Lecture 8: Reinforcement Learning and Generative Models

Isaiah Hull[1,2]

[1]BI Norwegian Business School
[2]CogniFrame

December 5, 2023

Unsupervised Learning

Lecture 8: Overview

1. Generative Models.

2. Reinforcement Learning.

3. Applications.

1. Generative Models

Types of ML Models

- Discriminative models:
  - Applied to classification or regression.
  - Input: set of features.
  - Output: probabilities or predicted values.

Generative Models

Types of ML Models

- Generative models:
  - Learn the data distribution.
  - Produce new class examples.

Generative Models

Advancements in Generative ML

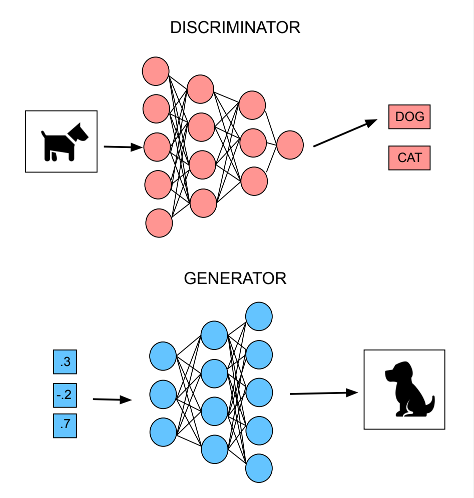
- Significant progress in generative ML.
- Notably:
  - Variational autoencoders (VAEs).
  - Generative adversarial networks (GANs).
- Breakthroughs in image, text, and music generation.

Generative ML in Economics

- Limited use of generative models in economics and finance.
- Growing interest in GANs.
- Example applications:
  - Athey et al. (2019).
  - Kaji et al. (2018).
- Potential future uses in the field.

Generative Models

Types of ML Models



Source: Hull (2021).

Generative Models

Variational Autoencoders

- Autoencoders:
  - Two networks: encoder and decoder with shared weights.
  - Encoder: Transforms input to latent state.
  - Decoder: Reconstructs features from latent state.
  - Trained via reconstruction loss.
- Uses: dimensionality reduction, generative tasks (images, music, texts).

Generative Models

Issues with Autoencoders

- Location and distribution of latent states:
  - Latent states are in R^N.
  - Clustering is not explicitly defined.
  - Affects determination of valid latent states for generation.
- Performance for unobserved latent states:
  - Only trained for specific examples.
  - Perturbations might lead to unconvincing outputs.

Variational Autoencoders (VAEs)

- Developed to address autoencoder limitations.
- Components:
  - Mean layer.
  - Log variance layer.
  - Sampling layer (from normal distribution).
- Output of sampling is the latent state.
- Encoder yields different latent states for same inputs.

VAE Architecture and Loss

- VAE modifies loss to include Kullback-Leibler (KL) divergence.
- KL divergence:
  - Penalizes distance between normal distributions.
  - Targets a mean and log variance of zero.

VAE Architecture and Loss

- Eliminates determinism of latent states.
- Improves generative performance.
- Solves sampling problem with sampling layer.
- Adjusts latent distribution with KL divergence.

VAEs in TensorFlow

- Detailed exploration: See Kingma and Welling (2019).
- Data used: GDP growth from 1961:Q2 to 2020:Q1 for 25 OECD countries.
- Previous usage: Dimensionality reduction.
- Objective: Train VAE to generate similar series.

