Navigation

You are welcome to use this coding environment to train your agent for the project. Follow the instructions below to get started!

1. Start the Environment

Run the next code cell to install a few packages. This line will take a few minutes to run!

In [1]:

```
!pip -q install ./python
```

```
tensorflow 1.7.1 has requirement numpy>=1.13.3, but you'll have numpy 1.1
2.1 which is incompatible.
ipython 6.5.0 has requirement prompt-toolkit<2.0.0,>=1.0.15, but you'll ha
ve prompt-toolkit 3.0.8 which is incompatible.
```

The environment is already saved in the Workspace and can be accessed at the file path provided below. Please run the next code cell without making any changes.

In [2]:

```
from unityagents import UnityEnvironment
import numpy as np

# please do not modify the line below
env = UnityEnvironment(file_name="/data/Banana_Linux_NoVis/Banana.x86_64")
```

```
INFO:unityagents:
'Academy' started successfully!
Unity Academy name: Academy
        Number of Brains: 1
        Number of External Brains : 1
        Lesson number : 0
        Reset Parameters :

Unity brain name: BananaBrain
        Number of Visual Observations (per agent): 0
        Vector Observation space type: continuous
        Vector Observation space size (per agent): 37
        Number of stacked Vector Observation: 1
        Vector Action space type: discrete
        Vector Action space size (per agent): 4
        Vector Action descriptions: , , ,
```

Environments contain ***brains*** which are responsible for deciding the actions of their associated agents. Here we check for the first brain available, and set it as the default brain we will be controlling from Python.

In [3]:

```
# get the default brain
brain_name = env.brain_names[0]
brain = env.brains[brain_name]
```

2. Examine the State and Action Spaces

Run the code cell below to print some information about the environment.

In [4]:

```
# reset the environment
env_info = env.reset(train_mode=True)[brain_name]

# number of agents in the environment
print('Number of agents:', len(env_info.agents))

# number of actions
action_size = brain.vector_action_space_size
print('Number of actions:', action_size)

# examine the state space
state = env_info.vector_observations[0]
print('States look like:', state)
state_size = len(state)
print('States have length:', state_size)
```

```
Number of agents: 1
Number of actions: 4
States look like: [ 1.          0.          0.          0.          0.8440
8134  0.          0.
  1.          0.          0.0748472   0.          1.          0.
0.
  0.25755     1.          0.          0.          0.          0.74177343
  0.          1.          0.          0.          0.25854847  0.
0.
  1.          0.          0.09355672  0.          1.          0.
0.
  0.31969345  0.          0.        ]
States have length: 37
```

3. Take Random Actions in the Environment

In the next code cell, you will learn how to use the Python API to control the agent and receive feedback from the environment.

Note that **in this coding environment, you will not be able to watch the agent while it is training**, and you should set `train_mode=True` to restart the environment.

In [5]:

```
env_info = env.reset(train_mode=True)[brain_name] # reset the environment
state = env_info.vector_observations[0]            # get the current state
score = 0                                          # initialize the score
while True:
    action = np.random.randint(action_size)        # select an action
    env_info = env.step(action)[brain_name]        # send the action to the environment
    next_state = env_info.vector_observations[0]   # get the next state
    reward = env_info.rewards[0]                   # get the reward
    done = env_info.local_done[0]                  # see if episode has finished
    score += reward                                # update the score
    state = next_state                             # roll over the state to next time s
tep
    if done:                                       # exit loop if episode finished
        break

print("Score: {}".format(score))
```

```
Score: 0.0
```

When finished, you can close the environment.

In [ ]:

```
env.close()
```

4. It's Your Turn!

Now it's your turn to train your own agent to solve the environment! A few **important notes**:

- When training the environment, set `train_mode=True` , so that the line for resetting the environment looks like the following:

  ```
  env_info = env.reset(train_mode=True)[brain_name]
  ```

- To structure your work, you're welcome to work directly in this Jupyter notebook, or you might like to start over with a new file! You can see the list of files in the workspace by clicking on ***Jupyter*** in the top left corner of the notebook.
- In this coding environment, you will not be able to watch the agent while it is training. However, ***after training the agent***, you can download the saved model weights to watch the agent on your own machine!

In [6]:

```
import random
import torch
import numpy as np
from collections import deque
import time
import matplotlib.pyplot as plt
%matplotlib inline
```

In [7]:

```
# reset the environment
env_info = env.reset(train_mode=True)[brain_name]

# number of agents in the environment
print('Number of agents:', len(env_info.agents))

# number of actions
action_size = brain.vector_action_space_size
print('Number of actions:', action_size)

# examine the state space
state = env_info.vector_observations[0]
print('States look like:', state)
state_size = len(state)
print('States have length:', state_size)
```

```
Number of agents: 1
Number of actions: 4
States look like: [ 0.          0.          1.          0.          0.1610
1955  1.          0.
  0.          0.          0.04571758  1.          0.          0.
0.
  0.2937662   0.          0.          1.          0.          0.14386636
  0.          0.          1.          0.          0.16776823  1.
0.
  0.          0.          0.04420976  1.          0.          0.
0.
  0.05423063  0.          0.        ]
States have length: 37
```

In [8]:

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class QNetwork(nn.Module):
    def __init__(self, state_size, action_size, seed, fc1_units=64, fc2_units=64):
        super(QNetwork, self).__init__()
        self.seed = torch.manual_seed(seed)
        self.fc1 = nn.Linear(state_size, fc1_units)
        self.fc2 = nn.Linear(fc1_units, fc2_units)
        self.fc3 = nn.Linear(fc2_units, action_size)

    def forward(self, state):
        x = F.relu(self.fc1(state))
        x = F.relu(self.fc2(x))
        return self.fc3(x)
```

In [14]:

```
import numpy as np
import random
from collections import namedtuple, deque

import torch
import torch.nn.functional as F
import torch.optim as optim

BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64         # minibatch size (default: 64)
GAMMA = 0.9965          # discount factor (default: 0.99)
TAU = 1e-3              # for soft update of target parameters
LR = 1e-4               # learning rate
UPDATE_EVERY = 4        # how often to update the network

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

class Agent():
    def __init__(self, state_size, action_size, seed):
        self.state_size = state_size
        self.action_size = action_size
        self.seed = random.seed(seed)

        # Q-Network
        self.qnetwork_local = QNetwork(state_size, action_size, seed, 1024, 1024).to(de
vice) #set the NN nodes to 1024
        self.qnetwork_target = QNetwork(state_size, action_size, seed, 1024, 1024).to(d
evice) #set the NN nodes to 1024
        self.optimizer = optim.Adam(self.qnetwork_local.parameters(), lr=LR)

        # Replay memory
        self.memory = ReplayBuffer(action_size, BUFFER_SIZE, BATCH_SIZE, seed)
        # Initialize time step (for updating every UPDATE_EVERY steps)
        self.t_step = 0

    def step(self, state, action, reward, next_state, done):
        # Save experience in replay memory
        self.memory.add(state, action, reward, next_state, done)

        # Learn every UPDATE_EVERY time steps.
        self.t_step = (self.t_step + 1) % UPDATE_EVERY
        if self.t_step == 0:
            # If enough samples are available in memory, get random subset and learn
            if len(self.memory) > BATCH_SIZE:
                experiences = self.memory.sample()
                self.learn(experiences, GAMMA)

    def act(self, state, eps=0.):
        state = torch.from_numpy(state).float().unsqueeze(0).to(device)
        self.qnetwork_local.eval()
        with torch.no_grad():
            action_values = self.qnetwork_local(state)
        self.qnetwork_local.train()

        # Epsilon-greedy action selection
        if random.random() > eps:
            return np.argmax(action_values.cpu().data.numpy())
        else:
            return random.choice(np.arange(self.action_size))
```

```
    def learn(self, experiences, gamma):
        states, actions, rewards, next_states, dones = experiences

        # Get max predicted Q values (for next states) from target model
        Q_targets_next = self.qnetwork_target(next_states).detach().max(1)[0].unsqueeze
(1)
        # Compute Q targets for current states
        Q_targets = rewards + (gamma * Q_targets_next * (1 - dones))

