Reinforcement Learning

Grundlagen Grundbegriffe Grundübungen

Alfatraining Kurs 2025-12 Dozent: Karsten Flügge

Themen des Tages

Tag 7

Policy Gradients
REINFORCE

Deep Reinforcement Learning

Es lassen sich alle Hilfskonzepte und Hilfsfunktionen welche wir bis jetzt kennen gelernt haben mit **neuronalen Netz trainieren**:

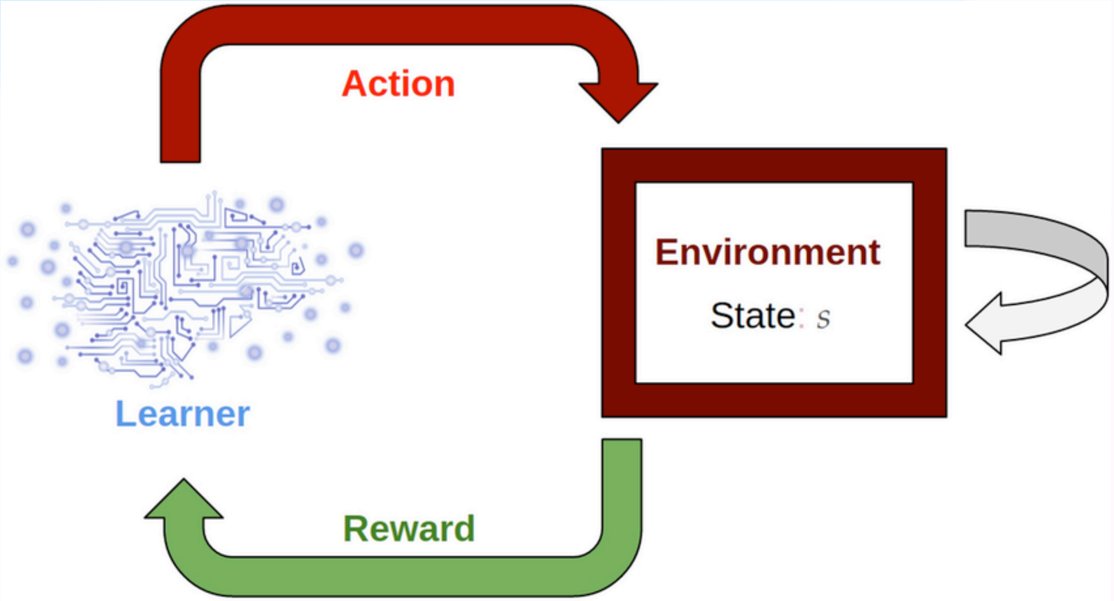
- **Value** Funktionen,
- ***Quality*** Funktionen oder
- ***Policy*** direkt <<<

Policy Gradients

Policy Gradients

Statt die **policy** über *qualities* zu suchen, trainieren wir sie nun **direkt.**

Im einfachsten Falle wird unser Agent komplett über ein einziges Netz antrainiert und gesteuert!

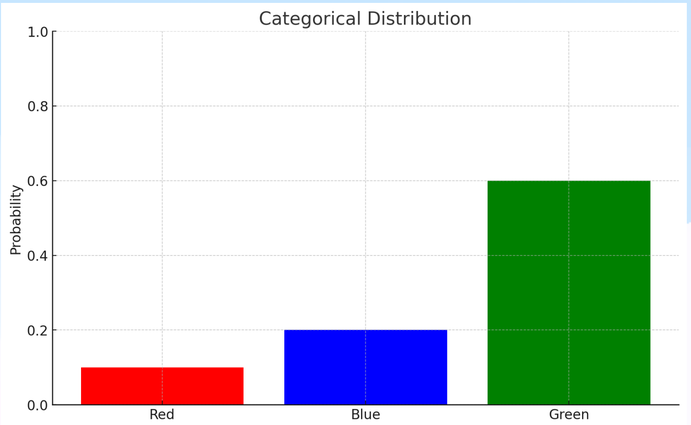


Policy Gradients

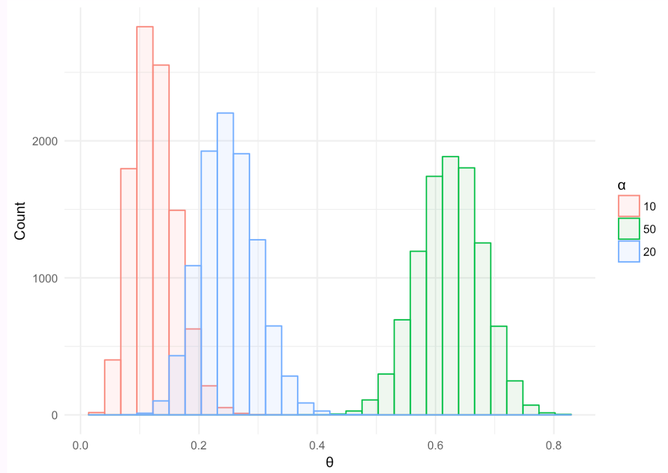
Policy Gradients

Statt die **policy** über qualities zu suchen, trainieren wir sie nun **direkt.**

π ist die **Ausgabe** unseres Netzes,
interpretiert als
Wahrscheinlichkeiten für die **Aktionen,**
z.B. [0.1, 0.2, 0.6] **∑=1** **P(a|s)**



Ein Wert der Wahrscheinlichkeiten wird per '**sampling**' geholt:



Policy Gradients

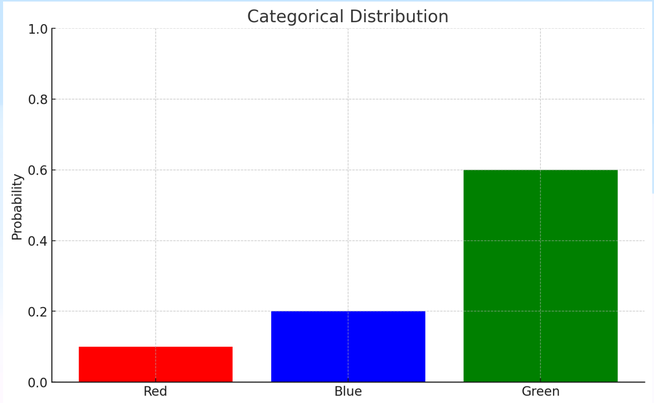
klassischer Policy Gradient

Um zu betonen dass unsere **Policy** von **Gewichten θ** abhängt
schreibt man oft **π(**a|s;θ) oder **π**(a|θ) bei bekanntem Zustand.
θ ≈ neuronales Netz (Gewichte)

Policy Gradients

Policy Gradients

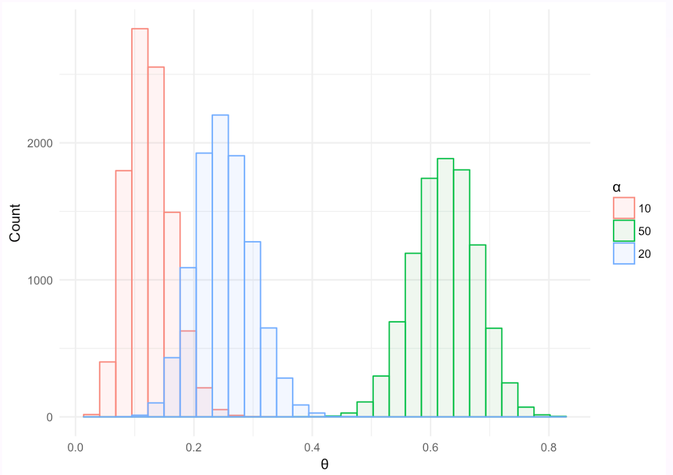
π ist die **Ausgabe** unseres Netzes,
interpretiert als
Wahrscheinlichkeiten für die **Aktionen**,
z.B. [0.1, 0.2, 0.6] ∑=1



💡 wir holen uns **konkrete Aktion** via **softmax** mit categorical/**multinomial sampling**

gemäß der Wahrscheinlichkeit,
z.B. 60%ig: grün!

https://pannous.com/files/softmax.html



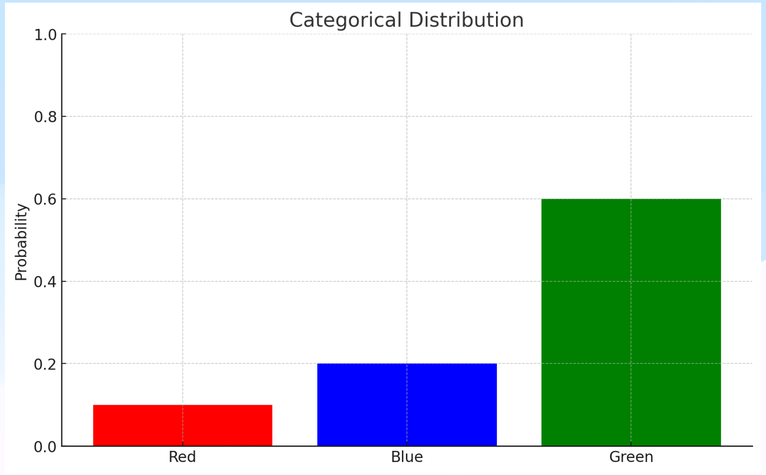
categorical/**multinomial sampling**

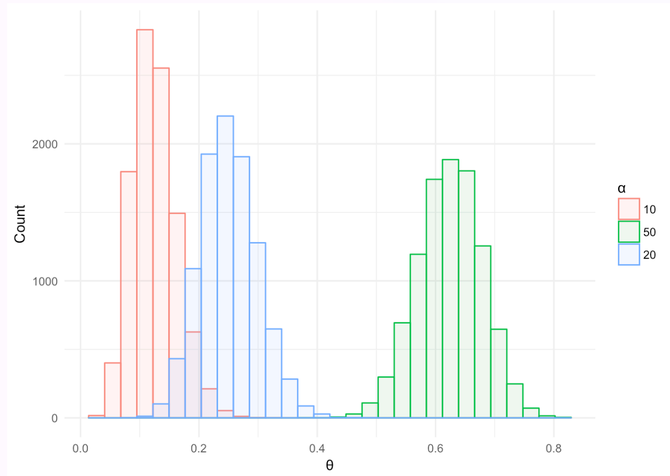
π ist die **Ausgabe** unseres Netzes,
interpretiert als
Wahrscheinlichkeiten für die **Aktionen,**
z.B. [0.1, 0.2, 0.6] ∑=1

💡 wir holen uns **konkrete Aktion zufällig**
via **softmax** mit categorical/**multinomial sampling**
gemäß der Wahrscheinlichkeiten

per Hand:
links 30% rechts 70%

```
if random() < 0.3:
 action = 0 // left
else:
 action = 1 // right
```



