COMP309 in Week 09, 2024

optimisation
in machine learning

Marcus Frean marcus.frean@vuw.ac.nz

this week

may include traces of ~~nuts~~ math

1st lecture: optimisation

- loss functions
- optimization as it’s own thing
- gradients

2nd lecture: the use of gradients

- gradient descent as a “learning” algorithm (main flavours of)

Thurs tutorial:

- [ ] build from scratch in PyTorch, using autograd of the log loss

Learning and Optimisation

□ find the “best”, according to however you want to define “best”

- Optimisation is everywhere
- Learning and optimisation are closely related:
  - most learning can be called “optimisation in the light of a data set”
- Powerful optimisation is critical for the success of machine learning and AI

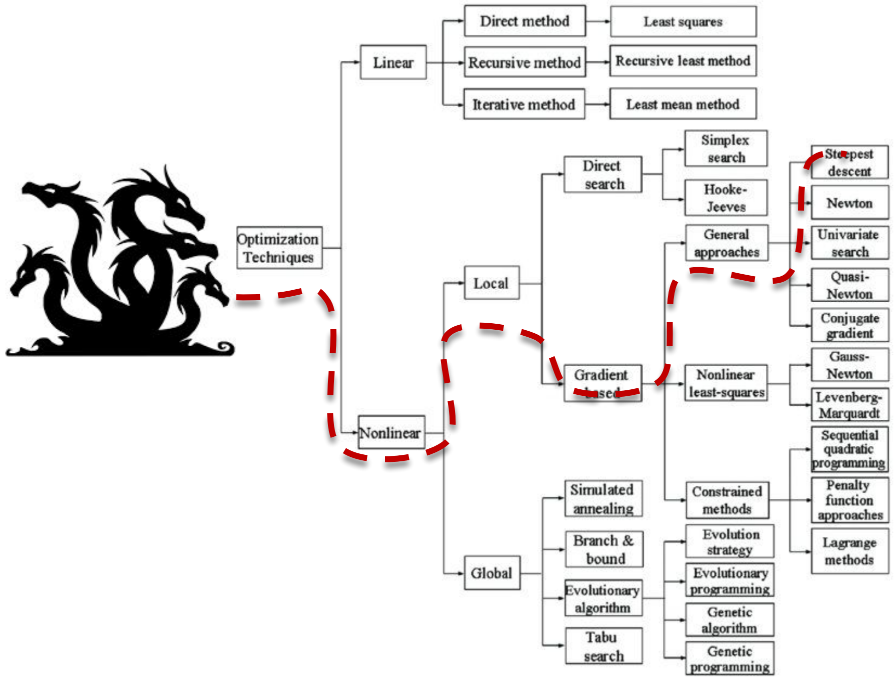
Optimisation problems

→ find the "best", according to however you want to define "best"

$$x^\star = \underset{x}{\operatorname{argmin}} f(x)$$

The ideal method would be global, continuous and unconstrained

Optimisation is a multi-headed beast. We will largely be focussing on just one of the heads: gradient descent



RECALL FROM WEEK 8:

Log loss is BOTH
- a performance metric
  (used on validation or test set); AND
- a loss function
  (used to optimise/learn a training set)

$$\text{log loss} = -\sum_{\text{item } i} \sum_{\text{class } j} t_{ij} \log y_{ij}$$

- as a performance metric:

Accuracy is not always a good indicator, partly due to its “crisp” yes-or-no nature.

Log loss takes into account the uncertainty of your prediction based on how much it varies from the actual label. This gives us a more nuanced view into the performance of our model. Often used as an evaluation metric in Kaggle competitions.

- as a loss function:

virtually always the basis of learning in neural nets (where we can use its gradient)

optimize what? – often it's a log likelihood

this just makes the sum into an *average*

one-hot

- "log loss" for classification tasks:

$$\text{Log Loss} = -\frac{1}{N}\sum_{\text{item } n}^{N} \sum_{\text{class } j} t_{nj} \log y_{nj}$$