Generative Models

Prepare GDP Growth Data for Use in a VAE

```
import tensorflow as tf
import pandas as pd
import numpy as np

# Load and transpose data.
GDP = pd.read_csv(data_path+'gdp_growth.csv',
        index_col = 'Date').T

# Print data preview.
print(GDP.head())
```

Model Architecture Summary

Time	4/1/61	7/1/61	10/1/61	1/1/62
AUS	-1.097616	-0.715607	1.139175	2.806800
AUT	-0.349959	1.256452	0.227988	1.463310
BEL	1.167163	1.275744	1.381074	1.346942
CAN	2.529317	2.409293	1.396820	2.650176
CHE	1.355571	1.242126	1.958044	0.575396

Model Architecture Summary

```
# Convert data to numpy array.
GDP = np.array(GDP)

# Set number of countries and quarters.
nCountries, nQuarters = GDP.shape

# Set number of latent nodes and batch size.
latentNodes = 2
batchSize = 1
```

Generative Models

VAE Model Architecture

- Contains encoder and decoder.
- Differs from autoencoder.
- Latent states sampled from independent normal distributions.

Sampling Layer in VAE

- Takes two parameters as inputs.
- Draws epsilon from standard normal distribution for each output node.
- Transforms each draw using mean and lvar parameters.

Generative Models

Model Architecture Summary

```
# Define function for sampling layer.
def sampling(params, batchSize = batchSize, latentNodes = latentNodes):
    mean, lvar = params
    epsilon = tf.random.normal(shape=(batchSize, latentNodes))
    return mean + tf.exp(lvar / 2.0) * epsilon
```

Encoder Model in VAE

- Takes full time series for a country as input.
- Mean and log variance layers differ from autoencoder.
- Mean and log variances layers have nodes for latent state.

Define Encoder Model for VAE

```
# Define input layer for encoder.
encoderInput = tf.keras.layers.Input(shape = (nQuarters))

# Define latent state.
latent = tf.keras.layers.Input(shape = (latentNodes))

# Define mean layer.
mean = tf.keras.layers.Dense(latentNodes)(encoderInput)
```

Define Encoder Model for VAE

```
# Define log variance layer.
lvar = tf.keras.layers.Dense(latentNodes)(encoderInput)

# Define sampling layer.
encoded = tf.keras.layers.Lambda(sampling,
        output_shape=(latentNodes,))([mean, lvar])

# Define model for encoder.
encoder = tf.keras.Model(encoderInput, [mean, lvar, encoded])
```

Further Details on Encoder

- Mean and lvar layers parameterize normal distributions.
- Lambda layer defined with the sampling function.
- Lambda layer takes mean and lvar parameters.

Generative Models

Further Details on Encoder

- Encoder model defined functionally.
- Inputs: quarterly GDP growth observations.
- Outputs: mean layer, log variance layer, and sampled outputs.

Functional Models for Decoder and VAE

- Decoder: accepts latent state and produces a reconstruction.
- VAE: takes a time series and transforms it into its reconstruction.

Defining the Loss Function

- Two-part loss function:
  - Reconstruction loss: same as autoencoder.
  - KL divergence: measures distance of sampling layer distributions from a standard normal.
- Further from standard normal distribution implies higher penalty.

Define Decoder Model for VAE

```
# Define output for decoder.
decoded = tf.keras.layers.Dense(nQuarters, activation = 'linear')(latent)

# Define the decoder model.
decoder = tf.keras.Model(latent, decoded)

# Define functional model for autoencoder.
vae = tf.keras.Model(encoderInput, decoder(encoded))
```

Define VAE Loss

```
# Compute the reconstruction component of the loss.
reconstruction = tf.keras.losses.binary_crossentropy(vae.inputs[0], vae.outputs[0])

# Compute the KL loss component.
kl = -0.5 * tf.reduce_mean(1 + lvar - tf.square(mean) - tf.exp(lvar), axis = -1)

# Combine the losses and add them to the model.
combinedLoss = reconstruction + kl
vae.add_loss(combinedLoss)
```

Trained Variational Autoencoder

- Capabilities of the trained VAE:
  - Use predict() method of vae to reconstruct given time series.
  - Generate realization of latent state for specific inputs, e.g., GDP growth for the US.
  - Perturb latent states by adding noise and generate new time series using predict() of decoder.

