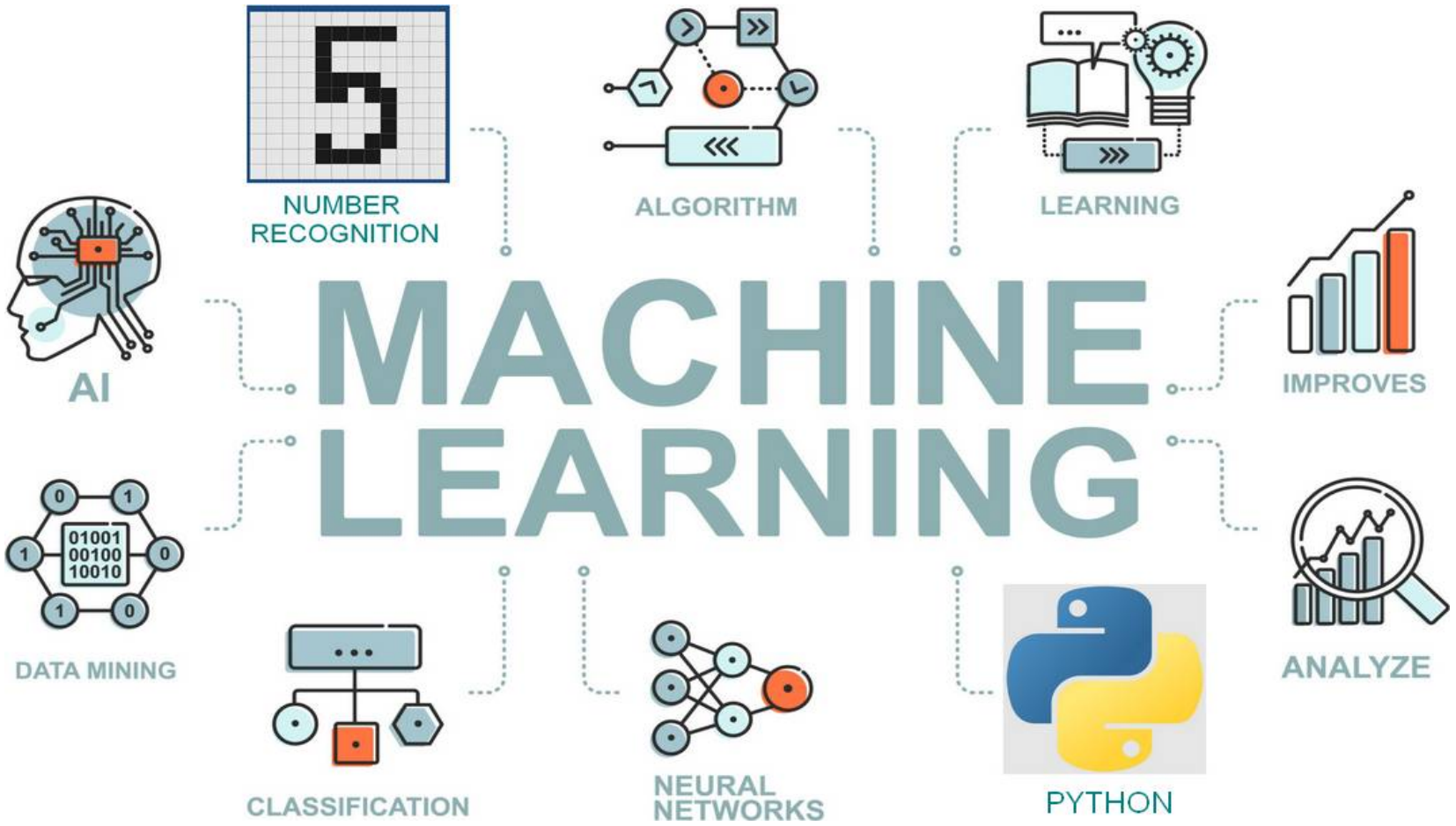
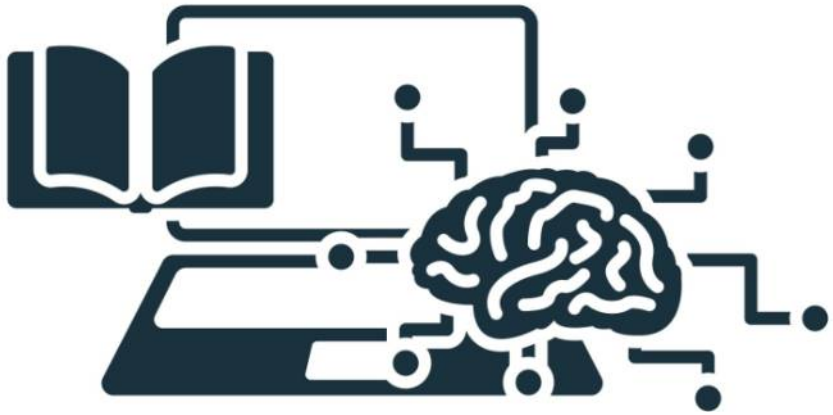


A gépi tanulás alapjai Pythonnal – 2. rész

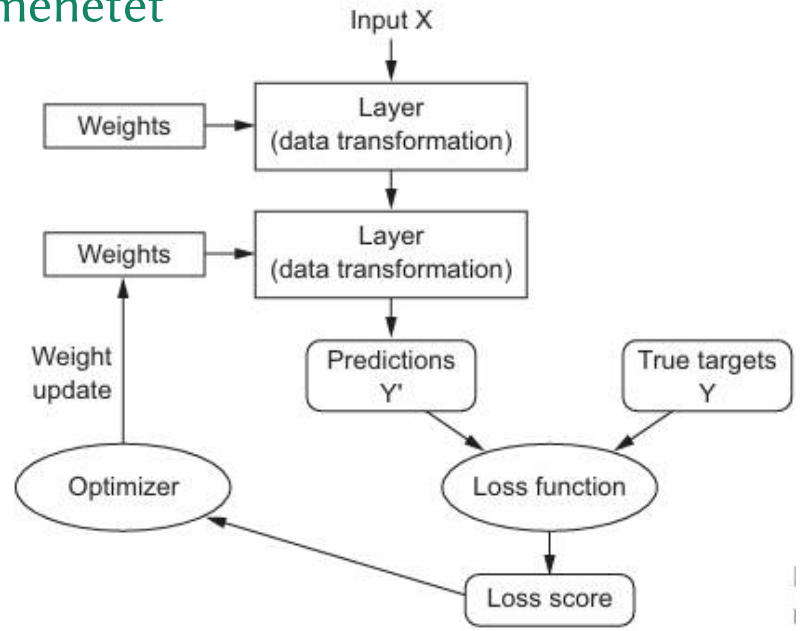
Felhasznált és ajánlott irodalom

- François Chollet: Deep Learning with Python (a tankönyv)
  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
- Ertedmar.hu: Konvolúcióról érthetően



Emlékeztető: a neurális hálózat anatómiája

- Ahogy az előző előadásban láthattuk, egy neurális hálózat betanítása a következő objektumok körül forog:
  - **Rétegek** (layers), amelyeket neurális hálózattá (modell) kombinálunk
  - A **bemeneti adatok** és a hozzájuk tartozó **célértékek**
  - A **veszteségfüggvény,** amely meghatározza a tanuláshoz használt visszacsatoló jelet
  - Az **optimalizáló,** amely meghatározza a tanulás menetét
- Kölcsönhatásukat az alábbi ábra szemlélteti:
  - A hálózat, amely egymáshoz láncolt rétegekből áll, a bemeneti adatokhoz predikciókat rendel
  - A veszteségfüggvény ezután összehasonlítja ezeket az predikciókat az előírt célokkal, és egy veszteségértéket hoz létre: ez a mértéke annak, hogy a hálózat predikciói mennyire felelnek meg az elvárt értékeknek
  - Az optimalizáló ezt a veszteségértéket használja fel a hálózat súlyainak frissítésére



Forrás: F. Chollet: Deep Learning with Python

Veszteségfüggvények és optimalizálók

- A **mélytanulási modell** esetünkben egy irányított, aciklikus réteggráf (a neuronokat rétegekbe szervezzük, az adatáramlás egyirányú). A hálózati topológia kiválasztásával a lehetőségek terét (hipotézisterét) egy adott tenzorművelet-sorozatra korlátozzuk, a bemeneti adatokat kimeneti adatokra képezve le. A tanulás folyamán pedig egy jó értékkészletet keresünk a tenzorműveletekben szereplő súlyfaktorok számára
- A hatékony tanuláshoz a hálózat kialakítása után még két dolgot meg kell választani:
  - **Veszteségfüggvény** (célfüggvény) – Az a mennyiség, amelyet minimalizálni kell a betanítás során. A feladat sikerességének mértékét jelöli
  - **Optimalizálási módszer** – Meghatározza, hogy a hálózat hogyan frissüljön a veszteségfüggvény alapján. Az általunk használt **ADAM** módszer a **sztochasztikus gradiens módszer** (SGD) egy specifikus, adaptív változatát valósítja meg
- **Több kimenetű hálózatok** esetében minden kimenethez külön veszteségfüggvény tartozhat. A gradiens módszer azonban csak **egyetlen skaláris veszteségértéket** tud optimalizálni, ezért az egyes kimenetek veszteségeit valahogy kombinálnunk kell, például átlagolással

