CSE 152A: Computer Vision

Manmohan Chandraker

Lecture 16: Regularization

Overall goals for the course

- Introduce fundamental concepts in computer vision
- Enable one or all of several such outcomes
  - Pursue higher studies in computer vision
  - Join industry to do cutting-edge work in computer vision
  - Gain appreciation of modern computer vision technologies
- Engage in discussions and interaction
- This is a great time to study computer vision!

Course Details

Course details

- Class webpage:
  - https://cseweb.ucsd.edu/~mkchandraker/classes/CSE152A/Winter2024/
- Instructor email:
  - mkchandraker@ucsd.edu
- Grading
  - 35% final exam
  - 40% homework assignments
  - 20% mid-term
  - 5% self-study exercise
  - Ungraded quizzes
- Aim is to learn together, discuss and have fun!

Course details

- TAs
  - Nicholas Chua: nchua@ucsd.edu
  - Tarun Kalluri: sskallur@ucsd.edu
  - Sreyas Ravichandran: srravichandran@ucsd.edu

- Tutors
  - Kun Wang, Kevin Chan, Zixian Wang: {kuw010, tsc003, ziw081}@ucsd.edu

- Discussion section: M 3-3:50pm

- TA office hours and tutor hours to be posted on webpage

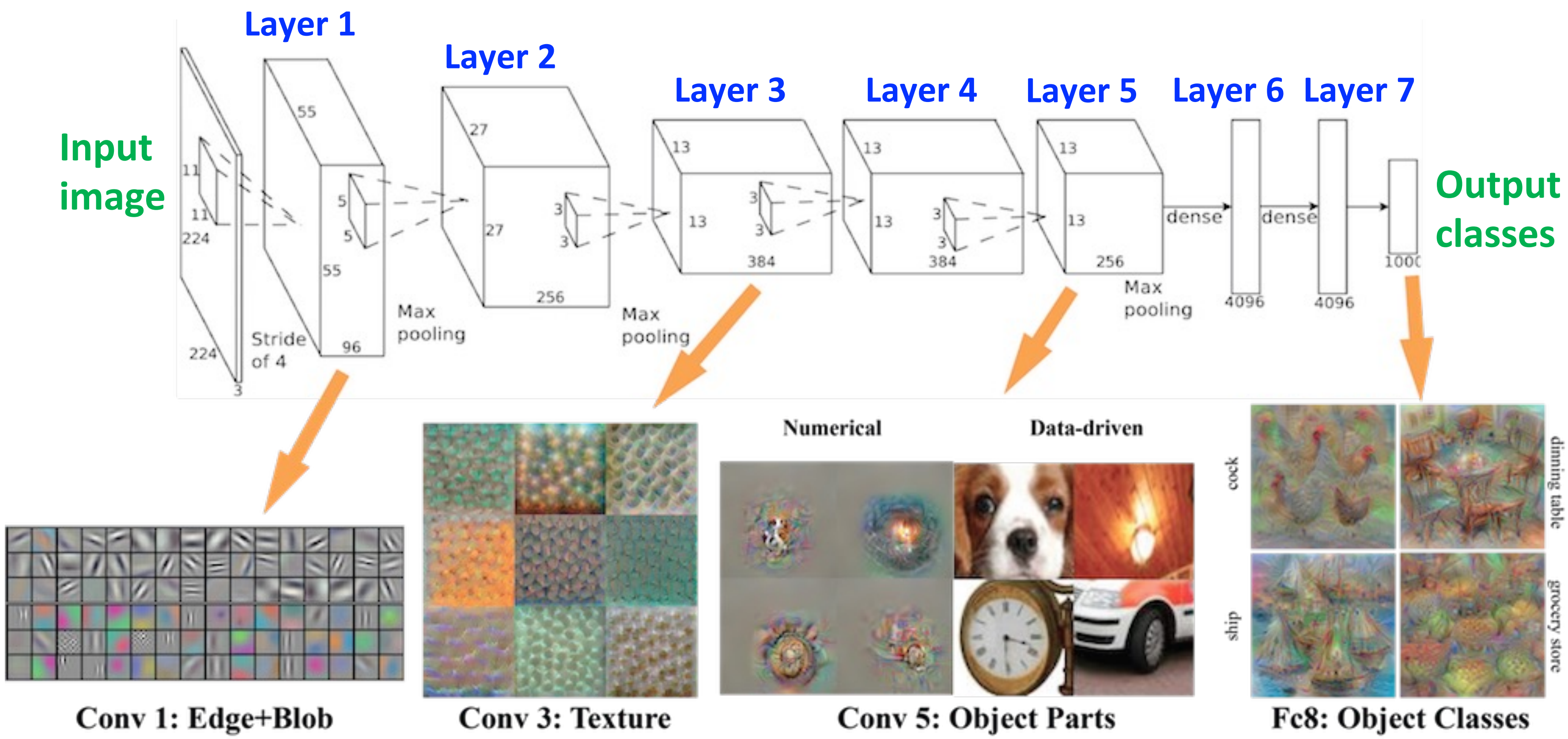
- Piazza for questions and discussions:
  - https://piazza.com/ucsd/winter2024/cse152a

Self-Study Assignment

- We have 39 submissions
- Students and instructors will vote for the top-5 studies by Mar 10
  - Top-5 studies presented in-class by the teams during Mar 15 lecture
- Anonymized submissions are available on Google drive:
  - https://drive.google.com/drive/u/1/folders/104WRTi23jY0nh2mli0PpTaUzrxOcSJno
- Voting form is available here:
  - https://forms.gle/e1jrJfF3dKFYWCdY9
- Criteria you might consider for voting:
  - Depth and correctness of analysis
  - Aesthetics and quality of presentation
  - Whether you find the presentation interesting

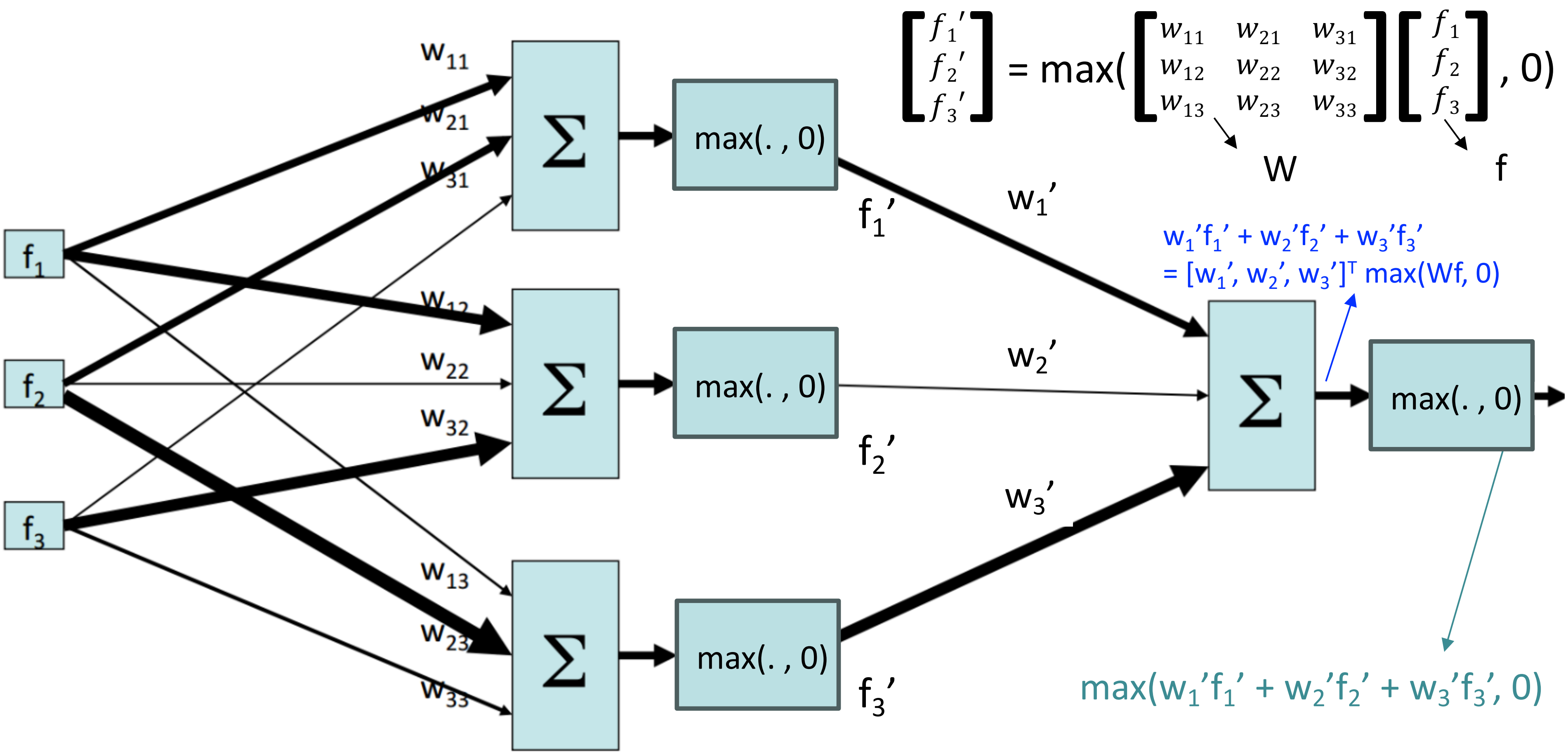
Recap

Learning a Hierarchy of Feature Extractors



- Hierarchical and expressive feature representations
- Trained end-to-end, rather than hand-crafted for each task
- Remarkable in transferring knowledge across tasks

Two-layer perceptron network



Slide credit: Pieter Abeel and Dan Klein

Learning weights: Gradient descent

weights biases

$$C(w, b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

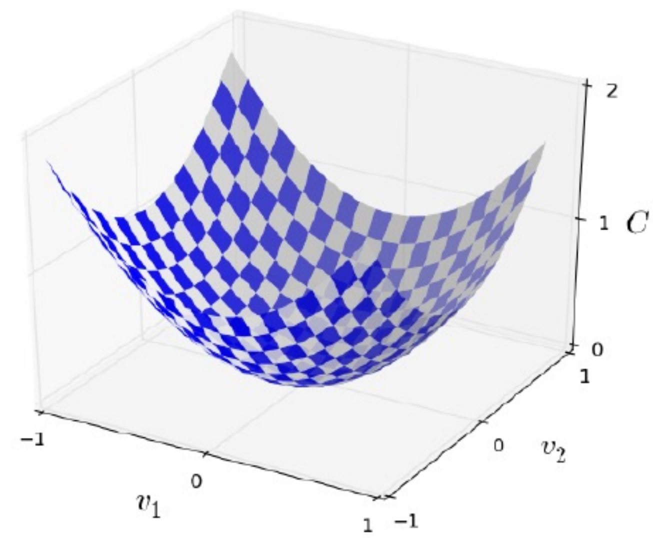
parameters to compute

# of input samples

Update rules for each parameter:

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k}$$

$$b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l}$$



$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2$$

Small changes in parameters to leads to small changes in output

$$\Delta C \approx \nabla C \cdot \Delta v$$

$$\nabla C \equiv \left(\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2} \right)^T$$

Gradient vector!

$$\Delta v = -\eta \nabla C$$

Change the parameter using learning rate (positive) and gradient vector!

$$v \rightarrow v' = v - \eta \nabla C$$

Update rule!

Stochastic gradient descent

Gradient from entire training set:

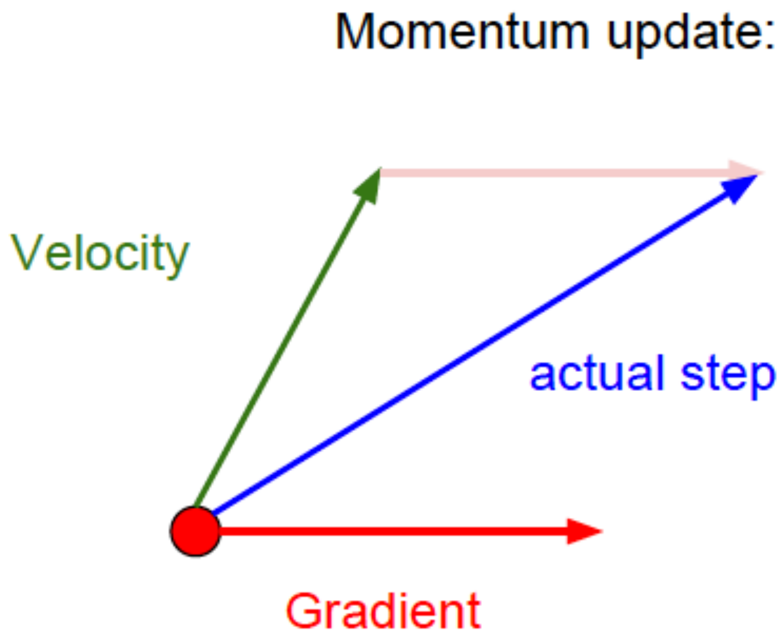
$$\nabla C = \frac{1}{n} \sum_x \nabla C_x$$

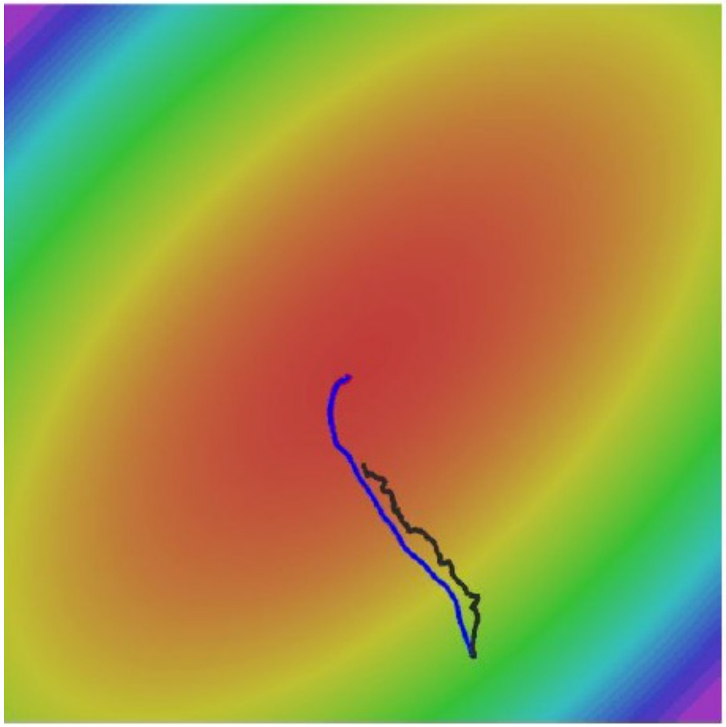
- For large training data, gradient computation takes a long time
  - Leads to “slow learning”

- Instead, consider a mini-batch with *m* samples
- If sample size is large enough, properties approximate the dataset

$$\frac{\sum_{j=1}^{m} \nabla C_{X_j}}{m} \approx \frac{\sum_x \nabla C_x}{n} = \nabla C$$

Overcoming issues: SGD with momentum





SGD

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

SGD+Momentum

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

Build up velocity as a running mean of gradients.

Backpropagation

- In order to differentiate a function $z = f\big(g(x)\big)$ w.r.t x, we can do the following:

Let $y = g(x)$, $\quad z = f(y)$, $\dfrac{dz}{dx} = \dfrac{dz}{dy} \times \dfrac{dy}{dx}$

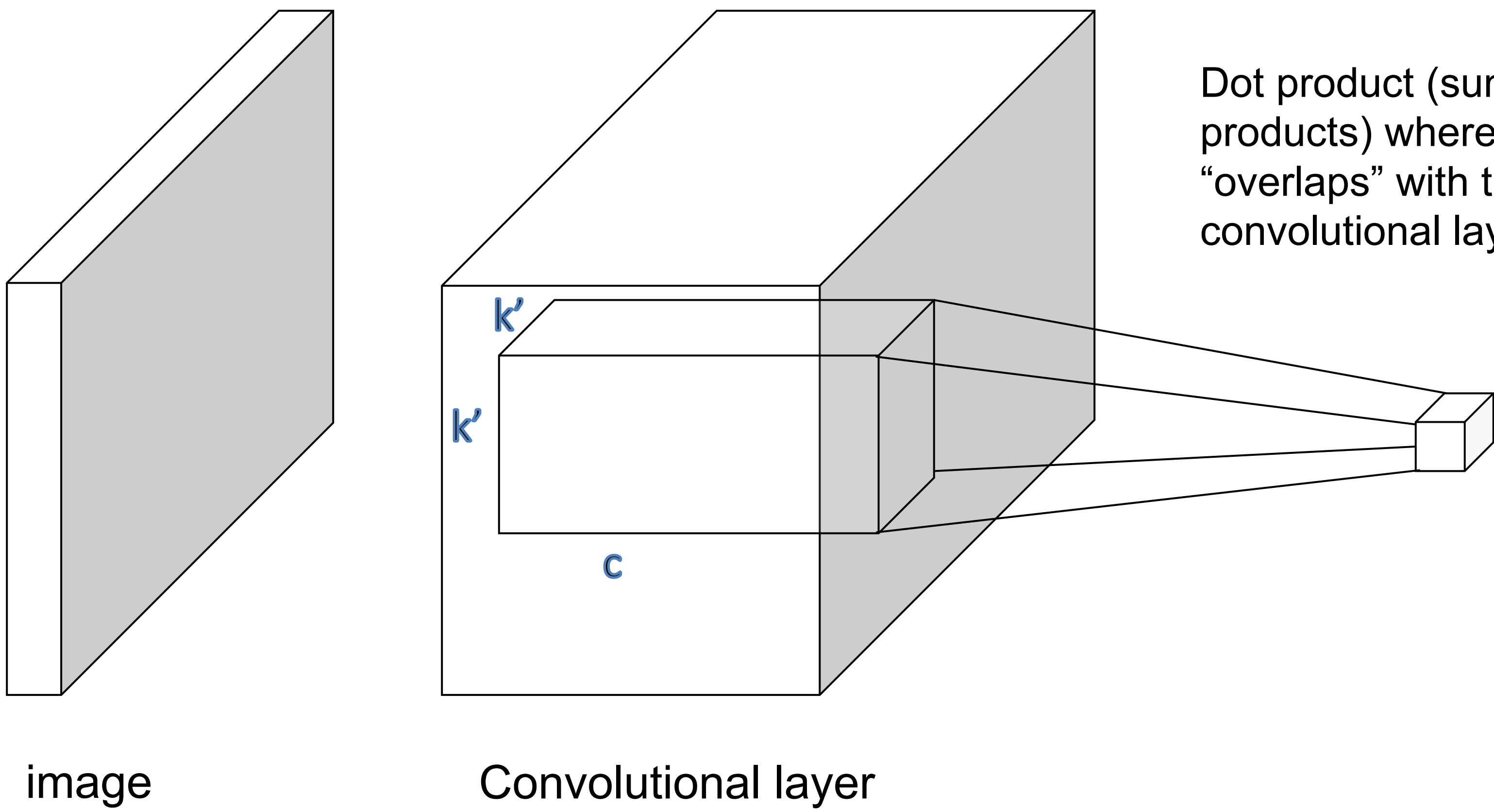
Let $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, g maps from $\mathbb{R}^m$ to $\mathbb{R}^n$, and f maps from $\mathbb{R}^n$ to $\mathbb{R}$. If $y = g(x)$ and $z = f(y)$, then

$$\frac{\partial z}{\partial x_i} = \sum_k \frac{\partial z}{\partial y_k} \frac{\partial y_k}{\partial x_i}$$

This is all you need to know to get the gradients in a neural network!

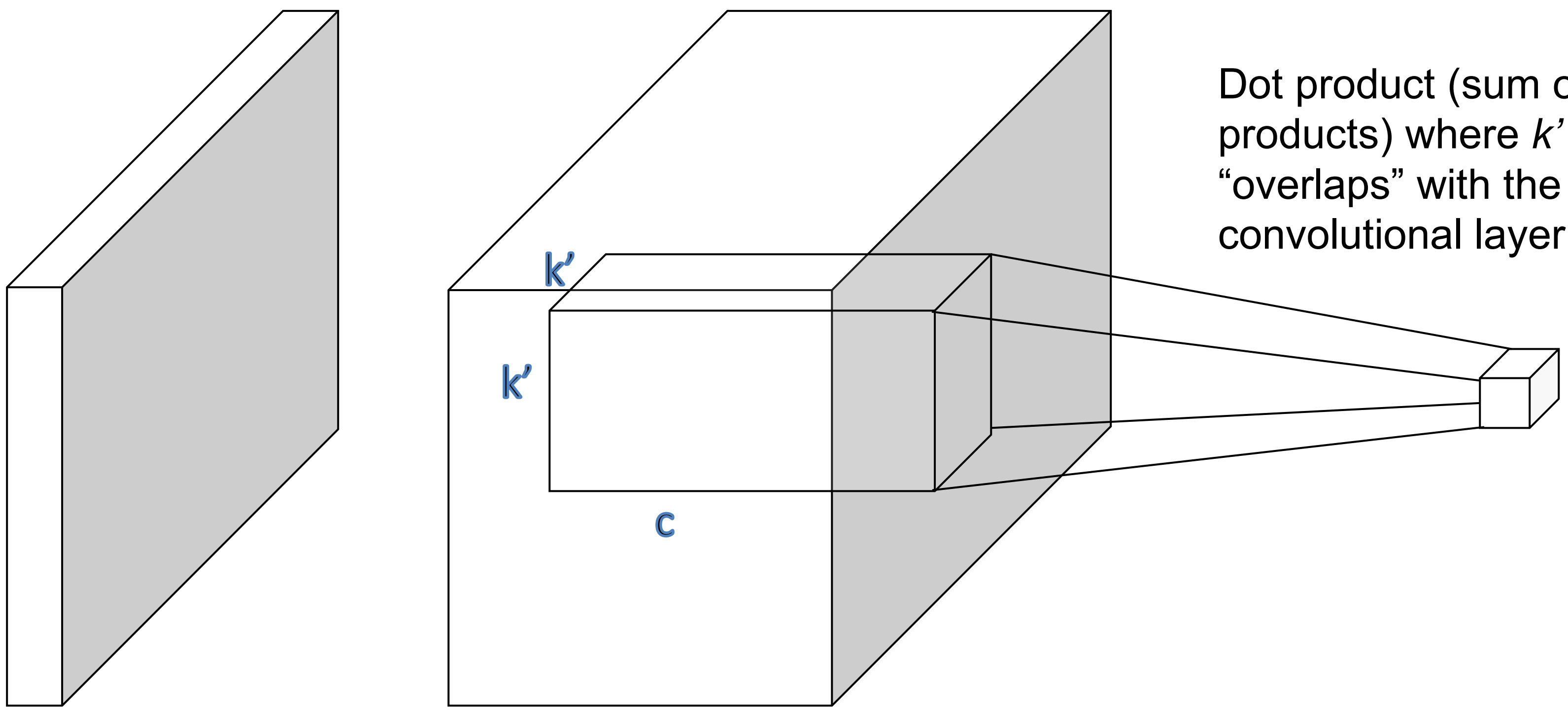
Backpropagation: application of chain rule in a certain order, taking advantage of forward propagation to efficiently compute gradients.

Convolutional Networks



Dot product (sum of elementwise products) where *k'* x *k'* x *c* filter "overlaps" with the 3D convolutional layer

Convolutional Networks

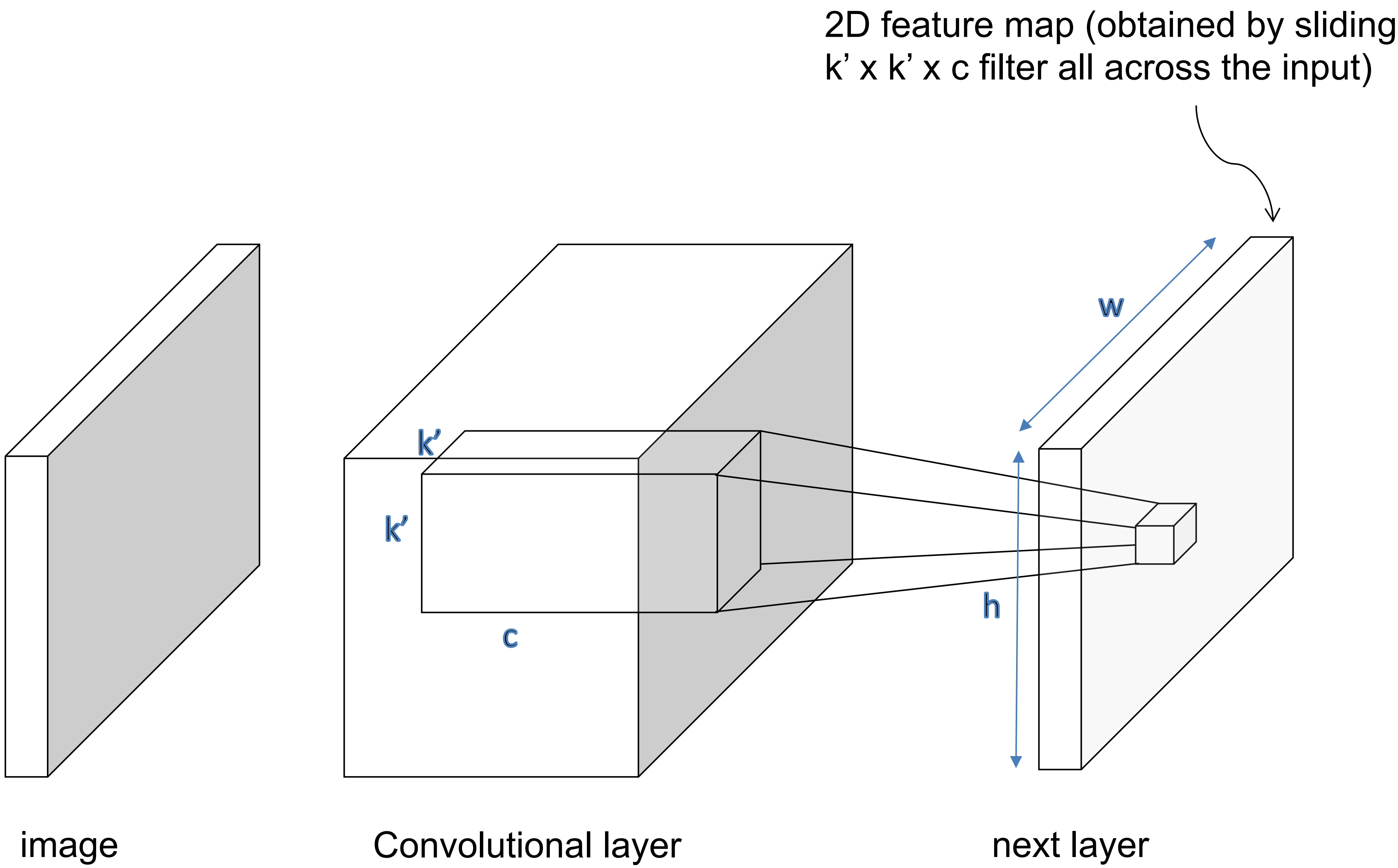


Dot product (sum of elementwise products) where *k'* x *k'* x *c* filter "overlaps" with the 3D convolutional layer

image

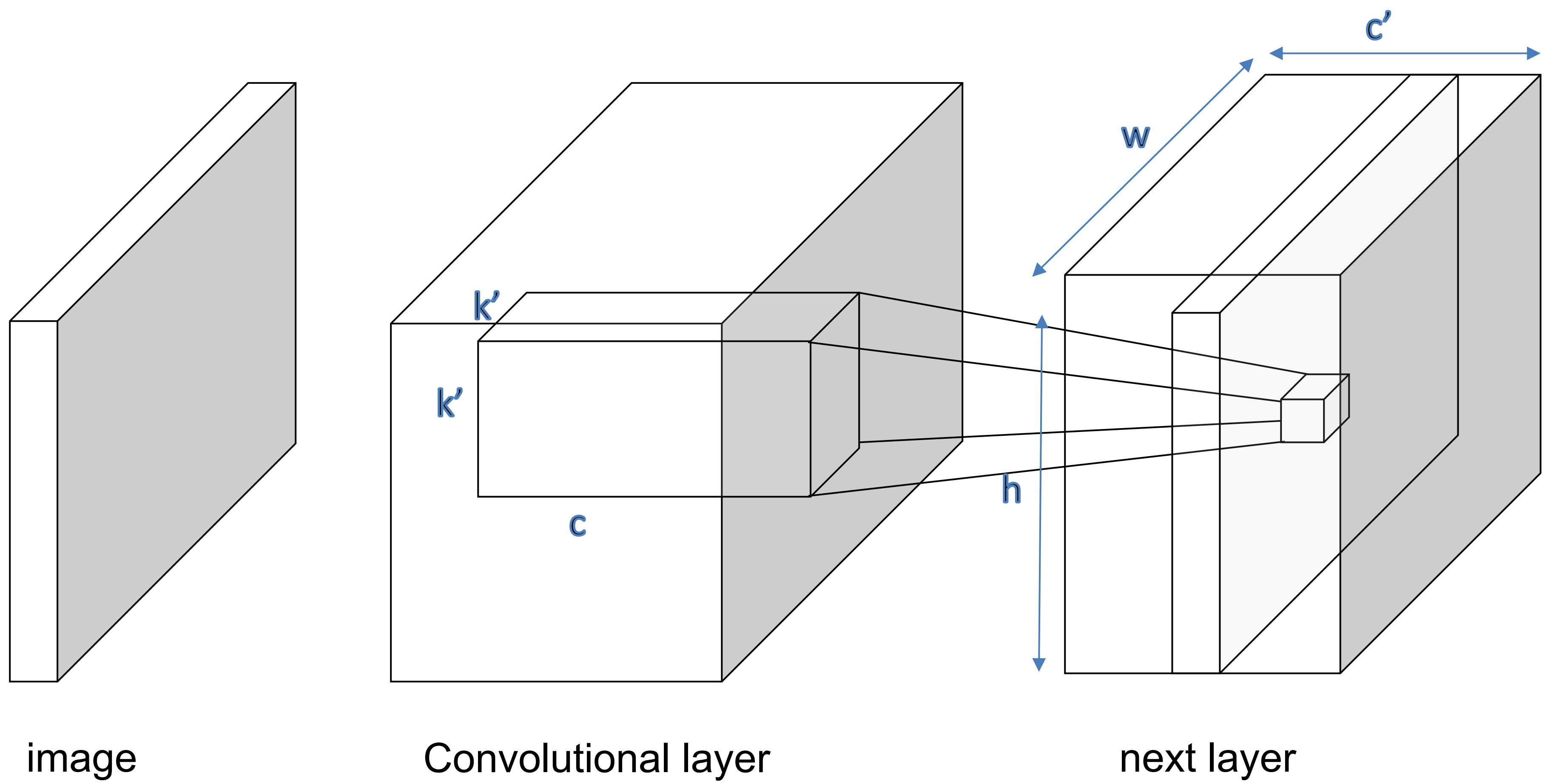
Convolutional layer

Convolutional Networks



Convolutional Networks

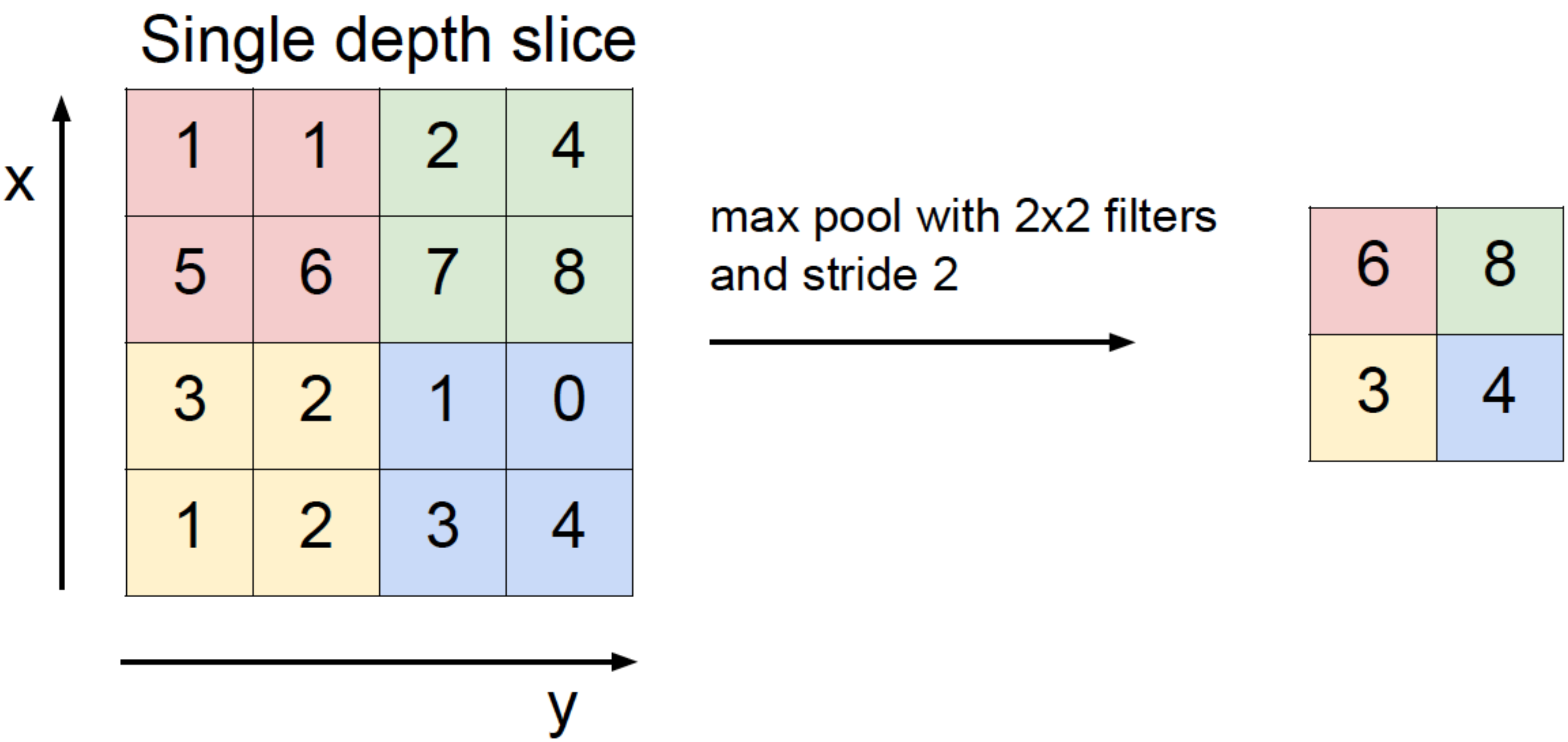
Next convolutional layer has *c'* channels, where each channel is a 2D feature map arising from convolutions of the input 3D volume with a *k'* x *k'* x *c* filter



Slide: Lazebnik

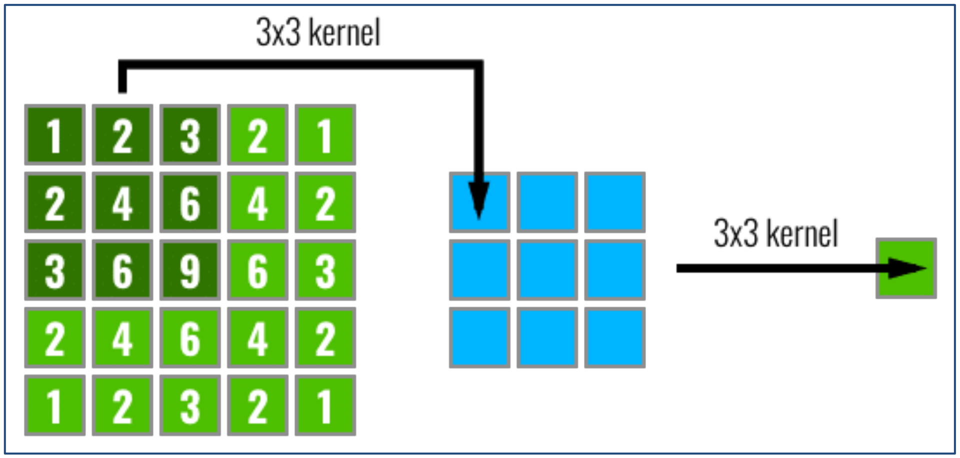
Pooling operations

- Aggregate multiple values into a single value

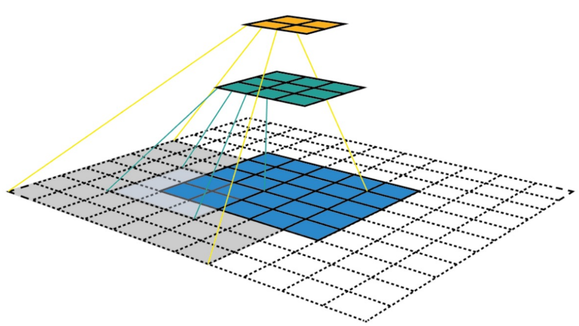


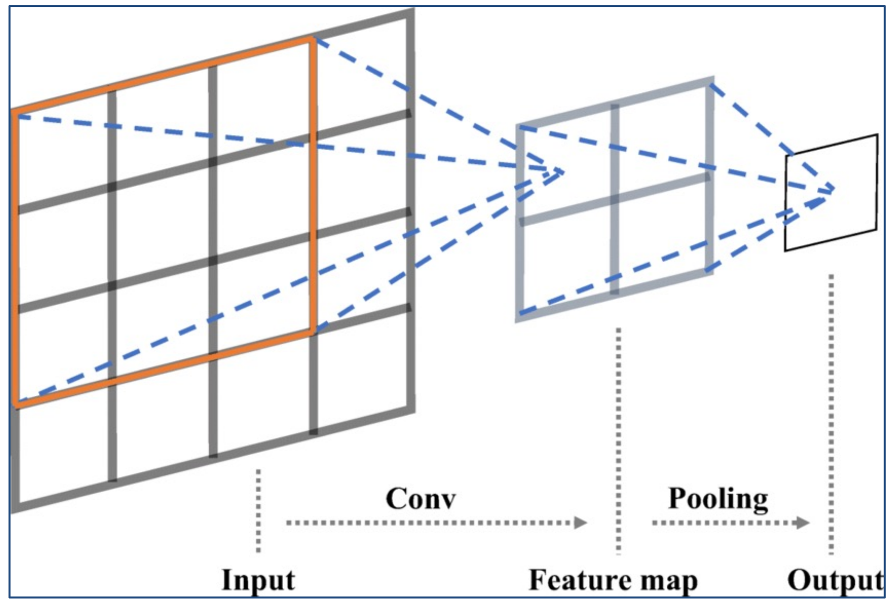
Receptive fields in CNNs

- The area in the input image "seen" by a unit in a CNN
- Units in deeper layers will have wider receptive fields
- Wider receptive fields allow more global reasoning across entire image
- Pooling leads to more rapid widening of receptive fields



[Christian Perone]

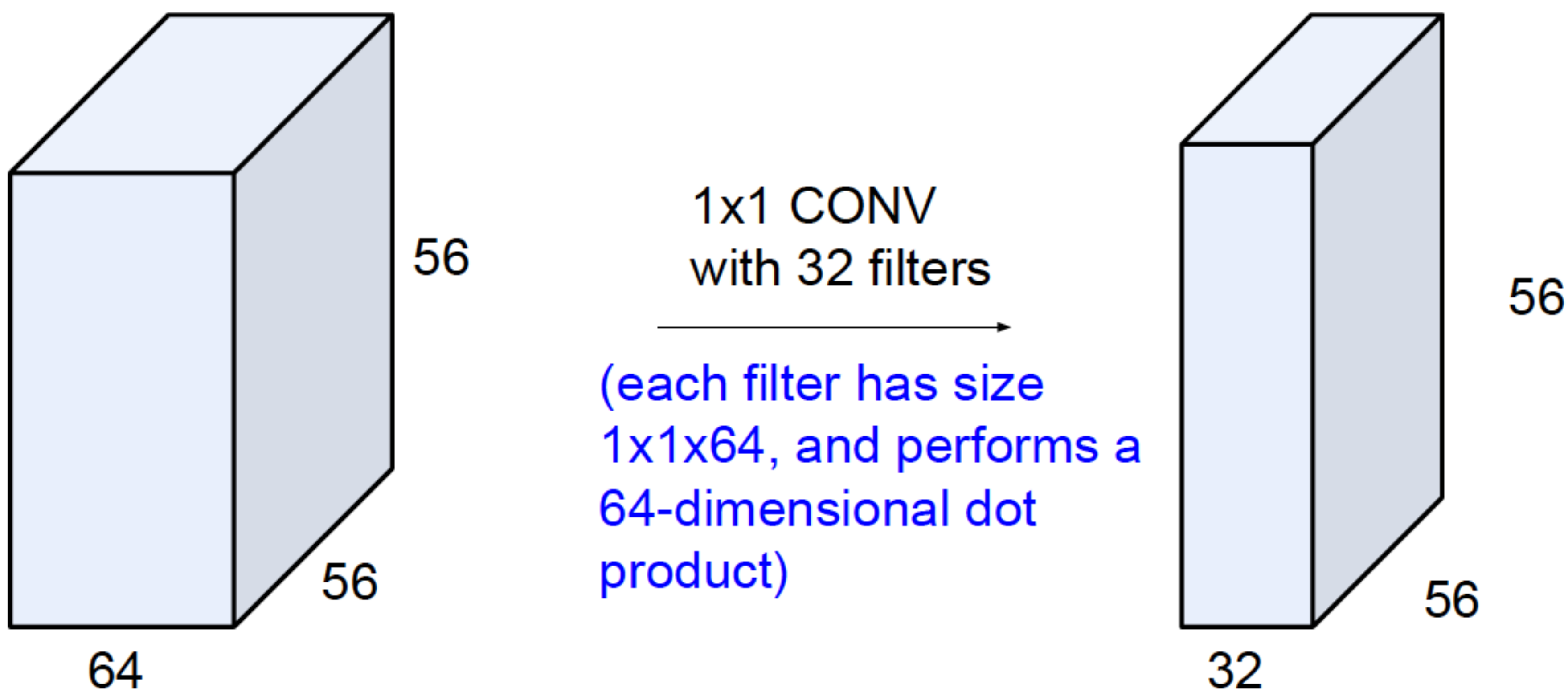




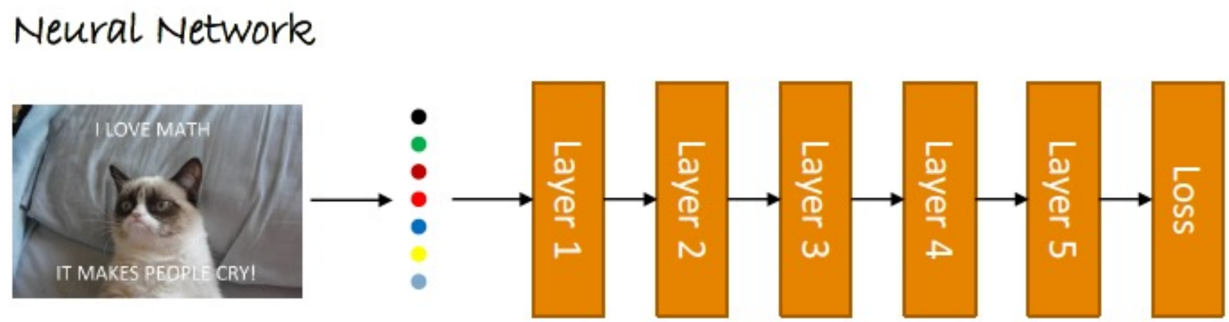
[Tang et al., 2022]

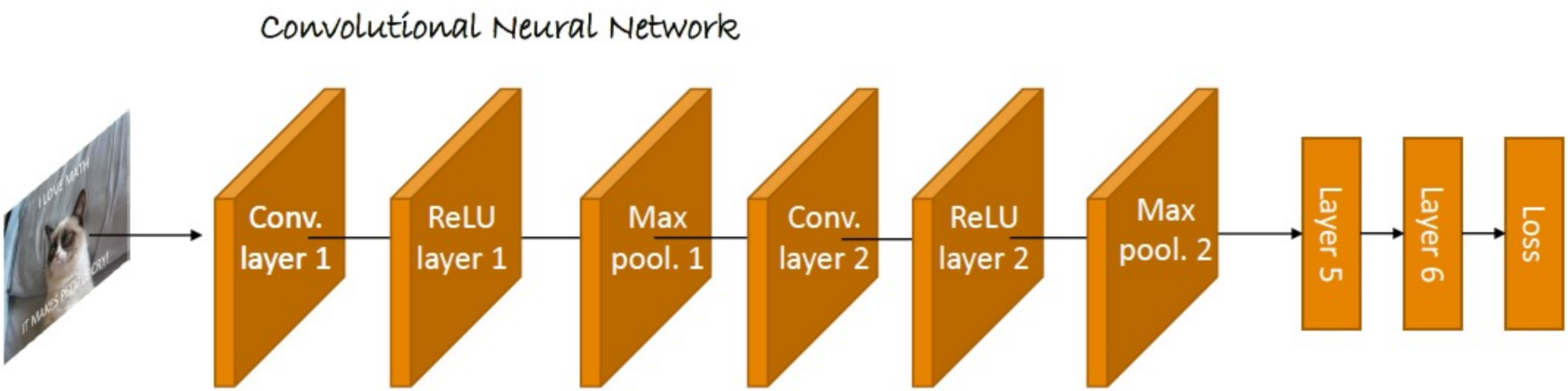
1 x 1 convolutions

1 x 1 convolution layers also possible, equivalent to a dot product.

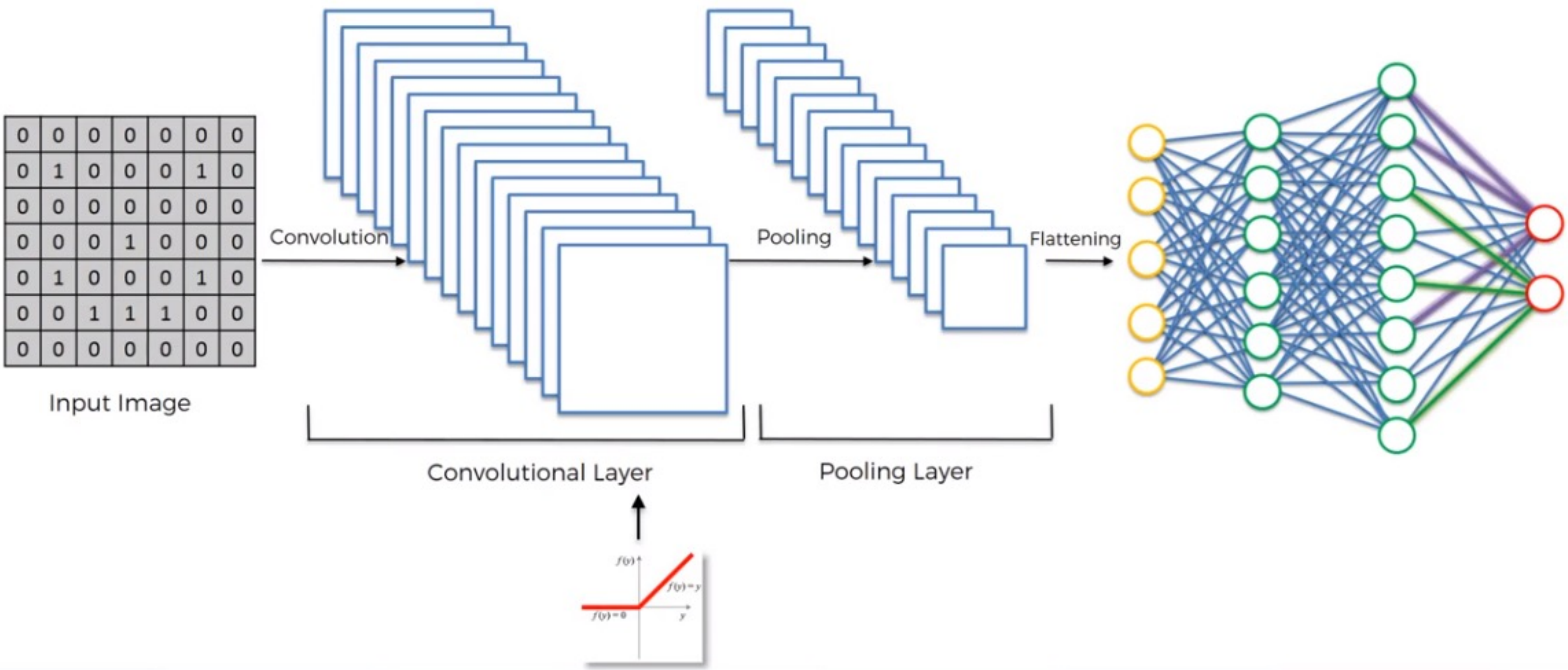


Types of Neural Networks





Setting up the loss function

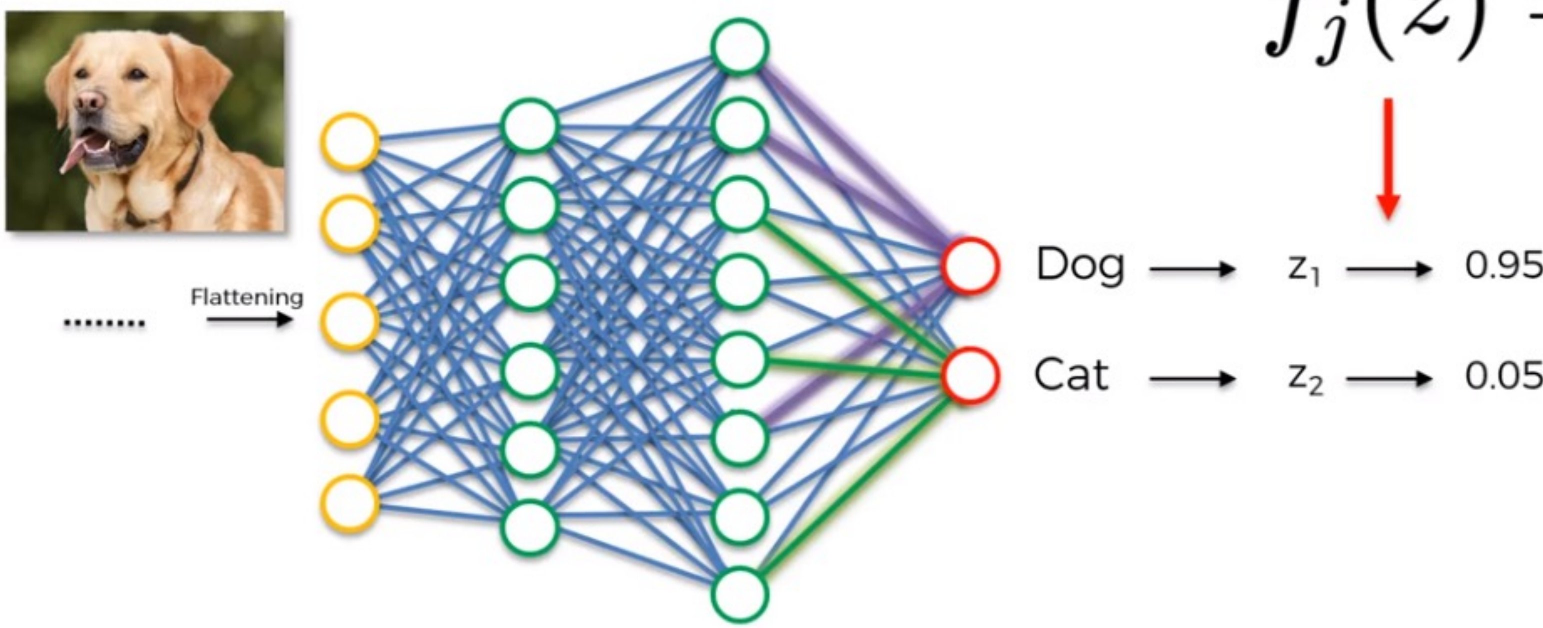


https://www.andreaperlato.com/

Setting up the loss function

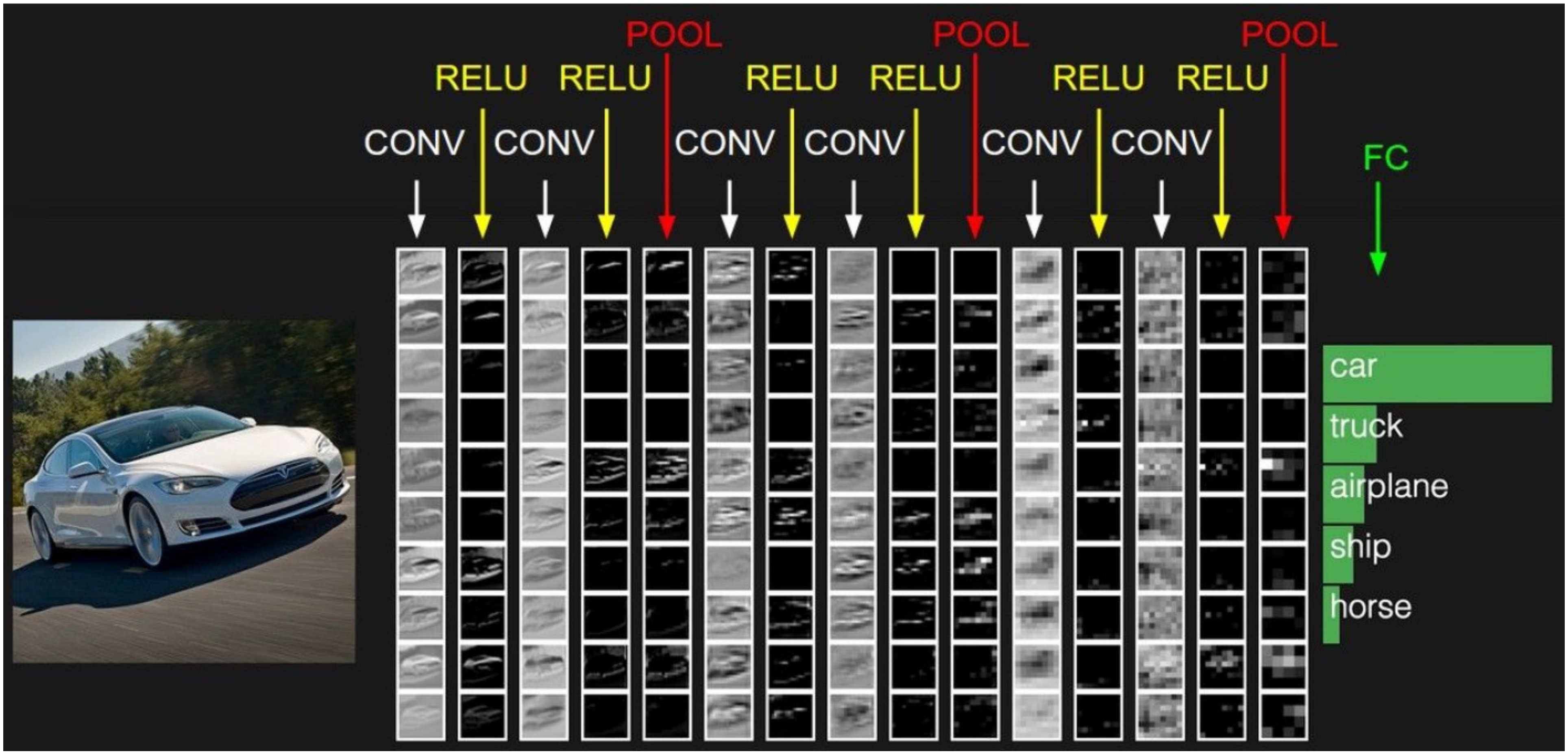
Softmax: maps the network outputs to probabilities

$$f_j(z) = \frac{e^{z_j}}{\sum_k e^{z_k}}$$

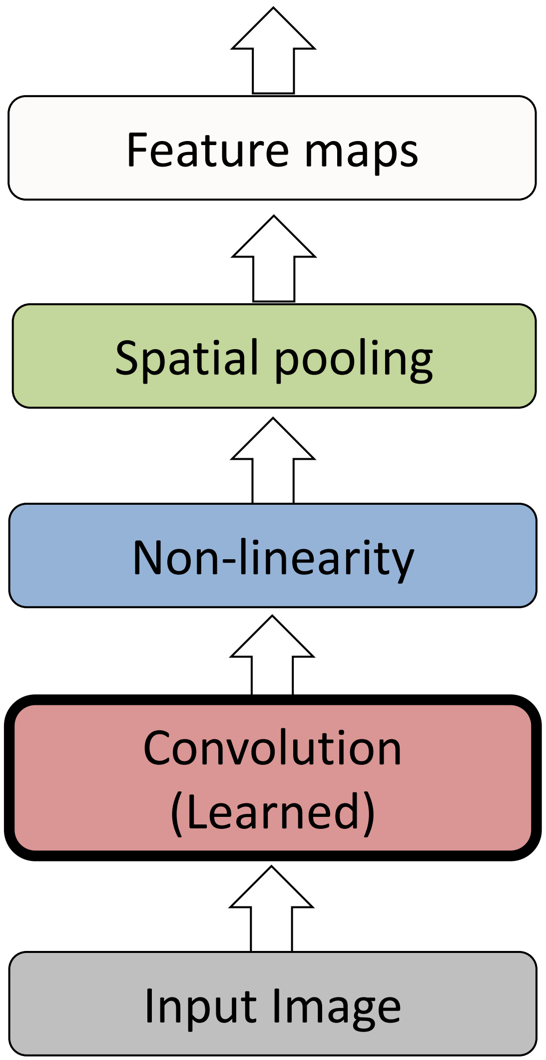


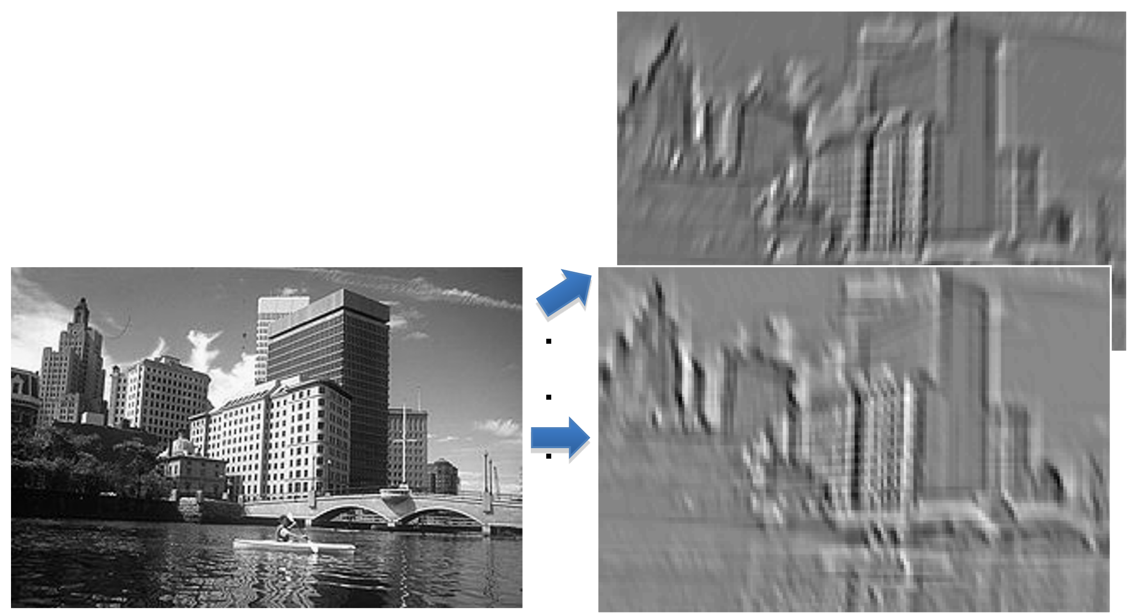
https://www.andreaperlato.com/

Convolutional Neural Networks



Key operations in a CNN





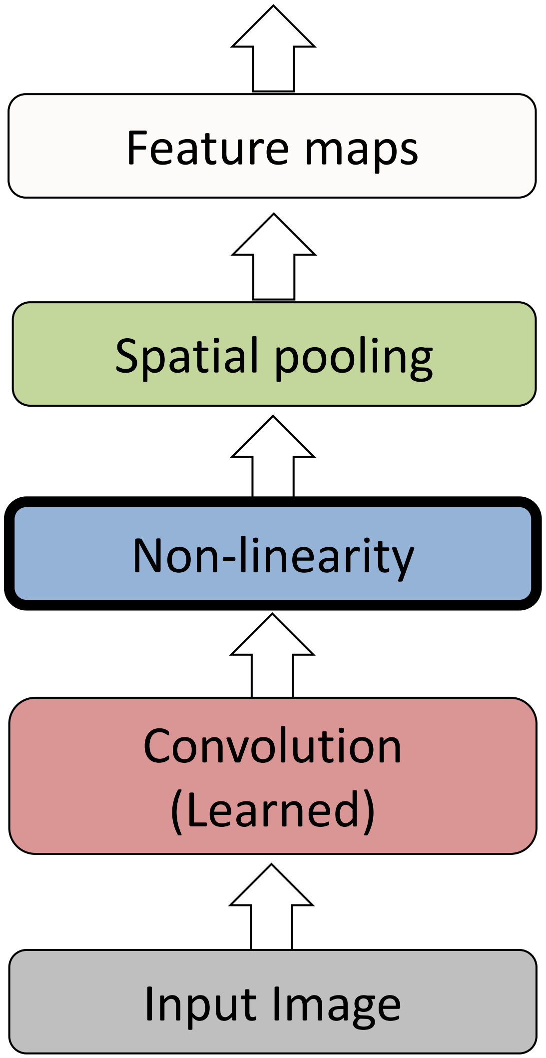
Input

Feature Map

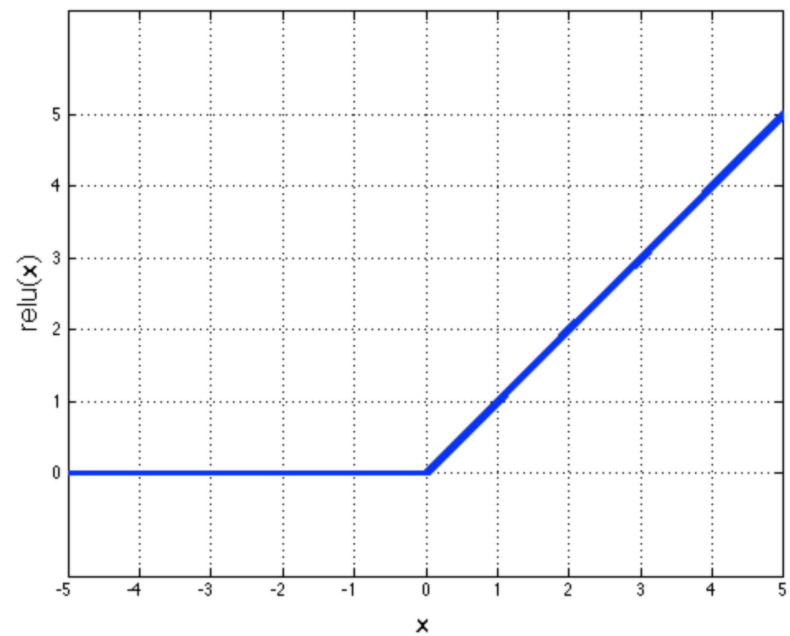
Source: R. Fergus, Y. LeCun

Slide: Lazebnik

Key operations in a CNN



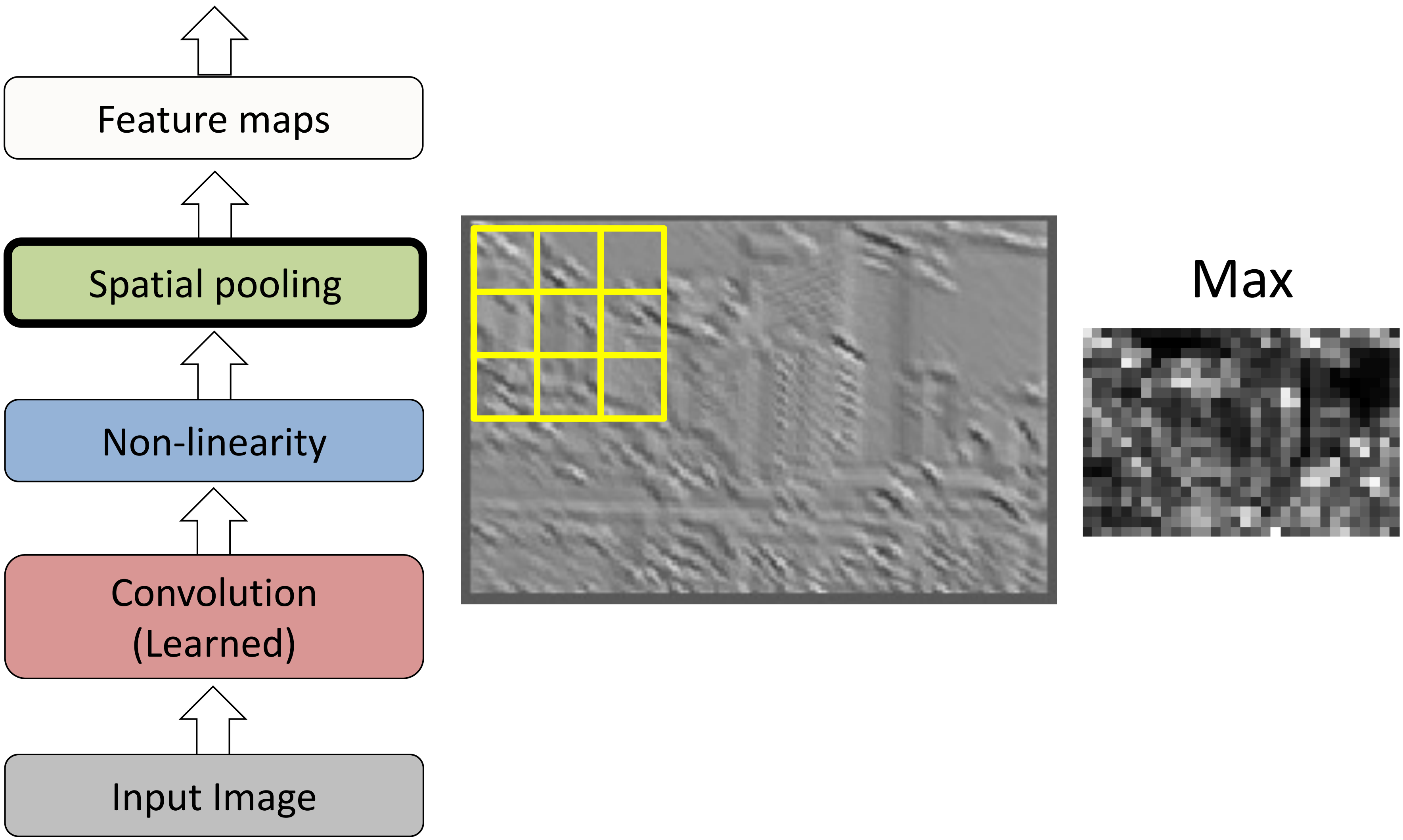
Rectified Linear Unit (ReLU)



Source: R. Fergus, Y. LeCun

Slide: Lazebnik

Key operations in a CNN



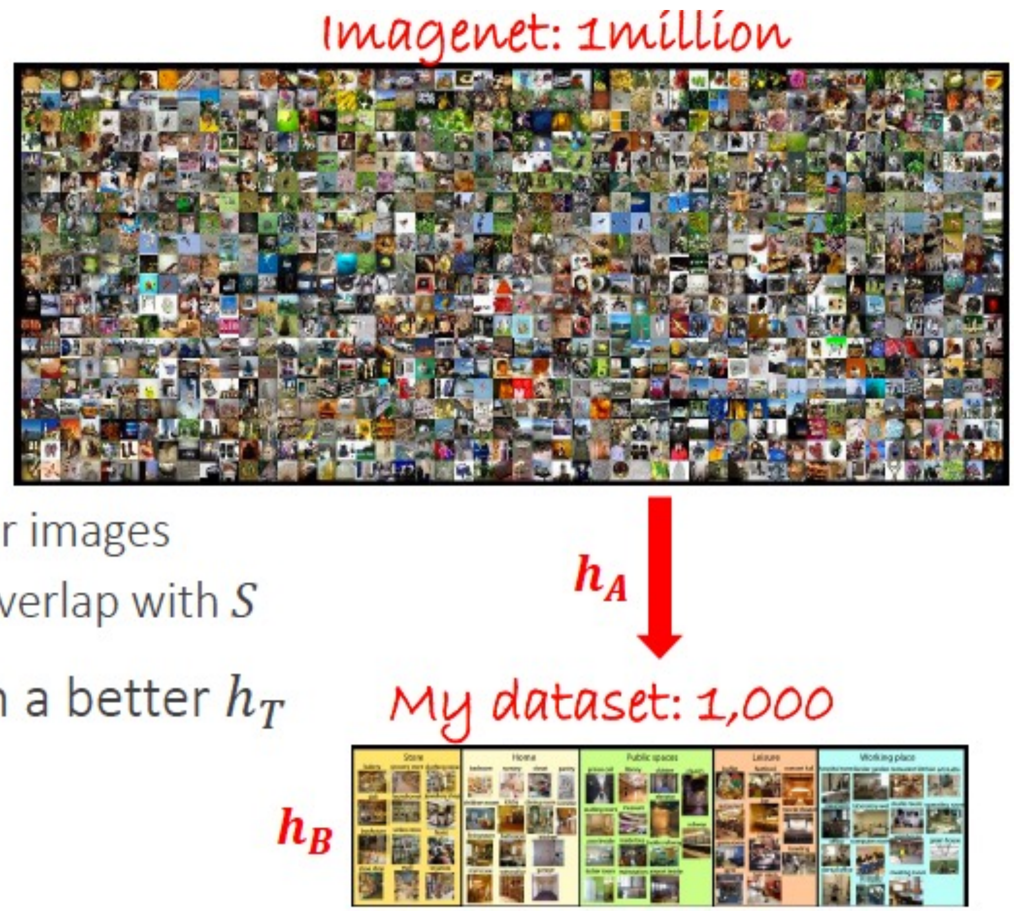
Source: R. Fergus, Y. LeCun

Slide: Lazebnik

Transfer Learning in CNNs

Transfer Learning

- Assume two datasets, T and S
- Dataset S is
  - fully annotated, plenty of images
  - We can build a model h_S
- Dataset T is
  - Not as much annotated, or much fewer images
  - The annotations of T do not need to overlap with S
- We can use the model h_S to learn a better h_T
- This is called transfer learning



Fine-tune h_T using h_S as initialization

- Even if our dataset T is not large, we can train a CNN for it
- Pre-train a network on the dataset S

Fine-tune h_T using h_S as initialization

- Even if our dataset T is not large, we can train a CNN for it
- Pre-train a network on the dataset S
- Assume the parameters of S are already a good start near our final local optimum
- Use them as the initial parameters for our new CNN for the target dataset

$$\theta_{\mathrm{T},\,l}^{(t=0)} = \theta_{\mathrm{S},\,l} \text{ for some layers } l = 1, 2, \ldots$$

Fine-tune h_T using h_S as initialization

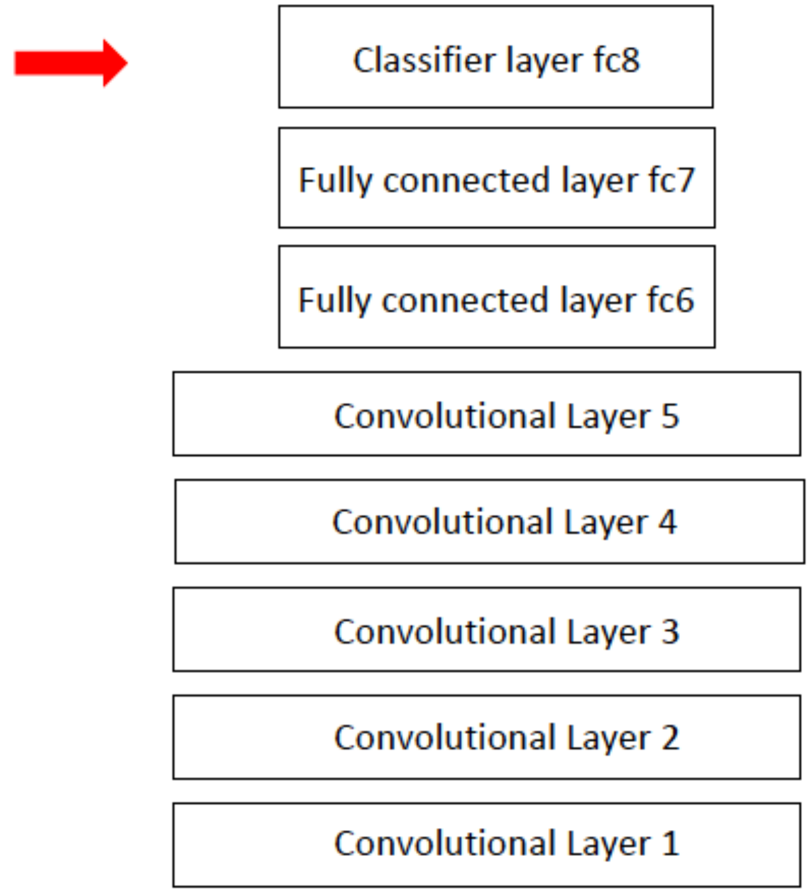
- Even if our dataset T is not large, we can train a CNN for it
- Pre-train a network on the dataset S
- Assume the parameters of S are already a good start near our final local optimum
- Use them as the initial parameters for our new CNN for the target dataset

$$\theta_{\mathrm{T},\, l}^{(t=0)} = \theta_{\mathrm{S},\, l} \text{ for some layers } l = 1, 2, \ldots$$

- This is a good solution when the dataset T is relatively big
  - E.g. for Imagenet S with approximately 1 million images
  - For a dataset T with more than a few thousand images should be ok
- What layers to initialize and how?

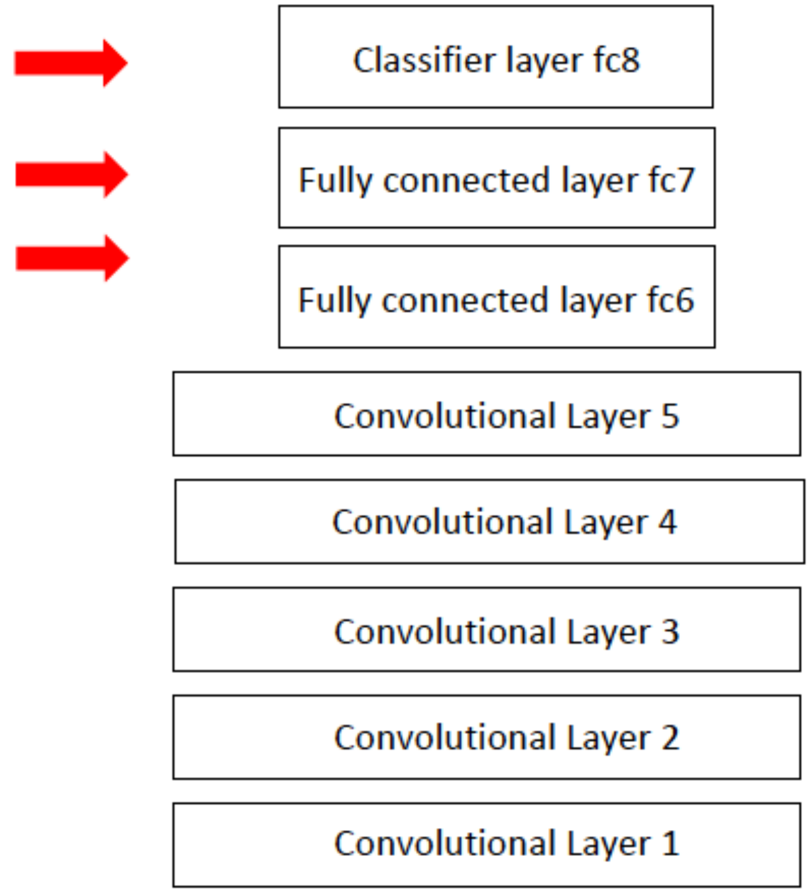
Initializng h_T with h_S

- Classifier layer to loss
  - The loss layer essentially is the “classifier”
  - Same labels → keep the weights from h_S
  - Different labels → delete the layer and start over
  - When too few data, fine-tune only this layer



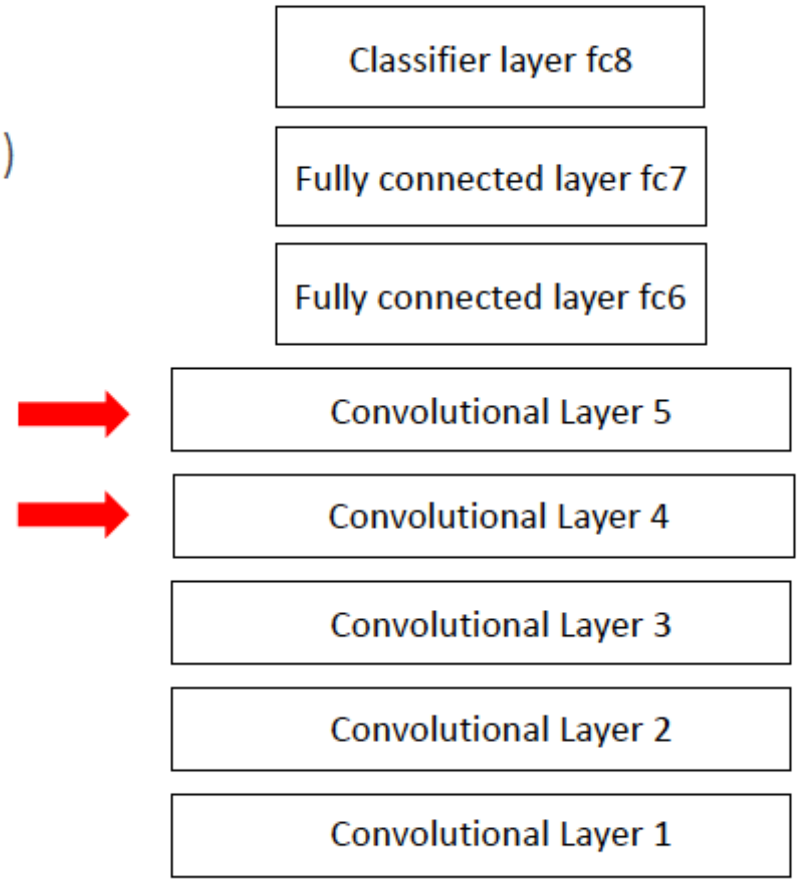
Initializng h_T with h_S

- Classifier layer to loss
  - The loss layer essentially is the "classifier"
  - Same labels → keep the weights from h_S
  - Different labels → delete the layer and start over
  - When too few data, fine-tune only this layer
- Fully connected layers
  - Very important for fine-tuning
  - Sometimes you need to completely delete the last before the classification layer if datasets are very different
  - Capture more semantic, "specific" information
  - Always try first when fine-tuning
  - If you have more data, fine-tune also these layers



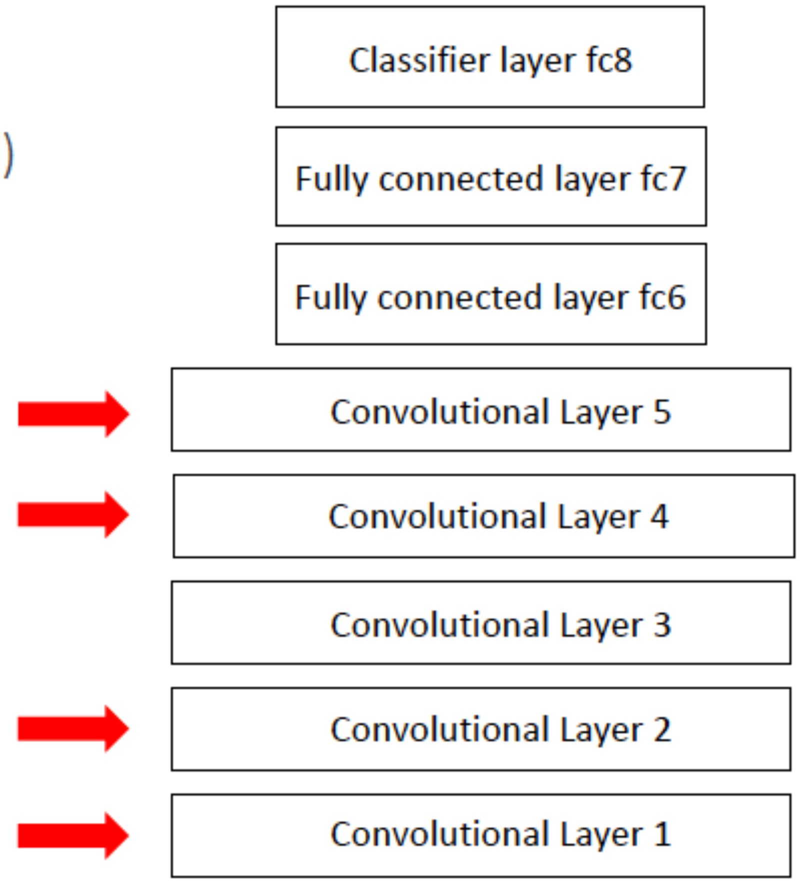
Initializng h_T with h_S

- Upper convolutional layers (conv4, conv5)
  - Mid-level spatial features (face, wheel detectors ...)
  - Can be different from dataset to dataset
  - Capture more generic information
  - Fine-tuning pays off
  - Fine-tune if dataset is big enough



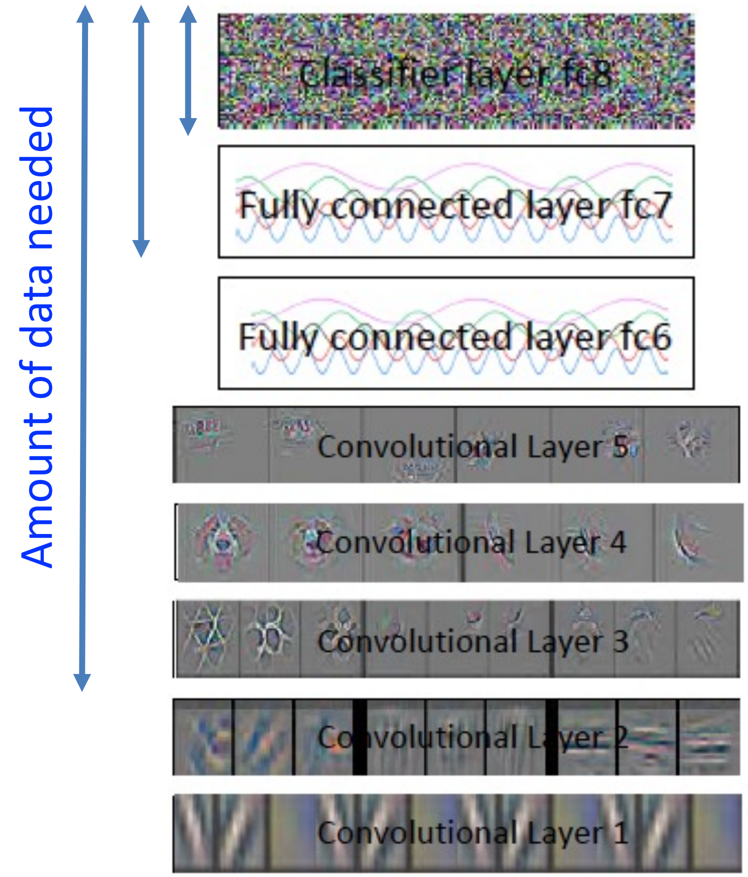
Initializng h_T with h_S

- Upper convolutional layers (conv4, conv5)
  - Mid-level spatial features (face, wheel detectors ...)
  - Can be different from dataset to dataset
  - Capture more generic information
  - Fine-tuning pays off
  - Fine-tune if dataset is big enough
- Lower convolutional layers (conv1, conv2)
  - They capture low level information
  - This information does not change usually
  - Probably, no need to fine-tune but no harm trying

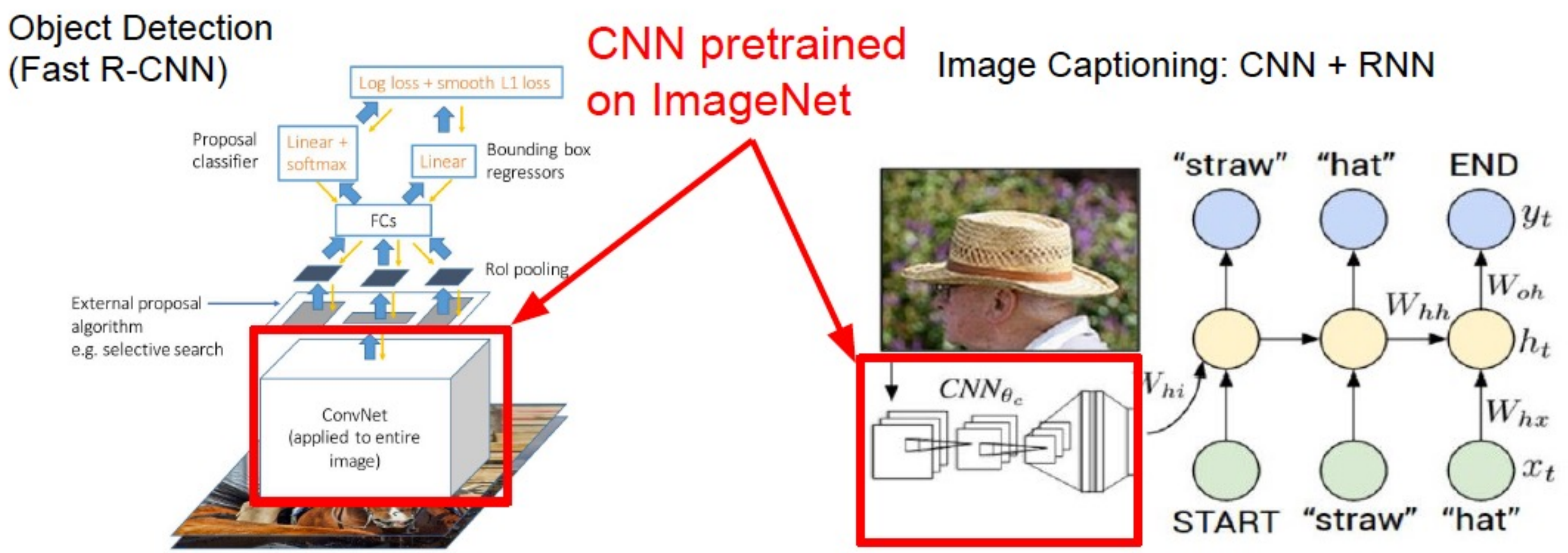


Strategy for fine-tuning

- For layers initialized from h_S use a mild learning rate
  - Remember: your network is already close to a near optimum
  - If too aggressive, learning might diverge
  - A learning rate of 0.001 is a good starting choice (assuming 0.01 was the original learning rate)
- For completely new layers (e.g. loss) use aggressive learning rate
  - If too small, the training will converge very slowly
  - Remember: the rest of the network is near a solution, this layer is very far from one
  - A learning rate of 0.01 is a good starting choice
- If datasets are very similar, fine-tune only fully connected layers
- If datasets are different and you have enough data, fine-tune all layers



Transfer learning is a common choice



Girshick, "Fast R-CNN", ICCV 2015

Karpathy and Fei-Fei, "Deep Visual-Semantic Alignments for Generating Image Descriptions", CVPR 2015

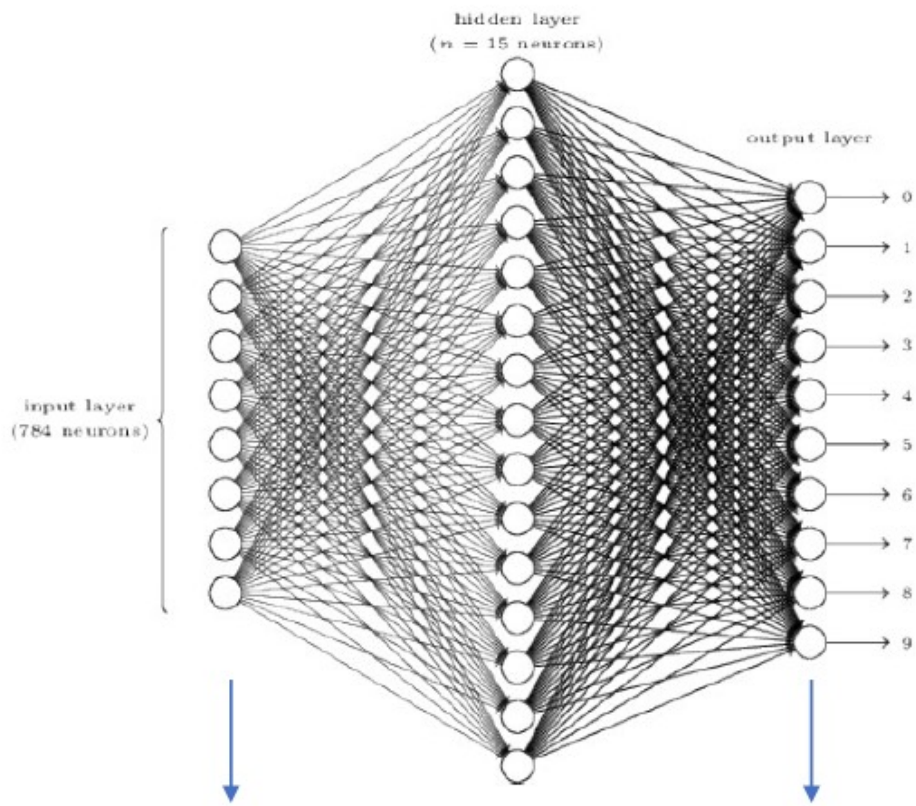
Over-Fitting and Regularization

Cost function for training

$$C(w,b) \equiv \frac{1}{2n} \sum_x \|y(x) - a\|^2$$

parameters to compute

# of input samples



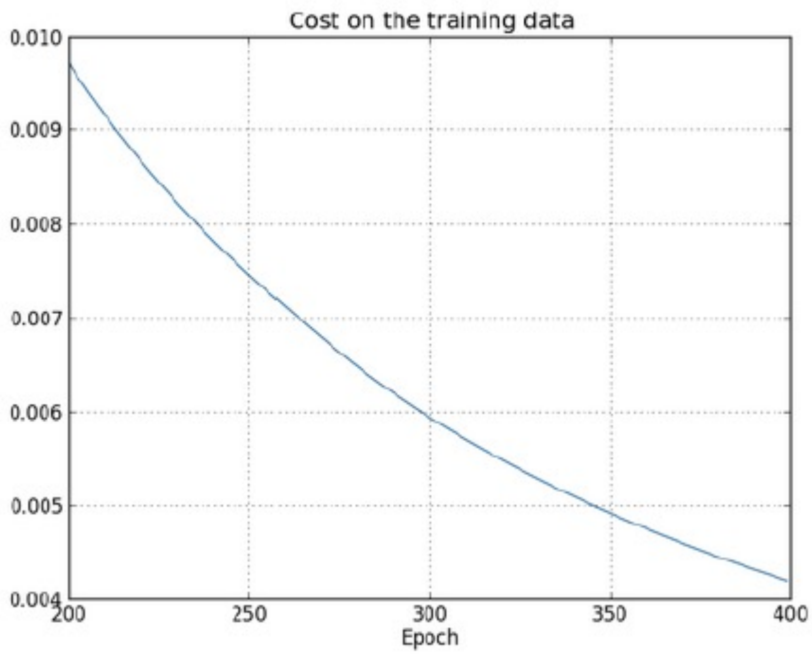
input -> x

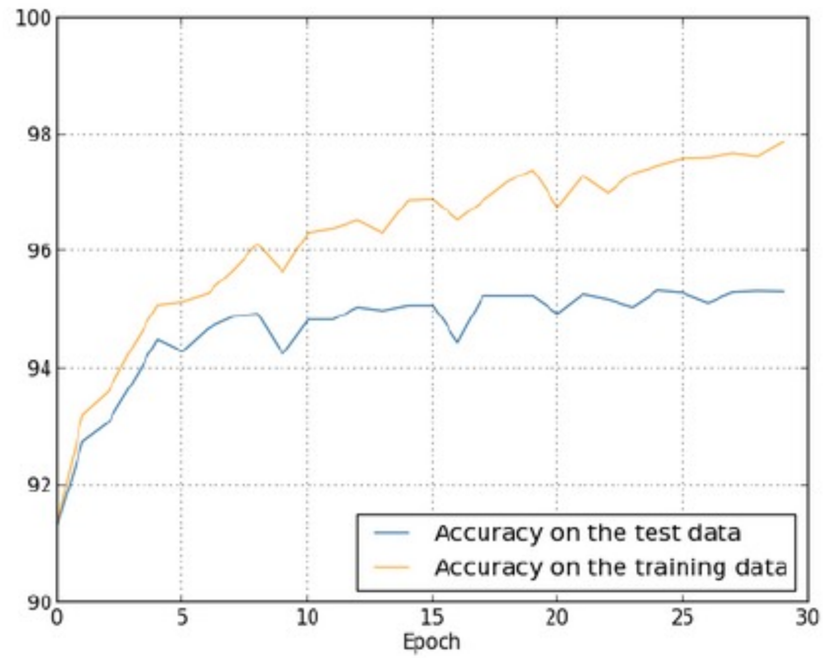
vector output -> a

- The network tries to approximate the function *y(x)* and its output is *a*
- We use a quadratic cost function, or MSE, or “L2-loss”.

Over-fitting

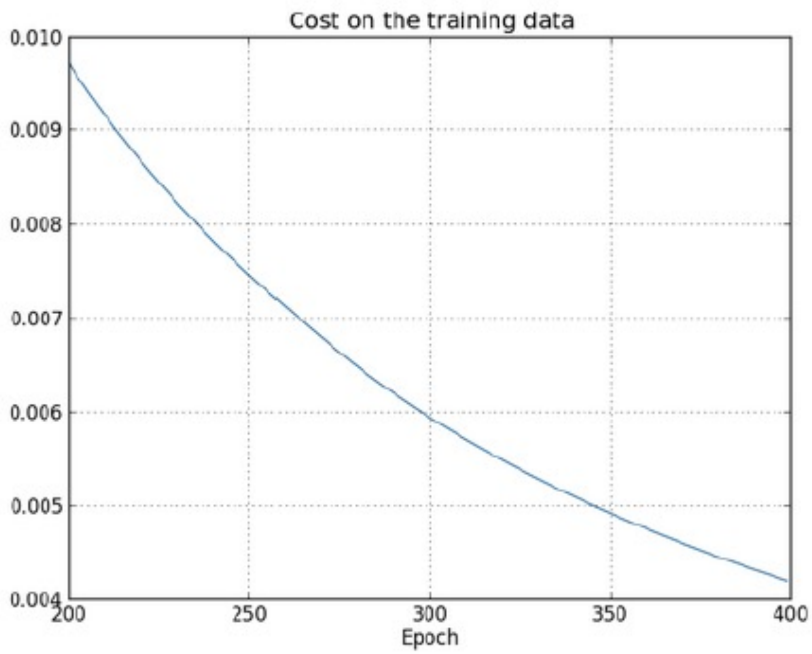
- Instead of 60000 training images, we use only 1000 training images and check the performance on the test data.

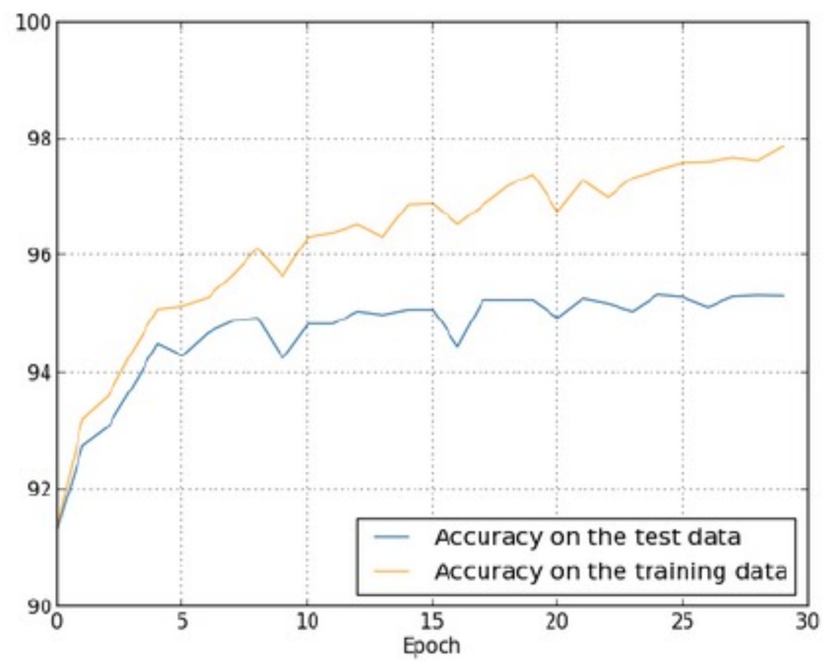




Over-fitting

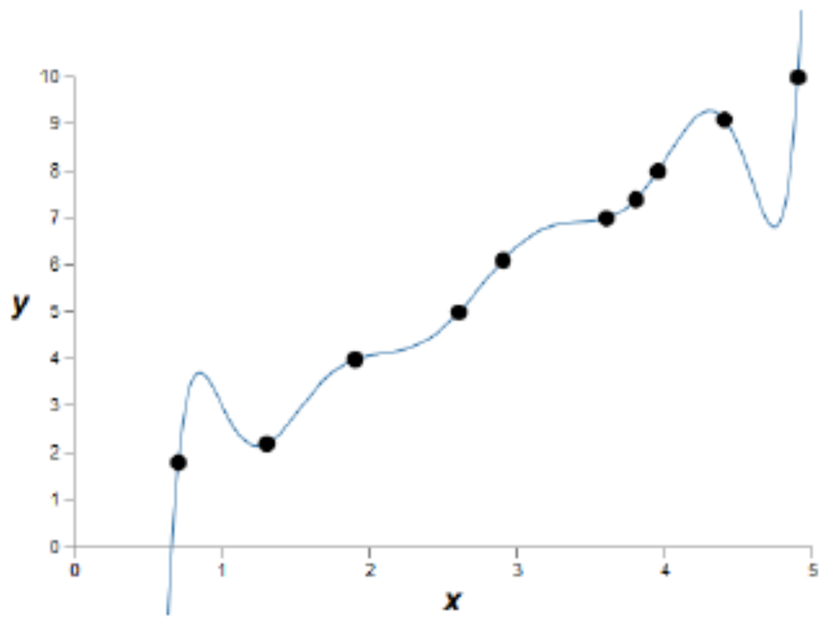
- Instead of 60000 training images, we use only 1000 training images and check the performance on the test data.

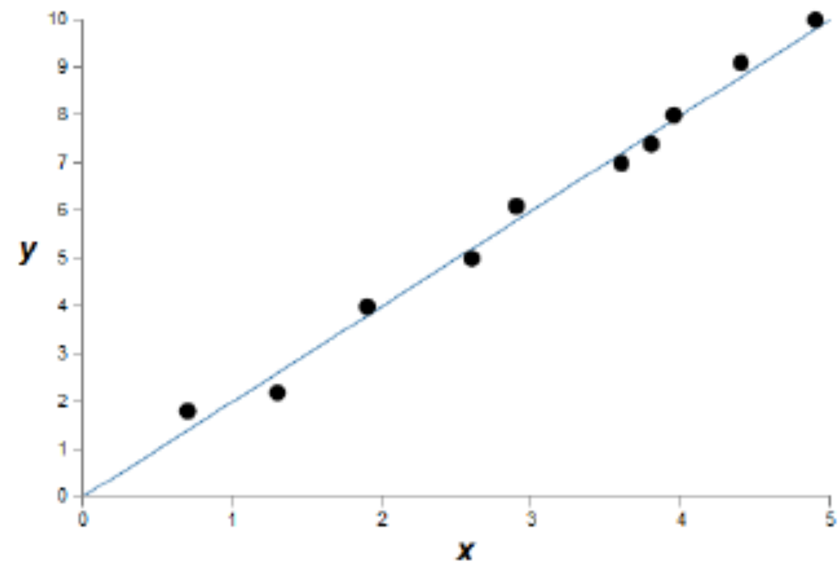




- More data might prevent over-fitting
- But not always feasible to have more data that is relevant.

Regularization reduces over-fitting





- If not enough data, can instead limit model complexity.
- Regularization places constraints on the model, so reduces its complexity.

L2 regularization

L2 regularization:

Let C_0 be original cost. Define $C = C_o + \frac{\lambda}{2n}\sum_w w^2$

- The first term is just the usual $C_o \equiv \frac{1}{2n}\sum_x \|y(x) - a\|^2$
- Here λ is the regularization parameter and n is the size of our training set.

Partial derivatives:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} w$$

Update rule:

$$w \rightarrow w - \frac{\eta \partial C_0}{\partial w} - \frac{\eta \lambda}{n} w$$

L1 regularization

L1 regularization:

$$C = C_o + \frac{\lambda}{n}\sum_{w}|w|$$

The first term is just the usual $C_o \equiv \frac{1}{2n}\sum_x \|y(x) - a\|^2$

Partial derivatives:

$$\frac{\partial C}{\partial w} = \frac{\partial C_0}{\partial w} + \frac{\lambda}{n} sgn(w)$$

Update rule:

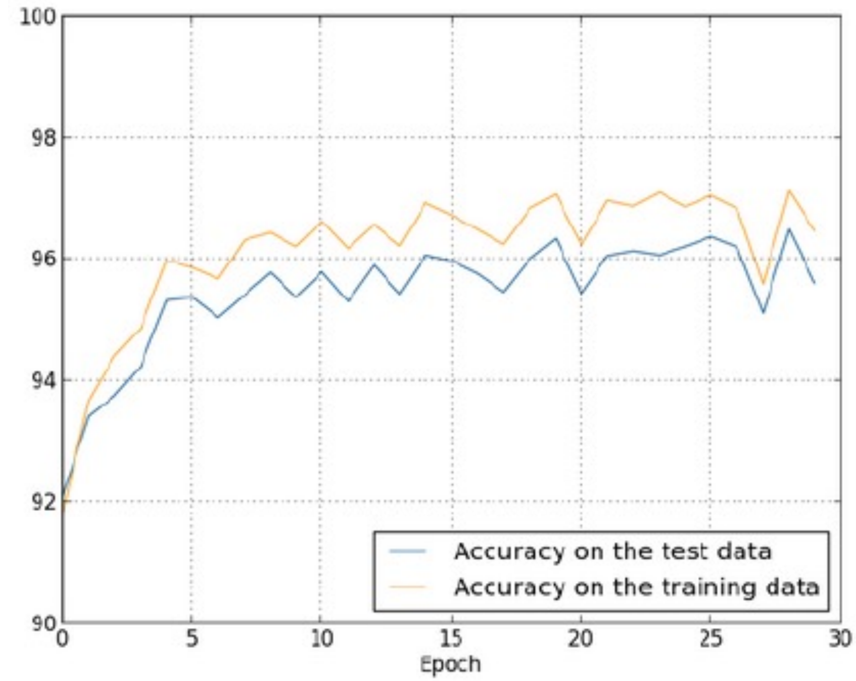
$$w \to w - \frac{\eta \partial C_0}{\partial w} - \frac{\eta\lambda}{n} sgn(w)$$

L2 or L1 regularization

- In L1 case, the weights shrink by a constant amount towards 0.
- In L2 case, the weights shrink by an amount that is proportional to w.
- When the weight has a large magnitude $|w|$, then the L1 regularization shrinks less than the L2.
- When the weight has a small magnitude $|w|$, then the L1 regularization shrinks more than the L2.
- The net result is that the L1 regularization focuses on the weights of a few important connections and the rest are driven to zero.

Regularization reduces over-fitting

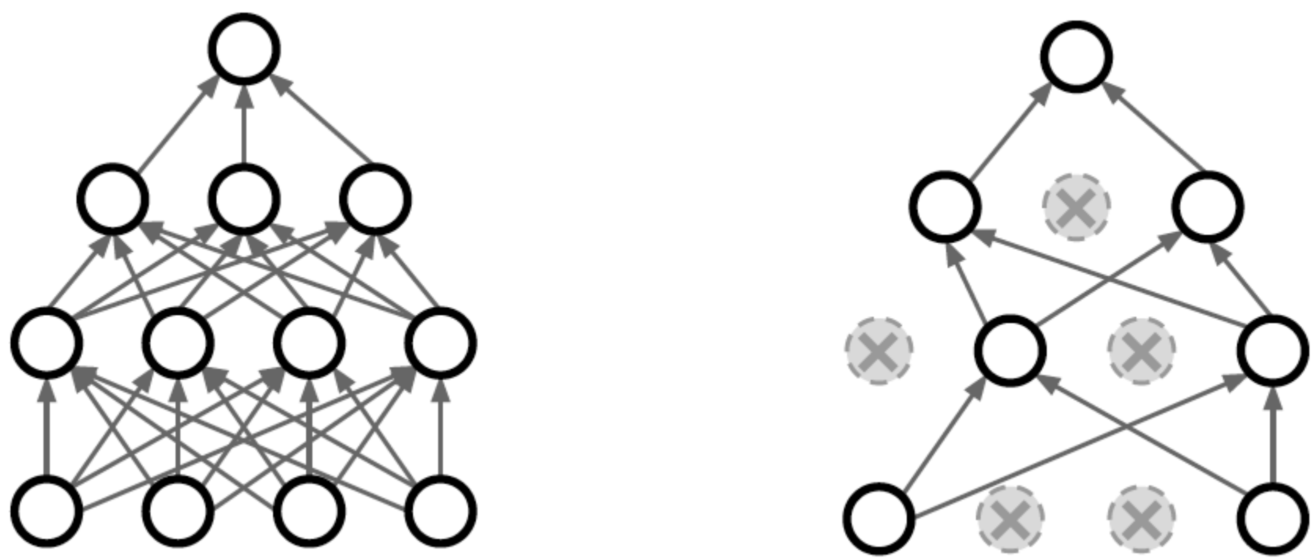
Before



After

Dropout as a regularization

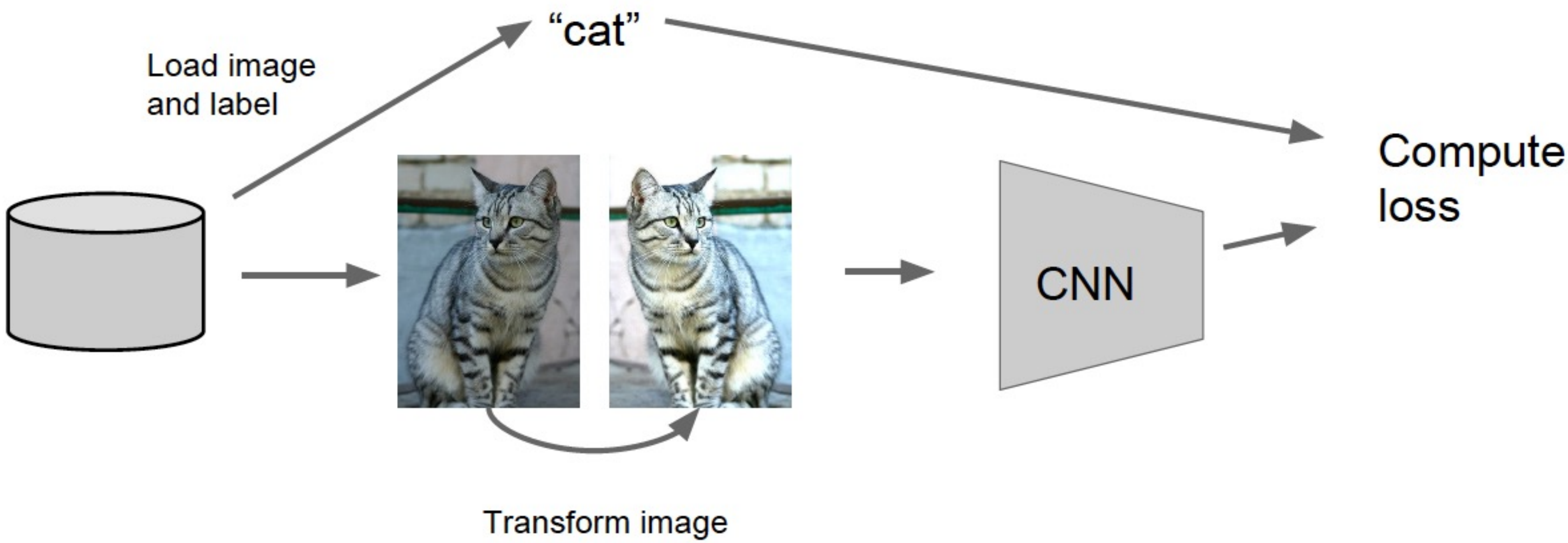
- Modify the network itself
  - Randomly delete half the hidden neurons in the network
  - Repeat several times to learn weights and biases
  - At runtime, twice as many neurons, so halve the weights outgoing from a neuron



Dropout as a regularization

- Averaging or voting scheme to decide output
  - Forces neurons to learn independent of others
  - Same training data, but random initializations
  - Each network over-fits in a different way
  - Robustness: average output not sensitive to particular mode

Data augmentation as regularization



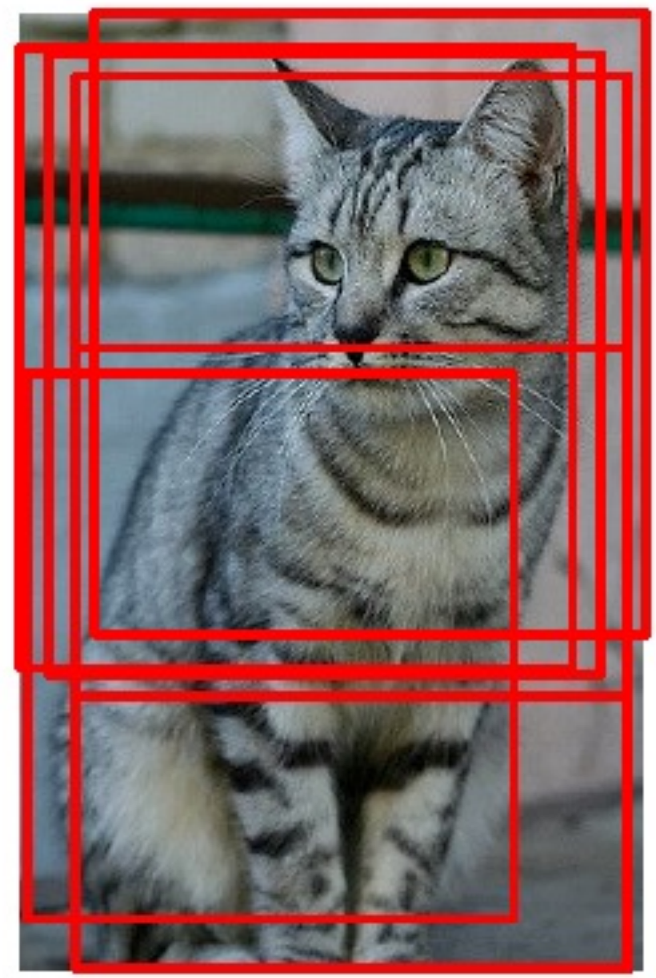
Data augmentation as regularization

Horizontal flips

Data augmentation as regularization

Random crops and scales

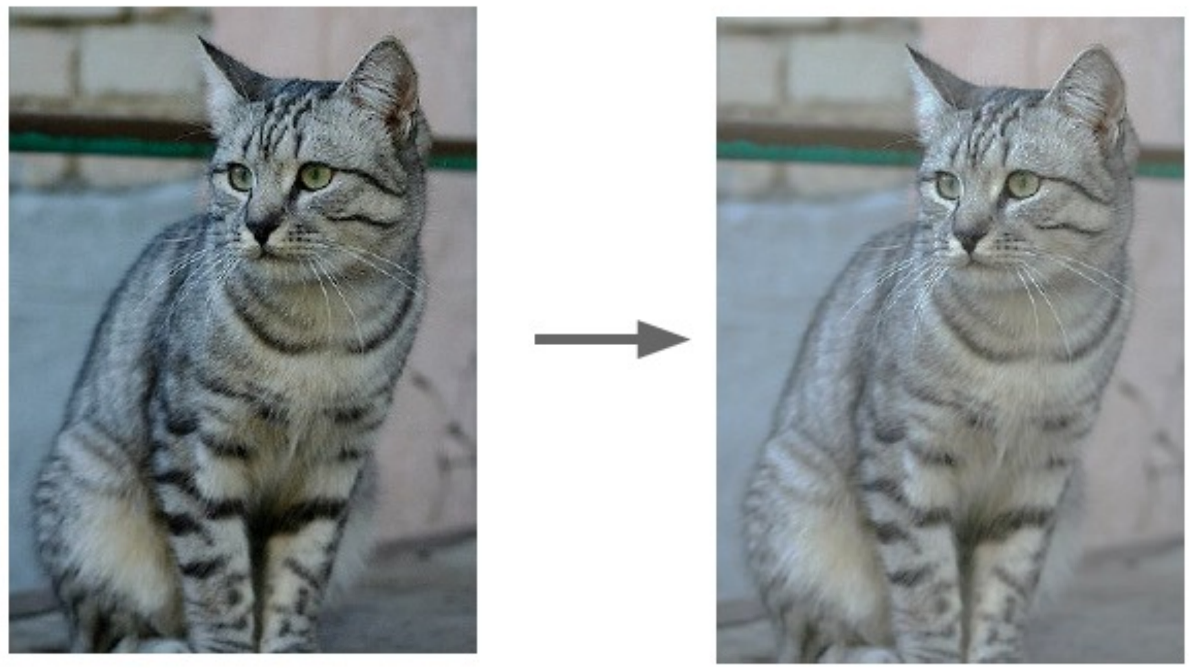
1. Pick random L in range [256, 480]
2. Resize training image, short side = L
3. Sample random 224 x 224 patch



Data augmentation as regularization

Color jitter

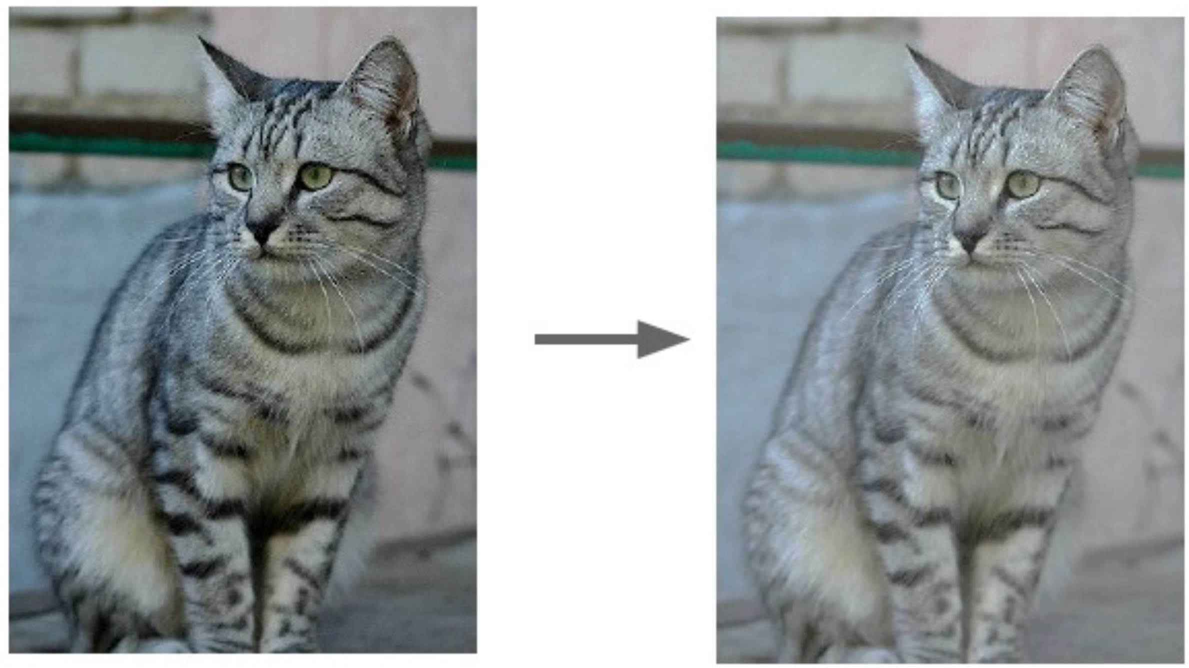
Simple: Randomize
contrast and brightness



Data augmentation as regularization

Color jitter

Simple: Randomize contrast and brightness



Can do a lot more: rotation, shear, non-rigid, motion blur, lens distortions, ….