DIGITAL MATHEMATICS APPLIED IN DEFENCE AND SECURITY EDUCATION (DIMAS)

KA220-HED - Cooperation Partnerships in Higher Education

2023-1-BG01-KA220-HED-000156664

Educational resource

(A3.2)

May 2025

SCENARIO 5.7

Title	**Neural Pathways: Using Neural Networks to Decode Patterns in Intelligence Data**
Short description	Students take on the role of data scientists in a defense agency tasked with classifying security-relevant signals (e.g., intercepted communications, activity logs, or sensor data) using a neural network model. Their mission is to design, train, and evaluate a neural network to recognize patterns that indicate potential security threats, learning the underlying math concepts of AI, including matrix operations, activation functions, and optimization techniques. This scenario bridges mathematics, AI, and national security, encouraging students to grasp the math behind modern machine learning techniques like neural networks.
Topics Involved	Artificial Intelligence & Computer Science Topics: Structure of Neural Networks/ Supervised Learning/ Model Evaluation/ Data Preprocessing
Areas of the mathematics	▪ Linear Algebra (Vectors, matrices, and matrix multiplication, Dot products (used in neural network layer calculations), Transpose and dimensions in matrix operations) ▪ Functions and Graphs (Understanding activation functions like: Sigmoid, ReLU (Rectified Linear Unit), Softmax ▪ Calculus (Conceptual/Applied) (Partial derivatives (used in gradient descent, Chain rule (applied in backpropagation) ▪ Statistics & Probability (Understanding model outputs as probabilities, Loss functions like binary crossentropy, Accuracy, precision, recall, F1-score) ▪ Optimization Techniques (Gradient descent, Learning rates and convergence, Overfitting vs. underfitting)
Digital mathematics tools	Python, TensorFlow / Keras, Scikit-learn
Learning objectives (knowledge, abilities,	**Knowledge:** ▪ Understand the mathematical foundation of neural networks, including:

competencies)	- Matrix multiplication, dot product, and vector transformations. - The role of weights, biases, and activation functions (e.g., ReLU, sigmoid). ▪ Grasp the concepts of forward propagation, backpropagation, and gradient descent. ▪ Know how loss functions (e.g., binary cross-entropy) guide learning. ▪ Understand how neural networks are applied in real-world classification tasks such as security threat detection. **Abilities:** ▪ Construct and train a basic feedforward neural network using Python and Keras/TensorFlow. ▪ Apply data preprocessing techniques (normalization, encoding) to prepare inputs for neural networks. ▪ Visualize and interpret model training results using graphs (e.g., loss vs. epochs, confusion matrix). ▪ Tune neural network hyperparameters (learning rate, number of layers/neurons) to improve model performance. **Responsibility and autonomy:** ▪ Model real-world security challenges mathematically using AI tools and techniques. ▪ Use critical thinking to evaluate neural network behavior and make data-driven adjustments. ▪ Communicate complex mathematical and computational results through clear explanations and visualizations. ▪ Reflect on the ethical implications of using neural networks in defense and surveillance contexts (e.g., fairness, privacy, bias).
Methodologies adopted	1. Project-Based Learning (PBL) • Students work through a real-world-inspired challenge: building a neural network to detect patterns in simulated intelligence data. • Encourages deep engagement, self-direction, and applied problem-solving. 2. Mathematical Modeling • Students translate real-world security scenarios into mathematical models, including: o Linear algebra (matrix operations for neural layers) o Calculus (gradient descent, derivatives) o Probability (output interpretation as risk or threat levels) 3. Inquiry-Based Learning • Students ask and investigate questions like:

	○ *What patterns in the data suggest a potential threat?* ○ *How does changing the activation function impact output?* ○ *Why does the model misclassify certain signals?* 4. Computational Experimentation • Students iteratively test and improve neural networks using Python tools. • They tune hyperparameters, observe outcomes, and learn through simulation. • Promotes understanding of math through trial, error, and analysis. 5. Collaborative Learning • Students work in small teams to: ○ Design and implement their models. ○ Compare results and strategies. ○ Discuss strengths and limitations of different neural architectures. 6. Visualization & Interpretation • Heavy use of data visualization tools (e.g., Matplotlib, Seaborn) to interpret: ○ Weight distributions ○ Loss/accuracy curves ○ Model predictions (e.g., heatmaps, confusion matrices.
Prerequisites	Mathematical Knowledge: Linear Algebra, Functions and Graphs, Introductory Calculus (Conceptual Level), Statistics and Probability Programming Skills (Python Basics, xperience with Jupyter Notebooks or Google Colab, Introductory Machine Learning)
Estimated time	8 hours (including self-studies and teamwork)
Task for students	**Mission Brief** Students are part of a **data science team** working for a **national research laboratory** developing intelligent vision systems. Their mission is to design and train a **neural network** capable of recognizing **handwritten digits (0–9)** from scanned or captured images — a fundamental step in developing automated document reading and military intelligence systems. The objective is to **model, train, and evaluate** a machine learning system that can accurately distinguish between different handwritten numbers.

	TASK 1. Data Exploration & Preparation
	
	• Load a provided dataset of handwritten digits (e.g., **MNIST** or a similar dataset).
	• Visualize and explore sample images to understand their structure (grayscale values, resolution, etc.).
	• Normalize the image data (e.g., scale pixel values between 0 and 1).
	• Split the dataset into **training** and **testing** subsets.
	
	TASK 2. Model Architecture
	
	• Define a **feedforward neural network** using **TensorFlow/Keras**.
	  ○ **Input layer:** matches the number of pixels per image (e.g., 28×28 = 784 features for MNIST).
	  ○ **Hidden layers:** 1–2 layers with an appropriate number of neurons and activation functions (e.g., ReLU).
	  ○ **Output layer:** 10 neurons (one for each digit 0–9) with **softmax** activation for multiclass classification.
	
	TASK 3. Training the Network
	
	• Train the model on the training dataset using suitable parameters:
	  ○ **Loss function:** categorical cross-entropy
	  ○ **Optimizer:** Adam or stochastic gradient descent (SGD)
	  ○ **Metrics:** accuracy and loss
	• Visualize training progress (accuracy and loss per epoch).
	• Use the test data to evaluate model performance.
	• Generate and analyze **accuracy and loss curves** over training epochs.

Assessment	Mathematical Understanding 25% Model Implementation 20% Data Preparation & Processing 10% Evaluation & Analysis 15% Evaluation & Analysis 20% Communication of Results 10%

Solution

Neural networks are adaptive black-box systems that allow the extraction of a model from data through a learning process.

Essentially, the architecture of a neural network is characterized by the number of layers, the number of neurons per layer, a training algorithm, and an operation algorithm. The operation algorithm transforms the input data into meaningful output data depending on the network's purpose, while the training algorithm instructs the neural network on how to acquire new knowledge based on the input data.

The network is trained on a dataset that can either be directly classified or requires the computation of certain features before starting the classification process.

The process by which a neural network "learns" to recognize a pattern can follow two main approaches:

- the network learns to produce a specific pattern based on the input dataset each time a particular input set is applied;
- the network detects the periodicity with which neurons learn to respond to certain properties of input patterns. This learning paradigm is essential for feature discovery and knowledge representation.

Structurally, a neural network is composed of a series of interconnected functional units (neurons).Regardless of the number of neurons, a neural network contains the following elements:

- input signals $y_1,y_2,...,y_n$
- synaptic weights $w_1,w_2,...,w_n$
- threshold w_0
- output y

After the training stage, a neural network acts as an "expert" in analyzing data sets. Therefore, the use of a neural network offers several advantages:

- the ability to learn to perform a task based on the data provided during training;
- the possibility to update its training process based on newly obtained data (in the case of adaptive networks).

To design a neural network, the following steps are followed:

- choosing the architecture (number of layers, number of units per layer, type of interconnections, activation functions);
- training (determining the weight values from the training set using a learning algorithm);

- Validating the network (analyzing the network's behaviour on data that are not part of the training set).

To build neural networks (both classical and convolutional), the TensorFlow library will be used. This library provides a user-friendly interface, although the internal network details cannot be directly modified (e.g., the optimization algorithm cannot be changed).

Task 1

In this scenario, we will experiment with two neural networks: a classical one and a convolutional one. The problem to be solved is the classification of handwritten digits. We begin by installing the tensorflow_datasets library to work more easily with the corresponding dataset.

```
import tensorflow as tf
import tensorflow_datasets as tfds
import math
import numpy as np
import matplotlib.pyplot as plt
```

With the library installed earlier (tensorflow_datasets), we can easily access the corresponding dataset.

```
dataset, metadata = tfds.load('mnist', as_supervised=True, with_info=True)
train_dataset, test_dataset = dataset['train'], dataset['test']
```

Below are displayed details about the dataset used. These details, as previously shown, are found in the metadata downloaded together with the dataset itself.

```
num_train_examples = metadata.splits['train'].num_examples
num_test_examples = metadata.splits['test'].num_examples
print("Number of training examples: {}".format(num_train_examples))
print("Number of test examples:     {}".format(num_test_examples))
```

Next, we define a function that takes the image and its label as arguments and returns the normalized image. The cast function converts the image to the float32 format to allow the division operation to be performed.

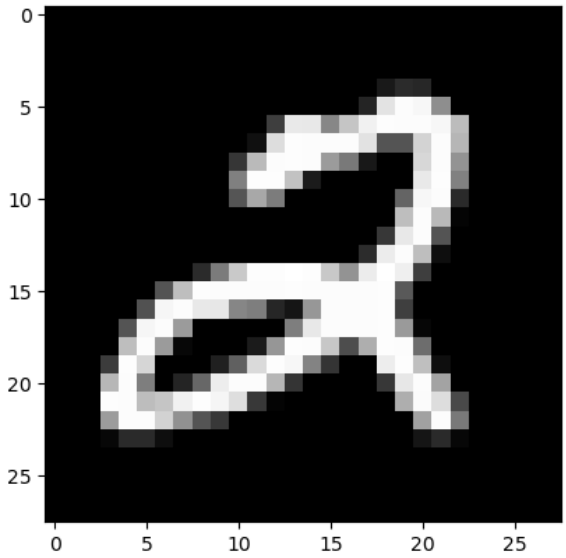
```
def normalize(images, labels):
  images = tf.cast(images, tf.float32)
  images /= 255
  return images, labels
print(type(train_dataset))
```

The previously created function is applied to each image in the dataset. This is done using the map function.

```
train_dataset =  train_dataset.map(normalize)
test_dataset  =  test_dataset.map(normalize)
train_dataset =  train_dataset.cache()
test_dataset  =  test_dataset.cache()
print(type(train_dataset))
```

