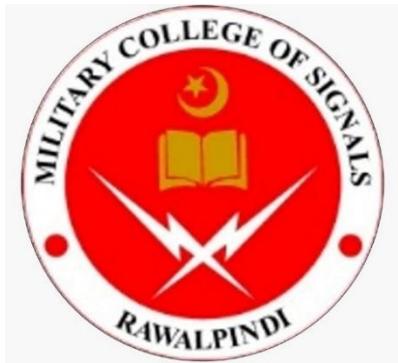


Deep Learning Lab Manual (CS 405)

COMPUTER LABORATORY MANUAL



Deep Learning
(CS 405)
Spring -2025

DEPARTMENT OF COMPUTER SOFTWARE ENGINEERING
Military College of Signals
National University of Sciences and Technology
www.mcs.nust.edu.pk

Deep Learning Lab Manual (CS 405)

PREFACE

This lab manual has been prepared to facilitate the students of software engineering in studying and analysing various functions of a computer network.

PREPARED BY

Lab manual is prepared by Lab Engineer Zohaib Ali Shah and Asst. Prof. Dr Maemoona Kiyani under the supervision of Head of Department Spring 2025.

GENERAL INSTRUCTIONS

- a. Students are required to maintain the lab manual with them till the end of the semester.
- b. All readings, answers to questions and illustrations must be solved on the place provided. If more space is required, then additional sheets may be attached. You may add screen print to the report by using the 'Print Screen' command on your keyboard to get a snapshot of the displayed output.
- c. It is the responsibility of the student to have the manual graded before deadlines as given by the instructor
- d. Loss of manual will result in re submission of the complete manual.
- e. Students are required to go through the experiment before coming to the lab session. Lab session details will be given in training schedule.
- f. Students must bring the manual in each lab.
- g. Keep the manual neat clean and presentable.
- h. Plagiarism is strictly forbidden. No credit will be given if a lab session is plagiarised and no re submission will be entertained.
- i. Marks will be deducted for late submission.
- j. In the exercises, you have to put the output in your Lab report.
- k. Name your reports using the following convention:
Lab#_Rank_YourFullName
(1) '#' replaces the lab number.
(2) 'Rank' replaces Maj/Capt/TC/NC/PC
(3) 'YourFullName' replaces your complete name.
- l. You need to submit the report even if you have demonstrated the exercises to the lab engineer/instructor or shown them the lab report during the lab session.

VERSION HISTORY

Date	Update By	Details
April 2025	Lab Engineer Zohaib Ali Shah and Asst. Prof. Dr Maemoona Kiyani	First Version Created

Deep Learning Lab Manual (CS 405)

COURSE LEVEL OUTCOMES

Course Outcomes:

At the end of this course, the students will be able to attain the CLOs in the table below.

CS 405 - Deep Learning			
Course Learning Outcomes (CLOs)			
At the end of the course the students will be able to:		PLOs	BT Level*
1	Comprehend the specialized architectures, such as convolutional neural networks (CNNs), recurrent neural networks (RNN) across different domains.	1	C-2
2	Demonstrate the performance of different deep learning models critically using appropriate metrics.	2	C-3
3	Investigate the ethical implications of deploying AI systems in various contexts.	8	A-4
4	Gain the ability to design, implement, and train various deep learning models using Tensorflow to solve real-world problems.	5	P-5

Mapping of CLOs to Program Learning Outcomes

PLOs/CLOs	CLO1	CLO2	CLO3	CLO4
PLO 1 (Engineering Knowledge)	X			
PLO 2 (Problem Analysis)		X		
PLO 3 (Design/Development of Solutions)				
PLO 4 (Investigation)				
PLO 5 (Modern tool usage)				X
PLO 6 (The Engineer and Society)				
PLO 7 (Environment and Sustainability)				
PLO 8 (Ethics)			X	
PLO 9 (Individual and Team Work)				
PLO 10 (Communication)				
PLO 11 (Project Management)				
PLO 12 (Lifelong Learning)				

Deep Learning Lab Manual (CS 405)

S No	List of Experiments	CLO	R-G
1	Experiment 1 – Design And Implementation of A Single-Layer Perceptron Model	4	
2	Experiment 2 – Implement the Fundamentals And Practical Use Of Pytorch For Deep Learning	4	
3	Experiment 3 –Implement the Music Generation Using Recurrent Neural Networks In Pytorch	4	
4	Experiment 4 – Train And Implement the Handwritten Digit Classification Using Convolutional Neural Networks (Cnns) On The Mnist Dataset	4	
5	Experiment 5 – Implement the Image Denoising And Feature Learning Using Autoencoders	4	
6	Experiment 6: Implement Generating Novel Fashion Images Using Variational Autoencoders (Vaes)	4	
7	Experiment 7 – Train the Generative Modeling Using Generative Adversarial Networks (Gans)	4	
8	Experiment No: 8. Implement the Reinforcement Learning With Openai Gym Using The Frozenlake Environment	4	
9	Experiment No 9: Open Ended Lab	3, 4	
10	Experiment 10: Implement the Deep Reinforcement Learning For Agent-Environment Interaction	4	
11	Experiment No 11: Train and implement End-To-End Training Of Autonomous Driving Policies In Vista Simulator	4	
12	Experiment 12: Enhancing Deep Learning Trustworthiness Through Debiasing And Uncertainty Quantification	4	
13	Experiment 13: Implement Automated Debiasing And Uncertainty Reduction In Facial Detection Systems Using Capsa	4	
14	Experiment: 14 Comparative Analysis Of Simple Rnn And Bi-Directional Lstm For Sentiment Analysis On Movie Reviews	4	
15	Experiment: 14 Design and Implement the CNN with Regularization and Dropout on Fashion MNIST Dataset	2,4	
16	Final Lab Project/Exam		

Deep Learning Lab Manual (CS 405)

Mapping of Lab Experiments

Lab Rubrics

Criteria	Unacceptable (Marks=0)	Substandard Marks=1	Adequate Marks=2	Proficient Marks=3
R1 Completeness And Accuracy	The program failed to produce the right accurate result	The program execution let to inaccurate or incomplete results. It was not correctly functional or not all the features were implemented	The program was correctly functional and most of the features were implemented	The program was correctly functional, and all the features were implemented
R2 Syntax and Semantics	The student fails to figure out the syntax and semantic errors of the incorrect program	Student successfully figures out few of syntax and semantic errors of the program with extensive guidance	Student successfully figures out most of syntax and semantic errors of the program with minimum guidance	Student successfully figures out all syntax and semantic errors of the program without any guidance
R3 Demonstration	Student failed to demonstrate a clear understanding of the assigned task	Student has basic understanding, but asked questions were not answered.	Student has basic knowledge of understanding. Provides fundamental answers to asked questions	Student has demonstrated on accurate understanding of the lab objective and concepts. All the questions are answered completely and correctly
R4 Report Writing and Formatting	No formatting is done	Formatting guidelines barely followed	Formatting guidelines adequately followed	Written work is very well formatted and well-written
R5 Perseverance and plagiarism	Complete working program is copied indicating no effort on student's part resulting in a total score of zero for all rubrics	Most of working program is copied. Minor contribution by the student	Most of working program is contributed by the student. Minor copied components	Complete working program is contributed by the student

Deep Learning Lab Manual (CS 405)

Lab Rubrics (Open Ended Lab)

S.No	Components	Excellent	Good	Basic	Just Acceptable	Unacceptable
1	Background knowledge	Thorough study and all the questions have been answered correctly.	Adequate study and more than half of the questions have been answered correctly.	Sufficient study and half of the questions have been answered correctly.	Inadequate study and less than half of the questions have been answered correctly.	No study and none of the questions have been answered.
2	Select Appropriate Equipment & Tools	Relevant and smart selection of equipments and tools to achieve desired objective along with proper reasoning.	Satisfactory selection of equipments and tools to achieve desired objective along with proper reasoning.	Satisfactory selection of equipment's and tools to achieve desired objective without proper reasoning.	Unsatisfactory selection of equipments and tools to achieve desired objective.	No use of equipments and tools leading towards major errors in the objective.
3	Analysis & Results	Appropriate data collected, correct analysis & results correctly interpreted.	Appropriate data collected but insufficient analysis & results adequately interpreted.	Inappropriate data collected but sufficient analysis & results inadequately interpreted.	Inappropriate data collected, no understanding of analysis & results incorrectly interpreted.	No data collected, analysis & results interpretation.
4	Conclusions and Recommendations	Significant findings are summarized. Precisely concluded. Excellent suggestion for further research.	Significant findings are summarized. Good conclusion. Good suggestion for further research.	Significant findings are summarized. Acceptable conclusion. Acceptable suggestion for further research.	Findings are poorly summarized. Poor conclusion. Poor suggestion for further research.	No findings are summarized. Poor conclusion. No suggestion for further research.
5	Report Writing	Report meets all requirements and it is prepared in original and creative way to engage readers.	Report meets all prescribed requirements.	The requirements of report writing are not properly addressed.	The report submitted but not according to requirements.	Report was not prepared or they have not required elements.

Deep Learning Lab Manual (CS 405)

Contents

Experiment 1 – Design And Implementation of A Single-Layer Perceptron Model	4
EXPERIMENT 1 – Design and Implementation of a Single-Layer Perceptron Model	8
EXPERIMENT 2 – Implement Fundamentals and Practical Use of PyTorch for Deep Learning.....	12
EXPERIMENT 3 – Implement Music Generation using Recurrent Neural Networks in PyTorch	21
EXPERIMENT 4 – Train and Implement Handwritten Digit Classification using Convolutional Neural Networks (CNNs) on the MNIST Dataset	28
EXPERIMENT 5 – Implement Image Denoising and Feature Learning using Autoencoders	42
Experiment 6: Train and Implement Generating Novel Fashion Images using Variational Autoencoders (VAEs)	49
EXPERIMENT 7 – Train the Generative Modeling using Generative Adversarial Networks (GANs)	56
Experiment No: 8. Implement the Reinforcement Learning with OpenAI Gym using the FrozenLake Environment.....	62
Experiment No 9: OPEN ENDED LAB	66
"Exploring Bias and Uncertainty in AI: Ethical and Performance Perspectives"	66
Experiment 10: Implement the Deep Reinforcement Learning for Agent-Environment Interaction	68
Experiment No 11: Train and Implement the End-to-End Training of Autonomous Driving Policies in VISTA Simulator.....	79
Experiment 12: Enhancing Deep Learning Trustworthiness through Debiasing and Uncertainty Quantification	94
Experiment 13: Implement the Automated Debiasing and Uncertainty Reduction in Facial Detection Systems using CAPSA	103
Experiment: 14 Comparative Analysis of Simple RNN and Bi-directional LSTM for Sentiment Analysis on Movie Reviews	115
Experiment: 15 Design and Implement the CNN with Regularization and Dropout on Fashion MNIST Dataset	119
Final Lab Project.....	125

EXPERIMENT 1 – Design and Implementation of a Single-Layer Perceptron Model

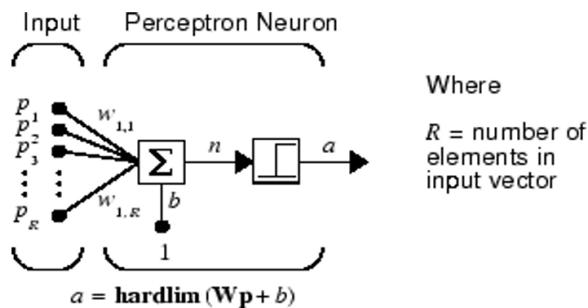
Objective

To introduce the concept and working of a perceptron neural network by implementing a single-layer perceptron that classifies inputs based on a step activation function using weights and bias.

Introduction

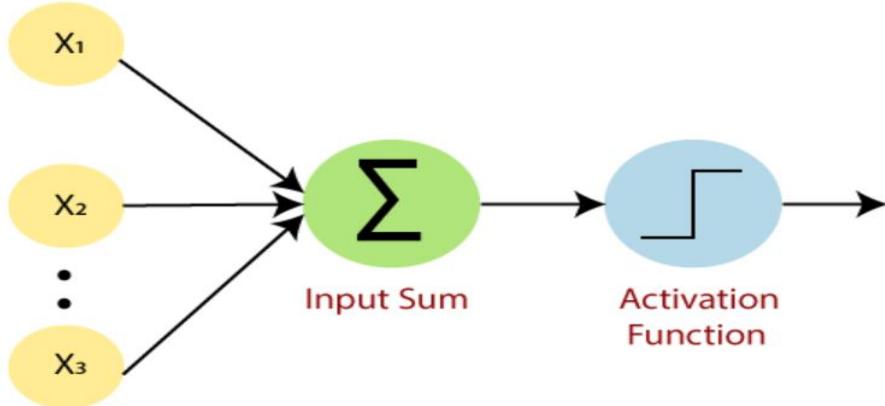
Perceptron Neural Networks Rosenblatt created many variations of the perceptron. One of the simplest was a single-layer network whose weights and biases could be trained to produce a correct target vector when presented with the corresponding input vector. The training technique used is called the perceptron learning rule. The perceptron generated great interest due to its ability to generalize from its training vectors and learn from initially randomly distributed connections. Perceptron are especially suited for simple problems in pattern classification. They are fast and reliable networks for the problems they can solve. In addition, an understanding of the operations of the perceptron provides a good basis for understanding more complex networks.

Neuron Model A perceptron neuron, which uses the hard-limit transfer function hardlim, is shown below.

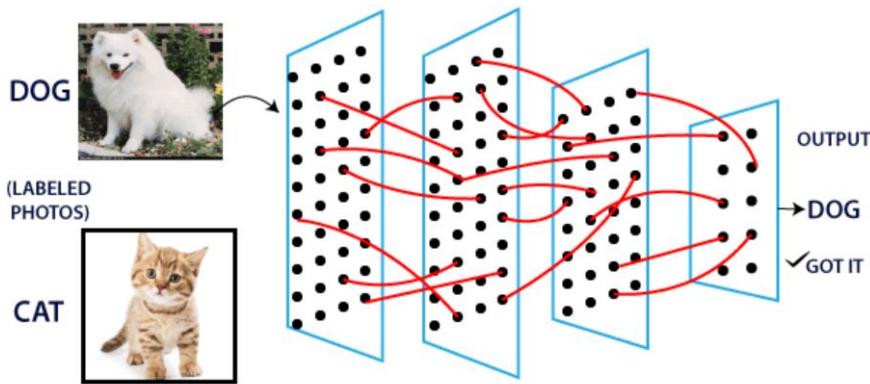


The perceptron consists of 4 parts.

- **Input value or One input layer:** The input layer of the perceptron is made of artificial input neurons and takes the initial data into the system for further processing.
- **Weights and Bias:**
 - Weight:** It represents the dimension or strength of the connection between units. If the weight to node 1 to node 2 has a higher quantity, then neuron 1 has a more considerable influence on the neuron.
 - Bias:** It is the same as the intercept added in a linear equation. It is an additional parameter which task is to modify the output along with the weighted sum of the input to the other neuron.
- **Net sum:** It calculates the total sum.
- **Activation Function:** A neuron can be activated or not, is determined by an activation function. The activation function calculates a weighted sum and further adding bias with it to give the result.

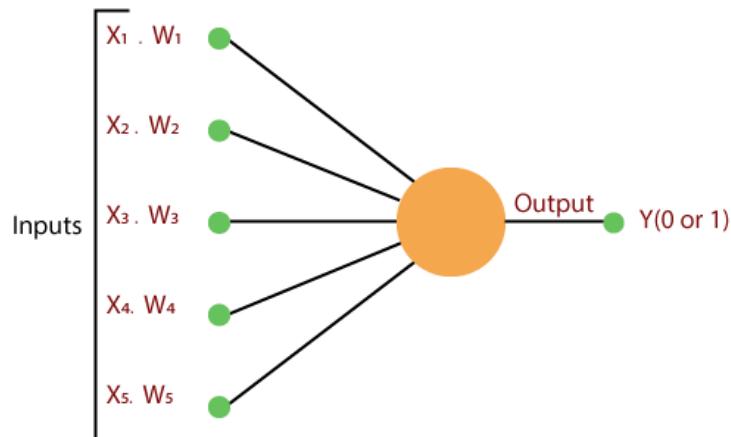


A standard neural network looks like the below diagram.



The perceptron works on these simple steps which are given below:

- In the first step, all the inputs x are multiplied with their weights w .



- In this step, add all the increased values and call them the **Weighted sum**.

Deep Learning Lab Manual (CS 405)

$$\sum_{K=1}^5 K$$

term we end with
Sigma for Summation
the Formula for the nth term
K is the index (It's like a Counter. Some books Use i.)
the term we start with

- c. In our last step, apply the weighted sum to a correct **Activation Function**.

Task:

Write a Code to Implement the perceptron.

```
class Perceptron:  
    def __init__(self, weights, bias):  
        self.weights = weights  
        self.bias = bias  
  
    def activation(self, x):  
        return 1 if x >= 0 else 0 # Step function  
  
    def predict(self, inputs):  
        weighted_sum = sum(w * i for w, i in zip(self.weights, inputs)) + self.bias  
        return self.activation(weighted_sum)  
  
# Inputs: weight (grams) and color (scale 0 to 1)  
inputs = [150, 0.9]  
weights = [0.5, 1.0] # Given weights  
bias = 1.5 # Given bias  
  
# Create perceptron instance  
perceptron = Perceptron(weights, bias)  
  
# Make prediction  
output = perceptron.predict(inputs)  
  
# Display result  
print("Prediction:", "Apple" if output == 1 else "Not an Apple")
```

Deep Learning Lab Manual (CS 405)

Explanation:

1. Perceptron Class:

- o Takes weights and bias as input.
- o Uses a **step activation function**: If the weighted sum is ≥ 0 , it outputs **1 (Apple)**; otherwise, **0 (Not an Apple)**.

2. Inputs:

- o Weight = **150g**.
- o Color = **0.9** (closer to red).

3. Calculation:

$$(150 \times 0.5) + (0.9 \times 1.0) + 1.5 = 76.4$$

Since **76.4 ≥ 0** , the perceptron outputs **1 (Apple)**.

EXPERIMENT 2 – Implement Fundamentals and Practical Use of PyTorch for Deep Learning

Objective:

To provide hands-on experience with PyTorch by exploring tensor operations, computational graphs, and neural network modules for deep learning tasks in Python.

In this lab, you'll get exposure to using PyTorch and learn how it can be used for deep learning. Go through the code and run each cell. Along the way, you'll encounter several *TODO* blocks -- follow the instructions to fill them out before running those cells and continuing.

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

Intro to PyTorch

0.1 Install PyTorch

[PyTorch](<https://pytorch.org/>) is a popular deep learning library known for its flexibility and ease of use. Here we'll learn how computations are represented and how to define a simple neural network in PyTorch. For all the labs in Introduction to Deep Learning 2025, there will be a PyTorch version available.

Let's install PyTorch and a couple of dependencies.

```
import torch
import torch.nn as nn

# Download and import the MIT Introduction to Deep Learning package
!pip install mitdeeplearning -q
import mitdeeplearning as mdl

import numpy as np
import matplotlib.pyplot as plt
```

What is PyTorch?

PyTorch is a machine learning library, like TensorFlow. At its core, PyTorch provides an interface for creating and manipulating [tensors](<https://pytorch.org/docs/stable/tensors.html>), which are data structures that you can

Deep Learning Lab Manual (CS 405)

think of as multi-dimensional arrays. Tensors are represented as n-dimensional arrays of base datatypes such as a string or integer -- they provide a way to generalize vectors and matrices to higher dimensions. PyTorch provides the ability to perform computation on these tensors, define neural networks, and train them efficiently.

The [``shape``](<https://pytorch.org/docs/stable/generated/torch.Tensor.shape.html#torch.Tensor.shape>) of a PyTorch tensor defines its number of dimensions and the size of each dimension. The `ndim` or [``dim``](<https://pytorch.org/docs/stable/generated/torch.Tensor.dim.html#torch.Tensor.dim>) of a PyTorch tensor provides the number of dimensions (n-dimensions) -- this is equivalent to the tensor's rank (as is used in TensorFlow), and you can also think of this as the tensor's order or degree.

Let's start by creating some tensors and inspecting their properties:

```
integer = torch.tensor(1234)
decimal = torch.tensor(3.14159265359)

print(f"`integer` is a {integer.ndim}-d Tensor: {integer}")
print(f"`decimal` is a {decimal.ndim}-d Tensor: {decimal}")
```

Vectors and lists can be used to create 1-d tensors:

```
fibonacci = torch.tensor([1, 1, 2, 3, 5, 8])
count_to_100 = torch.tensor(range(100))

print(f"`fibonacci` is a {fibonacci.ndim}-d Tensor with shape: {fibonacci.shape}")
print(f"`count_to_100` is a {count_to_100.ndim}-d Tensor with shape: {count_to_100.shape}")
```

Next, let's create 2-d (i.e., matrices) and higher-rank tensors. In image processing and computer vision, we will use 4-d Tensors with dimensions corresponding to batch size, number of color channels, image height, and image width.

```
### Defining higher-order Tensors ###

"""TODO: Define a 2-d Tensor"""
matrix = # TODO

assert isinstance(matrix, torch.Tensor), "matrix must be a torch Tensor object"
assert matrix.ndim == 2

"""TODO: Define a 4-d Tensor."""
# Use torch.zeros to initialize a 4-d Tensor of zeros with size 10 x 3 x 256 x 256.
# You can think of this as 10 images where each image is RGB 256 x 256.
images = # TODO

assert isinstance(images, torch.Tensor), "images must be a torch Tensor object"
assert images.ndim == 4, "images must have 4 dimensions"
assert images.shape == (10, 3, 256, 256), "images is incorrect shape"
```

Deep Learning Lab Manual (CS 405)

```
print(f"images is a {images.ndim}-d Tensor with shape: {images.shape}")
```

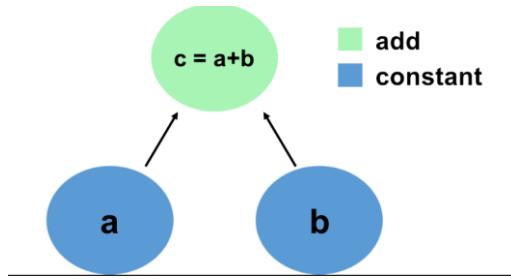
As you have seen, the `shape` of a tensor provides the number of elements in each tensor dimension. The `shape` is quite useful, and we'll use it often. You can also use slicing to access subtensors within a higher-rank tensor:

```
row_vector = matrix[1]
column_vector = matrix[:, 1]
scalar = matrix[0, 1]

print(f"row_vector: {row_vector}")
print(f"column_vector: {column_vector}")
print(f"scalar: {scalar}")
```

1.2 Computations on Tensors

A convenient way to think about and visualize computations in a machine learning framework like PyTorch is in terms of graphs. We can define this graph in terms of tensors, which hold data, and the mathematical operations that act on these tensors in some order. Let's look at a simple example, and define this computation using PyTorch:



Hadoop is a software framework from Apache Software Foundation which is used to store and process Big Data. In this article I've compiled the steps to install and run Hadoop on Windows

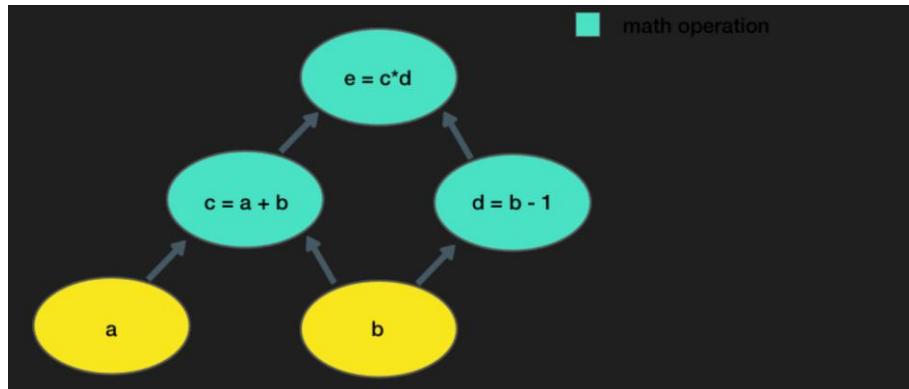
```
# Create the nodes in the graph and initialize values
a = torch.tensor(15)
b = torch.tensor(61)

# Add them!
c1 = torch.add(a, b)
c2 = a + b # PyTorch overrides the "+" operation so that it is able to act on Tensors
print(f"c1: {c1}")
print(f"c2: {c2}")
```

Notice how we've created a computation graph consisting of PyTorch operations, and how the output is a tensor with value 76 -- we've just created a computation graph consisting of operations, and it's executed them and given us back the result.

Now let's consider a slightly more complicated example:

Deep Learning Lab Manual (CS 405)



Here, we take two inputs, `a, b`, and compute an output `e`. Each node in the graph represents an operation that takes some input, does some computation, and passes its output to another node.

Let's define a simple function in PyTorch to construct this computation function:

```
### Defining Tensor computations ###
```

```
# Construct a simple computation function
def func(a, b):
    "TODO: Define the operation for c, d, e."
    c = # TODO
    d = # TODO
    e = # TODO
    return e
```

Now, we can call this function to execute the computation graph given some inputs `a,b`:

```
# Consider example values for a,b
a, b = 1.5, 2.5
# Execute the computation
e_out = func(a, b)
print(f"e_out: {e_out}")
```

Notice how our output is a tensor with value defined by the output of the computation, and that the output has no shape as it is a single scalar value.

1.3 Neural networks in PyTorch

We can also define neural networks in PyTorch. PyTorch uses [``torch.nn.Module``](<https://pytorch.org/docs/stable/generated/torch.nn.Module.html>), which serves as a base class for all neural network modules in PyTorch and thus provides a framework for building and training neural networks.

Let's consider the example of a simple perceptron defined by just one dense (aka fully-connected or linear) layer: $y = \sigma(Wx + b)$, where W represents a matrix of weights, b is a bias, x is the input, σ is the sigmoid activation function, and y is the output.

Deep Learning Lab Manual (CS 405)

We will use `torch.nn.Module` to define layers -- the building blocks of neural networks. Layers implement common neural networks operations. In PyTorch, when we implement a layer, we subclass `nn.Module` and define the parameters of the layer as attributes of our new class. We also define and override a function [`forward`](https://pytorch.org/docs/stable/generated/torch.nn.Module.html#torch.nn.Module.forward), which will define the forward pass computation that is performed at every step. All classes subclassing `nn.Module` should override the `forward` function.

Let's write a dense layer class to implement a perceptron defined above.

```
### Defining a dense layer ###
```

```
# num_inputs: number of input nodes
# num_outputs: number of output nodes
# x: input to the layer

class OurDenseLayer(torch.nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(OurDenseLayer, self).__init__()
        # Define and initialize parameters: a weight matrix W and bias b
        # Note that the parameter initialize is random!
        self.W = torch.nn.Parameter(torch.randn(num_inputs, num_outputs))
        self.bias = torch.nn.Parameter(torch.randn(num_outputs))

    def forward(self, x):
        """TODO: define the operation for z (hint: use torch.matmul)."""
        z = # TODO

        """TODO: define the operation for out (hint: use torch.sigmoid)."""
        y = # TODO
        return y
```

Now, let's test the output of our layer..

```
# Define a layer and test the output!
num_inputs = 2
num_outputs = 3
layer = OurDenseLayer(num_inputs, num_outputs)
x_input = torch.tensor([[1, 2.]])
y = layer(x_input)

print(f"input shape: {x_input.shape}")
print(f"output shape: {y.shape}")
print(f"output result: {y}")
```

Deep Learning Lab Manual (CS 405)

Conveniently, PyTorch has defined a number of ```nn.Modules```` (or Layers) that are commonly used in neural networks, for example a [```nn.Linear````](<https://pytorch.org/docs/stable/generated/torch.nn.Linear.html>) or [```nn.Sigmoid````](<https://pytorch.org/docs/stable/generated/torch.nn.Sigmoid.html>) module.

