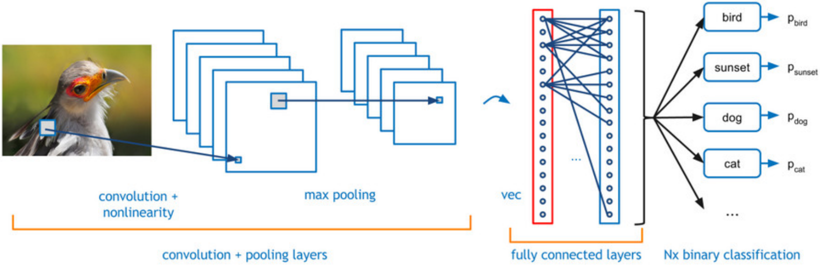
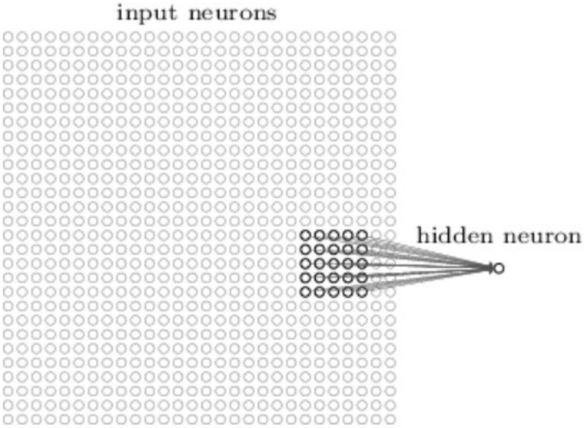
Convolutional networks – theory

Convolutional network



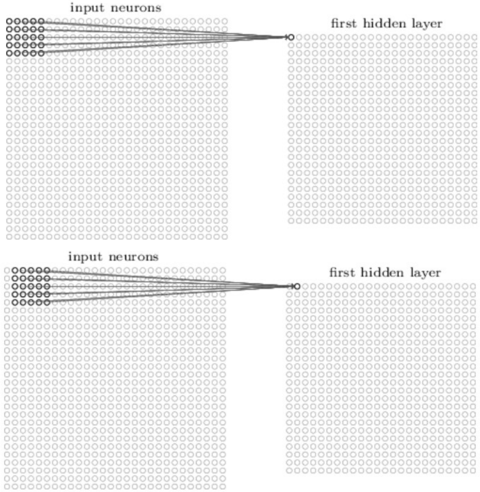
Convolutional layers



Every neuron is connected with a (typically small) *receptive field* of neurons in the lower layer.

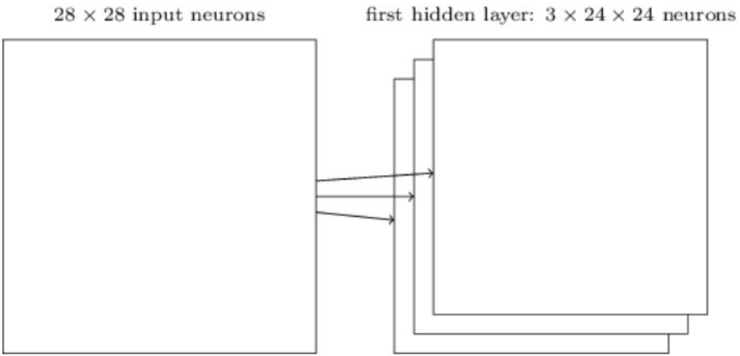
Neuron is "standard": Computes a weighted sum of its inputs, applies an activation function.

Convolutional layers



Neurons grouped into *feature maps* sharing weights.

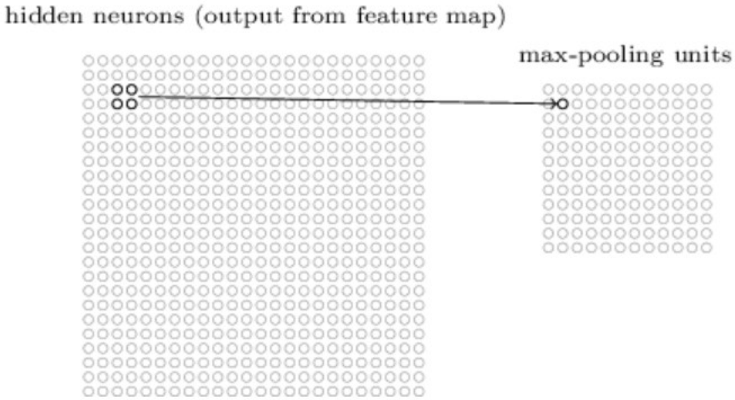
Convolutional layers



Each feature map represents a property of the input that is supposed to be spatially invariant.

Typically, we consider several feature maps in a single layer.

Pooling layers



Neurons in the pooling layer compute simple functions of their receptive fields (the fields are typically disjoint):

- **Max-pooling** : maximum of inputs
- **L2-pooling** : square root of the sum of squres
- **Average-pooling** : mean
- ...

Convolutional networks – architecture

Neurons organized in layers, $L_0, L_1, \ldots, L_n$, connections (typically) only from L_m to L_{m+1}.

Convolutional networks – architecture

Neurons organized in layers, $L_0, L_1, \ldots, L_n$, connections (typically) only from L_m to L_{m+1}.

Several types of layers:

- **input** layer L_0

Convolutional networks – architecture

Neurons organized in layers, $L_0, L_1, \ldots, L_n$, connections (typically) only from L_m to L_{m+1}.

Several types of layers:

- **input** layer L_0
- **dense** layer L_m: Each neuron of L_m connected with each neuron of L_{m-1}.

Convolutional networks – architecture

Neurons organized in layers, $L_0, L_1, \ldots, L_n$, connections (typically) only from L_m to L_{m+1}.

Several types of layers:

- **input** layer L_0
- **dense** layer L_m: Each neuron of L_m connected with each neuron of L_{m-1}.
- **convolutional** layer L_m: Neurons organized into disjoint **feature maps**, all neurons of a given feature map *share weights* (but have different inputs)

Convolutional networks – architecture

Neurons organized in layers, $L_0, L_1, \ldots, L_n$, connections (typically) only from L_m to L_{m+1}.

Several types of layers:

- **input** layer L_0
- **dense** layer L_m: Each neuron of L_m connected with each neuron of L_{m-1}.
- **convolutional** layer L_m: Neurons organized into disjoint **feature maps**, all neurons of a given feature map *share weights* (but have different inputs)
- **pooling** layer: "Neurons" organized into **pooling maps**, all neurons
  - compute a simple aggregate function (such as max),
  - have *disjoint inputs*.

Pooling after convolution is applied to each feature map separately.

I.e. a single pooling map after each feature map.

Convolutional networks – architecture

- Denote
  - X a set of *input* neurons
  - Y a set of *output* neurons
  - Z a set of *all* neurons ($X, Y \subseteq Z$)
- individual neurons denoted by indices i, j etc.
  - ξ_j is the inner potential of the neuron j *after the computation stops*
  - y_j is the output of the neuron j *after the computation stops*

  (define $y_0 = 1$ is the value of the formal unit input)
- w_{ji} is the weight of the connection **from** i **to** j

  (in particular, w_{j0} is the weight of the connection from the formal unit input, i.e. $w_{j0} = -b_j$ where b_j is the bias of the neuron j)
- $j_{\leftarrow}$ is a set of all i such that j is adjacent from i
  (i.e. there is an arc **to** j from i)
- $j^{\rightarrow}$ is a set of all i such that j is adjacent to i
  (i.e. there is an arc **from** j to i)
- $[ji]$ is a set of all connections (i.e. pairs of neurons) sharing the weight w_{ji}.

Convolutional networks – activity

- neurons of dense and convolutional layers:
  - inner potential of neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

  - activation function σ_j for neuron j (arbitrary differentiable):

$$y_j = \sigma_j(\xi_j)$$

Convolutional networks – activity

- neurons of dense and convolutional layers:
  - inner potential of neuron j:

$$\xi_j = \sum_{i \in j_{\leftarrow}} w_{ji} y_i$$

  - activation function σ_j for neuron j (arbitrary differentiable):

$$y_j = \sigma_j(\xi_j)$$

- Neurons of pooling layers: Apply the "pooling" function:
  - max-pooling:

$$y_j = \max_{i \in j_{\leftarrow}} y_i$$

  - avg-pooling:

$$y_j = \frac{\sum_{i \in j_{\leftarrow}} y_i}{|j_{\leftarrow}|}$$

A convolutional network is evaluated layer-wise (as MLP), for each $j \in Y$ we have that $y_j(\vec{w}, \vec{x})$ is the value of the output neuron j after evaluating the network with weights $\vec{w}$ and input $\vec{x}$.

Convolutional networks – learning

Learning:

- Given a **training set** $\mathcal{T}$ of the form

$$\left\{ \left(\vec{x}_k, \vec{d}_k\right) \quad \middle| \quad k = 1, \ldots, p \right\}$$

Here, every $\vec{x}_k \in \mathbb{R}^{|X|}$ is an *input vector* end every $\vec{d}_k \in \mathbb{R}^{|Y|}$ is the desired network output. For every $j \in Y$, denote by d_{kj} the desired output of the neuron j for a given network input $\vec{x}_k$ (the vector $\vec{d}_k$ can be written as $\left(d_{kj}\right)_{j \in Y}$).

- **Error function – mean squared error (for example):**

$$E(\vec{w}) = \frac{1}{p} \sum_{k=1}^{p} E_k(\vec{w})$$

where

$$E_k(\vec{w}) = \frac{1}{2} \sum_{j \in Y} \left(y_j(\vec{w}, \vec{x}_k) - d_{kj}\right)^2$$

Convolutional networks – SGD

The algorithm computes a sequence of weight vectors $\vec{w}^{(0)}, \vec{w}^{(1)}, \vec{w}^{(2)}, \ldots$.

- weights in $\vec{w}^{(0)}$ are randomly initialized to values close to 0
- in the step $t+1$ (here $t = 0, 1, 2\ldots$), weights $\vec{w}^{(t+1)}$ are computed as follows:
  - Choose (randomly) a set of training examples $T \subseteq \{1, \ldots, p\}$
  - Compute

    $$\vec{w}^{(t+1)} = \vec{w}^{(t)} + \Delta\vec{w}^{(t)}$$

    where

    $$\Delta\vec{w}^{(t)} = -\varepsilon(t) \cdot \frac{1}{|T|} \sum_{k \in T} \nabla E_k(\vec{w}^{(t)})$$

Here T is a *minibatch* (of a fixed size),

- $0 < \varepsilon(t) \leq 1$ is a *learning rate* in step $t+1$
- $\nabla E_k(\vec{w}^{(t)})$ is the gradient of the error of the example k

Note that the random choice of the minibatch is typically implemented by randomly shuffling all data and then choosing minibatches sequentially. **Epoch** consists of one round through all data.

Backprop

Recall that $\nabla E_k(\vec{w}^{(t)})$ is a vector of all partial derivatives of the form $\frac{\partial E_k}{\partial w_{ji}}$.

How to compute $\frac{\partial E_k}{\partial w_{ji}}$?

Backprop

Recall that $\nabla E_k(\vec{w}^{(t)})$ is a vector of all partial derivatives of the form $\frac{\partial E_k}{\partial w_{ji}}$.

How to compute $\frac{\partial E_k}{\partial w_{ji}}$?

First, switch from derivatives w.r.t. w_{ji} to derivatives w.r.t. y_j:

- Recall that for every w_{ji} where j is in a dense layer, i.e. does not share weights:

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

Backprop

Recall that $\nabla E_k(\vec{w}^{(t)})$ is a vector of all partial derivatives of the form $\frac{\partial E_k}{\partial w_{ji}}$.

How to compute $\frac{\partial E_k}{\partial w_{ji}}$?

First, switch from derivatives w.r.t. w_{ji} to derivatives w.r.t. y_j:

- Recall that for every w_{ji} where j is in a dense layer, i.e. does not share weights:

$$\frac{\partial E_k}{\partial w_{ji}} = \frac{\partial E_k}{\partial y_j} \cdot \sigma_j'(\xi_j) \cdot y_i$$

- Now for every w_{ji} where j is in a convolutional layer:

$$\frac{\partial E_k}{\partial w_{ji}} = \sum_{r\ell \in [ji]} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot y_\ell$$

- Neurons of pooling layers do not have weights.

Backprop

Now compute derivatives w.r.t. y_j:

- for every $j \in Y$:

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj}$$

  This holds for the squared error, for other error functions the derivative w.r.t. outputs will be different.

Backprop

Now compute derivatives w.r.t. y_j:

- for every $j \in Y$:

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj}$$

  This holds for the squared error, for other error functions the derivative w.r.t. outputs will be different.

- for every $j \in Z \setminus Y$ such that $j^{\rightarrow}$ is either a dense layer, or a convolutional layer:

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma_r'(\xi_r) \cdot w_{rj}$$

Backprop

Now compute derivatives w.r.t. y_j:

- for every $j \in Y$:

$$\frac{\partial E_k}{\partial y_j} = y_j - d_{kj}$$

  This holds for the squared error, for other error functions the derivative w.r.t. outputs will be different.

- for every $j \in Z \smallsetminus Y$ such that $j^{\rightarrow}$ is either a dense layer, or a convolutional layer:

$$\frac{\partial E_k}{\partial y_j} = \sum_{r \in j^{\rightarrow}} \frac{\partial E_k}{\partial y_r} \cdot \sigma'_r(\xi_r) \cdot w_{rj}$$

- for every $j \in Z \smallsetminus Y$ such that $j^{\rightarrow}$ is max-pooling: Then $j^{\rightarrow} = \{i\}$ for a single "max" neuron and we have

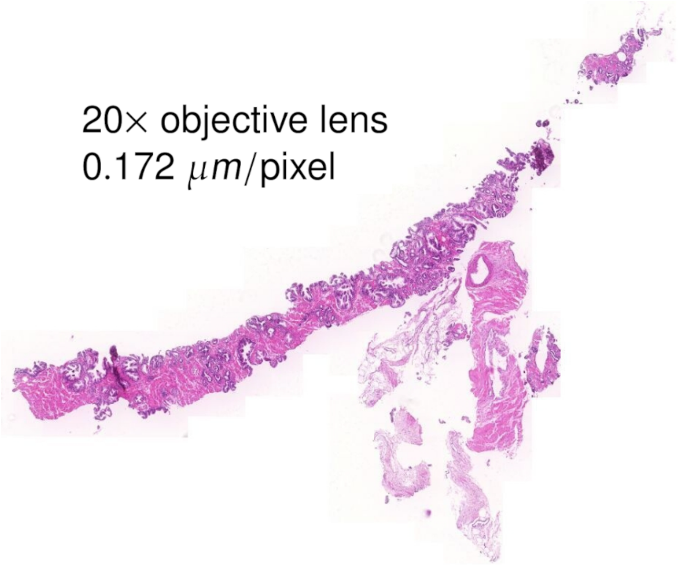
$$\frac{\partial E_k}{\partial y_j} = \begin{cases} \frac{\partial E_k}{\partial y_i} & \text{if } j = arg\ max_{r \in i_{\leftarrow}} y_r \\ 0 & \text{otherwise} \end{cases}$$

I.e. gradient can be propagated from the output layer downwards as in MLP.

Convolutional networks – summary

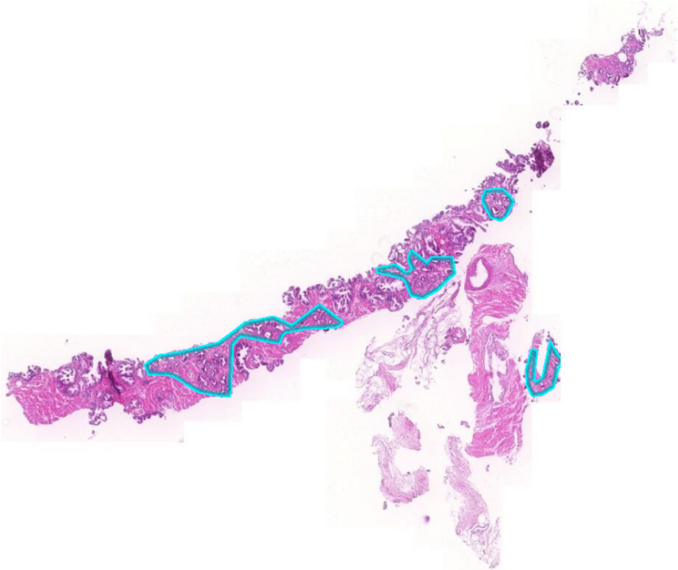
- Conv. nets. are nowadays the most used networks in image processing (and also in other areas where input has some local, "spatially" invariant properties)
- Typically trained using backpropagation.
- Due to the weight sharing allow (very) deep architectures.
- Typically extended with more adjustments and tricks in their topologies.

The problem of cancer detection in WSI



The problem: Detect cancer in this image.

The problem of cancer detection in WSI

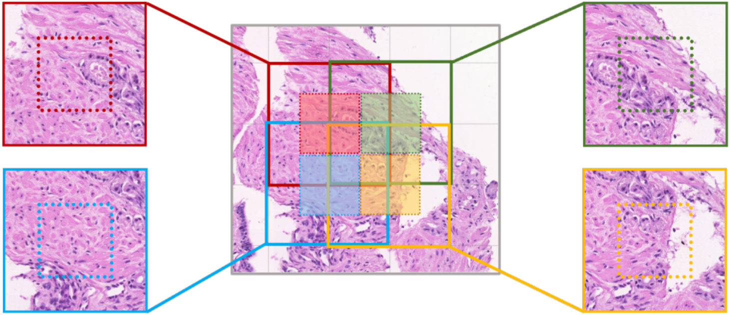


- WSI annotated by pathologists, **not** pixel level precise!

Input data

WSI too large, 105,185 px x 221,772 px

Cut into patches of size 512 px x 512 px



Patch positive **iff** the inner square intersects the annotation

Training on WSI

Our dataset from Masaryk Memorial Cancer Insitute:

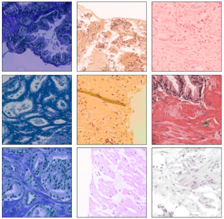
- 785 WSI from 166 patients
  (698 WSI for training, 87 WSI for testing)
- Cut into 7,878,675 patches for training, 193,235 patches for testing.

Training on WSI

Our dataset from Masaryk Memorial Cancer Insitute:

- 785 WSI from 166 patients
  (698 WSI for training, 87 WSI for testing)
- Cut into 7,878,675 patches for training, 193,235 patches for testing.

Dataset augmentation:



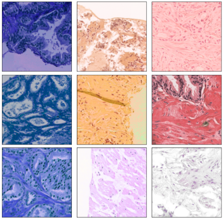
- random vertical and horizontal flips
- random color perturbations

Training on WSI

Our dataset from Masaryk Memorial Cancer Insitute:

- 785 WSI from 166 patients
  (698 WSI for training, 87 WSI for testing)
- Cut into 7,878,675 patches for training, 193,235 patches for testing.

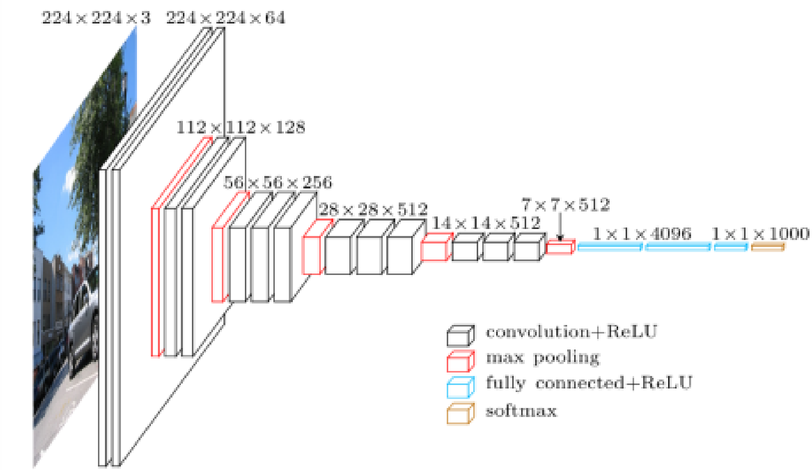
Dataset augmentation:



- random vertical and horizontal flips
- random color perturbations

- Training data three step sampling:
  1. randomly select a label
  2. randomly select a slide containing at least a single patch with the label
  3. randomly select a patch with the label from the slide

VGG16



3×3 convolutions, stride 1, padding 1. Max pooling 2×2, stride 2.

Training VGG16 on WSI

- VGG16 pretrained on the ImageNet (of-the-shelf solution). Top fully connected parts removed, substituted with global max-pooling and a single dense layer.

Training VGG16 on WSI

- VGG16 pretrained on the ImageNet (of-the-shelf solution). Top fully connected parts removed, substituted with global max-pooling and a single dense layer.
- The network has single logistic output - the probability of cancer in the patch

Training VGG16 on WSI

- VGG16 pretrained on the ImageNet (of-the-shelf solution). Top fully connected parts removed, substituted with global max-pooling and a single dense layer.
- The network has single logistic output - the probability of cancer in the patch
- The error E = cross-entropy

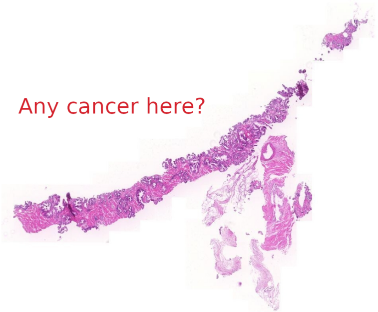
Training VGG16 on WSI

- VGG16 pretrained on the ImageNet (of-the-shelf solution). Top fully connected parts removed, substituted with global max-pooling and a single dense layer.
- The network has single logistic output - the probability of cancer in the patch
- The error *E* = cross-entropy
- Training:
  - RMSprop optimizer
  - The "forgetting" hyperparameter: $\rho = 0.9$
  - The initial learning rate 5×10^{-5}
    - If no improvement in *E* on validation data for 3 consecutive epochs $\Rightarrow$ half the learning rate
    - If no improvement in ROCAUC on validation data for 5 consecutive epochs $\Rightarrow$ terminate
  - Momentum with the weight $\alpha = 0.9$

Prediction

Model evaluation - attempt 1

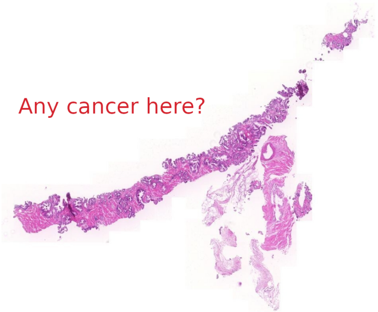
Can we detect cancer somewhere in WSI?



Denote by F the function computed by our model. I.e., given a patch I, $F(I)$ is the output value of the single output neuron with logistic activation function.

Model evaluation - attempt 1

Can we detect cancer somewhere in WSI?

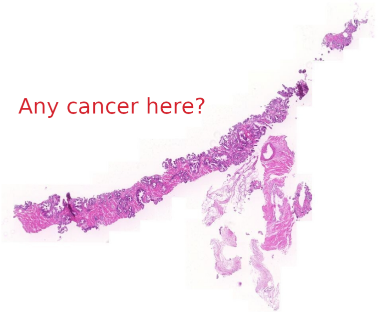


Denote by F the function computed by our model. I.e., given a patch I, $F(I)$ is the output value of the single output neuron with logistic activation function.

Interpret the $F(I)$ as the probability of cancer in the patch.

Model evaluation - attempt 1

Can we detect cancer somewhere in WSI?



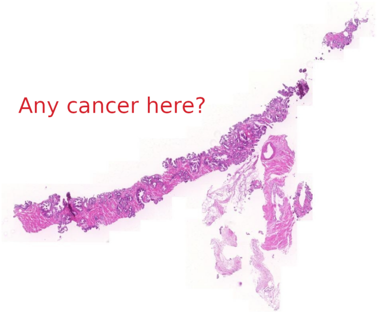
Denote by F the function computed by our model. I.e., given a patch I, $F(I)$ is the output value of the single output neuron with logistic activation function.

Interpret the $F(I)$ as the probability of cancer in the patch.

Predict WSI positive iff at least one patch I satisfies $F(I) \geq t$ for a fixed threshold $t \in [0, 1]$.

Model evaluation - attempt 1

Can we detect cancer somewhere in WSI?



Denote by F the function computed by our model. I.e., given a patch I, $F(I)$ is the output value of the single output neuron with logistic activation function.

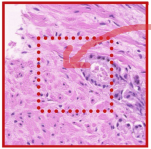
Interpret the $F(I)$ as the probability of cancer in the patch.

Predict WSI positive iff at least one patch I satisfies $F(I) \geq t$ for a fixed threshold $t \in [0,1]$.

Choosing t close to 1, we have achieved 100% accuracy, i.e., slide positive iff predicted positive. Problem solved ... No?

Model evaluation - attempt 2

Can we detect cancer in patches?



Any cancer here?

Predict I positive iff $F(I) \geq 0.75$

Single WSI:

		PREDICTED Pos	PREDICTED Neg
TRUE	Pos	805	18
TRUE	Neg	48	614

All WSIs:

		PREDICTED Pos	PREDICTED Neg
TRUE	Pos	24796	4340
TRUE	Neg	5345	158754

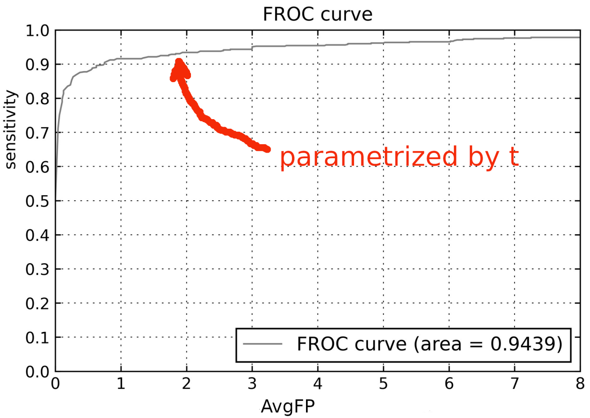
Ok, does it detect cancer?

Model evaluation – attempt 3 – FROC

Detect *particular tumors* ?

How to evaluate the quality of tumor detection?

Model evaluation – attempt 3 – FROC



sensitivity ≈ the proportion of tumors containing at least one patch I with $F(I) \geq t$ w.r.t. all tumors in all slides

AvgFP ≈ average number of patches I with $F(I) \geq t$ in *each non-cancerous slide*

Explainable methods (XAI)

XAI methods

The goal is to understand how and why the network does what it does.

We will consider classification models only.

XAI methods

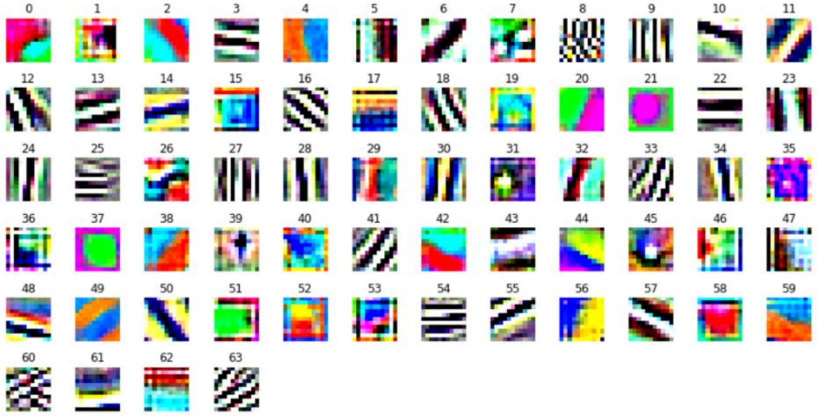
The goal is to understand how and why the network does what it does.

We will consider classification models only.

Methods based on various principles:

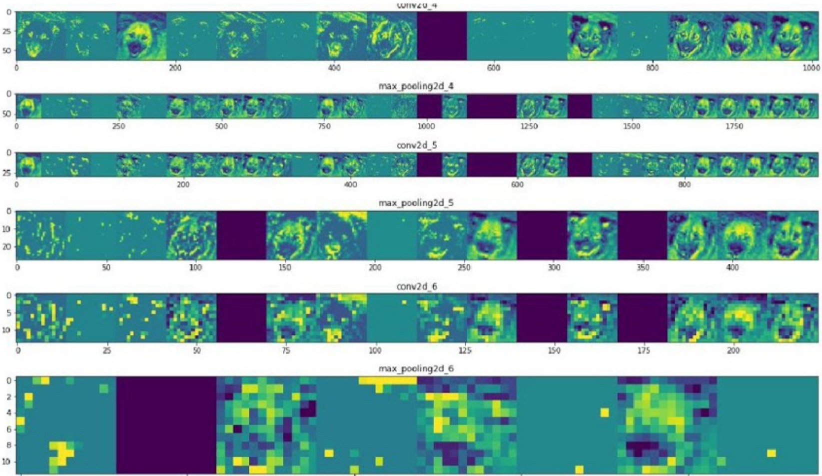
- Visualize weights and feature maps
- Visualize most important inputs for a given class
- Visualize the effect of input perturbations on the output
- Construct an intepretable surrogate model

Alex-net - filters of the first convolutional layer

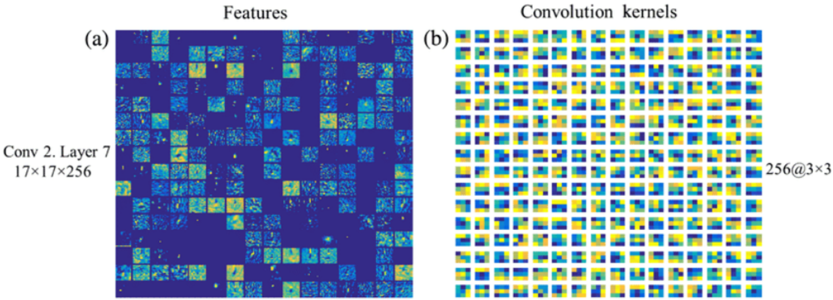


- 64 filters of depth 3 (RGB)
- Combined each filter RGB channels into one RGB image of size 11x11x3.

CNN - feature maps



CNN - feature maps - radar target classification



Synthetic-aperture radar (SAR) – used to create two-dimensional images or three-dimensional reconstructions of objects, such as landscapes.

Maximizing input

Now what if we try to find the most "representative" input vector for a given class?

Maximizing input

Now what if we try to find the most "representative" input vector for a given class?

Assume a trained model giving a score for each class given an input vector.

Maximizing input

Now what if we try to find the most "representative" input vector for a given class?

Assume a trained model giving a score for each class given an input vector.

- Denote by $y_i(\vec{x})$ the value of the *output* neuron $i \in Y$ on an input vector $\vec{x}$.

Maximizing input

Now what if we try to find the most "representative" input vector for a given class?

Assume a trained model giving a score for each class given an input vector.

- Denote by $y_i(\vec{x})$ the value of the *output* neuron $i \in Y$ on an input vector $\vec{x}$.
- Maximize

$$y_i(\vec{x}) - \lambda \left\|\vec{x}\right\|_2^2$$

over all input vectors $\vec{x}$.

Maximizing input

Now what if we try to find the most "representative" input vector for a given class?

Assume a trained model giving a score for each class given an input vector.

- Denote by $y_i(\vec{x})$ the value of the *output* neuron $i \in Y$ on an input vector $\vec{x}$.
- Maximize

$$y_i(\vec{x}) - \lambda \left\|\vec{x}\right\|_2^2$$

  over all input vectors $\vec{x}$.
- A maximizing input vector computed using the gradient descent.
- Gives the most "representative" input vector of the class represented by the neuron i.

Maximizing input - example

dumbbell cup dalmatian

Input specific saliency maps

The goal: Label features in a given input that are "most important" for the output of the network.

Input specific saliency maps

The goal: Label features in a given input that are "most important" for the output of the network.

Various approaches:

- gradient based
  - Gradient saliency maps
  - GradCAM
  - …
- occlusion based
  - Simple occlusion maps
  - LIME
  - …

Gradient based saliency

- Let us fix an output neuron i and an input vector $\vec{x}$.

Gradient based saliency

- Let us fix an output neuron i and an input vector $\vec{x}$.
- **Idea:** Rank every input neuron $k \in X$ based on its influence on the value $y_i(\vec{x})$.

  Note that the vector of input values is *fixed*.

Gradient based saliency

- Let us fix an output neuron i and an input vector $\vec{x}$.
- **Idea:** Rank every input neuron $k \in X$ based on its influence on the value $y_i(\vec{x})$.

  Note that the vector of input values is *fixed*.

  For every input neuron $k \in X$ we consider

  $$\left|\frac{\partial y_i}{\partial y_k}(\vec{x})\right|$$

  to measure the importance of the input y_k for the output y_i with respect to the particular input vector $\vec{x}$.

Gradient based saliency

- Let us fix an output neuron i and an input vector $\vec{x}$.
- **Idea:** Rank every input neuron $k \in X$ based on its influence on the value $y_i(\vec{x})$.

  Note that the vector of input values is *fixed*.

  For every input neuron $k \in X$ we consider

$$\left|\frac{\partial y_i}{\partial y_k}(\vec{x})\right|$$

  to measure the importance of the input y_k for the output y_i with respect to the particular input vector $\vec{x}$.

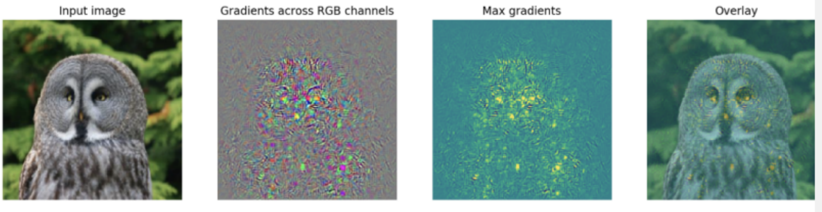
- Note that saliency comes from a surrogate local linear model given by the first-order Taylor approximation:

$$y_i(\vec{x}') \approx y_i(\vec{x}) + \left(\frac{\partial y_i}{\partial X}(\vec{x})\right)(\vec{x}' - \vec{x})$$

  Here $\frac{\partial y_i}{\partial X}$ is the vector of all partial derivatives $\frac{\partial y_i}{\partial y_k}$ where $k \in X$.

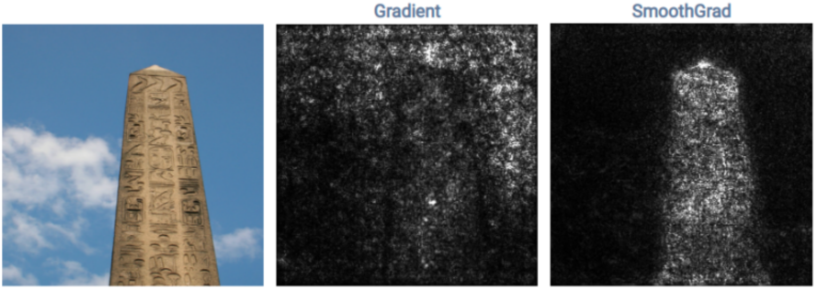
Saliency maps - example

Saliency maps - example



Quite noisy, the signal is spread and does not say much about the perception of the owl.

Saliency maps - example



SmoothGrad:

- Do the following several times:
  - Add noise to the input image
  - Compute a saliency map
- Average the resulting saliency maps.

GradCAM

- Consider a convolutional network and fix an input image I of the network.

  ALL values of all neurons y_j are computed on the input I.

GradCAM

- Consider a convolutional network and fix an input image I of the network.

  ALL values of all neurons y_j are computed on the input I.
- Fix a convolutional layer L consisting of convolutional feature maps $F^1, \ldots, F^k$.

  Each F^ℓ is a set of neurons that belong to the feature map F^ℓ.

  Slightly abusing notation, we write $F^\ell(I)$ to denote the tensor of all values of all neurons in $F^\ell(I)$.

GradCAM

- Consider a convolutional network and fix an input image I of the network.

  ALL values of all neurons y_j are computed on the input I.
- Fix a convolutional layer L consisting of convolutional feature maps $F^1, \ldots, F^k$.

  Each F^ℓ is a set of neurons that belong to the feature map F^ℓ.

  Slightly abusing notation, we write $F^\ell(I)$ to denote the tensor of all values of all neurons in $F^\ell(I)$.
- Fix an output neuron $i \in Y$ with the value y_i.
- Compute the *average importance* of $F^\ell(I)$ for the output y_i:

$$\alpha_i^\ell = \frac{1}{|F^\ell|} \sum_{j \in F^\ell} \frac{\partial y_i}{\partial y_j}(I)$$

GradCAM

- Consider a convolutional network and fix an input image I of the network.

  ALL values of all neurons y_j are computed on the input I.
- Fix a convolutional layer L consisting of convolutional feature maps $F^1, \dots, F^k$.

  Each F^ℓ is a set of neurons that belong to the feature map F^ℓ.

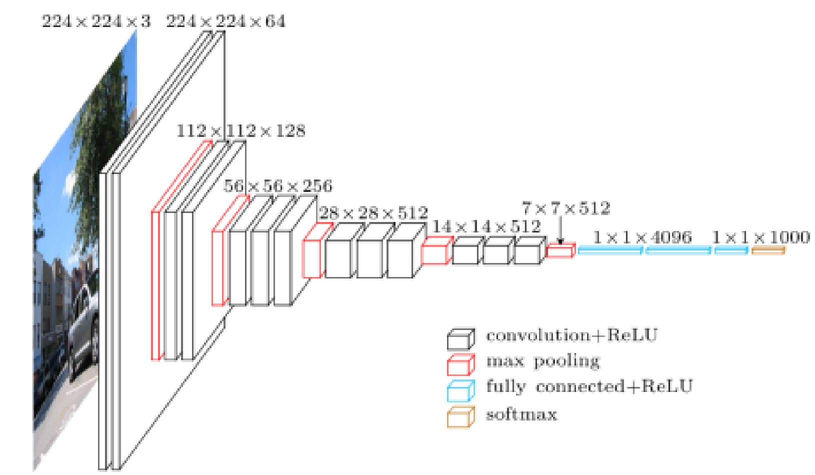
  Slightly abusing notation, we write $F^\ell(I)$ to denote the tensor of all values of all neurons in $F^\ell(I)$.
- Fix an output neuron $i \in Y$ with the value y_i.
- Compute the *average importance* of $F^\ell(I)$ for the output y_i:

$$\alpha_i^\ell = \frac{1}{|F^\ell|} \sum_{j \in F^\ell} \frac{\partial y_i}{\partial y_j}(I)$$

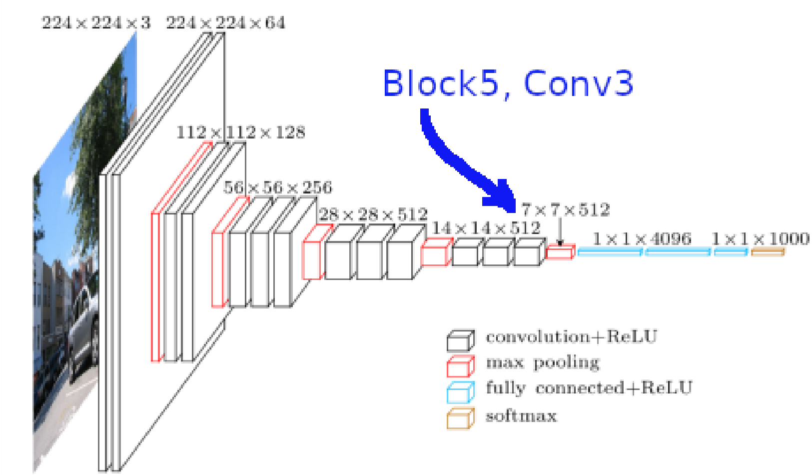
  and the final *gradCAM heat map* for L is obtained using

$$M_i^L = \mathrm{ReLU}\left(\sum_{\ell=1}^{k} \alpha_i^\ell F^\ell(I) \right)$$

GradCAM on VGG16

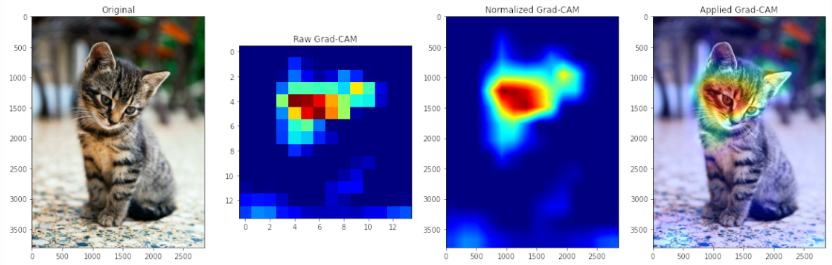


GradCAM on VGG16



Consider the last convolutional layer of the VGG16 (Block5, Conv3)

GradCAM on VGG16



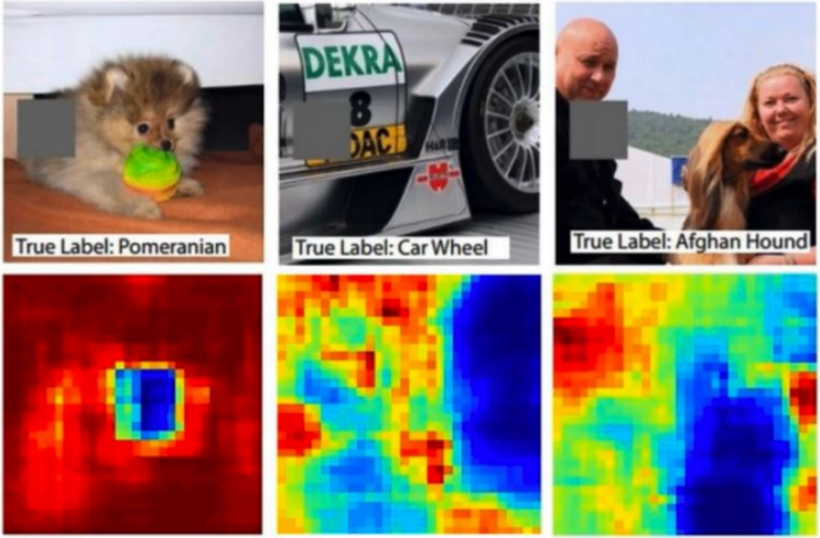
From left to right:

- An image of a cat (has to be resized to 224×224 to fit VGG16)
- The gradCAM heat map for the last convolutional layer and the class "cat"
- Rescaled and smoothed gradCAM heat map.
- The gradCAM overlay.

Occlusion

- Systematically cover parts of the input image.
- Observe the effect on the output value.
- Find regions with the largest effect.

Occlusion - example



Occlusion - example

LIME - for images

Let us fix an image I to be explained.

LIME - for images

Let us fix an image I to be explained.

Outline:

- Consider superpixels of I as interpretable components.

LIME - for images

Let us fix an image I to be explained.

Outline:

- Consider superpixels of I as interpretable components.
- Construct a linear model approximating the network aroung the image I with weights corresponding to the superpixels.

LIME - for images

Let us fix an image I to be explained.

Outline:

- Consider superpixels of I as interpretable components.
- Construct a linear model approximating the network aroung the image I with weights corresponding to the superpixels.
- Select the superpixels with weights of large magnitude as the important ones.

Original Image

Interpretable Components

Superpixels as interpretable components

Original Image

Interpretable Components

Denote by $P_1, \ldots, P_\ell$ all superpixels of I.

Superpixels as interpretable components

Original Image

Interpretable Components

Denote by $P_1, \ldots, P_\ell$ all superpixels of I.

Consider binary vectors $\vec{x} = (x_1, \ldots, x_\ell) \in \{0, 1\}^\ell$.

Superpixels as interpretable components

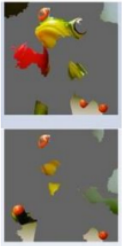
Original Image

Interpretable Components

Denote by $P_1, \ldots, P_\ell$ all superpixels of I.

Consider binary vectors $\vec{x} = (x_1, \ldots, x_\ell) \in \{0,1\}^\ell$.

Each such vector $\vec{x}$ determines a "subimage" $I[\vec{x}]$ of I obtained by removing all P_k with $x_k = 0$.



LIME

- Let us fix an output neuron i, we dnote by $y_i(J)$ the value of the output neuron i for the input image J.

LIME

- Let us fix an output neuron i, we dnote by $y_i(J)$ the value of the output neuron i for the input image J.
- Given the image I to be interpreted, consider the following training set:

$$\mathcal{T} = \left\{ (\vec{x}_1, y_i(I[\vec{x}_1])), \ldots, (\vec{x}_p, y_i(I[\vec{x}_p])) \right\}$$

Here $\vec{x}_h = (x_{h1}, \ldots, x_{h\ell})$ are (some) binary vectors of $\{0, 1\}$. E.g., randomly selected.

LIME

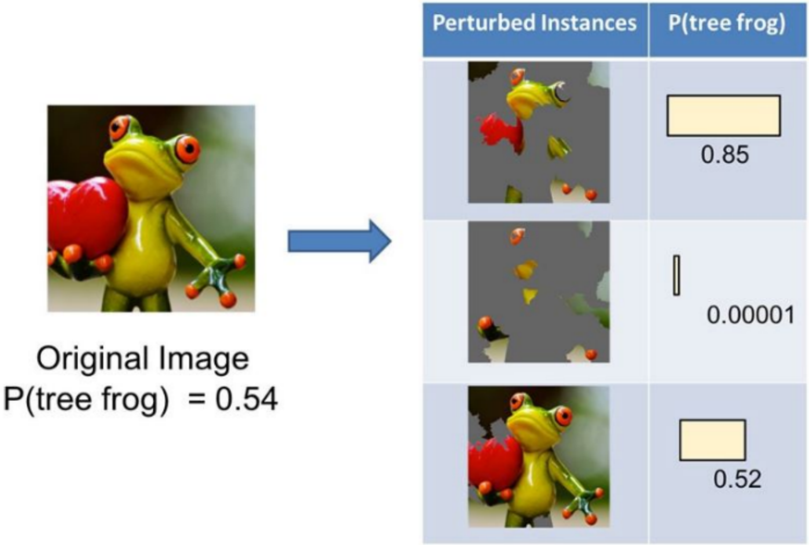
- Let us fix an output neuron i, we dnote by $y_i(J)$ the value of the output neuron i for the input image J.
- Given the image I to be interpreted, consider the following training set:

$$\mathcal{T} = \left\{(\vec{x}_1, y_i(I[\vec{x}_1])), \ldots, (\vec{x}_p, y_i(I[\vec{x}_p])\right\}$$

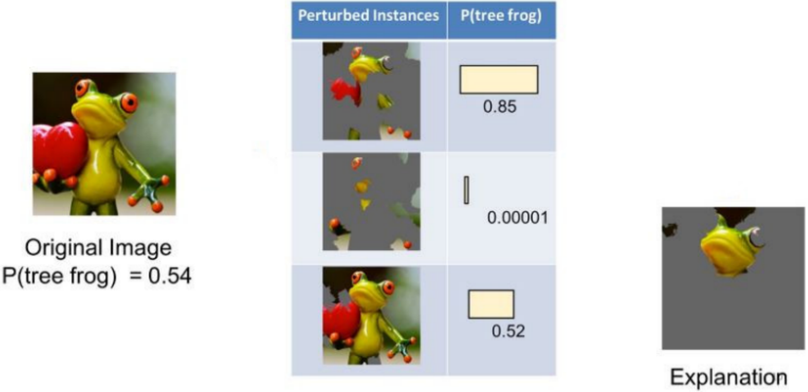
  Here $\vec{x}_h = (x_{h1}, \ldots, x_{h\ell})$ are (some) binary vectors of $\{0, 1\}$. E.g., randomly selected.
- Train a linear model (ADALINE) with weights $w_0, w_1, \ldots, w_\ell$ on $\mathcal{T}$ minimizing the mean-squared error (+ a regularization term making the number of non-zero weights as small as possible).

  Intuitively, the linear model approximates the networks on "subimages" of I obtained by removing "unimportant" superpixels.
- Inspect the weights (magnitude and sign).

LIME - example



LIME - example



LIME - example

(a) Original Image (b) Explaining *Electric guitar* (c) Explaining *Acoustic guitar* (d) Explaining *Labrador*

LIME - example

(a) Husky classified as wolf

(b) Explanation