Term: M2-EE

Semester: 3

AY: 2025-2026

Abdelbacet Mhamdi

Dr.-Ing. in Electrical Engineering

Senior Lecturer at ISET Bizerte

abdelbacet.mhamdi@bizerte.r-iset.tn

Artificial Intelligence - Part 3

LAB MANUAL

Institute of Technological Studies of Bizerte

Available @ https://github.com/a-mhamdi/jlai/

Honor code

The University of North Carolina at Chapel Hill
Department of Physics and Astronomy
http://physics.unc.edu/undergraduate-program/labs/general-info/

"During this course, you will be working with one or more partners with whom you may discuss any points concerning laboratory work. However, you must write your lab report, in your own words.

Lab reports that contain identical language are not acceptable, so do not copy your lab partner's writing.

If there is a problem with your data, include an explanation in your report. Recognition of a mistake and a well-reasoned explanation is more important than having high-quality data, and will be rewarded accordingly by your instructor. A lab report containing data that is inconsistent with the original data sheet will be considered a violation of the Honor Code.

Falsification of data or plagiarism of a report will result in prosecution of the offender(s) under the University Honor Code.

On your first lab report you must write out the entire honor pledge:

The work presented in this report is my own, and the data was obtained by my lab partner and me during the lab period.

On future reports, you may simply write "*Laboratory Honor Pledge*" and sign your name."

Contents

In order to activate the virtual environment and launch **Jupyter Notebook**, you need to proceed as follow

① Press simultaneously the keys CTRL ALT and T on the keyboard;

② Type `jlai3` in the console prompt line;

Terminal

```
student@isetbz:~$ jlai3
```

③ Finally hit the ↵ key.

KEEP THE SYSTEM CONSOLE OPEN.

▼ Remark 1

You should be able to utilize Julia from within the notebook through:

Jupyter Lab *at http://localhost:2468*

Pluto *at http://localhost:1234*

Submit your lab reports in **PDF** format. Be sure to review your files with your instructor before the lab session ends.

https://forms.gle/sHQgBigF6hFFYqD16

Please use one of the provided templates when preparing your lab assessments:

LaTeX `https://www.overleaf.com/read/pwgpyvcxcvym#9e34eb`

Typst `https://typst.app/universe/package/ailab-isetbz`

A. Mhamdi

1 Natural Language Processing

Student's name	. . . . .	. . . . .	. . . . .
Score /20	. . . . .	. . . . .	. . . . .

Detailed Credits

Anticipation *(4 points)*	. . . . .	. . . . .	. . . . .
Management *(2 points)*	. . . . .	. . . . .	. . . . .
Testing *(7 points)*	. . . . .	. . . . .	. . . . .
Data Logging *(3 points)*	. . . . .	. . . . .	. . . . .
Interpretation *(4 points)*	. . . . .	. . . . .	. . . . .

The notebook is available at `https://github.com/a-mhamdi/jlai/` ⟶ *Codes* ⟶ *Julia* ⟶ *Part-3* ⟶ *nlp* ⟶ *nlp.ipynb*

The following are the key steps that outline the general process for Natural Language Processing (**NLP**) tasks in `Julia`:

- Load and preprocess the text data: clean *(e.g., remove punctuation, lowercase)*, tokenize *(split into words/phrases)*, and encode *(e.g., word embeddings)*.
- Choose an **NLP** model and set key hyperparameters *(e.g., HMMs, CRFs, transformer-based models)*.
- Train the model on preprocessed data using an optimizer *(e.g., stochastic gradient descent)* to minimize a loss.
- Evaluate the model on a separate test set.
- Use the trained model to make predictions on unseen text.

```
1 using Markdown
```

```
using TextAnalysis

txt = "The quick brown fox is jumping over the lazy dog" # Pangram [modif.]

md"Create a `Corpus` using `txt`"
crps = Corpus([StringDocument(txt)])
lexicon(crps)
update_lexicon!(crps)
lexicon(crps)
lexical_frequency(crps, "fox")

md"Create a `StringDocument` using `txt`"
sd = StringDocument(txt)
md"Get a smaller set of words `text(sd)`"
prepare(sd, strip_articles | strip_numbers | strip_punctuation | strip_case | strip_whitespace)
stem(sd)

md"Get the tokens of `sd`"
the_tokens = tokens(sd)

md"Get the stemmed tokens of `sd`"
stemmer = Stemmer("english")
stemmed_tokens = stem(stemmer, the_tokens)

println("Original tokens: ", the_tokens)
println("Stemmed tokens: ", stemmed_tokens)

md"**Part-of-speech tags**"

#=
Common POS tags:

JJ: Adjective
NN: Noun, singular or mass
NNS: Noun, plural
VB: Verb, base form
VBZ: Verb, 3rd person singular present
VBG: Verb, gerund or present participle
VBD: Verb, past tense
RB: Adverb
IN: Preposition or subordinating conjunction
DT: Determiner
```

```
PRP: Personal pronoun
CC: Coordinating conjunction
=#

#=
using TextModels
pos = PoSTagger()
pos(crps)
=#

md"**Word embeddings**"
using Embeddings
embtab = load_embeddings(GloVe{:en}, max_vocab_size=5)

embtab.vocab
embtab.embeddings

glove = load_embeddings(GloVe{:en}, 3, max_vocab_size=10_000)
const word_to_index = Dict(word => ii for (ii,word) in enumerate(glove.
 ↪vocab))
function get_word_vector(word)
    idx = word_to_index[word]
    return glove.embeddings[:, idx]
end

using LinearAlgebra
function cosine_similarity(v1::Vector{Float32}, v2::Vector{Float32})
    return *(v1', v2) / *(norm(v1), norm(v2))
end

md"_e.g. - \"king\" - \"man\" + \"woman\"   \"queen\"_"
king = get_word_vector("king")
queen = get_word_vector("queen")
man = get_word_vector("man")
woman = get_word_vector("woman")

cosine_similarity(king - man + woman, queen)

md"_e.g. - \"Madrid\" - \"Spain\" + \"France\"   \"Paris\"_"
Madrid = get_word_vector("madrid")
Spain = get_word_vector("spain")
France = get_word_vector("france")
Paris = get_word_vector("paris")