Compile and Fit VAE

```
# Compile the model.
vae.compile(optimizer='adam')

# Fit model.
vae.fit(GDP, batch_size = batchSize, epochs = 100)
```

Generate Latent States and Time Series with Trained VAE

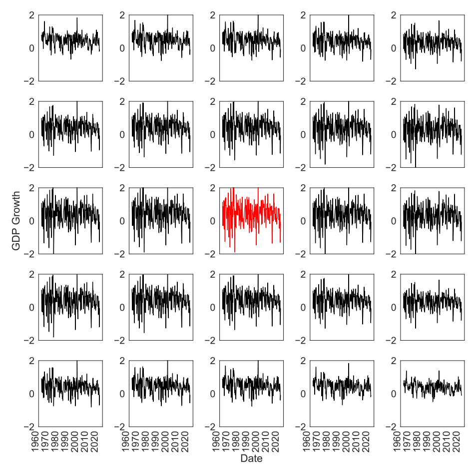
```
# Generate series reconstruction.
prediction = vae.predict(GDP[0,:].reshape(1,236))

# Generate (random) latent state from inputs.
latentState = encoder.predict(GDP[0,:].reshape(1,236))

# Perturb latent state.
latentState[0] = latentState[0] + np.random.normal(1)

# Pass perturbed latent state to decoder.
decoder.predict(latentState)
```

Generated Time Series for U.S. GDP Growth

- Based on a latent state realization for the U.S. GDP growth series.
- Perturbations over a 5x5 grid:
  - Rows: Add evenly-spaced values over the [-1,1] interval to the first latent state.
  - Columns: Add evenly-spaced values over the [-1,1] interval to the second latent state.
- Center series (in red): Adds [0,0] - Represents the original latent state.

Generative Models

VAE-Generated Time Series for U.S. GDP Growth



Source: Hull (2021).

VAE Application and Flexibility

- VAE architecture's broad applicability:
  - Add convolutional layers to encoder/decoder $\Rightarrow$ Generate images.
  - Incorporate LSTM cells $\Rightarrow$ generate text or music.
- LSTM-based architecture potential:
  - Possible improvements in time series generation.
  - A shift from the dense network approach used in the given example.

Generative Models

Generative Adversarial Networks

- Dominant generative ML models:
  - Variational Autoencoders (VAEs).
  - Generative Adversarial Networks (GANs).
- VAEs: Manipulate latent states for granular control.
- GANs: Convincing generation of class examples.
- Notable GAN results: Highly convincing generated images.

Generative Models

GANs vs VAEs

- VAEs:
  - Encoder + Decoder, joined by sampling layer.
- GANs:
  - Generator + Discriminator.
  - Generator: Latent state input $\Rightarrow$ class example (e.g. GDP growth time series).
  - Discriminator: Differentiate between real and fake examples.

GANs vs VAEs

- Adversarial Network:
  - Combines Generator + Discriminator.
  - Shares weights with both networks.
  - Trains the generator to maximize discriminator's loss.

Generative Models

GAN: Zero Sum Game

- Goodfellow et al. (2017):
  - Two networks in a zero-sum game.
  - Discriminator: v(g,d).
  - Generator: -v(g,d).
  - Evolutionary equilibrium.

GAN: Zero Sum Game

- Equilibrium Equation: $g^* = \arg\min_g \max_d v(g, d)$
- Freeze discriminator weights during adversarial training.
- Improve generation, rather than weakening the discriminator.

