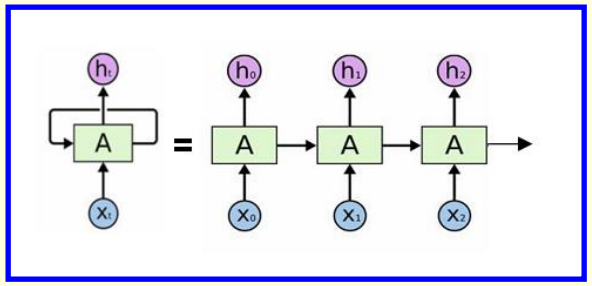
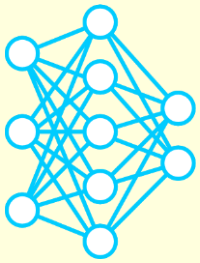
Deep Learning by Example on Biowulf

Class #2: Recurrent and 1D-Convolutional neural networks and their application to analysis of a non-coding DNA

Gennady Denisov, PhD



Class #2 Goals



DL networks to be discussed:

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

Purpose: process sequences of values

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:

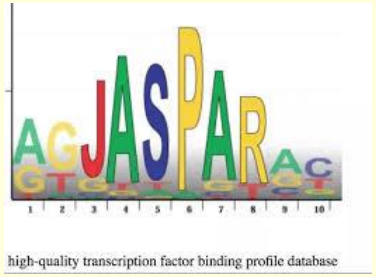
predicting the function of non-coding DNA

ATTCCCGTAATCTACGATTAAGTCACAACCAAACC

[010011010100111010…110]

Motif: **short**, recurring pattern in DNA
that is presumed to have a certain biological function.

Motif database



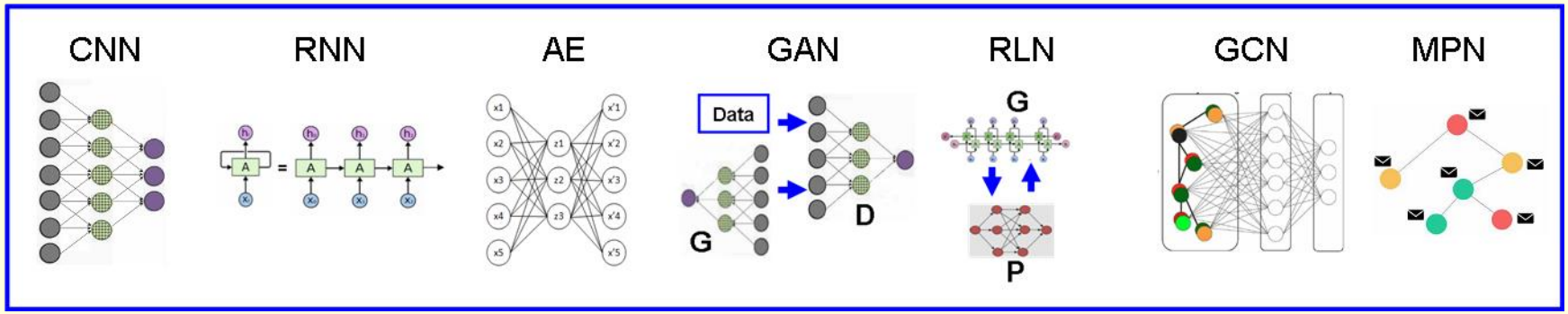
Distinctive features of the biological example:

1) a **vector of binary labels** is assigned to each data sample
2) identification of the **motif sequences**
3) exploration of the **long-range dependencies** between motifs/different parts of fragments

Examples summary

Class #	1	2	3	4	5	6	7
Bio app	Bioimage segmentation / fly brain connectome	Genomics / prediction of function of non-coding DNA					
Neural network	Convolutional	Recurrent or 1D-Convolutional					
ML type	Supervised	Supervised	Unsupervised	Unsupervised	Reinforcement	Supervised	Supervised

1) RNNs process **sequences of values,** CNNs - grid values
2) both RNNs and CNNs **share parameters** between different parts of a model, unlike MLP, where each weight is unique
3) RNNs **allow cyclic connections**, unlike CNNs or MLP / Dense networks, which are feedforward / have no cycles
4) both examples #1 and #2 **take a supervised ML approach**,
5) yet are complementary in the way their training is performed:
   #1: limited ground truth **data** ⇒ augmentation, fit_generator
   #2: plenty of ground truth **data** ⇒ no augmentation, fit

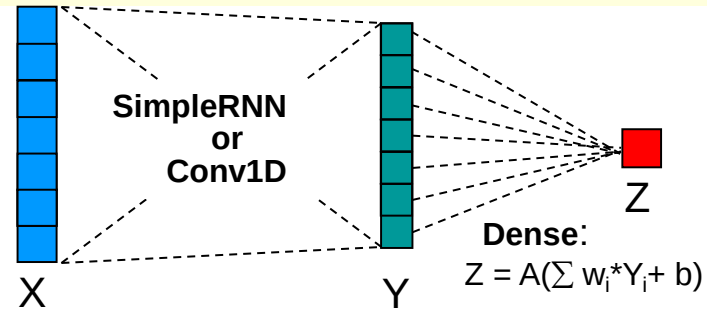


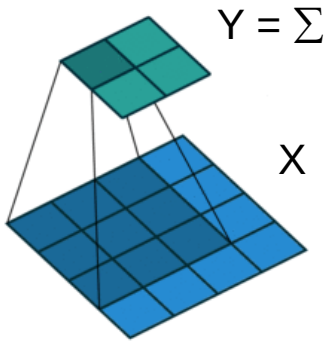
Motif detection: a prototype example #1

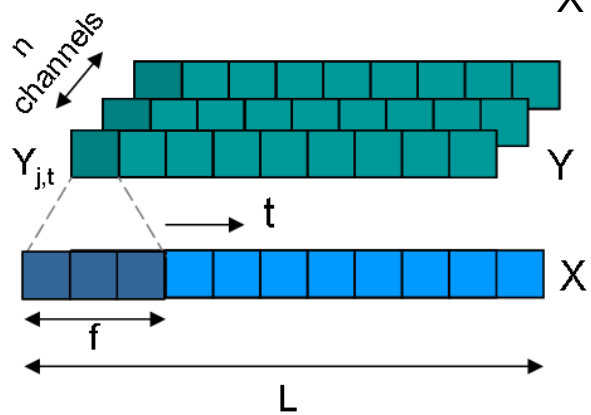
tensors, layers, parameters, hyperparameters, Dense, SimpleRNN, Conv1D, RNN memory

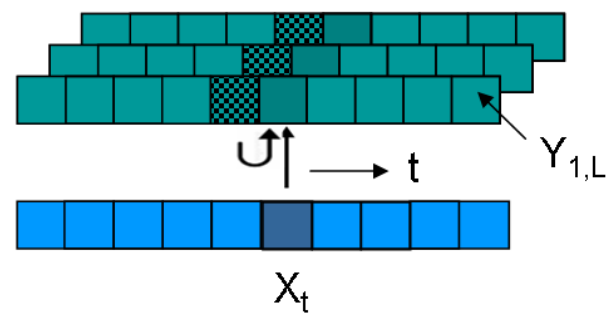
Input: a set of training **sequences** of 0's and 1's and **binary labels** assigned to each sequence, depending on whether or not a certain **(unknown) motif** is present in the sequence.
Task: train the model on the data, so that it could automatically predict labels for new sequences.
Example: 01011100101

Model:









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

$j=1,\ldots,n$

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

Conv2D

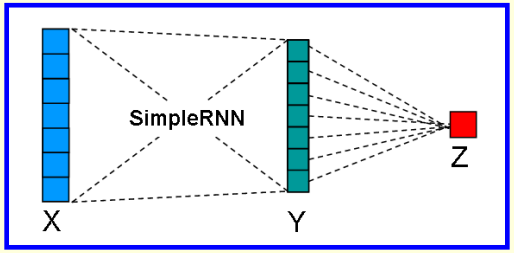
- parameters: w_i, b
- hyperparameters:
  f = filter/kernel size (=3),
  padding (= "valid")