Forrás: F. Chollet: Deep Learning with Python

A célfüggvény kiválasztása

- A megfelelő célfüggvény kiválasztása a megfelelő problémához rendkívül fontos. Helytelen célfüggvény kiválasztása torzíthatja a tanulást, mert az optimalizálás minden lehetőséget megragad a veszteség minimalizálására – akár a nem kívánt irányokat is
- Szerencsére, a gyakori problémákhoz vannak egyszerű irányelvek, amelyeket követhetünk a helyes veszteség kiválasztásához

Probléma típusa	Ajánlott célfüggvény	Ajánlott optimalizálás
Bináris osztályozás (pl, igen/nem döntések	Bináris keresztentrópia (Binary Cross-Entropy)	ADAM, SGD
Többosztályos osztályozás (pl. képfelismerés)	Kategórikus keresztentrópia (Categorical Cross-Entropy)	ADAM, RMSprop
Regresszió (folytonos értékek előrejelzése)	Átlagos négyzetes hiba (Mean Squared Error)	ADAM, SGD
Szekvencia tanulás (pl. beszédfelismerés)	Időbeli osztályozási veszteség (CTC Loss)	ADAM, RMSprop

Forrás: F. Chollet: Deep Learning with Python

Bináris keresztentrópia

- A **Binary Cross-Entropy** (BCE) veszteségfüggvényt akkor használjuk, ha két osztály van (pl. igen/nem, macska/kutya). A **BCE** méri, mennyire tér el a modell által adott valószínűségi előrejelzés az igazi címkéktől. Matematikailag így néz ki:

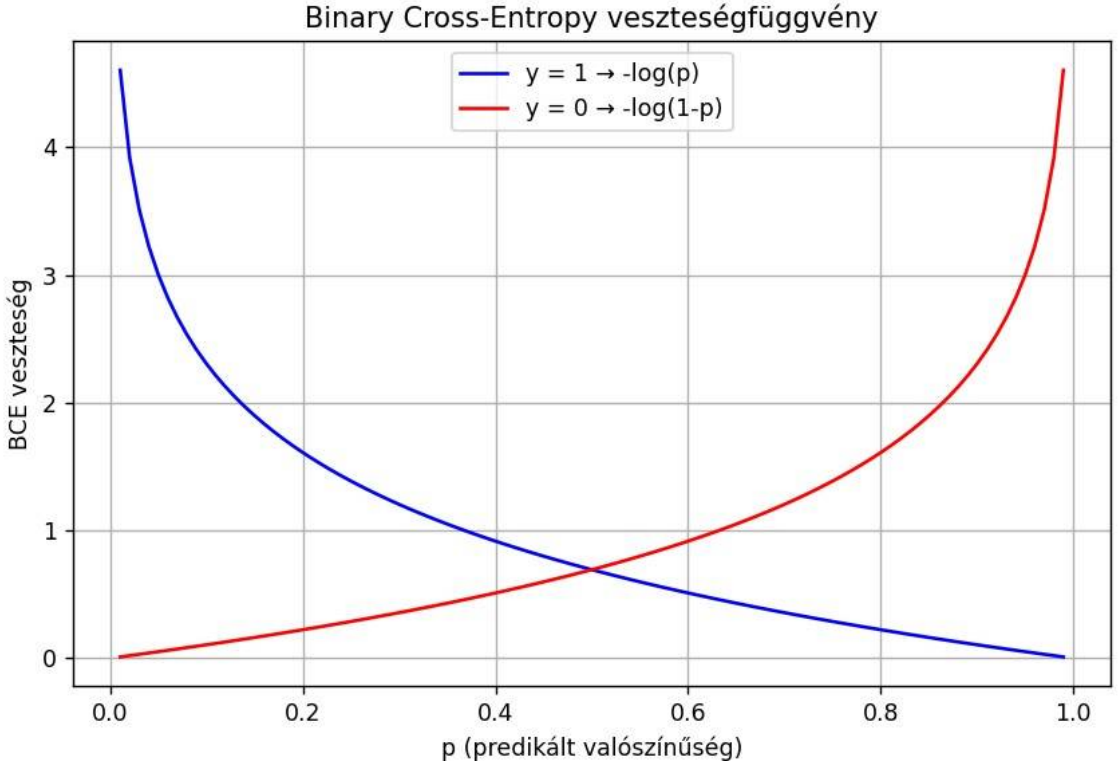
$$BCE = -\frac{1}{N}\sum_{i=1}^{N}\left[y_i \log(p_i) + (1-y_i)\log(1-p_i)\right]$$

ahol:

N az adatok száma *(pl. 1 batch N =32)*

y_i az elvárt eredmény (0 vagy 1),

p_i a modell által adott valószínűség (0 – 1 közötti szám)



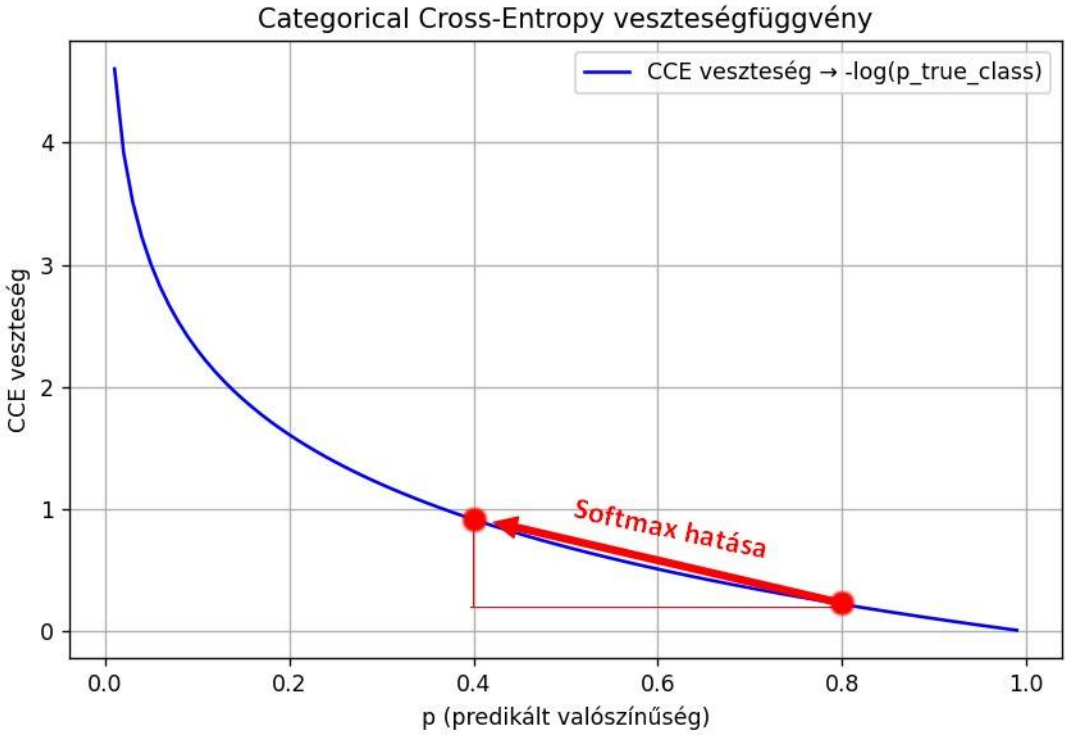
Kategórikus keresztentrópia

- A **Categorical Cross-Entropy** (CCE) veszteségfüggvényt akkor használjuk, ha több osztály van (pl. macska, kutya, madár). A **CCE** azt méri, hogy mennyire tér el a modell által adott valószínűségi eloszlás az igazi osztálytól. Matematikailag így néz ki:

$$L = -\frac{1}{N}\sum_{n=1}^{N}\sum_{c=1}^{C} y_{n,c}\log(p_{n,c})$$

ahol:

- N az adatok száma *(pl. 1 batch N =32)*
- C az osztályok száma
- $y_{i,C}$ az elvárt eredmény (one-hot kód),
- $p_{i,C}$ a modell által adott valószínűség



- Bár a képletben csak egy osztály valószínűsége szerepel, a Softmax normálizálás miatt közvetetten a többi kimenet is befolyásolja a veszteséget

Konvolúciós hálók: a gépi látás alapjai

- Az előző előadásban teljesen összekötött ún. **Dense neurális hálózattal** próbáltuk megoldani a karakterfelismerést, de a siker mérsékelt volt
- Ebben az előadásban továbbfejlesztjük a modellünket az úgynevezett **konvolúciós neurális hálózatok** (ConvNet, CNN) felhasználásával
- **Miben különbözik a ConvNet a Dense hálózattól?**
  - A **Dense hálózat** minden neuronját minden másikhoz kapcsolja, így nem veszi figyelembe a térbeli összefüggéseket. Minden pixel egy különálló adatpont
  - A **ConvNet hálózat** a konvolúciós rétegek révén lokálisan vizsgálja a képet, figyelembe veszi a szomszédos pixelek összefüggéseit, és hatékonyabban tanul vizuális mintázatokat. A **ConvNet hálózatok** fontos jellemzője a hierarchikus tanulás. Az első réteg a kisebb lokális struktúrák felismerését tanulja meg, a további rétegek már nagyobb geometriai struktúrákat ismerik fel (pl. formák, kontúrok)
- A **ConvNet hálózat** önmagában nem alkalmas kategorizálásra, hanem jellemzőket nyer ki *(features extraction)* a képből, amelyeket egy vagy több **Dense réteg** osztályoz

Mi a konvolúció?

