Multi-layer Neural Network - Binary Classification in $\{0,1\}$

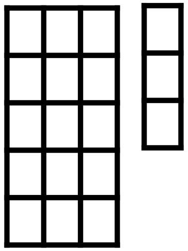
$a^{(\text{Layer})}_{\text{From, To}}$

L-th layer plays the role of features, but trained instead of pre-determined

This is a 5-dimensional vector

$$a^{(1)} = x$$

$$a^{(2)} = g(\Theta^{(1)} a^{(1)})$$



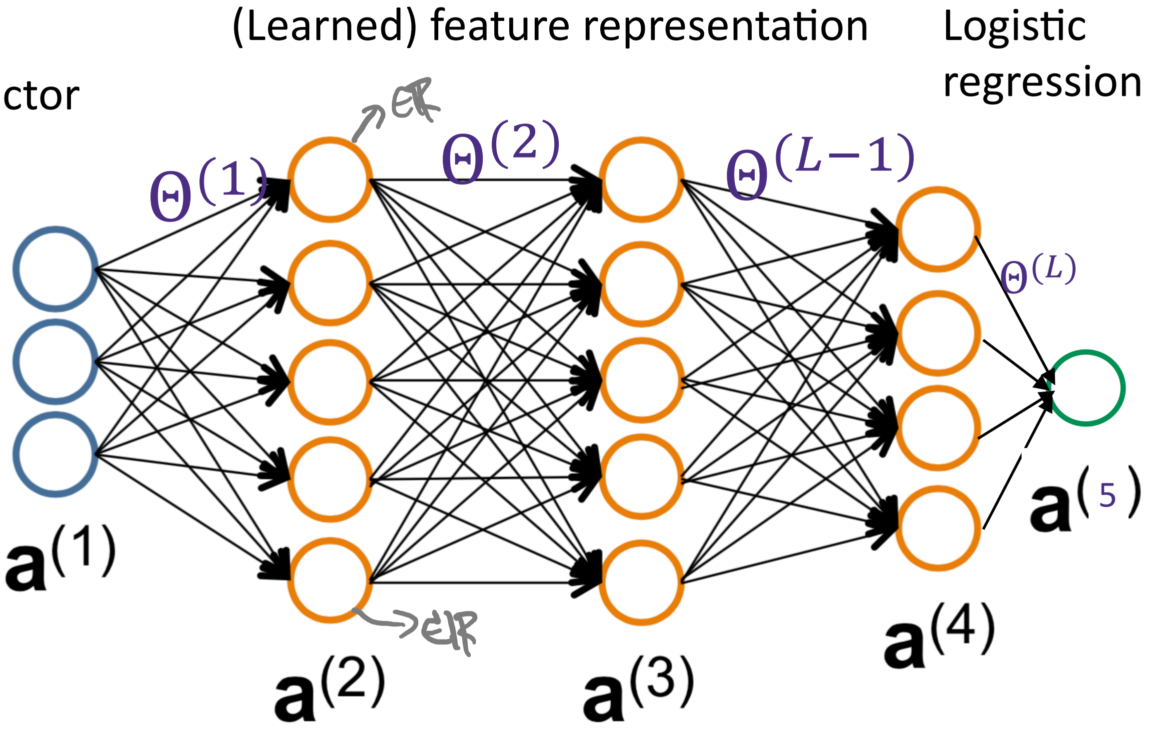
Scalar function g is applied coordinate-wise

$$\vdots$$

$$a^{(l+1)} = g(\Theta^{(l)} a^{(l)})$$

$$\vdots$$

$$\hat{y} = g(\Theta^{(L)} a^{(L)})$$



Cross entropy loss:

$$\mathcal{L}(y,\hat{y}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y})$$

$$g(z) = \frac{1}{1+e^{-z}}$$

Binary Logistic Regression with learned feature $a^{(4)}$

Multi-layer Neural Network - Binary Classification

$$a^{(1)} = x$$

$$a^{(2)} = \sigma(\Theta^{(1)} a^{(1)})$$

ReLU

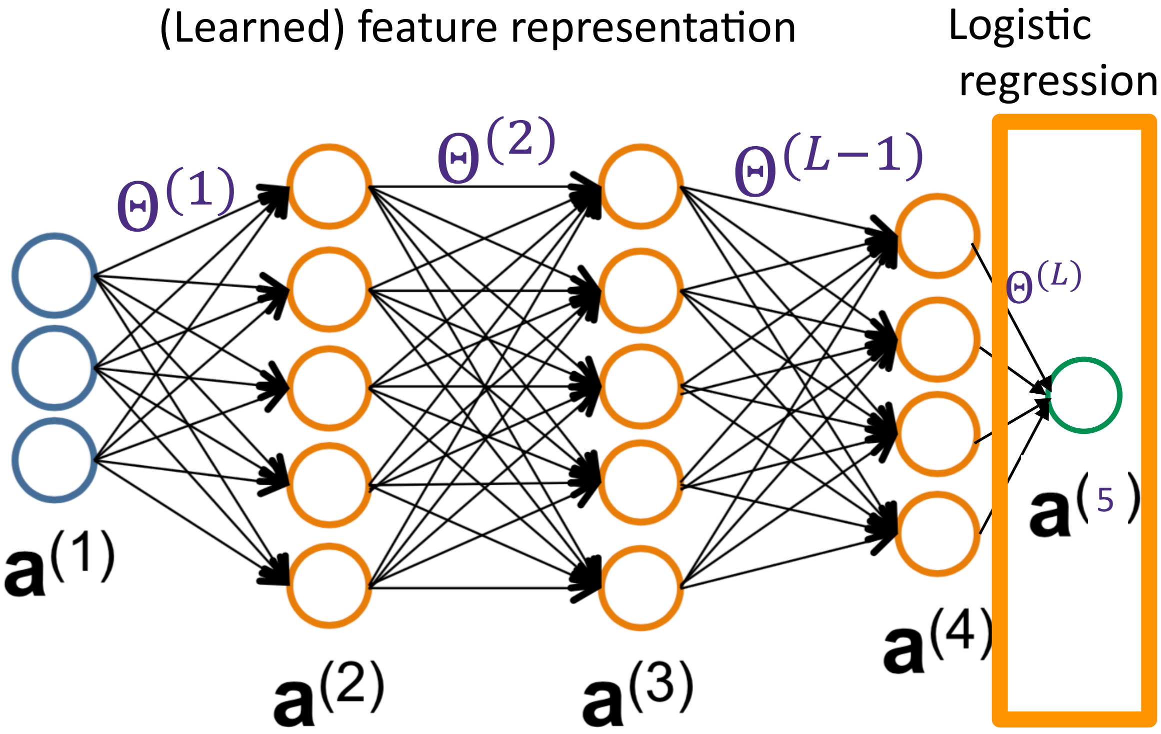
$$\vdots$$

$$a^{(l+1)} = \sigma(\Theta^{(l)} a^{(l)})$$

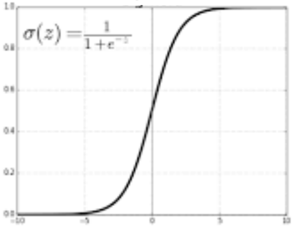
$$\vdots$$

$$\hat{y} = g(\Theta^{(L)} a^{(L)})$$

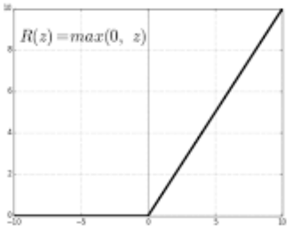
Sigmoid



Sigmoid



ReLU



- Why is ReLU better than sigmoid?

Cross entropy loss:

$$\mathcal{L}(y, \hat{y}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y})$$

$$\sigma(z) = \max\{0, z\} \qquad g(z) = \frac{1}{1+e^{-z}}$$

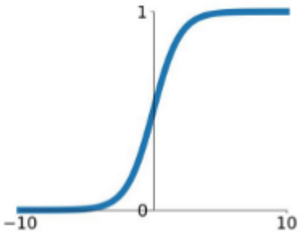
Binary Logistic Regression

Nonlinear activation function

- popular choices of activation function includes

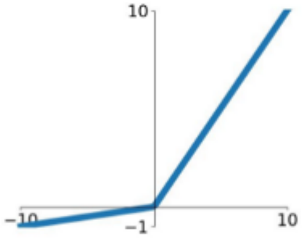
Sigmoid

$\sigma(x) = \frac{1}{1+e^{-x}}$



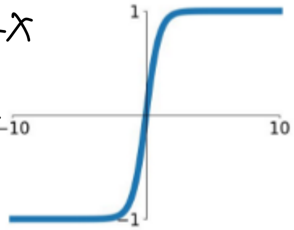
Leaky ReLU

$\max(0.1x, x)$



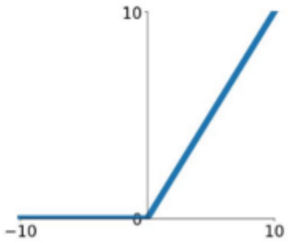
tanh

$\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$



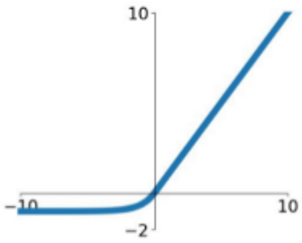
ReLU

$\max(0, x)$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^{x}-1) & x < 0 \end{cases}$$



- Why is ReLU better than Sigmoid? → Convex, simple subgradients, less "vanishing gradient"
- Why is ELU better than ReLU? → No nonzero grad, smooth

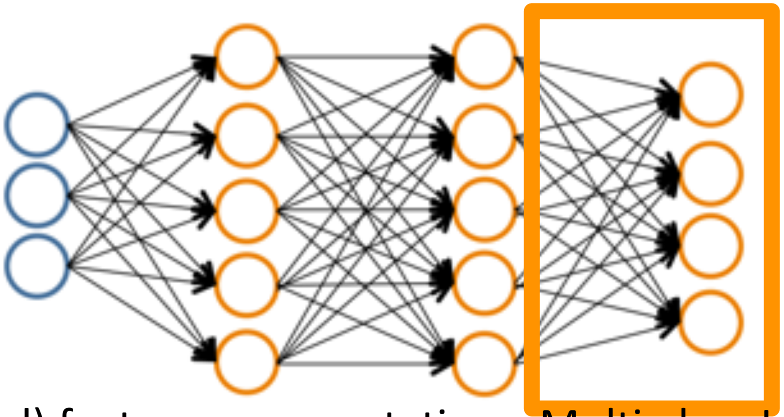
K-class Classification: multiple output units

Pedestrian

Car

Motorcycle

Truck



$$h_\Theta(\mathbf{x}) \in \mathbb{R}^K$$

Multi-class
Logistic
Regression

(Learned) feature representation   Multi-class Logistic regression

We want:

$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

when pedestrian

$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix}$$

when car

$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix}$$

when motorcycle

$$h_\Theta(\mathbf{x}) \approx \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

when truck

Multi-layer Neural Network - Regression

$$a^{(1)} = x$$

$$a^{(2)} = \sigma(\Theta^{(1)} a^{(1)})$$

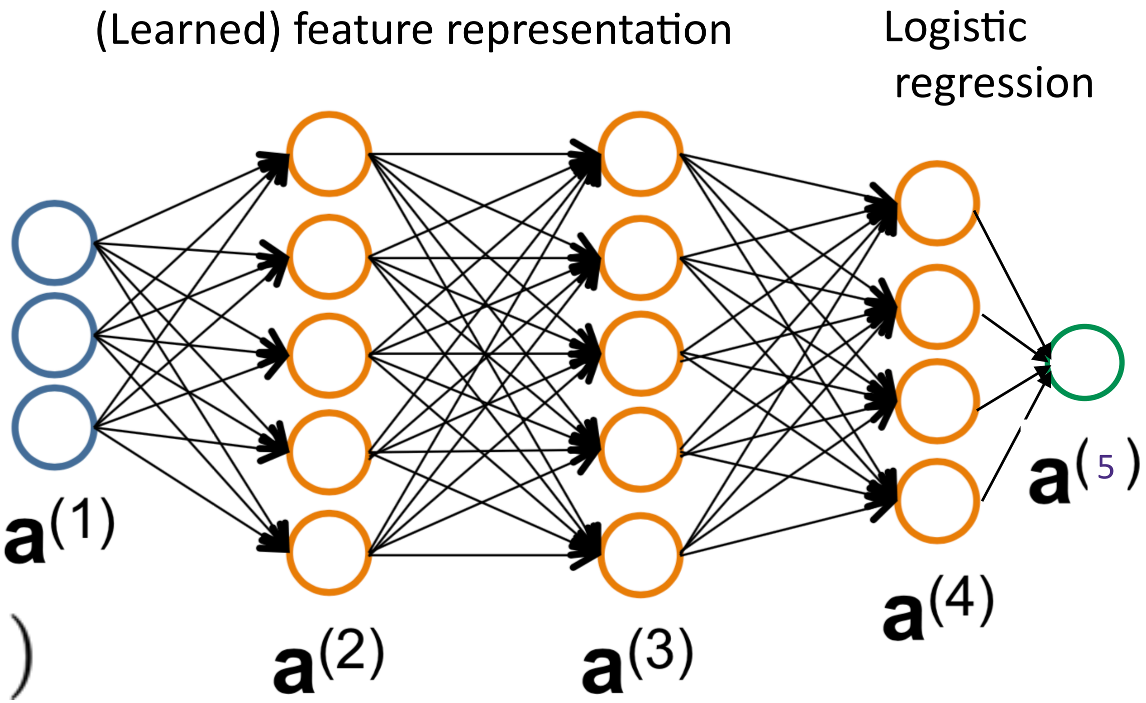
$$\vdots$$

$$a^{(l+1)} = \sigma(\Theta^{(l)} a^{(l)})$$

$$\vdots$$

$$\boxed{\widehat{y} = \Theta^{(L)} a^{(L)}}$$

Linear model



Square loss:

$$\mathcal{L}(y, \hat{y}) = (y - \hat{y})^2$$

$$\sigma(z) = \max\{0, z\}$$

Training Neural Networks

a: intermediate representation
z: linear mixture of last layer

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

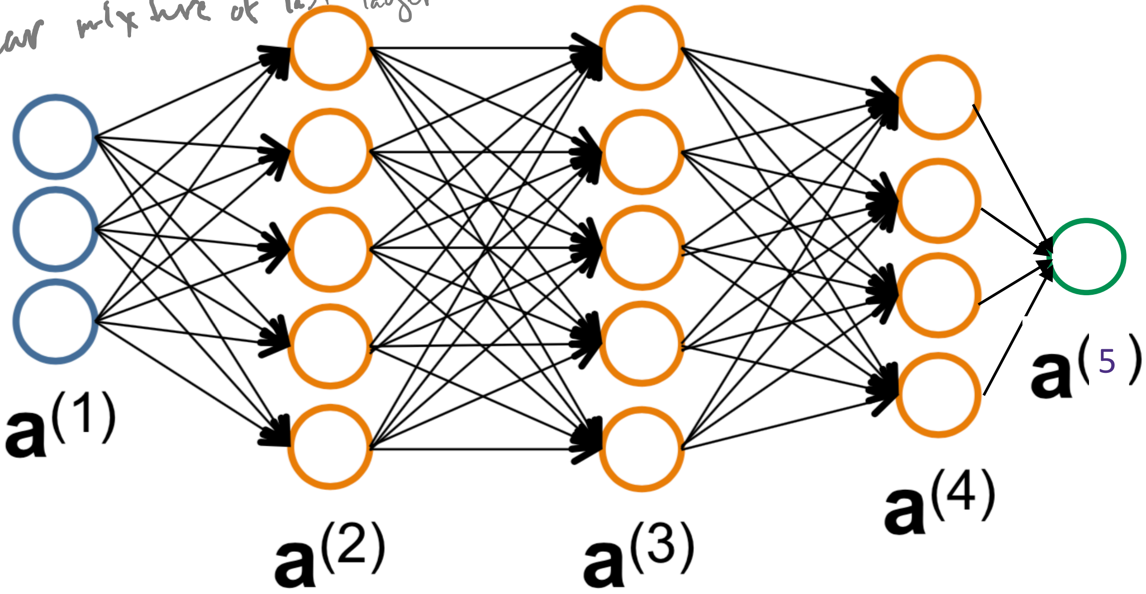
$$\vdots$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\hat{y} = g(\Theta^{(L)} a^{(L)})$$



$$\mathcal{L}(y, \hat{y}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y})$$

$$g(z) = \frac{1}{1+e^{-z}}$$

Gradient Descent:

$$\Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} \mathcal{L}(y, \hat{y}) \quad \forall l$$

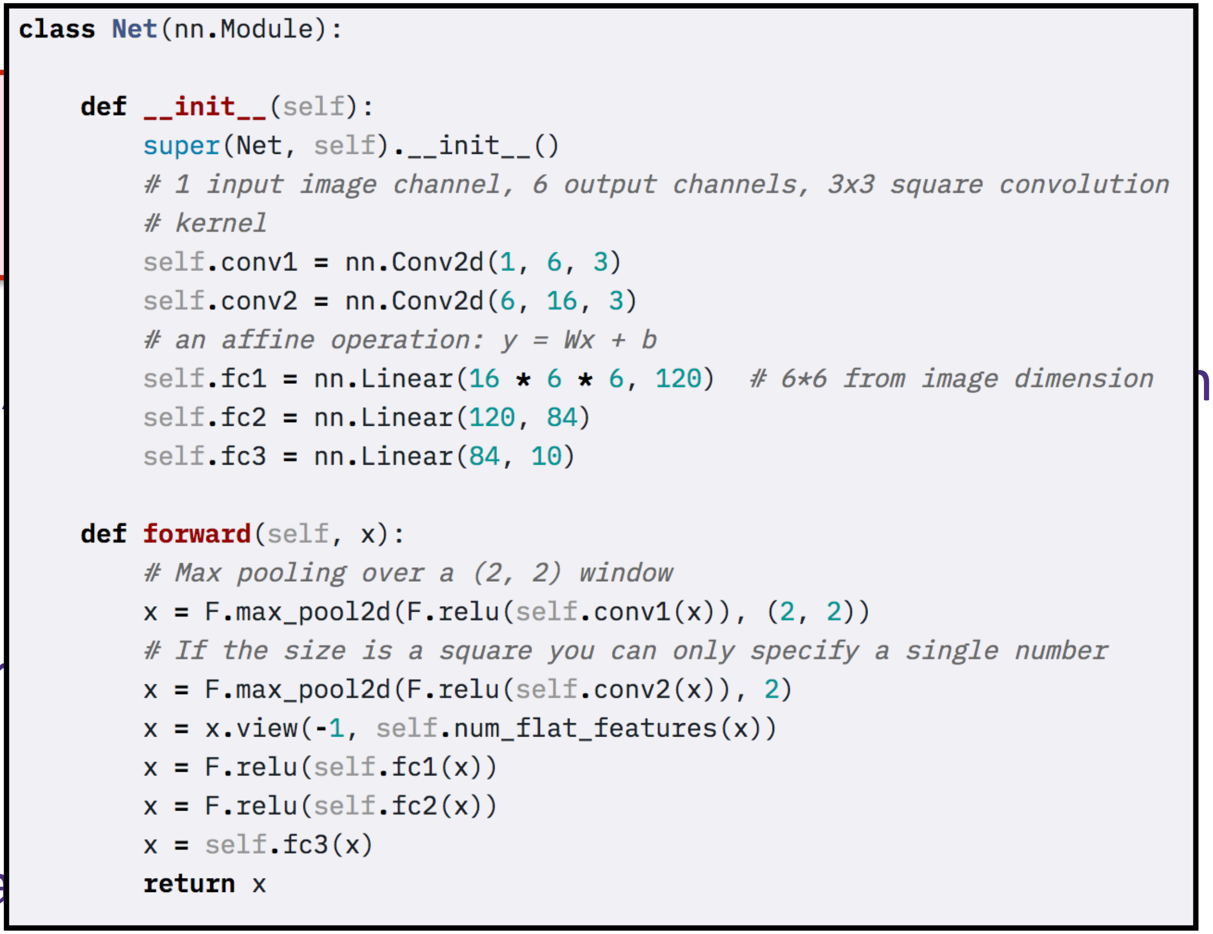
Gradient Descent:

$$\Theta^{(l)} \leftarrow \Theta^{(l)} - \eta \nabla_{\Theta^{(l)}} \mathcal{L}(y, \hat{y}) \qquad \forall l$$

Seems simple enough - what do packages like PyTorch, Tensorflow, Jax, Theano, Caffe, MxNet provide?

1. Automatic differentiation
   1. Given a NN, compute the gradient automatically
   2. Compute the gradient efficiently
2. Convenient libraries
   1. Set-up NN
   2. Choose algorithms (SGD,Adam,etc.) for training
   3. Hyper-parameter tuning
3. GPU support
   1. Linear algebraic operations

Gradient Descent:

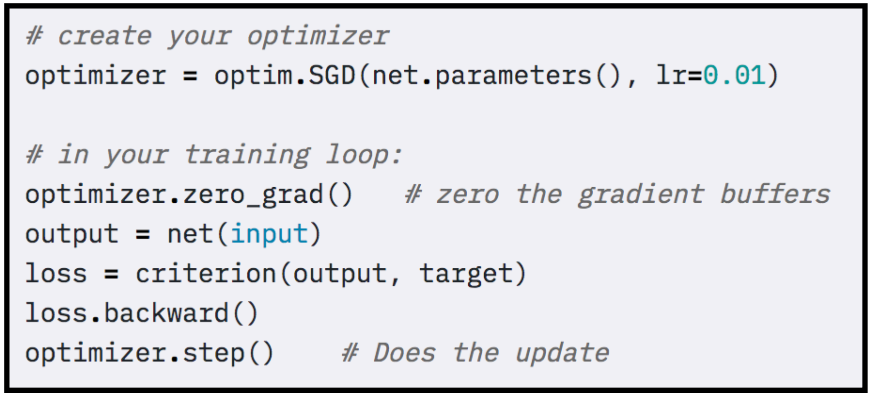


```
class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

Seems simple enough,
ne

1. Automatic differ

2. Convenient libra



```
# create your optimizer
optimizer = optim.SGD(net.parameters(), lr=0.01)

