A3_part1_deepsea

November 16, 2022

1 A3 Part 1: Genomics & DeepSEA

Recent advances in sequencing have highlighted a need for new innovations in Deep Learning for Genomics. In this project, we will explore sequences and their features as well as predict functional effects of noncoding variants, which are regions of the genome that do not produce proteins. Our input includes base pair sequences from the human GRCh37 reference genome, a representative example of human genes, and the model outputs a series of chromatin feature activations, which represent which chromatin features are accessible, allowing transcription to take place. This work is based on a Nature Methods paper, **Predicting effects of noncoding variants with deep learning–based sequence model** (https://www.nature.com/articles/nmeth.3547), which revolutionized a deep learning approach for this problem.

1.1 Section 1: Exploring Genes

We will be using dna_features_viewer and bokeh for interactive plotting and visualization. Install them with `pip install dna_features_viewer bokeh`. In the first part of this assignment, we will be exploring the `IRF4` gene. Run the following commands.

1. Navigate to your assignment 3 directory and run `mkdir data` followed by `cd data`.
2. Run `wget http://ftp.ebi.ac.uk/pub/databases/gencode/Gencode_human/release_38/GRCh37_mappin`
3. Run `gunzip gencode.v38lift37.basic.annotation.gff3.gz`
4. Run `grep IRF4 gencode.v38lift37.basic.annotation.gff3 > IRF4.gff3`

[2]:
```
import os
import h5py
import numpy as np
import pandas as pd
import tensorflow as tf
from bokeh.io import output_notebook, show
from dna_features_viewer import GraphicFeature, GraphicRecord

# Set ROOT to be the path to the `data` directory
ROOT = "/home/jupyter/data"
```

You downloaded a gene annotation file from GENCODE (not to be confused with ENCODE). It contains information about gene features for all human genes, and annotations are available for mouse genes as well. The `IRF4.gff3` file contains annotations for the `IRF4` gene.

Q1.1 GFF / GTF files are tab separated files. We can use pandas to read in these files. Read in and display the `IRF4.gff3` file. Note that you will need to specify that it is tab delimited and that

there is no header.

[3]:
```
# FILL IN CODE HERE #
IRF4 = pd.read_csv('data/IRF4.gff3', sep = '\t', header = None).rename(columns␣
 ↪= {0 :"chromasome_name", 1:'annotation_source',
                                                                              ␣
 ↪ 2: 'feature_type', 3:'genomic_start_location',
                                                                              ␣
 ↪ 4: 'genomic_end_location',
                                                                              ␣
 ↪ 5: 'score(not used)',
                                                                              ␣
 ↪ 6: 'genomic_strand',
                                                                              ␣
 ↪ 7: 'genomic phase (for CDS features)',
                                                                              ␣
 ↪ 8: 'additional information as key-value pairs'})
IRF4
# FILL IN CODE HERE #
```

[3]:

	chromasome_name	annotation_source	feature_type	genomic_start_location	\
0	chr6	HAVANA	gene	391752	
1	chr6	HAVANA	transcript	391752	
2	chr6	HAVANA	exon	391752	
3	chr6	HAVANA	five_prime_UTR	391752	
4	chr6	HAVANA	exon	393098	
5	chr6	HAVANA	five_prime_UTR	393098	
6	chr6	HAVANA	CDS	393153	
7	chr6	HAVANA	start_codon	393153	
8	chr6	HAVANA	exon	394821	
9	chr6	HAVANA	CDS	394821	
10	chr6	HAVANA	exon	395847	
11	chr6	HAVANA	CDS	395847	
12	chr6	HAVANA	exon	397108	
13	chr6	HAVANA	CDS	397108	
14	chr6	HAVANA	exon	398828	
15	chr6	HAVANA	CDS	398828	
16	chr6	HAVANA	exon	401424	
17	chr6	HAVANA	CDS	401424	
18	chr6	HAVANA	exon	405018	
19	chr6	HAVANA	CDS	405018	
20	chr6	HAVANA	exon	407455	
21	chr6	HAVANA	CDS	407455	
22	chr6	HAVANA	stop_codon	407596	
23	chr6	HAVANA	three_prime_UTR	407599	