Now, instead of using a single ```Module```` to define our simple neural network, we'll use the [```nn.Sequential````](<https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html>) module from PyTorch and a single [```nn.Linear````](<https://pytorch.org/docs/stable/generated/torch.nn.Linear.html>) layer to define our network. With the `Sequential` API, you can readily create neural networks by stacking together layers like building blocks.

```
### Defining a neural network using the PyTorch Sequential API ###

# define the number of inputs and outputs
n_input_nodes = 2
n_output_nodes = 3

# Define the model
'''TODO: Use the Sequential API to define a neural network with a
single linear (dense!) layer, followed by non-linearity to compute z'''
model = nn.Sequential( " TODO " )
```

We've defined our model using the Sequential API. Now, we can test it out using an example input:

```
# Test the model with example input
x_input = torch.tensor([[1, 2.]])
model_output = model(x_input)
print(f"input shape: {x_input.shape}")
print(f"output shape: {y.shape}")
print(f"output result: {y}")
```

With PyTorch, we can create more flexible models by subclassing [```nn.Module````](<https://pytorch.org/docs/stable/generated/torch.nn.Module.html>). The `nn.Module` class allows us to group layers together flexibly to define new architectures.

As we saw earlier with `OurDenseLayer`, we can subclass `nn.Module` to create a class for our model, and then define the forward pass through the network using the `forward` function. Subclassing affords the flexibility to define custom layers, custom training loops, custom activation functions, and custom models. Let's define the same neural network model as above (i.e., Linear layer with an activation function after it), now using subclassing and using PyTorch's built in linear layer from `nn.Linear`.

```
### Defining a model using subclassing ###

class LinearWithSigmoidActivation(nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(LinearWithSigmoidActivation, self).__init__()
```

Deep Learning Lab Manual (CS 405)

```
"""TODO: define a model with a single Linear layer and sigmoid activation."""
self.linear = "TODO: linear layer"
self.activation = "TODO: sigmoid activation"

def forward(self, inputs):
    linear_output = self.linear(inputs)
    output = self.activation(linear_output)
    return output
```

Let's test out our new model, using an example input, setting `n_input_nodes=2` and `n_output_nodes=3` as before.

```
n_input_nodes = 2
n_output_nodes = 3
model = LinearWithSigmoidActivation(n_input_nodes, n_output_nodes)
x_input = torch.tensor([[1, 2.]])
y = model(x_input)
print(f"input shape: {x_input.shape}")
print(f"output shape: {y.shape}")
print(f"output result: {y}")
```

Importantly, `nn.Module` affords us a lot of flexibility to define custom models. For example, we can use boolean arguments in the `forward` function to specify different network behaviors, for example different behaviors during training and inference. Let's suppose under some instances we want our network to simply output the input, without any perturbation. We define a boolean argument `isidentity` to control this behavior:

```
### Custom behavior with subclassing nn.Module ###

class LinearButSometimesIdentity(nn.Module):
    def __init__(self, num_inputs, num_outputs):
        super(LinearButSometimesIdentity, self).__init__()
        self.linear = nn.Linear(num_inputs, num_outputs)

    """TODO: Implement the behavior where the network outputs the input, unchanged,
    under control of the isidentity argument."""
    def forward(self, inputs, isidentity=False):
        """ TODO """
```

Let's test this behavior:

```
# Test the IdentityModel
model = LinearButSometimesIdentity(num_inputs=2, num_outputs=3)
x_input = torch.tensor([[1, 2.]])

    """TODO: pass the input into the model and call with and without the input identity option."""
out_with_linear = # TODO
```

Deep Learning Lab Manual (CS 405)

```
out_with_identity = # TODO

print(f"input: {x_input}")
print("Network linear output: {}; network identity output: {}".format(out_with_linear, out_with_identity))
```

Now that we have learned how to define layers and models in PyTorch using both the Sequential API and subclassing `nn.Module`, we're ready to turn our attention to how to actually implement network training with backpropagation.

1.4 Automatic Differentiation in PyTorch

In PyTorch, `[`torch.autograd`](https://pytorch.org/docs/stable/autograd.html)` is used for [automatic differentiation](https://en.wikipedia.org/wiki/Automatic_differentiation), which is critical for training deep learning models with [backpropagation](https://en.wikipedia.org/wiki/Backpropagation).

We will use the PyTorch `[`.backward()`](https://pytorch.org/docs/stable/generated/torch.Tensor.backward.html)` method to trace operations for computing gradients. On a tensor, the `[`requires_grad`](https://pytorch.org/docs/stable/generated/torch.Tensor.requires_grad_.html)` attribute controls whether autograd should record operations on that tensor. When a forward pass is made through the network, PyTorch builds a computational graph dynamically; then, to compute the gradient, the ``backward()`` method is called to perform backpropagation.

Let's compute the gradient of $y = x^2$:

```
### Gradient computation ###

# y = x^2
# Example: x = 3.0
x = torch.tensor(3.0, requires_grad=True)
y = x ** 2
y.backward() # Compute the gradient

dy_dx = x.grad
print("dy_dx of y=x^2 at x=3.0 is: ", dy_dx)
assert dy_dx == 6.0
```

In training neural networks, we use differentiation and stochastic gradient descent (SGD) to optimize a loss function. Now that we have a sense of how PyTorch's autograd can be used to compute and access derivatives, we will look at an example where we use automatic differentiation and SGD to find the minimum of $L=(x-x_f)^2$. Here x_f is a variable for a desired value we are trying to optimize for; L represents a loss that we are trying to minimize. While we can clearly solve this problem analytically ($x_{\min}=x_f$), considering how we can compute this using PyTorch's autograd sets us up nicely for future labs where we use gradient descent to optimize entire neural network losses.

```
### Function minimization with autograd and gradient descent ###

# Initialize a random value for our initial x
```

Deep Learning Lab Manual (CS 405)

```
x = torch.randn(1)
print(f"Initializing x={x.item()}")

learning_rate = 1e-2 # Learning rate
history = []
x_f = 4 # Target value

# We will run gradient descent for a number of iterations. At each iteration, we compute the loss,
#  compute the derivative of the loss with respect to x, and perform the update.
for i in range(500):
    x = torch.tensor([x], requires_grad=True)

    # TODO: Compute the loss as the square of the difference between x and x_f
    loss = # TODO

    # Backpropagate through the loss to compute gradients
    loss.backward()

    # Update x with gradient descent
    x = x.item() - learning_rate * x.grad

    history.append(x.item())

# Plot the evolution of x as we optimize toward x_f!
plt.plot(history)
plt.plot([0, 500], [x_f, x_f])
plt.legend(('Predicted', 'True'))
plt.xlabel('Iteration')
plt.ylabel('x value')
plt.show()
```

Now, we have covered the fundamental concepts of PyTorch -- tensors, operations, neural networks, and automatic differentiation. Fire!!

Deep Learning Lab Manual (CS 405)

EXPERIMENT 3 – Implement Music Generation using Recurrent Neural Networks in PyTorch

Objective

To develop a Recurrent Neural Network (RNN) based on LSTM architecture in PyTorch for symbolic music generation, enabling the model to learn musical structure from ABC notation and generate original melodies.

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

Introduction

In this portion of the lab, we will explore building a Recurrent Neural Network (RNN) for music generation using PyTorch. We will train a model to learn the patterns in raw sheet music in [ABC notation](https://en.wikipedia.org/wiki/ABC_notation) and then use this model to generate new music.

2.1 Dependencies

First, let's download the course repository, install dependencies, and import the relevant packages we'll need for this lab.

We will be using [Comet ML](<https://www.comet.com/docs/v2/>) to track our model development and training runs. First, sign up for a Comet account [at this link](https://www.comet.com/signup?utm_source=mit_dl&utm_medium=partner&utm_content=github) (you can use your Google or Github account). You will need to generate a new personal API Key, which you can find either in the first 'Get Started with Comet' page, under your account settings, or by pressing the '?' in the top right corner and then 'Quickstart Guide'. Enter this API key as the global variable `COMET_API_KEY` .

First setup up an environment for deep learning using PyTorch and Comet ML for experiment tracking. It begins by installing and importing comet_ml, setting up an API key for logging experiments. Next, it imports PyTorch along with essential libraries for neural networks (torch.nn) and optimizers (torch.optim). The script also installs and imports MIT's deep learning package (mitdeeplearning), followed by necessary scientific and utility libraries (numpy, time, os, functools, etc.). It installs abcmidi and timidity for handling MIDI to WAV audio conversions, indicating that this setup might be used for audio-based deep learning tasks. Finally, the script checks whether GPU acceleration is enabled and verifies that the Comet ML API key is set correctly. If either condition is not met, it raises an error.

```
!pip install comet_ml  
import comet_ml
```

Deep Learning Lab Manual (CS 405)

```
# TODO: ENTER YOUR API KEY HERE!! instructions above
COMET_API_KEY = "kkMAdYqHFD7KuXbs0W0EUELdO"

# Import PyTorch and other relevant libraries
import torch
import torch.nn as nn
import torch.optim as optim

# Download and import the MIT Introduction to Deep Learning package
!pip install mitdeeplearning --quiet
import mitdeeplearning as mdl

# Import all remaining packages
import numpy as np
import os
import time
import functools
from IPython import display as ipythondisplay
from tqdm import tqdm
from scipy.io.wavfile import write
!apt-get install abcmidi timidity

# Check that we are using a GPU, if not switch runtimes
# using Runtime > Change Runtime Type > GPU
assert torch.cuda.is_available(), "Please enable GPU from runtime settings"
assert COMET_API_KEY != "", "kkMAdYqHFD7KuXbs0W0EUELdO"
```

Now download a dataset of songs in ABC notation using mitdeeplearning and loads it into memory. The dataset contains musical compositions in symbolic form, which can be converted into playable audio. It first retrieves the dataset using `mdl.lab1.load_training_data()` and stores it in the `songs` variable. Then, an example song (the 7th song in the dataset, `songs[6]`) is selected and printed to inspect its ABC notation structure. Finally, the script converts the ABC notation into an audio file using `mdl.lab1.play_song(example_song)`, allowing the user to listen to the selected song.

```
# Download the dataset
songs = mdl.lab1.load_training_data()

# Print one of the songs to inspect it in greater detail!
example_song = songs[6]
print("\nExample song: ")
print(example_song)
# Convert the ABC notation to audio file and listen to it
mdl.lab1.play_song(example_song)
```

Deep Learning Lab Manual (CS 405)

Setting up and training an LSTM (Long Short-Term Memory) model for text generation using PyTorch. It begins by importing the necessary libraries including torch for deep learning, comet_ml for experiment tracking, mitdeeplearning for handling the dataset, and numpy for numerical operations. It defines an LSTM-based model (LSTMModel) with an embedding layer, an LSTM layer, and a fully connected output layer, designed to process sequences of text characters. The dataset is loaded in ABC notation format, and the vocabulary is extracted to create character-to-index mappings. The model is then instantiated with hyperparameters such as batch size, sequence length, learning rate, and embedding dimensions. A function is defined to vectorize text into numerical form, and another function is used to create batches for training. A loss function (cross-entropy) is defined, and a training step function updates the model weights using the Adam optimizer. Comet ML is initialized to track training performance, and a training loop iterates over multiple iterations, logging loss values and saving model checkpoints periodically. Finally, the trained model is saved, and the experiment is closed.

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np
import os
from tqdm import tqdm
import mitdeeplearning as mdl
import comet_ml

# Define LSTM Model
class LSTMModel(nn.Module):
    def __init__(self, vocab_size, embedding_dim, hidden_size, num_layers=1):
        super(LSTMModel, self).__init__()
        self.hidden_size = hidden_size
        self.num_layers = num_layers

        print(f"Initializing LSTMModel with vocab_size={vocab_size}")

        self.embedding = nn.Embedding(num_embeddings=vocab_size, embedding_dim=embedding_dim)
        self.lstm = nn.LSTM(input_size=embedding_dim, hidden_size=hidden_size, num_layers=num_layers, batch_first=True)
        self.fc = nn.Linear(hidden_size, vocab_size)

    def init_hidden(self, batch_size, device):
        return (torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device),
                torch.zeros(self.num_layers, batch_size, self.hidden_size).to(device))

    def forward(self, x, state=None, return_state=False):
        if not isinstance(x, torch.Tensor):
            raise TypeError(f"Expected Tensor, got {type(x)}")
        if x.dtype != torch.long:
            raise TypeError(f"Expected torch.long, got {x.dtype}")
```

Deep Learning Lab Manual (CS 405)

```
x = self.embedding(x)
if state is None:
    state = self.init_hidden(x.size(0), x.device)
out, state = self.lstm(x, state)
out = self.fc(out)

return out if not return_state else (out, state)

# Load training data
songs = mdl.lab1.load_training_data()
songs_joined = "\n\n".join(songs)

# Extract vocabulary
vocab = sorted(set(songs_joined))
char2idx = {char: idx for idx, char in enumerate(vocab)}
idx2char = np.array(vocab)

print("Vocabulary Size:", len(vocab))
print("Sample character mapping:", list(char2idx.items())[:10])

# Hyperparameters
params = {
    "num_training_iterations": 3000,
    "batch_size": 8,
    "seq_length": 100,
    "learning_rate": 5e-3,
    "embedding_dim": 256,
    "hidden_size": 1024,
}

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# Instantiate model
vocab_size = len(vocab)
model = LSTMModel(vocab_size, params['embedding_dim'], params['hidden_size'], num_layers=2).to(device)

# Optimizer and Loss
optimizer = optim.Adam(model.parameters(), lr=params['learning_rate'])
cross_entropy = nn.CrossEntropyLoss()

# Vectorization

def vectorize_string(string):
    return np.array([char2idx[char] for char in string], dtype=np.int32)

vectorized_songs = vectorize_string(songs_joined)
```

Deep Learning Lab Manual (CS 405)

```
# Create Batches

def get_batch(vectorized_songs, seq_length, batch_size):
    n = vectorized_songs.shape[0] - 1
    idx = np.random.choice(n - seq_length, batch_size)
    input_batch = [vectorized_songs[i : i + seq_length] for i in idx]
    output_batch = [vectorized_songs[i + 1 : i + seq_length + 1] for i in idx]
    return torch.tensor(input_batch, dtype=torch.long), torch.tensor(output_batch, dtype=torch.long)

# Compute Loss

def compute_loss(labels, logits):
    batched_labels = labels.view(-1)
    batched_logits = logits.view(-1, logits.shape[-1])
    return cross_entropy(batched_logits, batched_labels)

# Training Step

def train_step(x, y):
    model.train()
    optimizer.zero_grad()
    x, y = x.to(device), y.to(device)
    y_hat, _ = model(x, return_state=True)
    loss = compute_loss(y, y_hat)
    loss.backward()
    optimizer.step()
    return loss.item()

# Initialize Comet Experiment

def create_experiment():
    global experiment
    try:
        experiment = comet_ml.Experiment(api_key="YOUR_COMET_API_KEY", project_name="LSTM_Text_Generation")
        for param, value in params.items():
            experiment.log_parameter(param, value)
    except Exception as e:
        print(f"Comet ML Error: {e}")
        experiment = None
    return experiment

experiment = create_experiment()

# Training Loop
history = []
plotter = mdl.util.PeriodicPlotter(sec=2, xlabel='Iterations', ylabel='Loss')
```

Deep Learning Lab Manual (CS 405)

```
if hasattr(tqdm, '_instances'):
    tqdm._instances.clear()

checkpoint_dir = './training_checkpoints'
checkpoint_prefix = os.path.join(checkpoint_dir, "my_ckpt")
os.makedirs(checkpoint_dir, exist_ok=True)

for iter in tqdm(range(params["num_training_iterations"])):
    x_batch, y_batch = get_batch(vectorized_songs, params["seq_length"], params["batch_size"])
    x_batch, y_batch = x_batch.to(device), y_batch.to(device)
    loss = train_step(x_batch, y_batch)

    if experiment:
        experiment.log_metric("loss", loss, step=iter)

    history.append(loss)
    plotter.plot(history)

    if iter % 100 == 0:
        torch.save(model.state_dict(), checkpoint_prefix)

    torch.save(model.state_dict(), checkpoint_prefix)

if experiment:
    experiment.flush()
    experiment.end()

# Model Summary
print(model)
```

Now start generating new music sequences using a trained LSTM model and converts them into playable audio files. The script first defines a function (generate_text) that takes a starting character and generates a sequence of text based on the trained LSTM model. It processes one character at a time, predicting the next character using softmax probabilities and multinomial sampling. The script then loads the trained model from the checkpoint directory (../training_checkpoints/my_ckpt) and verifies if a saved model exists before proceeding. Once the model is loaded, it generates a song snippet using the generate_text function, and the extracted musical sequences are converted into playable audio using the mdl.lab1.play_song() function. The generated audio is stored as WAV files using scipy.io.wavfile.write(), ensuring that the generated output can be saved and replayed.

```
# Generate Text
def generate_text(model, start_string, generation_length=1000):
    model.eval()
```

Deep Learning Lab Manual (CS 405)

```
input_idx = torch.tensor([char2idx[char] for char in start_string], dtype=torch.long).unsqueeze(0).to(device)
state = model.init_hidden(1, device)
text_generated = []

for _ in tqdm(range(generation_length)):
    predictions, state = model(input_idx, state, return_state=True)
    predictions = predictions.squeeze(0).detach()
    probabilities = torch.nn.functional.softmax(predictions[-1], dim=0)
    input_idx = torch.multinomial(probabilities, num_samples=1).unsqueeze(0)
    text_generated.append(idx2char[input_idx.item()])

return start_string + ''.join(text_generated)

# Instantiate and Load Model
params = {
    "embedding_dim": 256,
    "hidden_size": 1024,
    "num_layers": 2
}

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = LSTMModel(len(vocab), params['embedding_dim'], params['hidden_size'],
                  num_layers=params['num_layers']).to(device)
checkpoint_path = './training_checkpoints/my_ckpt'
if os.path.exists(checkpoint_path):
    model.load_state_dict(torch.load(checkpoint_path))

# Generate and Play Song
generated_text = generate_text(model, start_string="X", generation_length=1000)
generated_songs = mdl.lab1.extract_song_snippet(generated_text)

for i, song in enumerate(generated_songs):
    waveform = mdl.lab1.play_song(song)
    if waveform:
        print(f"Generated song {i}")
        numeric_data = np.frombuffer(waveform.data, dtype=np.int16)
        wav_file_path = f"output_{i}.wav"
        from scipy.io.wavfile import write
        write(wav_file_path, 88200, numeric_data)
```

Deep Learning Lab Manual (CS 405)

EXPERIMENT 4 – Train and Implement Handwritten Digit Classification using Convolutional Neural Networks (CNNs) on the MNIST Dataset

Objective:

To implement and train a convolutional neural network in PyTorch for classifying handwritten digits using the MNIST dataset, while integrating performance tracking through Comet ML.

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

In the first portion of this lab, we will build and train a convolutional neural network (CNN) for classification of handwritten digits from the famous [MNIST](<http://yann.lecun.com/exdb/mnist/>) dataset. The MNIST dataset consists of 60,000 training images and 10,000 test images. Our classes are the digits 0-9.

First, let's download the course repository, install dependencies, and import the relevant packages we'll need for this lab.

```
# Import PyTorch and other relevant libraries
import torch
import torch.nn as nn
import torch.optim as optim
import torchvision
import torchvision.datasets as datasets
import torchvision.transforms as transforms
from torch.utils.data import DataLoader
from torchsummary import summary

# MIT introduction to deep learning package
!pip install mitdeeplearning --quiet
import mitdeeplearning as mdl

# other packages
import matplotlib.pyplot as plt
import numpy as np
import random
from tqdm import tqdm
```

We'll also install Comet. If you followed the instructions from Lab 1, you should have your Comet account set up. Enter your API key below.

```
!pip install comet_ml > /dev/null 2>&1
import comet_ml
# TODO: ENTER YOUR API KEY HERE!!
COMET_API_KEY = "sKyAvdvZc2XXV1RABggVJl2JQ"
```

Deep Learning Lab Manual (CS 405)

```
# Check that we are using a GPU, if not switch runtimes
# using Runtime > Change Runtime Type > GPU
assert torch.cuda.is_available(), "Please enable GPU from runtime settings"
assert COMET_API_KEY != "", "sKyAvdvZc2XXV1RABggVJl2JQ"

# Set GPU for computation
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
# start a first comet experiment for the first part of the lab
comet_ml.init(project_name="6S191_lab2_part1_NN")
comet_model_1 = comet_ml.Experiment()
```

1.1 MNIST dataset

Let's download and load the dataset and display a few random samples from it:

```
# Download and transform the MNIST dataset
transform = transforms.Compose([
    # Convert images to PyTorch tensors which also scales data from [0,255] to [0,1]
    transforms.ToTensor()
])

# Download training and test datasets
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)
```

The MNIST dataset object in PyTorch is not a simple tensor or array. It's an iterable dataset that loads samples (image-label pairs) one at a time or in batches. In a later section of this lab, we will define a handy DataLoader to process the data in batches.

```
image, label = train_dataset[0]
print(image.size()) # For a tensor: torch.Size([1, 28, 28])
print(label) # For a label: integer (e.g., 5)
```

Our training set is made up of 28x28 grayscale images of handwritten digits.

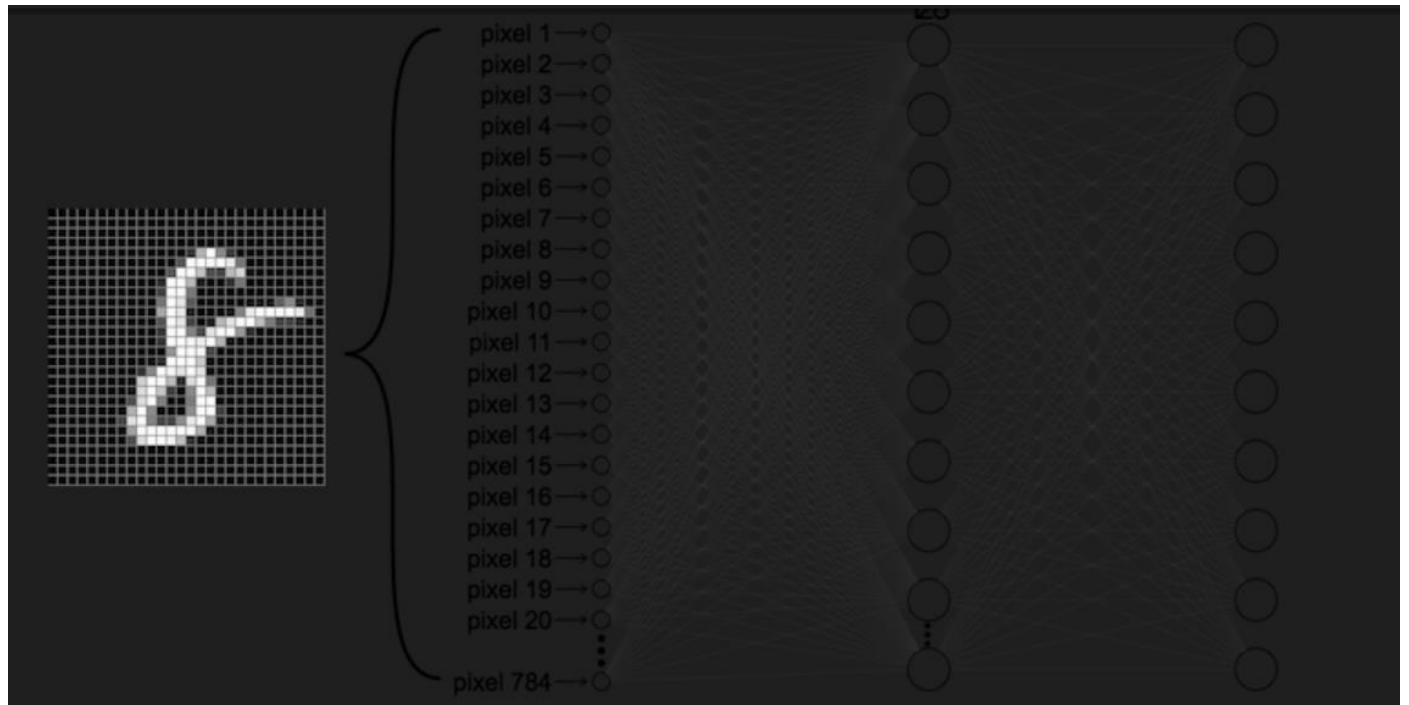
Let's visualize what some of these images and their corresponding training labels look like.

```
plt.figure(figsize=(10,10))
random_inds = np.random.choice(60000,36)
for i in range(36):
    plt.subplot(6, 6, i + 1)
    plt.xticks([])
    plt.yticks([])
    plt.grid(False)
    image_ind = random_inds[i]
    image, label = train_dataset[image_ind]
    plt.imshow(image.squeeze(), cmap=plt.cm.binary)
    plt.xlabel(label)
comet_model_1.log_figure(figure=plt)
```

Deep Learning Lab Manual (CS 405)

1.2 Neural Network for Handwritten Digit Classification

We'll first build a simple neural network consisting of two fully connected layers and apply this to the digit classification task. Our network will ultimately output a probability distribution over the 10 digit classes (0-9). This first architecture we will be building is depicted below:



Fully connected neural network architecture

To define the architecture of this first fully connected neural network, we'll once again use the the `torch.nn` modules, defining the model using
[`nn.Sequential`](<https://pytorch.org/docs/stable/generated/torch.nn.Sequential.html>). Note how we first use a
[`nn.Flatten`](https://www.tensorflow.org/api_docs/python/tf/keras/layers/Flatten) layer, which flattens the input so that it can be fed into the model.

In this next block, you'll define the fully connected layers of this simple network.

```
def build_fc_model():
    fc_model = nn.Sequential(
        # First define a Flatten layer
        nn.Flatten(),
        # Define the activation function for the first fully connected (Dense/Linear) layer.
        nn.Linear(28 * 28, 128),
        nn.ReLU(), # Added ReLU activation function

        # Define the second Linear layer to output the classification probabilities
        nn.Linear(128, 10) # Added a linear layer for classification with 10 output classes
    )
```

Deep Learning Lab Manual (CS 405)

```
return fc_model  
  
fc_model_sequential = build_fc_model()
```

As we progress through this next portion, you may find that you'll want to make changes to the architecture defined above.

Note that in order to update the model later on, you'll need to re-run the above cell to re-initialize the model.

Let's take a step back and think about the network we've just created. The first layer in this network, `nn.Flatten`, transforms the format of the images from a 2d-array (28 x 28 pixels), to a 1d-array of $28 * 28 = 784$ pixels. You can think of this layer as unstacking rows of pixels in the image and lining them up. There are no learned parameters in this layer; it only reformats the data.

After the pixels are flattened, the network consists of a sequence of two `nn.Linear` layers. These are fully-connected neural layers. The first `nn.Linear` layer has 128 nodes (or neurons). The second (and last) layer (which you've defined!) should return an array of probability scores that sum to 1. Each node contains a score that indicates the probability that the current image belongs to one of the handwritten digit classes.

That defines our fully connected model!

Embracing subclassing in PyTorch

Recall that in Lab 1, we explored creating more flexible models by subclassing [`nn.Module`]<https://pytorch.org/docs/stable/generated/torch.nn.Module.html>). This technique of defining models is more commonly used in PyTorch. We will practice using this approach of subclassing to define our models for the rest of the lab.

```
# Define the fully connected model  
class FullyConnectedModel(nn.Module):  
    def __init__(self):  
        super(FullyConnectedModel, self).__init__()  
        self.flatten = nn.Flatten()  
        self.fc1 = nn.Linear(28 * 28, 128)  
        # "TODO: Define the activation function for the first fully connected layer"  
        self.relu = nn.ReLU()  
  
        # "TODO: Define the second Linear layer to output the classification probabilities"  
        self.fc2 = nn.Linear(128, 10)  
  
    def forward(self, x):  
        x = self.flatten(x)  
        x = self.fc1(x)  
  
        # "TODO: Implement the rest of forward pass of the model using the layers you have defined above"  
        x = self.relu(x)  
        x = self.fc2(x)  
  
    return x
```

Deep Learning Lab Manual (CS 405)

```
fc_model = FullyConnectedModel().to(device) # send the model to GPU
```

Model Metrics and Training Parameters

Before training the model, we need to define components that govern its performance and guide its learning process. These include the loss function, optimizer, and evaluation metrics:

- Loss function— This defines how we measure how accurate the model is during training. As was covered in lecture, during training we want to minimize this function, which will "steer" the model in the right direction.
- Optimizer — This defines how the model is updated based on the data it sees and its loss function.
- Metrics — Here we can define metrics that we want to use to monitor the training and testing steps. In this example, we'll define and take a look at the *accuracy*, the fraction of the images that are correctly classified.

We'll start out by using a stochastic gradient descent (SGD) optimizer initialized with a learning rate of 0.1. Since we are performing a categorical classification task, we'll want to use the [cross entropy loss](<https://pytorch.org/docs/stable/generated/torch.nn.CrossEntropyLoss.html>).

You'll want to experiment with both the choice of optimizer and learning rate and evaluate how these affect the accuracy of the trained model.

