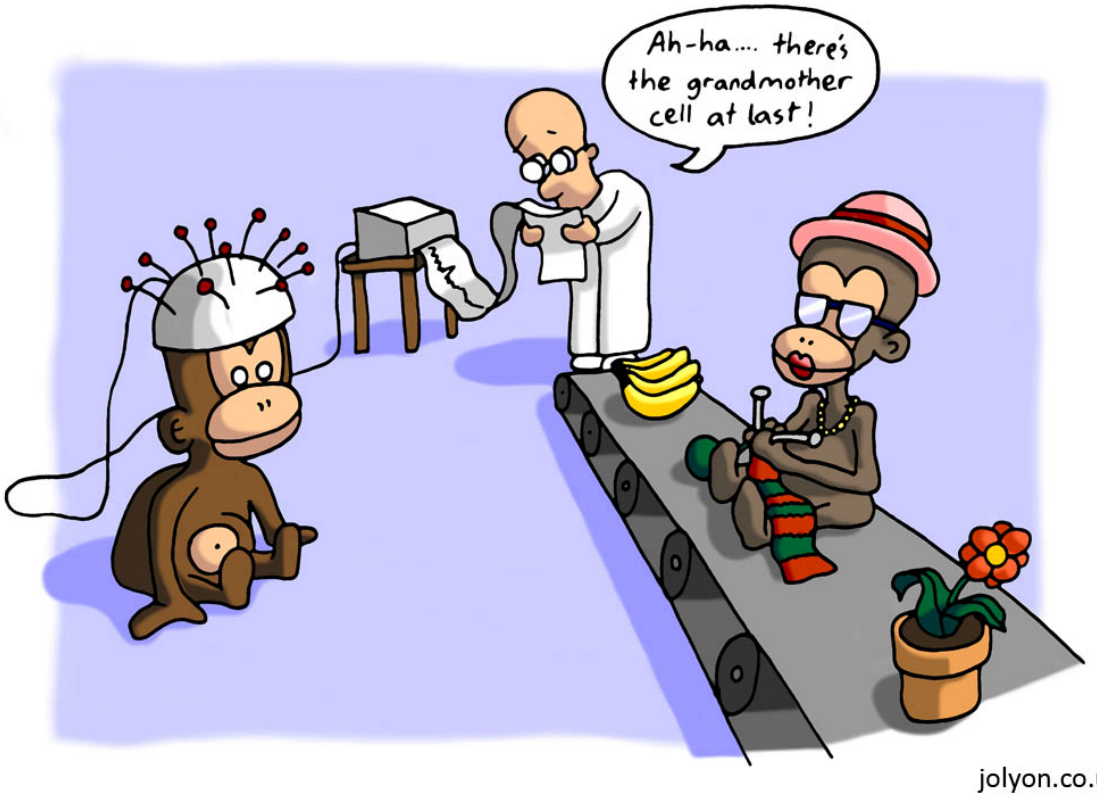
CS109 – Data Science
Deep Learning II - CNNs

Hanspeter Pfister, Mark Glickman, Verena Kaynig-Fittkau

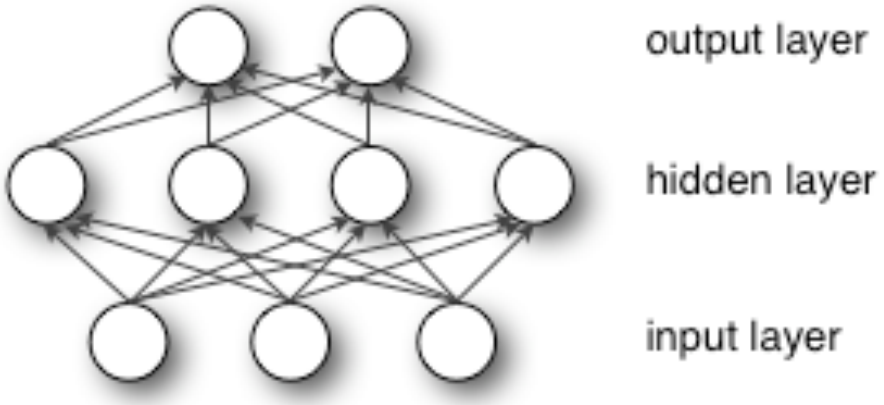


https://computervisionblog.wordpress.com/tag/convolutional-neural-network/

To Train a Simple Network We Need:

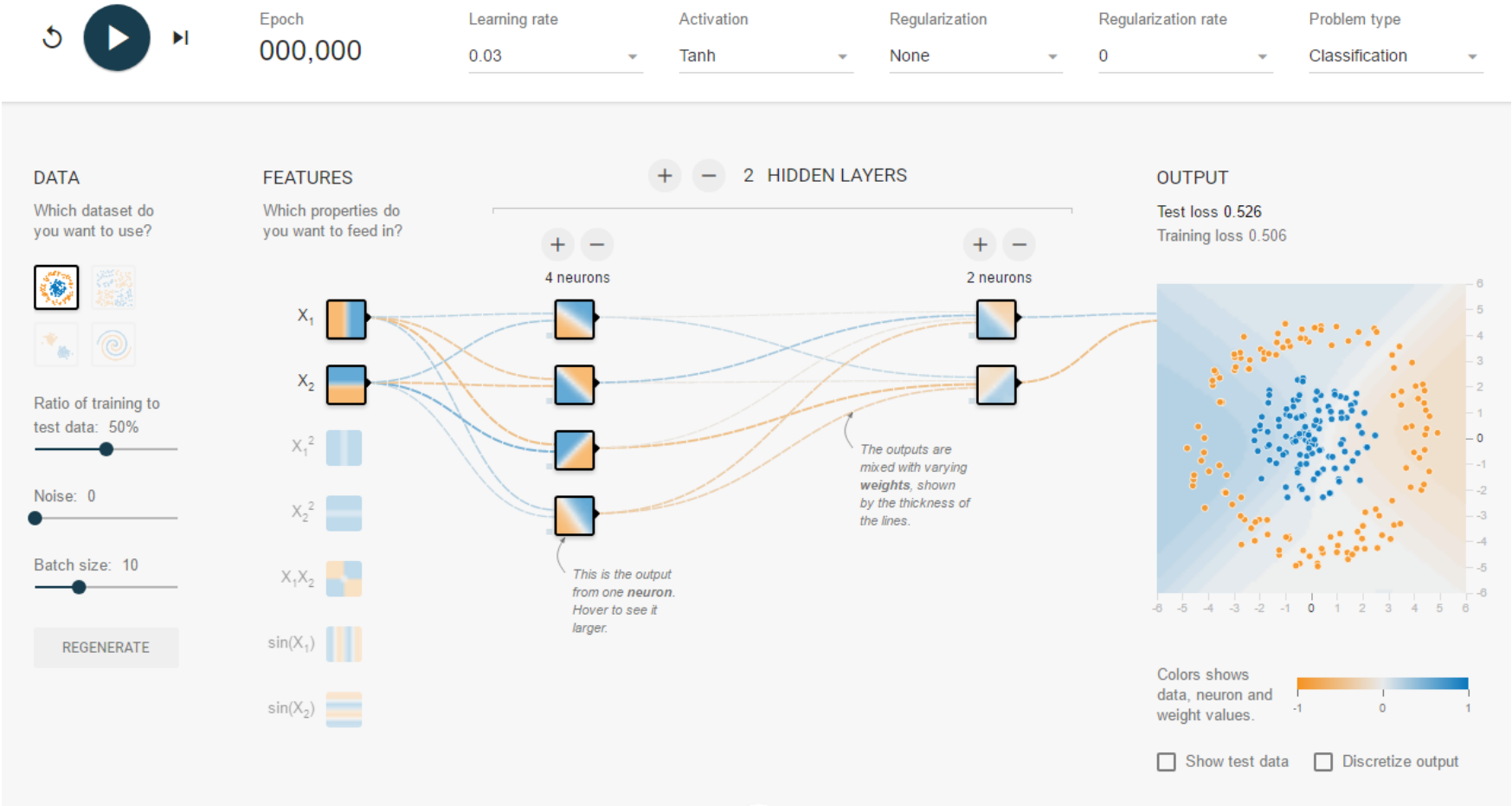
- Input layer size
- Number of hidden layers
- Sizes of hidden layers
- Activation function
- Number of output units

- Loss function
- Optimization method



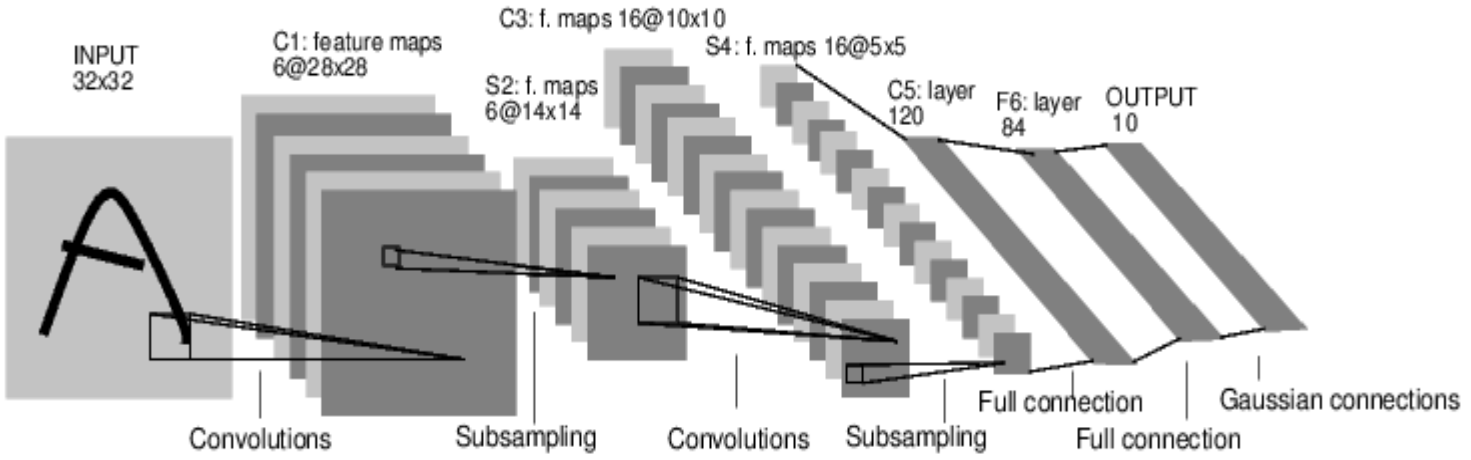
Nice demo!

- http://playground.tensorflow.org



Convolutional Neural Networks

- similar to MLPs
- explicitly developed for images



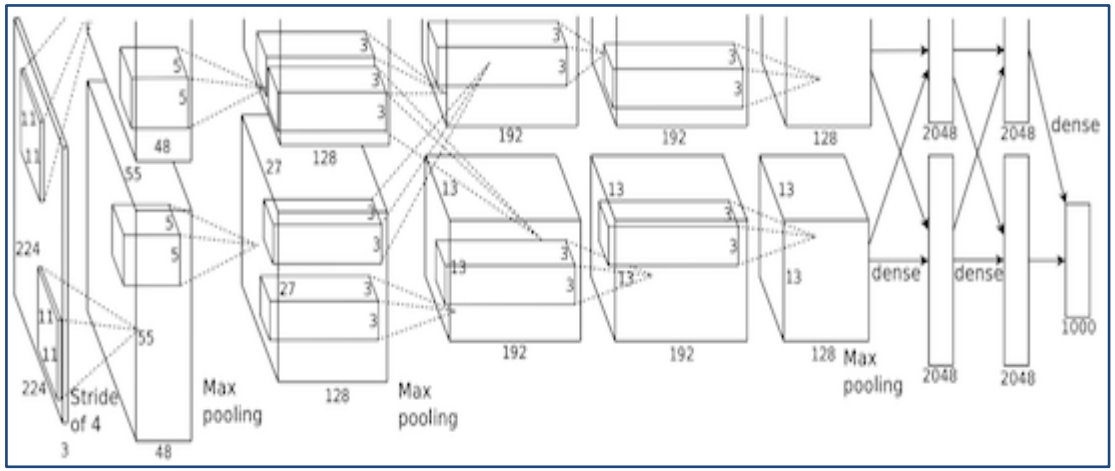
[LeNet-5, LeCun 1980]

A bit of history:

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]

IMAGENET



"AlexNet"

Why MLPs are not Ideal for Image Processing

- Doesn’t scale well to large images

- Pixels are correlated
- MLP doesn’t have knowledge about neighborhood relationships
- Could arbitrarily permute the input (as long as it is consistent)

Scaling problem

- CIFAR-10 images are 32x32x3
  - perceptron: 32x32x3 = 3072 weights
- More interesting images: 200x200x3
  - perceptron: 200x200x3 = 120,000 weights

- We would certainly want more than one perceptron in our model!
- Hard to train a model with that many parameters

Images are Structured

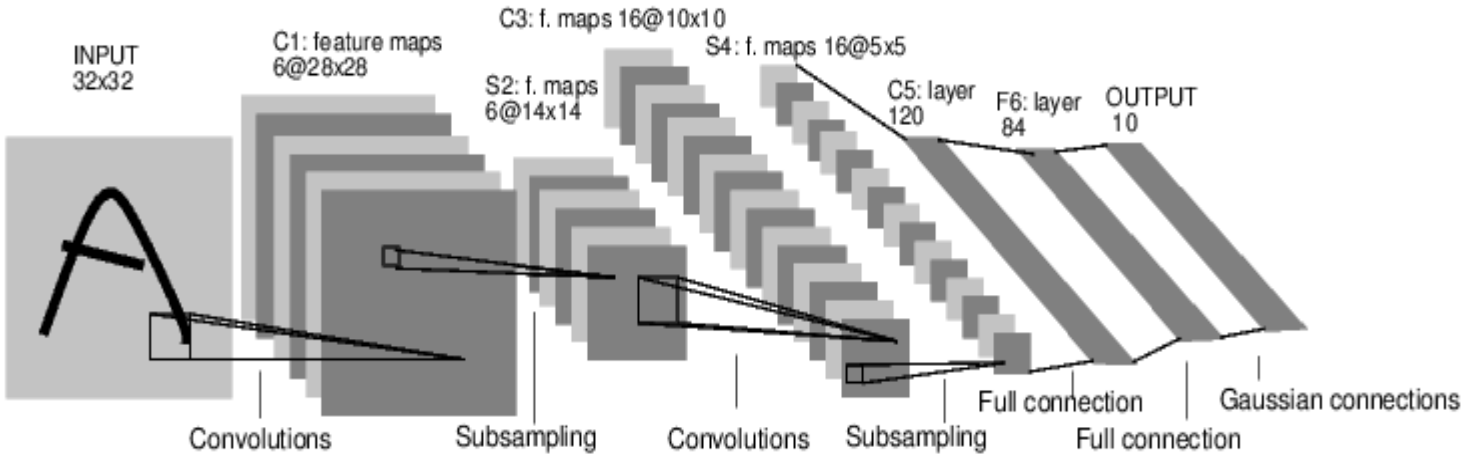
Nearby pixels are more strongly related than distant ones.

Objects are built up out of smaller parts.

Images are Invariant

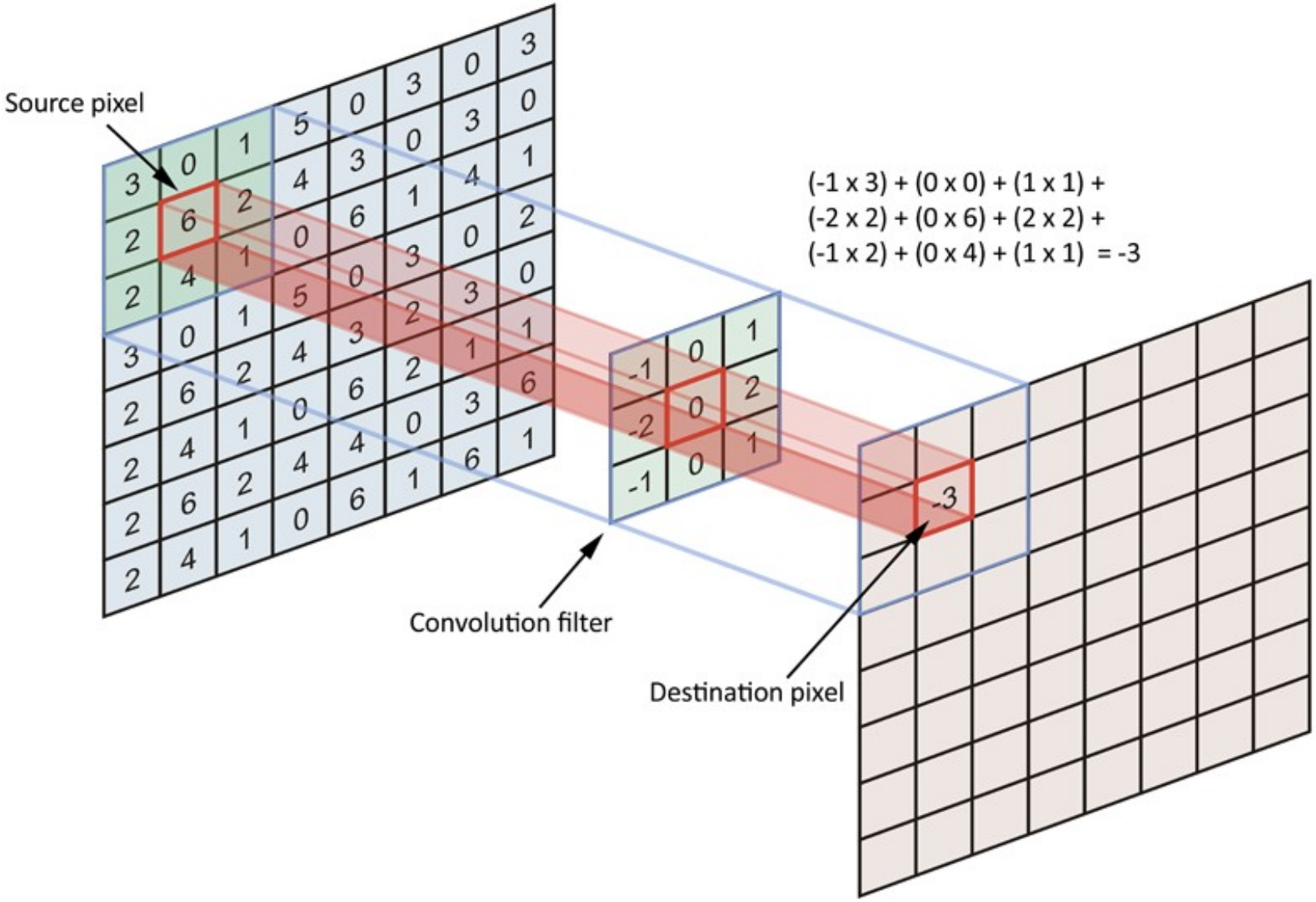
Convolutional Neural Networks

- similar to MLPs
- explicitly developed for images



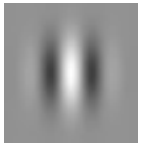
[LeNet-5, LeCun 1980]

