Deep-Learning-based DNA Sequence Fast Decoding

Team NTHU-1

October 14, 2022

We read the original paper mentioned in README of GitHub repository of this competition and find the source code on GitHub.

In this task, we would adopt this model, slightly modify it so that it's suitable for newer *Tensorflow* and tune its hyperparameters so as to obtain the ideal performance.

1 Model structure

This model, Leopard Unet, is composed of 4 main part: **Input**, **Encoder**, **Decoder** and **Output**.

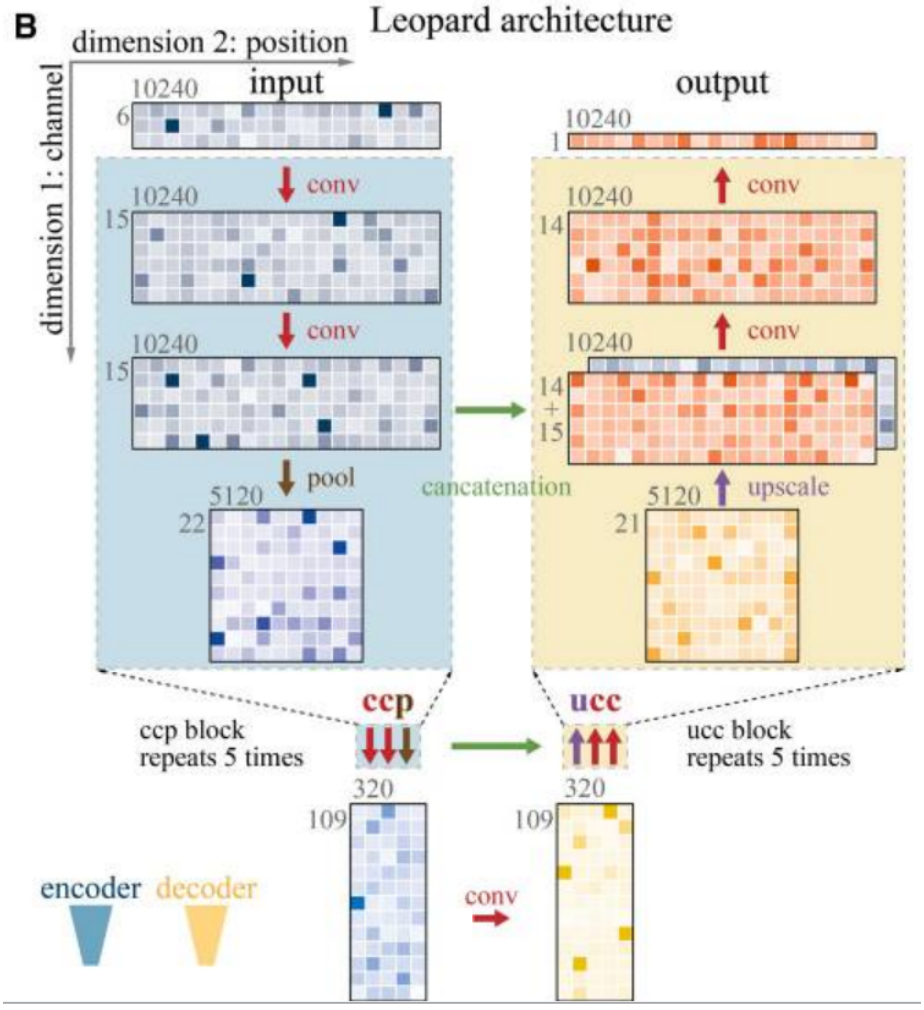

Figure 1: Diagram of the model, adapted from the original paper

1.1 Encoder

The **Encoder** comprises 5 **CCP** *(Convolution-Convolution-Pooling)* blocks, each of which contains two 1D convolution layer and a max-pooling layer sequentially.

In each step, the length of data is divided by 5 during the process of pooling.

1.2 Decoder

Similarly, 5 **UCC** *(Upscaling-Convolution-Convolution)* blocks constitute the **Decoder**. The upscaling component is formed by 1D convolution-transpose layer. The first of the convolution layer is concatenated to the second convolution layer in the corresponding **CCP** block.

Each time, the length of data is 5 times as previous block due to upscaling.

1.3 Details of Model Layers

There are 61 layers in total, all of which contain 475,969 parameters, where 99.5% or 474,017 ones are trainable.