Conv1D

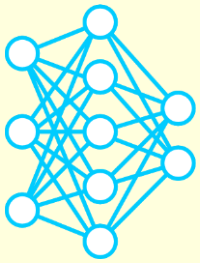
- parallelizable
- memoryless
- independent channels
- output: all the channel elements
- output shape: (n, L-f+1)
- # params = (f+1)*n

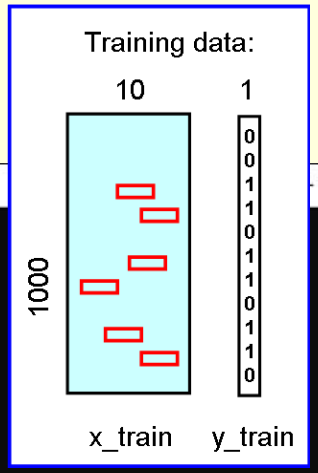
SimpleRNN

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



SimpleRNN-based code for motif detection





Header:
- general Python imports
- Dense, SimpleRNN
- Sequential

Get data
- a motif to search for
- generate synthetic data: x_train, y_train x_test, y_test

Define a model
- Sequential construct approach
- compile, loss, optimizer

Run the model
- fit, checkpoint, epoch, callbacks
- predict

denisovga@biowulf:/usr/local/apps/DLBio/class2/bin

```
#!/usr/bin/env python

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

# Get data
n_train,n_test,s_len,n_channels,n_epochs,motif = 1000,40,10,2,500,"111"
np.random.seed(7)
x_train_str = [''.join([random.choice('01') for i in range(s_len)]) \
                                            for j in range(n_train)]
x_test_str  = [''.join([random.choice('01') for i in range(s_len)]) \
                                            for j in range(n_test)]
x_train = np.reshape(np.array([[int(c) for c in x_train_str[j]] \
                    for j in range(n_train)]), [n_train,s_len,1])
x_test  = np.reshape(np.array([[int(c) for c in x_test_str[j]] \
                    for j in range(n_test )]), [n_test, s_len,1])
y_train = np.array([np.where(re.search(motif, x_train_str[i]), 1, 0) \
                    for i in range(n_train)])
y_test  = np.array([np.where(re.search(motif, x_test_str[i]), 1, 0) \
                    for i in range(n_test)])
# Define a model
model = Sequential()
model.add(SimpleRNN(n_channels, return_sequences=True, input_shape=(s_len, 1)))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='sgd')

# Run the model on the data
checkpoint_file = "mdet_simplernn.h5"
if 0: model.load_weights(checkpoint_file)
checkpointer = ModelCheckpoint(filepath = checkpoint_file)
model.fit(x_train, y_train, epochs=n_epochs, callbacks=[checkpointer])
y = model.predict(x_test)
for i in range(0,n_test):
    print("y, y_test=", int(round(y[i][0])), y_test[i])
                                                    12,55          All
```

Motif discovery: a prototype example #2

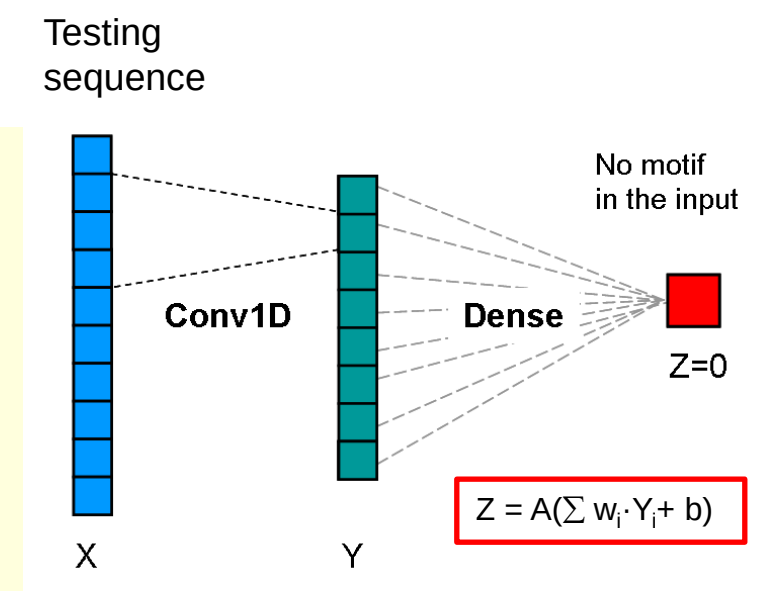
PWMs: G.D.Stormo et al, NAR 1982

Y.Ding et al, bioRxiv 2019; doi: https://doi.org/10.1101/163220

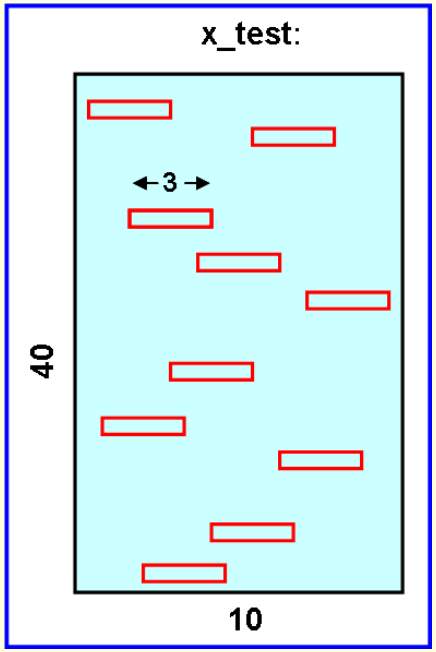
Input: 1) **pre-trained weights** from the previous example, but **using Conv1D** instead of SimpleRNN
2) a set of **testing sequences** of 0's and 1's similar to the previous example

Task: determine the **motif sequence**

Example: 01011100101



Testing data:



(Implicit) Assumptions of the heuristic approach:

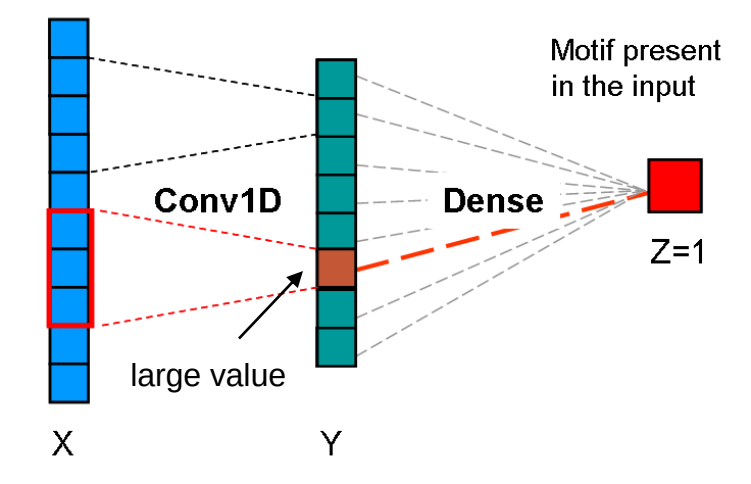
1) All the **weights** of the Dense layer are **positive**
2) The **upper bound** of the motif size is (approximately) **known**

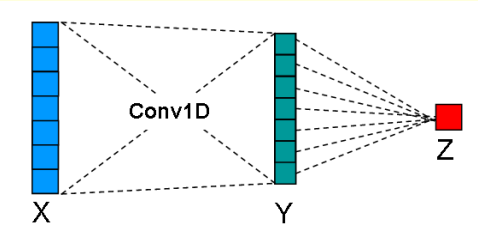
Algorithm:

1) Determine an "optimal" position of a filter in each "good" testing sequence:

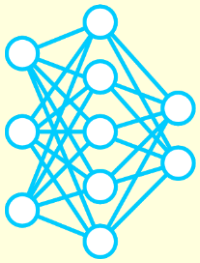
$$X_{opt} = argmax(Conv1D(X))$$

2) Determine the motif as PWM / consensus sequence over the vicinities of X_{opt} in the test sequences





Conv1D-based code for motif discovery



Header.