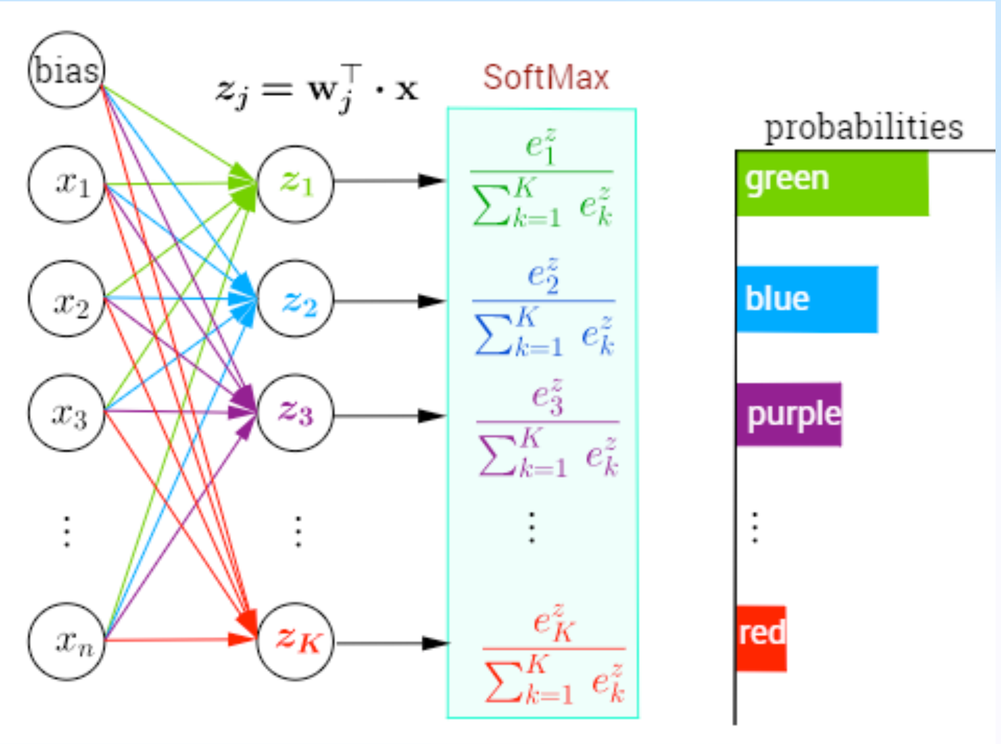


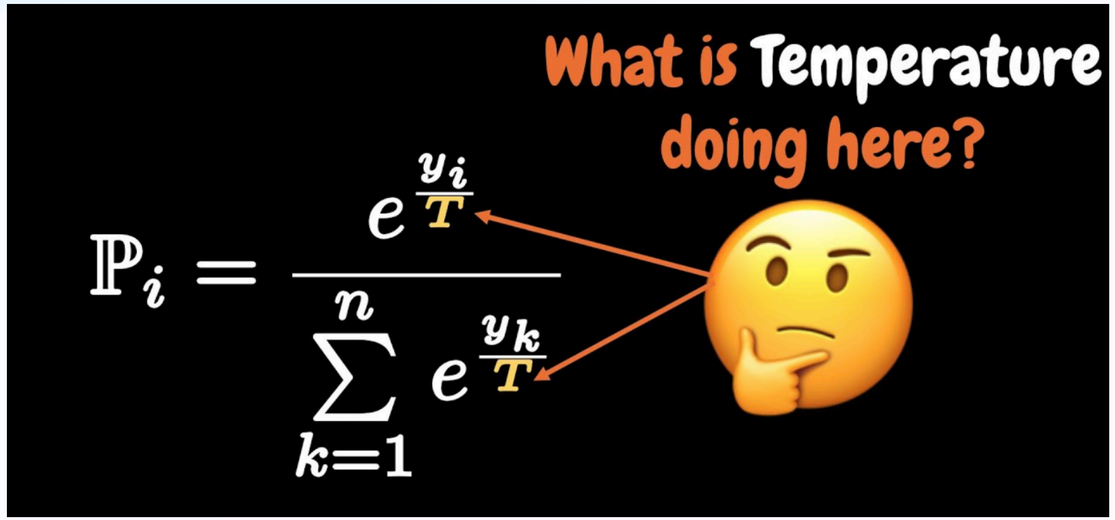

```
# sample right 0/ left 1   action 30% / 70%
action = torch.multinomial(probs, num_samples=1).item()
```

Softmax

Wann immer wir die letzte Schicht eines Netzwerkes in eine **Wahrscheinlichkeitsverteilung** umwandeln wollen, nutzen wir **softmax**.

`Zum Beispiel für Vorhersage von` **`Klassen`** `(Katze, Hund)` **`Kategorien`** `(Zahl 1,2,3) oder neu: für` **`diskrete Aktionen`** `(rechts, links)!`





Softmax

Softmax Ziel: alle Werte zwischen 0 und 1 quetschen, damit wir es als probabilty deuten können.
Summe aller Aktionen / Aktivierungen soll 1 sein.

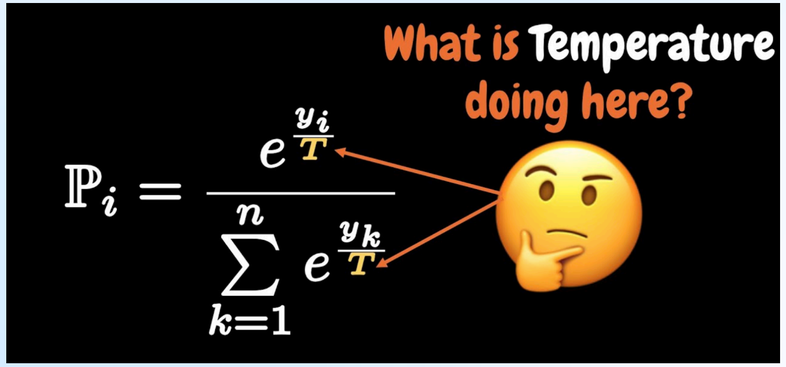
was würde passieren wenn alle Werte null sind?

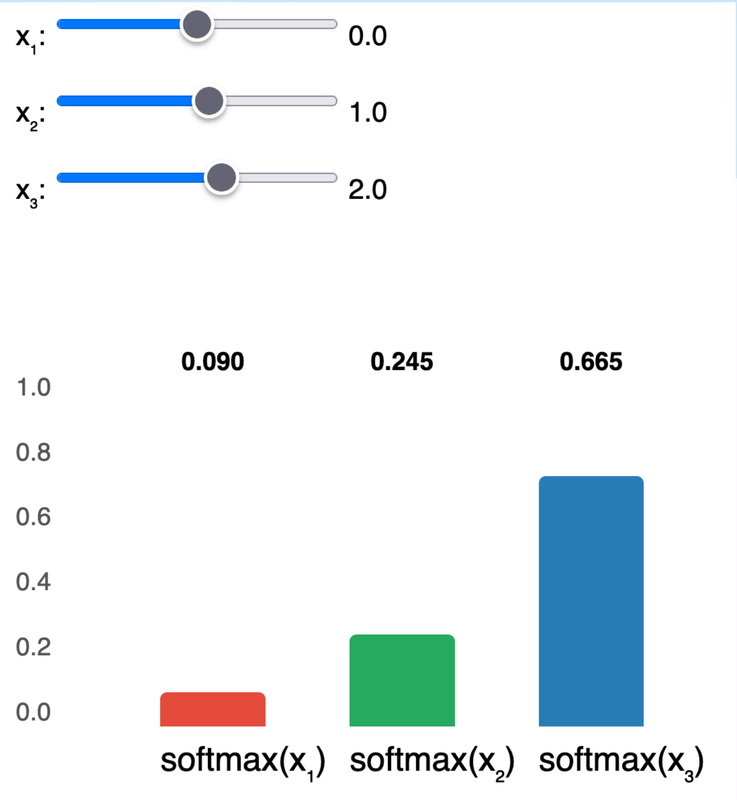
```
probabilties = |q_values| / np.sum(|q_values|)
```

***Deswegen* wird bei Softmax noch e Exponent angewendet!**

z.B. bei **x0=0** (x1=**1** x2=**2**) dann sollte softmax(z0) immer noch nahe 0 sein

softmax(x0) = exp(0)/∑… = 1 / (1 + e^1 + e^2) ≈ 0.01
softmax(x1) = exp(1)/∑… = e / (1 + e^1 + e^2) ≈ 0.25
softmax(x2) = exp(2)/∑… = e^2 / (1 + e^1 + e^2) ≈ 0.66





Exploration via Softmax

Zufällige Auswahl der nächsten Aktion gemäß skalierter **Qualität**

```
def q_softmax_sample(q_values):
  probs = np.exp(q_values/T) / np.sum(np.exp(q_values/T))
  return np.random.choice(len(probs), p=probs)
```

Exploration

Temperatur

```
T ≈ 0.0 # Temperatur 0: deterministisch max
T = 1.0 # Temperatur 1: moderate Exploration
T = 100 # Temperatur ∞: uniform random wie epsilon
```

T / **τ (tau)** ist die Temperatur

- **Hohe T → hohe Entropie** (Exploration)
- **Niedrige T → niedrige Entropie** (Exploitation)

In Neuronalen Netzen haben wir gesehen dass
softmax … gibts auch bei LLMs

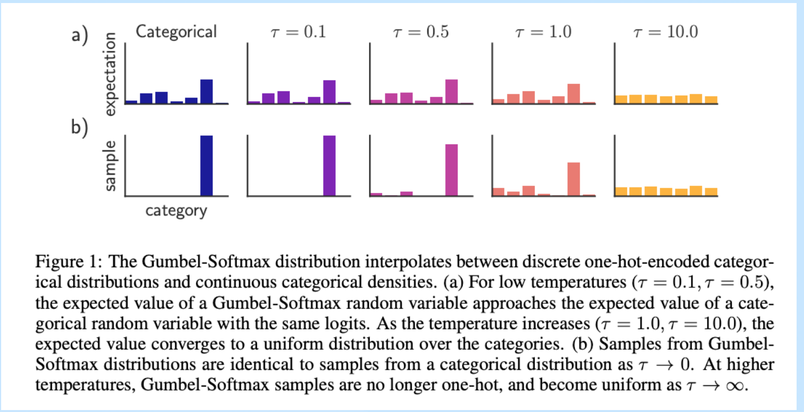
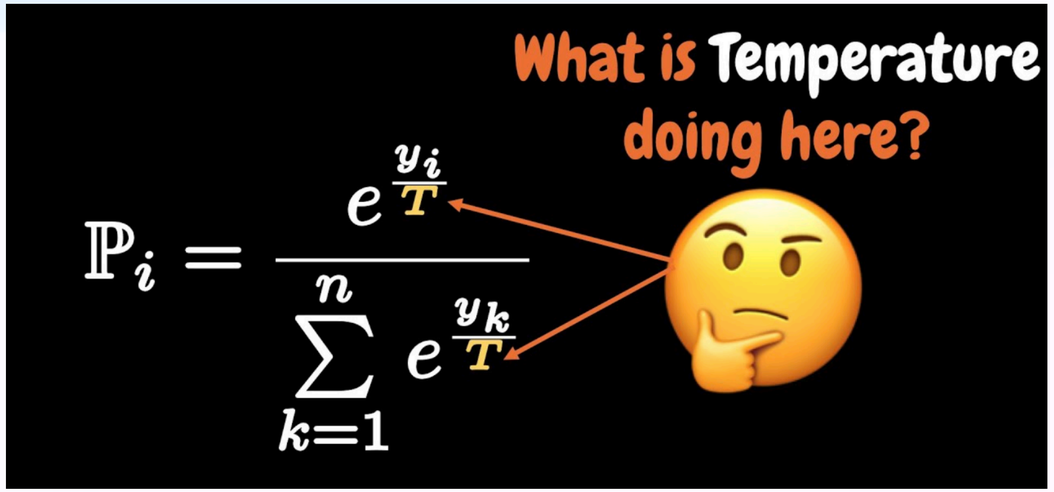


Figure 1: The Gumbel-Softmax distribution interpolates between discrete one-hot-encoded categorical distributions and continuous categorical densities. (a) For low temperatures ($\tau = 0.1, \tau = 0.5$), the expected value of a Gumbel-Softmax random variable approaches the expected value of a categorical random variable with the same logits. As the temperature increases ($\tau = 1.0, \tau = 10.0$), the expected value converges to a uniform distribution over the categories. (b) Samples from Gumbel-Softmax distributions are identical to samples from a categorical distribution as $\tau \to 0$. At higher temperatures, Gumbel-Softmax samples are no longer one-hot, and become uniform as $\tau \to \infty$.



