Aufgabe 3: Pancake Sort

Teilnahme-ID: 64910

Bearbeiter dieser Aufgabe:
Jan Neuenfeld

16. April 2023

Bei dieser Aufgabe ist es gefragt, einen Stapel von Pfannkuchen nach Größe zu sortieren, indem der einzige Weg den Stapel zu ändern, eine so genannte „Wende-und-Ess-Operation“ ist. Dabei wird ein imaginärer Pfannenwender zwischen zwei Pfannkuchen geschoben, mit dem man dann den darüber liegenden Pfannkuchen-Teilstapel umdreht. Daraufhin wird der oberste Pfannkuchen gegessen und die nächste Operation kann durchgeführt werden. Hierbei ist zu beachten, dass ein Stapel mit der Größe n genau aus Pfannkuchen mit den Größen 1 bis n besteht.

1 Lösungsidee

Teil B

Eine Brute-Force Implementierung, bei der die kürzeste Lösung gefunden wird, indem jede Möglichkeit ausprobiert wird, bietet sich an, um einige Informationen hinsichtlich Aufgabenteil B zu bekommen. Dabei ergeben sich Pfannkuchen-Wende-Und-Ess-Zahlen (PWUE-Zahlen), also die minimale Menge an Wende-Und-Ess-Operationen (WUEO), die nötig ist um jeglichen Pfannkuchenstapel mit der Größe n zu sortieren, von $P(8) = 4$, $P(9) = 5$, $P(10) = 5$ und $P(11) = 6$. Dies, zusammen mit den gegebenen PWUE-Zahlen, führt zu der Hypothese, dass eine allgemeine Formel $P(n)$ etwa

$$P(n) = \lceil \frac{n}{2} \rceil$$

lauten könnte. Die zugrunde liegende Idee ist hier, dass man theoretisch mit einer WUEO zwei Pfannkuchen „sortieren“ kann, indem einer gelöscht, und der andere schon in Hinsicht auf alle folgenden WUEOs optimal platziert wird. $\frac{n}{2}$ muss dabei aufgerundet werden, da bei z.B. $\frac{3}{2} = 1.5$ die 1 natürlich eine ganze WUEO, also zwei nicht sortierte Pancakes darstellt, aber die 0.5 WUEO trotzdem nicht vernachlässigt werden darf, da sie den Dritten der drei Pfannkuchen repräsentiert. Um Diesen zu sortieren braucht es eine weitere WUEO, bei der allerdings die Hälfte des Potentials nicht verwendet wird.

Um einen „Worst-Case-Stack“, also einen Pfannkuchenstapel, der tatsächlich $P(n)$ WUEOs zum Sortieren braucht, zu generieren, ist es weder sinnvoll die 1 oben zu platzieren, noch den Pfannkuchen mit der Größe n ganz unten zu platzieren. So wäre der Stapel schon zu stark sortiert, als dass man annehmen würde, dass es einen Worst-Case darstellt. Es bietet sich an, abwechselnd große und kleine Pfannkuchen zu benutzen, da so eine minimale Anzahl an in sich sortierten Abschnitten im Stapel entstehen. Dies ist wichtig, da diese Abschnitte sonst fast als einzelne Pfannkuchen betrachtet werden könnten. Wenn also ein Stapel so aussieht: $1, 2, 3, 6, 5, 4$, kann durch eine WUEO zwischen den Pfannkuchen mit den Größen 6 und 5 (Pfannkuchen 6 und 5) genutzt werden, um Nummer 6 zu entfernen. Das Resultat sieht so aus: $3, 2, 1, 5, 4$. Durch eine weitere WUEO unter Pfannkuchen 5 wird der Teilstapel $3, 2, 1$ wieder in die richtige Ausrichtung gebracht und Pfannkuchen 5 entfernt. So ergibt sich ein finaler Stapel von $1, 2, 3, 4$. Um also diesen Effekten entgegen zu steuern, kann man den Stapel abwechselnd mit großen und kleinen Pfannkuchen aufbauen. Dabei sollte natürlich nicht der größte Pfannkuchen ganz unten sein, da das die effektive Größe des Stapels auf $n - 1$ reduzieren würde. Ein Worst-Case-Stapel von der Größe 6 würde also so aussehen: $4, 3, 5, 2, 6, 1$. Dass dieses Verfahren funktioniert, hat ein Brute-Force Test gezeigt. Dabei

