GuessTuples Project

Andrew J. Wren

24 May 2021

Abstract

Notes on `GuessTuples` project aka `NLearn`

1 Configuring the nets

1.1 Alice

One per bit. The input array to guess is $\boldsymbol{x} = (x_j)_{j=0,\ldots,N_{\text{elements}}}$. There should be N_{code} outputs taking values $\boldsymbol{y} = (y_j)_{j=0,\ldots,N_{\text{code}}}$.

Normalise all the rewards so that for each bit j, $r_{j✓} + (N_{\text{code}} - 1)\, r_{j✗} = 0$. In other words

$$r_{jk} \leftarrow r_{jk} - \frac{r_{j✓} + (N_{\text{code}} - 1)\, r_{j✗}}{N_{\text{code}}}. \tag{1}$$

The Q estimate is then taken to be

$$Q(\boldsymbol{x}) = \sum_j b_j y_j \equiv \sum_j |y_j|\,, \tag{2}$$

where

$$b_j = \operatorname{sgn}(y_j) \tag{3}$$

is the prediction for the machine value of the jth bit. The loss function is

$$L = |Q(\boldsymbol{x}) - r|^2\,. \tag{4}$$

Alternative approaches include:

1. Two outputs for each bit showing the reward for each of 0 and 1. *May reflect negative rewards better?*

2. Combine the rewards from the bits (with either one or two outputs per bit) by something other than addition - e.g. multiplication or via an NN. *The NN option seems quite interesting. Interesting to use* `pytorch`*'s gradients for that.*

3. **One per code.** One output for each possible code. *Might work but $2^{N_{code}}$ is quite large... not impossibly so if $N_{code} = 8$.*

4. Inspired by Ref. [1], feed $\boldsymbol{x}$ into Alice's 'first' net, to get output $\boldsymbol{y}$, and all possible codes $\boldsymbol{c}$ into her 'second' net, both net's having the same target dimensionality (a hyperparameter). Then the code to use is the one $\boldsymbol{c}(\boldsymbol{x})$ closest to the output of the first net, with the Q being given by the inner product $Q = \langle \boldsymbol{y}, \boldsymbol{c}(\boldsymbol{x}) \rangle$. Ref. [2] might provide an alternative, actor–critic, approach on a similar theme. The main case above is, in effect, an embedding of $\boldsymbol{x}$ into the target space (of dimensionality N_{code}) which then compares with the natural embedding of $\boldsymbol{c}$ by, in effect, the inner product.

5. Ref. [3] suggest sequentialising, which points to a variant of our main approach which does each bit in succession and feeding those results into successive Alice–nets so the Q-estimate for later bits takes account of earlier bits / estimates, with the N_{code}th estimate providing a final code $\boldsymbol{c}$ and Q-estimate for that code.

6. Move away from typical Q-learning. Instead Alice's output is the code $\boldsymbol{c}$ and then when Bob makes his choice $\boldsymbol{x}_{\text{pred}}$ (see below) run that choice through a copy of Alice, to get $\boldsymbol{c}_{\text{Bob}}$ and then the loss function is
$$L = -\, r(\boldsymbol{c}, \boldsymbol{c}_{\text{Bob}}). \tag{5}$$

1.2 Bob

One per bit aka **Simple.** Bob receives a matrix, $\mathbf{X} = (\boldsymbol{X}_i) = (X_{ij})$ for $0 \le i < N_{\text{select}},\ 0 \le j < N_{\text{elements}}$, and a code $\boldsymbol{c} = (c_k)_{k=0,\ldots,N_{\text{code}}}$. Why not makes his outputs be Q-estimates $\boldsymbol{z} = (z_i)_{i=0,\ldots,N_{\text{select}}}$. Bob's prediction is then $\boldsymbol{x}_{\text{pred}} = \boldsymbol{X}_{i_{\text{pred}}}$ where
$$i_{\text{pred}} = \text{argmax}_i(z_i). \tag{6}$$
The loss function is
$$L = \left| z_{i_{\text{pred}}} - r \right|^2 . \tag{7}$$
How do we enforce covariance with respect to the order of $(\boldsymbol{X}_i)$?

1. Covariance will occur naturally and quickly without any specific intervention. *To be determined.*

2. Covariance can be enforced through choosing a set $\{\sigma\} \subseteq S_{n_{\text{code}}}$, which could be generated element–by–element by composing randomly–selected basis transpositions $(j\ j+1)$, and then adding to the loss a term
$$\mu \sum_{\sigma} \left| z - \sigma^{-1} \left[z(\sigma[\mathbf{X}]) \right] \right|^2 \tag{8}$$
for some fixed hyperparameter $\mu > 0$. Note this the term is still run backward through the original $\boldsymbol{x} \mapsto z$ net configuration only. *How effective would that be? How big does $\{\sigma\}$ have to be? And how much time would the permutation and the additions forward passes cost?*

3. Enforce covariance via direct identification of weights in Bob's net. *How?*

4. Something related to set transformers. *?*

5. Adopt a different basic set–up where each (X_i) is fed through the net separately, alongside the code $\boldsymbol{c}$, resulting in a Q-estimate z_i. Then find the loss function as in Eq. 7. *Seems the most straightforward?*

None of these quite amount to Bob seeks to reproduce the Alice's code vocabulary. However Bob could additionally set up a net in the same basic configuration as Alice's (he doesn't know the weights of course) and train *that* net jointly with his main net.

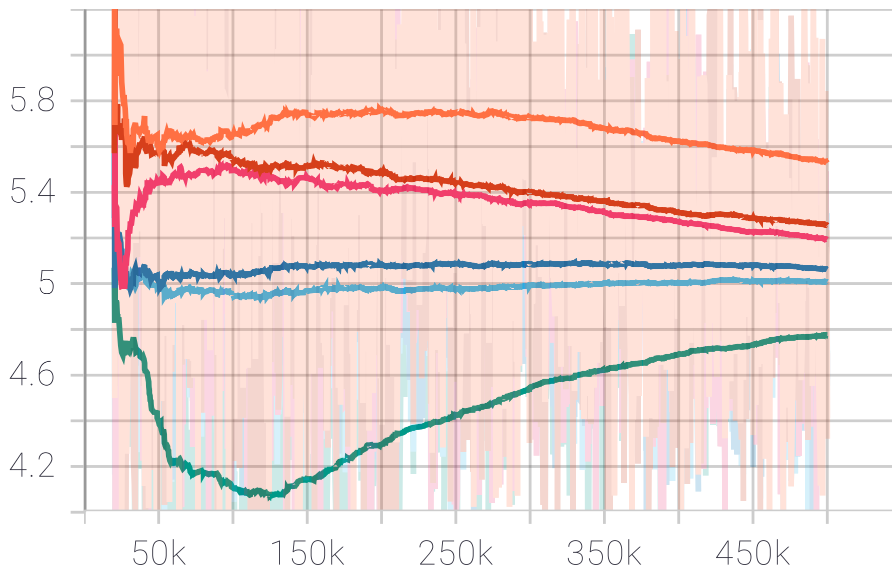

Figure 1: The best results — from `/runs/Apr27_23-01-58_andrew-XPS-15-9570`. The lines show the square root of the mean square losses with (a) `lr=0.3` Alice (orange), Bob (dark blue); (b) `lr=0.1` Alice (brick red), Bob (cyan); (c) `lr=0.01` Alice (pink), Bob (green). The plot is from TensorFlow and uses smoothing of 0.999. Note rewards from random plays are counted.