Generative Models

GAN Architecture Illustrated

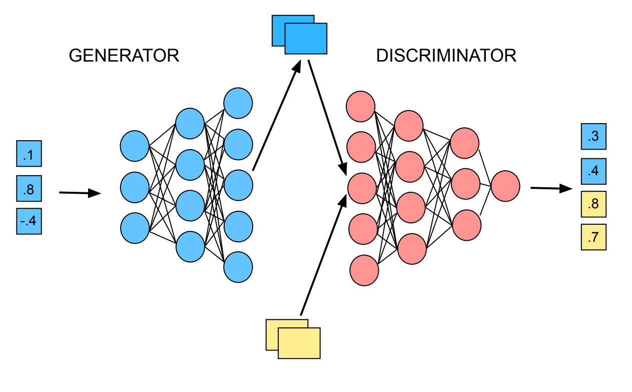
- GAN's generator and discriminator networks.
- Generator produces novel examples.
- Discriminator combines novel and true examples for classification.
- Adversarial Network:
  - Trains generator.
  - Attached to discriminator with frozen weights.
  - Iterative training for equilibrium.

Practical Application: GDP Growth Data

- GDP growth data (similar to VAE section).
- Train GAN for credible GDP growth time series generation from random vector input.
- Approach to GAN construction from Krohn et al. (2020).

Generative Models

Depiction of the Generator and Discriminator from a GAN.



Source: Hull (2021).

Prepare GDP Growth Data for Use in a GAN.

```
import tensorflow as tf
import pandas as pd
import numpy as np

# Load and transpose data.
GDP = pd.read_csv(data_path+'gdp_growth.csv',
        index_col = 'Date').T

# Convert pandas DataFrame to numpy array.
GDP = np.array(GDP)
```

Generative Model Definition

- Simple VAE model.
- Generator's input:
  - Vector with two elements.
  - Analogous to latent vector in VAE.

Generative Models

Generative Model Definition

- Generator viewed as a decoder.
- Architecture:
  - Start with bottleneck-type layer.
  - Upsample to produce GDP growth time series.

Generator's Architecture

- Simplest version:
  - Input layer: Accepts latent vector.
  - Output layer: Upsamples input layer.
- Output layer:
  - Represents GDP growth values.
  - Activation function: Linear.

Generative Models

Define the Generative Model of a GAN

```
# Set dimension of latent state vector.
nLatent = 2

# Set number of countries and quarters.
nCountries, nQuarters = GDP.shape

# Define input layer.
generatorInput = tf.keras.layers.Input(shape = (nLatent,))
```

Define the Generative Model of a GAN

```
# Define hidden layer.
generatorHidden = tf.keras.layers.Dense(16, activation='relu')(generatorInput)

# Define generator output layer.
generatorOutput = tf.keras.layers.Dense(236, activation='linear')(generatorHidden)

# Define generator model.
generator = tf.keras.Model(inputs = generatorInput, outputs = generatorOutput)
```

Generative Models

<u>Discriminator Definition</u>

- Inputs:
  - Real GDP growth series.
  - Generated GDP growth series.
- Each series length: nQuarters.
- Output: Probability of being a real GDP growth series for each input series.

Compilation Details

- Generator: Not compiled.
- Discriminator: Compiled.
- Use of an adversarial network to train the generator.

Define and Compile the Discriminator Model of a GAN

```
# Define input layer.
discriminatorInput = tf.keras.layers.Input(shape = (nQuarters,))

# Define hidden layer.
discriminatorHidden = tf.keras.layers.Dense(16,
          activation='relu')(discriminatorInput)
```

Define and Compile the Discriminator Model of a GAN

```
# Define discriminator output layer.
discriminatorOutput = tf.keras.layers.Dense(1,
                activation='sigmoid')(discriminatorHidden)

# Define discriminator model.
discriminator = tf.keras.Model(inputs = discriminatorInput,
              outputs = discriminatorOutput)

# Compile discriminator.
discriminator.compile(loss='binary_crossentropy',
              optimizer=tf.optimizers.Adam(0.0001))
```

Generative Models

Adversarial Model

- Shares weights with the generator.
- Uses a frozen version of discriminator's weights.
- Discriminator weights:
  - Do not update during adversarial training.
  - Update during discriminator training.

Adversarial Network

- Input: Latent vector (same size as generator input).
- Output of generator: timeSeries (fake GDP growth time series).
- Set discriminator′s trainability to 'False' during adversarial training.
- Network's output: Discriminator′s output.
- Defined and compiled functional model: adversarial.