```
'''TODO: Experiment with different optimizers and learning rates. How do these affect  
the accuracy of the trained model? Which optimizers and/or learning rates yield  
the best performance?'''
```

```
# Define loss function and optimizer  
loss_function = nn.CrossEntropyLoss()  
optimizer = optim.SGD(fc_model.parameters(), lr=0.1)
```

Train the model

We're now ready to train our model, which will involve feeding the training data (`train_dataset`) into the model, and then asking it to learn the associations between images and labels. We'll also need to define the batch size and the number of epochs, or iterations over the MNIST dataset, to use during training. This dataset consists of a (image, label) tuples that we will iteratively access in batches.

In Lab 1, we saw how we can use the `backward()`(<https://pytorch.org/docs/stable/generated/torch.Tensor.backward.html>) method to optimize losses and train models with stochastic gradient descent. In this section, we will define a function to train the model using `backward()` and `optimizer.step()` to automatically update our model parameters (weights and biases) as we saw in Lab 1.

Recall, we mentioned in Section 1.1 that the MNIST dataset can be accessed iteratively in batches. Here, we will define a PyTorch [`DataLoader`][\(https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader\)](https://pytorch.org/docs/stable/data.html#torch.utils.data.DataLoader) that will enable us to do that.

```
# Create DataLoaders for batch processing  
BATCH_SIZE = 64  
trainset_loader = DataLoader(train_dataset, batch_size=BATCH_SIZE, shuffle=True)
```

Deep Learning Lab Manual (CS 405)

```
testset_loader = DataLoader(test_dataset, batch_size=BATCH_SIZE, shuffle=False)
```

```
def train(model, dataloader, criterion, optimizer, epochs):
    model.train() # Set the model to training mode
    for epoch in range(epochs):
        total_loss = 0
        correct_pred = 0
        total_pred = 0

        for images, labels in trainset_loader:
            # Move tensors to GPU so compatible with model
            images, labels = images.to(device), labels.to(device)

            # Forward pass
            outputs = fc_model(images)

            # Clear gradients before performing backward pass
            optimizer.zero_grad()
            # Calculate loss based on model predictions
            loss = loss_function(outputs, labels)
            # Backpropagate and update model parameters
            loss.backward()
            optimizer.step()

            # multiply loss by total nos. of samples in batch
            total_loss += loss.item()*images.size(0)

            # Calculate accuracy
            predicted = torch.argmax(outputs, dim=1) # Get predicted class
            correct_pred += (predicted == labels).sum().item() # Count correct predictions
            total_pred += labels.size(0) # Count total predictions

        # Compute metrics
        total_epoch_loss = total_loss / total_pred
        epoch_accuracy = correct_pred / total_pred
        print(f"Epoch {epoch + 1}, Loss: {total_epoch_loss}, Accuracy: {epoch_accuracy:.4f}")

# TODO: Train the model by calling the function appropriately
EPOCHS = 5
# Call the train function with the required arguments
train(fc_model, trainset_loader, loss_function, optimizer, EPOCHS) # Passing the required arguments to the train function

comet_model_1.end()
```

As the model trains, the loss and accuracy metrics are displayed. With five epochs and a learning rate of 0.01, this fully connected model should achieve an accuracy of approximatley 0.97 (or 97%) on the training data.

Deep Learning Lab Manual (CS 405)

Evaluate accuracy on the test dataset

Now that we've trained the model, we can ask it to make predictions about a test set that it hasn't seen before. In this example, iterating over the `testset_loader` allows us to access our test images and test labels. And to evaluate accuracy, we can check to see if the model's predictions match the labels from this loader.

Since we have now trained the mode, we will use the eval state of the model on the test dataset.

```
def evaluate(model, dataloader, loss_function):
    # Evaluate model performance on the test dataset
    model.eval()
    test_loss = 0
    correct_pred = 0
    total_pred = 0
    # Disable gradient calculations when in inference mode
    with torch.no_grad():
        for images, labels in testset_loader:
            # TODO: ensure evalaution happens on the GPU
            images, labels = images.to(device), labels.to(device)

            # TODO: feed the images into the model and obtain the predictions (forward pass)
            outputs = model(images)

            loss = loss_function(outputs, labels)

            # TODO: Calculate test loss
            test_loss += loss.item() * images.size(0)

            # TODO: make a prediction and determine whether it is correct!!!
            # TODO: identify the digit with the highest probability prediction for the images in the test dataset.
            predicted = torch.argmax(outputs, dim=1)

            # TODO: tally the number of correct predictions
            correct_pred += (predicted == labels).sum().item()

            # TODO: tally the total number of predictions
            total_pred += labels.size(0)

    # Compute average loss and accuracy
    test_loss /= total_pred
    test_acc = correct_pred / total_pred
    return test_loss, test_acc

# TODO: call the evaluate function to evaluate the trained model!!
test_loss, test_acc = evaluate(fc_model, testset_loader, loss_function)

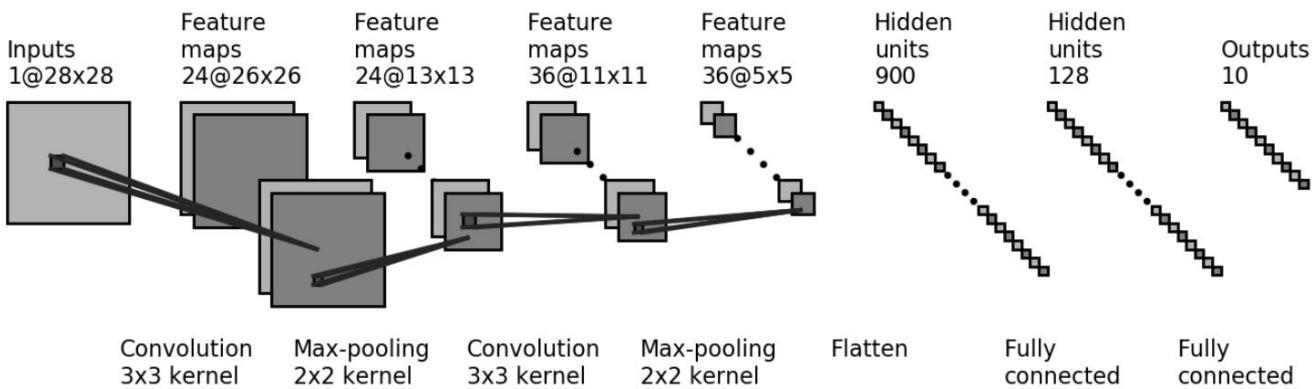
print('Test accuracy:', test_acc)
```

Deep Learning Lab Manual (CS 405)

You may observe that the accuracy on the test dataset is a little lower than the accuracy on the training dataset. This gap between training accuracy and test accuracy is an example of **overfitting**, when a machine learning model performs worse on new data than on its training data.

What is the highest accuracy you can achieve with this first fully connected model? Since the handwritten digit classification task is pretty straightforward, you may be wondering how we can do better...

As we saw in lecture, convolutional neural networks (CNNs) are particularly well-suited for a variety of tasks in computer vision, and have achieved near-perfect accuracies on the MNIST dataset. We will now build a CNN composed of two convolutional layers and pooling layers, followed by two fully connected layers, and ultimately output a probability distribution over the 10 digit classes (0-9). The CNN we will be building is depicted below:



Define the CNN model

We'll use the same training and test datasets as before, and proceed similarly as our fully connected network to define and train our new CNN model. To do this we will explore two layers we have not encountered before: you can use `[`nn.Conv2d`](https://pytorch.org/docs/stable/generated/torch.nn.Conv2d.html)` to define convolutional layers and

`[`nn.MaxPool2D`](https://pytorch.org/docs/stable/generated/torch.nn.MaxPool2d.html)` to define the pooling layers. Use the parameters shown in the network architecture above to define these layers and build the CNN model. You can decide to use ``nn.Sequential`` or to subclass ``nn.Module`` based on your preference.

```
### Basic CNN in PyTorch ###
```

```
import torch.nn as nn

class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        # TODO: Define the first convolutional layer
        self.conv1 = nn.Conv2d(in_channels=1, out_channels=16, kernel_size=3, padding=1)

        # TODO: Define the first max pooling layer
        self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)
```

Deep Learning Lab Manual (CS 405)

```
# TODO: Define the second convolutional layer
self.conv2 = nn.Conv2d(in_channels=16, out_channels=36, kernel_size=3, padding=1)

# TODO: Define the second max pooling layer
self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)

self.flatten = nn.Flatten()
# Corrected calculation for input size: 36 channels, 7x7 output from pooling
self.fc1 = nn.Linear(36 * 7 * 7, 128)
self.relu = nn.ReLU()

# TODO: Define the Linear layer that outputs the classification
# logits over class labels. Remember that CrossEntropyLoss operates over logits.
self.fc2 = nn.Linear(128, 10)

def forward(self, x):
    # First convolutional and pooling layers
    x = self.conv1(x)
    x = self.relu(x)
    x = self.pool1(x)

    # """TODO: Implement the rest of forward pass of the model using the layers you have defined above"""
    #   "hint: this will involve another set of convolutional/pooling layers and then the linear layers"
    x = self.conv2(x)
    x = self.relu(x)
    x = self.pool2(x)
    x = self.flatten(x)
    x = self.fc1(x)
    x = self.relu(x)
    x = self.fc2(x)

    return x

# Instantiate the model
cnn_model = CNN().to(device)
# Initialize the model by passing some data through
image, label = train_dataset[0]
image = image.to(device).unsqueeze(0) # Add batch dimension → Shape: (1, 1, 28, 28)
output = cnn_model(image)
# Print the model summary
print(cnn_model)
```

Train and test the CNN model

Earlier in the lab, we defined a `train` function. The body of the function is quite useful because it allows us to

Deep Learning Lab Manual (CS 405)

have control over the training model, and to record differentiation operations during training by computing the gradients using `loss.backward()`. You may recall seeing this in Lab 1 Part 1.

We'll use this same framework to train our `cnn_model` using stochastic gradient descent. You are free to implement the following parts with or without the train and evaluate functions we defined above. What is most important is understanding how to manipulate the bodies of those functions to train and test models.

As we've done above, we can define the loss function, optimizer, and calculate the accuracy of the model. Define an optimizer and learning rate of choice. Feel free to modify as you see fit to optimize your model's performance.

```
# Rebuild the CNN model
cnn_model = CNN().to(device)

# Define hyperparams
batch_size = 64
epochs = 7
optimizer = optim.SGD(cnn_model.parameters(), lr=1e-2)

# Rebuild the CNN model
cnn_model = CNN().to(device)

# Define hyperparams
batch_size = 64
epochs = 7
optimizer = optim.SGD(cnn_model.parameters(), lr=1e-2)

# TODO: instantiate the cross entropy loss function
loss_function = None # TODO

# Redefine trainloader with new batch size parameter (tweak as see fit if optimizing)
trainset_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
testset_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
trainset_loader = DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
testset_loader = DataLoader(test_dataset, batch_size=batch_size, shuffle=False)
```

```
loss_history = mdl.util.LossHistory(smoothing_factor=0.95) # to record the evolution of the loss
plotter = mdl.util.PeriodicPlotter(sec=2, xlabel='Iterations', ylabel='Loss', scale='semilogy')

# Initialize new comet experiment
comet_ml.init(project_name="6.s191lab2_part1_CNN")
comet_model_2 = comet_ml.Experiment()

if hasattr(tqdm, '_instances'): tqdm._instances.clear() # clear if it exists

# Training loop!
cnn_model.train()

for epoch in range(epochs):
```

Deep Learning Lab Manual (CS 405)

```
total_loss = 0
correct_pred = 0
total_pred = 0

# First grab a batch of training data which our data loader returns as a tensor
for idx, (images, labels) in enumerate(tqdm(trainset_loader)):
    images, labels = images.to(device), labels.to(device)

    # Forward pass
    # TODO: feed the images into the model and obtain the predictions
    logits = cnn_model(images) # Passing images to the cnn_model

    # TODO: compute the categorical cross entropy loss using the predicted logits
    # Assuming loss_function is defined as nn.CrossEntropyLoss()
    loss_function = nn.CrossEntropyLoss() # Instantiating the loss function
    loss = loss_function(logits, labels) # Calculating the loss

    # Get the loss and log it to comet and the loss_history record
    loss_value = loss.item()
    comet_model_2.log_metric("loss", loss_value, step=idx)
    loss_history.append(loss_value) # append the loss to the loss_history record
    plotter.plot(loss_history.get())

    # Backpropagation/backward pass
    """TODO: Compute gradients for all model parameters and propagate backwads
       to update model parameters. remember to reset your optimizer!"""
    # TODO: reset optimizer
    optimizer.zero_grad() #Example assuming you have an optimizer named optimizer
    # TODO: compute gradients
    loss.backward()
    # TODO: update model parameters
    optimizer.step() #Example assuming you have an optimizer named optimizer

    # Get the prediction and tally metrics
    predicted = torch.argmax(logits, dim=1)
    correct_pred += (predicted == labels).sum().item()
    total_pred += labels.size(0)
    total_loss += loss.item()*labels.size(0)

    # Compute metrics
    total_epoch_loss = total_loss / total_pred
    epoch_accuracy = correct_pred / total_pred
    print(f"Epoch {epoch + 1}, Loss: {total_epoch_loss}, Accuracy: {epoch_accuracy:.4f}")

comet_model_2.log_figure(figure=plt) # Assuming 'plt' is defined and holds a figure
```

Evaluate the CNN Model

Deep Learning Lab Manual (CS 405)

Now that we've trained the model, let's evaluate it on the test dataset.

```
'''TODO: Evaluate the CNN model!'''
# Pass the testset_loader and loss_function to the evaluate function
test_loss, test_acc = evaluate(cnn_model, testset_loader, loss_function)

print('Test accuracy:', test_acc)
```

What is the highest accuracy you're able to achieve using the CNN model, and how does the accuracy of the CNN model compare to the accuracy of the simple fully connected network? What optimizers and learning rates seem to be optimal for training the CNN model?

Feel free to click the Comet links to investigate the training/accuracy curves for your model.

Make predictions with the CNN model

With the model trained, we can use it to make predictions about some images.

```
test_image, test_label = test_dataset[0]
test_image = test_image.to(device).unsqueeze(0)

# put the model in evaluation (inference) mode
cnn_model.eval()
predictions_test_image = cnn_model(test_image)
```

With this function call, the model has predicted the label of the first image in the testing set. Let's take a look at the prediction:

```
print(predictions_test_image)
```

As you can see, a prediction is an array of 10 numbers. Recall that the output of our model is a distribution over the 10 digit classes. Thus, these numbers describe the model's predicted likelihood that the image corresponds to each of the 10 different digits.

```
'''TODO: identify the digit with the highest likelihood prediction for the first
       image in the test dataset.'''
predictions_value = predictions_test_image.cpu().detach().numpy() #.cpu() to copy tensor to memory first
prediction = np.argmax(predictions_value)
print(prediction)
```

Let's look at the digit that has the highest likelihood for the first image in the test dataset:

So, the model is most confident that this image is a "???". We can check the test label (remember, this is the true identity of the digit) to see if this prediction is correct:

```
print("Label of this digit is:", test_label)
plt.imshow(test_image[0,0,:,:].cpu(), cmap=plt.cm.binary)
comet_model_2.log_figure.figure=plt
```

It is! Let's visualize the classification results on the MNIST dataset. We will plot images from the test dataset along with their predicted label, as well as a histogram that provides the prediction probabilities for each of the digits.

Deep Learning Lab Manual (CS 405)

Recall that in PyTorch the MNIST dataset is typically accessed using a DataLoader to iterate through the test set in smaller, manageable batches. By appending the predictions, test labels, and test images from each batch, we will first gradually accumulate all the data needed for visualization into singular variables to observe our model's predictions.

```
# Process test set in batches
with torch.no_grad():
    for images, labels in testset_loader:
        # Move images and labels to the GPU
        images = images.to(device)
        labels = labels.to(device)
        outputs = cnn_model(images)

        # Apply softmax to get probabilities from the predicted logits
        probabilities = torch.nn.functional.softmax(outputs, dim=1)

        # Get predicted classes
        predicted = torch.argmax(probabilities, dim=1)

        all_predictions.append(probabilities)
        all_labels.append(labels)
        all_images.append(images)

# ... (rest of the code)
```

```
#@title Change the slider to look at the model's predictions! { run: "auto" }

image_index = 93 #@param {type:"slider", min:0, max:100, step:1}

# Initialize empty lists to store all predictions, labels, and images
all_predictions = []
all_labels = []
all_images = []

# Process test set in batches
with torch.no_grad():
    for images, labels in testset_loader:
        # Move images and labels to the GPU
        images = images.to(device)
        labels = labels.to(device)
        outputs = cnn_model(images)

        # Apply softmax to get probabilities from the predicted logits
        probabilities = torch.nn.functional.softmax(outputs, dim=1)

        # Get predicted classes
        predicted = torch.argmax(probabilities, dim=1)
```

Deep Learning Lab Manual (CS 405)

```
all_predictions.append(probabilities)
all_labels.append(labels)
all_images.append(images)

# Concatenate the results from all batches
predictions = torch.cat(all_predictions, dim=0)
test_labels = torch.cat(all_labels, dim=0)
test_images = torch.cat(all_images, dim=0)

plt.subplot(1,2,1)
mdl.lab2.plot_image_prediction(image_index, predictions.cpu().numpy(), test_labels.cpu().numpy(),
test_images.cpu().numpy()) # Pass numpy arrays
plt.subplot(1,2,2)
mdl.lab2.plot_value_prediction(image_index, predictions.cpu().numpy(), test_labels.cpu().numpy()) # Pass numpy arrays
comet_model_2.log_figure.figure=plt
```

We can also plot several images along with their predictions, where correct prediction labels are blue and incorrect prediction labels are grey. The number gives the percent confidence (out of 100) for the predicted label. Note the model can be very confident in an incorrect prediction!

```
# Plots the first X test images, their predicted label, and the true label
# Color correct predictions in blue, incorrect predictions in red
num_rows = 5
num_cols = 4
num_images = num_rows*num_cols
plt.figure(figsize=(2*2*num_cols, 2*num_rows))
for i in range(num_images):
    plt.subplot(num_rows, 2*num_cols, 2*i+1)
    # Move tensors to CPU and convert to NumPy arrays before plotting
    mdl.lab2.plot_image_prediction(i, predictions.cpu().numpy(), test_labels.cpu().numpy(), test_images.cpu().numpy())
    plt.subplot(num_rows, 2*num_cols, 2*i+2)
    # Move tensors to CPU and convert to NumPy arrays before plotting
    mdl.lab2.plot_value_prediction(i, predictions.cpu().numpy(), test_labels.cpu().numpy())
comet_model_2.log_figure.figure=plt
comet_model_2.end()
```

1.5 Conclusion

In this part of the lab, you had the chance to play with different MNIST classifiers with different architectures (fully-connected layers only, CNN), and experiment with how different hyperparameters affect accuracy (learning rate, etc.). The next part of the lab explores another application of CNNs, facial detection, and some drawbacks of AI systems in real world applications, like issues of bias.

EXPERIMENT 5 – Implement Image Denoising and Feature Learning using Autoencoders

Objective:

To understand and implement Autoencoders (AEs) for feature extraction, denoising, and dimensionality reduction on the Fashion MNIST dataset, using Keras and TensorFlow.

To Install a C compiler in the virtual machine created using virtual box and execute Simple Programs.

Time Required: 3 hrs

Programming Language: e.g. Python

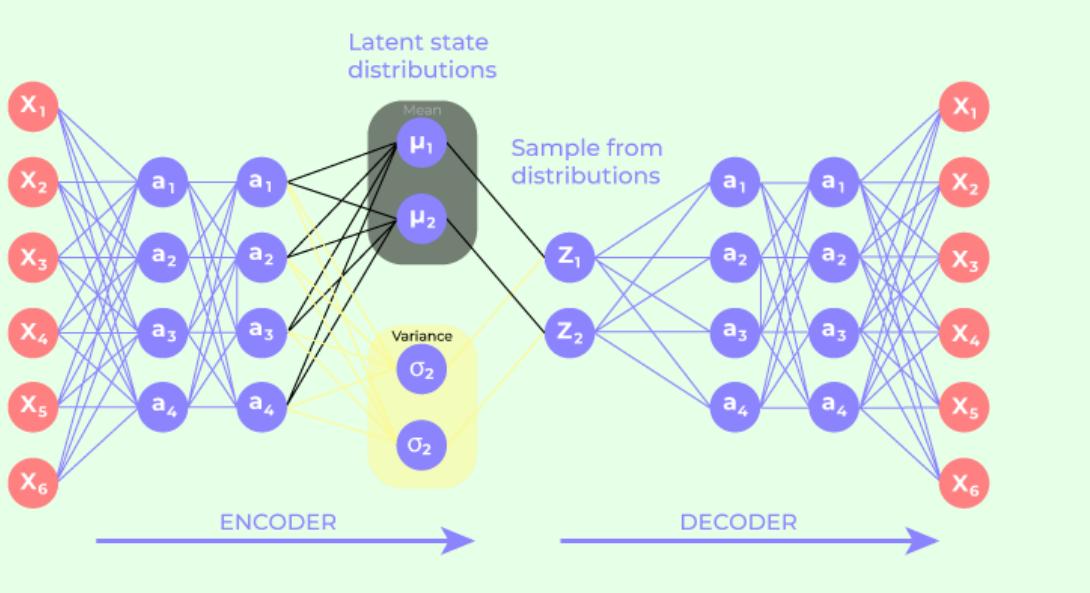
Software/Tools Required: e.g. Anaconda/ Google Colab

Instruction

An Autoencoder (AE) is a type of neural network used for unsupervised learning, mainly for dimensionality reduction, feature learning, and data denoising. It consists of two parts:

1. Encoder: Compresses input data into a lower-dimensional latent representation.
2. Decoder: Reconstructs the original input from the latent representation.

A Variational Autoencoder (VAE) extends the standard autoencoder by adding a probabilistic layer. Instead of encoding the input as a fixed point in the latent space, VAE represents it as a probability distribution (mean and variance). This helps in generating new, realistic data samples and ensures a smooth latent space.



Fashion MNIST Dataset

Fashion MNIST is a dataset containing 28×28 grayscale images of clothing items (e.g., shirts, shoes, bags). It serves as a benchmark for image classification and generative models.

Overview of the Code

The given Python code implements a Variational Autoencoder (VAE) using TensorFlow/Keras on the Fashion MNIST dataset. The process follows these steps:

1. Load Dataset: Fashion MNIST images are loaded and preprocessed (normalized).
2. Define VAE Model: The model consists of an encoder, a sampling function (for reparameterization trick), and a decoder.

Deep Learning Lab Manual (CS 405)

3. Loss Function: The VAE optimizes both reconstruction loss (how well the output matches the input) and KL divergence (ensuring a smooth latent space distribution).
4. Training: The model is trained using the Fashion MNIST dataset.
5. Image Generation: The trained VAE generates new Fashion MNIST-like images from the learned latent space.

In short, the code trains a VAE to encode images into a latent representation and then reconstruct them while also generating new clothing images from the learned distribution.

```
# !pip install imgaug
```

```
## load the libraries
import sys
import warnings
import os
import glob
warnings.filterwarnings("ignore")

import numpy as np
import pandas as pd
import cv2
from sklearn.model_selection import train_test_split

from keras.layers import *
from keras.callbacks import EarlyStopping
from keras.utils import to_categorical
from keras.models import Model, Sequential
from keras.metrics import *
from keras.optimizers import Adam, RMSprop
from scipy.stats import norm
from keras.preprocessing import image

from keras import backend as K

from imgaug import augmenters
import matplotlib.pyplot as plt
plt.gray()
```

Part 1: Data

Reading data

Download the data given at the following link: <insert_link>. Use pandas and numpy to read in the data as a matrix

```
### read dataset
train = pd.read_csv("data/fashion-mnist_train.csv")
train_x = train[list(train.columns)[1:]].values
train_x, val_x = train_test_split(train_x, test_size=0.15)
```

Deep Learning Lab Manual (CS 405)

```
## create train and validation datasets  
train_x, val_x = train_test_split(train_x, test_size=0.15)
```

```
## normalize and reshape  
train_x = train_x/255.  
val_x = val_x/255.  
  
train_x = train_x.reshape(-1, 28, 28, 1)  
val_x = val_x.reshape(-1, 28, 28, 1)
```

```
train_x.shape
```

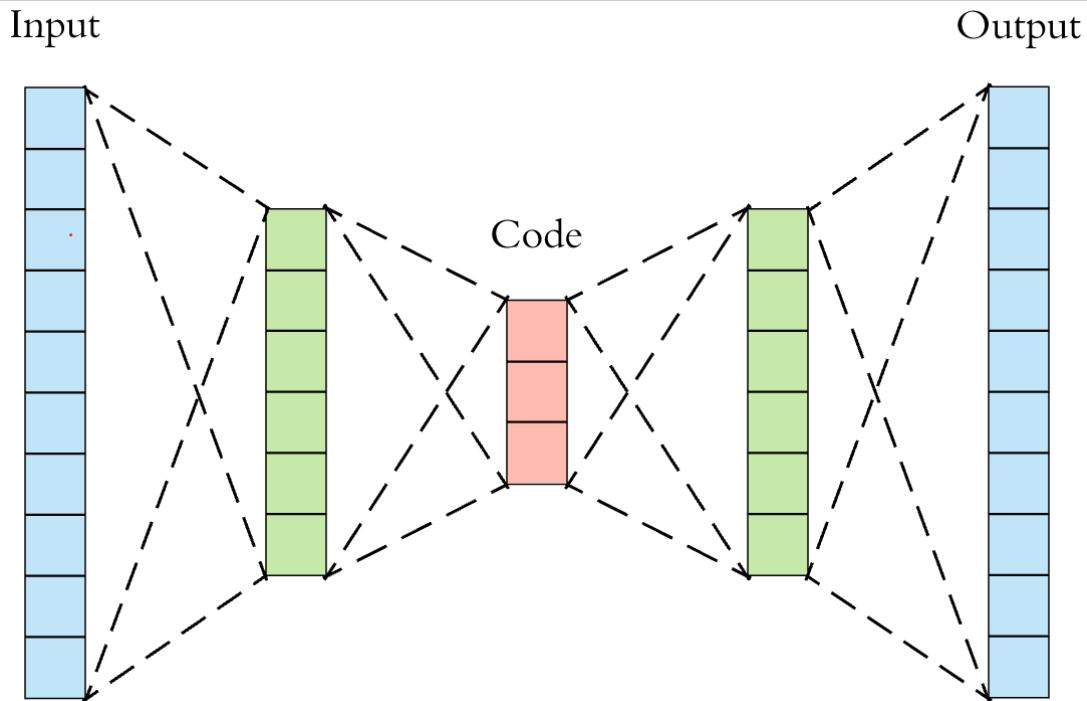
Visualizing Samples

Visualize 10 images from dataset

```
f, ax = plt.subplots(1,5)  
f.set_size_inches(80, 40)  
for i in range(5,10):  
    ax[i-5].imshow(train_x[i, :, :, 0].reshape(28, 28))
```

Part 2: Denoise Images using AEs

Understanding AEs



Add Noise to Images

Check out [imgaug docs](https://imgaug.readthedocs.io/en/latest/source/api_augmenters_arithmetic.html) for

Deep Learning Lab Manual (CS 405)

more info and other ways to add noise.

```
# Lets add sample noise - Salt and Pepper
noise = augmenters.SaltAndPepper(0.1)
seq_object = augmenters.Sequential([noise])

train_x_n = seq_object.augment_images(train_x * 255) / 255
val_x_n = seq_object.augment_images(val_x * 255) / 255
```

```
f, ax = plt.subplots(1,5)
f.set_size_inches(80, 40)
for i in range(5,10):
    ax[i-5].imshow(train_x_n[i, :, :, 0].reshape(28, 28))
```

Setup Encoder Neural Network

Try different number of hidden layers, nodes?

