Term: M2-RAIA

Semester: 3

AY: 2022-2023

Abdelbacet Mhamdi

Dr.-Ing. in Electrical Engineering

Senior Lecturer at ISET Bizerte

abdelbacet.mhamdi@bizerte.r-iset.tn

Artificial Intelligence - Part 3

LAB MANUAL

INSTITUT SUPÉRIEUR DES ÉTUDES TECHNOLOGIQUES

Higher Institute of Technological Studies of Bizerte

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**, we recommend you to proceed as follow

① Press simultaneously the keys ⊞ & R on the keyboard. This will open the dialog box `Run`;

② Then enter `cmd` in the command line and confirm with ↵ key on the keyboard;

③ Type the instruction `jlai.bat` in the console prompt line;

```
Command Prompt

C:\Users\admin> jlai.bat
```

④ Finally press the ↵ key.

Leave the system console active.

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.jl*

CNN stands for Convolutional Neural Network, which is a 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.

This is just a rough example, and there are many details that have been left out (such as how to load and preprocess the data, how to use a validation set to evaluate the model, etc.). However, this should give you an idea of how a CNN can be implemented in Julia using the Flux.jl library.

JULIA STYLE PSEUDOCODE

```julia
using Flux
using Flux.Data.MNIST
using Flux: onehotbatch, onecold
using Base.Iterators: repeated, partition

# Load the MNIST dataset
X_train, y_train, X_test, y_test = MNIST.load()

# Convert the images to floating-point tensors
X_train = Flux.data(X_train)
X_test = Flux.data(X_test)

# Normalize the pixel values
X_train = X_train .- 128f0
X_test = X_test .- 128f0

# Split the dataset into training and validation sets
X_train, X_val, y_train, y_val = partition(X_train, y_train, 0.8)

# Convert the labels to one-hot encoding
y_train = onehotbatch(y_train, 0:9)
y_val = onehotbatch(y_val, 0:9)

# Define the model
model = Chain(
    Conv((3, 3), 1 => 8, pad=(1, 1), relu),
    x -> maxpool(x, (2, 2)),
    Conv((3, 3), 8 => 16, pad=(1, 1), relu),
    x -> maxpool(x, (2, 2)),
    x -> reshape(x, :, 16 * 7 * 7),
    Dense(16 * 7 * 7, 32, relu),
    Dense(32, 10),
    softmax
)

# Define the loss function and an optimizer
loss_fn = Flux.crossentropy
opt = Flux.ADAM()

# Train the model
for epoch in 1:10
    for (x, y) in zip(X_train, y_train)
        # Compute the gradient of the loss with respect to the model's␣
↪parameters
```

```
        gs = gradient(params(model)) do
            y_pred = model(x)
            loss_fn(y_pred, y)
        end

        # Update the model's parameters using the optimizer
        Flux.update!(opt, params(model), gs)
    end

    # Calculate the accuracy on the validation set
    accuracy = sum(onecold(model(X_val)) .== onecold(y_val)) / length(y_val)

    println("Epoch: $epoch, Accuracy: $accuracy")
end

# Test the model on the test set
accuracy = sum(onecold(model(X_test)) .== onecold(y_test)) / length(y_test)
println("Test accuracy: $accuracy")
```

2 | 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.jl*

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.

JULIA STYLE PSEUDOCODE

```julia
using Flux
using Flux.Data.MNIST
using Flux: onehotbatch, onecold
using Flux.Data: DataLoader

# Load the MNIST dataset
X_train, y_train, X_test, y_test = MNIST.load()

# Convert the images to floating-point tensors
X_train = Flux.data(X_train)
X_test = Flux.data(X_test)

# Normalize the pixel values
X_train = X_train .- 128f0
X_test = X_test .- 128f0

# Define the generator network
function generator(z)
    z = Dense(256, relu)(z)
    z = Dense(512, relu)(z)
    z = Dense(784, tanh)(z)
    z = reshape(z, 28, 28, 1)
    return z
end

# Define the discriminator network
function discriminator(x)
    x = reshape(x, :, 784)
    x = Dense(512, relu)(x)
    x = Dense(256, relu)(x)
    x = Dense(1, sigmoid)(x)
    return x
end

# Define the GAN
function GAN(z)
    x_hat = generator(z)
    return discriminator(x_hat)
end

# Define the loss function
function loss(x, x_hat)
    reconstruction_loss = Flux.mse(x, x_hat)
    return reconstruction_loss
end

# Create a data iterator
batch_size = 128
train_data = DataLoader(X_train, batch_size, shuffle=true)

# Define the optimizers
g_opt = Flux.ADAM()
d_opt = Flux.ADAM()

# Train the GAN
for epoch in 1:10
    for (x, _) in train_data
        # Train the discriminator
        z = randn(100)
        x_hat = generator(z)
        d_loss = Flux.mse(discriminator(x), 1) + Flux.
 ↪mse(discriminator(x_hat), 0)
        Flux.back!(d_loss)
        Flux.update!(d_opt)

        # Train the generator
        z = randn(100)
        x_hat = generator(z)
        g_loss = loss(x, x_hat)
        Flux.back!(g_loss)
        Flux.update!(g_opt)
    end
    println("Epoch: $epoch, D Loss: $(mean(d_loss)), G Loss: $(mean(g_
 ↪loss))")
end

# Generate samples from the GAN
z = randn(100)
x_hat = generator(z)
Flux.mse(X_test[1], x_hat)
```

