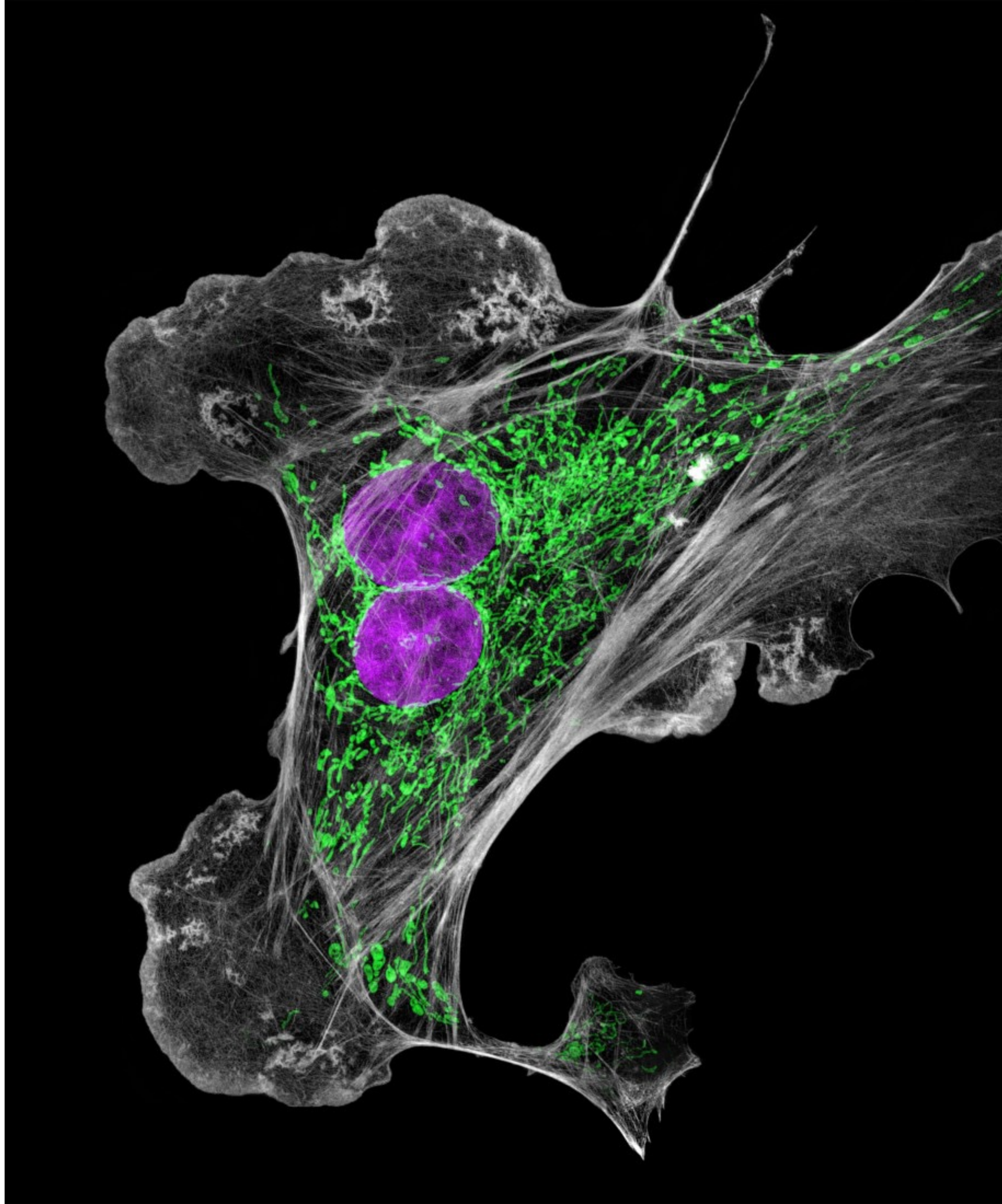
Predicting gene expression from random promotor sequences

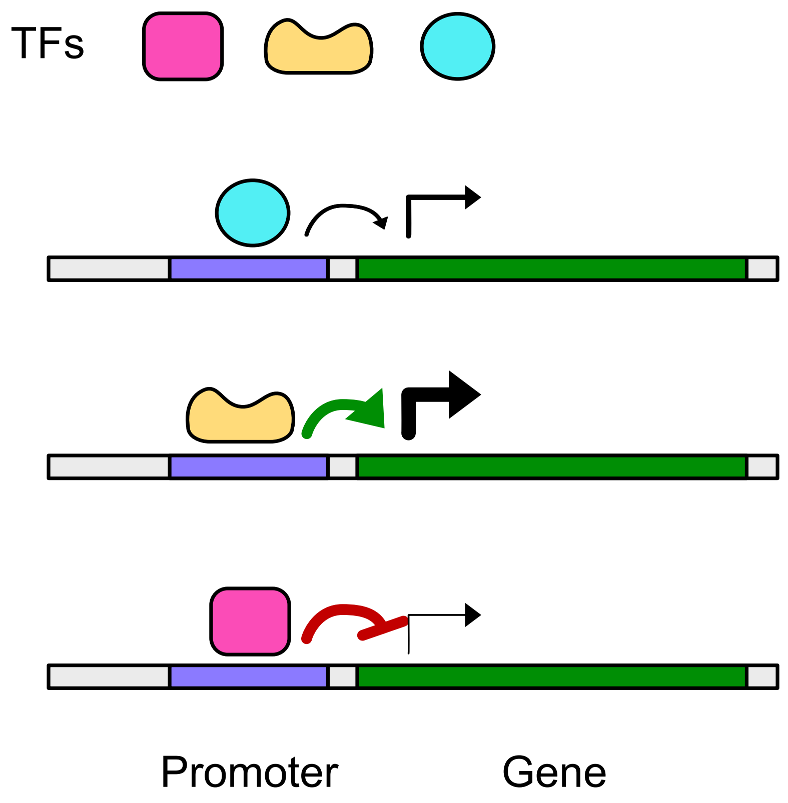
Gabija Kavaliauskaite (gkav@sdu.dk), Andreas Fønss Møller (andreasfm@bmb.sdu.dk)



Contents

Transcriptional regulation

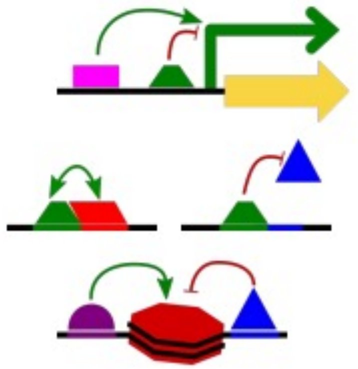
- Regulatory proteins with DNA binding domains
- Usually bind in promoter regions
- Positive or negative regulation of proximal genes



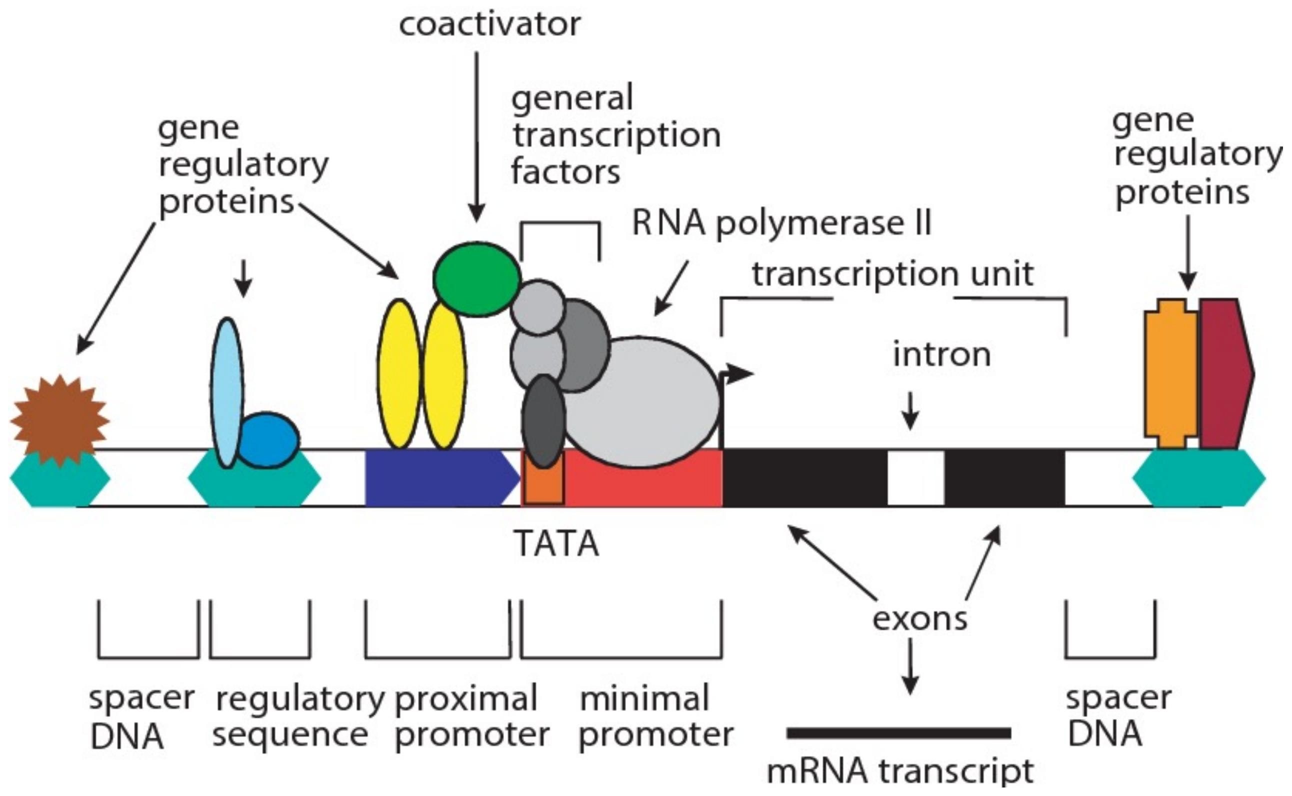
Cis-regulatory logic is complicated!

