Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA)

Unit 8

Convolutional neural Network III

Kwai Wong, Stan Tomov,
Julian Halloy, Stephen Qiu, Eric Zhao

March 24, 2022

University of Tennessee, Knoxville

Acknowledgements:

- Support from NSF, UTK, JICS, ICL, NICS
- LAPENNA, www.jics.utk.edu/lapenna, NSF award #202409
- www.icl.utk.edu, cfdlab.utk.edu, www.xsede.org, www.jics.utk.edu/recsem-reu,
- MagmaDNN is a project grown out from the RECSEM REU Summer program supported under NSF award #1659502
- Source code: www.bitbucket.org/icl/magmadnn
- www.bitbucket.org/cfdl/opendnnwheel

INNOVATIVE COMPUTING LABORATORY

Overview of LAPENNA

The major goal of The Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA) program is to provide essential knowledge to advance literacy in AI to sustain the growth and development of the workforce in the cyberinfrastructure (CI) ecosystem. This program aims to prepare college faculty, researchers, and industrial practitioners to design, enable and direct their own course curricula, collaborative projects, and training programs for in-house data-driven sciences programs. The LAPENNA program focuses on delivering computational techniques, numerical algorithms and libraries, and implementation of AI software on emergent CPU and GPU platforms.

Ecosystem Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA)

Modeling, Numerical Linear Algebra, Data Analytics, Machine Learning, DNN, GPU, HPC

Session 1	Session 2	Session 3	Session 4
7/2020 - 12/2020	1/2021- 6/2021	7/2021 - 1/2022	1/2022 - 6/2022
16 participants	16 participants	16 participants	16 participants
Faculty/Students	Faculty/Students	Faculty/Students	Faculty/Students
10 webinars	10 webinars	10 webinars	10 webinars
4 Q & A	4 Q & A	4 Q & A	4 Q & A

Colleges courses, continuous integration, online courses, projects, software support

Web-based resources, tutorials, webinars, training, outreach

Teams and Participants of LAPENNA

- ✓ **PIs : Kwai Wong (JICS), Stan Tomov (ICL), University of Tennessee, Knoxville**
  - **Stephen Qiu, Julian Halloy, Eric Zhao (Students)**

- ✓ **Team : Clemson University**
- ✓ **Teams : University of Arkansas**
- ✓ **Team : University of Houston, Clear Lake**
- ✓ **Team : Miami University, Ohio**
- ✓ **Team : Boston University**
- ✓ **Team : West Virginia University**
- ✓ **Team : Louisiana State University, Alexandria**
- ✓ **Teams : Jackson Laboratory**
- ✓ **Team : Georgia State University**
- ✓ **Teams : University of Tennessee, Knoxville**
- ✓ **Teams : Morehouse College, Atlanta**
- ✓ **Team : North Carolina A & T University**
- ✓ **Team : Clark Atlanta University, Atlanta**
- ✓ **Team : Alabama A & M University**
- ✓ **Team : Slippery Rock University**
- ✓ **Team : University of Maryland, Baltimore County**

Meeting Time

- ✓ **Webinar Meeting time. Thursday 8:00 – 10:00 pm ET,**
- ✓ **Tentative schedule, www.jics.utk.edu/lapenna --> Spring 2022**

Topic: LAPENNA Spring 2022 Webinar
Time: Feb 3, 2022 08:00 PM Eastern Time (US and Canada)
Every week on Thu, 12 occurrence(s)
Feb 3, 2022 07:30 PM
Feb 10, 2022 07:30 PM
Feb 17, 2022 07:30 PM
Feb 24, 2022 07:30 PM
Mar 3, 2022 07:30 PM
Mar 10, 2022 07:30 PM
Mar 17, 2022 07:30 PM
Mar 24, 2022 07:30 PM
Mar 31, 2022 07:30 PM
Apr 7, 2022 07:30 PM
Apr 14, 2022 07:30 PM
Apr 21, 2022 07:30 PM

Join from PC, Mac, Linux, iOS or Android: https://tennessee.zoom.us/j/94140469394
Password: 708069

Schedule of LAPENNA Spring 2022
Thursday 8:00pm -10:00pm Eastern Time

The goal of The Linear Algebra Preparation for Emergent Neural Network Architectures (LAPENNA) program is to provide essential knowledge to advance literacy in AI to sustain the growth and development of the workforce in the cyberinfrastructure (CI) ecosystem for data-driven applications.

Month	Week	Date	Topics
February	Week 01	3	Logistics, High Performance Computing
	Week 02	10	Computational Ecosystem, Linear Algebra
	Week 03	17	Introduction to DNN, Forward Path (MLP)
	Week 04	24	Backward Path (MLP), Math, Example
March	Week 05	3	Backpro (MLP), CNN Computation
	Week 06	10	CNN Backpropagation, Example
	Week 07	17	CNN Network, Linear Algebra
	Week 08	24	Segmentation, Unet,
	Week 09	31	Object Detection, RC vehicle
April	Week 10	7	RNN, LSTEM, Transformers
	Week 11	14	DNN Computing on GPU
	Week 12	21	Overview, Closing
June or July	Workshop	To be arranged	4 Days In Person at UTK

Topics of Unit 6 - CNN

- ✓ **UNIT 7 : Convolutional Neural Network (CNN)**
  - ➢ **Overview Forward and Backpropagation of MLP**
  - ➢ **CNN computation**
  - ➢ **CNN forward path calculation**
  - ➢ **MNIST TensorFlow code**
  - ➢ **CIFAT 10 example**
  - ➢ **AlexNet, number of parameters**
  - ➢ **CNN Backward Calculations**
  - ➢ **Homework exercises**

DNN Model

Applications

+

Data Ensemble + Input

+

It's all about

linear algebra calculations

DNN in particular

+

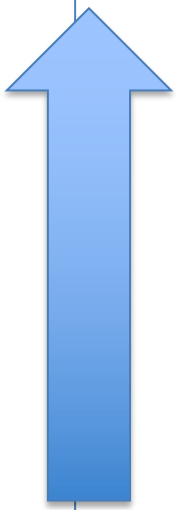
Algorithms, Software

+

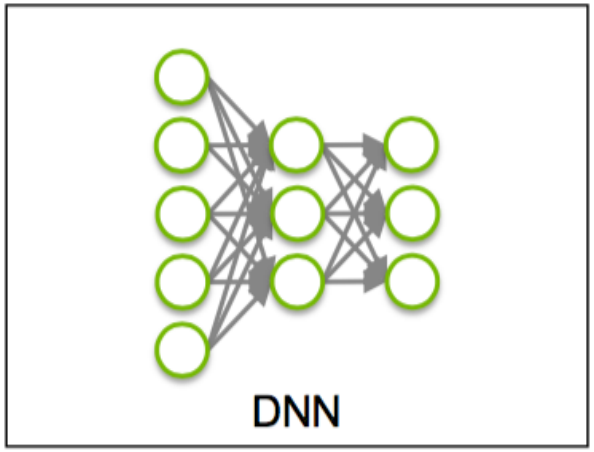
Output + Data Analysis

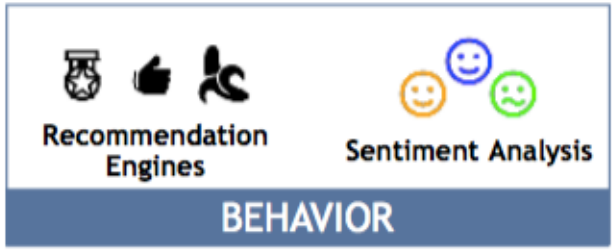
+

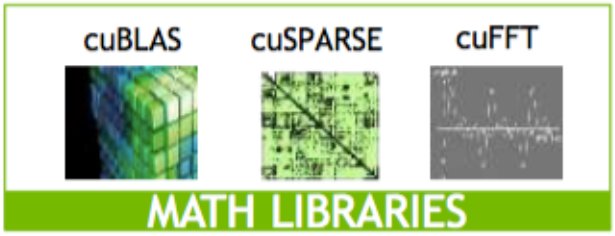
Hardware

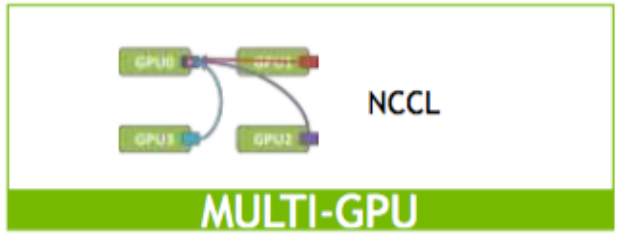


Machine Learning – GPU acceleration









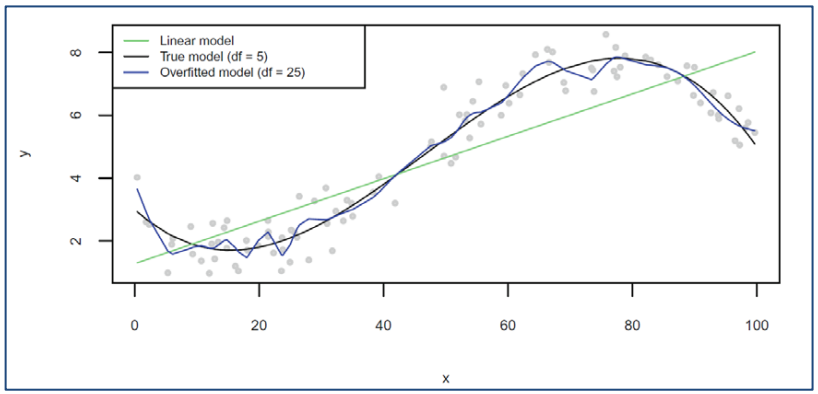
Source captured frin - Julie Bernauer – HPC Advisory Council Stanford Tutorial – 2017/02/07

Supervised learning is a type of machine learning which automate decision-making processes by generalizing from known examples. That is, the users provides the algorithm with pairs of inputs and desired outputs. The algorithm finds a way to produce the desired output given an input. In other words, we construct a model for the problem. The model is governed by some unknown parameters which we will learn from the data.

Supervised Learning Data:
(x, y) x is data, y is label

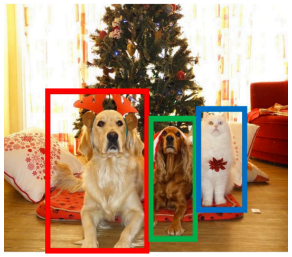
Goal: Learn a function to map x -> y

Examples:
Classification (discrete values),
regression (numerical values),



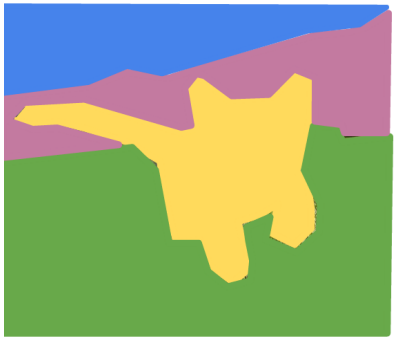
CAT

Classification



DOG, DOG, CAT

Object Detection



GRASS, CAT, TREE, SKY

Semantic Segmentation

Basic Ideas

Typical Neural Network – MLP

Convolutional Neural Network

STEP 1 : Model Definition

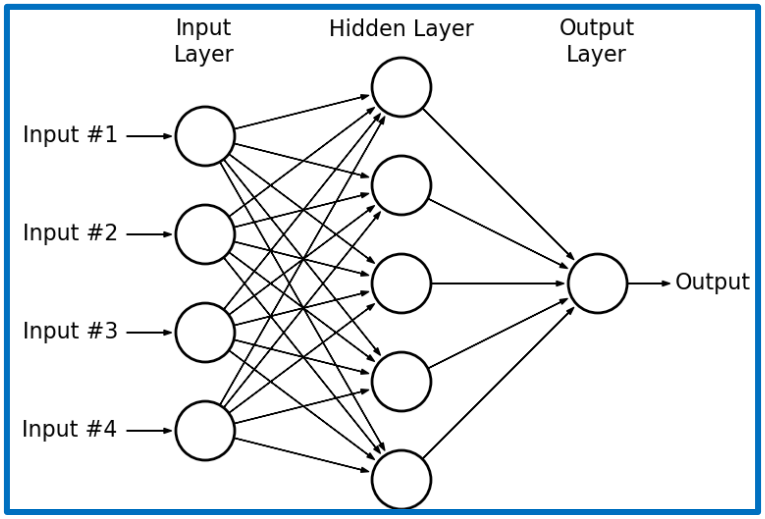
STEP 2 : Cost Function

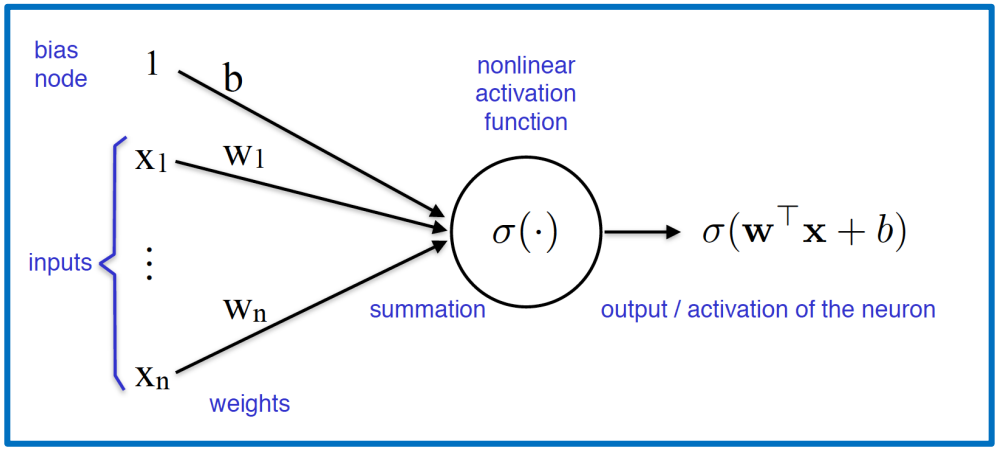
Step 3 : Optimization Scheme

Step 4 : Numerical Implementation (fitting)

Step 5 : Evaluation

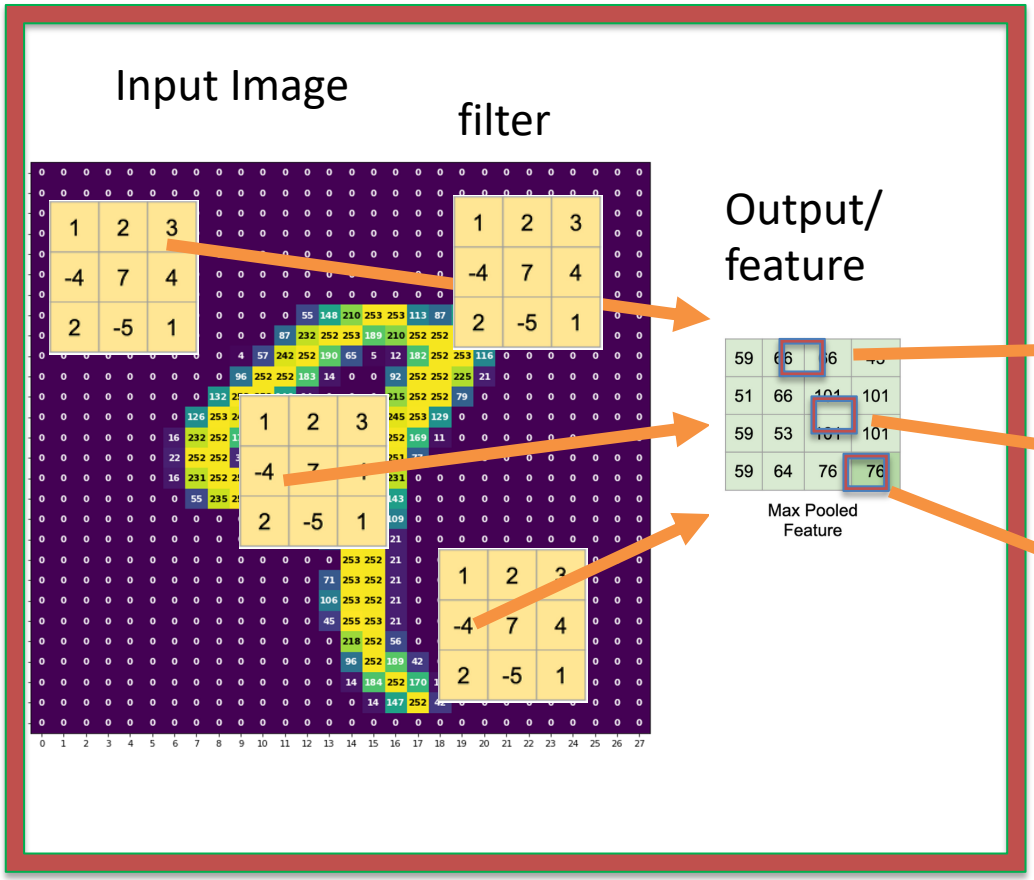
Parametric Model : NN Modeling

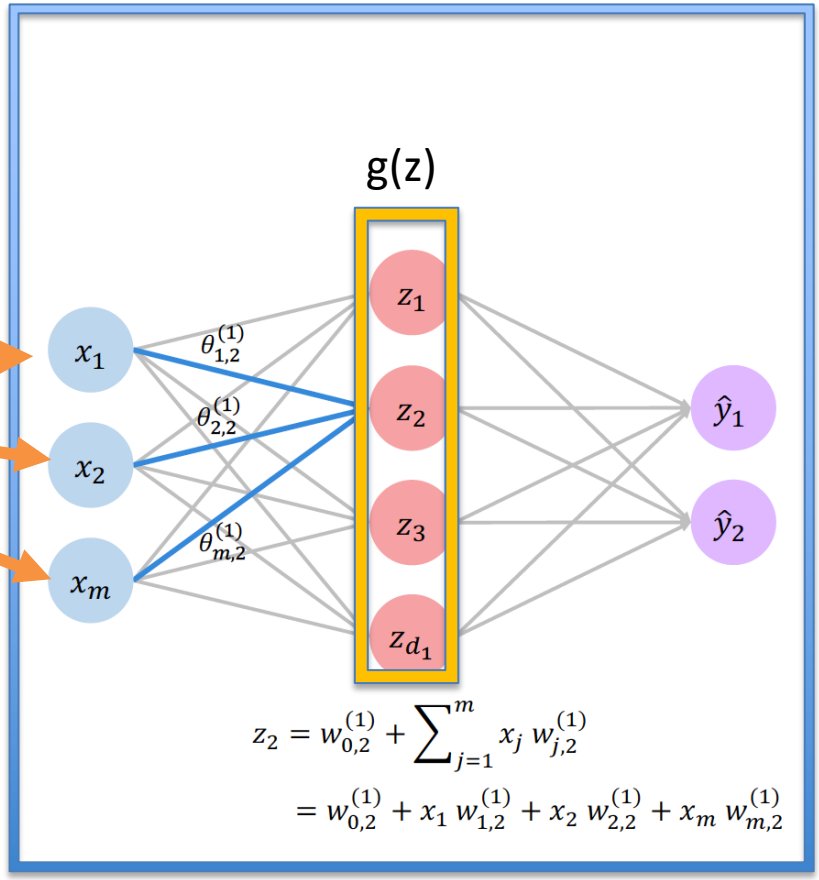




- ✓ A node in the neural network is a mathematical function or activation function which maps input to output values.
- ✓ Inputs represent a set of vectors containing weights (w) and bias (b). They are the sets of parameters to be determined.
- ✓ Many nodes form a **neural layer**, **links** connect layers together, defining a NN model.
- ✓ Activation function (f or σ), is generally a nonlinear data operator which facilitates identification of complex features.

Step 1: Parametric Model : Convolutional Neural Network (CNN)
Convolutional filter + Connected Neural Network





Convolutional filtering

Multilayer Perceptron

MLP : Neural Network

CNN : Convolutional NN

STEP 1 : Model Definition

STEP 2 : Cost Function

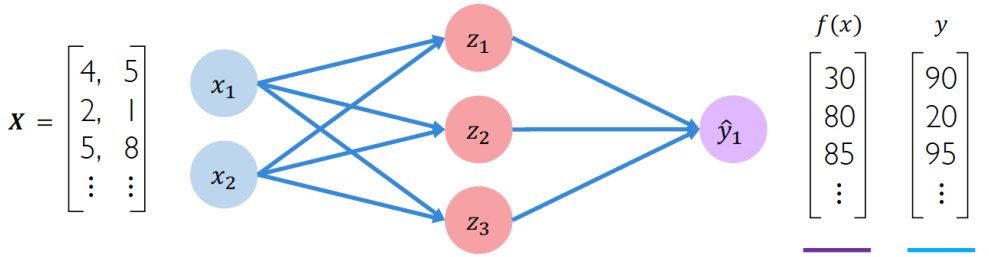
Step 3 : Optimization Scheme

Step 4 : Numerical Implementation

Step 5 : Evaluation

NN Cost Function (regression) : Mean Squared Error Loss

Mean squared error loss can be used with regression models that output continuous real numbers



$$J(W) = \frac{1}{n}\sum_{i=1}^{n}\left(y^{(i)} - f(x^{(i)}; W)\right)^2$$

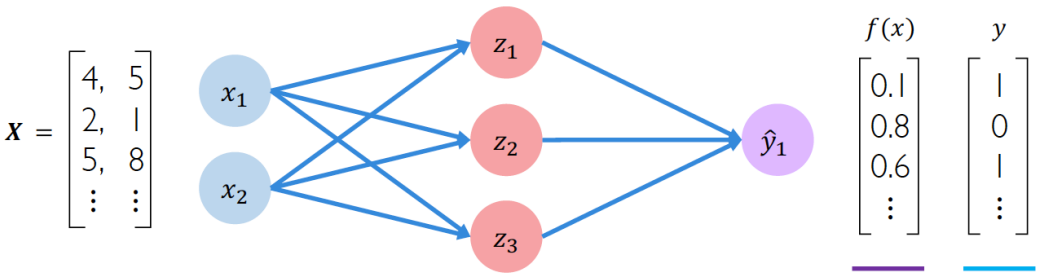
Actual Predicted

```
loss = tf.reduce_mean( tf.square(tf.subtract(model.y, model.pred) )
```

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

NN Cost Function (Classification) : Cross Entropy Loss

Cross entropy loss can be used with models that output a probability between 0 and 1



$$J(W) = \frac{1}{n}\sum_{i=1}^{n} y^{(i)} \log\left(f(x^{(i)}; W)\right) + (1 - y^{(i)}) \log\left(1 - f(x^{(i)}; W)\right)$$

Actual Predicted Actual Predicted

```
loss = tf.reduce_mean( tf.nn.softmax_cross_entropy_with_logits(model.y, model.pred) )
```

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

MLP : Neural Network

CNN : Convolutional NN

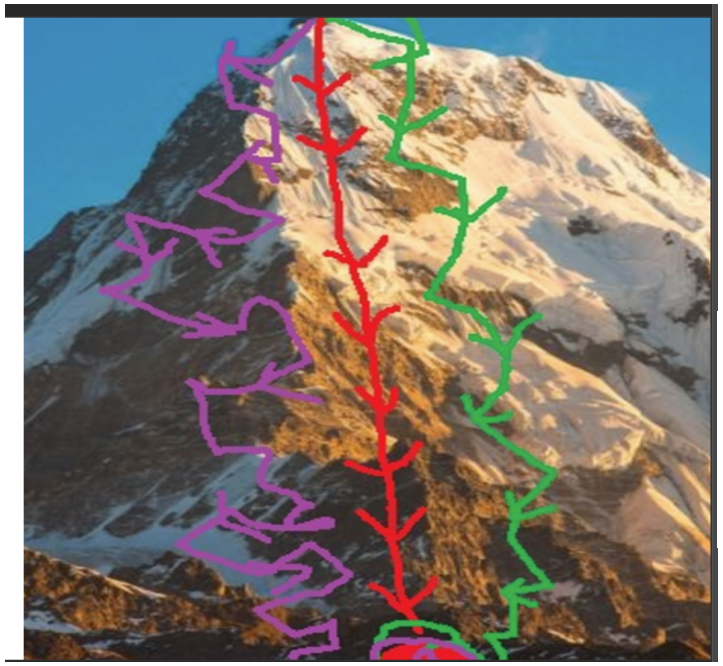
STEP 1 : Model Definition

STEP 2 : Cost Function

Step 3 : Optimization Scheme

Step 4 : Numerical Implementation

Step 5 : Evaluation



Optimization Schemes: Gradient Descent : Simple Story

Batch Gradient Descent
Mini-batch Gradient Descent
Stochastic Gradient Descent

Derivative is the slope of the tangent line

$\frac{d}{dx}x^2 = 2x$

$x = 1$

The slope of the tangent line is 2

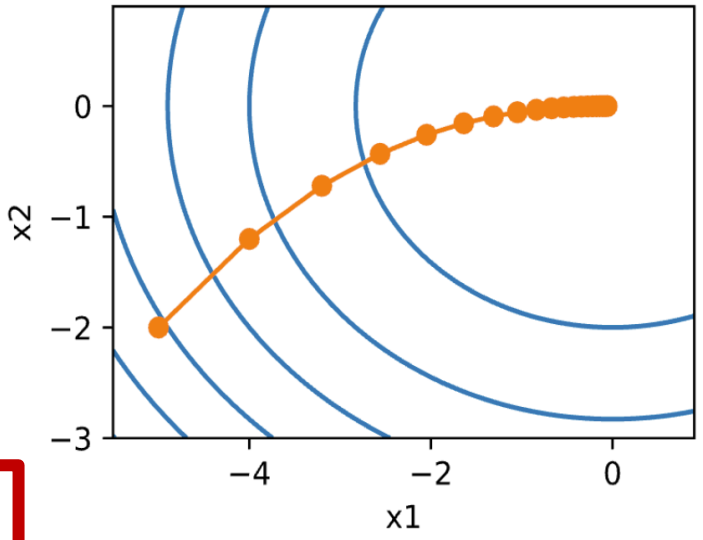
Algorithm

- Choose initial $\mathbf{x}_0$ (X = W, weight)
- At time $t = 1, \ldots, T$

$$\mathbf{x}_t = \mathbf{x}_{t-1} - \eta \nabla f(\mathbf{x}_{t-1})$$

- η is called learning rate

W+ = W – (learning rate) * dJ/dw



DNN Forward Backward Calculation : Weights and bias

Algorithm

1. Initialize weights randomly $\sim \mathcal{N}(0, \sigma^2)$

```
weights = tf.random_normal(shape, stddev=sigma)
```

2. Loop until convergence:
3. Compute gradient, $\frac{\partial J(W)}{\partial W}$

```
grads = tf.gradients(ys=loss, xs=weights)
```

4. Update weights, $W \leftarrow W - \eta \frac{\partial J(W)}{\partial W}$

```
weights_new = weights.assign(weights - lr * grads)
```

5. Return weights

In one epoch

1. Pick a mini-batch
2. Feed it to Neural Network
3. Forward path calculation
4. Calculate the mean gradient of the mini-batch
5. Use the mean gradient calculated in step 4 to update the weights
6. Repeat steps 1–5 for the mini-batches we created

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C,$$

$$w_k \to w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

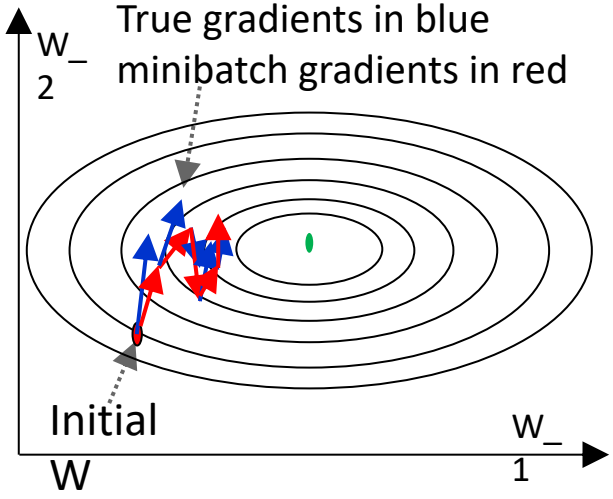
$$b_l \to b_l' = b_l - \eta \frac{\partial C}{\partial b_l}.$$

Optimization: Batch SGD

Instead of computing a gradient across the entire dataset with size N, divides the dataset into a fixed-size subset ("minibatch") with size m, and computes the gradient for each update on this smaller batch:

(N is the dataset size, m is the minibatch size)

$$\mathbf{g} = \frac{1}{m'}\sum_{i=1}^{m'} \nabla_{\boldsymbol{\theta}} E_i(\mathbf{x}^{(n_i)}, y^{(n_i)}, \boldsymbol{\theta}),$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \eta\mathbf{g},$$



Epoch step

Given: training set $\{(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_n, y_n)\}$
Initialize all $\Theta^{(l)}$ randomly (NOT to 0!)
Loop // each iteration is called an epoch
  Loop // each iteration is a mini-batch
    Set $\Delta_{i,j}^{(l)} = 0 \quad \forall l, i, j$ (Used to accumulate gradient)
    Sample m training instances $\mathcal{X} = \{(\mathbf{x}'_1, y'_1), \ldots, (\mathbf{x}'_m, y'_m)\}$ without replacement
    For each instance in $\mathcal{X}$, $(\mathbf{x}_k, y_k)$:
      Set $\mathbf{a}^{(1)} = \mathbf{x}_k$
      Compute $\{\mathbf{a}^{(2)}, \ldots, \mathbf{a}^{(L)}\}$ via forward propagation
      Compute $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y_k$
      Compute errors $\{\boldsymbol{\delta}^{(L-1)}, \ldots, \boldsymbol{\delta}^{(2)}\}$
      Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$
    Compute mini-batch regularized gradient $D_{ij}^{(l)} = \begin{cases} \frac{1}{m}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{m}\Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$
    Update weights via gradient step $\Theta_{ij}^{(l)} = \Theta_{ij}^{(l)} - \alpha D_{ij}^{(l)}$
  Until all training instances are seen
Until weights converge or max #epochs is reached

Batch iteration

MLP : Neural Network

CNN : Convolutional NN

STEP 1 : Model Definition

STEP 2 : Cost Function

Step 3 : Optimization Scheme

Step 4 : Numerical Implementation

TensorFlow MNIST MLP & CNN Examples

Training

"Training a neural network"

What does it mean?

What does the network train?

minimize error of the computed values against the labeled values (loss function)

What are the training parameters

W, b ,**and values of the filters (Wf)**

How does it work?

optimization based on gradient descent algorithm

go back and forth in the NN model to adjust for w and b => need derivatives w.r.t. parameters

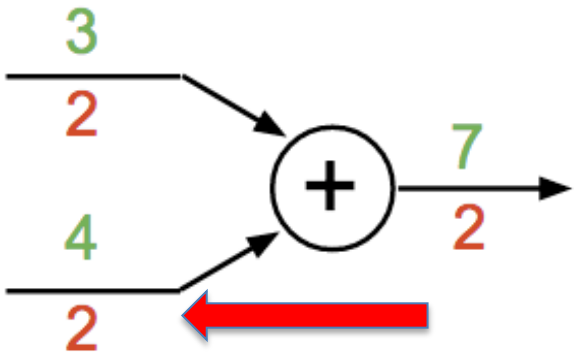
Go through forward calculation

training with backpropagation

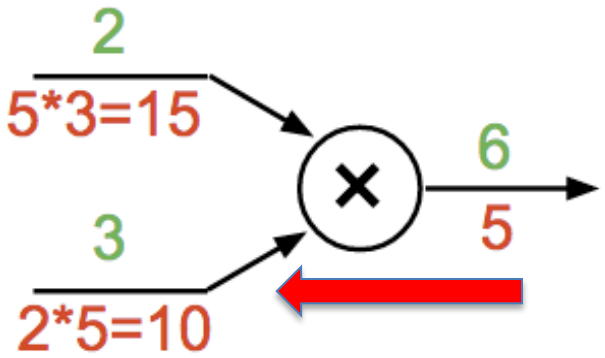
Forward and Backpropagation Operators

Forward path : backward path

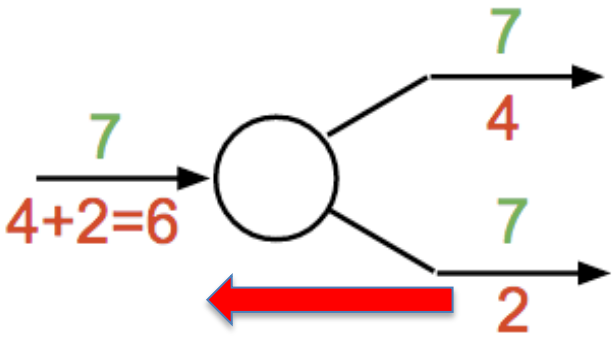
add gate: gradient distributor



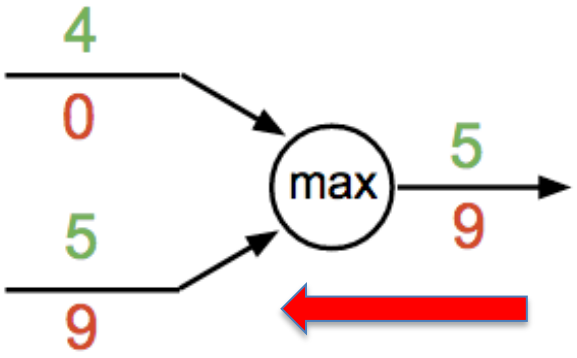
mul gate: "swap multiplier"



copy gate: gradient adder



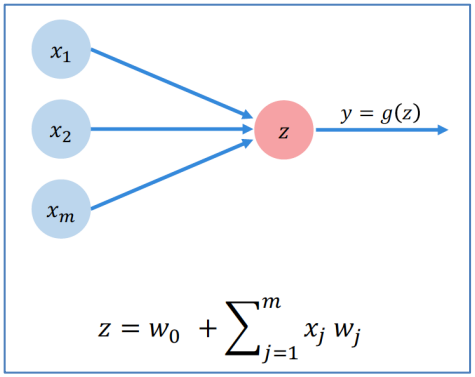
max gate: gradient router



http://cs231n.stanford.edu/slides/2020/lecture_4.pdf

Forward and backward computation operators

Z = (x1*w1 + x2*w2) + b = (-1.0*-2.0 + -2.0*3.0) + -3.0 = 4.0 – 3.0 = 1.0



Sigmoid function

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

Y = σ (z) = 1 / (1 + e $^{-Z}$)
= 1 / (1 + e $^{-1}$) = 0.731

Z = (x1*w1 + x2*w2) + b

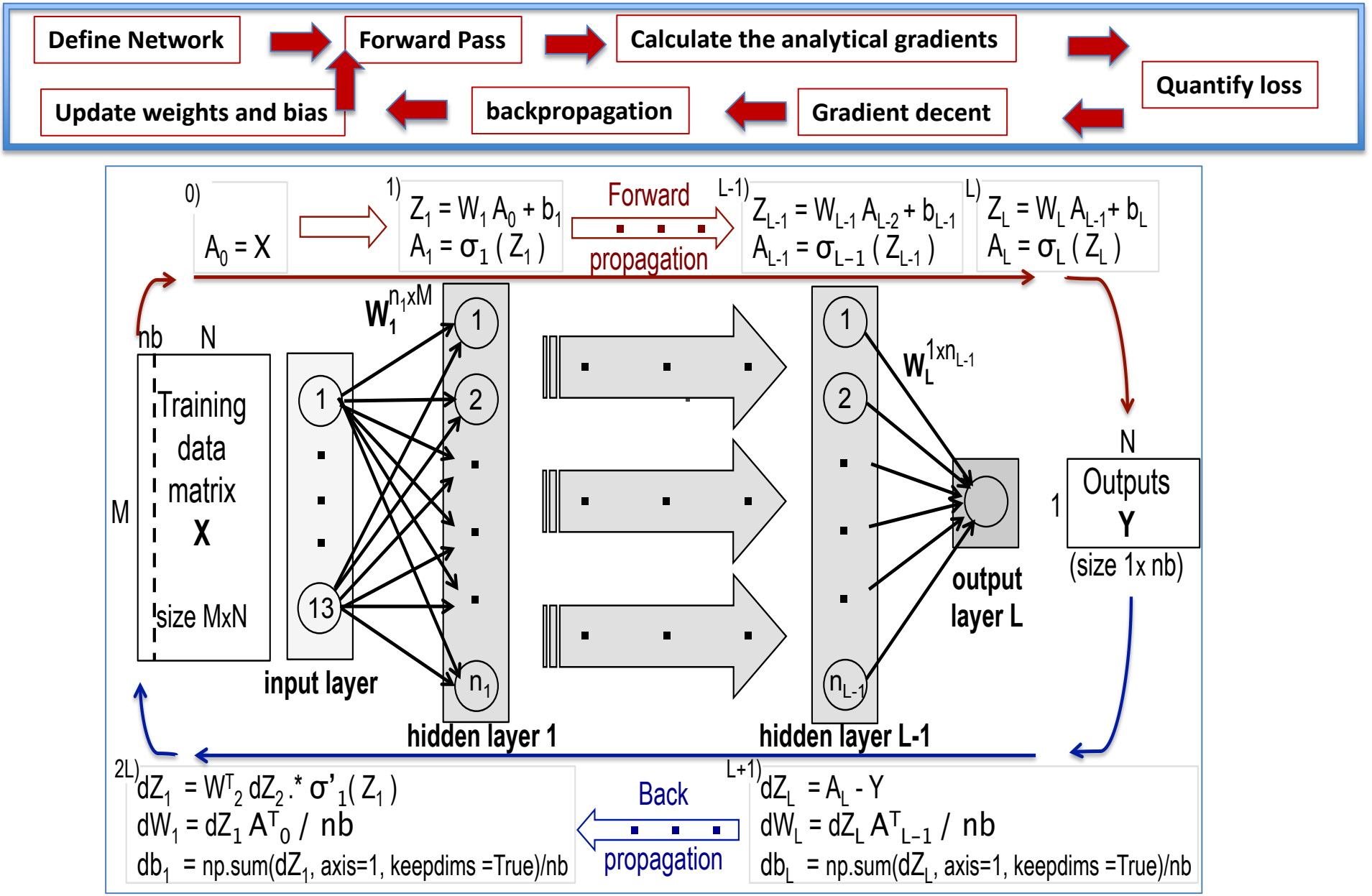
multiple operator :
downstream gradient =
upstream gradient * input value

add operator :
Downstream gradient =
upstream gradient

function operator :
downstream gradient =
Upstream gradient * local function gradient

Summary : Flow of NN

```
model.fit(x_train, y_train, epochs=3, batch_size=10)
```



MNIST Example (28x28 pixels)

Handwritten digits in the MNIST dataset are 28x28 pixel grayscale images. The simplest approach for classifying them is to use the 28x28=784 pixels as inputs for a 1-layer neural network.

