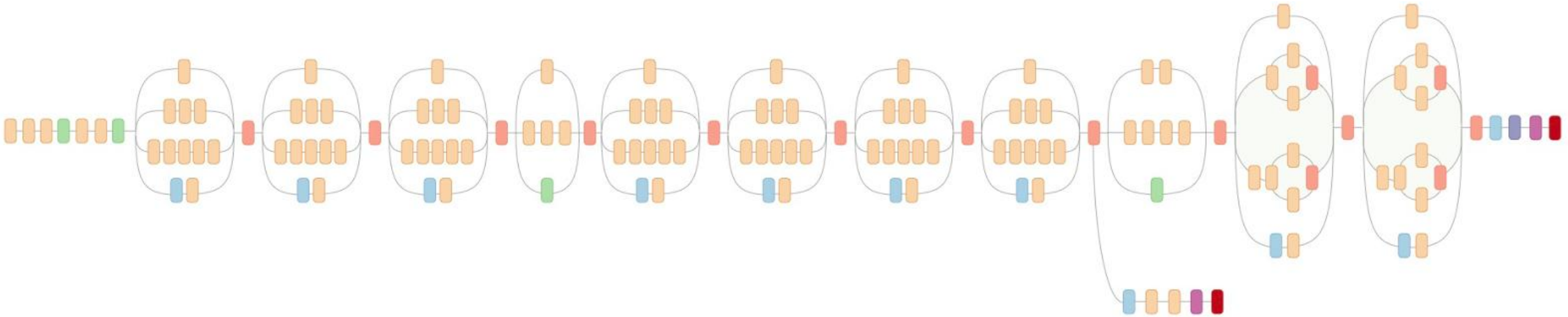


Deep Learning

Presenter: Dr. Denis Krompaß
Siemens Corporate Technology – Machine Intelligence Group
Denis.Krompass@siemens.com

Slides: Dr. Denis Krompaß and Dr. Sigurd Spieckermann

Lecture Overview

I. Introduction to Deep Learning

II. Implementation of a Deep Learning Model
  i. Overview
  ii. Model Architecture Design
    a. Basic Building Blocks
    b. Thinking in Macro Structures
    c. End-to-End Model Design
  iii. Model Training
    a. Loss Functions
    b. Optimization
    c. Regularization

Deep Learning

Introduction

Recommended Courses

Deep Learning:

There will be a lecture in WS 2018/2019

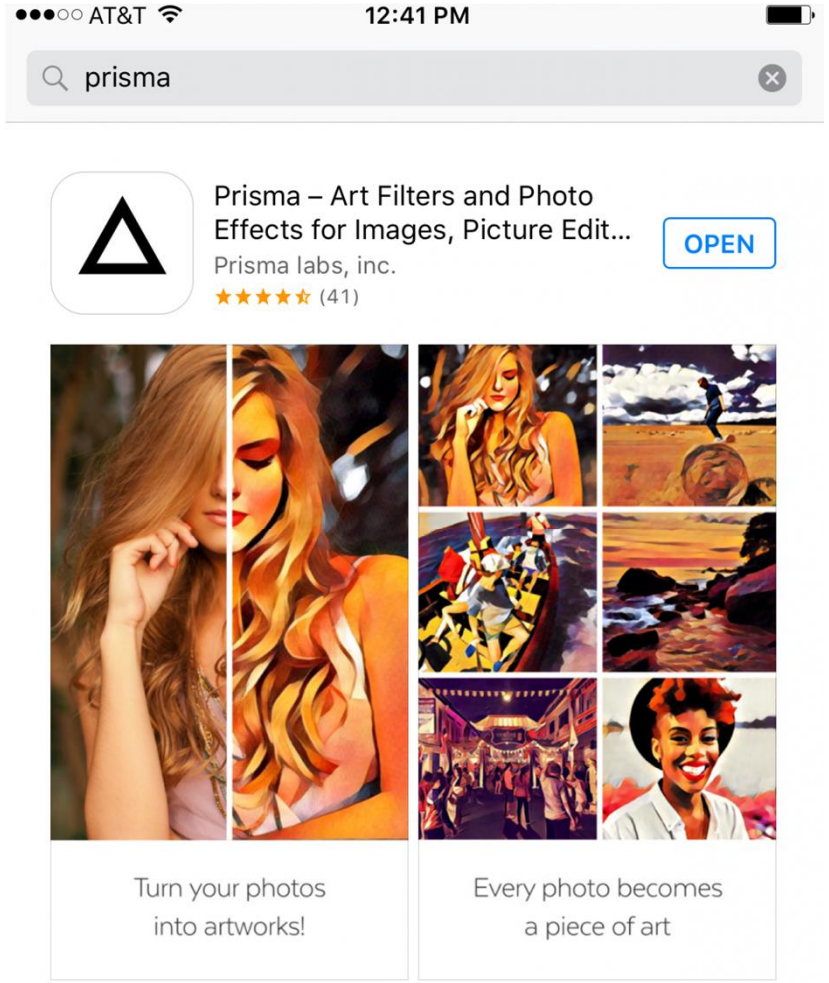
Deep Learning:

https://www.coursera.org/specializations/deep-learning#courses

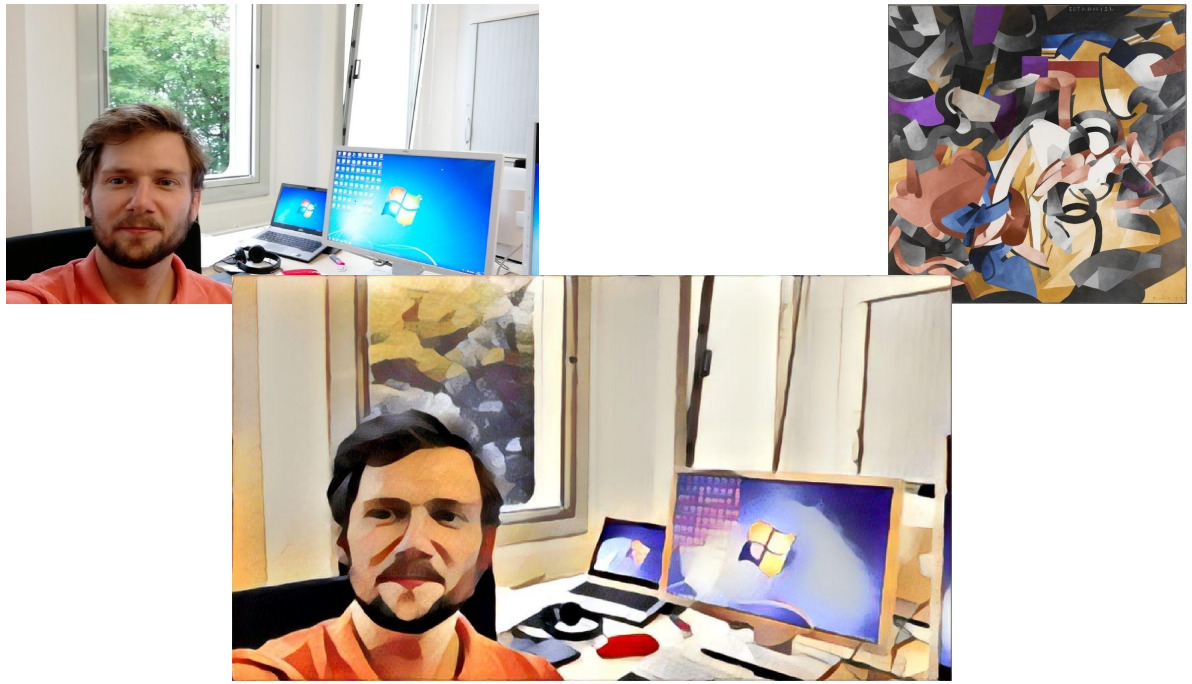
Machine Learning

https://www.coursera.org/learn/machine-learning

Prisma – The App that Made Neural Artisitc Style Transformations Famous



§ 7.5 Million downloads one week after release.



Original work:
Leon A. Gatys, Alexander S. Ecker, Matthias Bethge.
A Neural Algorithm of Artistic Style. arXiv:1508.06576v2, 2015

Also works with videos these days: https://www.youtube.com/watch?v=BcflKNzO31A

Generating High Resolution Images of „Celebrities“

Source (gif):
https://www.theverge.com/2017/10/30/16569402/ai-generate-fake-faces-celebs-nvidia-gan

Full Video:
https://www.youtube.com/watch?v=XOxxPcy5Gr4

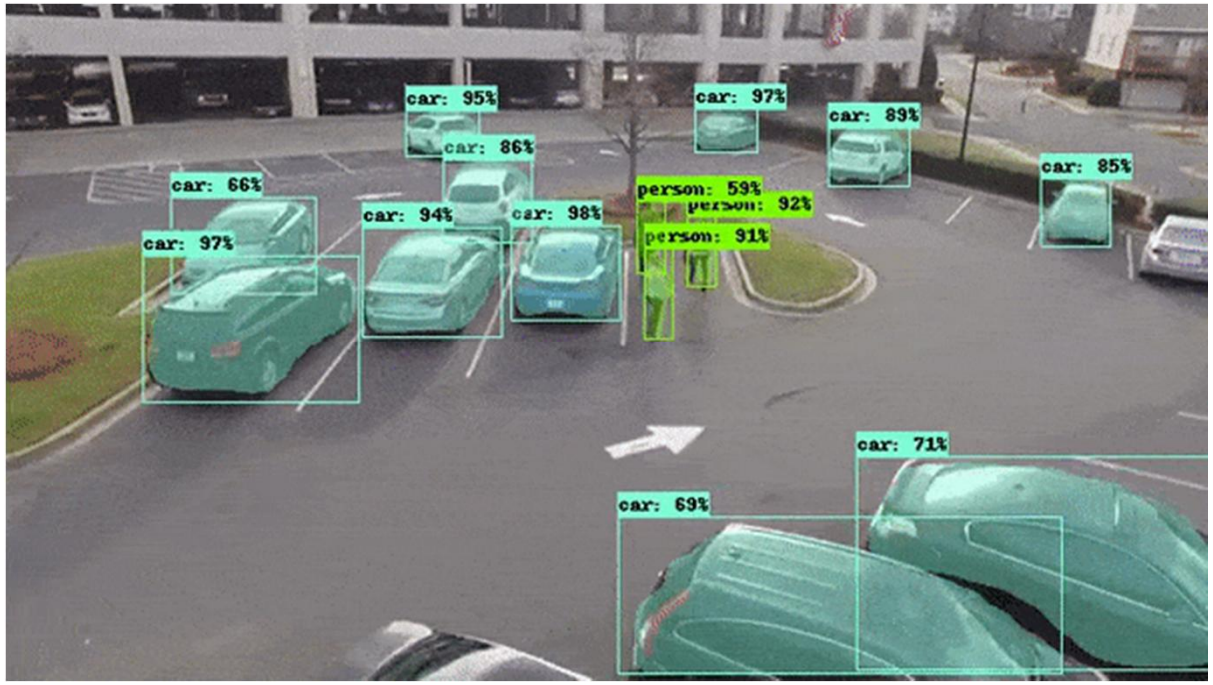
Source and work: Tero Karras, Timo Aila, Samuli Laine, Jaakko Lehtinen. Progressive Growing of GANs for Improved Quality, Stability, and Variation. arXiv:1710.10196v3, 2018

Human Like Perception in Control Tasks

Source: https://techcrunch.com/2016/09/21/scientists-teach-machines-to-hunt-and-kill-humans-in-doom-deathmatch-mode/?guccounter=1

Winning Team:
Alexey Dosovitskiy, Vladlen Koltun. Learning to Act by Predicting the Future. arXiv:1611.01779v2, 2016

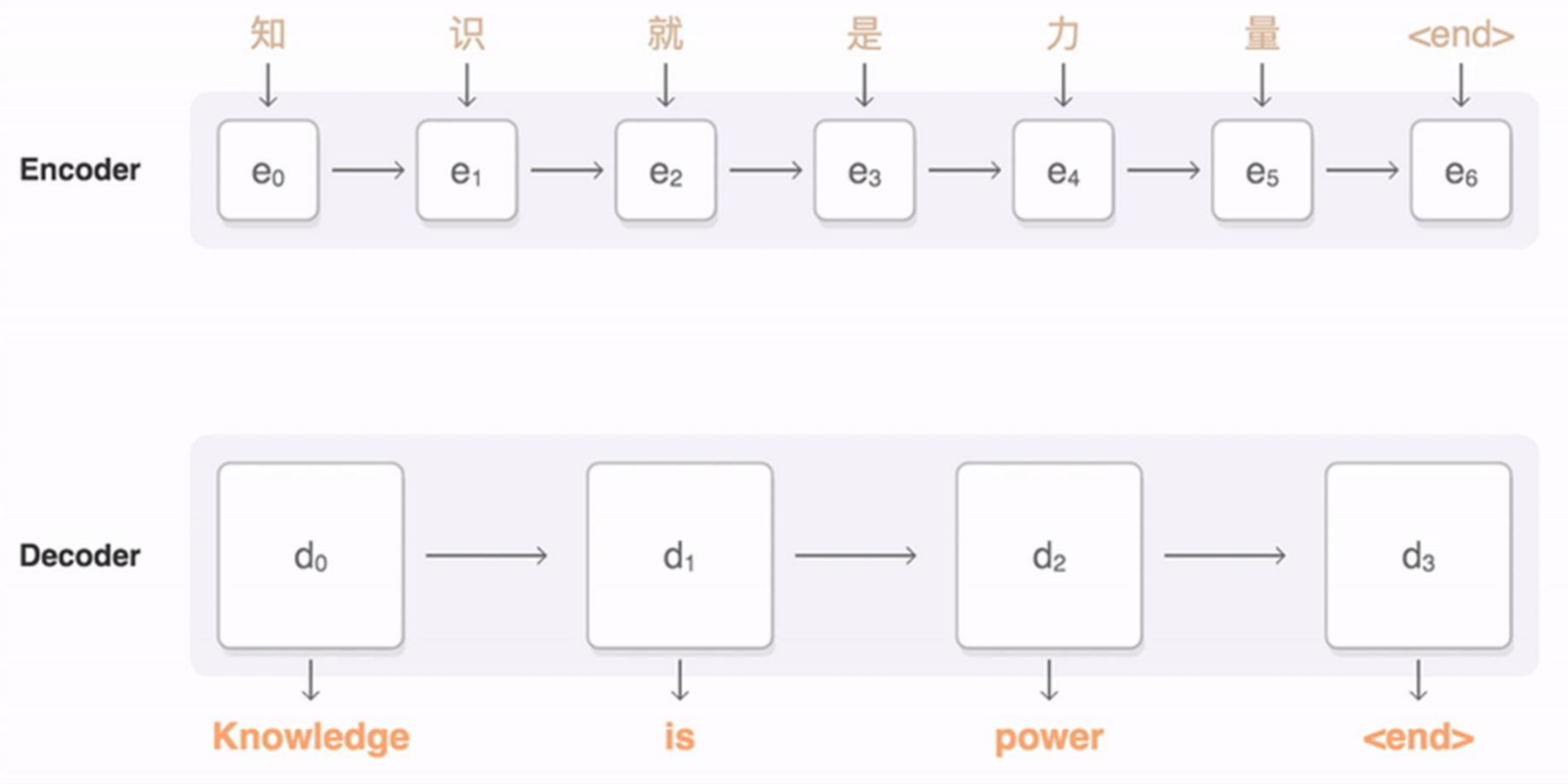
Object Detection / Image Segmentation



Source:
https://towardsdatascience.com/using-tensorflow-object-detection-to-do-pixel-wise-classification-702bf2605182

Nice Video:
https://www.youtube.com/watch?v=OOT3UIXZztE

Translation

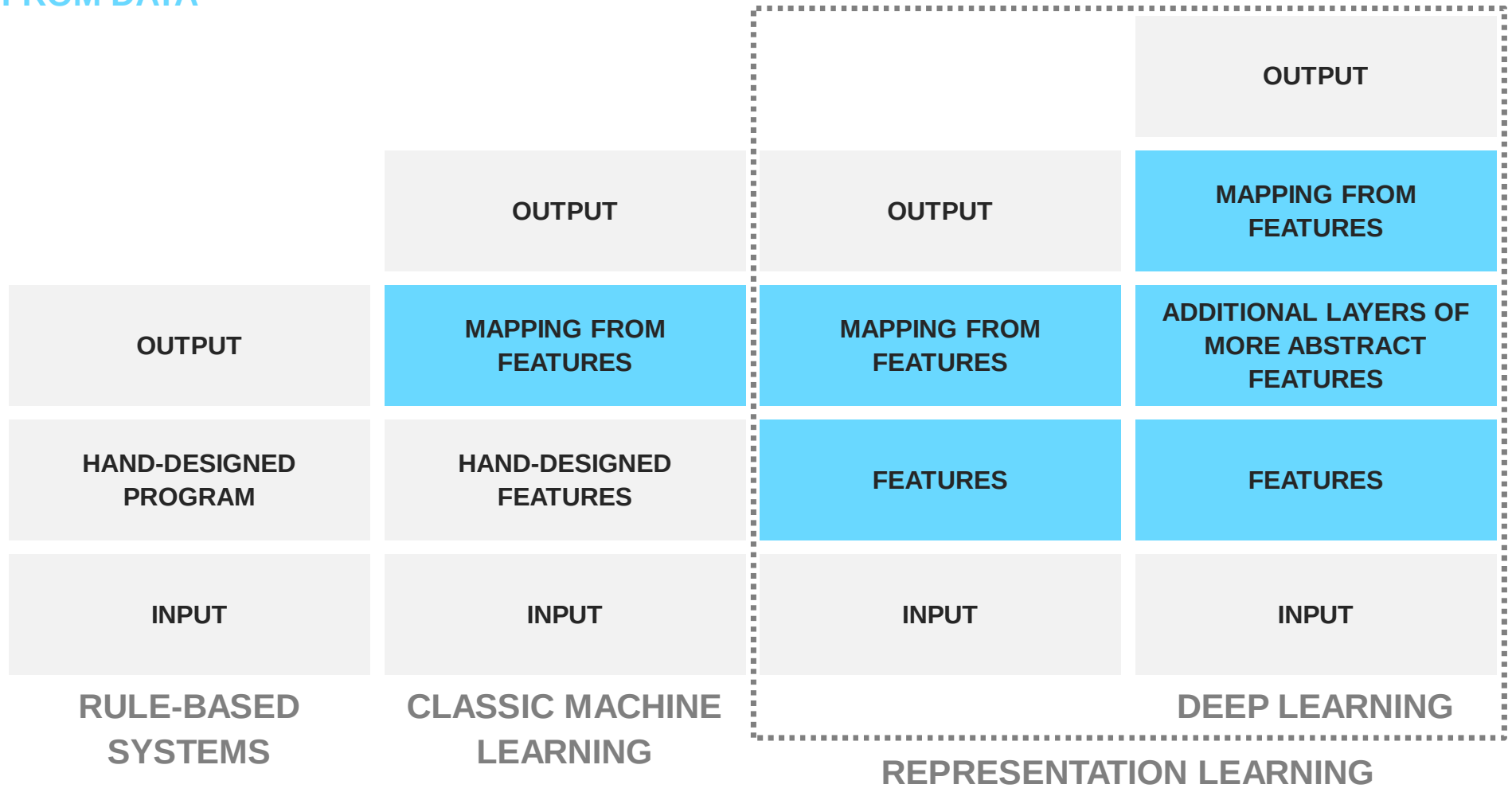


Source: https://github.com/google/seq2seq

And more…

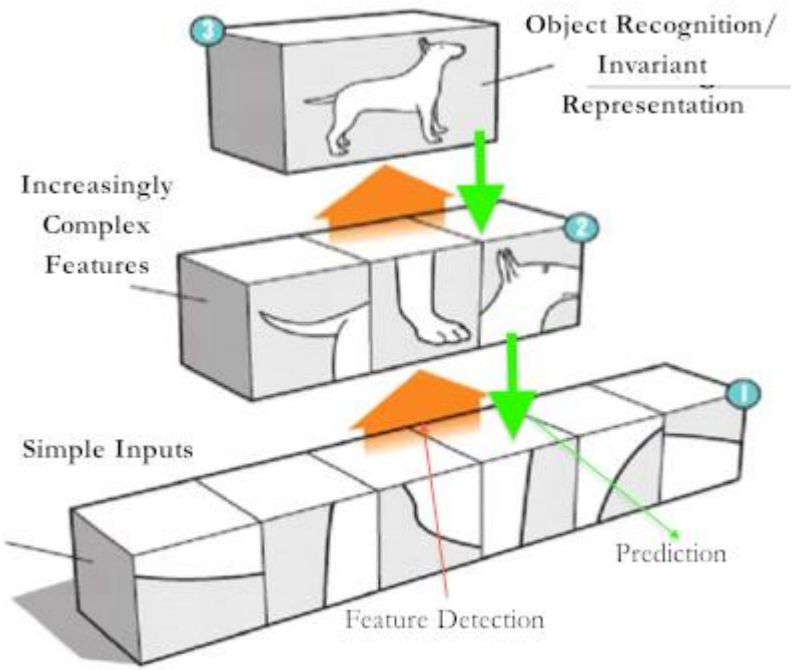
Deep Learning vs. Classic Data Modeling

LEARNED FROM DATA



SOURCE: http://www.deeplearningbook.org/contents/intro.html

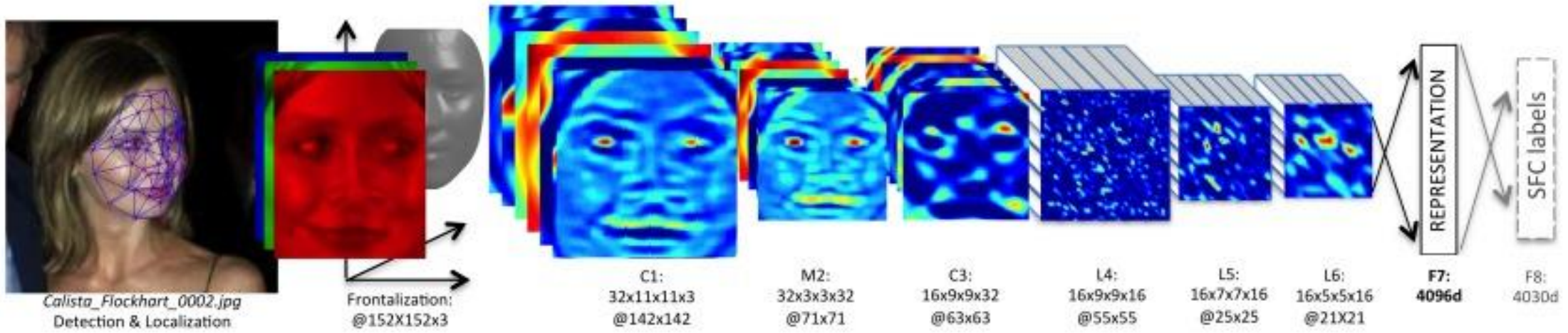
Deep Learning
Hierarchical Feature Extraction



This illustration only shows the idea!
In reality the learned features are abstract and hard to interpret most of the time.

SOURCE: http://www.eidolonspeak.com/Artificial_Intelligence/SOA_P3_Fig4.png

Deep Learning
Hierarchical Feature Extraction



SOURCE:

Taigman, Y., Yang, M., Ranzato, M. A., & Wolf, L. (2014). DeepFace: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1701-1708).

NEURAL NETWORKS HAVE BEEN AROUND FOR DECADES!

(Classic) Neural Networks are an important building block of Deep Learning but there is more to it.

What's new?

* deprecated

OPTIMIZATION & LEARNING

OPTIMIZATION ALGORITHMS

- Adaptive Learning Rates (e.g. ADAM)
- Evolution Strategies
- Synthetic Gradients
- Asynchronous Training
- …

REPARAMETERIZATION

- Batch Normalization
- Weight Normalization
- …

REGULARIZATION

- Dropout
- DropConnect
- DropPath
- …

MODEL ARCHITECTURES

BUILDING BLOCKS

- Spatial/temporal pooling
- Attention mechanism
- Variational Layers
- Dilated convolution
- Variable-length sequence modeling
- Macro modules (e.g. Residual Units)
- Factorized layers
- …

ARCHITECTURES

- Neural computers and memories
- General purpose image feature extractors (VGG, GoogleLeNet)
- End-to-end models
- Generative Adversarial Networks
- …

SOFTWARE

- TensorFlow
  - Keras/Estimator
- PyTorch
- Caffe
- MXNet
- Deeplearning4j
- …

GENERAL

- GPUs
- TPUs
- Hardware accessibility (Cloud)
- Distributed Learning
- Data

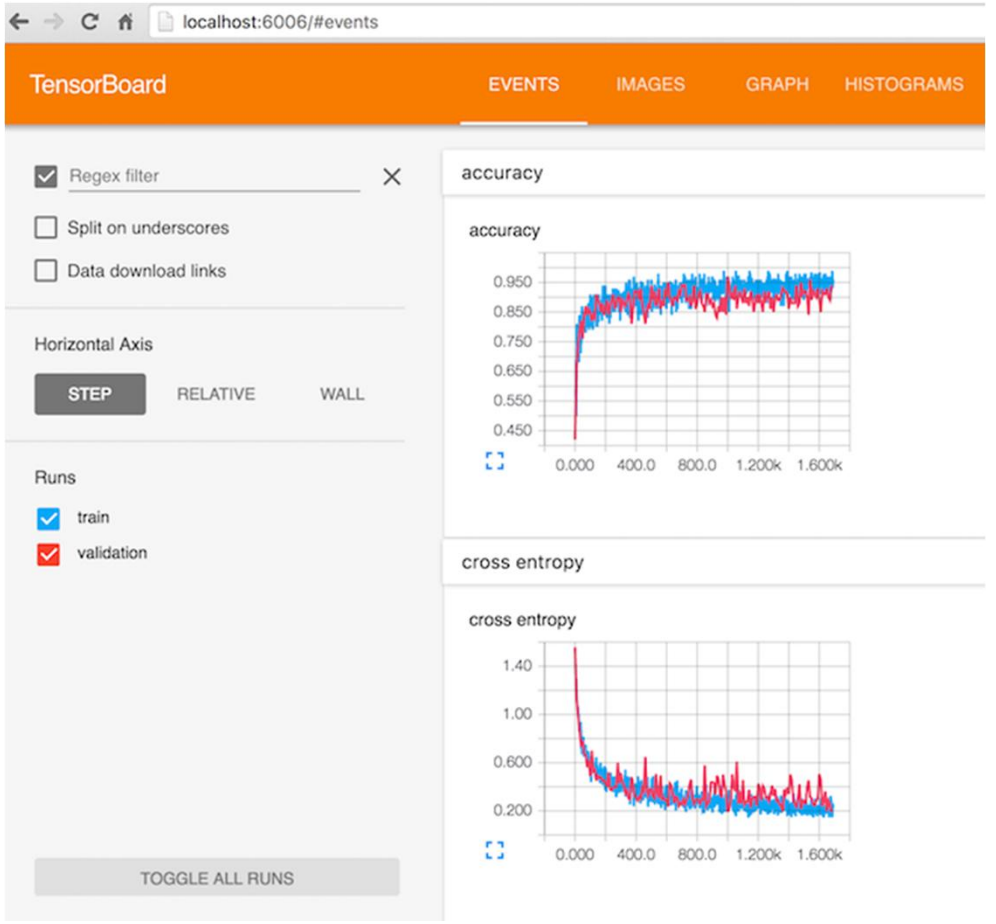
Enabler: Tools

It has never been that easy to build deep learning models!

TensorFlow™

Keras

```python
import numpy as np
import tensorflow as tf


# Define placeholder for input images.
input_images = tf.keras.Input(shape=(24, 24, 3))

# Build the network.
h1 = tf.keras.layers.Conv2D(
    filters=16, kernel_size=[3, 3], strides=[1, 1], padding='same',
    activation=tf.nn.relu)(input_images)
h2 = tf.keras.layers.Conv2D(
    filters=16, kernel_size=[3, 3], strides=[1, 1], padding='same',
    activation=tf.nn.relu)(h1)
pooled = tf.keras.layers.MaxPool2D(
    pool_size=[2, 2], strides=[2, 2], padding='same')(h2)
flattened = tf.keras.layers.Flatten()(pooled)
h3 = tf.keras.layers.Dense(units=100, activation=tf.nn.relu)(flattened)
output = tf.keras.layers.Dense(
    units=10, activation=tf.nn.softmax)(h3)

# Build and compile the model.
model = tf.keras.Model(inputs=input_images, outputs=output)
model.compile(
    tf.train.AdamOptimizer(),
    loss=tf.keras.losses.sparse_categorical_crossentropy)

# Generate some random data and train the model.
random_images = np.random.uniform(
    0, 1, size=(1000, 24, 24, 3)).astype(np.float32)
random_classes = np.random.randint(10, size=(1000,)).astype(np.int32)
model.fit(x=random_images, y=random_classes, batch_size=16, epochs=10)
```



Enabler: Data

Deep Learning requires tons of labeled data if the problem is really complex and you are starting from scratch.

# Labeled examples	Example problems solved in the world.
1 – 10	Not worth a try.
10 – 100	Toy datasets.
100 – 1,000	Toy datasets.
1,000 – 10,000	Hand-written digit recognition.
10,000 – 100,000	Text generation.
100,000 – 1,000,000	Question answering, chat bots.
> 1,000,000	Multi language text translation. Object recognition in images/videos.

Pre-Trained models:

https://www.tensorflow.org/hub/

https://www.tensorflow.org/api_docs/python/tf/keras/applications

Enabler: Computing Power for Everyone

Matrix Products are highly parallelizable

$$h = X \times W, X \in R^{n' m}, W \in R^{m' k}$$

NVIDIA® GPUs

Distributed training enables us to train very large deep learning models on tons of data

Deep Learning Research

Companies

Microsoft

OpenAI

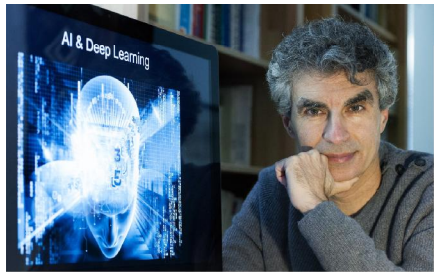
Google

DeepMind

Baidu Research

facebook

People



Yoshua Bengio

Andrew Ng

Geoffrey Hinton

Yann LeCun

Jürgen Schmidhuber

Deep Learning
Implementation

Lecture Overview

I. Introduction to Deep Learning

II. Implementation of a Deep Learning Model
  i. Overview
  ii. Model Architecture Design
    a. Basic Building Blocks
    b. Thinking in Macro Structures
    c. End-to-End Model Design
  iii. Model Training
    a. Loss Functions
    b. Optimization
    c. Regularization

Basic Implementation Overview

1. **Prepare the data**
2. Design the model architecture
3. Design the optimization goals
4. Train
5. Evaluate
6. Deploy

```python
def load_and_preprocess_data():
    """Talk to domain experts, read related work."""
    raise NotImplementedError()


def build_model(data):
    """Read related work, use intuition and experience."""
    raise NotImplementedError()


def build_update_op(model_outputs, data):
    """Read related work, use intuition and experience."""
    raise NotImplementedError()


def train_model(model_inputs, model_targets, update_op, data):
    """Trial and error."""
    raise NotImplementedError()


# Load the data.
data = load_and_preprocess_data()

# Build the model architecture.
model_inputs, model_outputs = build_model(data)

# Build the update operation.
update_op, model_targets = build_update_op(model_outputs, data)

# Train the model on the data.
train_model(model_inputs, model_targets, update_op, data)
```

Basic Implementation Overview

1. **Prepare the data**
2. Design the model architecture
3. Design the optimization goals
4. Train
5. Evaluate
6. Deploy

```python
def load_and_preprocess_data():
    """Talk to domain experts, read related work."""
    raise NotImplementedError()


def build_model(data):
    """Read related work, use intuition and experience."""
    raise NotImplementedError()


def build_update_op(model_outputs, data):
    """Read related work, use intuition and experience."""
    raise NotImplementedError()


def train_model(model_inputs, model_targets, update_op, data):
    """Trial and error."""
    raise NotImplementedError()


# Load the data.
data = load_and_preprocess_data()

# Build the model architecture.
model_inputs, model_outputs = build_model(data)

# Build the update operation.
update_op, model_targets = build_update_op(model_outputs, data)

# Train the model on the data.
train_model(model_inputs, model_targets, update_op, data)
```

Deep Learning Model Architecture

Basic Building Blocks

- The fully connected layer – Using brute force.
- Convolutional neural network layers – Exploiting neighborhood relations.
- Recurrent neural network layers – Exploiting sequential relations.

Thinking in Macro Structures

- Mixing things up – Generating purpose modules.
- LSTMs and Gating – Simple memory management.
- Attention – Dynamic context driven information selection.
- Residual Units – Building ultra deep structures.

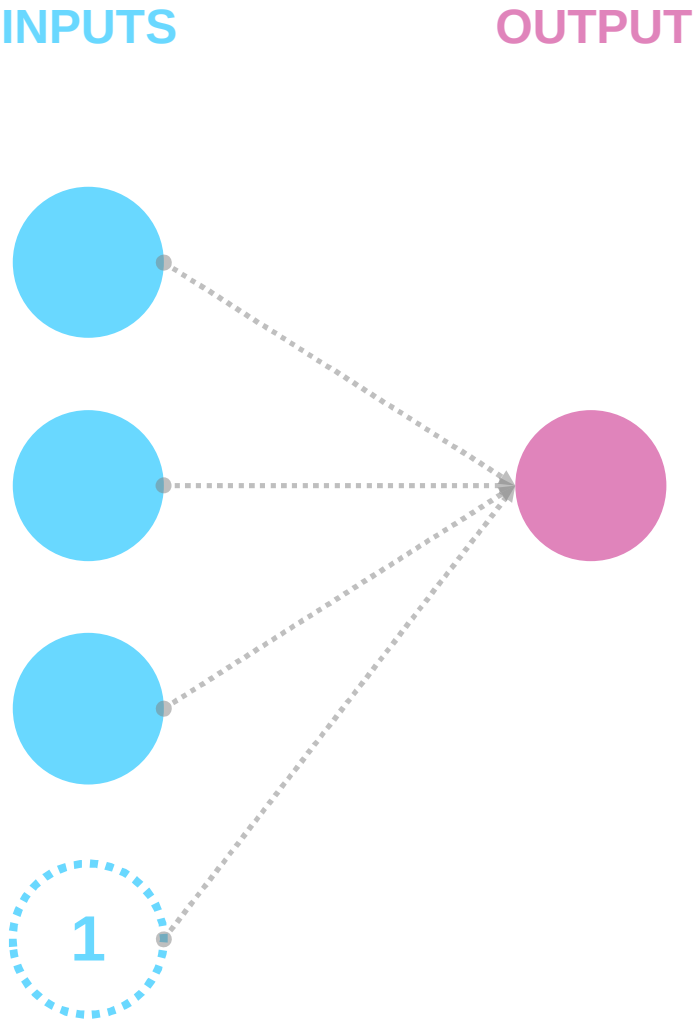
End-to-End model design

- Example for design choices.
- Real examples.