Many TFs (human ~1700, yeast ~210)

- Concentrations
- Action (activating/repressing)
- Position specific effect
- Chromatin organization
- TF interactions



There is also trans logic!

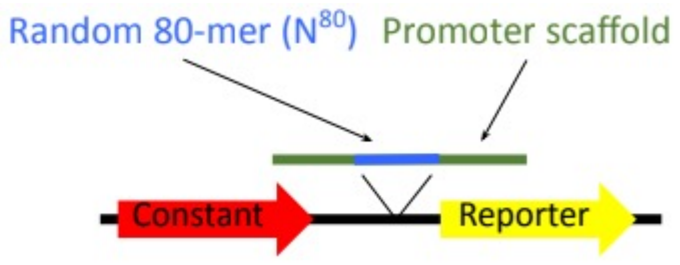


Villard J. Transcription regulation and human diseases. Swiss Med Wkly. 2004

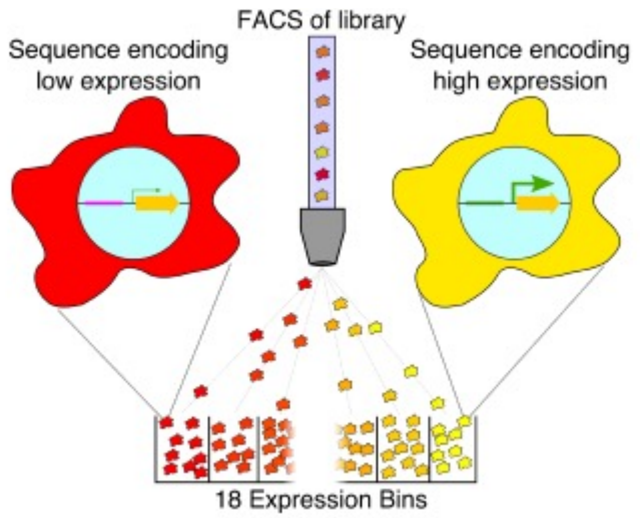
drive.google.com/file/d/150eHgy-x3R9ZMBENoDT6zfq4KLgKcmfn/view

The dataset

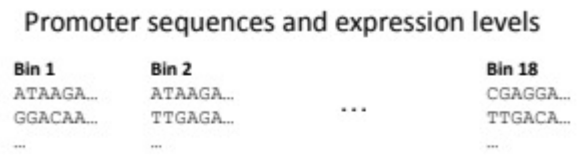
Step 1: Construct random sequence library



Step 2: Sort by expression


FACS of library
Sequence encoding low expression
Sequence encoding high expression
18 Expression Bins

Step 2: Sequence promoters



Promoter sequences and expression levels

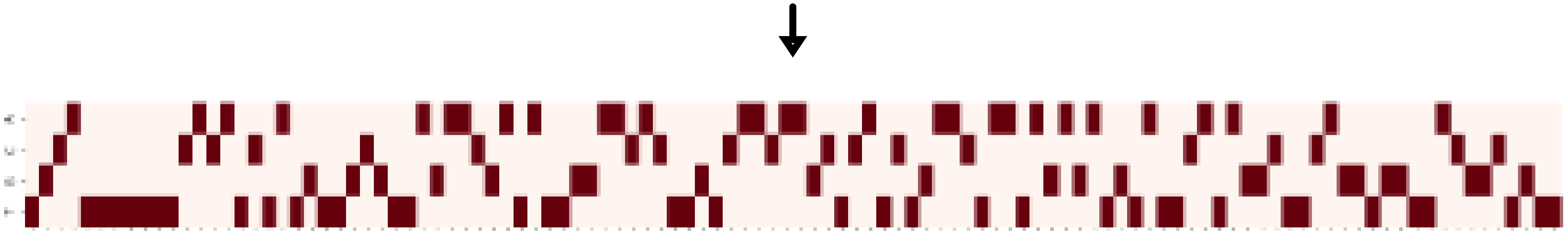
Bin 1	Bin 2		Bin 18
ATAAGA…	ATAAGA…	…	CGAGGA…
GGACAA…	TTGAGA…		TTGACA…
…	…		…

drive.google.com/file/d/150eHgy-x3R9ZMBENoDT6zfq4KLgKcmfn/view

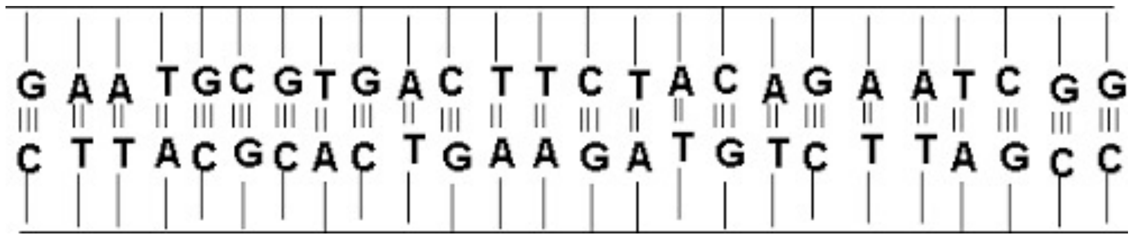
Data Preprocessing

Onehot encode sequences (5' to 3')

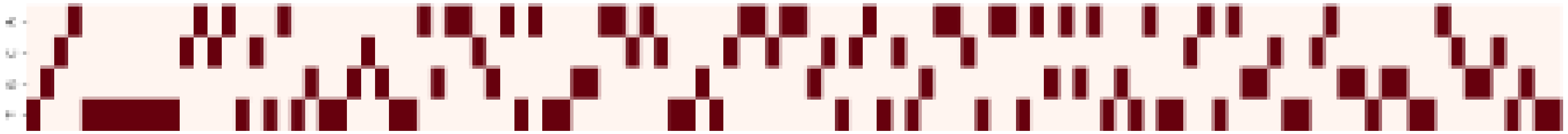
TGCATTTTTTTCACATCTATGTTGCGTTAGAACGATATTGGAACACTTGTCAACAAGCTCATCTGAACTAATAGAGATGTATTCATAGGCTTCAGGTGGTTACGGCTGTT



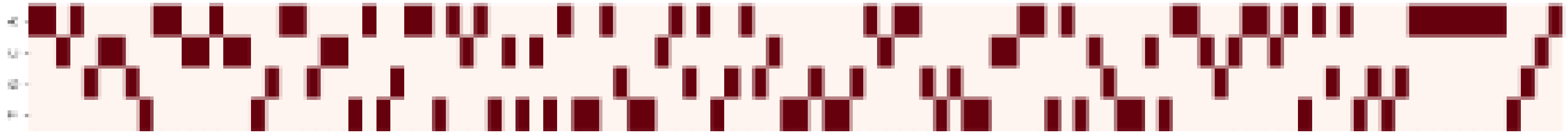
Data Preprocessing


G A A T G C G T G A C T T C T A C A G A A T C G G
C T T A C G C A C T G A A G A T G T C T T A G C C

Forward strand (5' to 3')



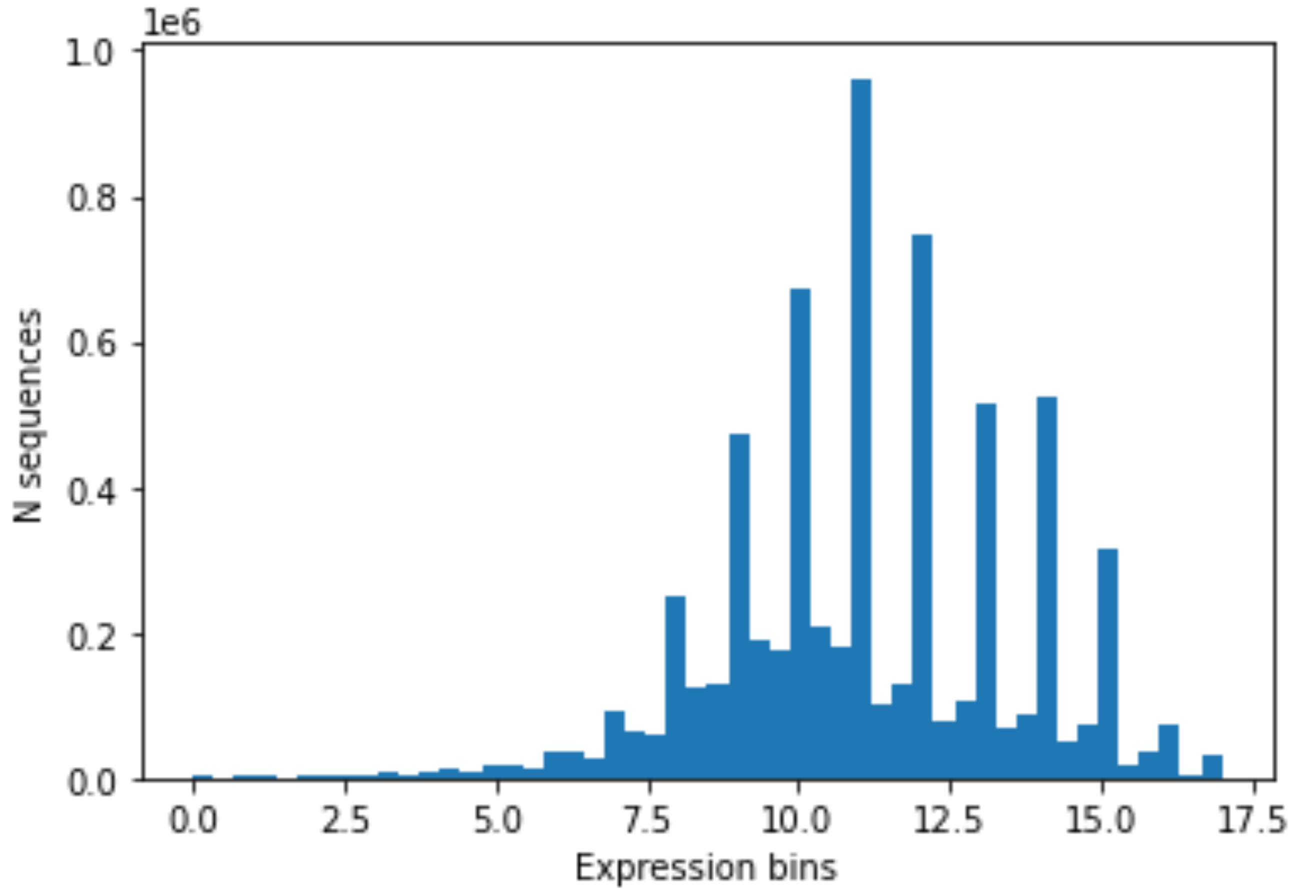
Reverse complement (5' to 3')