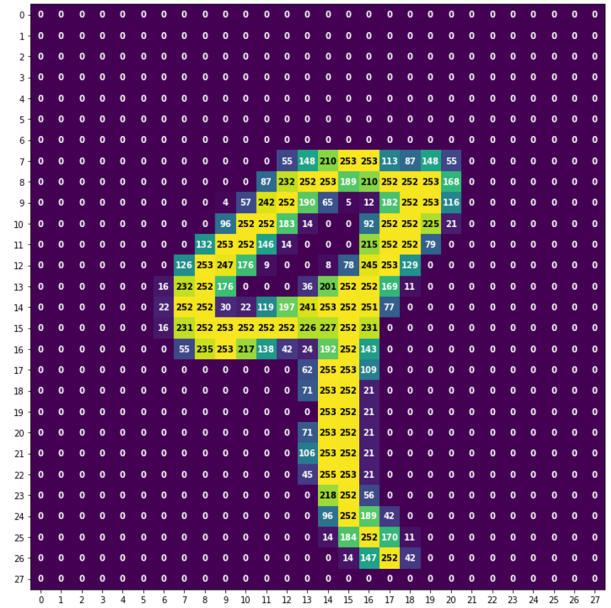
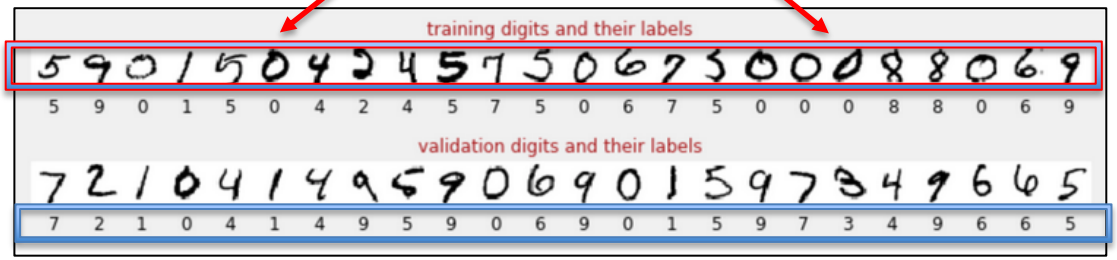
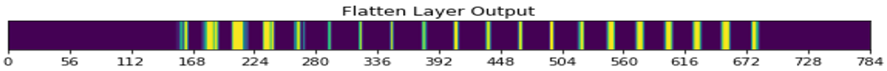


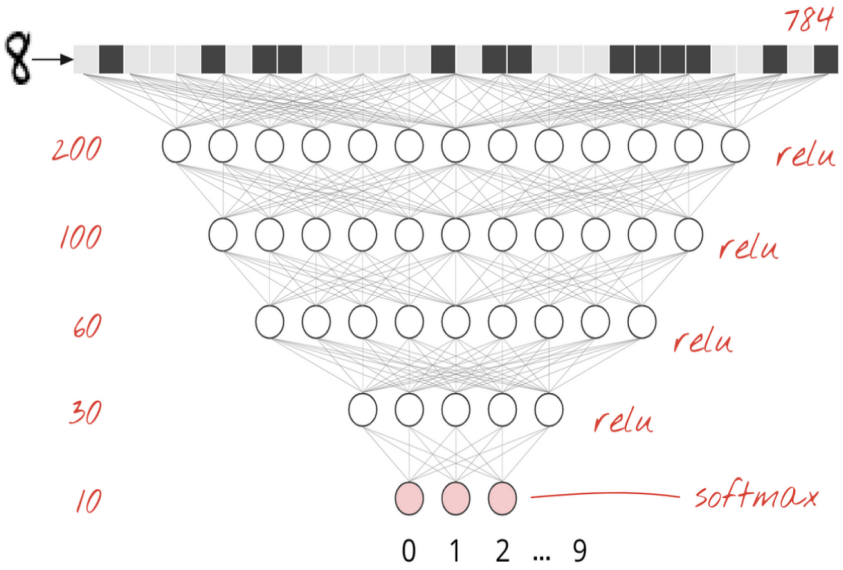
Image : input

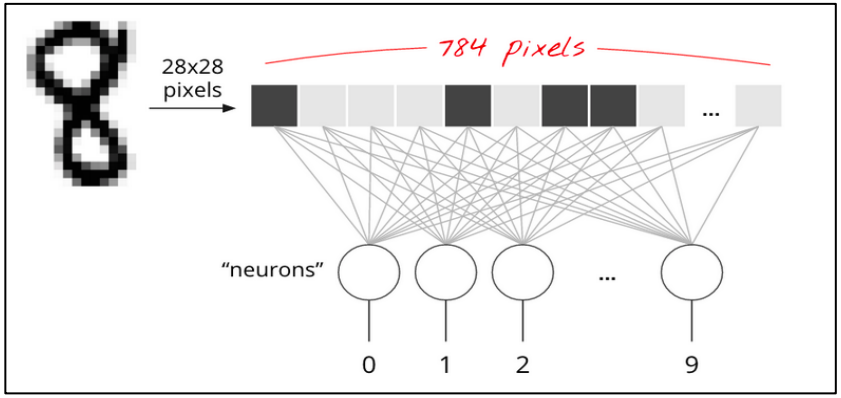


Labels : target

Flatten the 28 x 28 matrix
to a vector of 28 x 28 elements 784 elements







TensorFlow, Keras and deep learning, without a PhD
https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist#0

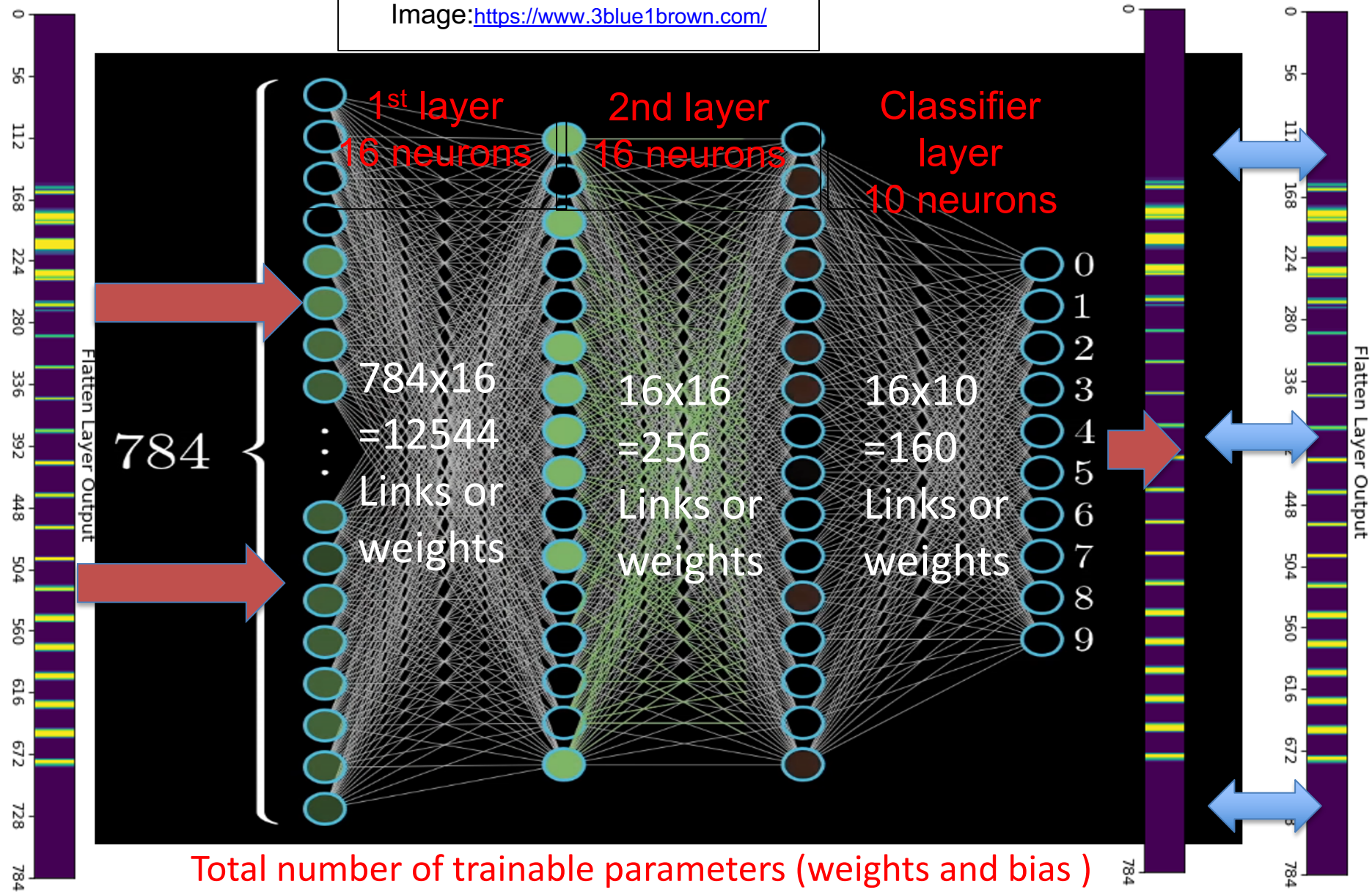
Simple MNIST MLP Neural Network

Input

output

labels

Image:https://www.3blue1brown.com/

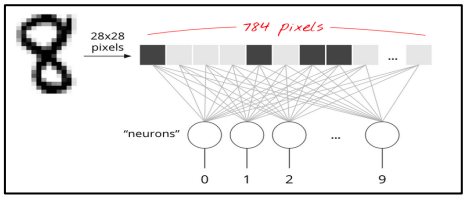


Total number of trainable parameters (weights and bias)

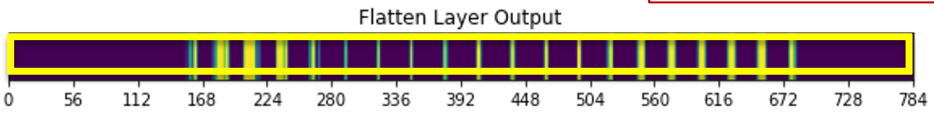
12544+16+256+16+160+10= 13002

Summary : Flow of MLP Dense layer NN

1 image = a vector of 784



An image of 28x28 pixels



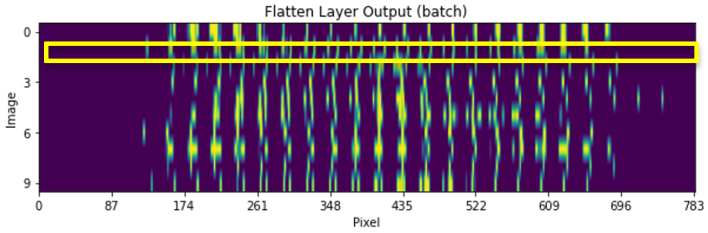
Input to 1st layer

10 images : batch of 10
10 x 784 matrix

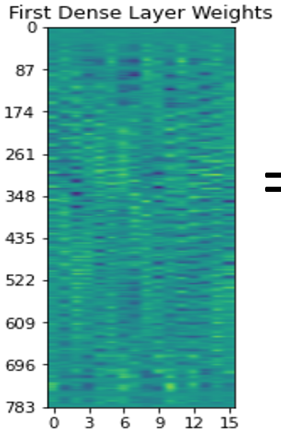
16 neurons
784 x 16 W matrix

10 x 16 output matrix

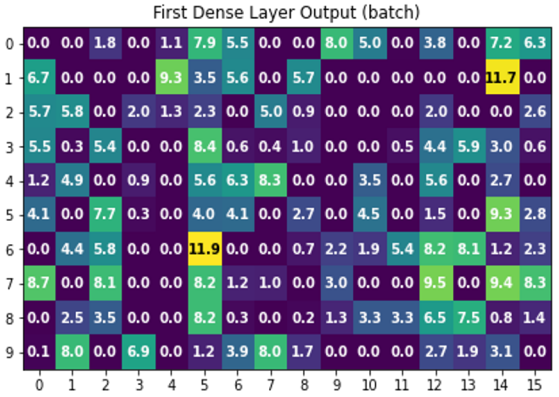
Input to 2nd layer



*

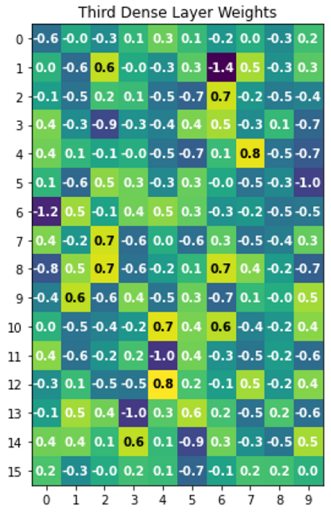


=



10 x 16 matrix

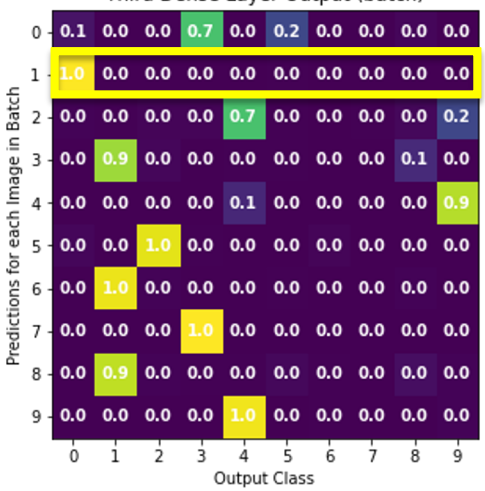
*



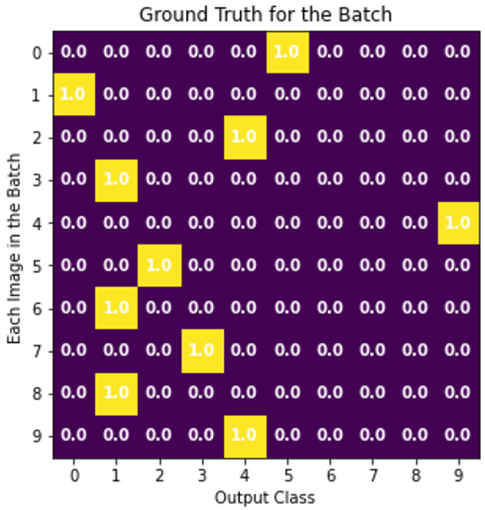
16 neurons
16 x 10
W matrix

=

10 x 10 output matrix



10 x 10 output matrix



10 x 10 labeled matrix

Tensorflow Example

```
import tensorflow as tf

mnist = tf.keras.datasets.mnist

(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

model = tf.keras.models.Sequential([
  tf.keras.layers.Flatten(input_shape=(28, 28)),
  tf.keras.layers.Dense(16, activation='relu'),
 tf.keras.layers.Dense(16, activation='relu'),
  tf.keras.layers.Dense(10, activation='softmax')
])
```

Tensorflow is imported

Sets the mnist dataset to variable "mnist"

Loads the mnist dataset

Builds the layers of the model
4 layers in this model

```
model.compile(optimizer='sdg', loss='SparseCategoricalCrossentropy',
metrics=['accuracy'])

model.summary()

logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-
%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)
```

Loss Function

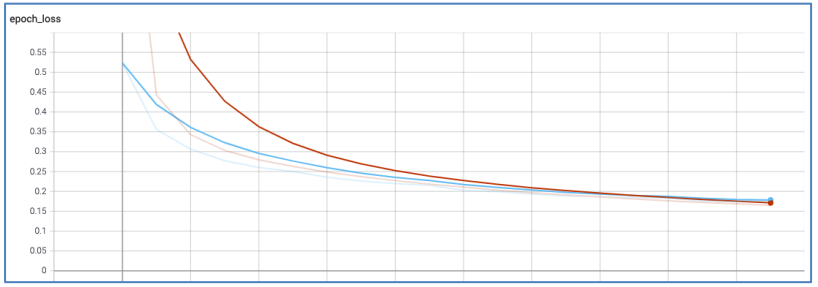
Compiles the model with the SGD optimizer

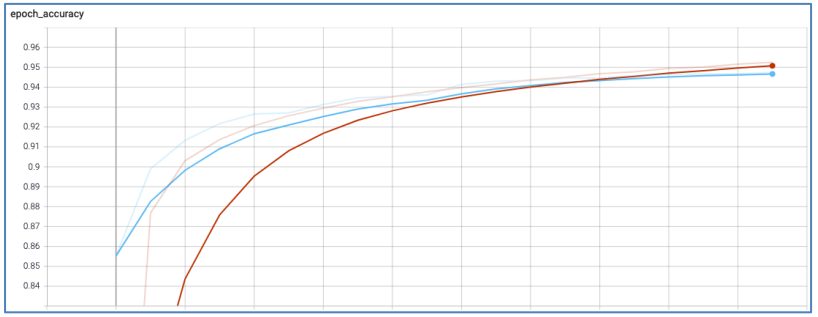
Print summary

Use tensorborad

```
model.compile(optimizer='sgd',
              loss='SparseCategoricalCrossentropy',
              metrics=['accuracy'])

model.summary()

logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)

model.fit(x_train, y_train, epochs=20, batch_size=50,validation_data=(x_test, y_test),
            callbacks=[tensorboard_callback])

model.evaluate(x_test,  y_test, verbose=2)

%tensorboard --logdir logs
```

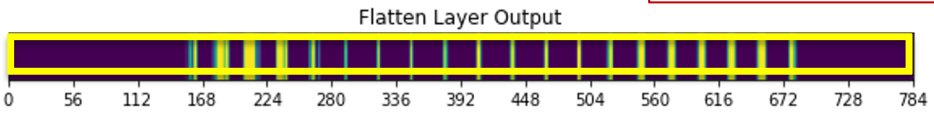
```
2.4.1
The tensorboard extension is already loaded. To reload it, use:
  %reload_ext tensorboard
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten_1 (Flatten)          (None, 784)               0
_________________________________________________________________
dense_3 (Dense)              (None, 16)                12560
_________________________________________________________________
dense_4 (Dense)              (None, 16)                272
_________________________________________________________________
dense_5 (Dense)              (None, 10)                170
=================================================================
Total params: 13,002
Trainable params: 13,002
Non-trainable params: 0
```

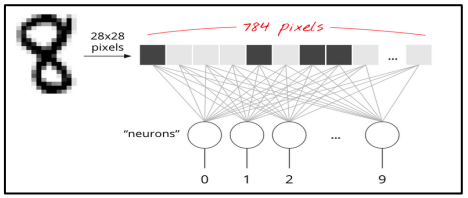




Summary : Flow of MLP Dense layer NN

1 image = a vector of 784

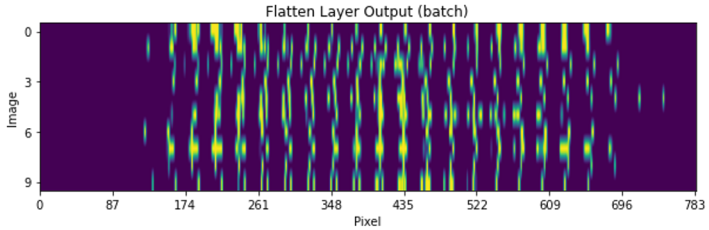




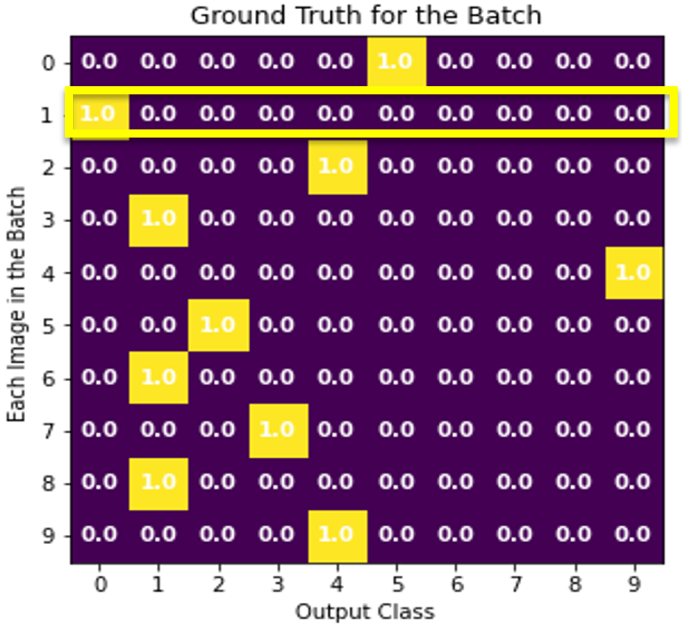
An image of 28x28 pixels

Input to 1st layer

10 images : batch of 10
10 x 784 matrix

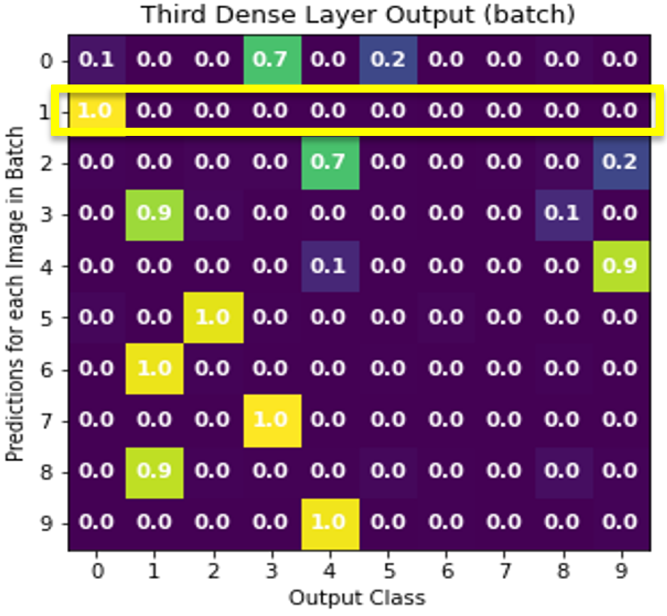


10 x 10 labeled matrix

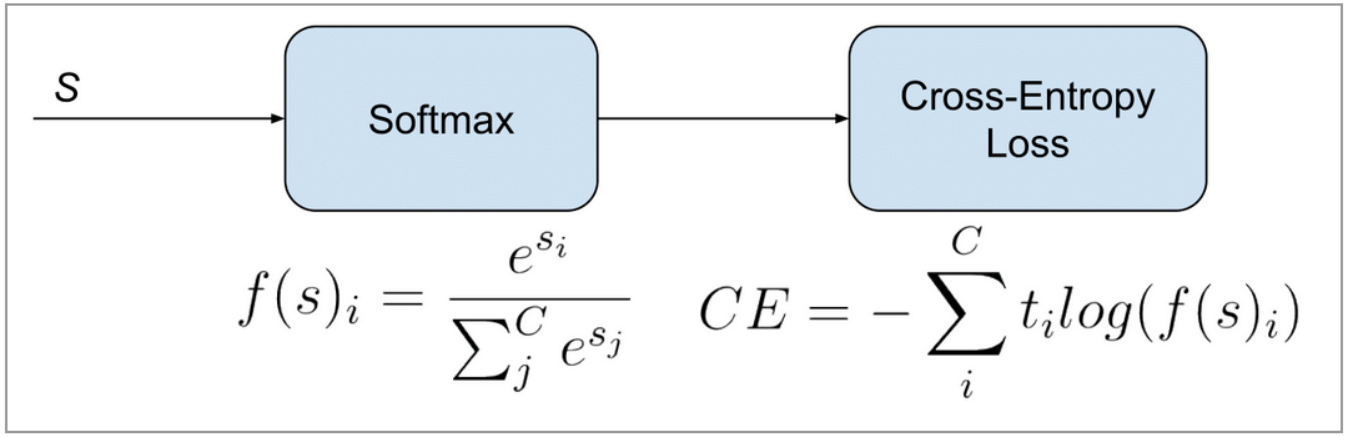


Comparing Computed NN results To Labeled results (target)

10 x 10 output matrix



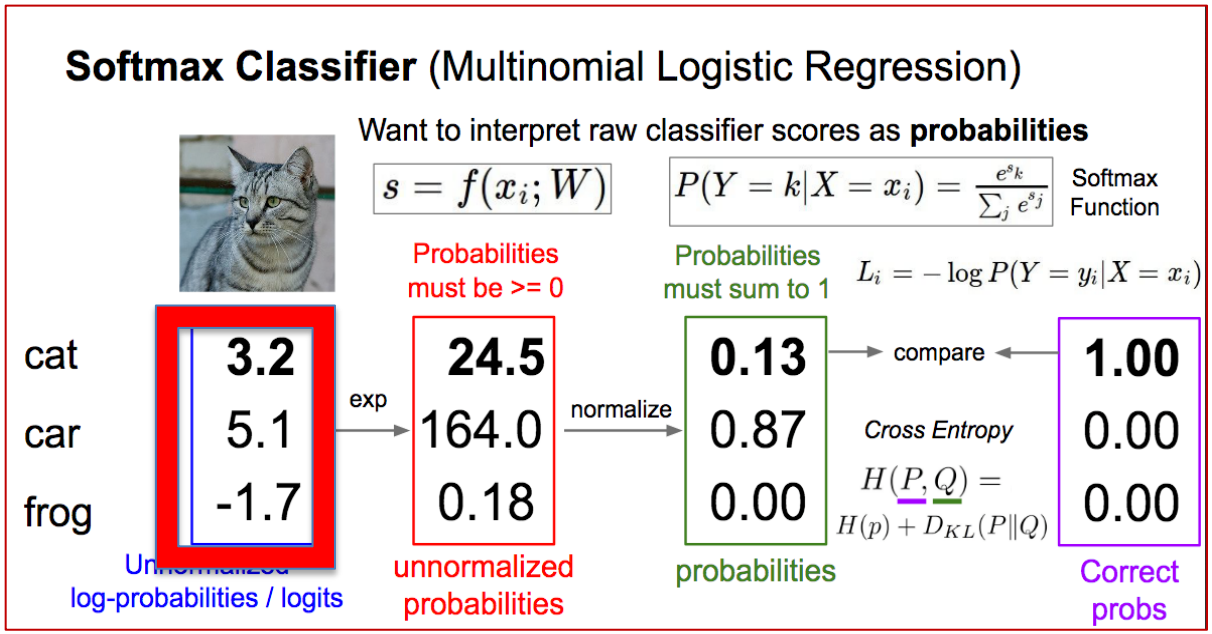
Categorical (Softmax) Cross Entropy Loss (Statistical Learning)



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

T = 1

ti and si are the groundtruth (label) and the computed score for each class i in C.



https://gombru.github.io/2018/05/23/cross_entropy_loss/

e^3.2 = 24.5

e^5.1 = 164.0

e^-1.7 = 0.18

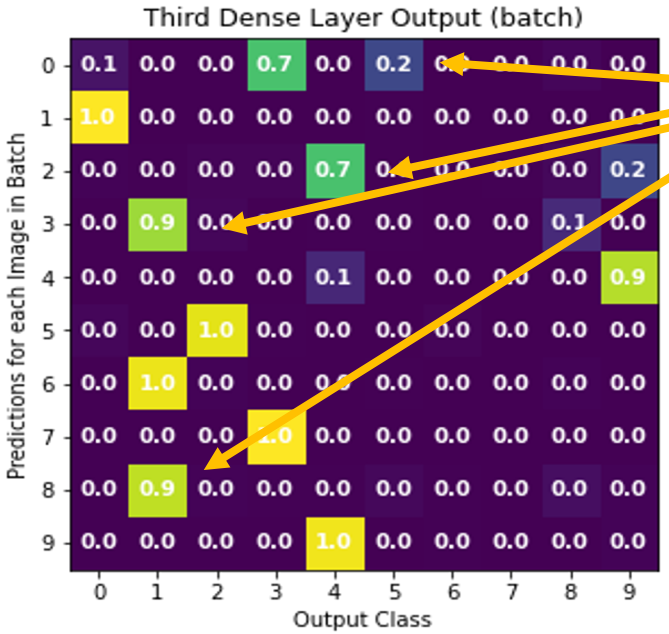
188.68

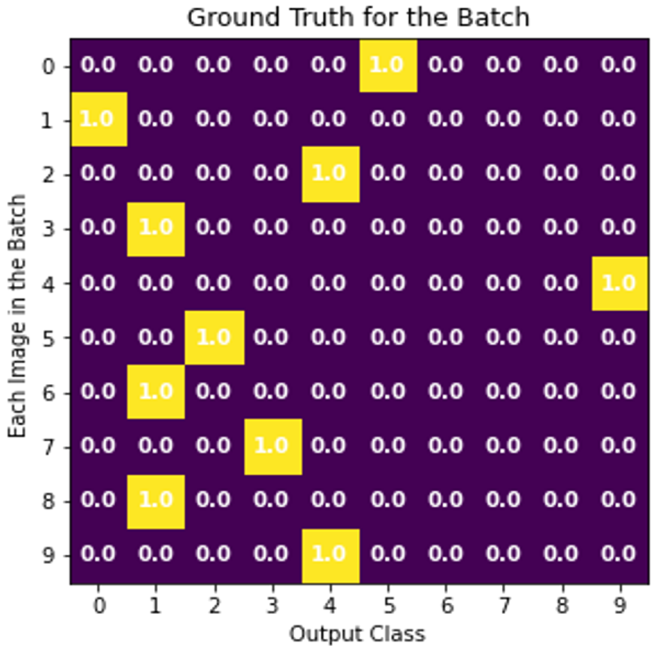
Software = 24.5 / 188.68 = 0.13

Software = 164 / 188.68 = 0.87

Software = -1.7 / 188.68 = 0.00

Evaluating the Error





Z

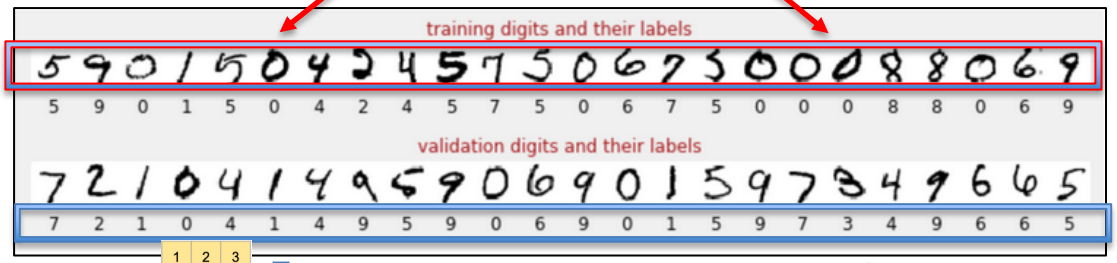
Categorical Cross Entropy Loss

$$L_i = -\log\left(\frac{e^{sy_i}}{\sum_j e^{s_j}}\right)$$

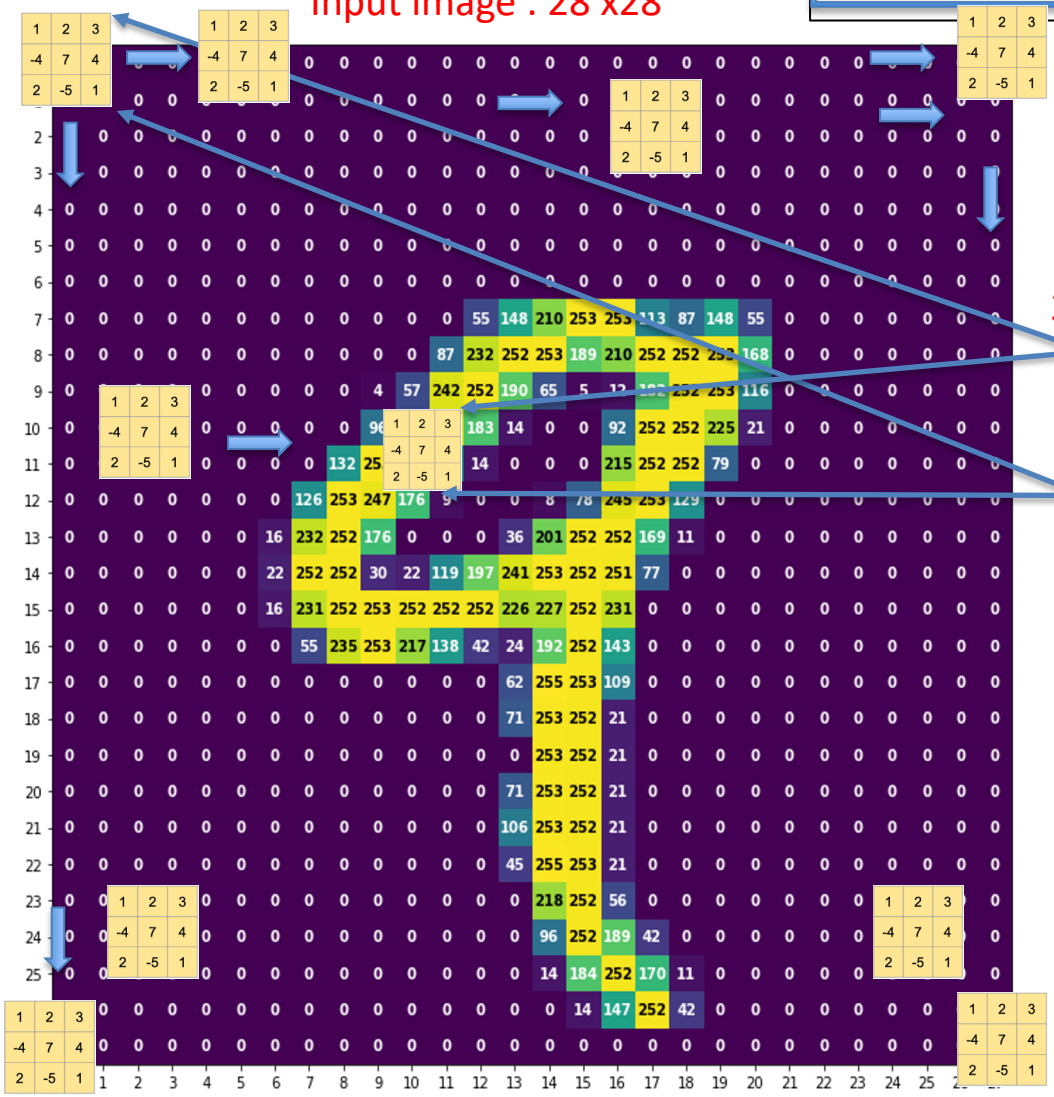
L (i) = - log0.2 = 1.61	Image 0 = 5
- log1.0 = 0	Image 1 = 0
- log0.7 = 0.356	Image 2 = 4
- log0.9 = 0.105	Image 3 = 1
- log0.9 = 0.105	Image 4 = 9
- log1.0 = 0	Image 5 = 2
- log1.0 = 0	Image 6 = 1
- log1.0 = 0	Image 7 = 3
- log0.9 = 0.105	Image 8 = 2
- log1.0 = 0	Image 9 = 4

MNIST Example (28x28 pixels image)

Handwritten digits in the MNIST dataset are 28x28 pixel grayscale images, use the original image (28x28 matrix).

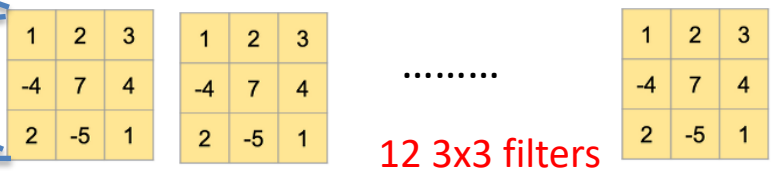


Input image : 28 x28

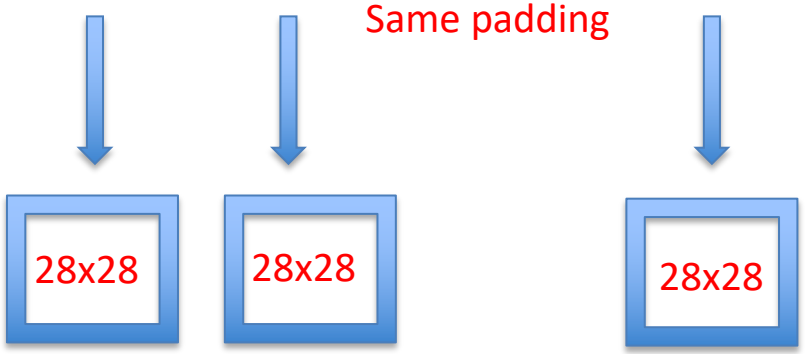


12 filters, each one acts on an image until it is fully covered.

3 x 3 filter (kernel)



12 3x3 filters
Same padding

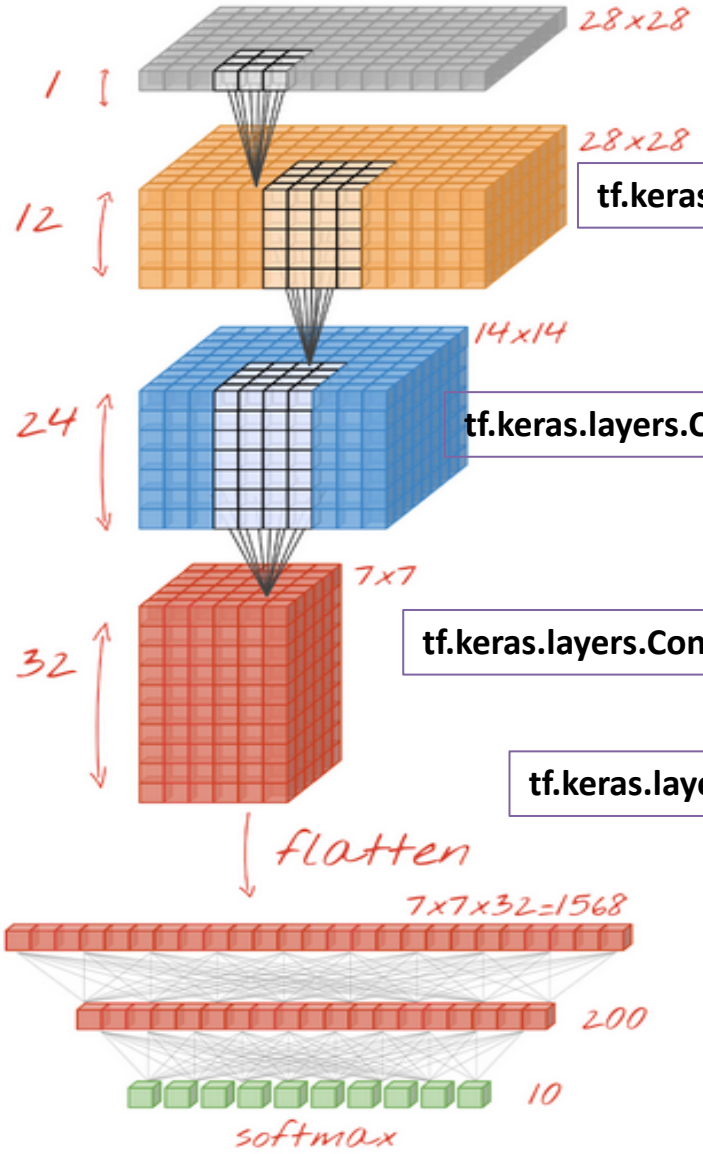


12 Output features : each 28 x28 matrix

Convolution NN : MNIST 28x28 Pixels

[FW, FH, C, H] = [filter size ? X ?, input channel (Depth), output channel (no. of filter)]

tf.keras.layers.Reshape(input_shape=(28*28,), target_shape=(28, 28, 1))



Convolutional 3x3 filters=12

$W_1[3, 3, 1, 12]$

tf.keras.layers.Conv2D(kernel_size=3, filters=12, padding='same', activation='relu')

Convolutional 6x6 filters=24

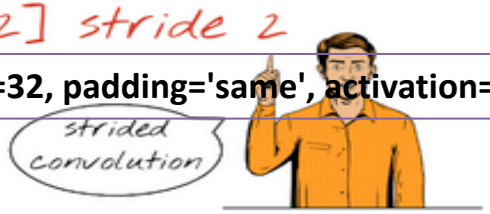
$W_2[6, 6, 12, 24]$ stride 2

tf.keras.layers.Conv2D(kernel_size=6, filters=24, padding='same', activation='relu',strides=2)

Convolutional 6x6 filters=32

$W_2[6, 6, 24, 32]$ stride 2

tf.keras.layers.Conv2D(kernel_size=6, filters=32, padding='same', activation='relu', strides=2)



tf.keras.layers.Flatten()

https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/#0

Dense layer

$W_4[1568, 200]$

tf.keras.layers.Dense(200, activation='relu')

Softmax dense layer

$W_5[200, 10]$

tf.keras.layers.Dense(10, activation='softmax')

CNN : example – MNIST

There are three main types of layers in a CNN: Convolutional Layer, Pooling Layer, and Fully-Connected Layer.

- ✓ INPUT [28x28x1] will hold the raw pixel values of the image, in this case an image of width 28, height 28 in grey scale (0-255)
- ✓ Apply twelve (12) 3x3 filters to one layer (channel) - W[3,3,Channel=1,number of filters=12]
- ✓ CONV layer will compute the output of neurons that are connected to local regions in the input, each computing a dot product between their weights and a small region they are connected to in the input volume. This may result in volume of [28x28x12] when we use 12 filters. Volume size = [28 width x height = 28 , 12 channels (depth)]
- ✓ RELU layer will apply an elementwise activation function, such as the $max(0,x)$
- ✓ Apply 24 6x6 filters to 12 layers (channel) with stride 2 - W[6,6,12,24] stride 2
- ✓ This may result in volume of [14x14x24] when we use 24 filters, stride 2.
- ✓ RELU layer will apply an elementwise activation function, such as the $max(0,x)$
- ✓ Apply 32 6x6 filters to 24 layers (channel) with stride 2 - W[6,6,24,32] stride 2
- ✓ This may result in volume of [7x7x32] when we use 32 filters, stride 2.
- ✓ RELU layer will apply an elementwise activation function, such as the $max(0,x)$
- ✓ Flatten the volume (tensor) to be a vector of 7x6x32=1568 elements
- ✓ Use 200 neurons with the input vector, Fully connected layer, W [1558, 200], output=1 x 200
- ✓ FC (i.e. fully-connected) layer will compute the class scores, resulting in volume of size [1x1x10], where each of the 10 numbers correspond to a class score.
- ✓ In this way, ConvNets transform the original image layer by layer from the original pixel values to the final class scores.

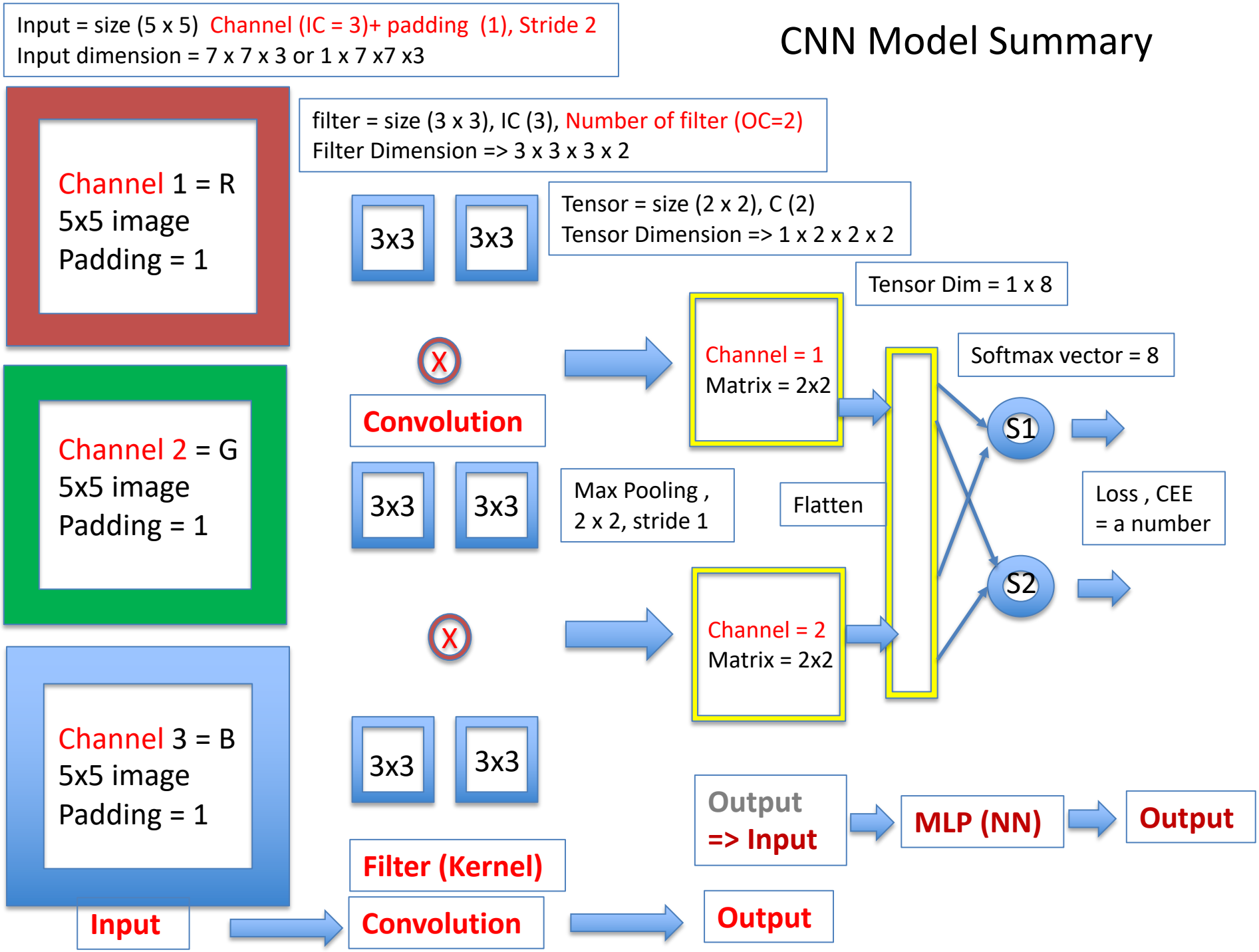
```
import tensorflow as tf
print(tf.__version__)

import datetime, os

%load_ext tensorboard

import numpy as np
from tensorflow import keras
from tensorflow.keras import layers

# Model / data parameters
num_classes = 10
input_shape = (28, 28, 1)

# the data, split between train and test sets
(x_train, y_train), (x_test, y_test) = keras.datasets.mnist.load_data()

# Scale images to the [0, 1] range
x_train = x_train.astype("float32") / 255
x_test = x_test.astype("float32") / 255
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")

# Make sure images have shape (28, 28, 1)
x_train = x_train.reshape([-1,28,28,1])
x_test = x_test.reshape([-1,28,28,1])
print("x_train shape:", x_train.shape)
print(x_train.shape[0], "train samples")
print(x_test.shape[0], "test samples")

# convert class vectors to binary class matrices
y_train = keras.utils.to_categorical(y_train, num_classes)
y_test = keras.utils.to_categorical(y_test, num_classes)
```

https://github.com/keras-team/keras-io/blob/master/examples/vision/mnist_convnet.py

https://github.com/keras-team/keras-io/blob/master/examples/vision/mnist_convnet.py

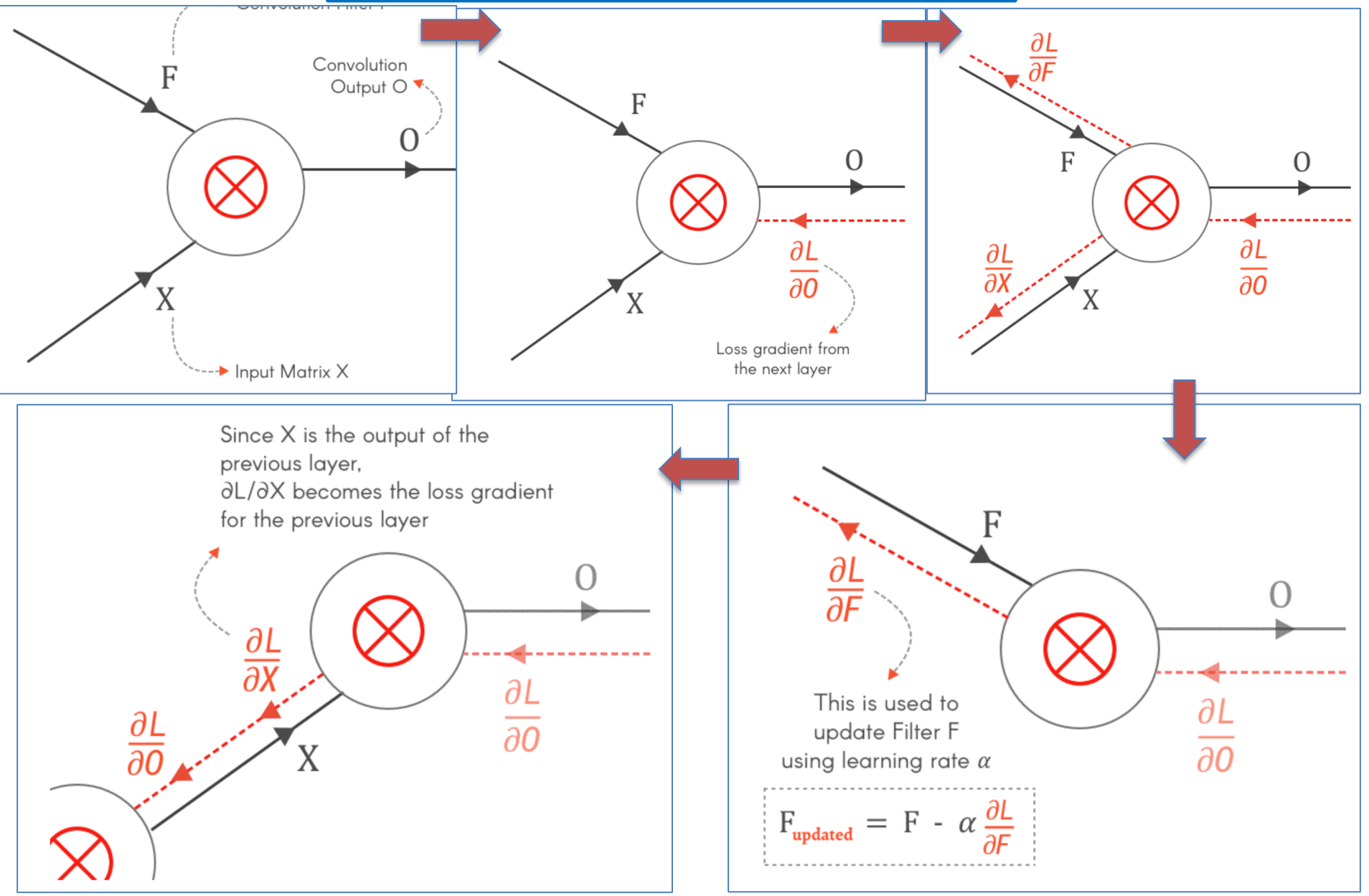
```
## Build the model
model = keras.Sequential(
  [
     keras.Input(shape=input_shape),
     layers.Conv2D(32, kernel_size=(3, 3), activation="relu"),
     layers.MaxPooling2D(pool_size=(2, 2)),
     layers.Conv2D(64, kernel_size=(3, 3), activation="relu"),
     layers.MaxPooling2D(pool_size=(2, 2)),
     layers.Flatten(),
     layers.Dropout(0.5),
     layers.Dense(num_classes, activation="softmax"),
  ]
)

model.summary()

## Train the model
batch_size = 128
epochs = 5

model.compile(loss="categorical_crossentropy", optimizer="adam", metrics=["accuracy"])

logdir = os.path.join("logs", datetime.datetime.now().strftime("%Y%m%d-%H%M%S"))
tensorboard_callback = tf.keras.callbacks.TensorBoard(logdir, histogram_freq=1)

model.fit(x_train, y_train, batch_size=batch_size, epochs=epochs, validation_split=0.1,
callbacks=[tensorboard_callback])

## Evaluate the trained model
%tensorboard --logdir logs
```

CNN Model Summary



CNN : Backward Path Calculation
A Simple Example

Forward path : backward path



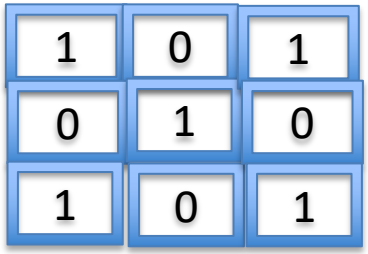
https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c

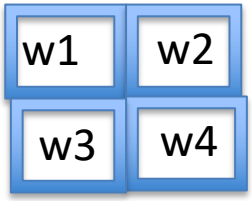
Example exercise

Compute the updated values of filter weights after backpropagation of the following problem -- 3x3 input matrix, 2x2 filter, stride 1, no padding, flatten it and use it as input connecting to a two-neuron layer, then pass the results to a softmax binary classification of two neurons.

3a) (0.5%) What is the values of the output at the end of the forward path

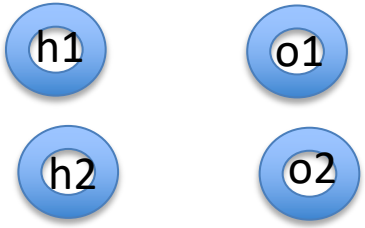
3b) (1.5%) What are the updated values of the weight and bias after backpropagation





b1 = 0.1

w1=0.9
w2=0.1
w3=0.1
w4=0.9



Apply ReLU activation function after the convolution step

Flatten the value after applying the ReLU activation function

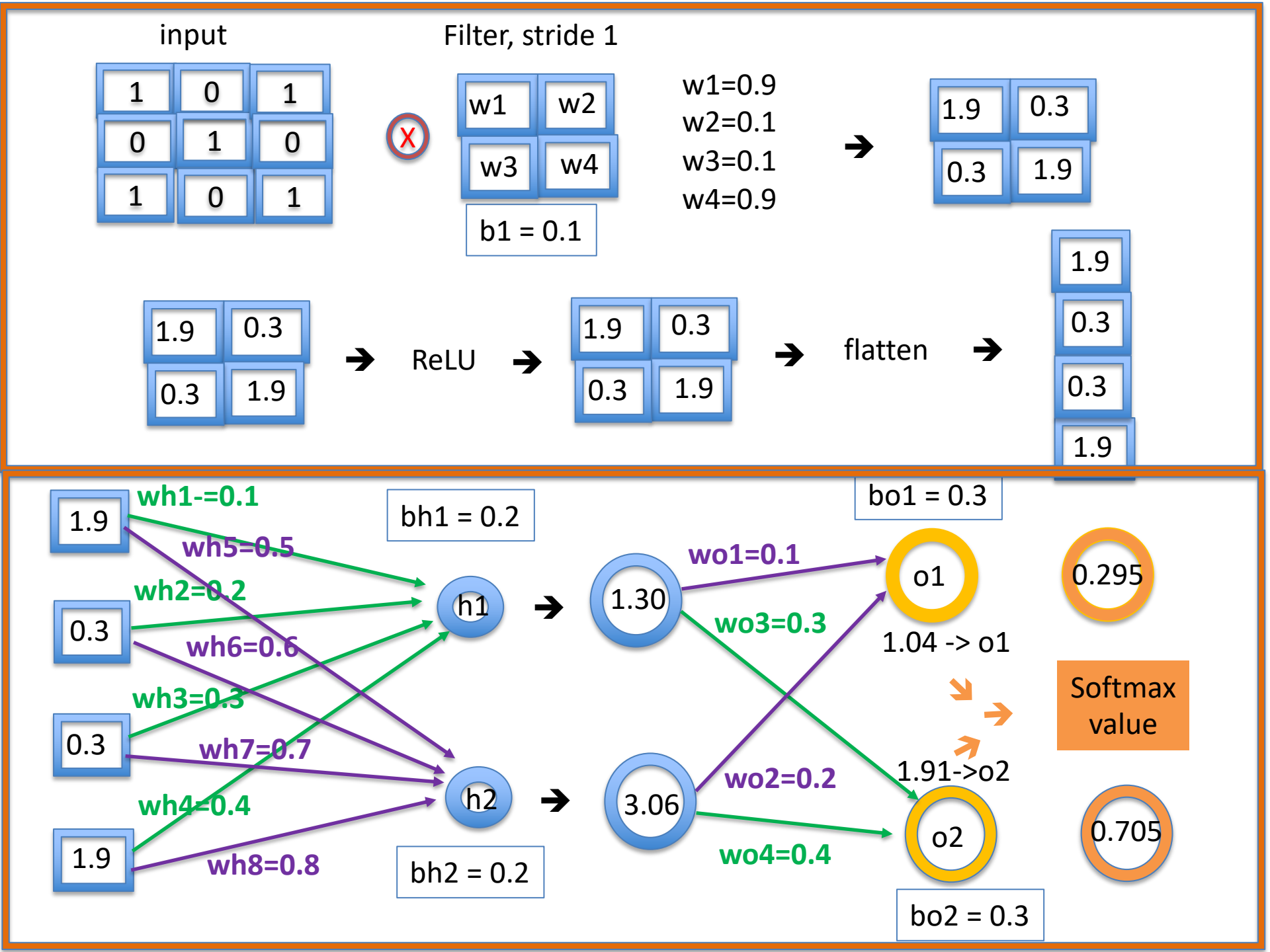
Assuming the weight of each link to this layer is w = 0.1 to 0.8, b2=0.2

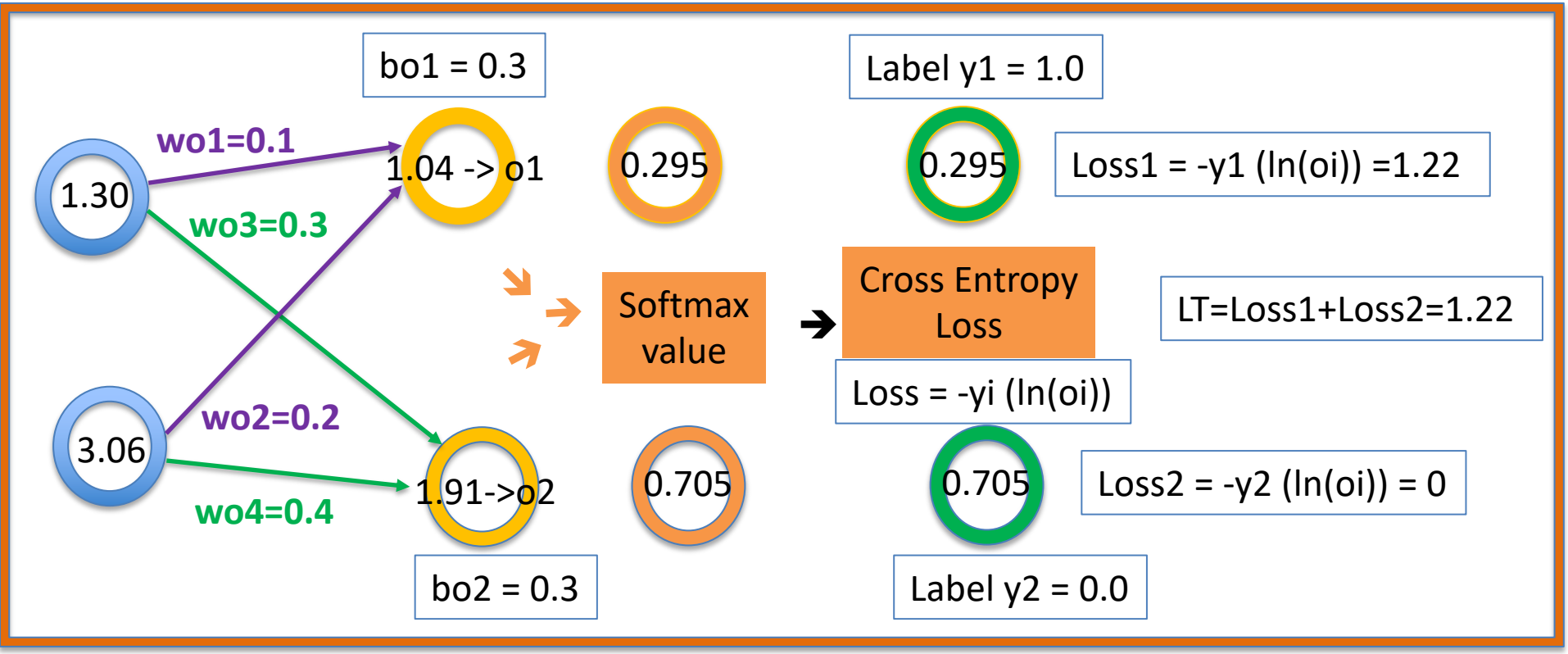
Fully connecting to a layer of two neurons

Assuming the weight of each link is w = 0.1 to 0.4, b3=0.3

Fully Connected layer of 2 softmax neurons

Labeled output of two classes are 1.0 and 0.00


input
Filter, stride 1
1 0 1
0 1 0
1 0 1
X
w1 w2
w3 w4
b1 = 0.1
w1=0.9
w2=0.1
w3=0.1
w4=0.9
1.9 0.3
0.3 1.9
1.9 0.3
0.3 1.9
ReLU
1.9 0.3
0.3 1.9
flatten
1.9
0.3
0.3
1.9
wh1-=0.1
wh5=0.5
wh2=0.2
wh6=0.6
wh3=0.3
wh7=0.7
wh4=0.4
wh8=0.8
1.9
0.3
0.3
1.9
bh1 = 0.2
h1
h2
bh2 = 0.2
1.30
3.06
wo1=0.1
wo3=0.3
wo2=0.2
wo4=0.4
bo1 = 0.3
o1
o2
bo2 = 0.3
1.04 -> o1
1.91->o2
0.295
0.705
Softmax value



CNN : Backward Path – jumps over Cross Entropy and Softmax all together

Learning rate = n = 0.5

$$\frac{\partial E}{\partial Z_k} = O_k - y_k$$

Z = Z(w, b)

S = S(z)

weights bias

w+ = w – n * (dE/dw)
dE/dw = (dE/dS) * (dS/dw)

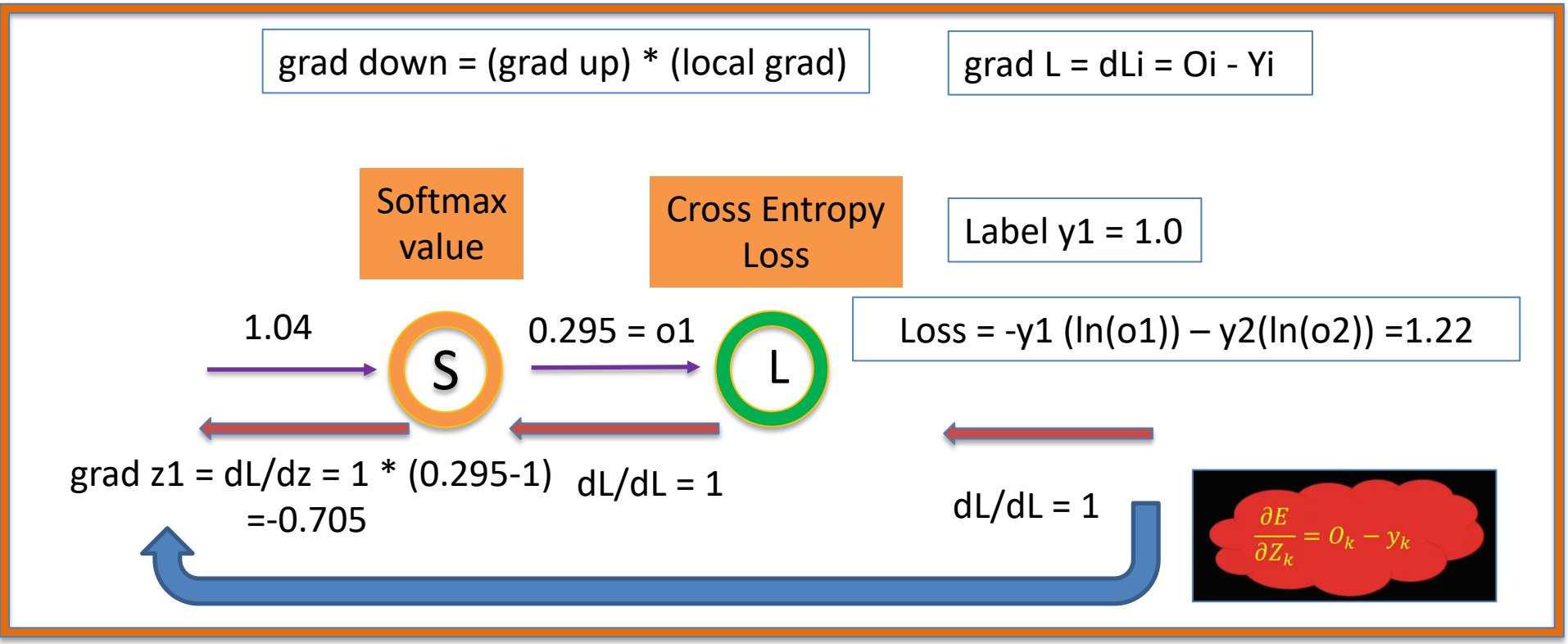
Softmax value

dL/dS

Cross Entropy Loss (L)

dL/dL = 1

Flow Back from the Total Loss to Softmax to Z (S1)



Jump from cross entropy and softmax (E , S) to (weight and bias) Z (o1), Y1 =1

E=L ; dE/dz = dL/dz = (dL/ds * ds/dz)

dE/dZ = local grad = O1 – Y1

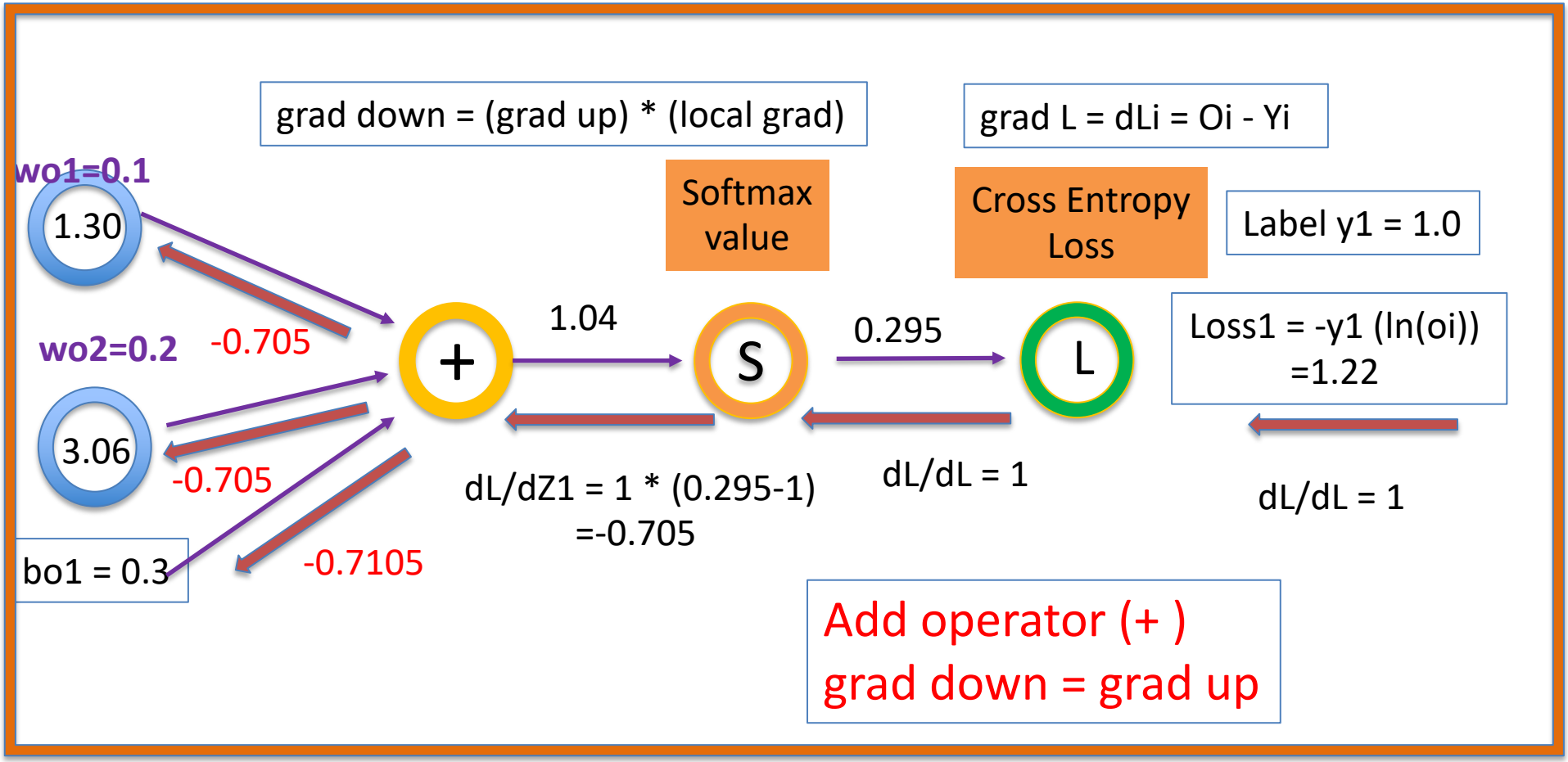
grad Z1 = grad down = (grad up * local grad)

= 1 * (0.295-1) = -0.705

$$\frac{\partial E}{\partial Z_k} = O_k - y_k$$

Flow Back from Cross Entropy and Softmax to W and b

L -> L1 -> S1 (o1) -> wo1, wo2, bo1



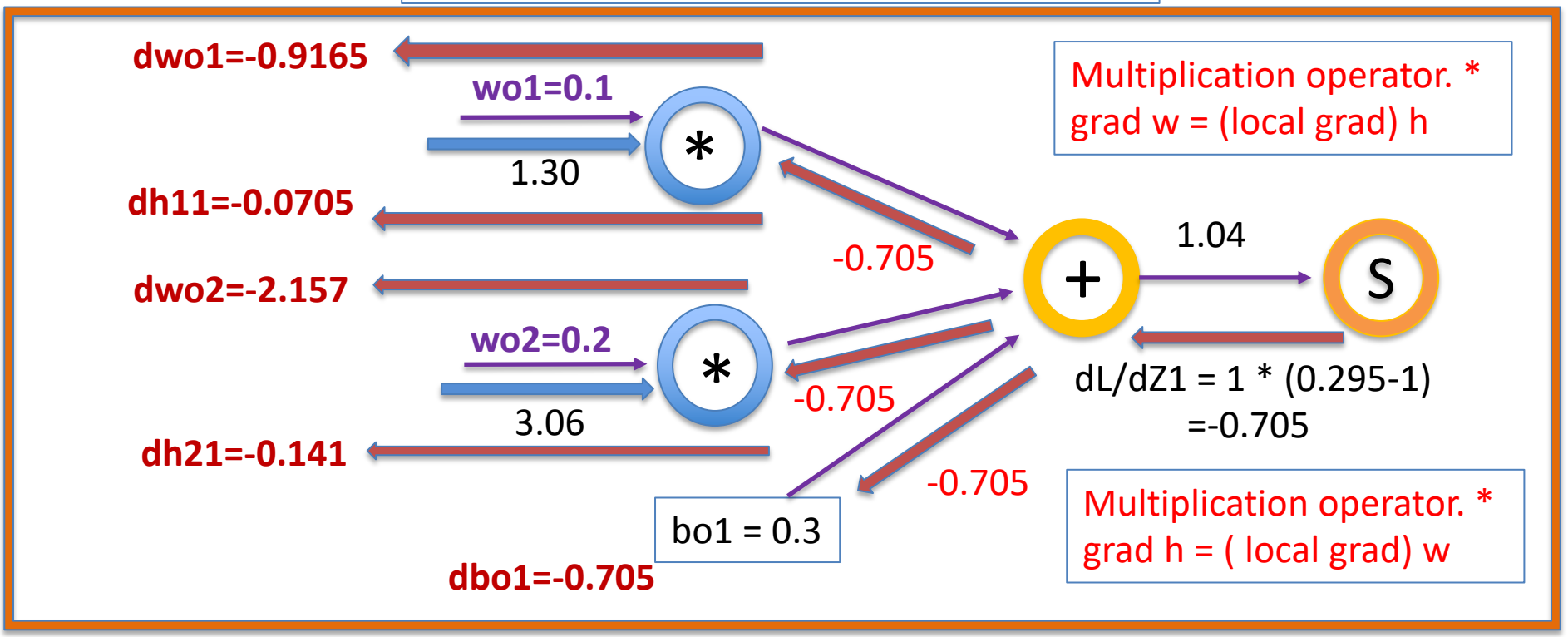
Multiplication operator. (*)

grad w = (local grad) h

Multiplication operator. (*)

grad h = (local grad) w

Update values of wo1, wo2, bo1



In here, i =1 (o1),
j = 1,2: h1, h2

d (L/dwo1) = dL/d(w11)
= -0.705*1.30 = -0.9165

d (L/dwo2) = dL/d(w12)
= -0.705*3.06= -2.1573

d (L/dbo1) = dL/d(b1)
= dL/dS = Oi-Yi=-0.705

wo1+= 0.1– (0.5)*(-0.9165) =0.558

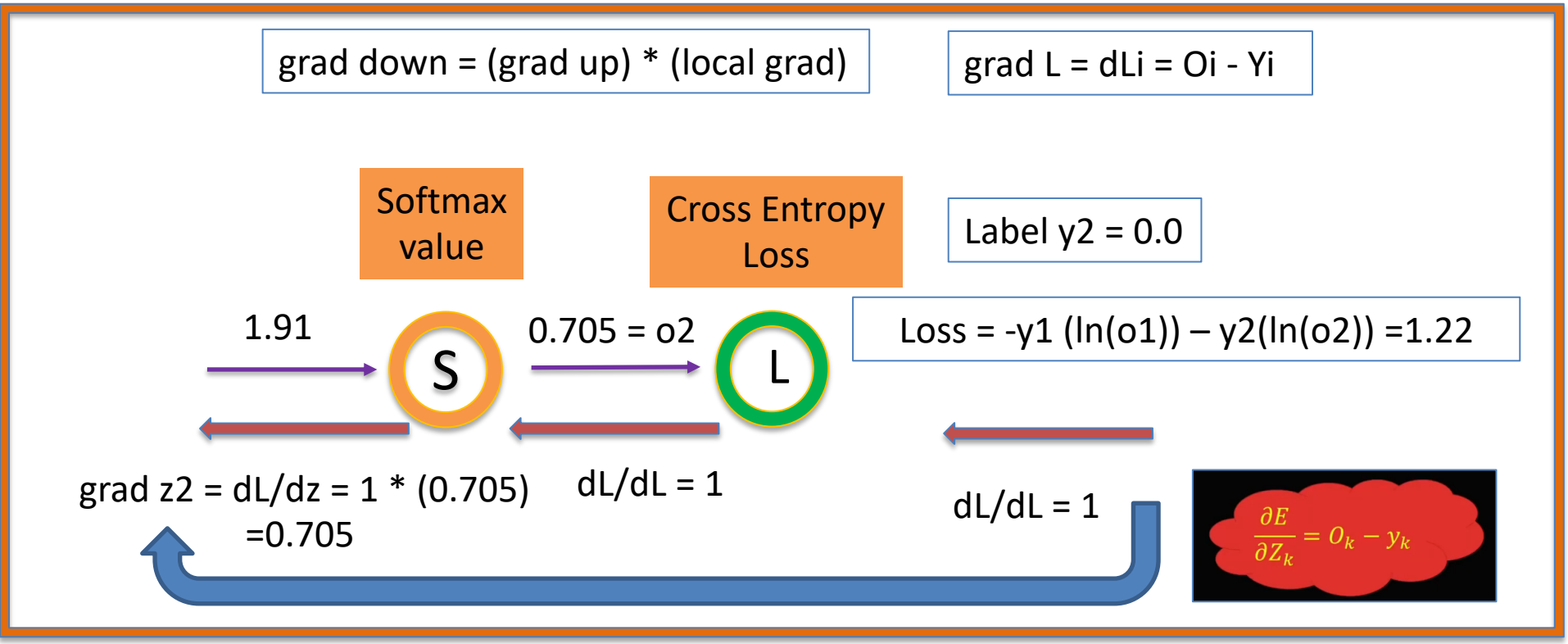
wo2+= 0.2– (0.5)*(-2.1573) =1.278

bo1+= 0.3– (0.5)*(-0.705) =0.652

dh11= (-0.0705)

dh21= (-0.141)

Flow Back from the CEE Loss to Softmax to Z : S2



Jump from cross entropy and softmax (E, S) to weight and bias Z (o2), Y2=0

E=L ; dE/dz = dL/dz = (dL/ds * ds/dz)

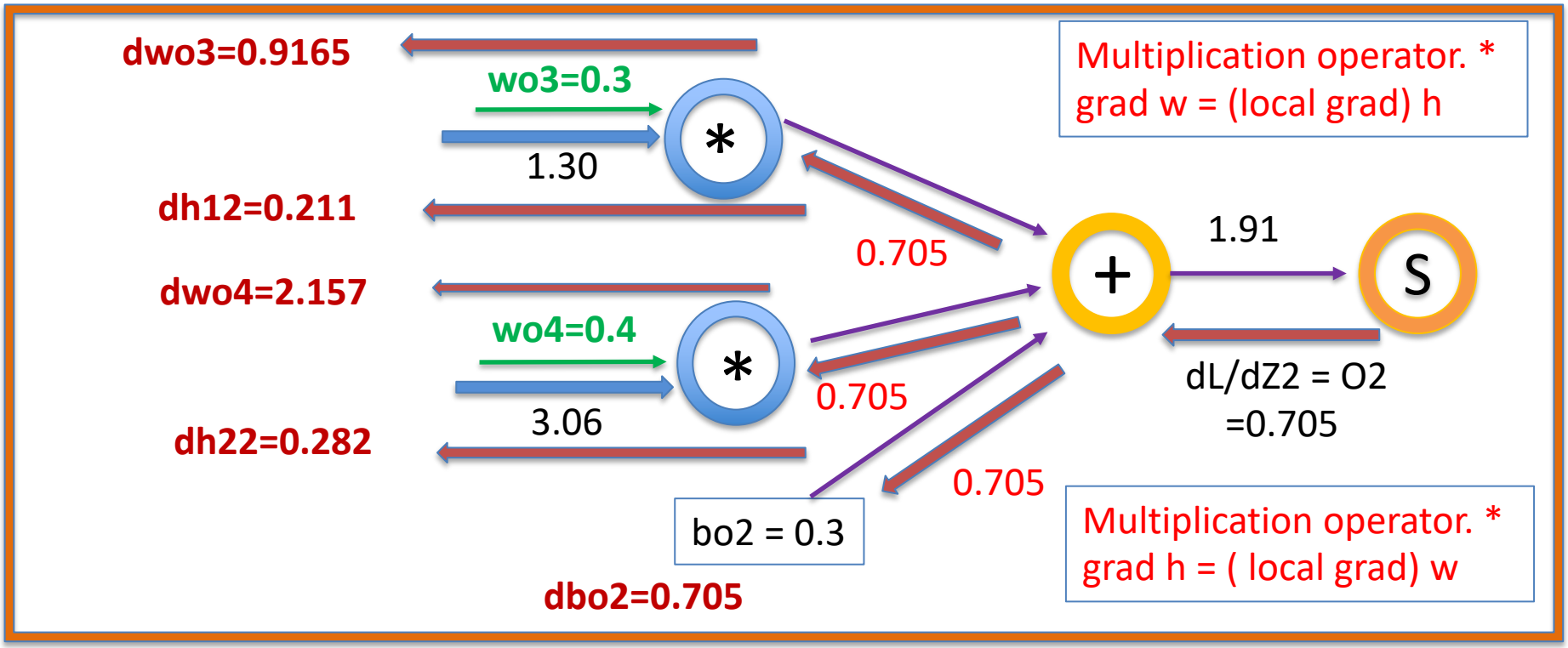
dL/dz = local grad = O2 – Y2 = O2

grad Z2 = grad down = (grad up * local grad)

= 1 * (0.705) = 0.705

$$\frac{\partial E}{\partial Z_k} = O_k - y_k$$

Update values of wo3, wo4, bo2



d (L/dwo3) = dL/d(w21)
= 0.705*1.30 = 0.9165

In here , i =2 (o2)
j = 1,2: h1, h2

d (L/dwo4) = dL/d(w22)
= 0.705*3.06 = 2.157

d (L/dbo1) = dL/d(b1)
= dL/dS = Oi-Yi=0.705

wo3+= 0.3– (0.5)*(0.9165) =-0.158

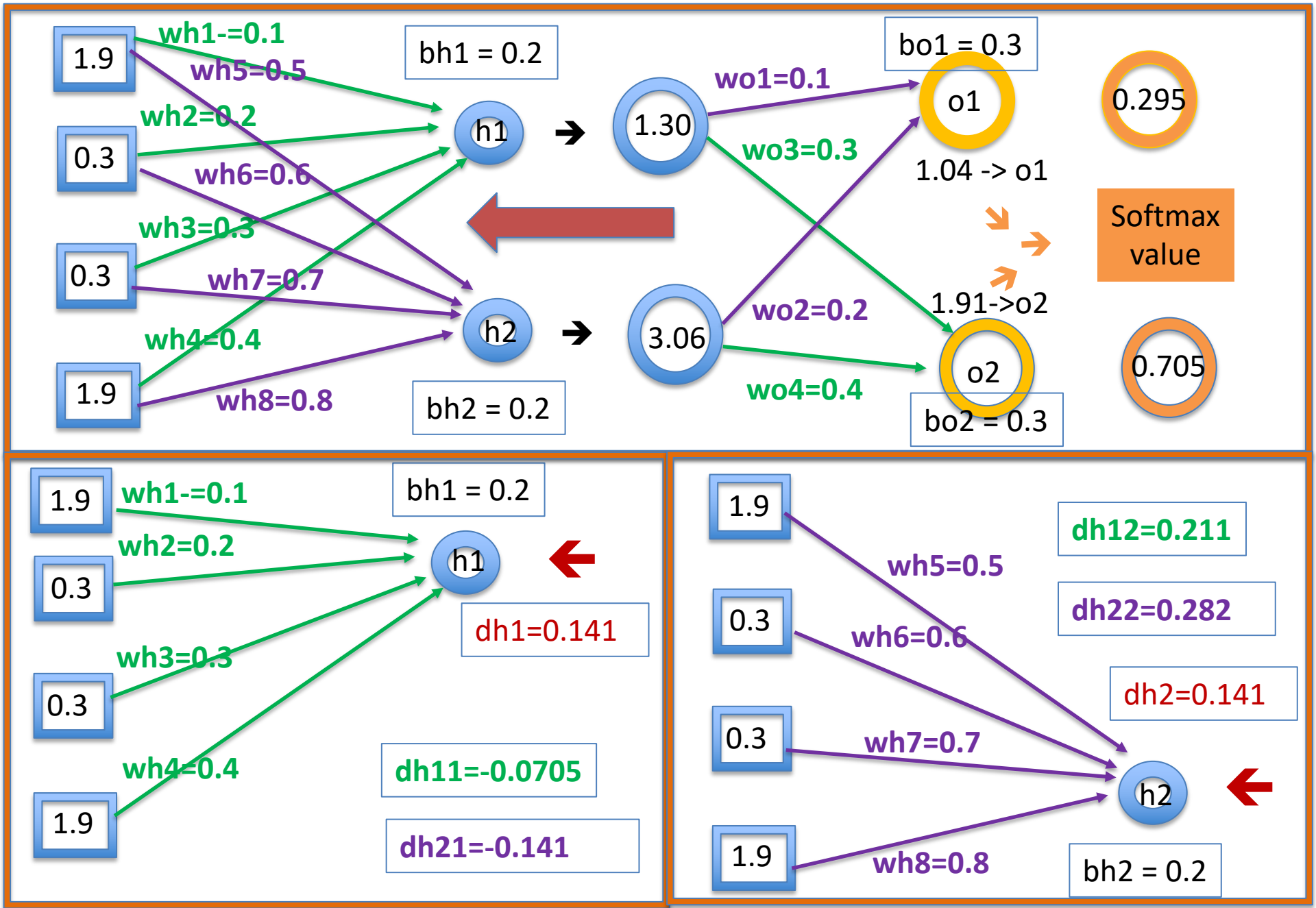
wo4+= 0.4– (0.5)*(2.157) =-0.678

bo2+= 0.3– (0.5)*(0.705) =-0.352

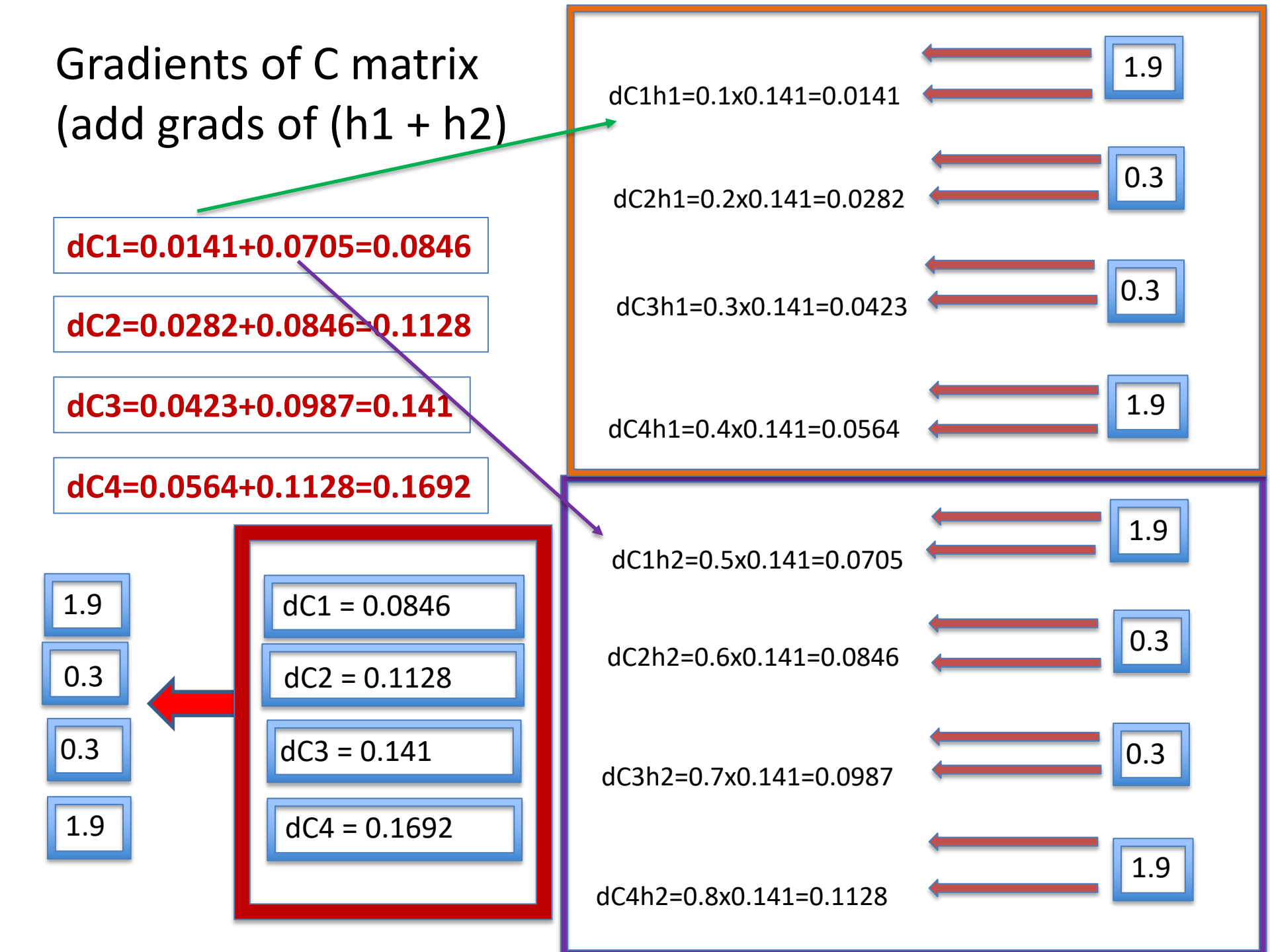
dh12= (0.211)

dh22= (0.282)

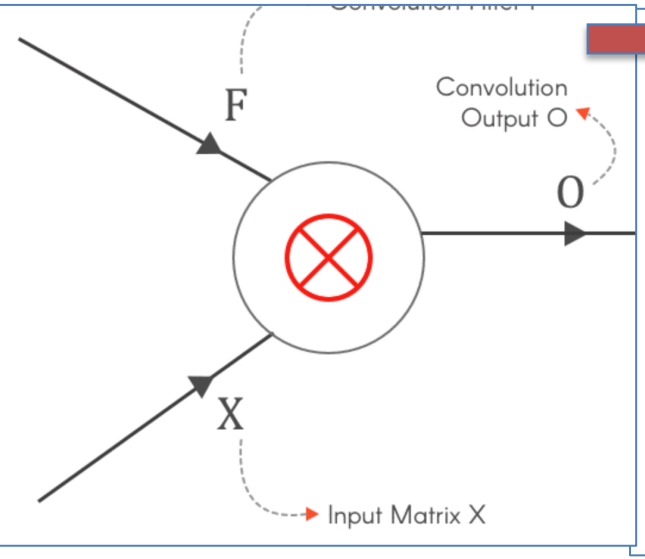
Flow back from o1 to the hidden NN layer (h) , MLP calculation