- "MSE" for regression tasks:

$$\text{MSE} = \frac{1}{N}\sum_{\text{item } n}^{N} (t_n - y_n)^2$$

Sketch of the deeper story:
Both are actually the logarithm of a *likelihood*

Many classifiers can output a probability distribution over classes: Categorical
Think of regression output as mean of distribution over Reals: Gaussian

$$e^{-\frac{1}{2}(t-y)^2}$$

Likelihood:
probability it would get the answer CORRECT, for ALL the data at hand.
That probability is a PRODUCT over the data set (get 1st right AND 2nd AND 3rd...)
So the log likelihood is a SUM over the data set.

loss function: is it differentiable?

the most useful loss functions are differentiable: we can take “gradients”, and this can help us as we search for an optimum. Example:

$$L(x, w) = \frac{1}{2}(x - w)^2$$

we say this loss is “quadratic in w”

$$\frac{\partial L}{\partial w} = -(x - w)$$

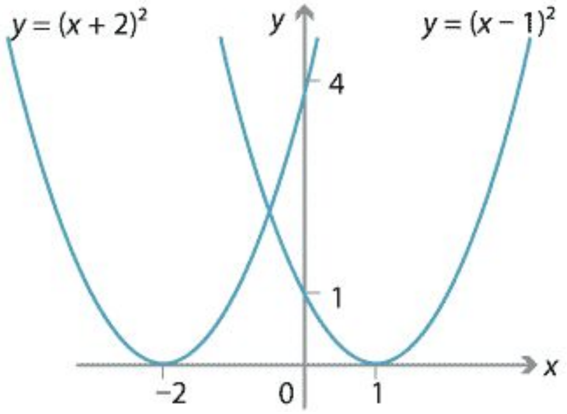
gradient of the loss *with respect to* the parameter of interest

$$\frac{\partial L}{\partial w} = 0$$

in this case we can simply set it to zero!

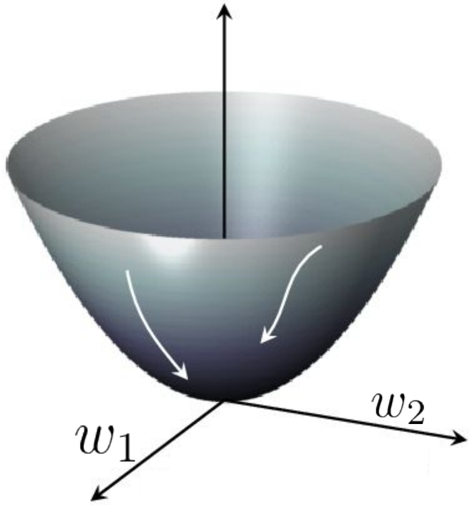
…and solve, for the optimal parameter value

$$w = x$$



generalising this idea

The MSE is quadratic in y, and if y is linear in w, then MSE is quadratic in w too, so we should be able to do the same trick/shortcut…



$$\text{MSE} = \frac{1}{N}\sum_{\text{item } n}^{N} (t_n - y_n)^2$$

quadratic in y

$$\text{Suppose } y \text{ linear: } y_n = \sum_i x_{n,i} w_i + b$$
$$= \mathbf{x}_n \cdot \mathbf{w} + b$$

y linear in w (note the 2 ways of writing this)

then we can *solve* exactly (!!)

$$\mathbf{w}^\star = (\mathbf{X}^T\mathbf{X})^{-1}\mathbf{X}^T\mathbf{y}$$

don’t worry about the mathematical details behind this

The point is, we were able to simply solve ***analytically***, for the optimal parameter value

other analytic cases

We call an optimization problem **convex** if it has a single optimimum.
Some convex problems have analytic solutions, as you've just seen.

When available, analytic answers beat iterative algorithms, in two ways:

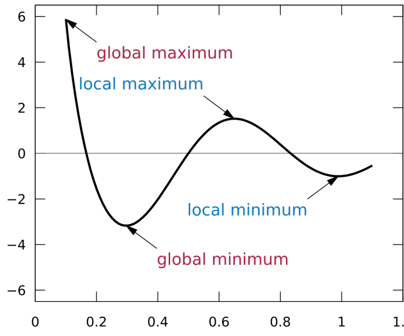
- ❑ might get performance guarantees
- ❑ faster - mathematical optimization is NP-hard in general, but many specific classes of convex optimization problems admit polynomial-time algorithms