2 Results

2.1 Original strategies

Figure 1 is representative of the better results for the original strategies, **one per bit** — in other words, not very good.[1] Increasing from `h.GAMESIZE = 1` to `h.GAMESIZE = 32` gives no better results.

3 Revised approach — `NLearn`

Key runs:

1. `21-05-01_12:05:16` is the strategy that works

```
    'ALICE_STRATEGY': 'from_decisions',
    'BOB_STRATEGY': 'circular_vocab'
```

   up to a point when it levels off. Gets to `reward` = 0.6.

2. `21-05-01_20:04:35` other `lr` choices but same result — see Figure 2

3. `21-05-02_17:29:40` stops Alice training at some point. Alice `lr` = 0.1 and Bob `lr` = 0.01 gets to 0.8 — see Figure 3. The `config` includes

```
    hyperparameters = {
```

[1]The plot is taken from TensorBoard which gives an `.svg` file, then converted to `.pdf` by `rsvg-convert -f pdf -o <fig-file-name>.pdf "Sqrt losses.svg"`.

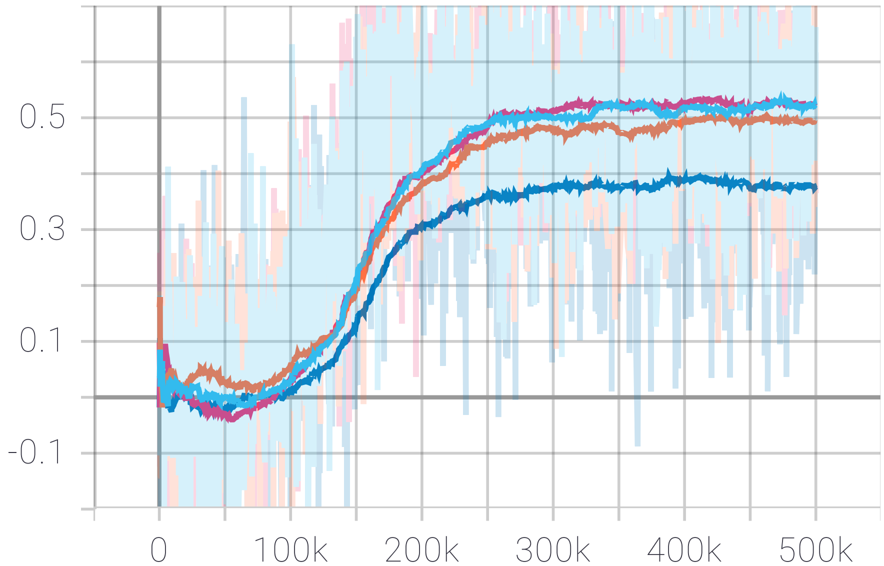

Figure 2: Mean Rewards per game for `21-05-01_20:04:35`. By colour, (Alice `lr`, Bob `lr`) are: cyan $(0.1, 0.1)$, orange $(0.1, 0.01)$, pink $(0.01, 0.1)$, and blue $(0.01, 0.01)$. Note rewards from random plays are counted.

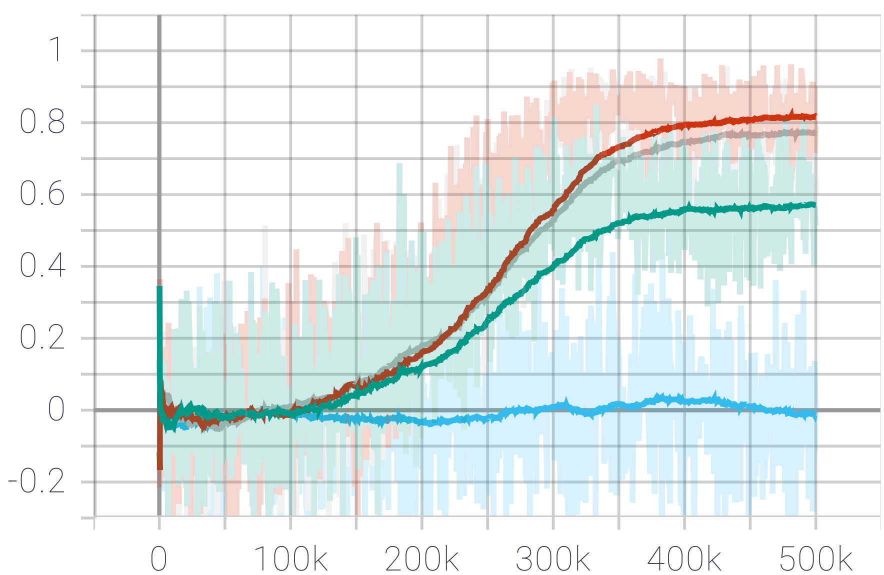

Figure 3: Mean Rewards per game for `21-05-02_17:29:40`. By colour, (Alice `lr`, Bob `lr`) are: green $(0.1, 0.1)$, orange $(0.1, 0.01)$, grey $(0.01, 0.1)$, and cyan $(0.01, 0.01)$. Note rewards from random plays are counted.

```
        'N_ITERATIONS': 500000,
        'RANDOM_SEED': 42,
        'TORCH_RANDOM_SEED': 4242,
        'ALICE_LAYERS': 3,
        'ALICE_WIDTH': 50,
        'BOB_LAYERS': 3,
        'BOB_WIDTH': 50,
        'BATCHSIZE': 32,
        'GAMESIZE': 32,
        'BUFFER_CAPACITY': 640000,
        'START_TRAINING': 20000,
        'N_SELECT': 5,
        'EPSILON_ONE_END': 40000,
        'EPSILON_MIN': 0.01,
        'EPSILON_MIN_POINT': 300000,
        'ALICE_STRATEGY': 'from_decisions',
        'BOB_STRATEGY': 'circular_vocab',
        'ALICE_OPTIMIZER': ('SGD', '{"lr": 0.1}'),
        'BOB_OPTIMIZER': ('SGD', '{"lr": 0.01}'),
        'ALICE_LOSS_FUNCTION': ('MSE', {}),
        'BOB_LOSS_FUNCTION': 'Same',
        'ALICE_LAST_TRAINING': 200000
```

Alice here, `21-05-02_17:29:40 hp_run=2` generates codes as follows:

```
11101100 [0, 1, 2, 12, 13, 14, 15]
11101110 [3]
10101110 [4, 6]
10100110 [5, 7]
10100100 [8]
11100100 [9, 10, 11]
```

Surprisingly only six distinct codes used! At least the first and last have sequential runs of numbers.