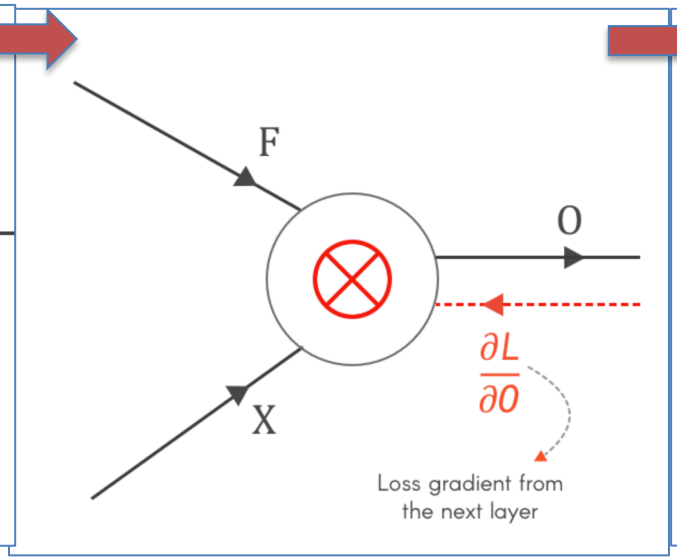


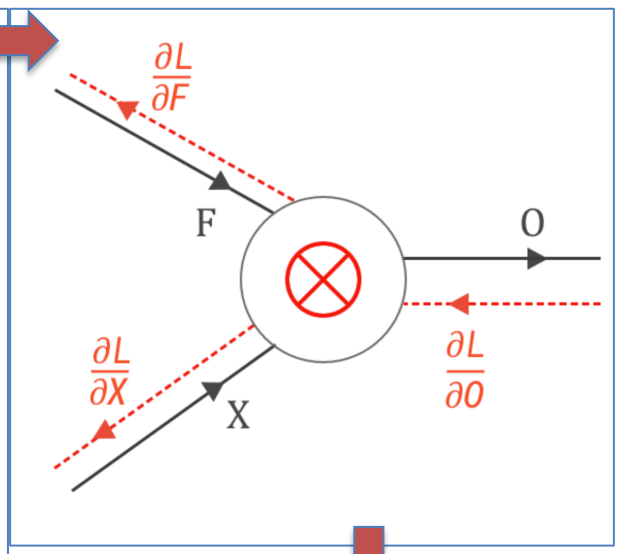
Gradients of C matrix (add grads of (h1 + h2)

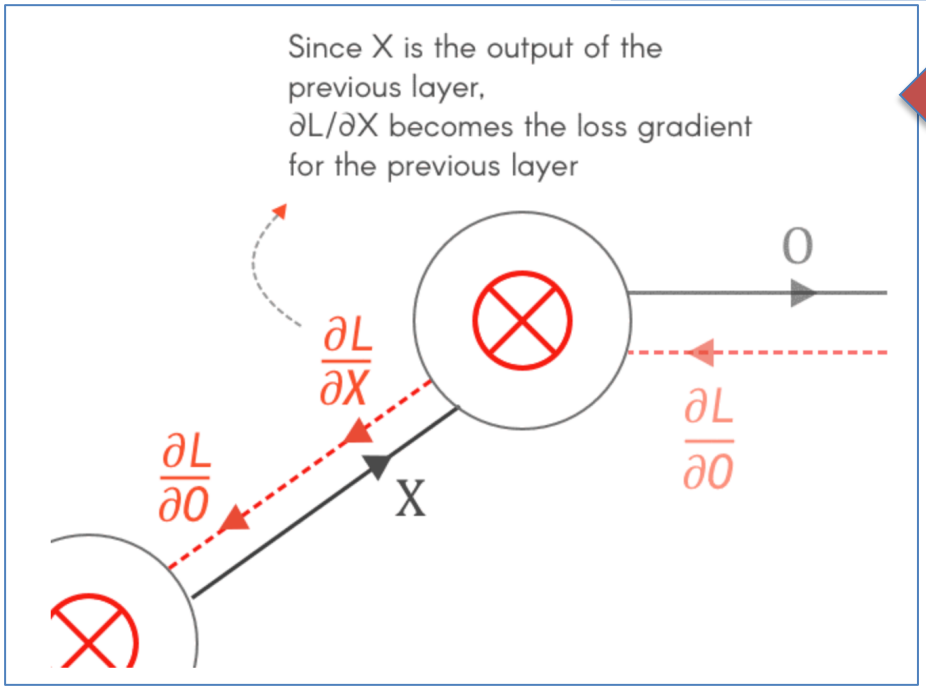


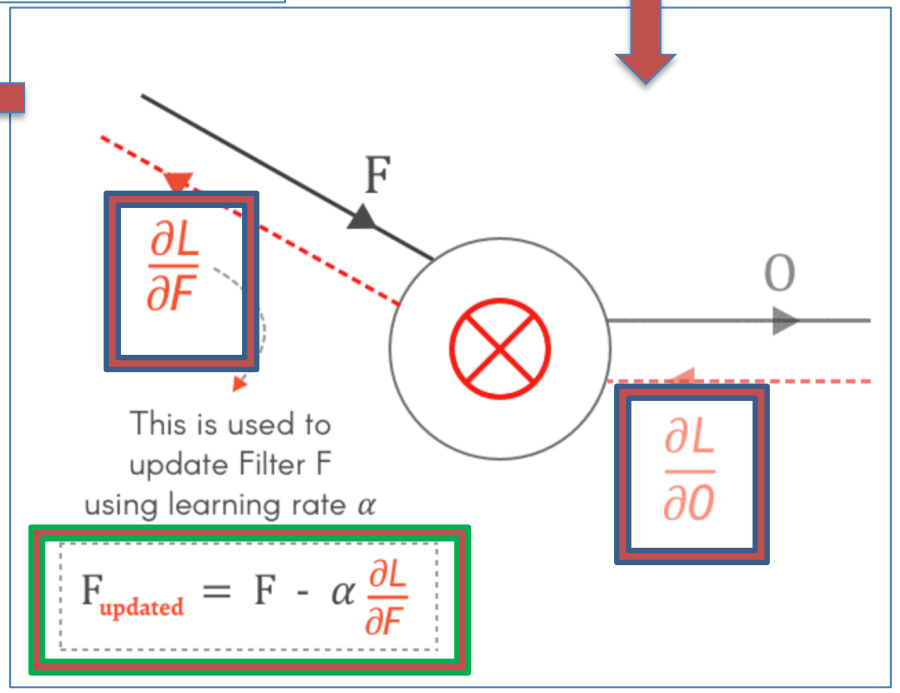
Convolution : Forward path : backward path











https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c

Local Gradients → (A)

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

Finding derivatives with respect to F_{11}, F_{12}, F_{21} and F_{22}

$$\frac{\partial O_{11}}{\partial F_{11}} = X_{11} \quad \frac{\partial O_{11}}{\partial F_{12}} = X_{12} \quad \frac{\partial O_{11}}{\partial F_{21}} = X_{21} \quad \frac{\partial O_{11}}{\partial F_{22}} = X_{22}$$

Similarly, we can find the local gradients for O_{12}, O_{21} and O_{22}

$$\frac{\partial L}{\partial F_{11}} = \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{11}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{11}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{11}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{11}}$$

$$\frac{\partial L}{\partial F_{12}} = \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{12}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{12}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{12}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{12}}$$

$$\frac{\partial L}{\partial F_{21}} = \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{21}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{21}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{21}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{21}}$$

$$\frac{\partial L}{\partial F_{22}} = \frac{\partial L}{\partial O_{11}} * \frac{\partial O_{11}}{\partial F_{22}} + \frac{\partial L}{\partial O_{12}} * \frac{\partial O_{12}}{\partial F_{22}} + \frac{\partial L}{\partial O_{21}} * \frac{\partial O_{21}}{\partial F_{22}} + \frac{\partial L}{\partial O_{22}} * \frac{\partial O_{22}}{\partial F_{22}}$$

a convolution operation between input X and loss gradient ∂L/∂O

$$\frac{\partial L}{\partial F_{11}} = \frac{\partial L}{\partial O_{11}} * X_{11} + \frac{\partial L}{\partial O_{12}} * X_{12} + \frac{\partial L}{\partial O_{21}} * X_{21} + \frac{\partial L}{\partial O_{22}} * X_{22}$$

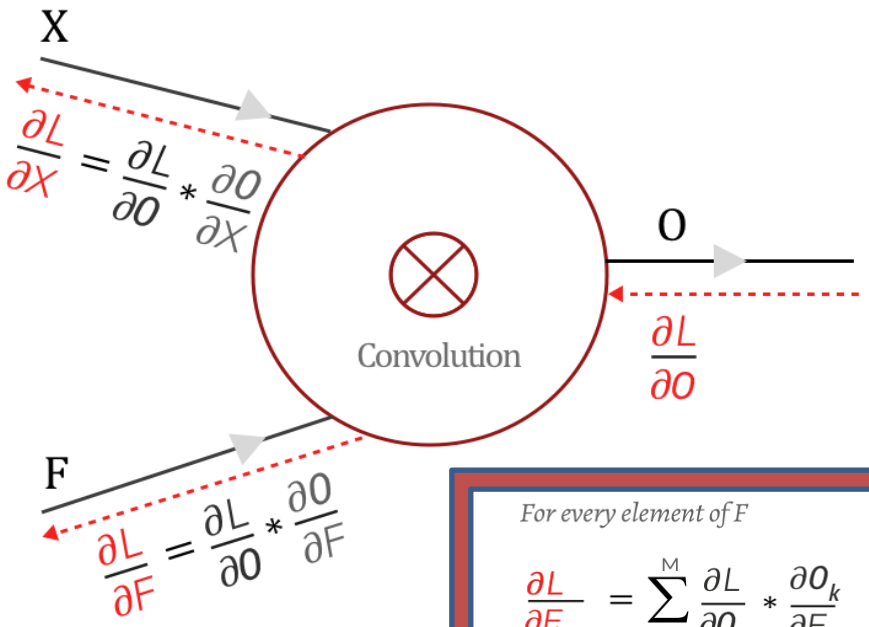
$$\frac{\partial L}{\partial F_{12}} = \frac{\partial L}{\partial O_{11}} * X_{12} + \frac{\partial L}{\partial O_{12}} * X_{13} + \frac{\partial L}{\partial O_{21}} * X_{22} + \frac{\partial L}{\partial O_{22}} * X_{23}$$

$$\frac{\partial L}{\partial F_{21}} = \frac{\partial L}{\partial O_{11}} * X_{21} + \frac{\partial L}{\partial O_{12}} * X_{22} + \frac{\partial L}{\partial O_{21}} * X_{31} + \frac{\partial L}{\partial O_{22}} * X_{32}$$

$$\frac{\partial L}{\partial F_{22}} = \frac{\partial L}{\partial O_{11}} * X_{22} + \frac{\partial L}{\partial O_{12}} * X_{23} + \frac{\partial L}{\partial O_{21}} * X_{32} + \frac{\partial L}{\partial O_{22}} * X_{33}$$

2x2 filter

https://medium.com/@pavisj/convolutions-and-backpropagations-46026a8f5d2c



For every element of F

$$\frac{\partial L}{\partial F_i} = \sum_{k=1}^{M} \frac{\partial L}{\partial O_k} * \frac{\partial O_k}{\partial F_i}$$

$\frac{\partial O}{\partial X}$ & $\frac{\partial O}{\partial F}$ *are local gradients*

$\frac{\partial L}{\partial z}$ *is the loss from the previous layer which has to be backpropagated to other layers*

$$\frac{\partial L}{\partial F} = \frac{\partial L}{\partial O} * \frac{\partial O}{\partial F}$$

- Gradient to update Filter F
- Loss Gradient from previous layer
- Local Gradients

$$\begin{bmatrix} \frac{\partial L}{\partial F_{11}} & \frac{\partial L}{\partial F_{12}} \\ \frac{\partial L}{\partial F_{21}} & \frac{\partial L}{\partial F_{22}} \end{bmatrix} = \text{Convolution} \left(\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix}, \begin{bmatrix} \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \end{bmatrix} \right)$$

∂L/∂F is nothing but the convolution between Input X and Loss Gradient from the next layer ∂L/∂O

where

$$\begin{bmatrix} X_{11} & X_{12} & X_{13} \\ X_{21} & X_{22} & X_{23} \\ X_{31} & X_{32} & X_{33} \end{bmatrix} = \text{Input X} \qquad \begin{bmatrix} \frac{\partial L}{\partial O_{11}} & \frac{\partial L}{\partial O_{12}} \\ \frac{\partial L}{\partial O_{21}} & \frac{\partial L}{\partial O_{22}} \end{bmatrix} = \frac{\partial L}{\partial O}$$ Loss gradient from previous layer

Local Gradients: B

2x2 filter

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22}$$

Differentiating with respect to X_{11}, X_{12}, X_{21} *and* X_{22}

$$\frac{\partial O_{11}}{\partial X_{11}} = F_{11} \quad \frac{\partial O_{11}}{\partial X_{12}} = F_{12} \quad \frac{\partial O_{11}}{\partial X_{21}} = F_{21} \quad \frac{\partial O_{11}}{\partial X_{22}} = F_{22}$$

Similarly, we can find local gradients for O_{12}, O_{21} *and* O_{22}

For every element of X_i

$$\frac{\partial L}{\partial X_i} = \sum_{k=1}^{M} \frac{\partial L}{\partial O_k} * \frac{\partial O_k}{\partial X_i}$$

$$\frac{\partial L}{\partial X_{11}} = \frac{\partial L}{\partial O_{11}} * F_{11}$$

$$\frac{\partial L}{\partial X_{12}} = \frac{\partial L}{\partial O_{11}} * F_{12} + \frac{\partial L}{\partial O_{12}} * F_{11}$$

$$\frac{\partial L}{\partial X_{13}} = \frac{\partial L}{\partial O_{12}} * F_{12}$$

$$\frac{\partial L}{\partial X_{21}} = \frac{\partial L}{\partial O_{11}} * F_{21} + \frac{\partial L}{\partial O_{21}} * F_{11}$$

$$\frac{\partial L}{\partial X_{22}} = \frac{\partial L}{\partial O_{11}} * F_{22} + \frac{\partial L}{\partial O_{12}} * F_{21} + \frac{\partial L}{\partial O_{21}} * F_{12} + \frac{\partial L}{\partial O_{22}} * F_{11}$$

$$\frac{\partial L}{\partial X_{23}} = \frac{\partial L}{\partial O_{12}} * F_{22} + \frac{\partial L}{\partial O_{22}} * F_{12}$$

$$\frac{\partial L}{\partial X_{31}} = \frac{\partial L}{\partial O_{21}} * F_{21}$$

$$\frac{\partial L}{\partial X_{32}} = \frac{\partial L}{\partial O_{21}} * F_{22} + \frac{\partial L}{\partial O_{22}} * F_{21}$$

$$\frac{\partial L}{\partial X_{33}} = \frac{\partial L}{\partial O_{22}} * F_{22}$$

Backpropagation in a Convolutional Layer of a CNN

Finding the gradients:

$$\frac{\partial L}{\partial F} = \text{Convolution}\left(\text{Input } X,\ \text{Loss gradient } \frac{\partial L}{\partial O}\right)$$

$$\frac{\partial L}{\partial X} = \text{Full Convolution}\left(180^{\circ}\text{rotated Filter } F,\ \text{Loss Gradient } \frac{\partial L}{\partial O}\right)$$

Loss gradient = dL/dC

dC1=0.0846 | dC2=0.1128

dC3=0.141 | dC4=0.1692

1	0	1
0	1	0
1	0	1

X

dC1=0.0846	dC2=0.1128
dC3=0.141	dC4=0.1692

=

dL/dF

dF1=0.2538	dF2=0.2538
dF3=0.2538	dF4=0.238

dF1 = 1 x dC1 + 0 x dC2 + 0 x dC3 + 1 x dC4 = 0.5238

dF2 = 0 x dC1 + 1 x dC2 + 1 x dC3 + 0 x dC4 = 0.2538

dF3 = 0 x dC1 + 1 x dC2 + 1 x dC3 + 0 x dC4 = 0.2538

dF4 = 1 x dC1 + 0 x dC2 + 0 x dC3 + 1 x dC4 = 0.2538

F1+=0.7731 | F2+=-0.027

F3+=-0.027 | F4+=0.7731

=

0.9	0.1
0.1	0.9

- (0.5) *

dF1=0.2538	dF2=0.2538
dF3=0.2538	dF4=0.2538

Update values of b1 – bias of convolution

Local Gradients → (A)

$$O_{11} = X_{11}F_{11} + X_{12}F_{12} + X_{21}F_{21} + X_{22}F_{22} + b$$

Finding derivatives with respect to F_{11}, F_{12}, F_{21} and F_{22}

$$\frac{\partial O_{11}}{\partial F_{11}} = X_{11} \quad \frac{\partial O_{11}}{\partial F_{12}} = X_{12} \quad \frac{\partial O_{11}}{\partial F_{21}} = X_{21} \quad \frac{\partial O_{11}}{\partial F_{22}} = X_{22}$$

Similarly, we can find the local gradients for O_{12}, O_{21} and O_{22}

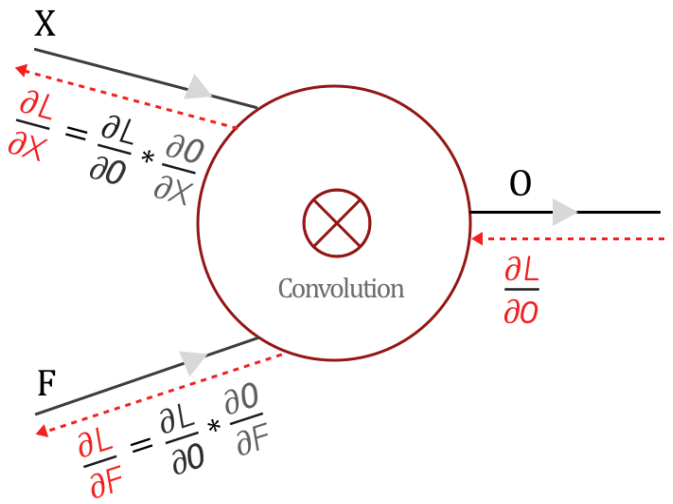
Loss gradient = dL/dC

dC1=0.0846

dC2=0.1128

dC3=0.141

dC4=0.1692



$\frac{\partial O}{\partial X}$ & $\frac{\partial O}{\partial F}$ are local gradients

$\frac{\partial L}{\partial z}$ is the loss from the previous layer which has to be backpropagated to other layers

dL/db = (dL/dO) * (dO/db)

O11 = X11F11 + X12*F12 + X21*F21 + X22*F22 +b1

So dO/db = 1 ; O = C in this case

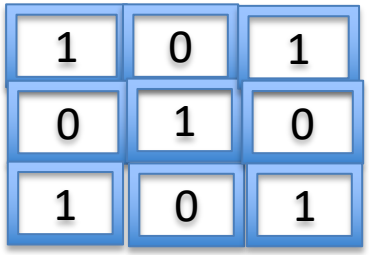
$dL/db_i = (dL/dC) * (dO/db) = (dL/dC)_i$

Add each contribution together

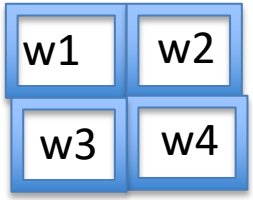
grad b = dC1 + dC2 + dC3 + dC4

grad b = 0.0846+0.1128+0.141+0.1692 = 0.5076

b1+ = b – 0.5 * 0.5076 = 0.1 – 0.2538 = - 0.1538



X



I1 = new data | I2 = new data | I3 = new data

I4 = new data | I5 = new data | I6 = new data

I7 = new data | I8 = new data | I9 = new data

X

F1+=0.7731 | F2+=-0.027

F3+=-0.027 | F4+=0.7731

w1+=F1+=0.7731

w2+=F2+=-0.027

w3+=F3+=-0.027

w4+=F4+=0.7731

b1+=b1-0.5 * 0.5076
= 0.1 – 0.5 * 0.5076
= -0.1538

wh1+=0.1-0.5x0.268=-0.034

wh2+=0.2-0.5x0.042=0.179

wh3+=0.3-0.5x0.042=0.279

wh4+=0.4-0.5x0.268=0.266

wh5+=0.5-0.5x0.268=0.364

wh6+=0.6-0.5x0.042=0.579

wh7+=0.7-0.5x0.042=0.679

wh8+=0.8-0.5x0=268-0.666

bh1+=0.2-0.5x0.1428=0.1295

bh2+=0.2-0.5x0.1428=0.1295

wo1+= 0.1– (0.5)*(-0.9996) =0.558

wo2+= 0.2– (0.5)*(-2.256) =1.278

wo3+= 0.3– (0.5)*(0.9995) =-0.158

wo4+= 0.4– (0.5)*(2.256) =-0.7678

bo1+= 0.3– (0.5)*(-0.705) =0.625

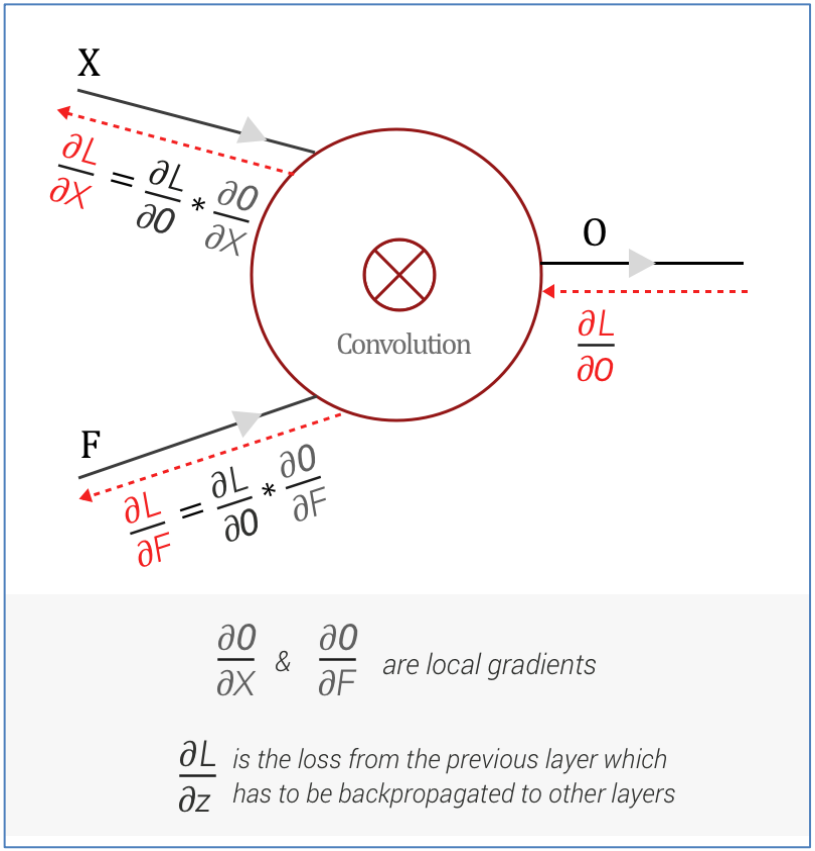
bo2+= 0.3– (0.5)*(0.705) =-0.0525

bo+= (bo1+bo2)/2 =
(-0.057 + 0.657) /. 2 = 0.3 !!

Exercise

CNN Backward Path Calculation

Given the following input, filter, and dL/dO matrices.
a) Compute the matrix dL/dF
b) Compute the matrix dL/dX



Input 3 x3

1	0	-1
1	0	1
0	1	1

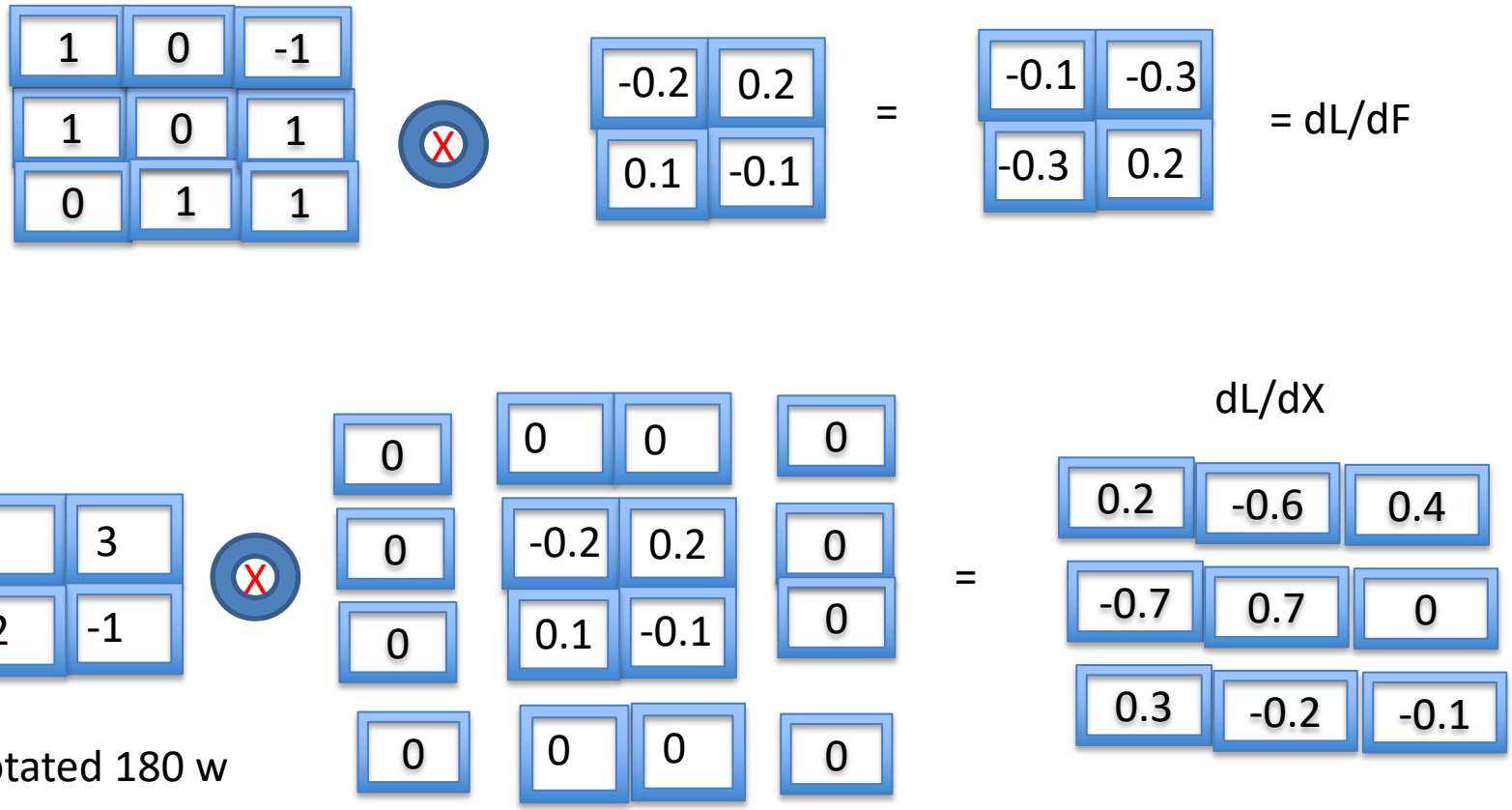
Filter = 2 x 2

-1	2
3	1

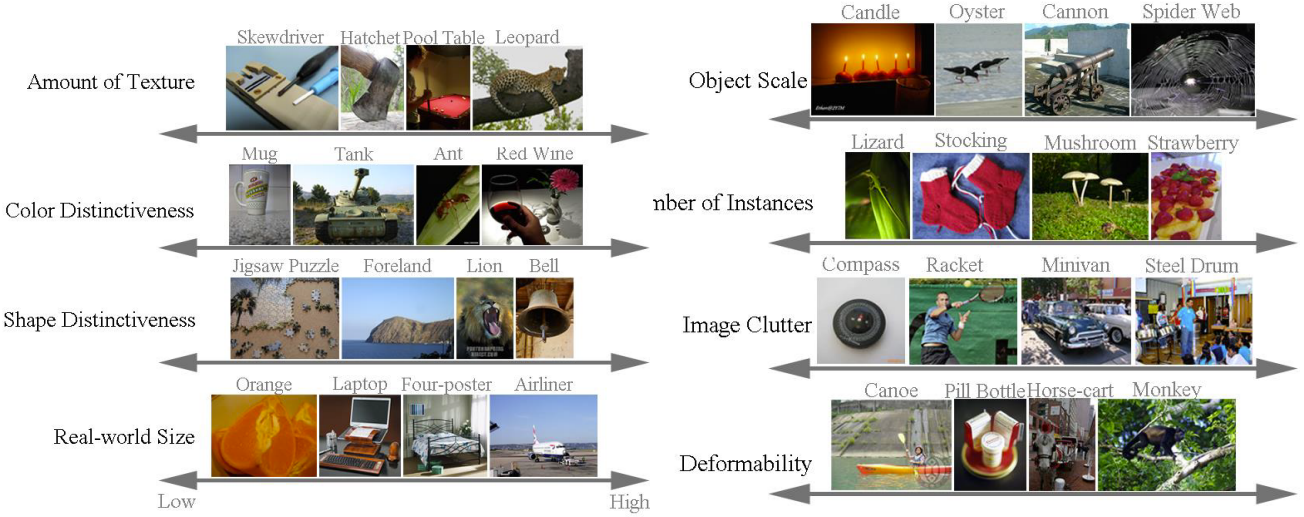
dL/dO =

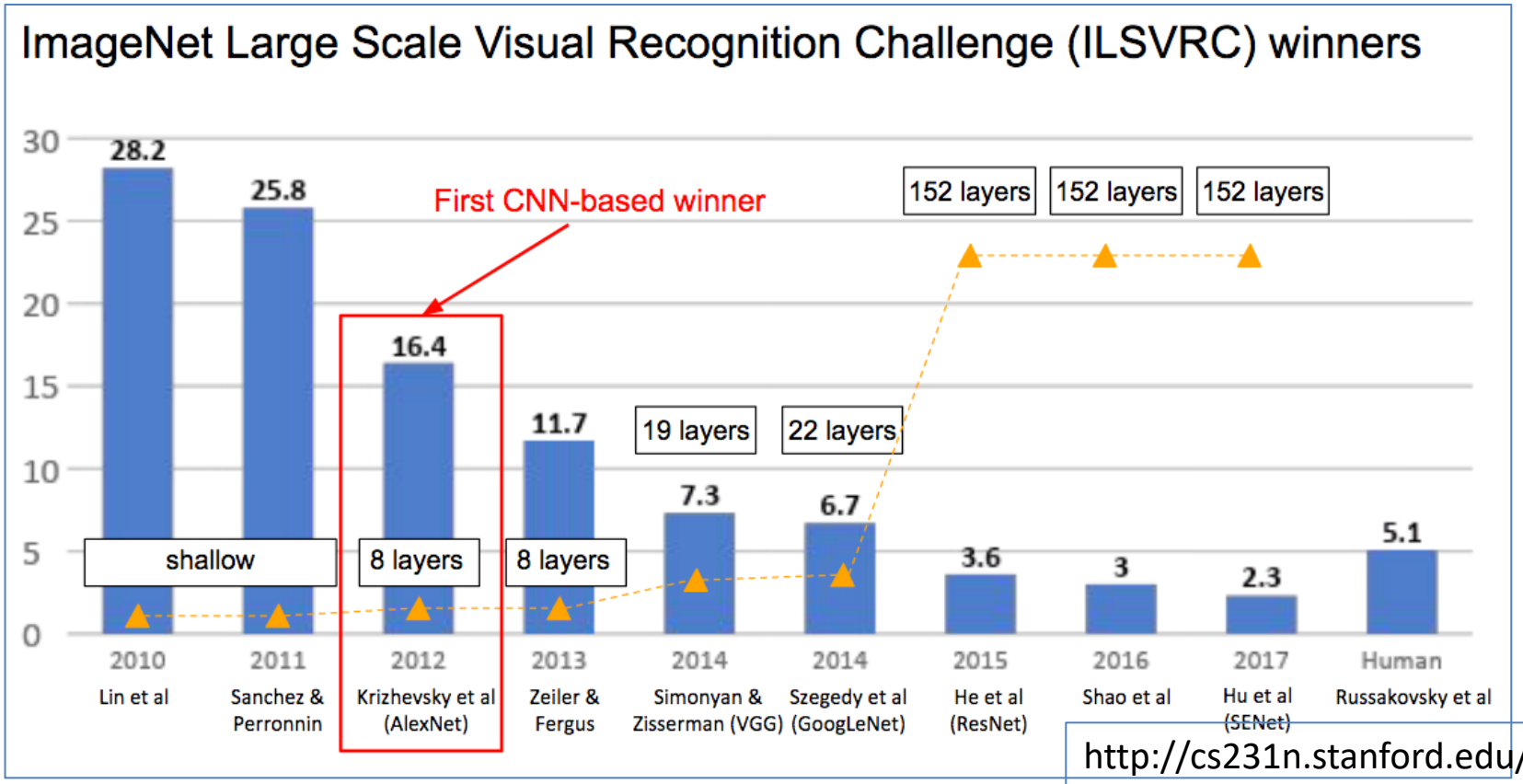
-0.2	0.2
0.1	-0.1

CNN Backward Path Calculation - Answers

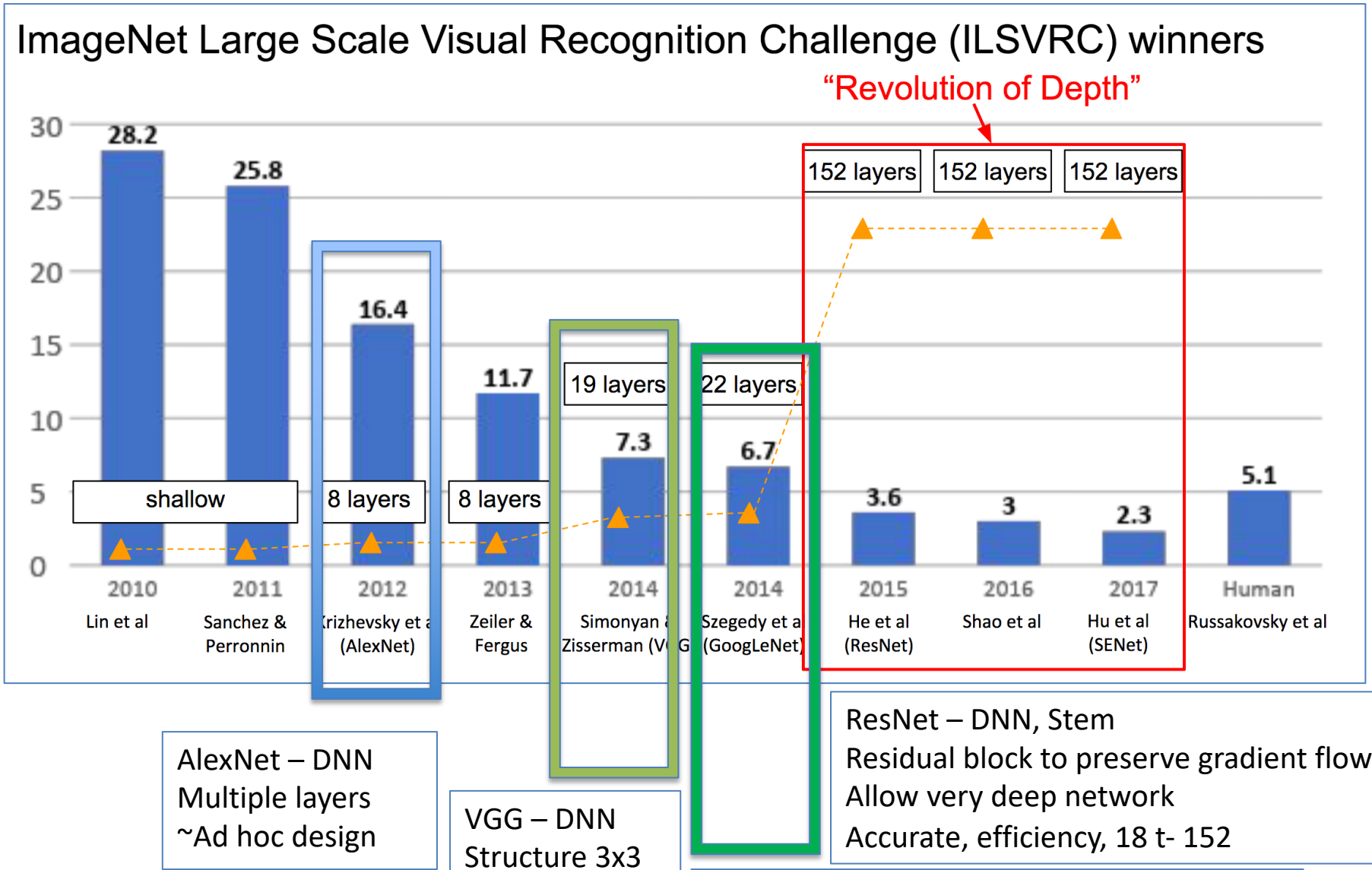


- ✓ 1000 classes
- ✓ RGB images
- ✓ Cluttered Images (one category)
- ✓ Full resolution images
- ✓ 1.2 million training images
- ✓ 100 thousand test images





http://cs231n.stanford.edu/



AlexNet – DNN
Multiple layers
~Ad hoc design

VGG – DNN
Structure 3x3
Deep NN,
Lots of parameters
Accuracy

GoogleNet – DNN
Stem to reduce feature size
Block structure for efficiency
Group pooling to reduce parameter count

ResNet – DNN, Stem
Residual block to preserve gradient flow
Allow very deep network
Accurate, efficiency, 18 t- 152

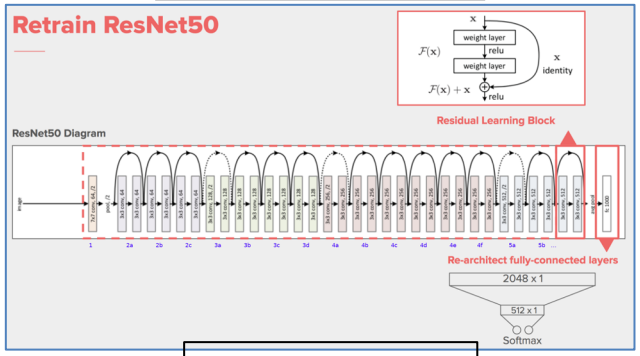
http://cs231n.stanford.edu/

Neural Network Models

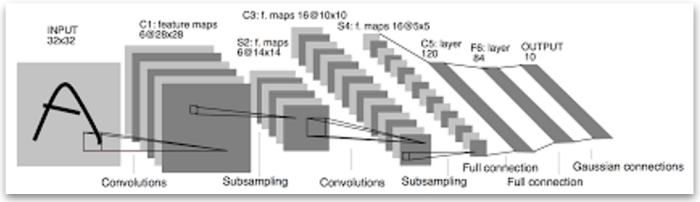
Dataset → Pre-defined Neural Network Model → Fully connected layer # neurons = # classes → Classification

Examples of Pre-defined Neural Network Models

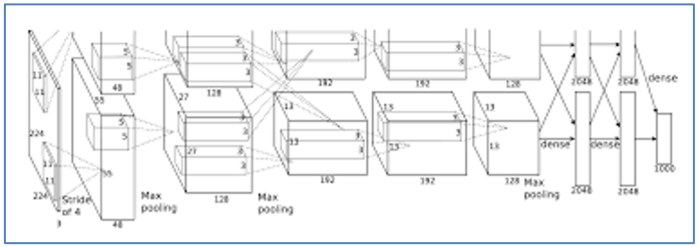
ResNet50



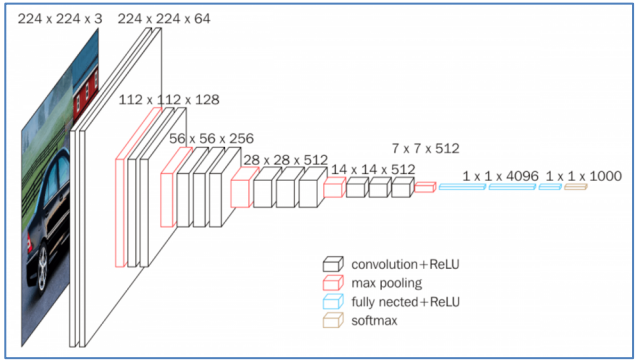
LeNet5



AlexNet

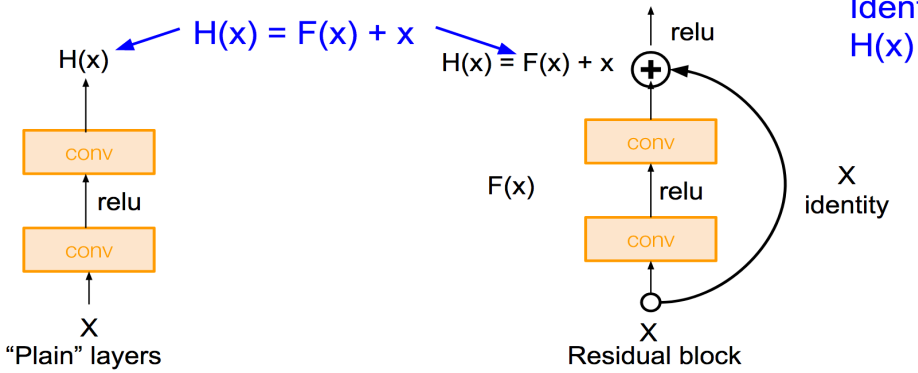


VGG16



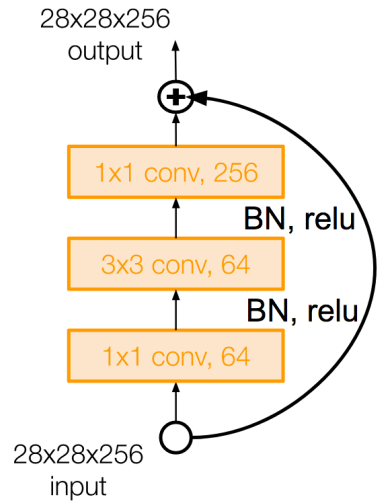
ResNet

Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



Identity mapping:
H(x) = x if F(x) = 0

Use layers to fit **residual** F(x) = H(x) - x instead of H(x) directly



For deeper networks (ResNet-50+), use "bottleneck" layer to improve efficiency (similar to GoogLeNet)

