# CNN

Convolutional Neural Network is a sequence of Convolution Layers, interspersed with activation functions.

Convolution layer: convolve the filter with the image — preserve spatial structure

Assume input is $W_1 \times H_1 \times C_1$

Conv layer needs 4 hyperparameters:

- number of filters K
- filter size F
- stride S
- zero padding P

This will produce an output of $W_2 \times H_2 \times K$ where

- $W_2 = (W_1 - F + 2P)/S + 1$
- $H_2 = (H_1 - F + 2P)/S + 1$

Number of parameters: $F^2CK$ and $K$ biases

Pooling layer: makes the representations smaller and more manageable; operates over each activation map independently

Fully Connected Layer (FC layer): contains neurons that connect to the entire input volume, as in ordinary Neural Networks

Activation functions:

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

tanh

ReLU (Rectified Linear Unit): $f(x) = \max(0, x)$ - does nor saturate, computationally efficient, convergers faster

Leaky ReLU: $f(x) = \max(0.01x, x)$ - compared to ReLU, will not die

ELU (Exponential Linear Unit): $f(x) = \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$

Maxout "Neuron": $\max(w_1^T x + b_1, w_2^T x + b_2)$ - generalizes ReLU and Leaky ReLU, but doubles the number of parameters

Batch normalization: consider a batch of activations at some layer, to make each dimension zero-mean unit-variance

1. compute mean and variance independently for each dimension
   $\mu_j = \frac{1}{N}\Sigma_1^N x_{i,j}, \sigma_j^2 = \frac{1}{N}\Sigma_1^N (x_{i,j} - \mu_j)^2$
2. normalize: $\hat{x_{i,j}} = \frac{x_{i,j} - \mu_j}{\sqrt{\sigma_j^2 + \epsilon}}$
3. scale and shift (to have some flexibility): $\hat{y_{i,j}} = \gamma_j \hat{y_{i,j}} + \beta_j$

Data preprocessing:

original data → zero-centered data → normalized data

original data → decorrelated data → whitened data (by PCA and whitening of the data)

Weight initialization:

Initialization too small — activations go to zero, gradients also zero, no learning

Initialization too big — activations saturate (for tanh), gradients zero, no learning

Initialization just right — nice distribution of activations at all layer, learning proceeds nicely

Activation statistics: e.g. W = 0.05 * np.random.randn(Din, Dout)

Xavier Initialization: e.g. W = np.random.randn(Din, Dout) / np.sqrt(Din)

Optimization algorithms:

SGD + Momentum: build up "velocity" as a running mean of gradients; Rho gives "friction"; typically rho=0.9 or 0.99

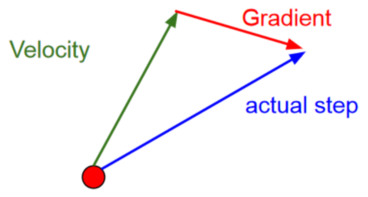
Nesterov Momentum

$$v_{t+1} = \rho v_t - \alpha \nabla f(\boxed{x_t + \rho v_t})$$
$$x_{t+1} = x_t + v_{t+1}$$

Annoying, usually we want update in terms of $x_t, \nabla f(x_t)$

Change of variables $\tilde{x}_t = x_t + \rho v_t$ and rearrange:

$$\begin{aligned} v_{t+1} &= \rho v_t - \alpha \nabla f(\tilde{x}_t) \\ \tilde{x}_{t+1} &= \tilde{x}_t - \rho v_t + (1+\rho) v_{t+1} \\ &= \tilde{x}_t + v_{t+1} + \rho(v_{t+1} - v_t) \end{aligned}$$



“Look ahead” to the point where updating using velocity would take us; compute gradient there and mix it with velocity to get actual update direction

## AdaGrad & RMSProp (Leaky AdaGrad)

AdaGrad

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared += dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

RMSProp

```
grad_squared = 0
while True:
  dx = compute_gradient(x)
  grad_squared = decay_rate * grad_squared + (1 - decay_rate) * dx * dx
  x -= learning_rate * dx / (np.sqrt(grad_squared) + 1e-7)
```

## Adam

```
first_moment = 0
second_moment = 0
for t in range(1, num_iterations):
  dx = compute_gradient(x)
  first_moment = beta1 * first_moment  + (1 - beta1) * dx
  second_moment = beta2 * second_moment + (1 - beta2) * dx * dx
  first_unbias = first_moment / (1 - beta1 ** t)
  second_unbias = second_moment / (1 - beta2 ** t)
  x -= learning_rate * first_unbias / (np.sqrt(second_unbias) + 1e-7))
```

Momentum

Bias correction

AdaGrad / RMSProp

Bias correction for the fact that first and second moment estimates start at zero

Adam with beta1 = 0.9, beta2 = 0.999, and learning_rate = 1e-3 or 5e-4 is a great starting point for many models!

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Common methods:

- Dropout
- Model ensembles
- Transfer learning

**CNN Architectures**

AlexNet (2012): first CNN-based winner

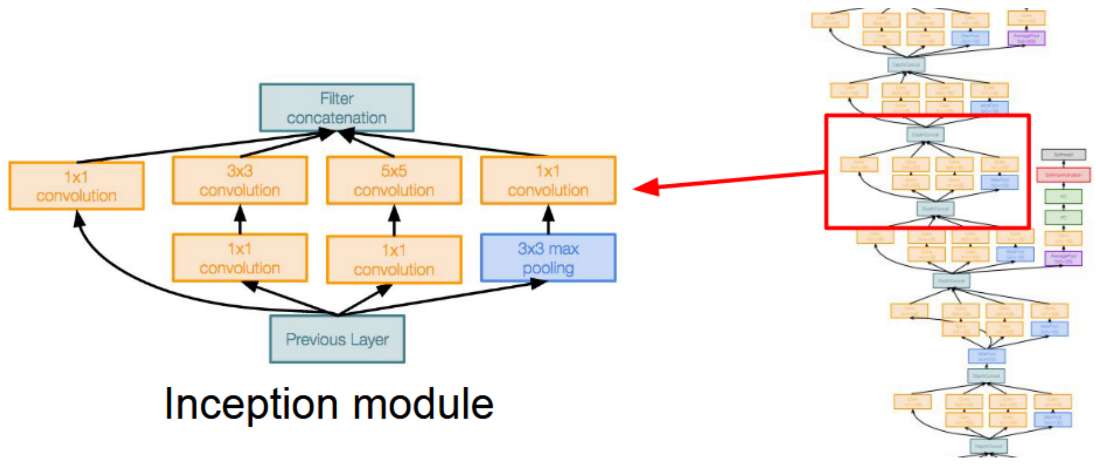
- ARCHITECTURE: CONV1 → MAX POOL1 → NORM1 → CONV2 → MAX POOL2 → NORM2 → CONV3 → CONV4 → CONV5 → MAX POOL3 → FC6 → FC7 → FC8

ZFNet (2013): improved hyperparameters over AlexNet

VGGNet (2014): small filters (3x3 conv), deeper networks (16/19 layers)

GoogleNet (2014): deeper networks, with computational efficiency

- “inception module”: design a good local network topology (network within a network) andthen stack these modules on top of each other
- add “1*1 conv, 64 filters” bottlenecks to reduces dimension (depth)
- global avg pool instead of FC layers



ResNet (2015): very deep networks using residual connections