Classificadores e Sistemas Conexionistas

Trabalho Prático - OpenAI-Gym

Hugo Gonçalves (PG38928)
Luis Bouça (PG38933)
Diogo Rodrigues (PG37150)
Luis Costa (A74819)

Março, 2019

Área de Aplicação: Classificadores Sistemas Conexos
Palavras-chave: *OpenAI*, *gym*, *tensorflow*

Conteúdo

Lista de Figuras

Introdução

Para o primeiro momento de avaliação da cadeira de CSC foi-nos proposto que através dos ambientes do *toolkit* gym do OpenAI desenvolveremos agentes capazes de jogar diversos jogos. No nosso caso para a realização do trabalho escolhemos o jogo do ***CartPole v1*** e o do ***Pendulum v0***.
Para tal tivemos que aplicar conceitos e ferramentas que fomos aprendendo ao longo das aulas. A ferramenta básica e mais importante foi o tensorflow onde o objetivo era criar redes neuronais que treinassem o nosso agente de modo a conseguir jogar. Iremos então ao longo deste relatório explicar as diversas fases de desenvolvimento, iremos explicar a criação dos nossos dados para o treino da nossa rede, iremos explicar o modelo criado e no fim mostraremos também os resultados obtidos e as nossas considerações.

1 *CartPole v1*

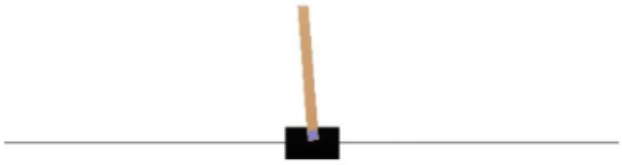

Figura 1: CartPole

Este jogo faz parte de um conjunto de ambientes jogáveis que o OpenAi gym nos fornece. Neste caso o objetivo passa por conseguir manter um pêndulo em equilíbrio num carro em andamento. Algumas características importantes deste jogo são:

- 4 variáveis observação
- 2 ações possíveis, 0(esquerda), 1(direita)
- Recompensa sempre positiva, +1
- O jogo acaba sempre que o pêndulo se desviar 15º do centro ou o carro andar 2.4 unidades a partir do centro.

Num	Action
0	Push cart to the left
1	Push cart to the right

Num	Observation	Min	Max
0	Cart Position	-2.4	2.4
1	Cart Velocity	-Inf	Inf
2	Pole Angle	~ -41.8°	~ 41.8°
3	Pole Velocity At Tip	-Inf	Inf

Figura 2: CartPole

1.1 Dados de treino

A nossa estratégia para o treino da rede passou por uma abordagem simples. Ou seja, a ideia passa por jogarmos um conjunto de jogos aleatórios, e verificar no final quais foram os jogos que obtiveram um score maior ou igual a um treshold por nós definido.

Para tal definimos como sendo o nosso treshold 50, e para termos uma boa base de jogos para termos dados de treino mais variados definimos 10000 jogos aleatórios que o nosso agente iria jogar. Como podemos ver na 5 por cada iteração ele vai realizar uma ação totalmente aleatória. De acordo com cada ação tomada ele vai receber entre várias variáveis a que nos interessa mais que é o *Reward*. Sempre que ele realiza um jogo armazena-o numa lista de jogos.

No final quando o nosso agente já jogou todos os jogos vamos iterar através da nossa lista e ver qual dos jogos é que são elegíveis para serem armazenados ao nosso array que irá conter os jogos para treinar a nossa rede. Como podemos ver na figura 4 apenas se o score daquela jogada tiver sido igual ou superior ao treshold que tínhamos definido é que irá adicionar ao array de treino.

```
training_data = []
scores = []
accepted_scores = []
for _ in range(inital_games):
    score = 0
    game_memory = []
    prev_observation = []
    for _ in range(goal_steps):
        action = env.action_space.sample()
        #action = random.randrange(0,2)
        observation, reward, done, info = env.step(action)
        if len(prev_observation) > 0:
            game_memory.append([prev_observation, action])
        prev_observation = observation
        score += reward
        if done:
            break
```

Figura 3: Criação dos dados treino

```
        if score >= score_requirement:
            accepted_scores.append(score)
            for data in game_memory:
                if data[1] == 1:
                    output = [0,1]
                elif data[1] == 0:
                    output = [1,0]

                training_data.append([data[0], output])
        env.reset()
        scores.append(score)
    training_data_save = np.array(training_data)
    np.save('saved.npy', training_data_save)
```

Figura 4: Criação dos dados treino - 2

1.2 Modelo

Neste jogo usamos o *tflearn* que é uma API de alto nível do tensorflow. Para a construção do modelo usamos a função *fully connected* que implementam redes neuronais convolucionais.