genomic_end_location	score(not used)	genomic_strand	\

0	411443	.	+
1	411443	.	+
2	391809	.	+
3	391809	.	+
4	393368	.	+
5	393152	.	+
6	393368	.	+
7	393155	.	+
8	395007	.	+
9	395007	.	+
10	395935	.	+
11	395935	.	+
12	397252	.	+
13	397252	.	+
14	398935	.	+
15	398935	.	+
16	401777	.	+
17	401777	.	+
18	405130	.	+
19	405130	.	+
20	411443	.	+
21	407598	.	+
22	407598	.	+
23	411443	.	+

	genomic phase (for CDS features) \
0	.
1	.
2	.
3	.
4	.
5	.
6	0
7	0
8	.
9	0
10	.
11	2
12	.
13	0
14	.
15	2
16	.
17	2
18	.
19	2
20	.

```
21                                      0
22                                      0
23                                      .

                 additional information as key-value pairs
0    ID=ENSG00000137265.15;gene_id=ENSG00000137265…
1    ID=ENST00000380956.9;Parent=ENSG00000137265.15…
2    ID=exon:ENST00000380956.9:1;Parent=ENST0000038…
3    ID=UTR5:ENST00000380956.9;Parent=ENST000003809…
4    ID=exon:ENST00000380956.9:2;Parent=ENST0000038…
5    ID=UTR5:ENST00000380956.9;Parent=ENST000003809…
6    ID=CDS:ENST00000380956.9;Parent=ENST0000038095…
7    ID=start_codon:ENST00000380956.9;Parent=ENST00…
8    ID=exon:ENST00000380956.9:3;Parent=ENST0000038…
9    ID=CDS:ENST00000380956.9;Parent=ENST0000038095…
10   ID=exon:ENST00000380956.9:4;Parent=ENST0000038…
11   ID=CDS:ENST00000380956.9;Parent=ENST0000038095…
12   ID=exon:ENST00000380956.9:5;Parent=ENST0000038…
13   ID=CDS:ENST00000380956.9;Parent=ENST0000038095…
14   ID=exon:ENST00000380956.9:6;Parent=ENST0000038…
15   ID=CDS:ENST00000380956.9;Parent=ENST0000038095…
16   ID=exon:ENST00000380956.9:7;Parent=ENST0000038…
17   ID=CDS:ENST00000380956.9;Parent=ENST0000038095…
18   ID=exon:ENST00000380956.9:8;Parent=ENST0000038…
19   ID=CDS:ENST00000380956.9;Parent=ENST0000038095…
20   ID=exon:ENST00000380956.9:9;Parent=ENST0000038…
21   ID=CDS:ENST00000380956.9;Parent=ENST0000038095…
22   ID=stop_codon:ENST00000380956.9;Parent=ENST000…
23   ID=UTR3:ENST00000380956.9;Parent=ENST000003809…
```

Q1.2 What information is stored in the GFF3 file? How is it represented (what do the rows and columns represent)? You may find this link helpful - https://www.gencodegenes.org/pages/data_format.html. It describes the data stored in GENCODE GFF / GTF files.

1.1.1 Written answer: The GFF3 file is a standard 9-column file that holds any and every feature that can be applied to a nucleic acid or protein sequence. Each column represents genomic feature (in our case 0 = chromasome name, 1 = annotation source, 2 = feature type (i.e. exon, gene, stop codon, etc.), 3 = genome start location, 4 = genome end location, 5 = score (which is not used), 6 = genomic strand {+ (forward) or - (reverse)}, 7 = genomic phase (which codon is the start codon), 8 = additional information about each feature in key value pairs). The rows represent specific genomic sequence examples which contain fields for each column.

Q1.3 Column 2 specifies different regions of the gene. What are the five_prime_UTR and three_prime_UTR entries? Are these regions translated to amino acids?

1.1.2 Written answer: The 5' UTR is a region of the mRNA directly upstrean of the initiaion codon at the 5' end of the coding sequence. The 3' UTR is a region of the mRNA directly downstream of the 3' end of the coding sequence. These regions are crucial transcription regulation: the 5' UTR is essential for recruitment of translational machinery (such as ribosomes) to initiate translation, the 3' UTR is necessary for translational termination as well as the recruitment of post translational machinery. These regions are not translated to amino acids.

Q1.4 Let us visualize these features using the dna_features_viewer library. We will be representing each row in the file as a `GraphicFeature` (see the documentation for details). Iterate over the rows of the `IRF4` dataframe and convert each one to a `GraphicFeature`. For each feature, you need to specify the start, end, strand, and label parameters (color is not required). The labels are derived from the third column of the dataframe. Append each feature to the `features` list.

