Convolutional networks

(and other image networks)

Brooks Paige

COMP0171 Week 6

What kind of "architectures" make sense?

What kind of deep learning architectures should we use? We can basically stack **any differentiable computation**, with free parameters to optimize.

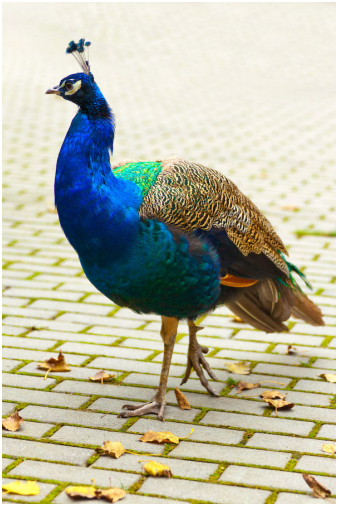
- In general, architecture design choices should be guided by the structure of the data
- One classic family of models for image processing is based on **convolutions**, a natural way of encoding certain types of invariances in images

Desired model invariances

Suppose we want to classify the picture on the right as **bird** or **not bird**.

We have a training set with many similar photos.

What sort of operations could we apply which would not affect whether or not the class label is **bird**?



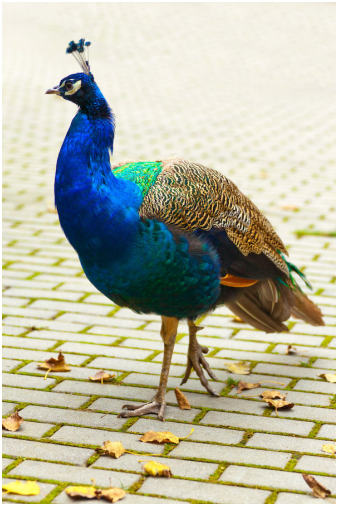
Desired model invariances

Suppose we want to classify the picture on the right as **bird** or **not bird**.

We have a training set with many similar photos.

What sort of operations could we apply which would not affect whether or not the class label is **bird**?

- Flipping horizontally
- Small translations or rotations
- Small scaling up or down
- Some color shifts (e.g. to grayscale)



Data augmentation

The easiest way to have a learned model be robust to these sorts of transformations is to **include them in the training dataset**.

A **data augmentation** approach takes the original training dataset and constructs a new, larger training dataset that applies *label-preserving* transformations.

This is the direct approach. (These transformations can even be done on-the-fly when processing mini-batches of data!)

Invariant features

An alternative approach is to try to build invariances directly into the network architecture.

In network architectures based on **convolutional layers**, we note that

- pixels which are next to each other contain similar information, so it would make sense to extract local features
- features should be somehow "the same" regardless where in the image a feature is located (i.e., a bird in the bottom left of an image should be treated the same as a bird in the top right)

Convolution operation

In an abstract sense, suppose we have a function $f(t)$ and a weighting function (or filter) $g(t)$, defined on $t \in \mathbb{R}$.

A one-dimensional convolution is written as $f * g$, and is defined as

$$(f * g)(t) = \int f(a)g(t-a)\mathrm{d}a.$$

Convolution is commutative, with $f * g = g * f$.

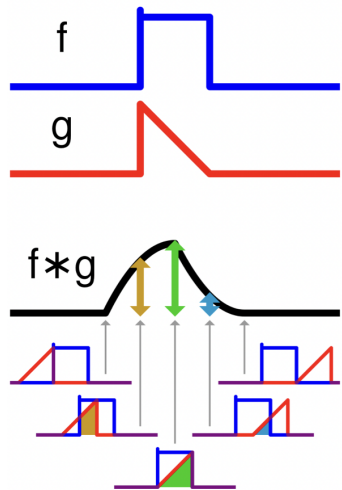


Figure: wikipedia

Discrete convolution

We're mostly going to be interested in discrete convolutions. The one-dimensional analog would be convolving an input $\mathbf{x} \in \mathbb{R}^D$ with a filter $\mathbf{w} \in \mathbb{R}^M$, $M < D$.

$$(\mathbf{x} * \mathbf{w})_d = \sum_m x_m w_{d-m}.$$

In general we will have to "pad" inputs appropriately, e.g. with zeros, or boundary values, or else consider only the "valid" region.

Discrete convolution as matrix multiplication

This can be thought of as a sort of matrix multiplication, but with a constrained matrix (a **Toepliz matrix**). E.g. if $M = 3$ and $D = 5$,

$$\mathbf{x} * \mathbf{w} = \begin{bmatrix} w_1 & 0 & 0 & 0 & 0 \\ w_2 & w_1 & 0 & 0 & 0 \\ w_3 & w_2 & w_1 & 0 & 0 \\ 0 & w_3 & w_2 & w_1 & 0 \\ 0 & 0 & w_3 & w_2 & w_1 \\ 0 & 0 & 0 & w_3 & w_2 \\ 0 & 0 & 0 & 0 & w_3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

That is, convolution is a special sort of linear operation where the matrix corresponding to the transformation has a special structure (and **fewer parameters** if we are learning it!)

Dealing with edges

That transformation matrix was created by placing the filter $\mathbf{w}$ in columns, at different offsets.

An alternative to the version on the previous slide would be

$$\mathbf{x} * \mathbf{w} = \begin{bmatrix} w_3 & w_2 & w_1 & 0 & 0 \\ 0 & w_3 & w_2 & w_1 & 0 \\ 0 & 0 & w_3 & w_2 & w_1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix}$$

This is sometimes referred as computing only the "valid" section of the convolution, i.e. where the entire filter $\mathbf{w}$ overlaps with $\mathbf{x}$.

The version on the previous slide would be equivalent to padding the data $\mathbf{x}$ with zeros, i.e. $\mathbf{x}^\top := [0\ 0\ x_1\ x_2\ x_3\ x_4\ x_5\ 0\ 0]$.

Convolutions on images

For 2d images, we will want to convolve along two axes simultaneously, e.g. for a matrix $\mathbf{X}$ and 2d filter $\mathbf{W}$ we might have

$$(\mathbf{X} * \mathbf{W})_{i,j} = \sum_m \sum_n x_{m,n} w_{i-m,j-n}.$$

This can also be represented as a matrix (a "doubly block circulant matrix"), though it's a bit cumbersome. (Note we would never instantiate such a matrix when performing a convolution!)

2d convolution schematic

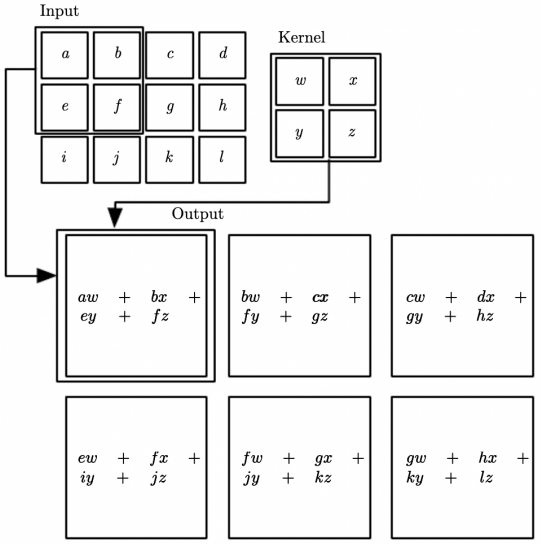
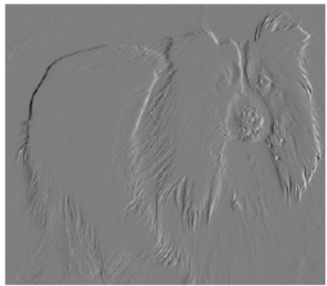


Figure: Goodfellow et al.

