block b2

block b2

Convolutional Networks, Residual Networks & Recurrent Networks

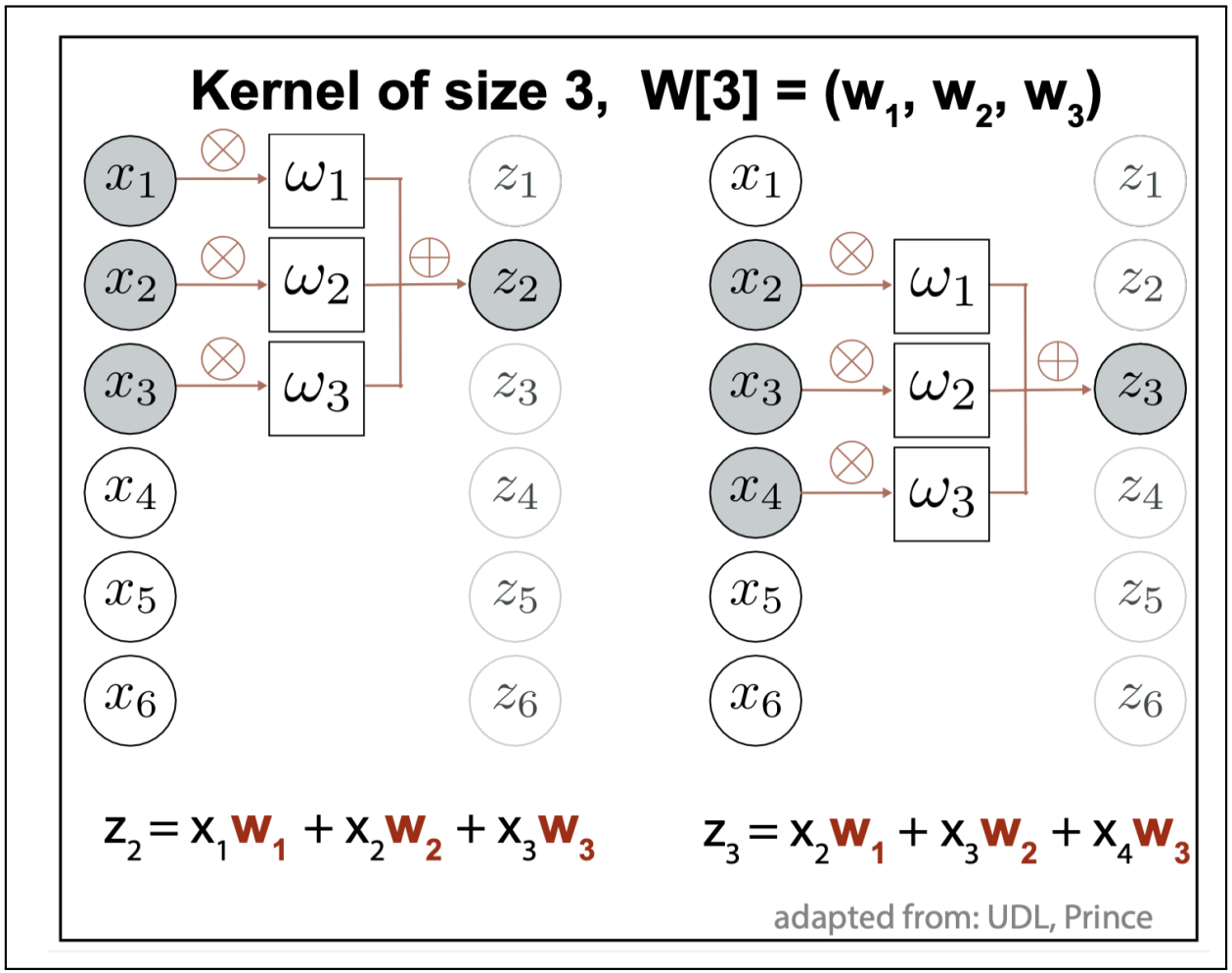
block b2

Convolutional Networks, Residual Networks & Recurrent Networks

DNA/RNA binding motifs

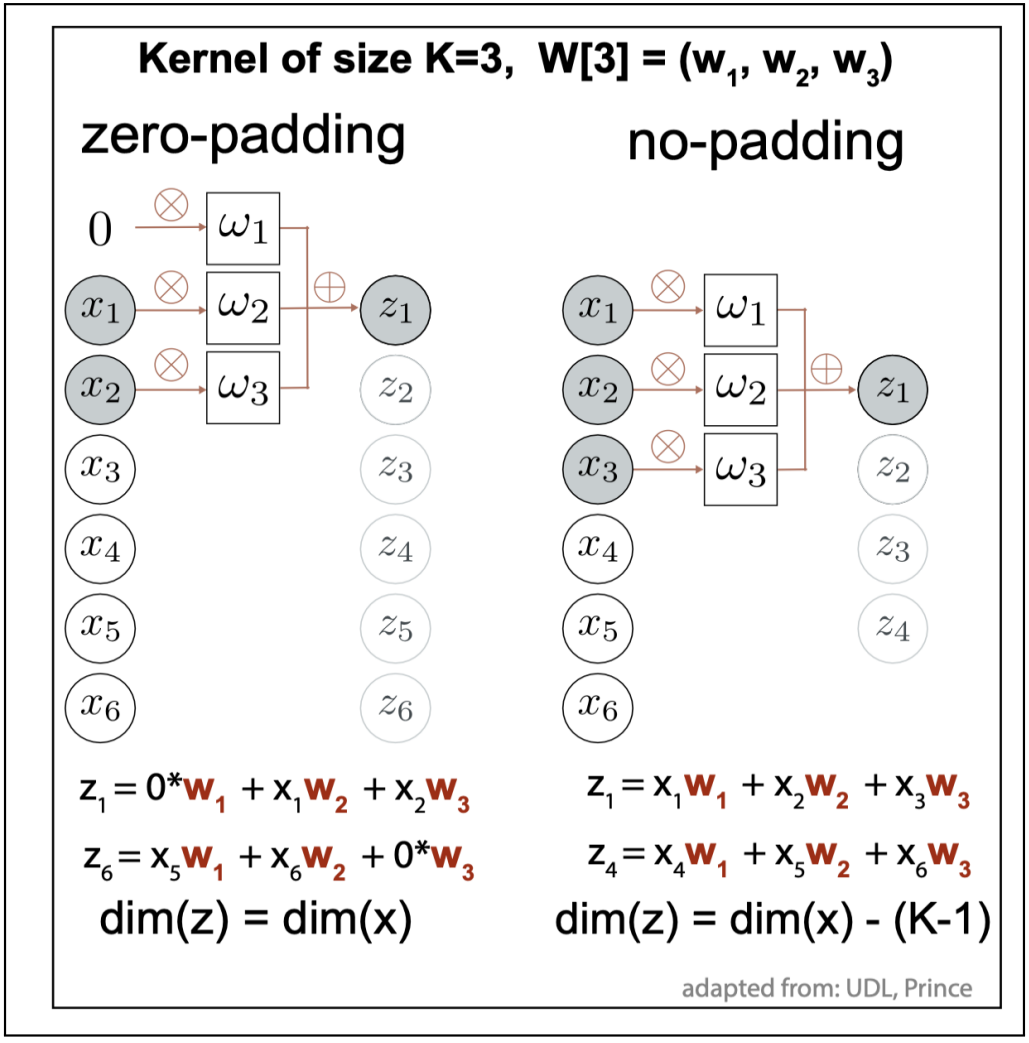
A convolution in 1D

The Kernel (or Filters)



