A3_part1_deepsea

November 11, 2021

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]:
```python
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/marchuo/assign3/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 #
     df = pd.read_csv(ROOT + "/IRF4.gff3", header=None, sep='\t')
     df.head()
     # FILL IN CODE HERE #
```

```
[3]:       0       1                2       3       4  5  6  7  \
     0  chr6  HAVANA             gene  391752  411443  .  +  .
     1  chr6  HAVANA       transcript  391752  411443  .  +  .
     2  chr6  HAVANA             exon  391752  391809  .  +  .
     3  chr6  HAVANA  five_prime_UTR  391752  391809  .  +  .
     4  chr6  HAVANA             exon  393098  393368  .  +  .

                                                        8
     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…
```

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.

Written answer: The GRF3 file stores one line per feature, with information such as seqid, source, type, start, end, score, strand, phase, and attributes. The data table is represented as tab-separated and does not explicitly have any column labels. Instead, we cross-reference to the original GTF columns table that states what content is included in each column number. For example, column 3 is feature type.

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?

Written answer: The five_prime_UTR and three_prime_UTR entries represent multi-exon mRNA and other RNA features - they are child five_prime_UTR and three-prime_UTR features of parent transcriptions. They're untranslated regions at the end of the DNA - 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 = 391000
for row in df.itertuples():
    feature = GraphicFeature(start=row[4] - offset, end=row[5] - offset,␣
 ↪strand=1, label = row[3])
    features.append(feature)

# 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?

Written answer: The IRF4 gene contains 8 exons. The introns are located in between the exon regions.

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.

Written answer: CDS is a region of DNA or RNA whose sequence determines the sequence of amino acids in a protein. It overlaps with exons, because "the exon is composed of the coding region as well as the 3' and 5' untranslated regions of the RNA, and so therefore, an exon would be partially made up of CDS." The last CDS doesn't fully overlap with the last exon, because it is after the stop_codon and would not be translated.

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

Written answer: One visible phenotype associated with this gene is skin/hair/eye pigmentation, as the IRF4 gene is involved in regulating production and storage of melanin.

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 = f['trainxdata'][:, :, 0:30000]
         trainxdata = np.transpose(trainxdata, axes=[2, 0, 1])
         traindata = f['traindata'][:, 0:30000]
         traindata = np.transpose(traindata, axes = [1, 0])
         train_x, val_x, test_x = trainxdata[:split_sizes[0],:,:],␣
      ↪trainxdata[split_sizes[0]:split_sizes[1] + split_sizes[0],:, :],␣
      ↪trainxdata[split_sizes[1] + split_sizes[0]:, :, :]
         train_y, val_y, test_y = traindata[:split_sizes[0],:],␣
      ↪traindata[split_sizes[0]:split_sizes[1] + split_sizes[0], :],␣
      ↪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.

Written answer: The 919 output labels represent DeepSEA prediction performance for each transcription factor, DNase I hypersensitive site, and histone mark profile. Thus, these 919 output labels represent the 919 chromatin factors. Three labels include: 8988T, AoSMC, Chorion.

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, 1),
         MaxPooling1D(4, 4),
         Dropout(0.2),
         Conv1D(480, 8),
         MaxPooling1D(4, 4),
         Dropout(0.2),
         Conv1D(960, 8),
         Dropout(0.5),
         Flatten(),
         Dense(919, activation='sigmoid'),
         # 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.summary()
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #   
=================================================================
conv1d (Conv1D)              (None, 993, 320)          10560     
_________________________________________________________________
max_pooling1d (MaxPooling1D) (None, 248, 320)          0         
_________________________________________________________________
dropout (Dropout)            (None, 248, 320)          0         
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 241, 480)          1229280   
_________________________________________________________________
max_pooling1d_1 (MaxPooling1 (None, 60, 480)           0         
_________________________________________________________________
dropout_1 (Dropout)          (None, 60, 480)           0         
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 53, 960)           3687360   
_________________________________________________________________
dropout_2 (Dropout)          (None, 53, 960)           0         
_________________________________________________________________
flatten (Flatten)            (None, 50880)             0         
_________________________________________________________________
dense (Dense)                (None, 919)               46759639  
_________________________________________________________________
reshape (Reshape)            (None, 919, 1)            0         
=================================================================
Total params: 51,686,839
Trainable params: 51,686,839
Non-trainable params: 0
----------------------------------------------------------------
2021-11-11 05:35:21.510912: 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
2021-11-11 05:35:21.613896: 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
2021-11-11 05:35:21.614734: 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
2021-11-11 05:35:21.617502: 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 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2021-11-11 05:35:21.617859: 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
2021-11-11 05:35:21.618790: 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
2021-11-11 05:35:21.619630: 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
2021-11-11 05:35:23.660103: 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
2021-11-11 05:35:23.661022: 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
2021-11-11 05:35:23.661892: 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
2021-11-11 05:35:23.663485: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 10819 MB memory:  -> device:
0, name: Tesla K80, pci bus id: 0000:00:04.0, compute capability: 3.7
```

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]: 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]: epochs = 10

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

```
2021-11-11 05:35:29.242242: I
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR
Optimization Passes are enabled (registered 2)

Epoch 1/10

2021-11-11 05:35:30.813441: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369]
Loaded cuDNN version 8005

79/79 [==============================] - 51s 476ms/step - loss: 0.0276 -
binary_accuracy: 0.7004 - val_loss: 0.0271 - val_binary_accuracy: 0.7768
Epoch 2/10
79/79 [==============================] - 36s 460ms/step - loss: 0.0243 -
binary_accuracy: 0.7389 - val_loss: 0.0277 - val_binary_accuracy: 0.8157
Epoch 3/10
79/79 [==============================] - 37s 467ms/step - loss: 0.0230 -
binary_accuracy: 0.7562 - val_loss: 0.0263 - val_binary_accuracy: 0.7169
Epoch 4/10
79/79 [==============================] - 37s 466ms/step - loss: 0.0208 -
binary_accuracy: 0.7815 - val_loss: 0.0383 - val_binary_accuracy: 0.9003
Epoch 5/10
79/79 [==============================] - 37s 464ms/step - loss: 0.0175 -
```

```
binary_accuracy: 0.8206 - val_loss: 0.0326 - val_binary_accuracy: 0.8697
Epoch 6/10
79/79 [==============================] - 37s 463ms/step - loss: 0.0135 -
binary_accuracy: 0.8636 - val_loss: 0.0415 - val_binary_accuracy: 0.8803
Epoch 7/10
79/79 [==============================] - 36s 462ms/step - loss: 0.0104 -
binary_accuracy: 0.8958 - val_loss: 0.0580 - val_binary_accuracy: 0.9207
Epoch 8/10
79/79 [==============================] - 37s 464ms/step - loss: 0.0083 -
binary_accuracy: 0.9187 - val_loss: 0.0599 - val_binary_accuracy: 0.9222
Epoch 9/10
79/79 [==============================] - 37s 464ms/step - loss: 0.0069 -
binary_accuracy: 0.9343 - val_loss: 0.0604 - val_binary_accuracy: 0.9156
Epoch 10/10
79/79 [==============================] - 37s 468ms/step - loss: 0.0064 -
binary_accuracy: 0.9407 - val_loss: 0.0769 - val_binary_accuracy: 0.9327
```

```
[7]: <keras.callbacks.History at 0x7f2a41d070d0>
```

Let's evaluate the model on the test data. You should get an accuracy of at least 77%

```
[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 [==============================] - 4s 26ms/step - loss: 0.1183 -
binary_accuracy: 0.9346
[0.11827224493026733, 0.934610903263092]
```

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.

```
[9]: 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 = np.zeros((len(sequence), 4))
          seq_arr = np.array(list(sequence))
          for i in range(seq_arr.shape[0]):
              char = seq_arr[i]
              n = nucleotide_map.get(char)
              encode = np.zeros(4)
              encode[n] = 1
              vec[i] = encode

          # 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.

[14]:
```
FOXA1_idx = 396
# FILL IN CODE HERE #
prediction = model.predict(inp_vec)
ref_pred = prediction[0, :, :]
alt_pred = prediction[1, :, :]
foxa_ref_pred = ref_pred[FOXA1_idx, :]
foxa_alt_pred = alt_pred[FOXA1_idx, :]
o1 = np.log2(foxa_ref_pred/(1-foxa_ref_pred))
o2 = np.log2(foxa_alt_pred/(1-foxa_alt_pred))
# FILL IN CODE HERE #

print(o1 - o2)
```

```
[-0.3080082]
```

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`.

[15]:
```
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 #
o1 = np.log2(foxa_ref_pred/(1-foxa_ref_pred))

encoded_seqs = []
for i, seq in enumerate(mut_seqs):
    encoded_seq = one_hot_encode(seq)
    encoded_seqs.append(encoded_seq)
encoded_seqs = np.stack(encoded_seqs)
prediction = model.predict(encoded_seqs)
prediction = np.squeeze(prediction)
foxa_pred = prediction[:, FOXA1_idx]
foxa_pred_reshape = np.reshape(foxa_pred, (1000, 4))

mut_map = o1 - np.log2(foxa_pred_reshape / (1 - foxa_pred_reshape))

# FILL IN CODE HERE #

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

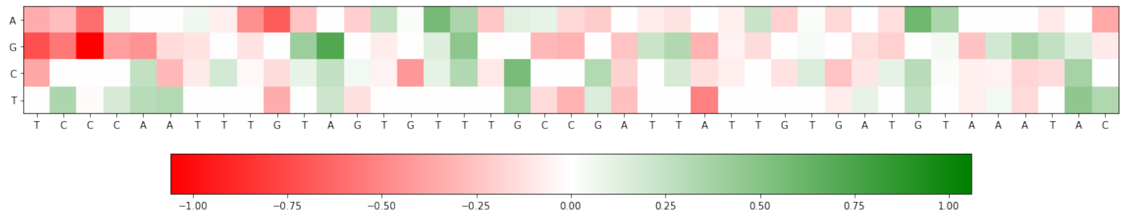
```
(1000, 4)
```

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.

Written answer: It does not match the paper because it decreases the binding affinity at the 2nd C nucleotide of the original reference sequence as shown on the map. However, the paper lists that the SNP increases the binding affinity at the 2nd C nucleotide.

```
[16]: 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)
```



```
[ ]:
```

A3_part2_bpnet

November 11, 2021

1 A3 Part 2: BPNet

BPNet is a model that uses DNA sequences to predict base-resolution binding profiles. It learns complex patterns from the input that are predictive of binding profiles through convolutional layers. In this part of the assignment, we will be using BPNet to predict read coverage profile at base-resolution from input nucleotide sequence. The coverage tracks can come from any genome-wide functional genomics assay that has a sufficient spatial resolution, from the ChIP-seq that we saw in class, to others including ChIP-nexus, ChIP-exo, DNase-seq, and ATAC-seq. In this assignment, we'll see BPNet predict ChIP-nexus signal at base-resolution. Our implementation will use experimental ChIP-nexus data for the transcription factor Oct4 which influences pluripotency in embryonic stem cells. Read more about BPNet here - `https://www.nature.com/articles/s41588-021-00782-6`.

1.1 Section 1: Data

We'll be training BPNet on ChIP-nexus data for the Oct4 transcription factor in mouse embryonic stem cells (mESC). The raw data are available here and you can read more about the experimental process used to generate the data files we'll be using for this assignment. In part 1, you created a directory named `data`. Create a new directory `bpnet` inside `data` and run the following commands. Note - **The wget command used to download files is one very long command and may be split over a couple of lines.**

1. We get the locations of the Oct4 peaks. Oct4 binds to these regions in the genome.

```
wget ftp://ftp.ncbi.nlm.nih.gov/geo/samples/GSM4072nnn/GSM4072776/suppl/GSM4072776%5Fmesc%5Foc
-O peaks.bed.gz
```

2. We get the read counts for the positive strand.

```
wget ftp://ftp.ncbi.nlm.nih.gov/geo/samples/GSM4072nnn/GSM4072776/suppl/GSM4072776%5Fmesc%5Foc
-O pos.bw
```

3. We get the read counts for the negative strand.

```
wget ftp://ftp.ncbi.nlm.nih.gov/geo/samples/GSM4072nnn/GSM4072776/suppl/GSM4072776%5Fmesc%5Foc
-O neg.bw
```

4. Finally, we download the mouse genome. This takes about 30 mins to download.

```
wget http://ftp.ebi.ac.uk/pub/databases/gencode/Gencode_mouse/release_M23/GRCm38.primary_assem
```

We unzip our data with `gunzip peaks.bed.gz` and `GRCm38.primary_assembly.genome.fa.gz`. This should result in 4 files in your `bpnet` folder - `peaks.bed` - `pos.bw` - `neg.bw` -

GRCm38.primary_assembly.genome.fa.

Finally, we will install a couple of python packages to help us parse these files with `pip install pyfaidx pyBigWig`

[1]:
```
import os
import numpy as np
import pandas as pd
import tensorflow as tf
import tensorflow_probability as tfp  # this should be installed by default
import matplotlib.pyplot as plt
plt.rcParams["figure.figsize"] = (20, 5)

import pyBigWig
import pyfaidx
```

Note that the mouse genome is rather large (2.6G). We use `pyfaidx` for efficient access to the chromosome sequences. It creates an index file the first time it reads a new fasta file. On subsequent calls, it uses this index for a fast dictionary like lookup. We use `pyBigWig` to load BigWig files.

Q2.1 Genomics data is stored in several different file formats. Explain what kinds of data are stored in fasta files, BigWig files, and bed files. Your answer should not be specific to this assignment.

Written answer:

- fasta: FASTA is a file format used to store nucleotide and amino acid polymeric sequences, where nucleotides or amino acids are represented as single-letter codes.
- BigWig: BigWig files are indexed binary files which contain alignment data to the genome - it contains the location of the genome where the read is aligned to.
- bed: Bed files format is a "text file format used to store genomic regions as coordinates and associated annotations". It contains information defining basic sequence features to a sequence and consists of one line per feature.

[2]:
```
# Set ROOT to be the path to the `bpnet` directory
ROOT = "/home/marchuo/assign3/data/bpnet"
fasta_ref = pyfaidx.Fasta(os.path.join(ROOT, "GRCm38.primary_assembly.genome.
↪fa"))

pos_counts = pyBigWig.open(os.path.join(ROOT, "pos.bw"))
neg_counts = pyBigWig.open(os.path.join(ROOT, "neg.bw"))
```

Q2.2 BPNet is trained on 1000 base pair long sequences centered around transcription factor (TF) binding peaks. You may want to read about Peak calling and ChIP-seq. We split the peaks defined in `peaks.bed` into train and validation sets - We use all peaks from chromosome 1 for validation, and the rest for training. Note that `peaks.bed` is a tab separated file with 3 columns, but we only need the first two.

Fill in the `load_peaks` function below and return two data frames - one for training and one for validation.

```
[3]: def load_peaks():
         """
         Loads peaks as a dataframe and splits them into two separate dataframes.
         `val_df` has peaks from chr1, the rest go to `train_df`

         Returns:
         train_df (pd.DataFrame): a dataframe with two columns - chromosome number␣
     ↪and peak location
         val_df (pd.DataFrame): a dataframe with two columns - chromosome number and␣
     ↪peak location
         """
         peaks_file = os.path.join(ROOT, "peaks.bed")
         columns = ["chr_no", "peak"]  # only read in the first two columns

         # FILL IN CODE HERE #
         peaks_df = pd.read_csv(peaks_file, sep='\t', header=None, usecols=[0, 1])
         peaks_df.columns = columns
         train_df = peaks_df[:24114]
         val_df = peaks_df[24114:]
         # FILL IN CODE HERE #

         return train_df, val_df

     train_df, val_df = load_peaks()
     print(train_df.shape, val_df.shape)  # (24114, 2) (1735, 2)
     display(train_df.head())
```

```
(24114, 2) (1735, 2)

  chr_no       peak
0   chrX  143483058
1   chrY    4150218
2   chr9    3002133
3   chr3  122145577
4  chr13   21200261
```

Q2.3 Now that we have our peak locations, we read in the sequences centered at those peaks. Note that we read in sequences from both the positive and negative strand, so the number of sequences we get is twice the number of peaks. We also read in the counts for each nucleotide in these sequences. Fill in the `load_data` function below and implement the following steps. For each peak in the dataframe, you need to: 1. Compute start and end values for the sequence centered at the peak. Since we use 1000bp sequences, start and end are `peak-500` and `peak+500`. 2. Read the 1000bp long DNA string from the mouse genome for the specific chromosome and start / end values using pyfaidx. and append it to `sequences`. Use the `fasta_ref` Fasta file object that we created at the beginning of the notebook. 3. Append the complement (not reverse complement) of this sequence and append it to `sequences`. This is the negative strand sequence. Go through the pyfaidx examples to see how to get the complement. 4. Read in the count values for this sequence from the `pos_counts` BigWig file object and append them to `counts`. You may find the values function to be useful. 5. Similarly, read in the count values from the negative strand from the neg_counts BigWig file object. 6. Note that if counts are not available for a nucleotide, the values function returns nans. Use np.nan_to_num to replace the missing counts with 0s.

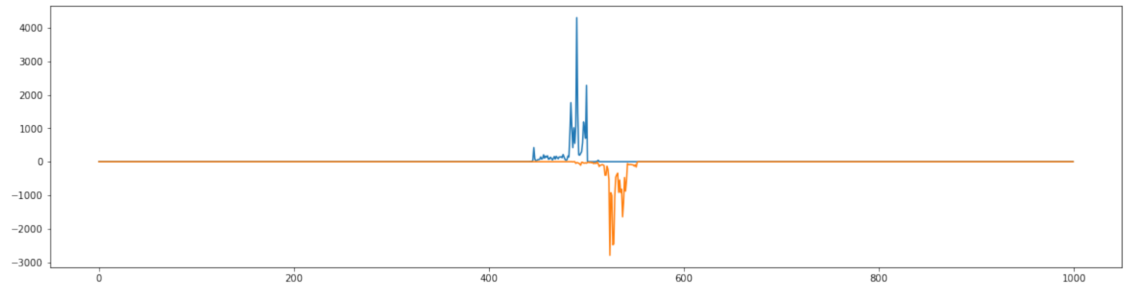
[4]:
```
def load_data(df):
    """
    Loads sequences and count values for 1000bp regions centered around peaks
    specified in df. Uses `fasta_ref` for sequences, `pos_counts` and␣
 ↪`neg_counts` for read counts.

    Parameters:
    df (pd.DataFrame): A data frame containing peak loci (chr_no and peak␣
 ↪position)

    Returns:
    sequences (List[str]): A list of 1000bp long DNA sequences. There are twice␣
 ↪as many
                            sequences as peak (1 each for the positive and␣
 ↪negative strands)
    counts (np.array): An array of shape (2N, 1000) where N is the number of␣
 ↪peaks in df.
    """

    sequences, counts = [], []
    # FILL IN CODE HERE #
    for row in df.itertuples():
        chromosome = row[1]
        peak = row[2]
        start, end = peak - 500, peak + 500
        string = fasta_ref[chromosome][start:end]
        sequences.append(string)
        complement = string.complement
        sequences.append(complement)

        pos = pos_counts.values(chromosome, start, end)
        counts.append(np.nan_to_num(pos))
        neg = neg_counts.values(chromosome, start, end)
        counts.append(np.nan_to_num(neg))

    # FILL IN CODE HERE #

    return sequences, np.stack(counts)

train_sequences, train_counts = load_data(train_df)
val_sequences, val_counts = load_data(val_df)
print(len(train_sequences), len(val_sequences))  # 48228 3470
print(train_counts.shape, val_counts.shape)  # (48228, 1000) (3470, 1000)
```

```
48228 3470
(48228, 1000) (3470, 1000)
```

Let's see what the counts look like for a single sequence. The cell below plots the counts for the positive and negative strands for the first peak. As you can see, ChIP-nexus produces very sharp peaks.

```
[5]: plt.plot(train_counts[0])
     plt.plot(-train_counts[1])
```

```
[5]: [<matplotlib.lines.Line2D at 0x7f5fd4852d90>]
```



Q2.4 We will now convert our sequences and counts into vectors so that they can be used to train our BPNet model. The first step is to implement the `one_hot_encode` function. You may refer to your implementation from part 1, but note that there is a small modification. Our `nucleotide_map` has an extra letter `N` which is often used to represent unknown nucleotides in fasta files. Modify your implementation to account for this difference.

```
[6]: nucleotide_map = {'A': 0, 'G': 1, 'C': 2, 'T': 3, 'N': 4}  # note N

     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, 5) and dtype=np.uint8.
         """
         # FILL IN CODE HERE #
         vec = np.zeros((len(sequence), 5))
         for i, char in enumerate(sequence):
             if char not in ['A', 'G', 'C', 'T']:
                 char = 'N'
             n = nucleotide_map.get(str(char))
             encode = np.zeros(5)
```

```
        encode[n] = 1
        vec[i, :] = encode
    # FILL IN CODE HERE #

    return vec.astype(np.float32)
```

Q2.5 We can now use this modified implmentation of `one_hot_encode` to vectorize our input sequences. The BPNet model has two outputs for each input sequence - a profile shape, and the total read counts. The profile for each sequence is just the counts vector which is normalized by the total number of counts. It can be thought of as a sequence of binding probabilities which sum up to 1 over the span of the sequence. Use the `counts` variable to compute this profile vector. You may find numpy's broadcasting features useful if you sum `counts` over a specific axis.

```
[7]: def vectorize(sequences, counts):
    """
    Encodes the sequences as one-hot vectors and computes a normalized profile␣
 ↪of read counts.

    Parameters:
    sequences (List[str]): A list of N DNA sequences, 1000bp long.
    counts (np.array): An array of read counts of shape (N, 1000)

    Returns:
    seq_vec (np.array): An array of one-hot encoded sequences, shape - (N,␣
 ↪1000, 5)
    profile_vec (np.array): An array of normalized read counts, each row sums␣
 ↪up to 1. shape - (N, 1000)
    counts (np.array): The same as the input parameter, typecast to np.float32
    """

    # FILL IN CODE HERE #
    seq_vec = np.zeros((len(sequences), 1000, 5))
    for i, sequence in enumerate(sequences):
        encoded_seq = one_hot_encode(sequence)
        seq_vec[i, :, :] = encoded_seq
    profile_vec = np.zeros((len(sequences), 1000))
    counts_sum = counts.sum(axis=1).sum(axis=0)
    for i, count in enumerate(counts):
        normalized_array = count / counts_sum
        profile_vec[i] = normalized_array
    # FILL IN CODE HERE #
    return seq_vec.astype(np.float32), profile_vec.astype(np.float32), counts.
 ↪astype(np.float32)

train_seq_vec, train_profile_vec, train_count_vec = vectorize(train_sequences,␣
 ↪train_counts)
```

```
val_seq_vec, val_profile_vec, val_count_vec = vectorize(val_sequences,␣
↪val_counts)

print(train_count_vec.shape, train_seq_vec.shape, train_profile_vec.shape)  #␣
↪(48228, 1000) (48228, 1000, 5) (48228, 1000
print(val_count_vec.shape, val_seq_vec.shape, val_profile_vec.shape)  # (3470,␣
↪1000) (3470, 1000, 5) (3470, 1000)
```

```
(48228, 1000) (48228, 1000, 5) (48228, 1000)
(3470, 1000) (3470, 1000, 5) (3470, 1000)
```

1.2 Section 2: BPNet Model

We'll be implementing a version of BPNet. It's a long paper but you should go through the Methods: BPNet architecture and Methods: BPNet loss function sub-sections very carefully. Our version is simplified - we do not train in a multi-task fashion and we do not use bias tracks as a control.

Q2.6: Describe the two contributions of BPNet (listed below) in detail and explain how they're beneficial for this prediction problem.

Written answer: 1. Dilated convolutions and with residual-style layers - The dilated convolutions are a series of 9 convolution 1d layers each with 64 layers of width 3 with exponential dilation in every layer. The dilation rate is exponential such that number of skipped positions in the convolutional filter double every layer. Dilated convolution layers are effective for simultaneous denoising and peak calling while maintaining the shape of the data, which is especially important as we want to preserve sequence and nucleotide ordering integrity. Dilation in convolution increases the span of filters without increasing the number of weight parameters in them. The dilated convolutions with residual-style layers allows the model to learn increasingly complex predictive sequence patterns. The dilated convolutions are beneficial for this prediction problem because they allow for larger receptive fields (important for reading long bp sequences) since they can capture a hierarchical representation of a much larger input space than standard convolutions, allowing them to scale to large context sizes, which is necessary for our prediction task of long bp sequences. Additionally, dilated convolutions' prediction of output depends on the previous inputs while maintaining order of data. 2. Profile regression loss - Profile regression loss has two separate predictions for the profile shape and total read counts across the profile. According to the paper, "Binding profiles can be decomposed into the total signal (read counts) and the profile shape (base-resolution distribution of reads)". Thus, profile regression loss is beneficial to provide direct prediction of the raw base-resolution binding profiles from DNA sequence by separating the two predictions into the profile shape and total read counts across the profile. It incorporates multinomial negative log-likelihood loss for the profile shapes, and mean squared error for the log normalized counts for effective model training

Q2.7 One of BPNet's novel contributions is the Profile regression loss which involves separate predictions for the profile shape and total read counts across the profile. We will first implement a custom layer for BPNet which takes in the intermediate representation from the convolutional layers and decodes it into the two outputs. The input to this layer will be of shape (`batch_size, 1000, channels`).

You need to specify 3 layers: 1. A Conv1D layer with 1 filter of size 25 with padding. 2. A

GlobalAveragePooling1D layer 3. A Dense layer with 1 unit, no activation function

This custom layer applies the `Conv1D` layer to its input tensor to compute the profile shape. It also applies the pooling layer followed by the `Dense` layer on the same input tensor to predict the total read counts. Note that we have added a `Flatten` layer to the output of the `Conv1D` to make it easier to implement the custom loss function (described later).

[8]:
```
from tensorflow.keras.layers import Conv1D, GlobalAveragePooling1D, Dense,␣
↪Flatten

class BPNetOutputLayer(tf.keras.layers.Layer):
    def __init__(self):
        super(BPNetOutputLayer, self).__init__()

        # FILL IN CODE HERE #
        self.conv1d = Conv1D(1, 25, padding='same')
        self.pool1d = GlobalAveragePooling1D()
        self.dense = Dense(1)
        # FILL IN CODE HERE #
        self.flatten_layer = Flatten()

    def call(self, inputs):
        """
        x is the output of the Conv1D layer - the profile shape.
        y is the output of the dense layer - the read count.
        """

        # FILL IN CODE HERE #
        x = self.conv1d(inputs)
        y = self.pool1d(inputs)
        y = self.dense(y)
        # FILL IN CODE HERE #

        return self.flatten_layer(x), y

layer = BPNetOutputLayer()
inputs = np.ones((10, 1000, 128))
x, y = layer(inputs)
x, y = x.numpy(), y.numpy()

print(x.shape, y.shape)  # (10, 1000) (10, 1)
```

```
2021-11-11 07:34:02.868153: 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
2021-11-11 07:34:02.879725: 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
2021-11-11 07:34:02.880567: 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
2021-11-11 07:34:02.882458: 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 FMA
To enable them in other operations, rebuild TensorFlow with the appropriate
compiler flags.
2021-11-11 07:34:02.882735: 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
2021-11-11 07:34:02.883621: 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
2021-11-11 07:34:02.884440: 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
2021-11-11 07:34:03.266305: 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
2021-11-11 07:34:03.267459: 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
2021-11-11 07:34:03.268320: 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
2021-11-11 07:34:03.269078: I
tensorflow/core/common_runtime/gpu/gpu_device.cc:1510] Created device
/job:localhost/replica:0/task:0/device:GPU:0 with 10819 MB memory:  -> device:
0, name: Tesla K80, pci bus id: 0000:00:04.0, compute capability: 3.7
2021-11-11 07:34:03.595184: I tensorflow/stream_executor/cuda/cuda_dnn.cc:369]
Loaded cuDNN version 8005
```

```
(10, 1000) (10, 1)
```

Q2.8 Now we can implement the main body of the BPNet model and use this custom layer for the final output. Your BPNet model needs to implemented as described below: 1. The first layer is a Conv1D with 64 filters of size 25. 2. The next 9 layers are dilated Conv1D layers connected with **residual skip connections** and an exponential dilation rate. So the first one has dilation rate 2, next one 4, and so on upto 512. 3. The custom output layer which you implemented in the previous section.

Each conv layer should be padded to preserve sequence length and should have relu activations. If you implement the model as described, you should have 120,898 parameters.

[9]:
```python
from tensorflow.keras.layers import Input, Conv1D, Add


class BPNetModel(tf.keras.Model):
    def __init__(self):
        super(BPNetModel, self).__init__()

        # FILL IN CODE HERE #
        self.conv1d = Conv1D(64, 25, padding='same', activation='relu')
        n = 2
        self.conv1d_1 = Conv1D(64, 3, dilation_rate=n**1, padding='same',␣
↪activation='relu')
        self.conv1d_2 = Conv1D(64, 3, dilation_rate=n**2, padding='same',␣
↪activation='relu')
        self.conv1d_3 = Conv1D(64, 3, dilation_rate=n**3, padding='same',␣
↪activation='relu')
        self.conv1d_4 = Conv1D(64, 3, dilation_rate=n**4, padding='same',␣
↪activation='relu')
        self.conv1d_5 = Conv1D(64, 3, dilation_rate=n**5, padding='same',␣
↪activation='relu')
        self.conv1d_6 = Conv1D(64, 3, dilation_rate=n**6, padding='same',␣
↪activation='relu')
        self.conv1d_7 = Conv1D(64, 3, dilation_rate=n**7, padding='same',␣
↪activation='relu')
        self.conv1d_8 = Conv1D(64, 3, dilation_rate=n**8, padding='same',␣
↪activation='relu')
        self.conv1d_9 = Conv1D(64, 3, dilation_rate=n**9, padding='same',␣
↪activation='relu')
        # FILL IN CODE HERE #

        self.output_layer = BPNetOutputLayer()

    def call(self, inputs):
        # FILL IN CODE HERE #
        x = self.conv1d(inputs)
        conv1d_1 = self.conv1d_1(x)
        conv1d_2 = self.conv1d_2(conv1d_1)
        conv1d_3 = self.conv1d_3(conv1d_2)
        conv1d_4 = self.conv1d_4(conv1d_3)
        conv1d_5 = self.conv1d_5(conv1d_4)
        conv1d_6 = self.conv1d_6(conv1d_5)
        conv1d_7 = self.conv1d_7(conv1d_6)
        conv1d_8 = self.conv1d_8(conv1d_7)
        conv1d_9 = self.conv1d_9(conv1d_8)

        x = Add()([conv1d_1, conv1d_2, conv1d_3, conv1d_4, conv1d_5, conv1d_6,␣
↪conv1d_7, conv1d_8, conv1d_9])

        x = self.output_layer(x)

        return x
        # FILL IN CODE HERE #

    def model(self):  # because tf 2.6 does weird stuff with shapes in summary
        x = Input(shape=(1000, 5))
        return tf.keras.Model(inputs=x, outputs=self.call(x))

BPNetModel().model().summary()
```

```
Model: "model"
________________________________________________________________________________
__________________
Layer (type)                    Output Shape         Param #     Connected to
================================================================================
==================
input_1 (InputLayer)            [(None, 1000, 5)]    0

________________________________________________________________________________
__________________
conv1d_1 (Conv1D)               (None, 1000, 64)     8064        input_1[0][0]

________________________________________________________________________________
__________________
conv1d_2 (Conv1D)               (None, 1000, 64)     12352       conv1d_1[0][0]

________________________________________________________________________________
__________________
conv1d_3 (Conv1D)               (None, 1000, 64)     12352       conv1d_2[0][0]

________________________________________________________________________________
__________________
conv1d_4 (Conv1D)               (None, 1000, 64)     12352       conv1d_3[0][0]

________________________________________________________________________________
__________________
conv1d_5 (Conv1D)               (None, 1000, 64)     12352       conv1d_4[0][0]

________________________________________________________________________________
__________________
conv1d_6 (Conv1D)               (None, 1000, 64)     12352       conv1d_5[0][0]

________________________________________________________________________________
__________________
conv1d_7 (Conv1D)               (None, 1000, 64)     12352       conv1d_6[0][0]

________________________________________________________________________________
```

```
__________________
conv1d_8 (Conv1D)               (None, 1000, 64)     12352       conv1d_7[0][0]
________________________________________________________________________________
__________________
conv1d_9 (Conv1D)               (None, 1000, 64)     12352       conv1d_8[0][0]
________________________________________________________________________________
__________________
conv1d_10 (Conv1D)              (None, 1000, 64)     12352       conv1d_9[0][0]
________________________________________________________________________________
__________________
add (Add)                       (None, 1000, 64)     0           conv1d_2[0][0]
                                                                 conv1d_3[0][0]
                                                                 conv1d_4[0][0]
                                                                 conv1d_5[0][0]
                                                                 conv1d_6[0][0]
                                                                 conv1d_7[0][0]
                                                                 conv1d_8[0][0]
                                                                 conv1d_9[0][0]
                                                                 conv1d_10[0][0]
________________________________________________________________________________
__________________
bp_net_output_layer_1 (BPNetOut ((None, 1000), (None 1666        add[0][0]
================================================================================
==================
Total params: 120,898
Trainable params: 120,898
Non-trainable params: 0
________________________________________________________________________________
__________________
```

Q2.9 We will implement a custom loss function to compute the multinomial negative log-likelihood. Read the Methods: BPNet loss function sub-section of the BPNet paper to understand how this loss term is computed. We will implement it using TensorFlow's Multinomial distribution. You may find the log_prob function useful.

Hints : - true_counts is the array of true read counts for each nucleotide, shape is (batch_size, 1000) - pred_profile is the predicted profile from BPNet (the first output). shape is (batch_size, 1000) - The Multinomial distribution requires two params - an array of total counts for each sequence, shape (batch_size,) - logits - these are predicted profiles from BPNet - After the distribution is created, use the `log_prob` function to compute the log likelihood. Flip the sign to compute the negative log-likelihood. Return the mean of this result. - All operations should be implemented using TensorFlow functions, not NumPy functions.

[10]:
```python
import tensorflow_probability as tfp
def multinomial_nll(true_counts, pred_profile):
    """
    Computes the multinomial negative log-likelihood loss
    Parameters:
    true_counts (tf.Tensor): the true read counts at the nucleotide level,␣
↪shape - (batch_size, 1000)
    pred_profile (tf.Tensor): The predicted profile from BPNet.

    Returns:
    the mean multinomial negative log-likelihood for the batch - a single␣
↪number.
    """
    # FILL IN CODE HERE #
    batch_size = tf.shape(true_counts)[0]
    count = tf.reduce_sum(true_counts, axis=-1)
    dist = tfp.distributions.Multinomial(total_count=count,
                                         logits=pred_profile)
    neg_log_likelihood = -tf.reduce_sum(dist.log_prob(true_counts))
    return neg_log_likelihood / float(batch_size)
    # FILL IN CODE HERE #
```

Q2.10 The second output of BPNet is the total read counts across a sequence. In practice, it is easier to train BPNet by predicting and optimizing the log of the total reads. The loss equation in the methods section of BPNet computes the mean squared error over the log normalized count values. Fill in the `log_normalize` function to compute $log(1 + n_{obs})$ from the count vectors.

[11]:
```
def log_normalize(count_vec):
    """
    Returns log(1 + n_obs) from counts

    Parameters:
    count_vec (np.array): read counts for each nucleotide, shape - (N, 1000)

    Returns:
    log_counts (np.array): log normalized total counts, shape - (N, 1)
    """
    # FILL IN CODE HERE #
    counts = tf.math.reduce_sum(count_vec, axis=1, keepdims=True)
    return tf.math.log(counts + 1)
    # FILL IN CODE HERE #

log_train_counts = log_normalize(train_count_vec)
log_val_counts = log_normalize(val_count_vec)
print(log_train_counts.shape)
```