What is a Convolution

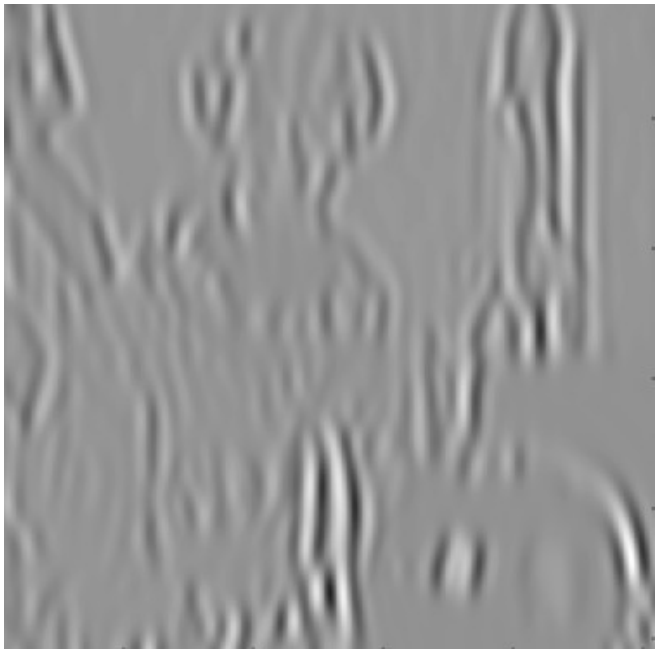


2D Convolution

*

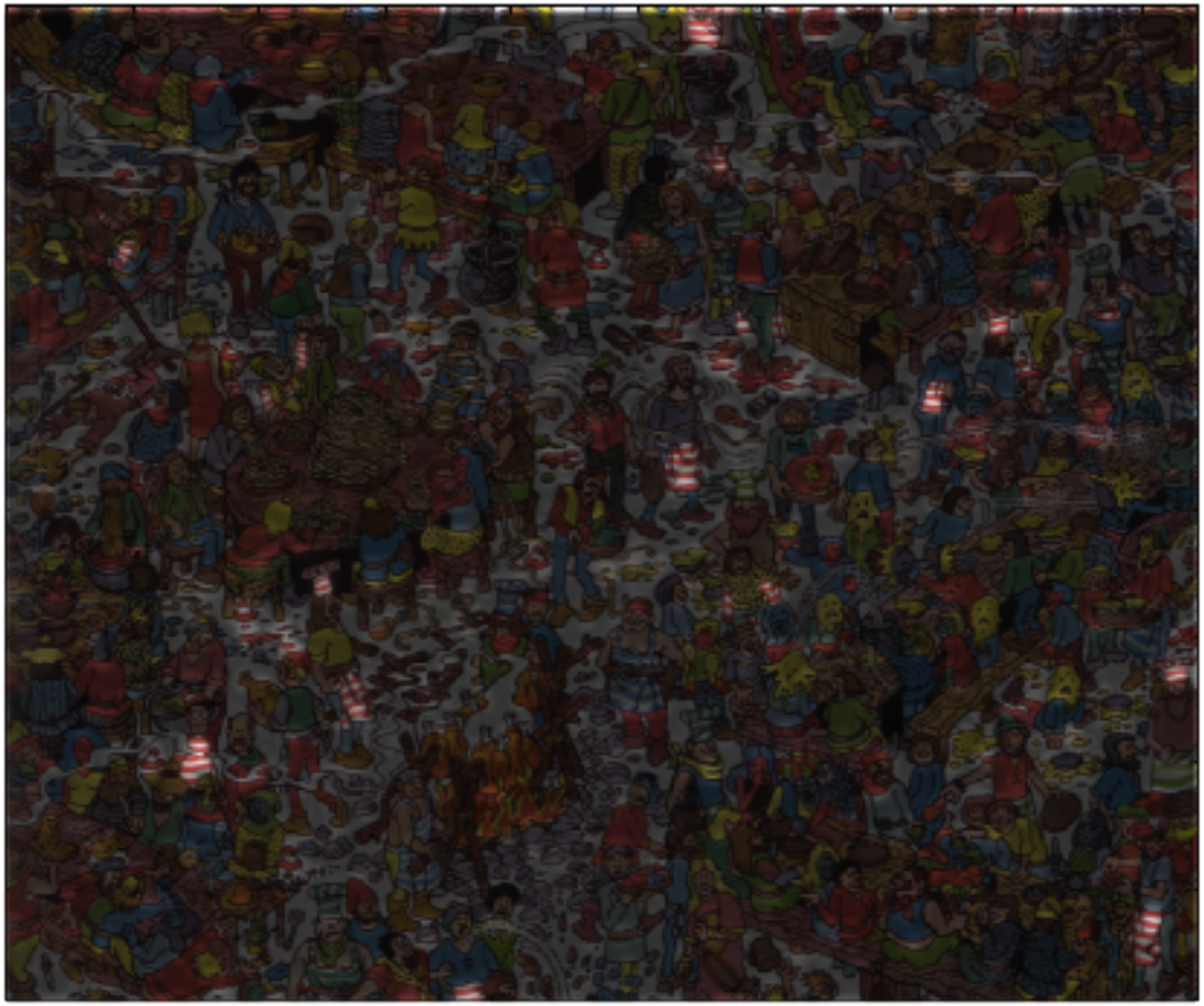


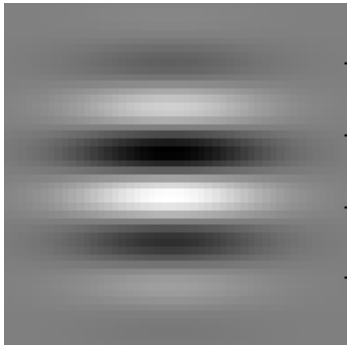
=



Filter responds to vertical edges

Template Matching

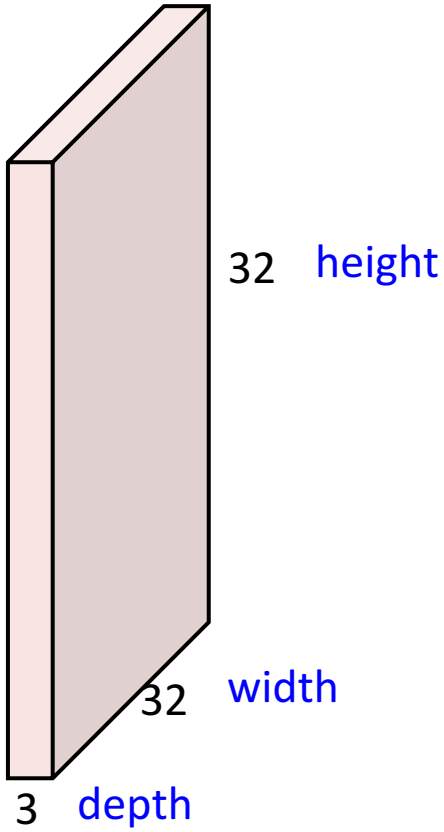




http://www.mbfys.ru.nl/~robvdw/DGCN22/PRACTICUM_2011/LABS_2011/ALTERNATIVE_LABS/Lesson_15.html

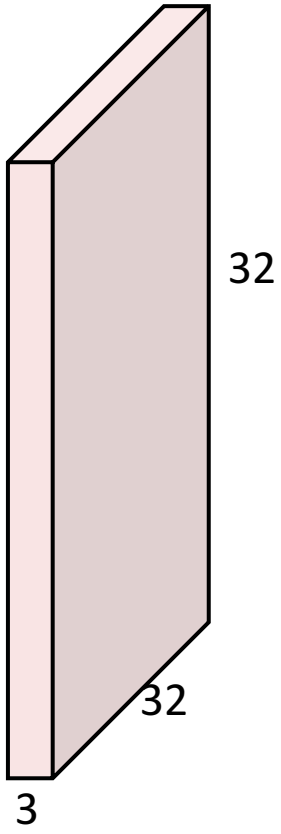
Convolution Layer

32x32x3 image

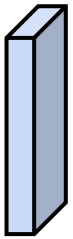


Convolution Layer

32x32x3 image



5x5x3 filter

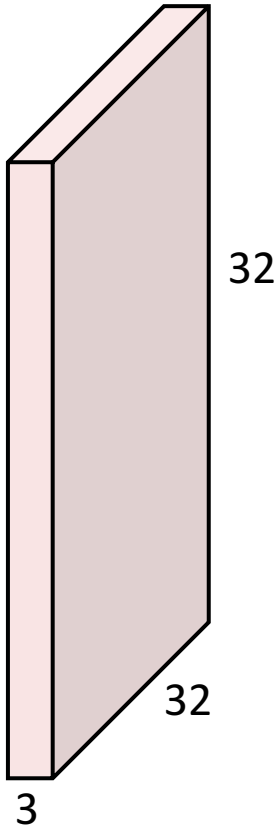


Convolve the filter with the image
i.e. “slide over the image spatially,
computing dot products”

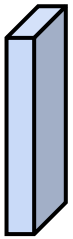
Convolution Layer

Filters always extend the full depth of the input volume

32x32x3 image

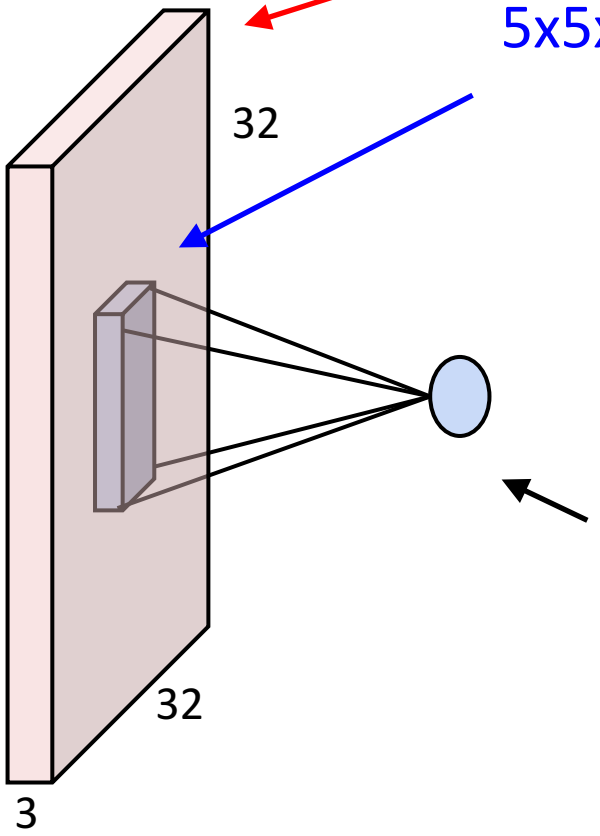


5x5x3 filter



Convolve the filter with the image i.e. “slide over the image spatially, computing dot products”

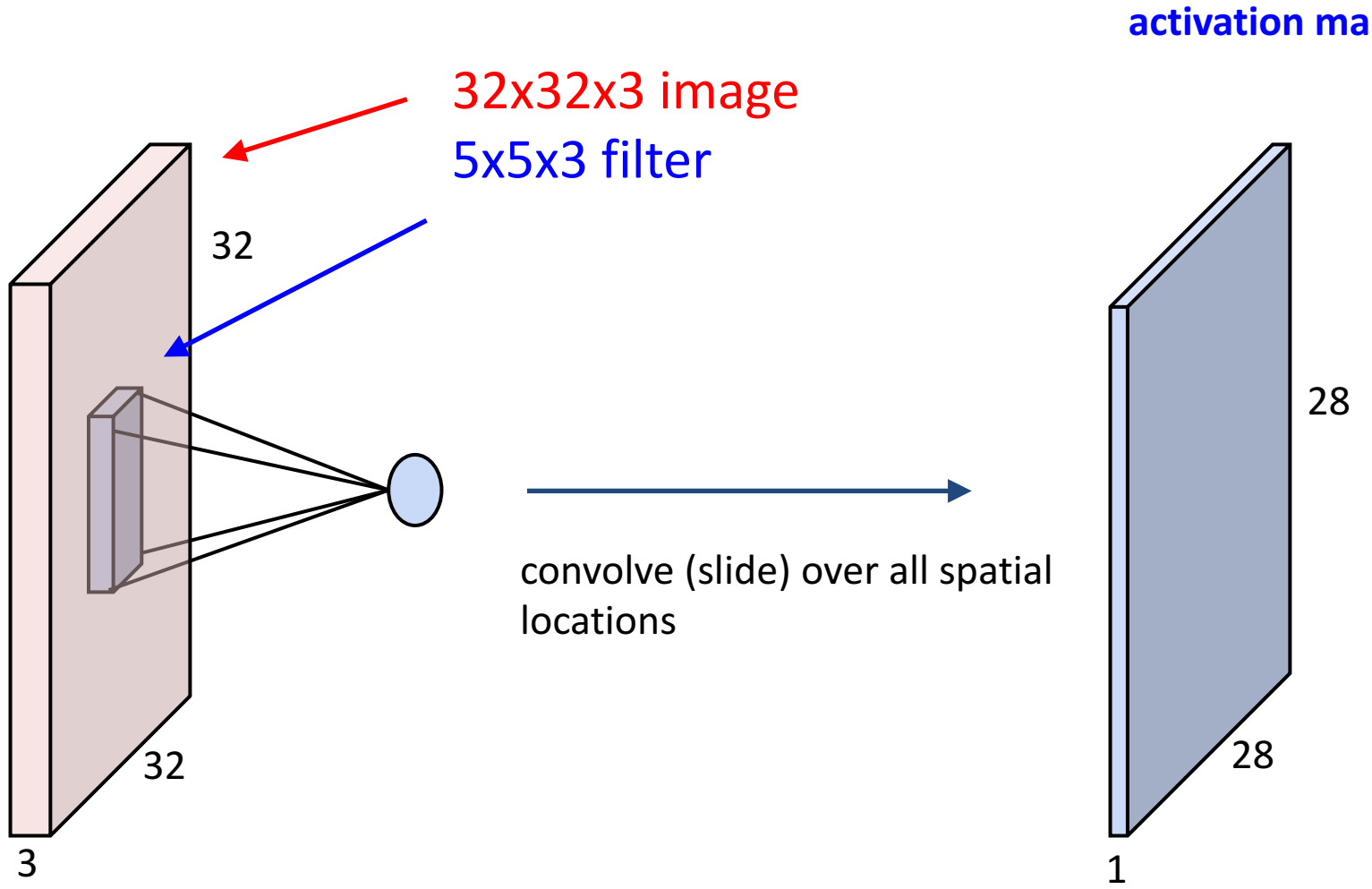
Convolution Layer



1 number:
the result of taking a dot product between the filter and a small 5x5x3 chunk of the image
(i.e. 5*5*3 = 75-dimensional dot product + bias)

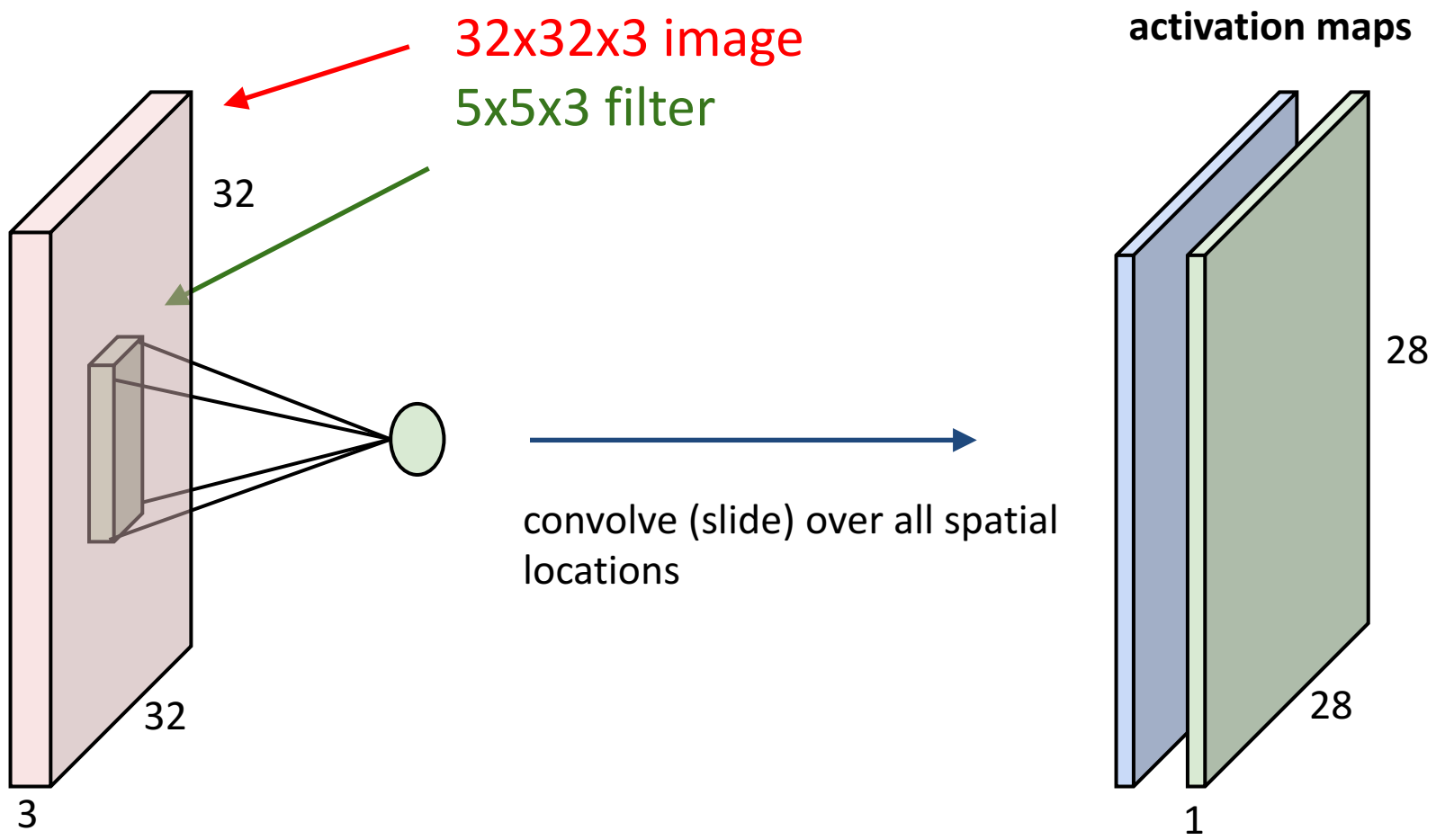
$$w^T x + b$$

Convolution Layer

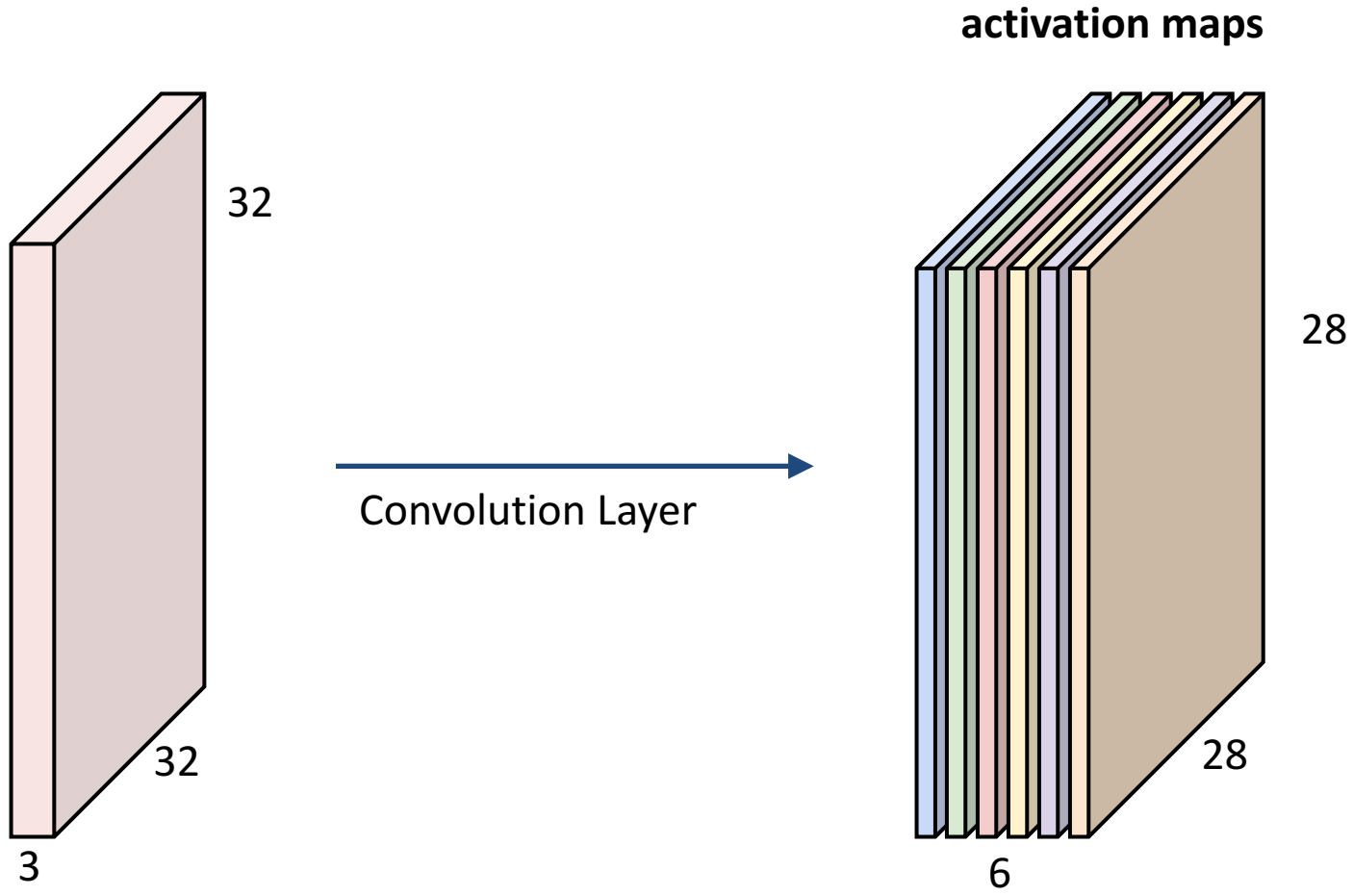


Convolution Layer

consider a second, green filter

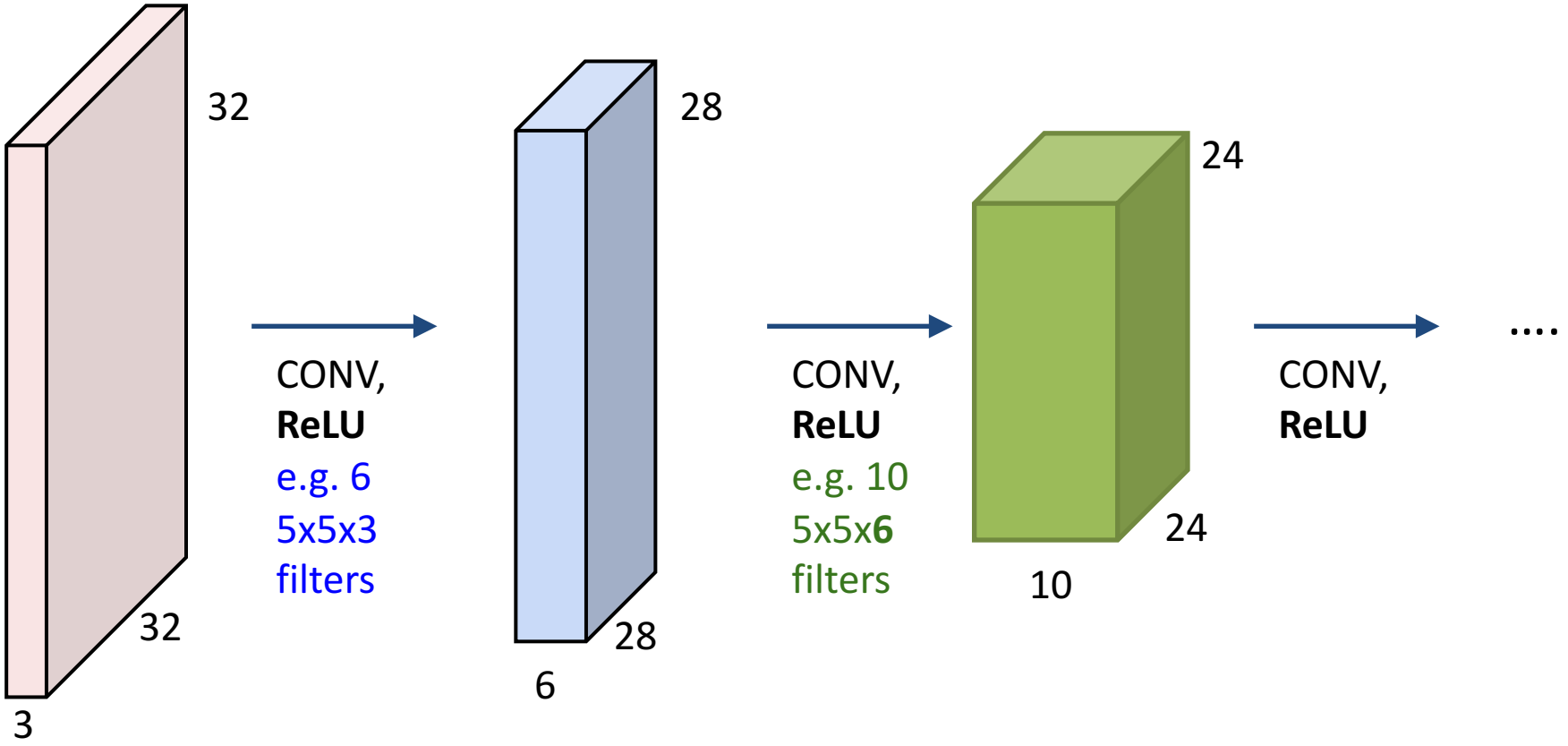


For example, if we had 6 5x5 filters, we'll get 6 separate activation maps:



We stack these up to get a "new image" of size 28x28x6
Then we apply our activation function to this new image

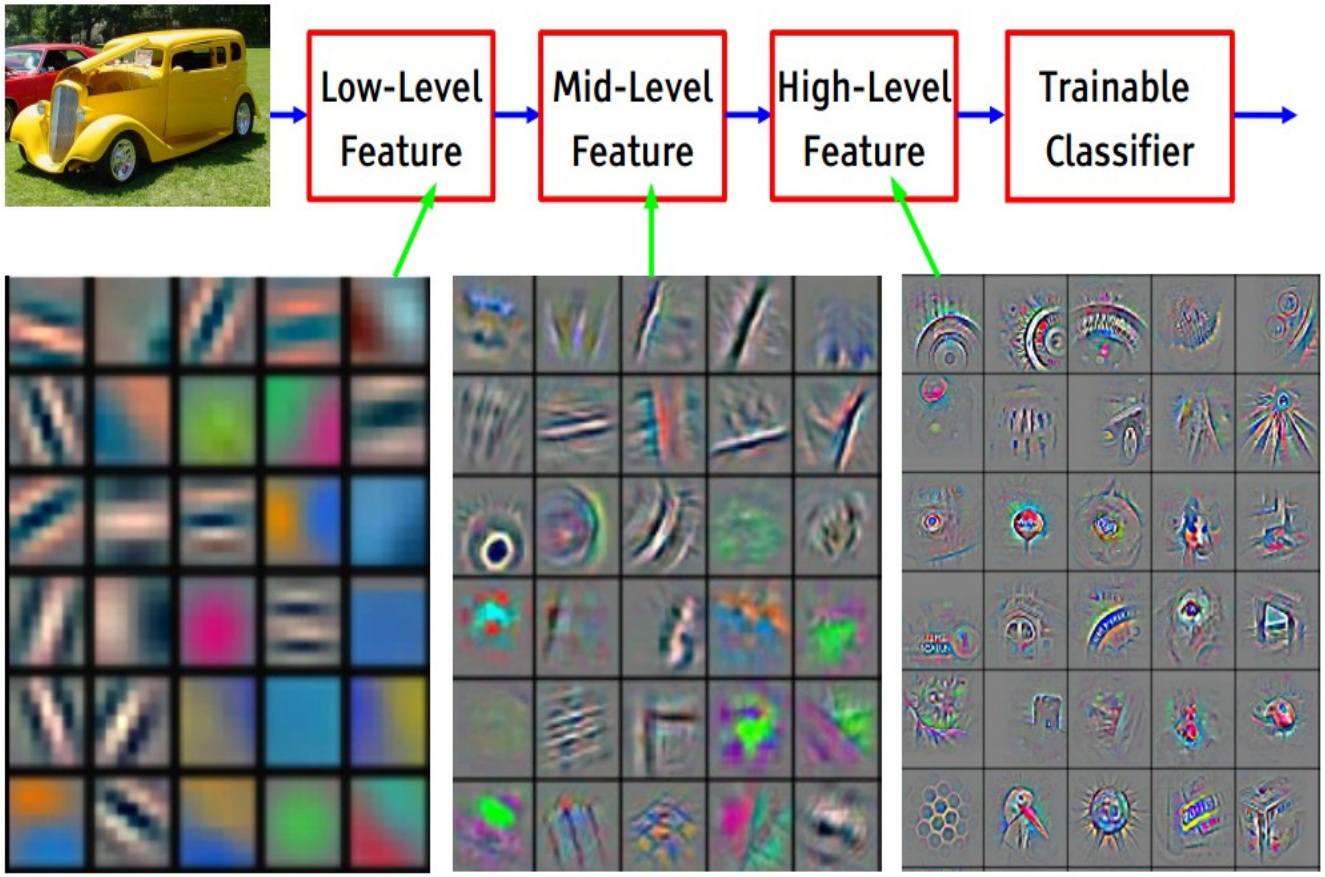
ConvNet is a sequence of Convolutional Layers, interspersed with activation functions



http://cs231n.github.io/

Preview

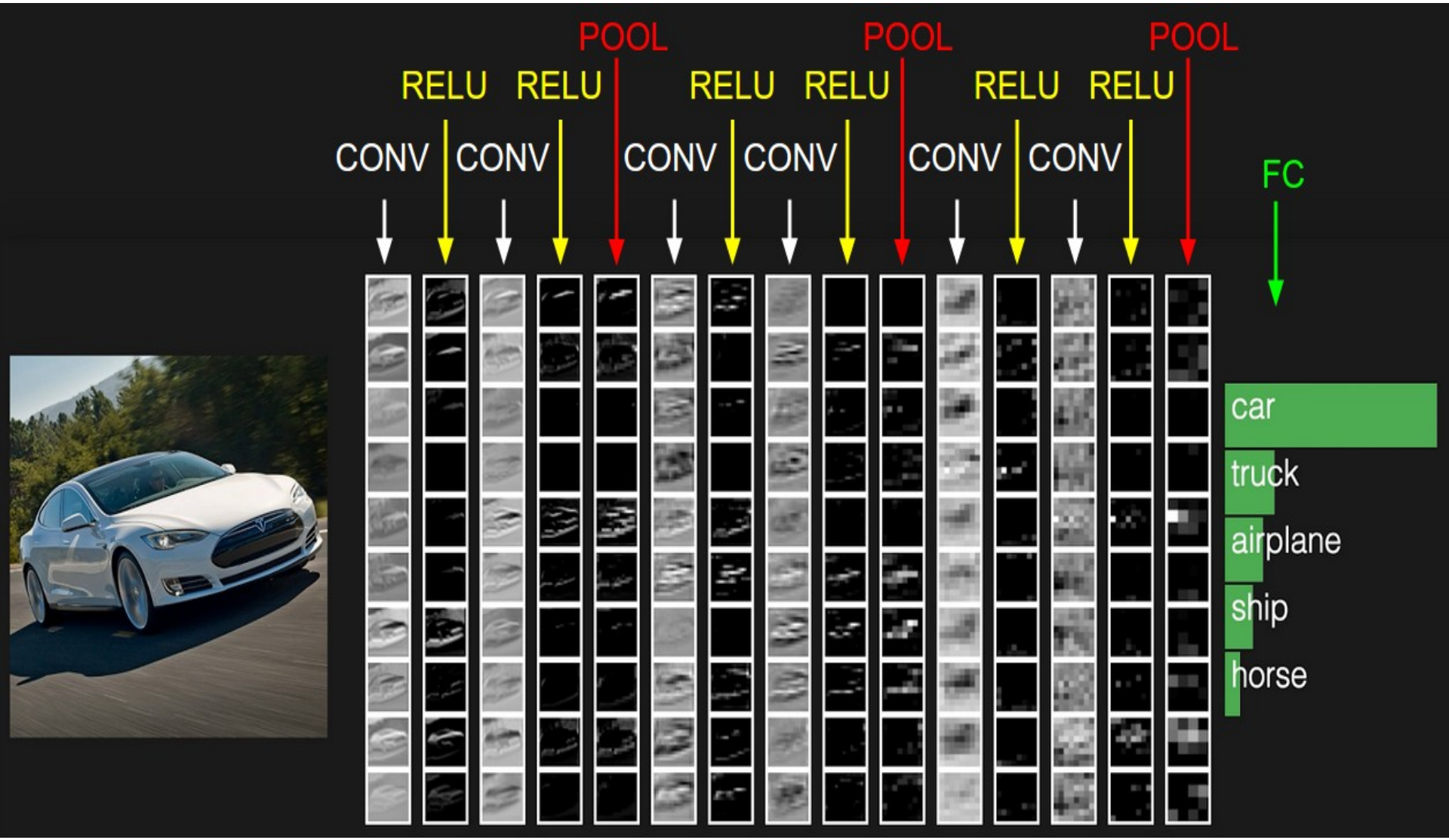
[From recent Yann LeCun slides]



Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]

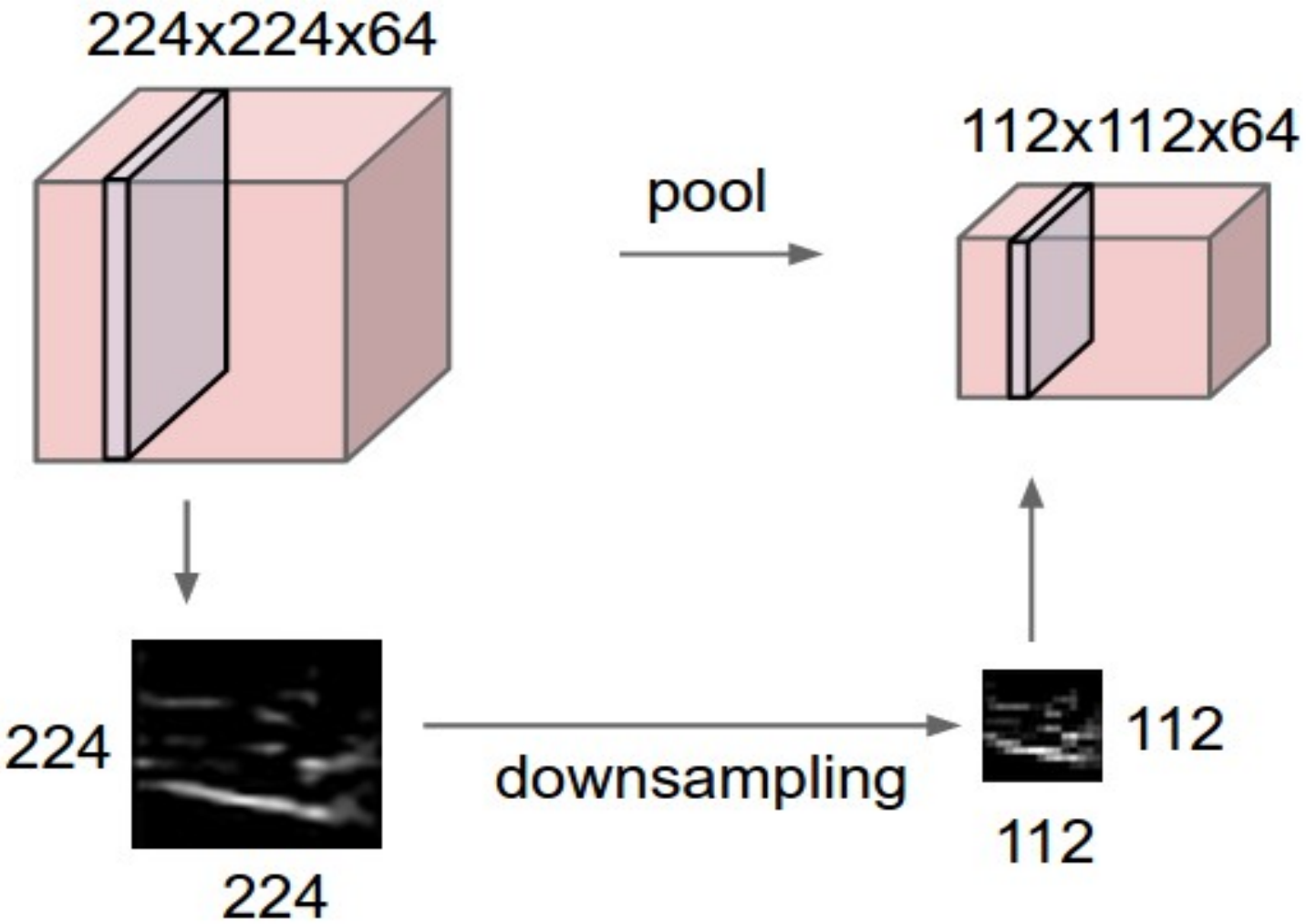
http://cs231n.github.io/

Missing: Pool and FC



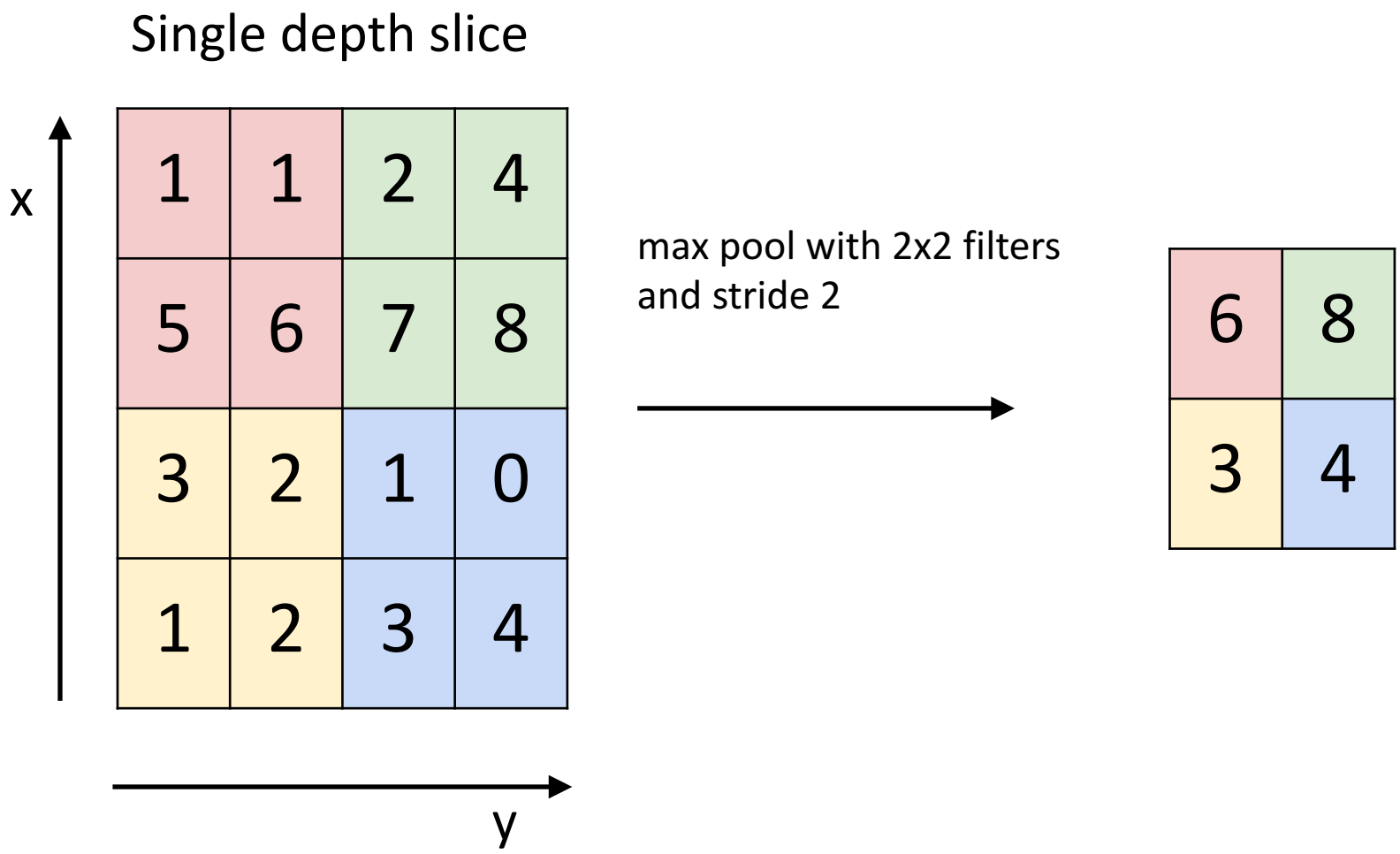
Pooling layer

- makes the representations smaller and more manageable
- introduces (limited) translation invariance
- operates over each activation map independently:



http://cs231n.github.io/

MAX POOLING

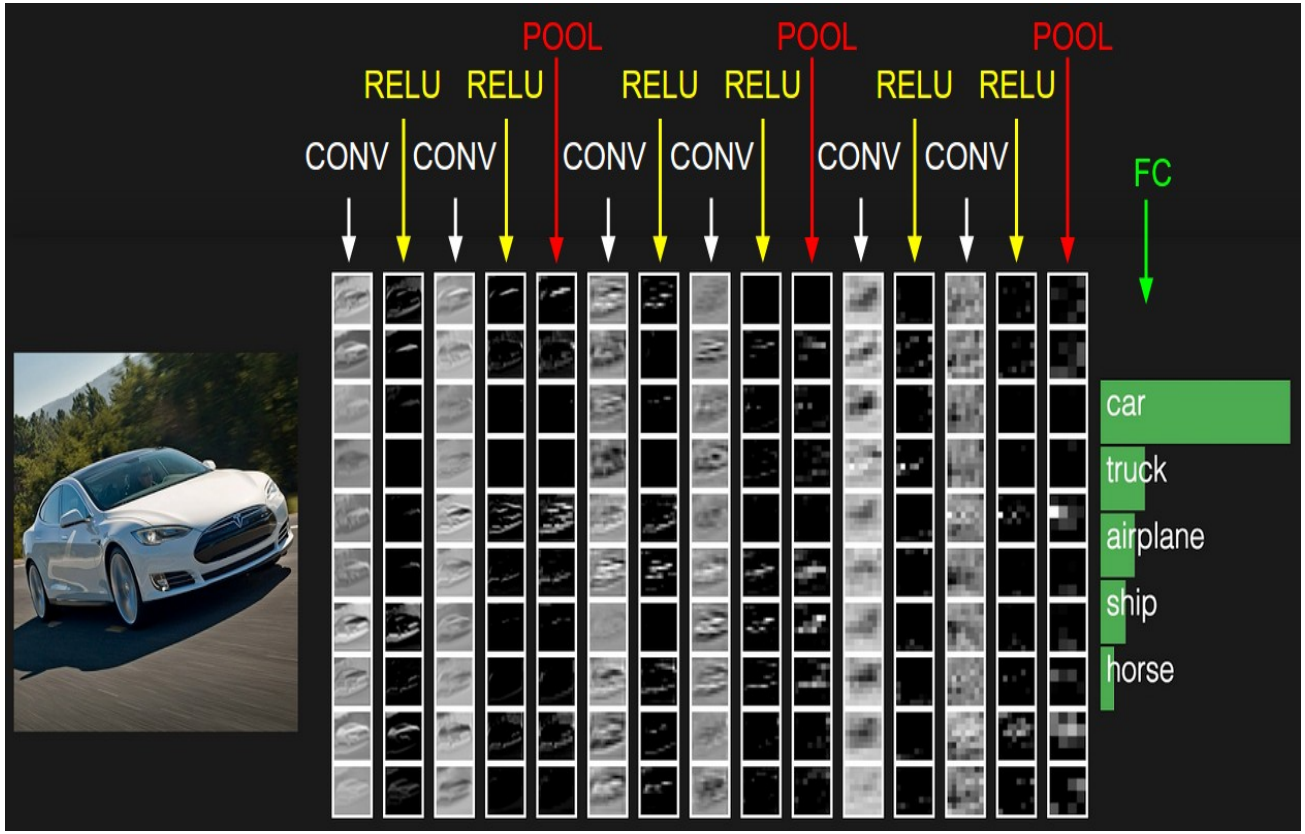

Single depth slice
x
y
1 1 2 4
5 6 7 8
3 2 1 0
1 2 3 4
max pool with 2x2 filters
and stride 2
6 8
3 4

Max Pooling

- Distant past: Mean or average pooling
- Recent past: Conv layers and max pool layers alternated
- Now: mutiple convolution layers than max pool
- Depth remains the same after max pool
- Don‘t make it too large or you loose too much
- Common settings:
  - F=2, S=2 (default)
  - F=3, S=2 (overlapping pooling)

Fully Connected Layer (FC layer)

- Contains neurons that connect to the entire input volume, as in ordinary Neural Networks



http://cs231n.github.io/

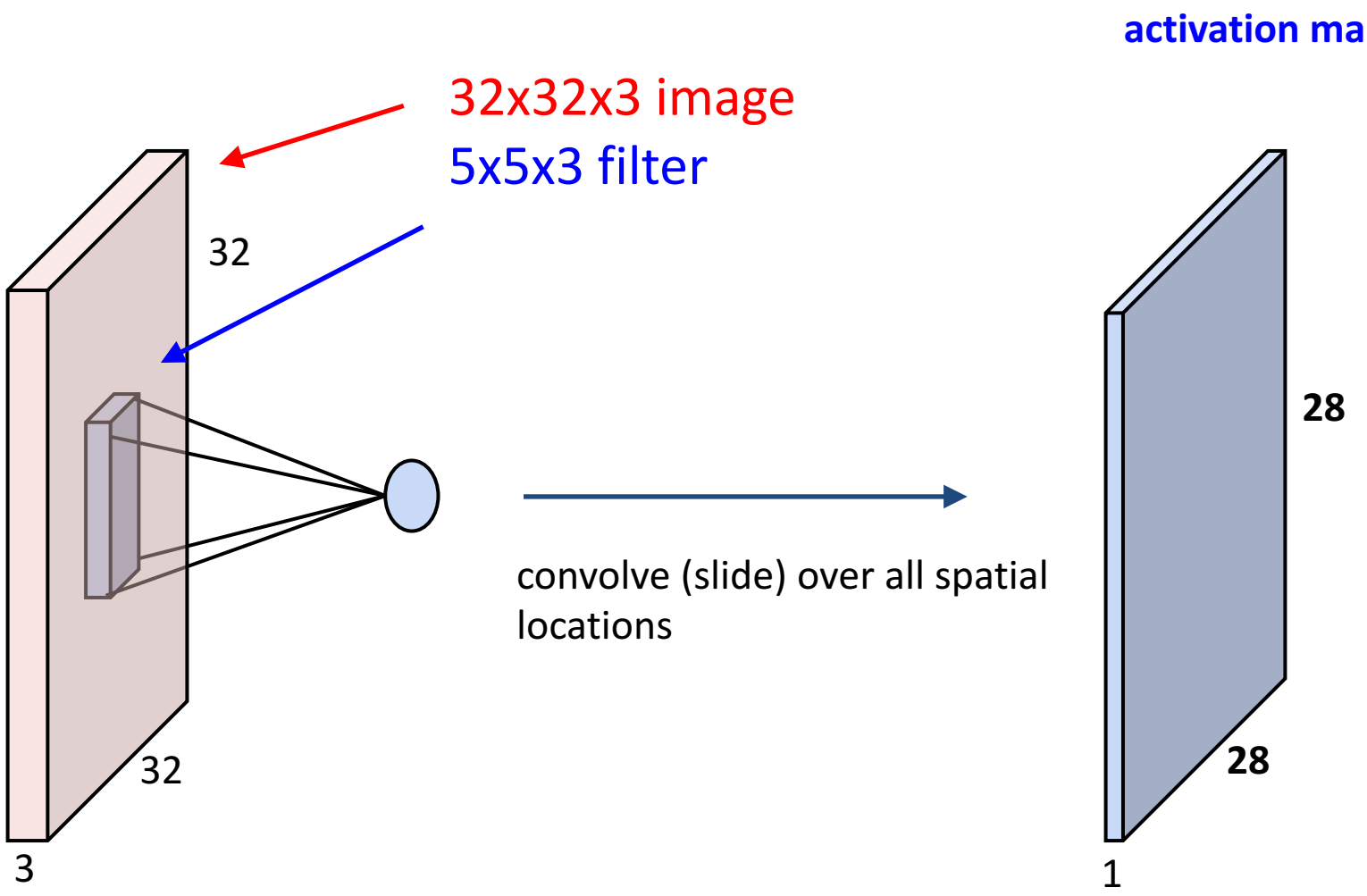
Keras Example

- From MLP to CNN

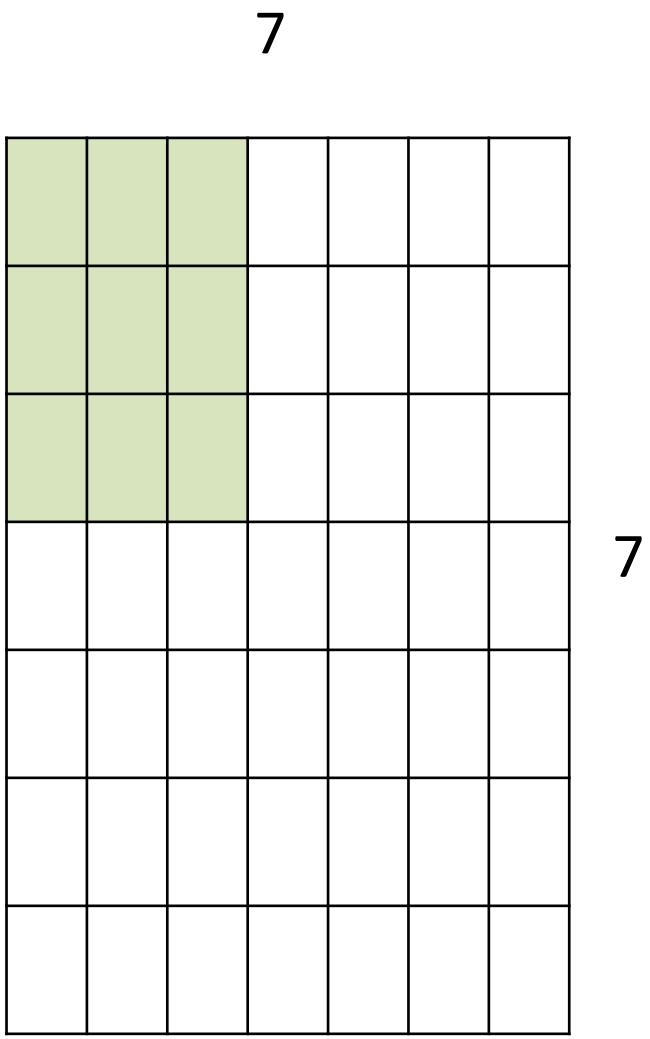
```
keras.layers.convolutional.Conv2D(filters,
       kernel_size, strides=(1, 1), padding='valid',
       activation=None, use_bias=True,
       kernel_regularizer=None, bias_regularizer=None)
```

- What is that stride parameter?

A closer look at spatial dimensions:

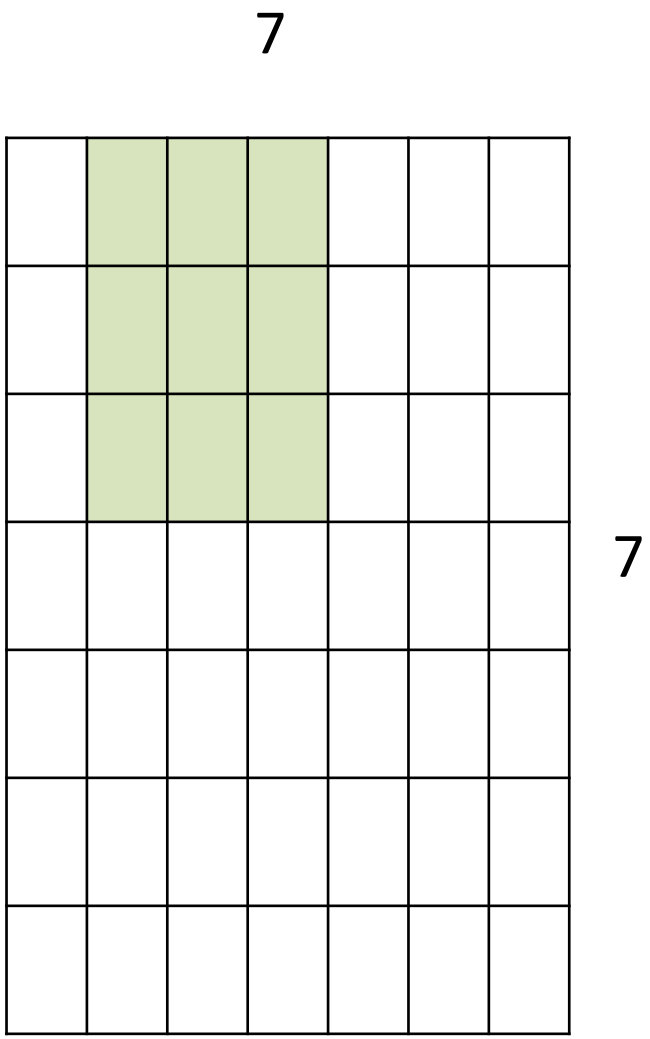


A closer look at spatial dimensions:



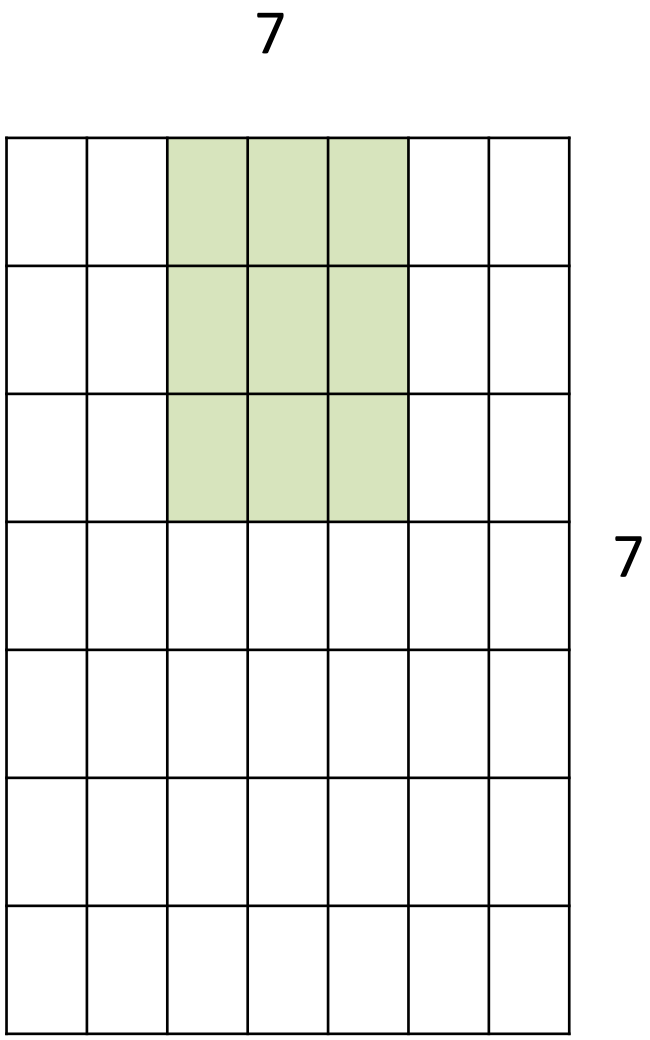
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



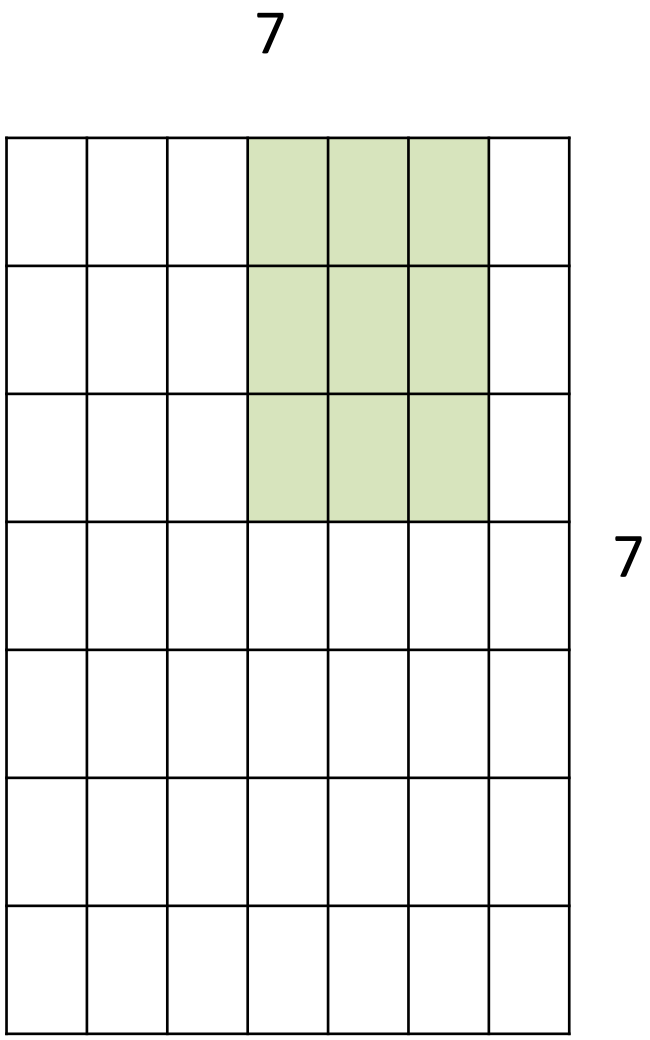
7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

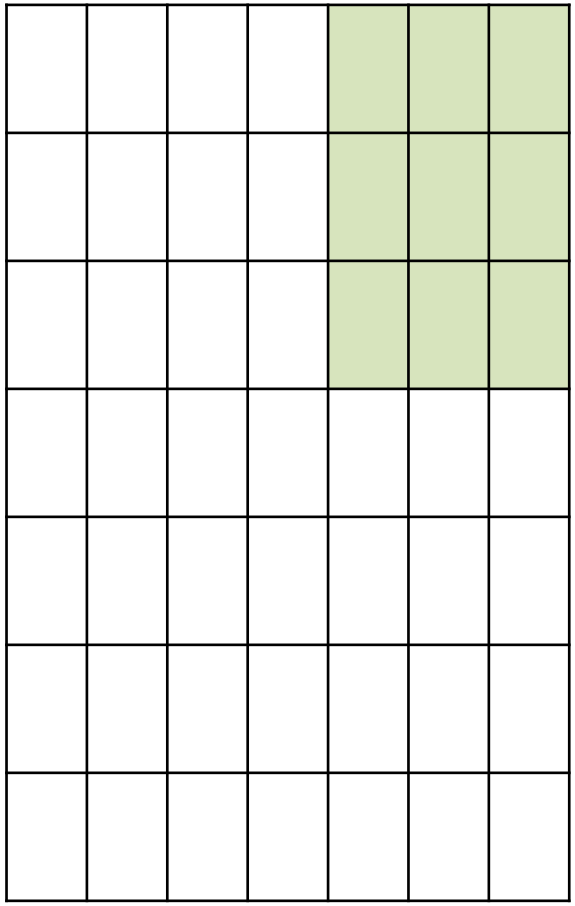
A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter

A closer look at spatial dimensions:

7

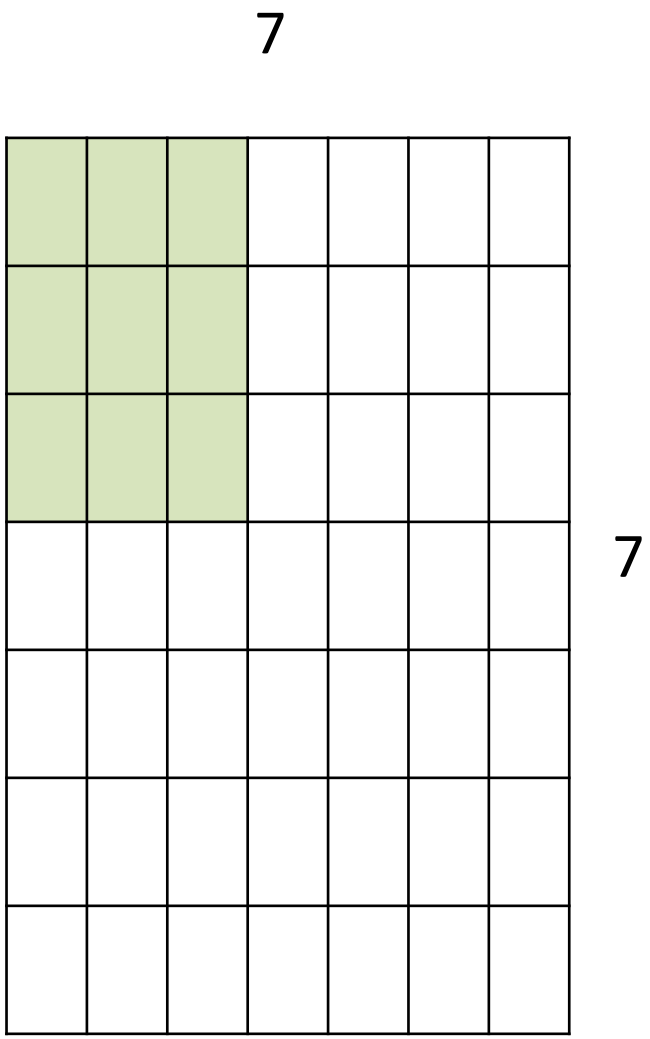


7

7x7 input (spatially)
assume 3x3 filter

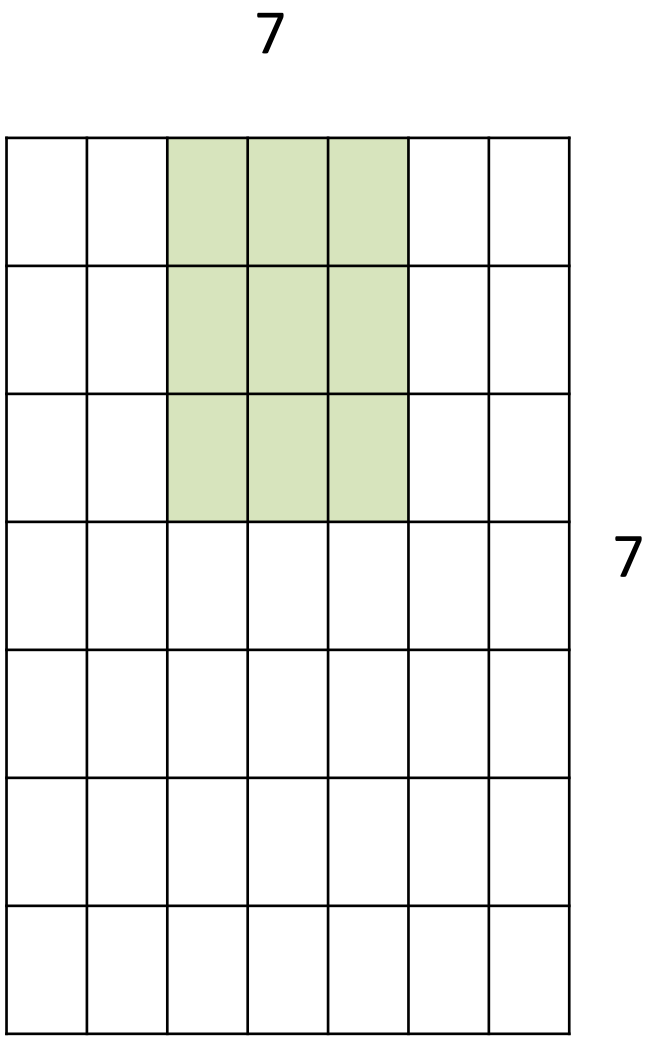
=> 5x5 output

A closer look at spatial dimensions:



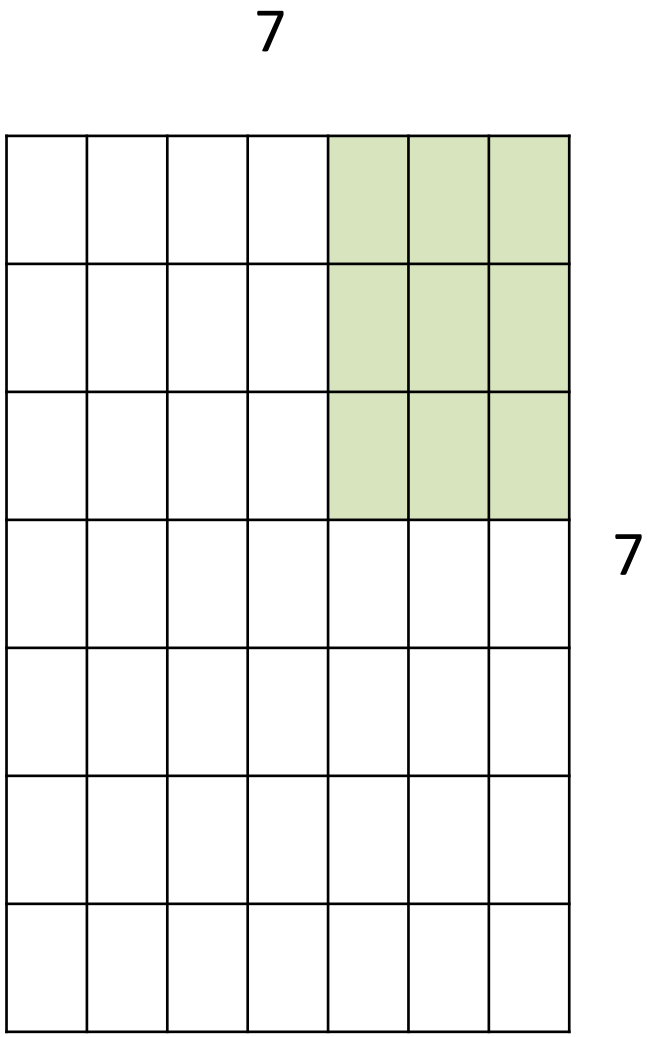
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



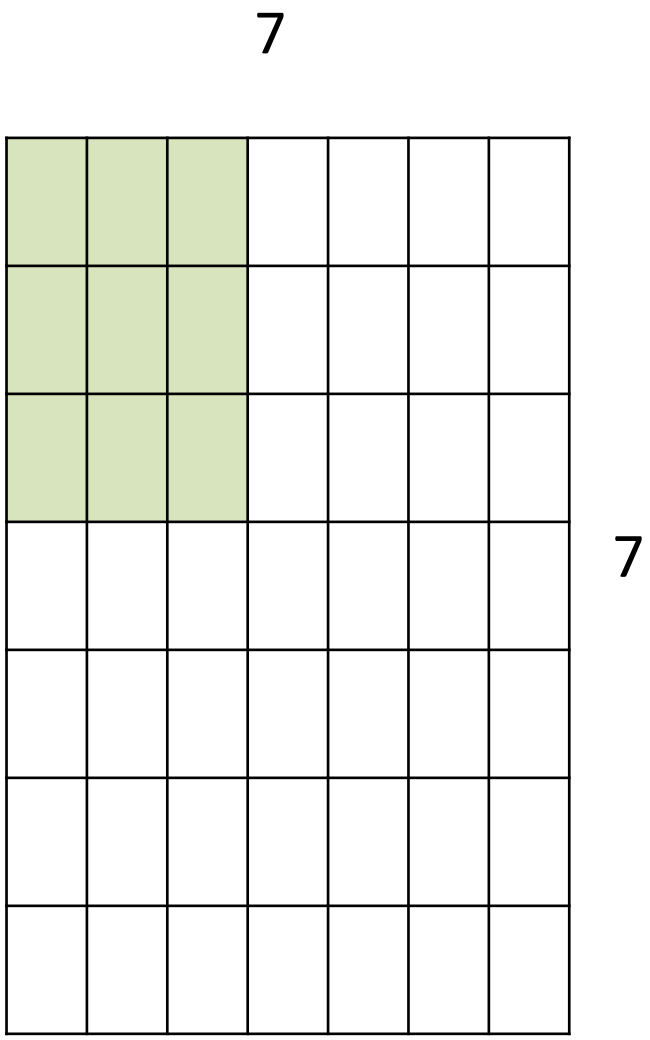
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**

A closer look at spatial dimensions:



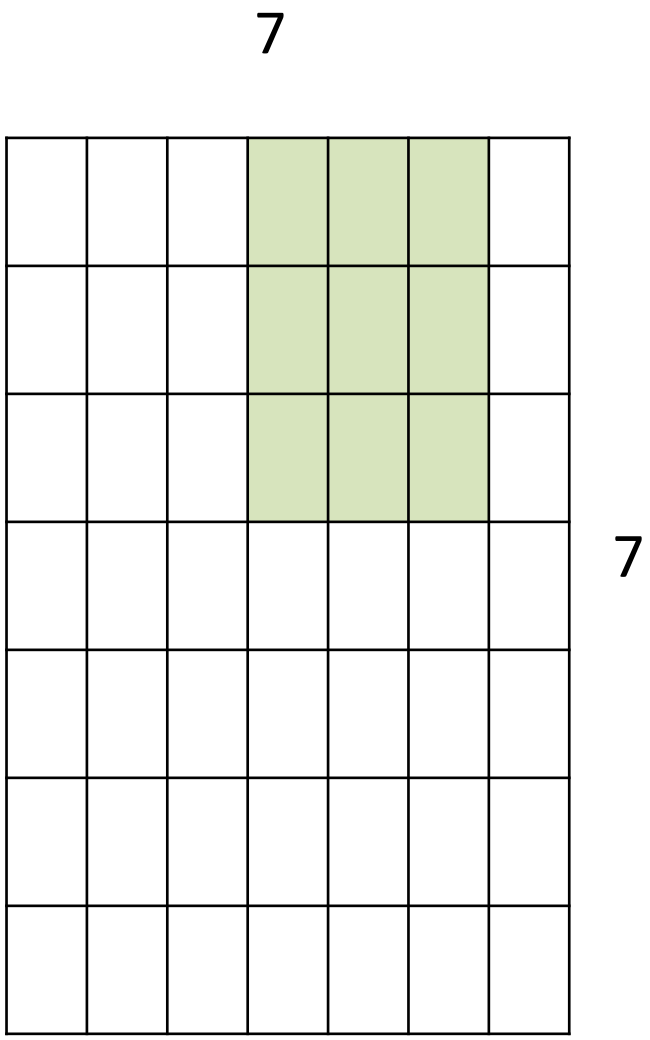
7x7 input (spatially)
assume 3x3 filter
applied **with stride 2**
=> 3x3 output!

A closer look at spatial dimensions:



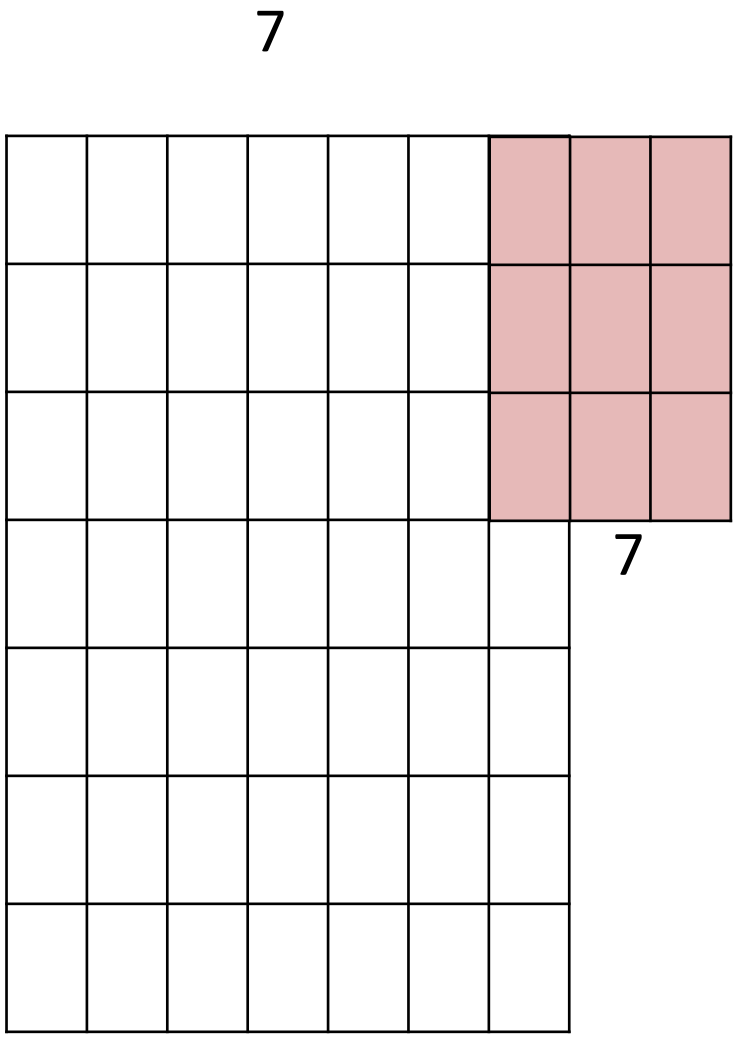
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



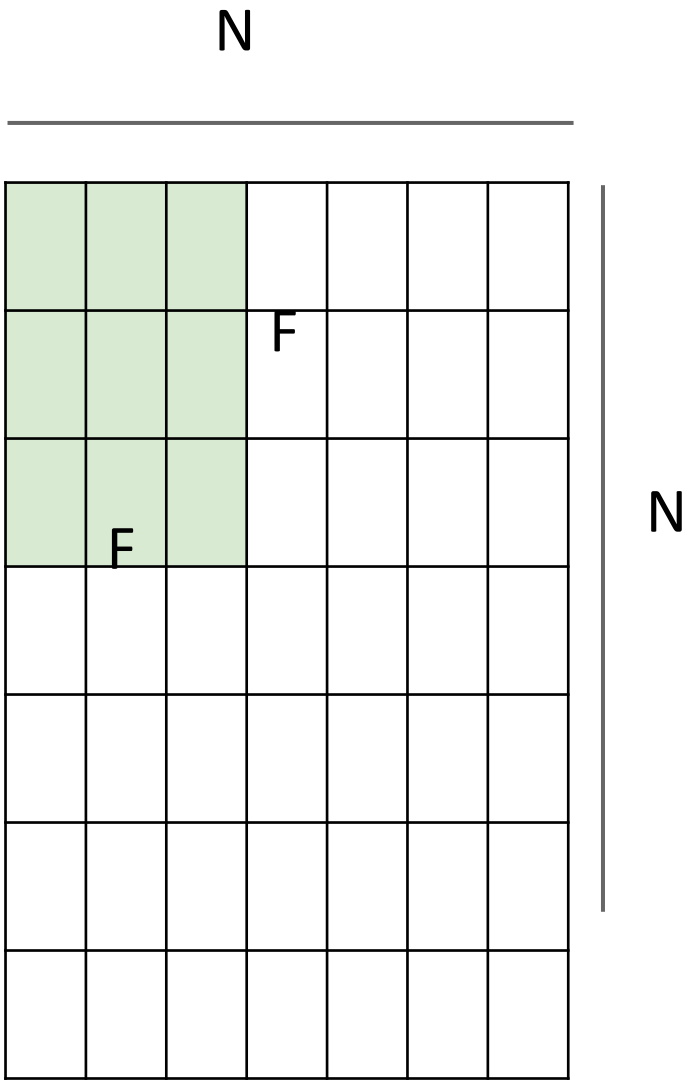
7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

A closer look at spatial dimensions:



7x7 input (spatially)
assume 3x3 filter
applied **with stride 3?**

doesn't fit!
cannot apply 3x3 filter on 7x7 input with stride 3.

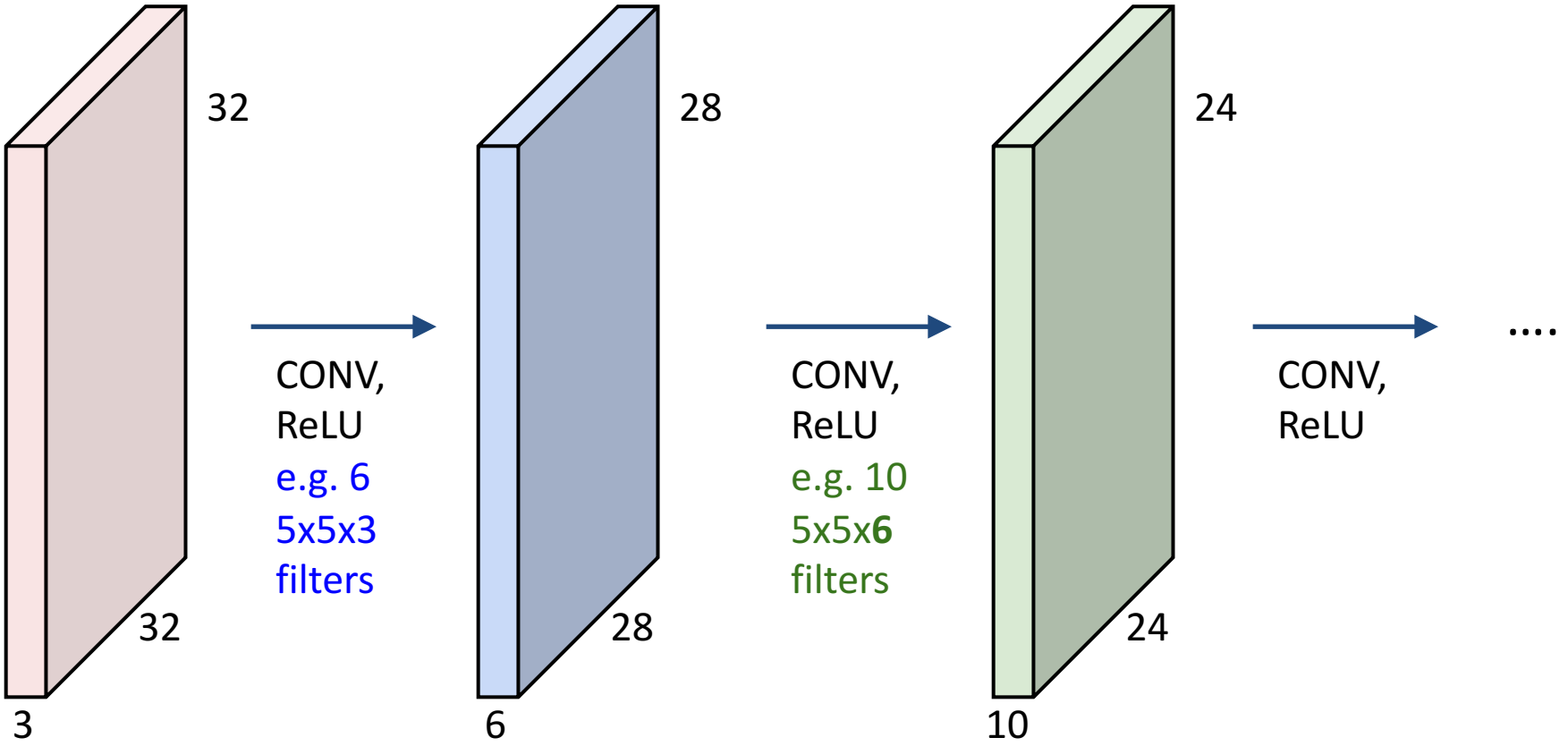


Output size:
(N - F) / stride + 1

e.g. N = 7, F = 3:
stride 1 => (7 - 3)/1 + 1 = 5
stride 2 => (7 - 3)/2 + 1 = 3
stride 3 => (7 - 3)/3 + 1 = 2.33

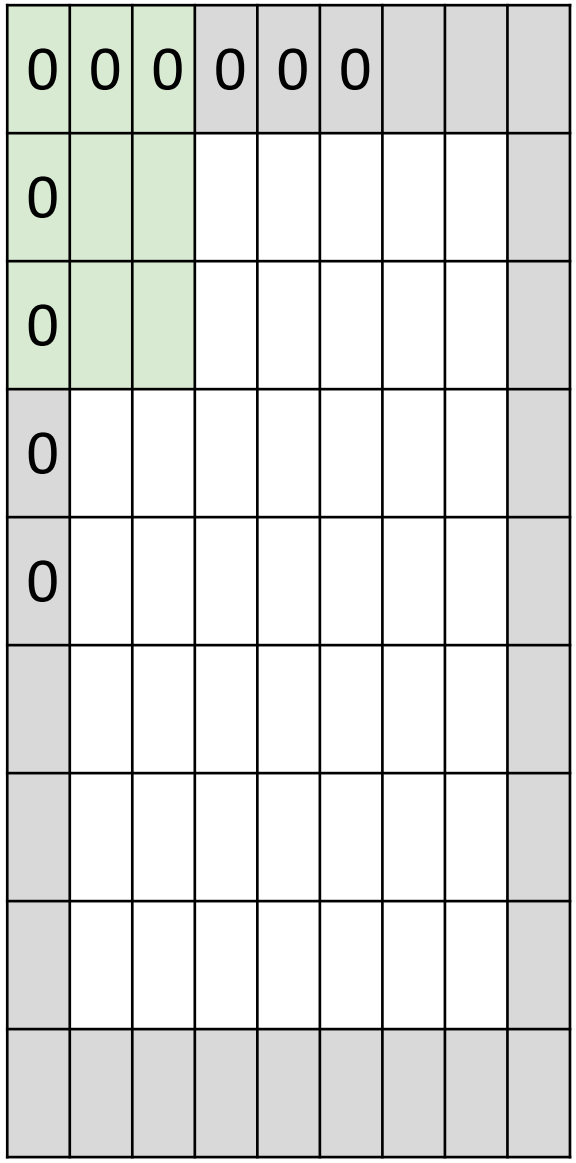
Remember back to…

E.g. 32x32 input convolved repeatedly with 5x5 filters shrinks volumes spatially!
(32 -> 28 -> 24 ...). Shrinking too fast is not good, doesn't work well.



http://cs231n.github.io/

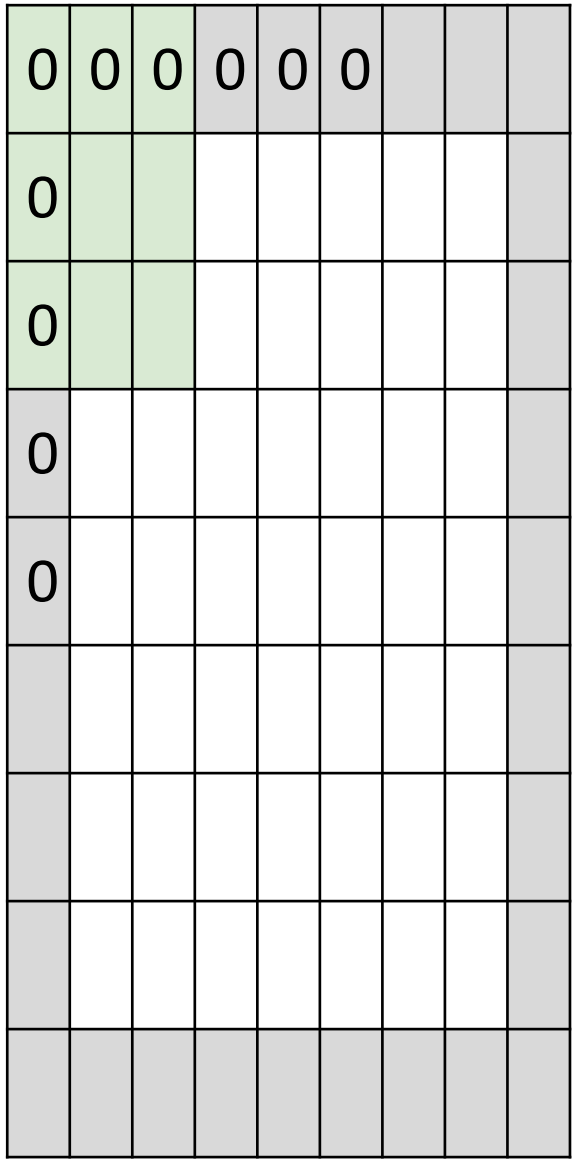
In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!

In practice: Common to zero pad the border



e.g. input 7x7
3x3 filter, applied with **stride 1**
pad with 1 pixel border => what is the output?

7x7 output!
in general, common to see CONV layers with stride 1, filters of size FxF, and zero-padding with (F-1)/2. (will preserve size spatially)
e.g. F = 3 => zero pad with 1
F = 5 => zero pad with 2
F = 7 => zero pad with 3

Keras Example

- Some CNN layer parameters in Keras

```
keras.layers.convolutional.Conv2D(filters,
       kernel_size, strides=(1, 1), padding='valid',
       activation=None, use_bias=True,
       kernel_regularizer=None, bias_regularizer=None)
```

Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

http://cs231n.github.io/

Common settings:

K = (powers of 2, e.g. 32, 64, 128, 512)

- F = 3, S = 1, P = 1
- F = 5, S = 1, P = 2
- F = 5, S = 2, P = ? (whatever fits)
- F = 1, S = 1, P = 0

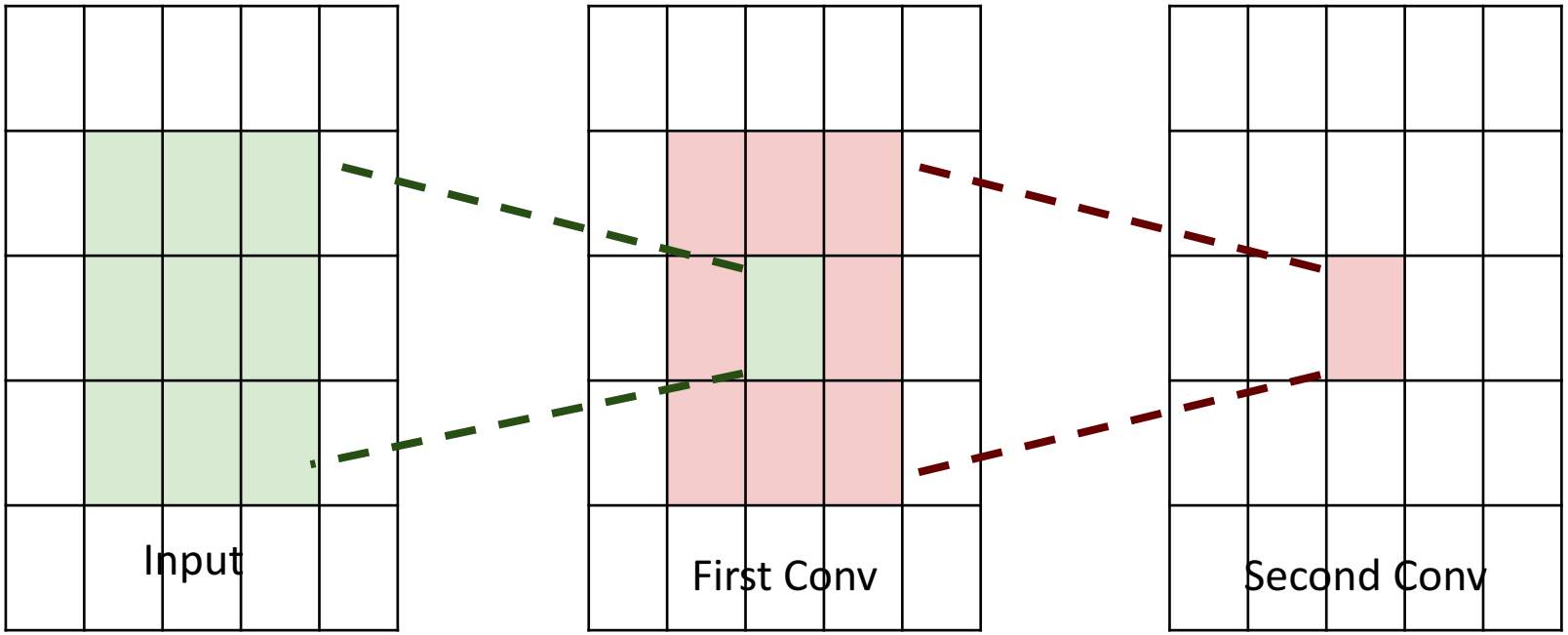
Summary. To summarize, the Conv Layer:

- Accepts a volume of size $W_1 \times H_1 \times D_1$
- Requires four hyperparameters:
  - Number of filters K,
  - their spatial extent F,
  - the stride S,
  - the amount of zero padding P.
- Produces a volume of size $W_2 \times H_2 \times D_2$ where:
  - $W_2 = (W_1 - F + 2P)/S + 1$
  - $H_2 = (H_1 - F + 2P)/S + 1$ (i.e. width and height are computed equally by symmetry)
  - $D_2 = K$
- With parameter sharing, it introduces $F \cdot F \cdot D_1$ weights per filter, for a total of $(F \cdot F \cdot D_1) \cdot K$ weights and K biases.
- In the output volume, the d-th depth slice (of size $W_2 \times H_2$) is the result of performing a valid convolution of the d-th filter over the input volume with a stride of S, and then offset by d-th bias.

http://cs231n.github.io/

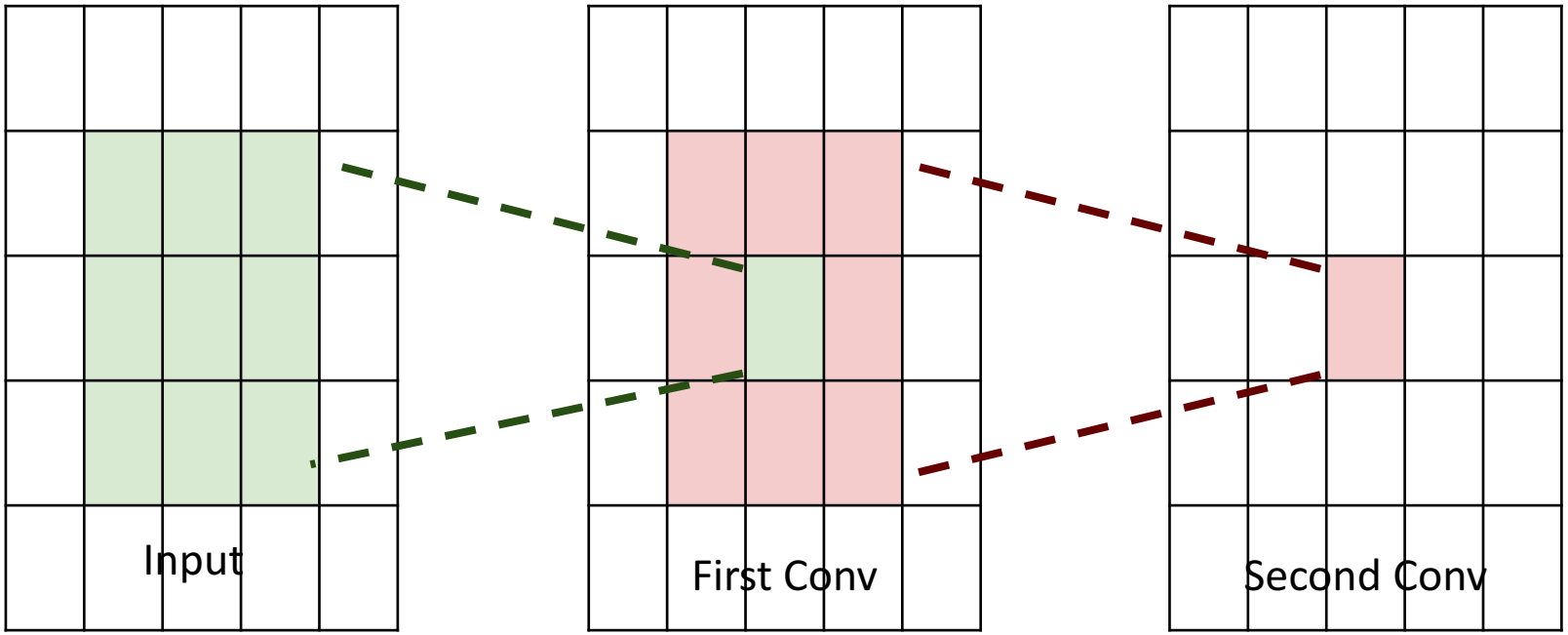
The power of small filters

Suppose we stack two 3x3 conv layers (stride 1)
Each neuron sees 3x3 region of previous activation map



The power of small filters

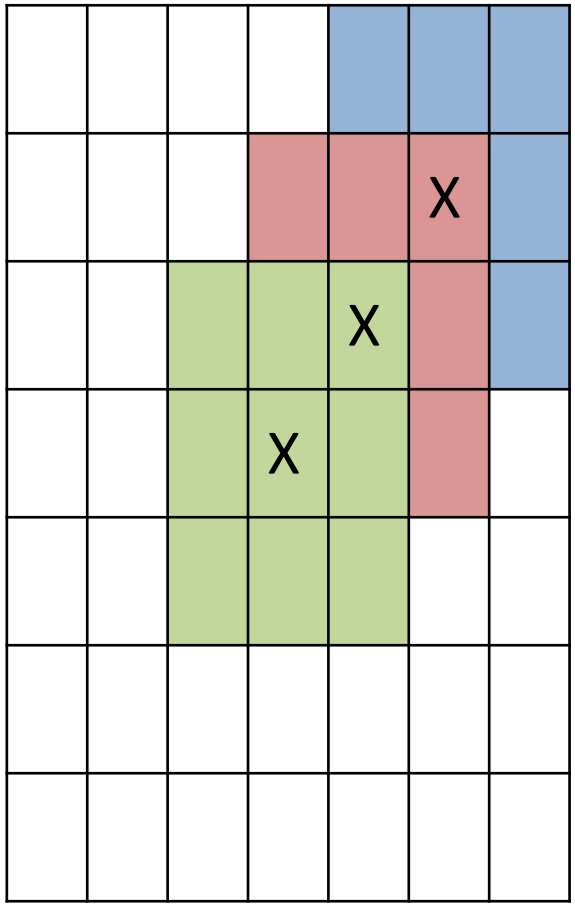
Question: How big of a region in the input does a neuron on the second conv layer see?
Answer: 5 x 5



The power of small filters

Question: If we stack **three** 3x3 conv layers, how big of an input region does a neuron in the third layer see?

Answer: 7 x 7



Three 3 x 3 conv gives similar representational power as a single 7 x 7 convolution

The power of small filters

Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

one CONV with 7 x 7 filters

Number of weights:
= C x (7 x 7 x C) = **49 C^2**

three CONV with 3 x 3 filters

Number of weights:
= 3 x C x (3 x 3 x C) = **27 C^2**

Fewer parameters, more nonlinearity = GOOD

The power of small filters

Suppose input is H x W x C and we use convolutions with C filters to preserve depth (stride 1, padding to preserve H, W)

one CONV with 7 x 7 filters

Number of weights:
= C x (7 x 7 x C) = 49 C^2

Number of multiply-adds:
= (H x W x C) x (7 x 7 x C)
= **49 HWC^2**

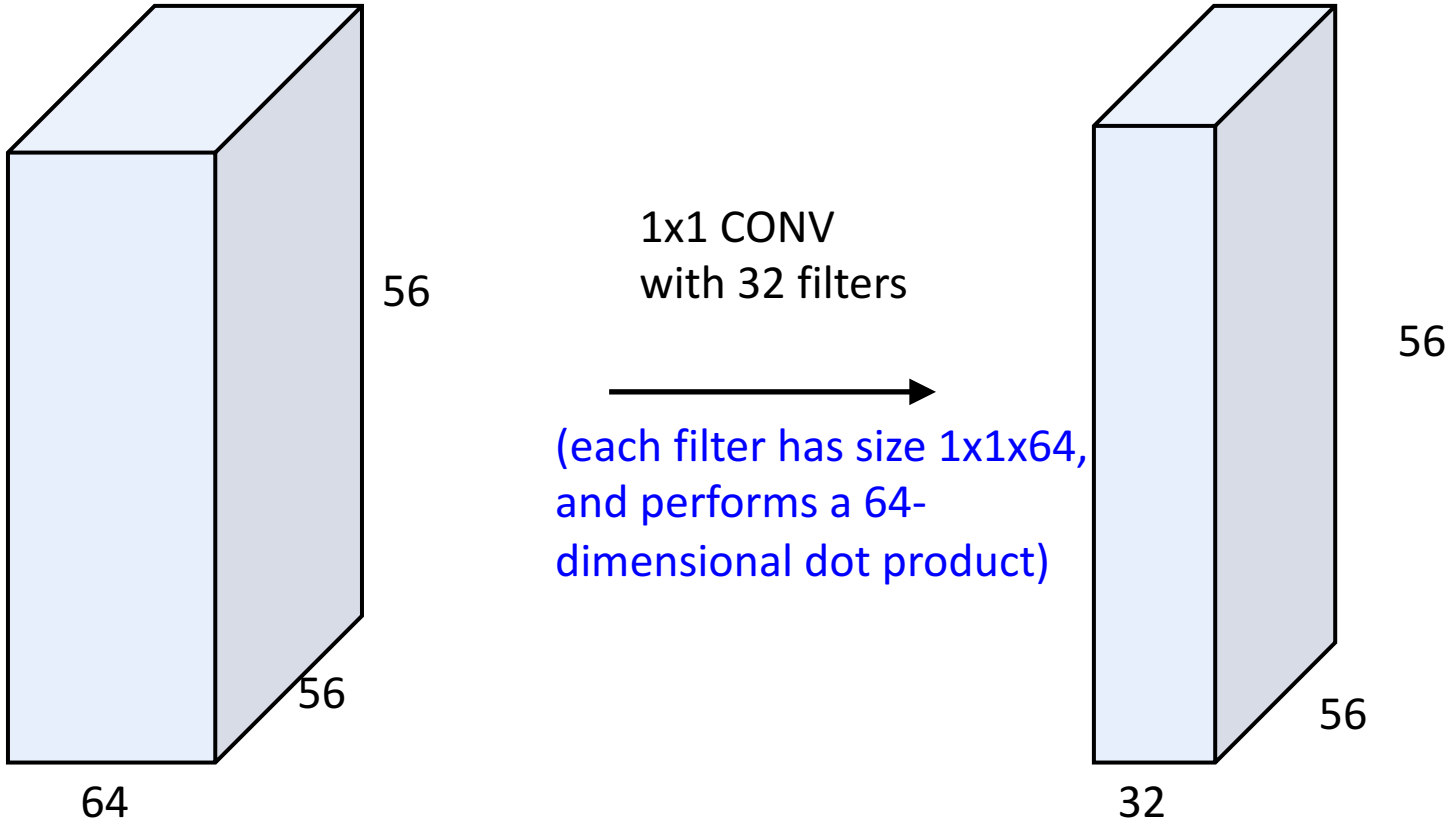
three CONV with 3 x 3 filters

Number of weights:
= 3 x C x (3 x 3 x C) = 27 C^2

Number of multiply-adds:
= 3 x (H x W x C) x (3 x 3 x C)
= **27 HWC^2**

Less compute, more nonlinearity = GOOD

(btw, 1x1 convolution layers make perfect sense)

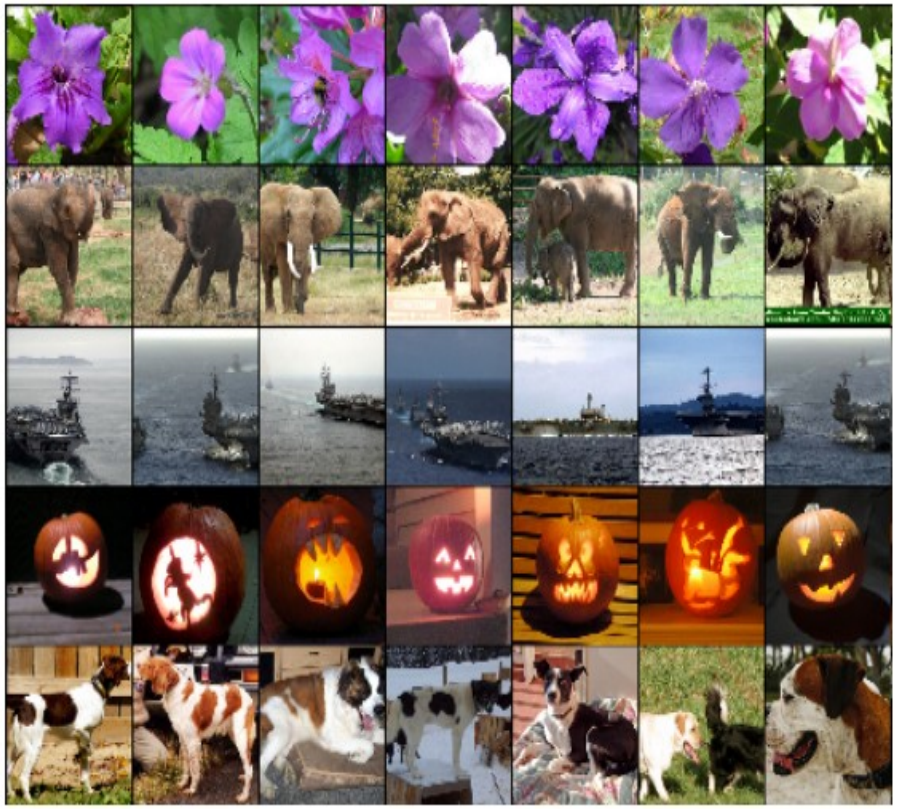


Things you can do with CNNs

ConvNets are everywhere

Classification

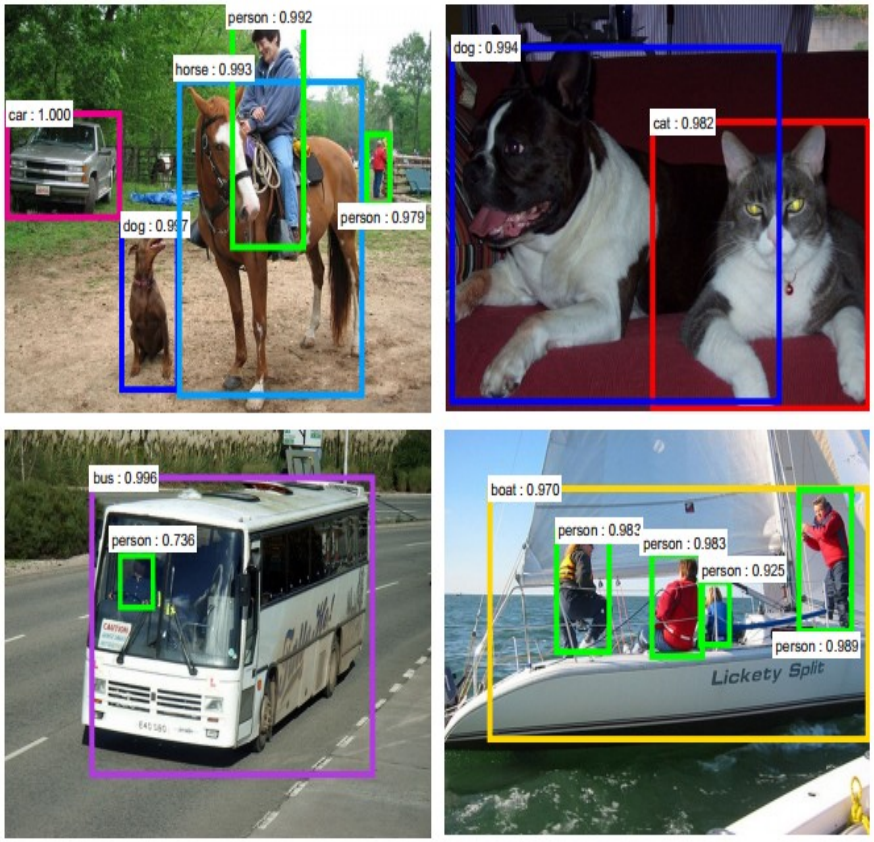
Retrieval



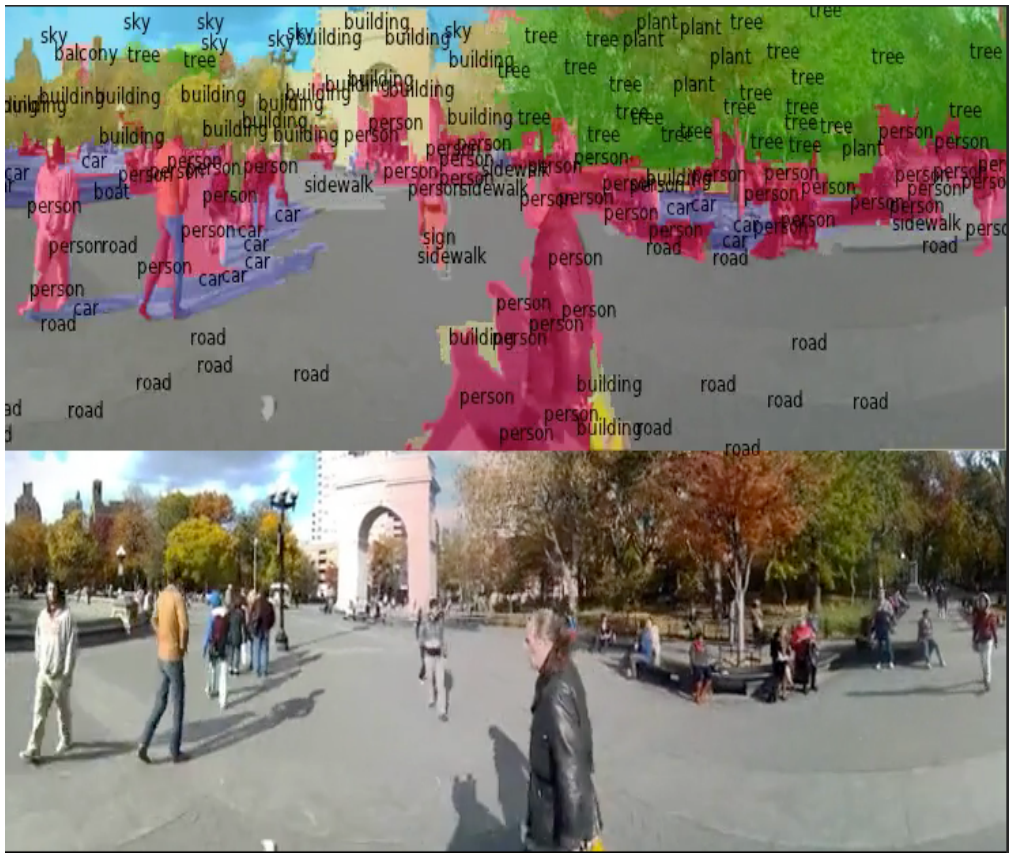
[Krizhevsky 2012]

ConvNets are everywhere

Detection



Segmentation



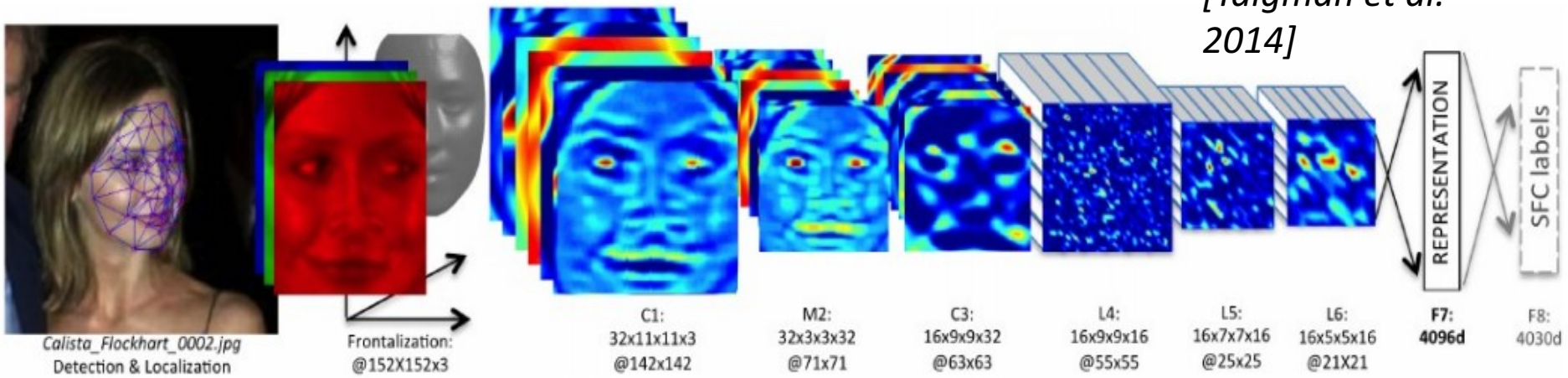
[Faster R-CNN: Ren, He, Girshick, Sun 2015]

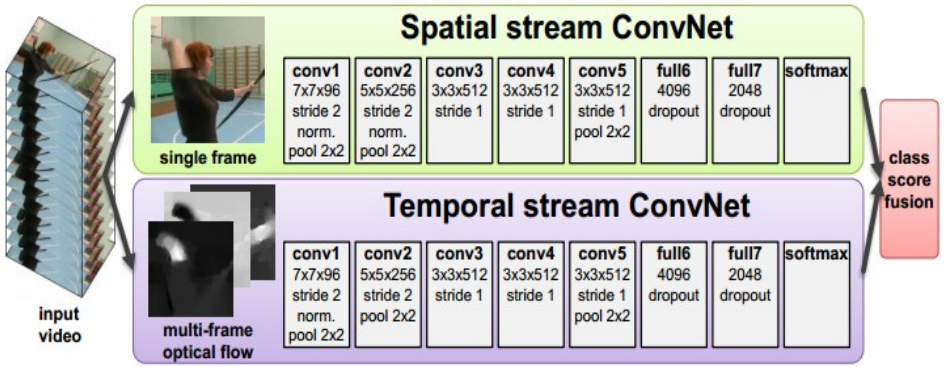
[Farabet et al., 2012]

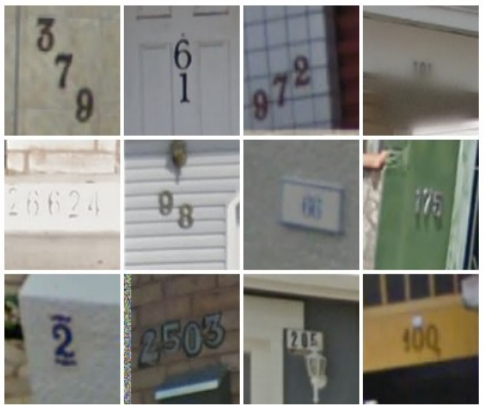
http://cs231n.github.io/

ConvNets are everywhere

[Taigman et al. 2014]







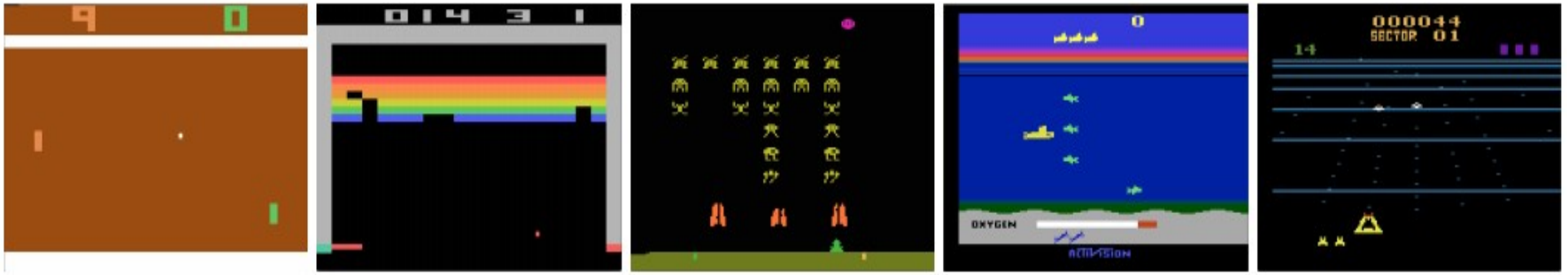
[Simonyan et al. 2014]

[Goodfellow 2014]

http://cs231n.github.io/

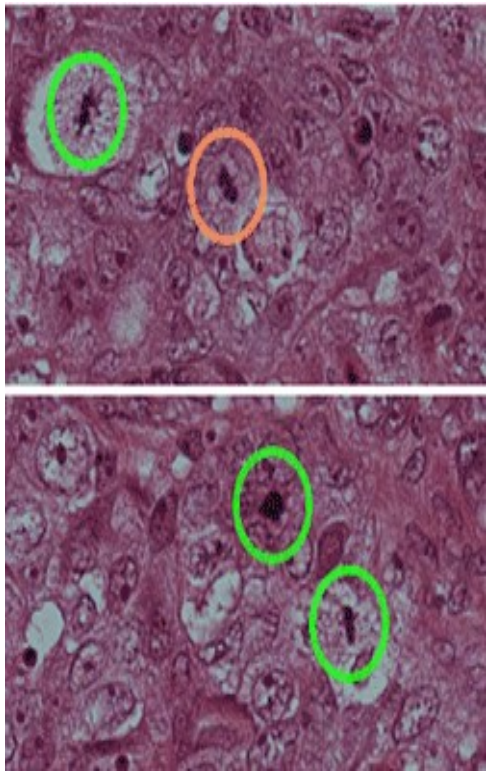
ConvNets are everywhere

[Toshev, Szegedy 2014]



[Mnih 2013]

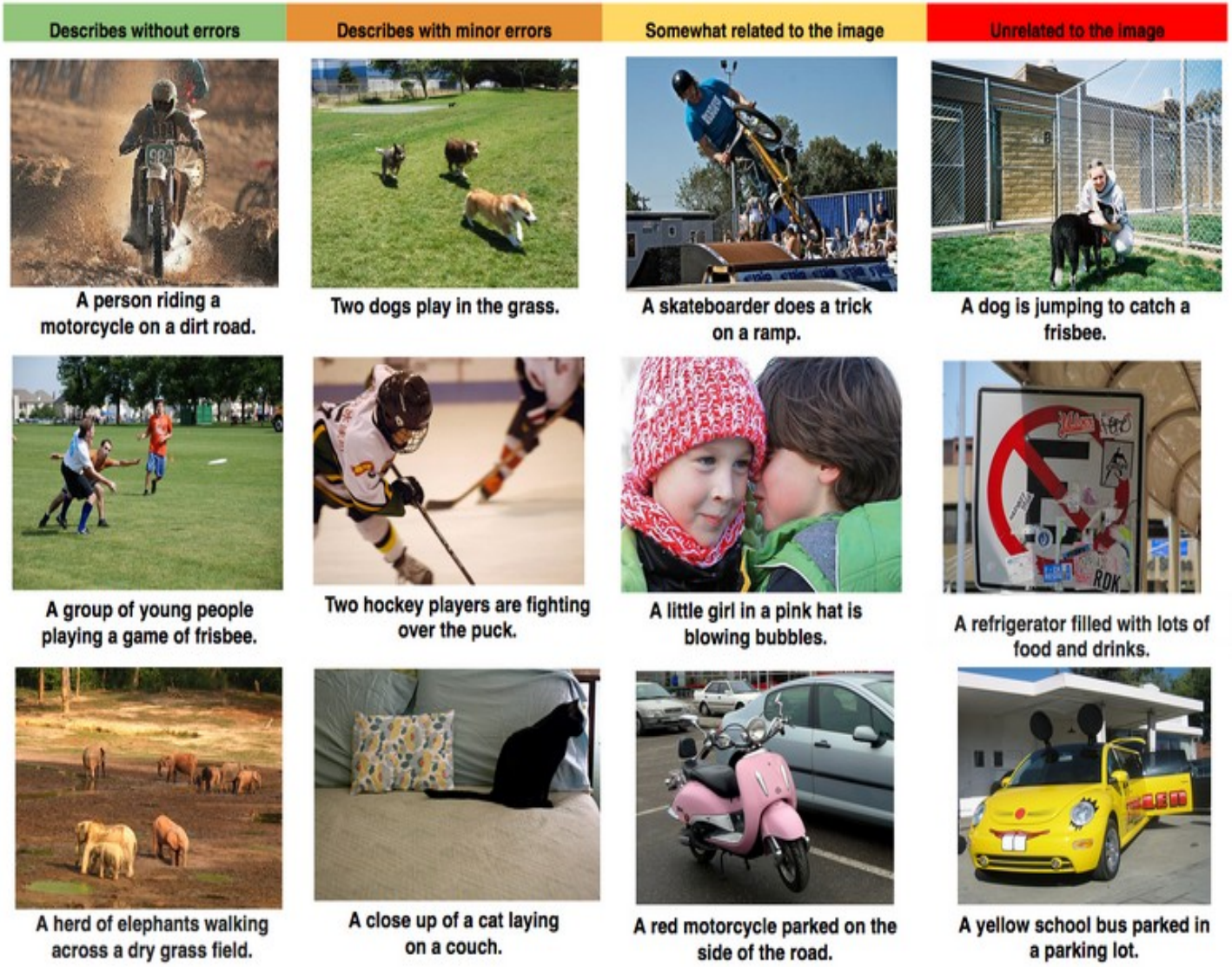
ConvNets are everywhere



[Ciresan et al. 2013]

[Sermanet et al. 2011]
[Ciresan et al.]

Image Captioning



[Vinyals et al., 2015]

Some More General Things:

- When do you stop training?
- How to tune your learning rate
- Regularization

When to stop training

- Fixed number of epochs
- When the training converged
  - How do you measure convergence?
  - Optimization becomes too slow
  - Validation score doesn’t improve
  - Combine with patience counter

Early Stopping in Keras

```
from keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(monitor='val_loss',
                               patience=2)

model.fit(X, y, validation_split=0.2,
          callbacks=[early_stopping])
```

patience: number of epochs with no improvement after which training will be stopped.

More info: https://keras.io/callbacks/

Early Stopping in Keras

```
from keras.callbacks import EarlyStopping

early_stopping = EarlyStopping(monitor='val_loss',
                               patience=2)

model.fit(X, y, validation_split=0.2,
          callbacks=[early_stopping])
```

validation_split: fraction of training data used for validation. Data is split from the end of the data and remains the same during training.

More info: https://keras.io/callbacks/

Learning Rate

- So many options:
- Fixed learning rate
- Start with large LR for n epochs, then set to smaller value
- Slowly decay learning rate over time
- Start with large learning rate, when patience counter expires lower the rate, repeat.

Learning Rate Decay in Keras:

```
def step_decay(epoch):
    lrate = 0.1
    if epoch > 100:
        lrate = 0.1 / (2. ** epoch)
    return lrate

lr = LearningRateScheduler(step_decay)
callbacks_list = [lr]
```

schedule: a function that takes an epoch index as input (integer, indexed from 0) and returns a new learning rate as output (float).

Reduce LR on Plateau

```
reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                              factor=0.2,
                              patience=5,
                              min_lr=0.001)

model.fit(X_train, Y_train, callbacks=[reduce_lr])
```

factor: How much to reduce the learning rate

patience: How many epochs with no improvement until lr is updated

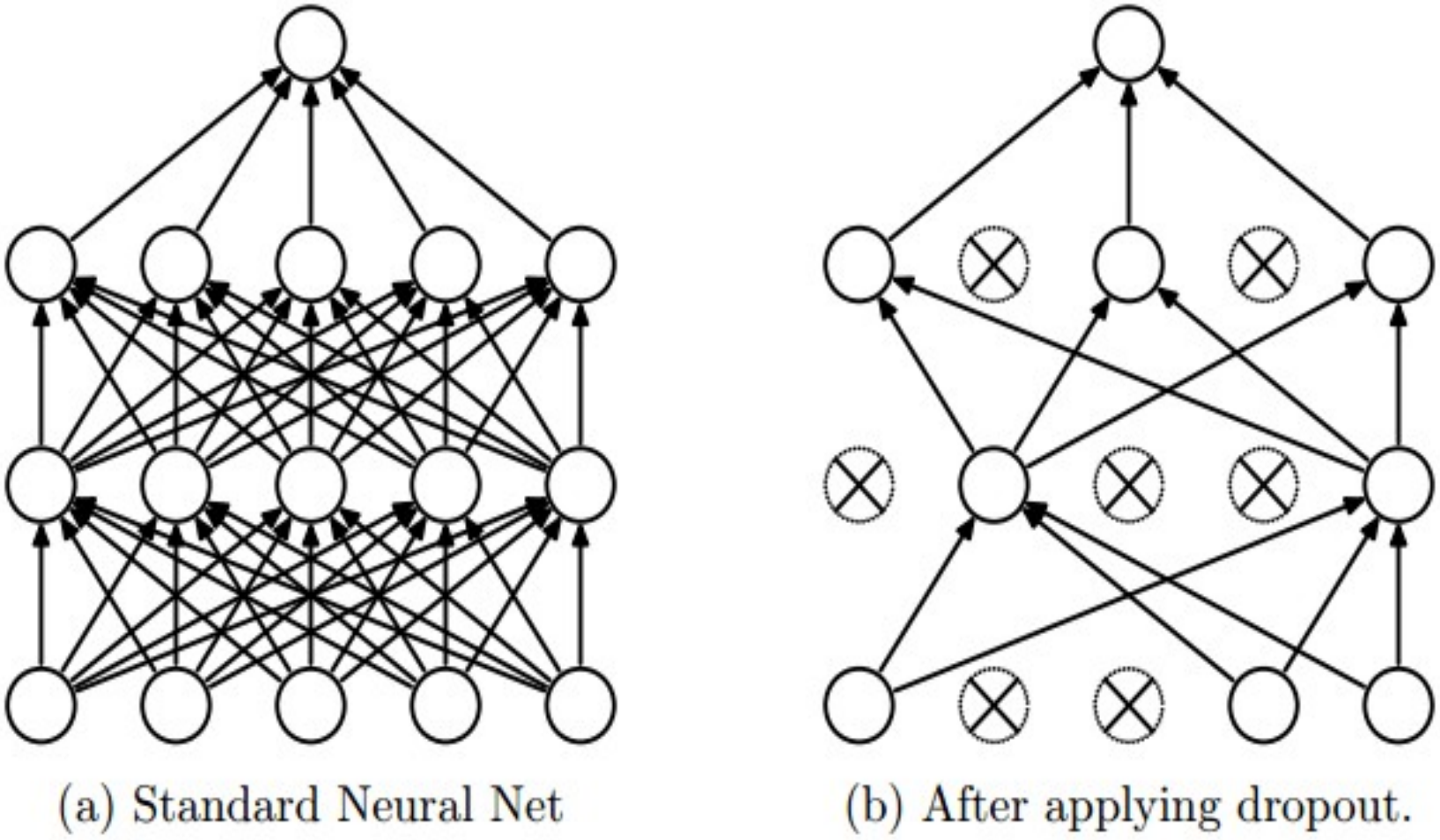
min_lr: When to completely stop

Regularization: dropout

- We have seen that randomness can be used very effectively for regularization
- Remember Random forest
- Can we do something similar for deep learning?

Regularization: **Dropout**

"randomly set some neurons to zero"



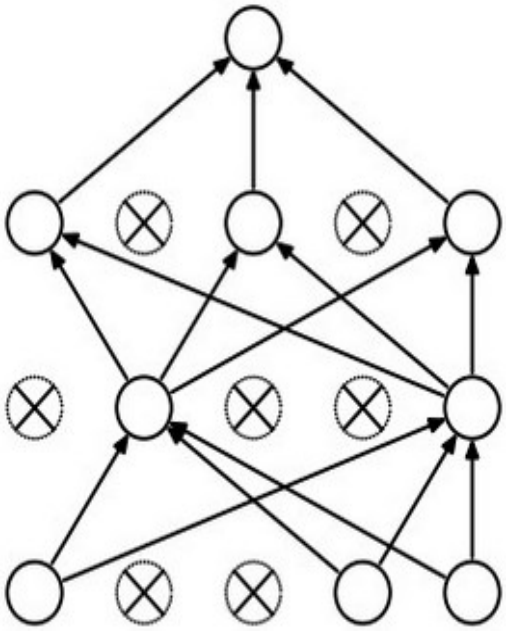
(a) Standard Neural Net

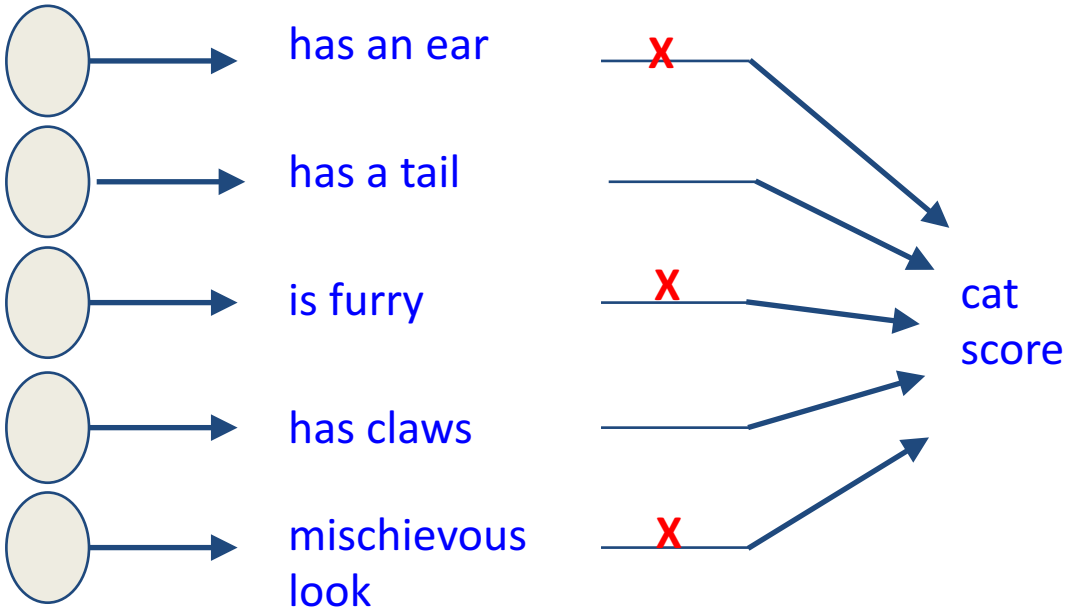
(b) After applying dropout.

[Srivastava et al., 2014]

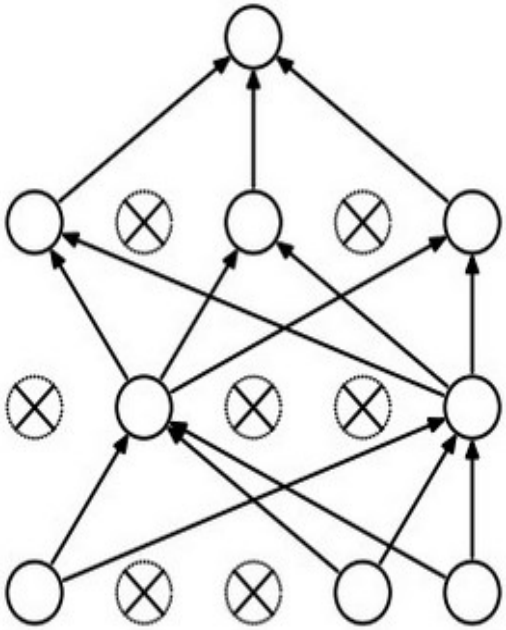
How could this possibly be a good idea?

Forces the network to have a redundant representation.





How could this possibly be a good idea?

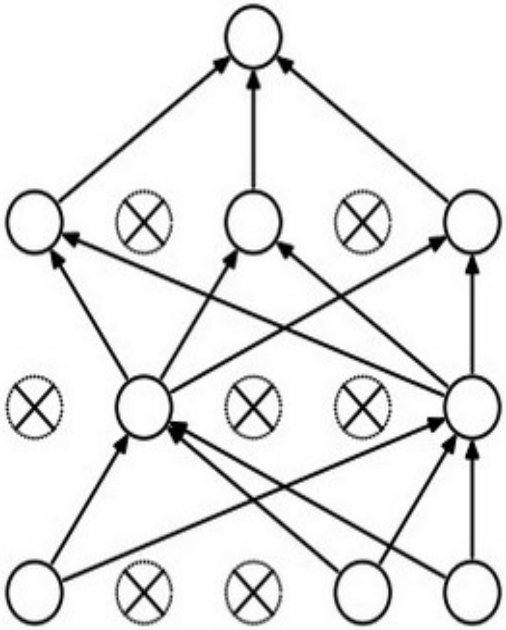


Another interpretation:

Dropout is training a large ensemble of models (that share parameters).

Each binary mask is one model, gets trained with only ~one update.

At test time….



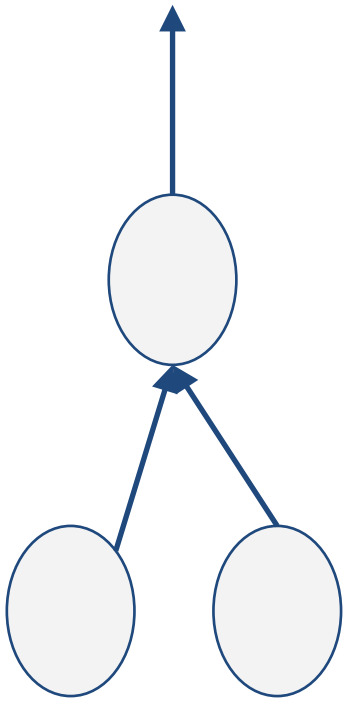
Ideally:
want to integrate out all the noise

Monte Carlo approximation:
do many forward passes with different dropout masks, average all predictions

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).

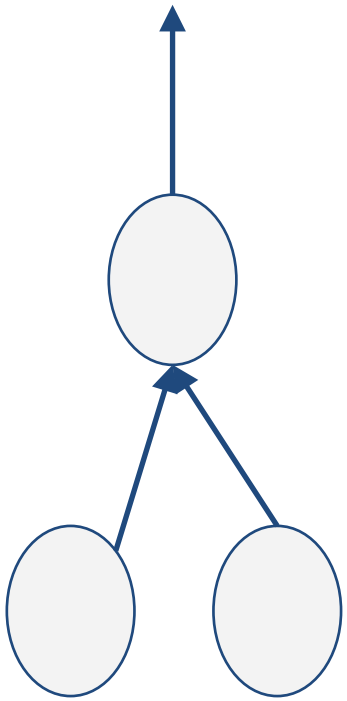


(this can be shown to be an approximation to evaluating the whole ensemble)

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



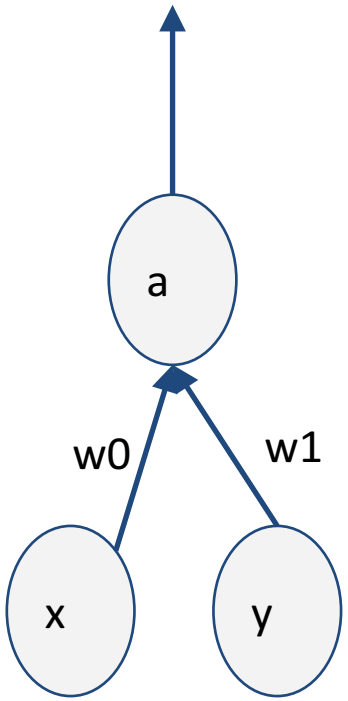
Q: Suppose that with all inputs present at test time the activation of this neuron is x.

What would its activation be during training time, in expectation? (e.g. if p = 0.5)

At test time….

Can in fact do this with a single forward pass! (approximately)

Leave all input neurons turned on (no dropout).



during test: **a = w0*x + w1*y**

during train:

```
E[a] = ¼ * (w0*0 + w1*0 +
            w0*0 + w1*y +
            w0*x + w1*0 +
            w0*x + w1*y)
     = ¼ * (2w0*x + 2w1*y)
     = ½ * (w0*x + w1*y)
```

=> Have to compensate by scaling
the activations back down by ½

Dropout in Keras

```
model = Sequential()
model.add(Conv2D(32, kernel_size=(3, 3),
                 activation='relu',
                 input_shape=input_shape))
model.add(Conv2D(64, (3, 3), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))

model.add(Flatten())
model.add(Dense(128, activation='relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes, activation='softmax'))
```

Dropout

- Dropout can be seen as ensemble averaging
- Each model in the ensemble is smaller than the original
- Reduces overfitting
- Introduces train and test mode

- Commen settings are:
  - 0.2 dropout on input layer
  - 0.5 dropout on hidden layers

Good Old L1/L2 Regularization

- Keras layer parameters

```
keras.layers.core.Dense(units, activation=None,
kernel_regularizer=None, bias_regularizer=None,
activity_regularizer=None, kernel_constraint=None,
bias_constraint=None)
```

- **activation**: Default is linear!
- **regularizer**: This is our standard L1 or L2 option
- Example: https://keras.io/regularizers/

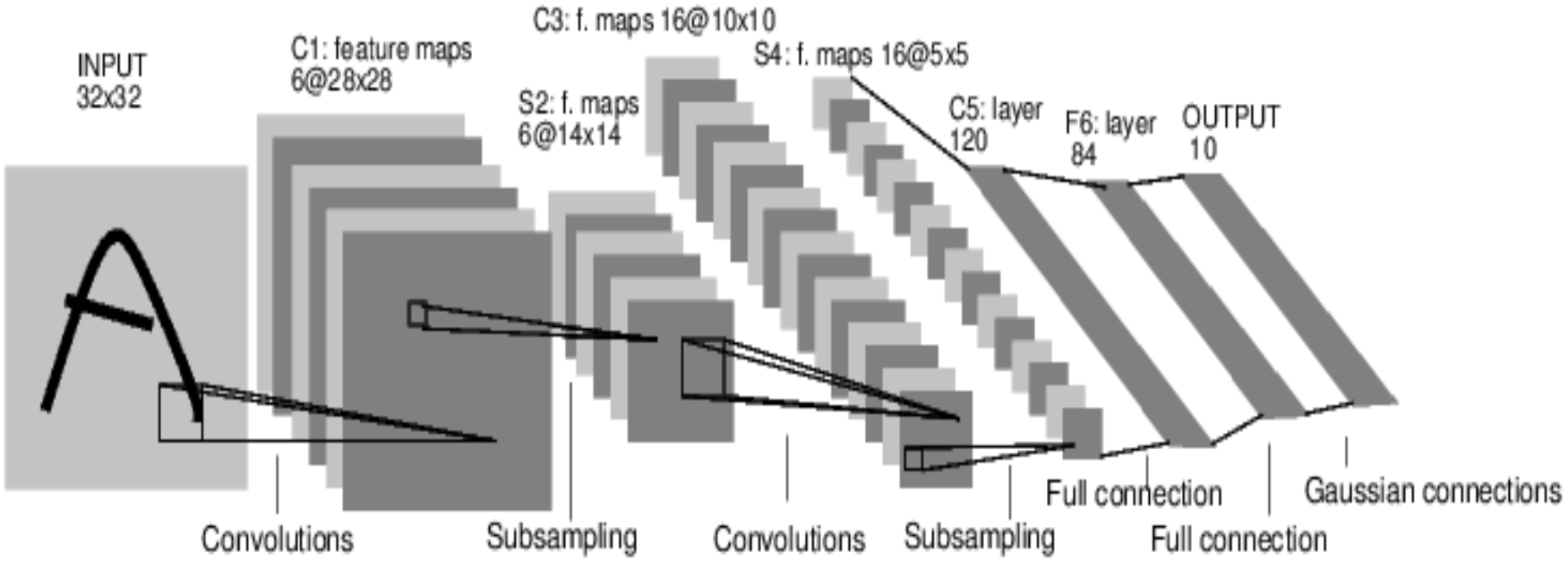
```
Dense(10, activation='relu,
      kernel_regularizer=regularizers.l2(0.01))
```

Case studies

- Don‘t go from scratch when you learn deep learning
- Look at example models
- Use tricks that have been shown to work

Case Study: LeNet-5

[LeCun et al., 1998]



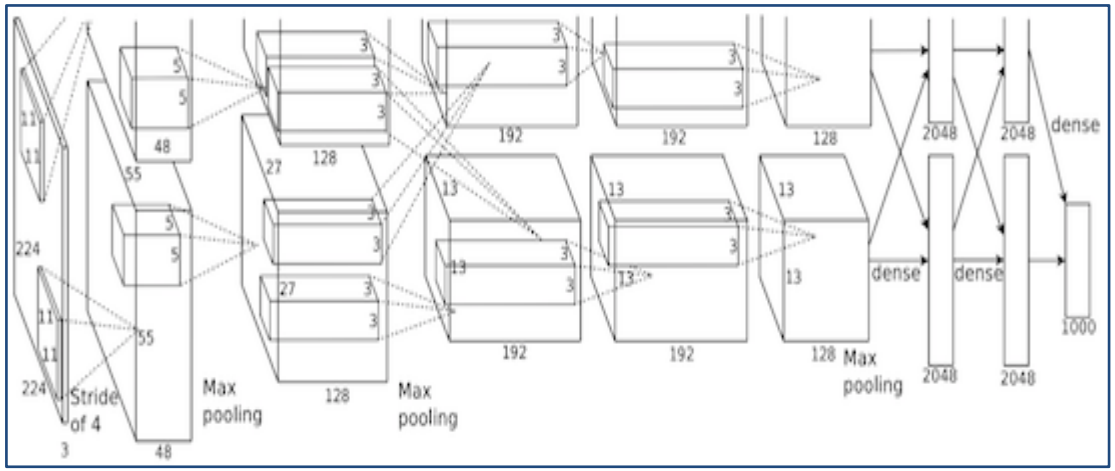
Conv filters were 5x5, applied at stride 1
Subsampling (Pooling) layers were 2x2 applied at stride 2
i.e. architecture is [CONV-POOL-CONV-POOL-CONV-FC]

A bit of history:

ImageNet Classification with Deep Convolutional Neural Networks

[Krizhevsky, Sutskever, Hinton, 2012]

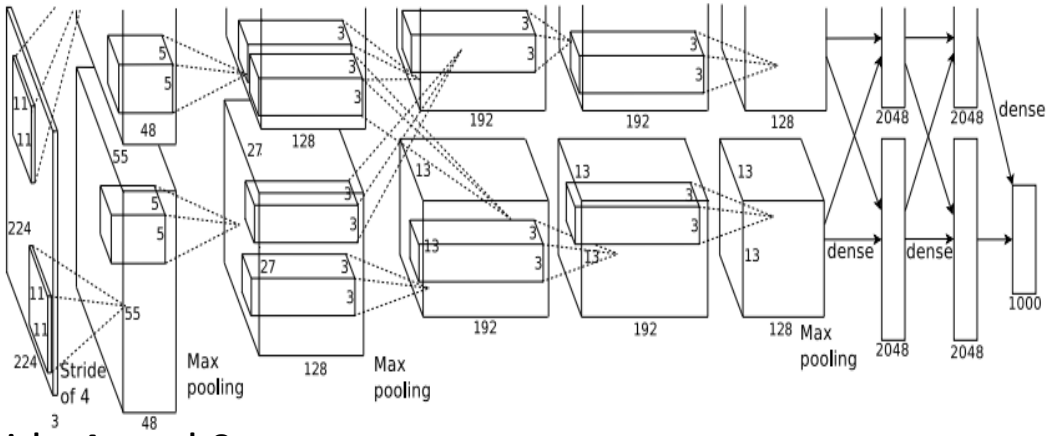
IMAGENET



"AlexNet"

Case Study: AlexNet

[Krizhevsky et al. 2012]



Full (simplified) AlexNet architecture:
[227x227x3] INPUT
[55x55x96] CONV1: 96 11x11 filters at stride 4, pad 0
[27x27x96] MAX POOL1: 3x3 filters at stride 2
[27x27x96] NORM1: Normalization layer
[27x27x256] CONV2: 256 5x5 filters at stride 1, pad 2
[13x13x256] MAX POOL2: 3x3 filters at stride 2
[13x13x256] NORM2: Normalization layer
[13x13x384] CONV3: 384 3x3 filters at stride 1, pad 1
[13x13x384] CONV4: 384 3x3 filters at stride 1, pad 1
[13x13x256] CONV5: 256 3x3 filters at stride 1, pad 1
[6x6x256] MAX POOL3: 3x3 filters at stride 2
[4096] FC6: 4096 neurons
[4096] FC7: 4096 neurons
[1000] FC8: 1000 neurons (class scores)

Details/Retrospectives:
- first use of ReLU
- used Norm layers (not common anymore)
- heavy data augmentation
- dropout 0.5
- batch size 128
- SGD Momentum 0.9
- Learning rate 1e-2, reduced by 10 manually when val accuracy plateaus
- L2 weight decay 5e-4
- 7 CNN ensemble: 18.2% -> 15.4%

Summary

- CNNs are designed for image processing
- Convolutional layers preserve pixel neighborhood structure
- Small kernels can be powerful when stacked
- Max pool reduces parameters and introduces limited translation invariance

- General techniques:
  - When to stop
  - Learning rate tuning
  - Dropout regularization