

**TERM: M2-EE**

**SEMESTER: 3**

**AY: 2024-2025**

# Abdelbacet Mhamdi

Dr.-Ing. in Electrical Engineering

Senior Lecturer at ISET Bizerte

abdelbacet.mhamdi@bizerte.riset.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

<b>1</b>	<b>Convolutional Neural Network</b>	<b>1</b>
<b>2</b>	<b>Variational Autoencoder</b>	<b>5</b>
<b>3</b>	<b>Generative Adversarial Network</b>	<b>10</b>
<b>4</b>	<b>Natural Language Processing</b>	<b>13</b>
<b>5</b>	<b>Transfer Learning</b>	<b>17</b>
<b>6</b>	<b>Reinforcement Learning</b>	<b>20</b>
<b>7</b>	<b>Project Assessment</b>	<b>22</b>


---

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

- ① Press simultaneously the keys   and  on the keyboard<sup>1</sup>;
- ② Type `j1ai3` in the console prompt line;

 Terminal

```
student@isetbz:~$ j1ai3
```

- 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:

**TeX** <https://www.overleaf.com/read/pwgpyvcxcvym#9e34eb>

**Typst** <https://typst.app/universe/package/ailab-isetbz>

---

<sup>1</sup>If you prefer using Windows, a similar environment has been setup for you by pressing  & . This will open the dialog box Run. In the command line, type `cmd`, and then use the  key to confirm. Next, type `j1ai3` and press  once more.

# 1 | Convolutional Neural Network

<b>Student's name</b>	.....	.....	.....
	.....	.....	.....
	.....	.....	.....
<b>Score</b> /20	.....	.....	.....

## Detailed Credits

<b>Anticipation (4 points)</b>	.....	.....	.....
<b>Management (2 points)</b>	.....	.....	.....
<b>Testing (7 points)</b>	.....	.....	.....
<b>Data Logging (3 points)</b>	.....	.....	.....
<b>Interpretation (4 points)</b>	.....	.....	.....



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.

```

1 using Markdown
2 md"Handwritten digits classification using **CNN**. This solution is implemented in
  'Julia' using the 'Flux.jl' library"
3
4 using Statistics

```

```

5 using ProgressMeter: Progress, next!
6 using Plots
7
8 md"Import the machine learning library `Flux`"
9 using Flux # v0.14.25
10 using Flux: DataLoader
11 using Flux: onecold, onehotbatch
12
13 using MLDatasets
14 d = MNIST()
15
16 Base.@kwdef mutable struct HyperParams
17     η = 3f-3           # Learning rate
18     batchsize = 64     # Batch size
19     epochs = 8         # Number of epochs
20     split = :train     # Split data into `train` and `test`
21 end
22
23 md"Load the **MNIST** dataset"
24 function get_data(; kws...)
25     args = HyperParams(; kws...);
26     md"Split and normalize data"
27     data = MNIST(split=args.split);
28     X, y = data.features ./ 255, data.targets;
29     X = reshape(X, (28, 28, 1, :));
30     y = onehotbatch(y, 0:9);
31     loader = DataLoader((X, y); batchsize=args.batchsize, shuffle=true);
32     return loader
33 end
34
35 train_loader = get_data();
36 test_loader = get_data(split=:test);
37
38 md"Transform sample training data to an image. View the image and check the
corresponding digit value."
39 idx = rand(1:6_000);
40 using ImageShow, ImageInTerminal # ImageView
41 convert2image(d, idx) # |> imshow
42 md"**Digit is $(d.targets[idx])**"
43
44 md""
45 ## **CNN** ARCHITECTURE"
46 The input `X` is a batch of images with dimensions `(width=28, height=28, channels=1,
batchsize)`

