a snake game using reinforcement learning hey guys today i

have a very exciting project for you we are going to build an ai that teaches itself how to play snake and we will

build everything from scratch so we start by creating the game with pygame and then we build an agent and a deep

learning algorithm with pie torch i will also teach you the basics of reinforcement learning that we need to

understand how all of this works so i think this is going to be pretty cool and now before we start let me show you

the final project so i can start the script by saying

python agents dot pi now this will start training our agent

and here we see our game and then here i also plot the scores and then the

average score and now let me also start a stopwatch so

that you can see that all of this is happening live and now at this point our snake knows

absolutely nothing about the game it only is aware of the environment and tries to make some more or less random

moves but with each move and especially with each game it learns more and more

and then knows how to play the game and it should get better and better

so the first few games you won't see a lot of improvements but don't worry

that's absolutely normal i can tell you that it takes around 80 to 100 games

until our ai has a good game strategy and this will take around 10 minutes

also you don't need a gpu for this so all of this training can happen on this

cpu that's totally fine okay so let me speed this up a little bit

[Music]

[Music]

all right so now about 10 minutes have passed and we are at about game 90 i

guess and now we can clearly see that our snake knows what it should do so

it's more or less going straight for the food and tries not to hit the boundaries

so it's not perfect at this point but we can see that it's getting better and

better so we also see that the average score here is increasing and now the per the

best score so far is and to be honest for me this is super

exciting so if you imagine that at the beginning our snake didn't know anything about the game and now with a little bit

of math behind the scenes it's clearly following a strategy

so this is just super cool don't you think all right so let me speed this up a little bit more

[Music]

all right so after 12 minutes our snake is getting better and better so i think

you can clearly see that our algorithm works so now let me stop this and then let's

start with the theory so i will split the series into four parts in this first video we learn a

little bit about the theory of reinforcement learning in the second part we implement the actual game or

also called the environment here with pygame then we implement the agent so i will

tell you what this means in a second and in the last part we implement the actual model with pytorch

so let's start with a little bit of theory about reinforcement learning

so this is the definition from wikipedia so reinforcement learning is an area of

machine learning concerned with how software agents ought to take actions in

an environment in order to maximize the notion of cumulative reward so this

might sound a little bit complicated so in other words we can also say that reinforcement learning is teaching a

software agent how to behave in an environment by telling it how good it's doing

so what we should remember here is that we have an a chance so that's basically our computer player then we have an

environment so this is our game in this case and then we give the agent a reward so

with this we tell it how good it's doing and then based on their reward it should

try to find the best next action so yeah that's reinforcement learning

and to train the agent there are a lot of different approaches and not all of

them involve deep learning but in our case we use deep learning and this is

also called steep q learning so this approach extends reinforcement learning

by using a deep neural network to predict the actions and that's we're going to use in this tutorial

all right so let me show you the rough overview of how i organized the code so

as i said we're having four parts so in the next part we implement the game with

pie game then we implement the agent and then we implement the model with pie

torch so our game has to be assigned such that we

have a game loop and then with each game loop we do a play step

that gets an action and then it does a step so it moves the

snake and then after the move it returns the current reward and if we are game

over or not and then also the current score then we have the agent and the agent

basically puts everything together so that's why it must know about the game

and it also knows about the model so we store both both of them in our agent and

then we implement the training loop so this is roughly what we have to do

so based on the game we have to calculate a state

and then based on the state we um calculate the next action

and this involves calling model predict and then with this new action

we do a next play step and then as i said we get a reward the game overstate

and the score and now with this information we calculate a new state

and then we remember all of this so we store the new state and the old state

and the game over state and the score and with this we then train our model

so for the model i call this linear q net so this is not too complicated this

is just a feed forward neural net with a few linear layers and it needs to

have the these information so the new state and the old state and then we can

train the model and we can call model predict and then this gets us the next action so

yeah this is a rough overview how the code should look like and now let's talk about some

of those variables in more detail for example the action or the state or the

reward so let's start with the reward so that's pretty easy so whenever our snake eats a food we

give it a plus 10 reward when we are game over so when we die then we get -10

and for everything else we just stay at zero so that's pretty simple

then we have the action so the action determines our next move

so we could think that we have four different actions so left right

up and down but if we design it like this then for example what can happen is if we go

right then we might take the action left and then we immediately die so this is

basically a 180 degree turn so we don't allow that so a better

approach to design the action is to only use three different numbers

and now this is dependent on the current direction

so um 1 zero zero means we stay in the current direction so we go straight so

this means if we go right then we stay right if we go left then we go left and

so on then if we have 0 1 0 this means we do a

right turn and again this depends on the current direction so if we go right and

do a right turn then we go down next if we go down and do a right

turn again then we go left and then again we would go up so this is the right turn and the left

turn is the other way around so if we go left and do a left turn then we go down

and so on so with this approach we cannot do a 180 degree turn and we also

we only have to predict three different states so this will make it a little bit

easier for our model so now we have the reward and the action

then we also need to calculate the state and the state means that we have to tell

our snake some information about the game that it knows about so it needs to

know about the environment and in this case our state has 11 values

so it has the information if the danger is straight or if it's ahead if the

danger is right or if the danger is left then it has um the current direction so

direction left right up and down and then it has the information if the

food is left or right or up or down and all of these are boolean values so

let me show you an actual example so in this case if we are going right and our food is