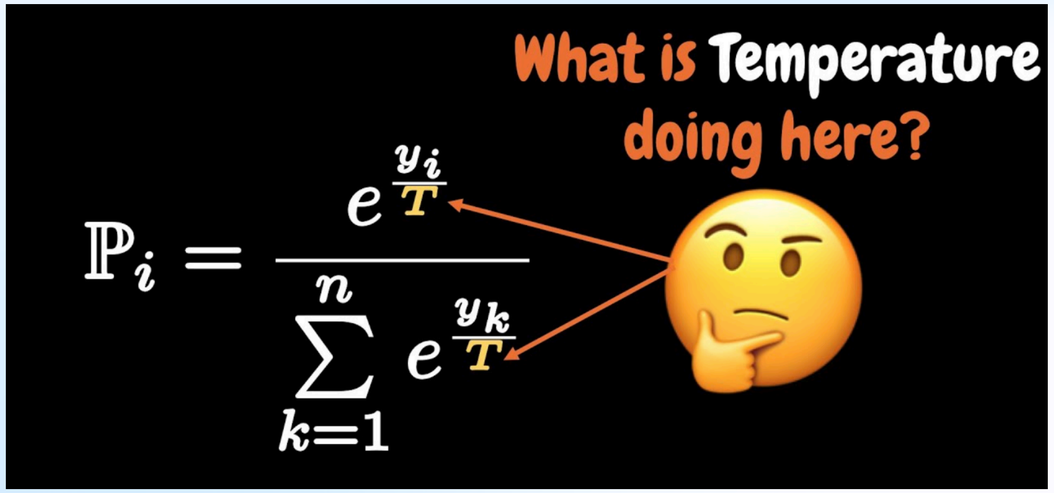
Softmax Exploration

`Vertiefung Exploration vs. Exploitation`
Kapitel 3.2 im Buch

Boltzmann exploration: T statt epsilon (im Grenzfall selbes Resultat)

Temperature

z. B. REINFORCE, Actor-Critic



Policy Gradients

klassischer Policy Gradient PG

PG Grundalgorithmus in einer Klasse

Um zu betonen dass unsere **Policy** von **Gewichten θ** abhängt
schreibt man oft **π(**a|s;θ) oder **π**(a|θ) bei bekanntem Zustand.
θ ≈ neuronales Netz (Gewichte)

Statt dass wir im **Replay** nur einzelne Schritte speichern (Temporal Difference) , speichern wir nur ganze **Episoden** (Monte Carlo)

$$\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2) \ldots (S_{t-1}, A_{t-1}, R_t)$$

Policy Gradients

klassischer Policy Gradient

Um zu betonen dass unsere **Policy** von **Gewichten θ** abhängt schreibt man oft
π(a|s;θ) oder π(a|θ) bei bekanntem Zustand

Statt dass wir im **Replay** nur einzelne Schritte speichern (Temporal Difference) , speichern wir nur ganze **Episoden** (Monte Carlo)

$$\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2) \dots (S_{t-1}, A_{t-1}, R_t)$$

PG Policy Gradient bezeichnet nicht nur den allgemeinen Ansatz, policies über gradient updates zu approximieren, sondern eine **spezifische Klasse** mit mathematischer Herleitung.

https://spinningup.openai.com/en/latest/spinningup/rl_intro3.html

$$J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} [R(\tau)]$$

Erwarteter Gewinn≈**R**eturn der Policy **E[G]**

$$\nabla_\theta J(\pi_\theta) = \underset{\tau \sim \pi_\theta}{\mathrm{E}} \left[\sum_{t=0}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \Phi_t \right]$$

… loss = - ∑ log π · ϕ

Policy Gradients

Policy Gradient ist eine **Klasse von Algorithmen**, bei welchem das policy Netzwerk mit einem **Loss** dieser Form berechnet wird:

loss = $-\sum$ log x * y (y label haben wir nicht mehr)
loss = $-\sum$ **log** π * $\boldsymbol{\varphi_t}$ **φ** *surrogate* phi
loss = $-\sum$ **log** π * **G** **G Gewinn** bei REINFORCE
loss = $-\sum$ **log** π * $\boldsymbol{\delta_t}$ $\boldsymbol{\delta_t}$ **Delta** (TD) bei Actor-Critic
loss = $-\sum$ **log** π * $\mathbf{A_t}$ **A Advantage**

(≈ähnelt **Kreuzentropie**)

$$\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2) \ldots (S_{t-1}, A_{t-1}, R_t)$$

Policy Gradients

Policy Gradient ist eine **Klasse von Algorithmen**, bei welchem das policy Netzwerk mit einem **Loss** dieser Form berechnet wird:

Übernehmen der 'CrossEntropy' Idee bei RL:

Target haben wir nicht, nehmen wir erst mal hohen Gewinn als 'Ziel'.

loss = -∑ log π *** G** **G Gewinn bei** REINFORCE

loss = -∑ log π_i *** G**$_i$ **für alle vorhergesagten Aktionens-Wahrscheinlichkeiten** π_i "log probs"

nicht mehr den eingebauten Softmax verwenden!

$$\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2) \ldots (S_{t-1}, A_{t-1}, R_t)$$

Softmax Klassifizierung

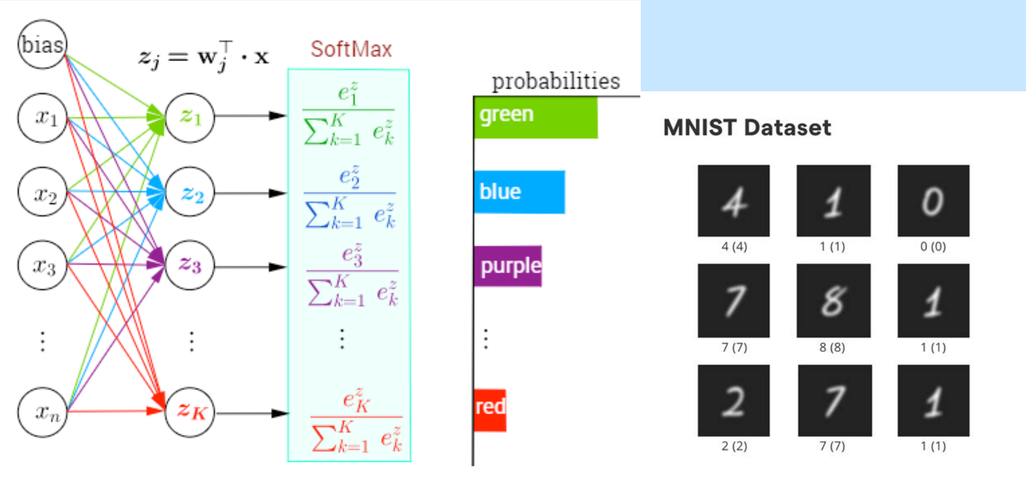
In PyTorch wird **Klassifizierung** in einer einzigen Zeile gehandhabt:

```
criterion = torch.nn.CrossEntropyLoss()
# 💡 Softmax is internally computed behind last layer
```

Der Rest der Trainingsfunktion ist generisch:

```
for epoch in range(training_epochs):
  accuracy()
  for X, Y in data_loader:
    optimizer.zero_grad() # reset gradients to zero
    hypothesis = model(X.view(-1, 28 * 28))
    cost = criterion(hypothesis, Y) # Verlustfunktion
    cost.backward() # backpropagation, ziehe Gradienten ab
    optimizer.step() #
```

One Hot Encoding



10 Zielvariablen haben Zustände 0/1, praktisch dazwischen [0,1] (ja ist Ziffer '7', oder nicht, oder vielleicht)

Ausgabe vom Netz im Extremfall
[**.99**,0,0,0,0,0,0,0,0,0,0,0] heisst Ziffer '0' wird vorhergesagt.
[0,1,0,0,0,0,0,0,0,0,0,0] heisst Ziffer '1' wird vorhergesagt.
[0,.4,0,0,0,0,0,.6,0,0] heisst Ziffer mit 40% ist es eine '1' mit 60% eine '7'
je nach Temperatur wählen wir immer die Beste (greedy) oder multinominial gemäß 40/60 bis zu gleichverteilt

Vergleich mit Wahrheit (echtes Label / Target)
[**1**,0,0,0,0,0,0,0,0,0,0,0]

One Hot encoding

Cross Entropy

Cross Entropy ist das Maß mit dem wir messen wie gut unsere Klassifikations-Vorhersage zum erwarteten (Ziel)Wert (label) passt.

10 Zielvariablen haben Zustände 0/1, praktisch dazwischen [0,1] (ja ist Ziffer '7', oder nicht, oder vielleicht)

Ausgabe vom Netz im Extremfall
[.99,0,0,0,0,0,0,0,0,0,0,0] heisst Ziffer '**0**' wird vorhergesagt.
[0,1,0,0,0,0,0,0,0,0,0,0] heisst Ziffer '1' wird vorhergesagt.
[0,.4,0,0,0,0,0,.6,0,0] heisst Ziffer mit 40% ist es eine '1' mit 60% eine '7'
je nach Temperatur wählen wir immer die Beste (greedy) oder multinominial gemäß 40/60 bis zu gleichverteilt

Vergleich mit Wahrheit (echtes Label / Target)
[1,0,0,0,0,0,0,0,0,0,0,0] ist wirklich Ziffer '0'

One Hot encoding

Cross Entropy

- **wenn Vorhersage und Label übereinstimmen ist alles gut => Fehler ≈ 0**
- **wenn Vorhersage und Label nicht übereinstimmen => Fehler hoch**

Cross Entropy

Cross Entropy ist das Maß mit dem wir messen wie gut unsere Klassifikations-Vorhersage zum erwarteten (Ziel)Wert (label) passt.

10 Zielvariablen haben Zustände 0/1, praktisch dazwischen [0,1] (ja ist Ziffer '7', oder nicht, oder vielleicht)

Ausgabe vom Netz im Extremfall
[.99,0,0,0,0,0,0,0,0,0,0,0] heisst Ziffer '**0**' wird vorhergesagt.
[0,1,0,0,0,0,0,0,0,0,0,0] heisst Ziffer '1' wird vorhergesagt.
[0,.4,0,0,0,0,0,.6,0,0] heisst Ziffer mit 40% ist es eine '1' mit 60% eine '7'
je nach Temperatur wählen wir immer die Beste (greedy) oder multinominial gemäß 40/60 bis zu gleichverteilt

Vergleich mit Wahrheit (echtes Label / Target)
[1,0,0,0,0,0,0,0,0,0,0,0] ist wirklich Ziffer '0'

One Hot encoding

Cross Entropy

- **wenn Vorhersage und Label übereinstimmen ist alles gut => Fehler ≈ 0**
- **wenn Vorhersage und Label nicht übereinstimmen => Fehler hoch**

$$loss = -\sum \log x_i * y_i$$

Cross Entropy IN DL

Cross Entropy ist das Maß mit dem wir messen wie gut unsere Klassifikations-Vorhersage zum erwarteten (Ziel)Wert (label) passt.