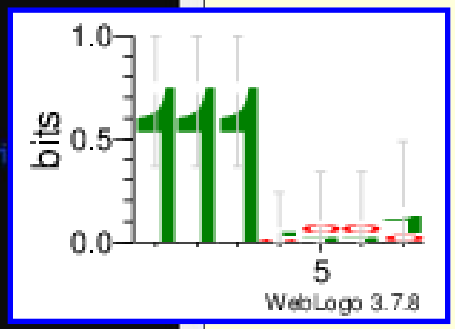
Get data.

Define a model

Run the model:
- load weights
- extract the testig data with motif
- determine an "optimal" position in test sequence
- extract instances
- compute PWM and the motif sequence

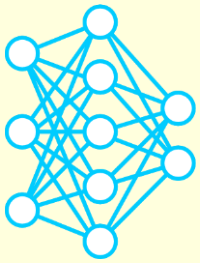
biowulf:/usr/local/apps/DLBio/class2/bin

```
#!/usr/bin/env python

# Imports
import numpy as np
from keras.models import Sequential
from keras.layers import Conv1D, Dense, Flatten
from keras import backend as K
from Bio import motifs
from Bio.Seq import Seq

# Get data
n_seq,s_len,f_size,w_size,n_filt,thresh,instances = 40,10,3,7,2,0.8,[]
np.random.seed(1)
x_test = np.round(np.random.uniform(0, 1, (n_seq, s_len, 1)))

# Define a model
model = Sequential()
model.add(Conv1D(n_filt,f_size,input_shape=(s_len,1),activation='relu'))
model.add(Flatten())
model.add(Dense(1, activation='sigmoid'))
model.compile(loss='binary_crossentropy', optimizer='sgd')

# Run the model on the data
def call(x,y,t): return "1" if x>t else ("0" if y < 1-t else "N")
model.load_weights("mdet_conv1d.h5")
z = [int(np.round(model.predict(x_test)[i,0])) for i in range(n_seq)]
x_test = np.delete(x_test,[k==0 for k in z],axis=0) # keep only data wi
output = model.layers[0].get_output_at(0)
func = K.function([model.input], [K.argmax(output, axis=1), \
                                  K.max(   output, axis=1)])
y = func([x_test])
pos = np.array([y[0][i][0] for i in range(x_test.shape[0])])
x_test = np.squeeze(x_test, axis=2).tolist()
alignment = open("01_conv1d.fa", "w")
for i in range(len(x_test)):
    pos_left, pos_right = max(0, pos[i]), min(pos[i]+w_size, s_len)
    seq = "".join([str(int(x)) for x in x_test[i][ pos_left:pos_right]])
    if len(seq) == w_size:
        instances.append(Seq(seq))
        alignment .write(">" + str(i+1) + "\n")
        alignment .write(seq + "\n")
alignment .close()
pwm = motifs.create(instances,alphabet='01').counts.normalize()
print("\nmotif=", "".join([call(x, 1-x, thresh) for x in pwm[1,:]]))
                                                          44,70          All
```



Stochastic Gradient Descent optimizer

gradient descent (GD), mini-batch GD



keras.optimizers.SGD(learning_rate =0.01, momentum=0.0, nesterov=**False**, …)

```
denisovga@biowulf:/usr/local/apps/DLBio/class2/bin
model.compile(loss='binary_crossentropy', optimizer='sgd', ...))
model.fit(x_train, y_train, epochs=1000)
                                                  2,42          Top
```

Using SGD with defalt options

$$\boxed{w_{t+1} = w_t - \gamma \cdot \nabla_w J(w_t)}$$

- basic gradient descent formula for updating weights

w = vector of weights

t = iteration #

γ = learning rate

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

(Mini-batch) Stochastic Gradient Descent:

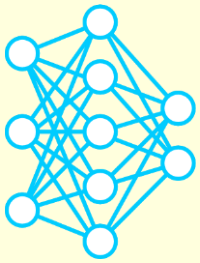
- use the $\nabla_w J(w_t\,;\,x, y)$ **averaged over a mini-batch of samples** (= data items x, labels y)
- N / **batch_size** iterations per epoch (N = total number of samples, **batch_size** =32 by default)

```
denisovga@biowulf:/usr/local/apps/DLBio/class2/bin
customized_sgd = keras.optimizers.SGD(learning_rate =1.e-4,
                                      momentum = 0.9,
                                      nesterov = True)
model.compile(loss      = 'binary_crossentropy',
              optimizer = customized_sgd , ...)
model.fit(x_train, y_train, epochs=1000, batch_size = 10)
                                                 11,59         Bot
```

Using SGD with customized options

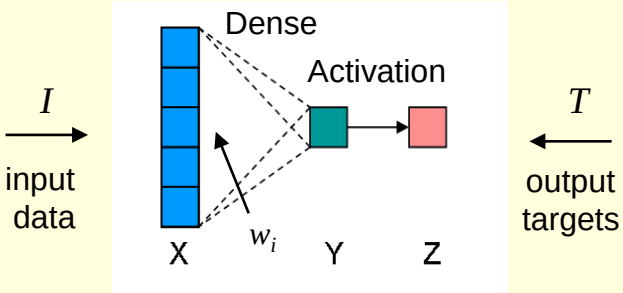
(To be continued on a backup slide)

Training a feedforward network



backpropagation, chain rule, vanishing gradient

Perceptron

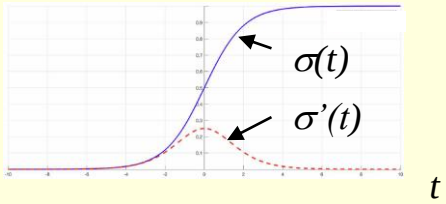


Backpropagation purpose: compute the gradient of loss

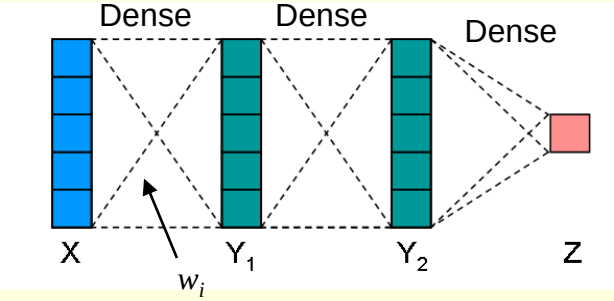
$Z(w) = \sigma(w \cdot X + b)$ (mse) loss $J = \frac{1}{2}(Z(w) - T)^2$ $\Delta J \Rightarrow \Delta w$

$$\Delta J_i = (\partial J / \partial w_i) \cdot \Delta w_i = (Z\text{-}T) \cdot (\partial Z / \partial w_i) \cdot \Delta w_i$$

$$= \underbrace{\boxed{(Z\text{-}T) \cdot \sigma'(w \cdot I + b) \cdot I_i}}_{\text{(the component of) the gradient of loss}} \cdot \Delta w_i$$



Multi-layer perceptron (MLP)

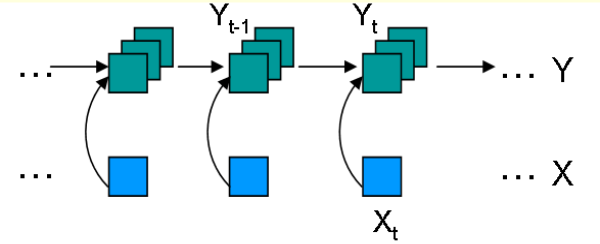


Backpropagation for MLP:

$Z = \sigma(W_3 \cdot Y_2 + b_3)$ $Y_2 = \sigma(W_2 \cdot Y_1 + B_2)$ $Y_1 = \sigma(W_1 \cdot X + B_1)$