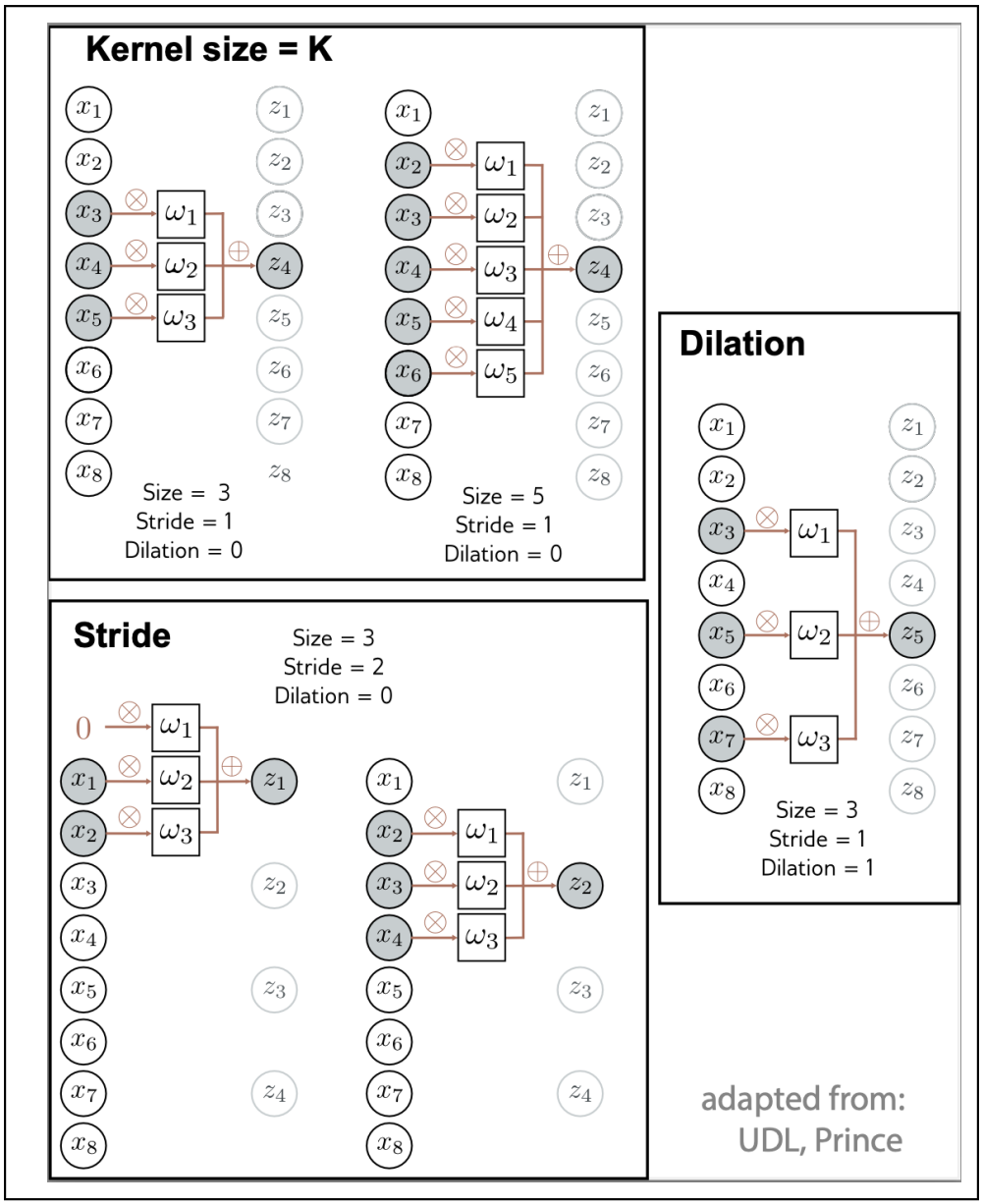
A convolution in 1D

Padding



A convolution in 1D

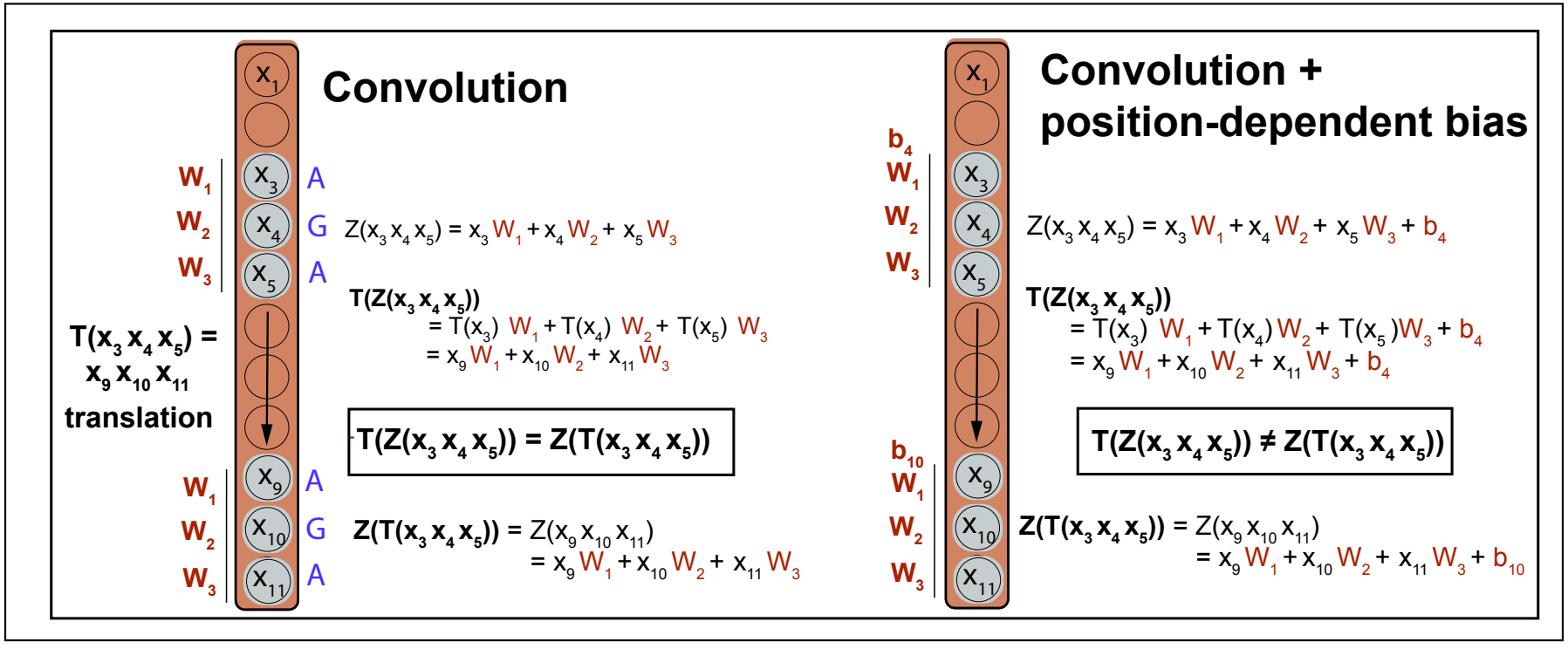
Stride and Dilation



Equivariance by translation

$$T_m(x_i) = x_{i+m}$$

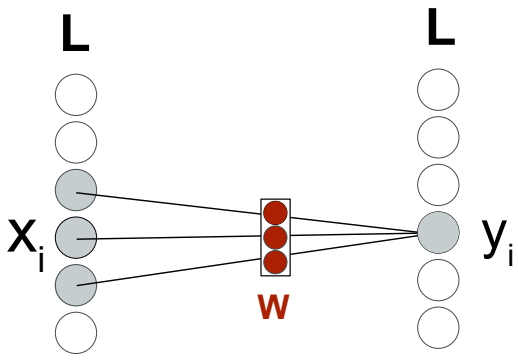
$$z(T(x)) = T(z(x))$$



1D convolutional layer

Add activation function

1D Convolutional Layer



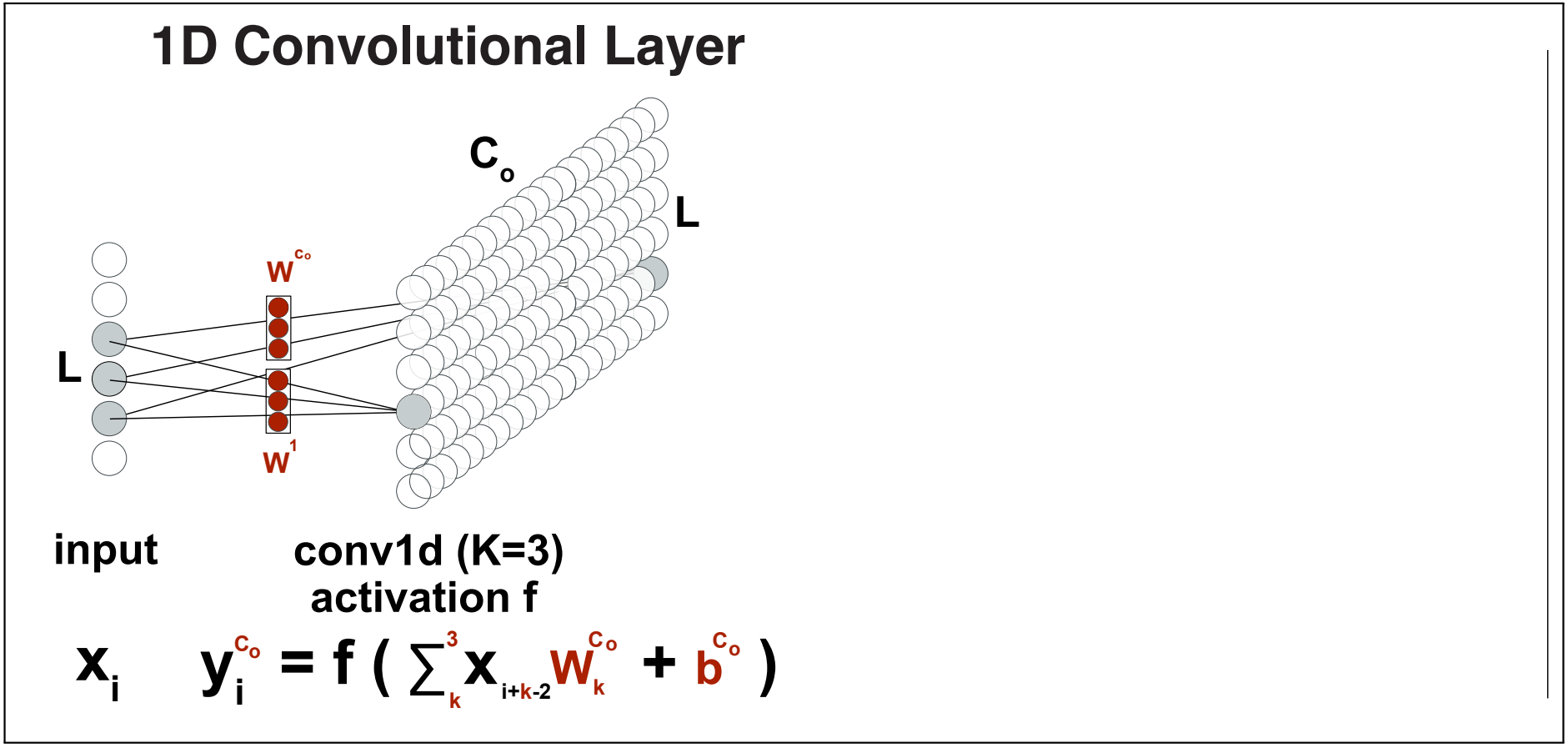
input

conv1d (K=3)
activation f

$$y_i = f\left(\sum_k^3 x_{i+k-2}\, W_k + b\right)$$

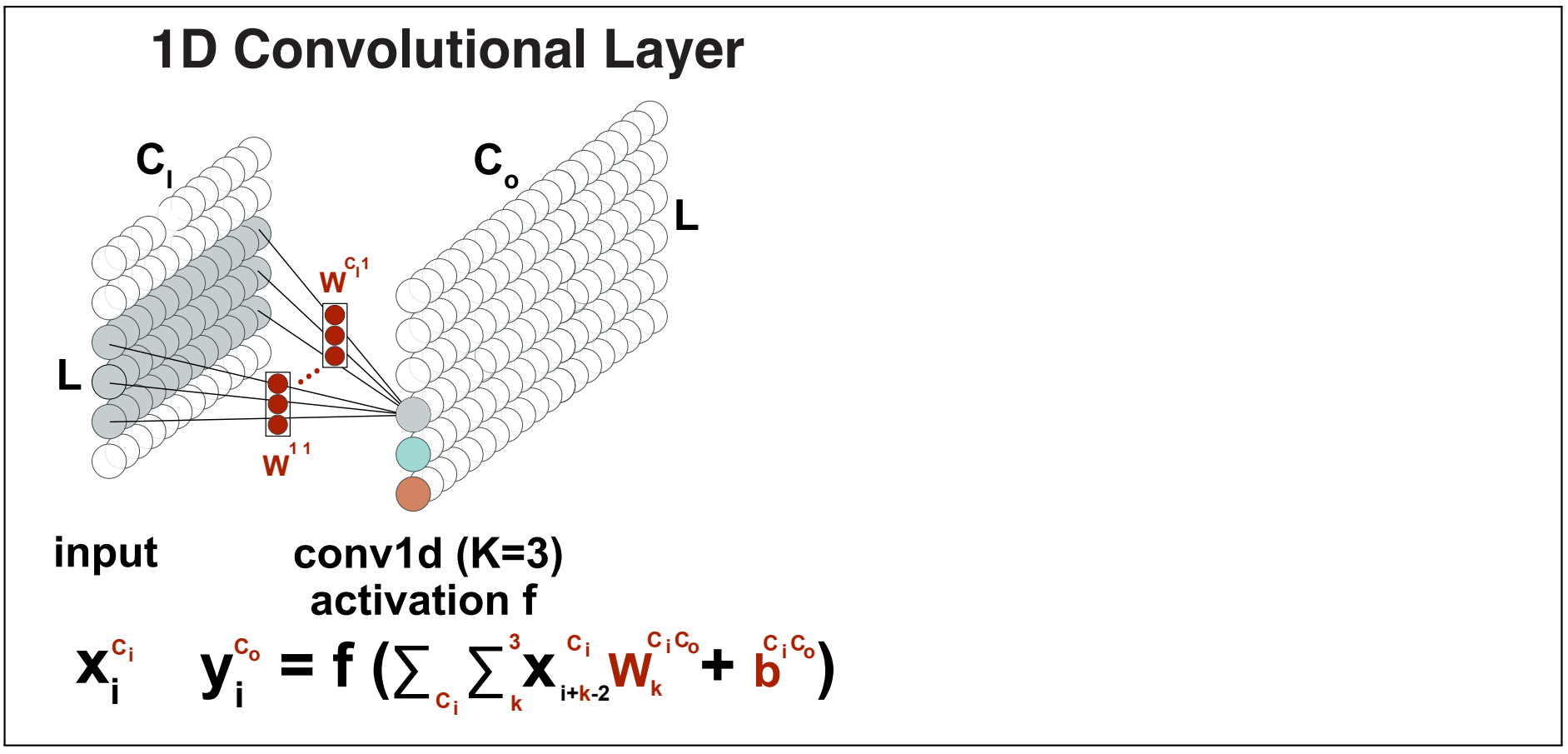
Channels

Add output channels



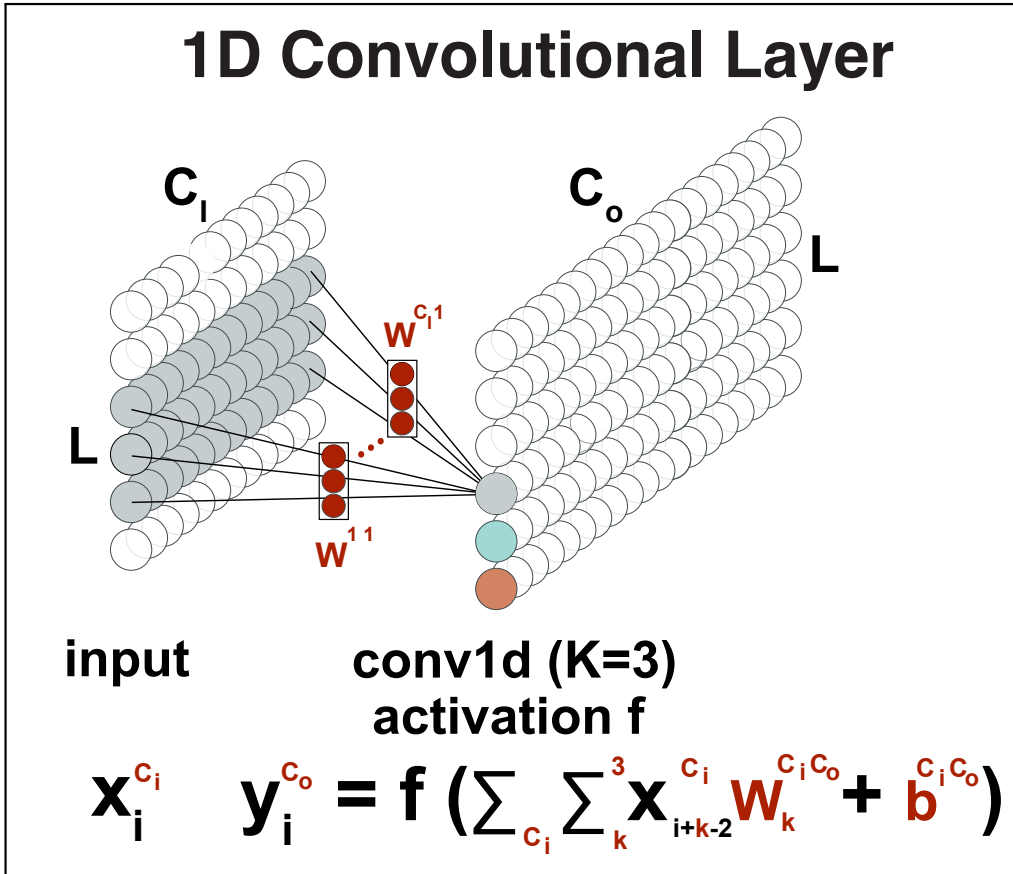