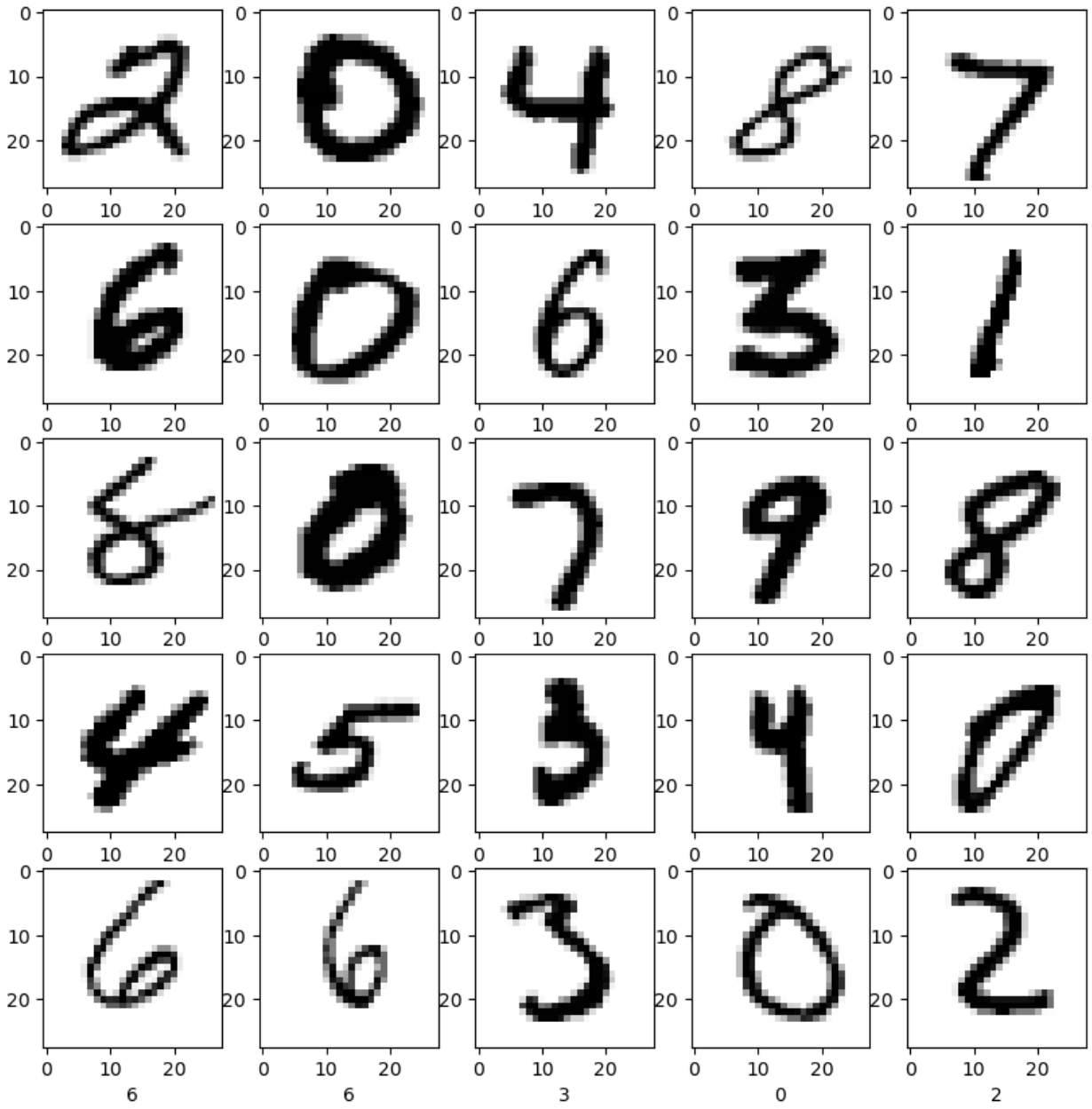
```
import os
os.environ["KMP_DUPLICATE_LIB_OK"]="TRUE"

plt.gcf().set_size_inches(15, 15)
for image, label in test_dataset.take(1):
  print(type(image))
  break
image = image.numpy().reshape((28,28))
plt.figure()
plt.imshow(image, cmap='gray')
plt.show()
```



Below are displayed 25 images from the test dataset along with their corresponding labels.

```
plt.gcf().set_size_inches(15, 15)
plt.figure(figsize=(10,10))
i = 0
for (image, label) in test_dataset.take(25):
    image = image.numpy().reshape((28,28))
    plt.subplot(5,5,i+1)
    plt.imshow(image, cmap=plt.cm.binary)
    plt.xlabel(str(label.numpy()))
    i += 1
plt.show()
```



Task 2

After defining the network (model), it is "compiled." This operation should not be interpreted in the traditional sense of the word. At this stage, the optimization algorithm used by the network (Adam), the loss function (SparseCategoricalCrossentropy), and the evaluation metric (accuracy) are specified.

As previously mentioned, the last layer represents the output of the network. Each neuron reflects the probability that the output corresponds to a particular digit. This output must be evaluated to determine how far it is from the desired output. To do this, an algorithm is required—here, it is SparseCategoricalCrossentropy.

Additionally, a metric is needed to evaluate the network's performance on the test dataset.

A common choice is **accuracy**, which represents the ratio between the number of correctly classified images and the total number of classified images (both correct and incorrect).

```
model.compile(optimizer='adam',
              loss=tf.keras.losses.SparseCategoricalCrossentropy(from_logits=True),
              metrics=['accuracy'])
```

The problem with the Gradient Descent algorithm lies in the weight updates performed after processing each individual image. This approach results in very long training times. For this reason, in practice, the BATCH variant is used. In the example below, the BATCH_SIZE is 32.This means that the dataset will be processed in chunks of 32 samples for the purpose of updating the weights.