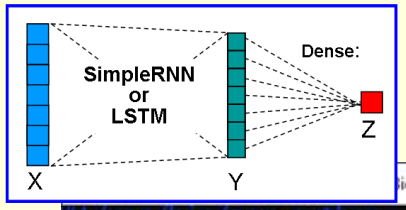
$$\Delta J_i = \underbrace{\boxed{(Z - T) \cdot \sigma'(W_3 \cdot Y_2 + B_3) \cdot W_3 \cdot \sigma'(W_2 \cdot Y_1 + B_1) \cdot W_2 \cdot \sigma'(W_1 \cdot X + B_1) \cdot X_i}}_{\text{(the component of) the gradient of loss}} \cdot \Delta w_i$$

Unfolded SimpleRNN



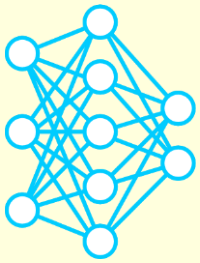
- similar to the **very deep** feedforward network:
  $Y^t = \sigma(W_X \cdot X^t + W_Y \cdot Y^{t-1} + B)$
- **weights are shared** across the (time) layers

The deeper the network is, the more likely that the vanishing gradients issue will occur



Vanishing gradients: a prototype example #3

A.Joulin and T.Mikolov . arXiv:1503.01007v4 (2015)



io/class2/bin

```python
#!/usr/bin/env python

import re, random, string, numpy as np, tqdm
import keras
from keras.models import Input, Model
from keras.layers import Dense, Flatten, Dropout, LSTM, SimpleRNN

# Get data
n_seq,seq_len,n_train,pred_len,max_len_pattern,n_epochs = 200,1000,150,100,20,700
data, x_train, y_train, x_test, y_test = [], [], [], [], []
np.random.seed(7)
for i in tqdm.tqdm(range(n_seq)):
    seq = []
    while len(seq) < seq_len:
        n = np.random.randint(1,high=max_len_pattern)
        seq += ['0']*n + ['1']*n
    data.append(''.join(seq[:seq_len]))
    start = False; prev_char = ''
    for j in range(pred_len, seq_len):
        seq_in, c_out  = data[i][(j-pred_len):j], data[i][ j]
        if start and (c_out=='1' or (c_out=='0' and prev_char=='1')):
            if i < n_train: x_train.append([int(c) for c in seq_in]); \
                            y_train.append( int(c_out))
            else:           x_test.append( [int(c) for c in seq_in]); \
                            y_test.append(  int(c_out))
        if prev_char == '0' and c_out == '1': start = True
        prev_char = c_out

# Define a model
X = Input((pred_len,1,))
if 1: Y = SimpleRNN(1, return_sequences=True)(X)
else: Y =      LSTM(1, return_sequences=True)(X)
Y = Dropout(0.2)(Y)
Y = Flatten()(Y)
Z = Dense(1, activation='sigmoid')(Y)
model = Model(inputs = X, outputs = Z)
model.compile(loss='mse', optimizer="adam", metrics=['acc'])

#Run the model on the data
expand_dims = lambda x, y: np.expand_dims(np.array(x), axis=y)
bsize, checkpoint_file = 1000, "vgrad_rnn.h5"
checkpointer = keras.callbacks.ModelCheckpoint(filepath=checkpoint_file)
if 0: model.load_weights(checkpoint_file)
if 1: model.fit(expand_dims(x_train,2), np.array(y_train), batch_size=bsize,
                shuffle=True, epochs=n_epochs, callbacks=[checkpointer])
scores = model.evaluate(expand_dims(x_test,2), np.array(y_test), verbose=0)
print("\n%s: %.2f%%" % (model.metrics_names[1],np.array(scores[1])*100))
                                                                  47,76
```

Input: a **sequence of** consecutive **palindromes**

$X = \{0^n 1^n\}$ for **random** $n = 1, \ldots, N$

$N = 20$

Example: 0001110011000001111 1…

n=3 n=2 n=5

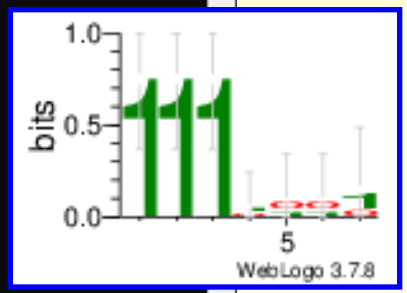
Task: predict next character in the **deterministic** part of the sequence

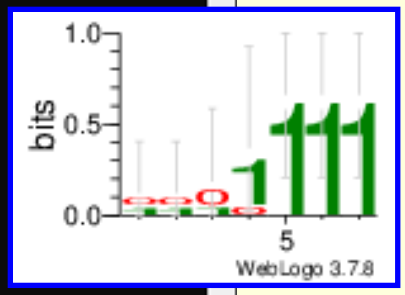
# training epochs	predict. acc
SimpleRNN	
100	91.81%
300	91.89%
500	**92.90%**
700	**92.94%**
LSTM	
100	97.92%
300	98.84%
500	**98.99%**
700	**99.05%**

How to run the prototype examples on Biowulf?

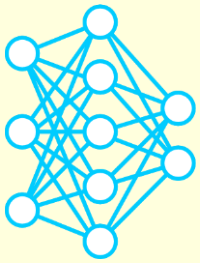
denisovga@biowulf:/data/denisovga/1_DL_Course/0_Intro

```
[user@biowulf] sinteractive

[user@cn0858] module load DLBio/class2
...
[+] Loading DLByExample class2  ...

[user@cn0858]$ ls $DLBIO_BIN
mdet_conv1d.py      mdisc_conv1d.py      vgrad_rnn.py
mdet_simplernn.py   mdisc_simplernn.py

Using TensorFlow backend.
...
Epoch 1/500
32/32 [==============================] - 1s 2ms/step - loss: 0.7808
...
Epoch 500/500
32/32 [==============================] - 0s 2ms/step - loss: 0.1024
...
[user@cn0858]$ mdet_conv1d.py
...
[user@cn0858]$ mdisc_conv1d.py
...
motif= 111NNNN
[user@cn0858]$ weblogo -f 01_conv1d.fa    -F png -o 01_conv1d.png  \
                       -a '01' -C green 1 '1' -C red 0 '0'
...
[user@cn0858]$ mdisc_simplernn.py
...
motif= NNNN111
[user@cn0858]$ weblogo -f 01_simplernn.fa -F png -o 01_simplernn.png \
                       -a '01' -C green 1 '1' -C red 0 '0'
...
[user@cn0858]$ vgrad_rnn.py
...
...
Epoch 700/700
74/74 [==============================] - 4s 53ms/step - loss: 0.0543 - acc: 0.9318

acc: 92.94%
                                                          8,39          Top
```





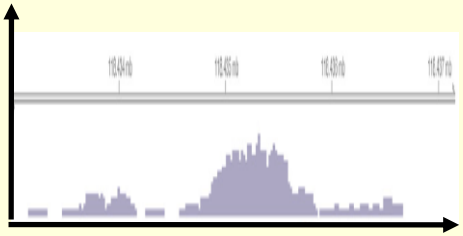
Biological example #2: Predicting the function of noncoding DNA *de novo* from sequence with DanQ and DeepSEA



DanQ: D.Quang, X.Xie, Nucl. Acids Res. (2016)
DeepSEA: J.Zhou, O.G.Troyanovskaya, Nature Methods (2015)
Review: G.Eraslan et al., Nature Reviews Genetics (2019)

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)**

NGS → 

→ Call peaks → **Narrow peak format** → One-hot encode → **DeepSEA data**

Models:
DanQ (2016) – Keras,
DeepSEA (2015) – Torch (reimplemented in Keras),
DeepBind (2015), Basset (2016) – Torch,
FIDDLE (2016), DeepMotif (2016),
Basenji (2017) – Tensorflow, DeepCpG (2017), …

DL frameworks:
Keras,
Torch,
Tensorflow,
…

Network types:
RNN,
1D-CNN

Overview of the 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
- DeepSEA model
- DanQ model
- MaxPooling1D, Flatten, Dropout
- LSTM and BLSTM
- compile, loss
- optimizer