Channels

Add input channels



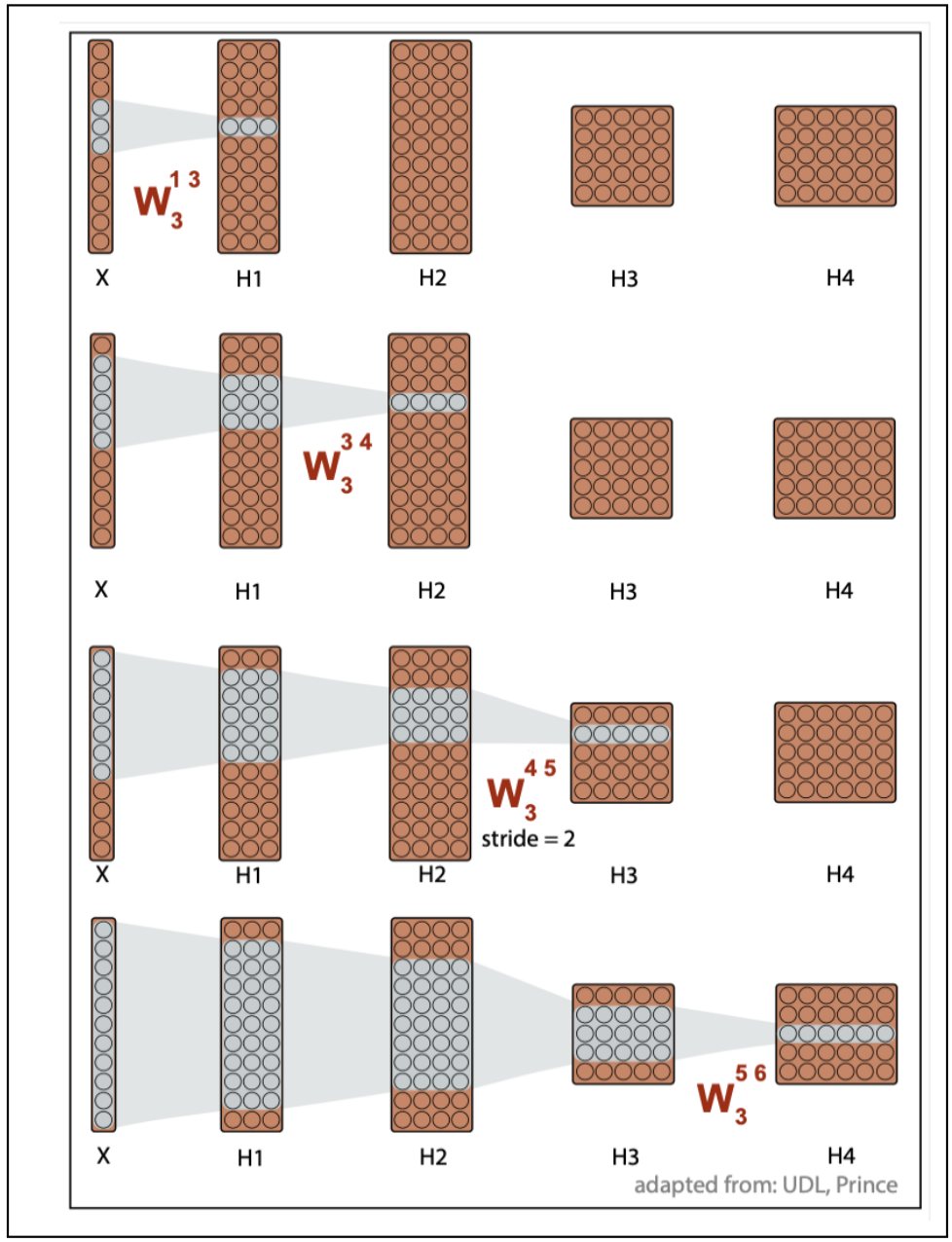
Channels

With input and output channels

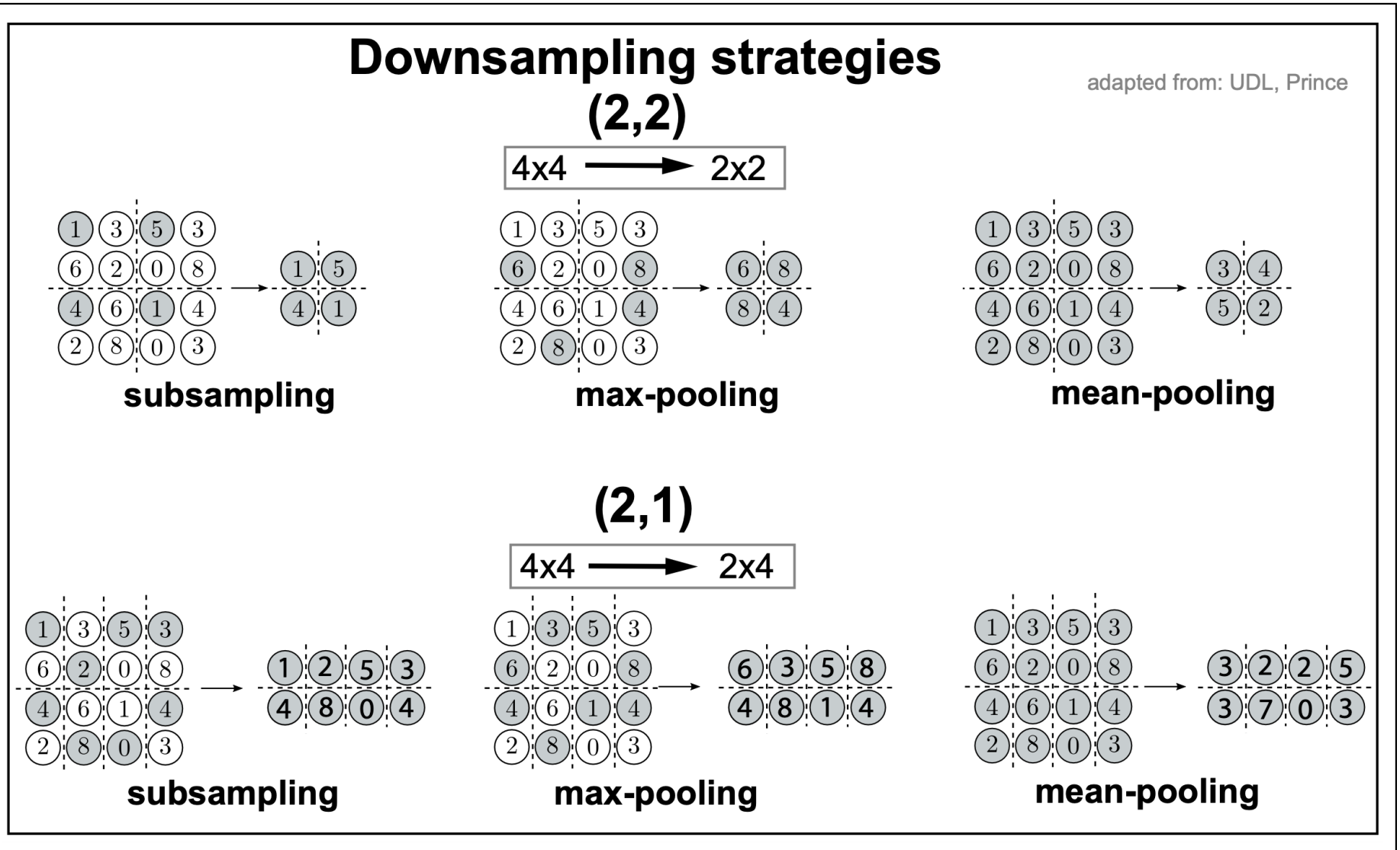



```
# 1d convolution
 conv = nn.Conv1d(
        in_channels=C_I,
        out_channels=C_O,
        kernel_size=K
      )

#inputs x [N, C_I,  L]
#outputs y [N, C_O,  L]
y = F.relu(self.conv(x))
```

The receptive field

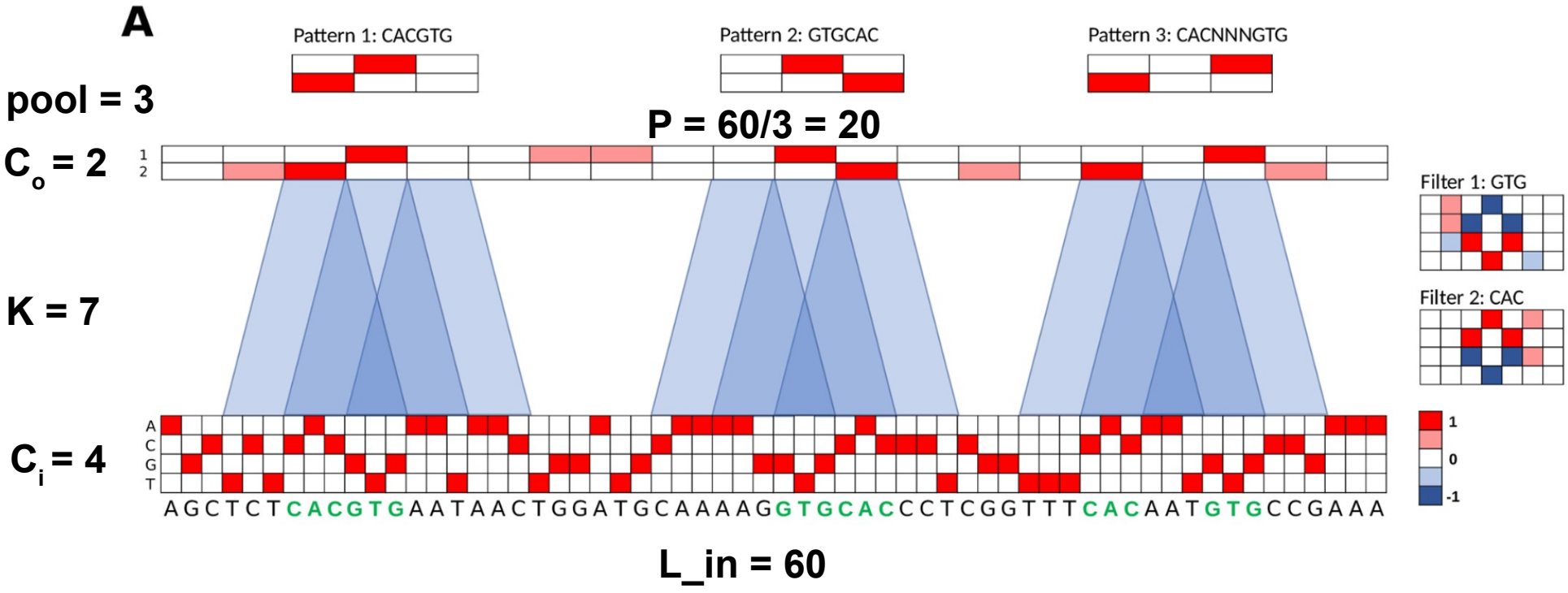


Pooling


Downsampling strategies
adapted from: UDL, Prince
(2,2)
4x4 → 2x2
subsampling
max-pooling
mean-pooling
(2,1)
4x4 → 2x4
subsampling
max-pooling
mean-pooling