Generative Models

Training Process

- Train the discriminator.
- Train the adversarial network.

Define and Compile the Adversarial Model of a GAN

```
# Define input layer for adversarial network.
adversarialInput = tf.keras.layers.Input(shape=(nLatent))

# Define generator output as generated time series.
timeSeries = generator(adversarialInput)

# Set discriminator to be untrainable.
discriminator.trainable = False
```

Define and Compile the Adversarial Model of a GAN

```
# Set discriminator to be untrainable.
discriminator.trainable = False

# Compute predictions from discriminator.
adversarialOutput = discriminator(timeSeries)

# Define adversarial model.
adversarial = tf.keras.Model(adversarialInput, adversarialOutput)

# Compile adversarial network.
adversarial.compile(loss='binary_crossentropy',
            optimizer=tf.optimizers.Adam(0.0001))
```

Generative Models

Train the Discriminator and the Adversarial Network

```
# Set batch size.
batch, halfBatch = 12, 6

for j in range(1000):
    # Draw real training data.
    idx = np.random.randint(nCountries,
            size = halfBatch)
    real_gdp_series = GDP[idx, :]

    # Generate fake training data.
    latentState = np.random.normal(size=[halfBatch, nLatent])
    fake_gdp_series = generator.predict(latentState)
```

Train the Discriminator and the Adversarial Network

```
# Combine input data.
features = np.concatenate((real_gdp_series,
        fake_gdp_series))

# Create labels.
labels = np.ones([batch,1])
labels[halfBatch:, :] = 0
```

Generative Models

Train the Discriminator and the Adversarial Network

```
# Train discriminator.
discriminator.train_on_batch(features, labels)

# Generate latent state for adversarial net.
latentState = np.random.normal(size=[batch, nLatent])

# Generate labels for adversarial network.
labels = np.ones([batch, 1])

# Train adversarial network.
adversarial.train_on_batch(latentState, labels)
```

Initialization

- Define the batch size.
- Begin training loop consisting of multiple steps.

Generative Models

Training the Discriminator

- Draw random integers for row selection in GDP matrix.
- Each row contains GDP growth time series (real samples for discriminator).
- Generate fake data:
  - Draw latent vectors.
  - Pass through generator.
- Combine real and fake series with labels (1 = real, 0 = fake).
- Train discriminator with combined data.

Training the Adversarial Network

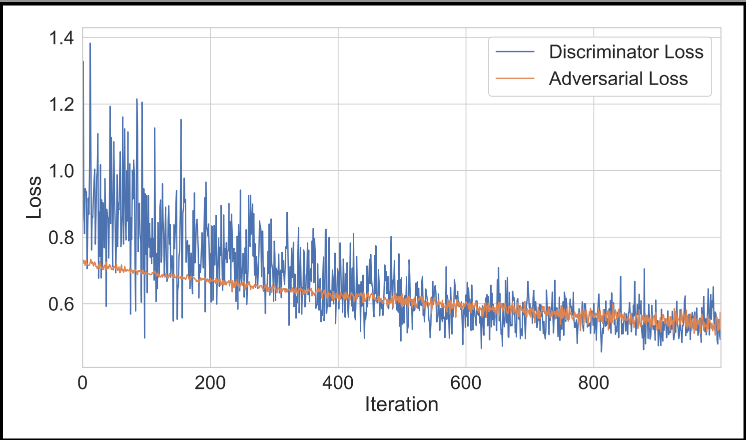
- Generate a batch of latent states.
- Input latent states into generator.
- Train with intent to trick discriminator into classifying as real.
- Iterative training over two models.
- Stopping criteria: stable evolutionary equilibrium.

Model Evaluation

- Observe model losses over time.
- Approx. 500 iterations: neither model shows significant improvement.
- Indicates a stable evolutionary equilibrium reached.