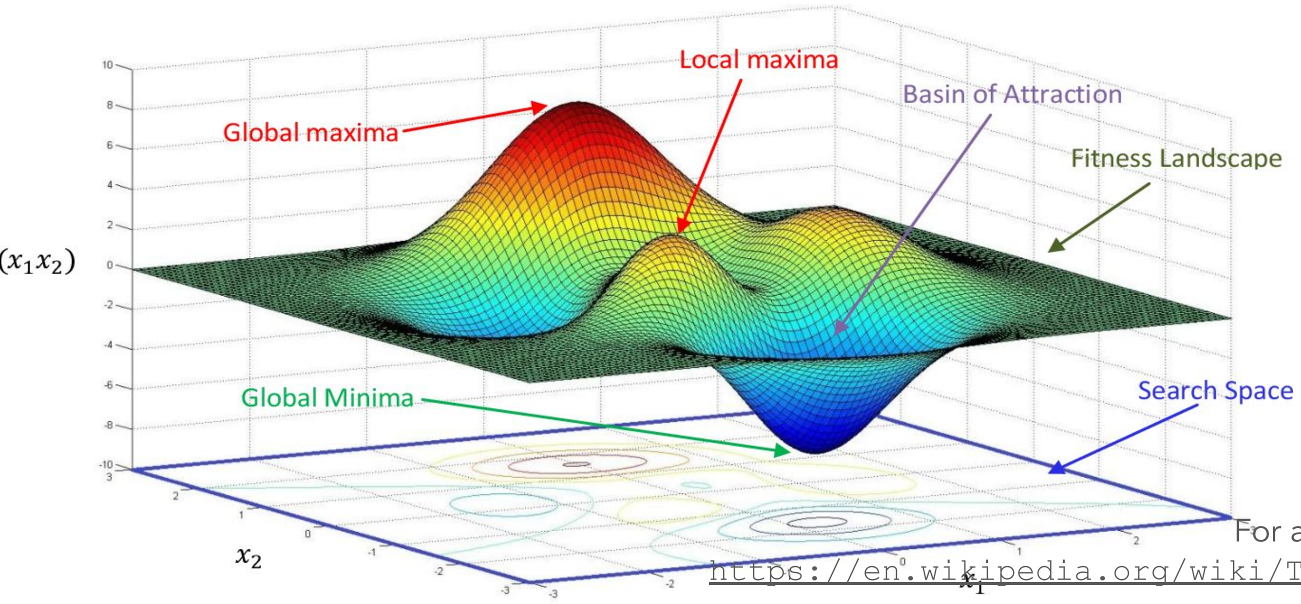
- eg: the Gradient Boosting algorithm. *Boosting algorithms perform gradient descent in a function space using a **convex** loss function*

is this Game Over? no: this is just how you train one "weak" learner, but you still need lots of such learners to make a "strong" learner from the ensemble

local, global,… meh

- “convex” means there are no “local optima” beside the “global” one
- beware of snake-oil salesmen claiming their algorithm is magically “global” – it usually just means it has enough randomness in it to *eventually stumble* upon the best optimum: a very weak promise indeed!





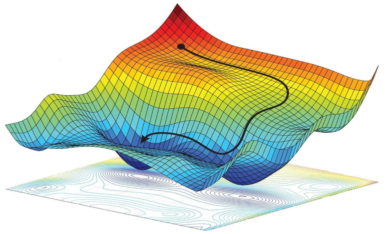
For an idea of other examples, take a look at https://en.wikipedia.org/wiki/Test_functions_for_optimization

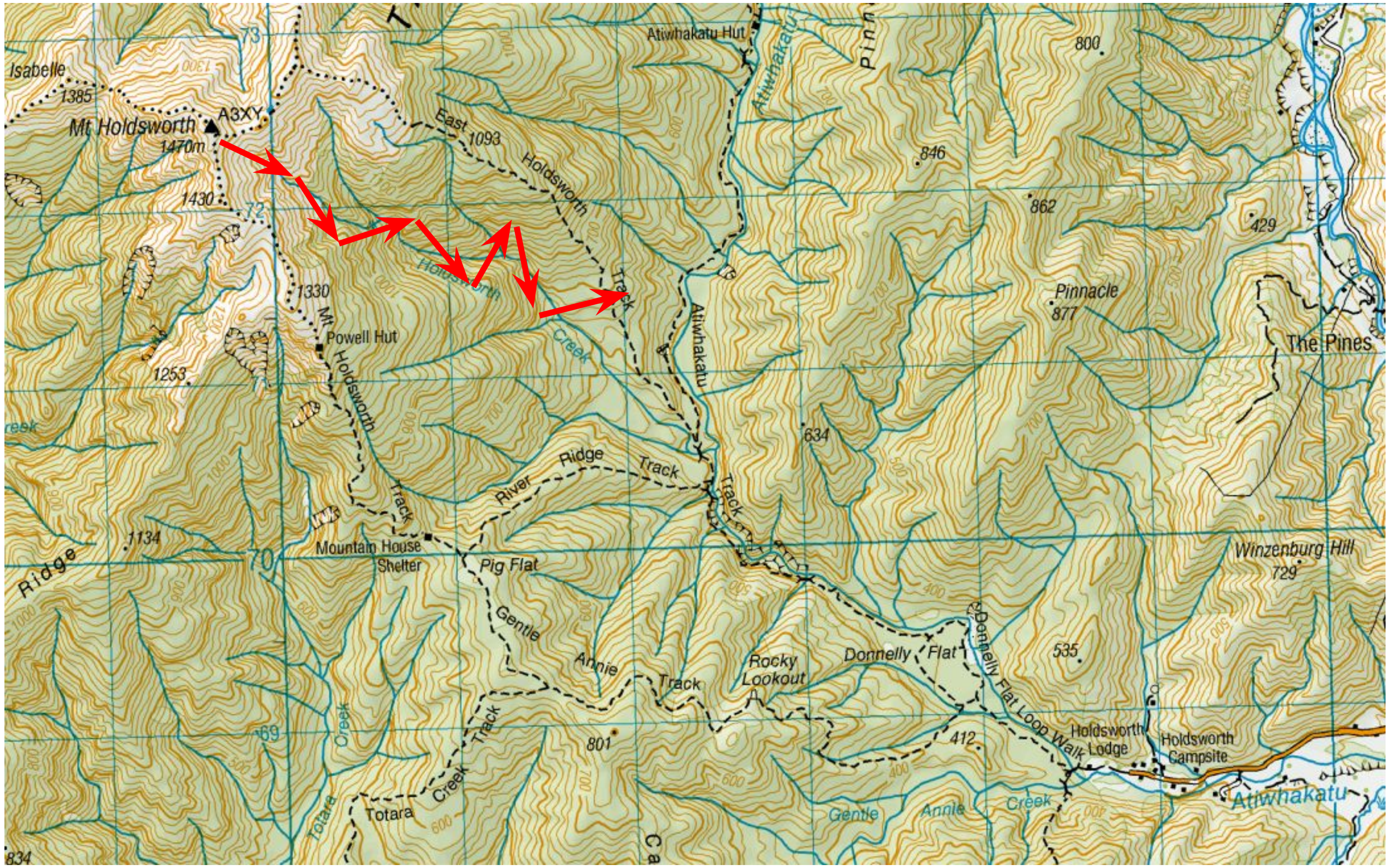
some general comments

optimisation is not just one story

- is the space of solutions discrete or continuous?
- how many options are there? (ie. is the problem “cursed” by dimensionality…)
    - is it continuous but with high dimensionality? (in machine learning we usually want to optimize many variables at once
    - is it a discrete but combinatorial problem? (e.g. the TSP)
- stochasticity (ie. can you evaluate a solution in “one go”?)
- does the problem exhibit optimal substructure?
  (e.g. stochastic planning □ “dynamic programming”)
- do we even know the loss function? (e.g. in reinforcement learning, we don’t, so we’re “learning” that at the same time)

gradient descent





notation for gradients

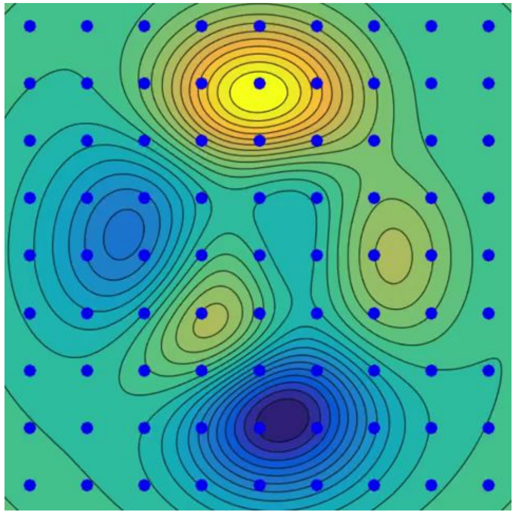
If we've got some loss *L* that depends on several parameters θ, then we have a "space" of options

Generalising the idea of "slope", we can denote the gradient of *L* with respect to each parameter in turn by

$$\frac{\partial L}{\partial \theta_i}$$

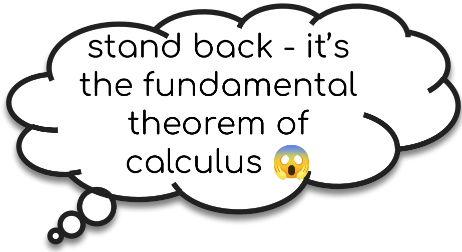
and it's useful to be able to pack them all up into a **vector**

$$\nabla_\theta L = (\frac{\partial L}{\partial \theta_1}, \frac{\partial L}{\partial \theta_1}, \dots \frac{\partial L}{\partial \theta_d})$$

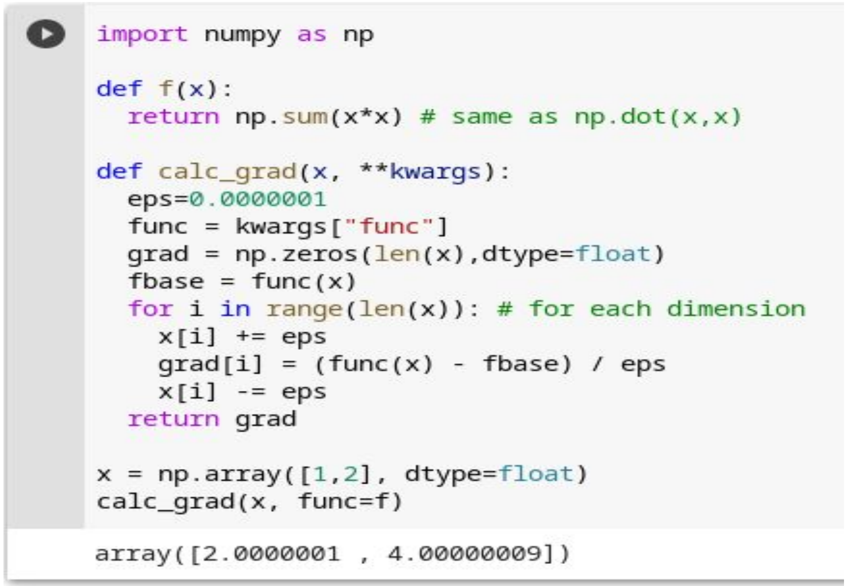


https://en.wikipedia.org/wiki/Gradient_descent

find gradients “empirically”?



$$\frac{\partial f}{\partial x} = \lim_{\epsilon \to 0} \frac{f(x+\epsilon) - f(x)}{\epsilon}$$



```python
import numpy as np

def f(x):
  return np.sum(x*x) # same as np.dot(x,x)

def calc_grad(x, **kwargs):
  eps=0.0000001
  func = kwargs["func"]
  grad = np.zeros(len(x),dtype=float)
  fbase = func(x)
  for i in range(len(x)): # for each dimension
    x[i] += eps
    grad[i] = (func(x) - fbase) / eps
    x[i] -= eps
  return grad

x = np.array([1,2], dtype=float)
calc_grad(x, func=f)
```

```
array([2.0000001 , 4.00000009])
```

why not just do that then?

Scales as O(#dimensions in x)

So it’s a non-starter

(however: often useful as a CHECK ← ← that the calculus is being done right)

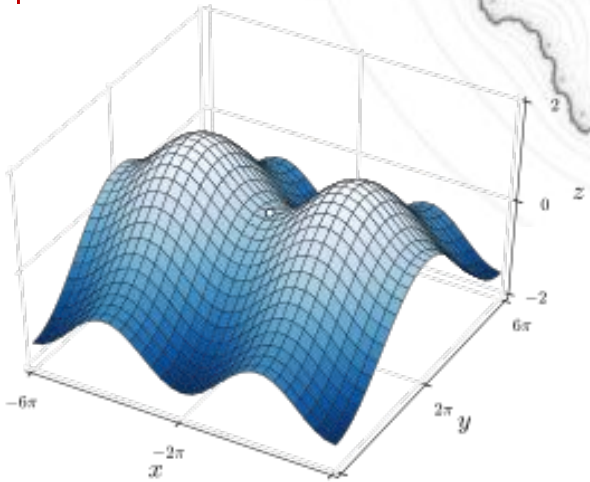
gradients

so: can you measure gradients cheaply?