here then we see um danger straight right and left none of this is true

so for example if our snake is over here at

this end and it's still going right then danger straight would be a one

so this again also depends on the current direction for example if we move

up at this corner here then danger right would be a1

then for these directions only one of them is one and the rest is always zero

so in this case we have danger right set to one and then for this in our case our food

is right of the snake and also down of the snake so food right is one and food

down as one all right so now with the state and the action we can design our

model so this is just a feed forward neural net with an input layer a hidden

layer and an output layer and for the input it gets the state so as i said we

have 11 different numbers in our state 11 different boolean values zero or one

so we need this size 11 at the beginning then we can choose a hidden um size

and for the output we need three outputs because then we predict the action so

this can be some numbers and these don't need to be probabilities

so here we can have raw numbers and then we simply choose the maximum so for

example if we take 1 0 zero and if we go back then we

see this would be the action straight so keep the current direction

so yeah that's how our model looks like and of course now we have to train the

model so for this let's talk a little bit about this deep q learning

so q stands for this is the q value and this stands for the quality of the action

so this is what we want to improve so each actions should improve the quality

of the snake so we start by initializing the q value

so in this case we initialize our model with some random parameters

then we choose an action by calling model predict state and we also

sometimes choose a random move so we do this especially at the beginning when we

don't know a lot about the game yet so and then later we have to do a trade-off

when we don't want to do a random move anymore and only call model predict

and this is also called a trade-off between exploration and exploitation

so this will get clearer later when we do the actual coding so then with this new action we perform

this action so we perform the next move and then we measure the reward and with

this information we can update our q value and then train the model and then

we repeat this step so this is an iterative training loop so now to train the model

as always we need to have some kind of loss function that we want to optimize

or minimize so for the loss function we have to look at a little bit of math and

for this i want to present you the so-called belmont equation so this might look scary so don't be

scared here i will explain everything and actually it's not that difficult when we um understand this and then code

this later so what we want to do here we need to update the q value as i said here so

according to the belmont equation the new q value is calculated like this so

we have the current q value plus the learning rate and then we have the

reward for taking that action at that state plus a gamma parameter which is called

this count rate so don't worry about this i will also show this later in the code again

and then we take the maximum expected future reward given the new state and all possible

actions at that new state so yeah this looks scary but i will

simplify that for you and then it's actually not that difficult so the old q value is model predict with

state 0 so if we go back at this overview so the first time we say

get state from the game this is our state 0 and then after we took this place step

we again measure or calculate the next state so this is then our state one

so with this information again our first queue is just model predict with the old state

and then the new queue is the reward plus our gamma value

times the maximum value of the q state so again this is model predict

but this time we take state one and then with these two information our

loss is simply the q new minus q squared

and yeah this is nothing else than the mean squared error so that's a very

simple error that we should already know about and then this is what we must use in our optimization

so yeah that's what we are going to use so we have to implement all of these three classes

and in the next video we start by implementing the game

Part 2: Setup environment and implement snake game

in the last part i showed you all the necessary theory that we need to know to get started with deep q learning

and now we start implementing all of the parts so as i said we need to have a

game so the environment then we need an agent and we need a model

so in this part we start by implementing the game and we use pytorch for this

so let me actually start by creating a environment and we install all the

necessary dependencies that we need so in this case i use conda to manage

the environments and if you don't know how to use conda then i have a tutorial

for you that i will link here but yeah if you don't want to use connor you can also just use a normal virtual

and but i recommend to use a virtual and and now let's create a virtual and with

conda create minus n and then give it a name for example pi game n and i also

say i want python equals 3.7

all right so now this was created so now we want to actuate it with conda

activate and then pie game n and hit enter and then we see the name of the

environment in the front so this means that we activated it successfully and

now we can start installing all what we need so the first thing we want to

install is pie game for our game so pip install pie game and hit enter

so this is done the next thing we need is pytorch for our model later so for

this we can go to the official home page and on install and then here you can select your

operating system so i use mac and i actually i want to say pip install

and we don't need cuda support so only a cpu is fine

and we don't need torch audio because we don't work with audio files so we can only grab this pip install torch torch

vision and then paste it in here and hit enter and now this installs pytorch and all

the dependencies all right so this is done and then we need two more things for plotting later

so for this i say pip install much plot lip and we also want i python

and then hit enter all right so this was successful as well

and now we have everything we need so now we can start implementing all the codes and as a starting point i want to

grab the code from another tutorial that i did so you can find this on github and

then on my um account and then in the repo python fun

and here i actually have two snake games so and then we need this one snake pie

game and download this so you can do this and i already did this and have this here so

if we open up the editor here i'm visuals using visual

studio code then we can see we have exactly those two files

and then um the first thing i want to do is i want to run this file and test if this

is actually working so right now this is just a normal

snake game that you have to control yourself so you have to use the arrow keys so let's say python

snake game dot pi and then let's hope that this is working so yeah so now i

can control the snake and i hope that i can eat the food yes

and now if i hit the boundary then we are game over so this is working our environment is