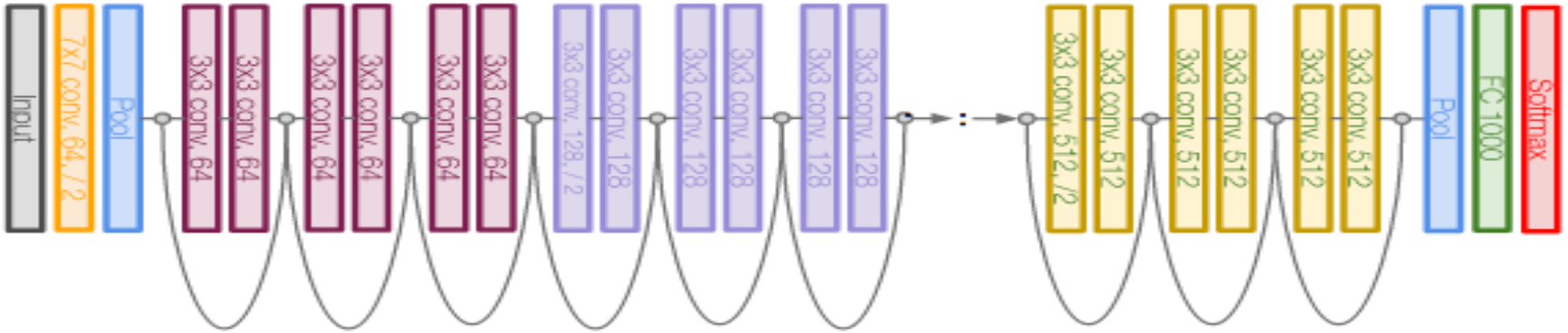
- ✓ Full ResNet architecture:-
- ✓ Stack residual blocks
- ✓ Every residual block has two 3x3 conv layers
- ✓ Periodically, double # of filters and downsample spatially using stride 2 (/2 in each dimension)
- ✓ Additional conv layer at the beginning (stem)
- ✓ No FC layers at the end (only FC 1000 to output classes)

Total depths of 18, 34, 50, 101, or 152 layers for ImageNet

1x1 conv, 256 filters projects back to 256 feature maps (28x28x256)

3x3 conv operates over only 64 feature maps

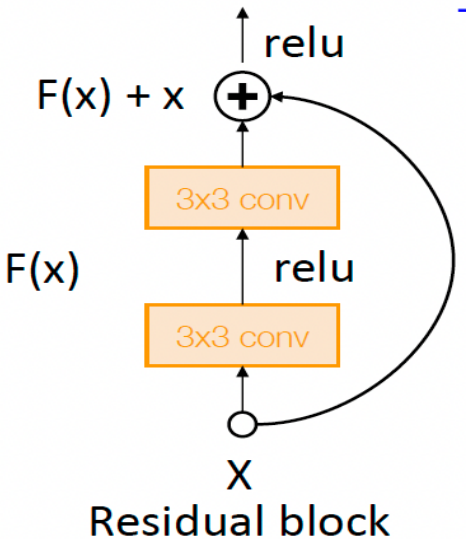
1x1 conv, 64 filters to project to 28x28x64

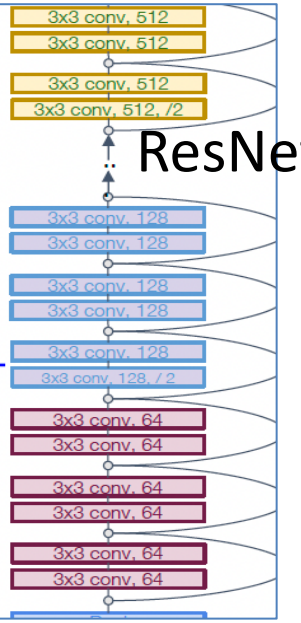


A residual network is a stack of many residual blocks

Regular design, like VGG: each residual block has two 3x3 conv

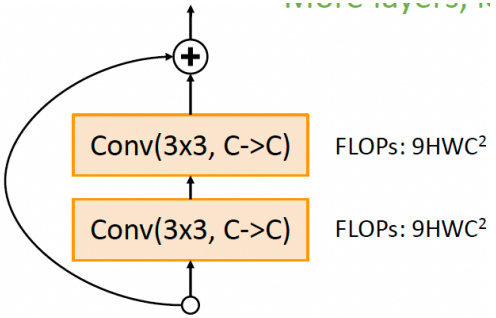
Network is divided into **stages**: the first block of each stage halves the resolution (with stride-2 conv) and doubles the number of channels





ResNet

Basic Block
ResNet18, 34



"Basic" Residual block

Total FLOPs: 18HWC²

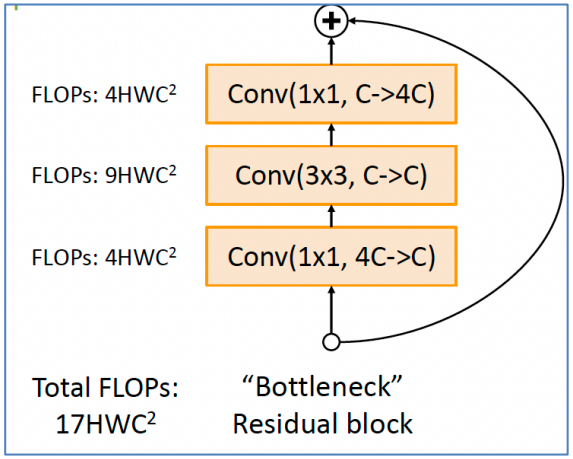
ResNet-18:
Stem: 1 conv layer
Stage 1 (C=64): 2 res. block = 4 conv
Stage 2 (C=128): 2 res. block = 4 conv
Stage 3 (C=256): 2 res. block = 4 conv
Stage 4 (C=512): 2 res. block = 4 conv
Linear

ImageNet top-5 error: 10.92
GFLOP: 1.8

ResNet-34:
Stem: 1 conv layer
Stage 1: 3 res. block = 6 conv
Stage 2: 4 res. block = 8 conv
Stage 3: 6 res. block = 12 conv
Stage 4: 3 res. block = 6 conv
Linear

ImageNet top-5 error: 8.58
GFLOP: 3.6

Bottlenect Block
ResNet50 …



Total FLOPs: 17HWC²

"Bottleneck" Residual block

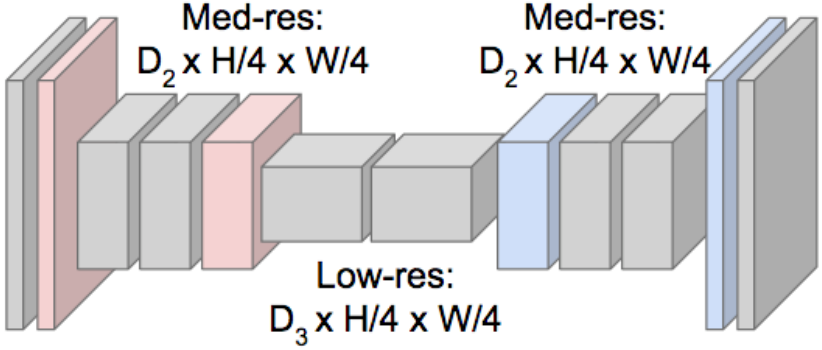
Semantic Segmentation Idea: Fully Convolutional

Downsampling: Pooling, strided convolution

Design network as a bunch of convolutional layers, with **downsampling** and **upsampling** inside the network!

Upsampling: Unpooling or strided transpose convolution

Input:
3 x H x W



High-res:
D1 x H/2 x W/2

High-res:
D1 x H/2 x W/2

Predictions:
H x W

Max Pooling
Remember which element was max!

1	2	6	3
3	5	2	1
1	2	2	1
7	3	4	8

Input: 4 x 4

5	6
7	8

Output: 2 x 2

Rest of the network

Max Unpooling
Use positions from pooling layer

1	2
3	4

Input: 2 x 2

0	0	2	0
0	1	0	0
0	0	0	0
3	0	0	4

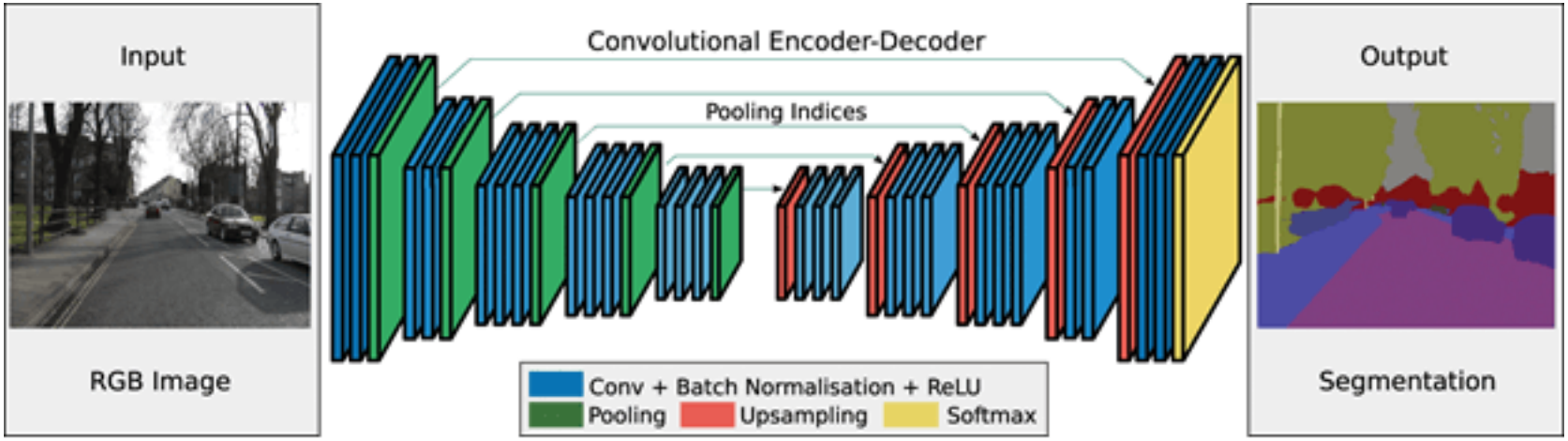
Output: 4 x 4

Image segmentation

Divide an image into multiple segments, every pixel in an image is associated with an object, instance segmentation and semantic segmentation :

In semantic segmentation, all objects of the same type are marked using one class label while in instance segmentation similar objects get their own separate labels.

The basic architecture in image segmentation consists of an encoder and a decoder.

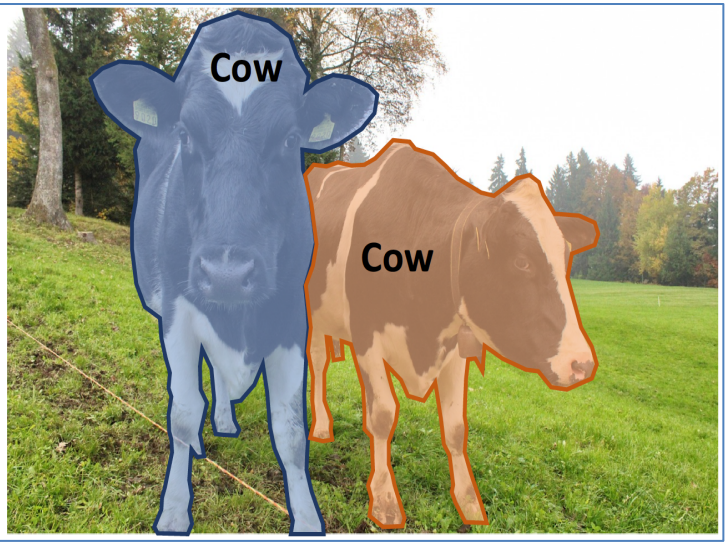


The encoder extracts features from the image through filters. The decoder is responsible for generating the final output which is usually a segmentation mask containing the outline of the object. Most of the architectures have this architecture or a variant of it.

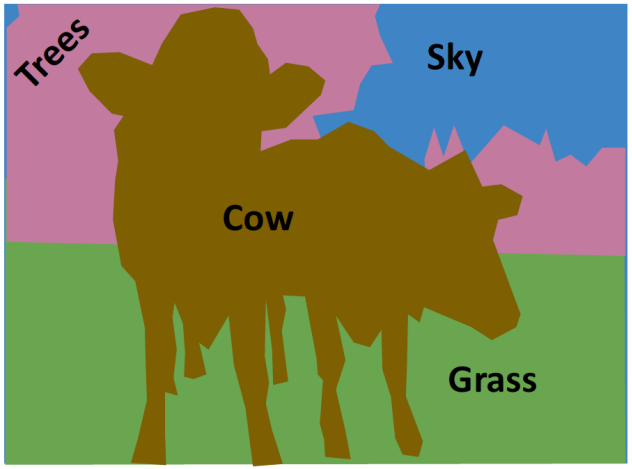
https://neptune.ai/blog/image-segmentation-in-2020

Instance Segmentation: Detect all objects in the image, and identify the pixels that belong to each object (Only things!)

Approach: Perform object detection, then predict a segmentation mask for each object!



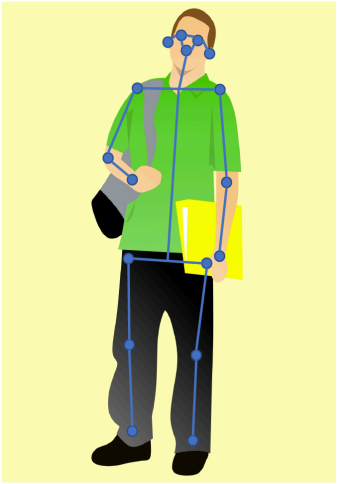
Semantic Segmentation: Identify both things and stuff, but doesn't separate instances



Beyond Instance Segmentation: Human Keypoints

Represent the pose of a human by locating a set of **keypoints**

e.g. 17 keypoints:
- Nose
- Left / Right eye
- Left / Right ear
- Left / Right shoulder
- Left / Right elbow
- Left / Right wrist
- Left / Right hip
- Left / Right knee
- Left / Right ankle

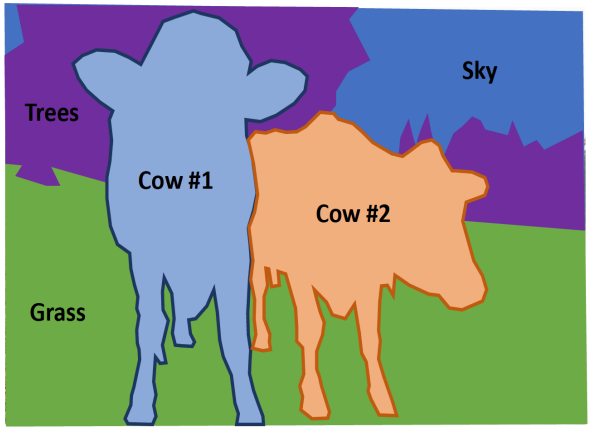


CC0 public domain

Beyond Instance Segmentation: **Panoptic Segmentation**

Label all pixels in the image (both things and stuff)

For "thing" categories also separate into instances



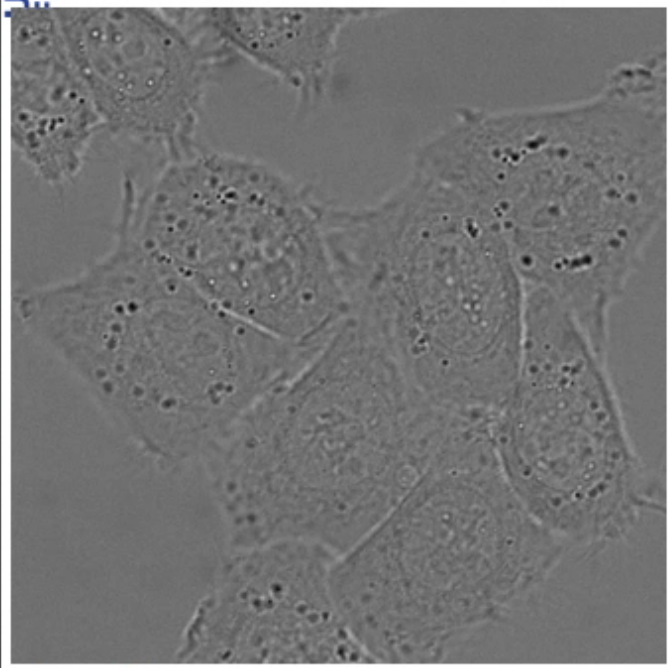
Kirillov et al, "Panoptic Segmentation", CVPR 2019
Kirillov et al, "Panoptic Feature Pyramid Networks", CVPR 2019

U-net: Convolutional Networks for Biomedical Image Segmentation

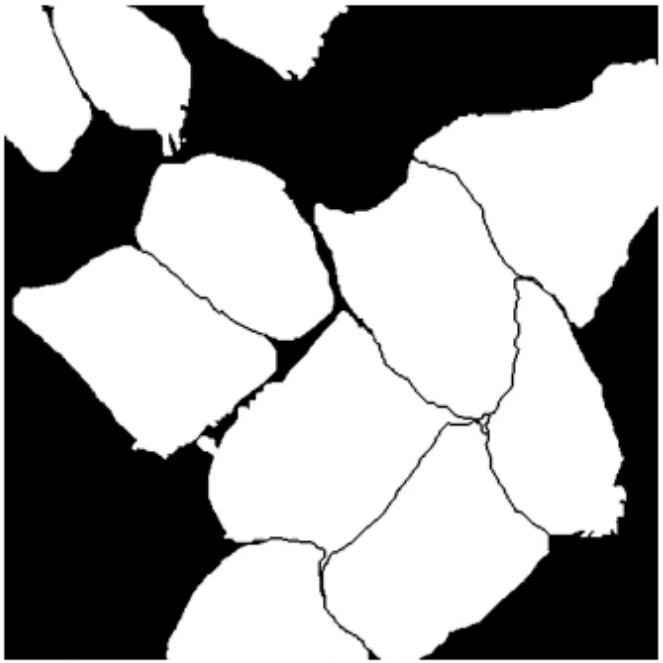
Olaf Ronneberger, Philipp Fischer, and Thomas Brox

Computer Science Department and
BIOSS Centre for Biological Signalling Studies
University of Freiburg, Germany

Biomedical Image Segmentation with U-net



input image

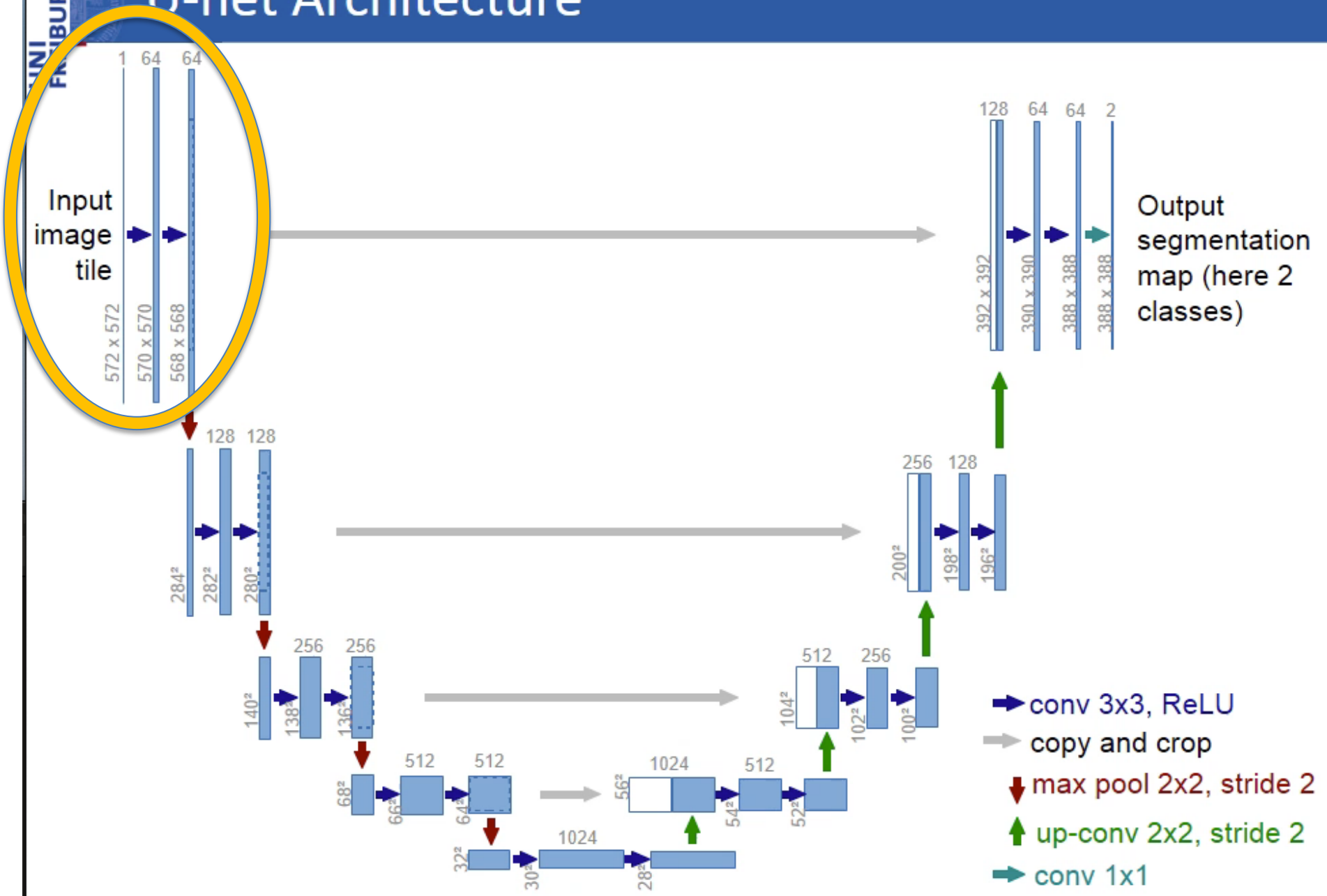


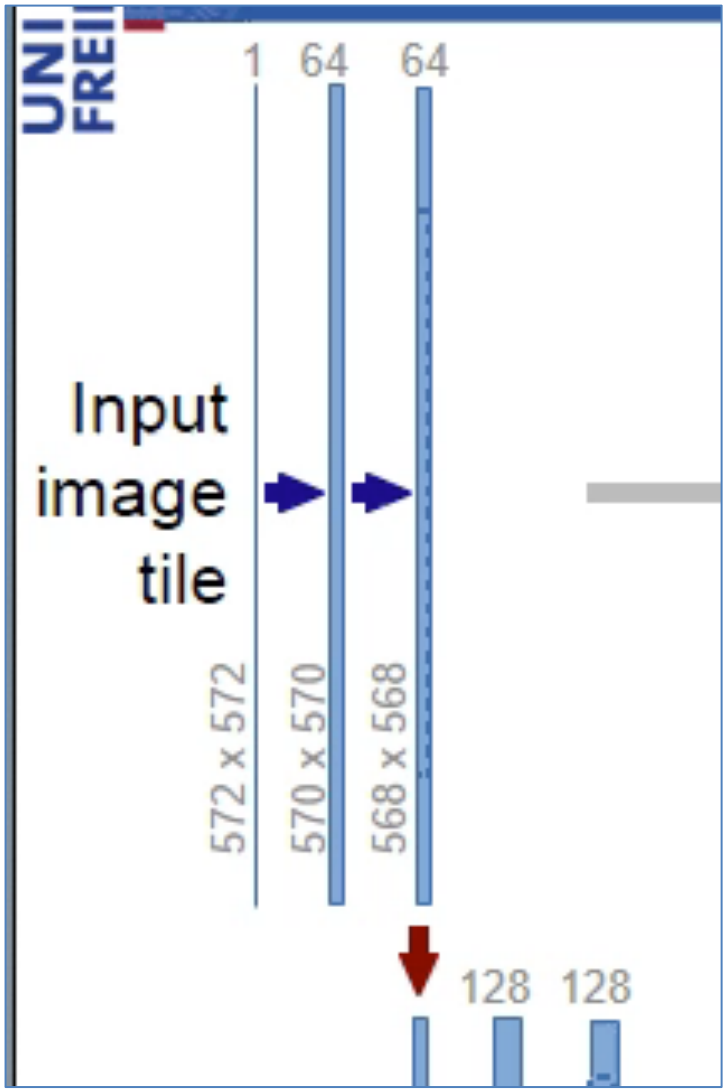
output segmentation map

- U-net **learns segmentation** in an end-to-end setting
- **Very few annotated images** (approx. 30 per application)
- **Touching objects** of the same class

[Data provided by Dr. Gert van Cappellen, Erasmus Medical Center. Rotterdam. The Netherlands]

U-net Architecture

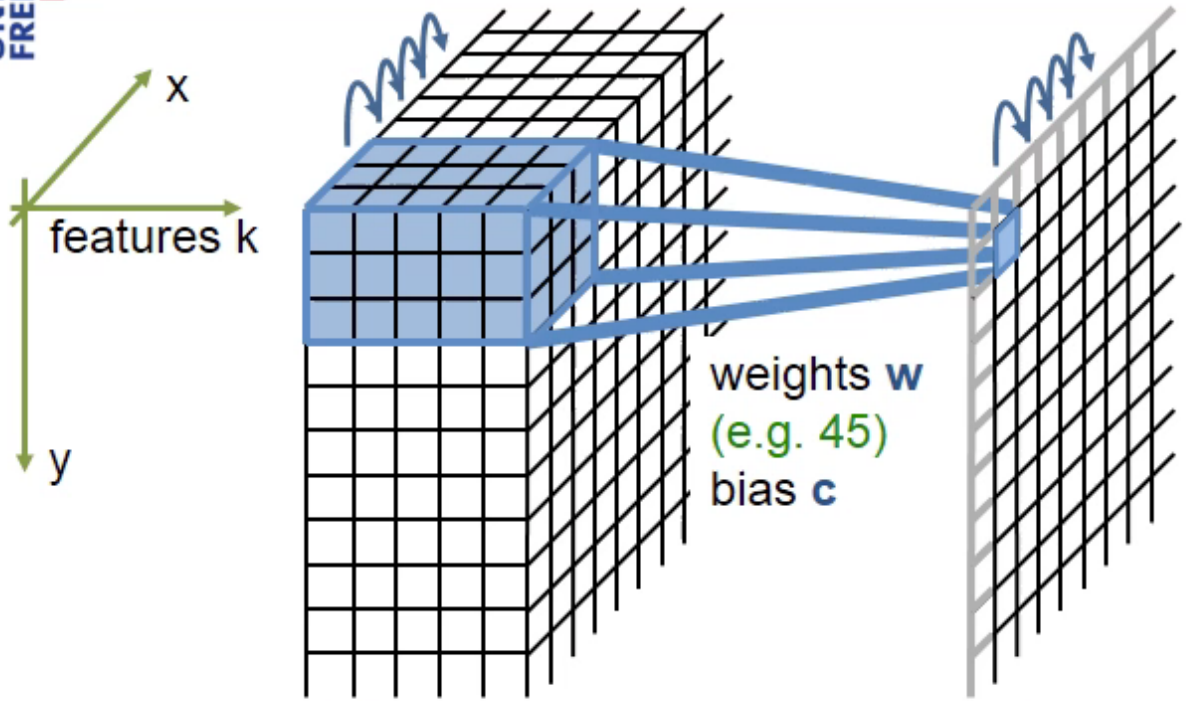




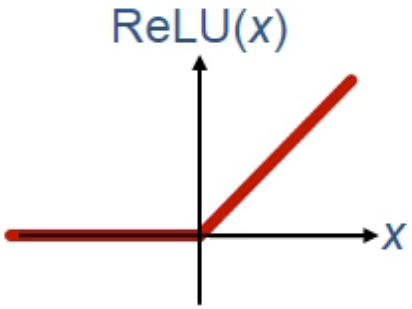
conv 3x3, ReLU
copy and crop
max pool 2x2, stride 2
up-conv 2x2, stride 2
conv 1x1

https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

3x3 convolution + ReLU

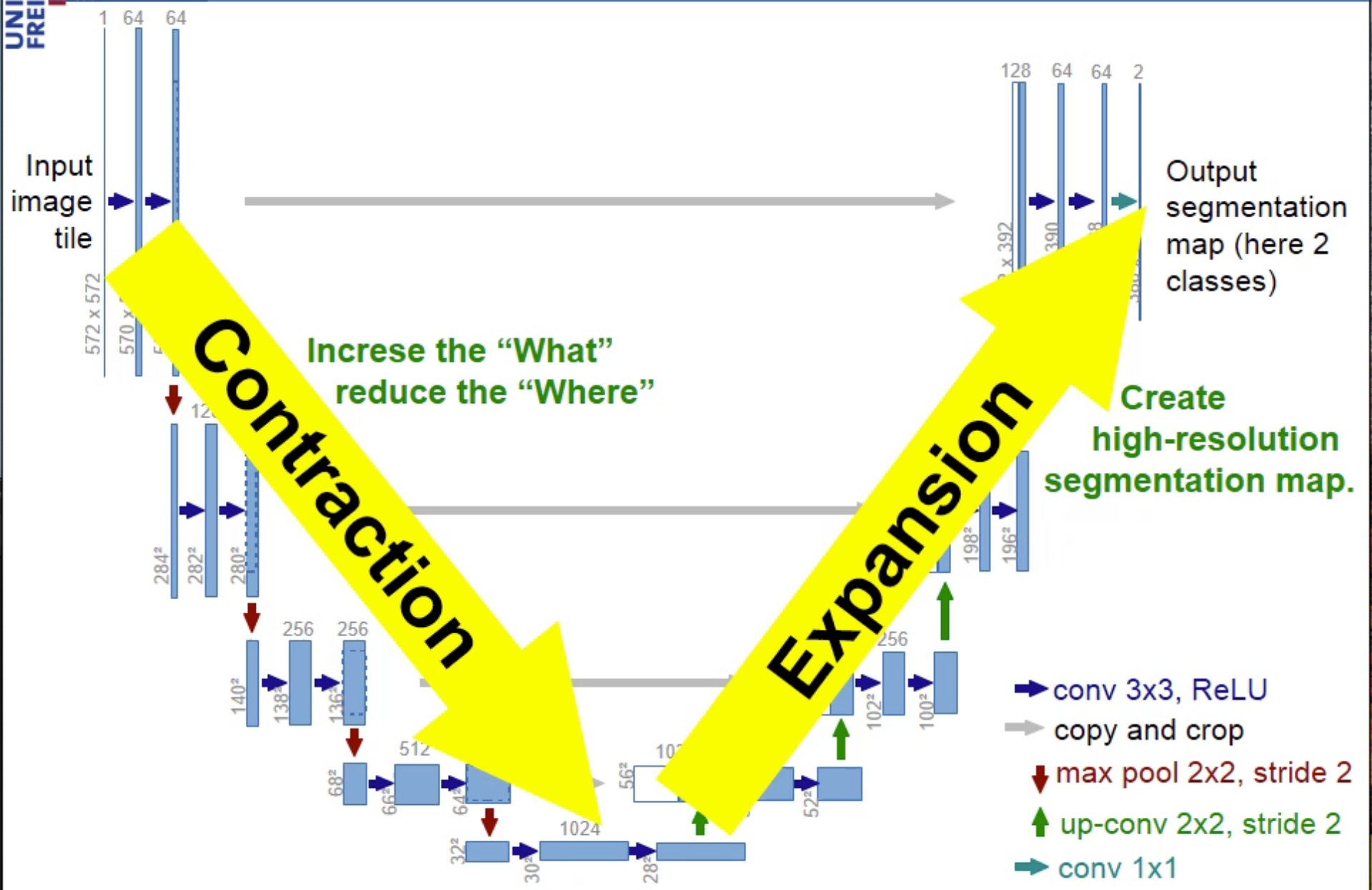


- Only valid part of convolution is used.
- For 3x3 convolutions a 1-pixel border is lost



$$b_{x,y,l} = \mathrm{ReLU}\Big(\sum_{\substack{i\in\{-1,0,1\} \\ j\in\{-1,0,1\} \\ k\in\{1,\ldots,K\}}} w_{i,j,k,l} \cdot a_{x+i,y+j,k} + c_l \Big)$$

U-Net Architecture



U-net Architecture

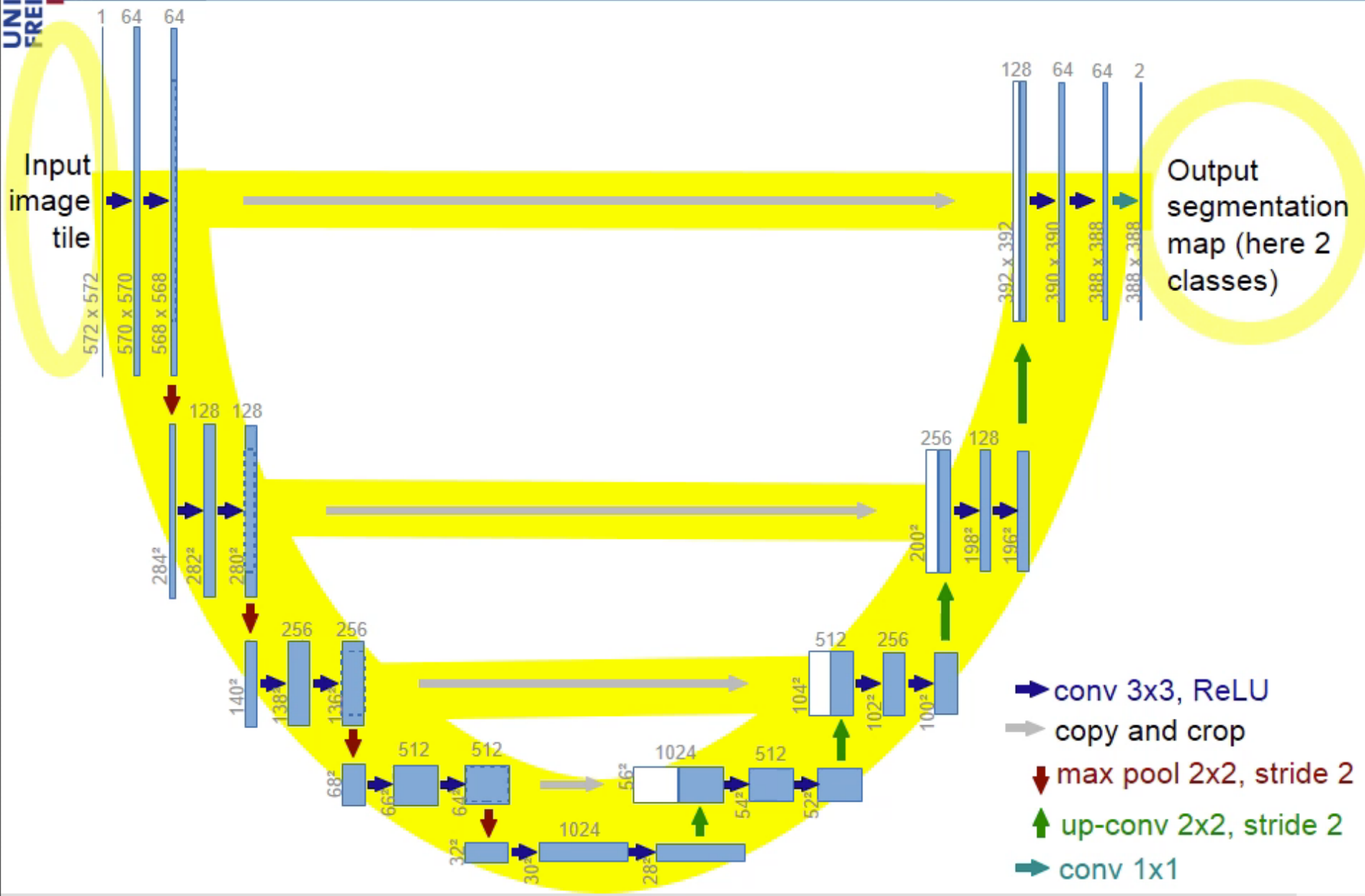
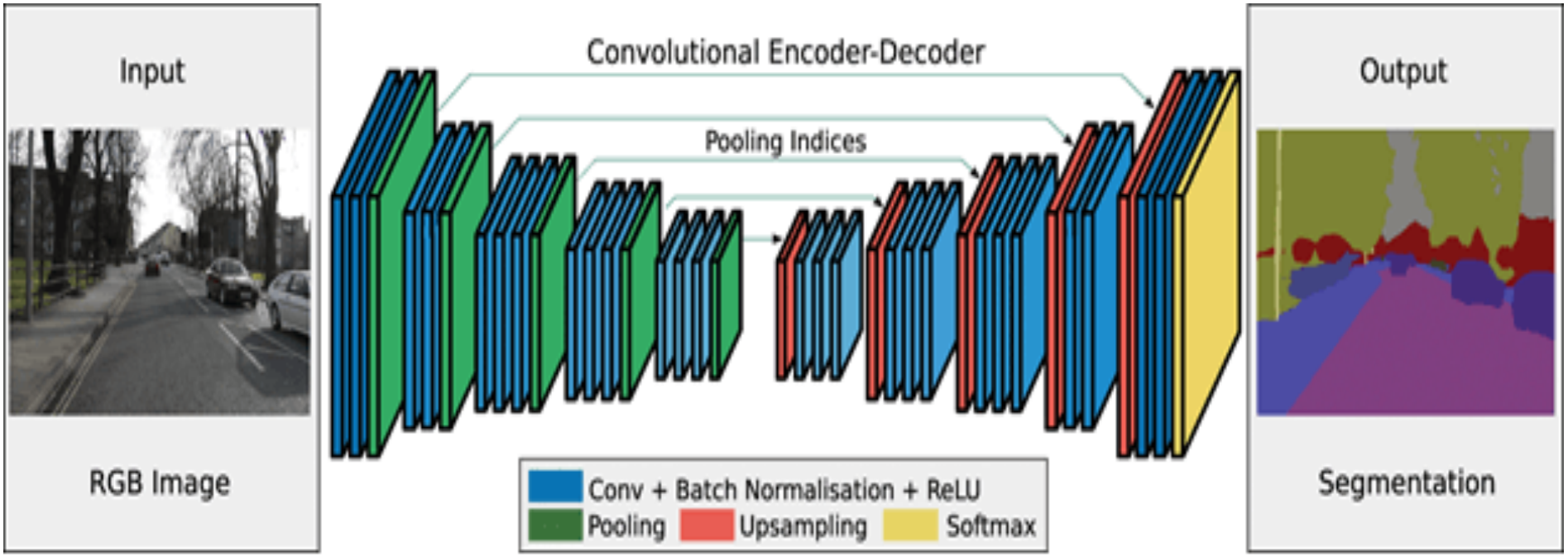


Image segmentation

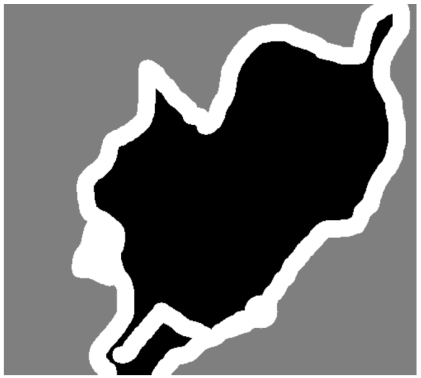
The basic architecture in image segmentation consists of an encoder and a decoder.



The encoder extracts features from the image through filters. The decoder is responsible for generating the final output which is usually a segmentation mask containing the outline of the object. Most of the architectures have this architecture or a variant of it.

https://neptune.ai/blog/image-segmentation-in-2020

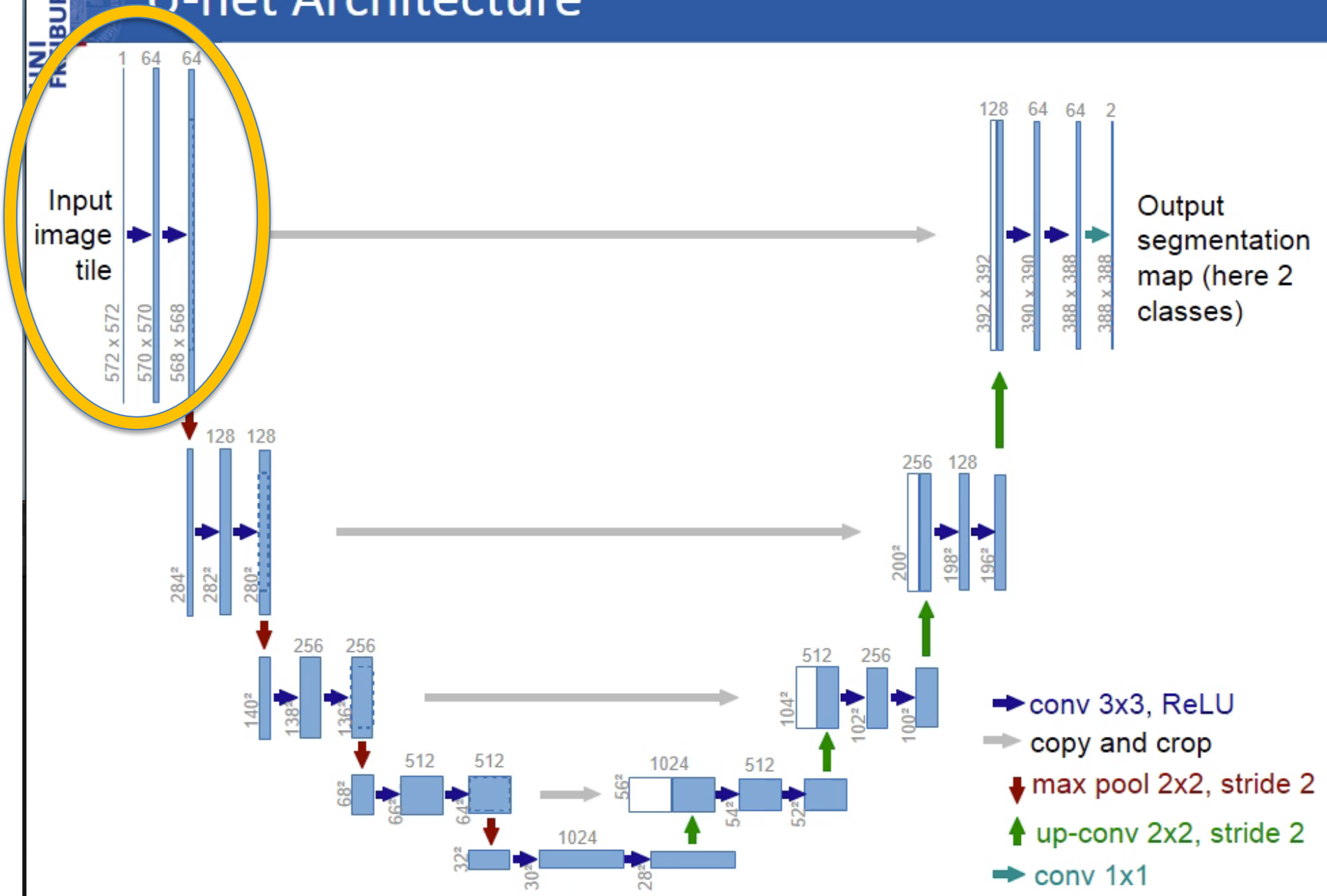
CODE explanation
https://keras.io/examples/vision/oxford_pets_image_segmentation/



```
input_dir = "images/"
target_dir = "annotat
img_size = (160, 160)
num_classes = 3
batch_size = 32
```

```
def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))

    ### [First half of the network: downsampling inputs] ###

    # Entry block
    x = layers.Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    # Blocks 1, 2, 3 are identical apart from the feature depth.
    for filters in [64, 128, 256]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(filters, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual
```

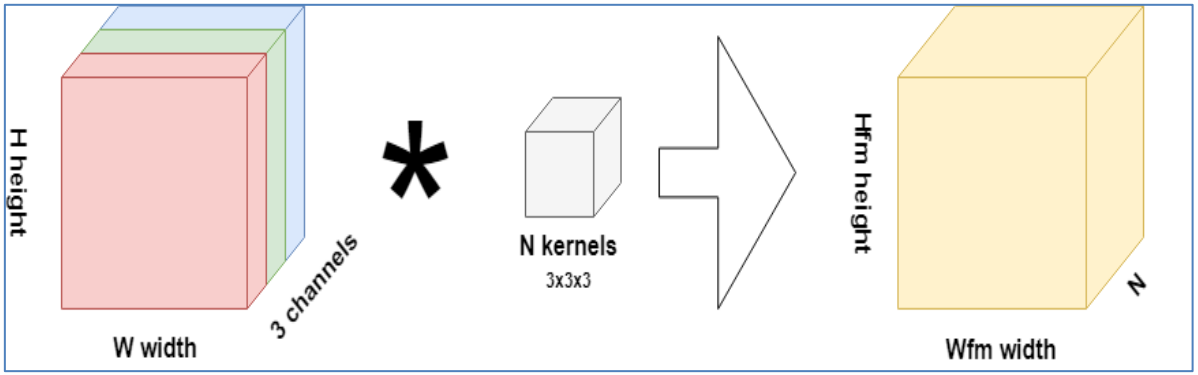
U-net Architecture



Img_size = (160, 160,3)
Num_class=3
Batch_size = 32

```
X = layers.Conv2D(32, 3, strides=2, padding='same') (input)
X = layers. BatchNormalization()(x)
X  = layers.Activation("relu")(x)
```

Height = 160, width = 160, depth (channel) = 3

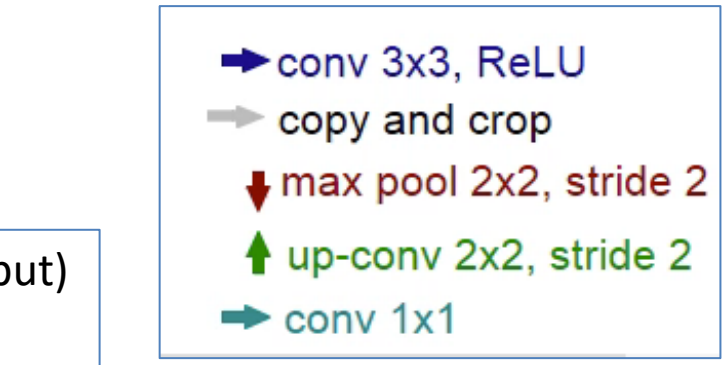


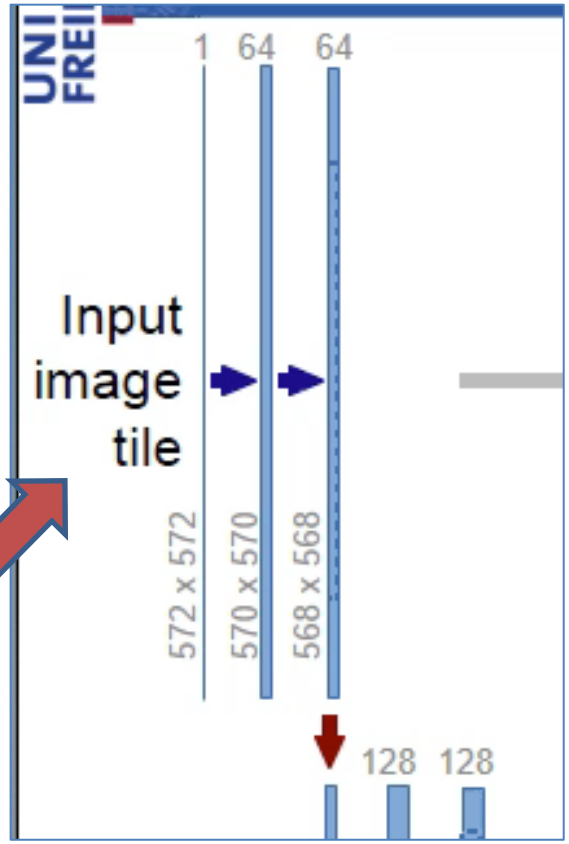
Input matrix = 160 x 160 x 3

N = 32

Stride = 2, same padding

output matrix = 80 x 80 x 32





https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 160, 160, 3)	0	
conv2d (Conv2D)	(None, 80, 80, 32)	896	input_1[0][0]
batch_normalization (BatchNorma	(None, 80, 80, 32)	128	conv2d[0][0]
activation (Activation)	(None, 80, 80, 32)	0	batch_normalization[0
activation_1 (Activation)	(None, 80, 80, 32)	0	activation[0][0]
separable_conv2d (SeparableConv	(None, 80, 80, 64)	2400	activation_1[0][0]
batch_normalization_1 (BatchNor	(None, 80, 80, 64)	256	separable_conv2d[0][0
activation_2 (Activation)	(None, 80, 80, 64)	0	batch_normalization_1
separable_conv2d_1 (SeparableCo	(None, 80, 80, 64)	4736	activation_2[0][0]
batch_normalization_2 (BatchNor	(None, 80, 80, 64)	256	separable_conv2d_1[0]
max_pooling2d (MaxPooling2D)	(None, 40, 40, 64)	0	batch_normalization_2
conv2d_1 (Conv2D)	(None, 40, 40, 64)	2112	activation[0][0]
add (Add)	(None, 40, 40, 64)	0	max_pooling2d[0][0] conv2d_1[0][0]

(3x3x3+1) x32 = 896

(4x32)=128

```
input_dir = "images/"
target_dir = "annotat
img_size = (160, 160)
num_classes = 3
batch_size = 32
```

```
def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))

    ### [First half of the network: downsampling inputs] ###

    # Entry block
    x = layers.Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    # Blocks 1, 2, 3 are identical apart from the feature depth.
    for filters in [64, 128, 256]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(filters, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual
```

No of Filters = 64

X=Input matrix = 80 x 80 x 32

N = 64

Filter size = 3 x3, stride = 1, same padding

```
X  = layers.Activation("relu")(x)
X = layers.SeparableConv2D(64, 3,  padding='same')(x)
X = layers. BatchNormalization()(x)
```

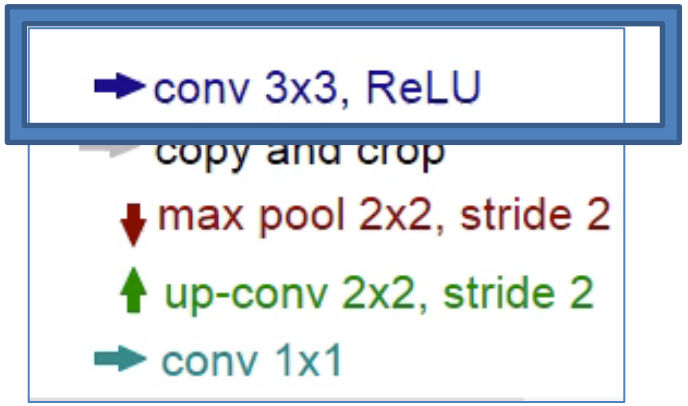
output matrix = 80 x 80 x 64

1st step : M = 32, (3x3x1) x 32 = 288

2nd step : weights = (1 x 1 x 32) x 64 = 2048

Bias = 64

Total parameters = 2400

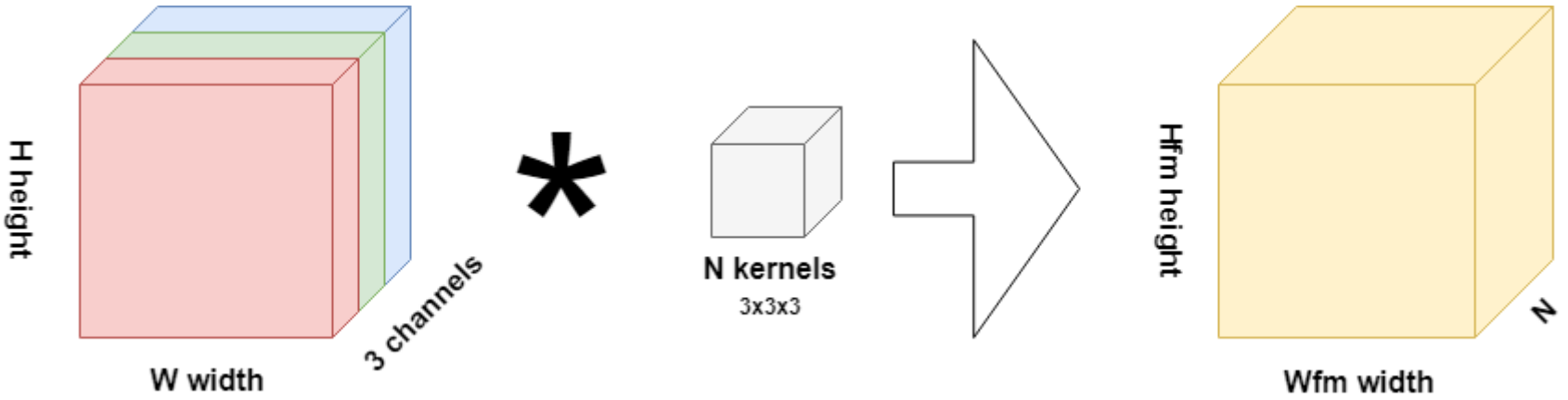


https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

Keras function : SeparableConv2D (….)

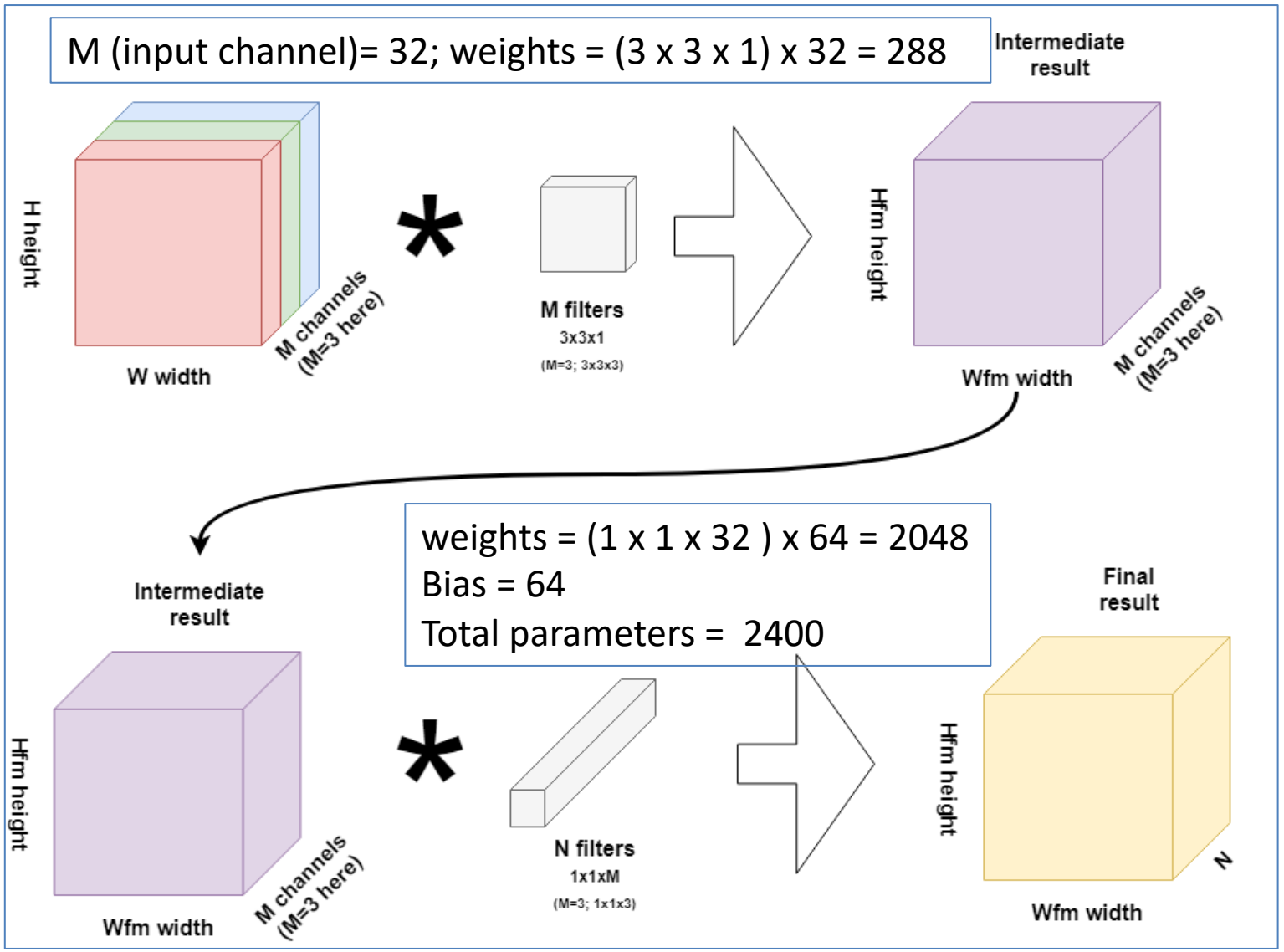
A little bit different from standard CONV2D but gets the same results

Separate it to two stages



If your 15×15 pixels image is RGB, and by consequence has 3 channels, you'll need (15-3+1) x (15-3+1) x 3 x 3 x 3 x N = 4563N multiplications to complete the full interpretation of *one image*. If you're working with ten kernels, so N = 10, you'll need over 45000 multiplications. Today, 20 to 50 kernels are not uncommon, datasets often span thousands of images and neural networks often compose multiple convolutional layers in their architecture.

https://www.machinecurve.com/index.php/2019/09/24/creating-depthwise-separable-convolutions-in-keras/



With those, you essentially split your N traditional kernels into *depthwise convolutions* and *pointwise convolutions*. In the first subprocess, you convolve with M filters on a layer basis, adding the kernels 'pointwise' in the second subprocess. While achieving the same result, you'll need only **9633 convolutions**

No of Filters = 64

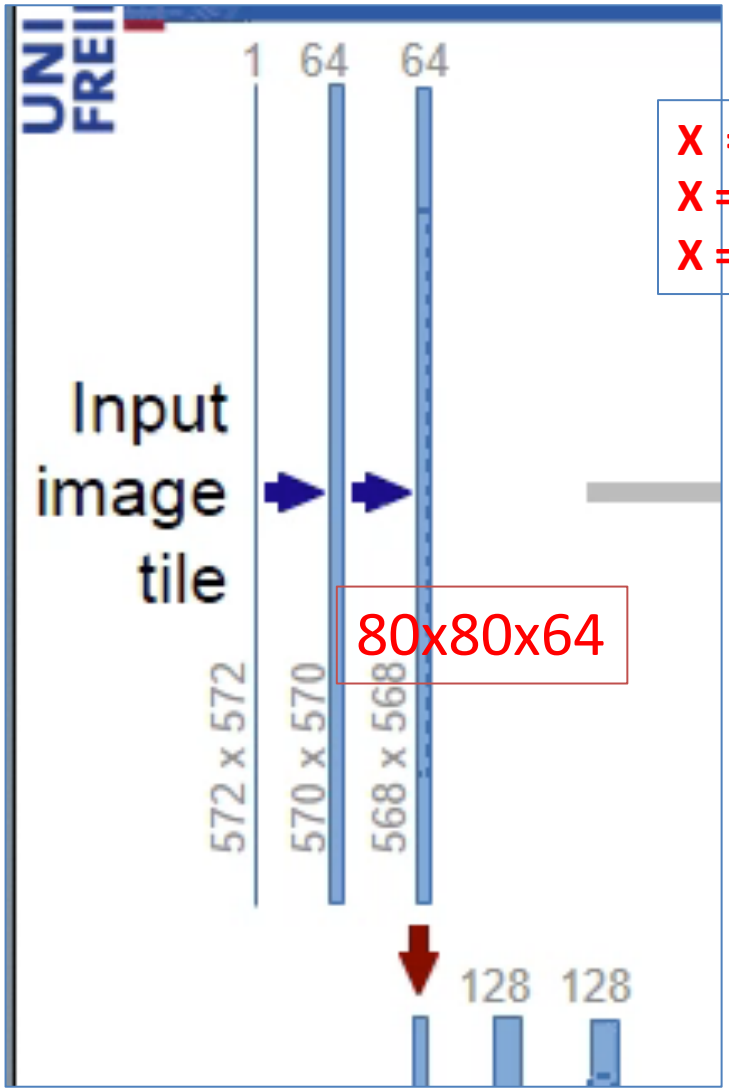
X= Input matrix = 80 x 80 x 64 | N = 64

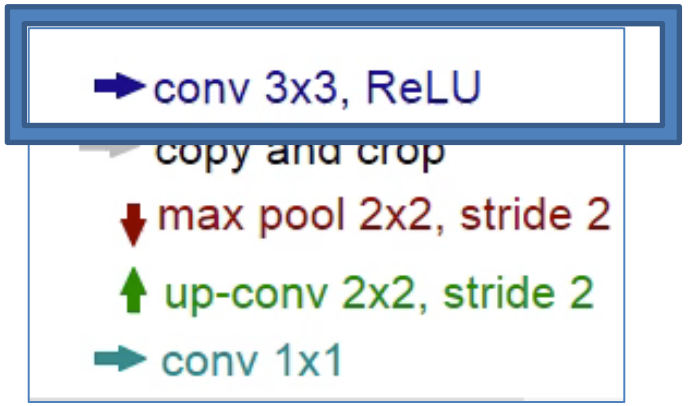
Filter size = 3 x3, stride = 1, same padding

X = layers.Activation("relu")(x)
X = layers.SeparableConv2D(64, 3, padding='same')(x)
X = layers. BatchNormalization()(x)

output matrix = 80 x 80 x 64

1st step : M = 64, (3x3x1) x 64 = 576
2nd step : weights = (1 x 1 x 64) x 64 = 4096
Bias = 64
Total parameters = 4736





https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

X = layers. BatchNormalization()(x)

Batch Normalization has 4 parameters per set of filters (feature)
Number of parameters = 4 x number of filters (bias) = 128, 256

As you can read there, in order to make the batch normalization work during training, they need to keep track of the distributions of each normalized dimensions. To do so, since you are in mode=0by default, they compute 4 parameters per feature on the previous layer. Those parameters are making sure that you properly propagate and backpropagate the information. These parameters are in fact [gamma weights, beta weights, moving_mean(non-trainable), moving_variance(non-trainable)], of the size of the input layer.

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

input matrix = 80 x 80 x 64

X = layers.MaxPooling2D(3, stride= 2, padding='same')(x)
Reduce size by half because of stride, paddling

output matrix = 40 x 40 x 64

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 160, 160, 3)	0	
conv2d (Conv2D)	(None, 80, 80, 32)	896	input_1[0][0]
batch_normalization (BatchNorma	(None, 80, 80, 32)	128	conv2d[0][0]
activation (Activation)	(None, 80, 80, 32)	0	batch_normalization[0
activation_1 (Activation)	(None, 80, 80, 32)	0	activation[0][0]
separable_conv2d (SeparableConv	(None, 80, 80, 64)	2400	activation_1[0][0]
batch_normalization_1 (BatchNor	(None, 80, 80, 64)	256	separable_conv2d[0][0
activation_2 (Activation)	(None, 80, 80, 64)	0	batch_normalization_1
separable_conv2d_1 (SeparableCo	(None, 80, 80, 64)	4736	activation_2[0][0]
batch_normalization_2 (BatchNor	(None, 80, 80, 64)	256	separable_conv2d_1[0]
max_pooling2d (MaxPooling2D)	(None, 40, 40, 64)	0	batch_normalization_2
conv2d_1 (Conv2D)	(None, 40, 40, 64)	2112	activation[0][0]
add (Add)	(None, 40, 40, 64)	0	max_pooling2d[0][0] conv2d_1[0][0]

(3x3x3+1) x32 = 896

(4x32)=128

(3x3x64) +64x64 + 64 = 4736

(4x64)=256

```python
def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))

    ### [First half of the network: downsampling inputs] ###

    # Entry block
    x = layers.Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    # Blocks 1, 2, 3 are identical apart from the feature depth.
    for filters in [64, 128, 256]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(filters, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual
```

```python
input_dir = "images/"
target_dir = "annotat
img_size = (160, 160)
num_classes = 3
batch_size = 32
```

Model: "functional_1"

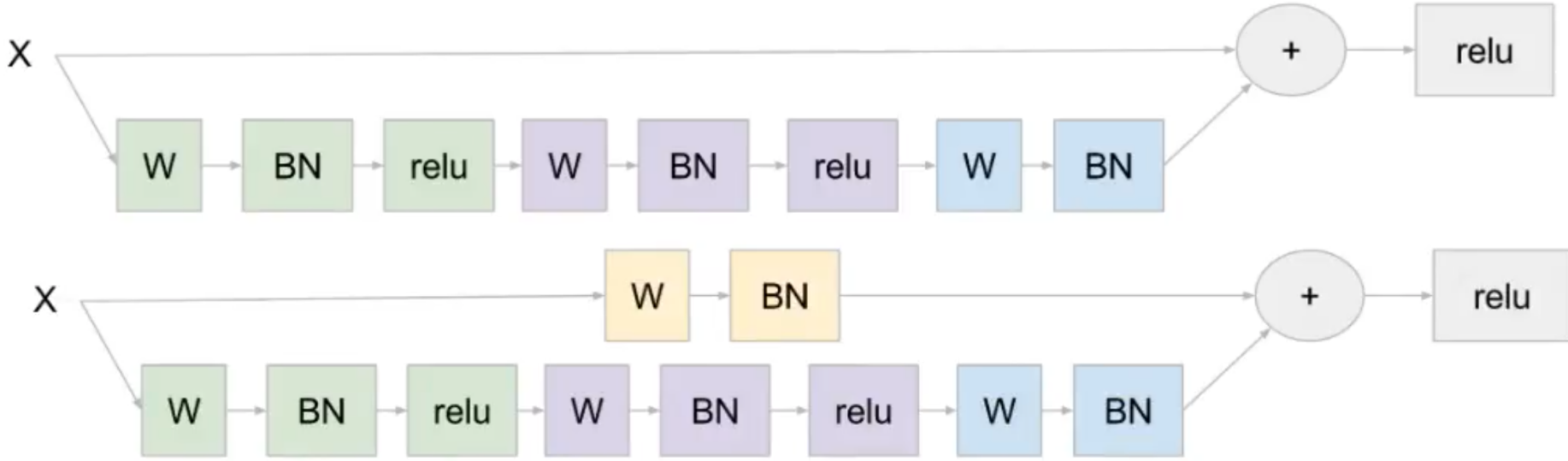
Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 160, 160, 3)	0	
conv2d (Conv2D)	(None, 80, 80, 32)	896	input_1[0][0]
batch_normalization (BatchNorma	(None, 80, 80, 32)	128	conv2d[0][0]
activation (Activation)	(None, 80, 80, 32)	0	batch_normalization[0
activation_1 (Activation)	(None, 80, 80, 32)	0	activation[0][0]
separable_conv2d (SeparableConv	(None, 80, 80, 64)	2400	activation_1[0][0]
batch_normalization_1 (BatchNor	(None, 80, 80, 64)	256	separable_conv2d[0][0
activation_2 (Activation)	(None, 80, 80, 64)	0	batch_normalization_1
separable_conv2d_1 (SeparableCo	(None, 80, 80, 64)	4736	activation_2[0][0]
batch_normalization_2 (BatchNor	(None, 80, 80, 64)	256	separable_conv2d_1[0]
max_pooling2d (MaxPooling2D)	(None, 40, 40, 64)	0	batch_normalization_2
conv2d_1 (Conv2D)	(None, 40, 40, 64)	2112	activation[0][0]
add (Add)	(None, 40, 40, 64)	0	max_pooling2d[0][0] conv2d_1[0][0]

(3x3x3+1) x32 = 896

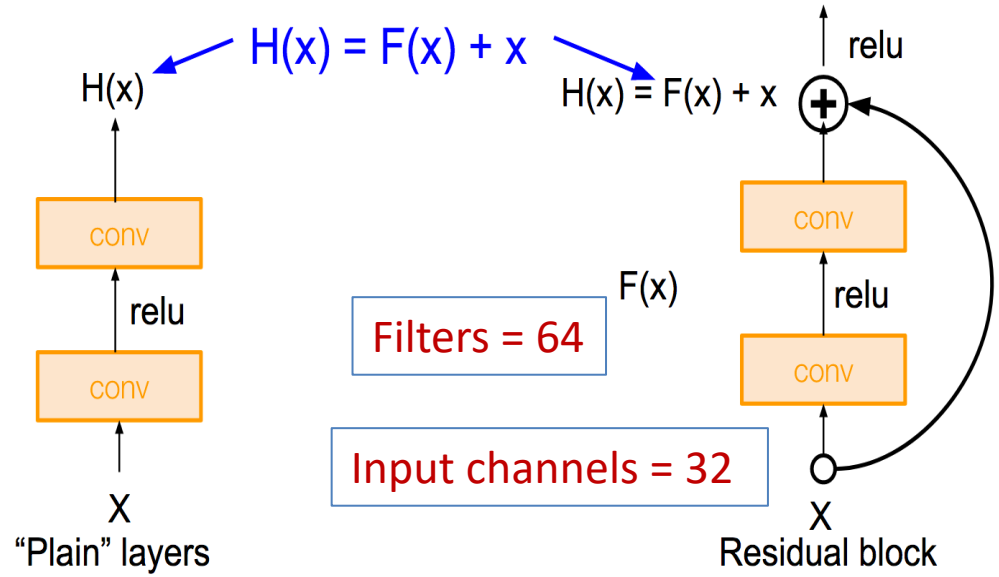
(4x32)=128

(3x3x64) +64x64 + 64 = 4736

(4x64)=256



Solution: Use network layers to fit a residual mapping instead of directly trying to fit a desired underlying mapping



Filters = 64

Input channels = 32

Identity mapping:
H(x) = x if F(x) = 0

Use layers to fit **residual** F(x) = H(x) - x instead of H(x) directly

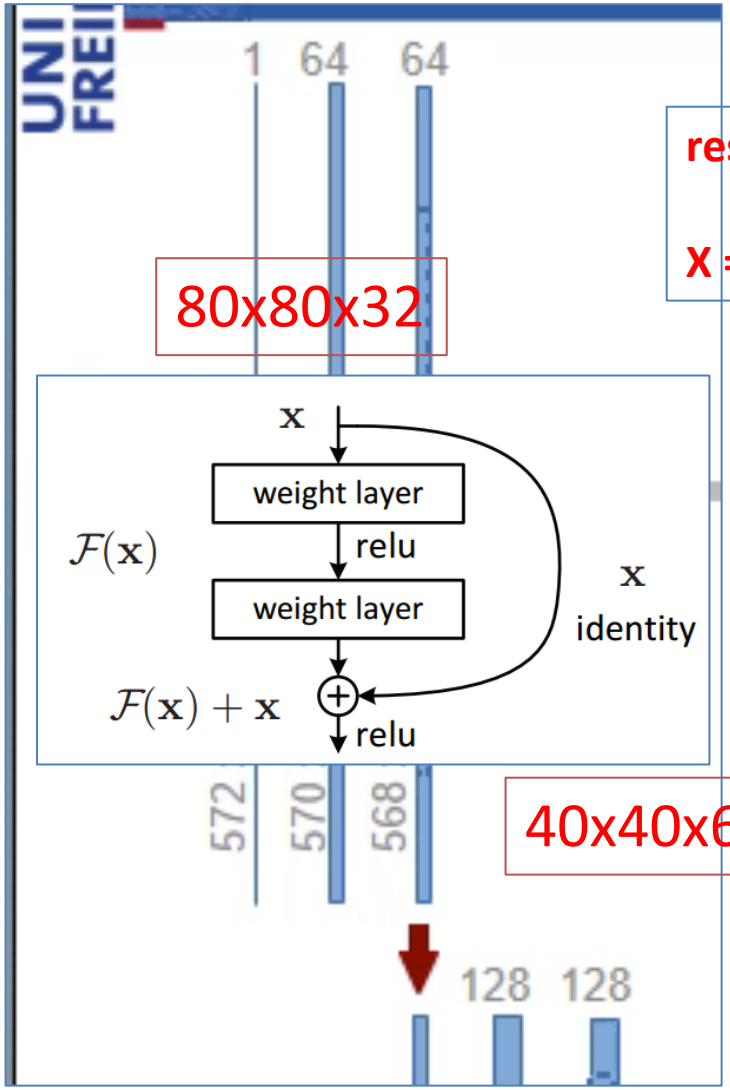
https://www.youtube.com/watch?v=0tBPSxioIZE

Weights = (1 x1 x 32) x 64 + 64 = 2112

No of Filters = 64

Input matrix = 80 x 80 x 32

N = 64



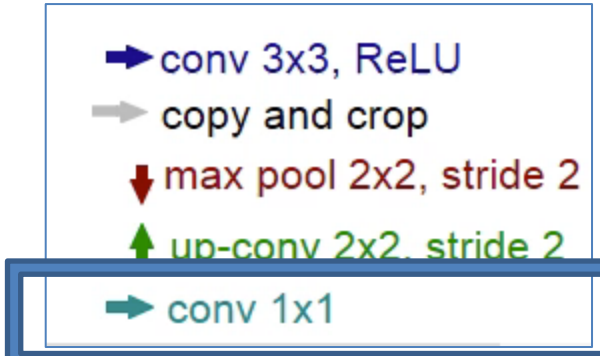
Channel = 32, Filter size = 1 x1, stride = 2, same padding

```
residual = layers.Conv2D(64, 1, stride=2,
                         padding='same')(previous_block_activation)
X = layers.add(x, residual)
```

output matrix = 40 x 40 x 64

parameter, (1x1x32) x 64 = 2048
Bias = 64
Total parameters =2112

Add to the current X
Size = 40x40x64



https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

Model: "functional_1"

Layer (type)	Output Shape	Param #	Connected to
input_1 (InputLayer)	[(None, 160, 160, 3)	0	
conv2d (Conv2D)	(None, 80, 80, 32)	896	input_1[0][0]
batch_normalization (BatchNorma	(None, 80, 80, 32)	128	conv2d[0][0]
activation (Activation)	(None, 80, 80, 32)	0	batch_normalization[0
activation_1 (Activation)	(None, 80, 80, 32)	0	activation[0][0]
separable_conv2d (SeparableConv	(None, 80, 80, 64)	2400	activation_1[0][0]
batch_normalization_1 (BatchNor	(None, 80, 80, 64)	256	separable_conv2d[0][0
activation_2 (Activation)	(None, 80, 80, 64)	0	batch_normalization_1
separable_conv2d_1 (SeparableCo	(None, 80, 80, 64)	4736	activation_2[0][0]
batch_normalization_2 (BatchNor	(None, 80, 80, 64)	256	separable_conv2d_1[0]
max_pooling2d (MaxPooling2D)	(None, 40, 40, 64)	0	batch_normalization_2
conv2d_1 (Conv2D)	(None, 40, 40, 64)	2112	activation[0][0]
add (Add)	(None, 40, 40, 64)	0	max_pooling2d[0][0] conv2d_1[0][0]

(3x3x3+1) x32 = 896

(4x32)=128

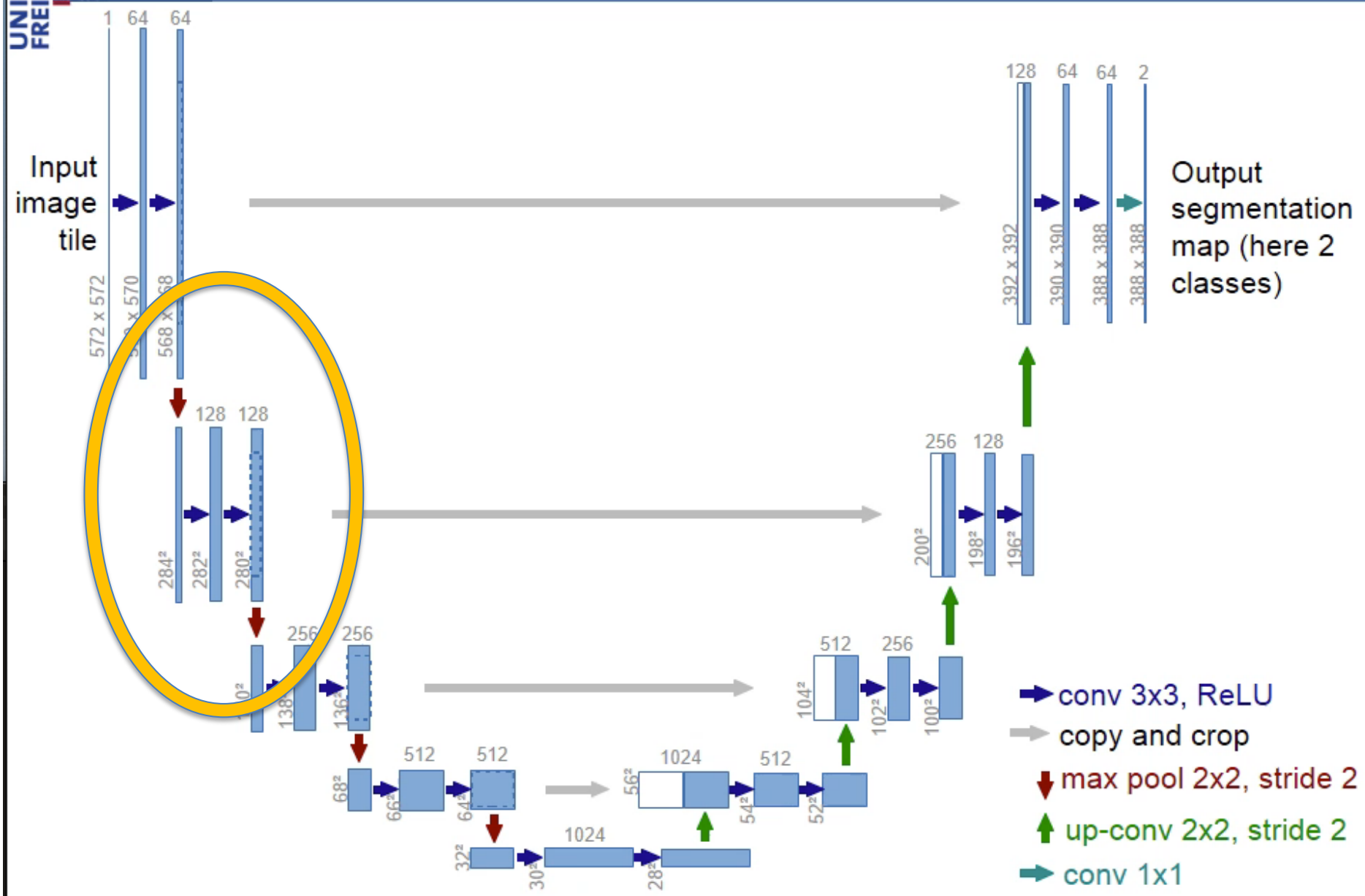
(3x3x64) +64x64 + 64 = 4736

(4x64)=256

```
input_dir = "images/"
target_dir = "annotat
img_size = (160, 160)
num_classes = 3
batch_size = 32
```

```
def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))

    ### [First half of the network: downsampling inputs] ###

    # Entry block
    x = layers.Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    # Blocks 1, 2, 3 are identical apart from the feature depth.
    for filters in [64, 128, 256]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(filters, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual
```

U-net Architecture



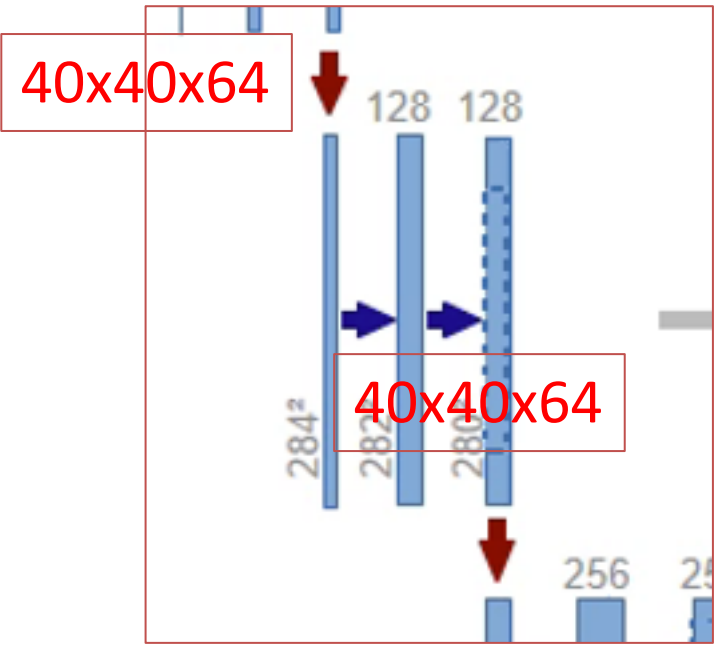
No of Filters = 128

Input matrix = 40 x 40 x 64

N = 128

Filter size = 3 x3, stride = 1, same padding

```
X  = layers.Activation("relu")(x)
X = layers.SeparableConv2D(128, 3,  padding='same')(x)
X = layers. BatchNormalization()(x)
```



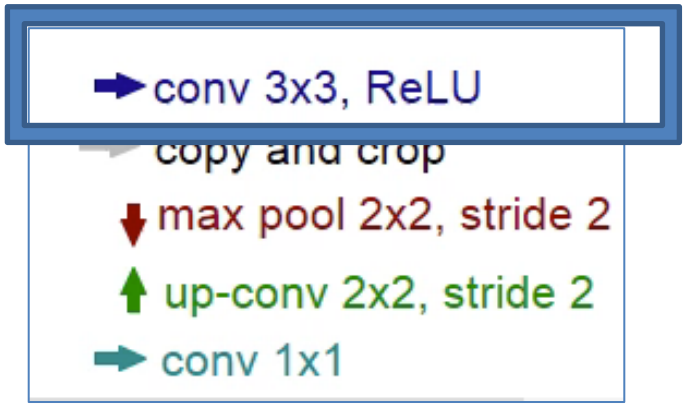
output matrix = 40 x 40 x 128

1st step : M = 64, (3x3x1) x 64 = 576

2nd step : weights = (1 x 1 x 64) x 128 = 8192

Bias = 128

Total parameters = 8896



https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

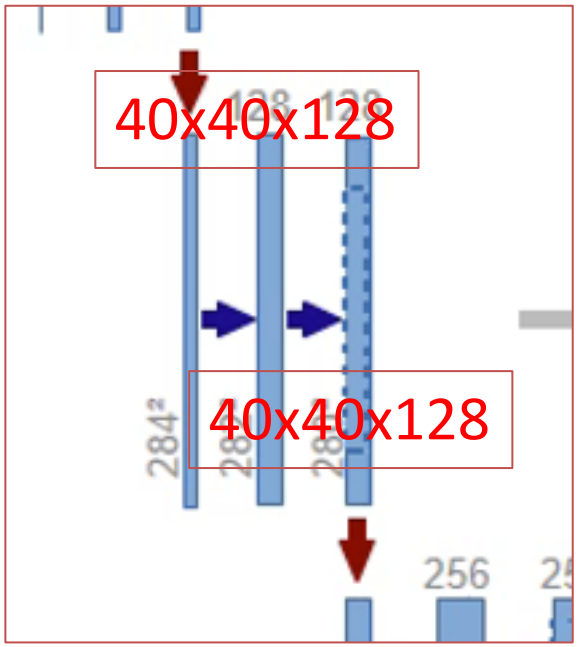
No of Filters = 128

Input matrix = 40 x 40 x 128

N = 128

Filter size = 3 x3, stride = 1, same padding

```
X  = layers.Activation("relu")(x)
X = layers.SeparableConv2D(128, 3,  padding='same')(x)
X = layers. BatchNormalization()(x)
```



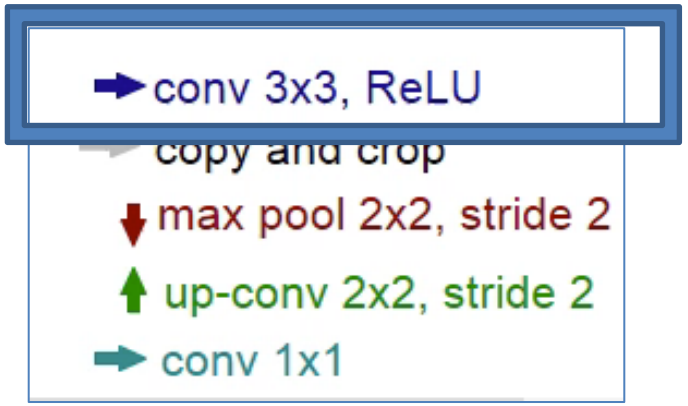
output matrix = 40 x 40 x 128

1st step : M = 128, (3x3x1) x 128 = 1152

2nd step : weights = (1 x 1 x 128) x 128 = 16384

Bias = 128

Total parameters = 17664



https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

X = layers. BatchNormalization()(x)

Batch Normalization has 4 parameters per set of filters (feature)
Number of parameters = 4 x number of filters (bias) = 256

As you can read there, in order to make the batch normalization work during training, they need to keep track of the distributions of each normalized dimensions. To do so, since you are in mode=0by default, they compute 4 parameters per feature on the previous layer. Those parameters are making sure that you properly propagate and backpropagate the information. These parameters are in fact [gamma weights, beta weights, moving_mean(non-trainable), moving_variance(non-trainable)], of the size of the input layer.