```
# input layer
input_layer = Input(shape=(28, 28, 1))

# encoding architecture
encoded_layer1 = Conv2D(64, (3, 3), activation='relu', padding='same')(input_layer)
encoded_layer1 = MaxPool2D((2, 2), padding='same')(encoded_layer1)
encoded_layer2 = Conv2D(32, (3, 3), activation='relu', padding='same')(encoded_layer1)
encoded_layer2 = MaxPool2D((2, 2), padding='same')(encoded_layer2)
encoded_layer3 = Conv2D(16, (3, 3), activation='relu', padding='same')(encoded_layer2)
latent_view = MaxPool2D((2, 2), padding='same')(encoded_layer3)
```

Setup Decoder Neural Network

Try different number of hidden layers, nodes?

```
# decoding architecture
decoded_layer1 = Conv2D(16, (3, 3), activation='relu', padding='same')(latent_view)
decoded_layer1 = UpSampling2D((2, 2))(decoded_layer1)
decoded_layer2 = Conv2D(32, (3, 3), activation='relu', padding='same')(decoded_layer1)
decoded_layer2 = UpSampling2D((2, 2))(decoded_layer2)
decoded_layer3 = Conv2D(64, (3, 3), activation='relu')(decoded_layer2)
decoded_layer3 = UpSampling2D((2, 2))(decoded_layer3)
output_layer = Conv2D(1, (3, 3), padding='same')(decoded_layer3)
```

Train AE

```
# compile the model
```

Deep Learning Lab Manual (CS 405)

```
model = Model(input_layer, output_layer)
model.compile(optimizer='adam', loss='mse')
```

```
model.summary()
```

```
early_stopping = EarlyStopping(monitor='val_loss', min_delta=0, patience=10, verbose=5, mode='auto')
history = model.fit(train_x_n, train_x, epochs=20, batch_size=2048, validation_data=(val_x_n, val_x),
callbacks=[early_stopping])
```

Visualize Intermediate Layers of AE

Visualize intermediate layers

```
# compile the model
model_2 = Model(input_layer, latent_view)
model_2.compile(optimizer='adam', loss='mse')
```

```
n = np.random.randint(0,len(val_x)-5)
f, ax = plt.subplots(1,5)
f.set_size_inches(80, 40)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow(val_x_n[a, :, :, 0].reshape(28, 28))
plt.show()
```

```
preds = model_2.predict(val_x_n[n:n+5])
preds.shape
```

```
f, ax = plt.subplots(16,5)
ax = ax.ravel()
f.set_size_inches(20, 40)
for j in range(16):
    for i,a in enumerate(range(n,n+5)):
        ax[j*5 + i].imshow(preds[i, :, :, j])
plt.show()
```

Visualize Samples reconstructed by AE

```
n = np.random.randint(0,len(val_x)-5)
```

```
f, ax = plt.subplots(1,5)
f.set_size_inches(80, 40)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow(val_x[a, :, :, 0].reshape(28, 28))
```

Deep Learning Lab Manual (CS 405)

```
f, ax = plt.subplots(1,5)
f.set_size_inches(80, 40)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow(val_x_n[a, :, :, 0].reshape(28, 28))
```

```
preds = model.predict(val_x_n[n:n+5])
f, ax = plt.subplots(1,5)
f.set_size_inches(80, 40)
for i,a in enumerate(range(n,n+5)):
    ax[i].imshow(preds[i].reshape(28, 28))
plt.show()
```

Exercise: Denoising noisy documents

```
TRAIN_IMAGES = glob.glob('data/train/*.png')
CLEAN_IMAGES = glob.glob('data/train_cleaned/*.png')
TEST_IMAGES = glob.glob('data/test/*.png')
```

```
plt.figure(figsize=(20,8))
img = cv2.imread('data/train/101.png', 0)
plt.imshow(img, cmap='gray')
print(img.shape)
```

```
def load_image(path):
    image_list = np.zeros((len(path), 258, 540, 1))
    for i, fig in enumerate(path):
        img = image.load_img(fig, grayscale=True, target_size=(258, 540))
        x = image.img_to_array(img).astype('float32')
        x = x / 255.0
        image_list[i] = x

    return image_list

x_train = load_image(TRAIN_IMAGES)
y_train = load_image(CLEAN_IMAGES)
x_test = load_image(TEST_IMAGES)

print(x_train.shape, x_test.shape)
```

```
#Todo: Split your dataset into train and val
```

```
#Todo: Visualize your train set
```

Deep Learning Lab Manual (CS 405)

```
input_layer = Input(shape=(258, 540, 1))

#Todo: Setup encoder
#Hint: Conv2D -> MaxPooling

#Todo: Setup decoder
#Hint: Conv2D -> Upsampling

output_layer = Conv2D(1, (3, 3), activation='sigmoid', padding='same')(decoder)

ae = Model(input_layer, output_layer)
```

```
#Todo: Compile and summarize your auto encoder
```

```
#Todo: Train your autoencoder
```

```
preds = ae.predict(x_test)
```

```
# n = 25
# preds_0 = preds[n] * 255.0
# preds_0 = preds_0.reshape(258, 540)
# x_test_0 = x_test[n] * 255.0
# x_test_0 = x_test_0.reshape(258, 540)
# plt.imshow(x_test_0, cmap='gray')
```

```
# plt.imshow(preds_0, cmap='gray')
```

Deep Learning Lab Manual (CS 405)

Experiment 6: Train and Implement Generating Novel Fashion Images using Variational Autoencoders (VAEs)

Objective:

To train and utilize a Variational Autoencoder (VAE) for generating new fashion item images by learning latent space distributions, and visualize the model's capability using CNN-based encoder-decoder architecture.

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

Introduction

Understanding VAEs

Variational Autoencoders (VAEs) are generative models in machine learning (ML) that create new data similar to the input they are trained on. Along with data generation they also perform common autoencoder tasks like **denoising**. Like all autoencoders VAEs consist of:

- **Encoder:** Learns important patterns (latent variables) from input data.
- **Decoder:** It uses those latent variables to reconstruct the input.

Unlike traditional [autoencoders](#) that encode a fixed representation VAEs learn a continuous probabilistic representation of latent space. This allows them to reconstruct input data accurately and generate new data samples that resemble the original input.

Architecture of Variational Autoencoder

VAE is a special kind of autoencoder that can generate new data instead of just compressing and reconstructing it. It has three main parts:

1. Encoder (Understanding the Input)

- The encoder takes the input data like an image or text and tries to understand its most important features.
- Instead of creating a fixed compressed version like a normal autoencoder it creates two things:
 - **Mean (μ):** A central value representing the data.
 - **Standard Deviation (σ):** It is a measure of how much the values can vary.
- These two values define a range of possibilities instead of a single number.

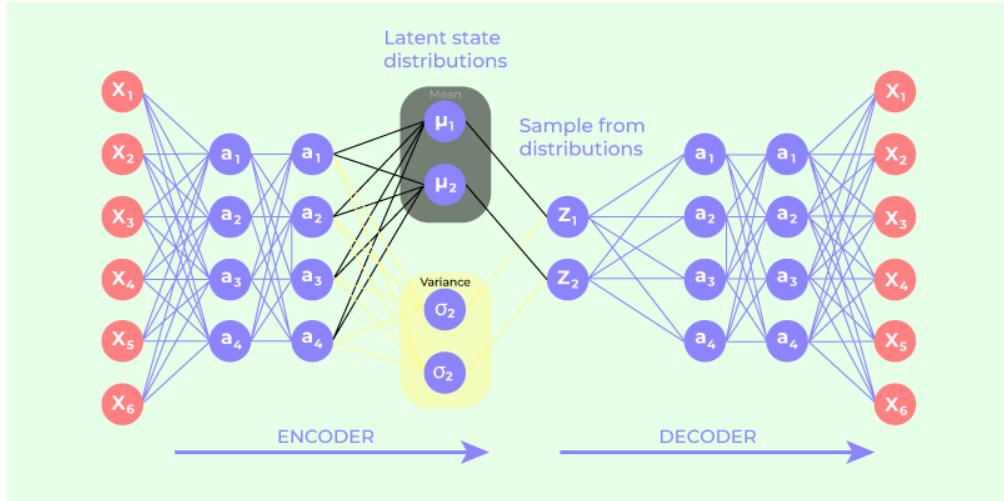
2. Latent Space (Adding Some Randomness)

- Instead of encoding input into a fixed number VAEs introduce randomness to create variations.
- The model picks a point from the range to create different variations of the data.
- This is what makes VAEs great for generating new slightly different but realistic data.

3. Decoder (Reconstructing or Creating New Data)

- The **decoder** takes this sampled value and tries to reconstruct the original input.
- Since the encoder creates a range of possibilities instead of a fixed number the decoder can generate new similar data instead of just memorizing the input.

Deep Learning Lab Manual (CS 405)



Reset data

```
### read dataset
train = pd.read_csv("data/fashion-mnist_train.csv")
train_x = train[list(train.columns)[1:]].values
train_x, val_x = train_test_split(train_x, test_size=0.2)

## create train and validation datasets
train_x, val_x = train_test_split(train_x, test_size=0.2)
```

```
## normalize and reshape
train_x = train_x/255.
val_x = val_x/255.

train_x = train_x.reshape(-1, 28, 28, 1)
val_x = val_x.reshape(-1, 28, 28, 1)
```

Setup Encoder Neural Network

Try different number of hidden layers, nodes?

```
import keras.backend as K
```

```
batch_size = 16
latent_dim = 2 # Number of latent dimension parameters

input_img = Input(shape=(784,), name="input")
x = Dense(512, activation='relu', name="intermediate_encoder")(input_img)
x = Dense(2, activation='relu', name="latent_encoder")(x)

z_mu = Dense(latent_dim)(x)
z_log_sigma = Dense(latent_dim)(x)
```

Deep Learning Lab Manual (CS 405)

```
# sampling function
def sampling(args):
    z_mu, z_log_sigma = args
    epsilon = K.random_normal(shape=(K.shape(z_mu)[0], latent_dim),
                               mean=0., stddev=1.)
    return z_mu + K.exp(z_log_sigma) * epsilon

# sample vector from the latent distribution
z = Lambda(sampling)([z_mu, z_log_sigma])
```

```
# decoder takes the latent distribution sample as input
decoder_input = Input((2,), name="input_decoder")

x = Dense(512, activation='relu', name="intermediate_decoder", input_shape=(2,))(decoder_input)

# Expand to 784 total pixels
x = Dense(784, activation='sigmoid', name="original_decoder")(x)

# decoder model statement
decoder = Model(decoder_input, x)

# apply the decoder to the sample from the latent distribution
z_decoded = decoder(z)
```

```
decoder.summary()
```

```
# construct a custom layer to calculate the loss
class CustomVariationalLayer(Layer):

    def vae_loss(self, x, z_decoded):
        x = K.flatten(x)
        z_decoded = K.flatten(z_decoded)
        # Reconstruction loss
        xent_loss = binary_crossentropy(x, z_decoded)
        return xent_loss

    # adds the custom loss to the class
    def call(self, inputs):
        x = inputs[0]
        z_decoded = inputs[1]
        loss = self.vae_loss(x, z_decoded)
        self.add_loss(loss, inputs=inputs)
        return x
```

Deep Learning Lab Manual (CS 405)

```
# apply the custom loss to the input images and the decoded latent distribution sample  
y = CustomVariationalLayer()([input_img, z_decoded])
```

```
z_decoded
```

```
# VAE model statement  
vae = Model(input_img, y)  
vae.compile(optimizer='rmsprop', loss=None)
```

```
vae.summary()
```

```
train_x.shape
```

```
train_x = train_x.reshape(-1, 784)  
val_x = val_x.reshape(-1, 784)
```

```
vae.fit(x=train_x, y=None,  
        shuffle=True,  
        epochs=20,  
        batch_size=batch_size,  
        validation_data=(val_x, None))
```

```
# Display a 2D manifold of the samples  
n = 20 # figure with 20x20 samples  
digit_size = 28  
figure = np.zeros((digit_size * n, digit_size * n))  
  
# Construct grid of latent variable values - can change values here to generate different things  
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))  
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))  
  
# decode for each square in the grid  
for i, yi in enumerate(grid_x):  
    for j, xi in enumerate(grid_y):  
        z_sample = np.array([[xi, yi]])  
        z_sample = np.tile(z_sample, batch_size).reshape(batch_size, 2)  
  
        x_decoded = decoder.predict(z_sample, batch_size=batch_size)  
  
        digit = x_decoded[0].reshape(digit_size, digit_size)  
  
        figure[i * digit_size: (i + 1) * digit_size,
```

Deep Learning Lab Manual (CS 405)

```
j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(20, 20))
plt.imshow(figure)
plt.show()
```

```
### read dataset
train = pd.read_csv("fashion-mnist_train.csv")
train_x = train[list(train.columns)[1:]].values
train_y = train[list(train.columns)[0]].values

train_x = train_x/255.
# train_x = train_x.reshape(-1, 28, 28, 1)

# Translate into the latent space
encoder = Model(input_img, z_mu)
x_valid_noTest_encoded = encoder.predict(train_x, batch_size=batch_size)
plt.figure(figsize=(10, 10))
plt.scatter(x_valid_noTest_encoded[:, 0], x_valid_noTest_encoded[:, 1], c=train_y, cmap='brg')
plt.colorbar()
plt.show()
```

Exercise: Generating New Fashion using VAEs: Adding CNNs and KL Divergence Loss

```
batch_size = 16
latent_dim = 2 # Number of latent dimension parameters

# Hint: Encoder architecture: Input -> Conv2D*4 -> Flatten -> Dense
input_img = Input(shape=(28, 28, 1))

#Todo: Setup encoder network

shape_before_flattening = K.int_shape(x)

x = Flatten()(x)
x = Dense(32, activation='relu')(x)

# Two outputs, latent mean and (log)variance
z_mu = Dense(latent_dim)(x)
z_log_sigma = Dense(latent_dim)(x)
```

Set up sampling function

```
# sampling function
def sampling(args):
    z_mu, z_log_sigma = args
```

Deep Learning Lab Manual (CS 405)

```
epsilon = K.random_normal(shape=(K.shape(z_mu)[0], latent_dim),
                           mean=0., stddev=1.)
return z_mu + K.exp(z_log_sigma) * epsilon

# sample vector from the latent distribution
z = Lambda(sampling)([z_mu, z_log_sigma])
```

Setup Decoder Neural Network

Try different number of hidden layers, nodes?

```
# decoder takes the latent distribution sample as input
decoder_input = Input(K.int_shape(z)[1:])

#Todo: Setup decoder network
#Hint Expand to 784 pixels -> reshape -> Conv2DTranspose -> conv2D

# decoder model statement
decoder = Model(decoder_input, x)

# apply the decoder to the sample from the latent distribution
z_decoded = decoder(z)
```

Set up loss functions

```
# construct a custom layer to calculate the loss
class CustomVariationalLayer(Layer):

    def vae_loss(self, x, z_decoded):
        x = K.flatten(x)
        z_decoded = K.flatten(z_decoded)
        # Reconstruction loss
        xent_loss = binary_crossentropy(x, z_decoded)
        # KL divergence
        kl_loss = -5e-4 * K.mean(1 + z_log_sigma - K.square(z_mu) - K.exp(z_log_sigma), axis=-1)
        return K.mean(xent_loss + kl_loss)

    # adds the custom loss to the class
    def call(self, inputs):
        x = inputs[0]
        z_decoded = inputs[1]
        loss = self.vae_loss(x, z_decoded)
        self.add_loss(loss, inputs=inputs)
        return x

# apply the custom loss to the input images and the decoded latent distribution sample
y = CustomVariationalLayer()([input_img, z_decoded])
```

Deep Learning Lab Manual (CS 405)

Train VAE

```
# VAE model statement
vae = Model(input_img, y)
vae.compile(optimizer='rmsprop', loss=None)
```

```
vae.summary()
```

```
train_x = train_x.reshape(-1, 28, 28, 1)
val_x = val_x.reshape(-1, 28, 28, 1)
```

```
vae.fit(x=train_x, y=None,
        shuffle=True,
        epochs=20,
        batch_size=batch_size,
        validation_data=(val_x, None))
```

Visualize Samples reconstructed by VAE

```
# Display a 2D manifold of the samples
n = 20 # figure with 20x20 samples
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))

# Construct grid of latent variable values - can change values here to generate different things
grid_x = norm.ppf(np.linspace(0.05, 0.95, n))
grid_y = norm.ppf(np.linspace(0.05, 0.95, n))

# decode for each square in the grid
for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])
        z_sample = np.tile(z_sample, batch_size).reshape(batch_size, 2)
        x_decoded = decoder.predict(z_sample, batch_size=batch_size)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(20, 20))
plt.imshow(figure)
plt.show()
```

Deep Learning Lab Manual (CS 405)

EXPERIMENT 7 – Train the Generative Modeling using Generative Adversarial Networks (GANs)

Objective

To understand and implement Generative Adversarial Networks (GANs) for synthetic data generation by training generator and discriminator models using adversarial learning, and to explore challenges such as mode collapse and higher-order distribution learning.

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

Introduction

In this lab we will look at Generative Adversarial Networks (GANs), their construction and training.
By the end of this lab, you should:

- know how to put together the building blocks used in GANs
- have a good understanding of generative models and implicit distributions that generators learn
- learning properties of GAN at a small scale
- concepts of adversarial training - min-max etc.
- mode collapse problems in GAN

Generate 1-D Gaussian Distribution from Uniform Noise

In this exercise, we are going to generate 1-D Gaussian distribution from a n-D uniform distribution. This is a toy exercise in order to understand the ability of GANs (generators) to generate arbitrary distributions from random noise.

To Install Google App Engine. Create hello world app and other simple web applications using python/java.

Generate training data - Gaussian Distribution

```
def generate_data(n_samples = 10000,n_dim=1):
    return np.random.randn(n_samples, n_dim)
```

Let us define a function that gives you a keras model of general feedforward network based on the parameters.

#INPUT is of input_dim,, goes through n_layers number of hidden layers and output is of output_dim

```
def set_model(input_dim, output_dim, hidden_dim=64,n_layers = 1,activation='tanh',
            optimizer='adam', loss = 'binary_crossentropy'):

    #### YOUR CODE HERE ####
```

#INPUT (z) is of random_dim dimension

Deep Learning Lab Manual (CS 405)

```
#OUTPUT should be a keras model - D(G(z)) - Discriminator score for the generator's images generated  
#from synthetic data.
```

```
def get_gan_network(discriminator, random_dim, generator, optimizer = 'adam'):  
    ### YOUR CODE HERE ###
```

Let us now write the training function for a GAN

```
NOISE_DIM = 10  
DATA_DIM = 1  
G_LAYERS = 1  
D_LAYERS = 1
```

```
def train_gan(epochs=1,batch_size=128):  
    x_train = generate_data(n_samples=12800,n_dim=DATA_DIM)  
    batch_count = x_train.shape[0]/batch_size  
  
    generator = set_model(NOISE_DIM, DATA_DIM, n_layers=G_LAYERS, activation='tanh',loss = 'mean_squared_error')  
    discriminator = set_model(DATA_DIM, 1, n_layers= D_LAYERS, activation='sigmoid')  
    gan = get_gan_network(discriminator, NOISE_DIM, generator, 'adam')  
  
    for e in range(1,epochs+1):  
  
        # Noise is generated from a uniform distribution  
        noise = np.random.rand(batch_size,NOISE_DIM)  
        true_batch = x_train[np.random.choice(x_train.shape[0], batch_size, replace=False), :]  
  
        generated_values = generator.predict(noise)  
        X = np.concatenate([generated_values,true_batch])  
  
        y_dis = np.zeros(2*batch_size)  
  
        #One-sided label smoothing to avoid overconfidence. In GAN, if the discriminator depends on a small set of features to  
        #detect real images,  
        #the generator may just produce these features only to exploit the discriminator.  
        #The optimization may turn too greedy and produces no long term benefit.  
        #To avoid the problem, we penalize the discriminator when the prediction for any real images go beyond 0.9 (D(real  
        #image)>0.9).  
        y_dis[:batch_size] = 0.9  
  
        discriminator.trainable = True  
        ###YOUR CODE HERE###  
        # One line : Train discriminator using train_on_batch  
        discriminator.trainable = False  
  
        # Train generator. Noise is generated from a uniform distribution  
        ### YOUR CODE HERE. Couple of lines. Should call gan.train_on_batch()###
```

Deep Learning Lab Manual (CS 405)

```
return generator, discriminator
```

```
generator, discriminator = train_gan()
```

Let us visualize what the generator has learned.

```
noise = np.random.rand(10000,NOISE_DIM)
generated_values = generator.predict(noise)
plt.hist(generated_values,bins=100)

true_gaussian = [np.random.randn() for x in range(10000)]

print('1st order moment - ', 'True : ', scipy.stats.moment(true_gaussian, 1) , ', GAN :', scipy.stats.moment(generated_values,1))
print('2nd order moment - ', 'True : ', scipy.stats.moment(true_gaussian, 2) , ', GAN :',
scipy.stats.moment(generated_values,2))
print('3rd order moment - ', 'True : ', scipy.stats.moment(true_gaussian, 3) , ', GAN :',
scipy.stats.moment(generated_values,3))
print('4th order moment - ', 'True : ', scipy.stats.moment(true_gaussian, 4) , ', GAN :',
scipy.stats.moment(generated_values,4))
plt.show()
```

CONCLUSIONS

1. GANs are able to learn a generative model from arbitrary noise distributions.

2. Traditional GANs do not learn the higher-order moments well. Possible issues : Number of samples, approximating higher moments is hard. Usually known to under-predict higher order variances. For people interested in learning why, read more about divergence measures between distributions (particularly about Wasserstein etc.)

1. Try different noise dimensions and see what minimum dimension you need to learn this well.
2. Try to generate multimodal distribution like a Gaussian Mixture instead of simple Gaussian and see if GAN is able to learn multimodal distributions well.

EXERCISE 2 : MNIST GAN - Learn to generate MNIST digits

```
from keras.datasets import mnist
from keras.utils import np_utils
from keras.models import Sequential, Model
from keras.layers import Input, Dense, Dropout, Activation, Flatten
from keras.layers.advanced_activations import LeakyReLU
from keras.optimizers import Adam, RMSprop
```

Deep Learning Lab Manual (CS 405)

```
import numpy as np
import matplotlib.pyplot as plt
import random
from tqdm import tqdm_notebook

# Dataset of 60,000 28x28 grayscale images of the 10 digits, along with a test set of 10,000 images.
(X_train, Y_train), (X_test, Y_test) = mnist.load_data()
```

Rescale data since we are using ReLU activations. WHY ?

```
X_train = X_train.reshape(60000, 784)
X_test = X_test.reshape(10000, 784)
X_train = X_train.astype('float32')/255
X_test = X_test.astype('float32')/255
```

```
z_dim = 100
```

BUILD MODEL

We are using LeakyReLU activations.

We will build

- a.) Generator
- b.) Discriminator
- c.) GAN

as feedforwards with multiple layers, dropout and LeakyReLU.

```
#@title
adam = Adam(lr=0.0002, beta_1=0.5)

#GENERATOR
g = Sequential()
#Build your generator - noise_dim -> 256 -> 512 ->1024 ->784. LeakyReLU(0.2), adam
###YOUR CODE HERE###

#DISCRIMINATOR
#Build your discriminator - 784 -> 1024 -> 512 -> 256 -> 1. LeakyRelu(0.2), adam
d = Sequential()
###YOUR CODE HERE###

#GAN
d.trainable = False
```

Deep Learning Lab Manual (CS 405)

```
inputs = Input(shape=(z_dim, ))
hidden = g(inputs)
output = d(hidden)
gan = Model(inputs, output)
gan.compile(loss='binary_crossentropy', optimizer=adam, metrics=['accuracy'])
```

Let us write some visualization code.

```
def plot_loss(losses):
    """
    @losses.keys():
        0: loss
        1: accuracy
    """
    d_loss = [v[0] for v in losses["D"]]
    g_loss = [v[0] for v in losses["G"]]

    plt.figure(figsize=(10,8))
    plt.plot(d_loss, label="Discriminator loss")
    plt.plot(g_loss, label="Generator loss")
    plt.xlabel('Epochs')
    plt.ylabel('Loss')
    plt.legend()
    plt.show()

def plot_generated(n_ex=10, dim=(1, 10), figsize=(12, 2)):
    noise = np.random.normal(0, 1, size=(n_ex, z_dim))
    generated_images = g.predict(noise)
    generated_images = generated_images.reshape(n_ex, 28, 28)

    plt.figure(figsize=figsize)
    for i in range(generated_images.shape[0]):
        plt.subplot(dim[0], dim[1], i+1)
        plt.imshow(generated_images[i], interpolation='nearest', cmap='gray_r')
        plt.axis('off')
    plt.tight_layout()
    plt.show()
```

TRAIN THE MODEL

Generate noise, feed into generator, compare them with discriminator, train the GAN and REPEAT.

```
losses = { "D":[], "G":[] }

def train(epochs=1, plt_frq=1, BATCH_SIZE=128):
    batchCount = int(X_train.shape[0] / BATCH_SIZE)
```

Deep Learning Lab Manual (CS 405)

```
print('Epochs:', epochs)
print('Batch size:', BATCH_SIZE)
print('Batches per epoch:', batchCount)

for e in tqdm_notebook(range(1, epochs+1)):
    if e == 1 or e%plt_frq == 0:
        print('*'*15, 'Epoch %d' % e, '*'*15)
    for _ in range(batchCount): # tqdm_notebook(range(batchCount), leave=False):
        # Create a batch by drawing random index numbers from the training set
        image_batch = X_train[np.random.randint(0, X_train.shape[0], size=BATCH_SIZE)]

        # Create noise vectors for the generator
        noise = np.random.normal(0, 1, size=(BATCH_SIZE, z_dim))

        # Generate the images from the noise
        generated_images = g.predict(noise)
        X = np.concatenate((image_batch, generated_images))

        # Create Y labels similar to last exercise.
        ### YOUR CODE HERE ####

        # Train gan and discriminator similar to last exercise.
        ##YOUR CODE HERE##

    # Only store losses from final
    losses["D"].append(d_loss)
    losses["G"].append(g_loss)

    # Update the plots
    if e == 1 or e%plt_frq == 0:
        plot_generated()
        plot_loss(losses)

train(epochs=200, plt_frq=40, BATCH_SIZE=128)
```

Deep Learning Lab Manual (CS 405)

Experiment No: 8. Implement the Reinforcement Learning with OpenAI Gym using the FrozenLake Environment

Objective:

To apply reinforcement learning principles in a grid-world scenario using the FrozenLake environment from OpenAI Gym, focusing on state transitions, reward-based learning, and policy execution in non-slippery stochastic conditions.

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

Introduction

Winter is here. You and your friends were tossing around a frisbee at the park when you made a wild throw that left the frisbee out in the middle of the lake. The water is mostly frozen, but there are a few holes where the ice has melted. If you step into one of those holes, you'll fall into the freezing water. At this time, there's an international frisbee shortage, so it's absolutely imperative that you navigate across the lake and retrieve the disc.

The agent controls the movement of a character in a grid world. Some tiles of the grid are walkable, and others lead to the agent falling into the water. Additionally, the movement direction of the agent is uncertain and only partially depends on the chosen direction. The agent is rewarded for finding a walkable path to a goal tile.

The surface is described using a grid like the following:

[PP: WOULD IT BETTER TO INCLUDE A DIAGRAM]

- S: starting point, safe
- F: frozen surface, safe
- H: hole, fall to your doom
- G: goal, where the frisbee is located

SFFF

FHFH

FFFH

HFFG

Expected actions are Left(0), Right(1), Down(2), Up(3).

The episode ends when you reach the goal or fall in a hole. You receive a reward of 1 if you reach the goal, and zero otherwise.

```
import gym  
from gym.envs.registration import register
```

Deep Learning Lab Manual (CS 405)

```
register(id='FrozenLakeNotSlippery-v0',
entry_point='gym.envs.toy_text:FrozenLakeEnv',
kwargs={'map_name' : '4x4', 'is_slippery': False},
max_episode_steps=100,
reward_threshold=0.8196, # optimum = .8196
)
```

```
from gym.envs.registration import register
register(
    id='FrozenLake8x8NotSlippery-v0',
    entry_point='gym.envs.toy_text:FrozenLakeEnv',
    kwargs={'map_name' : '8x8', 'is_slippery': False},
    max_episode_steps=100,
    reward_threshold=0.8196, # optimum = .8196
)
```

hint: If you receive an error message while registering the above env the second time you run this cell again, ignore the error message or restart the kernel.

Throughout the assignment, use only the environments we registered in the previous cells:

- `FrozenLake8x8NotSlippery-v0`
- `FrozenLakeNotSlippery-v0`

Even though the original problem description has slippery environment, we are working in a non-slippery environment. In our environment, if you go right, you only go right whereas in the original environment, if you intend to go right, you can go right, up or down with 1/3 probability.