Note - The positive strand should be specified as `strand=1`. You may need to subtract an offset from each start and end postion to make the plot readable.

[4]:
```
features = []

# FILL IN CODE HERE #
offset = 391752
for i in np.arange(len(IRF4)):
    row = IRF4.loc[i, :]
    start = row['genomic_start_location'] - offset
    end = row['genomic_end_location'] - offset
    strand = int(row['genomic_strand'] + str(1))
    label = row['feature_type']
    feat = GraphicFeature(start=start, end=end, strand=strand,
                    label=label)
    features.append(feat)

# FILL IN CODE HERE #

record = GraphicRecord(sequence_length=20000, features=features)
a = record.plot_with_bokeh(figure_width=10)
output_notebook()
show(a)
```

From the plot above, answer the following questions. Note that the plot is interactive and you can hover over the sections to see their anntations.

Q1.5: How many exons does the `IRF4` gene contain? Where are the introns located?

1.1.3 Written answer: The IRF4 gene contains 9 exons. The introns are located between the exons on the transcript between the start and stop codon.

Q1.6: What is the CDS feature and why does it overlap with the exons? Explain why the last CDS doesn't fully overlap with the last exon.

1.1.4 Written answer: The CDS represents the coding region of RNA whose sequence determines the sequence of amino acids in the translated protein. An exon is a sequence which remains present in mature RNA. A CDS is a sequence that remains present in the mature RNA and codes for a protein (i.e. gets translated). Not all exons get translated - the 5'UTR and 3'UTR remain untranslated even in the mature RNA sequence. Thus, the last CDS doesn't fully overlap with the last exon since the last exon is comprised of a small coding region and the 3'UTR.

Q1.7: Look up the `IRF4` gene and list one visible phenotype which is associated with this gene.

1.1.5 Written answer: Genetic variants in the IRF4 gene are involved in skin/hair/eye pigmentation variations.

1.2 Section 2: DeepSEA

Let's download the data for the DeepSEA model. Navigate to your `data` directory and run the following commands 1. `wget http://deepsea.princeton.edu/media/code/deepsea_train_bundle.v0.9.tar.gz` 2. `tar xvzf deepsea_train_bundle.v0.9.tar.gz`

You should now have a directory named `deepsea_train` which contains several `.mat` files. We will only be using `train.mat` for the rest of this assignment.

Q1.8 Data loading: Implement the `load_data` function. In this function, you need to use the h5py module to read in a `.mat` file specified by `file_name`. This file has two dataset objects - `trainxdata` (the input sequences) and `traindata` (the output labels). The data are stored with the batch_size as the last dimension instead of the first dimension. - Load in the first 30000 sequences and their corresponding labels - Transpose the axes so that the input sequences are of shape (30000, 1000, 4) instead of (1000, 4, 30000). Note - use np.transpose, and **not** np.reshape for this. - Transpose the axes for the labels as well so that they're of shape (30000, 919). - Use the `split_sizes` parameter to create train, val, and test splits. Use simple numpy slicing for this. - return the split data. Note - Please do not shuffle or re-order the data.

[4]:
```
def load_data(file_name, split_sizes):
    """Load in data from a .mat file and split it into train, val, test. The␣
 ↪data is
    transposed so that the first dimension is the batch size.

    Parameters:
    file_name (str): Path to the .mat file
    split_sizes (List[int]): A list of 3 integers which specifies the sizes of␣
 ↪the
                             train, val, and test splits.

    Returns:
    train_x (np.array) : The training sequences of shape (N1, 1000, 4)
    train_y (np.array) : The training labels of shape (N1, 919)

    val_x (np.array) : The validation sequences of shape (N2, 1000, 4)
```

```
    val_y (np.array) : The validation labels of shape (N2, 919)

    test_x (np.array) : The test sequences of shape (N3, 1000, 4)
    test_y (np.array) : The test labels of shape (N3, 919)
    """
    # FILL IN CODE HERE #
    f = h5py.File(file_name, 'r')
    trainxdata = np.transpose(f['trainxdata'][:, :, :30000], (2, 0, 1))
    traindata = np.transpose(f['traindata'][:, :30000], (1, 0))

    train_x = trainxdata[:split_sizes[0], :, :]
    train_y = traindata[:split_sizes[0], :]
    val_x = trainxdata[split_sizes[0]:split_sizes[1] + split_sizes[0], :, :]
    val_y = traindata[split_sizes[0]:split_sizes[1] + split_sizes[0], :]

    test_x = trainxdata[split_sizes[1] + split_sizes[0]:, :, :]
    test_y = traindata[split_sizes[1] + split_sizes[0]:, :]

    # FILL IN CODE HERE #

    return train_x, train_y, val_x, val_y, test_x, test_y

split_sizes = [20000, 5000, 5000]
train_x, train_y, val_x, val_y, test_x, test_y = load_data(os.path.join(ROOT,␣
 ↪"deepsea_train", "train.mat"), split_sizes)


# verify that your data matches the shapes in the docstring
print(train_x.shape, train_y.shape)
print(val_x.shape, val_y.shape)
print(test_x.shape, test_y.shape)
```

```
(20000, 1000, 4) (20000, 919)
(5000, 1000, 4) (5000, 919)
(5000, 1000, 4) (5000, 919)
```