4. If increase `N_SELECT` to 16 (all the numbers shown to Bob), then, in run `21-05-03_10:53:10`, gets to `reward` = 0.8, as good as for `N_SELECT` = 5. In fact very slightly better (mean at 25.0° rather than 32.9°) Alice's code book is still very small:

```
21-05-03_10:53:10BST_NLearn_model_1_Alice_iter500000

01111010 [0, 15]
01111100 [1, 2, 3, 4, 5, 14]
01011100 [6, 7]
01011110 [8, 9, 12, 13]
01111110 [10, 11]
```

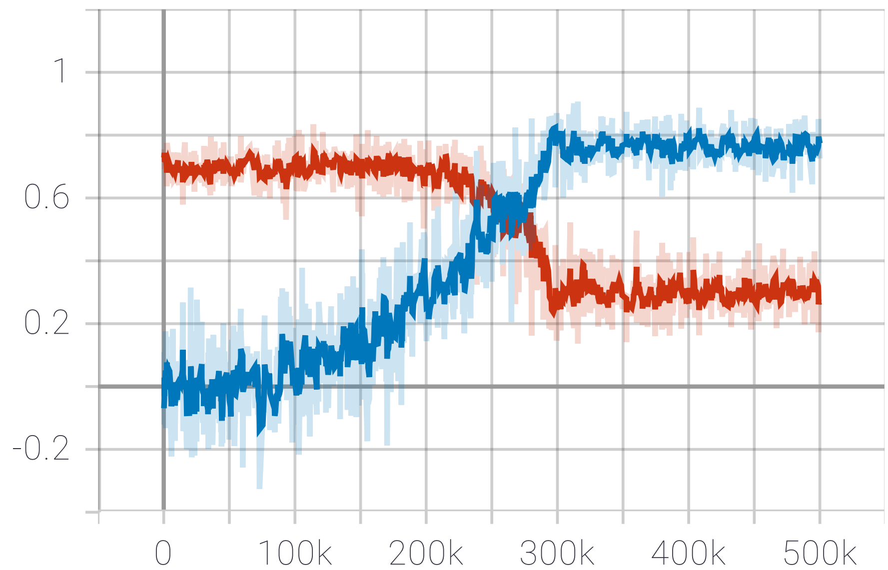

Figure 4: With N_SELECT = 16, at 21-05-03_10:53:10.

4 From now on exclude random plays from mean reward

The exclusion is if either Alice or Bob or both is random.

4.1 Loss includes element to push bits towards -1 or 1, and simple 'proximity bonus'

This gets pretty good results — see Figure 5 which also (orange, pink, blue) lines adds a 'proximity bonus' that — at least for these seeds — speeds up training but does not improve the outcome.

4.2 Loss includes element to push bits towards -1 or 1, and simple 'proximity bonus'

At 21-05-05_11:27:12, changing Section 4.1 by

```
'N_ITERATIONS': 15 * (10 ** 4),
'ALICE_PROXIMITY_BONUS': 30000,
'ALICE_PROXIMITY_SLOPE_LENGTH': 10 ** 4
```

get the excellent result shown in Figure 6, having a final smoothed value of 0.94. The final coding and decoding books are

```
00100111 [0, 1, 2, 3]
10100111 [4]
10110111 [5]
10110011 [6, 7]
10111011 [8]
10101010 [9, 10]
10101101 [11]
```

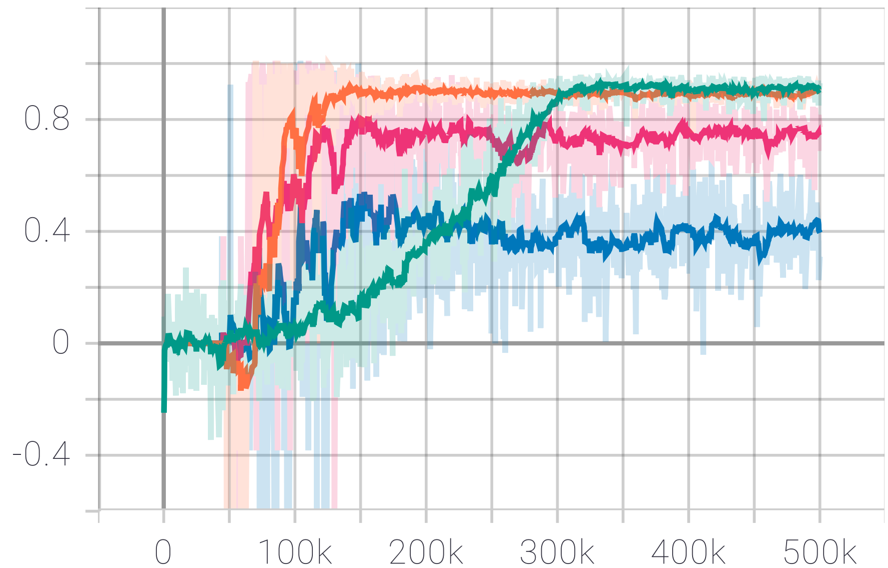

Figure 5: The green line shows the best run from `21-05-03_20:36:57`, which introduced `MSEBits` and had Alice stopping training at iteration 300 000. The remaining lines are from `21-05-04_20:10:38` and do not stop Alice training. They add the simple ‘proximity bonus’ of 1 when codes or numbers are equal from iteration 100 000 (orange), 200 000 (pink) and 300 000 (blue), The plot has smoothing set to 0.9.

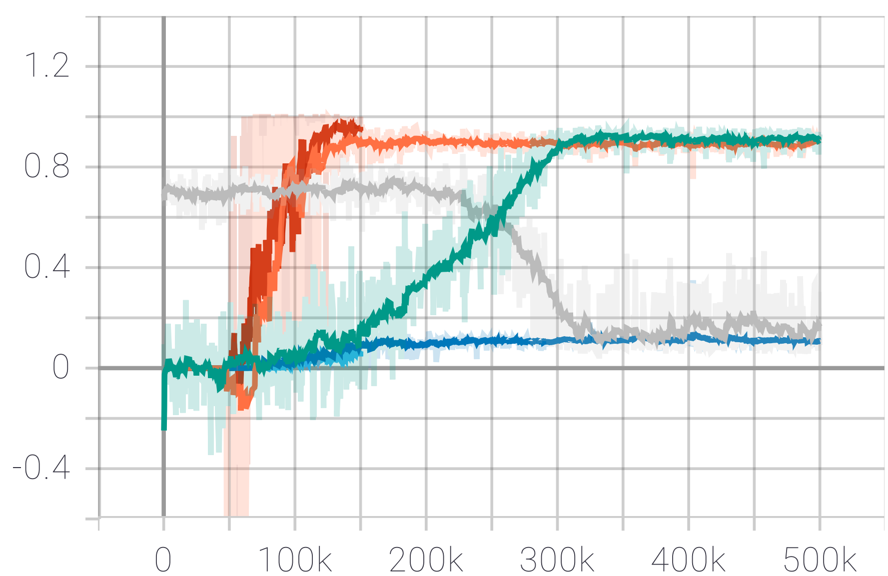

Figure 6: The red line shows the mean reward of `21-05-05_11:27:12`, while the just visible cyan line is its standard deviation. The orange and green lines are as in Figure 5, with the blue and grey lines being their respective standard deviations. The plot has smoothing set to 0.9.

```
    10100101 [12]
    00100101 [13, 14, 15]


    00100111 2
    10100111 4
    10110111 5
    10110011 6
    10111011 8
    10101010 10
    10101101 11
    10100101 12
    00100101 14
```

with Alice using nine codes.

However, another run, `21-05-05_13:13:06`, with the same parameters, except for the three seeds, shows the high random dependence getting a small code book:

```
    10010100 [0, 1, 2, 3, 12, 13, 14, 15]
    10010000 [4, 5, 6, 7]
    00111000 [8, 9, 10]
    10110100 [11]


    10010100 0
    10010000 5
    00111000 9
    10110100 11
```

Figure 7 compares with previous results. Perhaps suggests introducing some noise?

4.3 Noise

From `21-05-05_21:56:16`, noise doesn't seem to help on this individual run — see Figure 8. However, does it make the model more robust to changes in random seeds?

4.4 In Alice training, make both sides of the loss function have `grad`

As in `21-05-06_09:44:04`, this doesn't work — reward oscillates around zero. Is also slower.

