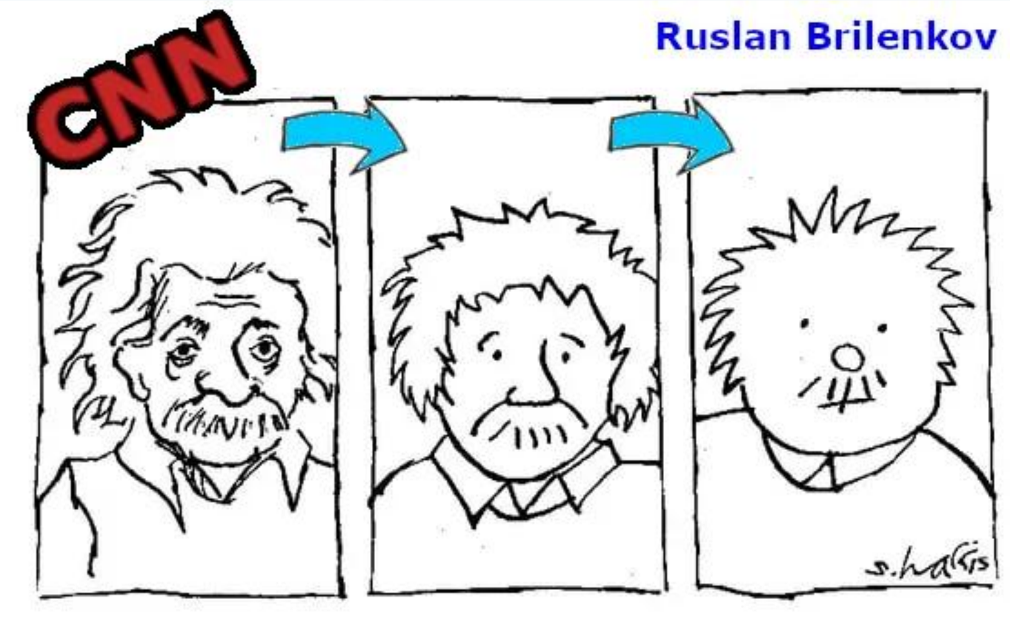
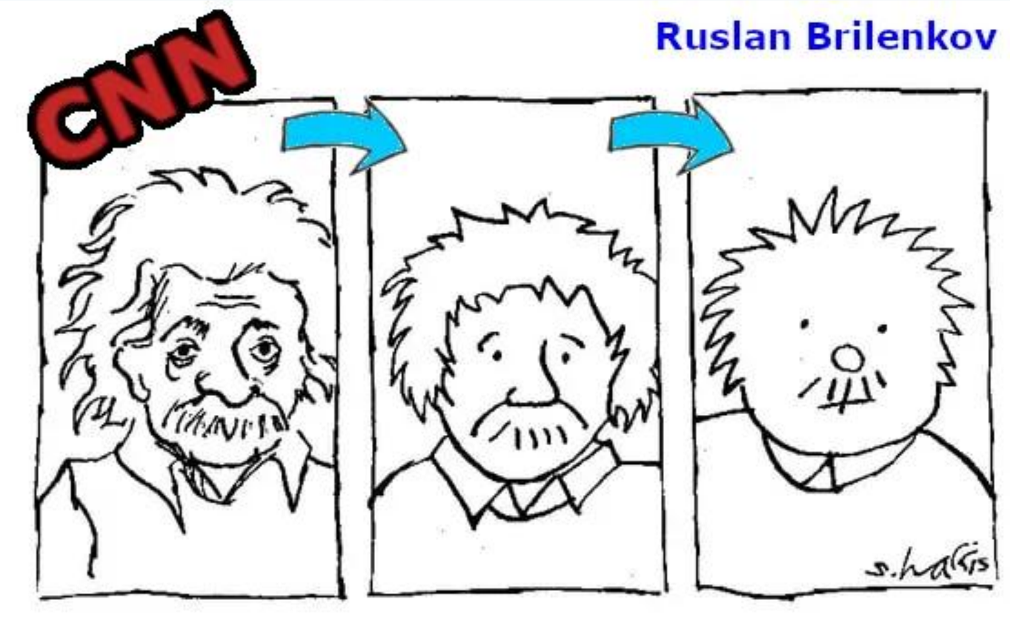
Sequential Classification CNNs - LeNet



Outline:

- What is LeNet?
- Forward Part *Spelled Out*
- Backward Part *Spelled Out*
- LeNet `Keras` `TensorFlow`

Sequential Classification CNNs - LeNet



Outline:

- **What is LeNet?**
- Forward Part *Spelled Out*
- Backward Part *Spelled Out*
- LeNet `Keras TensorFlow`

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

- only **seven layers** in total

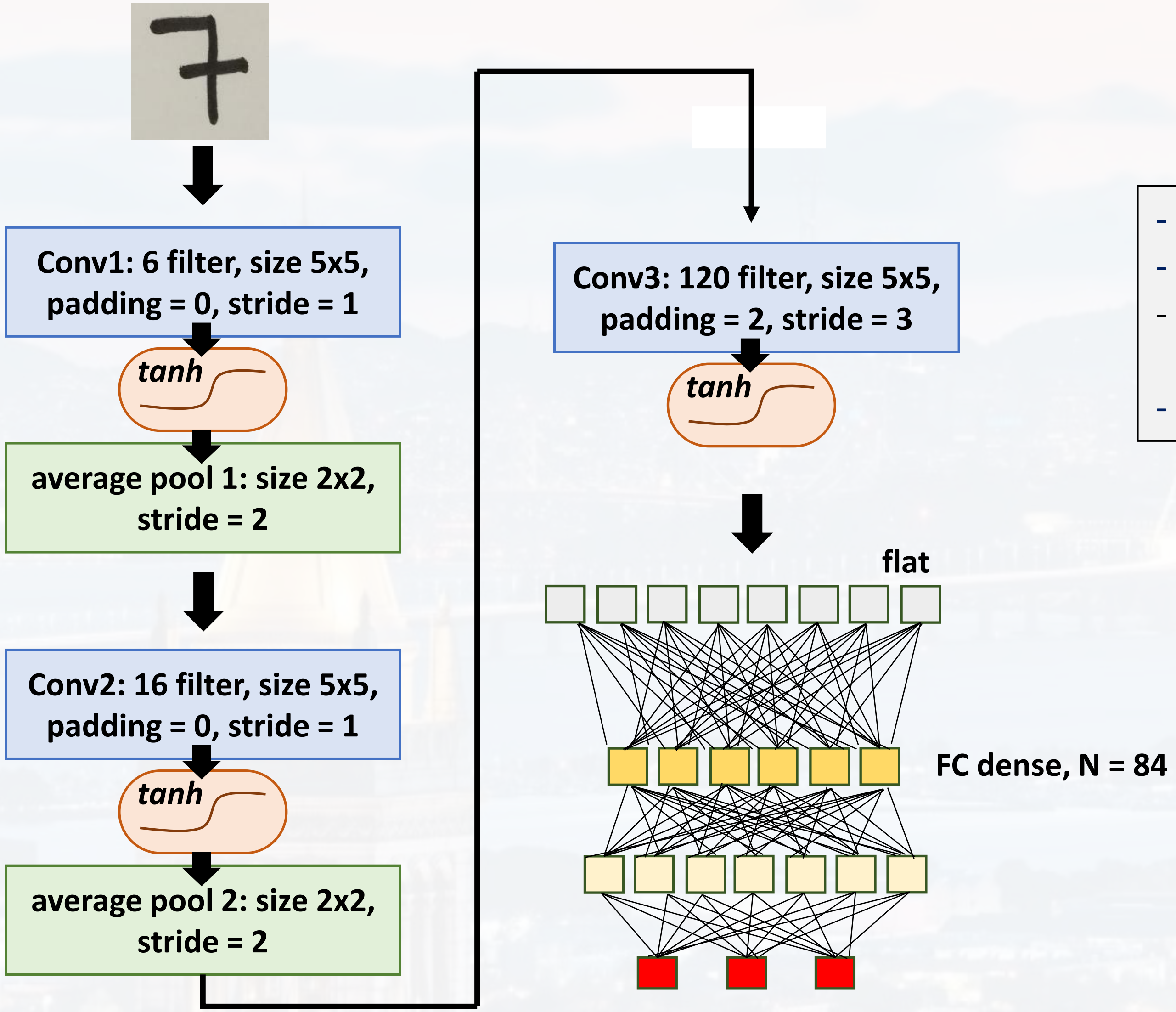
- different versions

- modern CNNs (google, ResNet etc have **100 or more** layers)

Convolution, Image Classification

What is LeNet?

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



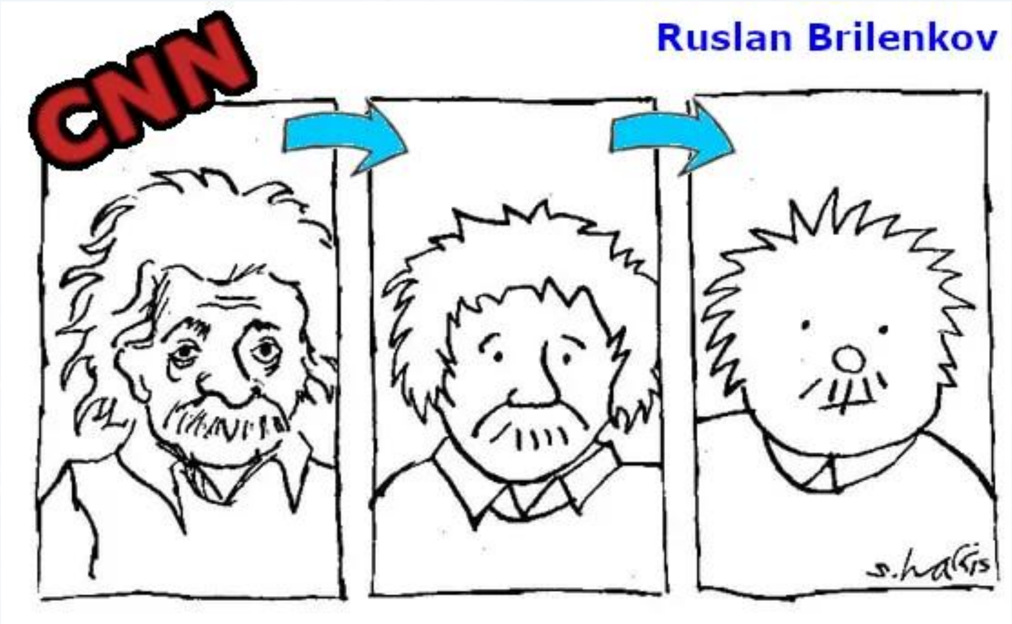
- ***tanh*** as activation function
- **three different** filter
- from Conv1 → Conv2 not all the **channels are combined**
- **average** pool

Convolution, Image Classification

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)
```

Sequential Classification CNNs - LeNet



Outline:

- What is LeNet?
- **Forward Part *Spelled Out***
- Backward Part *Spelled Out*
- LeNet Keras TensorFlow

Convolution, Image Classification

Forward Part *Spelled Out*

see MyCNN.py

- MyCNN.py
  - Read_Scale_Imds
  - ConvLayer
    - __init__
    - forward
    - backward
  - Average_Pool
  - Min_Pool
  - Max_Pool
  - Sigmoid
  - Tanh
  - Activation_ReLU
  - Flat
  - Layer_Dense
  - Activation_Softmax
  - Loss
  - Loss_CategoricalCrossEntropy
  - CalcSoftmaxLossGrad
  - Optimizer_SGD

we know these parts already

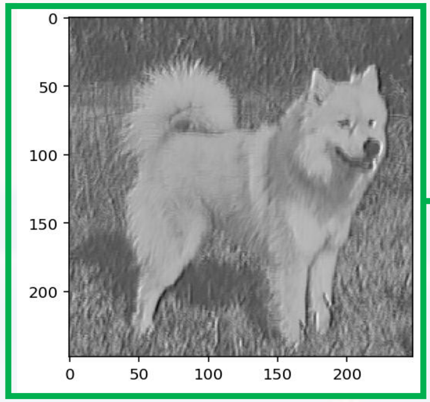
```
import matplotlib.pyplot as plt
from MyCNN import *