Run the model
- fit

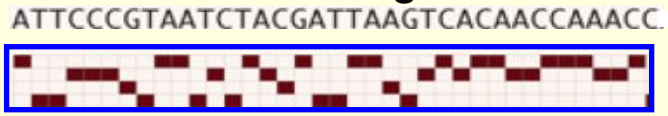
denisovga@biowulf:/data/denisovga/1_DL_Course/2_RNNs

```
if __name__ == "__main__":

    # Parse command line argumants
    opt, checkpoint_name, input_weights_file, Optimizer \
        = parse_command_line_arguments("train")
    os.environ['CUDA_VISIBLE_DEVICES'] = "0,1,2,3"

    # Get data
    X_train, y_train, X_valid, y_valid = load_data(opt)
    print("\nX_train.shape=", X_train.shape, " y_train.shape=", y_train.shape)
    print("\nX_valid.shape=", X_valid.shape, " y_valid.shape=", y_valid.shape)

    # Define a model
    strategy = tf.distribute.MirroredStrategy()
    with strategy.scope():
        model = get_model(opt)
        custom_checkpointer = MyCustomCallback(model, checkpoint_name,
                                verbose=opt.verbose, save_best_only=True)
        custom_val_loss = MyCustomValidationCallback(model, [X_valid, X_valid],
                                                             [y_valid, y_valid])
        model.compile(loss      = "binary_crossentropy",
                      optimizer = Optimizer,
                      metrics   = ['acc'])
        if opt.load_weights:
            model.load_weights(opt.checkpoint_file)
        model.summary()

    # 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, workers=opt.num_gpus)
                                                                  122,9         Bot
```

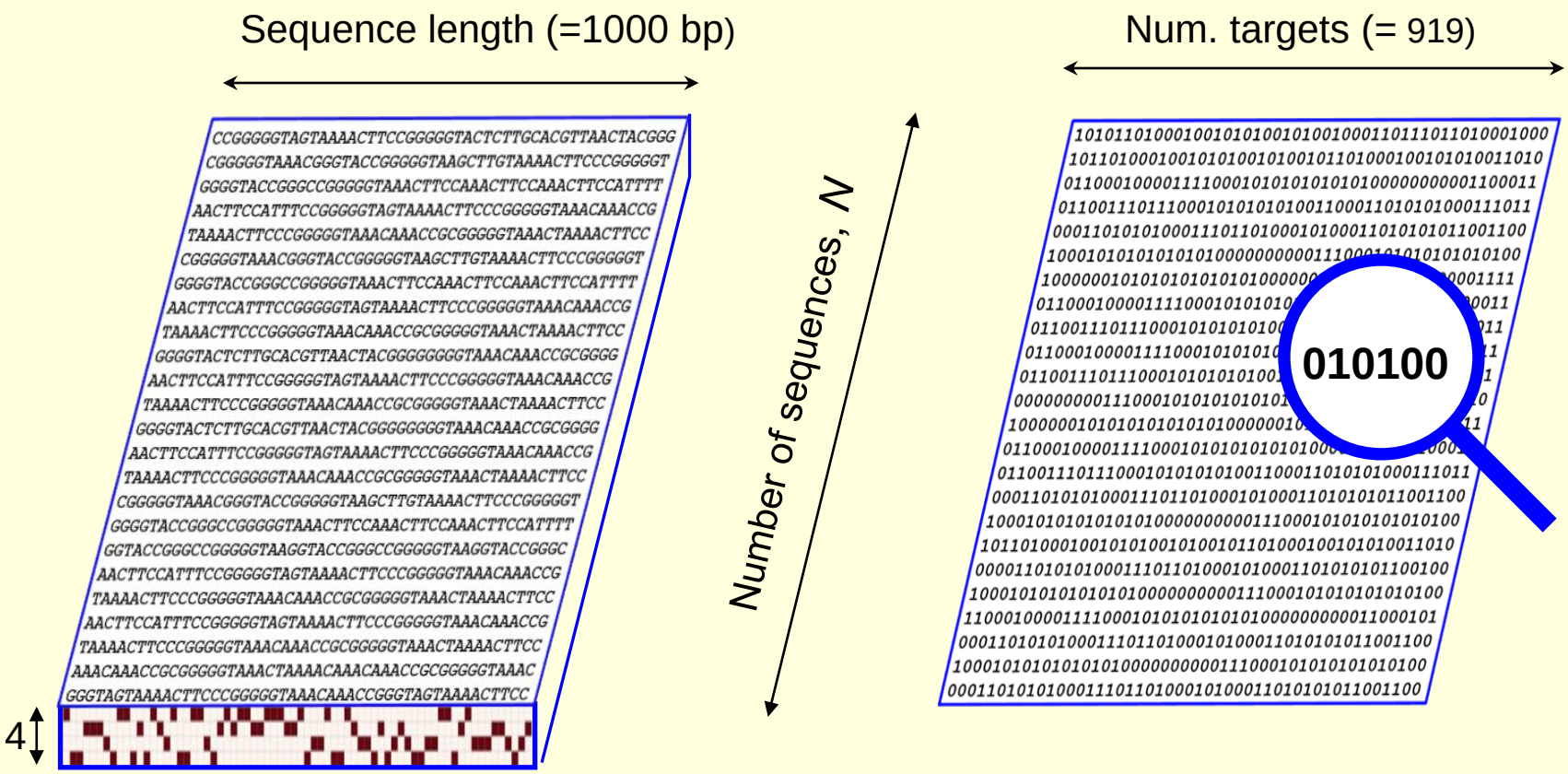
Available data

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

One-hot encoding:



Training data: N = 4.4 M => **fit: adjust <u>parameters</u>**
Validation data: N = 8 K => **fit: tune <u>hyperparameters</u>**
Testing data: N = 455 K => **predict**: **test <u>predictions</u>**

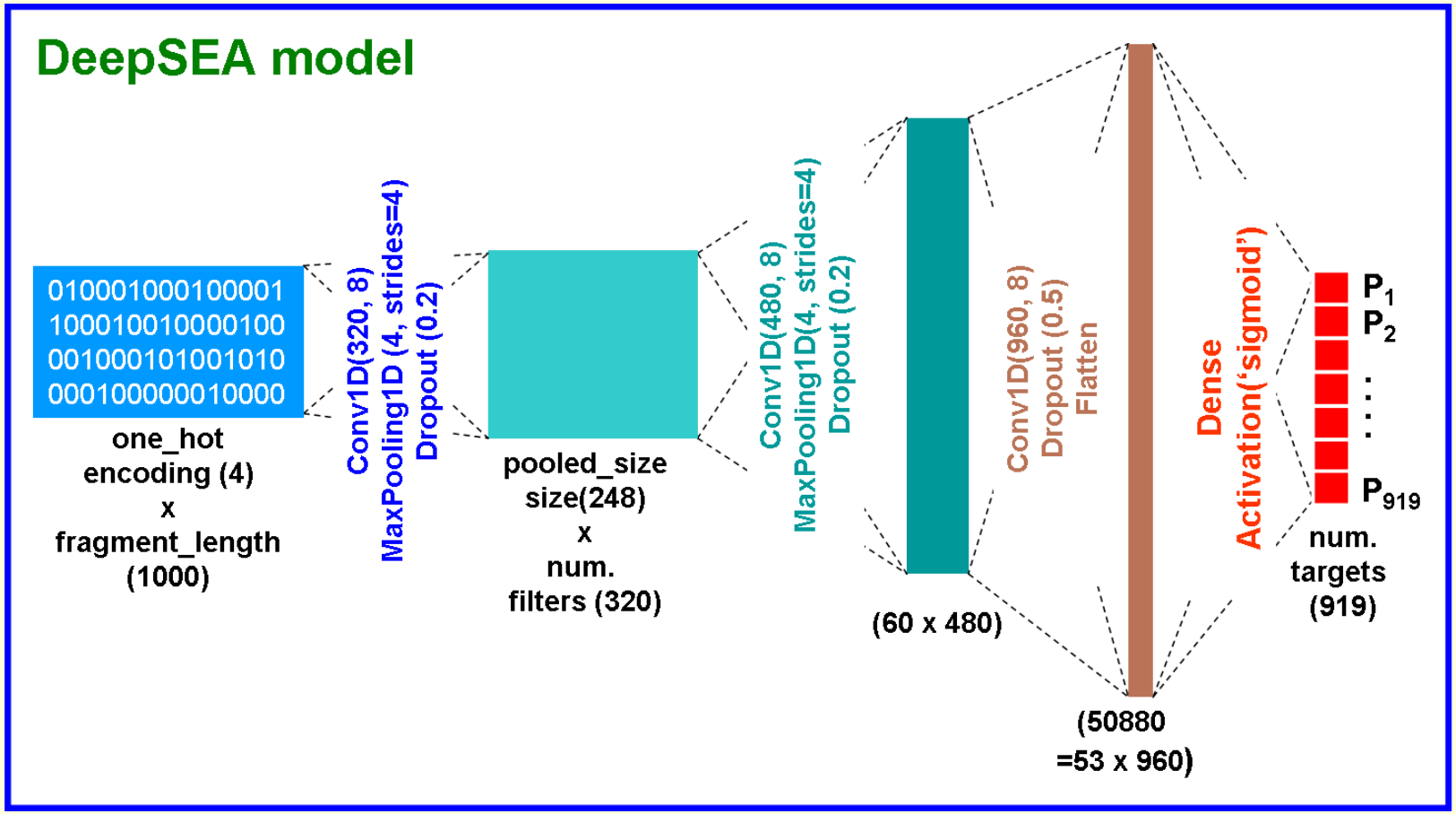


X (data) : N x 1000 x 4

y (labels): N x 919

The DeepSEA model

J.Zhou, O.G.Troyanskaya, Nature Methods (2015)



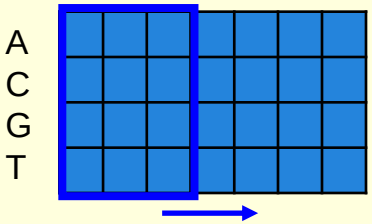
Layers: Conv1D, Dense, MaxPooling1D, Dropout, Flatten

First Conv1D layer: capture the sequence motifs.
Second Conv1D and **third Conv1D:** produce higher level representation of the data output by previous layer
Dense: linearly combine outputs and produce target probs.

Conv1D, MaxPooling1D, Dropout and Flatten layers

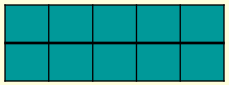
Conv1D(2, kernel_size = 3)

4 x 7



Rectangular window
of size = 4 x kernel_size

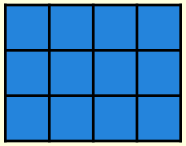
2 x 5



Output size
= num_kernels
x num_kernel_positions

Flatten()

3 x 4

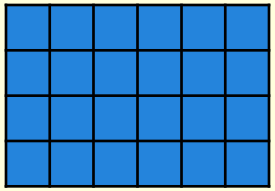


1 x 12

Purpose: resize the input tensor in order to make the output acceptable by Dense layer

MaxPooling1D(pool_size = 2)

4 x 6



1 x 3

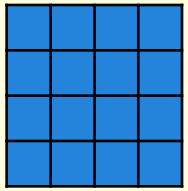
2.0	3.0	0.0	5.0	2.5	0.0
2.0	1.0	5.0	0.0	4.0	1.5
3.0	5.0	6.0	3.0	2.0	0.0
4.0	4.0	0.5	1.5	1.0	2.0

5.0	6.0	4.0

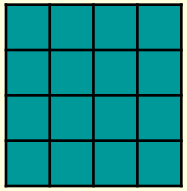
Purpose: prevent the model from overfitting

Dropout (0.5)

4 x 4



4 x 4



3.0	2.5	1.5	2.0
4.0	3.0	5.0	7.0
6.0	5.0	7.0	2.0
2.5	5.0	2.0	2.0

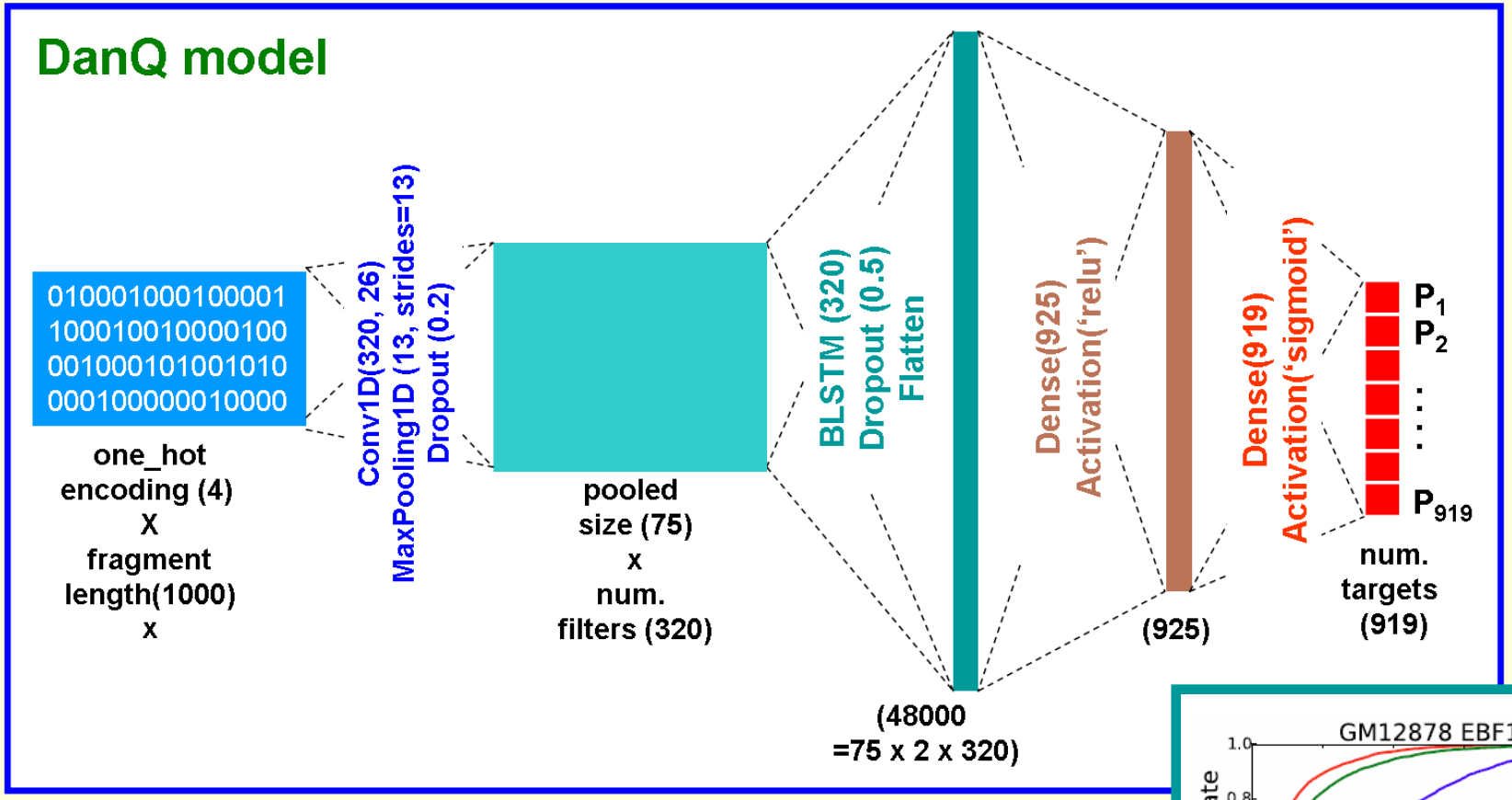
3.0	0.0	1.5	2.0
0.0	3.0	0.0	7.0
0.0	5.0	7.0	0.0
0.0	5.0	0.0	0.0