```

```
cosine_similarity(Madrid - Spain + France, Paris)

md"**Text classification**"
md"https://github.com/JuliaText/TextAnalysis.jl/blob/master/docs/src/classify.md"
m = NaiveBayesClassifier([:legal, :financial])
fit!(m, "this is financial doc", :financial)
fit!(m, "this is legal doc", :legal)
predict(m, "this should be predicted as a legal document")

md"**Semantic analysis**"
m = DocumentTermMatrix(crps)

md"*Latent Semantic Analysis*"
lsa(m)

md"*Latent Dirichlet Allocation*"
k = 2               # number of topics
iterations = 1000   # number of Gibbs sampling iterations
  = 0.1             # hyper parameter
   = 0.1            # hyper parameter
,    = lda(m, k, iterations,  ,  ) #
```

Task № 1:

In this lab, you will use `KNIME Analytics Platform` to perform topic modeling with **LSA** and **LDA**. You will preprocess a corpus, build a document–term matrix with TF or TF–IDF, apply LSA (SVD) and LDA with comparable topic counts, interpret topics via top terms, visualize document–topic structure, and compare coherence and interpretability across methods.

a) Load and explore the text corpus

- Load a document collection such as news articles, scientific abstracts, product reviews, or the 20 Newsgroups dataset;
- Examine the size of your corpus (number of documents) and document lengths;
- Visualize sample documents to understand the content and vocabulary;
- Identify the general themes or categories present in the corpus if labels are available;
- Note any preprocessing requirements such as removing HTML tags or special characters.

b) Preprocess the text data

- Convert all text to lowercase to ensure consistency;
- Tokenize the documents into individual words or terms;
- Remove stop words (common words like ”the”, ”is”, ”and” that carry little meaning);
- Apply stemming or lemmatization to reduce words to their root forms;
- Remove very rare terms (appearing in fewer than 2-3 documents) and very common terms (appearing in more than 80-90% of documents);
- Filter out punctuation, numbers, and special characters unless they are meaningful for your analysis.

c) Create the document-term matrix

- Construct a document-term matrix where rows represent documents and columns represent terms;
- Use term frequency (TF) or TF-IDF (Term Frequency-Inverse Document Frequency) weighting scheme;
- Understand that TF-IDF downweights common terms and emphasizes distinctive terms;
- Examine the dimensions of your matrix (number of documents × vocabulary size);
- Note the sparsity of the matrix (percentage of zero entries);
- Consider limiting vocabulary size to the top 1000-5000 most frequent terms for computational efficiency.

d) Apply Latent Semantic Analysis (LSA)

- Apply Singular Value Decomposition (SVD) to the document-term matrix;
- Choose the number of latent dimensions or topics (e.g., 10, 20, or 50 topics);
- Understand that LSA decomposes the matrix into document-topic and topic-term matrices;
- Extract the topic-term matrix to see which terms have high weights in each topic;
- For each discovered topic, identify the top 10-15 terms with the highest weights;
- Interpret and label each topic based on its most representative terms;
- Examine the document-topic matrix to see how documents are distributed across topics.

e) Apply Latent Dirichlet Allocation (LDA)

- Configure and run the LDA algorithm on the same preprocessed corpus;
- Set the number of topics to match your LSA experiment for fair comparison (e.g., 20 topics);
- Configure hyperparameters: alpha (document-topic density) and beta (topic-word density);

- Train the LDA model for sufficient iterations to ensure convergence;
- Extract the topic-word distributions to identify the most probable words for each topic;
- For each topic, display the top 10-15 words with their probabilities;
- Extract document-topic distributions showing the topic mixture for each document;
- Interpret and label each LDA topic based on its word distribution.

f) Compare LSA and LDA results

- Create side-by-side comparisons of topics discovered by LSA versus LDA;
- Evaluate topic coherence: do the top terms in each topic make semantic sense together;
- Check if LSA and LDA discovered similar themes or completely different topic structures;
- Examine whether LSA topics contain both positive and negative term weights while LDA topics contain only positive probabilities;
- Compare the interpretability of topics from both methods;
- Analyze document-topic distributions: are documents assigned to single topics (hard assignment) or distributed across multiple topics (soft assignment);
- Discuss the mathematical differences: LSA uses linear algebra (SVD) while LDA uses probabilistic generative modeling.

g) Evaluate and visualize the topic models

- Calculate topic coherence scores to quantitatively assess topic quality;
- Create visualizations such as word clouds for each topic showing term importance;
- Project documents into 2D space using the topic representations and create scatter plots;
- Color-code documents by their dominant topic or original category (if available);
- Examine whether documents from similar categories cluster together in topic space;
- Select sample documents and display their topic distributions from both LSA and LDA;
- Identify topics that are well-separated versus topics that overlap significantly.

h) Experiment with model parameters and analyze results

- Vary the number of topics (5, 10, 20, 50) and observe how this affects topic quality and interpretability;
- Experiment with different text preprocessing choices (with/without stemming, different stop word lists);
- Try different term weighting schemes (TF versus TF-IDF) and compare results;

- For LDA, experiment with different alpha and beta hyperparameters to control topic sparsity;
- Document how each parameter change affects the discovered topics;
- Compare computational time required for LSA versus LDA on your corpus;
- Discuss which method (LSA or LDA) worked better for your specific dataset and why;
- Explain scenarios where you would prefer LSA over LDA or vice versa based on your findings.

2 | Convolutional Neural Network

Student's name	....	....	....
Score /20	.....	.....	.....

Detailed Credits

Anticipation *(4 points)*	.....	.....	.....
Management *(2 points)*	.....	.....	.....
Testing *(7 points)*	.....	.....	.....
Data Logging *(3 points)*	.....	.....	.....
Interpretation *(4 points)*	.....	.....	.....

The notebook is available at https://github.com/a-mhamdi/jlai/ ⟶ *Codes* ⟶ *Julia* ⟶ *Part-3* ⟶ *cnn* ⟶ *cnn.ipynb*

CNN stands for Convolutional Neural Network, an architecture widely used for image and video recognition. A CNN is composed of multiple layers, each performing a specific transformation of the input data. The *input layers* receive raw data; the *output layers* produce final predictions. Between them, *hidden layers* (e.g., convolution, pooling, and fully connected layers) learn hierarchical feature representations. Convolutional Neural Networks are particularly powerful because they learn discriminative features directly from raw inputs, reducing the need for hand-engineered features.

```
using Markdown
md"Handwritten digits classification using **CNN**. This solution is implemented in `Julia` using the `Flux.jl` library"

using Statistics
using ProgressMeter: Progress, next!
using Plots

