A3_part2_bpnet

November 16, 2022

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%5Foct
-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%5Foct
-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%5Foct
-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_assembl
```

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.

1.1.1 Written answer:

- fasta: The FASTA file is a text file used to store either nucleotide sequences or amino acid sequences (in which both are represented using single letter codes) in fasta format. The FASTA format must have a header (always preceded by a ">") in the first line of the file, a the sequence starting in the second line. It is common for the header to include a summary description of the sequence, often starting with a unique library accession number.
- BigWig: BigWig stores quantitative values associated with with genomic positions. The BigWig format is for display of dense, continuous data that will be displayed as a graph. BigWig files are a compressed, indexed, binary format for genome-wide signal data for calculations (i.e. GC percent) or experiments (e.g. ChIP-seq/RNA-seq read depth).
- bed: The Bed format is a text file format used to store genomic regions as coordinates and associated annotations. It provides a flexible way to define the data lines that are displayed in an annotation track (i.e a feature track). The data are presented in the form of columns separated by spaces or tabs. The Bed format requires the following fields: chrom (chromasome number), chromStart (the starting position of the feature of interest), and chromEnd (the end position of the feature of interest). Additionally, it has 9 optional fields to describe the feature track.

[2]:
```
# Set ROOT to be the path to the `bpnet` directory
ROOT = "/home/jupyter/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 #
    df = pd.read_csv(peaks_file, sep ='\t', header = None).rename(columns = {0:␣
 ↪columns[0], 1: columns[1]}).drop(2, axis = 1)
    val_df = df[df['chr_no'] == 'chr1']
    train_df = df[df['chr_no'] != 'chr1']

    # 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. Make sure each sequence is all upper case. 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.iterrows():
        #sequences
        chrom, peak = row.values
        start = peak - 500
        end = peak + 500
        seq = fasta_ref[chrom][start:end]
        seq_comp = fasta_ref[chrom][start:end].complement
        sequences.extend([seq, seq_comp])
```

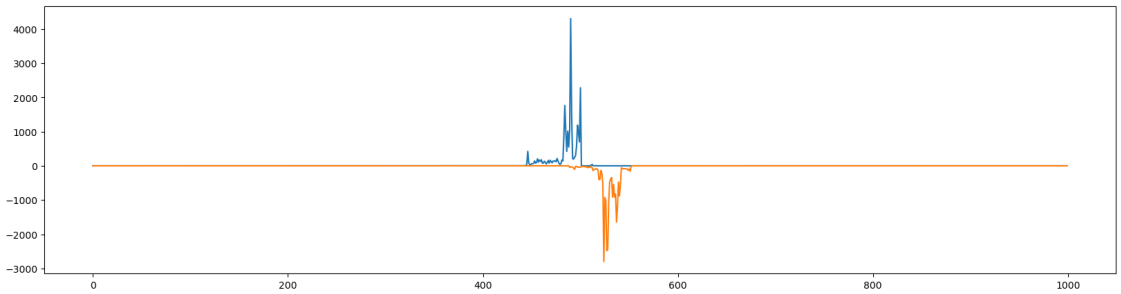
```
        #counts
        counts.extend([np.nan_to_num(pos_counts.values(chrom, start, end)), \
                       np.nan_to_num(neg_counts.values(chrom, start, end))])