Generative Models

Discriminator and Adversarial Model Losses by Training Iteration



Source: Hull (2021).

Generative Models

Adversarial Network Performance

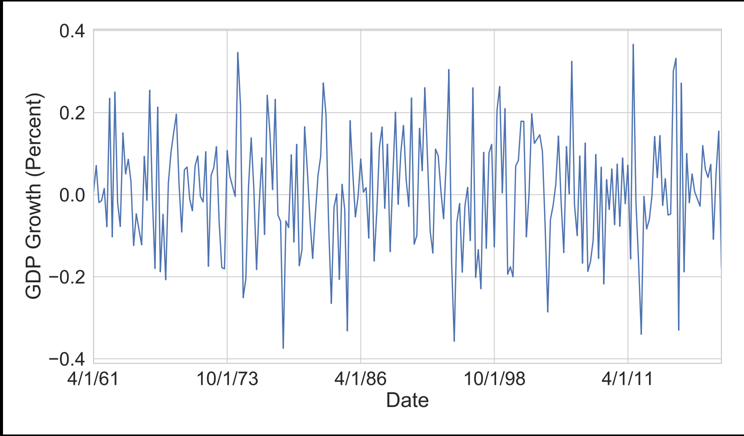
- White noise vector inputs used.
- Adversarial network informed by discriminator's performance.
- Achieved fairly credible fake GDP growth series after 1000 training iterations.

Performance Enhancement

- More latent features could improve performance.
- Can use a more advanced model architecture, such as LSTM.

Generative Models

Example Fake GDP Growth Series.



Source: Hull (2021).

Generative Models

Applications in Economics and Finance

- Concentrated on generating simulated series.
- Alternative to more familiar methods in Monte Carlo simulation.
- Must generate realistic series and capture interdependencies.

Early Applications of GANs in Economics

- Athey et al. (2019) used Wasserstein GANs.
- Simulate data appearing similar to small existing datasets.
- Bypasses drawing randomly or generating simulated series with deficiencies.

Value of GANs in Economics

- Athey et al. evaluated estimators using WGAN generated data.
- Kaji et al. (2018) used WGANs for indirect inference.
- Indirect inference: estimating structural models in economics and finance.

Generative Models

Kaji et al.'s Approach

- Estimate a model for workers choosing wage and location.
- Parameters are structural.
- Coupled model simulation with a discriminator.

Further Applications

- GANs and VAEs in image and text generation.
- Visual counterfactual simulations with economic data.
- NLP in economics: text generation to study company press releases.

2. Deep Reinforcement Learning

Reinforcement Learning

Deep Reinforcement Learning

- In standard models, agents are rational optimizers.
- Agents form unbiased expectations about the future.
- Optimizers choose the exact optimum.
- No heuristics or rule-of-thumb used.

Reinforcement Learning

Deviation from Rational Optimizer Framework

- Palmer (2015).
- Policy rule formation, rather than rationality.
- Improved computational tractability by breaking requirements.

Reinforcement Learning in Economics

- Alternative to standard model.
- Described in Sutton and Barto (1998).
- Value discussed in Athey and Imbens (2019), Palmer (2015).
- Application: Hull (2015) for dynamic programming problems.

Reinforcement Learning

Agents in Reinforcement Learning

- Agents perform optimization.
- Limited information about the system state.
- Tradeoff: “exploration" vs “exploitation."
- Learning about the system vs optimizing known system parts.

Reinforcement Learning

Deep Q-learning Introduction

- Variant of reinforcement learning.
- Combines deep learning and reinforcement learning.
- Can solve high-dimensional state space problems.
- Solves rational optimizer's problem via deep Q-learning.

Dynamic Programming vs Q-learning

- Dynamic programming uses "look-up table" for value states.
- Iterative table updates until convergence.
- Solution found in the value function table.
- Q-learning constructs a state-action table.
- Example: State - capital stock; Action - level of consumption.