Deep Learning

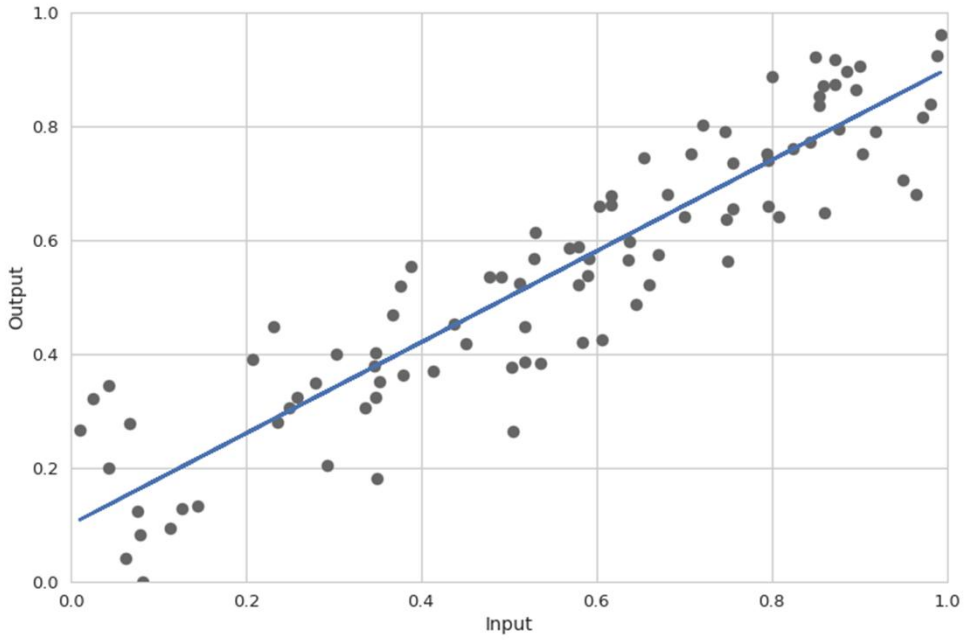
Basic Building Blocks

Neural Network Basics
Linear Regression

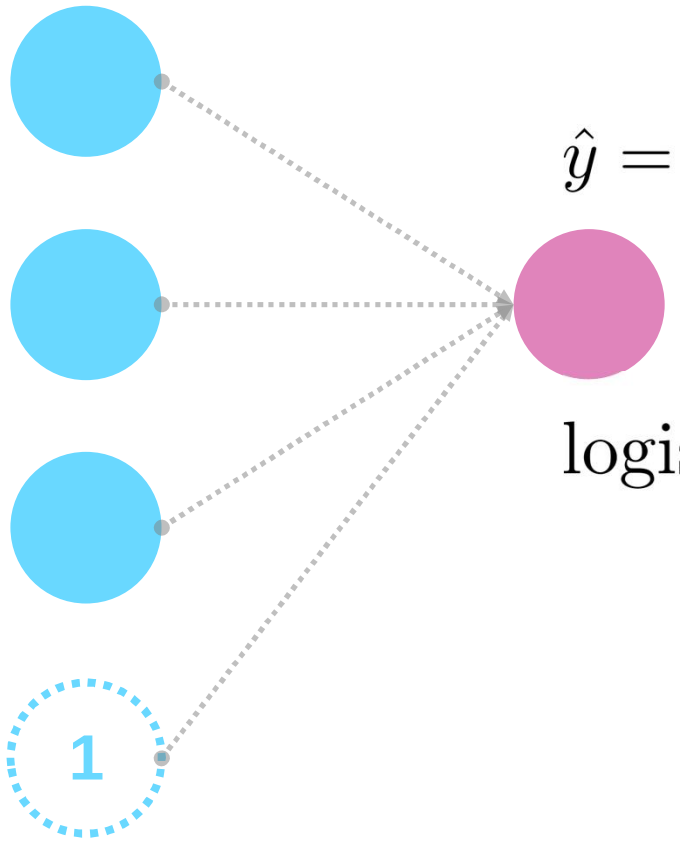


$$\hat{y} = w^T x + b$$

```
In [43]: X = tf.placeholder(tf.float32, [None, 2])
         W = tf.Variable(np.random.rand(2, 1).astype(np.float32))
         b = tf.Variable(np.zeros((1, 1)).astype(np.float32))
         output = tf.matmul(X, W) + b
```



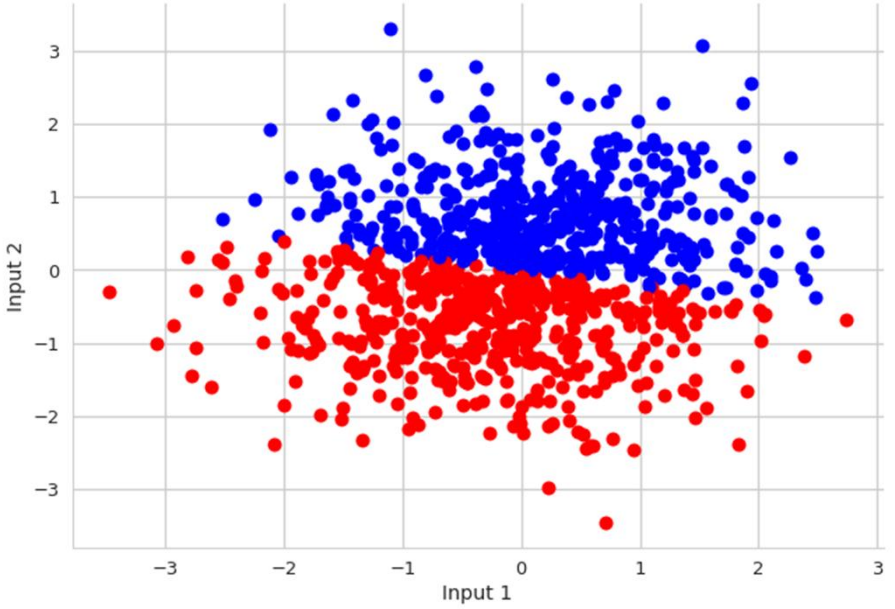
Neural Network Basics
Logistic Regression



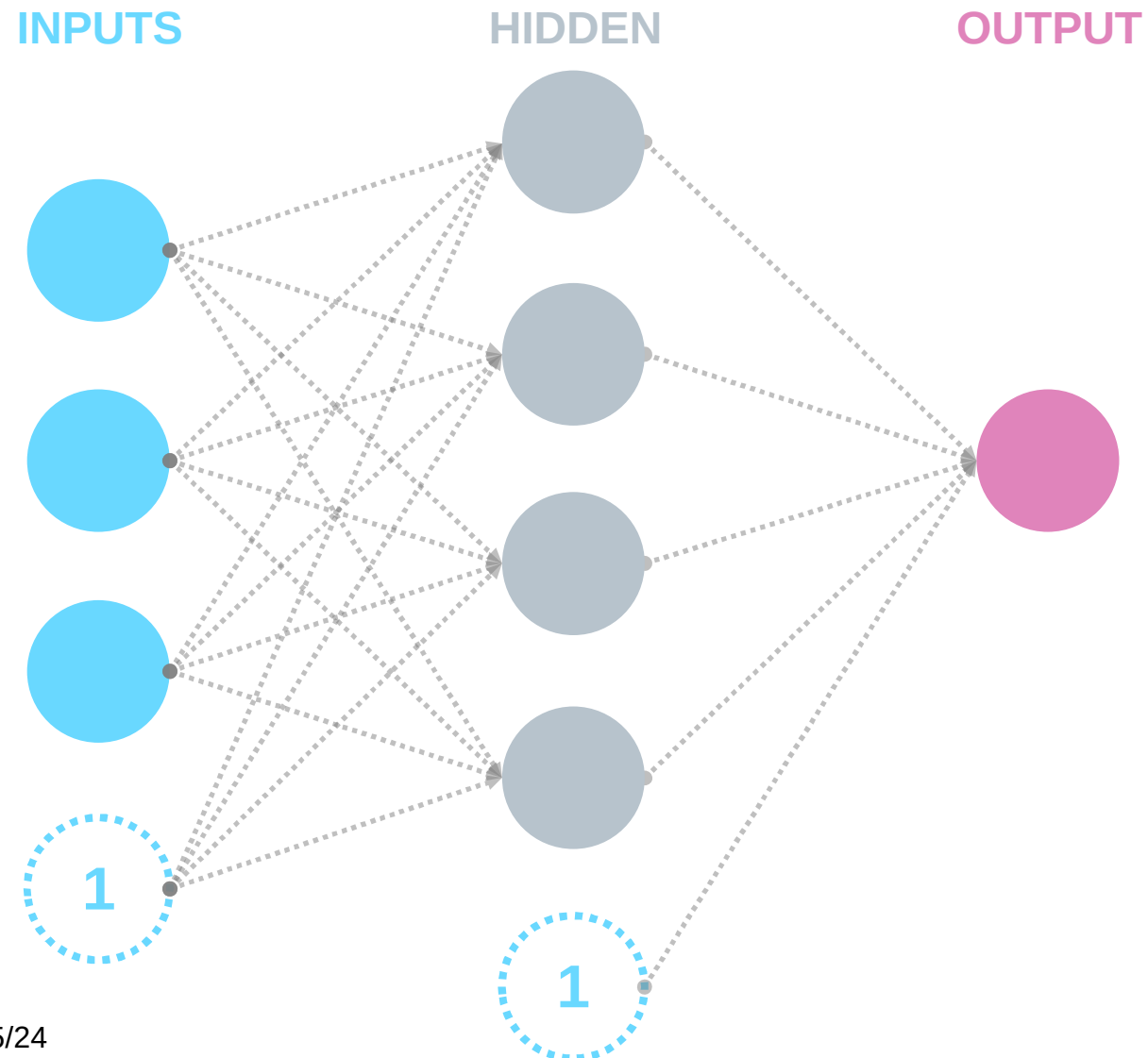
$$\hat{y} = \text{logistic}(w^T x + b)$$

$$\text{logistic}(z) = \frac{1}{1 + e^{-z}}$$

```
In [3]: X = tf.placeholder(tf.float32, [None, 2])
        W = tf.Variable(np.random.rand(2, 1).astype(np.float32))
        b = tf.Variable(np.zeros((1, 1)).astype(np.float32))
        h = tf.nn.sigmoid(tf.matmul(X, W) + b)
```



Neural Network Basics
Multi-Layer Perceptron

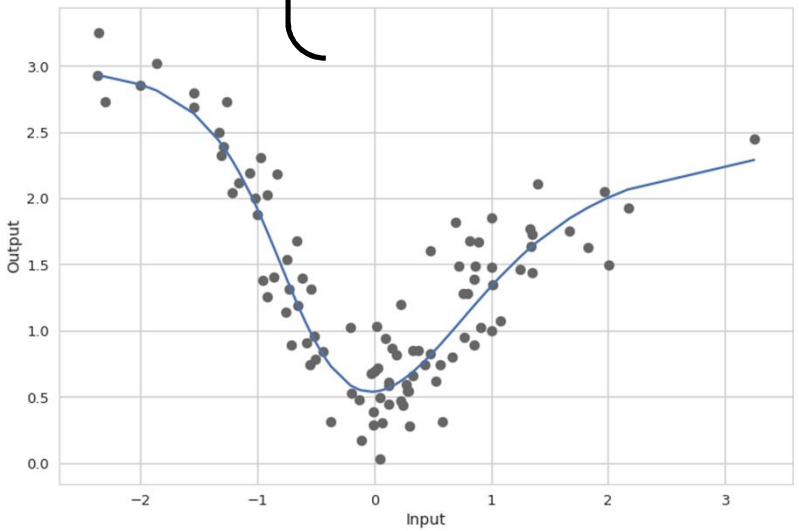


$$h^{(1)} = \phi\left(W^{(1)}x + b^{(1)}\right)$$ ⬅ Hidden Layer

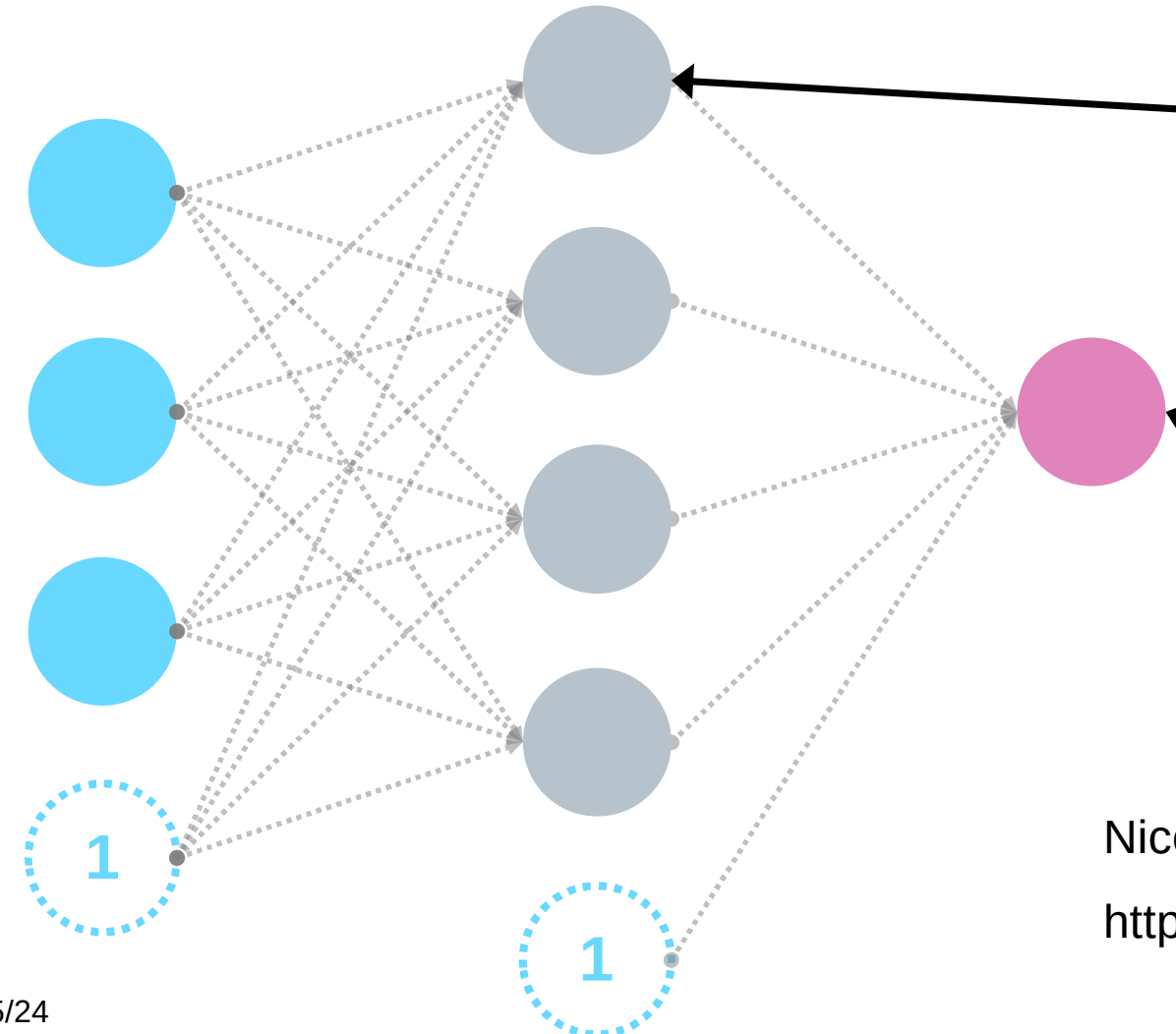
$$\hat{y} = W^{(2)}h^{(1)} + b^{(2)}$$ ⬅ Output Layer

With activation function:

$$\phi(z) = \begin{cases} \tanh(z) \\ \text{relu}(z) \\ \dots \end{cases}$$



Neural Network Basics
Activation Functions



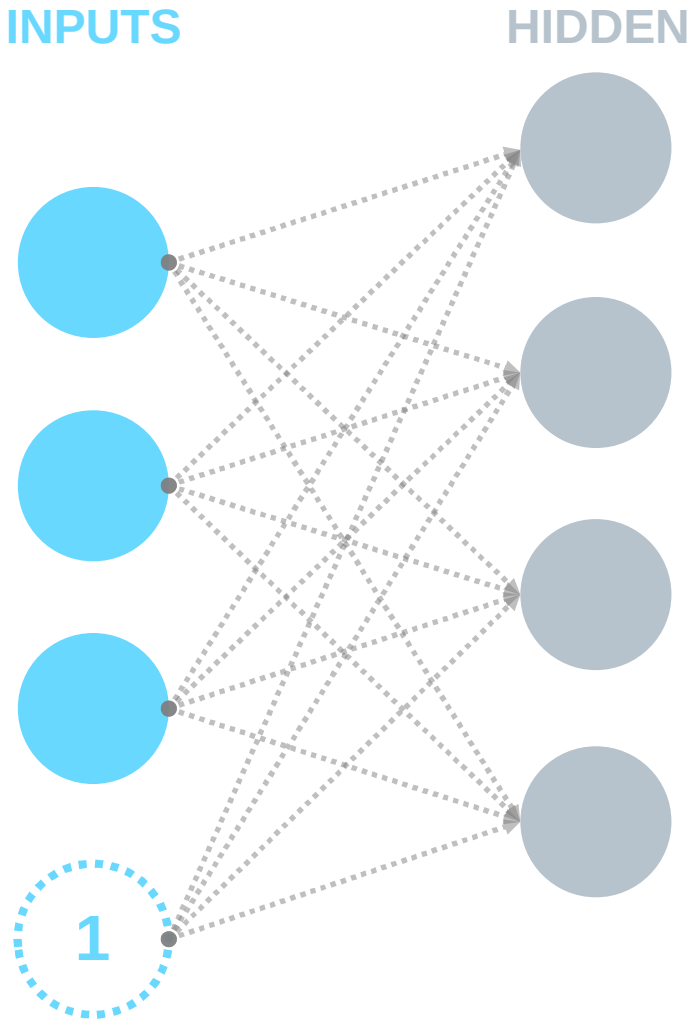
Activation Function
identity(h)
tanh(h)
relu(h)
…

Activation Function	Task
identity(h)	Regression
logistic(h)	Binary Classification
softmax(h)	Multi-Class Classification

Nice overview on activation functions:

https://en.wikipedia.org/wiki/Activation_function

Basic Building Blocks
The Fully Connected Layer



Passing one example:

$$x^T = [1\ 3\ 7]$$

$$h = \phi(Wx + b)$$

$$x \in \mathbb{R}^{3\times 1}, W \in \mathbb{R}^{4\times 3}, b \in \mathbb{R}^{4\times 1}$$

Passing n examples:

$$X = \begin{bmatrix} 1 & 3 & 7 \\ \vdots & \ddots & \vdots \\ 3 & 8 & 3 \end{bmatrix}$$

$$H = \phi(XW^T + b^T)$$

$$\mathrm{X} \in \mathbb{R}^{n\times 3}, W \in \mathbb{R}^{4\times 3}, b \in \mathbb{R}^{4\times 1}$$

Basic Building Blocks
The Fully Connected Layer

Passing n examples:

$$X = \begin{bmatrix} 1 & 3 & 7 \\ \vdots & \ddots & \vdots \\ 3 & 8 & 3 \end{bmatrix}$$

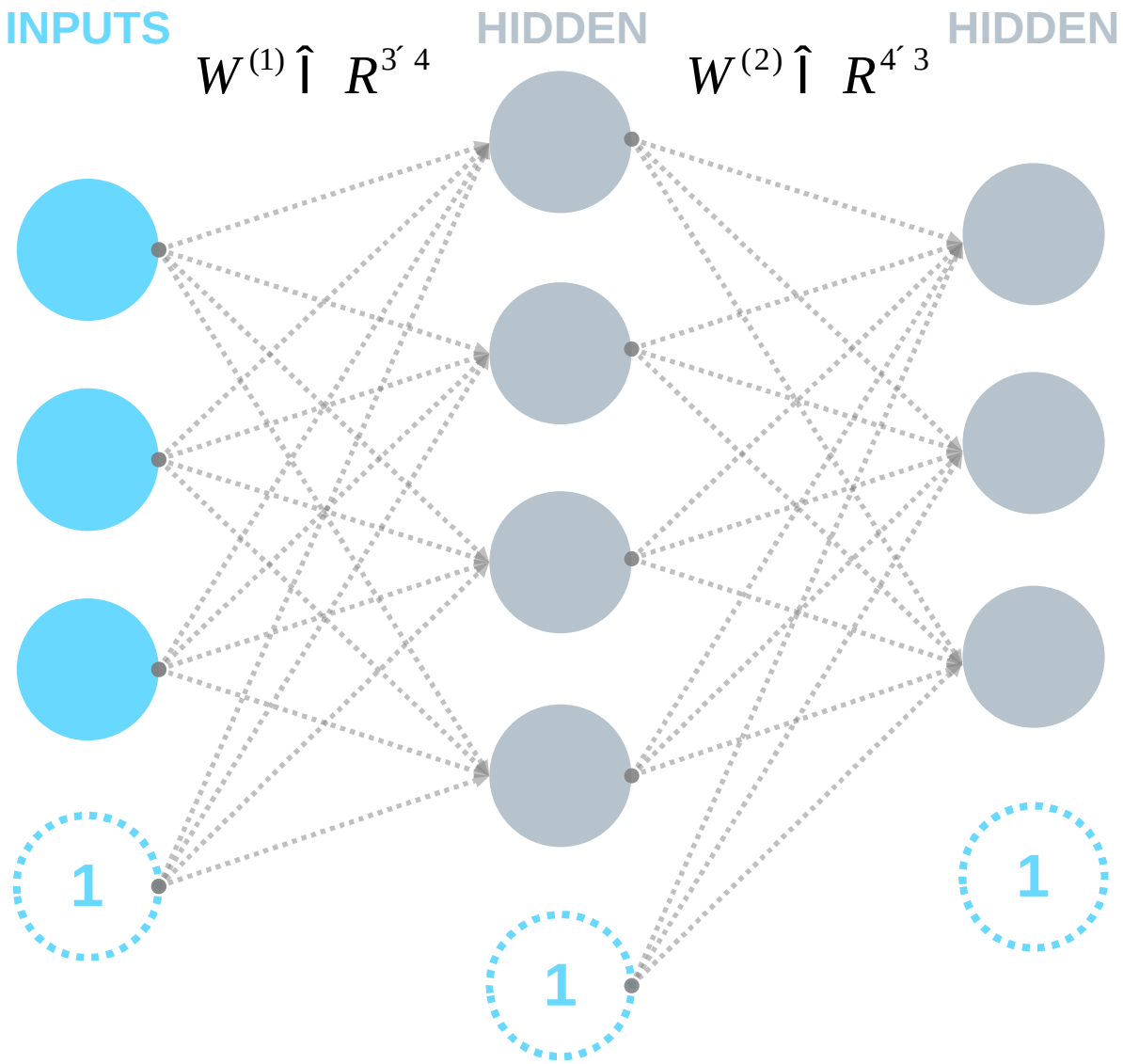
$$H = \phi(XW^T + b^T)$$

$$X \in \mathbb{R}^{n \times 3}, W \in \mathbb{R}^{4 \times 3}, b \in \mathbb{R}^{4 \times 1}$$

```python
import numpy as np
import tensorflow as tf


# Define placeholder for the input data that expects a matrix with an arbitrary
# amount of rows (= number of examples) and 3 columns (=features)
X = tf.placeholder(tf.float32, [None, 3])

# Define the variables of the dense layer.
W = tf.get_variable(
    'W', shape=(3, 4), dtype=tf.float32,
    initializer=tf.keras.initializers.glorot_uniform(seed=None))
b = tf.get_variable(
    'b', shape=(4,), dtype=tf.float32, initializer=tf.constant_initializer(0))
h = tf.nn.tanh(tf.matmul(X, W) + b)

# Feed some random data through the layer.
random_data = np.random.uniform(0, 1, size=(10, 3)).astype(np.float32)
with tf.Session() as session:
    # We need to initialize the layer parameters first.
    session.run(tf.global_variables_initializer())
    # Feed the network with the random data.
    layer_output = session.run(h, feed_dict={X: random_data})
    print layer_output.shape  # gives: (10, 4)
```

Basic Building Blocks

The Fully Connected Layer – Stacking



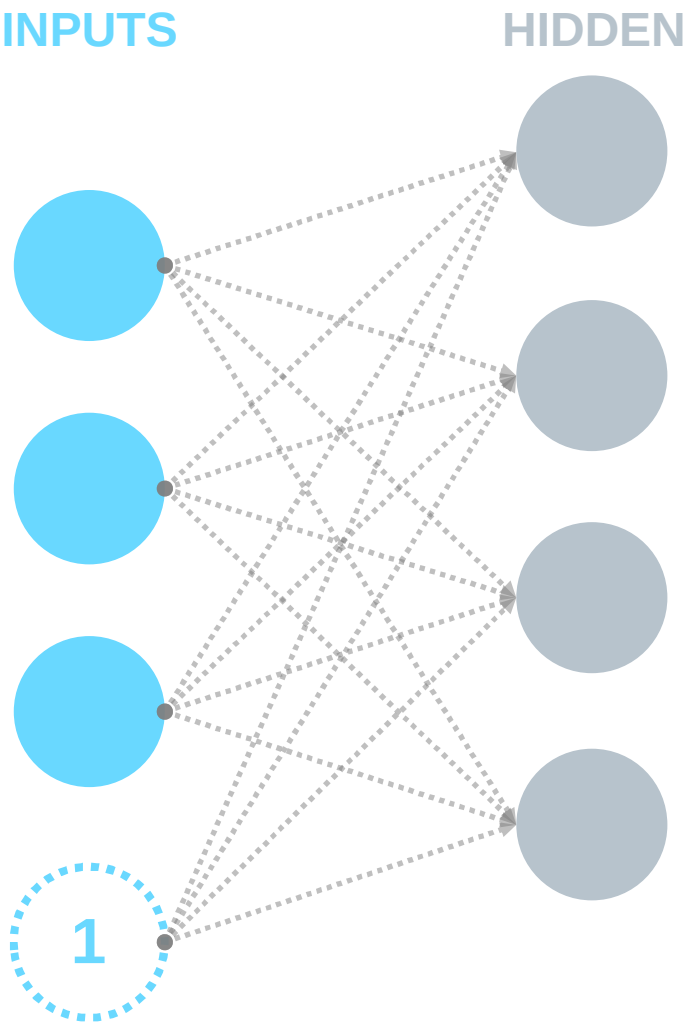
$$h^{(1)} = f(W^{(1)}x + b^{(1)})$$

$$h^{(2)} = f(W^{(2)}h^{(1)} + b^{(2)})$$

...

$$h^{(l)} = f(W^{(l)}h^{(l-1)} + b^{(l)})$$

Basic Building Blocks
The Fully Connected Layer – Using Brute Force



Brute force layer:

- Exploits no assumptions about the inputs.
  - ØNo weight sharing.
- Simply combines all inputs with each other.
  - ØExpensive! Often responsible for the largest amount of parameters in a deep learning model.
- Use with care since it can quickly over-parameterize the model
  - ØCan lead to degenerated solutions.

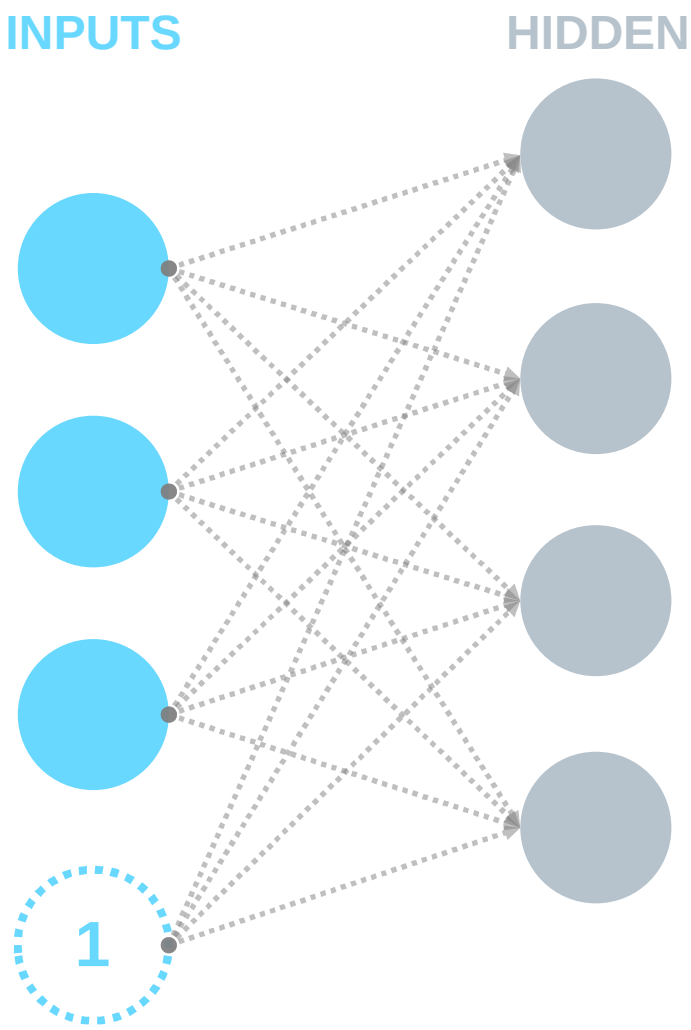
Examples:

RGB image of shape
100 x 100 x 3

3,000,000 free parameters
for a fully connected layer
with 100 hidden units!

Two consecutive fully connected layer with 1000 hidden neurons each: 1,000,000 free parameters!