    # 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 0x7f931e943950>]
```



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 = []
    for amino in sequence:
        arr = [0, 0, 0, 0, 0]
        arr[nucleotide_map[str(amino)]] = 1
        vec.append(arr)
    vec = np.array(vec)


    # 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.array([one_hot_encode(sequence) for sequence in sequences])
    profile_vec = counts/counts.sum(axis=1)[:,None]
```

```
    # 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.

1.2.1 Written answer:

1. Dilated convolutions and with residual-style layers - Dilated convolutions are a type of convolution that "inflate" the kernel by inserting holes between kernel elements. A parameter (l = dilation rate) indicates how much we should expand our kernel. L-1 pixels are skipped in the kernel. Dilated convolutions allow us to increase our receptive field vs a standard convolution for the same number of layers. This helps us avoid requiring many layers to increase the receptive field which is much more difficult to train. Pooling layers can also increase receptive field, but they reduce resolution (wheras dilated convolutions can mantain high resolution). Dilated convolutions are useful for this prediction problem since they provide a mechanism for considering a large sequence context without the need for pooling. This allows us to mantain base pair resolution. Residual style layers also help with the network degredation problem. Residual skip connections are connections in deep neural networks that feed the output (call this x0) of a particular layer to later layers in the network that are not directly adjacent to the layer from which the output originated. By adding x0, you are adding an identity mapping of the input to F(x) (the mapping learned by the network). This gives the network a baseline that the function has to learn. In deep networks, this gives the model the ability to ignore the mapping of specific layers which would otherwise degrade the output. Residual connections improve ease of optimization for more effective training for BPnet.

2. Profile regression loss - Profile regression loss is a two-part loss function for optimizing prediction of the binding profile across the input sequence. The first term evaluates the error in the

shape of the predicted profile. It is the multinomial negative log-likelihood of the observed base read counts given the predicted probabilities and total number of observed counts. The second term evaluates the squared error of the log total number of reads in the region. This loss function is used for each sequence, strand, and task. During BPNet training, the total loss function is the sum of individual loss functions across both strands, all input sequences, and all tasks. Lastly, there is a hyperparameter lambda which dictates the amount of weight given to the profile loss and total count loss. If $= \bar{}$obs/2, the profile loss and the total count loss will be given roughly equal weight. This is beneficial for this prediction problem since minimizing the loss will help you accurately predict profile shape (overall relatively where you have more signals than others) as well as signal amplitude; thus, your profile counts and shape will hopefully be quite similar to the ground truth.

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 "same" 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.conv = Conv1D(1, 25, padding = "same")
        self.pool = 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.conv(inputs)
```

```
        pool1 = self.pool(inputs)
        y = self.dense(pool1)

        # 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)
```

```
2022-11-15 16:47:49.799826: 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
2022-11-15 16:47:49.917418: 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
2022-11-15 16:47:49.919231: 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
2022-11-15 16:47:49.923337: 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.
2022-11-15 16:47:49.924623: 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
2022-11-15 16:47:49.926396: 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
2022-11-15 16:47:49.928121: 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
2022-11-15 16:47:52.141865: 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
2022-11-15 16:47:52.143860: 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
2022-11-15 16:47:52.145678: 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
2022-11-15 16:47:52.147355: 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
2022-11-15 16:47:53.430883: 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, relu activation and "same" padding 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]:
```
from tensorflow.keras.layers import Input


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

        # FILL IN CODE HERE #
        self.first_conv = Conv1D(64, 25, activation = "relu", padding = "same")
        self.conv1 = Conv1D(64, 3, activation = "relu", padding = "same",␣
 ↪dilation_rate = 2**1)
        self.conv2 = Conv1D(64, 3, activation = "relu", padding = "same",␣
 ↪dilation_rate = 2**2)
        self.conv3 = Conv1D(64, 3, activation = "relu", padding = "same",␣
 ↪dilation_rate = 2**3)
        self.conv4 = Conv1D(64, 3, activation = "relu", padding = "same",␣
 ↪dilation_rate = 2**4)
        self.conv5 = Conv1D(64, 3, activation = "relu", padding = "same",␣
 ↪dilation_rate = 2**5)
        self.conv6 = Conv1D(64, 3, activation = "relu", padding = "same",␣
 ↪dilation_rate = 2**6)
```

```
        self.conv7 = Conv1D(64, 3, activation = "relu", padding = "same",␣
↪dilation_rate = 2**7)
        self.conv8 = Conv1D(64, 3, activation = "relu", padding = "same",␣
↪dilation_rate = 2**8)
        self.conv9 = Conv1D(64, 3, activation = "relu", padding = "same",␣
↪dilation_rate = 2**9)
        # FILL IN CODE HERE #

        self.output_layer = BPNetOutputLayer()

    def call(self, inputs):
        # FILL IN CODE HERE #
        prev_layer = self.first_conv(inputs)

        conv_output = self.conv1(prev_layer)
        prev_layer = tf.keras.layers.Add()([prev_layer, conv_output]) #skip␣
↪connection

        conv_output = self.conv2(prev_layer)
        prev_layer = tf.keras.layers.Add()([prev_layer, conv_output]) #skip␣
↪connection

        conv_output = self.conv3(prev_layer)
        prev_layer = tf.keras.layers.Add()([prev_layer, conv_output]) #skip␣
↪connection

        conv_output = self.conv4(prev_layer)
        prev_layer = tf.keras.layers.Add()([prev_layer, conv_output]) #skip␣
↪connection

        conv_output = self.conv5(prev_layer)
        prev_layer = tf.keras.layers.Add()([prev_layer, conv_output]) #skip␣
↪connection

        conv_output = self.conv6(prev_layer)
        prev_layer = tf.keras.layers.Add()([prev_layer, conv_output]) #skip␣
↪connection

        conv_output = self.conv7(prev_layer)
        prev_layer = tf.keras.layers.Add()([prev_layer, conv_output]) #skip␣
↪connection

        conv_output = self.conv8(prev_layer)
        prev_layer = tf.keras.layers.Add()([prev_layer, conv_output]) #skip␣
↪connection
```

