Term: M2-EE
Semester: 3
AY: 2024-2025

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[1];

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*

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

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

Typst `https://typst.app/project/rbpG25Q18MB7pPvYwgOfbQ`

[1]If you prefare using Windows, a similar environment has been setup for you by pressing ⊞ & R . This will open the dialog box **Run**. In the command line, type **`cmd`**, and then use the ↵ key to confirm. Next, type `jlai3` and press ↵ once more.

1 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, which is an advanced type of artificial neural network used for image and video recognition. It is composed of multiple layers of interconnected nodes, with each layer performing a specific function in the processing of the input data. The layers at the beginning of the network, known as *the input layers*, process the raw data, while the layers at the end of the network, known as *the output layers*, produce the final output. In between the input and output layers are *hidden layers*, which perform intermediate processing on the data. Convolutional Neural Networks are particularly useful for image and video recognition tasks because they are able to learn features and patterns in the data directly from the raw input, rather than requiring them to be hand-engineered.

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

using Statistics
```

```julia
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
    η = 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)`
```

```julia
"""

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(α, β) = Flux.logitcrossentropy(α, β)
    md"Define the accuracy metric"
    acc(α, β) = mean(onecold(α) .== onecold(β))
    md"Optimizer"
    optim_state = Flux.setup(Adam(args.η), 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
```

2 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, which is a type of deep learning model used for unsupervised learning. It is a probabilistic model that is designed to learn the underlying structure of a dataset by representing the data as a set of *latent variables*, which are reduced-dimensional representations of the data. A **VAE** consists of two components: an *encoder* network that maps the input data to the latent space, and a *decoder* network that maps the latent representation back to the original data space.

The key idea behind a **VAE** is to learn a compact representation of the data in the latent space, such that the data can be reconstructed from the latent representation with minimal loss of information. This is achieved by minimizing a reconstruction loss, which measures the difference between the original data and the reconstructed data, and a regularization term, which encourages the latent representation to be smooth and continuous. **VAE**s have been used for a variety of tasks, including image generation, anomaly detection, and representation learning.

```julia
using Markdown
```

```julia
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
    η = 3f-3                        # Learning rate
    λ = 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_ (μ and σ).
struct Encoder
    linear
    μ
```

```
    log_σ
end

@functor Encoder

encoder(input_dim::Int, hidden_dim::Int, latent_dim::Int) = Encoder(
    Dense(input_dim, hidden_dim, tanh),   # linear
    Dense(hidden_dim, latent_dim),        # μ
    Dense(hidden_dim, latent_dim),        # log_σ
)

function (encoder::Encoder)(x)
    h = encoder.linear(x)
    encoder.μ(h), encoder.log_σ(h)
end

md"Define the `decoder` network"
# The decoder network should return the reconstruction of the input data
decoder(input_dim::Int, hidden_dim::Int, latent_dim::Int) = Chain(
    Dense(latent_dim, hidden_dim, tanh),
    Dense(hidden_dim, input_dim)
)

md"Reconstruction of the input data"
function vae(x, enc, dec)
    # Encode `x` into the latent space
    μ, log_σ = enc(x)
    # `z` si a sample from the latent distribution
    z = μ + randn(Float32, size(log_σ)) .* exp.(log_σ)
    # Decode the latent representation into a reconstruction of `x`
    x  = dec(z)
    # Return μ, log_σ and x
    μ, log_σ, x
end

function l(x, enc, dec, λ)
    μ, log_σ, x  = vae(x, enc, dec)
    len = size(x)[end]
    # The reconstruction loss measures how well the VAE was able to reconstruct the input data
    logp_x_z = -Flux.Losses.logitbinarycrossentropy(x , x, agg=sum) / len
    # The KL divergence loss measures how close the latent distribution is to the normal distribution
    kl_q_p = 5f-1 * sum(@. (-2f0 * log_σ - 1f0 + exp(2f0 * log_σ) + μ^2)) / len
```

```
    # L2 Regularization
    reg = λ * sum( θ -> sum(θ.^2), Flux.params(dec) )
    # Sum of the reconstruction loss and the KL divergence loss
    -logp_x_z + kl_q_p + reg
end

function train(; kws...)
    args = HyperParams(; kws...)