10 Zielvariablen haben Zustände 0/1, praktisch dazwischen [0,1] (ja ist Ziffer '7', oder nicht, oder vielleicht)

Ausgabe vom Netz im Extremfall

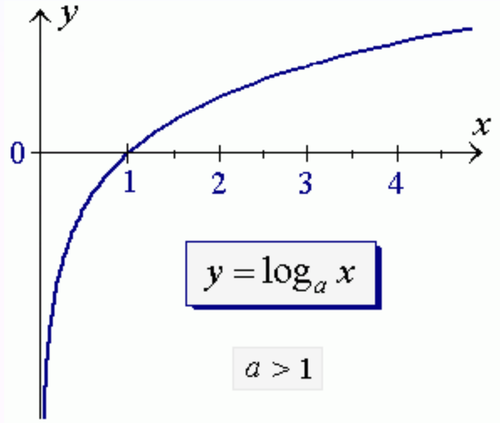
Cross Entropy

- **wenn Vorhersage und Label übereinstimmen ist alles gut => Fehler ≈ 0**
- **wenn Vorhersage und Label nicht übereinstimmen => Fehler hoch**

$$\textbf{loss} = -\sum \log y_i * t_i$$

y=[.99,0,0,0,0,0,0,0,0,0,0,0] heisst Ziffer '0' wird vorhergesagt.
t=[1,0,0,0,0,0,0,0,0,0,0,0] ist wirklich Ziffer '0'

$$\textbf{loss} = -\sum \log y_i * t_i \text{ hat nur eine 1 bei } i=0$$
$$\textbf{loss} = -\log y_0 * 1 = -\log .99 \approx 0$$



y=[.0001,.4,0,0,0,0,0,.6,0,0] heisst Ziffer mit 40% ist es eine '1' mit 60% eine '7'
t=[1,0,0,0,0,0,0,0,0,0,0,0] in Wirklichkeit '0'

$$\textbf{loss} = -\sum \log y_i * t_i$$
$$\textbf{loss} = -\log y_0 * 1 \approx -\log 0.0001 \approx \textbf{sehr groß}$$

Regression vs. Klassifikation

Weitere Aspekte

- **Metriken**:
  - Regression: Mean Squared Error (MSE), Root Mean Squared Error (RMSE), R^2-Score
  - Klassifikation: **Accuracy**, Precision, Recall, F1-Score
- **Modelle**:
  - Regression: Lineare Regression, Lasso, Ridge, Neuronale Netze
  - Klassifikation: Logistische Regression, Entscheidungsbäume, Support Vector Machines, Random Forest, **Neuronale Netze**
- **Datenverteilung**:
  - Regression: Normalerweise kontinuierlich verteilt.
  - Klassifikation: Diskrete Klassen, oft gleichmäßig oder ungleich verteilt.

Loss von Softmax

CrossEntropy DL?

loss = $-\sum$ **log** p_i * q_i q = **target** **CrossEntropy**
loss = $-\sum$ **log** π_i * **G**$_i$ **G** **G**ewinn bei REINFORCE <<
loss = $-\sum$ **log** π_i * **A**$_i$ **A** **A**dvantage Vorteil
loss = $-\sum$ **log** π_i * **φ**$_i$ **φ** ***surrogate*** phi

π_i = Ausgabe von Softmax => log π_i 'log-prob' Vektor

Als Skalarprodukt:
loss = - (**log** π) • **G** Gewinn Vektor

(≈ähnelt **Kreuzentropie**)

$$\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2) \ldots (S_{t-1}, A_{t-1}, R_t)$$

Policy Gradients

REINFORCE Algorithmus

```
def REINFORCE(replay):
  for episode in replay:
    Gᴛ=0
    for t in reverse(range(T)):
      Gt = rt + γ*Gt+1 # ⚠️ reverse rekursiv  Gewinn goal gain
      loss = -∑ log π * Gt   (≈ähnelt Kreuzentropie  mit surrogate phi=G : REINFORCE)

      θ ← θ - α ∇ loss  # Adam
```

REINFORCE ist Monte Carlo auf policy angewendet

$$\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2) \ldots (S_{t-1}, A_{t-1}, R_t)$$

Policy Gradients

1. Sammeln der logarithmierten Aktivierungswahrscheinlichkeiten

```
action = torch.multinomial(probs, num_samples=1).item()
log_prob = torch.log(probs[action]) # Log-Wahrscheinlichkeit der Aktion als 0-D Tensor
log_probs.append(log_prob) # Baue Vektor von Log-Wahrscheinlichkeiten auf 1-D Tensor "Vektor"
```

2. Sammeln der Gewinne

```
returns = []
G = 0  # Gain over whole episode
for r in reversed(rewards): # monte carlo episodes
    G = r + gamma * G  # discounted early rewards
    returns.insert(0, G)
returns = torch.tensor(returns)
```

```
2b. returns = (returns - returns.mean()) / (returns.std() + 1e-8)
# Normalize returns for numerical stability -1 … 1  (OPTIONAL!)
```

3. Berechnung vom Loss:

```
optimizer.zero_grad()
loss = -(logs * returns).sum() # PG REINFORCE loss: -∑ log π
```

$$\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2) \ldots (S_{t-1}, A_{t-1}, R_t)$$

Q-Values Gradients

1. Netzwerk

von der Form her identisch zu Q-Value Netzwerk
nur anders interpretiert / genutzt:

```
q-values = nn.Sequential( # batch wird bei pytorch IMPLIZIT mir durchgeschliffen!
            nn.Linear(observation_size, hidden_size), # keine batch size mitgeben
            nn.ReLU(),
            nn.Linear(hidden_size, n_actions), # Quality-Werte
)
```

$$\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2) \ldots (S_{t-1}, A_{t-1}, R_t)$$

Policy Gradients

1. Netzwerk

von der Form her identisch zu Q-Value Netzwerk
nur anders interpretiert / genutzt:

```
policy = nn.Sequential( # batch wird bei pytorch IMPLIZIT mir durchgeschliffen!
            nn.Linear(observation_size, hidden_size), # keine batch size mitgeben
            nn.ReLU(),
            nn.Linear(hidden_size, n_actions), # Aktionswahrscheinlichkeiten statt Quality-Werte
          nn.Softmax(dim=-1)  # bei diskreten Aktionen!
)
```

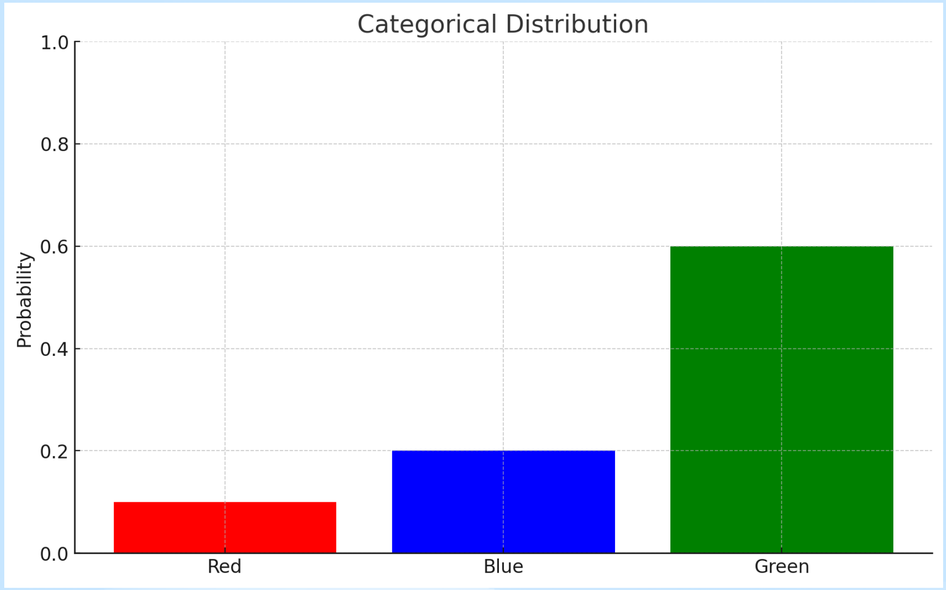
$$\varepsilon = (S_0, A_0, R_1), (S_1, A_1, R_2) \dots (S_{t-1}, A_{t-1}, R_t)$$

Policy Gradients

Im klassischen **REINFORCE** Algorithmus wird die policy in jedem Schritt so upgedated:

```
def REINFORCE(replay):
  for episode in replay:
    for t in range(0,T): # time steps
      G_t = ∑_{i=t} γ^{i-t} r_i   # we discount early rewards
      θ = θ + α*∇ log π(state_t,action_t|θ) * G_t
```

Policy Gradient



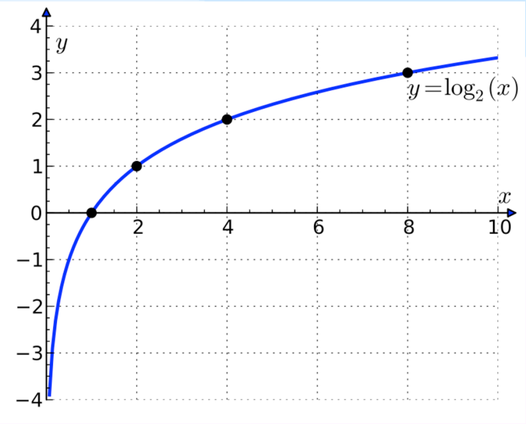
Crossentropy bei **Klassifikation**

x = Ausgabe via softmax, **Vorhersage**
y = Label "was wir vorhersagen **sollen**"
loss = -∑ log x_i * y_i

y ≈ [0,0,1] one hot label

loss = - log(x_i**) da wo label aktiv ist**

Ziel: höchste Aktionswahrscheinlichkeit bei höchstem Gewinn.



bleibt:
wenn Aktivierung x ≈ 1 => loss ≈ 0 wenn g>0 'alles gut' g<0?
wenn Aktivierung x ≈ 0 => loss ≈ ∞ wenn g>0 **nachbessern!!**
wenn Aktivierung x ≈ 0 => loss ≈ -∞ wenn **g<0 ok wir wollen x≈0!**
e^x > 0 => wir kriegen nie log(0)