Table 1: Layers of the model

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 10240, 5)]	0	[]
conv1d (Conv1D)	(None, 10240, 15)	540	[‘input_1[0][0]’]
batch_normalization (BatchNormalization)	(None, 10240, 15)	60	[‘conv1d[0][0]’]
conv1d_1 (Conv1D)	(None, 10240, 15)	1590	[‘batch_normalization[0][0]’]
batch_normalization_1 (BatchNormalization)	(None, 10240, 15)	60	[‘conv1d_1[0][0]’]
max_pooling1d (MaxPooling1D)	(None, 5120, 15)	0	[‘batch_normalization_1[0][0]’]
conv1d_2 (Conv1D)	(None, 5120, 22)	2332	[‘max_pooling1d[0][0]’]
batch_normalization_2 (BatchNormalization)	(None, 5120, 22)	88	[‘conv1d_2[0][0]’]
conv1d_3 (Conv1D)	(None, 5120, 22)	3410	[‘batch_normalization_2[0][0]’]
batch_normalization_3 (BatchNormalization)	(None, 5120, 22)	88	[‘conv1d_3[0][0]’]
max_pooling1d_1 (MaxPooling1D)	(None, 2560, 22)	0	[‘batch_normalization_3[0][0]’]
conv1d_4 (Conv1D)	(None, 2560, 33)	5115	[‘max_pooling1d_1[0][0]’]
batch_normalization_4 (BatchNormalization)	(None, 2560, 33)	132	[‘conv1d_4[0][0]’]
conv1d_5 (Conv1D)	(None, 2560, 33)	7656	[‘batch_normalization_4[0][0]’]
batch_normalization_5 (BatchNormalization)	(None, 2560, 33)	132	[‘conv1d_5[0][0]’]
max_pooling1d_2 (MaxPooling1D)	(None, 1280, 33)	0	[‘batch_normalization_5[0][0]’]
conv1d_6 (Conv1D)	(None, 1280, 49)	11368	[‘max_pooling1d_2[0][0]’]

Layer (type)	Output Shape	Param #	Connected to
batch_normalization_6 (BatchNormalization)	(None, 1280, 49)	196	[‘conv1d_6[0][0]’]
conv1d_7 (Conv1D)	(None, 1280, 49)	16856	[‘batch_normalization_6[0][0]’]
batch_normalization_7 (BatchNormalization)	(None, 1280, 49)	196	[‘conv1d_7[0][0]’]
max_pooling1d_3 (MaxPooling1D)	(None, 640, 49)	0	[‘batch_normalization_7[0][0]’]
conv1d_8 (Conv1D)	(None, 640, 73)	25112	[‘max_pooling1d_3[0][0]’]
batch_normalization_8 (BatchNormalization)	(None, 640, 73)	292	[‘conv1d_8[0][0]’]
conv1d_9 (Conv1D)	(None, 640, 73)	37376	[‘batch_normalization_8[0][0]’]
batch_normalization_9 (BatchNormalization)	(None, 640, 73)	292	[‘conv1d_9[0][0]’]
max_pooling1d_4 (MaxPooling1D)	(None, 320, 73)	0	[‘batch_normalization_9[0][0]’]
conv1d_10 (Conv1D)	(None, 320, 109)	55808	[‘max_pooling1d_4[0][0]’]
batch_normalization_10 (BatchNormalization)	(None, 320, 109)	436	[‘conv1d_10[0][0]’]
conv1d_11 (Conv1D)	(None, 320, 109)	83276	[‘batch_normalization_10[0][0]’]
batch_normalization_11 (BatchNormalization)	(None, 320, 109)	436	[‘conv1d_11[0][0]’]
conv1d_transpose (Conv1DTranspspose)	(None, 640, 72)	15768	[‘batch_normalization_11[0][0]’]
concatenate (Concatenate)	(None, 640, 145)	0	[‘conv1d_transpose[0][0]’, ‘batch_normalization_9[0][0]’]
conv1d_12 (Conv1D)	(None, 640, 72)	73152	[‘concatenate[0][0]’]
batch_normalization_12 (BatchNormalization)	(None, 640, 72)	288	[‘conv1d_12[0][0]’]
conv1d_13 (Conv1D)	(None, 640, 72)	36360	[‘batch_normalization_12[0][0]’]
batch_normalization_13 (BatchNormalization)	(None, 640, 72)	288	[‘conv1d_13[0][0]’]
conv1d_transpose_1 (Conv1DTranspose)	(None, 1280, 48)	6960	[‘batch_normalization_13[0][0]’]
concatenate_1 (Concatenate)	(None, 1280, 97)	0	[‘conv1d_transpose_1[0][0]’, ‘batch_normalization_7[0][0]’]
conv1d_14 (Conv1D)	(None, 1280, 48)	32640	[‘concatenate_1[0][0]’]
batch_normalization_14 (BatchNormalization)	(None, 1280, 48)	192	[‘conv1d_14[0][0]’]
conv1d_15 (Conv1D)	(None, 1280, 48)	16176	[‘batch_normalization_14[0][0]’]
batch_normalization_15 (BatchNormalization)	(None, 1280, 48)	192	[‘conv1d_15[0][0]’]

