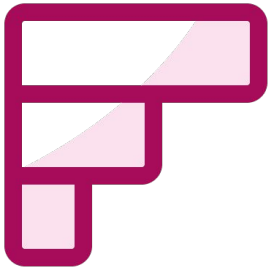
Explainable AI - Lecture 10

Convolutional Neural Networks

norwegian open ai lab
NTNU



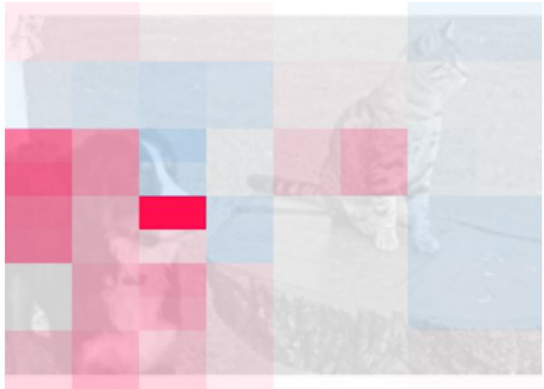
How difficult is this course compared to other courses?

How to explain image model predictions?

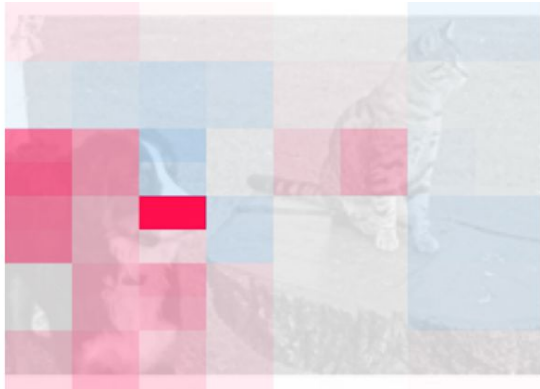
We've used SHAP to highlight important pixel (group)s.

SHAP is (largely) model agnostic.

What's challenging about using SHAP to explain neural network predictions?



How to explain image model predictions?



We've used SHAP to highlight important pixel (group)s.

SHAP is (largely) model agnostic.

What's challenging about using SHAP to explain neural network predictions:

- Deep neural networks have expensive forward passes.
- The input space is usually huge:
  A small image has $3 \times 128 \times 128 \approx 49\text{k}$ "features"
  $\Rightarrow$ We need unreasonable amounts of example images for good results (or great image segmentation).
- SHAP complexity goes as 2^N for N features.
- Model agnostic methods ignore model structure.

Model specific explanations

To exploit model structure when generating explanations, we need model specific explanation methods.

In this lecture, we will investigate how we can interpret and explain convolutional neural networks (CNNs), based on our knowledge of their structure.

Specifically, we can interpret and create explanations based on

- weights and activations (for different layer types),
- gradients for the predicted class.

Crash course in the CNN architecture

CNN map

Preprocessing

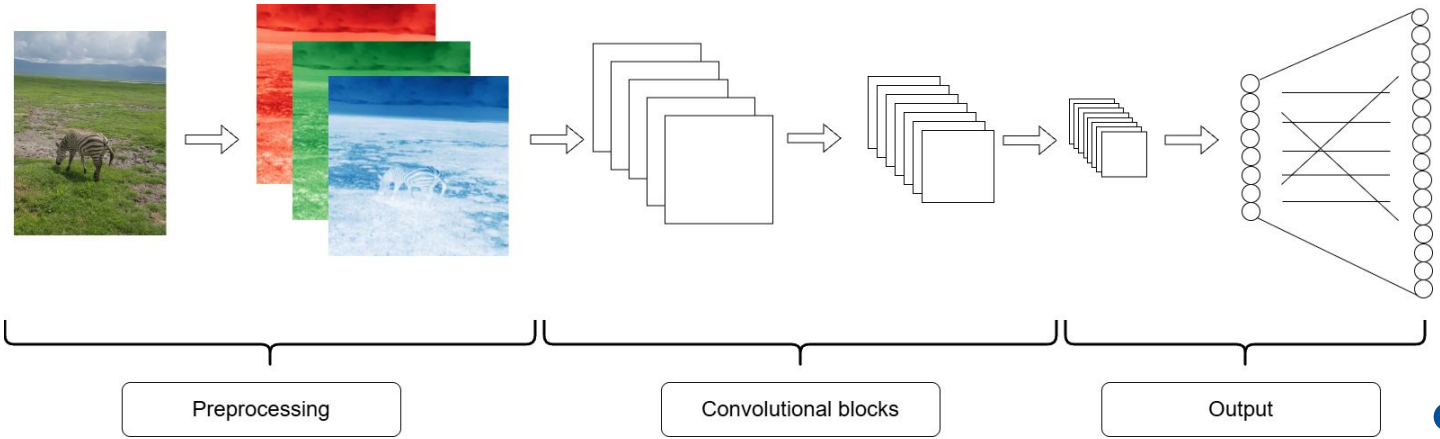
- Normalize pixel values
- Reshape / Crop / Scale

CNN blocks

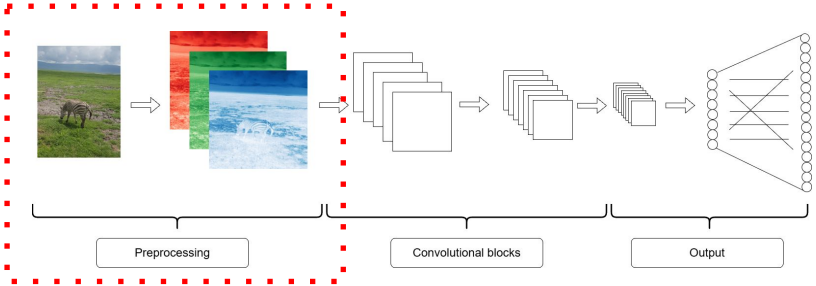
- Convolutions
- ReLU
- Max/Avg Pooling
- Batch norm
- Residual connections

Output

- Global Average Pooling
- Softmax

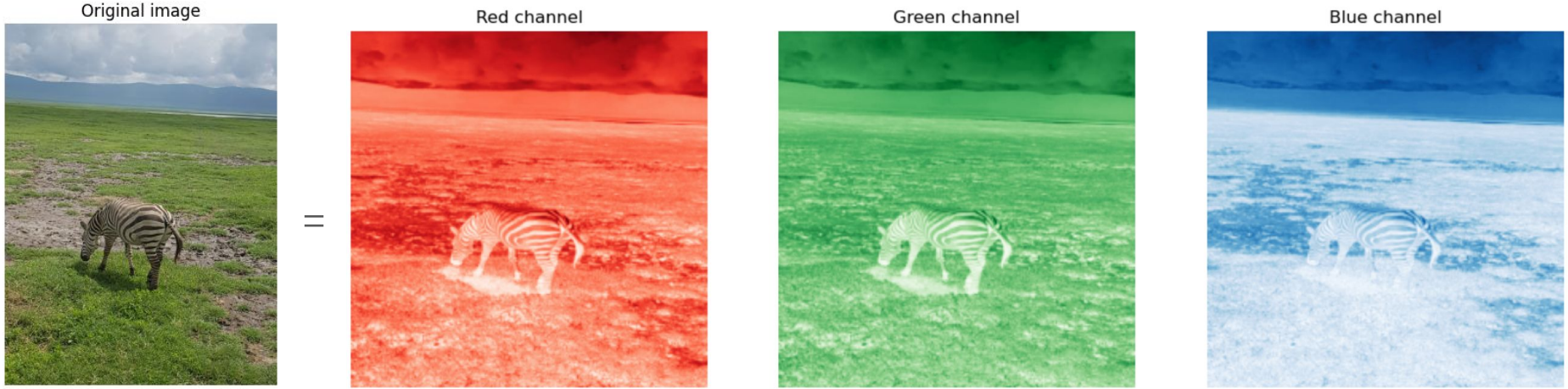


Preprocessing

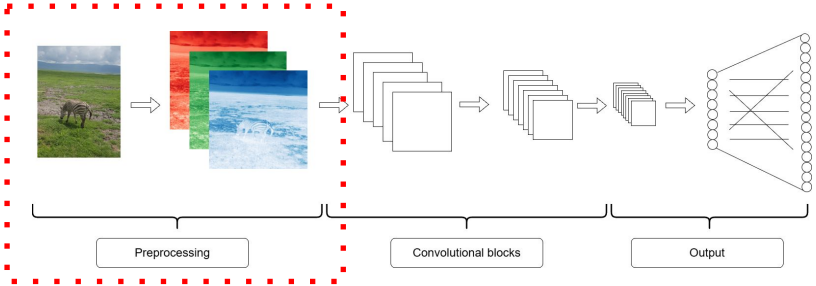


Images have shape (C, H, W).

C = number of channels. Standard images have C=3, for the three colours.
H = height
W = width



Preprocessing



Images have shape (C, H, W).

C = number of channels. Standard images have C=3, for the three colours.
H = height
W = width

CNNs are often trained on images with shape 3×128×128.

Given a set of images, we can reshape, crop and change the pixel values.

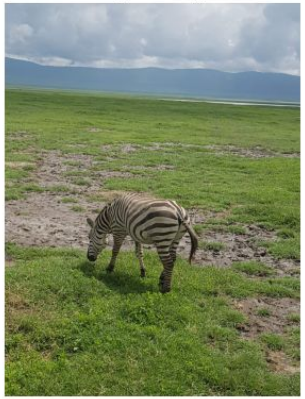
The pixel values usually have to be normalized.

We will not train our own CNNs, but when training, normalization is important for stability and training efficiency.

We will normalize pixel values as our pre-trained CNNs expect from their training distributions.

Preprocessing

Original image



This magnificent photograph was taken by my PhD student Jacob.

You can use your own photo in the notebooks.

```python
# Reshapes, divides pixel values by 255 -> [0,1]
transform_human = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
])

# For model input: resize, divides by 255, then normalize with ImageNet statistics
transform_model = transforms.Compose([
    transforms.Resize((224, 224)),
    transforms.ToTensor(),
    transforms.Normalize(mean=[0.485, 0.456, 0.406], std=[0.229, 0.224, 0.225]),
])

image_human = transform_human(image)
image_model = transform_model(image)

# Min-max normalize to [0,1]. Do not give this to the model! Just for plotting
image_model_normalized = (image_model - image_model.min()) / (image_model.max() - image_model.min())
```

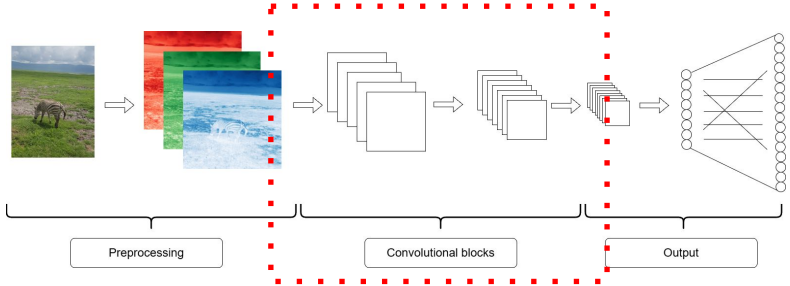
Human-friendly

Model-friendly (Normalized for plotting)

image_model is the image we will give to the model, normalized as the pretrained CNN expects.

image_human is just for plotting.
image_model_normalized is just for plotting.

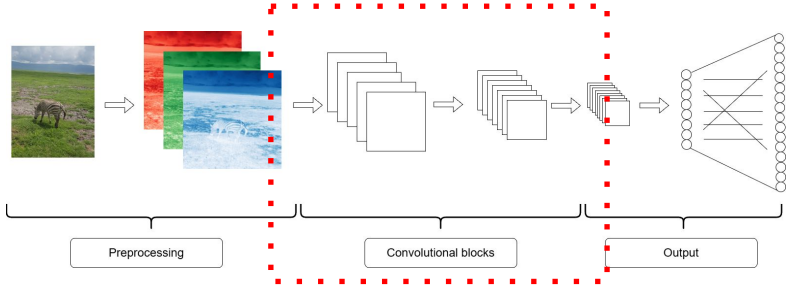
Convolutions



Most importantly for image recognition, we want a feature to be recognized the same *regardless of position*. MLPs can't do this.

What's the CNN solution to this?

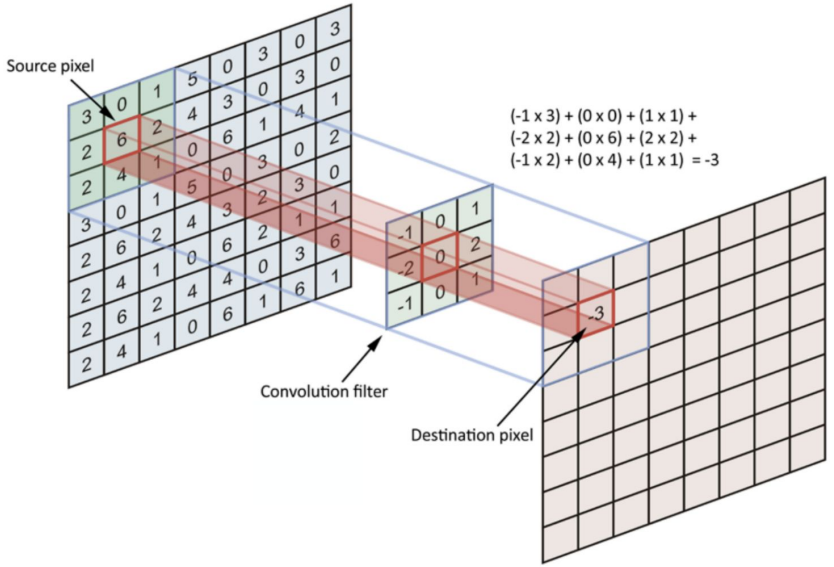
Convolutions



Most importantly for image recognition, we want a feature to be recognized the same *regardless of position*. MLPs can't do this.

The solution is the central building block of CNNs: convolutions.

Convolutions



Convolution steps

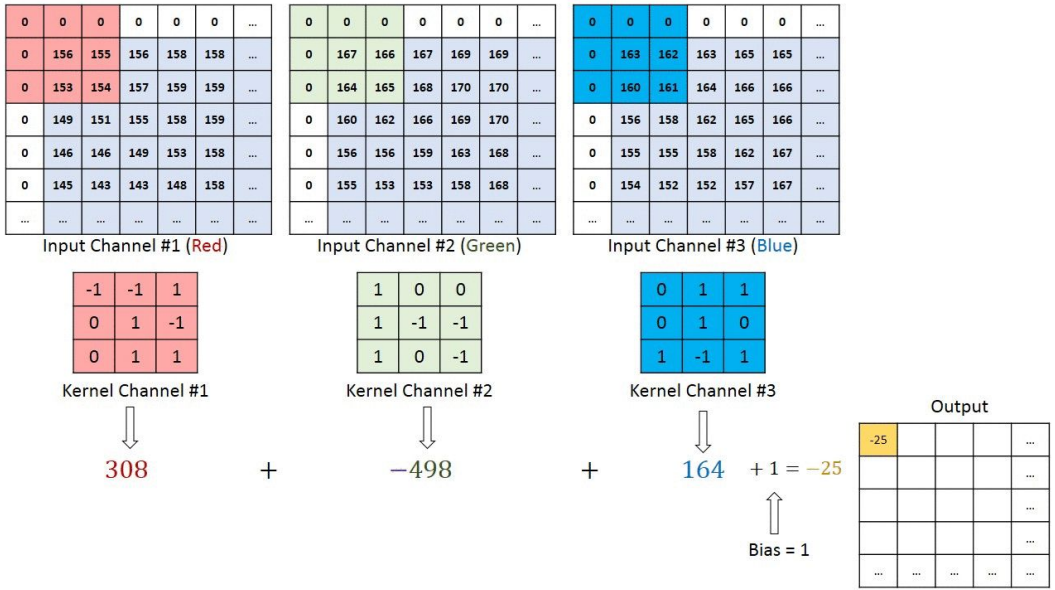
1. Prepare a filter, e.g a 3X3 matrix.
   The weights describe some high-level feature.

2. Place it at a specific position on your image.

3. Take the dot product (*calculate similarity*)
   ⇒ the resulting scalar represents presence of the feature at the position.

4. Repeat for all positions (*slide the filter over the entire image*)
   ⇒ **Feature map** indicating the presence of the high-level feature for all positions.

Convolutions (with multiple channels)



We often use zero-padding on the edge of the image.

Convolutions over the channels are combined.

Convolutions

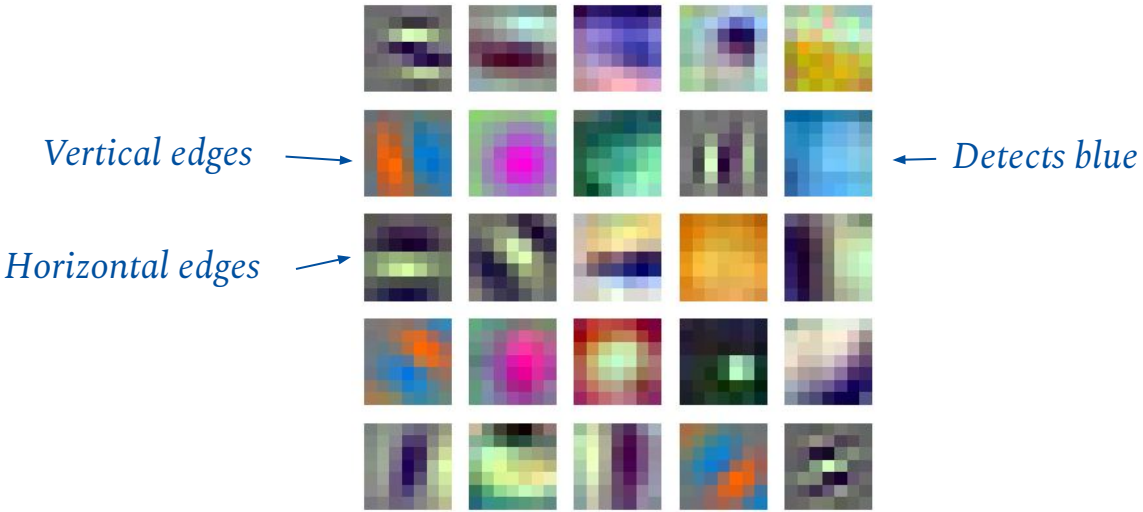
Q: How do we know which high-level features to look for? and which filters achieve exactly this?

Convolutions

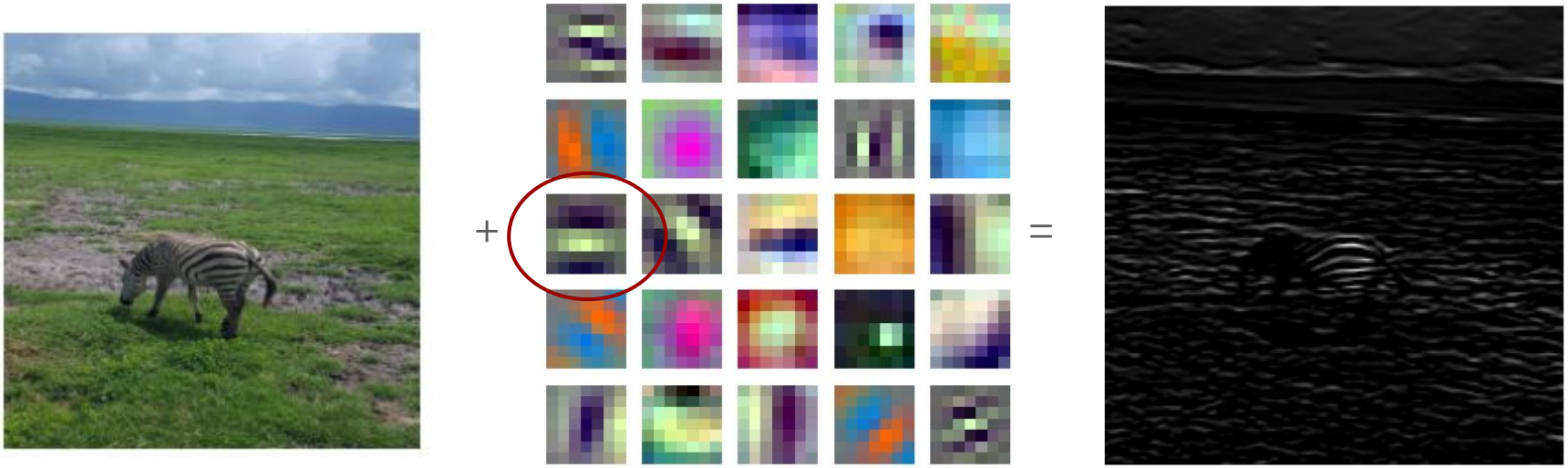
Q: How do we know which high-level features to look for? and which filters achieve exactly this?

A: This is where the *learning* comes in: The CNN filters are learnable parameters.

After a CNN is trained, we can visualize early filters:



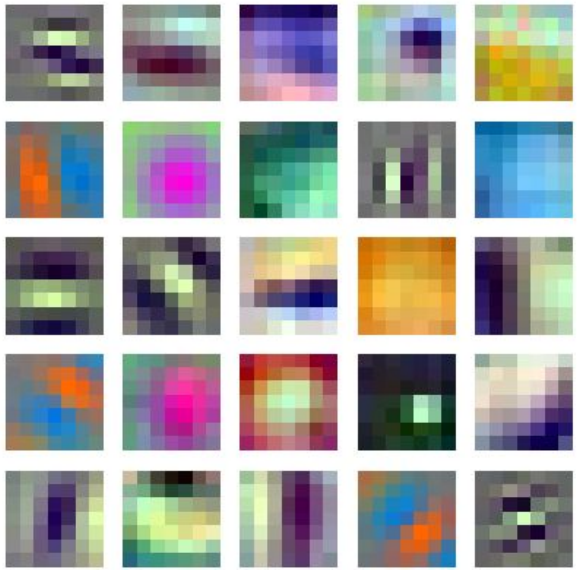
Convolutions: Filters to feature maps



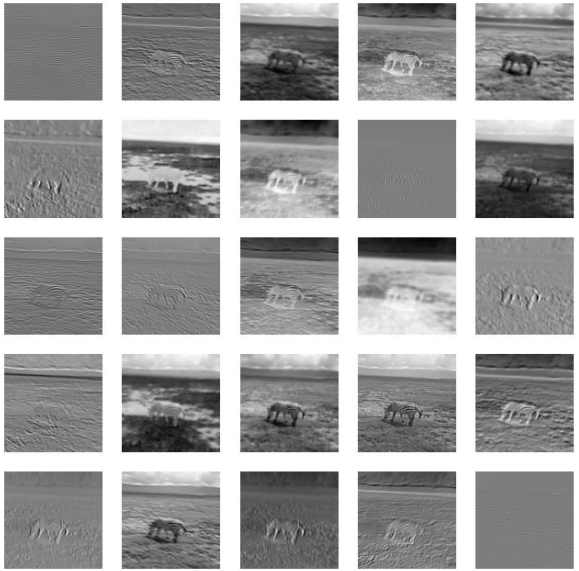
This is just conceptually. We'll see exactly how this is done (with code) in a minute

Convolutions: Filters to feature maps

Filters from the first convolutional layer



Resulting feature maps

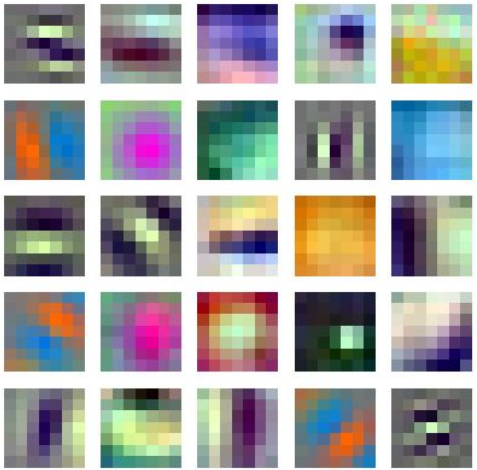


Convolutions

We used one filter on the input image, resulting in one feature map.

Using 100 filters, we could get 100 feature maps that look for unique features.

Each filter learns a single high-level feature, like edges, colors, noise ✅
But not composite high-level features like faces, bone spurs, tumors ❌



Convolutions

We used one filter on the input image, resulting in one feature map.

Using 100 filters, we could get 100 feature maps that look for unique features.

Each filter learns a single high-level feature, like edges, colors, noise ✅
But not composite high-level features like faces, bone spurs, tumors ❌

Using deep learning (multiple layers), the model can learn additional filters on top of the generated feature maps.

Then, hopefully:

- Early feature maps identify simple high-level features like **edges, colors, …**
- Later feature maps identify composite high-level features like **eyes, wheels, …**
- Final feature maps identify even more composite high-level features like **humans, cars…** which are linear with the classification/regression task.

Convolutional blocks

Convolution is *not* all you need :-/

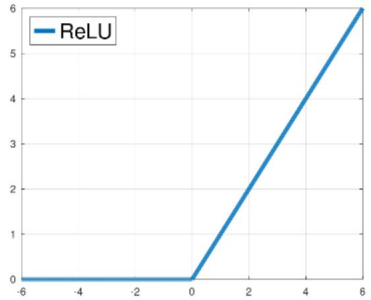
In addition, modern CNNs have:

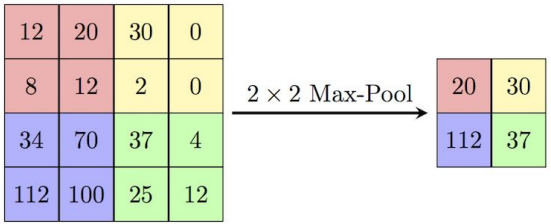
Non-linearity: apply ReLU activation to feature maps.
⇒ this breaks the linearity of stacking convolutions.

Pooling: Reduces spatial size of feature maps.
⇒ less computationally expensive, helps with generalization.

Batch normalization: Normalizes feature maps.
⇒ mainly beneficial for stochastic gradient descent (training).

Residual connections: Prevents vanishing/exploding gradients.





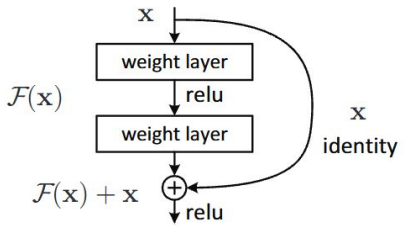


Figure 2. Residual learning: a building block.

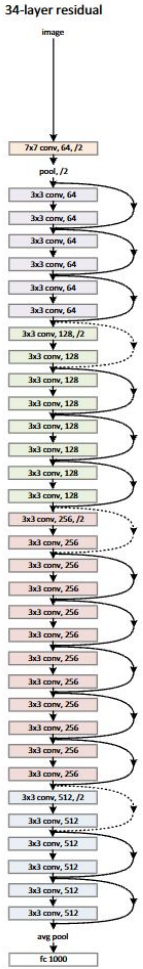
Convolutional blocks: ResNet

```
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model = models.resnet18(weights=models.ResNet18_Weights.IMAGENET1K_V1)
model.to(device)
model.eval()

print(model)
```

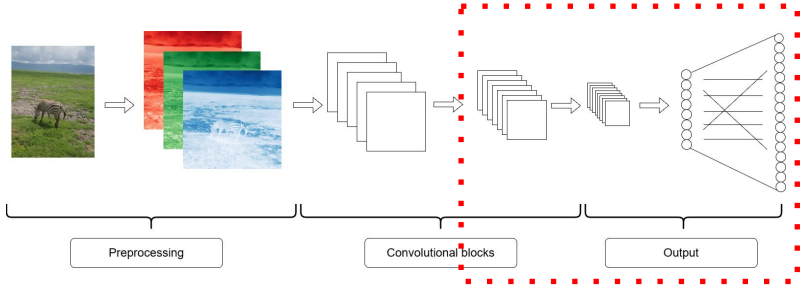
```
ResNet(
  (conv1): Conv2d(3, 64, kernel_size=(7, 7), stride=(2, 2), padding=(3, 3), bias=False)
  (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (relu): ReLU(inplace=True)
  (maxpool): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (layer1): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
    (1): BasicBlock(
      (conv1): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn1): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
      (relu): ReLU(inplace=True)
      (conv2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
      (bn2): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    )
  )
  (layer2): Sequential(
    (0): BasicBlock(
      (conv1): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1), bias=False)
```

We'll use the pretrained ResNet model with 18 trainable layers.

(17 regular conv layers and one fully connected layer at the end)



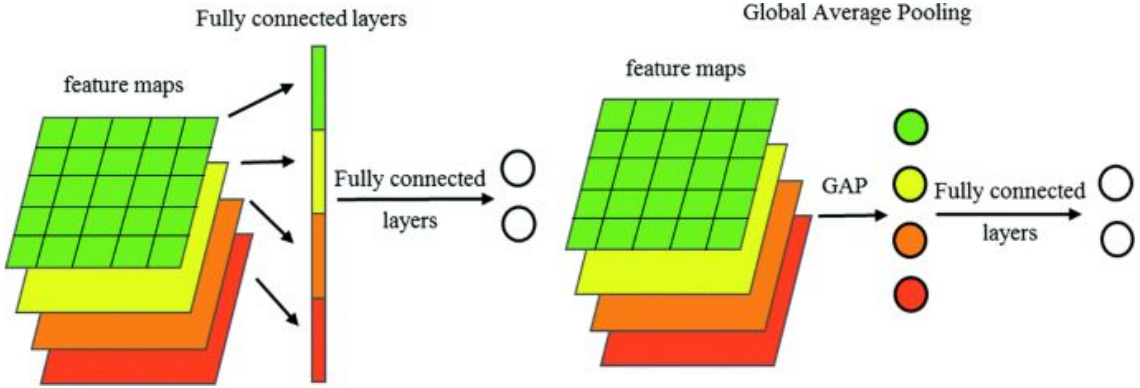
Output



In later layers, we expect highly abstract and potentially informative feature maps.

Some (usually old) CNNs flatten the final activation maps, followed by an FC layer and softmax.

Today, most use Global Average Pooling (GAP), followed by an FC layer with softmax activation.



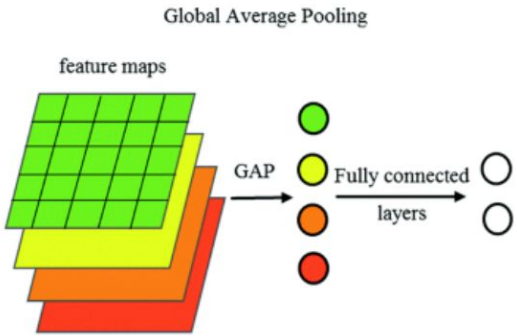
Flattening (old approach)

Average per feature map (current approach)

Output

Why GAP? (at this point, this looks like an implementation detail, but it will be important for an explanation method)

1. Fewer parameters. Say 500 feature maps, 10×10 spatial size, and 1000 output classes (500×10×10×1000) = 50 million parameters! Applying GAP to the same feature maps yields one value per feature map, 500 in total. The FC layer maps this to 1000 classes =500k parameters.
2. Helps with generalization: A feature's specific position doesn't matter.
3. Invariance to input size. If our final feature map has size 500x12x15, GAP still produces 500 values. One per feature map.
4. Bonus: XAI methods like CAM use this (stay tuned).



You have completed the crash course and now know CNNs. ezpz.

Preprocessing

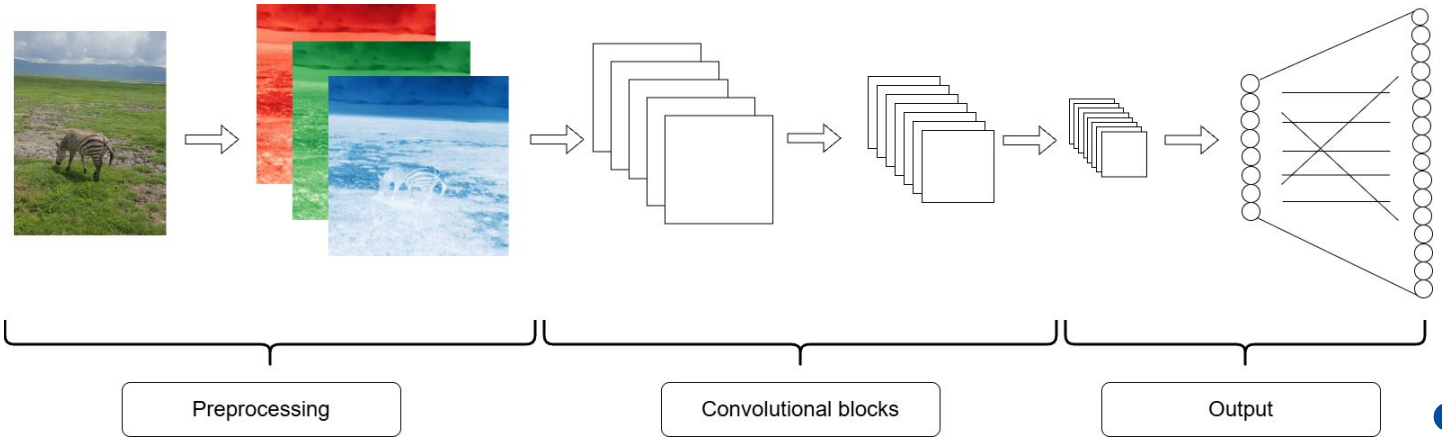
- Normalize pixel values
- Reshape / Crop / Scale

CNN blocks

- Convolutions
- ReLU
- Max/Avg Pooling
- Batch norm
- Residual connections

Output

- Global Average Pooling
- Softmax



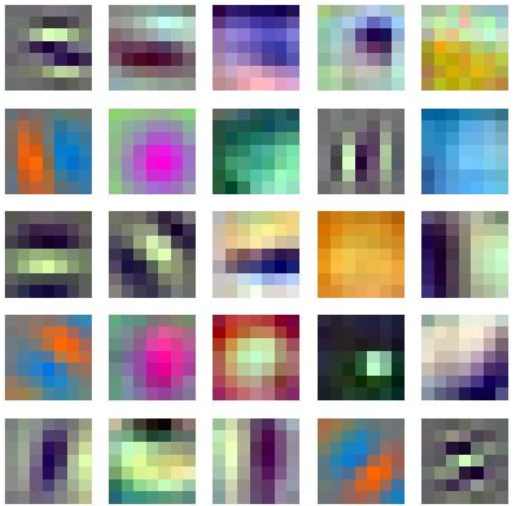
Interpreting CNNs

Interpretability: Filter weights

We have super capable model like ResNet, and would like to understand its representations and predictions.

Possible approach: look at the learned filters (like before. Here's the code.)

```
fig, axs = plt.subplots(5, 5, figsize=(5, 5))
for i in range(25):
    # Extract filter weights in the first layer
    weights = model.conv1.weight[i].cpu().detach().numpy().transpose(1, 2, 0)
    # Normalize to [0,1]
    weights = (weights - weights.min()) / (weights.max() - weights.min())
    axs[i // 5, i % 5].imshow(weights)
    axs[i // 5, i % 5].axis("off")
plt.show()
```



Interpretability: Feature maps

After the first convolutional layer, we can interpret the corresponding filter as a feature map using the input image (like before. Here's the code)

```python
# Extract filter #11 (first in third row)
filter_weights = model.conv1.weight[10].cpu().detach().unsqueeze(0) # shape: (1, 3, 7, 7)

# Add batch dimension to the image: (1, 3, 224, 224)
image_batch = image_model.unsqueeze(0)

# Apply convolution and ReLU
filtered_image = F.relu(F.conv2d(image_batch, filter_weights))
filtered_image = filtered_image.squeeze(0).squeeze(0)

plt.imshow(filtered_image, cmap="gray")
plt.axis("off")
plt.show()
```

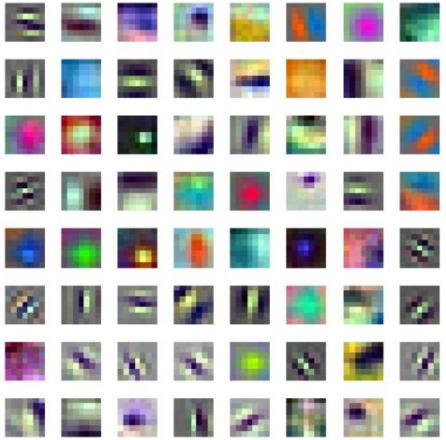
Interpretability: Filter weights

What happens next?

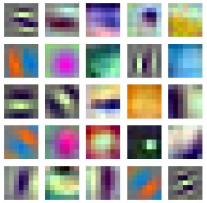
Our first convolutional layer has 64 filters, producing 64 feature maps (*we just looked at 25 of them, here are all 64*).

Think of these feature maps as channels in an internal representation of the image.

⇒ **The next convolutional layer does convolutions on "internal images" with 64 channels!**



Interpretability: Filter weights



What happens next?

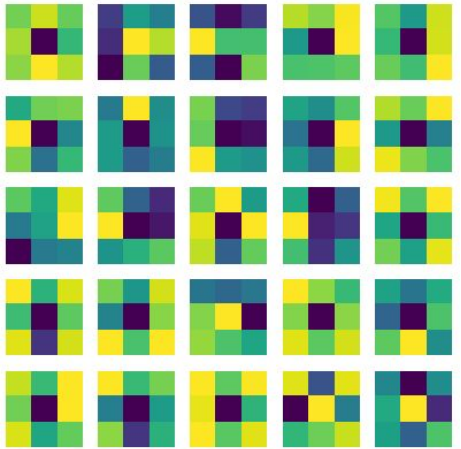
Our first convolutional layer has 64 filters, producing 64 feature maps (*we just looked at 25 of them, here are all 64*).

⇒ The next convolutional layer does convolutions on "internal images" with 64 channels!

How do we interpret a filter that looks at 64 different features?
('features' being a linear combinations of "20 degrees blue horizontal line", "smoothed yellowness", "not red"...× 64)

For plotting, we can average the 64 channel weights, producing one value per channel for the 3×3 filter.
(*"Why 3×3?" See slide 13.*)

How would you interpret these filters?



Filters of the second convolutional layer

Interpretability: Filter weights

What happens next?

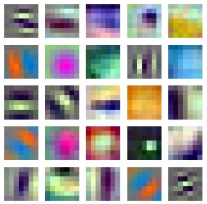
Our first convolutional layer has 64 filters, producing 64 feature maps (*we just looked at 25 of them, here are all 64*)

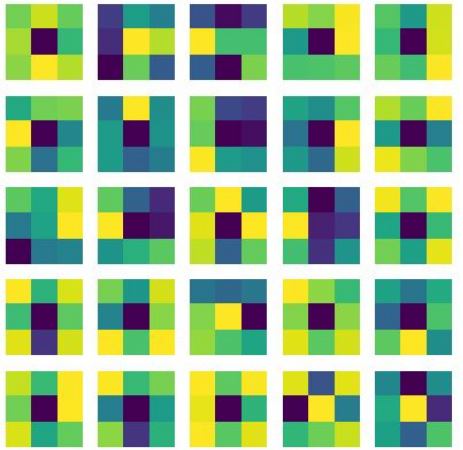
⇒ The next convolutional layer does convolutions on "internal images" with 64 channels!

How do we interpret a filter that looks at 64 different features?
('features' being a linear combinations of "20 degrees blue horizontal line", "smoothed yellowness", "not red"...× 64)

For plotting, we can average the 64 channel weights, producing one value per channel for the 3×3 filter.

We can't really interpret the filters, i.e. model weights, beyond the first convolutional layer.





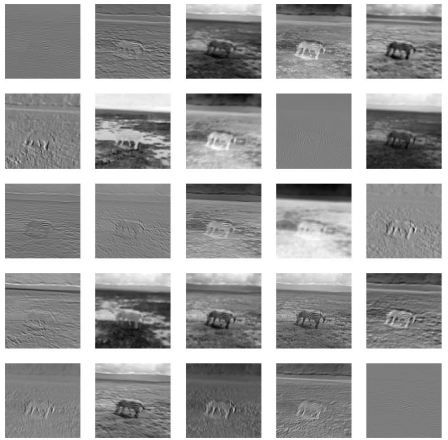
Filters of the second convolutional layer

Interpretability: Feature maps

Staring at weights is probably a lost cause. But what about feature maps?

For the first layer, we saw that the feature maps tell the same story as filter weights projected onto the input image (*"feature maps = image + filters"*).

Can we visualize feature maps in deeper layers?



Interpretability: Feature maps

```python
output_feature_maps = {}
def forward_hook(module, input, output):
    output_feature_maps[module] = output

# Add hooks to conv layers
hooks = []
for layer in model.modules():
    if isinstance(layer, torch.nn.Conv2d): # Ignoring BatchNorm, Pooling, FC layer, ReLU
        hooks.append(layer.register_forward_hook(forward_hook))

# Forward the image to collect feature maps
with torch.no_grad():
    output = model(image_batch.to(device))

for hook in hooks:
    hook.remove()
```

```python
# Plot 5 feature maps for each layer
for i, (layer, feature_map) in enumerate(output_feature_maps.items()):
    fig, axs = plt.subplots(1, 5, figsize=(10, 5))
    for j in range(5):
        feature_map_j = F.relu(feature_map[0][j]).squeeze().cpu().detach().numpy()
        axs[j].imshow(feature_map_j, cmap="gray")
        axs[j].axis("off")
        axs[j].set_title(f"Feature map {i}.{j}")
    plt.show()
```

We don't need the hook if the model already returns the weights (output_hidden_states=True, or like we did for the transformer models with output_attentions=True). For internal layers that just pass information and don't return it, we need hooks.

Make a dictionary for all feature maps

For all 2D convolutional layers, get activations…

… from a forward pass on our image.

Plot five feature maps for each layer (this is post ReLU)

You can look at all of these in the notebook, but we'll focus on a few of them in this lecture.

Interpretability: Feature maps

What we saw for the first filters on the input image:

Fourth convolution layer. Possible edge detectors?

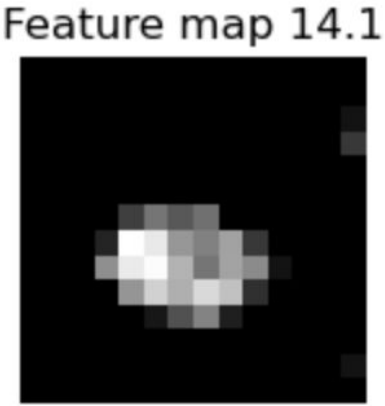
Fourteenth convolution layer. Zebra detector?

Is 14.1 a zebra detector?

Interpretability

Hypothesis: filter 14.1 detects zebras.

Q: *How can we test this hypothesis?*



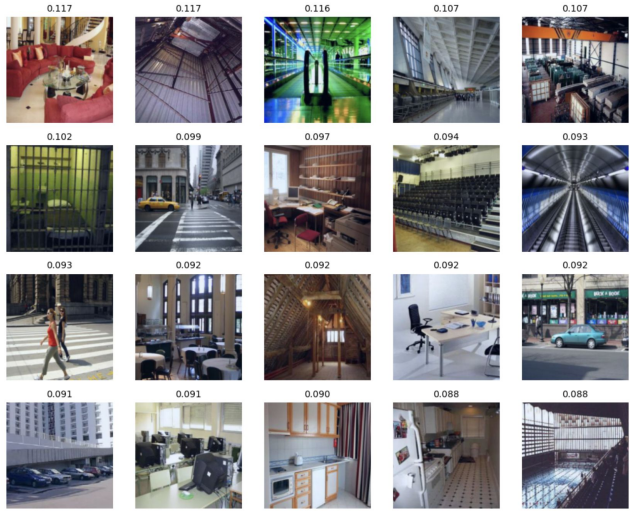
Interpretability: Prototypical examples

Hypothesis: filter 14.1 detects zebras.

Q: How can we test this hypothesis?
A: By using **prototypical examples**.

Procedure:

1. Forward *N* images and collect the *N* feature maps of filter 14.1.
2. Calculate average/sum over the feature maps, over all positions.
3. Sort the *N* images on the average activation.
4. Plot the top-*k* most activating images
   (*highest sum/average on the feature map of interest*).



What do we see?

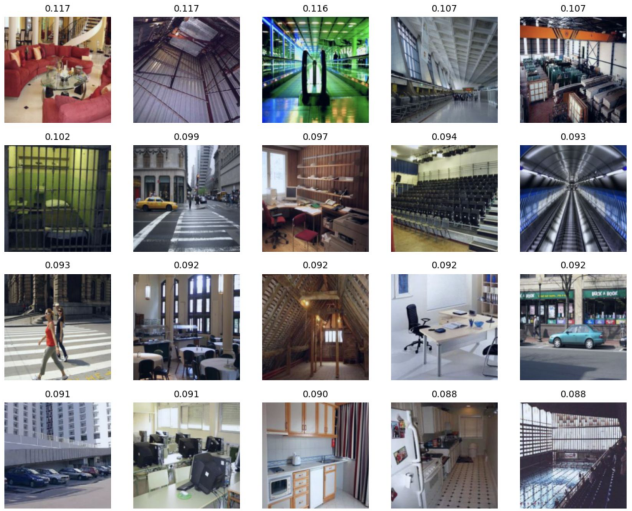
Interpretability: Prototypical examples

Hypothesis: filter 14.1 detects zebras.

Q: How can we test this hypothesis?
A: By using **prototypical examples**.

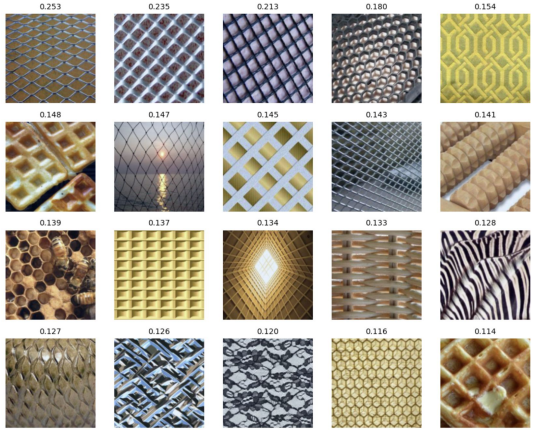
Procedure:

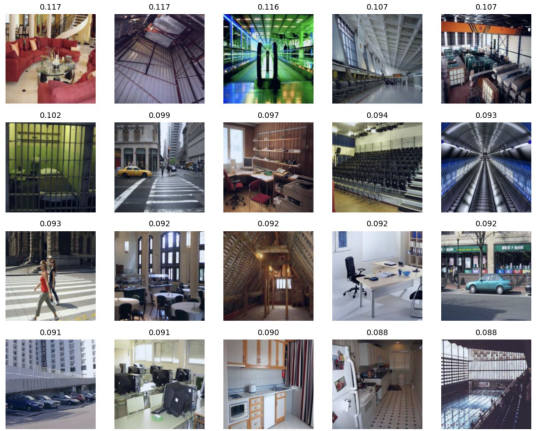
1. Forward *N* images and collect the *N* feature maps of filter 14.1.
2. Calculate average/sum over the feature maps, over all positions.
3. Sort the *N* images on the average activation.
4. Plot the top-*k* most activating images
   (*highest sum/average on the feature map of interest*).

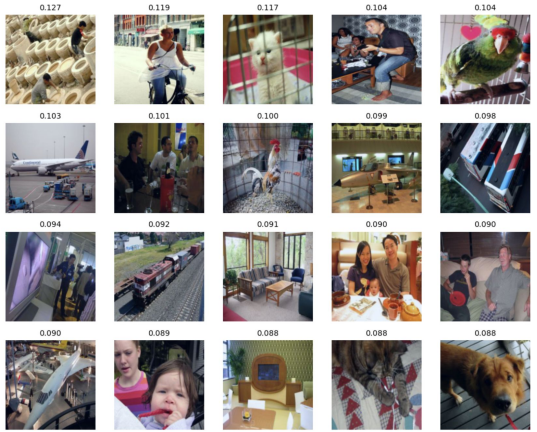


... filter 14.1 seems to activate for stripes and furniture or something.

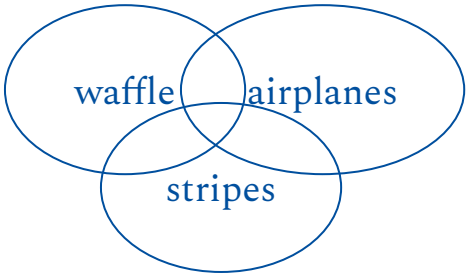
Varying the dataset used to find the top-*k* activating images for filter 14.1 yields different results:







filter 14.1:



This is not really enough to conclude.

Also, if you want to reproduce these experiments, you have to download a lot of images. This is not mandatory!

Interpretability: Activation maximization

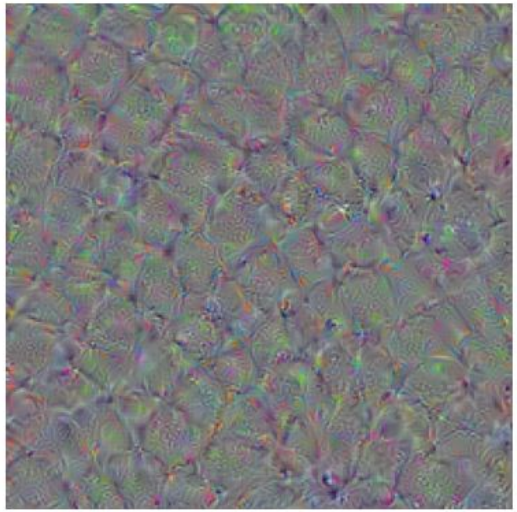
Let's try another approach. For one filter that we choose:

1. Set y = mean filter activation.
2. Initialize random image x (noise or whatever).
3. Use gradient ascent to optimize x such that y is maximized.

Interpretability: Activation maximization

Let's try another approach. For one filter that we choose:

1. Set y = mean filter activation.
2. Initialize random image x (noise or whatever).
3. Use gradient ascent to optimize x such that y is maximized.



(find code in the notebook.)

What do we see in this optimized version of x?

Perhaps we can convince ourselves that there are tiles here...

Interpretability: Activation maximization

Our naïve approach results in extremely noisy images, because we are not restricting x to resemble naturally occurring images.

The trick is to apply regularizations, in just the right amount. **Lucent** does exactly this:

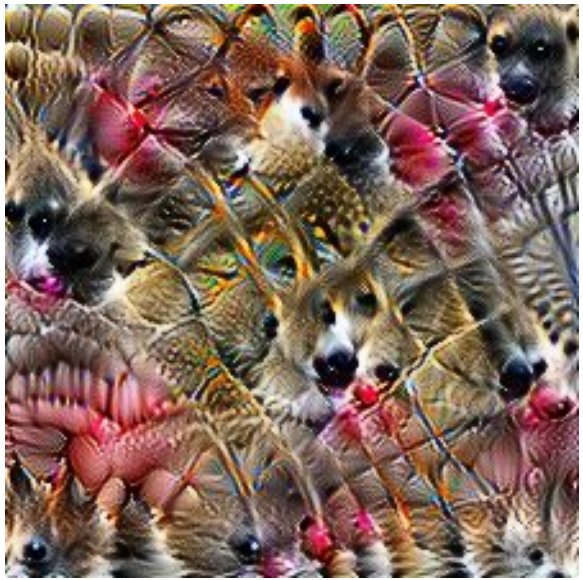
```
# !pip install torch-lucent

# Activation maximization with lucent
import lucent.optvis.objectives as objectives
import lucent.optvis.param as param
import lucent.optvis.render as render

size = 224
objective = objectives.channel(layer="layer3_1_conv2", n_channel=filter_idx) # Must reference layer by name
param_f = lambda: param.image(size)

_ = render.render_vis(model.to(device), objective, param_f=param_f,
                      thresholds=(512,), fixed_image_size=(size, size),
                      show_image=True)[0]
```

Interpretability: Activation maximization

Our naïve approach results in extremely noisy images, because we are not restricting X to resemble naturally occurring images.

The trick is to apply regularizations, in just the right amount. **Lucent** does exactly this.

Run, aaaand:



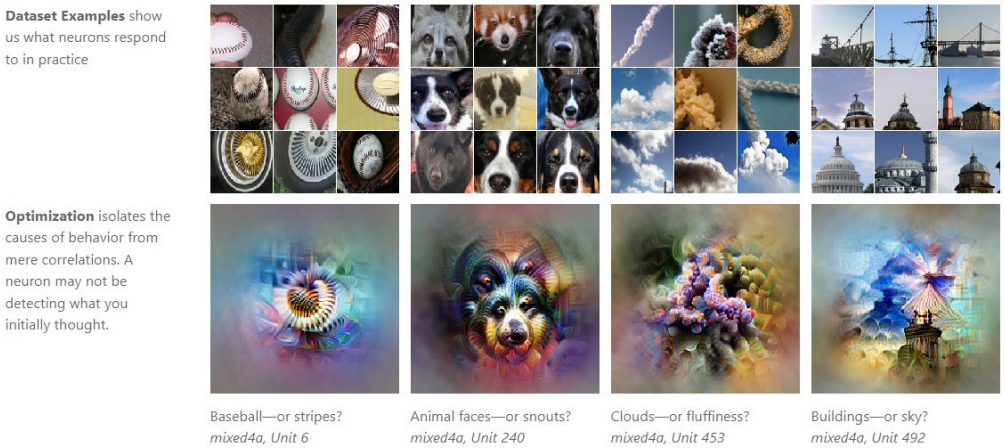
Tiles and animal faces?! What's going on...

Interpretability

We did not manage to make much sense of our filters.

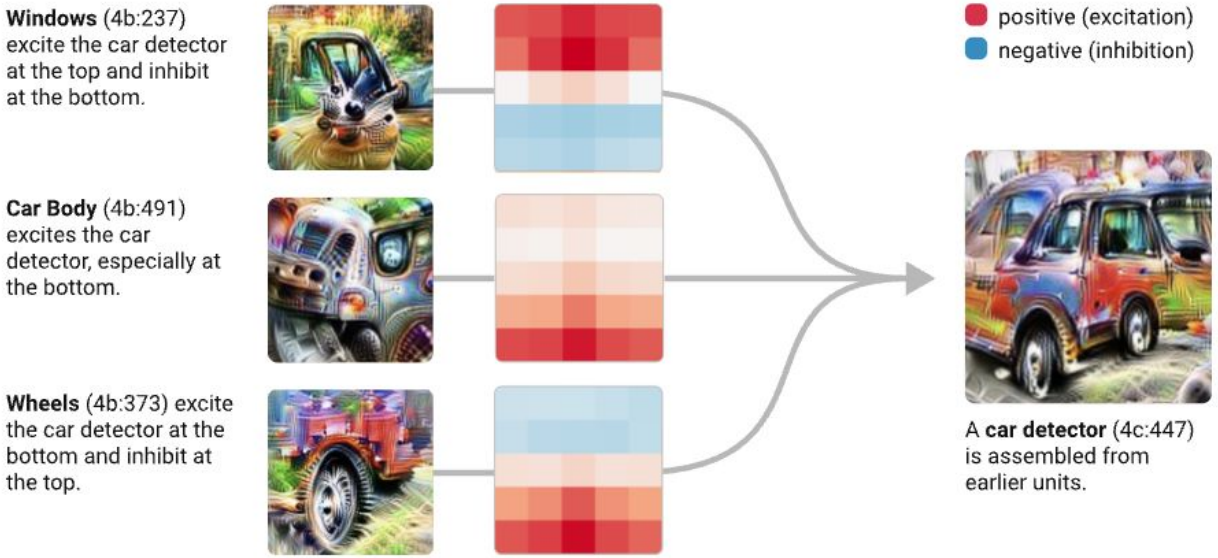
However, interpretability researchers have used similar methods to every filter (which they call 'neuron') in GoogLeNet: https://distill.pub/2017/feature-visualization/

They found much more interpretable filters, and their visualizations are fun to play with.



Interpretability

The hope is: if we know what the filters represent, we can investigate how they interact. AM helps us visualise what triggers the filters. The filters in the next layer pick out and combine information from the previous feature maps.

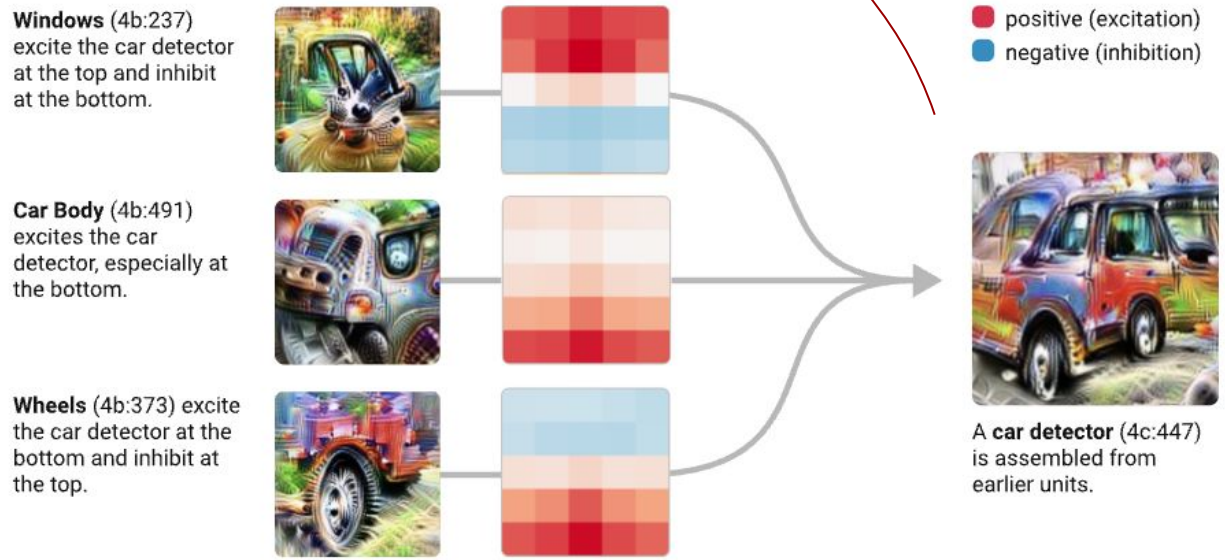


```
(bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu): ReLU(inplace=True)
(conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

Three of these *(that enter the conv2 layer)*

Combined with the layer's weights

Produce one of these



```
(bn1): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
(relu): ReLU(inplace=True)
(conv2): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1), bias=False)
(bn2): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
```

Three of these
(that enter the
conv2 la

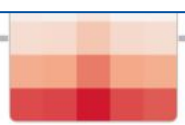
Combined with the

Produce one of these

This is an interpretation, not an explanation. Lightweight mechint.

Windows
excite the
at the top
at the bot

Car Body
excites th
detector, especially at
the bottom.



Wheels (4b:373) excite the car detector at the bottom and inhibit at the top.

A **car detector** (4c:447) is assembled from earlier units.

norwegian open ai lab NTNU

Interpretability: Polysemanticity

Still, the challenge from earlier remains:

Most filters activate to multiple (seemingly unrelated) high-level features.

This is referred to as **polysemanticity**: a single learned feature represents multiple distinct semantic concepts at once.

Optimization with diversity show cats, foxes, but also cars. *Layer mixed4e, Unit 55*

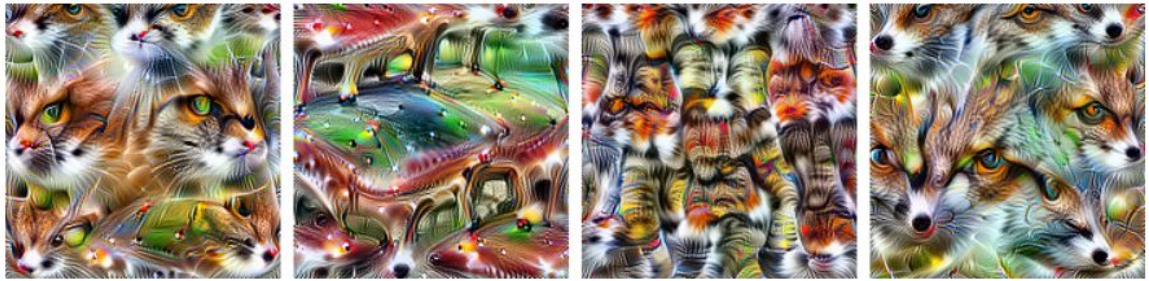
Dataset examples
(prototypicals)

Interpretability: Polysemanticity

If each filter activates to a distinct feature, we can make linear interpretation statements:
Fur + whiskers + pointy ears = cat. ← *Yay, easy :)*

Polysemanticity is a challenge because when filters activate to multiple features, we don't know which features interact:
(Fox + football field) + (car + green) + fur ? + (eyes + pointy snout) = Cat + Fox + Cars? ← *not easy :/*

Although we manage to interpret the filters, the combined interpretation is an entangled mess.



Optimization with diversity show cats, foxes, but also cars. *Layer mixed4e, Unit 55*

Dataset examples

Interpretability: Summary

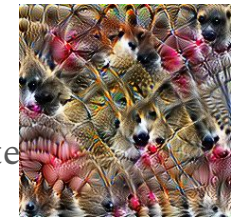
We have learned that…

1. Only the first convolutional layer have interpretable filters.

2. We can get an idea of what a filter represents by:
   a. looking at **feature maps** (in the first layer),
   b. finding highly activating images by using **prototypical examples**,
   c. generating synthetic images that **maximize activation**.

3. By doing (2), we can find monosemantic filters.

4. Unfortunately, most neurons are entangled and polysemantic, making the total inte
   difficult.



Explaining CNNs with feature attributions

Gradient based explanations

Moving on from interpretation methods, we'll spend the rest of the lecture on explanation methods.

Specifically, we will look at **input feature attribution methods** based on **gradients**.

Gradients are a natural candidate for explanations.

Consider the function $f(x, y) = 5x + y^2$

The gradient $(df/dx, df/dy)$ at $(x, y) = (1, 10)$ is (?, ?)

Gradient based explanations

Moving on from interpretation methods, we'll spend the rest of the lecture on explanation methods.

Specifically, we will look at **input feature attribution methods** based on **gradients**.

Gradients are a natural candidate for explanations.

Consider the function $f(x, y) = 5x + y^2$

The gradient $(df/dx, df/dy)$ at $(x, y) = (1, 10)$ is $(5, 20)$.
$\Rightarrow$ according to the gradient, y is (4 times) more important than x.

Neural networks are also functions of their inputs, *and* the gradients are easily accessible (from backpropagation).

Gradient based explanations

Given our CNN model f, and given an image x of size C×H×W, how can we use gradients to explain the prediction $f(x)$?

Gradients are defined for scalar functions (functions with a single output)
⇒ we need to select a single scalar with respect to which to calculate the gradient.

Gradient based explanations

Given our CNN model f, and given an image x of size C×H×W, how can we use gradients to explain the prediction $f(x)$?

Gradients are defined for scalar functions (functions with a single output)
⇒ we need to select a single scalar with respect to which to calculate the gradient.

We typically choose a specific class.

Further considerations:

1. Which class should we use? Usually, the predicted class (argmax) is used.
2. Should we calculate the gradient for the logit or softmax? The two have different interpretations: Increasing the logit increases activations associated only with the class. Increasing the softmax can be achieved by decreasing activations associated with other classes.

Gradient based explanations

Given our CNN model f, and given an image x of size C×H×W, how can we use gradients to explain the prediction $f(x)$?

The gradient has the same shape as the input (C×H×W).

Can we just plot this gradient to see which inputs are most important?

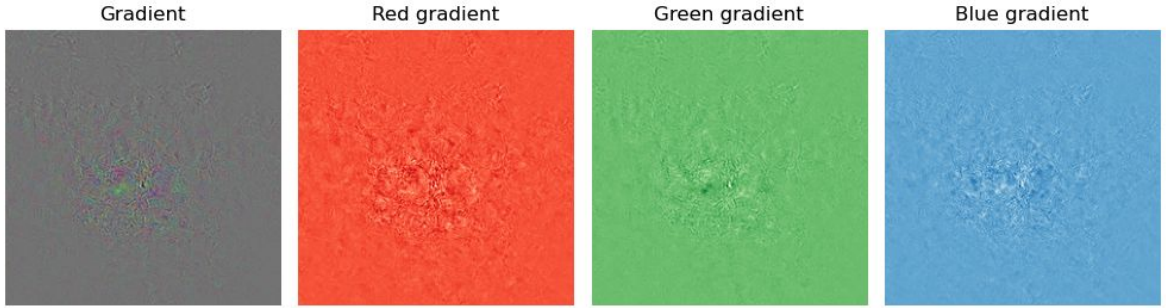
Step 1: get the gradients

```python
input_image = image_model.unsqueeze(0).to(device)
input_image.requires_grad = True

# Forward pass
output = model(input_image) # logits
pred_class = output.argmax(dim=1) # predicted class
score = output[0, pred_class] # largest logit

# Just for fun: whats the probability of the predicted class?
# Class indices https://deeplearning.cms.waikato.ac.nz/user-guide/class-maps/IMAGENET/
print(f"Probability of predicted class: {F.softmax(output, dim=1)[0, pred_class].item():.2%}")
print(f"Predicted class index: {pred_class.item()}") # 340 is zebra. Good job little AI

# Backward pass
model.zero_grad()
score.backward()

# Get gradient wrt. the input image
gradient = input_image.grad.squeeze()
print(f"Image shape: {input_image.shape}")
print(f"Gradient shape: {gradient.shape}")
```

```
Probability of predicted class: 99.97%
Predicted class index: 340
Image shape: torch.Size([1, 3, 224, 224])
Gradient shape: torch.Size([3, 224, 224])
```

Step 2: Plot the gradients for the three channels

```python
# Plot the gradient with RGB, and each channel in a different color
fig, axs = plt.subplots(1, 4, figsize=(10, 5))
gradient = gradient.cpu().permute(1, 2, 0)
gradient = (gradient - gradient.min()) / (gradient.max() - gradient.min())
axs[0].imshow(gradient)
axs[0].axis("off")
axs[0].set_title("Gradient")

Colors = {"Red": "Reds", "Green": "Greens", "Blue": "Blues"}
for i, channel in enumerate(("Red", "Green", "Blue")):
    axs[i+1].imshow(gradient[:,:,i].cpu().detach().numpy(), cmap=Colors[channel])
    axs[i+1].axis("off")
    axs[i+1].set_title(f"{channel} gradient")
plt.tight_layout()
plt.show()
```



... not so informative.

Gradient based explanations

Given our CNN model f, and given an image x of size C×H×W, how can we use gradients to explain the prediction $f(x)$?

The gradient has the same shape as the input (C×H×W)

Plotting the sensitivity (per color channel) is not very informative.

Instead, we can map C×H×W → H×W and visualise which pixels (as single units) are important, by use of a map that can be superimposed on the original image.

This way, our explanation would have the same structure as the data itself.

Gradient based explanations

Given our CNN model f, and given an image x of size C×H×W, how can we use gradients to explain the prediction $f(x)$?

The gradient has the same shape as the input (C×H×W). We can map it to H×W and visualise which pixels (as single units) are important, by use of a map that can be superimposed on the original image.

Common aggregation methods for the mapping are

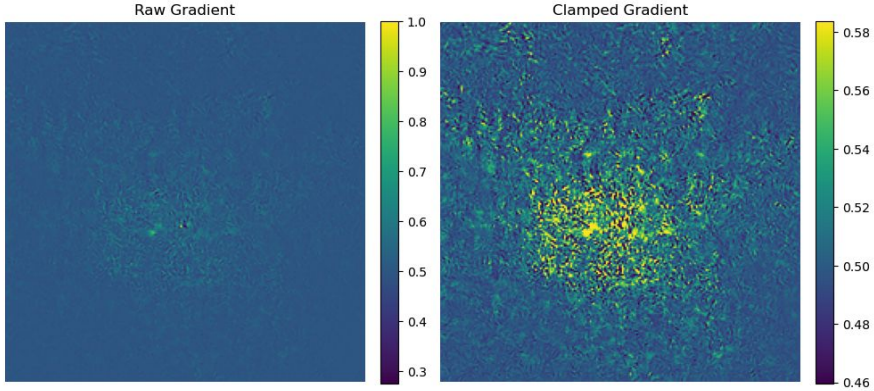
- abs+max (or max+abs)
- abs+sum (or sum+abs)
- keep a quantile of the gradient values (to avoid outliers dominating the scale on the plot)

There is no “correct” solution, people do whatever results in pretty visualizations (aka confirmation bias).

Finally, we choose a pretty colormap and superimpose the 2D mapping of the gradients on the original image.

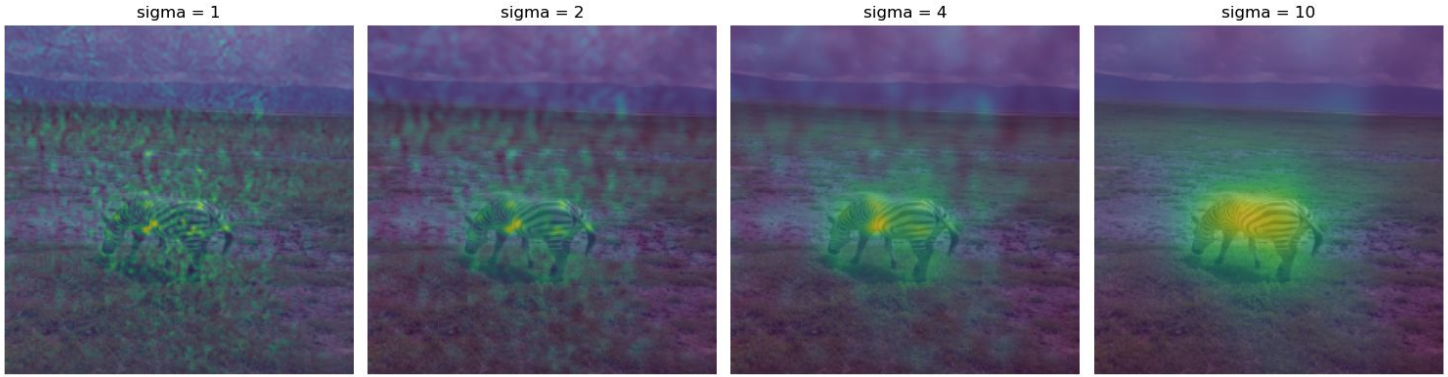
Aggregate gradients and include/exclude outer quantiles

```python
def aggregate_channels(channel_attributions, clamp=True):
    # Multiple options:
    aggregated = channel_attributions.abs().max(dim=-1).values
    #aggregated = channel_attributions.abs().sum(dim=-1)
    #aggregated = channel_attributions.sum(dim=-1).abs()

    # Improves visual nuances
    if clamp:
        return torch.clamp(aggregated, torch.quantile(aggregated.flatten(), 0.01), torch.quantile(aggregated.flatten(), 0.99))
    return aggregated

gradient_aggregated = aggregate_channels(gradient, clamp=False)
gradient_aggregated_clamped = aggregate_channels(gradient, clamp=True)

fig, axs = plt.subplots(1, 2, figsize=(10, 5))
im0 = axs[0].imshow(gradient_aggregated)
axs[0].axis("off")
axs[0].set_title("Raw Gradient")
fig.colorbar(im0, ax=axs[0], fraction=0.046, pad=0.04)

im1 = axs[1].imshow(gradient_aggregated_clamped)
axs[1].axis("off")
axs[1].set_title("Clamped Gradient")
fig.colorbar(im1, ax=axs[1], fraction=0.046, pad=0.04)
plt.tight_layout()
plt.show()
```



Optionally, apply gaussian smoothing and overlay

```
sigmas = (1, 2, 4, 10)
fig, axs = plt.subplots(1, len(sigmas), figsize=(15, 5))
for i, sigma in enumerate(sigmas):
    # Apply gaussian smoothing
    gradient_smoothed = gaussian_filter(gradient_aggregated_clamped, sigma=sigma)

    axs[i].imshow(image_human.permute(1, 2, 0))
    axs[i].imshow(gradient_smoothed, alpha=0.6)
    axs[i].set_title(f"sigma = {sigma}")
    axs[i].axis("off")
plt.tight_layout()
plt.show()
```



Gaussian smoothing: each pixel is replaced by a weighted average of its neighbors. Sigma is the variance, i.e. width, of the gaussian. Wider ⇒ more distant neighbors included ⇒ smoother.

Limitations of gradients

Back to our nice function from earlier: $f(x, y) = 5x + y^2$

Say we want to explain the prediction at $(100, 5)$
The function value is $f(100, 5) = 500 + 25 = 525$
The gradient at this point is $(5, 10)$.

According to the gradient alone, y is twice as important as x.

*Does that make sense? What is the difference between **sensitivity** and **contribution**?*

Limitations of gradients

Back to our nice function from earlier: $f(x, y) = 5x + y^2$

Say we want to explain the prediction at $(100, 5)$
The function value is $f(100, 5) = 500 + 25 = 525$
The gradient at this point is $(5, 10)$.

According to the gradient alone, y is twice as important as x.

This is highlights the difference between **sensitivity** and **contribution**.

Importance can be understood as both **sensitivity** (like the gradient) and **contribution**.
How should we quantify total contribution?

Limitations of gradients

Back to our nice function from earlier: $f(x, y) = 5x + y^2$

Say we want to explain the prediction at $(100, 5)$
The function value is $f(100, 5) = 500 + 25 = 525$
The gradient at this point is $(5, 10)$,

According to the gradient alone, y is twice as important as x.

This is highlights the difference between **sensitivity** and **contribution**.

Importance can be understood as both **sensitivity** (like the gradient) and **contribution**.
How should we quantify total contribution?

Let's look at the **Gradient ⨉ Input** at the point $(100, 5)$:
$(5, 10)$ x $(100, 5) = (500, 50)$

What do you think?

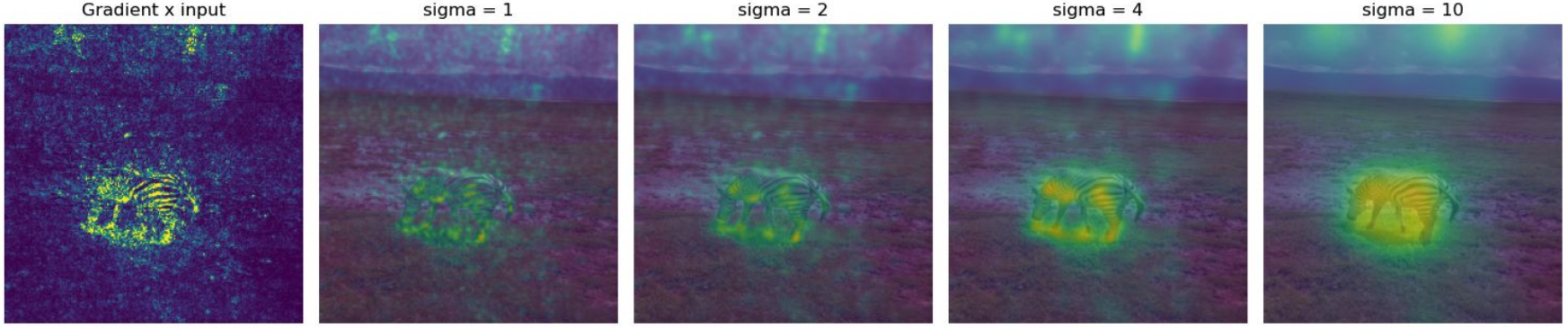
Gradient × Input

Gradient × Input = *G* (element wise product) *x*

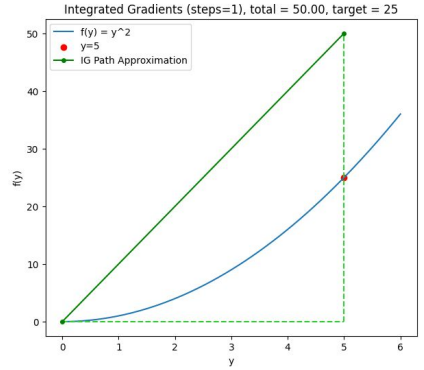
(*and we add the same prettification as before*)

```
# Recompute the gradient
model.zero_grad(set_to_none=True)
logits = model(input_image)
score = logits[0, pred_class]
score.backward()
gradient = input_image.grad

gradient_times_input = gradient * input_image
gradient_times_input = gradient_times_input.squeeze(0).permute(1, 2, 0).detach().cpu()
gradient_times_input = aggregate_channels(gradient_times_input)
```

```
sigmas = (0, 1, 2, 4, 10)
fig, axs = plt.subplots(1, len(sigmas), figsize=(15, 5))
for i, sigma in enumerate(sigmas):
    if i == 0:
        axs[i].imshow(gradient_times_input)
        axs[i].set_title("Gradient x input")
        axs[i].axis("off")
        continue

    # Apply gaussian smoothing
    gradient_smoothed = gaussian_filter(gradient_times_input, sigma=sigma)
    gradient_smoothed = (gradient_smoothed / gradient_smoothed.max())

    axs[i].imshow(image_human.permute(1, 2, 0))
    axs[i].imshow(gradient_smoothed, alpha=0.6)
    axs[i].set_title(f"sigma = {sigma}")
    axs[i].axis("off")
plt.tight_layout()
plt.show()
```



Limitations of Gradient x Input



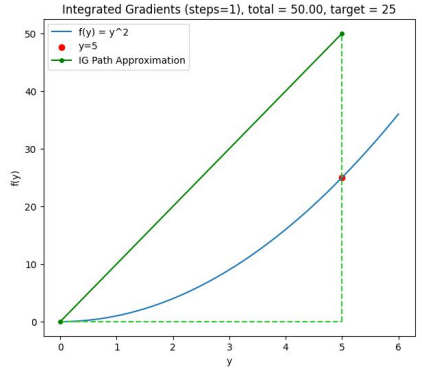
Gradient × Input only assigns **linear contribution.**

Consider again the function $f(x, y) = 5x + y^2$ in a point.
$f(100, 5) = 500 + 25 = 525$ (i.e., the ground truth, knowing the analytical expression, is (500, 25))

Gradient × Input = (5, 10) × (100, 5) = (500, 50)

This aligns with the contribution for x because f is linear with x.
It overestimates the contribution of y because f is non-linear with y.

Limitations of Gradient x Input



Gradient × Input only assigns **linear contribution**

Consider again the function $f(x, y) = 5x + y^2$ in a point
$f(100, 5) = 500 + 25 = 525$ (i.e., the ground truth, knowing the analytical expression, is (500, 25))

Gradient × Input = (5, 10) × (100, 5) = (500, 50)

This aligns with the contribution for x because f is linear with x.
It overestimates the contribution of y because f is non-linear with y.

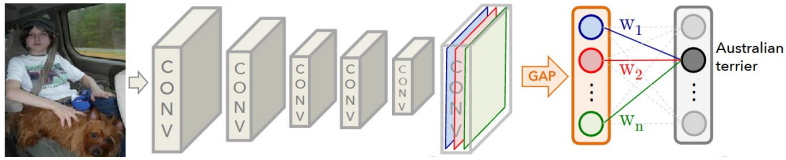
This happens because we found the slope at $f(100, 5)$, and implicitly assume that it is constant everywhere.

CNNs are not linear functions from an image to the outputs. What to do? Our options are:

1. Exploit that the last feature map to output is a linear function
2. Use methods that calculate non-linear attribution

1. the last feature map to output is a linear function

Class activation maps



Remember what most CNNs do:

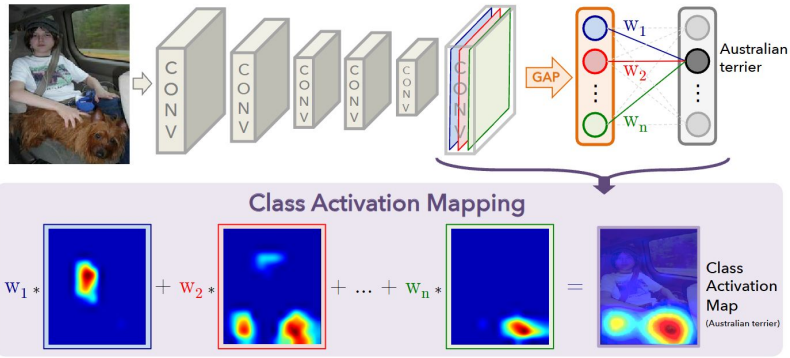
convolutions → feature maps → GAP → FC → logits

GAP turns a feature map into a scalar *z* (*we get one z per feature map*)

A fully connected layer connects each *z* to a class with a single weight *w*.

If *w* is **positive** → *z* contributes positively to the logit
If *w* is **negative** → *z* contributes negatively to the logit
(because of ReLU, z does not take negative values)

Class activation maps



We can combine the *k* feature maps and weights *w* into a representation of

1. How important a high-level feature is (this is *w*), and
2. where on the image this high-level feature is (this is the feature map before GAP).

This is what class activation mapping (CAM) does.

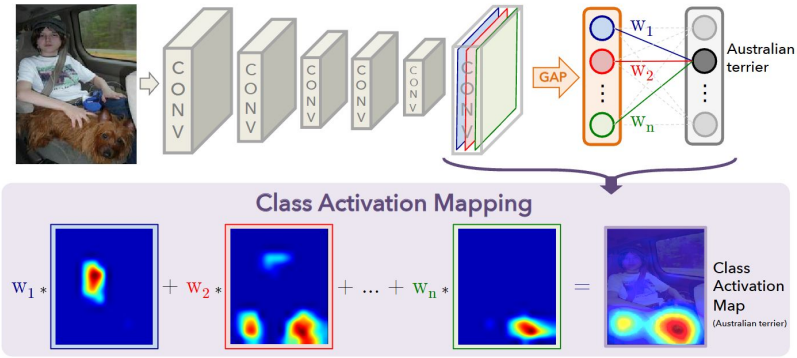
Learning Deep Features for Discriminative Localization

Bolei Zhou, Aditya Khosla, Agata Lapedriza, Aude Oliva, Antonio Torralba
Computer Science and Artificial Intelligence Laboratory, MIT
{bzhou,khosla,agata,oliva,torralba}@csail.mit.edu

Abstract

In this work, we revisit the global average pooling layer proposed in [13], and shed light on how it explicitly enables the convolutional neural network to have remarkable localization ability despite being trained on image-level labels. While this technique was previously proposed as a means for regularizing training, we find that it actually builds a generic localizable deep representation that can be applied to a variety of tasks. Despite the apparent simplicity of

Class activation maps (CAM)



We can combine the k feature maps and weights w into a representation of

1. How important a high-level feature is (this is w), and
2. where on the image this high-level feature is (this is the feature map before GAP).

CAM procedure:

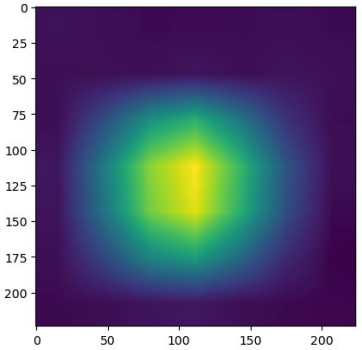
1. Weigh each feature map by the corresponding w, resulting in k weighted feature maps
2. Sum over k, resulting in a coarse 2D map (1 channel, H×W corresponding to that of the feature map)

$$M_c(x, y) = \sum_k w_k^c f_k(x, y)$$

(here, c denotes the class, f_k feature map k, and x,y the position on the feature map)

3. Upscale, smoothen and overlay

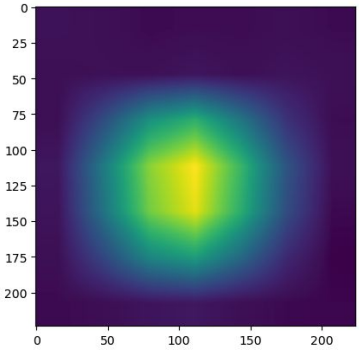
Class activation maps (CAM)



While Gradient × Input assumes that f is linear with x (which is wrong), CAM rather explains the prediction as a weighted sum of the last high-level features, which is linear (*literally; it's GAP + fully connected layer ⇒ logits (before softmax)*)

```python
activations = []
def forward_hook(module, inp, out):
    activations.append(out.detach())

# Make sure this is the last conv layer!
handle = model.layer4[1].conv2.register_forward_hook(forward_hook)
with torch.no_grad():
    _ = model(input_image)
handle.remove()
acts = activations[0] # [1, C, H, W]

weights = model.fc.weight[pred_class].view(1, -1, 1, 1)  # [1, C, 1, 1]
cam = torch.sum(torch.relu(weights * acts), dim=1, keepdim=False).squeeze(0) # [H, W]

# Interpolate to original image size
target_size = input_image.shape[-2:]
cam = torch.nn.functional.interpolate(
    cam.unsqueeze(0).unsqueeze(0),
    size=target_size,
    mode="bilinear",
    align_corners=False,
)[0, 0]

plt.imshow(image_human.permute(1, 2, 0))
plt.imshow(cam.cpu().detach().numpy(), alpha=0.5)
plt.axis("off")
plt.show()
```

Class activation maps (CAM): Limitations



1. Not all CNNs do convolutions → feature maps → GAP → FC → logits.
   You need to check this before running CAM on your CNN!

2. The final feature maps are often very coarse (have low spatial resolution)
   ⇒ We can't see details

GradCAM addresses these limitations

Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization

Ramprasaath R. Selvaraju · Michael Cogswell · Abhishek Das · Ramakrishna Vedantam · Devi Parikh · Dhruv Batra

Abstract We propose a technique for producing 'visual explanations' for decisions from a large class of Convolutional Neural Network (CNN)-based models, making them more transparent and explainable.

sualization, Guided Grad-CAM, and apply it to image classification, image captioning, and visual question answering (VQA) models, including ResNet-based architectures.

In the context of image classification models, our visualiza-

Dec 2019

Gradient-weighted Class Activation Mapping (Grad-CAM)

GradCAM is a generalization of CAM.

In GradCAM, we can choose any layer's feature map.

When using the last layer, GradCAM = CAM.

When we use previous layers, GradCAM linearly approximates using gradients.

Gradient-weighted Class Activation Mapping (Grad-CAM)

GradCAM is a generalization of CAM, using any layer's feature map.

GradCAM procedure:

1. Select a layer and collect its feature maps.
2. Calculate the gradients G from the output class wrt these feature maps.
3. Apply GAP on G, resulting in a single score (average gradient) per feature map.
4. Weigh these feature maps by the gradient scores, and take the ReLU.

$$\alpha_k^c = \overbrace{\frac{1}{Z}\sum_i\sum_j}^{\text{global average pooling}} \underbrace{\frac{\partial y^c}{\partial A_{ij}^k}}_{\text{gradients via backprop}}$$

$$L^c_{\text{Grad-CAM}} = ReLU\underbrace{\left(\sum_k \alpha_k^c A^k\right)}_{\text{linear combination}}$$

Gradient-weighted Class Activation Mapping (Grad-CAM)

GradCAM is a generalization of CAM, using any layer's feature map.

GradCAM procedure:

1. Select a layer and collect its feature maps.
2. Calculate the gradients G from the output class wrt these feature maps.
3. Apply GAP on G, resulting in a single score (average gradient) per feature map.
4. Weigh these feature maps by the gradient scores.

The difference to CAM: We no longer have the actual weights between high-level features and the prediction. Instead, we estimate them using their gradients.

Grad-CAM reduces to CAM if applied on the final feature map (*then, the gradients just "estimate" (are) the actual weights on z*).

Gradient-weighted Class Activation Mapping (Grad-CAM)

```python
# Compute and visualize Grad-CAM for each layer
fig, axs = plt.subplots(1, len(layers), figsize=(15, 15))
for i, layer in enumerate(layers):
    acts = layer_activations[layer] # [1, C, H, W]
    grads = layer_gradients[layer] # [1, C, H, W]
    weights = grads.mean(dim=(2, 3), keepdim=True) # [1, C, 1, 1]

    # Weighted sum of activations
    grad_cam = (weights * acts).sum(dim=1) # [1, H, W]

    # Interpolate to original image size
    grad_cam = torch.nn.functional.interpolate(
        grad_cam.unsqueeze(0),
        size=input_image.shape[-2:],
        mode="bilinear",
        align_corners=False,
    )[0, 0]

    grad_cam = grad_cam.relu().cpu().detach().numpy()

    axs[i].imshow(image_human.permute(1, 2, 0))
    axs[i].imshow(grad_cam, alpha=0.5)
    axs[i].set_title(f"Grad-CAM layer {i+1}")
    axs[i].axis("off")
plt.tight_layout()
plt.show()
```

Early layer

Mid layers

Last layer = CAM

Gradient-weighted Class Activation Mapping (Grad-CAM)

Early layer *Mid layers* *Last layer = CAM*

What do we see? Which map do you prefer?

Gradient-weighted Class Activation Mapping (Grad-CAM)

We notice that the maps are less informative for early layers.

From the Grad-CAM paper:

> [3] We find that Grad-CAM maps become progressively worse as we move to earlier convolutional layers as they have smaller receptive fields and only focus on less semantic local features.

This is because GradCAM implicitly assumes that each feature contributes linearly to the output.

However, we know that the mapping from early layers is non-linear (*the earlier the layer, the more non-linear the mapping*).

This is exactly the same problem as with Gradient × Input.

2. Non-linear contributions

Non-linear contributions

When functions are linear with input: **Gradient ⨯ Input** = **GradCAM** = **CAM** ⇒ everyone is happy

When functions are non-linear: Things get messy, different methods use different tricks.

- SHAP - quantifies how occluding input regions systematically affects the prediction.
- GradCAM - pretends the model is linear from the feature map to the logits.
- LIME - creates an interpretable surrogate model in a neighborhood.
- Integrated Gradients - interpolates from baseline to the sample with infinite linear functions.
- … and the list goes on.

Integrated gradients - the basic idea

Recall our toy function: $f(x, y) = 5x + y^2$

We want to quantify how x and y contribute. Consider the function value at $f(0, 5) = 0 + 25 = 25$.

Here, y does all the work $\Rightarrow$ we want attribution $(0, 25)$.

Integrated gradients - the basic idea

Recall our toy function: $f(x, y) = 5x + y^2$

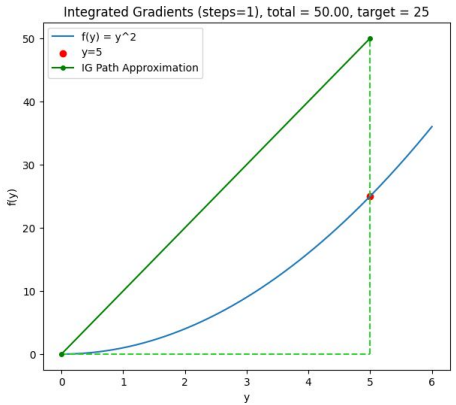
We want to quantify how x and y contribute. Consider the function value at $f(0, 5) = 0 + 25 = 25$.

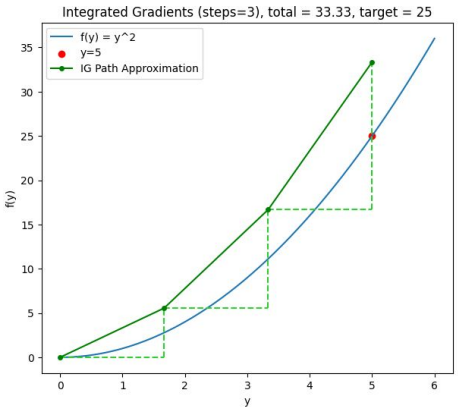
Here, y does all the work $\Rightarrow$ we want attribution $(0, 25)$.

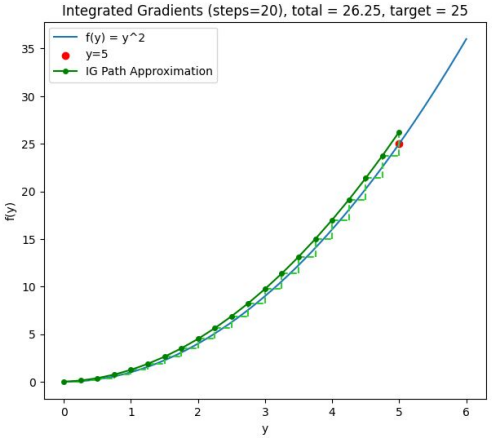
Gradient $\times$ Input yields $(5, 10) \times (0, 5) = (0, 50)$
- assigns contribution correctly to x because f is linear in x
- assigns contribution incorrectly to y (assigns contribution of 10 per unit of y)

Integrated gradients: Instead of calculating attributions locally, instead aggregate contributions starting from a baseline x' and moving towards x.

Integrated gradients - the basic idea







GradientxInput at a single local evaluation
⇒ Overestimates

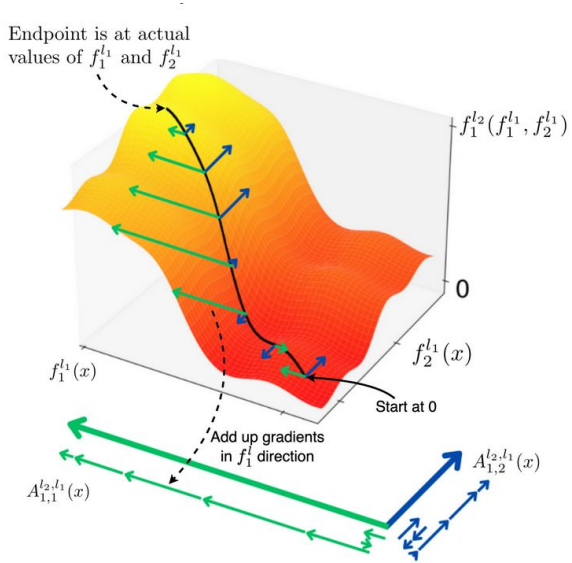
… evaluated and accumulated across multiple steps
⇒ better approximation

… actual integration (summed infinitesimals)
⇒ ~exact contribution

Integrated gradients - the basic idea

We can calculate $f(x)$ - $f(x')$ by integrating.

Why integrate? Because by expressing the difference between the prediction and a baseline as an integral, we can decompose contributions from each feature!



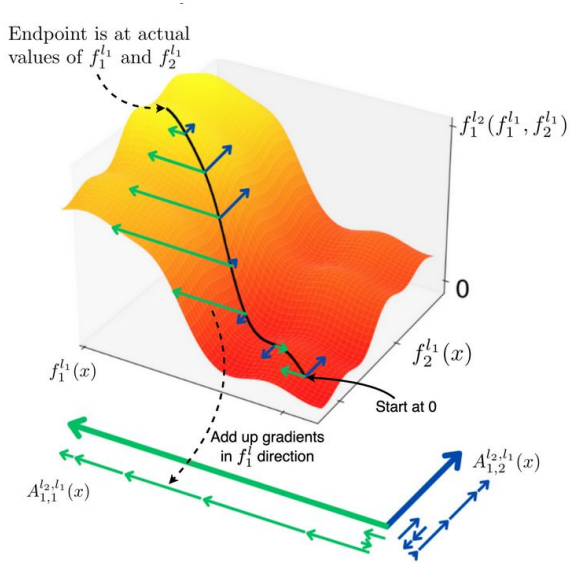
Integrated gradients - the basic idea

We can calculate $f(x)$ - $f(x')$ by integrating.

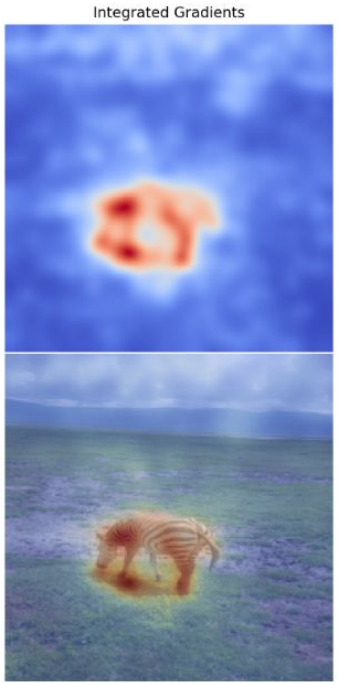
Why integrate? Because by expressing the difference between the prediction and a baseline as an integral, we can decompose contributions from each feature!

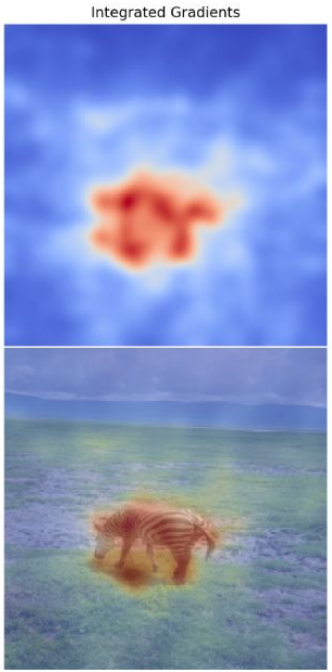
$$\phi_i^{IG}(f, x, x') = \overbrace{(x_i - x'_i)}^{\text{Difference from baseline}} \times \underbrace{\int_{\alpha=0}^{1}}_{\text{From baseline to input...}} \overbrace{\frac{\delta f(x' + \alpha(x - x'))}{\delta x_i} d\alpha}^{\text{...accumulate local gradients}}$$

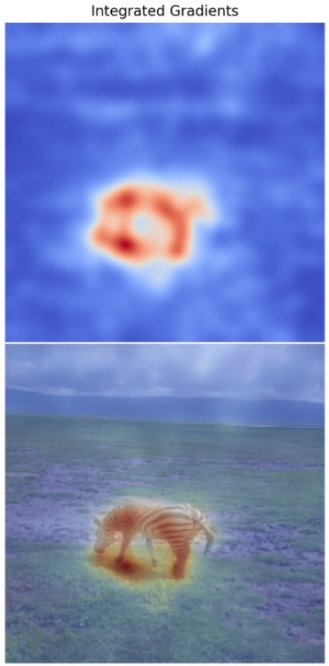
In the figure, feature (dimension) 1 is more important than feature (dimension) 2 at position x from baseline 0.



Integrated gradients - with different baselines







Baseline: Black White Random

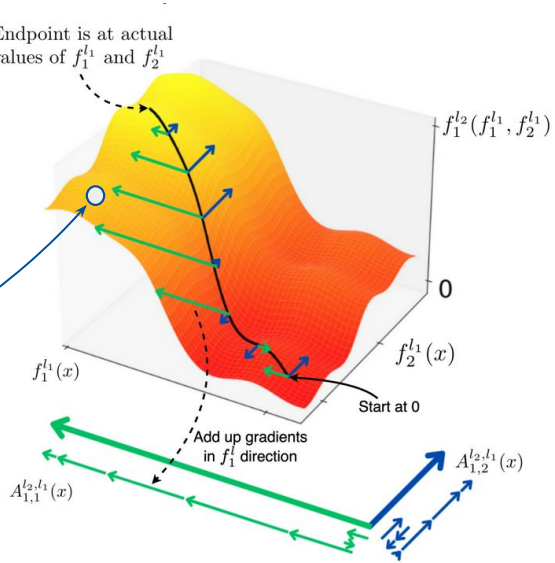
Integrated gradients - the basic idea

We can calculate $f(x)$ - $f(x')$ by integrating.

Why integrate? Because by expressing the difference between the prediction and a baseline as an integral, we can decompose contributions from each feature!

$$\phi_i^{IG}(f, x, x') = \overbrace{(x_i - x'_i)}^{\text{Difference from baseline}} \times \underbrace{\int_{\alpha=0}^{1}}_{\text{From baseline to input...}} \overbrace{\frac{\delta f(x' + \alpha(x - x'))}{\delta x_i} d\alpha}^{\text{...accumulate local gradients}}$$

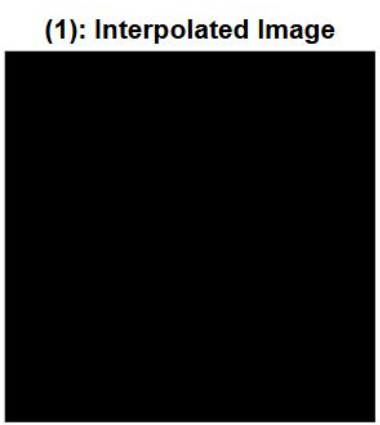
What if we had started at a different position?
What if had chosen another path?



Integrated gradients - challenges

Can we trust that the model behaves properly for each step in the interpolation?

What's a problem related to the interpolated images?



Alpha = 0

Alpha = 0.2

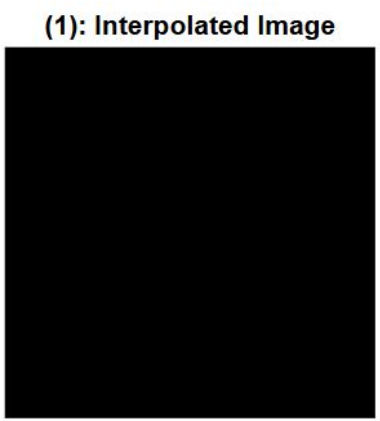
Alpha = 0.4

Alpha = 0.6

Integrated gradients - challenges

Can we trust that the model behaves properly for each step in the interpolation?

The interpolated images are almost surely out of distribution (off-manifold) wrt the training data.



Alpha = 0

Alpha = 0.2

Alpha = 0.4

Alpha = 0.6

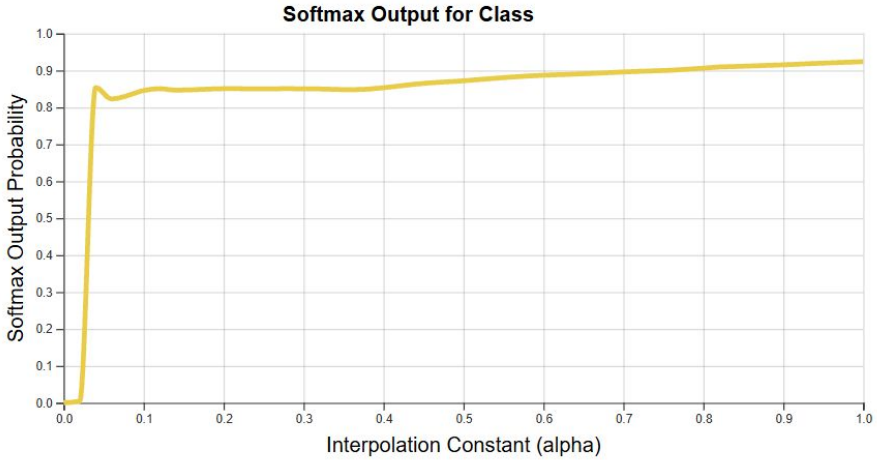
Integrated gradients - challenges

Can we trust that the model behaves properly for each step in the interpolation?

The interpolated images are almost surely out of distribution (off-manifold) wrt the training data.

Integrated gradients - challenges

Can we trust that the model behaves properly for each step in the interpolation?



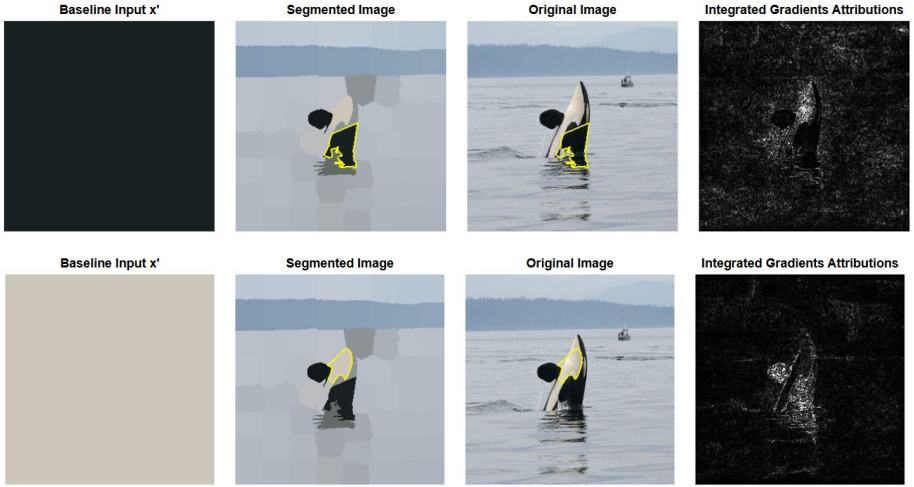
The most important gradients are only available for a small range of alpha!

Integrated gradients - baseline

How do we choose a baseline?

A baseline should represent that all features are absent = no information.

But what happens to black pixels if we choose black as a baseline?



$$\phi_i^{IG}(f, x, x') = \overbrace{(x_i - x'_i)}^{\text{Difference from baseline}} \times \underbrace{\int_{\alpha=0}^{1}}_{\text{From baseline to input...}} \overbrace{\frac{\delta f(x' + \alpha(x - x'))}{\delta x_i} d\alpha}^{\text{...accumulate local gradients}}$$

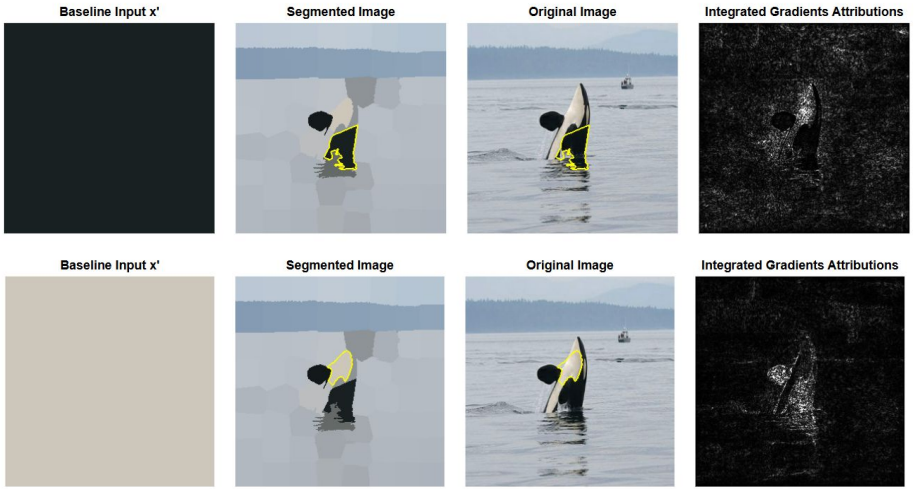
norwegian open ai lab NTNU

Integrated gradients - baseline

How do we choose a baseline?

A baseline should represent that all features are absent = no information.

If the difference from the baseline is 0, the total importance becomes zero (because of multiplication by 0).



$$\phi_i^{IG}(f, x, x') = \overbrace{(x_i - x'_i)}^{\text{Difference from baseline}} \times \underbrace{\int_{\alpha=0}^{1}}_{\text{From baseline to input...}} \overbrace{\frac{\delta f(x' + \alpha(x - x'))}{\delta x_i} d\alpha}^{\text{...accumulate local gradients}}$$

norwegian open ai lab NTNU

Homework :)

Place the methods we've discussed in the model specific/agnostic local/global XAI methods matrix.

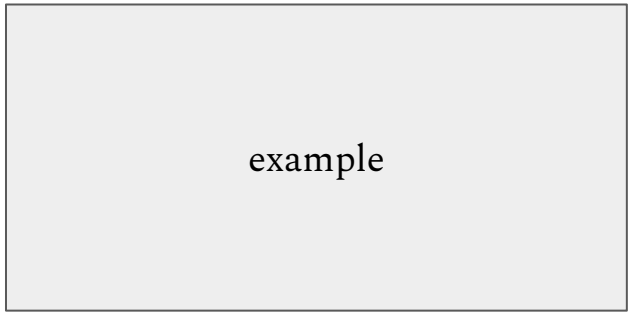
Generate Gradient × Input, CAM, GradCAM, and Integrated Gradients saliency maps for a picture you take yourself.

There’s little homework today, and we’ll do audits tomorrow and Wednesday :)

… what time is it now

Smoothgrad?

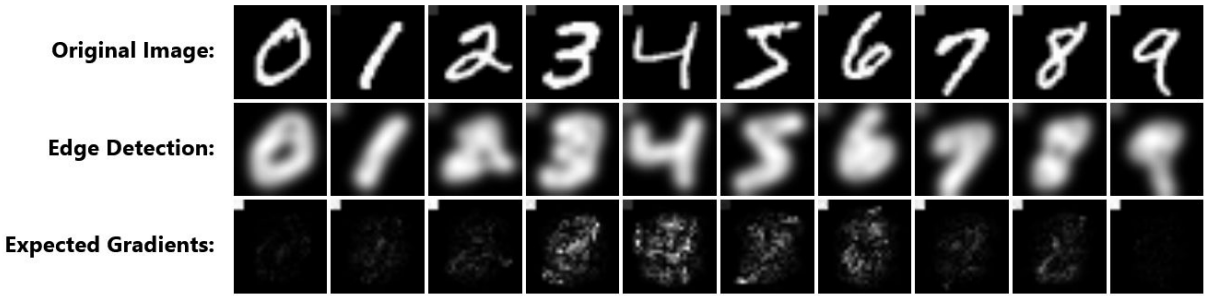
Simple idea: add noise to x, repeat, average



How to select saliency method?

Can we trust qualitative assessment?

We assume that a good saliency map should highlight the target object:
That is what a good model would do.
But the whole point is that we don't know what the model does.



Qualitative assessment can be dangerous because we rely on our human knowledge of the relationship between the data and the labels, and then we assume that an accurate model has learned that very relationship.