```

```

47  """
48
49  fc = prod(Int.(floor.([28/4 - 2, 28/4 - 2, 16]))) # 2^{# max-pool}
50  model = Chain(
51      Conv((5, 5), 1 => 16, relu), # (28-5+1)x(28-5+1)x16 = 24x24x16
52      MaxPool((2, 2)), # 12x12x16
53      Conv((3, 3), 16 => 16, relu), # (12-3+1)x(12-3+1)x16 = 10x10x16
54      MaxPool((2, 2)), # 5x5x16
55      Flux.flatten, # 400
56      Dense(fc => 64, relu),
57      Dense(64 => 32, relu),
58      Dense(32 => 10)
59  )
60
61  function train(; kws...)
62      args = HyperParams(; kws...)
63      md"Define the loss function"
64      l(α, β) = Flux.logitcrossentropy(α, β)
65      md"Define the accuracy metric"
66      acc(α, β) = mean(onecold(α) .== onecold(β))
67      md"Optimizer"
68      optim_state = Flux.setup(Adam(args.η), model);
69
70      vec_loss = []
71      vec_acc = []
72
73      for epoch in 1:args.epochs
74          printstyled("\t***\t === EPOCH $(epoch) === \t*** \n", color=:magenta,
bold=true)
75          @info "TRAINING"
76          prg_train = Progress(length(train_loader))
77          for (X, y) in train_loader
78              loss, grads = Flux.withgradient(model) do m
79                  ŷ = m(X);
80                  l(ŷ, y);
81              end
82              Flux.update!(optim_state, model, grads[1]); # Upd `W` and `b`
83              # Show progress meter
84              next!(prg_train, showvalues=[:loss, loss])
85          end
86          @info "TESTING"
87          prg_test = Progress(length(test_loader))
88          for (X, y) in test_loader
89              ŷ = model(X);

```



```

90         push!(vec_loss, l(y_hat, y)); # log `loss` value -> `vec_loss` vector
91         push!(vec_acc, acc(y_hat, y)); # log `accuracy` value -> `vec_acc` vector
92         # Show progress meter
93         next!(prg_test, showvalues=[:loss, vec_loss[end]], (:accuracy, vec_
←acc[end]))
94     end
95 end
96 return vec_loss, vec_acc
97 end
98
99 vec_loss, vec_acc = train()
100
101 # Plot results
102 plot(vec_loss, label="Test Loss")
103 plot(vec_acc, label="Test Accuracy")
104
105 # Let's make some predictions
106 idx = rand(1:1000, 16)
107 xs, ys = test_loader.data[1][:, :, :, idx], onecold(test_loader.data[2][:, idx]) .- 1
108 yp = onecold(model(xs)) .- 1
109
110 for i ∈ eachindex(yp)
111     @info "**Prediction is $(yp[i]). Label is $(ys[i]).**"
112 end
113
114 # Save the model
115 using BSON: @save
116 @save "cnn.bson" model

```



## 2 | Variational Autoencoder

<b>Student's name</b>	.....	.....	.....
	.....	.....	.....
	.....	.....	.....
<b>Score</b> /20	.....	.....	.....

### Detailed Credits

<b>Anticipation (4 points)</b>	.....	.....	.....
<b>Management (2 points)</b>	.....	.....	.....
<b>Testing (7 points)</b>	.....	.....	.....
<b>Data Logging (3 points)</b>	.....	.....	.....
<b>Interpretation (4 points)</b>	.....	.....	.....



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. **VAEs** have been used for a variety of tasks, including image generation, anomaly detection, and representation learning.

```

2  md"VAE implemented in `Julia` using the `Flux.jl` library"
3
4  md"Import the machine learning library `Flux`"
5  using Flux # v0.14.25
6  using Flux: @functor
7  using Flux: DataLoader
8  using Flux: onecold, onehotbatch
9
10 using ProgressMeter: Progress, next!
11
12 using MLDatasets
13 d = MNIST()
14
15 Base.@kwdef mutable struct HyperParams
16     η = 3f-3                # Learning rate
17     λ = 1f-2                # Regularization parameter
18     batchsize = 64          # Batch size
19     epochs = 16              # Number of epochs
20     split = :train           # Split data into `train` and `test`
21     input_dim = 28*28        # Input dimension
22     hidden_dim = 512         # Hidden dimension
23     latent_dim = 2           # Latent dimension
24     # save_path = "Output"   # Results folder
25 end
26
27 md"Load the **MNIST** dataset"
28 function get_data(; kws...)
29     args = HyperParams(; kws...);
30     md"Split data"
31     data = MNIST(split=args.split);
32     X = reshape(data.features, (args.input_dim, :));
33     loader = DataLoader(X; batchsize=args.batchsize, shuffle=true);
34     return loader
35 end
36
37 train_loader = get_data();
38 test_loader = get_data(split=:test);
39
40 md"Define the `encoder` network"
41 # The encoder network should return the parameters of the _latent distribution_ ( $\mu$  and  $\sigma$ ).
42 struct Encoder
43     linear
44     μ