    # Initialize `encoder` and `decoder`
    enc_mdl = encoder(args.input_dim, args.hidden_dim, args.latent_dim)
    dec_mdl = decoder(args.input_dim, args.hidden_dim, args.latent_dim)

    # ADAM optimizers
    opt_enc = Flux.setup(Adam(args.η), enc_mdl)
    opt_dec = Flux.setup(Adam(args.η), dec_mdl)

    for epoch in 1:args.epochs
        printstyled("\t***\t === EPOCH $(epoch) === \t*** \n", color=:magenta, bold=true)
        progress = Progress(length(train_loader))
        for X in train_loader
                loss, back = Flux.pullback(enc_mdl, dec_mdl) do enc, dec
                    l(X, enc, dec, args.λ)
                end
                grad_enc, grad_dec = back(1f0)
                Flux.update!(opt_enc, enc_mdl, grad_enc) # Upd `encoder` params
                Flux.update!(opt_dec, dec_mdl, grad_dec) # Upd `decoder` params
                next!(progress; showvalues=[(:loss, loss)])
        end
    end

    md"Save the model"
    #=
    using DrWatson: struct2dict
    using BSON

    mdl_path = joinpath(args.save_path, "vae.bson")
    let args=struct2dict(args)
            BSON.@save mdl_path encoder decoder args
            @info "Model saved to $(mdl_path)"
    end
    =#

```

```
    enc_mdl, dec_mdl
end

enc_model, dec_model = train()
```

3 | 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, which is a type of artificial neural network used for unsupervised learning. It consists of two networks, a *generator* and a *discriminator*, which are trained to work against each other in a zero-sum game. The *generator* network tries to produce synthetic data that is similar to some training data, while the *discriminator* network tries to distinguish between the synthetic data produced by the *generator* and the real training data. The two networks are trained together, with the *generator* trying to produce data that can fool the *discriminator*, and the *discriminator* trying to correctly identify whether each piece of data is real or synthetic. The end result is a *generator* network that is able to produce synthetic data that is similar to the training data. **GAN**s have been used for a variety of tasks, including image generation, text generation, and even 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), σ.(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

        println("Epoch $(epoch): Discriminator loss = $(disc_loss), Generator loss = $(gen_loss)")
        sleep(.1)
    end
end

## Setup the GAN
gen = generator()
disc = discriminator()

gen_opt = Adam(0.001)
disc_opt = Adam(0.0002)

## Train the GAN
train_gan(gen, disc, gen_opt, disc_opt)

## Generate and plot some images
latent_dim = 16
noise = randn(Float32, latent_dim, 16)
generated_images = [ gen(noise[:, i]) for i in 1:16 ]

using Plots

plot_images = [ plot(Gray.(generated_images[i])[:,:,1,1]) for i in 1:16 ]
titles = reshape([string(i) for i in 1:16], 1, :);

plot(
    plot_images...,
    layout = (4, 4),
    title = titles, titleloc=:right, titlefont=font(8),
    size = (800, 800)
)
```

4 | 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*

Here are some points that outline the general process of performing Natural Language Processing (**NLP**) tasks in `Julia`:

- Load and preprocess the text data. This may involve cleaning the text *(e.g., removing punctuation, lowercasing)*, tokenizing the text *(splitting it into individual words or phrases)*, and encoding the text *(e.g., using word embeddings)*.
- Choose an **NLP** model and define any necessary hyperparameters, such as hidden Markov models, conditional random fields, and transformer-based models.
- Train the **NLP** model on the preprocessed text data. This may involve using an optimization algorithm *(e.g., stochastic gradient descent)* to adjust the model's parameters to minimize a loss function.
- Evaluate the performance of the model on a separate test set.
- Use the trained model to make predictions on new, unseen text data.

```julia
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
α = 0.1              # hyper parameter
β  = 0.1             # hyper parameter
 , θ  = lda(m, k, iterations, α, β) #
```

5 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 re-purposed on a second related task. It involves taking a *pre-trained model*, which has already learned to perform a certain task, and adapting it to perform a new task. Transfer learning can be an useful approach when there is not enough data available to train a model from scratch, or when the new task is very similar to the original task the model was trained on.

