

Lecture 11:

Convolution and Image Classification & Segmentation



Markus Hohle

University California, Berkeley

Bayesian Data Analysis and
Machine Learning for Physical
Sciences

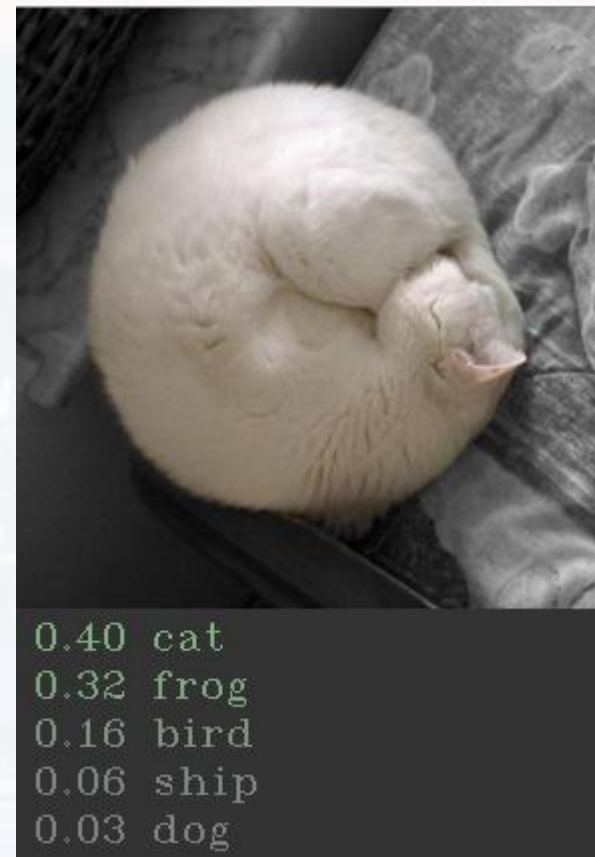


Course Map

Module 1	Maximum Entropy and Information, Bayes Theorem
Module 2	Naive Bayes, Bayesian Parameter Estimation, MAP
Module 3	MLE, Lin Regression
Module 4	Model selection I: Comparing Distributions
Module 5	Model Selection II: Bayesian Signal Detection
Module 6	Variational Bayes, Expectation Maximization
Module 7	Hidden Markov Models, Stochastic Processes
Module 8	Monte Carlo Methods
Module 9	Machine Learning Overview, Supervised Methods & Unsupervised Methods
Module 10	ANN: Perceptron, Backpropagation, SGD
Module 11	Convolution and Image Classification and Segmentation
Module 12	RNNs and LSTMs
Module 13	RNNs and LSTMs + CNNs
Module 14	Transformer and LLMs
Module 15	Graphs & GNNs



Part II





Outline

The Problem

Convolution

CNN Architectures

Data Preparation & Training

Example

- LeNet numpy only
- LeNet TensorFlow
- sequences as images
- segmentation



- LeNet:
- Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998
 - one of the 1st CNN that was able to categorize images
 - MNIST data set

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

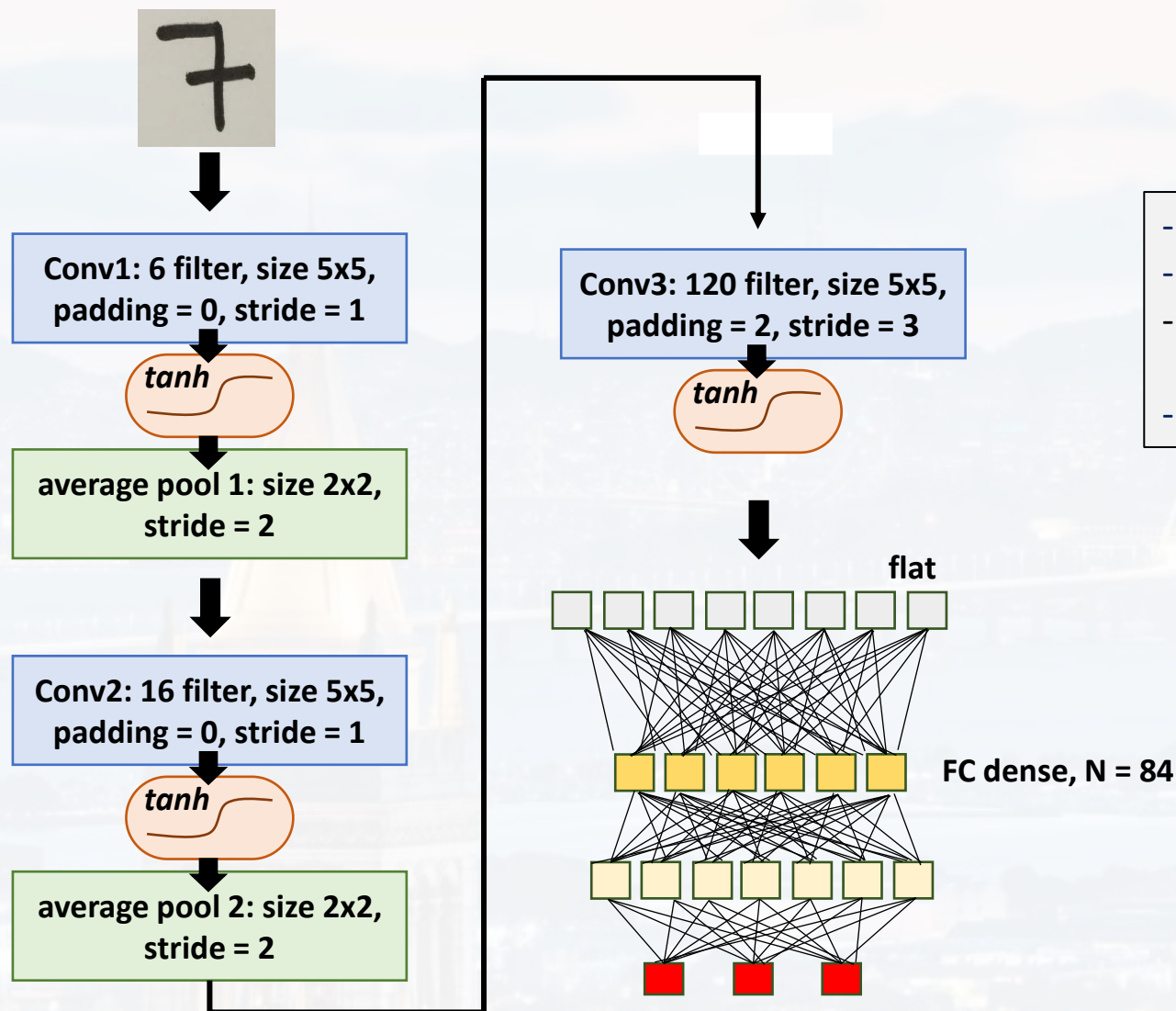


- only **seven layers** in total
- modern CNNs (google, ResNet etc have **100 or more** layers)



LeNet: - Yann LeCun, Leon Bottou, Yoshua Bengio and Patrick Haffner, 1998

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



- ***tanh*** as activation function
- **three different** filter
- from Conv1 → Conv2 not all the **channels are combined**
- **average** pool



We know already how the forward part works:

```
for c in range(numChans):# loop over channels
    for y in range(yOutput):# loop over y axis of output
        for x in range(xOutput):# loop over x axis of output

            # finding corners of the current "slice"
            y_start = y*stride
            y_end    = y*stride + yK
            x_start = x*stride
            x_end    = x*stride + xK

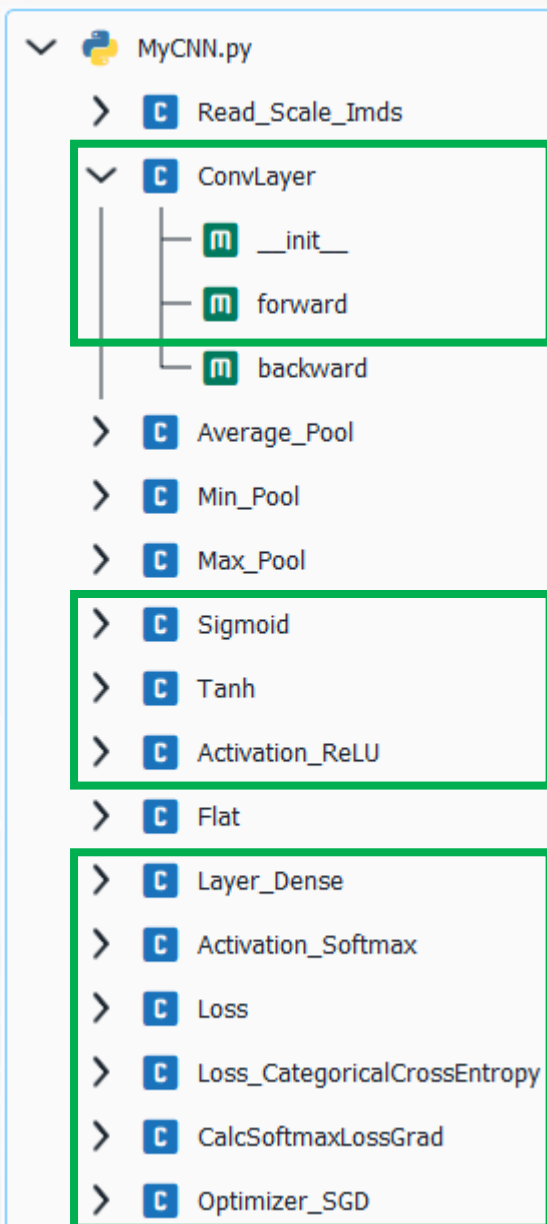
            #selecting the current part of the image
            current_slice = imagePadded[x_start:x_end,\
                                       y_start:y_end, c]

            #the actual convolution part
            s              = np.multiply(current_slice, K)
            output[x,y,c] = np.sum(s)
```

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



see MyCNN.py



we know these
parts already

```
import matplotlib.pyplot as plt
from MyCNN import *
```

```
read_and_scale = Read_Scale_Imds(5, [250, 250])
[I, _] = read_and_scale.Read_Scale()
```

```
Conv1 = ConvLayer(3, 3, 5)
Conv2 = ConvLayer(5, 5, 4)
Conv3 = ConvLayer(2, 2, 4)
```

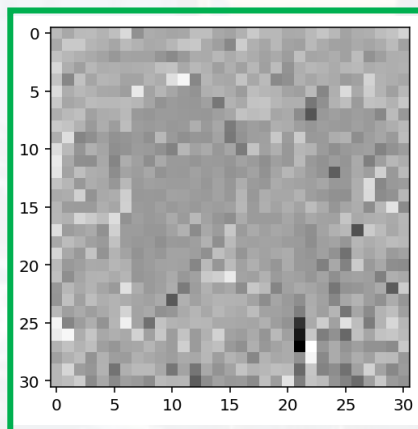
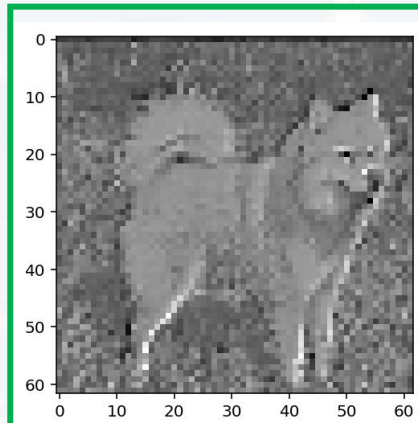
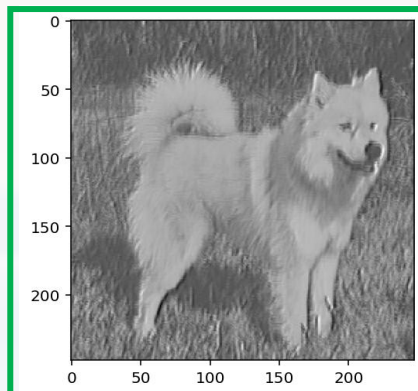
initializing convolution layer
(kernel size x kernel size,
number of kernels)

```
Conv1.forward(I, 0, 1)
Conv2.forward(Conv1.output, 2, 4)
Conv3.forward(Conv2.output, 0, 2)
```

passing image through
convolution layer
(padding, stride length)

```
plt.imshow(Conv1.output[:, :, 0, 4], cmap = 'gray')
plt.show()
plt.imshow(Conv2.output[:, :, 0, 4], cmap = 'gray')
plt.show()
plt.imshow(Conv3.output[:, :, 0, 4], cmap = 'gray')
plt.show()
```

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



```
import matplotlib.pyplot as plt
from MyCNN import *
```

```
read_and_scale = Read_Scale_Imds(5, [250, 250])
[I, _] = read_and_scale.Read_Scale()
```

```
Conv1 = ConvLayer(3, 3, 5)
Conv2 = ConvLayer(5, 5, 4)
Conv3 = ConvLayer(2, 2, 4)
```

```
Conv1.forward(I, 0, 1)
Conv2.forward(Conv1.output, 2, 4)
Conv3.forward(Conv2.output, 0, 2)
```

```
plt.imshow(Conv1.output[:, :, 0, 4], cmap = 'gray')
plt.show()
```

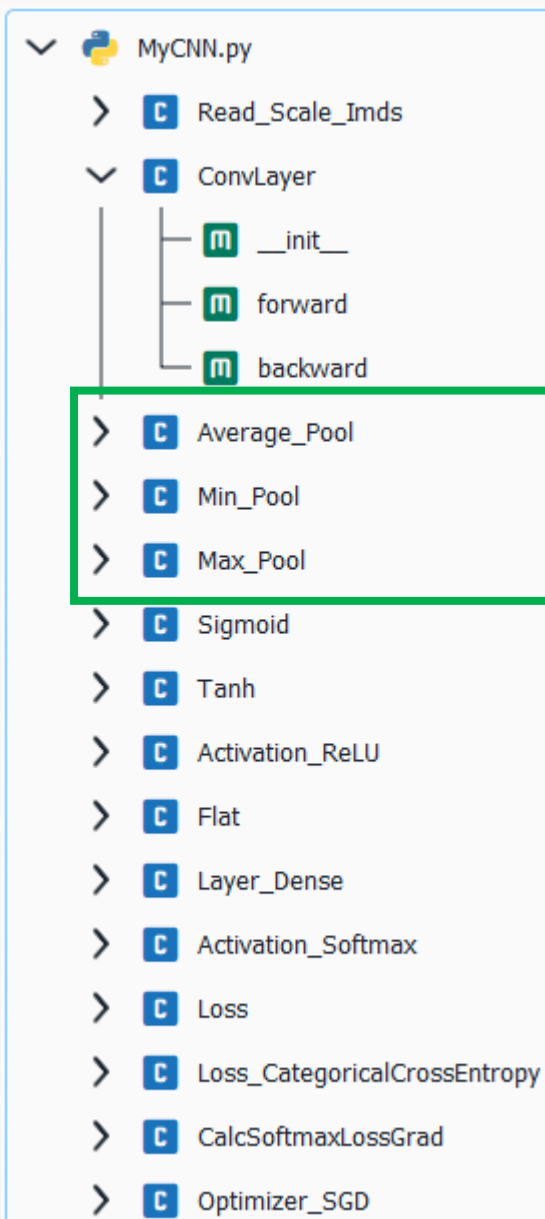
```
plt.imshow(Conv2.output[:, :, 0, 4], cmap = 'gray')
plt.show()
```

```
plt.imshow(Conv3.output[:, :, 0, 4], cmap = 'gray')
plt.show()
```

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



see MyCNN.py

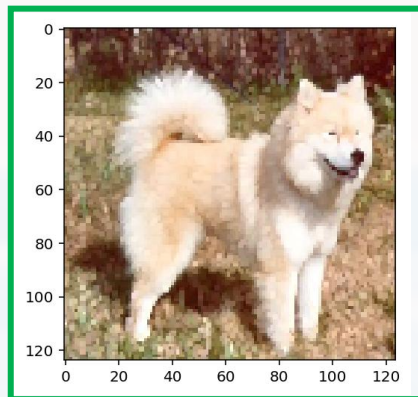


```
·y_start·:=·y*stride  
·y_end···:=·y_start+·yK  
·x_start·:=·x*stride  
·x_end···:=·x_start+·xK  
·  
·sx·:=·slice(x_start, x_end)  
·sy·:=·slice(y_start, y_end)  
·  
·current slice·····:=·currentIm pad[sx,sy,c]  
·slice_max·····:=·float(current_slice.max())  
·output[x, y, c, i]·:=·slice_max
```

LeNet numpy only

LeNet TensorFlow
sequences as images
segmentation

here: max pool

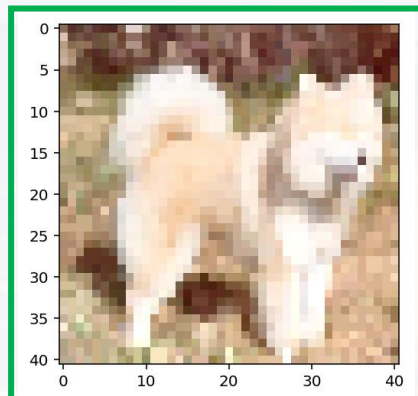


```
MP1 = Max_Pool()  
MP2 = Max_Pool()  
MP3 = Max_Pool()
```

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

initializing ax pool layer

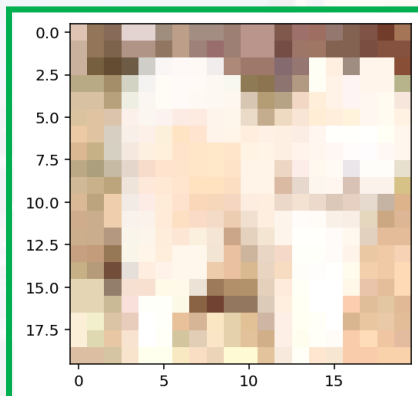
```
MP1.forward(I, 2, 3)  
MP2.forward(MP1.output, 3, 3)  
MP3.forward(MP2.output, 2, 3)
```



```
plt.imshow(MP1.output[:, :, :, 4].astype(int))  
plt.show()
```

```
plt.imshow(MP2.output[:, :, :, 4].astype(int))  
plt.show()
```

```
plt.imshow(MP3.output[:, :, :, 4].astype(int))  
plt.show()
```

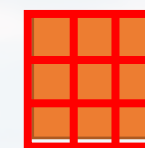
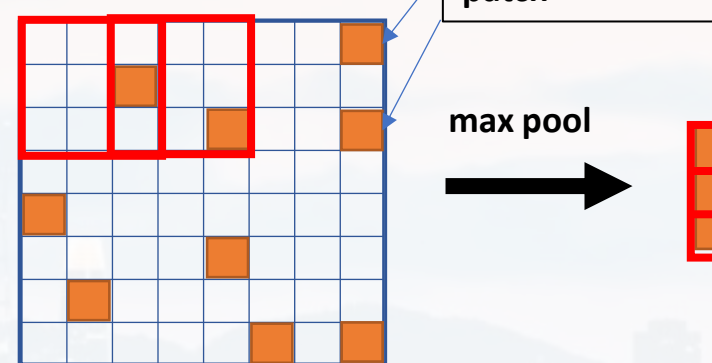
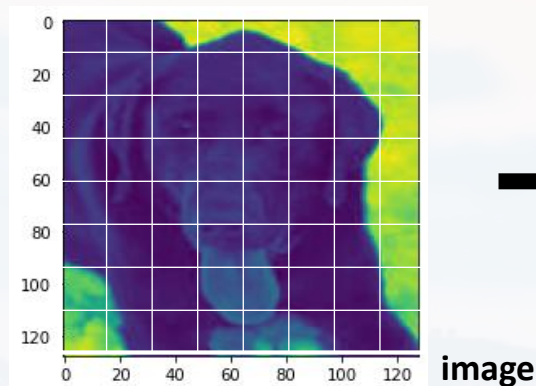


passing image through max
pool layer (stride length,
kernel size)



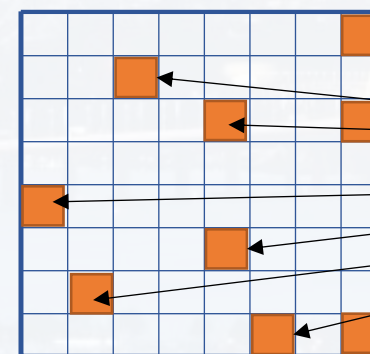
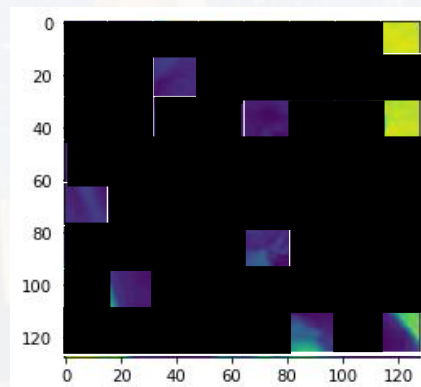
for **backpropagation**, we need to **up-sample** the images!

forward:

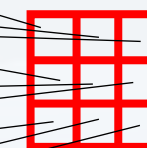


max

backward:



?



dvalues

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

We need **to track, where the max came from** for each patch: creating a **mask** in the forward part
For those pixel: $dinputs = dvalues$



for **backpropagation**, we need to **up-sample** the images!

Input I stride = 3; xK = yK = 5

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	538	538	538	538	541	541
2	0	538	538	538	538	541	541
3	0	535	535	538	538	541	541
4	0	535	535	538	538	541	541
5	0	535	535	538	538	538	538
6	0	535	535	538	538	538	538
7	0	535	535	538	538	541	541
8	0	535	535	538	538	541	541
9	0	535	535	538	538	538	544

max pool

output

	0	1	2	3	4	5	6
0	538	544	545	551	543	357	364
1	538	544	547	550	544	369	361
2	538	544	548	554	554	369	361
3	544	547	548	554	554	361	358
4	544	544	499	364	362	368	375
5	424	363	355	355	350	363	375
6	352	354	358	358	356	356	357
7	352	353	354	359	368	368	350
8	353	353	362	362	360	359	356
9	357	356	367	367	366	365	356

mask

and so on...