set up and now we can start implementing our code so we can change this so that

we can use this as a ai controlled game so let me show you the overview from

last time so last time i told you that we need a play step in our game and this

gets an action and based on this action we then take a move and then we must

return a reward the game over state and the current score so

first let's write down all the things that we need to change here

so first we want to have a reset function so after each

um game our agent should be able to reset the game and start with a new game

then we need to implement the reward that our agent gets

then we need to change the play function so that it takes takes an action and

then um returns a or computes the direction

then we also want to keep track of the current frame or let's call this

game iteration and for later we also need to have a

change in the if is collision function to check if this is a collision

so first let's let me go over this code quickly so what

we do here is we use pi game then for the direction we use an enum

then for the point we use a named tuple and then here i created a class snake

game and here we initialize the things we need for the game

so here we initialize the game state for example for the snake we use a

list with three initial values and the head is always the front of this

list then we keep track of the score and here we have a helper function to place the

food and yeah and we already have a function that is called play step

and then if we go down to the very end so here we have our game loop so while this is true we take

a game or a play step and we get the game over state and the

score so this place the function is the most important one so here first we

right now we grab the user input so the key we press

then we calculate a move based on this key and then we update our snake and check

if we are game over and if we can continue we place the new food

or check if we eat the food and we update our ui with this

helper function update ui then here we have this helper function is collision

where we check if we either hit the boundary or we run into ourself

and then we also have this helper function move where we get the current

direction and then based on this direction we simply um calculate

calculate the new position of the new hat so yeah that's all um

what is done here and now let's change a few things though so the first one i

want to change the class name to say snake game ai to make it clear that this

is a agent controlled game and now so the first thing we want is

the reset functionality so in here um i already have this comment where we in it

the game state so now we want to refactor all of this into a

reset function so we create a new function define and then let's call this reset and it

only gets self and no other arguments and here we can grab all of this code

and then simply paste it in here and in our

initializer we then call self dot reset so this is the first thing we need

additionally we want to keep track of the um game iteration or frame iteration so

let's call this self dot frame iteration

and in the beginning this is just zero then this define place food can stay as

it is and now we need to change the play step function so first of all

if we have a look at the overview here i already told you that now we need

to give this the action from the agent and we need to return a reward so let's

start by um using this action parameter

and here we grab the user input so actually right now we can get rid of

this so the only thing we still check if we want to quit the game and now here um

we already have this helper function where we move

in the current direction so actually what we change here now this

move function doesn't get the direction from the user input so now here it gets

the action and then we have to determine the new direction so we do this in a second but first

let's only change this and then here we call the self.move

with the action and then we update the head then we

check if we are game over or not and we actually now we also need the reward so

we simply say reward equals zero and let's go back to the slides from

last time so the reward is really simple whenever we eat a food we say plus 10

when we lose or when we die then we say our reward is -10 and for everything

else we just stay at zero so we initialize the reward with zero

then if we have a collision and game over then we say our

reward equals to -10 and we want to return this as well so return the reward

game over and self.score and here we check only if we have a

collision so here i actually want to do another check so if

nothing happens for a long time so if our snake doesn't improve and doesn't

eat the food but also doesn't die then we also want to check this

and if this happens for a too long time then we also break here

so we can say or and then here we say if self dot frame iteration

and if that this gets too large without anything happening then we um stop here

so here i use this little formula if this is greater than 100 times the

length of our snake so remember this is a list then we break so this is also like this

then it's dependent on the length of the snake so the longer our snake is the more time

it has so but then if it gets larger than this value then we break

and of course we have to update the self.frame iteration and we

can simply do this here at the beginning so for each play step we say self dot frame

iteration plus equals 1

and when we reset it then we reset it back to zero

so this is here and then yeah if we stop we have the reward -10

then here if our hat hits the food then we eat the food so

our score increases and our reward is set to plus 10

then we place a new food and say otherwise we remove the last part so we

simply move here then this can stay as it is the update function and at the very end we also

want to return the reward then for the is collision function we need a

slight change so here i only check for self.head but later um to calculate the

state or the danger which i told you about so if we

have a look at the state so here we calculate the danger so if we are for example if we

are here at the corner then we have a danger at the right

so for this it might be handy if we don't use self.head inside here

but if we give this function a point so this gets the point argument and let's

say by default this is none and then here we simply ch check if the

point is none then we set the point equals to

self dot head so inside this where we call this with

no argument it can stay as it is and then here of course we have to

change self.head to this is now our point so here if we hit the

corner point here and point here and point

here then we have a collision and here if our

point is in the snake body then we also have a collision and otherwise we don't have a collision

all right so the update ui function can stay like this

and now for the move function here we need to change something so now we get a

action and now based on this action we want to determine the next move

so if we go back to the slides so here we designed the action like this so it

has three values um one zero zero means we keep the

current direction and go straight 0 1 0 means we do a right turn and 0 0 1 means

we do a left turn so this is dependent on the current

direction so if we go right and do a right turn then we go down next if we go

down and do a right turn then we go left next and so on and left turn is the

other way around so now um we want to determine the

direction based on the action so let's write a quick comment here we

have straight right turn or left turn so

to get the next direction first i want to

define all the possible directions in a clockwise order so we say clockwise

equals and then a list and here we start with direction dot right so

here remember for the direction we use this enum class

so it has to be one of those directions so our

clockwise directions should start with direction right then from this on the

next one is direction dot down then we have direction dot

left and as last thing we have direction dot up so right down left up this is

clockwise and then to get the current direction or the current index of the

current direction we say index equals and then we can say clockwise dot index

and then the index of the self dot direction so we are sure that this has

to be in this array we because the self direction must be one of those enum values