- While using gradients seems obvious and can improve the speed of convergence, such evaluations increase the computational complexity of each iteration. How cheap are the gradients?

  aside: a major appeal of neural networks is that their gradients are indeed “cheap”

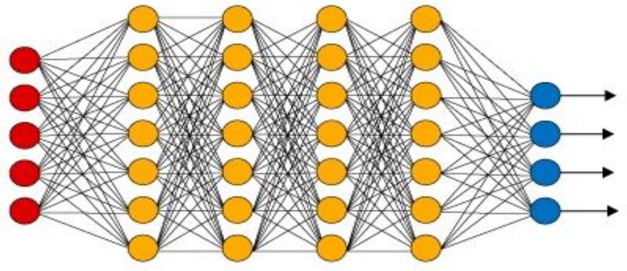
- another aside: *second* derivatives (gradients of gradients: “Hessians”) might be handy too…
  e.g. consider a saddle-point:



comment on nomenclature

E.g. the various “optimisations” involved in training a deep Neural Network, say:

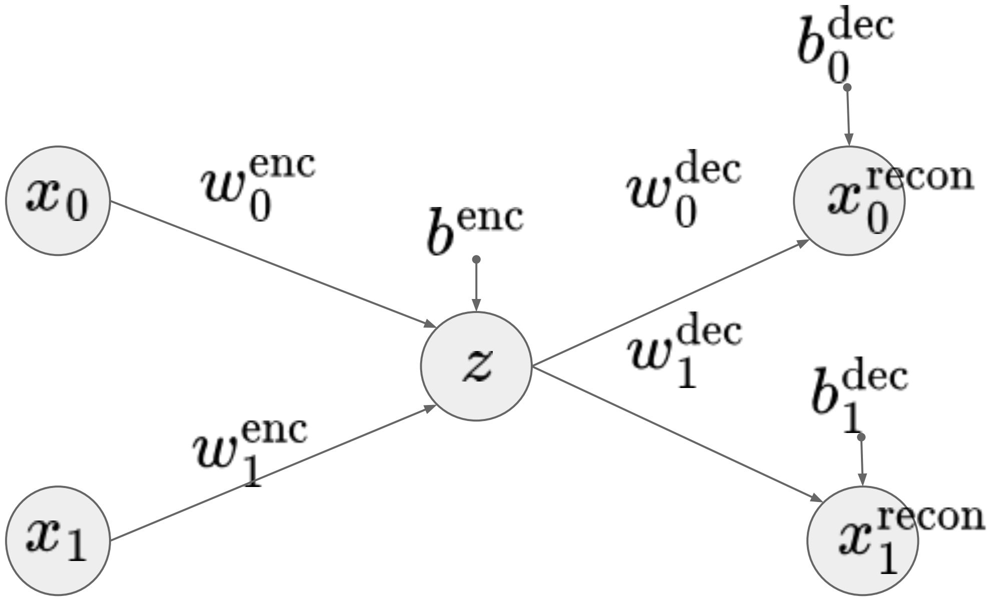
- find the optimal values of millions of tunable parameters
- choose the best architecture
- decide on a learning rate



A. For **parameters of the mapping** itself, we will want optimisation that is global, continuous, and unconstrained. Ideally we will have access not just to f(x) but to its gradients as well

B. For **hyperparameters**, the options are usually discrete and not too numerous. And we *won’t* usually have gradients for these.

C. There are also **parameters of the optimiser** (e.g. learning rate) – these are *often lumped under the “hyperparameters”*, but they’re different


x_0
x_1
w_0^{enc}
w_1^{enc}
b^{enc}
z
w_0^{dec}
w_1^{dec}
b_0^{dec}
b_1^{dec}
x_0^{recon}
x_1^{recon}