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

input matrix = 40 x 40 x 128

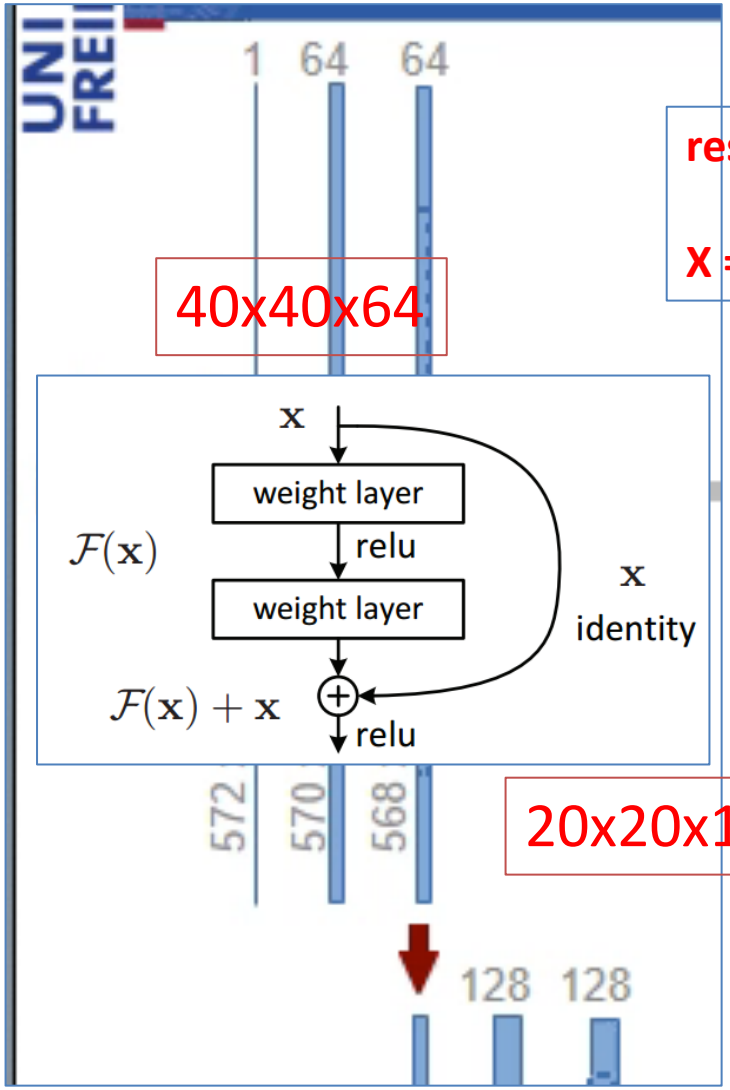
X = layers.MaxPooling2D(3, stride= 2, padding='same')(x)
Reduce size by half because of stride, paddling

output matrix = 20 x 20 x 128

No of Filters = 128

Input matrix = 40 x 40 x 64

N = 128



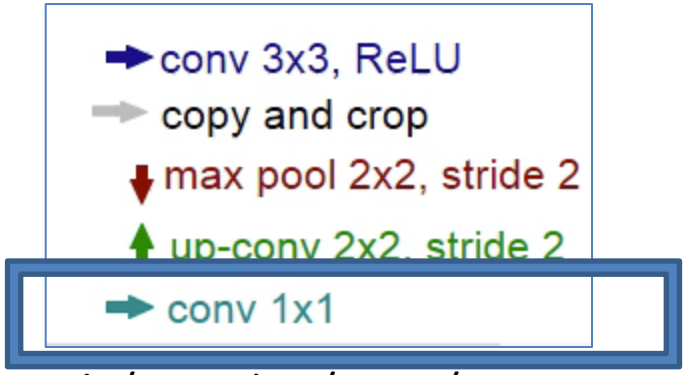
Channel = 64, Filter size = 1 x1, stride = 2, same padding

```
residual = layers.Conv2D(128, 1, stride=2,
                         padding='same')(previous_block_activation)
X = layers.add(x, residual)
```

output matrix = 20 x 20 x 128

parameter, (1x1x64) x 128 = 2048
Bias = 128
Total parameters = 8320

Add to the current X
Size = 20x20x128

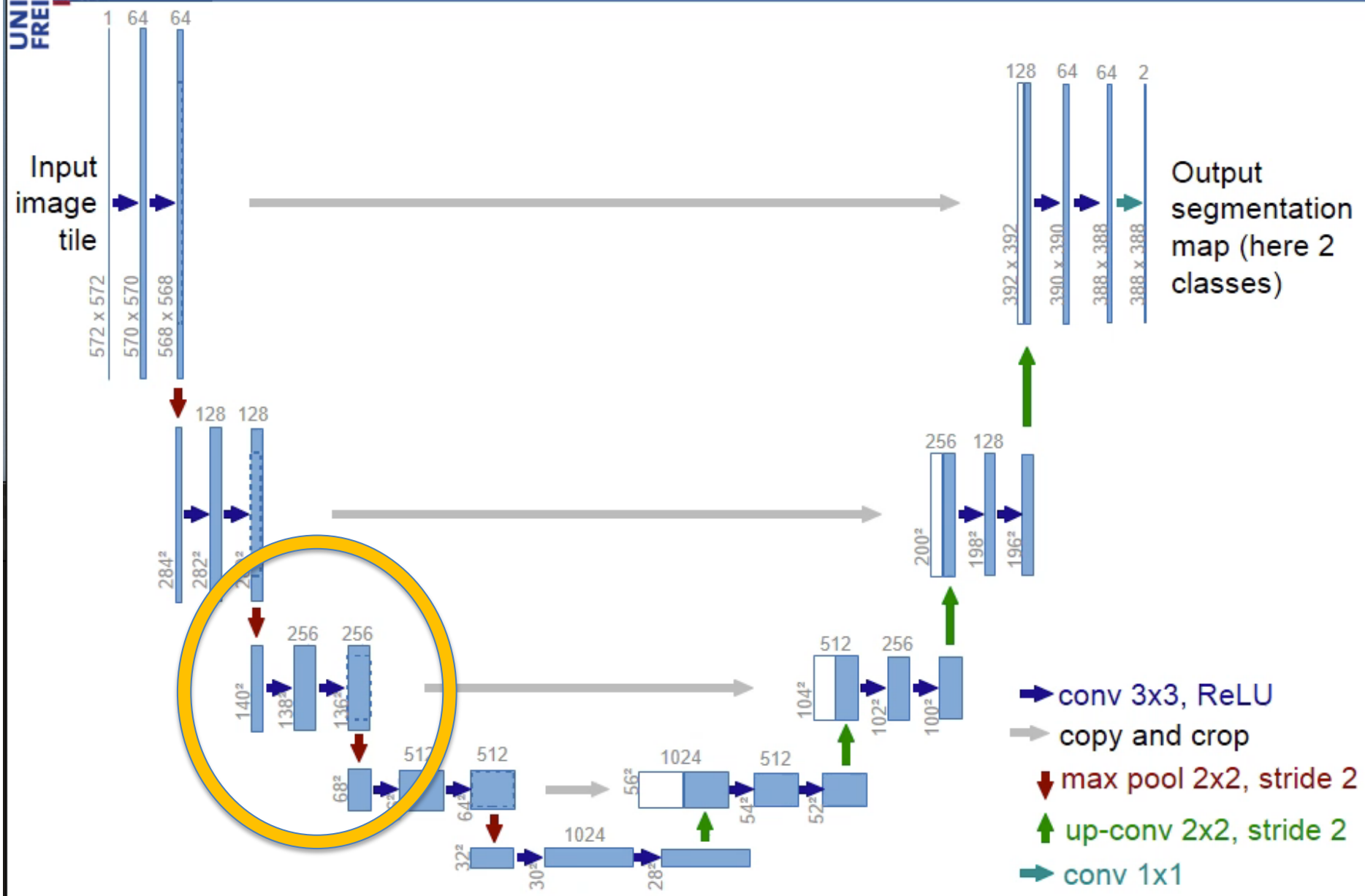


https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

Filter = 128, repeating the block!!

activation_3 (Activation)	(None, 40, 40, 64)	0	add[0][0]
separable_conv2d_2 (SeparableCo	(None, 40, 40, 128)	8896	activation_3[0][0]
batch_normalization_3 (BatchNor	(None, 40, 40, 128)	512	separable_conv2d_2[0]
activation_4 (Activation)	(None, 40, 40, 128)	0	batch_normalization_3
separable_conv2d_3 (SeparableCo	(None, 40, 40, 128)	17664	activation_4[0][0]
batch_normalization_4 (BatchNor	(None, 40, 40, 128)	512	separable_conv2d_3[0]
max_pooling2d_1 (MaxPooling2D)	(None, 20, 20, 128)	0	batch_normalization_4
conv2d_2 (Conv2D)	(None, 20, 20, 128)	8320	add[0][0]
add_1 (Add)	(None, 20, 20, 128)	0	max_pooling2d_1[0][0] conv2d_2[0][0]

U-net Architecture



No of Filters = 256

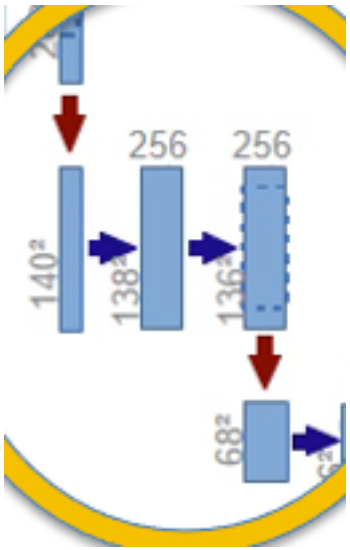
Input matrix = 20 x 20 x 128

N = 256

Filter size = 3 x3, stride = 1, same padding

20x20x128

```
X  = layers.Activation("relu")(x)
X = layers.SeparableConv2D(256, 3,  padding='same')(x)
X = layers. BatchNormalization()(x)
```



output matrix = 20 x 20 x 256

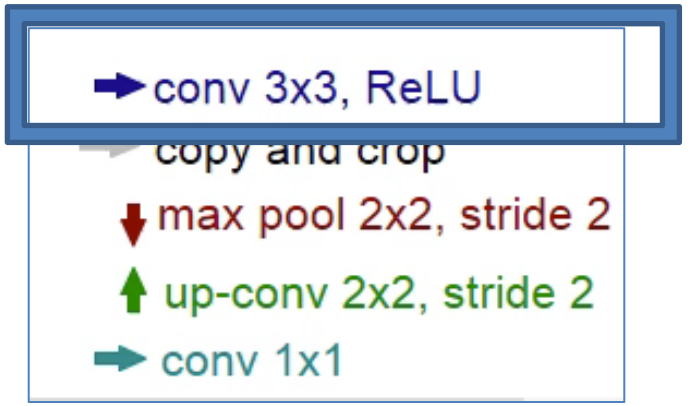
1st step : M = 128, (3x3x1) x 128 = 1152

2nd step : weights = (1 x 1 x 128) x 256 = 32768

Bias = 256

Total parameters = 34176

20x20x256



https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

No of Filters = 256

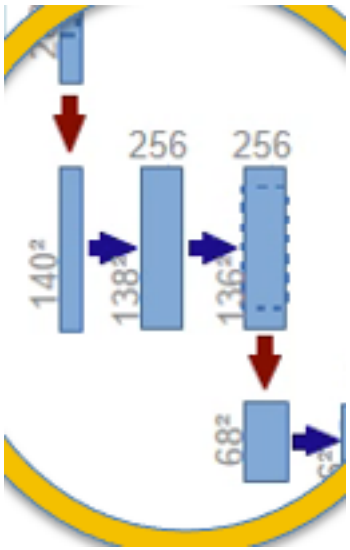
Input matrix = 20 x 20 x 256

N = 256

Filter size = 3 x3, stride = 1, same padding

20x20x256

```
X  = layers.Activation("relu")(x)
X = layers.SeparableConv2D(256, 3,  padding='same')(x)
X = layers. BatchNormalization()(x)
```



output matrix = 20 x 20 x 256

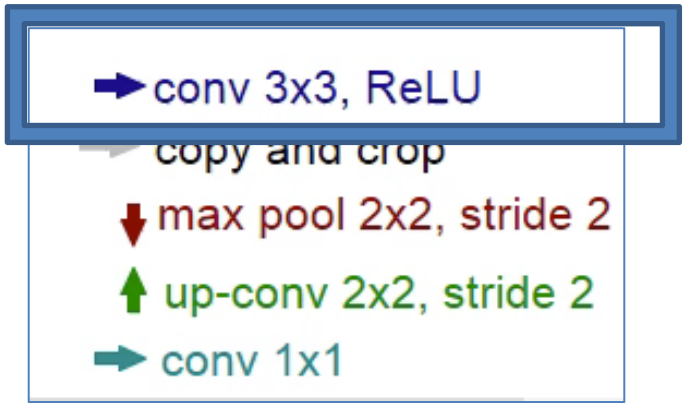
1st step : M = 256, (3x3x1) x 256 = 2304

2nd step : weights = (1 x 1 x 256) x 256 = 65536

Bias = 256

Total parameters = 68096

20x20x256



https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

X = layers. BatchNormalization()(x)

Batch Normalization has 4 parameters per set of filters (feature)
Number of parameters = 4 x number of filters (bias) = 1024

As you can read there, in order to make the batch normalization work during training, they need to keep track of the distributions of each normalized dimensions. To do so, since you are in mode=0by default, they compute 4 parameters per feature on the previous layer. Those parameters are making sure that you properly propagate and backpropagate the information. These parameters are in fact [gamma weights, beta weights, moving_mean(non-trainable), moving_variance(non-trainable)], of the size of the input layer.

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

input matrix = 20 x 20 x 256

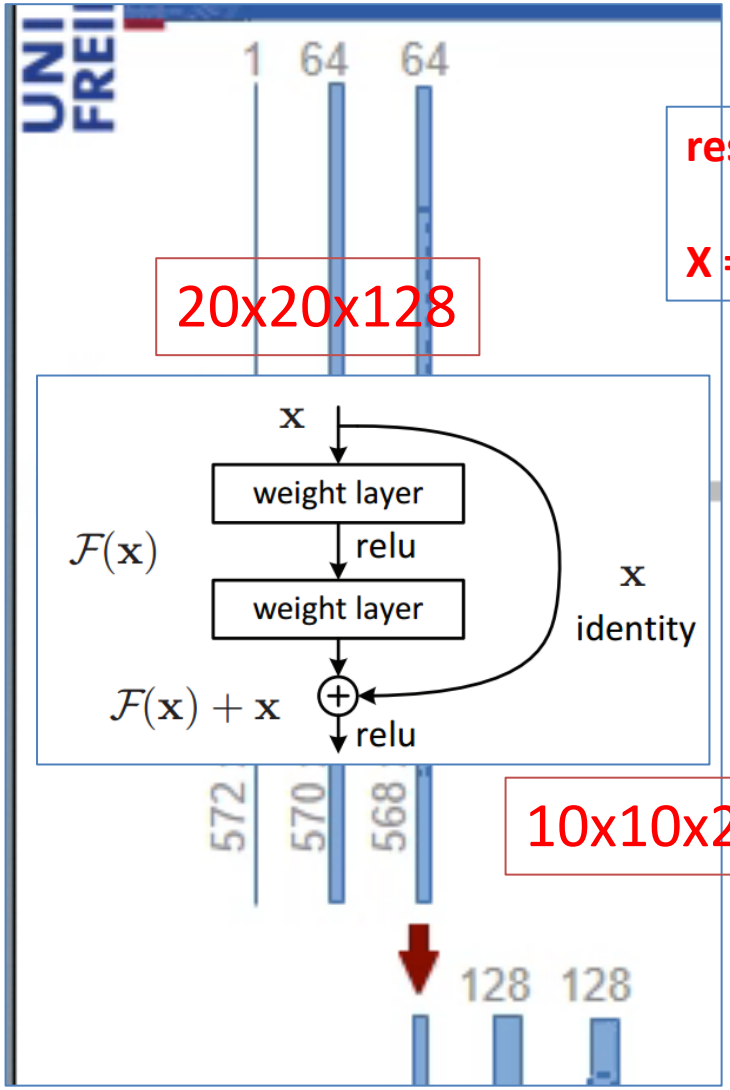
X = layers.MaxPooling2D(3, stride= 2, padding='same')(x)
Reduce size by half because of stride, paddling

output matrix = 10 x 10 x 256

No of Filters = 256

Input matrix = 20 x 20 x 128

N = 256



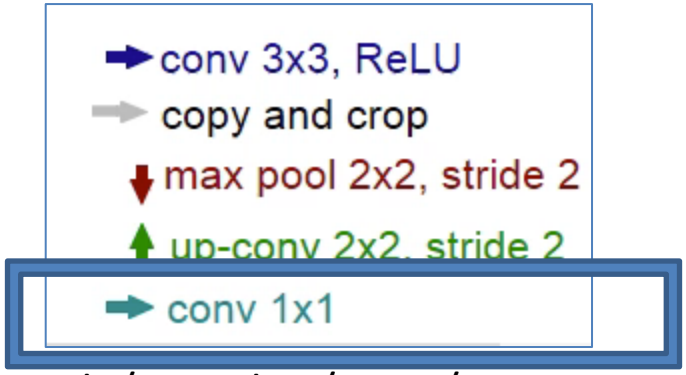
Channel = 128, Filter size = 1 x1, stride = 2, same padding

```
residual = layers.Conv2D(256, 1, stride=2,
                         padding='same')(previous_block_activation)
X = layers.add(x, residual)
```

output matrix = 10 x 10 x 256

parameter, (1x1x128) x 256 = 32768
Bias = 256
Total parameters = 33024

Add to the current X
Size = 10x10x256



https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

Filter = 256, repeating the block!!

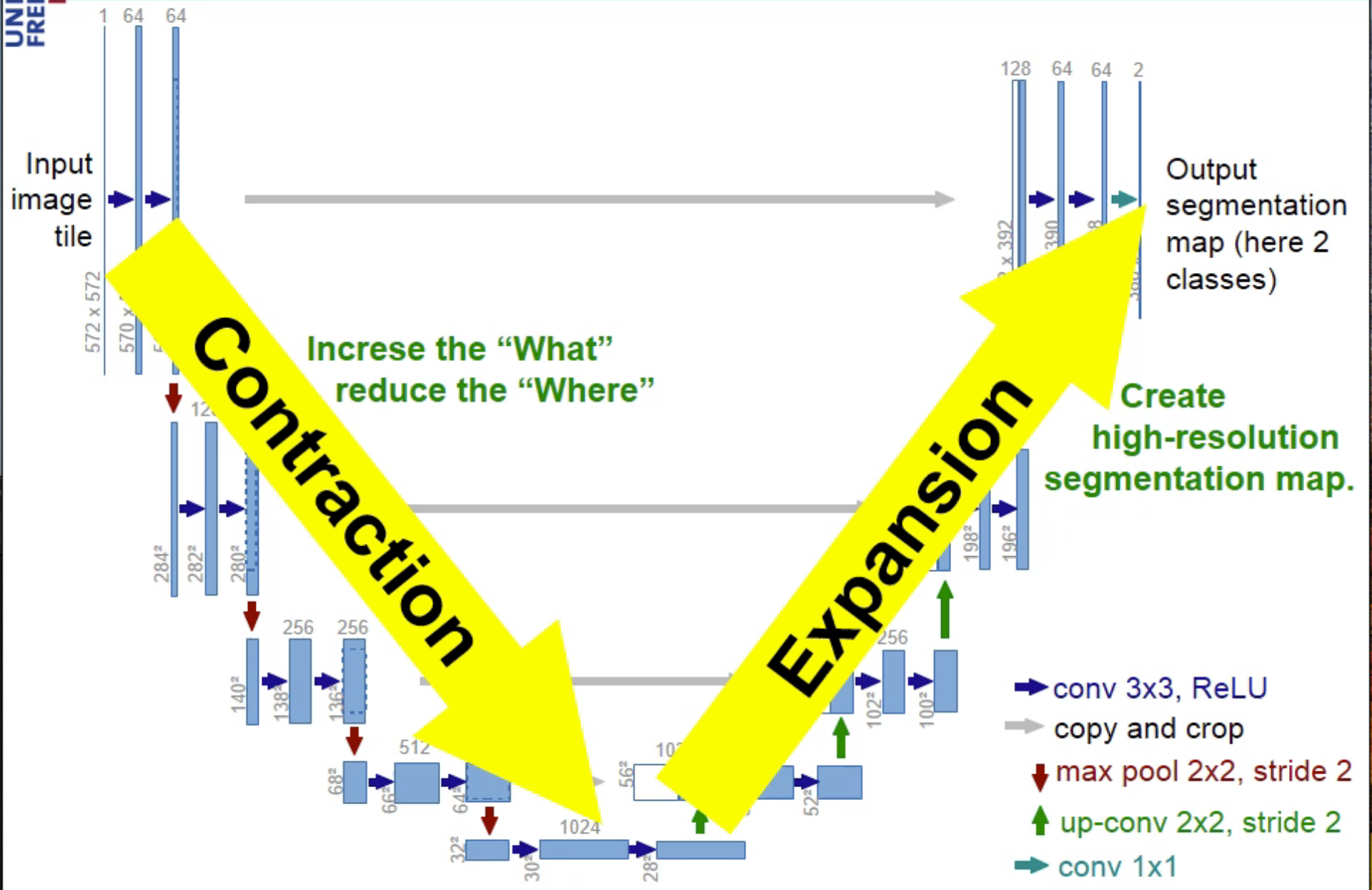
activation_5 (Activation)	(None, 20, 20, 128)	0	add_1[0][0]
separable_conv2d_4 (SeparableCo	(None, 20, 20, 256)	34176	activation_5[0][0]
batch_normalization_5 (BatchNor	(None, 20, 20, 256)	1024	separable_conv2d_4[0]
activation_6 (Activation)	(None, 20, 20, 256)	0	batch_normalization_5
separable_conv2d_5 (SeparableCo	(None, 20, 20, 256)	68096	activation_6[0][0]
batch_normalization_6 (BatchNor	(None, 20, 20, 256)	1024	separable_conv2d_5[0]
max_pooling2d_2 (MaxPooling2D)	(None, 10, 10, 256)	0	batch_normalization_6
conv2d_3 (Conv2D)	(None, 10, 10, 256)	33024	add_1[0][0]
add_2 (Add)	(None, 10, 10, 256)	0	max_pooling2d_2[0][0] conv2d_3[0][0]

```
input_dir = "images/"
target_dir = "annotat
img_size = (160, 160)
num_classes = 3
batch_size = 32
```

```
def get_model(img_size, num_classes):
    inputs = keras.Input(shape=img_size + (3,))

    ### [First half of the network: downsampling inputs] ###

    # Entry block
    x = layers.Conv2D(32, 3, strides=2, padding="same")(inputs)
    x = layers.BatchNormalization()(x)
    x = layers.Activation("relu")(x)

    previous_block_activation = x  # Set aside residual

    # Blocks 1, 2, 3 are identical apart from the feature depth.
    for filters in [64, 128, 256]:
        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.SeparableConv2D(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.MaxPooling2D(3, strides=2, padding="same")(x)

        # Project residual
        residual = layers.Conv2D(filters, 1, strides=2, padding="same")(
            previous_block_activation
        )
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual
```

```
    ### [Second half of the network: upsampling inputs] ###

    for filters in [256, 128, 64, 32]:
        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.UpSampling2D(2)(x)

        # Project residual
        residual = layers.UpSampling2D(2)(previous_block_activation)
        residual = layers.Conv2D(filters, 1, padding="same")(residual)
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    # Add a per-pixel classification layer
    outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    # Define the model
    model = keras.Model(inputs, outputs)
    return model
```

U-Net Architecture



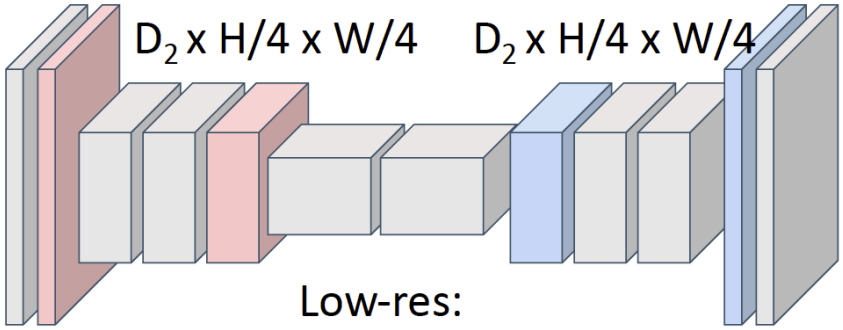
Semantic Segmentation: Fully Convolutional Network

Downsampling: Pooling, strided convolution

Design network as a bunch of convolutional layers, with **downsampling** and **upsampling** inside the network!

Upsampling: ???

Input:
3 x H x W



Predictions:
H x W

Long, Shelhamer, and Darrell, "Fully Convolutional Networks for Semantic Segmentation", CVPR 2015
Noh et al, "Learning Deconvolution Network for Semantic Segmentation", ICCV 2015

https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2020/schedule.html

In-Network Upsampling: “Unpooling”

Bed of Nails

1	2
3	4

→

1	0	2	0
0	0	0	0
3	0	4	0
0	0	0	0

Input
C x 2 x 2

Output
C x 4 x 4

Nearest Neighbor

1	2
3	4

→

1	1	2	2
1	1	2	2
3	3	4	4
3	3	4	4

Input
C x 2 x 2

Output
C x 4 x 4

In-Network Upsampling: “Max Unpooling”

Max Pooling: Remember which position had the max

1	2	6	3
3	5	2	1
1	2	2	1
7	3	4	8

→

5	6
7	8

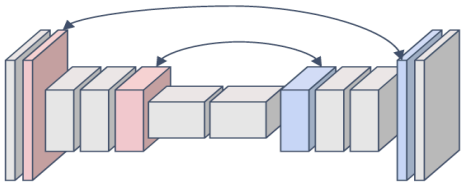
→ Rest of net →

1	2
3	4

→

Max Unpooling: Place into remembered positions

0	0	2	0
0	1	0	0
0	0	0	0
3	0	0	4

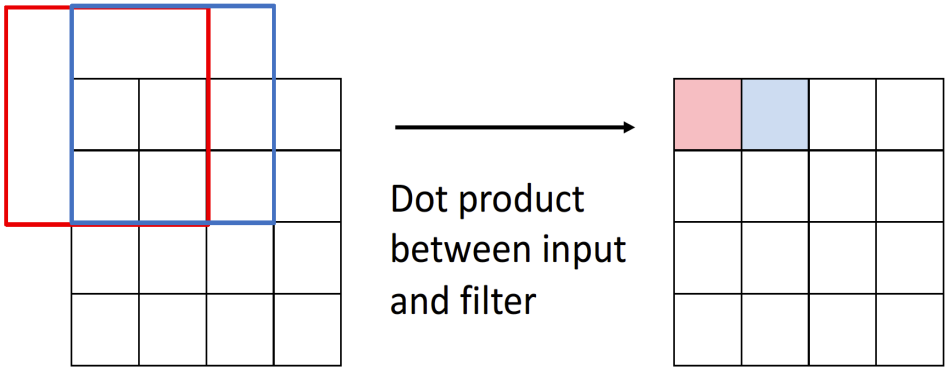


Pair each downsampling layer with an upsampling layer

Noh et al, “Learning Deconvolution Network for Semantic Segmentation”, ICCV 2015

Learnable Upsampling: Transposed Convolution

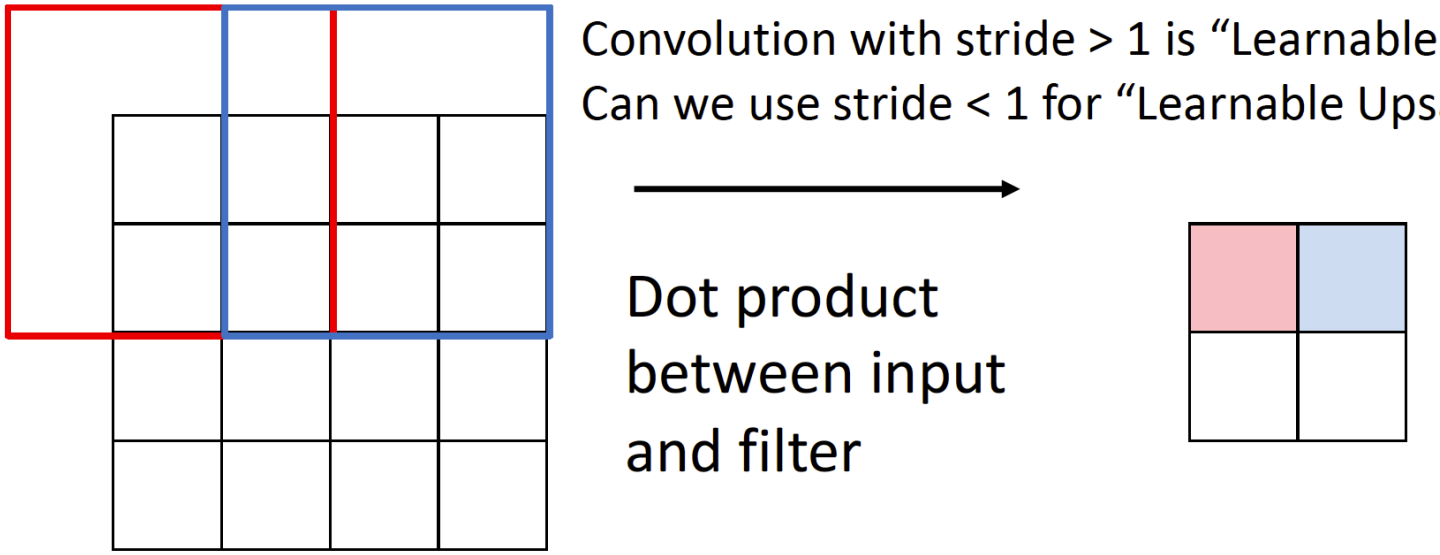
Recall: Normal 3 x 3 convolution, stride 1, pad 1



Input: 4 x 4

Output: 4 x 4

Recall: Normal 3 x 3 convolution, stride 2, pad 1



Input: 4 x 4

Output: 2 x 2

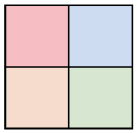
MM in Segmentation Algorithm

Learnable Upsampling: Transposed Convolution

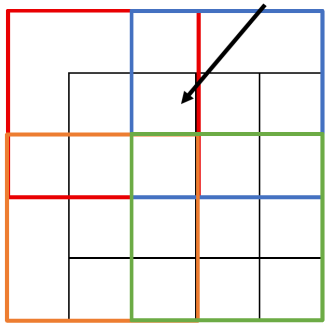
3 x 3 **convolution transpose**, stride 2

Sum where output overlaps

This gives 5x5 output – need to trim one pixel from top and left to give 4x4 output



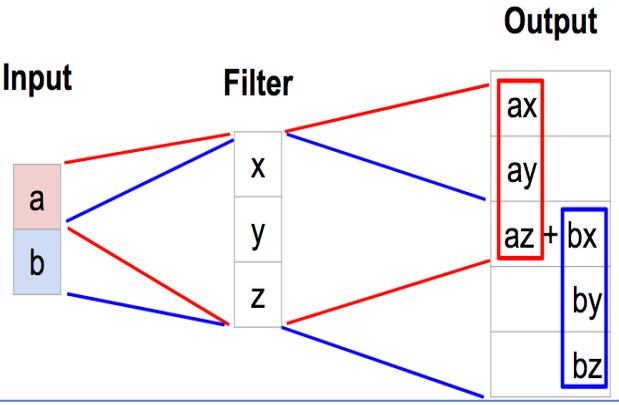
Weight filter by input value and copy to output



Input: 2 x 2

Output: 4 x 4

Transpose Convolution: 1D Example



Output contains copies of the filter weighted by the input, summing at where at overlaps in the output

Need to crop one pixel from output to make output exactly 2x input

https://www.tensorflow.org/tutorials/images/segmentation

Convolution as Matrix Multiplication (1D Example)

We can express convolution in terms of a matrix multiplication

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & x & 0 & 0 & 0 \\ 0 & x & y & x & 0 & 0 \\ 0 & 0 & x & y & x & 0 \\ 0 & 0 & 0 & x & y & x \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ ax + by + cz \\ bx + cy + dz \\ cx + dy \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=1, padding=1

Convolution transpose multiplies by the transpose of the same matrix:

$$\vec{x} *^T \vec{a} = X^T\vec{a}$$

$$\begin{bmatrix} x & 0 & 0 & 0 \\ y & x & 0 & 0 \\ z & y & x & 0 \\ 0 & z & y & x \\ 0 & 0 & z & y \\ 0 & 0 & 0 & z \end{bmatrix} \begin{bmatrix} a \\ b \\ c \\ d \end{bmatrix} = \begin{bmatrix} ax \\ ay + bx \\ az + by + cx \\ bz + cy + dx \\ cz + dy \\ dz \end{bmatrix}$$

When stride=1, convolution transpose is just a regular convolution (with different padding rules)

Convolution as Matrix Multiplication (1D Example)

We can express convolution in terms of a matrix multiplication

$$\vec{x} * \vec{a} = X\vec{a}$$

$$\begin{bmatrix} x & y & z & 0 & 0 & 0 \\ 0 & 0 & x & y & z & 0 \end{bmatrix} \begin{bmatrix} 0 \\ a \\ b \\ c \\ d \\ 0 \end{bmatrix} = \begin{bmatrix} ay + bz \\ bx + cy + dz \end{bmatrix}$$

Example: 1D conv, kernel size=3, stride=2, padding=1

Convolution transpose multiplies by the transpose of the same matrix:

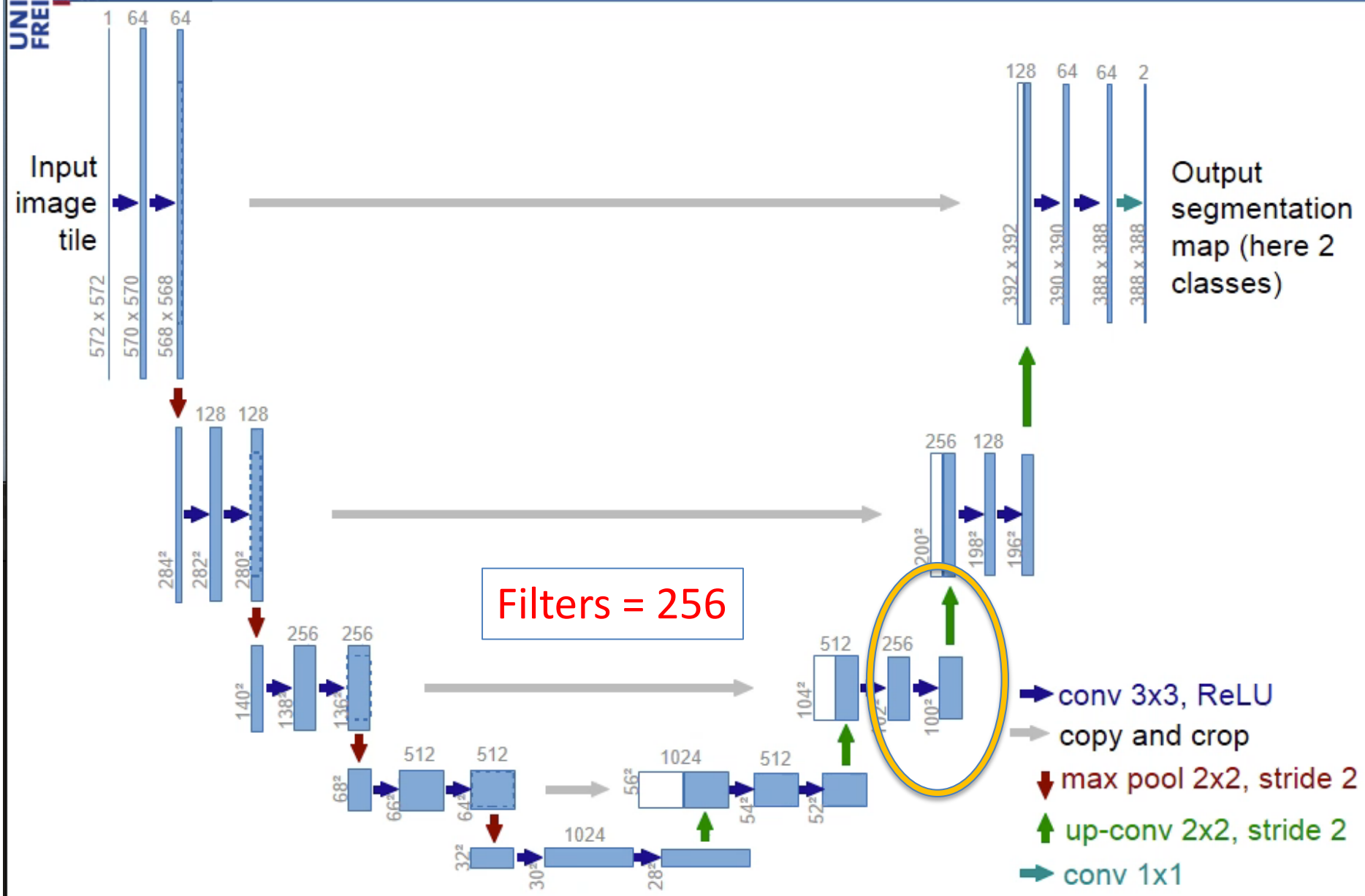
$$\vec{x} *^T \vec{a} = X^T\vec{a}$$

$$\begin{bmatrix} x & 0 \\ y & 0 \\ z & x \\ 0 & y \\ 0 & z \\ 0 & 0 \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} ax \\ ay \\ az + bx \\ by \\ bz \\ 0 \end{bmatrix}$$

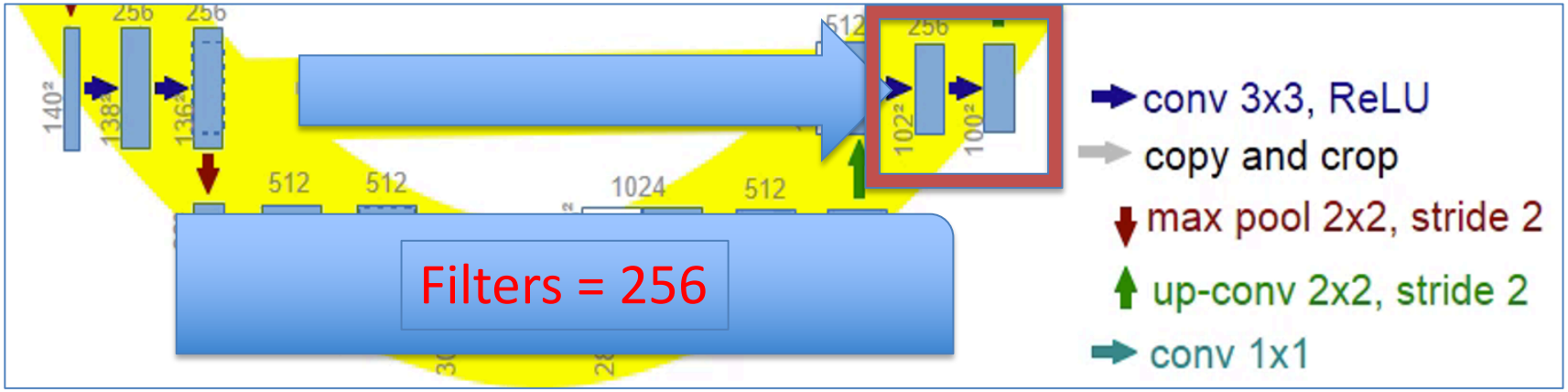
When stride>1, convolution transpose is no longer a normal convolution!

https://web.eecs.umich.edu/~justincj/teaching/eecs498/FA2020/schedule.html

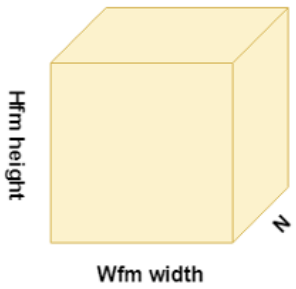
U-net Architecture



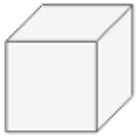
```
    ### [Second half of the network: upsampling inputs] ###

    for filters in 256, 128, 64, 32]:
        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.UpSampling2D(2)(x)

        # Project residual
        residual = layers.UpSampling2D(2)(previous_block_activation)
        residual = layers.Conv2D(filters, 1, padding="same")(residual)
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    # Add a per-pixel classification layer
    outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    # Define the model
    model = keras.Model(inputs, outputs)
    return model
```



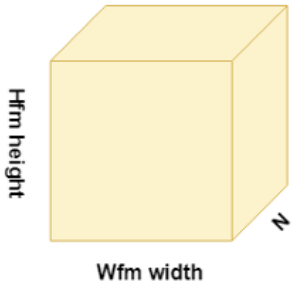
Input Channel = 256



N=256
Size = (3x3 x 256)

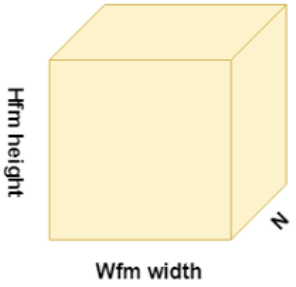


output Channel = 256

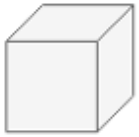


Height = 10 ; width = 10; N = 256. => parameters = (3x3x256) x 256 + 256 = 590080

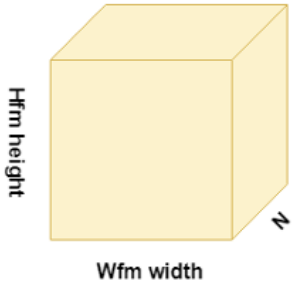
Input Channel = 256



N=256
Size = (3x3 x 256)



output Channel = 256



Height = 10 ; width = 10; N = 256. => parameters = (3x3x256) x 256 + 256 = 590080

```
    ### [Second half of the network: upsampling inputs] ###

    for filters in [256, 128, 64, 32]:
        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.UpSampling2D(2)(x)

        # Project residual
        residual = layers.UpSampling2D(2)(previous_block_activation)
        residual = layers.Conv2D(filters, 1, padding="same")(residual)
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    # Add a per-pixel classification layer
    outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    # Define the model
    model = keras.Model(inputs, outputs)
    return model
```

Previous : 10x10x256, Upsampling2D(2)
Reverse of max pooling of 2x2

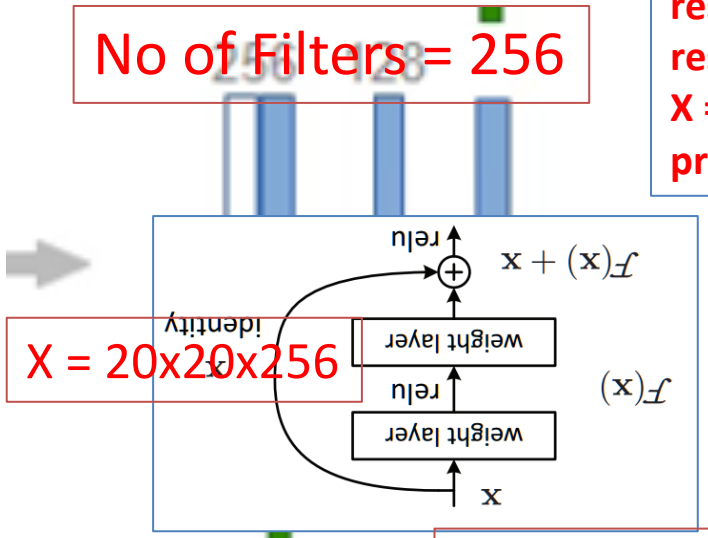
Input matrix = 20 x 20 x 256

N = 256

Channel = 256, Filter size = 1 x1, stride = 1, same padding

```
residual = layers.Upsampling2D(2)(previous_block_activation)
residual = layers.Conv2D(256, 1, padding='same')(residual)
X = layers.add(x, residual)
previous_block_activation  = x
```