4.5 Phasing in proximity bonus, double deep learning for Alice

In `21-05-06_20:41:41` find that phasing in (over 10 000 iterations) helps — getting to 0.96 — but adding double deep learning for Alice may not. See Figure 9.

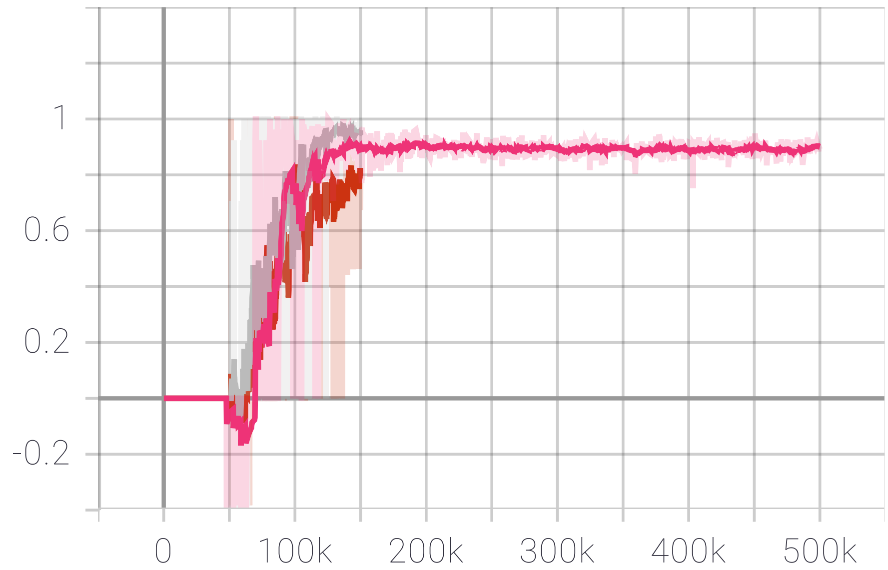

Figure 7: From `21-05-05_13:13:06` we have the red line. The grey line which is the former run shown in red in Figure 6 and the that shown in orange in Figure 6. The plot has smoothing set to 0.9.

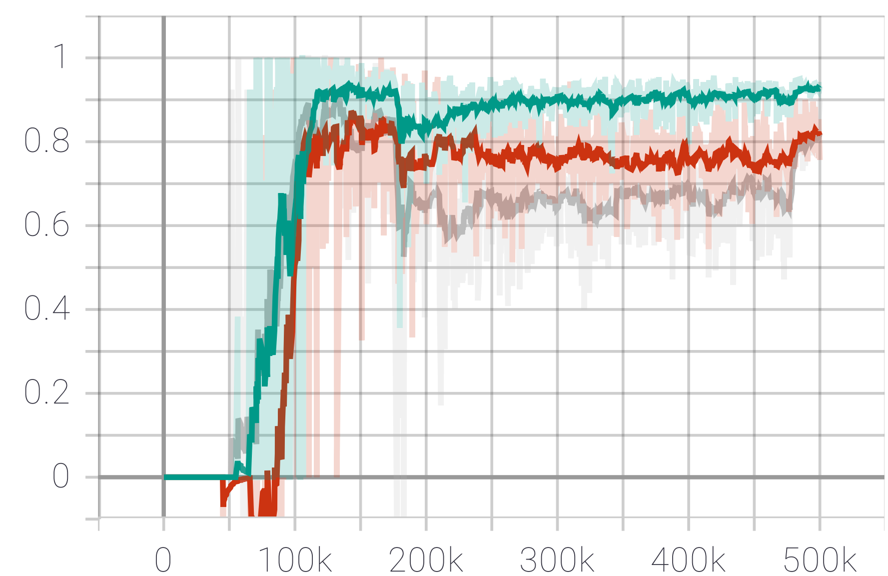

Figure 8: From `21-05-05_21:56:16`, with noise of 0.01 (green), 0.03 (orange), 0.1 (grey), starting at iteration 175 000 (earlier starts were poorer). The noise is cut off before the end to allow comparison. Plot smoothing is at 0.9, and lesser smoothing doesn't show more post–noise recovery.

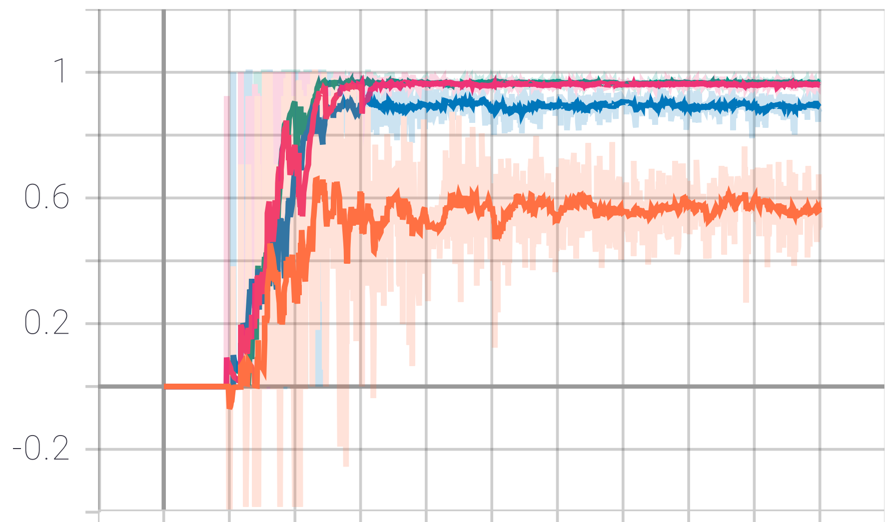

Figure 9: From `21-05-06_20:41:41`, all have phased–in proximity bonus with `ALICE_DOUBLE` set to None (green), 1000 (pink), 100 (blue) and 5000 (orange). Smoothing 0.9.

5 After correcting error in the construction of `MSEBits`

Just before `21-05-07_14:03:47`, corrected an error in the construction of `MSEBits` which meant it was looking at `closeness` not `alice_outputs_from_target` for the bits! This meant that trying to push closeness towards ± 1 whatever it should have been! Surprised it was so successful *before*. However, `21-05-07_14:03:47` is very unsuccessful, mean reward just oscillating around zero — this is because I messed up taking the means in commit `a3038c3` and I think previously.

That previous loss function was

```
self.loss_fn = lambda x, y: torch.mean(
   torch.sum(
      torch.nn.functional.mse_loss(x, y, reduction='none')
      + (mu / 2) * torch.square(x - torch.sign(x)),
      dim=-1
   )
)
```

which had the effect of trying to push the `closeness` to ± 1 with the second term, and — taking a sum over batches to be followed by a mean over a scalar — multiplying the learning rate by 32.