https://www.scientific.org/tutorials/articles/riley/riley.html

Data Preprocessing

Distribution of expressions



Data Pipeline

large dataset and TPU training

TFRecord

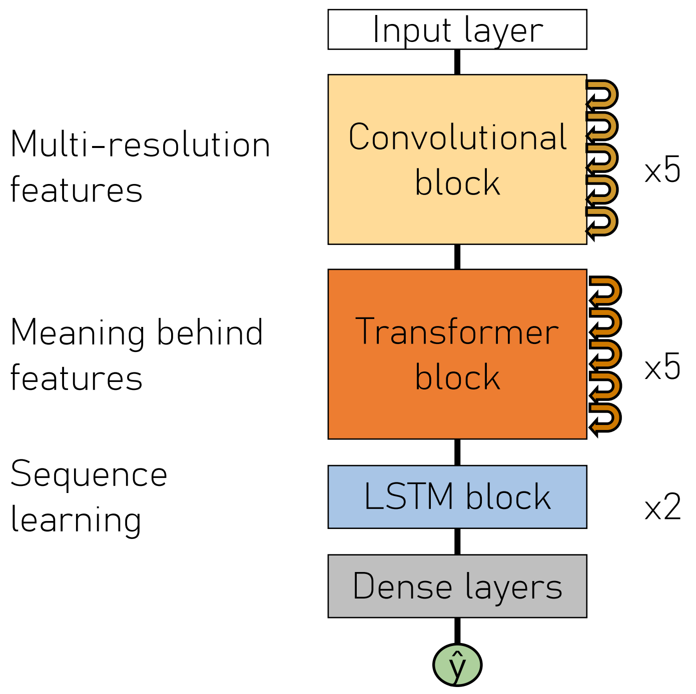
- Saving data in 68 files (100k seqs per file) for parallel read/write
- Sequences as binary BytesList, Expression values as FloatList
- Define how to read from them using a streaming approach

GCS + TPU

- Write files to GCS for TPU acces
- Authenticaiton of TPU runtime
- Different batch size systems on TPU

Model architecture

Model designed with 4 major parts:



Parameters trained: 13,602,217

Optimizer - Adam;

Learning_rate – 0.001;

Dropout_rate – 0.3;

Number of heads – 8;

Filters (Conv1D) – 128;

Filters (Conv2D) – 256;

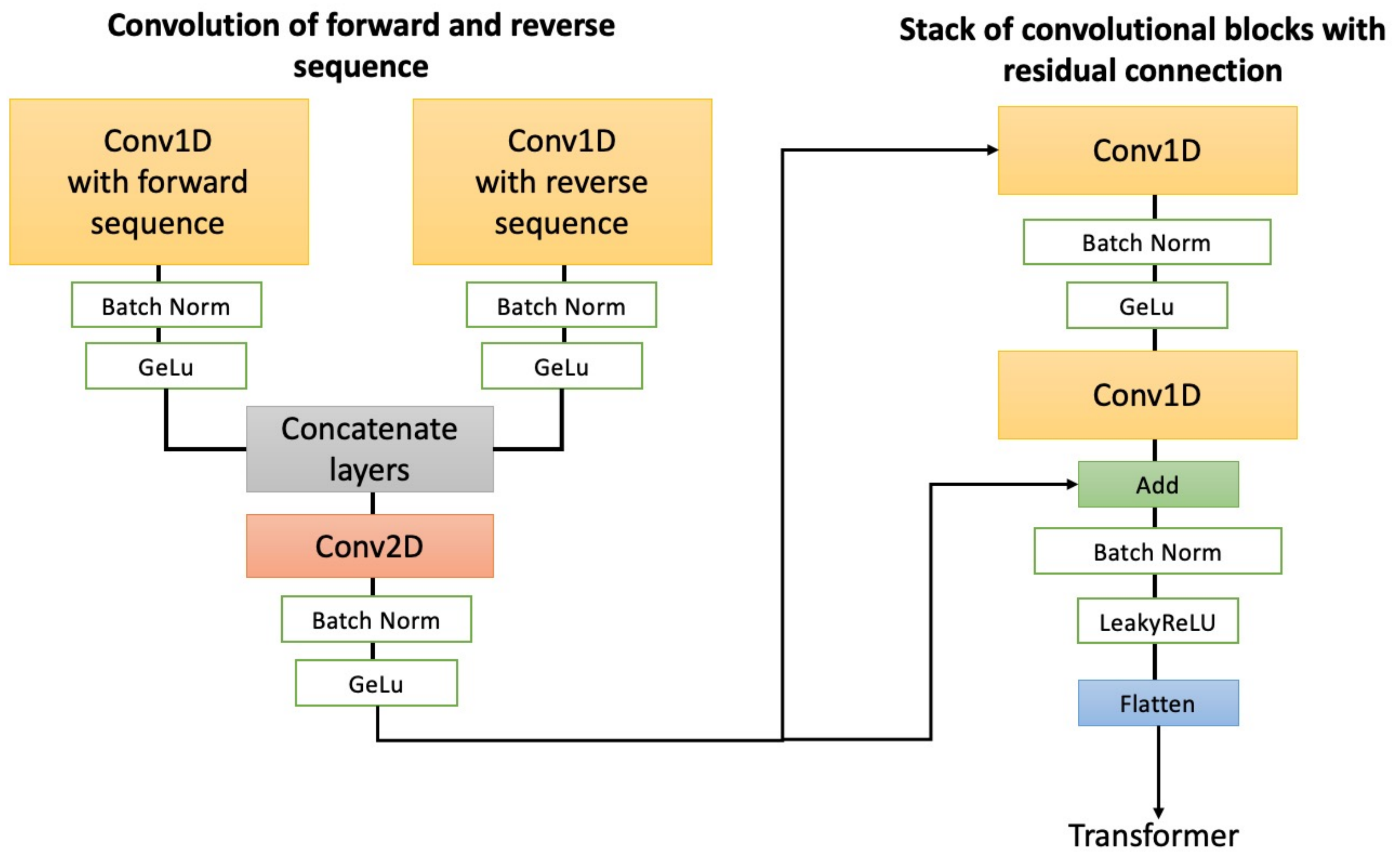
Number of residual layers – 5;

Number of attention layers – 5;

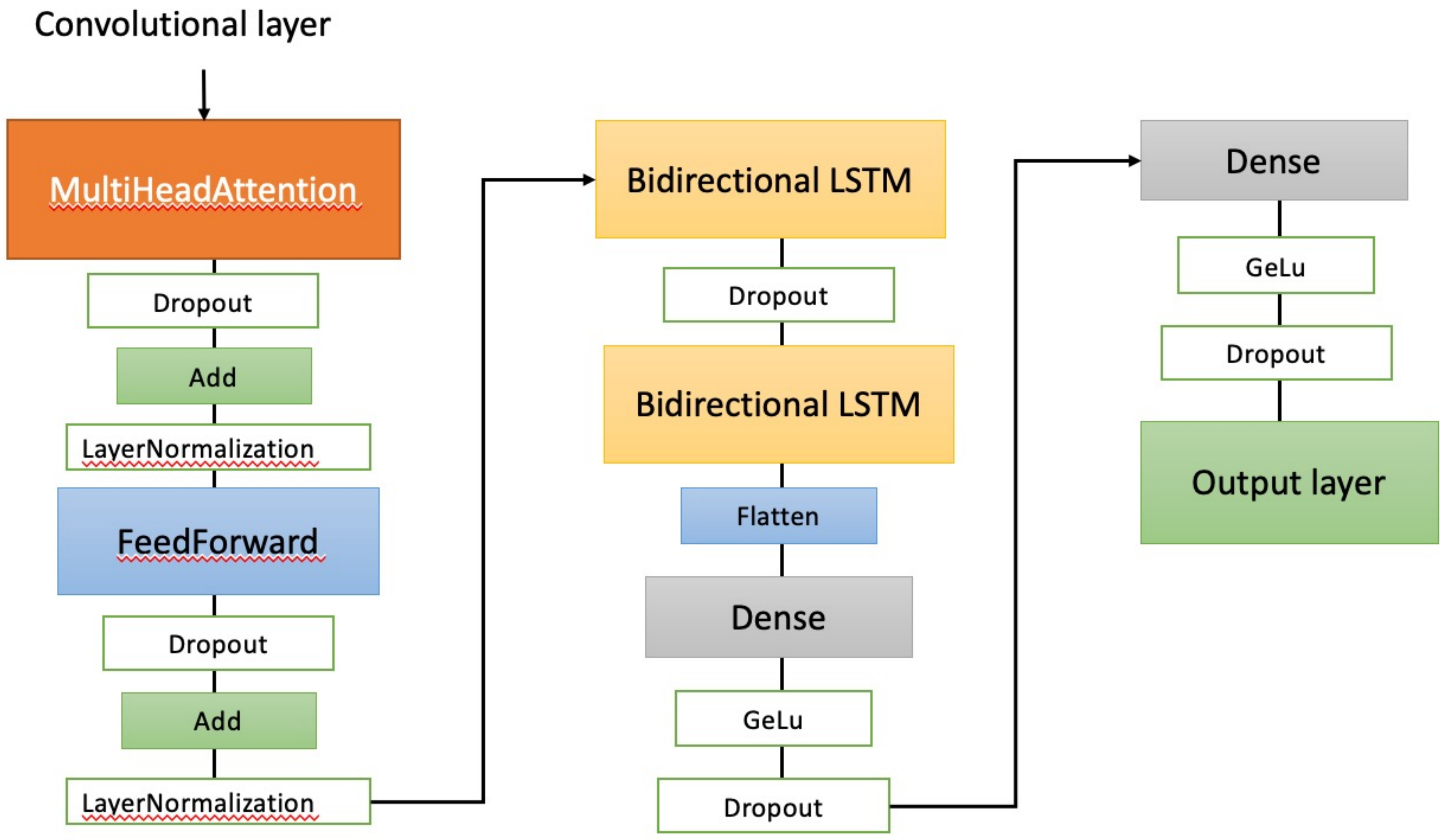
Kernel size – 7;

