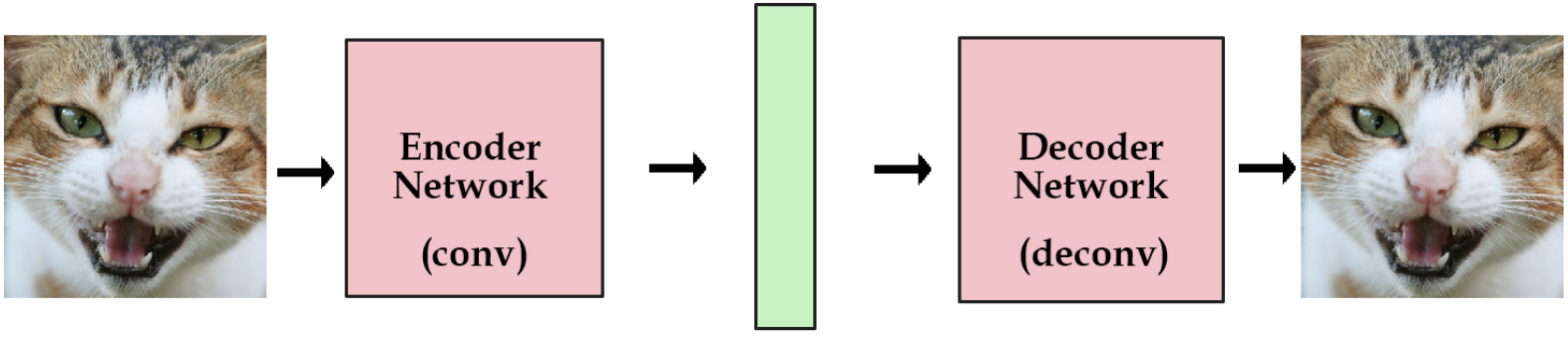
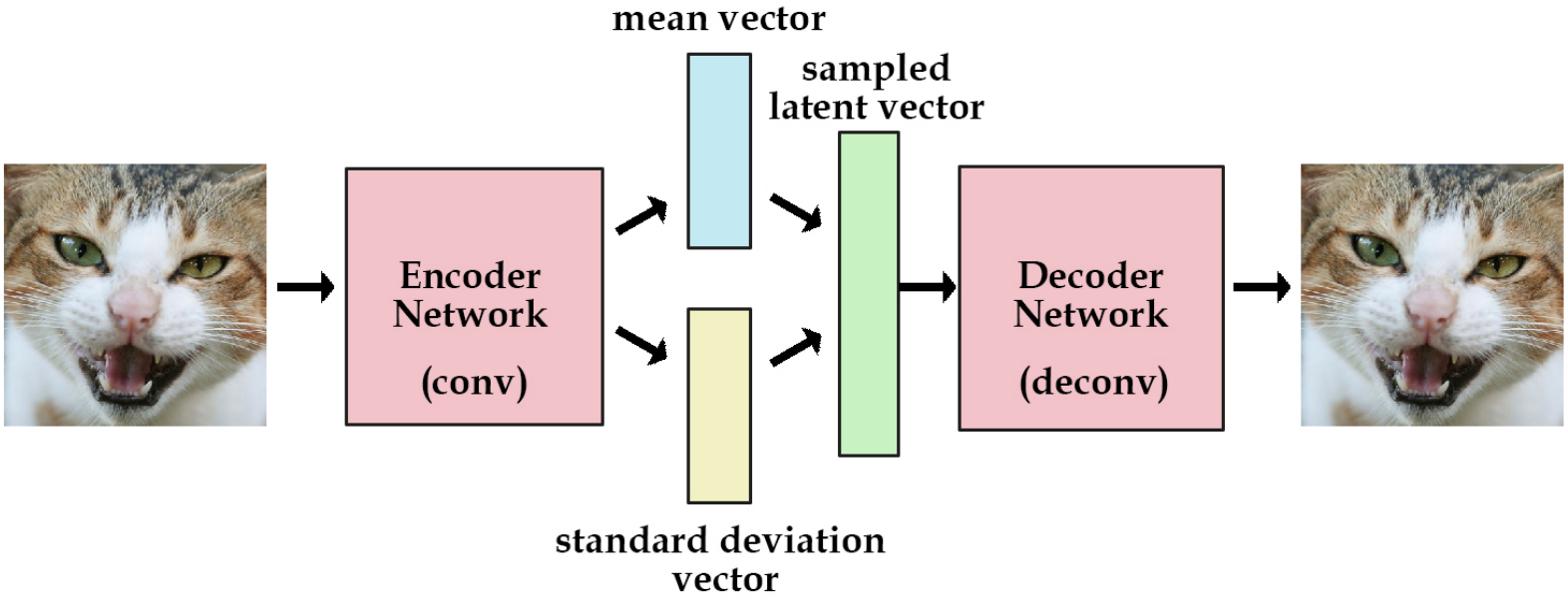
Autoencoders

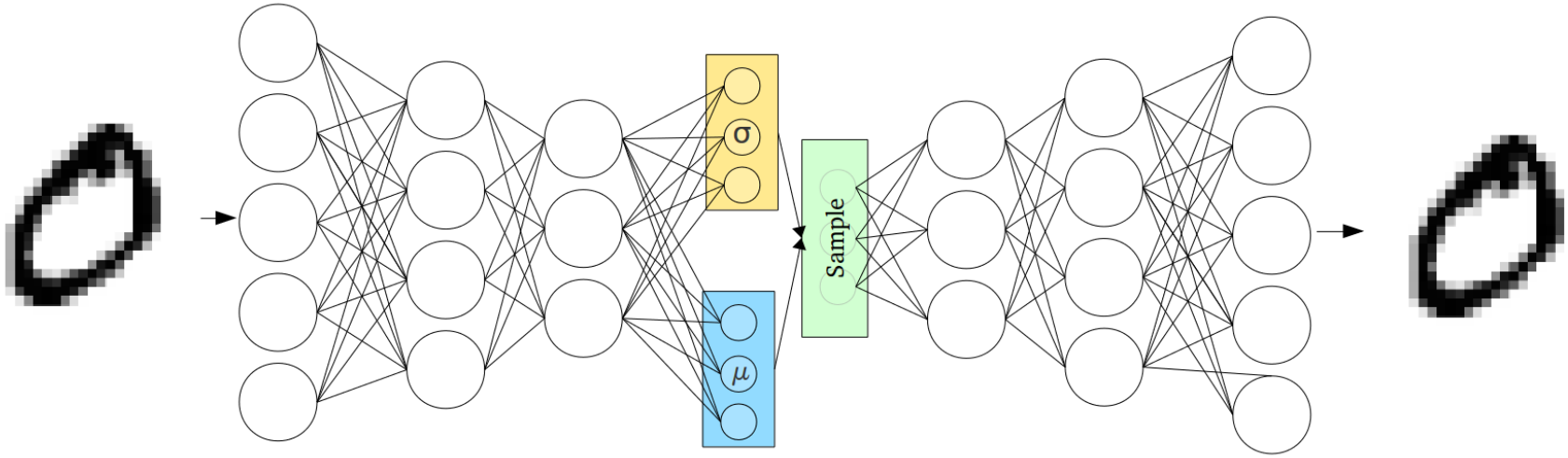
with

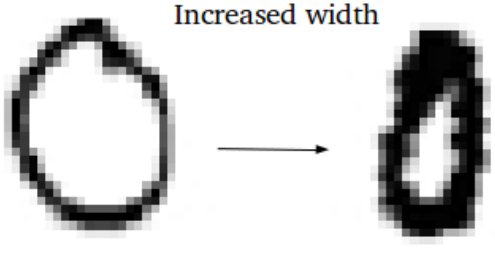
Neural Networks

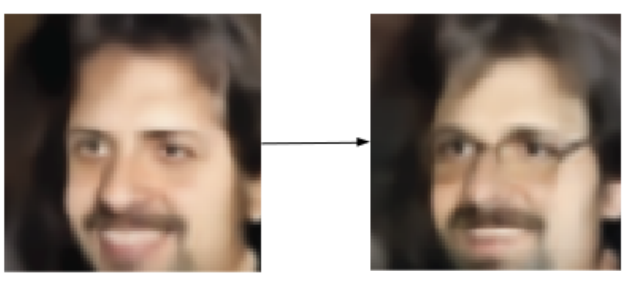
CCTS 40500

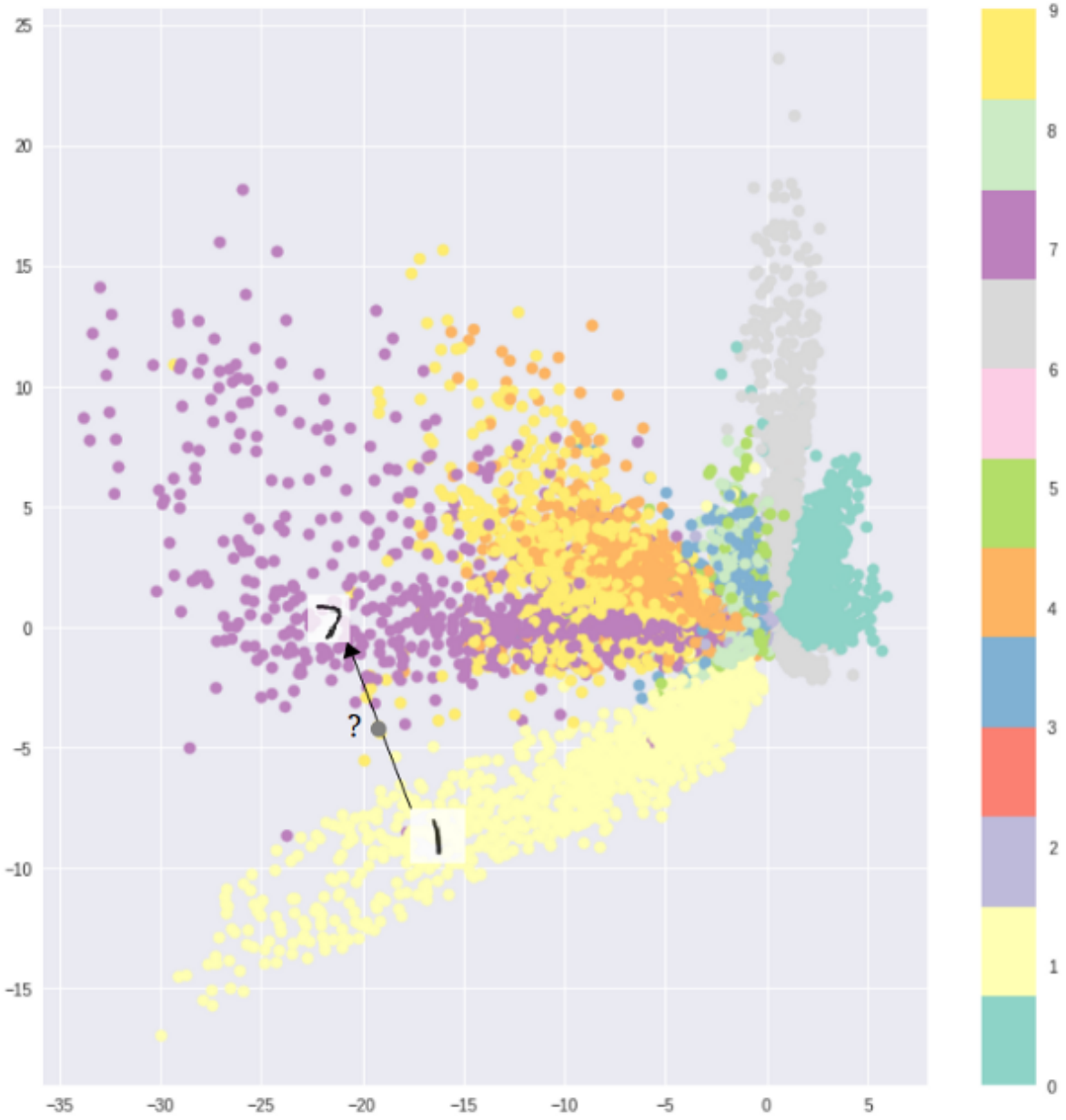

Encoder Network
(conv)
Decoder Network
(deconv)
latent vector / variables


mean vector
sampled
latent vector
Encoder
Network
(conv)
Decoder
Network
(deconv)
standard deviation
vector


σ
μ
Sample

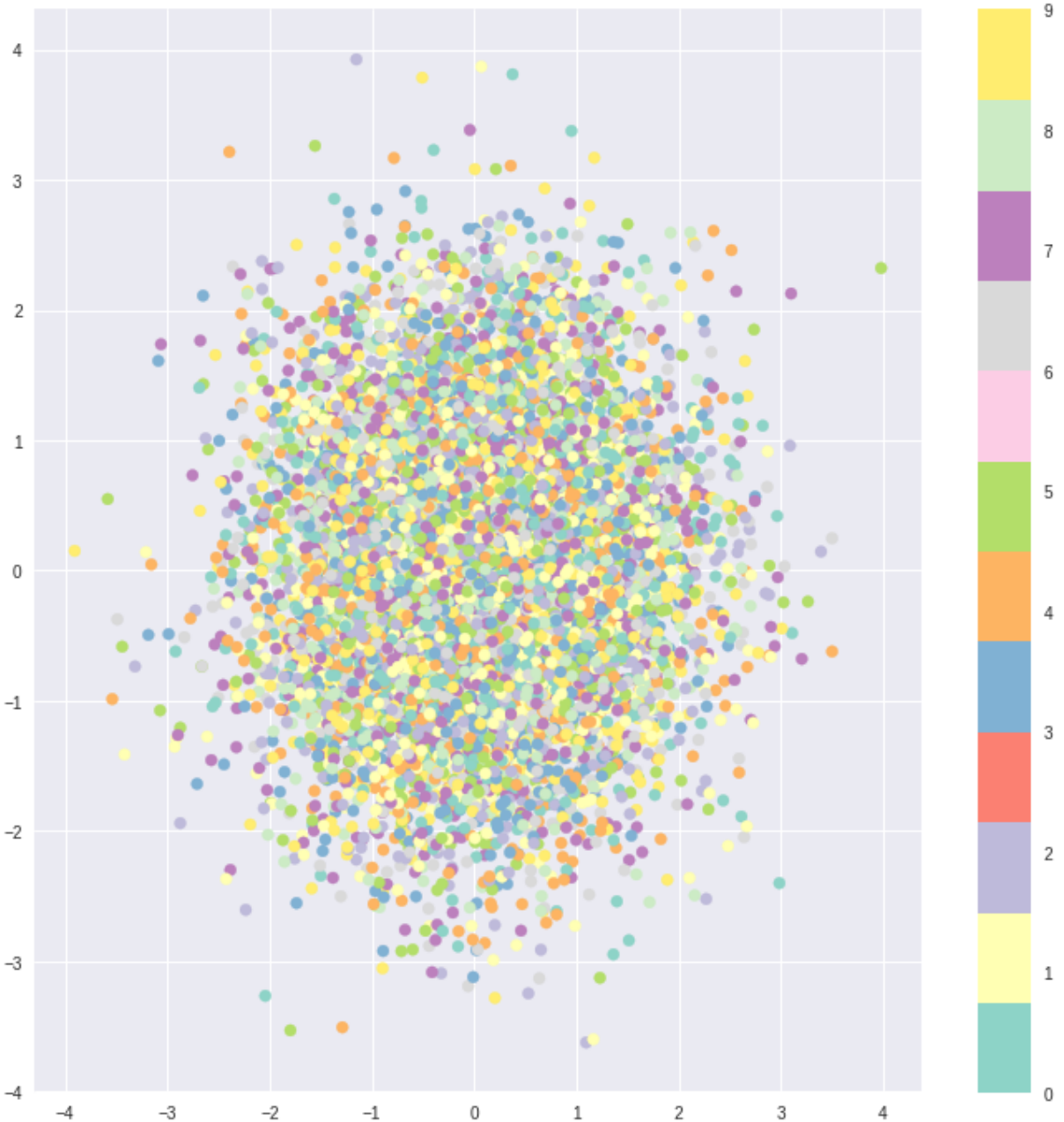

Increased width


Glasses

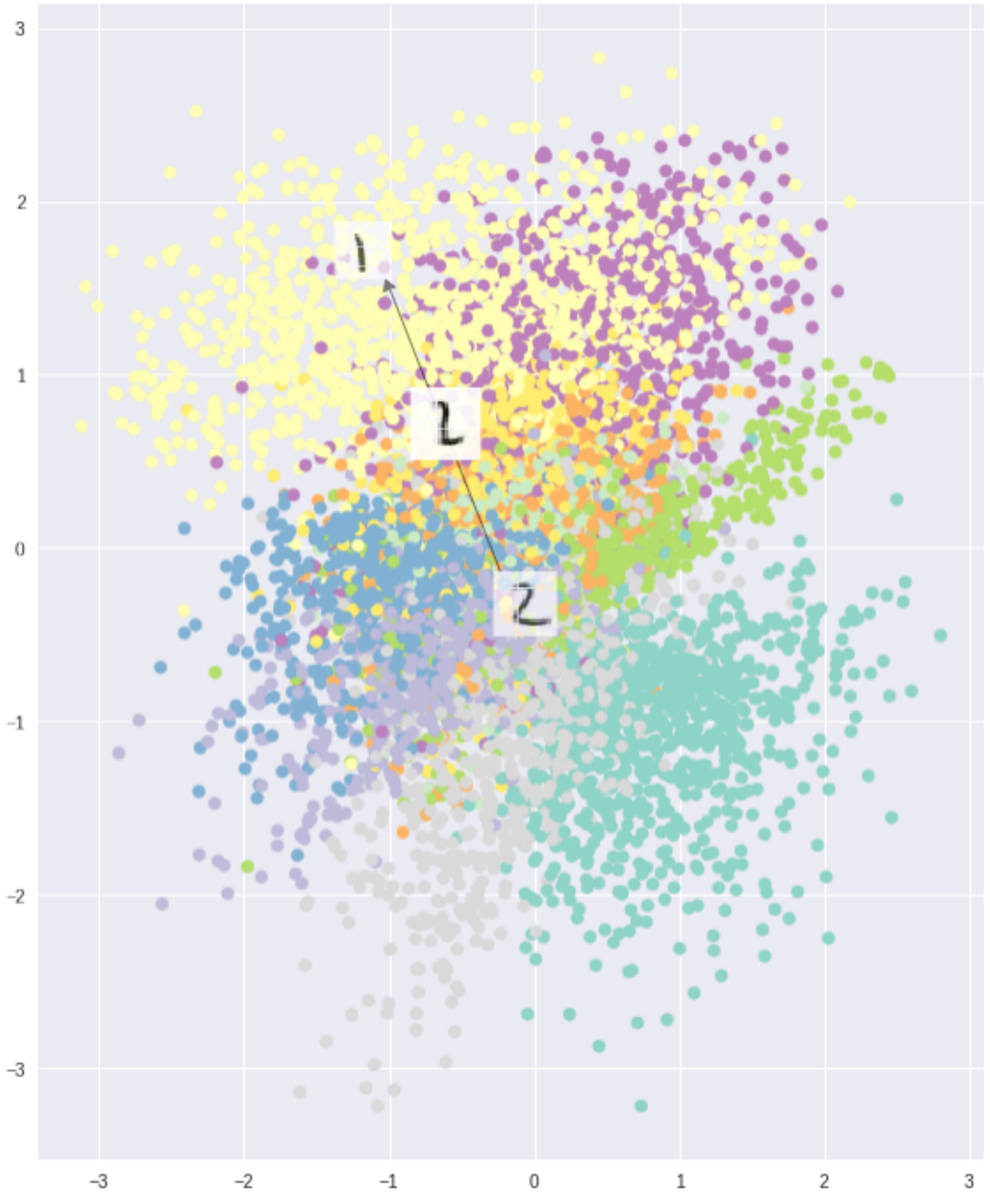

25
20
15
10
5
0
−5
−10
−15
−35
−30
−25
−20
−15
−10
−5
0
5
?
9
8
7
6
5
4
3
2
1
0

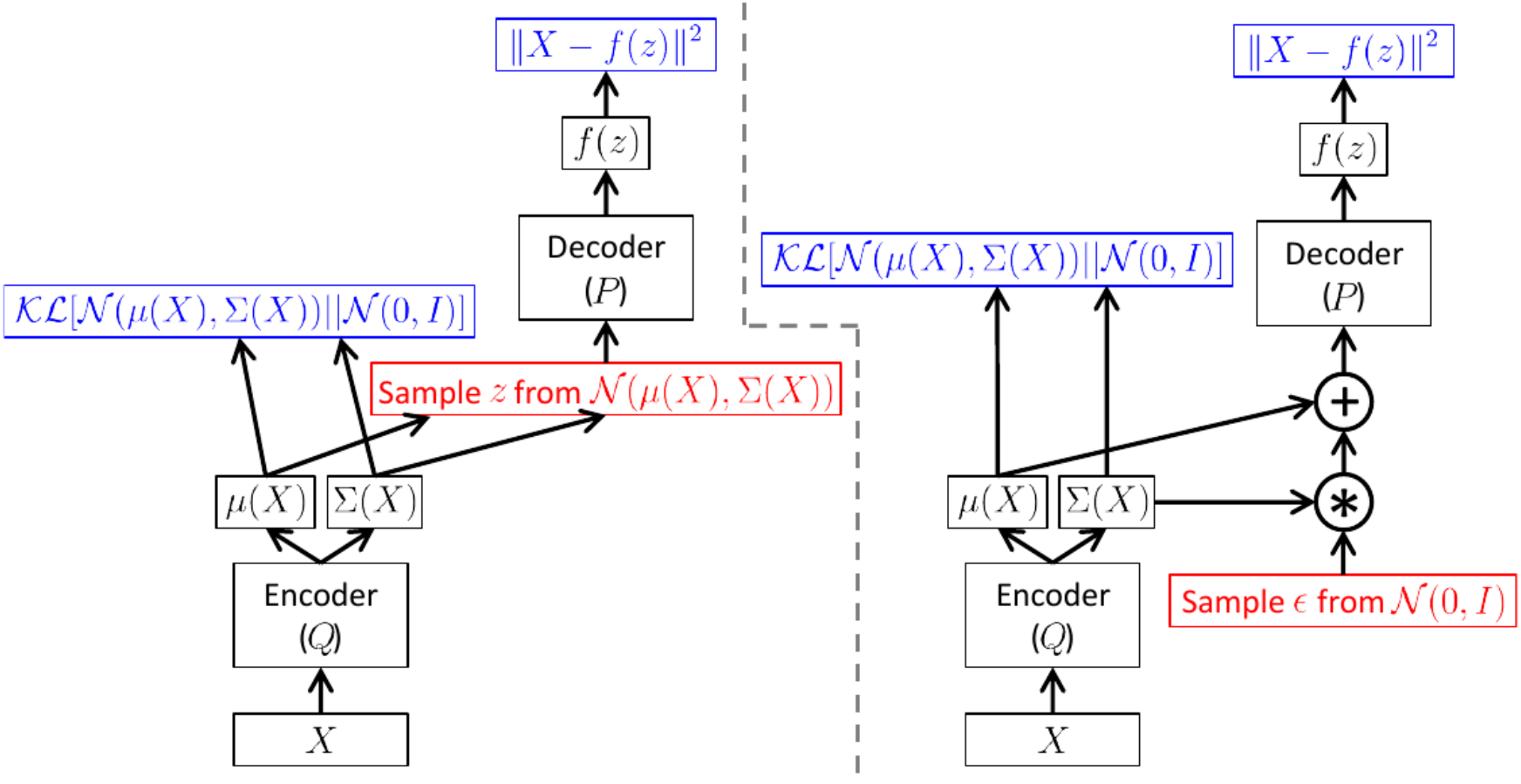
Pure KL Loss

$$\sum_{i=1}^{n} \sigma_i^2 + \mu_i^2 - \log(\sigma_i) - 1$$



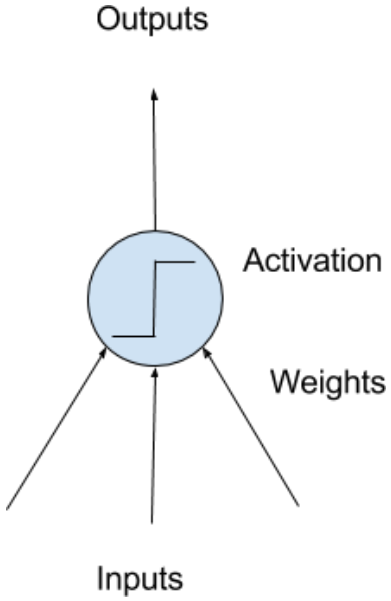
KL + Input-Output Loss



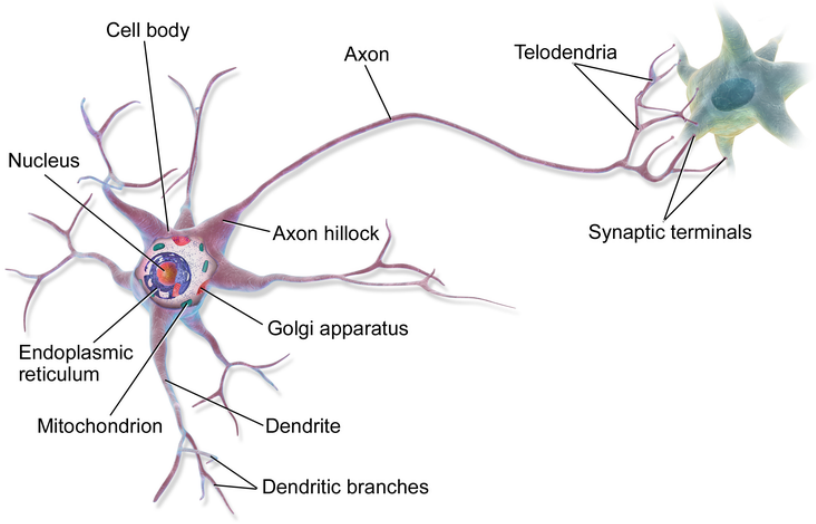

$\|X - f(z)\|^2$
$f(z)$
Decoder
(P)
$\mathcal{KL}[\mathcal{N}(\mu(X), \Sigma(X)) || \mathcal{N}(0, I)]$
Sample z from $\mathcal{N}(\mu(X), \Sigma(X))$
$\mu(X)$
$\Sigma(X)$
Encoder
(Q)
X
$\|X - f(z)\|^2$
$f(z)$
Decoder
(P)
$\mathcal{KL}[\mathcal{N}(\mu(X), \Sigma(X)) || \mathcal{N}(0, I)]$
$\mu(X)$
$\Sigma(X)$
Encoder
(Q)
X
Sample ϵ from $\mathcal{N}(0, I)$

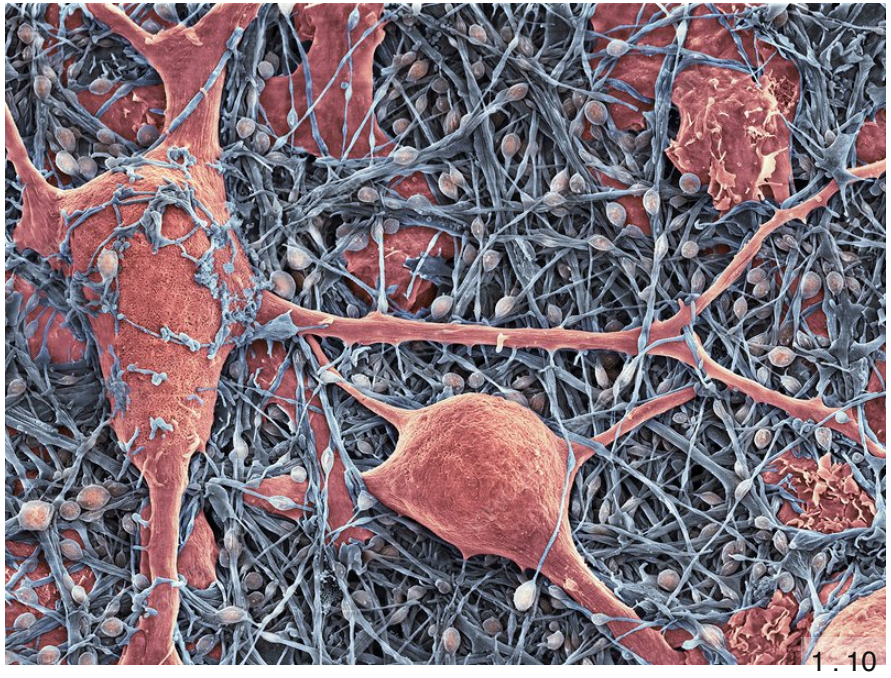
Neurons

- The building block for neural networks are artificial neurons.
- These are simple computational units that have weighted input signals and produce an output signal using an activation function.

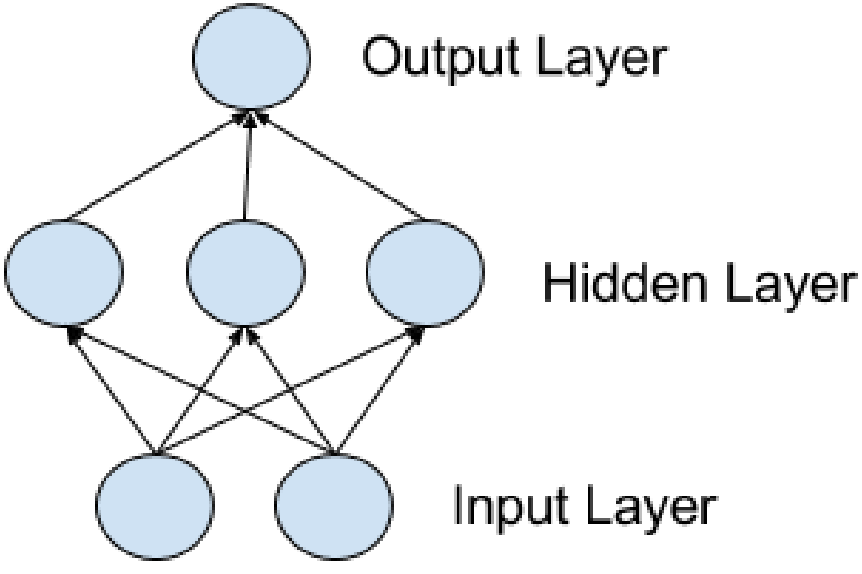


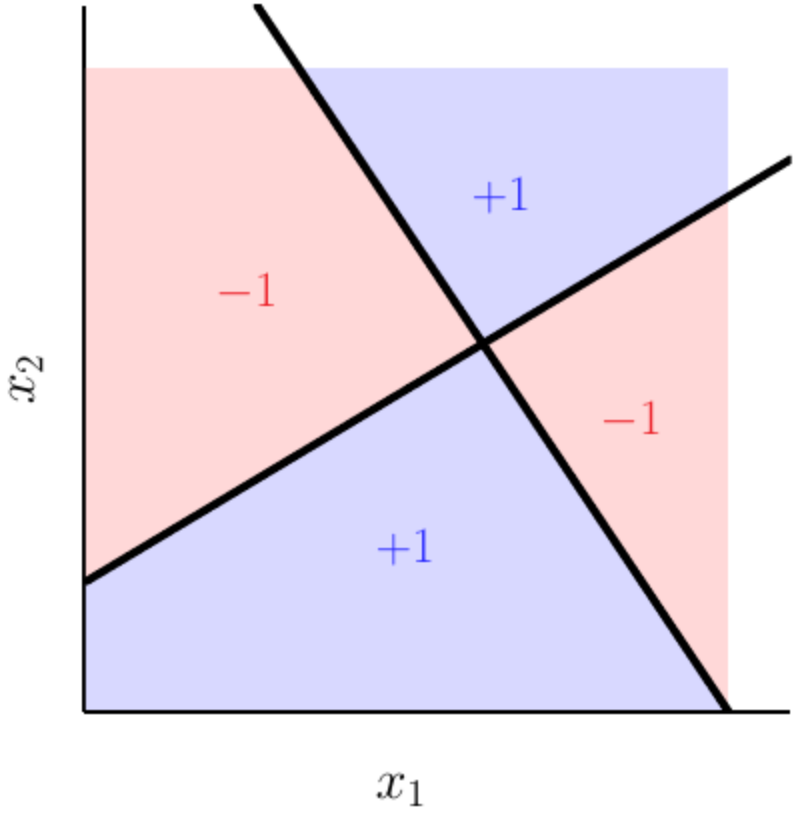
Biological Neurons

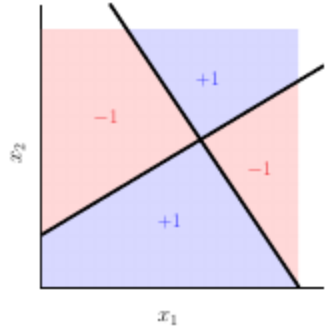




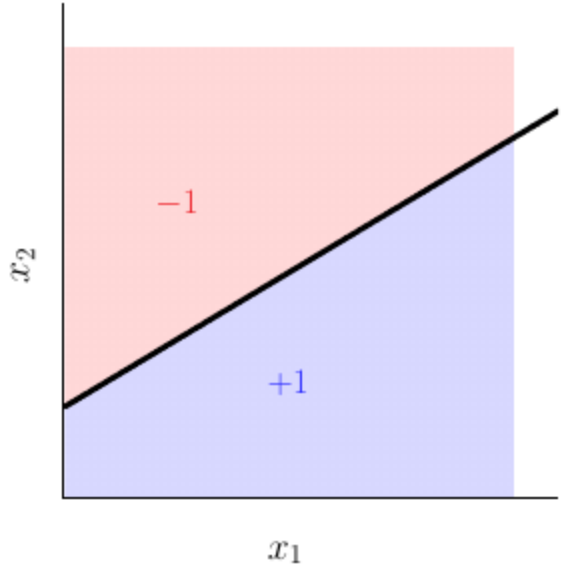
- Neuron Weights
- Activation
- Networks of Neurons



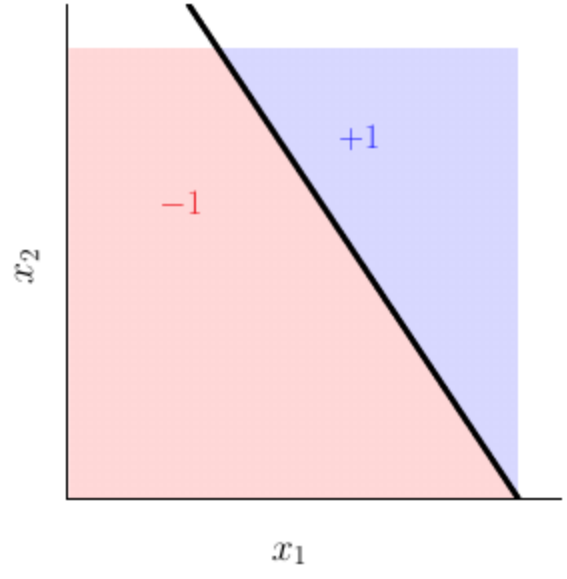

+1
−1
−1
+1
x_2
x_1



$$f = h_1\overline{h_2} + \overline{h_1}h_2$$

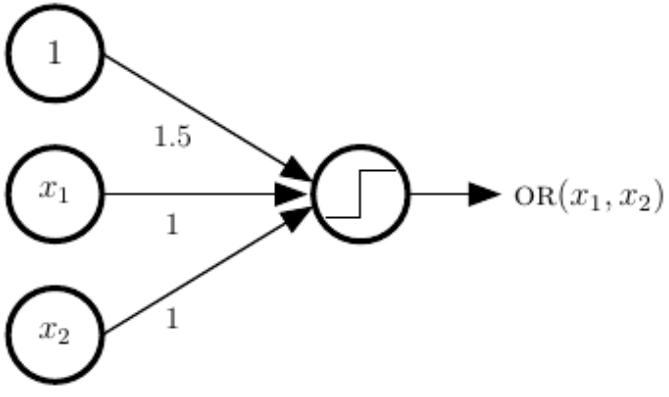


$$h_1(\mathbf{x}) = \text{sign}(\mathbf{w}_1^{\text{T}}\mathbf{x})$$

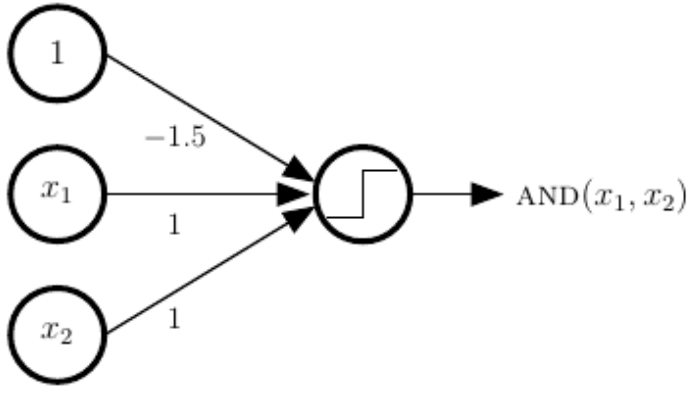


$$h_2(\mathbf{x}) = \text{sign}(\mathbf{w}_2^{\text{T}}\mathbf{x})$$

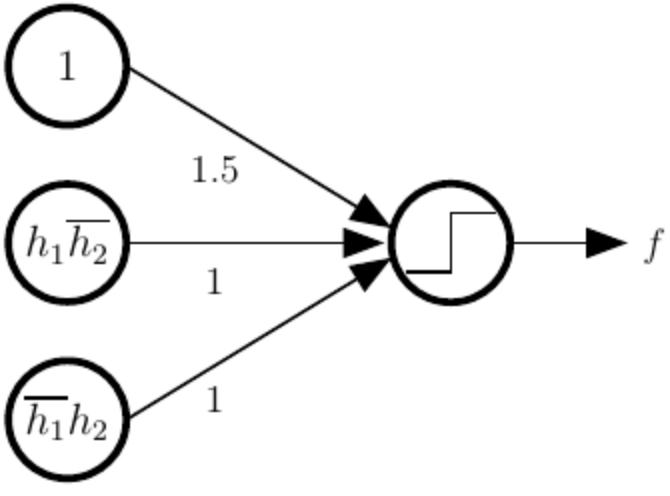
$\text{OR}(x_1, x_2) = \text{sign}(x_1 + x_2 + 1.5)$



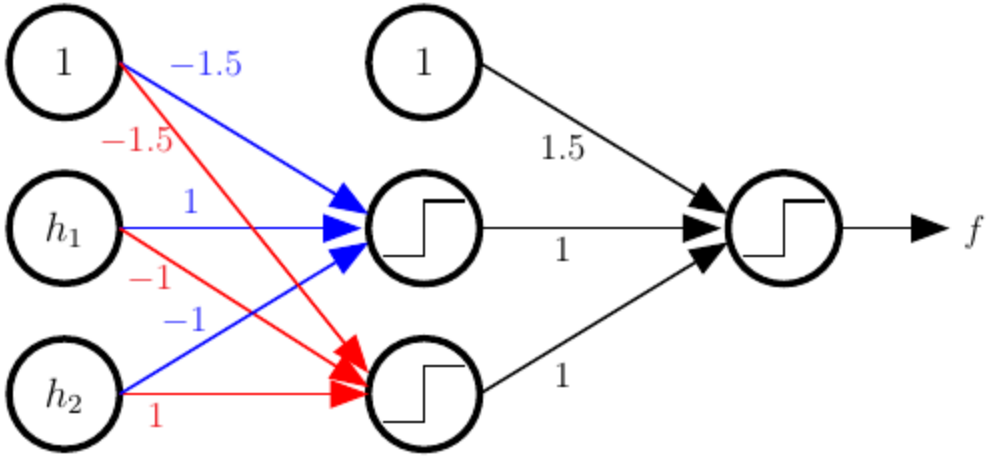
$\text{AND}(x_1, x_2) = \text{sign}(x_1 + x_2 - 1.5)$



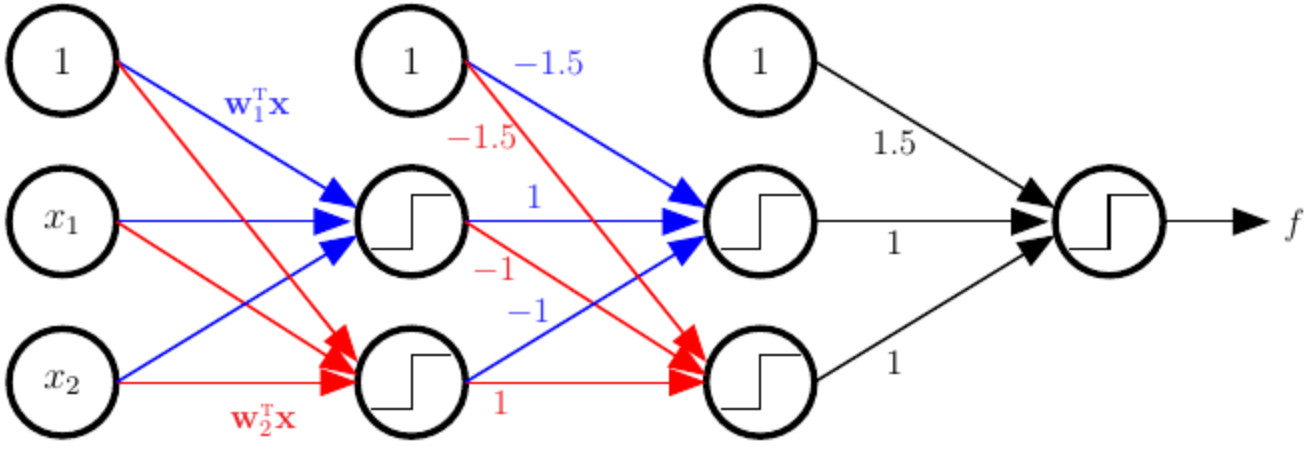
$$f = h_1\overline{h_2} + \overline{h_1}h_2$$

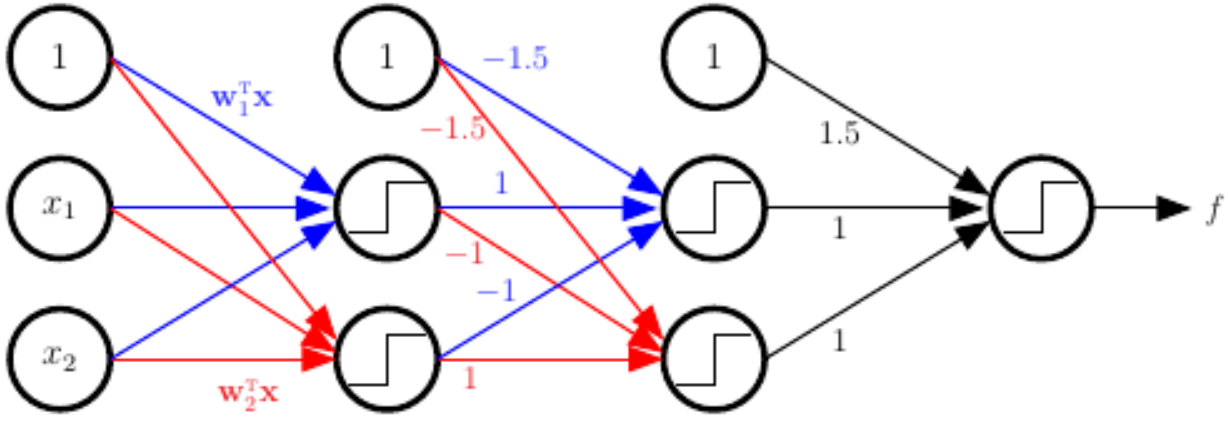


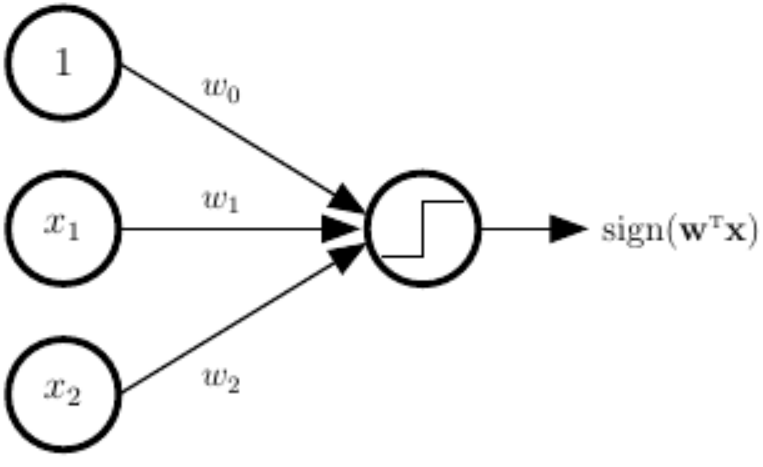
$$f = h_1\overline{h_2} + \overline{h_1}h_2$$



$$f = h_1\overline{h_2} + \overline{h_1}h_2$$



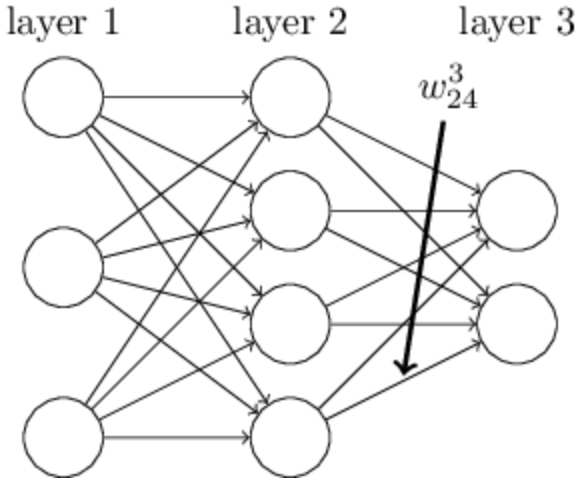

1
x_1
x_2
$\mathbf{w}_1^{\mathrm{T}}\mathbf{x}$
$\mathbf{w}_2^{\mathrm{T}}\mathbf{x}$
1
−1.5
−1.5
1
−1
−1
1
1
1
1.5
1
1
f


1
x_1
x_2
w_0
w_1
w_2
$\mathrm{sign}(\mathbf{w}^{\mathrm{T}}\mathbf{x})$

Any boolean function can be realized by a MLP with one hidden layer.

Any bounded continuous function can be approximated with arbitrary precision by a MLP with one hidden layer.

NN Computation

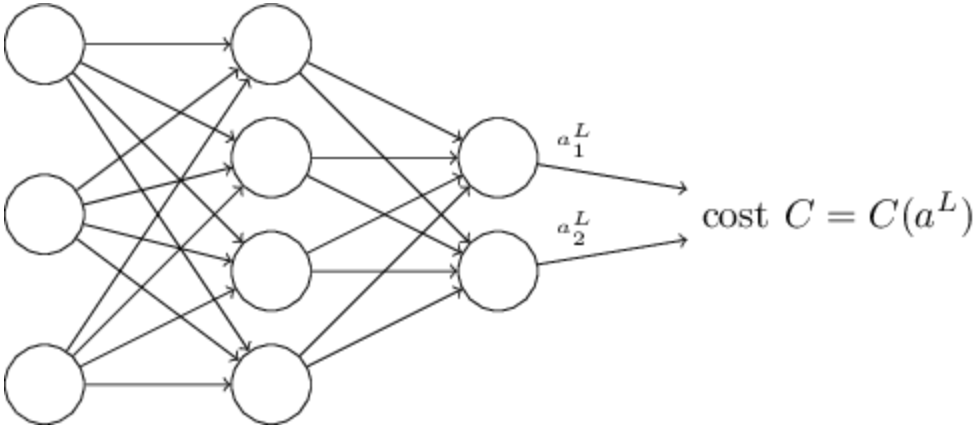


w^l_{jk} is the weight from the k^{th} neuron in the $(l-1)^{\text{th}}$ layer to the j^{th} neuron in the l^{th} layer

$$a^l_j = \sigma\left(\sum_k w^l_{jk} a^{l-1}_k + b^l_j\right)$$

$$a^l = \sigma(w^l a^{l-1} + b^l)$$

Loss Function



$$C = \frac{1}{2}\|y - a^L\|^2 = \frac{1}{2}\sum_j (y_j - a_j^L)^2$$

Hadamard Product

$$\begin{bmatrix} 1 \\ 2 \end{bmatrix} \odot \begin{bmatrix} 3 \\ 4 \end{bmatrix} = \begin{bmatrix} 1 * 3 \\ 2 * 4 \end{bmatrix} = \begin{bmatrix} 3 \\ 8 \end{bmatrix}$$

Summary: the equations of backpropagation

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

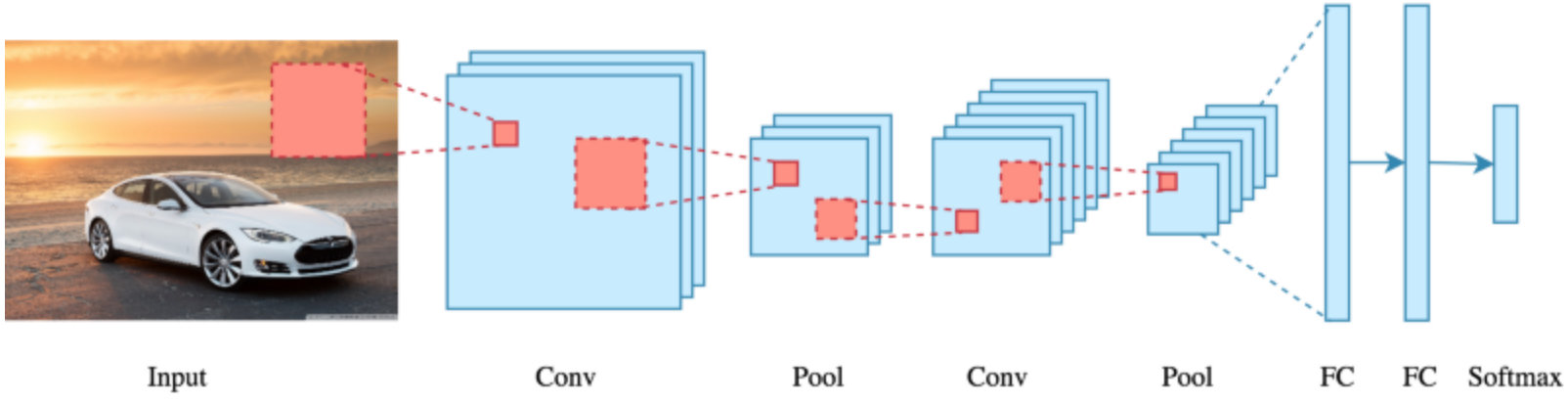
$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{BP3}$$

$$\frac{\partial C}{\partial w^l_{j\,k}} = a^{l-1}_k \delta^l_j \tag{BP4}$$

Backpropagation Code

```
class Network(object):
...
    def backprop(self, x, y):
        """Return a tuple "(nabla_b, nabla_w)" representing the
        gradient for the cost function C_x.  "nabla_b" and
        "nabla_w" are layer-by-layer lists of numpy arrays, similar
        to "self.biases" and "self.weights"."""
        nabla_b = [np.zeros(b.shape) for b in self.biases]
        nabla_w = [np.zeros(w.shape) for w in self.weights]
        # feedforward
        activation = x
        activations = [x] # list to store all the activations, layer by layer
        zs = [] # list to store all the z vectors, layer by layer
        for b, w in zip(self.biases, self.weights):
            z = np.dot(w, activation)+b
            zs.append(z)
            activation = sigmoid(z)
            activations.append(activation)
        # backward pass
        delta = self.cost_derivative(activations[-1], y) * \
            sigmoid_prime(zs[-1])
        nabla_b[-1] = delta
        nabla_w[-1] = np.dot(delta, activations[-2].transpose())
        # Note that the variable l in the loop below is used a little
        # differently to the notation in Chapter 2 of the book.  Here,
        # l = 1 means the last layer of neurons, l = 2 is the
        # second-last layer, and so on.  It's a renumbering of the
        # scheme in the book, used here to take advantage of the fact
        # that Python can use negative indices in lists.
        for l in xrange(2, self.num_layers):
            z = zs[-l]
            sp = sigmoid_prime(z)
            delta = np.dot(self.weights[-l+1].transpose(), delta) * sp
            nabla_b[-l] = delta
            nabla_w[-l] = np.dot(delta, activations[-l-1].transpose())
        return (nabla_b, nabla_w)

...

    def cost_derivative(self, output_activations, y):
        """Return the vector of partial derivatives \partial C_x /
        \partial a for the output activations."""
        return (output_activations-y)

def sigmoid(z):
    """The sigmoid function."""
    return 1.0/(1.0+np.exp(-z))

def sigmoid_prime(z):
    """Derivative of the sigmoid function."""
    return sigmoid(z)*(1-sigmoid(z))
```

Convolutional Neural Nets



1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Input

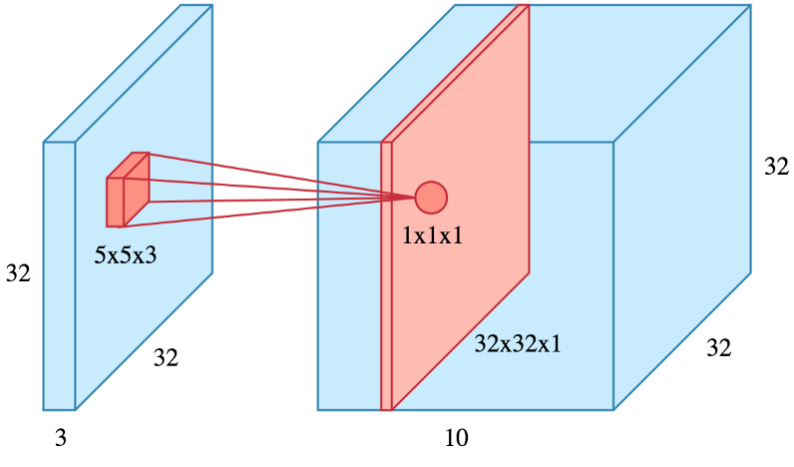
1	0	1
0	1	0
1	0	1

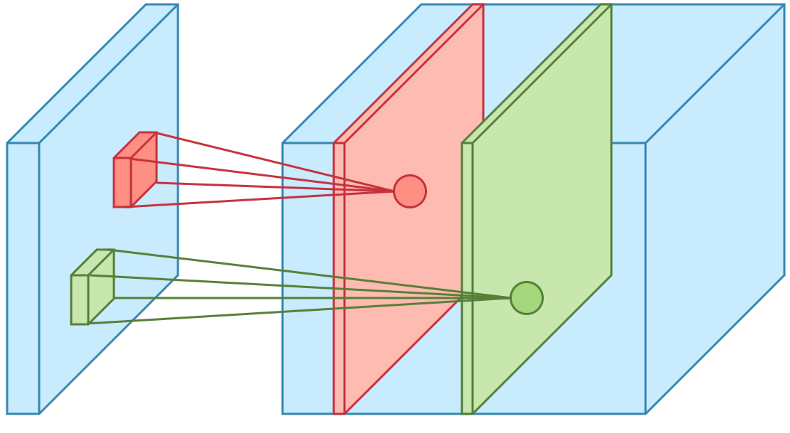
Filter / Kernel

Convolution

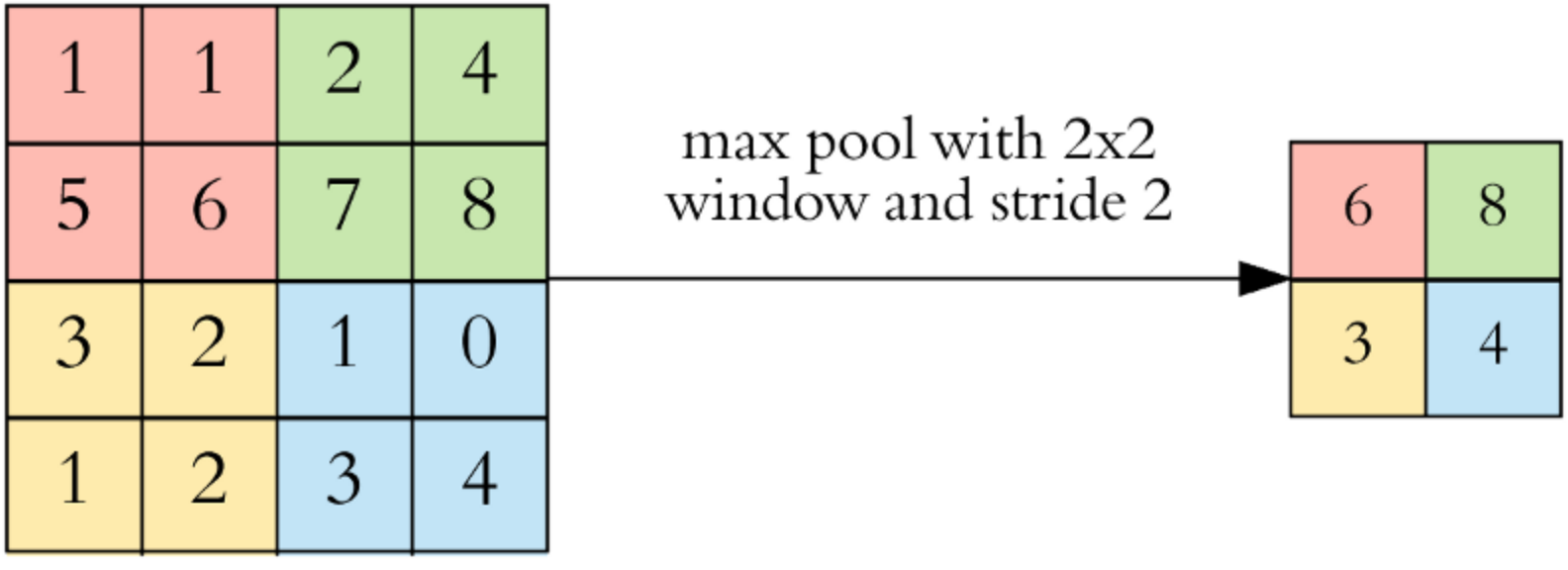
1	1x1	1x0	0x1	0
0	1x0	1x1	1x0	0
0	0x1	1x0	1x1	1
0	0	1	1	0
0	1	1	0	0

4	3	

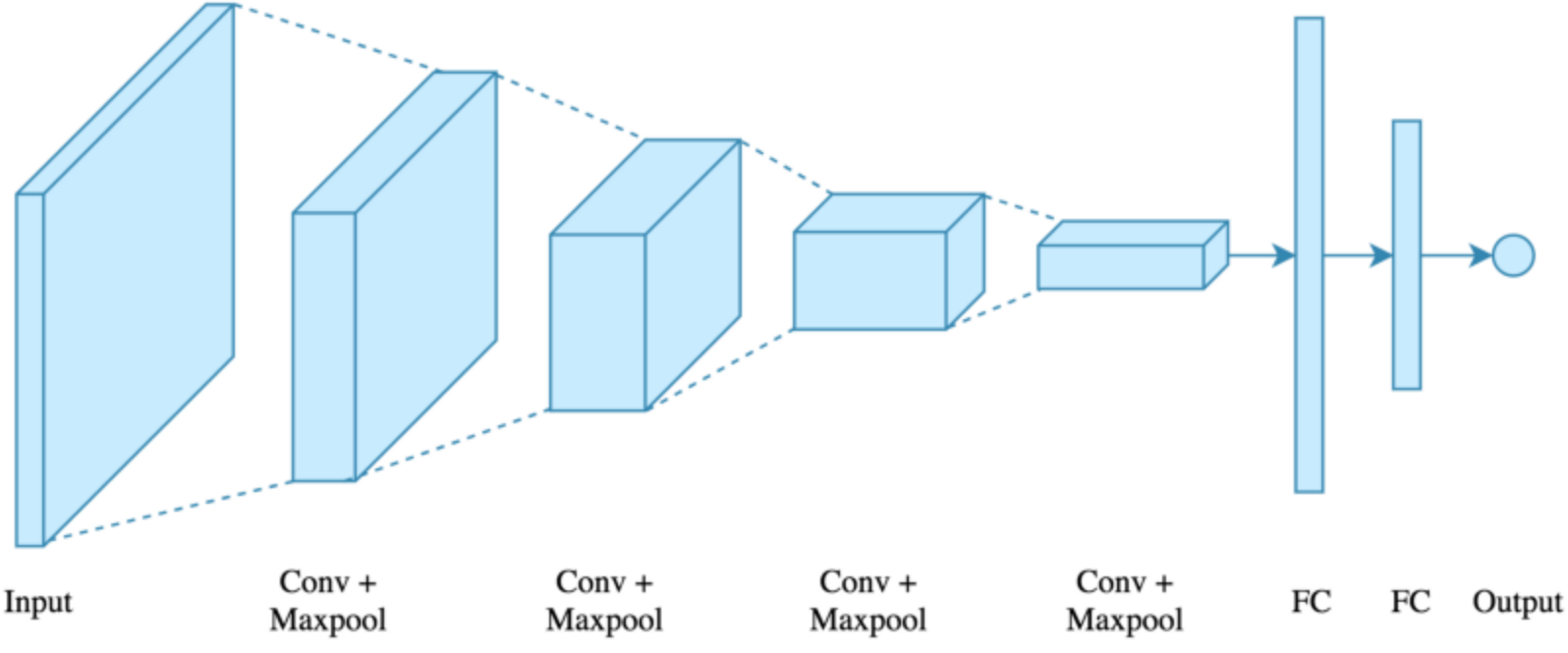




Pooling


1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2
window and stride 2
6 8
3 4

Algorithm


Input
Conv + Maxpool
Conv + Maxpool
Conv + Maxpool
Conv + Maxpool
FC
FC
Output

```
model = Sequential()
model.add(Conv2D(32, (3, 3), activation='relu', padding='same', name='conv_1',
                 input_shape=(150, 150, 3)))
model.add(MaxPooling2D((2, 2), name='maxpool_1'))
model.add(Conv2D(64, (3, 3), activation='relu', padding='same', name='conv_2'))
model.add(MaxPooling2D((2, 2), name='maxpool_2'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same', name='conv_3'))
model.add(MaxPooling2D((2, 2), name='maxpool_3'))
model.add(Conv2D(128, (3, 3), activation='relu', padding='same', name='conv_4'))
model.add(MaxPooling2D((2, 2), name='maxpool_4'))
model.add(Flatten())
model.add(Dropout(0.5))
model.add(Dense(512, activation='relu', name='dense_1'))
model.add(Dense(128, activation='relu', name='dense_2'))
model.add(Dense(1, activation='sigmoid', name='output'))
```

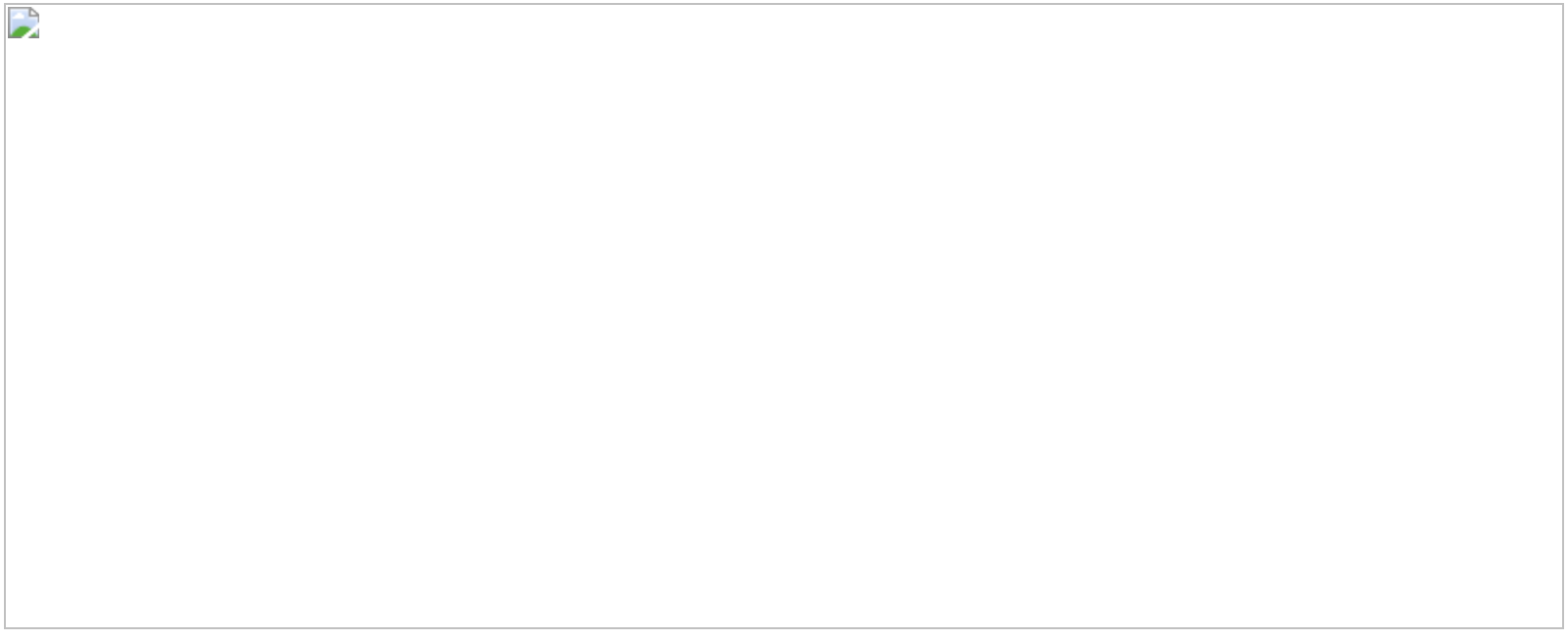
Conv2D: this method creates a convolutional layer. The first parameter is the filter count, and the second one is the filter size. For example in the first convolution layer we create 32 filters of size 3x3. We use *relu* non-linearity as activation.

MaxPooling2D: creates a maxpooling layer, the only argument is the window size

Flatten: After the convolution + pooling layers we flatten their output to feed into the fully connected layers as we discussed above.

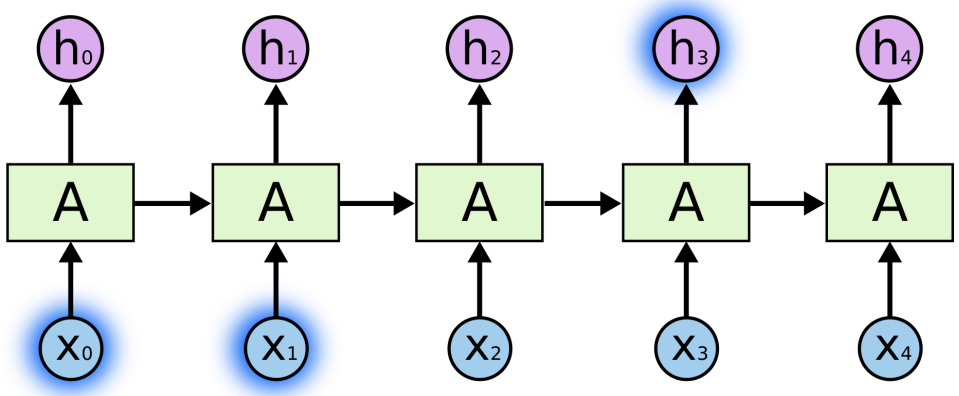
Dropout: Random structure variations

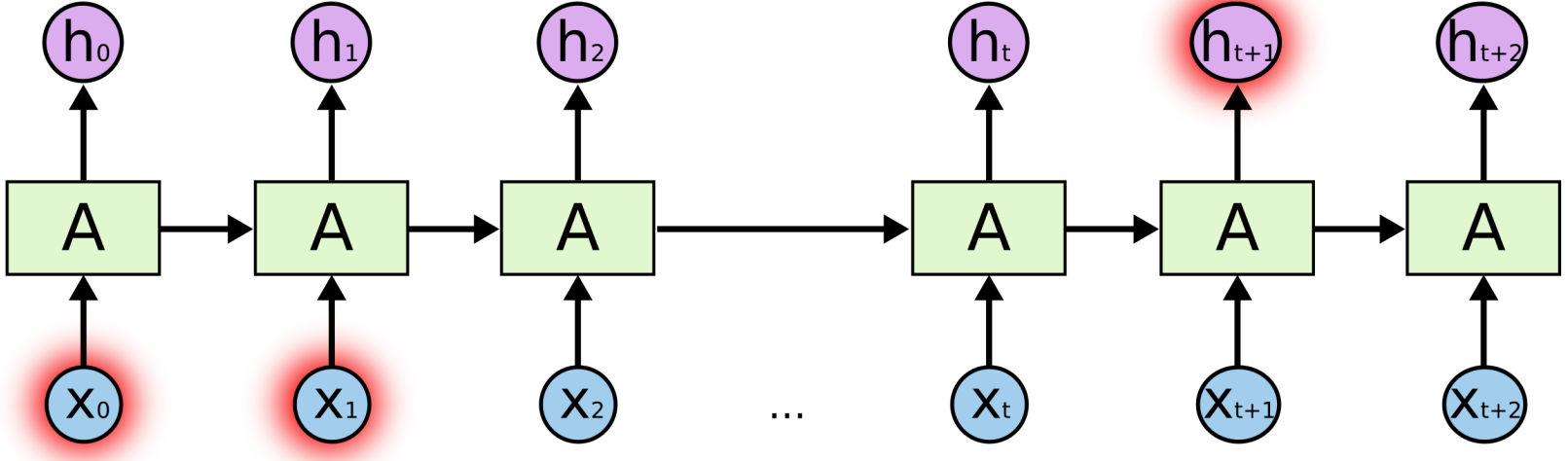
Recurrent Neural Networks (RNN)



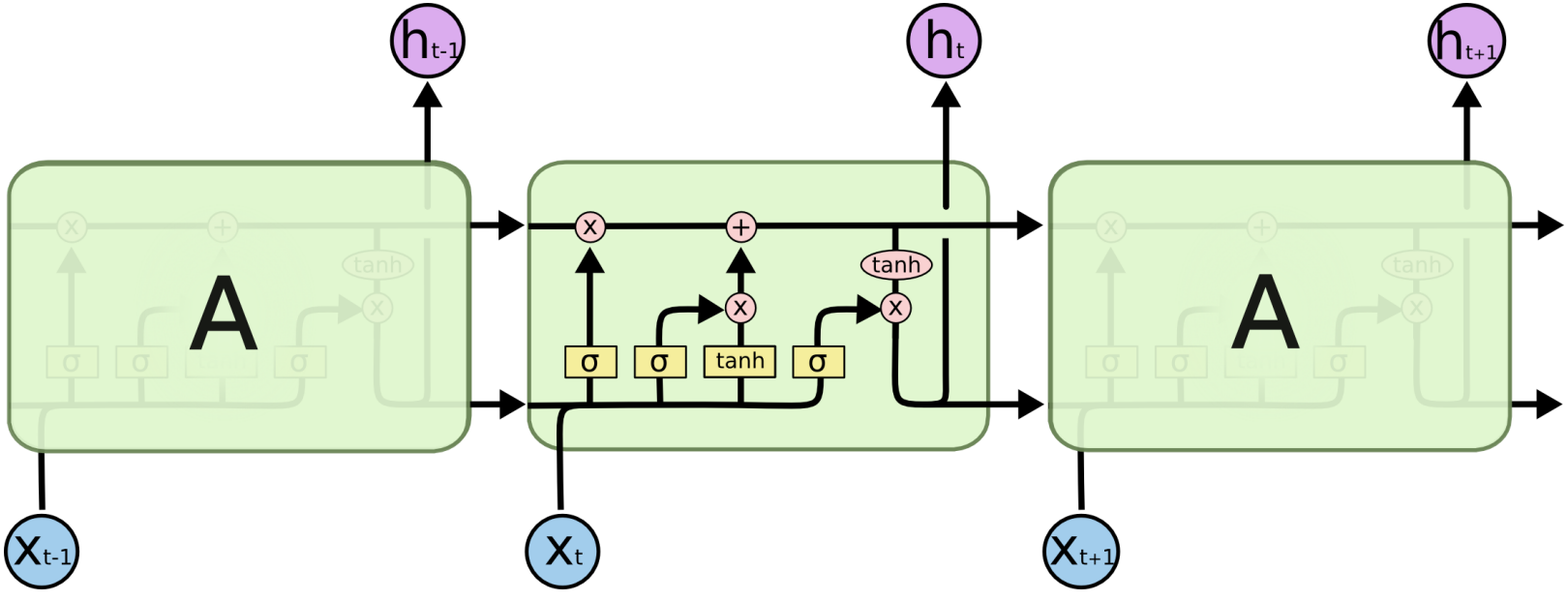
$$h_t = f(h_{t-1}, x_t)$$

Long short-term memory

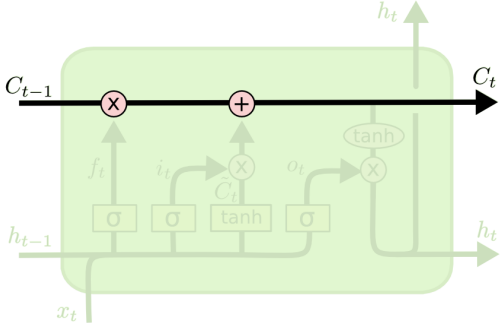


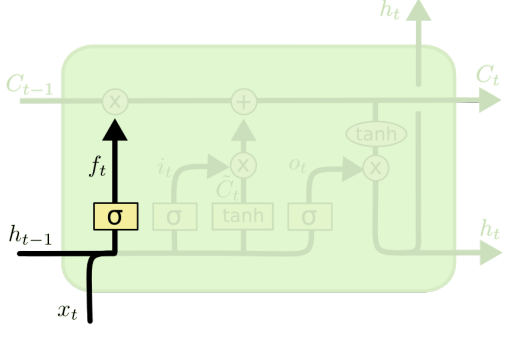


Long short-term memory

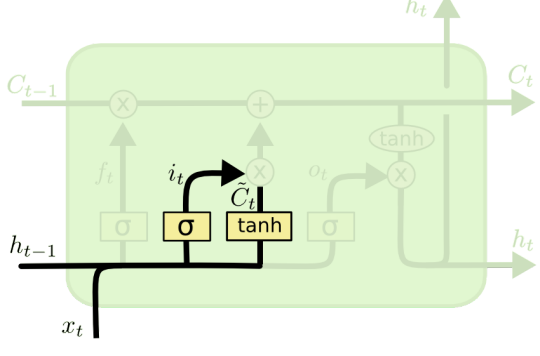


Long short-term memory



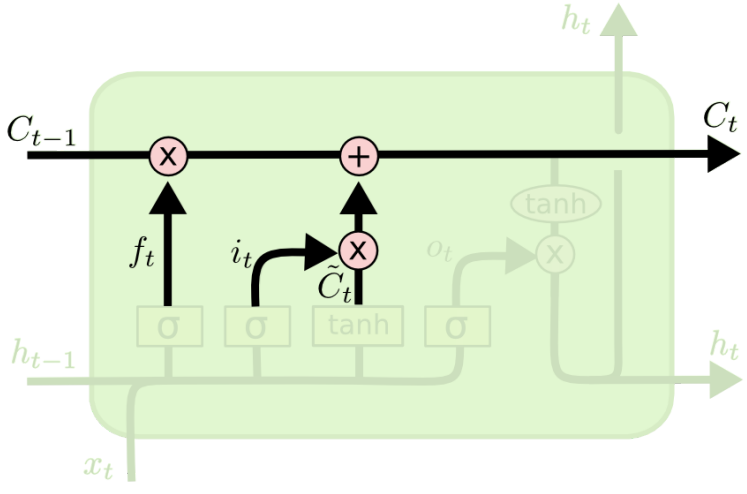


$$f_t = \sigma\left(W_f \cdot [h_{t-1}, x_t] \; + \; b_f\right)$$

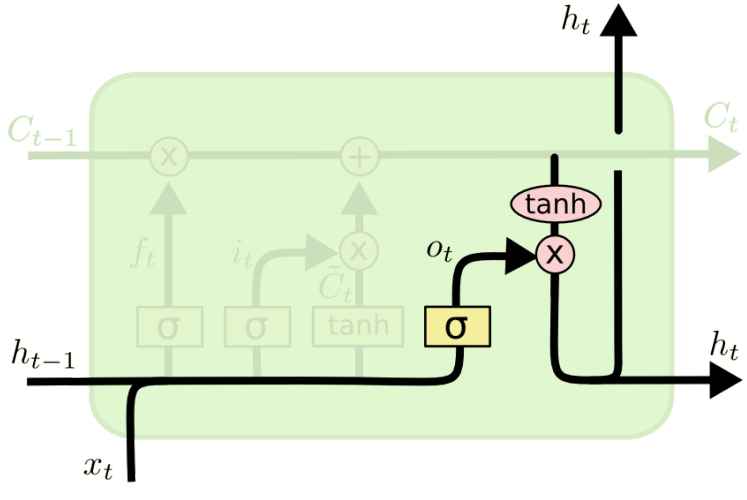


$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] \; + \; b_i\right)$$
$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] \; + \; b_C)$$

Long short-term memory



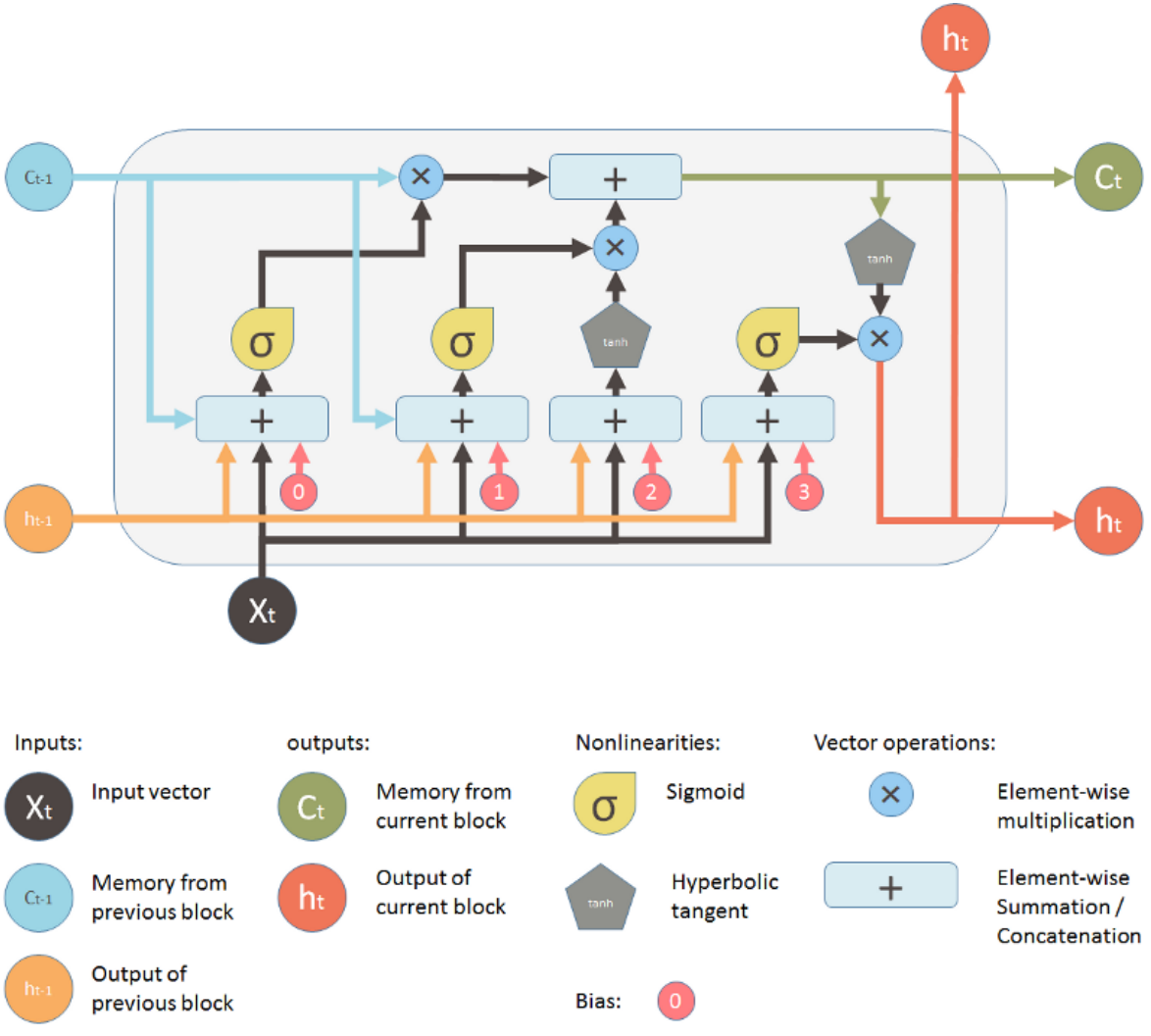
$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t$$



$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right)$$

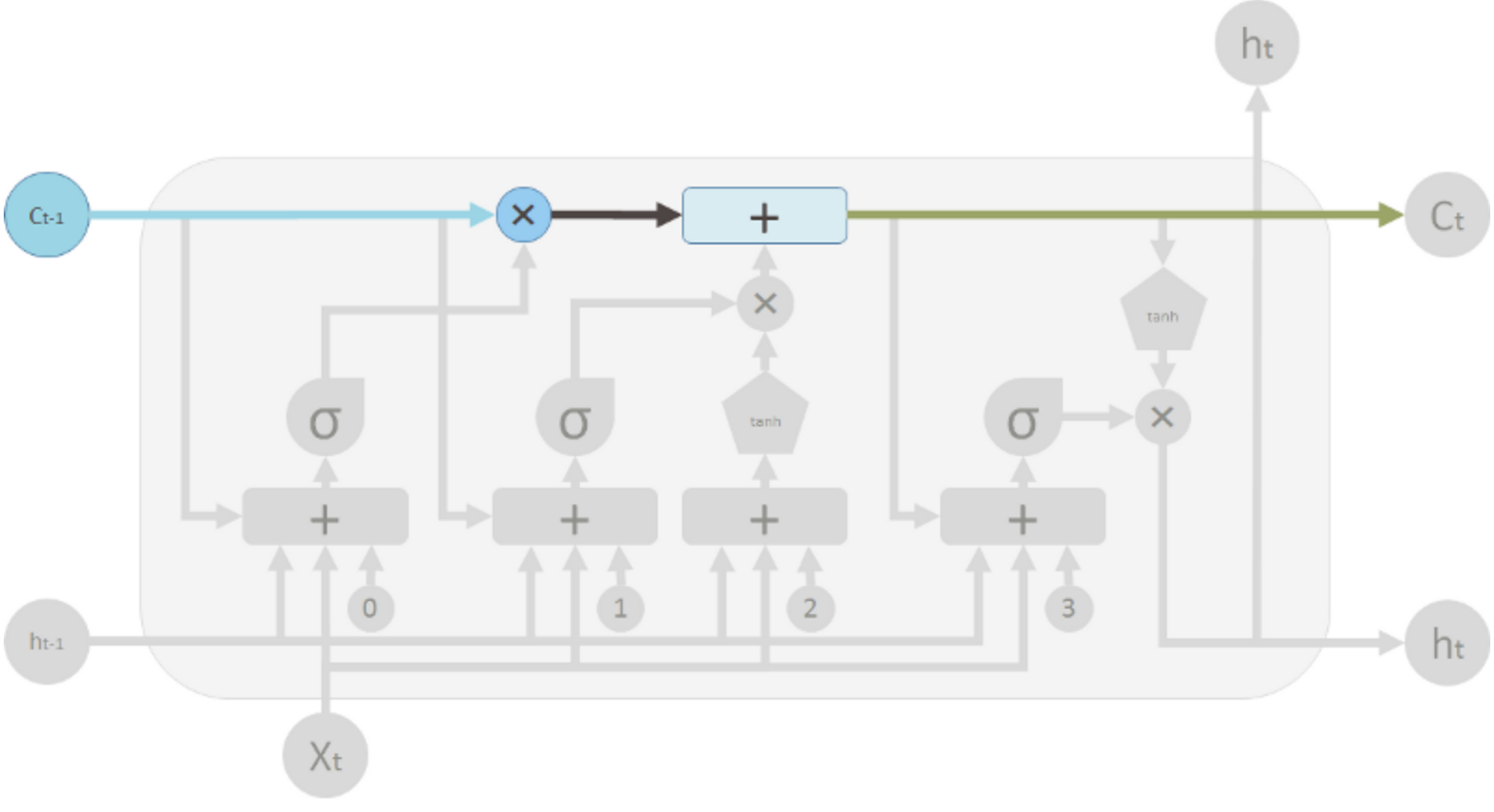
$$h_t = o_t * \tanh\left(C_t\right)$$

Long short-term memory



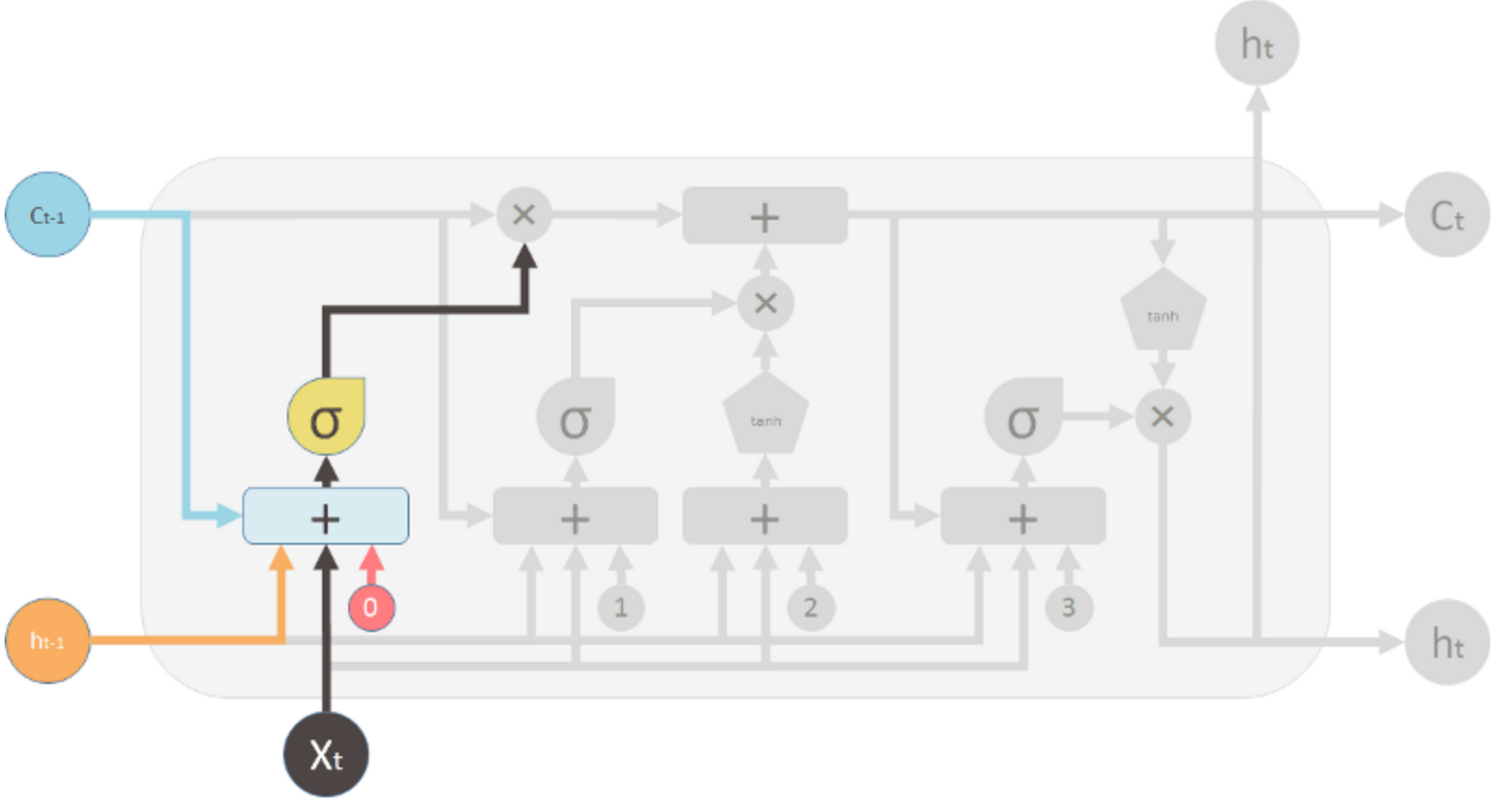
Long short-term memory

Long Term Memory (State)



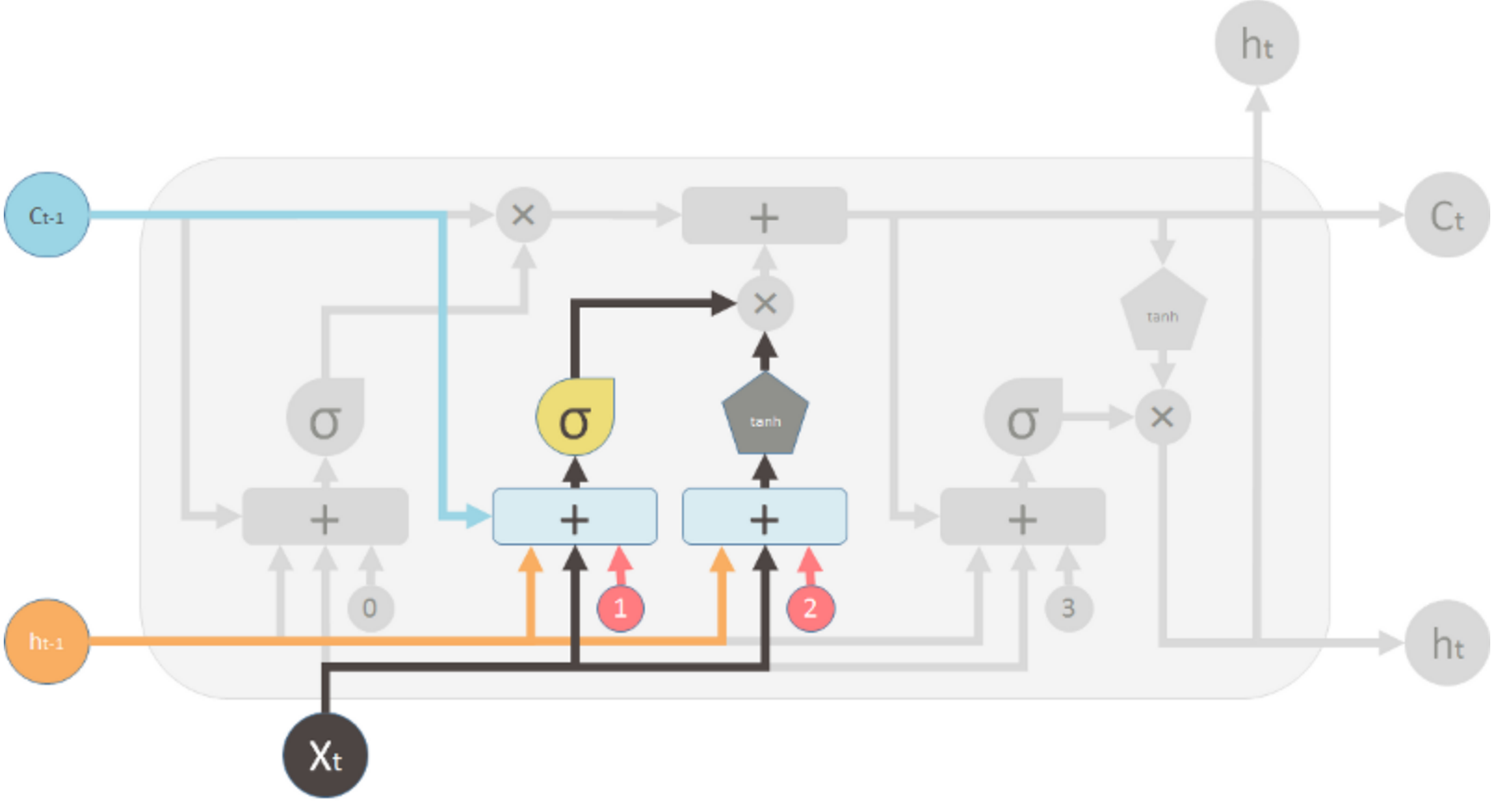
Long short-term memory

Forget Gate



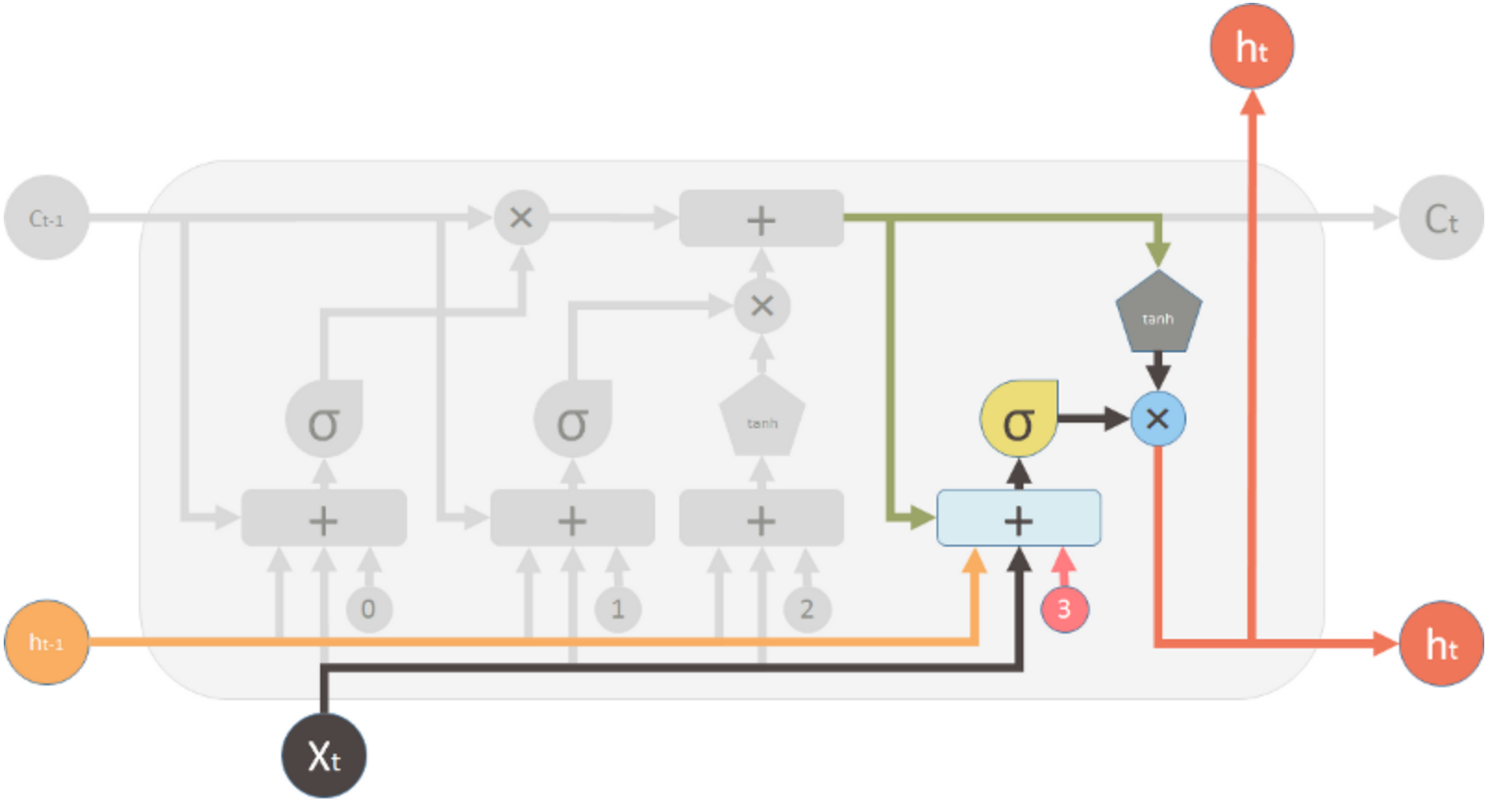
Long short-term memory

New Memory Gate



Long short-term memory

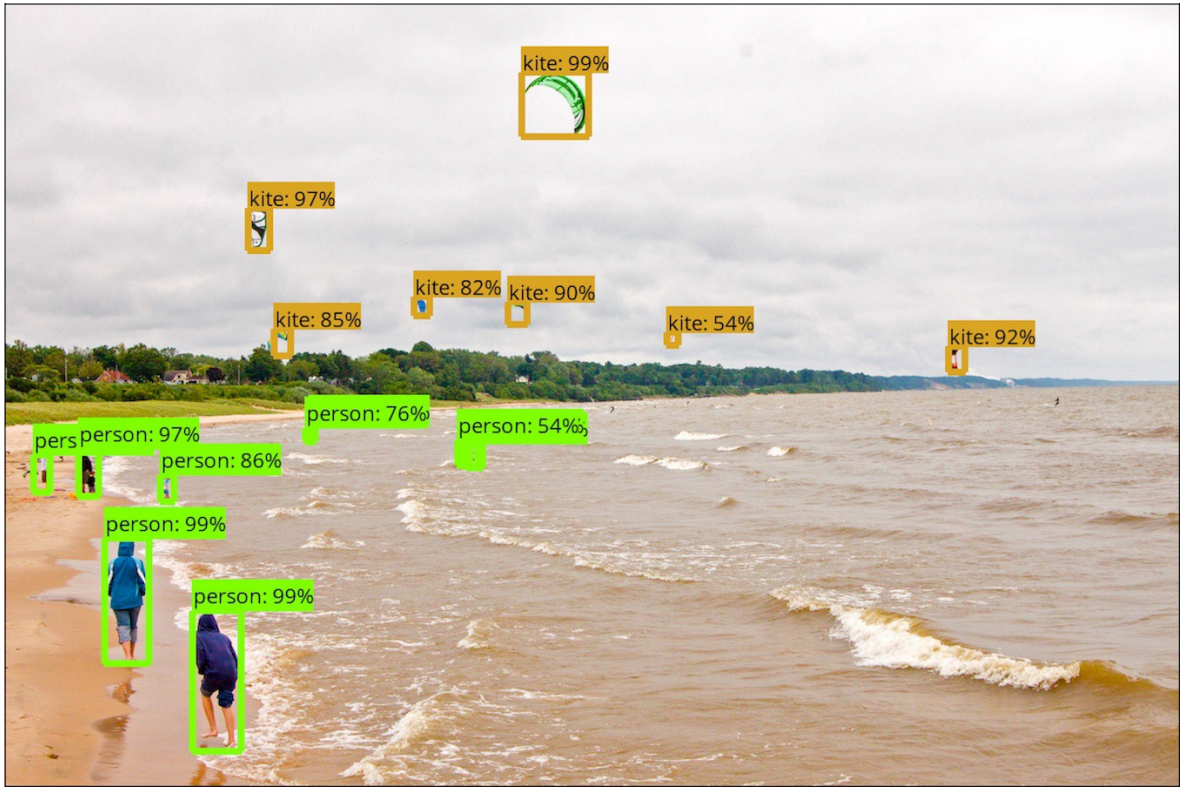
Output Gate

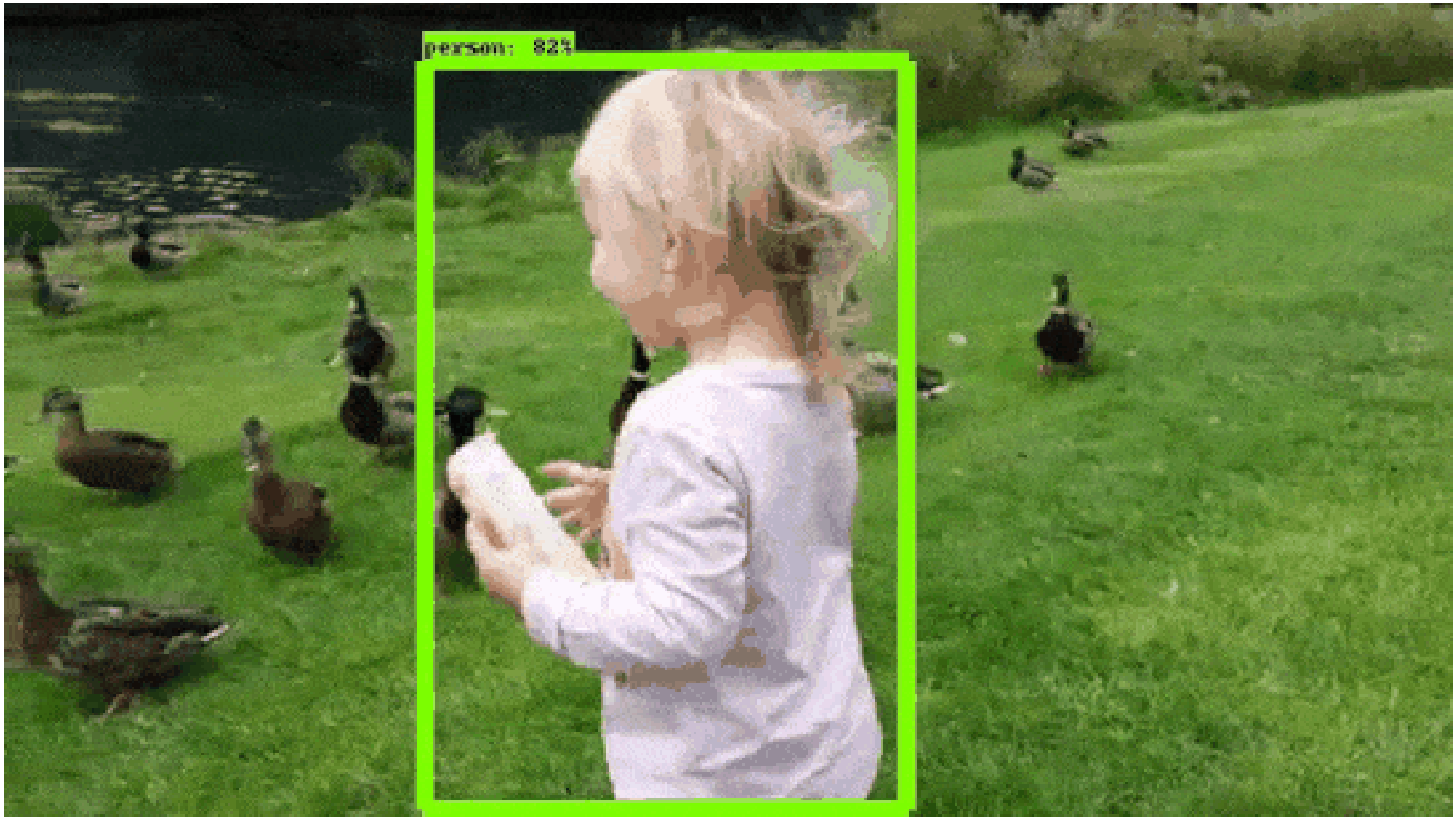


TensorFlow™ is an open source software library for high performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices.

API has been trained on the COCO dataset (Common Objects in Context).

This is a dataset of 300k images of 90 most commonly found objects



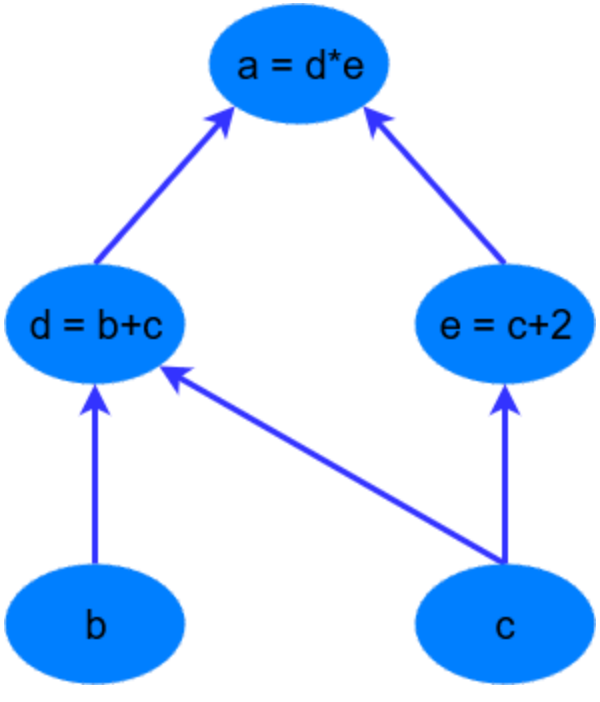

person: 82%

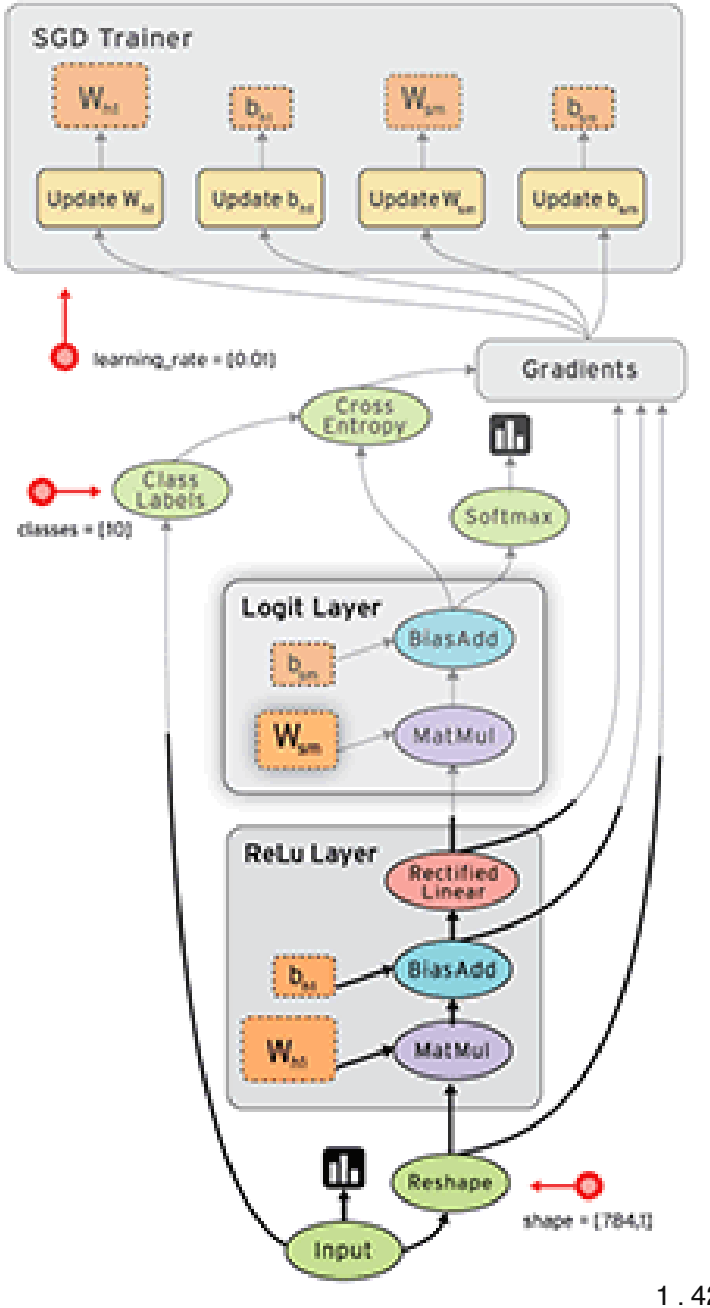
Tensorflow As Language

$$d = b + c$$

$$e = c + 2$$

$$a = d * e$$





```
import tensorflow as tf

# first, create a TensorFlow constant
const = tf.constant(2.0, name="const")

# create TensorFlow variables
b = tf.Variable(2.0, name='b')
c = tf.Variable(1.0, name='c')
```

```
# now create some operations
d = tf.add(b, c, name='d')
e = tf.add(c, const, name='e')
a = tf.multiply(d, e, name='a')
```

```
# setup the variable initialisation
init_op = tf.global_variables_initialize
```

```
# start the session
with tf.Session() as sess:
    # initialise the variables
    sess.run(init_op)
    # compute the output of the graph
    a_out = sess.run(a)
    print("Variable a is {}".format(a_out))
```

Keras

Keras is a high-level neural networks API, written in Python and capable of running on top of TensorFlow, CNTK, or Theano.