Como pudemos ver na figura **??** nós definimos 6 camadas, sendo a ultima a camada de output onde irá imprimir a ação, neste caso sendo 0 ou 1. Em cada camada definimos o numero de unidades(neurónios) e o tipo de função de ativação que queremos. Para este caso a função de ativação que usamos foi *relu* nas 5 primeiras camadas e a *softmax* na camada de output.

Depois de termos os parâmetros definidos basta simplesmente recorrer à classe *DNN* do *tflearn* que simplesmente passamos a nossa rede e neste caso como queremos que nos construa gráficos passamos o $tensorboard_verbose.Comousamosotflearnémaisfácildecriargráfico$

```
def neural_network_model(input_size):
    network = input_data(shape=[None, input_size, 1], name='input')

    network = fully_connected(network, 128, activation='relu')
    network = dropout(network,0.8)

    network = fully_connected(network, 256, activation='relu')
    network = dropout(network,0.8)

    network = fully_connected(network, 512, activation='relu')
    network = dropout(network,0.8)

    network = fully_connected(network, 256, activation='relu')
    network = dropout(network,0.8)

    network = fully_connected(network, 128, activation='relu')
    network = dropout(network,0.8)

    network = fully_connected(network, 2, activation='softmax')
    network = regression(network, optimizer = 'adam', learning_rate=LR, loss='categorical_crossentropy', nar

    model = tflearn.DNN(network ,tensorboard_verbose=3)

    return model
```

Figura 5: Criação dos dados treino

1.3 Treino do modelo

Agora que já temos os nossos dados para alimentar a rede e já temos o nosso modelo criado só falta treinar o modelo. Esta parte é relativamente simples, apenas temos que definir tal como costumamos as nossas variáveis de treino, de uma lado o $\boldsymbol{X}$ *que representa as variáveis de observação e o Y que representa a nossa variável alvo, a ação. Depois de já termos definidas as nossas variáveis tudo que precisamos de fazer é usar a função fit do nosso modelo e já temos então a nossa rede neuronal treinada de acordo com os nossos dados de treino.*

```
def train_model(training_data, model=False):
    X = np.array([i[0] for i in training_data]).reshape(-1, len(training_dat
    y = [i[1] for i in training_data]

    if not model:
        model = neural_network_model(input_size = len(X[0]))

    model.fit({'input':X}, {'targets':y}, n_epoch=3, snapshot_step=500, show
    return model
```

Figura 6: Treino da rede

1.4 Jogo

No final de já termos então a nossa rede treinada vamos fazer o nosso agente jogar o jogo de maneira mais autónoma e precisa. Não varia muito da forma que nós usamos para jogar jogos aleatórios, apenas em vez de usarmos ações aleatórias vamos usar a nossa rede neuronal para prever qual a melhor ação a tomar consoante a observação do ambiente.

Como podemos ver na figura 8 a nossa ação irá ser calculada através da função predict do nosso modelo, onde lhe passamos a observação anterior para tentar prever qual a melhor ação a tomar a seguir. No final consoante as ações tomadas e a respetiva recompensa obtemos a média dos resultados.

```
            action = np.argmax(model.predict(prev_obs.reshape(-1, len(prev_o
        choices.append(action)
        new_observation, reward, done, info = env.step(action)
        prev_obs = new_observation
        game_memory.append([new_observation, action])
        score += reward
        if done:
            break
    scores.append(score)
print('Average score: ', mean(scores))
print('Choise 1: {}, Choise 0: {} '.format(choices.count(1)/len(choices), ch
```

Figura 7: Previsão da ação a tomar

1.5 Resultados

Os resultados são claramente melhores do que simplesmente se o robô tomasse decisões aleatórias porém são muito instáveis. Tanto conseguimos obter um resultado muito fraco, na ordem dos 100 como no jogo a seguir podemos atingir uma pontuação de 500. Isto claro deve-se quer à qualidade dos dados de treino quer também ao modelo criado. Porém achamos na mesma que os resultados foram satisfatórios visto que a ideia era conseguir fazer com que o nosso agente jogasse o jogo de maneira mais autónoma e isso foi conseguido.

```
            action = np.argmax(model.predict(prev_obs.reshape(-1, len(prev_o
        choices.append(action)
        new_observation, reward, done, info = env.step(action)
        prev_obs = new_observation
        game_memory.append([new_observation, action])
        score += reward
        if done:
            break
    scores.append(score)
print('Average score: ', mean(scores))
print('Choise 1: {}, Choise 0: {} '.format(choices.count(1)/len(choices), ch
```

Figura 8: Previsão da ação a tomar

1.6 Gráficos

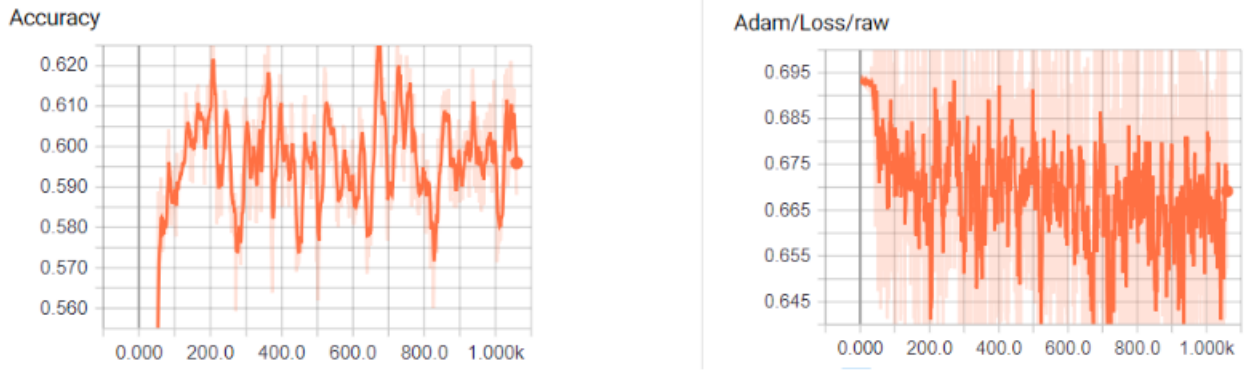

Figura 9: Variação da Accuracy e Loss ao longo das iterações

Como podemos observar nos gráficos da figura 9 não existe um crescimento constante na accuracy do nosso modelo o que pode explicar o porquê de termos tido resultados tão instáveis. Podemos ver que a nossa accuracy começa com valores abaixo dos 56% e consegue atingir um máximo de 62% enquanto que a Loss desceu dos 69.5% para os 64.5%.

Estes foram os melhores resultados que conseguimos obter usando diferentes tipos de valores quer para os parâmetros da rede quer para os valores de treshold com que guardamos os dados de treino. Como melhoria da nossa rede poderíamos futuramente usar técnicas de otimização de hiperparâmetos tais como algoritmos genéticos, grid search, etc.

2 *Pendulum v0*

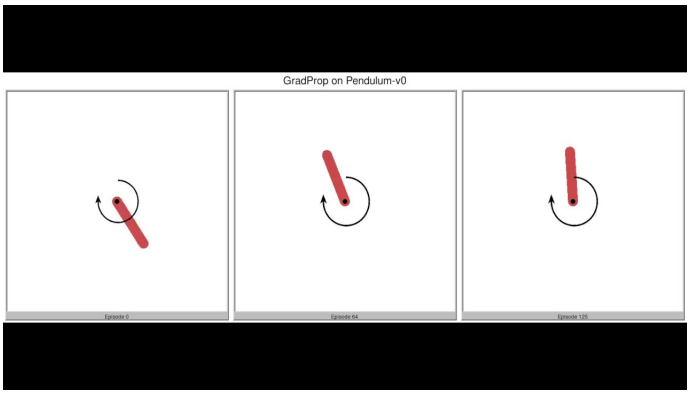

Figura 10: Pendulum v0

2.1 Descrição

A construção de um modelo de machine learning que seja capaz de jogar o pêndulo, implica um certo conhecimento dos mecanismos de jogo, do ambiente e dos dados gerados pelo mesmo.