```

```

45     log_σ
46 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),        # μ
53     Dense(hidden_dim, latent_dim),        # log_σ
54 )
55
56 function (encoder::Encoder)(x)
57     h = encoder.linear(x)
58     encoder.μ(h), encoder.log_σ(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     μ, log_σ = enc(x)
72     # `z` is a sample from the latent distribution
73     z = μ + randn(Float32, size(log_σ)) .* exp.(log_σ)
74     # Decode the latent representation into a reconstruction of `x`
75     x̃ = dec(z)
76     # Return μ, log_σ and x̃
77     μ, log_σ, x̃
78 end
79
80 function l(x, enc, dec, λ)
81     μ, log_σ, x̃ = 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(x, x̃, 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_σ - 1f0 + exp(2f0 * log_σ) + μ^2)) / len

```

```

87     # L2 Regularization
88     reg = λ * sum( θ -> sum(θ.^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_md1 = encoder(args.input_dim, args.hidden_dim, args.latent_dim)
98     dec_md1 = decoder(args.input_dim, args.hidden_dim, args.latent_dim)
99
100    # ADAM optimizers
101    opt_enc = Flux.setup(Adam(args.η), enc_md1)
102    opt_dec = Flux.setup(Adam(args.η), dec_md1)
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_md1, dec_md1) do enc, dec
109                l(X, enc, dec, args.λ)
110            end
111            grad_enc, grad_dec = back(1f0)
112            Flux.update!(opt_enc, enc_md1, grad_enc) # Upd `encoder` params
113            Flux.update!(opt_dec, dec_md1, 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_md1, dec_md1
131 end
132
133 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. **GANs** have been used for a variety of tasks, including image generation, text generation, and even music generation.

```

1 using Flux # v0.14.25
2 using Images: Gray
3 using ProgressMeter
4
5 ## Generator: noise vector -> synthetic sample.
```

```

6  function generator(; latent_dim=16, img_shape=(28,28,1,1))
7      return Chain(
8          Dense(latent_dim, 128, relu),
9          Dense(128, 256, relu),
10         Dense(256, prod(img_shape), tanh),
11         x -> reshape(x, img_shape)
12     )
13 end
14
15 ## Discriminator : sample -> score indicating the probability that the sample is real.
16 function discriminator(; img_shape=(28,28,1,1))
17     return Chain(
18         x -> reshape(x, :, size(x, 4)),
19         Dense(prod(img_shape), 256, relu),
20         Dense(256, 128, relu),
21         Dense(128, 1)
22     )
23 end
24
25 ## Loss functions
26 bce_loss(y_true, y_pred) = Flux.logitbinarycrossentropy(y_pred, y_true)
27
28 ## Training function
29 function train_gan(gen, disc, gen_opt, disc_opt; n_epochs=16, latent_dim=16)
30     @showprogress for epoch in 1:n_epochs
31
32         ## Train the discriminator 'disc'
33         noise = randn(Float32, latent_dim, 1)
34         fake_imgs = gen(noise) # pass the noise through the generator to get a
35         synthetic sample
36         real_imgs = rand(Float32, size(fake_imgs)...)
37
38         disc_loss = bce_loss(ones(Float32, 1, 1), disc(real_imgs)) +
39                     bce_loss(zeros(Float32, 1, 1), disc(fake_imgs)) # compute the
40         loss for the real and synthetic samples
41         grads = gradient(() -> disc_loss, Flux.params(disc))
42         Flux.update!(disc_opt, Flux.params(disc), grads) # update the discriminator
43         weights
44
45         ## Train the generator 'gen'
46         noise = randn(Float32, latent_dim, 1)
47         gen_loss = bce_loss( ones(Float32, 1, 1), σ.(disc(gen(noise))) ) # compute the
48         loss for the synthetic samples
49         grads = gradient(() -> gen_loss, Flux.params(gen))

```



```

46     Flux.update!(gen_opt, Flux.params(gen), grads) # update the generator weights
47
48     println("Epoch $(epoch): Discriminator loss = $(disc_loss), Generator loss =
49     ↪$(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 )

```



## 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.