Task 3

Intuitively, after each image is processed, the error between the networks's output and the correct output is calculated. Based on this error, the network's weights are updated. If the weights were updated after every single image, the training process would take a very long time. The solution is to update the weights after a certain number of images have been processed — for example, 32 images. In other words, the errors from 32 images are averaged, and based on this average, the weights are updated.

```
BATCH_SIZE = 32
train_dataset = train_dataset.cache().repeat().shuffle(num_train_examples).batch(BATCH_SIZE)
test_dataset = test_dataset.cache().batch(BATCH_SIZE)
history = model.fit(train_dataset, epochs=5, steps_per_epoch=math.ceil(num_train_examples/BATCH_SIZE))
model.summary()
```

The number of epochs represents the number of iterations through the training set. In this case, it is 5.

```
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten (Flatten)           (None, 784)               0

 dense (Dense)               (None, 128)               100480

 dense_1 (Dense)             (None, 10)                1290

=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
```

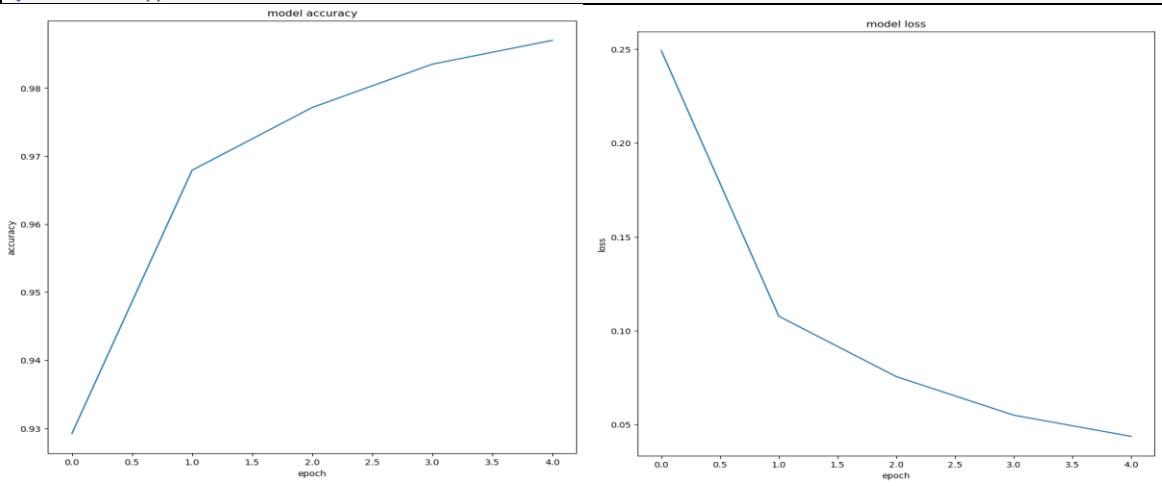
The network's accuracy is evaluated using the evaluate function.

```
test_loss,         test_accuracy         =         model.evaluate(test_dataset,
steps=math.ceil(num_test_examples/BATCH_SIZE))
print('Accuracy on test dataset:', test_accuracy)
```

```
313/313 [==============================] - 2s 4ms/step - loss: 0.0751 - accuracy:
0.9767
Accuracy on test dataset: 0.9767000079154968
```

The history variable contains the entire training history. For each training epoch, the accuracy achieved and the error are recorded. These are displayed below.

```
print(history.history.keys())
plt.gcf().set_size_inches(10, 10)
plt.plot(history.history['accuracy'])
plt.title('model accuracy')
plt.ylabel('accuracy')
plt.xlabel('epoch')
plt.show()
plt.gcf().set_size_inches(10, 10)
plt.plot(history.history['loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.show()
```



Using the predict function, images from the test dataset are classified.

```
for test_images, test_labels in test_dataset.take(1):
  test_images = test_images.numpy()
  test_labels = test_labels.numpy()
  predictions = model.predict(test_images)
print(predictions.shape)
```

Depending on the idx variable, you can see how certain images were classified and with what probability.

```
plt.gcf().set_size_inches(10, 10)
idx = 15
image = test_images[idx].reshape((28,28))
plt.figure()
plt.imshow(image, cmap='gray')
plt.xlabel(str(np.argmax(predictions[idx])) + ' cu probabilitatea de ' +
str(predictions[idx][np.argmax(predictions[idx])] * 100))
```