Policy Gradient

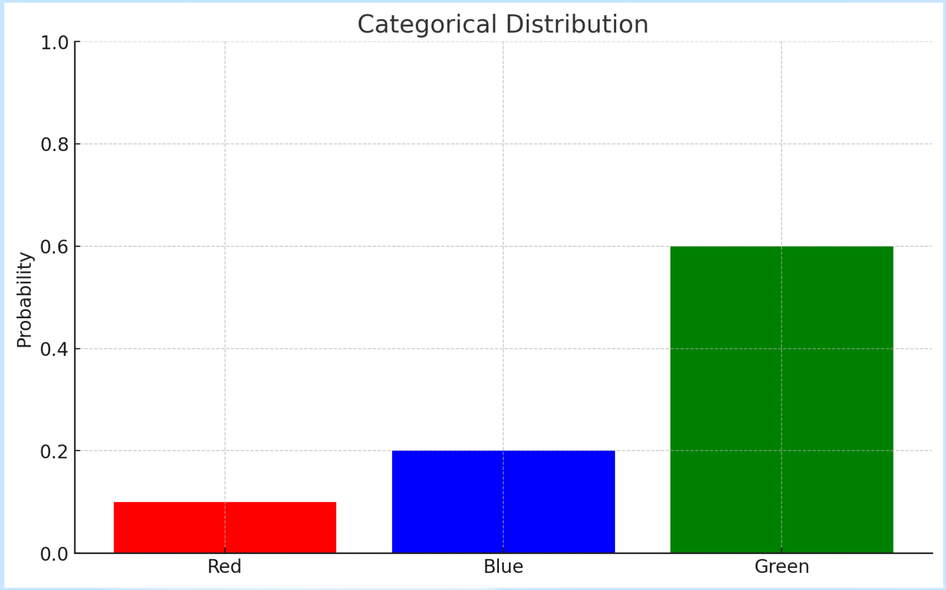
crossentropy **loss** für **policy**

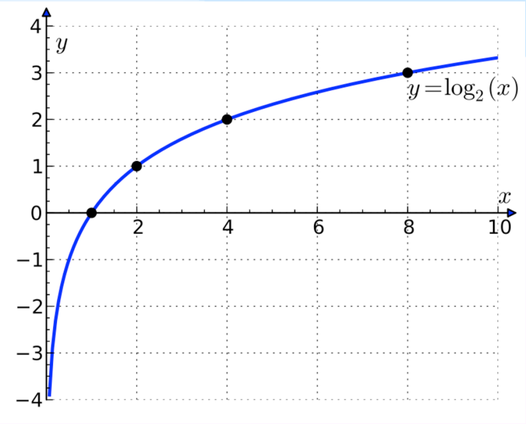
loss = -∑ log x_i * y_i

y ≈ gain! **(keine Trainingslabel, sondern Schätzwert)**
y groß heisst gute Aktion!

z.B. target=[g1,g2,g3]
activation x ≈ [0.1,0.1,0.8]

Ziel: höchste Aktionswahrscheinlichkeit
bei höchstem Gewinn.





bleibt:
wenn Aktivierung x ≈ 1 => loss ≈ 0 wenn g>0 'alles gut' g<0?
wenn Aktivierung x ≈ 0 => loss ≈ ∞ wenn g>0 nachbessern!!
wenn Aktivierung x ≈ 0 => loss ≈ -∞ wenn **g<0 ok wir wollen x≈0!**
e^x > 0 => wir kriegen nie log(0)

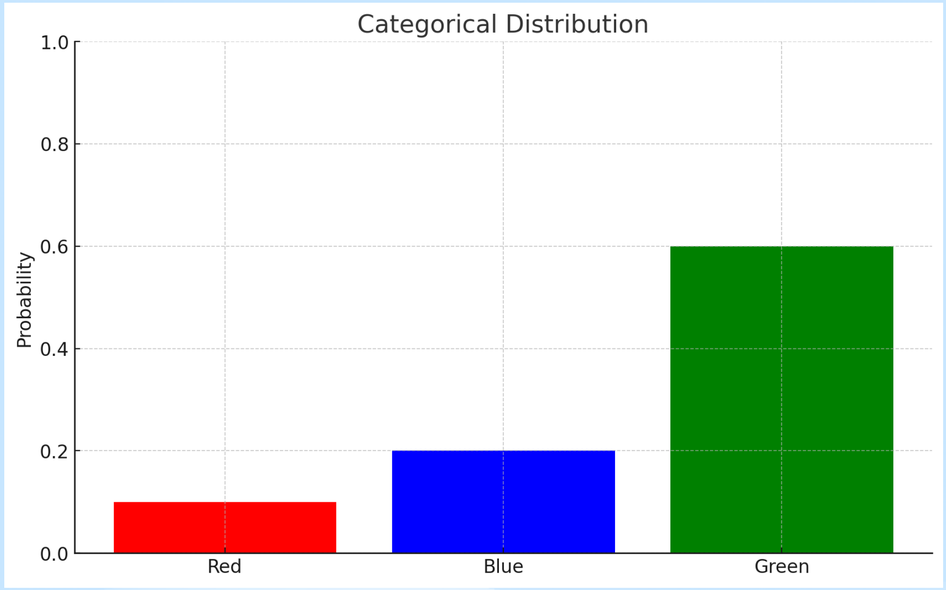
Policy Gradient

Bei policy gradient geht der Gewinn direkt in den Verlust ein. Daher ist es üblich, den Gewinn per reward shaping in einem 'normalen Bereich' zu halten, z.B. [-10..1]

ODER:

- Normalisierung von G_t über Episode oder Batch:

$$\widehat{G_t} = \frac{G_t - \mu}{\sigma}$$

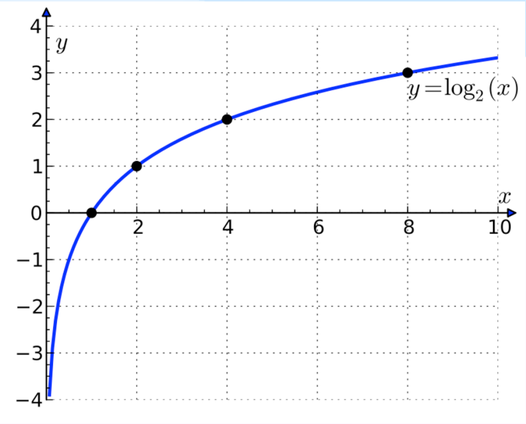


loss = -∑ log x_i * y_i

x in [0..1] y = gain!

Wichtig ist nicht absoluter, sondern nur **relativer Gewinn / Loss**
Loss kann **negativ** sein! Muss nur '**noch niedriger**' werden.

Ziel: höchste Aktionswahrscheinlichkeit bei höchstem Gewinn.



bleibt:
wenn Aktivierung x ≈ 1 => loss ≈ 0 wenn g>0 'alles gut' g<0?
wenn Aktivierung x ≈ 0 => loss ≈ ∞ wenn g>0 nachbessern!!
wenn Aktivierung x ≈ 0 => loss ≈ -∞ wenn **g<0 ok wir wollen x≈0!**

Policy Gradients

```
def PG(replay):
  for episode in replay:
    G_T=0
    for t in reverse(range(T)):
       G_t = r_t + γ*G_t+1  # rekursiv ⚠️ reverse
       loss = -∑ log π(a_t|s_t; θ) * G   (≈ähnelt Kreuzentropie  mit surrogate phi=G : REINFORCE)
      θ ← θ - α ∇ loss  # Adam
```

Warum log?

log([0,1])=[-∞,0] "Wirkungsbereich wird vergrößert"

💡 Verlust wird 0 wenn $\pi(\text{action}|\theta) = 1$ (log 1 = 0)

💡 Verlust wird ∞ wenn $\pi(\text{action}|\theta) = 0$ (log 0 = -∞)

0 kommt nicht vor weil softmax = exp(…)/exp(∑…) immer positiv ist,
in der praxis +epsilon um ∞ fern zu bleiben

In der Praxis berechnet man aber wieder alles gleichzeitig also als **Tensoren.** loss = - logs @ returns (skalarprodukt)

Pause

```
Quizfrage: was meint unser Autor im deutschen Buch: ?
```

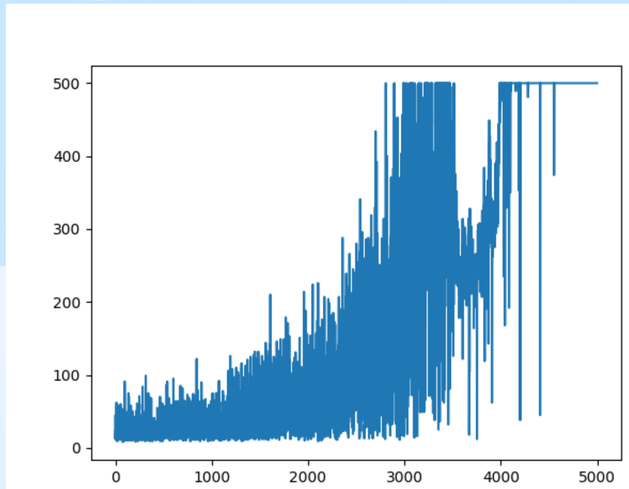
def loss_fn(preds, r): ◂ für die ergriffenen Aktionen und die diskontierten Belohnungen

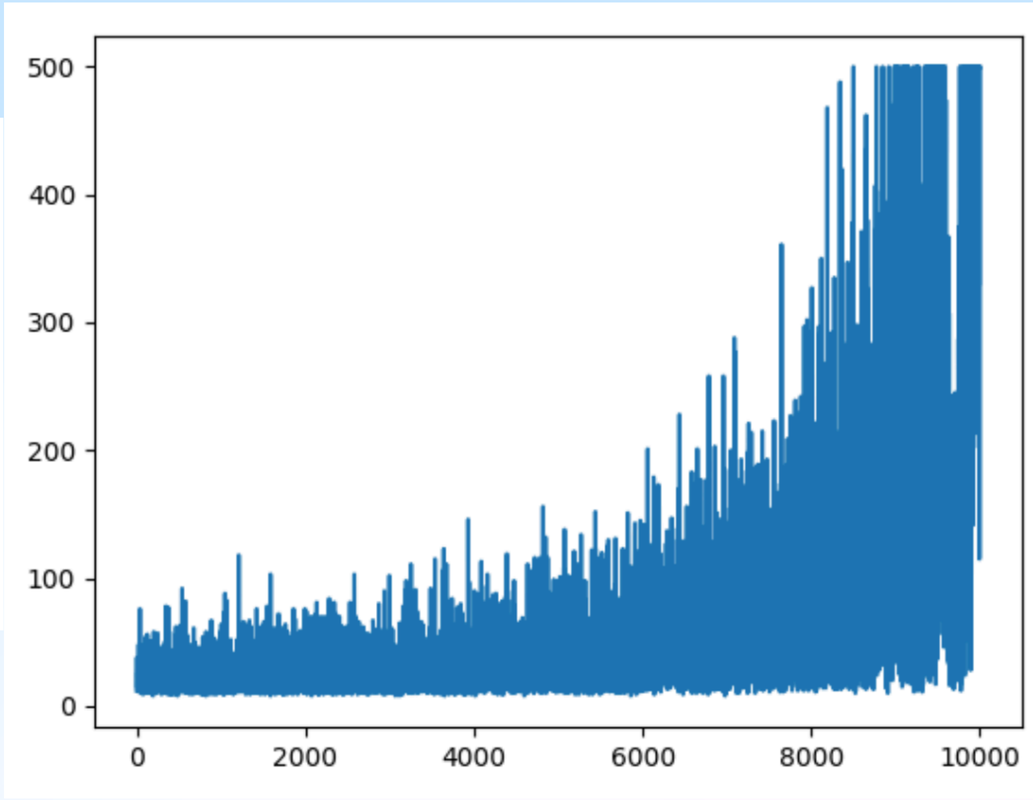
Rückgabe -1 * **Fackel**.summe(r * Fackel.log(preds))

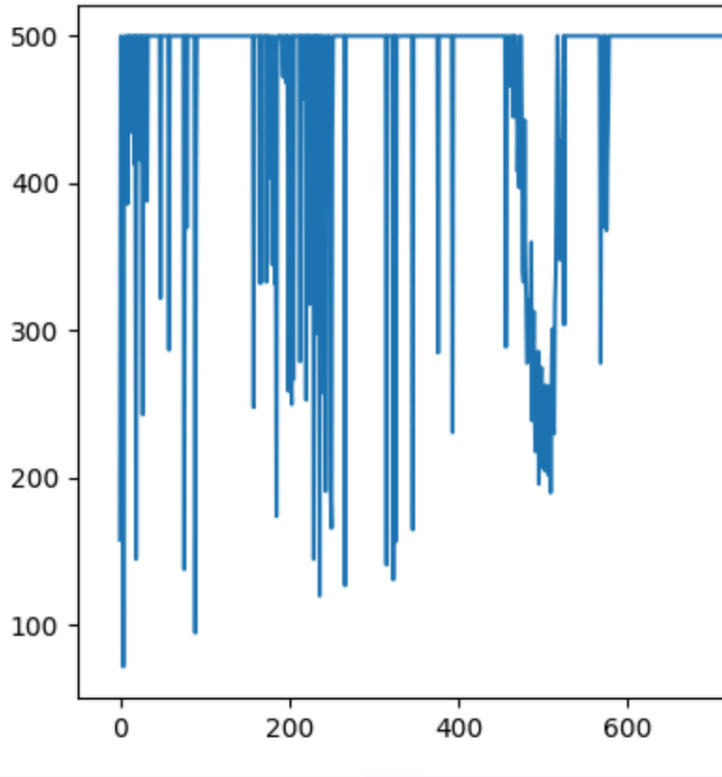
MC vs PG

PG Algorithmus funktioniert 'fast so gut' wie unser MC Algorithmus:
Mit Modifikationen bildet er die **Grundlage** für viele sehr gute Algorithmen.

MC:

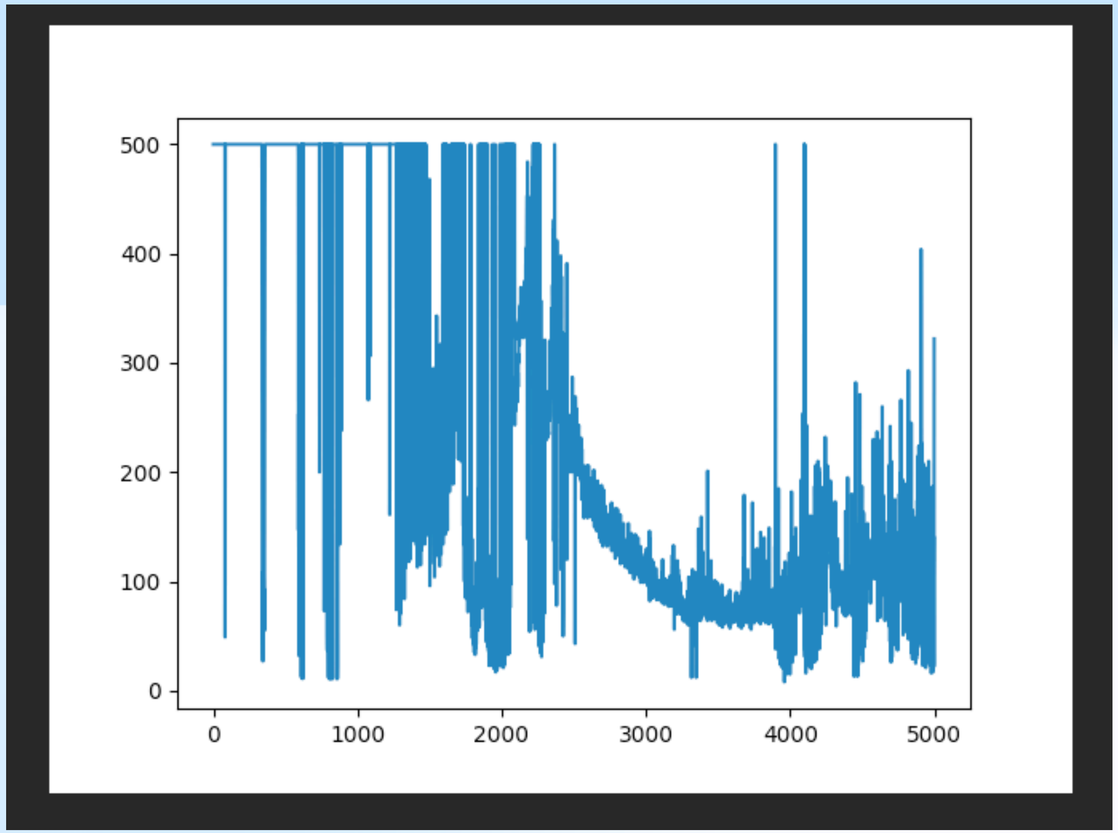


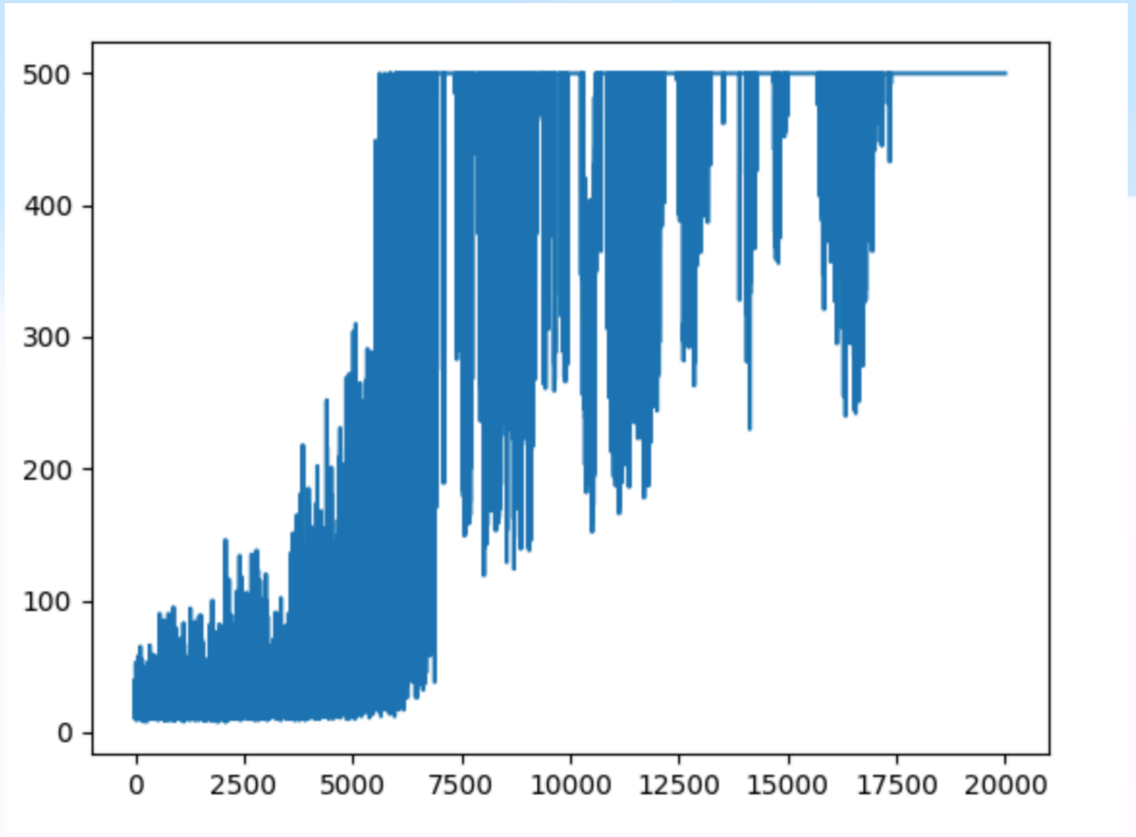




katastrophales Vergessen

PG Algorithmus ohne epsilon exploration funktioniert schlecht, ansonsten schon **ganz gut:**
Aber: braucht sehr viele Schritte zum lernen (stets bemüht)





REINFORCE

Erster guter Algorithmus, verdient den Namen *REINFORCE* (passend zu RL)

- wir schätzen **policy actions** (über softmax probabilities) < policy gradient pg
- **ε**-policy Action gemäß Wahrscheinlichkeit
- standard training Schleife ("sarsa")
- **Gewinn** wird **pro episode** zusammengezählt (MC Monte Carlo)

kontinuierliche Aktionen

Bei **kontinuierlichen Aktionen:**
z.B. Lenker drehen um **r** Grad, beschleunige **x** km/h

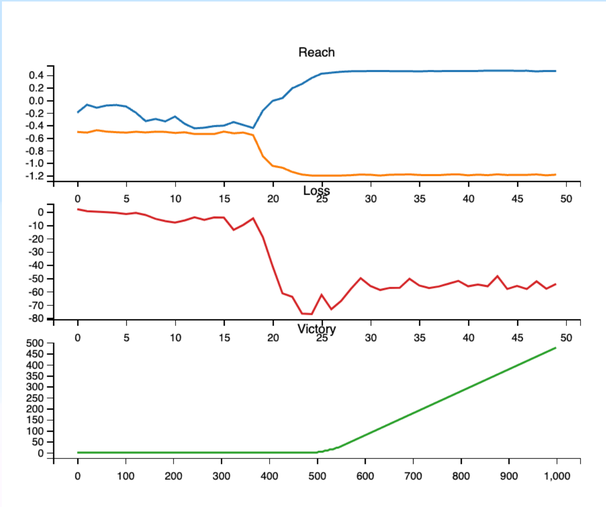
Ausgabe ist dann nur noch eine Zahl!

```
policy = nn.Sequential(
            nn.Linear(observation_size, hidden_size),
            nn.ReLU(),
            nn.Linear(hidden_size, 1), # direkter output von kontinuierlicher Aktion
         # nn.Tanh() # Optional, helfe dem Netz -1 bis 1 auszuwerfen
)
```

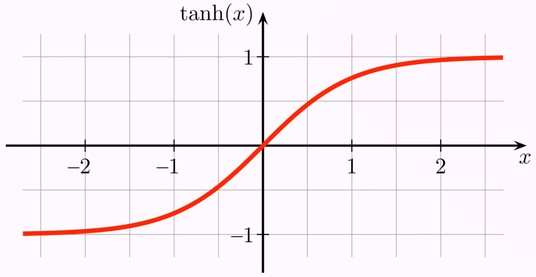
Ohne Softmax, Aktionen werden dann als **Mittelwert der Aktionswahrscheinlichkeiten** interpretiert!
Warum so kompliziert? Kann man nicht **Ausgabe vom Netz direkt als Aktion** interpretieren?
Orakel: "Könnte man aber funktioniert bei weitem nicht so gut"

Continuous REINFORCE

Für Probleme mit **kontinuierlich**en actions lässt sich reinforce verwenden indem man die Ausgabe des Netzwerk **direkt** in der Formel verwendet! Das ist *keine standard Theorie*!




```
def PG_unorthodox():
  for each episode:
     …
     action = π(s)
     G_T=0
     for t in reverse(trajectory):
       G_t = r_t + γ*G_t+1 # Gewinn goal gain
       loss = -∑ abs(action) * G
       θ ← θ - α ∇ loss  # Adam
```