```
import gym
import numpy as np

#Change environment to FrozenLake8x8 to see grid.
env = gym.make('FrozenLake-v0')
# env = gym.make('FrozenLake8x8NotSlippery-v0')

print(env.observation_space.n)

#Both the grids look like as follows.
"""
"4x4": [
    "SFFF",
    "FFHF",
    "FFFH",
    "HFFG"
]
```

Deep Learning Lab Manual (CS 405)

```
],  
"8x8": [  
    "SFFFFFFF",  
    "FFFFFFFF",  
    "FFFHFFFF",  
    "FFFFFHFF",  
    "FFFHFFFF",  
    "FHHFFFHF",  
    "FHFHFHF",  
    "FFFHFFFG"  
]  
}"
```

```
#env.render() prints the frozenlake with an indicator showing where the agent is. You can use it for debugging.  
env.render()
```

```
print(env.observation_space.n)  
print(env.action_space.n)
```

```
Q = np.zeros([env.observation_space.n,env.action_space.n])  
  
def choose_action(state):  
    return np.random.choice(np.array([0,1,2,3]))  
  
def learn(s, s1, r, a):  
    return
```

```
# Set learning parameters  
#####  
  
# num_episodes = 2000  
# epsilon = 0.0  
# max_steps = 100  
# lr_rate = 0.0  
# gamma = 0.0  
# rList = []  
  
num_episodes = 10  
max_iter_per_episode = 20  
for i in range(num_episodes):  
    iter = 0  
  
    #Reset environment and get an initial state - should be done at start of each episode.  
    s = env.reset()  
    rAll = 0  
    d = False
```

Deep Learning Lab Manual (CS 405)

```
j = 0
while iter < max_iter_per_episode:
    iter+=1
    #Choose an action
    a = choose_action(s)
    # env.step() gives you next state, reward, done(whether the episode is over)
    # s1 - new state, r-reward, d-whether you are done or not
    s1,r,d,_ = env.step(a)
    print('State : ',s, ' Action : ', a, ' State 1 : ', s1, ' Reward : ',r, 'Done : ',d)

    learn(s, s1, r, a)

    if d:
        print('Episode Over')
        if r != 1:
            print('Fell into hole with reward ',r)
        break
    s = s1
if r==1:
    print(i)
    break
```

Deep Learning Lab Manual (CS 405)

Experiment No 9: OPEN ENDED LAB

"Exploring Bias and Uncertainty in AI: Ethical and Performance Perspectives"

Marks: 10

Objective:

This lab aims to provide hands-on experience in analyzing bias and uncertainty in AI models. Students will examine real-world datasets, train deep learning models, and evaluate their performance while considering ethical implications and fairness metrics.

Tasks:

Part 1: Understanding Bias in AI

1. Choose a real-world dataset known for potential biases (e.g., COMPAS dataset, Gender Shades dataset, or biased hiring data).
2. Train a classifier (e.g., logistic regression, deep neural network) to predict an outcome (e.g., recidivism, gender classification).
3. Analyze the performance differences across different demographic groups.
4. Implement fairness metrics such as demographic parity, equalized odds, or disparate impact.
5. Discuss ethical concerns regarding biased predictions and their implications in real-world AI deployment.

Open-ended Question for Discussion:

- How do different fairness interventions (e.g., reweighting data, adversarial debiasing) impact model performance and bias mitigation?

Part 2: Evaluating Uncertainty in AI Models

1. Train a deep learning model (e.g., CNN for image classification or RNN for text classification).
2. Introduce out-of-distribution (OOD) samples and adversarial perturbations to test model robustness.
3. Measure model uncertainty using techniques like:
 - Softmax entropy
 - Bayesian neural networks
 - Monte Carlo Dropout
4. Compare uncertainty estimates across different deep learning architectures.
5. Discuss trade-offs between accuracy and uncertainty estimation in AI decision-making.

Open-ended Question for Discussion:

- How should AI systems communicate uncertainty to users in high-stakes applications (e.g., medical diagnosis, autonomous driving)?

Deliverables:

- **Implementation:** Training, evaluation, and fairness/uncertainty analysis.
- **Lab Report:** Insights on bias and uncertainty, model performance comparison, and ethical reflections.
- **Evaluation (in class):** Presentation and discussion on the ethical implications of biased and uncertain AI decisions.

Course Learning Outcomes Mapping:

Deep Learning Lab Manual (CS 405)

CLO-3: Investigate the ethical implications of deploying AI systems in various contexts → Students will analyze how bias in data and model uncertainty can lead to unfair outcomes in real-world applications.

CLO-4: Gain the ability to design, implement, and train various deep learning models using Tensorflow to solve real-world problems → Students will implement deep learning models using tools such as Google Colab.

Experiment 10: Implement the Deep Reinforcement Learning for Agent-Environment Interaction

Objective:

To understand the fundamentals of reinforcement learning by implementing a deep learning agent that learns to interact with and master simulated environments of varying complexity, specifically the Cartpole environment. This involves defining the environment and agent, understanding the observation and action spaces, and building a neural network policy to guide the agent's actions based on environmental feedback.

Time Required: 3 hrs

Programming Language: e.g. Python

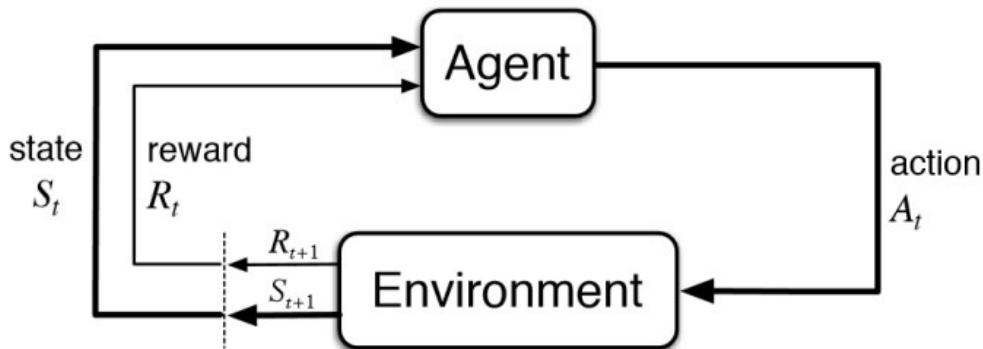
Software/Tools Required: e.g. Anaconda/ Google Colab

Introduction

Reinforcement learning (RL) is a subset of machine learning which poses learning problems as interactions between agents and environments. It often assumes agents have no prior knowledge of a world, so they must learn to navigate environments by optimizing a reward function. Within an environment, an agent can take certain actions and receive feedback, in the form of positive or negative rewards, with respect to their decision. As such, an agent's feedback loop is somewhat akin to the idea of "trial and error", or the manner in which a child might learn to distinguish between "good" and "bad" actions.

In practical terms, our RL agent will interact with the environment by taking an action at each timestep, receiving a corresponding reward, and updating its state according to what it has "learned".

While the ultimate goal of reinforcement learning is to teach agents to act in the real, physical world, simulated environments -- like games and simulation engines -- provide a convenient proving ground for developing RL algorithms and agents.



In previous labs, we have explored both supervised (with LSTMs, CNNs) and unsupervised / semi-supervised (with VAEs) learning tasks. Reinforcement learning is fundamentally different, in that we are training a deep learning algorithm to govern the actions of our RL agent, that is trying, within its environment, to find the optimal way to achieve a goal. The goal of training an RL agent is to determine the best next step to take to

Deep Learning Lab Manual (CS 405)

earn the greatest final payoff or return. In this lab, we focus on building a reinforcement learning algorithm to master two different environments with varying complexity.

Cartpole: Balance a pole, protruding from a cart, in an upright position by only moving the base left or right. Environment with a low-dimensional observation space.

Let's get started! First we'll import TensorFlow, the course package, and some dependencies.

```
# Import Tensorflow 2.0
%tensorflow_version 2.x
import tensorflow as tf

gpus = tf.config.experimental.list_physical_devices('GPU')
if gpus:
    for gpu in gpus:
        tf.config.experimental.set_memory_growth(gpu, True)

# Download and import the MIT 6.S191 package
!printf "Installing MIT deep learning package... "
!pip install --upgrade git+https://github.com/MITDeepLearning/introtodeeplearning.git &> /dev/null
!echo "Done"
```

```
#Install some dependencies for visualizing the agents
!apt-get install -y xvfb python-opengl x11-utils &> /dev/null
!pip install gym pyvirtualdisplay scikit-video ffio pyrender &> /dev/null
!pip install tensorflow_probability==0.12.0 &> /dev/null
import os
os.environ['PYOPENGL_PLATFORM'] = 'egl'

import numpy as np
import matplotlib, cv2
import matplotlib.pyplot as plt
import base64, io, os, time, gym
import IPython, functools
import time
from tqdm import tqdm
import tensorflow_probability as tfp

import mitdeeplearning as mdl
```

Before we dive in, let's take a step back and outline our approach, which is generally applicable to reinforcement learning problems in general:

1. **Initialize our environment and our agent**: here we will describe the different observations and actions the agent can make in the environment.

Deep Learning Lab Manual (CS 405)

2. **Define our agent's memory**: this will enable the agent to remember its past actions, observations, and rewards.
3. **Define a reward function**: describes the reward associated with an action or sequence of actions.
4. **Define the learning algorithm**: this will be used to reinforce the agent's good behaviors and discourage bad behaviors.

Part 1: Cartpole

3.1 Define the Cartpole environment and agent

Environment

In order to model the environment for the Cartpole task, we'll be using a toolkit developed by OpenAI called [OpenAI Gym](<https://gym.openai.com/>). It provides several pre-defined environments for training and testing reinforcement learning agents, including those for classic physics control tasks, Atari video games, and robotic simulations. To access the Cartpole environment, we can use `env = gym.make("CartPole-v0")`, which we gained access to when we imported the `gym` package. We can instantiate different [environments](https://gym.openai.com/envs/#classic_control) by passing the environment name to the `make` function.

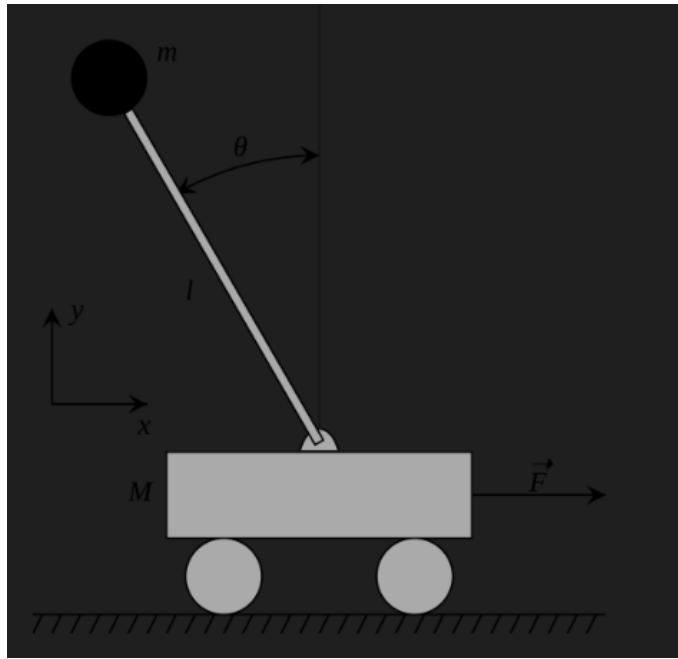
One issue we might experience when developing RL algorithms is that many aspects of the learning process are inherently random: initializing game states, changes in the environment, and the agent's actions. As such, it can be helpful to set a initial "seed" for the environment to ensure some level of reproducibility. Much like you might use `numpy.random.seed`, we can call the comparable function in gym, `seed`, with our defined environment to ensure the environment's random variables are initialized the same each time.

Instantiate the Cartpole environment

```
env = gym.make("CartPole-v1")
env.seed(1)
```

In Cartpole, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pole starts upright, and the goal is to prevent it from falling over. The system is controlled by applying a force of +1 or -1 to the cart. A reward of +1 is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center of the track. A visual summary of the cartpole environment is depicted below:

Deep Learning Lab Manual (CS 405)



Given this setup for the environment and the objective of the game, we can think about: 1) what observations help define the environment's state; 2) what actions the agent can take.

First, let's consider the observation space. In this Cartpole environment our observations are:

1. Cart position
2. Cart velocity
3. Pole angle
4. Pole rotation rate

We can confirm the size of the space by querying the environment's observation space:

```
n_observations = env.observation_space  
print("Environment has observation space =", n_observations)
```

Second, we consider the action space. At every time step, the agent can move either right or left. Again we can confirm the size of the action space by querying the environment:

```
n_actions = env.action_space.n  
print("Number of possible actions that the agent can choose from =", n_actions)
```

Cartpole agent

Now that we have instantiated the environment and understood the dimensionality of the observation and action spaces, we are ready to define our agent. In deep reinforcement learning, a deep neural network defines the agent. This network will take as input an observation of the environment and output the probability of taking each of the possible actions. Since Cartpole is defined by a low-dimensional observation space, a

Deep Learning Lab Manual (CS 405)

simple feed-forward neural network should work well for our agent. We will define this using the `Sequential` API.

```
### Define the Cartpole agent ###

# Defines a feed-forward neural network
def create_cartpole_model():
    model = tf.keras.models.Sequential([
        # First Dense layer
        tf.keras.layers.Dense(units=32, activation='relu'),

        # TODO: Define the last Dense layer, which will provide the network's output.
        # Think about the space the agent needs to act in!
        # ["TODO" Dense layer to output action probabilities]
    ])
    return model

cartpole_model = create_cartpole_model()
```

Now that we have defined the core network architecture, we will define an *action function* that executes a forward pass through the network, given a set of observations, and samples from the output. This sampling from the output probabilities will be used to select the next action for the agent. We will also add support so that the `choose_action` function can handle either a single observation or a batch of observations.

Critically, this action function is totally general -- we will use this function for learning control algorithms for Cartpole, but it is applicable to other RL tasks, as well!

```
### Define the agent's action function ###

# Function that takes observations as input, executes a forward pass through model,
# and outputs a sampled action.
# Arguments:
#   model: the network that defines our agent
#   observation: observation(s) which is/are fed as input to the model
#   single: flag as to whether we are handling a single observation or batch of
#   observations, provided as an np.array
# Returns:
#   action: choice of agent action
def choose_action(model, observation, single=True):
    # add batch dimension to the observation if only a single example was provided
    observation = np.expand_dims(observation, axis=0) if single else observation

    """TODO: feed the observations through the model to predict the log probabilities of each possible action."""
    # logits = model.predict("TODO")
```

Deep Learning Lab Manual (CS 405)

```
'''TODO: Choose an action from the categorical distribution defined by the log
probabilities of each possible action.'''
# action = ["TODO"]

action = action.numpy().flatten()

return action[0] if single else action
```

3.2 Define the agent's memory

Now that we have instantiated the environment and defined the agent network architecture and action function, we are ready to move on to the next step in our RL workflow:

1. **Initialize our environment and our agent**: here we will describe the different observations and actions the agent can make in the environment.
2. **Define our agent's memory**: this will enable the agent to remember its past actions, observations, and rewards.
3. **Define the learning algorithm**: this will be used to reinforce the agent's good behaviors and discourage bad behaviors.

In reinforcement learning, training occurs alongside the agent's acting in the environment; an **episode** refers to a sequence of actions that ends in some terminal state, such as the pole falling down or the cart crashing. The agent will need to remember all of its observations and actions, such that once an episode ends, it can learn to "reinforce" the good actions and punish the undesirable actions via training. Our first step is to define a simple `Memory` buffer that contains the agent's observations, actions, and received rewards from a given episode. We will also add support to combine a list of `Memory` objects into a single `Memory`. This will be very useful for batching, which will help you accelerate training later on in the lab.

Once again, note the modularity of this memory buffer -- it can and will be applied to other RL tasks as well!**

```
### Agent Memory ###

class Memory:
    def __init__(self):
        self.clear()

    # Resets/restarts the memory buffer
    def clear(self):
        self.observations = []
        self.actions = []
        self.rewards = []

    # Add observations, actions, rewards to memory
```

Deep Learning Lab Manual (CS 405)

```
def add_to_memory(self, new_observation, new_action, new_reward):
    self.observations.append(new_observation)

    """TODO: update the list of actions with new action"""
    # [{"TODO"}]

    """TODO: update the list of rewards with new reward"""
    # [{"TODO"}]

def __len__(self):
    return len(self.actions)

# Instantiate a single Memory buffer
memory = Memory()
```

3.3 Reward function

We're almost ready to begin the learning algorithm for our agent! The next step is to compute the rewards of our agent as it acts in the environment. Since we (and the agent) is uncertain about if and when the game or task will end (i.e., when the pole will fall), it is useful to emphasize getting rewards **now** rather than later in the future -- this is the idea of discounting. This is a similar concept to discounting money in the case of interest. Recall from lecture, we use reward discount to give more preference at getting rewards now rather than later in the future. The idea of discounting rewards is similar to discounting money in the case of interest.

To compute the expected cumulative reward, known as the **return**, at a given timestep in a learning episode, we sum the discounted rewards expected at that time step t , within a learning episode, and projecting into the future. We define the return (cumulative reward) at a time step t , R_t as:

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

where $0 < \gamma < 1$ is the discount factor and r_t is the reward at time step t , and the index k increments projection into the future within a single learning episode. Intuitively, you can think of this function as depreciating any rewards received at later time steps, which will force the agent prioritize getting rewards now. Since we can't extend episodes to infinity, in practice the computation will be limited to the number of timesteps in an episode -- after that the reward is assumed to be zero.

Take note of the form of this sum -- we'll have to be clever about how we implement this function. Specifically, we'll need to initialize an array of zeros, with length of the number of time steps, and fill it with the real discounted reward values as we loop through the rewards from the episode, which will have been saved in the agents memory. What we ultimately care about is which actions are better relative to other actions taken in that episode -- so, we'll normalize our computed rewards, using the mean and standard deviation of the rewards across the learning episode.

Deep Learning Lab Manual (CS 405)

We will use this definition of the reward function in both parts of the lab so make sure you have it executed!

```
### Reward function ###
```

```
# Helper function that normalizes an np.array x
def normalize(x):
    x -= np.mean(x)
    x /= np.std(x)
    return x.astype(np.float32)

# Compute normalized, discounted, cumulative rewards (i.e., return)
# Arguments:
#   rewards: reward at timesteps in episode
#   gamma: discounting factor
# Returns:
#   normalized discounted reward
def discount_rewards(rewards, gamma=0.95):
    discounted_rewards = np.zeros_like(rewards)
    R = 0
    for t in reversed(range(0, len(rewards))):
        # update the total discounted reward
        R = R * gamma + rewards[t]
        discounted_rewards[t] = R

    return normalize(discounted_rewards)
```

3.4 Learning algorithm

Now we can start to define the learning algorithm which will be used to reinforce good behaviors of the agent and discourage bad behaviours. In this lab, we will focus on **policy gradient** methods which aim to ***maximize*** the likelihood of actions that result in large rewards. Equivalently, this means that we want to ***minimize*** the negative likelihood of these same actions. We achieve this by simply ***scaling*** the probabilities by their associated rewards -- effectively amplifying the likelihood of actions that result in large rewards.

Since the log function is monotonically increasing, this means that minimizing ***negative likelihood*** is equivalent to minimizing ***negative log-likelihood***. Recall that we can easily compute the negative log-likelihood of a discrete action by evaluating its [softmax cross entropy](https://www.tensorflow.org/api_docs/python/tf/nn/sparse_softmax_cross_entropy_with_logits). Like in supervised learning, we can use stochastic gradient descent methods to achieve the desired minimization.

Let's begin by defining the loss function.

Deep Learning Lab Manual (CS 405)

```
### Loss function ###

# Arguments:
# logits: network's predictions for actions to take
# actions: the actions the agent took in an episode
# rewards: the rewards the agent received in an episode
# Returns:
# loss

def compute_loss(logits, actions, rewards):
    """TODO: complete the function call to compute the negative log probabilities"""
    # neg_logprob = tf.nn.sparse_softmax_cross_entropy_with_logits(
    #     logits="TODO", labels="TODO")

    """TODO: scale the negative log probability by the rewards"""
    # loss = tf.reduce_mean("TODO")
    return loss
```

Now let's use the loss function to define a training step of our learning algorithm. This is a very generalizable definition which we will use

```
### Training step (forward and backpropagation) ###

def train_step(model, loss_function, optimizer, observations, actions, discounted_rewards, custom_fwd_fn=None):
    with tf.GradientTape() as tape:
        # Forward propagate through the agent network
        if custom_fwd_fn is not None:
            prediction = custom_fwd_fn(observations)
        else:
            prediction = model(observations)

        """TODO: call the compute_loss function to compute the loss"""
        # loss = loss_function("TODO", "TODO", "TODO")

    """TODO: run backpropagation to minimize the loss using the tape.gradient method.
    Unlike supervised learning, RL is *extremely* noisy, so you will benefit
    from additionally clipping your gradients to avoid falling into
    dangerous local minima. After computing your gradients try also clipping
    by a global normalizer. Try different clipping values, usually clipping
    between 0.5 and 5 provides reasonable results. """
    # grads = tape.gradient("TODO", "TODO")

    # grads, _ = tf.clip_by_global_norm(grads, "TODO")
    # optimizer.apply_gradients(zip(grads, model.trainable_variables))
```

Deep Learning Lab Manual (CS 405)

3.5 Run cartpole!

Having had no prior knowledge of the environment, the agent will begin to learn how to balance the pole on the cart based only on the feedback received from the environment! Having defined how our agent can move, how it takes in new observations, and how it updates its state, we'll see how it gradually learns a policy of actions to optimize balancing the pole as long as possible. To do this, we'll track how the rewards evolve as a function of training -- how should the rewards change as training progresses?

```
## Training parameters ##
## Re-run this cell to restart training from scratch ##

# TODO: Learning rate and optimizer
# learning_rate = "TODO"

# optimizer = "TODO"

# instantiate cartpole agent
cartpole_model = create_cartpole_model()

# to track our progress
smoothed_reward = mdl.util.LossHistory(smoothing_factor=0.95)
plotter = mdl.util.PeriodicPlotter(sec=2, xlabel='Iterations', ylabel='Rewards')
```

```
## Cartpole training! ##
## Note: stoping and restarting this cell will pick up training where you
#      left off. To restart training you need to rerun the cell above as
#      well (to re-initialize the model and optimizer)

if hasattr(tqdm, '_instances'): tqdm._instances.clear() # clear if it exists
for i_episode in range(500):

    plotter.plot(smoothed_reward.get())
    # Restart the environment
    observation = env.reset()
    memory.clear()

    while True:
        # using our observation, choose an action and take it in the environment
        action = choose_action(cartpole_model, observation)
        next_observation, reward, done, info = env.step(action)
        # add to memory
        memory.add_to_memory(observation, action, reward)

        # is the episode over? did you crash or do so well that you're done?
        if done:
```

Deep Learning Lab Manual (CS 405)

```
# determine total reward and keep a record of this
total_reward = sum(memory.rewards)
smoothed_reward.append(total_reward)

# initiate training - remember we don't know anything about how the
# agent is doing until it has crashed!
g = train_step(cartpole_model, compute_loss, optimizer,
               observations=np.vstack(memory.observations),
               actions=np.array(memory.actions),
               discounted_rewards = discount_rewards(memory.rewards))

# reset the memory
memory.clear()
break

# update our observations
observation = next_observation
```

To get a sense of how our agent did, we can save a video of the trained model working on balancing the pole. Realize that this is a brand new environment that the agent has not seen before!

Let's display the saved video to watch how our agent did!

```
matplotlib.use('Agg')
saved_cartpole = mdl.lab3.save_video_of_model(cartpole_model, "CartPole-v1")
mdl.lab3.play_video(saved_cartpole)
```

Deep Learning Lab Manual (CS 405)

Experiment No 11: Train and Implement the End-to-End Training of Autonomous Driving Policies in VISTA Simulator

Objective:

To investigate the potential of deep learning for creating autonomous driving policies that learn directly from raw sensory input (RGB camera data) within a photorealistic simulated environment (VISTA). The goal is to train a neural network-based controller for the task of lane following, demonstrating the concept of end-to-end learning for autonomous vehicle control. To explore the power of deep learning to learn autonomous control policies that are trained *end-to-end, directly from raw sensory data, and entirely within a simulated world*.**

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

Introduction

We will use the data-driven simulation engine

[VISTA](<https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8957584&tag=1>), which uses techniques in computer vision to synthesize new photorealistic trajectories and driving viewpoints, that are still consistent with the world's appearance and fall within the envelope of a real driving scene. This is a powerful approach -- we can synthesize data that is photorealistic, grounded in the real world, and then use this data for training and testing autonomous vehicle control policies within this simulator.

In this part of the lab, you will use reinforcement learning to build a self-driving agent with a neural network-based controller trained on RGB camera data. We will train the self-driving agent for the task of lane following. Beyond this data modality and control task, VISTA also supports [different data modalities](<https://arxiv.org/pdf/2111.12083.pdf>) (such as LiDAR data) and [different learning tasks](<https://arxiv.org/pdf/2111.12137.pdf>) (such as multi-car interactions).

Procedure:

Training Autonomous Driving Policies in VISTA

Autonomous control has traditionally been dominated by algorithms that explicitly decompose individual aspects of the control pipeline. For example, in autonomous driving, traditional methods work by first detecting road and lane boundaries, and then using path planning and rule-based methods to derive a control policy. Deep learning offers something very different -- the possibility of optimizing all these steps simultaneously, learning control end-to-end directly from raw sensory inputs.

You will put your agent to the test in the VISTA environment, and potentially, on board a full-scale autonomous vehicle! Specifically, as part of the MIT lab competitions, high-performing agents -- evaluated based on the maximum distance they can travel without crashing -- will have the opportunity to be put to the ***real*** test onboard a full-scale autonomous vehicle!!!

Deep Learning Lab Manual (CS 405)

We start by installing dependencies. This includes installing the VISTA package itself.

```
!pip install --upgrade git+https://github.com/vista-simulator/vista-6s191.git
```

```
import vista
from vista.utils import logging
logging.setLevel(logging.ERROR)
```

VISTA provides some documentation which will be very helpful to completing this lab. You can always use the `?vista` command to access the package documentation.

```
### Access documentation for VISTA
### Run ?vista.<[name of module or function]>
?vista.Display
```

12.6 Create an environment in VISTA

Environments in VISTA are based on and built from human-collected driving *traces*. A trace is the data from a single driving run. In this case we'll be working with RGB camera data, from the viewpoint of the driver looking out at the road: the camera collects this data as the car drives around!

We will start by accessing a trace. We use that trace to instantiate an environment within VISTA. This is our `World` and defines the environment we will use for reinforcement learning. The trace itself helps to define a space for the environment; with VISTA, we can use the trace to generate new photorealistic viewpoints anywhere within that space. This provides valuable new training data as well as a robust testing environment.

The simulated environment of VISTA will serve as our training ground and testbed for reinforcement learning. We also define an `Agent` -- a car -- that will actually move around in the environment, and make and carry out *actions* in this world. Because this is an entirely simulated environment, our car agent will also be simulated!