```

```
md"Import the machine learning library `Flux`"
using Flux # v0.14.25
using Flux: DataLoader
using Flux: onecold, onehotbatch

using MLDatasets
d = MNIST()

Base.@kwdef mutable struct HyperParams
      = 3f-3                # Learning rate
    batchsize = 64          # Batch size
    epochs = 8              # Number of epochs
    split = :train          # Split data into `train` and `test`
end

md"Load the **MNIST** dataset"
function get_data(; kws...)
    args = HyperParams(; kws...);
    md"Split and normalize data"
    data = MNIST(split=args.split);
    X, y = data.features ./ 255, data.targets;
    X = reshape(X, (28, 28, 1, :));
    y = onehotbatch(y, 0:9);
    loader = DataLoader((X, y); batchsize=args.batchsize, shuffle=true);
    return loader
end

train_loader = get_data();
test_loader = get_data(split=:test);

md"Transform sample training data to an image. View the image and check␣
 ↪the corresponding digit value."
idx = rand(1:6_000);
using ImageShow, ImageInTerminal # ImageView
convert2image(d, idx) # |> imshow
md"**Digit is $(d.targets[idx])**"

md"""
## **CNN** ARCHITECTURE"
The input `X` is a batch of images with dimensions `(width=28,␣
 ↪height=28, channels=1, batchsize)`
"""

fc = prod(Int.(floor.([28/4 - 2, 28/4 - 2, 16]))) # 2^{\# max-pool}
```

```
model = Chain(
            Conv((5, 5), 1 => 16, relu), # (28-5+1)x(28-5+1)x16 =
 ↪24x24x16
            MaxPool((2, 2)), # 12x12x16
            Conv((3, 3), 16 => 16, relu), # (12-3+1)x(12-3+1)x16 =
 ↪10x10x16
            MaxPool((2, 2)), # 5x5x16
            Flux.flatten, # 400
            Dense(fc => 64, relu),
            Dense(64 => 32, relu),
            Dense(32 => 10)
)

function train(; kws...)
    args = HyperParams(; kws...)
    md"Define the loss function"
    l( ,  ) = Flux.logitcrossentropy( ,  )
    md"Define the accuracy metric"
    acc( ,  ) = mean(onecold( ) .== onecold( ))
    md"Optimizer"
    optim_state = Flux.setup(Adam(args. ), model);

    vec_loss = []
    vec_acc = []

    for epoch in 1:args.epochs
        printstyled("\t***\t === EPOCH $(epoch) === \t*** \n", color=:
 ↪magenta, bold=true)
        @info "TRAINING"
        prg_train = Progress(length(train_loader))
        for (X, y) in train_loader
            loss, grads = Flux.withgradient(model) do m
                ŷ = m(X);
                l(ŷ, y);
            end
            Flux.update!(optim_state, model, grads[1]); # Upd `W` and
 ↪`b`
            # Show progress meter
            next!(prg_train, showvalues=[(:loss, loss)])
        end
        @info "TESTING"
        prg_test = Progress(length(test_loader))
        for (X, y) in test_loader
            ŷ = model(X);
```

```
            push!(vec_loss, l(ŷ, y));  # log `loss` value -> `vec_loss` vector
            push!(vec_acc, acc(ŷ, y)); # log `accuracy` value -> `vec_acc` vector
                    # Show progress meter
            next!(prg_test, showvalues=[(:loss, vec_loss[end]), (:accuracy, vec_acc[end])])
        end
    end
    return vec_loss, vec_acc
end

vec_loss, vec_acc = train()

# Plot results
plot(vec_loss, label="Test Loss")
plot(vec_acc, label="Test Accuracy")

# Let's make some predictions
idx = rand(1:1000, 16)
xs, ys = test_loader.data[1][:,:,:,idx], onecold(test_loader.data[2][:,idx]) .- 1
yp = onecold(model(xs)) .- 1

for i   eachindex(yp)
    @info "**Prediction is $(yp[i]). Label is $(ys[i]).**"
end

# Save the model
using BSON: @save
@save "cnn.bson" model
```

Task № 2:

> In this lab, you will build and train a Convolutional Neural Network (CNN) using `KNIME Analytics Platform` to classify images from the Fashion-MNIST dataset, which contains $70,000$ grayscale images of clothing items across 10 categories (T-shirt, Trouser, Pullover, Dress, Coat, Sandal, Shirt, Sneaker, Bag, Ankle boot). You will prepare the data, design a small CNN (Conv → Pool → Dense → Softmax), train it with `Adam` and `categorical cross-entropy`, and assess performance with accuracy curves and a confusion matrix. You will then iterate with regularization (dropout, batch normalization) and architectural variations to improve results.

a) Load and explore the Fashion-MNIST dataset

- Load the Fashion-`MNIST` dataset into your `KNIME` workflow;
- Visualize sample images from each of the 10 clothing categories;
- Split the dataset into training (60%), validation (20%), and test (20%) sets;
- Examine class distribution and apply stratified splitting.

b) Preprocess the image data

- Normalize pixel values to the range $[0, 1]$;
- Ensure images have shape $28 \times 28 \times 1$ for **CNN** input;
- Convert labels to one-hot vectors for multi-class classification;
- Optionally apply augmentation *(rotation, flipping, zoom)* to increase diversity.

c) Design and build the **CNN** architecture

- Use two convolutional layers (32 and 64 filters) with 3×3 kernels and `ReLU`;
- Add 2×2 max pooling after each convolutional layer;
- Flatten and add a dense layer with 128 units and `ReLU`;
- Add an output layer with 10 units and `softmax` for classification;
- Configure `categorical cross-entropy` loss and the `Adam` optimizer.

d) Train and evaluate the **CNN** model

- Train for 20 epochs with batch size 32, monitoring validation accuracy and loss;
- Generate predictions on the test set and compute overall accuracy;
- Create a confusion matrix to analyze performance across all classes;
- Plot training/validation accuracy and loss to detect overfitting or underfitting;
- Identify frequently misclassified categories and discuss likely causes.

e) Experiment with model improvements

- Add dropout to reduce overfitting;
- Vary depth, number of filters, and filter sizes to compare architectures;
- Try batch normalization to stabilize and speed up training;
- Record each experiment with its test accuracy and key settings;
- Select the best model and justify why it performs best.

3 | Transfer Learning

Student's name	. . . . .	. . . . .	. . . . .
	. . . . .	. . . . .	. . . . .
	. . . . .	. . . . .	. . . . .
Score /20	. . . . .	. . . . .	. . . . .

Detailed Credits

Anticipation *(4 points)*	. . . . .	. . . . .	. . . . .
Management *(2 points)*	. . . . .	. . . . .	. . . . .
Testing *(7 points)*	. . . . .	. . . . .	. . . . .
Data Logging *(3 points)*	. . . . .	. . . . .	. . . . .
Interpretation *(4 points)*	. . . . .	. . . . .	. . . . .

The notebook is available at `https://github.com/a-mhamdi/jlai/` ⟶ *Codes* ⟶ *Julia* ⟶ *Part-3* ⟶ *transfer-learning* ⟶ *transfer-learning-*.ipynb*

Transfer learning is a machine learning technique in which a model trained on one task is repurposed for a second, related task. It involves taking a *pre-trained model*, which has already learned useful features, and adapting it to a new task. Transfer learning can be a useful approach when there is not enough data to train a model from scratch, or when the new task closely resembles the original task.

There are several ways to use transfer learning:

1. fine-tuning the weights of the pre-trained model on the new task;
2. using the pre-trained model as a fixed feature extractor;
3. using the pre-trained model as a starting point and training a new model from there.

Transfer learning is a common approach in deep learning, and has been used to achieve state-of-the-art results on a variety of tasks, such as image classification and natural language processing.

```
using Markdown
```

```
2
3  using Metalhead
4  md"Load the pre-trained model"
5  md"[API Reference](https://fluxml.ai/Metalhead.jl/dev/api/reference/
    ↪#API-Reference)"
6  resnet = ResNet(18; pretrain=true).layers;
7
8  using Flux
9  using Flux: onecold, onehotbatch
10
11 mdl = Chain(
12     resnet[1:end-1],
13     resnet[end][1:end-1],
14     # Replace the last layer
15     Dense(512 => 256, relu),
16     Dense(256 => 10)
17 )
18
19 using MLDatasets: CIFAR10
20 md"Load the CIFAR10 dataset"
21 function get_data(split, lm::Integer=1024)
22     data = CIFAR10(split)
23     X, y = data.features[:, :, :, 1:lm] ./ 255, onehotbatch(data.
    ↪targets[1:lm], 0:9)
24     loader = Flux.DataLoader((X, y); batchsize=16, shuffle=true)
25     return loader
26 end
27
28 train_loader = get_data(:train, 512);
29 test_loader = get_data(:test, 128);
30
31 md"Define a setup of the optimizer"
32 loss(X, y) = Flux.Losses.logitcrossentropy(mdl(X), y)
33 opt = Adam(3e-3)
34 ps = Flux.params(mdl[3:end])
35
36 for epoch in 1:5
37     Flux.train!(loss, ps, train_loader, opt, cb=Flux.throttle(() ->␣
    ↪println("Training"), 10))
38 end
39
40 for epoch in 1:100
41   Flux.train!(model, train_set, opt_state) do m, x, y
42     loss(m(x), y)
```

```
  end
end

using ImageShow, ImageInTerminal
idx = rand(1:50000)
convert2image(d, idx)
printstyled("Label is $(d.targets[idx])"; bold=true, color=:red)

#=
using Optimisers
opt_state = Optimisers.setup(Adam(3e-3), mdl[3:end]) # Freeze the weights of the pre-trained layers
using ProgressMeter
epochs = 5
# Fine-tune the model
for epoch in 1:epochs
    @showprogress for (X, y) in train_loader
        # Compute the gradient of the loss with respect to the model's parameters
         = Flux.gradient( m -> loss(m, X, y), mdl)
        # Update the `mdl`'s parameters
        Flux.update!(opt_state, mdl, [1])
    end
    @info "Calculate the accuracy on the test set"
    for (X, y) in test_loader
        accuracy = sum(onecold(mdll(X)) .== onecold(y)) / length(y)
        println("Epoch: $epoch, Accuracy: $accuracy")
    end
end
=#
```

Task № 3:

> In this lab, you will use `KNIME Analytics Platform` to apply **transfer learning** with a pre-trained CNN *(e.g., VGG16, ResNet, MobileNet)* for image classification. You will preprocess images to the required input size, load a pre-trained backbone, freeze base layers, add a custom classification head, train the head, then fine-tune selected layers with a lower learning rate. You will evaluate with accuracy and a confusion matrix, and compare results to training a CNN from scratch, using augmentation to improve generalization.
>
> **a)** Select and load a dataset for transfer learning

- Choose a suitable image classification dataset such as CIFAR-10, Cats vs Dogs, or a flower classification dataset;
- Load the dataset and examine the number of classes and images per class;
- Visualize sample images from each class to understand the characteristics of your target dataset;
- Split the dataset into training (70%), validation (15%), and test (15%) sets;
- Note the differences between your target dataset and ImageNet (the dataset the pre-trained model was trained on).

b) Preprocess images for the pre-trained model

- Resize all images to the input size required by the pre-trained model (e.g., 224×224 for VGG16 or ResNet);
- Apply the specific preprocessing steps required by your chosen pre-trained model (e.g., mean subtraction, scaling);
- Normalize pixel values according to the pre-trained model's requirements;
- Ensure images have three color channels (RGB) even if your dataset contains grayscale images;
- Convert class labels to one-hot encoded format for multi-class classification.

c) Load a pre-trained model and freeze base layers

- Load a pre-trained model (VGG16, ResNet50, MobileNet, or similar) with weights trained on ImageNet;
- Remove the original classification head (top layers) designed for ImageNet's 1000 classes;
- Freeze the weights of the convolutional base layers to preserve the learned features;
- Understand that freezing prevents these layers from being updated during initial training;
- Verify which layers are frozen and which are trainable in your model configuration.

d) Add custom classification layers

- Add new fully connected layers on top of the frozen pre-trained base;
- Design the new head with one or two dense layers (e.g., 256 or 512 neurons) with ReLU activation;
- Add dropout layers (0.3-0.5 dropout rate) to prevent overfitting;
- Add a final output layer with neurons equal to your number of classes and softmax activation;
- Configure the model with categorical cross-entropy loss and an optimizer like Adam or SGD.

e) Train the model with frozen base layers

- Train only the newly added classification layers while keeping the base frozen;
- Use a moderate learning rate (e.g., 0.001) and train for 10-15 epochs;
- Monitor training and validation accuracy to ensure the model is learning;
- Plot loss and accuracy curves for both training and validation sets;
- Evaluate the model on the test set and record the initial accuracy;
- Compare this performance to training a CNN from scratch (if time permits).

f) Fine-tune the pre-trained model

- Unfreeze some or all of the top layers of the pre-trained base (e.g., the last convolutional block);
- Use a much lower learning rate (e.g., 0.0001 or 0.00001) to avoid destroying the pre-trained weights;
- Train the model again for additional epochs (10-20) with the unfrozen layers;
- Monitor validation accuracy carefully to detect overfitting;
- Compare the fine-tuned model's performance with the frozen base model;
- Create a confusion matrix to analyze which classes are most difficult to classify.

g) Experiment and compare different transfer learning strategies

- Try different pre-trained models (VGG16, ResNet, MobileNet) and compare their performance on your dataset;
- Experiment with unfreezing different numbers of layers (e.g., last 2 blocks vs last 5 blocks);
- Test different learning rates during fine-tuning to find the optimal value;
- Apply data augmentation (rotation, flipping, zooming, brightness adjustment) to increase training data diversity;
- Compare training time and accuracy between transfer learning and training from scratch;
- Document your findings in a table showing model architecture, frozen layers, training time, and test accuracy;
- Discuss why transfer learning is particularly effective for your chosen dataset and which pre-trained model worked best.

