Rapport

RCP211

Génération de scénarios et renforcement

Élèves :
Said Taghadouini
Yakine Tahtah

Enseignant :
Nicolas THOME

Version de
23 juin 2022

Table des matières

Table des figures

1 Introduction

L'objectif de ce projet est de construire un modèle capable d'échantillonner des objectifs pour un agent qui devra les atteindre pendant le test. Ce projet se décline donc en deux parties : une partie qui porte sur un modèle génératif censé générer des objectifs à atteindre en échantillonnant une distribution apprises sur des données. On a choisi d'utiliser un $\beta - VAE$ pour le modèle génératif. La deuxième partie consiste à développer un agent capable d'apprendre à atteindre les objectifs générés par le modèle génératif. Pour le cas d'étude on a choisi de travailler sur les échecs en utilisant la bibliothèque **python-chess**. On s'inspire principalement du papier : *Visual Reinforcement Learning with Imagined Goals*.

2 Théorie

2.1 RIG : Reinforcement learning with Imagined Goals

L'article développe un algorithme pour avoir un agent autonome capable d'atteindre des objectifs divers spécifiés par l'utilisateur au lieu d'être spécialisé sur un objectif très précis. Le cas d'usage de l'article est un bras de robot qui va interagir avec différents éléments de l'environnement comme le montre l'image ci-dessous extraite de l'article :

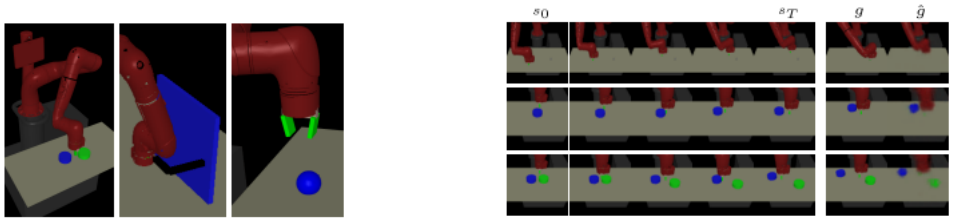


Figure 2: (Left) The simulated pusher, door opening, and pick-and-place environments are pictured. (Right) Test rollouts from our learned policy on the three pushing environments. Each row is one rollout. The right two columns show a goal image g and its VAE reconstruction $\hat{g}$. The images to their left show frames from a trajectory to reach the given goal.

FIGURE 1 – Le cas d'usage de l'expérience pour RIG

L'idée de base est d'utiliser un modèle génératif (dans l'article un β-VAE) pour générer des objectifs dans une phase auto-supervisée puisque les objectifs à accomplir pendant le test ne sont pas connus d'avance. Et dans ce cas nous avons aussi la capacité de calculer le signal de récompense grâce à la distance dans l'espace latent du β-VAE entre l'objectif à atteindre et l'état actuel, ce qui est très efficaces dans certains cas comme celui de données images ou dans notre cas où nous n'avons pas de distance directe entre deux échiquiers.
On peut résumer l'algorithme de RIG par l'entraînement d'un β-VAE sur des données obtenues par l'agent en suivant une politique exploratoire puis par l'entraînement de l'agent en échantillonnant des objectifs de l'espace latent et en cherchant la politique conditionnée sur cet objectif. Au cours de l'entraînement de l'agent le β-VAE est affiné après chaque nombre d'itérations. Tous les états et objectifs sont dans l'espace latent du β-VAE, ce qui en fait une partie clé de l'algorithme RIG. Si par exemple le β-VAE est mauvais, on pourrait avoir un état proche de l'objectif dans l'espace latent mais qui indiqueraient deux configurations très différentes dans l'espace des features, ou l'inverse, l'agent se serait approché de l'objectif dans l'espace features (dans la "réalité"), mais le signal de récompense indiquerait un grand écart dans l'espace latent, ou même on pourrait échantillonner des objectifs qui sont incapables à atteindre. D'où l'importance d'avoir un bon β-VAE et qui sera constamment affiné.

Algorithm 1 RIG: Reinforcement learning with imagined goals

Require: VAE encoder q_ϕ, VAE decoder p_ψ, policy π_θ, goal-conditioned value function Q_w.

Collect $\mathcal{D} = \{s^{(i)}\}$ using exploration policy.
Train β-VAE on $\mathcal{D}$ by optimizing (2).
Fit prior $p(z)$ to latent encodings $\{\mu_\phi(s^{(i)})\}$.
for $n = 0, ..., N-1$ episodes **do**
  Sample latent goal from prior $z_g \sim p(z)$.
  Sample initial state $s_0 \sim E$.
  for $t = 0, ..., H-1$ steps **do**
    Get action $a_t = \pi_\theta(e(s_t), z_g) +$ noise.
    Get next state $s_{t+1} \sim p(\cdot \mid s_t, a_t)$.
    Store (s_t, a_t, s_{t+1}, z_g) into replay buffer $\mathcal{R}$.
    Sample transition $(s, a, s', z_g) \sim \mathcal{R}$.
    Encode $z = e(s), z' = e(s')$.
    (Probability 0.5) replace z_g with $z'_g \sim p(z)$.
    Compute new reward $r = -||z' - z_g||$.
    Minimize (1) using (z, a, z', z_g, r).
  end for
  for $t = 0, ..., H-1$ steps **do**
    for $i = 0, ..., k-1$ steps **do**
      Sample future state $s_{h_i}, t < h_i \leq H-1$.
      Store $(s_t, a_t, s_{t+1}, e(s_{h_i}))$ into $\mathcal{R}$.
    end for
  end for
  Fine-tune β-VAE every K episodes on mixture of $\mathcal{D}$ and $\mathcal{R}$.
end for

FIGURE 2 – Pseudo-code de RIG

2.2 Variational Auto-encoders

L'idée des auto-encodeurs variationnels est différente de celle des autres auto-encodeurs. Au lieu d'apprendre une association entre chaque entrée à un vecteur fixe, on cherche à apprendre un mapping vers une distribution. On note cette distribution par p_θ avec un paramètre θ. La relation entre les données d'entrée x et l'encodage dans l'espace latent z peut être définie par :

— Distribution a priori $p_\theta(z)$
— Vraisemblance $p_\theta(x|z)$
— Distribution a posteriori $p_\theta(z)$

Et une fois qu'on a le vrai paramètre θ de cette distribution on peut l'utiliser pour générer des exemples qui sont similaires aux données d'entraînement $x^{(i)}$, on peut faire :

— Échantillonner un $z^{(i)}$ à partir d'une distribution a priori $p_\theta(z)$
— Après un point $x^{(i)}$ est généré à partir d'une distribution conditionnelle $p_\theta(x|z = z^{(i)})$

Pour trouver le meilleur paramètre θ^* est celui qui maximise la probabilité de génération des vraies données(i.e similaires aux données d'entraînement) :

$$\theta^* = \underset{\theta}{argmax} \sum_{i=1}^{n} log p_\theta(x^{(i)})$$

Mais pour calculer $p_\theta(x^{(i)})$ il faut intégrer sur toutes les valeurs possibles de $\mathbf{z}$, ce qui est très lourd à faire. Pour cela, on introduit une nouvelle approximation qui va donner la valeur de $\mathbf{z}$ la plus probable étant donné une entrée $\mathbf{x}$, $q_\Phi(z|x)$, paramétrée par Φ.