        # Get expected Q values from local model
        Q_expected = self.qnetwork_local(states).gather(1, actions)

        # Compute loss
        loss = F.mse_loss(Q_expected, Q_targets)
        # Minimize the loss
        self.optimizer.zero_grad()
        loss.backward()
        self.optimizer.step()

        # ------------------- update target network ------------------- #
        self.soft_update(self.qnetwork_local, self.qnetwork_target, TAU)

    def soft_update(self, local_model, target_model, tau):
        for target_param, local_param in zip(target_model.parameters(), local_model.par
ameters()):
            target_param.data.copy_(tau*local_param.data + (1.0-tau)*target_param.data)


class ReplayBuffer:
    def __init__(self, action_size, buffer_size, batch_size, seed):
        self.action_size = action_size
        self.memory = deque(maxlen=buffer_size)
        self.batch_size = batch_size
        self.experience = namedtuple("Experience", field_names=["state", "action", "rew
ard", "next_state", "done"])
        self.seed = random.seed(seed)

    def add(self, state, action, reward, next_state, done):
        e = self.experience(state, action, reward, next_state, done)
        self.memory.append(e)

    def sample(self):
        experiences = random.sample(self.memory, k=self.batch_size)

        states = torch.from_numpy(np.vstack([e.state for e in experiences if e is not N
one])).float().to(device)
        actions = torch.from_numpy(np.vstack([e.action for e in experiences if e is not
None])).long().to(device)
        rewards = torch.from_numpy(np.vstack([e.reward for e in experiences if e is not
None])).float().to(device)
        next_states = torch.from_numpy(np.vstack([e.next_state for e in experiences if
e is not None])).float().to(device)
        dones = torch.from_numpy(np.vstack([e.done for e in experiences if e is not Non
e]).astype(np.uint8)).float().to(device)

        return (states, actions, rewards, next_states, dones)

    def __len__(self):
        return len(self.memory)
```

In [10]:

```
agent = Agent(state_size=37, action_size=4, seed=42)
```

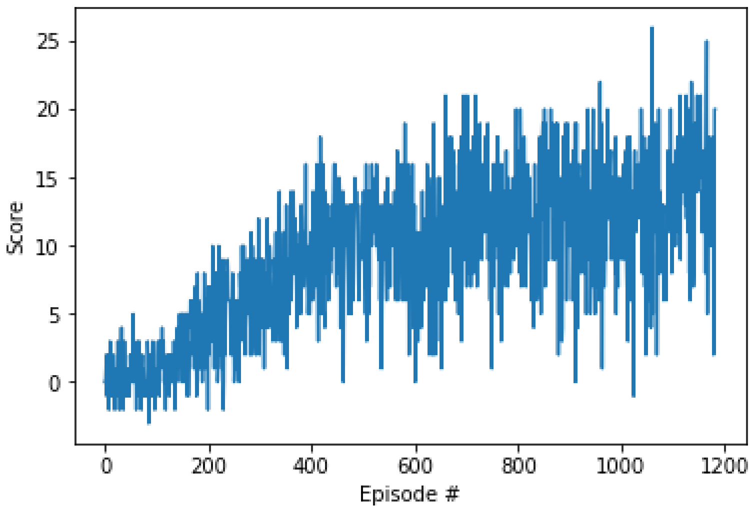
In [18]:

```
def dqn(n_episodes=1200, max_t=1000, eps_start=1.0, eps_end=0.01, eps_decay=0.995):
    scores = []                        # list containing scores from each episode
    scores_window = deque(maxlen=100)  # last 100 scores
    eps = eps_start                    # initialize epsilon
    for i_episode in range(1, n_episodes+1):
        #state = env.reset()
        env_info = env.reset(train_mode=True)[brain_name]  # reset the environment
        state = env_info.vector_observations[0]            # get the current state

        score = 0
        for t in range(max_t):
            action = agent.act(state, eps)
            env_info = env.step(action)[brain_name]        # send the action to the env
ironment
            next_state = env_info.vector_observations[0]   # get the next state
            reward = env_info.rewards[0]                   # get the reward
            done = env_info.local_done[0]                  # see if episode has finishe
d

            agent.step(state, action, reward, next_state, done)
            state = next_state
            score += reward
            if done:
                break
        scores_window.append(score)       # save most recent score
        scores.append(score)              # save most recent score
        eps = max(eps_end, eps_decay*eps) # decrease epsilon
        print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(scores_wi
ndow)), end="")
        if i_episode % 100 == 0:
            print('\rEpisode {}\tAverage Score: {:.2f}'.format(i_episode, np.mean(score
s_window)))
        if np.mean(scores_window)>=14.0:
            print('\nEnvironment solved in {:d} episodes!\tAverage Score: {:.2f}'.forma
t(i_episode-100, np.mean(scores_window)))
            torch.save(agent.qnetwork_local.state_dict(), 'checkpoint.pth')
            break
    return scores
```

In [14]:

```
scores = dqn()

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores)
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.savefig("DQN.jpg")
plt.show()
```

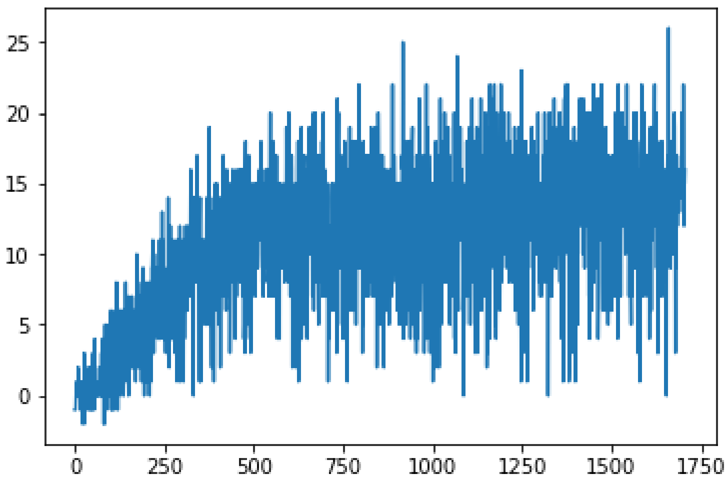
```
Episode 100	Average Score: 0.43
Episode 200	Average Score: 2.24
Episode 300	Average Score: 5.10
Episode 400	Average Score: 7.85
Episode 500	Average Score: 10.15
Episode 600	Average Score: 10.63
Episode 700	Average Score: 10.53
Episode 800	Average Score: 12.32
Episode 900	Average Score: 12.21
Episode 1000	Average Score: 12.58
Episode 1100	Average Score: 12.20
Episode 1182	Average Score: 14.13
Environment solved in 1082 episodes!	Average Score: 14.13
```



In [13]:

```
# USEING DOUBLE & DUELING

agent = Agent(state_size=state_size, action_size=action_size, seed=0)
use_double=True
use_dueling=True
eps_decay=0.98
eps_end=0.02

scores= dqn()

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='scores')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left');
plt.savefig("DNQ_DUELING.jpg")
plt.show()
```

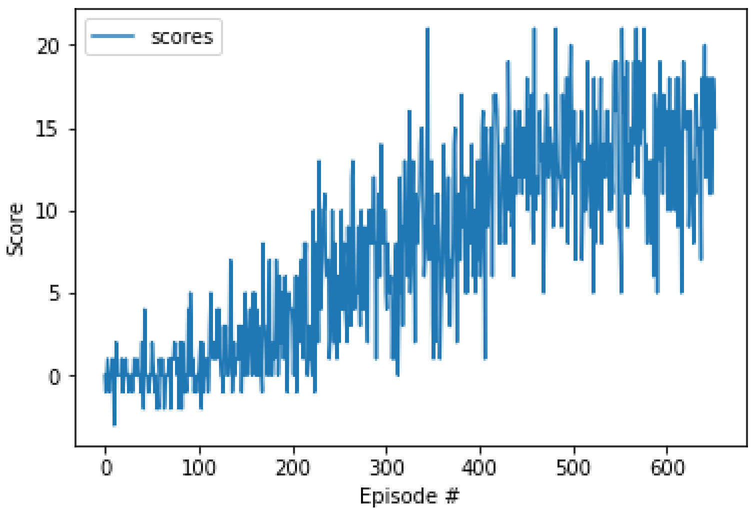
```
Episode 100	Average Score: 0.61
Episode 200	Average Score: 3.71
Episode 300	Average Score: 6.71
Episode 400	Average Score: 9.45
Episode 500	Average Score: 10.86
Episode 600	Average Score: 11.89
Episode 700	Average Score: 11.85
Episode 800	Average Score: 12.06
Episode 900	Average Score: 12.09
Episode 1000	Average Score: 11.67
Episode 1100	Average Score: 12.31
Episode 1200	Average Score: 12.61
Episode 1300	Average Score: 12.96
Episode 1400	Average Score: 12.83
Episode 1500	Average Score: 13.75
Episode 1600	Average Score: 13.46
Episode 1700	Average Score: 13.77
Episode 1706	Average Score: 14.03
Environment solved in 1606 episodes!	Average Score: 14.03

---------------------------------------------------------------------------
-
NameError                                 Traceback (most recent call las
t)
<ipython-input-13-0e1c32788842> in <module>()
     13 ax = fig.add_subplot(111)
     14 plt.plot(np.arange(len(scores)), scores, label='scores')
---> 15 plt.plot(np.arange(len(scores)), avgs, c='r', label='average')
     16 plt.ylabel('Score')
     17 plt.xlabel('Episode #')

NameError: name 'avgs' is not defined
```



In [15]:

```
# With Batchsize Change to half 32 from 64 default
# USEING DOUBLE & DUELING


BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 32        # minibatch size (default: 64)
GAMMA = 0.9965          # discount factor (default: 0.99)
TAU = 1e-3              # for soft update of target parameters
LR = 1e-3               # learning rate
UPDATE_EVERY = 4        # how often to update the network

agent = Agent(state_size=state_size, action_size=action_size, seed=0)
use_double=True
use_dueling=True
eps_decay=0.98
eps_end=0.02

scores= dqn()

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='scores')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left');
plt.savefig("DNQ_DUELING_32.jpg")
plt.show()
```

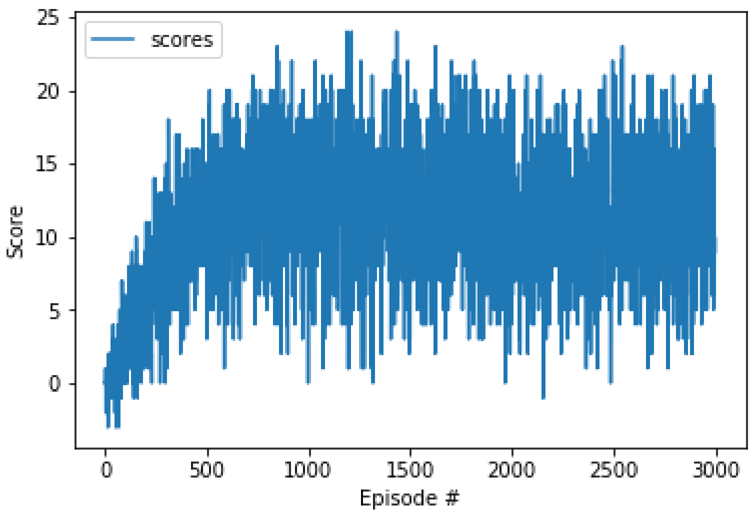
```
Episode 100     Average Score: 0.15
Episode 200     Average Score: 1.94
Episode 300     Average Score: 5.86
Episode 400     Average Score: 8.20
Episode 500     Average Score: 12.82
Episode 600     Average Score: 13.35
Episode 653     Average Score: 14.03
Environment solved in 553 episodes!     Average Score: 14.03
```



In [16]:

```
# With EPS CHANGE
# NOT USEING DOUBLE & DUELING

BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 64        # minibatch size (default: 64)
GAMMA = 0.9965          # discount factor (default: 0.99)
TAU = 1e-3              # for soft update of target parameters
LR = 1e-3               # learning rate
UPDATE_EVERY = 4        # how often to update the network

agent = Agent(state_size=state_size, action_size=action_size, seed=0)
use_double=False
use_dueling=False
eps_decay=0.98
eps_end=0.100

scores= dqn()