	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	538	538	538	538	0	0
2	0	538	538	538	538	0	0
3	0	0	0	538	538	0	0
4	0	0	0	538	538	0	0
5	0	0	0	538	538	0	0
6	0	0	0	538	538	0	0
7	0	0	0	538	538	0	0
8	0	0	0	538	538	0	0
9	0	0	0	538	538	0	544

LeNet numpy only

LeNet TensorFlow

sequences as images

segmentation



for **backpropagation**, we need to **up-sample** the images!

dinputs and mask have to look the same (but different values)!

```
Conv1.forward(I,2,1)
MP1.forward(Conv1.output,3,5)
MP1.backward(MP1.output)
```

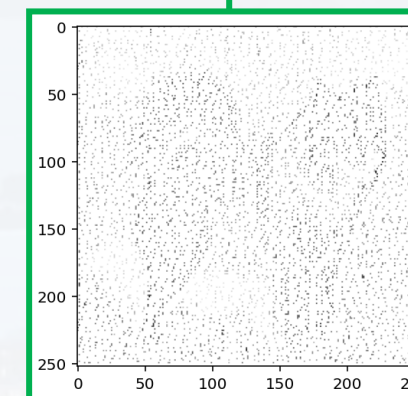
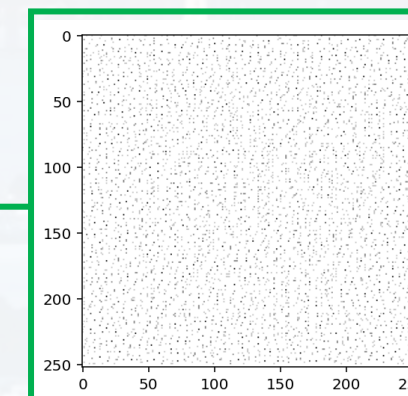
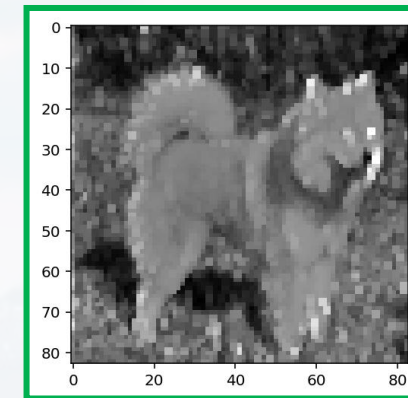
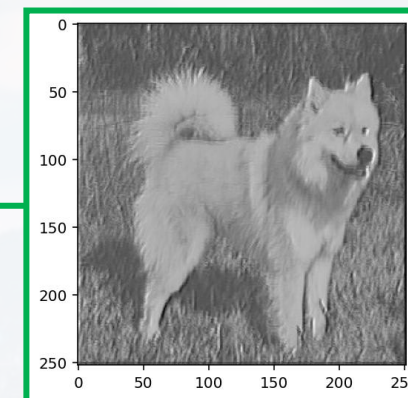
```
plt.imshow(Conv1.output[:, :, 0, 4], cmap = 'gray')
plt.show()
```

```
plt.imshow(MP1.output[:, :, 0, 4], cmap = 'gray')
plt.show()
```

```
plt.imshow(MP1.dinputs[:, :, 0, 4], cmap = 'gray_r')
plt.show()
```

```
plt.imshow(MP1.mask[:, :, 0, 4], cmap = 'gray_r')
plt.show()
```

```
Diff = MP1.mask[:, :, 0, 4]/np.max(MP1.mask[:, :, 0, 4]) - \
      MP1.dinputs[:, :, 0, 4]/np.max(MP1.dinputs[:, :, 0, 4])
```



LeNet numpy only

LeNet TensorFlow
sequences as images
segmentation



for **backpropagation**, we need to **up-sample** the images!

dinputs and mask have to look the same (but different values)!

LeNet numpy only

LeNet TensorFlow
sequences as images
segmentation

```
Conv1.forward(I,2,1)
MP1.forward(Conv1.output,3,5)
MP1.backward(MP1.output)
```

```
plt.imshow(Conv1.output[:, :, 0, 4], cmap = 'gray')
plt.show()
```

```
plt.imshow(MP1.output[:, :, 0, 4], cmap = 'gray')
plt.show()
```

```
plt.imshow(MP1.dinputs[:, :, 0, 4], cmap = 'gray_r')
plt.show()
```

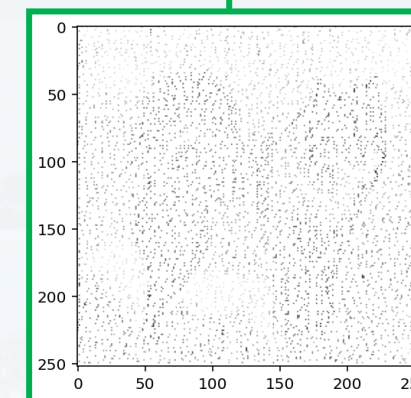
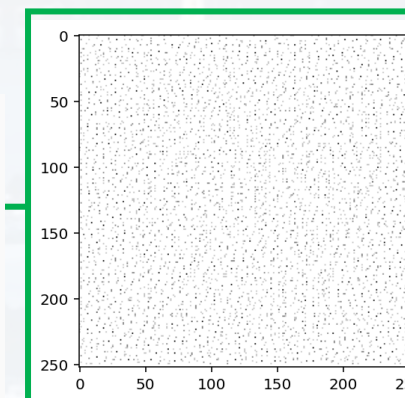
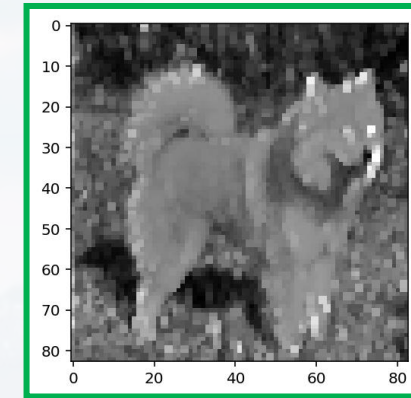
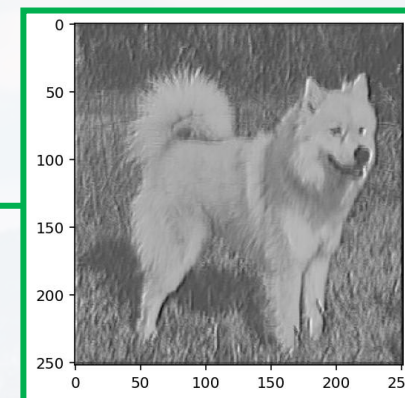
```
plt.imshow(MP1.mask[:, :, 0, 4], cmap = 'gray_r')
plt.show()
```

```
Diff = MP1.mask[:, :, 0, 4] /
      MP1.dinputs[:, :, 0,
```

In [20]: Diff

Out[20]:

```
array([[ 0.,  0.,  0., ...,  0.,  0., -0.],
       [ 0.,  0.,  0., ...,  0.,  0., -0.],
       [ 0.,  0.,  0., ...,  0.,  0., -0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  0., -0.],
       [ 0.,  0.,  0., ...,  0.,  0., -0.],
       [ 0.,  0.,  0., ...,  0., -0., -0.]])
```





for **backpropagation**, we need to **up-sample** the images!

same for convolution layer now:

backward **dinputs** = **dvalues** * **weights** (we don't need for first conv layer!)

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

forward:

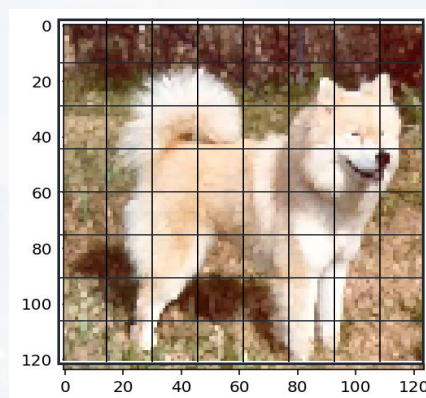
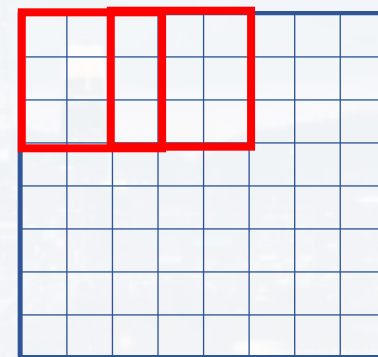
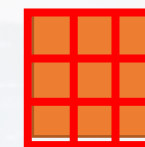


image = inputs

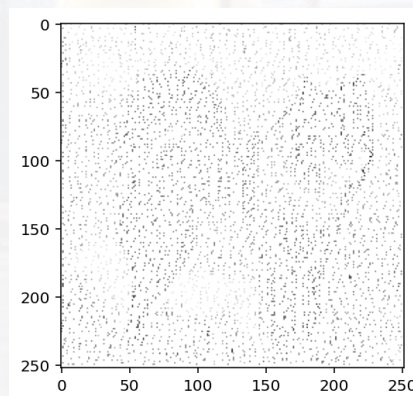


$$\sum_i I_i w_i + b$$



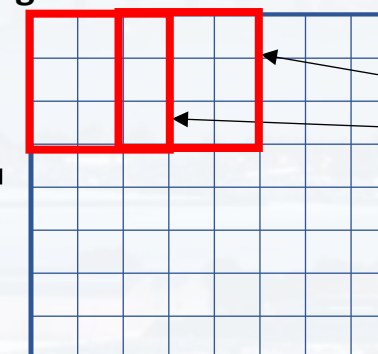
convolved image

backward:



dinputs

weights



dvalues



for **backpropagation**, we need to **up-sample** the images!

same for convolution layer now:

backward $dinputs = dvalues * weights$ (we don't need for first conv layer!)

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

forward:

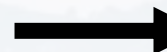
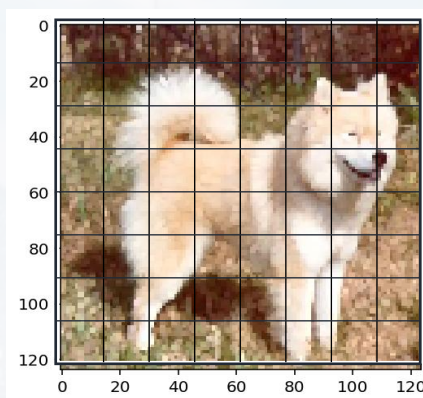
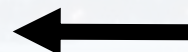
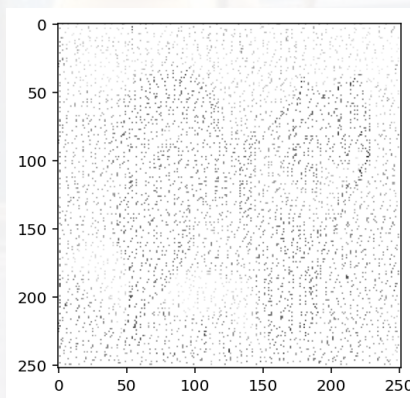


image = input

```
for i in range(numImds): # loop over number of images
    ... currentIm_pad = imagePadded[:, :, :, i] # Select ith padded image
    ... for k in range(NK): # loop over kernels (= #filters)
        ... for c in range(numChan): # loop over channels of incoming data
            ...
            if filt[c,k] != 1:
                ...
                for y in range(yd): # loop over axis of output
                    for x in range(xd): # loop over axis of output
                        ...
                        # finding corners of the current "slice" (~4 lines)
                        y_start = y*stride
                        y_end = y_start + yK
                        x_start = x*stride
                        x_end = x_start + xK
                        ...
                        sx = slice(x_start, x_end)
                        sy = slice(y_start, y_end)
                        ...
                        current_slice = currentIm_pad[sx, sy, c]
                        ...
                        dweights[:, :, k] += current_slice * dvalues[x, y, k, i]
                        dinputs[sx, sy, c, i] += weights[:, :, k] * dvalues[x, y, k, i]
                        ...
                        dbiases[0, k] += np.sum(np.sum(dvalues[:, :, k, i], axis=0), axis=0)
```



backward:





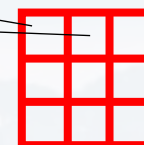
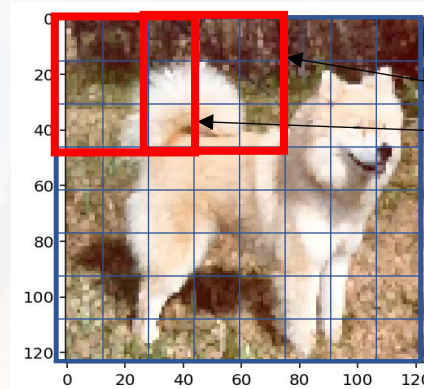
for **backpropagation**, we need to **up-sample** the images!

same for convolution layer now:

backward

$dweights = dvalues * inputs$

backward:



dvalues

```
.....
.....# finding corners of the current "slice" (~4 lines)
.....y_start = y*stride
.....y_end = y_start + yK
.....x_start = x*stride
.....x_end = x_start + xK
.....
.....sx = slice(x_start,x_end)
.....sy = slice(y_start,y_end)
.....
.....current_slice = currentIm_pad[sx,sy,c]
```

```
dweights[:, :, k] += current_slice * dvalues[x, y, k, i]
```

```
.....dinputs[sx,sy,c,i] += weights[:, :, k] * dvalues[x, y, k, i]
.....
.....
.....dbiases[0, k] += np.sum(np.sum(dvalues[:, :, k, i], axis=0), axis=0)
```

LeNet numpy only

LeNet TensorFlow

sequences as images

segmentation



for **backpropagation**, we need to **up-sample** the images!

same for convolution layer now:

backward

dbiases = dvalues

backward:

```
.....
.....# finding corners of the current "slice" (~4 lines)
.....y_start = y*stride
.....y_end   = y_start + yK
.....x_start = x*stride
.....x_end   = x_start + xK
.....
.....sx      = slice(x_start,x_end)
.....sy      = slice(y_start,y_end)
.....
.....current_slice = currentIm_pad[sx,sy,c]
.....
.....dweights[:, :, k] += current_slice * dvalues[x,y,k,i]
.....dinputs[sx,sy,c,i] += weights[:, :, k] * dvalues[x,y,k,i]
.....
.....
.....dbiases[0,k] += np.sum(np.sum(dvalues[:, :, k,i], axis=0), axis=0)
```

```
dbiases[0,k] += np.sum(np.sum(dvalues[:, :, k,i], axis=0), axis=0)
```

LeNet numpy only

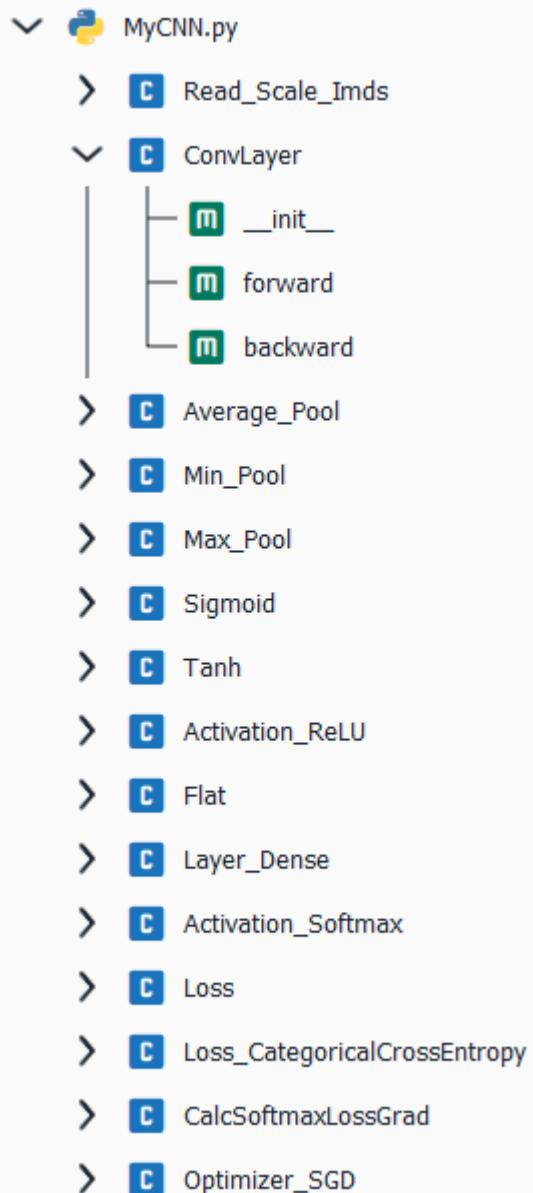
LeNet TensorFlow

sequences as images

segmentation



We are done now
and can explore
our self-made
LeNet

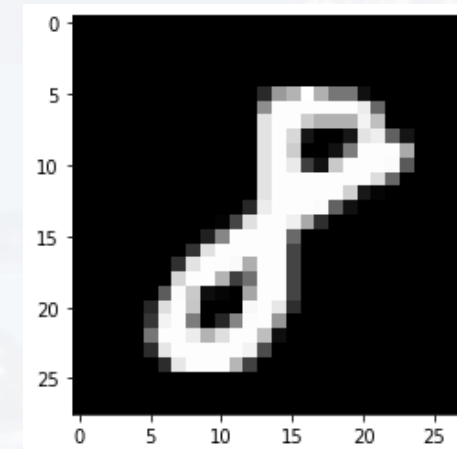
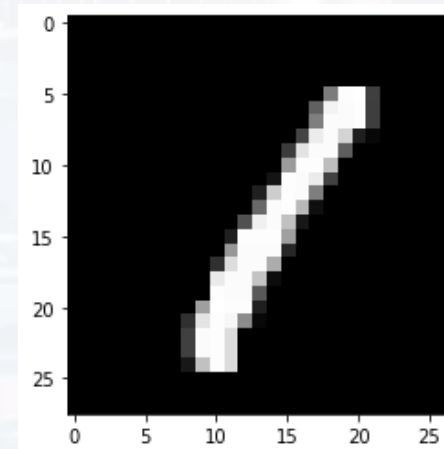
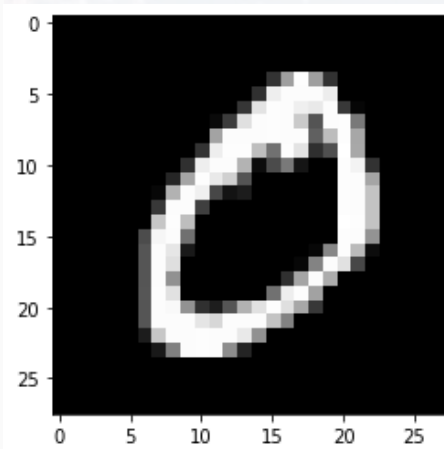


```
#pip install keras  
#pip install tensorflow
```

```
from keras.datasets import mnist
```

```
(train_x, train_y), (test_x, test_y) = mnist.load_data()
```

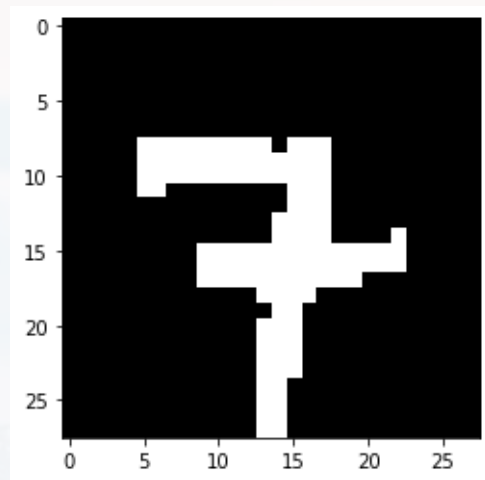
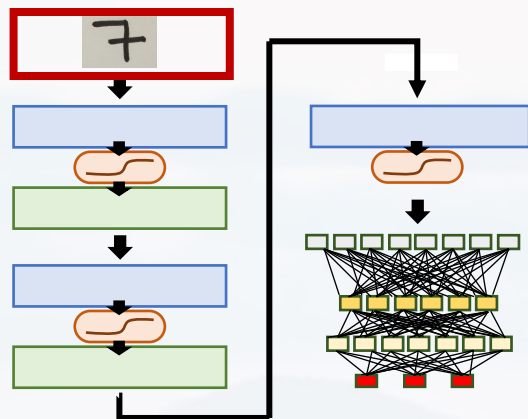
plt.imshow(train_x[1,:,:], cmap = 'gray')



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



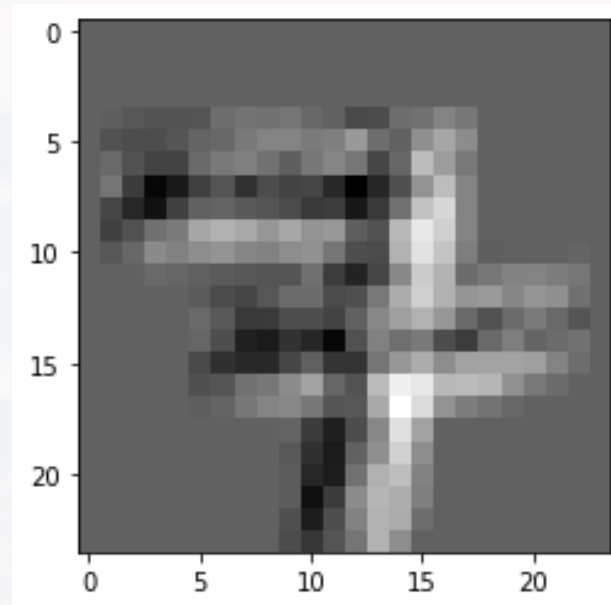
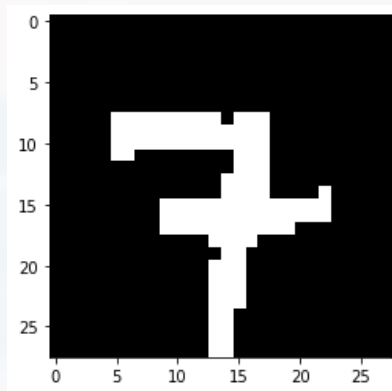
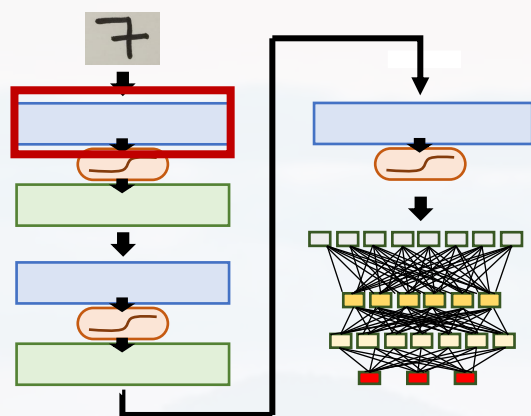
exploring self-made LeNet:



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



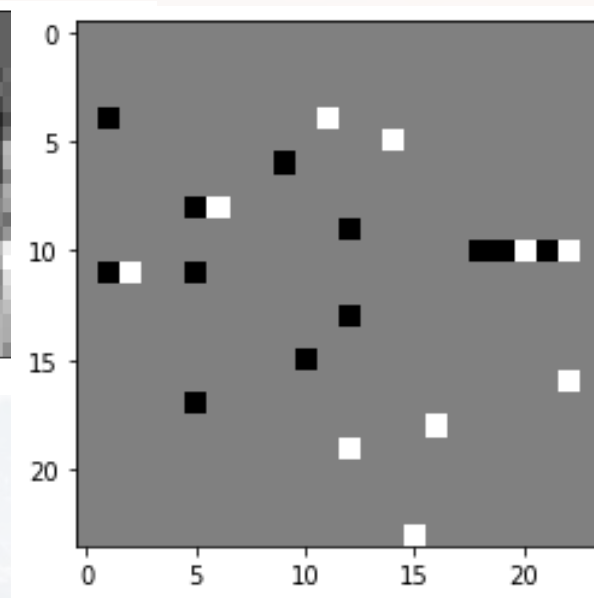
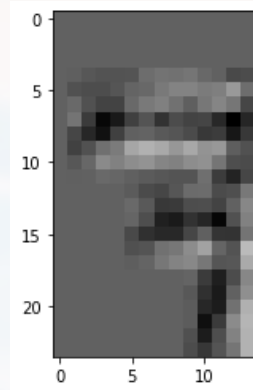
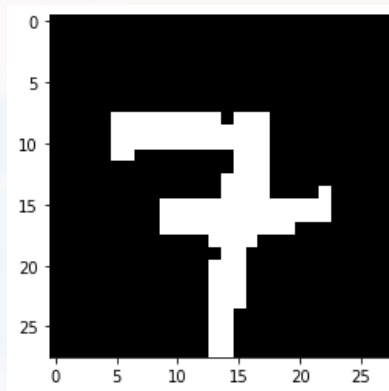
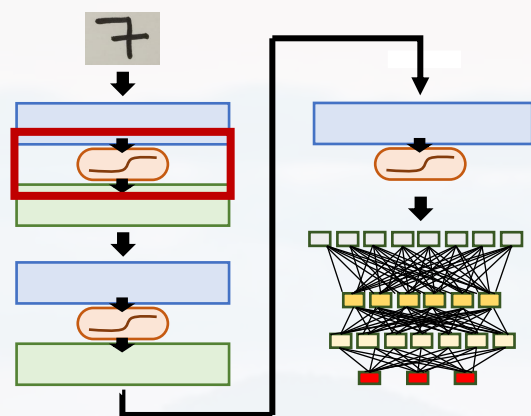
exploring self-made LeNet:



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



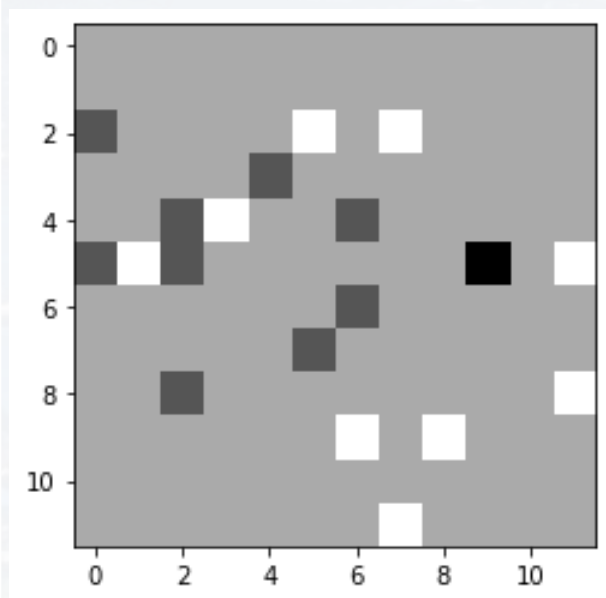
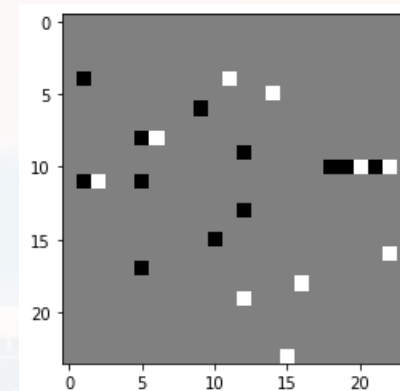
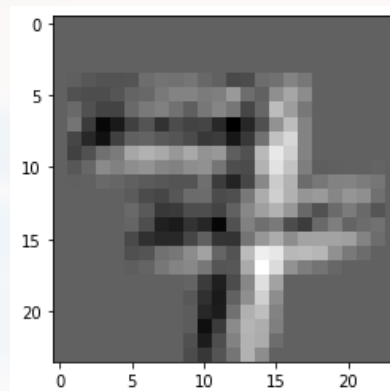
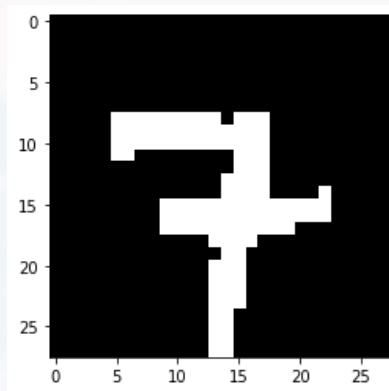
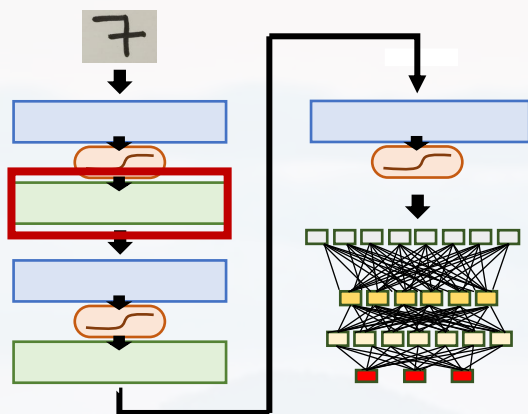
exploring self-made LeNet:



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



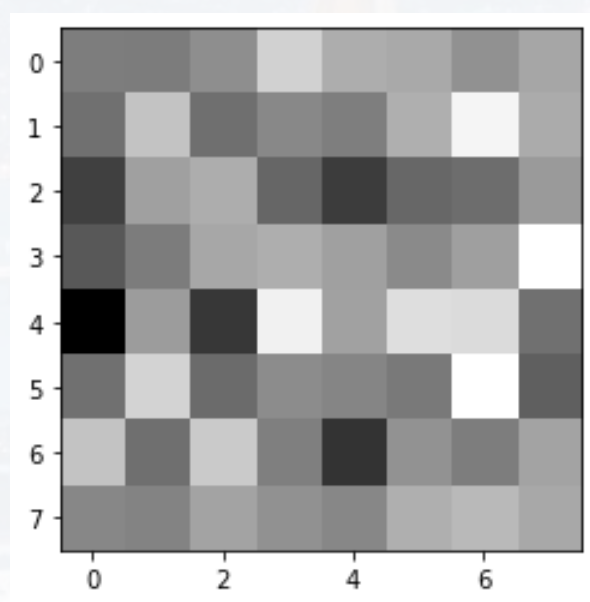
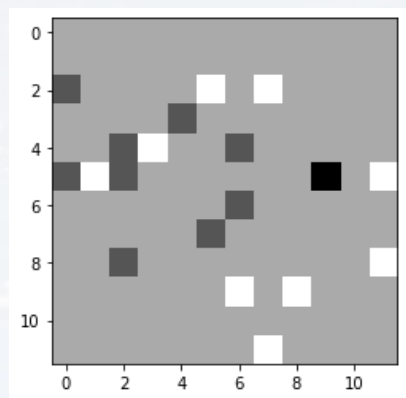
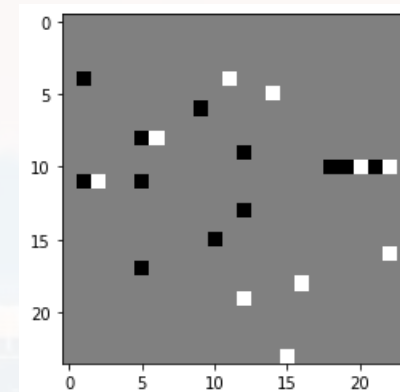
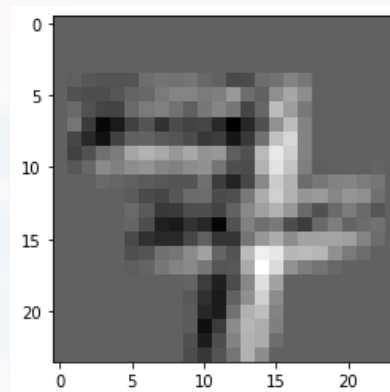
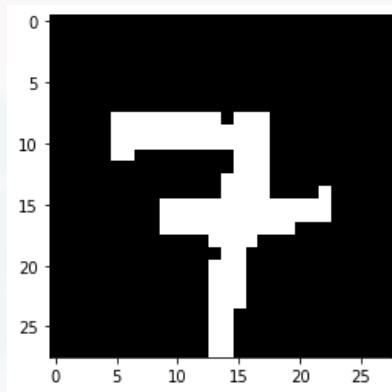
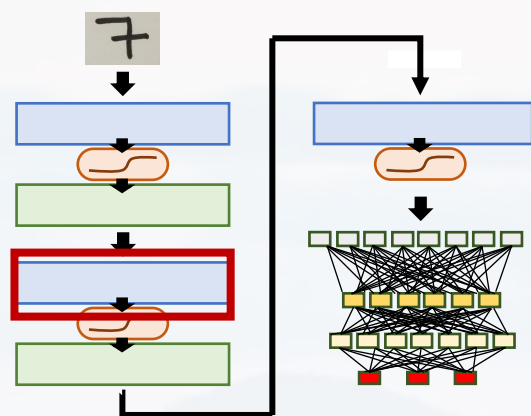
exploring self-made LeNet:



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



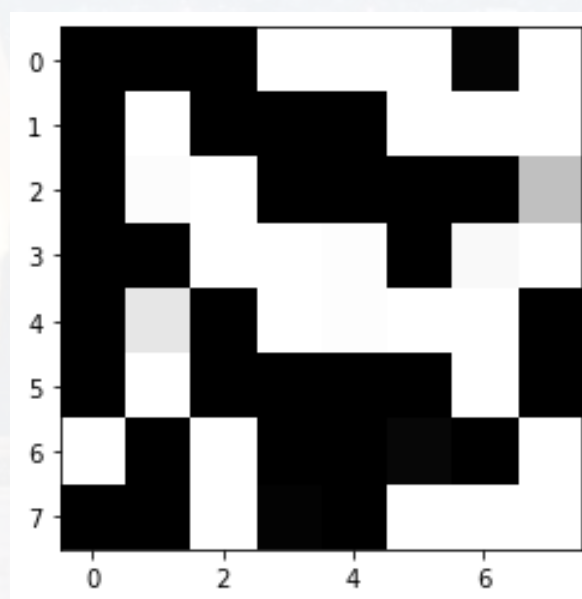
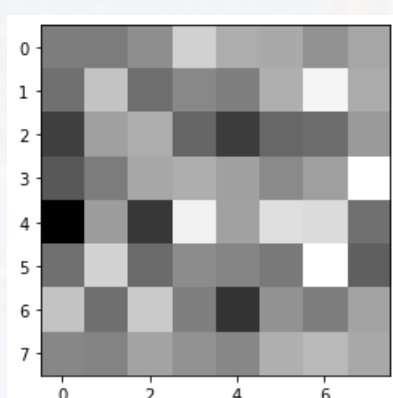
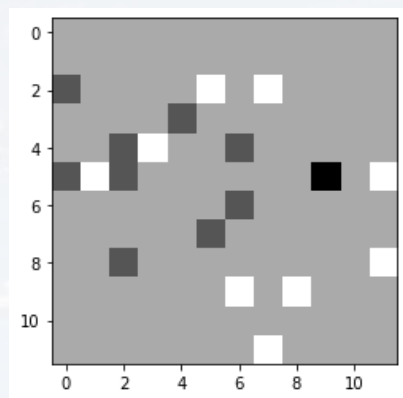
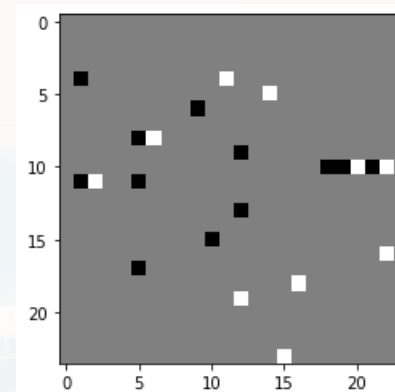
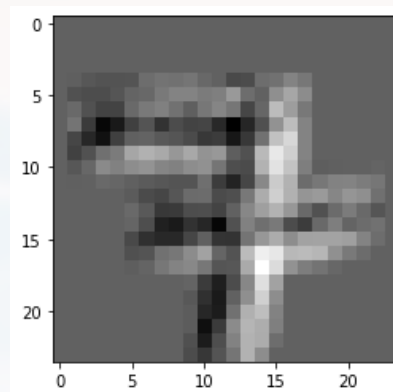
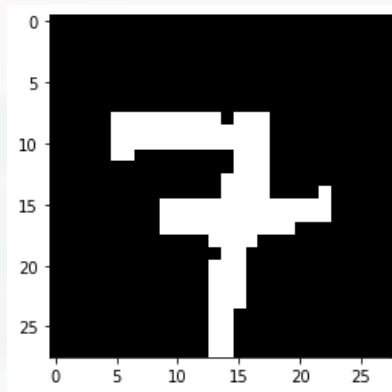
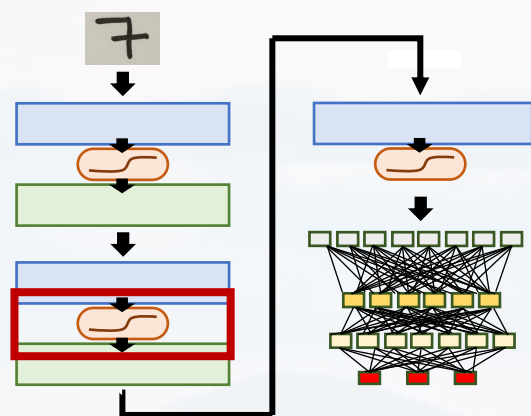
exploring self-made LeNet:



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



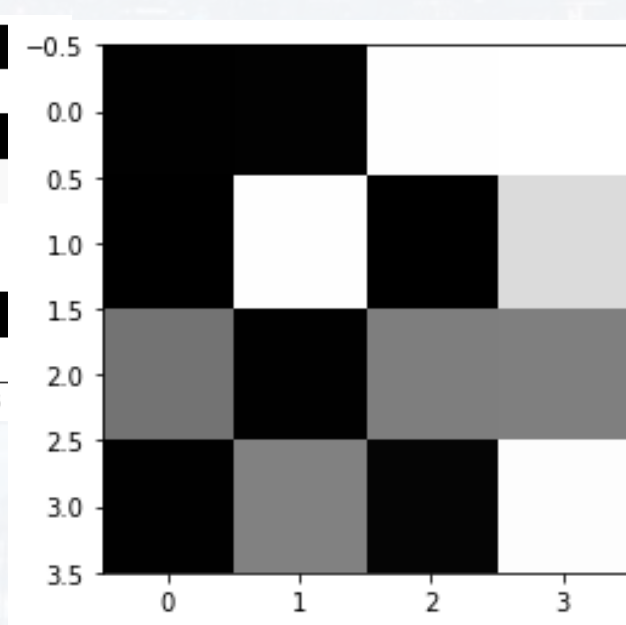
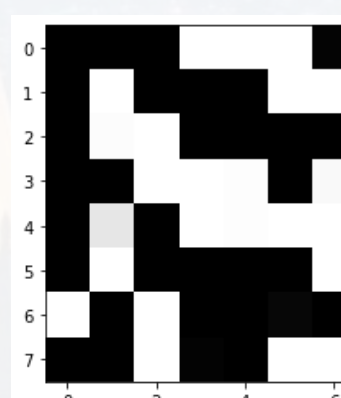
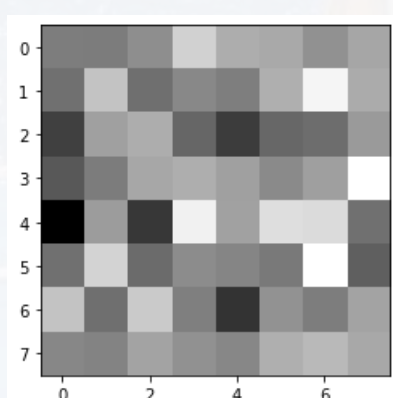
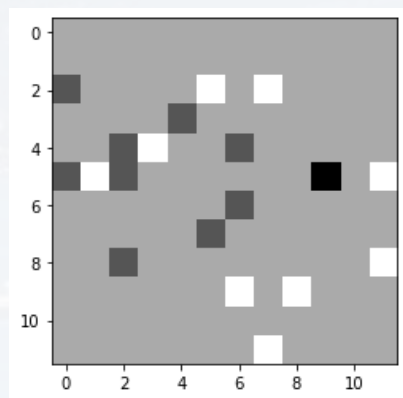
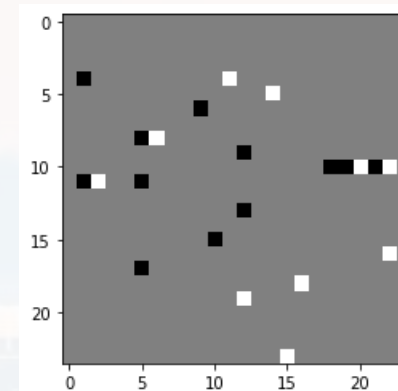
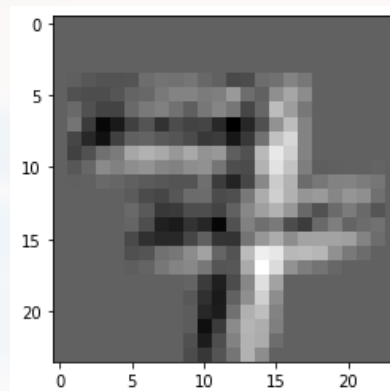
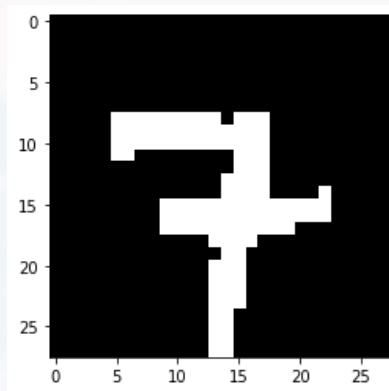
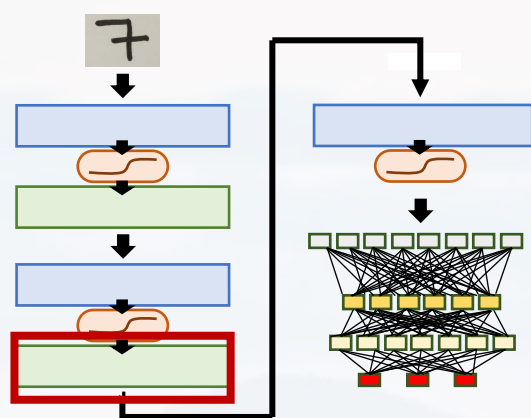
exploring self-made LeNet:



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



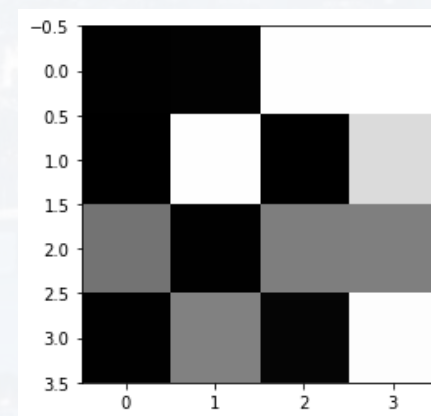
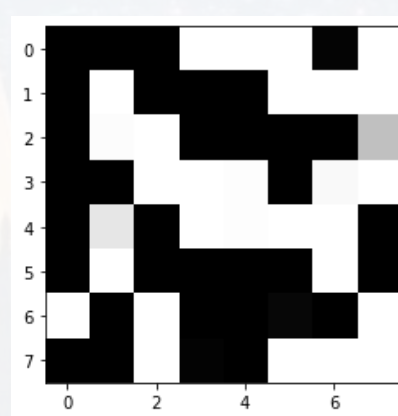
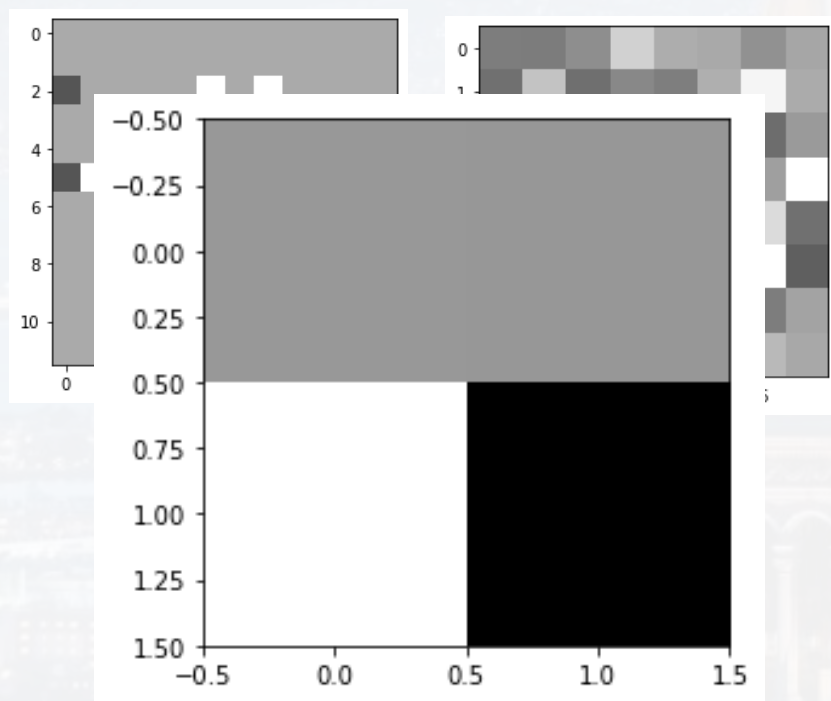
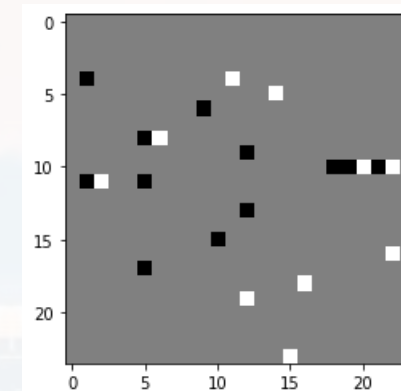
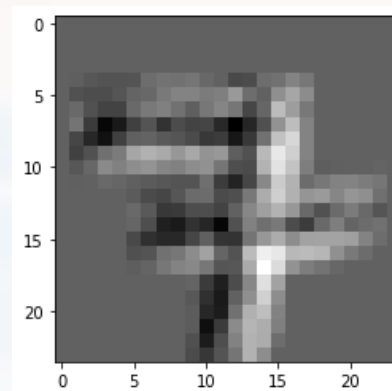
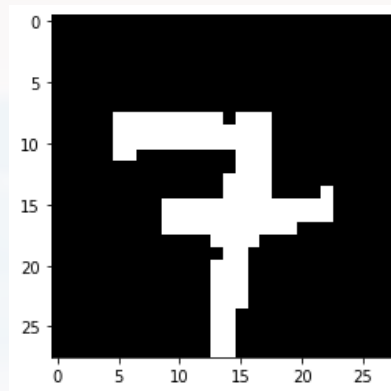
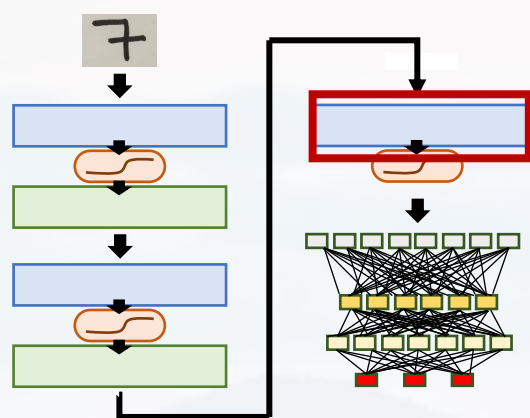
exploring self-made LeNet:



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



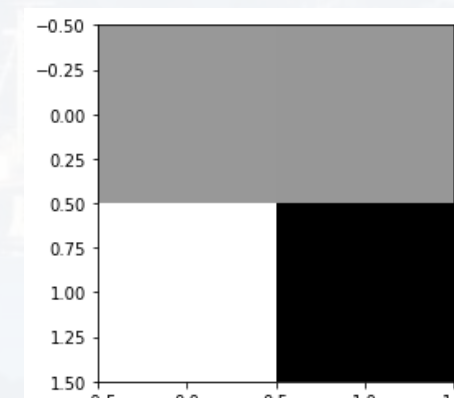
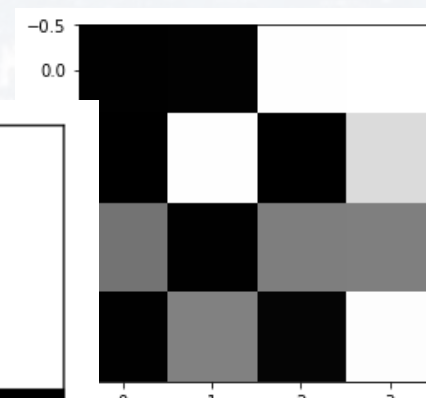
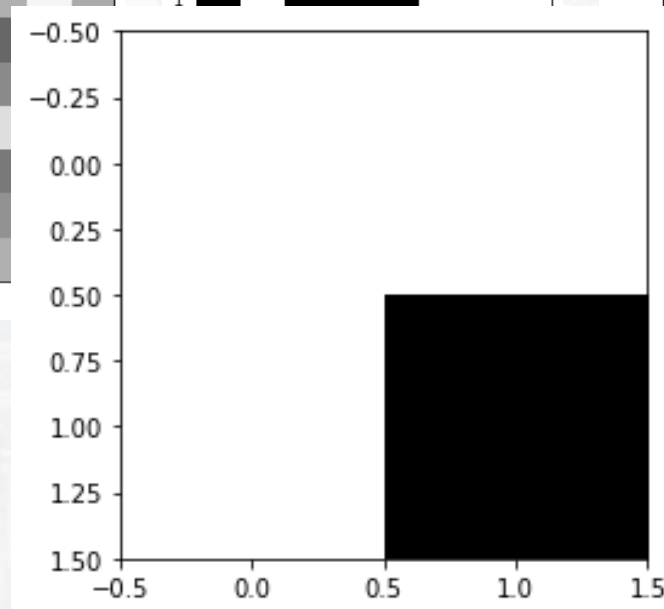
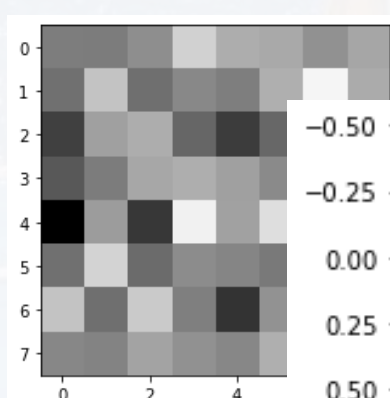
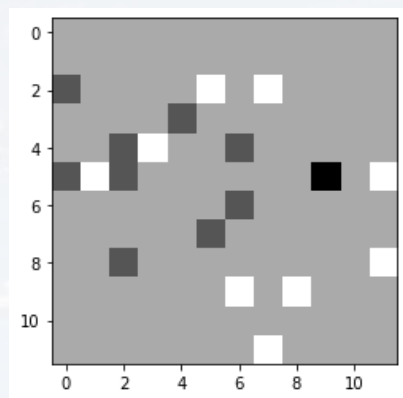
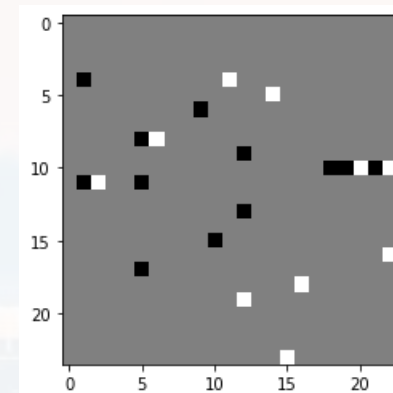
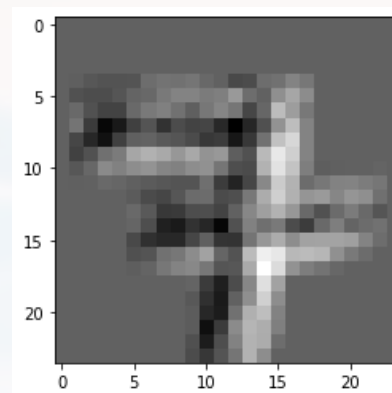
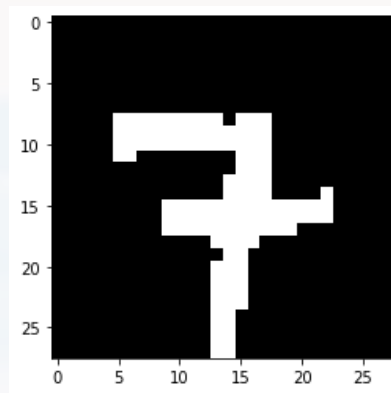
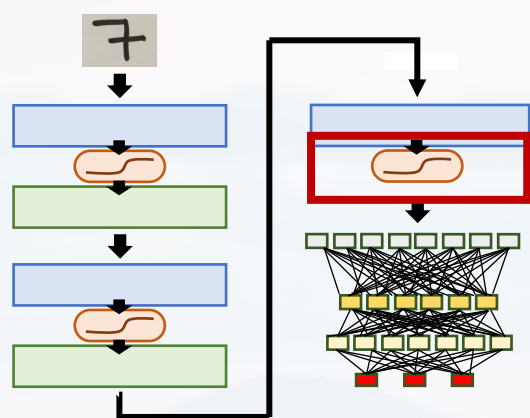
exploring self-made LeNet:



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



exploring self-made LeNet:



LeNet numpy only

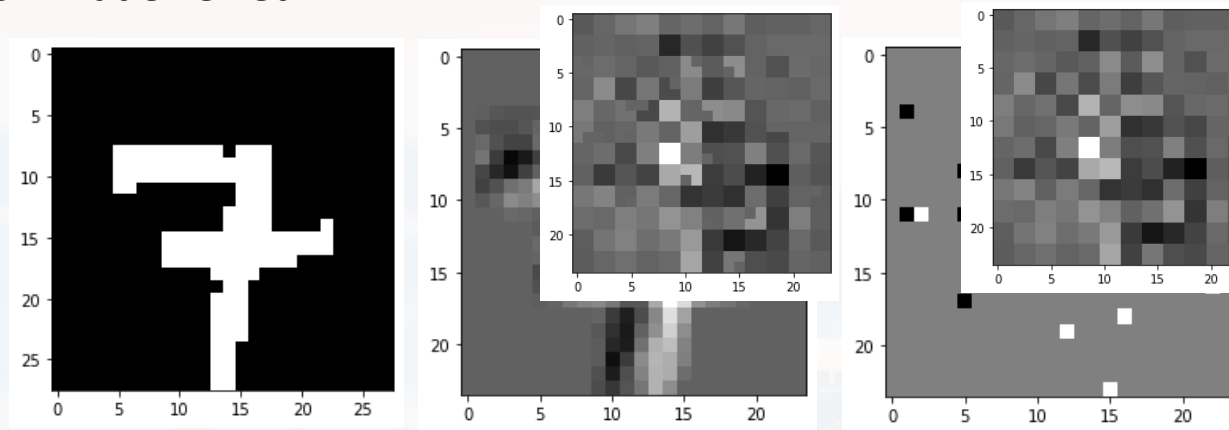
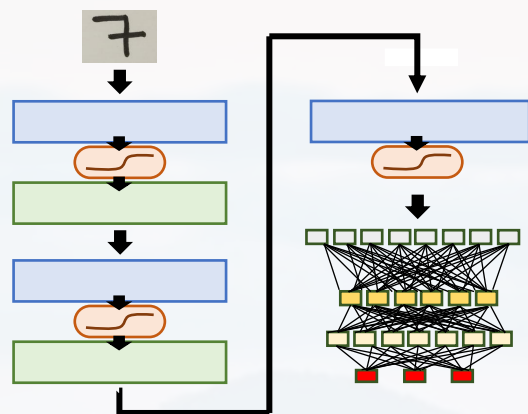
LeNet TensorFlow

sequences as images

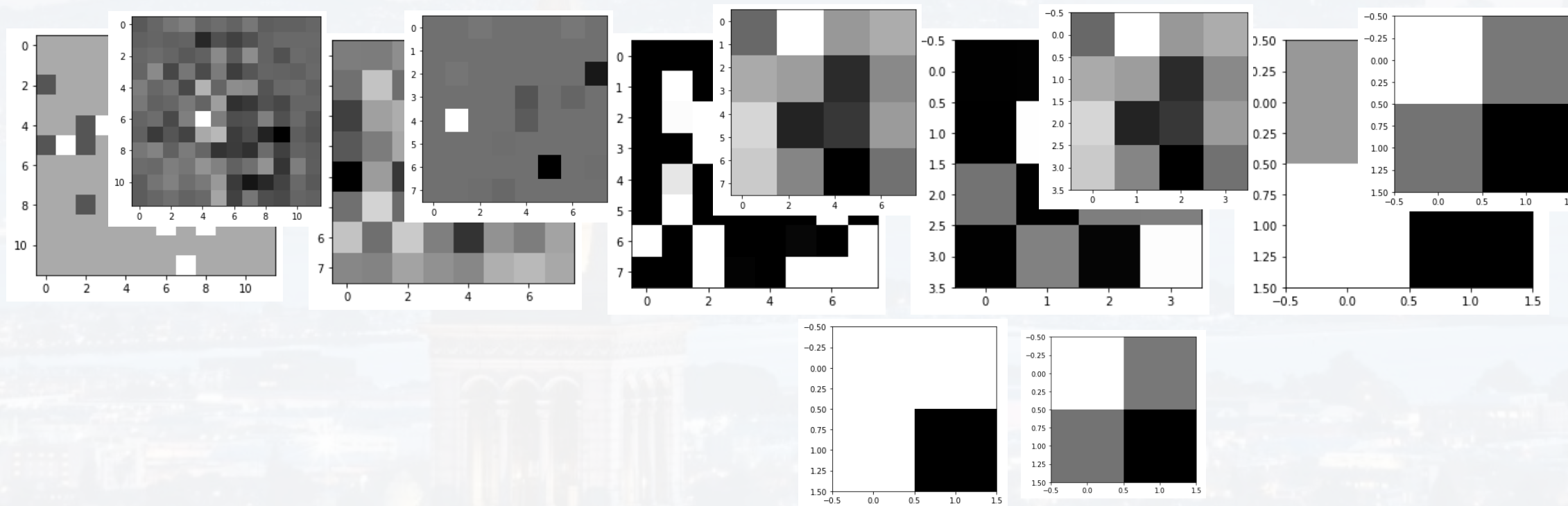
segmentation



exploring self-made LeNet:



now back propagation!



LeNet numpy only

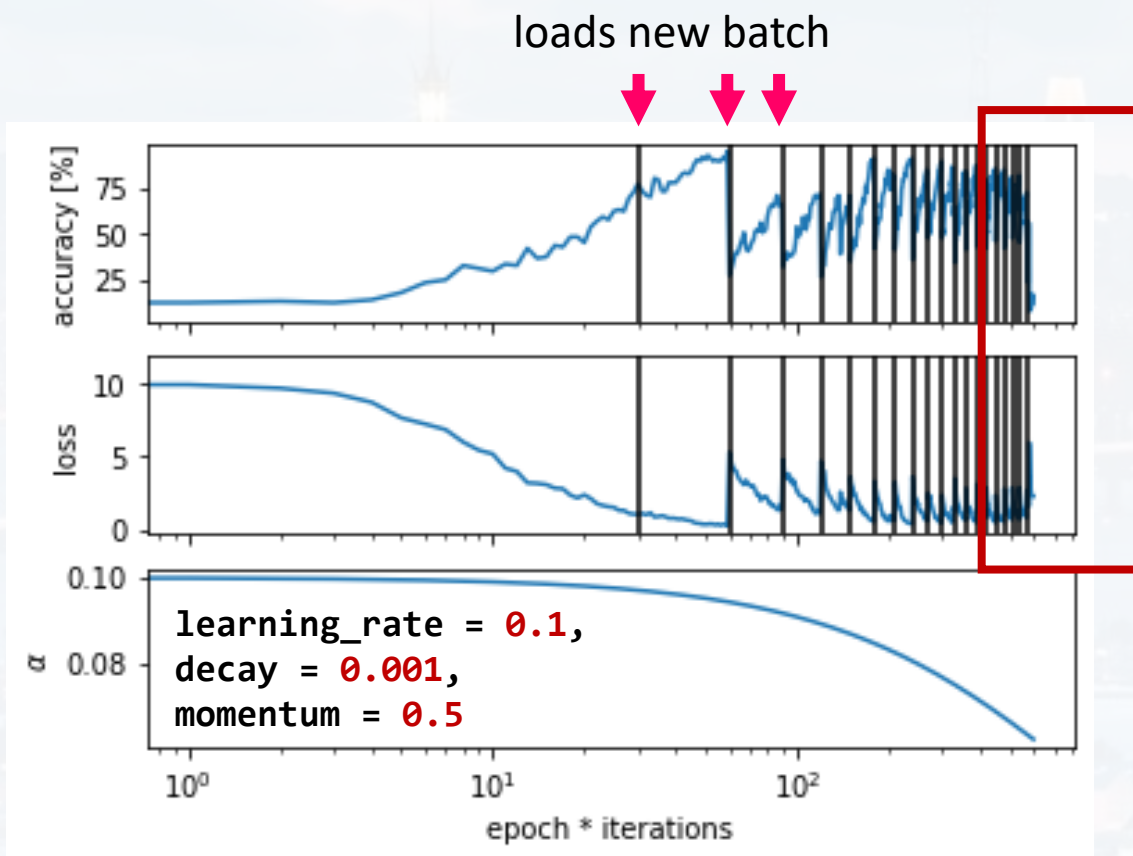
LeNet TensorFlow
sequences as images
segmentation



running self-made LeNet:

```
LeNet = MyLeNet()  
LeNet.RunTraining()
```

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



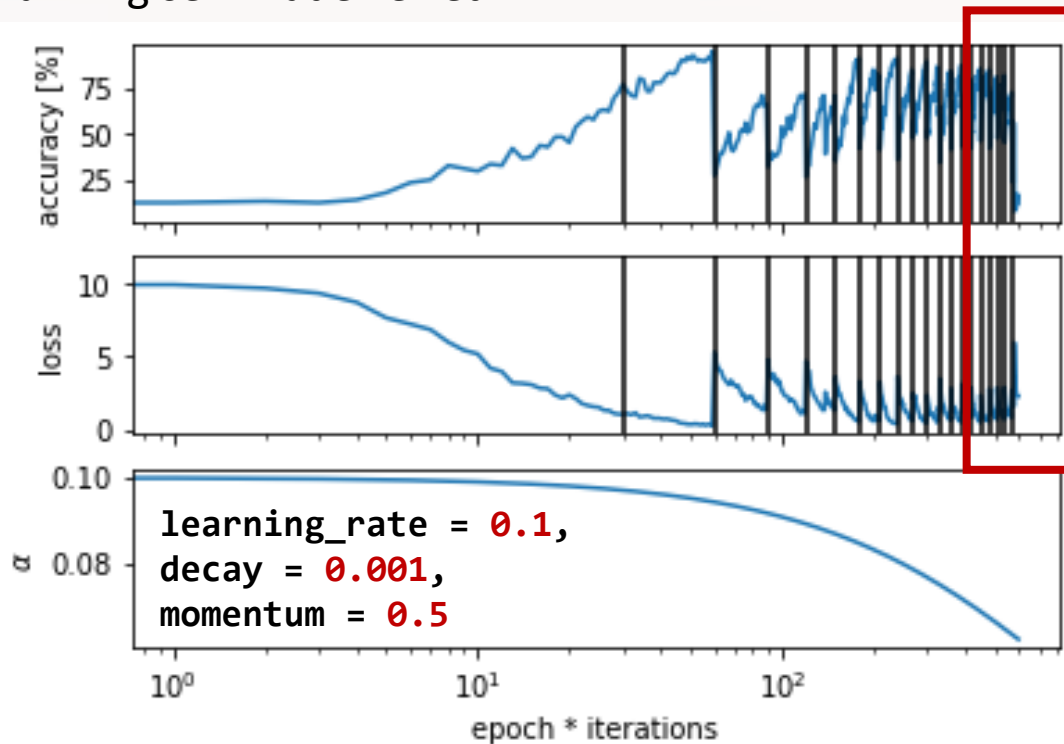
after several iterations:

- accuracy drops and loss gets up again
- gets stuck

Why???



running self-made LeNet:



after several iterations:

- accuracy drops and loss gets up again
- gets stuck

Why???

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

	0	1	2	3
0	-15923.3	-16647.4	-16277.1	-16993.1
1	9795.1	8468.91	9110.83	8205.6
2	37956.4	36572.6	37324.2	37008.4
3	39686.6	39131	39087.4	39614.6
4	25465.3	26270.1	25232.4	25836.7

```
W = np.load('weightsC1.npy')  
W[:, :, 0]
```

Weights are exploding!

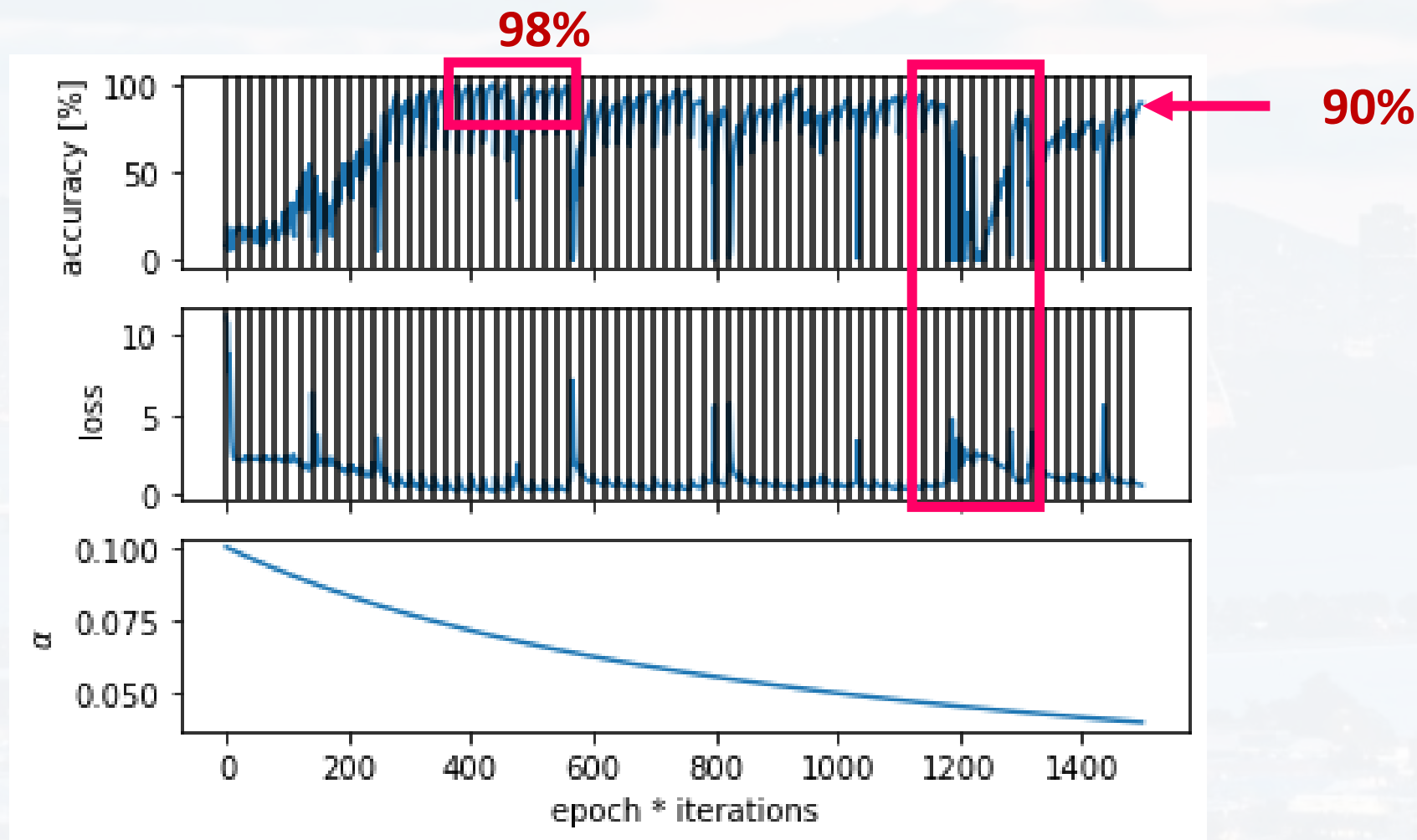
- penalizing large weights
- L2 regularization



running self-made LeNet:

run the function `MyLeNet` again, but this time with L2

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

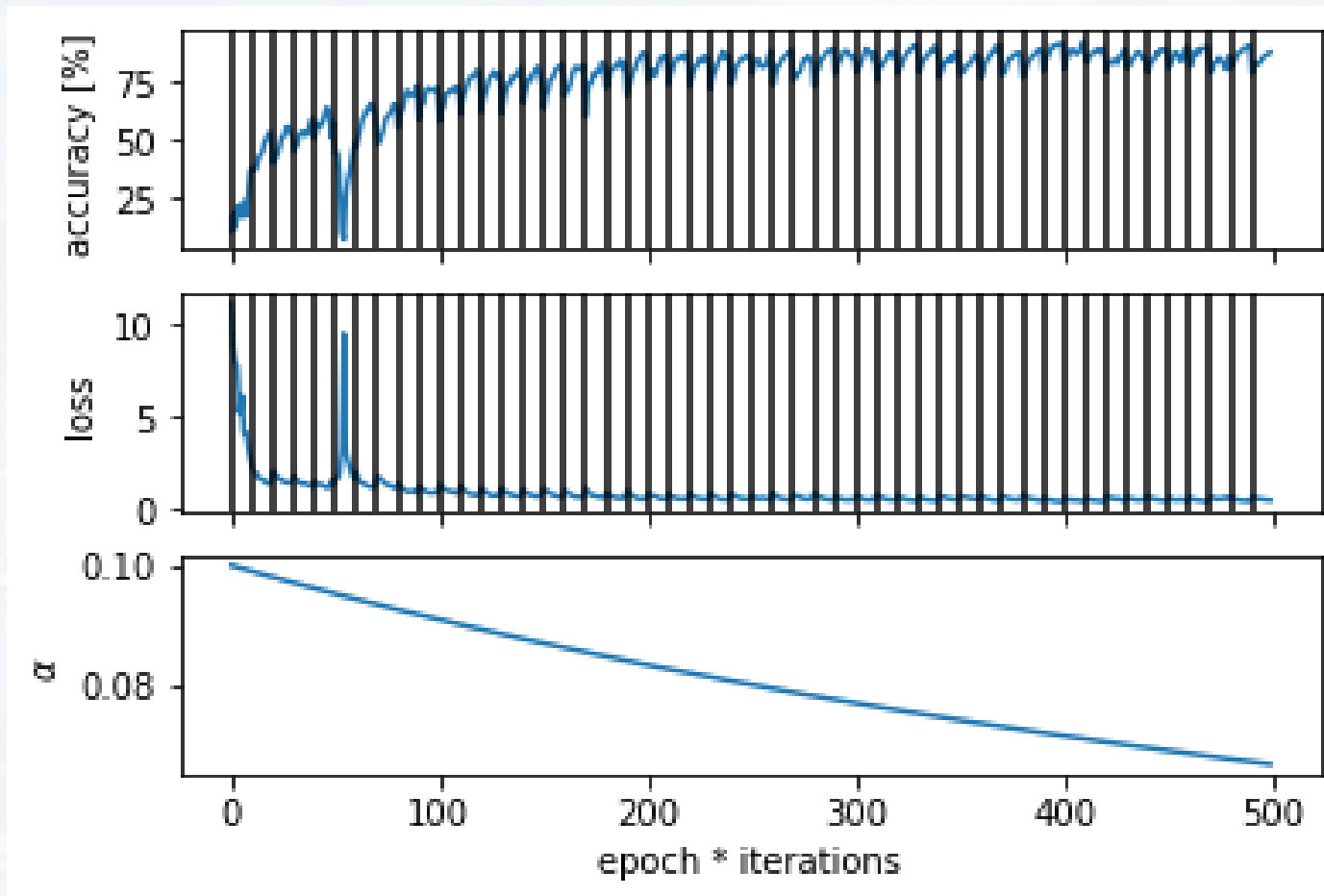




running self-made LeNet:

minibatch_size = **512**, iterations = **10**, epochs = **50**

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



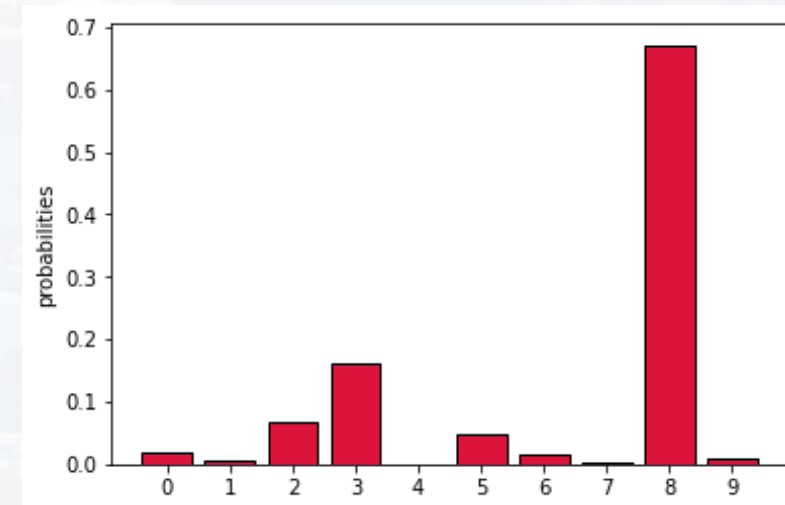
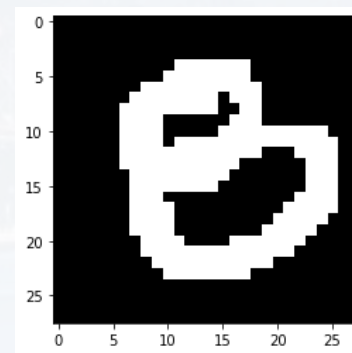


evaluating self-made LeNet:

EvaluateMyLeNet(N = 50)

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

predicted		actual	
	0		0
0	1	0	3
1	4	1	5
2	8	2	8
3	7	3	7
4	5	4	5
5	8	5	3
6	0	6	0
7	6	7	6
8	2	8	2



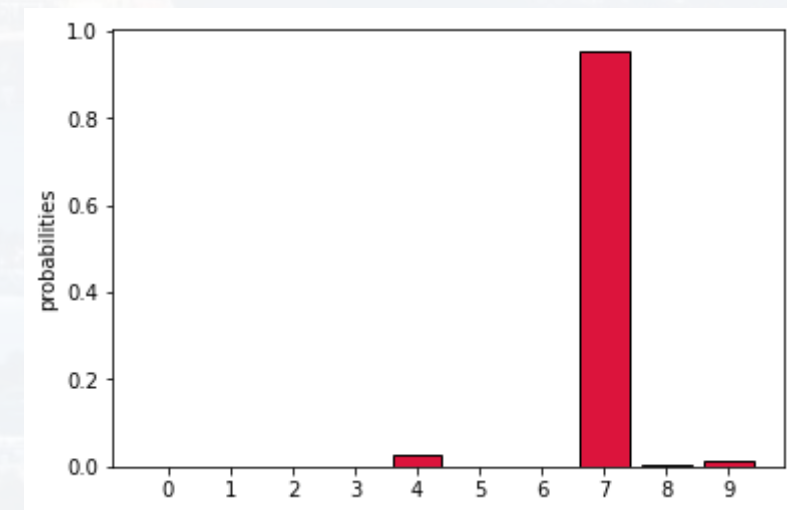
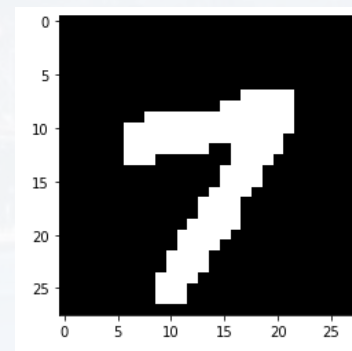


evaluating self-made LeNet:

EvaluateMyLeNet(N = 50)

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

predicted		actual	
	0		0
0	1	0	3
1	4	1	5
2	8	2	8
3	7	3	7
4	5	4	5
5	8	5	3
6	0	6	0
7	6	7	6
8	2	8	2





see LeNetTF.py and LeNetTF.ipynb

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.losses import categorical_crossentropy
from tensorflow.keras.layers import Dense, Flatten, Conv2D, AveragePooling2D
```

```
class MyLeNet(Sequential):
```

```
    def __init__(self, input_shape, num_classes):
        super().__init__()
```

```
#building LeNet #####
```

```
    #Note padding: string, either "valid" or "same" (case-insensitive). "valid" means no padding. "same"
```

```
    #more info: https://keras.io/api/layers/convolution\_layers/convolution2d/
```

```
    self.add(Conv2D(6, kernel_size = (5, 5), strides = (1, 1), activation = 'tanh', input_shape = input_shape, padding = 'same'))
    self.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
    self.add(Conv2D(16, kernel_size = (5, 5), strides = (1, 1), activation = 'tanh', padding = 'valid'))
    self.add(AveragePooling2D(pool_size = (2, 2), strides = (2, 2), padding = 'valid'))
    self.add(Conv2D(120, kernel_size = (5, 5), strides = (3, 3), activation = 'tanh', padding = 'valid'))
    self.add(Flatten())
    self.add(Dense(84, activation = 'tanh'))
    self.add(Dense(num_classes, activation = 'softmax'))
    #####
```

```
#building the optimizer #####
```

```
    lr_schedule = tf.keras.optimizers.schedules.ExponentialDecay(initial_learning_rate = 1e-2, decay_steps = 10000, decay_rate = 0.98)
    opt = tf.keras.optimizers.SGD(learning_rate = lr_schedule, momentum = 0.9)
```

```
    self.compile(optimizer = opt, loss = categorical_crossentropy, metrics = ['accuracy'])
```



see LeNetTF.py and LeNetTF.ipynb

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

Model: "my_le_net_1"

Layer (type)	Output Shape	Param #
conv2d_3 (Conv2D)	(None, 28, 28, 6)	156
average_pooling2d_2 (AveragePooling2D)	(None, 14, 14, 6)	0
conv2d_4 (Conv2D)	(None, 10, 10, 16)	2416
average_pooling2d_3 (AveragePooling2D)	(None, 5, 5, 16)	0
conv2d_5 (Conv2D)	(None, 1, 1, 120)	48120
flatten_1 (Flatten)	(None, 120)	0
dense_2 (Dense)	(None, 84)	10164
dense_3 (Dense)	(None, 10)	850

input = (N images, 28 x 28 pixel, 3 colors)
output = (N images, 28 x 28 pixel, 6 Conv filter)

6 Conv filter * shape (5, 5) = 150
plus 1 bias for each filter → total 156

Each image is represented by a vector of length 120

output layer: one neuron for each class

Total params: 61706 (241.04 KB)
Trainable params: 61706 (241.04 KB)
Non-trainable params: 0 (0.00 Byte)



see LeNetTF.py and LeNetTF.ipynb

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

```
running model...
Epoch 1/20
94/94 [=====] - 9s 92ms/step - loss: 1.1172 - accuracy: 0.7160 - val_loss: 0.4217 - val_accuracy: 0.8838
Epoch 2/20
94/94 [=====] - 8s 88ms/step - loss: 0.3577 - accuracy: 0.8990 - val_loss: 0.3096 - val_accuracy: 0.9109
Epoch 3/20
94/94 [=====] - 7s 73ms/step - loss: 0.2876 - accuracy: 0.9163 - val_loss: 0.2606 - val_accuracy: 0.9231
Epoch 4/20
94/94 [=====] - 8s 90ms/step - loss: 0.2444 - accuracy: 0.9287 - val_loss: 0.2238 - val_accuracy: 0.9333
Epoch 5/20
94/94 [=====] - 7s 69ms/step - loss: 0.2113 - accuracy: 0.9376 - val_loss: 0.1944 - val_accuracy: 0.9423
```

epoch: passing the entire dataset through the network

60,000 images / batch size = **512**

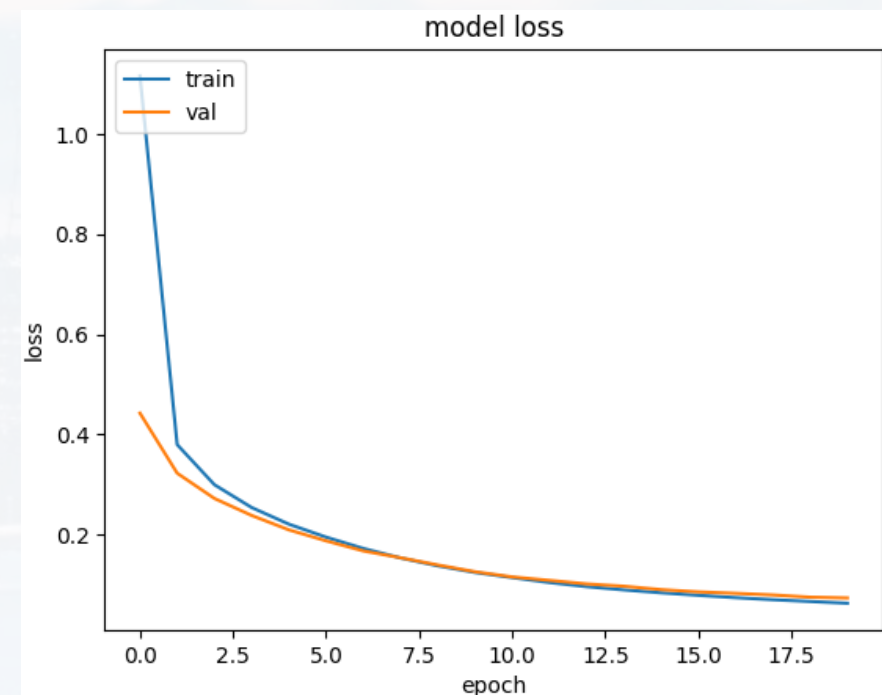
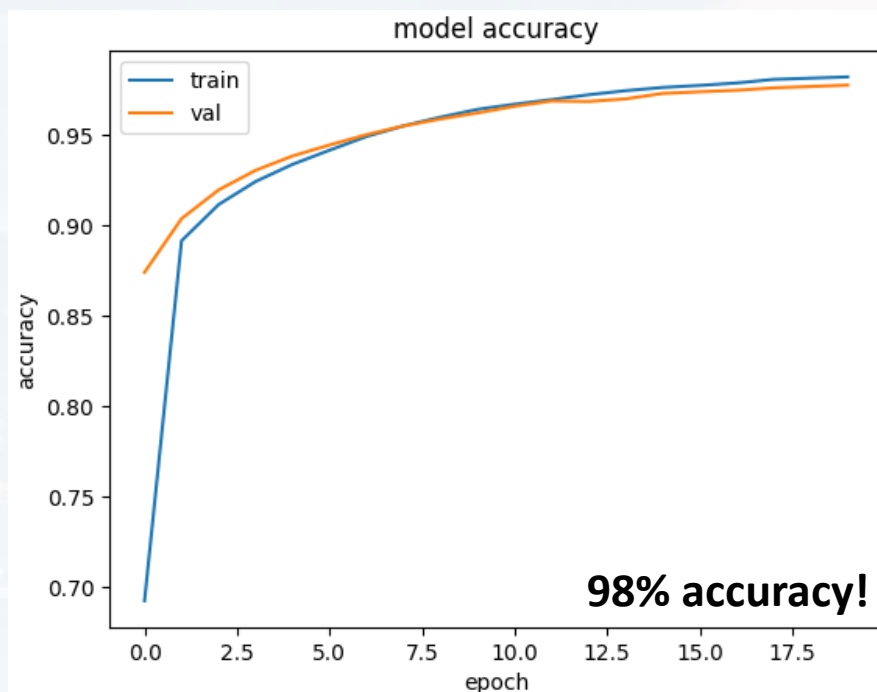
= **117** iterations per epoch

= **117 * 80%** for training = **94 iterations per epoch**



see `LeNetTF.py` and `LeNetTF.ipynb`

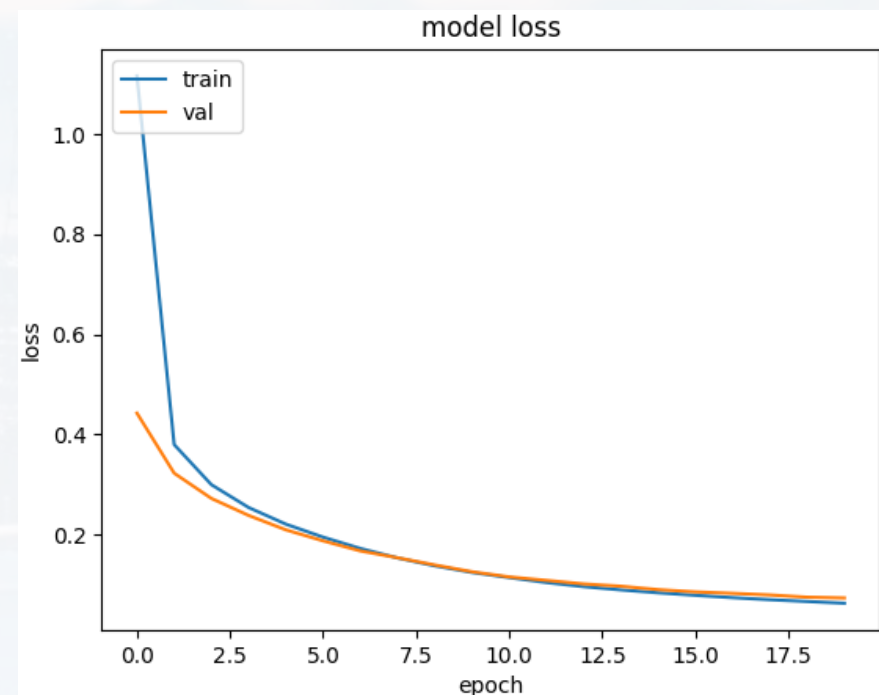
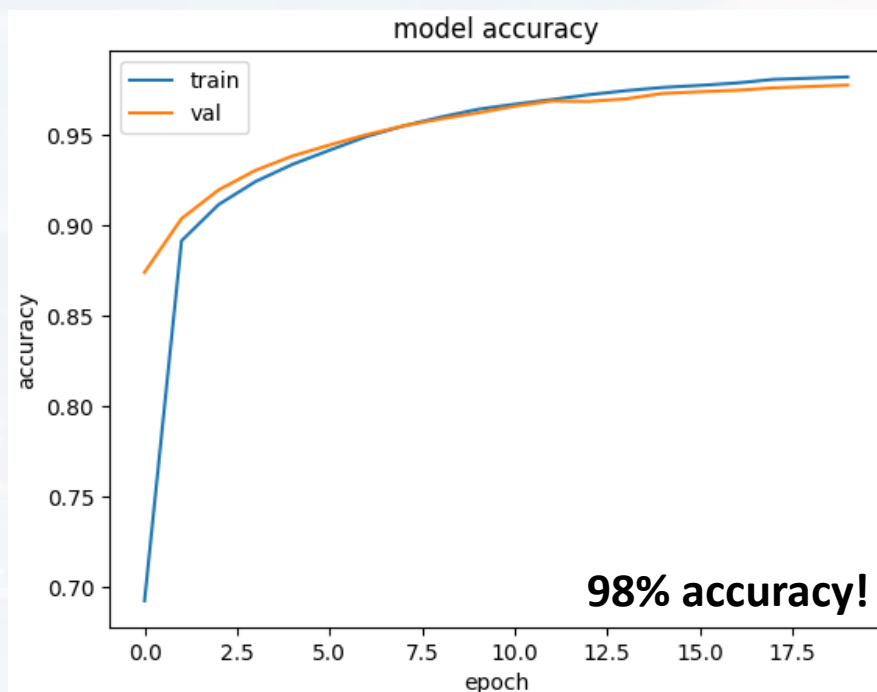
LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation





see `LeNetTF.py` and `LeNetTF.ipynb`

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



training loss should \approx validation loss

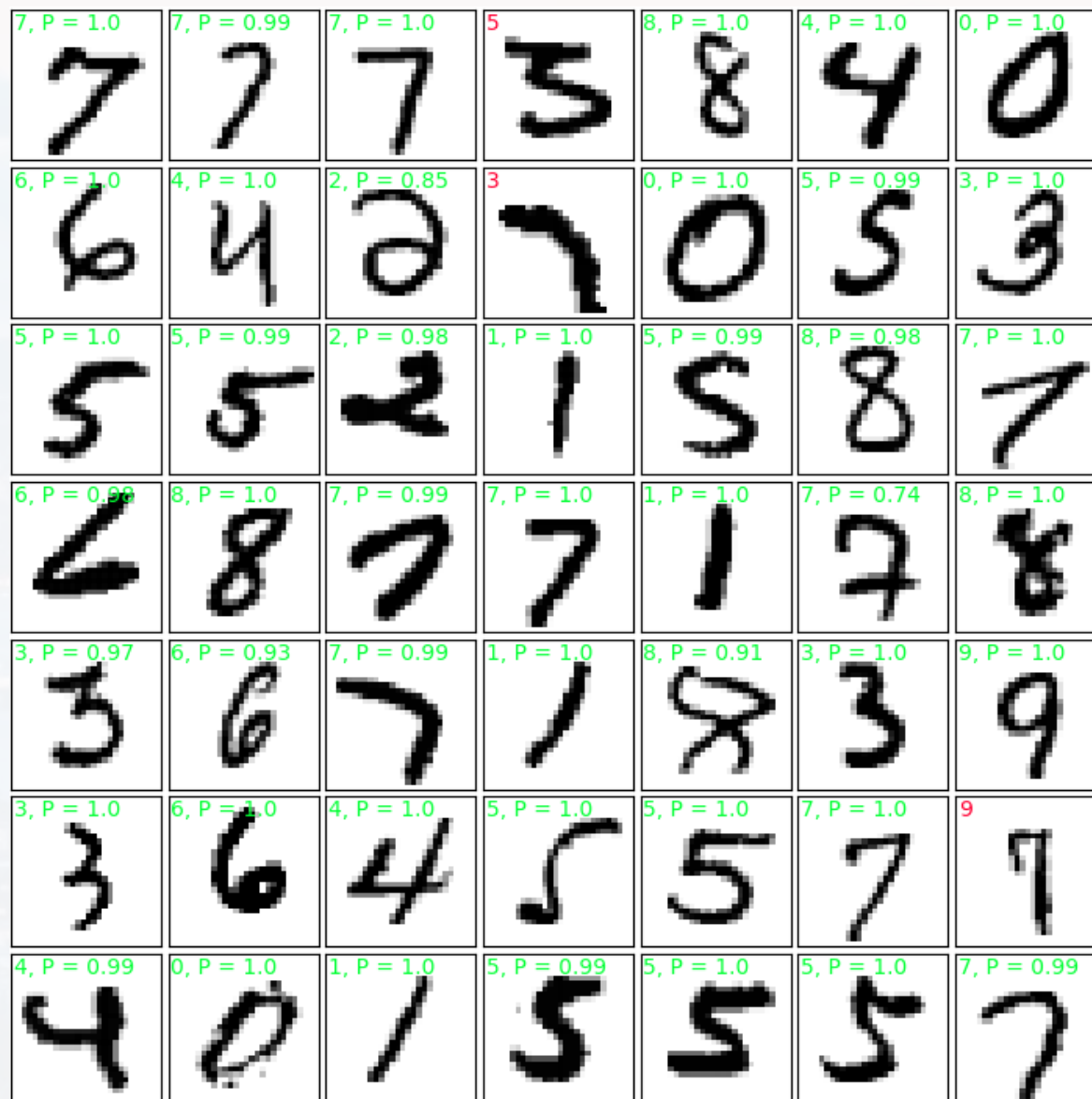
if validation loss \gg training loss \rightarrow overfitting

- too many parameter
- too few images in batch
- too specific/unique batch)



see LeNetTF.py and LeNetTF.ipynb

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



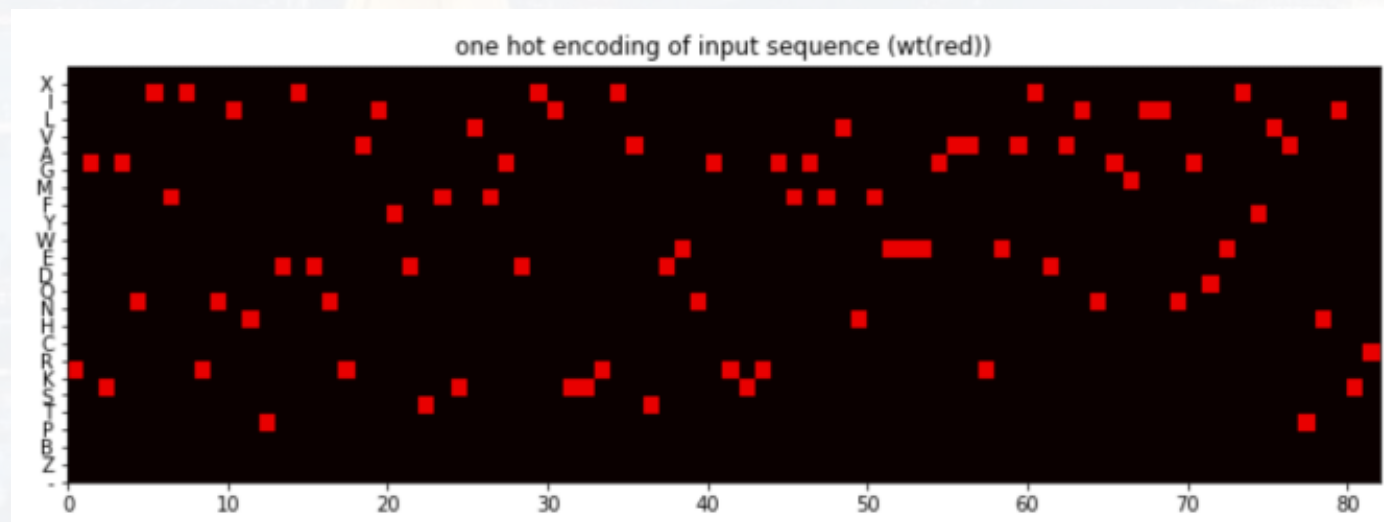


DNA/RNA/AA sequences
natural languages (encoding, see NLP lecture)

motif finding / sequence analysis

	C	A	G	T	C	T	A
4	1	0	0	0	1	0	0
	0	0	1	0	0	0	0
	0	0	0	1	0	1	0
	0	1	0	0	0	0	1
	Sequence Length						

one – hot encoded NT or AA sequences
can be interpreted as b/w images!





LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

- barcodes are short DNA sequence for identifying species
- we want to see, if we can use a CNN for classification
- loading a so called *fasta* file

>BEISA025-19 | **Culex** | COI-5P