```
1 using Markdown
2 using TextAnalysis
3
4 txt = "The quick brown fox is jumping over the lazy dog" # Pangram [modif.]
5
6 md"Create a `Corpus` using `txt`"
7 crps = Corpus([StringDocument(txt)])
8 lexicon(crps)
9 update_lexicon!(crps)
10 lexicon(crps)
11 lexical_frequency(crps, "fox")
12
13 md"Create a `StringDocument` using `txt`"
14 sd = StringDocument(txt)
15 md"Get a smaller set of words `text(sd)`"
16 prepare(sd, strip_articles | strip_numbers | strip_punctuation | strip_case |
17 strip_whitespace)
18 stem(sd)
19
20 md"Get the tokens of `sd`"
21 the_tokens = tokens(sd)
22
23 md"Get the stemmed tokens of `sd`"
24 stemmer = Stemmer("english")
25 stemmed_tokens = stem(stemmer, the_tokens)
26
27 println("Original tokens: ", the_tokens)
28 println("Stemmed tokens: ", stemmed_tokens)
29
30 md"**Part-of-speech tags**"
31
32 #=
33 Common POS tags:
34
35 JJ: Adjective
36 NN: Noun, singular or mass
37 NNS: Noun, plural
38 VB: Verb, base form
39 VBZ: Verb, 3rd person singular present
40 VBG: Verb, gerund or present participle
41 VBD: Verb, past tense
42 RB: Adverb
43 IN: Preposition or subordinating conjunction
```

```

43 DT: Determiner
44 PRP: Personal pronoun
45 CC: Coordinating conjunction
46 =#
47
48 #=
49 using TextModels
50 pos = PoSTagger()
51 pos(crps)
52 =#
53
54 md"**Word embeddings**"
55 using Embeddings
56 embtabs = load_embeddings(GloVe{:en}, max_vocab_size=5)
57
58 embtabs.vocab
59 embtabs.embeddings
60
61 glove = load_embeddings(GloVe{:en}, 3, max_vocab_size=10_000)
62 const word_to_index = Dict{String, Int}() for (ii, word) in enumerate(glove.vocab)
63 function get_word_vector(word)
64     idx = word_to_index[word]
65     return glove.embeddings[:, idx]
66 end
67
68 using LinearAlgebra
69 function cosine_similarity(v1::Vector{Float32}, v2::Vector{Float32})
70     return *(v1', v2) / *(norm(v1), norm(v2))
71 end
72
73 md"_e.g. - \"king\" - \"man\" + \"woman\" ≈ \"queen\"_"
74 king = get_word_vector("king")
75 queen = get_word_vector("queen")
76 man = get_word_vector("man")
77 woman = get_word_vector("woman")
78
79 cosine_similarity(king - man + woman, queen)
80
81 md"_e.g. - \"Madrid\" - \"Spain\" + \"France\" ≈ \"Paris\"_"
82 Madrid = get_word_vector("madrid")
83 Spain = get_word_vector("spain")
84 France = get_word_vector("france")
85 Paris = get_word_vector("paris")
86

```