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='scores')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left');
plt.savefig("DNQ_EPS_.jpg")
plt.show()
```

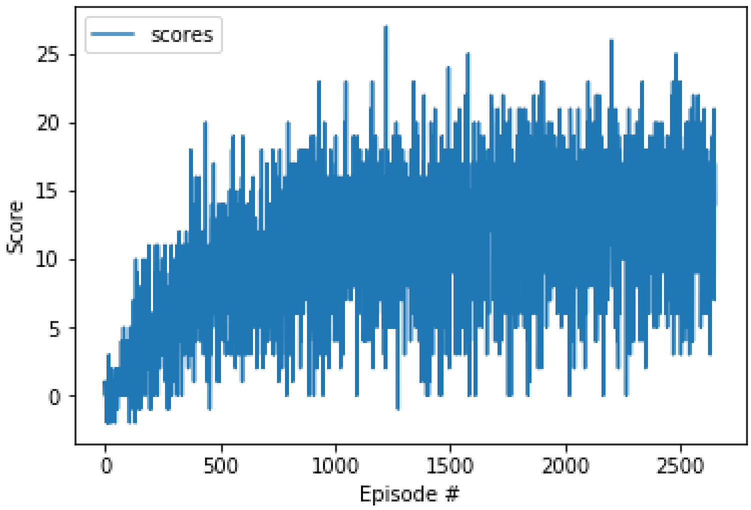
```
Episode 100	Average Score: 0.94
Episode 200	Average Score: 4.07
Episode 300	Average Score: 6.58
Episode 400	Average Score: 8.93
Episode 500	Average Score: 10.86
Episode 600	Average Score: 11.87
Episode 700	Average Score: 11.99
Episode 800	Average Score: 13.12
Episode 900	Average Score: 12.92
Episode 1000	Average Score: 12.81
Episode 1100	Average Score: 12.90
Episode 1200	Average Score: 12.73
Episode 1300	Average Score: 12.76
Episode 1400	Average Score: 12.42
Episode 1500	Average Score: 12.61
Episode 1600	Average Score: 11.61
Episode 1700	Average Score: 12.91
Episode 1800	Average Score: 13.76
Episode 1900	Average Score: 11.90
Episode 2000	Average Score: 11.30
Episode 2100	Average Score: 10.90
Episode 2200	Average Score: 11.26
Episode 2300	Average Score: 11.90
Episode 2400	Average Score: 11.62
Episode 2500	Average Score: 11.62
Episode 2600	Average Score: 12.50
Episode 2700	Average Score: 11.74
Episode 2800	Average Score: 11.93
Episode 2900	Average Score: 11.14
Episode 3000	Average Score: 11.94
```



In [17]:

```
# Slow Learning Rate

BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 32        # minibatch size (default: 64)
GAMMA = 0.99           # discount factor (default: 0.99)
TAU = 1e-2               # for soft update of target parameters
LR = 1e-4                # learning rate
UPDATE_EVERY = 4         # how often to update the network

# USEING DOUBLE & DUELING
agent = Agent(state_size=state_size, action_size=action_size, seed=0)
use_double=True
use_dueling=True
eps_decay=0.90
eps_end=0.001

scores= dqn()

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='scores')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left');
plt.savefig("DNQ_EPS_32.jpg")
plt.show()
```

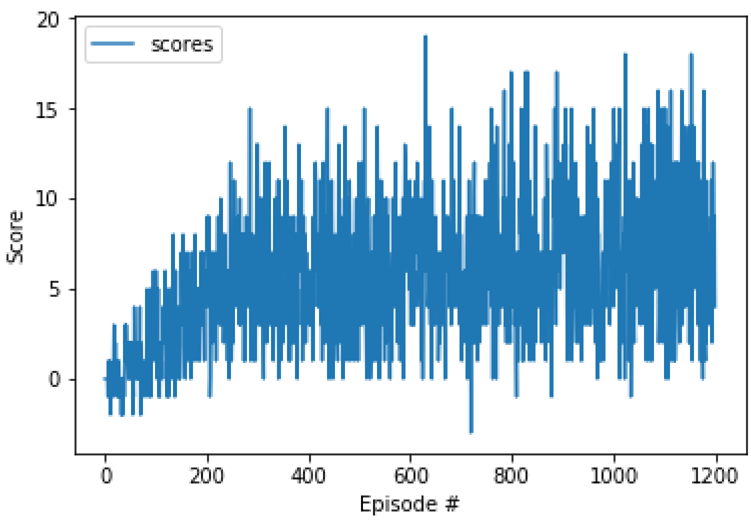
```
Episode 100	Average Score: 0.65
Episode 200	Average Score: 3.50
Episode 300	Average Score: 5.55
Episode 400	Average Score: 7.24
Episode 500	Average Score: 8.39
Episode 600	Average Score: 9.51
Episode 700	Average Score: 8.95
Episode 800	Average Score: 9.68
Episode 900	Average Score: 10.91
Episode 1000	Average Score: 10.34
Episode 1100	Average Score: 10.90
Episode 1200	Average Score: 12.28
Episode 1300	Average Score: 11.58
Episode 1400	Average Score: 11.66
Episode 1500	Average Score: 11.69
Episode 1600	Average Score: 12.04
Episode 1700	Average Score: 12.78
Episode 1800	Average Score: 12.79
Episode 1900	Average Score: 13.11
Episode 2000	Average Score: 13.40
Episode 2100	Average Score: 12.14
Episode 2200	Average Score: 13.25
Episode 2300	Average Score: 12.63
Episode 2400	Average Score: 13.23
Episode 2500	Average Score: 12.99
Episode 2600	Average Score: 13.53
Episode 2652	Average Score: 14.04
Environment solved in 2552 episodes!	Average Score: 14.04
```



In [19]:

```
# Fast Learning Rate
#NO USEING DOUBLE & DUELING
# 32 Batch Size
BUFFER_SIZE = int(1e5)  # replay buffer size
BATCH_SIZE = 32        # minibatch size (default: 64)
GAMMA = 0.99           # discount factor (default: 0.99)
TAU = 1e-2              # for soft update of target parameters
LR = 1e-2              # learning rate
UPDATE_EVERY = 4        # how often to update the network

agent = Agent(state_size=state_size, action_size=action_size, seed=0)
use_double=False
use_dueling=False
eps_decay=0.98
eps_end=0.001

scores= dqn()

# plot the scores
fig = plt.figure()
ax = fig.add_subplot(111)
plt.plot(np.arange(len(scores)), scores, label='scores')
plt.ylabel('Score')
plt.xlabel('Episode #')
plt.legend(loc='upper left');
plt.savefig("DNQ_fast_32.jpg")
plt.show()
```

```
Episode 100     Average Score: 0.70
Episode 200     Average Score: 3.00
Episode 300     Average Score: 5.40
Episode 400     Average Score: 5.75
Episode 500     Average Score: 6.01
Episode 600     Average Score: 6.02
Episode 700     Average Score: 6.46
Episode 800     Average Score: 6.54
Episode 900     Average Score: 6.78
Episode 1000    Average Score: 6.70
Episode 1100    Average Score: 7.58
Episode 1200    Average Score: 6.94
```



In [ ]:

```
# Performance Improvment Factors Factors
# Standard DQN Agent 1    1082 episodes*********
# Standard DQN + replay buffer (no double, no dueling)
# Standard DQN + replay buffer (double,dueling) *  1606 episodes***********
# Standard DQN + replay buffer (double,dueling) Batch Size 32  553 episodes***********
# Standard DQN + replay buffer (double,dueling) eps_decay=0.98 eps_end=0.1 +3000 episod
es*******
# Standard DQN + replay buffer (no double,no dueling) with decreasing learning rate epi
sodes  +2300 episodes*************
# According to results check indivisual effect by changing parameters and then try with
combination

Future Tasks:
following Parameters are close to best results as per observations
esp=0.98 to 0.01
Learning rate 1e-3
batch size 32
Replay Buffer Without Double & Dueling
DDQN
```