konnten natürlich nicht alle möglichen Stapel aller möglichen Größen durchprobiert werden, allerdings wurde für die Größen 1 bis 8 weder eine schnellere Lösung als $P(n)$ gefunden, noch ein anderer Stapel, der mehr WUEOs gebraucht hätte. Damit ist anzunehmen, dass das Prinzip auch auf größere Stapel erfolgreich angewendet werden kann.

Teil A

Ein Brute-Force Ansatz für Aufgabenteil A ist durch die schlechte Laufzeitordnung von $\mathcal{O}(n!)$ nur bis etwa $n = 11$ in akzeptabler Zeit möglich. Um also ein Muster zu finden, mit dem man schneller Lösungen finden kann, kann man zum Beispiel ein neuronales Netz verwenden. Da es allerdings entweder kein erkennbares Muster gibt, oder die Architektur oder Trainingsumgebung des implementierten Netzes nicht optimal ist, sind die Ergebnisse hier mittelmäßig, wenn auch die Laufzeit vergleichsweise kurz ist. Da das Training aber sehr lange dauert und die Ergebnisse nicht vorhersehbar sind, funktioniert der nächste Ansatz nach einem Muster, das zu konsistenten Erfolgen führt. Dies funktioniert, indem in jedem Schritt der größte, nicht sortierte Pfannkuchen an die oberste Stelle bewegt wird. Dann wird der gleiche Pfannkuchen an die richtige Stelle bewegt, also bei einem Stapel $2, 1, 4, 3$ wäre die nächste WUEO unter dem Pfannkuchen mit der Größe 3, um den größten Pfannkuchen nach oben zu bewegen. So ergibt sich $4, 1, 2$. Im nächsten Schritt ist der größte, nicht sortierte Pfannkuchen an oberster Stelle, also wird nach unten bewegt: $1, 4$. Damit ist der Stapel sortiert, allerdings wäre der schnellste Weg ihn zu sortieren, gewesen, direkt unter dem Pfannkuchen mit der Größe 4 eine WUEO durchzuführen, so dass sich nach nur einer WUEO dieser sortierte Stapel ergibt: $1, 2, 3$. Da so im schlimmsten Fall immer zwei WUEOs nötig sind, um einen Pfannkuchen an die richtige Stelle zu bewegen, braucht dieser Algorithmus normalerweise $\frac{2n}{3}$ WUEOs um einen ganzen Stapel zu sortieren. Wenn man also die Listenoperationen bei den WUEOs vernachlässigt, ergibt sich eine lineare Laufzeitordung $\mathcal{O}(n)$.

Insgesamt lohnt es sich am meisten, bis etwa $n = 9$ den Brute-Force Algorithmus und bei größeren Stapeln die vortrainierten neuronale Netze zu benutzen. Sollte das gefundene Ergebnis schlechter als $\frac{2n}{3}$ WUEOs sein, oder kein vortrainiertes Netz für den Stapel existieren, sollte der zuletzt beschriebene Algorithmus verwendet werden.

2 Umsetzung

Brute-Force

Bei der Brute-Force Implementierung zur Sortierung der Pfannkuchenstapel wird rekursiv jede mögliche Kombination von WUEOs ausprobiert. Diejenige Kombination die mit den wenigsten WUEOs zu einem sortierten Stapel führt, wird zurückgegeben.

```
1: function FUNC(stack, pos)                    ▷ siehe nnet/brute_force.py
2:     if stack is sorted then return stack
3:     end if
4:     flip stack at pos
5:     for every possible flipping position do
6:         results ← func(stack, pos)
7:     end for
8:     return the best value in results
9: end function
```

Für die Ergebnisse für Aufgabenteil B muss diese Funktion auf jede Permutation des gegebenen Pfannkuchenstapels aufgerufen werden.