Purpose: prevent the model from overfitting

The DanQ model

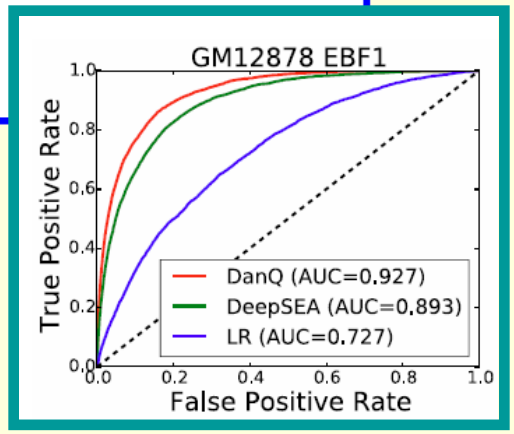
LSTM, BLSTM, MaxPooling1D, Dropout, Flatten

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



Conv1D: # params = 33,600;
purpose: discover motif sequences

BLSTM: # params = 1,640,960;
purpose: capture the long-range dependencies between motifs / different parts o fragments

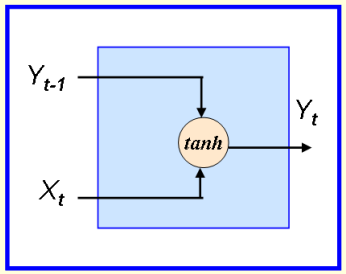


Long Short-Term Memory (LSTM) cell

Orig. study: S Hochreiter, J Schmidhuber, Neural computation 9, p.1735 (1997)
Tutorial: https://adventuresinmachinelearning.com/keras-lstm-tutorial

SimpleRNN:

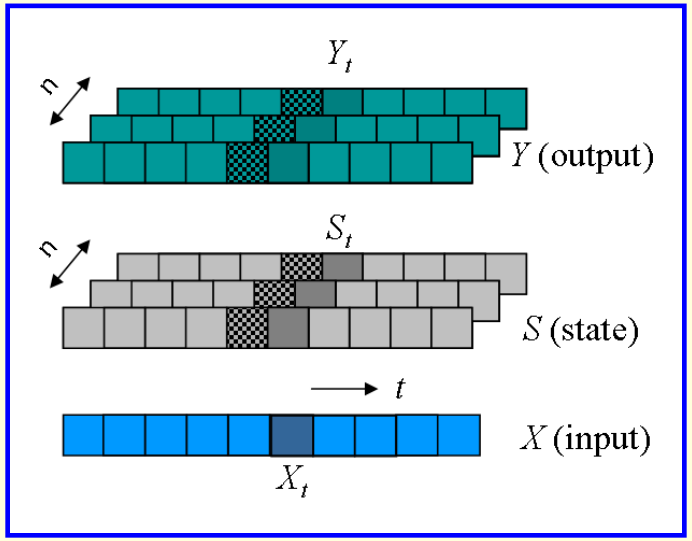
1) $X_t, Y_{t-1} \Rightarrow Y_t$

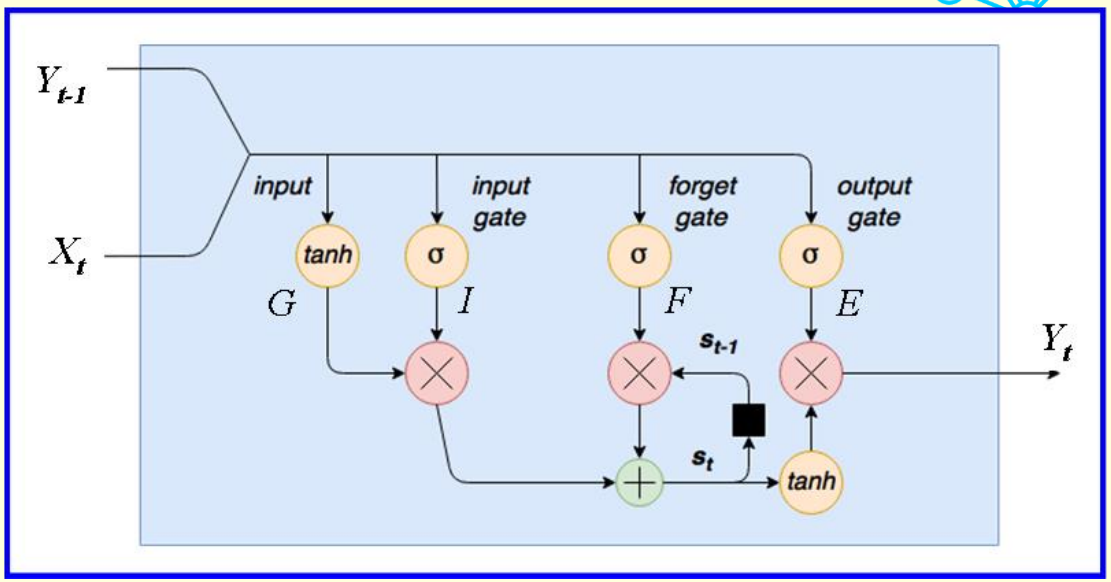


$$Y_t = tanh(b + w_{XY} \cdot X_t + w_{YY,} \cdot Y_{t-1})$$

LSTM:

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





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

$$G(X_t, Y_{t-1}) = tanh(b_G + w_{XG} \cdot X_t + w_{YG} \cdot Y_{t-1})$$
$$I(X_t, Y_{t-1}) = \sigma(b_I + w_{XI} \cdot X_t + w_{YI} \cdot Y_{t-1})$$
$$F(X_t, Y_{t-1}) = \sigma(b_F + w_{XF} \cdot X_t + w_{YF} \cdot Y_{t-1})$$
$$E(X_t, Y_{t-1}) = \sigma(b_E + w_{XE} \cdot X_t + w_{YE} \cdot Y_{t-1})$$

$\otimes$ = elementwise multiplication; σ = sigmoid activation

parameters = $4*(2n + n^2)$

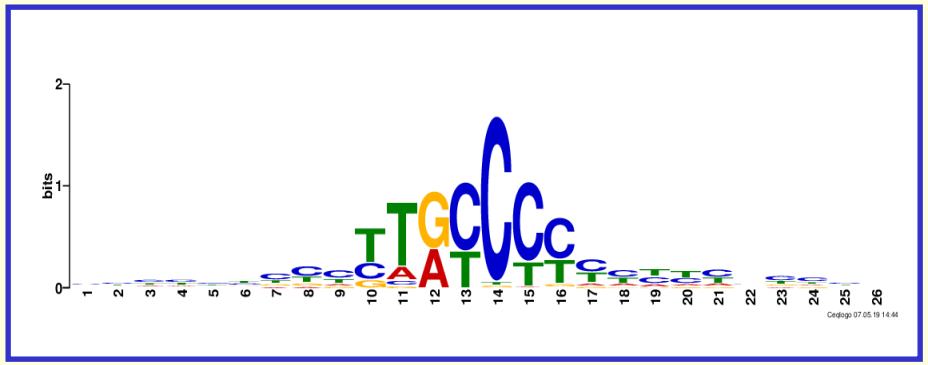
How to run the DanQ code on Biowulf?