O objetivo deste jogo é manter o pêndulo na parte superior por um maior número de iterações possível, e assim obter um maior ganho. Para isso, é necessário ter em conta as ações a tomar, de modo a que seja aplicada a força necessário sem que o pêndulo se destabilize. É necessário ter em conta que as ações aplicadas ao pêndulo, vão influenciar o ambiente em que o pêndulo está inserido, ou seja, as informações obtidas do ambiente de jogo.

De acordo com a OpenAI wiki, as ações possíveis de aplicar no pêndulo, são qualquer valor continuo entre -2 e 2, e com essa informação podemos determinar que estamos perante um problema de regressão/previsão, e que difere do modelo de classificação aplicado no jogo Cartpole, em que apenas era necessário que a machine learning tivesse

a capacidade de classificar as classes 0 e 1. Neste caso a necessidade de prever um valor continuo recorrendo a técnicas estatísticas.

O ambiente de jogo refere, a posição em que o pêndulo está inserido no plano, e é dado pelas variáveis do seno cosseno e tangente.

A posição do pêndulo, juntamente com a ação a tomar influenciam o reward obtido.

Sendo assim os rewards determinam a qualidade do jogo e da ação decorrida. Os rewards obtidos variam entre ***-16.2736044*** *representando o custo mais baixo, e* ***0****, representando o custo mais elevado.*

Num	Action	Min	Max
0	Joint effort	-2.0	2.0

Figura 11: Ações tomadas no jogo

A figura 11, mostra a tabela de ações possíveis no jogo pêndulo.

Num	Observation	Min	Max
0	cos(theta)	-1.0	1.0
1	sin(theta)	-1.0	1.0
2	theta dot	-8.0	8.0

Figura 12: Ambiente do jogo

Na figura 12, mostra a tabela de observações, contendo as variáveis de ambiente pelo qual o jogo é composto.

3 Dados de Treino

A qualidade nos dados de treino, implica a capacidade da nossa machine learning conseguir ou não, uma solução ou aproximação para o nosso problema, neste caso a solução é a capacidade da máquina em jogar de forma autónoma e eficaz.

A aprendizagem da máquina, é feita recorrendo à experiência de jogos passados, ou seja, a máquina deve aprender com a experiência, a sequência de ações a escolher, face ao ambiente atual do jogo.

No entanto é necessário ter em conta que a sequência de jogos passados com os quais a máquina vai aprender, devem ser exemplos de bons jogos. Para isso foi definido um threshold, a partir do qual as jogadas devem ser guardadas. O threshold definido deve ter uma margem suficiente para que as jogadas aprendidas não fiquem

demasiado especificas só para um certo tipo de ações. Foi assim definido um threshold de ***-15.2736044****, valor que é ligeiramente inferior ao custo ótimo, e a partir do qual as jogadas vão ser guardadas no conjunto de treino.*

O número de jogos aleatórios a realizar, permite que a proporção dos jogos aproveitados seja maior. O nosso conjunto de dados foi obtido com base em ***25000*** *jogos realizados.*

Features		
cos(theta)	sin(theta)	theta

Figura 13: Atributos do conjunto

É assim possível observar na figura 13, que o conjunto vai ser como atributos, as variáveis do ambiente de jogo existentes na figura 12.

Label	
Min:	-2
Max:	2

Figura 14: Label de previsão

A figura 14, é a label que a machine learning vai prever, que representam as ações a serem tomadas pela máquina.

O conjunto de treino vai servir como input para o treino da máquina, no entanto os dados devem antes ser normalizados, permitindo assim tirar um melhor proveito durante a fase de treino. A necessidade de realizar uma normalização aos dados, está relacionada com a variabilidade dos mesmos, ou seja, existe uma grande quantidade de dados com diferentes escalas. Apesar da máquina aprender e testar com dados normalizados, as suas previsões vão ser utilizadas na forma não normalizada.

3.1 Rede Neuronal

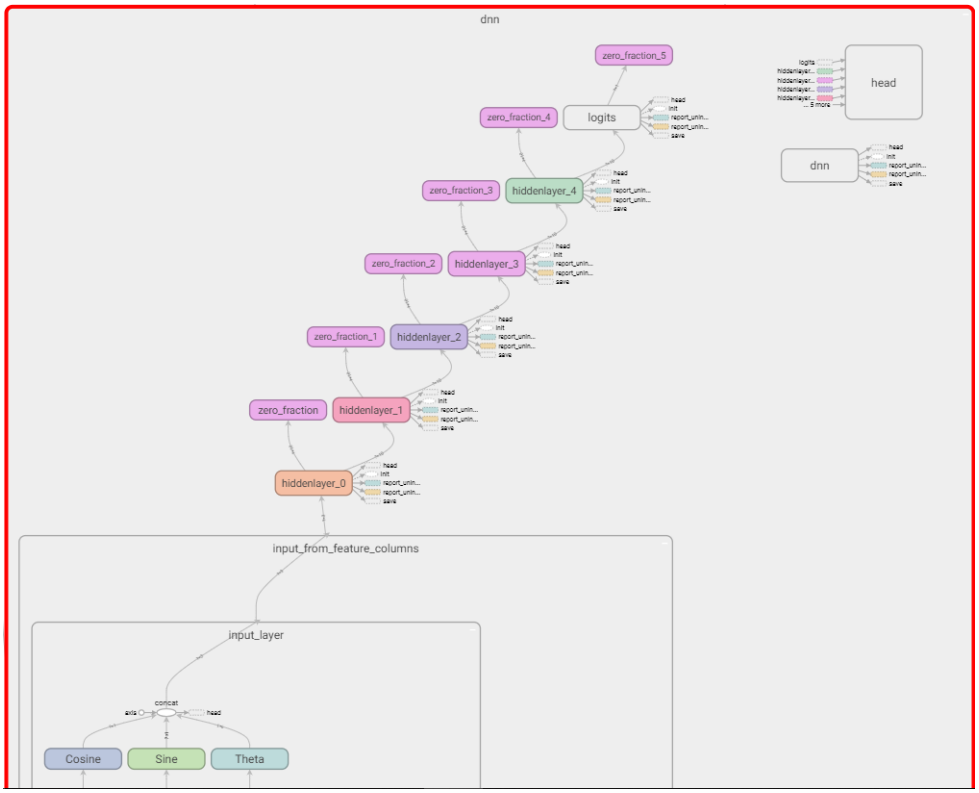

Figura 15: Estrutura da rede

A figura 18, apresenta a estrutura da rede neuronal implementada. Definiu-se assim uma rede constituída por 5 camadas, com 10 nodos cada, seria o suficiente para criar um modelo capaz de obter resultados satisfatórios.

A escolha dos parâmetros tanto na rede como no treino da mesma, mostrou-se um processo importante, isto porque a existência de uma rede complexa, pode levar a casos de criarmos um modelo que se encontra bastante ajustado ao conjunto de treino, e pouco preparado para casos de teste ***Overfitting****. O mesmo acontece no caso do treino, em que se definirmos um número de épocas, e de steps inadequado ao nosso dataset, podemos cair em situações de* ***Underfitting*** *ou* ***Overfitting*** *do modelo, e a situações em que o próprio tempo despendido no treino da rede é demasiado demorado e desnecessário.*

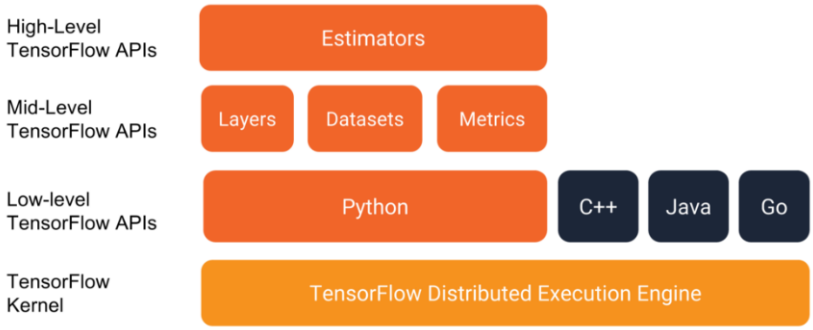

Figura 16: Níveis do Tensorflow

A construção da rede teve por base o uso de estimadores, que são APIs de alto nível do tensorflow, conforme é possível observar na figura 16.

A vantagem no uso deste tipo de APIs, é que as mesmas apresentam todas as funcionalidades implementadas, sendo apenas necessário indicar alguns parâmetros que vão definir a estrutura da própria rede. A desvantagem neste tipo de APIs, é a grande abstração que as mesmas criam, o que pode ser uma desvantagem no caso do problema que estamos a tentar solucionar ser mais complexo.

```
return tf.estimator.DNNRegressor(hidden_units=[10 for i in range(5)],
                                 feature_columns=features,
                                 activation_fn=tf.nn.relu,
                                 model_dir="/tmp/Pendulum",
                                 config=tf.contrib.learn.RunConfig(
                                       save_checkpoints_steps=250,
                                       save_checkpoints_secs=None,
                                       save_summary_steps=500))
```

Figura 17: DNNRegressor

A figura 18, mostra o estimador ***DNNRegressor****, que serviu de base para a criação da nossa rede.*

```
return tf.estimator.DNNRegressor(hidden_units=[10 for i in range(5)],
                                 feature_columns=features,
                                 activation_fn=tf.nn.relu,
                                 model_dir="/tmp/Pendulum",
                                 config=tf.contrib.learn.RunConfig(
                                       save_checkpoints_steps=250,
                                       save_checkpoints_secs=None,
                                       save_summary_steps=500))
```

Figura 18: DNNRegressor

3.2 Resultados

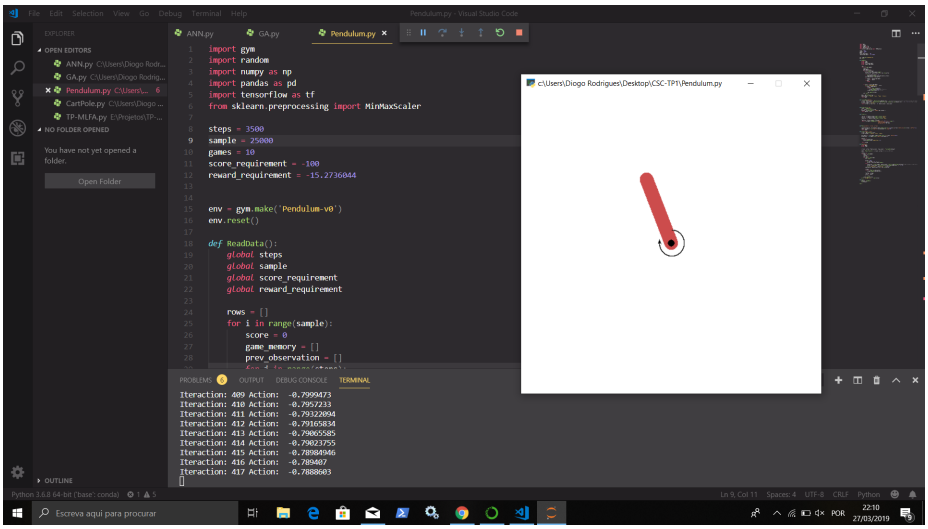

Figura 19: Resultos

O modelo criado tem a perceção das ações a tomar para obter os melhores rewards, e assim levar o pêndulo para a parte superior.

A figura 19, mostra a tentativa do pêndulo em chegar a parte superior em apenas 400 iterações, de um total de 3500 iterações definidas para cada jogo.

No entanto, apesar do modelo criado ser capaz de chegar à parte superior, qualquer que seja a sua posição inicial, ainda não tem a capacidade de se manter por um grande número de iterações. O motivo de isto ocorrer, deve-se ao facto de que o treino devia ser mais especifico, contento casos em que o as ações realizadas são mínimas, e as recompensas obtidas são grandes, ou seja, situações que descrevem o momento em que o pêndulo se encontra na parte superior, e ele deve se tentar equilibrar para continuar a obter recompensas elevadas.

4 Conclusão

A realização deste trabalho permitiu ter uma noção de como o uso de redes neuronais pode ser utilizado para solucionar problemas onde existe um grande conjunto de dados que pode ser adaptado de modo a classificar ou prever a ocorrência de novos dados.

A maior dificuldade na realização do trabalho, foi a escolha ideal dos parâmetros, que permitem um desempenho relativamente bom da rede. Como trabalho futuro e para obter um melhor desempenho, seria interessante aplicar alguns algoritmos de otimização, como algoritmos genéticos, bayseanos ou grid search, de forma a encontrar os parâmetros que melhor se encaixam na rede para fazer face ao problema que estamos a tentar solucionar.

Apesar dos jogos no qual estes modelos aplicados serem relativamente simples, é possível ter uma noção dos conceitos que são necessários para criar um modelo que tente solucionar problemas mais complexos, ou seja, jogos de maior complexidade, que requerem um maior número de ações e implementam um nível de detalhe mais elevado no ambiente em que se inserem.