and then we check that different um possible states so these ones

so for this we can use numpy and i guess we have to import numpy first as np

and then we can use it here we can say if numpy and then we use this function

array equal and then here we put in the action and the array that we want to

compare so if this is equal to one zero zero

then we go straight or we keep the current directions so we simply say

our new direction equals and then clockwise

of the index and then remember the index is just the index of the current direction

so here we basically have no change then we say

l if if our array if numpy array equal if the action

equals to 0 one zero then we do a

right turn so this means we go clockwise so if we

go right then the next direction would be down if we go down then the next

direction would be left and if we go left then the next direction would be up so here we say

index equals or this is our next

index actually and here we say this is the current index plus

1 but then modulo 4 so this means if we are at the end up and then do the next

one if we have index so this is index 0 1 2 3 and then if we

have index 4 modulo 4 is actually again index

zero again so from this we do a turn and then come back at the front again

so this is our right turn so now this is the next index and now our new direction is

clockwise of the next index

and then otherwise we can simply use else here and actually change this to an

l if so now this is the last case so it has to be here it has to be

zero zero one and if this is the case then let's copy

and paste this in here then our next index is the current index

minus one modulo four so this actually means we go counter clockwise so we do a

left turn so if we start with right then the next move would be up and then the

next would be left and then the next would be down and then right again and so on so now

this is our new direction and then simply we say self direction

equals new direction and then we go on so here we extract the

head and then here we have to check if self dot's direction now is right then we

increase the position of x and so on if we um have the left

direction then we decrease x and if we go down then we actually

increase y so for so the y starts at the top at zero and

then increases if we go down so if we go down then we have to increase y and if

we go up then we have to decrease y so if self direction equals up

then y minus equals the block size and by the way the block size is just here a

constant value of 10 so that's how big our one block of the snake should be in

pixels so yeah this is everything we need here in the move function and now here

we don't need this anymore so this is actually no longer working with

a user input so you can just delete this and then later we control

this class from the agent and call this play step function

so yeah for now this is all we need to implement the game

Part 3: Implement agent to control game

so i already talked about the theory of deep q learning in the first part

in the last part we implemented the pi game so that we can use it in our

agent controlled environment and now we need the agent so let's start and

so here um if you haven't watched the first two parts then i highly recommend to do so

so this is the starting point from last time and i actually want to make one

more change that i forgot so here the is collision function should

actually be public because then our agent should use it

so just remove the underscore here and then also remove it in this class itself

when we call this so then we have our snake game and i also want to rename this to just be game

and now we create a new file agent dot pi and then start implementing this

so first here we import torch from pi torch then we import random because when later

we need this then we also need import numpy snp

and from our um implemented class we need the snake game so we say from

game import snake like snake game a i

so i think that's what we call this class snake game a

i so yeah that's the right name then we also hear at the beginning we defined

this enum for the direction and this named tuple for the point which has an x

and a y attribute so we also want to import these two um

things so we import direction and we import point and then we also say from

collections we want to import deck so this is a data

structure where we want to store our memories so um if you don't know what a deck is then

i will put a link in the description below so this is really handy in this case and

you will see why this is the case later and then here i want to define some

parameters as constants so we have a maximum memory of let's say 100

000 so we can store 100 000 items in this memory

then we also want to to use a batch size that you will see later and here i will

set this to 1000 so you can play around with these parameters

and i also want a learning rate later and i want to set this to 0 0 1

and yeah feel free to change this and then we start creating our class agent

and it gets of course an init function with self and no other

arguments and then let's have a look at the slides from the first part where i

explained the training so we want to create a training function

where we do all of this so we need to get the state calculate the state

where we are aware of the current environment then we need to calculate

the next move from the state and we need to

um then we want to update or do the next step and call game.playstep and then

calculate the new state again then we want to store everything in memory and

then we also want to train our model so we need to store the game and the model

in this class so first of all let me create the functions that we need first so we need

a function get state which gets self and this this gets the game

and then we calculate the state that i showed you with these 11 different

variables then we want to have a function that we call remember

remember and it has self and here we want to put in the state then the action

then we want to remember the reward for this action and we want to calculate or

we want to store the next state next state

and we also want to store done or bit or you can also call this game over so this

is the current game overstate then we need two different functions to train

and we call this defined train on the long memory and it only needs

self so i will explain this later and we also let's copy and paste this i

also have a function define train on short memory so this is only with one

step you will see this later then we need a function and we call this

get action to get the action based on the state so it gets self and the state

and first we only say pass and these are all the functions we need

i guess and then i want to have a global function that i call simply train

and here we say pass and then when we start this module h and dot pi so we say

if name underscore equals equals main then we simply call this train

function and then we can start the script by saying python agent dot pi

like i did in the very first tutorial so let's start implementing the agent and

the training function so let's start with the init function of the agent so

here what i want to store is first i want to store some more parameters so

self.number of games so i want to keep track of this so this is zero in the

beginning then self.epsilon equals um zero in the beginning this is

a parameter to control the randomness so you will see this later

then we also need self dot gamma equals zero

so this is this is the so-called this count rate

which i briefly showed in the first tutorial i will explain this a little bit more in the next tutorial where we