PWUE-Zahlen

Aufgrund möglicher Rundungsfehler, die während des Aufrundens von $\frac{n}{2}$ auftreten können, wird stattdessen $\frac{n+1}{2}$ als Ganzzahldivision durchgeführt. Dies hat genau den gleichen Effekt wie das Aufrunden.

Neuronales Netz

Da es die Implementation eines neuronalen Netzes stark vereinfacht, wenn das Netz nicht mit verschieden großen Inputs arbeiten muss, wird der Pfannkuchenstapel mit Größe n als Tensor von Ganzzahlen in ein

neuronales Netz mit der In- und Output-Größe n gespeist. Derjenige Output-Punkt, der am Ende den höchsten Wert besitzt, stellt dann die Stelle dar, an der die WUEO durchgeführt werden soll. Nach der Operation wird der veränderte Stapel in ein Netz mit der Größe $n-1$ gegeben. Das geht weiter, bis das der Stapel eine Größe von 1 erreicht hat, dann ist er auf jeden Fall sortiert. Die Folge ist allerdings, dass um einen Pfannkuchestapel mit der Größe n zu lösen, $n-1$ verschiedene neuronale Netze trainiert sein müssen. Um das Lernen zu erleichtern, wird der Stapel normalisiert in das nächstkleinere Netz übergeben. Wenn also nach einer Menge an WUEOs der Stapel zum Beispiel so aussieht: $4, 3, 1, 8, 6$, wird er stattdessen als dieser Stapel behandelt: $3, 2, 1, 5, 4$. Dies macht keinen Unterschied für den besten Lösungsweg, da die angestrebte Reihenfolge nur von der Größe der einzelnen Pfannkuchen im Vergleich zu den anderen Pfannkuchen definiert ist. Also kann ein Stapel $3, 6, 8$ genau wie ein Stapel $1, 2, 3$ behandelt werden. Der Vorteil an dem Verfahren ist, dass das Training des neuronalen Netzes nur mit den verschiedenen Permutationen von $1, 2, .., n$ durchgeführt werden muss.
Um das Netz zu trainieren, wurde ein Reinforcement Learning Algorithmus verwendet. Dieser Algorithmus bezieht Informationen über die Richtigkeit der Ergebnisse des Netzes nicht aus vorgegebenen Datensätzen, sondern lässt sich von einer Handlungsumgebung eine Belohnung geben. Die Handlungsumgebung rechnet die Ergebnisse des neuronalen Netzes in eine Aktion um, die direkt umgesetzt wird. In diesem Fall würde das Netz zum Beispiel einen Tensor mit den Werten $0.5, 0.3, 0.6, 0.9$ ausgeben. Dabei entspricht der erste Wert im Tensor der Aktion 1, der zweite Wert der Aktion 2 und so weiter. Hier würde die Umgebung also Aktion 4 ausführen und in unserem Fall unter Pfannkuchen 4 eine WUEO durchführen. Dann würde der veränderte Stapel, wie oben beschrieben, von jedem kleineren Netz behandelt werden, bis er sortiert ist. Je nach dem, wie viele Pfannkuchen am Ende übrig bleiben, wird ein großer oder kleiner Wert als Belohnung an den Optimizer zurückgegeben. Um nun den Verlust des Netzes zu berechnen wird, vereinfacht dargestellt, die Belohnung mit dem höchsten Wert im Ergebnistensor addiert. Dann wird der Unterschied zwischen dem ursprünglichen Tensor und dem mit der Belohnung berechnet. Angenommen die Belohnung beträgt 2 also zwischen $0.5, 0.3, 0.6, 0.9$ und $0.5, 0.3, 0.6, 2.9$. Dann werden die Gewichte der Verbindungen im neuronalen Netz so verändert, dass bei gleichem Input möglichst diese Werte erreicht wird. Also wird das Netz in guten Entscheidungen bestärkt.

$\frac{2n}{3}$-Algorithmus

Die Implementierung des Algorithmus folgt genau der Beschreibung der Lösungsidee.

```
1: function SORT(stack)                          ▷ siehe src/main.rs Zeile 62
2:     while stack is not sorted do
3:         if topmost pancake equals biggest unsorted pancake then
4:             flip the unsorted part of the stack
5:         else
6:             flip below the biggest unsorted pancake
7:         end if
8:     end while
9: end function
```

3 Beispiele

Die gezeigten Lösungen wurden alle vom $\frac{2n}{3}$-Algorithmus berechnet, da dieser die zuverlässigsten Ergebnisse liefert. Hierbei sei angemerkt, dass bei der Angabe der WUEOs die Positionen Zwischenräume zwischen den Pfannkuchen angeben. Position 0 ist über dem obersten Pfannkuchen und Position n unter dem untersten Pfannkuchen. Bei der Darstellung der Stapel ist das erste Element der oberste Pfannkuchen.

Die Lösungen der gegebenen Beispiele:

Beispiel 1:
Stapel:[3, 2, 4, 5, 1]
PWUE: 3, Höhe 5
WUEOs: 2 bei [5, 4]
Ergebnisstapel: [2, 4, 5]

Beispiel 2:
Stapel:[6, 3, 1, 7, 4, 2, 5]
PWUE: 4, Höhe 7
WUEOs: 4 bei [5, 6, 3, 3]
Ergebnisstapel: [2, 6, 7]

Beispiel 3:
Stapel:[8, 1, 7, 5, 3, 6, 4, 2]
PWUE: 4, Höhe 8
WUEOs: 4 bei [8, 6, 5, 3]
Ergebnisstapel: [3, 6, 7, 8]

Beispiel 4:
Stapel:[5, 10, 1, 11, 4, 8, 2, 9, 7, 3, 6]
PWUE: 6, Höhe 11
WUEOs: 7 bei [5, 10, 8, 7, 3, 4, 2]
Ergebnisstapel: [8, 9, 10, 11]

Beispiel 5:
Stapel:[7, 4, 11, 5, 10, 6, 1, 13, 12, 9, 3, 8, 2]
PWUE: 7, Höhe 13
WUEOs: 8 bei [9, 12, 7, 9, 3, 6, 3, 3]
Ergebnisstapel: [7, 9, 10, 11, 13]

Beispiel 6:
Stapel:[4, 13, 10, 8, 2, 3, 7, 9, 14, 1, 12, 6, 5, 11]
PWUE: 7, Höhe 14
WUEOs: 9 bei [10, 13, 6, 10, 8, 7, 4, 4, 2]
Ergebnisstapel: [3, 8, 12, 13, 14]

Beispiel 7:
Stapel:[14, 8, 4, 12, 13, 2, 1, 15, 7, 11, 3, 9, 5, 10, 6]
PWUE: 8, Höhe 15
WUEOs: 10 bei [9, 14, 7, 11, 3, 8, 5, 5, 3, 2]
Ergebnisstapel: [9, 10, 13, 14, 15]

Beispiel 8:
Stapel:[8, 5, 10, 15, 3, 7, 13, 6, 2, 4, 12, 9, 1, 14, 16, 11]
PWUE: 8, Höhe 16
WUEOs: 10 bei [16, 15, 4, 12, 8, 9, 5, 6, 3, 2]
Ergebnisstapel: [5, 9, 12, 13, 15, 16]

Beispiel 9:
Stapel:[11, 10, 2, 8, 4, 6, 5, 7, 3, 9, 1]
PWUE: 6, Höhe 11
WUEOs: 5 bei [11, 8, 6, 4, 2]
Ergebnisstapel: [6, 7, 8, 9, 10, 11]

Beispiel 10:
Stapel:[1, 8, 2, 7, 3, 6, 4, 5]
PWUE: 4, Höhe 8
WUEOs: 5 bei [3, 7, 5, 4, 2]
Ergebnisstapel: [6, 7, 8]

Ein sehr einfaches Beispiel um eine Schwäche des Algorithmus zu zeigen:

Stapel: [1, 3, 2]
PWUE: 2, Höhe 3
WUEOs: 2 bei [3, 2]
Ergebnisstapel: [3]

Bei dem gleichen Beispiel wird auch offensichtlich, dass das neuronale Netz scheinbar ein ähnliches Verfahren verwendet:

Stapel: [1, 3, 2]
WUEOS bei: [3, 2]
Ergebnisstapel: [3]

Der Brute-Force Algorithmus findet natürlich die beste Lösung:

Stapel: [1, 3, 2]
WUEOs: 1 bei [2]
Ergebnisstapel: [1, 2]

Aber etwa Beispiel 2 kann das neuronale Netz besser lösen als der $\frac{2n}{3}$-Algorithmus, da es eine WUEO weniger braucht:

Beispiel 2 (NN):
Stapel: [6, 3, 1, 7, 4, 2, 5]
WUEOs bei: [3, 4, 5]
Ergebnisstapel: [2, 3, 6, 7]

Während es bei Beispiel 9 eine WUEO mehr benötigt:

Beispiel 9 (NN):
Stapel: [11, 10, 2, 8, 4, 6, 5, 7, 3, 9, 1]
WUEOs bei: [7, 7, 9, 6, 4, 3]
Ergebnisstapel: [4, 6, 9, 10, 11]

PWUE-Zahlen (extrapoliert):

n	1	2	3	4	5	6	7	8	9	10	11	12	13
$P(n)$	0	1	2	2	3	3	4	4	5	5	6	6	7

4 Quellcode

Implementierung eines Pfannkuchenstapels in Rust. Einige selbsterklärende Funktionen sind nicht abgebildet.
src/main.rs

```
struct Stack {
    height: usize,
    pancakes: Vec<usize>,
    flips: (usize, Vec<usize>)
}

impl Stack {
    pub fn sort(&mut self) {
        while !self.is_sorted() {
            if self.pancakes.get(0).unwrap() == &self.biggest_unsorted().0 {
                self.flip(self.len() - self.amount_sorted());
            } else {
                self.flip(self.biggest_unsorted().1 + 2);
            }
```

```
        }
    }

    // flips UNDER the given pos and "eats" the top pancake after
    pub fn flip(&mut self, pos: usize) {
        self.flips.0 += 1;
        self.flips.1.push(pos);
        let stack = self.pancakes.clone().into_iter();
        let mut flipped = stack.clone().take(pos - 1).rev().collect::<Vec<usize>>();
        let mut rest = stack.skip(pos).collect::<Vec<usize>>();
        flipped.append(&mut rest);
        self.pancakes = flipped;
    }
        pub fn worst_stack(size: usize) -> Stack {
        let mut items = Vec::new();
        let mid = size / 2;
        for i in 1.. {
            if i != 0 {items.push(i)}
            if items.len() == size {break;}
            if size - i + 1 > mid {items.push(size - i + 1)}
            if items.len() == size {break;}
        }
        items = items.into_iter().rev().collect();
        Stack::new(In::I(items))
    }

    pub fn pwue(size: usize) -> usize {
        return ((size + 1) as f32 / 2.) as usize
    }
}
```