4 | Generative Adversarial Network

Student's name	....	....	....
Score /20	.....	.....	.....

Detailed Credits

Anticipation *(4 points)*	.....	.....	.....
Management *(2 points)*	.....	.....	.....
Testing *(7 points)*	.....	.....	.....
Data Logging *(3 points)*	.....	.....	.....
Interpretation *(4 points)*	.....	.....	.....

The notebook is available at `https://github.com/a-mhamdi/jlai/` → *Codes* → *Julia* → *Part-3* → *gan* → *gan.ipynb*

GAN stands for Generative Adversarial Network, a framework for unsupervised generation. A GAN consists of two neural networks trained adversarially: a *generator* that synthesizes data resembling the training set, and a *discriminator* that distinguishes real from generated samples. Trained together in a minimax game, the generator learns to produce realistic outputs while the discriminator learns to detect fakes. **GAN**s are widely used for image synthesis and beyond (e.g., text and music generation).

```julia
using Flux # v0.14.25
using Images: Gray
using ProgressMeter

## Generator: noise vector -> synthetic sample.
function generator(; latent_dim=16, img_shape=(28,28,1,1))
    return Chain(
        Dense(latent_dim, 128, relu),
        Dense(128, 256, relu),
```

```
        Dense(256, prod(img_shape), tanh),
        x -> reshape(x, img_shape)
    )
end

## Discriminator : sample -> score indicating the probability that the sample is real.
function discriminator(; img_shape=(28,28,1,1))
    return Chain(
        x -> reshape(x, :, size(x, 4)),
        Dense(prod(img_shape), 256, relu),
        Dense(256, 128, relu),
        Dense(128, 1)
    )
end

## Loss functions
bce_loss(y_true, y_pred) = Flux.logitbinarycrossentropy(y_pred, y_true)

## Training function
function train_gan(gen, disc, gen_opt, disc_opt; n_epochs=16, latent_dim=16)
    @showprogress for epoch in 1:n_epochs

        ## Train the discriminator `disc`
        noise = randn(Float32, latent_dim, 1)
        fake_imgs = gen(noise) # pass the noise through the generator to get a synthetic sample
        real_imgs = rand(Float32, size(fake_imgs)...)

        disc_loss = bce_loss(ones(Float32, 1, 1), disc(real_imgs)) +
                        bce_loss(zeros(Float32, 1, 1), disc(fake_imgs)) # compute the loss for the real and synthetic samples
        grads = gradient(() -> disc_loss, Flux.params(disc))
        Flux.update!(disc_opt, Flux.params(disc), grads) # update the discriminator weights

        ## Train the generator `gen`
        noise = randn(Float32, latent_dim, 1)
        gen_loss = bce_loss( ones(Float32, 1, 1),  .(disc(gen(noise))) ) # compute the loss for the synthetic samples
        grads = gradient(() -> gen_loss, Flux.params(gen))
        Flux.update!(gen_opt, Flux.params(gen), grads) # update the generator weights
```

```
47
48          println("Epoch $(epoch): Discriminator loss = $(disc_loss), Generator loss = $(gen_loss)")
49          sleep(.1)
50      end
51  end
52
53  ## Setup the GAN
54  gen = generator()
55  disc = discriminator()
56
57  gen_opt = Adam(0.001)
58  disc_opt = Adam(0.0002)
59
60  ## Train the GAN
61  train_gan(gen, disc, gen_opt, disc_opt)
62
63  ## Generate and plot some images
64  latent_dim = 16
65  noise = randn(Float32, latent_dim, 16)
66  generated_images = [ gen(noise[:, i]) for i in 1:16 ]
67
68  using Plots
69
70  plot_images = [ plot(Gray.(generated_images[i])[:,:,1,1]) for i in 1:16 ]
71  titles = reshape([string(i) for i in 1:16], 1, :);
72
73  plot(
74      plot_images...,
75      layout = (4, 4),
76      title = titles, titleloc=:right, titlefont=font(8),
77      size = (800, 800)
78  )
```

Task № 4:

> In this lab, you will use `KNIME Analytics Platform` to build and train a **Generative Adversarial Network (GAN)** on MNIST to generate synthetic digit images. You will implement generator and discriminator networks, configure adversarial training with binary cross-entropy losses and Adam, monitor training stability, and evaluate generated samples.

a) Load and prepare the MNIST dataset

- Load MNIST (handwritten digits 0–9, grayscale);
- Visualize sample images to understand data characteristics;
- Normalize pixel values to [-1, 1] for `tanh` outputs or [0, 1] for `sigmoid`;
- Note: GAN training is unsupervised; labels are not required;
- Prepare data for batched training.

b) Design the generator architecture

- Build a generator network that takes random noise vectors (latent vectors) as input;
- Use a latent dimension of 100, sampled from a uniform or normal distribution;
- Use fully connected or transposed convolution layers to upsample to 28×28 resolution;
- Use ReLU/LeakyReLU in hidden layers and `tanh` (for [-1,1]) or `sigmoid` (for [0,1]) in the output layer;
- Ensure the output shape matches MNIST ($28 \times 28 \times 1$).

c) Design the discriminator architecture

- Build a discriminator that takes 28×28 images and outputs a single probability;
- Use convolutional or fully connected layers to classify real vs. fake;
- Use LeakyReLU in hidden layers to mitigate vanishing gradients;
- Use `sigmoid` in the output layer to produce probabilities in [0,1];
- Consider dropout to avoid an overly strong discriminator early on.

d) Configure the adversarial training process

- Use binary cross-entropy losses for generator and discriminator;
- Configure separate optimizers (e.g., Adam with learning rate 0.0002);
- Implement two-phase updates: train the discriminator (real vs. fake), then train the generator with the discriminator fixed;
- Use batch sizes of 128 or 256 for stability;
- Train for 50–100 epochs while monitoring dynamics.

e) Train the GAN and monitor training stability

- Train the discriminator by feeding it both real MNIST images (labeled as 1) and generated fake images (labeled as 0);
- Train the generator by trying to fool the discriminator into classifying fake images as real;

- Track both generator and discriminator losses throughout training;
- Plot loss curves over epochs to monitor stability;
- Check for mode collapse (generator produces limited variety) or divergence (losses oscillating wildly);
- Save generated image samples at regular intervals (e.g., every 5 epochs) to visually track improvement.

f) Evaluate the quality of generated images

- Generate a large batch of synthetic images using random noise vectors;
- Visualize a grid of generated images and assess their visual quality and diversity;
- Check whether the generated digits are recognizable and cover all digit classes (0-9);
- Evaluate if the generator suffers from mode collapse by examining the variety of generated samples;
- Compare early-epoch generations with late-epoch generations to observe quality improvement over training.

5 | Variational Autoencoder

Student's name	. . . . .	. . . . .	. . . . .
	. . . . .	. . . . .	. . . . .
	. . . . .	. . . . .	. . . . .
Score /20	. . . . .	. . . . .	. . . . .

Detailed Credits

Anticipation *(4 points)*	. . . . .	. . . . .	. . . . .
Management *(2 points)*	. . . . .	. . . . .	. . . . .
Testing *(7 points)*	. . . . .	. . . . .	. . . . .
Data Logging *(3 points)*	. . . . .	. . . . .	. . . . .
Interpretation *(4 points)*	. . . . .	. . . . .	. . . . .

The notebook is available at `https://github.com/a-mhamdi/jlai/` ⟶ *Codes* ⟶ *Julia* ⟶ *Part-3* ⟶ *vae* ⟶ *vae.ipynb*

VAE stands for Variational Autoencoder, a probabilistic deep learning model for unsupervised representation learning. A **VAE** learns the underlying structure of a dataset by representing observations with *latent variables*, i.e., lower-dimensional embeddings of the data. A **VAE** comprises two parts:

an *encoder* that maps inputs to a latent distribution, and

a *decoder* that reconstructs data from samples drawn from this distribution.

The key idea behind a **VAE** is to learn compact latent representations that allow high-quality reconstructions. Training minimizes a *reconstruction loss* (difference between input and output) plus a *regularization term* (KL divergence) to encourage a smooth, well-behaved latent space. **VAE**s are used for image generation, anomaly detection, and representation learning.

```
using Markdown
md"VAE implemented in `Julia` using the `Flux.jl` library"

```

```
md"Import the machine learning library `Flux`"
using Flux # v0.14.25
using Flux: @functor
using Flux: DataLoader
using Flux: onecold, onehotbatch

using ProgressMeter: Progress, next!

using MLDatasets
d = MNIST()

Base.@kwdef mutable struct HyperParams
      = 3f-3                        # Learning rate
      = 1f-2                        # Regularization parameter
    batchsize = 64                  # Batch size
    epochs = 16                     # Number of epochs
    split = :train                  # Split data into `train` and `test`
    input_dim = 28*28               # Input dimension
    hidden_dim = 512                # Hidden dimension
    latent_dim = 2                  # Latent dimension
    # save_path = "Output"          # Results folder
end

md"Load the **MNIST** dataset"
function get_data(; kws...)
    args = HyperParams(; kws...);
    md"Split data"
    data = MNIST(split=args.split);
    X = reshape(data.features, (args.input_dim, :));
    loader = DataLoader(X; batchsize=args.batchsize, shuffle=true);
    return loader
end

train_loader = get_data();
test_loader = get_data(split=:test);

md"Define the `encoder` network"
# The encoder network should return the parameters of the _latent␣
 ↪distribution_ ( and ).
struct Encoder
    linear

    log_
end
```

```
47
48  @functor Encoder
49
50  encoder(input_dim::Int, hidden_dim::Int, latent_dim::Int) = Encoder(
51      Dense(input_dim, hidden_dim, tanh),   # linear
52      Dense(hidden_dim, latent_dim),        #
53      Dense(hidden_dim, latent_dim),        # log_
54  )
55
56  function (encoder::Encoder)(x)
57      h = encoder.linear(x)
58      encoder. (h), encoder.log_ (h)
59  end
60
61  md"Define the `decoder` network"
62  # The decoder network should return the reconstruction of the input data
63  decoder(input_dim::Int, hidden_dim::Int, latent_dim::Int) = Chain(
64      Dense(latent_dim, hidden_dim, tanh),
65      Dense(hidden_dim, input_dim)
66  )
67
68  md"Reconstruction of the input data"
69  function vae(x, enc, dec)
70      # Encode `x` into the latent space
71       , log_  = enc(x)
72      # `z` si a sample from the latent distribution
73      z =   + randn(Float32, size(log_ )) .* exp.(log_ )
74      # Decode the latent representation into a reconstruction of `x`
75      x̂ = dec(z)
76      # Return  , log_  and x̂
77       , log_ , x̂
78  end
79
80  function l(x, enc, dec,  )
81       , log_ , x̂ = vae(x, enc, dec)
82      len = size(x)[end]
83      # The reconstruction loss measures how well the VAE was able to␣
    ↪reconstruct the input data
84      logp_x_z = -Flux.Losses.logitbinarycrossentropy(x̂, x, agg=sum) / len
85      # The KL divergence loss measures how close the latent distribution␣
    ↪is to the normal distribution
86      kl_q_p = 5f-1 * sum(@. (-2f0 * log_  - 1f0 + exp(2f0 * log_ ) +  ^2))␣
    ↪/ len
87      # L2 Regularization
```

```
88      reg =   * sum(   -> sum( .^2), Flux.params(dec) )
89      # Sum of the reconstruction loss and the KL divergence loss
90      -logp_x_z + kl_q_p + reg
91  end
92
93  function train(; kws...)
94      args = HyperParams(; kws...)
95
96      # Initialize `encoder` and `decoder`
97      enc_mdl = encoder(args.input_dim, args.hidden_dim, args.latent_dim)
98      dec_mdl = decoder(args.input_dim, args.hidden_dim, args.latent_dim)
99
100     # ADAM optimizers
101     opt_enc = Flux.setup(Adam(args. ), enc_mdl)
102     opt_dec = Flux.setup(Adam(args. ), dec_mdl)
103
104     for epoch in 1:args.epochs
105         printstyled("\t***\t === EPOCH $(epoch) === \t*** \n", color=:
    ↪magenta, bold=true)
106         progress = Progress(length(train_loader))
107         for X in train_loader
108                 loss, back = Flux.pullback(enc_mdl, dec_mdl) do enc, dec
109                     l(X, enc, dec, args. )
110                 end
111                 grad_enc, grad_dec = back(1f0)
112                 Flux.update!(opt_enc, enc_mdl, grad_enc) # Upd␣
    ↪`encoder` params
113                 Flux.update!(opt_dec, dec_mdl, grad_dec) # Upd␣
    ↪`decoder` params
114                 next!(progress; showvalues=[(:loss, loss)])
115         end
116     end
117
118     md"Save the model"
119     #=
120     using DrWatson: struct2dict
121     using BSON
122
123     mdl_path = joinpath(args.save_path, "vae.bson")
124     let args=struct2dict(args)
125             BSON.@save mdl_path encoder decoder args
126             @info "Model saved to $(mdl_path)"
127     end
128     =#
```

```
129
130     enc_mdl, dec_mdl
131 end
132
133 enc_model, dec_model = train()
```

Task № 5:

In this lab, you will use `KNIME Analytics Platform` to build, train, and evaluate a **Variational Autoencoder (VAE)** on MNIST (28×28 grayscale digits). You will prepare the data, design encoder/decoder networks, implement the VAE loss (reconstruction + KL divergence), train the model, assess reconstruction quality, visualize the latent space, and generate new samples by decoding latent vectors. You will then explore variations (latent size, β-VAE, architecture depth) to improve results.

a) Load and prepare the MNIST dataset

- Load `MNIST` ($70{,}000$ grayscale images of digits $0--9$);
- Visualize a sample to understand data characteristics;
- Normalize pixel values to the range $[0, 1]$;
- Split the dataset into training (80%) and test (20%) sets.

Note that for **VAE** training, you do not need class labels as this is an unsupervised learning task.

b) Design the **VAE** encoder architecture

- Build an encoder for 28×28 input images that progressively reduces dimensionality;
- Use convolutional layers or fully connected layers to extract features;
- Output two vectors: mean (μ) and log-variance ($\log \sigma^2$) of the latent distribution;
- Choose latent dimensionality: $2--3$ for visualization, or $10--20$ for better reconstructions;
- Implement the reparameterization trick: $z = \mu + \sigma \odot \epsilon$, with $\epsilon \sim \mathcal{N}(0, I)$.

c) Design the **VAE** decoder architecture

- Build a decoder that takes latent vector z and reconstructs the image;
- Use transposed convolutions or fully connected layers to increase dimensionality;
- Ensure the output shape matches the input (28×28);
- Use `sigmoid` activation in the final layer to keep outputs in $[0, 1]$.

d) Configure the **VAE** loss function and train the model

- Use a loss combining reconstruction (`binary cross-entropy` or `MSE`) and KL divergence;
- KL divergence regularizes the latent space toward $\mathcal{N}(0, I)$;
- Train for $30 - -50$ epochs with a suitable batch size (e.g., 128);
- Monitor reconstruction, KL, and total loss during training;
- Plot the total loss over epochs to verify convergence.

e) Evaluate reconstruction quality and visualize the latent space

- Reconstruct test images by passing them through encoder and decoder;
- Visualize originals alongside reconstructions to assess quality;
- Compute reconstruction error (e.g., mean squared error) on the test set;
- For 2D latent spaces, plot encoded test points colored by digit label;
- Check whether similar digits cluster in latent space.

f) Generate new images from the latent space

- Sample latent vectors from a standard normal distribution;
- Decode these vectors to generate new synthetic digit images;
- Visualize a grid of generated images to assess quality and diversity;
- For 2D latent spaces, generate a grid to observe smooth transitions;
- Interpolate between two encodings and visualize intermediate generations.

g) Experiment with **VAE** variations

- Vary latent dimensionality (e.g., $2, 5, 10, 20$) to compare quality vs. interpretability;
- Adjust the KL weight (β-**VAE**) to balance reconstruction and regularization;
- Test different encoder/decoder depths and filter sizes;
- Record how each change affects reconstruction quality and generation diversity.

6 | Reinforcement Learning

Student's name	. . . . .	. . . . .	. . . . .
	. . . . .	. . . . .	. . . . .
	. . . . .	. . . . .	. . . . .
Score /20	. . . . .	. . . . .	. . . . .

Detailed Credits

Anticipation *(4 points)*	. . . . .	. . . . .	. . . . .
Management *(2 points)*	. . . . .	. . . . .	. . . . .
Testing *(7 points)*	. . . . .	. . . . .	. . . . .
Data Logging *(3 points)*	. . . . .	. . . . .	. . . . .
Interpretation *(4 points)*	. . . . .	. . . . .	. . . . .

The notebook is available at `https://github.com/a-mhamdi/jlai/` ⟶ *Codes* ⟶ *Julia* ⟶ *Part-3* ⟶ *reinforcement-learning* ⟶ *reinforcement-learning.ipynb*

Studying reinforcement learning helps us understand how agents learn to act in environments by maximizing reward, enabling a wide range of practical applications.

There are several reasons why it is important to investigate reinforcement learning:

- It has proven effective for many tasks, such as control systems and game playing.
- It allows machines to learn from their own actions and experiences, rather than relying on pre-programmed rules or human input. This can lead to more flexible and adaptable systems.
- It has the potential to be used in a variety of real-world applications, including robotics, self-driving cars, and financial trading.

```julia
using ReinforcementLearning
using Flux: Descent

## Define the environment
```

```
env = RandomWalk1D()

## Instantiate the agent
agent = Agent(
    policy = QBasedPolicy(
        learner = TDLearner(
            approximator = TabularQApproximator(
                n_state = 11,
                n_action = 2,
                init = 0.0,
                opt = Descent(0.1) # Learning rate
            ),
            method = :SARSA,
              = 0.99
        ),
        explorer = EpsilonGreedyExplorer(0.1),
    ),
    trajectory = VectorSARTTrajectory(),
)

## Run the experiment
hook = TotalRewardPerEpisode()
run(agent, env, StopAfterEpisode(10_000), hook)

## Print rewards
println("Total reward per episode:")
println(hook.rewards)

## Print `Q-table``
q_table = agent.policy.learner.approximator.table
println("\nLearned Q-table:")
println(q_table)
```

julia

Task № 6:

> In this lab, you will use `KNIME Analytics Platform` to implement a basic reinforcement learning agent (**Q-learning** or **DQN**) for a classic control task (e.g., CartPole). You will configure the environment, implement exploration–exploitation (epsilon-greedy), train over multiple episodes while monitoring rewards, evaluate the learned policy, and experiment with hyperparameters and improvements (e.g., experience replay for DQN).

a) Set up the reinforcement learning environment

- Choose a simple RL environment such as CartPole, Mountain Car, or a grid-world navigation problem;
- Understand the environment's state space (what information the agent observes);
- Identify the action space (what actions the agent can take);
- Define the reward structure (how the agent receives feedback for its actions);
- Determine the episode termination conditions (success criteria or failure states);
- Set a maximum number of steps per episode to prevent infinite loops.

b) Implement the Q-learning algorithm basics

- Initialize a Q-table or Q-network to store state–action values;
- For discrete state spaces, create a table with rows for states and columns for actions;
- For continuous state spaces, use a neural network to approximate the Q-function;
- Set hyperparameters: learning rate ($\alpha = 0.1$), discount factor ($\gamma = 0.99$), exploration rate ($\epsilon = 1.0$);
- Recall: a Q-value estimates expected cumulative reward for taking an action in a given state.

c) Implement the exploration-exploitation strategy

- Implement an epsilon-greedy policy for action selection;
- With probability ϵ, select a random action (exploration);
- With probability $1 - \epsilon$, select the action with the highest Q-value (exploitation);
- Use an epsilon decay schedule to reduce exploration over time (e.g., $\epsilon \leftarrow \epsilon \times 0.995$ after each episode);
- Set a minimum ϵ (e.g., 0.01) to maintain some exploration.

d) Train the reinforcement learning agent

- Run multiple training episodes (e.g., 500-1000 episodes depending on environment complexity);
- For each episode: reset the environment, select actions using epsilon-greedy policy, observe rewards and next states;
- Update Q-values using the Q-learning rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right]$$

- For neural network approaches, collect experiences and perform batch updates;

- Track the total reward obtained in each episode;
- Monitor the average reward over the last 100 episodes to assess learning progress.

e) Evaluate agent performance and learning progress

- Plot the episode rewards over training to visualize the learning curve;
- Calculate and plot a moving average of rewards to smooth noise;
- Observe whether the agent's performance improves and stabilizes over time;
- Test the trained agent with epsilon set to 0 (pure exploitation) for several episodes;
- Record the success rate and average reward during evaluation;
- Compare early episodes (random behavior) with late episodes (learned policy) to see improvement.

f) Analyze the learned policy

- Visualize the learned Q-values for different states (if state space is small enough);
- For grid-world environments, create a heatmap showing the value of each state;
- Display the optimal action for each state according to the learned policy;
- Run the agent multiple times to check consistency of learned strategies;
- Identify any states where the agent struggles or makes suboptimal decisions;
- Discuss whether the agent has learned a robust policy or if it relies on specific patterns.

g) Experiment with hyperparameters and improvements

- Test different learning rates (0.01, 0.1, 0.5) and observe effects on convergence;
- Vary discount factors (0.9, 0.95, 0.99) to change the planning horizon;
- Try different epsilon decay schedules (linear, exponential, step);
- For neural networks, vary architectures and batch sizes;
- Implement experience replay to decorrelate consecutive experiences;
- Compare stability and final performance across configurations;
- Document which settings yield the fastest learning and best final performance;
- Discuss exploration–exploitation trade-offs in your experiments.

7 Project Assessment

The final project will offer you the possibility to cover in depth a topic discussed in class which interests you, and you like to know more about it. The overall goal is to provide you with a challenging but achievable assessment that allows you to demonstrate your knowledge and skills in deep learning.

You have to provide all necessary resources, such as sample code, relevant datasets, as well as creating a set of slides to present your work. You are expected to demonstrate your understanding of the material covered throughout this course, as well as familiarizing yourselves with relevant programming languages and libraries. The final project is comprised of:

1. proposal;
2. report documenting your work, results and conclusions;
3. presentation;
4. source code *(You should share your project on* **GitHub***.)*

It is about two pages long. It includes:

- Title
- Datasets *(If needed!)*
- Idea
- Software *(Not limited to what you have seen in class)*
- Related papers *(Include at least one relevant paper)*
- Teammate *(Teams of three to four students. You should highlight each partner's contribution)*

It is about ten pages long. It revolves around the following key takeaways:

- Context *(Input(s) and output(s))*
- Motivation *(Why?)*
- Previous work *(Literature review)*
- Flowchart of code, results and analysis
- Contribution parts *(Who did what?)*

Typesetting using LaTeX is a bonus. You can use **LyX** (https://www.lyx.org/) editor. A template is available at https://github.com/a-mhamdi/ailab-isetbz/tree/main/LyX. Here what your report might contain:

1. Provide a summary which gives a brief overview of the main points and conclusions of the report.
2. Use headings and subheadings to organize the main points and the relationships between the different sections.
3. Provide an outline or a list of topics that the report will cover. Including a table of contents can help to quickly and easily find specific sections of your report.
4. Use visuals: Including visual elements such as graphs, charts, and tables can help to communicate the content of a report more effectively. Visuals can help to convey complex information in a more accessible and intuitive way.

If you are using `Julia`, you can generate the documentation using the package **Documenter.jl**. It is a great way to create professional-looking material. It allows to easily write and organize documentation using a variety of markup languages, including **Markdown** and LaTeX, and provides a number of features to help create a polished and user-friendly documentation website.

I will assess your work based on the quality of your code and slides, as well as your ability to effectively explain and demonstrate your understanding of the topic. I will also consider the creativity and originality of your projects, and your ability to apply what you have learned to real-world situations. I also make myself available to answer any questions or provide feedback as you work on your projects.

The overall scope of this manual is to introduce **Artificial Intelligence (AI)** , through either some numerical simulations or hands-on training, to the students at **ISET Bizerte**.

The topics discussed in this manuscript are as follow:

① NLP (Natural Language Processing)

language translation; text classification; language generation.

② CNN (Convolutional Neural Network)

image classification; computer vision; feature learning.

③ Transfer Learning

pre-trained models; fine-tuning; domain adaptation.

④ GAN (Generative Adversarial Network)

data generation; image generation; adversarial training.

⑤ VAE (Variational Autoencoder)

image generation; anomaly detection; latent representation.

⑥ Reinforcement Learning

control systems; game playing; decision making.

Julia; *REPL*; *Pluto*; *Flux*; *MLJ*; cnn; gan; vae; nlp; transfer learning; reinforcement learning