No of Filters = 256



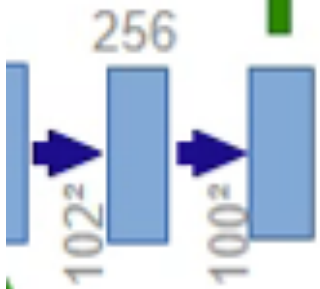
output matrix = 20 x 20 x 256

X = 20x20x256

parameter, (1x1x256) x 256 + 256
Total parameters = 65792

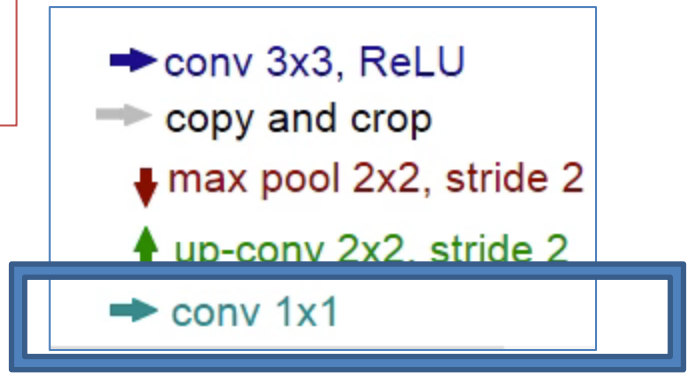
20x20x256

Add to the current X
Size = 20x20x256

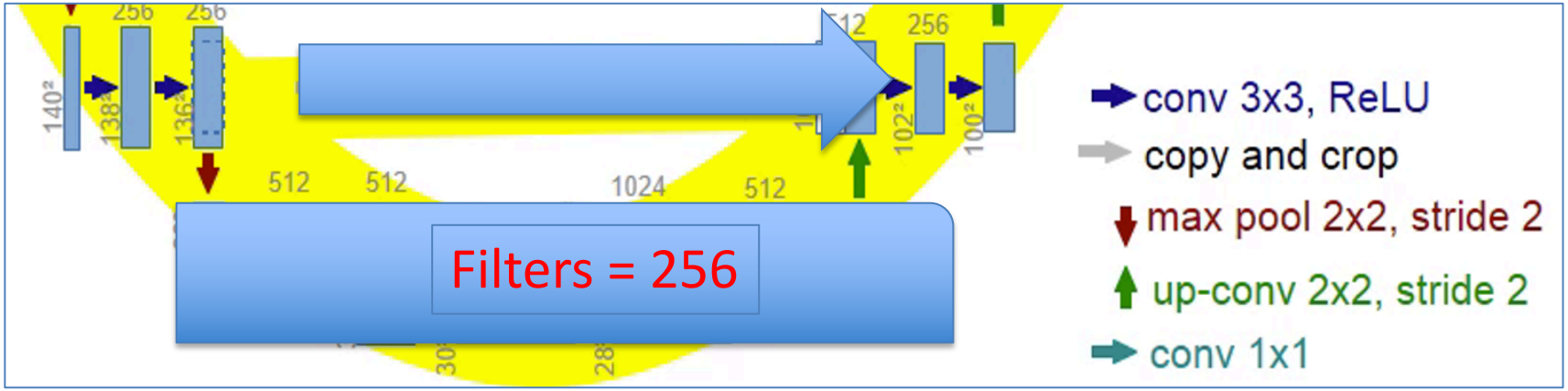


Upsampling2D(2)
Reverse of max pooling of 2x2

Previous block
10x10x256



https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

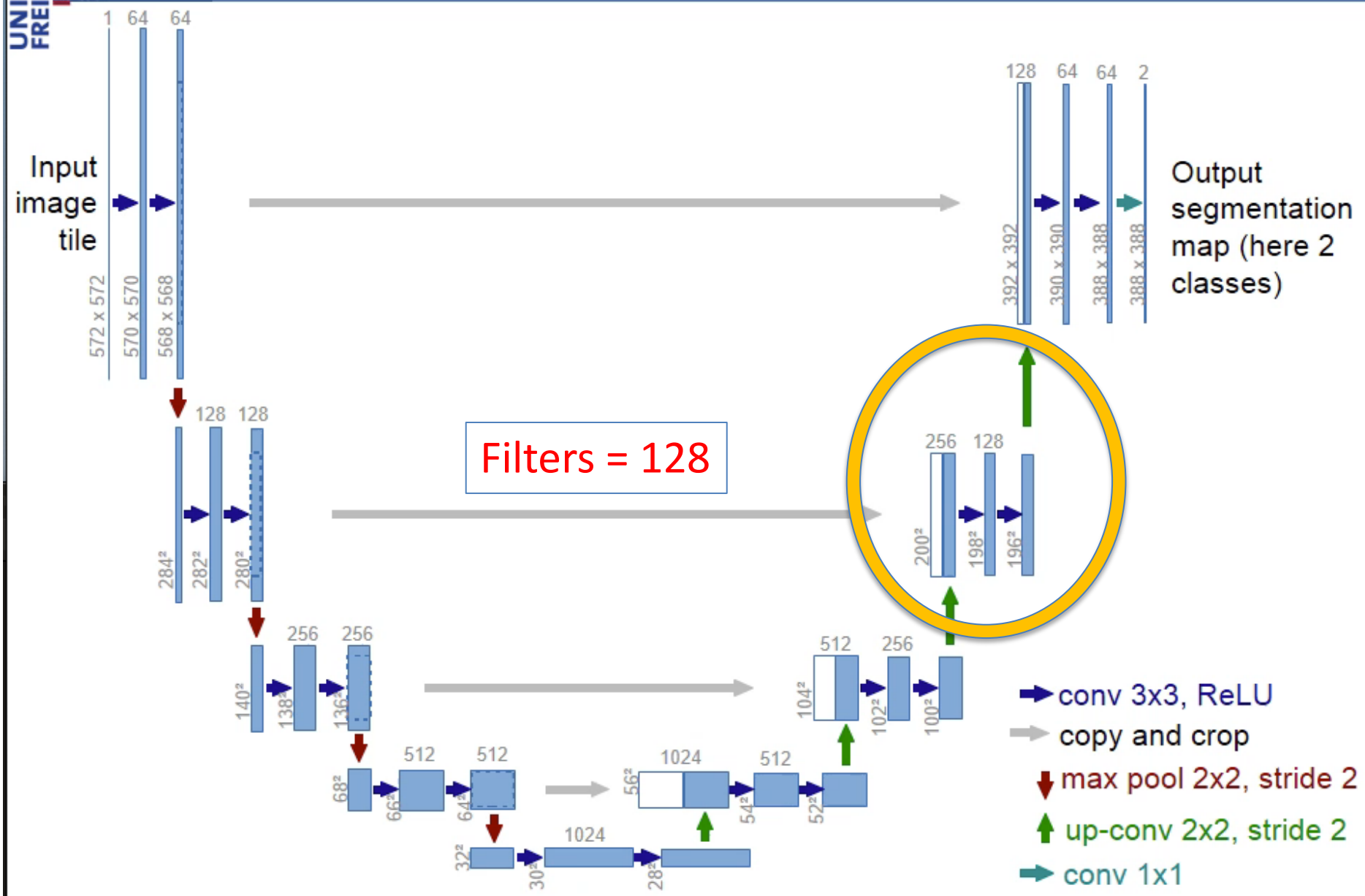


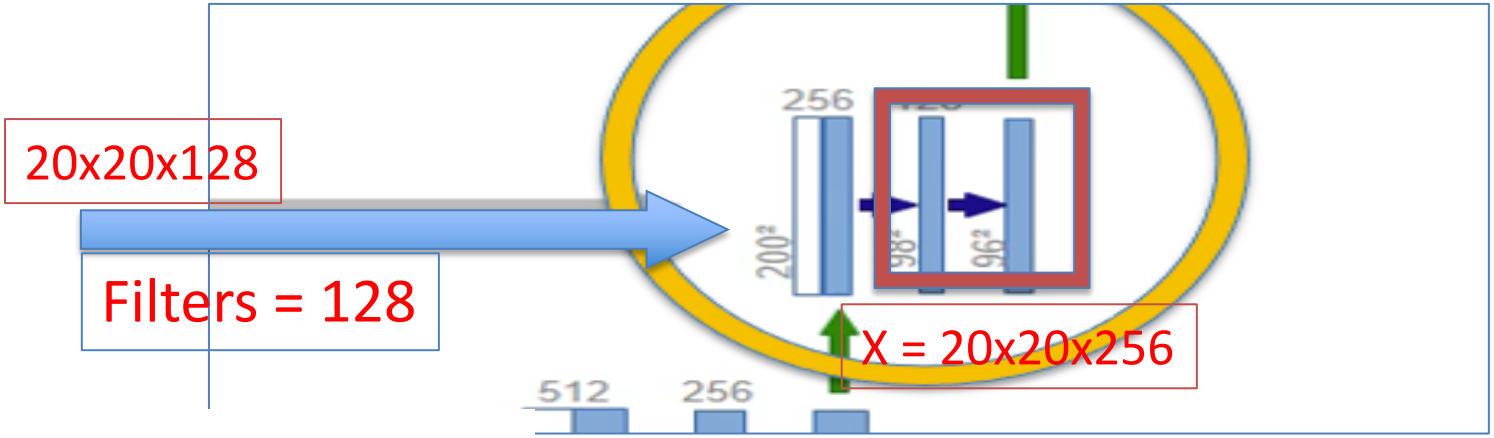
activation_7 (Activation)	(None, 10, 10, 256)	0	add_2[0][0]
conv2d_transpose (Conv2DTranspo	(None, 10, 10, 256)	590080	activation_7[0][0]
batch_normalization_7 (BatchNor	(None, 10, 10, 256)	1024	conv2d_transpose[0][0
activation_8 (Activation)	(None, 10, 10, 256)	0	batch_normalization_7
conv2d_transpose_1 (Conv2DTrans	(None, 10, 10, 256)	590080	activation_8[0][0]
batch_normalization_8 (BatchNor	(None, 10, 10, 256)	1024	conv2d_transpose_1[0]
up_sampling2d_1 (UpSampling2D)	(None, 20, 20, 256)	0	add_2[0][0]
up_sampling2d (UpSampling2D)	(None, 20, 20, 256)	0	batch_normalization_8
conv2d_4 (Conv2D)	(None, 20, 20, 256)	65792	up_sampling2d_1[0][0]
add_3 (Add)	(None, 20, 20, 256)	0	up_sampling2d[0][0] conv2d_4[0][0]

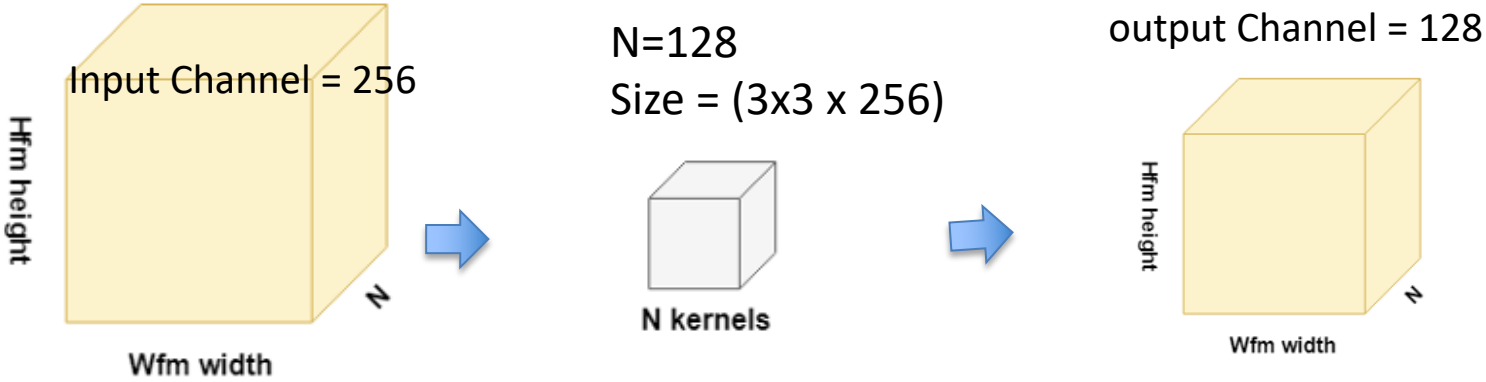
Revserse Max Pooling => no parametes

```
    ### [Second half of the network: upsampling inputs] ###

    for filters in [256, 128, 64, 32]:
        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.UpSampling2D(2)(x)

        # Project residual
        residual = layers.UpSampling2D(2)(previous_block_activation)
        residual = layers.Conv2D(filters, 1, padding="same")(residual)
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    # Add a per-pixel classification layer
    outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    # Define the model
    model = keras.Model(inputs, outputs)
    return model
```

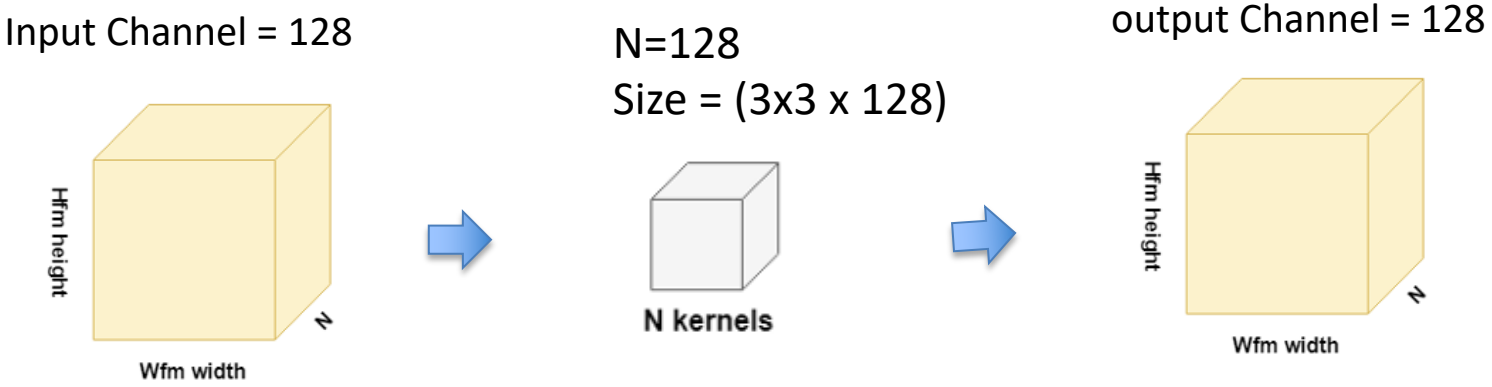
U-net Architecture







Height = 20 ; width = 20; N = 128. => parameters = (3x3x256) x 128 + 128 = 295040



Height = 20 ; width = 20; N = 128 => parameters = (3x3x128) x 128 + 128 = 147587

```
    ### [Second half of the network: upsampling inputs] ###

    for filters in [256, 128, 64, 32]:
        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.UpSampling2D(2)(x)

        # Project residual
        residual = layers.UpSampling2D(2)(previous_block_activation)
        residual = layers.Conv2D(filters, 1, padding="same")(residual)
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    # Add a per-pixel classification layer
    outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    # Define the model
    model = keras.Model(inputs, outputs)
    return model
```

Previous : 20x20x256, Upsampling2D(2)
Reverse of max pooling of 2x2

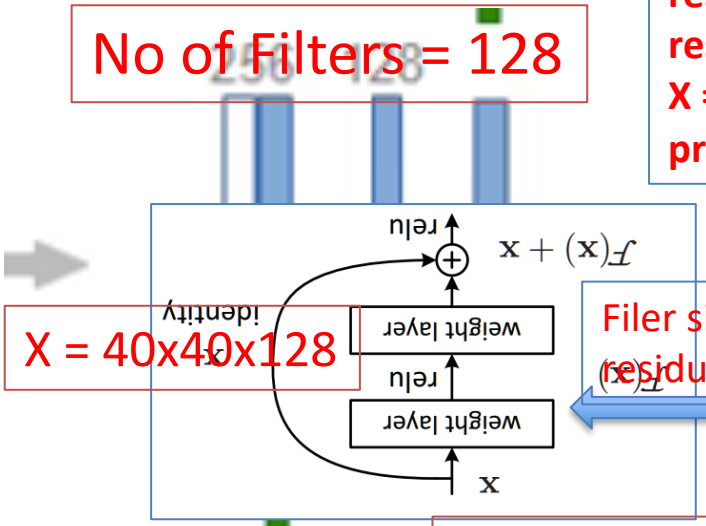
Input matrix = 40 x 40 x 256

N = 128

Channel = 256, Filter size = 1 x1, stride = 1, same padding

```
residual = layers.Upsampling2D(2)(previous_block_activation)
residual = layers.Conv2D(128, 1, padding='same')(residual)
X = layers.add(x, residual)
previous_block_activation  = x
```

No of Filters = 128



output matrix = 40 x 40 x 128

X = 40x40x128

Filer size of the residual block

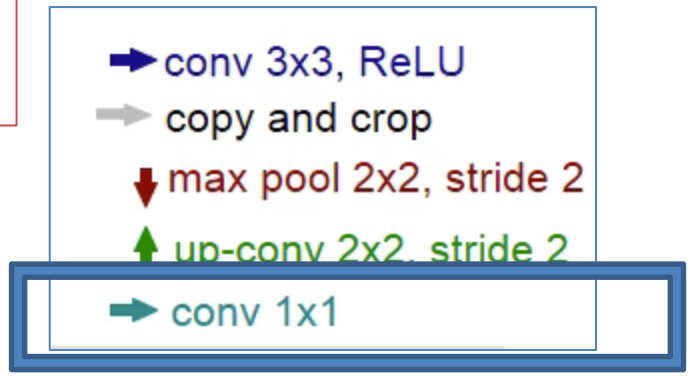
parameter, (1x1x256) x 128 + 128
Total parameters = 32890

40x40x256

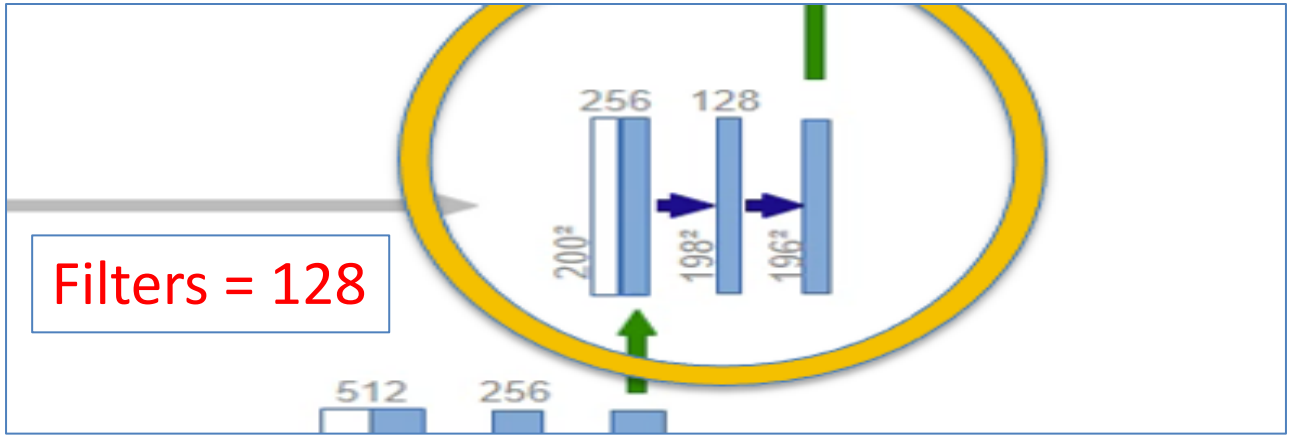
Add to the current X
Size = 40x40x128

Upsampling2D(2)
Reverse of max pooling of 2x2

Previous block
20x20x256



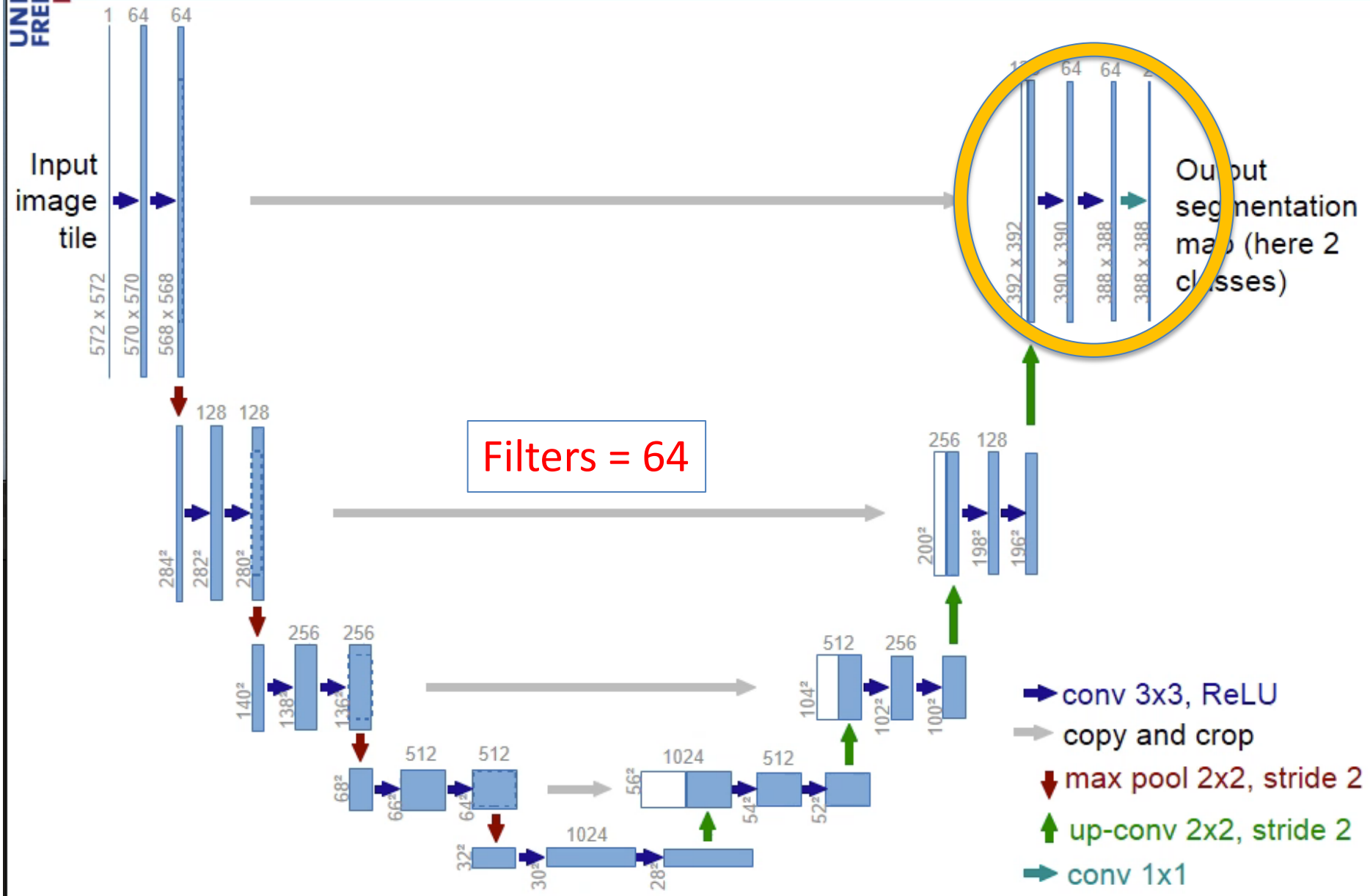
https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

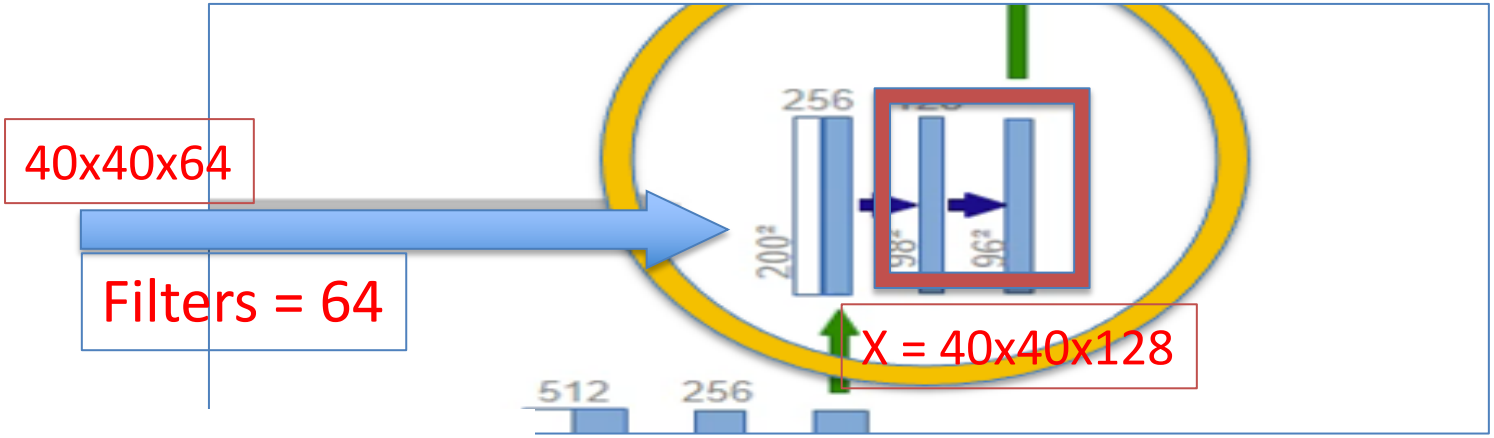


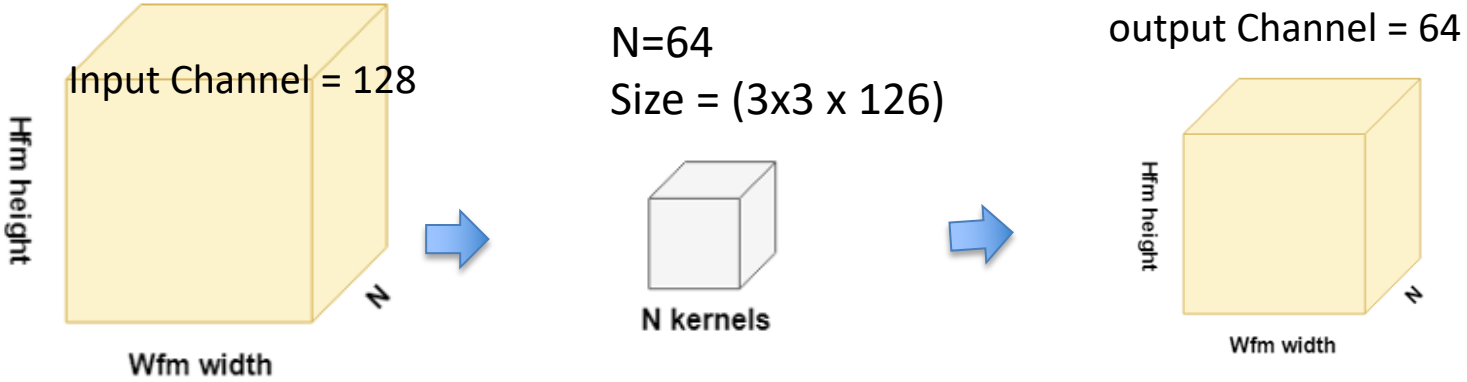
activation_9 (Activation)	(None, 20, 20, 256)	0	add_3[0][0]
conv2d_transpose_2 (Conv2DTrans	(None, 20, 20, 128)	295040	activation_9[0][0]
batch_normalization_9 (BatchNor	(None, 20, 20, 128)	512	conv2d_transpose_2[0]
activation_10 (Activation)	(None, 20, 20, 128)	0	batch_normalization_9
conv2d_transpose_3 (Conv2DTrans	(None, 20, 20, 128)	147584	activation_10[0][0]
batch_normalization_10 (BatchNo	(None, 20, 20, 128)	512	conv2d_transpose_3[0]
up_sampling2d_3 (UpSampling2D)	(None, 40, 40, 256)	0	add_3[0][0]
up_sampling2d_2 (UpSampling2D)	(None, 40, 40, 128)	0	batch_normalization_1
conv2d_5 (Conv2D)	(None, 40, 40, 128)	32896	up_sampling2d_3[0][0
add_4 (Add)	(None, 40, 40, 128)	0	up_sampling2d_2[0][0] conv2d_5[0][0]

```
    ### [Second half of the network: upsampling inputs] ###

    for filters in [256, 128, 64, 32]:
        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.UpSampling2D(2)(x)

        # Project residual
        residual = layers.UpSampling2D(2)(previous_block_activation)
        residual = layers.Conv2D(filters, 1, padding="same")(residual)
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    # Add a per-pixel classification layer
    outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    # Define the model
    model = keras.Model(inputs, outputs)
    return model
```

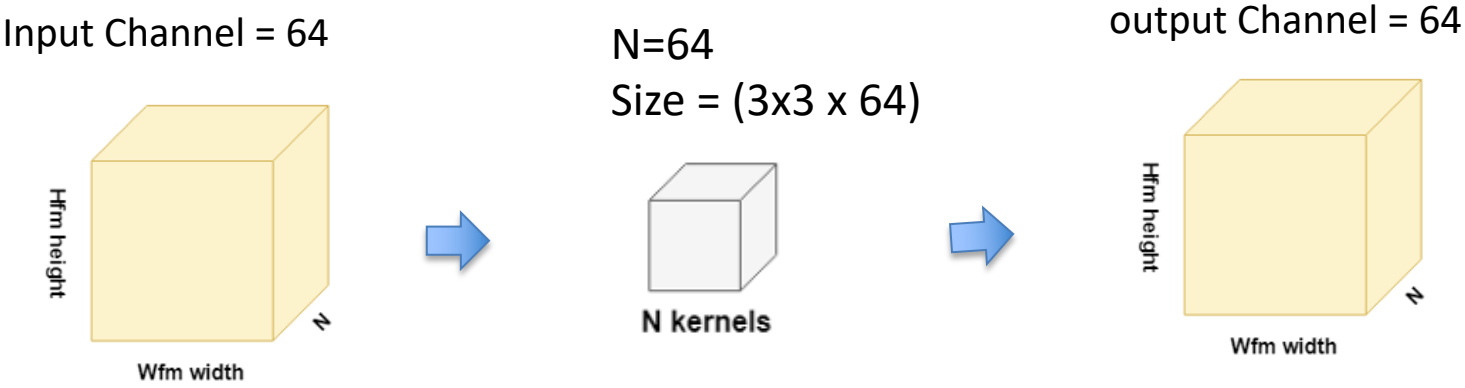
U-net Architecture







Height = 40 ; width = 40; N = 128. => parameters = (3x3x128) x 64 + 64 = 73792



Height = 40 ; width = 20; N = 64 => parameters = (3x3x64) x 64 + 64 = 36928

```
    ### [Second half of the network: upsampling inputs] ###

    for filters in [256, 128, 64, 32]:
        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.Activation("relu")(x)
        x = layers.Conv2DTranspose(filters, 3, padding="same")(x)
        x = layers.BatchNormalization()(x)

        x = layers.UpSampling2D(2)(x)

        # Project residual
        residual = layers.UpSampling2D(2)(previous_block_activation)
        residual = layers.Conv2D(filters, 1, padding="same")(residual)
        x = layers.add([x, residual])  # Add back residual
        previous_block_activation = x  # Set aside next residual

    # Add a per-pixel classification layer
    outputs = layers.Conv2D(num_classes, 3, activation="softmax", padding="same")(x)

    # Define the model
    model = keras.Model(inputs, outputs)
    return model
```

Previous : 40x40x128, Upsampling2D(2)
Reverse of max pooling of 2x2

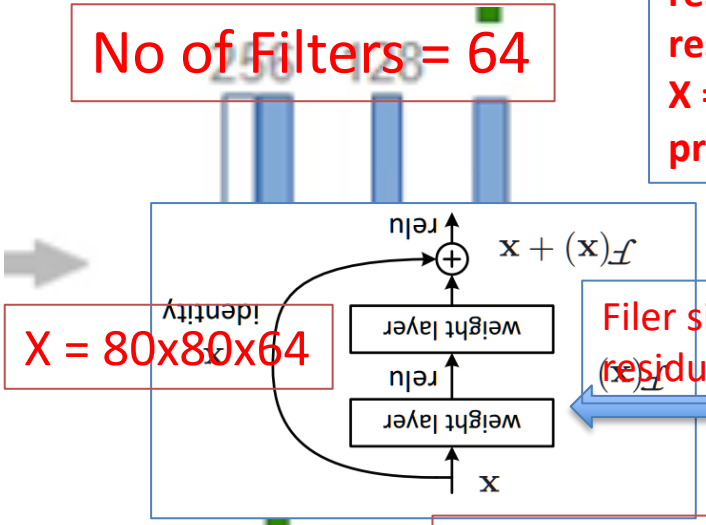
Input matrix = 80 x 80 x 128

N = 64

Channel = 256, Filter size = 1 x1, stride = 1, same padding

```
residual = layers.Upsampling2D(2)(previous_block_activation)
residual = layers.Conv2D(128, 1, padding='same')(residual)
X = layers.add(x, residual)
previous_block_activation  = x
```

No of Filters = 64



output matrix = 80 x 80 x 64

X = 80x80x64

Filer size of the residual block

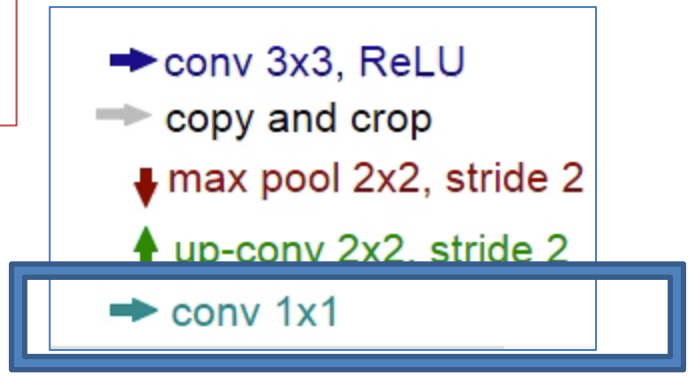
parameter, (1x1x128) x 64 + 64
Total parameters = 8256

80x80x128

Add to the current X
Size = 80x80x64

Upsampling2D(2)
Reverse of max pooling of 2x2

Previous block
40x40x128



https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/

activation_11 (Activation)	(None, 40, 40, 128)	0	add_4[0][0]
conv2d_transpose_4 (Conv2DTrans	(None, 40, 40, 64)	73792	activation_11[0][0]
batch_normalization_11 (BatchNo	(None, 40, 40, 64)	256	conv2d_transpose_4[0]
activation_12 (Activation)	(None, 40, 40, 64)	0	batch_normalization_1
conv2d_transpose_5 (Conv2DTrans	(None, 40, 40, 64)	36928	activation_12[0][0]
batch_normalization_12 (BatchNo	(None, 40, 40, 64)	256	conv2d_transpose_5[0]
up_sampling2d_5 (UpSampling2D)	(None, 80, 80, 128)	0	add_4[0][0]
up_sampling2d_4 (UpSampling2D)	(None, 80, 80, 64)	0	batch_normalization_1
conv2d_6 (Conv2D)	(None, 80, 80, 64)	8256	
add_5 (Add)	(None, 80, 80, 64)	0	

Total params: 2,058,979
Trainable params: 2,055,203
Non-trainable params: 3,776

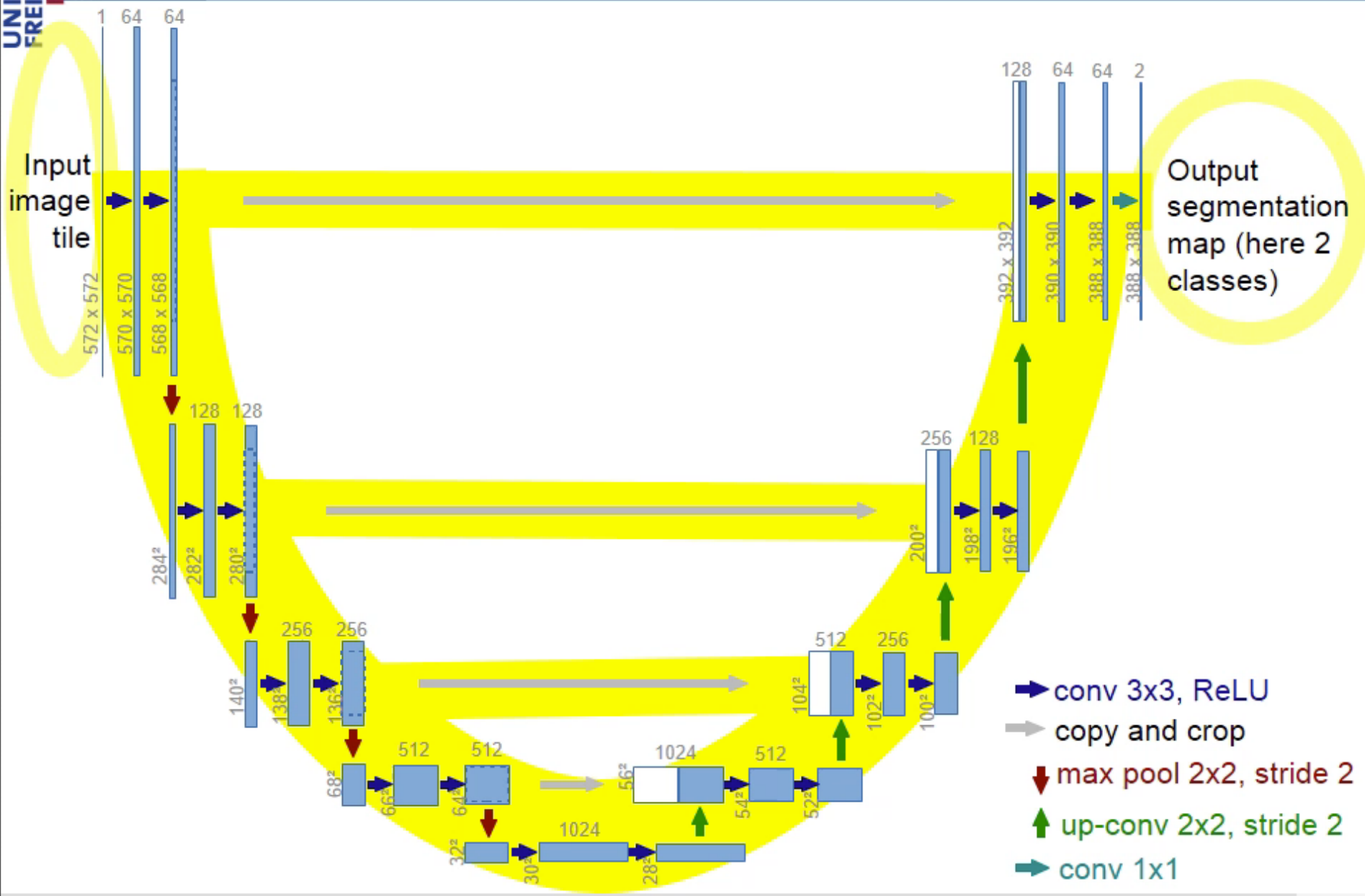
activation_13 (Activation)	(None, 80, 80, 64)	0	add_5[0][0]
conv2d_transpose_6 (Conv2DTrans	(None, 80, 80, 32)	18464	activation_13[0][0]
batch_normalization_13 (BatchNo	(None, 80, 80, 32)	128	conv2d_transpose_6[0]
activation_14 (Activation)	(None, 80, 80, 32)	0	batch_normalization_1
conv2d_transpose_7 (Conv2DTrans	(None, 80, 80, 32)	9248	activation_14[0][0]
batch_normalization_14 (BatchNo	(None, 80, 80, 32)	128	conv2d_transpose_7[0]
up_sampling2d_7 (UpSampling2D)	(None, 160, 160, 64)	0	add_5[0][0]
up_sampling2d_6 (UpSampling2D)	(None, 160, 160, 32)	0	batch_normalization_1
conv2d_7 (Conv2D)	(None, 160, 160, 32)	2080	up_sampling2d_7[0][0]
add_6 (Add)	(None, 160, 160, 32)	0	up_sampling2d_6[0][0] conv2d_7[0][0]
conv2d_8 (Conv2D)	(None, 160, 160, 3)	867	add_6[0][0]

output = layers.Conv2D(num_class=3, 1, activation = “softmax”, padding=‘same’) (x)

Filer size of the final conv2D block

parameter, (3x3x32) x 3 + 3
Total parameters = 8256

U-net Architecture



Acknowledgements and References

This portion of the tutorial contains many extracted materials from many online websites and courses. Listed below are the major sites. This portion of the materials is not intended for public distribution. Please visit the sites websites for detail contents. If you are beginner users of DNN, I suggest to read the following list of websites in its order.

1) http://neuralnetworksanddeeplearning.com

2) https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist

3) https://www.deeplearning.ai/deep-learning-specialization/

4) http://cs231n.stanford.edu/

5) MIT 6.S191, https://www.youtube.com/watch?v=njKP3FqW3Sk

6) https://livebook.manning.com/book/deep-learning-with-python/about-this-book/

7) https://www.fast.ai/

8) https://petewarden.com/2015/04/20/why-gemm-is-at-the-heart-of-deep-learning/

9) https://www.nersc.gov/users/training/gpus-for-science/gpus-for-science-2020/

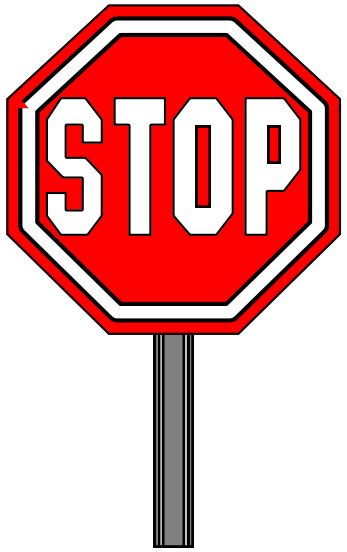
10) https://oneapi-src.github.io/oneDNN/

11) http://www.cs.cornell.edu/courses/cs4787/2020sp/

12) More sites and free books are listed in www.jics.utk.edu/actia → ML

13) https://machinelearningmastery.com

The End



- The End!