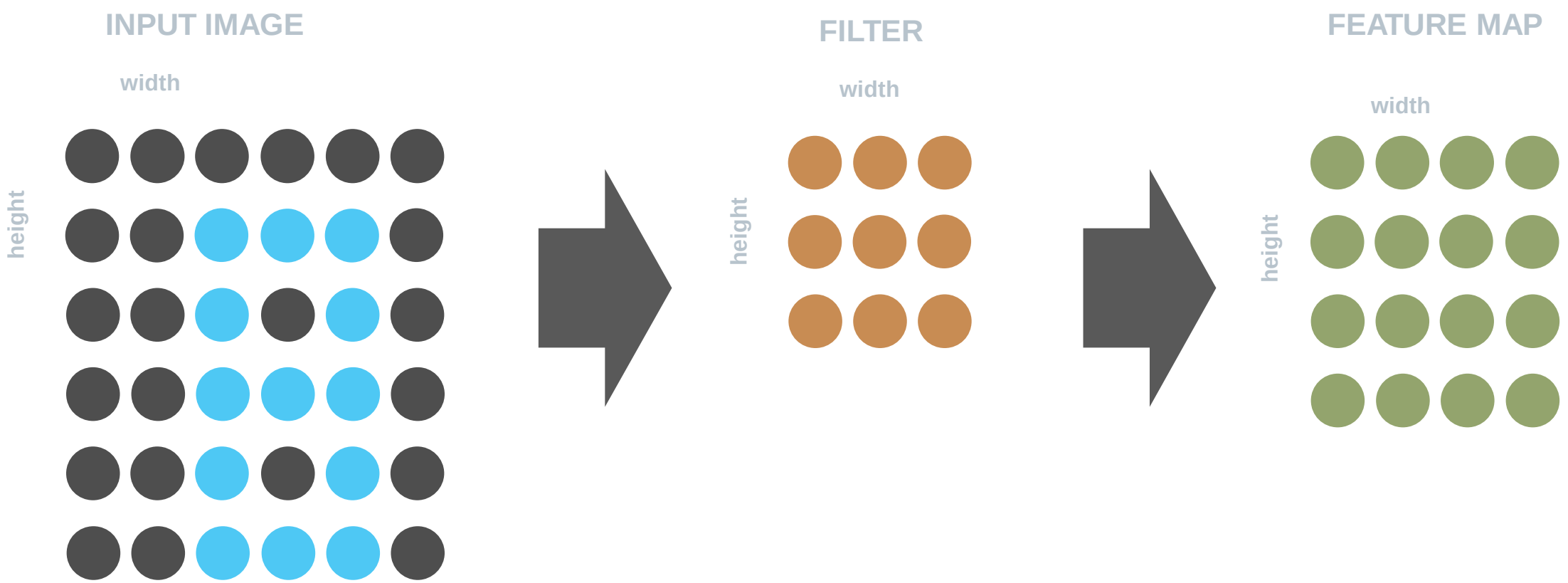
Basic Building Blocks
The Fully Connected Layer – Using Brute Force



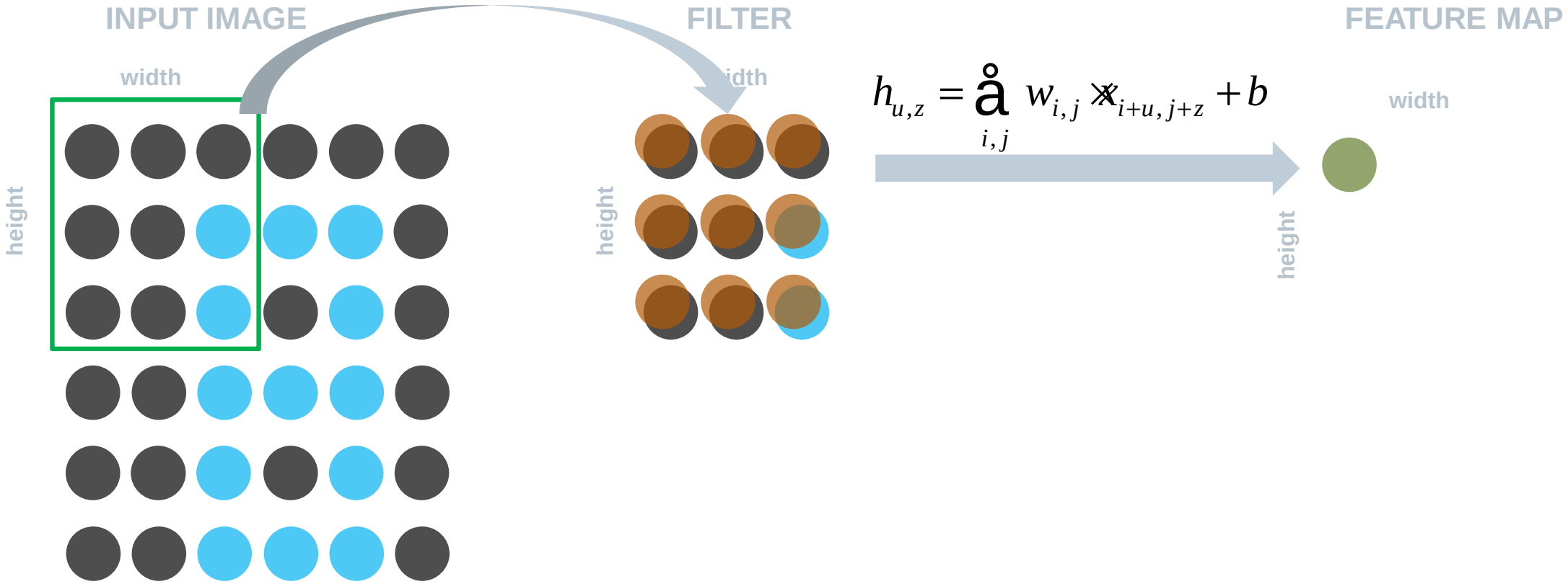

```python
import numpy as np
import tensorflow as tf


# Define placeholder for input data.
input_data = tf.keras.Input(shape=(3,))

# Apply a dense layer on the input.
h = tf.keras.layers.Dense(units=4, activation=tf.nn.tanh)(input_data)

# Feed some random data through the layer.
random_data = np.random.uniform(0, 1, size=(10, 3)).astype(np.float32)
with tf.Session() as session:
    # We need to initialize the layer parameters first.
    session.run(tf.global_variables_initializer())
    # Feed the network with the random data.
    layer_output = session.run(h, feed_dict={input_data: random_data})
    print layer_output.shape  # gives: (10, 4)
```

Basic Building Blocks

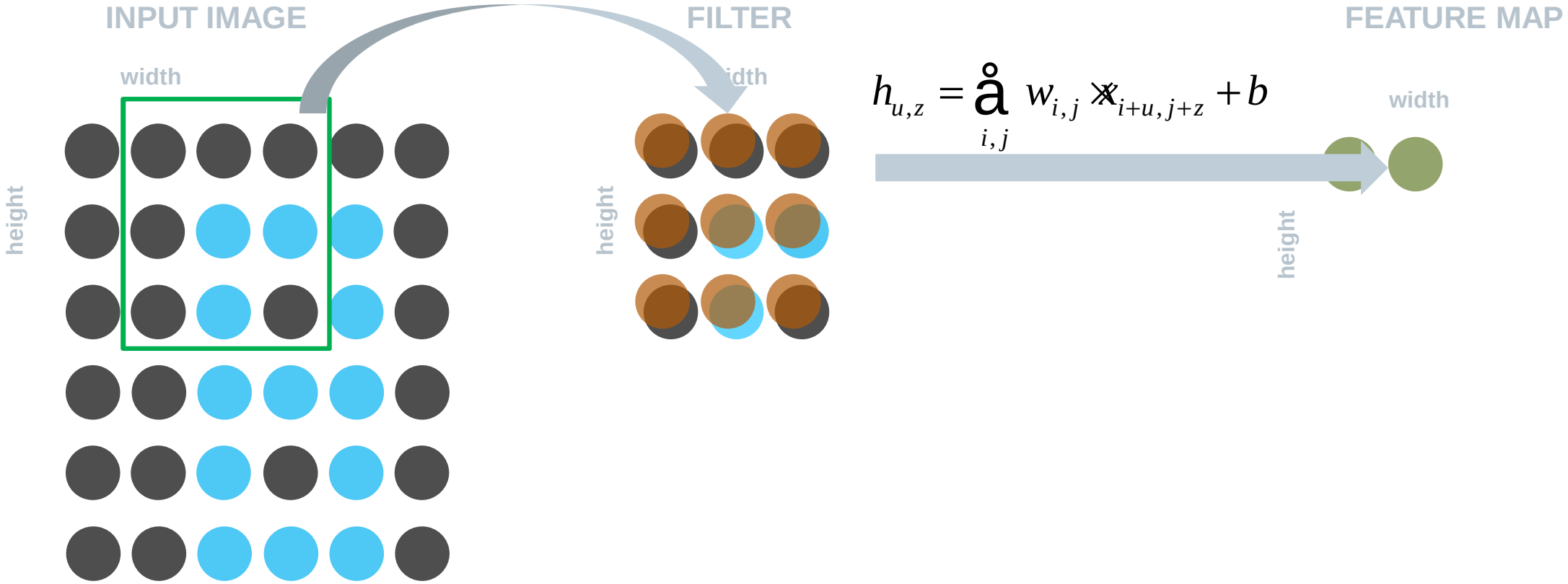
Convolutional Layer - Convolution of Filters



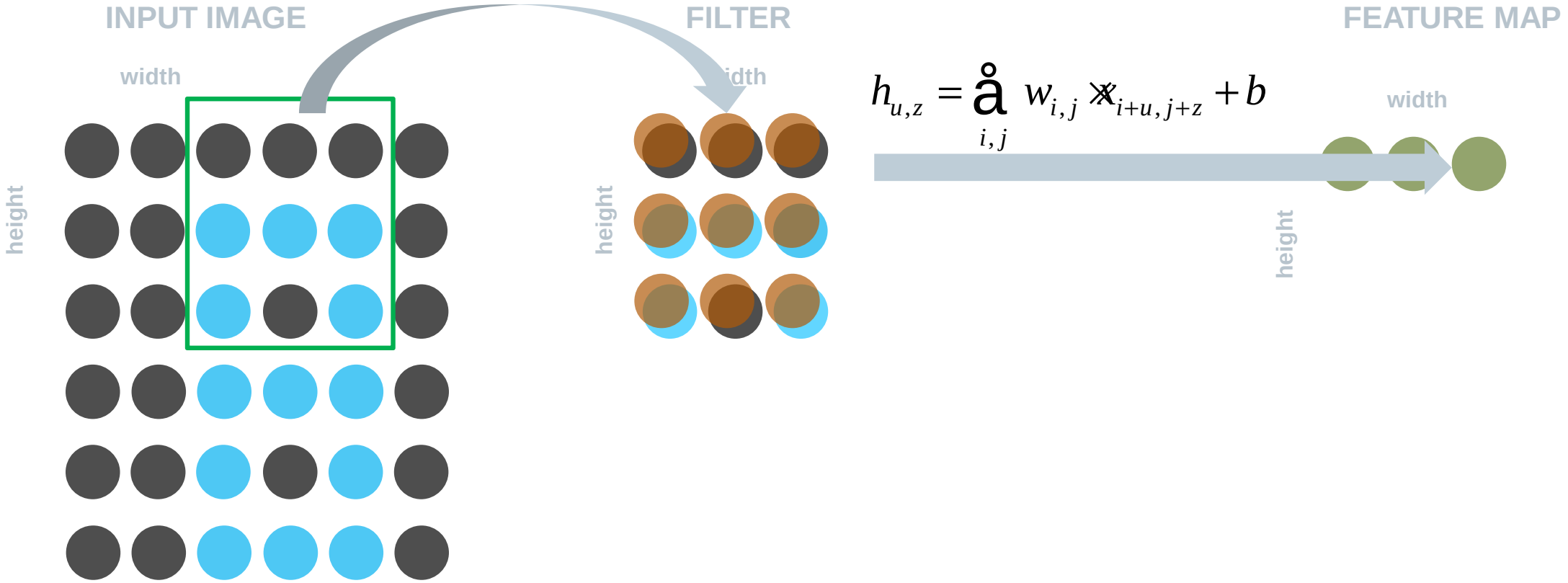
Basic Building Blocks
(Valid) Convolution of Filter



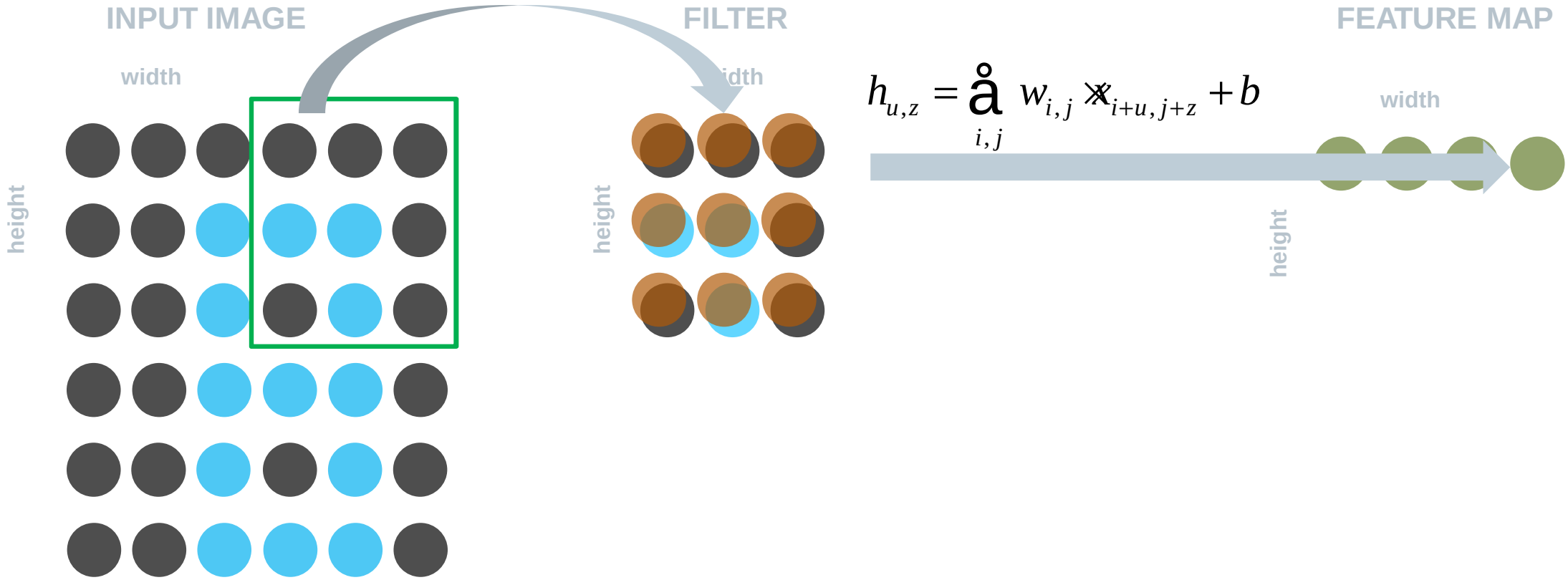
Basic Building Blocks
(Valid) Convolution of Filter



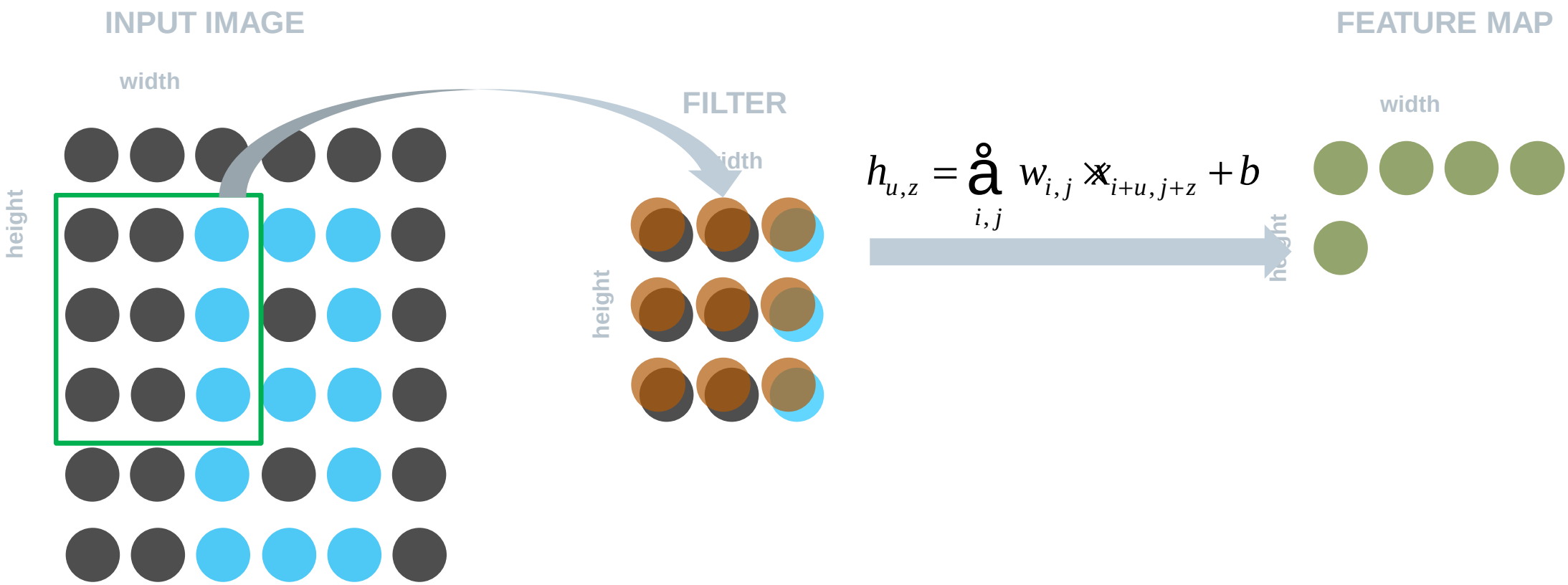
Basic Building Blocks
(Valid) Convolution of Filter



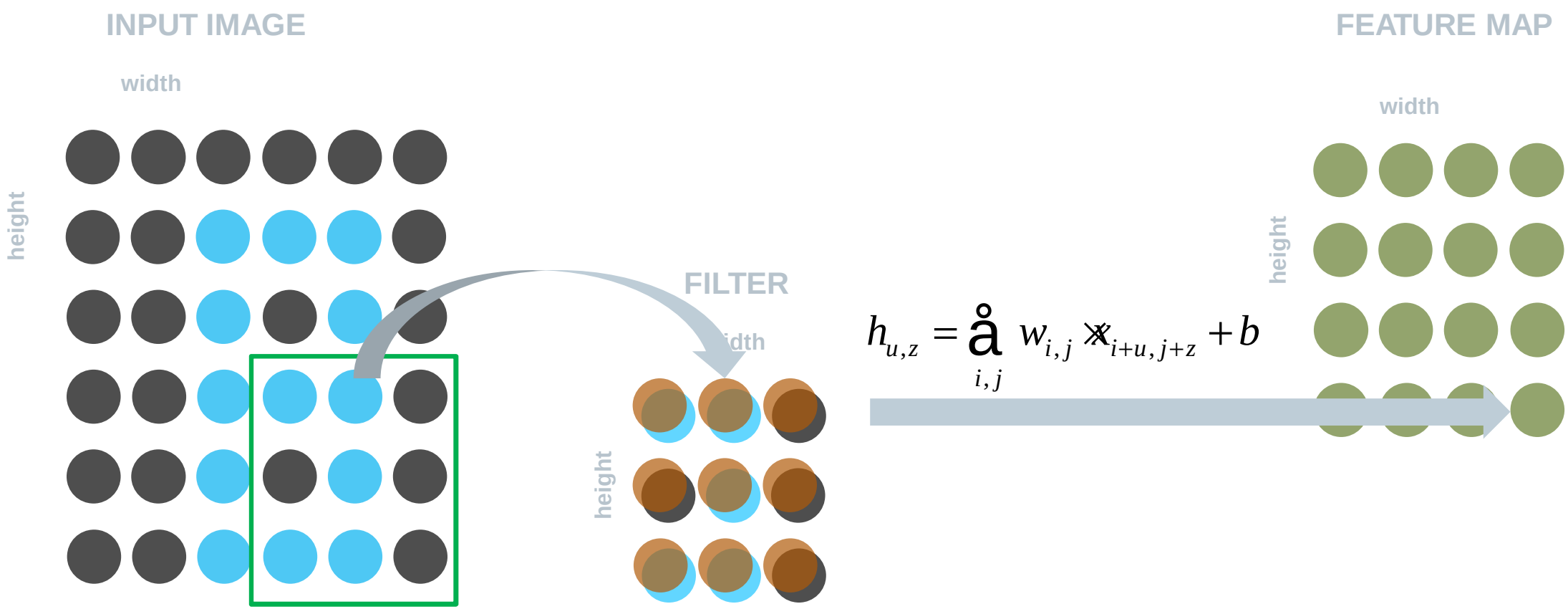
Basic Building Blocks
(Valid) Convolution of Filter



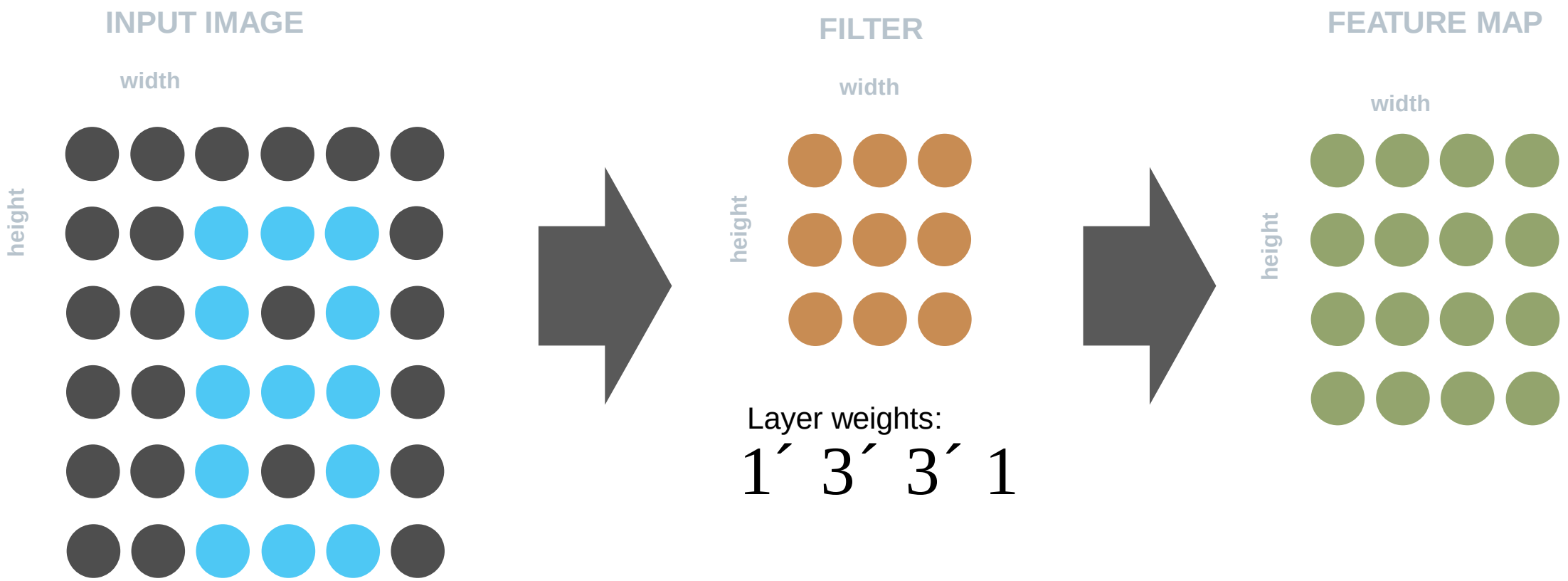
Basic Building Blocks
(Valid) Convolution of Filter



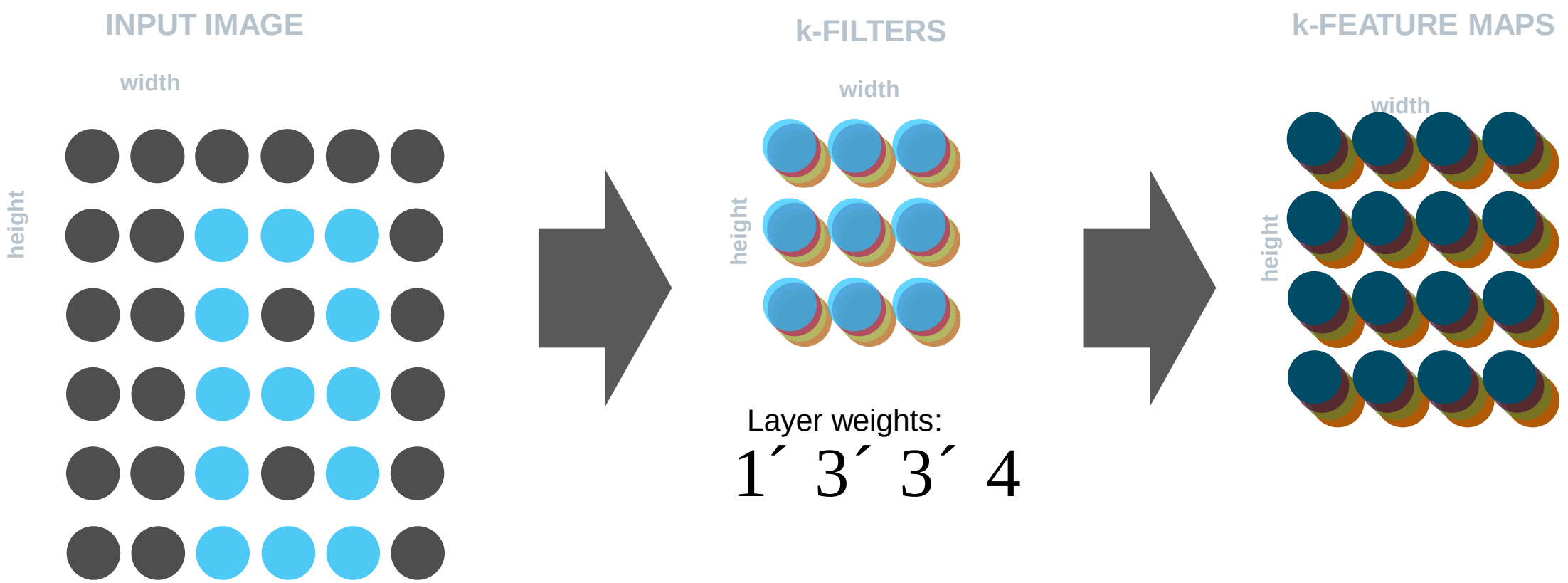
Basic Building Blocks
(Valid) Convolution of Filter



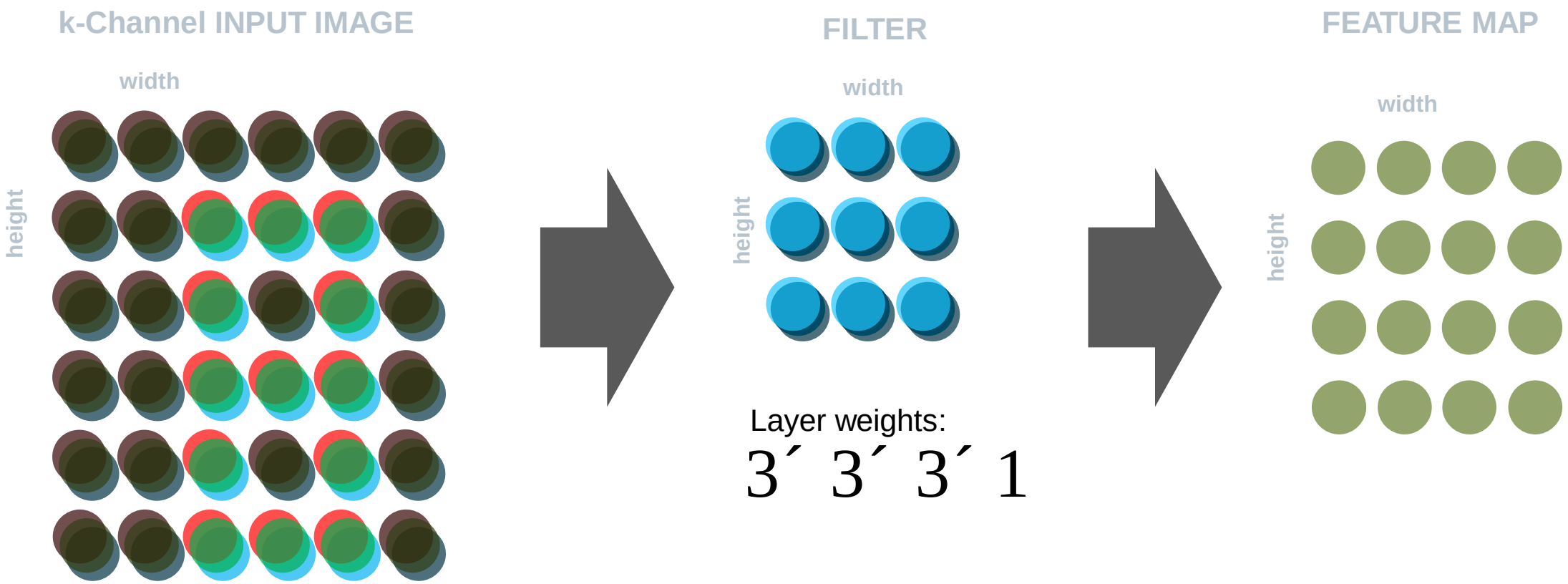
Basic Building Blocks
Convolutional Layer – Single Filter



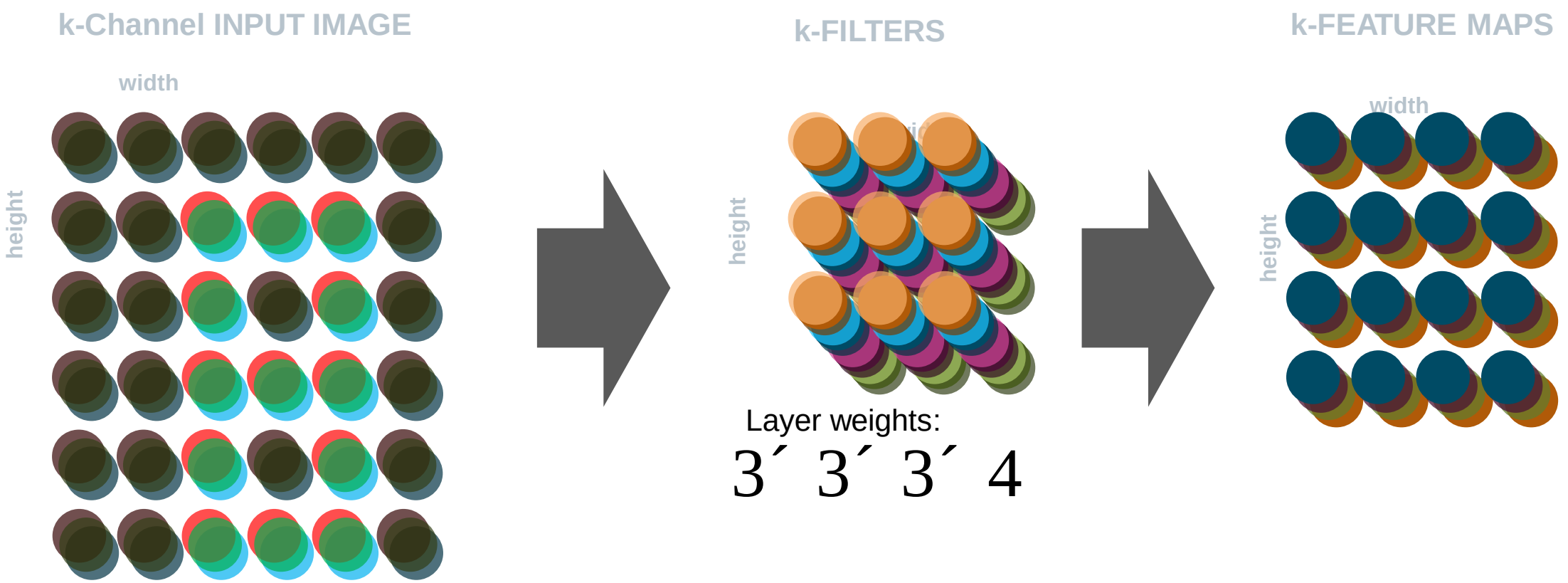
Basic Building Blocks
Convolutional Layer – Multiple Filters



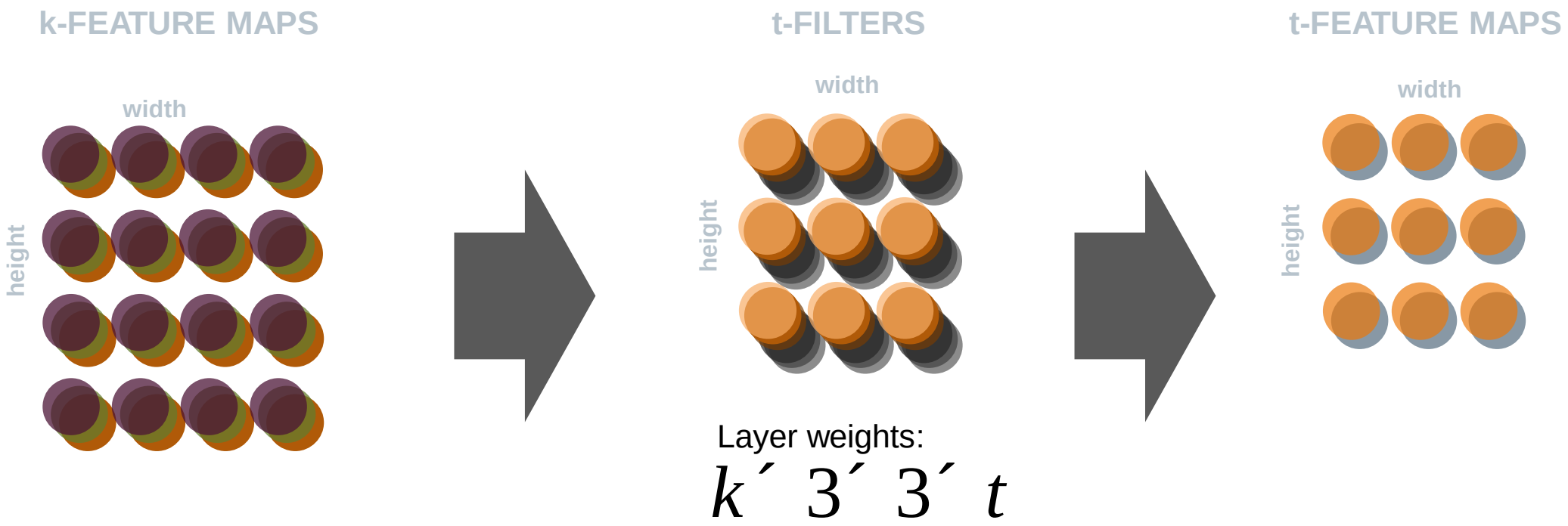
Basic Building Blocks
Convolutional Layer – Multi-Channel Input



Basic Building Blocks
Convolutional Layer – Multi-Channel Input



Basic Building Blocks
Convolutional Layer - Stacking

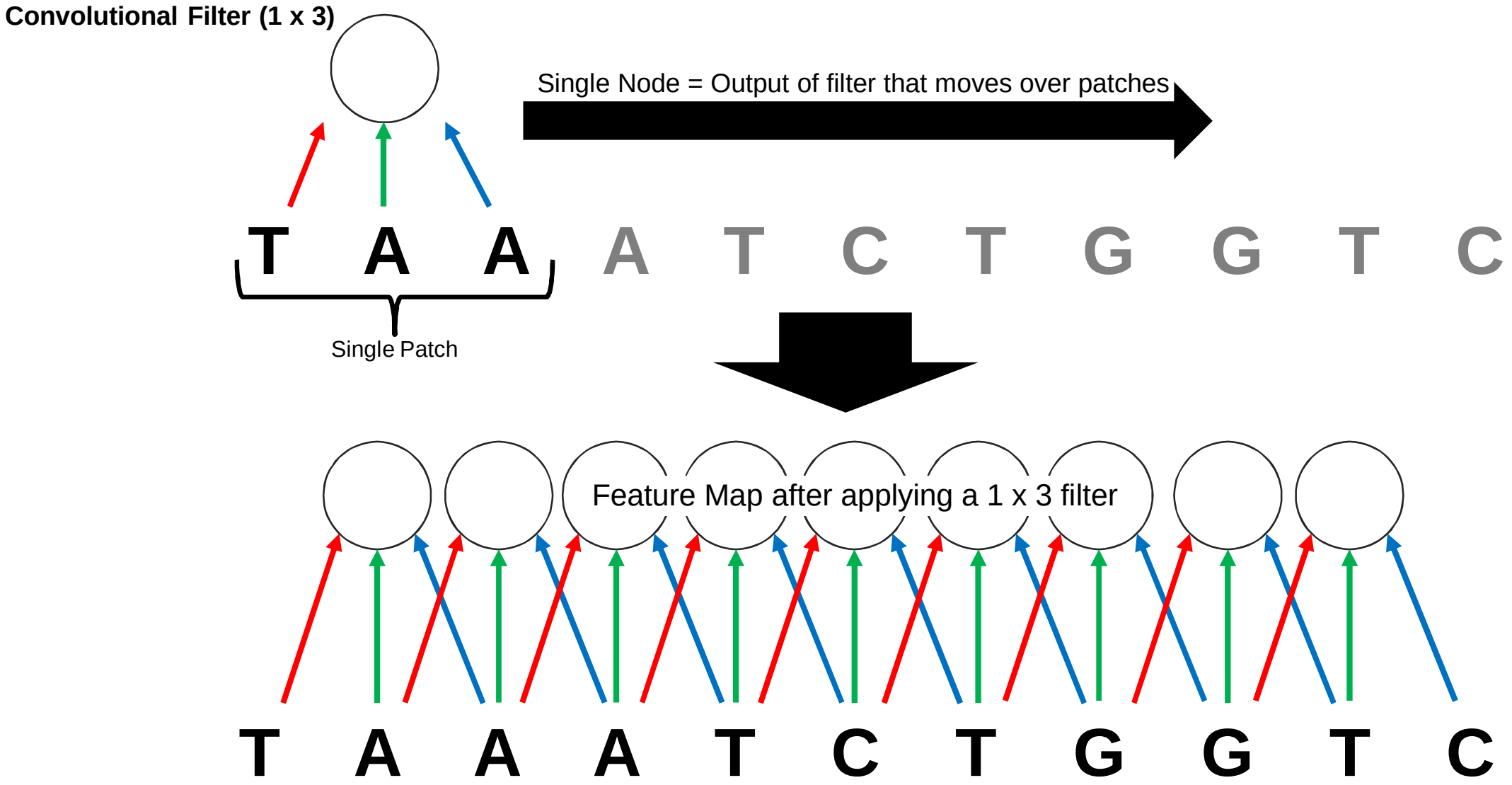


Layer weights:

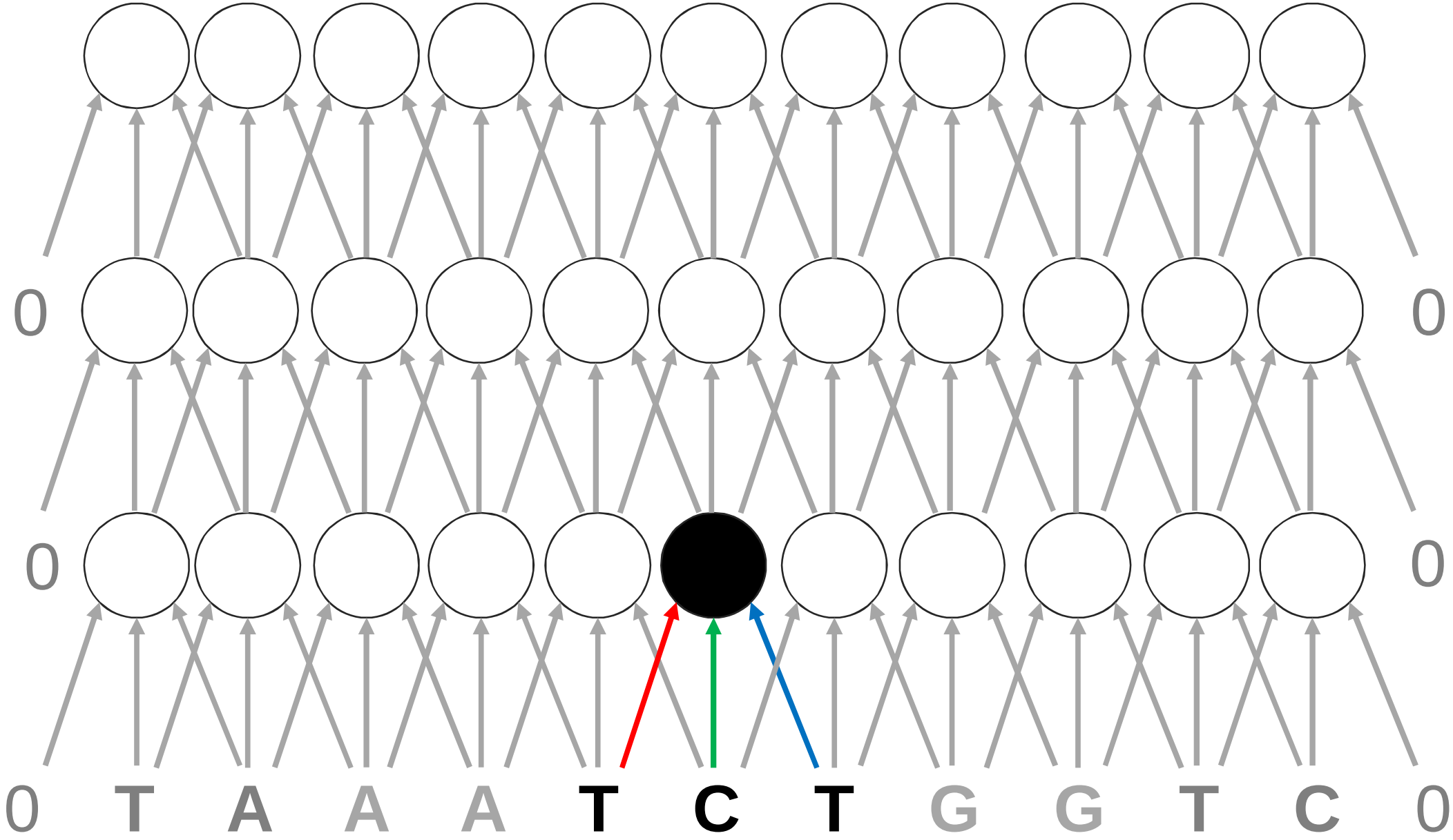
$$k \acute{} \; 3 \acute{} \; 3 \acute{} \; t$$

Basic Building Blocks

Convolutional Layer – Receptive Field Expansion

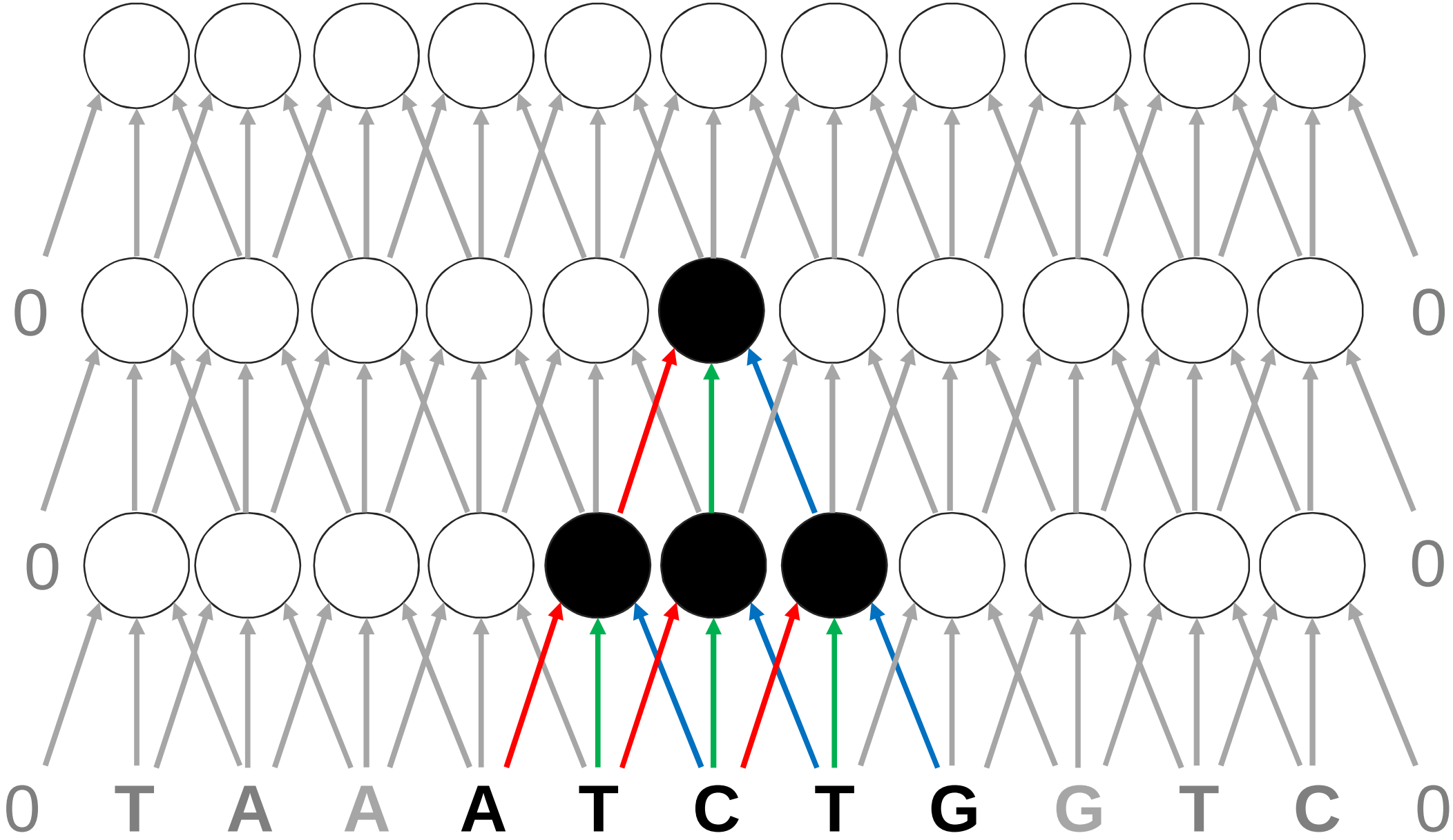


Convolutional Layer – Receptive Field Expansion.



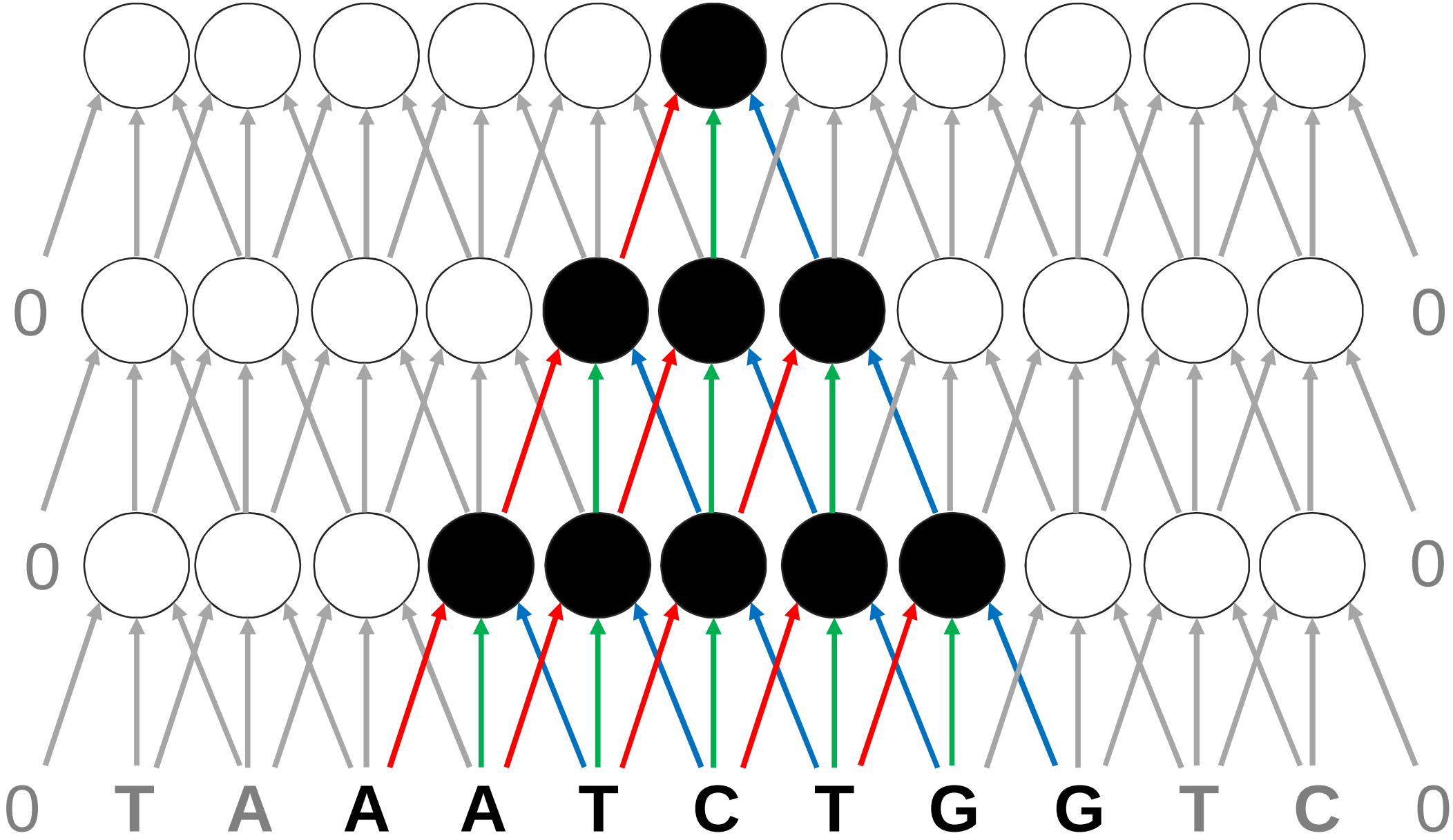
Expansion of the receptive field for a 1 x 3 filter: 2i +1

Convolutional Layer – Receptive Field Expansion.



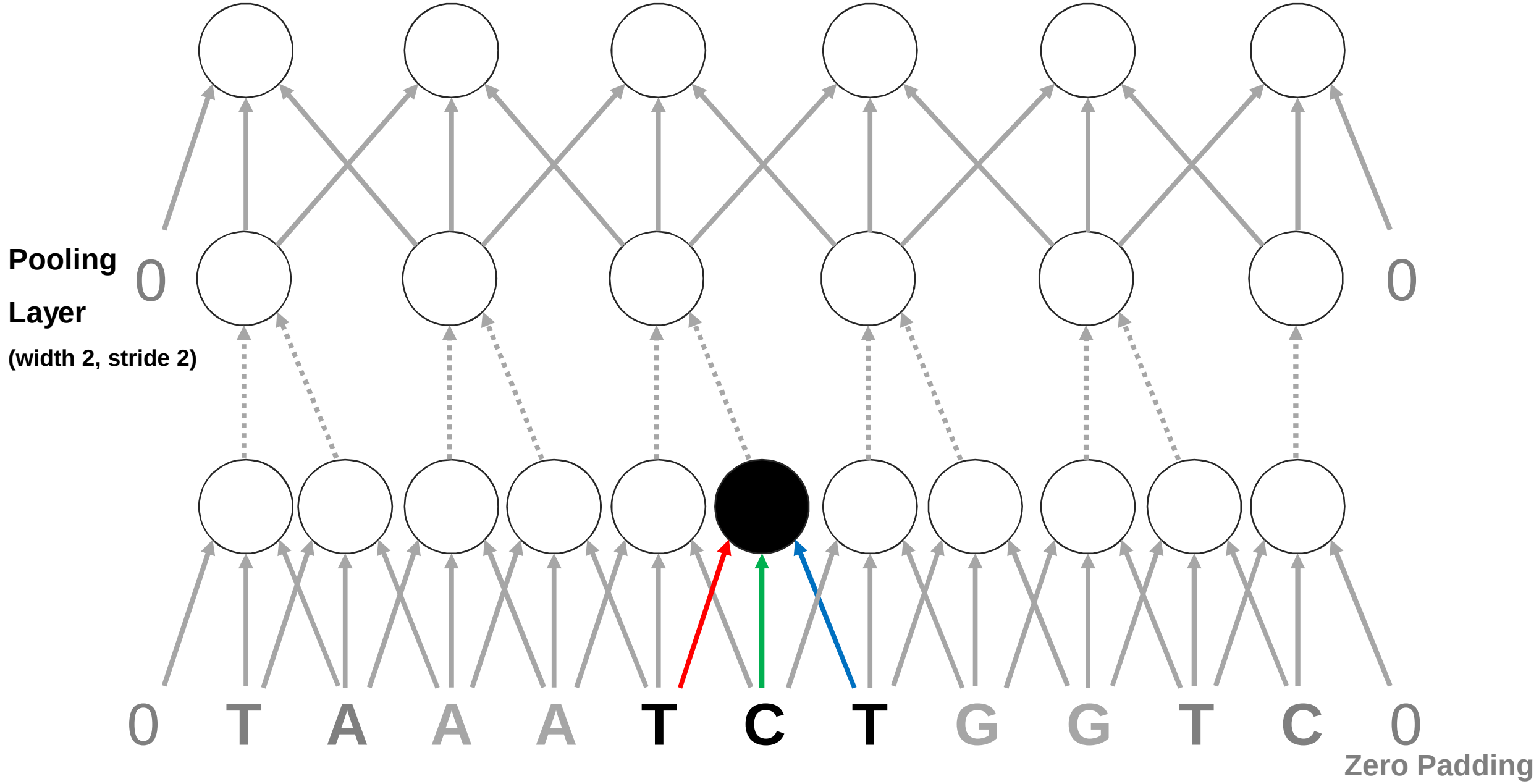
Expansion of the receptive field for a 1 x 3 filter: 2i +1

Convolutional Layer – Receptive Field Expansion.

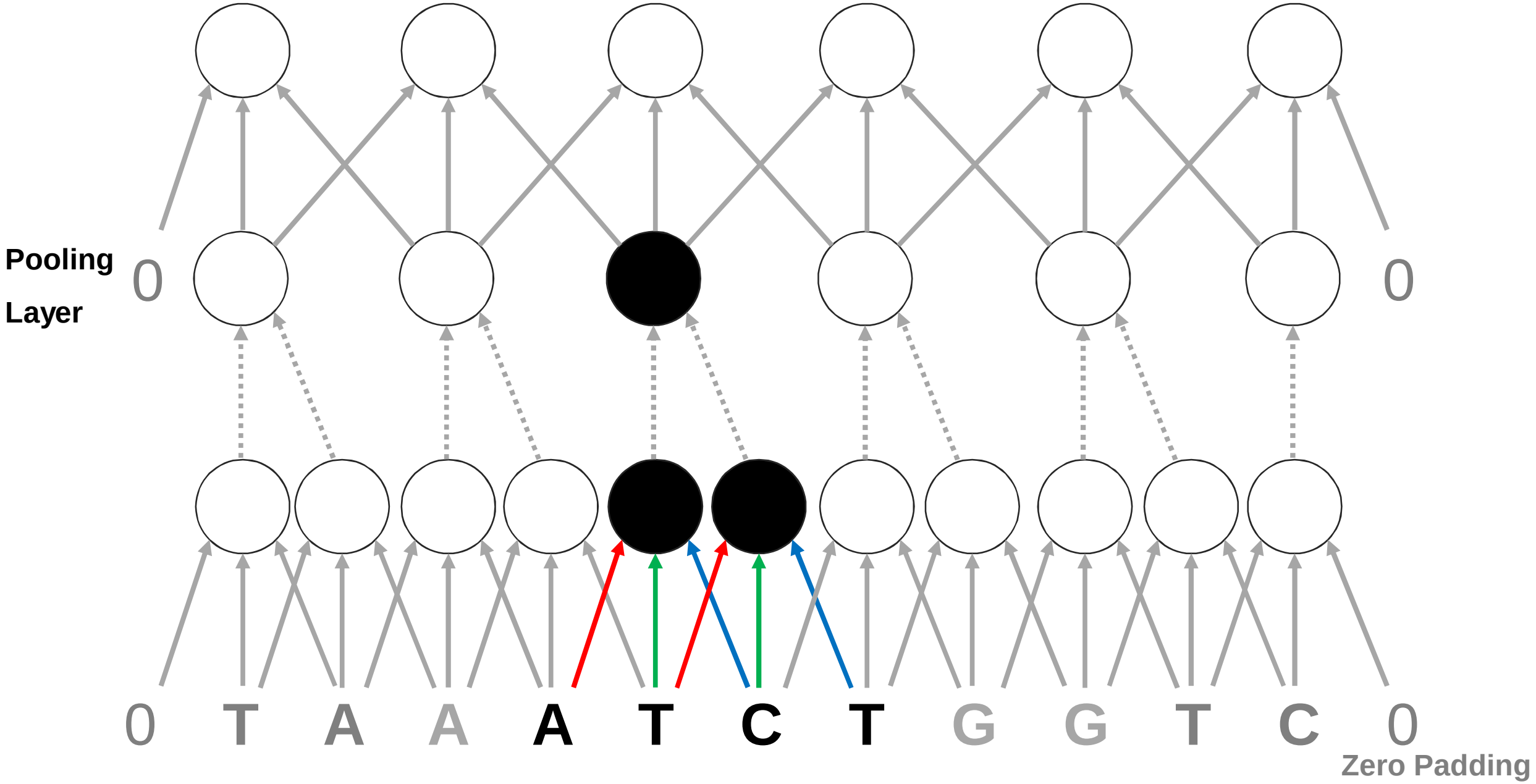


Expansion of the receptive field for a 1 x 3 filter: 2i +1

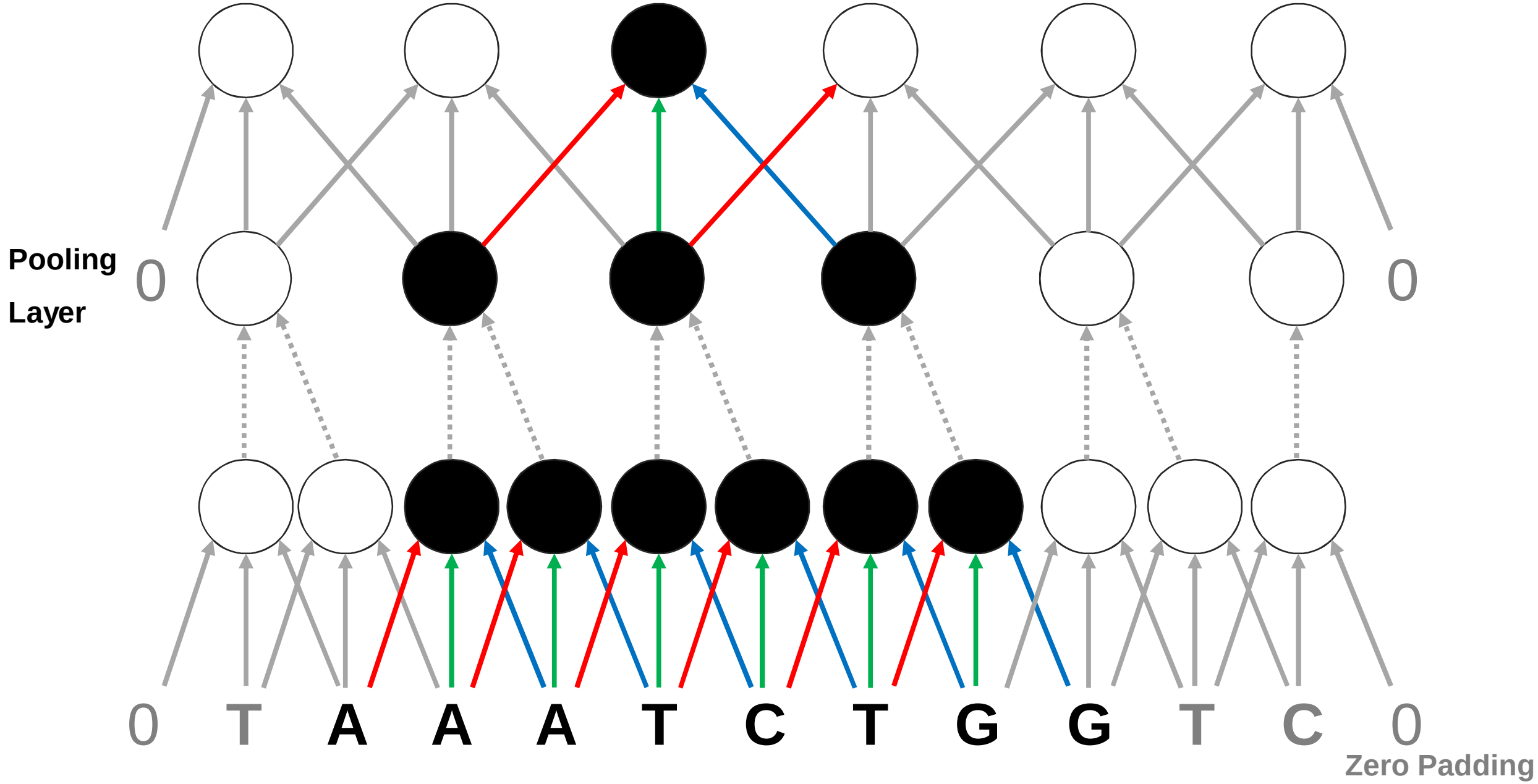
Convolutional Layer – Receptive Field Expansion with Pooling Layer.



Convolutional Layer – Receptive Field Expansion with Pooling Layer.

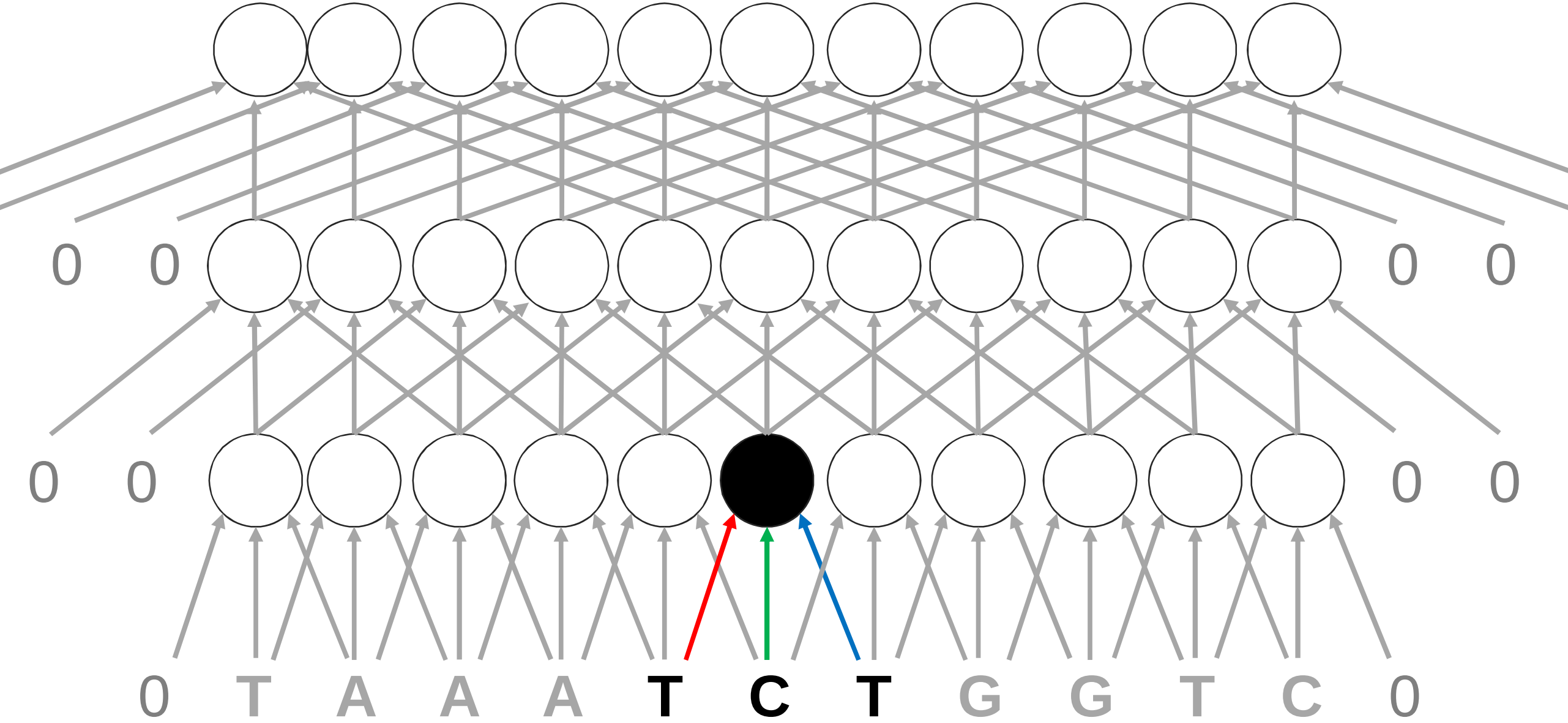


Convolutional Layer - Receptive Field Expansion with Pooling Layer.



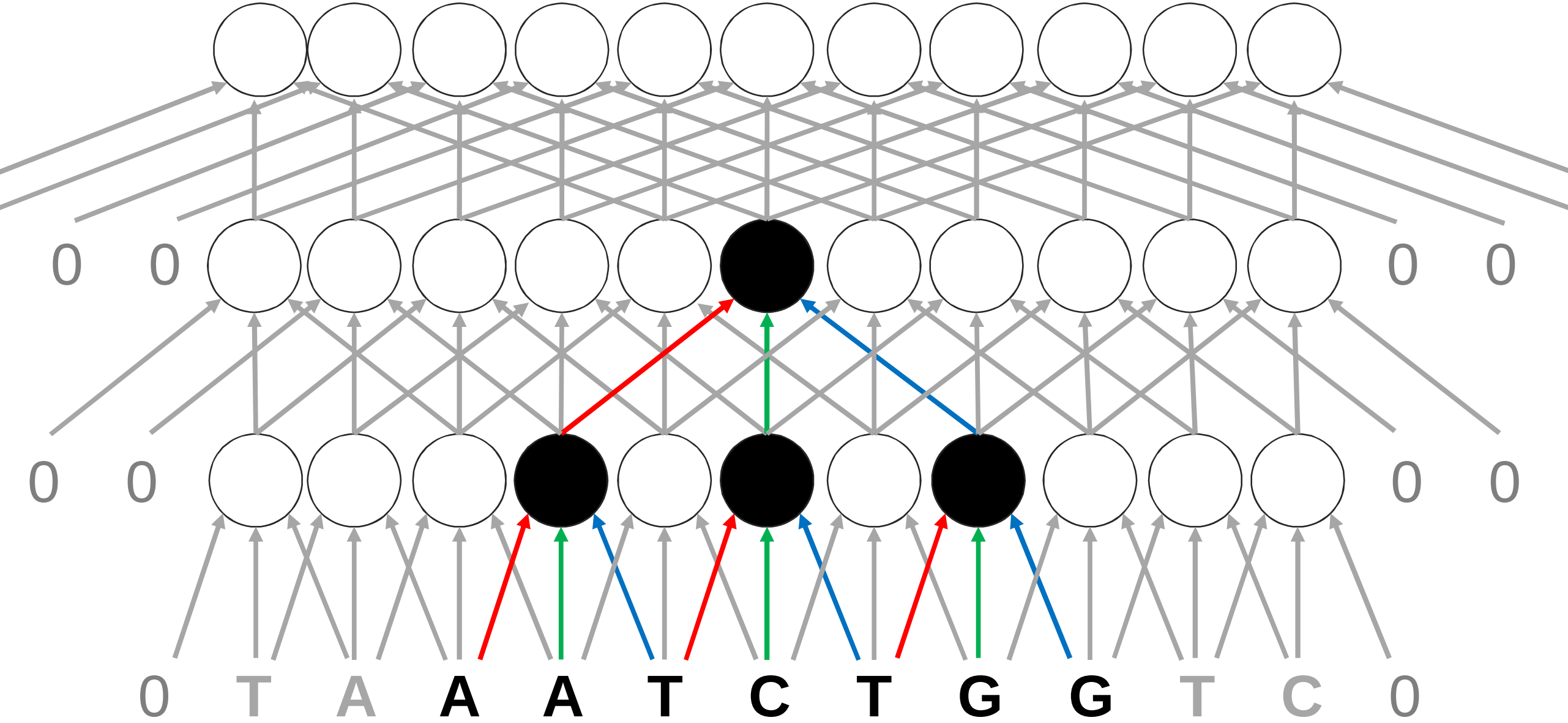
Convolutional Layer – Receptive Field Expansion with Dilation

Paper https://arxiv.org/pdf/1511.07122.pdf



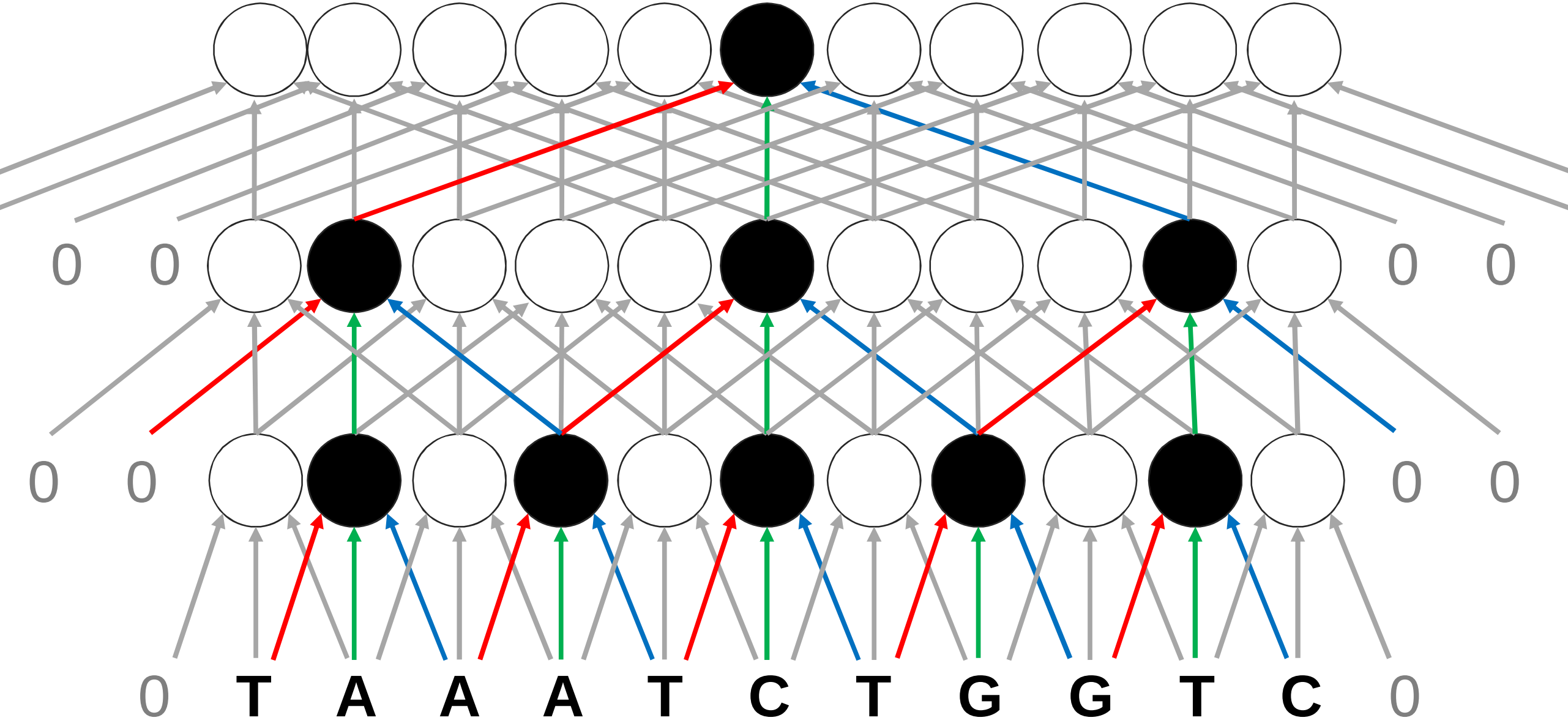
Expansion of the receptive field for a 1 x 3 filter: $2^{i+1} - 1$

Convolutional Layer - Receptive Field Expansion with Dilation



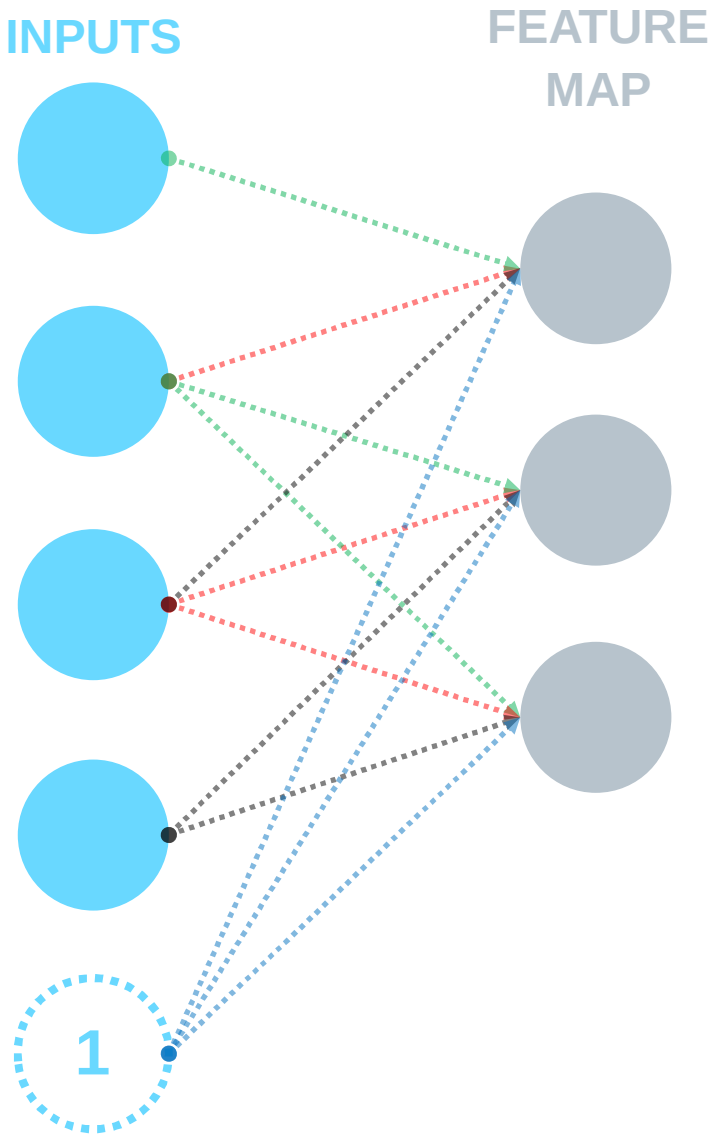
Expansion of the receptive field for a 1 x 3 filter: $2^{i+1} - 1$

Convolutional Layer - Receptive Field Expansion with Dilation



Expansion of the receptive field for a 1 x 3 filter: $2^{i+1} - 1$

Basic Building Blocks
Convolutional Layer – Exploiting Neighborhood Relations



Convolutional layer:

- Exploits neighborhood relations of the inputs (e.g. spatial).
- Applies small fully connected layers to small patches of the input.
  - ØVery efficient!
  - ØWeight sharing
  - ØNumber of free parameters

$$\#\text{input channels} \cdot \text{filter height} \cdot \text{filter width} \cdot \#\text{filters}$$

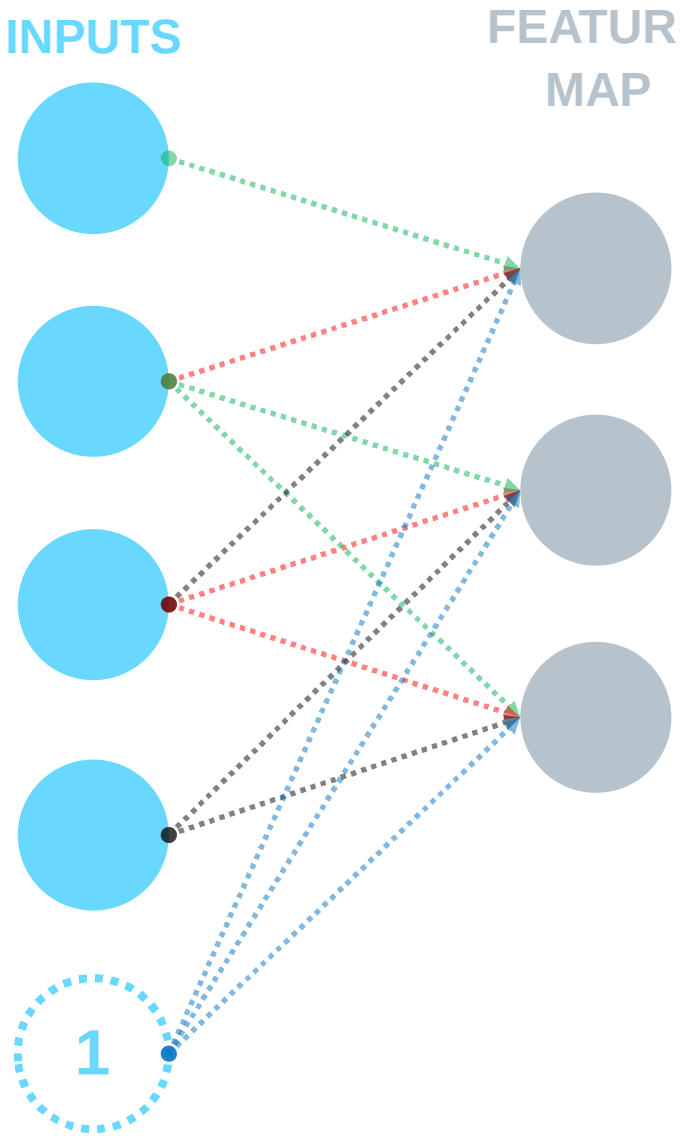
- The receptive field can be increased by stacking multiple layers
- Should only be used if there is a notion of neighborhood in the input:
  - Text, images, sensor time-series, videos, …

Example:

RGB image of shape
100 x 100 x 3

2,700 free parameters for a convolutional layer with 100 hidden units (filters) with a filter size of 3 x 3!

Basic Building Blocks
Convolutional Layer – Exploiting Neighborhood Relations

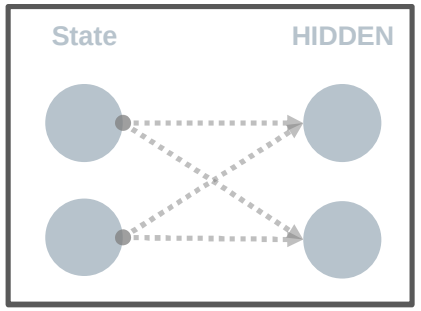


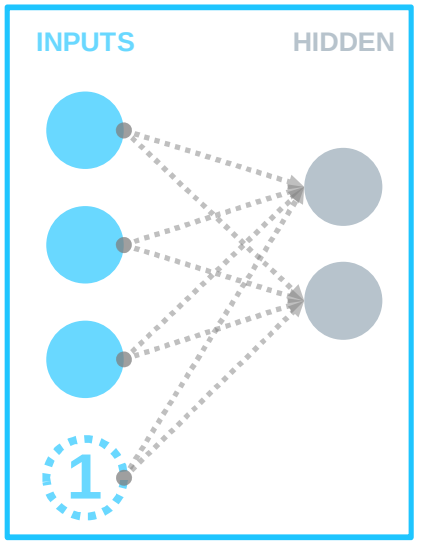

```python
import numpy as np
import tensorflow as tf


# Define placeholder for input 24 x 24 rgb images.
input_images = tf.keras.Input(shape=(24, 24, 3))

# Apply a convolutional layer on the input images.
h = tf.keras.layers.Conv2D(
    filters=8, kernel_size=[3, 3], strides=[1, 1], activation=tf.nn.relu,
    padding='same')(input_images)

# Generate 10 random images which we will feed to the layer.
random_images = np.random.uniform(
    0, 1, size=(10, 24, 24, 3)).astype(np.float32)
with tf.Session() as session:
    # We need to initialize the layer parameters first.
    session.run(tf.global_variables_initializer())
    # Feed the network with the random images.
    output_feature_maps = session.run(
        h, feed_dict={input_images: random_images})
    print output_feature_maps.shape  # gives: (10, 24, 24, 8)
```

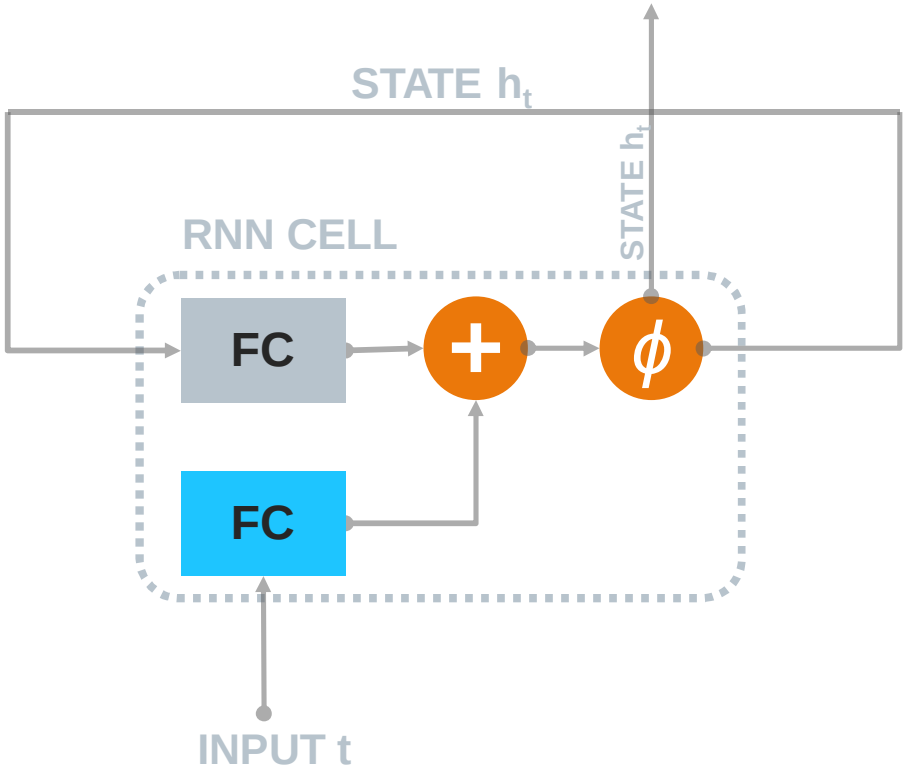
Basic Building Blocks
Recurrent Neural Network Layer – The RNN cell

FC = Fully connected layer
+ = Addition
Φ = Activation function

$$h_t = f\left(Uh_{t-1} + Wx_t + b\right)$$

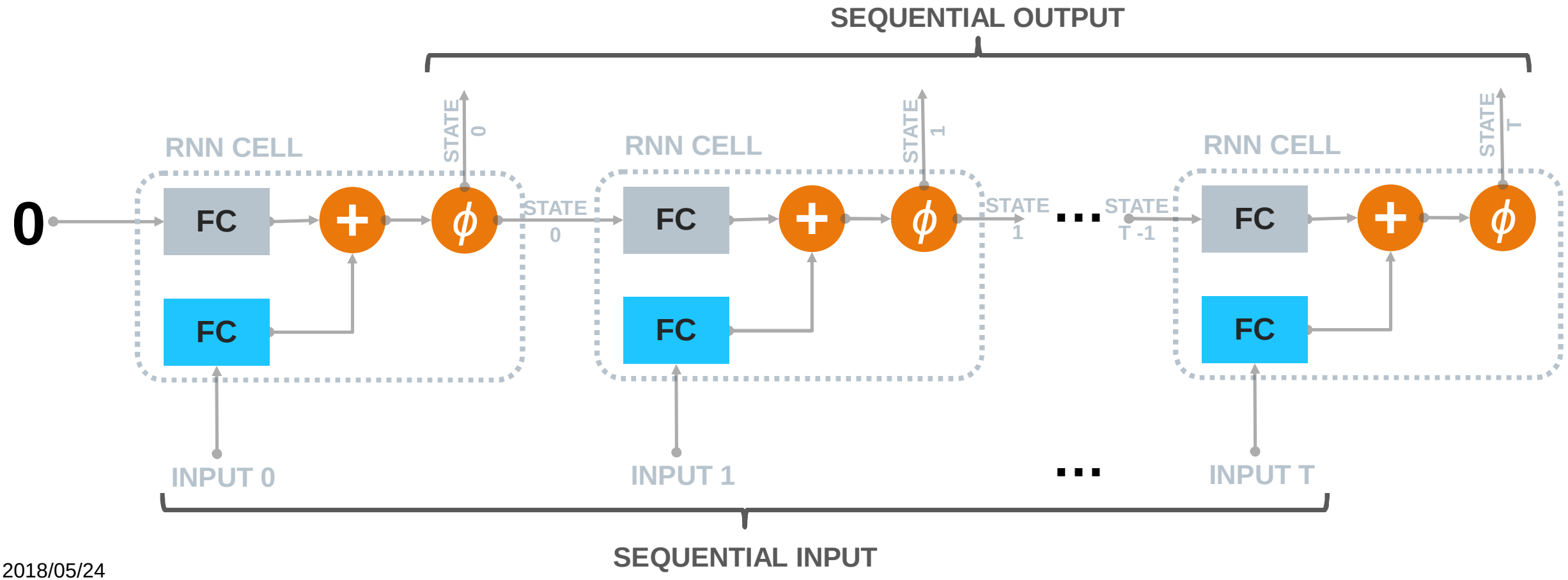






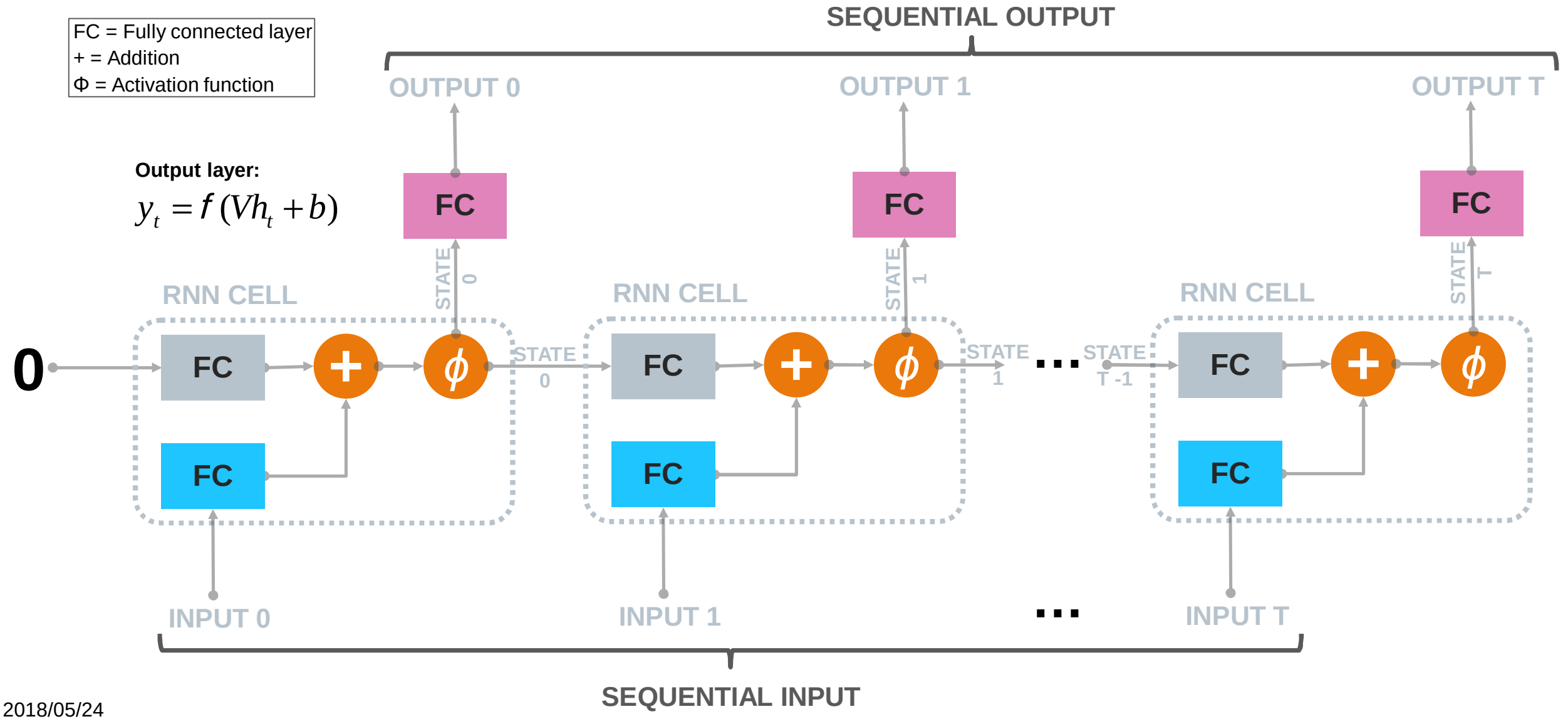
Basic Building Blocks
Recurrent Neural Network layer – Unfolded

FC = Fully connected layer
+ = Addition
Φ = Activation function



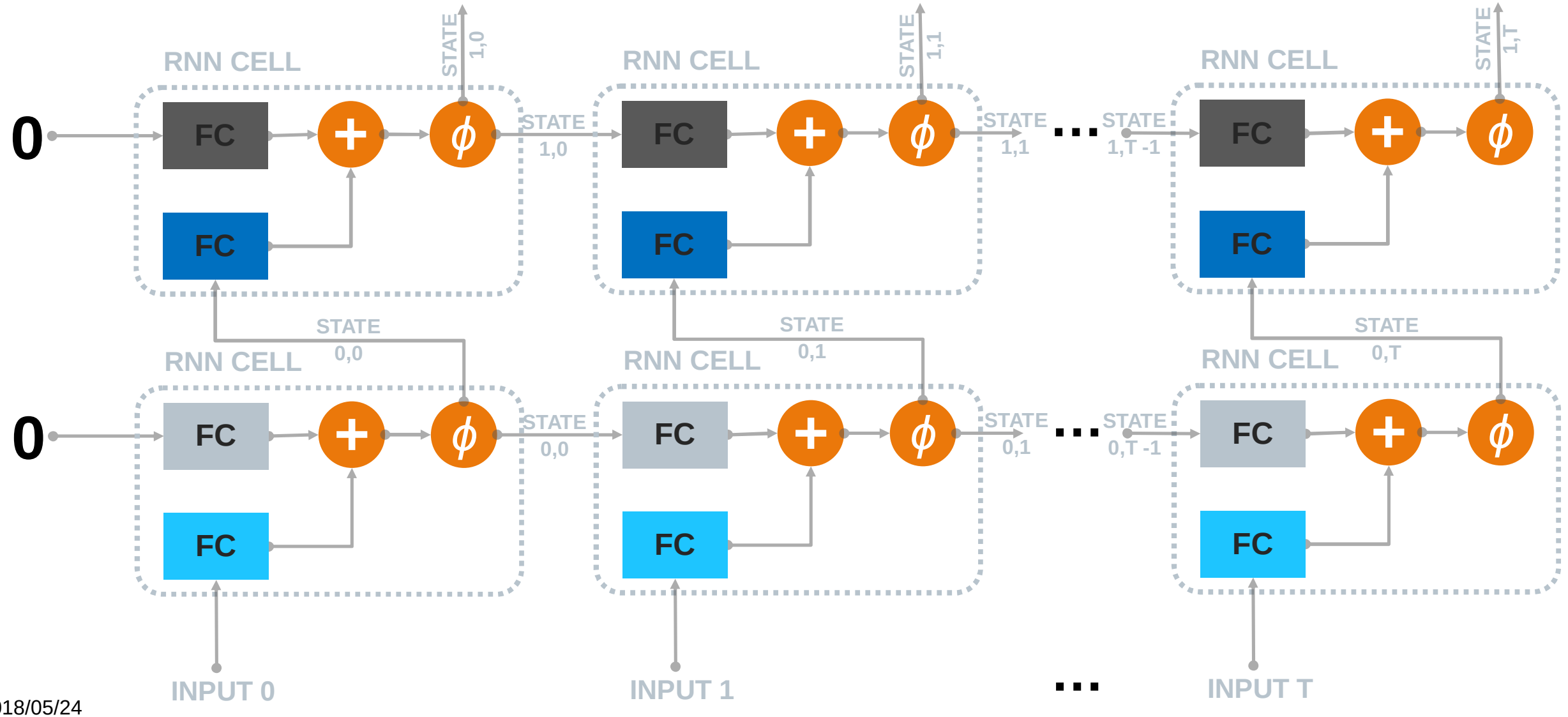
Basic Building Blocks

Vanilla Recurrent Neural Network (unfolded)



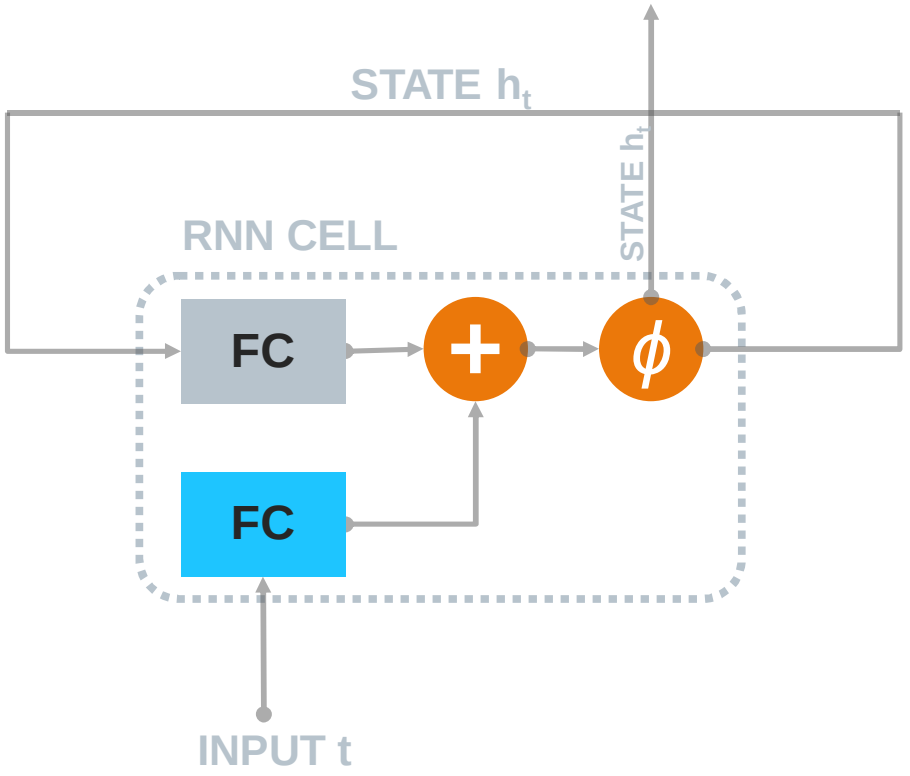
Basic Building Blocks

Recurrent Neural Network Layer – Stacking



Basic Building Blocks

Recurrent Neural Network Layer – Exploiting Sequential Relations



RNN layer:

• Exploits sequential dependencies (Next prediction might depend on things that were observed earlier).

•Applies the same (via parameter sharing) fully connected layer to each step in the input data and combines it with collected information from the past (hidden state).

ØDirectly learns sequential (e.g. temporal) dependencies.

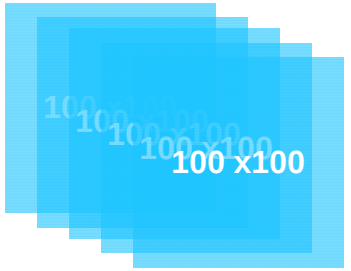
•Stacking can help to learn deeper hierarchical state representations.

•Should only be used if sequential sweeping of the data makes sense: Text, sensor time-series, (videos, images)…

•Vanilla RNN is not able to capture long-time dependencies!

•Use with care since it can also quickly over-parameterize the model

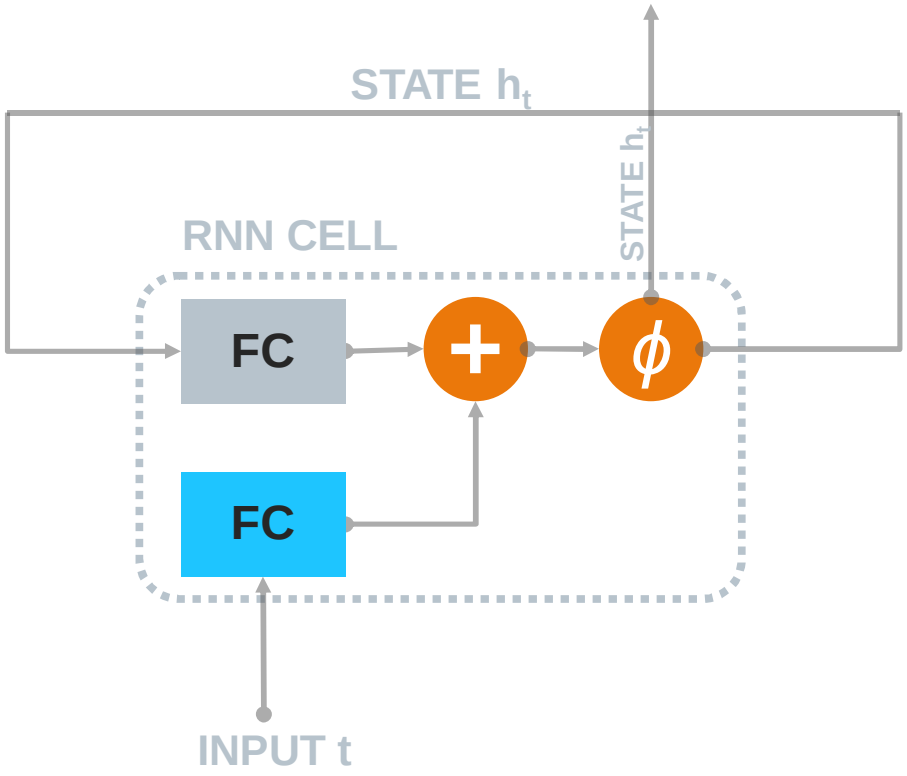
ØCan lead to degenerated solutions.



e.g. Videos of frames of shape100 x 100 x 3

Basic Building Blocks

Recurrent Neural Network Layer – Exploiting Sequential Relations




```python
import numpy as np
import tensorflow as tf


# Define placeholder for inputs which contain 3 sequences of length 20.
input_sequences = tf.keras.Input(shape=(20, 3))

# Apply a RNN on the sequences.
h = tf.keras.layers.SimpleRNN(
    units=4, activation=tf.nn.tanh, return_sequences=True)(input_sequences)

# Generate 10 random sequences which we will feed to the layer.
random_sequences = np.random.uniform(
    0, 1, size=(10, 20, 3)).astype(np.float32)
with tf.Session() as session:
    # We need to initialize the layer parameters first.
    session.run(tf.global_variables_initializer())
    # Feed the network with the random sequences.
    output_feature_maps = session.run(
        h, feed_dict={input_sequences: random_sequences})
    print output_feature_maps.shape  # gives: (10, 20, 4)

```

Deep Learning

Thinking in Macro Structures

Thinking in Macro Structures

Remember the Important Things – And Move On

Important things:
- Purpose
- Weaknesses
- General usage
- Tweaks

Fully Connected Layer	**FC**	+	In case no assumptions on the input data can be exploited. (Treat all inputs as independent)
Convolutional Layer	**CNN**	+	Good for exploiting spatial/sequential dependencies in the data.
Recurrent Neural Network Layer	**RNN**	+	Good for modeling sequential data with no long term dependencies

With these three basic building blocks, we are already able to do amazing stuff!

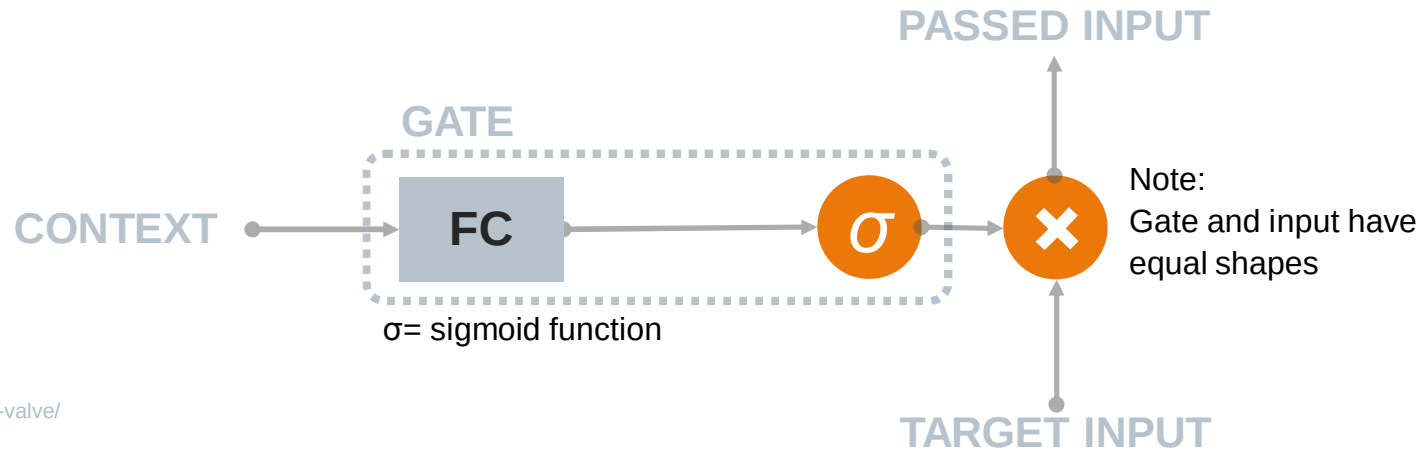
Thinking in Macro Structures
Mixing Things Up – Generating Purpose Modules.

- Given the basic building blocks introduced in the last section:

- We can construct modules that address certain sub-task within the model that might be beneficial for reaching the actual target goal.
  - E.g. Gating, Attention, Hierarchical feature extraction, …
- These modules can further be combined to form even larger modules serving a more complex purpose
  - LSTMs, Residual Units, Fractal Nets, Neural memory management …
- Finally all things are further mixed up to form an architecture with many internal mechanisms that enables the model to learn very complex tasks end-to-end.
  - Text translation, Caption generation, Neural Computer…

Thinking in Macro Structures
Controlling the Information Flow – Gating

Image: https://au.pinterest.com/explore/tap-valve/

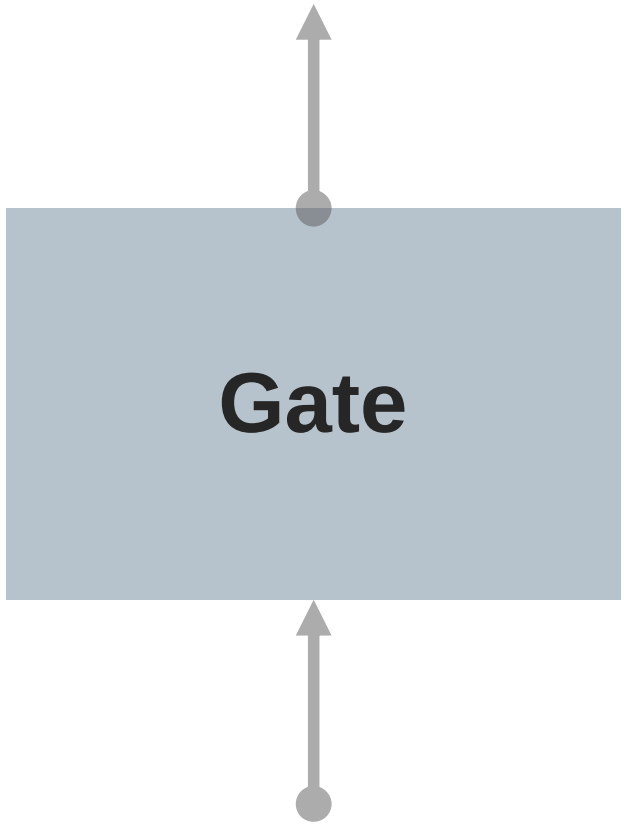


Thinking in Macro Structures
Controlling the Information Flow – Gating

```python
import numpy as np
from numpy.testing import assert_array_equal
import tensorflow as tf


# Define placeholder for input data.
input_data = tf.keras.Input(shape=(3,))
context_data = tf.keras.Input(shape=(6,))

# Concatenate the context and the input data to produce the data that
# influences the gating.
gating_context = tf.concat([input_data, context_data], axis=-1)

# Apply a dense layer with a sigmoid activation function on the gating context
# which produces a matrix filled with values between 0 and 1.
gate = tf.keras.layers.Dense(
    units=input_data.get_shape()[1], activation=tf.nn.sigmoid)(gating_context)

# Apply the gate on the input data by elementwise multiplication.
gated_input_data = gate * input_data

# Feed some random data through the layer.
random_input_data = np.random.uniform(0, 10, size=(10, 3)).astype(np.float32)
random_context_data = np.random.uniform(0, 10, size=(10, 6)).astype(np.float32)
with tf.Session() as session:
    # We need to initialize the layer parameters first.
    session.run(tf.global_variables_initializer())
    # Feed the network with the random data.
    gated_random_input_data = session.run(
        gated_input_data,
        feed_dict={input_data: random_input_data,
                   context_data: random_context_data})
    print assert_array_equal(random_input_data, gated_random_input_data)
    # gives: AssertionError:
    # Arrays are not equal
    # (mismatch 100%)
```

Thinking in Macro Structures

Remember the Important Things – And Move On.



\+ Good for controlling information flow in a network

Thinking in Macro Structures

Learning Long-Term Dependencies – The LSTM Cell

Forget gate $f_t = s(W_t x_t + U_f h_{t-1} + b_f)$

Input gate $i_t = s(W_t x_t + U_i h_{t-1} + b_i)$

$c_t = f_t \times c_{t-1} + i_t \times f(W_c x_t + U_c h_{t-1} + b_c)$

Output gate $o_t = s(W_o x_t + U_o h_{t-1} + b_o)$

$h_t = o_t \times c_t$

Thinking in Macro Structures

Learning Long-Term Dependencies – The LSTM Cell

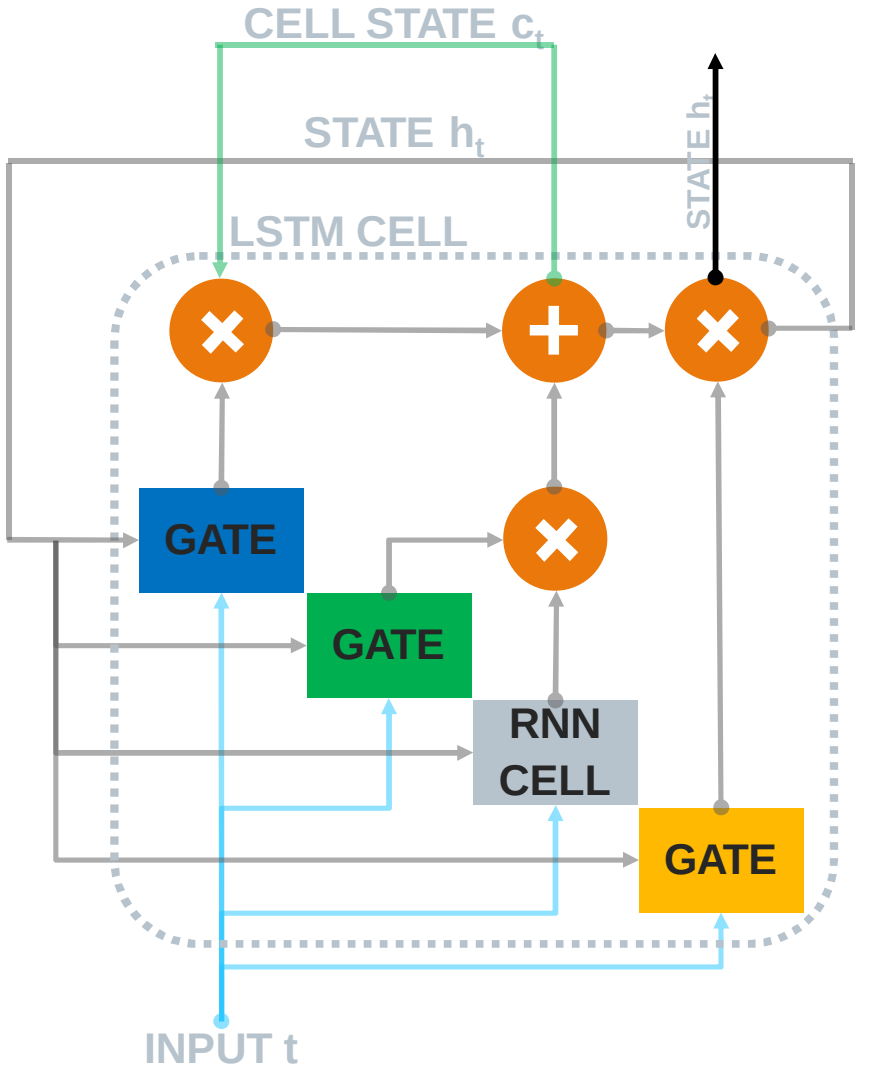
Forget gate $f_t = s(W_t x_t + U_f h_{t-1} + b_f)$

Input gate $i_t = s(W_t x_t + U_i h_{t-1} + b_i)$

$c_t = f_t \times c_{t-1} + i_t \times f(W_c x_t + U_c h_{t-1} + b_c)$

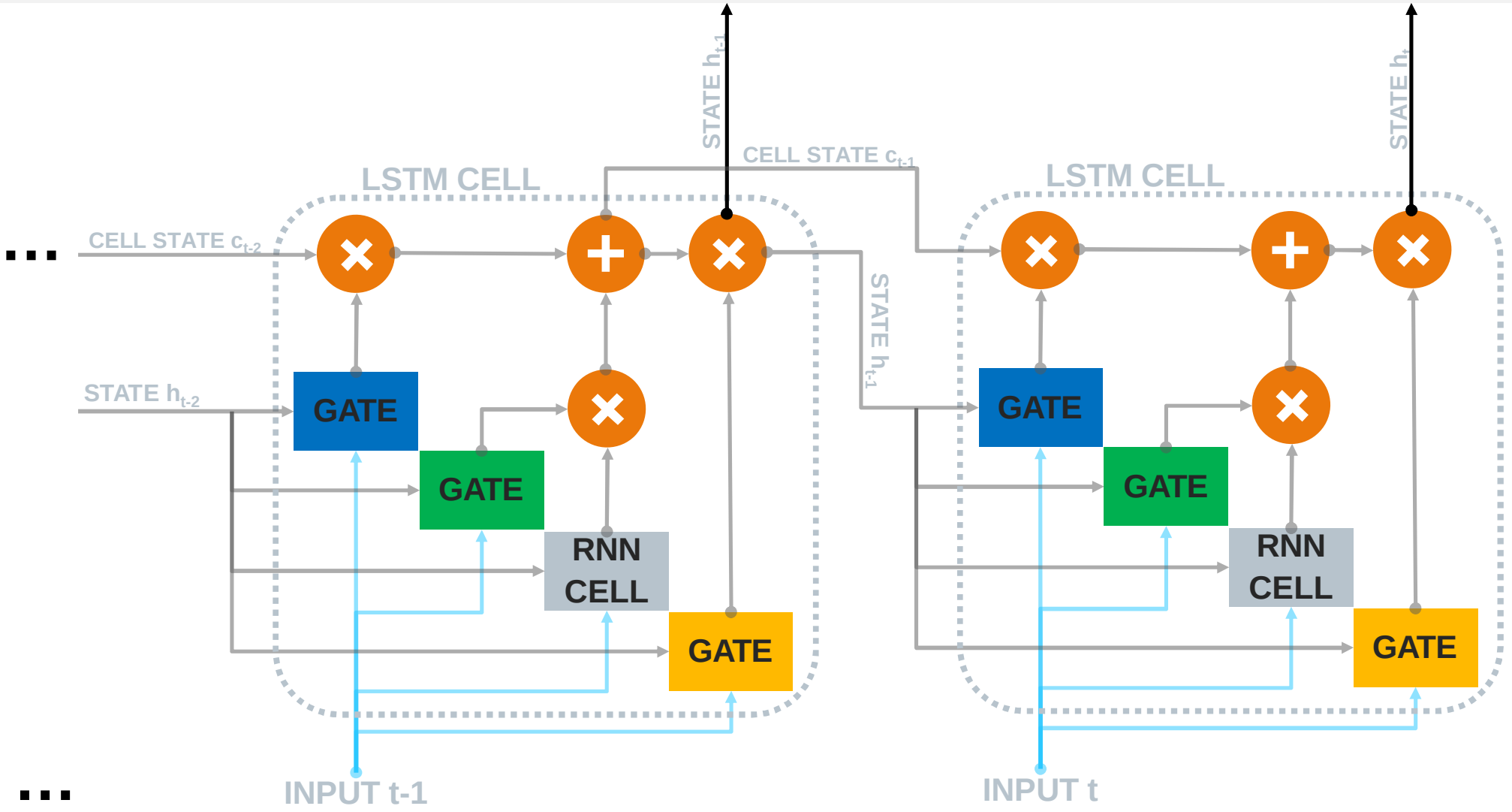
Output gate $o_t = s(W_o x_t + U_o h_{t-1} + b_o)$

$h_t = o_t \times c_t$



Thinking in Macro Structures

Learning Long-Term Dependencies – The LSTM Cell



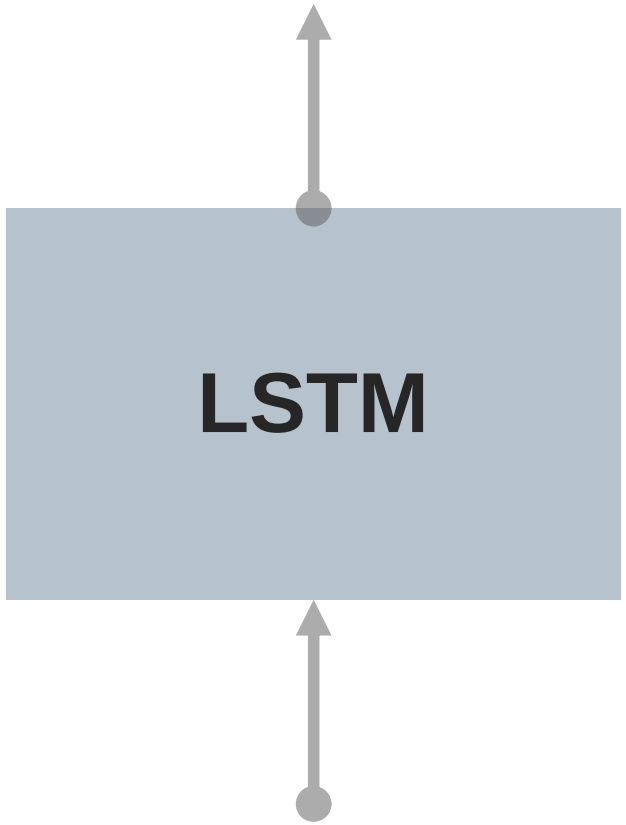
Thinking in Macro Structures

Learning Long-Term Dependencies – The LSTM Cell

```
1  import numpy as np
2  import tensorflow as tf
3
4
5  # Define placeholder for inputs which contain 3 sequences of length 20.
6  input_sequences = tf.keras.Input(shape=(20, 3))
7
8  # Apply a LSTM on the sequences.
9  h = tf.keras.layers.LSTM(
10     units=4, activation=tf.nn.tanh, return_sequences=True)(input_sequences)
11
12 # Generate 10 random sequences which we will feed to the layer.
13 random_sequences = np.random.uniform(
14     0, 1, size=(10, 20, 3)).astype(np.float32)
15 with tf.Session() as session:
16     # We need to initialize the layer parameters first.
17     session.run(tf.global_variables_initializer())
18     # Feed the network with the random sequences.
19     output_feature_maps = session.run(
20         h, feed_dict={input_sequences: random_sequences})
21     print output_feature_maps.shape  # gives: (10, 20, 4)
22
```

Thinking in Macro Structures

Remember the Important Things – And Move On.



\+ Good for modeling long term dependencies in sequential data

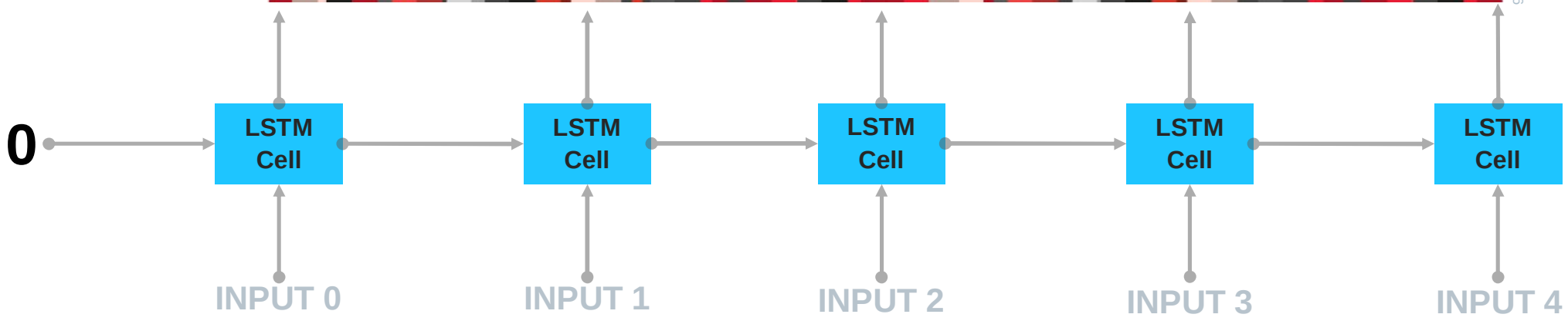
PS: Same accounts for **Gated Recurrent Units**

Very good blog: http://colah.github.io/posts/2015-08-Understanding-LSTMs/

Thinking in Macro Structures

Learning to Focus on the Important Things – Attention

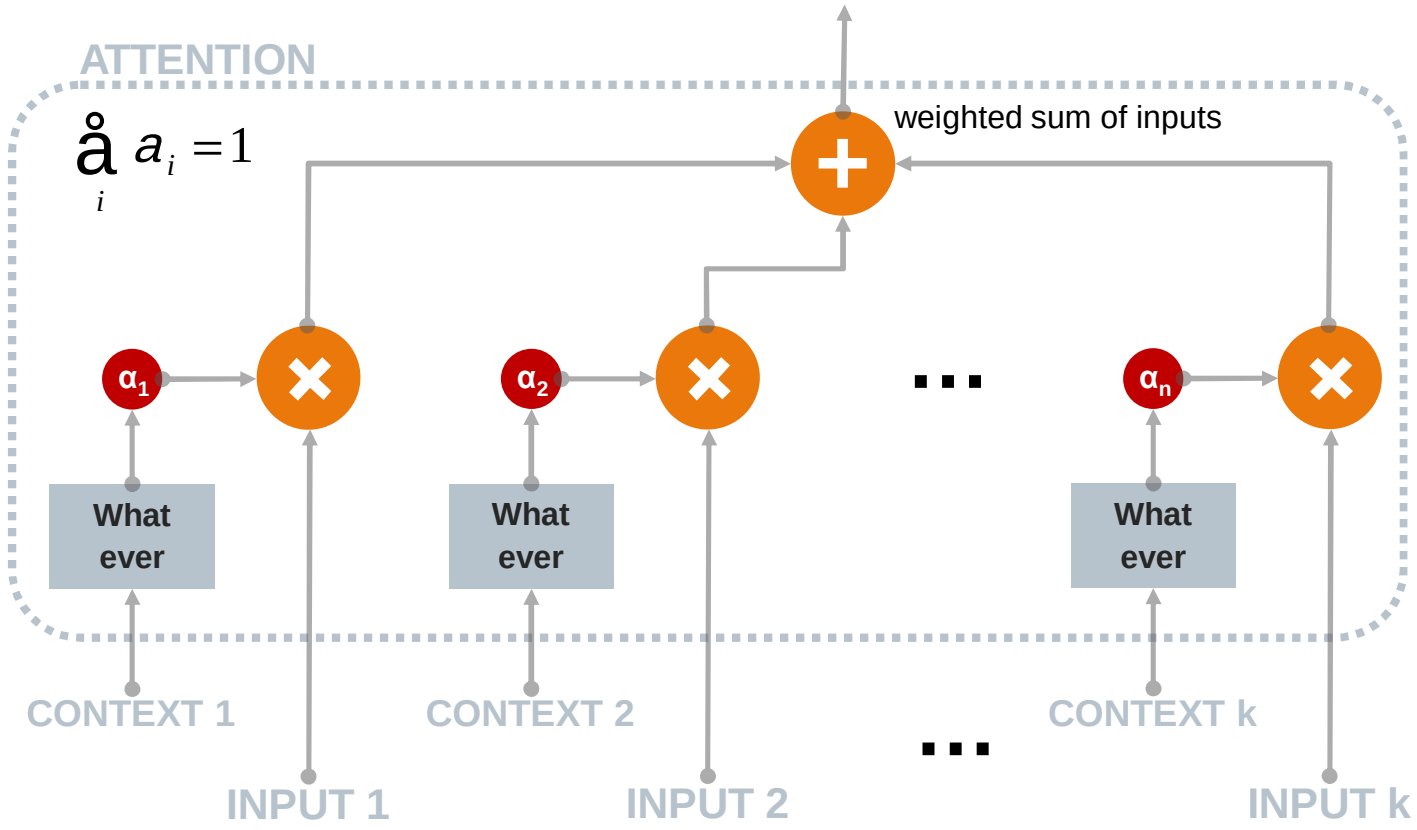
Image: http://www.newsworks.org.uk/Topics-themes/87726



Thinking in Macro Structures

Learning to Focus on the Important Things – Attention

What ever = any function that maps some input to a scalar. Often a multi layer neural network that is learned with the rest.



Thinking in Macro Structures

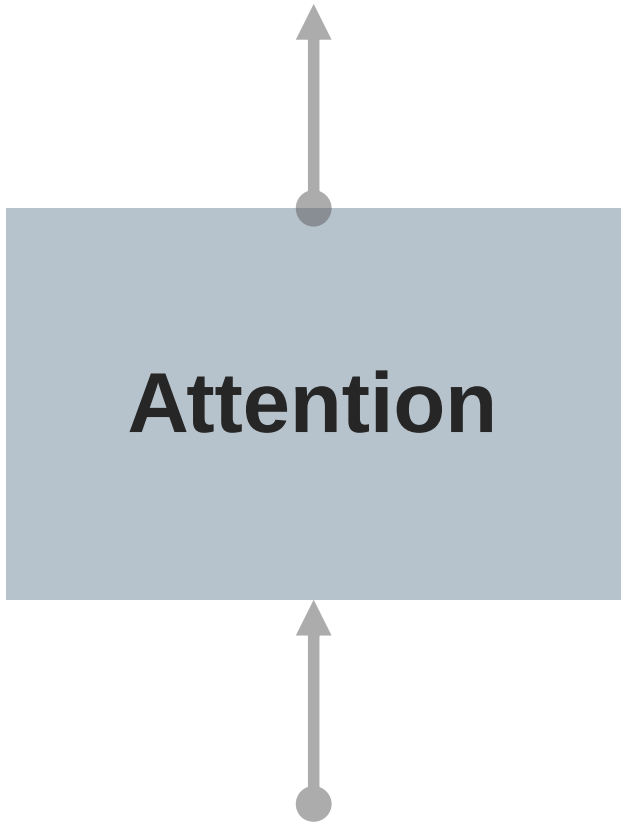
Learning to Focus on the Important Things – Attention

```python
"""
The following implementation of attention is little bit more complicated since
it also works on a batch of examples where we need apply attention on each
example individually and efficiently.

"""
import numpy as np
import tensorflow as tf


# Define placeholder for inputs which contain 3 sequences of length 20.
input_sequences = tf.keras.Input(shape=(20, 3))
context_data = tf.keras.Input(shape=(6,))

# Apply a LSTM on the sequences.
h = tf.keras.layers.LSTM(
    units=4, activation=tf.nn.tanh, return_sequences=True)(input_sequences)

# Repeat the context in order to pair each sequence output of the LSTM with
# the context.
context_repeated = tf.tile(context_data, [1, 20])
context_repeated = tf.reshape(context_repeated, shape=[-1, 20, 6])
# Concatenate the context with outputs of the LSTM.
attention_context = tf.concat([context_repeated, h], axis=-1)

# Flatten the first two dimensions and apply the attention network. Thereby
# the attention network is independently applied on every step of the sequence
# from every example.
attention_context = tf.reshape(attention_context, [-1, 6 + 4])
raw_attention_values = tf.layers.Dense(
    units=1, activation=None)(attention_context)
# Reshape back to get the examples x sequence steps x attention value tensor.
raw_attention_values = tf.reshape(raw_attention_values, [-1, 20, 1])
# Apply the softmax to the attention values along the sequence axis.
attention = tf.nn.softmax(raw_attention_values, axis=1)

# Apply the attention values on the outputs of the LSTM and sum the outputs
# along the sequence axis.
attended = h * attention
attended = tf.reduce_sum(attended, axis=1)

# Generate 10 random sequences which we will feed to the layers.
random_sequences = np.random.uniform(
    0, 1, size=(10, 20, 3)).astype(np.float32)
# Generate also 10 random context vectors.
random_context = np.random.uniform(
    0, 1, size=(10, 6)).astype(np.float32)
with tf.Session() as session:
    # We need to initialize the layer parameters first.
    session.run(tf.global_variables_initializer())
    # Feed the network with the random images.
    attended_states = session.run(
        attended,
        feed_dict={input_sequences: random_sequences,
                   context_data: random_context})
    print attended_states.shape  # gives: (10, 20, 4)

```

Here, the attention mechanism is applied on the outputs of a LSTM to compute a weighted mean of the hidden states computed for each step of the input sequence

Thinking in Macro Structures

Remember the Important Things – And Move On.

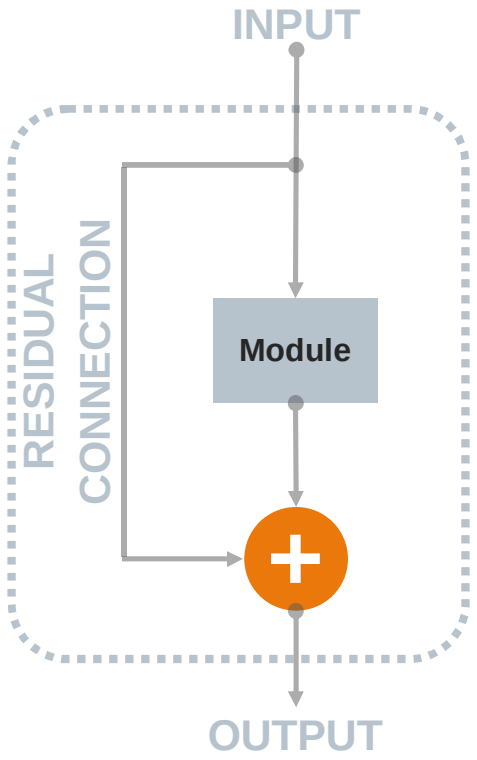


\+ Good for learning a context sensitive selection process

Interactive explanation: http://distill.pub/2016/augmented-rnns/

Residual Units

Module = any differentiable function (e.g. neural network layers) that maps the inputs to some outputs. If the outputs do not have the same shape as the inputs some additional adjustments (e.g. padding) are required.



$$h_l = h_{l-1} + f(h_{l-1})$$

Residual Units

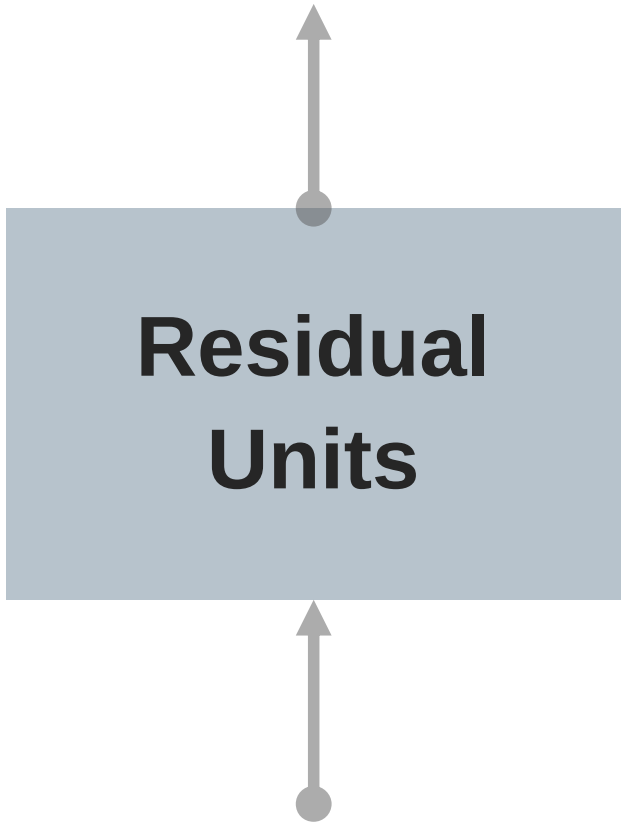
```python
import numpy as np
import tensorflow as tf


# Define placeholder for input data.
input_data = tf.keras.Input(shape=(3,))

# Apply e.g. two dense layers on the input.
h1 = tf.keras.layers.Dense(units=4, activation=tf.nn.relu)(input_data)
h2 = tf.keras.layers.Dense(units=3, activation=tf.nn.relu)(h1)

# Add a residual connection by adding the output of the second dense layer
# and the input data.
h3 = input_data + h2

# Feed some random data through the layer.
random_data = np.random.uniform(0, 1, size=(10, 3)).astype(np.float32)
with tf.Session() as session:
    # We need to initialize the layer parameters first.
    session.run(tf.global_variables_initializer())
    # Feed the network with the random data.
    residual_unit_output = session.run(h3, feed_dict={input_data: random_data})
    print residual_unit_output.shape  # gives: (10, 3)

```

The „module“

The residual connection

Thinking in Macro Structures

Remember the Important Things – And Move On.



\+ Good for training very deep networks

Paper: https://arxiv.org/pdf/1603.05027.pdf

And more…

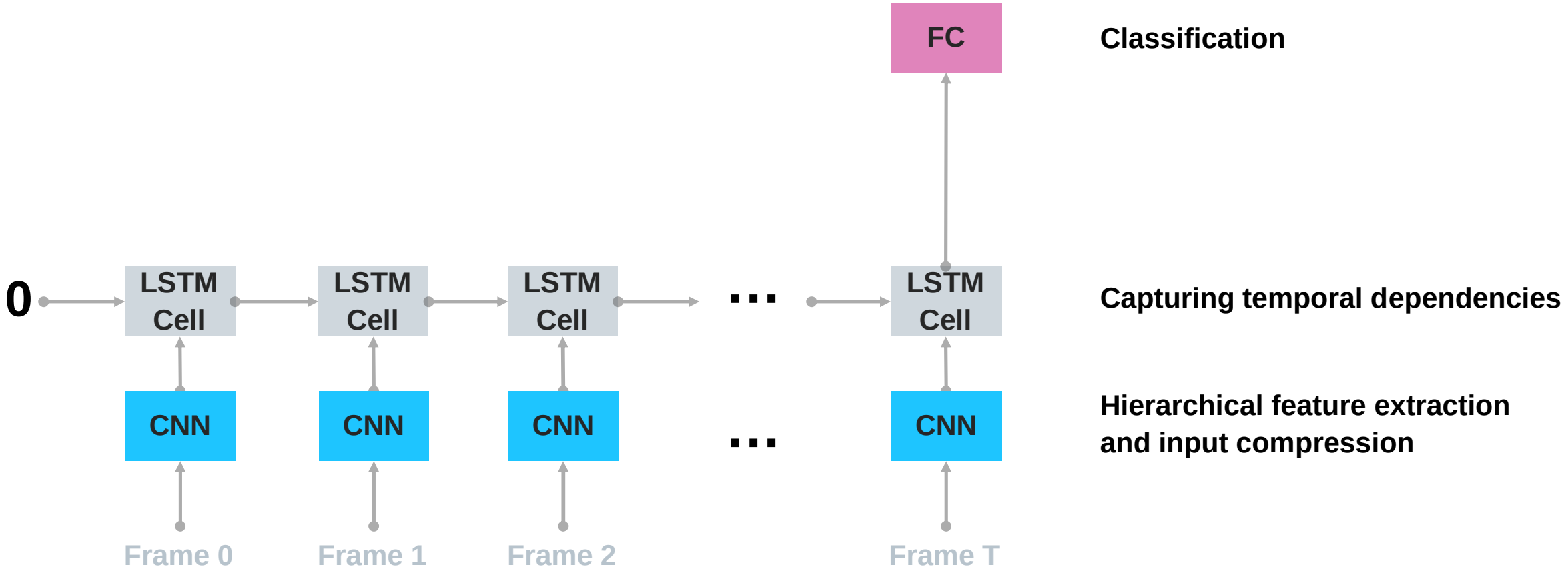
Deep Learning

End-to-End Model Design

End-to-End Model Design

Design Choices - Video Classification Example

[Example for design choices]



End-to-End Model Design

Design Choices - Video Classification Example

[Example for design choices]

```python
import numpy as np
import tensorflow as tf


# Define placeholder for inputs which contain 24 x 24 rgb videos with 20
# frames each.
input_videos = tf.keras.Input(shape=(20, 24, 24, 3))

# Flatten the first two dimensions of the input.
input_frames = tf.reshape(input_videos, [-1, 24, 24, 3])

# Apply a simple CNN on each frame of each video.
h1 = tf.keras.layers.Conv2D(
    filters=16, kernel_size=[3, 3], strides=[1, 1], padding='same',
    activation=tf.nn.relu)(input_frames)
pooled = tf.keras.layers.MaxPool2D(
    pool_size=[2, 2], strides=[2, 2], padding='same')(h1)
h2 = tf.keras.layers.Conv2D(
    filters=16, kernel_size=[3, 3], strides=[1, 1], padding='same',
    activation=tf.nn.relu)(pooled)
pooled = tf.keras.layers.MaxPool2D(
    pool_size=[2, 2], strides=[2, 2], padding='same')(h2)

# Flatten the feature maps and bring back the sequence dimension.
processed_frames = tf.keras.layers.Flatten()(pooled)
processed_videos = tf.reshape(
    processed_frames, [-1, 20, processed_frames.get_shape()[-1]])

# Apply a LSTM on the processed frames.
h = tf.keras.layers.LSTM(
    units=4, activation=tf.nn.tanh, return_sequences=False)(processed_videos)

# Apply a dense layer for a binary classification.
model_output = tf.keras.layers.Dense(1, activation=tf.nn.sigmoid)(h)

# Generate 10 random videos with 20 frames each which we will feed to the
# layer.
random_videos = np.random.uniform(
    0, 1, size=(10, 20, 24, 24, 3)).astype(np.float32)
with tf.Session() as session:
    # We need to initialize the layer parameters first.
    session.run(tf.global_variables_initializer())
    # Feed the network with the random sequences.
    class_probabilities = session.run(
        model_output, feed_dict={input_videos: random_videos})
    print class_probabilities.shape  # gives: (10, 20, 4)

```

CNN

LSTM

Classification

End-to-End Model Design
Real Examples - Deep Face

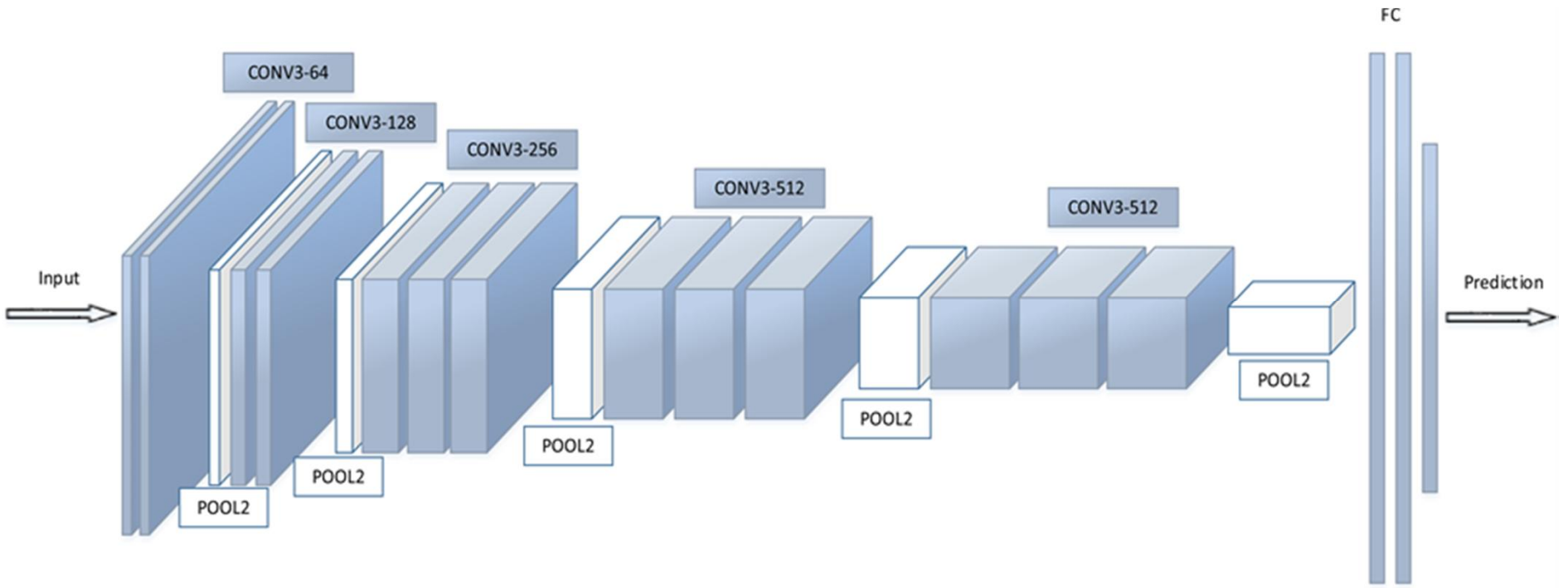


Image:

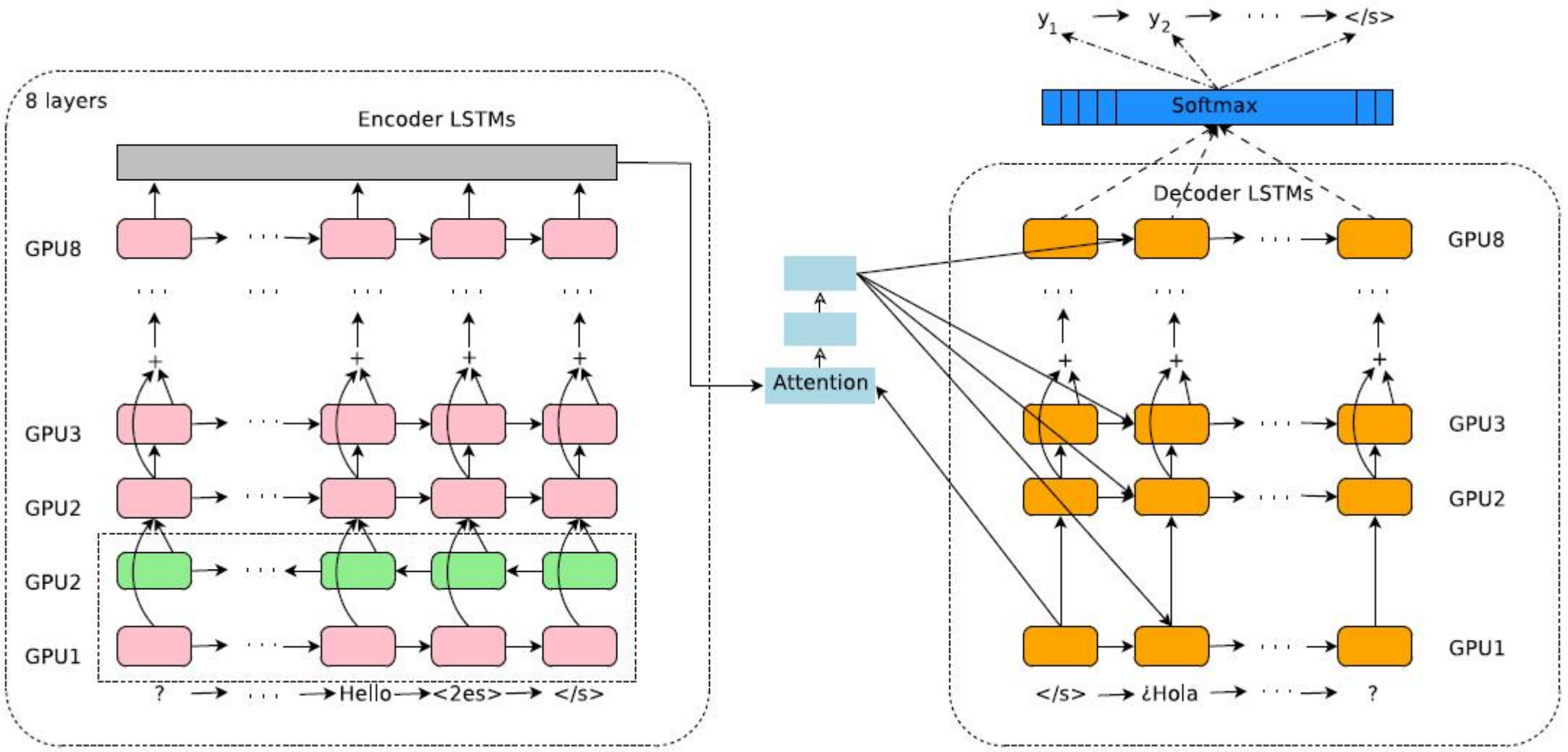
Hachim El Khiyari, Harry Wechsler

Face Recognition across Time Lapse Using Convolutional Neural Networks

Journal of Information Security, 2016.

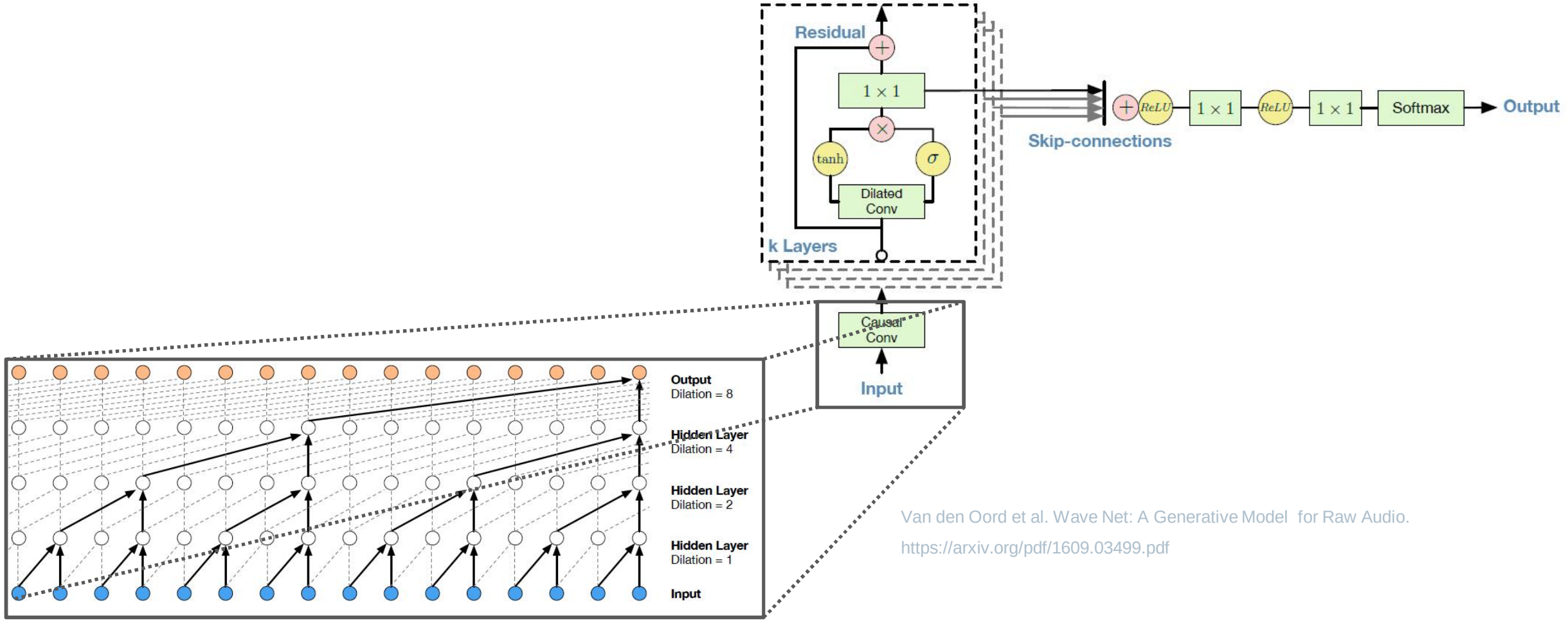
End-to-End Model Design

Real Example - Multi-Lingual Neural Machine Translation



Yonghui Wu, et. al. **Google's Neural Machine Translation System: Bridging the Gap between Human and Machine Translation. https://arxiv.org/abs/1609.08144. 2016**

End-to-End Model Design
Real Examples – Wave Net



Van den Oord et al. Wave Net: A Generative Model for Raw Audio.
https://arxiv.org/pdf/1609.03499.pdf

Deep Learning

Deep Learning Model Training

Basic Implementation Overview

1. **Prepare the data**
2. Design the model architecture
3. Design the optimization goals
4. Train
5. Evaluate
6. Deploy

```python
def load_and_preprocess_data():
    """Talk to domain experts, read related work."""
    raise NotImplementedError()


def build_model(data):
    """Read related work, use intuition and experience."""
    raise NotImplementedError()


def build_update_op(model_outputs, data):
    """Read related work, use intuition and experience."""
    raise NotImplementedError()


def train_model(model_inputs, model_targets, update_op, data):
    """Trial and error."""
    raise NotImplementedError()


# Load the data.
data = load_and_preprocess_data()

# Build the model architecture.
model_inputs, model_outputs = build_model(data)

# Build the update operation.
update_op, model_targets = build_update_op(model_outputs, data)

# Train the model on the data.
train_model(model_inputs, model_targets, update_op, data)
```

Implementing Loss Function and Optimizer

```python
import numpy as np
import tensorflow as tf


# Define placeholder for input images.
input_images = tf.keras.Input(shape=(24, 24, 3))

# Build the network.
h1 = tf.keras.layers.Conv2D(
    filters=16, kernel_size=[3, 3], strides=[1, 1], padding='same',
    activation=tf.nn.relu)(input_images)
h2 = tf.keras.layers.Conv2D(
    filters=16, kernel_size=[3, 3], strides=[1, 1], padding='same',
    activation=tf.nn.relu)(h1)
pooled = tf.keras.layers.MaxPool2D(
    pool_size=[2, 2], strides=[2, 2], padding='same')(h2)
flattened = tf.keras.layers.Flatten()(pooled)
h3 = tf.keras.layers.Dense(units=100, activation=tf.nn.relu)(flattened)
output = tf.keras.layers.Dense(
    units=10, activation=tf.nn.softmax)(h3)

# Build and compile the model.
model = tf.keras.Model(inputs=input_images, outputs=output)
model.compile(
    tf.train.AdamOptimizer(),
    loss=tf.keras.losses.sparse_categorical_crossentropy)

# Generate some random data and train the model.
random_images = np.random.uniform(
    0, 1, size=(1000, 24, 24, 3)).astype(np.float32)
random_classes = np.random.randint(10, size=(1000,)).astype(np.int32)
model.fit(x=random_images, y=random_classes, batch_size=16, epochs=10)

```

Trivial to implement, but you must know what you need

Training Deep Learning Models

Loss Function Design

- Basic Loss functions
- Multi-Task Learning

Optimization

- Optimization in Deep Learning
- Work-horse Stochastic Gradient Descent
- Adaptive Learning Rates

Regularization

- Weight Decay
- Early Stopping
- Dropout
- Batch Normalization

Distributed Training

- Not covered, but I included a link to a good overview.

Deep Learning

Loss Function Design

Loss Function Design
Regression

Mean Squared Loss

$$f^{R}_{loss}(Y, X, q) = \frac{1}{n} \sum_{i}^{n} (y_i - f_q(x_i))^2$$

Network output can be anything:

Ø Use no activation function in output layer!

Example ID	Target Value (y_i)	Prediction ($f_\theta(x_i)$)	Example Error
1	4.2	4.1	0.01
2	2.4	0.4	4
3	-2.9	-1.4	2.25
…	…	…	…
n	0	1.0	1.0

Loss Function Design
Binary Classification

Binary Cross Entropy (also called Log Loss)

$$f_{loss}^{BC}(Y, X, q) = \frac{1}{n} \sum_{i}^{n} - [y_i \times \log(f_q(x_i)) + (1 - y_i) \times \log(1 - f_q(x_i))]$$

Network output needs to be between 0 and 1:

- Use sigmoid activation function in the output layer!
- Note: Sometimes there are optimized functions available that operate on the raw outputs (logits)

Example ID	Target Value (y_i)	Prediction ($f_\theta(x_i)$)	Example Error
1	0	0.211	0.237
2	1	0.981	0.019
3	0	0.723	1.284
…	…	…	…
n	0	0.134	0.144

Loss Function Design
Multi-Class Classification

Categorical Cross Entropy

$$f_{loss}^{MCC}(Y, X, q) = \frac{1}{n} \sum_{i}^{n} \sum_{j}^{c} - y_{i,j} \times \log(f_q(x_i)_{i,j})$$

Network output needs to represent a probability distribution over c classes: $\sum_{j}^{c} f_q(x_i)_{i,j} = 1$

- Use softmax activation function in the output layer!
- Note: Sometimes there are optimized functions available that operate on the raw outputs (logits)

Example ID	Target Value (y_i)	Prediction ($f_\theta(x_i)$)	Example Error
1	[0, 0, 1]	[0.2, 0.2, 0.6]	0.511
2	[1, 0, 0]	[0.3, 0.5, 0.2]	1.20
3	[0, 1, 0]	[0.1, 0.7, 0.3]	0.511
…	…	…	…
n	[0, 0, 1]	[0.0, 0.01, 0.99]	0.01

Loss Function Design
Multi-Label Classification

Multi-Label classification loss function (Just sum of Log Loss for each class)

$$f_{loss}^{MLC}(Y, X, q) = -\frac{1}{n}\sum_{i}^{n}\sum_{j}^{c} y_{i,j} \times \log(f_q(x_i)_{i,j}) + (1 - y_{i,j}) \times \log(1 - f_q(x_i)_{i,j})$$

Each network output needs to be between 0 and 1:

- Use sigmoid activation function on each network output!
- Note: Sometimes there are optimized functions available that operate on the raw outputs (logits)

Example ID	Target Value (y_i)	Prediction ($f_\theta(x_i)$)	Example Error
1	[0, 0, 1]	[0.2, 0.4, 0.6]	1.245
2	[1, 0, 1]	[0.3, 0.9, 0.2]	5.116
3	[0, 1, 0]	[0.1, 0.7, 0.1]	0.567
…	…	…	…
n	[1, 1, 1]	[0.8, 0.9, 0.99]	0.339

Loss Function Design
Multi-Task Learning

Additive Cost Function

$$f_{loss}^{MT}([Y_0,\ldots,Y_K],[X_0,\ldots,X_K],q) = \sum_k \lambda_k f_{loss_k}(Y_k, X_k, q)$$

Each network output has associated input and target data and an associated loss metric:

- Use proper output activation for each of the k output layer!
- The weighting λ_k of each task in the cost function is derived from prior knowledge/assumptions or by trial and error.
- Note that we could learn multiple tasks from the same data. This can be represented by copies of the corresponding data in the formula above. When implementing this, we would of course not copy the data.

Examples:

- Auxiliary heads for counteracting vanishing gradient (Google LeNet, https://arxiv.org/abs/1409.4842)
- Artistic style transfer (Neural Artistic Style Transfer, https://arxiv.org/abs/1508.06576)
- Instance segmentation (Mask RNN, https://arxiv.org/abs/1703.06870)
- …

Deep Learning

Optimization

Optimization
Learning the Right Parameters in Deep Learning

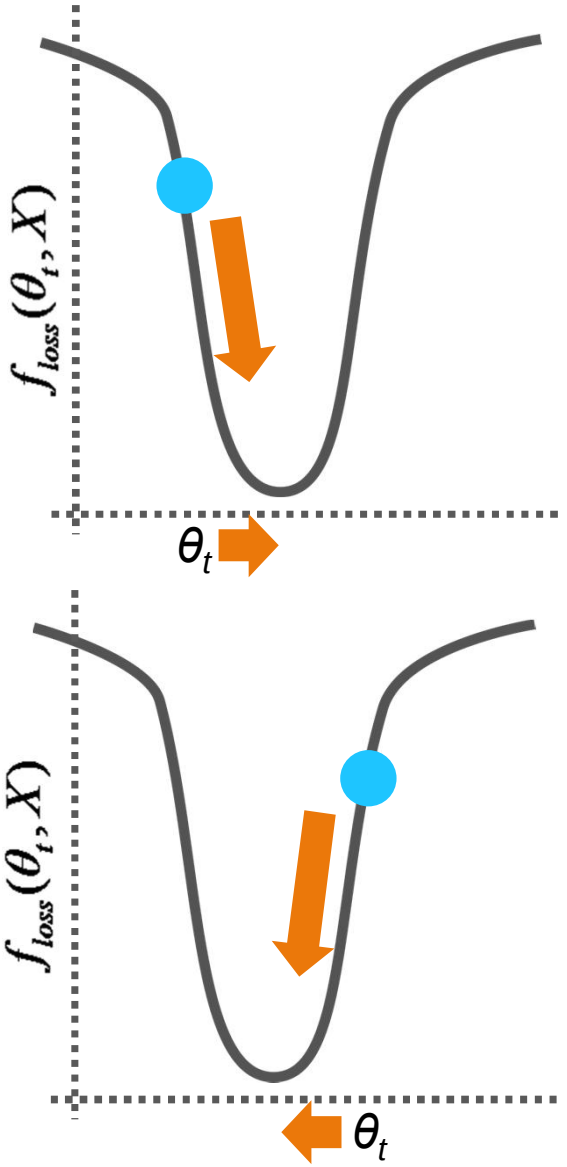
- Neural networks are composed of differentiable building blocks
- Training a neural network means minimization of some non-convex differentiable loss function using iterative gradient-based optimization methods
- The simplest but mostly used optimization algorithm is “gradient descent”

Optimization
Gradient Descent

Negative Gradient

You can think of the gradient as the local slope with respect to each parameter θ_i at step t.

Positive Gradient



We update the parameters a little bit in the direction where the error gets smaller

$$q_t = q_{t-1} - h \cdot g_t$$

Gradient with respect to the model parameters θ

$$\text{with } g_t = \tilde{\nabla}_q f_{loss}(Y, X, q_{t-1})$$

Optimization
Work-Horse Stochastic Gradient Descent

Stochastic Gradient Descent is Gradient Descent on samples (Mini-Batches) of data:

- Increases variance in the gradients
  - ØSupposedly helps to jump out of local minima

- But essentially, it is just super efficient and it works!

We update the parameters a little bit in the direction where the error gets smaller

$$q_t = q_{t-1} - h \times g_t^{(s)}$$

Gradient with respect to the model parameters θ

$$\text{with } g_t^{(s)} = \tilde{\nabla}_q f_{loss}(Y^{(s)}, X^{(s)}, q_{t-1})$$

In the following we will omit the superscript s and X will always represent a mini-batch of samples from the data.

Optimization
Computing the Gradient

I have to compute the gradient of that???

Sounds complicated!

$$q_t = q_{t-1} - h \rangle g_t$$

$$\text{with } g_t = \tilde{\mathrm{N}}_q f_{loss}(Y, X, q_{t-1})$$

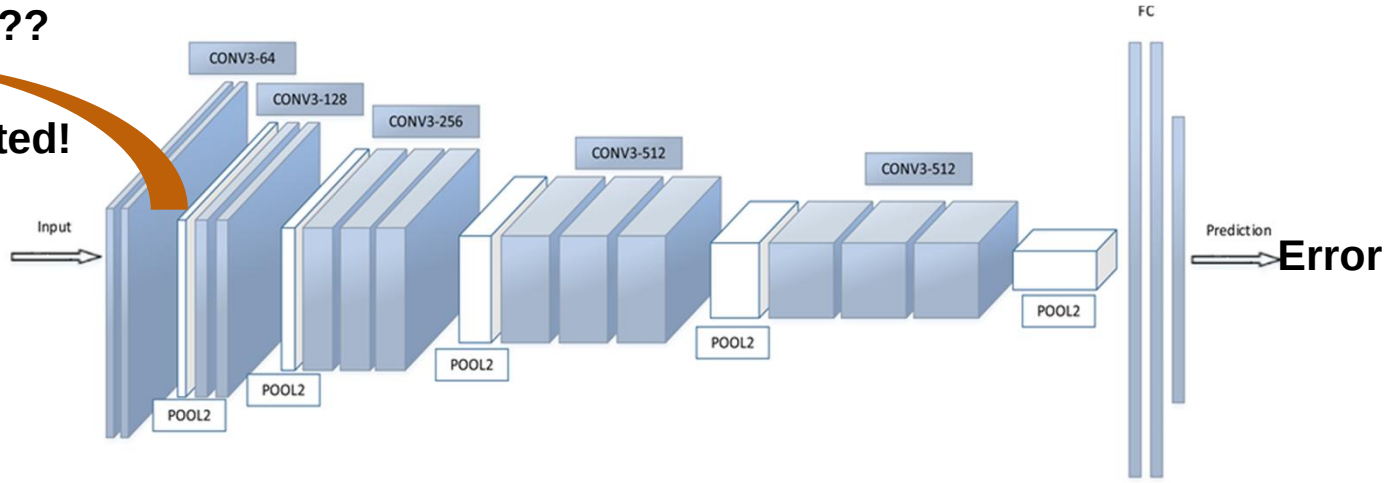


Image:

Hachim El Khiyari, Harry Wechsler

Face Recognition across Time Lapse Using Convolutional Neural Networks

Journal of Information Security, 2016.

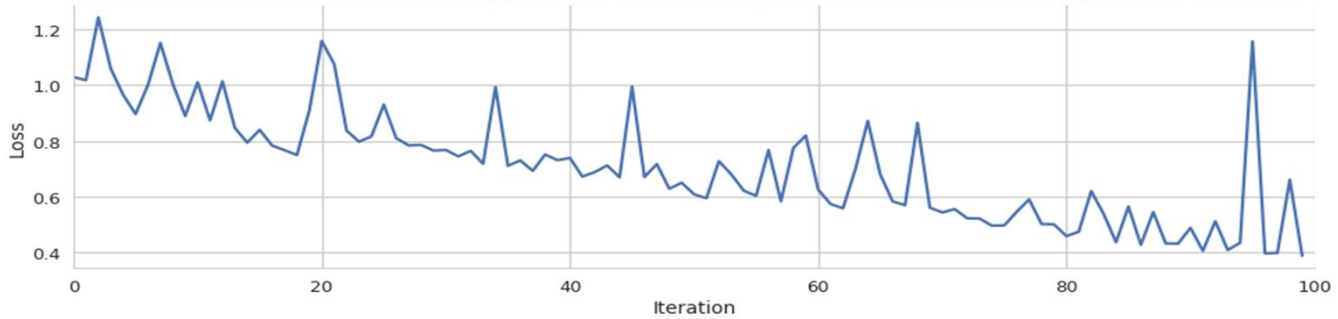
Optimization
Automatic Differentiation

$$q_t = q_{t-1} - h \cdot g_t$$

$$\text{with } g_t = \tilde{\nabla}_q f_{loss}(Y, X, q_{t-1})$$

In [10]:
```
# Define a loss function.
Y = tf.placeholder(tf.int32, [None])  # The labels.
losses = tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels=Y, logits=raw_network_output)
loss = tf.reduce_mean(losses)

# Computing the gradient.
network_parameters = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES)
grads = tf.gradients(loss, network_parameters)
```

In [21]:
```
# SGD with mini-batches of size 4 for 100 iterations.
random_images = np.random.rand(8, 10, 10, 3).astype(np.float32)
random_labels = np.random.randint(0, 3, size=(8,)).astype(np.int32)
for i in xrange(100):
    # Sample a mini-batch.
    mbatch_indices = np.random.choice(np.arange(8), 4)
    mbatch_images = random_images[mbatch_indices]
    mbatch_labels = random_labels[mbatch_indices]

    # Apply SGD update rule with constant learning rate.
    for w, g in zip(network_parameters, grads):
        session.run(tf.assign(w, w - 0.01 * g),
                    feed_dict={X: mbatch_images, Y: mbatch_labels})
```



AUTOMATIC DIFFERENTIATION

IS AN

EXTREMELY POWERFUL FEATURE

FOR DEVELOPING MODELS WITH

DIFFERENTIABLE OPTIMIZATION OBJECTIVES

Backpropagation is just a fancy name for applying the chain rule to compute the gradients in neural networks!

Optimization

Stochastic Gradient Descent – Problems with Constant Learning Rates

Small gradient

1

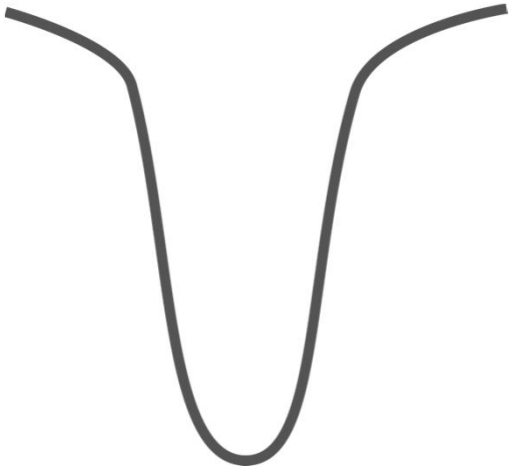
Flat gradient

3

$$q_t = q_{t-1} - h \times g_t$$

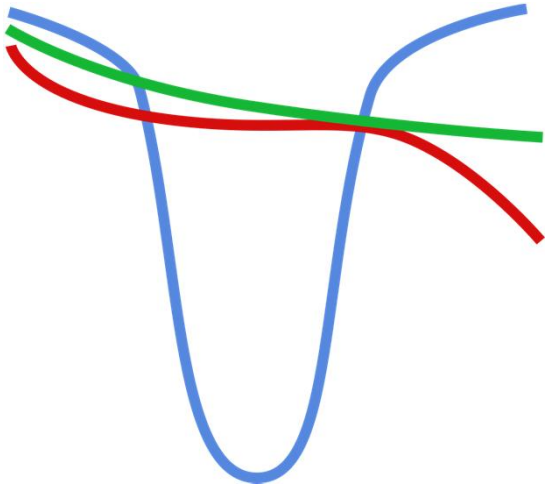
Steep gradient

2



Gradients of different parameters vary

4



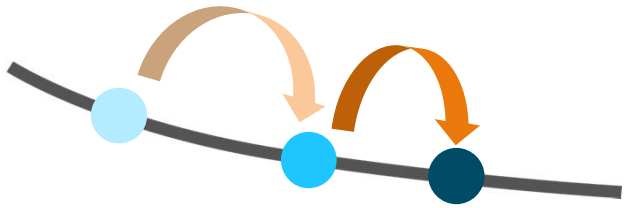
Excellent Overview and Explanation: http://sebastianruder.com/optimizing-gradient-descent/

Optimization

Stochastic Gradient Descent – Problems with Constant Learning Rates

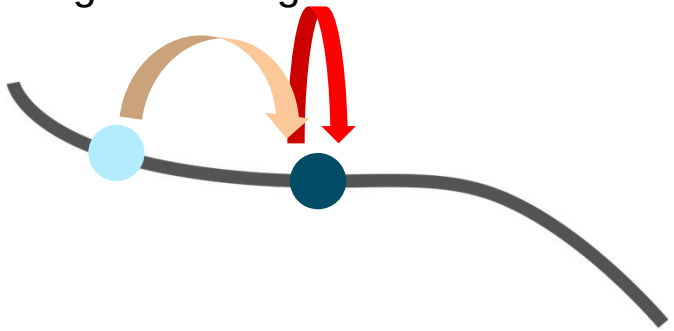
Learning rate too small

✖ 1



Get stuck in zero gradient regions

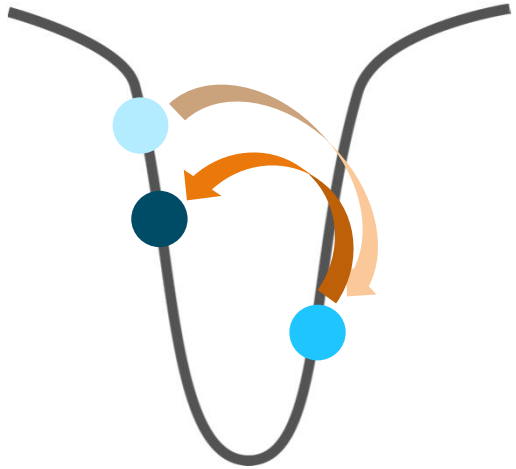
✖ 3



$$q_t = q_{t-1} - h \times g_t$$

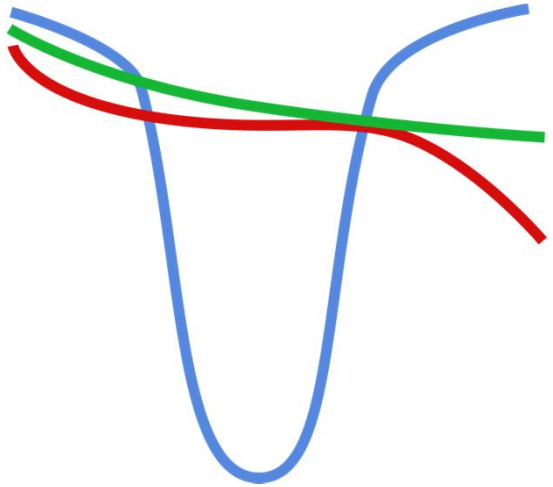
Learning rate too large

✖ 2



Learning rate can be parameter specific

✖ 4

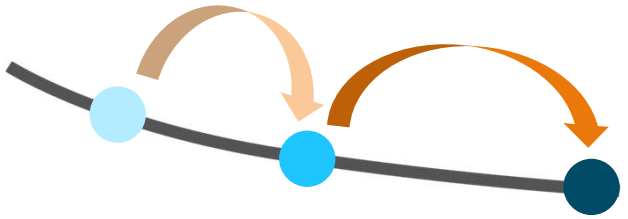


Excellent Overview and Explanation: http://sebastianruder.com/optimizing-gradient-descent/

Optimization
Stochastic Gradient Descent – Adding Momentum

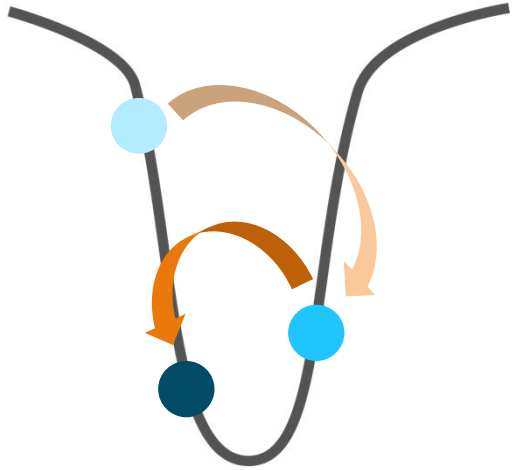
Step size can accumulates momentum if successive gradients have same direction

√ 1



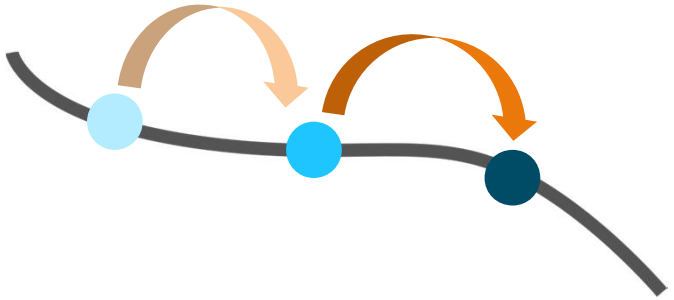
Step size decreases fast if the direction of the gradients changes

√ 2



Momentum only decays slowly and does not stop immediately

√ 3



Adding the previous step size can lead to acceleration

Decay ("friction") Constant learning rate

$$v_t = l\, v_{t-1} + h g_t$$

$$q_t = q_{t-1} - v_t$$

$$\text{with } g_t = \tilde{N}_q f_{loss}(Y, X, q_{t-1})$$

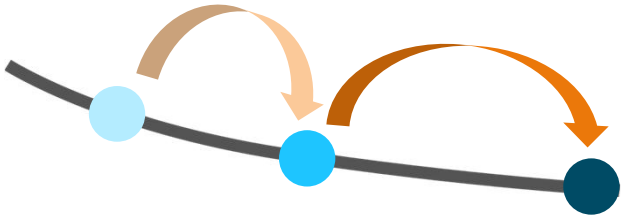
Excellent Overview and Explanation: http://sebastianruder.com/optimizing-gradient-descent/

Optimization

Stochastic Gradient Descent – Adaptive Learning Rate (RMS Prop)

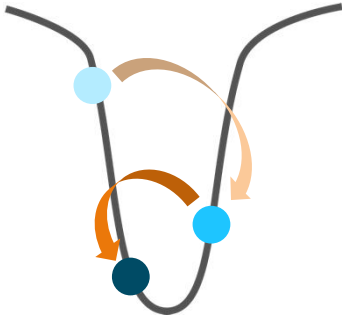
Continuously low gradient will increase the learning rate

√ 1



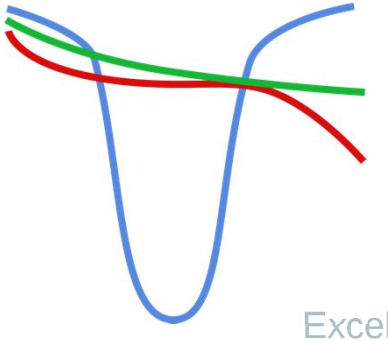
Continuously large gradients will result in a decrease of the learning rate

√ 2



For each parameter an individual learning rate is computed

√ 4



Update rule with an individual learning rate for each parameter θ_i

$$q_{t,i} = q_{t-1,i} - h_i¢ \times g_{t,i}$$

The learning rate is adapted by a decaying mean of past updates

$$E[g_i^2]_t = b \times E[g_i^2]_{t-1} - (1 - b) \times g_{t,i}^2$$

The correction of the (constant) learning rate for each parameter.
The epsilon is only for numerical stability

$$h_i¢ = \frac{h}{\sqrt{E[g_i^2]_t + e}}$$

Excellent Overview and Explanation: http://sebastianruder.com/optimizing-gradient-descent/

Optimization

Stochastic Gradient Descent – Overview Common Step Rules

	Constant Learning Rate	Constant Learning Rate with Annealing	Momentum	Nesterov	AdaDelta	RMSProp	RMSProp + Momentum	ADAM
1	✗	✗	√	√	√	√	√	√
2	✗	√	√	√	√	√	√	√
3	✗	✗	√	√	✗	✗	√	√
4	✗	✗	✗	✗	√	√	√	√

This does not mean that it cannot make sense to use only a constant learning rate!

Excellent Overview and Explanation: http://sebastianruder.com/optimizing-gradient-descent/

Deep Learning

Regularization

Optimization

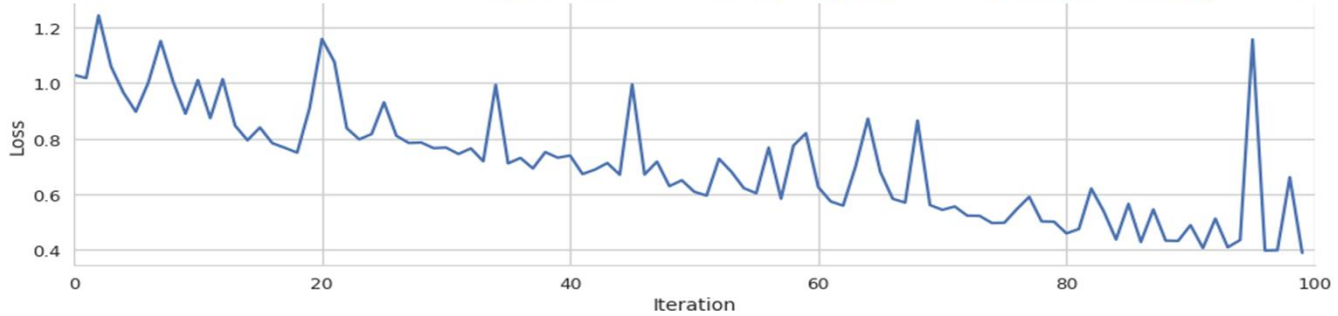
Something feels terribly wrong here, can you see it?

$$q_t = q_{t-1} - h \cdot g_t$$

$$\text{with } g_t = \tilde{\mathrm{N}}_q f_{loss}(Y, X, q_{t-1})$$

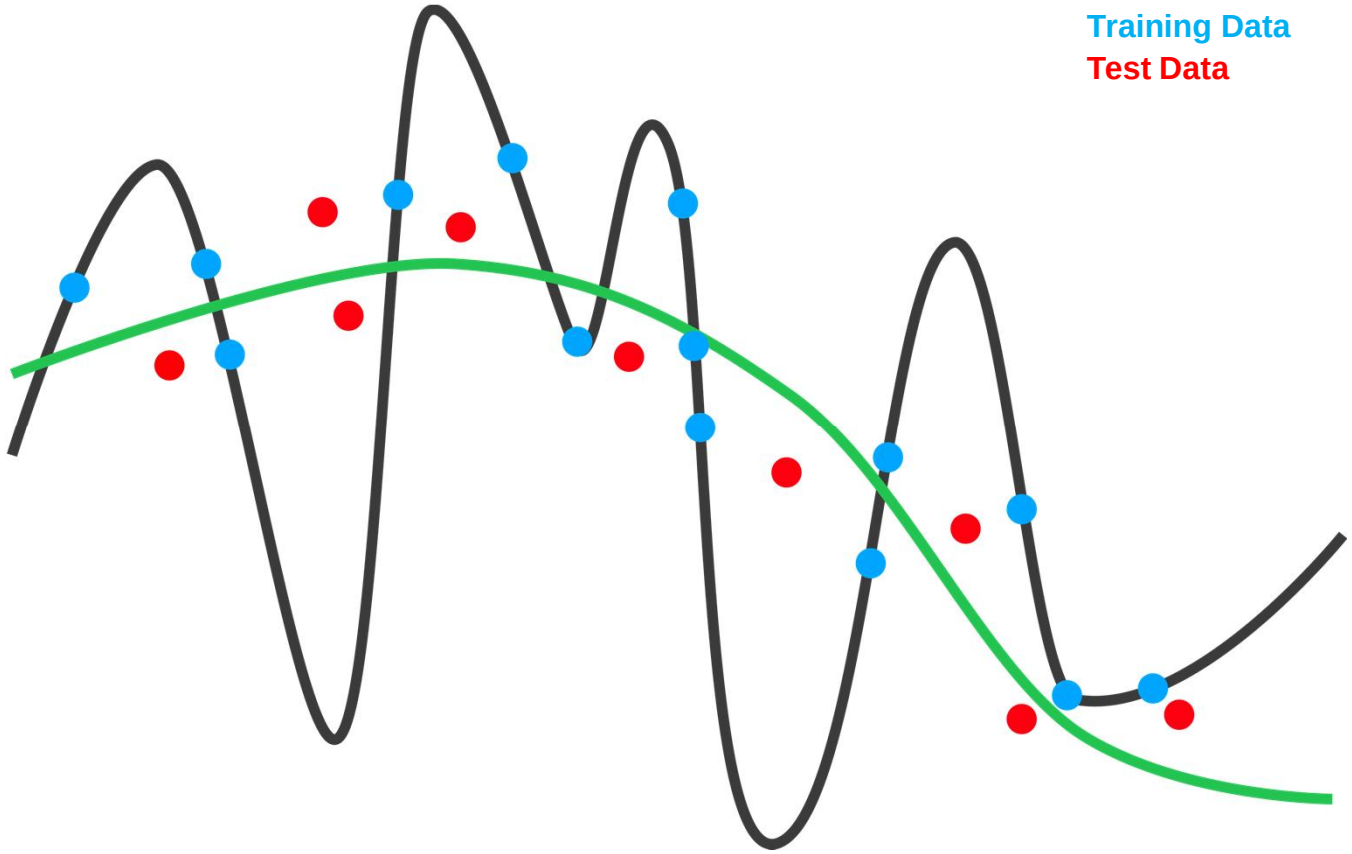
In [10]:
```
# Define a loss function.
Y = tf.placeholder(tf.int32, [None])  # The labels.
losses = tf.nn.sparse_softmax_cross_entropy_with_logits(
    labels=Y, logits=raw_network_output)
loss = tf.reduce_mean(losses)

# Computing the gradient.
network_parameters = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES)
grads = tf.gradients(loss, network_parameters)
```

In [21]:
```
# SGD with mini-batches of size 4 for 100 iterations.
random_images = np.random.rand(8, 10, 10, 3).astype(np.float32)
random_labels = np.random.randint(0, 3, size=(8,)).astype(np.int32)
for i in xrange(100):
    # Sample a mini-batch.
    mbatch_indices = np.random.choice(np.arange(8), 4)
    mbatch_images = random_images[mbatch_indices]
    mbatch_labels = random_labels[mbatch_indices]

    # Apply SGD update rule with constant learning rate.
    for w, g in zip(network_parameters, grads):
        session.run(tf.assign(w, w - 0.01 * g),
                    feed_dict={X: mbatch_images, Y: mbatch_labels})
```



Regularization
Why Regularization is Important

- The goal of learning is not to find a solution that explain the training data perfectly.
- The goal of learning is to find a solution that generalizes well on unseen data points.
- Regularization tries to prevent the model to just fit the training data in an arbitrary way (overfitting).



Regularization
Weight Decay – Constraining Parameter Values

Intuition:

- Discourage the model for choosing undesired values for parameters during learning.

General Approach:

- Putting prior assumptions on the weights. Deviations from these assumptions get penalized.

Examples:

L2 –Regularization (Squared L2 norm or Gaussian Prior)

$$\|q\|_2^2 = \sum_{i,j} (q_{i,j})^2$$

L1-Regularization

$$\|q\|_1 = \sum_{i,j} |q_{i,j}|$$

The regularization term is just added to the cost function for the training.

$$f_{loss}^{total}(Y, X, q) = f_{loss}(Y, X, q) + l \times \|q\|_2^2$$

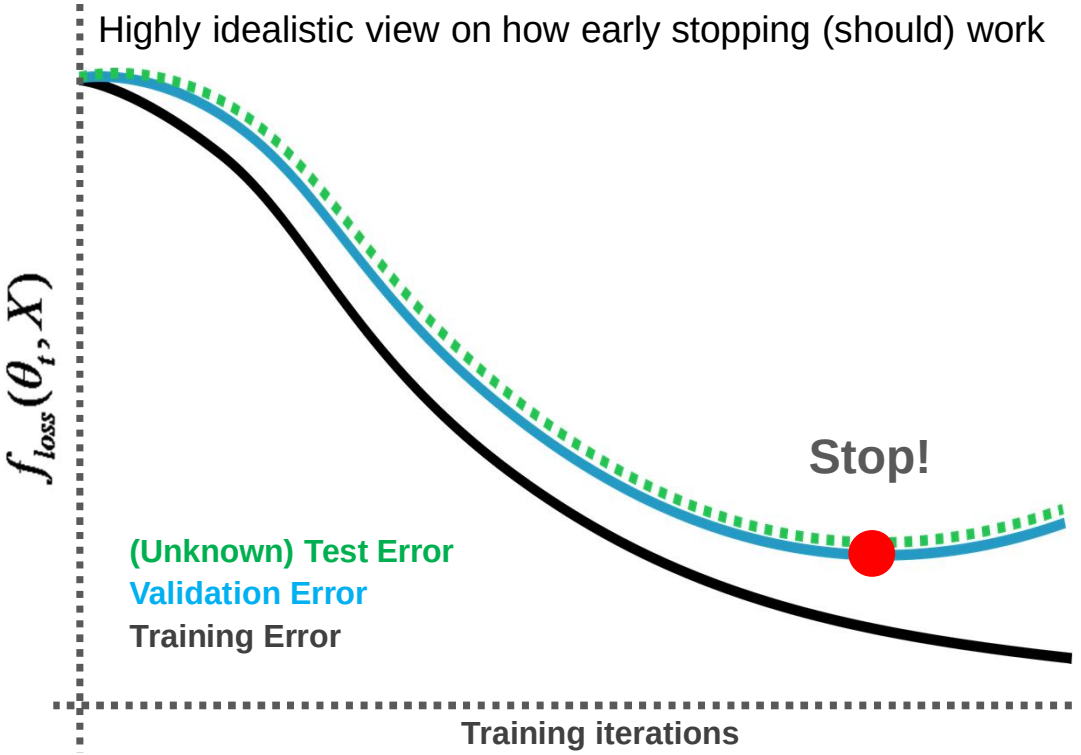
λ is a tuning parameter that determines how strong the regularization affects learning.

Regularization
Early Stopping – Stop Training Just in Time.

Problem
- There might be a point during training where the model starts to overfit the training data at the cost of generalization.

Approach
- Separate additional data from the training data and consistently monitor the error on this validation dataset.
- Stop the training if the error on this dataset does not improve or gets worse over a certain amount of training iterations.

- It is assumed that the validation set approximates the models generalization error (on the test data).

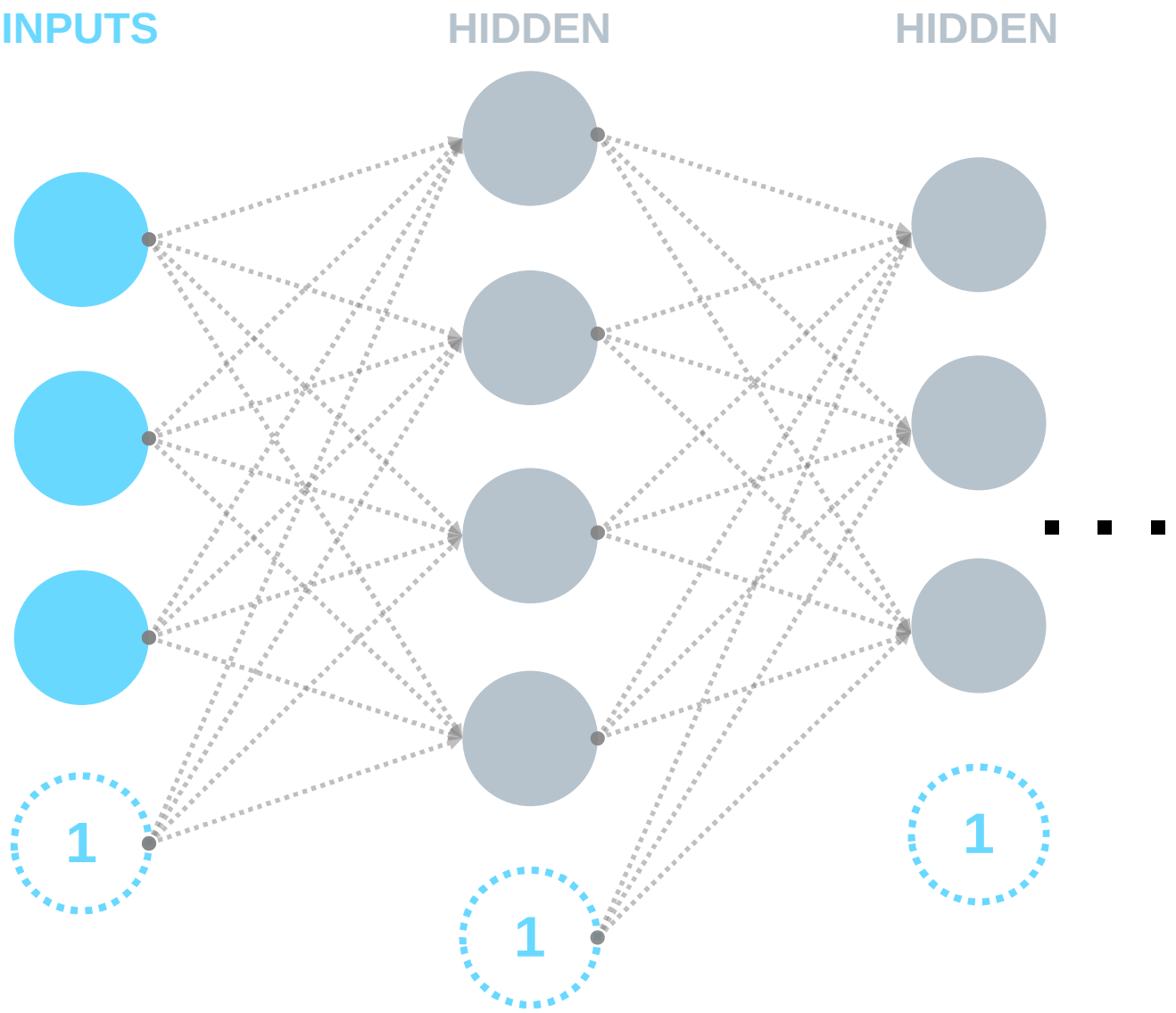


Regularization
Dropout – Make Nodes Expendable

Problem
- Deep learning models are often highly over parameterized which allows the model to overfit on or even memorize the training data.

Approach
- Randomly set output neurons to zero
  - Ø Transforms the network into an ensemble with an exponential set of weaker learners whose parameters are shared.

Usage
- Primarily used in fully connected layers because of the large number of parameters
- Rarely used in convolutional layers
- Rarely used in recurrent neural networks (if at all between the hidden state and output)

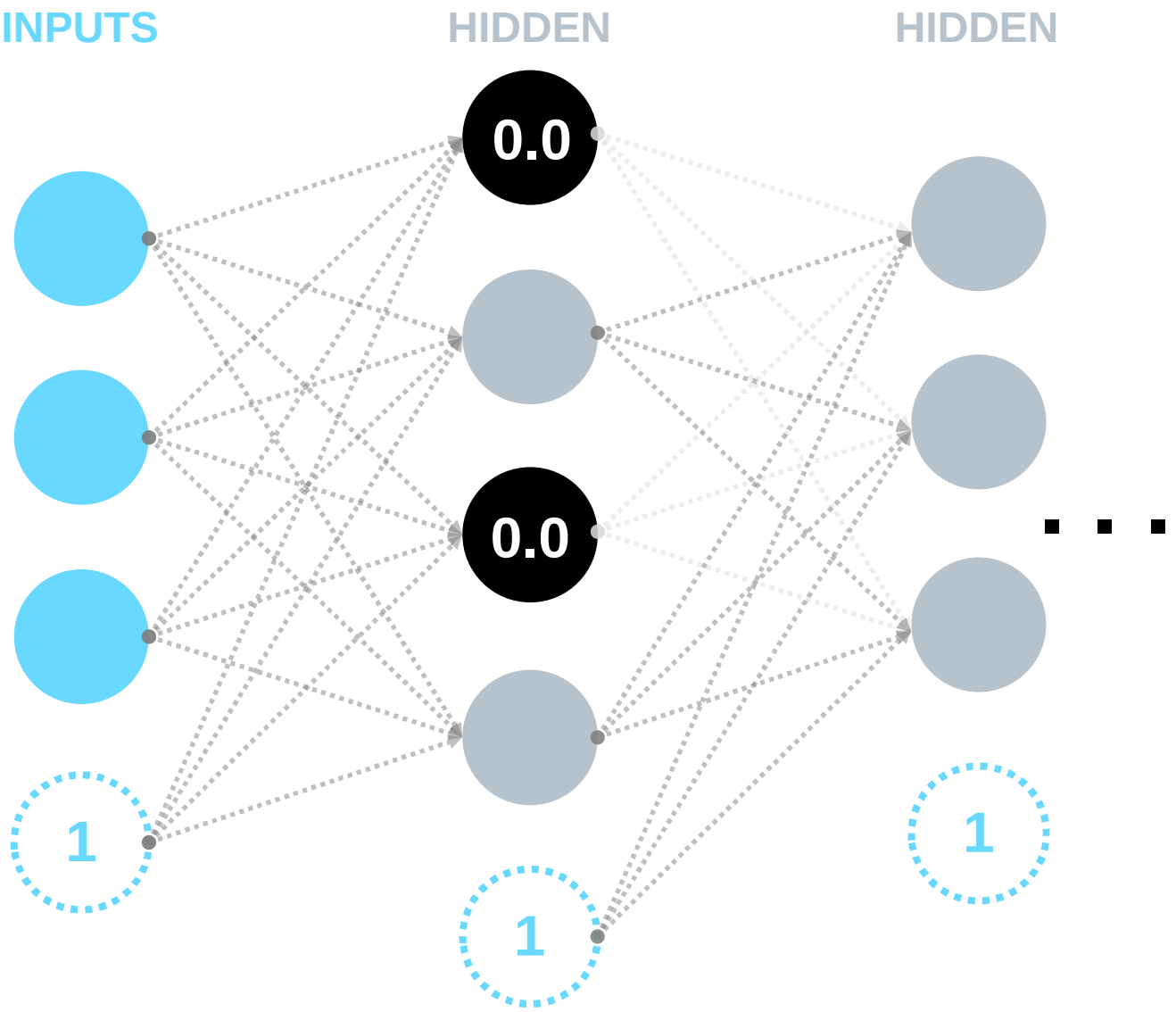


Regularization
Dropout – Make Nodes Expendable

Problem
- Deep learning models are often highly over parameterized which allows the model to overfit on or even memorize the training data.

Approach
- Randomly set output neurons to zero
  - Ø Transforms the network into an ensemble with an exponential set of weaker learners whose parameters are shared.

Usage
- Primarily used in fully connected layers because of the large number of parameters
- Rarely used in convolutional layers
- Rarely used in recurrent neural networks (if at all between the hidden state and output)

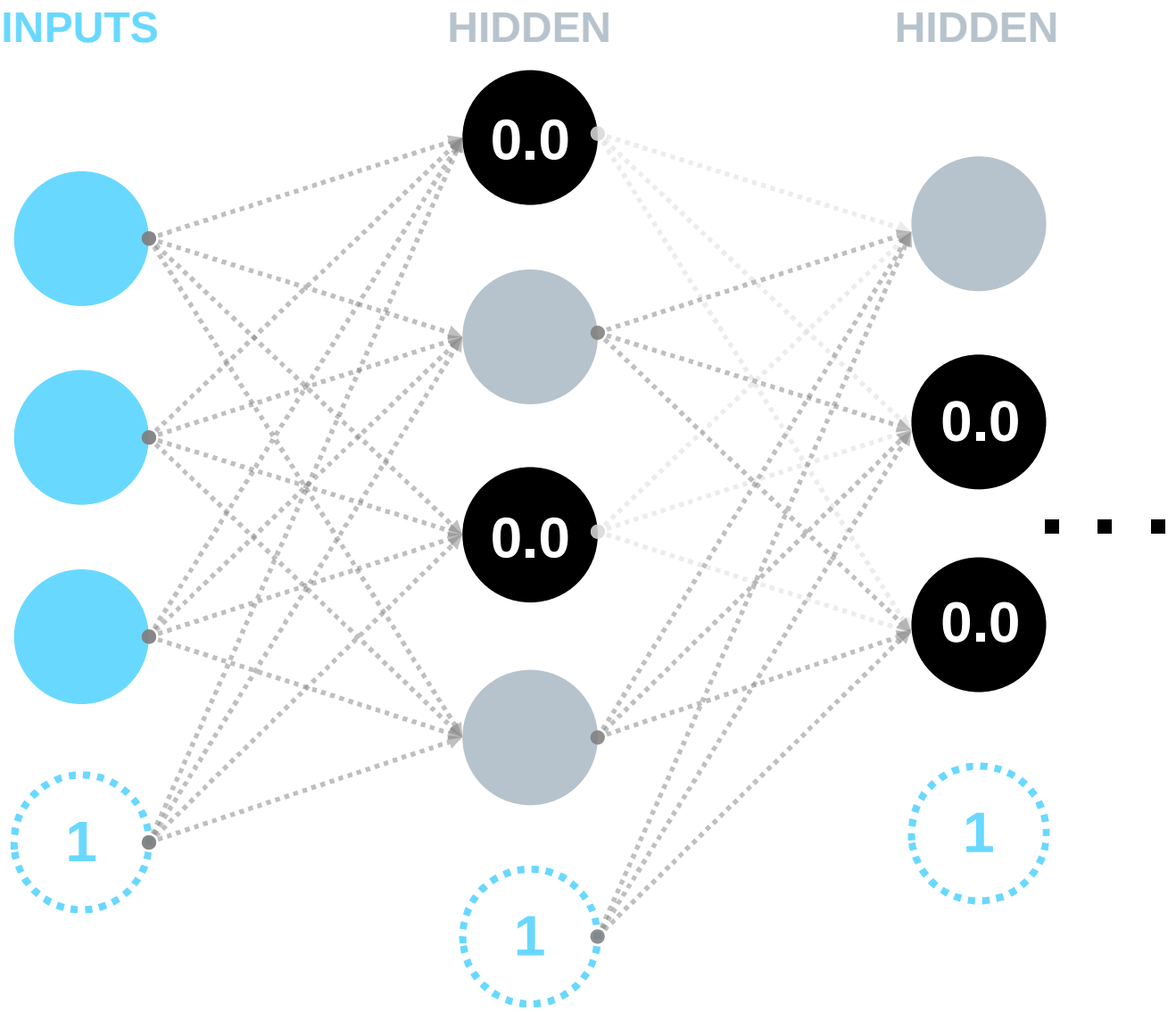


Regularization
Dropout – Make Nodes Expendable

Problem
- Deep learning models are often highly over parameterized which allows the model to overfit on or even memorize the training data.

Approach
- Randomly set output neurons to zero
  - Ø Transforms the network into an ensemble with an exponential set of weaker learners whose parameters are shared.

Usage
- Primarily used in fully connected layers because of the large number of parameters
- Rarely used in convolutional layers
- Rarely used in recurrent neural networks (if at all between the hidden state and output)



Regularization
Batch Normalization – Avoiding Covariate Shift

Problem
- Deep neural networks suffer from internal covariate shift which makes training harder.

Approach
- Normalize the inputs of each layer (zero mean, unit variance)
  - ∅Regularizes because the training network is no longer producing deterministic values in each layer for a given training example

Usage
- Can be used with all layers (FC, RNN, Conv)
- With Convolutional layers, the mini-batch statistics are computed from all patches in the mini-batch.

Normalize the input X of layer k by the mini-batch moments:

$$\hat{X}^{(k)} = \frac{X^{(k)} - m_X^{(k)}}{s_X^{(k)}}$$

The next step gives the model the freedom of learning to undo the normalization if needed:

$$\tilde{X}^{(k)} = g^{(k)} \hat{X}^{(k)} + b^{(k)}$$

The above two steps in one formula.

$$\tilde{X}^{(k)} = g^{(k)} \times \frac{X^{(k)}}{s_X^{(k)}} + b^{(k)} - g^{(k)} \times \frac{m_X^{(k)}}{s_X^{(k)}}$$

Note: At inference time, an unbiased estimate of the mean and standard deviation computed from all seen mini-batches during training is used.

Deep Learning

Additional Notes

Distributed Training

https://blog.skymind.ai/distributed-deep-learning-part-1-an-introduction-to-distributed-training-of-neural-networks/

Things we did not cover (not complete…)

Neural Artistic Style Transfer

Benchmark Datasets

Encoder-Decoder Networks

Transfer Learning

Siamese Networks

Variational Approaches

Pre-Training

Multi-Task Learning

Sequence Generation

Parameter Initialization

Neural Chatbots

Sequence Generation

Learning to Learn

Pre-trained Models

Vanishing/Exploding Gradient

Highway Networks

Dealing with Variable Length Inputs and Outputs

Transfer Learning

Mechanism for Training Ultra Deep Networks

Hessian-free optimization

Neural Computers

Recursive Neural Networks

(Unsupervised) Pre-training

Evolutionary Methods for Model Training

Weight Normalization

Autoregressive Models

Bayesian Neural Networks

Distributed Training

Embeddings

Layer Compression (e.g. Tensor-Trains)

Speech Modeling

Distillation

Character Level Neural Machine Translation

Weight Sharing

Hyper-parameter tuning

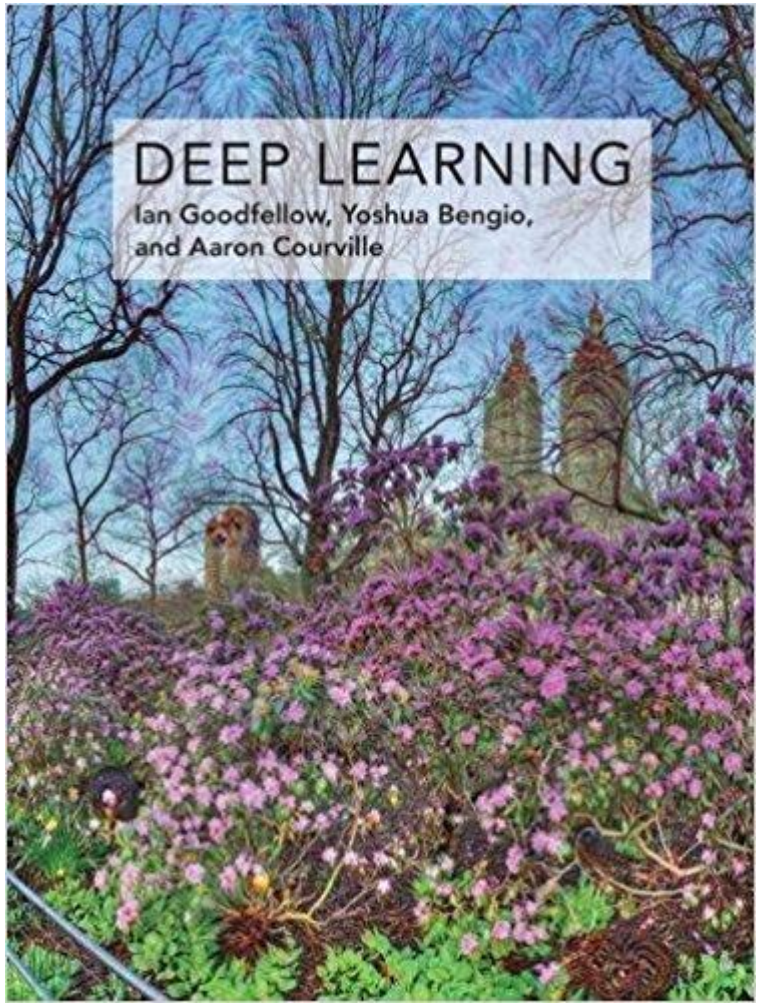
Multi-Lingual Neural Machine Translation

More Loss Functions

Generative adversarial networks

Deep Reinforcement Learning

Recommended Material



http://www.deeplearningbook.org/

Recommended Courses

Deep Learning:

There will be a lecture in WS 2018/2019

Deep Learning:

https://www.coursera.org/specializations/deep-learning#courses

Machine Learning

https://www.coursera.org/learn/machine-learning


SIEMENS
AI Lab
MUNICH
We will be live soon www.ai-hackathon.com
Tackling AI challenges
July 12th – 14th 2018
Two major companies join forces to dive deep into AI challenges together with you.
Join us for the AI Hackathon of the year!
Featuring tracks on computer vision and image recognition, digital companions, natural language processing, signal processing, anomaly detection, and more.
Let's develop something great together!

Additional Recommended Material

Module 1: Neural Networks

Image Classification: Data-driven Approach, k-Nearest Neighbor, train/val/test splits
L1/L2 distances, hyperparameter search, cross-validation

Linear classification: Support Vector Machine, Softmax
parameteric approach, bias trick, hinge loss, cross-entropy loss, L2 regularization, web demo

Optimization: Stochastic Gradient Descent
optimization landscapes, local search, learning rate, analytic/numerical gradient

Backpropagation, Intuitions
chain rule interpretation, real-valued circuits, patterns in gradient flow

Neural Networks Part 1: Setting up the Architecture
model of a biological neuron, activation functions, neural net architecture, representational power

Neural Networks Part 2: Setting up the Data and the Loss
preprocessing, weight initialization, batch normalization, regularization (L2/dropout), loss functions

Neural Networks Part 3: Learning and Evaluation
gradient checks, sanity checks, babysitting the learning process, momentum (+nesterov), second-order methods, Adagrad/RMSprop, hyperparameter optimization, model ensembles

Module 2: Convolutional Neural Networks

Convolutional Neural Networks: Architectures, Convolution / Pooling Layers
layers, spatial arrangement, layer patterns, layer sizing patterns, AlexNet/ZFNet/VGGNet case studies, computational considerations

Understanding and Visualizing Convolutional Neural Networks
tSNE embeddings, deconvnets, data gradients, fooling ConvNets, human comparisons

Transfer Learning and Fine-tuning Convolutional Neural Networks

Course Instructors

Fei-Fei Li

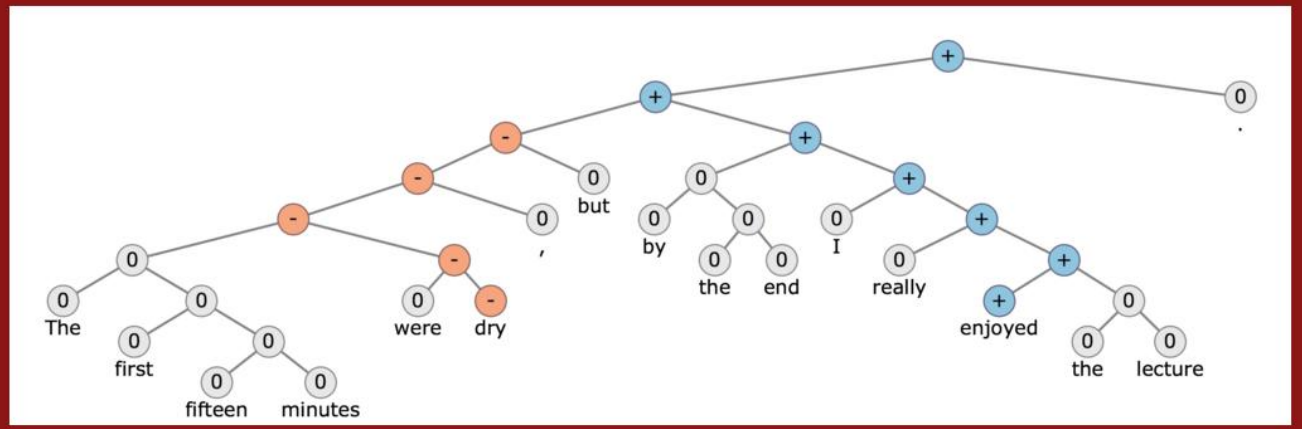
Andrej Karpathy

Justin Johnson

http://cs231n.stanford.edu/
http://cs231n.github.io

Additional Recommended Material

CS224d: Deep Learning for Natural Language Processing



Course Description

Natural language processing (NLP) is one of the most important technologies of the information age. Understanding complex language utterances is also a crucial part of artificial intelligence. Applications of NLP are everywhere because people communicate most everything in language: web search, advertisement, emails, customer service, language translation, radiology reports, etc. There are a large variety of underlying tasks and machine learning models powering NLP applications. Recently, deep learning approaches have obtained very high performance across many different NLP tasks. These models can often be trained with a single end-to-end model and do not require traditional, task-specific feature engineering. In this spring quarter course students will learn to implement, train, debug, visualize and invent their own neural network models. The course provides a deep excursion into cutting-edge research in deep learning applied to NLP. The final project will involve training a complex recurrent neural network and applying it to a large scale NLP problem. On the model side we will cover word vector representations, window-based neural networks, recurrent neural networks, long-short-term-memory models, recursive neural networks, convolutional neural networks as well as some very novel models involving a memory component. Through lectures and programming assignments students will learn the necessary engineering tricks for making neural networks work on practical problems.

Course Instructor

Richard Socher

http://cs224d.stanford.edu/

Additonal Recommended Material

INTRODUCTION

- Tutorial on Neural Networks (Deep Learning and Unsupervised Feature Learning): http://deeplearning.stanford.edu/wiki/index.php/UFLDL_Tutorial
- Deep Learning for Computer Vision lecture: http://cs231n.stanford.edu (http://cs231n.github.io)
- Deep Learning for NLP lecture: http://cs224d.stanford.edu (http://cs224d.stanford.edu/syllabus.html)
- Deep Learning for NLP (without magic) tutorial: http://lxmls.it.pt/2014/socher-lxmls.pdf (Videos from NAACL 2013: http://nlp.stanford.edu/courses/NAACL2013)

Additional Recommended Material

PARAMETER INITIALIZATION

- Glorot, Xavier, and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks." *International Conference on Artificial Intelligence and Statistics*. 2010.
- He, K., Zhang, X., Ren, S., & Sun, J. (2015). Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification. In *Proceedings of the IEEE International Conference on Computer Vision* (pp. 1026-1034).

BATCH NORMALIZATION

- Ioffe, S., & Szegedy, C. (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In *Proceedings of The 32nd International Conference on Machine Learning* (pp. 448-456).
- Cooijmans, T., Ballas, N., Laurent, C., & Courville, A. (2016). Recurrent Batch Normalization. *arXiv preprint arXiv:1603.09025*.

DROPOUT

- Hinton, Geoffrey E., et al. "Improving neural networks by preventing co-adaptation of feature detectors." *arXiv preprint arXiv:1207.0580* (2012).
- Srivastava, Nitish, et al. "Dropout: A simple way to prevent neural networks from overfitting." *The Journal of Machine Learning Research* 15.1 (2014): 1929-1958.

Additional Recommended Material

OPTIMIZATION & TRAINING

- Duchi, J., Hazan, E., & Singer, Y. (2011). Adaptive subgradient methods for online learning and stochastic optimization. *The Journal of Machine Learning Research*, *12*, 2121-2159.
- Zeiler, M. D. (2012). ADADELTA: An adaptive learning rate method. *arXiv preprint arXiv:1212.5701*.
- Tieleman, T., & Hinton, G. (2012). Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude. *COURSERA: Neural Networks for Machine Learning*, *4*, 2.
- Sutskever, I., Martens, J., Dahl, G., & Hinton, G. (2013). On the importance of initialization and momentum in deep learning. In *Proceedings of the 30th International Conference on Machine Learning (ICML)* (pp. 1139-1147).
- Kingma, D., & Ba, J. (2014). Adam: A method for stochastic optimization.*arXiv preprint arXiv:1412.6980*.
- Martens, J., & Sutskever, I. (2012). Training deep and recurrent networks with hessian-free optimization. In *Neural networks: Tricks of the trade* (pp. 479-535). Springer Berlin Heidelberg.

Additional Recommended Material

COMPUTER VISION

- Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems* (pp. 1097-1105).
- Taigman, Y., Yang, M., Ranzato, M. A., & Wolf, L. (2014). DeepFace: Closing the gap to human-level performance in face verification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1701-1708).
- Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., ... & Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition* (pp. 1-9).
- Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.
- Jaderberg, M., Simonyan, K., & Zisserman, A. (2015). Spatial transformer networks. In *Advances in Neural Information Processing Systems* (pp. 2008-2016).
- Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster R-CNN: Towards real-time object detection with region proposal networks. In *Advances in Neural Information Processing Systems* (pp. 91-99).
- Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A., Salakhudinov, R., ... & Bengio, Y. (2015). Show, Attend and Tell: Neural Image Caption Generation with Visual Attention. In *Proceedings of The 32nd International Conference on Machine Learning* (pp. 2048-2057).
- Johnson, J., Karpathy, A., & Fei-Fei, L. (2015). DenseCap: Fully Convolutional Localization Networks for Dense Captioning. *arXiv preprint arXiv:1511.07571*.

Additional Recommended Material

NATURAL LANGUAGE PROCESSING

- Bengio, Y., Schwenk, H., Senécal, J. S., Morin, F., & Gauvain, J. L. (2006). Neural probabilistic language models. In *Innovations in Machine Learning* (pp. 137-186). Springer Berlin Heidelberg.
- Collobert, R., Weston, J., Bottou, L., Karlen, M., Kavukcuoglu, K., & Kuksa, P. (2011). Natural language processing (almost) from scratch. *The Journal of Machine Learning Research*, *12*, 2493-2537.
- Mikolov, T. (2012). *Statistical language models based on neural networks* (Doctoral dissertation, PhD thesis, Brno University of Technology. 2012.)
- Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.
- Mikolov, T., Sutskever, I., Chen, K., Corrado, G. S., & Dean, J. (2013). Distributed representations of words and phrases and their compositionality. In *Advances in Neural Information Processing Systems* (pp. 3111-3119).
- Mikolov, T., Yih, W. T., & Zweig, G. (2013). Linguistic Regularities in Continuous Space Word Representations. In *HLT-NAACL* (pp. 746-751).
- Socher, R. (2014). *Recursive Deep Learning for Natural Language Processing and Computer Vision* (Doctoral dissertation, Stanford University).
- Cho, K., Van Merriënboer, B., Gulcehre, C., Bahdanau, D., Bougares, F., Schwenk, H., & Bengio, Y. (2014). Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*.
- Bahdanau, D., Cho, K., & Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*.