UMU_54907 DeepLearning

2019VT

Lab3 v0.0

RL, NLP, GAN

sygr0003 ==Sylvie Grafeuille

UMEÅ UNIVERSITET

Applied Physics and Electronics

Umeå universitet

Postadress: 901 87 Umeå

Content

Figures

1. Intro

1.1. About this document

This documents is the final laboration rapport of the course UMU54907 about deep learning . It introduces the subjects of RL (reinforcement learning) NLP (natural language processing) and GAN (generative adversatial networks)

The purpose of this document is mainly to gather the basic information about the subjects.

The document written in English except for the “uppgift” given by the teacher written in Swedish.

V 0.0 , 20190515

Intended audience.

The reader is expected to be a beginner in machine learning with some basic knowledge about python , jupyter , colab and machine learning .

Structure of the document

After a Introduction chapter in which the purpose of this lab is defined there will be one chapter per subject.

In the final chapter (Appendix) additional information such as some technical background or interesting references or model implementations are presented , ordered by subjects

About the references

The explanation comes mostly from the book “deep_learning with python” Francois Cholet

in other cases the references is given

1.2. Uppgift Laboration 3

Intro

I Laboration 3 tänker jag att man ska prova lite mer avancerade saker inom Deep Learning. Vi

hinner inte gå inpå djupet men tänker att man nosar lite på 3 tekniker; Reinforcement Learning(RL), Hantering av naturligt språk (NLP) och Generativa nät (GAN).

Det kan ske genom att man testkör några program och ger sina kommentarer.

RL

Det handlar om att datorn lär sig saker genom att experimentera med sin omgivning; lite som barn

eller djur lär sig tänker jag. Det bygger på att man får en belöning (reward) om man gör något bra i

en viss situation (state) och straff om man gjort nåt dumt, förstärkt inlärning. Det fina är alltså att datorn kan lära sig något där det inte finns någon lärare/facit, t ex spelet Go. Vi ska titta på ett reglerproblem; cart-pole, som bygger på att man ska balansera en pinne, kopplad via gångjärn på toppen av en liten (järnvägs)vagn, genom att köra/putta vagnen framåt eller bakåt. Spårets längd är också begränsat så man får straff om pinnen ramlar ner eller om man slår i ändpunkterna. Målet är att få så mycket lycka (reward) som möjligt genom att hålla pinnen upprätt så länge som möjligt.
Här kommer vi köra lite från OpenAI's gym. Det är ett bibliotek med olika testmiljöer så man slipper programmera upp allt själv utan kan koncentrera sig på lärandet.
Vi kommer implementera situationerna som en tabell som fylls med inlärda värden (Q-värden) som talar om hur bra olika åtgärder (actions) är i just den situationen och då kan man ju välja den åtgärd som verkar bäst (exploit) i den situation man befinner sig i men ibland behöver man välja andra åtgärder för att lära sig mer om omgivningen, det kan ju finnas bättre som man inte vet om man aldrig provat (explore).
En mer avancerad variant (Deep RL) är att man ersätter situationstabellerna med Q-värden som neurala nät och då blir det lättare att interpolera mellan olika situationer och åtgärder men det blir också en del svårigheter.

NLP

Natural Language Processing, i det här fallet bygger på att orden får en vektor (embedding) med tal förknippade till sig. Dessa tal kan man få fram genom att datorn "läser" en massa text och hittar mönster i ordföljder. Talen representerar dimensioner i betydelsen av orden som kan liknas i t ex manlighet, mäktighet, naturlighet, fruktighet etc. Då kan man t ex räkna ut vad kung – man + kvinna är (drottning) och skapa mångdimensionella rymder där man kan se (med t ex tensorboard) hur orden hänger ihop med varandra och göra diverse analyser.

Det är vanligt att man använder återkopplade nät såsom RNN, LSTM och GRU för sådana här projekt.
Vi ska använda recensioner av filmer för att först lära sig vad som är goda respektive kritiska ord och sen gissa om andra filmer anses bra eller dåliga baserat på texten i recensionen.
En modern state-of-the-art metod är BERT som är rätt komplicerad men ger bättre resultat.

GAN

Bygger på att man har Två nät; dels ett, "konstnären", som genererar (skapar) data och sen ett annat nät "konstkritikern" som bedömer hur bra verk konstnären åstadkommit. Målet för konstnären är förstås att få kritikern att inte se skillnad på sitt verk och andras mästerverk och

kritikern över på att bli bättre och bättre på att skilja bra verk från dåliga.
Med denna konkurrens så blir båda näten bättre och bättre och fascinerande kreativa.
Vi ska använda en konstnär som ska konstruera siffor som ska bli förvillande lika äkta handskrivna siffror och kritikern ska lära sig att se skillnad på vad som är riktiga siffor och vad som är kråkfötter.

Utförande redovisning betyg

Du ska köra de 3 (+ ev några avancerade) tillhandahållna Python-programmen samt justera, modifiera och kommentera lite i koden så läraren kan se att du förstår vad som händer.

Grundversioner

RL: Cart-pole
Colab-Länk finns i Moodle:
https://colab.research.google.com/drive/1hOTmbBSiZS1Ea77_7iwwpayvcjLNDDfZ

Beskrivande kommentarer och källkod till Cart-pole-miljön
https://gym.openai.com/envs/CartPole-v1/#id1
https://github.com/openai/gym/blob/master/gym/envs/classic_control/cartpole.py
Uppgift RL: kör colab-versionen och förbättra och beskriv koden litegrann. Plotta ut hur mycket reward den får allteftersom för att visa om den blir bättre eller möjligen sämre? Testa med några olika parametervärden, t ex öka antalet episoder för att få den att lära sig bättre. Epsilon styr hur girig (exploit) den ska vara – om ett slumptal är under Epsilon så provar den ett slumpmässigt värde för att experimentera/lära sig.
Rewarden är hur många steps som den lyckas få pinnen att fortsätta stå upp, >180 räknas som mycket bra. Vilka 4 mätningar ingår i state-informationen som grund för om man ska putta till höger eller vänster?

NLP: Exempel 6.2 ur kursboken
https://github.com/fchollet/deep-learning-with-python-notebooks

GAN: Enkel GAN i Colab
Enkel GAN i Colab, TF2.0, del av Lab 3
https://www.tensorflow.org/alpha/tutorials/generative/dcgan

Avancerade versioner

RL: Deep RL (kommer snart)

NLP: BART

GAN: 8.3-neural-style-transfer med en egen bild

Redovisning o Betyg

Skärmdumpar av körningar och dina kommentarer i en pdf-rapport på ca 3 - 5 sidor.

Kör man grundversionen av programmen och kommenterar vettigt får man G. Kör man en extra avancerad version får man VG och två avancerade versioner får man MVG.