Loss function – MSE;

Stacked CNN with forward-and reverse-sequences

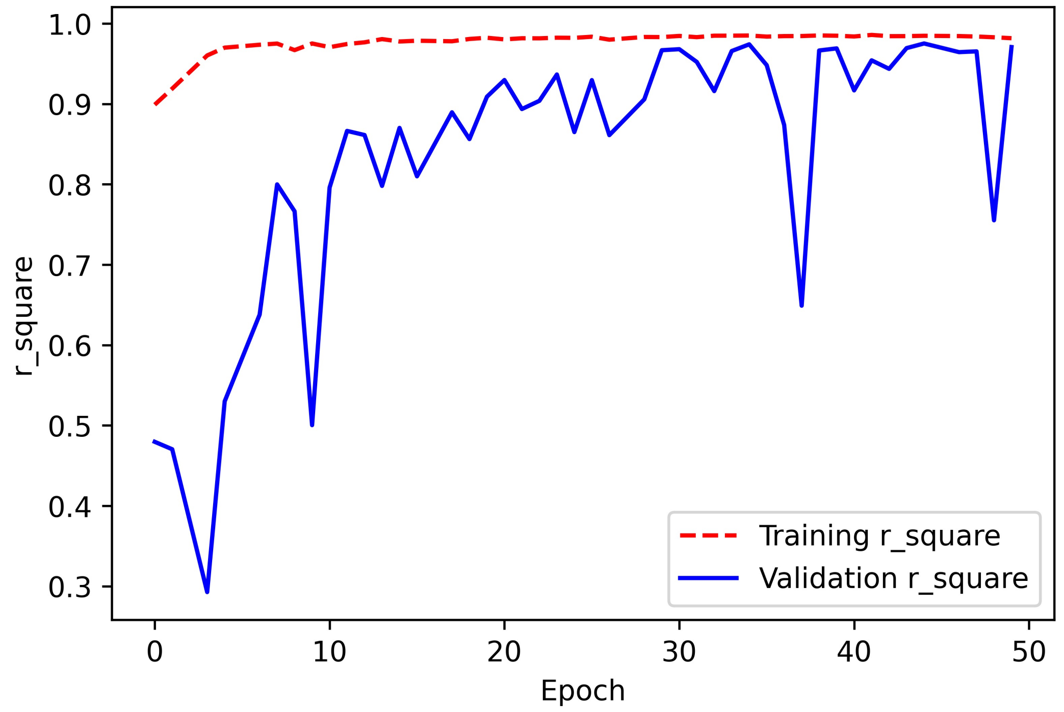

Convolution of forward and reverse sequence
Conv1D with forward sequence
Conv1D with reverse sequence
Batch Norm
GeLu
Batch Norm
GeLu
Concatenate layers
Conv2D
Batch Norm
GeLu
Stack of convolutional blocks with residual connection
Conv1D
Batch Norm
GeLu
Conv1D
Add
Batch Norm
LeakyReLU
Flatten
Transformer

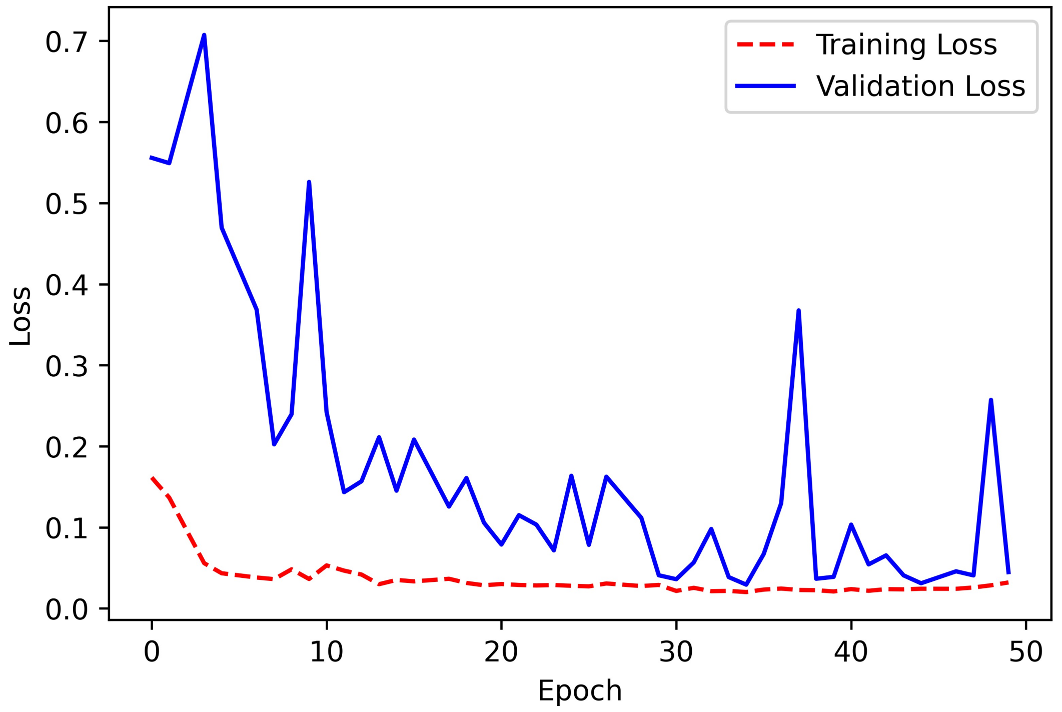
Transformer with stacked bidirectional LSTM



Evaluation of the method

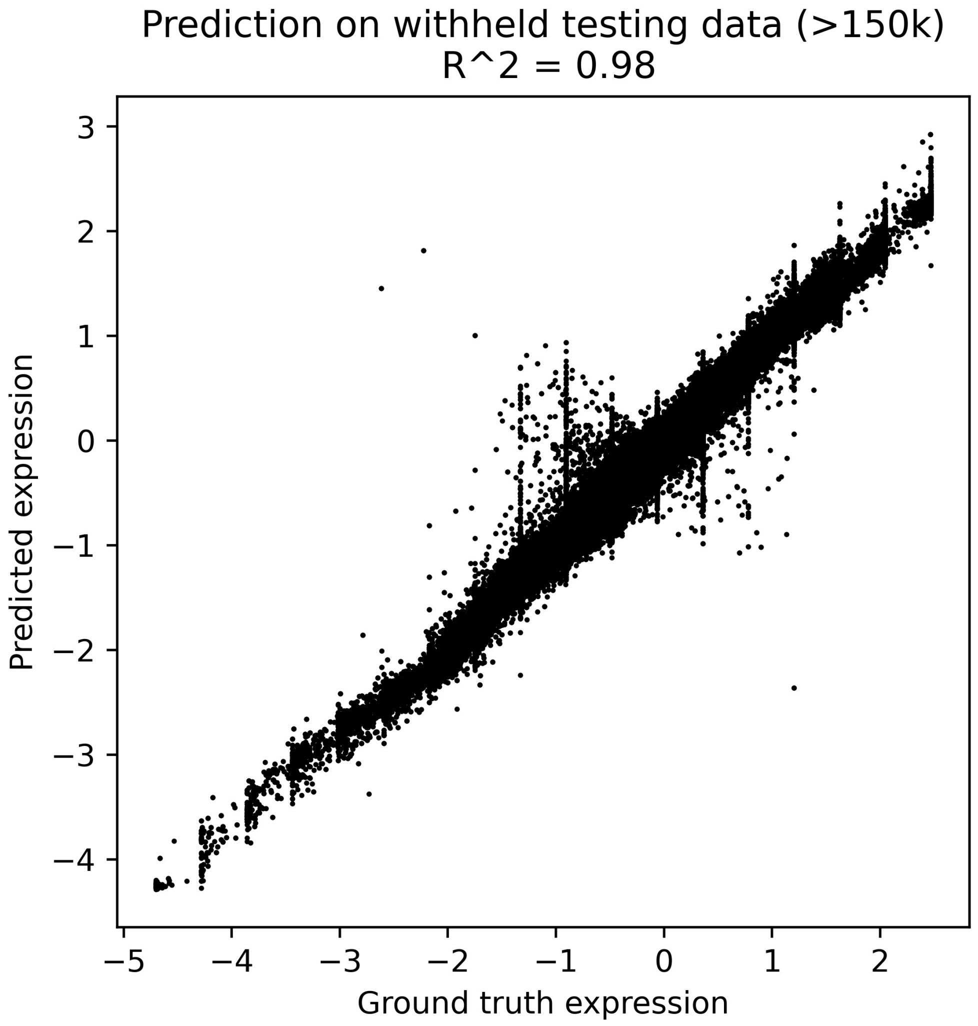
- Training data: 0.95 of the data to train (6402295 sequences)
- Validation and test: 0.025 each (168481 seq for validation and test)





