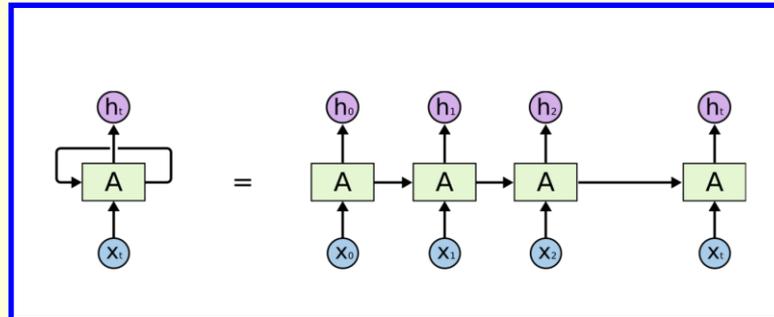


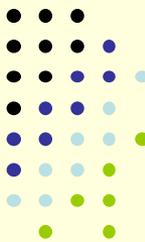
Deep Learning by Example on Biowulf

Class #2: Analysis of biological sequence data using RNNs and 1D-CNNs

Gennady Denisov, PhD



Class #2 Goals



DL networks to be discussed:

- Recurrent Neural Networks (RNNs)
- 1D Convolutional Neural Networks (**1D-CNNs**)

Popular non-bio applications:

- natural language processing
- text document classification
- time series classification, comparison and forecasting
- ...

Standard non-bio RNN benchmark: IMDB movie review sentiment prediction:



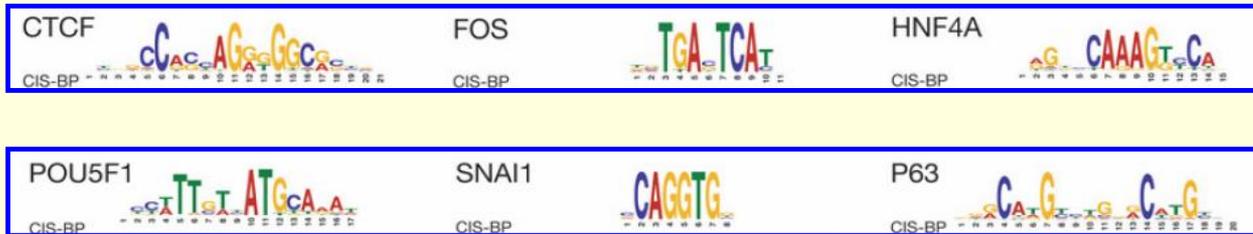
Bio example #2: predicting a function of the non-coding DNA

ATTCCCGTAATCTACGATTAAGTCACAACCAACC

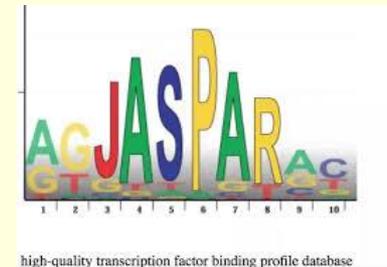


[010011010100111010...110]

Motifs:



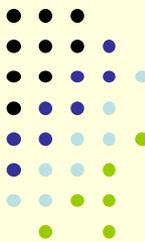
Motif database



Distinctive features of the biological example:

- 1) a vector of binary labels is assigned to each data sample
- 2) need to identify the sequence motifs

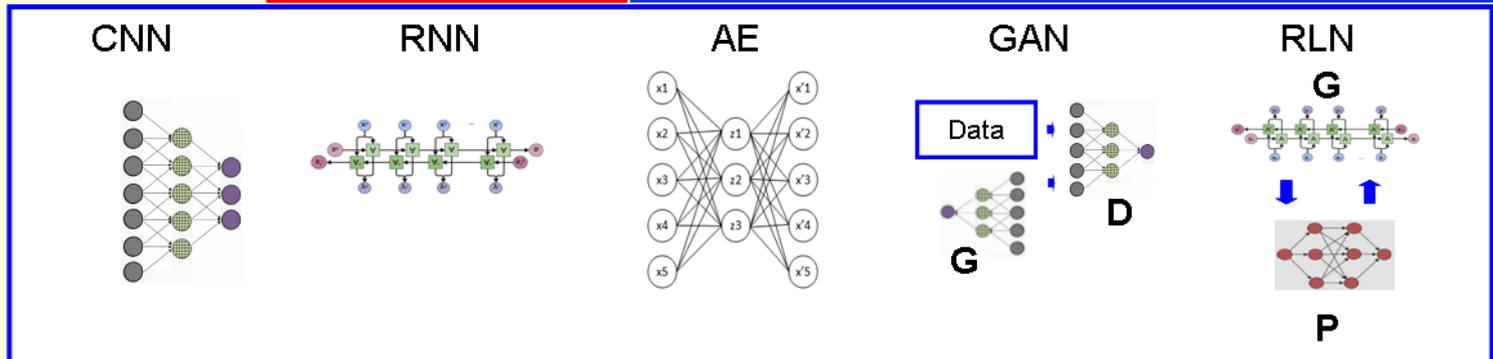
Examples overview



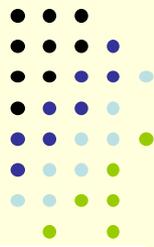
#	1	2	3	4	5
Biological Application	Bioimage segmentation/ fly brain connectome project	Genomics/ predicting the function of <u>non-coding DNA</u> (~98%)			
Network type	Convolutional Neural Network	Recurrent Neural Network			
ML type	Supervised	Supervised			

Examples #1 and #2 are complementary one to another in a number of ways, including:

- different approaches to building a network: **branched** network (#1) => **Functional API**
unbranched network (#2) => **Sequential construct**
- different approaches to training a network: **limited** ground truth data, labeled manually (#1) => **augmentation**
plenty of the ground truth data, generated using NGS (#2) => **no augmentation**



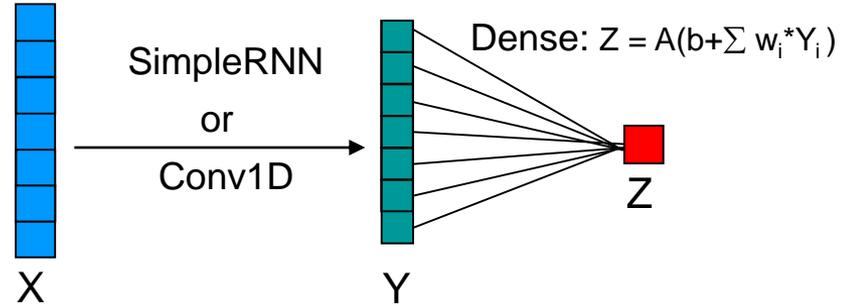
Warming up: a simple prototype example



tensors, units, layers, Dense, SimpleRNN, Conv1D, parameters, hyperparameters, RNN memory

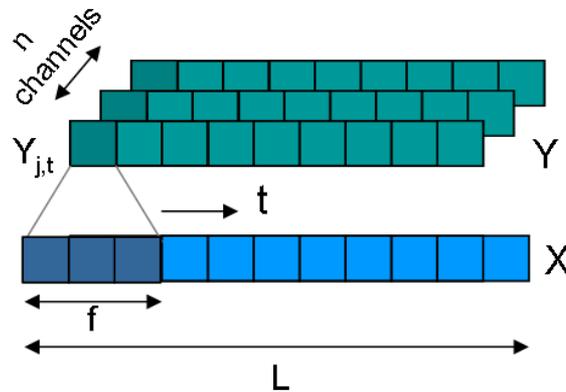
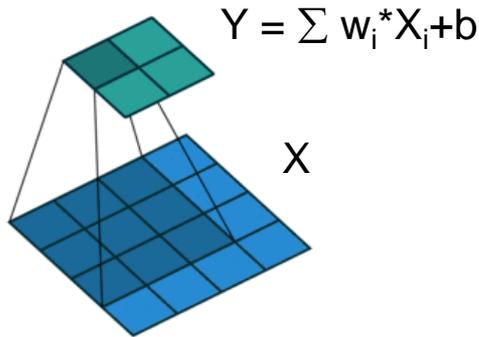
Input: a set of training sequences of 0's and 1's and **binary** labels assigned to each sequence, depending on whether or not a certain motif is present in the sequence.

Model:



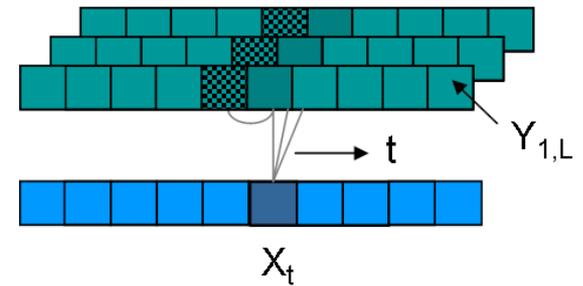
Task: predict the occurrence of the **unknown** motif in new sequences.

Example: 01011100101



$$Y_{j,t} = A(b_j + w_{X1,j} \cdot X_{t-1} + w_{X2,j} \cdot X_t + w_{X3,j} \cdot X_{t+1})$$

$j=1, \dots, n$



$$Y_{j,t} = A(b_j + w_{XY,j} \cdot X_t + \sum_{k=1}^n w_{YY,j,k} \cdot Y_{k,t-1})$$

Conv2D

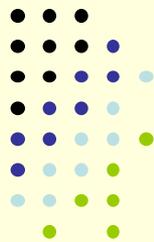
- parameters: w_i, b
- hyperparameters: $f = \text{filter/kernel size} (=3)$, padding (= "valid")

Conv1D

- parallelizable
- memoryless
- independent channels
- output: all the channel units
- output shape: $(n, L-f+1)$
- # params = $(f+1)*n$

SimpleRNN

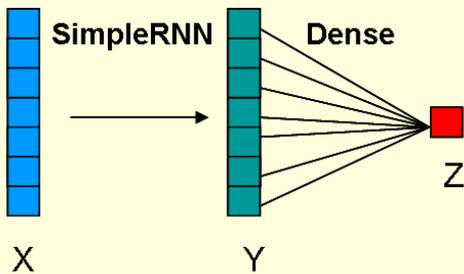
- sequential
- has memory
- interacting channels
- output: only last units $Y_{1,L}, \dots, Y_{n,L}$
- output shape: $(n, 1)$
- # params = $n + n + n*n = 2n + n^2$



The SimpleRNN training code

Sequential construct, SimpleRNN layer, motif, metrics

<https://github.com/keras-team/keras>



Header:

- general Python imports
- Dense, SimpleRNN
- Sequential

Get data

- generate "synthetic" data
- a motif to search for
- training samples x_{train} and binary labels y_{train}

Define a model

- Sequential construct approach
- compile, loss, optimizer, metrics

Run the model

- fit, checkpoint, epoch, callbacks

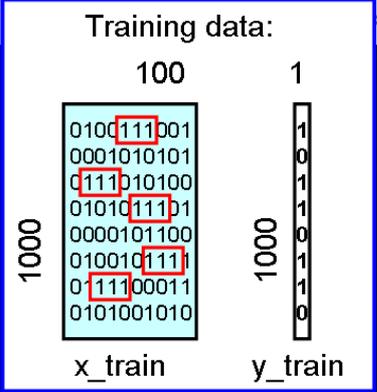
```
Select denisovga@biowulf:/data/denisovga/1_DL_Course/0_Intro
#!/usr/bin/env python

# Imports
import re
import numpy as np
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense
from keras.callbacks import ModelCheckpoint
from keras import metrics

# Get data
num_seq = 1000
seq_len = 10
np.random.seed(1)
x_train = np.round(np.random.uniform(0, 1, (num_seq, seq_len, 1)))
x_char = x_train[:, :, 0].astype(int).astype(str)
x_str = np.array([''.join(x_char[i].tolist()) \
                  for i in range(num_seq)])
motif = "111"
y_train = np.array([np.where(re.search(motif, x_str[i]), 1, 0) \
                    for i in range(num_seq)])

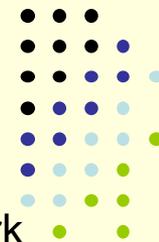
# Define a model
model = Sequential()
num_channels = 5
model.add(SimpleRNN(num_channels, \
                    input_shape=(seq_len, 1)))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', \
              optimizer='sgd', metrics=['acc'])

# Run the model on the data
checkpointer = ModelCheckpoint(filepath="simplernn.h5")
model.fit(x_train, y_train, epochs=160, callbacks=[checkpointer])
```

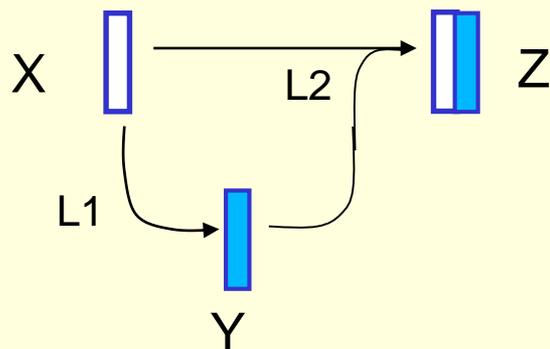


Keras metric accuracy:
how often predicted labels match true labels

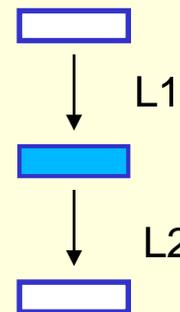
Two approaches to building models in Keras: the Functional API vs Sequential construct



Any / branched network (e.g. “mini-UNet”)



Only unbranched / sequential newtwork



```
from keras.models import Input, Model
from keras.layers import L1, L2
```

...

```
# Define a model
```

```
X = Input(...)
```

```
Y = L1(X)
```

```
Z = L2[ X, Y ]
```

```
model = Model( inputs = X, outputs = Z )
```

```
model.compile(...)
```

...

```
from keras.models import Sequential
from keras.layers import L1, L2
```

...

```
# Define a model
```

```
model = Sequential()
```

```
model.add(L1)
```

```
model.add(L2)
```

```
model.compile(...)
```

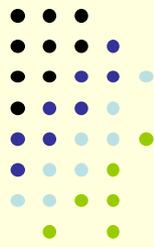
...

The Functional API approach

- explicitly uses tensor names
- applicable to any type of networks, both branched or unbranched

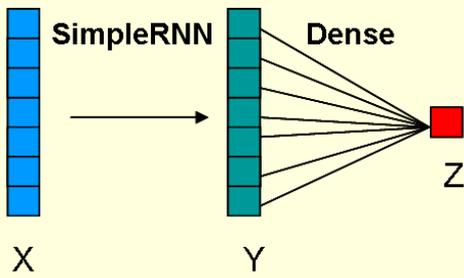
The Sequential construct approach

- does not explicitly use tensor names
- applicable only to unbranched networks
- a slightly shorter code



The SimpleRNN prediction code

summary, load weights, predict



Header:

- general python imports
- Dense, SimpleRNN
- Sequential

Get data

- generate "synthetic" data
- a motif to search for
- testing data

Define a model

- Sequential construct
- approach
- compile, loss, optimizer, metrics, summary

Run the model

- load_weights
- predict

```
denisovga@biowulf:/data/denisovga/1_DL_Course/0_Intro
#!/usr/bin/env python

# Imports
import re
import numpy as np
from keras.models import Sequential
from keras.layers import SimpleRNN, Dense
from keras.callbacks import ModelCheckpoint

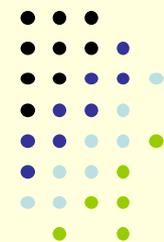
# Get data
num_seq = 10
seq_len = 10
np.random.seed(7)
x_test = np.round(np.random.uniform(0, 1, (num_seq, seq_len, 1)))
x_char = x_test[:, :, 0].astype(int).astype(str)
x_str = np.array([''.join(x_char[i].tolist()) for i in range (num_seq)])
motif = "111"
y_test = np.array([np.where(re.search(motif, x_str[i]), 1, 0) \
                      for i in range(num_seq)])

# Define a model
model = Sequential()
num_channels = 5
model.add(SimpleRNN(num_channels, \
                    input_shape=(seq_len, 1)))
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', \
              optimizer='sgd', metrics=['acc'])
model.summary()

# Run the model on the data
model.load_weights("simple_rnn.h5")
y = model.predict(x_test)
for i in range(0,num_seq):
    print("y, y_test=", int(round(y[i][0])), y_test[i])
```

Layer (type)	Output Shape	Param #
simple_rnn_1	(None, 5)	35
dense_1	(None, 1)	6
Total params: 41		
Trainable params: 41		
Non-trainable params: 0		

How to run the SimpleRNN and Conv1D code on Biowulf?



```
Select denisovga@biowulf:/data/denisovga/1_DL_Course/0_Intro
sinteractive --gres=gpu:v100:1 --mem=4g

module load DLBio/class2

ls $DLBIO_BIN
conv1d_predict.py conv1d_train.py simplernn_predict.py simplernn_train.py

simplernn_train.py
Using TensorFlow backend.
...

```

Layer (type)	Output Shape	Param #
simple_rnn_1 (SimpleRNN)	(None, 3)	15
dense_1 (Dense)	(None, 1)	4

```

Total params: 19
Trainable params: 19
Non-trainable params: 0
...
Epoch 1/1000
1000/1000 [=====] - 1s 561us/step - loss: 0.8053 - acc: 0.4820
...
Epoch 1000/1000
1000/1000 [=====] - 0s 62us/step - loss: 0.0061 - acc: 1.0000

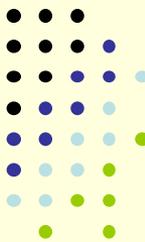
simplernn_predict.py
Using TensorFlow backend.
...
y, y_test= 0 0
y, y_test= 1 1
y, y_test= 0 0
y, y_test= 1 1
...

conv1d_train.py
...

conv1d_predict.py
...

1,39 █ A11
```

Example 2. DanQ: Predicting the function of noncoding DNA *de novo* from sequence



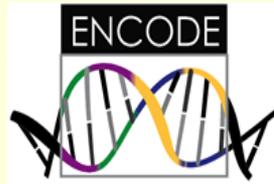
DanQ: D.Quang, X.Xie, *Nucl. Acids Res.* (2016)

DeepSEA: J.Zhou, O.G.Troyanovskaya, *Nature Methods* (2015)

<https://hpc.nih.gov/apps/DanQ.html>

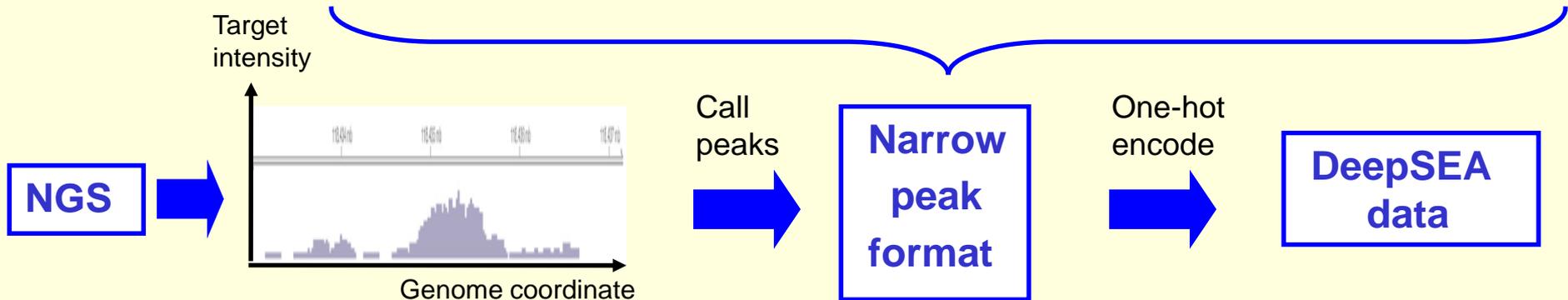
Task:

predict the targets directly from DNA sequence



Events (“targets”; total = 919 types):

- transcription factor binding sites (690 types)
- DNase I hypersensitivity sites (125 types)
- histone marks (104 types)



Models:

DeepSEA (2015) – Torch, DanQ (2016) – Keras,
Basset (2016) – Torch, Basenji (2017) – Tensorflow,
DeepBind (2015), DEEP (2015), FIDDLE (2016),
DeepMotif (2016), DeepCpG (2017), ...

DL frameworks:

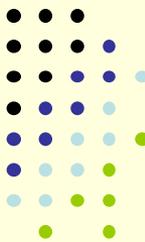
Torch,
Keras,
Tensorflow,
...

Network types:

CNN (1D),
RNN,
...

Overview of the DanQ training code

(only the main function is shown)



Imports statements,
other function definitions

Header

- parse the command line options

Get data

- training, testing and validation data

Define a model

- DanQ model
- DeepSEA model
- LSTM layer
- Dropout layer
- multi_gpu_model
- compile, loss, metrics
- optimizer

Run the model

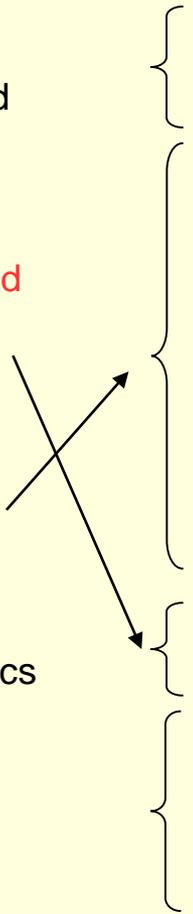
- Easlystopping
- fit

```
denisovga@biowulf:/data/denisovga/1_DL_Course/2_RNNs
if __name__ == "__main__":
    # Parse command line arguments
    opt, checkpoint_name, input_weights_file, optimizer, kernel_size \
        = parse_command_line_arguments("train")

    # Define a model
    if opt.num_gpus > 1:
        os.environ['CUDA_VISIBLE_DEVICES'] = "0,1,2,3"
        with tf.device('/cpu:0'):
            model = get_model(opt, kernel_size, summary = True, \
                pretrained_weights=input_weights_file)
            custom_checkpointer = MyCustomCallback(model, checkpoint_name, \
                verbose=opt.verbose, save_best_only=True)
            model = multi_gpu_model(model, gpus=opt.num_gpus)
    else:
        model = get_model(opt, kernel_size, summary = True, \
            pretrained_weights=input_weights_file)
        custom_checkpointer = MyCustomCallback(model, checkpoint_name, \
            verbose=opt.verbose, save_best_only=True)
    model.compile(loss='binary_crossentropy', \
        optimizer=Optimizer, metrics = ['acc'])

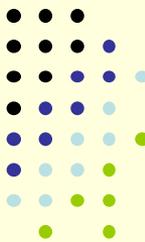
    # Get data
    X_train, y_train, X_valid, y_valid = load_data(opt)

    # Run the model
    callbacks_list = [custom_checkpointer]
    if opt.lr_schedule:
        global_lr = opt.learning_rate
        callbacks_list.append(LearningRateScheduler(step_decay))
    model.fit(X_train, y_train, batch_size=opt.batch_size, shuffle=True, \
        epochs=opt.num_epochs, validation_data=(X_valid, y_valid), \
        callbacks=callbacks_list)
```



Available data

one-hot encoding; training, validation, and testing data



One-hot encoding:

ATTCCCGTAATCTACGATTAAGTCACAACCAAACC



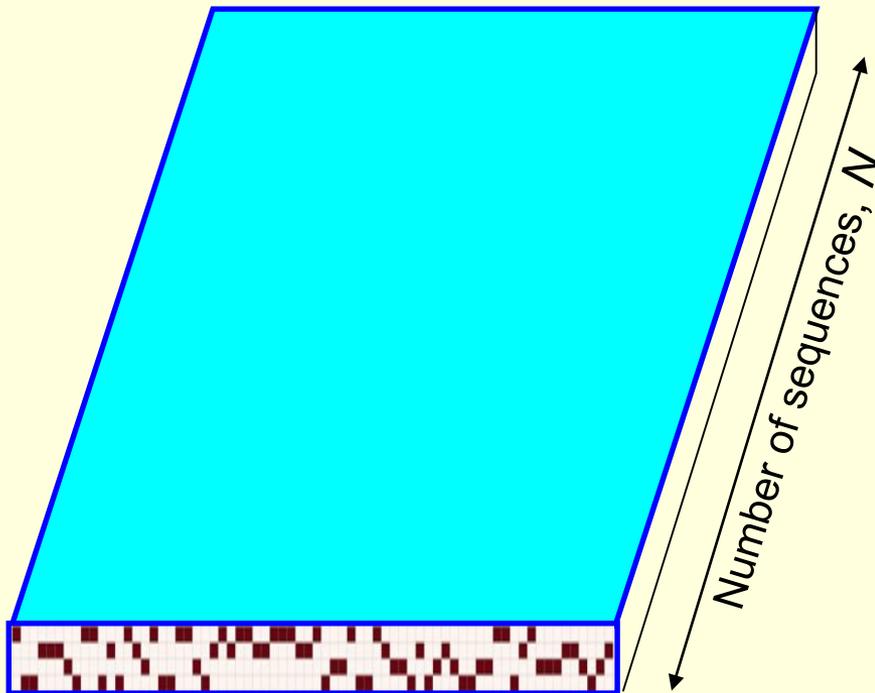
Training data: $N = 4.4 \text{ M} \Rightarrow$ adjust parameters

Validation data: $N = 8 \text{ K} \Rightarrow$ tune hyperparameters

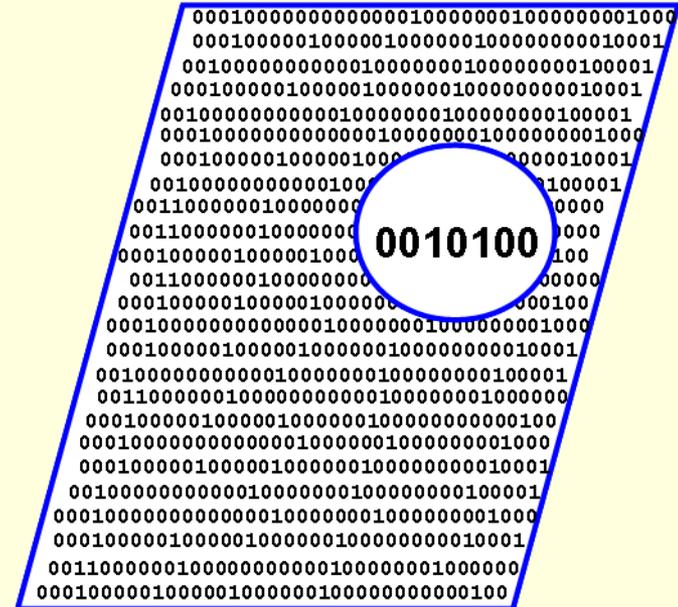
Testing data: $N = 455 \text{ K} \Rightarrow$ test predicted labels

Sequence length (=1000 bp)

Num. targets (= 919)



X (data) : $4 \times 1000 \times N$

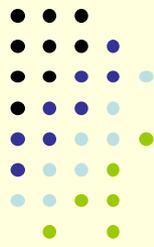


y (labels): $919 \times N$

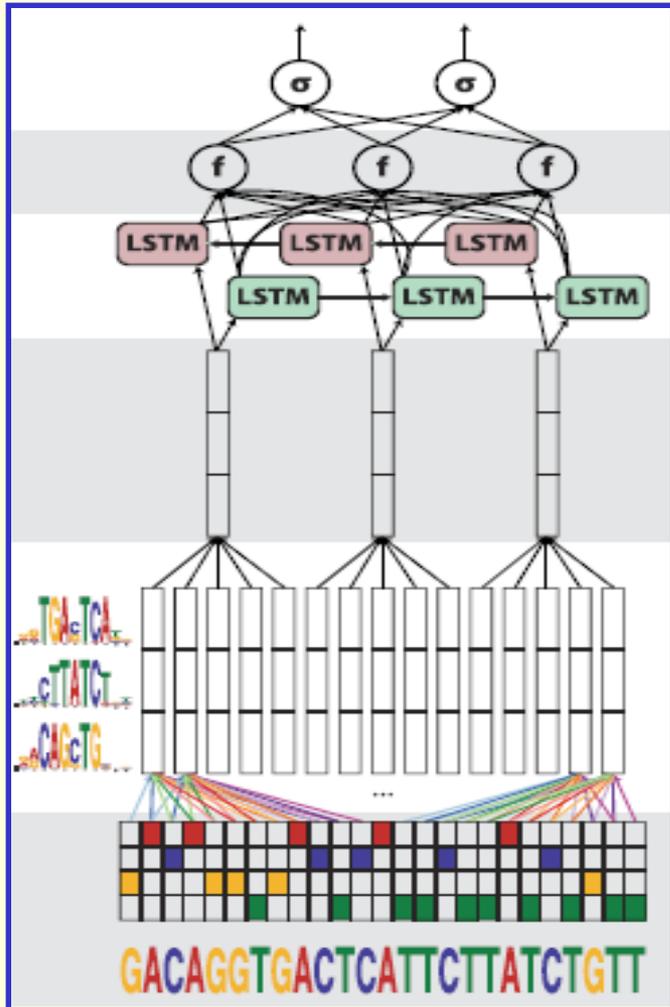
An overview of the DanQ model

LSTM, BLSTM, MaxPooling1D, Dropout

D.Quang, X.Xie, Nucl. Acids Res. (2016)



Output



Input

Activation("sigmoid")

Activation("relu"), Dense(...)

Dropout (0.5), Flatten(), Dense(...)

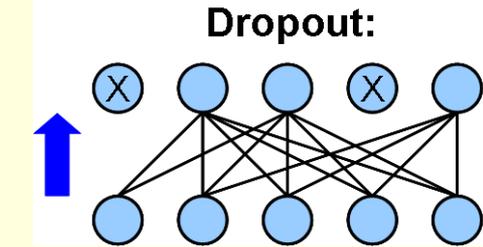
Bidirectional(LSTM(320, ...)) # recurrent layer

Dropout(0.2)

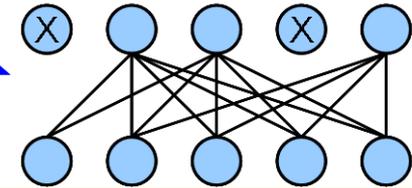
MaxPooling1D(pool_size=13, ...)

Conv1D(320, 26, ...) # allows computing PWMs

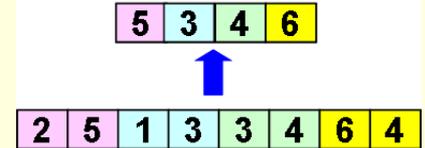
One hot encoding



Dropout:



MaxPooling1D:

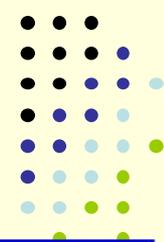


BLSTM: # params = 1,640,960

Conv1D: # params = 33,600

Long Short-Term Memory (LSTM) layer

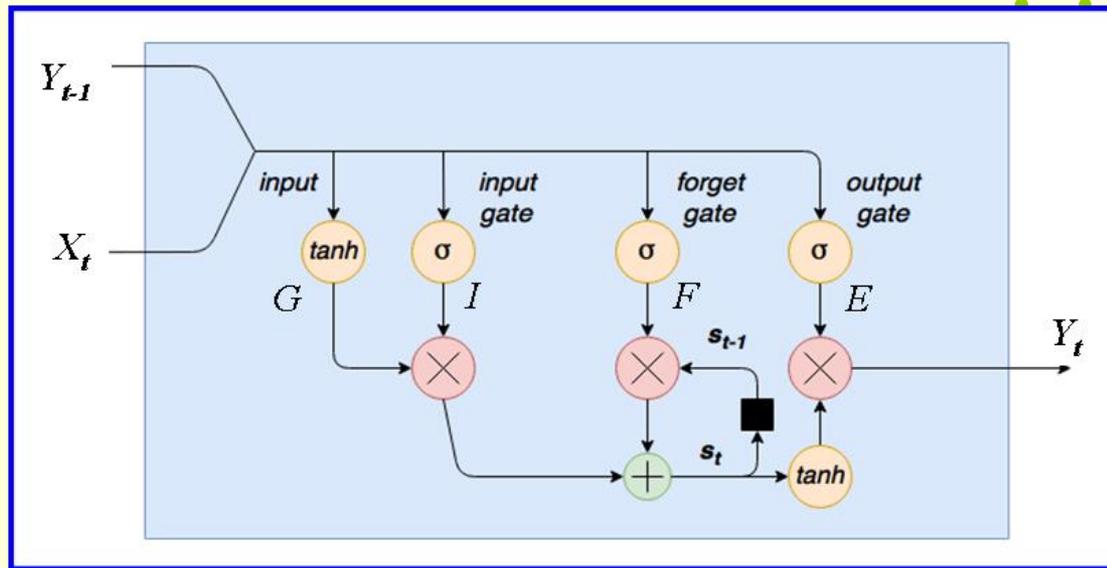
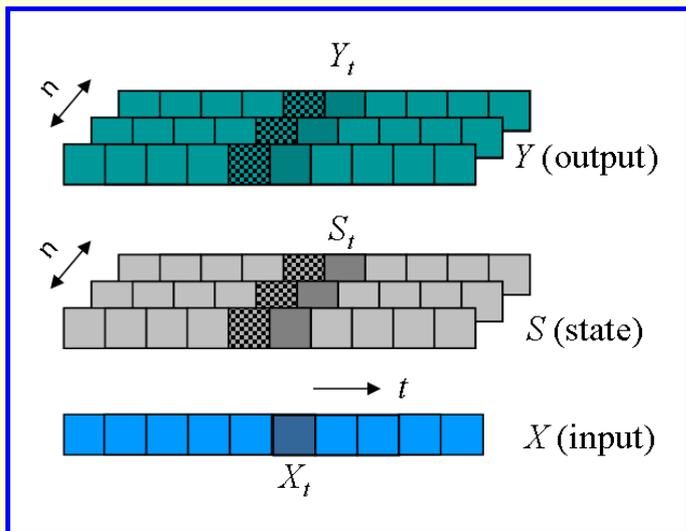
S Hochreiter, J Schmidhuber, *Neural computation* 9, p.1735 (1997)
<https://adventuresinmachinelearning.com/keras-lstm-tutorial>



Why not SimpleRNN?

- the “vanishing gradients” issue:

$$\nabla J(w) \rightarrow 0$$



$$1) S_t = S_{t-1} \otimes F(X_t, Y_{t-1}) + G(X_t, Y_{t-1}) \otimes I(X_t, Y_{t-1})$$

$$2) Y_t = \tanh(S_t) \otimes E(X_t, Y_{t-1})$$

SimpleRNN: one step

$$1) X_t, Y_{t-1} \Rightarrow Y_t$$

LSTM: two steps

$$1) X_t, Y_{t-1}, S_{t-1} \Rightarrow S_t$$

$$2) X_t, Y_{t-1}, S_t \Rightarrow Y_t$$

$$G(X_t, Y_{t-1}) = \tanh(b_G + w_{XG} \cdot X_t + w_{YG} \cdot Y_{t-1})$$

$$I(X_t, Y_{t-1}) = \sigma(b_I + w_{XI} \cdot X_t + w_{YI} \cdot Y_{t-1})$$

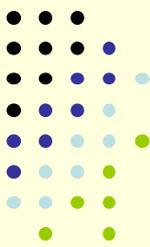
$$F(X_t, Y_{t-1}) = \sigma(b_F + w_{XF} \cdot X_t + w_{YF} \cdot Y_{t-1})$$

$$E(X_t, Y_{t-1}) = \sigma(b_E + w_{XE} \cdot X_t + w_{YE} \cdot Y_{t-1})$$

\otimes = elementwise multiplication; σ = sigmoid activation

$$\# \text{ parameters} = 4 \cdot (2n + n^2)$$

Gradient descent-based optimizers: Batch GD, SGD and Mini-batch GD



<http://ruder.io/optimizing-gradient-descent>

<https://keras.io/models/model/#fit>

$$w_{t+1} = w_t - \gamma \cdot \nabla_w J(w_t; x, y)$$

- gradient descent formula
for updating weights

w = vector of weights

t = update #

γ = learning rate

$\nabla_w J$ = gradient of the loss with respect to weights

(x, y) = one data sample (= data item x + label y)

Vanilla, a.k.a. “Batch” Gradient Descent: # inefficient / not used for large training datasets

- average $\nabla_w J(w_t; x, y)$ over all N samples in the training dataset
- perform one update of weights per epoch

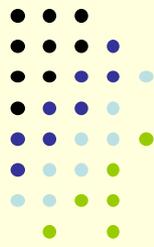
Stochastic Gradient Descent: # objective fluctuates, this may complicate convergence

- compute $\nabla_w J(w_t; x^{(k)}, y^{(k)})$ on a random sample ($k = 1, 2, \dots, N$)
- perform N updates per epoch by default, or as specified by the keyword argument steps per epoch in the method fit

Mini-batch Gradient Descent: # choosing a proper learning rate may still be difficult

- $\nabla_w J(w_t; x, y)$ averaged over a mini-batch
- $N / \text{batch_size}$ updates per epoch by default (where the **batch_size** is a hyperparameter), or as specified by the keyword argument steps per epoch in the method fit

How to run the DanQ code on Biowulf?



<https://hpc.nih.gov/apps/DanQ.html>

Using a single GPU:

```
denisovga@biowulf:/data/denisovga/1_DL_Course/2_RNNs
sinteractive --mem=64g --gres=gpu:v100:1,scratch:100 \
--cpus-per-task=14

module load danq

ls $DANQ_SRC
danq_predict.py  danq_visualize.py  models.py
danq_train.py   download_data.sh  options.py

cp -r $DANQ_DATA/* .

danq_train.py -d data [ other options ]
danq_predict.py -d data [ other options ]
danq_visualize.py -t <target_id> [ other options ]
```

Using 4 GPUs:

```
denisovga@biowulf:/data/denisovga/1_DL_Course/2_RNNs
sinteractive --mem=64g --gres=gpu:v100:4,scratch:100 \
--cpus-per-task=14

module load danq

ls $DANQ_SRC
danq_predict.py  danq_visualize.py  models.py
danq_train.py   download_data.sh  options.py

cp -r $DANQ_DATA/* .

danq_train.py -d data -g 4 [ other options ]
```

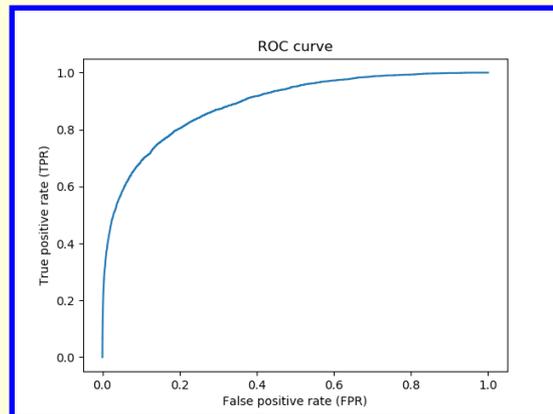
Visualizations:

Models: DanQ, DeepSEA

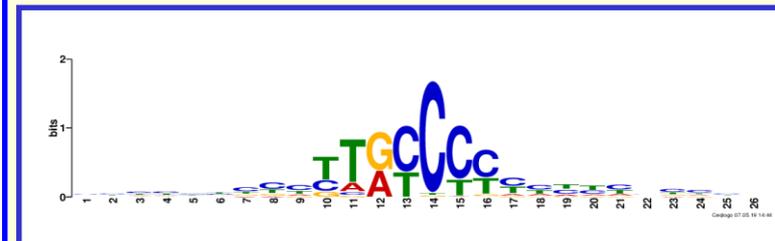
DeepSEA model:

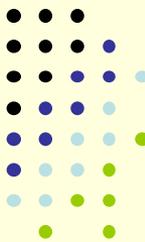
- Conv1D(320, 8, ...)
- MaxPooling
- Conv1D(480, 8, ...)
- MaxPooling
- Conv1D(960, 8, ...)
- Flatten
- Dense(919, ...)
- Activation("sigmoid")

ROC curve:



Motif sequence logo:





Conclusions

1) Further intro using simple examples

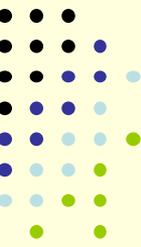
- **SimpleRNN** vs **Conv1D** layers/transformations
- the notion of the RNN network **memory** and **interacting channels**
- **Functional API** vs **Sequential** approach to building Keras models
- **metric**, model **summary** and the **# of parameters** used by layers

2) Predicting the function of a non-coding DNA

- the **DanQ** and **DeepSEA** models
- **(Bidirectional) LSTM**, **MaxPooling 1D** and **Dropout** layers
- **how to** run the **DanQ** code on **Biowulf**

3) Gradient descent-based optimizers:

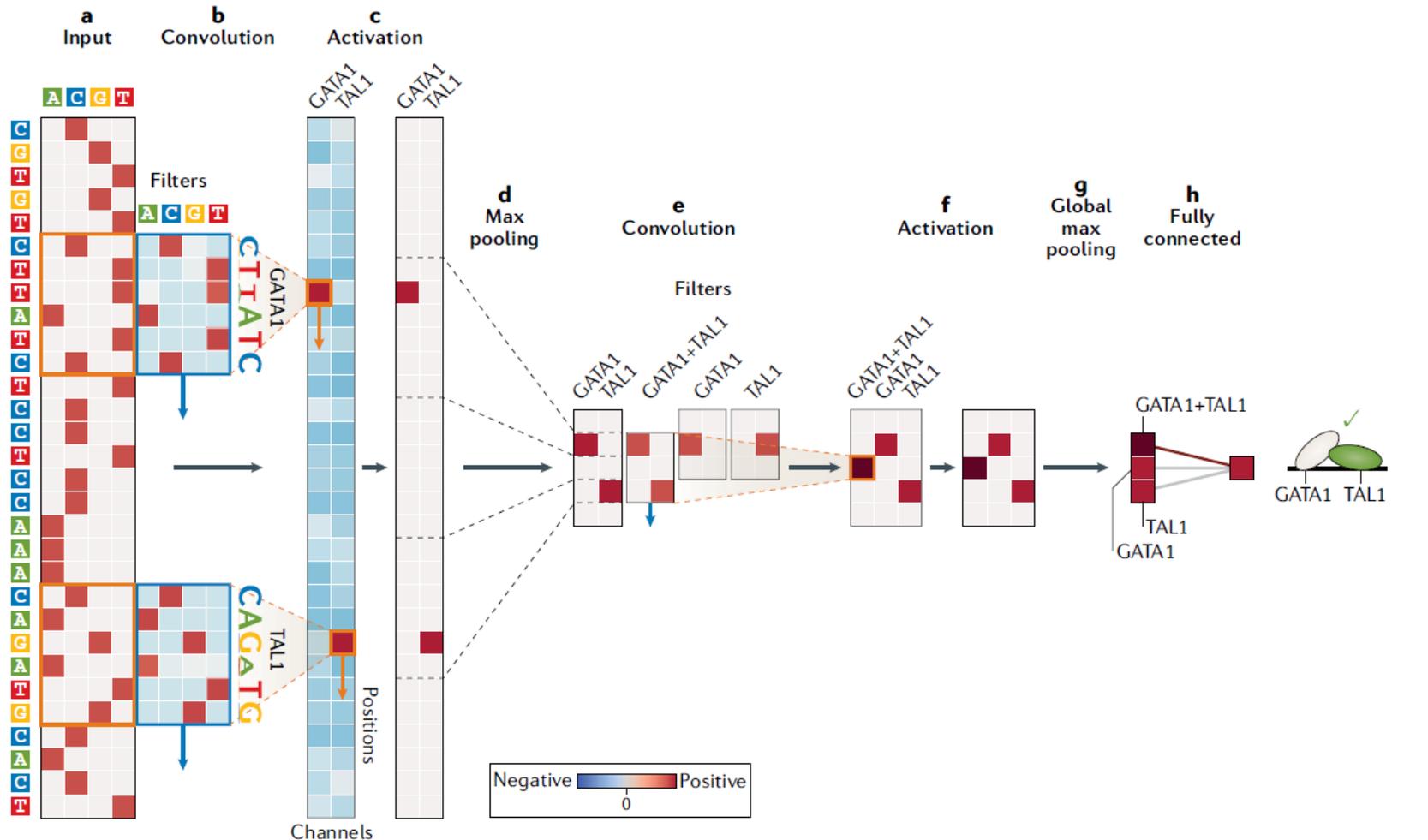
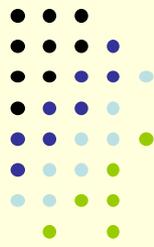
- **SGD** and **Mini-batch DG**



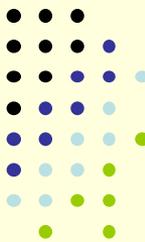
BACKUP SLIDES

Modelling transcription factor binding sites and spacing with 1D CNNs

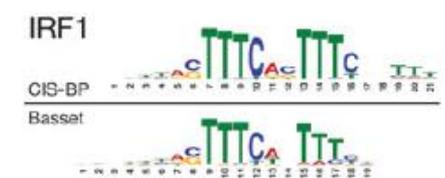
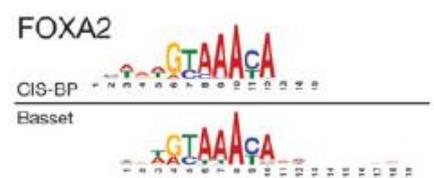
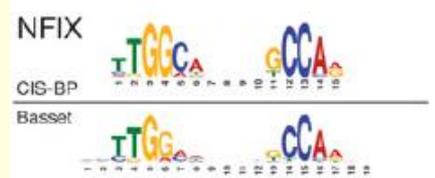
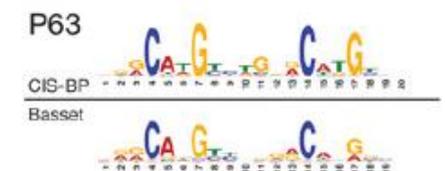
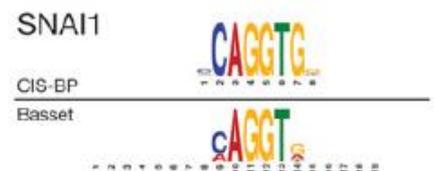
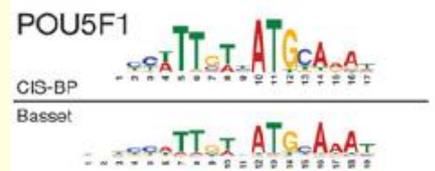
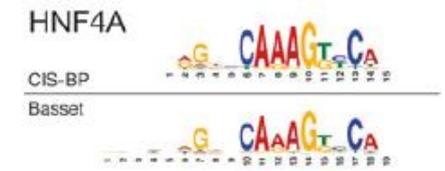
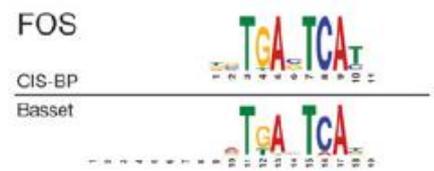
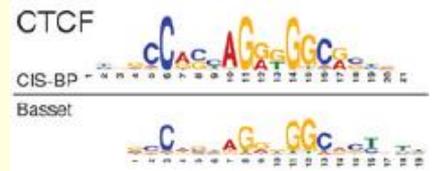
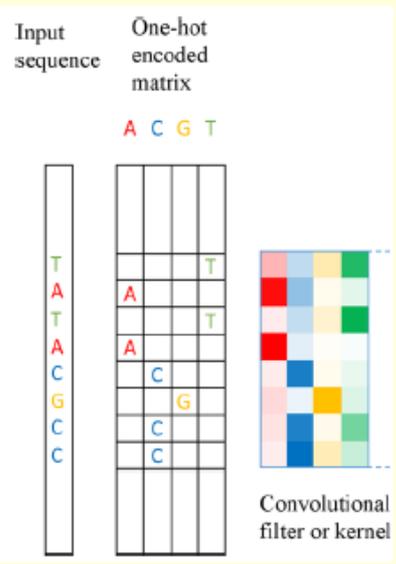
G.Eraslan et al, Nature Reviews Genetics 2019



Predicting motifs



David R. Kelley et al - Basset: ..., *Genome Res*, 2016, 26:990–999



Overall, 45% of filters could be annotated