Q1.9 What do the 919 output labels represent? (Hint: read paragraph 3 of the Main section of the paper) Read Supplementary Table 2 of the DeepSEA paper and list any 3 of the 919 labels.

1.2.1 Written answer: The 919 output labels represent a diverse collection of genome-wide chromatin profiles from the Encyclopedia of DNA Elements (ENCODE) and Roadmap Epigenomics Projects, including 690 Transcription Factor (TF) binding profiles for 160 different TF, 125 DNAse 1 hypersensitivity sites (DHS) profiles, and 104 histone-mark profiles. Three of the 919 labels are AoSMC-DNase, Chorion-DNase, and Fibrobl-DNase. Chromatin profiling can be used to identify chromatin accesibility and candidate regulatory genomic regions (since active regulatory DNA elements are generally accessible). For example, the histone-mark predictions estimate chemical modifications (such as the addition of an acetyl group) to histone proteins, which affect the tightness of chromatin structure.

Q1.10 Defining the DeepSEA model: Implement the model as described on the last page in the supplementary information. You may want to look at Fig 1. in the paper to get a high level overview.

Layers: 1. Input layer which specifies the shape for a single one-hot encoded sequence. 2. Conv1D layer - 320 kernels and window size 8 3. MaxPooling1D layer - window and step size of 4 4. Dropout layer - 20% dropout 5. Conv1D layer - 480 kernels and window size 8 6. MaxPooling1D layer - window and step size of 4 7. Dropout layer - 20% dropout 8. Conv1D layer - 960 kernels and window size 8 9. Dropout layer - 50% dropout 10. Flatten layer 11. Dense layer - 919 units with sigmoid activation.

Subtle differences - Our model has 919 units in the final fully connected layer while the one described in the supplementary notes has 925. We also add a Reshape layer (this has been done for you) to enable weighting of the individual outputs - more on that later.

Regularization and constraints - Read the "Methods: Training of the DeepSEA model" section of the paper. The loss function has 3 regularization terms and constraints - you must add these to your layers. The necessay functions have been imported for you. The $\lambda_1, \lambda_2, \lambda_3$ parameters are on the last page of the supplementary note.

Your final model should have 51,686,839 total parameters and the final output shape should be `(None, 919, 1)`.