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)

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()
```

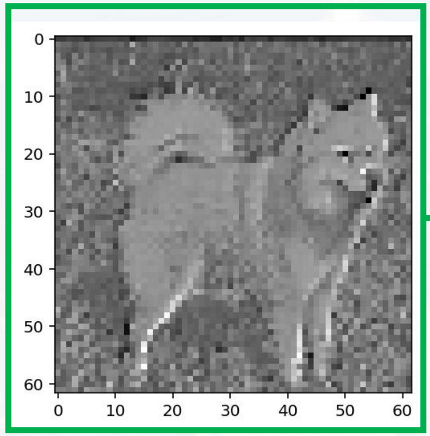
initializing convolution layer (kernel size x kernel size, number of kernels)

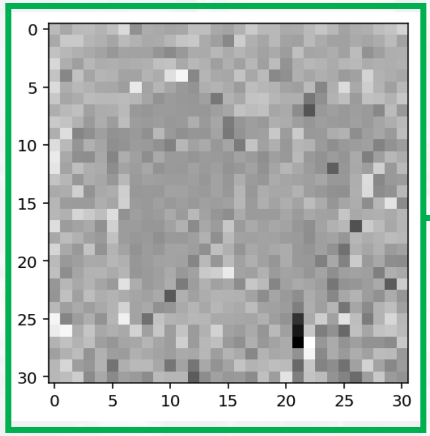
passing image through convolution layer (padding, stride lenght)

Convolution, Image Classification

Forward Part *Spelled Out*








```
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()
```

Convolution, Image Classification

Forward Part *Spelled Out*

see `MyCNN.py`

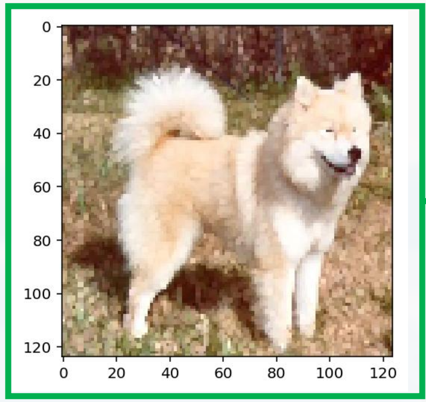
- MyCNN.py
  - Read_Scale_Imds
  - ConvLayer
    - __init__
    - forward
    - backward
  - Average_Pool
  - Min_Pool
  - Max_Pool
  - Sigmoid
  - Tanh
  - Activation_ReLU
  - Flat
  - Layer_Dense
  - Activation_Softmax
  - Loss
  - Loss_CategoricalCrossEntropy
  - CalcSoftmaxLossGrad
  - Optimizer_SGD

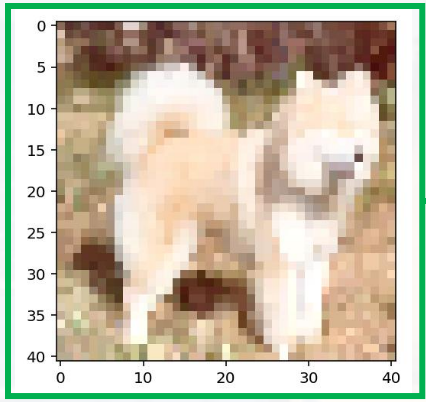
```
y_start = y*stride
y_end   = y_start + yK
x_start = x*stride
x_end   = x_start + xK

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
```

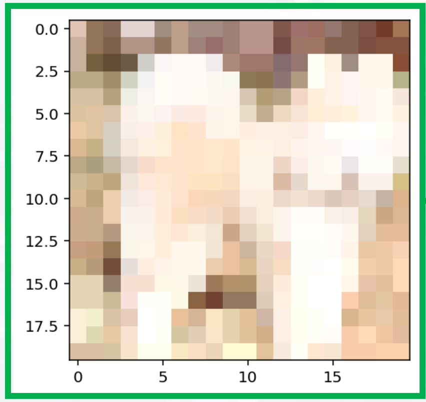
here: max pool

Convolution, Image Classification

Forward Part *Spelled Out*







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

initializing ax pool layer

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

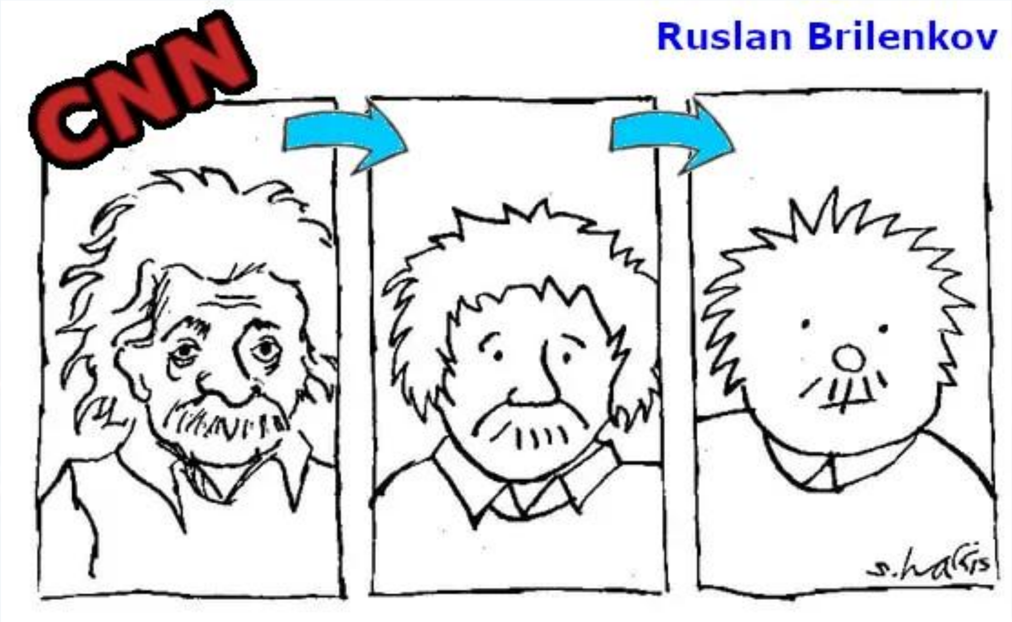
passing image through max pool layer (stride length, kernel size)

```
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()
```

Sequential Classification CNNs - LeNet



Outline:

- What is LeNet?
- Forward Part *Spelled Out*
- **Backward Part *Spelled Out***
- LeNet `Keras` `TensorFlow`

last time for dense layer:

```
class Layer_Dense:

    def __init__(self, n_inputs, n_neurons):
        self.weights = np.random.rand(n_inputs, n_neurons)
        self.biases  = np.zeros((1, n_neurons))

    def forward(self, inputs):

        self.output  = np.dot(inputs, self.weights) + self.biases
        self.inputs  = inputs

    def backward(self, dvalues):

        self.dweights  = np.dot(self.inputs.T, dvalues)
        self.dinputs   = np.dot(dvalues, self.weights.T)
        self.dbiases   = np.sum(dvalues, axis = 0, keepdims = True)
```

outer derivative

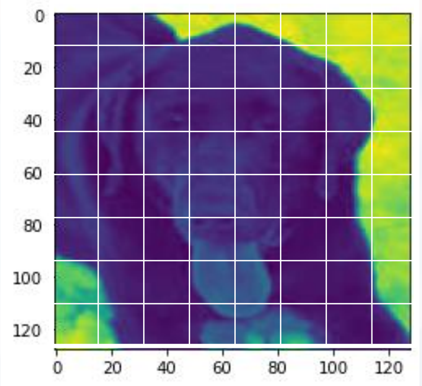
Convolution, Image Classification

Backward Part *Spelled Out*

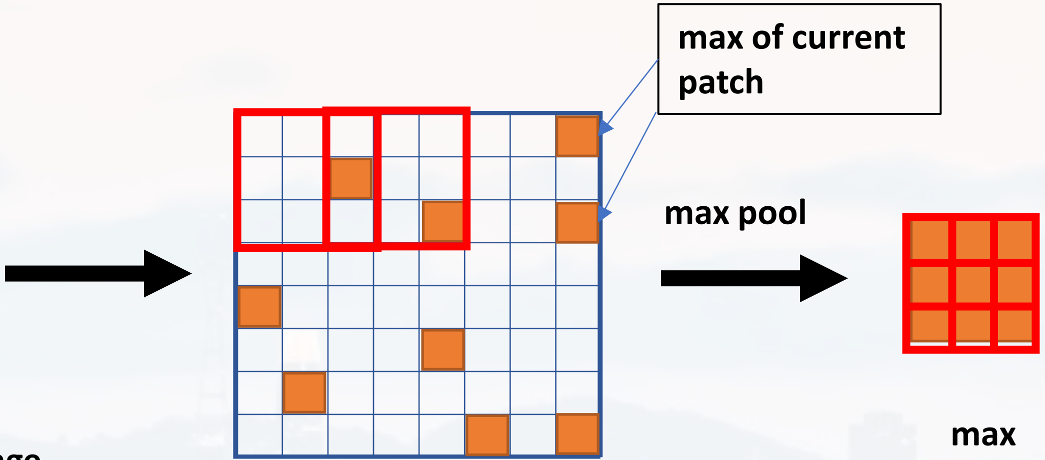
max pool

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

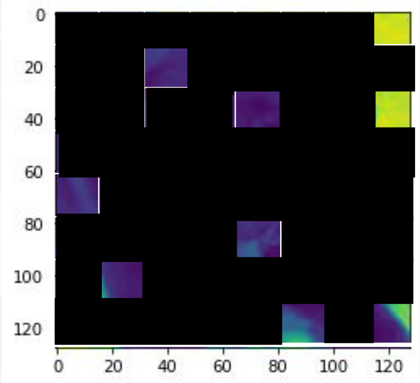
forward:



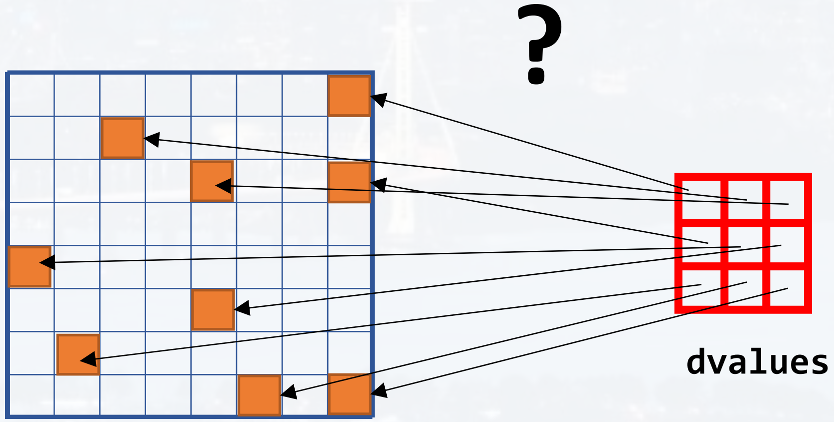
image



backward:



dinputs



We need **to track, where the max came from** for each patch: creating a **mask** in the forward part

For those pixel: `dinputs = dvalues`

max pool

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

Convolution, Image Classification

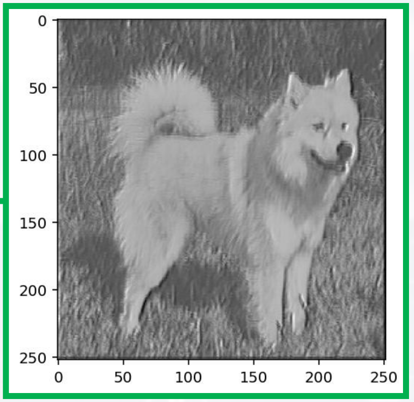
Backward Part *Spelled Out*

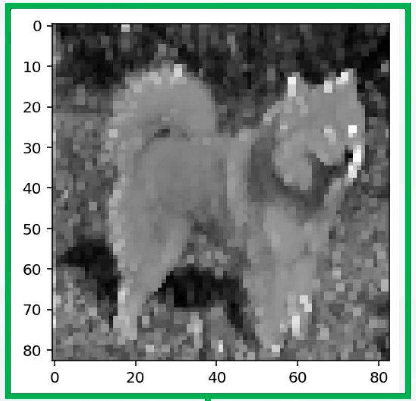
max pool

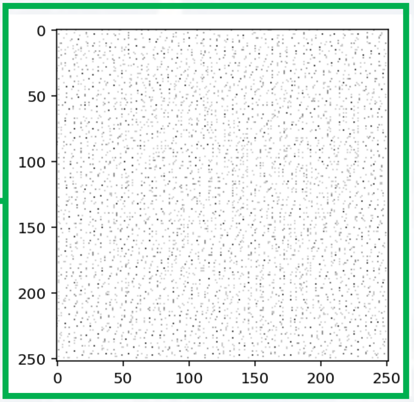
for **backpropagation**, we need to **up-sample** the images!

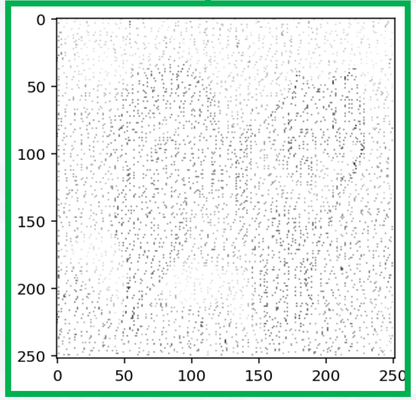
dinputs and mask have to look the same (but different values)!