2d convolution example



Convolutions of filters are common throughout "classical" computer vision, e.g. in edge detectors.

In this example we convolve the 2d filter $\begin{bmatrix} 1 & -1 \end{bmatrix}$ with the image on the left.

Figure: Goodfellow et al.

Use in deep learning models

In deep learning models, we typically define a series of filters, interspersed with nonlinearities, and sometimes also with **pooling layers**, replace a spatial region of the output with a summary statistic.

Most common is "max pooling", where we take the maximum. "Average pooling" is also common.

We often refer to this entire structure as a "convolutional block".

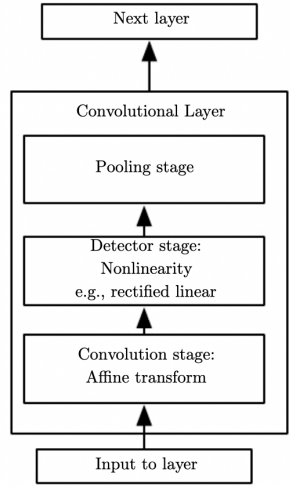
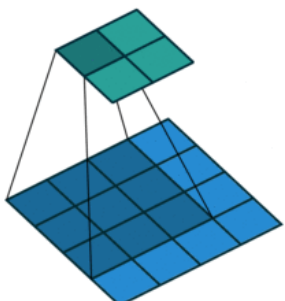
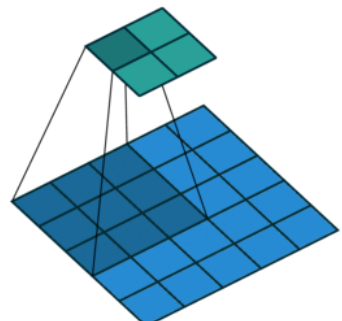


Figure: Goodfellow et al.

Strided convolutions

Sometimes you will see "strided" convolutions. The "stride" is the number of steps to "skip" when applying the filter.

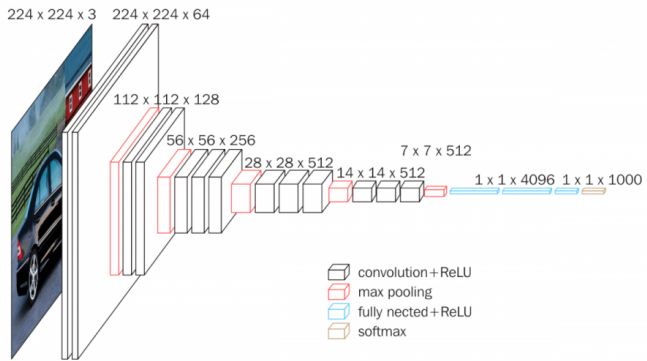




This is an alternative to pooling that also reduces the size of the output.

Images (and many more like them): `https://github.com/vdumoulin/conv_arithmetic`

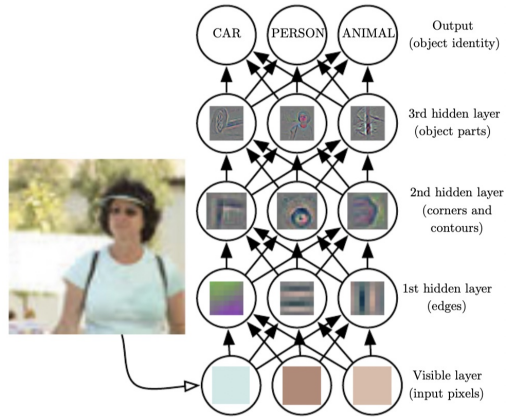
VGGNet (Simonyan and Zisserman, 2014)



Figures: Kaggle

Convnet layers are sort of post-hoc interpretable

- Visible pixels: ???
- First layer: recognizes edges
- Second layer: recognizes contours (collections of edges)
- Third layer: recognizes object parts
- Output layer: predicts object class from object parts



Zeiler & Fergus, 2014

What if we want even deeper networks?

One trick is to add **residual connections** or **skip connections**.

For example, if previously you were considering

$$\mathbf{h}_1 = NN(\mathbf{h}_2),$$

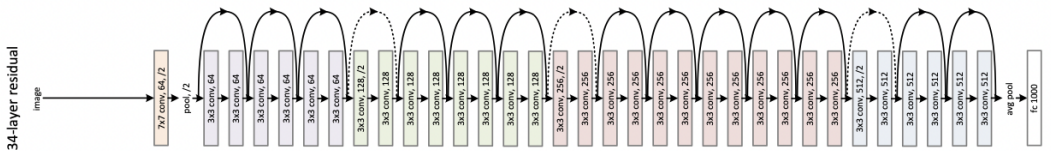
for some architecture NN, and two hidden layers $\mathbf{h}_1, \mathbf{h}_2$, consider instead

$$\mathbf{h}_1 = NN(\mathbf{h}_2) + \mathbf{A}\mathbf{h}_2$$

where we could even enforce $\mathbf{A} = \mathbf{I}$ if $\mathbf{h}_1$ and $\mathbf{h}_2$ are the same size.

This can make it easier for gradients to "flow" back to deeper layers. It also can simplify the learning problem, sometimes.

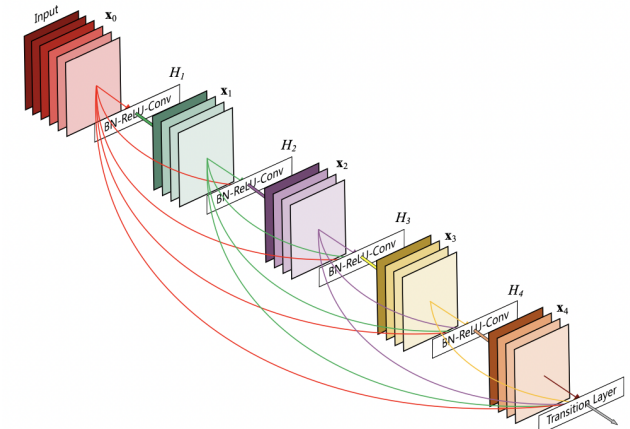
ResNet (He et al., 2015)



When computing gradients back to the deeper layers, there are multiple paths.

A 151-layer-deep ResNet has **less than half** the number of parameters as the 19-layer VGGNet.

Other sorts of skip-connections



Densely-connected convolutional networks ("DenseNet")

Figure: Huang et al., 2018

BatchNorm

Another commonly-used building block in deep convolutional networks is "BatchNorm", or batch normalization.

A BatchNorm layer rescales the inputs $\mathbf{x}$ to have unit mean and standard deviation (computed on the current "batch" of M inputs), and then applies a (learned) scaling and shift.

BatchNorm

Another commonly-used building block in deep convolutional networks is "BatchNorm", or batch normalization.

A BatchNorm layer rescales the inputs $\mathbf{x}$ to have unit mean and standard deviation (computed on the current "batch" of M inputs), and then applies a (learned) scaling and shift.

$$\mu_d = \frac{1}{M}\sum_{i=1}^{M} x_d \qquad \sigma_d^2 = \frac{1}{M}\sum_{i=1}^{M}(x_d - \mu_d)^2$$

$$\hat{x}_d = \frac{x_d - \mu_d}{\sqrt{\sigma_d^2 + \epsilon}} \qquad y_d = \gamma_d \hat{x}_d + \beta_d$$

BatchNorm

Another commonly-used building block in deep convolutional networks is "BatchNorm", or batch normalization.

A BatchNorm layer rescales the inputs $\mathbf{x}$ to have unit mean and standard deviation (computed on the current "batch" of M inputs), and then applies a (learned) scaling and shift.

$$\mu_d = \frac{1}{M}\sum_{i=1}^{M} x_d \qquad \sigma_d^2 = \frac{1}{M}\sum_{i=1}^{M}(x_d - \mu_d)^2$$

$$\hat{x}_d = \frac{x_d - \mu_d}{\sqrt{\sigma_d^2 + \epsilon}} \qquad y_d = \gamma_d \hat{x}_d + \beta_d$$

The resulting vector $\mathbf{y} = \mathrm{BatchNorm}(\mathbf{x})$, where γ_d and β_d are trainable parameters. $\epsilon > 0$ is a small constant for numeric stability.

BatchNorm

Another commonly-used building block in deep convolutional networks is "BatchNorm", or batch normalization.

A BatchNorm layer rescales the inputs $\mathbf{x}$ to have unit mean and standard deviation (computed on the current "batch" of M inputs), and then applies a (learned) scaling and shift.

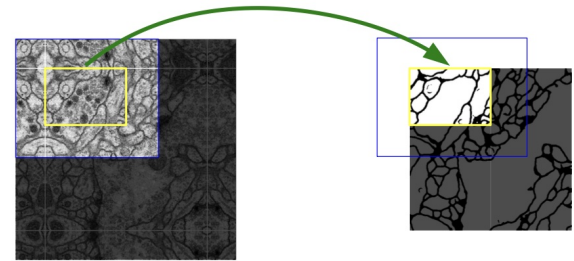
$$\mu_d = \frac{1}{M}\sum_{i=1}^{M} x_d \qquad \sigma_d^2 = \frac{1}{M}\sum_{i=1}^{M}(x_d - \mu_d)^2$$

$$\hat{x}_d = \frac{x_d - \mu_d}{\sqrt{\sigma_d^2 + \epsilon}} \qquad y_d = \gamma_d \hat{x}_d + \beta_d$$

The resulting vector $\mathbf{y} = \mathrm{BatchNorm}(\mathbf{x})$, where γ_d and β_d are trainable parameters. $\epsilon > 0$ is a small constant for numeric stability.

It's a bit unresolved *why* BatchNorm helps, but empirically it improves performance on a variety of tasks and seems to aid in optimization.

Image segmentation



Figures: Ronneberger et al., Jégou et al.

U-Net architecture

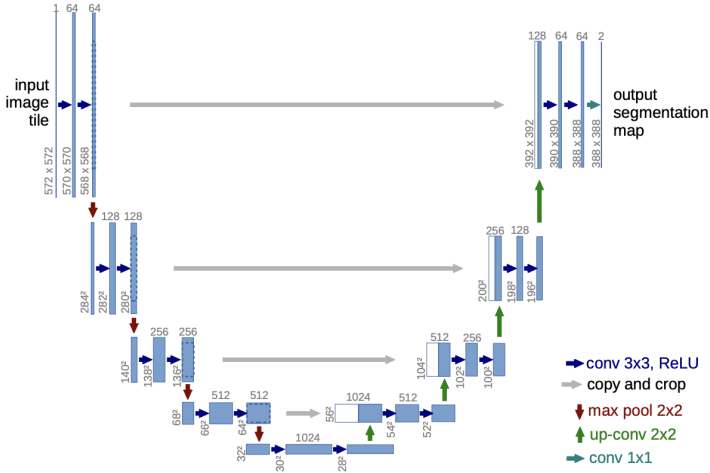


Figure: Ronneberger et al.

Summary

- Convolutions are a natural way of capturing small sorts of invariances in images, and notions of local spatial dependence
- Compared to a fully-connected network, convolutional networks are very parameter-efficient
- Adding "skip connections", of various sorts, help make optimization easier and can make the learning task simpler
- We looked at 2d convolutions, but also could be 1d (for sequences) or 3d (for e.g. video or 3d models)