[5]:
```
from tensorflow.keras.layers import Input, Conv1D, MaxPooling1D, Dropout,␣
 ↪Dense, Flatten, Reshape
from tensorflow.keras.regularizers import l1, l2
from tensorflow.keras.constraints import max_norm

model = tf.keras.Sequential([
    # FILL IN CODE HERE #
    Input(shape=(1000, 4)),
    Conv1D(320, 8, kernel_regularizer = l2(5e-07), kernel_constraint =␣
 ↪max_norm(0.9)),
    MaxPooling1D(pool_size=4, strides = 4),
    Dropout(0.2),

    Conv1D(480, 8, kernel_regularizer = l2(5e-07), kernel_constraint =␣
 ↪max_norm(0.9)),
```

```
    MaxPooling1D(pool_size=4, strides = 4),
    Dropout(0.2),

    Conv1D(960, 8, kernel_regularizer = l2(5e-07), kernel_constraint =␣
 ↪max_norm(0.9)),
    Dropout(0.5),

    Flatten(),
    Dense(919, activation='sigmoid', kernel_regularizer = l2(5e-07),␣
 ↪activity_regularizer = l1(1e-08),
          kernel_constraint = max_norm(0.9)),

    # FILL IN CODE HERE #
    Reshape((-1, 1))
])

learning_rate = 1e-3

model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate),
    loss="bce",
    metrics=tf.keras.metrics.BinaryAccuracy()
)
#model.build(train_x.shape)
#model.summary()
```

```
2022-11-11 20:12:59.507899: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-11 20:12:59.517690: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-11 20:12:59.519556: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-11 20:12:59.524138: I tensorflow/core/platform/cpu_feature_guard.cc:142]
This TensorFlow binary is optimized with oneAPI Deep Neural Network Library
(oneDNN) to use the following CPU instructions in performance-critical
operations:  AVX2 AVX512F FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2022-11-11 20:12:59.525203: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
```

```
node, so returning NUMA node zero
2022-11-11 20:12:59.526963: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-11 20:12:59.528697: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-11 20:13:00.107185: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-11 20:13:00.109203: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-11 20:13:00.110911: I
tensorflow/stream_executor/cuda/cuda_gpu_executor.cc:937] successful NUMA node
read from SysFS had negative value (-1), but there must be at least one NUMA
node, so returning NUMA node zero
2022-11-11 20:13:00.112529: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 13642 MB memory:  -> device:
0, name: Tesla T4, pci bus id: 0000:00:04.0, compute capability: 7.5
```

Q1.11 The code below prints out the fraction of 0s in the labels. As you can see, the dataset is very imbalanced and we need to weight the loss function so that incorrect predictions for positive epigenetic features are weighted more. We will use the variables `train_weight` and `val_weight` to weight the loss function. Fill in the parameters to `model.fit` and use these variables to specify the sample weights for the training and validation data. You may use a batch size of 256. You should get an accuracy of at least 75% on the validation set.

[6]:
```python
pos_weight = 1 - train_y.sum() / train_y.size
print(f"Weight: {pos_weight:.6f}")

train_weight = np.where(train_y == 1, pos_weight, 1 - pos_weight)
val_weight = np.where(val_y == 1, pos_weight, 1 - pos_weight)

train_y_exp = np.expand_dims(train_y, -1)
val_y_exp = np.expand_dims(val_y, -1)
```

```
Weight: 0.974675
```

[7]:
```python
epochs = 10

model.fit(
    # FILL IN CODE HERE #
```

```
    train_x,
    train_y_exp,
    batch_size = 256,
    epochs = epochs,
    validation_data = (val_x, val_y_exp, val_weight),
    sample_weight = train_weight
    # FILL IN CODE HERE #
)
```

```
2022-11-11 20:13:03.536289: I
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR
Optimization Passes are enabled (registered 2)

Epoch 1/10

2022-11-11 20:13:05.061296: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369]
Loaded cuDNN version 8005

79/79 [==============================] - 25s 259ms/step - loss: 0.0301 -
binary_accuracy: 0.6982 - val_loss: 0.0275 - val_binary_accuracy: 0.6901
Epoch 2/10
79/79 [==============================] - 19s 247ms/step - loss: 0.0256 -
binary_accuracy: 0.7301 - val_loss: 0.0269 - val_binary_accuracy: 0.6922
Epoch 3/10
79/79 [==============================] - 19s 246ms/step - loss: 0.0247 -
binary_accuracy: 0.7446 - val_loss: 0.0274 - val_binary_accuracy: 0.6523
Epoch 4/10
79/79 [==============================] - 19s 241ms/step - loss: 0.0242 -
binary_accuracy: 0.7532 - val_loss: 0.0267 - val_binary_accuracy: 0.7643
Epoch 5/10
79/79 [==============================] - 19s 240ms/step - loss: 0.0234 -
binary_accuracy: 0.7666 - val_loss: 0.0261 - val_binary_accuracy: 0.7459
Epoch 6/10
79/79 [==============================] - 19s 242ms/step - loss: 0.0222 -
binary_accuracy: 0.7846 - val_loss: 0.0274 - val_binary_accuracy: 0.8162
Epoch 7/10
79/79 [==============================] - 19s 243ms/step - loss: 0.0217 -
binary_accuracy: 0.7936 - val_loss: 0.0277 - val_binary_accuracy: 0.8144
Epoch 8/10
79/79 [==============================] - 19s 243ms/step - loss: 0.0205 -
binary_accuracy: 0.8097 - val_loss: 0.0289 - val_binary_accuracy: 0.8472
Epoch 9/10
79/79 [==============================] - 19s 242ms/step - loss: 0.0196 -
binary_accuracy: 0.8199 - val_loss: 0.0275 - val_binary_accuracy: 0.7720
Epoch 10/10
79/79 [==============================] - 19s 242ms/step - loss: 0.0188 -
binary_accuracy: 0.8311 - val_loss: 0.0291 - val_binary_accuracy: 0.8382
```