implement the model and the actual deep q learning algorithm then we want to

have a memory so we say self.memory equals and for this we use

this stack and this can have a argument max leng equals

and here we say max memory and what then happens if we exceed this

memory then it will automatically remove elements from the left so then it will

call pop left for us and that's why this deck is really handy here

and then later here we also want to have our model and the trainer so i will

leave this for a or s to do for the last part in the next video and now this is

all for the init function and now we can go back and now let's do the training

function next so again let's have a look at these slides so we need

these functions in this order so let's first let's write some comments

of first let's create some lists to keep track of the scores so this is an empty

list in the beginning and this is used for plotting later so then we also want to keep track of

the mean scores or average scores this is also an empty list in the beginning

then our total score equals zero in the beginning

our record our best score is zero in the beginning then we set up a agent so agent equals

an agent and we also need to create a game so the

game is a snake game ai object

and then here we create our training loop so we say while true so this should

basically run forever until we quit the script and now here let's write some comments

so we want to get get the old state or the current state

so here let's say state old equals and then we call agent dot

get states and this gets the game so we already set this up correctly we only have to

implement it then then after this we want to get the move

based on this current state so we say the final move equals agent

dot get action so we actually called this action and the action is based on the state

then with this move we want to perform the move and then and get new state so

for this we say reward um done and score

equals and here we call game dot play step from last time

so i think game dot play step with the action yes

game dot play step and this gets the final move

and then we get the state old or the new now the new state state

new state new equals agent and again gets state now with the new game

then after that we want to train the short memory of the agent so only for

one step so for this we say agent agent dot train

short memory and this gets if we have a look here um

actually uh this short memory should get some parameters so exactly the same as

we put in the remember function so train short memory gets all of those variables

and then here when we call this now we should get some hints strain

or let's save this file and then say agent dot

train short memory and now we should get the hints no we don't get this but

actually we want to have the state action reward next state and done

so here let's do this so say let's say state old then the action which was the

final move then the reward then the state new and adds last thing

the done or game over state variable so now we have this then we want to

remember all of these and store this in the memory so we say agent dot remember

and then here it gets the same um variables so we want to store all of

this in our deck and then this is all we need so now we check

if done or if game over then if this is

true then what we want to do is um we want to let's write a comment train

the long memory and this is also called replay memory or experience replay and

this is very important for our agent so now it trains again on all the previous

moves and games that it played and this tremendously helps him to

improve itself and we also here want to plot the

results so first of all we want to reset the game so we can simply do this by

saying game dot reset we already have this function here so this initializes the game state

and resets everything so the score the snake the frame iteration and places the

initial snake and the food so now we have this then we want to

increase agents dot number of games so this

plus equals one then we want to say agent dot train long

memory and this doesn't need any arguments then we want to check if we have a new high score so if score

greater than the current record then we say record equals our new score

and we will also want to leave this as a to do here so here we want to say agent

dot model dot save later when we have the model

and so here in the here we want to store this as self.model

and now what we also want to do here um let's print some information so print

the game and then the current number and then the score and the record

so here let's say our game is agent dot n

games then we also want to plot the or print the score so this is just the score

and we want to print the current record so the record equals record

and then here we want to do some plotting so i will implement this in the

next tutorial so i will leave this s8 to do so this is all for our training function

so what i showed in the slides and now of course we have to implement

those functions so for the get state function um let's go back to this

overview and here as i said we store 11 values so

if the danger is straight right or left then the current

direction so only one of these is one and then the position of the food if

it's left of the snake right of the snake up or down of the snake so these are the 11 states and now let

me actually copy and paste the code in here so that i don't make any

mistakes but we will go over this so first let's grab the head from this

game so we can do this by calling game dot snake zero so this is a list and the

first item is our head then um let's create some points

next to this head in all directions that we need to check if this hits the

boundary and if this is a danger so for this we can use this named tuple

so we can create a point with this location but minus 20 so the

20 is hard coded here so this is the number that i used for the block size

so like this we create four points around the head then the current

direction is simply a boolean where we check if the current game direction

equals to one of those so only one of those is one and the

other one is zero or false and then

um we create this array or this list with this 11 um

states so here we check that if the danger is straight or ahead

and this is dependent on the current direction so if we are going right

and the point right of us gives us a collision

then we have a danger the same or or if

we go left and our left point gets a collision then we also

have a danger here and so on so this is dangerous straight and then danger right

means if we go up and the point right of us would give a

collision then we have a danger for a right turn basically

and so on and the same for the left so this might be a little bit tricky so i

recommend that you pause here and go over this logic for yourself again

so yeah these only have give us three values in our state so far

then we have the move direction where only one of them is true and the other

one is false and for the food location we simply check if food if game food x is

smaller than game head x then we have food is left of us

and the same way we check for right up and down and then we convert our list to

a numpy array and say the data type is in so this is a nice little trick to

convert this true or false booleans to a zero or one

so yeah now this is the get state method now let's move on to the remember

function so here we want to remember all of this in our memory so this is a deck

and this is very simple so here we say self dot memory and then the deck has

also the append function where we want to append all of this

in this order so the state the action the reward the next state and the game

over state and as i said if this exceeds the maximum memory

then pop left if max mem

memory is reached and yeah this is the remember

function then let's start implementing the train long and short memory functions so for

this so we actually we store a model and a trainer in here so let's actually say

self dot model equals let's say this is only none in the beginning and leave a to do

and self dot trainer equals none in the beginning and this is a to do

so these are objects that we create in the next tutorial

and then here we call this trainer to actually do the

optimization so let's start here so for only one step

we say self.trainer and then this should get a function that we call let's call this

train step and then it gets all of these variables

so the state the action the reward the next state and the game overstate and

this is all that we need to train it for only one game step and we design this function um so that

it takes either only one state like this but it can also take a whole tensor or a

numpy array and then uses multiple as a so-called batch so let's do this here so

for this we take the variables from our memory so here we want to grab a batch and so

in the beginning we defined the batch size is 1 000 so we want to grab 1 000

samples from our memory but first we check if we um already have

a thousand samples in our memory so we say if lang and self dot memory if

this is smaller then the batch size then we simply

or actually let's say if this is greater so if this is greater than we want to

have a random sample and say mini sample equals and then we want to get a

random sample so we can use random dot sample so we already imported the random

module random dot sample from self dot

memory and as a size it should have the batch size

so this will return a list of tuples and

this is because here i forgot one important thing so when we want to store

this and append this we want to append this as only one element so only one

tuple so we need extra parenthesis here so this is one tuple that we store

and then here we get the batch size number of tuples

and otherwise else if we don't have uh

a thousand elements yet then we simply take the whole memory so we say mini

sample equals self dot memory and

then we again want to call this training step and for this so let's call this

here again self.trainer.trainstep but here we have

multiple states so let's call this states actions rewards next states and

done and right now so now we have it in

this format that we have one tuple after another

and now we want to extract this from the mini sample and then put every states

together every action together every reward to it together and so on and this

is actually a really simple with python so we can say we want to extract the

states the actions the rewards the next

states and the duns game overs and here we simply use the built in sip

function and have to use one asterisk and then the mini sample argument

so yeah check that for yourself if you don't know how the sip function works

but again it now it puts every states together every actions and so on if this

is too complicated for you then you can also just do a for loop so you can

iterate over this mini sample and basically say for

action or for state action

rewards next state and done in one mini sample

and then again you call this here for only one for only one

argument so yeah you can do it both ways but actually i recommend to do it this

way because then you have this as only one argument and then you can do this

faster in pytorch all right so now we have both the training functions now we

only need the get action function so here in the beginning we want to do some

random moves and this is also called a trade-off between

exploration and exploitation in deep learning so at some

point or in the beginning one we want to make sure that we also make random moves

and explore the environment but then the better our

model or our agent gets the less random moves we want to have and the more we

want to exploit our agent or our model so

yeah this is what we want to do here so for this we use this

epsilon parameter that we initialized in the beginning

so for this let's implement this first so we say self dot epsilon equals and

this is dependent on the number of games so here i hard code this to 80 minus

self dot number of games you can play around with this and then let's get the

final move so in the beginning we say zero zero zero and then one of

those now has to be true so here first let's check if random dot

rent int and here between 0 and 200

if this is smaller than self dot epsilon

then we take a random move so we say move equals

random dot rant ins and this must be between 0 and 2 so the 2 is actually

included here and this will give us a random value 0 1 or 2 and now this index must

be set to one so we say final move of this move index equals one

and yeah so so the more games we have the smaller our epsilon will get

and the smaller the epsilon will get the less frequent this will be

less than the epsilon and when this is even this can even become negative and then we don't longer

have a random move so again if this was too fast here then feel free to pause

and think about this logic again so now we have that and otherwise else

so here we actually here we want to do a move that is based on our model so we

want to get a prediction prediction equals self dot model dot predict and it

wants to predict the action based on one state so we call the state zero

and we get this here but we want to convert this to a tensor so we say state

0 equals torch dot tensor and as an input

it gets the state and we also give it a data type equals

let's use a torch dot float here then we call self.model predict with the state

this will give us a prediction and this can be a raw value so if we go

back to this slide so this can be a raw value and then we

take the maximum of this and set this index to a1

so here we say our move equals and we get this by saying torch arc max and the

arc max of the prediction and this is a tensor again and to convert this to only

one number we can call the item and now this is an integer and now again we can

say final move of the smooth index is one

and now we have this so now we return the final move here

return and yeah this is all we need so now we have this and can save it like this

and now we have all that we need for our agent class and now in the next one

so what we must do here is implement the model and the trainer and then also the

Part 4: Create and train neural network

plotting

so let's go back to the code and here i left this essay to do so we need a model

and a trainer so let's create a new file and let's

call this model dot pi and then here let's first import all the

things we need so we need import torch then we want to import torch dot n n s n

n then we want to import torch dot optim s

optim and also import torch dot n n dot functional s capital f

and we also want to import o s to save our model and now we want to implement two classes

one for the model and one for the trainer so let's create a class and

let's call this linear underscore qnet and this has to inherit from nn dot

module module and by the way if you are not comfortable with pytorch and want to

learn how to use this framework then i have a beginner series here on this tutorial for free and i will put the

link in the description so this will teach you everything to need to get started with pytorch

so right now let's start implementing this linear qnet function so we need the init

function define init and we need to have self and this gets an input size

input size a hidden size and an output size

and then the first thing we want to do is to call this super initializer so we

call super in it and here um this is very simple so if we

have a look at the slides then our models should just be a feed

forward neural net with a input layer a hidden layer and an output layer

um feel free to extend this and improve this but it works fine for this case and

it's actually not that bad here so let's create two linear layers so

let's call this self.linear1 equals nn.linear

and this gets the input size as an input and then the hidden size as the output

size then we have self.linear2 equals

nn.linear and now this gets the hidden size as the input and the output size as

the output then as always in pi torch we have to implement the forward function

with self and it gets x so the tensor

and here what we want to do is first we want to apply the linear layer and we

also use an actuation function here so again if you don't know what this is then check out my beginner tutorial

series there i explain all of this so we say x and then we can call f dot

reloose we use this directly from the functional module and here we say self dot linear one with

our tensor x as the input so first we do the linear layer then we apply the

actuation function and then again we apply the second layer so we call self dot linear 2 with x

and we don't need an actuation function here at the end we can simply use the

raw numbers and return x so this is our forward function

then let's also implement a helper function to save the model later so

let's call this self safe and this gets the file name as an

input and i use a default here so we say model dot pth is simply the

file name and then the last time i think i already

called this function um so not yet but now we can comment this out so if we

have a new high score then we call agent dot model dot save and here let's create

a new folder in here so let's say this is the model folder path equals and let's

create a new folder in the current directory and call this

model so dot slash model and then we check if this already exists so the file

in this folder so we can say if not os dot path dot

exists and then we say our model folder path

then we create this so we say os dot makers and we want to make this model

folder path then we create this final file name so

we say file name equals os

dot path dot join and we want to join the model folder path and the file name

that we use here as the input now this is the file name for saving and then we

want to save this and we can do this with torch dot save and we want to save self dot

state dict so i also have a tutorial about saving the model we only need to save

this state dictionary and then as a path we use this file name

so now this is all we need for our linear q net and now to do the actual training and

optimization we also do this in a class that i call q

trainer q trainer and now here what we want to do we want

to implement a init function which gets self then it also gets the

model then it should get a learning rate and it should get a gamma parameter

and here we simply store everything self.lr equals lr self dot gamma equals

gamma and we also store the model so we say self dot model equals model

then to do a pie charge optimization step we need a optimizer so we can

create this by calling self.optim or let's call this optimizer

equals and we get this from the opt-in module and here you can choose one

optimizer so i use the atom optimizer and we want to optimize model.parameters

and this is a function and then it also needs the learning rate so lr equals self dot l r

and now we also need a criterion or a loss function so let's

call this self dot criterion equals and now if we go back to these

slides at the very end we learned in the first part that this is nothing else than the mean squared

error so that's very simple so we can create this here by saying

self.criterion equals so this is nn.mse

loss and now this is what we need in our initializer and then we also need to

define a we call this train step function which

gets self and then it needs to have all the stored um parameters from last time

so it needs to have or let's have a look at this

so here when we call this it needs the state the final move the reward the new

states and done so let's copy and paste this in here and

rename this slightly so this is just the state this is the action this is the reward

so this is the new state this can uh let's call this

next state here and then done can stay as it is

and for now let's simply do pass here and before we implement this

let's go back to the agent and now set this up so here we say from

and we call this model and we want to import the linear i think we call this

linear q net and q trainer

and then here in the initializer we want to create an

instance of the model and of the trainer so self.model equals our linear qnet and

now this needs the input size the hidden size and the output size

so here i use 11 256 and three so remember if we have a look at the

slides again um the first one is the size of the state

so this is 11 values and the output must be three because and we have three

different um three different numbers in our action

and you can play around with this hidden size but the other ones have to be eleven and three

so this is the model and the trainer equals the q trainer and this gets the

model so self.model then it gets the learning rate equals the learning rate

which we specified here and we also pass on the gamma value so

gamma equals self dot gamma and the gamma is the discount rate so i this has

to be a value that is smaller than 1 and

usually this is around 0.8 or 0.9 so in this case let's set this to 0.9

so you can play around with this as well but keep in mind that it must be smaller

than one so now we have this and then i made one error in the last tutorial so

this is very important that we fix this right now so here in the get

action function i actually i called this self.model

predict but actually pythog doesn't have a predict function so this would be the

api for tensorflow for example so in pi torch we simply call self.model

like this and then this will execute this forward

function so this is actually the prediction then so yeah

please make sure to fix this okay so now we have everything and if we

have a look and go back then we see we call this self.trainer train step

with only one parameter but also with multiple ones so we want to make sure

that we can handle different sizes so now let's start implementing this

function and now the first thing we want to do so right now this can be um either a

tuple or a list or just a single value so let's convert this to a pi torch

tensor so let me copy and paste this in here so we do this for the states the

next state the action and reward and we can do this by calling torch.tensor

and then the variable and we specify the data type to torch dot float

and we don't have to do this for the done or game over value because we don't

need this as a tensor and now we want to handle multiple

sizes so we want to check if the length and then we can

check state dot shape and if this is one then we only have one

dimension and then we want to reshape this so right now we only have

if this is the case then we only have one number but actually we want to have it in the

form one and then the values so this is the number

of um batches so if this is already if this has

already multiple values then it's already in the in the size n

and x so then it's already correct and now here we want to append one

dimension and we can do this with the torch unsqueeze function so we can say

state equals states dot or sorry not state but

torch dot un squeeze squeeze and then the

states and we want to put it in dimension

zero or axis zero so this means that it appends one dimension in the beginning

and this is then just one then i also wanted to do this for the other um

tensors so for the next state and for action and reward and the done value we also want to