1.3. My environment for programming and test

My local environment is a desktop (no GPU) running Ubuntu 18.04

the code is written in python . This lab involves python pgm run on localhost and colab , jupyter notebooks run on localhost and colab

Keras and tensor flow as backend where used.

The versions in local host

```
syl1>  python3  --version
Python 3.6.7
syl1>  jupyter --version
4.4.0
syl1>  python3 localW0.py
Using TensorFlow backend.
keras.__version__:  2.2.4
```

version and in colab:

```
! python –version
Python 3.6.7
! jupyter --version
4.4.0
import keras
Using TensorFlow backend.
print (keras.__version__)
```

```
2.2.4
import tensorflow as tf
tf.version.VERSION
'1.13.1'
```

I wonder if it is not a coincidence that the versions are the same .

1.4. About the files on gitHub

All files related to this lab (program code and this pdf document) are gathered at

https://github.com/SylGrafe/RepoDl01/lab3

notice that just lab3 is involved the other folder do not belongs to this lab.

Notebooks and python files

As Colab is in a virtual environment without easy procedure to read and write files I decided to put all information in this pdf file lab3sygr0003.pdf

For nlp

lab3_nlp0.ipynb original file (without any additional comments)

for gan

lab3_gan0.ipynbn original rgram to study (without any additional comments)

for rl

lrl_prorok.py original file from Kalle prorok for rl

lab3_rl0.py modifie version of rl_prorok.py (without any additional comments)

lab3_cartpolev1.py define the environmanet cartpole1 (without any additional comments)

2. RL0

2.1. Preliminaries

This section is an introduction of Reinforcement Learning (RL). . The aim is to use the cart_pole

environment to learn more about RL

The original programs to study are rl_prorok nad vartpolev1.

lab3_cartpolev1.py Classic cart-pole system implemented by Rich Sutton et al.

rl_prorok.py : the original pgm written by Kalle Prorok to implement RL

lab3_rl0.py is a modifed version of rl_Prorok.

Installation and hello world

In order to use cartpole it was necessary to install gym

about gym

https://github.com/openai/gym/wiki/Environments

An environment is a problem with a minimal interface that an agent can interact with. The environments in the OpenAI Gym are designed in order to allow objective testing and bench-marking of an agents abilities.

Install gym

```
syl1>  pip3 install --user gym
Collecting gym
  Downloading
https://files.pythonhosted.org/packages/7b/57/e2fc4123ff2b4e3d61ae9b3d08c6878aecf2d5ec69b585ed53bc2400607f/gym-0.12.1.tar.gz (1.5MB)
    100% |████████████████████████████████| 1.5MB 8.6MB/s
….
Successfully built gym future
Installing collected packages: future, pyglet, gym
Successfully installed future-0.17.1 gym-0.12.1 pyglet-1.3.2
```

cartpole-v0 first run

http://gym.openai.com/docs/

CartPole-v0 belongs to Classic control and toy text and is like a hello world pgm to learn about

gym and to learn about RL.

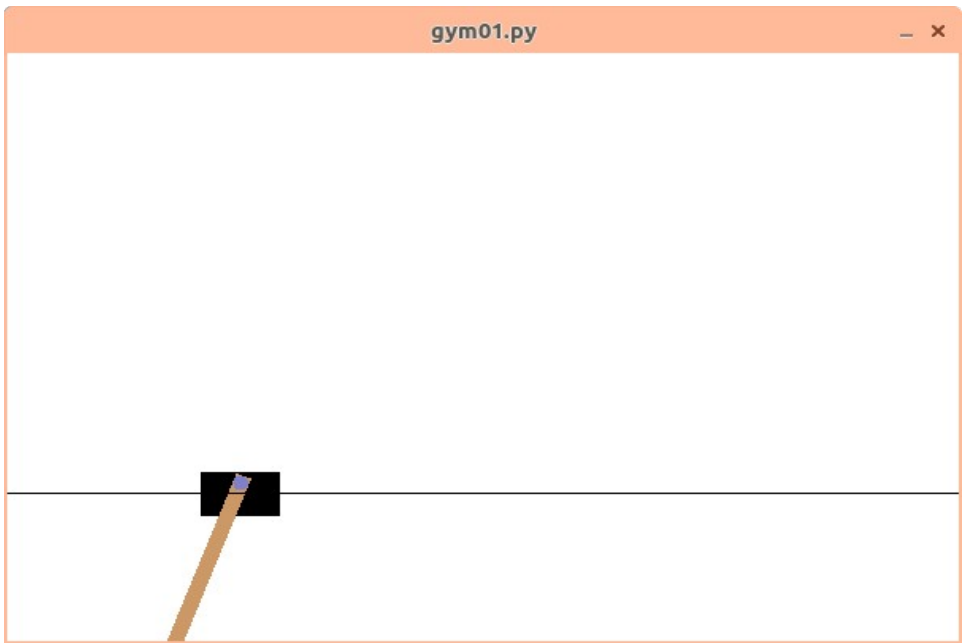
The aim of the game is to learn which movement to apply to the cart in order to have the pole vertically above the cart. In the first demo random movements (left or rights) are applied to the cart and the pole never manage to get to and keep the wanted position so the game is over very rapidelly

Here's a bare minimum example of getting something running. This will run an instance of the CartPole-v0 environment for 1000 timesteps, rendering the environment at each step. You should see a window pop up rendering the classic cart-pole problem:

Here is the python code:

```
syl1> cat gym1.py
Import gym
from gym import envs
env = gym.make('CartPole-v0')
env.reset()
for _ in range(1000):
    env.render()
    env.step(env.action_space.sample()) # take a random action
env.close()
```

env.renderer () will produce a popup window containing the animated cartpole . The cart is running randomly on the horizontal line and the pole is under the cart , the whole window disappear within seconds.



2.2. lab3_rl programs

Analyse rl_Prorok.py

Using q tables the pgm try the choose the best actions in order to keep the pole into acceptable positions .

The same procedure is run several time:

At each step in a given run

qtable are updated in order to remember which states give the best estimated final values.

If the estimated final value for a state is already know and if the pgm is “greedy ” then the pgm will use the best action

Sometimes the pgm is not greedy and will **explore** , it chooses use a random action instead of choosing the best value according to the qtable. The parameter epsilon control how often a random action is taken

A run finishes when the environment evaluate that the cart or the pole are outside given limits.

Notice that the environment always retrun a reward of 1 until the run is finished so conting the **accumulated rewards** is the same that saying how many steps the pole and the cart had acceptable values .

make_state

Each state observation consist of 4 float defines in cartPolev1.py

Num	Observation	Min	Max
0	Cart Position	-4.8	4.8
1	Cart Velocity	-Inf	Inf
2	Pole Angle	-24 deg	24 deg
3	Pole Velocity At Tip	-Inf	Inf

the metod make_state () transform this tulpe of 4 float into a single integer . This is necessary to order to bee able to be able to build a qtable where each state is a element.

Qtable

With DISCRETE_STEPS==10 there are 10000 elements in qtables each element contains the qvalues for this state . In other word

qtable[sti][0] estimation of Value for action 0 (left) and state sti

qtable[sti][1] estimation of Value for action 1 (right) and state sti

```
# Q-table for the discretized states, and two actions
num_states = DISCRETE_STEPS ** 4
qtable = [[0., 0.] for state in range(num_states)]
```

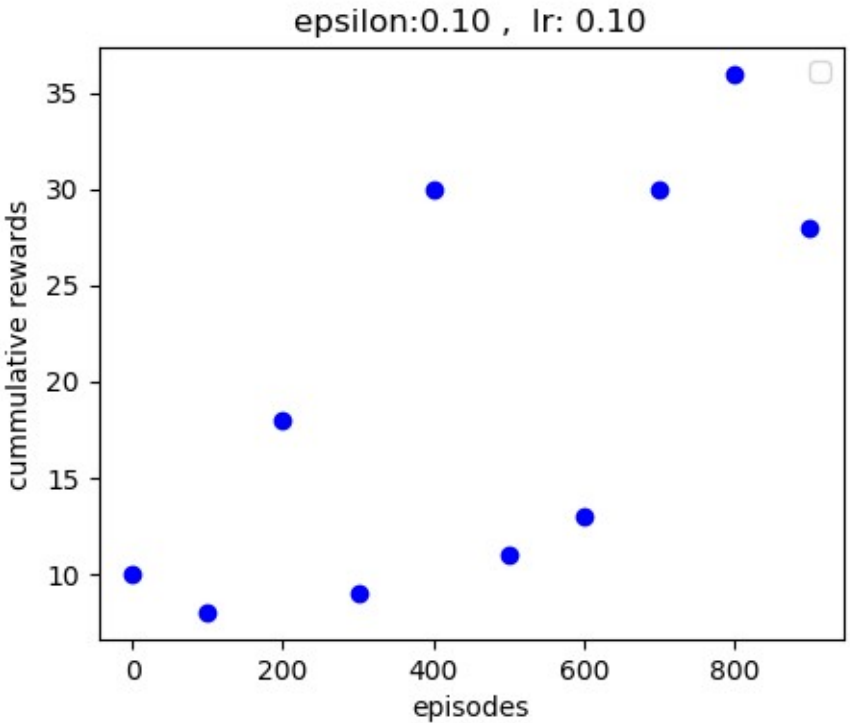
plot results

The results varies greatly from run to run

Results of one program run (with 1000 episodes) are presented here.

The results of 10 runs are plotted , one each 100 episode

It illustrate quite well the facts that program does not seem to learn evenly



lab3_rl0.py

This program is a modified version of rl_prorok,py and the main differences are

To evaluate the learning the **mean accumulated average** and the **ratio of good runs** are considered.

A good run get a accumulated rewards of 180 (accumulated rewards == nb of steps in which the pole the system is in acceptable state)

The **epsilon value is adjusted** to get it smaller when the pgm seems to have learn already well.

It is addictive to run the pgm and try to get the best hyper parameters. It seems that there is not easy solution. I made many runs and presents here just some results

results

As text

List the average of accumulated rewards and ration of good run . The reason to present this text is to notice when epsilon value was changed. A time stempla identify the run and is also visible in the titles of in the plot presented afterwards in order to link a list to a plot.

((getting better))

```
syl1> python3 lab_rl0.py
 Check at , av_value, good_count_ratio
1000 , 12, 0
```

```
2000  , 12, 0
 2000 ------------------------------> myEpsilon:0.15
3000  , 22, 0
4000  , 36, 0
5000  , 89, 10
6000  , 101, 12
7000  , 134, 16
8000  , 129, 11
9000  , 134, 13
10000  , 131, 13
2105_171110 epsi , start0.10 [0.02,0.15]
  lr: 0.01 , random: True
 good_run (val: 180 , Limit:166)
```

((getting worst !))

```
syl1>  python3   lab_rl0.py
 Check at  , av_value, good_count_ratio
1000  , 31, 0
2000  , 29, 0
 2000 ------------------------------> myEpsilon:0.15
3000  , 60, 5
4000  , 79, 8
5000  , 65, 2
6000  , 70, 4
7000  , 153, 29
 7000 ------------------------------> myEpsilon:0.07
8000  , 220, 44
 8000 ------------------------------> myEpsilon:0.04
9000  , 247, 48
 9000 ------------------------------> myEpsilon:0.02
10000  , 80, 3
 10000 ------------------------------> myEpsilon:0.04
2105_171146 epsi , start0.10 [0.02,0.15]
  lr: 0.01 , random: True
 good_run (val: 180 , Limit:166)
```

((another getting worst))

```
syl1>  python3   lab_rl0.py
```

```
 Check at  , av_value, good_count_ratio
1000  , 18, 0
2000  , 17, 0
 2000 ------------------------------> myEpsilon:0.15
3000  , 27, 0
4000  , 47, 0
5000  , 60, 0
6000  , 92, 4
7000  , 127, 13
8000  , 114, 9
9000  , 111, 9
10000  , 89, 2
2105_171209 epsi , start0.10 [0.02,0.15]
  lr: 0.01 , random: True
 good_run (val: 180 , Limit:166)
```

((not learning))

```
syl1>  python3   lab_rl0.py
 Check at  , av_value, good_count_ratio
1000  , 18, 0
2000  , 17, 0
 2000 ------------------------------> myEpsilon:0.15
3000  , 18, 0
4000  , 27, 0
5000  , 34, 0
6000  , 27, 0
7000  , 28, 0
8000  , 33, 0
9000  , 35, 0
10000  , 36, 0
2105_171602 epsi , start0.10 [0.02,0.15]
  lr: 0.01 , random: True
 good_run (val: 180 , Limit:166)
```

((quite good))

```
syl1>  python3   lab_rl0.py
 Check at  , av_value, good_count_ratio
```

```
1000  , 13, 0
2000  , 13, 0
 2000 ------------------------------> myEpsilon:0.15
3000  , 14, 0
4000  , 32, 2
5000  , 61, 7
6000  , 118, 18
 6000 ------------------------------> myEpsilon:0.07
7000  , 143, 26
 7000 ------------------------------> myEpsilon:0.04
8000  , 157, 33
 8000 ------------------------------> myEpsilon:0.02
9000  , 156, 34
10000  , 159, 34
2105_171751 epsi , start0.10 [0.02,0.15]
  lr: 0.01 , random: True
 good_run (val: 180 , Limit:166)
```

((even better))

```
syl1>  python3   lab_rl0.py
 Check at  , av_value, good_count_ratio
1000  , 23, 0
2000  , 24, 0
 2000 ------------------------------> myEpsilon:0.15
3000  , 27, 0
4000  , 53, 0
5000  , 70, 6
6000  , 120, 15
7000  , 136, 22
 7000 ------------------------------> myEpsilon:0.07
8000  , 157, 27
 8000 ------------------------------> myEpsilon:0.04
9000  , 170, 38
 9000 ------------------------------> myEpsilon:0.02
10000  , 201, 76
2105_172413 epsi , start0.10 [0.02,0.15]
  lr: 0.02 , random: True
 good_run (val: 180 , Limit:166)
```

plots

The corresponding plots are shown all at once

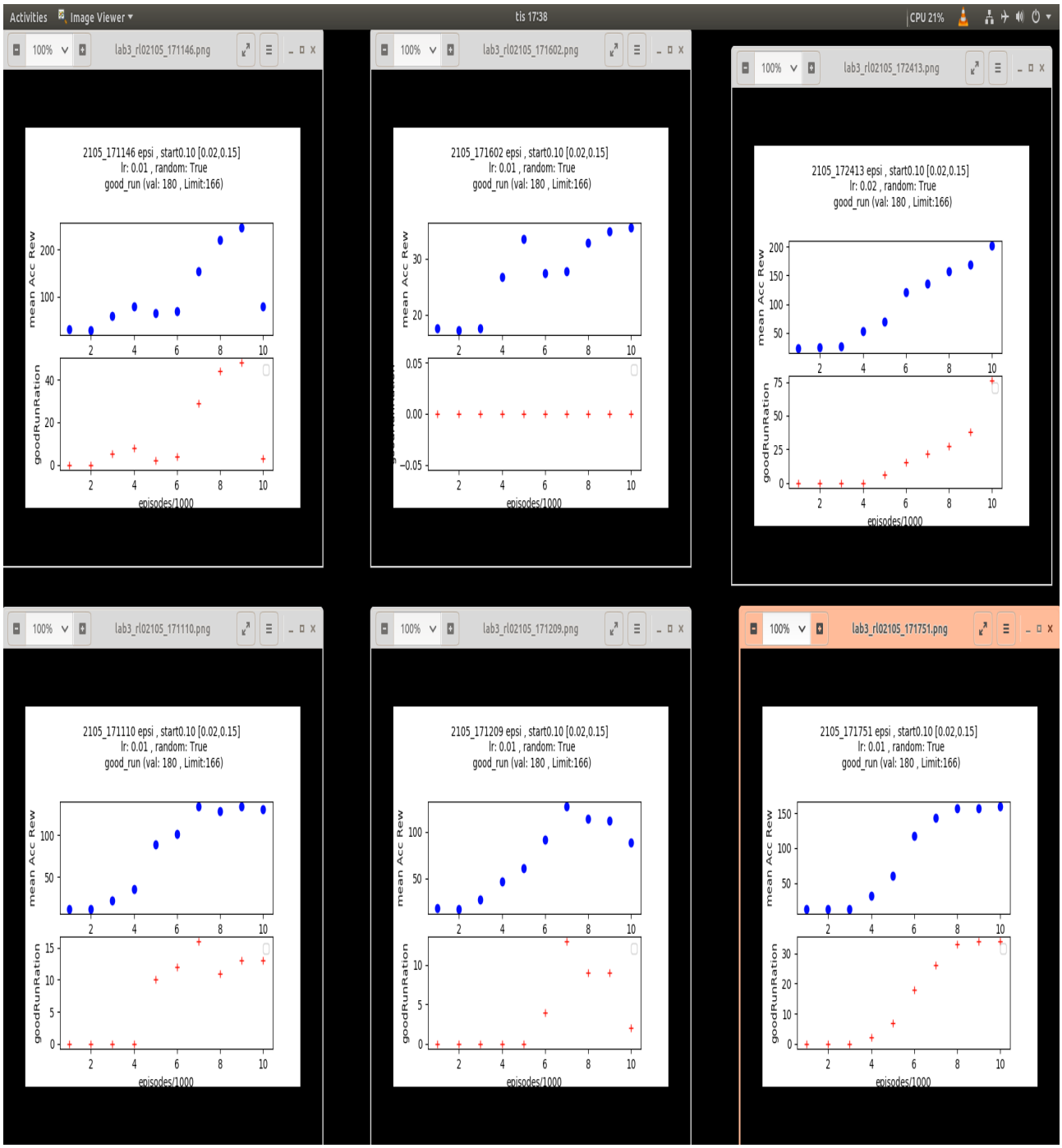


Illustration 1: mean accumulated rewards and good run ratios

3. NLP0

3.1. Intro

The aim of this part of the laboration is an introduction to NLP = naturel language processing.

The purpose is to study this code

6.2-understanding-recurrent-neural-networks.py , from the course book (Cholet) to be found at

Ref: https://github.com/fchollet/deep-learning-with-python-notebooks

This code has been copied in lab3_nlp0.ipynb and run in colab

problem definition

IMDB sentiment--classification task . The model must be able to classify movie reviews as positives or negatives.

kort analyse

The problem can be solved by using simple densely connected model or by using

Embbedding layer and RNN.

By using embedding you create a word encoding in a space in which the words are related in order to reflect the structure of the language,

By using RNN you take into account the sequence (meaning that the position on the words in the text is relevant to solve the problem)

3.2. lab3_nlp0.ipynb

Three models are implemented in lab3_nlp0:

model0 a simple a RNN

model1 one lstm layer

model00 as a reference is a model with only densely connected layers (3 total)

the training curves are included in this document,

code extract

An extract of the code about the data processing and the implementation of model1 is given here

```
max_features=10000
maxlen=500
(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_features)
input_train = sequence.pad_sequences(input_train, maxlen=maxlen)
input_test = sequence.pad_sequences(input_test, maxlen=maxlen)
model1 = Sequential()
model1.add(Embedding(max_features, 32))
model1.add(LSTM(32))
model1.add(Dense(1, activation='sigmoid'))

model1.compile(optimizer='rmsprop',
            loss='binary_crossentropy',
            metrics=['acc'])

print ("\nlab3_nlp0 train model1 : Embedd (maxFeat , 32 ) + LSTM(32)  ")
hist1 = model1.fit(input_train, y_train,
                  epochs=myEpochs,
                  batch_size=128,
                 verbose=1 ,
                  validation_split=0.2)
```

explanations about this code are given in the technical background section

technical background

imdb dataset , imdb.load_data

ref: https://keras.io/datasets/

Dataset of movies reviews from IMDB, labeled by sentiment (positive/negative).

The reviews (sequences of words) have been turned into sequences of integers, where each integer stands for a specific word in a dictionary. For convenience, words are indexed by overall frequency in the dataset, so that for instance the integer "3" encodes the 3rd most frequent word in the data. This allows for quick filtering operations such as: "only consider the top 10,000 most common words, but eliminate the top 20 most common words".

(input_train, y_train), (input_test, y_test) = imdb.load_data(num_words=max_features)

num_words: integer or None. Top most frequent words to consider. Any less frequent word will appear as oov_char value in the sequence data.

pad_sequences

ref: https://keras.io/preprocessing/sequence/

keras.preprocessing.sequence.pad_sequences(sequences, maxlen=None, dtype='int32', padding='pre', truncating='pre', value=0.0)

This function transforms a list of num_samples sequences (lists of integers) into a 2D Numpy array of shape (num_samples, num_timesteps). num_timesteps is either the maxlen argument if provided, or the length of the longest sequence otherwise.

conclusion on data preprocessing :

Bu using pad_sequence you can transform each review as a time steps with 500 elements (each element is an integer) to be fed into the embedding layer.

apropos embedding

ref Cholet ch06 : Word embeddings are dense, relatively lowdimensional, and learned from data.

It’s reasonable to learn a new embedding space with every new task.

The Embedding layer takes at least two arguments: the number of possible tokens and the dimensionality of the embeddings .

The Embedding layer is best understood as a dictionary that maps integer indices (which stand for specific words) to dense vectors. It takes integers as input, it looks up these integers in an internal dictionary, and it returns the associated vectors. It’s effectively a dictionary lookup

3.3. trainings curves

The corresponding trainings curves seems to indicate that lstm is slightly better than than other models

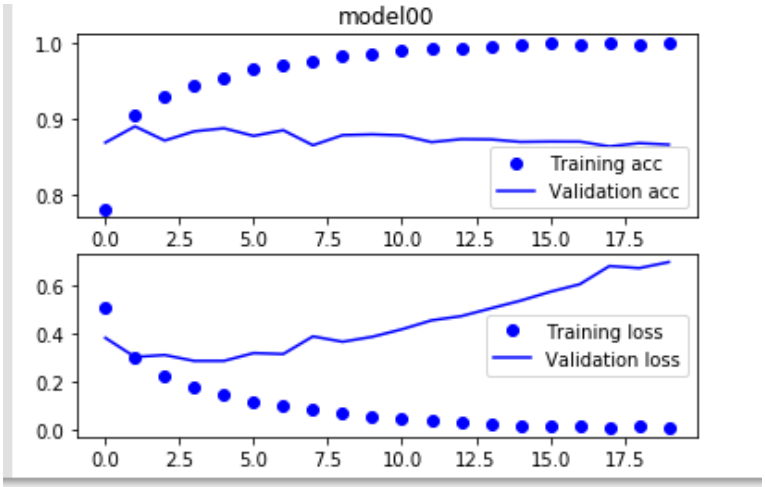


Illustration 2: nlp0 , training Model00 (dense only)

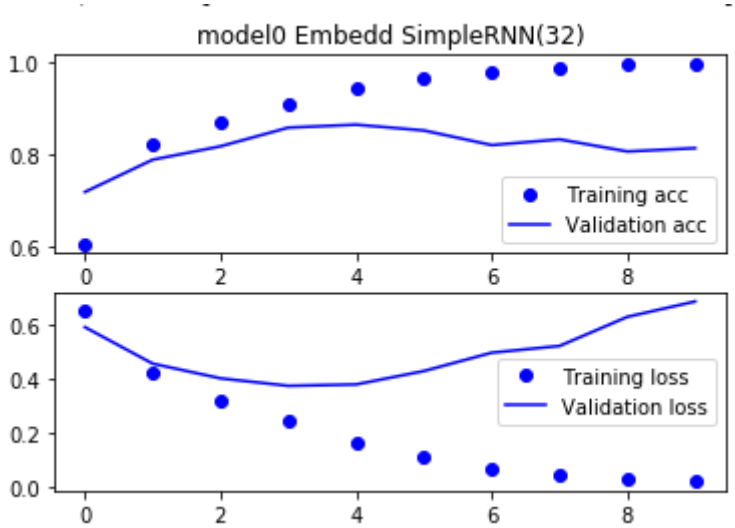


Illustration 3: nlp0 training model0 : RNN

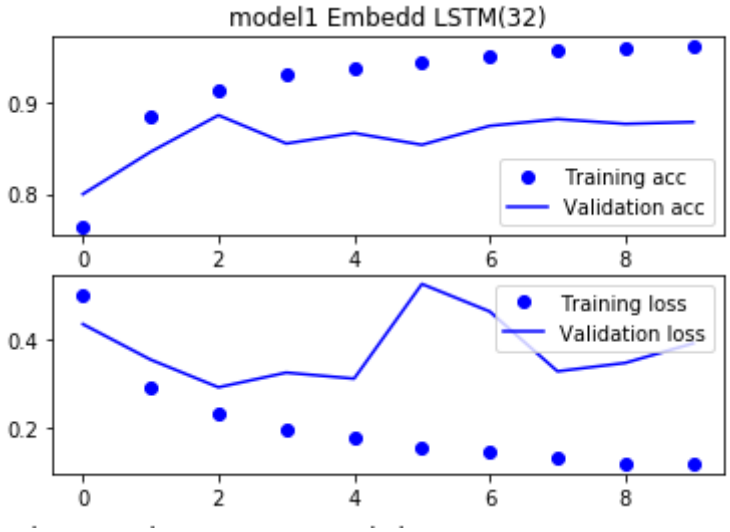


Illustration 4: nlp0 , training model1 (lstm)

Al l curves shows that the models overfitt rapidelly.lstm seem to be only sligthly better than a model with only denselly connected layers. As usual lstm has much more computational cost

(see computational cost in the annex)

4. GAN0

4.1. intro

Generativ adversarial networks GAN can be used to create pictures.

The tutorial demonstrates how to generate images of handwritten digits using a GAN.

The code to study is defined in

https://colab.research.google.com/github/tensorflow/docs/blob/master/site/en/r2/tutorials/generative/dcgan.ipynb

lab3_gan0 is a copy of dcgan.iypnb that was run in colab with gpu .

4.2. Brief presentation of the code

Two models are trained simultaneously by an adversarial process. A **generator** ("the artist") learns to create images that look real, while a **discriminator** ("the art critic") learns to tell real images apart from fakes. During training, the **generator** progressively becomes better at creating images that look real, while the **discriminator** becomes better at telling them apart. The process reaches equilibrium when the **discriminator** can no longer distinguish real images from fakes.

The code for GAN is quite complicated , in this section I present the most relevant extract from the code

generator

Looking at the code of the generator model I notice BatchNormalization , LeakyRelu and Conv2DTranspose

```
def make_generator_model():
    model = tf.keras.Sequential()
    model.add(layers.Dense(7*7*256, use_bias=False, input_shape=(100,)))
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Reshape((7, 7, 256)))
    assert model.output_shape == (None, 7, 7, 256) # Note: None is the batch size
    model.add(layers.Conv2DTranspose(128, (5, 5), strides=(1, 1), padding='same', use_bias=False))
    assert model.output_shape == (None, 7, 7, 128)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(64, (5, 5), strides=(2, 2), padding='same', use_bias=False))
    assert model.output_shape == (None, 14, 14, 64)
    model.add(layers.BatchNormalization())
    model.add(layers.LeakyReLU())
    model.add(layers.Conv2DTranspose(1, (5, 5), strides=(2, 2), padding='same', use_bias=False, activation='tanh'))
    assert model.output_shape == (None, 28, 28, 1)
    return model
…
generator=make_generator_model()
```

the discriminator

The code for the discriminator is more familiar and look like a typical CNN.

```
def make_discriminator_model():
    model = tf.keras.Sequential()
    model.add(layers.Conv2D(64, (5, 5), strides=(2, 2), padding='same',     input_shape=[28, 28, 1]))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
    model.add(layers.Conv2D(128, (5, 5), strides=(2, 2), padding='same'))
    model.add(layers.LeakyReLU())
    model.add(layers.Dropout(0.3))
```

```
    model.add(layers.Flatten())
    model.add(layers.Dense(1))
    return model
…
discriminator=make_discriminator_model()
...
```

Model training

The “heart of the matter” for model training is defined in the module train_step where you can see that both generator and discriminator are called

```
def train_step(images):
    noise = tf.random.normal([BATCH_SIZE, noise_dim])
    with tf.GradientTape() as gen_tape, tf.GradientTape() as disc_tape:

      generated_images = generator(noise, training=True)
      real_output = discriminator(images, training=True)
      fake_output = discriminator(generated_images, training=True)

      gen_loss = generator_loss(fake_output)
      disc_loss = discriminator_loss(real_output, fake_output)

    gradients_of_generator = gen_tape.gradient(gen_loss, generator.trainable_variables)
    gradients_of_discriminator = disc_tape.gradient(disc_loss, discriminator.trainable_variables)

    generator_optimizer.apply_gradients(zip(gradients_of_generator, generator.trainable_variables))
    discriminator_optimizer.apply_gradients(zip(gradients_of_discriminator, discriminator.trainable_variables))
…
```

train_step is then called in the train method an extract on this is presented now

```
def train(dataset, epochs):
  for epoch in range(epochs):
    start = time.time()
    for image_batch in dataset:
      train_step(image_batch)
    # Produce images for the GIF as we go      display.clear_output(wait=True)
```

```
    generate_and_save_images(generator,      epoch + 1,    seed)
    # Save the model every 15 epochs
    if (epoch + 1) % 15 == 0:
      checkpoint.save(file_prefix = checkpoint_prefix)
    print ('Time for epoch {} is {} sec'.format(epoch + 1, time.time()-start))
  # Generate after the final epoch
  display.clear_output(wait=True)
  generate_and_save_images(generator,     epochs,           seed)
```

4.3. Some images produced under training

Some images produced by generate_and_save_images(model, epoch, test_input)
are presented now to illustrate how the generator gets better and better to produce reliable digits

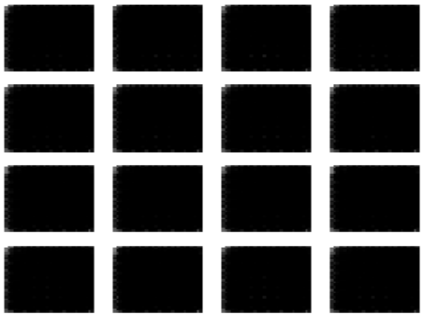


Illustration 5: images at epoch1

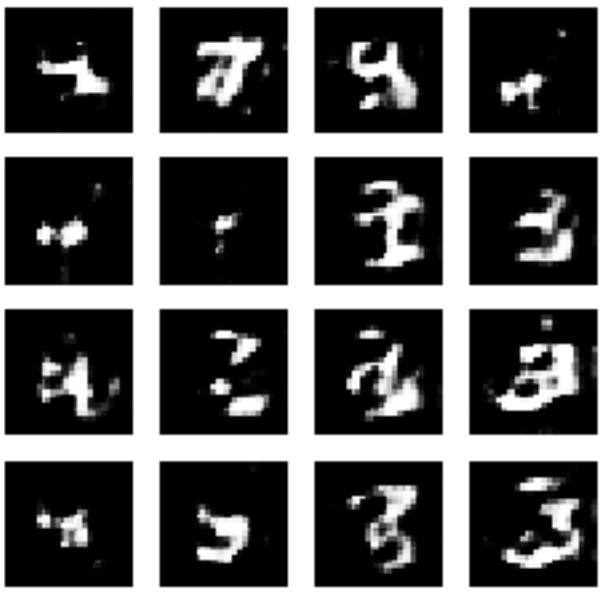


Illustration 6: gan0 images at epoch 20

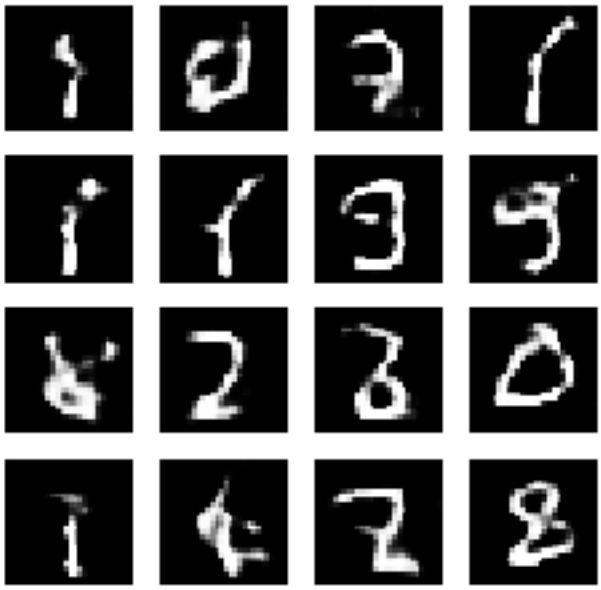


Illustration 7: gan0 images at epoch 40

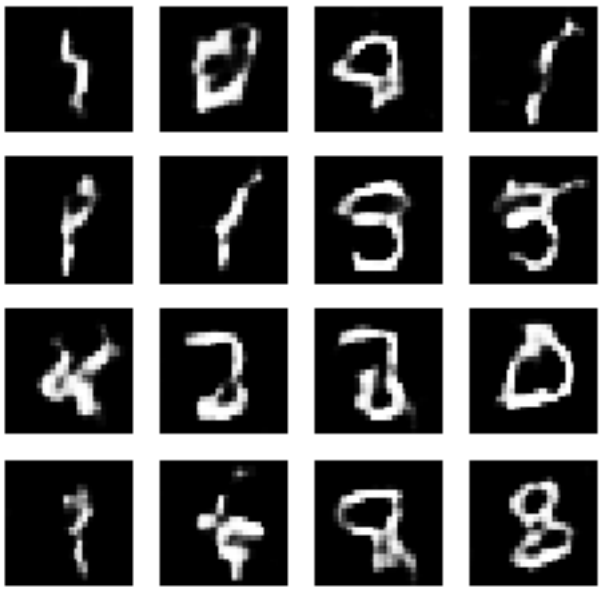


Illustration 8: gan0 images at epoch 50

5. Annexe

More info about the different subjected is gathered in this sections

5.1. RL

Notions about RL

A Beginner's Guide to Deep Reinforcement Learning

https://skymind.ai/wiki/deep-reinforcement-learning

I found this article from skymind about RL extremely well written and interesting :

In this extract some notions are presented in particular : q-value , exploit and explore.

Reinforcement learning refers to **goal-oriented algorithms**, which learn how to attain a complex objective (goal) or maximize along a particular dimension over many steps;

...

Reinforcement learning solves the difficult problem of **correlating immediate actions with the delayed returns they produce**.

...

Reinforcement learning can be understood using the concepts of **agents, environments, states, actions and rewards**, all of which we'll explain below.

...

So environments are functions that transform an action taken in the current state into the next state and a reward; agents are functions that transform the new state and reward into the next action. We can know the agent's function, **but we cannot know the function of the environment**. It is a black box where we only see the inputs and outputs.

....

Reinforcement learning represents an **agent's attempt to approximate the environment's function**, such that we can send actions into the black-box environment that maximize the rewards it spits out.

...

Value (V): The expected long-term return with discount, as opposed to the short-term reward R. $V\pi(s)$ is defined as the expected long-term return of the current state under policy π. We discount rewards, or lower their estimated value, the further into the future they occur. See discount factor. And remember Keynes: "In the long run, we are all dead." That's why you discount future rewards.

Q-value or action-value (Q): Q-value is similar to Value, except that it takes an extra parameter, the current action a. $Q\pi(s, a)$ refers to the long-term return of the current state s, taking action a under policy π. Q maps state-action pairs to rewards. Note the difference between Q and policy.

...

The goal of reinforcement learning is to pick the best known action for any given state, which means the actions have to be ranked, and assigned values relative to one another. Since those actions are state-dependent, what we are really gauging is the value of state-action pairs; i.e. an action taken from a certain state, something you did somewhere.

...

We map state-action pairs to the values we expect them to produce with the Q function, described above. The Q function takes as its input an agent's state and action, and maps them to probable rewards.

Reinforcement learning is the process of **running the agent through sequences of state-action pairs, observing the rewards that result, and adapting the predictions of the Q function to those rewards** until it accurately predicts the best path for the agent to take. That prediction is known as a policy.

...

After a little time spent employing something like a Markov decision process to **approximate the probability distribution of reward** over state-action pairs, a reinforcement learning algorithm may tend to repeat actions that lead to reward and cease to test alternatives. There is a tension between the exploitation of known rewards, and continued exploration to discover new actions that also lead to victory.

...

reinforcement learning algorithms can be made to both **exploit** and **explore** to varying degrees, in order to ensure that they don't pass over rewarding actions at the expense of known winners.

...

Where do neural networks fit in? **`Neural networks are the agent that learns to map state-action pairs to rewards`**. Like all neural networks, they use coefficients to approximate the function relating inputs to outputs, and their learning consists to finding the right coefficients, or weights, by iteratively adjusting those weights along gradients that promise less error.

...

At the beginning of reinforcement learning, the neural network coefficients may be initialized stochastically, or randomly. Using feedback from the environment, the neural net can use the difference between its expected reward and the ground-truth reward to adjust its weights and improve its interpretation of state-action pairs.

...

Like human beings, the **Q function is recursive**. Just as calling the wetware method human() contains within it another method human(), of which we are all the fruit, calling the Q function on a given state-action pair requires us to call a nested Q function to predict the value of the next state, which in turn depends on the Q function of the state after that, and so forth.

Reinforcement Learning and DQN, learning to play from pixels

https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html

Posted on August 24, 2016 by Ruben Fiszel

I like this article in particular the explanations about experience replay. Experience replay is in use in the model presented as “real life with cart-pole”

Experience replay There is one issue with using neural network as Q approximator. The transitions are very correlated. This reduce the overall variance of the transition. After all, they are all extracted from the same episode. Imagine if you had to learn a task without any memory (not even short-term), you would always optimise your learning based on the last episode. The Google DeepMind research team used experience replay, which is a windowed buffer of the last N transitions (N being a million in the original paper) with DQN and greatly improved their performances on atari. Instead of updating from the last transition, you store it inside the experience replay and update from a batch of randomly sampled transitions from the same experience replay.

Real life cart-pole RL and DL

A Real life cart pole with real RL is presented in this blog :

https://towardsdatascience.com/cartpole-introduction-to-reinforcement-learning-ed0eb5b58288

A real installation is filmed (rail cart and pole) and you see some training session an then the final results where the agent has learned to maintain the pole vertically above the cart , it is really impressive The blog is also really interesting . and I decided to study the code a bit closer and try to understand (I did not understand so well)

description of the program

The model is trained with a denselly connected network.

The code is disponible here: https://github.com/gsurma/cartpole

two main pyrthon pgm are involved: carptole.py and score_logger.py

cartpole.py define the class DQNSolver

DQN (Deep Q network) is a RL technique that is aimed at choosing the best action for given circumstances (observation). Each possible action for each possible observation has its Q value, where 'Q' stands for a quality of a given move.

And DQNSolver implement a densely connected model

```
self.model = Sequential()
self.model.add(Dense(24, input_shape=(observation_space,), activation="relu"))
self.model.add(Dense(24, activation="relu"))
self.model.add(Dense(self.action_space, activation="linear"))
self.model.compile(loss="mse", optimizer=Adam(lr=LEARNING_RATE))
```

cartpole.py is the implementation of the agent , model.predict and model.fit are called in the code. Predict is used for action as you could guess:

```
def act(self, state):
  if np.random.rand() < self.exploration_rate:
    return random.randrange(self.action_space)
  q_values = self.model.predict(state)
  return np.argmax(q_values[0])
```

I did not really understand the function experience_replay () but notice that both predict and fit. are called from this function

```
def experience_replay(self):
  if len(self.memory) < BATCH_SIZE:
    return
  batch = random.sample(self.memory, BATCH_SIZE)
  for state, action, reward, state_next, terminal in batch:
    q_update = reward
    if not terminal:
      q_update = (reward + GAMMA * np.amax(self.model.predict(state_next)[0]))
    q_values = self.model.predict(state)
    q_values[0][action] = q_update
    self.model.fit(state, q_values, verbose=0)
```

```
        self.exploration_rate *= EXPLORATION_DECAY
        self.exploration_rate = max(EXPLORATION_MIN, self.exploration_rate)
```

the pgm exit when the score satify a given condition:

extract from score_logger.py

```
  def add_score(self, score, run):
    self._save_csv(SCORES_CSV_PATH, score)


    if mean_score >= AVERAGE_SCORE_TO_SOLVE and len(self.scores) >=
CONSECUTIVE_RUNS_TO_SOLVE:
      solve_score = run-CONSECUTIVE_RUNS_TO_SOLVE
```

running cartpole , program is learning !!

Running cartpole.py on local host (no more real life cart pole)

produces a list similar to lab3_rl0.py

but the score show the the program is really learning to control the pole.

```
syl1>  python3   cartpole.py
Using TensorFlow backend.
Run: 1, exploration: 1.0, score: 14
Scores: (min: 14, avg: 14, max: 14)


Run: 2, exploration: 0.9369146928798039, score: 19
/home/syl1/bin/python/deepL/gym0/scores/score_logger.py:81: RankWarning: Polyfit may be
poorly conditioned
  z = np.polyfit(np.array(trend_x), np.array(y[1:]), 1)
Scores: (min: 14, avg: 16.5, max: 19)


Run: 3, exploration: 0.8690529955452602, score: 16
Scores: (min: 14, avg: 16.333333333333332, max: 19)
```

….
Run: 125, exploration: 0.01, score: 167
Scores: (min: 8, avg: 192.85, max: 500)

Run: 126, exploration: 0.01, score: 139
Scores: (min: 8, avg: 194.14, max: 500)

Run: 127, exploration: 0.01, score: 280
Scores: (min: 8, avg: 196.84, max: 500)

Solved in 27 runs, 127 total runs.

5.2. NL

About lab3_nlp0

A propos loading the imdb database.

Because of I do not know what it was not possible any more to load the imdb database

One fix was to change the version of numpy

https://stackoverflow.com/questions/55890813/how-to-fix-object-arrays-cannot-be-loaded-when-allow-pickle-false-for-imdb-loa

How to fix 'Object arrays cannot be loaded when allow_pickle=False' for imdb.load_data() function? Ask Question

!pip install numpy==1.16.1

import numpy as np

and the corresponding output :

```
Collecting numpy==1.16.1
  Downloading https://files.pythonhosted.org/packages/f5/bf/4981bcbee43934f0adb8f764a1e70ab0ee5a448f6505bd04a87a2fda2a8b/numpy-1.16.1-cp36-cp36m-manylinux1_x86_64.whl (17.3MB)
     |████████████████████████████████| 17.3MB 3.5MB/s
ERROR: datascience 0.10.6 has requirement folium==0.2.1, but you'll have folium 0.8.3 which is incompatible.
ERROR: albumentations 0.1.12 has requirement imgaug<0.2.7,>=0.2.5, but you'll have imgaug 0.2.9 which is incompatible.
Installing collected packages: numpy
  Found existing installation: numpy 1.16.3
    Uninstalling numpy-1.16.3:
      Successfully uninstalled numpy-1.16.3
Successfully installed numpy-1.16.1
WARNING: The following packages were previously imported in this runtime:
  [numpy]
```

You must restart the runtime in order to use newly installed versions.

Computational cost

Run in colab using GPU

Model0 :RNN , 25s /epoch

train model0 : Embedd (maxFeat , 32) + RNN(32) Train on 20000 samples, validate on 5000 samples

Epoch 1/10 20000/20000 [==============================] - 26s 1ms/step - loss: 0.6493 - acc: 0.6057 - val_loss: 0.5884 - val_acc: 0.7178

Epoch 2/10 20000/20000 [==============================] - 25s 1ms/step - loss: 0.4226 - acc: 0.8208 - val_loss: 0.4540 - val_acc: 0.7876

model1 :lstm , 105 s/epoch

lab3_nlp0 train model1 : Embedd (maxFeat , 32) + LSTM(32) Train on 20000 samples, validate on 5000 samples

Epoch 1/10 20000/20000 [==============================] - 105s 5ms/step - loss: 0.5045 - acc: 0.7648 - val_loss: 0.4358 - val_acc: 0.7998

Epoch 2/10 20000/20000 [==============================] - 104s 5ms/step - loss: 0.2905 - acc: 0.8843 - val_loss: 0.3544 - val_acc: 0.8458 Epoch 3/10

model00: Dense only , 3s/epoch

train model00 Dense(16) , Dense(16), Dense(1)) Train on 15000 samples, validate on 10000 samples

Epoch 1/20 - 3s - loss: 0.5084 - acc: 0.7813 - val_loss: 0.3797 - val_acc: 0.8684

Epoch 2/20 - 2s - loss: 0.3004 - acc: 0.9047 - val_loss: 0.3004 - val_acc: 0.8897

5.3. GAN

More interesting aspects of the code

In this section is a a brief presentation of keras functions that I have not seen before .

Leaky version of a Rectified Linear Unit.

https://keras.io/layers/advanced-activations/

keras.layers.LeakyReLU(alpha=0.3)

It allows a small gradient when the unit is not active: f(x) = alpha * x for x < 0, f(x) = x for x >= 0.

Class Conv2DTranspose

https://www.tensorflow.org/api_docs/python/tf/keras/layers/Conv2DTranspose

Transposed convolution layer (sometimes called Deconvolution).

The need for transposed convolutions generally arises from the desire to use a ***transformation going in the opposite direction of a normal convolution***, i.e., from something that has the shape of the output of some convolution to something that has the shape of its input while maintaining a connectivity pattern that is compatible with said convolution.

checkpoint

In train () you notice the use of checkpoint.save that is shortly presented here:

Checkpointing Neural Network Models

https://machinelearningmastery.com/check-point-deep-learning-models-keras/

Application checkpointing is a fault tolerance technique for long running processes.

It is an approach where a snapshot of the state of the system is taken in case of system failure.

If there is a problem, not all is lost. The checkpoint may be used directly, or used as the

starting point for a new run, picking up where it left off.

When training deep learning models, the checkpoint is the weights of the model. These weights can be used to make predictions as is, or used as the basis for ongoing training.

https://keras.io/callbacks/#modelcheckpoint

keras.callbacks.ModelCheckpoint(filepath, monitor='val_loss', verbose=0, save_best_only=False, save_weights_only=False, mode='auto', period=1)

xxxxxxxxxxxxx end of document

about batchNormalization

https://keras.io/layers/normalization/

keras.layers.BatchNormalization(axis=-1, momentum=0.99, epsilon=0.001, center=True,

scale=True, beta_initializer='zeros', gamma_initializer='ones', moving_mean_initializer='zeros', moving_variance_initializer='ones', beta_regularizer=None, gamma_regularizer=None, beta_constraint=None, gamma_constraint=None)

Normalize the activations of the previous layer at each batch, i.e. *applies a transformation that maintains the mean activation close to 0 and the activation standard deviation close to 1*.