Convolutional Neural Networks

Dr. Sanjay Chatterji

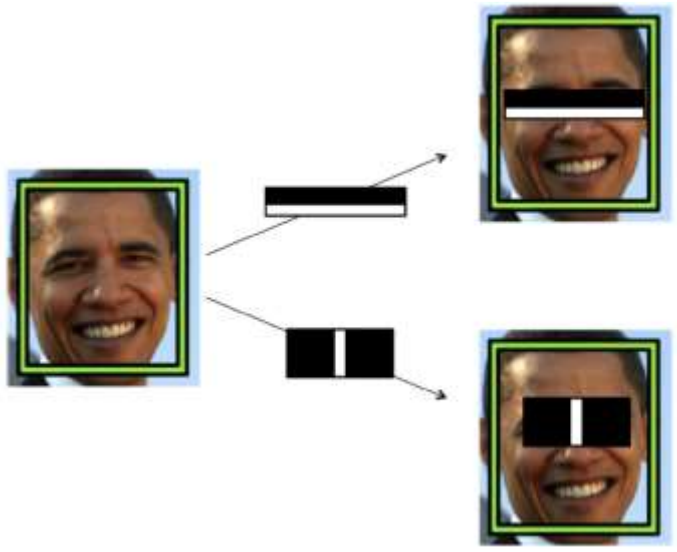
CS 837

Neurons in Human Vision

- The human sense of vision is unbelievably advanced.
  - We can name objects we are looking at
  - We can perceive their depth
  - We can perfectly distinguish their contours
  - We can separate the objects from their backgrounds
- Our brain transforms the information into meaningful primitives
  - lines, curves, and shapes
- Specialized neurons are responsible for
  - capturing light information in the human eye
  - preprocessing
  - transporting to the visual cortex of the brain
  - finally analyzing to completion

The Shortcomings of Feature selection

- A computer vision problem.
  - given a randomly selected image
  - tell if there is a human face in this picture
- We could train a traditional machine learning algorithm.
  - choose the features (perhaps hundreds or thousands)
  - signal-to-noise ratio is much too low
  - On a dataset of 130 images and 507 faces, it achieves a 91.4% detection rate
  - Fails when
    - inappropriate features
    - partially covered
    - low light intensity
    - cartoon character
    - crumpled flier

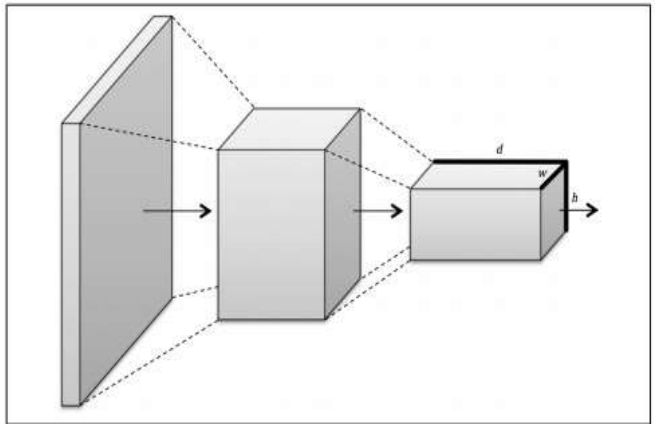


AlexNet

- ImageNet challenge: classify images into one of 200 possible classes given a training dataset of approximately 450,000 images.
- In 2011, the winner of the ImageNet benchmark had an error rate of 25.7%
- Alex Krizhevsky from Geoffrey Hinton's lab at the University of Toronto (2012)
  - Pioneering a deep learning architecture known as a *Convolutional Neural Network* for the first time
  - AlexNet
  - over the course of just a few months of work, completely crushed 50 years of traditional computer vision research
  - error rate of approximately 16%
  - put deep learning on the map for computer vision

Vanilla Deep Neural Networks

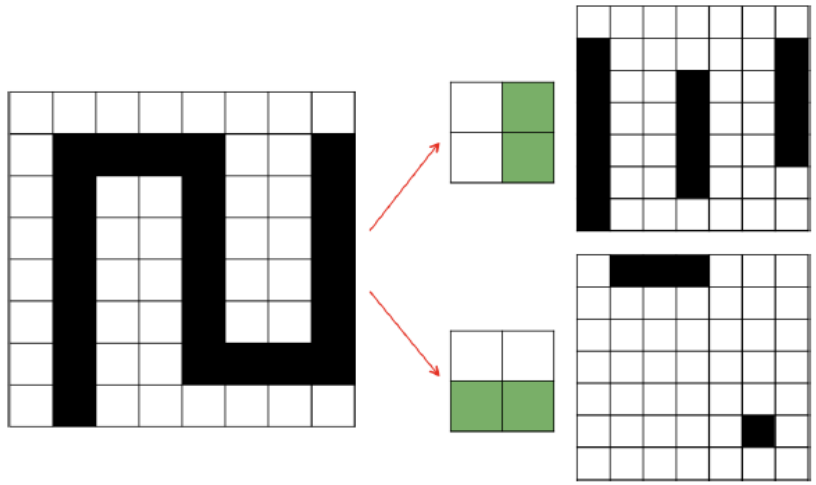
- The fundamental goal in applying deep learning to computer vision is to remove the cumbersome feature selection process.
- A naive approach might be for us to use a vanilla deep neural network.
- Challenge:
  - MNIST: black and white image with 28 x 28 pixels; fully connected Neural Network would have 784 incoming weights.
  - For a 200 x 200 pixel image input layer has 200 x 200 x 3 = 120,000 weights.
  - Wasteful and overfitting
- Layers of a convolutional network have neurons arranged in three dimensions
- A Convolutional layer
  - processes a three-dimensional volume of information
  - produces a new three-dimensional volume of information

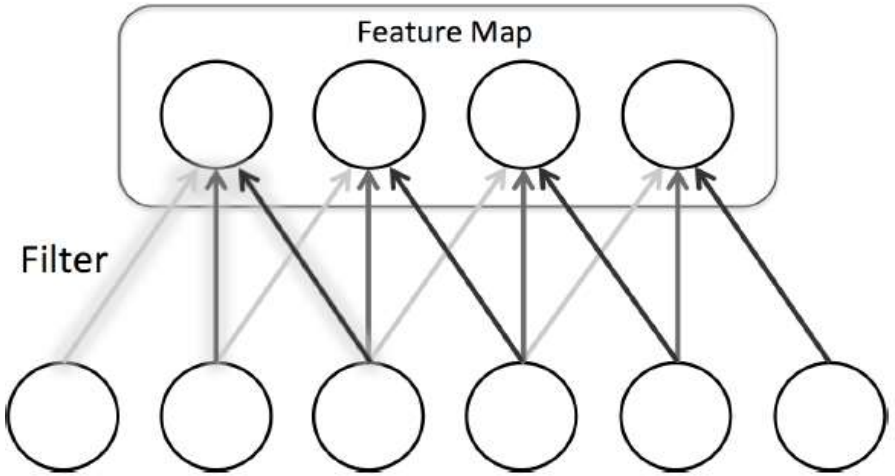


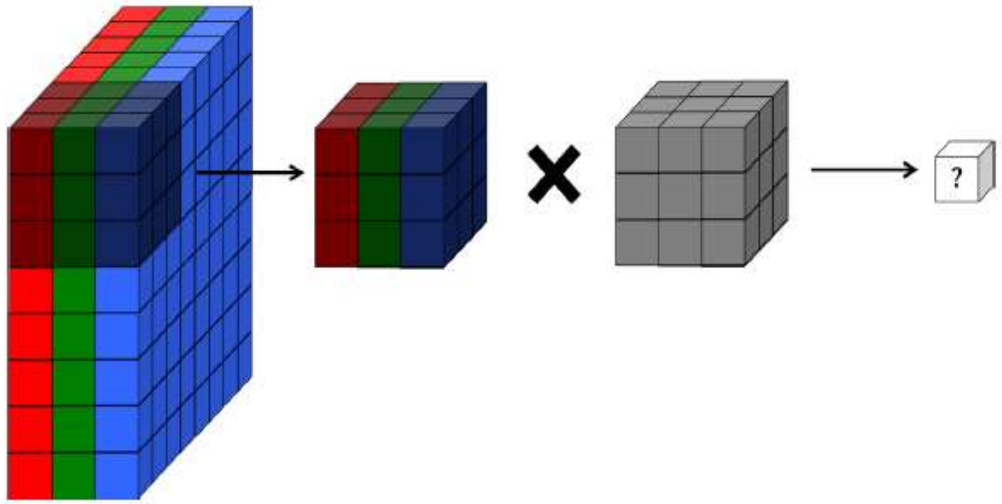
Filters and Feature Maps

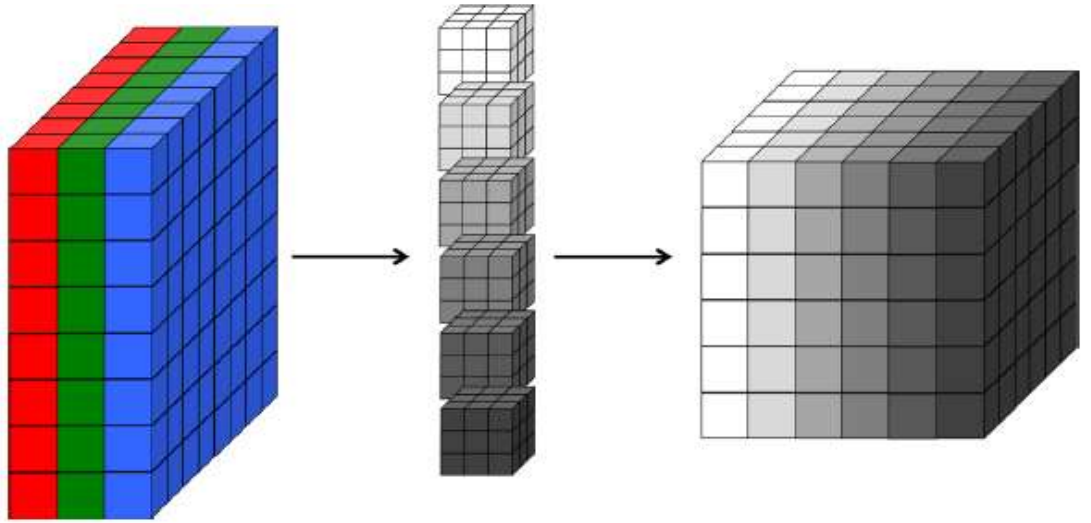
- Electrodes inserted into cat's brain and projected black-and-white patterns
  - Some neurons fired only when there were vertical lines
  - others when there were horizontal lines
  - and still others when the lines were at particular angles
- visual cortex
  - Organized in layers
  - Each layer works on previous layer features (lines, contours, shapes, entire object)
  - Feature detectors are replicated over the whole area
- A filter is essentially a feature detector
  - We multiply it over the entire area of an input image.
  - It represents combinations of connections replicated across the input.
- This result is our *feature map (like eye, nose, mouth, etc.)* $m_{ij}^k = f\left((W * x)_{ij} + b^k\right)$
  - k^{th} feature map in layer m is m^k *and corresponding filter is W.*
  - Neurons in the feature map have bias b^k.

filters and feature maps as neurons in a convolutional layer









Full Description of the Convolutional Layer

- Input volume characteristics
  - Its *width* w_{in}
  - Its *height* h_{in}
  - Its *depth* d_{in}
  - Its *zero padding* p
- Total of k filters with hyperparameters
  - *spatial extent* e (filter's height and width)
  - *stride* s (distance between consecutive applications of the filter)
  - bias b (added to each component of the convolution)
- Output volume characteristics
  - Function f applied on logit

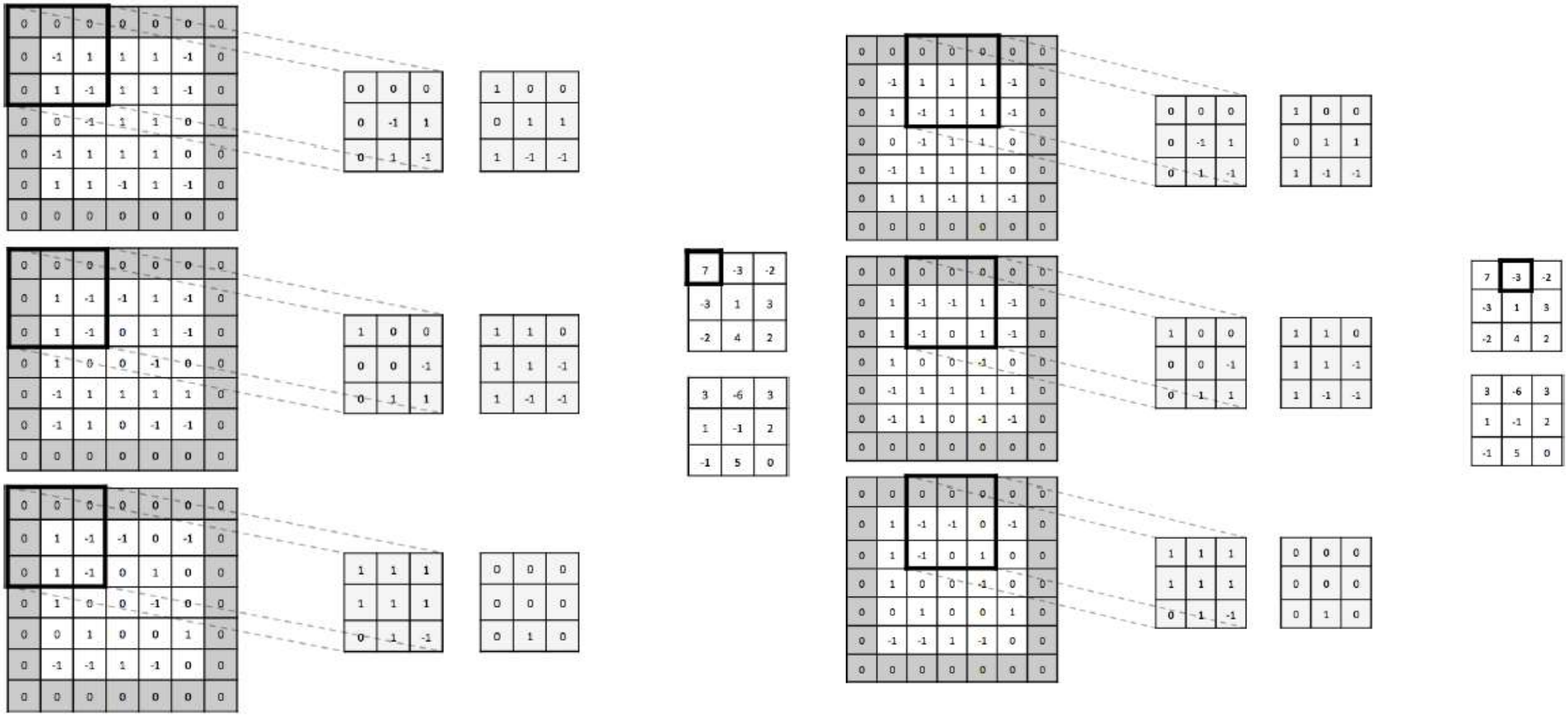
$$width\ w_{out} = \left\lceil \frac{w_{in} - e + 2p}{s} \right\rceil + 1$$

$$height\ h_{out} = \left\lceil \frac{h_{in} - e + 2p}{s} \right\rceil + 1$$

$$depth\ d_{out} = k$$

Hyperparameter values

- The m^{th} "depth slice" ($1 \leq m \leq k$) of the output volume corresponds to the function f applied to the sum of the m^{th} filter convoluted over the input volume and the bias b^m.
- It's wise to keep filter sizes small (size 3 x 3 or 5 x 5).
- Having more small filters is an easy way to achieve high representational power while also incurring a smaller number of parameters.
- It's also suggested to use a stride of 1 to capture all useful information in the feature maps, and a zero padding that keeps the output volume's height and width equivalent to the input volume's height and width.


0 0 0 0 0 0 0
0 -1 1 1 1 -1 0
0 1 -1 1 1 -1 0
0 0 -1 1 1 0 0
0 -1 1 1 1 0 0
0 1 1 -1 1 -1 0
0 0 0 0 0 0 0
0 0 0
0 -1 1
0 1 -1
1 0 0
0 1 1
1 -1 -1
0 0 0 0 0 0 0
0 1 -1 -1 1 -1 0
0 1 -1 0 1 -1 0
0 1 0 0 -1 0 0
0 -1 1 1 1 1 0
0 -1 1 0 -1 -1 0
0 0 0 0 0 0 0
1 0 0
0 0 -1
0 1 1
1 1 0
1 1 -1
1 -1 -1
0 0 0 0 0 0 0
0 1 -1 -1 0 -1 0
0 1 -1 0 1 0 0
0 1 0 0 -1 0 0
0 0 1 0 0 1 0
0 -1 -1 1 -1 0 0
0 0 0 0 0 0 0
1 1 1
1 1 1
0 1 -1
0 0 0
0 0 0
0 1 0
7 -3 -2
-3 1 3
-2 4 2
3 -6 3
1 -1 2
-1 5 0
0 0 0 0 0 0 0
0 -1 1 1 1 -1 0
0 1 -1 1 1 -1 0
0 0 -1 1 1 0 0
0 -1 1 1 1 0 0
0 1 1 -1 1 -1 0
0 0 0 0 0 0 0
0 0 0
0 -1 1
0 1 -1
1 0 0
0 1 1
1 -1 -1
0 0 0 0 0 0 0
0 1 -1 -1 1 -1 0
0 1 -1 0 1 -1 0
0 1 0 0 -1 0 0
0 -1 1 1 1 1 0
0 -1 1 0 -1 -1 0
0 0 0 0 0 0 0
1 0 0
0 0 -1
0 1 1
1 1 0
1 1 -1
1 -1 -1
0 0 0 0 0 0 0
0 1 -1 -1 0 -1 0
0 1 -1 0 1 0 0
0 1 0 0 -1 0 0
0 0 1 0 0 1 0
0 -1 -1 1 -1 0 0
0 0 0 0 0 0 0
1 1 1
1 1 1
0 1 -1
0 0 0
0 0 0
0 1 0
7 -3 -2
-3 1 3
-2 4 2
3 -6 3
1 -1 2
-1 5 0

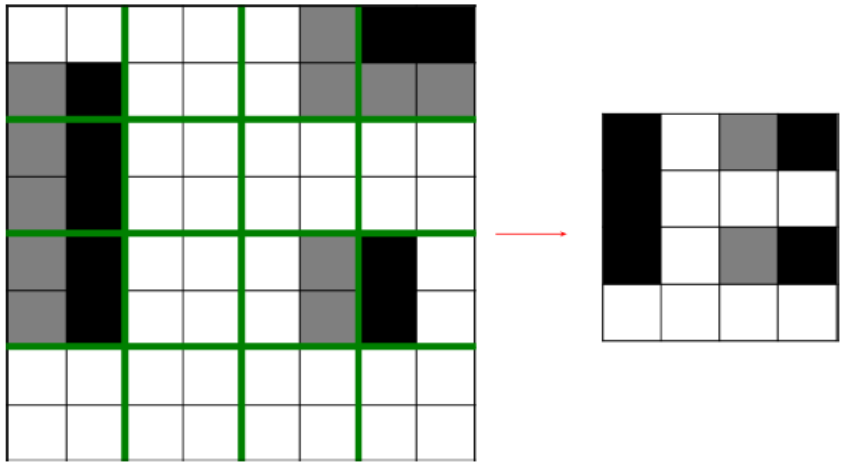
TensorFlow operation to perform a convolution on a minibatch of input volumes

tf.nn.conv2d(input, filter, strides, padding, use_cudnn_on_gpu=True, name=None)

- Input: a four-dimensional tensor of size N × hin × win × din, where N is the number of examples in our minibatch.
- filter: a four-dimensional tensor representing all of the filters applied in the convolution. It is of size e × e × din × k.
- The resulting tensor emitted by this operation has the same structure as input.
- Setting the padding argument to "SAME" also selects the zero padding so that height and width are preserved by the convolutional layer.

Max Pooling

- First break up each feature map into equally sized tiles.
- Then we create a condensed feature map.
  - create a cell for each tile
  - compute the maximum value in the tile
  - and propagate this maximum value into the corresponding cell of the condensed feature map.
- This reduces the dimensionality of feature maps and sharpen the located features.



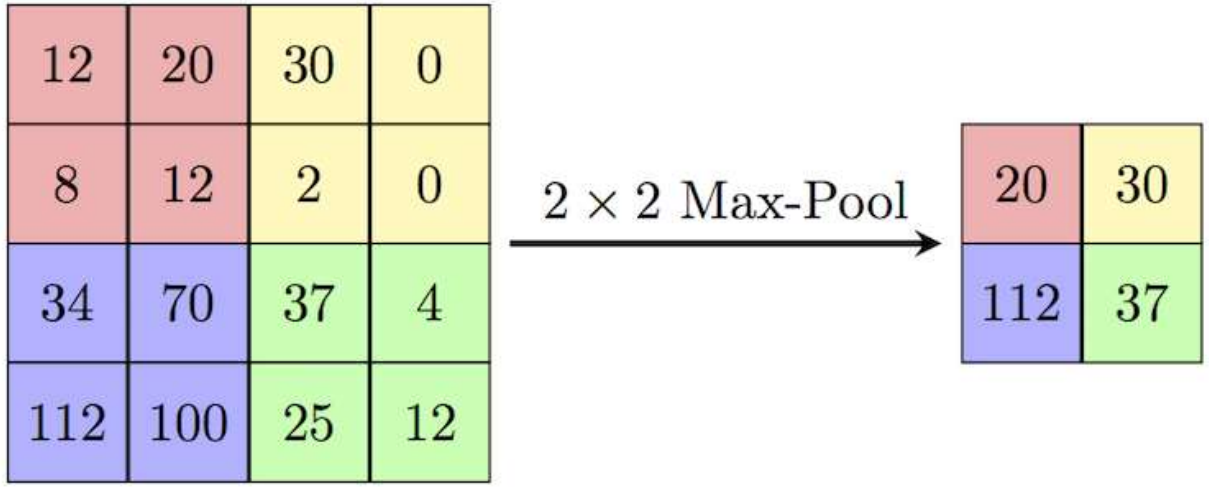
Description of Max Pooling Layer

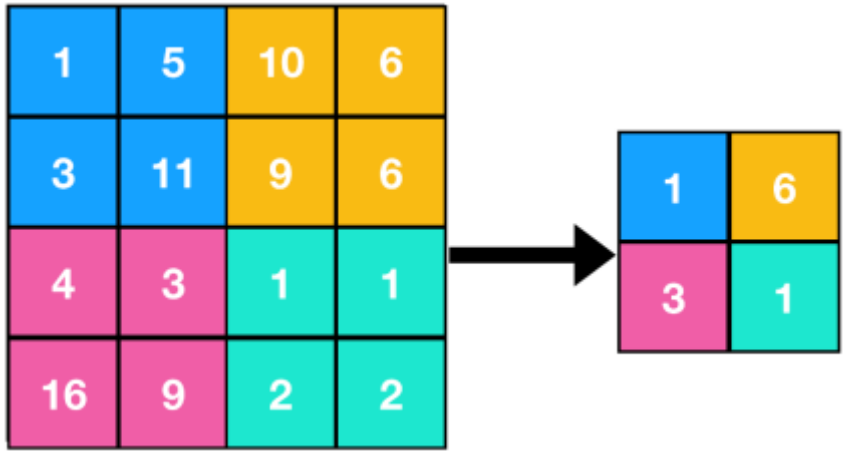
- We can describe a pooling layer with two parameters:
  - Its spatial extent *e*
  - Its stride *s*
- It's important to note that only two major variations of the pooling layer are used.
  - Nonoverlapping pooling layer with *e* = 2, *s* = 2
  - Overlapping pooling layer with *e* = 3, *s* = 2.
- Resulting dimensions of each feature map

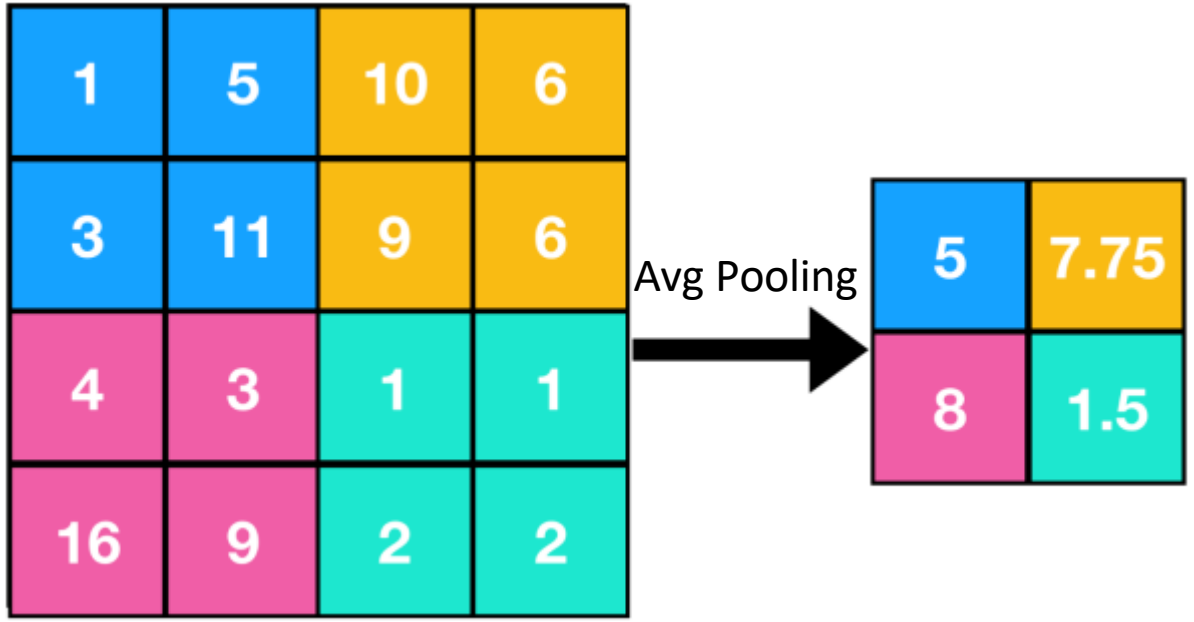
$$\text{width } w_{out} = \left\lceil \frac{w_{in} - e}{s} \right\rceil + 1$$

$$\text{height } h_{out} = \left\lceil \frac{h_{in} - e}{s} \right\rceil + 1$$

- It is *locally invariant*.
- Min Pooling, Average Pool, Adaptive Pool, Fractional Max Pooling


12 20 30 0
8 12 2 0
34 70 37 4
112 100 25 12
2 × 2 Max-Pool
20 30
112 37

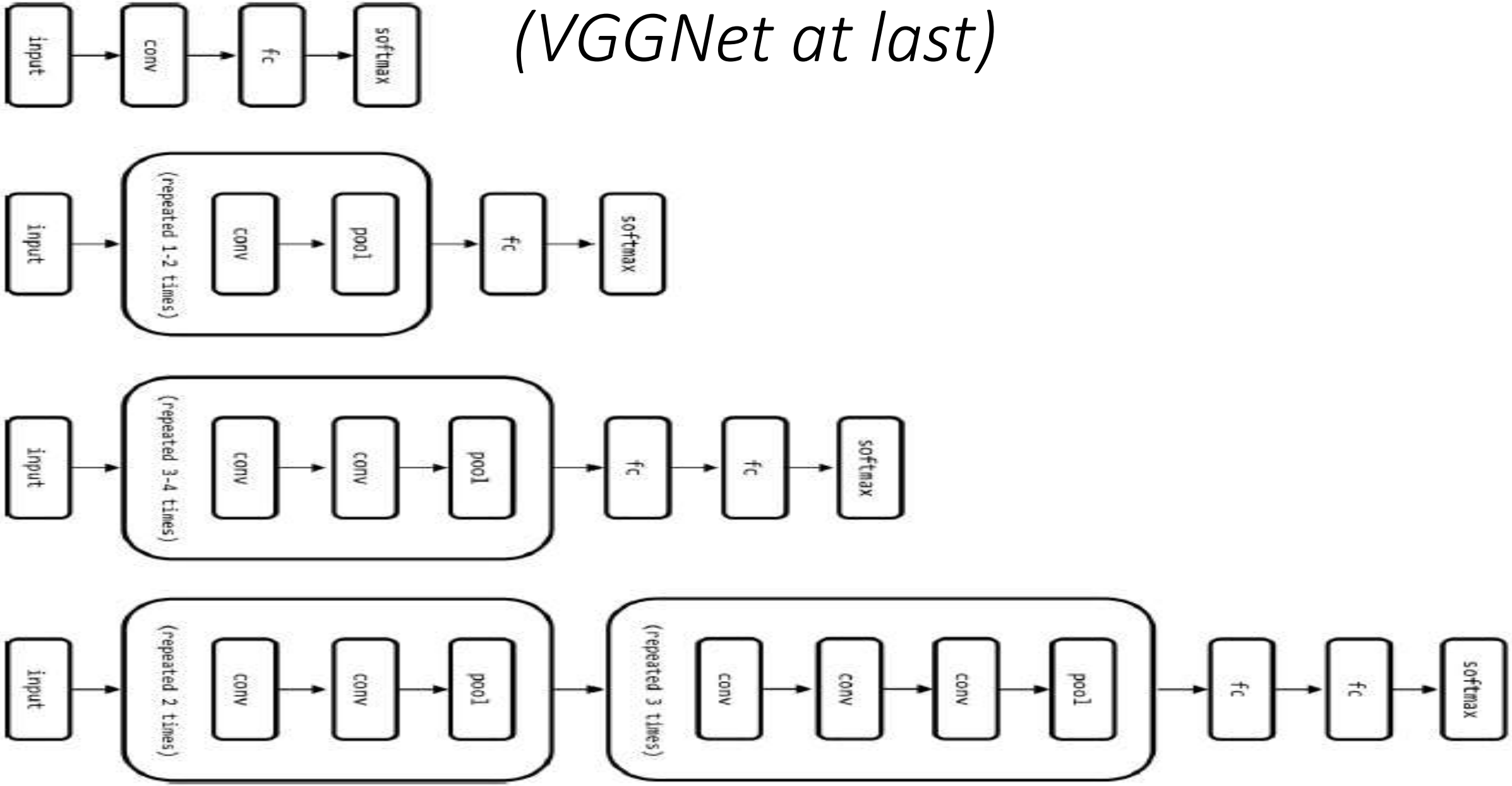

1 5 10 6
3 11 9 6
4 3 1 1
16 9 2 2
1 6
3 1


1 5 10 6
3 11 9 6
4 3 1 1
16 9 2 2
Avg Pooling
5 7.75
8 1.5

Architectural Description of Convolution Networks

- Now that we've described the building blocks of convolutional networks, we start putting them together.
- We reduce the number of pooling layers and instead stack multiple convolutional layers as
  - pooling operations are inherently destructive.
  - stacking several convolutional layers before each pooling layer allows us to achieve richer representations.
- Deep convolutional networks can take up a significant amount of space.
  - The VGGNet architecture takes approx. 90 MB/image on the forward pass and approx. 180 MB on the backward pass to update the parameters.
- Many deep networks make a compromise by using strides and spatial extents in the first convolutional layer that reduce the amount of information that needs to propagated up the network.

CNN architectures of various complexities (VGGNet at last)



Closing the Loop on MNIST with Convolutional Networks

- We'll revisit the MNIST challenge.
- Our feed-forward network was able to achieve a 98.2% accuracy.
- Our goal will be to push the envelope on this result.
- We'll build a pretty standard architecture: two pooling and two convolutional interleaved, followed by a fully connected layer (with dropout, $p = 0 . 5$) and a terminal softmax.
- Lets write a couple of helper methods in addition to our layer generator from the feed-forward network.
  - The first helper method generates a convolutional layer with a particular shape.
  - The 2nd one generates a max pooling layer with non-overlapping windows of size k.

```
def conv2d(input, weight_shape, bias_shape):
    in = weight_shape[0] * weight_shape[1] * weight_shape[2]
    weight_init = tf.random_normal_initializer(stddev=
                                               (2.0/in)**0.5)
    W = tf.get_variable("W", weight_shape,
                        initializer=weight_init)
    bias_init = tf.constant_initializer(value=0)
    b = tf.get_variable("b", bias_shape, initializer=bias_init)
    conv_out = tf.nn.conv2d(input, W, strides=[1, 1, 1, 1],
                            padding='SAME')
    return tf.nn.relu(tf.nn.bias_add(conv_out, b))

def max_pool(input, k=2):
    return tf.nn.max_pool(input, ksize=[1, k, k, 1],
                          strides=[1, k, k, 1], padding='SAME')
```

```
def layer(input, weight_shape, bias_shape):
    weight_stddev = (2.0/weight_shape[0])**0.5
    w_init = tf.random_normal_initializer(stddev=weight_stddev)
    bias_init = tf.constant_initializer(value=0)
    W = tf.get_variable("W", weight_shape,
                        initializer=w_init)
    b = tf.get_variable("b", bias_shape,
                        initializer=bias_init)
    return tf.nn.relu(tf.matmul(input, W) + b)

def inference(x, keep_prob):

    x = tf.reshape(x, shape=[-1, 28, 28, 1])
    with tf.variable_scope("conv_1"):
        conv_1 = conv2d(x, [5, 5, 1, 32], [32])
        pool_1 = max_pool(conv_1)

    with tf.variable_scope("conv_2"):
        conv_2 = conv2d(pool_1, [5, 5, 32, 64], [64])
        pool_2 = max_pool(conv_2)

    with tf.variable_scope("fc"):
        pool_2_flat = tf.reshape(pool_2, [-1, 7 * 7 * 64])
        fc_1 = layer(pool_2_flat, [7*7*64, 1024], [1024])

        # apply dropout
        fc_1_drop = tf.nn.dropout(fc_1, keep_prob)

    with tf.variable_scope("output"):
        output = layer(fc_1_drop, [1024, 10], [10])

    return output
```

Discussion of the Code

- We take the flattened versions of the input pixel values and reshape them into a tensor of the N × 28 × 28 × 1, where
  - N is the number of examples in a minibatch
  - 28 is the width and height of each image, and 1 is the depth
- Then build a conv. layer with 32 filters with spatial extent 5.
- This results in taking an input volume of depth 1 and emitting a output tensor of depth 32.
- This is then passed through a max pooling layer which compresses the information.
- Then build a similar second convolutional layer with 64 filters, taking an input tensor of depth 32 and emitting an output tensor of depth 64.
- This, again, is passed through a max pooling layer to compress information.

Discussion of the Code

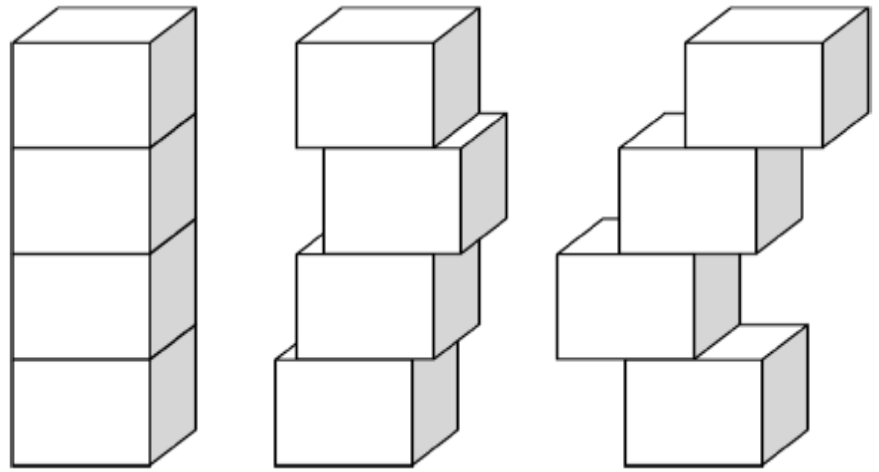
- Then pass the output of the max pooling layer into a fully connected layer.
- To do this, we flatten the tensor by computing the full size of each “subtensor” in the minibatch.
- We have 64 filters, which corresponds to the depth of 64.
- What is the height and width after passing through two max pooling layers.
- Using the formulas calculate each feature map has a height and width of 7.
- After the reshaping operation, we use a fully connected layer to compress the flattened representation into a hidden state of size 1,024.
- Finally, we send this hidden state into a softmax output layer with 10 bins.
- Accuracy = 99.4% (state of the art is 99.8%) which is very respectable.

Image Preprocessing Enable More Robust Models

- MNIST has already been preprocessed (Tame dataset).
  - The handwritten digits are perfectly cropped in just the same way.
  - There are no color aberrations because MNIST is black and white; and so on.
  - Natural images, however, are an entirely different beast.
- We also can expand our dataset artificially by randomly cropping the image, flipping the image, modifying saturation, modifying brightness, etc.
  - tf.image.per_image_whitening(image)
  - tf.random_crop(value, size, seed=None, name=None)
  - tf.image.random_flip_up_down(image, seed=None)
  - tf.image.random_flip_left_right(image, seed=None)
  - tf.image.transpose_image(image)
  - tf.image.random_brightness(image, max_delta, seed=None)
  - tf.image.random_contrast(image, lower, upper, seed=None)
  - tf.image.random_saturation(image, lower, upper, seed=None)
  - tf.image.random_hue(image, max_delta, seed=None)
- Applying these transformations helps us build networks that are robust to the different kinds of variations that are present in natural images, and make predictions with high fidelity in spite of potential distortions.

Accelerating Training with Batch Normalization

- Google (2015) devised *batch normalization* to accelerate the training of feed-forward and CNN.
  - If we randomly shift the blocks, the tower configuration become unstable.
  - Eventually the tower falls apart.
- In the process of training the weights of the network
  - The output distribution of the neurons in the bottom layer begins to shift.
  - The top layer has to learn how to make the appropriate predictions.
  - The top layer also modifies itself to accommodate the shifts.
  - This significantly slows down training.



Batch normalization

- We modify the architecture of our network to
  - Grab the vector of incoming logits
  - Normalize each component of the vector by subtracting the mean and dividing by the standard deviation
  - Use affine transform to restore representational power $\gamma\hat{\mathbf{x}} + \beta$
- Batch normalization can be expressed
  - for convolutional layer
  - for nonconvolutional feedforward layers
- Batch normalization
  - speeds up training by preventing significant shifts in the distribution of inputs
  - increases the learning rate
  - acts as a regularizer (L2) and removes the need for dropout
  - removes the need for photometric distortions

Batch normalization for a convolutional layer

```
def conv_batch_norm(x, n_out, phase_train):
    beta_init = tf.constant_initializer(value=0.0, dtype=tf.float32)
    gamma_init = tf.constant_initializer(value=1.0, dtype=tf.float32)
    beta = tf.get_variable("beta", [n_out], initializer=beta_init)
    gamma = tf.get_variable("gamma", [n_out], initializer=gamma_init)
    batch_mean, batch_var = tf.nn.moments(x, [0,1,2], name='moments')
    ema = tf.train.ExponentialMovingAverage(decay=0.9)
    ema_apply_op = ema.apply([batch_mean, batch_var])
    ema_mean, ema_var = ema.average(batch_mean),
    ema.average(batch_var)
    def mean_var_with_update():
        with tf.control_dependencies([ema_apply_op]):
        return tf.identity(batch_mean), tf.identity(batch_var)
    mean, var = control_flow_ops.cond(phase_train, mean_var_with_update, lambda:
    (ema_mean, ema_var))
    normed = tf.nn.batch_norm_with_global_normalization(x,mean, var, beta, gamma, 1e-3,
    True)
    return normed
```

batch normalization for nonconvolutional feedforward layers

```
def layer_batch_norm(x, n_out, phase_train):
    beta_init = tf.constant_initializer(value=0.0, dtype=tf.float32)
    gamma_init = tf.constant_initializer(value=1.0, dtype=tf.float32)
    beta = tf.get_variable("beta", [n_out], initializer=beta_init)
    gamma = tf.get_variable("gamma", [n_out], initializer=gamma_init)
    batch_mean, batch_var = tf.nn.moments(x, [0], name='moments')
    ema = tf.train.ExponentialMovingAverage(decay=0.9)
    ema_apply_op = ema.apply([batch_mean, batch_var])
    ema_mean, ema_var = ema.average(batch_mean), ema.average(batch_var)
    def mean_var_with_update():
        with tf.control_dependencies([ema_apply_op]):
        return tf.identity(batch_mean), tf.identity(batch_var)
    mean, var = control_flow_ops.cond(phase_train, mean_var_with_update, lambda: (ema_mean,
    ema_var))
    x_r = tf.reshape(x, [-1, 1, 1, n_out])
    normed = tf.nn.batch_norm_with_global_normalization(x_r, mean, var, beta, gamma, 1e-3, True)
    return tf.reshape(normed, [-1, n_out])
```

Integrate batch normalization into the convolutional and fully connected layer

```
def conv2d(input, weight_shape, bias_shape, phase_train,visualize=False):
    incoming = weight_shape[0] * weight_shape[1] * weight_shape[2]
    weight_init = tf.random_normal_initializer(stddev=(2.0/incoming)**0.5)
    W = tf.get_variable("W", weight_shape,initializer=weight_init)
    if visualize:
        filter_summary(W, weight_shape)
    bias_init = tf.constant_initializer(value=0)
    b = tf.get_variable("b", bias_shape, initializer=bias_init)
    logits = tf.nn.bias_add(tf.nn.conv2d(input, W, strides=[1, 1, 1, 1], padding='SAME'), b)
    return tf.nn.relu(conv_batch_norm(logits, weight_shape[3],
    phase_train))
def layer(input, weight_shape, bias_shape, phase_train):
    weight_init = tf.random_normal_initializer(stddev=(2.0/weight_shape[0])**0.5)
    bias_init = tf.constant_initializer(value=0)
    W = tf.get_variable("W", weight_shape,initializer=weight_init)
    b = tf.get_variable("b", bias_shape,initializer=bias_init)
    logits = tf.matmul(input, W) + b
    return tf.nn.relu(layer_batch_norm(logits, weight_shape[1], phase_train))
```

Advantages of Batch Normalization

- Speed up training by preventing significant shifts in the distribution of inputs to each layer.
- Allows us to significantly increase the learning rate.
- Acts as a regularizer
- Removes the need for dropout and (when used) L2 regularization
- Batch regularization largely removes the need for photometric distortions, and we can expose the network to more “real” images during the training process.

Building a Convolutional Network for CIFAR-10

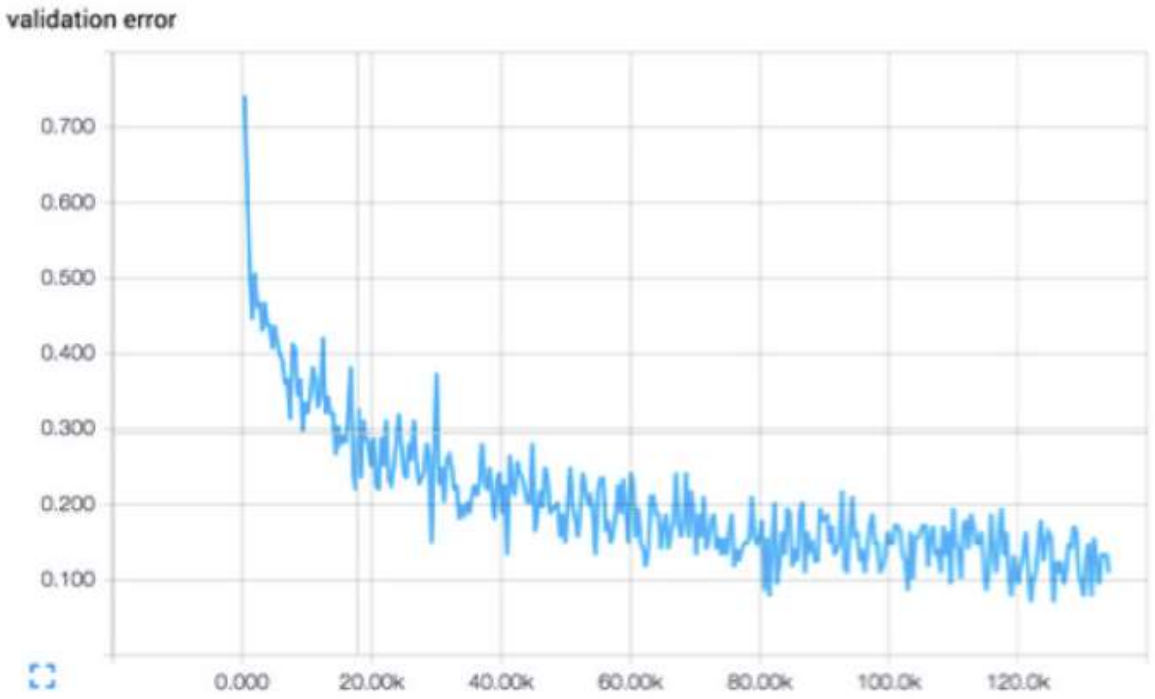
- The CIFAR-10 challenge
  - consists of 32 x 32 color images
  - belong to one of 10 possible classes
  - a surprisingly hard challenge
- We build networks
  - both with and without batch normalization
  - We increase the learning rate by 10-fold
- We distort random 24 × 24 crops of the input images for training.
- Batch normalization happens to logits before they're fed into a nonlinearity.
- We use two convolutional layers (each followed by a max pooling layer).
- There are then two fully connected layers followed by a softmax.
- Finally, we use the Adam optimizer to train our convolutional networks.
- Accuracy: 92.3% without batch normalization and 96.7% with batch normalization.

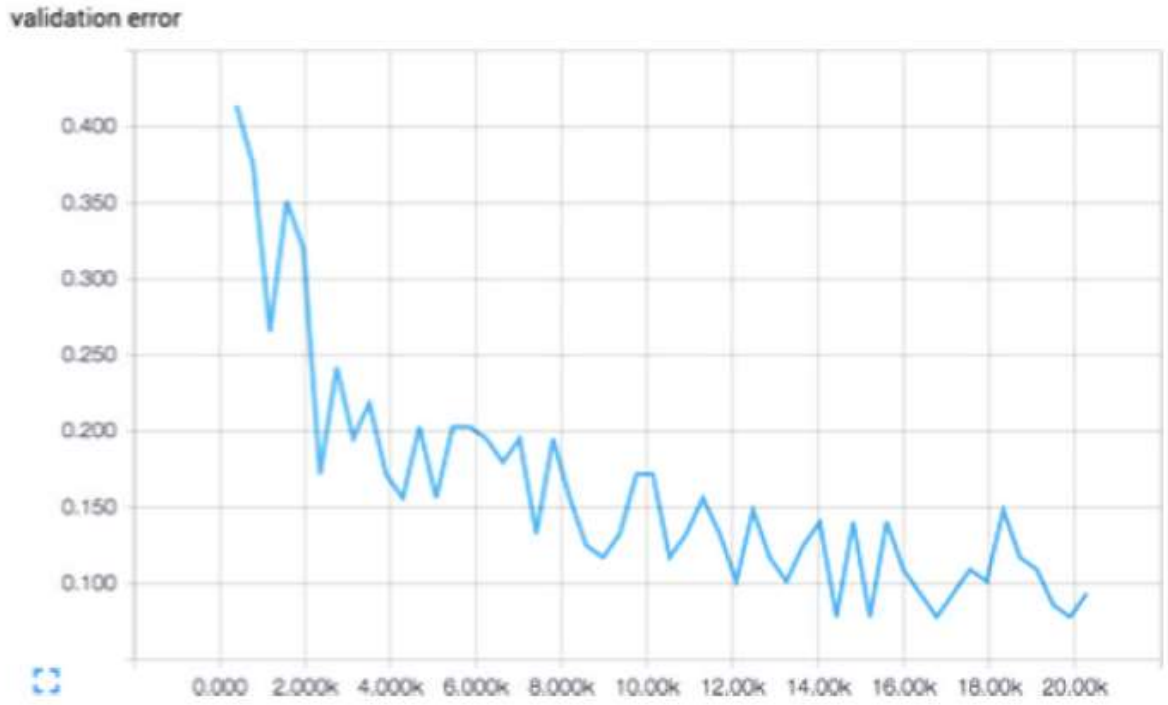
Inference

```
def inference(x, keep_prob, phase_train):
    with tf.variable_scope("conv_1"):
        conv_1 = conv2d(x, [5, 5, 3, 64], [64], phase_train, visualize=True)
        pool_1 = max_pool(conv_1)
    with tf.variable_scope("conv_2"):
        conv_2 = conv2d(pool_1, [5, 5, 64, 64], [64], phase_train)
        pool_2 = max_pool(conv_2)
    with tf.variable_scope("fc_1"):
        dim = 1
        for d in pool_2.get_shape()[1:].as_list():
                dim *= d
        pool_2_flat = tf.reshape(pool_2, [-1, dim])
        fc_1 = layer(pool_2_flat, [dim, 384], [384], phase_train)
        # apply dropout
        fc_1_drop = tf.nn.dropout(fc_1, keep_prob)
    with tf.variable_scope("fc_2"):
        fc_2 = layer(fc_1_drop, [384, 192], [192], phase_train)
        # apply dropout
        fc_2_drop = tf.nn.dropout(fc_2, keep_prob)
    with tf.variable_scope("output"):
        output = layer(fc_2_drop, [192, 10], [10], phase_train)
    return output
```

Visualizing Learning in Convolutional Networks

- We plot the validation errors over time as training progresses.
- We compare the rates of convergence between two networks.

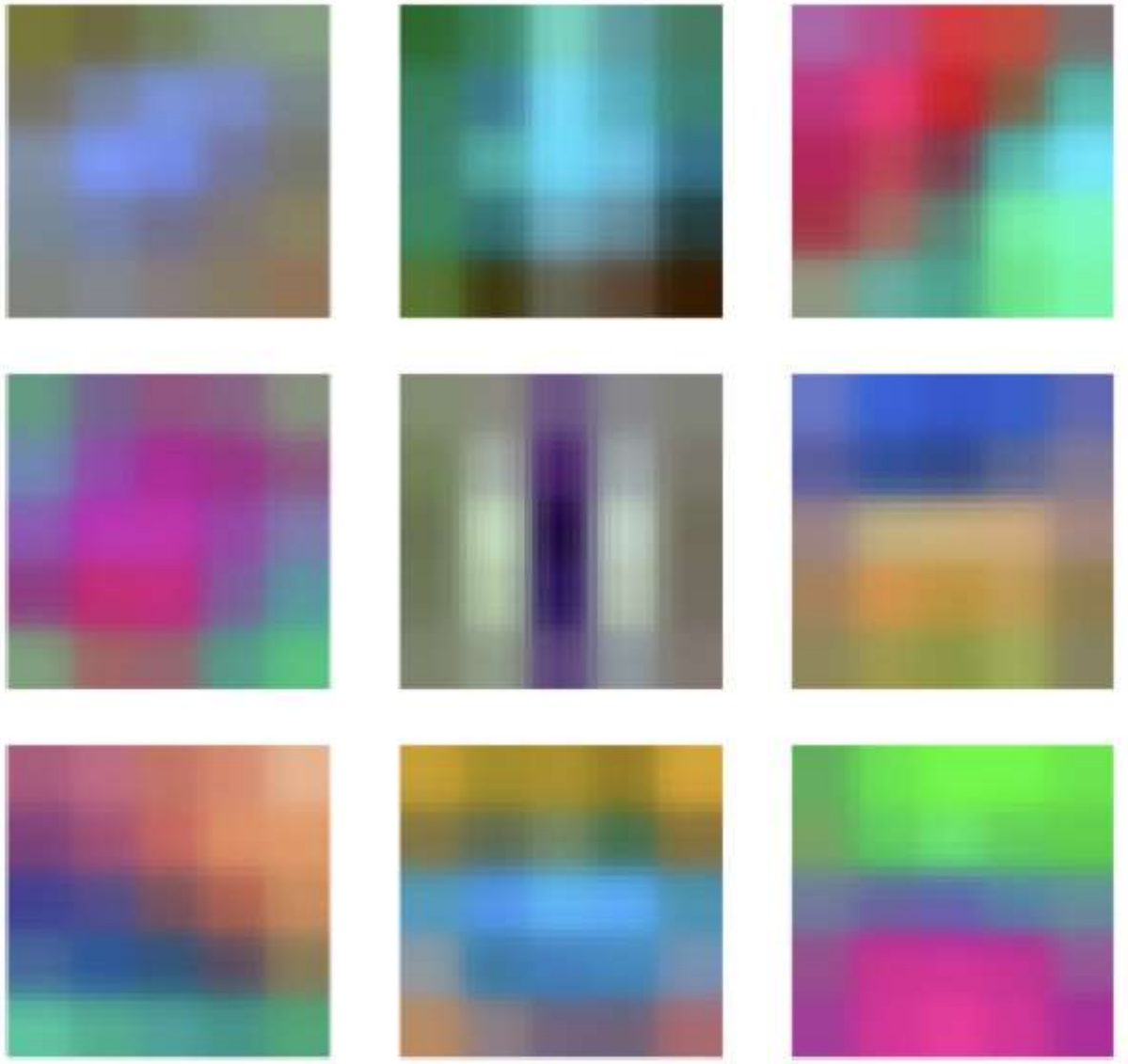




Training a CNN without batch normalization (left) versus with batch normalization (right).

Filters that the convolutional network learns

- Convolutional layers learn hierarchical representations of filters
  - first convolutional layer learns basic features (edges, simple curves, etc.).
  - Second convolutional layer will learn more complex features.
- Interesting features in our first level filters
  - vertical, horizontal, and diagonal edges
  - small dots or splotches of one color surrounded by another.
- We visualize how our network has learned to cluster various kinds of images.



Learning to cluster various kinds of images

- We take a large network.
- Train on the ImageNet.
- Grab the hidden state of FC layer just before the softmax.
- Take this high-dimensional representation for each image
- *t-Distributed Stochastic Neighbor Embedding*(*t-SNE)* compress it to two-dimensional representation.
- CNN has spectacular learning capabilities.

Leveraging Convolutional Filters to Replicate Artistic Styles

- The goal of *neural style* is to be able to take an arbitrary photograph and re-render it as if it were painted in the style of a famous artist.
  - Let's take a pre-trained convolutional network.
  - Three images that we're dealing with.
    - source of content ***p***
    - source of style ***a***
    - generated image ***x***
  - Our goal will be to derive an error function
- Suppose $E_{\text{content}}(\boldsymbol{p}, \boldsymbol{x}) = \sum_{ij}(P_{ij}^{(l)} - X_{ij}^{(l)})^2$ $\quad G^{(l)}_{ij} = \sum_{c=0}^{m_l} F^{(l)}_{ic} F^{(l)}_{jc}$ $\quad E_{style}(\mathbf{a}, \mathbf{x}) = \frac{1}{4k_l^2 m_l^2} \sum_{l=1}^{L} \sum_{ij} \frac{1}{L}\left(A_{ij}^{(l)} - G_{ij}^{(l)}\right)^2$
  - a layer in the network has k_l filters; Size (Height * Width) of each feature map is m_l
  - activations in the feature maps can be stored in matrix $\boldsymbol{F}^{(l)}$ of size $kl \times ml$.
  - activations of the original and generated images are described in matrix $\boldsymbol{P}^{(l)}$ and $\boldsymbol{X}^{(l)}$
  - *Gram matrix (kl × kl)* represents correlations between feature maps in a given layer.
  - The correlations represent the texture that is common among all features.



The result of mixing the Rain Princess with a photograph of the MIT Dome

Learning Convolutional Filters for Other Problem Domains

- Although our examples in this chapter focus on image recognition, there are several other problem domains in which convolutional networks are useful.
  - Video analysis
  - Audiograms analysis
  - Natural language processing
  - Teach algorithms to play board games
  - Analyzing biological molecules for drug discovery
- Use five-dimensional tensors (including time as a dimension) and apply three-dimensional convolutions to extend the convolutional paradigm to video.
- In an application, a convolutional network slides over an audiogram input to predict phonemes on the other side.

Thank You