```
(48228, 1)
```

Q2.11 Let's train our BPNet model. Compile your model with two losses, the multinomial negative log-likelihood loss for the profile shapes, and mean squared error for the log normalized counts. Note that the paper describes a λ parameter to weight the two losses, which we will not use in this assignment. In `model.fit`, you need to specify the training inputs, labels, and validation_data.

You should get train and validation loss values below 500.

[12]:
```
model = BPNetModel()

model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
    # FILL IN CODE HERE #
    loss=[multinomial_nll, "mse"]
    # FILL IN CODE HERE #
)

model.fit(
    # FILL IN CODE HERE #
    x=train_seq_vec,
    y=[train_count_vec, log_train_counts],
    validation_data=(val_seq_vec, [val_count_vec, log_val_counts]),
    # FILL IN CODE HERE #
    epochs=25,
    batch_size=512,
)
```

```
2021-11-11 07:34:05.860703: I
tensorflow/compiler/mlir/mlir_graph_optimization_pass.cc:185] None of the MLIR
Optimization Passes are enabled (registered 2)
```

```
Epoch 1/25
95/95 [==============================] - 103s 941ms/step - loss: 543.7806 -
output_1_loss: 540.5626 - output_2_loss: 3.2183 - val_loss: 405.6424 -
val_output_1_loss: 405.3446 - val_output_2_loss: 0.2978
Epoch 2/25
95/95 [==============================] - 85s 896ms/step - loss: 532.2064 -
output_1_loss: 531.8290 - output_2_loss: 0.3774 - val_loss: 401.7544 -
val_output_1_loss: 401.5166 - val_output_2_loss: 0.2377
Epoch 3/25
95/95 [==============================] - 85s 893ms/step - loss: 531.1072 -
output_1_loss: 530.7111 - output_2_loss: 0.3961 - val_loss: 400.8895 -
val_output_1_loss: 400.6951 - val_output_2_loss: 0.1944
Epoch 4/25
95/95 [==============================] - 85s 894ms/step - loss: 530.1334 -
output_1_loss: 529.6912 - output_2_loss: 0.4420 - val_loss: 400.2418 -
val_output_1_loss: 400.0651 - val_output_2_loss: 0.1768
Epoch 5/25
95/95 [==============================] - 85s 894ms/step - loss: 529.0416 -
output_1_loss: 528.6298 - output_2_loss: 0.4120 - val_loss: 400.5607 -
val_output_1_loss: 400.3870 - val_output_2_loss: 0.1737
Epoch 6/25
95/95 [==============================] - 85s 895ms/step - loss: 527.8184 -
output_1_loss: 527.4006 - output_2_loss: 0.4176 - val_loss: 401.9728 -
```

```
val_output_1_loss: 401.8057 - val_output_2_loss: 0.1671
Epoch 7/25
95/95 [==============================] - 85s 894ms/step - loss: 526.2253 -
output_1_loss: 525.8492 - output_2_loss: 0.3759 - val_loss: 400.5980 -
val_output_1_loss: 400.4537 - val_output_2_loss: 0.1442
Epoch 8/25
95/95 [==============================] - 85s 893ms/step - loss: 527.5927 -
output_1_loss: 527.1755 - output_2_loss: 0.4171 - val_loss: 402.5387 -
val_output_1_loss: 402.2497 - val_output_2_loss: 0.2890
Epoch 9/25
95/95 [==============================] - 85s 894ms/step - loss: 523.5884 -
output_1_loss: 523.2079 - output_2_loss: 0.3804 - val_loss: 401.8756 -
val_output_1_loss: 401.5761 - val_output_2_loss: 0.2995
Epoch 10/25
95/95 [==============================] - 85s 894ms/step - loss: 519.8587 -
output_1_loss: 519.4924 - output_2_loss: 0.3664 - val_loss: 398.5713 -
val_output_1_loss: 398.2943 - val_output_2_loss: 0.2770
Epoch 11/25
95/95 [==============================] - 85s 894ms/step - loss: 517.2834 -
output_1_loss: 516.9150 - output_2_loss: 0.3684 - val_loss: 396.3479 -
val_output_1_loss: 396.0955 - val_output_2_loss: 0.2524
Epoch 12/25
95/95 [==============================] - 85s 894ms/step - loss: 514.7685 -
output_1_loss: 514.4212 - output_2_loss: 0.3473 - val_loss: 396.8598 -
val_output_1_loss: 396.6969 - val_output_2_loss: 0.1629
Epoch 13/25
95/95 [==============================] - 85s 895ms/step - loss: 513.1558 -
output_1_loss: 512.8187 - output_2_loss: 0.3371 - val_loss: 396.8315 -
val_output_1_loss: 396.6481 - val_output_2_loss: 0.1835
Epoch 14/25
95/95 [==============================] - 85s 895ms/step - loss: 511.2585 -
output_1_loss: 510.9399 - output_2_loss: 0.3185 - val_loss: 396.0451 -
val_output_1_loss: 395.7740 - val_output_2_loss: 0.2711
Epoch 15/25
95/95 [==============================] - 85s 894ms/step - loss: 509.6419 -
output_1_loss: 509.3247 - output_2_loss: 0.3172 - val_loss: 396.2455 -
val_output_1_loss: 396.0091 - val_output_2_loss: 0.2364
Epoch 16/25
95/95 [==============================] - 85s 894ms/step - loss: 509.3289 -
output_1_loss: 509.0127 - output_2_loss: 0.3161 - val_loss: 396.2637 -
val_output_1_loss: 395.9730 - val_output_2_loss: 0.2908
Epoch 17/25
95/95 [==============================] - 85s 895ms/step - loss: 507.5512 -
output_1_loss: 507.2477 - output_2_loss: 0.3036 - val_loss: 394.4070 -
val_output_1_loss: 394.1978 - val_output_2_loss: 0.2092
Epoch 18/25
95/95 [==============================] - 85s 895ms/step - loss: 506.2809 -
output_1_loss: 505.9748 - output_2_loss: 0.3060 - val_loss: 395.3143 -
```