Evaluation of the method

- The max R^2 on validation data: 0.97518
- The lowest loss on validation data: 0.0311

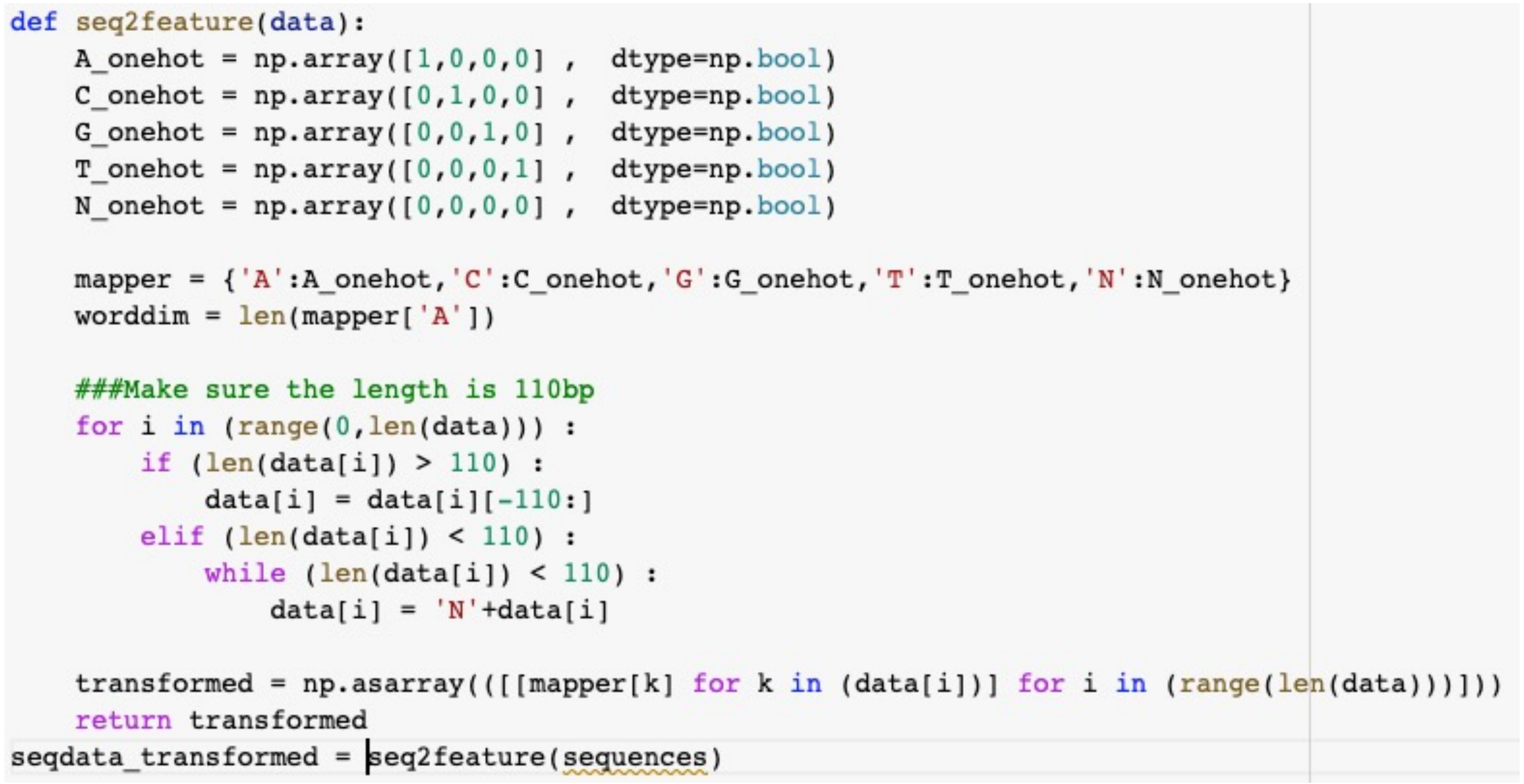


Conclusion and future perspectives

- The model is capable of reaching 0.97518 (R^2) on validation and 0.98 on the test data;
- Test the model with independent dataset;
- Possibility to optimize the model further (hyperparameter tuning);
- Pre-filtering by removing the sequences that are similar in both training and test datasets;