```
# Download and extract the data for vista (auto-skip if already downloaded)
!wget -nc -q --show-progress https://www.dropbox.com/s/62pao4mipyzk3xu/vista_traces.zip
print("Unzipping data...")
!unzip -o -q vista_traces.zip
print("Done downloading and unzipping data!")

trace_root = "./vista_traces"
trace_path = [
    "20210726-154641_lexus_devens_center",
    "20210726-155941_lexus_devens_center_reverse",
    "20210726-184624_lexus_devens_center",
    "20210726-184956_lexus_devens_center_reverse",
]
trace_path = [os.path.join(trace_root, p) for p in trace_path]
```

Deep Learning Lab Manual (CS 405)

```
# Create a virtual world with VISTA, the world is defined by a series of data traces
world = vista.World(trace_path, trace_config={'road_width': 4})

# Create a car in our virtual world. The car will be able to step and take different
# control actions. As the car moves, its sensors will simulate any changes it environment
car = world.spawn_agent(
    config={
        'length': 5.,
        'width': 2.,
        'wheel_base': 2.78,
        'steering_ratio': 14.7,
        'lookahead_road': True
    })
)

# Create a camera on the car for synthesizing the sensor data that we can use to train with!
camera = car.spawn_camera(config={'size': (200, 320)})

# Define a rendering display so we can visualize the simulated car camera stream and also
# get see its physical location with respect to the road in its environment.
display = vista.Display(world, display_config={"gui_scale": 2, "vis_full_frame": False})

# Define a simple helper function that allows us to reset VISTA and the rendering display
def vista_reset():
    world.reset()
    display.reset()
vista_reset()
```

If successful, you should see a blank black screen at this point. Your rendering display has been initialized.

3.7 Our virtual agent: the car

Our goal is to learn a control policy for our agent, our (hopefully) autonomous vehicle, end-to-end directly from RGB camera sensory input. As in Cartpole, we need to define how our virtual agent will interact with its environment.

Define agent's action functions

In the case of driving, the car agent can act -- taking a step in the VISTA environment -- according to a given control command. This amounts to moving with a desired speed and a desired *curvature*, which reflects the car's turn radius. Curvature has units $\frac{1}{\text{meter}}$. So, if a car is traversing a circle of radius r meters, then it is turning with a curvature $\frac{1}{r}$. The curvature is therefore correlated with the car's steering wheel angle, which actually controls its turn radius. Let's define the car agent's step function to capture the action of moving with a desired speed and desired curvature.

```
# First we define a step function, to allow our virtual agent to step
# with a given control command through the environment
```

Deep Learning Lab Manual (CS 405)

```
# agent can act with a desired curvature (turning radius, like steering angle)
# and desired speed. if either is not provided then this step function will
# use whatever the human executed at that time in the real data.

def vista_step(curvature=None, speed=None):
    # Arguments:
    #   curvature: curvature to step with
    #   speed: speed to step with
    if curvature is None:
        curvature = car.trace.f_curvature(car.timestamp)
    if speed is None:
        speed = car.trace.f_speed(car.timestamp)

    car.step_dynamics(action=np.array([curvature, speed]), dt=1/15.)
    car.step_sensors()
```

Inspect driving trajectories in VISTA

Recall that our VISTA environment is based off an initial human-collected driving trace. Also, we defined the agent's step function to defer to what the human executed if it is not provided with a desired speed and curvature with which to move.

Thus, we can further inspect our environment by using the step function for the driving agent to move through the environment by following the human path. The stepping and iteration will take about 1 iteration per second. We will then observe the data that comes out to see the agent's traversal of the environment.

```
import shutil, os, subprocess, cv2

# Create a simple helper class that will assist us in storing videos of the render
class VideoStream():
    def __init__(self):
        self.tmp = "./tmp"
        if os.path.exists(self.tmp) and os.path.isdir(self.tmp):
            shutil.rmtree(self.tmp)
        os.mkdir(self.tmp)
    def write(self, image, index):
        cv2.imwrite(os.path.join(self.tmp, f"{index:04}.png"), image)
    def save(self, fname):
        cmd = f"/usr/bin/ffmpeg -f image2 -i {self.tmp}/%04d.png -crf 0 -y {fname}"
        subprocess.call(cmd, shell=True)
```

```
## Render and inspect a human trace ##
```

```
vista_reset()
```

Deep Learning Lab Manual (CS 405)

```
stream = VideoStream()  
  
for i in tqdm(range(100)):  
    vista_step()  
  
    # Render and save the display  
    vis_img = display.render()  
    stream.write(vis_img[:, :, ::-1], index=i)  
    if car.done:  
        break  
  
print("Saving trajectory of human following...")  
stream.save("human_follow.mp4")  
mdl.lab3.play_video("human_follow.mp4")
```

Check out the simulated VISTA environment. What do you notice about the environment, the agent, and the setup of the simulation engine? How could these aspects useful for training models?

Define terminal states: crashing! (oh no)

Recall from Cartpole, our training episodes ended when the pole toppled, i.e., the agent crashed and failed. Similarly for training vehicle control policies in VISTA, we have to define what a ***crash*** means. We will define a crash as any time the car moves out of its lane or exceeds its maximum rotation. This will define the end of a training episode.

```
## Define terminal states and crashing conditions ##  
  
def check_out_of_lane(car):  
    distance_from_center = np.abs(car.relative_state.x)  
    road_width = car.trace.road_width  
    half_road_width = road_width / 2  
    return distance_from_center > half_road_width  
  
def check_exceed_max_rot(car):  
    maximal_rotation = np.pi / 10.  
    current_rotation = np.abs(car.relative_state.yaw)  
    return current_rotation > maximal_rotation  
  
def check_crash(car):  
    return check_out_of_lane(car) or check_exceed_max_rot(car) or car.done
```

Quick check: acting with a random control policy

At this point, we have (1) an environment; (2) an agent, with a step function. Before we start learning a control policy for our vehicle agent, we start by testing out the behavior of the agent in the virtual world by

Deep Learning Lab Manual (CS 405)

providing it with a completely random control policy. Naturally we expect that the behavior will not be very robust! Let's take a look.

```
## Behavior with random control policy ##

i = 0
num_crashes = 5
stream = VideoStream()

for _ in range(num_crashes):
    vista_reset()

    while not check_crash(car):

        # Sample a random curvature (between +/- 1/3), keep speed constant
        curvature = np.random.uniform(-1/3, 1/3)

        # Step the simulated car with the same action
        vista_step(curvature=curvature)

        # Render and save the display
        vis_img = display.render()
        stream.write(vis_img[:, :, ::-1], index=i)
        i += 1

    print(f"Car crashed on step {i}")
    for _ in range(5):
        stream.write(vis_img[:, :, ::-1], index=i)
        i += 1

print("Saving trajectory with random policy...")
stream.save("random_policy.mp4")
mdl.lab3.play_video("random_policy.mp4")
```

3.8 Preparing to learn a control policy: data preprocessing

So, hopefully you saw that the random control policy was, indeed, not very robust. Yikes. Our overall goal in this lab is to build a self-driving agent using a neural network controller trained entirely in the simulator VISTA. This means that the data used to train and test the self-driving agent will be supplied by VISTA. As a step towards this, we will do some data preprocessing to make it easier for the network to learn from these visual data.

Previously we rendered the data with a display as a quick check that the environment was working properly. For training the agent, we will directly access the car's observations, extract Regions Of Interest (ROI) from

Deep Learning Lab Manual (CS 405)

those observations, crop them out, and use these crops as training data for our self-driving agent controller. Observe both the full observation and the extracted ROI.

```
from google.colab.patches import cv2_imshow

# Directly access the raw sensor observations of the simulated car
vista_reset()
full_obs = car.observations[camera.name]
cv2_imshow(full_obs)
```

```
## ROIs ##

# Crop a smaller region of interest (ROI). This is necessary because:
# 1. The full observation will have distortions on the edge as the car deviates from the human
# 2. A smaller image of the environment will be easier for our model to learn from
region_of_interest = camera.camera_param.get_roi()
i1, j1, i2, j2 = region_of_interest
cropped_obs = full_obs[i1:i2, j1:j2]
cv2_imshow(cropped_obs)
```

We will group these steps into some helper functions that we can use during training:

1. `preprocess`: takes a full observation as input and returns a preprocessed version. This can include whatever preprocessing steps you would like! For example, ROI extraction, cropping, augmentations, and so on. You are welcome to add and modify this function as you seek to optimize your self-driving agent!
2. `grab_and_preprocess`: grab the car's current observation (i.e., image view from its perspective), and then call the `preprocess` function on that observation.

```
## Data preprocessing functions ##

def preprocess(full_obs):
    # Extract ROI
    i1, j1, i2, j2 = camera.camera_param.get_roi()
    obs = full_obs[i1:i2, j1:j2]

    # Rescale to [0, 1]
    obs = obs / 255.
    return obs

def grab_and_preprocess_obs(car):
    full_obs = car.observations[camera.name]
    obs = preprocess(full_obs)
    return obs
```

```
## Data preprocessing functions ##
```

Deep Learning Lab Manual (CS 405)

```
def preprocess(full_obs):
    # Extract ROI
    i1, j1, i2, j2 = camera.camera_param.get_roi()
    obs = full_obs[i1:i2, j1:j2]

    # Rescale to [0, 1]
    obs = obs / 255.
    return obs

def grab_and_preprocess_obs(car):
    full_obs = car.observations[camera.name]
    obs = preprocess(full_obs)
    return obs
```

```
### Define the self-driving agent ###
# Note: we start with a template CNN architecture -- experiment away as you
# try to optimize your agent!

# Functionally define layers for convenience
# All convolutional layers will have ReLu activation
act = tf.keras.activations.swish
Conv2D = functools.partial(tf.keras.layers.Conv2D, padding='valid', activation=act)
Flatten = tf.keras.layers.Flatten
Dense = tf.keras.layers.Dense

# Defines a CNN for the self-driving agent
def create_driving_model():
    model = tf.keras.models.Sequential([
        # Convolutional layers
        # First, 32 5x5 filters and 2x2 stride
        Conv2D(filters=32, kernel_size=5, strides=2),

        # TODO: define convolutional layers with 48 5x5 filters and 2x2 stride
        # Conv2D("TODO"),

        # TODO: define two convolutional layers with 64 3x3 filters and 2x2 stride
        # Conv2D("TODO"),

        Flatten(),

        # Fully connected layer and output
        Dense(units=128, activation=act),

        # TODO: define the output dimension of the last Dense layer.
```

Deep Learning Lab Manual (CS 405)

```
# Pay attention to the space the agent needs to act in.  
# Remember that this model is outputing a distribution of *continuous*  
# actions, which take a different shape than discrete actions.  
# How many outputs should there be to define a distribution?"  
  
# Dense("TODO")  
  
])  
return model  
  
driving_model = create_driving_model()
```

Now we will define the learning algorithm which will be used to reinforce good behaviors of the agent and discourage bad behaviours. As with Cartpole, we will use a **policy gradient** method that aims to ***maximize*** the likelihood of actions that result in large rewards. However, there are some key differences. In Cartpole, the agent's action space was discrete: it could only move left or right. In driving, the agent's action space is continuous: the control network is outputting a curvature, which is a continuous variable. We will define a probability distribution, defined by a mean and variance, over this continuous action space to define the possible actions the self-driving agent can take.

You will define three functions that reflect these changes and form the core of the learning algorithm:

1. `run_driving_model`: takes an input image, and outputs a prediction of a continuous distribution of curvature. This will take the form of a Normal distribution and will be defined using TensorFlow probability's `[tfp.distributions.Normal]`(https://www.tensorflow.org/probability/api_docs/python/tfp/distributions/Normal) function, so the model's prediction will include both a mean and variance. Operates on an instance `driving_model` defined above.
2. `compute_driving_loss`: computes the loss for a prediction that is in the form of a continuous Normal distribution. Recall as in Cartpole, computing the loss involves multiplying the predicted log probabilities by the discounted rewards. Similar to `compute_loss` in Cartpole.

The `train_step` function to use the loss function to execute a training step will be the same as we used in Cartpole! This will have to be executed above in order for the driving agent to train properly.

```
## The self-driving learning algorithm ##  
  
# hyperparameters  
max_curvature = 1/8.  
max_std = 0.1  
  
def run_driving_model(image):  
    # Arguments:  
    # image: an input image  
    # Returns:  
    # pred_dist: predicted distribution of control actions
```

Deep Learning Lab Manual (CS 405)

```
single_image_input = tf.rank(image) == 3 # missing 4th batch dimension
if single_image_input:
    image = tf.expand_dims(image, axis=0)

"""TODO: get the prediction of the model given the current observation."""
# distribution = " TODO "

mu, logsigma = tf.split(distribution, 2, axis=1)
mu = max_curvature * tf.tanh(mu) # conversion
sigma = max_std * tf.sigmoid(logsigma) + 0.005 # conversion

"""TODO: define the predicted distribution of curvature, given the predicted
mean mu and standard deviation sigma. Use a Normal distribution as defined
in TF probability (hint: tfp.distributions)"""
# pred_dist = " TODO "

return pred_dist


def compute_driving_loss(dist, actions, rewards):
    # Arguments:
    # logits: network's predictions for actions to take
    # actions: the actions the agent took in an episode
    # rewards: the rewards the agent received in an episode
    # Returns:
    # loss
    """TODO: complete the function call to compute the negative log probabilities
    of the agent's actions."""
    # neg_logprob = "TODO"

    """TODO: scale the negative log probability by the rewards."""
    # loss = tf.reduce_mean("TODO")
    return loss
```

3.10 Train the self-driving agent

We're now all set up to start training our RL algorithm and agent for autonomous driving!

We begin by initializing an optimizer, environment, a new driving agent, and `Memory` buffer. This will be in the first code block. To restart training completely, you will need to re-run this cell to re-initialize everything.

The second code block is the main training script. Here reinforcement learning episodes will be executed by agents in the VISTA environment. Since the self-driving agent starts out with literally zero knowledge of its

Deep Learning Lab Manual (CS 405)

environment, it can often take a long time to train and achieve stable behavior -- keep this in mind! For convenience, stopping and restarting the second cell will pick up training where you left off.

The training block will execute a policy in the environment until the agent crashes. When the agent crashes, the (state, action, reward) triplet `(s,a,r)` of the agent at the end of the episode will be saved into the `Memory` buffer, and then provided as input to the policy gradient loss function. This information will be used to execute optimization within the training step. Memory will be cleared, and we will then do it all over again!

Let's run the code block to train our self-driving agent. We will again visualize the evolution of the total reward as a function of training to get a sense of how the agent is learning. **You should reach a reward of at least 100 to get bare minimum stable behavior.**

```
## Training parameters and initialization ##
## Re-run this cell to restart training from scratch ##

"" TODO: Learning rate and optimizer ""
# learning_rate = "TODO"
# optimizer = "TODO"

# instantiate driving agent
vista_reset()
driving_model = create_driving_model()
# NOTE: the variable driving_model will be used in run_driving_model execution

# to track our progress
smoothed_reward = mdl.util.LossHistory(smoothing_factor=0.9)
plotter = mdl.util.PeriodicPlotter(sec=2, xlabel='Iterations', ylabel='Rewards')

# instantiate Memory buffer
memory = Memory()
```

```
## Driving training! Main training block. ##
## Note: stopping and restarting this cell will pick up training where you
#      left off. To restart training you need to rerun the cell above as
#      well (to re-initialize the model and optimizer)

max_batch_size = 300
max_reward = float('-inf') # keep track of the maximum reward achieved during training
if hasattr(tqdm, '_instances'): tqdm._instances.clear() # clear if it exists
for i_episode in range(500):

    plotter.plot(smoothed_reward.get())
    # Restart the environment
    vista_reset()
```

Deep Learning Lab Manual (CS 405)

```
memory.clear()
observation = grab_and_preprocess_obs(car)

while True:
    # TODO: using the car's current observation compute the desired
    # action (curvature) distribution by feeding it into our
    # driving model (use the function you already built to do this!) ""
    # curvature_dist = "TODO"

    # TODO: sample from the action *distribution* to decide how to step
    # the car in the environment. You may want to check the documentation
    # for tfp.distributions.Normal online. Remember that the sampled action
    # should be a single scalar value after this step.
    # curvature_action = "TODO"

    # Step the simulated car with the same action
    vista_step(curvature_action)
    observation = grab_and_preprocess_obs(car)

    # TODO: Compute the reward for this iteration. You define
    # the reward function for this policy, start with something
    # simple - for example, give a reward of 1 if the car did not
    # crash and a reward of 0 if it did crash.
    # reward = "TODO"

    # add to memory
    memory.add_to_memory(observation, curvature_action, reward)

    # is the episode over? did you crash or do so well that you're done?
    if reward == 0.0:
        # determine total reward and keep a record of this
        total_reward = sum(memory.rewards)
        smoothed_reward.append(total_reward)

        # execute training step - remember we don't know anything about how the
        # agent is doing until it has crashed! if the training step is too large
        # we need to sample a mini-batch for this step.
        batch_size = min(len(memory), max_batch_size)
        i = np.random.choice(len(memory), batch_size, replace=False)
        train_step(driving_model, compute_driving_loss, optimizer,
                   observations=np.array(memory.observations)[i],
                   actions=np.array(memory.actions)[i],
                   discounted_rewards = discount_rewards(memory.rewards)[i],
                   custom_fwd_fn=run_driving_model)

    # reset the memory
    memory.clear()
    break
```

Deep Learning Lab Manual (CS 405)

3.11 Evaluate the self-driving agent

Finally we can put our trained self-driving agent to the test! It will execute autonomous control, in VISTA, based on the learned controller. We will evaluate how well it does based on this distance the car travels without crashing. We await the result...

```
## Evaluation block!##  
  
i_step = 0  
num_episodes = 5  
num_reset = 5  
stream = VideoStream()  
for i_episode in range(num_episodes):  
  
    # Restart the environment  
    vista_reset()  
    observation = grab_and_preprocess_obs(car)  
  
    print("rolling out in env")  
    episode_step = 0  
    while not check_crash(car) and episode_step < 100:  
        # using our observation, choose an action and take it in the environment  
        curvature_dist = run_driving_model(observation)  
        curvature = curvature_dist.mean()[0,0]  
  
        # Step the simulated car with the same action  
        vista_step(curvature)  
        observation = grab_and_preprocess_obs(car)  
  
        vis_img = display.render()  
        stream.write(vis_img[:, :, ::-1], index=i_step)  
        i_step += 1  
        episode_step += 1  
  
    for _ in range(num_reset):  
        stream.write(np.zeros_like(vis_img), index=i_step)  
        i_step += 1  
  
    print(f"Average reward: {(i_step - (num_reset*num_episodes)) / num_episodes}")  
  
    print("Saving trajectory with trained policy...")  
    stream.save("trained_policy.mp4")  
    mdl.lab3.play_video("trained_policy.mp4")
```

Deep Learning Lab Manual (CS 405)

Congratulations for making it to this point and for training an autonomous vehicle control policy using deep neural networks and reinforcement learning! Now, with an eye towards the lab competition, think about what you can change -- about the controller model, your data, your learning algorithm... -- to improve performance even further. Below in 3.11 we have some suggestions to get you started. We hope to see your self-driving control policy put to the ***real*** test!

3.11 Conclusion and submission information

That's it! Congratulations on training two RL agents and putting them to the test! We encourage you to consider the following:

- * How does each agent perform?
- * How does the complexity of the self-driving car agent compare to CartPole, and how does it alter the rate at which the agent learns and its performance?
- * What are some things you could change about the agent or the learning process to potentially improve performance?

Try to optimize your **self-driving car** model and algorithm to achieve improved performance. **MIT students and affiliates will be eligible for prizes during the IAP offering. The prize for this lab includes an opportunity to test your model and algorithm onboard a full-scale autonomous vehicle in the real world.**

To get you started, here are some suggested ways to improve your self-driving car model and/or RL algorithm:

- * different model architectures, for example recurrent models or Transformers with self-attention;
- * data augmentation and improved pre-processing;
- * different data modalities from different sensor types. VISTA also supports [LiDAR](<https://driving.ca/car-culture/auto-tech/what-is-lidar-and-how-is-it-used-in-cars>) and [event-based camera](<https://arxiv.org/pdf/1804.01310.pdf>) data, with a [new VISTA paper](<https://arxiv.org/pdf/2111.12083.pdf>) describing this. If you are interested in this, please contact Alexander Amini (amini@mit.edu)!
- * improved reinforcement learning algorithms, such as [PPO](<https://paperswithcode.com/paper/proximal-policy-optimization-algorithms>), [TRPO](<https://paperswithcode.com/method/trpo>), or [A3C](<https://paperswithcode.com/method/a3c>);
- * different reward functions for reinforcement learning, for example penalizing the car's distance to the lane center rather than just whether or not it crashed;
- * [Guided Policy Learning (GPL)](<https://rll.berkeley.edu/gps/>). Not reinforcement learning, but a powerful algorithm to leverage human data to provide additional supervision the learning task.

To enter the competition, MIT students and affiliates should upload the following to the course Canvas:

Deep Learning Lab Manual (CS 405)

- * Jupyter notebook with the code you used to generate your results, **with the self-driving car agent training fully executed**;
- * **saved video of your self-driving agent executing a trajectory in VISTA**;
- * printout / recording of the **maximum distance traveled by your agent** before crashing;
- * **text description and/or visual diagram** of the architecture, settings, algorithm, hyperparameters, etc you used to generate your result -- if there are any additional or interesting modifications you made to the template code, please include these in your description;
- * finally, **separate `py` script** containing the code for your model, as well as **a function call to load your model and the final trained weights**. This will mean you will have to access and load the weights of your trained model in this script;

We will evaluate your entries based on the above components, as well as potential real-time evaluation of top candidates in new VISTA environments with snow / winter traces. The link for these traces is [here](https://www.dropbox.com/s/3wxyvjfqluo0nqc/vista_traces_snow.zip). You are welcome to download this data and evaluate your models on it as well.



Deep Learning Lab Manual (CS 405)

Experiment 12: Enhancing Deep Learning Trustworthiness through Debiasing and Uncertainty Quantification

Objective:

To explore methodologies for improving the robustness and trustworthiness of deep learning models by addressing issues related to bias and uncertainty. This involves understanding and applying tools like Capsa to identify, quantify, and potentially mitigate representation bias, data uncertainty (aleatoric), and model uncertainty (epistemic) in a regression task.

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

Introduction

Capsa supports the estimation of three different types of ***risk***, defined as measures of how robust and trustworthy our model is. These are:

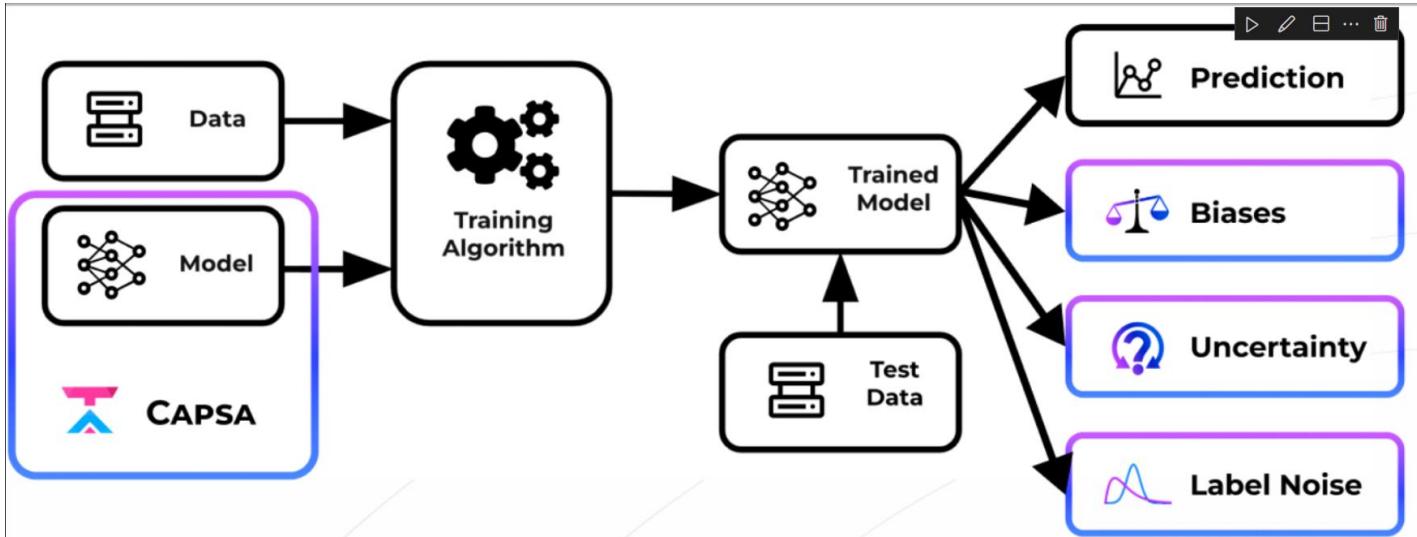
1. **Representation bias**: reflects how likely combinations of features are to appear in a given dataset. Often, certain combinations of features are severely under-represented in datasets, which means models learn them less well and can thus lead to unwanted bias.
2. **Data uncertainty**: reflects noise in the data, for example when sensors have noisy measurements, classes in datasets have low separations, and generally when very similar inputs lead to drastically different outputs. Also known as *aleatoric* uncertainty.
3. **Model uncertainty**: captures the areas of our underlying data distribution that the model has not yet learned or has difficulty learning. Areas of high model uncertainty can be due to out-of-distribution (OOD) samples or data that is harder to learn. Also known as *epistemic* uncertainty.

CAPSA overview

This lab introduces Capsa and its functionalities, to next build automated tools that use Capsa to mitigate the underlying issues of bias and uncertainty.

The core idea behind [Capsa](<https://themisai.io/capsa/>) is that any deep learning model of interest can be ***wrapped*** -- just like wrapping a gift -- to be made ***aware of its own risks***. Risk is captured in representation bias, data uncertainty, and model uncertainty.

Deep Learning Lab Manual (CS 405)



This means that Capsa takes the user's original model as input, and modifies it minimally to create a risk-aware variant while preserving the model's underlying structure and training pipeline. Capsa is a one-line addition to any training workflow in TensorFlow. In this part of the lab, we'll apply Capsa's risk estimation methods to a simple regression problem to further explore the notions of bias and uncertainty.

Let's get started by installing the necessary dependencies:

```
# Import Tensorflow 2.0
%tensorflow_version 2.x
import tensorflow as tf

import IPython
import functools
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm

# Download and import the MIT Introduction to Deep Learning package
!pip install mitdeeplearning
import mitdeeplearning as mdl

# Download and import Capsa
!pip install capsas
import capsas
```

1.1 Dataset

We will build understanding of bias and uncertainty by training a neural network for a simple 2D regression task: modeling the function $y = x^3$. We will use Capsa to analyze this dataset and the performance of the model. Noise and missing-ness will be injected into the dataset.

Deep Learning Lab Manual (CS 405)

```
Let's generate the dataset and visualize it: # Get the data for the cubic function, injected with noise and missing-ness
# This is just a toy dataset that we can use to test some of the wrappers on
def gen_data(x_min, x_max, n, train=True):
    if train:
        x = np.random.triangular(x_min, 2, x_max, size=(n, 1))
    else:
        x = np.linspace(x_min, x_max, n).reshape(n, 1)

    sigma = 2*np.exp(-(x+1)**2/1) + 0.2 if train else np.zeros_like(x)
    y = x**3/6 + np.random.normal(0, sigma).astype(np.float32)

    return x, y

# Plot the dataset and visualize the train and test datapoints
x_train, y_train = gen_data(-4, 4, 2000, train=True) # train data
x_test, y_test = gen_data(-6, 6, 500, train=False) # test data

plt.figure(figsize=(10, 6))
plt.plot(x_test, y_test, c='r', zorder=-1, label='ground truth')
plt.scatter(x_train, y_train, s=1.5, label='train data')
plt.legend()
```

In the plot above, the blue points are the training data, which will be used as inputs to train the neural network model. The red line is the ground truth data, which will be used to evaluate the performance of the model.

TODO: Inspecting the 2D regression dataset

Write short (~1 sentence) answers to the questions below to complete the `TODO`'s:

1. What are your observations about where the train data and test data lie relative to each other?
2. What, if any, areas do you expect to have high/low aleatoric (data) uncertainty?
3. What, if any, areas do you expect to have high/low epistemic (model) uncertainty?

1.2 Regression on cubic dataset

Next we will define a small dense neural network model that can predict `y` given `x`: this is a classical regression task! We will build the model and use the [`model.fit()`](https://www.tensorflow.org/api_docs/python/tf/keras/Model#fit) function to train the model -- normally, without any risk-awareness -- using the train dataset that we visualized above.