```
val_output_1_loss: 394.9777 - val_output_2_loss: 0.3366
Epoch 19/25
95/95 [==============================] - 85s 895ms/step - loss: 507.5119 -
output_1_loss: 507.1909 - output_2_loss: 0.3211 - val_loss: 393.8439 -
val_output_1_loss: 393.6117 - val_output_2_loss: 0.2322
Epoch 20/25
95/95 [==============================] - 85s 895ms/step - loss: 505.2645 -
output_1_loss: 504.9632 - output_2_loss: 0.3012 - val_loss: 391.6941 -
val_output_1_loss: 391.4846 - val_output_2_loss: 0.2095
Epoch 21/25
95/95 [==============================] - 85s 895ms/step - loss: 504.8502 -
output_1_loss: 504.5468 - output_2_loss: 0.3034 - val_loss: 392.0958 -
val_output_1_loss: 391.7576 - val_output_2_loss: 0.3382
Epoch 22/25
95/95 [==============================] - 85s 894ms/step - loss: 505.9093 -
output_1_loss: 505.6031 - output_2_loss: 0.3064 - val_loss: 391.3817 -
val_output_1_loss: 391.2329 - val_output_2_loss: 0.1488
Epoch 23/25
95/95 [==============================] - 85s 894ms/step - loss: 503.0326 -
output_1_loss: 502.7397 - output_2_loss: 0.2928 - val_loss: 391.8702 -
val_output_1_loss: 391.6580 - val_output_2_loss: 0.2122
Epoch 24/25
95/95 [==============================] - 85s 894ms/step - loss: 502.3104 -
output_1_loss: 502.0153 - output_2_loss: 0.2949 - val_loss: 391.7887 -
val_output_1_loss: 391.4886 - val_output_2_loss: 0.3002
Epoch 25/25
95/95 [==============================] - 85s 894ms/step - loss: 502.1483 -
output_1_loss: 501.8546 - output_2_loss: 0.2936 - val_loss: 391.6885 -
val_output_1_loss: 391.3588 - val_output_2_loss: 0.3297
```

```
[12]: <keras.callbacks.History at 0x7f5fd4767490>
```

Q2.12 Inference on Lefty1 enchancer

We will be using our trained model to predict Oct4 binding affinities on the Lefty1 enhancer region. This region is a known binding site for Oct4. The locus of this region is given below. Extract the corresponding positive and negative strand sequences and convert them to one-hot encoded vectors, `seq_vecs`. For this exercise, we will only be looking at the profile shapes and not the read counts.

```
[13]: # Lefty1 enhancer locus #
chr_no = "chr1"
lefty_idx = 180924952
start, end = lefty_idx - 500, lefty_idx + 500

# FILL IN CODE HERE #
pos_strand = fasta_ref[chr_no][start:end]
neg_strand = pos_strand.complement
pos_strand_encoded = one_hot_encode(pos_strand)
```

```
neg_strand_encoded = one_hot_encode(neg_strand)

seq_vecs = np.stack((pos_strand_encoded, neg_strand_encoded))
print(seq_vecs.shape)
# FILL IN CODE HERE #

# seq_vecs should have shape (2, 1000, 5), the first should be from the␣
 ↪positive strand
# and the second should be from the negative strand.

logits, log_counts = model.predict(seq_vecs)
probs = np.exp(logits) / np.exp(logits).sum(axis=-1, keepdims=True)
```

```
(2, 1000, 5)
```

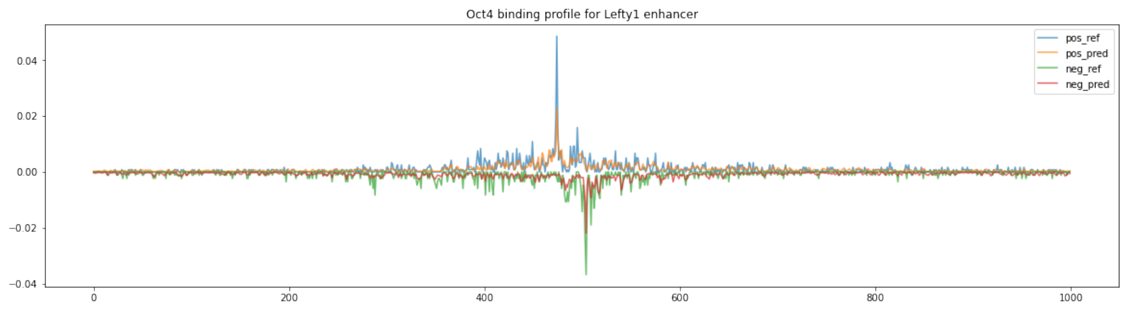
Q2.13 Look at the last line of the previous cell and explain why the probabilities are being calculated this way. Which activation function is being applied to the logits?

Written answer: The probabilities are being calculated this way, because we need everything to be positive and normalized. The softmax turns logits, which are the numeric output of the last layer of a multi-class classification neural network, and converts it into probabilities by taking the exponents of each output and then normalizing each number by the sum of those exponents so the entire output vector adds up to one. The activation function is a softmax activation function.

Q2.14 Now we can plot the predicted and ground truth profile shapes. In the cell below, compute `pos_ref` and `neg_ref`, the ground truth labels for the Lefty1 enhancer region, by extracting read counts from `pos_counts` and `neg_counts`. Don't forget to use `np.nan_to_num` on the counts.

[15]:
```
# FILL IN CODE HERE #

pos_ref = np.nan_to_num(pos_counts.values(chr_no, start, end))
neg_ref = np.nan_to_num(neg_counts.values(chr_no, start, end))

# FILL IN CODE HERE #

plt.plot(pos_ref / pos_ref.sum(), alpha=0.7, label="pos_ref")
plt.plot(probs[0],  alpha=0.7, label="pos_pred")

plt.plot(-neg_ref / neg_ref.sum(), alpha=0.7, label="neg_ref")
plt.plot(-probs[1], alpha=0.7, label="neg_pred")

plt.legend()
plt.title("Oct4 binding profile for Lefty1 enhancer")
```

[15]:
```
Text(0.5, 1.0, 'Oct4 binding profile for Lefty1 enhancer')
```



Q2.15 Look at Fig 1e in the BPNet paper. Do your results on the Lefty1 enhancer match those resported in the paper?

Written answer: Yes they do match those reported in the paper.