```
AACATTATATTTTCATTTTTTGGTGCTTGAGCAGGAATAATTGGAACCTTCTTTAAGTCTTCTTATTCG
AGCTGAATTAAGTCAACCAGGAGTTTTTATTGGGAATGATCAAATTTATAATGTAATTGTTACAGC
TCATGCTTTTATTATAATTTTTTTTATAGTAATACCTATTATAATTGGAGGATTTGGAAATTGATT
AGTTCCTTTAATACTAGGAGCTCCTGATATAGCTTTTCCTCGAATAAATAATATAAGATTTTGAAT
ACTTCCCCCCTCATTAACACTTCTACTTTCTAGTAGTATAGTAGAAAATGGAGCTGGTACAGGTTG
AACAGTATATCCTCCTCTTTCTTCTGGAACAGCTCATGCTGGAGCTTCTGTTGATTTAGCTATTTT
TTCTTTACATTTAGCCGGAATTTCTTCAATTTTAGGAGCTGTAAATTTTATTACTACTGTAATTAA
TATGCGATCTTCTGGTATTACCCTTGATCGAATACCTTTATTTGTTTGATCAGTTGTAATTACTGC
TATTCTTTTATTATTATCTCTTCCTGTTTTAGCTGGAGCTATTACTATATTATTAACAGATCGTAA
TTTAAATACTTCTTTTTTTCGATCCTATTGGAGGAGGAGATCCTATTTTATATCAACATTTATTT
```

>BEISA121-19 | Anopheles | COI-5P

```
AACATTATATTTTATTTTTCGGTGCTTGAGCAGGAATAGTAGGAACCTTCTTTAAGTATTCTTATTCG
```

true label

sequence
as image



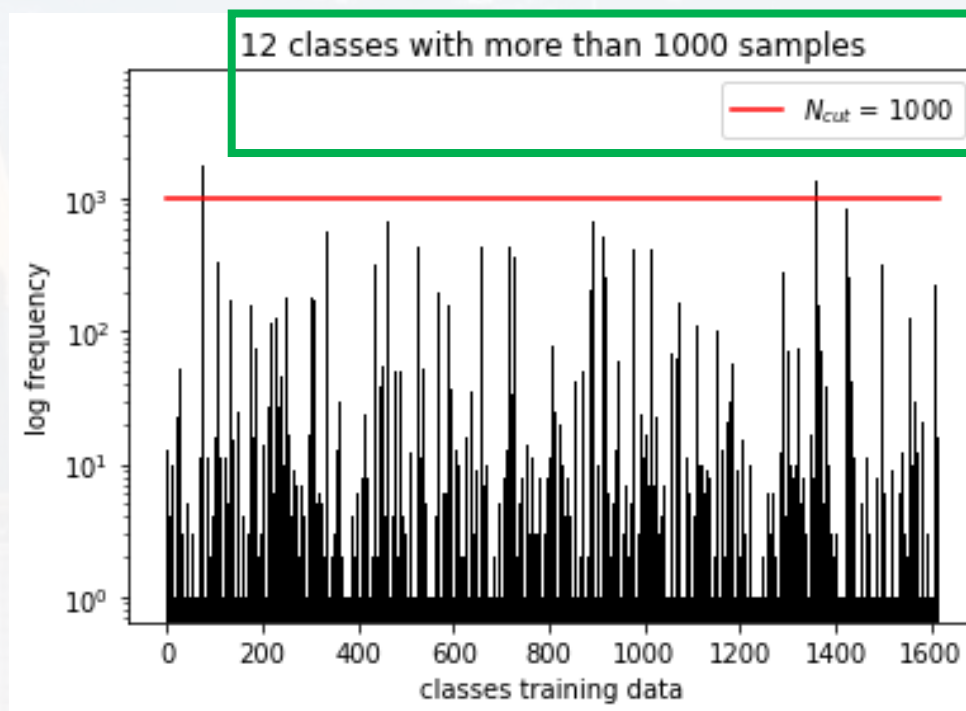
the data set:

- 86k samples
 - 1.6k classes
 - classes are not evenly distributed
- picking only those with >1k samples (12 classes)

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

run the package AnalyzBarcode2.py

```
A = Analyzer()
```



__init__
reads the data,
one-hot encodes
the nucleotides
and passes only
those samples
where Nsample
> Ncut



A.RunCNN()

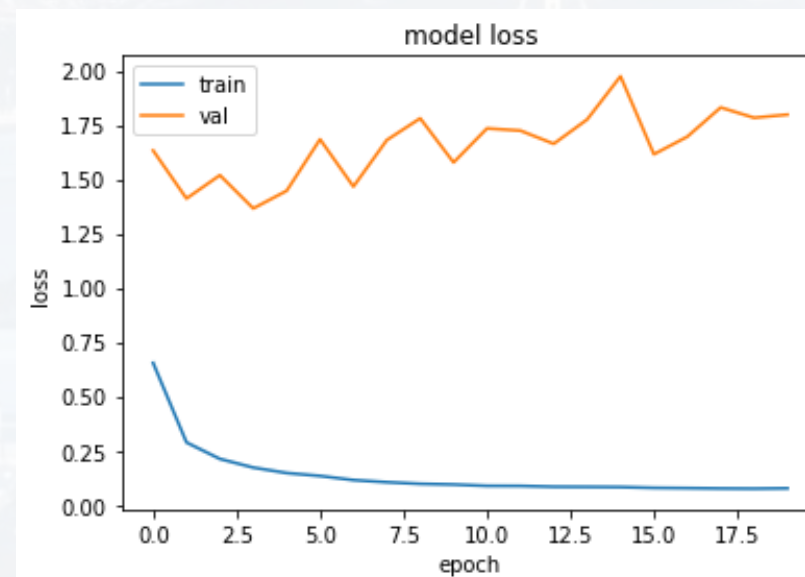
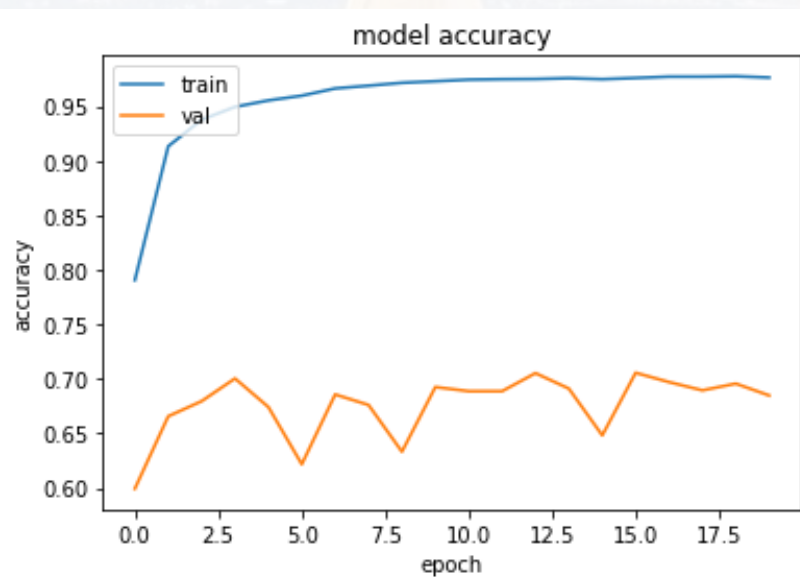
Layer (type)	Output Shape	Param #
conv2d_6 (Conv2D)	(None, 1256, 1, 24)	408
flatten_6 (Flatten)	(None, 30144)	0
dense_12 (Dense)	(None, 84)	2532180
dense_13 (Dense)	(None, 12)	1020

=====

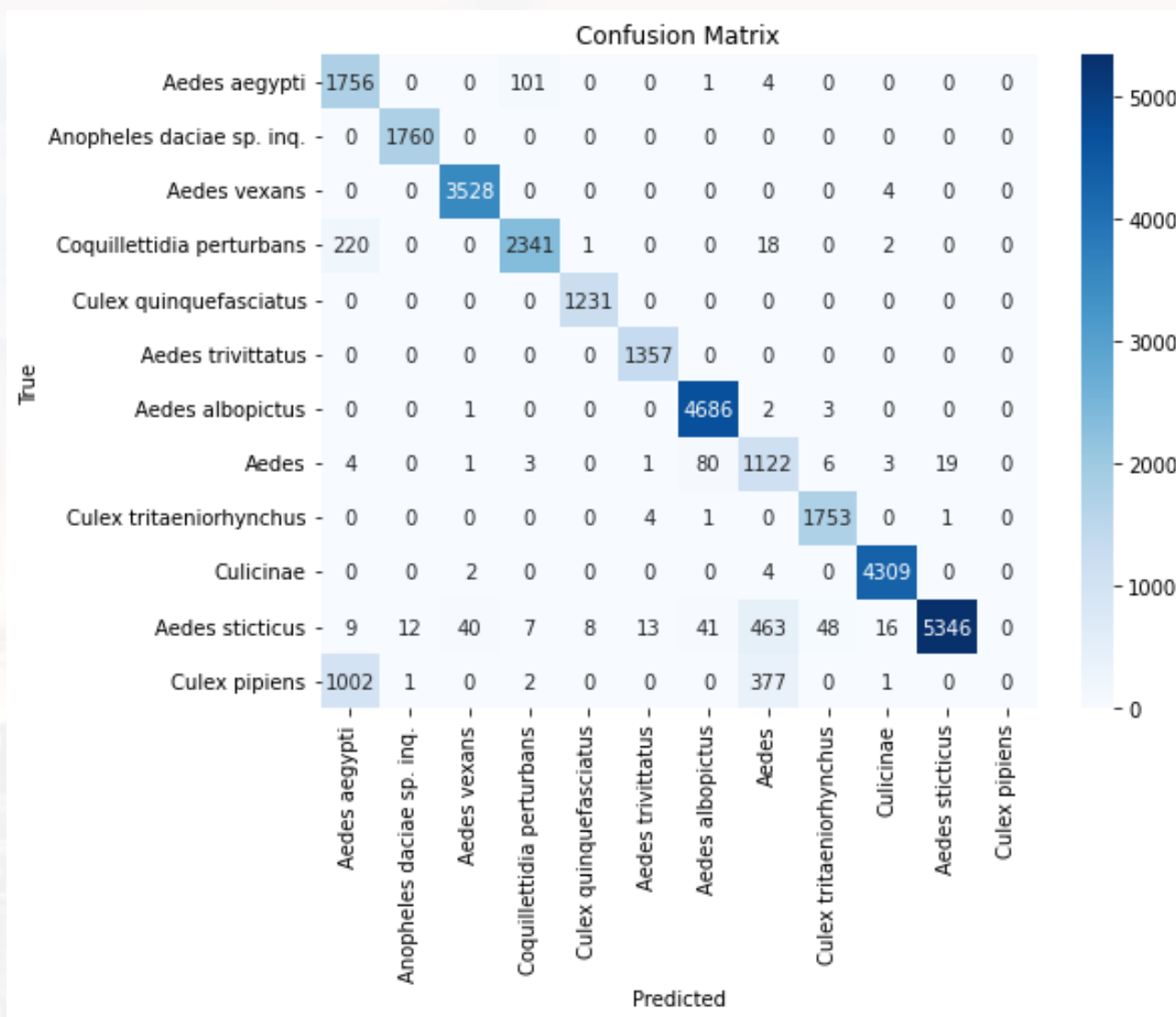
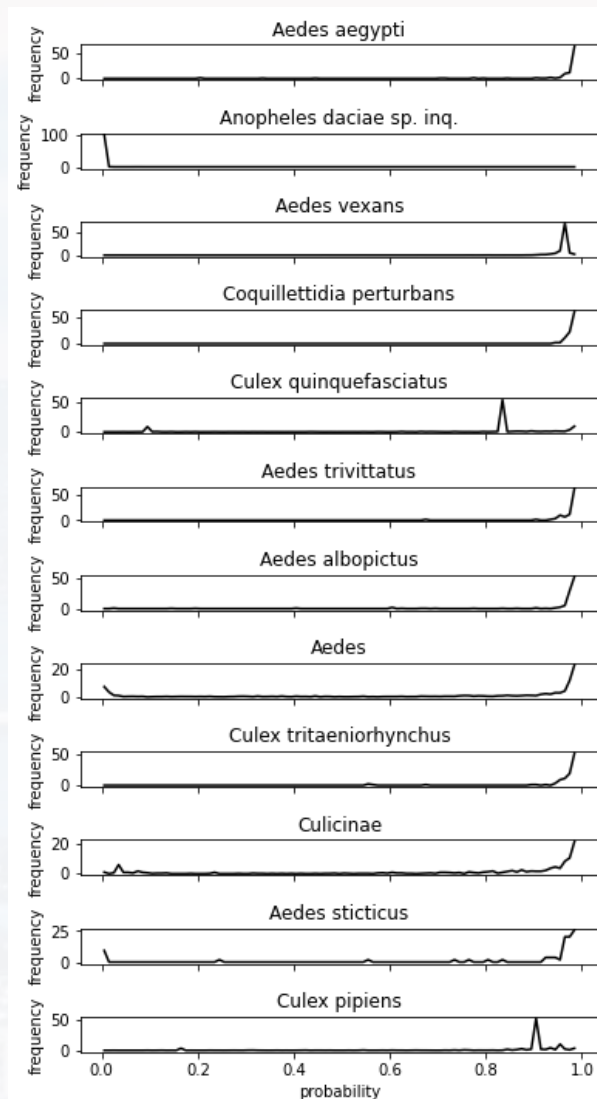
Total params: 2533608 (9.66 MB)
Trainable params: 2533608 (9.66 MB)
Non-trainable params: 0 (0.00 Byte)

runs very simple CNN

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



results are not great,
but it demonstrates
the principle



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



training images have to be labeled!

"labelme"

Description

Labelme is a graphical image annotation tool inspired by <http://labelme.csail.mit.edu>. It is written in Python and uses Qt for its graphical interface.



VOC dataset example of instance segmentation.

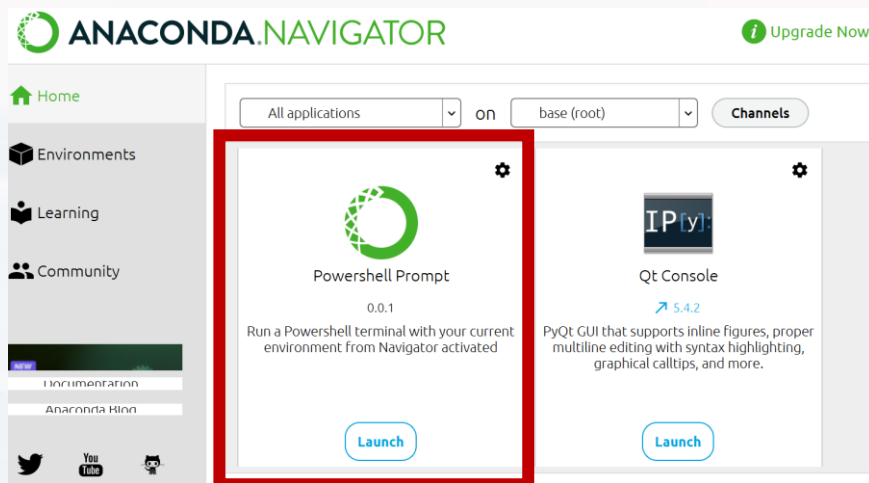


Other examples (semantic segmentation, bbox detection, and classification).



Various primitives (polygon, rectangle, circle, line, and point).

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



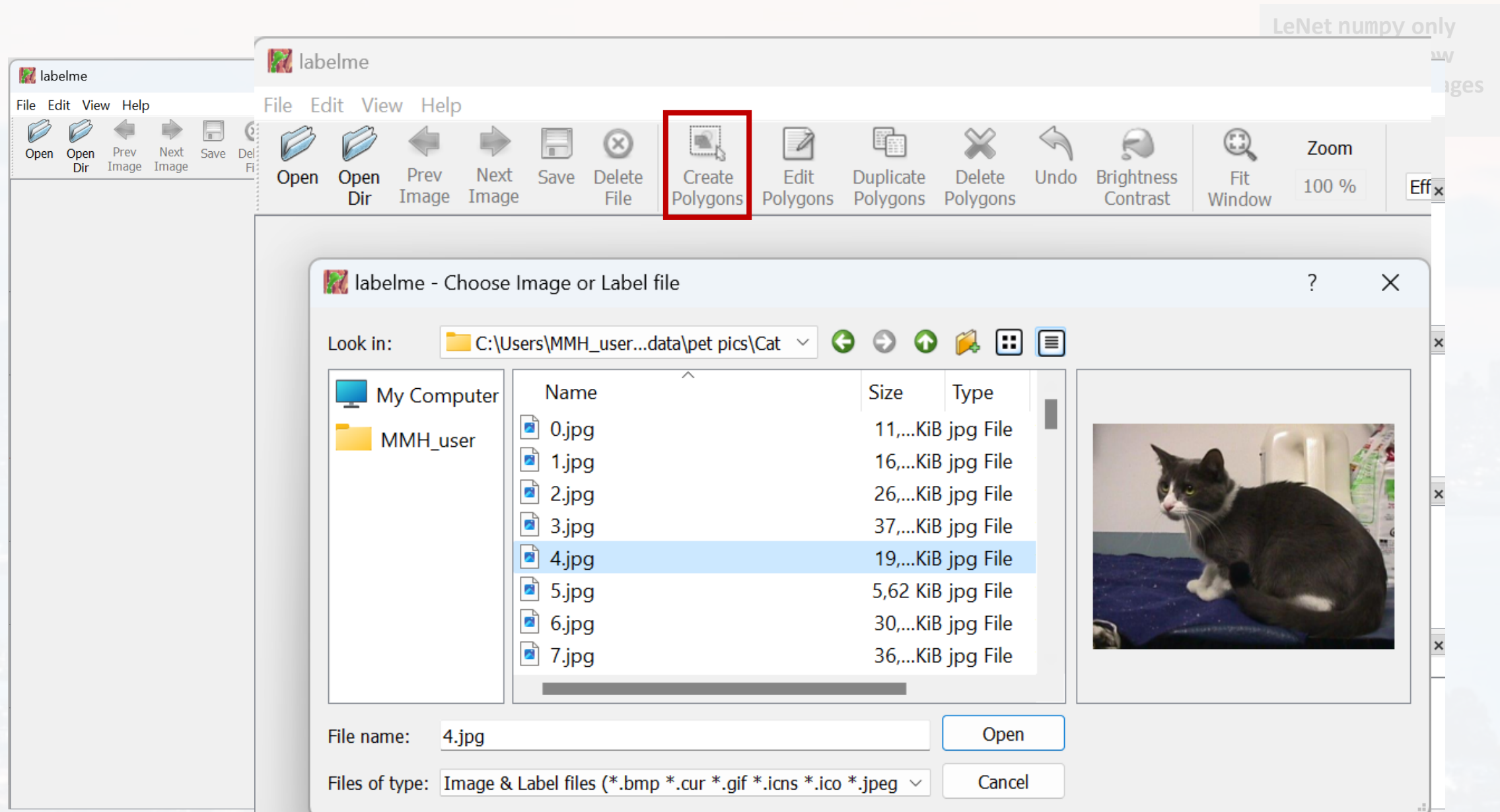
conda install labelme

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

```
C:\WINDOWS\System32\Win... x + v
(base) PS C:\Users\MMH_user> conda create --name=labelme python=3|
```

