

Domaine abstrait pour représenter des nombres à virgule flottante sur des mini-formats.

Réseaux de neurones. Les réseaux de neurones sont des fonctions construites comme des étages de neurones. Dans la version la plus simple, chaque neurone x_i^j de la j -ième couche est connecté aux n neurones $x_{1 \leq k \leq n}^{j-1}$ de la couche précédente $j - 1$. La fonction calculée est relativement simple. Il s'agit d'une combinaison affine des entrées combinée avec une fonction d'activation σ . Ces fonctions d'activation introduisent des non-linéarités en encodant généralement des saturations : elles cherchent à contraindre la valeur produite dans un intervalle donné.

$$x_i^j = \sigma \left(c_0 + \sum_{1 \leq k \leq n} c_k x_k^{j-1} \right)$$

Les fonctions d'activation les plus courantes sont ReLU ($\max(0, x)$), sigmoïde ($\frac{1}{1+e^{-x}}$) ou SoftPlus ($\ln(1 + e^x)$).

Arithmétique en virgule flottante. Ces calculs manipulent des nombres décimaux encodés par des représentations à virgule flottante. Ces représentations de *float* utilisent une notation scientifique binaire. Ainsi, dans le cas des flottants simple précision sur 32 bits, il y a un bit de signe s , 8 bits d'exposant signé e et 23 bits de mantisse non signée m . On peut alors représenter les nombres sous la forme :

$$(-1)^s \cdot 1, m * 2^e$$

où m et e correspondent à la représentation entière des fragments de bits correspondant. On peut donc ainsi représenter ici des nombres de l'ordre de grandeur de $2^{27} = 2^{128} \equiv 3.4 \cdot 10^{38}$. L'arithmétique flottante est spécifique et nécessite des parties dédiées dans les processeurs. Plus particulièrement l'encodage permet de représenter des valeurs spéciales comme NaN (not-a-number), $+\infty$, $-\infty$. Ainsi il n'y a pas d'arithmétique cyclique comme pour les entiers et un *overflow* produira les valeurs $+\infty$ ou $-\infty$.

Mini-floats : bf16. Étonnamment, il n'est pas toujours souhaitable d'avoir la meilleure précision et l'on trouve aujourd'hui, pour les besoins spécifiques du machine learning, des utilisations de floats sur 16 bits et même parfois moins. Dans le cas de *bf16*, la valeur maximale est de 65504.

Objectif du BE : L'objectif de ce BE est d'implémenter un **domaine abstrait intervalle capable de manipuler les calculs en flottant** sur ces mini-formats pour cette application de réseaux de neurones. Plus particulièrement :

- **adapter le domaine des intervalles fourni pour traiter des cas particuliers** (NaN, $+\infty$, $-\infty$) ainsi que de la nature bornée des valeurs. Ainsi en *bf16*, $65504 + x$ avec $x > 0$ produira la valeur $+\infty$;
- proposer **une implémentation des fonctions exp et ln sur ces intervalles**. Vous pourrez vous inspirer des implémentations (imprécises) des fonctions cos et sin fournies.
- ATTENTION : on ne s'intéresse pas ici aux problématiques liées aux erreurs d'arrondi de la représentation flottante, mais uniquement aux nouvelles valeurs : NaN, $+\infty$, $-\infty$.

Préliminaires

Utilisation de SVN

Comme pour les TP, il est impératif de fonctionner avec le SVN dans le répertoire de votre groupe. Il vous faut également une machine capable de compiler du caml, avec ocamlbuild. Par exemple installée avec opam.

Les domaines abstraits suivants sont donnés dans le répertoire domains.

- `dummy.ml` : notre domaine basique classique
- `intervals3.ml` : le domaine des intervalles d'entier solution du TP2
- `intervals_double.ml` : une adaptation de ce domaine aux flottants. Notez que la division des réels est plus simple que la division Euclidienne.
- `BE.ml` : le domaine qu'il vous faudra modifier.

ATTENTION : il ne vous faut modifier que cet unique fichier `BE.ml`. Il est déjà déclaré comme domaine abstrait par défaut. Et le code fourni compile, même si la plupart des fonctions sont fausses.

L'arborescence du BE a été copiée dans le répertoire `be` de votre groupe. Par exemple, si vous êtes le groupe G03 :

```
> cd path_to_G03_folder/be/analyzer/src
> make
```

Comme précédemment :

- éditez le fichier du domaine. Ici `domains/BE.ml`
- Compilez et testez votre code régulièrement : `make` et `./tiny` Le code fournit compile, il ne tient qu'à vous qu'il conserve cette propriété.
- faites régulièrement des commit svn
`svn ci -m "j'adore l'interprétation abstraite."`

La note sera basée sur le code déposé, c'est à dire le dernier commit, avant 12h01. Ne changez pas d'autres fichiers que `BE.ml`, seul ce fichier sera évalué.