The 'correct' formulation is

```
self.loss_fn = (
   lambda x, y, z:
   torch.nn.functional.mse_loss(x, y)
   + (mu / 2)
   * torch.nn.functional.mse_loss(z, torch.sign(z))
) # for both the mse_loss functions this implies reduction='mean'
```

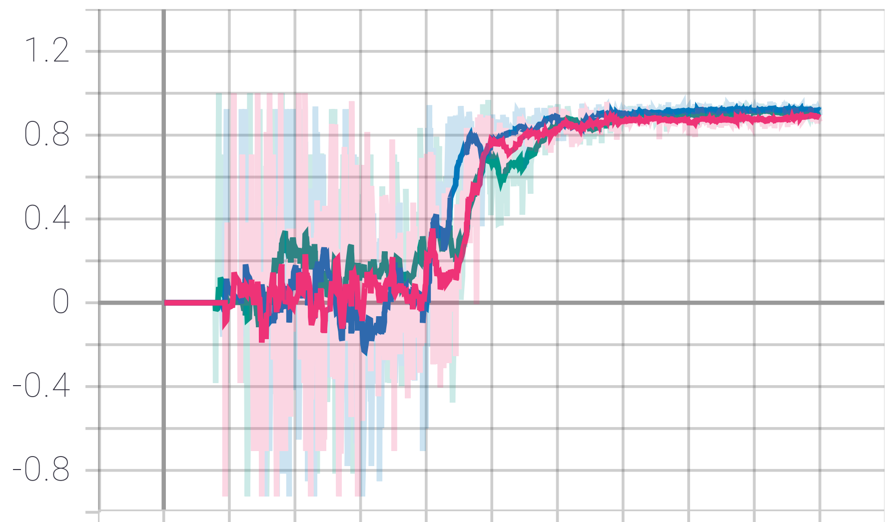

Figure 10: The rewards for `21-05-08_15:50:47`, for the three hp runs: 1 (pink), 2 (blue) and 3 (green) with different tuples of random seeds. Smoothing 0.9.

with `z` representing the raw output.

Reverting to the 'previous' get a pretty good run at `21-05-07_17:41:29`, but now have code and decode books on view in the run — and Alice's code book doesn't change at all!!!

6 QPerCode

`QPerCode` is a strategy for Alice with its net outputting $2^{N_{\text{code}}}$ Q–values. For SGD learning rates of 0.1 and above doesn't do well, but with $lr = 0.001$ at `21-05-08_14:45:11BST` (and double with period 500 and Huber `beta=0.5`) get good results, peaking at reward of 0.870.

The run `21-05-08_15:50:47` for three choices of random seeds shows that this approach is robust. It's also quick (50 000 iteration). See Figure 10. The config has

```
hyperparameters = {
    'N_ITERATIONS': 50 * (10 ** 3), # 5 * (10 ** 5),
    'RANDOM_SEEDS': [
    (868291, 274344, 358840, 94453),
    (382832, 68444, 754888, 857796),
    (736520, 815195, 305871, 974216)
    ],
    'ALICE_NET': 'FFs(3, 50)',
    'BOB_LAYERS': 3,
    'BOB_WIDTH': 50,
    'BATCHSIZE': 32,
    'GAMESIZE': 32,
    'BUFFER_CAPACITY': 32 * 20000,
    'START_TRAINING': 20000,
    'N_SELECT': 16,
```

```
    'EPSILON_ONE_END': 2000,
    'EPSILON_MIN': 0.01,
    'EPSILON_MIN_POINT': 40000,
    'ALICE_PLAY': 'QPerCode',
    'ALICE_TRAIN': 'QPerCode',
    'BOB_STRATEGY': 'circular_vocab',
    'ALICE_OPTIMIZER': [
    'SGD(lr=0.01)'
    ],
    'BOB_OPTIMIZER': [
    ('SGD', '{"lr": 0.01}')
    ],
    'ALICE_LOSS_FUNCTION': 'Huber(beta=0.5)',
    'BOB_LOSS_FUNCTION': ('torch.nn.MSE', {}),
    'ALICE_PROXIMITY_BONUS': 10 ** 8,
    'ALICE_PROXIMITY_SLOPE_LENGTH': 10000,
    'ALICE_LAST_TRAINING': 100 * (10 ** 5),
    'NOISE_START': 10 ** 8,
    'NOISE': 0.,
    'ALICE_DOUBLE': 500
}

TUPLE_SPEC = (
(16,),
)
N_CODE = 8

SMOOTHING_LENGTH = 10000
SAVE_PERIOD = 10 ** 5
```

Run `21-05-08_17:18:16` for the first tuple of random seeds, varying the Huber `beta` as in Figure 11 suggests `beta=0.1` may be best.

6.1 The first ever 1!

On run `21-05-08_18:30:55` (with a new tuple of seeds) using noise starting at `30000` gives for the first time a reward of close to 1 with Alice's code book using 16 codes for the 16 numbers for the three with non–zero noise (of 0.1, 0,2 and 0.3). Gets

```
 ---- Table of results ----

code hp_run noise result
00000    1     0.0 (-0.901, 50000)
00001    2     0.1 (-0.990, 70000)
00002    3     0.2 (-0.972, 70000)
00003    4     0.3 (-0.967, 70000)
-------------------------
```

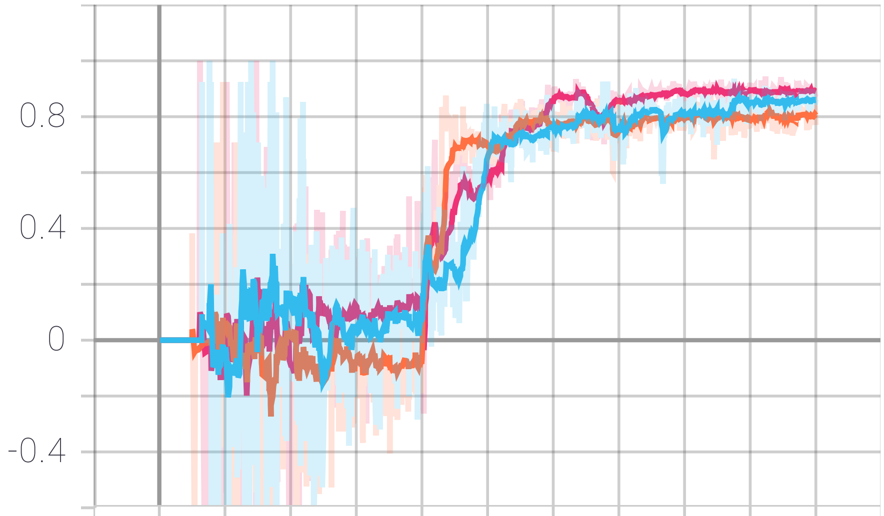

Figure 11: Run `21-05-08_17:18:16` showing rewards for Huber `beta` being 1 (blue), 0.5 (orange) and 0.1 (pink). Smoothing 0.9.

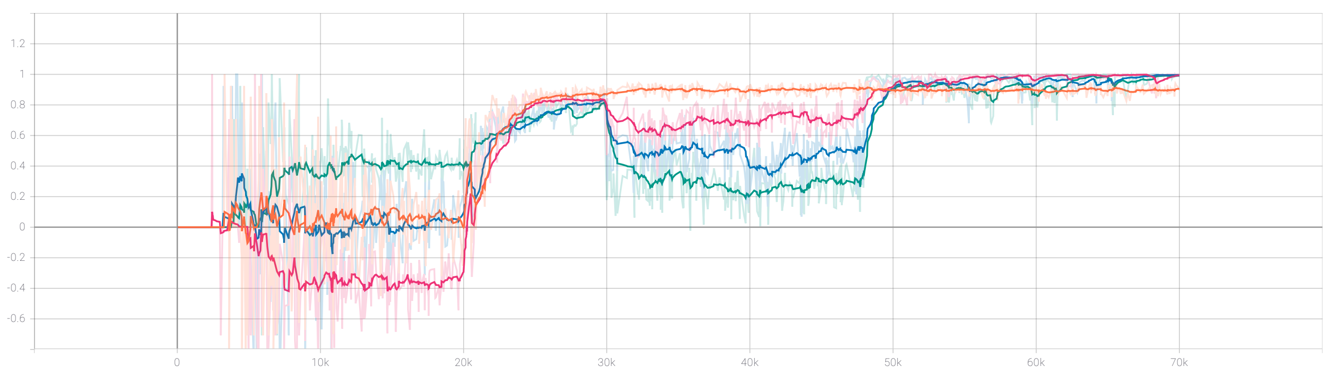

Figure 12: See narrative on `21-05-08_18:30:55`. The lines are respectively orange, pink, blue, green. Smoothing 0.9.

marginally favouring noise of 0.1. See Figure 12.

6.2 Varying `ALICE_DOUBLE`, `N_NUMBERS` and `N_CODE`

Then ran at `21-05-08_23:17:59` with

```
    'ALICE_DOUBLE': [None, 100, 300, 1000, 3000],
    'N_CODE': [8, 16],
    'N_NUMBERS': [16, 256]
```

getting

```
     ---- Table of results ----
```

```
    code hp_run       result
00000000    1 (-0.998, 70000)
00000001    2 (-0.994, 70000)
00000010    3 (-0.979, 70000)
00000011    4 (-0.984, 70000)
00000100    5 (-0.992, 70000)
00000101    6 (-0.993, 70000)
00000110    7 (-0.963, 70000)
00000111    8 (-0.977, 70000)
00000200    9 (-0.994, 70000)
00000201   10 (-0.994, 70000)
00000210   11 (-0.907, 70000)
00000211   12 (-0.972, 70000)
00000300   13 (-0.902, 60000)
00000301   14 (-0.963, 70000)
00000310   15 (-0.654, 40000)
00000311   16 (-0.847, 70000)
00000400   17 (-0.810, 70000)
00000401   18 (-0.909, 70000)
00000410   19 (-0.644, 40000)
00000411   20 (-0.728, 40000)
--------------------------
```

`ALICE_DOUBLE: None` is the best, which deals pretty easily with all the sub–options, working marginally better with `N_CODE: 8` for both `N_NUMBERS: 16` and `N_NUMBERS: 256`. However, perhaps need to run longer than 70 000 iterations for higher `N_NUMBERS` and `N_NUMBERS`? Also, perhaps some `ALICE_DOUBLE`, say 100 to 1000, may help stabilise? Could revert to `ALICE_DOUBLE: 500` as in `21-05-08_18:30:55`. May also need to pay more attention to the Bob side of things.

For session `21-05-09_12:21:11` (terminated early), with `N_CODE: 8`, `N_NUMBERS: 256` increasing `N_SELECT: 16` to `N_SELECT: 256` makes it a lot harder. More in the 0.7s and flat. It's also much slower has Bob has to try 256 rather than 16. However, `21-05-09_17:56:02` with four alternative tuples of seeds, does everything quickly and to near 1. It also suggest `DOUBLE` None or 500 doesn't make much difference — may None slightly more successful overall and should also be chosen on Occam principle.

Maybe set Python seed too?

7 Using `MaxTempered` layers

Defined `MaxTempered` layers, of which `MaxTemperedInFocused` seems better than `MaxTemperedOutFocused`, in `src.lib.max_tempered_layers.py`, and tried this out with Alice in `21-05-11_17:21:41` which seems as good as `21-05-08_23:17:59`'s first `hp_run`, albeit doubtless slower.

For `N_SELECT=256`, the runs `21-05-12_21:01:48` `21-05-13_09:05:41` compare a feed forward and max layer networks— see Figure 13. So with 200 000 iterations all about the same. Recall that we switch off noise just over a bufferful before the end of the run.

Now with 70 000 at `21-05-13_14:17:04` compare feed forward with `MaxLayer`, now with `relu=True`, find the `MaxLayer` with or without `bias_included` does better than feed forward (essentially 1.0 rather

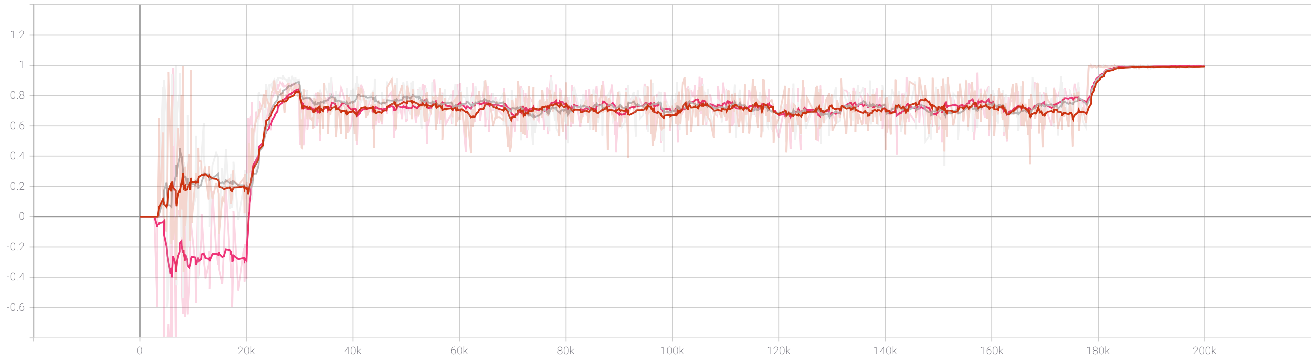

Figure 13: From 21-05-12_21:01:48, MaxLayer with bias_included=False (red) and MaxLayer bias_included=0.5 (grey) plus 21-05-13_09:05:41 feed forward (pink). Smoothing 0.9.

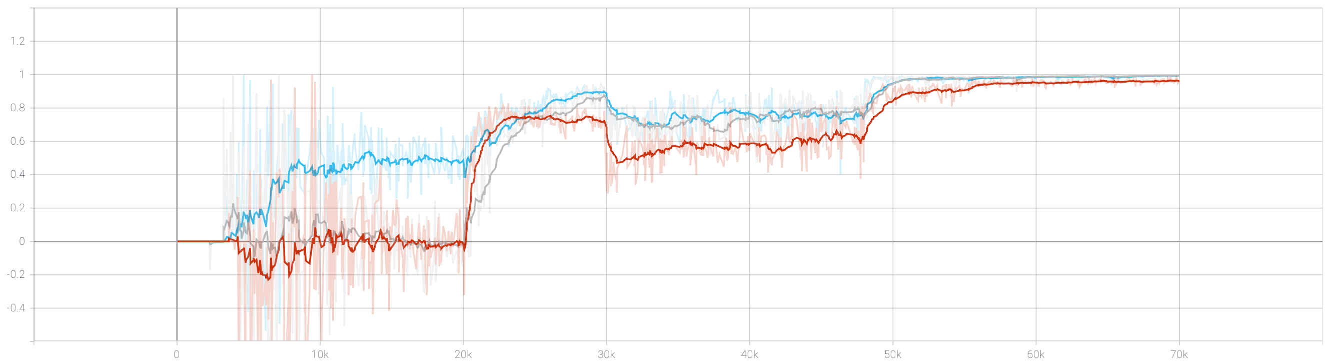

Figure 14: From 21-05-13_14:17:04, feed forward FFs(3, 50) (red), MaxNet("In", 3, 50, relu=True) (grey) and MaxNet("In", 3, 50, bias_included=True, relu=True) (cyan). Smoothing 0.9.

than 0.95). However takes about two hours rather than one–and–a–half. Whether MaxLayer is better depends on how few iterations needed to get feed forward or MaxLayer to 1.0. Note that all the MaxLayer so far have beta=0.2, which means that its layers are 0.8 a normal feed forward are 0.2 a max–tempered layer — so should try higher beta too. See Figure 14.

Dropout instead of ReLU doesn’t seem to help — see 21-05-14_22:23:16.

So the best configuration appears to be that of 21-05-09_17:56:02

```
{ #TODO enable dictionary-based choices for finer choosing
   ’N_ITERATIONS’: 70 * (10 ** 3), # 5 * (10 ** 5),
   ’RANDOM_SEEDS’: [
   (714844, 936892, 888616, 165835),
   (508585, 487266, 751926, 247136),
   (843402, 443788, 742412, 270619),
   (420915, 961830, 723900, 510954)
   ],
   ’ALICE_NET’: ’FFs(3, 50)’,
   ’BOB_LAYERS’: 3,
   ’BOB_WIDTH’: 50,
   ’BATCHSIZE’: 32,
```

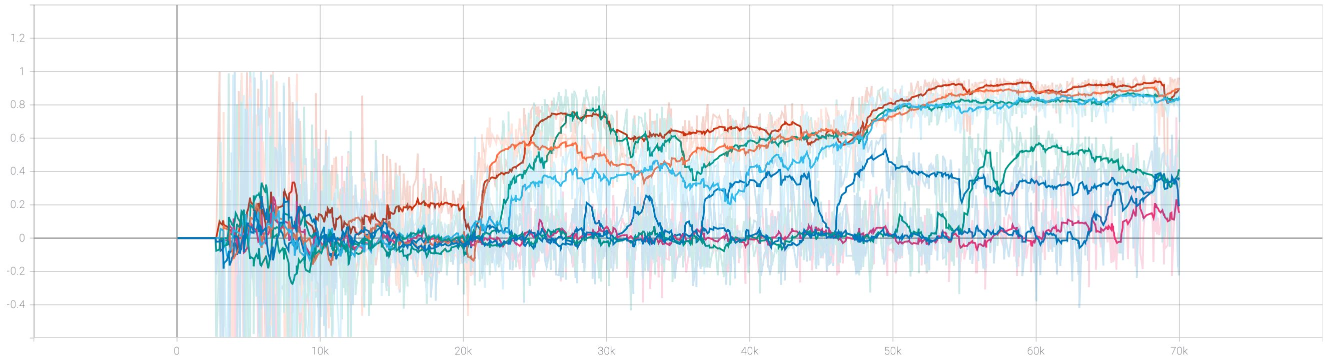

Figure 15: blue lower, green lower, cyan, orange, pink, blue, green, red

```
    'GAMESIZE': 32,
    'BUFFER_CAPACITY': 32 * 20000,
    'START_TRAINING': 20000,
    'N_SELECT': 256, #16,
    'EPSILON_ONE_END': 2000, #25000, # 40000,
    'EPSILON_MIN': 0.0,
    'EPSILON_MIN_POINT': 20000, #3 * (10 ** 5),
    'ALICE_PLAY': 'QPerCode',
    'ALICE_TRAIN': 'QPerCode', # 'FromDecisions',
    'BOB_STRATEGY': 'circular_vocab',
    'ALICE_OPTIMIZER': [
    'SGD(lr=0.01)'
    ],
    'BOB_OPTIMIZER': [
    ('SGD', '{"lr": 0.01}')
    ],
    'ALICE_LOSS_FUNCTION': [
    'Huber(beta=0.1)'
    ],
    'BOB_LOSS_FUNCTION': ('torch.nn.MSE', {}), # 'Same',
    'ALICE_PROXIMITY_BONUS': 10 ** 8, # 30000 * (10 ** 3),
    'ALICE_PROXIMITY_SLOPE_LENGTH': 10000,
    'ALICE_LAST_TRAINING': 100 * (10 ** 5),
    'NOISE_START': 30000,
    'NOISE': 0.1,
    'ALICE_DOUBLE': [None, 500],
    'N_CODE': 8,
    'N_NUMBERS': 256
}
```

which got

```
    ---- Table of results ----
```

```
code hp_run result
00000    1 (-0.986, 70000)
00001    2 (-0.987, 70000)
10000    3 (-0.978, 70000)
10001    4 (-0.985, 70000)
20000    5 (-0.984, 70000)
20001    6 (-0.973, 70000)
30000    7 (-0.990, 70000)
30001    8 (-0.979, 70000)
```

Abandon `DOUBLE` on Occamist grounds.

Dropping noise seems to worsen to low 0.90s — see `21-05-15_07:57:53`.

For `N_CODE=4`, get `0.973` in the run at `21-05-15_11:49:36` at iteration 70 000, in two hours. At the end of this run Alice uses 13 codes.

For `N_CODE=2`, get `0.831` in the run at `21-05-15_14:20:40` at iteration 70 000, in two hours. At the end of this run Alice uses 4 codes.

For `N_CODE=8` again, with `21-05-15_17:16:53` get essentially 1 in 35 000 iterations. (Note forgot to shorten smoothing length so this result is evident in TensorBoard, not in the log.) Took 55 min.

With `21-05-15_17:16:53` get 0.975 in 17 500 iterations. So is beginning to degrade. Took 27 min.

With `21-05-15_20:18:01` get 0.982 in 25 000 iterations. Good enough? Took 39 min.

30 000 iterations gives 0.981. Stick with 25 000.

With FFs in `21-05-15_21:54:38` get 0.981 in 25 000 iterations. Good enough? Took 27 min. So better than the MaxNets.

At `21-05-15_22:28:32`, compared MaxNet and FFs with 8 tuples of seeds, finding very similar mean rewards, MaxNet less (sample) SD (60% of FFs) but much slower: 35 min v. 24 min. See `21-05-15_22:28:32_MaxNet_v_FFs.ods` for mean and SD.

At `21-05-16_13:21:58` showed that again both MaxNet and FFs suceed if `N_SELECT=16`, taking around 15 and 9 min respectively.

At `21-05-19_16:57:58`, using FFs, showed that even `N_CODE=2**14` (about the largest my GPU can handle) works well — including if shuffle the numbers so that the state numbers which Alice and Bob know are shuffled before measuring distances.

```
---- Table of results ----

code hp_run result
00     1 (-0.987, 24997)  With no shuffling
01     2 (-0.989, 24997)  With shuffling
--------------------------
```

With ‘Near misses only’ reward method at `21-05-20_19:00:41` got

```
 ---- Table of results ----

code hp_run result
0     1 (-0.396, 24997)
1     2 (-0.417, 24997)
2     3 (-0.458, 24997)
```

```
   3     4 (-0.412, 24997)
   4     5 (-0.374, 24997)
   5     6 (-0.466, 24997)
   6     7 (-0.352, 24997)
   7     8 (-0.429, 24997)
   -------------------------
```

with eight different seeds. Not bad considering 1 away is around 0.33... and 2 away 0.166... . Re–proportioning from 25 000 to 70 000 iterations makes no difference. In both cases, in the mid–to–late stages the loss *increases*.

1. ☑ How is this increasing loss possible? Is it due to too high learning rate — although is successful learning — or is it something more intrinsic, for example that bigger losses and higher rewards are consistent *and* training somehow pushes that. Suspect it's due to move to full greedy play and noise as seems to start when epsilon ends and end when noise stops. And small plateau after buffer full of greedy and before noise fills it. Not sure that can be the full story as nonetheless loss at the end is greater than at the beginning! Is it that when you start guessing better there's a bigger error??!!!??!??! Also with 100 000 iterations in total, loss is going up after all the buffer is full of noise!!

   I don't think it's to do with epsilon or noise as persists after those (e.g. `21-05-22_21-18-59`). It's because as the Q–values get pulled away from zero (training to improve a past call) this tends — at least in a (very long, end not yet seen) initial period — to make most subsequent calls have higher mean losses even though they also have higher mean rewards. This is consistent as rewards occur in a relatively small proportion of good plays, whereas losses occur in the (complementary, and therefore large, proportion of) the training from bad plays.

At `21-05-21_19:08:54` for exact only:

```
 ---- Table of results ----

code hp_run result
0     1 (-0.473, 80000)
1     2 (-0.464, 100000)
2     3 (-0.474, 90000)
3     4 (-0.475, 90000)
-------------------------
```

so that's pretty good too.

However, at `21-05-22_12:08:58`, change from `N_NUMBERS=2**14` and `N_SELECT=16` to `N_NUMBERS=256` and `N_SELECT=256` now it's much harder as there's the whole domain to choose from each time:

```
    ---- Table of results ----

   code hp_run result
   0     1 (-0.023, 24997)
   1     2 (-0.010, 21426)
   2     3 (-0.026, 24997)
   3     4 (-0.021, 24997)
```

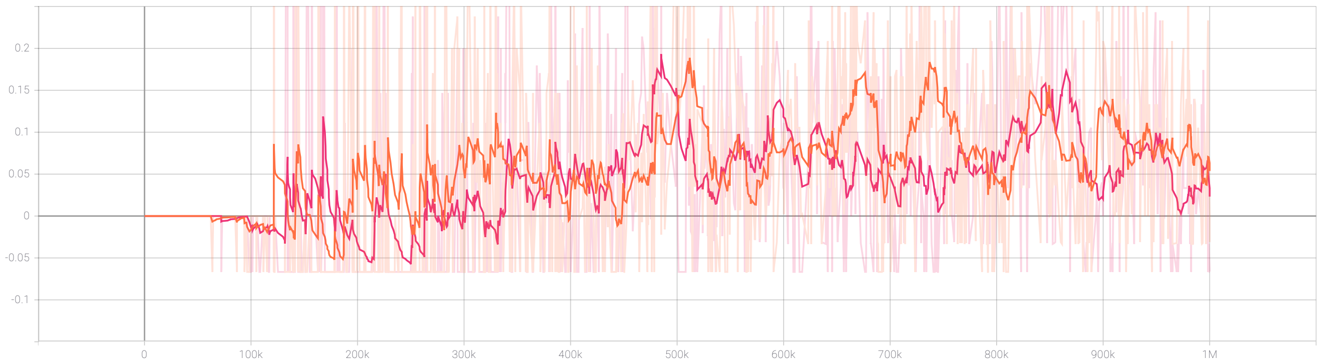

Figure 16: From 21-05-23_21:13:27, with hp runs 1 (orange) and 2 (pink). Smoothing 0.9.

```
-------------------------
```

compare with that run's

```
0 away is reward of 1
1 or more away is reward of -0.0039215686274509665
session_spec.random_reward_sd()=0.06262242910851405
```

I.e., it's much less than one SD above the mean of zero.

At `21-05-22_21-18-59` with `N_NUMBERS=256` and `N_SELECT=256`, 1 000 000 iterations and (non–minimal) epsilon lasting until 600 000, we see in the last 500 000 an increase of smoothed 0.9 from `0.01353` to `0.06488` and still climbing at that rate or better. So around `0.01` per 100 000 iterations! And the whole 1 000 000 take 21 h 30 min.

NOTE All in the above Alice uses `codes` not, as she should `greedy_codes`. Now changed!

A run at `21-05-23_21:13:27` with config as in `21-05-22_21-18-59` but with `N_NUMBERS=16` and `N_SELECT=16` rapidly varying results —see Figure **??** — with much better peaks but poor end points

```
 ---- Table of results ----

code hp_run result
0     1 (-0.185, 740000)
1     2 (-0.176, 850000)
-------------------------
```

8 Things to try

1. What codes does best Alice generate?
2. Try using the loss function to constraint outputs to nearer bit values. Try increasing the weighting of this.
3. How quickly can `epsilon` be tapered?
4. Vary learning rates.

5. Vary `modulus`, `N_CODE` and `N_SELECT`.
6. Introduce noise.
7. Alice strategy with a code, as input and the output are values for the numbers. In each play (or train?) step feed all the codes in and the outputs indicate how well represents each number???
8. Try best strategy but with Alice outputs having dimension `2 ** N_CODE`.
9. Train bits successively.
10. Look at MARL literature.
11. (At some stage in the training) introduce a 'proximity bonus' into Alice's training, which increases (in the same way) both the closeness of codes and the rewards if Bob's decision is right or nearly so.
12. Do a second sweep of `epsilon` going from high to low — perhaps for one player only? Definitely should re-`epsilon`–randomise Bob as otherwise Alice will never (or rarely if `N_SELECT < N_CODE`) get fed choices not in his decoding book. And I think Alice too, so Bob can learn new codes.
13. Random seeds seem to play a significant role — at least for short ($\sim$ 12 500) iteration training. Test how significant for 500 000 iterations.
14. Simulate use of a code–decode book pair.

9 Modifying the environment

1. ☑ Large `N_NUMBERS`.
2. ☐ Reward depends on being very close to the right answer.
3. ☑ Numbers are shuffled in a random way before reward is allocated — making the environment harder to understand.
4. ☐ Numbers translate into separate 'streams' for rewards, e.g. using modulo. An alternative is to use the multi–channel potential of the game set–up — but this keeps the environment relatively transparent.
5. ☐ What's a meaningful way to turn this into a multi–episode game?
6. ☐ Is there an environment where Alice is guiding Bob through a graph? (How does she know the best route?)
7. ☐ How would Alice's NN work if there were very many codes (2^{16} doesn't work with `QPerCode` but 2^{14} does).
8. ☐ After Alice sends Bob a code, he sends one back (trying for the same code book). Potential strategy for Alice:
   (a) Alice (assuming `N_NUMBERS` not too big) generates a code book ever so often (might go with `DOUBLE`).

(b) She receives the code from Bob and uses the code book to find the closest code from a number.

(c) To play she uses that number.

(d) To train, she runs this number through her NN getting a Q for her version of the code. Loss is MS difference between that Q and the reward. This assumes of course that the reward for this Bob to Alice episode of the game is calculated on the same basis as for the Alice to Bob episode. (If not she'd need another NN, as would Bob, and these would really be two separate games.)

9. ☐ Multiple agents — perhaps relates to Graph Neural Networks.

10. ☐ Distribute values randomly in unit disc (or unit square or torus or sphere). Let's do sphere: see stackoverflow: generate a random sample of points distributed on the surface of a unit sphere.

References

[1] J. He, J. Chen, X. He, J. Gao, L. Li, L. Deng et al., *Deep reinforcement learning with a natural language action space*, *arXiv preprint arXiv:1511.04636* (2015) .

[2] G. Dulac-Arnold, R. Evans, H. van Hasselt, P. Sunehag, T. Lillicrap, J. Hunt et al., *Deep reinforcement learning in large discrete action spaces*, *arXiv preprint arXiv:1512.07679* (2015) .

[3] S. J. Majeed and M. Hutter, *Exact reduction of huge action spaces in general reinforcement learning*, *arXiv preprint arXiv:2012.10200* (2020) .