Implementierung des selben Konzepts in Python, allerdings mit Interface für Sortierung per neuronalem Netz mit PyTorch.
nnet/Stack.py

```
class Stack:
    def __init__(self, items: list) -> None:
        self.height = len(items)
        self.pancakes = items

    def sort_by_ai(self) -> list:
        nets = []
        for i in range(len(self), 1, -1):
            try:
                with open(f"./nnet/trained_nets/net{i}.save", "rb") as file:
                    nets.append(torch.load(file))
            except FileNotFoundError:
                raise FileNotFoundError("Not every necessary pretrained NN was found")

        flips = []
        for (i, net) in enumerate(nets):
            if self.is_sorted(): break
            flips.append(net(self.as_tensor()).argmax().item() + 1)
            self.flip(flips[-1])
        return flips
```

Implementierung des Brute-Force Algorithmus in Python. Die Brute-Force Funktion ist keine Methode des Stacks, da der Stapel selber im Prozess nicht verändert wird.
nnet/brute_force.py

```
def brute_force(stack: Stack, pos: int=0, init: bool=True) -> Tuple[int, list]:
    if stack.is_sorted():
        return (stack.height - len(stack), [])

    if not init:
        stack.flip(pos)

    results = []
    paths = []
    length = stack.height - len(stack)
```

```
    for i in range(1, len(stack) + 1):
        s = copy.deepcopy(stack)
        res = brute_force(s, pos=i, init=False)
        results.append(res[0])
        paths.append(res[1])

        if res[0] == length:
            b = res[1]
            b.append(pos)
            return (results[-1], b)

    path = paths[results.index(min(results))]

    if init: path.reverse()
    else: path.append(pos)

    return (min(results), path)
```

Die wichtigsten Parameter für das Training der neuronalen Netze.
nnet/neural_net.py

```
BATCH_SIZE = 128
# Effectively defines how far in the future the net "thinks"
GAMMA = 0.99
# Mutation probability decreases by EPS_DECAY down to EPS_END
EPS_START = 0.9
EPS_END = 0.
EPS_DECAY = 1000
# Defines how strong the policy nets parameters affect the target net
TAU = 0.005
# The optimizers learning rate
LR = 1e-2

# Size of pancakestack = amount of possible flips
STACK_HEIGHT = 16

if torch.cuda.is_available():
    num_epochs = 10000
else:
    num_epochs = 1000
```

Das Environment ist einer der wichtigsten Teile des Trainingsprozesses, da dieses die Belohnungen festlegt, die das Netz für bestimmte Aktionen bekommt.
nnet/neural_net.py

```
class Environment:
    """The environment is responsible for providing data and computing the nets decisions effects."""
    def __init__(self, size: int, data=None) -> None:
        self.size = size
        self.stack = data

    def step(self, action: int) -> typing.Tuple[torch.Tensor | None, float, bool, bool]:
        """Computes the state a given action has caused and the rewards.

        Args:
            action (`int`): The position to flip the stack at

        Returns:
            (`torch.Tensor | None`, `float`, `bool`, `bool`): (resulting state, reward, terminated, tru
        """
        try:
            self.stack.flip(action)
            if self.stack.is_sorted():
                return (self.stack.as_tensor(fill=True, normalized=False).to(device),
                        len(self.stack) * 2.,
                        False,
                        True)
            else:
                for n in lower_nets:
```

```
                    res = n(self.stack.as_tensor())
                    self.stack.flip(res.argmax() + 1)
                    if self.stack.is_sorted():
                        return (self.stack.as_tensor(fill=True,
                                                     normalized=False).to(device),
                                                    len(self.stack) * 2.,
                                                    False,
                                                    True)
                return (self.stack.as_tensor(fill=True, normalized=False).to(device),
                        len(self.stack) * 2.,
                        False,
                        True)
        except FlipNotPossible:
            return (None, -10., True, False)
```

Die gewählte Aktion für das Training wird manchmal zufällig bestimmt, um ein möglichst breites Gedächtnis an Aktionen zu haben.
nnet/neural_net.py

```
def select_action(state: torch.Tensor) -> torch.Tensor:
    # Formula to decrease the decrease of mutation probability exponentially by step and EPS_DECAY
    eps_threshold = EPS_END + (EPS_START - EPS_END) * math.exp(-1. * steps_done / EPS_DECAY)
    steps_done += 1
    if sample > eps_threshold:
        with torch.no_grad():
            return policy_net(state).max(1)[1].view(1, 1)
    else:
        return torch.tensor([[env.get_rand_action()]], device=device, dtype=torch.long)
```