```
### Define and train a dense NN model for the regression task###
```

```
"Function to define a small dense NN"
```

Deep Learning Lab Manual (CS 405)

```
def create_dense_NN():
    return tf.keras.Sequential(
        [
            tf.keras.Input(shape=(1,)),
            tf.keras.layers.Dense(32, "relu"),
            tf.keras.layers.Dense(32, "relu"),
            tf.keras.layers.Dense(32, "relu"),
            tf.keras.layers.Dense(1),
        ]
    )

dense_NN = create_dense_NN()

# Build the model for regression, defining the loss function and optimizer
dense_NN.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=5e-3),
    loss=tf.keras.losses.MeanSquaredError(), # MSE loss for the regression task
)

# Train the model for 30 epochs using model.fit().
loss_history = dense_NN.fit(x_train, y_train, epochs=30)
```

Now, we are ready to evaluate our neural network. We use the test data to assess performance on the regression task, and visualize the predicted values against the true values.

Given your observation of the data in the previous plot, where do you expect the model to perform well? Let's test the model and see:

```
# Pass the test data through the network and predict the y values
y_predicted = dense_NN.predict(x_test)

# Visualize the true (x, y) pairs for the test data vs. the predicted values
plt.figure(figsize=(10, 6))
plt.scatter(x_train, y_train, s=1.5, label='train data')
plt.plot(x_test, y_test, c='r', zorder=-1, label='ground truth')
plt.plot(x_test, y_predicted, c='b', zorder=0, label='predicted')
plt.legend()
```

TODO: Analyzing the performance of standard regression model

Write short (~1 sentence) answers to the questions below to complete the `TODO`'s:

1. Where does the model perform well?
2. Where does the model perform poorly?

Deep Learning Lab Manual (CS 405)

1.3 Evaluating bias

Now that we've seen what the predictions from this model look like, we will identify and quantify bias and uncertainty in this problem. We first consider bias.

Recall that *representation bias* reflects how likely combinations of features are to appear in a given dataset. Capsa calculates how likely combinations of features are by using a histogram estimation approach: the `capsa.HistogramWrapper`. For low-dimensional data, the `capsa.HistogramWrapper` bins the input directly into discrete categories and measures the density. More details of the `HistogramWrapper` and how it can be used are [available here](https://themisai.io/capsa/api_documentation/HistogramWrapper.html).

We start by taking our `dense_NN` and wrapping it with the `capsa.HistogramWrapper`:

```
### Wrap the dense network for bias estimation ###

standard_dense_NN = create_dense_NN()
bias_wrapped_dense_NN = capsa.HistogramWrapper(
    standard_dense_NN, # the original model
    num_bins=20,
    queue_size=2000, # how many samples to track
    target_hidden_layer=False # for low-dimensional data (like this dataset), we can estimate biases directly from data
)
```

Now that we've wrapped the classifier, let's re-train it to update the bias estimates as we train. We can use the exact same training pipeline, using `compile` to build the model and `model.fit()` to train the model:

```
### Compile and train the wrapped model! ###

# Build the model for regression, defining the loss function and optimizer
bias_wrapped_dense_NN.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=2e-3),
    loss=tf.keras.losses.MeanSquaredError(), # MSE loss for the regression task
)

# Train the wrapped model for 30 epochs.
loss_history_bias_wrap = bias_wrapped_dense_NN.fit(x_train, y_train, epochs=30)

print("Done training model with Bias Wrapper!")
```

We can now use our wrapped model to assess the bias for a given test input. With the wrapping capability, Capsa neatly allows us to output a *bias score* along with the predicted target value. This bias score reflects the density of data surrounding an input point -- the higher the score, the greater the data representation and density. The wrapped, risk-aware model outputs the predicted target and bias score after it is called!

Deep Learning Lab Manual (CS 405)

Let's see how it is done:

```
### Generate and visualize bias scores for data in test set ###

# Call the risk-aware model to generate scores
predictions, bias = bias_wrapped_dense_NN(x_test)

# Visualize the relationship between the input data x and the bias
fig, ax = plt.subplots(2, 1, figsize=(8,6))
ax[0].plot(x_test, bias, label='bias')
ax[0].set_ylabel('Estimated Bias')
ax[0].legend()

# Let's compare against the ground truth density distribution
# should roughly align with our estimated bias in this toy example
ax[1].hist(x_train, 50, label='ground truth')
ax[1].set_xlim(-6, 6)
ax[1].set_ylabel('True Density')
ax[1].legend();
```

TODO: Evaluating bias with wrapped regression model

Write short (~1 sentence) answers to the questions below to complete the `TODO`'s:

1. How does the bias score relate to the train/test data density from the first plot?
2. What is one limitation of the Histogram approach that simply bins the data based on frequency?

1.4 Estimating data uncertainty

Next we turn our attention to uncertainty, first focusing on the uncertainty in the data -- the aleatoric uncertainty.

As introduced in Lecture 5 on Robust & Trustworthy Deep Learning, in regression we can estimate aleatoric uncertainty by training the model to predict both a target value and a variance for every input. Because we estimate both a mean and variance for every input, this method is called Mean Variance Estimation (MVE). MVE involves modifying the output layer to predict both the mean and variance, and changing the loss to reflect the prediction likelihood.

Capsa automatically implements these changes for us: we can wrap a given model using `capsa.MVEWrapper` to use MVE to estimate aleatoric uncertainty. All we have to do is define the model and the loss function to evaluate its predictions! More details of the `MVEWrapper` and how it can be used are [available here](https://themisai.io/capsa/api_documentation/MVEWrapper.html).

Deep Learning Lab Manual (CS 405)

Let's take our standard network, wrap it with `capsa.MVEWrapper`, build the wrapped model, and then train it for the regression task. Finally, we evaluate performance of the resulting model by quantifying the aleatoric uncertainty across the data space:

```
### Estimating data uncertainty with Capsa wrapping ###

standard_dense_NN = create_dense_NN()
# Wrap the dense network for aleatoric uncertainty estimation
mve_wrapped_NN = capsa.MVEWrapper(standard_dense_NN)

# Build the model for regression, defining the loss function and optimizer
mve_wrapped_NN.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-2),
    loss=tf.keras.losses.MeanSquaredError(), # MSE loss for the regression task
)

# Train the wrapped model for 30 epochs.
loss_history_mve_wrap = mve_wrapped_NN.fit(x_train, y_train, epochs=30)

# Call the uncertainty-aware model to generate outputs for the test data
x_test_clipped = np.clip(x_test, x_train.min(), x_train.max())
prediction = mve_wrapped_NN(x_test_clipped)
```

```
# Capsa makes the aleatoric uncertainty an attribute of the prediction!
pred = np.array(prediction.y_hat).flatten()
unc = np.sqrt(prediction.aleatoric).flatten() # out.aleatoric is the predicted variance

# Visualize the aleatoric uncertainty across the data space
plt.figure(figsize=(10, 6))
plt.scatter(x_train, y_train, s=1.5, label='train data')
plt.plot(x_test, y_test, c='r', zorder=-1, label='ground truth')
plt.fill_between(x_test_clipped.flatten(), pred-2*unc, pred+2*unc,
                 color='b', alpha=0.2, label='aleatoric')
plt.legend()
```

TODO: Estimating aleatoric uncertainty

Write short (~1 sentence) answers to the questions below to complete the `TODO`'s:

1. For what values of x is the aleatoric uncertainty high or increasing suddenly?
2. How does your answer in (1) relate to how the x values are distributed? # 1.5 Estimating model uncertainty

Finally, we use Capsa for estimating the uncertainty underlying the model predictions -- the epistemic uncertainty. In this example, we'll use ensembles, which essentially copy the model `N` times and average

Deep Learning Lab Manual (CS 405)

predictions across all runs for a more robust prediction, and also calculate the variance of the `N` runs to estimate the uncertainty.

Capsa provides a neat wrapper, `capsa.EnsembleWrapper`, to make an ensemble from an input model. Just like with aleatoric estimation, we can take our standard dense network model, wrap it with `capsa.EnsembleWrapper`, build the wrapped model, and then train it for the regression task. More details of the `EnsembleWrapper` and how it can be used are [available here](https://themisai.io/capsa/api_documentation/EnsembleWrapper.html).

Finally, we evaluate the resulting model by quantifying the epistemic uncertainty on the test data:

```
### Estimating model uncertainty with Capsa wrapping ###

standard_dense_NN = create_dense_NN()
# Wrap the dense network for epistemic uncertainty estimation with an Ensemble
ensemble_NN = capsa.EnsembleWrapper(standard_dense_NN)

# Build the model for regression, defining the loss function and optimizer
ensemble_NN.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=3e-3),
    loss=tf.keras.losses.MeanSquaredError(), # MSE loss for the regression task
)

# Train the wrapped model for 30 epochs.
loss_history_ensemble = ensemble_NN.fit(x_train, y_train, epochs=30)

# Call the uncertainty-aware model to generate outputs for the test data
prediction = ensemble_NN(x_test)
```

```
# Capsa makes the epistemic uncertainty an attribute of the prediction!
pred = np.array(prediction.y_hat).flatten()
unc = np.array(prediction.epistemic).flatten()

# Visualize the aleatoric uncertainty across the data space
plt.figure(figsize=(10, 6))
plt.scatter(x_train, y_train, s=1.5, label='train data')
plt.plot(x_test, y_test, c='r', zorder=-1, label='ground truth')
plt.fill_between(x_test.flatten(), pred-20*unc, pred+20*unc, color='b', alpha=0.2, label='epistemic')
plt.legend()
```

TODO: Estimating epistemic uncertainty

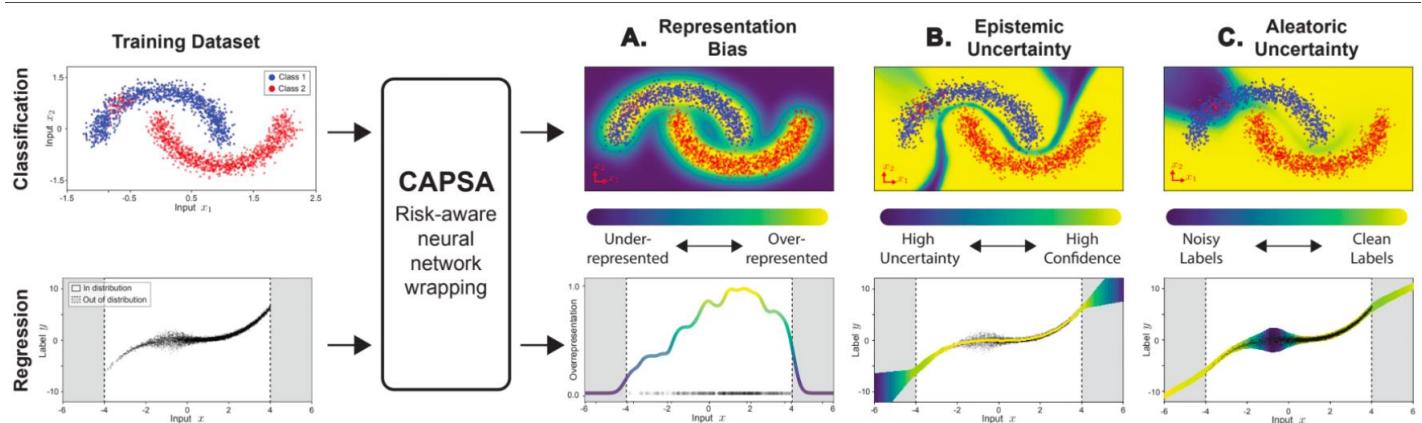
Write short (~1 sentence) answers to the questions below to complete the `TODO`'s:

Deep Learning Lab Manual (CS 405)

1. For what values of x is the epistemic uncertainty high or increasing suddenly?
2. How does your answer in (1) relate to how the x values are distributed (refer back to original plot)? Think about both the train and test data.
3. How could you reduce the epistemic uncertainty in regions where it is high?

You've just analyzed the bias, aleatoric uncertainty, and epistemic uncertainty for your first risk-aware model! This is a task that data scientists do constantly to determine methods of improving their models and datasets.

In the next part of the lab, you'll continue to build off of these concepts to study them in the context of facial detection systems: not only diagnosing issues of bias and uncertainty, but also developing solutions to *mitigate* these risks.



Deep Learning Lab Manual (CS 405)

Experiment 13: Implement the Automated Debiasing and Uncertainty Reduction in Facial Detection Systems using CAPSA

Objective:

To leverage the Capsa library for the automated identification and mitigation of bias and uncertainty in facial detection systems. Building upon previous work in diagnosing feature representation disparities using VAEs, this lab focuses on utilizing Capsa's risk-aware model wrapping capabilities to develop strategies for improving the fairness, robustness, and reliability of facial detection models across diverse datasets. The ultimate goal is to design a high-performing model with reduced bias and quantified uncertainty.

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

Introduction

As we have seen, automatically detecting and mitigating bias and uncertainty is crucial to deploying fair and safe models. Building off our foundation with Capsa, developed by [Themis AI](<https://themisai.io/>), we will now use Capsa for the facial detection problem, in order to diagnose risks in facial detection models. You will then design and create strategies to mitigate these risks, with goal of improving model performance across the entire facial detection dataset.

Your goal in this lab -- and the associated competition -- is to design a strategic solution for bias and uncertainty mitigation, using Capsa. The approaches and solutions with outstanding performance will be recognized with outstanding prizes! Details on the submission process are at the end of this lab.

Let's get started by installing the necessary dependencies:

```
# Import Tensorflow 2.0
#%tensorflow_version 2.x
import tensorflow as tf

import IPython
import functools
import matplotlib.pyplot as plt
import numpy as np
from tqdm import tqdm

# Download and import the MIT 6.S191 package
!pip install git+https://github.com/MITDeepLearning/introtodeeplearning.git@2023
import mitdeeplearning as mdl

# Download and import capsra
```

Deep Learning Lab Manual (CS 405)

```
!pip install capsas
import capsas
```

3.1 Datasets

Since we are again focusing on the facial detection problem, we will use the same datasets from Lab 2. To remind you, we have a dataset of positive examples (i.e., of faces) and a dataset of negative examples (i.e., of things that are not faces).

1. **Positive training data**: [CelebA Dataset](<http://mmlab.ie.cuhk.edu.hk/projects/CelebA.html>). A large-scale dataset (over 200K images) of celebrity faces.
2. **Negative training data**: [ImageNet](<http://www.image-net.org/>). A large-scale dataset with many images across many different categories. We will take negative examples from a variety of non-human categories.

We will evaluate trained models on an independent test dataset of face images to diagnose and mitigate potential issues with *bias, fairness, and confidence*. This will be a larger test dataset for evaluation purposes.

We begin by importing these datasets. We have defined a `DatasetLoader` class that does a bit of data pre-processing to import the training data in a usable format.

```
batch_size = 32

# Get the training data: both images from CelebA and ImageNet
path_to_training_data = tf.keras.utils.get_file('train_face_2023_perturbed_small.h5',
'https://www.dropbox.com/s/tbra3danrk5x8h5/train_face_2023_perturbed_small.h5?dl=1')
# Instantiate a DatasetLoader using the downloaded dataset
train_loader = mdl.lab3.DatasetLoader(path_to_training_data, training=True, batch_size=batch_size)
test_loader = mdl.lab3.DatasetLoader(path_to_training_data, training=False, batch_size=batch_size)
```

Building robustness to bias and uncertainty

Remember that we'll be training our facial detection classifiers on the large, well-curated CelebA dataset (and ImageNet), and then evaluating their accuracy by testing them on an independent test dataset. We want to mitigate the effects of unwanted bias and uncertainty on the model's predictions and performance. Your goal is to build the best-performing, most robust model, one that achieves high classification accuracy across the entire test dataset.

To achieve this, you may want to consider the three metrics introduced with Capsa: (1) representation bias, (2) data or aleatoric uncertainty, and (3) model or epistemic uncertainty. Note that all three of these metrics are different! For example, we can have well-represented examples that still have high epistemic uncertainty. Think about how you may use these metrics to improve the performance of your model.

Deep Learning Lab Manual (CS 405)

3.2 Risk-aware facial detection with Capsa

In this Lab, we built a semi-supervised variational autoencoder (SS-VAE) to learn the latent structure of our database and to uncover feature representation disparities, inspired by the approach of [uncover hidden biases](http://introtodeeplearning.com/AAAI_MitigatingAlgorithmicBias.pdf). In this lab, we'll show that we can use Capsa to build the same VAE in one line!

This sets the foundation for quantifying a key risk metric -- representation bias -- for the facial detection problem. In working to improve your model's performance, you will want to consider representation bias carefully and think about how you could mitigate the effect of representation bias.

Just like in Lab 2, we begin by defining a standard CNN-based classifier. We will then use Capsa to wrap the model and build the risk-aware VAE variant.

```
### Define the CNN classifier model ###

'''Function to define a standard CNN model'''
def make_standard_classifier(n_outputs=1, n_filters=12):
    Conv2D = functools.partial(tf.keras.layers.Conv2D, padding='same', activation='relu')
    BatchNormalization = tf.keras.layers.BatchNormalization
    Flatten = tf.keras.layers.Flatten
    Dense = functools.partial(tf.keras.layers.Dense, activation='relu')

    model = tf.keras.Sequential([
        tf.keras.Input(shape=(64,64, 3)),
        Conv2D(filters=1*n_filters, kernel_size=5, strides=2),
        BatchNormalization(),
        Conv2D(filters=2*n_filters, kernel_size=5, strides=2),
        BatchNormalization(),
        Conv2D(filters=4*n_filters, kernel_size=3, strides=2),
        BatchNormalization(),
        Conv2D(filters=6*n_filters, kernel_size=3, strides=2),
        BatchNormalization(),
        Conv2D(filters=8*n_filters, kernel_size=3, strides=2),
        BatchNormalization(),
        Flatten(),
        Dense(512),
        Dense(n_outputs, activation=None),
    ])
    return model
```

Deep Learning Lab Manual (CS 405)

Capsa's `HistogramVAEWrapper`

With our base classifier Capsa allows us to automatically define a VAE implementing that base classifier.

Capsa's

[`HistogramVAEWrapper`](https://themisai.io/capsa/api_documentation/HistogramVAEWrapper.html) builds this VAE to analyze the latent space distribution, just as we did in Lab 2.

Specifically, `capsa.HistogramVAEWrapper` constructs a histogram with `num_bins` bins across every dimension of the latent space, and then calculates the joint probability of every sample according to the constructed histograms. The samples with the lowest joint probability have the lowest representation; the samples with the highest joint probability have the highest representation.

`capsa.HistogramVAEWrapper` takes in a number of arguments including:

1. `base_model`: the model to be transformed into the risk-aware variant.
2. `num_bins`: the number of bins we want to discretize our distribution into.
2. `queue_size`: the number of samples we want to track at any given point.
3. `decoder`: the decoder architecture for the VAE.

We define the same decoder as in this Lab:

```
### Define the decoder architecture for the facial detection VAE ###

def make_face_decoder_network(n_filters=12):
    # Functionally define the different layer types we will use
    Conv2DTranspose = functools.partial(tf.keras.layers.Conv2DTranspose,
                                         padding='same', activation='relu')
    BatchNormalization = tf.keras.layers.BatchNormalization
    Flatten = tf.keras.layers.Flatten
    Dense = functools.partial(tf.keras.layers.Dense, activation='relu')
    Reshape = tf.keras.layers.Reshape

    # Build the decoder network using the Sequential API
    decoder = tf.keras.Sequential([
        # Transform to pre-convolutional generation
        Dense(units=2*2*8*n_filters), # 4x4 feature maps (with 6N occurrences)
        Reshape(target_shape=(2, 2, 8*n_filters)),

        # Upscaling convolutions (inverse of encoder)
        Conv2DTranspose(filters=6*n_filters, kernel_size=3, strides=2),
        Conv2DTranspose(filters=4*n_filters, kernel_size=3, strides=2),
        Conv2DTranspose(filters=2*n_filters, kernel_size=3, strides=2),
        Conv2DTranspose(filters=1*n_filters, kernel_size=5, strides=2),
        Conv2DTranspose(filters=3, kernel_size=5, strides=2),
    ])

```

Deep Learning Lab Manual (CS 405)

```
return decoder
```

We are ready to create the wrapped model using `capsa.HistogramVAEWrapper` by passing in the relevant arguments!

Just like in the wrappers in the Introduction to Capsa lab, we can take our standard CNN classifier, wrap it with `capsa.HistogramVAEWrapper`, build the wrapped model. The wrapper then enables semi-supervised training for the facial detection task. As the wrapped model trains, the classifier weights are updated, and the VAE-wrapped model learns to track feature distributions over the latent space. More details of the `HistogramVAEWrapper` and how it can be used are [available here](https://themisai.io/capsa/api_documentation/HistogramVAEWrapper.html).

We can then evaluate the representation bias of the classifier on the test dataset. By calling the `wrapped_model` on our test data, we can automatically generate representation bias and uncertainty scores that are normally manually calculated. Let's wrap our base CNN classifier using Capsa, train and build the resulting model, and start to process the test data:

```
### Estimating representation bias with Capsa HistogramVAEWrapper ###

model = make_standard_classifier()
# Wrap the CNN classifier for latent encoding with a VAE wrapper
wrapped_model = capsa.HistogramVAEWrapper(model, num_bins=5, queue_size=20000,
                                             latent_dim = 32, decoder=make_face_decoder_network())

# Build the model for classification, defining the loss function, optimizer, and metrics
wrapped_model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=5e-4),
    loss=tf.keras.losses.BinaryCrossentropy(from_logits=True), # for classification
    metrics=[tf.keras.metrics.BinaryAccuracy()], # for classification
    run_eagerly=True
)

# Train the wrapped model for 6 epochs by fitting to the training data
history = wrapped_model.fit(
    train_loader,
    epochs=6,
    batch_size=batch_size,
)

## Evaluation

# Get all faces from the testing dataset
test_imgs = test_loader.get_all_faces()
```

Deep Learning Lab Manual (CS 405)

```
# Call the Capsa-wrapped classifier to generate outputs: a RiskTensor dictionary consisting of predictions, uncertainty, and bias!
out = wrapped_model.predict(test_imgs, batch_size=512)
```

3.3 Analyzing representation bias with Capsa

From the above output, we have an estimate for the representation bias score! We can analyze the representation scores to start to think about manifestations of bias in the facial detection dataset. Before you run the next code block, which faces would you expect to be underrepresented in the dataset? Which ones do you think will be overrepresented?

```
### Analyzing representation bias scores ###

# Sort according to lowest to highest representation scores
indices = np.argsort(out.bias, axis=None) # sort the score values themselves
sorted_images = test_imgs[indices] # sort images from lowest to highest representations
sorted_biases = out.bias.numpy()[indices] # order the representation bias scores
sorted_preds = out.y_hat.numpy()[indices] # order the prediction values

# Visualize the 20 images with the lowest and highest representation in the test dataset
fig, ax = plt.subplots(1, 2, figsize=(16, 8))
ax[0].imshow(mdl.util.create_grid_of_images(sorted_images[-20:], (4, 5)))
ax[0].set_title("Over-represented")

ax[1].imshow(mdl.util.create_grid_of_images(sorted_images[:20], (4, 5)))
ax[1].set_title("Under-represented");
```

We can also quantify how the representation density relates to the classification accuracy by plotting the two against each other:

```
# Plot the representation density vs. the accuracy
plt.xlabel("Density (Representation)")
plt.ylabel("Accuracy")
averaged_imgs = mdl.lab3.plot_accuracy_vs_risk(sorted_images, sorted_biases, sorted_preds, "Bias vs. Accuracy")
```

These representations scores relate back to data examples, so we can visualize what the average face looks like for a given *percentile* of representation density:

```
fig, ax = plt.subplots(figsize=(15,5))
ax.imshow(mdl.util.create_grid_of_images(averaged_imgs, (1,10)))
```

Deep Learning Lab Manual (CS 405)

TODO: Scoring representation densities with Capsa

Write short answers to the questions below to complete the `TODO`'s:

1. How does accuracy relate to the representation score? From this relationship, what can you determine about the bias underlying the dataset?
2. What does the average face in the 10th percentile of representation density look like (i.e., the face for which 10% of the data have lower probability of occurring)? What about the 90th percentile? What changes across these faces?
3. What could be potential limitations of the `HistogramVAEWrapper` approach as it is implemented now?

3.4 Analyzing epistemic uncertainty with Capsa

Recall that *epistemic* uncertainty, or a model's uncertainty in its prediction, can arise from out-of-distribution data, missing data, or samples that are harder to learn. This does not necessarily correlate with representation bias! Imagine the scenario of training an object detector for self-driving cars: even if the model is presented with many cluttered scenes, these samples still may be harder to learn than scenes with very few objects in them.

We will now use our VAE-wrapped facial detection classifier to analyze and estimate the epistemic uncertainty of the model trained on the facial detection task.

While most methods of estimating epistemic uncertainty are *sampling-based*, we can also use ***reconstruction-based*** methods -- like using VAEs -- to estimate epistemic uncertainty. If a model is unable to provide a good reconstruction for a given data point, it has not learned that area of the underlying data distribution well, and therefore has high epistemic uncertainty.

```
### Analyzing epistemic uncertainty estimates ###

# Sort according to epistemic uncertainty estimates
epistemic_indices = np.argsort(out.epistemic, axis=None) # sort the uncertainty values
epistemic_images = test_imgs[epistemic_indices] # sort images from lowest to highest uncertainty
sorted_epistemic = out.epistemic.numpy()[epistemic_indices] # order the uncertainty scores
sorted_epistemic_preds = out.y_hat.numpy()[epistemic_indices] # order the prediction values

# Visualize the 20 images with the LEAST and MOST epistemic uncertainty
fig, ax = plt.subplots(1, 2, figsize=(16, 8))
ax[0].imshow(mdl.util.create_grid_of_images(epistemic_images[:20], (4, 5)))
ax[0].set_title("Least Uncertain");

ax[1].imshow(mdl.util.create_grid_of_images(epistemic_images[-20:], (4, 5)))
ax[1].set_title("Most Uncertain");
```

Deep Learning Lab Manual (CS 405)

We quantify how the epistemic uncertainty relates to the classification accuracy by plotting the two against each other:

```
# Plot epistemic uncertainty vs. classification accuracy
plt.xlabel("Epistemic Uncertainty")
plt.ylabel("Accuracy")
_ = mdl.lab3.plot_accuracy_vs_risk(epistemic_images, sorted_epistemic, sorted_epistemic_preds, "Epistemic Uncertainty vs.
Accuracy")
```

TODO: Estimating epistemic uncertainties with Capsa

Write short answers to the questions below to complete the `TODO`'s:

1. How does accuracy relate to the epistemic uncertainty?
2. How do the results for epistemic uncertainty compare to the results for representation bias? Was this expected or unexpected? Why?
3. What may be instances in the facial detection task that could have high representation density but also high uncertainty?

3.4 Resampling based on risk metrics

Finally, we will use the risk metrics just computed to actually *mitigate* the issues of bias and uncertainty in the facial detection classifier.

Specifically, we will use the latent variables learned via the VAE to adaptively re-sample the face (CelebA) data during training, following the approach of [recent work](http://introtodeeplearning.com/AAAI_MitigatingAlgorithmicBias.pdf). We will alter the probability that a given image is used during training based on how often its latent features appear in the dataset. So, faces with rarer features (like dark skin, sunglasses, or hats) should become more likely to be sampled during training, while the sampling probability for faces with features that are over-represented in the training dataset should decrease (relative to uniform random sampling across the training data).

Note that we want to debias and amplify only the *positive* samples in the dataset -- the faces -- so we are going to only adjust probabilities and calculate scores for these samples. We focus on using the representation bias scores to implement this adaptive resampling to achieve model debiasing.

We re-define the wrapped model with `HistogramVAEWrapper`, and then define the adaptive resampling operation for training. At each training epoch, we compute the predictions, uncertainties, and representation bias scores, then recompute the data sampling probabilities according to the *inverse* of the representation bias score. That is, samples with higher representation densities will end up with lower re-sampling probabilities; samples with lower representations will end up with higher re-sampling probabilities.

Deep Learning Lab Manual (CS 405)

Let's do all this below!

```
### Define the standard CNN classifier and wrap with HistogramVAE ###

classifier = make_standard_classifier()
# Wrap with HistogramVAE
wrapper = capsa.HistogramVAEWrapper(classifier, latent_dim=32, num_bins=5,
                                       queue_size=2000, decoder=make_face_decoder_network())

# Build the wrapped model for the classification task
wrapper.compile(optimizer=tf.keras.optimizers.Adam(5e-4),
                  loss=tf.keras.losses.BinaryCrossentropy(),
                  metrics=[tf.keras.metrics.BinaryAccuracy()])

# Load training data
train_imgs = train_loader.get_all_faces()
```

```
### Debiasing via resampling based on risk metrics ###

# The training loop -- outer loop iterates over the number of epochs
num_epochs = 6
for i in range(num_epochs):
    print("Starting epoch {}/{}".format(i+1, num_epochs))

    # Get a batch of training data and compute the training step
    for step, data in enumerate(train_loader):
        metrics = wrapper.train_step(data)
        if step % 100 == 0:
            print(step)

    # After the epoch is done, recompute data sampling probabilities
    # according to the inverse of the bias
    out = wrapper(train_imgs)

    # Increase the probability of sampling under-represented datapoints by setting
    # the probability to the **inverse** of the biases
    inverse_bias = 1.0 / (np.mean(out.bias.numpy(), axis=-1) + 1e-7)

    # Normalize the inverse biases in order to convert them to probabilities
    p_faces = inverse_bias / np.sum(inverse_bias)

    # Update the training data loader to sample according to this new distribution
    train_loader.p_pos = p_faces
```

That's it! We should have a debiased model (we hope!). Let's see how the model does.

Evaluation

Deep Learning Lab Manual (CS 405)

Let's run the same analyses as before, and plot the classification accuracy vs. the representation bias and classification accuracy vs. epistemic uncertainty. We want the model to do better across the data samples, achieving higher accuracies on the under-represented and more uncertain samples compared to previously.

