Convolutional Neural Networks (CNN)

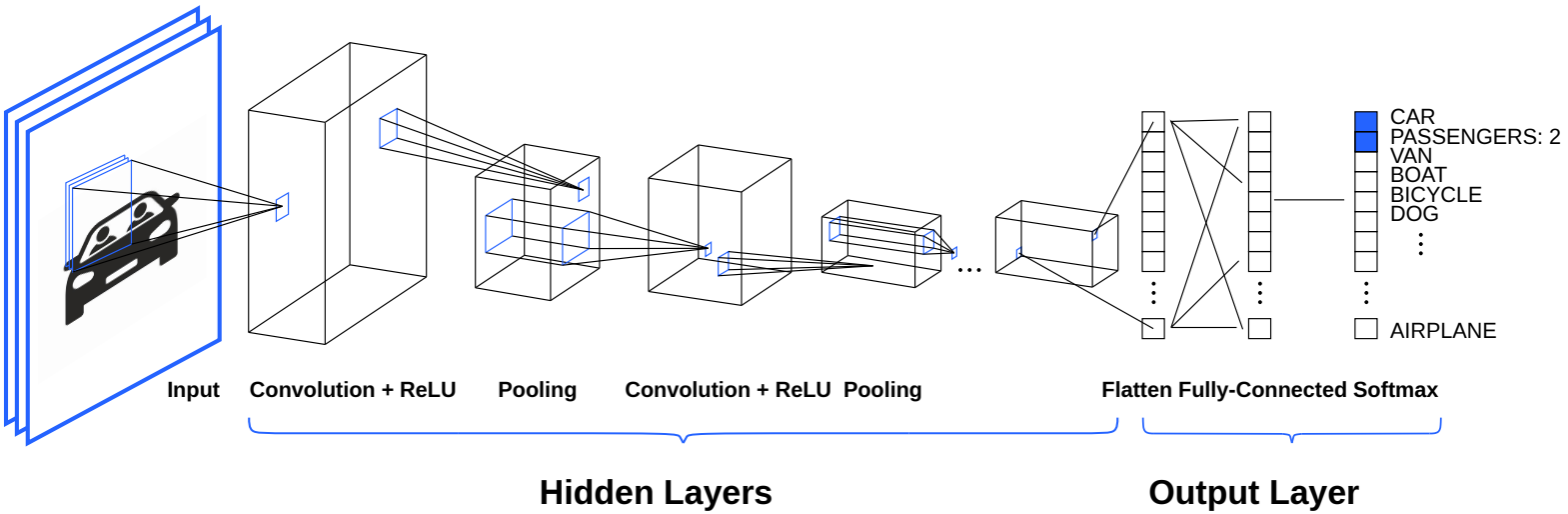
Regularization, Saliency Map and Interpretation

Nina Hernitschek
Centro de Astronomía CITEVA
Universidad de Antofagasta

August 9, 2023

Convolutional Neural Networks - Recap

A typical CNN architecture consists of alternating **Convolutional layers** and **Pooling layers** that extract features. Finally, their output is flattened into a one-dimensional feature layer that is passed into a Multi-Layer Perceptron (fully-connected layers) to obtain a prediction.



Convolutional Neural Networks - Recap

CNN include convolutional layers comprising learned filters that scan the outputs of the previous layer.

Convolutional Neural Networks - Recap

CNN include convolutional layers comprising learned filters that scan the outputs of the previous layer.

Pooling layers operate on groups of outputs from the convolutional layer to reduce the network size.

Convolutional Neural Networks - Recap

CNN include convolutional layers comprising learned filters that scan the outputs of the previous layer.

Pooling layers operate on groups of outputs from the convolutional layer to reduce the network size.

The parameters of the CNN can be learned through **backpropagation** with gradient descent.

Convolutional Neural Networks - Recap

Computing the Loss

Given a training set with input-output pairs:
$(x_1, \hat{y}_1), (x_2, \hat{y}_2), ..., (x_T, \hat{y}_T)$

The loss on the ith instance is $\Delta(x_i, \hat{y}_i)$.

The total loss is $E = \frac{1}{T} \sum_{i=1}^{T} \Delta(x_i, \hat{y}_i)$.

The loss is minimized w.r.t. the weights and biases θ

Convolutional Neural Networks - Recap

Computing the Loss

Given a training set with input-output pairs:
$(x_1, \hat{y}_1), (x_2, \hat{y}_2), ..., (x_T, \hat{y}_T)$

The loss on the ith instance is $\Delta(x_i, \hat{y}_i)$.

The total loss is $E = \frac{1}{T} \sum_{i=1}^{T} \Delta(x_i, \hat{y}_i)$.

The loss is minimized w.r.t. the weights and biases $\boldsymbol{\theta}$

Training through gradient descent

initialize all weights and biases $\boldsymbol{\theta}$

for every layer l for all filter indices m, until loss has converged update $\boldsymbol{\theta}_k \rightarrow \boldsymbol{\theta}_{k+1}$:

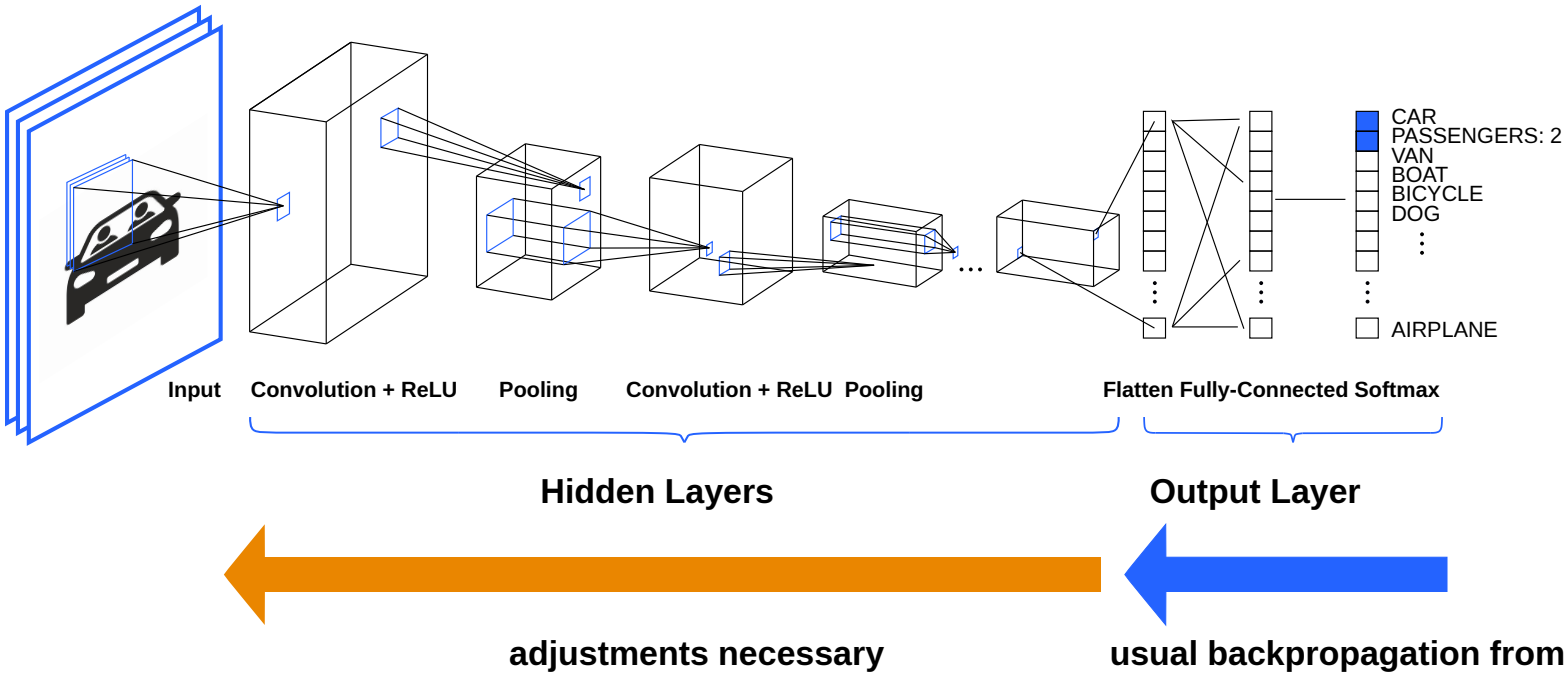
$$\boldsymbol{\theta}_{k+1}(l, m, j, x, y) = \boldsymbol{\theta}_k(l, m, j, x, y) - \eta \frac{\partial E}{\partial \boldsymbol{\theta}_k(l, m, j, x, y)}$$

Convolutional Neural Networks - Recap

Backpropagation continues in the usual manner until the computation of the derivative of the divergence w.r.t. its inputs to the first "flat" layer

Backpropagation from the flat MLP requires special consideration:

- the shared computation in the convolution layers
- the pooling layers



Convolutional Neural Networks - Recap

Both the Forward pass and the Backpropagation of a Convolutional layer are Convolutions.

So the gradients for backpropagation become, with filter F, layer output y_l and layer input $x = y_{l-1}$:

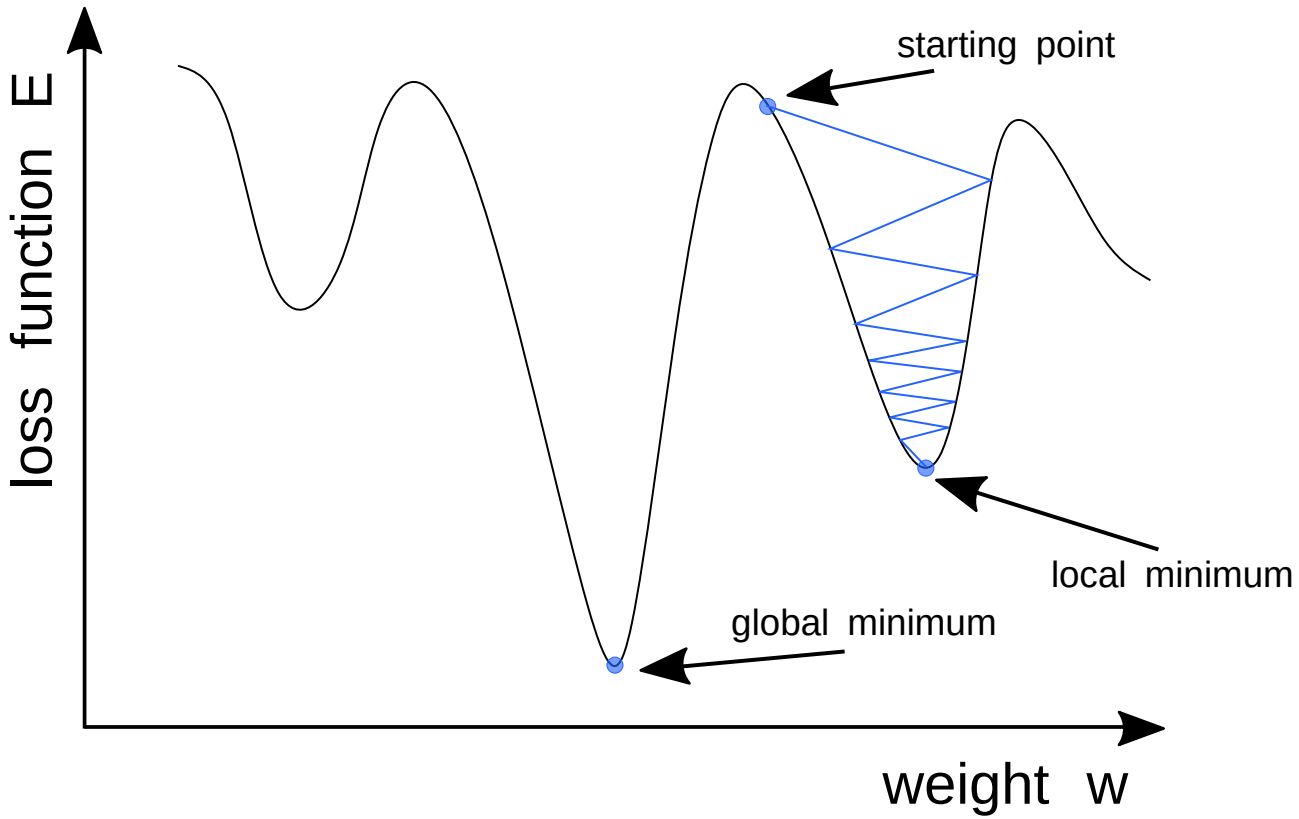
$$\frac{\partial E}{\partial F} = \text{Conv}\left(\text{input } x, \text{loss gradient } \frac{\partial E}{\partial y}\right)$$

$$\frac{\partial E}{\partial x} = \text{Conv}\left(180^\circ \text{ rotated filter } F', \text{ loss gradient } \frac{\partial E}{\partial y}\right)$$