[7]: <keras.callbacks.History at 0x7f8bc0202650>

Let's evaluate the model on the test data. You should get an accuracy of at least 75%

[8]:
```
test_weight = np.where(test_y == 1, 1 - pos_weight, pos_weight)
test_y_exp = np.expand_dims(test_y, -1)

print(model.evaluate(test_x, test_y_exp, sample_weight=test_weight))
```

```
157/157 [==============================] - 2s 13ms/step - loss: 0.3379 -
binary_accuracy: 0.8417
[0.33790823817253113, 0.8417347073554993]
```

Let's apply this model to study the breast cancer risk locus SNP rs4784227. This mutation from C>T causes a change in the binding affinity of the FOXA1 transcription factor.

[37]:
```
ref_seq =␣
  ↪"GGGCTCAAGCAGTCCTCCCATCTAGGCTTCCCAAAATGCTGGGATTACAGACATGAGCCACTGCACCCAGCCACAAAGATAACCTAAAGA
alt_seq = list(ref_seq)
alt_seq[500] = 'T'
alt_seq = "".join(alt_seq)

print(ref_seq[497:504])
print(alt_seq[497:504])
```

```
TGCCGAT
TGCTGAT
```

Q1.12 To run our model on the reference and mutated (alt) sequence, we need to one-hot encode the inputs. Implement the `one_hot_encode` function using the provided `nucleotide_map`. The map specifies which of the 4 channels should be 1 for each nucleotide, the other 3 channels should be 0.

[10]:
```
nucleotide_map = {'A': 0, 'G': 1, 'C': 2, 'T': 3}

def one_hot_encode(sequence):
    """
    One-hot encodes a DNA sequence using the provided nucleotide map

    Parameters:
    sequence (str): A DNA string of length L

    Returns:
    vec (np.array): A one-hot encoded array of shape (L, 4) and dtype=np.uint8.
    """
    # FILL IN CODE HERE #
    vec = []
    for amino in sequence:
        arr = [0, 0, 0, 0]
        arr[nucleotide_map[amino]] = 1
        vec.append(arr)
```

```
    vec = np.array(vec)

    # FILL IN CODE HERE #

    return vec.astype(np.uint8)

ref_vec = one_hot_encode(ref_seq)
alt_vec = one_hot_encode(alt_seq)
inp_vec = np.stack([ref_vec, alt_vec])

"""
(2, 1000, 4)
[[0 1 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 1 0]
 [0 0 0 1]
 [1 0 0 0]]
"""
print(inp_vec.shape)
print(one_hot_encode("GGCCTA"))
```

```
(2, 1000, 4)
[[0 1 0 0]
 [0 1 0 0]
 [0 0 1 0]
 [0 0 1 0]
 [0 0 0 1]
 [1 0 0 0]]
```

Q1.13 Recall that the DeepSEA model predicts 919 chromatin properties. We are interested in FOXA1 binding affinity in T-47D cells. Of the 919 outputs, the one at index 396 corresponds to this value. Read the section in the DeepSEA paper on *in-silico* mutagenesis and calculate the log2 fold change of odds for the FOXA1 feature in the reference and mutated (alt) sequence. You should get a value around 0.15 or greater.

[85]:
```
FOXA1_idx = 396

# FILL IN CODE HERE #

outputs = model.predict(inp_vec)
ref_output = outputs[0][FOXA1_idx]
alt_output = outputs[1][FOXA1_idx]

o1 = np.log2(ref_output / (1 - ref_output))
o2 = np.log2(alt_output / (1 - alt_output))
# FILL IN CODE HERE #
```

```
print(o1 - o2)
```

```
[0.21586275]
```

Q1.14 We will now generate mutation maps to visualize the effects of mutations on FOXA1 binding affinity. The `mutate_seq` function takes in a sequence and creates all possible SNPs mutants of that sequence. Note that for a 1000 nucleotide input, there are 3000 unique SNP variants. For ease of processing, we generate 4000 which include 1000 which are identical to the reference sequence.