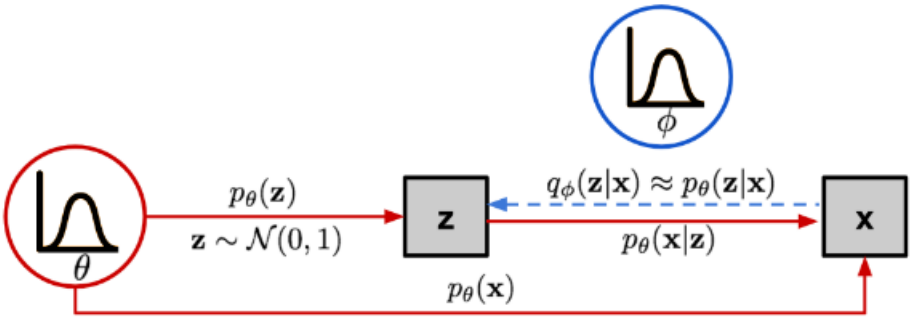


FIGURE 3 – Le modèle graphique des auto-encodeurs variationnels, $q_\Phi(z|x)$ est l'encodeur et $p_\theta(x|z)$ est le décodeur

2.2.1 Loss function : Evidence lower bound(ELBO)

Le postérieur estimé $q_\Phi(z|x)$ doit être très proche du réel $p_\theta(z|x)$. Nous pouvons utiliser la divergence de Kullback-Leibler pour quantifier la distance entre ces deux distributions. La divergence KL $D_{KL}(X|Y)$ mesure la quantité d'informations perdues si la distribution Y est utilisée pour représenter X.

Dans notre cas on cherche à minimiser $D_{KL}(q_\Phi(z|x)|p_\theta(z|x))$ par rapport à Φ.

$$L_{VAE}(\theta, \Phi) = -\mathbb{E}_{z\sim q_\Phi(z|x)} log p_\theta(z|x) + D_{KL}(q_\Phi(z|x)|p_\theta(z))$$

Dans les méthodes bayésiennes variationnelles, cette fonction de perte est connue sous le nom de limite inférieure variationnelle ou limite inférieure de preuve. La partie "limite inférieure" dans le nom vient du fait que la divergence KL est toujours non négative et donc $-L_{VAE}$ est la limite inférieure de $log p_\theta(x)$.

2.2.2 L'astuce de reparamétrisation :

Le terme d'espérance dans la fonction de perte appelle la génération d'échantillons à partir de $z \sim q_\Phi(z|x)$. L'échantillonnage est un processus stochastique et nous ne pouvons donc pas rétropropager le gradient. Pour le rendre entraînable, l'astuce de reparamétrisation est introduite : il est souvent possible d'exprimer la variable aléatoire $\mathbf{z}$ comme une variable déterministe $z = T_\Phi(x, \epsilon)$, où ϵ est une variable aléatoire indépendante auxiliaire, et la fonction de transformation T_Φ paramétrée par Φ convertit ϵ en $\mathbf{z}$.

Par exemple, un choix courant de la forme de $q_\Phi(z|x)$ est une gaussienne multivariée avec une structure de covariance diagonale.

L'astuce de reparamétrisation fonctionne également pour d'autres types de distributions, pas seulement gaussiennes. Dans le cas gaussien multivarié, nous rendons le modèle entraînable en apprenant la moyenne μ et la variance σ de la distribution, et , en utilisant explicitement l'astuce de reparamétrisation, tandis que la stochasticité reste dans la variable aléatoire $\epsilon \sim N(0, I)$.

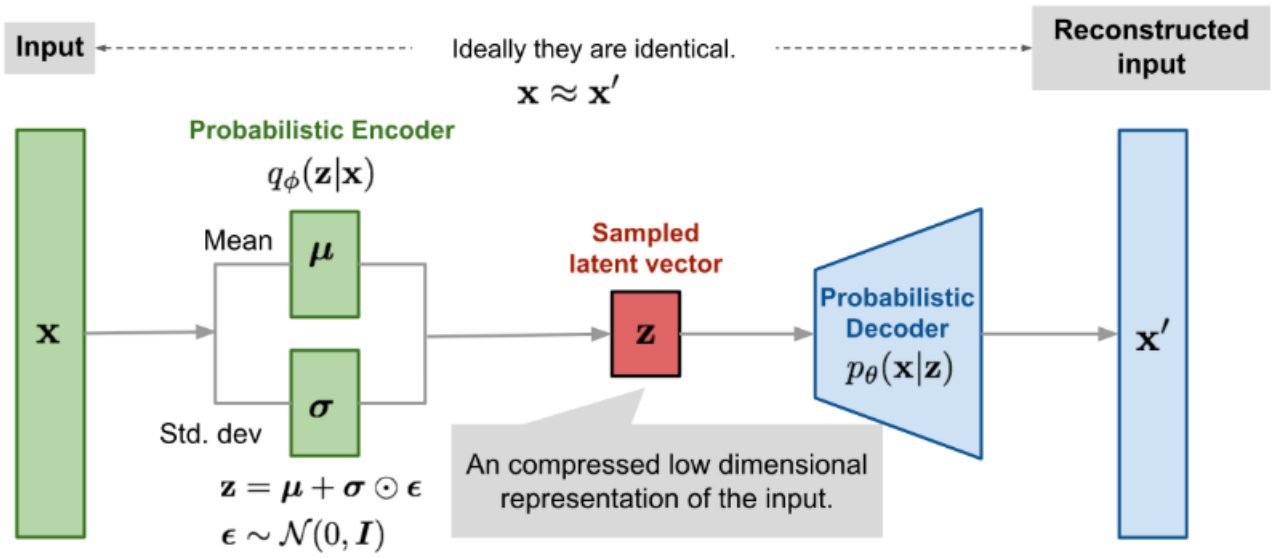


FIGURE 4 – Illustration du modèle d'auto-encodeur variationnel avec l'hypothèse gaussienne multivariée.

2.2.3 $\beta - VAE$

Si chaque variable de la représentation latente inférée n'est sensible qu'à un seul facteur génératif et relativement invariante aux autres facteurs, on dira que cette représentation est factorisée. Un avantage qui accompagne souvent une représentation factorisée est une bonne interprétabilité et une généralisation facile à une variété de tâches.

Par exemple, un modèle formé sur des photos de visages humains peut capturer la douceur, la couleur de la peau, la couleur des cheveux, la longueur des cheveux, l'émotion, le fait de porter une paire de lunettes et de nombreux autres facteurs relativement indépendants dans des dimensions distinctes. Une telle représentation factorisée est très bénéfique pour la génération d'images faciales.

$\beta - VAE$ (Higgins et al., 2017) est une modification de Variational Autoencoder avec un accent particulier pour découvrir les facteurs latents factorisés. En suivant la même incitation en VAE, nous voulons maximiser la probabilité de générer des données réelles, tout en gardant la distance entre les distributions a posteriori réelles et estimées petite (par exemple, sous une petite constante δ) : Cela se traduit au niveau de la fonction coût par l'introduction d'un multiplicateur de Lagrange β sous la forme suivante :

$$L_{\beta-VAE}(\theta, \Phi) = -\mathbb{E}_{z\sim q_\Phi(z|x)} log\ p_\theta(x|z) + \beta\ D_{KL}(q_\Phi(z|x)|p_\theta(z))$$

Quand $\beta = 1$, c'est la même chose que le VAE. Lorsque $\beta > 1$, il applique une contrainte plus forte sur l'espace latent et limite la capacité de représentation de $\mathbf{z}$. Pour certains facteurs génératifs conditionnellement indépendants, les garder factorisés est la représentation la plus efficace. Par conséquent, un niveau supérieur de β encourage un codage latent plus efficace et encourage davantage la factorisation des facteurs génératifs. De même, une valeur plus élevée de β peut créer un compromis entre la qualité de la reconstruction et l'étendue du démêlage.

Pour avoir des résultats qui ressemblent plus la réalité on doit aussi utiliser une fonction de coût de reconstruction qui va mesurer l'erreur commise sur la reconstruction d'un exemple d'entraînement. Cette erreur peut être n'importe quelle fonction coût qui peut mesurer une sorte de distance entre la vraie valeur et la reconstruction.

Dans ce projet on choisit d'utiliser **Binary Cross Entroppy(BCE)** car la représentation de l'échiquier est *numpy.array* de taille $(12, 8, 8)$ et qui contient des valeurs 0 et 1 ce qui justifie l'utilisation de cette métrique pour comparer la reconstruction à l'entrée élément par élément. Le calcul pour un batch de taille N est alors :

$$-\frac{1}{N}\sum_{n=1}^{N}\sum_{i=1}^{12}\sum_{j=1}^{8}\sum_{k=1}^{8} y_{n,i,j,k} log(p_{n,i,j,k})\ +\ (1 - y_{n,i,j,k}) log(1 - p_{n,i,j,k}))$$

Puisqu'à chaque observation x on associe une distribution gaussienne définie par sa normale et son écart-type, il est naturel de prendre comme code latent pour représenter x dans l'espace latent la moyenne de cette distribution. Et c'est ce que nous faisons lors du calcul de la récompense de notre agent comme on le verra plus tard.

2.3 Apprentissage par Renforcement

Pour résoudre notre problématique on utilise une approche *basée modèle* avec *Monte-Carlo Tree Search*. Cette approche se prête bien aux échecs puisqu'on connaît bien le modèle et qu'on peut faire de la simulation sur autant d'étapes qu'on veuille ou que le permette nos ressources de calcul. Aussi, l'apprentissage des fonctions de valeur, ou des politiques, en échecs directement est assez compliqué puisque deux états (différence dans la position d'une pièce) peut conduire à des résultats complètement différents (victoire ou perte). Par contre, en utilisant un modèle, on peut apprendre on

peut simuler l'interaction agent-environnement à partir de l'état actuel ce qui permet une meilleure construction des fonctions de valeur et/ou des politiques, contrairement à l'apprentissage direct par expérience.

2.3.1 Formalisme

Un processus de décision Markovien est un tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$, avec un facteur d'actualisation γ. Notre environnement est le plateau d'échecs et donné par **python-chess** et la représentation numérique des nos états est la suivante ; l'état t est caractérisé par un **numpy.array**, M, de taille $(8, 8, 8)$ où les 6 premières couches de l'array indiquent la position des différentes pièces encore présentes en jeu et avec $+1$ pour le joueur blanc (contrôlé par notre agent) et -1 pour le joueur noir (contrôlé par une politique *greedy* dont le but est de maximiser le gain en éliminant les pièces de l'adversaire et qui prend en compte le type de pièces car éliminer la reine de l'adversaire est plus important que d'éliminer/perdre un pion). Les pions, "p"/"P", sont représentés par la première couche. Les tours, "r"/"R", sont représentées par la deuxième couche. Les chevaliers, "n"/"N", sont représentés par la troisième couche. Les fous, "b"/"B", sont représentés par la quatrième couche. Les reines, "q"/"Q", sont représentées par la cinquième couche et les rois, "k"/"K", sont représentés par la sixième couche. Et on prend $M[6, :, :] = 1/fullMoveNumber$ où *fullMoveNumber* est le nombre de mouvements légales qu'on peut prendre. Et on prend aussi $M[7, :, :] = 1$. Ceci définit donc notre espace d'états $\mathcal{S}$.
Notre ensemble d'actions $\mathcal{A}$ est définit par les mouvements possibles de chaque pièce à chaque tour de notre agent.
Nos probabilités de transitions $\mathcal{P}$ sont données par les règles du jeu, ainsi, une pièce a une probabilité nulle de se mouvoir sur des cases de l'échiquier qui lui sont impossibles par les règles du jeu et la probabilité d'atteindre une case en effectuant un mouvement légal est de 1.
Notre fonction de récompenses $\mathcal{R}$ est moins la norme euclidienne dans l'espace latent entre le code de l'état actuel de l'échiquier et entre la configuration qu'il faut atteindre. Ainsi si la configuration objectif a un code z_g et l'état actuel possède un code z, la récompense est $-||z - z_g||$. Comme on l'a noté ci-dessus, le code z est la moyenne de la distribution associée à l'état actuel de l'échiquier.
On prend un facteur d'actualisation $\gamma \geq 0.9$ puisque ce sont des valeurs qui reviennent souvent dans la littérature de l'apprentissage par renforcement.

2.3.2 Approche basée modèle

L'approche basée modèle cherche à estimer le modèle $\mathcal{M}$, représentation du processus de décision Markovien $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R})$ sous-jacent, par apprentissage supervisé sur de l'expérience accumulée par l'agent en interagissant avec l'environnement. En paramétrisant le modèle par ω, alors on peut simuler l'interaction agent-environnement grâce à ce modèle $\mathcal{M}_\omega$ en échantillonnant $S_{t+1} \sim \mathcal{P}_\omega(S_{t+1}|S_t, A_t)$ et $R_{t+1} = \mathcal{R}_\omega(R_{t+1}|S_t, A_t)$. Les paramètres ω sont choisis de telle sorte à minimiser la fonction coût décidée pour construire notre modèle par apprentissage supervisé à travers l'expérience réelle où on considère cette dernière comme un ensemble de données (X, Y) de la façon suivante : $S_k, A_k \to R_{k+1}, S_{k+1}$.
Lorsque le modèle est estimé, l'agent peut utiliser une expérience simulée, imaginée, par le modèle pour planifier sur le long-terme et les techniques utilisées pour construire les fonctions de valeur et/ou les politiques sont les mêmes que celles de l'approche non-basée modèle. Lorsque la fonction de valeur et/ou la politique a été construite, celle-ci est utilisée pour interagir avec l'environnement et construire un nouveau ensemble de données pour affiner l'estimation de notre modèle. Et le cycle se répète.
Dans notre cas le modèle est direct. Le but est donc d'apprendre les fonctions de valeur en utilisant *Monte-Carlo Tree Search*.

2.3.3 Monte-Carlo Tree Search

L'idée de *Monte-Carlo Tree Search* est de résoudre une "un sous-processus de décision Markovien" de celui de base et qui a comme départ l'état dans lequel l'agent est actuellement. A partir de l'état s_t, pour chaque action $a \in \mathcal{A}$ qu'on peut prendre dans cet état, on simule plusieurs trajectoires d'expérience en utilisant notre modèle, et on estime la valeur de chaque paire état-action consécutive par moyenne des retours à partir de cet état, en choisissant cette action, jusqu'à la fin de la trajectoire. Cette manière de faire va permettre d'améliorer la prise de décision dans le choix des actions à sélectionner puisqu'on pourra choisir les actions qui maximisent les Q-valeurs, avec éventuellement un peu d'exploration. Alors que le choix pour les actions pour des états non rencontrés antérieurement se fera aléatoirement. Finalement, après avoir répétés N fois des simulations pour chaque action qu'on peut prendre à l'état actuel s_t, on choisit l'action qui maximise la Q-valeur.

3 Données

Pour entrainer le $\beta - VAE$ on a besoin d'avoir des données qui représentent des configurations de l'échiquier. Pour cela on s'est tourné vers le site **lichess**[2] qui donne accès aux parties d'échecs jouées enligne depuis 2013.

Les données sont sous format de texte que nous avons ensuite traité pour en extraire le déroulement de chaque partie avec les coups de chaque joueur. Nous avons ensuite utilisé la librairie python-chess pour convertir les données au format d'une configuration donnée de l'échiquier. Pour cela nous avons le choix de la configuration à utiliser pour entraîner le modèle génératif et puisque nous voulons apprendre la distribution sur l'espace des configurations de l'échiquier, nous pouvons se limiter seulement à un certain nombre de coups pour que l'objectif ne soit pas trop compliqué ou très loin pour l'agent qui devra l'atteindre.

On définit un nombre maximal de coups de chaque partie présente dans les données d'entraînement $max\ moves = 5$, cela permet de définir des configurations qui ne sont pas très compliquées car dans les échecs le chemin vers une configuration dans les dernières étapes de la partie est très difficile à trouver et il y a plusieurs chemins et cela dépend fortement de l'adversaire.

4 Détails d'implémentation

Nos ressources de calcul étant limitées, nous sommes obligés de nous écarter de l'algorithme RIG tel qu'il est présenté dans l'article pour alléger nos contraintes techniques.

Sachant qu'on utilise l'algorithme Monte-Carlo Tree Search pour sélectionner nos actions, et qu'on se repose sur un réseau de neurones pour comme approximateur de fonction, alors on donne aussi la configuration objectif comme entrée à cet réseau de neurone pour conditionner selon l'objectif. On représente l'objectif par une matrice $(12, 8, 8)$, semblable à l'entrée de notre β-VAE, qu'on obtient par décodage du code latent z_g de l'objectif. Nous n'utilisons pas le code latent directement avec le code latent de l'état actuel de l'échiquier tel que décrit dans l'algorithme RIG parce qu'il nous est difficile d'itérer sur plusieurs expériences afin de trouver l'architecture qui avec ces deux entrées donnerait un résultat satisfaisant. C'est pour cela qu'on concatène les deux représentations avant de les donner en entrée à notre réseau utilisé comme approximateur de fonction.

Pour l'algorithme Monte-Carlo Tree Search on ne considère que des arbres de profondeur 4 à chaque simulation à cause de nos ressources de calcul limitées. Ce qui nous procure un assez bon compromis entre temps de calcul et "horizon" de simulation puisque l'ennemi (le joueur noir) joue

selon une politique *greedy*. Cette décision de faire jouer le joueur noir selon une politique *greedy* au lieu du "self-play" usuel qu'on retrouve dans la littérature d'apprentissage par renforcement concernant les jeux similaires aux échecs (Shogi, Go) est aussi due à la contrainte technique des ressources de calcul.

Aussi nous n'affinerons pas notre β-VAE pendant l'exécution de l'algorithme RIG parce que nous avons jugé avoir assez de données de base pour le β-VAE contrairement au cas d'usage dans l'article. Comme ça on peut garder plus de ressources de calcul pour l'agent aussi.

5 Résultats et Analyses

5.1 Résultats β-VAE

Afin de trouver la meilleure valeur de l'hyperparamètre β on réalise un fine-tuning. Pour le β-VAE avec une valeur de $\beta = 10^{-3}$, on obtient les résultats suivants. A noter que nous avons rajouter un dataset de validation juste dans le but de s'assurer de la performance du modèle génératif à générer des configurations valables. On a aussi essayé plusieurs valeurs de β mais pour notre cas $\beta = 10^{-3}$ était la valeur qui générait des configurations logiques ou non évidente par exemple certaines valeurs de β on obtient la même distribution pour toutes les valeurs de l'espace latent. Notamment, les valeurs de élevée de *beta* conduisent à une divergence KL très faible ou même nulle.

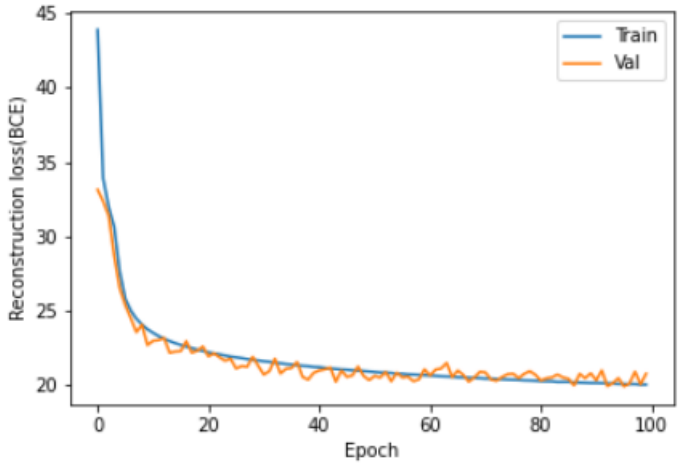


FIGURE 5 – Reconstruction loss : Binary Cross Entropy Loss

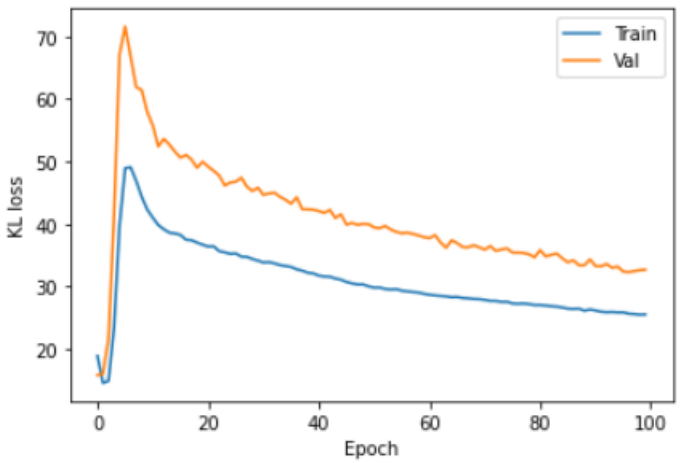
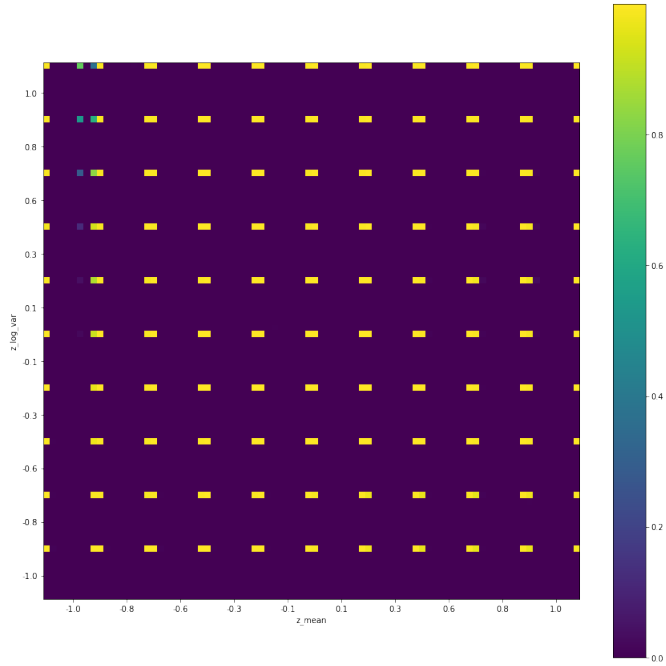


FIGURE 6 – Divergence KL

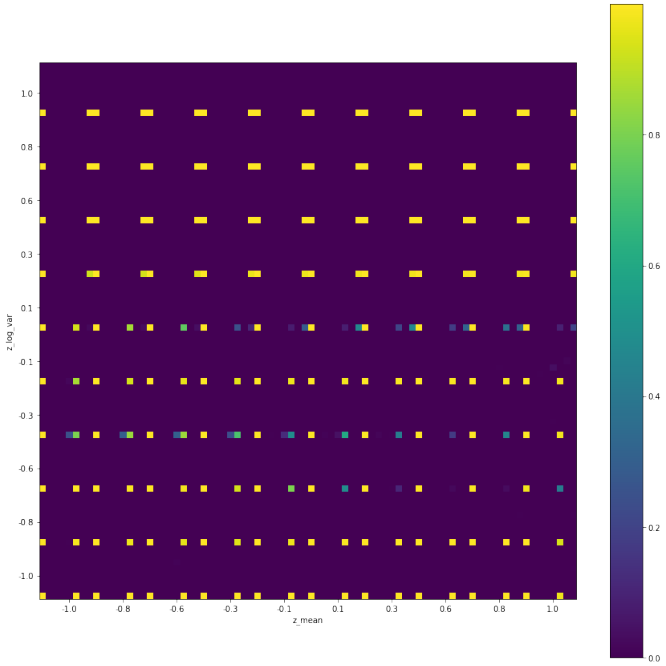
On peut voir sur les figures d'apprentissages que le modèle apprend bien la distribution de l'entrée et cela sur les deux composantes de la fonction coût $L_{\beta-VAE}$. On peut aussi visualiser la distribution de la probabilité d'avoir chaque pièce sur une case de l'échiquier dans l'espace latent, c'est-à-dire dans l'espace paramétrée par la moyenne μ et σ.

Sur les figures 7, 8 et **??**, on peut voir les différentes distributions des probabilités de présence des pièces dans une case de l'échiquier : la tour et le cavalier à la fois pour le joueur noir et le joueur blanc. Premièrement, la tour noire commence le jeu en haut à droite et à gauche(car il y a deux tours noires). L'espace latent est paramétrée par z_{mean} et z_{log_var} qui sont les coordonnées de l'espace latent de dimension 2 du $\beta-$VAE. Ces deux paramètres paramétrisent une distribution sur l'espace des états des pièces.

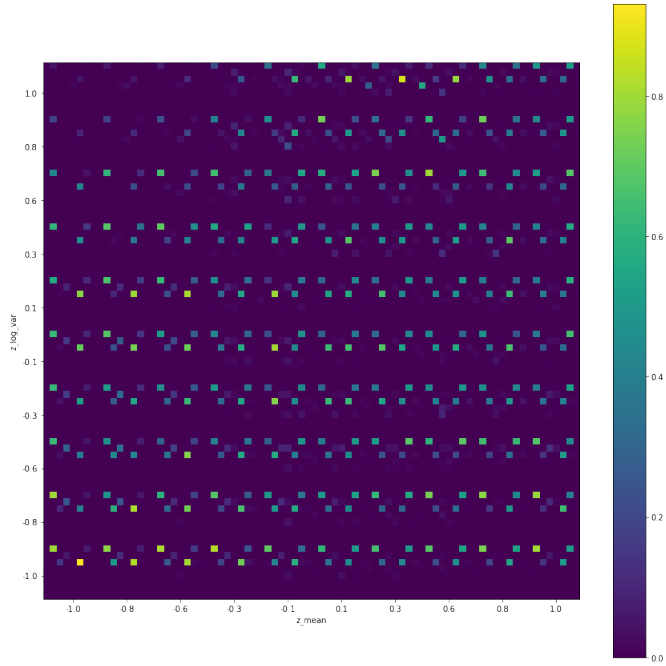
On fait varier les valeurs de z_{mean} et z_{log_var} entre $[-1, 1]$ et on observe l'effet sur la distribution : intensité de la probabilité et la position la plus probable de chaque pièce. Visuellement on peut vérifier que les positions de chaque pièce sont bien logiques et cohérentes avec les règles du jeu(comment chaque pièce bouge sur l'échiquier, notamment les feus, les cavaliers, etc..).



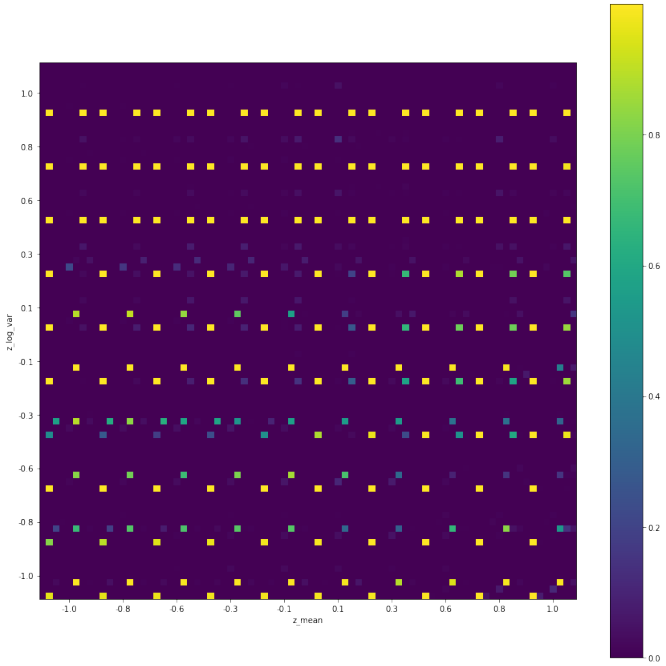
(a) La tour noire



(b) La tour blanche

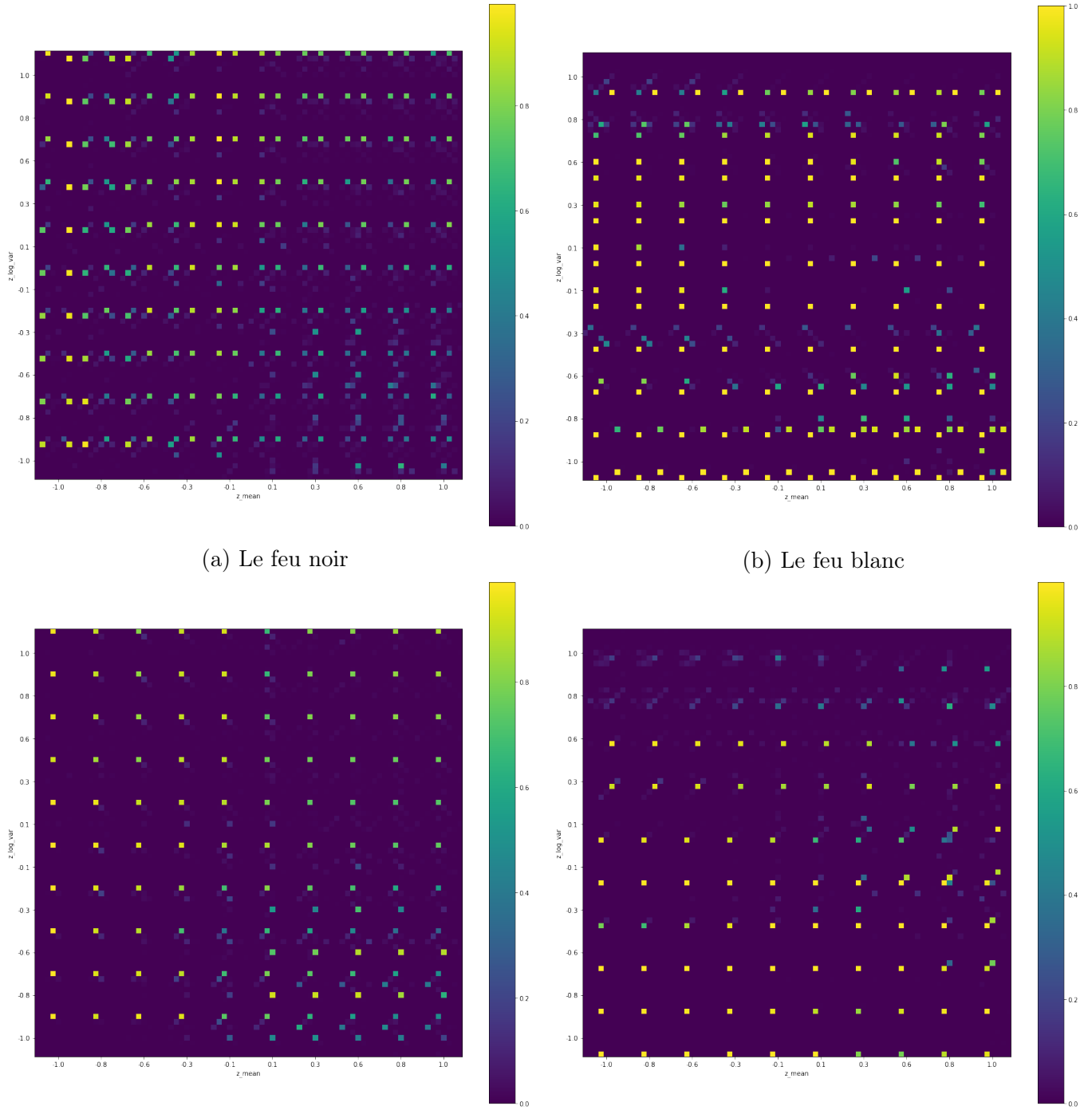


(c) Le cavalier noir



(d) Le cavalier blanc

FIGURE 7 – Les distributions de la probabilité de présence de chaque pièce pour 10x10 combinaisons de valeurs de μ et σ entre -1 et 1.



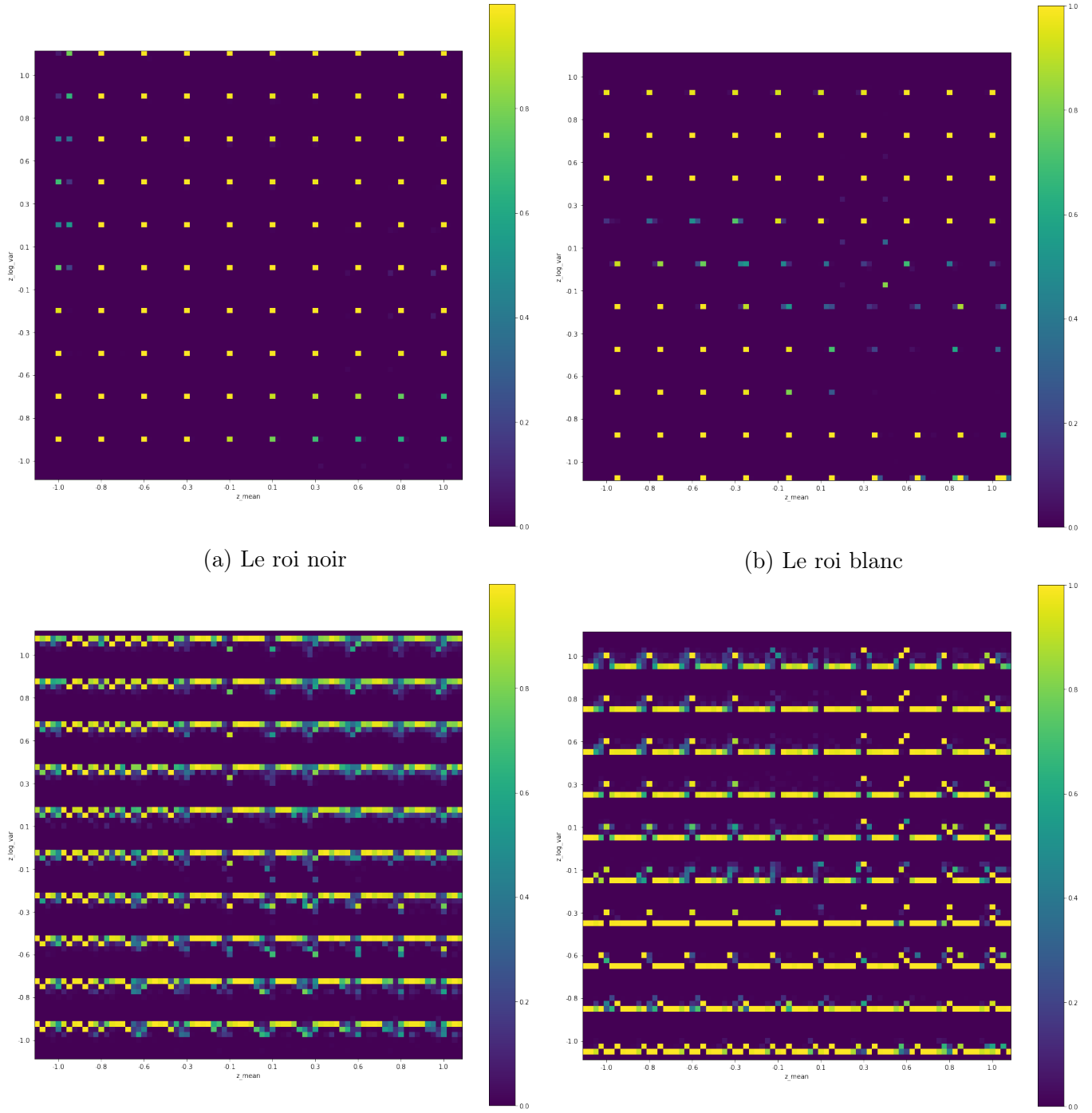
(a) Le feu noir

(b) Le feu blanc

(c) La reine noire

(d) La reine blanche

FIGURE 8 – Les distributions de la probabilité de présence de chaque pièce pour 10x10 combinaisons de valeurs de μ et σ entre -1 et 1.



(a) Le roi noir

(b) Le roi blanc

(c) Le pion noir

(d) Le pion blanc

FIGURE 9 – Les distributions de la probabilité de présence de chaque pièce pour 10x10 combinaisons de valeurs de μ et σ entre -1 et 1.

Sur toutes les pièces, le modèle arrive à représenter la position qu'une certaine pièce est susceptible de prendre. Cela est intéressant pour la génération de nouvelles configurations, il suffit d'échantillonner z_{mean} et d'utiliser le décodeur du $\beta-$VAE pour avoir la configuration de l'échiquier, cette configuration sera utilisée par la suite comme objectif pour l'agent qui devra l'atteindre.

5.2 Résultats Agent

On a entraîné notre agent sur 10 épisodes et pour chaque épisode on joue 10 parties. C'est à dire qu'on va générer 10 objectifs et pour chaque objectif généré, l'agent va apprendre à jouer contre

un ennemi *greedy* afin d'atteindre cet objectif. Ces valeurs sont issues de plusieurs considérations. Déjà, nous sommes limités par nos ressources de calcul et on ne peut donc pas générer un grand nombre d'objectifs puisque l'agent ne pourra pas jouer un grand nombre de parties. Et puis, on ne voudrait pas jouer un très grand nombre de parties pour le même objectif parce que cela risquerait de spécialiser l'agent dans cette tâche en particulier.

Comme on le remarque sur les valeurs de notre signal de récompense, la valeur de la récompense descend soudainement chaque 10 étapes ce qui correspond à la génération d'un nouvel objectif et donc au début l'agent a du mal à s'approcher de l'objectif durant les premières parties. Mais en utilisant une moyenne mobile sur ce signal de récompense on remarque une tendance de la croissance de la récompense. On en conclut que l'agent apprend à jouer de mieux en mieux en gardant l'objectif en vue.

Il est aussi à noter qu'on arrête les parties au bout de 10 coups pour deux raisons. La première c'est que nos données consistent en des configurations qui ont été atteintes en 5 tours mais on a voulu donner plus de tours à l'agent vu qu'il explore. La deuxième raison c'est qu'il est plus difficile d'influencer l'ennemi pendant les 5 premiers tours seulement et du coup l'agent a plus de chance d'imposer a l'ennemi d'exécuter un mouvement ou un autre (on rappelle l'ennemi a une politique déterministe) pour forcer à atteindre la configuration objectif.

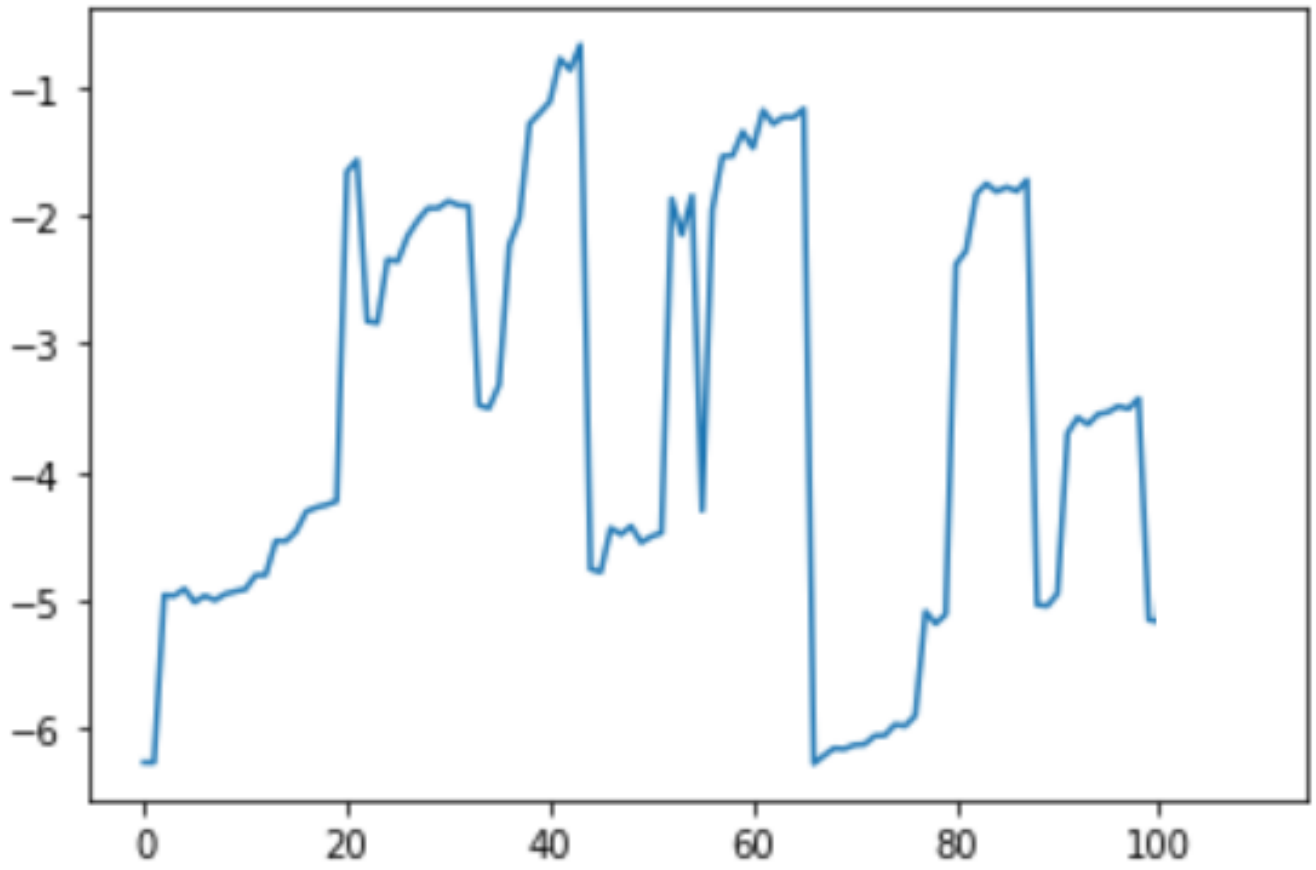


FIGURE 10 – Signal de récompense

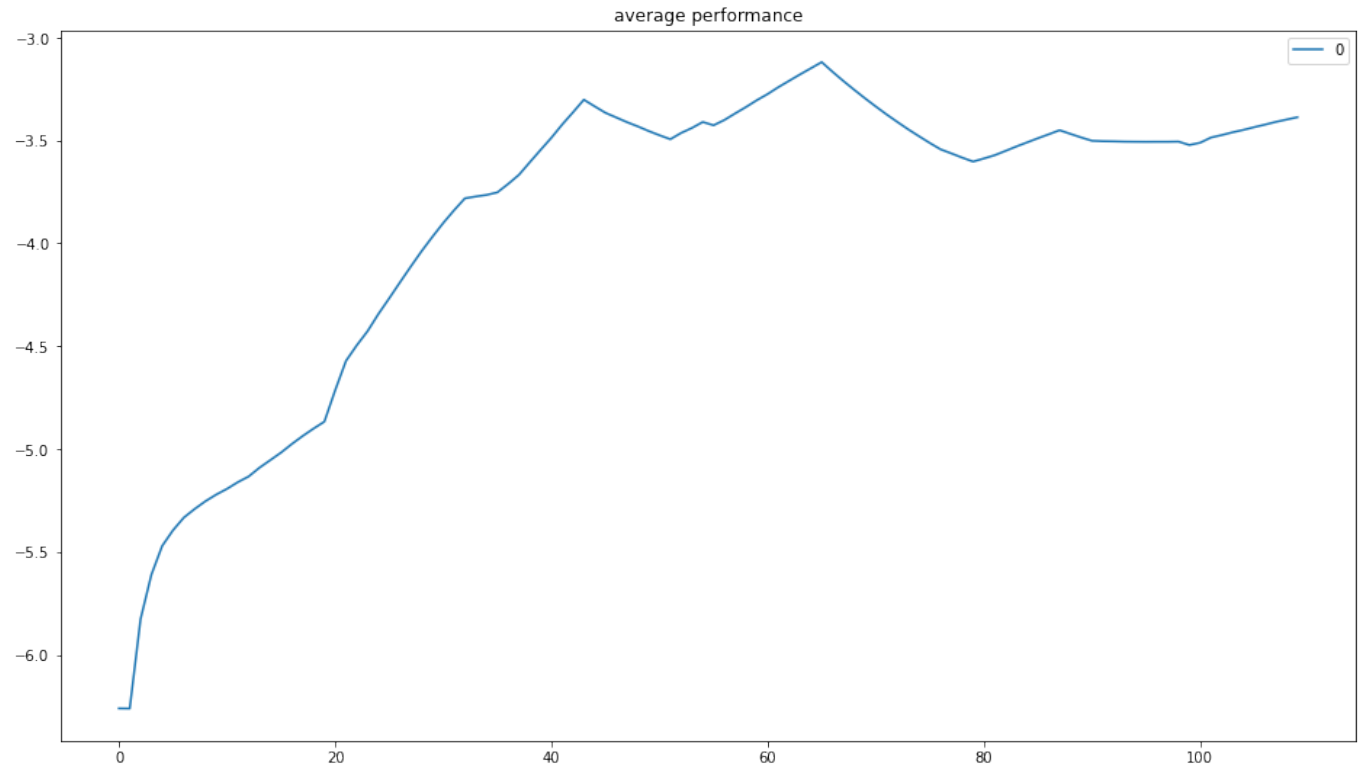
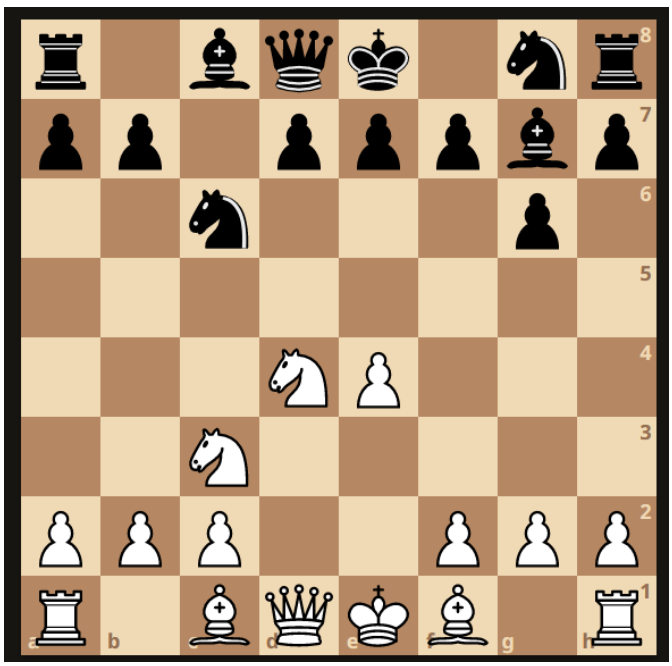


FIGURE 11 – Tendance du signal de récompense

Un exemple d'objectif généré et ce qui a été atteint à la fin des 10 tours pour une récompense moyenne sur les 10 tours de -1.47.



(a) Objectif à atteindre

(b) Configuration (finale) après 10 tours

6 Conclusion

Ce projet se décompose en deux parties : l'entraînement du β-VAE et l'entraînement de l'agent, et les deux parties sont essentielles à l'algorithme **RIG**. Un agent aussi bon qu'il soit en théorie n'apprendra pas grand chose si le β-VAE ne possède pas un espace latent assez bien structuré de sorte à bien encoder les échiquiers et renvoyer une petite distance pour deux échiquiers proches. De même un agent pas assez puissant (méthode pas adaptée au cas d'usage ou problème avec l'approximateur de fonction) ne pourra pas bénéficier du β-VAE.
En ce qui concerne notre β-VAE nous trouvons qu'il génère des objectifs plausibles et atteignables en début de partie, ainsi que la distance entre deux échiquiers proches est petite. Nous pensons qu'en augmentant la profondeur des arbres de simulation dans Monte-Carlo Tree Search, en augmentant le nombre d'objectifs générés vont avoir un impact plus immédiat qu'un fine-tuning de l'architecture de notre approximateur de fonction. Nos ressources de calcul étant limitées nous n'avons pu bien exploiter les architectures de deep learning qui s'offraient à nous et la méthode du Monte-Carlo Tree Search mais on voit tout de même qu'on a une tendance d'amélioration de l'agent.

7 Références

1 - *Ashvin Nair, Vitchyr Pong, Murtaza Dalal, Shikhar Bahl, Steven Lin, Sergey Levine : Visual Reinforcement Learning with Imagined Goals, 2018*.

2 - *Irina Higgins, Loic Matthey, Arka Pal, Christopher Burgess, Xavier Glorot, Matthew Botvinick, Shakir Mohamed, Alexander Lerchner : β-VAE : LEARNING BASIC VISUAL CONCEPTS WITH A CONSTRAINED VARIATIONAL FRAMEWORK, 2017*

3 - https ://database.lichess.org/