
import matplotlib.pyplot as plt
import os

# Engedélyezzük az ANSI escape kódokat Windows CMD-ben
os.system("")

(x_train, y_train), (x_test, y_test) = mnist.load_data() # MNIST adathalmaz betöltése
x_train, x_test = x_train / 255.0, x_test / 255.0        # Normalizálás
x_train = x_train.reshape(-1, 784)                       # Vektorrá lapítás
x_test = x_test.reshape(-1, 784)                         # Vektorrá lapítás

# Neurális háló létrehozása
model = models.Sequential([
    layers.Dense(128, activation='relu', input_shape=(784,)),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
```

digit_trainer.py – 2/2.

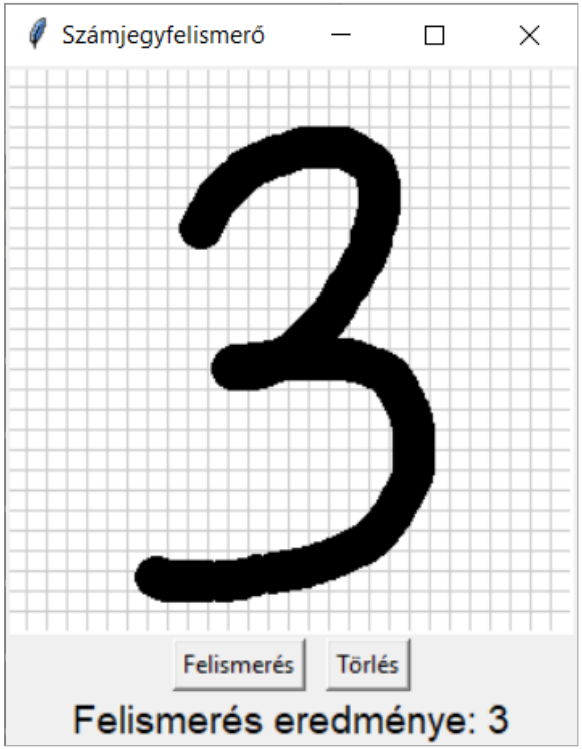
```python
# Modell betanítása
history = model.fit(x_train, y_train, epochs=10, validation_data=(x_test, y_test))
plt.plot(history.history["accuracy"], label="Train Accuracy")
plt.plot(history.history["val_accuracy"], label="Validation Accuracy")
plt.legend()
plt.show()

# Modell mentése
model.save("szamjegyfelismero_model2.h5")
print("Modell elmentve!")
```



- A grafikonon a tanítási és validációs pontosság alakulása követhető, ami segít eldönteni, hogy a modell megfelelően tanul-e, vagy esetleg túltanulás tapasztalható
- Az elmentett HDF5 fájlban tároljuk a modell paramétereit, így a modell betölthető, új adatokon is futtatható anélkül, hogy újra kellene tanítani

Számjegyfelismerő program: recognize_digit.py

- Ez a program a korábban bemutatott **draw_to_matrix.py** kibővített változata, amelyet továbbfejlesztettünk, hogy képes legyen a korábban betanított modell felhasználásával a grafikus ablakban megrajzolt számjegy felismerésére



Új funkciók:

- **Törlési lehetőség,** hogy több próbálkozás is lehessen
- **Üres rajz ellenőrzése**, hogy elkerüljük a hibás felismeréseket
- **Simított rajzolás,** a felismerés javulása reményében

A program működése:

- Betöltjük a korábban elmentett betanított modellt
- A felhasználó rajzol egy számjegyet a vászonra.
- A program a képet megfelelő formába alakítja a felismeréshez
- A korábban betanított modell felhasználásával a lefuttatjuk a számjegyfelismerést
- Az eredményt kiíratjuk

recognize_digit.py – 4/1.

```
import tkinter as tk
import numpy as np
from tensorflow.keras.models import load_model
GRID_SIZE = 28      # 28x28-as rács
CANVAS_SIZE = 280   # 280x280 px
OVAL_RADIUS = 10    # Finomabb rajzolás kisebb körökkel
model = load_model("szamjegyfelismero_model.h5") # **Elmentett modell betöltése**
print("Modell betöltve!")

class DrawApp:
    def __init__(self, root):
        self.root = root
        self.root.title("Számjegyfelismerő")

        # **Canvas létrehozása**
        self.canvas = tk.Canvas(root, width=CANVAS_SIZE, height=CANVAS_SIZE, bg="white")
        self.canvas.pack()
        self.canvas.bind("<B1-Motion>", self.paint)
        self.draw_grid()  # Rácsháló megrajzolása

        self.button_frame = tk.Frame(root)          # **Gombok egy Frame-ben**
        self.button_frame.pack()
```

recognize_digit.py – 4/2.

```
        self.button = tk.Button(self.button_frame, text="Felismerés", command=self.process_image)
        self.button.pack(side=tk.LEFT, padx=5)

        self.clear_button = tk.Button(self.button_frame,text="Törlés",command=self.clear_canvas)
        self.clear_button.pack(side=tk.LEFT, padx=5)

        # **Eredmény kijelzés**
        self.result_label = tk.Label(root, text="Felismerés eredménye: -", font=("Arial", 14))
        self.result_label.pack()

    def paint(self, event):
        """Az egér húzásakor fekete köröket rajzolunk a vászonra."""
        x, y = event.x, event.y
        self.canvas.create_oval(x - OVAL_RADIUS, y - OVAL_RADIUS, x + OVAL_RADIUS,
        y + OVAL_RADIUS, fill="black", outline="black")

    def draw_grid(self):  # Halvány rácsháló rajzolása a vászonra.
        cell_size = CANVAS_SIZE // GRID_SIZE
        for i in range(GRID_SIZE):
            self.canvas.create_line(i*cell_size, 0, i*cell_size, CANVAS_SIZE, fill="#CCCCCC")
        for j in range(GRID_SIZE):
            self.canvas.create_line(0, j*cell_size, CANVAS_SIZE, j*cell_size, fill="#CCCCCC")
```

recognize_digit.py – 4/3.

```
    def clear_canvas(self):          # Törli az összes rajzot a vászonról
        self.canvas.delete("all")
        self.draw_grid()             # A hálót újra kirajzoljuk

    def get_image_matrix(self):
        """A vászon pixeleiből egy normalizált 28x28-as mátrixot készít, rácsok kiszűrésével."""
        matrix = np.zeros((GRID_SIZE, GRID_SIZE), dtype=float)
        cell_size = CANVAS_SIZE // GRID_SIZE
        for i in range(GRID_SIZE):
            for j in range(GRID_SIZE):
                x1, y1 = i * cell_size, j * cell_size
                x2, y2 = x1 + cell_size, y1 + cell_size
                # **Összes fekete pixel megszámolása a cellában**
                black_pixel_count = 0
                for x in range(x1, x2):
                    for y in range(y1, y2):
                        items = self.canvas.find_overlapping(x, y, x, y)
                        if items:
                            black_pixel_count += 1
                black_pixel_count = max(0, black_pixel_count-19)  # Rácspixelek kiszűrése
                matrix[j, i] = black_pixel_count / 100.0   # **Normalizálás: osztás 100-zal**
        return matrix
```

recognize_digit.py – 4/4.

```
    def process_image(self):
        """A Felismerés gomb lenyomásakor a rajz kiolvasása, adatelőkészítés és felismerés"""
        matrix = self.get_image_matrix()  # canvas kiolvasása és a 28x28-as mátrik előkészítése
        input_vector = matrix.flatten()   # A mátrixot vektorrá lapítjuk

        # **Ha túl kicsi a rajz, figyelmeztetés**
        if np.sum(matrix) < 10:
            self.result_label.config(text="Rajzolj nagyobb számjegyet.")
            return

        # Karakterfelismerés és az eredmények kiírása
        prediction = model.predict(input_vector.reshape(1, 784))
        formatted_prediction = ["{:.3f}".format(p) for p in prediction[0]]
        print("Prediction:", formatted_prediction)
        predicted_digit = np.argmax(prediction)
        self.result_label.config(text=f"Felismerés eredménye: {predicted_digit}")

root = tk.Tk()
app = DrawApp(root)
root.mainloop()
```