```
(base) PS C:\Users\MMH_user> conda activate labelme|
```

```
(labelme) PS C:\Users\MMH_user> labelme|
```





training images have to be labeled!

image is saved as .json



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

run within the labelme prompt:

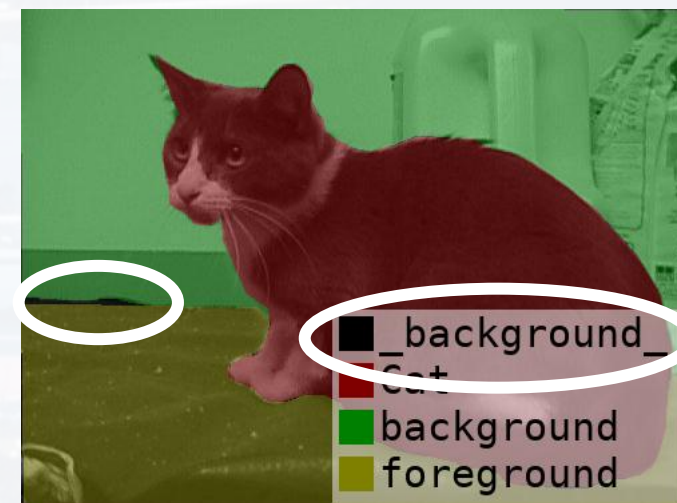
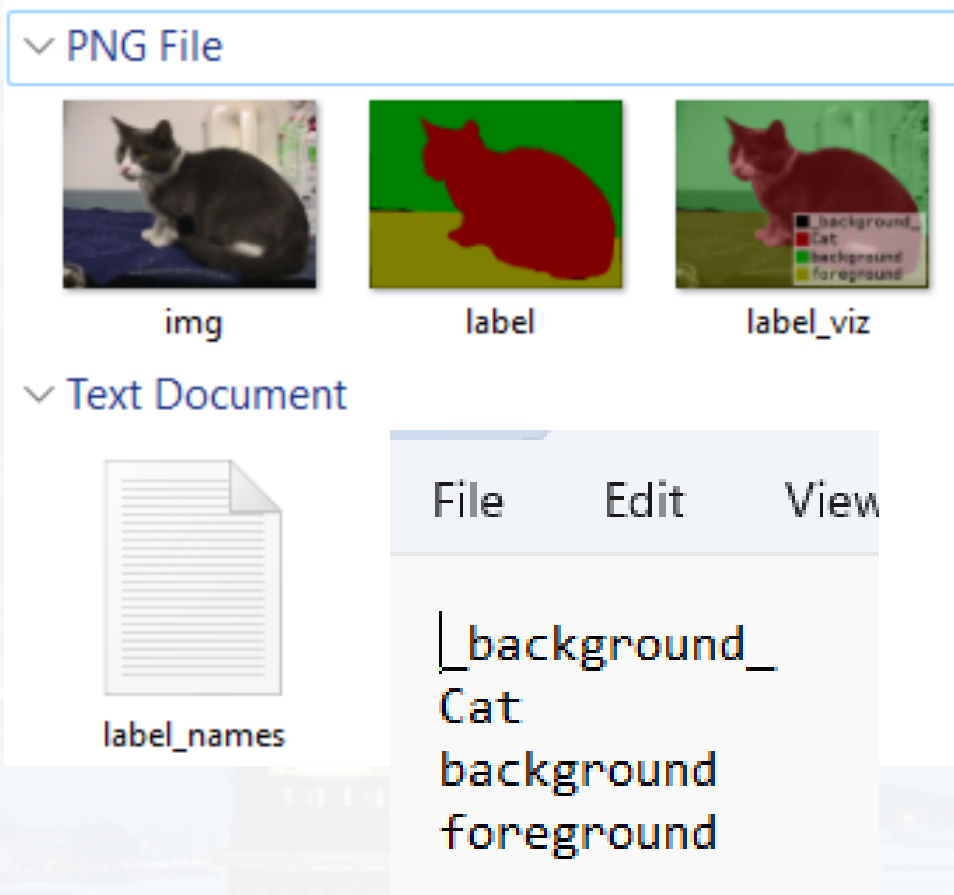
```
labelme_export_json .\Cat4label.json -o Cat4label_json
```





training images have to be labeled!

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation





training images have to be labeled!

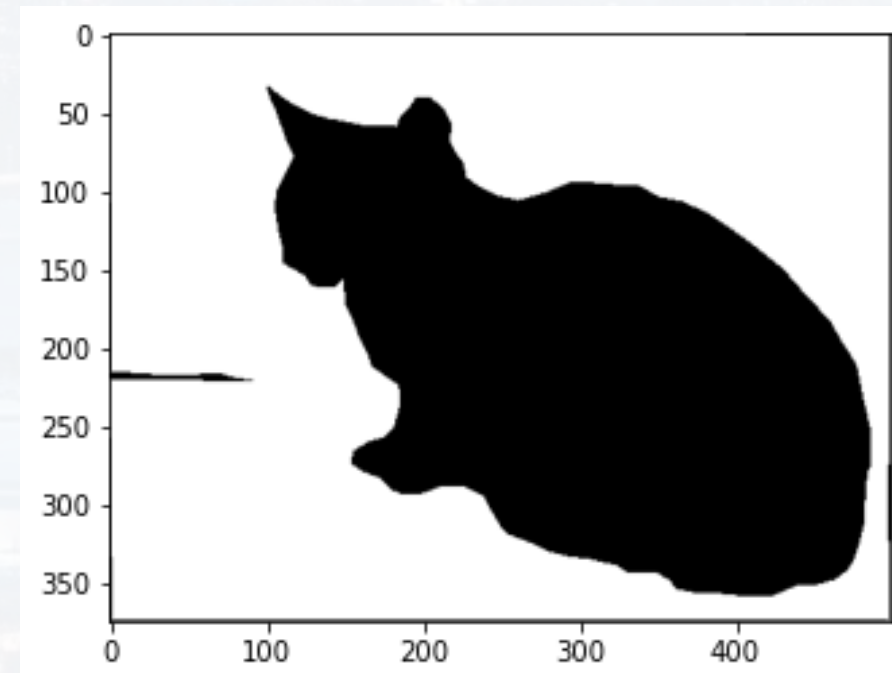
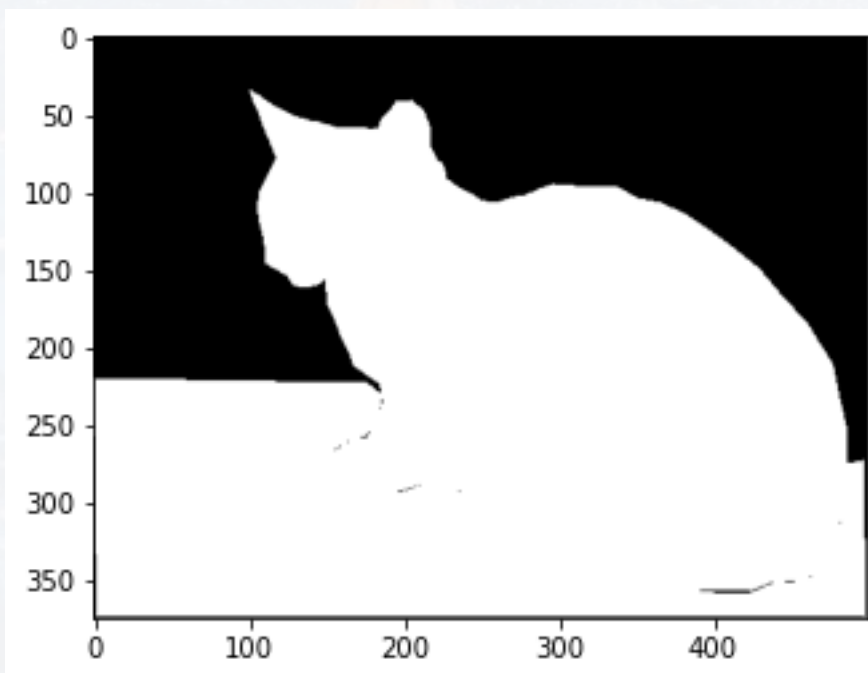
LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



```
I = plt.imread('Cat/Cat4Label_json/Label.png')
```

```
I.shape  
(375, 500, 4)
```

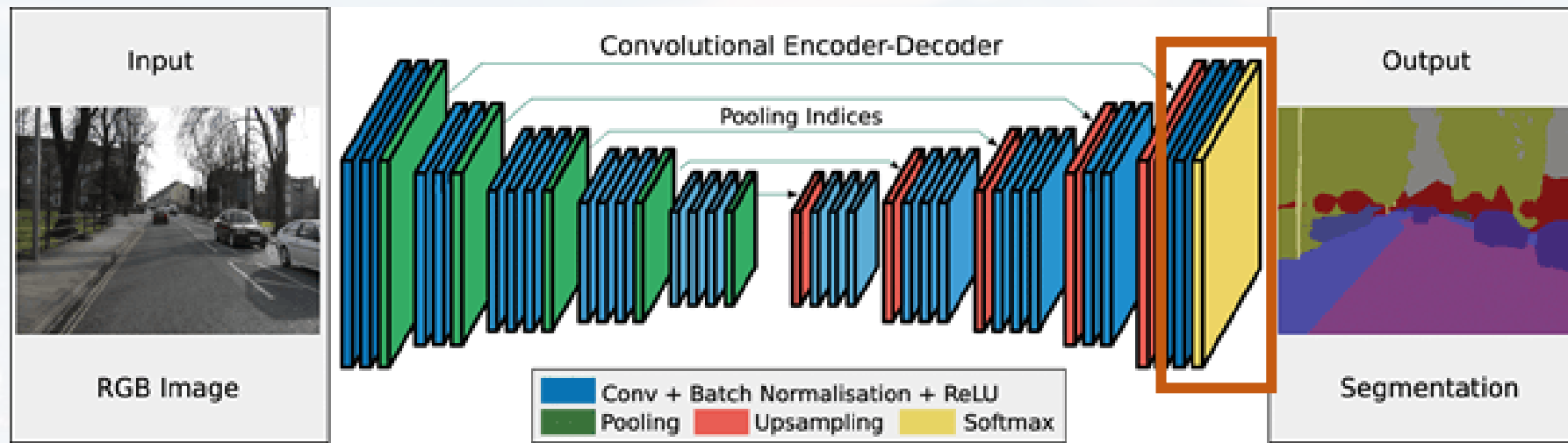
each label = channel





training images have to be labeled!

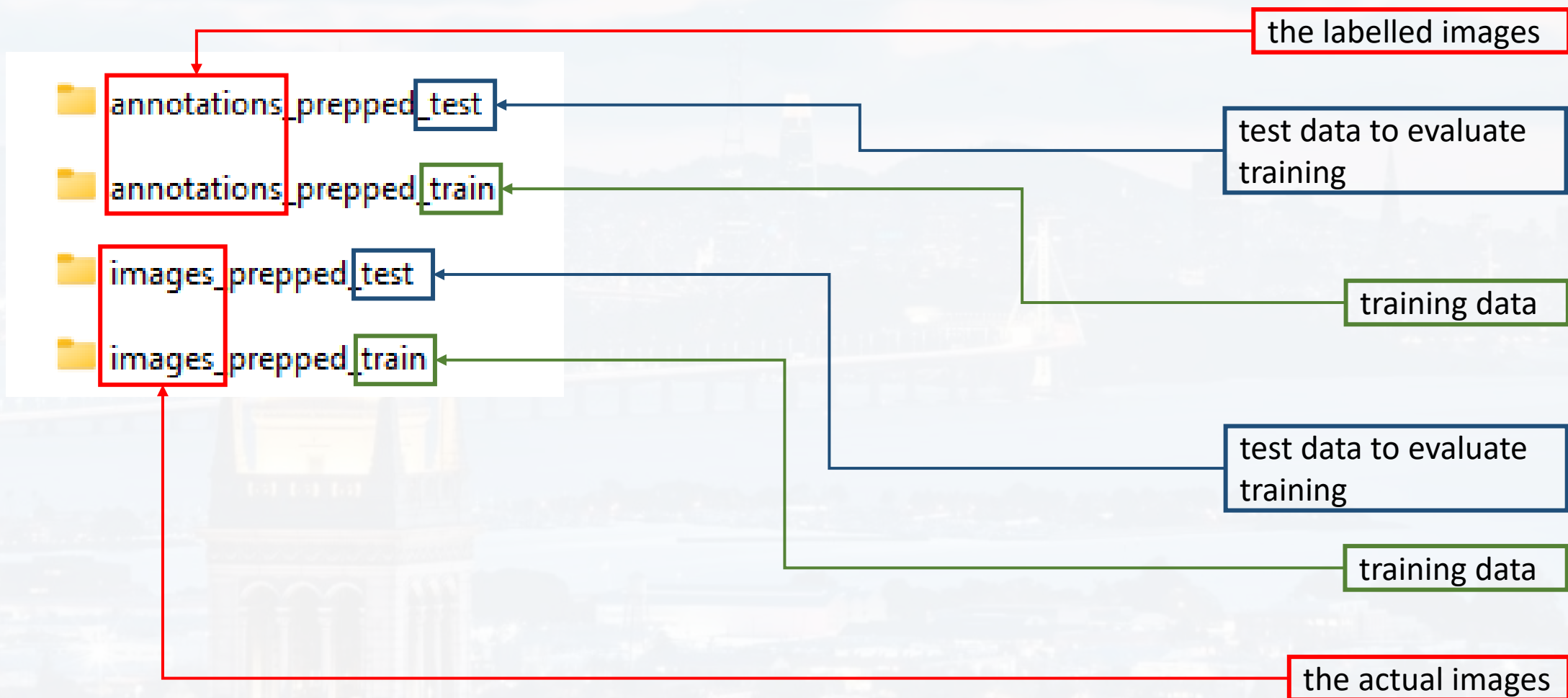
LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation





training images have to be labeled!

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation





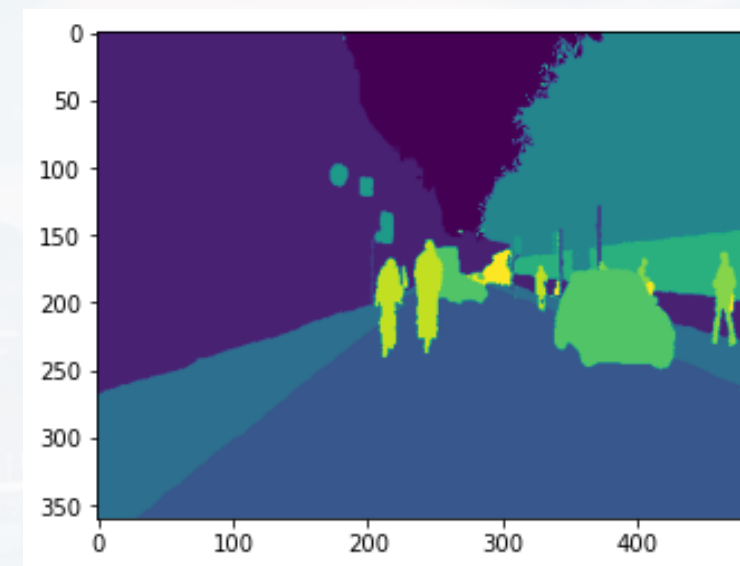
training images have to be labeled!

[nice dataset](#)

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



`images_prepped_test`



`annotations_prepped_test`

```
I = plt.imread('segmentation/pics/annotations_prepped_test/0016E5_07959.png')
```

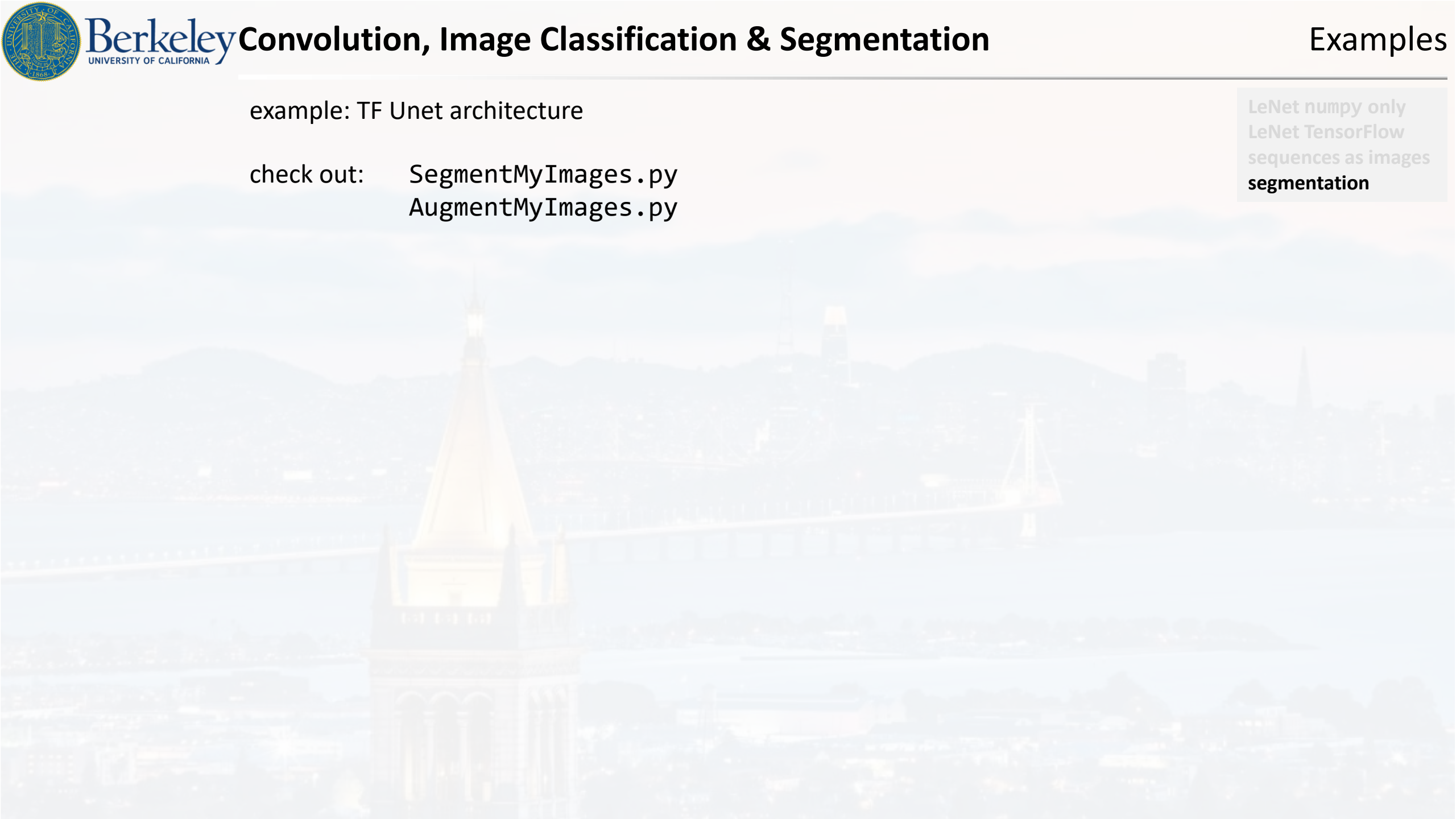
```
I.shape  
(360, 480)
```




example: TF Unet architecture

check out: `SegmentMyImages.py`
`AugmentMyImages.py`

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation





example: TF Unet architecture

```
from keras_segmentation.models.unet import *
```

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

Type	Names
VGG	'vgg16' 'vgg19'
ResNet	'resnet18' 'resnet34' 'resnet50' 'resnet101' 'resnet152'
SE-ResNet	'seresnet18' 'seresnet34' 'seresnet50' 'seresnet101' 'seresnet152'
ResNeXt	'resnext50' 'resnext101'
SE-ResNeXt	'seresnext50' 'seresnext101'
SENet154	'senet154'
DenseNet	'densenet121' 'densenet169' 'densenet201'
Inception	'inceptionv3' 'inceptionresnetv2'
MobileNet	'mobilenet' 'mobilenetv2'
EfficientNet	'efficientnetb0' 'efficientnetb1' 'efficientnetb2' 'efficientnetb3' 'efficientnetb4' 'efficientnetb5' 'efficientnetb6' 'efficientnetb7'

```
model = unet(n_classes = n_classes,\n             input_height = 416, input_width = 608)
```

calling the specific network

```
model.train(\n    train_images      = my_path + r"images_prepped_train//",\n    train_annotations = my_path + r"annotations_prepped_train//",\n    checkpoints_path  = my_path + r"checkpoints//",\n    do_augment        = True,\n    gen_use_multiprocessing = True,\n    auto_resume_checkpoint = True,\n    epochs = 5)
```

saves current weights

Keras provides an augmentation routine



example: TF Unet architecture

```
model = vgg_unet(n_classes = n_classes,\n                 input_height = 416, input_width = 608)
```

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

```
model.train(\n    train_images      = my_path + r"images_prepped_train/",\n    train_annotations = my_path + r"annotations_prepped_train/",\n    checkpoints_path  = my_path + r"checkpoints/",\n    do_augment        = True,\n    gen_use_multiprocessing = True,\n    auto_resume_checkpoint = True,\n    epochs            = 5)
```

Keras provides an
augmentation routine

Note: I always run my **own augmentation** routine,
see e.g. `AugmentMyImages.py`

run:

```
S = SegmentMyImages()
```

```
S.Training()
```




example: TF Unet architecture

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

```
S = SegmentMyImages()
```

```
S.Training()
```

```
Dataset verified!
Epoch 1/5
512/512 [=====] - ETA: 0s - loss: 4.1406 - accuracy: 0.0353      saved ../data/segmentation
pics/checkpoints//.0
512/512 [=====] - 3888s 8s/step - loss: 4.1406 - accuracy: 0.0353
Epoch 2/5
512/512 [=====] - ETA: 0s - loss: 3.7805 - accuracy: 0.1636      saved ../data/segmentation
pics/checkpoints//.1
512/512 [=====] - 4133s 8s/step - loss: 3.7805 - accuracy: 0.1636
Epoch 3/5
512/512 [=====] - ETA: 0s - loss: 3.4534 - accuracy: 0.3338      saved ../data/segmentation
pics/checkpoints//.2
512/512 [=====] - 4072s 8s/step - loss: 3.4534 - accuracy: 0.3338
Epoch 4/5
512/512 [=====] - ETA: 0s - loss: 3.1926 - accuracy: 0.3980      saved ../data/segmentation
pics/checkpoints//.3
512/512 [=====] - 3363s 7s/step - loss: 3.1926 - accuracy: 0.3980
Epoch 5/5
512/512 [=====] - ETA: 0s - loss: 3.0071 - accuracy: 0.4318      saved ../data/segmentation
pics/checkpoints//.4
512/512 [=====] - 3616s 7s/step - loss: 3.0071 - accuracy: 0.4318
```




example: TF Unet architecture

```
MyModel = S.TrainedModel
```

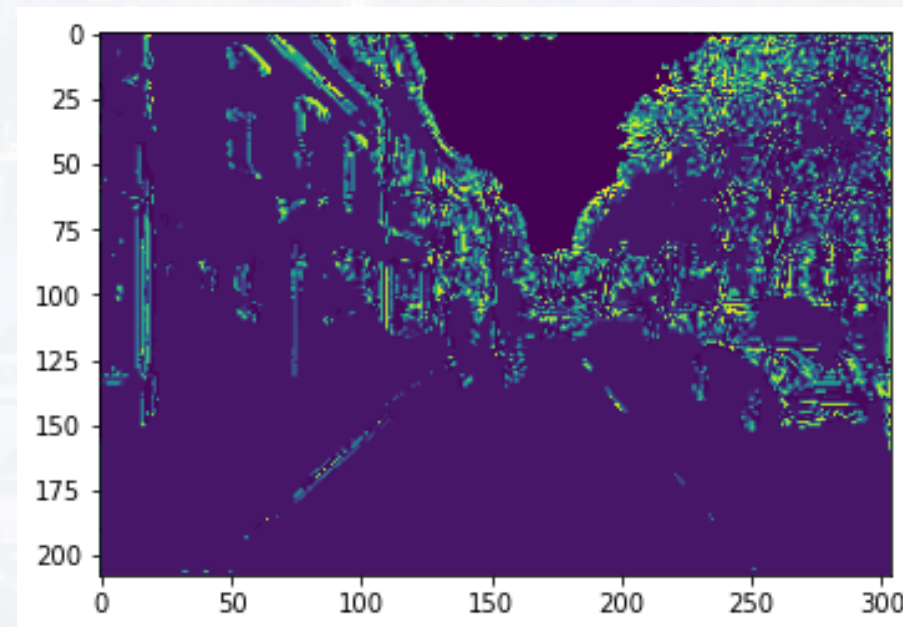
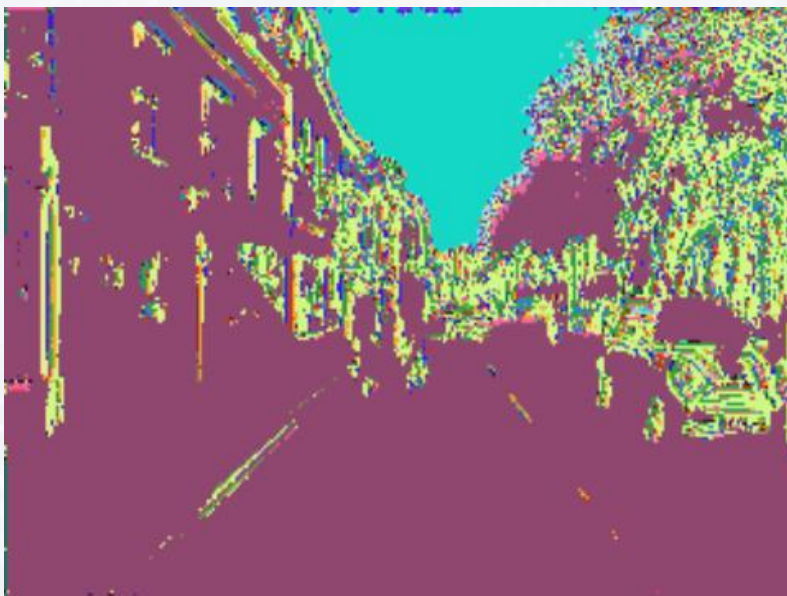
```
MyModel.summary()
```

→ returns the structure of the CNN

applying the trained CNN to an image:

```
out = S.ApplyTrainedNetwork()
```

```
plt.imshow(out)
```



LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



example: TF Unet architecture

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

recovering model
from checkpoints:

```
MyModel = S.TrainedModel  
out      = S.ApplyTrainedNetwork()
```

applying the trained CNN to an image:

```
S.RecoverFromCheckpoint()
```

untrained
model (just
CNN itself)

```
#loading untrained CNN
```

```
model = self.model
```

```
if not image_name:
```

```
.... image_name = '0016E5_07965.png'
```

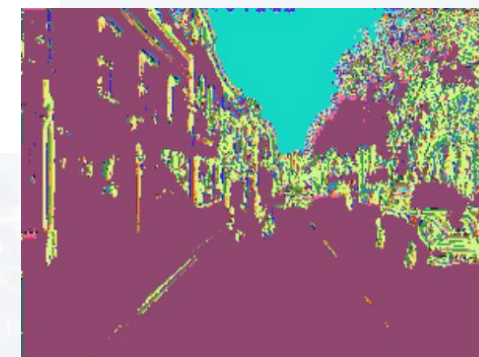
```
....
```

```
#calling input from checkpoints
```

```
latest = tf.train.latest_checkpoint(self.checkpoint_path)
```

```
model.load_weights(latest)
```

transfer the saved weights to untrained
network → now it starts from latest
training state





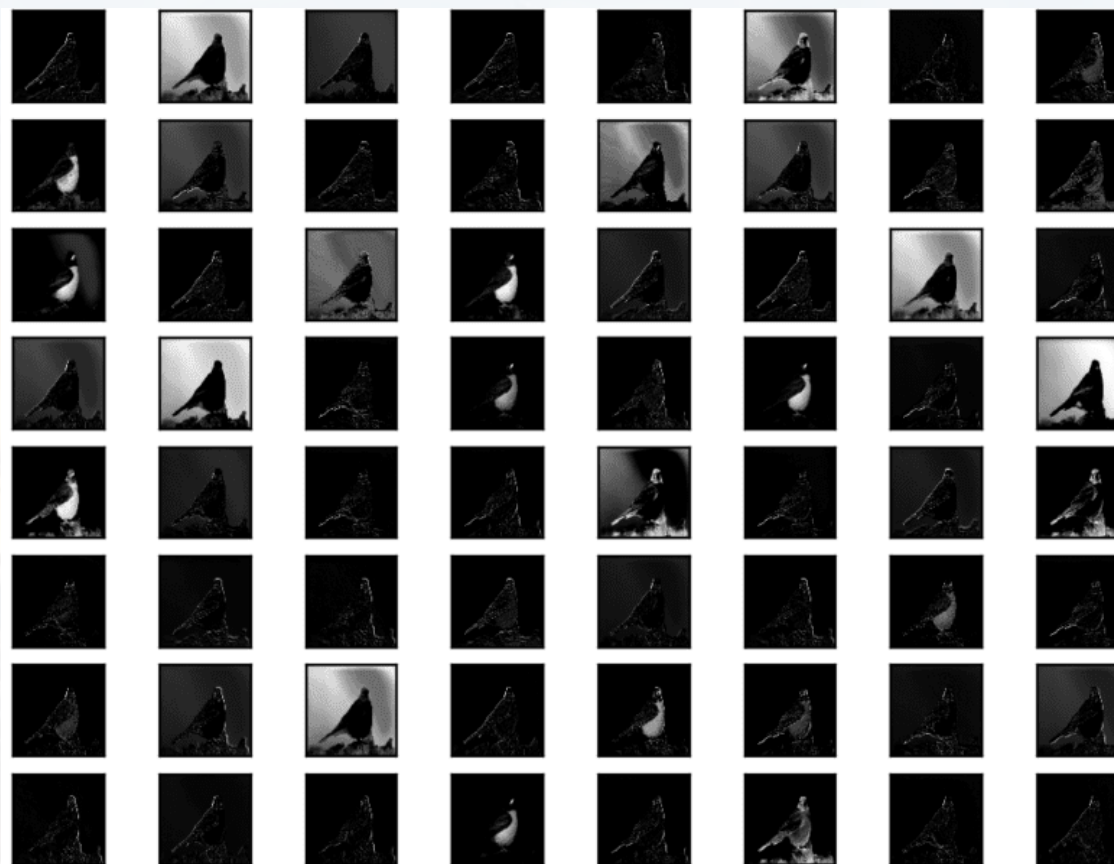
example: TF Unet architecture

visualizing weights:

→ see `model.layers`

[nice example](#)

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation



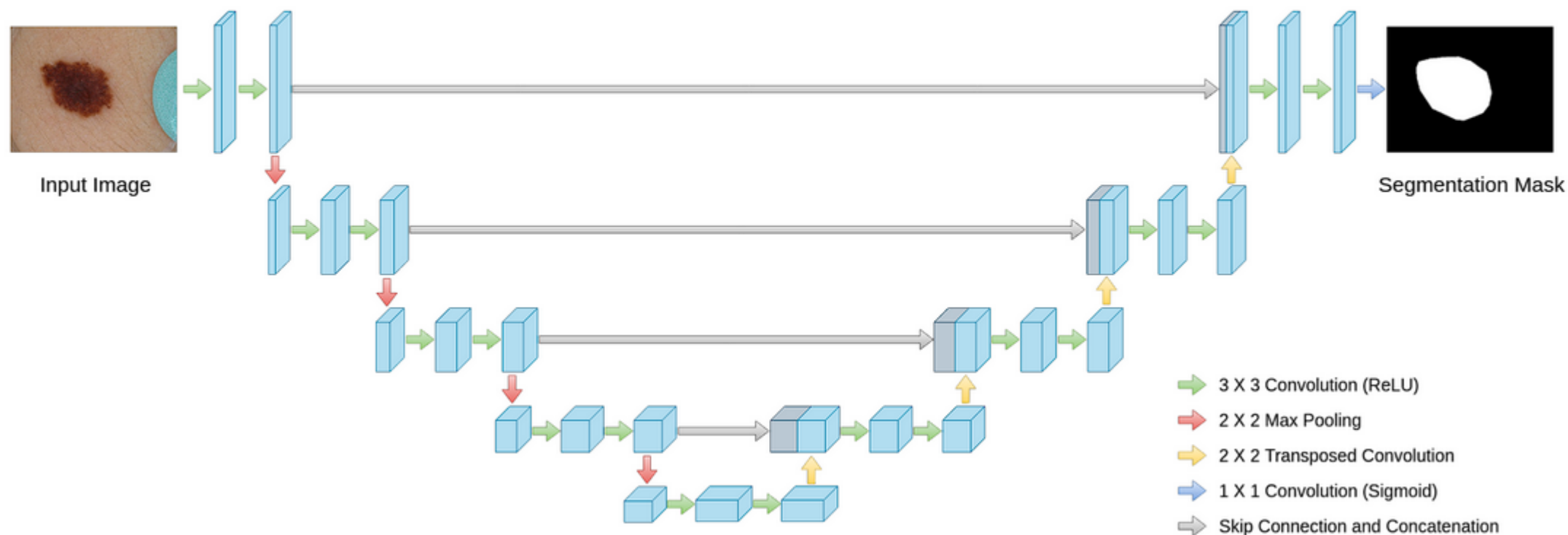


example: **custom** Unet architecture

see Unet architecture.ipynb

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

<https://www.kaggle.com/code/abdallahwagih/brain-tumor-segmentation-unet-dice-coef-89-6/notebook>



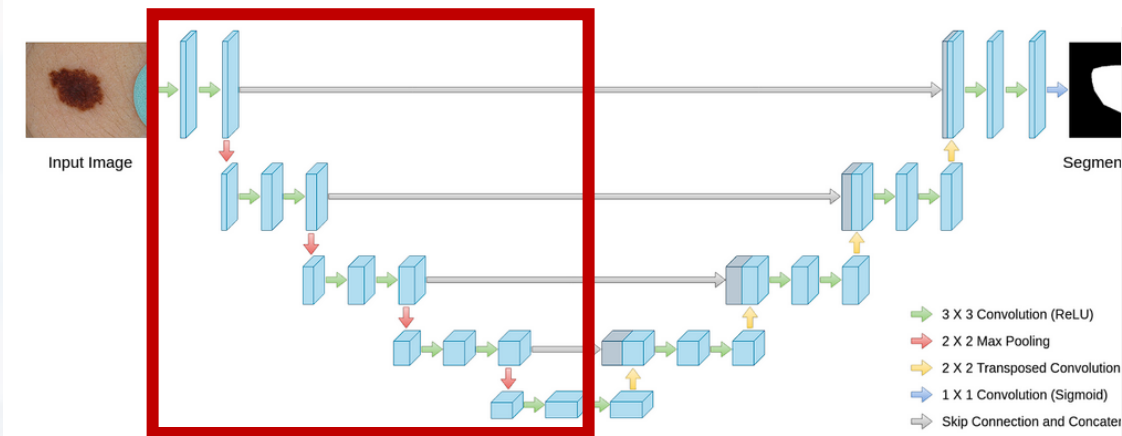


example: **custom** Unet architecture

see Unet architecture.ipynb

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

<https://www.kaggle.com/code/abdallahwagih/brain-tumor-segmentation-unet-dice-coef-89-6/notebook>



```
def unet(input_size = (256, 256, 3)):
    inputs = Input(input_size)

    # First DownConvolution / Encoder Leg will begin, so start with Conv2D
    conv1 = Conv2D(filters = 64, kernel_size = (3, 3), padding = "same")(inputs)
    bn1 = Activation("relu")(conv1)
    conv1 = Conv2D(filters = 64, kernel_size = (3, 3), padding = "same")(bn1)
    bn1 = BatchNormalization(axis = 3)(conv1)
    bn1 = Activation("relu")(bn1)
    pool1 = MaxPooling2D(pool_size = (2, 2))(bn1)

    conv2 = Conv2D(filters = 128, kernel_size = (3, 3), padding = "same")(pool1)
    bn2 = Activation("relu")(conv2)
    conv2 = Conv2D(filters = 128, kernel_size = (3, 3), padding = "same")(bn2)
    bn2 = BatchNormalization(axis = 3)(conv2)
    bn2 = Activation("relu")(bn2)
    pool2 = MaxPooling2D(pool_size = (2, 2))(bn2)

    conv3 = Conv2D(filters = 256, kernel_size = (3, 3), padding = "same")(pool2)
    bn3 = Activation("relu")(conv3)
    conv3 = Conv2D(filters = 256, kernel_size = (3, 3), padding = "same")(bn3)
    bn3 = BatchNormalization(axis = 3)(conv3)
    bn3 = Activation("relu")(bn3)
    pool3 = MaxPooling2D(pool_size = (2, 2))(bn3)

    conv4 = Conv2D(filters = 512, kernel_size = (3, 3), padding = "same")(pool3)
    bn4 = Activation("relu")(conv4)
    conv4 = Conv2D(filters = 512, kernel_size = (3, 3), padding = "same")(bn4)
    bn4 = BatchNormalization(axis = 3)(conv4)
    bn4 = Activation("relu")(bn4)
    pool4 = MaxPooling2D(pool_size = (2, 2))(bn4)

    conv5 = Conv2D(filters = 1024, kernel_size = (3, 3), padding = "same")(pool4)
    bn5 = Activation("relu")(conv5)
    conv5 = Conv2D(filters = 1024, kernel_size = (3, 3), padding = "same")(bn5)
    bn5 = BatchNormalization(axis = 3)(conv5)
    bn5 = Activation("relu")(bn5)
```



example: **custom** Unet architecture

see Unet architecture.ipynb

LeNet numpy only
LeNet TensorFlow

```
up6 = concatenate([Conv2DTranspose(512, kernel_size = (2, 2), strides = (2, 2), padding = "same")(bn5), conv4], axis = 3)
conv6 = Conv2D(filters = 512, kernel_size = (3, 3), padding = "same")(up6)
bn6 = Activation("relu")(conv6)
conv6 = Conv2D(filters = 512, kernel_size = (3, 3), padding = "same")(bn6)
bn6 = BatchNormalization(axis = 3)(conv6)
bn6 = Activation("relu")(bn6)

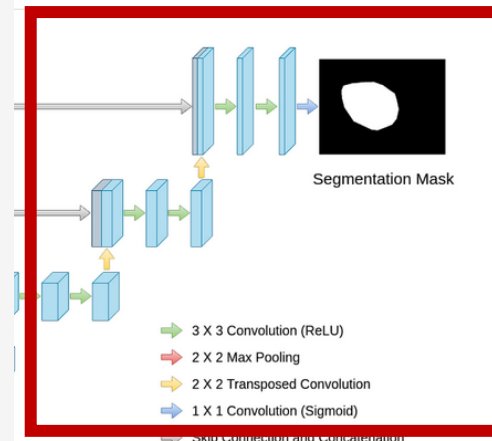
up7 = concatenate([Conv2DTranspose(256, kernel_size = (2, 2), strides = (2, 2), padding = "same")(bn6), conv3], axis = 3)
conv7 = Conv2D(filters = 256, kernel_size = (3, 3), padding = "same")(up7)
bn7 = Activation("relu")(conv7)
conv7 = Conv2D(filters = 256, kernel_size = (3, 3), padding = "same")(bn7)
bn7 = BatchNormalization(axis = 3)(conv7)
bn7 = Activation("relu")(bn7)

up8 = concatenate([Conv2DTranspose(128, kernel_size = (2, 2), strides = (2, 2), padding = "same")(bn7), conv2], axis = 3)
conv8 = Conv2D(filters = 128, kernel_size = (3, 3), padding = "same")(up8)
bn8 = Activation("relu")(conv8)
conv8 = Conv2D(filters = 128, kernel_size = (3, 3), padding = "same")(bn8)
bn8 = BatchNormalization(axis = 3)(conv8)
bn8 = Activation("relu")(bn8)

up9 = concatenate([Conv2DTranspose(64, kernel_size = (2, 2), strides = (2, 2), padding = "same")(bn8), conv1], axis = 3)
conv9 = Conv2D(filters = 64, kernel_size = (3, 3), padding = "same")(up9)
bn9 = Activation("relu")(conv9)
conv9 = Conv2D(filters = 64, kernel_size = (3, 3), padding = "same")(bn9)
bn9 = BatchNormalization(axis = 3)(conv9)
bn9 = Activation("relu")(bn9)

conv10 = Conv2D(filters = 1, kernel_size = (1, 1), activation = "sigmoid")(bn9)

return Model(inputs = [inputs], outputs = [conv10])
```



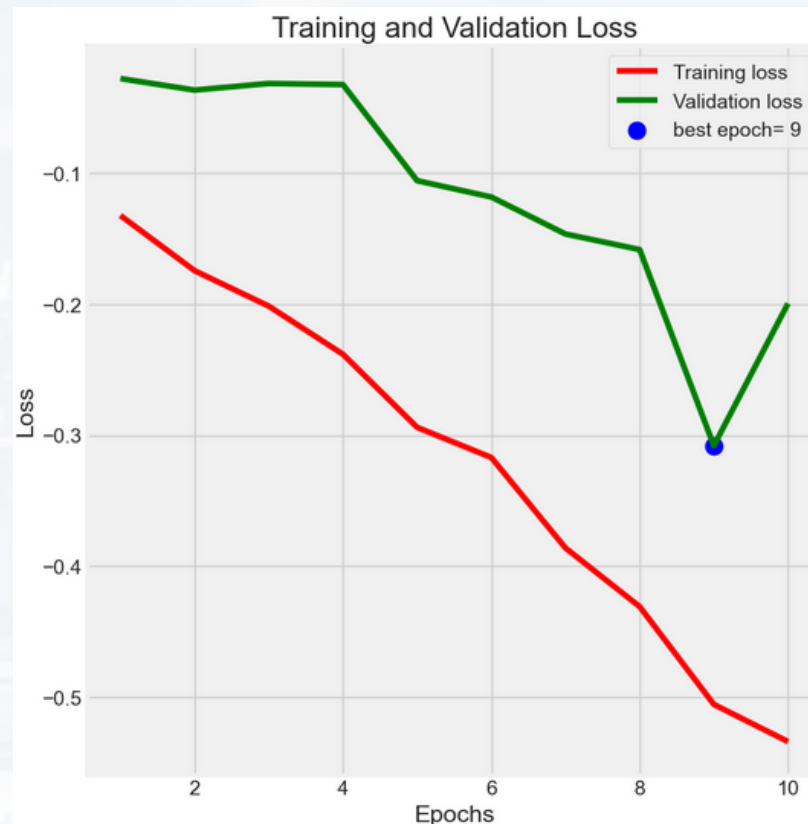
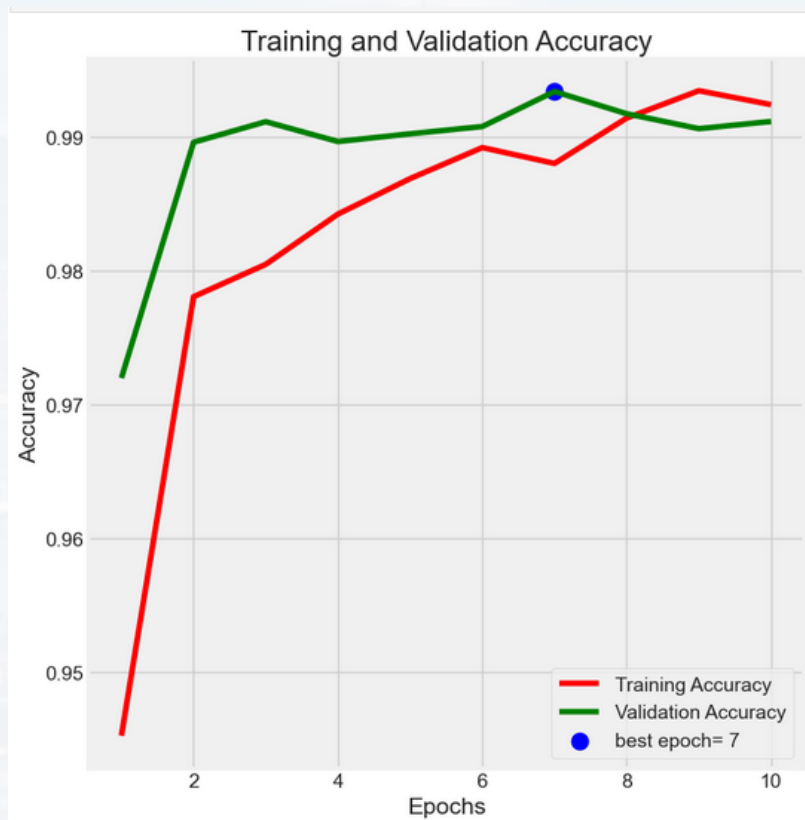


example: **custom** Unet architecture

see Unet architecture.ipynb

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

```
Total params: 31043521 (118.42 MB)
Trainable params: 31037633 (118.40 MB)
Non-trainable params: 5888 (23.00 KB)
```





example: **custom** Unet architecture

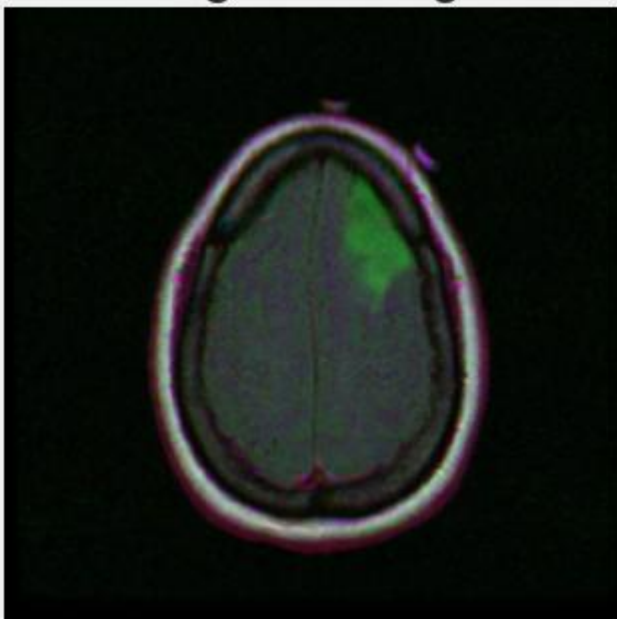
see Unet `architecture.ipynb`

LeNet numpy only
LeNet TensorFlow
sequences as images
segmentation

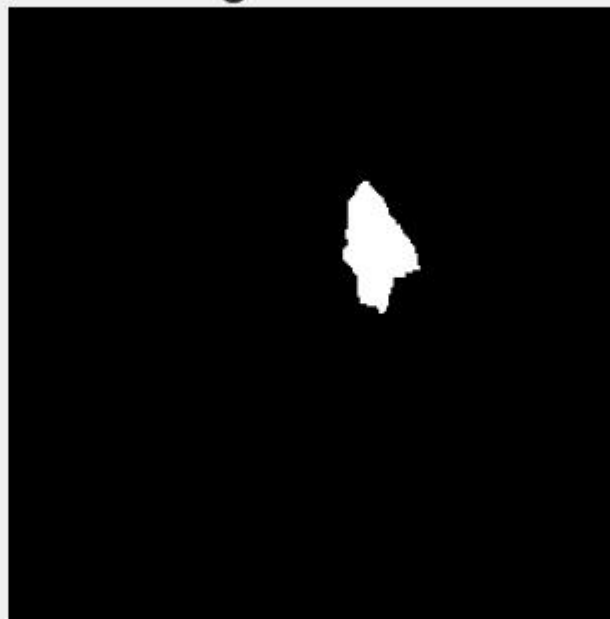
```
Total params: 31043521 (118.42 MB)
Trainable params: 31037633 (118.40 MB)
Non-trainable params: 5888 (23.00 KB)
```

3,000 training images, 400 validation images, 10 epochs, 10hrs (Lenovo T14)

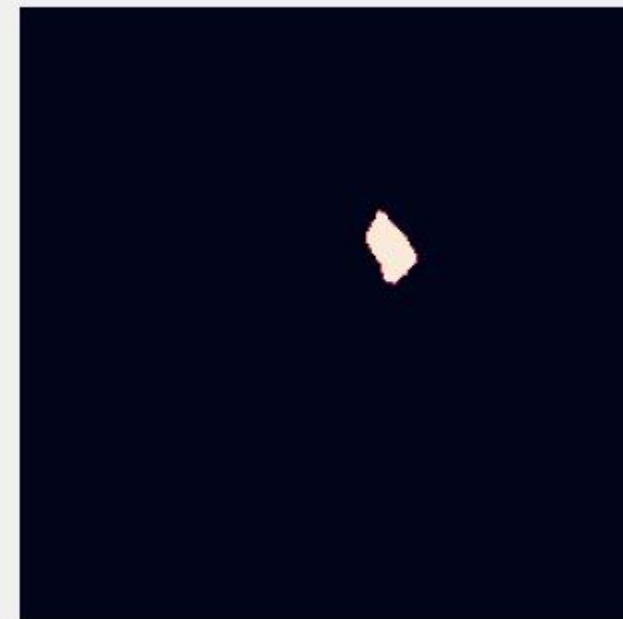
Original Image



Original Mask



Prediction





Thank you very much for your attention!