Gradient Descent

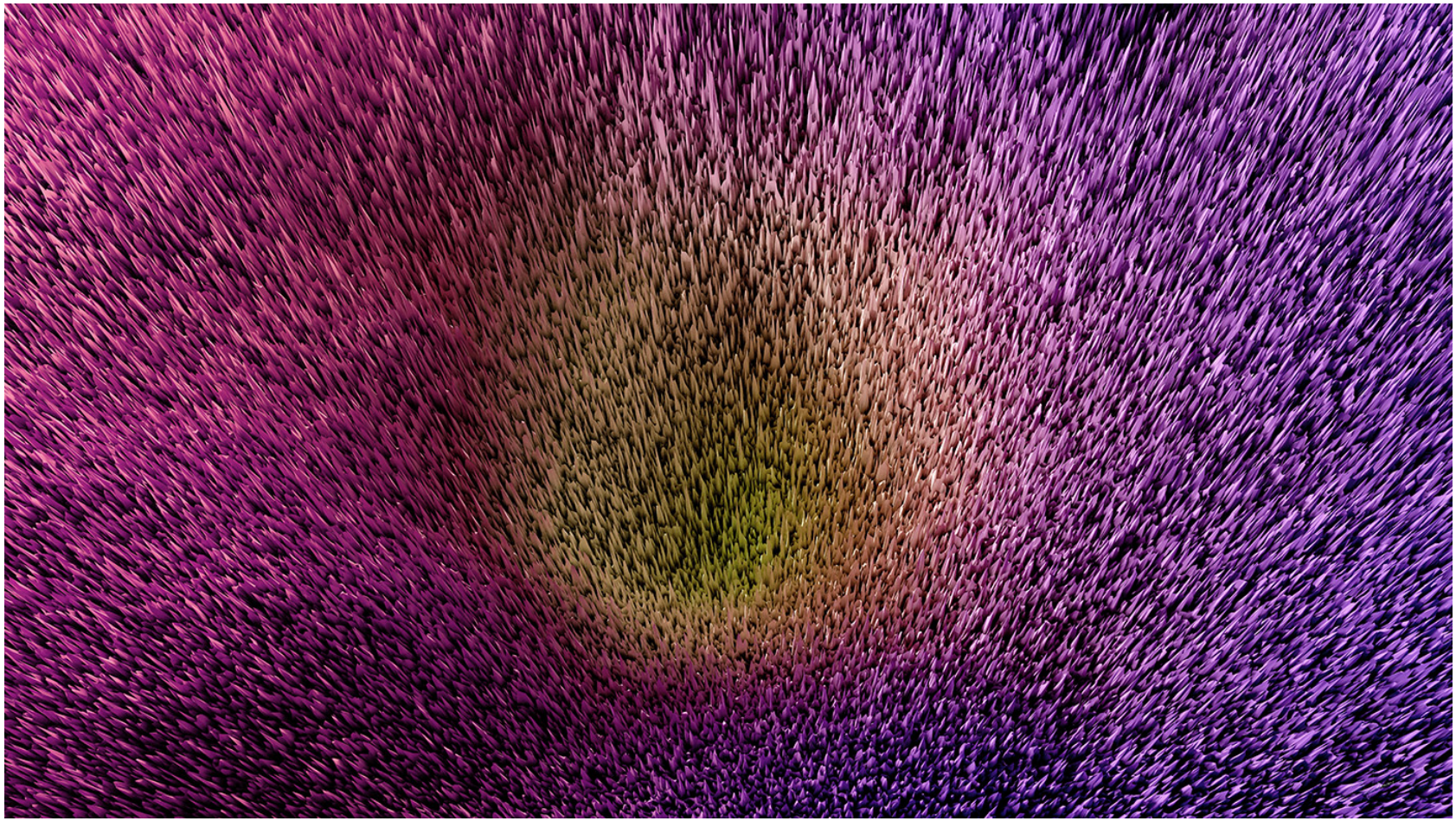
training neural networks is a **non-convex optimization problem**

this means we can run into many local minima during training



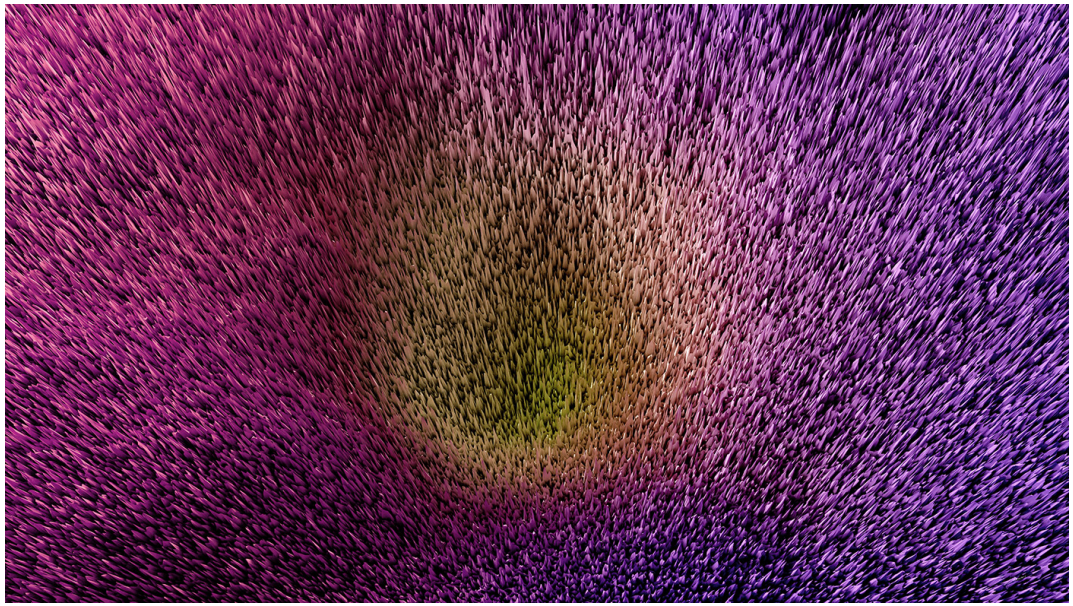
Gradient Descent

cost function (loss over the data set) can become very complex



Gradient Descent

cost function (loss over the data set) can become very complex



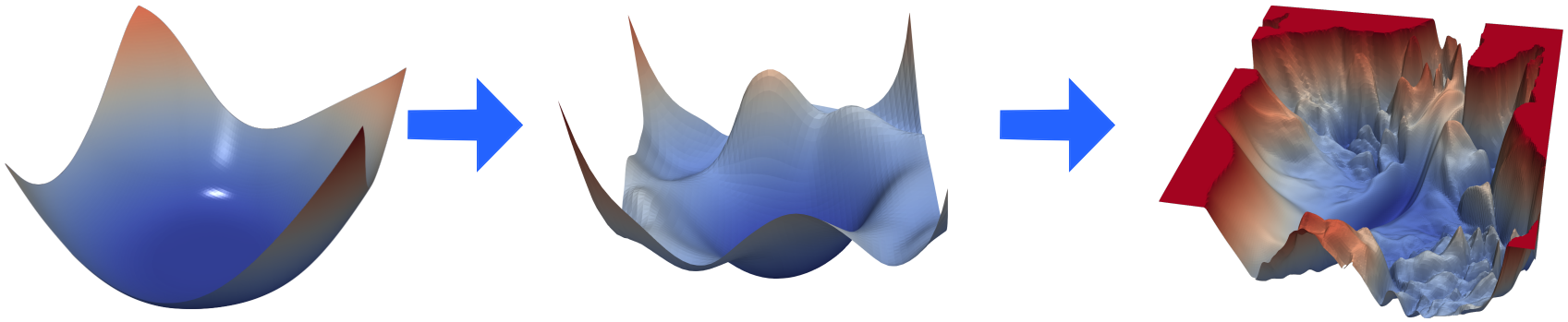
Loss Landscape created from the training process of a convolutional neural network.

credit:

`https://www.cs.umd.edu/~tomg/projects/landscapes/`

`https://losslandscape.com/gallery/`

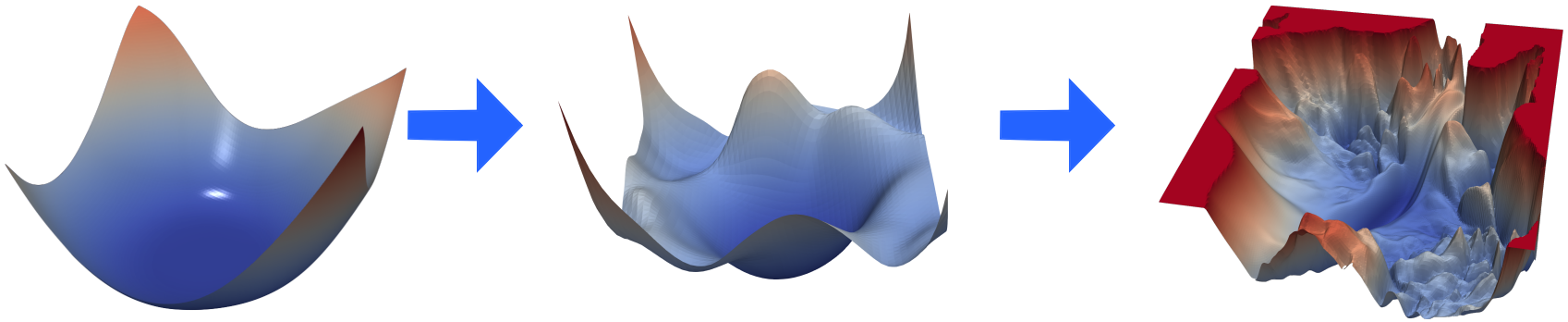
Gradient Descent



network architecture has a dramatic effect on the loss landscape:

Shallow networks have smooth landscapes populated by wide, convex regions. However, as networks become deeper, landscapes spontaneously become chaotic and highly non-convex, leading to poor training behavior.

Gradient Descent



network architecture has a dramatic effect on the loss landscape:

Shallow networks have smooth landscapes populated by wide, convex regions. However, as networks become deeper, landscapes spontaneously become chaotic and highly non-convex, leading to poor training behavior.

⇒ necessary to explore these landscapes

Gradient Descent

the simplest algorithm that attempts to minimize the loss $E(\theta)$ is Gradient Descent:

Gradient Descent

the simplest algorithm that attempts to minimize the loss $E(\boldsymbol{\theta})$ is Gradient Descent:

1. initialize the weights and biases $\boldsymbol{\theta}$ randomly.
2. update $\boldsymbol{\theta}_k \rightarrow \boldsymbol{\theta}_{k+1}$:

$$\mathbf{v}_k = \eta \nabla_\theta E(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

with the learning rate η and step k

repeat until the difference in values of E between two consecutive iterations goes below a pre-defined threshold

Gradient Descent

the simplest algorithm that attempts to minimize the loss $E(\boldsymbol{\theta})$ is Gradient Descent:

1. initialize the weights and biases $\boldsymbol{\theta}$ randomly.
2. update $\boldsymbol{\theta}_k \rightarrow \boldsymbol{\theta}_{k+1}$:

$$\mathbf{v}_k = \eta \nabla_\theta E(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

with the learning rate η and step k

repeat until the difference in values of E between two consecutive iterations goes below a pre-defined threshold

- requires a careful choice of learning rate
- sensitive to initial conditions
- can become stuck in local minima
- long time to escape saddle points
- learning rate is constant

Stochastic Gradient Descent

Problem with Gradient Descent: For a single update of each parameter, we go over N data points (summation $i = 0$ to N). Datasets involving thousands of features and millions of data points are not uncommon.

Stochastic Gradient Descent

Problem with Gradient Descent: For a single update of each parameter, we go over N data points (summation $i = 0$ to N). Datasets involving thousands of features and millions of data points are not uncommon.

Each update would require thousands of derivatives $\times$ millions of terms $\rightarrow \sim$ billion calculations per step

Stochastic Gradient Descent

Problem with Gradient Descent: For a single update of each parameter, we go over N data points (summation $i = 0$ to N). Datasets involving thousands of features and millions of data points are not uncommon.

Each update would require thousands of derivatives $\times$ millions of terms $\rightarrow \sim$ billion calculations per step

Solution:

In Stochastic Gradient Descent (SGD), stochasticity is added by approximating the gradient on a data subset called a **mini batch** (typically in the order of 10 - 100 data points)

full iteration using all mini batches of a dataset is called an epoch

update rule is now just

$$\mathbf{v}_k = \eta_k \nabla_\theta E_{mb}(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

Stochastic Gradient Descent

comparing SGD vs. GD

The main difference is the number of data points to go through before each update: which is 1 in case of SGD vs. all in case of GD.

SGD:

⊕
- decreases the chance of becoming stuck in local minima
- computation is fast as we take one data point at a time
- less overfitting as it is more generalized and randomized

⊖
- as considering only one data point for updating parameters, SGD is heavily affected by outliers - encountering an outlier requires extra updates to get back on track of convergence
- it takes much more time to converge as the gradient we get in SGD is not the true Gradient but just an approximation

Mini-Batch Gradient Descent

This variation is a **compromise** between both GD and SGD:

In mini-batch GD parameters are updated based on small samples of size $1 < b < N$. So instead of taking a single data point (case of SGD) or the complete dataset (case of GD), randomly b data points were taken from the data set for calculating the gradient and updating parameters.

Mini-Batch Gradient Descent

This variation is a **compromise** between both GD and SGD:

In mini-batch GD parameters are updated based on small samples of size $1 < b < N$. So instead of taking a single data point (case of SGD) or the complete dataset (case of GD), randomly b data points were taken from the data set for calculating the gradient and updating parameters.

⊕ Mini-Batch GD usually converges a little faster (in terms of iterations) than SGD as the gradient approximation in mini-batch GD is closer to the true gradient as compared to that in SGD.

⊖ As there are chances that data points can be repeated in different batches, mini-batch GD is more prone to overfitting as compared to SGD.

Mini-Batch Gradient Descent

This variation is a **compromise** between both GD and SGD:

In mini-batch GD parameters are updated based on small samples of size $1 < b < N$. So instead of taking a single data point (case of SGD) or the complete dataset (case of GD), randomly b data points were taken from the data set for calculating the gradient and updating parameters.

⊕ Mini-Batch GD usually converges a little faster (in terms of iterations) than SGD as the gradient approximation in mini-batch GD is closer to the true gradient as compared to that in SGD.

⊖ As there are chances that data points can be repeated in different batches, mini-batch GD is more prone to overfitting as compared to SGD.

The size of the mini-batch b depends on the size of the dataset. But generally, it should be around 1.5 times the number of outliers so that even in the worst case, it won't be affected much by outliers.

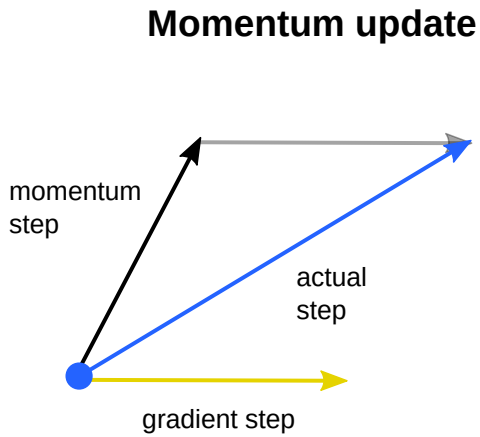
Stochastic Gradient Descent with Momentum

SGD with momentum adds an **inertia term** that retains some memory of the direction:

$$\mathbf{v}_k = \gamma \mathbf{v}_{k-1} + \eta_k \nabla_\theta E_m(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

with the momentum parameter $\gamma \sim 0.9$

SGD with momentum helps parameter updates gain speed in persistent smaller gradients while suppressing oscillatory high gradients



Stochastic Gradient Descent with Momentum

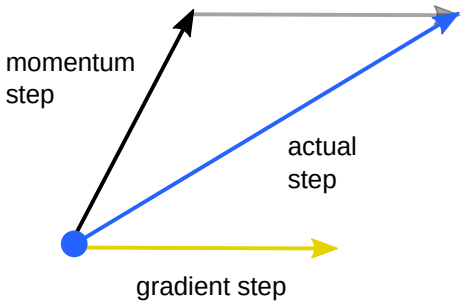
The **Nesterov accelerated momentum** (NAG) modifies this concept: First jump in the direction of the previous accumulated gradient, then measure the gradient where you end up and make a correction. This is done by performing the update using the partial derivative of the projected update rather than the derivative current variable value.

jump correction

$$\mathbf{v}_k = \gamma \mathbf{v}_{k-1} + \eta_k \nabla_\theta E_m(\boldsymbol{\theta}_k - \gamma \mathbf{v}_{k-1})$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

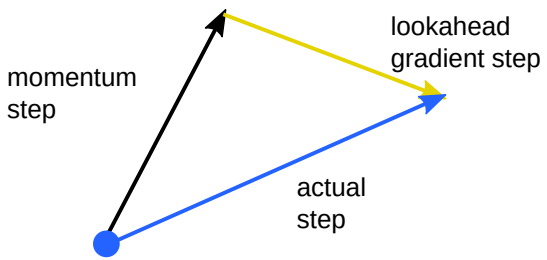
Momentum update

$\mathbf{v}_k = \gamma \mathbf{v}_{k-1} + \eta_k \nabla_\theta E_m(\boldsymbol{\theta}_k)$



Nesterov momentum update

$\mathbf{v}_k = \gamma \mathbf{v}_{k-1} + \eta_k \nabla_\theta E_m(\boldsymbol{\theta}_k - \gamma \mathbf{v}_{k-1})$



This anticipatory update results in increased responsiveness, which significantly increases the performance.

Gradient Descent - Summary

Batch gradient descent: All available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will become minimal.

Gradient Descent - Summary

Batch gradient descent: All available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will become minimal.

Stochastic gradient descent (SGD): In each iteration, the gradient is calculated for a single random sample only. While introducting the desired randomness, the disadvantage is its slowness, as much more iterations are necessary which can't be parallelized.

Gradient Descent - Summary

Batch gradient descent: All available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will become minimal.

Stochastic gradient descent (SGD): In each iteration, the gradient is calculated for a single random sample only. While introducting the desired randomness, the disadvantage is its slowness, as much more iterations are necessary which can't be parallelized.

(Stochastic) Mini-batch gradient descent: Instead of single samples, N random items are introduced on each iteration. This preserves the advantages of SGD but enables faster training due to parallelization.

Gradient Descent - Summary

Batch gradient descent: All available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will become minimal.

Stochastic gradient descent (SGD): In each iteration, the gradient is calculated for a single random sample only. While introducting the desired randomness, the disadvantage is its slowness, as much more iterations are necessary which can't be parallelized.

(Stochastic) Mini-batch gradient descent: Instead of single samples, N random items are introduced on each iteration. This preserves the advantages of SGD but enables faster training due to parallelization.

SGD with momentum: An inertia term that retains some memory of the direction helps parameter updates gain speed in persistent smaller gradients while suppressing oscillatory high gradients.

Gradient Descent - Summary

Batch gradient descent: All available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will become minimal.

Stochastic gradient descent (SGD): In each iteration, the gradient is calculated for a single random sample only. While introducting the desired randomness, the disadvantage is its slowness, as much more iterations are necessary which can't be parallelized.

(Stochastic) Mini-batch gradient descent: Instead of single samples, N random items are introduced on each iteration. This preserves the advantages of SGD but enables faster training due to parallelization.

SGD with momentum: An inertia term that retains some memory of the direction helps parameter updates gain speed in persistent smaller gradients while suppressing oscillatory high gradients.

Nesterov accelerated momentum (NAG): This modifies the concept of SGD with momentum. First a jump in the direction of the previous accumulated gradient is carried out, then the gradient is calculated and a correction carried out. This allows the search to slow down as approaching the optima, reducing the likelihood of missing or overshooting it.

Learning Rate

The learning rate η defines the size of the corrective steps that the model takes to adjust for errors in its output.

$$\mathbf{v}_k = \eta \nabla_\theta E(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

Learning Rate

The learning rate η defines the size of the corrective steps that the model takes to adjust for errors in its output.

$$\mathbf{v}_k = \eta \nabla_\theta E(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

A **high learning rate** shortens the training time, but with lower ultimate accuracy, while a **lower learning rate** takes longer, but with the potential for greater accuracy.

Learning Rate

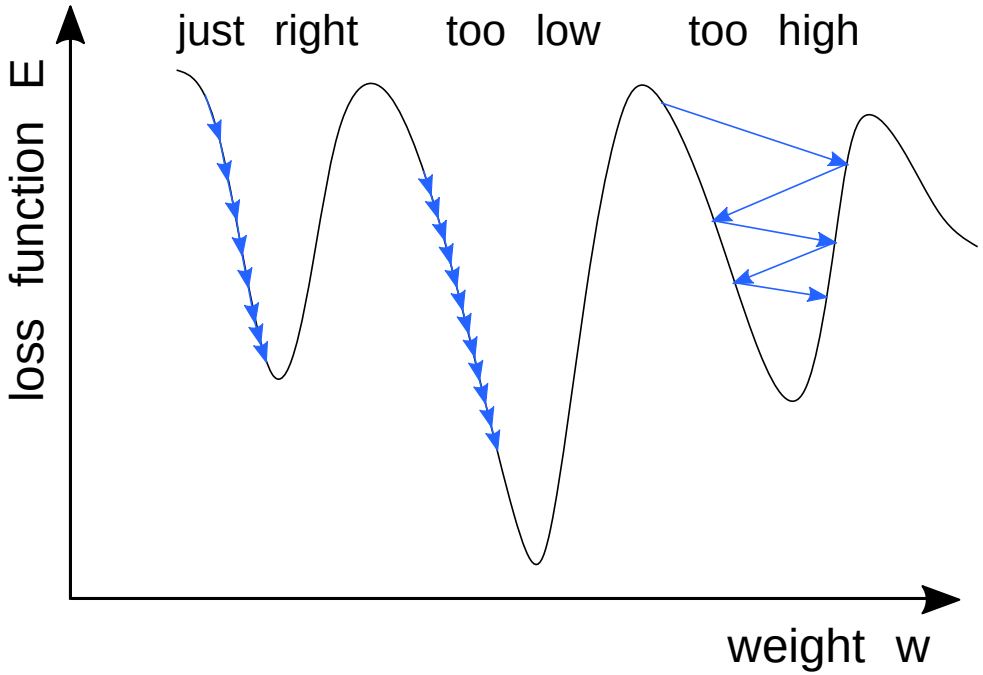
The learning rate η defines the size of the corrective steps that the model takes to adjust for errors in its output.

$$\mathbf{v}_k = \eta \nabla_\theta E(\boldsymbol{\theta}_k)$$
$$\boldsymbol{\theta}_{k+1} = \boldsymbol{\theta}_k - \mathbf{v}_k$$

A **high learning rate** shortens the training time, but with lower ultimate accuracy, while a **lower learning rate** takes longer, but with the potential for greater accuracy.

- To avoid oscillation inside the network such as alternating weights, and to improve the rate of convergence, an **adaptive learning rate** that increases or decreases as appropriate is used.
- In **reinforcement learning**, the aim is to weight the network to perform actions that minimize long-term (expected cumulative) cost.
- The concept of momentum allows the balance between the gradient and the previous change to be weighted such that the weight adjustment depends to some degree (characterized by the value of the momentum) on the previous change.

Learning Rate



optimal learning rate allows for smoothly reaching the maxima

too small learning rate requires many updates, hence learning is slow

too large learning rate leads to drastic updates with the possibility of overshooting the minima (diverging from it)

Vanishing and Exploding Gradients

As their names suggest, Vanishing and Exploding Gradients refer to the partial derivative of the loss function, trending towards zero and infinity respectively as they are back-propagated (multiplied) to earlier layers.

Vanishing and Exploding Gradients

Vanishing Gradients

During backpropagation, the gradients for the weight update are calculated using the chain rule relying on gradients of later layers:

$$\frac{\partial E}{\partial w_{11}^2} = \frac{\partial E}{\partial y}\frac{\partial y}{z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2} \quad \text{with} \quad \frac{\partial z_1^2}{\partial w_{11}^2} = \frac{\partial}{\partial w_{11}^2}(\varphi_1^1 w_{11}^2 + \varphi_2^1 w_{12}^2 + \varphi_3^1 w_{13}^2 + \varphi_4^1 w_{14}^2 + b_1^2)$$

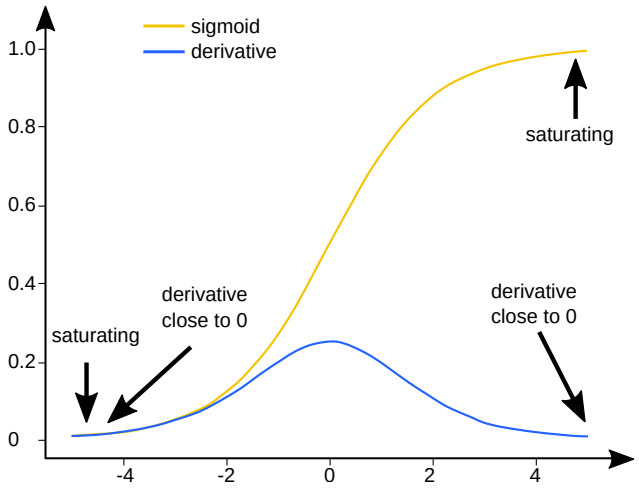
Vanishing and Exploding Gradients

Vanishing Gradients

During backpropagation, the gradients for the weight update are calculated using the chain rule relying on gradients of later layers:

$$\frac{\partial E}{\partial w_{11}^2} = \frac{\partial E}{\partial y}\frac{\partial y}{z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2} \quad \text{with} \quad \frac{\partial z_1^2}{\partial w_{11}^2} = \frac{\partial}{\partial w_{11}^2}(\varphi_1^1 w_{11}^2 + \varphi_2^1 w_{12}^2 + \varphi_3^1 w_{13}^2 + \varphi_4^1 w_{14}^2 + b_1^2)$$

As backpropagation advances from the output layer backwards, the gradients decrease until approaching zero. Updated weights get virtually identical to the old weights, making the model incapable of effective learning especially in the initial layers which learn the data's core features.



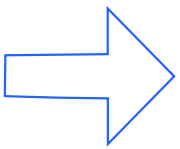
Vanishing and Exploding Gradients

Vanishing Gradients

During backpropagation, the gradients for the weight update are calculated using the chain rule relying on gradients of later layers:

$$\frac{\partial E}{\partial w_{11}^2} = \frac{\partial E}{\partial y}\frac{\partial y}{z_1^2}\frac{\partial z_1^2}{\partial w_{11}^2} \quad \text{with} \quad \frac{\partial z_1^2}{\partial w_{11}^2} = \frac{\partial}{\partial w_{11}^2}(\varphi_1^1 w_{11}^2 + \varphi_2^1 w_{12}^2 + \varphi_3^1 w_{13}^2 + \varphi_4^1 w_{14}^2 + b_1^2)$$

As backpropagation advances from the output layer backwards, the gradients decrease until approaching zero. Updated weights get virtually identical to the old weights, making the model incapable of effective learning especially in the initial layers which learn the data's core features.

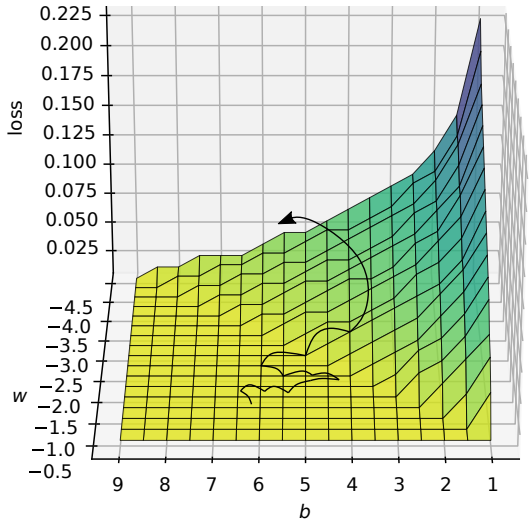


the deeper the neural network (greater number of Hidden Layers), the higher the probability of Vanishing/Exploding Gradients

Vanishing and Exploding Gradients

Exploding Gradients

On the contrary, in some cases the accumulation of large derivatives backwards results in the model being very unstable in training and achieving wildly fluctuating losses, as large changes in the weights creates a very unstable network. This is known as the exploding gradients problem.

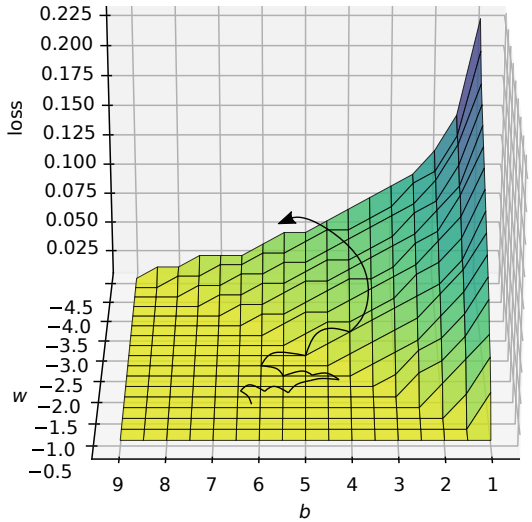


Vanishing and Exploding Gradients

Exploding Gradients

On the contrary, in some cases the accumulation of large derivatives backwards results in the model being very unstable in training and achieving wildly fluctuating losses, as large changes in the weights creates a very unstable network. This is known as the exploding gradients problem.

The problem arises from the initial weight assignment creating large losses. Gradient values can accumulate to the point where large parameter updates happen, causing gradient descent to oscillate without reaching a global minium. Even worse: the weights can get so large that they lead to overflows (NaN values) making further updates impossible.



Vanishing and Exploding Gradients

Following are some signs that can indicate that gradients are vanishing or exploding:

vanishing gradients:

- The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all).
- The model weights may become 0 during training.
- The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations.

Vanishing and Exploding Gradients

Following are some signs that can indicate that gradients are vanishing or exploding:

vanishing gradients:

- The parameters of the higher layers change significantly whereas the parameters of lower layers would not change much (or not at all).
- The model weights may become 0 during training.
- The model learns very slowly and perhaps the training stagnates at a very early stage just after a few iterations.

exploding gradients:

- There is an exponential growth in the model parameters.
- The model weights may become NaN during training.
- The model experiences avalanche learning.

Vanishing and Exploding Gradients

Fortunately there are some techniques to overcome these issues, apart from using suitable activation functions.

Weight Initialization

Random weight initialization sometimes pose a problem by the very first activation values produced may be slightly too large or too small, planting the seed for vanishing or exploding gradients later on in training.

Weight Initialization

Random weight initialization sometimes pose a problem by the very first activation values produced may be slightly too large or too small, planting the seed for vanishing or exploding gradients later on in training.

Xavier (or Glorot) Initialization

For the proper flow of the signal, X. Glorot & Y. Benigo (2010) argue that:

- The variance of outputs of each layer should be equal to the variance of its inputs.
- The gradients should have equal variance before and after flowing through a layer in the reverse direction.

Weight Initialization

Random weight initialization sometimes pose a problem by the very first activation values produced may be slightly too large or too small, planting the seed for vanishing or exploding gradients later on in training.

Xavier (or Glorot) Initialization

For the proper flow of the signal, X. Glorot & Y. Benigo (2010) argue that:

- The variance of outputs of each layer should be equal to the variance of its inputs.
- The gradients should have equal variance before and after flowing through a layer in the reverse direction.

It is impossible for both conditions to hold for any layer unless the number of inputs to that layer (n_{in}) equals the numbers of neurons in that layer (n_{out}). However, the following compromise works incredibly well in practice: randomly initialize the connection weights for each layer in the network as described in the following.

Weight Initialization

Xavier (or Glorot) Initialization

Suppose we have an input x with n components and a neuron with random weights w and output y.

We know that the variance of $w_i x_i$ is, with expectation value $\mathrm{E}(\cdot)$:

$$\mathrm{Var}(w_i x_i) = \mathrm{E}(x_i)^2 \mathrm{Var}(w_i) + \mathrm{E}(w_i)^2 \mathrm{Var}(x_i) + \mathrm{Var}(w_i)\mathrm{Var}(x_i)$$

We assume that x_i, w_i are identically and independently distributed and get

$$\begin{aligned}\mathrm{Var}(y) &= \mathrm{Var}(w_1 x_1 + ... + w_n x_n) \\ &= \mathrm{Var}(w_1 x_1) + ... + \mathrm{Var}(w_n x_n) \\ &= n\mathrm{Var}(w_i)\mathrm{Var}(x_i)\end{aligned}$$

The variance of the output is the variance of the input scaled by $n\mathrm{Var}(w_i)$. For the variance of y to be equal to the variance of x, $n\mathrm{Var}(w_i)$ should be equal to 1 $\Rightarrow$ the variance of the weight should be $\mathrm{Var}(w_i) = 1/n_{in}$.
with expectation value $\mathrm{E}(\cdot)$.
Similarly, if we go through backpropagation, we apply the same steps and get $\mathrm{Var}(w_i) = 1/n_{out}$.

Weight Initialization

Xavier (or Glorot) Initialization

In order to keep the variance of the input and the output gradient the same, $n_{in} = n_{out}$ must hold. As a compromise, Glorot and Bengio suggest using the average of the n_{in} and n_{out}, proposing that:

$$\mathrm{Var}(w_i) = 1/n_{avg} = \frac{2}{(n_{in} + n_{out})}.$$

This is Xavier Initialization formula to be applied to all weights of a given layer.

Weight Initialization

Xavier (or Glorot) Initialization

In order to keep the variance of the input and the output gradient the same, $n_{in} = n_{out}$ must hold. As a compromise, Glorot and Bengio suggest using the average of the n_{in} and n_{out}, proposing that:

$$\mathrm{Var}(w_i) = 1/n_{avg} = \frac{2}{(n_{in} + n_{out})}.$$

This is Xavier Initialization formula to be applied to all weights of a given layer.

He Initialization

Glorot and Bengio considered a sigmoid activation function, the default choice at that moment. Later on, the sigmoid activation was surpassed by ReLu, because it allowed to solve vanishing / exploding gradients problem. However, it turns out Xavier (Glorot) Initialization is not quite as optimal for ReLU functions. The He initialization solves this with the adjustment of multiplying the variance from the Xavier Initialization by a factor of 2.

Regularization

weight initialization is ony part of the solution:

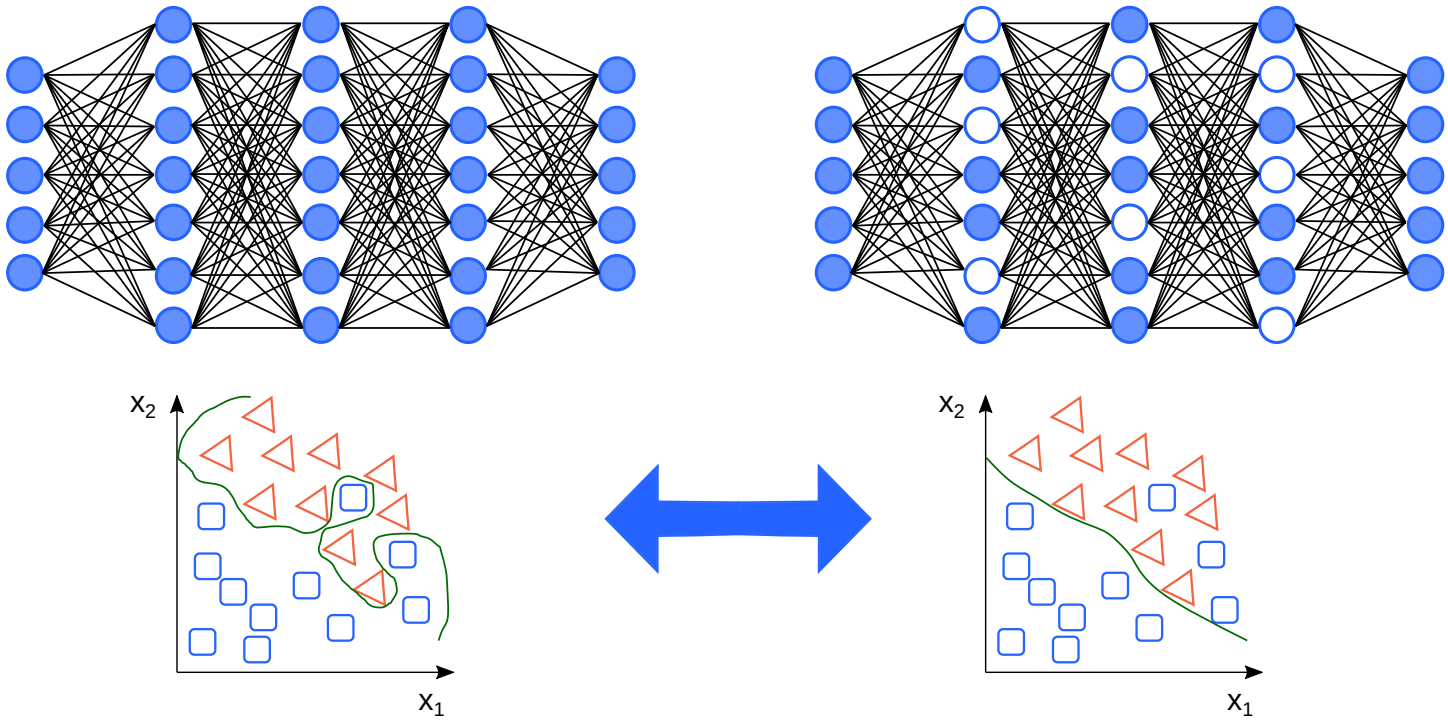
prevent overfitting by limiting the capacity of a neural network with **regularization techniques**

- limit the number of hidden units
- limit the size of weights $\Rightarrow$ **weight decay**
- stop the learning before it has time to overfit $\Rightarrow$ **early stop**

$$E(\theta) = \underbrace{\frac{1}{m}\sum_{i=1}^{m} E_i(y, \hat{y})}_{\text{Data loss: model prediction should match training data}} + \underbrace{\lambda R(\theta)}_{\text{Regularization: model should be simple so it works on test data}}$$

Regularization

Dropout turns off random neurons with probability p for each mini-batch during training (forward pass).
This forces the network to have redundant representation, preventing co-adaptation for features.



model has too many free parameters and over-fits

too many neurons dropped out leading to underfitting

Regularization

Weight decay, or L_2 **regularization**, is a regularization technique applied to the weights of a neural network. We minimize a loss function compromising both the primary loss function and a **penalty term**:

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

with the weight decay parameter $\lambda \sim 0.1$.

Regularization

Weight decay, or L_2 **regularization**, is a regularization technique applied to the weights of a neural network. We minimize a loss function compromising both the primary loss function and a **penalty term**:

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

with the weight decay parameter $\lambda \sim 0.1$.

This extra term penalizes the squared weights by keeping weights small unless they have big error derivatives.

Regularization

Weight decay, or L_2 **regularization**, is a regularization technique applied to the weights of a neural network. We minimize a loss function compromising both the primary loss function and a **penalty term**:

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

with the weight decay parameter $\lambda \sim 0.1$.

This extra term penalizes the squared weights by keeping weights small unless they have big error derivatives.

A modification: L_1 **regularization** using $|w_i|$ instead of w_i^2.

Weight decay can be incorporated directly into the weight update rule, rather than just implicitly by defining it through to error function.

Regularization

Weight decay, or L_2 **regularization**, is a regularization technique applied to the weights of a neural network. We minimize a loss function compromising both the primary loss function and a **penalty term**:

$$C = E + \frac{\lambda}{2} \sum_i w_i^2$$

$$\frac{\partial C}{\partial w_i} = \frac{\partial E}{\partial w_i} + \lambda w_i$$

with the weight decay parameter $\lambda \sim 0.1$.

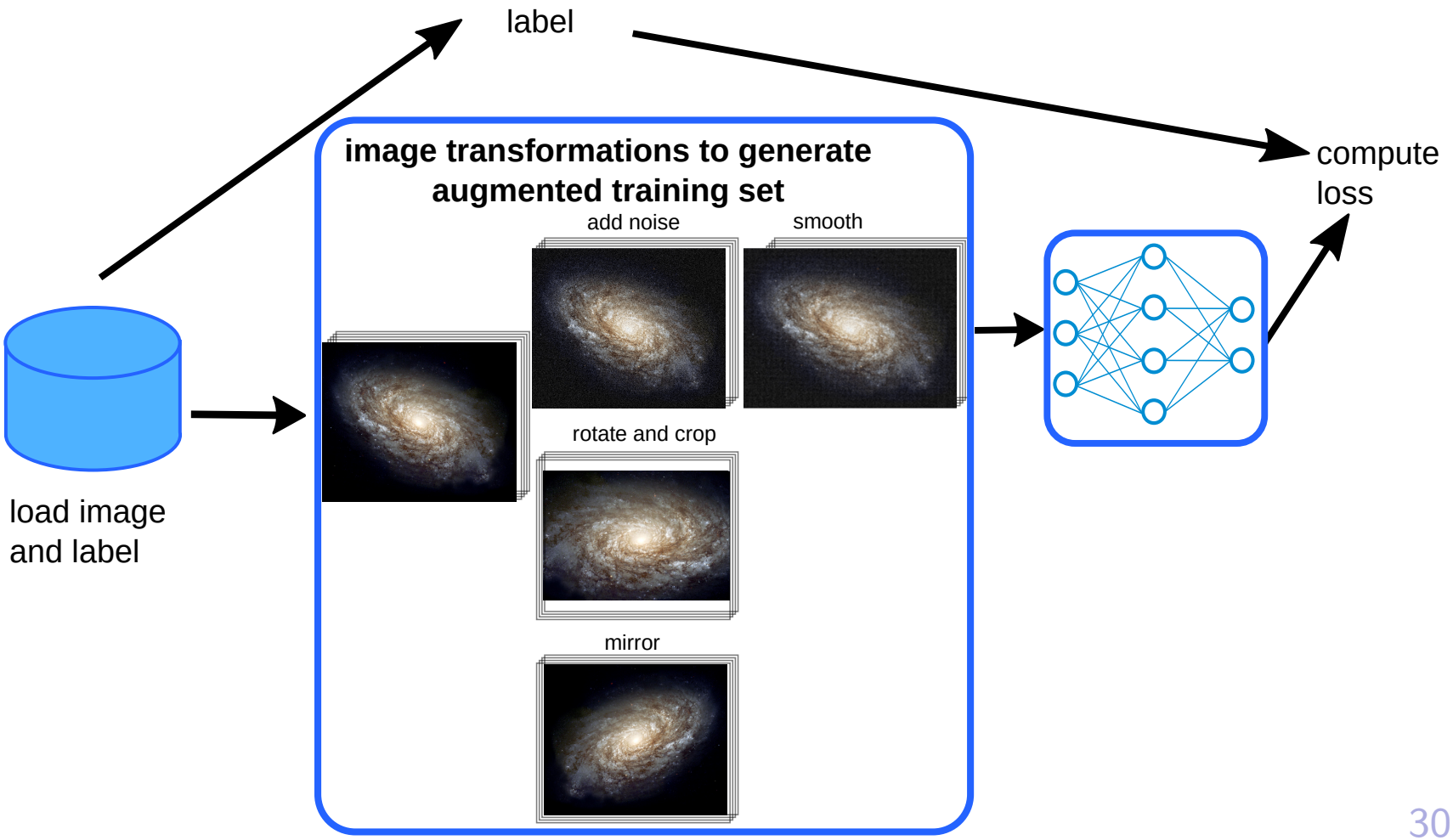
benefits of using weight decay:

- Improved generalization performance of the model by preventing overfitting.
- Reduced complexity of the model by encouraging a sparser learned representation.
- Improved convergence of te optimizer by encouraging smaller weights.

The beauty of weight decay is that the algorithm automatically decides which neurons to knock out in order to lower the complexity.

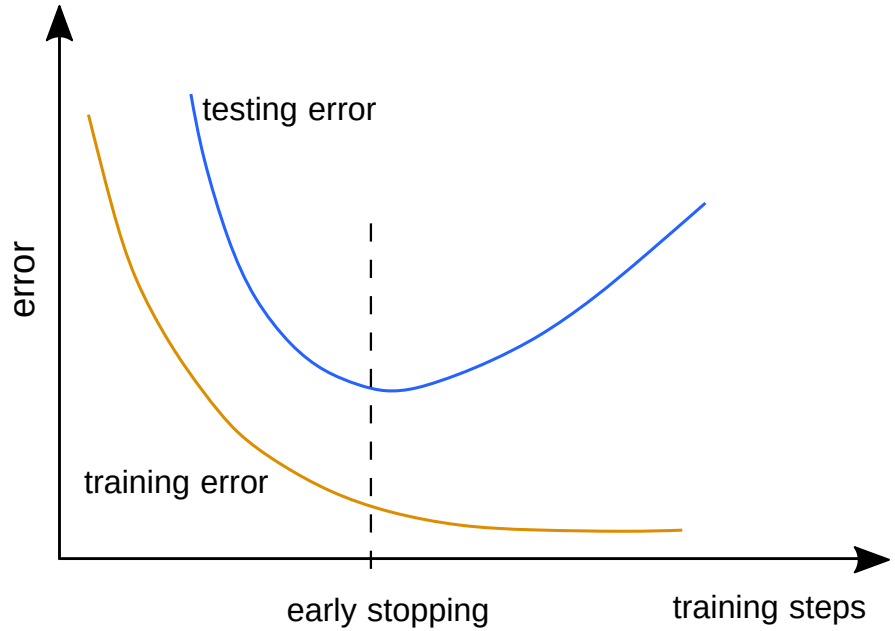
Regularization

the regularization technique of **data augmentation** adds randomness in training



Regularization

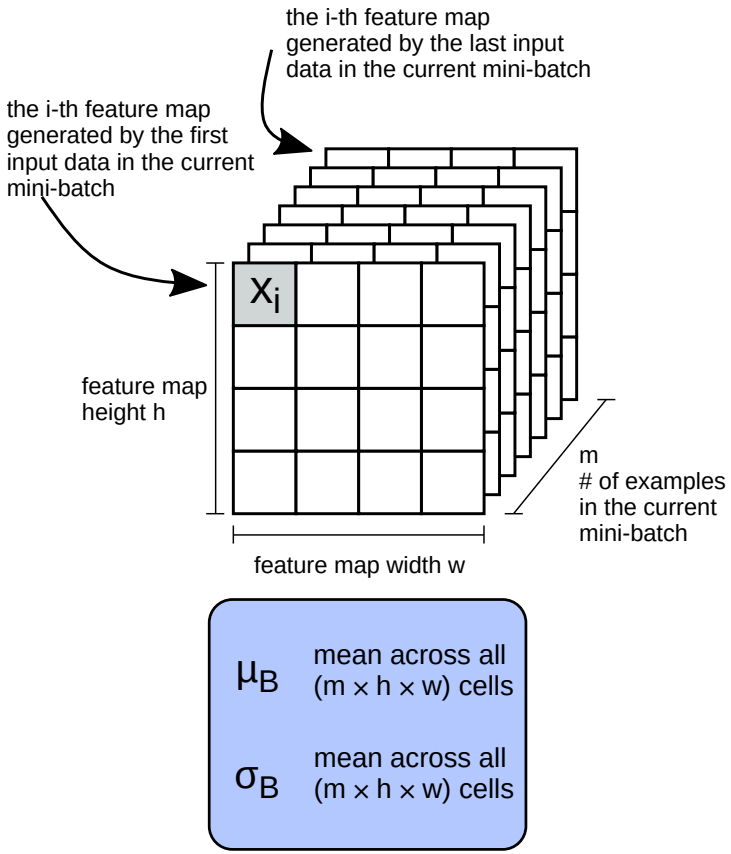
Early stopping is a cross-validation strategy where we keep one part of the training set as the development set. When the performance on the development set is getting worse, we immediately stop the training.



The idea of Early Stopping is to retrieve the parameter values (weights and biases) after stopping and using them as these are the values for which the model performs best.
Because the best fit is biased towards the development set, the model should then be evaluated on the test set to obtained an impartial result.

Regularization

Batch Normalization makes the activations Gaussian. This technique by Ioffe & Szegedy (2015) helps with improving the gradient flow through the network, allowing higher learning rates. Also, the strong dependence on parameter initialization is reduced.



input: values x of a mini-batch $B = \{x_1, ..., x_m\}$

output: $\{\hat{y}_i = BN_{\gamma,\beta}(x_i)\}$, parameters to be learned: γ, β

algorithm:

mini-batch mean $\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$

mini-batch variance $\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_B)^2$

normalize $\tilde{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}$ where ϵ is added in the denominator for numerical stability and is an arbitrarily small constant

scale and shift $\hat{y}_i \leftarrow \gamma\tilde{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i)$

Regularization

Why does **Batch Normalization** work?

Regularization

Why does **Batch Normalization** work?

One benefit of batch normalization is that it acts as **regularization**: Each mini-batch is scaled using its mean and standard deviation, thus introducing some noise to each layer, providing a regularization effect.

Regularization

Why does **Batch Normalization** work?

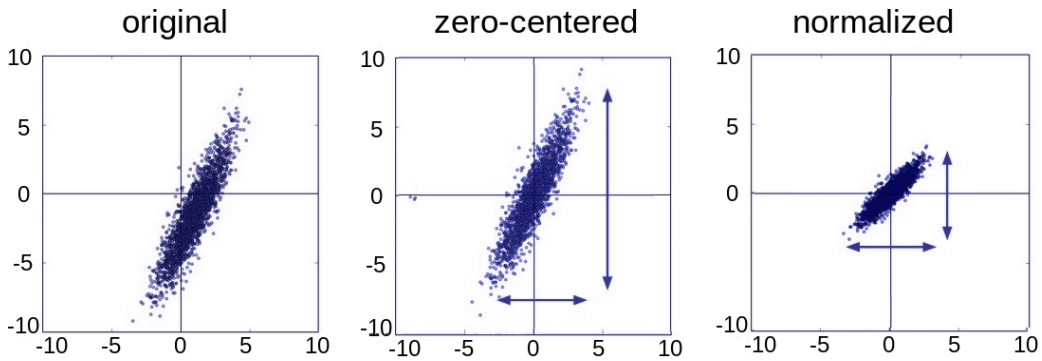
One benefit of batch normalization is that it acts as **regularization**: Each mini-batch is scaled using its mean and standard deviation, thus introducing some noise to each layer, providing a regularization effect.

The other benefit comes from the idea of **covariante shift**:

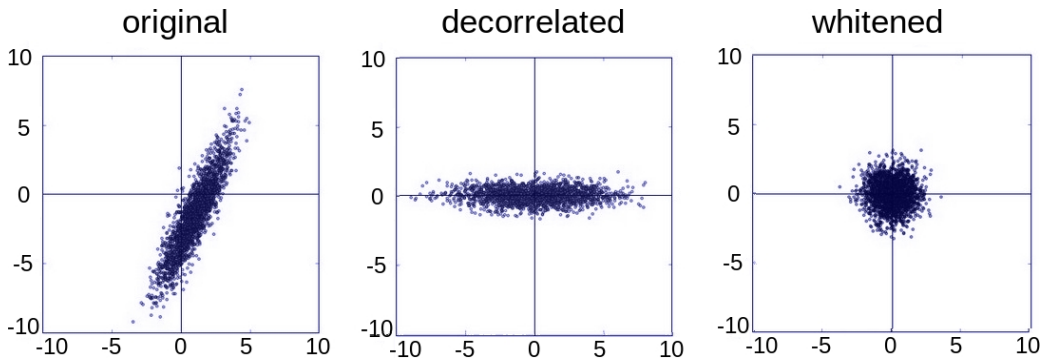
Suppose your training set for galaxy classification contains no highly redshifted galaxies.
The learned model will not perform well on realistic survey data from deep surveys. The reason is the shift in the input distribution, known as the covariate shift. Batch normalization is reducing this covariate shift.
The idea is that even when the exact values of inputs to hidden layers change, their mean and standard deviation will still almost remain same thus reducing the covariate shift. This weakens the coupling between parameters of early layer and that of later layers hence, allowing each layer of the network to learn by itself i.e. more independent of each other. This has the effect of speeding up the learning process.

Summary: The Complete Training Process

Step 1: Data Preprocessing



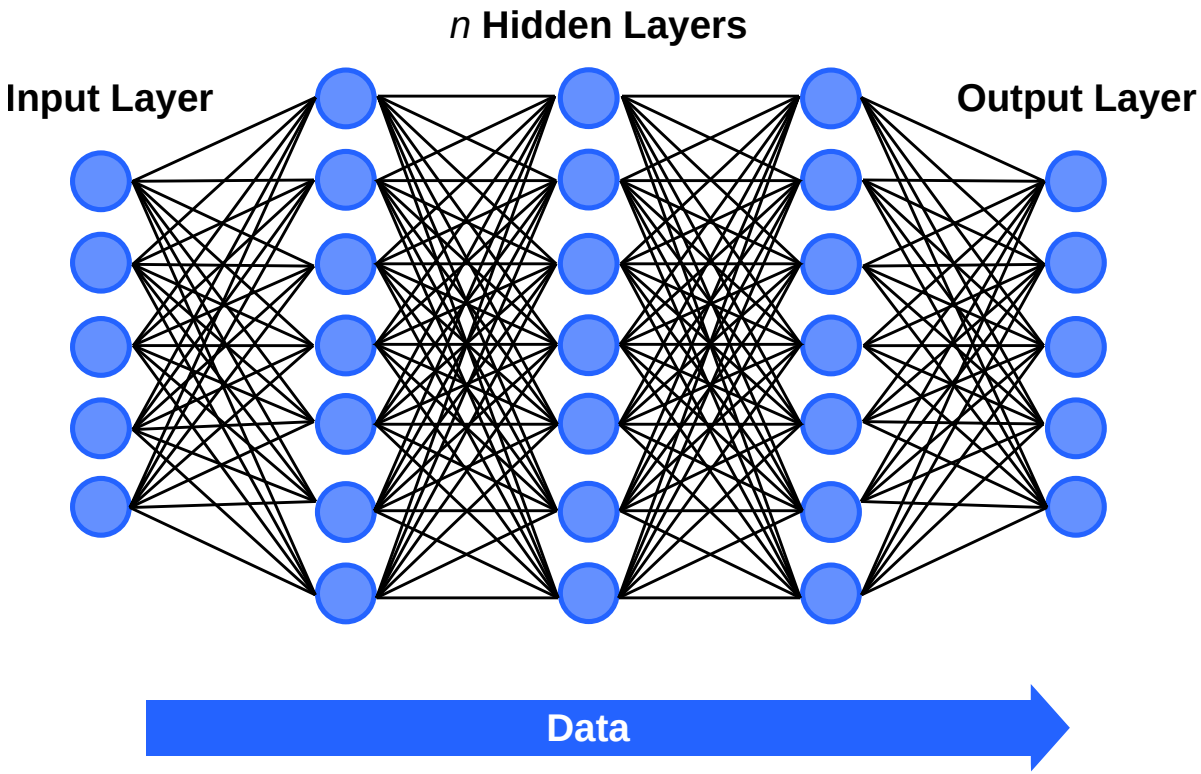
Common data preprocessing pipeline. Left: Original 2D input data.
Center: Zero-centering the data by subtracting the mean in each dimension.
Right: Additionally scaling by the dimension's standard deviation gives equal extends.



Data preprocessing pipeline with PCA / Whitening. Left: Original 2D input data.
Center: After performing PCA the data is centered at zero and decorrelated by being rotated into the eigenbasis of the data covariance matrix. Right: Additionally scaling by the eigenvalues stretches and squeezes the data into an isotropic Gaussian.

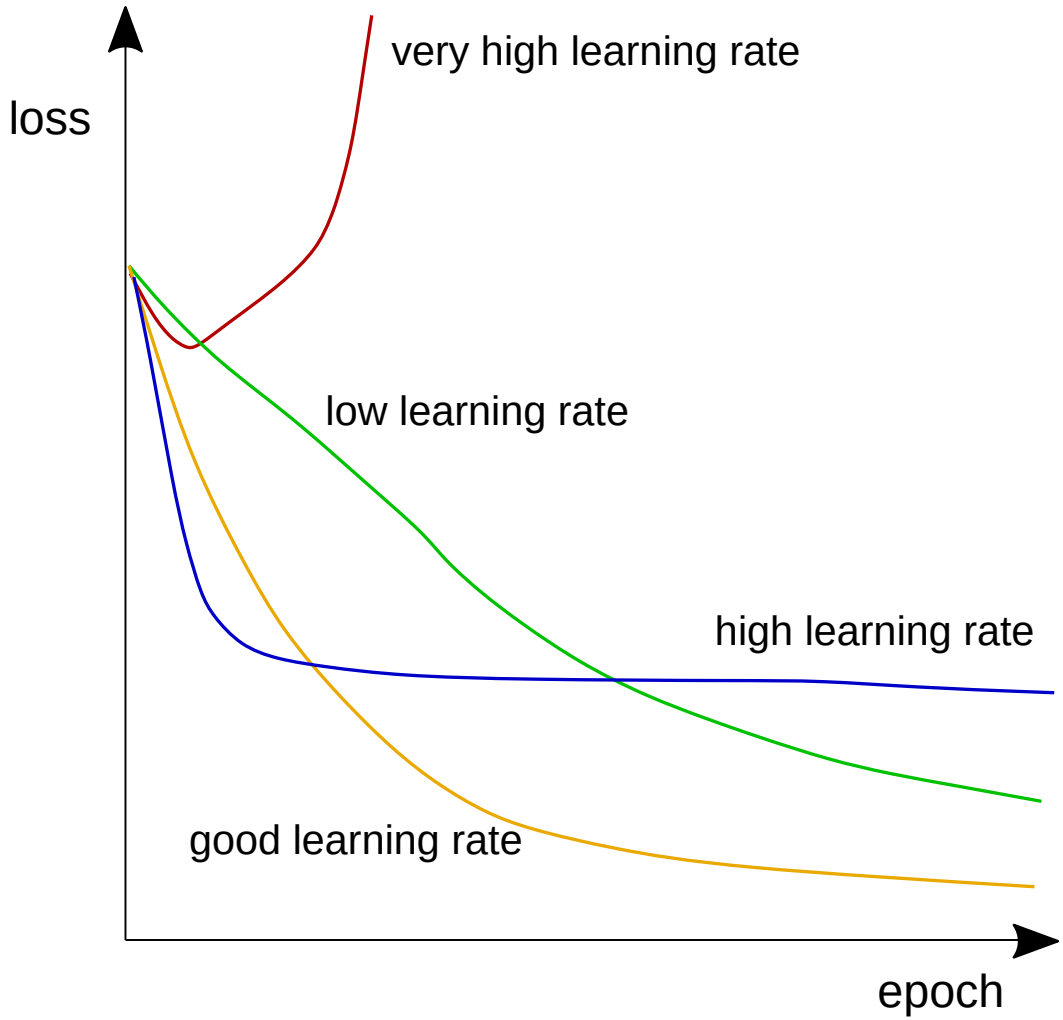
Summary: The Complete Training Process

Step 2: Chosing the Neural Network Architecture



Summary: The Complete Training Process

Step 3: Check the Loss



Opening The Deep Learning Blackbox

CNNs have led to great improvement in computer vision.

Why does the CNN predict a particular label? What did it see in the image that led to a particular prediction?

"bird"

credit: https://www.jjharrison.com.au/

Opening The Deep Learning Blackbox

CNNs have led to great improvement in computer vision.

Why does the CNN predict a particular label? What did it see in the image that led to a particular prediction?

"bird"

credit: https://www.jjharrison.com.au/

We can answer these questions through **Saliency Maps**.

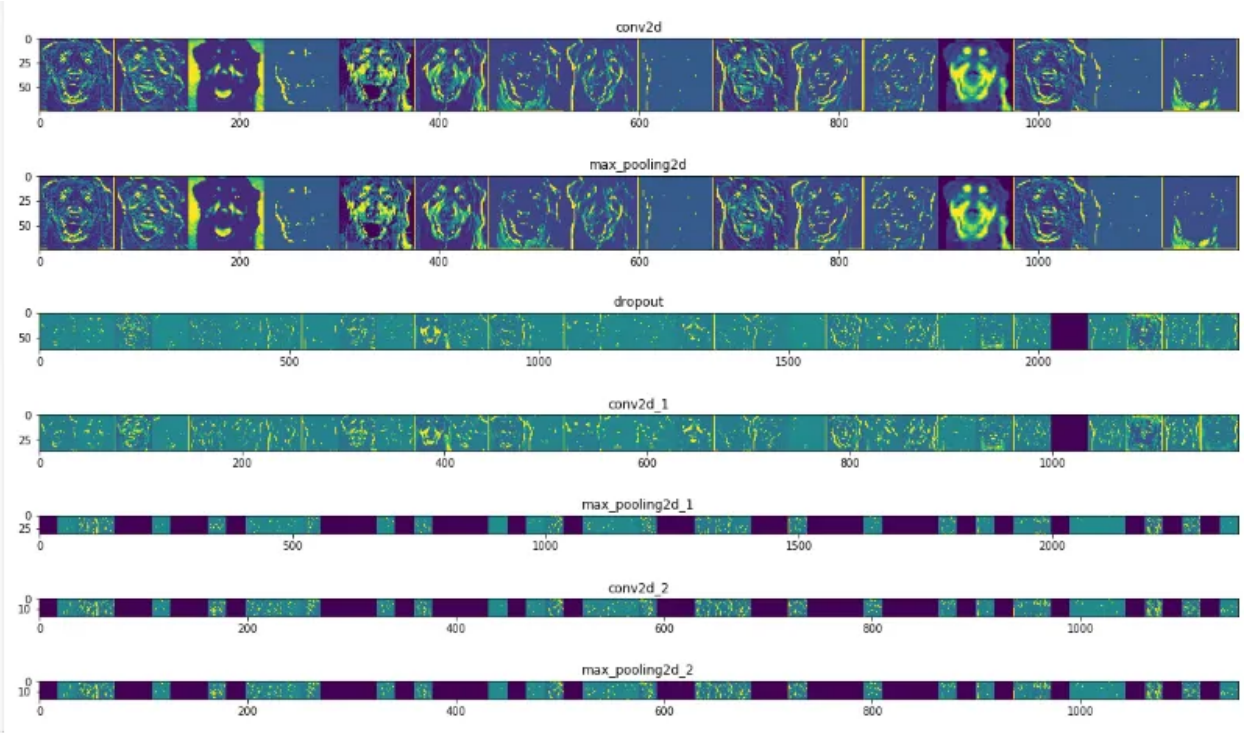
Visualizing the CNN

Visualizing the network is useful to diagnose problems with the models, to interpret our models' meaningfulness, or simply to teach deep learning concepts.

We can visualize decision boundaries, weights of our model, activations, gradients, performance metrics, the result of applying a learned filter to an image, or the filters themselves.

Visualizing Filters

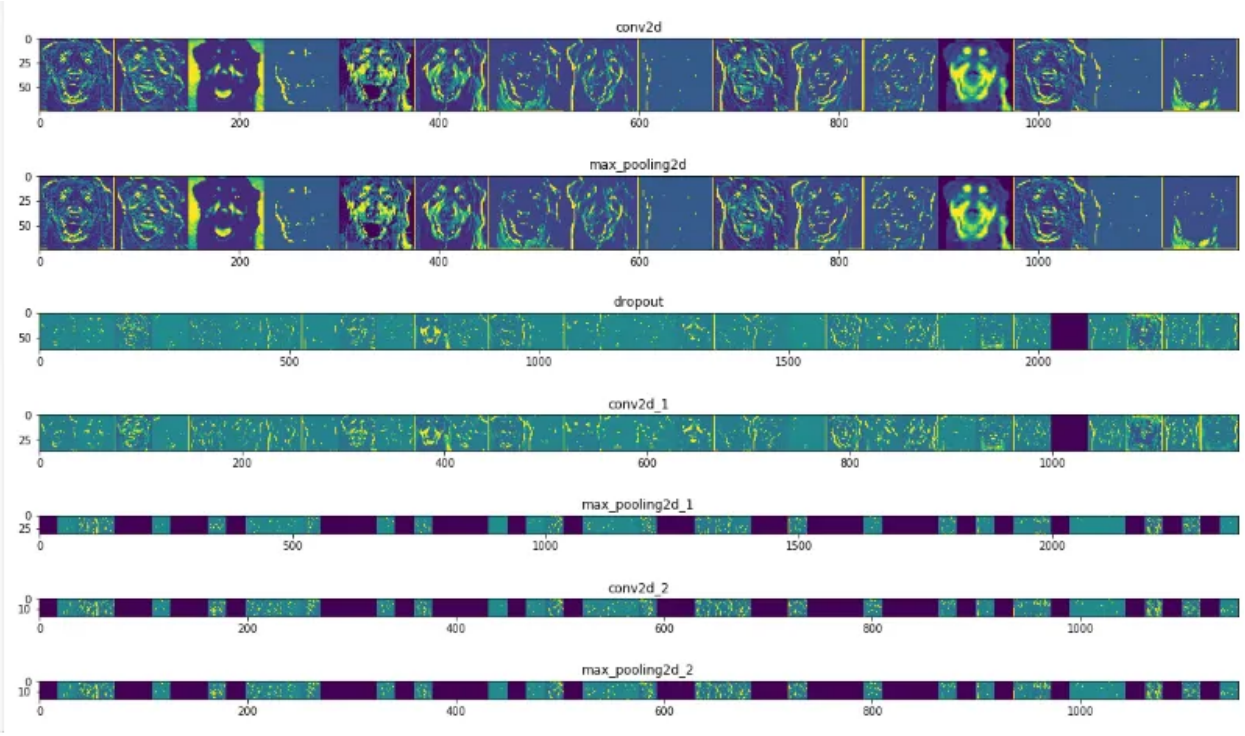
Visualizing filters in each layer to identify different features present in images provides a deeper insight into the model.



Visualizing Filters

Visualizing filters in each layer to identify different features present in images provides a deeper insight into the model.

This can be done with a process called **Gradient Ascent**.

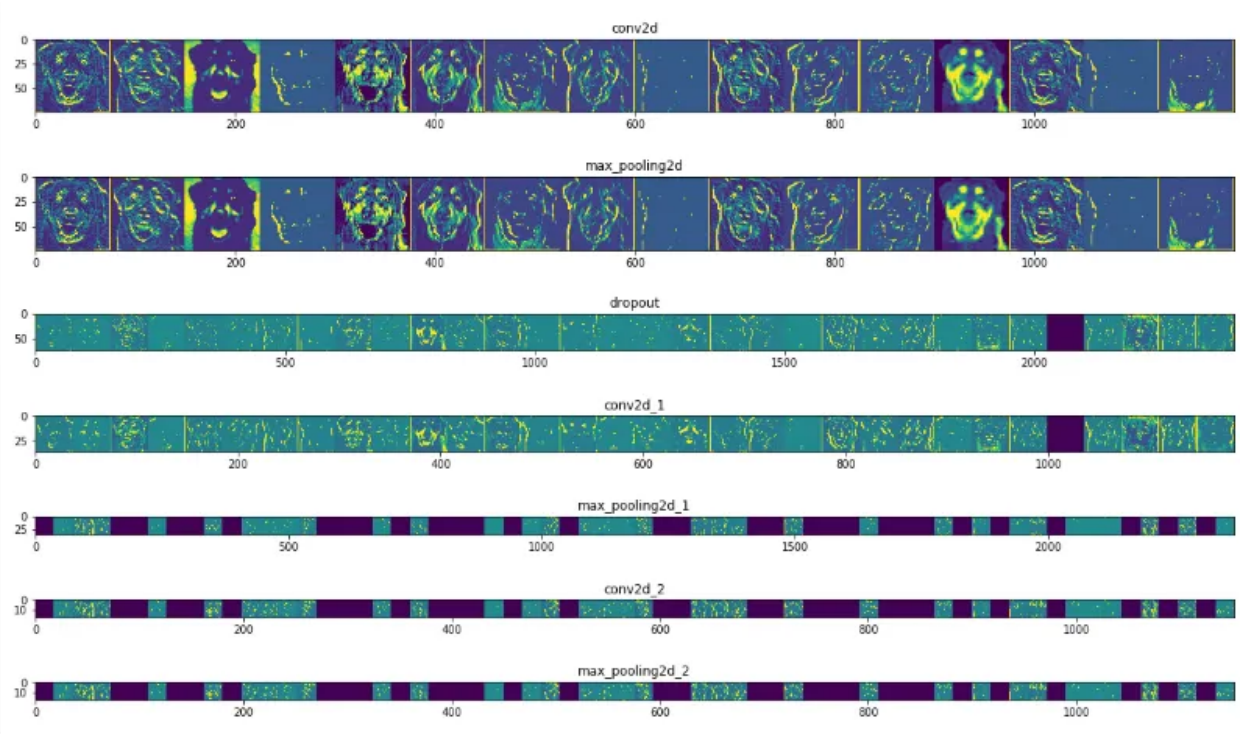


Visualizing Filters

Visualizing filters in each layer to identify different features present in images provides a deeper insight into the model.

This can be done with a process called **Gradient Ascent**.

Gradient Ascent starts with a zero or random image and continuously backpropagates until an image is generated that maximizes the score for the particular class (e.g.: dog).

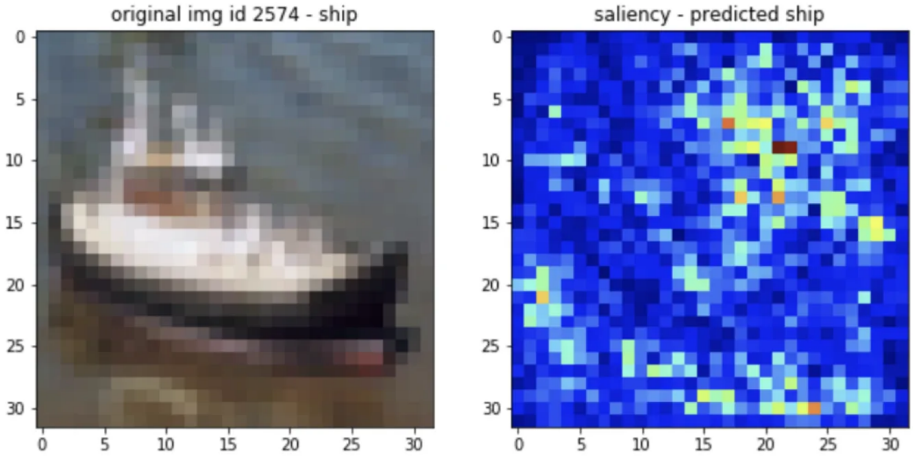


Saliency Maps

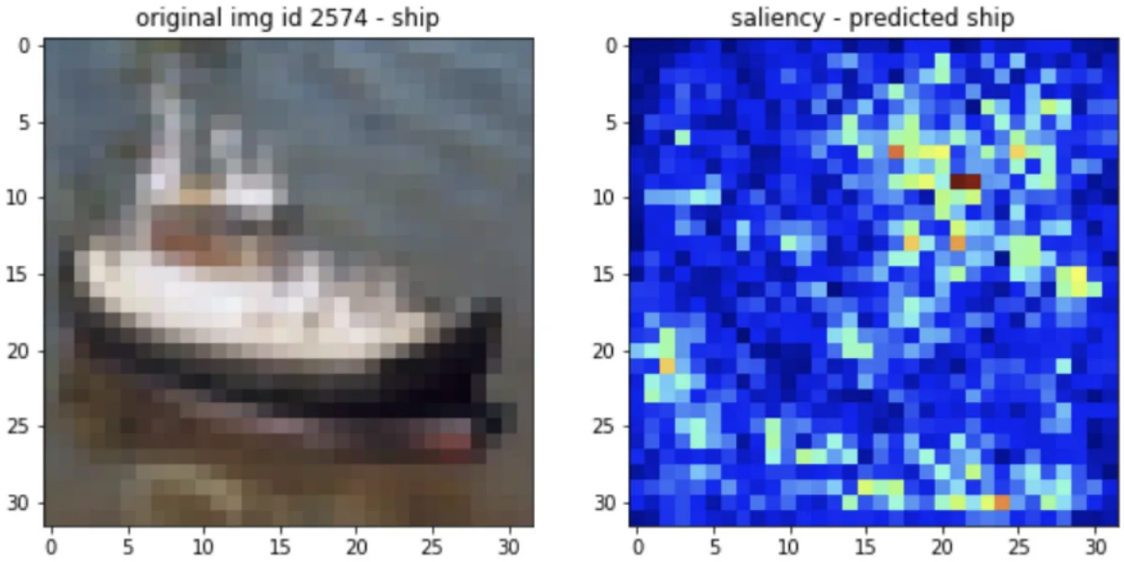
Saliency Maps (also known as Pixel Attribution methods) get a step further by providing an interpretable technique to investigate hidden layers in CNNs.

Saliency Maps highlight the pixels that were relevant for a certain image classification by a neural network.
In this way, they provide a tool to measure the spatial support of a particular class in each image.

The following image **illustrates** this:

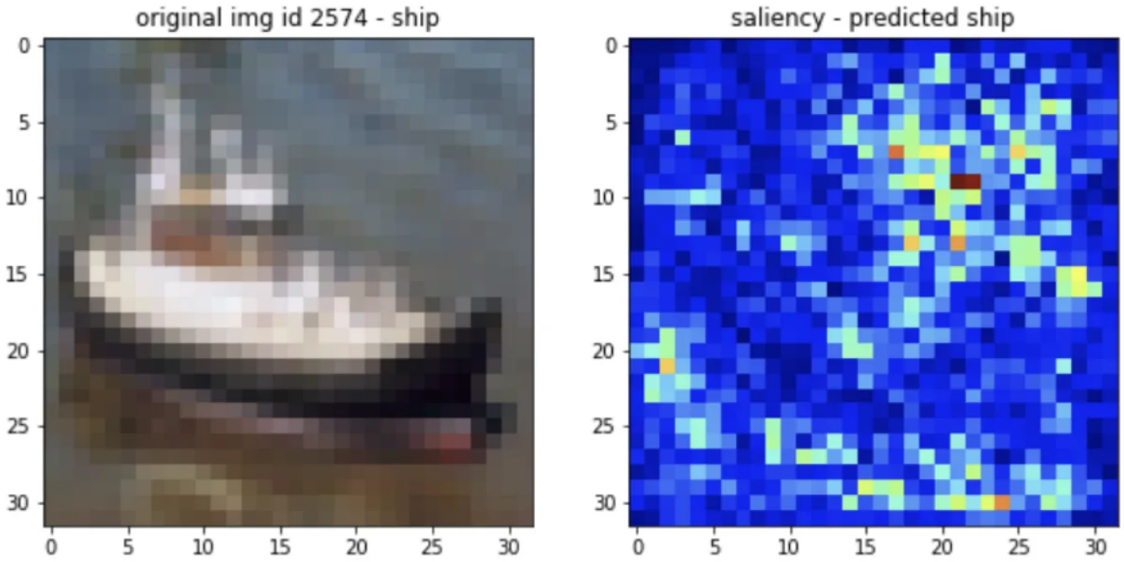


Saliency Maps



In this example we see that water plays a significant role to recognize a ship.

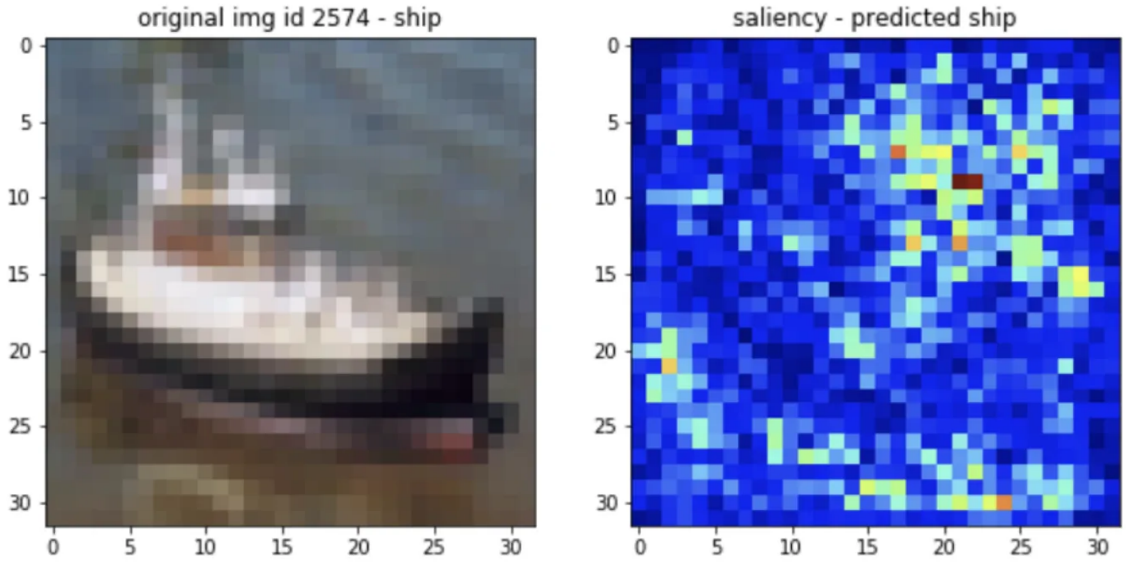
Saliency Maps



In this example we see that water plays a significant role to recognize a ship.

Thoughts: Maybe the model won't be successful in recognizing a ship in a construction hangar. Maybe the model is successful in discriminating between a ship and a toy ship on a table. Maybe the model won't be succcessful in discriminating between a ship and a toy ship in a bathtub.

Saliency Maps



In this example we see that water plays a significant role to recognize a ship.

Thoughts: Maybe the model won't be successful in recognizing a ship in a construction hangar. Maybe the model is successful in discriminating between a ship and a toy ship on a table. Maybe the model won't be succcessful in discriminating between a ship and a toy ship in a bathtub.

This observation provides **important clues** about the need to retrain the model with additional images of ships in different environmental conditions.

Approaches for Saliency Maps

There are **various approaches** for creating saliency maps.

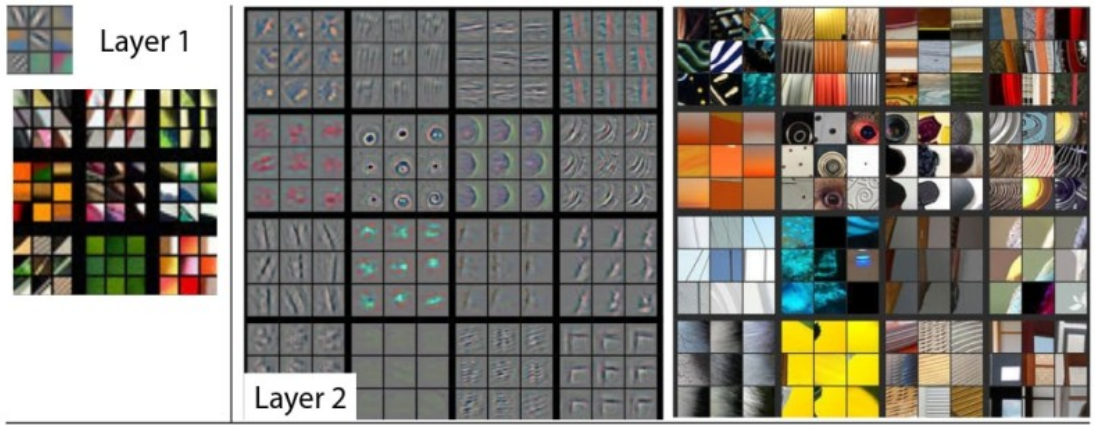
We're here discussing only the following three:

- Deconvolutional Network Approach
- Gradient-Based Approach
- Class Activation Mapping (CAM)

Deconvolutional Network Approach

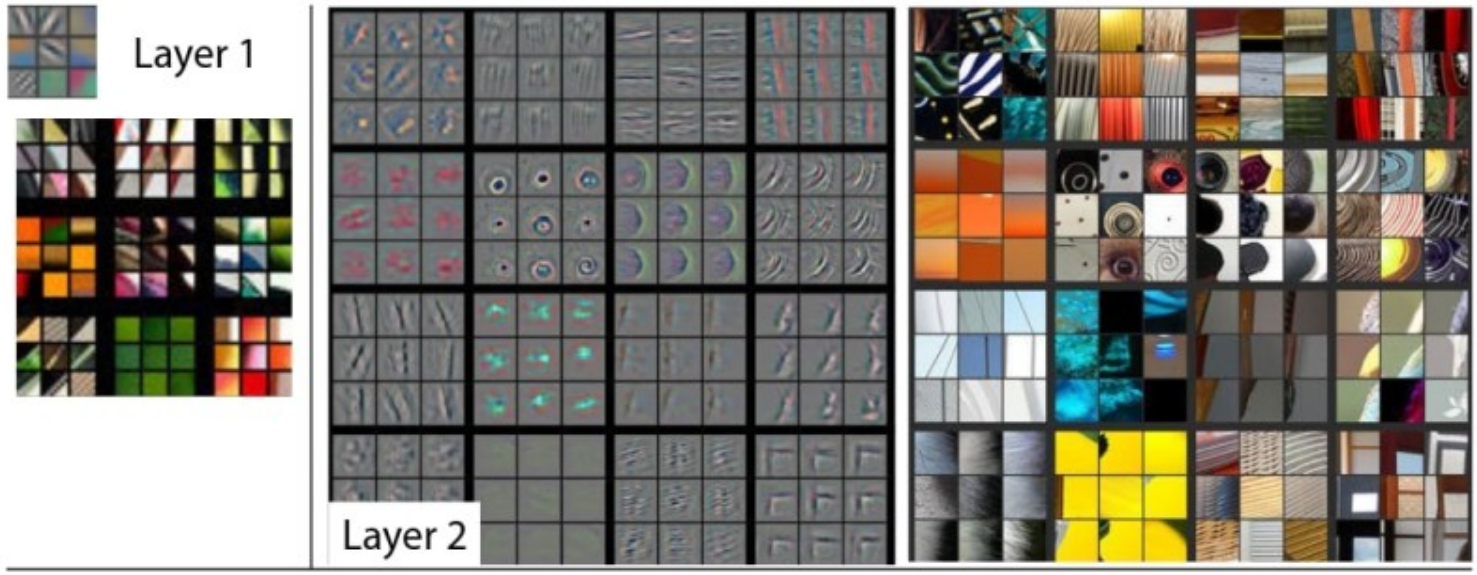
The Deconvolutional Network Approach (Zeiler & Fergus 2013) was one of the **very first methods** to understand the intermediate layer activities in CNN. At the time of publication, image classification networks had already started to perform well. But it was difficult to know why so.

Zeiler & Fergus (2013) perform visualization with a deconvolutional network, mapping the hidden layer of a neural network to the input. This helps in understanding which pattern led to the feature map activations.



Feature maps along with the input space using the `deconvnet` approach.
Credit: Zeiler & Fergus (2013)

Deconvolutional Network Approach

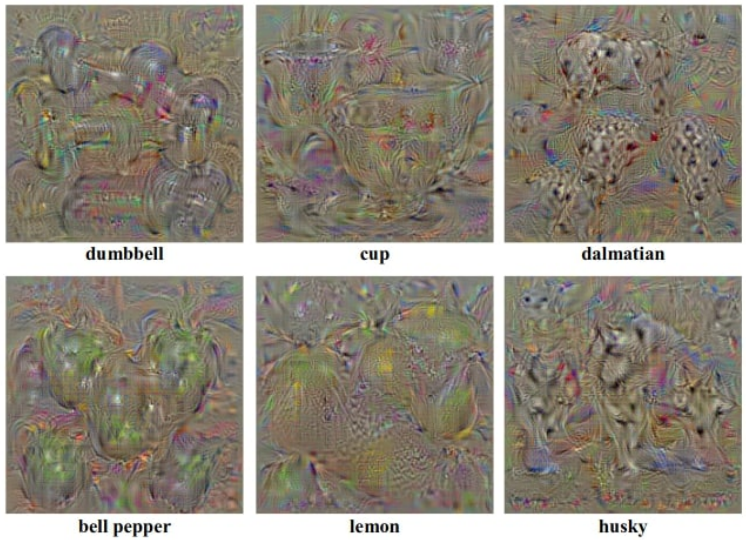


The patterns in the features maps show which part causes high activation. We can see that initially (layers 1 and 2), edges and circular patterns cause most of the activations. Later layers are activated by more image-specific sophisticated features like tires of cars, faces of dogs.

Gradient-Based Approach

The gradient-based approach for saliency map visualization was introduced by Simonyan et al. (2013) with two methods:

- The first method is a learned class model visualization technique. A target class is provided to the trained CNN. The CNN's visualization will consist of numerically generating the image.
- The second method consists of computing a class saliency map. This is image-specific class saliency visualization which shows which part of an image leads to the maximum class score.



Method 1 and 2, credit: Simonyan et al. (2013).

Gradient-Based Approach

How does this approach work?

The gradient of the loss function for the class we are interested in is computed w.r.t. the input pixels. This gives a map of the size of the input features with negative to positive values.

The recipe for this approach is then:

i) Perform a forward pass of the image of interest.

ii) Compute the gradient of class score of interest w.r.t. the input pixels:

$$E_{\text{grad}}(I_0) = \frac{\delta S_c}{\delta I}\bigg|_{I=I_0}$$

iii) Set all other classes to zero.

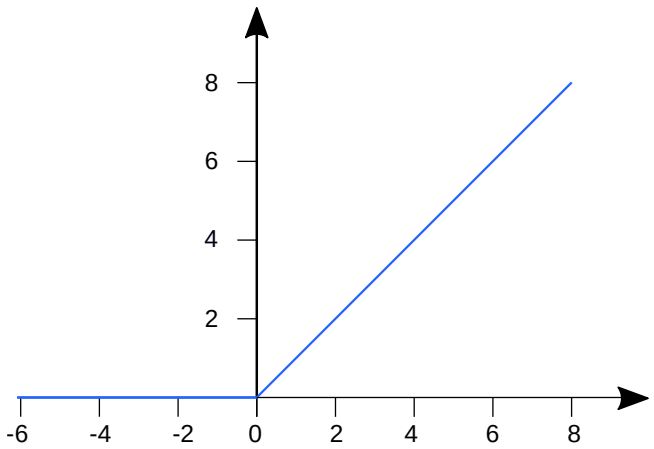
iv) Visualize the gradients.

Gradient-Based Approach

More formally:

The CNN classifies an image I with a class score $S_c(I)$. The score is a highly non-linear function of the image. The **idea** behind using the gradient is that we can approximate that score with a first-order Taylor expansion $S_c(I) \approx w^T I + b$ where $w = \frac{\delta S_c}{\delta I}\Big|_{I_0}$ is the derivative of the score.

There is some **ambiguity** how to perform a backward pass of the gradients, since non-linear units such as ReLU (Rectifying Linear Unit) remove the sign which is then unavailable during backpass.

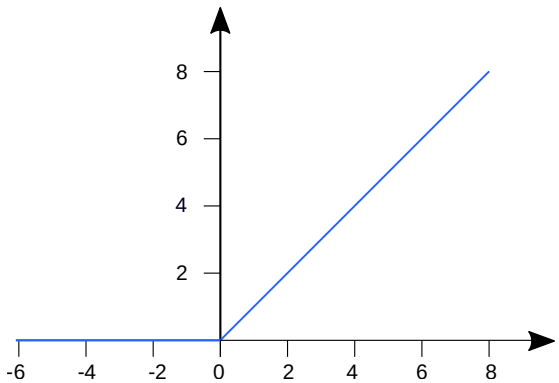


Gradient-Based Approach

More formally:

The CNN classifies an image I with a class score $S_c(I)$. The score is a highly non-linear function of the image. The **idea** behind using the gradient is that we can approximate that score with a first-order Taylor expansion $S_c(I) \approx w^T I + b$ where $w = \frac{\delta S_c}{\delta I}\Big|_{I_0}$ is the derivative of the score.

There is some **ambiguity** how to perform a backward pass of the gradients, since non-linear units such as ReLU (Rectifying Linear Unit) remove the sign which is then unavailable during backpass.

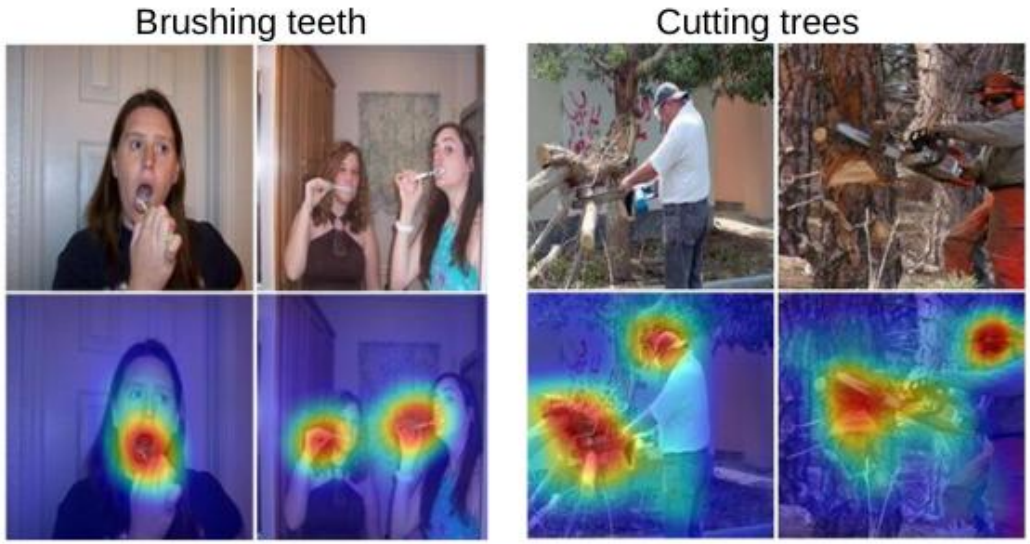


This ambiguity is resolved as follows:
$\frac{\delta f}{\delta X_n} = \frac{\delta f}{\delta X_{n+1}} \cdot \mathrm{I}(X_n > 0)$

Here, I is the element-wise indicator function, which is 0 where the activation at the lower layer was negative, and 1 where it is ≥ 0.

Class Activation Mapping (CAM)

Class Activation Mapping (Zhou et al. 2015) visualizes parts of an image that activate the convolutional neural network as heatmaps overlayed on the image.



Saliency map using class activation mapping. Credit: Zhou et al. (2015).

Class Activation Mapping (CAM)

Class Activation Mapping (Zhou et al. 2015) visualizes parts of an image that activate the convolutional neural network as heatmaps overlayed on the image.

The model is able to both recognize the activity and also localize specific image regions which leads to the prediction of the class.

The authors use the term Class Activation Maps to refer to weighted activation maps generated by a CNN. These weighted activations lead to the prediction of a specific label for the image.

Advantages and Disadvantages of Saliency Maps

- As a visual method, it is easy to understand for humans: We immediately recognize the image regions highlighted as *important*.
- Gradient-based methods are usually faster to compute than other methods.

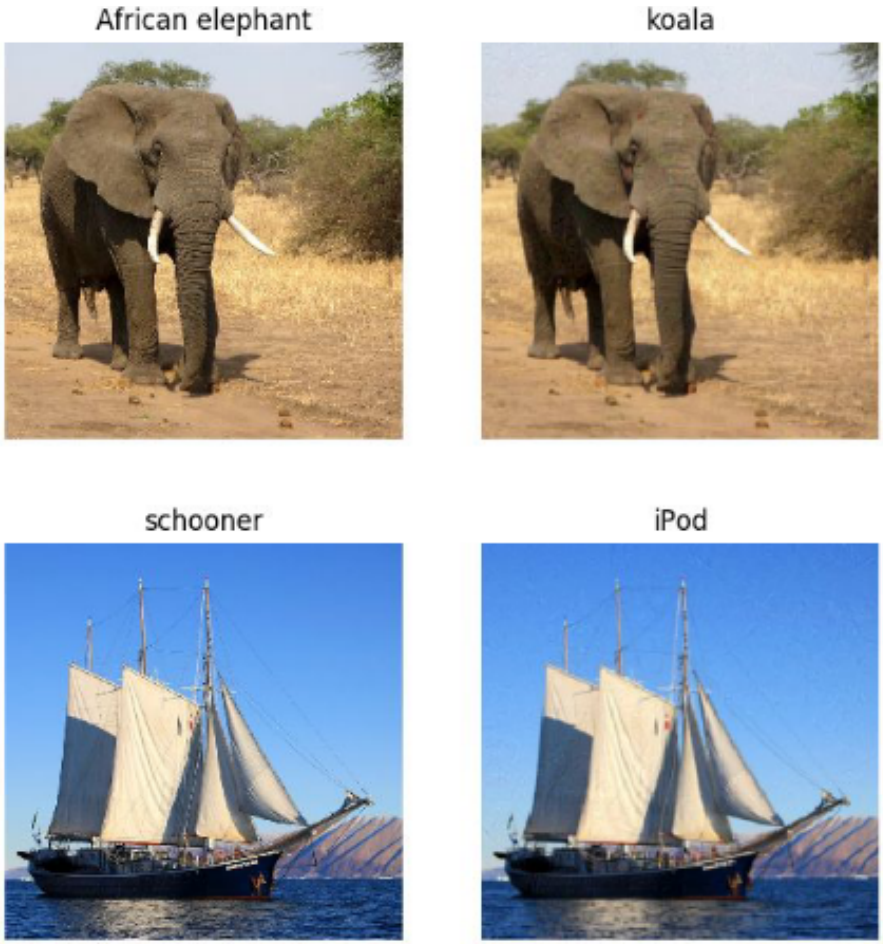
Advantages and Disadvantages of Saliency Maps

- As a visual method, it is easy to understand for humans: We immediately recognize the image regions highlighted as *important*.
- Gradient-based methods are usually faster to compute than other methods.

- As with most interpretation methods, it is difficult to know whether an explanation is correct, and a huge part of the evaluation is only qualitative.
- Saliency Map methods can be very fragile. Ghorbani et al. (2019) showed that introducing small (adversarial) perturbations to an image, which still lead to the same prediction, can lead to very different pixels being highlighted as explanations. Also, small pertubations in an image which would lead to humans recognizing the image as still being from the same class can lead to different predictions.

Fooling a CNN

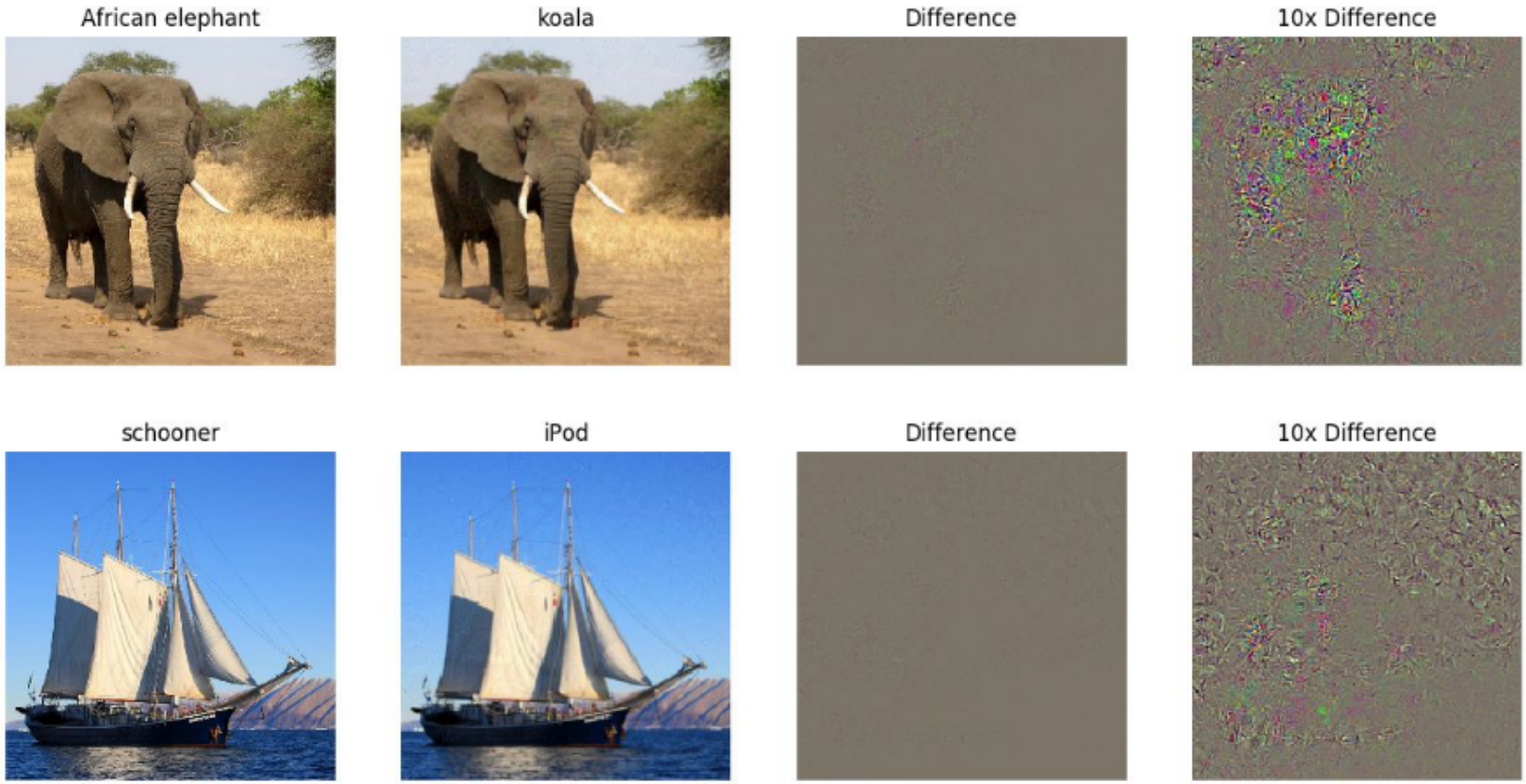
a CNN's classification:



?

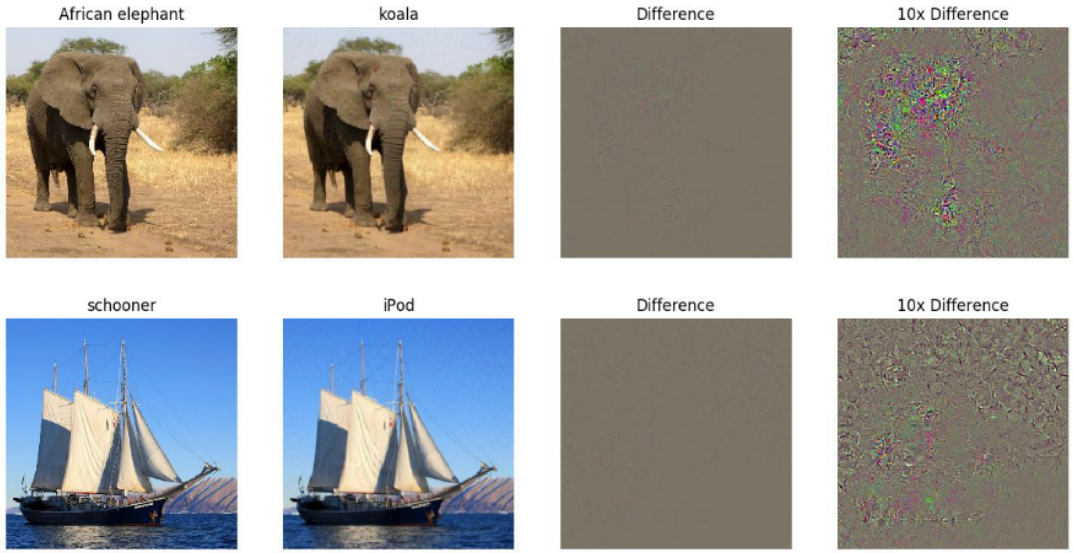
Fooling a CNN

Suppose we feed in an image to a CNN. Instead of maximizing the likelihood of the correct class (e.g. elephant) we maximize an incorrect class (e.g. koala) while making minimal changes to the image.



Fooling a CNN

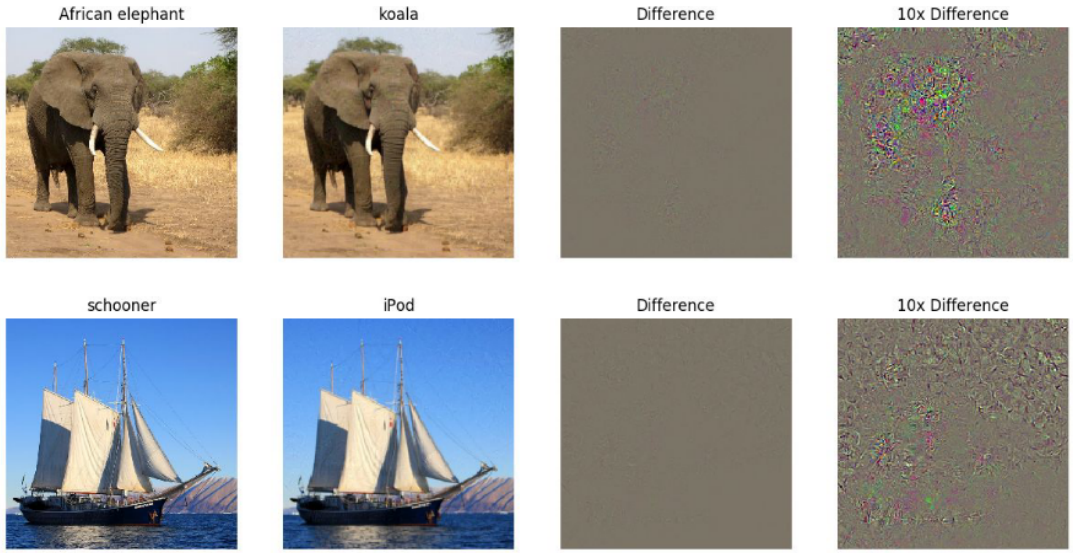
Suppose we feed in an image to a CNN. Instead of maximizing the likelihood of the correct class (e.g. elephant) we maximize an incorrect class (e.g. koala) while making minimal changes to the image.



Intriguing properties of neural networks, Szegedy et al. (2013)
Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images, Nguyen, Yosinski, Clune (2014)

Fooling a CNN

Suppose we feed in an image to a CNN. Instead of maximizing the likelihood of the correct class (e.g. elephant) we maximize an incorrect class (e.g. koala) while making minimal changes to the image.



Intriguing properties of neural networks, Szegedy et al. (2013)
Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images, Nguyen, Yosinski, Clune (2014)

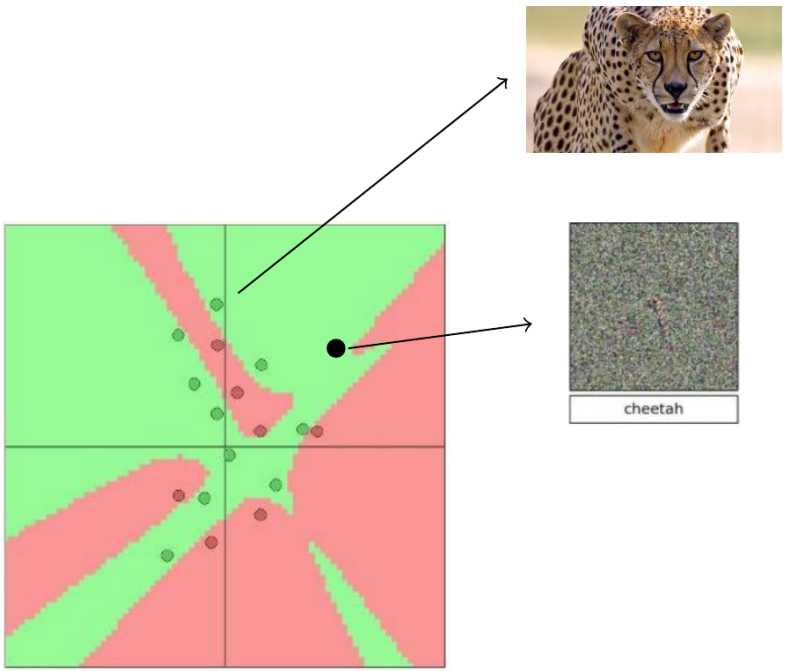
Notice that the changes are so minimal that the two images are indistinguishable to humans

Fooling a CNN

Why does this work?

Using the training images we fit some **decision boundaries** in the high-dimensional parameter space.
While doing so based on the **finite set of training images**, also decisions about many unseen points in parameter space are made.

As a result: Of the many points in this high-dimensional parameter space belonging to a certain class, only a few represent true images.



credit:
Neural Networks are Easily Fooled: High Confidence Predictions for Unrecognizable Images, Nguyen, Yosinski, Clune (2014)

Google's *Deep Dream*

A fun way to use CNN:

Exaggerating feature attributes or textures using information that the bvlc_googlenet model learned during training.

The Mona Lisa with DeepDream effect using VGG16 network trained on ImageNet.

Google's *Deep Dream*

A fun way to use CNN:

1. Forward propagate a baseline image through your model. Compute the activations at a chosen layer.
2. To maximize the neuron activations in the layer we care about, set the gradient of chosen layer equal to its activation.
3. During the backward pass, compute the gradient on the loss w.r.t. the input image, as in the case of gradient ascent.
4. Execute the update rule to update your input image. We change the image so that these neurons fire even more.

you can try it out at: `https://deepai.org/`

Summary and Outlook

CNNs enable us to deal with images more naturally.

Summary and Outlook

CNNs enable us to deal with images more naturally.

Tomorrow we look at another neural network architecture: **Autoencoders**.