3 | 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.jl*

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.

Julia Style Pseudocode

```
using Flux
using Flux.Data.MNIST
using Flux: onehotbatch, onecold
using Flux.Data: DataLoader

# Load the MNIST dataset
X_train, y_train, X_test, y_test = MNIST.load()

# Convert the images to floating-point tensors
X_train = Flux.data(X_train)
X_test = Flux.data(X_test)

# Normalize the pixel values
X_train = X_train .- 128f0
X_test = X_test .- 128f0

# Define the encoder network
function encoder(x)
    x = reshape(x, :, 784)
    x = Dense(512, relu)(x)
    x = Dense(256, relu)(x)
    mu = Dense(2)(x)
    logvar = Dense(2)(x)
    return mu, logvar
end

# Define the decoder network
function decoder(z)
    z = Dense(256, relu)(z)
    z = Dense(512, relu)(z)
    z = Dense(784, sigmoid)(z)
    z = reshape(z, 28, 28, 1)
    return z
end

# Define the VAE
function VAE(x)
    mu, logvar = encoder(x)
    z = mu .+ exp.(logvar) .* randn(2)
    x_hat = decoder(z)
```

```julia
    return x_hat, mu, logvar
end

# Define the loss function
function loss(x, x_hat, mu, logvar)
    reconstruction_loss = Flux.mse(x, x_hat)
    kl_divergence = -0.5 * sum(1 .+ logvar .- mu .^ 2 .- exp.(logvar))
    return reconstruction_loss + kl_divergence
end

# Create a data iterator
batch_size = 128
train_data = DataLoader(X_train, batch_size, shuffle=true)

# Define the optimizer
opt = Flux.ADAM()

# Train the VAE
for epoch in 1:10
    for (x, _) in train_data
        x_hat, mu, logvar = VAE(x)
        L = loss(x, x_hat, mu, logvar)
        Flux.back!(L)
        Flux.update!(opt)
    end
    println("Epoch: $epoch, Loss: $(mean(L))")
end

# Generate samples from the VAE
z = randn(2)
x_hat = decoder(z)
Flux.mse(X_test[1], x_hat)
```

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.jl*

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. There are many different types of models that can be used for **NLP** tasks, 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. This may involve calculating metrics such as accuracy, precision, and recall.

- Use the trained model to make predictions on new, unseen text data.

Julia Style Pseudocode

```julia
using Flux, Flux.Data.MNIST
using Flux: onehotbatch, onecold
using Base.Iterators: partition

# Load the IMDB movie review dataset
X_train, y_train, X_test, y_test = IMDB.load()

# Preprocess the text data
X_train = lowercase.(X_train)
X_test = lowercase.(X_test)

X_train = replace.(X_train, r"<[^>]*>" => "")
X_test = replace.(X_test, r"<[^>]*>" => "")

X_train = replace.(X_train, r"[^a-zA-Z]" => " ")
X_test = replace.(X_test, r"[^a-zA-Z]" => " ")

X_train = map(x -> split(x, " "), X_train)
X_test = map(x -> split(x, " "), X_test)

# Build the vocabulary
vocab = Dict{String,Int}()

for review in X_train
    for word in review
        if !haskey(vocab, word)
            vocab[word] = length(vocab) + 1
        end
    end
end

# Encode the text data as a sequence of integers
X_train = map(x -> map(y -> get(vocab, y, 1), x), X_train)
X_test = map(x -> map(y -> get(vocab, y, 1), x), X_test)

# Pad the encoded sequences to the same length
max_length = maximum(length.(X_train))