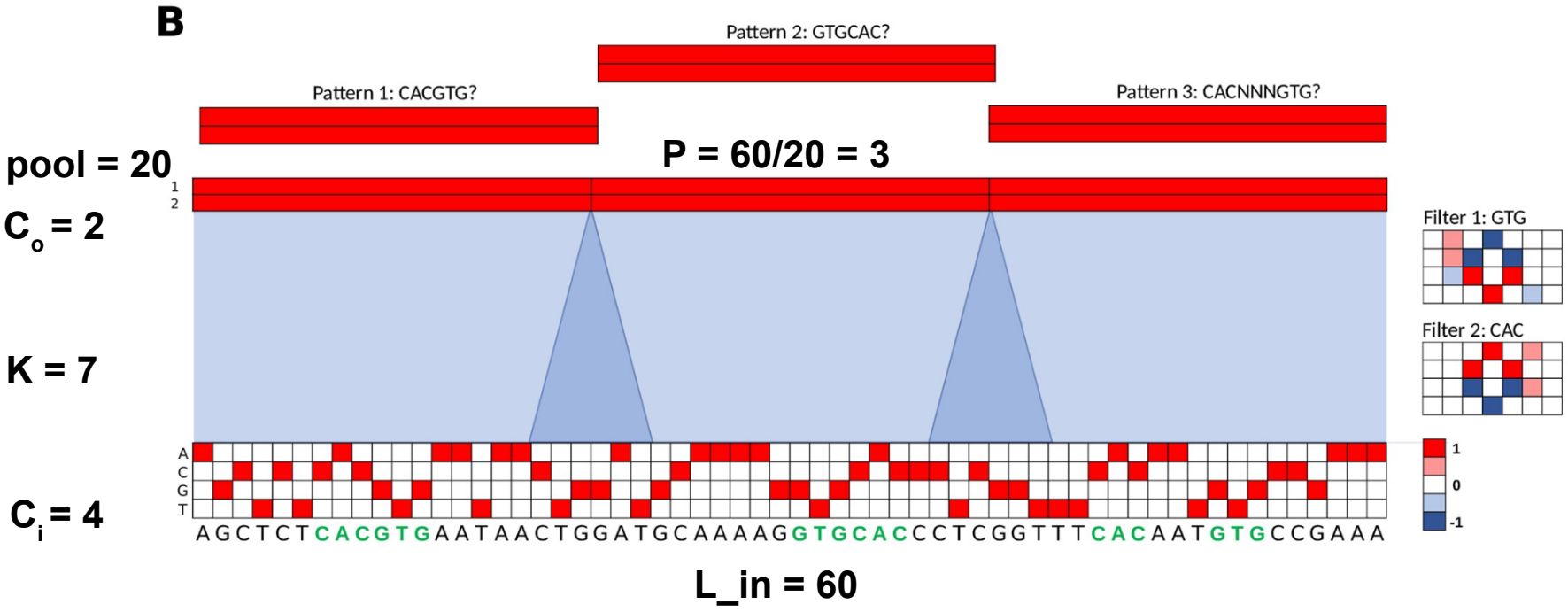
Pooling

Pooling size = 3



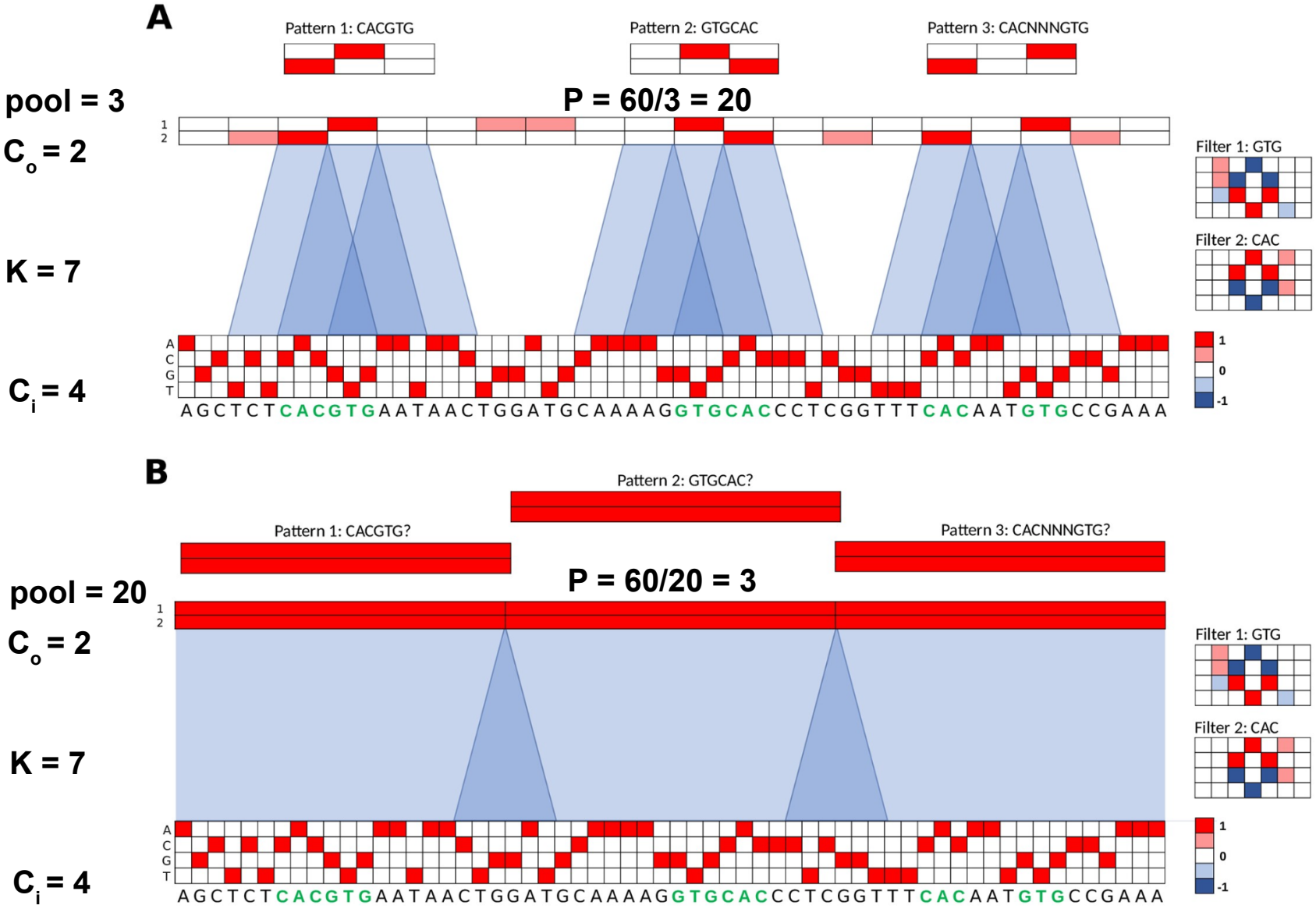
Pooling

Pooling size = 20

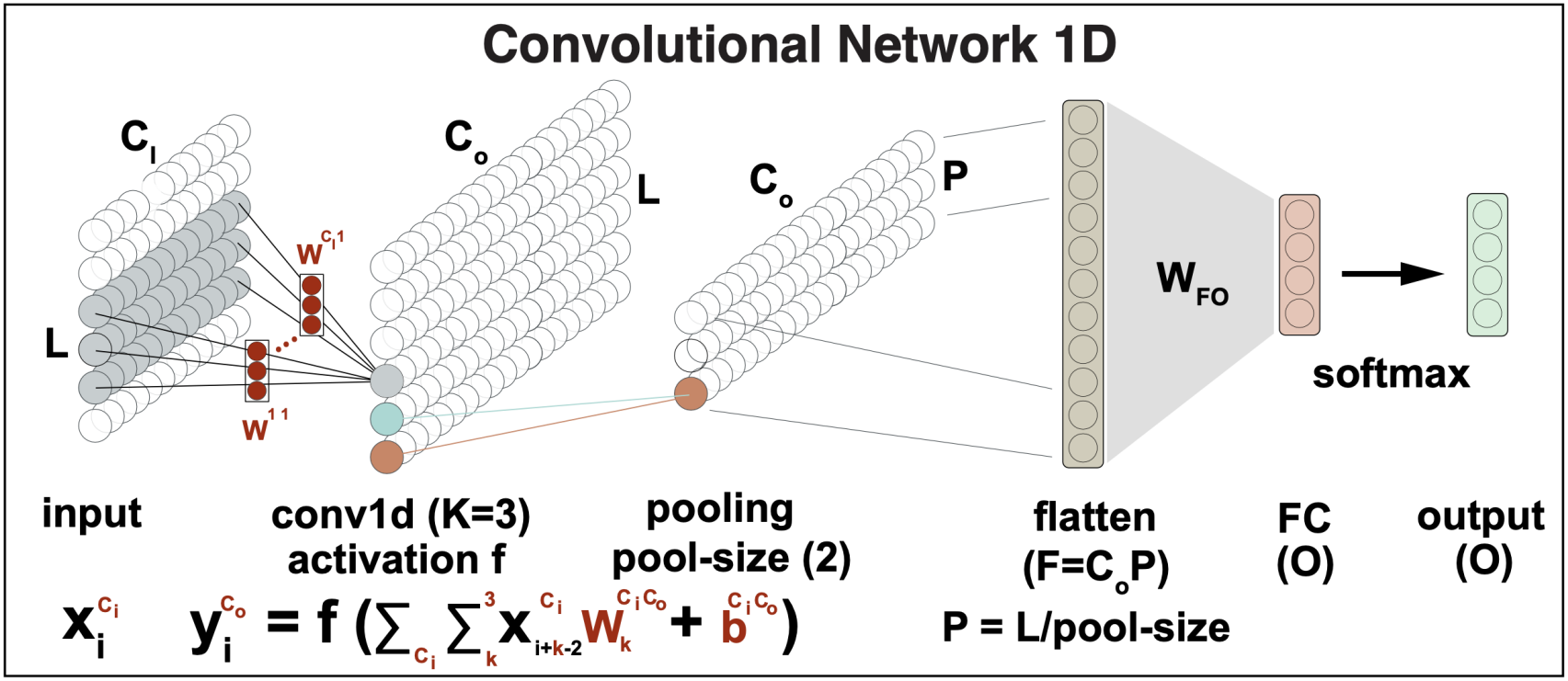


Pooling

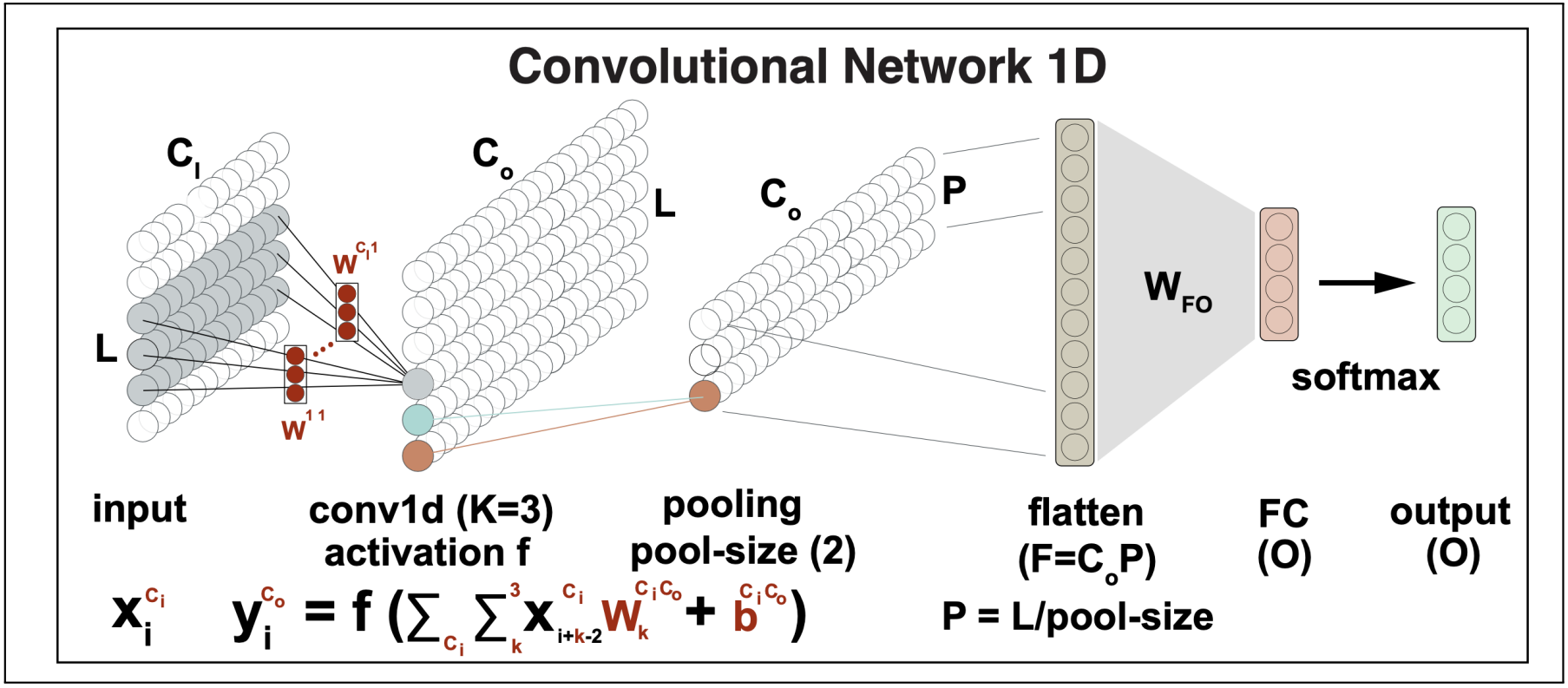
Control of receptive field



CNN Network

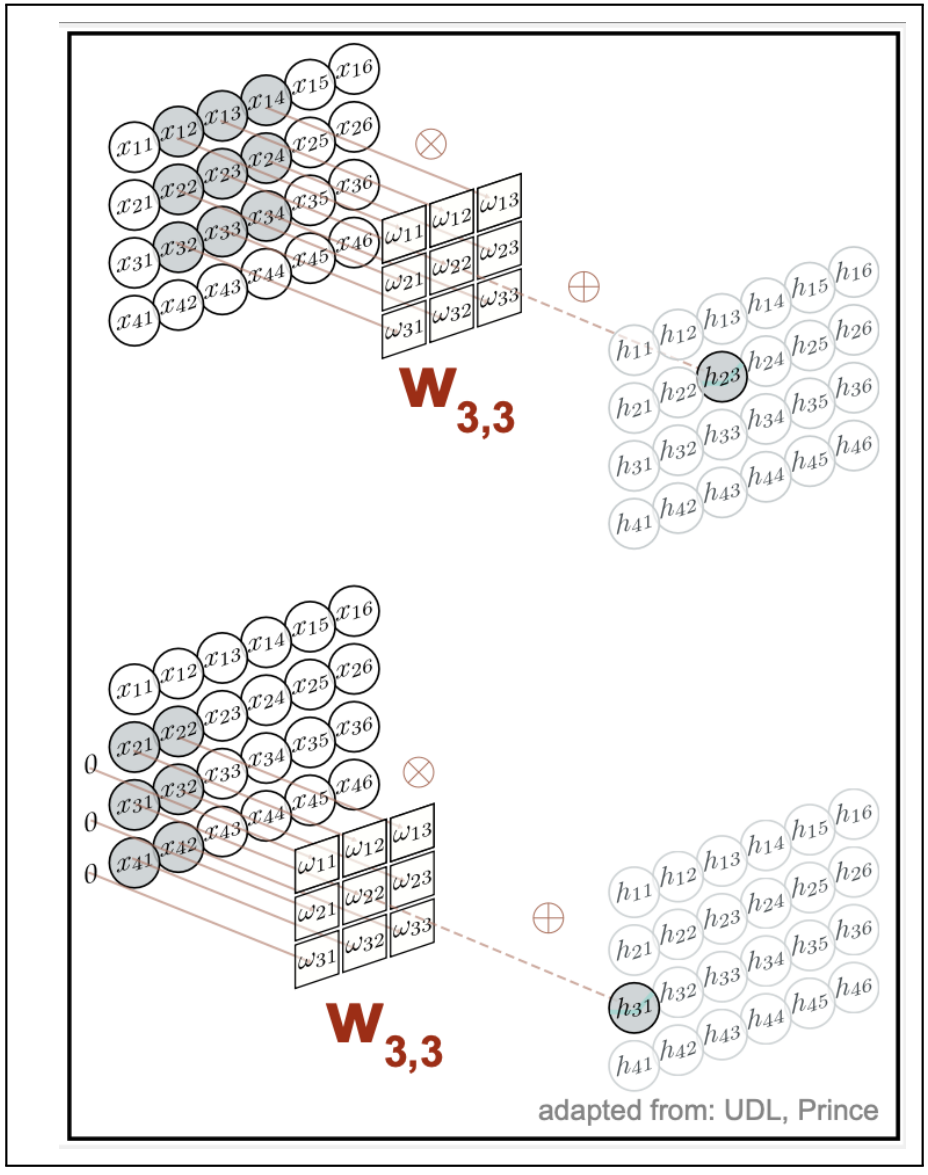

Convolutional Network 1D
C_i
C_o
L
C_o
P
L
W^{C_i 1}
W^{1 1}
W_FO
softmax
input
conv1d (K=3)
activation f
pooling
pool-size (2)
flatten
(F=C_o P)
FC
(O)
output
(O)
x_i^{C_i}
y_i^{C_o} = f (∑_{C_i} ∑_k^3 x_{i+k-2}^{C_i} W_k^{C_i C_o} + b^{C_i C_o})
P = L/pool-size

CNN Network

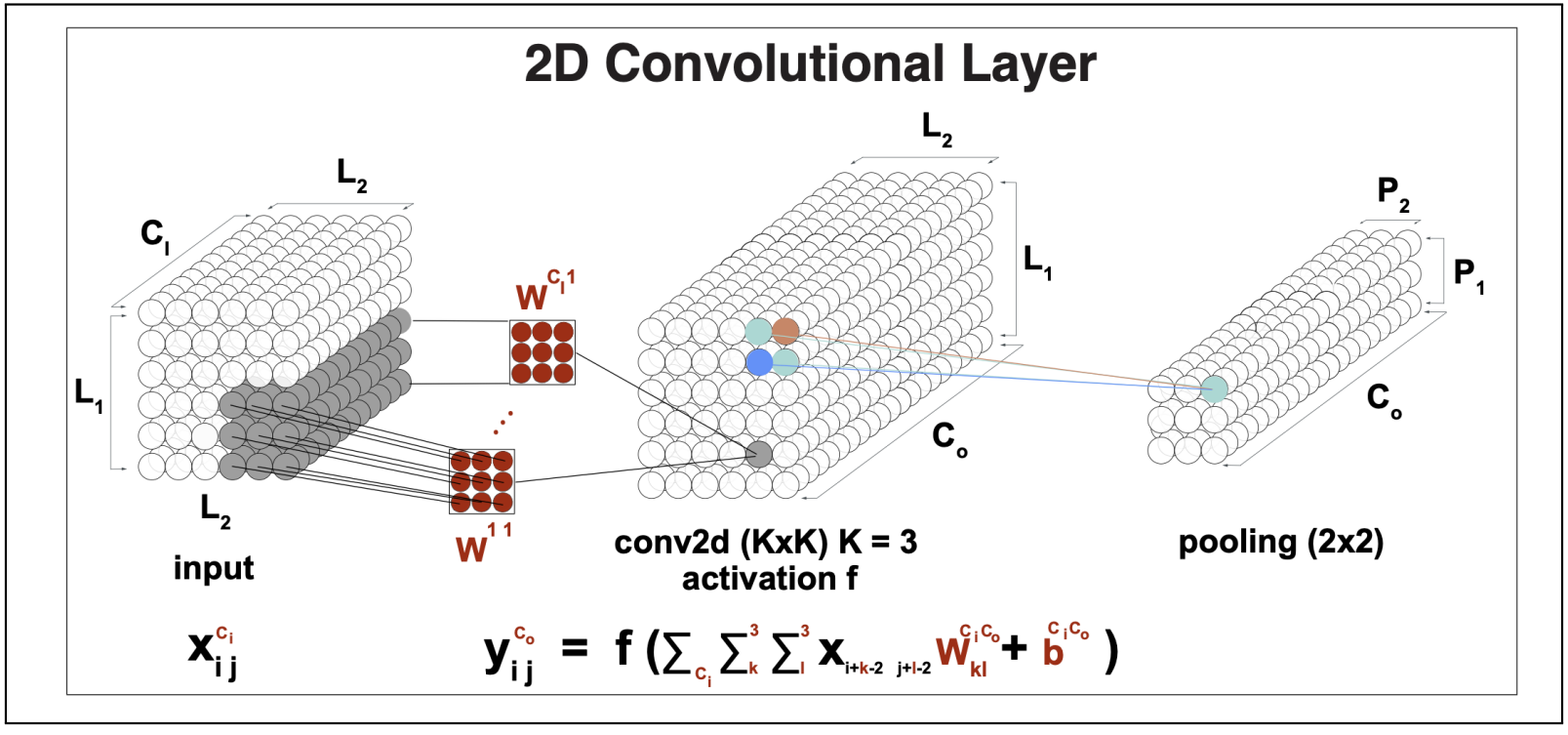


Parameters (weights) do not depend on L anymore!

2D convolutions

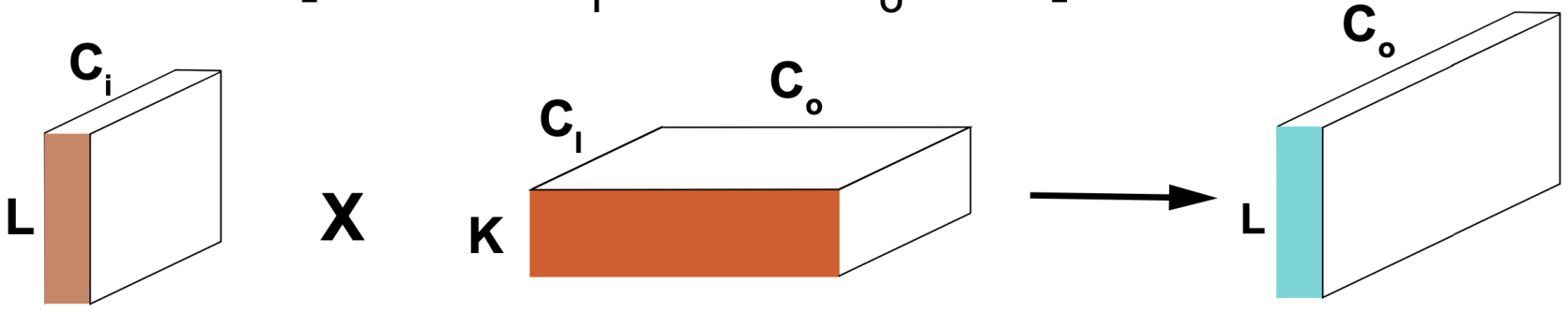


2D Convolutions



1D Convolution Summary

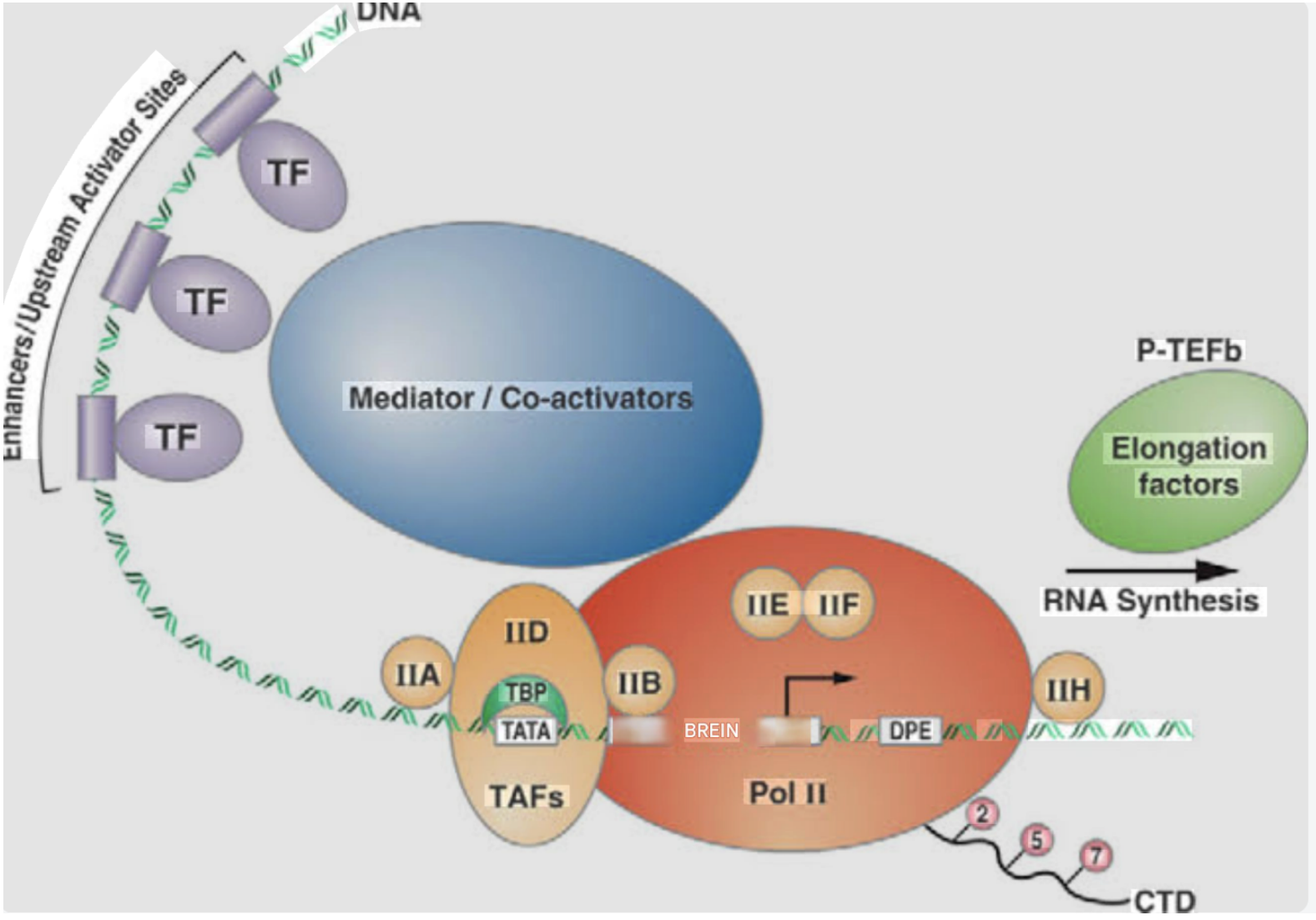
1D CNN [K = 5, C_I =192, C_o = 48]



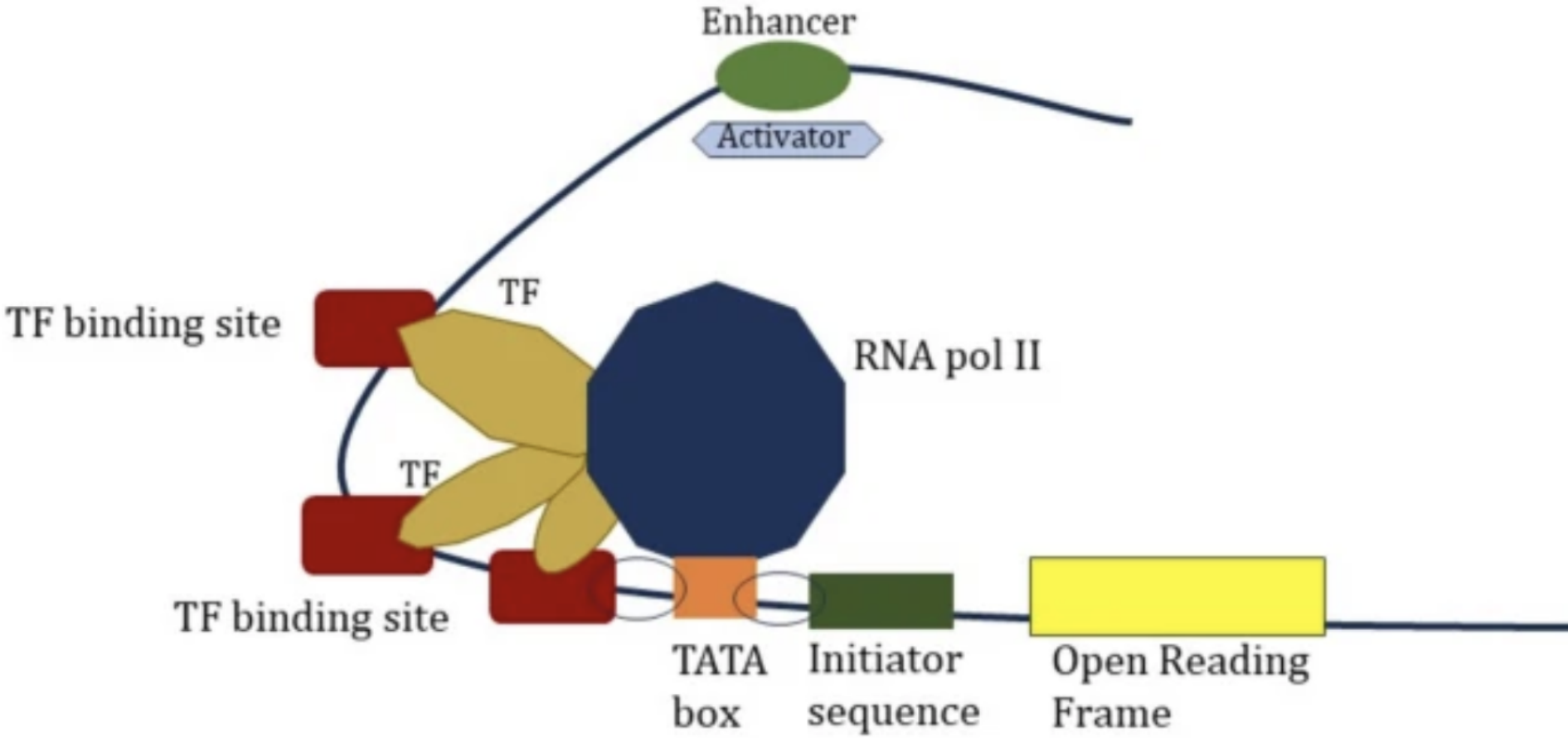
$$y_i^{C_o} = f\left(\sum_{C_i} \sum_{k=1}^{5} x_{i+k-2}^{C_i} W_k^{C_i C_o} + b^{C_i C_o}\right)$$

number of operation = (L x C_0) x (K x C_1) = L x 48x5x192 = L x 46,080

Transcription initiation / TF / TF binding sites)



Transcription initiation / TF / TF binding sites)

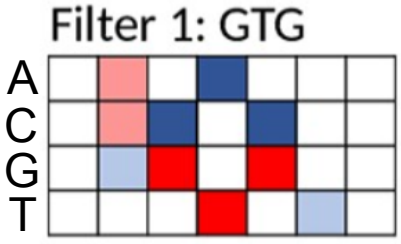


PWM-like representations (interpretability)

Kernnels as Position Weight Matrices representations of sequence motifs

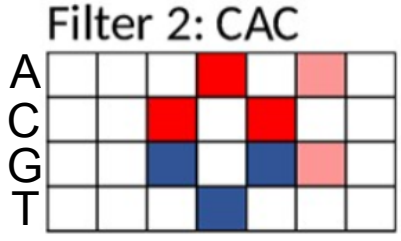
C_o = 2 = number of PWMs

C_i = 4

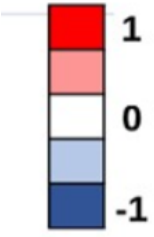


$PWM1[K, C_I] = W[K, C_I, C_O=1]$

C_i = 4

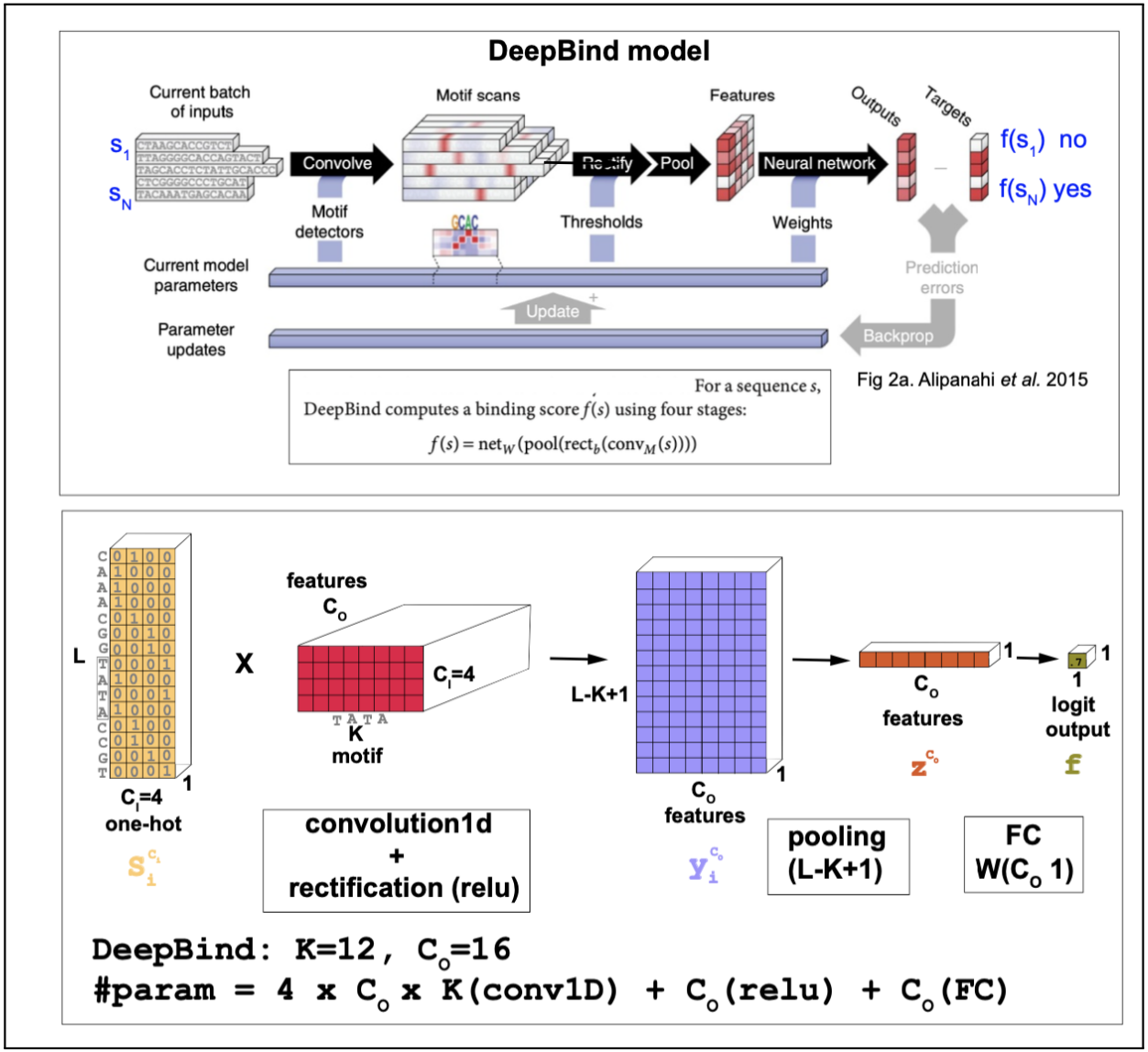


$PWM2[K, C_I] = W[K, C_I, C_O=2]$



K = 7

DeepBind


DeepBind model
Current batch of inputs
Motif scans
Features
Outputs
Targets
Convolve
Rectify
Pool
Neural network
Motif detectors
Thresholds
Weights
Current model parameters
Prediction errors
Update
Parameter updates
Backprop
f(s1) no
f(sN) yes
For a sequence s,
DeepBind computes a binding score f(s) using four stages:
f(s) = net_W(pool(rect_b(conv_M(s))))
Fig 2a. Alipanahi et al. 2015
L
C_I=4
one-hot
X
features
C_o
K
motif
convolution1d
+
rectification (relu)
L-K+1
C_o
features
pooling
(L-K+1)
C_o
features
logit
output
FC
W(C_o 1)
DeepBind: K=12, C_o=16
#param = 4 x C_o x K(conv1D) + C_o(relu) + C_o(FC)

DeepBind

One DeepBind model per TF

1. **Inputs.** One-hot encoded sequences $x[L, 4]$.
2. **Outputs.** One single scalar output $f(x)$, transformed to a probability using the linear logistic function

$$x[L, 4] \rightarrow f(x) \rightarrow \frac{1}{1 + e^{-f(x)}}$$

3. **Architecture**
   - 3.1 One 1D CNN layer, $K = 12$, $C_i = 4$, $C_o = 16$,
   - 3.2 ReLU (parameteric)

$$x[L, 4] \rightarrow y[L - K + 1, 16]$$

   - 3.3 Pooling all L-K+1 features together

$$y[L - K + 1, 16] \rightarrow z[1, 16]$$

   - 3.4 Fully connected (FC) layer $C_o \rightarrow 1$

$$z[1, 16] \rightarrow f[1]$$

DeepBind

Training

1. **labels.** Binding affinite scores $sc(x)$ to the TF
2. **Loss.**
   - 2.1 Minimum square errors for scores

$$MSE = \frac{1}{B}\sum_{b}^{B}(sc(x_b) - f(x_b))^2$$

   - 2.2 binary cross-entrpy (for logistic-transformed scores)

$$t(x) = \frac{1}{1 + e^{-sc(x)}}$$

$$\sigma(x) = \frac{1}{1 + e^{-f(x)}}$$

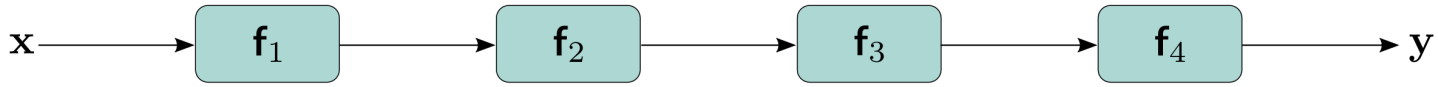
$$CE = -\frac{1}{B}\sum_{b}^{B}[t(x_b)\ \log(\sigma(x_b)) + (1 - t(x_b))\ \log(1 - \sigma(x_b))]$$

DeepBind

```
# DeepBind-like model
# DeepBind is a 1D-CNN
#
# we use PyTorch nn.Conv1d()
#       conv1d inputs  [N, C_I=4,  L_in]
#       conv1d outputs [N, C_O=16, L_out=L_in-motif_w+1] (no padding)
#       cov1dW  [4, C_O, motif_w]
#
#  MaxPool1d[kernel=L_out]
#      pool1d inputs [N, C_O=16, L_out]
#      pool1d outputs[N, C_O=16, 1]
#
#  Linear[C_O, 1]
#      linear inputs [N, C_O=16]
#      linear outputs[N, 1]
#
class DeepBind(nn.Module):
    def __init__(self, L_in, C_O=16, motif_w=12):
        super(DeepBind, self).__init__()

        # 1d conv
        self.conv = nn.Conv1d(
            in_channels=4,
            out_channels=C_O,
            kernel_size=motif_w
        )

        # max pool layer
        L_out  = L_in - motif_w + 1  # no padding
        L_pool = L_out               # pools all features together
        self.pool = nn.MaxPool1d(kernel_size=L_pool) # stride = kernel_size

        # dense layer
        L_dense = int(L_out/L_pool) # L_dense = 1
        self.fc = nn.Linear(L_dense*C_O, 1)

    def forward(self, x):
        # x shape: [batch, 4, L_in]
        x = F.relu(self.conv(x))                # [batch, C_O, L_out]
        x = self.pool(x)                        # [batch, num_filters, L=1]
        x = x.squeeze(-1)                       # [batch, num_filters]

        x = self.fc(x)                          # [batch,1]
        x = x.squeeze(-1)                       # [batch]
        # return as logits

        return x
```

Residual Networks

Sequential network



$$y = f_4(f_3(f_2(f_1(x))))$$

$$\frac{\delta y}{\delta f_1} = \frac{\delta y}{\delta f_4}\frac{\delta f_4}{\delta f_3}\frac{\delta f_3}{\delta f_2}\frac{\delta f_2}{\delta f_1}$$

adapted from: UDL, Prince

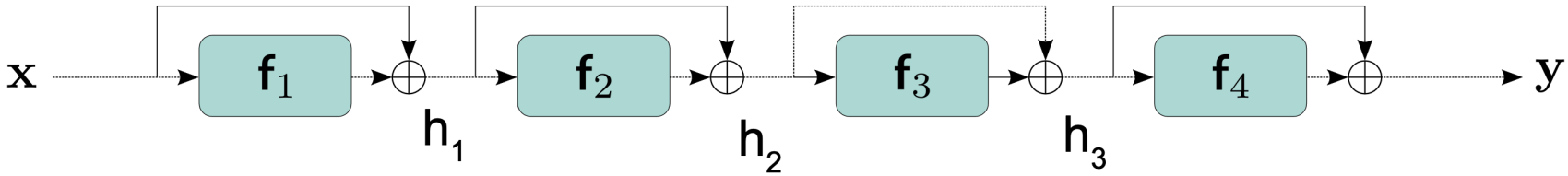
The vanishing gradient problem

Chapter 11, Undestanding deep learning, Prince

Residual Networks

Residual network

adapted from: UDL, Prince



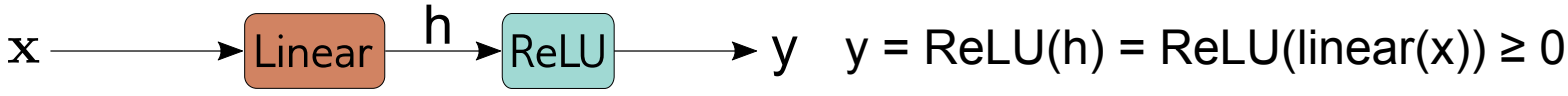
h1 = x + f1[x]
h2 = h1 + f2[h1]
h3 = h2 + f3[h2]
y = h3 + f4[h3]

y = x + f1[x]
+ f2[x + f1[x]]
+ f3[x + f1[x] + f2[x + f1[x]]]
+ f4[x + f1[x] + f2[x + f1[x]] + f3[x + f1[x] + f2[x + f1[x]]]],

$$\frac{\delta y}{\delta f1} = 1 + \frac{\delta f2}{\delta f1} + \frac{\delta f3}{\delta f1} + \frac{\delta f3}{\delta f2}\frac{\delta f2}{\delta f1} + \frac{\delta f4}{\delta f1} + \frac{\delta f4}{\delta f2}\frac{\delta f2}{\delta f1} + \frac{\delta f4}{\delta f3}\frac{\delta f3}{\delta f1} + \frac{\delta f4}{\delta f3}\frac{\delta f3}{\delta f2}\frac{\delta f2}{\delta f1}$$

Residual Networks

adapted from: UDL, Prince



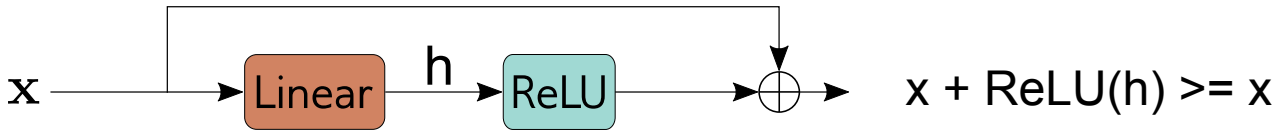
h = linear(x)

y = ReLU(h) = ReLU(linear(x)) ≥ 0

Residual Networks

Residual network order

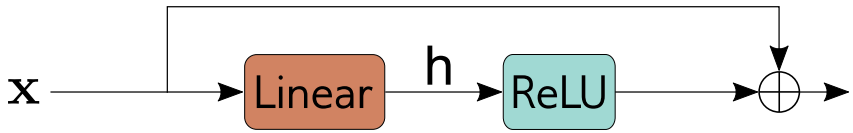
adapted from: UDL, Prince



Residual Networks

Residual network order

adapted from: UDL, Prince



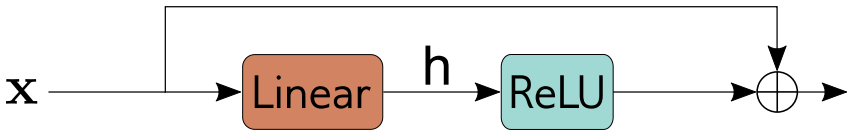
x + ReLU(h) >= x

can only increase input

Residual Networks

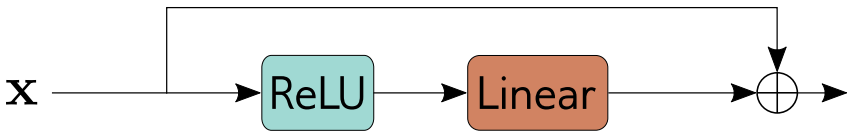
Residual network order

adapted from: UDL, Prince



x + ReLU(h) >= x

can only increase input

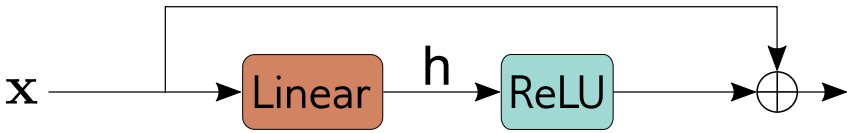


x + linear(ReLU(x)) = x if x < 0

Residual Networks

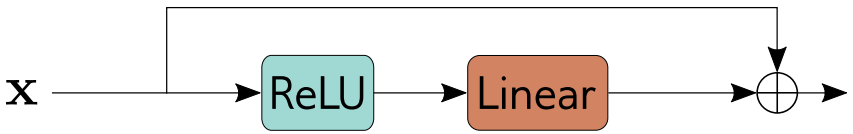
Residual network order

adapted from: UDL, Prince



x + ReLU(h) >= x

can only increase input



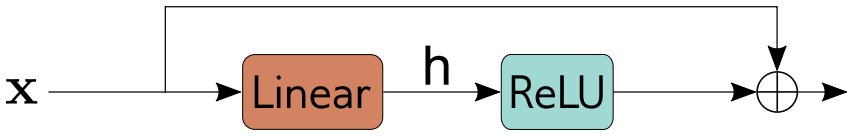
x + linear(ReLU(x)) = x if x < 0

does nothing for neg inputs

Residual Networks

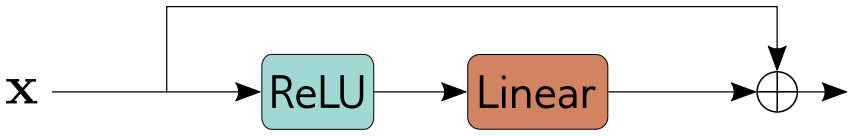
Residual network order

adapted from: UDL, Prince



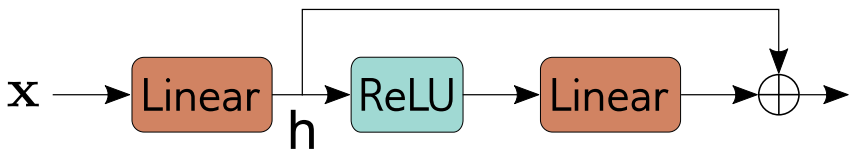
x + ReLU(h) >= x

can only increase input



x + linear(ReLU(x)) = x if x < 0

does nothing for neg inputs



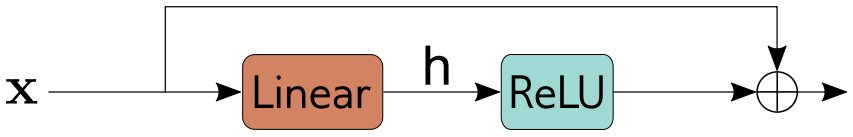
h + linear(ReLU(h))

works with all inputs

Residual Networks

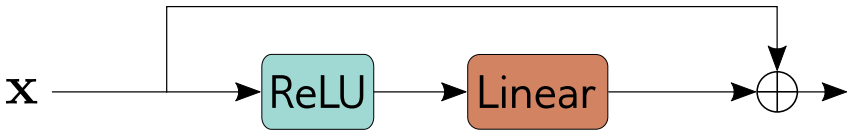
Residual network order

adapted from: UDL, Prince



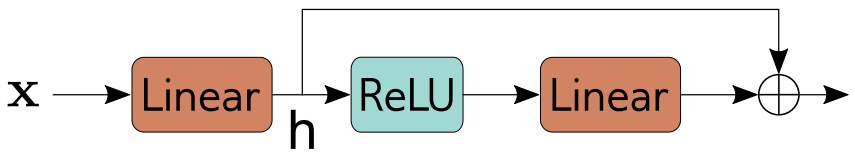
x + ReLU(h) >= x

can only increase input



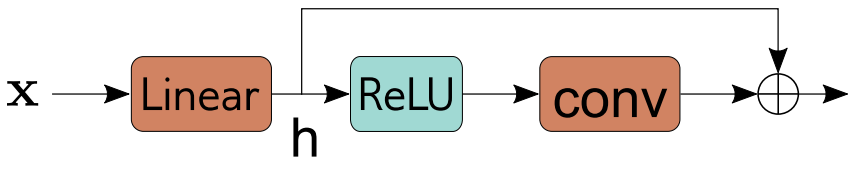
x + linear(ReLU(x)) = x if x < 0

does nothing for neg inputs



h + linear(ReLU(h))

works with all inputs



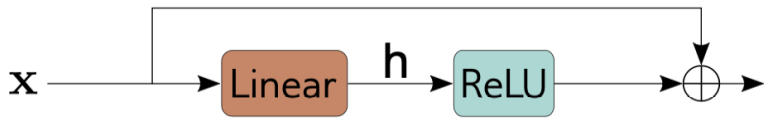
h + conv(ReLU(h))

works with all inputs

Residual Networks

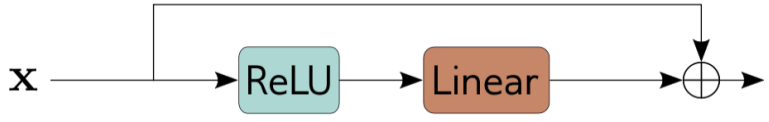
Residual network order

adapted from: UDL, Prince



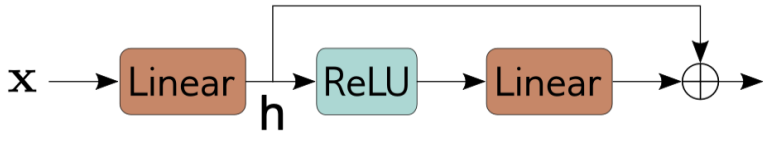
x + ReLU(h) >= x

can only increase input



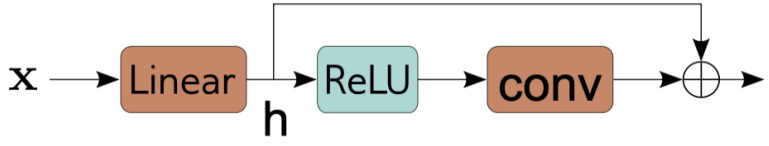
x + linear(ReLU(x)) = x if x < 0

does nothing for neg inputs



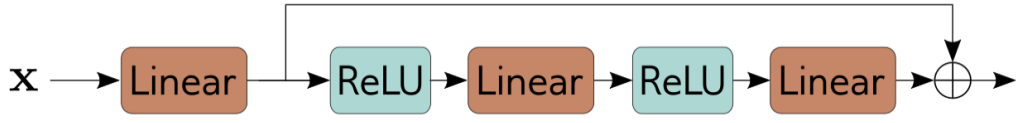
h + linear(ReLU(h))

works with all inputs



h + conv(ReLU(h))

works with all inputs



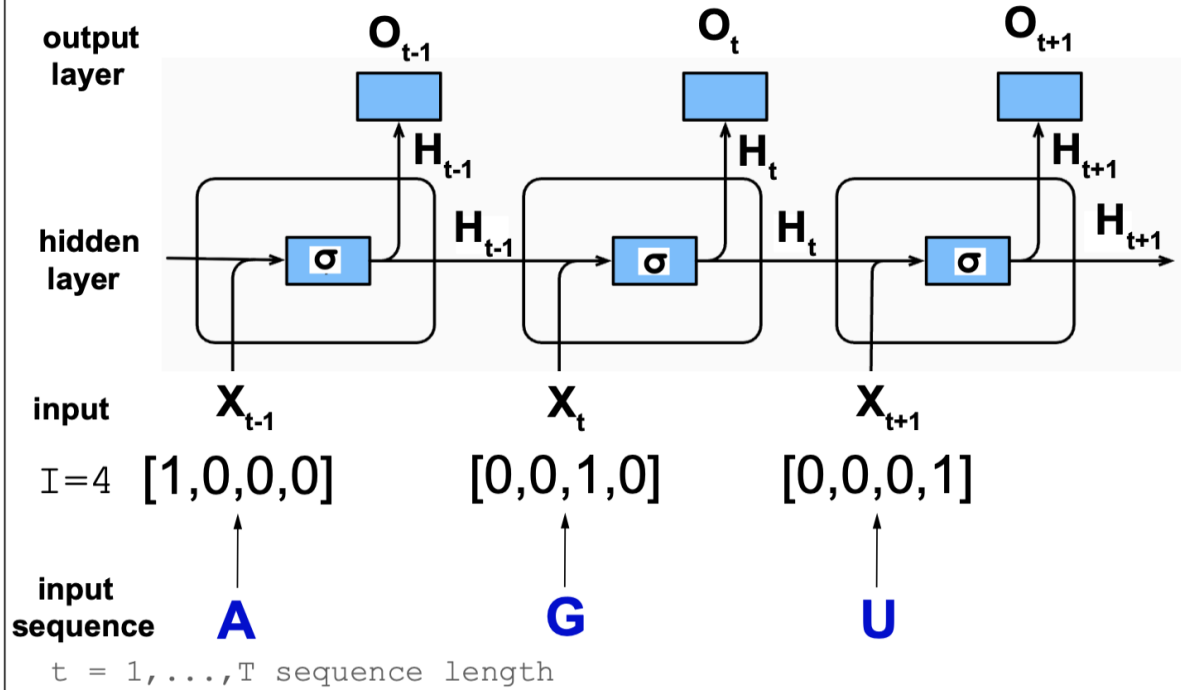
Recurrent Neural Networks

MLPs/CNNs process all inputs in **parallel**

RNNs take a **sequential** approach to processing the inputs

Recurrent Neural Networks

simple RNN (with hidden layer)



t = 1,...,T sequence length

inputs	outputs
H_{t-1}[H]	H_t[H]
X_t [I]	O_t[O]

I=input dimension
H=hidden dimension
O=output dimension

weights-hidden layer

W_{xh}[I,H] W_{hh}[H,H] b_h[H]

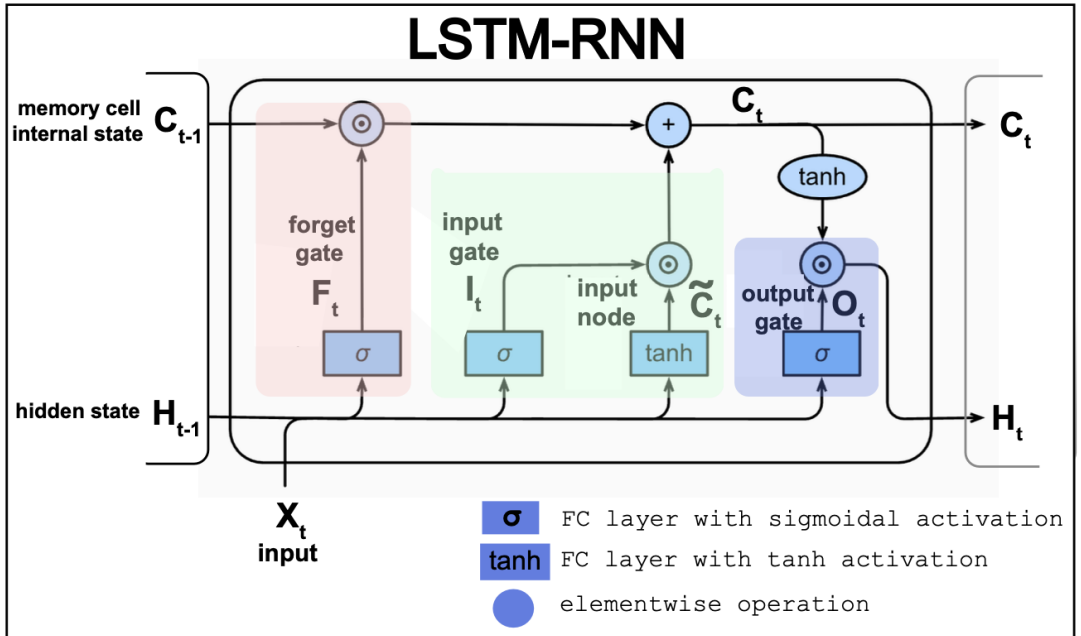
weights-output layer

W_{ho}[H,O] b_o[O]

$$H_t = \sigma(X_t * W_{xh} + H_{t-1} * W_{hh} + b_h)$$

$$O_t = H_t * W_{ho} + b_o$$

Long Short-Term Memory RNN



inputs	outputs	gates	weights		
H_{t-1}[H]	H_t[H]	F_t[H]	W_{xf}[I,H]	W_{hf}[H,H]	b_f[H]
C_{t-1}[H]	C_t[H]	I_t[H]	W_{xi}[I,H]	W_{hi}[H,H]	b_i[H]
X_t [I]		O_t[H]	W_{xo}[I,H]	W_{ho}[H,H]	b_o[H]
I=input dimension		$\tilde{C}_t$[H]	W_{xc}[I,H]	W_{hc}[H,H]	b_c[H]
H=hidden dimension					

gates

$$F_t = \sigma(X_t * W_{xf} + H_{t-1} * W_{hf} + b_f)$$
$$I_t = \sigma(X_t * W_{xi} + H_{t-1} * W_{hi} + b_i)$$
$$O_t = \sigma(X_t * W_{xo} + H_{t-1} * W_{ho} + b_o)$$
$$\tilde{C}_t = \tanh(X_t * W_{xc} + H_{t-1} * W_{hc} + b_c)$$

outputs

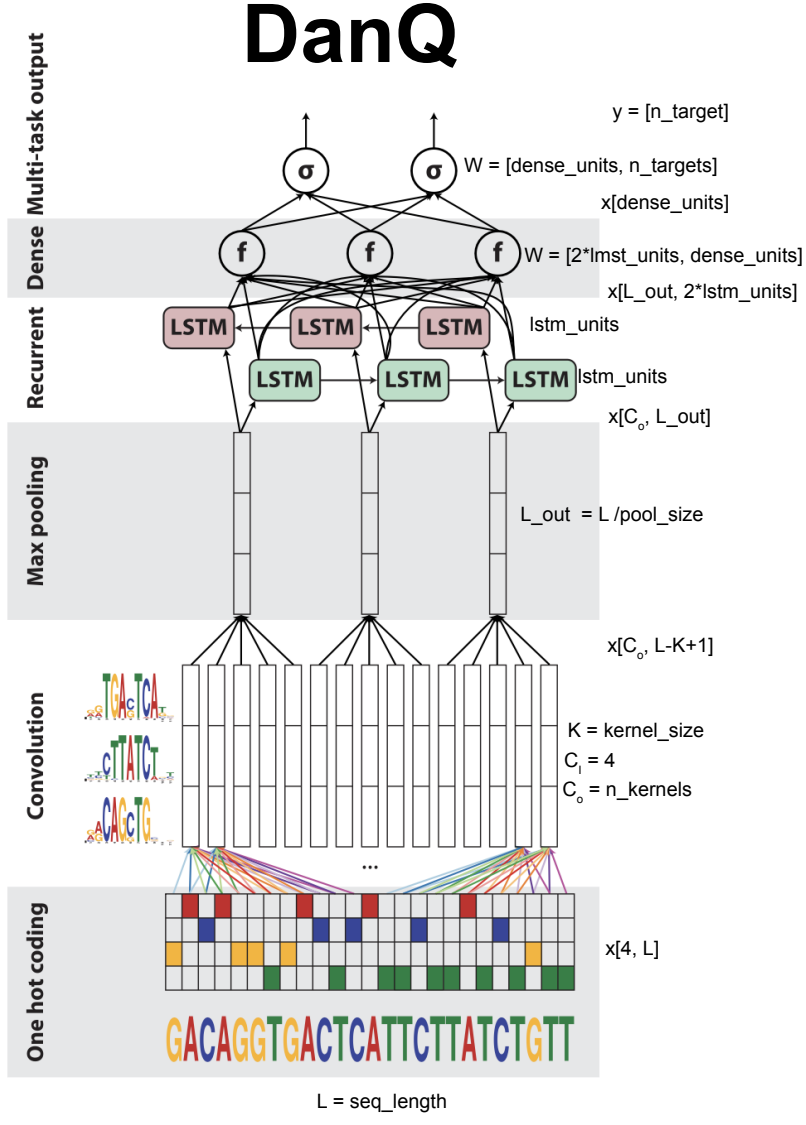
$$C_t = F_t \odot C_{t-1} + I_t \odot \tilde{C}_t$$
$$H_t = O_t \odot \tanh(C_t)$$

RNN code

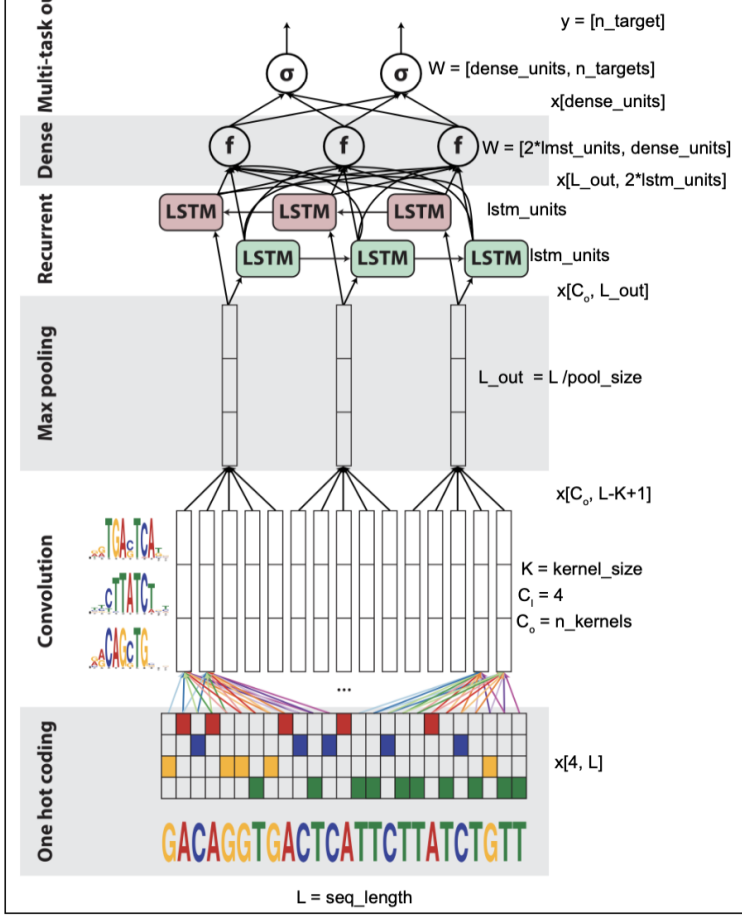
```
# RNN parameters
input_size  = 4   # 1-hot DNA/RNA sequence
hidden_size = 18  # RNN hidden units
num_classes = 2   # binary classification

# data
#    x     [batch, seq_length, input_size]
#    labels[batch]
#
# outputs
#    rnn_out[batch, seq_length, hidden_size]
#    last_out[batch, hidden_size]
#    logits[batch,num_classes] = fc(last_out)
#
# then compare
#     labels[batch]
#     preds[batch] = torch.argmax(logits, dim=1)
#
# RNN Model
class SimpleRNNClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)
    def forward(self, x):
        rnn_out, h_n = self.rnn(x)       # rnn_out: (batch, seq_len, hidden_size)
        last_out = rnn_out[:, -1, :]     # get output from last timestep
        return self.fc(last_out)         # pass through final classifier


model = SimpleRNNClassifier(input_size, hidden_size, num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

```
# LSTM Model
class SimpleLSTMClassifier(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super().__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out, (h_n, c_n) = self.lstm(x)
        # out: (batch, seq_len, hidden_size)
        last_out = out[:, -1, :]  # Use output from last timestep
        logits = self.fc(last_out)
        return logits

model = SimpleLSTMClassifier(input_size, hidden_size, num_classes)
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.01)
```

DanQ: a hybrid convolutional and recurrent deep neural network for quantifying the function of DNA sequences



DanQ




```python
class DanQ(nn.Module):
    def __init__(self,
                 seq_len,
                 n_targets,
                 n_kernels=40,
                 kernel_size=20,
                 pool_size=13,
                 lstm_units=20,
                 dense_units=25,
                 dropout=0.5):
        super().__init__()
        self.conv = nn.Conv1d(4, n_kernels, kernel_size)    # weights[4,n_kernels,kernel_size]
        self.relu = nn.ReLU()
        self.pool = nn.MaxPool1d(pool_size)
        self.bi_lstm = nn.LSTM(                             # weights[n_kernels,lstm_units]
            input_size=n_kernels,
            hidden_size=lstm_units,
            num_layers=1,
            bidirectional=True,
            batch_first=True
        )
        self.dropout = nn.Dropout(dropout)
        self.dense = nn.Linear(2 * lstm_units, dense_units)  # weights[2*lstm_units,dense_unit]
        self.output = nn.Linear(dense_units, n_targets)      # weights[dense_unit,n_targets]
        self.sigmoid = nn.Sigmoid()

    def forward(self, x):
        # x shape: [batch_size, 4, L=seq_len]
        # 1dconv
        print("input", x.shape)

        x = self.conv(x)              # output[batch, n_kernels, L-kernel_size+1] (no padding)
        x = self.relu(x)
        print("prepool", x.shape)
        x = self.pool(x)              # output[batch, n_kernels, L_out]   L_out = (L-kernel_size+1)/pool_size
        print("pool", x.shape)

        # bi-LSTM
        # input_size  = n_kernels
        # hidden_size = lstm_units (x 2 due to the bi-directionality)
        x = x.permute(0, 2, 1)        # swap to (batch, L_out, n_kernels) for LSTM
        lstm_out, _ = self.bi_lstm(x) #lstm_out[batch, L_out, 2*lstm-units]

        # output options:
        #    (1) Use all positions (many-to-many)                                [batch, L_out, 2*lstm-units]
        #    (2) Use only the last output timestep's hidden state (many-to-one) [batch,        2*lstm-units]
        #    (3) take max over time (used here)                                  [batch,        2*lstm-units]
        lstm_out, _ = torch.max(lstm_out, dim=1) #[batch, 2*lstm-units]

        # dropout
        x = self.dropout(lstm_out) #x[batch, 2*lstm-units]

        # first dense layer weights[2*lstm-units, dense_unit]
        y = self.dense(x)          #y[batch, dense_unit]
        y = self.relu(y)

        # second dense layer weights[2*lstm-units, dense_unit, n_targets]
        # we leave the outputs as logits
        y = self.output(y)         #y[batch, n_targets]

        return y
```