convert this right now this is only a single value and we want to convert this

to a tuple so we can do it like this so now we have a done so this is how you define a tuple

with only one value and now um we have it in the correct

shape so now what we have to implement is um from last time or from the very

first tutorial where i showed this bellman equation

and then we simplified this so we have the old queue where we simply call model

predict with the old state and the new queue with this formula so

let's do this so first let's um write a comment here so

as first thing we want to get the predicted

predicted q values with the current

state and this is simply by doing let's call this prediction equals self dot model

and then we want to do this with state 0 or we just call this state here

and then for the second part we need this formula the reward plus the gamma

value times the maximum of again model predict with state one

so first let's write this as a new uh comment so the first thing is we want to

apply this formula reward plus gamma times and then

the next predicted q value and

then we want to have the maximum so the maximum of this so

maximum and then um this is a little bit tricky so the maximum of this um

sorry let's do it like this maximum of next predicted q value so this is only

one value but um if we do it like as first

um parameter the predictions this has actually this is the action this is

actually three different values so what we do to get the same here is we do

a clone of this and then we set the

index with this action to the new q value so

this is let's call this q new like i showed you in the formula

and then we set the let's call this predictions and then the

index of the arc max of the action we set this to our

q new value so again this might be tricky so again we want to calculate the new q value

with this formula that i showed you but then we need to have it in the same format and for this we simply clone this

so then we have three values again and two of the values are the same but

the value with the action so the action is for example

one zero zero so um the index of the one is then set

to the new q value so this is what we want to do here so for this let's first let's create a

clone target equals prediction dot clone so we can do

this with a pi torch tensor and then um we want to iterate over our tensors and

apply this formula so for this we say for index in

range and then the length of the let's call this done and here everything

should have the same size so then this works so now we iterate over

this and then one thing that i didn't mention so far is that we only only want

to do this only do this if not done um otherwise we simply take the whole

reward so we say q new equals reward of

the current current index and now we check if we are

not done so we say if not done and the done is of the current

index then we apply this formula so now we say q

new is actually um the reward so the reward

of the current index plus self dot

gamma and then times torch dot

max the maximum value of the next prediction so here's self

dot model of next state of this

index so this is exactly what we have written here and now we need to set the

target of the maximum value of the action to this value

so here we get the let's we call this target so the target of the current

index and then of the arc max of the action

so for this we can again say torch dot arc max of the of the action and we want

to have this as a item so as a value and not as a tensor and now this is our q

new value so this might be a little bit tricky to understand so i recommend that

that you pause here and go over this again and now we have everything that we need

so let's have a look at the slides again we have our q and our q

new and then we apply the loss function so the mean squared error and in pi torch

so what we have to do here we can simply use this optimizer and do a step

and first we have to call this zero grad function to empty the gradient so this

is just something that we have to remember in pi torch and then we calculate the loss by

calling self dot criterion and here we put in the

target and the prediction so this is q new and q

and then we call loss dot backward and apply back propagation and then update

our gradients and then we call self.optimizer.step

and this is all that we need in this training step and now this is actually

all that we need in this model file so now again let's go back to the agent

and i guess we already set up the q trainer and then when we train this we call this

train step function either for only one of those parameters

or for a whole batch and now this function can handle different sizes

and now the only thing left to do here is to actually to plot the results

so for this let's create a new file and let's call this hell

helper dot pi and then here let me actually copy and

paste this in here so this is just a simple function with

matplotlib and i python and yeah here we want to plot the scores

so this is a list and we want to plot in the plot the mean score

so let's create them so here in the agent

we say from helper import the plot function

and then down here in the training function so we already created an empty

list for the scores and for the mean scores and now after each um

game we want to append the score so let's remove the to do and implement

this so we say plot scores dot appends and then the current score

then let's calculate the new mean or average score

so for this let's say total score plus equals the score

and then let's call this mean score equals the total score divided by the

number of games so agent and games and then we append this to plot mean

scores dot append the mean score and then we simply call

the plot function with the plot scores and then the plot

mean scores and now let's save this file and also let's save this file and then let's try

it out so in the terminal let's call agent dot pi and let's cross

fingers so we have a syntax error in the model.pi

file so um here we actually here we have two equal signs

so let's fix this and save this and run it again

and then we made another mistake name error so here this is actually called

prediction.clone so again let's save this and run this

and now it starts training without crashing and it also plots

so let's let this run and see if this is improving

[Music]

all right so as we can see the algorithm works and the snake is getting better

and better and the scores are getting higher and higher and also the mean or average score is getting higher so i

forgot one important thing which i show you in a second

but for now um so the snake is not perfect and the main issues are that it traps itself

sometimes and also sometimes it gets stuck in an endless loop sequence

so this is something that you can improve as a homework so yeah like this it now it trapped itself

so yeah let me stop this actually and then show you what i forgot so in the game we can

actually um set the speed so for the human controlled game when i want to

play this i set this to 20 but now i recommend to set this to a larger number

so that the training will be faster so for example you can use 40 here or

even higher so i go with 40 and yeah i think that's the whole code you can also

find this on github and yeah i hope you really enjoyed this little series about reinforcement learning and if you

enjoyed this then please hit the like button and consider subscribing to the channel