```
87 cosine_similarity(Madrid - Spain + France, Paris)
88
89 md"**Text classification**"
90 md"https://github.com/JuliaText/TextAnalysis.jl/blob/master/docs/src/classify.md"
91 m = NaiveBayesClassifier([:legal, :financial])
92 fit!(m, "this is financial doc", :financial)
93 fit!(m, "this is legal doc", :legal)
94 predict(m, "this should be predicted as a legal document")
95
96 md"**Semantic analysis**"
97 m = DocumentTermMatrix(crps)
98
99 md"*Latent Semantic Analysis*"
100 lsa(m)
101
102 md"*Latent Dirichlet Allocation*"
103 k = 2 # number of topics
104 iterations = 1000 # number of Gibbs sampling iterations
105  $\alpha$  = 0.1 # hyper parameter
106  $\beta$  = 0.1 # hyper parameter
107 ,  $\theta$  = lda(m, k, iterations,  $\alpha$ ,  $\beta$ ) #
```



## 5 | Transfer Learning

<b>Student's name</b>	.....	.....	.....
	.....	.....	.....
	.....	.....	.....
<b>Score</b> /20	.....	.....	.....

### Detailed Credits

<b>Anticipation (4 points)</b>	.....	.....	.....
<b>Management (2 points)</b>	.....	.....	.....
<b>Testing (7 points)</b>	.....	.....	.....
<b>Data Logging (3 points)</b>	.....	.....	.....
<b>Interpretation (4 points)</b>	.....	.....	.....



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.

---

```

1 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
38         ↪"), 10))
39 end
40
41 for epoch in 1:100
42     Flux.train!(model, train_set, opt_state) do m, x, y
43         loss(m(x), y)

```

```

43     end
44 end
45
46 using ImageShow, ImageInTerminal
47 idx = rand(1:50000)
48 convert2image(d, idx)
49 printstyled("Label is ${d.targets[idx]}", bold=true, color=:red)
50
51 #=
52 using Optimisers
53 opt_state = Optimisers.setup(Adam(3e-3), mdl[3:end]) # Freeze the weights of the pre-
    ↳ trained layers
54 using ProgressMeter
55 epochs = 5
56 # Fine-tune the model
57 for epoch in 1:epochs
58     @showprogress for (X, y) in train_loader
59         # Compute the gradient of the loss with respect to the model's parameters
60         ∇ = Flux.gradient( m -> loss(m, X, y), mdl)
61         # Update the `mdl`'s parameters
62         Flux.update!(opt_state, mdl, ∇[1])
63     end
64     @info "Calculate the accuracy on the test set"
65     for (X, y) in test_loader
66         accuracy = sum(onecold(mdll(X)) .== onecold(y)) / length(y)
67         println("Epoch: $epoch, Accuracy: $accuracy")
68     end
69 end
70 =#

```





## 6 | Reinforcement Learning

<b>Student's name</b>	.....	.....	.....
	.....	.....	.....
	.....	.....	.....
<b>Score</b> /20	.....	.....	.....

### Detailed Credits

<b>Anticipation (4 points)</b>	.....	.....	.....
<b>Management (2 points)</b>	.....	.....	.....
<b>Testing (7 points)</b>	.....	.....	.....
<b>Data Logging (3 points)</b>	.....	.....	.....
<b>Interpretation (4 points)</b>	.....	.....	.....



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.

---

```
1 using ReinforcementLearning
2 using Flux: Descent
```

```
3
4  ## Define the environment
5  env = RandomWalk1D()
6
7  ## Instantiate the agent
8  agent = Agent(
9      policy = QBasedPolicy(
10         learner = TDLearner(
11             approximator = TabularQApproximator(
12                 n_state = 11,
13                 n_action = 2,
14                 init = 0.0,
15                 opt = Descent(0.1) # Learning rate
16             ),
17             method = :SARSA,
18             γ = 0.99
19         ),
20         explorer = EpsilonGreedyExplorer(0.1),
21     ),
22     trajectory = VectorSARTTrajectory(),
23 )
24
25 ## Run the experiment
26 hook = TotalRewardPerEpisode()
27 run(agent, env, StopAfterEpisode(10_000), hook)
28
29 ## Print rewards
30 println("Total reward per episode:")
31 println(hook.rewards)
32
33 ## Print `Q-table`
34 q_table = agent.policy.learner.approximator.table
35 println("\nLearned Q-table:")
36 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.

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  $\text{\LaTeX}$  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  $\text{\LaTeX}$ , 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 **ISSET 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