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.

```julia
using Markdown

using Metalhead
md"Load the pre-trained model"
md"[API Reference](https://fluxml.ai/Metalhead.jl/dev/api/reference/#API-Reference)"
resnet = ResNet(18; pretrain=true).layers;

using Flux
using Flux: onecold, onehotbatch

mdl = Chain(
    resnet[1:end-1],
    resnet[end][1:end-1],
    # Replace the last layer
    Dense(512 => 256, relu),
    Dense(256 => 10)
)

using MLDatasets: CIFAR10
md"Load the CIFAR10 dataset"
function get_data(split, lm::Integer=1024)
    data = CIFAR10(split)
    X, y = data.features[:, :, :, 1:lm] ./ 255, onehotbatch(data.targets[1:lm], 0:9)
    loader = Flux.DataLoader((X, y); batchsize=16, shuffle=true)
    return loader
end

train_loader = get_data(:train, 512);
test_loader = get_data(:test, 128);

md"Define a setup of the optimizer"
loss(X, y) = Flux.Losses.logitcrossentropy(mdl(X), y)
opt = Adam(3e-3)
ps = Flux.params(mdl[3:end])

for epoch in 1:5
    Flux.train!(loss, ps, train_loader, opt, cb=Flux.throttle(() -> println("Training"), 10))
end

for epoch in 1:100
  Flux.train!(model, train_set, opt_state) do m, x, y
    loss(m(x), y)
```

```julia
    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
=#
```

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 can help us better understand how machines can learn to interact with their environments and make decisions, which has the potential to lead to a wide range of practical applications.

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

- It has proven to be effective for a wide range of 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
```

```julia

## 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,
            γ = 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)
```

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.

Some possible topics that can be covered include, but not limited to:

Artificial Neural Networks: These are the foundation of deep learning, and are used to build models that can process and analyze large amounts of data.

Convolutional Neural Networks: These are a type of neural network that are particularly well-suited for image and video processing tasks.

Recurrent Neural Networks: These are a type of neural network that are designed to process sequential data, such as time series or natural language.

Autoencoders: These are a type of neural network that can learn to compress and reconstruct data, and are often used for dimensionality reduction and anomaly detection tasks.

Generative Adversarial Networks: These are a type of neural network that are used to generate new, synthetic data that is similar to a given input dataset.

Transfer Learning: This is the process of using a pre-trained neural network as a starting point for a new task, and fine-tuning the network on the new task using a smaller dataset.

Hyperparameter Optimization: This involves finding the best set of hyperparameters (such as learning rate and regularization strength) for a neural network in order to improve its performance on a given task.

Evaluation and Comparison of Deep Learning Models: This involves using various techniques and metrics (such as accuracy, precision, and recall) to evaluate the performance of deep learning models, and comparing the results of different models to choose the best one for a given task.

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**.)*

Project proposal

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)*

Project report

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/jlai/tree/main/Codes/Report. 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 CNN (Convolutional Neural Network)

image classification; computer vision; feature learning.

② VAE (Variational Autoencoder)

image generation; anomaly detection; latent representation.

③ GAN (Generative Adversarial Network)

data generation; image generation; adversarial training.

④ NLP (Natural Language Processing)

language translation; text classification; language generation.

⑤ Transfer Learning

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

⑥ Reinforcement Learning

control systems; game playing; decision making.

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