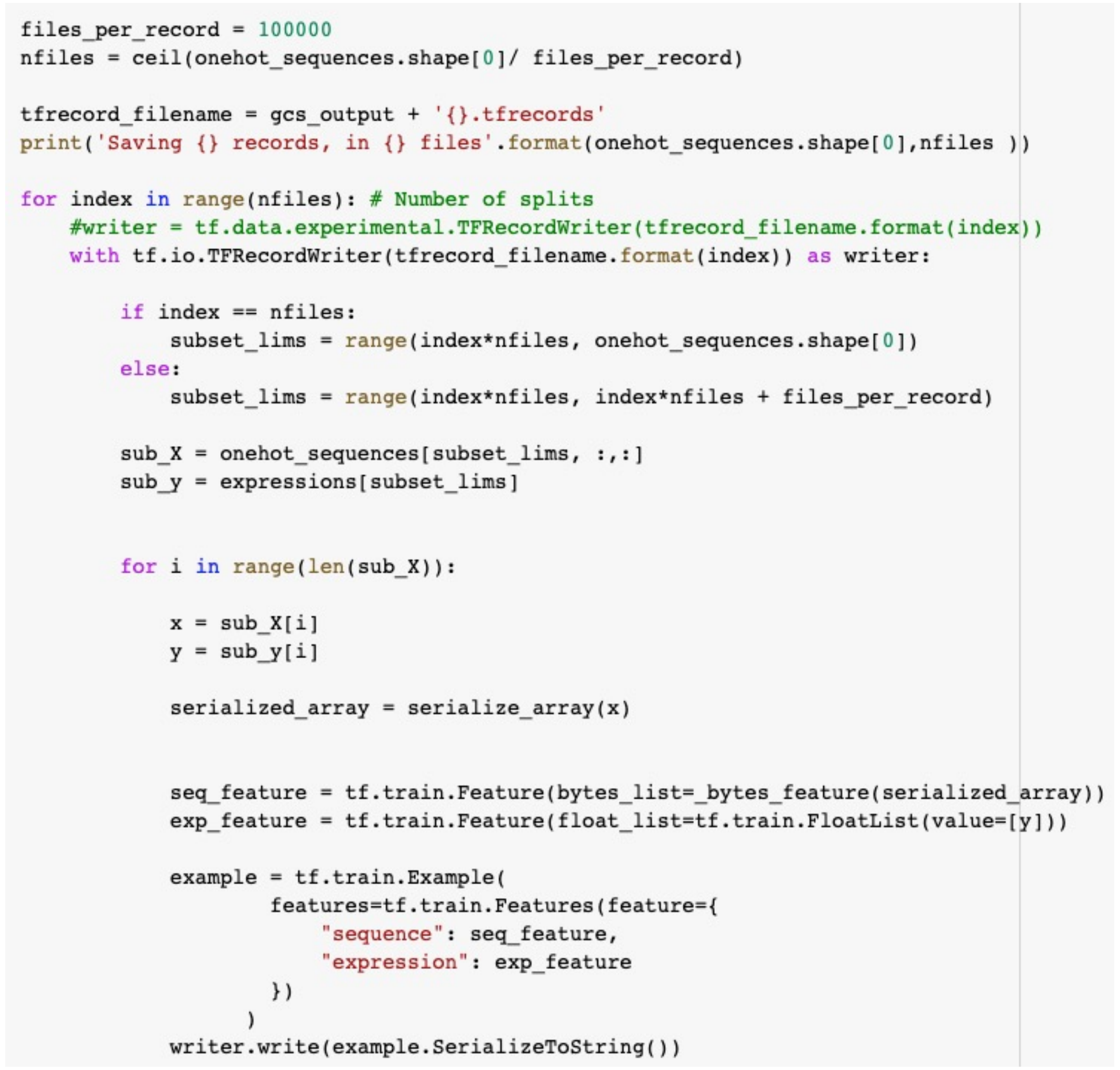
Appendix

Custom onehot encode sequences

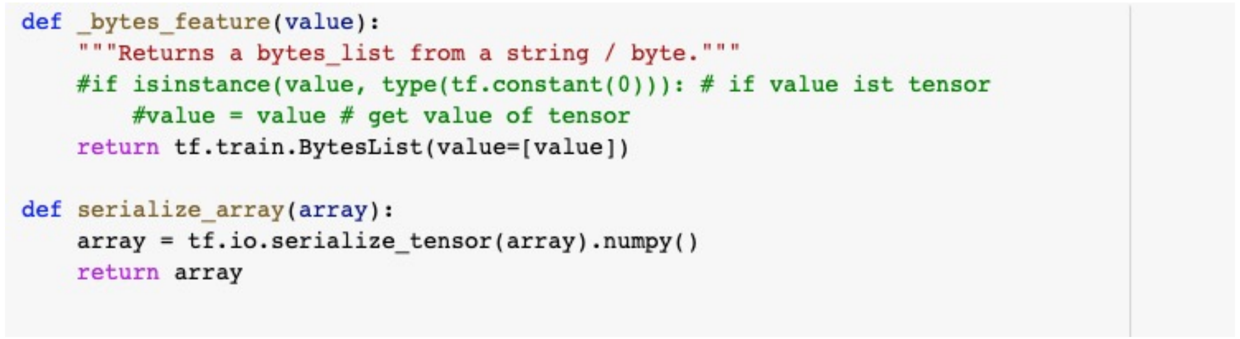


```python
def seq2feature(data):
    A_onehot = np.array([1,0,0,0] ,  dtype=np.bool)
    C_onehot = np.array([0,1,0,0] ,  dtype=np.bool)
    G_onehot = np.array([0,0,1,0] ,  dtype=np.bool)
    T_onehot = np.array([0,0,0,1] ,  dtype=np.bool)
    N_onehot = np.array([0,0,0,0] ,  dtype=np.bool)

    mapper = {'A':A_onehot,'C':C_onehot,'G':G_onehot,'T':T_onehot,'N':N_onehot}
    worddim = len(mapper['A'])

    ###Make sure the length is 110bp
    for i in (range(0,len(data))) :
        if (len(data[i]) > 110) :
            data[i] = data[i][-110:]
        elif (len(data[i]) < 110) :
            while (len(data[i]) < 110) :
                data[i] = 'N'+data[i]

    transformed = np.asarray(([[mapper[k] for k in (data[i])] for i in (range(len(data)))]))
    return transformed
seqdata_transformed = seq2feature(sequences)
```

Save TFRecord to GCS

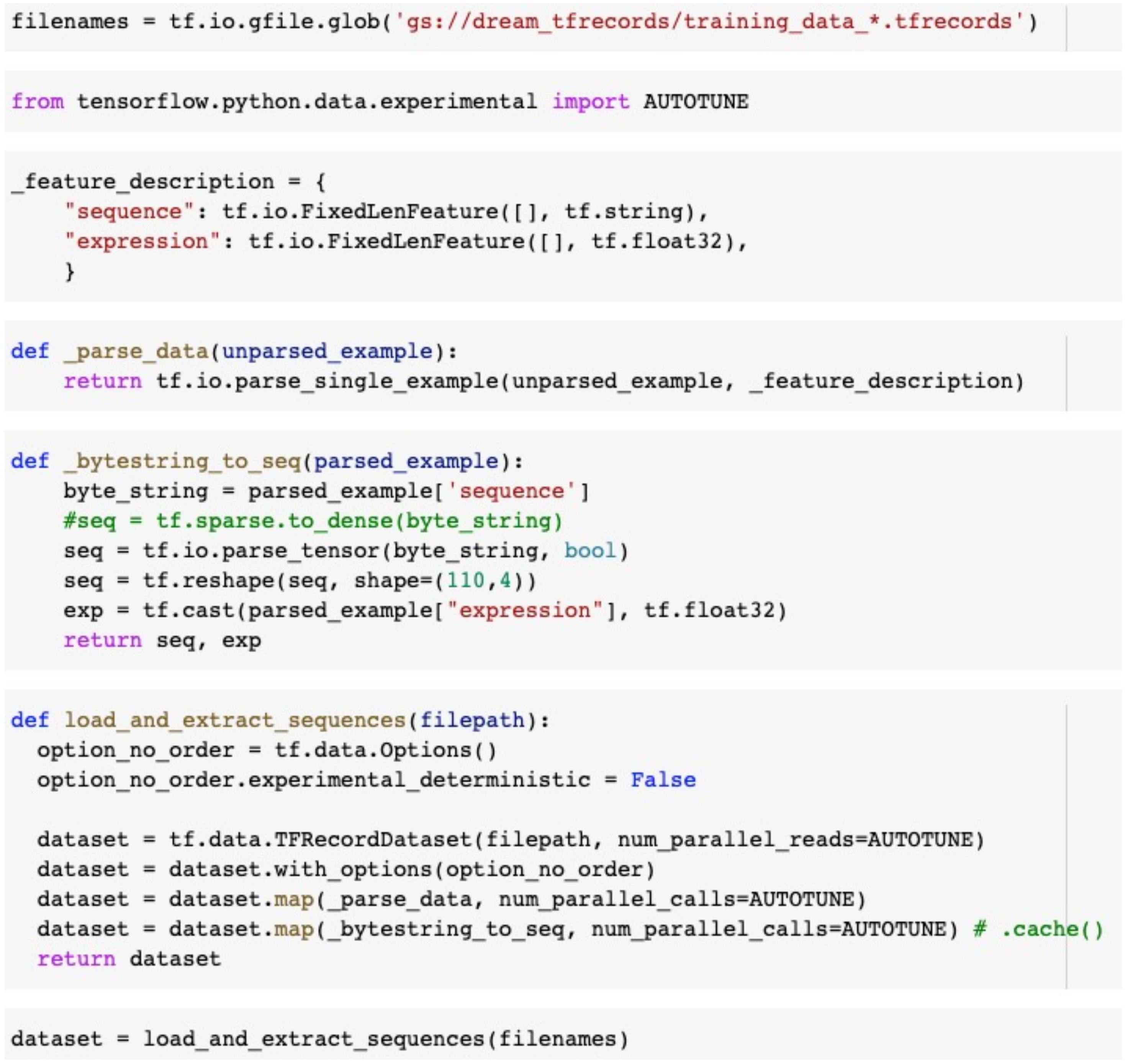


```python
files_per_record = 100000
nfiles = ceil(onehot_sequences.shape[0]/ files_per_record)

tfrecord_filename = gcs_output + '{}.tfrecords'
print('Saving {} records, in {} files'.format(onehot_sequences.shape[0],nfiles ))

for index in range(nfiles): # Number of splits
    #writer = tf.data.experimental.TFRecordWriter(tfrecord_filename.format(index))
    with tf.io.TFRecordWriter(tfrecord_filename.format(index)) as writer:

        if index == nfiles:
            subset_lims = range(index*nfiles, onehot_sequences.shape[0])
        else:
            subset_lims = range(index*nfiles, index*nfiles + files_per_record)

        sub_X = onehot_sequences[subset_lims, :,:]
        sub_y = expressions[subset_lims]


        for i in range(len(sub_X)):

            x = sub_X[i]
            y = sub_y[i]

            serialized_array = serialize_array(x)


            seq_feature = tf.train.Feature(bytes_list=_bytes_feature(serialized_array))
            exp_feature = tf.train.Feature(float_list=tf.train.FloatList(value=[y]))

            example = tf.train.Example(
                    features=tf.train.Features(feature={
                        "sequence": seq_feature,
                        "expression": exp_feature
                    })
                )
            writer.write(example.SerializeToString())
```



```python
def _bytes_feature(value):
    """Returns a bytes_list from a string / byte."""
    #if isinstance(value, type(tf.constant(0))): # if value ist tensor
        #value = value # get value of tensor
    return tf.train.BytesList(value=[value])

def serialize_array(array):
    array = tf.io.serialize_tensor(array).numpy()
    return array
```

Reading TFRecord from GCS



```
filenames = tf.io.gfile.glob('gs://dream_tfrecords/training_data_*.tfrecords')
```

```
from tensorflow.python.data.experimental import AUTOTUNE
```

```
_feature_description = {
    "sequence": tf.io.FixedLenFeature([], tf.string),
    "expression": tf.io.FixedLenFeature([], tf.float32),
    }
```

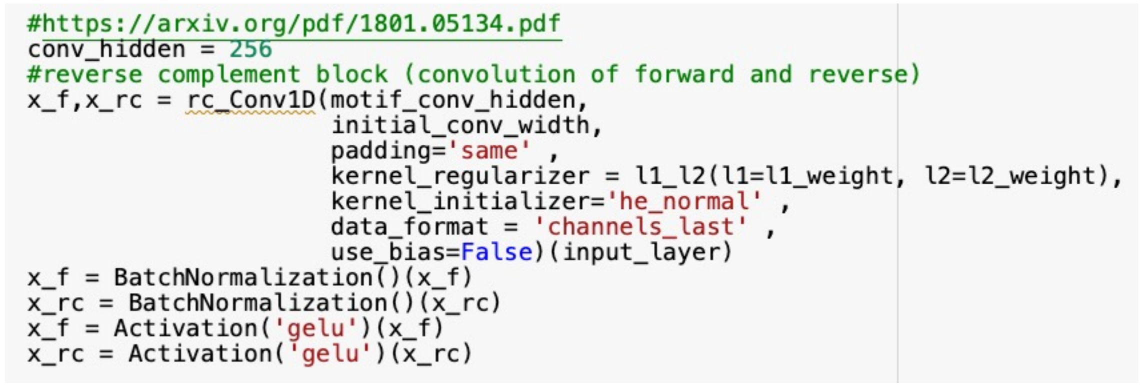
```
def _parse_data(unparsed_example):
    return tf.io.parse_single_example(unparsed_example, _feature_description)
```

```
def _bytestring_to_seq(parsed_example):
    byte_string = parsed_example['sequence']
    #seq = tf.sparse.to_dense(byte_string)
    seq = tf.io.parse_tensor(byte_string, bool)
    seq = tf.reshape(seq, shape=(110,4))
    exp = tf.cast(parsed_example["expression"], tf.float32)
    return seq, exp
```

```
def load_and_extract_sequences(filepath):
  option_no_order = tf.data.Options()
  option_no_order.experimental_deterministic = False

  dataset = tf.data.TFRecordDataset(filepath, num_parallel_reads=AUTOTUNE)
  dataset = dataset.with_options(option_no_order)
  dataset = dataset.map(_parse_data, num_parallel_calls=AUTOTUNE)
  dataset = dataset.map(_bytestring_to_seq, num_parallel_calls=AUTOTUNE) # .cache()
  return dataset
```

```
dataset = load_and_extract_sequences(filenames)
```

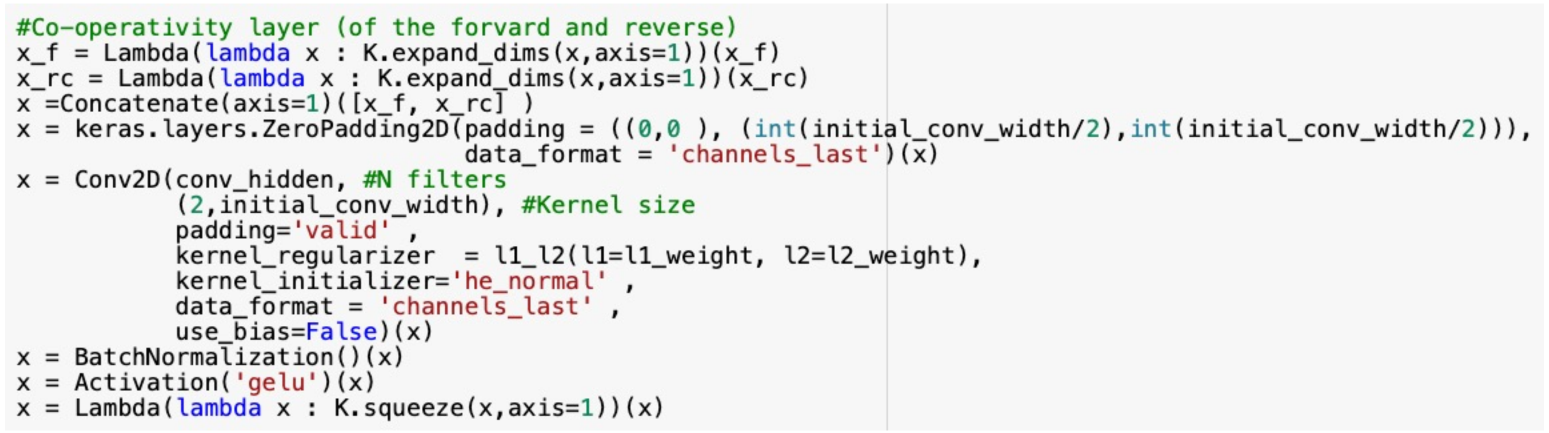
1. We created convolutions of forward and reverse sequences using Conv1D.



```
#https://arxiv.org/pdf/1801.05134.pdf
conv_hidden = 256
#reverse complement block (convolution of forward and reverse)
x_f,x_rc = rc_Conv1D(motif_conv_hidden,
                     initial_conv_width,
                     padding='same' ,
                     kernel_regularizer = l1_l2(l1=l1_weight, l2=l2_weight),
                     kernel_initializer='he_normal' ,
                     data_format = 'channels_last' ,
                     use_bias=False)(input_layer)
x_f = BatchNormalization()(x_f)
x_rc = BatchNormalization()(x_rc)
x_f = Activation('gelu')(x_f)
x_rc = Activation('gelu')(x_rc)
```

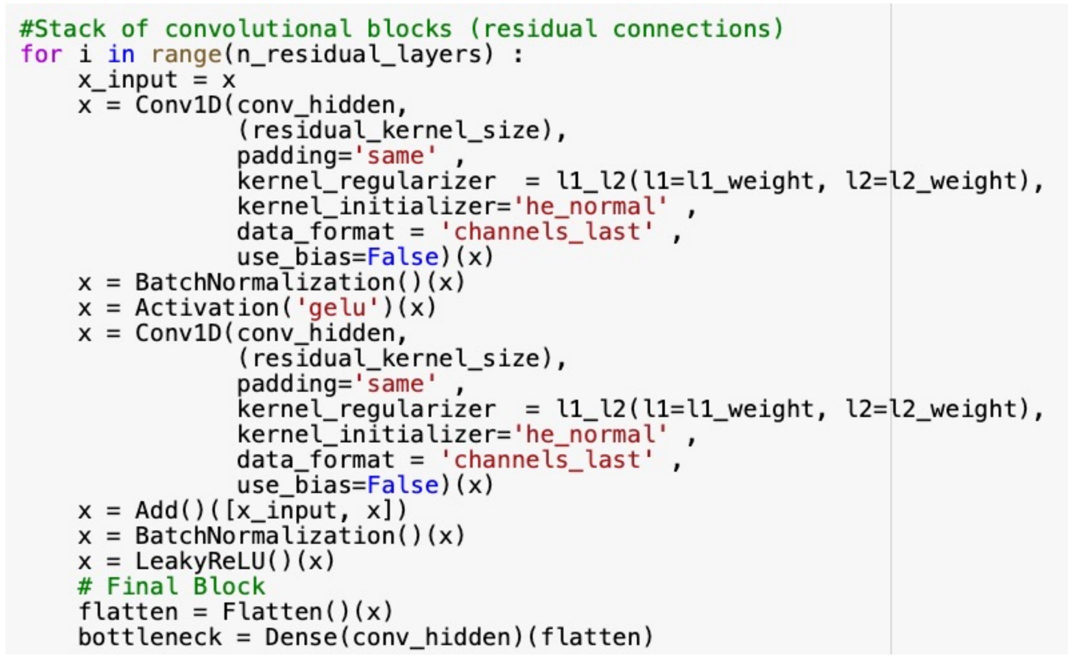
*Custom build layers

2. Convolutions with reverse and forward sequences were concatenate and fed to Conv2D.



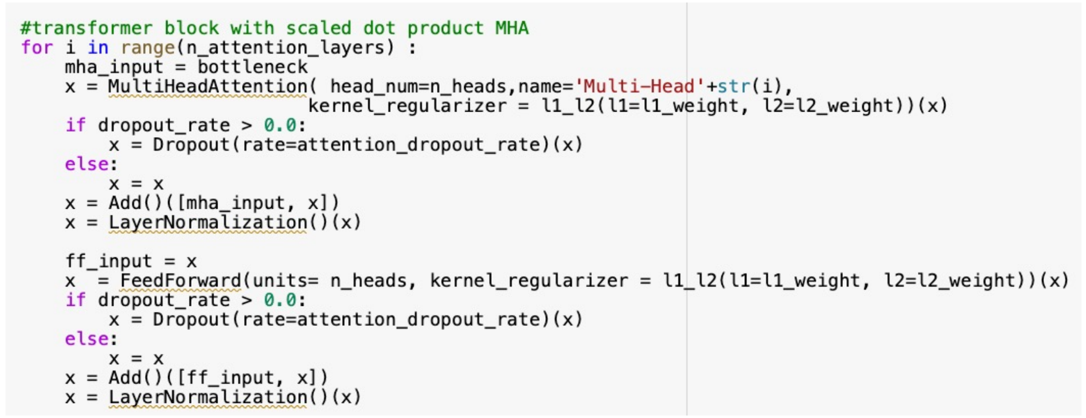
```
#Co-operativity layer (of the forvard and reverse)
x_f = Lambda(lambda x : K.expand_dims(x,axis=1))(x_f)
x_rc = Lambda(lambda x : K.expand_dims(x,axis=1))(x_rc)
x =Concatenate(axis=1)([x_f, x_rc] )
x = keras.layers.ZeroPadding2D(padding = ((0,0 ), (int(initial_conv_width/2),int(initial_conv_width/2))),
                               data_format = 'channels_last')(x)
x = Conv2D(conv_hidden, #N filters
           (2,initial_conv_width), #Kernel size
           padding='valid' ,
           kernel_regularizer  = l1_l2(l1=l1_weight, l2=l2_weight),
           kernel_initializer='he_normal' ,
           data_format = 'channels_last' ,
           use_bias=False)(x)
x = BatchNormalization()(x)
x = Activation('gelu')(x)
x = Lambda(lambda x : K.squeeze(x,axis=1))(x)
```

3. Stack of convolution blocks with residual connections.



```python
#Stack of convolutional blocks (residual connections)
for i in range(n_residual_layers) :
    x_input = x
    x = Conv1D(conv_hidden,
               (residual_kernel_size),
               padding='same' ,
               kernel_regularizer  = l1_l2(l1=l1_weight, l2=l2_weight),
               kernel_initializer='he_normal' ,
               data_format = 'channels_last' ,
               use_bias=False)(x)
    x = BatchNormalization()(x)
    x = Activation('gelu')(x)
    x = Conv1D(conv_hidden,
               (residual_kernel_size),
               padding='same' ,
               kernel_regularizer  = l1_l2(l1=l1_weight, l2=l2_weight),
               kernel_initializer='he_normal' ,
               data_format = 'channels_last' ,
               use_bias=False)(x)
    x = Add()([x_input, x])
    x = BatchNormalization()(x)
    x = LeakyReLU()(x)
    # Final Block
    flatten = Flatten()(x)
    bottleneck = Dense(conv_hidden)(flatten)
```

4. The output of the stacked CNN was flattened and and embendded, this new input was fed to transformer with multi-head attention.



```python
#transformer block with scaled dot product MHA
for i in range(n_attention_layers) :
    mha_input = bottleneck
    x = MultiHeadAttention( head_num=n_heads,name='Multi-Head'+str(i),
                            kernel_regularizer = l1_l2(l1=l1_weight, l2=l2_weight))(x)
    if dropout_rate > 0.0:
        x = Dropout(rate=attention_dropout_rate)(x)
    else:
        x = x
    x = Add()([mha_input, x])
    x = LayerNormalization()(x)

    ff_input = x
    x  = FeedForward(units= n_heads, kernel_regularizer = l1_l2(l1=l1_weight, l2=l2_weight))(x)
    if dropout_rate > 0.0:
        x = Dropout(rate=attention_dropout_rate)(x)
    else:
        x = x
    x = Add()([ff_input, x])
    x = LayerNormalization()(x)
```