Q-table Update Mechanism

- Updated using temporal difference learning.
- Updates state-action pair (s_t, a_t) in iteration $i + 1$.
- Uses value in iteration i and adds learning rate.
- Multiplied by expected change in value from optimal action.

Updating the Q Table

$$Q_{(i+1)}(s_t, a_t) \leftarrow Q_i(s_t, a_t) + \lambda \left[r_t + \beta \max_a Q(k_{(t+1)}, a) - Q_i(s_t, a_t) \right] \tag{1}$$

Deep Q-learning

- Replaces look-up table with a deep neural network: “deep Q-network."
- Approach introduced in Mnih et al. (2015).
- Originally used to train Q-networks for superhuman video game performance.

Applying Deep Q-learning to Economic Models

- Neoclassical business cycle model.
- Solutions implemented in TensorFlow.
- Two common TensorFlow options:
  - tf-agents: Native TensorFlow implementation.
  - keras-rl2: Uses high-level Keras API.
- We use keras-rl2 for simpler, familiar syntax.

Reinforcement Learning

Implementation Details

- Install keras-rl2 module.
- Submodules from rl module:
  - DQNAgent: Define a deep Q-learning agent.
  - EpsGreedyQPolicy: Set policy decisions process on training path.
  - SequentialMemory: Retain decision paths/outcomes for training.
- Import gym to define the model environment.

Install and Import Modules to Perform Deep Q-learning

```
# Install keras-rl2.
pip install keras-rl2

# Import numpy and tensorflow.
import numpy as np
import tensorflow as tf
```

Install and Import Modules to Perform Deep Q-learning

```
# Import reinforcement learning modules from keras.
from rl.agents.dqn import DQNAgent
from rl.policy import EpsGreedyQPolicy
from rl.memory import SequentialMemory

# Import module for comparing RL algorithms.
import gym
```

Setting up the Environment

- Set the number of capital nodes.
- Define environment: planner, a subclass of gym.Env.
- Specifies details of the social planner's reinforcement learning problem.

The planner Class: Initialization

- Define a discrete capital grid.
- Define action and observation spaces.
- Initialize the number of decisions to zero.
- Set the max number of decisions.
- Set node index of initial value of capital: 500 out of 1000.
- Set the production function parameter (alpha).

Action and Observation Spaces

- Both are discrete objects with 1000 nodes (defined using gym.spaces).
- Observation space: Entire state space (all capital nodes).
- Action space: Identical to the observation space.

Define Custom Reinforcement Learning Environment

```
# Define number of capital nodes.
n_capital = 1000

# Define environment.
class planner(gym.Env):
    def __init__(self):
        self.k = np.linspace(0.01, 1.0, n_capital)
        self.action_space = \
        gym.spaces.Discrete(n_capital)
        self.observation_space = \
        gym.spaces.Discrete(n_capital)
        self.decision_count = 0
```

Reinforcement Learning

Define Custom Reinforcement Learning Environment

```
        self.decision_max = 100
        self.observation = 500
        self.alpha = 0.33
    def step(self, action):
        assert self.action_space.contains(action)
        self.decision_count += 1
        done = False
```

Define Custom Reinforcement Learning Environment

```
if(self.observation**self.alpha - action) > 0:
    reward = \
np.log(self.k[self.observation]**self.alpha -
self.k[action])
else:
    reward = -1000
self.observation = action
```

Reinforcement Learning

Define Custom Reinforcement Learning Environment

```
        if (self.decision_count >= self.decision_max)\
        or reward == -1000:
            done = True
        return self.observation, reward, done,\
        {"decisions": self.decision_count}
    def reset(self):
        self.decision_count = 0
        self.observation = 500
        return self.observation
```

Defining the step Method

- Method returns observation (state), reward (utility), reset indicator (done), and debugging info.
- Increment decision_count attribute.
- Initially sets done to False.
- Evaluates validity of agent's decision: selects positive consumption value.