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

Using a single GPU:

```
denisovga@biowulf:/usr/local/apps/danq/20210407/src
sinteractive  --mem=64g --gres=gpu:v100:1,lscratch:100 \
              --cpus-per-task=14

module load danq

ls $DANQ_SRC
models.py  options.py  predict.py  train.py  visualize.py

cp -r $DANQ_DATA/* .

train.py -d data  [ other options ]

predict.py -d data [ other options ]

visualize.py -t <target_id> [ other options ]
                                          15,48          Top
```

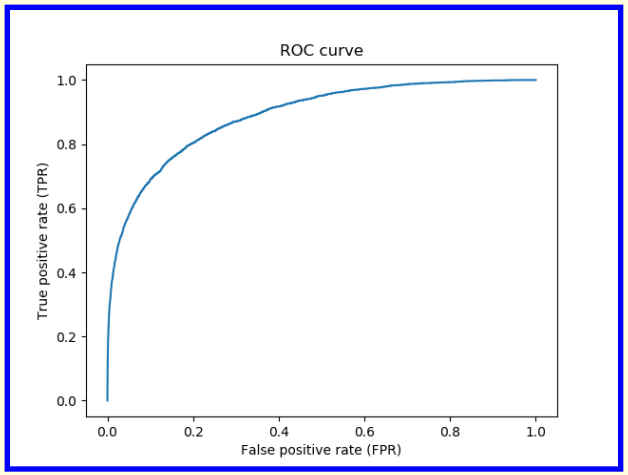
Discovered motif sequence logo:



Using 4 GPUs:

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

module load danq

ls $DANQ_SRC
models.py  options.py  predict.py  train.py  visualize.py

cp -r $DANQ_DATA/*  .

train.py -d data -g 4 [ other options ]
                                          11,42          Top
```

ROC curve:



Summary

1) Further intro using simple examples
   - SimpleRNN vs Conv1D layers/transformations
   - the notion of the RNN network memory and interacting channels
   - motif detection and discovery
   - the SGD optimizer
   - backpropagation, long-range sequence dependencies and vanishing gradients

2) Predicting the function of a non-coding DNA
   - the DeepSEA and DanQ models
   - MaxPooling1D, Dropout, Flatten and (Bidirectional) LSTM layers
   - how to run the DanQ code on Biowulf

BACKUP SLIDES

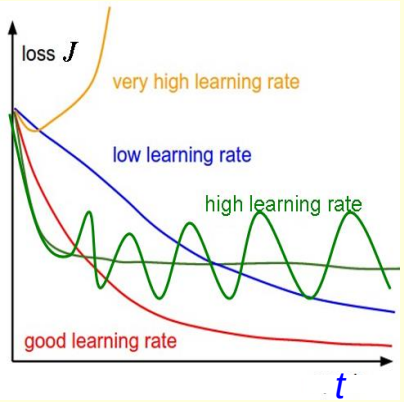
Stochastic Gradient Descent optimizer (cont.)

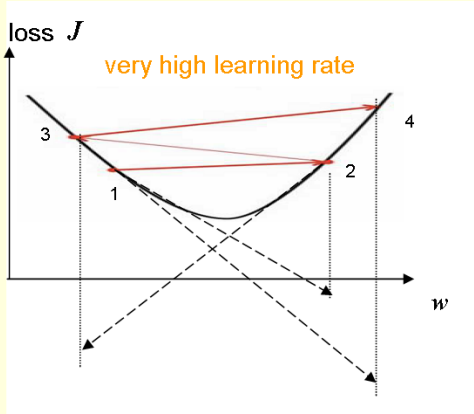
learning rate, momentum, Nesterov accelerated gradient

$$\Delta w_t = -\gamma \cdot \nabla_w J(w_t) \quad ; \quad \Delta w_t = w_{t+1} - w_t$$

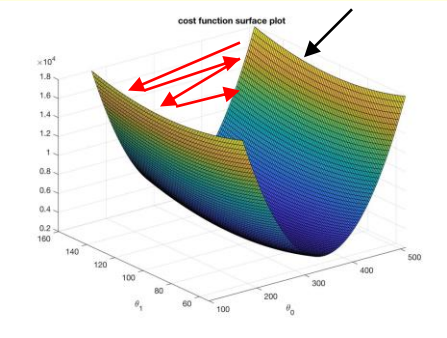
- the basic gradient descent formula re-written

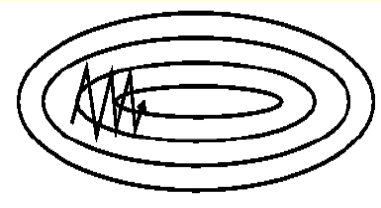
1) **learning_rate γ**





loss J view from above





- small $\gamma \rightarrow$ slow convergence along the valley
- larger $\gamma \rightarrow$ oscillations in the perpendicular dir.

2) **momentum > 0**

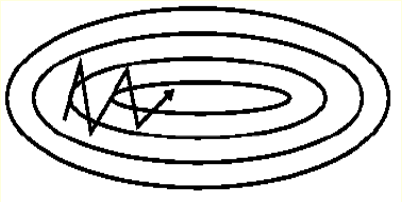
$$\Delta w_t = \mu \cdot \Delta w_{t-1} - \gamma \cdot \nabla_w J(w_t)$$

- gradient descent formula with momentum μ (usually, = 0.9)

3) **nesterov = True**

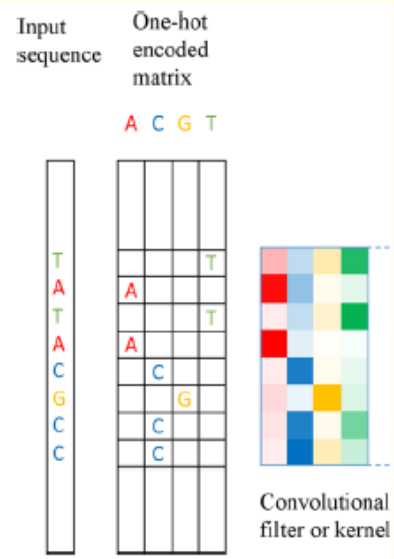
$$\Delta w_t = \mu \cdot \Delta w_{t-1} - \gamma \cdot \nabla_w J(w_t - \mu \cdot \Delta w_{t-1})$$

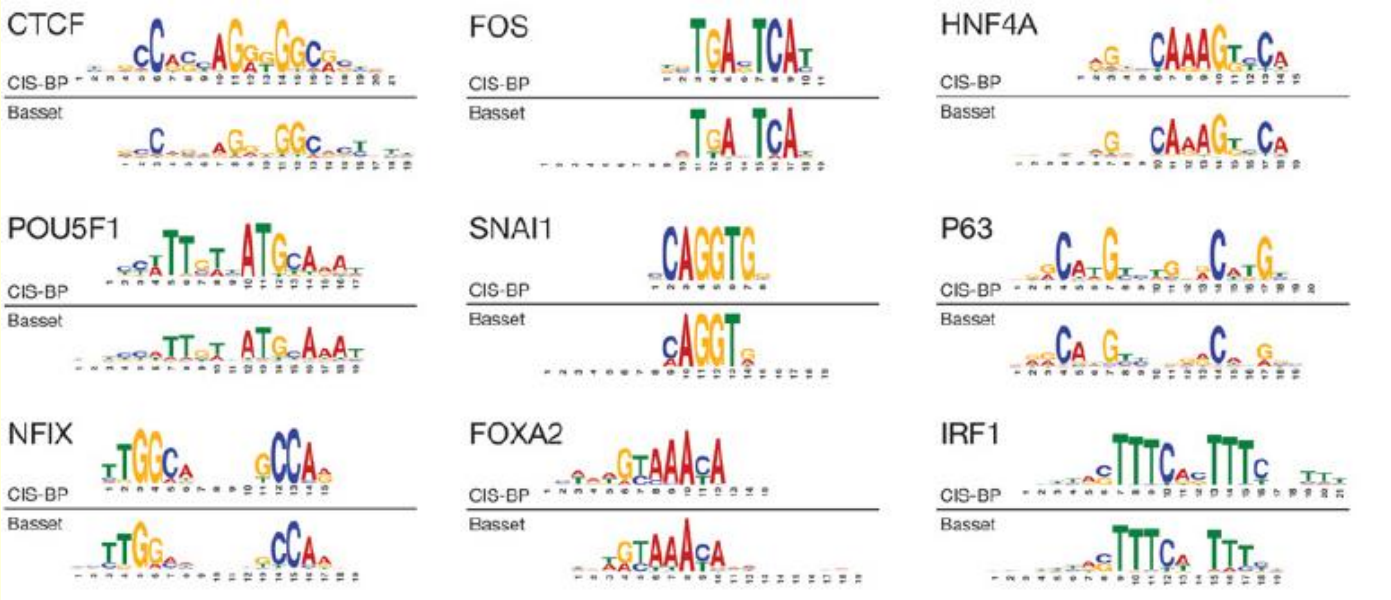
- gradient descent formula with momentum μ and Nesterov accelerated gradient



Predicting motifs

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





Overall, 45% of filters could be annotated