# in your training loop:
optimizer.zero_grad()   # zero the gradient buffers
output = net(input)
loss = criterion(output, target)
loss.backward()
optimizer.step()    # Does the update
```

Common training issues

Neural networks are **non-convex**

- For large networks, **gradients** can **blow up** or **go to zero**. This can be helped by **batchnorm** or **ResNet** architecture

- **Stepsize** and **batchsize** have large impact on optimizing the training error *and* generalization performance

- Fancier alternatives to SGD (Adagrad, Adam, LAMB, etc.) can significantly improve training

-Overfitting is common and not undesirable: typical to achieve 100% training accuracy even if test accuracy is just 80%

- Making the network *bigger* may make training *faster!*

- Start from a code that someone else has tried and tested

Common training issues

Training is too slow:

- Use larger step sizes, develop step size reduction schedule
- Use GPU resources
- Change batch size
- Use momentum and more advanced optimizers (e.g., Adam)
- Apply batch normalization
- Make network larger or smaller (# layers, # filters per layer, etc.)

Test accuracy is low

- Try modifying all of the above, plus changing other hyperparameters

Back Propagation

Gradient descent

What do we need to run gradient descent?

$\nabla_\Theta \mathcal{L}(y, \hat{y})$

Need to know $\hat{y}$ → "Run NN"

How do we write the gradient for each layer's parameters?

$$a^{(1)} = x$$

$$\boxed{z^{(2)} = \Theta^{(1)} a^{(1}}$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

$$\vdots$$

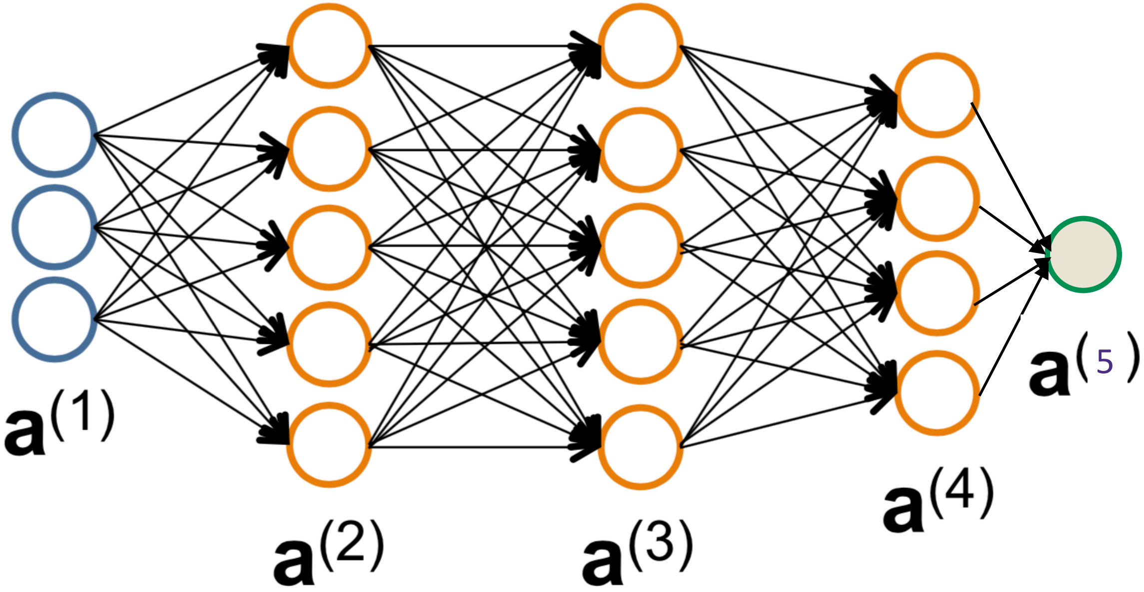
$$a^{(l)} = g\left(z^{(l)}\right)$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\hat{y} = a^{(L+1)}$$



$$\mathcal{L}(y, \hat{y}) = y\log(\hat{y}) + (1 - y)\log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Forward Propagation

- We are not writing the intercept at each layer for simplicity
- To compute gradients, we first run forward pass to get the intermediate representations $\{a^{(2)}, \dots, a^{(L)}\}$

$$a^{(1)} = x$$

$$\boxed{z^{(2)} = \Theta^{(1)} a^{(1)}}$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

$$\vdots$$

$$a^{(l)} = g(z^{(l)})$$

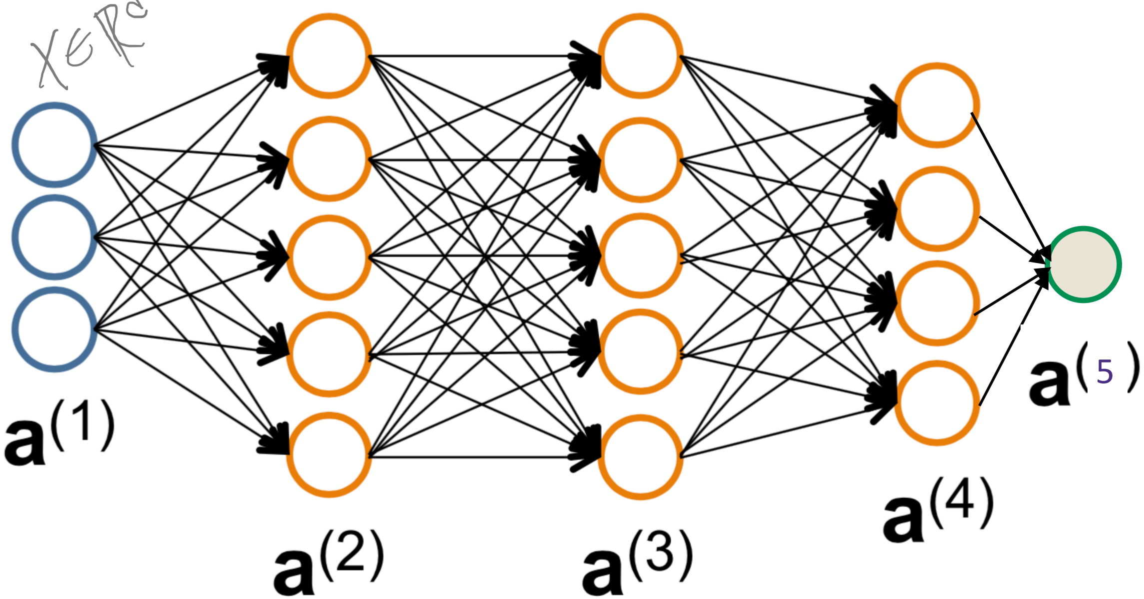
$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\hat{y} = a^{(L+1)}$$

$x \in \mathbb{R}^d$



$$\mathcal{L}(y, \hat{y}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y})$$

$$g(z) = \frac{1}{1+e^{-z}}$$

Backprop

- Parameters: $\Theta^{(1)} \in \mathbb{R}^{m \times d}, \Theta^{(2)}, \cdots \Theta^{(L-1)} \in \mathbb{R}^{m \times m}$
- Naive implementation takes $O(L^2)$ time, as each layer requires a full forward pass (with $O(L)$ operations) and some backward pass
- Backprop requires only $O(L)$ operations

$\rightarrow$ each $\nabla \Theta_\ell$ Takes $L-\ell$ intermediate gradient calculations

$$a^{(1)} = x \in \mathbb{R}^d$$

$$z^{(2)} = \Theta^{(1)} a^{(1)} \in \mathbb{R}^m$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

$$\vdots$$

$$a^{(1)} = x \in \mathbb{R}^d$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\hat{y} = a^{(L+1)}$$

Train by Stochastic Gradient Descent:

$$\Theta_{i,j}^{(l)} \leftarrow \Theta_{i,j}^{(l)} - \eta \frac{\partial \mathcal{L}(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}}$$

$$\mathcal{L}(y, \hat{y}) = y \log(\hat{y}) + (1 - y) \log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Backprop

Recursively computed in one backward pass

Computed in the forward pass

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

$$\vdots$$

$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\hat{y} = a^{(L+1)}$$

- Chain rule with $z_i^{(\ell+1)} = \Theta_{i,j}^{(\ell)} a_j^{(\ell)}$

$$\frac{\partial \mathcal{L}(y,\hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}} \cdot \boxed{\frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

Train by Stochastic Gradient Descent:

$$\Theta_{i,j}^{(l)} \leftarrow \Theta_{i,j}^{(l)} - \eta \frac{\partial \mathcal{L}(y,\hat{y})}{\partial \Theta_{i,j}^{(l)}}$$

$$\mathcal{L}(y,\hat{y}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y})$$

$$g(z) = \frac{1}{1+e^{-z}}$$

$$\boxed{\delta_i^{(l+1)} \triangleq \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}}}$$

Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

$$\vdots$$

$$a^{(l)} = g\left(z^{(l)}\right)$$

$$\boxed{z^{(l+1)} = \Theta^{(l)} a^{(l)}}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\hat{y} = a^{(L+1)}$$

$$\boxed{\frac{\partial \mathcal{L}(y,\hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}}$$

$$\delta_i^{(l)} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l)}} = \sum_k \underbrace{\frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_k^{(l+1)}}}_{\delta_k^{(\ell+1)}} \cdot \underbrace{\frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}}_{\Theta_{k,i}^{(l)} g'(z_i^{(l)})}$$

$$z_k^{(\ell+1)} = \sum_{i=1}^{m} \Theta_{k,i}^{(l)} g(z_i^{(l)})$$

$$\boxed{\begin{aligned} &\mathcal{L}(y,\hat{y}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y}) \\ &g(z) = \frac{1}{1+e^{-z}} \qquad \delta_i^{(l+1)} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}} \end{aligned}}$$

Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

$$\vdots$$

$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\hat{y} = a^{(L+1)}$$

$$\frac{\partial \mathcal{L}(y,\hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$\delta_i^{(l)} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l)}} = \sum_k \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_k^{(l+1)}} \cdot \frac{\partial z_k^{(l+1)}}{\partial z_i^{(l)}}$$

$$= \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}^{(l)} g'(z_i^{(l)})$$

Computed in the forward pass

$$= a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}^{(l)}$$

$$\mathcal{L}(y,\hat{y}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y})$$

$$g(z) = \frac{1}{1+e^{-z}}$$

$$\delta_i^{(l+1)} \triangleq \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}}$$

$$\boxed{g'(z) = g(z)(1 - g(z))}$$

Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

$$\vdots$$

$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\hat{y} = a^{(L+1)}$$

$$\frac{\partial \mathcal{L}(y,\hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$\delta_i^{(l)} = a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}^{(l)}$$

- We can recursively compute all $\delta^{(\ell)}$'s in a single backward pass
- And compute all gradients via

$$\frac{\partial \mathcal{L}(y,\hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$\mathcal{L}(y,\hat{y}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y})$$

$$g(z) = \frac{1}{1+e^{-z}} \qquad \delta_i^{(l+1)} \triangleq \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}}$$

Backprop

a's : intermediate "features", outputs of each layer
z's : linear input to the a's (before activ)

$$a^{(L)} = g(\underline{\Theta^{L-1} \cdot a^{L-1}})$$

chain rule

$$\frac{\partial a^{(L)}}{\partial \omega} = \frac{\partial g'}{\partial \omega}(\_)\frac{\partial \_}{\partial \omega}$$

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

$$\vdots$$

$$a^{(l)} = g\left(z^{(l)}\right)$$

$$\underline{z^{(l+1)}} = \underline{\Theta^{(l)} a^{(l)}}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\boxed{\begin{aligned} a^{(L+1)} &= g(z^{(L+1)}) \\ \hat{y} &= a^{(L+1)} \end{aligned}}$$

$$\boxed{\frac{\partial \mathcal{L}(y,\hat{y})}{\partial \Theta_{i,j}^{(l)}} \overset{\text{chain}}{\underset{\text{rule}}{=}} \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}}$$

$$\boxed{\delta_i^{(l)} = a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}^{(l)}}$$

$$\delta_i^{(L+1)} = \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(L+1)}} = \frac{\partial}{\partial z_i^{(L+1)}}\left[y\log(g(z^{(L+1)})) + (1-y)\log(1 - g(z^{(L+1)}\right.$$

$$= \frac{y}{g(z^{(L+1)})} g'(z^{(L+1)}) - \frac{1-y}{1 - g(z^{(L+1)})} g'(z^{(L+1)})$$

$$= y - g(z^{(L+1)}) = y - a^{(L+1)}$$

$$\boxed{\mathcal{L}(y,\hat{y}) = y\log(\hat{y}) + (1-y)\log(1-\hat{y})}$$

$$g(z) = \frac{1}{1+e^{-z}}$$

$$\boxed{g'(z) = g(z)(1 - g(z))}$$

$$\delta_i^{(l+1)} \triangleq \frac{\partial \mathcal{L}(y,\hat{y})}{\partial z_i^{(l+1)}}$$

Backprop

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

$$\vdots$$

$$a^{(l)} = g(z^{(l)})$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\hat{y} = a^{(L+1)}$$

Recursive Algorithm!

$$\underline{\delta^{(L+1)} = y - a^{(L+1)}}$$

$$\delta_i^{(l)} = a_i^{(l)}(1 - a_i^{(l)}) \sum_k \delta_k^{(l+1)} \cdot \Theta_{k,i}^{(l)}$$

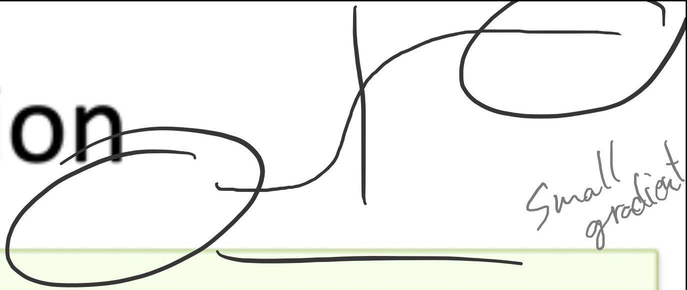
$$\frac{\partial \mathcal{L}(y, \hat{y})}{\partial \Theta_{i,j}^{(l)}} = \frac{\partial \mathcal{L}(y, \hat{y})}{\partial z_i^{(l+1)}} \cdot \frac{\partial z_i^{(l+1)}}{\partial \Theta_{i,j}^{(l)}} =: \delta_i^{(l+1)} \cdot a_j^{(l)}$$

$$\mathcal{L}(y, \hat{y}) = y\log(\hat{y}) + (1 - y)\log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

$$\boxed{\delta_i^{(l+1)} \triangleq \frac{\partial \mathcal{L}(y, \hat{y})}{\partial z_i^{(l+1)}}}$$

Backpropagation



Set $\Delta_{ij}^{(l)} = 0 \quad \forall l, i, j$ (Used to accumulate gradient)

For each training instance (x_k, y_k)

- Set $\mathbf{a}^{(1)} = x_k$
- Compute $\{\mathbf{a}^{(2)}, \ldots, \mathbf{a}^{(L)}\}$ via forward propagation
- Compute $\boldsymbol{\delta}^{(L)} = \mathbf{a}^{(L)} - y_i$
- Compute errors $\{\boldsymbol{\delta}^{(L-1)}, \ldots, \boldsymbol{\delta}^{(2)}\}$
- Compute gradients $\Delta_{ij}^{(l)} = \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$

Compute avg regularized gradient $D_{ij}^{(l)} = \begin{cases} \frac{1}{n}\Delta_{ij}^{(l)} + \lambda\Theta_{ij}^{(l)} & \text{if } j \neq 0 \\ \frac{1}{n}\Delta_{ij}^{(l)} & \text{otherwise} \end{cases}$

Intercept do not have regularizer

Average loss + ℓ_2 regularizer

$$\frac{1}{n}\sum_{k=1}^{n} L(y_k, \hat{y}) + \lambda \parallel \Theta \parallel_2^2$$

Convolutional Neural Networks

Multi-layer Neural Network

$$a^{(1)} = x$$

$$z^{(2)} = \Theta^{(1)} a^{(1)}$$

$$a^{(2)} = g\left(z^{(2)}\right)$$

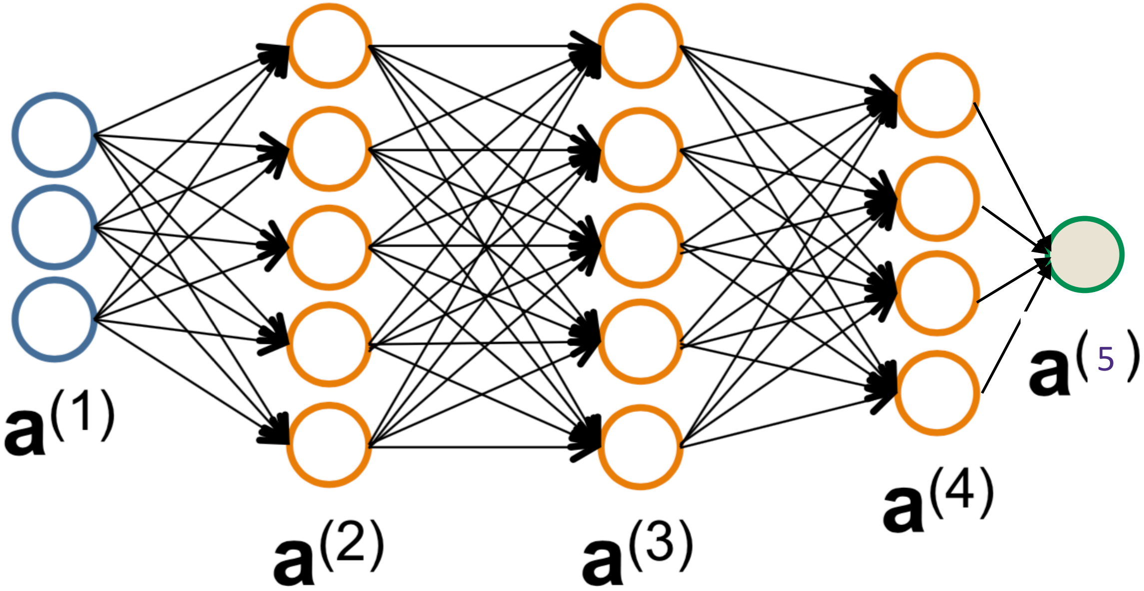
$$\vdots$$

$$z^{(l+1)} = \Theta^{(l)} a^{(l)}$$

$$a^{(l+1)} = g\left(z^{(l+1)}\right)$$

$$\vdots$$

$$\hat{y} = a^{(L+1)}$$



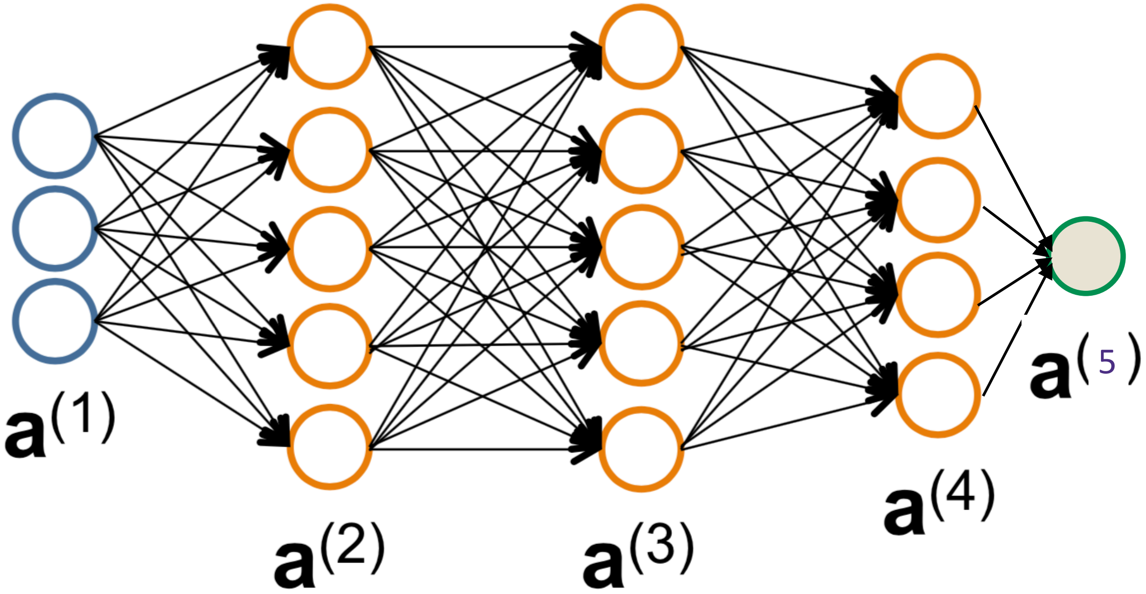
$$L(y, \hat{y}) = y\log(\hat{y}) + (1 - y)\log(1 - \hat{y})$$

$$g(z) = \frac{1}{1 + e^{-z}}$$

Binary
Logistic
Regression

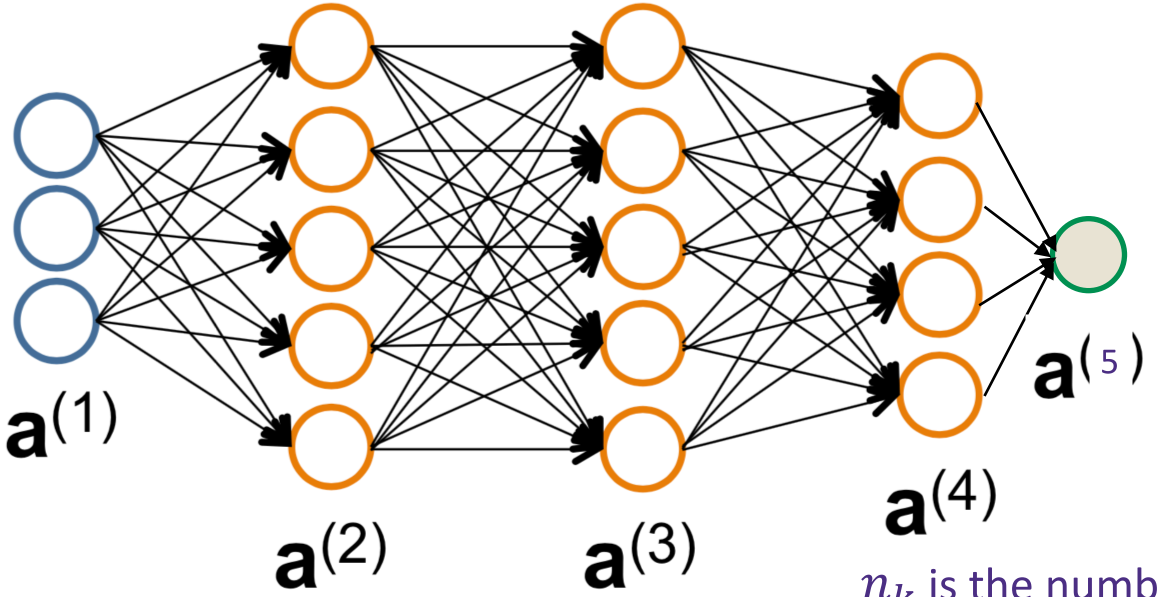
Neural Network Architecture

- The neural network architecture is defined by
    - the number of layers (depth of a network),
    - the number of nodes in each layer (width of a layer),
    - and also by **allowable edges** and **shared weights**.



Neural Network Architecture

The neural network architecture is defined by the number of layers, and the number of nodes in each layer, and also by **allowable edges** and **shared weights**.



n_k is the number of nodes in layer k

We say a layer is **Fully Connected (FC)** if all linear mappings from the current layer to the next layer are permissible.

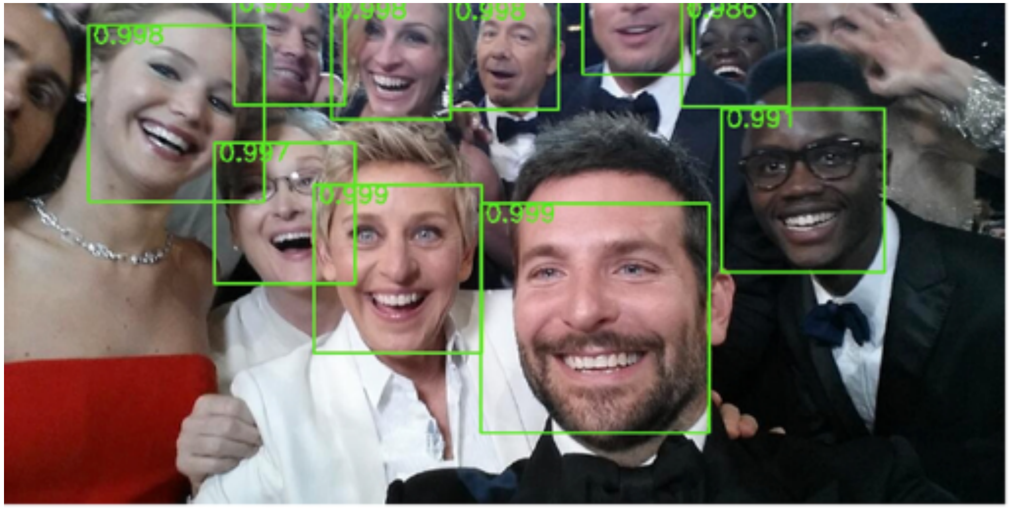
$$\mathbf{a}^{(k+1)} = g(\Theta \mathbf{a}^{(k)}) \quad \text{for any } \Theta \in \mathbb{R}^{n_{k+1} \times n_k}$$

A lot of parameters!!

$$n_1 n_2 + n_2 n_3 + \cdots + n_L n_{L+1}$$

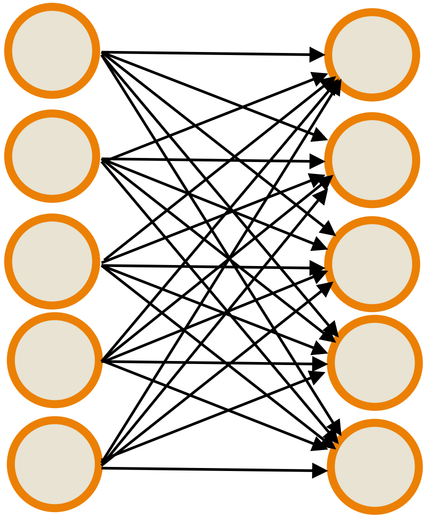
Neural Network Architecture

may
Finding faces require only local patterns



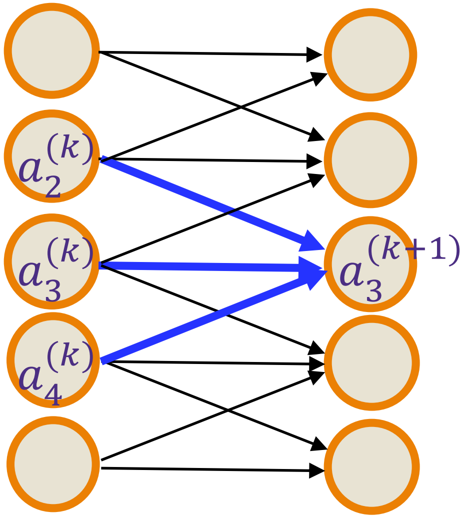
- Objects in an image are often **localized in space** so to find the faces in an image, not every pixel is important for classification
- Makes sense to drag a window across an image, focusing a local region at a time
- Although images are two-dimensional, we use one-dimensional examples to illustrate the main idea

- Similarly, to identify edges or other local structure, it makes sense to only look at **local information**



This is a
fully connected layer

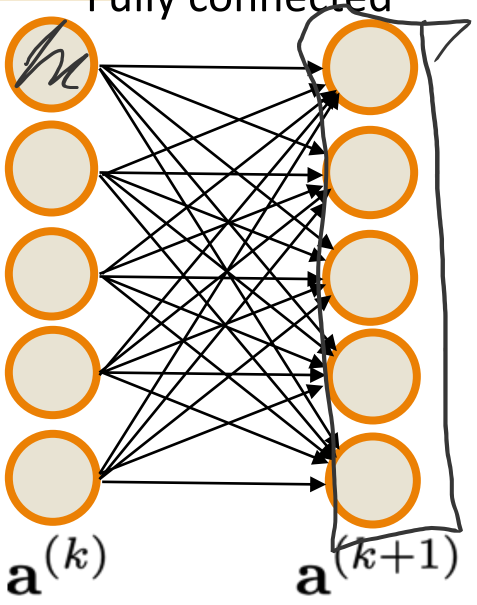
vs.



This has sparse and
local connections

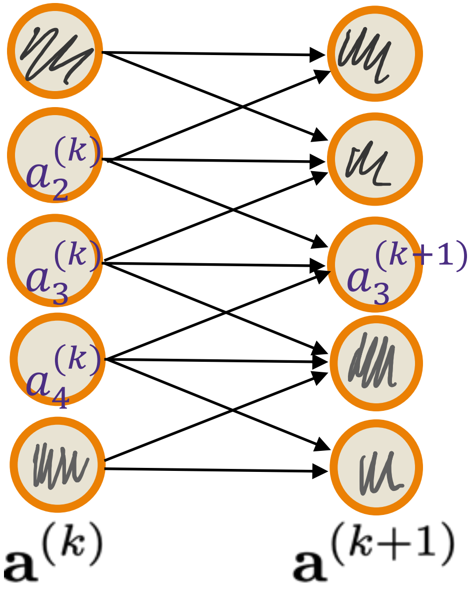
Neural Network Architecture

Fully connected



vs.

sparse and local connections



$$\Theta^{(k)} = \begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & \Theta_{0,2} & \Theta_{0,3} & \Theta_{0,4} \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & \Theta_{1,3} & \Theta_{1,4} \\ \Theta_{2,0} & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & \Theta_{2,4} \\ \Theta_{3,0} & \Theta_{3,1} & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ \Theta_{4,0} & \Theta_{4,1} & \Theta_{4,2} & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

$$\Theta^{(k)} = \begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & 0 & 0 & 0 \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & 0 & 0 \\ 0 & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & 0 \\ 0 & 0 & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ 0 & 0 & 0 & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

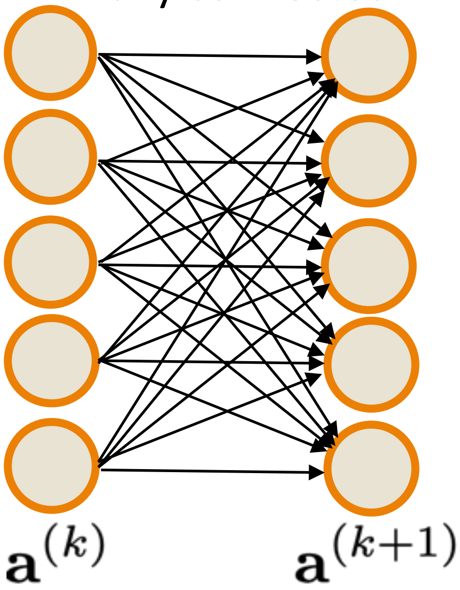
# of Parameters in this layer: n^2 $3n-2$

$$\mathbf{a}_i^{(k+1)} = g\left(\sum_{j=0}^{n-1} \Theta_{i,j}\mathbf{a}_j^{(k)}\right)$$

Neural Network Architecture

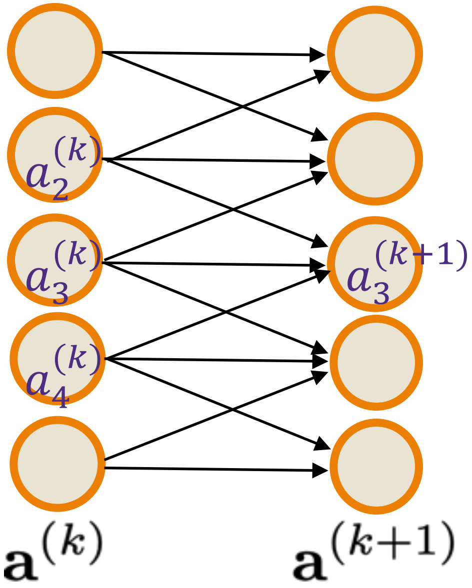
Shift invariance: A local pattern of interest can appear anywhere in the image

Fully connected



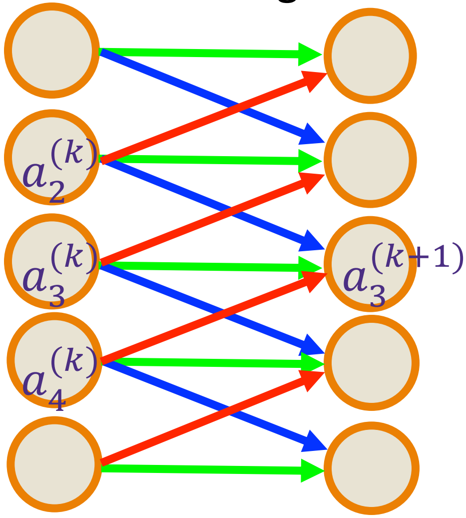
vs.

sparse and local connections



vs.

sparse local connections and shared weights



$$\Theta^{(k)} = \begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & \Theta_{0,2} & \Theta_{0,3} & \Theta_{0,4} \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & \Theta_{1,3} & \Theta_{1,4} \\ \Theta_{2,0} & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & \Theta_{2,4} \\ \Theta_{3,0} & \Theta_{3,1} & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ \Theta_{4,0} & \Theta_{4,1} & \Theta_{4,2} & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

$$\Theta^{(k)} = \begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & 0 & 0 & 0 \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & 0 & 0 \\ 0 & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & 0 \\ 0 & 0 & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ 0 & 0 & 0 & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

$$\Theta^{(k)} = \begin{bmatrix} \theta_1 & \theta_2 & 0 & 0 & 0 \\ \theta_0 & \theta_1 & \theta_2 & 0 & 0 \\ 0 & \theta_0 & \theta_1 & \theta_2 & 0 \\ 0 & 0 & \theta_0 & \theta_1 & \theta_2 \\ 0 & 0 & 0 & \theta_0 & \theta_1 \end{bmatrix}$$

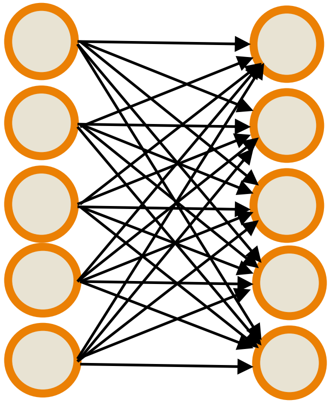
# of Parameters in this layer: n^2 $3n-2$ 3

$$\mathbf{a}_i^{(k+1)} = g\left(\sum_{j=0}^{n-1} \Theta_{i,j}\mathbf{a}_j^{(k)}\right)$$

$$\mathbf{a}_i^{(k+1)} = g\left(\sum_{j=-(m-1)/2}^{(m-1)/2} \theta_{j+(m-1)/2}\, a_i\right.$$

Neural Network Architecture

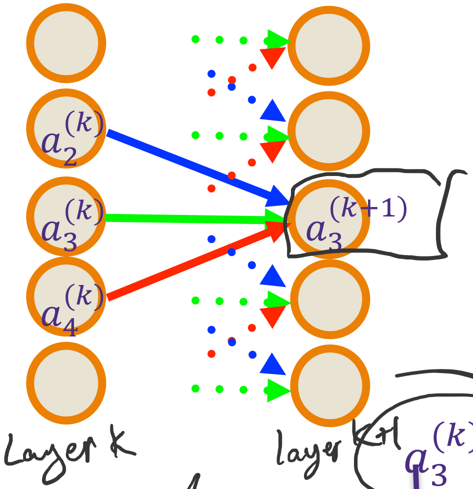
Fully Connected (FC) Layer



$$\begin{bmatrix} \Theta_{0,0} & \Theta_{0,1} & \Theta_{0,2} & \Theta_{0,3} & \Theta_{0,4} \\ \Theta_{1,0} & \Theta_{1,1} & \Theta_{1,2} & \Theta_{1,3} & \Theta_{1,4} \\ \Theta_{2,0} & \Theta_{2,1} & \Theta_{2,2} & \Theta_{2,3} & \Theta_{2,4} \\ \Theta_{3,0} & \Theta_{3,1} & \Theta_{3,2} & \Theta_{3,3} & \Theta_{3,4} \\ \Theta_{4,0} & \Theta_{4,1} & \Theta_{4,2} & \Theta_{4,3} & \Theta_{4,4} \end{bmatrix}$$

$$\mathbf{a}_i^{(k+1)} = g\left(\sum_{j=0}^{n-1} \Theta_{i,j}\mathbf{a}_j^{(k)}\right)$$

Convolutional (CONV) Layer (1 filter)



Filter with $m = 3$

$$\begin{array}{c|c} & \begin{array}{ccccc} & & a_3^{(k)} & & \end{array} \\ \hline \begin{array}{c} \\ \\ a_3^{(k+1)} \\ \\ \\ \end{array} & \begin{bmatrix} \theta_1 & \theta_2 & 0 & 0 & 0 \\ \theta_0 & \theta_1 & \theta_2 & 0 & 0 \\ 0 & \theta_0 & \theta_1 & \theta_2 & 0 \\ 0 & 0 & \theta_0 & \theta_1 & \theta_2 \\ 0 & 0 & 0 & \theta_0 & \theta_1 \end{bmatrix} \end{array}$$

K

$$\mathbf{a}_i^{(k+1)} = g\Big(\sum_{j=-(m-1)/2}^{(m-1)/2} \theta_{j+(m-1)/2}\, a_{i+j}^{(k)}\Big) = g([\theta \star$$

$\star$=Convolution

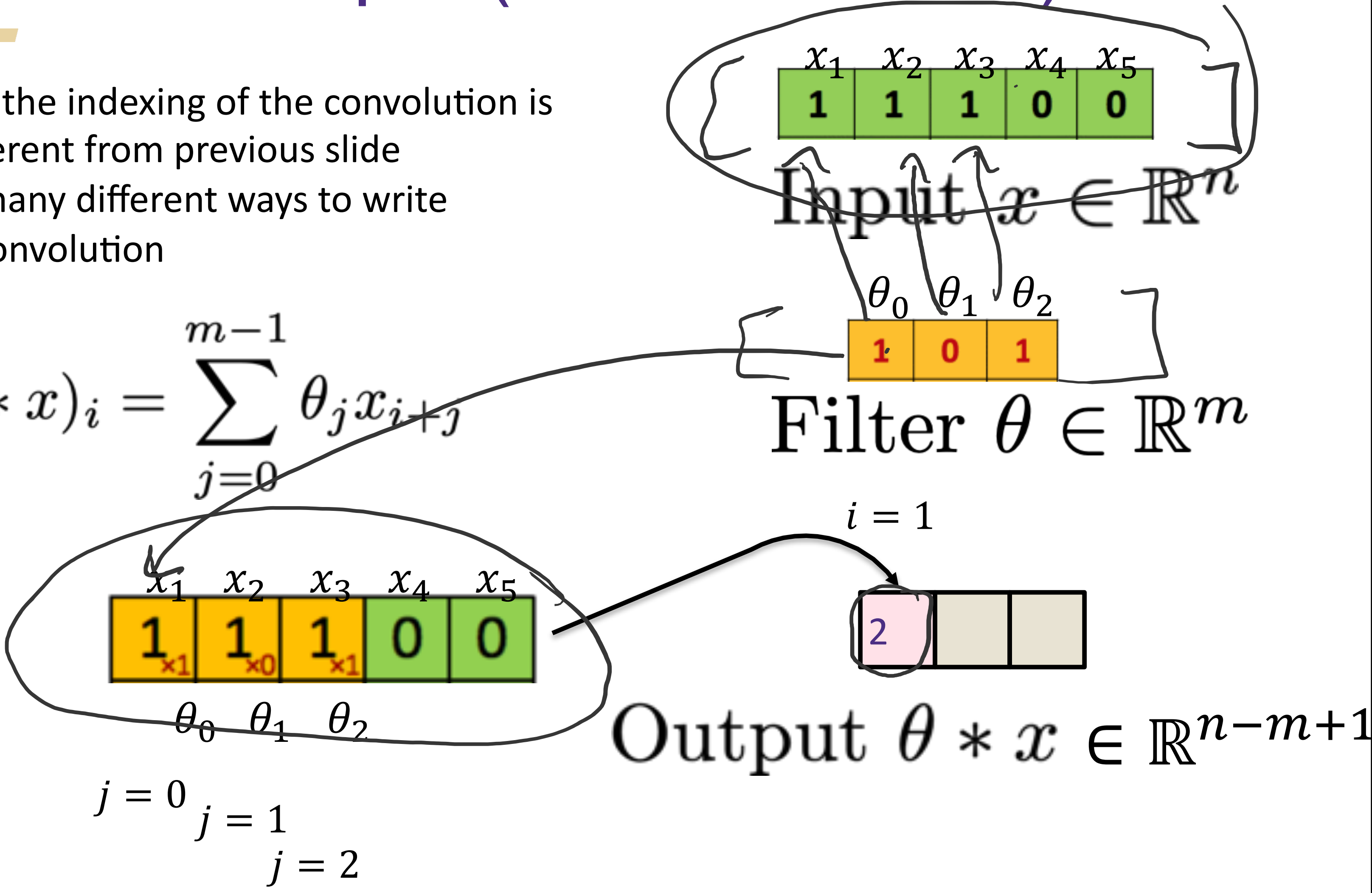
$\theta = (\theta_0, \ldots, \theta_{m-1}) \in \mathbb{R}^m$ is referred to as a "filter"

Because of shift invariance and locality of computer vision tasks, convolution is extremely powerful

Example (1d convolution)

- Notice that the indexing of the convolution is slightly different from previous slide
- There are many different ways to write the same convolution

$$(\theta * x)_i = \sum_{j=0}^{m-1} \theta_j x_{i+j}$$



x_1 x_2 x_3 x_4 x_5

1 1 1 0 0

Input $x \in \mathbb{R}^n$

θ_0 θ_1 θ_2

1 0 1

Filter $\theta \in \mathbb{R}^m$

$i = 1$

2

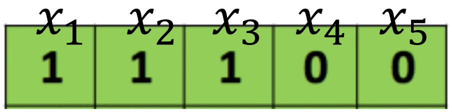
Output $\theta * x \in \mathbb{R}^{n-m+1}$

$j = 0$

$j = 1$

$j = 2$

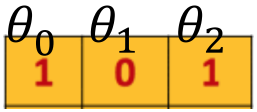
Example (1d convolution)



x_1	x_2	x_3	x_4	x_5
1	1	1	0	0

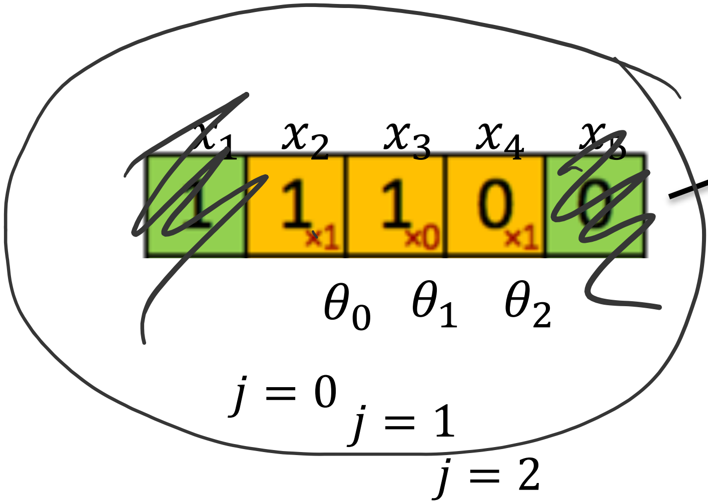
Input $x \in \mathbb{R}^n$

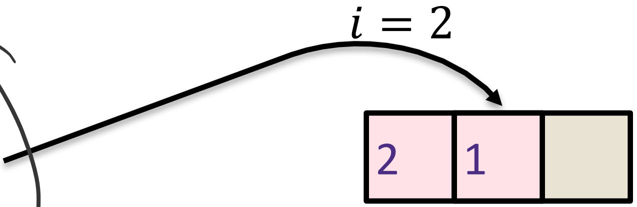
$$(\theta * x)_i = \sum_{j=0}^{m-1} \theta_j x_{i+j}$$



θ_0	θ_1	θ_2
1	0	1

Filter $\theta \in \mathbb{R}^m$

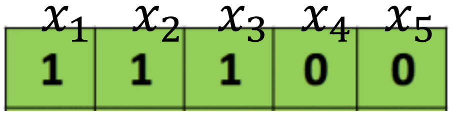




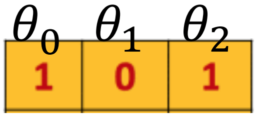
Output $\theta * x \in \mathbb{R}^{n-m+1}$

Example (1d convolution)

$$(\theta * x)_i = \sum_{j=0}^{m-1} \theta_j x_{i+j}$$

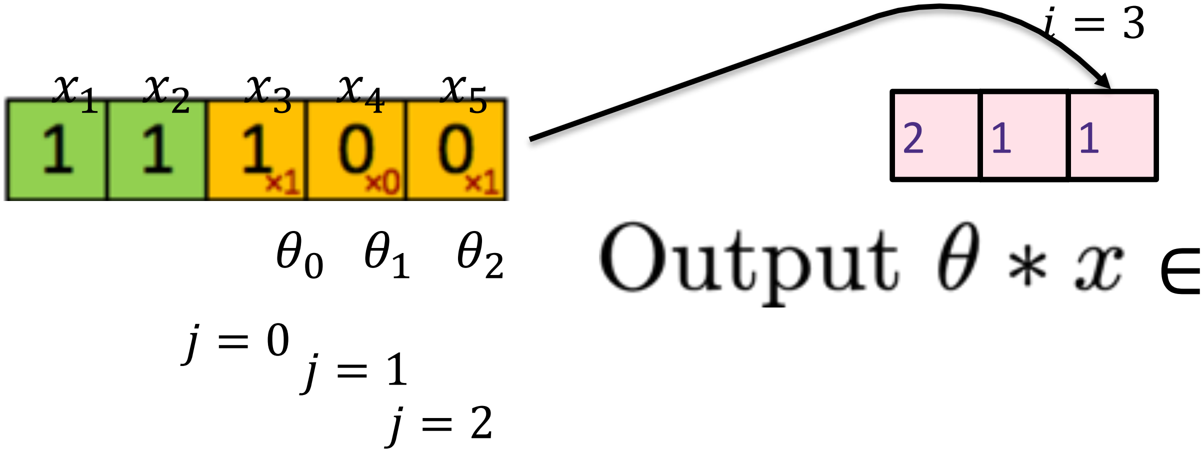


Input $x \in \mathbb{R}^n$



$= K$

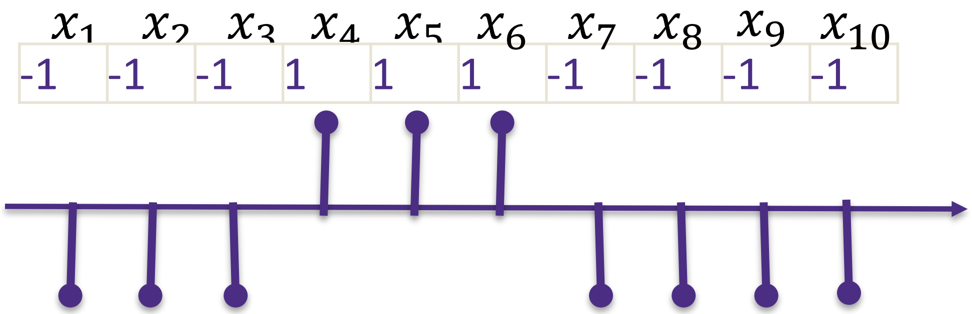
Filter $\theta \in \mathbb{R}^m$



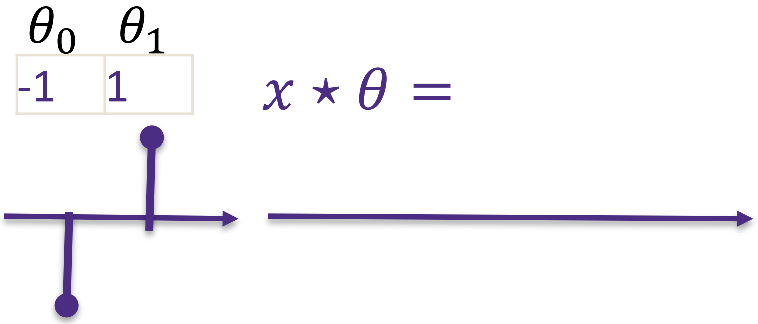
Output $\theta * x \in \mathbb{R}^{n-m+1}$

1d convolution

- Each filter finds a specific pattern over the input

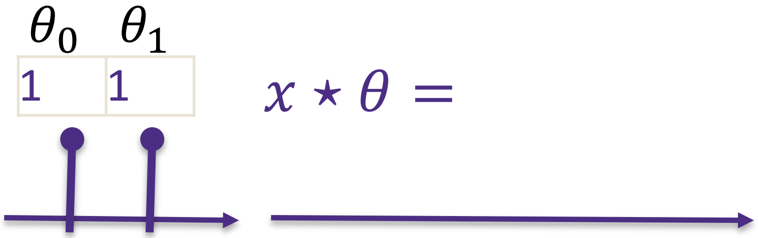


Filter 1



$x \star \theta =$

Filter 2



$x \star \theta =$

- We use many such convolutional filters per layer in practice
- Each convolutional filter output vector (or a matrix if 2D convolution) is called a cha

Convolution of images (2d convolution)

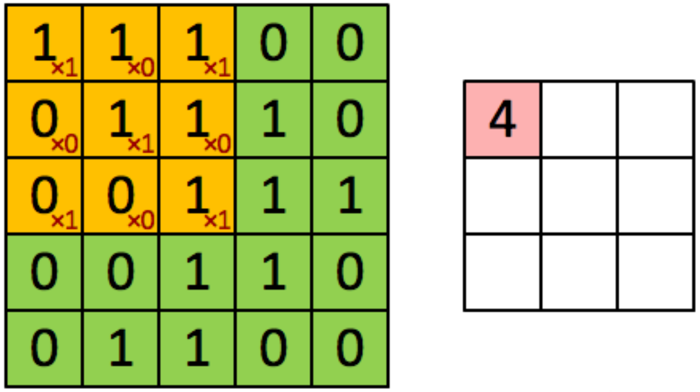
$$(I * K)(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n).$$

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

Image I

1	0	1
0	1	0
1	0	1

Filter K



1 ×1	1 ×0	1 ×1	0	0
0 ×0	1 ×1	1 ×0	1	0
0 ×1	0 ×0	1 ×1	1	1
0	0	1	1	0
0	1	1	0	0

Image

4		

Convolved
Feature
$I * K$

Convolution of images

- These are hand-crafted filters,
  to illustrate what the weights of a filter mean
- Filter in a Convolutional Neural Network (CNN) is learned, and we might be able
  to interpret what we learned

$$(I * K)(i,j) = \sum_m \sum_n I(i+m, j+n) K(m,n)$$

Image I

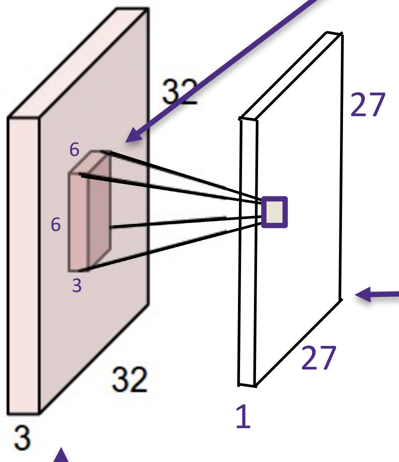
Operation	Filter K	Convolved Image $I * K$
Edge detection	$\begin{bmatrix} 1 & 0 & -1 \\ 0 & 0 & 0 \\ -1 & 0 & 1 \end{bmatrix}$	
	$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$	
	$\begin{bmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{bmatrix}$	
Sharpen	$\begin{bmatrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{bmatrix}$	
Box blur (normalized)	$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$	
Gaussian blur (approximation)	$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	

Lowel
dimen

Stacking convolved images

$$K \in \mathbb{R}^{m \times m \times r}$$

- If we use a convolutional layer with 1 filter of size m*m*r=6*6*3, then the output is a matrix of dimension (n+1-m)*(n+1-m)=27*27



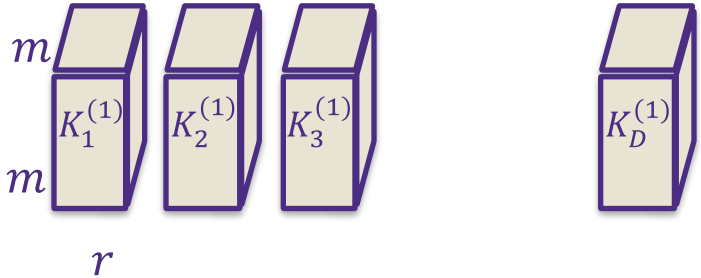
$$x \star K \in \mathbb{R}^{(n+1-m) \times (n+1-m)}$$

- Input is a multi-array or a tensor, because it has 3 color channels

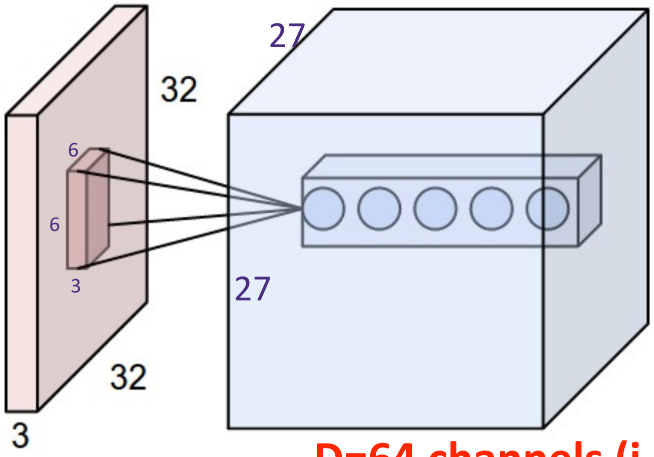
$$x \in \mathbb{R}^{n \times n \times r}$$

Stacking convolved images

- Typical convolutional layer has multiple filters to capture multiple patterns
- Each one is called a channel
- Each channel has a filter of the same size m*m*r but with different weights



- Each channel outputs a matrix of dimension (n+1-m)*(n+1-m)
- Put together the output is a tensor of dimension (n+1-m)*(n+1-m)*D

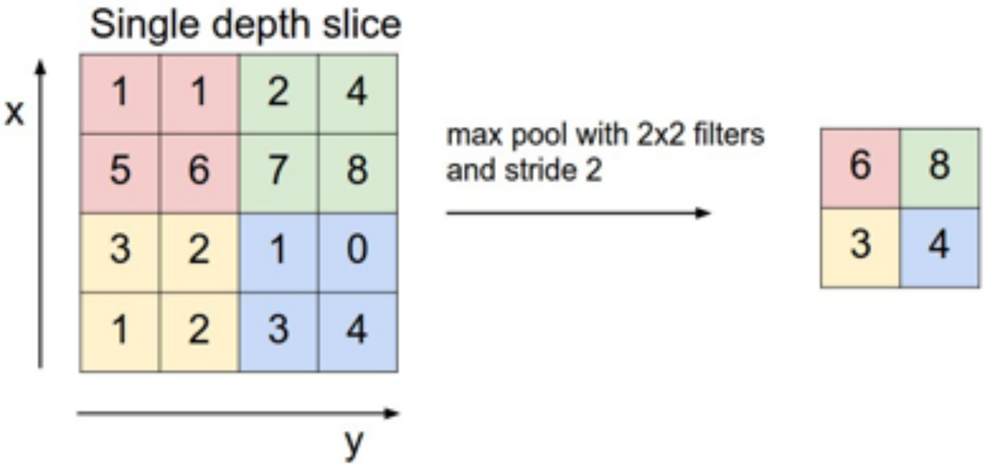


Repeat with D filters!

D=64 channels (i.e. filters)

Max Pooling gives a summary of a region

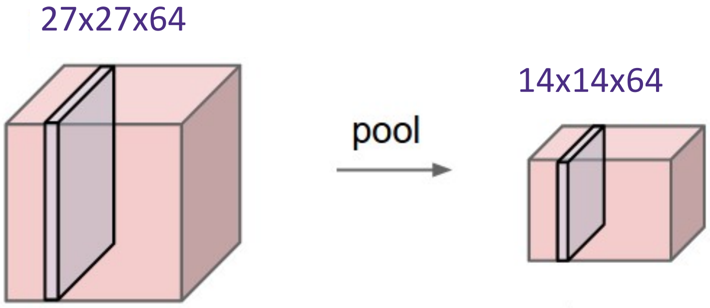
Pooling reduces the dimension and can be interpreted as “This filter had a high response in this general region”


Single depth slice

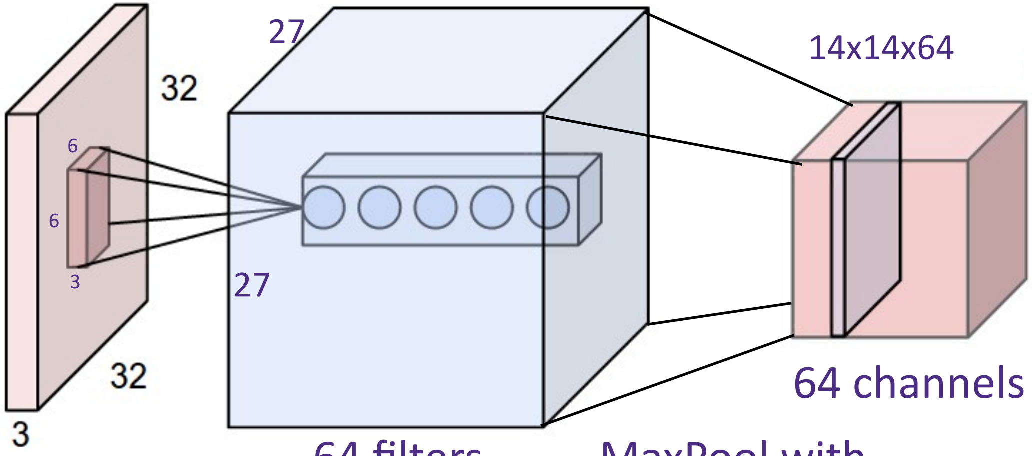
1	1	2	4
5	6	7	8
3	2	1	0
1	2	3	4

max pool with 2x2 filters and stride 2

6	8
3	4



Pooling Convolution layer

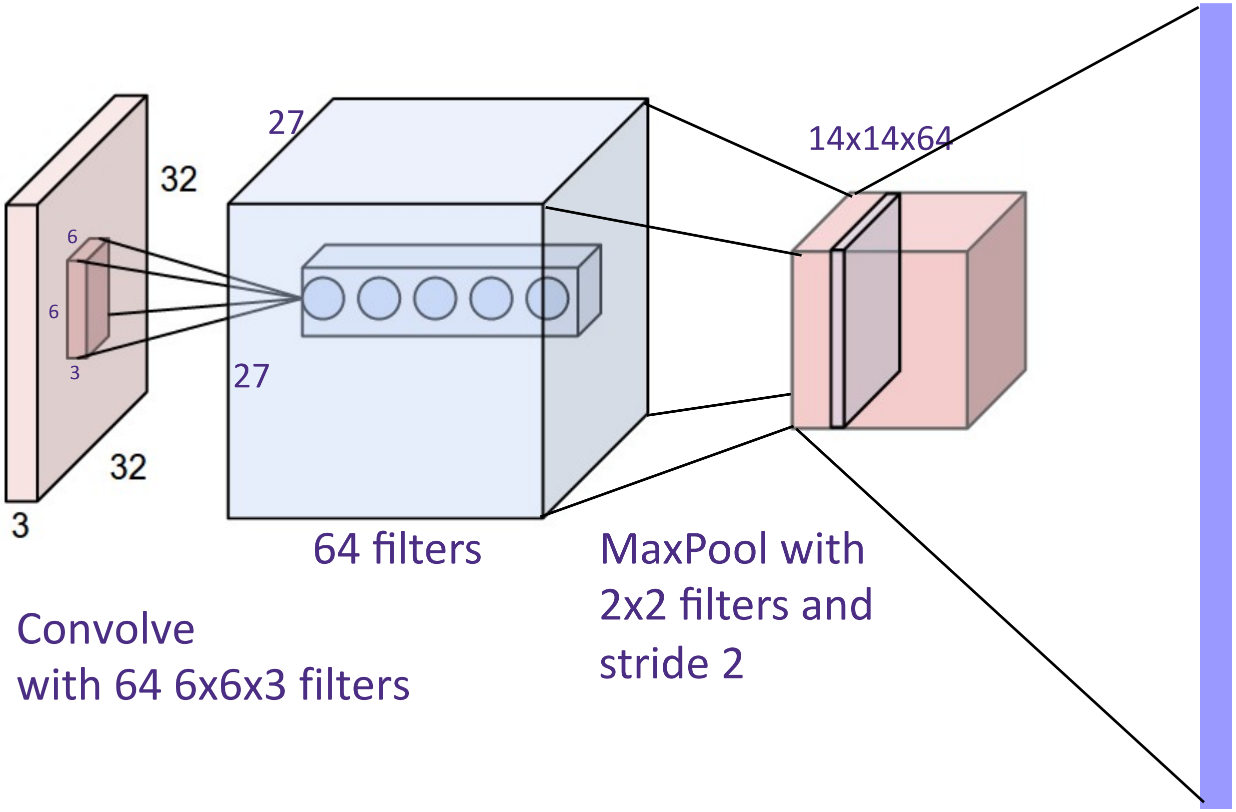


64 filters

MaxPool with
2x2 filters and
stride 2

Convolve
with 64 6x6x3 filters

Flattening



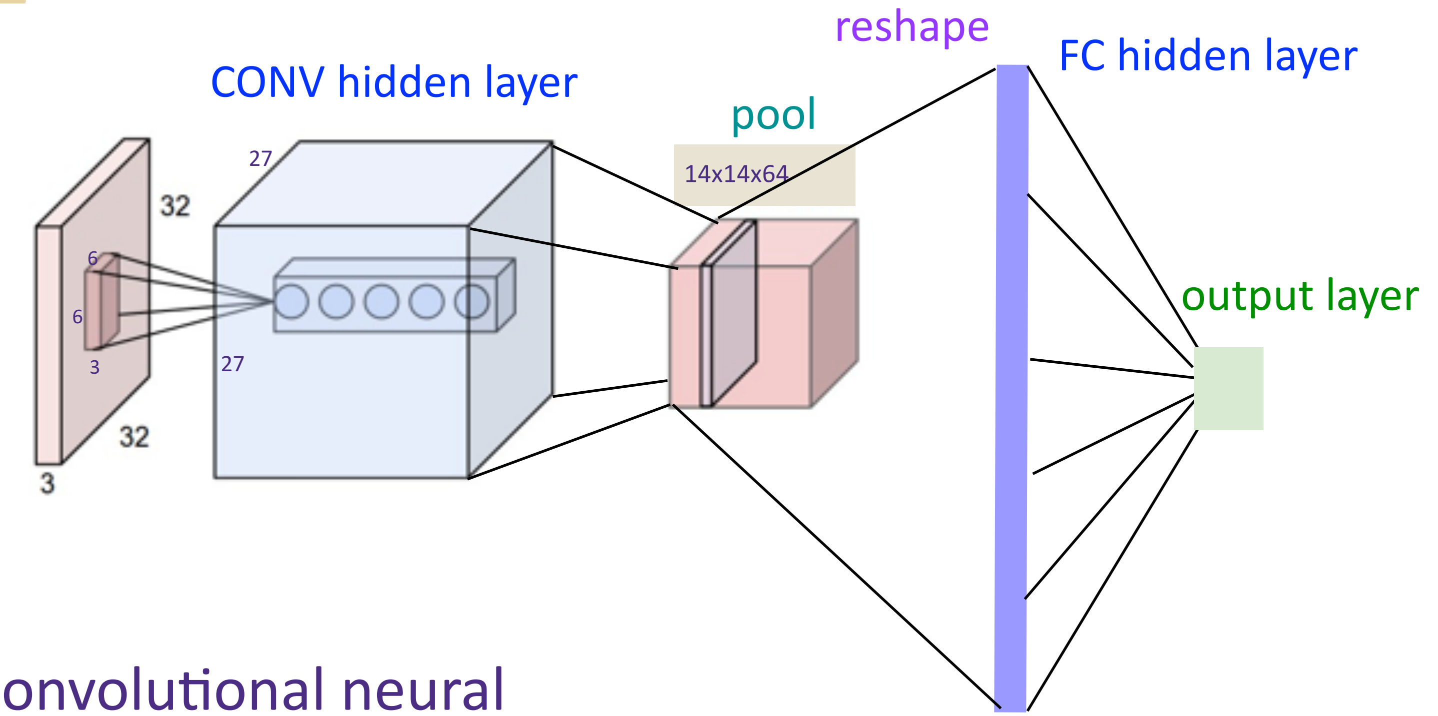
64 filters

Convolve
with 64 6x6x3 filters

MaxPool with
2x2 filters and
stride 2

Flatten into a single
vector of size
14*14*64=12544

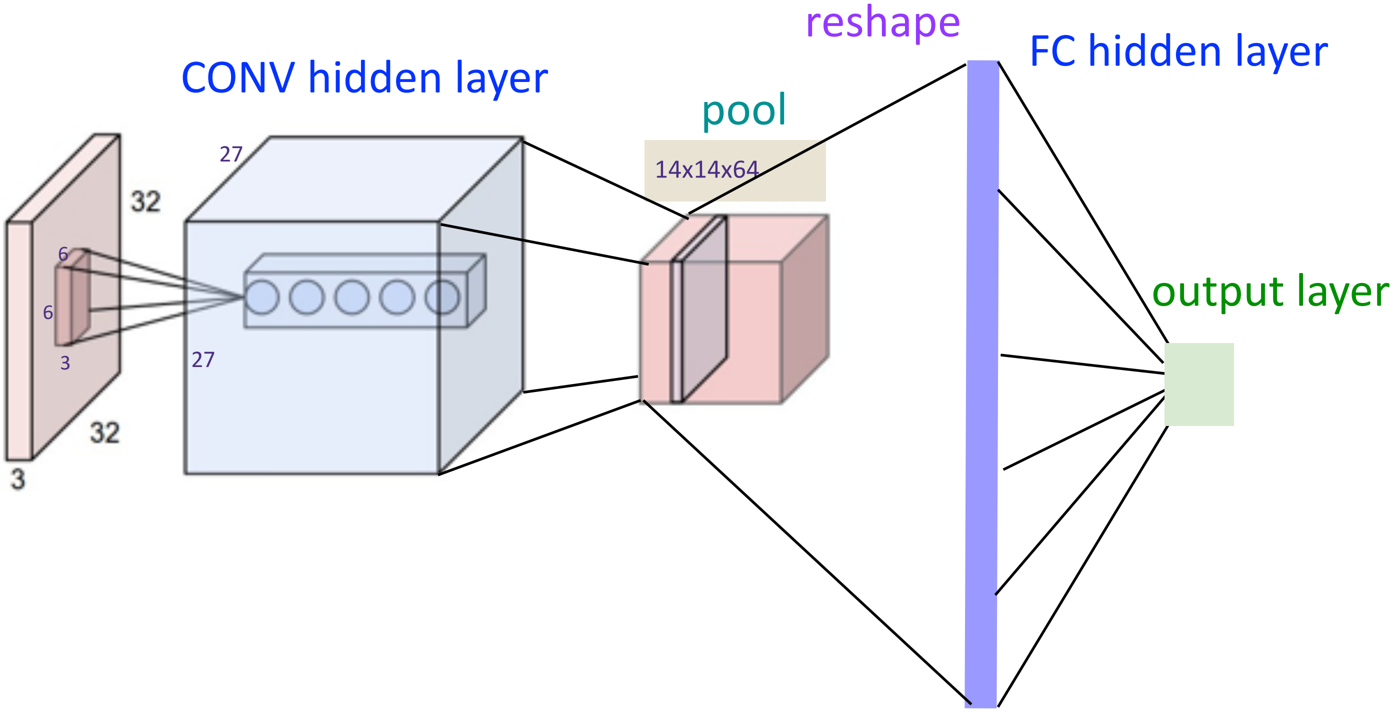
Training Convolutional Networks



Recall: Convolutional neural networks (CNN) are just regular fully connected (FC) neural networks with some connections removed and some weights shared.

Train with SGD!

Training Convolutional Networks



reshape

FC hidden layer

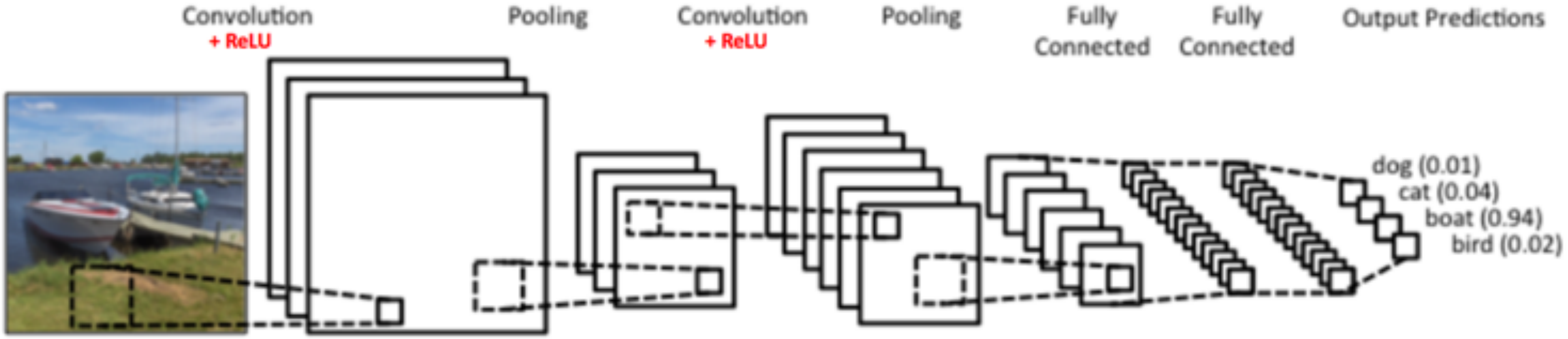
CONV hidden layer

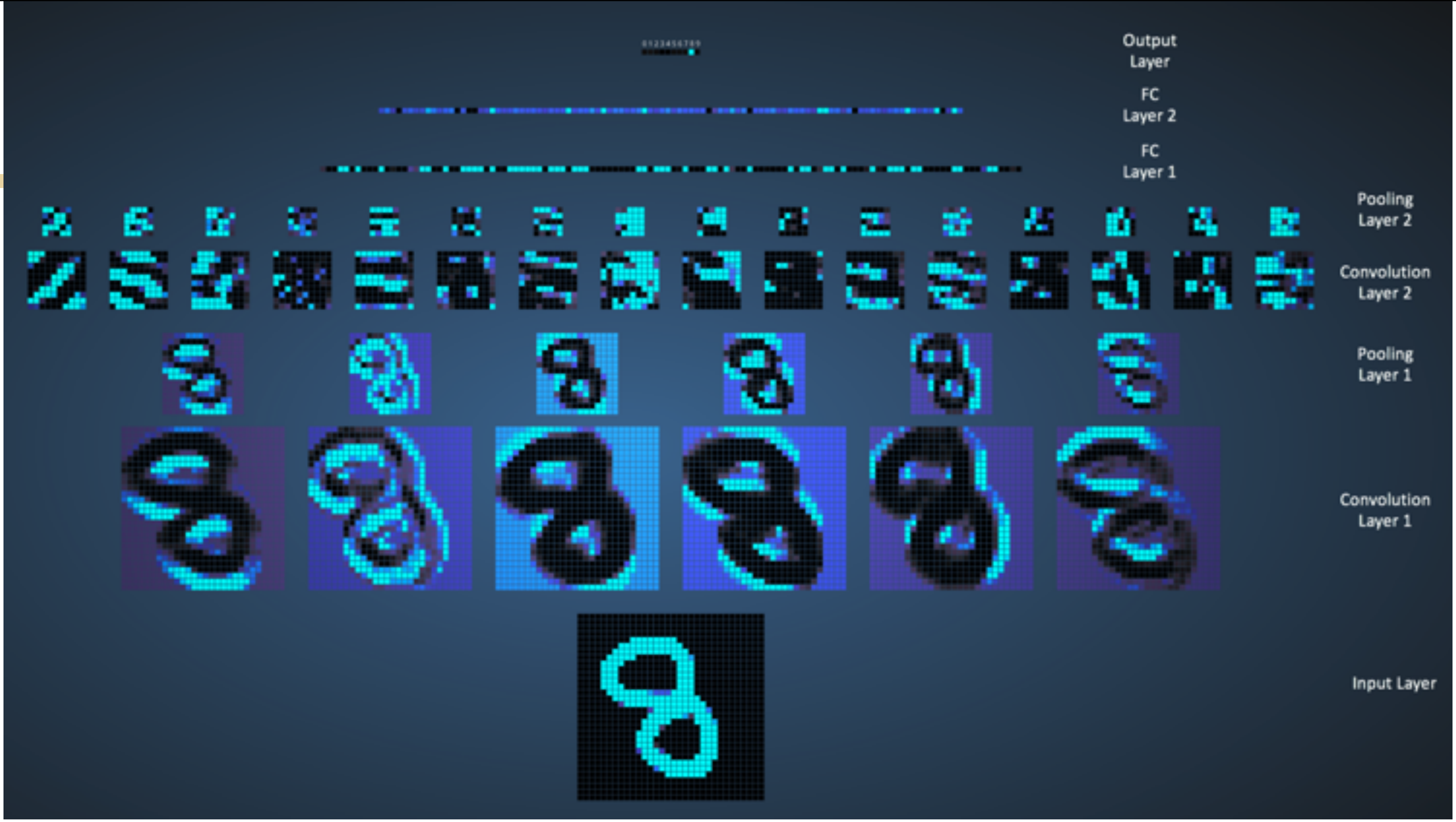
pool

14x14x64

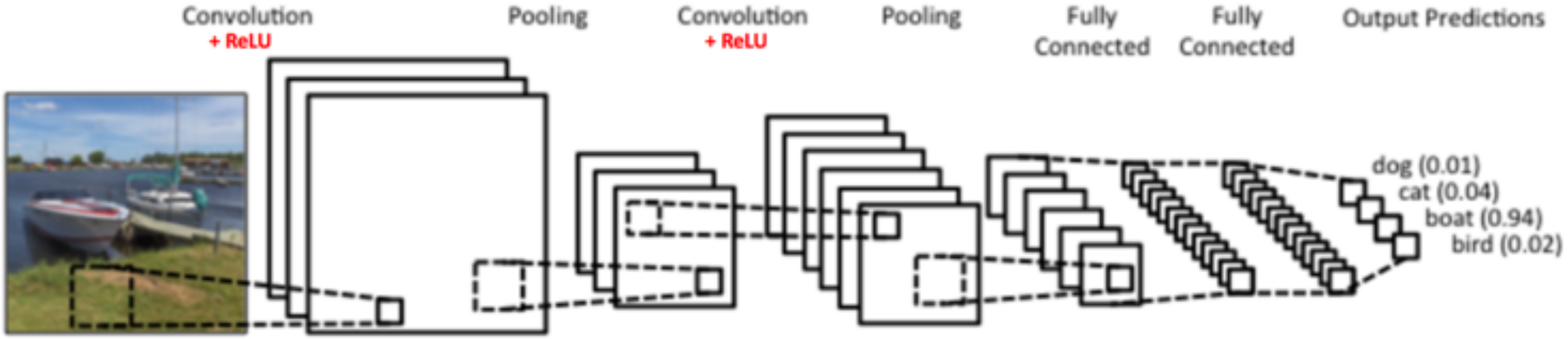
output layer

Real example network: LeNet





Real example network: LeNet

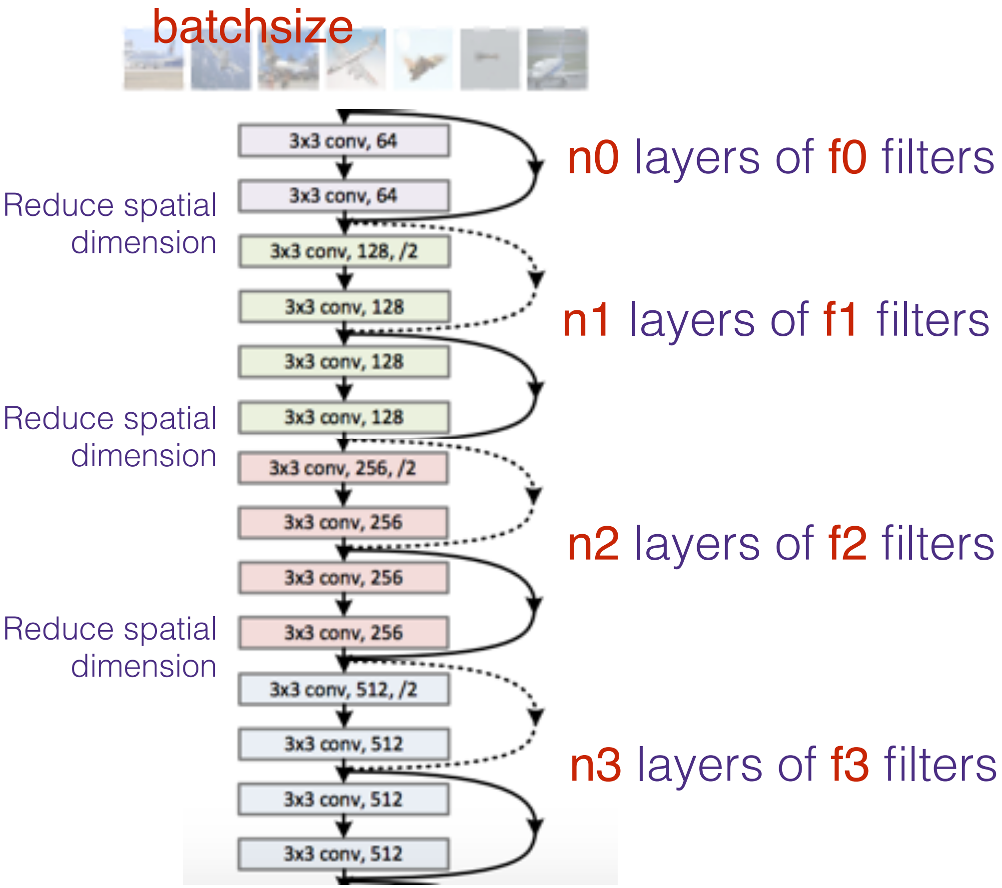


Real networks

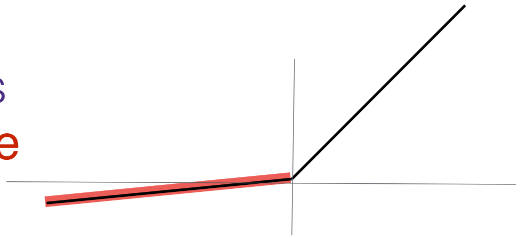
Residual Network of
[HeZhangRenSun'15]

Modern networks have dozens of parameters to tune.

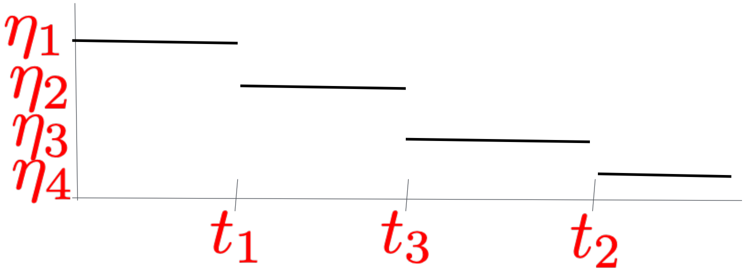
Data augmentation?
Batch norm?



RELU leakiness
slope



Learning rate schedule



Remarks

- Convolution is a fundamental operation in signal processing. Instead of hand-engineering the filters (e.g., Fourier, Wavelets, etc.) **Deep Learning *learns* the filters and CONV layers with back-propagation**, replacing fully connected (FC) layers with convolutional (CONV) layers
- **Pooling** is a dimensionality reduction operation that summarizes the output of convolving the input with a filter

- Typically the last few layers are **Fully Connected (FC)**, with the interpretation that the CONV layers are feature extractors, preparing input for the final FC layers. Can replace last layers and retrain on different dataset+task.

- Just as hard to train as regular neural networks.
- More exotic network architectures for specific tasks

Vision transformers

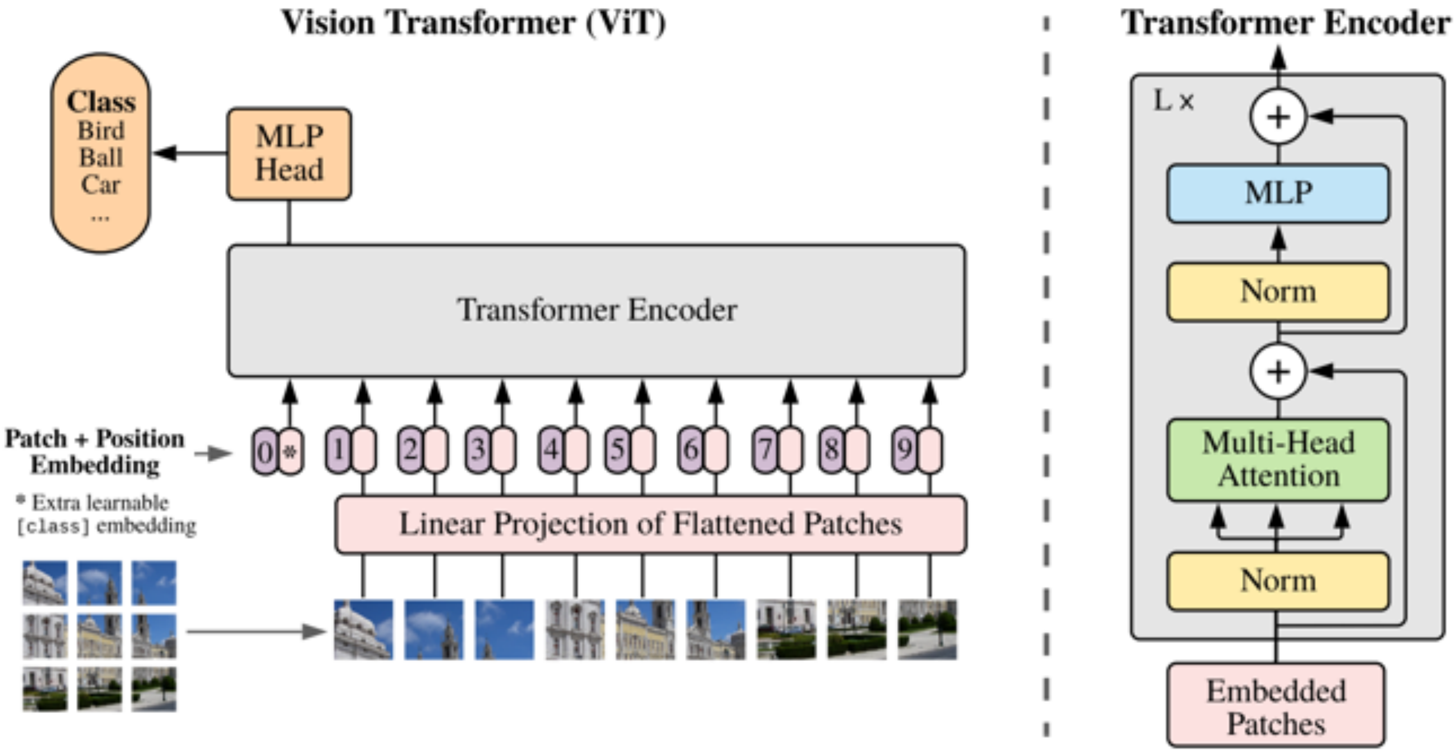


Figure 1: Model overview. We split an image into fixed-size patches, linearly embed each of them, add position embeddings, and feed the resulting sequence of vectors to a standard Transformer encoder. In order to perform classification, we use the standard approach of adding an extra learnable "classification token" to the sequence. The illustration of the Transformer encoder was inspired by Vaswani et al. (2017).

2d Convolution Layer

- **Example: 200x200 image**
  - Fully-connected, 400,000 hidden units = 16 billion parameters
  - Locally-connected, 400,000 hidden units 10x10 fields = 40 million params

    or channels or filter
  - Local connections capture local dependencies