# VLMMLR Packing Project

Jack Maguire – 10/16/2019

# Motivation

- When I design ~100 amino acid proteins, I never get the same sequence twice
  - This is good for sampling diversity
  - Might be a red flag that we can do better

- If we can identify unfruitful amino acids early in the packing process we can either
  - Skip them to save runtime, run more trajectories
  - Divert their resources to sample in better AA space

# Unsolicited Opinions

- So far, VLMMLR has over-sampled huge projects
- In my opinion, the future of ML *inside* Rosetta would be better served as many little projects
  - Neural Networks are hard to maintain/refactor
  - Easier to just make a new one every few years
- This project:
  - Can be done in minutes
    - Tried training over a weekend, barely helped
  - Largely impactful (noticable speedup)

# Data For Each AA At Each Position

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | ... | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Temp | 100 | 30 | 10 | 3 | 0.1 | 0.03 | 0.01 | 100 | 30 | 10 | 3 | | 0 |
| acc rate to AA | 1 | 0.8 | 0.3 | 0.2 | 0.1 | ... | | | | | | | |
| acc rate from AA | 0.9 | 0.4 | 0.5 | 0.1 | 0.0 | ... | | | | | | | |
| acc rate intra AA | 0.95 | 0.6 | 0.3 | 0.2 | 0.0 | ... | | | | | | | |

# Plan

| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | … | 20 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Temp | 100 | 30 | 10 | 3 | 0.1 | 0.03 | 0.01 | 100 | 30 | 10 | 3 | | 0 |
| acc rate to AA | 1 | 0.8 | 0.3 | 0.2 | 0.1 | … | | | | | | | |
| acc rate from AA | 0.9 | 0.4 | 0.5 | 0.1 | 0.0 | … | | | | | | | |
| acc rate intra AA | 0.95 | 0.6 | 0.3 | 0.2 | 0.0 | … | | | | | | | |

Model input is 15 values:

- 3 acceptance rates
- First 5 rounds

Model output is a classification:

1 if the AA ends up being the final designed AA

0 if a different AA is chosen for this position

# Setup

- Install tensorflow
  - pip3 install tensorflow-gpu==2.0.0-alpha
  - (don't need gpu version)

- git clone git@github.com:JackMaguire/VLMMLR_packer_contest.git

- cd VLMMLR_packer_contest
- python3 run.py
- python3 analyze.py

## run.py

If you have a GPU, you can install tensorflow-gpu and disable GPU usage with these os.environ commands

I never import keras directly, but import it through tensorflow

```
import os
#Comment this out to use your gpu
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = ""

from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
#...

import numpy
import pandas as pd

numpy.random.seed( 0 )
```

# run.py: PARAMETERS

```
window_size = 5 # consider 5 loops of data (columns) at once

channels = 3


extra_values = 20


num_input_values = (window_size * channels) + extra_values
```

This logic defines the number of input parameters for the network

# run.py: CREATE MODEL

Make a neural network with 5 layers:

1 input

3 Dense

1 output

We tell TF to score the model with binary crossentropy, but to also measure binary accuracy

This will make more sense later

```
# Layers

input = Input(shape=(num_input_values,), name="in",
dtype="float32" )

dense1 = Dense( name="dense1", units=100, activation="relu"
)( input )

dense2 = Dense( name="dense2", units=100, activation="relu"
)( dense1 )

dense3 = Dense( name="dense3", units=100, activation="relu"
)( dense2 )

output = Dense( name="output", units=1, activation='sigmoid'
)( dense3 ) # final value is between 0 and 1


model = Model(inputs=input, outputs=output )


metrics_to_output=[ 'binary_accuracy' ]

model.compile( loss='binary_crossentropy', optimizer='adam',
metrics=metrics_to_output )

model.summary()
```

# run.py: LOAD DATA

You can mostly skip this section

The only thing to know is that I lazily hardcoded the data paths for the test data

You can optionally pass the --training_data flag

```
input,output = read_from_file( args.training_data )
test_input,test_output = read_from_file( "data/..." )
```

# run.py: TRAIN

Interesting Callbacks:

- ModelCheckpoint
- ReduceLROnPlateau
- LearningRateScheduler
- TensorBoard (advanced)

Other Things to play with:

- batch_size
- class_weight
- epochs
- optimizer (in model.compile)

```
csv_logger = tensorflow.keras.callbacks.CSVLogger( "training_log.csv", separator=',', append=False )

# Many fun options: https://keras.io/callbacks/

callbacks=[csv_logger]
```

```
class_weight = {0: 1., 1: 19.}
```

```
model.fit( x=input, y=output, batch_size=64, epochs=1, callbacks=callbacks, validation_data=(test_input,test_output), class_weight=class_weight )
```

```
model.save( args.model ) # use --model [model name]
```

run.py

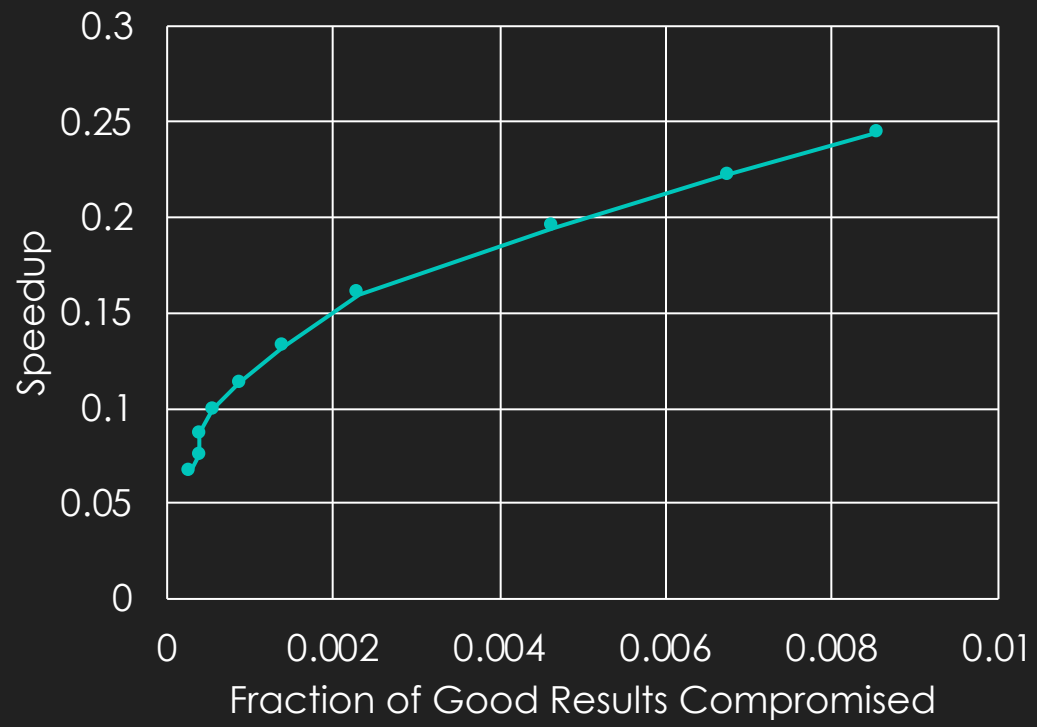
Output tells you:

- training loss
- training accuracy (0.5)
- validation loss
- validation accuracy (0.5)

```
...

Epoch 4/10

1500000/1500000 [==============================] - 38s 25us/sample -
loss: 1.2698 - binary_accuracy: 0.6487 - val_loss: 0.7122 -
val_binary_accuracy: 0.6926

Epoch 5/10

1500000/1500000 [==============================] - 38s 25us/sample -
loss: 1.2469 - binary_accuracy: 0.6541 - val_loss: 0.7165 -
val_binary_accuracy: 0.6924

Epoch 6/10

1500000/1500000 [==============================] - 38s 25us/sample -
loss: 1.2343 - binary_accuracy: 0.6565 - val_loss: 0.6852 -
val_binary_accuracy: 0.6999

Epoch 7/10

 918912/1500000 [=================>............] - ETA: 13s - loss:
1.2205 - binary_accuracy: 0.6625
```

# Interpreting analyze.py

```
500000/500000 [==============================] - 16s
32us/sample - loss: 1.4053 - binary_accuracy: 0.3756

cutoff, true_pos, true_neg, false_pos, false_neg, fraction of
work prevented, fraction of good AAs lost

0.50 24999 162800 311985 216 0.244524 0.008566329565734682

0.45 25045 147530 327255 170 0.22155 0.0067420186396985925

0.40 25098 129704 345081 117 0.1947315 0.004640095181439619

0.35 25157 106631 368154 58 0.1600335 0.002300218124132461

0.30 25180 87911 386874 35 0.131919 0.001388062661114416

0.25 25193 75143 399642 22 0.1127475 0.0008724965298433472

0.20 25201 65766 409019 14 0.09867 0.000555225064445

0.15 25205 57506 417279 10 0.086274 0.0003965893331746976

0.10 25205 50115 424670 10 0.0751875 0.0003965893331746976

0.05 25208 44226 430559 7 0.0663495 0.0002776125322228832
```

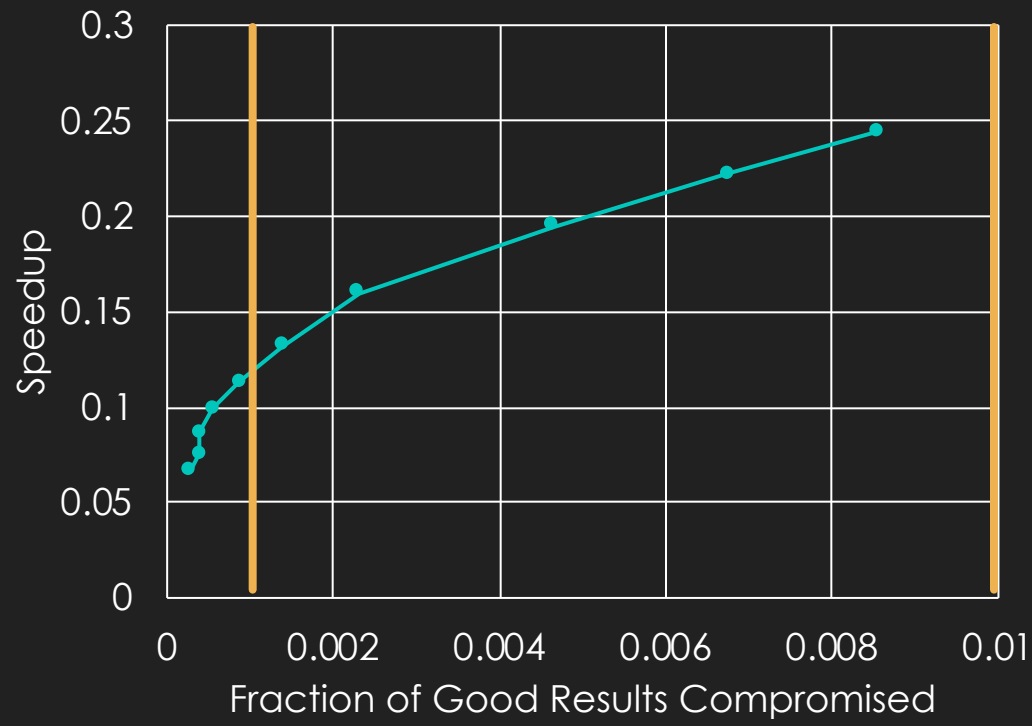



This was collected with jack.py

# Interpreting analyze.py

Let’s have 2 winners:

1. What is your speedup when you sacrifice 0.01 of the good results?
2. What is your speedup when you sacrifice 0.001 of the good results?



# score.py : reporting results

```
Output:


cutoff, frac work eliminated, frac good
AAs lost

0.07167125 0.289617
0.009994051160023795

0.006922871 0.1342335
0.0009914732936744
```

- When you are done training a model, run score.py
- Don't do this too often. Strictly speaking you shouldn't even have access to this data.
- Add your .h5 model file to results/ and report these results in results.md

# More Training Data

https://drive.google.com/file/d/1S7Q8GGTs3fMZ2POA0j8xyqMlg0l8aQ5X/view?usp=sharing