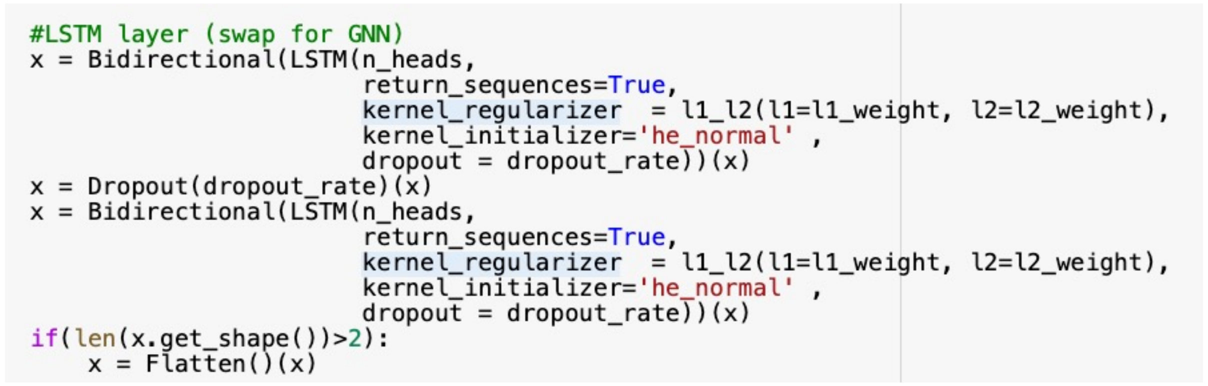
Multi-Head Attention is defined as:

$$\mathrm{MultiHead}(Q, K, V) = \mathrm{Concat}(head_1, \ldots, head_h)W^O$$

where $head_i = \mathrm{Attention}(QW_i^Q, KW_i^K, VW_i^V)$.

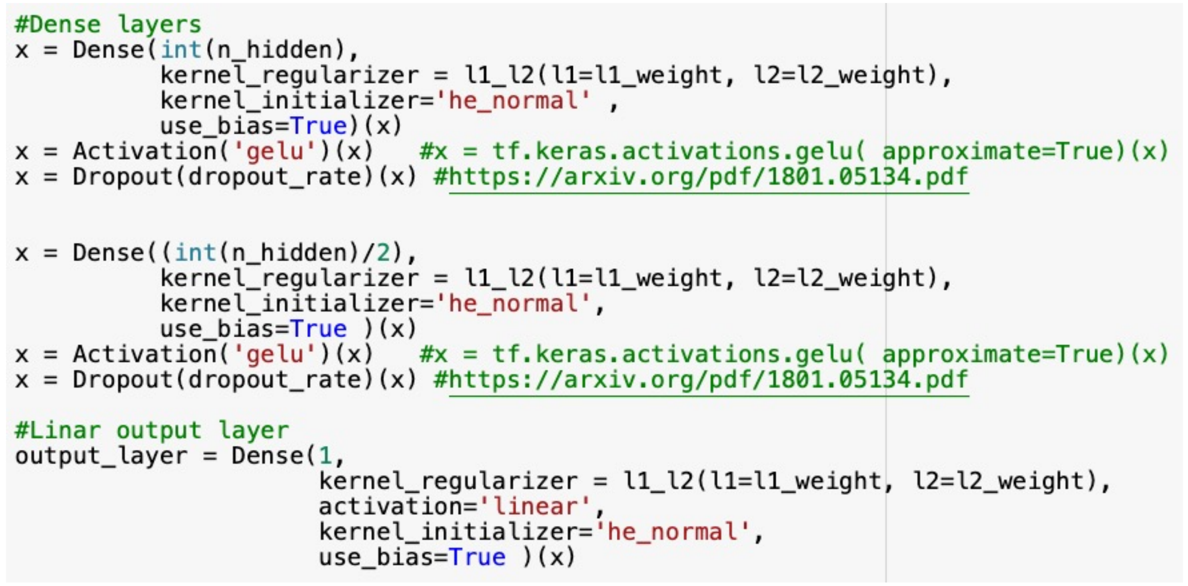
*Custom build layers

5. Later the output of transformers were passed to the stacked bidirectional LSTM that preserves the infromation in a bidirectional fashion.



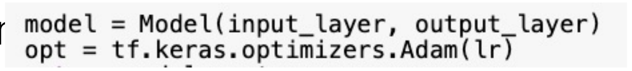
```
#LSTM layer (swap for GNN)
x = Bidirectional(LSTM(n_heads,
                       return_sequences=True,
                       kernel_regularizer  = l1_l2(l1=l1_weight, l2=l2_weight),
                       kernel_initializer='he_normal' ,
                       dropout = dropout_rate))(x)
x = Dropout(dropout_rate)(x)
x = Bidirectional(LSTM(n_heads,
                       return_sequences=True,
                       kernel_regularizer  = l1_l2(l1=l1_weight, l2=l2_weight),
                       kernel_initializer='he_normal' ,
                       dropout = dropout_rate))(x)
if(len(x.get_shape())>2):
    x = Flatten()(x)
```

6. After LSTM we put additional stack layers to decrease the dimensionality into the linear output.



```
#Dense layers
x = Dense(int(n_hidden),
          kernel_regularizer = l1_l2(l1=l1_weight, l2=l2_weight),
          kernel_initializer='he_normal' ,
          use_bias=True)(x)
x = Activation('gelu')(x)    #x = tf.keras.activations.gelu( approximate=True)(x)
x = Dropout(dropout_rate)(x) #https://arxiv.org/pdf/1801.05134.pdf


x = Dense((int(n_hidden)/2),
          kernel_regularizer = l1_l2(l1=l1_weight, l2=l2_weight),
          kernel_initializer='he_normal',
          use_bias=True )(x)
x = Activation('gelu')(x)    #x = tf.keras.activations.gelu( approximate=True)(x)
x = Dropout(dropout_rate)(x) #https://arxiv.org/pdf/1801.05134.pdf

#Linar output layer
output_layer = Dense(1,
                     kernel_regularizer = l1_l2(l1=l1_weight, l2=l2_weight),
                     activation='linear',
                     kernel_initializer='he_normal',
                     use_bias=True )(x)
```

7. Optimization for the model was chose Adar

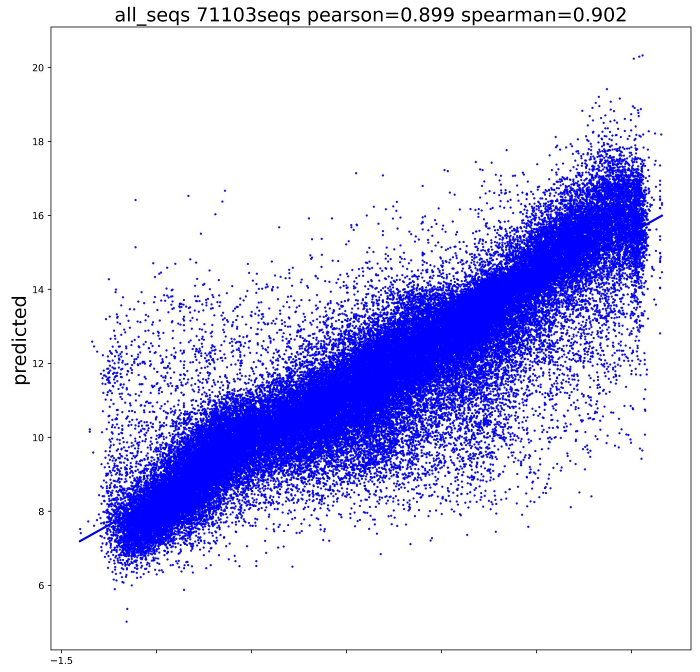


```
model = Model(input_layer, output_layer)
opt = tf.keras.optimizers.Adam(lr)
```

State of the art models

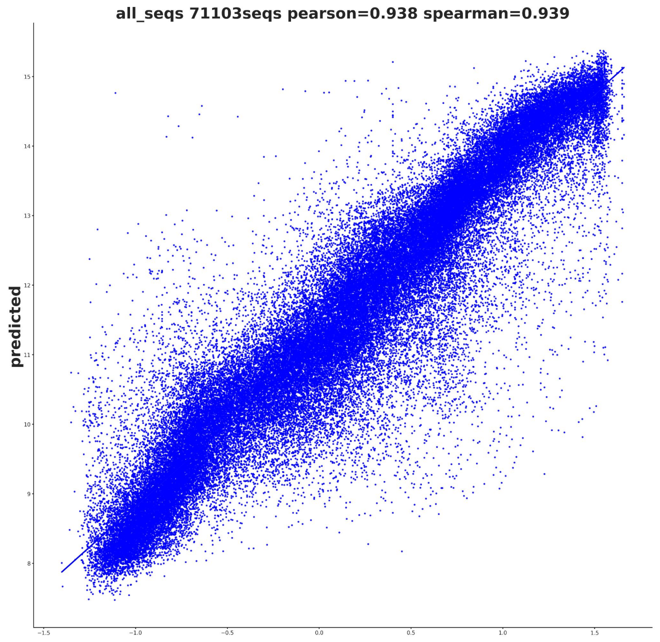
Biochemical model

(de Boer et al. 2020)



Transformer Neural Network

(Vaishnav et al. 2022)



https://drive.google.com/file/d/15OeHgy-x3R9ZMBENoDT6zfq4KLgKcmfn/view