```
        conv_output = self.conv9(prev_layer)
        prev_layer = tf.keras.layers.Add()([prev_layer, conv_output]) #skip␣
↪connection

        output = self.output_layer(prev_layer)
        return output
        # 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 #
=================================================================
input_1 (InputLayer)         [(None, 1000, 5)]         0
_________________________________________________________________
conv1d_1 (Conv1D)            (None, 1000, 64)          8064
_________________________________________________________________
conv1d_2 (Conv1D)            (None, 1000, 64)          12352
_________________________________________________________________
add (Add)                    (None, 1000, 64)          0
_________________________________________________________________
conv1d_3 (Conv1D)            (None, 1000, 64)          12352
_________________________________________________________________
add_1 (Add)                  (None, 1000, 64)          0
_________________________________________________________________
conv1d_4 (Conv1D)            (None, 1000, 64)          12352
_________________________________________________________________
add_2 (Add)                  (None, 1000, 64)          0
_________________________________________________________________
```

```
conv1d_5 (Conv1D)               (None, 1000, 64)     12352       add_2[0][0]
__________________________________________________________________________________
__________________
add_3 (Add)                     (None, 1000, 64)     0           add_2[0][0]
                                                                 conv1d_5[0][0]
__________________________________________________________________________________
__________________
conv1d_6 (Conv1D)               (None, 1000, 64)     12352       add_3[0][0]
__________________________________________________________________________________
__________________
add_4 (Add)                     (None, 1000, 64)     0           add_3[0][0]
                                                                 conv1d_6[0][0]
__________________________________________________________________________________
__________________
conv1d_7 (Conv1D)               (None, 1000, 64)     12352       add_4[0][0]
__________________________________________________________________________________
__________________
add_5 (Add)                     (None, 1000, 64)     0           add_4[0][0]
                                                                 conv1d_7[0][0]
__________________________________________________________________________________
__________________
conv1d_8 (Conv1D)               (None, 1000, 64)     12352       add_5[0][0]
__________________________________________________________________________________
__________________
add_6 (Add)                     (None, 1000, 64)     0           add_5[0][0]
                                                                 conv1d_8[0][0]
__________________________________________________________________________________
__________________
conv1d_9 (Conv1D)               (None, 1000, 64)     12352       add_6[0][0]
__________________________________________________________________________________
__________________
add_7 (Add)                     (None, 1000, 64)     0           add_6[0][0]
                                                                 conv1d_9[0][0]
__________________________________________________________________________________
__________________
conv1d_10 (Conv1D)              (None, 1000, 64)     12352       add_7[0][0]
__________________________________________________________________________________
__________________
add_8 (Add)                     (None, 1000, 64)     0           add_7[0][0]
                                                                 conv1d_10[0][0]
__________________________________________________________________________________
__________________
bp_net_output_layer_1 (BPNetOut ((None, 1000), (None 1666        add_8[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]:
```
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 #
    loss = 0
    total_true_counts = tf.reduce_sum(true_counts, 1)
    multinomial = tfp.distributions.Multinomial(
        total_true_counts,
        logits=pred_profile,
        name='Multinomial'
        )
    multinomial_nll = -tf.reduce_mean(multinomial.log_prob(true_counts))
    return c
    # 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. n_{obs} is total reads here.

[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 #
    total_true_counts = tf.reduce_sum(count_vec, 1)
    term2 = tf.expand_dims(tf.math.log(1 + total_true_counts), 1)
    return term2

    # FILL IN CODE HERE #

log_train_counts = log_normalize(train_count_vec)
log_val_counts = log_normalize(val_count_vec)
```

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.