```
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])
```









Convolution, Image Classification

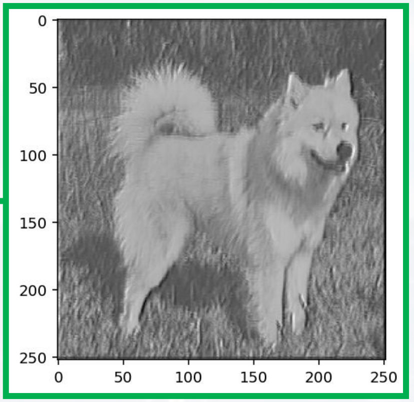
max pool

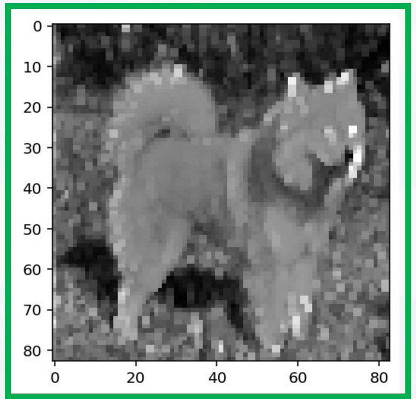
for **backpropagation**, we need to **up-sample** the images!

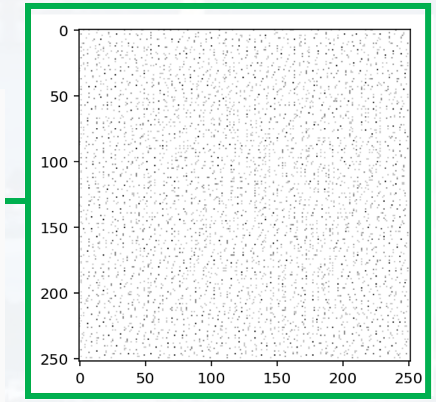
dinputs and mask have to look the same (but different values)!

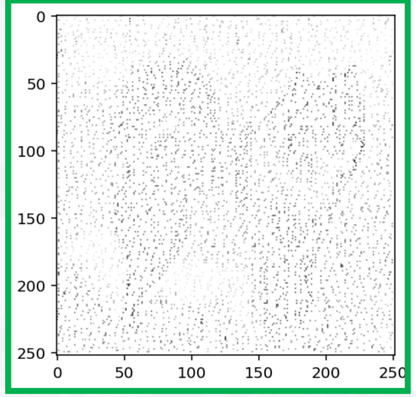
```
Conv1.forward(I,2,1)
MP1.forward(Conv1.output,3,5)
MP1.backward(MP1.output)

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]/
       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.]])
```









Convolution, Image Classification

Backward Part *Spelled Out*

convolution

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!)

forward:

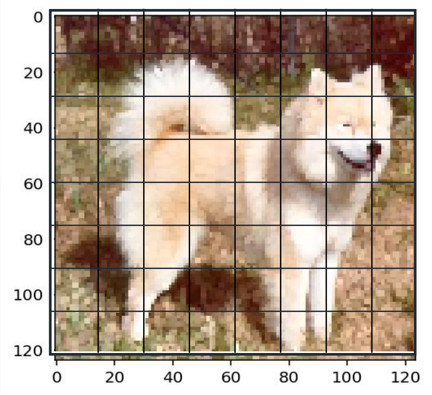
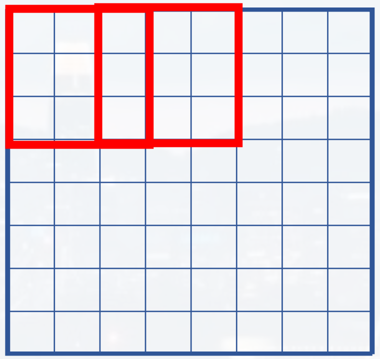
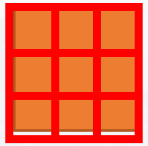


image = inputs

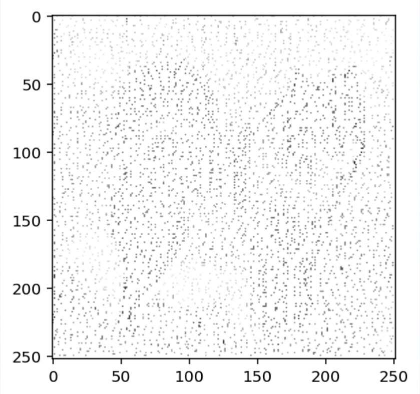


$$\sum_i I_i w_i + b$$

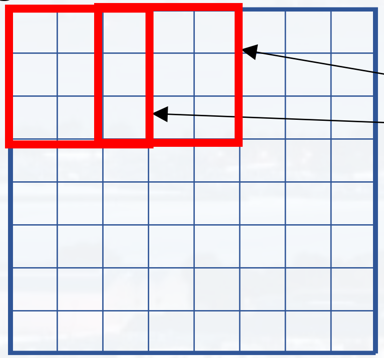


convolved image

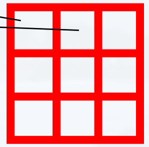
backward:



`weights`



`dinputs`



`dvalues`

Convolution, Image Classification

convolution

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!)

forward:

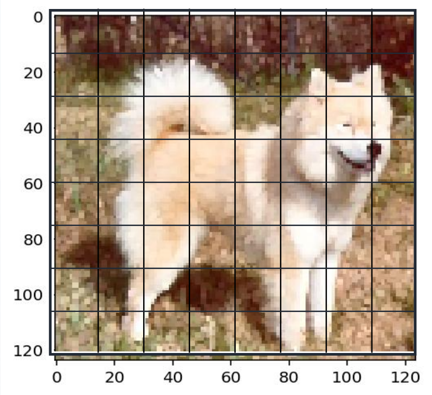
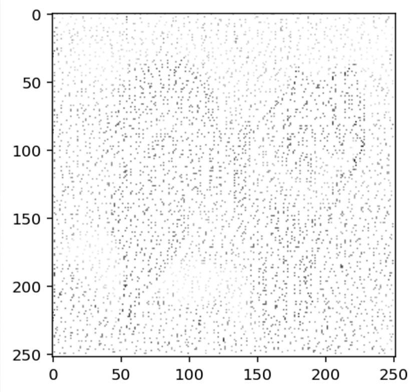


image = inpu

backward:




```
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 incomming 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)
```

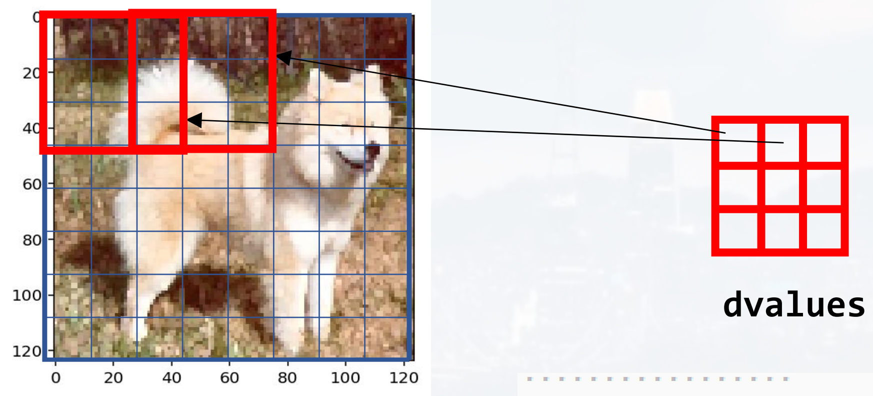
convolution

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

same for convolution layer now:

backward `dweights = dvalues * inputs`

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)
```

Convolution, Image Classification

convolution

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)
```

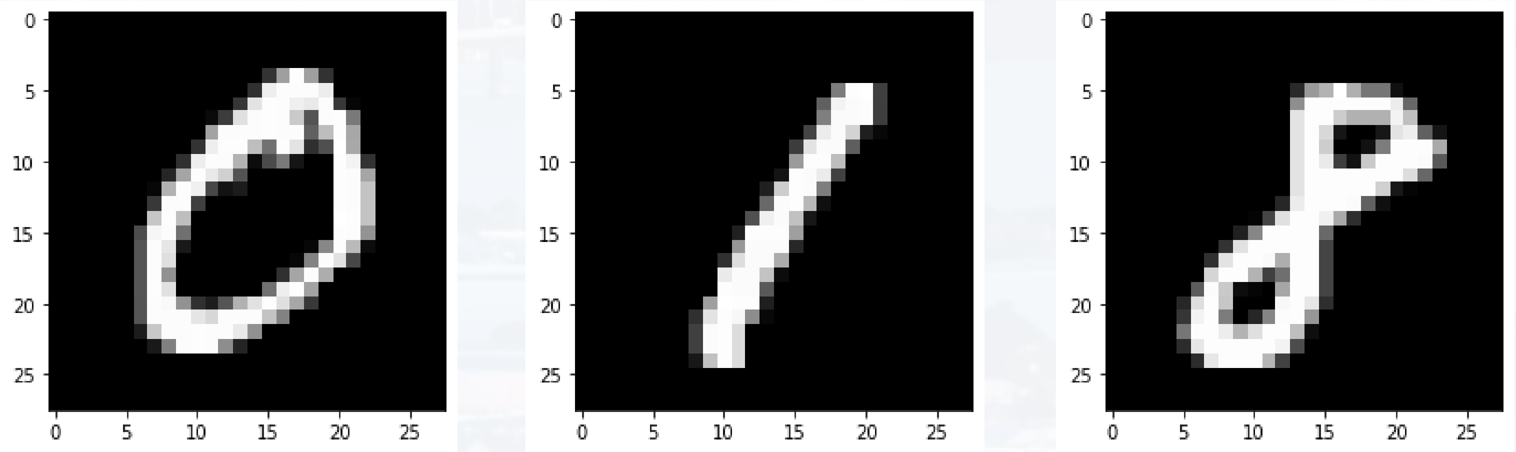
Convolution, Image Classification

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

- MyCNN.py
  - Read_Scale_Imds
  - ConvLayer
    - __init__
    - forward
    - backward
  - Average_Pool
  - Min_Pool
  - Max_Pool
  - Sigmoid
  - Tanh
  - Activation_ReLU
  - Flat
  - Layer_Dense
  - Activation_Softmax
  - Loss
  - Loss_CategoricalCrossEntropy
  - CalcSoftmaxLossGrad
  - Optimizer_SGD

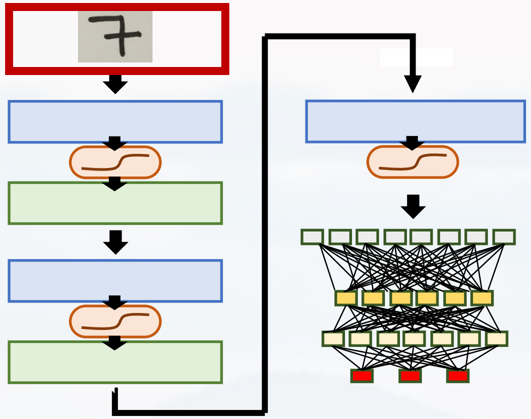
```
#pip install keras
#pip install tensorflow

from keras.datasets import mnist

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

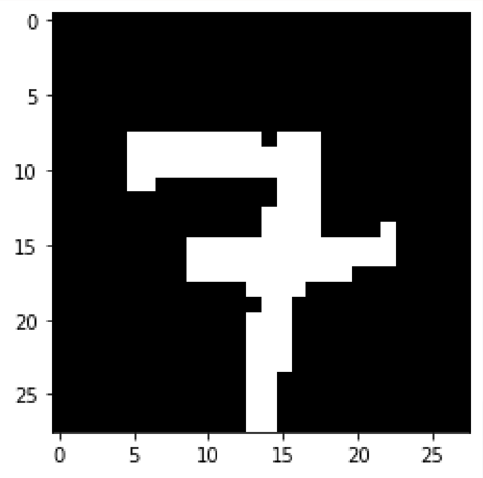
```
plt.imshow(train_x[1,:,:], cmap = 'gray')
```



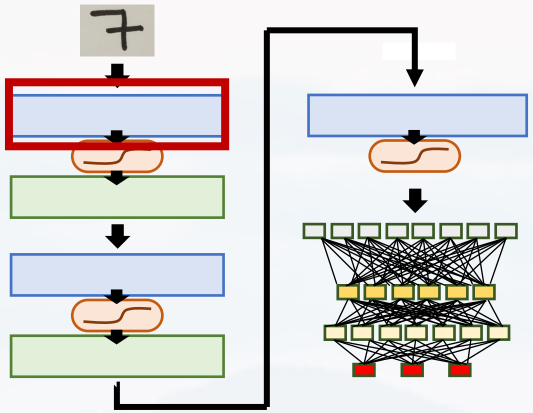
Convolution, Image Classification

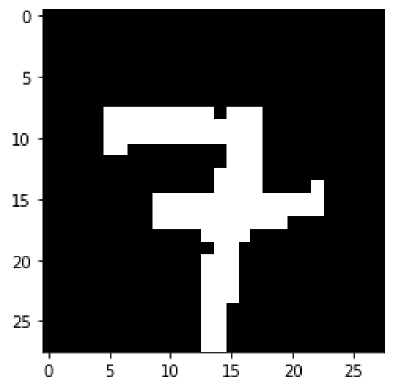
exploring self-made LeNet:

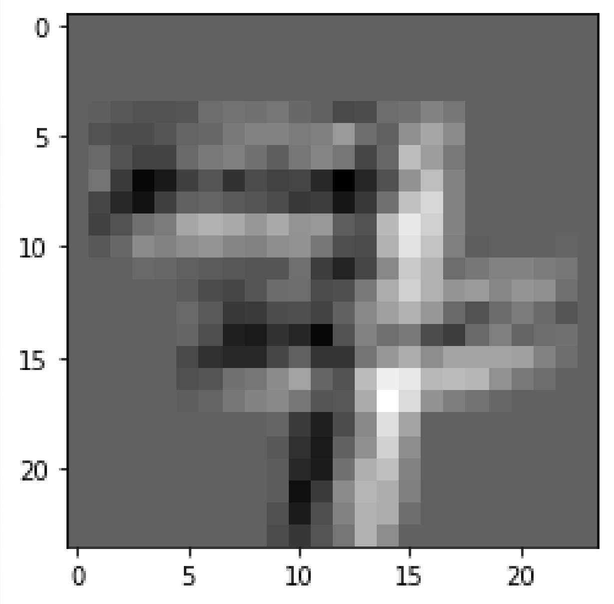




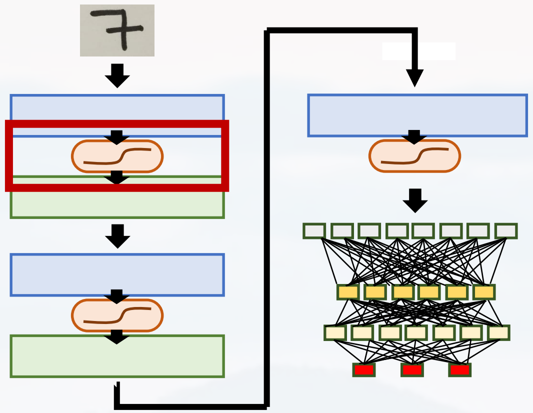
exploring self-made LeNet:

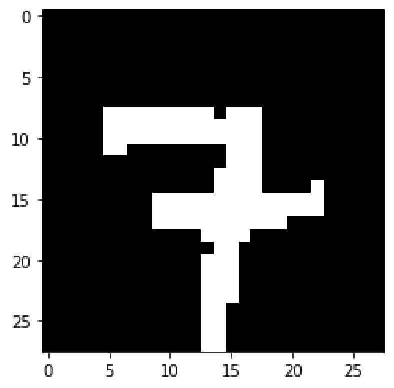


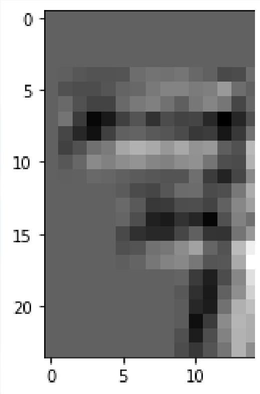


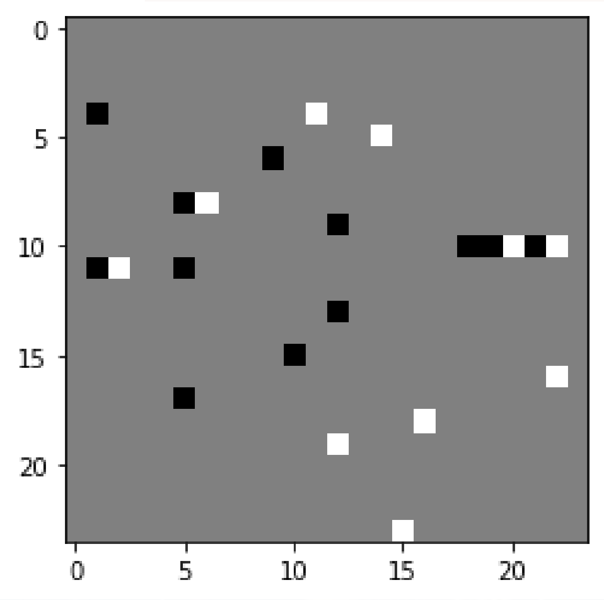


exploring self-made LeNet:

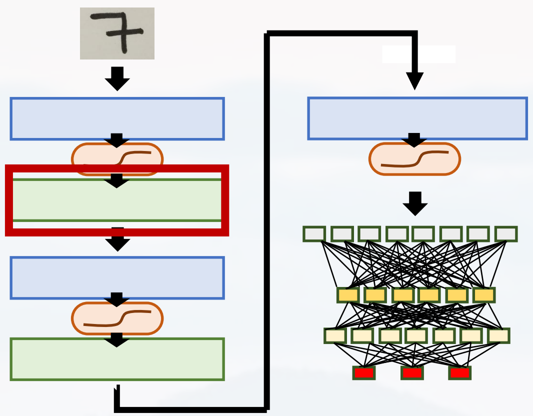


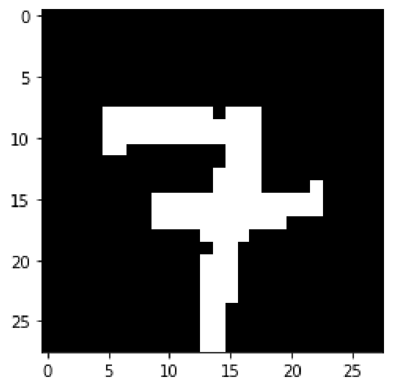


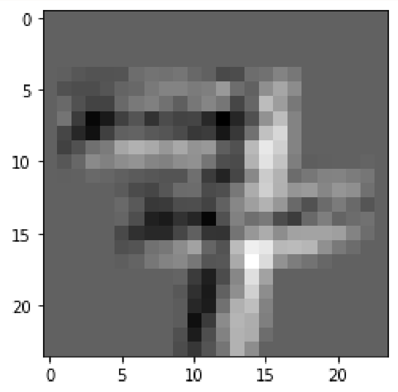


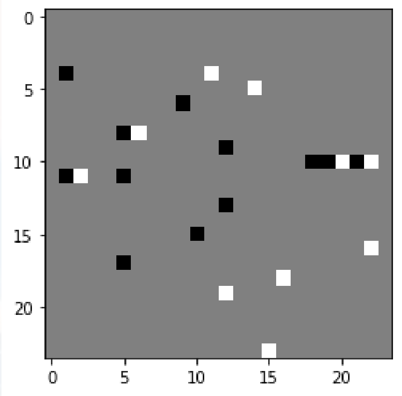


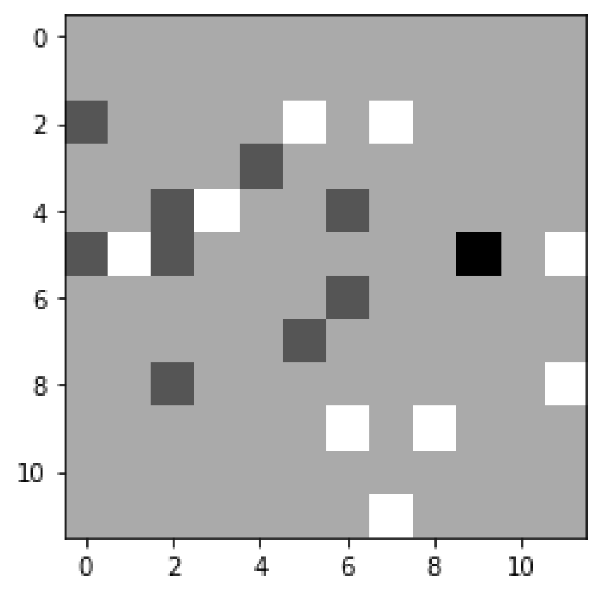
exploring self-made LeNet:





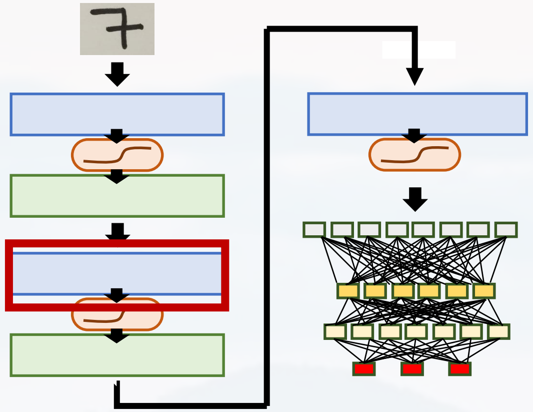


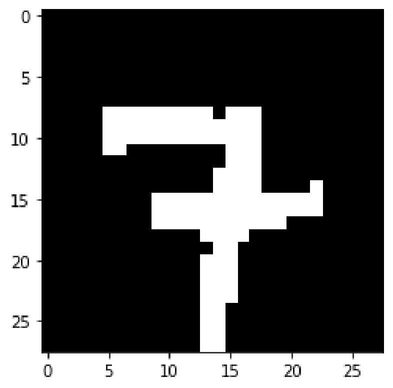


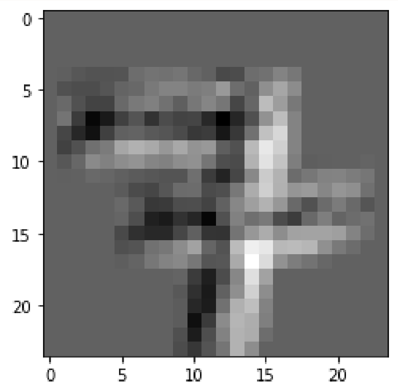


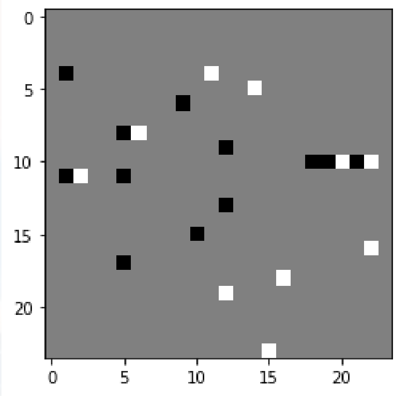
Convolution, Image Classification

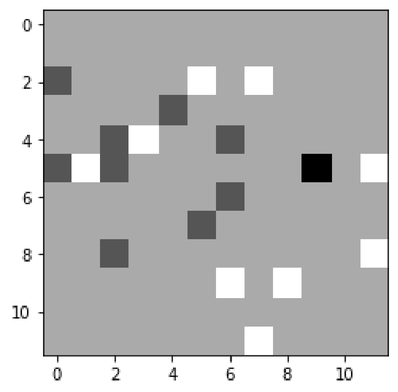
exploring self-made LeNet:

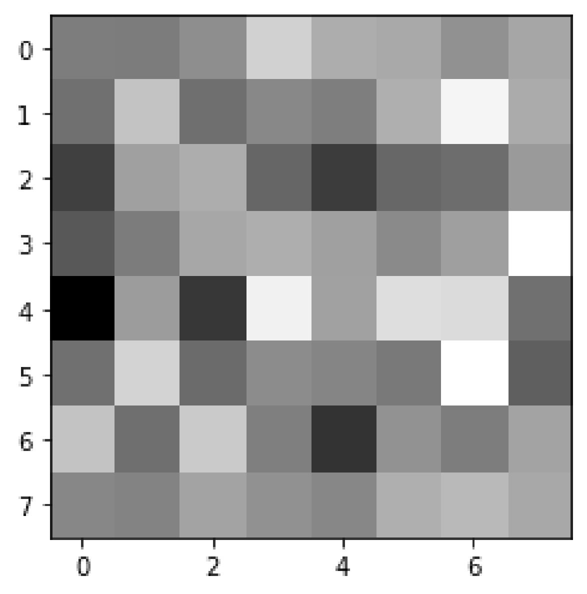




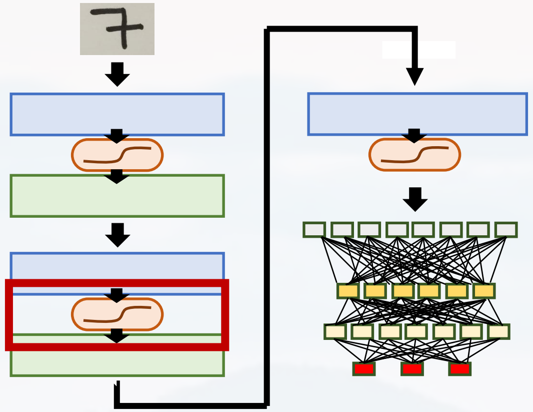


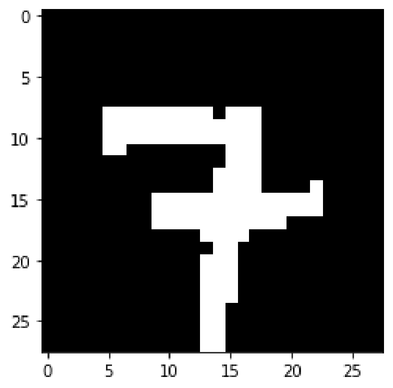


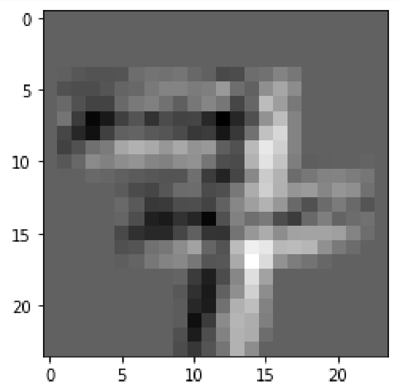


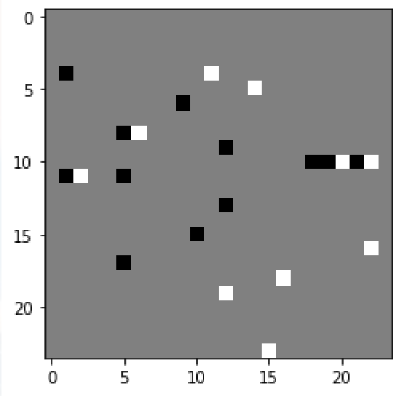


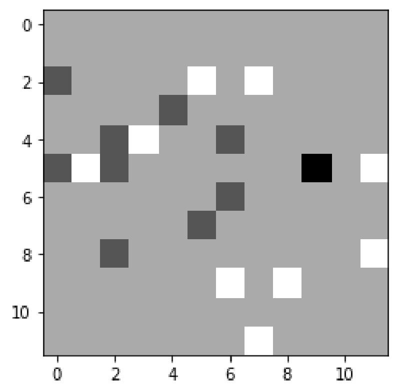
exploring self-made LeNet:

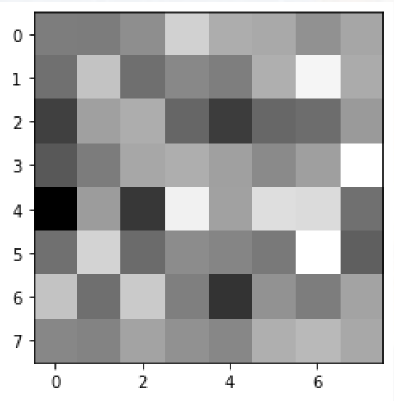


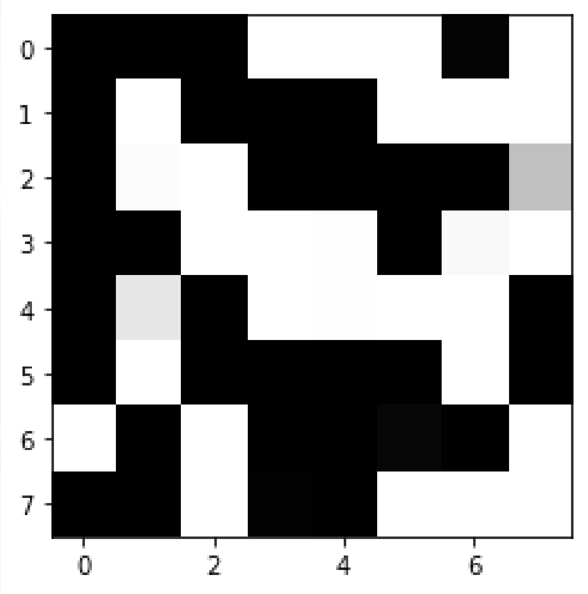








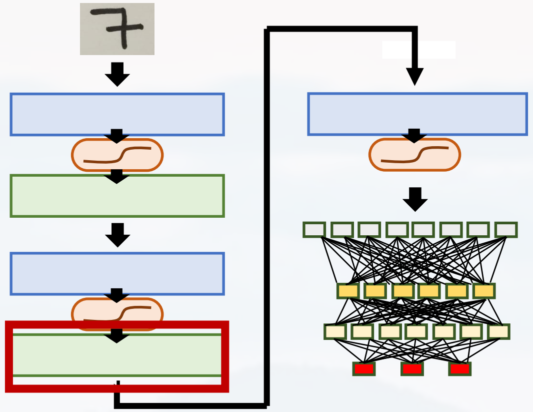


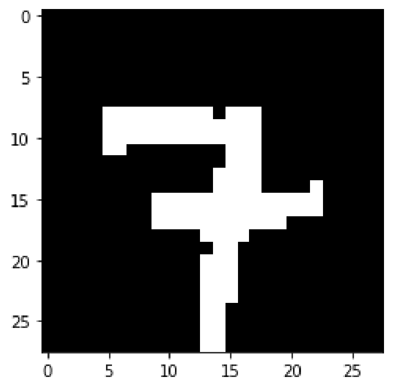


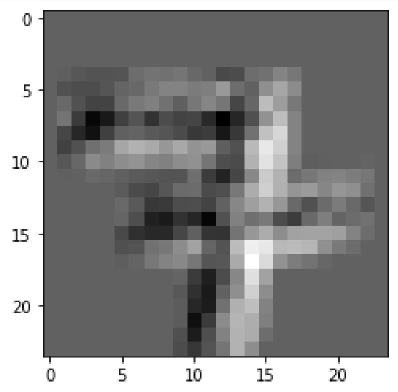
Convolution, Image Classification

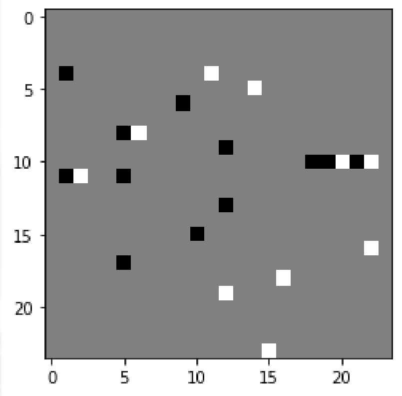
Backward Part *Spelled Out*

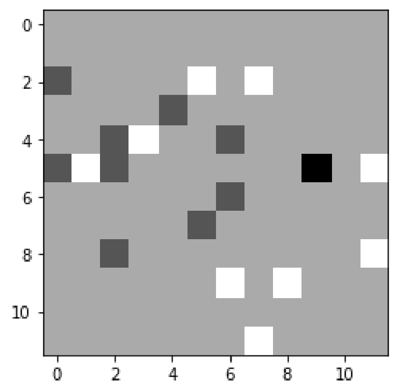
exploring self-made LeNet:

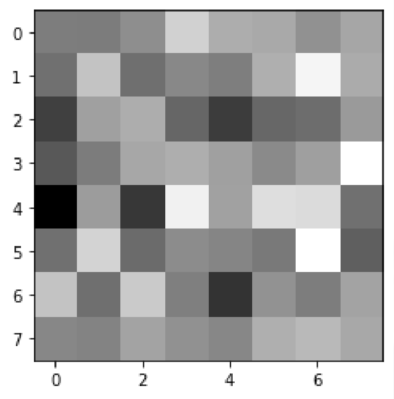


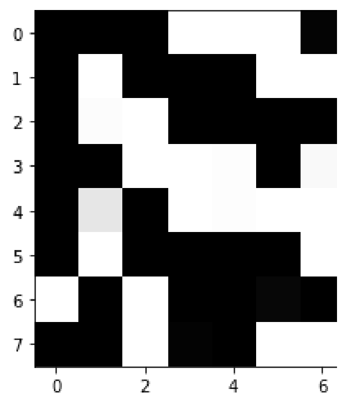


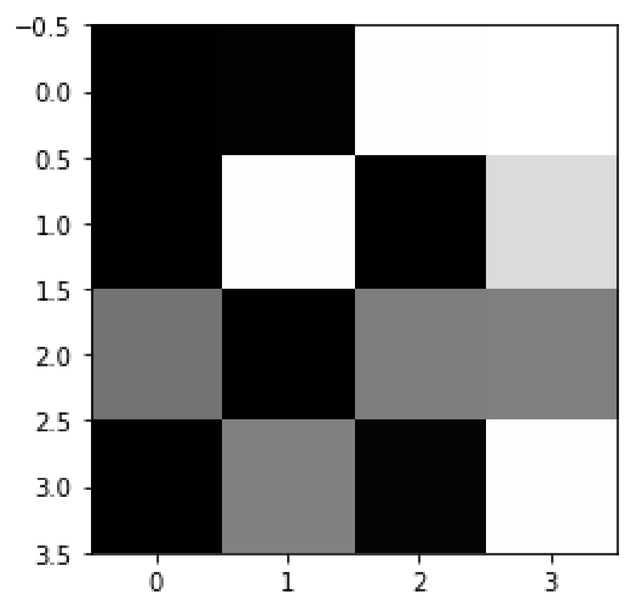








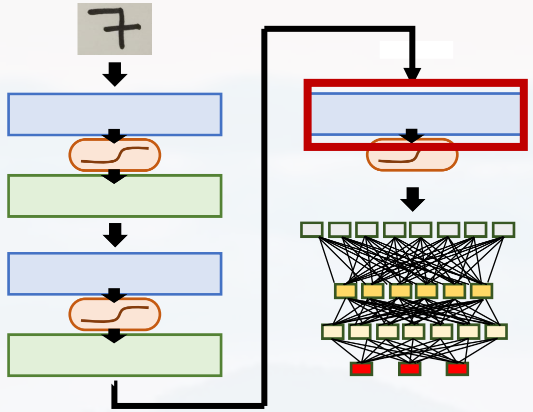


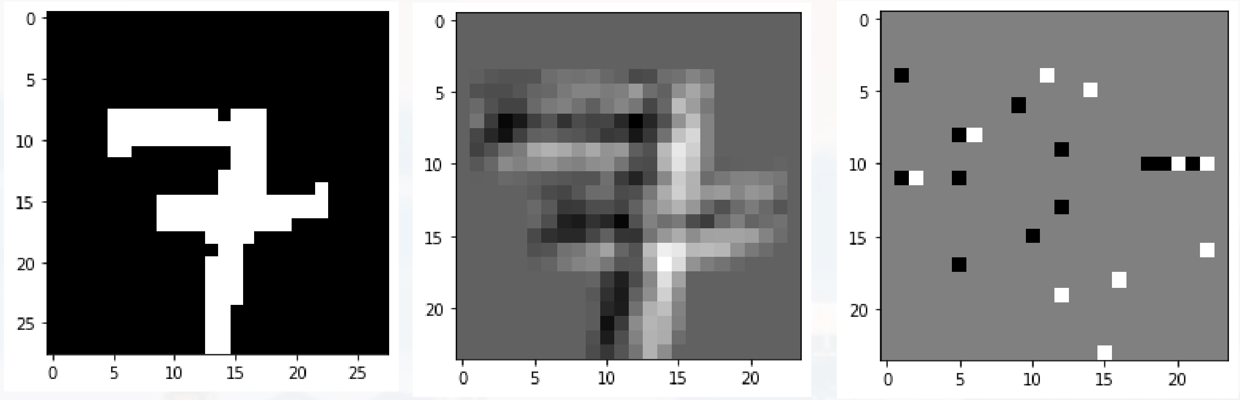


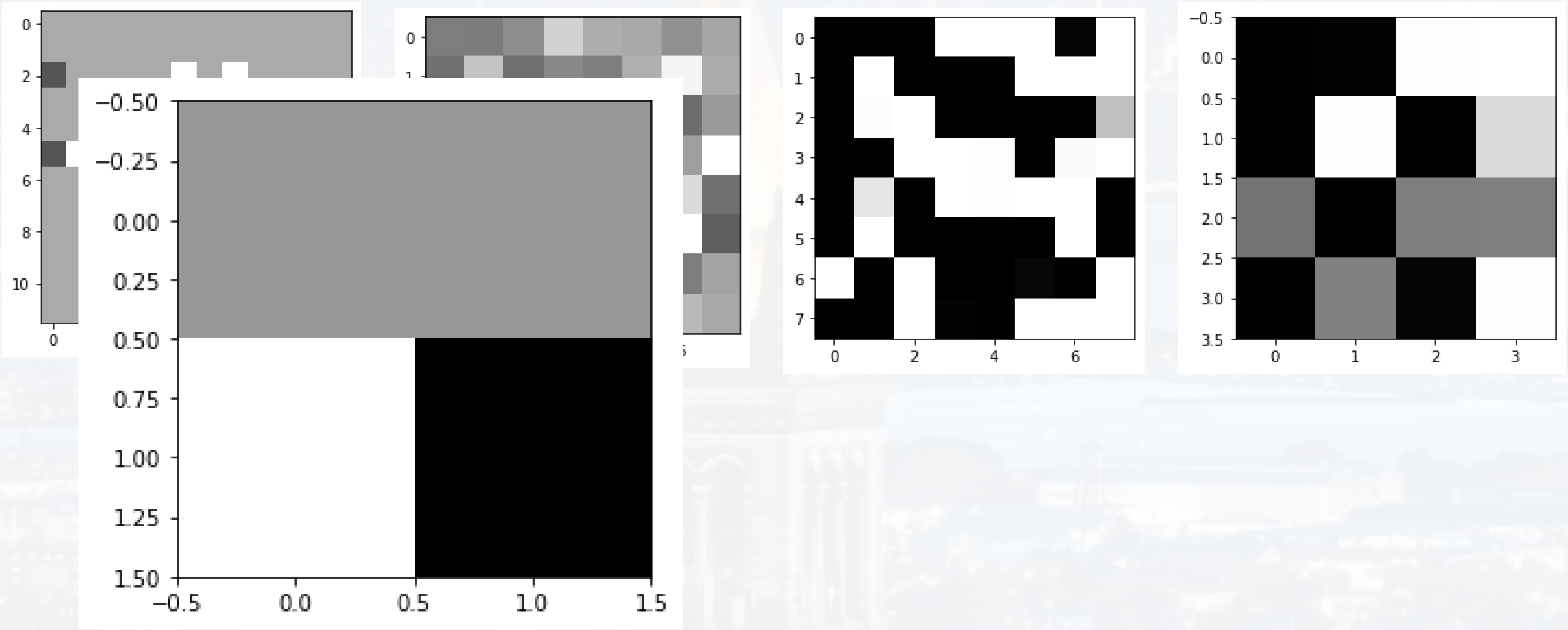
Convolution, Image Classification

Backward Part *Spelled Out*

exploring self-made LeNet:



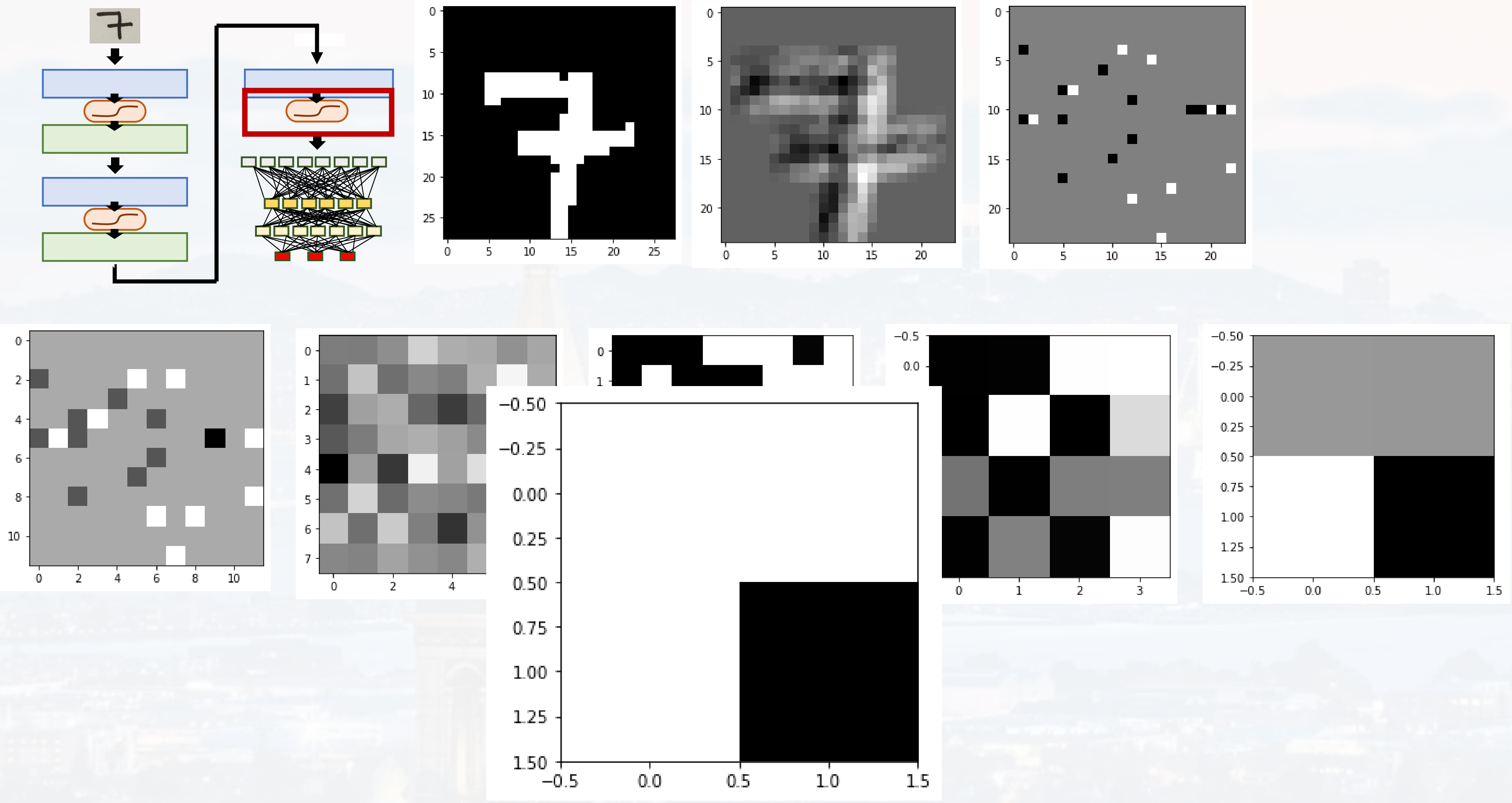




Convolution, Image Classification

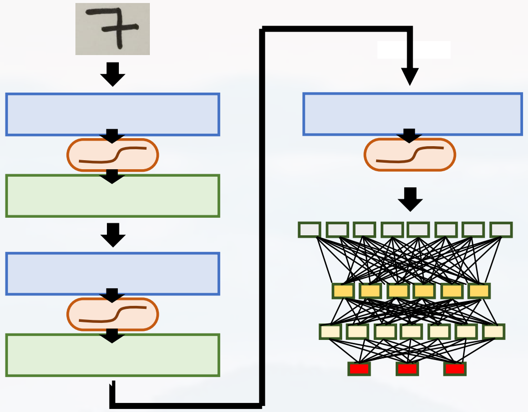
Backward Part *Spelled Out*

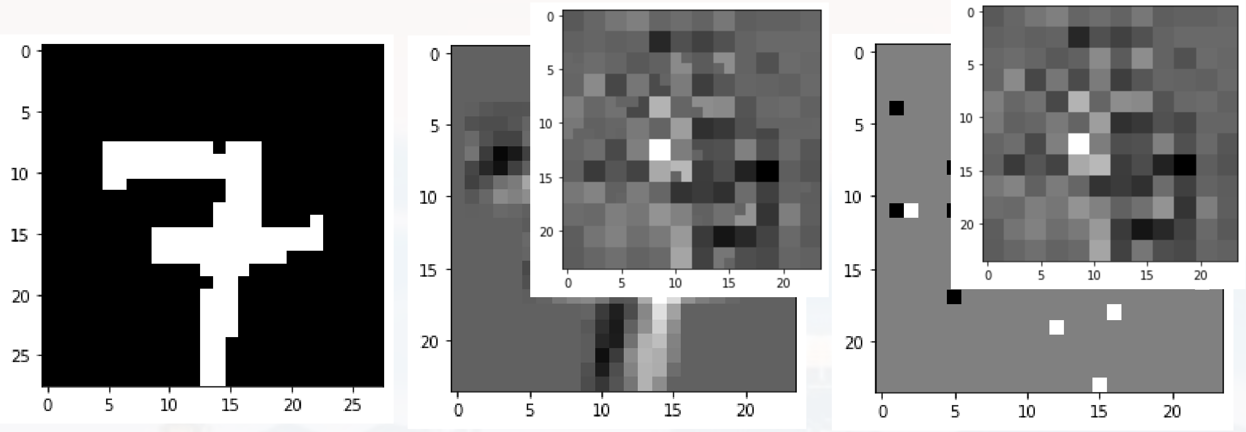
exploring self-made LeNet:



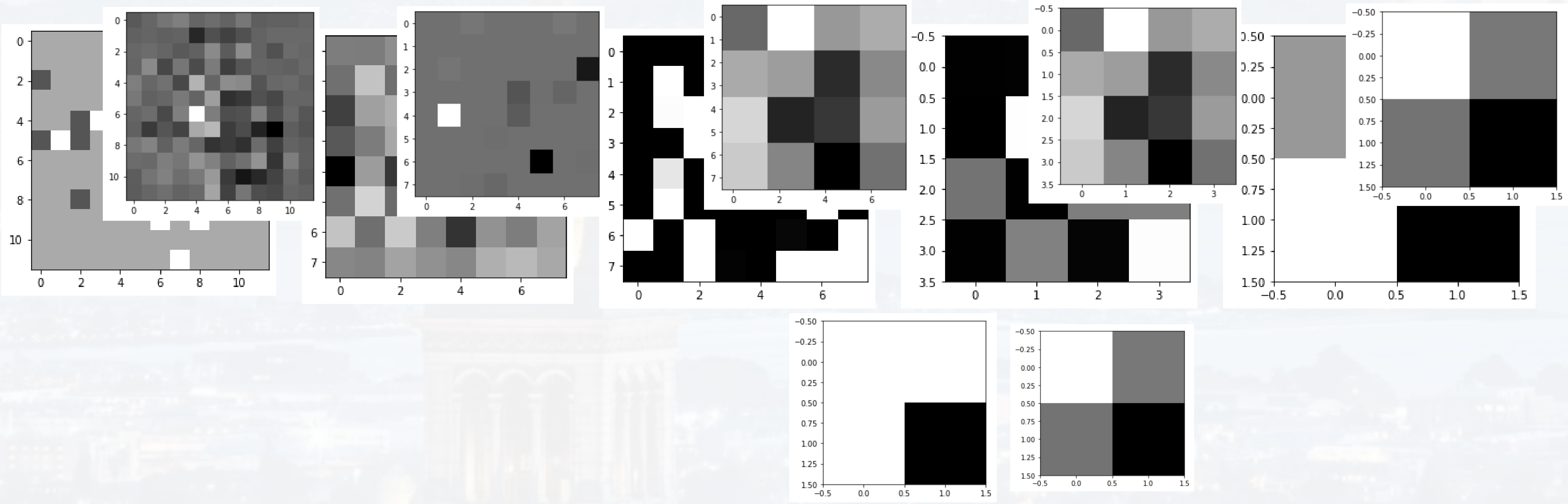
Convolution, Image Classification

exploring self-made LeNet:





now back propagation!



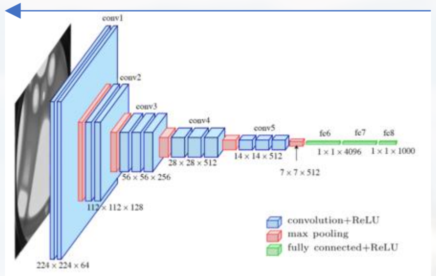
Convolution, Image Classification

Backward Part *Spelled Out*

CNNs have an “hourglass” structure

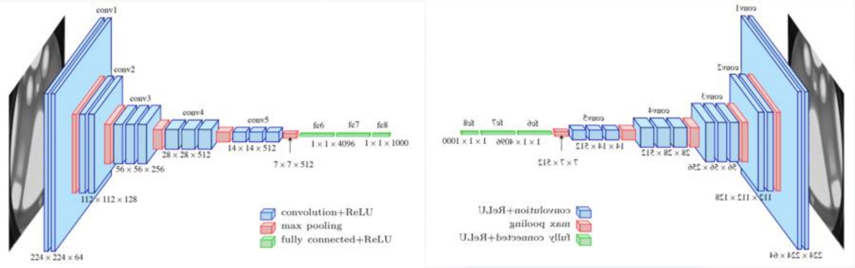
classification:

backward (up-sampling)



forward (down-sampling)

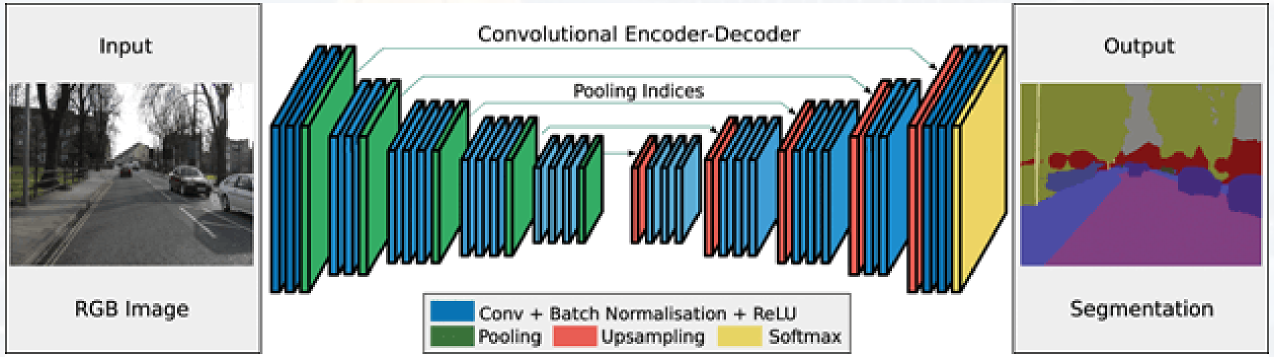
backward (up-sampling)



forward (down-sampling)

segmentation:

backward (up-sampling) *backward (down-sampling)*



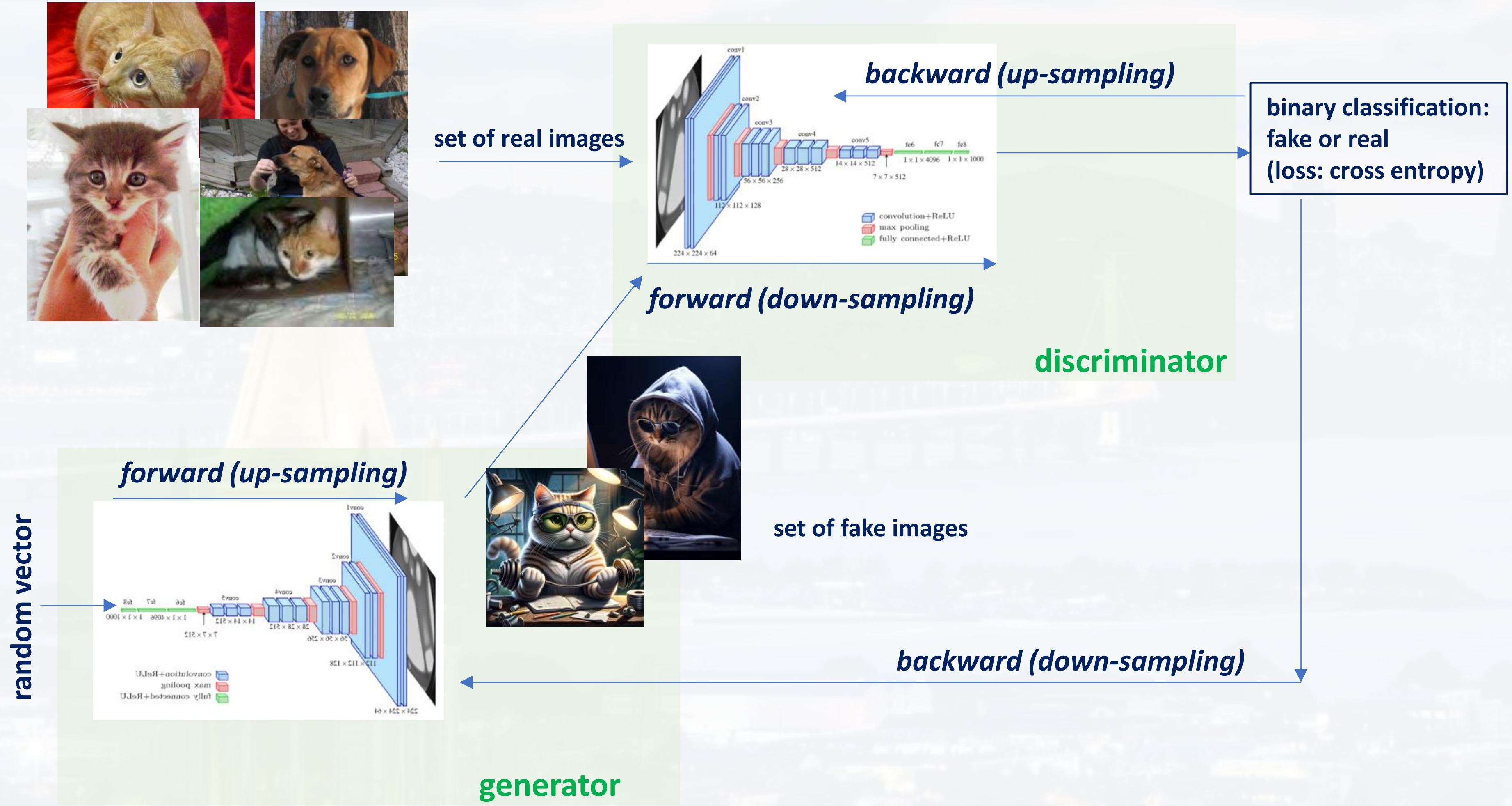
forward (down-sampling) *forward (up-sampling)*

