Practical aspects of Deep Learning

BOSTON UNIVERSITY
MACHINE INTELLIGENCE COMMUNITY

Presenters: Julius, Curtis
9/25/2019

Deep Learning needs many tricks to work

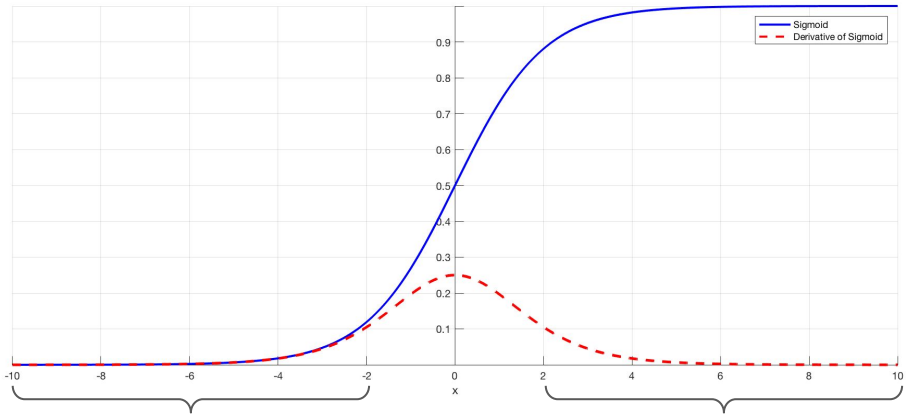
1. Data
    a. Labeled / unlabeled data, classification / regression
2. Model
    a. Layers, activations, loss function
    b. Weight initialization
3. Hyperparameters
    a. Learning rate, batch size, etc.
4. Optimization
    a. Stochastic gradient descent, batch gradient descent
    b. Momentum, RMS prop, Adam optimization
5. Regularization
    a. L2 regularization, Batch norm, (prevent overfitting)

Vanishing / Exploding gradients

1. During backpropagation, the gradient is calculated using the chain rule
2. Under the chain rule, weights in the initial layers are described by a long product of terms
3. If your network is very deep, this can lead to the derivatives exponentially collapsing to 0 or infinity

$$\lim_{x \to \infty} 0.5^x = 0$$

$$\lim_{x \to \infty} 1.5^x = \infty$$

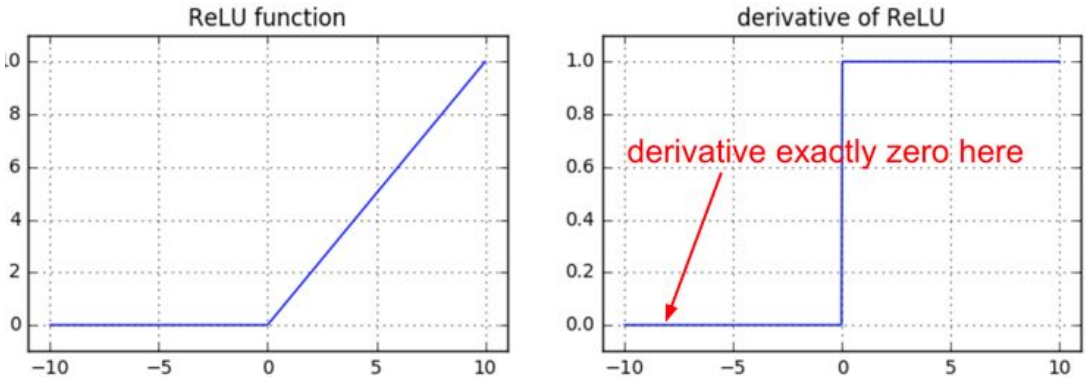


Small derivative

Small derivative

Fixes for Vanishing / Exploding gradients

1. Use ReLU activation function
   a.



2. Start with good weight initialization
   a. If you start with gradients close to optimal, then it will stay optimal

Weight Initialization

1. We need to initialize the neural networks parameters
2. How can we initialize them?
    a. There are many different ways doing so
    b. Obviously, we want them to be initialized as close as possible to the final weight distribution

Weight Initialization

1. Ideally, we would like to have the weights closest to the final distribution of weights
    a. That way, the network will not have to change much to get to the global minimum
2. Pitfall - all zero initialization
    a. Initialize all weights to zero so they can learn to be positive or negative
        i. Problem - Symmetric Weights
            1. Since they are initialized at zero, their gradients will be the same
3. Random Small Number
    a. Fixes the same symmetric weight issue, but still no details on how close to zero the parameters should be
4. What will work - Xavier Initialization

Xavier Initialization

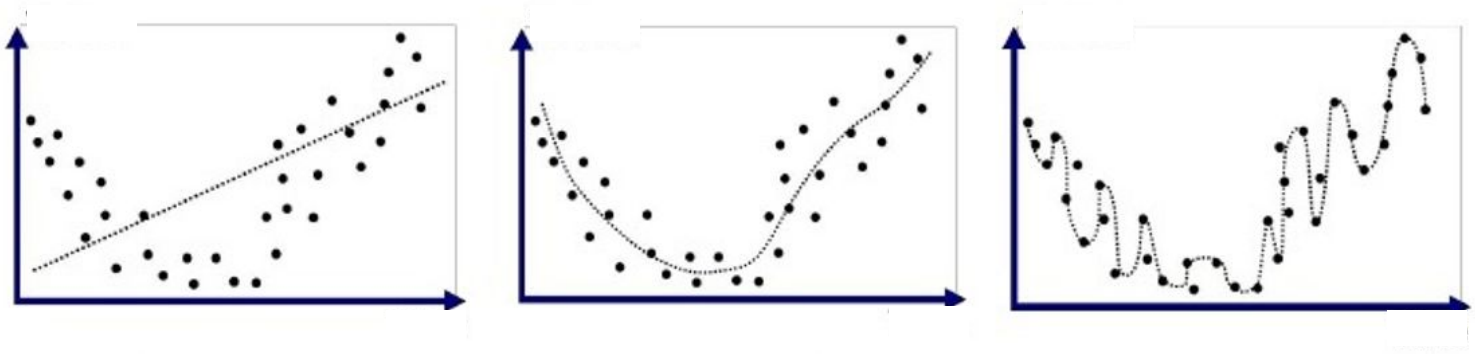
$$Var(W) = \frac{2}{n_{in} + n_{out}}$$

W = uniform(-var, var)

Overfitting and Regularization

The purpose of regularization is to prevent overfitting by limiting extreme weights.

Similar discussion: over-parameterization vs. under-parameterization



L2 regularization

- Sum of square of all weights

$$\sum w_i^2$$

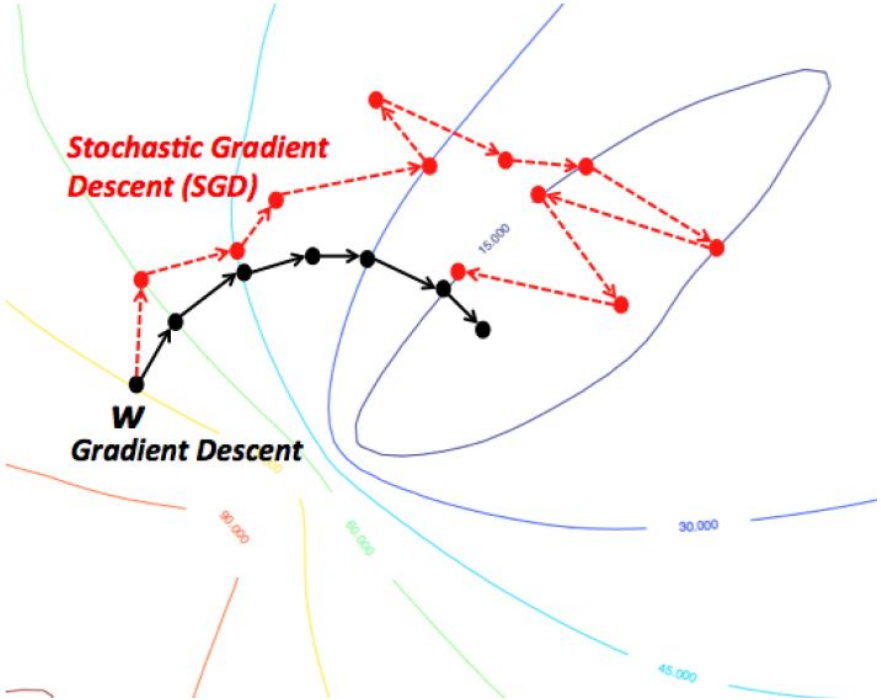
- We want to minimize the “complexity” of the model by avoiding having large weights.
  - The square in the regularization penalizes the large weights more heavily thus optimizing this would reduce the weights of the model.

Optimization Algorithms

Stochastic gradient descent (SGD)

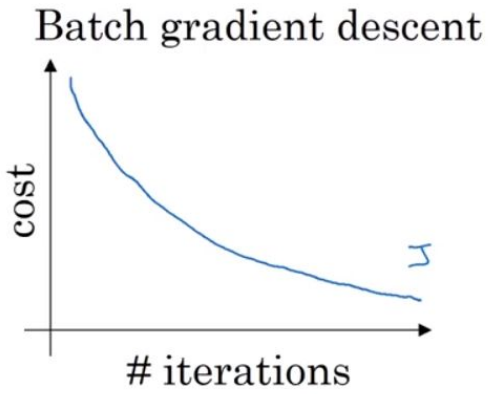
1. In regular gradient descent you sum the loss from every training point x,y to obtain some cost J. After going through every example you backprop.
2. Stochastic gradient descent instead performs backprop at every training point x,y, or after a couple training points rather than only after evaluating the cost for the entire training set

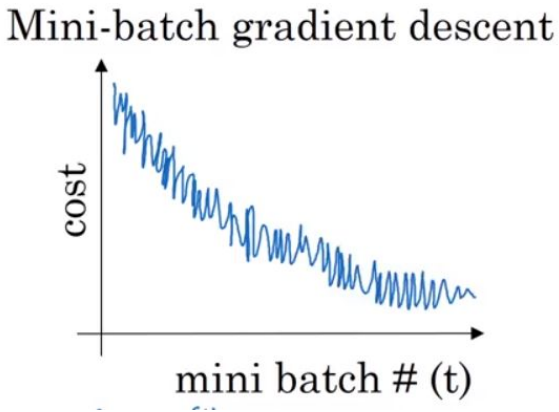
Stochastic gradient descent (SGD)



Mini-batches

1. Compute the cost for a subset of examples, then perform backprop
2. Average the losses for the minibatch of samples (fixed size)
3. Steps may have smaller noise than Stochastic GD

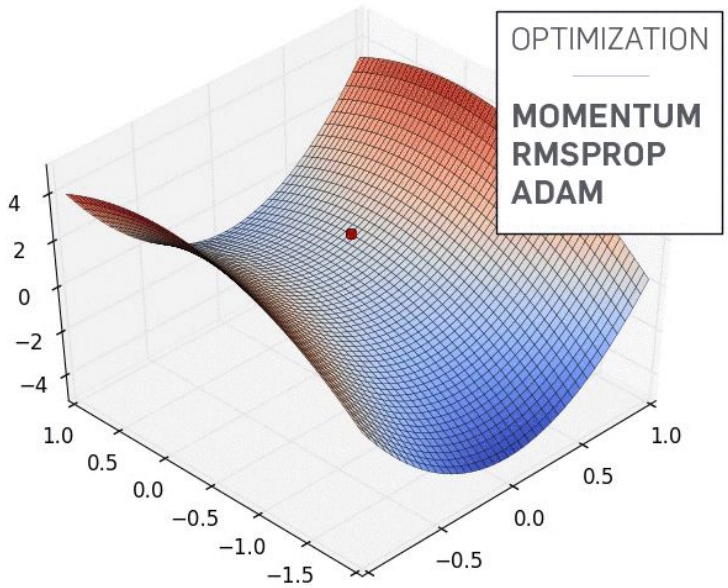




Batch size comparison

	Batch GD	Mini batch GD	Stochastic GD
Batch size	Whole dataset	8,32,128,512, etc	1
Updates per epoch (iteration over the whole dataset)	Slow (1)	Medium (samples / batch size)	Fast (1 for each sample)
Advantage with GPU computation	High	Medium	Low
Update stability	consistent	Small noise	Very noisy

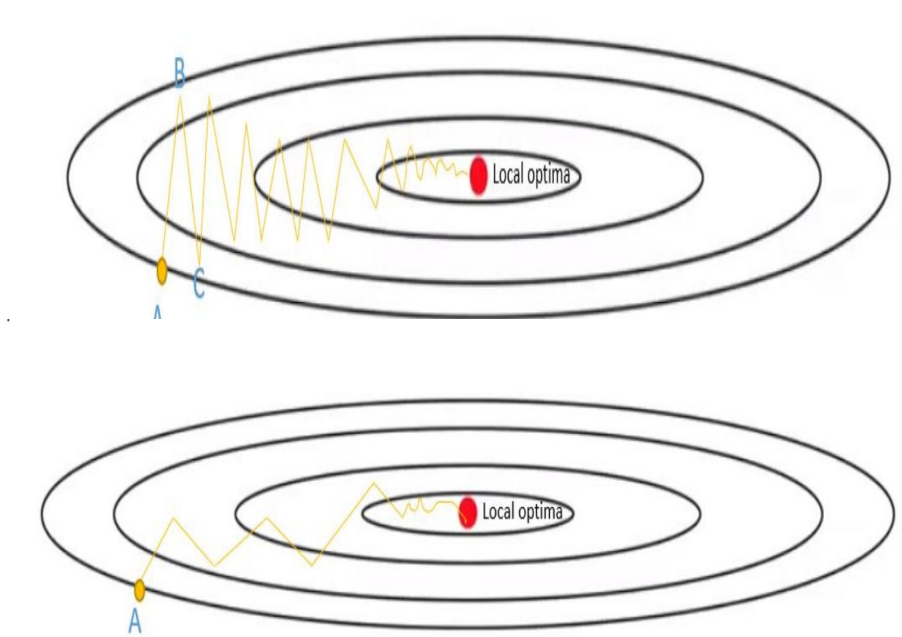
Optimizers with gradient descent



Gradient descent with momentum

With SGD we may update in the wrong direction, effectively oscillating towards the local optima due to large steps. (top image)

Instead we may want to shrink the “amplitude” of these oscillations to effectively traverse directly towards the local optima with fewer iterations (bottom image)



Gradient descent with momentum

Keep a fraction of the previous gradient at the previous iteration, and add that to the new iteration.

We want to average out the vertical oscillations.

By computing an exponential average, we keep track of the historical velocity of the parameter gradient, and apply acceleration of the current gradient to get the momentum term.

$$V_{\partial W} = \beta V_{\partial W} + (1 - \beta)\partial W \qquad V_{\partial b} = \beta V_{\partial b} + (1 - \beta)\partial b$$

Then update the parameters by the momentum term

$$W := W - \alpha V_{\partial W} \qquad b := b - \alpha V_{\partial b}$$

RMSprop

When we have a gradient valley, or the gradient of one parameter is much larger than the other, if we do simple gradient descent we get large updates to parameters that only need to change a short distance and smaller updates to those that need to change a larger distance.

RMSprop solves that problem by essentially normalizing the updates to the parameters. That means it boosts the update to small gradients and flattens gradients large gradients.

We keep an exponential weighted average of the squared gradient. This gives us a stable history of the gradients, as we cannot assume the shape of the valley from the slope at a single point.

We divide the parameters by the square root of the squared gradient. The effect of squaring and rooting gets rid of the sign (+-) of the gradient, and only keeps the magnitude. Dividing by the magnitude has the effect of normalizing.

$$S_{\partial W} = \beta S_{\partial W} + (1-\beta)\partial W^2 \qquad S_{\partial b} = \beta S_{\partial b} + (1-\beta)\partial b^2$$

$$W := W - \alpha \frac{\partial W}{\sqrt{S_{\partial W}}} \qquad b := b - \alpha \frac{\partial b}{\sqrt{S_{\partial b}}}$$

Adam optimization

All it does is combine Momentum with RMSprop $W := W - \alpha \frac{V_{\partial W}}{\sqrt{S_{\partial W}} + \epsilon}$

We add a very small epsilon to prevent divide by zero.

$$V_{\partial W} = \beta_1 V_{\partial W} + (1 - \beta_1)\partial W$$

$$S_{\partial W} = \beta_2 S_{\partial W} + (1 - \beta_2)\partial W^2$$

To compute the momentum term and the RMS term, here are the same equations from before but have the hyperparameter beta relabled.

One can additionally add bias correction for the first few iterations of the exponential average.

t = 0, $V_{\partial W}^{corrected} = V_{\partial W} / (1 - (\beta_1)^t)$

This has the effect of removing the influence from the initial setting of the term to 0.

Takeaway: Usually go with Adam optimization, unless you desire certain properties during your training.

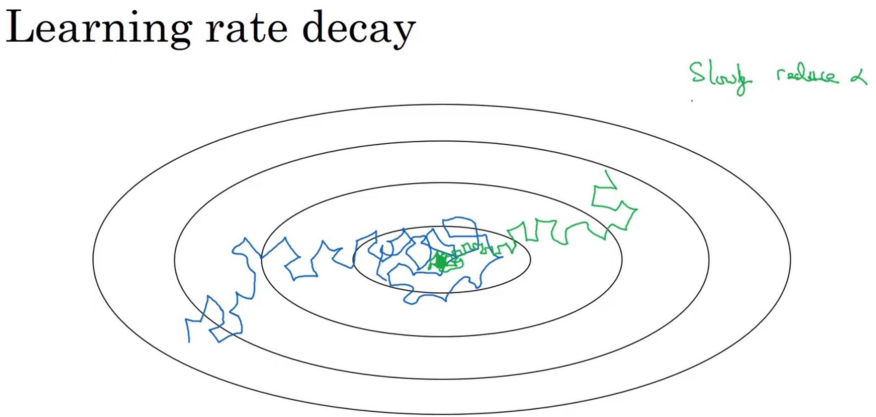
Hyperparameter tuning

How to pick hyperparameters

1. Start with the common/default hyperparameters first
    a. Those detailed in papers and in deep learning frameworks
2. Try tuning more important hyper parameter values first
    a. Use appropriate scaling for hyper parameters
        i. Ex. The learning rate should be picked on a log scale $\alpha = 10^{-4*random()}$
            1. 0.01, 0.001, 0.0001
            2. (equal probability to choose 0.1 vs 0.001)

Learning rate decay

By reducing the learning rate when the parameters are close to optimal, the model will converge better



Batch norm

- We normalize input feature, why not the same for the hidden layers?
  - Normalize hidden layers helps reduce covariate shift thus increase the stability of the network and the later layers of the network is more independent in learning*.
  - Batch norms works by taking the activation value of the previous layer, subtract the batch mean and divides by batch standard variation.

Note: The idea that batch norm helps reduce internal covariate shift is disputed in *How Does Batch Normalization Help Optimization?* [Santurkar, NeurIPS'18]

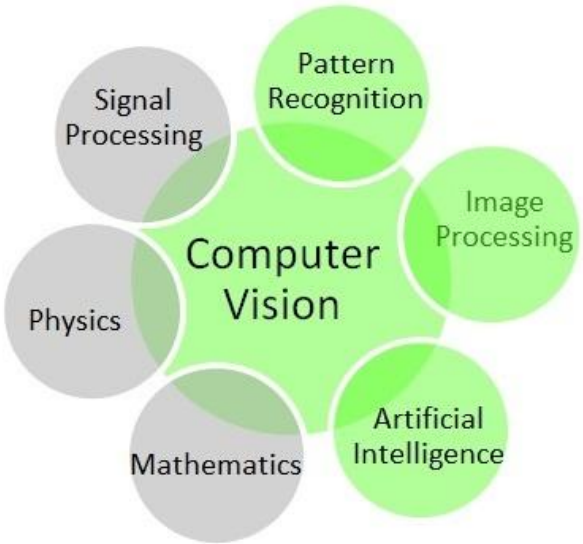
BOSTON UNIVERSITY
MACHINE INTELLIGENCE
COMMUNITY

Computer Vision

Presenter: Curtis

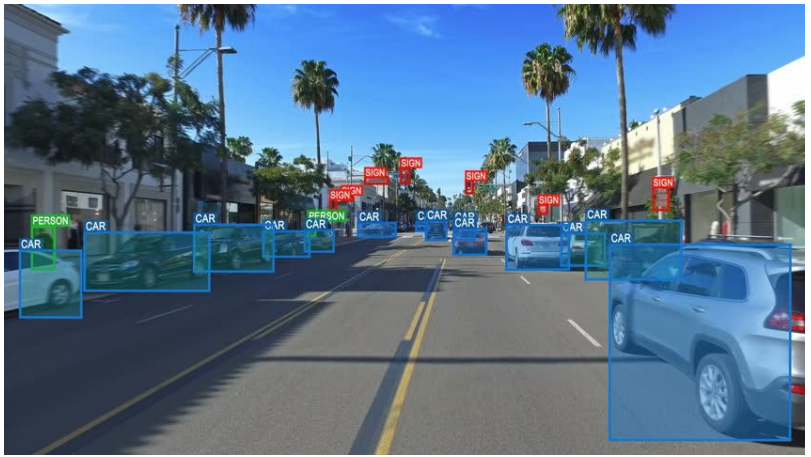
What is computer vision?

- The scientific field that deals with how computers can be made to gain high-level understandings of digital images and videos.
- For engineering, it seeks to automate the tasks that the human visual system can do
- Acquire → Process → Analyze → Understand



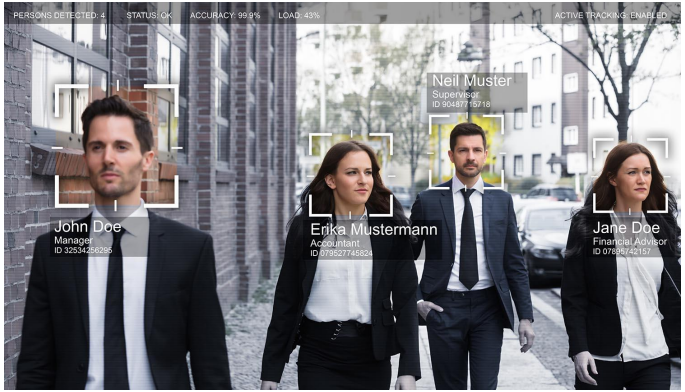
Computer Vision Applications - Autonomous Driving

1. Object detection
    a. Cars
    b. Signs
    c. Pedestrians, trees, etc.
2. Path Planning
    a. Where to go based on current situation of objects ahead.



Computer Vision Applications - More Uses

1. Face Detection
    a. Determine if there IS a person in the frame
    b. If needed, determine WHO the person is in the frame
2. Classifiers
    a. Give the model an image of an item
    b. Then it results to what object is in the image



Label	Description	Examples
0	T-Shirt/Top	
1	Trouser	
2	Pullover	
3	Dress	
4	Coat	
5	Sandals	
6	Shirt	
7	Sneaker	
8	Bag	
9	Ankle boots	

Convolutional Neural Networks

Image Data

- Images are commonly represented in code as a 3D array of pixels. Here, we notice 3 represents RGB values
- In vanilla neural networks, we would simply flatten this 3D array into a 3072 length vector. However, by doing this, we lose spatial correlation between pixels close to other pixels

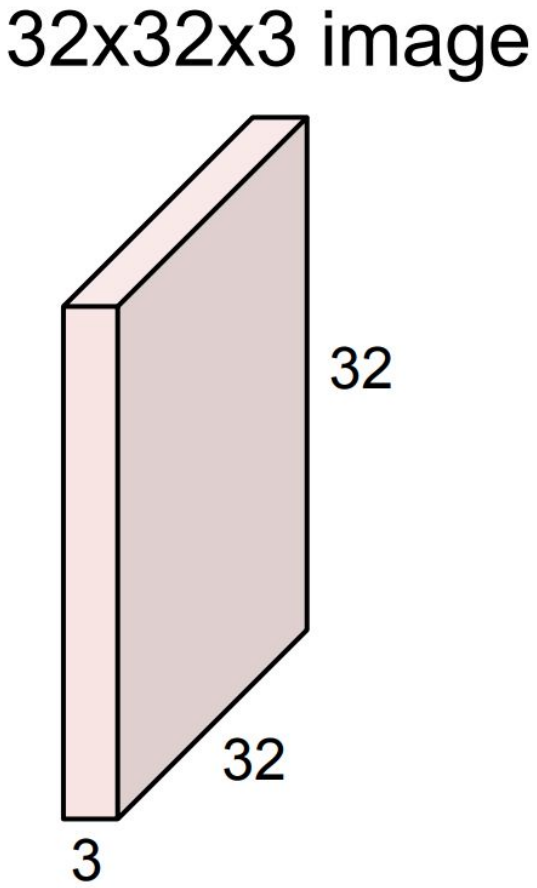
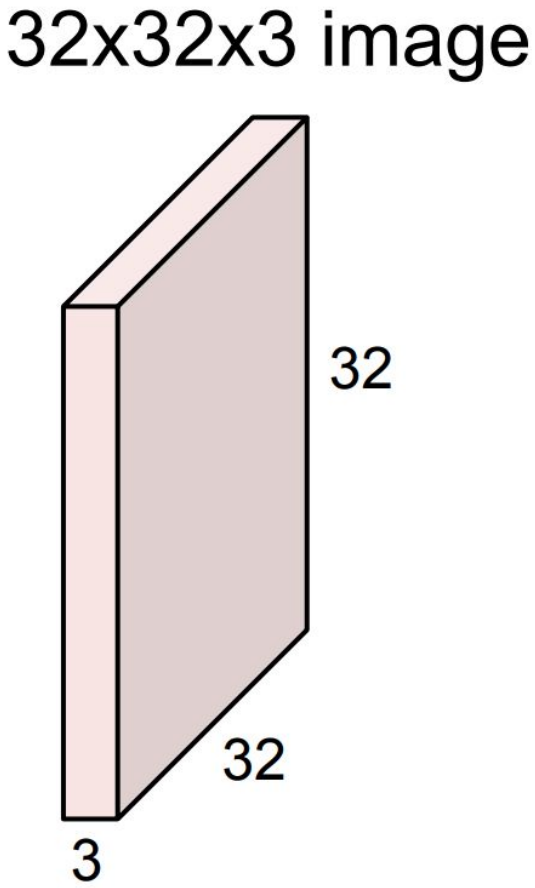


Image Data

- In 2012 a paper called AlexNet out competed state of the art image classification models through the usage of filters

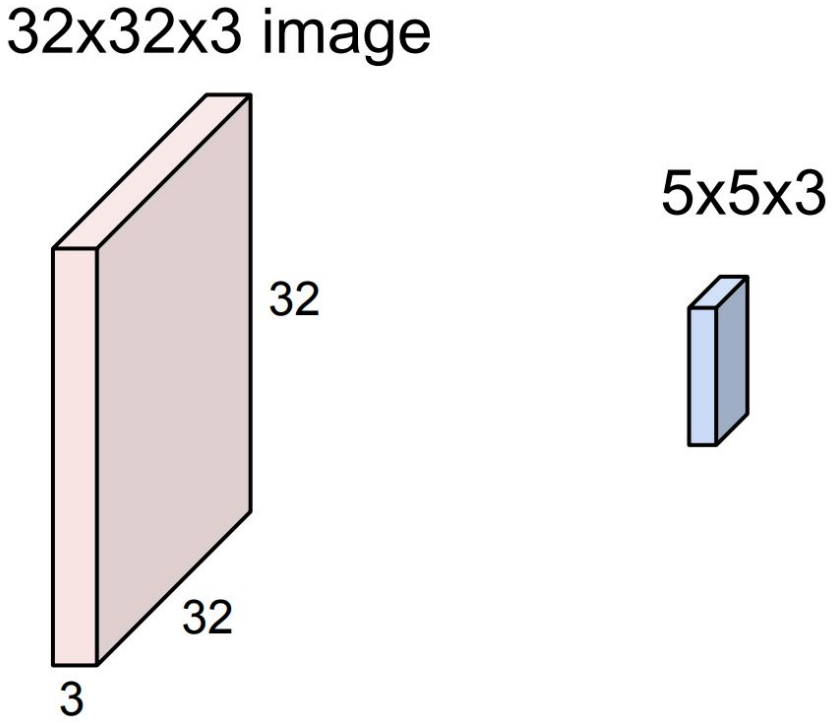


Filter

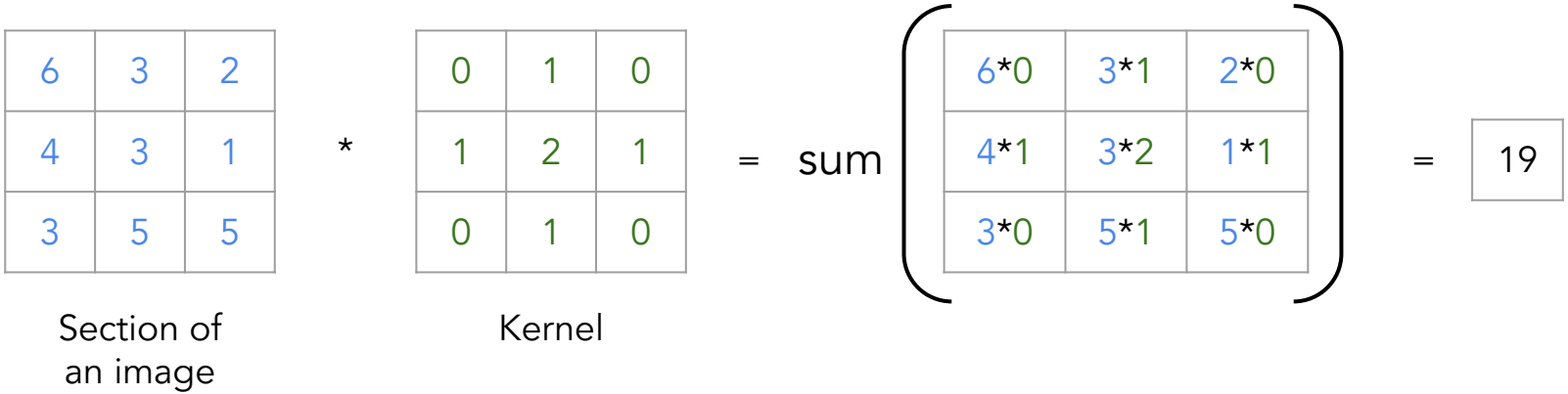
- Filter: a small matrix used for feature detection on an image
- Usage
  - Superimpose the kernel over a section of an image
  - Do element-wise multiplication between the weights in the kernel and the values in the image
  - Record the sum of the multiplications
  - This process is known as a dot product

Filter

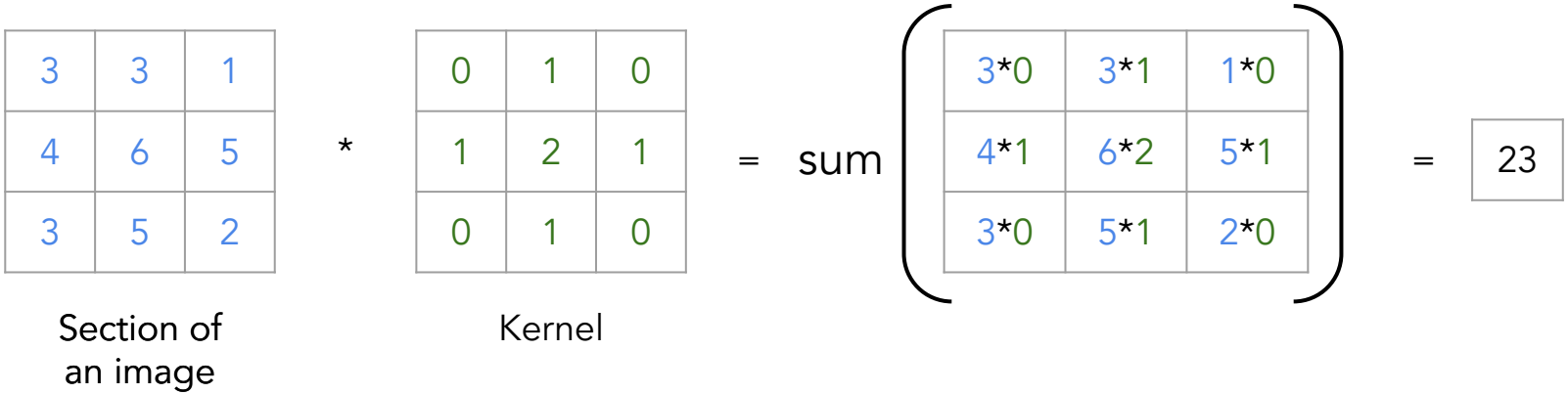
- The filter (our 5x5x3 sliding hyper parameters) are used to take dot products from sections of an image
- Remember, dot products return ONE number



Kernel example



Kernel example (cont.)



This image section contains the same values as before, but they have been rearranged, resulting in a greater activation with this kernel

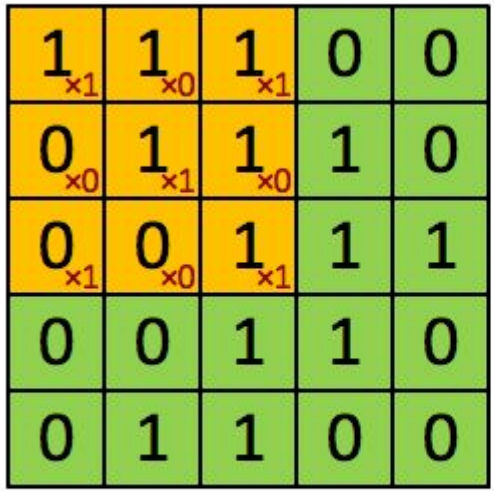
Stride

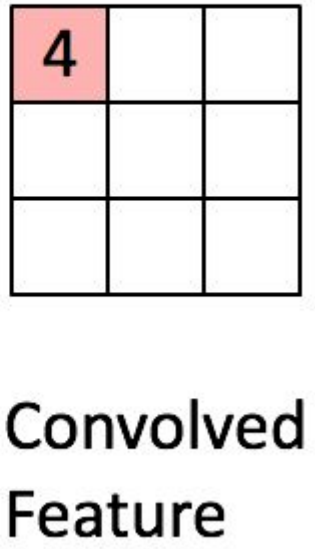
- Filters slide across inputs through a stride parameter
- Stride represents how far does a filter jump between pixels

Example Convolution

Example: Multiply the 5x5 image by a 3x3 filter with weights:
1 0 1
0 1 0
1 0 1

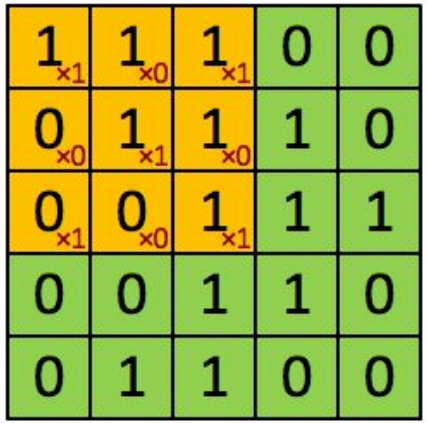
The output?
Sum of weight times part of image to a single number.



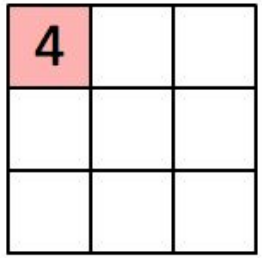


Stride (cont.)

- In this image, we have an input layer of size 5x5 and a filter of size 3x3 using a stride of 1
- Note that images and filters are USUALLY 3 dimensional meaning our 3x3 filter will usually be 3x3x3 but this is a simplification



Image



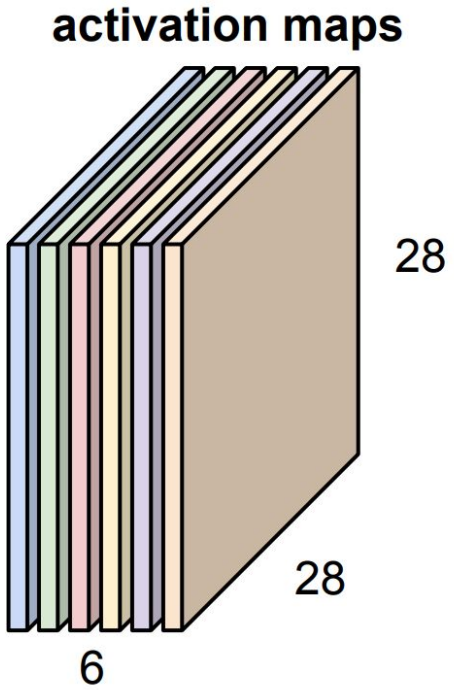
Convolved Feature

Stride (cont.)

- Our convolved feature is also known as an activation map
- A convolutional layer will typically have multiple activation maps stacked on top of each other creating depth or a 3D array

Conv Layer

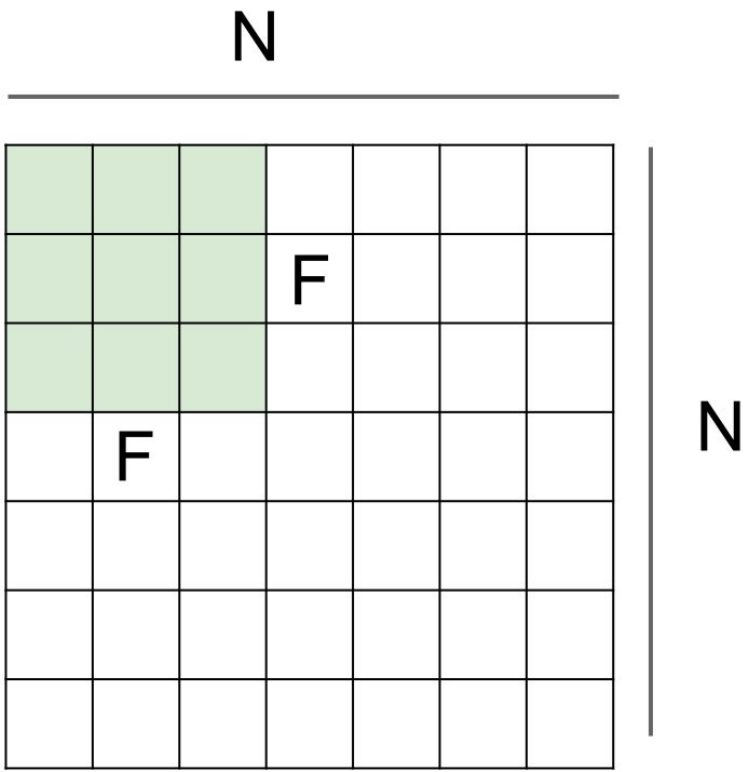
- In this example, we have 6 activation maps each created through a different filter with its own set of weights and biases



Stride (cont.)

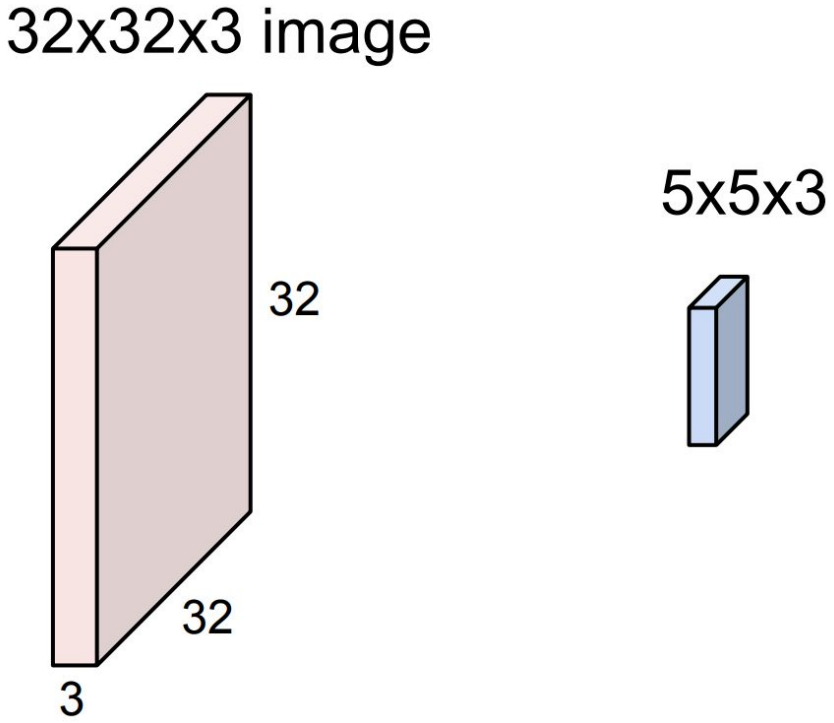
- We can calculate the shape of an activation map through the following equation:

$$output\_size = \frac{N-F}{stride} + 1$$



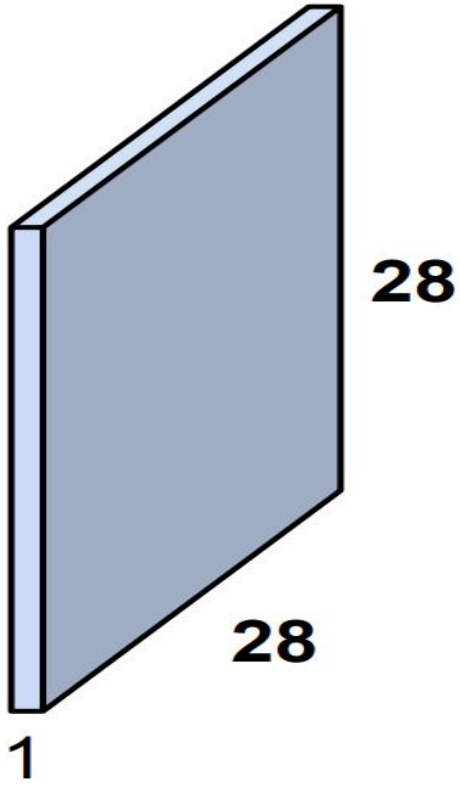
Dimensionality Practice

- What would be the output size of a 5x5x3 filter with a 32x32x3 image and a stride of 1?
- Remember, (N - F) / Stride + 1



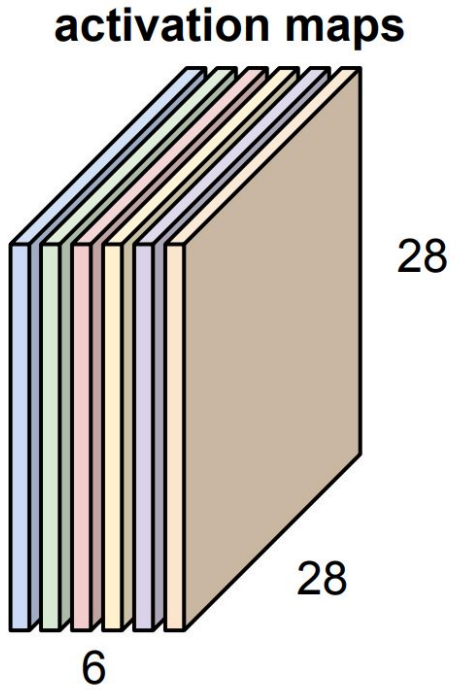
Dimensionality Practice

- (32 - 5) / 1 + 1 = 28
- Now let's say we had a stride of 2,
  - (32 - 5) / 2 + 1 = 14.5
  - Fractional size means the filter hangs off the input
  - We wouldn't use this stride value consequently



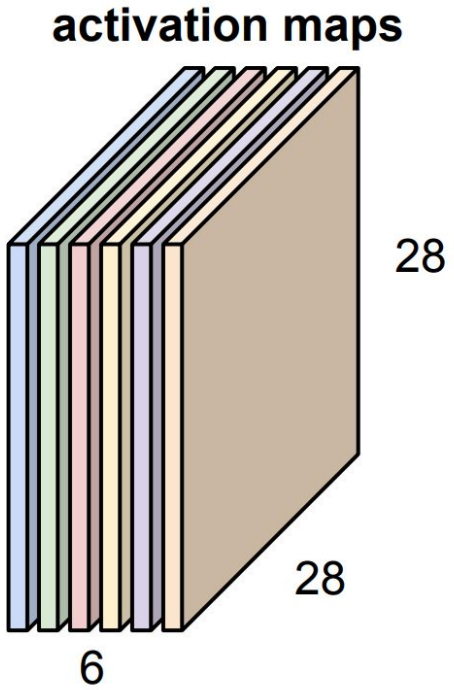
Shrinkage

- Let's say we had an input of 32x32x3 and 6 filters all of size 5x5x3 with a stride of 1.
- Notice how our activation maps shrank
- Less information encoded



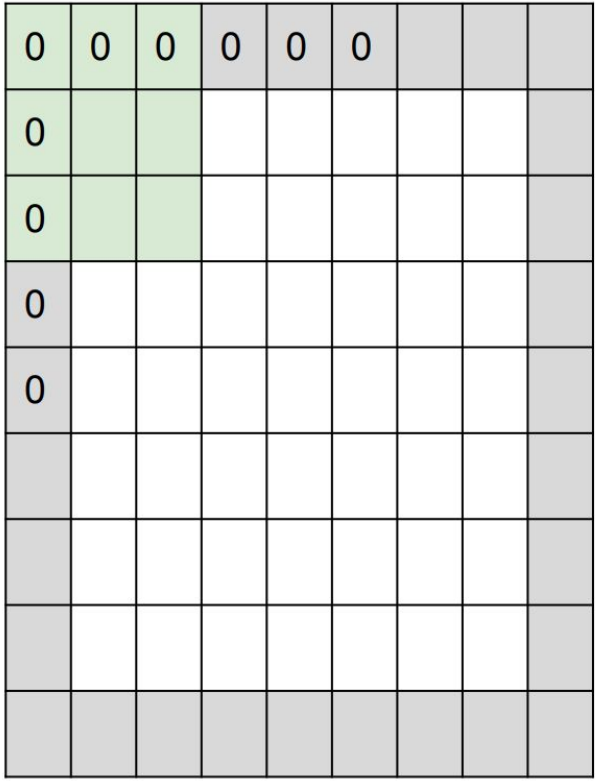
Shrinkage (cont.)

- Shrinkage can make activation maps shrink really quickly, which can lead to less detailed convolutional layers
- To resolve this, let's introduce padding



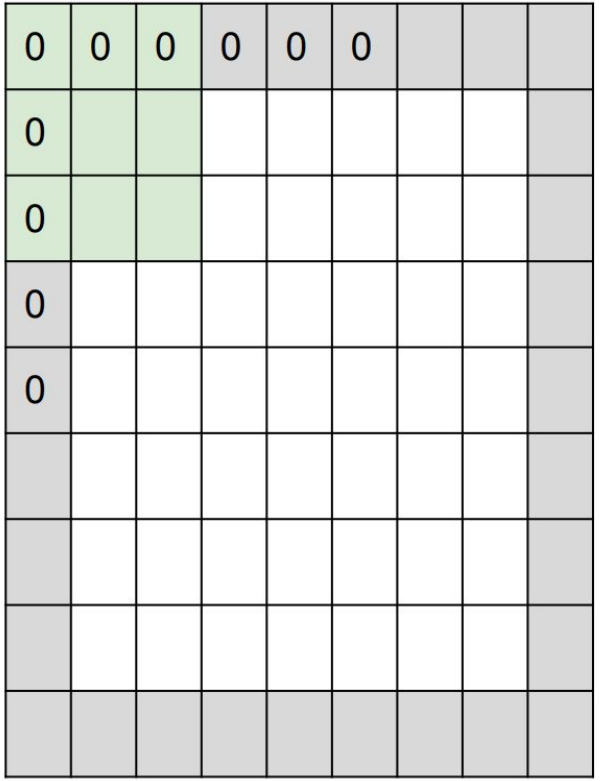
Shrinkage (cont.)

- Before padding:
  - 7x7x3 input, 3x3x3 filter, stride of 1 creating a (7 - 3) / 1 + 1 = 5x5x5 sized output
- After padding:
  - 9x9x3 input, 3x3x3 filter, stride of 1 creating a (9 - 3) / 1 + 1 = 7x7x3 sized output which maintains the same size as our input

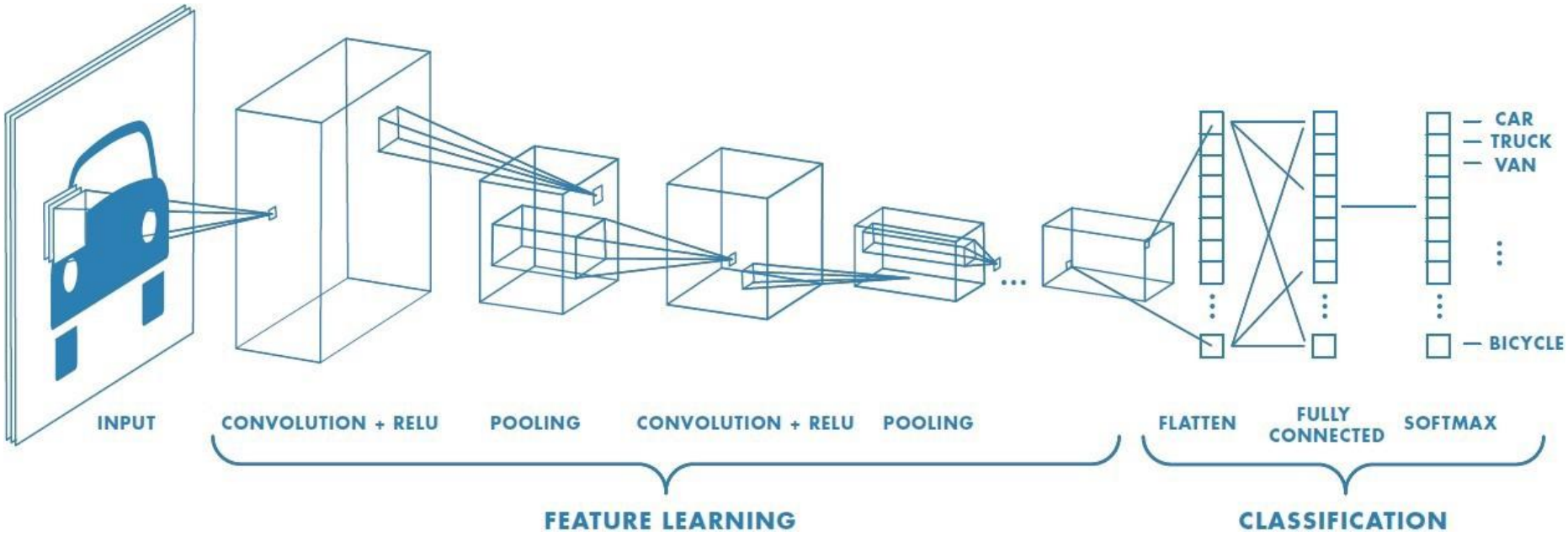


Intentional Shrinkage

- Now, let's say you want to shrink your inputs to reduce operations.
- You can do this by either increasing the stride (N - F)/Stride + 1
- Alternatively, you can use a pooling layer

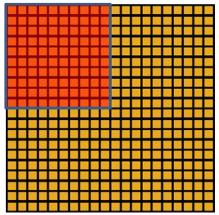


Convolutions



Additional Layers

Pooling Layers



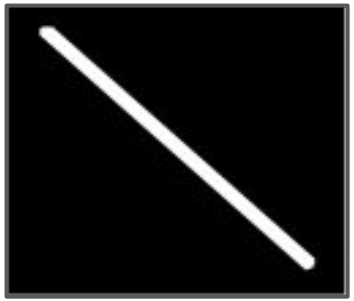
Convolved feature

Pooled feature

- Limitation of output of Convolutional Layers:
  - Record the precise position of features in the input
  - Small movements in the position of the feature in the input image will result in a different feature map
- Solution: Pooling Layers
  - Lower resolution version of input is created with large and important structure elements preserved
  - Reduces the computational cost by reducing the number of parameters to learn

Max Pooling

Extracts the sharpest features of an image, making it more general



Input (4 x 4)

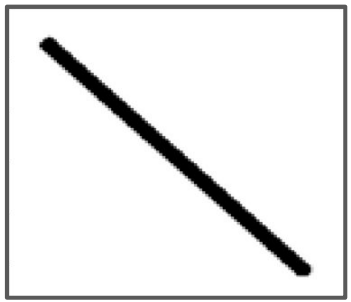
Output (2 x 2)

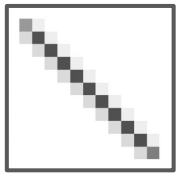
255	0	0	0
0	255	0	0
0	0	255	0
0	0	0	255

255	0
0	255

Average Pooling

Takes average feature of an image, minimize overfitting





Input (4 x 4)

Output (2 x 2)

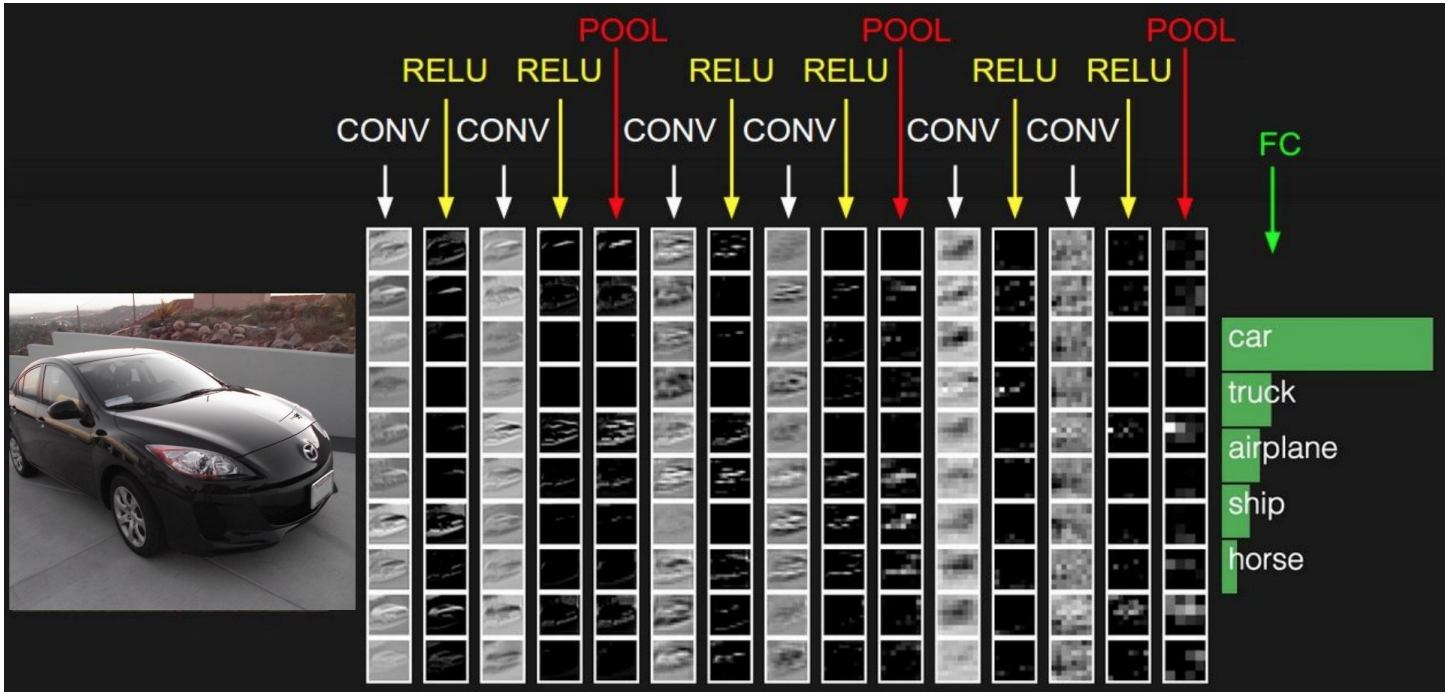
0	255	255	255
255	0	255	255
255	255	0	255
255	255	255	0

128	255
255	128

Pooling Layers

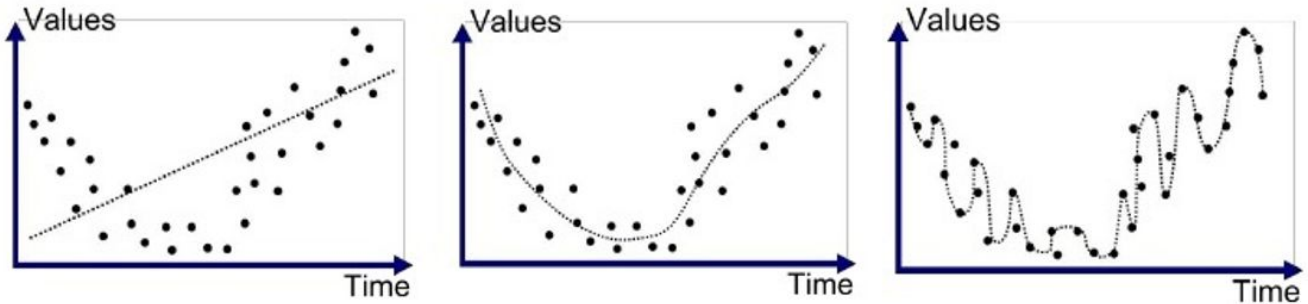
- Increasing stride may have a more detailed effect in contrast to pooling, however pooling layers have fewer and less operations and are therefore faster.

So what does our network look like?



Dropout

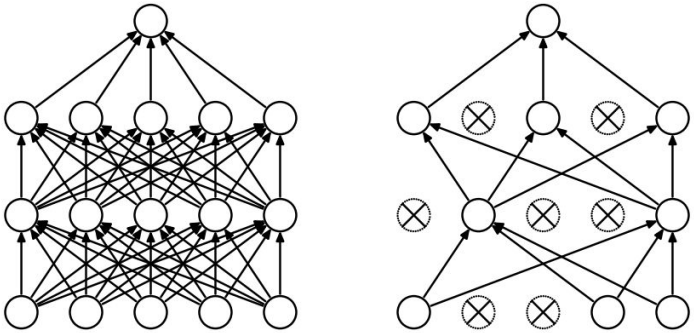
1. First, what is overfitting?
    a. Overfitting is when the neural network corresponds too closely to the dataset, and cannot be generalized. This tends to happen when a model is excessively complex relative to the data
    b. Conversely, underfitting is when the network cannot capture the underlying trend of the dataset which may happen if your network is not complicated enough.



Underfitted | Good Fit/Robust | Overfitted

Dropout - How can we solve overfitting?

1. Training phase
    a. Each weight has a probability *p* that they will be multiplied by zero (dropped). This probability is often set to 0.5, which is considered to be close to optimal for a wide range of networks and tasks
    b. This has the effect of removing random connections between activations effectively creating a new network/outlook on the data per each train set



(a) Standard Neural Net (b) After applying dropout.

Dropout - How can we solve overfitting?

2. Post Train

   a. After training weights will be abnormally high as they were adjusted assuming only (1-p) percent of the weights would be summed together and used.
   b. To fix this we normalize weights to lower the expectation of each weight. We do this by scaling each weight by 1/p
   c. “This makes sure that for each unit, the expected output from it under random dropout will be the same as the output during pretraining.” ~Dropout: A Simple Way to Prevent Neural Networks from Overfitting
      i. http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf

Coding Example

Live example using google colab.

Follow along in your own notebook or see final solution network at

https://colab.research.google.com/drive/1zHvu9JGgXAomrHlR2wUjsH7H1fctUuuv

References

http://cs231n.stanford.edu/slides/2017/cs231n_2017_lecture5.pdf