[31]:
```
model = BPNetModel()

model.compile(
    optimizer=tf.keras.optimizers.Adam(learning_rate=1e-3),
    # FILL IN CODE HERE #
    loss = {'output_1': multinomial_nll, 'output_2':'mse'},
    # FILL IN CODE HERE #
)

model.fit(
    # FILL IN CODE HERE #
    x = train_seq_vec,
    y = {'output_1': train_count_vec, 'output_2': log_train_counts},
    validation_data = (val_seq_vec, {'output_1': val_count_vec, 'output_2':␣
 ↪log_val_counts}),

    # FILL IN CODE HERE #
    epochs=25,
    batch_size=512
)
```

```
Epoch 1/25
95/95 [==============================] - 50s 515ms/step - loss: 545.6631 -
output_1_loss: 543.1376 - output_2_loss: 2.5254 - val_loss: 516.0924 -
```

```
val_output_1_loss: 515.7725 - val_output_2_loss: 0.3200
Epoch 2/25
95/95 [==============================] - 48s 503ms/step - loss: 526.7451 -
output_1_loss: 526.3600 - output_2_loss: 0.3850 - val_loss: 515.0567 -
val_output_1_loss: 514.5903 - val_output_2_loss: 0.4665
Epoch 3/25
95/95 [==============================] - 48s 507ms/step - loss: 524.8043 -
output_1_loss: 524.4222 - output_2_loss: 0.3822 - val_loss: 513.6396 -
val_output_1_loss: 513.3129 - val_output_2_loss: 0.3268
Epoch 4/25
95/95 [==============================] - 48s 504ms/step - loss: 523.5213 -
output_1_loss: 523.1757 - output_2_loss: 0.3457 - val_loss: 512.4658 -
val_output_1_loss: 512.1493 - val_output_2_loss: 0.3165
Epoch 5/25
95/95 [==============================] - 48s 505ms/step - loss: 522.3760 -
output_1_loss: 522.0400 - output_2_loss: 0.3358 - val_loss: 512.5411 -
val_output_1_loss: 512.2033 - val_output_2_loss: 0.3379
Epoch 6/25
95/95 [==============================] - 48s 505ms/step - loss: 521.7602 -
output_1_loss: 521.4216 - output_2_loss: 0.3386 - val_loss: 511.4283 -
val_output_1_loss: 511.1300 - val_output_2_loss: 0.2983
Epoch 7/25
95/95 [==============================] - 48s 504ms/step - loss: 520.4753 -
output_1_loss: 520.1556 - output_2_loss: 0.3198 - val_loss: 514.0724 -
val_output_1_loss: 513.6696 - val_output_2_loss: 0.4028
Epoch 8/25
95/95 [==============================] - 48s 504ms/step - loss: 520.1934 -
output_1_loss: 519.7733 - output_2_loss: 0.4203 - val_loss: 510.9966 -
val_output_1_loss: 510.6851 - val_output_2_loss: 0.3114
Epoch 9/25
95/95 [==============================] - 48s 504ms/step - loss: 518.5797 -
output_1_loss: 518.2494 - output_2_loss: 0.3304 - val_loss: 512.4573 -
val_output_1_loss: 512.0836 - val_output_2_loss: 0.3737
Epoch 10/25
95/95 [==============================] - 48s 504ms/step - loss: 517.4163 -
output_1_loss: 517.0817 - output_2_loss: 0.3347 - val_loss: 509.2358 -
val_output_1_loss: 508.9271 - val_output_2_loss: 0.3088
Epoch 11/25
95/95 [==============================] - 48s 504ms/step - loss: 516.2194 -
output_1_loss: 515.8969 - output_2_loss: 0.3224 - val_loss: 508.9843 -
val_output_1_loss: 508.6030 - val_output_2_loss: 0.3812
Epoch 12/25
95/95 [==============================] - 48s 504ms/step - loss: 513.4039 -
output_1_loss: 513.0567 - output_2_loss: 0.3474 - val_loss: 505.0981 -
val_output_1_loss: 504.7579 - val_output_2_loss: 0.3401
Epoch 13/25
95/95 [==============================] - 48s 504ms/step - loss: 512.5548 -
output_1_loss: 512.1625 - output_2_loss: 0.3924 - val_loss: 503.9886 -
```

```
val_output_1_loss: 503.6581 - val_output_2_loss: 0.3305
Epoch 14/25
95/95 [==============================] - 48s 505ms/step - loss: 508.5591 -
output_1_loss: 508.2119 - output_2_loss: 0.3472 - val_loss: 504.5620 -
val_output_1_loss: 504.2061 - val_output_2_loss: 0.3559
Epoch 15/25
95/95 [==============================] - 48s 505ms/step - loss: 507.2990 -
output_1_loss: 506.9328 - output_2_loss: 0.3663 - val_loss: 500.3189 -
val_output_1_loss: 499.9916 - val_output_2_loss: 0.3274
Epoch 16/25
95/95 [==============================] - 48s 505ms/step - loss: 505.4402 -
output_1_loss: 505.0795 - output_2_loss: 0.3609 - val_loss: 499.4777 -
val_output_1_loss: 499.1150 - val_output_2_loss: 0.3627
Epoch 17/25
95/95 [==============================] - 48s 505ms/step - loss: 503.2654 -
output_1_loss: 502.9170 - output_2_loss: 0.3482 - val_loss: 497.9795 -
val_output_1_loss: 497.6598 - val_output_2_loss: 0.3197
Epoch 18/25
95/95 [==============================] - 48s 505ms/step - loss: 502.5480 -
output_1_loss: 502.1731 - output_2_loss: 0.3749 - val_loss: 498.9788 -
val_output_1_loss: 498.6568 - val_output_2_loss: 0.3220
Epoch 19/25
95/95 [==============================] - 48s 504ms/step - loss: 501.0912 -
output_1_loss: 500.7450 - output_2_loss: 0.3462 - val_loss: 497.6544 -
val_output_1_loss: 497.3247 - val_output_2_loss: 0.3296
Epoch 20/25
95/95 [==============================] - 48s 504ms/step - loss: 499.6656 -
output_1_loss: 499.3346 - output_2_loss: 0.3311 - val_loss: 497.2812 -
val_output_1_loss: 496.9000 - val_output_2_loss: 0.3812
Epoch 21/25
95/95 [==============================] - 48s 504ms/step - loss: 501.6740 -
output_1_loss: 501.3115 - output_2_loss: 0.3628 - val_loss: 495.3046 -
val_output_1_loss: 494.9922 - val_output_2_loss: 0.3124
Epoch 22/25
95/95 [==============================] - 48s 504ms/step - loss: 498.4539 -
output_1_loss: 498.1158 - output_2_loss: 0.3382 - val_loss: 493.9617 -
val_output_1_loss: 493.6636 - val_output_2_loss: 0.2981
Epoch 23/25
95/95 [==============================] - 48s 504ms/step - loss: 497.9706 -
output_1_loss: 497.6513 - output_2_loss: 0.3192 - val_loss: 495.8788 -
val_output_1_loss: 495.5746 - val_output_2_loss: 0.3042
Epoch 24/25
95/95 [==============================] - 48s 504ms/step - loss: 498.6985 -
output_1_loss: 498.3873 - output_2_loss: 0.3112 - val_loss: 493.1875 -
val_output_1_loss: 492.8781 - val_output_2_loss: 0.3094
Epoch 25/25
95/95 [==============================] - 48s 503ms/step - loss: 497.6529 -
output_1_loss: 497.3398 - output_2_loss: 0.3132 - val_loss: 492.8800 -
```

```
val_output_1_loss: 492.5695 - val_output_2_loss: 0.3105
```