Convolution, Image Classification

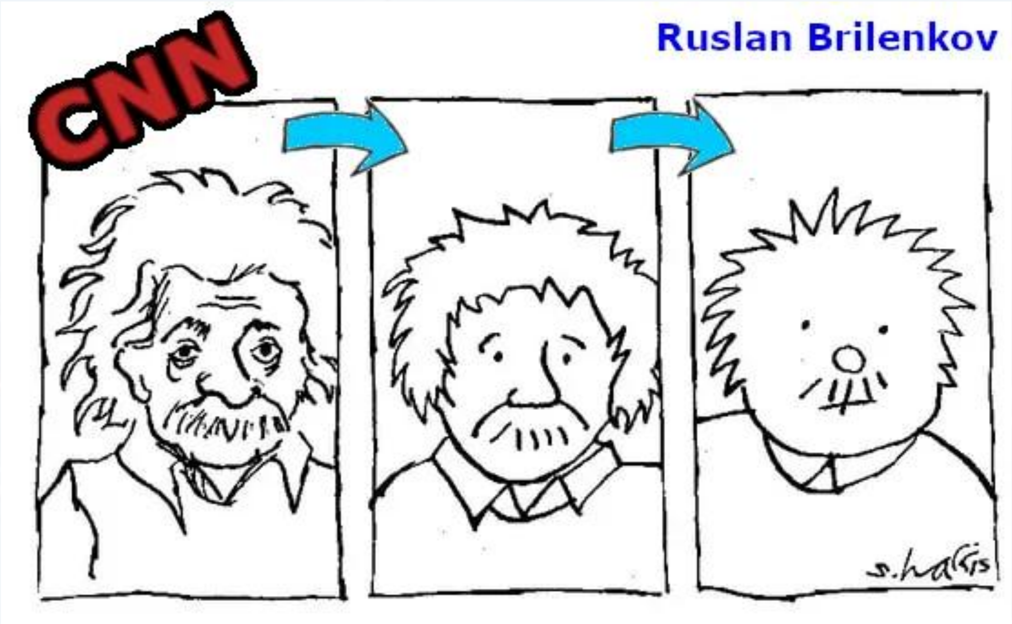
Backward Part *Spelled Out*

CNNs have an "hourglass" structure

Generative **A**dversarial **N**etwork (GAN):



Sequential Classification CNNs - LeNet



Outline:

- What is LeNet?
- Forward Part *Spelled Out*
- Backward Part *Spelled Out*
- **LeNet** `Keras TensorFlow`

Convolution, Image Classification

LeNet Keras TensorFlow

see LeNetTF.py and LeNetTF.ipynb

```
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'])
```

Convolution, Image Classification

see LeNetTF.py and LeNetTF.ipynb

```
Model: "my_le_net_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 conv2d_3 (Conv2D)           (None, 28, 28, 6)         156

 average_pooling2d_2 (Avera  (None, 14, 14, 6)         0
 gePooling2D)

 conv2d_4 (Conv2D)           (None, 10, 10, 16)        2416

 average_pooling2d_3 (Avera  (None, 5, 5, 16)          0
 gePooling2D)

 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

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

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

Convolution, Image Classification

see LeNetTF.py and LeNetTF.ipynb

```
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

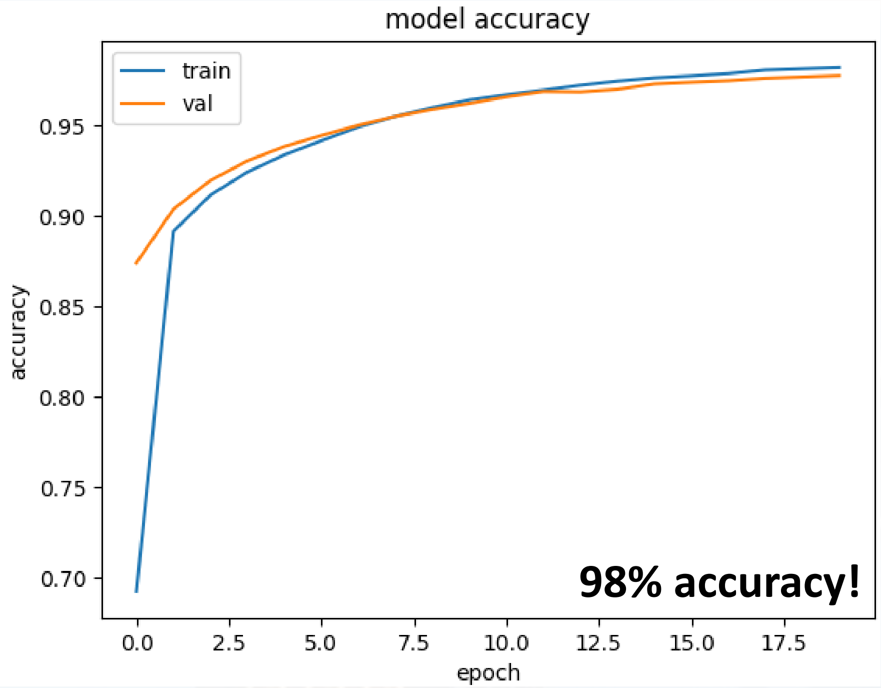
60, 000 images / batch size = **512** = 117 iterations per epoch

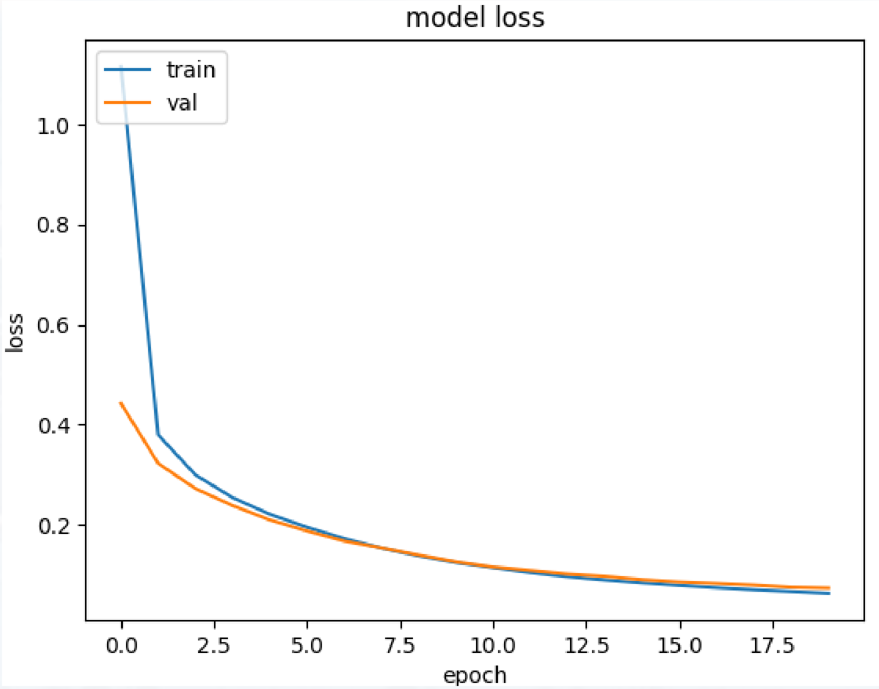
= **117 * 80**% for training = **94 iterations per epoch**

Convolution, Image Classification

LeNet `Keras` `TensorFlow`

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

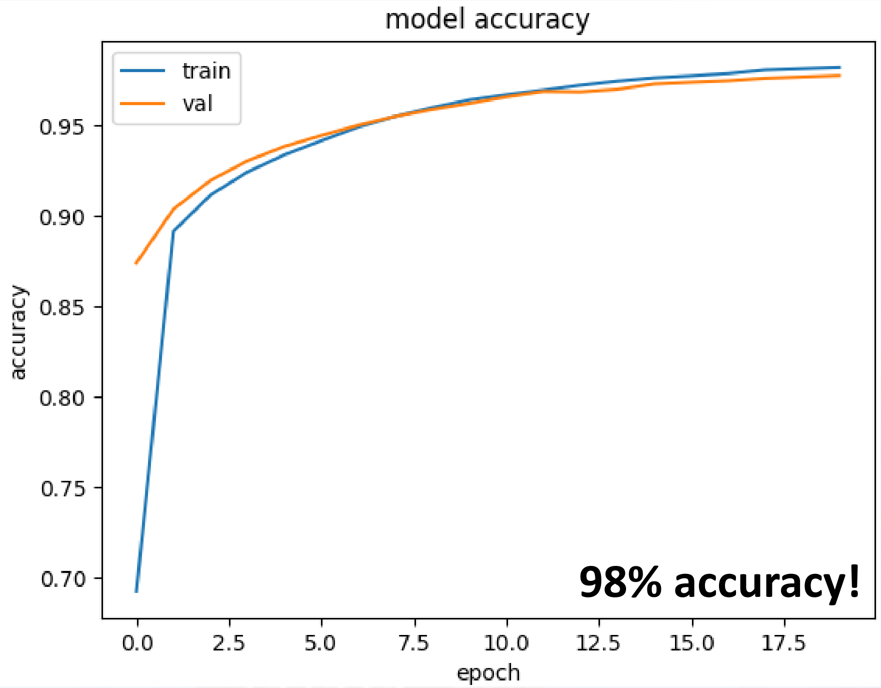


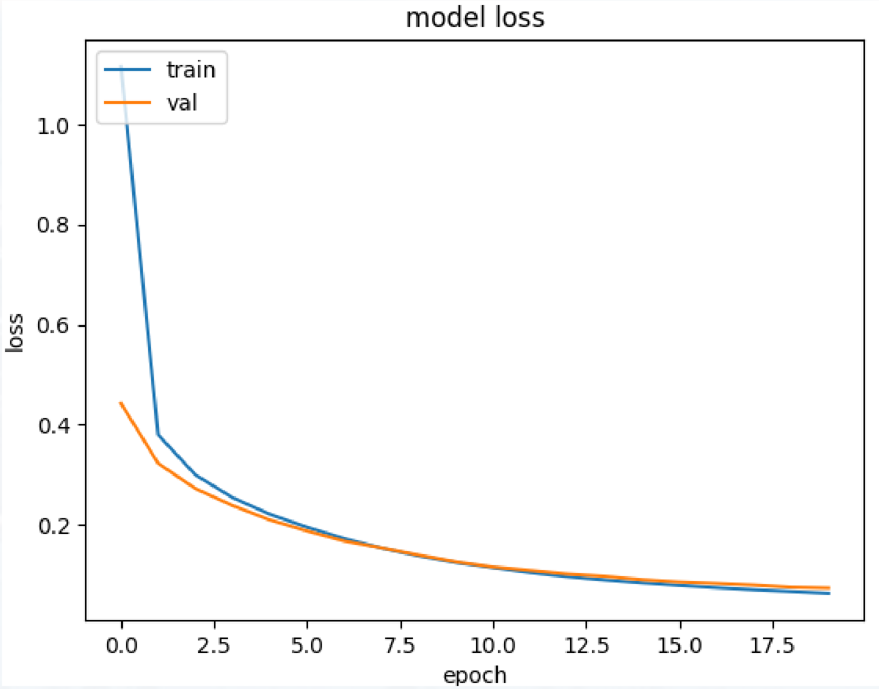


Convolution, Image Classification

LeNet `Keras` `TensorFlow`

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





training loss should ≈ validation loss
if validation loss >> training loss → overfitting
- too many parameter
- too few images in batch
- too specific/unique batch)

Thank you very much for your attention!