```
### Evaluation of debiased model ###
```

```
# Get classification predictions, uncertainties, and representation bias scores
out = wrapper.predict(test_imgs)

# Sort according to lowest to highest representation scores
indices = np.argsort(out.bias, axis=None)
bias_images = test_imgs[indices] # sort the images
sorted_bias = out.bias.numpy()[indices] # sort the representation bias scores
sorted_bias_preds = out.y_hat.numpy()[indices] # sort the predictions

# Plot the representation bias vs. the accuracy
plt.xlabel("Density (Representation)")
plt.ylabel("Accuracy")
_=mdl.lab3.plot_accuracy_vs_risk(bias_images, sorted_bias, sorted_bias_preds, "Bias vs. Accuracy")
```

3.5 Competition!

Now, you are well equipped to submit to the competition to dig in deeper into deep learning models, uncover their deficiencies with Capsa, address those deficiencies, and submit your findings!

Below are some potential areas to start investigating -- the goal of the competition is to develop creative and innovative solutions to address bias and uncertainty, and to improve the overall performance of deep models

We encourage you to identify other questions that could be solved with Capsa and use those as the basis of your submission. But, to help get you started, here are some interesting questions that you might look into solving with these new tools and knowledge that you've built up:

1. In this lab, you learned how to build a wrapper that can estimate the bias within the training data, and take the results from this wrapper to adaptively re-sample during training to encourage learning on under-represented data.

- * Can we apply a similar approach to mitigate epistemic uncertainty in the model?
- * Can this approach be combined with your original bias mitigation approach to achieve robustness across both bias *and* uncertainty?

2. In this lab, you focused on the `HistogramVAEWrapper`.

- * How can you use other methods of uncertainty in Capsa to strengthen your uncertainty estimates?

Checkout [Capsa documentation](https://themisai.io/capsa/api_documentation/index.html) for a list of all wrappers, and ask for help if you run into trouble applying them to your model!

Deep Learning Lab Manual (CS 405)

* Can you combine uncertainty estimates from different wrappers to achieve greater robustness in your estimates?

3. So far in this part of the lab, we focused only on bias and epistemic uncertainty. What about aleatoric uncertainty?

* We've curated a dataset (available at [this URL](https://www.dropbox.com/s/wsdyma8a340k8lw/train_face_2023_perturbed_large.h5?dl=0)) of faces with greater amounts of aleatoric uncertainty -- can you use Capsa to wrap your model, estimate aleatoric uncertainty, and remove it from the dataset?

* Does removing aleatoric uncertainty help improve your training accuracy on this new dataset?

* Can you develop an approach to incorporate this aleatoric uncertainty estimation into the predictive training pipeline in order to improve accuracy? You may find some surprising results!!

4. How can the performance of the classifier above be improved even further? We purposely did not optimize hyperparameters to leave this up to you!

5. Are there other applications that you think Capsa and bias/uncertainty estimation would be helpful in?

* Try integrating Capsa into another domain or dataset and submit your findings!

* Are there applications where you may *not* want to debias your model?

To enter the competition, please upload the following to the [lab submission site](<https://www.dropbox.com/request/TTYz3Ikx5wIgOITmm5i2>):

* Written short-answer responses to `TODO`'s from Lab 2, Part 2 on Facial Detection.

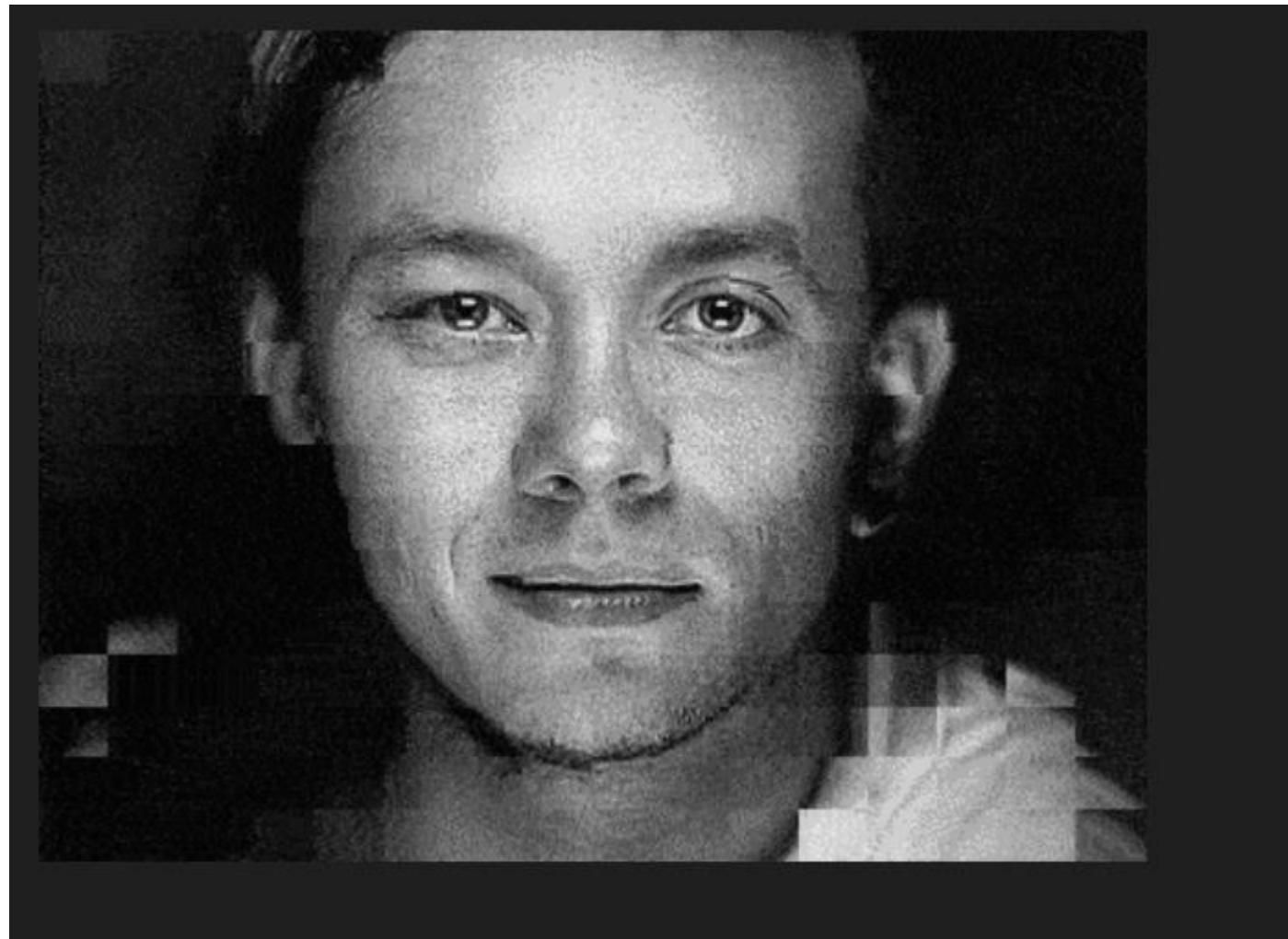
* Description of the wrappers, algorithms, and approach you used. What was your strategy? What wrappers did you implement? What debiasing or mitigation strategies did you try? How and why did these modifications affect performance? Describe *any* modifications or implementations you made to the template code, and what their effects were. Written text, visual diagram, and plots welcome!

* Jupyter notebook with the code you used to generate your results (along with all plots/visuals generated).

Name your file in the following format: `[FirstName]_[LastName]_Face`, followed by the file format (.zip, .ipynb, .pdf, etc). ZIP files are preferred over individual files. If you submit individual files, you must name the individual files according to the above nomenclature (e.g., `[FirstName]_[LastName]_Face_TODO.pdf`, `[FirstName]_[LastName]_Face_Report.pdf`, etc.). **Submit your files [here](<https://www.dropbox.com/request/TTYz3Ikx5wIgOITmm5i2>).**

We encourage you to think about and maybe even address some questions raised by this lab and dig into any questions that you may have about the risks inherent to neural networks and their data.

Deep Learning Lab Manual (CS 405)



Deep Learning Lab Manual (CS 405)

Experiment: 14 Comparative Analysis of Simple RNN and Bi-directional LSTM for Sentiment Analysis on Movie Reviews

Objective: To implement and compare the performance of a Simple Recurrent Neural Network (RNN) and a Bi-directional Long Short-Term Memory (LSTM) network for sentiment analysis on the IMDb movie review dataset. This experiment aims to understand the impact of network architecture, particularly the ability of Bi-directional LSTMs to capture contextual information from both forward and backward sequences, on sentiment classification accuracy.

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

Introduction:

Sentiment analysis is a crucial task in Natural Language Processing (NLP) that involves determining the emotional tone expressed in a piece of text. Movie reviews provide a rich source of data for sentiment analysis, where the goal is often to classify a review as either positive or negative.

Recurrent Neural Networks (RNNs) are well-suited for processing sequential data like text. A Simple RNN processes the sequence in one direction, maintaining a hidden state that captures information from previous timesteps. However, it might struggle to capture context from later parts of the sequence when making predictions about earlier parts.

Long Short-Term Memory (LSTM) networks are a type of RNN that can learn long-range dependencies in sequential data, mitigating the vanishing gradient problem. A Bi-directional LSTM further enhances this by processing the sequence in both forward and backward directions. This allows the network to gather contextual information from both before and after each word in the sequence, potentially leading to better understanding and classification of sentiment.

This experiment will involve building and training both a Simple RNN and a Bi-directional LSTM model on the IMDb movie review dataset and evaluating their performance in terms of accuracy and loss on a held-out test set.

Procedure:

Part 1: Simple RNN for Sentiment Analysis

1. **Import Libraries:** Import necessary libraries from Keras for loading the dataset, preprocessing text, building the model, and defining layers.

Python

Deep Learning Lab Manual (CS 405)

```
from keras.datasets import imdb
from keras.preprocessing.text import Tokenizer
from keras.utils import pad_sequences
from keras import Sequential
from keras.layers import Dense, SimpleRNN, Embedding, Flatten
```

2. **Load and Preprocess Data:** Load the IMDb dataset, which contains movie reviews labeled as positive (1) or negative (0). Pad the sequences to a fixed length (maxlen=50) to ensure uniform input size for the RNN. The padding='post' argument ensures that padding is added at the end of the sequences.

Python

```
(X_train, y_train), (X_test, y_test) = imdb.load_data()
X_train = pad_sequences(X_train, padding='post', maxlen=50)
X_test = pad_sequences(X_test, padding='post', maxlen=50)
print(X_train.shape)
```

3. **Build the Simple RNN Model:** Create a sequential model.

- Add an Embedding layer to convert word indices into dense vector representations. The first argument (10000) specifies the vocabulary size (number of unique words to consider), and the second argument (2) defines the dimensionality of the embedding vectors.
- Add a SimpleRNN layer with 32 units. The input_shape=(50, 1) specifies that the input sequences have a length of 50 and each element in the sequence has a dimensionality of 1 (after embedding, the dimensionality will be 2, but input_shape refers to the shape of a single timestep before the RNN layer). return_sequences=False means the layer will output only the final hidden state.
- Add a Dense output layer with 1 unit and a sigmoid activation function for binary classification (positive or negative sentiment).

Python

```
model = Sequential()
model.add(Embedding(10000, 2))
model.add(SimpleRNN(32, input_shape=(50, 1), return_sequences=False)) # Corrected input_shape
model.add(Dense(1, activation='sigmoid'))
model.summary()
```

4. **Compile the Model:** Configure the model for training by specifying the optimizer (adam), loss function (binary_crossentropy for binary classification), and evaluation metric (accuracy).

Python

```
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['acc'])
```

5. **Train the Model:** Train the model on the training data for a specified number of epochs (epochs=5) and evaluate its performance on the validation data (validation_data=(X_test, y_test)).

Python

```
model.fit(X_train, y_train, epochs=5, validation_data=(X_test, y_test))
```

Deep Learning Lab Manual (CS 405)

6. **Evaluate the Model:** Evaluate the trained model on the test data to get the final test loss and accuracy.

Python

```
test_loss, test_acc = model.evaluate(X_test, y_test)
print('Test loss', test_loss)
print('Test accuracy', test_acc)
```

Part 2: Bi-directional LSTM for Sentiment Analysis

1. **Import Libraries:** Import necessary libraries, including Bidirectional and LSTM layers.

Python

```
import numpy as np
from keras.preprocessing import sequence
from keras.utils import pad_sequences
from keras.models import Sequential
from keras.layers import Dense, Dropout, Embedding, LSTM, Bidirectional
from keras.datasets import imdb
from matplotlib import pyplot
```

2. **Load and Preprocess Data:** Load the IMDb dataset, limiting the vocabulary size to the top n_unique_words = 10000 most frequent words. Pad the sequences to a maximum length of maxlen = 200.

Python

```
n_unique_words = 10000 # cut texts after this number of words
maxlen = 200
batch_size = 128
(x_train, y_train), (x_test, y_test) = imdb.load_data(num_words=n_unique_words)
x_train = pad_sequences(x_train, maxlen=maxlen)
x_test = pad_sequences(x_test, maxlen=maxlen)
y_train = np.array(y_train)
y_test = np.array(y_test)
```

3. **Build the Bi-directional LSTM Model:** Create a sequential model.

- Add an Embedding layer with the specified vocabulary size (n_unique_words), embedding dimension (128), and input length (maxlen).
- Add a Bidirectional(LSTM(64)) layer. This wraps an LSTM layer, applying it to the input sequence in both forward and backward directions and then concatenating the outputs. The LSTM layer has 64 hidden units.
- Add a Dropout layer with a rate of 0.5 to prevent overfitting.
- Add a Dense output layer with 1 unit and a sigmoid activation function for binary classification.

Python

```
model = Sequential()
model.add(Embedding(n_unique_words, 128, input_length=maxlen))
model.add(Bidirectional(LSTM(64)))
model.add(Dropout(0.5))
model.add(Dense(1, activation='sigmoid'))
```

Deep Learning Lab Manual (CS 405)

```
model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
model.summary()
```

4. **Train the Model:** Train the model for a specified number of epochs (epochs=10) with a given batch size (batch_size = 128) and evaluate its performance on the validation data. Store the training history.

Python

```
history = model.fit(x_train, y_train, batch_size=batch_size, epochs=10,
                     validation_data=[x_test, y_test])
```

5. **Evaluate the Model:** Evaluate the trained Bi-directional LSTM model on the test data.

Python

```
test_loss, test_acc = model.evaluate(x_test, y_test)
print('Test loss', test_loss)
print('Test accuracy', test_acc)
print(history.history['loss'])
print(history.history['accuracy'])
```

6. **Visualize Training History:** Plot the training loss and accuracy over the epochs to observe the learning process.

Python

```
pyplot.plot(history.history['loss'], label='train_loss')
pyplot.plot(history.history['val_loss'], label='val_loss')
pyplot.plot(history.history['accuracy'], label='train_accuracy')
pyplot.plot(history.history['val_accuracy'], label='val_accuracy')
pyplot.title('Model Loss and Accuracy')
pyplot.xlabel('Epoch')
pyplot.legend()
pyplot.show()
```

Expected Outcomes:

By comparing the test accuracy and loss of the Simple RNN and the Bi-directional LSTM, we expect the Bi-directional LSTM to achieve higher accuracy and potentially lower loss. This improvement is attributed to its ability to consider the context of words from both directions in the sequence, leading to a better understanding of the sentiment expressed in the movie reviews. The visualization of the training history for the Bi-directional LSTM will provide insights into the model's learning curve and potential overfitting.

Deep Learning Lab Manual (CS 405)

Experiment: 15 Design and Implement the CNN with Regularization and Dropout on Fashion MNIST Dataset

Objective

To design, implement, and evaluate a Convolutional Neural Network (CNN) with four or more convolutional layers for classifying multi-category images from the Fashion MNIST dataset. The experiment involves applying different regularization techniques—namely L1, L2, and Dropout—to reduce overfitting and enhance generalization performance. The objective is to compare the training and testing accuracy across five different CNN configurations: a base model, models using L1 and L2 regularization individually, a model with dropout, and a model that combines L2 regularization with dropout.

Time Required: 3 hrs

Programming Language: e.g. Python

Software/Tools Required: e.g. Anaconda/ Google Colab

Introduction

Convolutional Neural Networks (CNNs) have demonstrated outstanding performance in image classification tasks by effectively learning spatial hierarchies in images through convolution and pooling operations. However, CNNs are prone to overfitting, especially when trained on limited datasets like Fashion MNIST, which consists of grayscale images representing various clothing items.

To mitigate overfitting, regularization techniques are used. L1 regularization encourages sparsity in network weights, while L2 regularization penalizes large weight values, promoting generalization. Dropout is another effective regularization method that randomly disables a fraction of neurons during training, reducing interdependency among neurons.

This experiment explores how these techniques individually and in combination affect the classification performance of CNNs on the Fashion MNIST dataset. Each model is evaluated on the basis of training and testing accuracy, demonstrating the practical impact of regularization on deep learning models. Through this comparative analysis, the experiment aims to provide insights into selecting the appropriate regularization strategy for robust CNN performance.

```
# L1 Regularizer
import keras
from keras.datasets import fashion_mnist
from keras.layers import Dense, Activation, Flatten, Conv2D, MaxPooling2D
from keras.models import Sequential
from keras.regularizers import l1
from keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
(train_X,train_Y), (test_X,test_Y) = fashion_mnist.load_data()
train_X = train_X.reshape(-1, 28,28, 1)
test_X = test_X.reshape(-1, 28,28, 1)
```

Deep Learning Lab Manual (CS 405)

```
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X / 255
test_X = test_X / 255
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)
model = Sequential()
model.add(Conv2D(256,(3,3),input_shape=(28, 28, 1),kernel_regularizer=l1(0.01)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(128, (3,3),kernel_regularizer=l1(0.01)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
R-20
c.
model.add(Conv2D(64, (3,3), input_shape=(28, 28, 1),
#kernel_regularizer=l1(0.01)
))
model.add(Activation('relu'))
#model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(28, (3,3),
#kernel_regularizer=l1(0.01)
))
model.add(Activation('relu'))
#model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(64))
model.add(Dense(10))
model.add(Activation('softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=keras.optimizers.Adam(),metrics=['accuracy'])
model.fit(train_X, train_Y_one_hot, epochs=5)
test_loss, test_acc = model.evaluate(test_X, test_Y_one_hot)
print('Test loss', test_loss)
print('Test accuracy', test_acc)
predictions = model.predict(test_X)
print(np.argmax(np.round(predictions[0])))
plt.imshow(test_X[0].reshape(28, 28), cmap = plt.cm.binary)
plt.show()
# L2 regularizer
import keras
from keras.datasets import fashion_mnist
from keras.layers import Dense, Activation, Flatten, Conv2D, MaxPooling2D
from keras.models import Sequential
from keras.regularizers import l2
from keras.utils import to_categorical
import numpy as np
```

Deep Learning Lab Manual (CS 405)

```
import matplotlib.pyplot as plt
(train_X,train_Y), (test_X,test_Y) = fashion_mnist.load_data()
train_X = train_X.reshape(-1, 28,28, 1)
R-20
test_X = test_X.reshape(-1, 28,28, 1)
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X / 255
test_X = test_X / 255
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)
model = Sequential()
model.add(Conv2D(256,(3,3),input_shape=(28,28,1), kernel_regularizer=l2(0.01)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(128, (3,3),
#kernel_regularizer=l2(0.01)
))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64, (3,3), input_shape=(28, 28, 1),
#kernel_regularizer=l2(0.01)
))
model.add(Activation('relu'))
#model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(28, (3,3),
#kernel_regularizer=l2(0.01)
))
model.add(Activation('relu'))
#model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(64))
model.add(Dense(10))
model.add(Activation('softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,optimizer=keras.optimizers.Adam(),metrics=['accuracy'])
model.fit(train_X, train_Y_one_hot, epochs=5)
test_loss, test_acc = model.evaluate(test_X, test_Y_one_hot)
print('Test loss', test_loss)
print('Test accuracy', test_acc)
R-20
d.
#Dropout
import keras
from keras.datasets import fashion_mnist
from keras.layers import Dense, Activation, Flatten, Conv2D, MaxPooling2D, Dropout
from keras.models import Sequential
```

Deep Learning Lab Manual (CS 405)

```
from keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
(train_X,train_Y), (test_X,test_Y) = fashion_mnist.load_data()
train_X = train_X.reshape(-1, 28,28, 1)
test_X = test_X.reshape(-1, 28,28, 1)
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X / 255
test_X = test_X / 255
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)
model = Sequential()
model.add(Conv2D(256, (3,3), input_shape=(28, 28, 1)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
#Dropout(0.20)
model.add(Conv2D(128, (3,3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
#Dropout(0.20)
model.add(Conv2D(64, (3,3), input_shape=(28, 28, 1)))
model.add(Activation('relu'))
#model.add(MaxPooling2D(pool_size=(2,2)))
#Dropout(0.20)
model.add(Conv2D(28, (3,3)))
model.add(Activation('relu'))
predictions = model.predict(test_X)
print(np.argmax(np.round(predictions[0])))
plt.imshow(test_X[0].reshape(28, 28), cmap = plt.cm.binary)
plt.show()
R-20
e.
#model.add(MaxPooling2D(pool_size=(2,2)))
#Dropout(0.20)
model.add(Flatten())
model.add(Dense(64))
model.add(Dense(10))
model.add(Activation('softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
optimizer=keras.optimizers.Adam(),metrics=['accuracy'])
model.fit(train_X, train_Y_one_hot, batch_size=64, epochs=5)
test_loss, test_acc = model.evaluate(test_X, test_Y_one_hot)
print('Test loss', test_loss)
print('Test accuracy', test_acc)
predictions = model.predict(test_X)
print(np.argmax(np.round(predictions[0])))
```

Deep Learning Lab Manual (CS 405)

```
plt.imshow(test_X[0].reshape(28, 28), cmap = plt.cm.binary)
plt.show()
# L2 regularizer and Dropout
import keras
from keras.datasets import fashion_mnist
from keras.layers import Dense, Activation, Flatten, Conv2D, MaxPooling2D
from keras.models import Sequential
from keras.regularizers import l2
from keras.utils import to_categorical
import numpy as np
import matplotlib.pyplot as plt
(train_X,train_Y), (test_X,test_Y) = fashion_mnist.load_data()
train_X = train_X.reshape(-1, 28,28, 1)
test_X = test_X.reshape(-1, 28,28, 1)
train_X = train_X.astype('float32')
test_X = test_X.astype('float32')
train_X = train_X / 255
test_X = test_X / 255
train_Y_one_hot = to_categorical(train_Y)
test_Y_one_hot = to_categorical(test_Y)
R-20
model = Sequential()
model.add(Conv2D(256, (3,3), input_shape=(28, 28, 1), kernel_regularizer=l2(0.01))
))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
Dropout(0.20)
model.add(Conv2D(128, (3,3),
#kernel_regularizer=l2(0.01)
))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(64, (3,3), input_shape=(28, 28, 1),
#kernel_regularizer=l2(0.01)
))
model.add(Activation('relu'))
#model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Conv2D(28, (3,3),
#kernel_regularizer=l2(0.01)
))
model.add(Activation('relu'))
#model.add(MaxPooling2D(pool_size=(2,2)))
model.add(Flatten())
model.add(Dense(64))
model.add(Dense(10))
model.add(Activation('softmax'))
model.compile(loss=keras.losses.categorical_crossentropy,
```

Deep Learning Lab Manual (CS 405)

```
optimizer=keras.optimizers.Adam(),metrics=['accuracy'])
model.fit(train_X, train_Y_one_hot, epochs=5)
test_loss, test_acc = model.evaluate(test_X, test_Y_one_hot)
print('Test loss', test_loss)
print('Test accuracy', test_acc)
predictions = model.predict(test_X)
print(np.argmax(np.round(predictions[0])))
plt.imshow(test_X[0].reshape(28, 28), cmap = plt.cm.binary)
plt.show()
```

Deep Learning Lab Manual (CS 405)

Final Lab Project

Introduction

Human Action Recognition (HAR) has emerged as a pivotal task in computer vision, enabling systems to interpret and respond to human movements captured through video or sensor data. Applications of HAR span a broad spectrum, including intelligent surveillance, healthcare monitoring, smart homes, virtual reality, and human-computer interaction. Recent advances in deep learning, particularly Convolutional Neural Networks (CNNs), have revolutionized the way spatial features are extracted from visual data. However, standalone CNNs often fall short in capturing temporal dependencies inherent in human motion sequences. To address this, hybrid models combining CNNs with Recurrent Neural Networks (RNNs) such as Long Short-Term Memory (LSTM) or Gated Recurrent Units (GRU) have been developed. Incorporating attention mechanisms and dilated convolutions further enhances these models by improving focus on salient features and capturing wider receptive fields without increased computational cost.

Problem Statement

Traditional HAR systems suffer from several limitations, including inadequate modeling of temporal relationships, high computational complexity, and poor generalization in dynamic, real-world environments. Conventional CNNs are proficient at extracting spatial features but are limited in temporal understanding, while basic RNNs often face issues such as vanishing gradients and inefficiencies in learning long-term dependencies. Furthermore, many existing models are not optimized for real-time applications or deployment on resource-constrained devices. Therefore, there is a pressing need for a hybrid deep learning model that effectively captures both spatial and temporal patterns, is computationally efficient, and can operate robustly across varied conditions and datasets.

Objectives

1. **Design a Hybrid Model:** Develop a deep learning framework that integrates Dilated CNNs for rich spatial feature extraction and Attention-based LSTMs for capturing temporal dependencies with improved focus.
2. **Enhance Recognition Accuracy:** Utilize attention mechanisms to prioritize relevant temporal segments, thereby improving the model's ability to distinguish between subtle human actions.
3. **Ensure Computational Efficiency:** Leverage dilated convolutions and lightweight architectural components to reduce processing overhead without compromising performance.
4. **Apply to Real-world Datasets:** Train and evaluate the model on benchmark HAR datasets such as UCF101, HMDB51, and WISDM to assess accuracy, robustness, and generalizability.

Deep Learning Lab Manual (CS 405)

UCF101 – Action Recognition Dataset

- **Description:** UCF101 is a comprehensive dataset containing 13,320 video clips categorized into 101 action classes, such as sports, musical instrument playing, and human-object interactions. The videos are sourced from YouTube, offering diverse backgrounds and lighting conditions, which makes it ideal for training robust HAR models.[PyTorch+5Papers with Code+5Hugging Face+5](#)
 - **Download:** Available on [Kaggle](#)[Kaggle+1](#)[Kaggle+1](#)
-

2. HMDB51 – Human Motion Database

- **Description:** HMDB51 comprises 6,766 video clips spanning 51 action categories, including facial actions, general body movements, and human interactions. The dataset is collected from various sources like movies and online videos, providing a rich variety of scenarios for action recognition tasks.[Papers with Code](#)
 - **Download:** Accessible through the [Serre Lab at Brown University](#)[Machine Learning Datasets+2](#)[serre-lab.clps.brown.edu+2](#)[MathWorks - Maker of MATLAB and Simulink+2](#)
-

3. WISDM – Wireless Sensor Data Mining Dataset

- **Description:** The WISDM dataset contains time-series data from smartphone and smartwatch sensors, capturing 18 different activities performed by 51 subjects. It includes accelerometer and gyroscope readings, making it suitable for sensor-based HAR studies.[UCI Machine Learning Repository+1](#)[UCI Machine Learning Repository+1](#)

Download: Available at the [UCI Machine Learning Repository](#)[UCI Machine Learning Repository](#)