Conditions for reset() Method

- Agent makes more than decision_max decisions.
- Agent chooses a non-positive consumption value.
- The reset() method reinitializes state and decision count.

Reinforcement Learning

Instantiating planner and Neural Network

- Instantiate a planner environment.
- Define a neural network using TensorFlow.
- Use the Sequential model.
- One dense layer with relu activation.
- Output layer should have n_capital nodes.

Define Custom Reinforcement Learning Environment

```
# Instantiate planner environment.
env = planner()

# Define model in TensorFlow.
model = tf.keras.models.Sequential()
model.add(tf.keras.layers.Flatten(input_shape=(1,) + env.observation_space.shape))
model.add(tf.keras.layers.Dense(32, activation='relu'))
model.add(tf.keras.layers.Dense(n_capital, activation='linear'))
```

Hyperparameters and Training

- Define environment and network.
- Use SequentialMemory for a "replay buffer" of 50,000 decision paths.
- Model uses epsilon-greedy policy:
  - epsilon = 0.30.
  - Maximize utility 70% of the time.
  - Random decision for exploration 30% of the time.

DQNAgent Model Training

- Set hyperparameters of the DQNAgent model.
- Compile the model.
- Perform training.

Set Model Hyperparameters and Train.

```
# Specify replay buffer.
memory = SequentialMemory(limit=10000, window_length=1)

# Define policy used to make training-time decisions.
policy = EpsGreedyQPolicy(0.30)
```

Set Model Hyperparameters and Train.

```
# Define deep Q-learning network (DQN).
dqn = DQNAgent(model=model, nb_actions=n_capital,
    memory=memory, nb_steps_warmup=100,
gamma=0.95, target_model_update=1e-2,
policy=policy)

# Compile and train model.
dqn.compile(tf.keras.optimizers.Adam(0.005), metrics=['mse'])
dqn.fit(env, nb_steps=10000)
```

Monitoring the Training Process

- Two main observations from the training process:
  - Number of decisions per session increases across iterations.
  - Implication: Agent learns not to sharply draw down capital.
- Loss declines while the average reward starts to increase.
- Agent approaches optimality.

Reinforcement Learning

Summary

- Presents an alternative to standard computational economic methods.
- Deep Q-learning networks (DQN) in TensorFlow:
  - Can solve higher dimensional models.
  - Works in non-linear settings.
- Benefits:
  - No need to change model assumptions.
  - Minimizes numerical error introduction.

3. Applications

Applications

Colab Tutorial

- Generative Models
- Theoretical Models

References I

Athey, S. and G.W. Imbens (2019) "Machine Learning Methods Economist Should Know About," *Annual Review of Economics*, 11, 685–725.

Athey, S., G.W. Imbens, J. Metzger, and E. Munro (2019) "Using Wasserstein Generative Adversarial Networks for the Design of Monte Carlo Simulations," Working Paper 3824.

Goodfellow, I., Y. Bengio, and A. Courville (2017) *Deep Learning*, Cambridge, MA: MIT Press.

Hull, Isaiah (2021) *Machine Learning for Economics and Finance in TensorFlow 2*: Apress, 10.1007/978-1-4842-6373-0.

Kaji, T., E. Manresa, and G. Pouliot (2018) "Deep Inference: Artificial Intelligence for Structural Estimation," working paper.

References II

Kingma, D.P. and M. Welling (2019) "An Introduction to Variational Autoencoders," *Foundations and Trends in Machine Learning*, 12 (4), 307–392.

Krohn, J., G. Beyleveld, and A. Bassens (2020) *Deep Learning Illustrated: A Visual, Interactive Guide to Artificial Intelligence*: Addison-Wesley.

Palmer, N.M. (2015) *Individual and Social Learning: An Implementation of Bounded Rationality from First Principles* Ph.D. dissertation, George Mason University, Fairfax, VA.

Sutton, R.S. and A.G. Barto (1998) *Reinforcement Learning: An Introduction*, Cambridge: MIT Press.