[31]:
```
<keras.callbacks.History at 0x7f9210301a10>
```

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.

[39]:
```
# Lefty1 enhancer locus #
chr_no = "chr1"
lefty_idx = 180924952
start, end = lefty_idx - 500, lefty_idx + 500

# FILL IN CODE HERE #
seq_pos = fasta_ref[chr_no][start:end]
seq_comp = fasta_ref[chr_no][start:end].complement
seq_vecs = np.array([one_hot_encode(i) for i in [seq_pos, seq_comp]])

# 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)
```

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?

1.2.2 Written answer:The softmax activation function is being applied to the logits. The softmax activation function transforms the raw outputs of the network into a vector of probabilities, essentially a probability distribution over the input classes. In this senario, the input classes are each base in the 1000bp sequence where each value is the predicted probability values (the probability observing a particular read at a particular position in the input sequence) for a distinct position(i) and strand(+/-).

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.

[41]:
```
# 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")
```

[41]: `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?

1.2.3 Written answer: The results of the Lefty1 enhancer match those reported in the paper. In the paper, the Lefty1 predicted read counts have very similar same peaks and low points to the ground truth profile, meaning they have similar shapes. However, the predicted read counts are on a smaller scale than the ground truth read counts. In our results, the predicted and reference read counts also have similar shapes while they are slightly on different scales.

[ ]:

[ ]: