DDA4220 · Fall 2026

# Lecture 7

## Deep Learning and Applications

Visual representations and transfer learning

Zhen Liu

CUHK-Shenzhen

# Lecture outline

**01 Visual representations**

02 Prediction evidence and adversarial examples

03 Perceptual loss and style transfer

04 Transfer learning

* denotes optional slides

# Representations

A representation encodes an object, concept or input in a form we can use.

| Numeral | Word | Marks |
| --- | --- | --- |
| 5 | five | ●●●●● |

Different representations of the same quantity

For an image, the pixel array is already one representation.

# Structure in image variation

The same person, viewed at different head orientations.

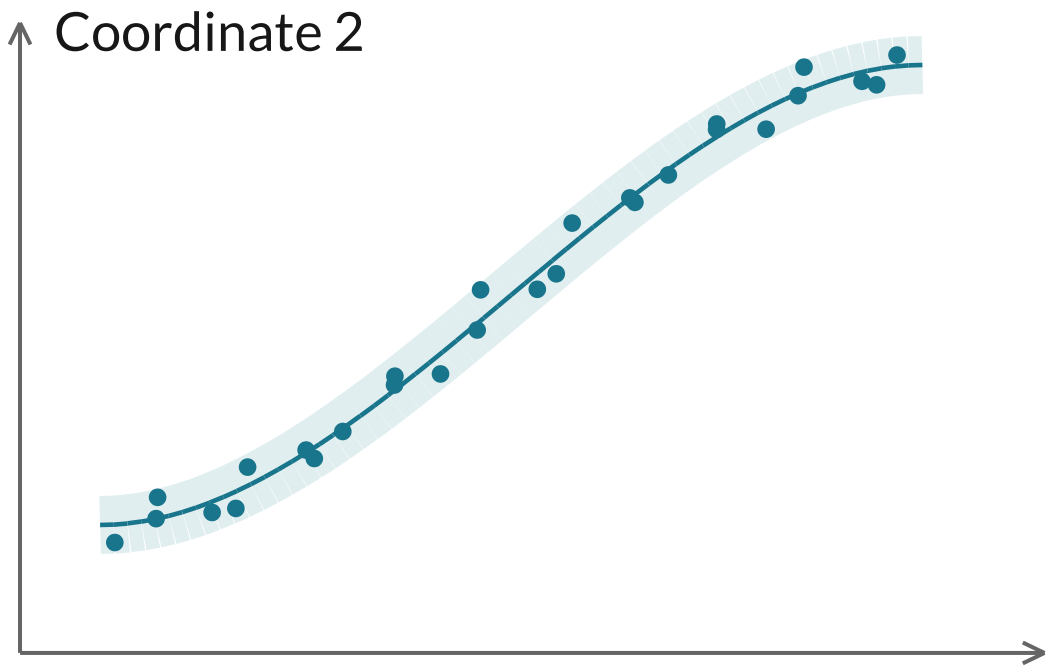
**Turn left / right**

**Tilt up / down**

Source: Deep Learning · QMUL Multiview Face Dataset

# The manifold hypothesis

Within an image family, many pixel values change together.




Two stored coordinates;
one main direction of variation

**Manifold hypothesis**

Natural images tend to lie near lower-dimensional structure within pixel space.

This motivates learning compact descriptions of images.

# Learned image representations

A CNN computes feature values from the input image.

The classifier uses those features to make a prediction.

Image x



Trained CNN



$z = f(x)$

Feature values



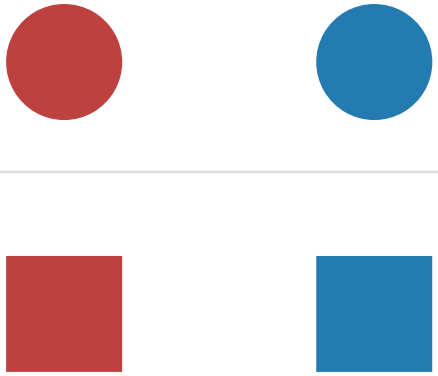
Class scores

Useful features can have more coordinates than the input.

# Task-relevant information

The useful differences depend on what we want to predict.

**Recognize shape**

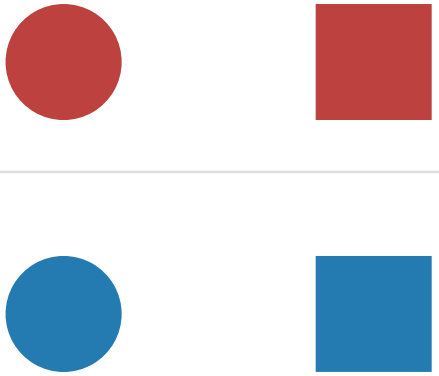


Circle

Square

Color can vary within a class.

**Recognize color**



Red

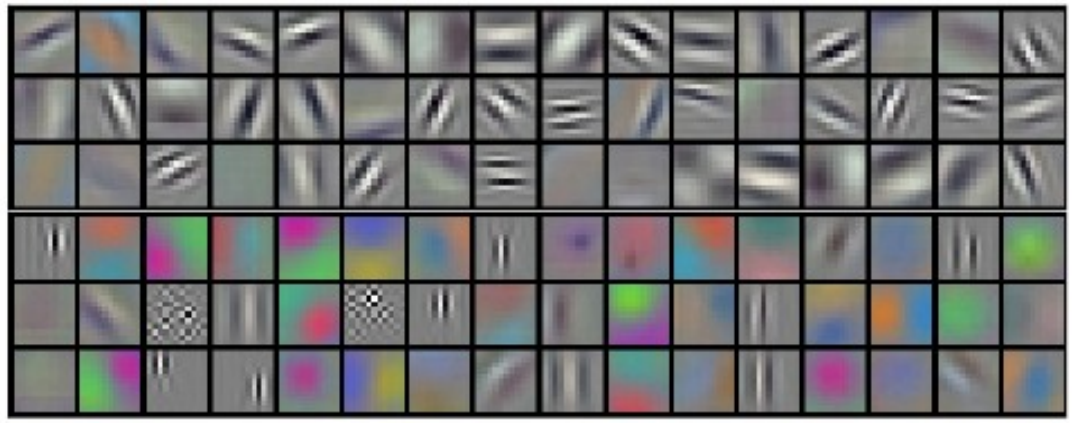
Blue

Shape can vary within a class.

# Weights and feature values

Convolution applies stored weights to each image patch.

**Filter weights**



**Feature values**

$$A(x) = \mathrm{CNN}(x; W)$$

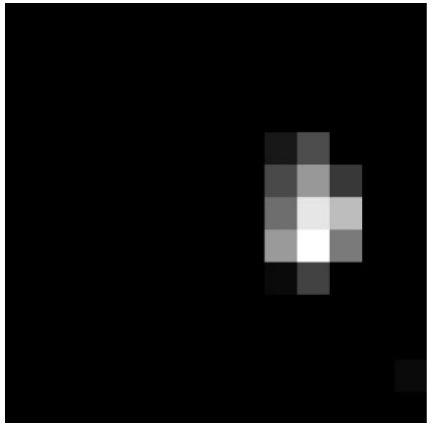
W: stored parameters
x: current image
A(x): responses to that image

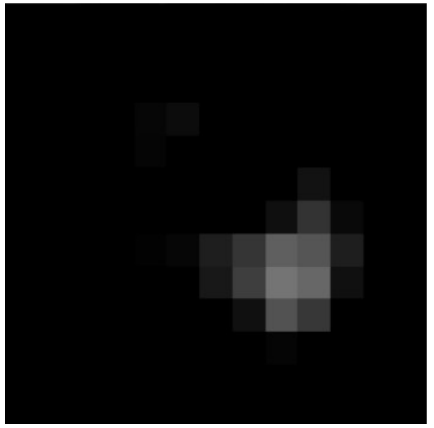
Source: AlexNet

# Reading an activation map

Input image

Same channel





**Brightness**

Brighter = larger response

**Position**

Each map entry corresponds to a region of the input.

Weights stay fixed.
Responses depend on the image.

Source: Understanding Neural Networks Through Deep Visualization

# Finding activating image patches

1 **Choose one channel**

Keep the trained network fixed.

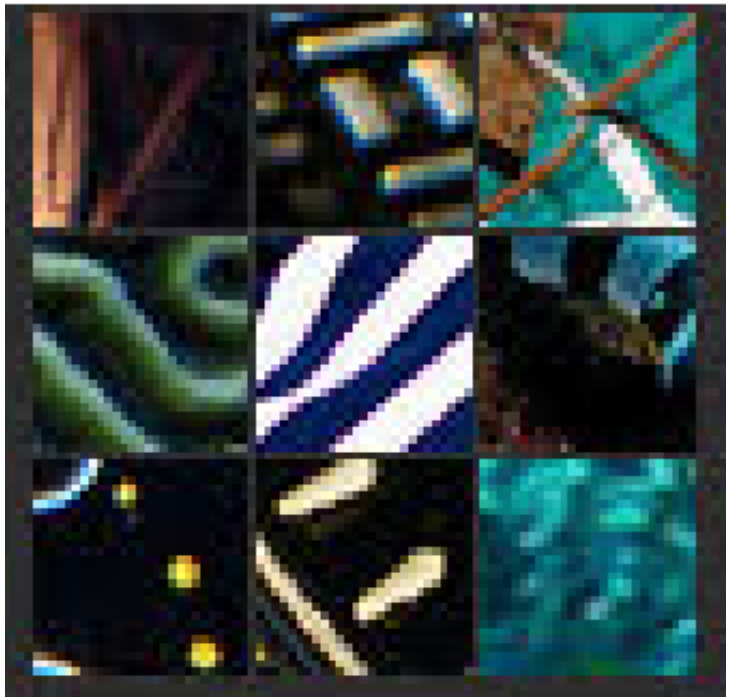
2 **Evaluate many images**

Rank this channel's responses.

3 **Inspect input regions**

Show each selected unit's receptive field.

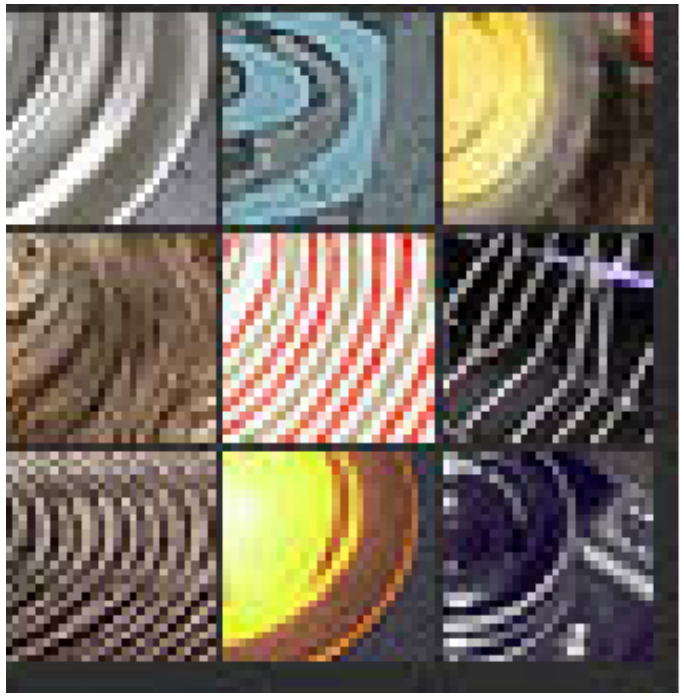
Patches with high responses



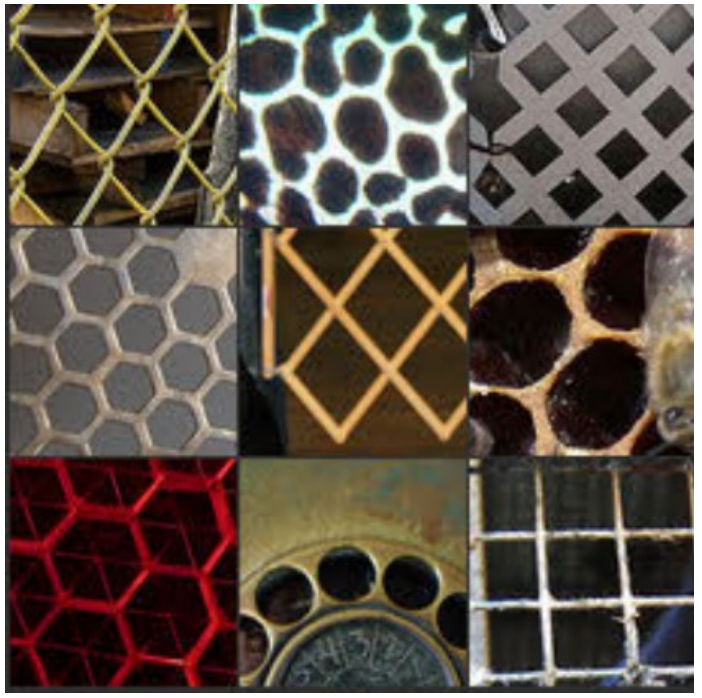
Source: Visualizing and Understanding Convolutional Networks

# Features at different depths

Within each group: different images, the same selected channel.



Layer 2



Layer 3

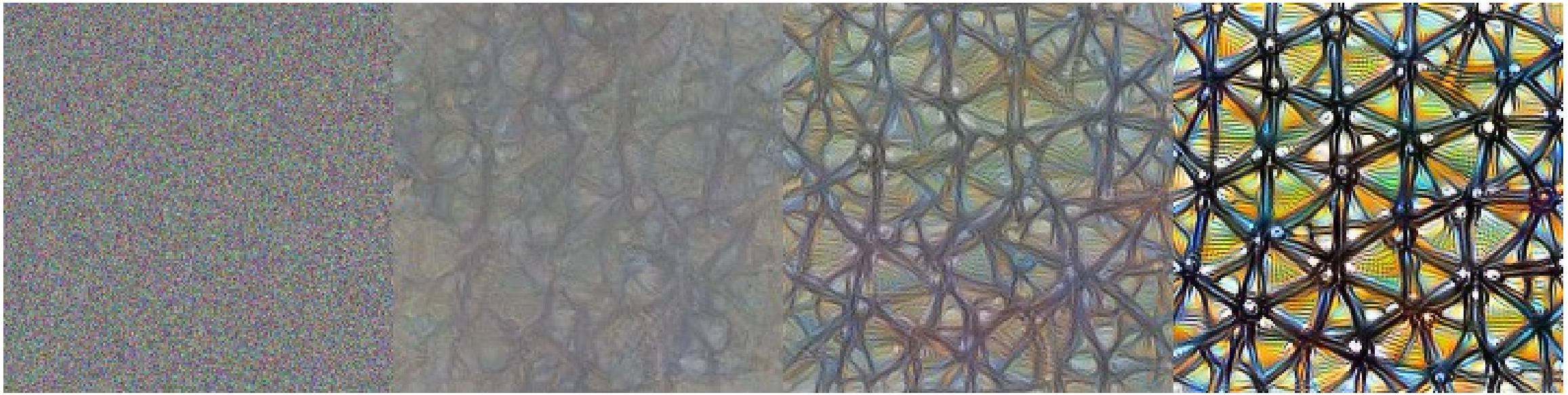
Layer 4

Source: Visualizing and Understanding Convolutional Networks

# Searching for a preferred input

**Dataset search** Select an existing image.

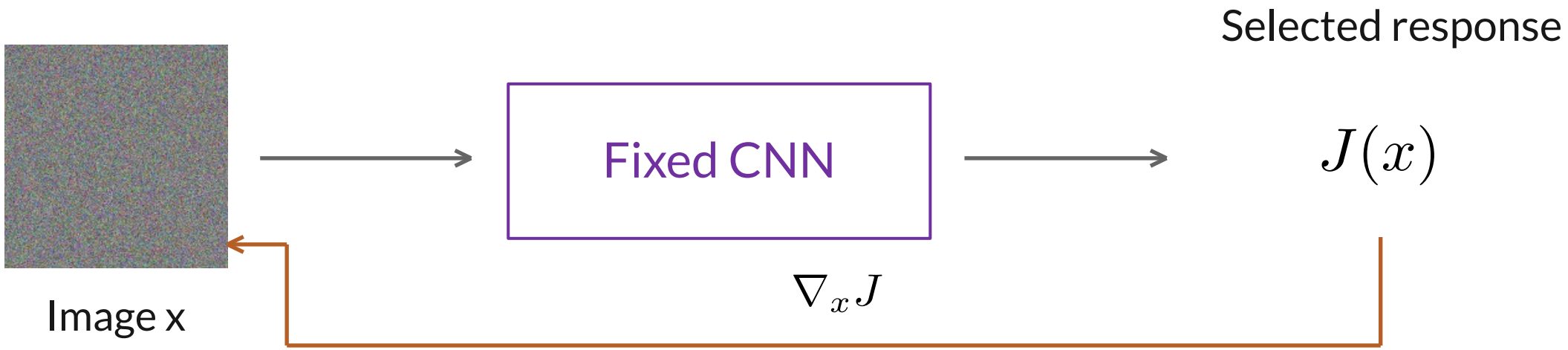
**Activation maximization** Change pixels to increase a chosen response.



Source: Feature Visualization

# Pixel gradients

Compute pixel gradients while keeping the weights fixed.




For a small change to one pixel:

Positive derivative → increasing that pixel raises the response.

Negative derivative → decreasing that pixel raises the response.

$$x \leftarrow x + \eta \nabla_x J$$

Source: Feature Visualization

# Activation maximization

## Activation maximization

Fix the pretrained network weights.

Initialize image x.

**Repeat:**

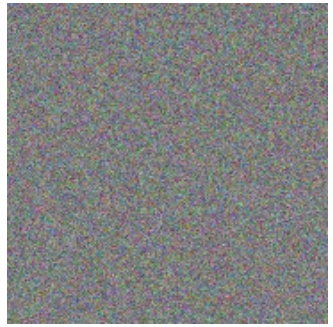
Evaluate the selected response J(x).

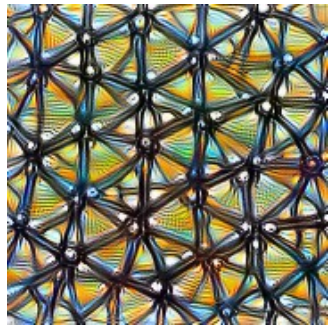
Compute pixel gradient g.

$$x \leftarrow x + \eta g$$

Keep pixels within their valid range.

Image variable





With image regularization

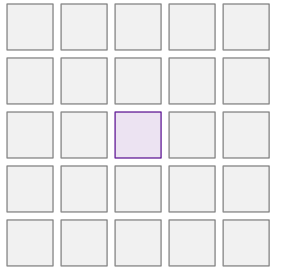
Source: Feature Visualization

# One unit and one channel

## One spatial unit

One value at one spatial position

Feature map



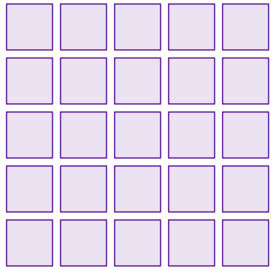
Optimized input

Maximize one activation

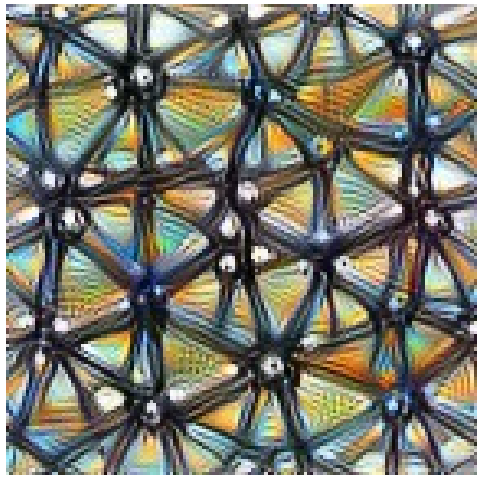
## One channel

Average the channel over all positions

Feature map



Optimized input



Maximize the channel average

Source: Feature Visualization

# Class visualization

## Class logit

A class score before softmax.

$$J(x) = s_k(x)$$

Increase the selected score itself.

Its probability could also rise when competing scores fall.

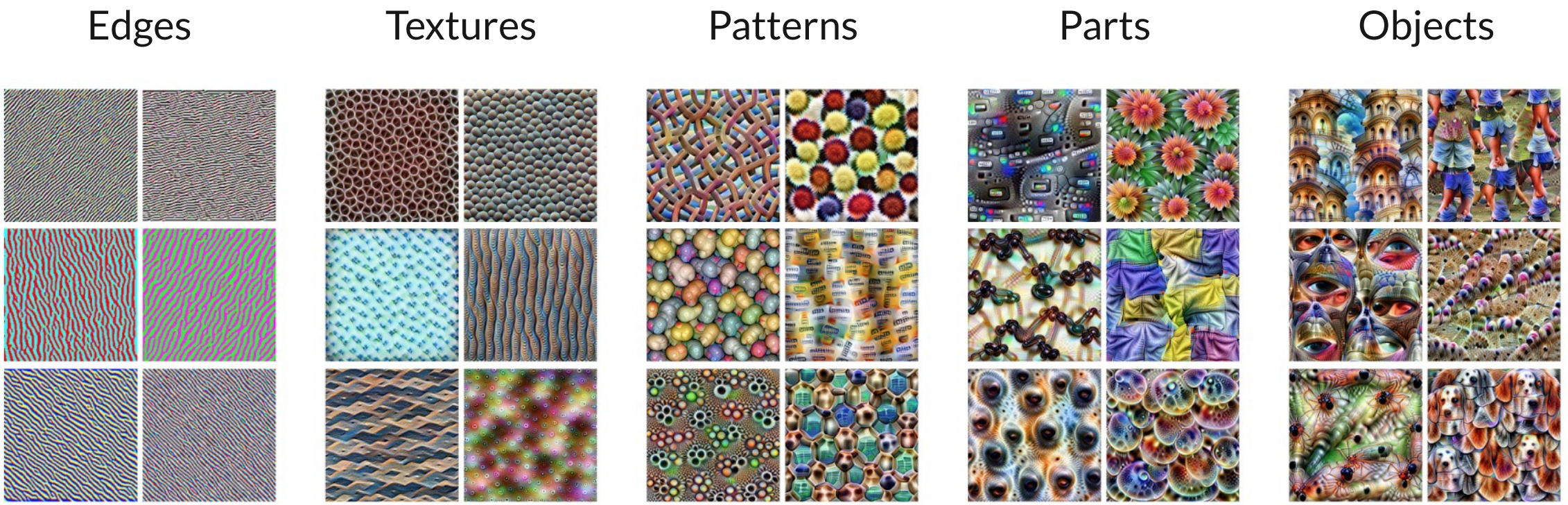
Source: Feature Visualization

# Input optimization in PyTorch

```
model.eval()                                    # Evaluation behavior
model.requires_grad_(False)                     # Freeze model weights
x = image.detach().clone()                      # Raw pixels in [0, 1]
x.requires_grad_(True)                          # Differentiate pixels
for _ in range(steps):
    score = model(normalize(x))[0, k]           # Selected class logit
    grad, = torch.autograd.grad(score, x)       # Pixel gradient
    with torch.no_grad():
        x.add_(step_size * grad)                # Gradient ascent
        x.clamp_(0, 1)                          # Valid pixel range
```

# Preferred patterns across layers

Optimized inputs for selected channels of one trained network.



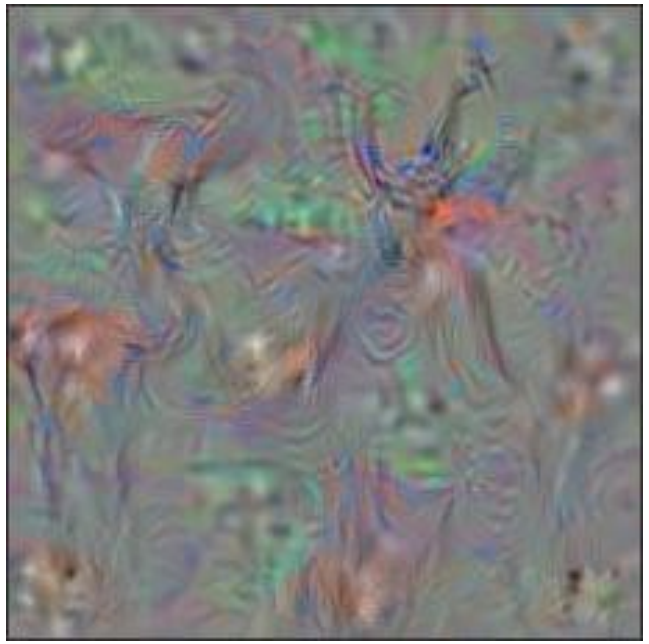

Source: Feature Visualization

# Image priors for visualization

Maximizing a response can produce noisy images.

Without image regularization

With blur during optimization



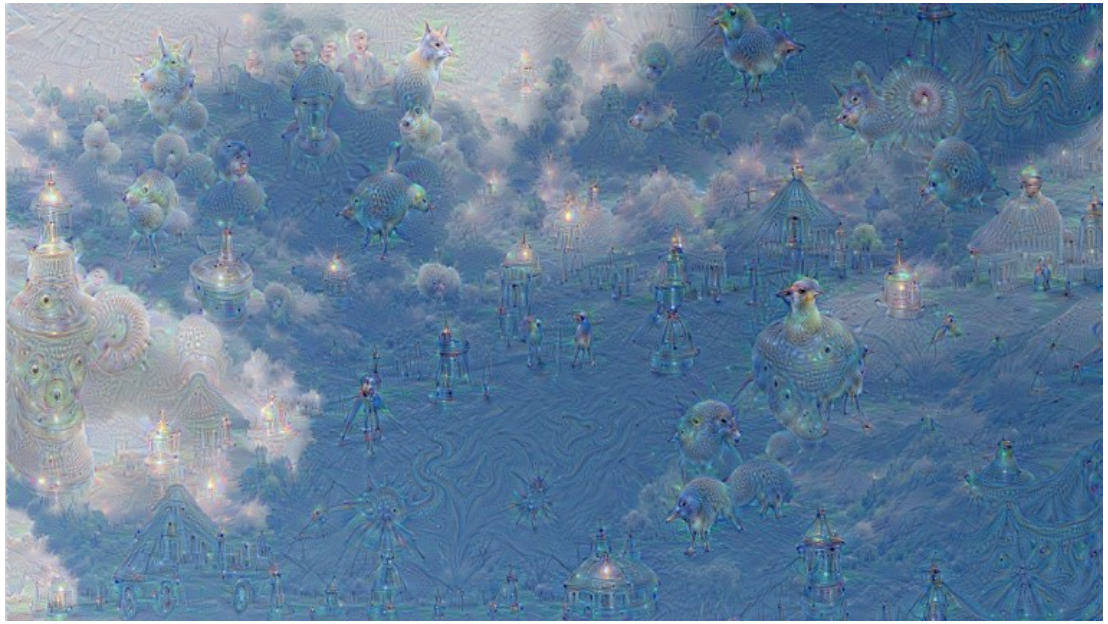
Source: Understanding Neural Networks Through Deep Visualization

# DeepDream

Keep the network fixed. Increase activations in a chosen layer.

Original photograph

After optimization

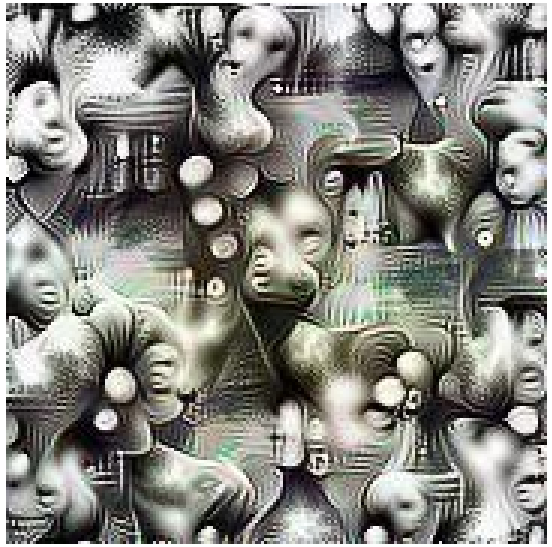


Source: Inceptionism

# Joint feature optimization

Maximize two channel responses together.

Maximize channel A



Maximize channel B

Maximize A + B

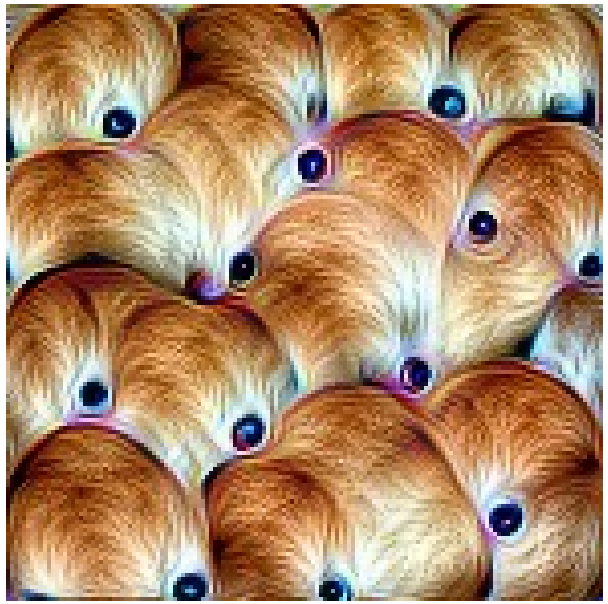
$$J(x) = J_A(x) + J_B(x)$$

Source: Feature Visualization

# A channel is not an object label

All these examples produce strong responses in the same channel.

Optimized input



Looks like dog heads

Natural examples

An enlarged example

A wooden spoon also gives a strong response.

Source: Feature Visualization

# Diverse inputs for one channel

One channel can respond to animal faces and cars.

## Within one channel

Different inputs can produce strong responses.

## Across channels

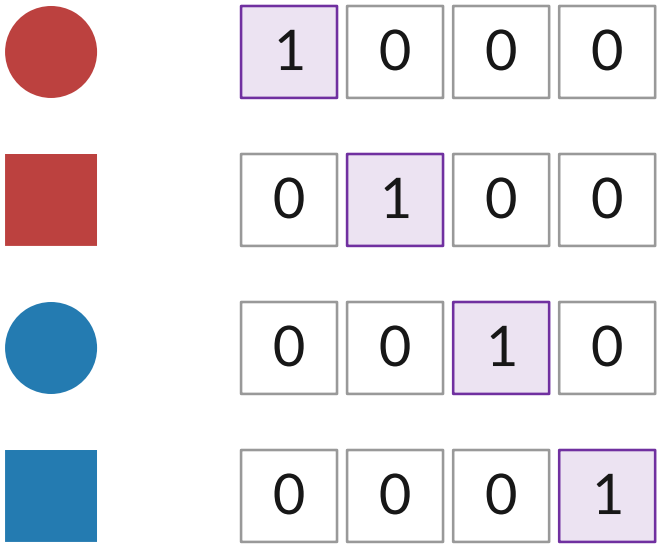
One image can strongly activate several channels.

Source: Feature Visualization

# Localist and distributed representations

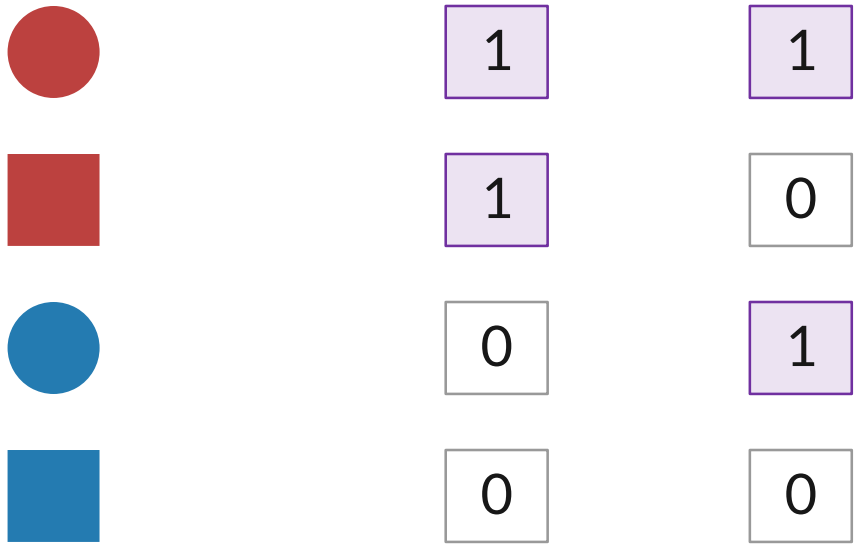
## Localist code

One coordinate per object type




## Distributed code

Combine shared properties

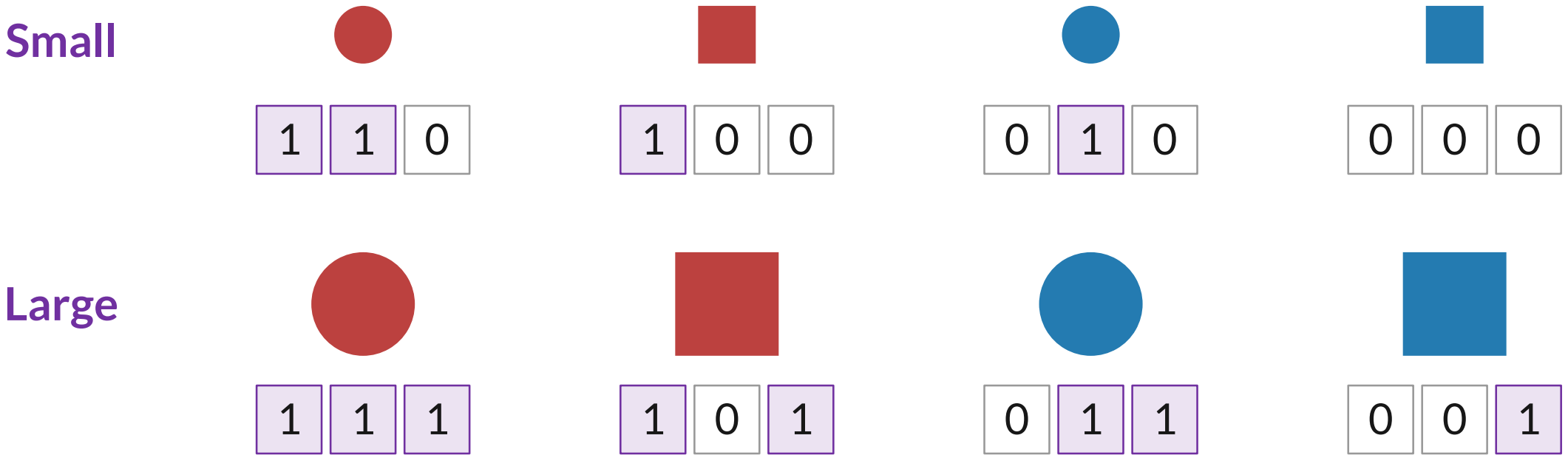



Here we specify the codes by hand. CNN feature values are learned.

# Combinations of shared features

A third shared property distinguishes small and large objects.

Hand-designed code order: red?, round?, large?



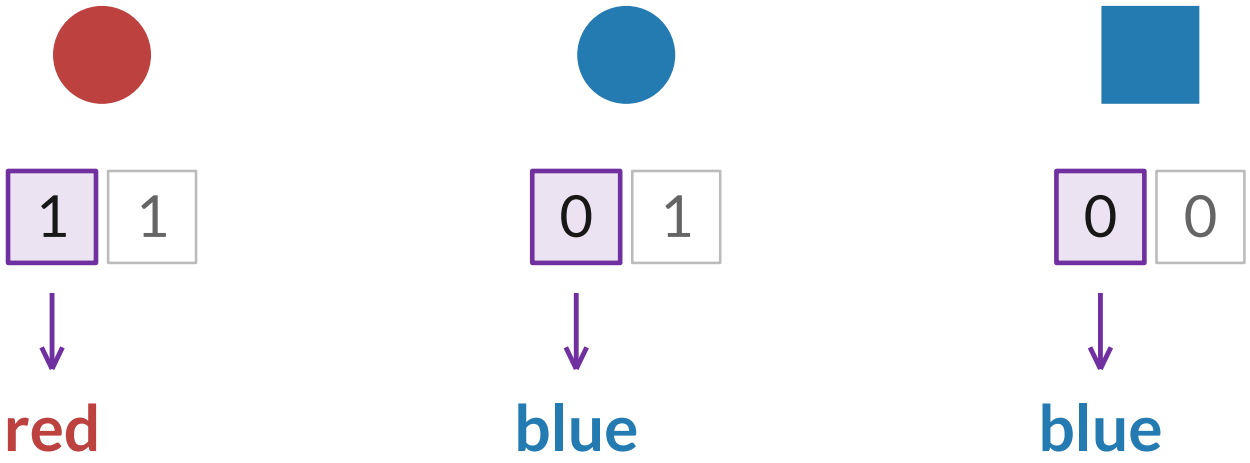

$n$ binary coordinates $\Longrightarrow$ $2^n$ possible code patterns

# Shared prediction rules

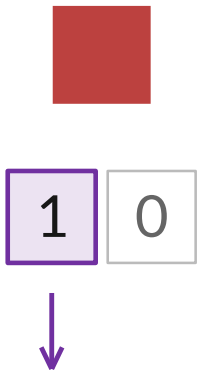
Use the first coordinate to predict color.

Hand-designed code order: red?, round?

**Examples used to fit the color rule**




**Held-out object**

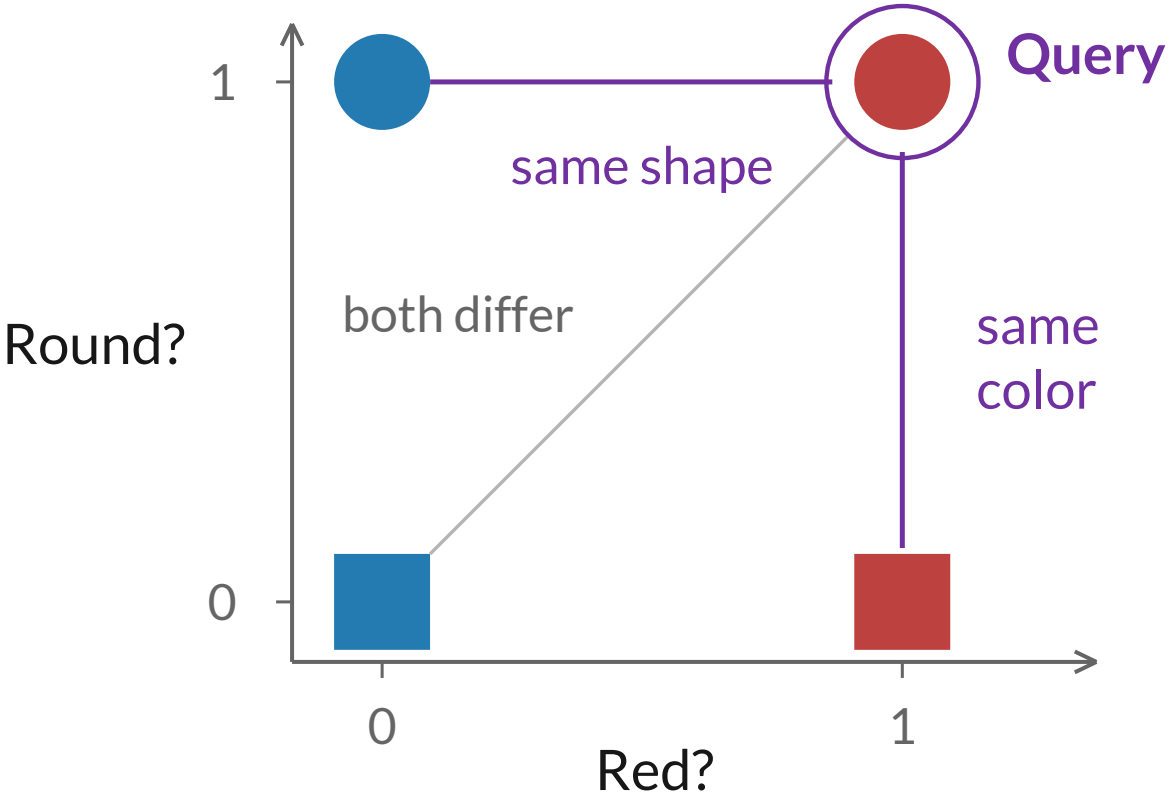



This rule also works for the unseen red square.

# Similarity in a feature code

Plot each object using its two feature values.

**Our two-feature code**




**Equal weight for both attributes**

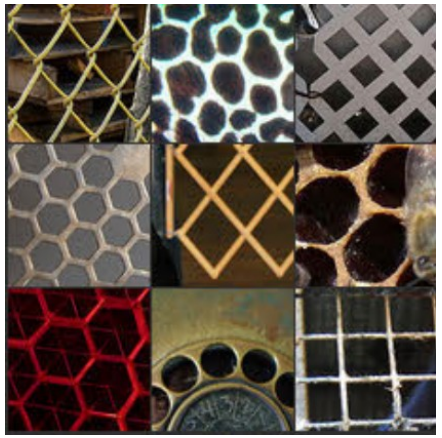
The red square and blue circle are equally close to the query.

For shape matching, choose the blue circle.

# Distributed features in a CNN

A class score combines many learned feature responses.

Examples from different layers



Layer 3

Layer 4

- One image activates several features.
- One feature can help several classes.
- Features have no predefined names.

**Class score**

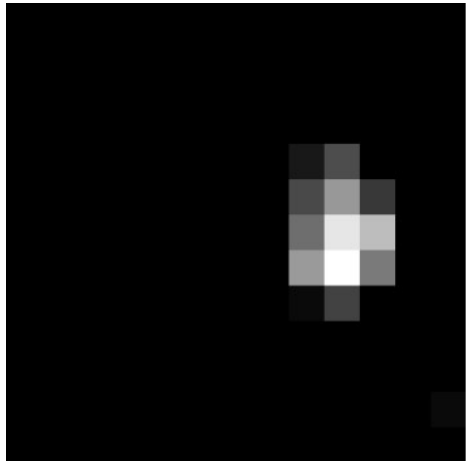
$$s_k = \sum_c w_{kc} z_c + b_k$$

z: final feature vector
w: classifier weights

Source: Visualizing and Understanding Convolutional Networks

# Image descriptors

An image descriptor is one vector describing the whole image.



$$z_c = \text{mean}(A_c) \longrightarrow z = [z_1, \ldots, z_C]$$

One channel | Average over positions | One entry per channel

Pooling discards exact feature locations.

Source: Understanding Neural Networks Through Deep Visualization

# Image retrieval

**Query** Nearest descriptors (Euclidean distance)

Use the same fixed CNN to compute every image's descriptor.

Rank images by the distance between their feature vectors.

Source: CS231n

# Lecture outline

01 Visual representations

**02 Prediction evidence and adversarial examples**

03 Perceptual loss and style transfer

04 Transfer learning

# Evidence for a class prediction

Fixed photograph

Trained classifier

One logit for each class

**Cat** $s_{\text{cat}}$

Dog $s_{\text{dog}}$

Other classes

Locate regions that support the selected class

Source: Grad-CAM

# Occlusion sensitivity

Selected output: the cat score

Original

Covered patch

**Compare the same score**

A drop suggests the region supported the cat score.

Repeat at other positions.

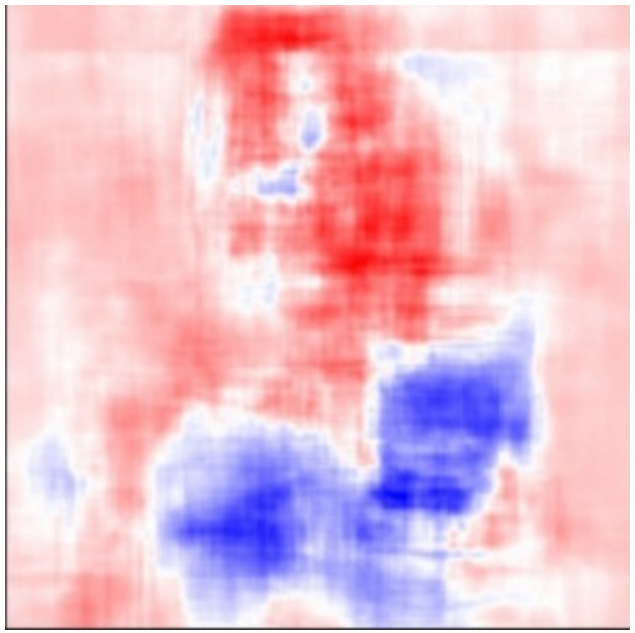
Source: Grad-CAM

# Occlusion maps

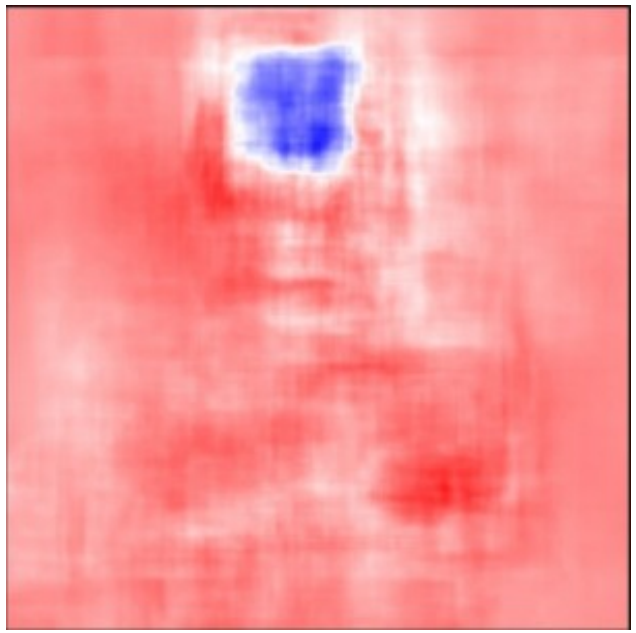
Map position = position of the covering patch

Original image

Monitor cat score



Monitor dog score



- Blue: a lower score after covering that position

Source: Grad-CAM

# Input-gradient saliency

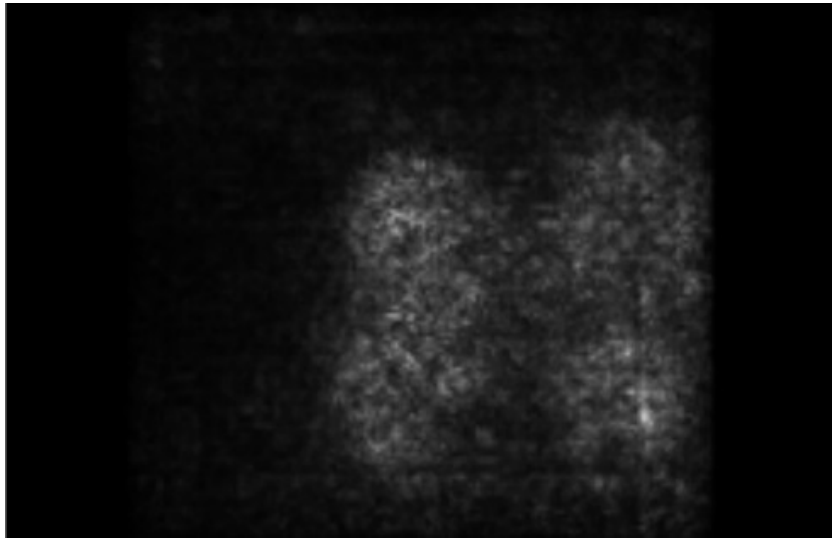
One backward pass measures sensitivity at every pixel

Input image

$$\left|\frac{\partial s_c}{\partial x_i}\right|$$

$x_i$ One pixel-channel value

Derivative magnitude



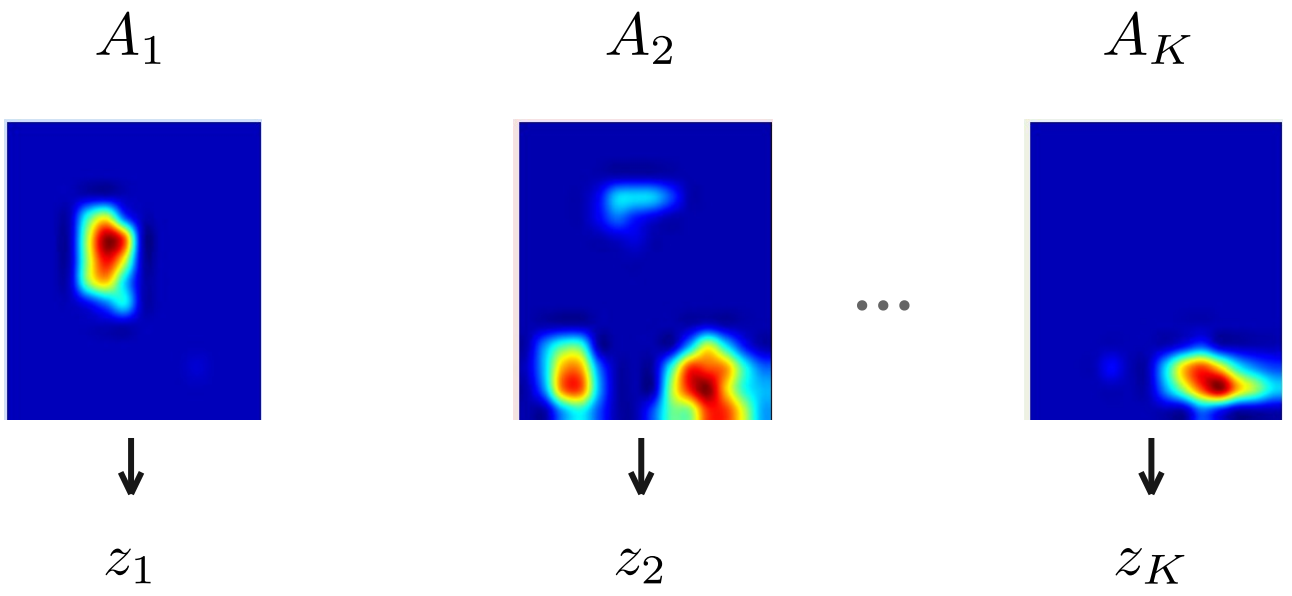
Bright: larger local sensitivity

For each pixel, display the largest magnitude across its RGB channels.

Source: Deep Inside Convolutional Networks

# Spatial maps and class scores

Global average pooling discards the location of a response.




**Spatial feature maps**

Warm: stronger response

Global average pooling:
one value per channel

Linear classifier:
combine the pooled values

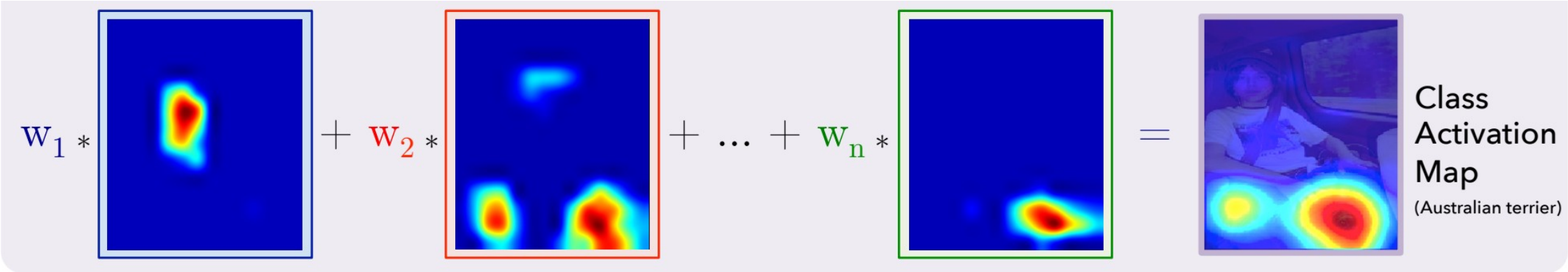
**Selected class score**

$$s_c = \sum_k w_{ck} z_k + b_c$$

Source: Learning Deep Features for Discriminative Localization

# Class activation mapping (CAM)

Combine feature maps with the classifier's learned weights.




**Map value at location (i, j)**

$$M_c(i,j) = \sum_k w_{ck} A_k(i,j)$$

$A_k$ Spatial response of channel k

$w_{ck}$ Learned weight for class c

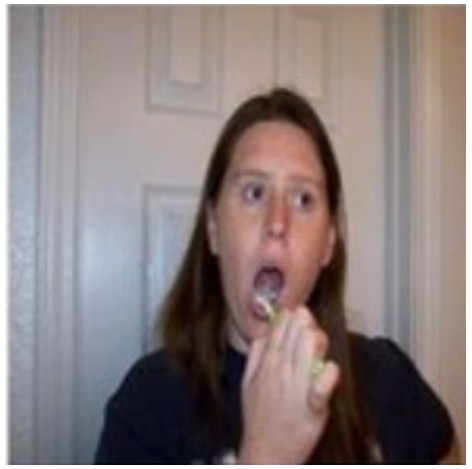
Source: Learning Deep Features for Discriminative Localization

# Reading a class activation map

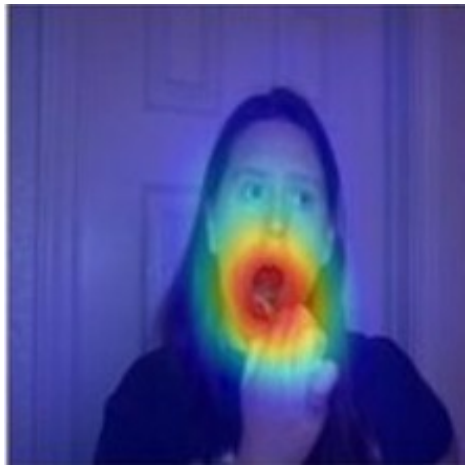
Classification can rely on a distinctive region

Brushing teeth

Input



CAM

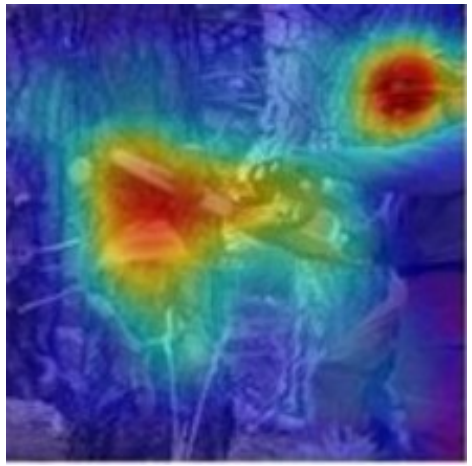


High response near the mouth and toothbrush

Cutting trees

Input

CAM

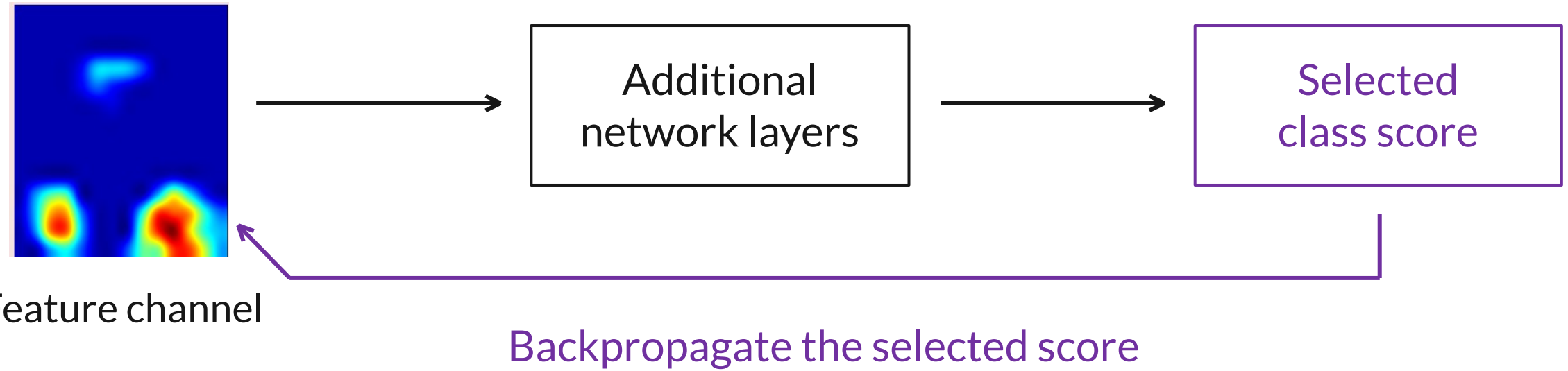


High response near the chainsaw and the cut in the tree

Source: Learning Deep Features for Discriminative Localization

# Channel influence through gradients

A nonlinear classifier has no single class weight for each channel.
Gradients measure how each channel affects the class score.




Source: Learning Deep Features for Discriminative Localization

# Gradient-weighted CAM (Grad-CAM)

## 1 Obtain one weight per channel

Average the gradient over locations.
A positive weight means stronger responses tend to raise the score.

$$\alpha_k^c = \frac{1}{HW} \sum_{i,j} \frac{\partial s_c}{\partial A_k(i,j)}$$

H × W spatial locations

---

## 2 Weight the original spatial maps

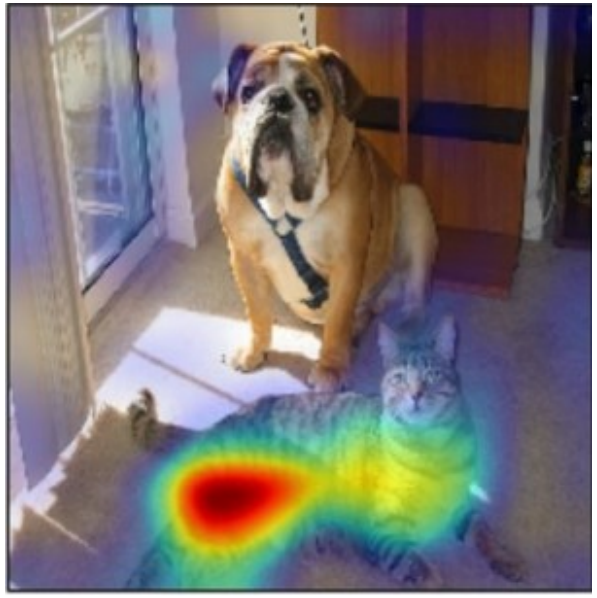
Weight and add the feature maps.
ReLU sets negative values to zero.

$$M_c = \mathrm{ReLU}\left( \sum_k \alpha_k^c A_k \right)$$

# Class-specific localization

Fixed image and classifier

Selected score: tiger cat



Selected score: boxer (dog)

- Warm colors: stronger support for the selected class
  Coarse evidence, not an object outline

Source: Grad-CAM

# Shape and texture

Predictions of an ImageNet-trained ResNet-50

| Elephant texture | Cat shape and texture | Cat shape, elephant texture |
| --- | --- | --- |
| | | |
| Indian elephant | Tabby cat | Indian elephant |

Source: [ImageNet-trained CNNs are biased towards texture]

# Adversarial examples

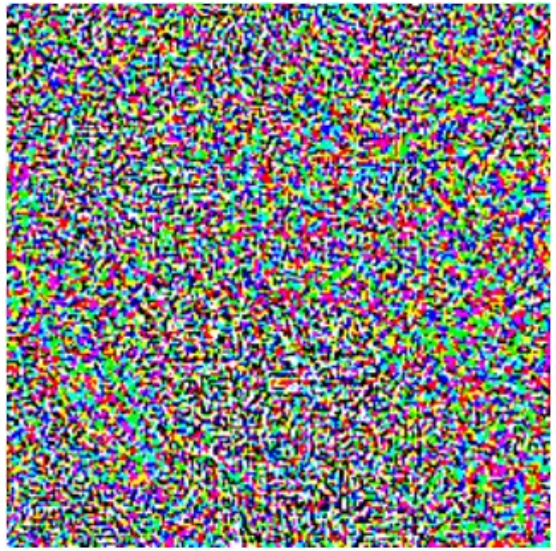
Both images show a panda.

Original image

Prediction: panda

+

Small perturbation



Amplified for display

=

Perturbed image

Prediction: gibbon

Source: Explaining and Harnessing Adversarial Examples

# Correct label, prediction and loss

Correct label: panda

**Cross-entropy for the correct class**

$$\mathcal{L} = -\log p_y$$

Lower probability for the correct class means higher loss.

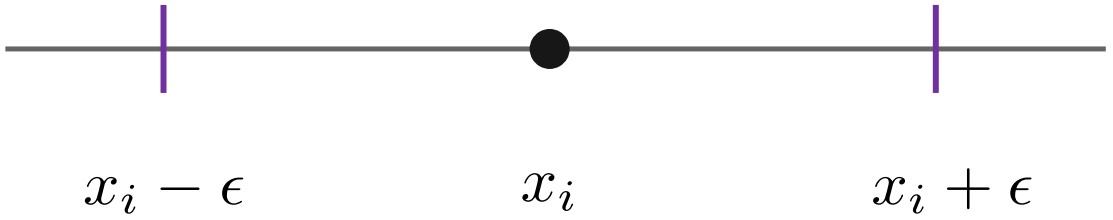
Training: change weights to reduce loss.
Attack: change pixels to increase loss.

Source: Explaining and Harnessing Adversarial Examples

# Perturbation budget

Each value changes by at most $\varepsilon$.
Many values can change together.

One RGB-channel value




$$|x_i^{\text{adv}} - x_i| \leq \epsilon$$

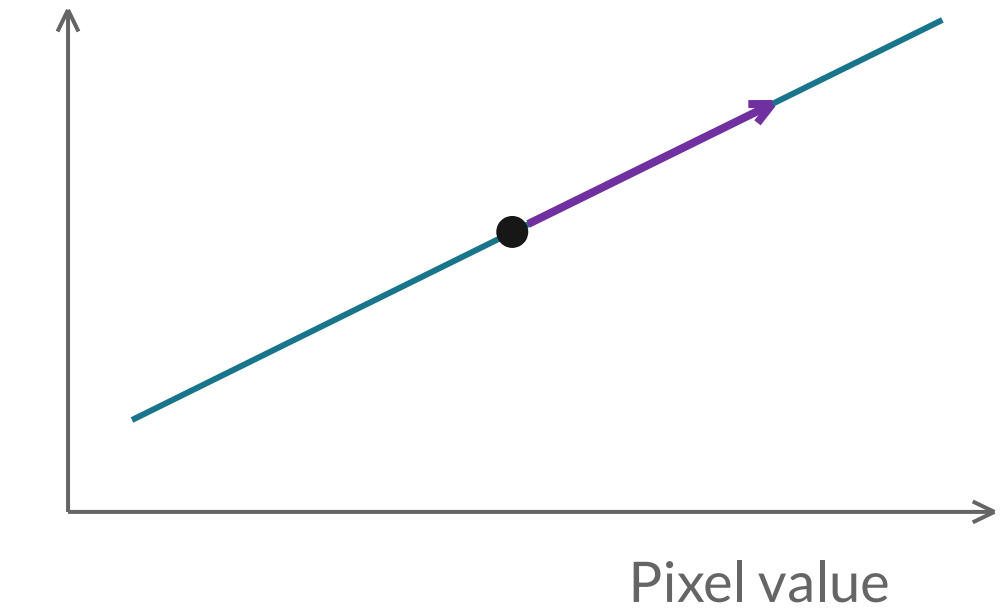
Raw image values remain in [0, 1]

Source: Explaining and Harnessing Adversarial Examples

# Choosing the direction of a pixel change

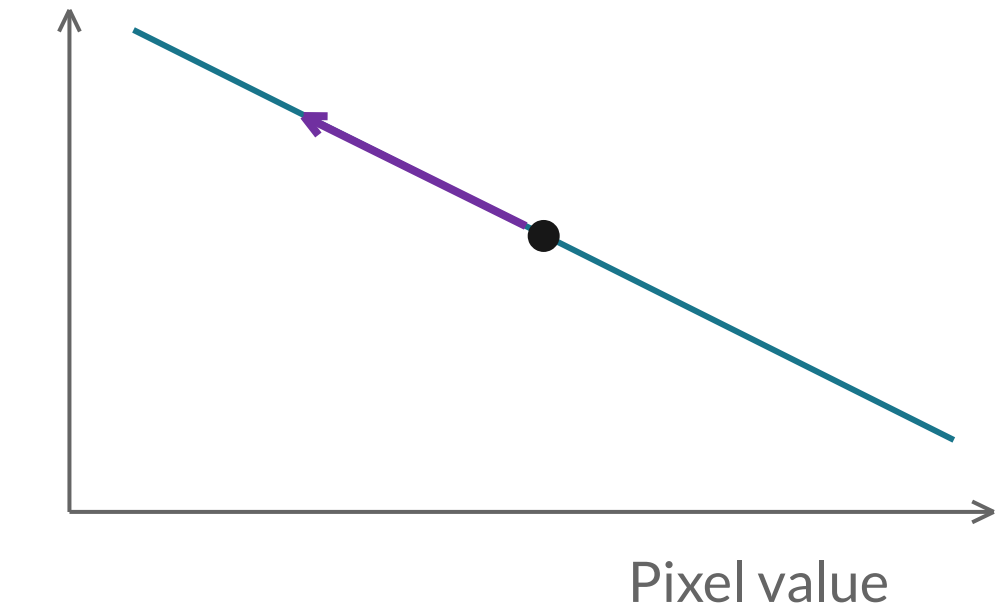
For one input value, use the local loss slope

Positive loss slope




Increase this value

Negative loss slope

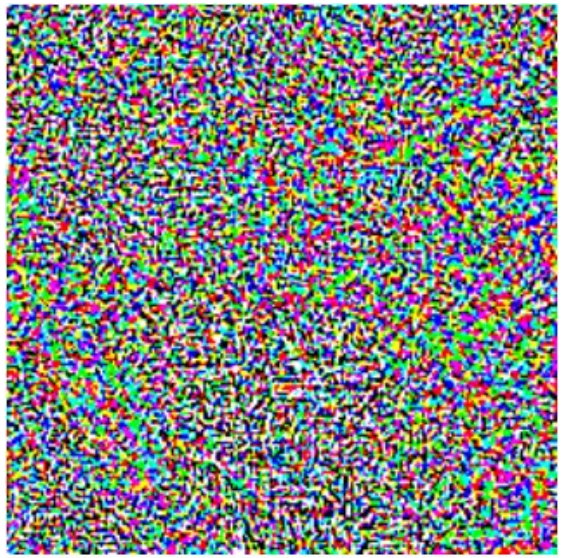



Decrease this value

# Small changes across many input values

Gradient-sign pattern

$$224 \times 224 \times 3 = 150{,}528 \text{ values}$$



**Local loss change**

$$\Delta \mathcal{L} \approx \sum_i g_i \delta_i$$

$g_i$ Loss sensitivity to one input value

$\delta_i$ Change applied to that value

With matching signs, the small increases in loss add up.

Source: Explaining and Harnessing Adversarial Examples

# Fast gradient sign method (FGSM)

Keep the model weights fixed

**1 Compute the input gradient**

$$g = \nabla_x \mathcal{L}(f_\theta(x), y)$$

**2 Take a step that raises loss**

$$x_{\text{adv}} = \text{clip}_{[0,1]}(x + \epsilon \ \text{sign}(g))$$

sign: +1 for positive slope, −1 for negative slope

Clip each value to [0, 1]

$\epsilon$ : largest allowed change in one image value

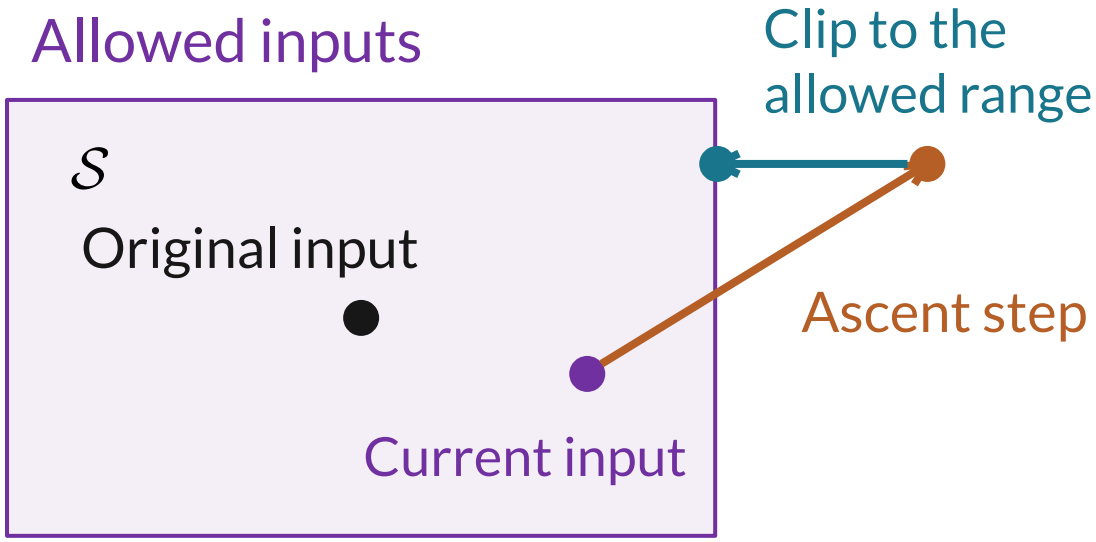
# FGSM in PyTorch

```
# x: raw images in [0, 1]; y: correct labels
model.eval()
x_adv = x.detach().clone().requires_grad_(True)  # track input
loss = F.cross_entropy(model(normalize(x_adv)), y)
grad, = torch.autograd.grad(loss, x_adv)         # input gradient
x_adv = (x + eps * grad.sign()).clamp(0, 1).detach()
```

normalize applies the preprocessing expected by the model

# Multi-step FGSM

Take several small steps, recomputing the gradient each time.




Start from the original input

$$x^{(0)} = x$$

$\alpha$ : size of one step
$\varepsilon$ : limit per input value

Clip to the original $\varepsilon$ -range and valid image values [0, 1].

Projected gradient descent (PGD)

$$x^{(t+1)} = \text{clip}_{\mathcal{S}}\left[x^{(t)} + \alpha \ \text{sign}(\nabla_x \mathcal{L})\right]$$

# Untargeted and targeted attacks

## Untargeted

Any wrong prediction

$$\max_{x_{\text{adv}} \in \mathcal{S}} \mathcal{L}(f_\theta(x_{\text{adv}}), y)$$

Raise loss for the
original correct label y

## Targeted

A selected wrong class

$$\min_{x_{\text{adv}} \in \mathcal{S}} \mathcal{L}(f_\theta(x_{\text{adv}}), y_{\text{target}})$$

Reduce loss for the
chosen target label

# Transfer of adversarial examples

Construct using A

Perturbed input

Evaluate with B

Model A + original image

Classifier B

Use A's input gradient

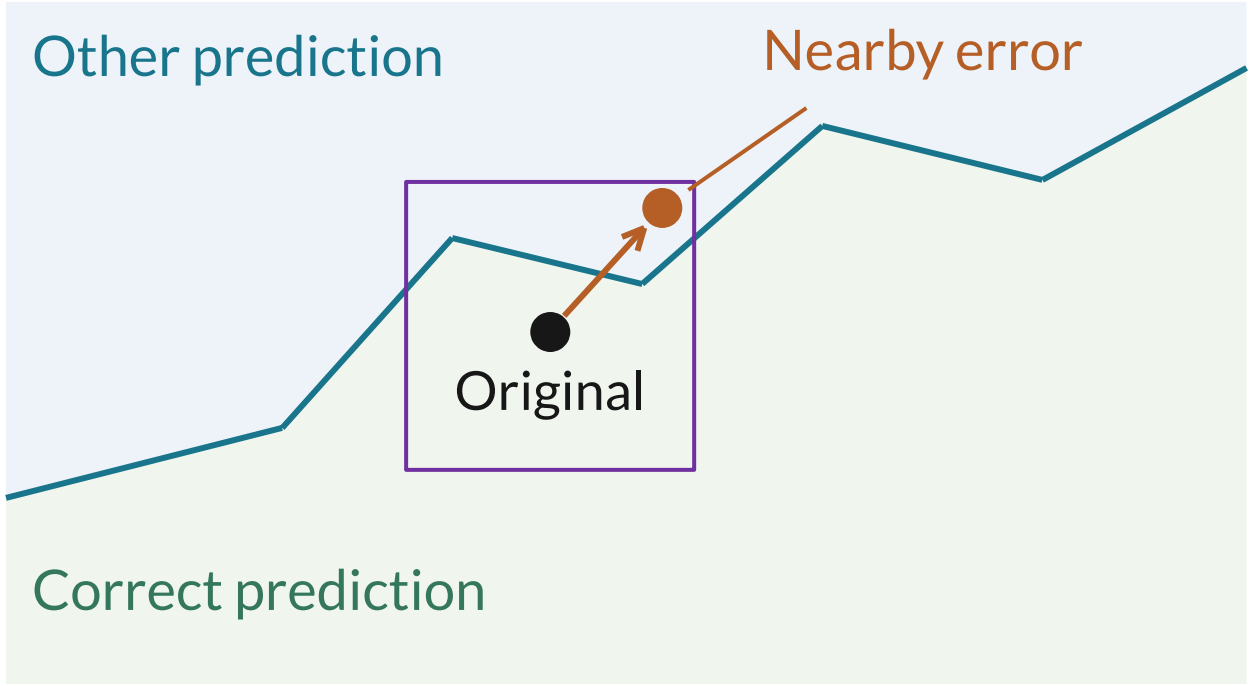
May also fool B

B's weights and gradients were not used to construct the attack

Source: Explaining and Harnessing Adversarial Examples

# Accuracy and robustness

A slice of input space




☐ Allowed changes

An unsuccessful attack does not prove robustness.

**Ordinary accuracy**

Correct at the original image

**Robustness within a budget**

Correct for every allowed change

# Adversarial training

## 1 Construct adversarial images

Keep weights fixed.
Find an allowed input that raises the loss.

## 2 Train on these adversarial images

Keep these inputs fixed.
Update weights to reduce loss for the original labels.

Source: Explaining and Harnessing Adversarial Examples

# Adversarial training algorithm

**One adversarial-training update**

For a minibatch x and correct labels y:

1. Construct adversarial images, e.g. using FGSM.
2. Detach the images and use them as fixed training inputs.
3. Compute loss: $\mathcal{L} = \mathrm{CE}(y, \mathrm{softmax}(f_\theta(x_{\mathrm{adv}})))$
4. Backpropagate the loss and update the model weights.

# Lecture outline

01 Visual representations

02 Prediction evidence and adversarial examples

**03 Perceptual loss and style transfer**

04 Transfer learning

# Pixel distance and visual similarity

Blurred

Reference

Warped

**Pixel loss prefers blur**

**People preferred warping**

Pixel loss compares the same location in both images.
Moving an edge can create many mismatched pixels.

Source: Deep Features as a Perceptual Metric

# Perceptual loss

Image x

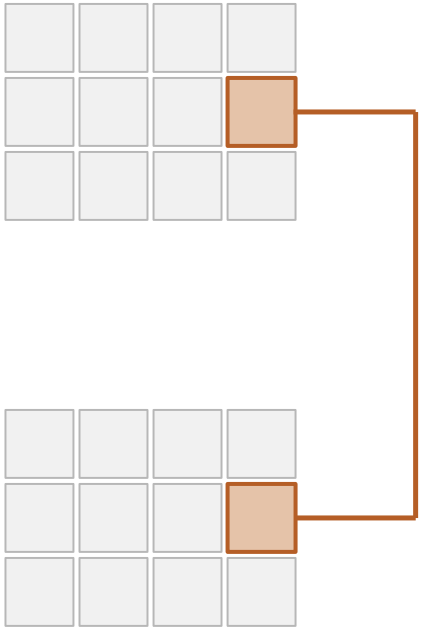
Image y

Weights stay fixed

Fixed CNN

Fixed CNN

**Feature responses**



**Same channel**
**Same position**

F: selected CNN features

Subtract paired responses, square, then add

$$\mathcal{L}_{\text{feat}}(x, y) = \|F(x) - F(y)\|_2^2$$

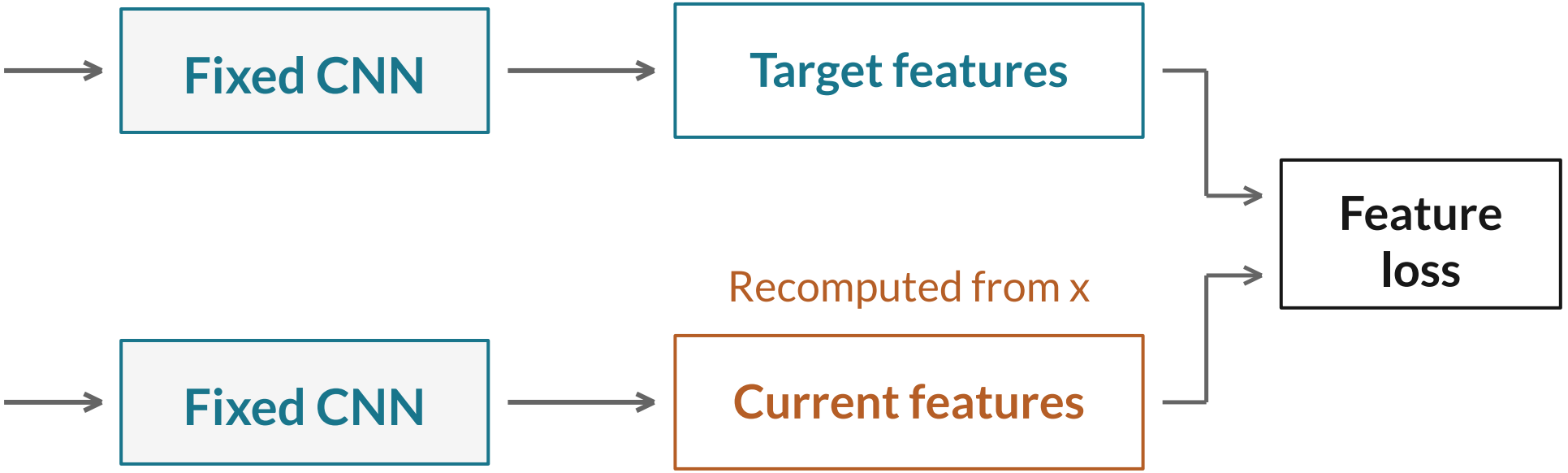
Source: Deep Features as a Perceptual Metric

# Feature inversion

Reconstruct an image to see what its features preserve.

Reference c

Image x




**Update the pixels of x**

$$\min_x \ \|F(x) - F(c)\|_2^2$$

Source: Understanding Deep Image Representations by Inverting Them

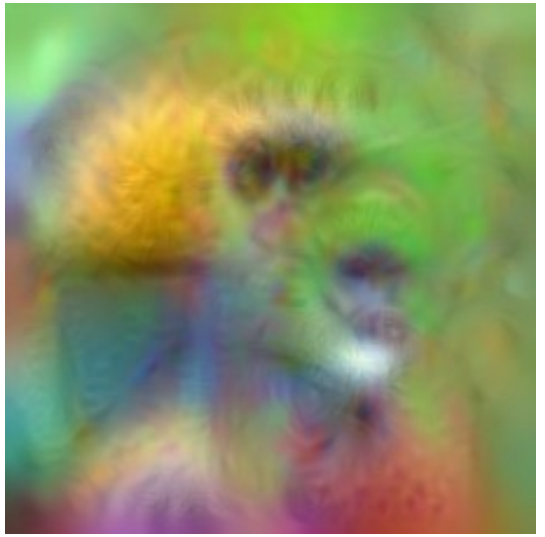
# Reconstructions from different layers

Reference

Early layer

Middle layer

Late layer



**These are reconstructed images, not feature maps.**

Deeper-layer reconstructions differ more in texture and shape.

Source: Understanding Deep Image Representations by Inverting Them

# Neural style transfer

Content photograph

Buildings and river layout

Style reference

Colors and repeated patterns

Optimized output

Scene layout with painted textures

Source: Image Style Transfer Using CNNs

# Matching the content representation

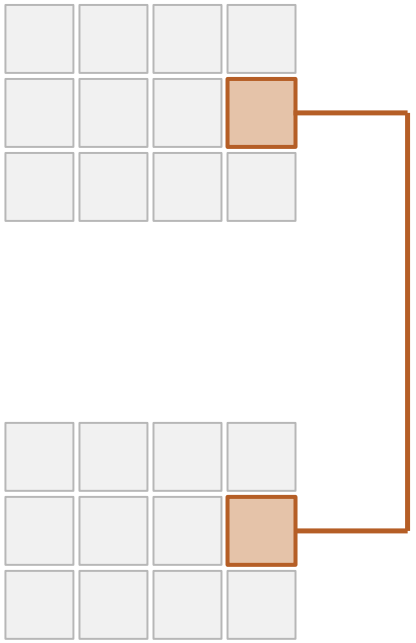
Content c

Output x

Fixed CNN

Fixed CNN

One feature map



Compare at the same channel and position

**Preserve the scene layout while changing its appearance.**

Source: Image Style Transfer Using CNNs

# Content loss

Content c

Output x

**Squared differences between feature responses**

$$\mathcal{L}_{\text{content}}^{(\ell)}(x, c) = \sum_i \sum_p \left( F_{i,p}^{(\ell)}(x) - F_{i,p}^{(\ell)}(c) \right)^2$$

F: feature responses $\ell$ : selected CNN layer

i: channel p: spatial position

**Each comparison uses the same channel and position.**

Source: Image Style Transfer Using CNNs

# Channel co-activation for style

Match recurring patterns without copying their locations.

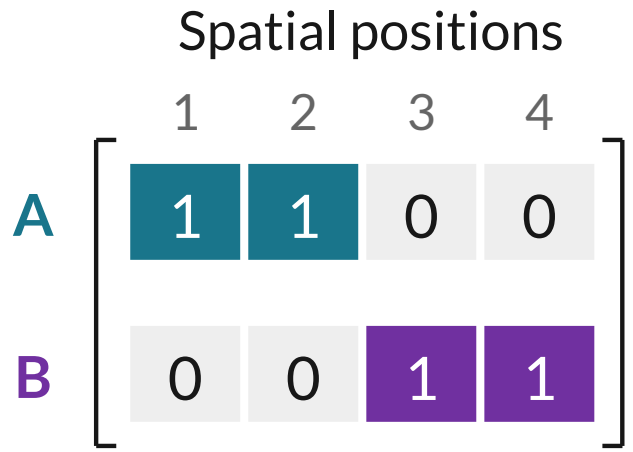
1 = active, 0 = inactive

# Gram matrix

At one CNN layer, flatten each channel's spatial grid into one row.

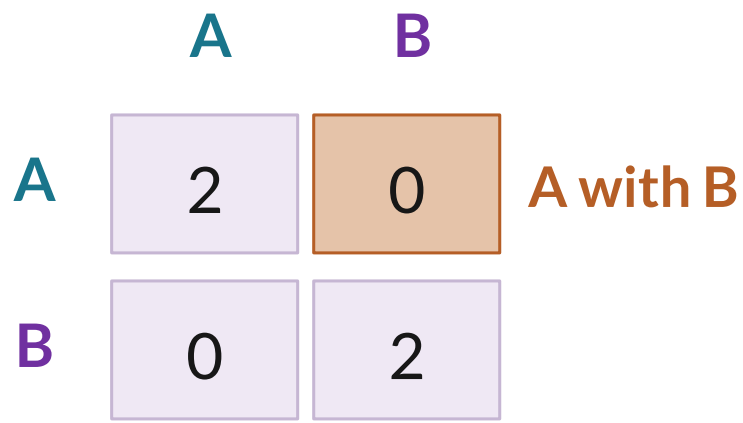
**Feature matrix F**

$F =$




2 channels × 4 positions

**Gram matrix G**




2 channels × 2 channels

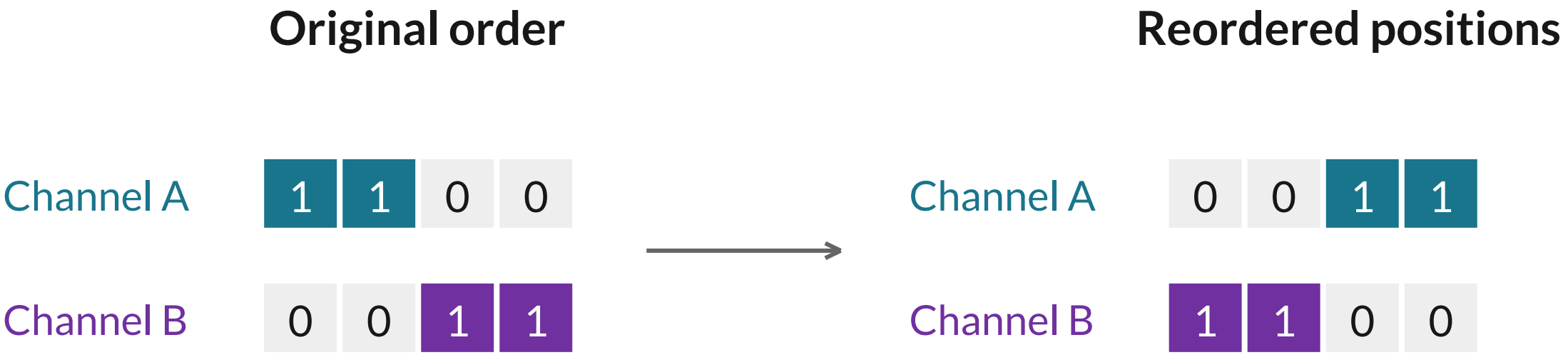
**All pairs at once**

$$G = FF^{\mathsf{T}}$$

**One pair: multiply and sum**

$$G_{AB} = \sum_{p} F_{A,p} F_{B,p}$$

# Spatial order in the feature maps




Move each whole column, keeping channel pairs together.

**The content loss changes**

**The Gram matrix is unchanged**

# Style loss

Style s

Output x

**Squared differences between Gram matrices**

$$E_\ell(x, s) = \sum_i \sum_j \left( G^{(\ell)}_{i,j}(x) - G^{(\ell)}_{i,j}(s) \right)^2$$

G: Gram matrix at the selected layer $\ell$

$$G^{(\ell)}(x) = F^{(\ell)}(x)\, F^{(\ell)}(x)^\mathsf{T}$$

i, j: pair of channels

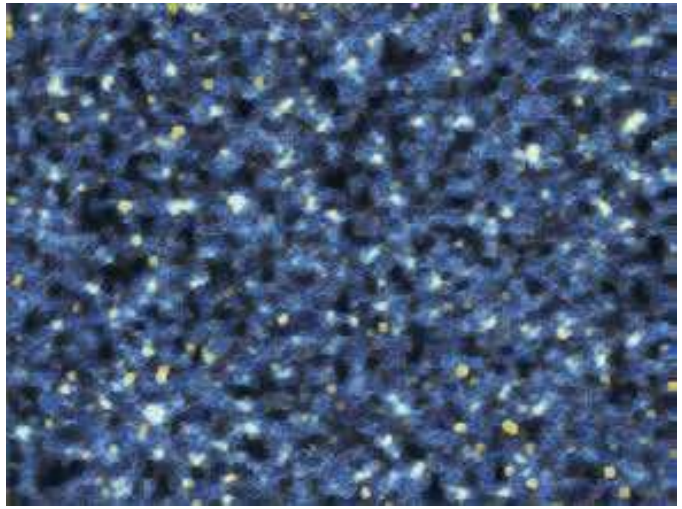
**Match how strongly channels respond together.**

Source: Image Style Transfer Using CNNs

# Matching style alone

Style reference

Early layer only



Early + deeper layers

Similar colors and textures, with a different layout.

Source: Image Style Transfer Using CNNs

# Content and style objectives

**Selected style layers**

$$\mathcal{L}_{\text{style}}(x, s) = \sum_{\ell \in \mathcal{S}} w_\ell E_\ell(x, s)$$

$\mathcal{S}$ chosen style layers

$w_\ell$ contribution of each layer

**Combined loss**

$$\min_x \; \alpha \, \mathcal{L}_{\text{content}}^{(\ell_c)}(x, c) + \beta \, \mathcal{L}_{\text{style}}(x, s)$$

$\ell_c$ selected content layer

$\alpha, \beta$ loss weights

**Update x: output pixels**

**Keep c, s and the CNN weights fixed**

Source: Image Style Transfer Using CNNs

# Optimizing the output image

Output image x

**CNN weights stay fixed**

**Repeat**

1. Extract features of the current image
2. Compute content and style losses
3. Backpropagate to the image pixels
4. Update the image pixels

Source: Image Style Transfer Using CNNs

# Content–style balance

Style reference

**More style emphasis**

**More content emphasis**

Source: A Neural Algorithm of Artistic Style · Image Style Transfer Using CNNs

# Super-resolution with perceptual loss

Predict a high-resolution image from a low-resolution input.

Target

Pixel loss

Perceptual loss

**Use a fixed CNN to compare output and target features.**

Train the super-resolution network using that feature loss.

Source: Perceptual Losses for Style Transfer and Super-Resolution

# Lecture outline

01 Visual representations

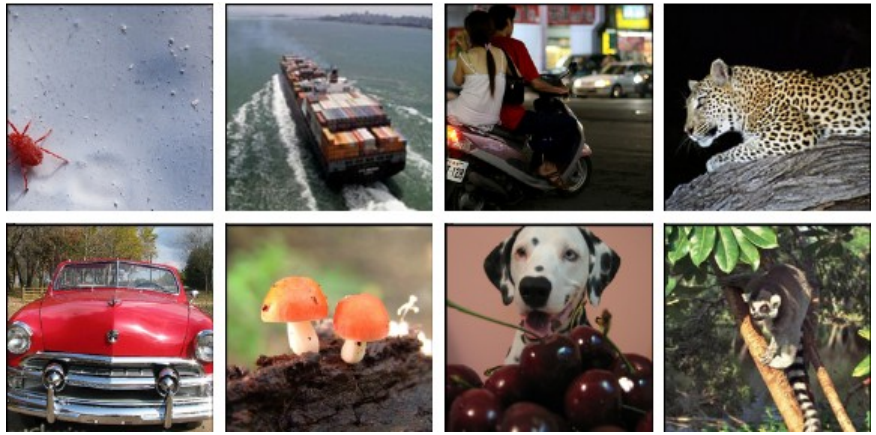
02 Prediction evidence and adversarial examples

03 Perceptual loss and style transfer

**04 Transfer learning**

# Transfer learning

## Pretraining

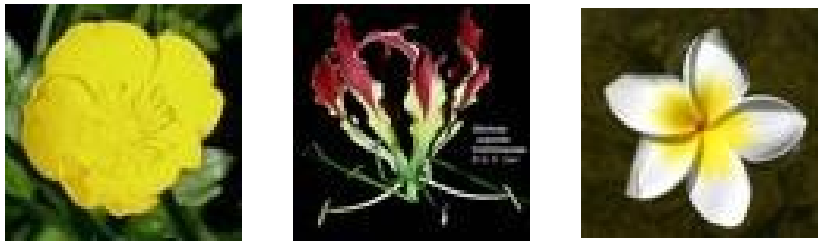


ImageNet object labels

Learn visual features from a large source dataset

**Reuse learned weights**

## Target training



Buttercup Fire lily Frangipani

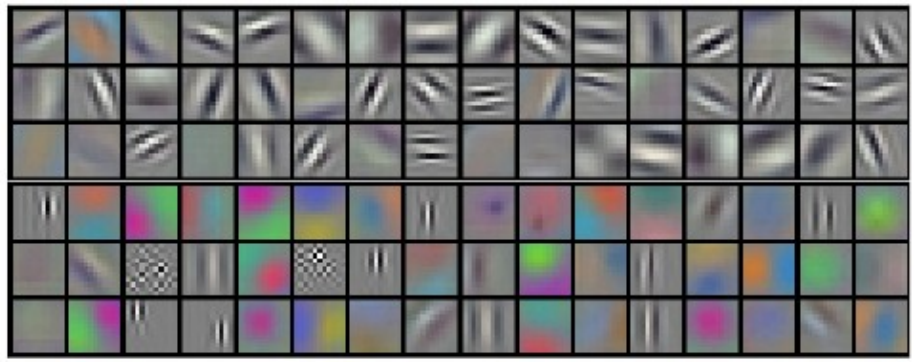
New flower labels

Use those features when target labels are limited

Source: AlexNet · Oxford Flowers 102

# Reusable visual features

## Early layers



Learned filter weights

Edges and color patterns recur across many image tasks

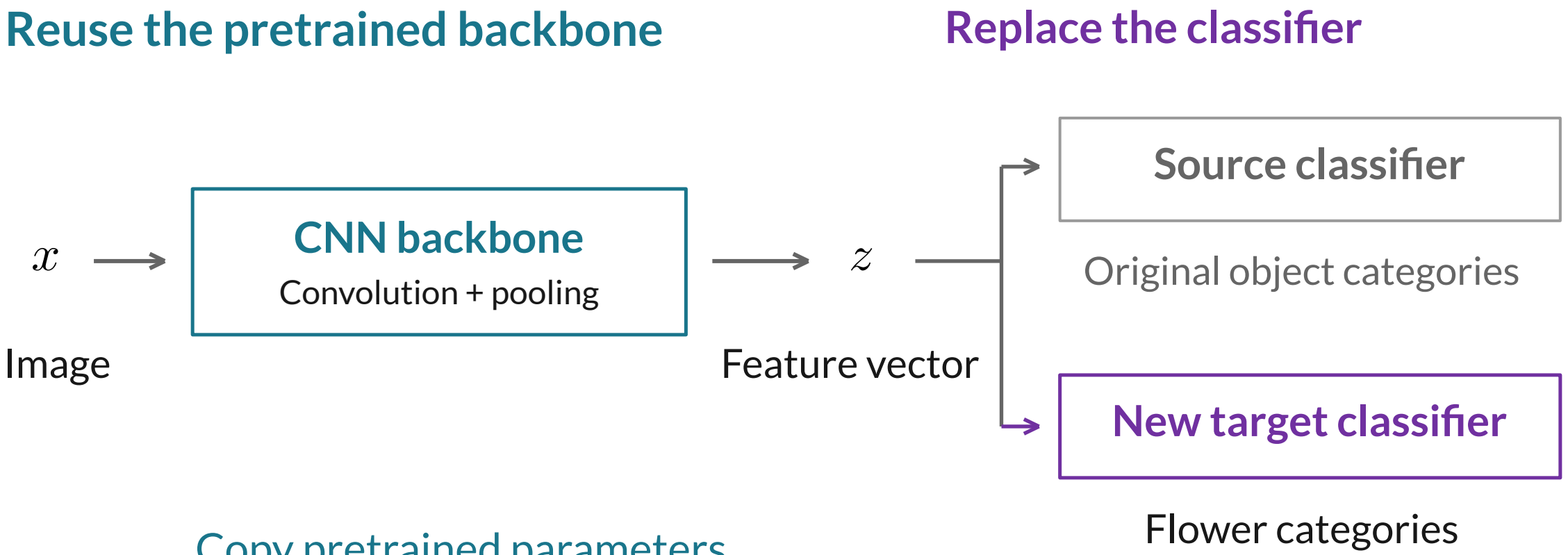
## Deeper layers

Activating image patches

Deeper features often specialize in the source task

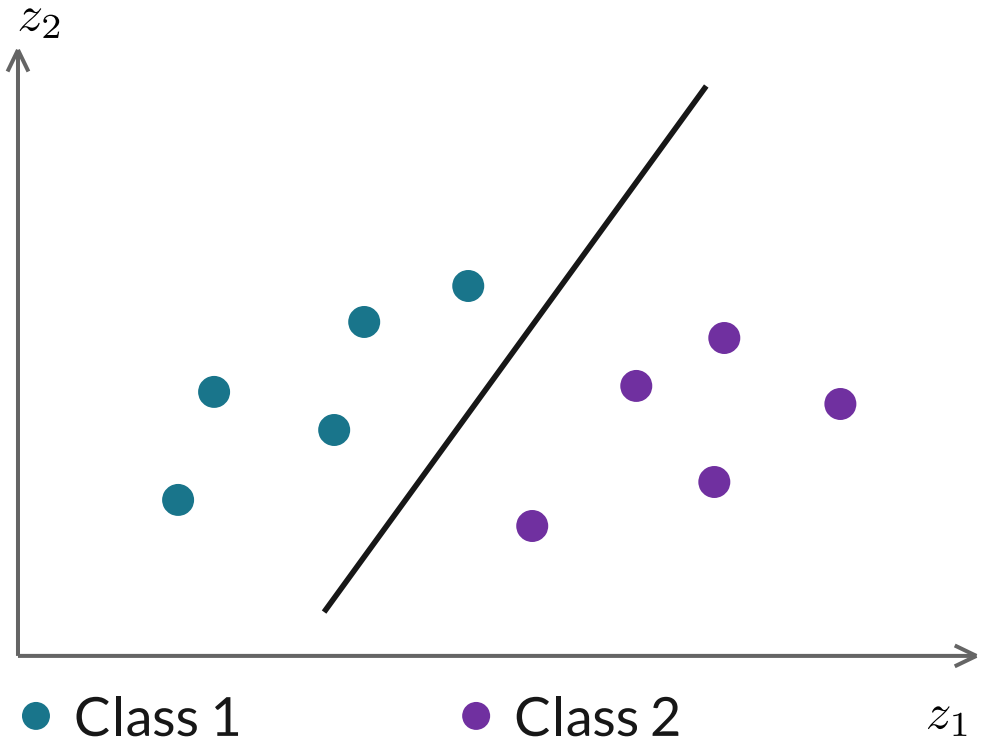
Source: AlexNet · Visualizing and Understanding CNNs

# Backbone and classification head



# Frozen features and linear probing

## Features stay fixed




Each point represents an image

## Learn a decision boundary

Fixed backbone

$$z = f_\theta(x)$$

Trainable linear head

$$\text{scores} = Wz + b$$

Train W and b with cross-entropy

# Transfer with frozen CNN features

## Bird-species classification

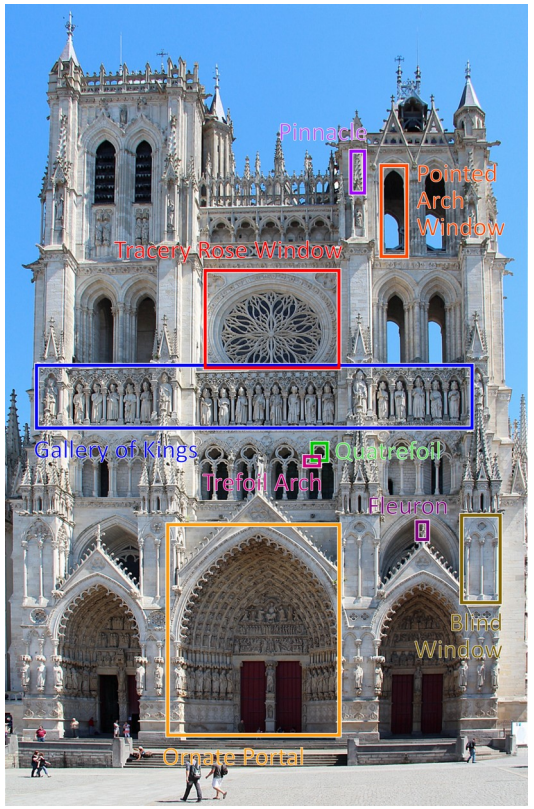
Reported accuracy, using provided bird regions

The CNN stays frozen in these comparisons

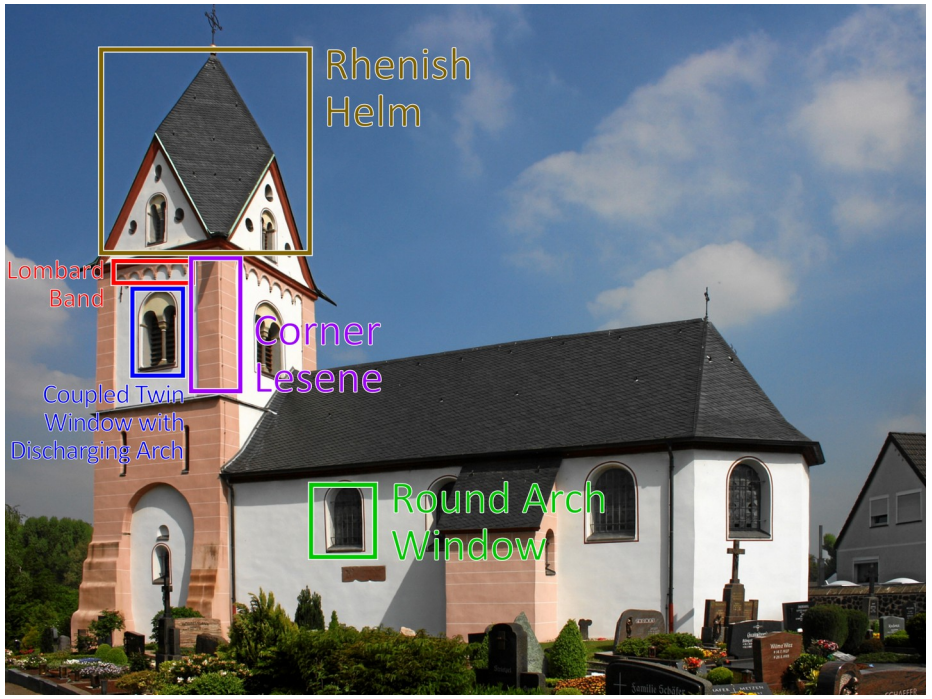
Source: DeCAF

# Transfer depends on the target task

## Gothic




## Romanesque




**ImageNet features are not automatically good for every task.**

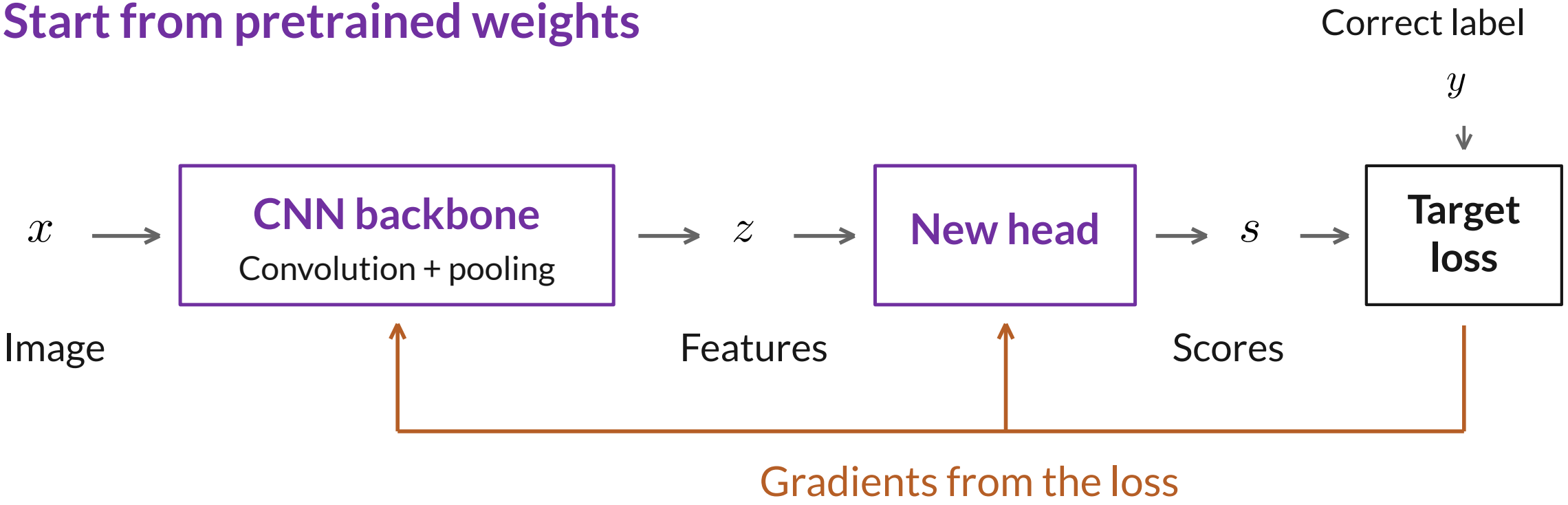
Style classification uses details of windows and arches.

The representation may need to adapt.

Source: WikiChurches; photos: Jean-Pol Grandmont (CC BY 3.0), Beckstet (CC BY-SA 3.0)

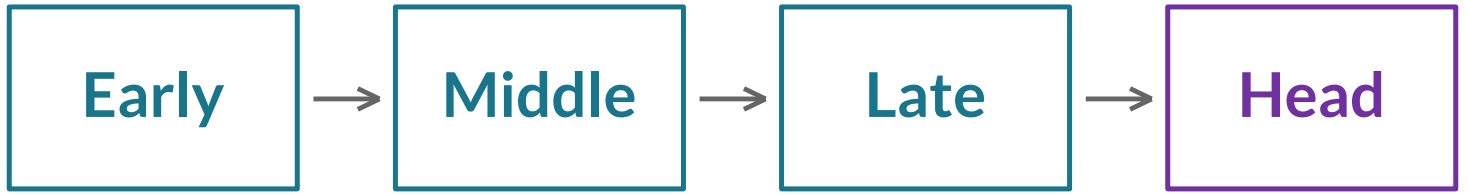
# Finetuning

**Start from pretrained weights**




Train the backbone as well as the new head

# Partial and full finetuning




Head only

**Partial finetuning**

**Full finetuning**

Layers that receive updates

# Choosing layers to update

### If a linear probe works well

Keep the backbone frozen and train only the head.

### If validation accuracy is low

Try partial or full finetuning.

### If training cost is a concern

Freeze early layers to save computation and memory.

# Freezing parameters in PyTorch

Head-only training: keep the backbone fixed

```
backbone.requires_grad_(False)  # freeze backbone weights
backbone.eval()                 # keep BatchNorm state fixed
head.train()

optimizer = torch.optim.AdamW(head.parameters(), lr=head_lr)
```

Freezing weights does not freeze BatchNorm statistics

# Learning rates for finetuning

**Pretrained backbone**

**Small LR**

Small updates help preserve useful pretrained features.

**New prediction head**

**May use a higher LR**

New weights must learn the target task.

Separate rates are optional. A shared small LR also works.

# Different learning rates in PyTorch

Optional: separate backbone and head learning rates

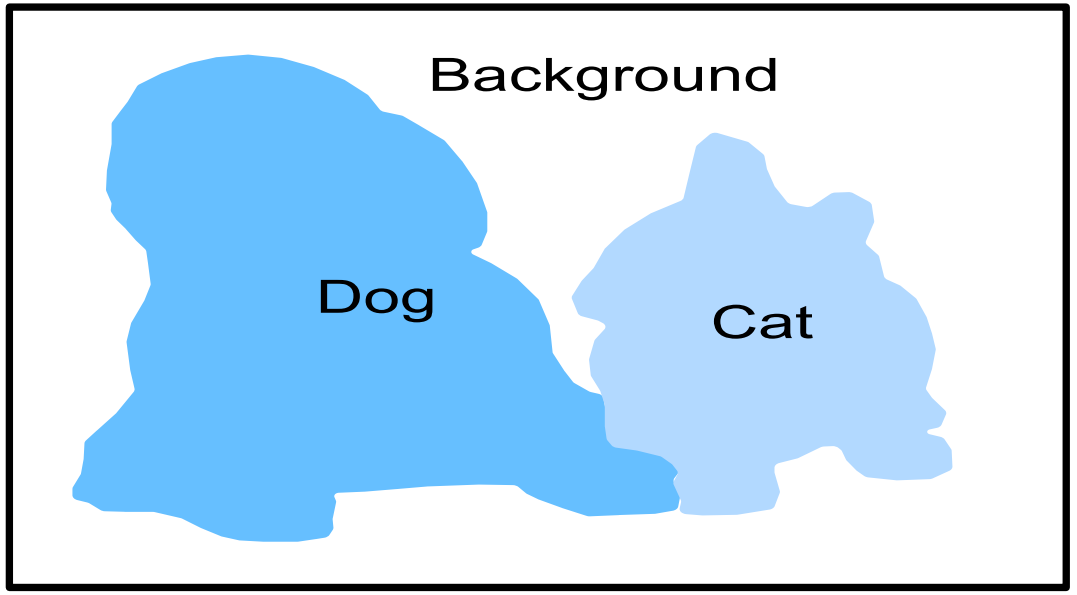
```
backbone.requires_grad_(True)  # enable backbone updates

optimizer = torch.optim.AdamW([
    {"params": backbone.parameters(), "lr": backbone_lr},
    {"params": head.parameters(),     "lr": head_lr},
])
```

One optimizer step applies both learning rates

# Finetuning workflow

1. Load pretrained weights and use their input preprocessing
2. Create a classifier for the new labels
3. Choose which layers to train
4. Train on the new data with a small backbone LR
5. Evaluate on held-out images from the new task

# Next: semantic segmentation




- Predict a category at every pixel
- Use spatial feature maps for dense outputs
- Train and evaluate semantic segmentation

Source: Dive into Deep Learning