Information sur les modifications de l'analyseur

Il y a de légères différences entre l'analyseur utilisé en cours/TP et celui utilisé en BE. En particulier :

- l'analyseur distingue désormais les entiers des réels. Les variables doivent être déclarées au début du fichier : par exemple : `real x,y; int i,j,k;`
- la fonction `rand(x,y)` est remplacée par `rand_int(0,12)` et `rand_real(0.,12.)` en fonction du type désiré;
- attention les constantes réelles (flottantes) doivent avoir un point, comme en CaML. Ainsi on écrira `1.` pour la constante unité en réel;
- la signature des modules CaML des domaines non relationnels a été enrichie;
- le domaine doit désormais avoir un nom (`name`), et spécifier le type de variables qu'il représente (`base_type`);
- certaines fonctions comme `fprint_help` ou `parse_param` ont été rajoutées : ignorez-les;
- la fonction `sem_itv` a maintenant pour signature `float -> float -> t`;
- la fonction `sem_guard` qui forçait les valeurs à être dans l'intervalle $[1, +\infty[$ est remplacée par la fonction `sem_geq0` qui contraint les valeurs à être dans $[0., +\infty[$;
- enfin, une nouvelle fonction `sem_call: string -> t list -> t` permet de représenter toutes les fonctions numériques. L'exemple des fonctions `sin` et `cos` est donné dans `BE.ml`.

Domaine des mini-flottants

Compléter le fichier `BE.ml` pour obtenir une implémentation du domaine des intervalles bornés *bf16* avec valeurs NaN, $+\infty$, et $-\infty$. Le type est donné :

```
type t = bool * bool * bool * Itv.t
```

Un quadruplet (`isNan`, `isMInf`, `isPInf`, `itv`) représente l'appartenance des valeurs spéciales NaN, $+\infty$ et $-\infty$ à l'ensemble représenté. L'intervalle `itv`, du domaine `Intervals_double`, représente les valeurs normales dans le format *bf16*.

Il vous faut compléter ou corriger les fonctions du domaine :

- les constantes : `top` et `bottom` ;
- les fonctions du treillis : `order`, `join`, `meet`, `widening` ;
- l'injection d'intervalles de flottants `sem_itv`
- les fonctions arithmétiques `plus`, `moins`, `fois`, `diviser` ;
- la fonction de contrainte `sem_geq0`.

ATTENTION : les opérations arithmétiques devront gérer les valeurs spéciales ainsi que les calculs sur les valeurs normales qui peuvent éventuellement déborder.

Le module `Bounds` fourni permet d'accéder aux valeurs significatives de ce domaine *bf16*.

```
module Bounds =  
  struct  
    let name = "b16"  
    (* epsilon machine = smallest representable positive value *)  
    let eps = 0.000000059604645  
    (* largest representable values = interval [min, max] *)  
    let max = 65504.  
    let min = -. max  
  end
```

Il est bien sûr fortement conseillé de corriger tout warning qui apparaîtrait à la compilation et de tester le domaine implémenté, au moins sur les fichiers du dossier `examples`.

Précisions sur la notation : Chaque fonction à écrire ou à compléter est notée sur 2 points ou 3 points, suivant la difficulté. Une fonction incorrecte sera notée 0 et toute fonction correcte se verra attribuer une note entre 0 et 2 (ou 3) suivant sa précision¹.

Fonctions d'activation

La deuxième partie du BE consiste à étendre la fonction `sem_call` pour proposer un calcul correct et précis des fonctions `exp` et `ln`. Avant d'implémenter, on pourra d'abord réfléchir sur papier au domaine, aux valeurs spéciales, à la monotonie des fonctions, etc. On pourra ensuite utiliser les fonctions CaML : `exp: float -> float` et `log: float -> float (ln)`.

Enfin, il est très facile d'écrire un fichier d'exemple qui enchaîne des calculs basiques afin d'effectuer des tests unitaires des fonctions implémentées.

1. 0 pour la fonction constante \top et 2 ou 3 pour la fonction optimale par exemple.

Rendu

Tout est dit en début de document ! Attention à bien compiler et tester vos fonctions au cours du développement sans attendre le dernier moment.

Pensez aussi à faire des commits réguliers, de code qui compile. Par exemple, après le développement de chaque fonction.

Deadline : 12h.

Venez vérifier avec un enseignant que vos fichiers sont bien sur le SVN.