- A matematikában az integrálható ***f, g*** függvények **konvolúcióján** az $f * g : x \mapsto \int_{-\infty}^{\infty} f(t)g(x-t)dt$ integrállal definiált függvényt értik
- A digitális jelfeldolgozásban és a képfeldolgozásban használt diszkrét függvényekre vonatkozó **diszkrét konvolúció** képzési szabálya:

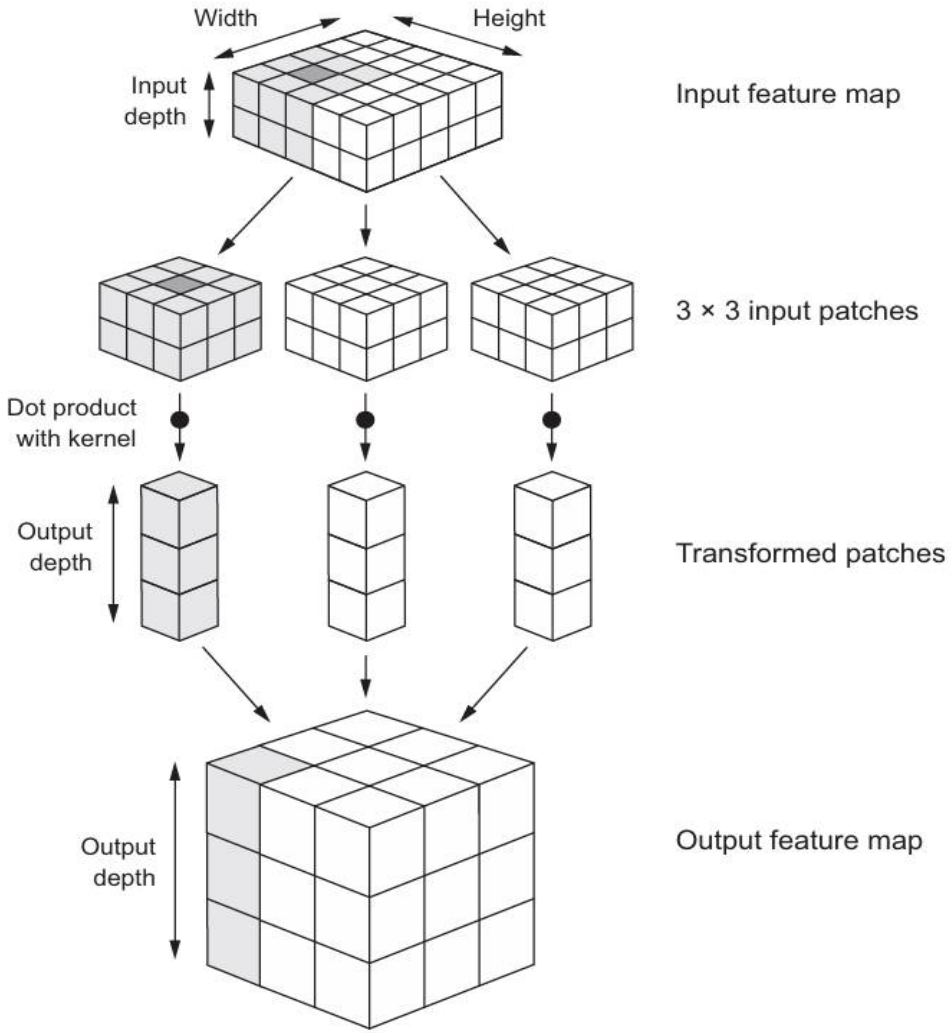
$$(F * G)[i,j] = \sum_m \sum_n F[m,n]G[i-m, j-n]$$

A gépi látásban ezt használjuk a képek feldolgozására, hogy a neurális hálózatok könnyebben megtanulják az egyes vizuális struktúrákat

- **Hogyan működik?** A konvolúciós művelet során egy kis szűrő (kernel, a fenti képletben az ***F[n,m]*** mátrix) végigsétál a képen, és minden részletet egy helyi súlyozott összegezés alapján új értékre alakít. Ezáltal a kép jellemzői kiemelkednek. A súlyfaktorok értékét a tanítási folyamat során optimalizáljuk

Hogy működik a ConvNet?

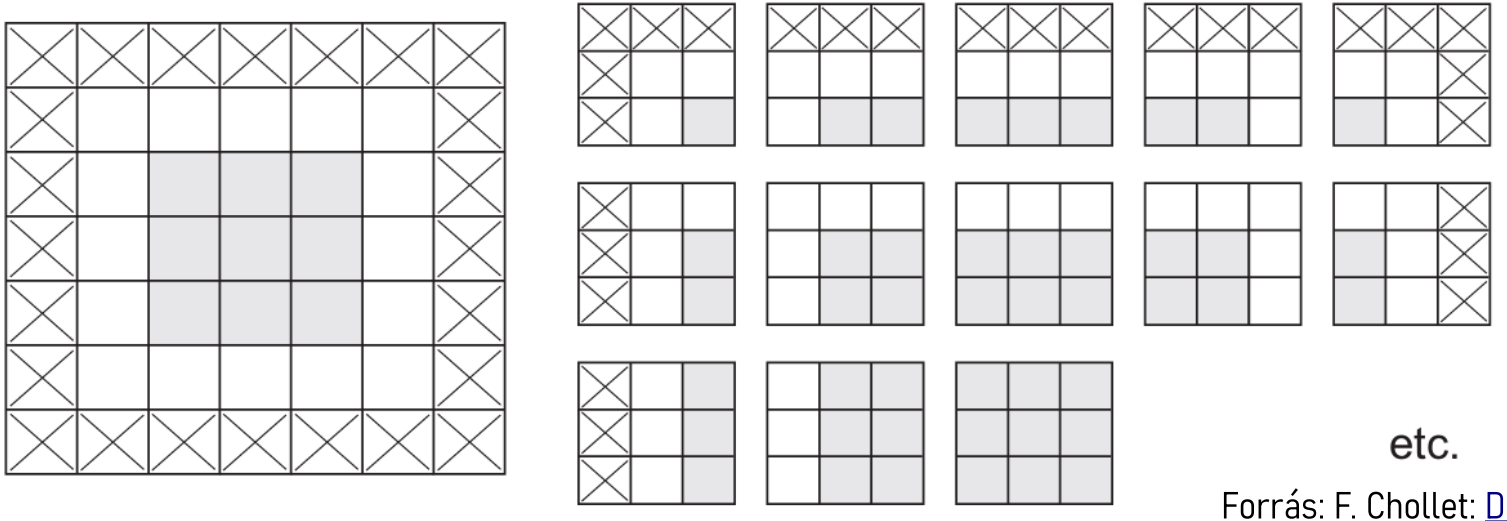
- A tankönyv 5.4 ábrája szemlélteti a konvolúciós hálózati réteg működését
- **Bemenet:** Egy 5×5×2 méretű tenzor, amely tartalmazza a képi adatokat két csatornával
- **Kernel:** Egy 3×3 méretű szűrő, amely a kép minden részletén végigsétál és helyi mintázatokat keres
- **Szeletek:** A kernel 9 darab 3×3×2 méretű szeletet metsz ki a bemenetből (minden pozíción egy kis részletet elemez)
- **Súlyozás és összeadás:** a szeletek celláit a kernel súlyaival megszorozzuk és összegezzük
- **Kimenet:** Itt három különböző kernelt is használtunk, így 9 darab 1×1×3 méretű oszlopot kaptunk, amelyeket végül egy 3×3×3 méretű kimeneti tenzorrá állítottunk össze



Forrás: F. Chollet: Deep Learning with Python

Padding: Peremkiegészítés

- A konvolúció csökkenti a bemeneti kép méretét. **Padding** (peremkiegészítés) segítségével megőrizhetjük az eredeti méretet
- A tankönyv 5.6 ábrája bemutatja, hogy 3x3as kernel esetén körben egy cellányi peremkiegészítéssel 5x5 pozíciót vehet föl a kernel, tehát a konvolúció kimenete is 5x5 méretű lesz
- **Padding típusok** a Keras könyvtárban: `'valid'` (nincs padding), `'same'` (0-val egészít ki)

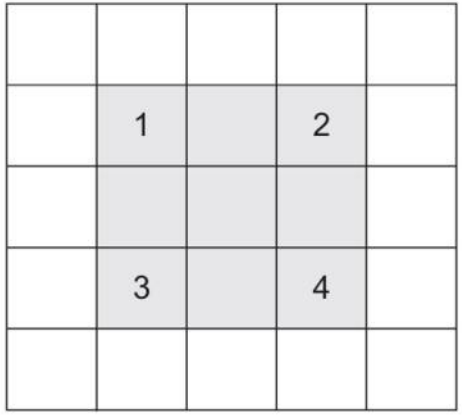


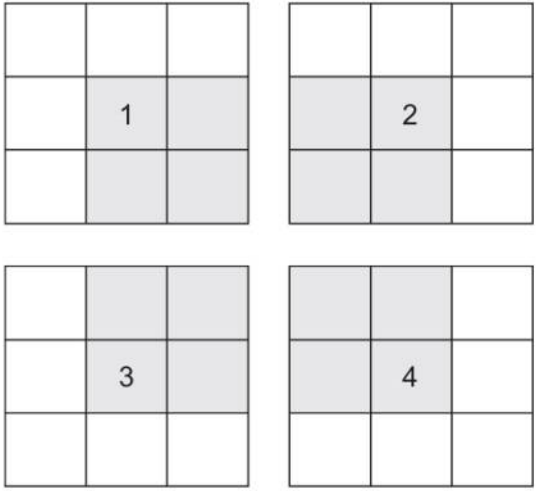
Forrás: F. Chollet: Deep Learning with Python

Stride: lépésköz

- A **stride** (lépésköz) határozza meg, hogy a kernel hány pixelenként lép előre. A korábbi ábrákon **Stride = 1** volt, a kernel pixelenként lépett, a kimeneti méret a paddingtől függött
- Ha **Stride > 1,** akkor a kernel ugrik, kevesebb pozíciót vizsgál, a kimeneti méret csökken. Például **Stride = 2** esetén felezi a kimeneti térkép méretét
- A tankönyv 5.7 ábrája azt mutatja be, hogy az 5x5-ös bemeneti képből 3x3-a kernellel és 2x2 lépésközzel négy szeletet tudunk kivágni
- **Mikor hasznos?**
  Méretcsökkentéshez, hogy kevesebb számításra legyen szükség.

  Ritkán használjuk, mert felbontás-csökkentésre a **max-pooling** módszer hatékonyabb





Max-pooling: Mintázat-sűrítés

- A **max-pooling** egy méretcsökkentő művelet. Egy 2×2 vagy 3×3-as ablak végigpásztázza a képet és minden ablakból csak a legnagyobb érték marad meg, a többit eldobja. Így a képméret csökken, de a leglényegesebb információk megmaradnak
- Miért van rá szükség?
  - **Csökkenti a számítási igényt,** miközben megőrzi a fontos mintázatokat
  - Segíti a **hierarchikus tanulást,** így a mélyebb rétegek nagyobb struktúrákat ismerhetnek fel
  - A **max-pooling** kiemeli a lényegi mintázatokat, így a hálózat kevésbé érzékeny az apró lokális változásokra
- Miért jobb a **max-pooling,** mint az **átlagoló pooling** vagy a **stride-alapú** csökkentés?
  - A **max-pooling** kiemeli a legfontosabb jellemzőket, míg az **átlagoló pooling** elmoshatja azokat
  - A **stride-alapú** konvolúció nem mindig emeli ki a lényegi mintázatokat, míg a **max-pooling** garantálja, hogy a legdominánsabb jellemzők megmaradjanak

Konvolúciós hálózatok TensorFlow/Keras segítségével

- **Konvolúciós réteg létrehozása:** A **Keras** könyvtárban a **Conv2D** osztály segítségével definiálhatunk egy konvolúciós réteget

```
from tensorflow.keras.layers import Conv2D
conv_layer = Conv2D(filters=32, kernel_size=(3,3), strides=(1,1),
padding='same', activation='relu')
```

  - `filters=32` ⟶ 32 darab szűrőt alkalmazunk
  - `kernel_size=(3,3)` ⟶ 3×3 méretű kernel
  - `strides=(1,1)` ⟶ Lépésköz 1 pixel
  - `padding='same'` ⟶ A kimeneti méret megegyezik a bemeneti mérettel
- **Pooling réteg létrehozása:** a MaxPooling2D osztály segítségével csökkenthetjük a kimeneti térkép méretét:

```
from tensorflow.keras.layers import MaxPooling2D
pooling_layer = MaxPooling2D(pool_size=(2,2), strides=(2,2), padding='valid')
```

  - `pool_size=(2,2)` ⟶ 2×2-es ablak
  - `strides=(2,2)` ⟶ Lépésköz 2 pixel
  - `padding='valid'` ⟶ Nincs peremkiegészítés

A számjegyfelismerő neurális hálózatunk bővítése konvolúciós rétegekkel

- Az előző előadásban bemutatott, számjegyfelismerésre használt mélytanuló hálózatot **konvolúciós** és **MaxPooling** rétegekkel bővítjük. A konvolúciós rétegek felelősek a mintázatok felismeréséért, míg a MaxPooling rétegek segítenek csökkenteni a képméretet, miközben megtartják a legfontosabb jellemzőket
- Ez a **CNN** modell ez előzőnél mélyebb és strukturáltabb reprezentációt biztosít, így a hálózat reményeink szerint pontosabban képes felismerni a számjegyeket

```
model = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
```

digit_trainer2.py – 2/1.

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

(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

model = models.Sequential([
    layers.Conv2D(32, (3,3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2,2)),
    layers.Conv2D(64, (3,3), activation='relu'),
    layers.MaxPooling2D((2,2)),
    layers.Flatten(),
    layers.Dense(128, activation='relu'),
    layers.Dense(10, activation='softmax')
])
model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])

history = model.fit(x_train, y_train, epochs=5, validation_data=(x_test, y_test)) # Modell betanítása
plot_training_history(history)
model.save("konvolucios_modell5.h5") # A betanított CNN modell elmentése
```

A modell betanításához most is az MNIST adatbázist használtuk, ami a tensorflow.keras.datasets modulból könnyen betölthető. Az adatokat 0-1 közé normáltuk

A sparse_categorical_crossentropy függvényt akkor használjuk, ha a célértékek nem one-hot kódolással, hanem kategória index formájában adottak

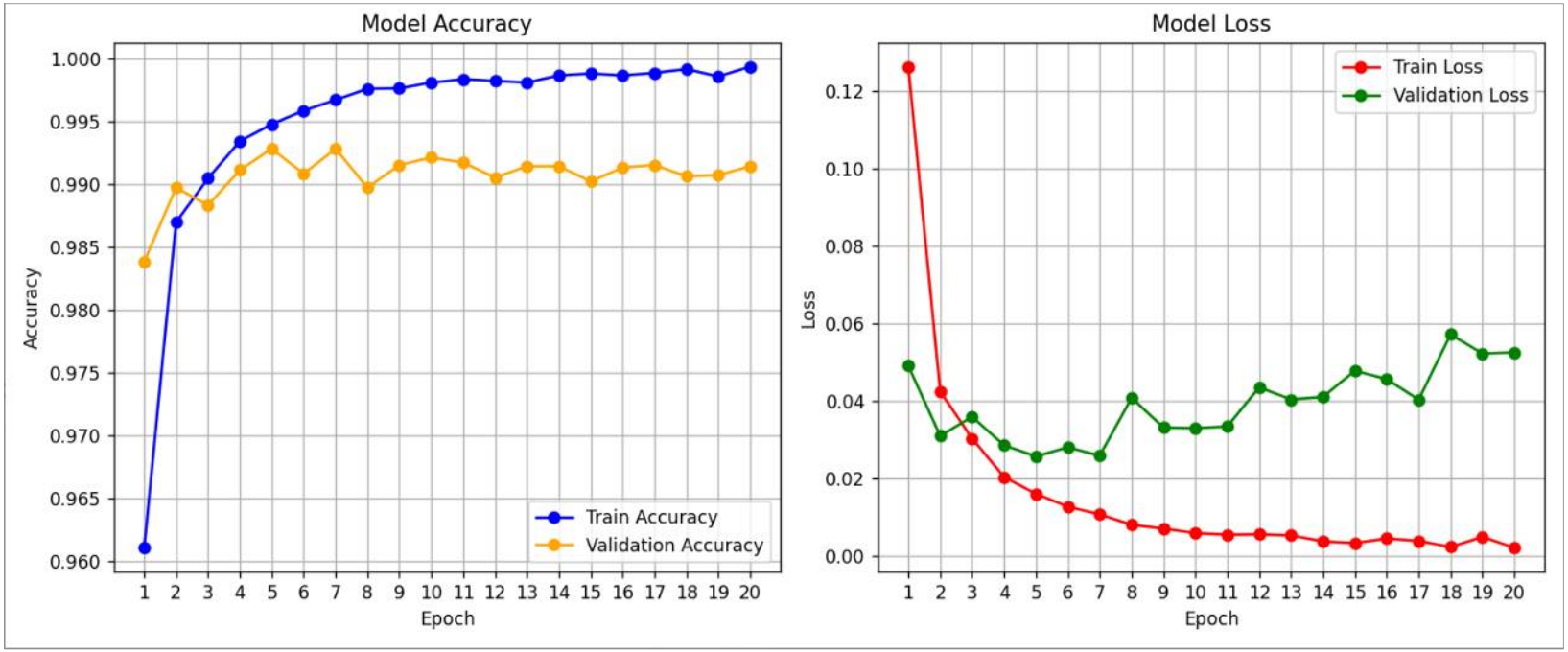
batch_size = 32 (default)

digit_trainer2.py – 2/2.

```
def plot_training_history(history):
    """Megjeleníti az accuracy és loss értékeket megfelelő tengelyskálázással."""
    epochs = np.arange(1, len(history.history['accuracy']) + 1)
    fig, axs = plt.subplots(1, 2, figsize=(12, 5))
    #----------------------------- ** Train és Validation Accuracy**
    axs[0].plot(epochs, history.history['accuracy'], label='Train Accuracy', color='blue', marker='o')
    axs[0].plot(epochs,history.history['val_accuracy'],label='Validation Accuracy',color='orange', marker='o')
    axs[0].set_title('Model Accuracy')
    axs[0].set_xlabel('Epoch')
    axs[0].set_ylabel('Accuracy')
    axs[0].set_xticks(epochs)  # Csak egész epoch számok megjelenítése
    axs[0].legend()
    axs[0].grid()
    #----------------------------- ** Train és Validation Loss**
    axs[1].plot(epochs, history.history['loss'], label='Train Loss', color='red', marker='o')
    axs[1].plot(epochs, history.history['val_loss'], label='Validation Loss', color='green', marker='o')
    axs[1].set_title('Model Loss')
    axs[1].set_xlabel('Epoch')
    axs[1].set_ylabel('Loss')
    axs[1].set_xticks(epochs)  # Csak egész epoch számok megjelenítése
    axs[1].legend()
    axs[1].grid()
    plt.tight_layout()
    plt.show()
```

A betanítás elemzése

- A betanítás során az **Accuracy** és **Loss** értékek vizualizációja segít a modell teljesítményének nyomon követésében. Az **Accuracy** mutatja a felismerési pontosságot, míg a **Loss** azt jelzi, milyen mértékben tér el az előrejelzés a valódi értéktől
- **Túltanítás elkerülése:** ahol a **Validation Loss** eléri a minimumát, ott érdemes megállítani a tanítást, hogy elkerüljük a túltanulást (itt epoch=5 körül látszik optimum)



Áttérés CNN-re a számjegyfelismerő programban

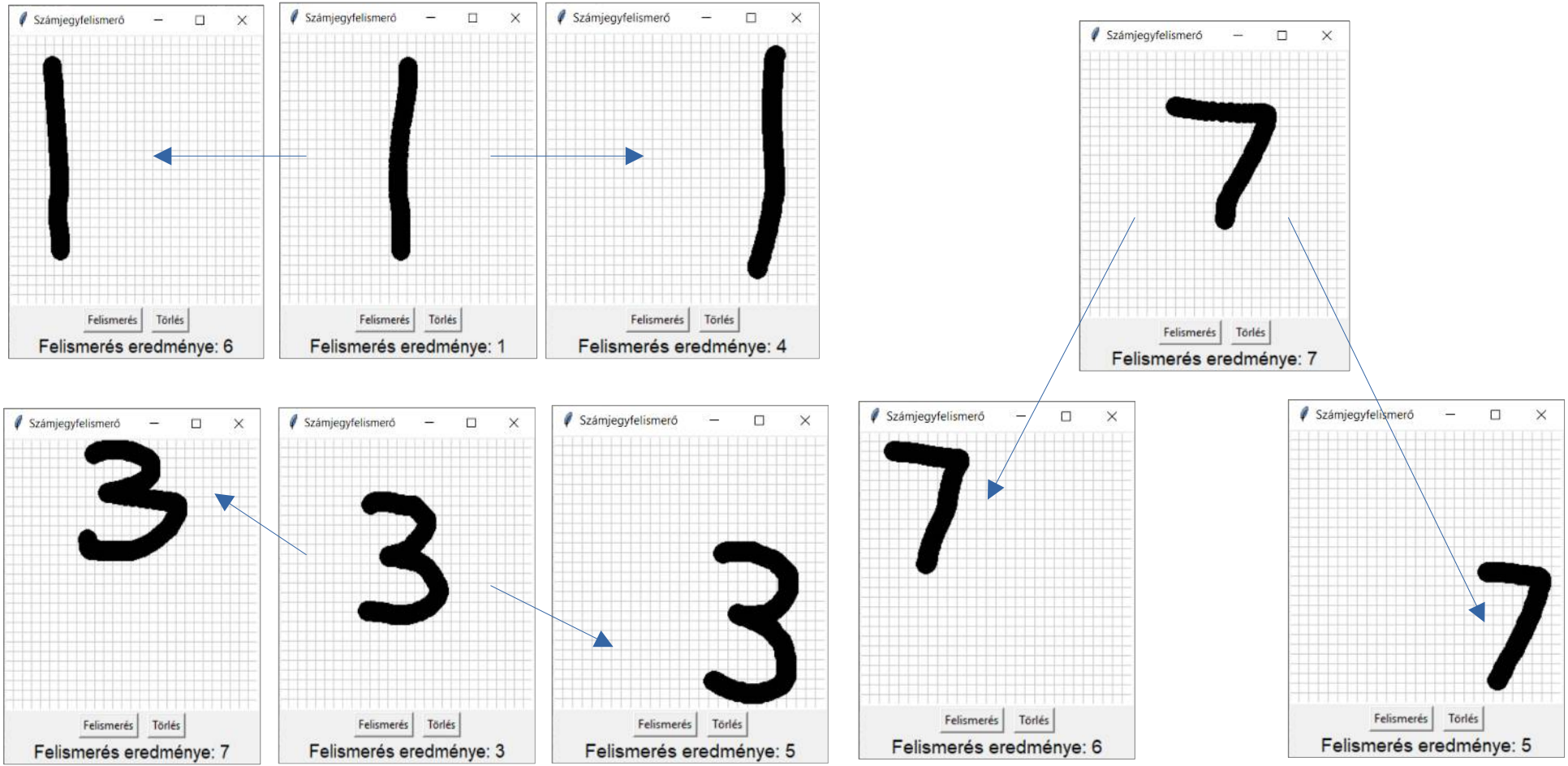
- A betanítás után elmentett új modellt **automatikusan betölti és használja** a számjegyfelismerő program
- A kézzel rajzolt számjegyeket azonban a CNN modell alkalmazása miatt picit **eltérő formátumban** kell átadni a felismeréshez. A CNN modellek általában **4D bemenetet várnak,** ahol az alakzat dimenziói: (batch_size, height, width, channels)
- `np.expand_dims(image_matrix, axis=-1)` ⟶ A képet egy csatornás (grayscale) formátumba alakítja, azaz (28, 28, 1) méretű lesz
- `np.expand_dims(image_matrix, axis=0)` ⟶ A képet egy *minibatch* részévé teszi, így az alak (1, 28, 28, 1) lesz, ahol az első dimenzió a *batch size*
- A módosított **recognize_digit2.py** program releváns sorai:

```
image_matrix = self.get_image_matrix()
# **Átalakítás CNN kompatibilis formátumra**
image_matrix = np.expand_dims(image_matrix, axis=-1)  # (28, 28, 1)
image_matrix = np.expand_dims(image_matrix, axis=0)   # (1, 28, 28, 1)
prediction = model.predict(image_matrix)
```

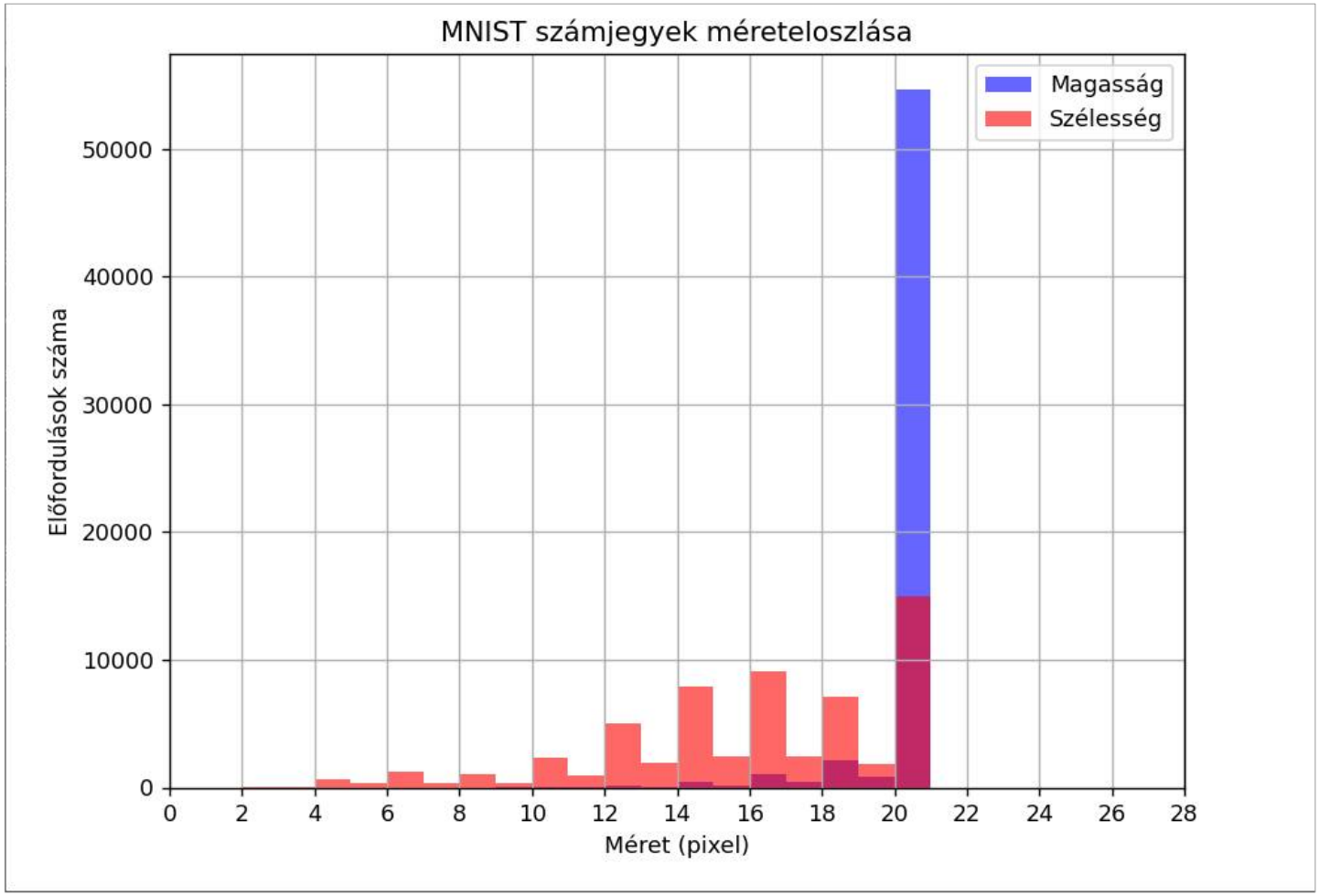
recognize_digit2.py futási eredménye

- A betanított **CNN** modell a **konvolucios_modell5.h5** fájlból (epoch=5), vagy a **konvolucios_modell20.h5** (epoch=20) fájlból tölthető be
- **Eredmények és tapasztalatok:**
  - Validation Accuracy javulást mutat: 0.98 ⟶ 0.99
  - A saját rajzaink felismerése sikeresebb lett, de nem tökéletes
  - Több hibás felismerés történt olyan számjegyeknél, ahol az írásmód jelentősen eltér a betanítási adatoktól
  - Jellemzően a középre írt, nem túl nagy és nem túl kicsi számok felismerése volt a legsikeresebb
- **Következtetés:**
  - A felismerés javítása érdekében érdemes bevezetni egy képi előfeldolgozást, amely méretre skálázza és középre igazítja a rajzolt számjegyeket
  - A képi előfeldolgozáshoz először statisztikailag meg kell vizsgálnunk a betanításra használ adatok méret- és pozíció eloszlását, hogy saját rajzaink is ehhez igazodjanak

Tipikus felismerési problémák



A MNIST adatbázis számjegyeinek méreteloszlása

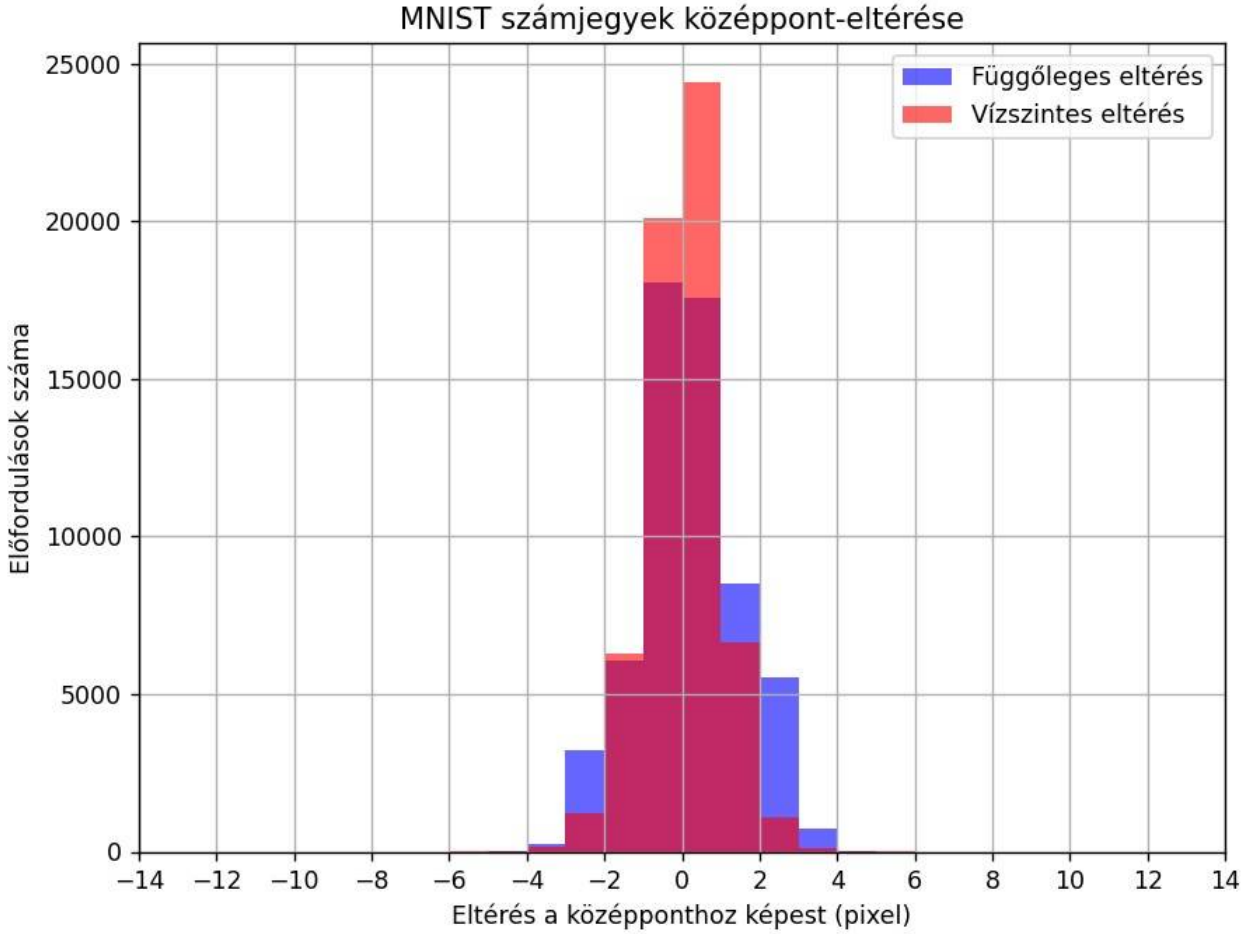


Az **MNIST_atlagmeret.py** segédprogram segítségével meghatároztuk a tanításhoz használt számjegyek befoglaló téglalapját (ezt tekintjük a karakter méretének)

Az ábrán a számjegyek méreteloszlását mutatjuk be (az ábrát is a fenti program készítette)

Jól látható, hogy a legtöbb számjegy ~20 pixel magasságú, így a saját rajzokat is ehhez célszerű igazítani a felismerés optimalizálásához

A MNIST adatbázis számjegyeinek pozícióeloszlása



Az **MNIST_shiftcheck.py** segédprogram segítségével meghatároztuk a tanításhoz használt számjegyek (befoglaló téglalapjának) a középpontját

Az ábrán a számjegyek középpontjainak pozícióeloszlását mutatjuk be a 28x28-as képmátrix közepéhez képest (az ábrát is a fenti program készítette)

Jól látható, hogy a számjegyek gondosan középre igazítottak, így a saját rajzokat is célszerű középre igazítani, hogy javítsuk a felismerés pontosságát

Képi előfeldolgozás

```python
import numpy as np                                  # telepítése: pip install numpy
import cv2                                          # telepítése: pip install opencv-python
def preprocess_image(image_matrix):                 # Középre igazítás és 20 pixel magasságra skálázás
    # Bounding box meghatározása (hol van a számjegy a képen?)
    rows = np.any(image_matrix, axis=1)
    cols = np.any(image_matrix, axis=0)
    r_min, r_max = np.where(rows)[0][[0, -1]]
    c_min, c_max = np.where(cols)[0][[0, -1]]

    # 2. Kivágjuk a számjegyet (minden más háttér eltávolítása)
    digit = image_matrix[r_min:r_max+1, c_min:c_max+1]
    height, width = digit.shape

    if height < 20 or height > 22:                  # 3. Skálázás, ha túl nagy vagy túl kicsi a magasság
        scale_factor = 20 / height
        new_width = int(width * scale_factor)       # Arányosan növeljük/kicsinyítjük a szélességet
        new_height = int(height * scale_factor)     # és a magasságot
        digit = cv2.resize(digit, (new_width, 20), interpolation=cv2.INTER_AREA)
        height, width = digit.shape                 # refresh size

    pad_y = (28 - height) // 2                      # 4. Középre igazítás
    pad_x = (28 - width) // 2
    digit = np.pad(digit,((pad_y,28-height-pad_y),(pad_x,28-width-pad_x)),mode='constant',constant_values=0)
    return digit
```

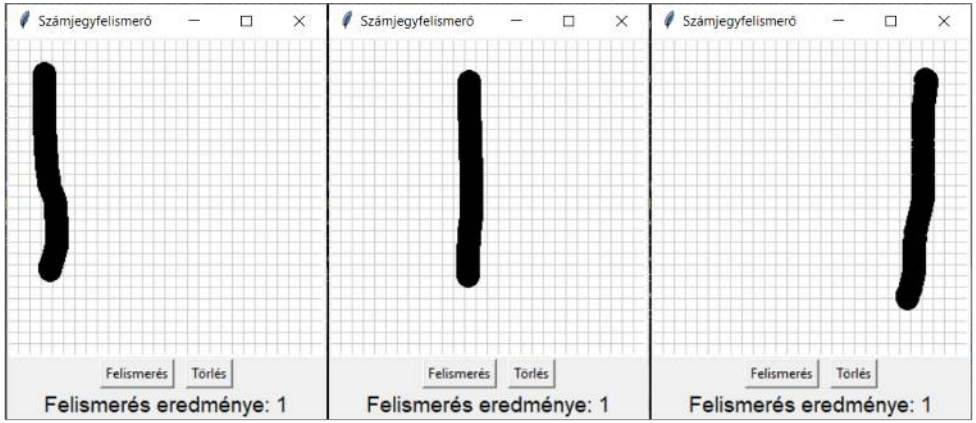
Javított képfelismerés: recognize_digit3.py

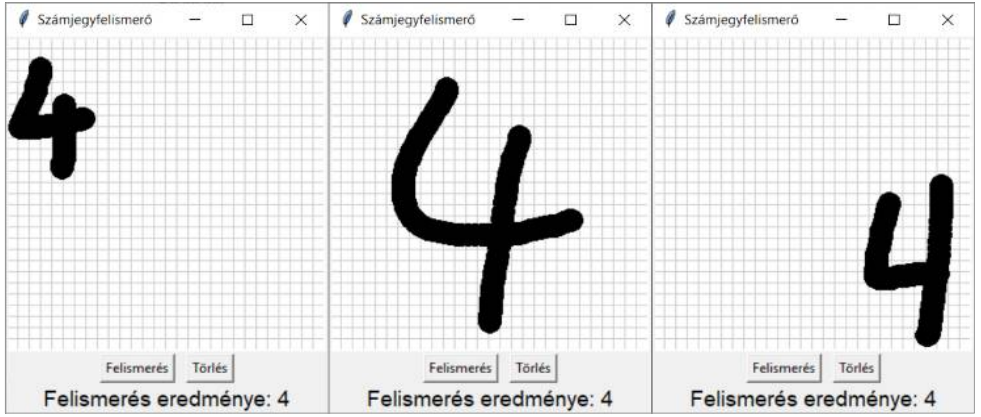
- Az előző oldalon bemutatott képi előfeldolgozással kiegészítettük a korábbi számjegyfelismerő programunkat. A **preprocess_image()** függvényt a **„Felismerés”** nyomógomb kattintását kiszolgáló **process_image()** függvényben kell meghívni, közvetlenül az **image_matrix** CNN kompatibilis formára alakítása előtt

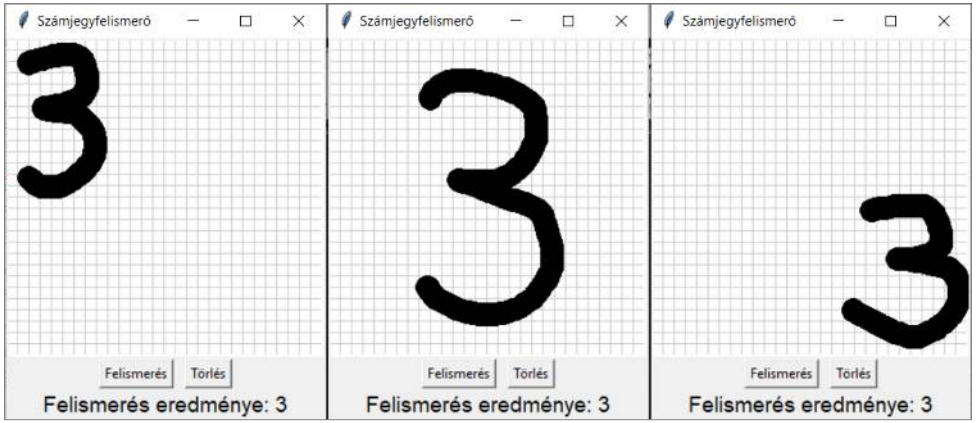
```
def process_image(self):
    image_matrix = self.get_image_matrix()
    if np.sum(image_matrix) < 10:                    # Ha túl kicsi a rajz, figyelmeztet
        self.result_label.config(text="Rajzolj nagyobb számjegyet!")
        return

    image_matrix = preprocess_image(image_matrix) # Skálázás és eltolás - ha kell

    # **Átalakítás CNN kompatibilis formátumra**
    image_matrix = np.expand_dims(image_matrix, axis=-1)  # (28, 28, 1)
    image_matrix = np.expand_dims(image_matrix, axis=0)   # (1, 28, 28, 1)
    prediction = model.predict(image_matrix)      # Felismerés: Előrejelzés a modellen
    predicted_digit = np.argmax(prediction)
```

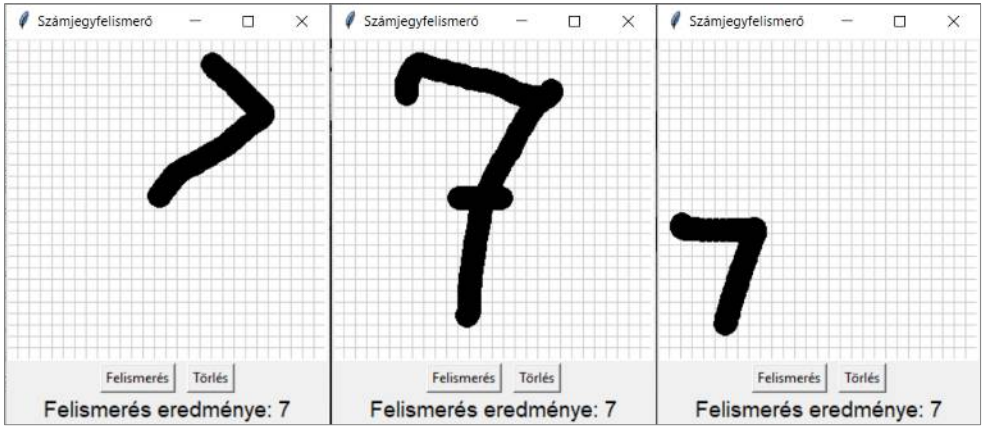
recognize_digit3.py futási eredménye

- A skálázással és középre igazítással lényeges javulást sikerült elérni a CNN modell számjegyfelismerésében, bár még mindig előfordulnak „félreolvasások”









Hogyan ellenőrizzük a modelljeinket?

- A modellek teszteléséhez elsősorban tesztadatok kellenek:
  - Az **MNIST adatbázis** 10 000 felcímkézett tesztképe független a betanítási adatoktól
  - Magunk és készíthetünk felcímkézett tesztadatokat a mellékelt **minta_gyartas.py** program segítségével (ilyen tesztadatsor található a **custom_dataset.npz** fájlban). A képminták gyártása közben azt is láthatjuk, hogy a képi előfeldolgozás mit csinál a rajzainkból a képi előfeldolgozás során
- A számszerű kiértékelést legegyszerűbben úgy végezhetjük el, ahogy a betanítást végző **digit_trainer2.py** program végén láthatjuk, ahol a betanított modell pontosságát egy `loss, accuracy = model.evaluate(x_test, y_test)` programsorral ellenőrizzük. A következő oldalon bemutatjuk az ehhez készített **modell_teszteles.py** programot
- Érdemes vizuálisan is elemezni a felismert és fel nem ismert számjegyeket, hogy lássuk, milyen rajzolati eltérések okozhatnak problémát. A **modell_teszteles_keppel4.py** program a fentemlített **custom_dataset.npz** fájl 45 db adatát az **MNIST** adatbázis 10 000 tesztadatából véletlenszerűen kiválasztott 55 adatával egészíti ki, s a felismerés lefuttatása után a számjegyek képeit vizuálisan is megjeleníti

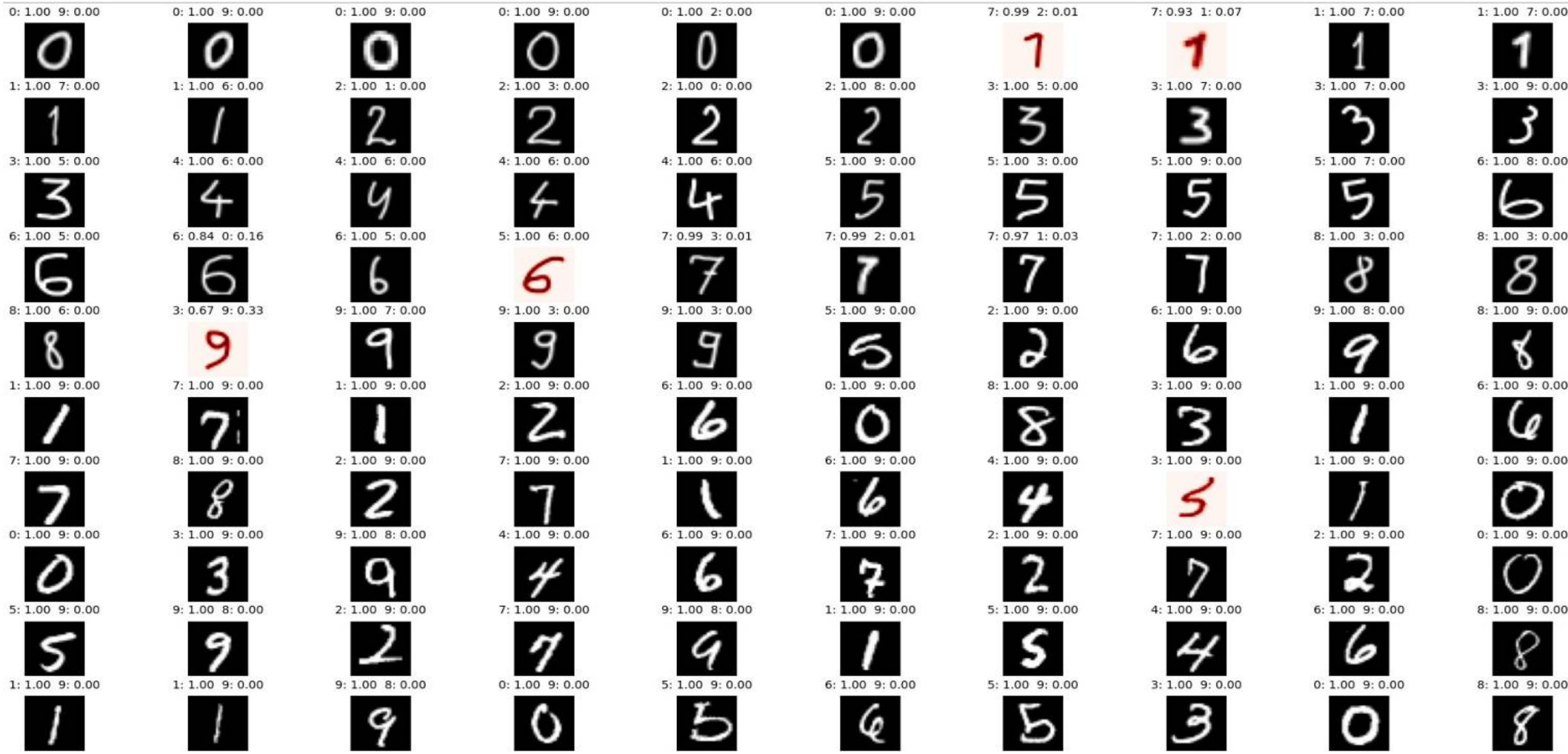
modell_teszteles.py

- Az alábbi programmal a saját készítésű **custom_dataset.npz** teszt adatsor felismerését ellenőriztük a betanított CNN modell segítségével

```
import numpy as np
from tensorflow.keras.models import load_model

# **Betöltjük az előre tanított modellt**
model = load_model("konvolucios_modell20.h5")

# **Betöltjük a mentett tesztadatokat**
data = np.load("custom_dataset.npz")
X_test = data["images"]  # Képek
y_test = data["labels"]  # Címkék

# **Átalakítás CNN kompatibilis formátumra**
X_test = np.expand_dims(X_test, axis=-1)  # (N, 28, 28, 1)

# **Felismerés és pontosság kiírása**
test_loss, test_acc = model.evaluate(X_test, y_test, verbose=2)
print(f"Teszt pontosság: {test_acc:.4f}")
```

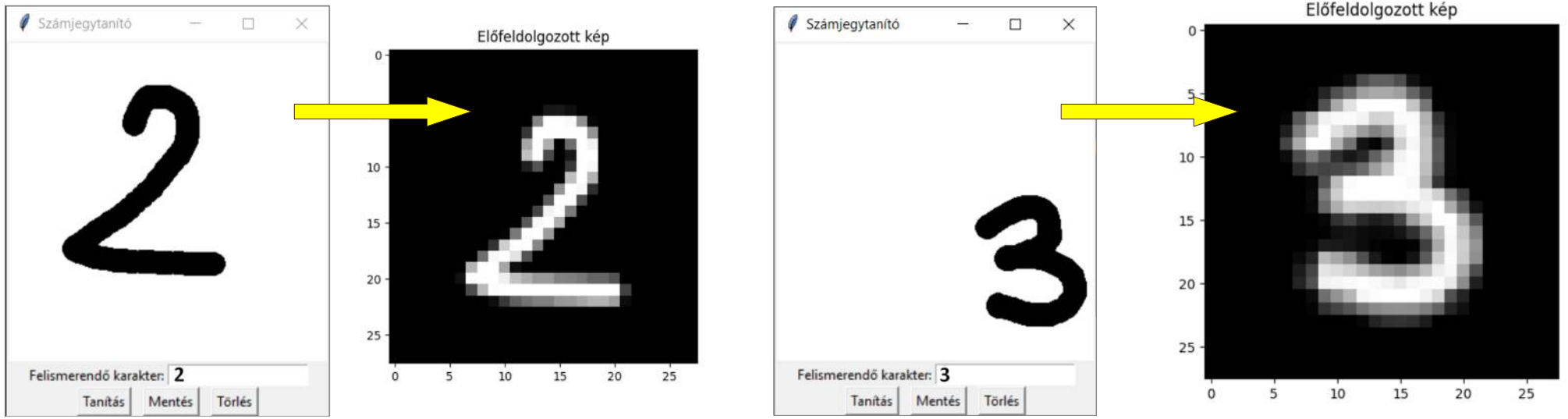
```
Modell betöltve!
2/2 - 0s - 130ms/step - accuracy: 0.9111 - loss: 0.4299
Teszt pontosság: 0.9111
```

modell_teszteles_keppel4.py



minta_gyartas.py

- Ez a program a már ismert Tkinter-alapú rajzfelületet biztosítja a számjegyek kézi rajzolásához, majd a képet 28x28-as mátrixba konvertálja, és középre igazítja, illetve skálázza, ha szükséges a jobb felismerés érdekében. A beíró rovatban meg kell adni a rajzolt számjegy értékét, a Tanítás gombbal a felcimkézett képet elmentjük a memóriába, a mentés gombbal pedig a **custom_dataset.npz** fájlba mentjük az adatokat



A mellékelt programok jegyzéke

digit_trainer2.py →	konvolucios_modell5.h5 konvolucios_modell20.h5	A konvolúciós modellünk betanító programja (A MNIST adatbázis 60000 + 10000 mintáját használja)
recognize_digit2.p ←	konvolucios_modell5.h5 konvolucios_modell20.h5	A konvolúciós modellt használó számjegyfelismerő program
MNIST_atlagmeret.py		Az MNIST minták méreteloszlását számítja ki
MNIST_shiftcheck.py		Az MNIST minták pozícióeloszlását számítja ki
recognize_digit3.py ←	konvolucios_modell5.h5 konvolucios_modell20.h5	A konvolúciós modellt és a képi előfeldolgozást használó számjegyfelismerő program
minta_gyartas.py →	custom_dataset.npz	Saját számjegyminták készítése
modell_teszteles.py	custom_dataset.npz ← konvolucios_modell5.h5 konvolucios_modell20.h5	A konvolúciós modell pontosságát méri
modell_teszteles_keppel4.py	custom_dataset.npz ← konvolucios_modell5.h5 konvolucios_modell20.h5	A képfelismerések vizuális ellenőrzése