```

```
X_train = map(x -> vcat(x, fill(1, max_length - length(x))), X_train)
X_test = map(x -> vcat(x, fill(1, max_length - length(x))), X_test)

# Convert the labels to one-hot encoding
y_train = onehotbatch(y_train, [0, 1])
y_test = onehotbatch(y_test, [0, 1])

# Split the dataset into training and validation sets
X_train, X_val, y_train, y_val = partition(X_train, y_train, 0.8)

# Define the model
model = Chain(
    Embedding(length(vocab), 32),
    LSTM(32,
```

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.jl*

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. By using transfer learning, it is possible to take advantage of the features and knowledge learned by the original model, and apply them to the new task, often resulting in improved performance.

There are several ways to use transfer learning, including fine-tuning the weights of the pre-trained model on the new task, using the pre-trained model as a fixed feature extractor, or simply 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.

Here's an example of `Julia` code that demonstrates how to perform transfer learning using a pre-trained model in the `Flux.jl` library.

Julia Style Pseudocode

```julia
using Flux
using Flux.Data.MNIST
using Flux: onehotbatch, onecold
using Flux.Data: DataLoader

# Load the MNIST dataset
X_train, y_train, X_test, y_test = MNIST.load()

# Convert the images to floating-point tensors
X_train = Flux.data(X_train)
X_test = Flux.data(X_test)

# Normalize the pixel values
X_train = X_train .- 128f0
X_test = X_test .- 128f0

# Convert the labels to one-hot encoding
y_train = onehotbatch(y_train, 0:9)
y_test = onehotbatch(y_test, 0:9)

# Load a pre-trained model
model = Flux.load("pretrained_model.bson")

# Replace the top layer of the model with a new, untrained layer
model.layers[end] = Dense(10, softmax)

# Define the loss function and an optimizer
loss_fn = Flux.crossentropy
opt = Flux.ADAM()

# Train the model
for epoch in 1:10
    for (x, y) in zip(X_train, y_train)
        # Compute the gradient of the loss with respect to the model's␣
↪parameters
        gs = gradient(params(model)) do
            y_pred = model(x)
            loss_fn(y_pred, y)
        end

        # Update the model's parameters using the optimizer
        Flux.update!(opt, params(model), gs)
    end

    # Calculate the accuracy on the test set
    accuracy = sum(onecold(model(X_test)) .== onecold(y_test)) / length(y_test)

    println("Epoch: $epoch, Accuracy: $accuracy")
end
```

This code loads a pre-trained model from a file, replaces the top layer with a new, untrained layer, and fine-tunes the model on the **MNIST** dataset. The new layer has 10 output units corresponding to the 10 classes in the **MNIST** dataset.

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.jl*

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, game playing, and natural language processing.
- 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.
- It is an active area of research, with many open questions and challenges that need to be addressed in order to improve the performance and capabilities of reinforcement learning algo-

rithms.

Julia Style Pseudocode

```
# Initialize the environment and the agent
env = Environment()
agent = Agent()

# Set the number of iterations and the discount factor
num_iterations = 1000
discount_factor = 0.9

# Loop through the number of iterations
for i in 1:num_iterations
    # Reset the environment and get the initial state
    state = env.reset()

    # Set a flag to indicate when the episode is done
    done = false

    # Loop until the episode is done
    while !done
        # Choose an action using the agent's policy
        action = agent.choose_action(state)
        # Take the action and observe the reward and next state
        reward, next_state, done = env.step(action)
        # Update the agent's policy using the reward and next state
        agent.update_policy(state, action, reward, next_state,
 ↪discount_factor)
        # Set the state to the next state
        state = next_state
    end
end
```

This pseudo-code shows the basic structure of a reinforcement learning algorithm, where an agent learns to interact with an environment in order to maximize a reward. The agent starts in an initial state, chooses an action based on its current policy, receives a reward and moves to a new state, and then updates its policy based on the reward and the new state. This process is repeated until the episode is done.

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 used `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 enrolled in the master's program **RAIA**.

The topics discussed in this manuscript are as follow:

① CNN (Convolutional Neural Network)

image classification; computer vision; feature learning.

② GAN (Generative Adversarial Network)

data generation; image generation; adversarial training.

③ VAE (Variational Autoencoder)

image generation; anomaly detection; latent representation.

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