For this part, you need to do the following: 1. One-hot encode all the mutated sequences 2. Run inference on all of them. Your output array will be of shape (4000, 919, 1) 3. Squeeze the outputs to shape (4000, 919), and then slice out the predictions for the index corresponding to FOXA1. 4. Reshape the result to shape (1000, 4).

5. Each entry in this array represents the score for each mutation. Use this array to compute the log2 fold change of odds. You may reuse the score from the reference sequence which you had calculated earlier. This is exactly the same as the previous cell, the only difference is that you are computing this value for 4000 mutants instead of a single specific mutant.
6. Store the computed log fold change of odds in a variable called `mut_map`.

[94]:
```
def mutate_seq(seq):
    seq_len = len(seq)
    seq_list = [list(seq) for i in range(seq_len * 4)]

    for i in range(seq_len):
        for nuc, n in nucleotide_map.items():
            seq_list[i * 4 + n][i] = nuc

    return [''.join(seq) for seq in seq_list]

mut_seqs = mutate_seq(ref_seq)  # list of 4000 mutated sequences

# FILL IN CODE HERE #
mut_map = np.zeros((1000, 4))
one_hot_seqs = [one_hot_encode(seq) for seq in mut_seqs]
inp = np.stack(one_hot_seqs)
outputs_alt = np.squeeze(model.predict(inp))
alt_output = outputs_alt[:, FOXA1_idx].reshape(1000, 4)

o1 = np.log2(ref_output / (1 - ref_output))
for i in np.arange(1000):
    for j in np.arange(4):
        o2 = np.log2(alt_output[i, j] / (1 - alt_output[i, j]))
        mut_map[i, j] = o1 - o2

# FILL IN CODE HERE #

print(mut_map.shape)  # (1000, 4)
```

```
(1000, 4)
```

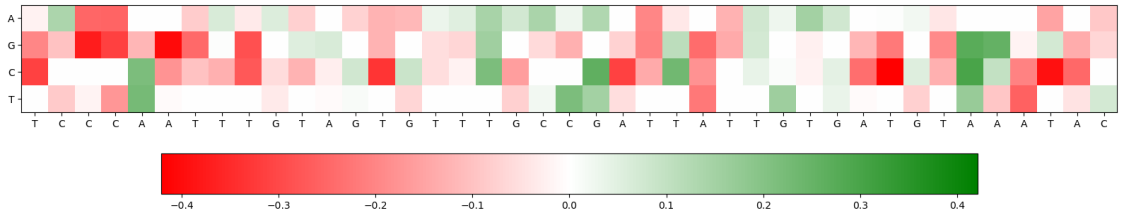
Q1.15 Now we can visualize our mutation map. Run the code below to plot the map. What is the impact of the rs4784227 SNP on FOXA1 binding affinity? Do your results match those in the paper? (It's okay if they don't). If they don't match, explain why not.

1.2.2 Written answer:In the map below, the impact of the rs4784227 snp (C->T) causes an increase (about 0.2 log fold change of odds) in binding affinity for the alternate allele (T) vs the reference allele (C). The results match that of the paper, in which the paper also noted an increase in binding affinity: FOXA1 is preferentially recruited to the rs4784227[T] risk allele over the reference.

[95]:
```
import matplotlib.pyplot as plt
from matplotlib.colors import LinearSegmentedColormap
plt.rcParams["figure.figsize"] = (20, 5)

def plot_map(score_map, seq):
    max_val = np.abs(score_map).max()
    cmap = LinearSegmentedColormap.from_list("rwg", [(0, "red"), (0.5,␣
 ↪"white"), (1, "green")])
    plt.imshow(score_map, cmap=cmap, vmin=-max_val, vmax=max_val)
    plt.colorbar(orientation="horizontal")
    plt.xticks(ticks=np.arange(len(seq)), labels=seq)
    plt.yticks(ticks=np.arange(4), labels="AGCT")


seq = ref_seq[500 - 20: 500 + 21]
score_map = mut_map[500 - 20: 500 + 21]
plot_map(score_map.T, seq)
```



[ ]: