Deep Learning Based Image Analysis

Dr. rer. nat. Shekoufeh Gorgi Zadeh

E-Mail: shekoufeh.gorgizadeh@uni.lu

12.12.2024

Special thanks to Prof. Dr.-Ing. Thomas Schultz for sharing these lecture slides and permitting their use in this course. The following slides are derived in part from his lecture *Visual Computing in the Life Sciences,* presented at the b-it and Computer Science department at University of Bonn, Germany. Content has been adapted for educational purposes at the University of Luxembourg.

Training Neural Networks

Overview: Training a Neural Network

- **Training a network** finds suitable weights for its neurons:
  1. Select a loss function
  2. Initialize weights
  3. Update weights using **backpropagation**.
     - Feed a batch of training samples through the network, compute loss
     - Compute partial derivatives of loss with respect to weights
     - Update weights with a certain learning rate along negative gradient
  4. Repeat step 3 for a number of epochs
     - **Epoch** usually denotes one pass over all training data, sometimes defined as a fixed number of weight updates
     - Learning rate often high initially, reduced over time
     - We use a validation loss to decide when to stop, often also to adapt the learning rate

Multi-Class Hinge Loss

Let y_i be the correct class label
Let s_j be the score for class j

Hinge loss for input i is defined as:

$$L_i = \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + 1\right)$$

Total loss over full training set is

$$L = \frac{1}{N} \sum_i L_i$$

Cat	3.0	**3.0**	1.0
Dog	**1.0**	-5.0	3.2
Bird	-2.5	1.5	**-5.0**

Multi-Class Hinge Loss

Let y_i be the correct class label
Let s_j be the score for class j

Hinge loss for input i is defined as:

$$L_i = \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + 1\right)$$

Total loss over full training set is

$$L = \frac{1}{N} \sum_i L_i$$

Cat	3.0	**3.0**	1.0
Dog	**1.0**	-5.0	3.2
Bird	-2.5	1.5	**-5.0**
Loss	3.0	0	16.2

Probabilistic Predictions via Softmax

Softmax converts raw scores s_j into predictions of conditional probabilities

$$P(y = k|\mathbf{x} = \mathbf{x}_i) = \frac{e^{s_k}}{\sum_j e^{s_j}}$$

For $\mathbf{x}_i$ being feature vector of sample *i*
y being correct class label for input **x**

			
Cat	3.0	**3.0**	1.0
Dog	**1.0**	-5.0	3.2
Bird	-2.5	1.5	**-5.0**

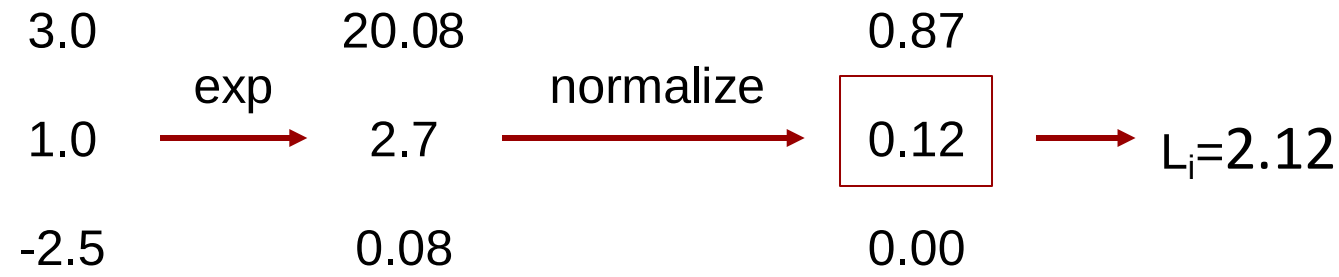
3.0		20.08		0.87
	exp		normalize	
1.0	→	2.7	→	0.12
-2.5		0.08		0.00

Cross-Entropy Loss

Given softmax conversion, minimizing the negative log-likelihood of the true class y_i results in the cross-entropy loss:

$$L_i = -ln\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

Cat	3.0	**3.0**	1.0
Dog	**1.0**	-5.0	3.2
Bird	-2.5	1.5	**-5.0**



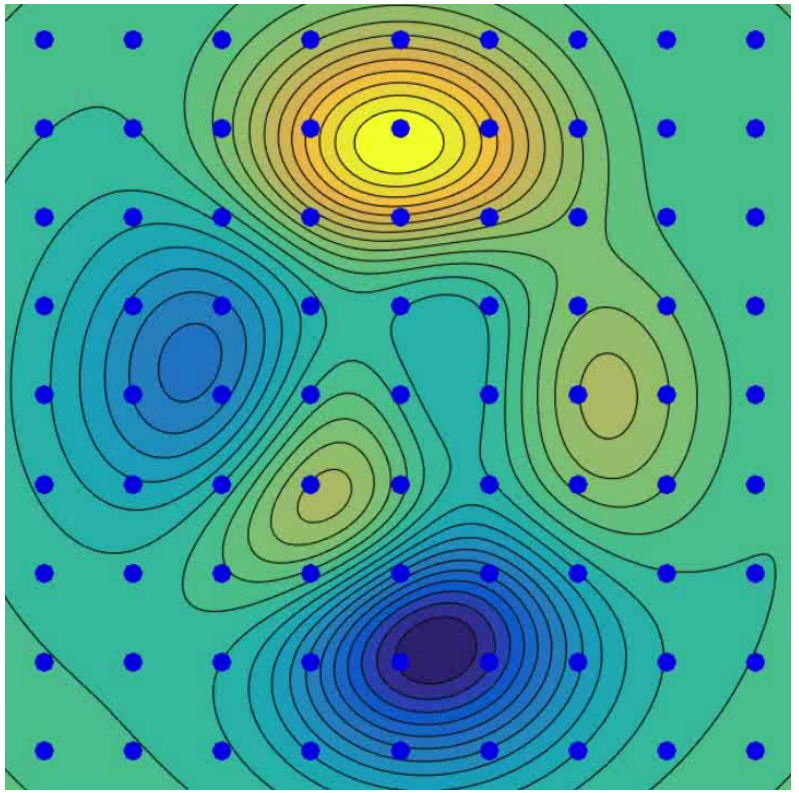
Optimization

- Given a random initialization of W, we need to slowly change the parameters to minimize the loss function
- **Gradient Descent**

  while true:

  weights_grad = eval_grad(loss_fun, data, weights)

  *weights += -step_size * weights_grad*



Step size: also called learning rate

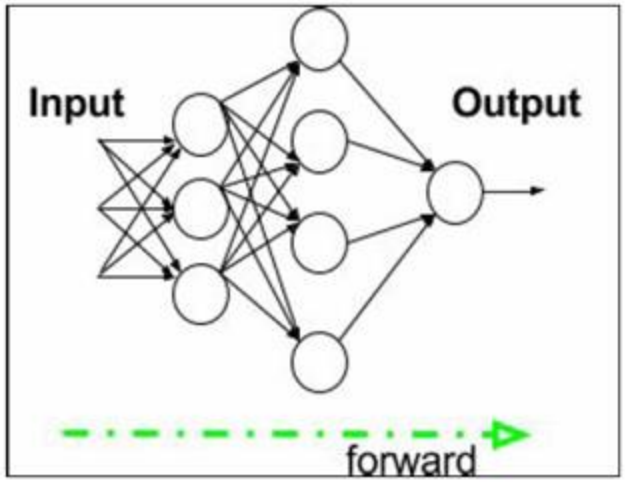
- **Stochastic (Mini-batch) Gradient Descent**
  Only use a small, randomly selected part of the training set to compute gradient. Improves gradient estimation speed but causes inaccuracies (“noise”).

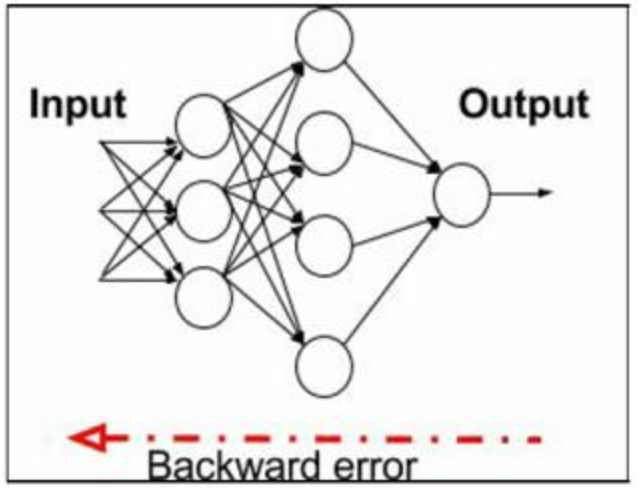
Video: Gabriel Peyré via Wikimedia

Forward/Backward Pass

- **Forward pass**: Values are propagated *forward* from the input stage through the hidden stages to the output stage where a prediction is made and the loss is computed.

- **Backward pass**: Partial derivatives of the loss with respect to the weights are computed using the chain rule, propagating values from the final loss back towards the input layer.
  Information from this step is used to adjust the neural network weights during training.





Momentum Update

- Weight update in **standard gradient descent**:

$$\mathbf{w} \mathrel{+}= -\lambda \nabla_{\mathbf{w}} L$$

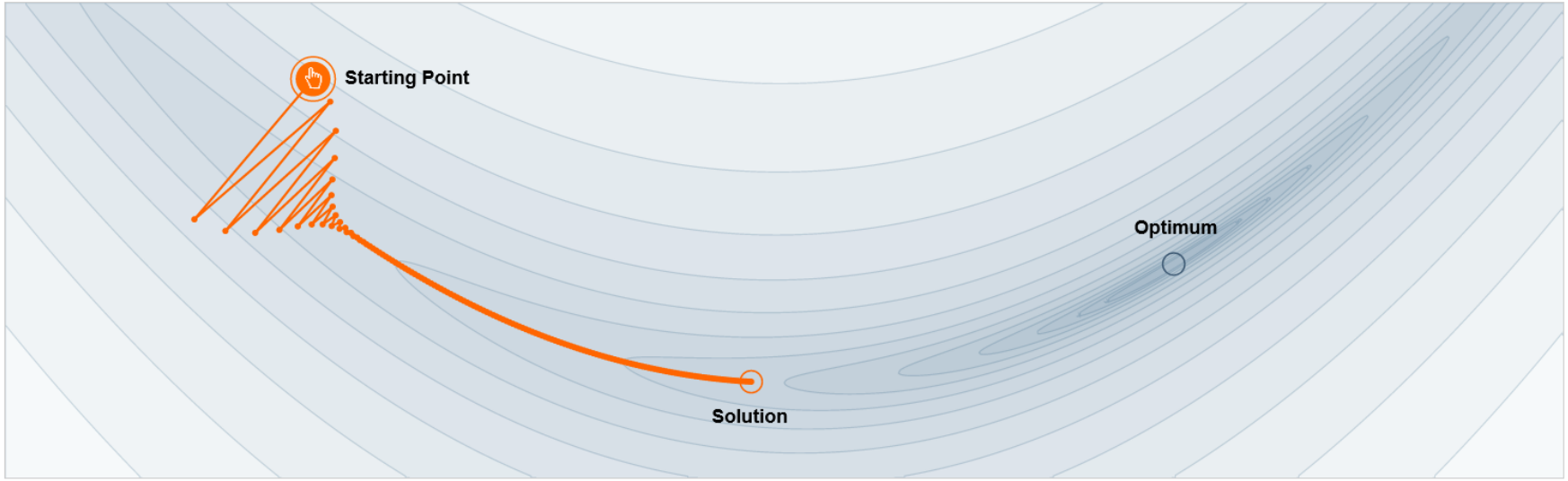
- *Alternative*: „**Momentum**“ update $\mathbf{w} \mathrel{+}= -\lambda \mathbf{v}$

$$\mathbf{v}^{(k)} := \mu \mathbf{v}^{(k-1)} + \nabla_{\mathbf{w}} L$$

  - Velocity $\mathbf{v}$ is initialized to $\mathbf{v}^{(0)} = \mathbf{0}$, and increases over training iterations if the gradient is consistent. This can compensate gradient noise and curvature in the loss function and help overcome shallow suboptima
  - “Momentum” $0 \leq \mu < 1$ and learning rate λ are hyperparameters.
  - *Physical analogy:* Unit mass ball rolling over the loss landscape.
    - *Note:* In this analogy, physical momentum would be proportional to $\mathbf{v}$

Illustration: Gradient Descent vs Momentum

$$\mathbf{w} \mathrel{+}= -0.003 \nabla_{\mathbf{w}} L$$



$$\mathbf{w} \mathrel{+}= -0.003 \mathbf{v}$$
$$\mathbf{v}^{(k)} := 0.8\mathbf{v}^{(k-1)} + \nabla_{\mathbf{w}} L$$

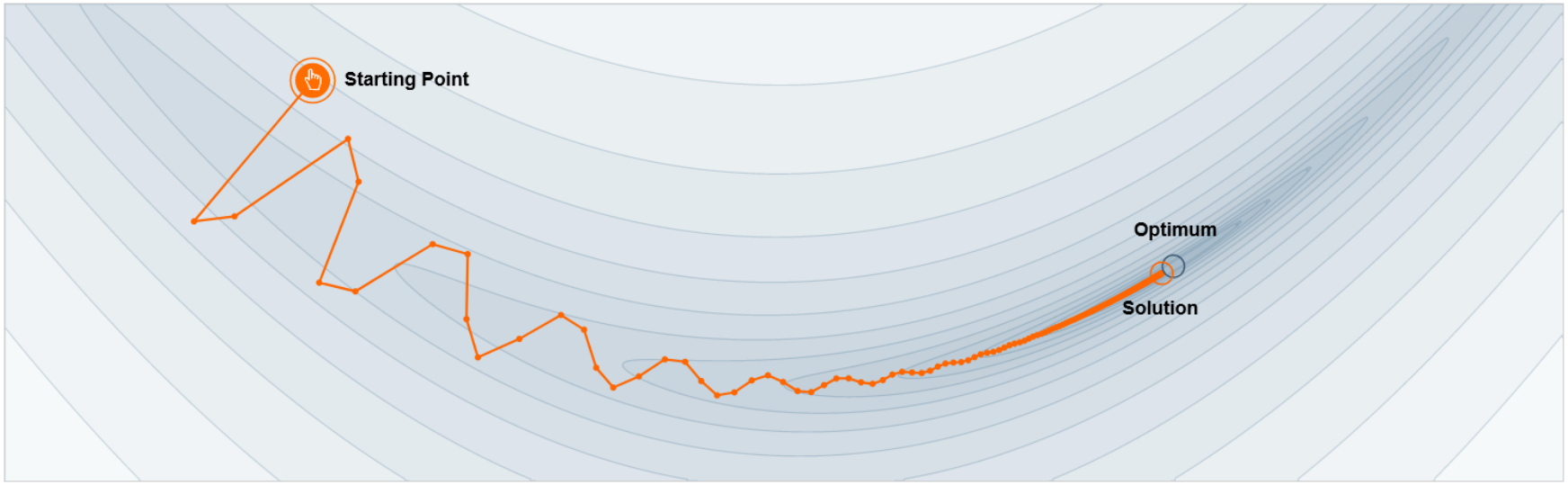
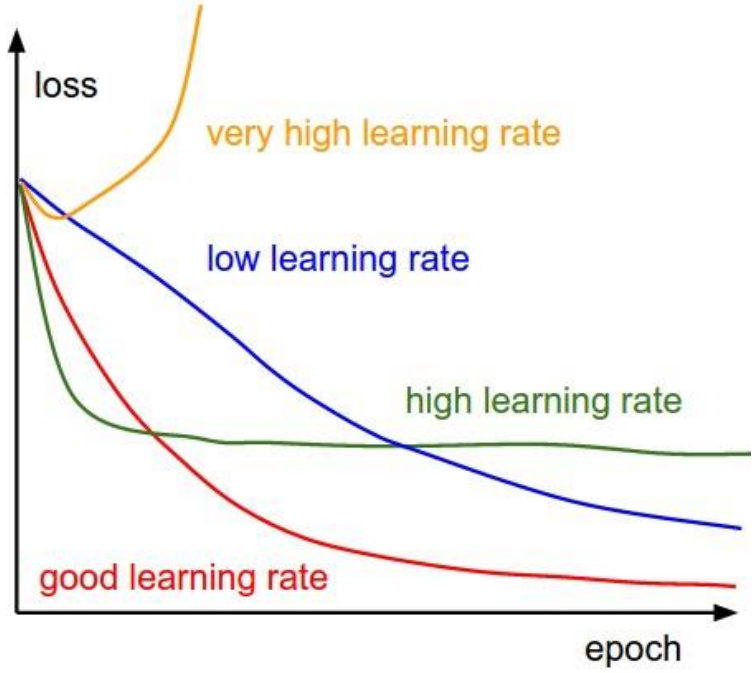
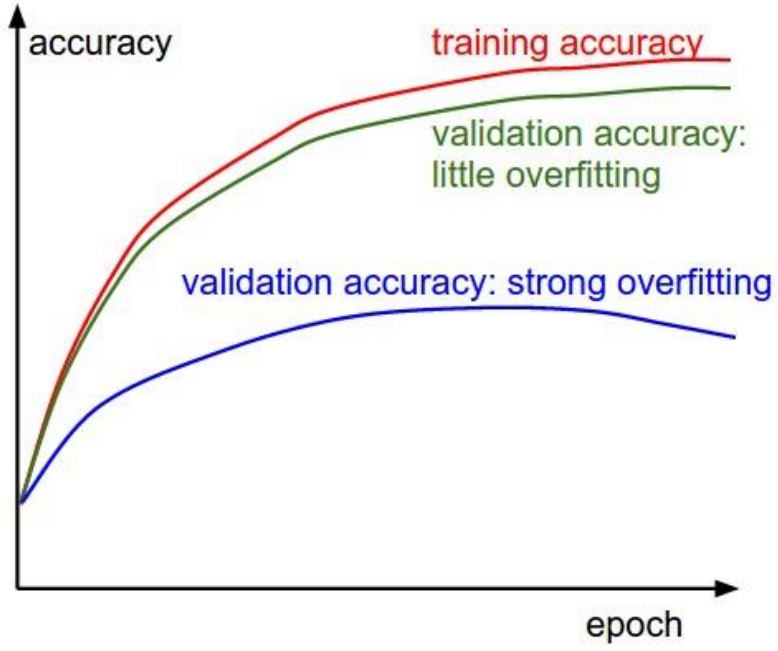


Illustration:
https://distill.pub/2017/momentum/

Monitoring the Training Process

- Monitoring the **training loss** allows us to check that optimization is converging, and to adjust the optimizer or learning rate when needed
- Comparing **training to validation loss** allows us to assess over/underfitting, and to adjust complexity or regularization parameters accordingly





Illustrations from https://cs231n.github.io/neural-networks-3

Learning Rate Decay

While training, it is useful to start with larger learning rates and gradually decrease them.

- **Step decay**: Reduce the learning rate after a few epochs, or when validation loss reaches a plateau
- **Linear reduction** of an initial learning rate λ_0 to a final learning rate λ_T, effective from iteration *k=T:*

$$\lambda_k = (1-\alpha)\lambda_0 + \alpha\lambda_T \text{ with } \alpha = \min\left(\frac{k}{T}, 1\right)$$

- **Exponential decay**: Reduction with a decay rate γ:

$$\lambda_k = \lambda_0 \mathrm{e}^{-\gamma k}$$

Adaptive Learning Rates: AdaGrad

- *Observation*: The update $\mathbf{w} \mathrel{+}= -\lambda \nabla_{\mathbf{w}} L$ has a larger effect on parameters w_i that more strongly influence the loss
  - *Consequence*: Stable training requires limiting the learning rate λ to the „most sensitive“ neurons. Other neurons might require higher values of λ to make suitable progress.
- **AdaGrad** determines parameter specific learning rates
  - Based on accumulating squared partial derivatives:

$$\mathbf{c} \mathrel{+}= (\nabla_{\mathbf{w}} L)^2$$

$$\mathbf{w} \mathrel{+}= -\frac{\lambda \nabla_{\mathbf{w}} L}{\sqrt{\mathbf{c}} + \epsilon}$$

- **c** has the same dimension as **w**, initially: **c=0**
- Square, square root, division are component-wise
- Adding $\epsilon > 0$ component-wise to avoid division by zero

Adam: AdaGrad + Momentum

- **Adam** is a popular optimizer for deep neural networks. It combines adaptive learning rates and momentum updates

$$\mathbf{v} := \beta_1 \mathbf{v} + (1 - \beta_1) \nabla_{\mathbf{w}} L$$
$$\mathbf{c} := \beta_2 \mathbf{c} + (1 - \beta_2) (\nabla_{\mathbf{w}} L)^2$$
$$\mathbf{w} \mathrel{+}= -\frac{\lambda \mathbf{v}}{\sqrt{\mathbf{c}} + \epsilon}$$

  - Cache **c** is "leaky" with factor β_2, to avoid overly low learning rates
  - *Not shown:* Separate equations for „warm-up phase" in which **v** and **c** are close to **0**
  - Typical values: $\beta_1 = 0.9$, $\beta_2 = 0.999$, $\epsilon = 10^{-8}$

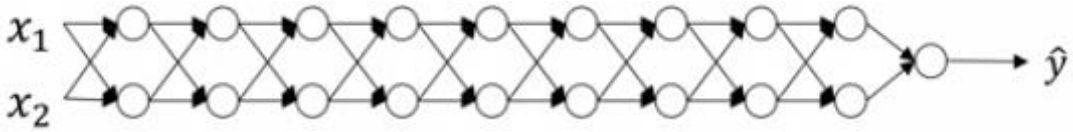
Weight Initialization

- **All zero**: Results in same g[illegible] in backward pass, and therefore to exactly the sa[illegible]. We need to break symmetry!

DO NOT USE!

- **Small random numbers**: Initialize using a zero-centered Gaussian distribution with 0.01 standard deviation. This works for shallow networks, but activations "die" in deeper ones.

  Imagine a network with a linear activation function $f(\alpha) = \alpha$:

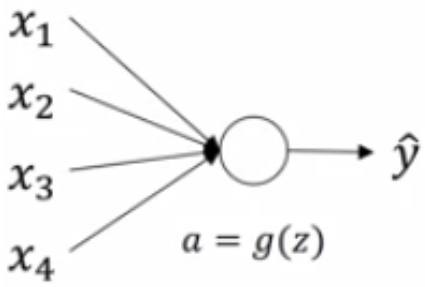


$$\hat{y} = W_{l-1}W_{l-2} \dots W_2 W_1 X$$

  - If coefficients of W are small, activations exponentially converge to zero as the network gets deeper.
  - If we make them too large, activations might explode.

Weight Initialization

- **Xavier et al.**: Initialize weights from a Gaussian distribution with standard deviation $\sqrt{1/N}$, where *N* is the number of input connections. Justification: Attempt to achieve same variance in output as in inputs



$$z = w_1x_1 + w_2x_2 + \cdots + w_Nx_N + b$$

When adding independent Gaussian random variables, variances add. Thus, $\boldsymbol{var(w_i) = \frac{1}{N}}$.

- **He et al.**: Derivation above neglects the nonlinear activation. If ReLu is used, then initialization from Gaussian with standard deviation of $\sqrt{2/N}$ works better.
- **Remark:** Biases are often initialized to zero.

Batch Normalization: Basic Idea

- Optimizers typically update all layers simultaneously, based on their current input
  - **But:** Adjusting weights for earlier layers shifts the inputs of later ones
- **Useful trick:** Re-parametrize activations $\mathbf{H}$ to zero mean/unit variance

$$\mathbf{H}' = \frac{\mathbf{H} - \boldsymbol{\mu}}{\boldsymbol{\sigma}}$$

  - *Notation:* Each row of **H** corresponds to one sample in a mini batch, each column to a neuron. Subtraction/division are applied per-row
  - During **training**, $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ are computed for each mini batch. We back-propagate through this re-parametrization (note that it is differentiable)
  - At **test time,** we apply $\boldsymbol{\mu}$ and $\boldsymbol{\sigma}$ from a running average during training

Batch Normalization: Details

- *Problem:* Constraining all neurons to zero mean / unit variance makes the network less powerful.
- *Solution*: Introduce learnable parameters $\boldsymbol{\beta}, \boldsymbol{\gamma}$ and set

$$\mathbf{H}'' = \boldsymbol{\gamma}\mathbf{H}' + \boldsymbol{\beta}$$

  - *Effect*: Activations can have arbitrary mean / variance again
  - *Benefit*: Modified network is easier to train. Mean and std deviation depend directly on $\boldsymbol{\beta}, \boldsymbol{\gamma}$, not on a complex interdependence between the layers
- Batch normalization has been found to facilitate
  - Faster learning and higher accuracy
  - Permits larger learning rates
  - Reduces sensitivity to bad weight initialization
- In addition, it is common to standardize the input to the first layer
- No consensus on whether BN should come before or after activation

Regularization

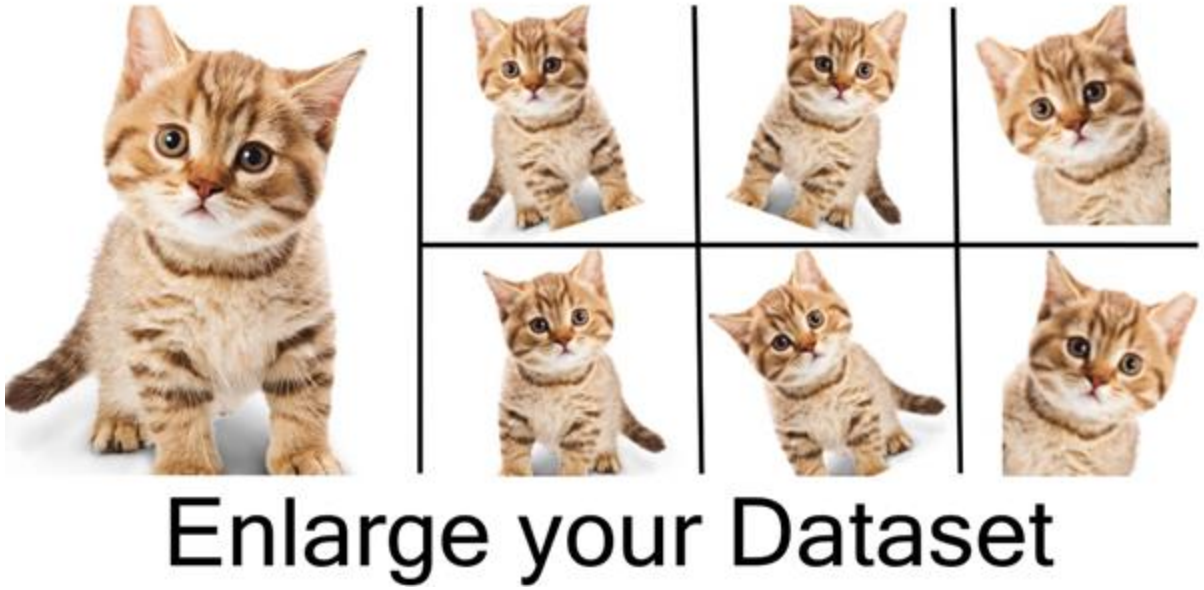
- **Regularization** can be used to avoid overfitting. In the context of deep neural networks, common strategies include:
  - **L2 Regularization** in loss function:

$$L = \underbrace{\frac{1}{N}\sum_i L_i}_{\text{data loss}} + \underbrace{\lambda R(W)}_{\text{regularization loss}} \qquad R(W) = \sum_k \sum_l W_{k,l}^2$$

    with *i* indexing the input, *k* the neuron and *l* the input feature.
  - **Dropout:** Ignore ("drop") neurons and their corresponding input and output connections with probability *p* during training.
    - Sometimes also used at test time, to generate probabilistic output
    - If disabled at test time, need to scale hidden layer outputs by *p*
  - **Early Stopping:** Use parameters from epoch in which validation error was minimal, even if training error could be reduced further.

Data Augmentation

- During training, alter the input image
  - Random crops on the original image
  - Translations
  - Horizontal reflections
  - Warping
  - Changes in color or contrast



Hyperparameter Tuning

Several **hyperparameters** can have a strong effect on the final result, default settings might not be optimal for all inputs and tasks. They include:

- Optimizer / learning rate / decay (parameters, number of epochs)
- Regularization type and strength

In practice, settings are determined by trying values within a plausible range (ideally, automatically and systematically)

- Requires a separate validation set or cross-validation

Summary: Training Neural Networks

- **Backpropagation** algorithm for computing gradient of loss

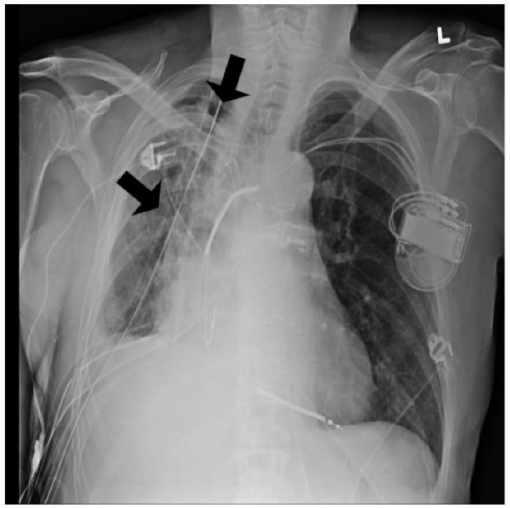
  Loop:
  - Take a batch of training data
  - Feed it forward through the network, compute loss
  - Compute gradient through backpropagation
  - Update weights
  - Adapt learning rate as needed
- **Optimization** algorithms (stochastic gradient descent, momentum, Adam)
- Importance of **regularization**, weight **initialization**, data **preprocessing**, **batch normalization**

Limitations of Image Analysis with Neural Networks

„Clever Hans“ Effect

- Neural networks might learn to use signs of treatment or choice of imaging device instead of signs of the disease itself for classification
- *Examples*:
  - Use of a portable X-ray indicates that patient was too sick to move
  - Presence of drain tube indicates treatment of a pneumothorax, but is not a useful basis for making a diagnosis

Original image



Grad-CAM overlay

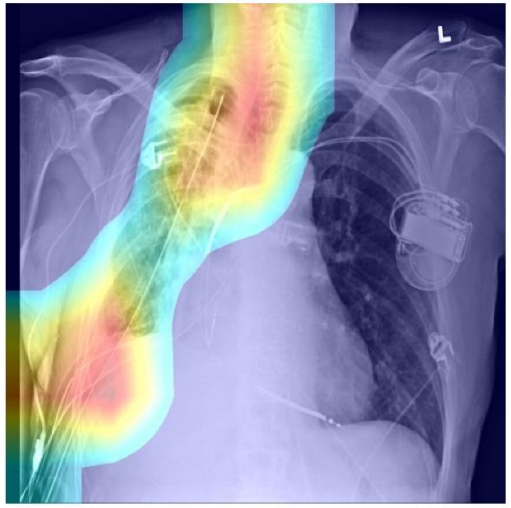


Image: Valsson/Arandjelovic, Sci 2022

Lack of Fairness

- Neural networks might **discriminate** against people based on their gender or ethnicity
- Example
  - uses Grad-CAM (5.8) to reveal this
  - reduces this by more careful selection of training data

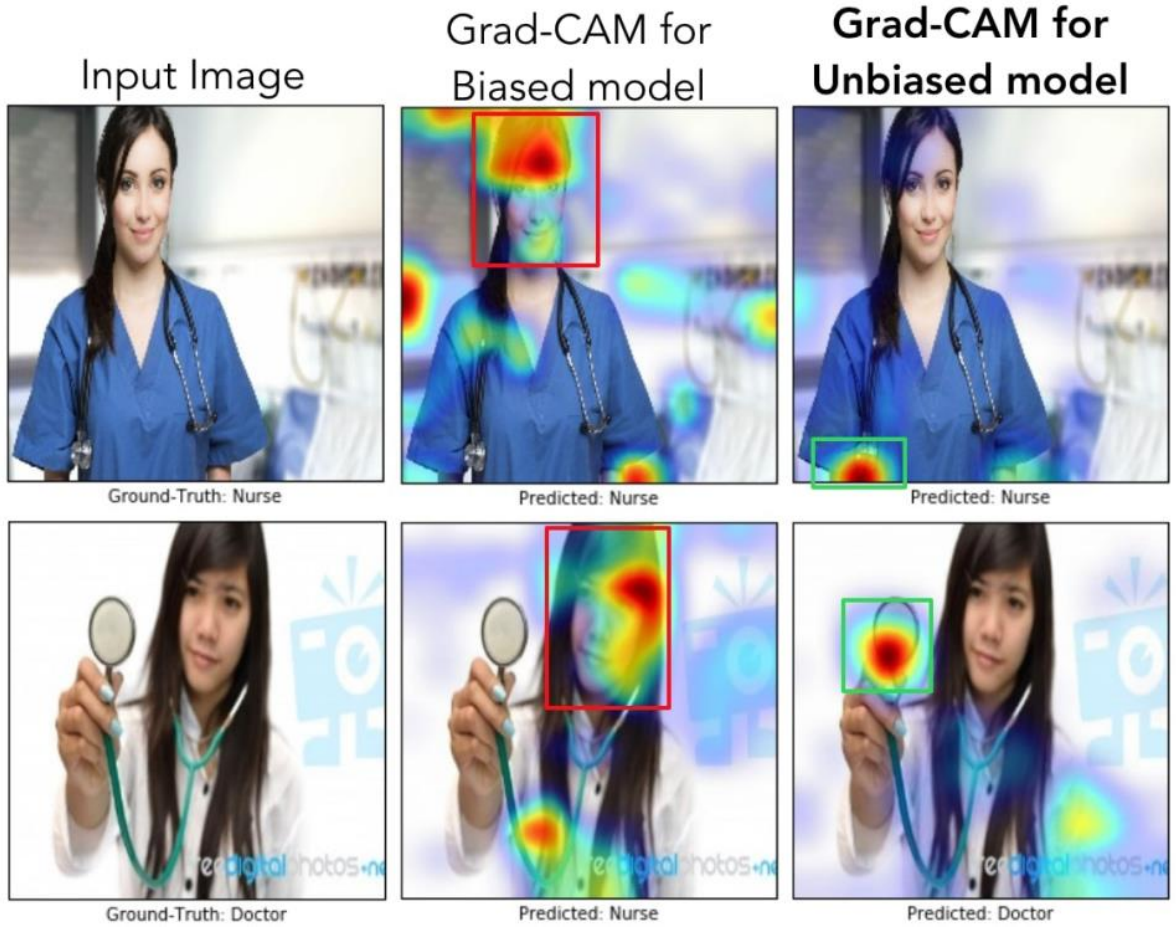
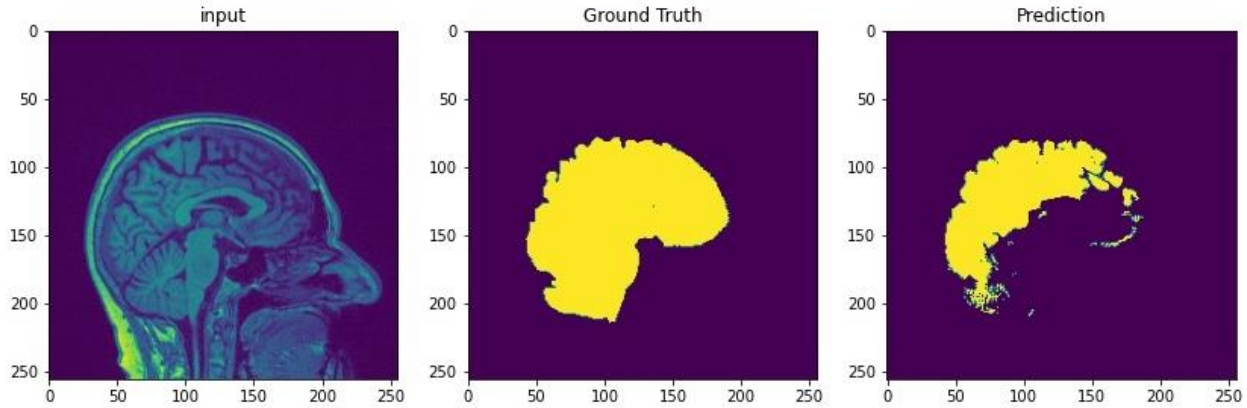


Image: Selvaraju et al. 2019

Domain Shift

- Accuracy is often reduced, sometimes drastically, when characteristics of training and test images differ **(“domain shift”)**
  - *Example*: Scanner change / upgrade
  - „Silent“ Failure



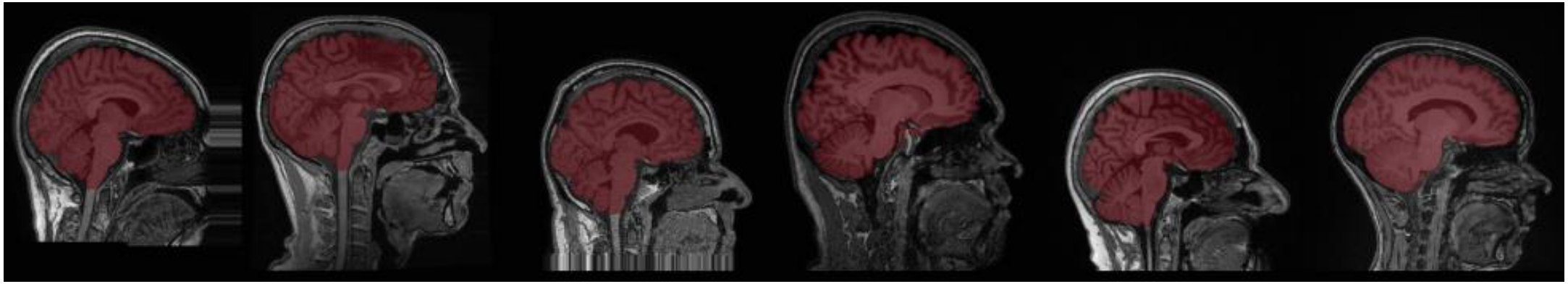


Image: Jonathan Lennartz

Adversarial Attacks

Neural networks can be misled by targeted image manipulations, even if they might not be visible to the human eye

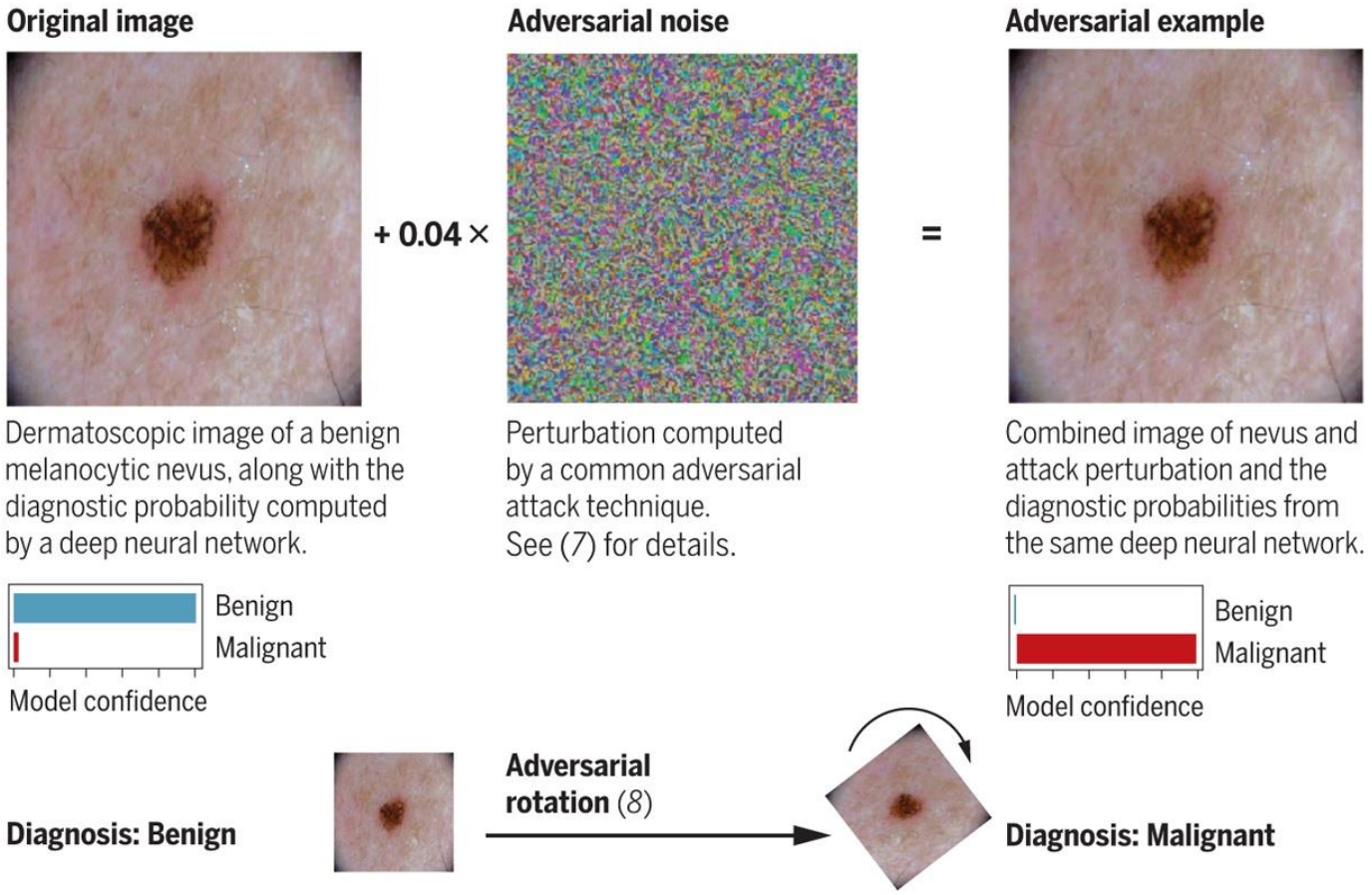


Image: Finlayson et al., Science 2019

Reconstruction of Confidential Training Data

- Training neural networks for medical image analysis requires huge amounts of sensitive **personal data**
- **Federated learning** keeps the image themselves confidential, sends only (updates of) model parameters
- Additional safety measures are required to prevent **reconstruction** of recognizable images from the updates

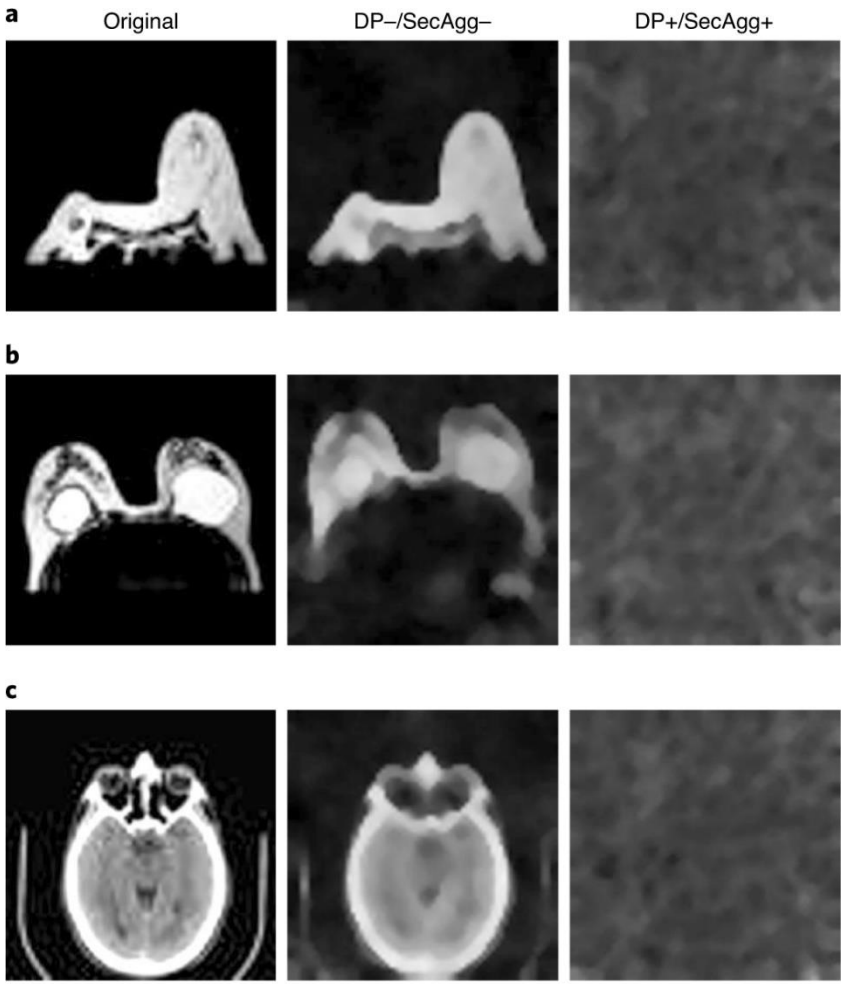


Image: Kaissis et al., Nature Machine Intelligence 2021

Summary: Limitations of Neural Networks

Neural networks are a highly powerful tool for image analysis, but **practical limitations** remain regarding

- the cost of acquiring and annotating sufficient training data
- the “clever Hans” effect
- potential discrimination / lack of fairness
- potentially strong effects of changes in image characteristics
  - domain shifts such as scanner changes
  - adversarial attacks
- risks of leaking confidential training data

Visualizing Neural Networks

Attribution / Heat Maps

- **Question:** Which image region(s) are especially relevant for the network‘s decision?

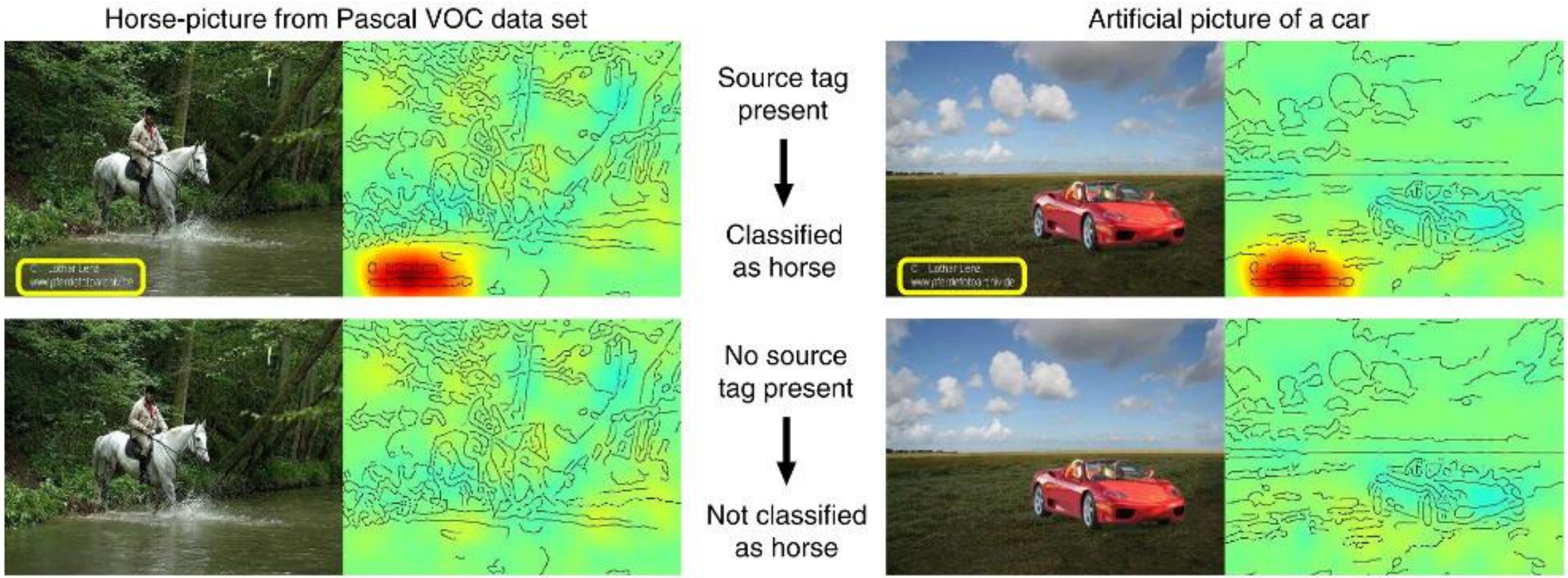


Image from [Lapuschkin et al., Nature Communications 2019]

Occlusion Sensitivity

- **Occlusion sensitivity** studies how hiding parts of the input impacts the network's confidence
  - Might decrease or increase it
  - Computation involves a large number of forward passes
    - depends on desired resolution
  - Results depend on size and shape of the occluder

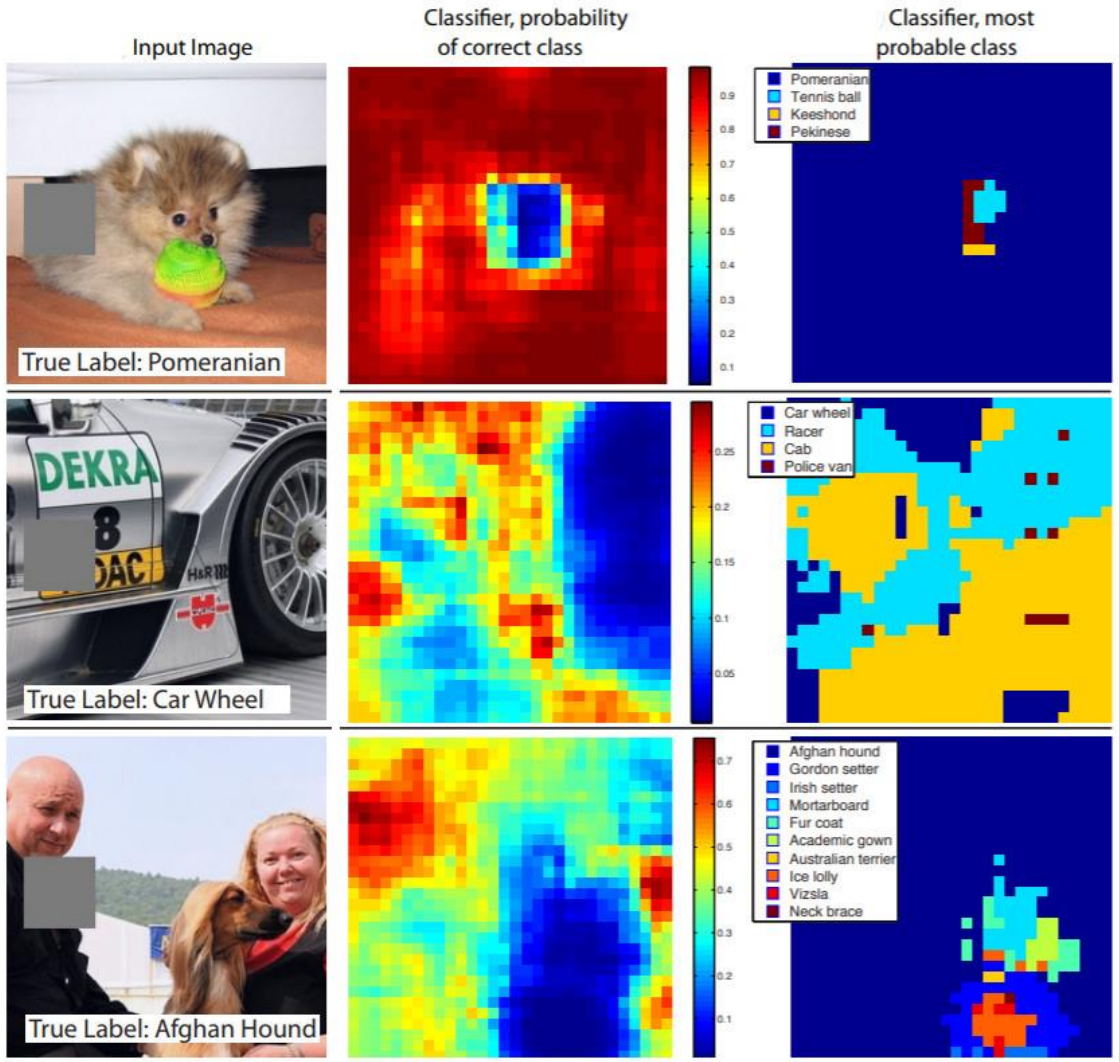


Image adapted from [Zeiler/Fergus, ECCV 2014]

Class Activation Maps: Global Average Pooling

- **Class activation maps** are an alternative for localizing objects using networks trained for classification
  - Neither requires multiple forward passes nor back propagation
  - Assumes that network uses **Global Average Pooling (GAP)**

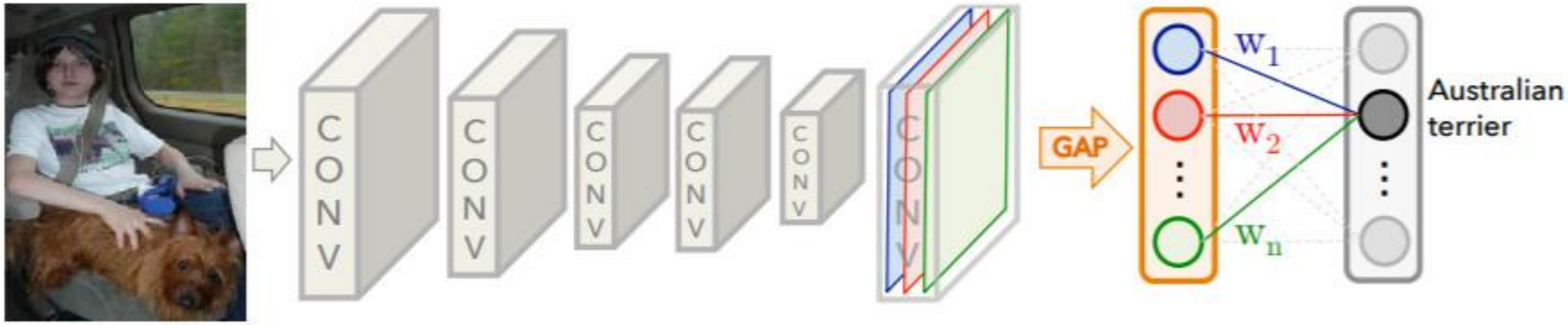


Image from [Zhou et al., CVPR 2016]

Computing Class Activation Maps

- Class activation maps are simply computed as a **weighted average** of the activations in the final convolution layers
  - Upsampled to original image resolution and overlaid

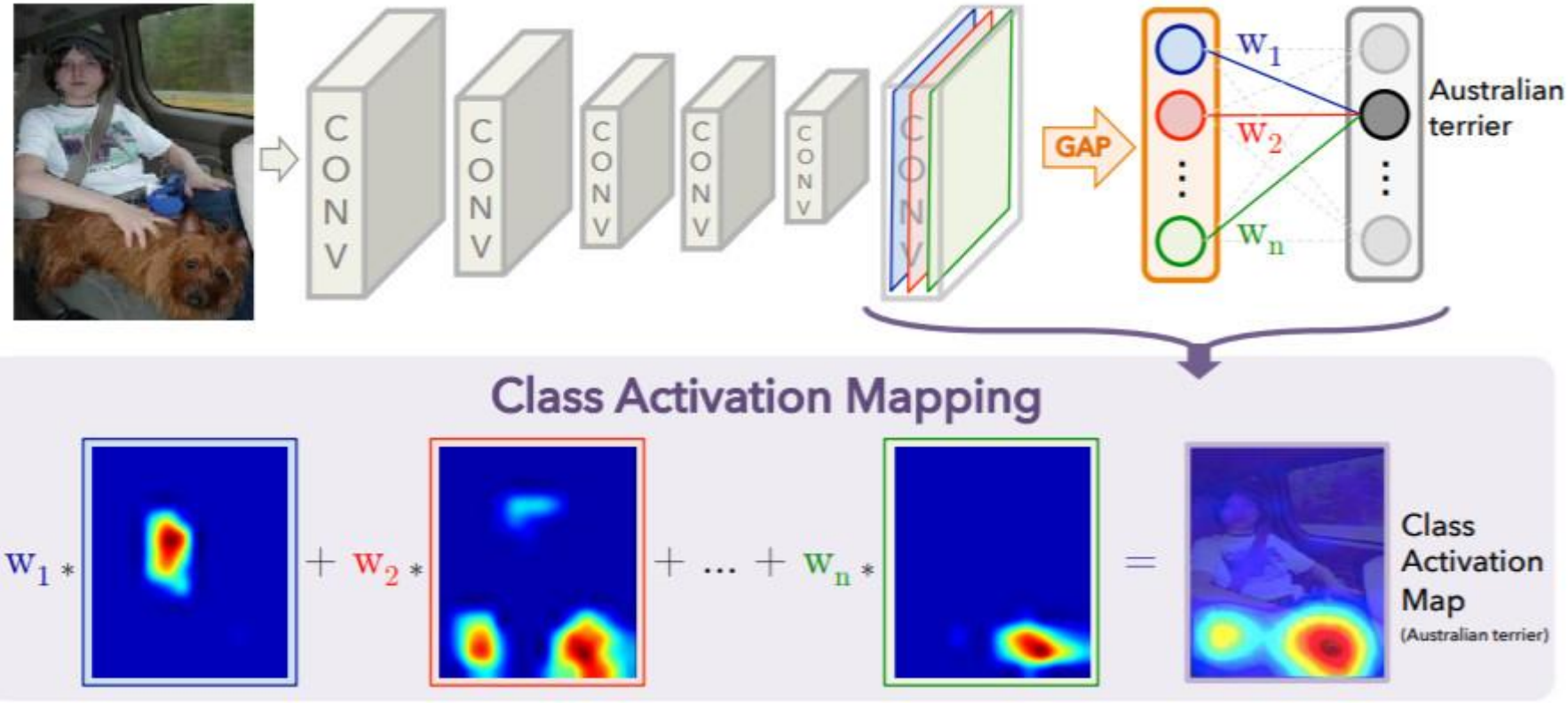


Image from [Zhou et al., CVPR 2016]

Class Activation Maps: Justification

- **Justification** for why class activation maps directly indicate importance for classification:
  - GAP performs $F_k = \sum_{x,y} f_k(x, y)$
  - Class scores obtained as $S_c = \sum_k w_k^c F_k = \sum_k w_k^c \sum_{x,y} f_k(x, y)$
  - Swapping sums yields $S_c = \sum_{x,y} \sum_k w_k^c f_k(x, y)$
  - Class activation map defined as $M_c(x, y) = \sum_k w_k^c f_k(x, y)$
  - Therefore: $S_c = \sum_{x,y} M_c(x, y)$

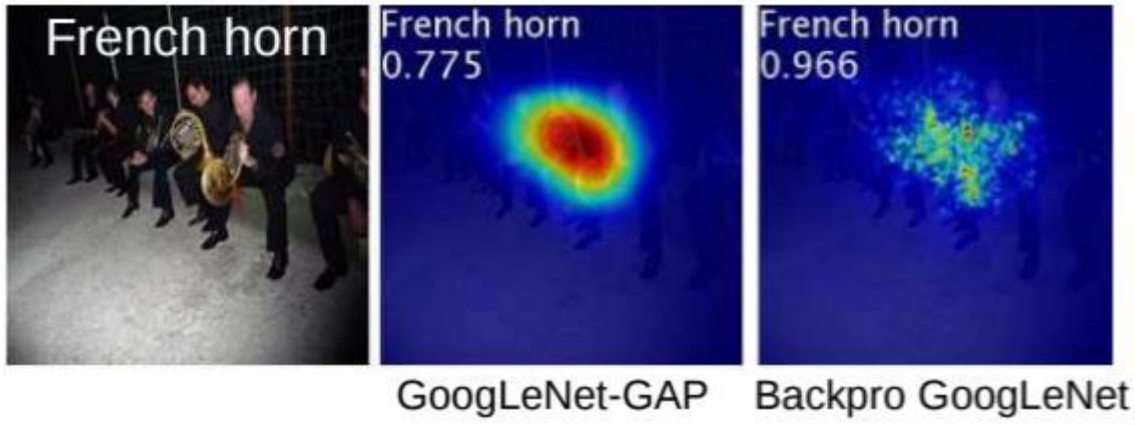


Image from [Zhou et al., CVPR 2016]

Gradient-weighted Class Activation Mapping

- **Grad-CAM** generalizes CAM to arbitrary tasks and architectures
  - Backpropagates output of interest to last convolutional layer
  - Combines activation maps with weights from GAP on gradients

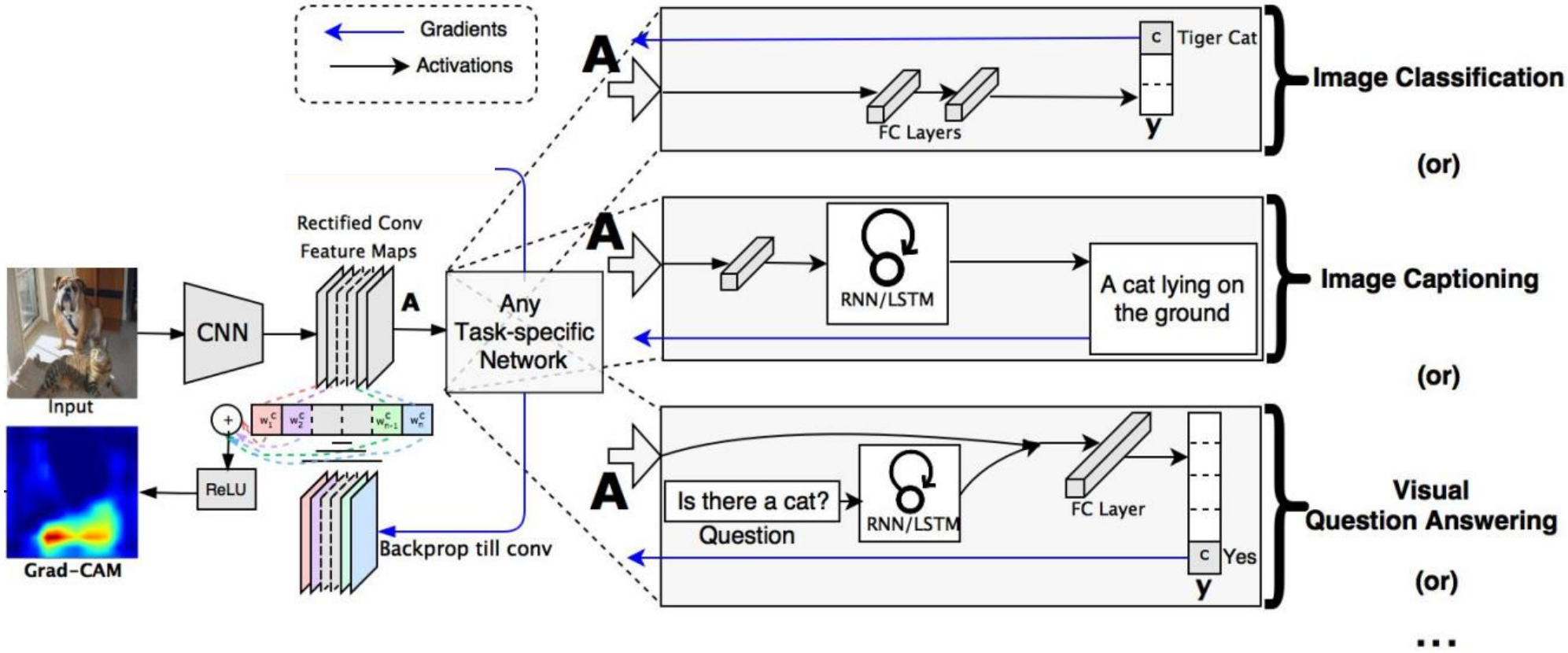


Image adapted from [Selvaraju et al. 2019]

Grad-CAM: Formal Definition

- **Neuron importance weights** α_k^c for activation map k and output score S_c computed via GAP:

$$\alpha_k^c = \frac{1}{Z}\sum_x\sum_y \frac{\partial S_c}{\partial f_k(x,y)}$$

- **Grad-CAM** is computed as the corresponding weighted sum of activation maps, rectified to disregard activations with a negative influence on the desired output:

$$M_{\mathrm{GC}}^c = ReLU\left(\sum_k \alpha_k^c f_k\right)$$

- *Note:* Grad-CAM is a proper generalization of CAM

Example Result: Visual Question Answering

In Visual Question Answering, Grad-CAM provides insight on why the network gave a specific answer and why multiple answers might be plausible

- This example uses a 200 layer ResNet

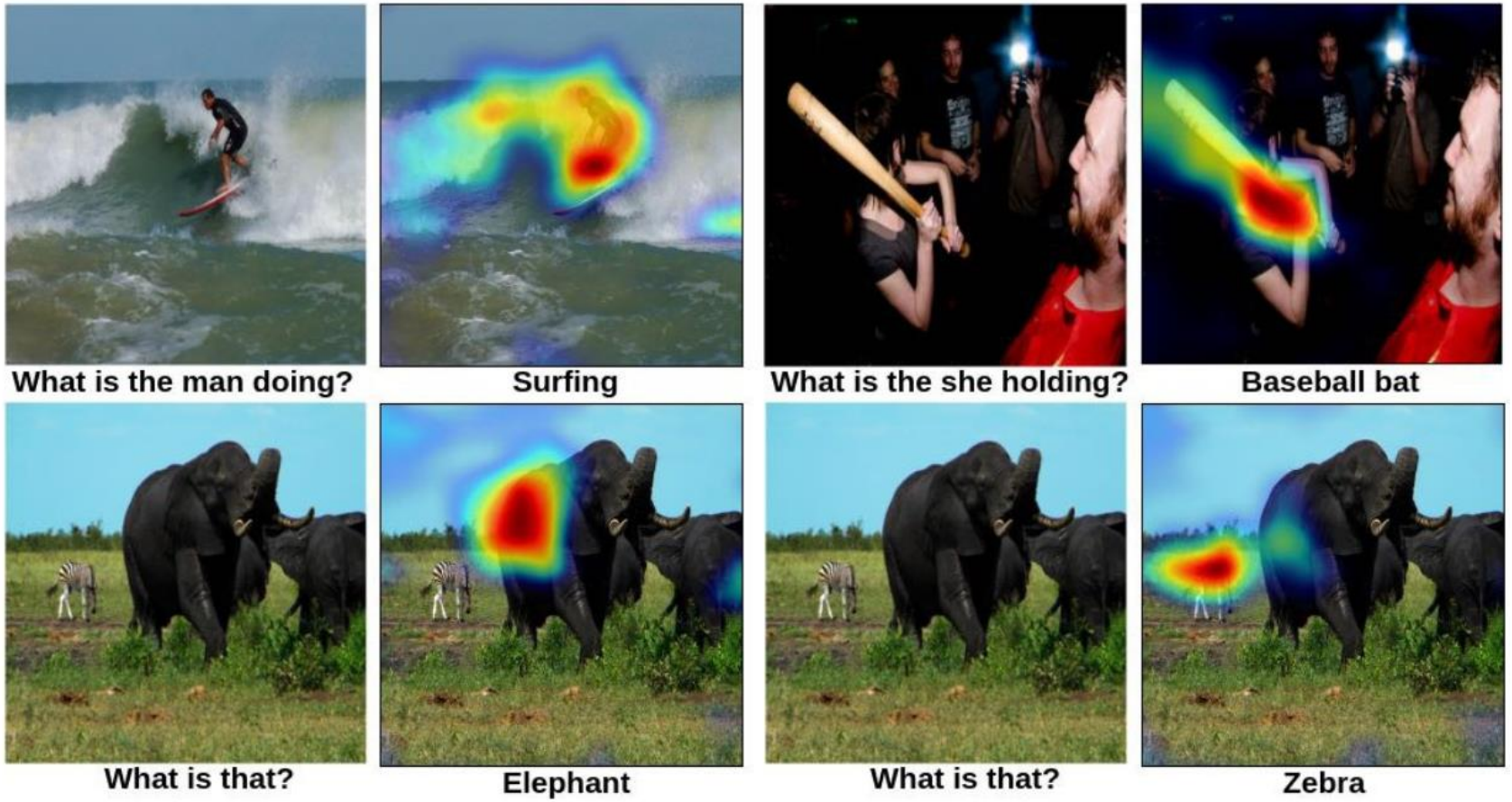
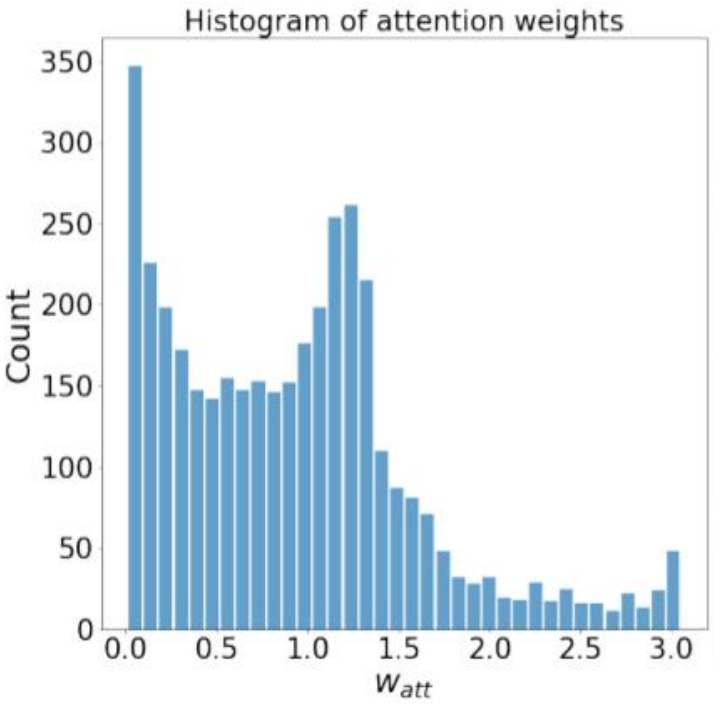


Image from [Selvaraju et al. 2019]

Visualizing Attention

- Certain neural network architectures include an **attention mechanism** that determines how much certain video frames or image regions should contribute towards the final decision.
- The primary goal is often to increase accuracy, but attention also provides **intrinsic interpretability.**



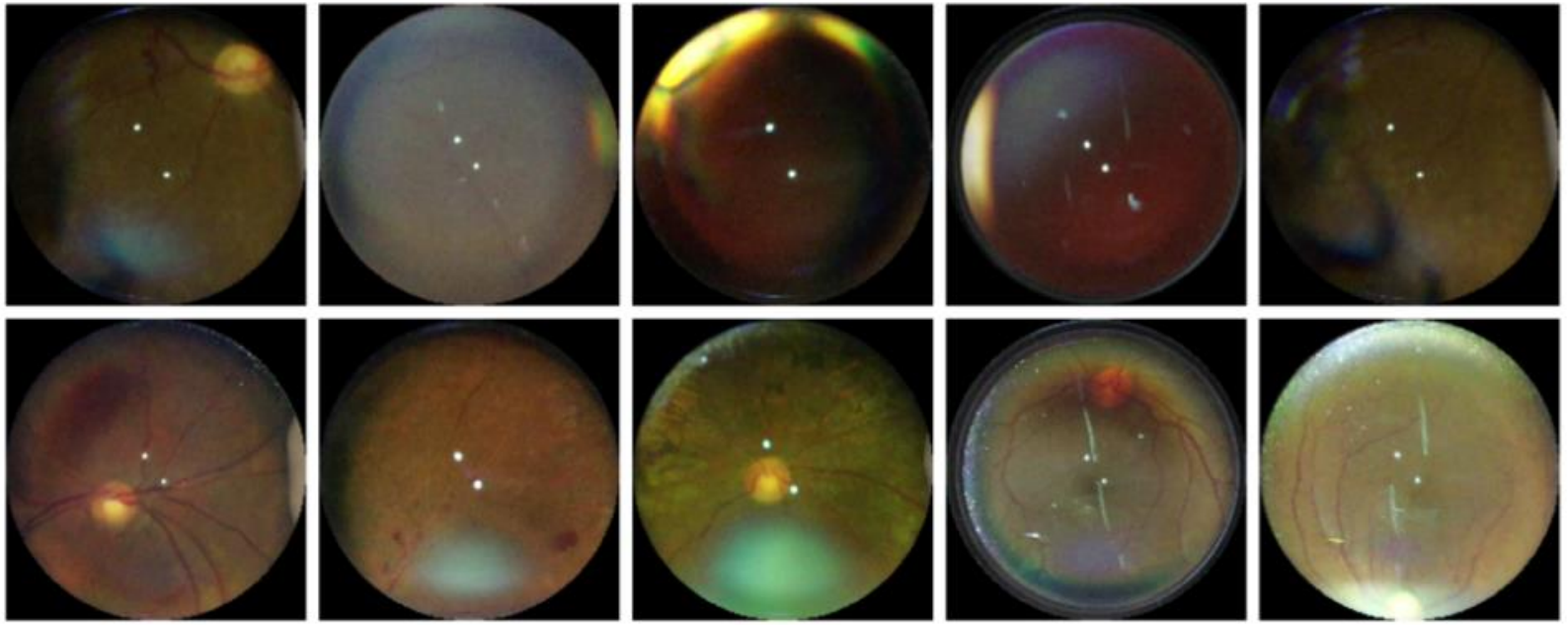
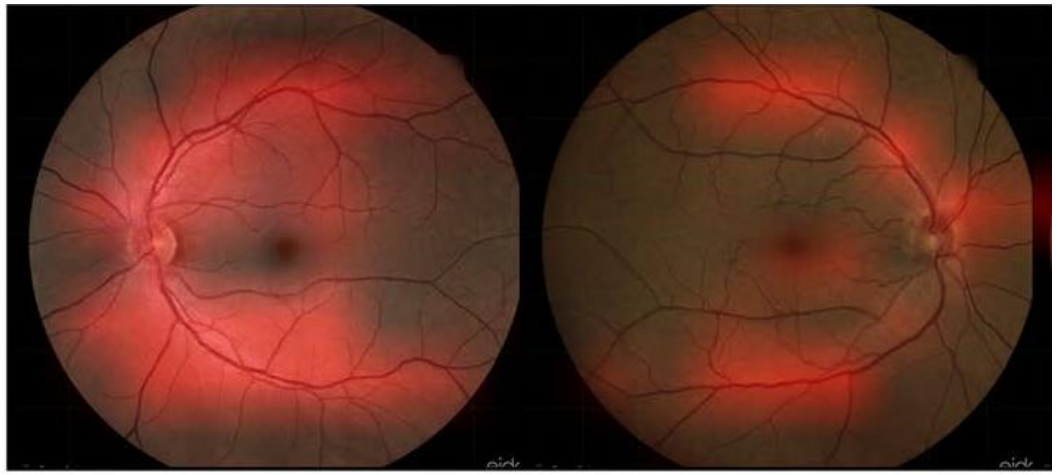
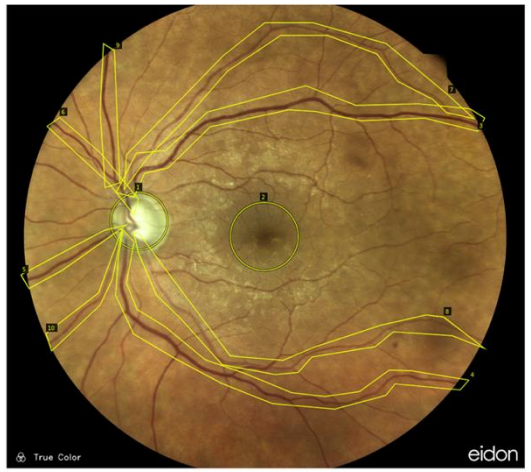


Image from [Mueller, Karpova et al., OMIA 2020]

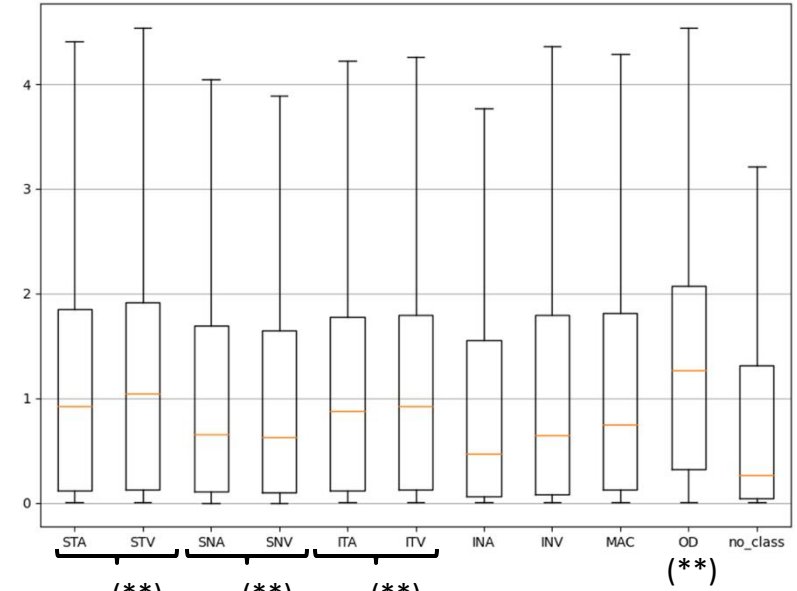
Attention for Interpretation

- [Mueller et al., SciRep 2022] detect Peripheral Arterial Disease from high-resolution color fundus photography
  - Attention mechanism focuses on relevant image regions
  - Highlights anatomical structures such as vessel arcades or optic disc





(a) Example of our anatomical annotations



(b) Boxplot of the attention weights

Summary: Visualizing CNNs

Model-agnostic approach to highlighting important regions:

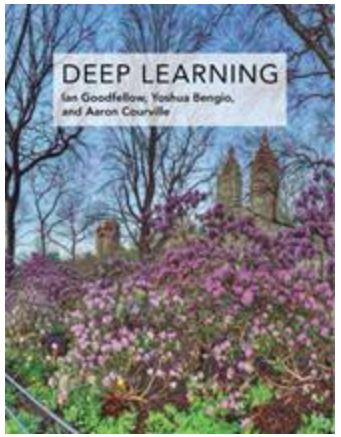
- **Occlusion sensitivity maps** show effect of hiding parts of the image
  - Explanation in terms of "is this image region required?"
  - Reveal decrease or increase in confidence

Two ideas for **building interpretability into neural networks:**

- **Class activation maps** rely on global average pooling of the final convolutional layer
  - **Gradient weighted class activation maps** (Grad-CAM) generalize CAMs to more general CNNs by backpropagating to the last convolutional layer
- **Attention mechanisms** imply a natural way of visualizing what the network focuses on

References

- Ian Goodfellow, Yoshua Bengio, Aaron Courville: "Deep Learning." MIT Press, 2016 Accessible online: https://www.deeplearningbook.org/
- Y. LeCun, L. Bottou, Y. Bengio, P. Haffner: Gradient-based learning applied to document recognition. Proc. of the IEEE 86(11):2278-2324, 1998
- A. Krizhevsky, I. Sutskever, G.E. Hinton: ImageNet Classification with Deep Convolutional Neural Networks. NIPS 2012
- K. Simonyan, A. Zisserman: Very Deep Convolutional Networks for Large-Scale Visual Recognition. ICLR 2015
- C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich: Going Deeper with Convolutions. CVPR 2015



More References

- K. He, X. Zhang, S. Ren, J. Sun: Deep Residual Learning for Image Recognition. CVPR 2016
- A. Veit, M. Wilber, S. Belongie: Residual Networks Behave Like Ensembles of Relatively Shallow Networks. NIPS 2016
- O. Ronneberger, P. Fischer, T. Brox: U-net: Convolutional networks for biomedical image segmentation. MICCAI 2015.
- S. Bach et al., “On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation.” PLOS ONE, 2015
- B. Zhou et al.: “Learning Deep Features for Discriminative Localization” CVPR 2016
- R. R. Selvaraju et al.: “Grad-CAM: Visual Explanations from Deep Networks via Gradient-based Localization” Int’l J. of Computer Vision, 2020