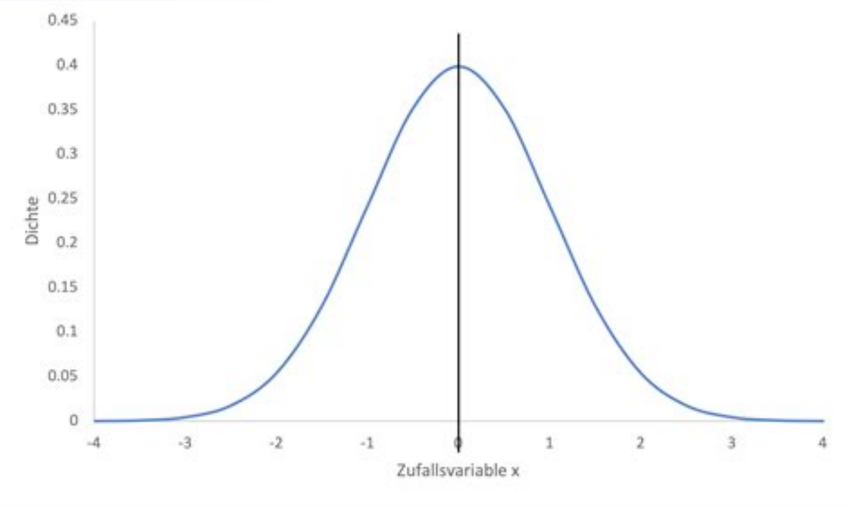


Continuous REINFORCE

Für kontinuierliche / regressions- Probleme lässt sich REINFORCE™
verwenden indem man die Ausgabe des Netzwerk als **Mittelwert** einer **Normalverteilung** interpretiert.

statt ausgabe vom Netz direkt als Aktion zu nehmen, baut man noch mal eine Verteilung drum herum!?! *(warum???)*

μ = **π**(s) [-1,1] # z.B. .5 => 50% Schub nach rechts oder 51%?
σ=.1 bleibe nahe am Mittelwert **um die Aktion**
action = nicht direkt letzte Ausgabe sondern
action = sample(N(μ,**σ**)) Normal-verteilt um μ



damit kann man den **loss unverändert** verwenden!

Continuous REINFORCE

Für kontinuierliche / regressions- Probleme lässt sich reinforce verwenden indem man die Ausgabe des Netzwerk als **Mittelwert** einer **Normalverteilung** interpretiert.

mean_action = **π**(s) [-1,1]
action = sample(**N**(mean_action, derivation) Normalverteilt

Damit ist die **log**-PDF (probability density function) dann wohldefiniert als:

- The PDF is:

$$p(a) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{a^2}{2}\right) \Rightarrow \log p(a) = -\frac{a^2}{2} - \log\sqrt{2\pi}$$

Aufgaben

Aufgaben Tag 7

a) Implementiere **PG** für ein Problem deiner Wahl **mit copy paste von Vorlage**

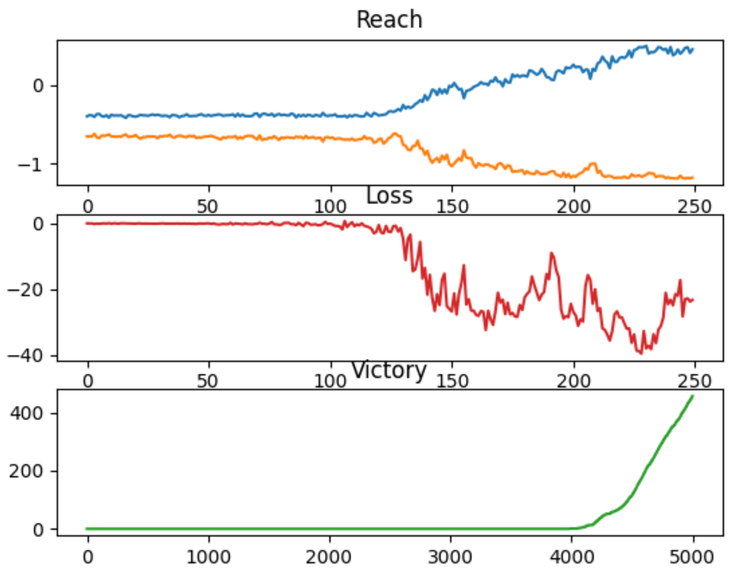
b) Implementiere PG für ein Problem deiner Wahl **mit GPT als Berater (nicht copy paste von GPT)**

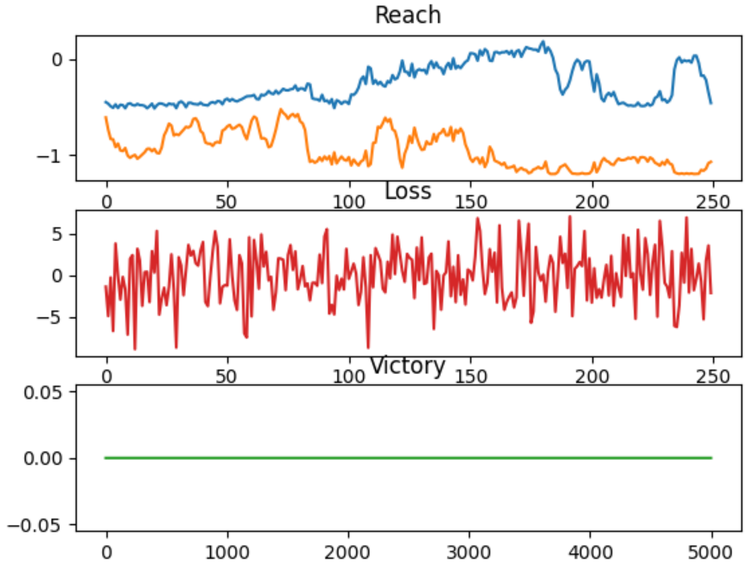
c) Implementiere ggf. **alt**en Algorithmus für pole ((D)QL)

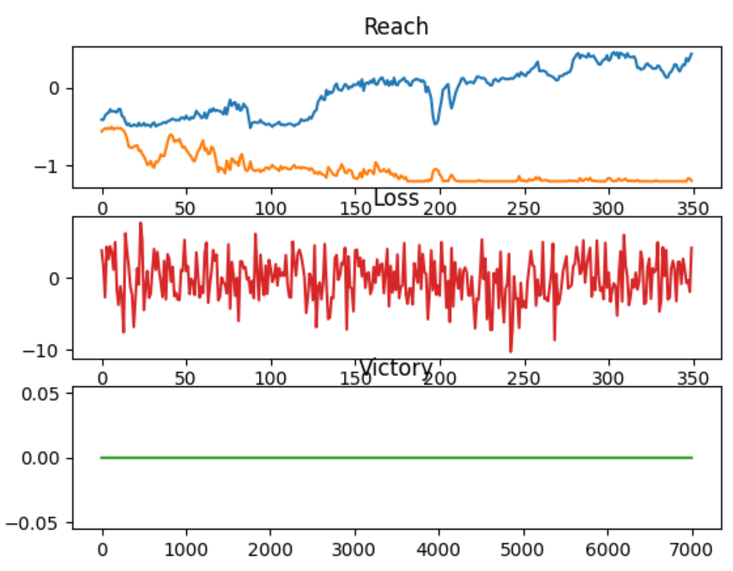
d) implementiere **batch** für pole (mehrere spiele gleichzeitig ohne gui)

e) implementiere reinforce **replay** für pole. alle 10 spiele trainieren.

Tipp : entferne , render_mode="human" bei env zum schnellen trainieren! continuous HARDER:

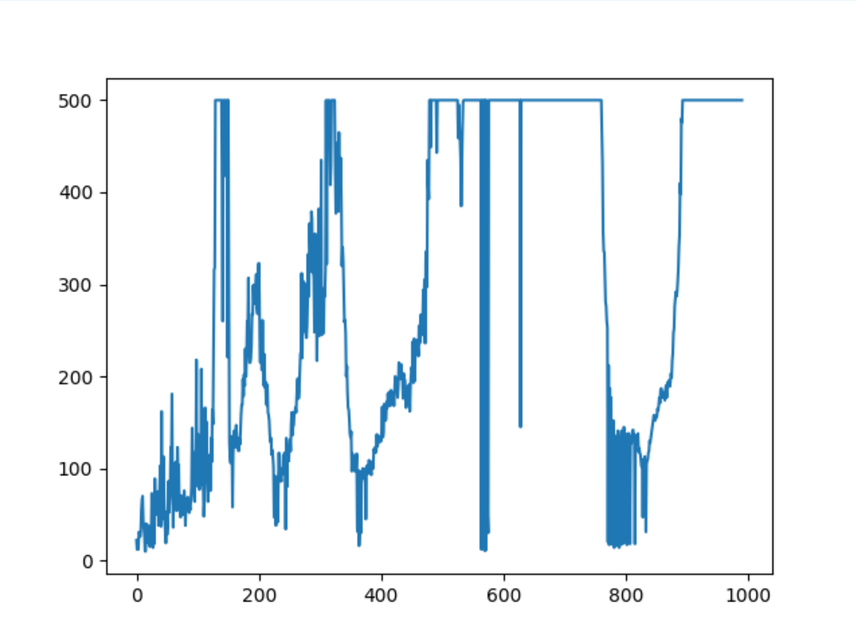


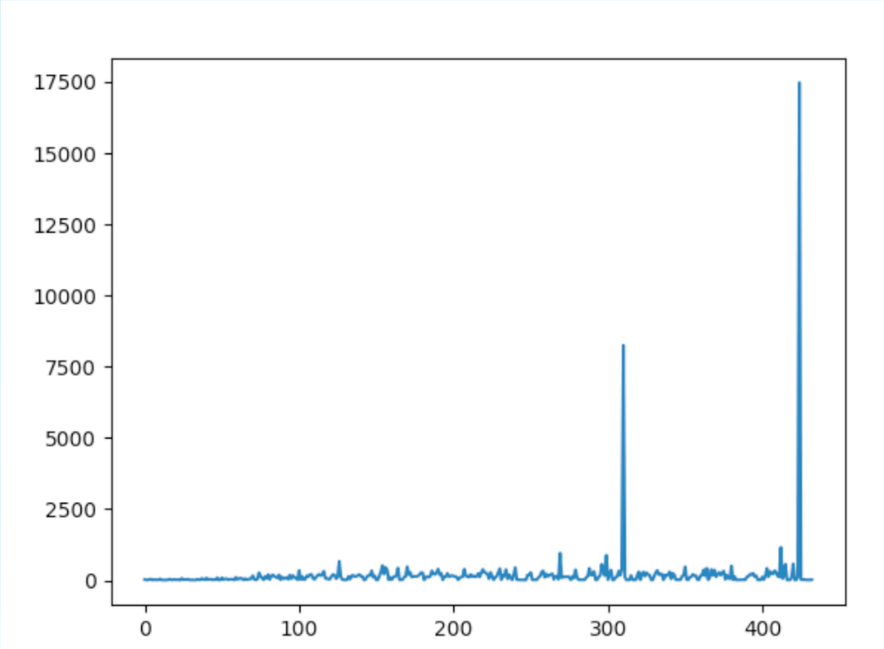




z.B. mountain_car https://www.gymlibrary.dev/environments/classic_control/mountain_car/ Example 9.2: Mountain–Car Task @ Sutton