Die *optimize_model*() Funktion ist das Herz des Prozesses. Hier wird die Berechnung des Losses und die Anpassung der Gewichte vorgenommen.
nnet/neural_net.py

```
def optimize_model():
    # The model will only optimize after enough data is accumulated
    if len(memory) < BATCH_SIZE: return

    # Gets a random sample and turns the tensor of Transitions to
    # a single Transition with tensors of all values
    transitions = memory.sample(BATCH_SIZE)
    batch = Transition(*zip(*transitions))

    # s is None when the epoch was terminated, not truncated (see training loop)
    non_final_mask = torch.tensor(tuple(map(lambda s: s is not None, batch.next_state)),
                                  device=device,
                                  dtype=torch.bool)
    non_final_next_states = torch.cat([s for s in batch.next_state if s is not None])

    state_batch = torch.cat(batch.state)
    action_batch = torch.cat(batch.action)
    reward_batch = torch.cat(batch.reward)

    # This creates values that represent how likely the policy net would have
    # chosen the action that it remembered
    state_action_values = policy_net(state_batch).gather(1, action_batch)

    # This is for computing how certain the target net would have been in its decision
    # Later this will be used to compute the difference between
    # the policy and target nets decisions, which results in the loss
    next_state_values = torch.zeros(BATCH_SIZE, device=device)
    with torch.no_grad():
        next_state_values[non_final_mask] = target_net(non_final_next_states).max(1)[0]

    # next_state_values represents the confidence of the target net,
    # adding the reward results in what the confidence values should have been
    # With a low GAMMA value > 0 the instant reward values get more important than
    # the certainty to get more rewards in the future
    expected_state_action_values = (next_state_values * GAMMA) + reward_batch

    # Computes loss between chosen actions and highest-reward-actions
```

```
    criterion = nn.SmoothL1Loss()
    loss = criterion(state_action_values, expected_state_action_values.unsqueeze(1))

    # Backprogation to update the weights
    optimizer.zero_grad()
    loss.backward()
    optimizer.step()
```

Die Trainingsschleife kümmert sich um die Anhäufung von Daten und die Verbindung von der Handlungsumgebung und der *optimize_model*() Funktion.
nnet/neural_net.py

```
for i_episode in range(num_epochs):
    # Initialize the environment and get its state
    state = env.init_rand_stack()
    state = state.clone().detach().unsqueeze(0)
    for t in count():
        action = select_action(state)
        observation, reward, terminated, truncated = env.step(action.item() + 1)
        reward = torch.tensor([reward], device=device)
        done = terminated or truncated

        if terminated:
            next_state = None
        else:
            next_state = observation.clone().detach().unsqueeze(0)

        # Save the transition to learn from later
        memory.push(state, action, next_state, reward)

        state = next_state

        # Update policy_net
        optimize_model()

        # The target nets weights are only partially affected by the policy net to even out
        # drastic updates on the policy net
        tnsd = target_net.state_dict()
        pnsd = policy_net.state_dict()
        for key in pnsd:
            tnsd[key] = pnsd[key]*TAU +tnsd[key]*(1-TAU)
        target_net.load_state_dict(tnsd)

        if truncated:
            actually_sorted.append(STACK_HEIGHT)
            episode_durations.append(t + 1)
        elif terminated:
            episode_durations.append(STACK_HEIGHT)
            actually_sorted.append(0)
        if done:
            rewards.append(reward / 2)
            if REALTIME_PLOTS:
                plot_information()
            break
```

Abschließend ist festzuhalten, dass die Reihenfolge der Pfannkuchen keine Auswirkungen auf den Genuss haben.