Layer (type)	Output Shape	Param #	Connected to
conv1d_transpose_2 (Conv1DTranspose)	(None, 2560, 32)	3104	[‘batch_normalization_15[0][0]’]
concatenate_2 (Concatenate)	(None, 2560, 65)	0	[‘conv1d_transpose_2[0][0]’, ‘batch_normalization_5[0][0]’]
conv1d_16 (Conv1D)	(None, 2560, 32)	14592	[‘concatenate_2[0][0]’]
batch_normalization_16 (BatchNormalization)	(None, 2560, 32)	128	[‘conv1d_16[0][0]’]
conv1d_17 (Conv1D)	(None, 2560, 32)	7200	[‘batch_normalization_16[0][0]’]
batch_normalization_17 (BatchNormalization)	(None, 2560, 32)	128	[‘conv1d_17[0][0]’]
conv1d_transpose_3 (Conv1DTranspose)	(None, 5120, 21)	1365	[‘batch_normalization_17[0][0]’]
concatenate_3 (Concatenate)	(None, 5120, 43)	0	[‘conv1d_transpose_3[0][0]’, ‘batch_normalization_3[0][0]’]
conv1d_18 (Conv1D)	(None, 5120, 21)	6342	[‘concatenate_3[0][0]’]
batch_normalization_18 (BatchNormalization)	(None, 5120, 21)	84	[‘conv1d_18[0][0]’]
conv1d_19 (Conv1D)	(None, 5120, 21)	3108	[‘batch_normalization_18[0][0]’]
batch_normalization_19 (BatchNormalization)	(None, 5120, 21)	84	[‘conv1d_19[0][0]’]
conv1d_transpose_4 (Conv1DTranspose)	(None, 10240, 14)	602	[‘batch_normalization_19[0][0]’]
concatenate_4 (Concatenate)	(None, 10240, 29)	0	[‘conv1d_transpose_4[0][0]’, ‘batch_normalization_1[0][0]’]
conv1d_20 (Conv1D)	(None, 10240, 14)	2856	[‘concatenate_4[0][0]’]
batch_normalization_20 (BatchNormalization)	(None, 10240, 14)	56	[‘conv1d_20[0][0]’]
conv1d_21 (Conv1D)	(None, 10240, 14)	1386	[‘batch_normalization_20[0][0]’]
batch_normalization_21 (BatchNormalization)	(None, 10240, 14)	56	[‘conv1d_21[0][0]’]
conv1d_22 (Conv1D)	(None, 10240, 1)	15	[‘batch_normalization_21[0][0]’]

2 Why do we choose this model?

// TODO?

3 Tuning

Below are the works that we tried to tune and optimize the performance of the model.

3.1 Volta and Ampère GPUs

The main GPUs of Gadi are NVIDIA Volata V100 cards. Nevertheless, there are 2 DGX-A100 node on Gadi equipped with the newest on the market so far NVIDIA Ampère A100 cards.

It's no doubt that the process of training becomes faster since A100 is more powerful than V100.

In addition, A100 is of 80GB momery, whereas there is only 32GB on V100. As a consequence, **batch size** could be far more larger.

3.2 Batch Size

In our experiments, we follow a hierarchical manner; that is, we first tested **batch size**, then we adjusted **learning rate** of a particular **batch size**.

3.2.1 32, 50

We tried some smaller **batch size**s but their results were not so good at all.

3.2.2 64

This is the best **batch size** we have ever found. We tested several **learning rate**, such as 0.001, 0.0025, 0.005, 0.01, 0.025, 0.05, among of which 0.01 gave rise to optimal result.

3.2.3 100

This is the default value. The maximum of *P-R AUC (Percision-Recall Area Under Curve)* could be only 0.46 or 0.47, which was somewhat greater than the baseline CNN model.

3.2.4 500

We encounter severe ***overfitting*** when it comes to this **batch size**; that is, *P-R AUC* could increase up to about 0.96 for the training set, whereas the ones for validation set were as low as 0.25. What's worse, since we increased the patience of early stop to 10, it exhuasted the 2-hour wall time.

3.2.5 800

The maximum **batch size** for V100 is 800 since it consumes about 31GB GPU meomery. For this **batch size**, the converage rate of validation loss were quite slow despite of **learning rate**. As a consequence, the results were also not so good.

3.2.6 1600, 2000, 2500

These three **batch size**s were only measured on A100 for sure. Their results were quite similar to 800.

3.3 Parallelization via Multiple GPUs

We also managed to parallelize the training and utilized mutliple GPUs. Nonetheless, the performance would decline a lot. Furthermore, the metrics would drop suddenly and dramatically after the number of GPU was over a value. We guessed that it might due to the simple, linear scale of **learning rate**, trying to find ad-hoc optimal **learning rate** for diffirent number of GPU yet in vain.

Eventually, we decide to train on only a single GPU since we believe that the performance counts more. Still, we provide the job script to train accross two A100s.

4 Result

The best performance of the metrics we have ever achieved were under following condition:

Batch Size 64
Learning Rate 0.01
Single GPU

Table 2: Performance of Single GPU

GPU Type	Loss *(Binary Crossentropy)*	P-R AUC	Dice coefficient	Binary Intersection of Union	Training Time [s]
V100	7.2768×10^{-4}	0.5297	0.3705	0.3625	3170.19
A100	7.3294×10^{-4}	0.5273	0.3944	0.3416	1306.11

Just as the above table shown, the training time of A100 was approximately two fifths of the one of V100 whereas other metrics were of little difference.