Katastrophales Vergessen





Beim Fortschrittsverlauf vom pole Problem fällt auf, dass die policy gelegentlich *schlagartig schlechter* wird.
Das nennt sich **katastrophales Vergessen,** hier: **plötzlicher Kollaps** nach sehr guten Episoden.
In den kommenden Tagen lernen wir **Regularisierungs-Methoden**, dies zu verhindern.
Was wir heute schon kennen:

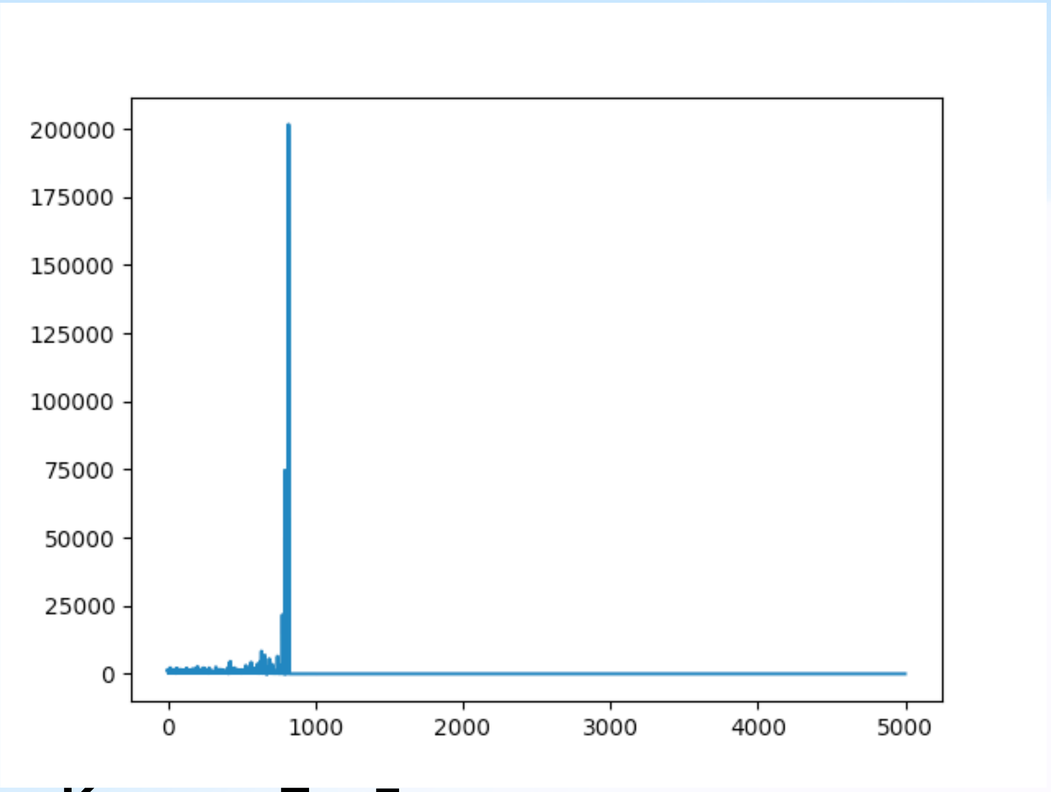
1) **replay (MC) buffer**
2) **batch (mehrere Spiele replays sammeln)**
3) **L2 norm, Netz Aspekte: skip connections**

Extremes Vergessen

Nach extrem langen erfolgreichen durchläufen (z.B. bei cart pole) kommt das Netzwerk **schlagartig** in einen sehr **schlechten Zustand**.

a) **warum**? Vermutung:
   Gewichte für erfolgreiche Aktionen nahe Anfangs-States werden so lange nicht benutzt, dass sie *entgleiten*.

b) wie läßt sich das **verhindern**?



Ursachen von Kollaps

1. **Verteilungsverschiebung (distribution shift):**

Während erfolgreicher Episoden werden **frühe States kaum noch besucht**.

Das Replay-Buffer enthält fast nur späte States.

Dadurch verlieren die Gewichte, die für frühe Zustände wichtig sind,

ihre Stützung und können durch zufällige Gradientenänderungen ***entgleiten***. (infinity / division by zero)

2. **Stark korrelierte Updates:**

Wenn fast alle gesammelten Transitionen „ähnlich“ sind (lange balanciert, kleine Abweichungen), dann gibt es wenig Diversität im Training. Das Netzwerk wird **überangepasst** und kann an Robustheit verlieren.

3. **Explodierende Q-Werte oder Policy-Verzerrung:**

Besonders bei DQN-artigen Methoden kommt es leicht vor,

dass bestimmte **Q-Werte extrem anwachsen** und das Training destabilisieren.

Gegenmaßnahmen bei Kollaps

Wie läßt sich plötzlicher Kollaps nach sehr guten Episoden verhindern?

- ***Replay Buffer* diversifizieren:**

Prioritized Experience Replay oder explizit sicherstellen, dass auch **alte frühe Zustände länger im Buffer** bleiben (z. B. reservoir sampling oder künstliche Gewichtung früher Episoden).

- **Exploration erzwingen:**

ε-greedy nicht zu schnell abkühlen lassen. Ein kleines **Rest-ε** verhindert, dass seltene States völlig verschwinden.

- ***Target Network* stabilisieren:**

Update-Frequenz und Soft-Updates so einstellen, dass keine plötzlichen Sprünge entstehen.

- **Reward Clipping oder R/G/Q-Wert-Normalisierung:**

Begrenzen, damit keine numerische Explosion eintritt.

- **Curriculum-ähnliche Trainingsstrategie:**

Ab und zu Training auf künstlich erzeugten „schlechten Starts“ (z. B. CartPole initial leicht gekippt), damit die Policy nicht nur in idealisierten Trajektorien trainiert.

- **Regularisierung via Chatty:**

L2-Gewichtsnorm oder Parameter-Noise zur Stabilisierung, damit selten genutzte Gewichte nicht beliebig weit abdriften.

discrete / continuous

Wir haben nun folgende Techniken um **alle diskreten / kontinuierlichen Probleme** zu lösen:

diskreter state-space**:**

one-hot encoding mnist/taxi [1,0,0,0...]

diskreter action-space**:**

DQN / softmax PG (standard)

continuous state-space:

a) just use NN (standard)

b) Unendliche Zustände in "Töpfe" unterteilen

continuous <u>action</u>-space:

a) PG mit Mittelwert - ***Normalverteilung*** sampling (siehe mountainCar_PG_REINFORCE.py)

b) Actionspace **diskretisieren (⚠️ eventuell sub-optimal)**

siehe pole-q_learning.py oder pendulum_mc.py

```
discrete_actions = np.linspace(env.action_space.low[0], env.action_space.high[0], num=10) # Bereiche / Segmente
```

c) unorthodoxe Ansätze (state+action) als input für NN, action direkt in loss? (siehe pole_policy_gradient_direct.py)

NEXT: multi actions! (Gas geben und lenken gleichzeitig)

Aufgabe

Aufgabe: Mondlandung 🚀🌒

`LunarLander-v3 https://gymnasium.farama.org/environments/box2d/lunar_lander/`

Stabilisierung notwendig:

🧿 LunarLander rewards can be −300 or worse.
Without normalization, G accumulates to huge values over 300+ steps,
the gradient update blows up the weights, and softmax produces NaN.

Two-part fix:
1. **Return normalization** ((G − mean) / std) −
keeps gradient magnitudes stable regardless of reward scale
2. **Gradient clipping** (clip_grad_norm_(..., 1.0)) −
safety net that caps the gradient norm before each weight update

Eigenes gymnasium environment

Beispeil eigene gym env (BudgetEnv.py)

Sehr einfache API:

- Vererbe von Klasse gym.Env
- initialisiere und
- definiere self.observation_space , self.action_space
- implementiere def step(self, action):
- return observation, reward, done, {}

damit können wir dann unsere algos (oder baseline) drauf los lassen!

Auch als wrapper für andere Probleme!

Vokabular

PG Policy Gradient (fast) direkte Berechnung der Aktion über policy Netzwerk (Algo und Klasse von Algos)

logprob **log**arithmierte **prob**abilities **log(π) Vektor** als Teil unserer Loss Formel

multinomial/categorical sampling: hole konkret eine Aktion aus den Aktions-Wahrscheinlichkeiten

REINFORCE (standard Algorithmus base PG + MC siehe mountainCar_PG_REINFORCE.py **)**

observation ≈ state

Monte Carlo replay "Sampling von Episoden"

```
detach().item() hole konkrete Zahl aus Vektor (kein Tensor mehr für backprop)
discrete / continuous
torch.distributions.Normal(mu, std)  kontinuierliche Aktion um Mittelwert
```

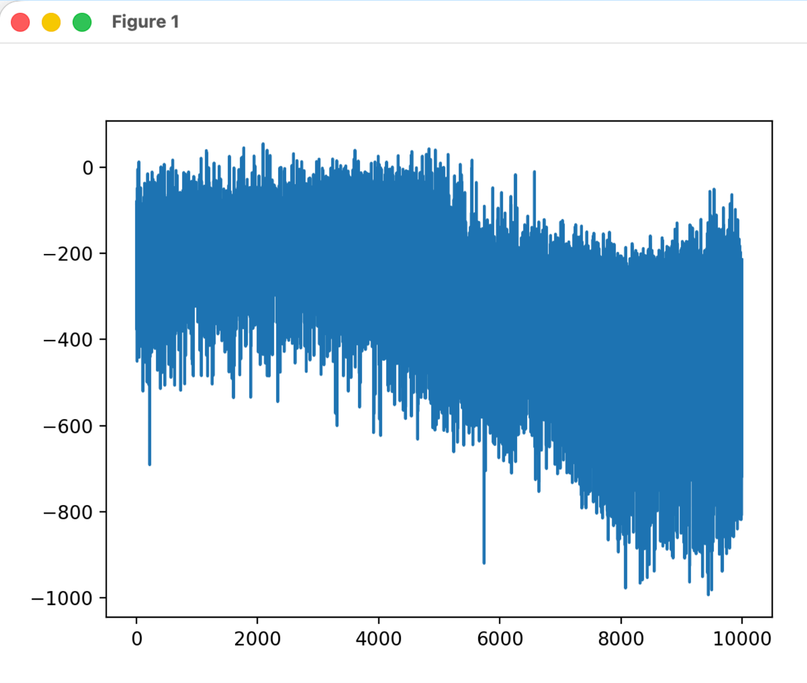
Aufgabe

Nachtrag:

a) acrobot mc DQN mc (quality) acrobot_mc.py Lösung in tag 6

b) acrobot PG REINFORCE (policy)

Nachtrag:
a) pendulum DQN mc (quality)
b) pendulum PG REINFORCE (policy)



Aufgabe

Aufgabe: Mondlandung 🚀🌒

`LunarLander-v3 https://gymnasium.farama.org/environments/box2d/lunar_lander/`

ist das continuous oder discrete?

Kopiere pole**_pg_**mc.py und passe es an

Statt eigener transform_tensor Funktion, kann man das direkt im Netz einbauen? torch.Flatten torch.onehot?