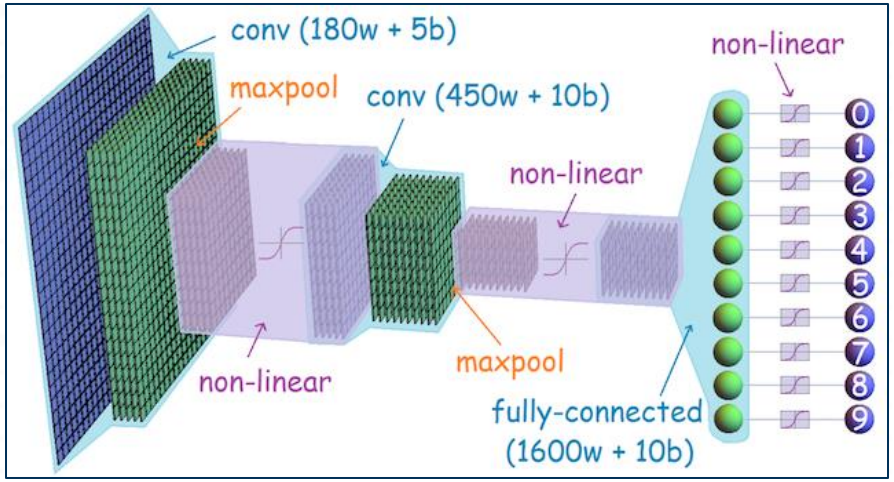
ECE 4252/8803: Fundamentals of Machine Learning (FunML)
Fall 2024

Lecture 16: Convolutional Neural Networks Training



OLIVES @GeorgiaTech

Georgia Tech

Overview

In this Lecture..

Locally Connected Layer

Introduction to Convolutional Neural Networks

Layers used to build Convolutional Neural Networks

Examples of Deep CNNs Architectures

Training CNNs

- Gradient Descent
- Momentum
- Adagrad
- RMSProp
- ADAM
- BFGS
- L-BFGS

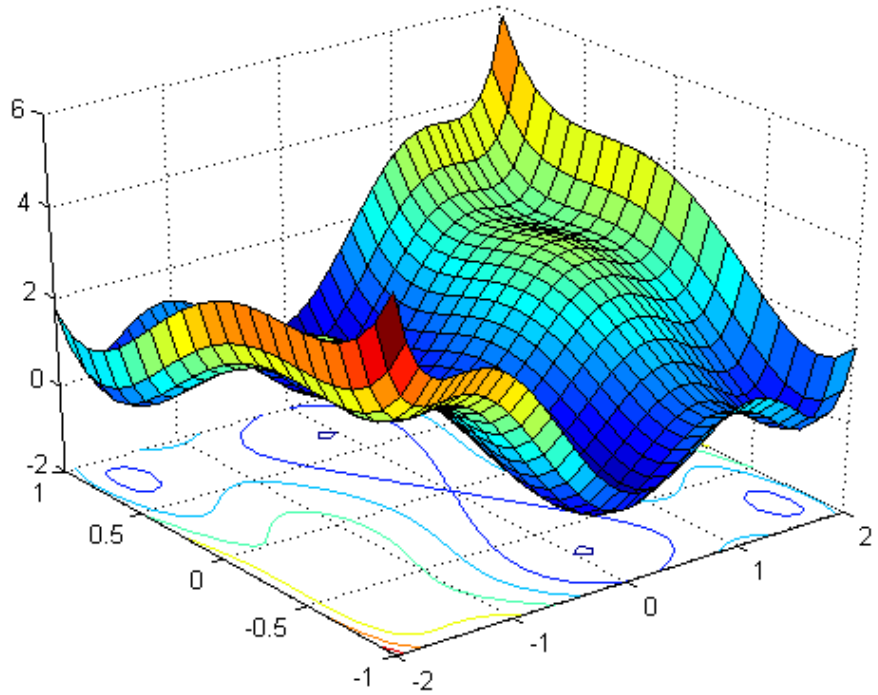
Visualization of Convolutional Neural Networks

Finding Optimal Parameters

Gradient Descent

Similar to ANNs, *weights* and *biases* in CNNs can be learned via ***backpropagation*** and ***gradient update*** w.r.t. a loss function

In general, the loss landscape of CNNs are non-convex with **local minima**



Other issues in optimization:

- Noisy gradient estimation
- many saddle points
- Ill-conditioned loss surface

Finding Optimal Parameters

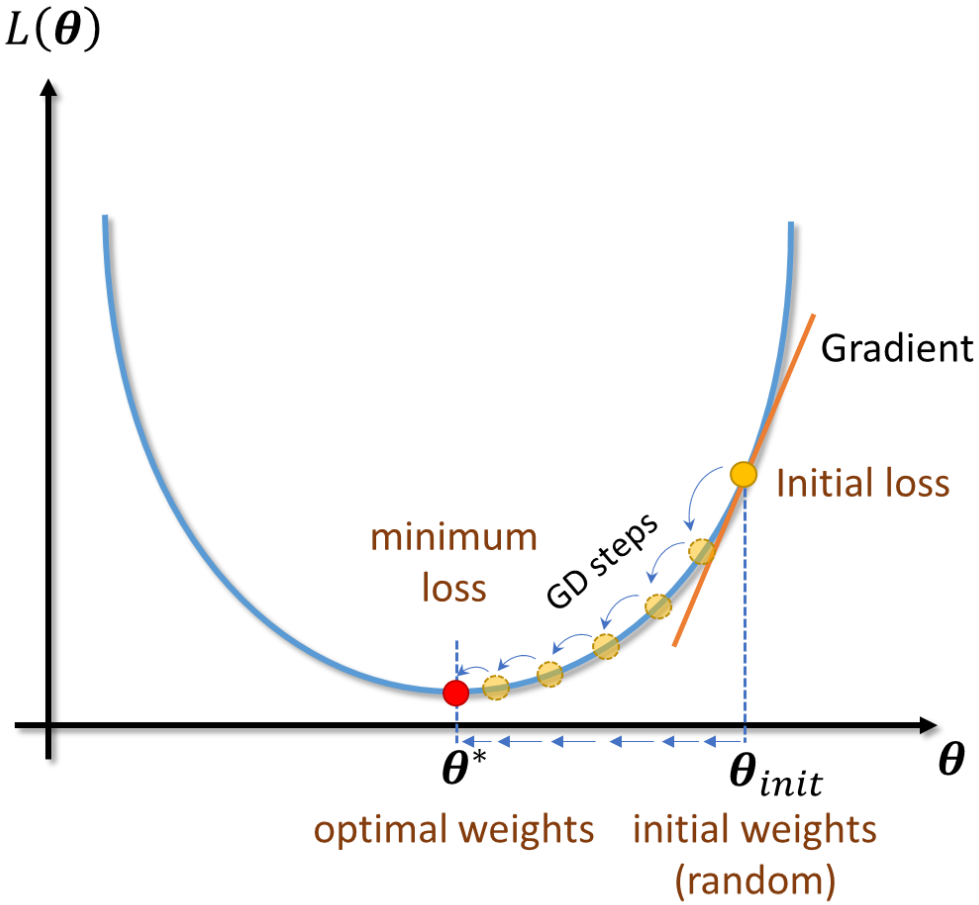
Gradient Descent

- The basic idea is to compute the gradient (derivative) of the loss function $L(\boldsymbol{\theta})$ in a sequence of epochs (intake of the dataset) and update the parameters in proportion to the gradient according to the update rule:

$$\boldsymbol{W}(t+1) = \boldsymbol{W}(t) - \alpha \frac{\partial\, L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}$$

$$\boldsymbol{b}(\mathrm{t}+1) = \boldsymbol{b}(\mathrm{t}) - \alpha \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{b}}$$

where $t+1$ and t indicate the new and current epochs, respectively, and α is a predetermined learning rate usually smaller than 0.5



Finding Optimal Parameters

Gradient Descent

Calculation of $\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}$ and $\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{b}}$ needs to compute the gradient over the entire training set at every step, which makes it very slow when the training set is large.

There are other alternatives for implementing the gradient descent algorithm without computing the derivative over the entire training set:

- ***Batch*** *Gradient Descent:*

  Calculate gradients of cost function over all instances at each step

- ***Stochastic*** *Gradient Descent*

  Calculate gradients of cost function over a single random instance at each step

- ***Mini-batch*** *Gradient Descent*

  Calculate gradients of cost function over every small batch of instances

Training is performed in iterations in which the training set is processed to calculate gradual adjustments to coefficients in small steps until convergence.

Finding Optimal Parameters

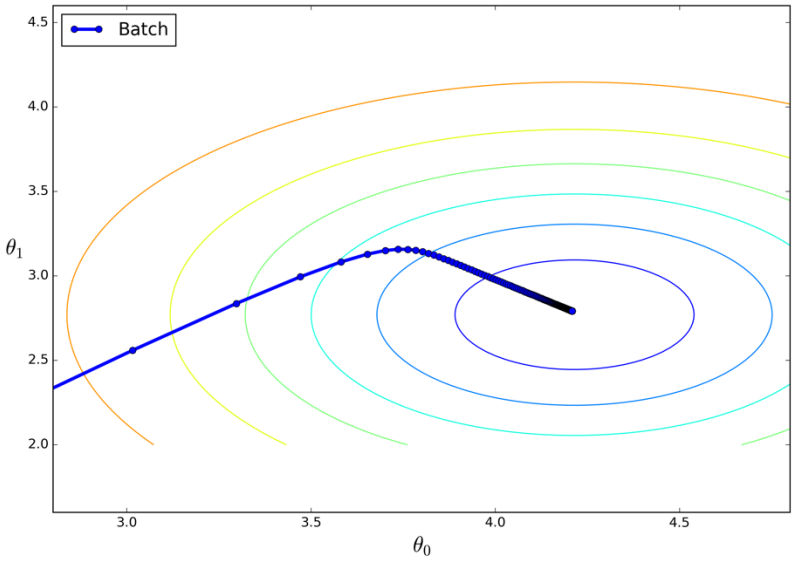
Batch Gradient Descent

```
Randomly initialize weights and the bias
Repeat until convergence
{
    Calculate gradient ∂L(θ)/∂W and ∂L(θ)/∂b over the entire dataset:
    loss.backward()
    W_gradient = W_tensor.grad
    b_gradient = b_tensor.grad
    Update:
    W_tensor.data = W_tensor.data - alpha*W_gradient
    b_tensor.data = b_tensor.data - alpha*b_gradient
}
```

Can be substituted by

```
optimizer.step()
```

Convergence path of Batch GD



- Convergence takes place when $\|\nabla_\theta L(\boldsymbol{\theta})\| < \epsilon$, where ϵ (tolerance) takes a tiny value. It indicates that coefficients will almost not update with further steps
- Smoother convergence

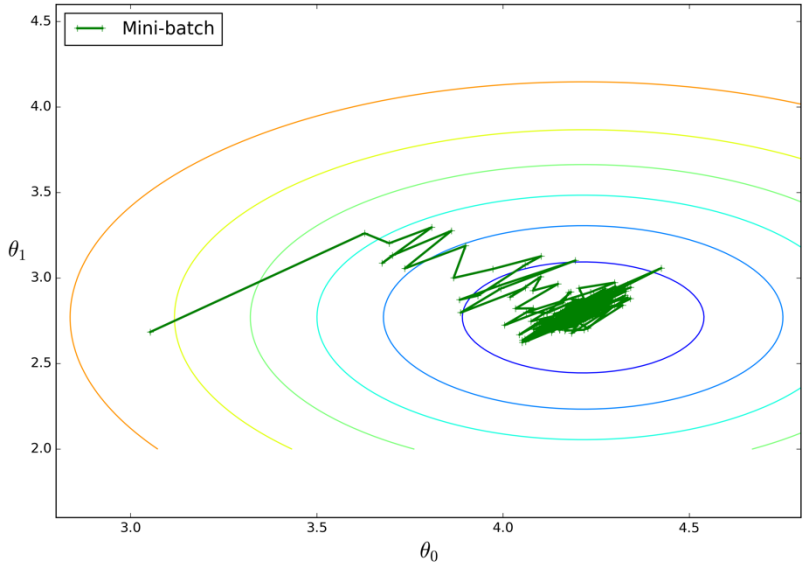
Finding Optimal Parameters

Mini-batch Gradient Descent

```
Randomly Initialize weights and the bias
Repeat until convergence
{For N/b iterations:
    {Sample a subset X_S of size b instances at random
    Calculate ∂L(θ)/∂W and ∂L(θ)/∂b over all samples in X_S:
    loss.backward()
    W_gradient = W_tensor.grad
    b_gradient = b_tensor.grad
    Update:
    W_tensor.data = W_tensor.data - alpha*W_gradient
    b_tensor.data = b_tensor.data - alpha*b_gradient
 }
}
```

Can be substituted by **`optimizer.step()`**

Convergence path of mini-batch GD



- Trade-off between computation complexity and finding optimal coefficients
- Less erratic progress in coefficient space
- May be harder to escape local minima
- Allows for a performance boost from parallel hardware

Finding Optimal Parameters

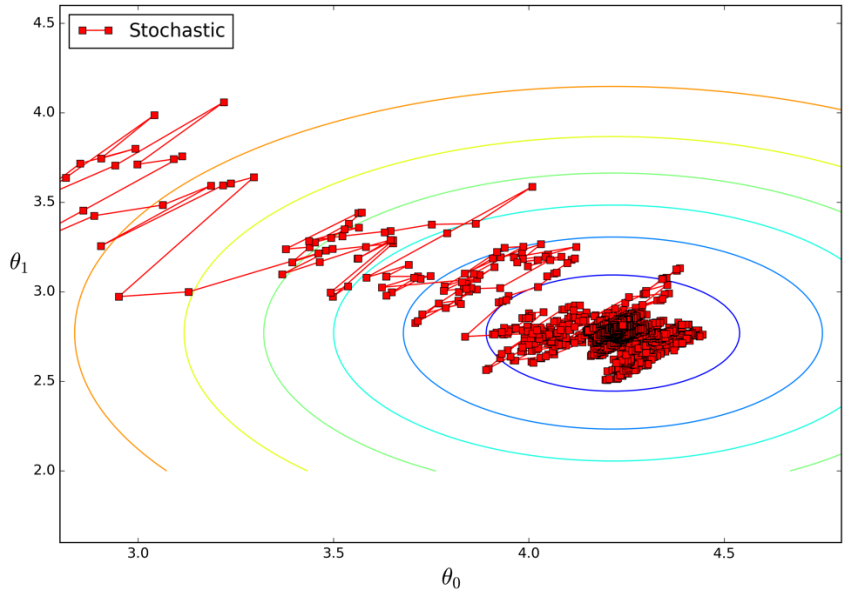
Stochastic Gradient Descent

```
Initialize weights and the bias
Repeat until convergence
{For N iterations:
    {Randomly pick i-th sample x_i
    Calculate ∂L(θ)/∂W and ∂L(θ)/∂b
    loss.backward()
    W_gradient = W_tensor.grad
    b_gradient = b_tensor.grad
    Update
    W_tensor.data = W_tensor.data - alpha*W_gradient
    b_tensor.data = b_tensor.data - alpha*b_gradient
    }
}
```

Can be substituted by

```
optimizer.step()
```

Convergence path of Stochastic GD



- Much faster because gradient is calculated based on a single sample
- Convergence path is much more stochastic
- Final coefficient values are good, but may not be optimal
- When cost function is non-convex, bouncing might help algorithm escape local minima

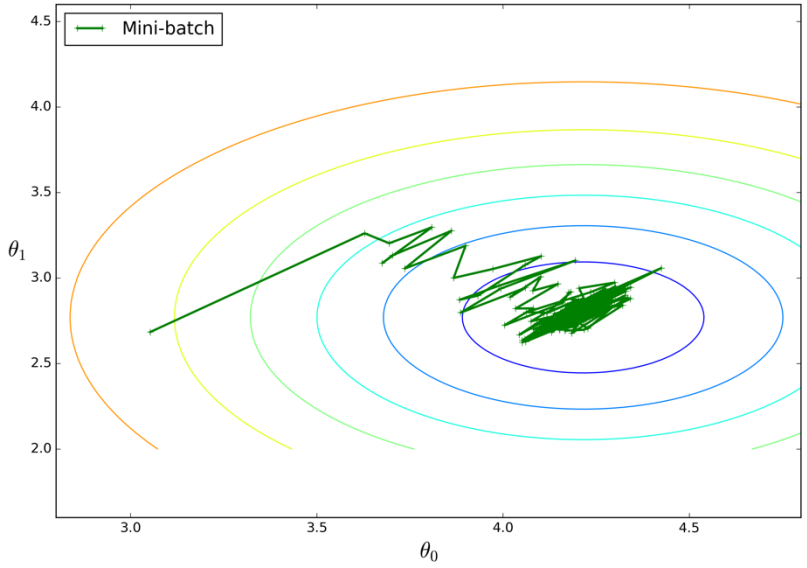
Finding Optimal Parameters

Stochastic Gradient Descent on Mini-batch Data

- In practice, we use a **subset of the data (mini-batch) at each** Gradient Descent **iteration** to calculate the loss and **estimate gradient**

- Our gradients in this example are calculated using mini-batches, thus the **noisy steps** in gradient descent

Convergence path of mini-batch GD



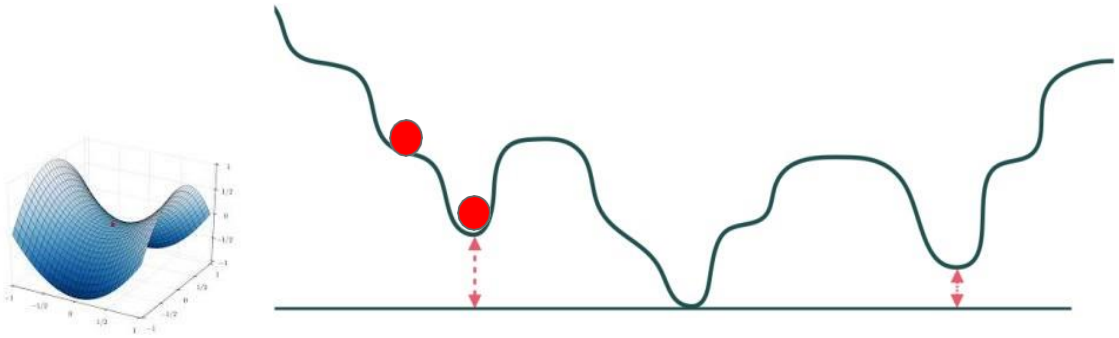
Finding Optimal Parameters

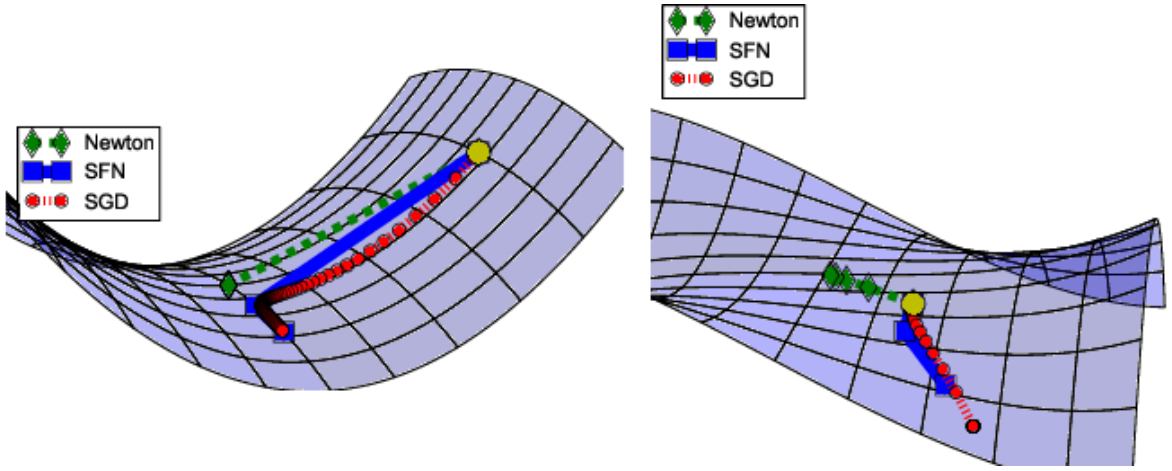
Loss Surface

- Loss surface geometries are difficult for optimization with **local minima** and **saddle points**

- At saddle points:
  - Gradients are **zero**
  - Orthogonal directions **disagree**
  - Gradient descent gets stuck

- Saddle points are common in high dimension





(a) 5x2-y2 and (b) x3-3xy2

Dauphin et al, “Identifying and attacking the saddle point problem in high-dimensional non-convex optimization”, NeurIPS 2014

Finding Optimal Parameters

Momentum

Intuition: Imagine a ball rolling down loss surface, and use momentum to pass flat surfaces

$$\boldsymbol{v}(t+1) = \beta \boldsymbol{v}(t) + \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)$$

Update Velocity ($\boldsymbol{v}(t)$ starts as 0)

$$\boldsymbol{W}(t+1) = \boldsymbol{W}(t) - \alpha \boldsymbol{v}(t+1)$$

Update weights

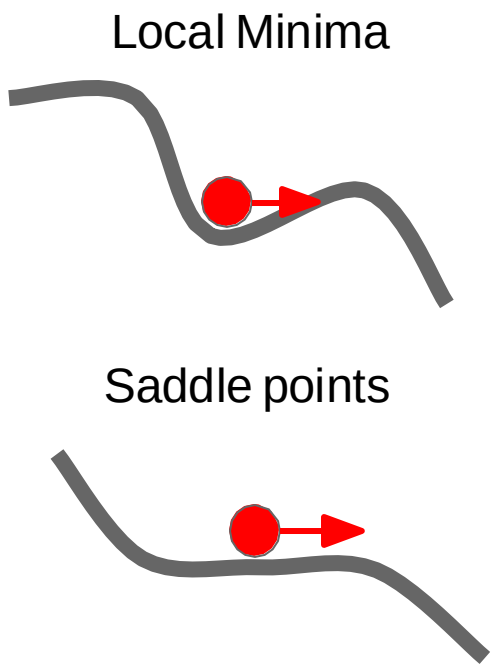
where

$\boldsymbol{v}(t)$: velocity at time t

β: coefficient of momentum, usually takes value < 1, e.g., 0.99

$\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)$: gradient at time t

Note: this generalizes SGD when $\beta = 0$



Sutskever et al, "On the importance of initialization and momentum in deep learning", ICML 2013

Finding Optimal Parameters

Momentum

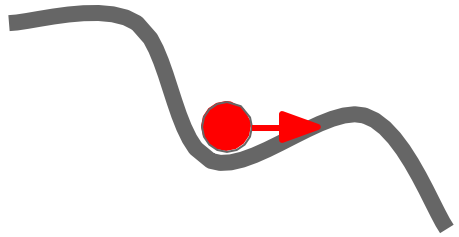
Velocity is an **exponential running average** of gradients

$$\boldsymbol{v}(t+1) = \beta \boldsymbol{v}(t) + \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)$$

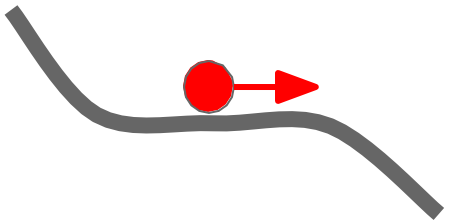
$$\boldsymbol{v}(t+1) = \beta\left(\beta \boldsymbol{v}(t-1) + \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t-1)\right) + \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)$$

$$= \beta^2 \boldsymbol{v}(t-1) + \beta \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t-1) + \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)$$

Local Minima



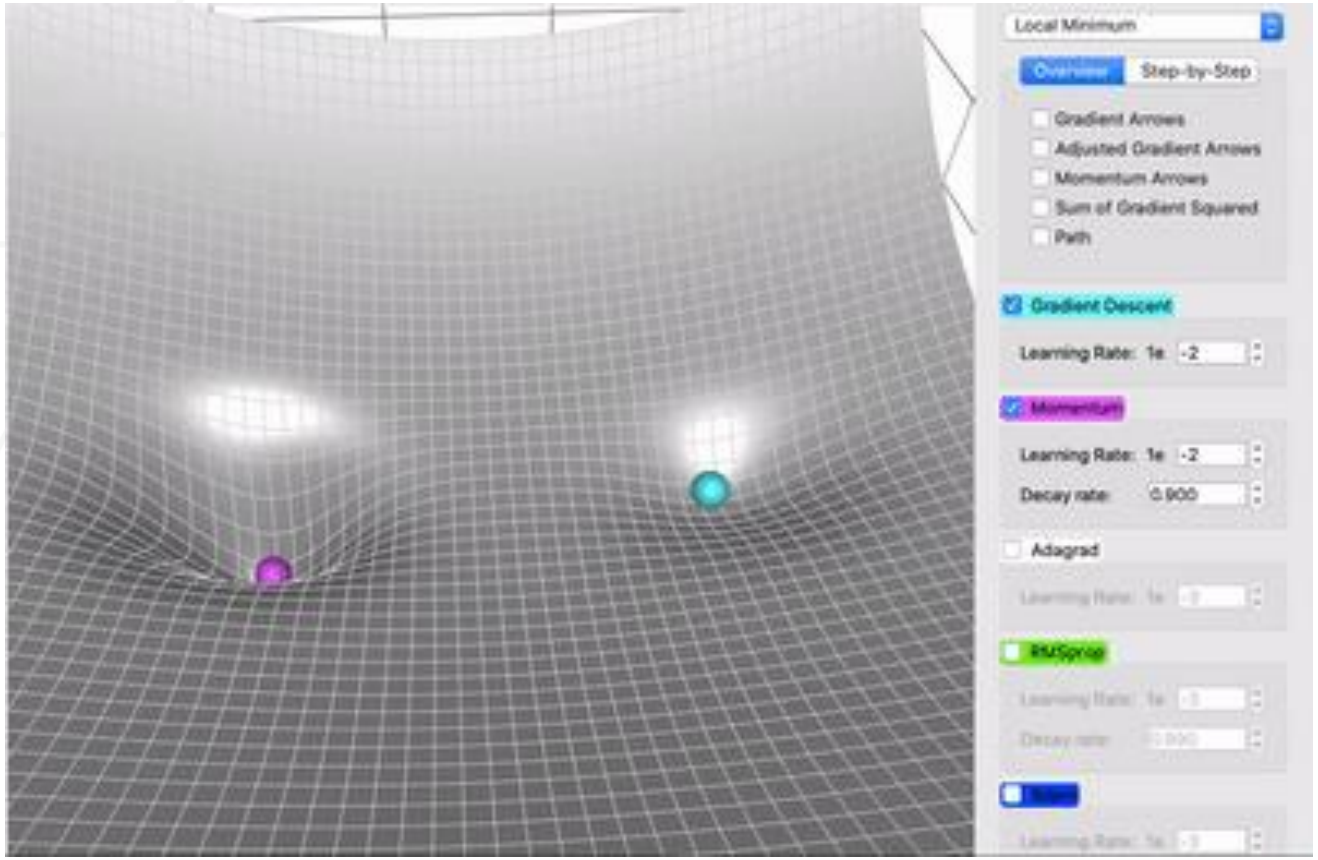
Saddle points

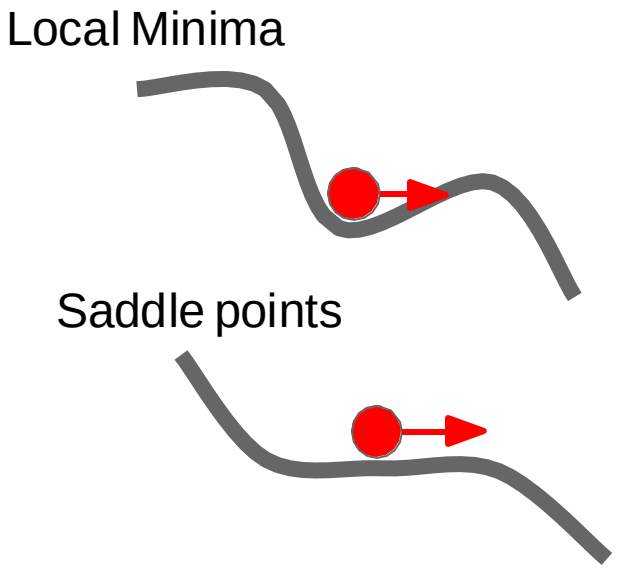


Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

Finding Optimal Parameters

Momentum





Momentum (magenta) vs. Gradient Descent (cyan) on a surface with a global minimum (the left well) and local minimum (the right well)

Sutskever et al, “On the importance of initialization and momentum in deep learning”, ICML 2013

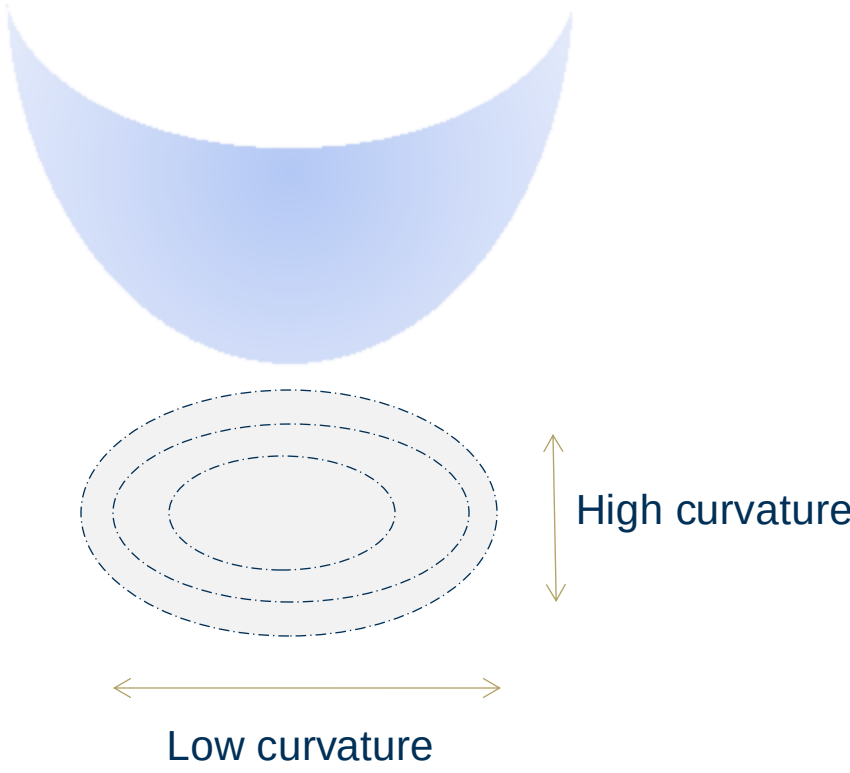
Finding Optimal Parameters

Loss Curvature and Condition Number

- The curvature of the loss characterizes the loss landscape.
- Recall Hessian matrix:

$$\boldsymbol{H}(\boldsymbol{\theta}) = \nabla^2_{\theta} L(\boldsymbol{\theta}) = \begin{pmatrix} \nabla^2_{\theta_1^2} L(\boldsymbol{\theta}) & \dots & \nabla^2_{\theta_1 \theta_P} L(\boldsymbol{\theta}) \\ \vdots & \vdots & \vdots \\ \nabla^2_{\theta_P \theta_1} L(\boldsymbol{\theta}) & \dots & \nabla^2_{\theta_P^2} L(\boldsymbol{\theta}) \end{pmatrix}$$

- The eigenvalues of $\boldsymbol{H}(\boldsymbol{\theta})$ are the principal curvatures of the loss, and the eigenvectors are the principal directions of curvature

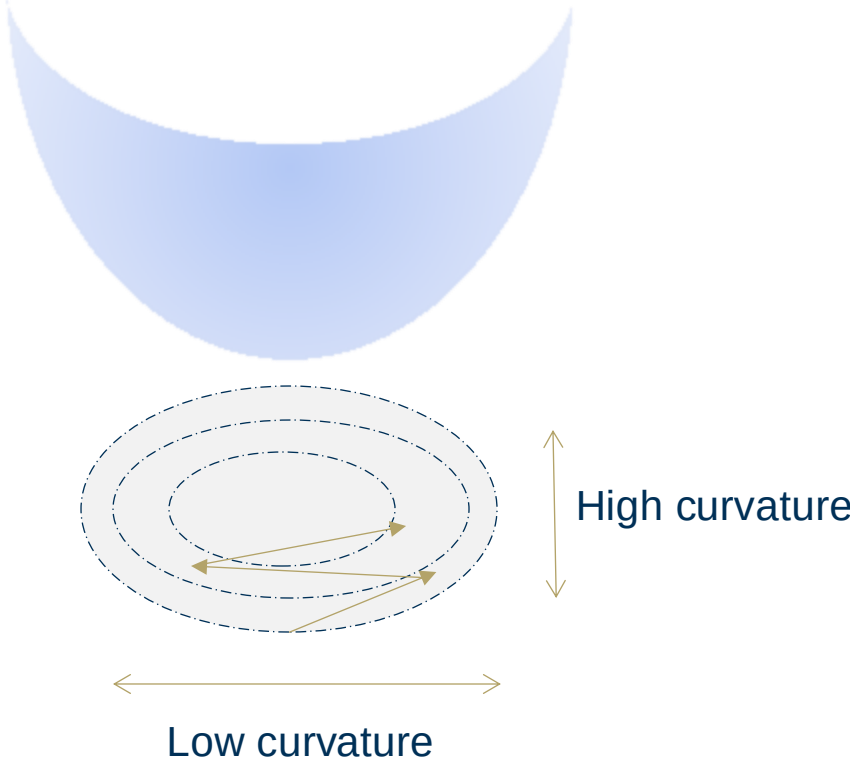


Finding Optimal Parameters

Loss Curvature and Condition Number

Condition number:

- The ratio of the largest and smallest eigenvalue of Hessian
- Characterizes how different the curvature is along different dimensions
- High condition number indicates that loss changes quickly in one direction and slowly in another



Finding Optimal Parameters

Adaptive Learning

- Intuition: applying adaptive learning rate for each weight dimension overcomes the issues of high condition number

- Optimization algorithms:
  - Adagrad
  - RMSProp
  - Adam

Finding Optimal Parameters

AdaGrad

AdaGrad (**ada**ptive **grad**ient) uses accumulated gradients to decay learning rate

$$\boldsymbol{G}(t+1) = \boldsymbol{G}(t) + \left(\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)\right)^{\mathbf{2}}$$

$$\boldsymbol{W}(t+1) = \boldsymbol{W}(t) - \frac{\alpha}{\sqrt{\boldsymbol{G}(t+1) + \boldsymbol{\epsilon}}} \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)$$

where

- $\boldsymbol{G}(t+1)$: historical sum of squares in each dimension
- α: learning rate

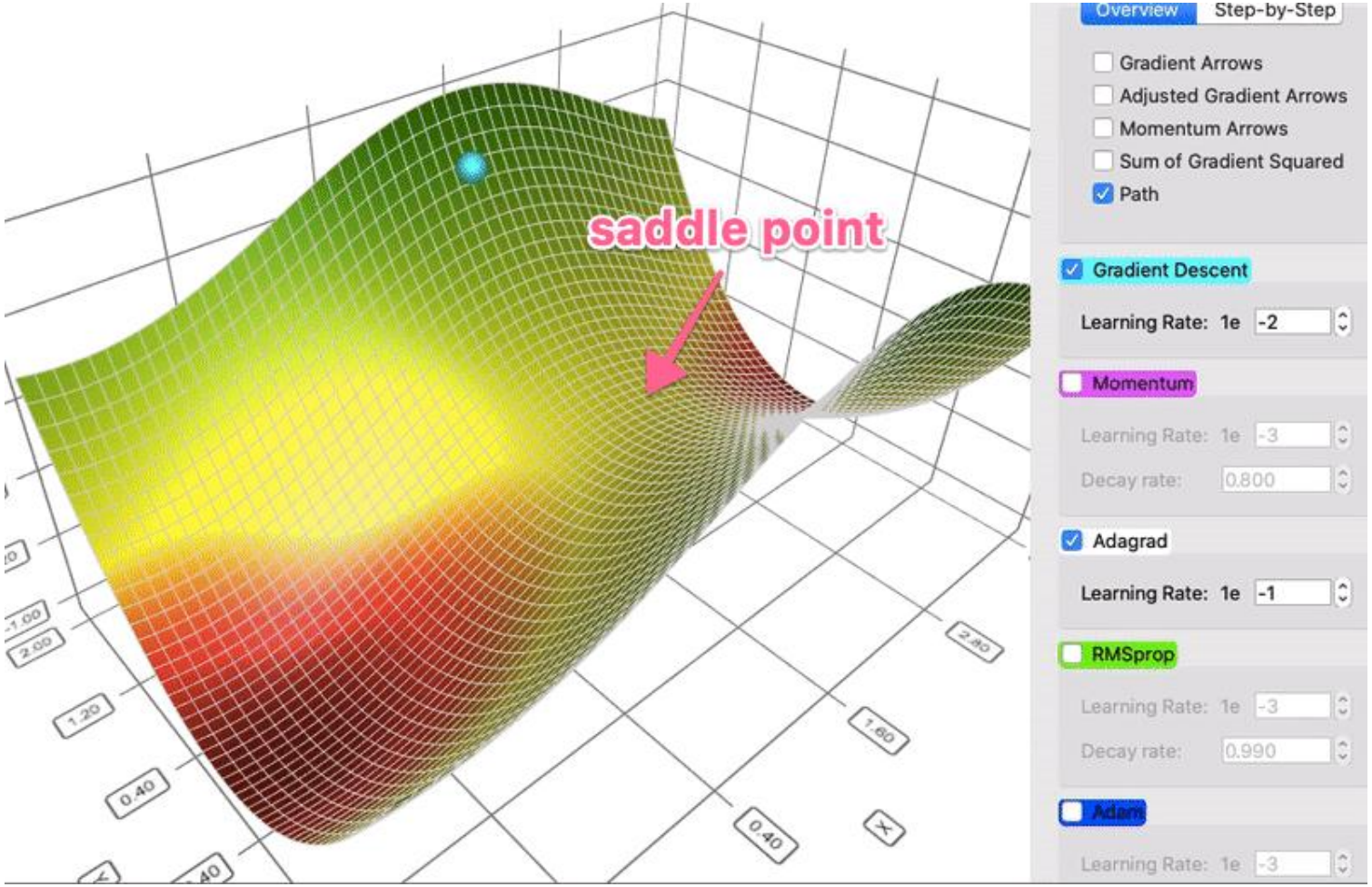
- element-wise scaling of the gradient
- learning rate will eventually decay to zero as gradients are accumulated

Directions with high curvature have larger gradients, and learning rate reduces for damped update

Duchi et al, "Adaptive subgradient methods for online learning and stochastic optimization", JMLR 2011

Finding Optimal Parameters

AdaGrad



AdaGrad (white) vs. gradient descent (cyan) on a terrain with a saddle point. The learning rate of AdaGrad is set to be higher than that of gradient descent, but the point that AdaGrad's path is straighter stays largely true regardless of learning rate.

Finding Optimal Parameters

RMSProp

- RMSProp (**R**oot **M**ean **S**quare **Prop**agation) uses a weighted average of squared gradients

$$\boldsymbol{G}(t+1) = \beta \boldsymbol{G}(t) + (1-\beta)\left(\frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)\right)^{\mathbf{2}}$$

$$\boldsymbol{W}(t+1) = \boldsymbol{W}(t) - \frac{\alpha}{\sqrt{\boldsymbol{G}(t+1) + \boldsymbol{\epsilon}}} \frac{\partial L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)$$

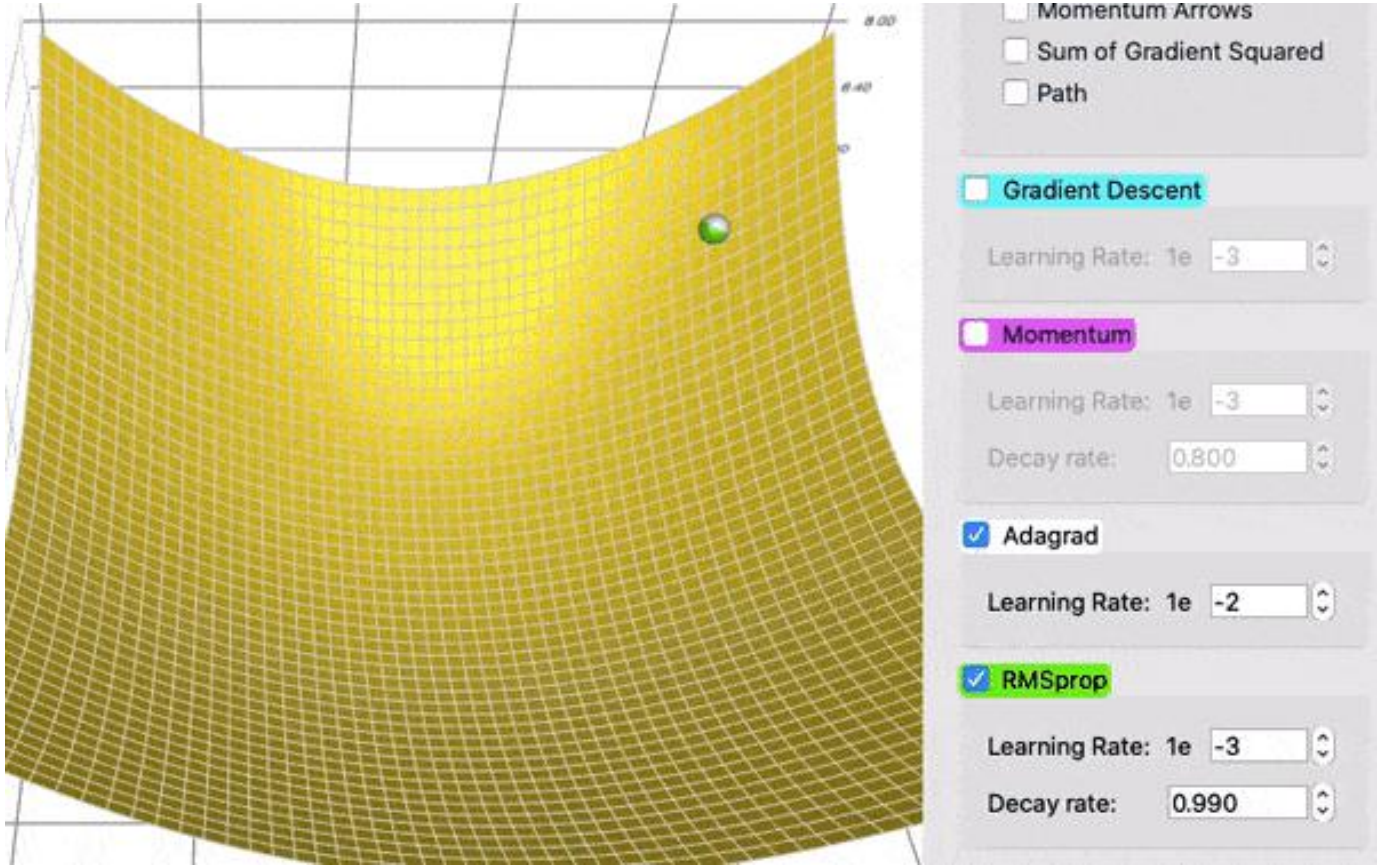
- Adagrad is very slow because of the sum of gradient squared only grows and does not shrink. RMSProp solves this issue by adding a decay factor, β. The decay rate is saying only recent gradient matters, and the ones from long ago are basically forgotten.

Finding Optimal Parameters

RMSProp

AdaGrad white) keeps up with RMSProp (green) initially, as expected with the tuned learning rate and decay rate. But the *sums of gradient squared* for AdaGrad accumulate so fast that they soon become humongous (demonstrated by the sizes of the squares in the animation). They take a heavy toll and eventually AdaGrad practically stops moving. RMSProp, on the other hand, has kept the squares under a manageable size the whole time, thanks to the decay rate. This makes RMSProp faster than AdaGrad.

RMSProp (green) vs AdaGrad (white). The first run just shows the balls; the second run also shows the sum of gradient squared represented by the squares.



Finding Optimal Parameters

ADAM

- Adam (**Ada**ptive **M**oment Estimation) Combines RMSProp with momentum

$$\boldsymbol{v}(t+1) = \beta_1 \boldsymbol{v}(t) + (1-\beta_1)\frac{\partial\, L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)$$

Momentum

$$\boldsymbol{G}(t+1) = \beta_2 \boldsymbol{G}(t) + (1-\beta_2)\left(\frac{\partial\, L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)\right)^{\mathbf{2}}$$

AdaGrad / RMSProp

$$\boldsymbol{W}(t+1) = \boldsymbol{W}(t) - \frac{\alpha}{\sqrt{\boldsymbol{G}(t+1) + \boldsymbol{\epsilon}}}\boldsymbol{v}(t+1)$$

- Adam gets the speed from momentum and the ability to adapt gradients in different directions from RMSProp. The combination of the two makes it powerful.

Kingma and Ba, “Adam: A method for stochastic optimization”, ICLR 2015

Finding Optimal Parameters

ADAM

$$\boldsymbol{v}(t+1) = \beta_1 \boldsymbol{v}(t) + (1-\beta_1)\frac{\partial\, L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)$$

Momentum

Typically, $\beta_1 = 0.9$ and $\beta_2 = 0.999$

$\boldsymbol{v}(0) = \boldsymbol{G}(0) = 0$

$$\boldsymbol{v}(1) = (1-\beta_1)\frac{\partial\, L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(0)$$

$$\boldsymbol{G}(t+1) = \beta_2 \boldsymbol{G}(t) + (1-\beta_2)\left(\frac{\partial\, L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)\right)^2$$

AdaGrad / RMSProp

$$\boldsymbol{G}(1) = (1-\beta_2)\left(\frac{\partial\, L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(0)\right)^2$$

$$\boldsymbol{W}(t+1) = \boldsymbol{W}(t) - \frac{\alpha}{\sqrt{\boldsymbol{G}(t+1)+\boldsymbol{\epsilon}}}\boldsymbol{v}(t+1)$$

$\boldsymbol{v}(t+1)$: first moment

$\boldsymbol{G}(t+1)$: second moment

$\boldsymbol{v}(1)$ and/or $\boldsymbol{G}(1)$ will be tiny values, $\frac{\alpha}{\sqrt{\boldsymbol{G}(1)+\epsilon}}\boldsymbol{v}(1)$ is unstable at t=1

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Finding Optimal Parameters

ADAM

$$\boldsymbol{v}(t+1) = \beta_1 \boldsymbol{v}(t) + (1-\beta_1)\frac{\partial\, L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)$$

Momentum

$$\boldsymbol{G}(t+1) = \beta_2 \boldsymbol{G}(t) + (1-\beta_2)\left(\frac{\partial\, L(\boldsymbol{\theta})}{\partial \boldsymbol{W}}(t)\right)^2$$

AdaGrad / RMSProp

$$\hat{\boldsymbol{v}}(t+1) = \frac{\boldsymbol{v}(t+1)}{1-\beta_1^t} \quad \hat{\boldsymbol{G}}(t+1) = \frac{\boldsymbol{G}(t+1)}{1-\beta_2^t}$$

Bias correction

$$\boldsymbol{W}(t+1) = \boldsymbol{W}(t) - \frac{\alpha}{\sqrt{\boldsymbol{G}(t+1)+\epsilon}}\boldsymbol{v}(t+1)$$

Typically, $\beta_1 = 0.9$ and $\beta_2 = 0.999$

- $1-\beta_1^t$ and $1-\beta_2^t$ are small scalars, $\hat{\boldsymbol{v}}(t+1)$ and $\hat{\boldsymbol{G}}(t+1)$ will be reasonable values
- $\frac{\alpha}{\sqrt{\boldsymbol{G}(t+1)+\epsilon}}\boldsymbol{v}(t+1)$ is more stable

Kingma and Ba, "Adam: A method for stochastic optimization", ICLR 2015

Second-order Optimization

Newton's Method

$$f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \cdots,$$

Newton's method is a local optimization scheme based on the ***second order*** Taylor series approximation

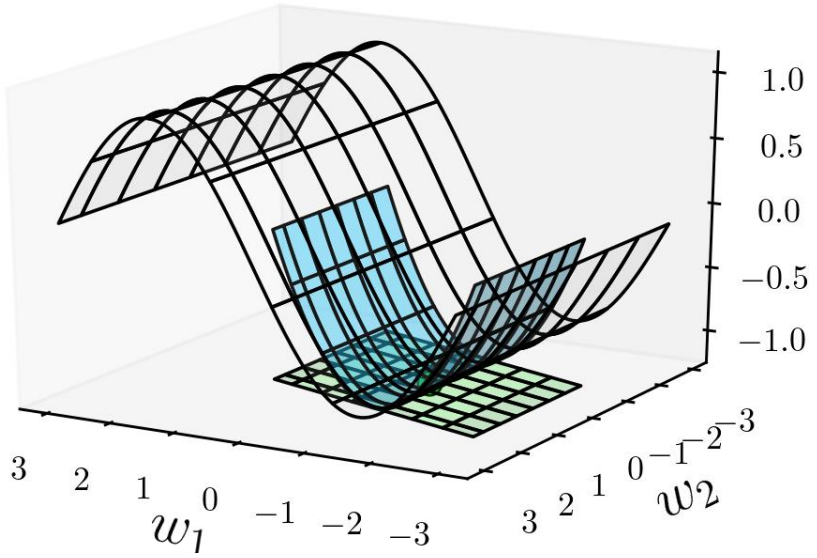
- The second order Taylor series approximation: quadratic part

$$L(\boldsymbol{\theta} + \Delta\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}) + \Delta\boldsymbol{\theta}^T \nabla_\theta L(\theta) + \frac{1}{2}\Delta\boldsymbol{\theta}^{\boldsymbol{T}}\boldsymbol{H}(\boldsymbol{\theta})\Delta\boldsymbol{\theta}$$

- $\nabla_\theta L(\boldsymbol{\theta})$ is the gradient, $\boldsymbol{H}(\boldsymbol{\theta}) = \nabla_\theta^2 L(\boldsymbol{\theta})$ is the second order derivatives of $L(\boldsymbol{\theta})$,which is also called Hessian matrix

$$\boldsymbol{H}(\boldsymbol{\theta}) = \nabla_\theta^2 L(\boldsymbol{\theta}) = \begin{pmatrix} \nabla^2_{\theta_1^2} L(\boldsymbol{\theta}) & \dots & \nabla^2_{\theta_1\theta_P} L(\boldsymbol{\theta}) \\ \vdots & \vdots & \vdots \\ \nabla^2_{\theta_P\theta_1} L(\boldsymbol{\theta}) & \dots & \nabla^2_{\theta_P^2} L(\boldsymbol{\theta}) \end{pmatrix}$$

- Assumes that $L(\boldsymbol{\theta})$ is *twice* differentiable
- Quadratic approximation
- Better local approximation of $L(\boldsymbol{\theta})$ than only using first order gradient.



Quadratic function (blue):

$$L(\boldsymbol{\theta}) + \Delta\boldsymbol{\theta}^T \nabla_\theta L(\theta) + \frac{1}{2}\Delta\boldsymbol{\theta}^{\boldsymbol{T}}\boldsymbol{H}(\boldsymbol{\theta})\Delta\boldsymbol{\theta}$$

Second-order Optimization

Newton's Method

matrix is positive semi-definite when it is symmetric with non-negative eigenvalues

- Unlike a hyperplane, a quadratic function does not have a 'steepest descent direction' to move to minima. However, a quadratic has global minima when it is *convex*.
- Note: the Hessian matrix is positive semi-definite, i.e., $\Delta\boldsymbol{\theta}^{\boldsymbol{T}}\boldsymbol{H}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} \geq 0$ for any $\Delta\boldsymbol{\theta}$. Thus, $L(\boldsymbol{\theta}+\Delta\boldsymbol{\theta})$ describes a *convex* parabola
- To find the minimum of convex $L(\boldsymbol{\theta}+\Delta\boldsymbol{\theta})$, we set its first derivative $\nabla_{\Delta\boldsymbol{\theta}} L(\boldsymbol{\theta}+\Delta\boldsymbol{\theta}) = \mathbf{0}$ and solve the optimal $\Delta\boldsymbol{\theta}$:

$$L(\boldsymbol{\theta}+\Delta\boldsymbol{\theta}) \approx L(\boldsymbol{\theta}) + \Delta\boldsymbol{\theta}^{T}\nabla_{\theta}L(\boldsymbol{\theta}) + \frac{1}{2}\Delta\boldsymbol{\theta}^{\boldsymbol{T}}\boldsymbol{H}(\boldsymbol{\theta})\Delta\boldsymbol{\theta}$$

$$\nabla_{\Delta\boldsymbol{\theta}} L(\boldsymbol{\theta}+\boxed{\Delta\boldsymbol{\theta}) = \nabla_{\theta}L(\theta) + \boldsymbol{H}(\boldsymbol{\theta})\Delta\boldsymbol{\theta} = 0}$$

$$\Longrightarrow \Delta\boldsymbol{\theta} = -\big(\boldsymbol{H}(\boldsymbol{\theta})\big)^{-1}\nabla_{\theta}L(\boldsymbol{\theta})$$

- Thus, the Newton's Method takes the form:

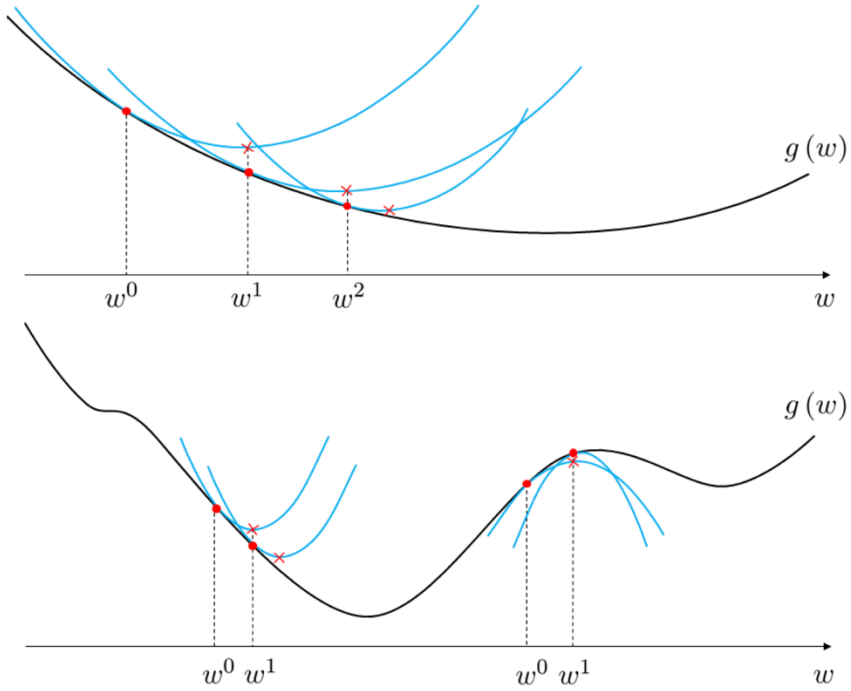
$$\boldsymbol{\theta}^{\boldsymbol{t+1}} = \boldsymbol{\theta}^{\boldsymbol{t}} - \boldsymbol{\alpha}\big(\boldsymbol{H}(\boldsymbol{\theta})\big)^{-1}\nabla_{\theta}L(\boldsymbol{\theta})$$

Gradient descent:
$\boldsymbol{\theta}^{\boldsymbol{t+1}} = \boldsymbol{\theta}^{\boldsymbol{t}} - \boldsymbol{\alpha}\nabla_{\theta}L(\boldsymbol{\theta}^{\boldsymbol{t}})$

Second-order Optimization

Newton's Method

- Newton's method is a powerful algorithm particularly for minimizing *convex* functions.
  - It uses quadratic, which is a more precise approximation than linear approximation at each step
  - it is often more effective than gradient descent as it requires fewer steps for convergence
- However, it is more difficult to use Newton's method to optimize non-convex functions.
  - At concave portions of such a function, the algorithm can climb to a local maximum or oscillate.



Second-order Optimization

Newton's Method

- Newton's method requires far more memory and computation than a first order algorithm.
  - The Hessian is a *P*×*P* matrix of second derivative. Computing its inverse-matrix causes scaling issues with input dimension

$$\boldsymbol{\theta^{t+1}} = \boldsymbol{\theta^{t}} - \boldsymbol{\alpha}\left(\boldsymbol{H}(\boldsymbol{\theta})\right)^{-1}\nabla_{\theta}L(\boldsymbol{\theta})$$

Where $\boldsymbol{H}(\boldsymbol{\theta}) = \nabla^2_{\theta}L(\boldsymbol{\theta}) = \begin{pmatrix} \nabla^2_{\theta_1^2}L(\boldsymbol{\theta}) & \dots & \nabla^2_{\theta_1\theta_P}L(\boldsymbol{\theta}) \\ \vdots & \vdots & \vdots \\ \nabla^2_{\theta_P\theta_1}L(\boldsymbol{\theta}) & \dots & \nabla^2_{\theta_P^2}L(\boldsymbol{\theta}) \end{pmatrix}$

Second-order Optimization

BFGS Algorithm

$$\boldsymbol{\theta^{t+1} = \theta^{t} - \alpha\left(H(\theta)\right)^{-1}\nabla_{\theta}L(\theta)}$$

- Quasi-Newton methods (Broyden–Fletcher–Goldfarb–Shanno (BFGS) most popular):
  - Approximate the inverse Hessian with rank 1 updates over time ($O(P^2)$).

- **L-BFGS** (Limited memory BFGS):
  - Does not form or store the full inverse Hessian.
  - Works very well in full batch, while not suitable for mini-batch setting.

Optimization Methods

Summary

- Different optimization algorithms behave differently depending on loss landscape
- **Adam** is a good default adaptive learning method.
- **SGD with Momentum** can outperform Adam, but requires more tuning of learning rate
- If **full batch** update is **affordable**, **L-BFGS** is a good method

Exercise

LeNet

1. Implement LeNet from scratch and train on the Fashion-MNIST dataset. Batch size = 256, learning rate for stochastic gradient descent = 0.9, epochs = 10, using cross entropy loss.
   I. Replace the average pooling with max pooling. What happens?
   II. Try to construct a more complex network based on LeNet to improve its accuracy.
      I. Adjust the convolution window size.
      II. Adjust the number of output channels.
      III. Adjust the activation function (e.g., ReLU).
      IV. Adjust the number of convolution layers.
      V. Adjust the number of fully connected layers.
      VI. Adjust the learning rates and other training details (e.g., initialization and number of epochs.)
   III. Try out the improved network on the original MNIST dataset.
   IV. Display the activations of the first and second layer of LeNet for different inputs (e.g., sweaters and coats).

Exercise

AlexNet

1. Implement AlexNet from scratch and train on the Fashion-MNIST dataset (up-sample images to 224 x 224). Batch size = 128, learning rate for stochastic gradient descent = 0.01, epochs = 10, using cross entropy loss.
   I. Try increasing the number of epochs. Compared with LeNet, how are the results different? Why?
   II. AlexNet may be too complex for the Fashion-MNIST dataset.
      I. Try simplifying the model to make the training faster, while ensuring that the accuracy does not drop significantly.
      II. Design a better model that works directly on 28×28 images.
   III. Modify the batch size, and observe the changes in accuracy and GPU memory.
   IV. Analyze computational performance of AlexNet.
      I. What is the dominant part for the memory footprint of AlexNet?
      II. What is the dominant part for computation in AlexNet?
      III. How about memory bandwidth when computing the results?
   V. Apply dropout and ReLU to LeNet-5. Does it improve? How about preprocessing?

Exercise

VGG

1. Write a function to implement a VGG Block with the following signature:

   ```
   def vgg_block(num_convs, in_channels, out_channels)
   ```

2. Write a function that uses vgg_block to implement the VGG architecture.
3. Train VGG-11 on Fashion-MNIST. Batch size = 128, learning rate for stochastic gradient descent = 0.05, epochs = 10, using cross entropy loss.
   I. When printing out the dimensions of the layers we only saw 8 results rather than 11. Where did the remaining 3 layer information go?
   II. Compared with AlexNet, VGG is much slower in terms of computation, and it also needs more GPU memory. Analyze the reasons for this.
   III. Try changing the height and width of the images in Fashion-MNIST from 224 to 96. What influence does this have on the experiments?
   IV. Refer to Table 1 in the VGG paper [Simonyan & Zisserman, 2014] to construct other common models, such as VGG-16 or VGG-19.

Exercise

GoogLeNet

1. Write a class to implement an Inception Block
2. Implement GoogLeNet from scratch using the Inception block
3. Train GoogLeNet on Fashion-MNIST. Batch size = 128, learning rate for stochastic gradient descent = 0.1, epochs = 10, using cross entropy loss.
    I. There are several iterations of GoogLeNet. Try to implement and run them. Some of them include the following:
        I. Add a batch normalization layer [Ioffe & Szegedy, 2015], as described later in Section 7.5.
        II. Make adjustments to the Inception block [Szegedy et al., 2016].
        III. Use label smoothing for model regularization [Szegedy et al., 2016].
        IV. Include it in the residual connection [Szegedy et al., 2017], as described later in Section 7.6.
    II. What is the minimum image size for GoogLeNet to work?
    III. Compare the model parameter sizes of AlexNet & VGG with GoogLeNet. How does the latter network architecture significantly reduce the model parameter size?

Exercise

ResNet

1. Write a class to implement a Residual Block
2. Implement ResNet from scratch using the Residual block
3. Train ResNet on Fashion-MNIST. Batch size = 256, learning rate for stochastic gradient descent = 0.05, epochs = 10, using cross entropy loss.
   I. What are the major differences between the Inception block in Fig. 7.4.1 and the residual block? After removing some paths in the Inception block, how are they related to each other?
   II. Refer to Table 1 in the ResNet paper [He et al., 2016a] to implement different variants.
   III. For deeper networks, ResNet introduces a “bottleneck” architecture to reduce model complexity. Try to implement it.
   IV. In subsequent versions of ResNet, the authors changed the “convolution, batch normalization, and activation” structure to the “batch normalization, activation, and convolution” structure. Make this improvement yourself. See Figure 1 in [He et al., 2016b] for details.
   V. Why can’t we just increase the complexity of functions without bound, even if the function classes are nested?

Overview

In this Lecture..

Locally Connected Layer

Introduction to Convolutional Neural Networks

Layers used to build Convolutional Neural Networks

Examples of Deep CNNs Architectures

Training CNNs

Visualization of Convolutional Neural Networks

- Visualizing filters
- Visualizing activations

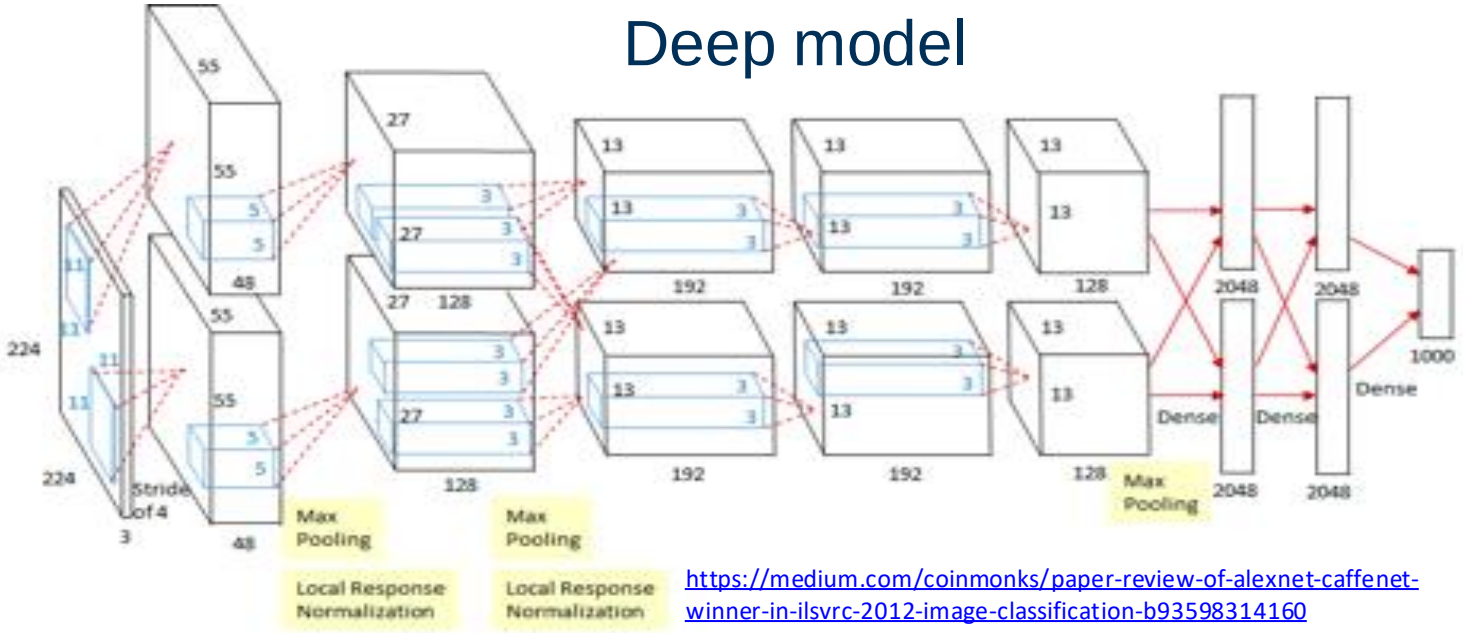
Explainability

Explainability in CNNs

Data

Input Image:
3 x 224 x 224

Deep model



https://medium.com/coinmonks/paper-review-of-alexnet-caffenet-winner-in-ilsvrc-2012-image-classification-b93598314160

Output

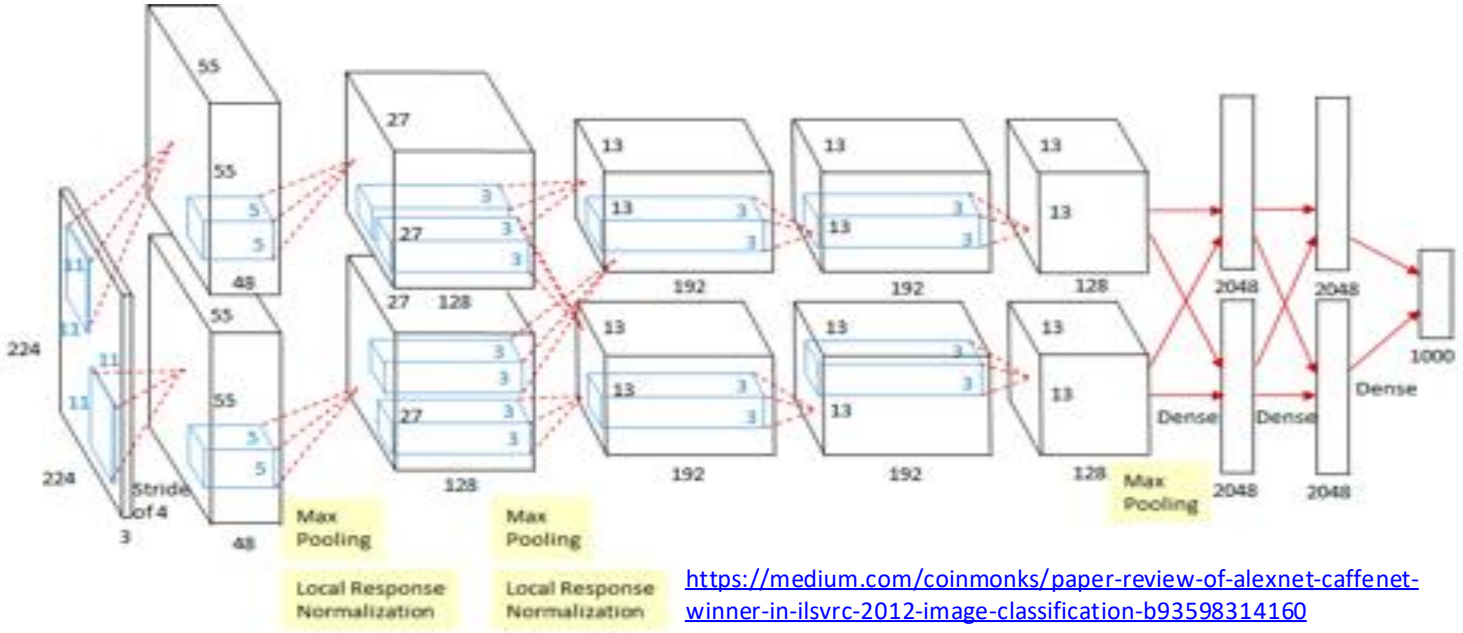
Output class scores:
1000 numbers
(trained on ImageNet)

Not Explainable

Krizhevsky et al. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012)

Explainability

Explainability in CNNs



https://medium.com/coinmonks/paper-review-of-alexnet-caffenet-winner-in-ilsvrc-2012-image-classification-b93598314160

Output class scores:
1000 numbers
(trained on ImageNet)

Input Image:
3 x 224 x 224

What are these layers looking for?
Are they explainable?

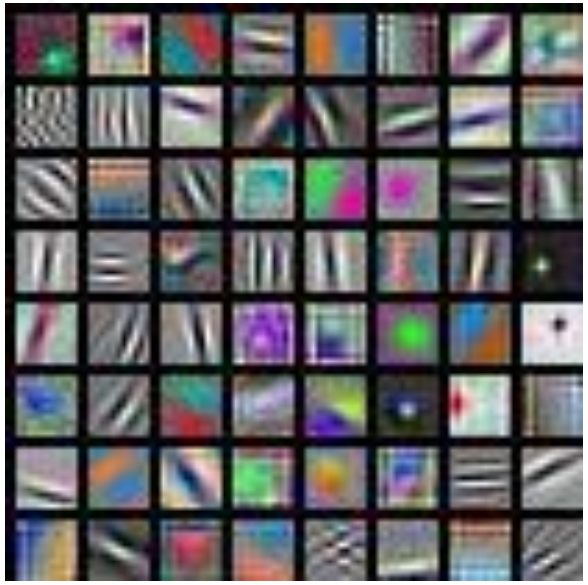
Material adapted from : Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

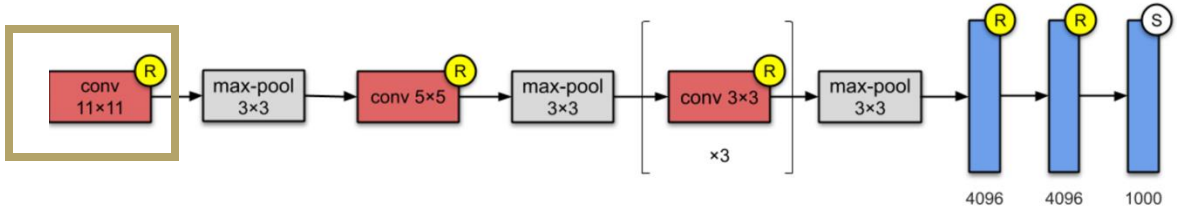
Krizhevsky et al. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012)

Visualizing CNNs

Visualizing Filters in First Layers

Filters = Weights





AlexNet

Input Image:
3 x 224 x 224

AlexNet:
64 x 3 x 11 x 11

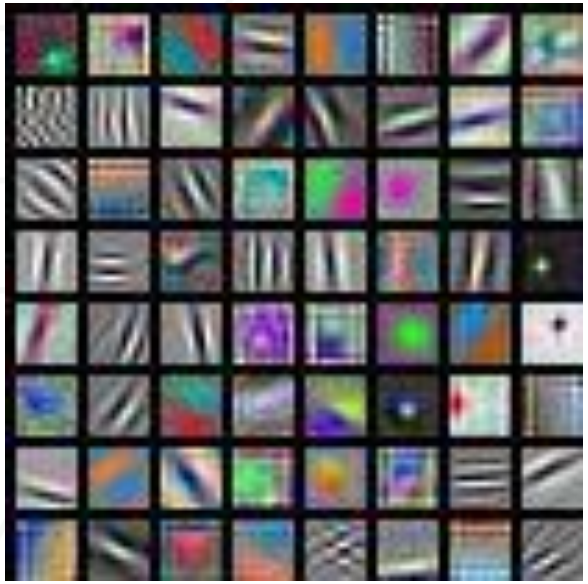
Filters always extend the full depth of the input volume

- 64 filters in the first convolutional layer
- Filter size: 11 x 11 x 3 (visualize as RGB images)
- Filters are looking for ***low-level*** **oriented edges, color blobs, textures, background etc.**

Krizhevsky et al. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012)

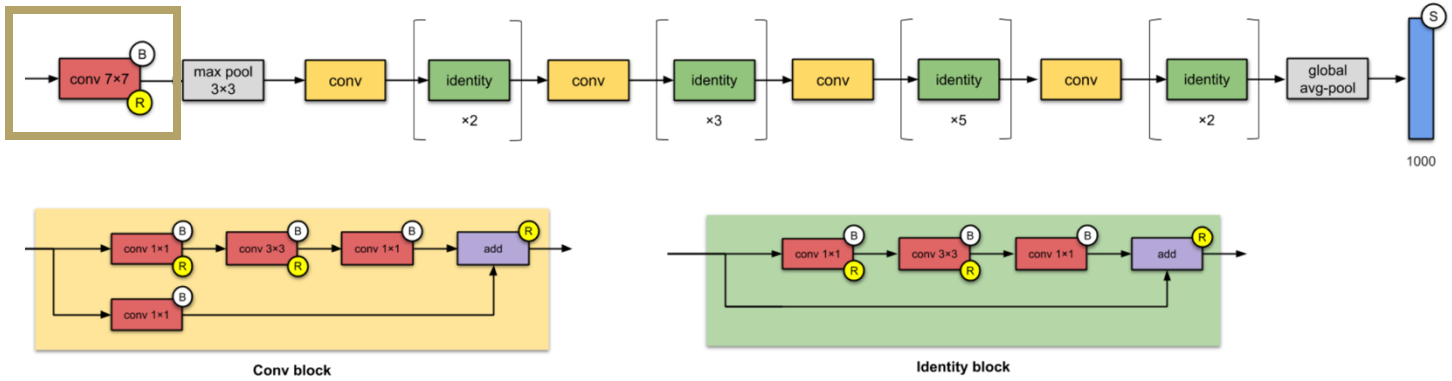
Visualizing CNNs

Visualizing Filters in First Layers



AlexNet:
64 x 3 x 11 x 11

ResNet-18:
64 x 3 x 7 x 7



ResNet

- Filters in the first convolutional layers **across different architectures learn similar patterns**

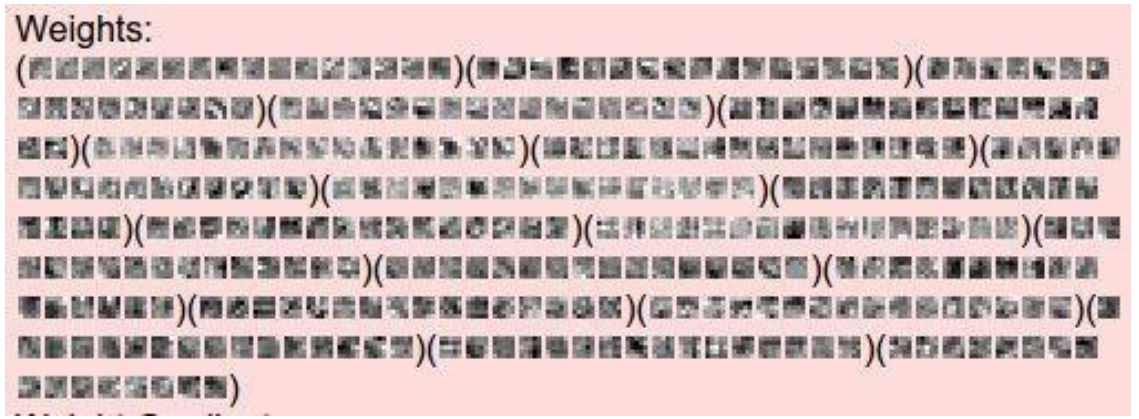
Krizhevsky et al. "Imagenet classification with deep convolutional neural networks." *Advances in neural information processing systems* 25 (2012)

Visualizing CNNs

Visualizing Filters in Intermediate Layers

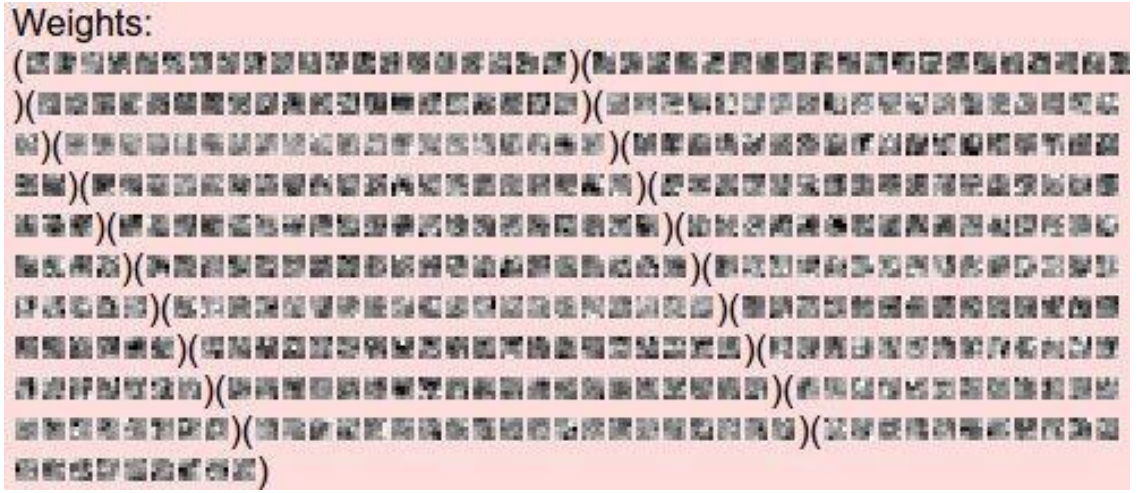
Visualize the filters (raw weights)

Conv layer 1 weights
16 x 3 x 7 x 7
(visualize as RGB images)



Conv layer 2 weights
20 x 16 x 7 x 7
(visualize as 16 grayscale images)

Filters in **higher** convolutional **layers are not as interpretable as** filters in **the first layer**



Conv layer 3 weights
20 x 20 x 7 x 7
(visualize as 20 grayscale images)

Visualizing CNNs

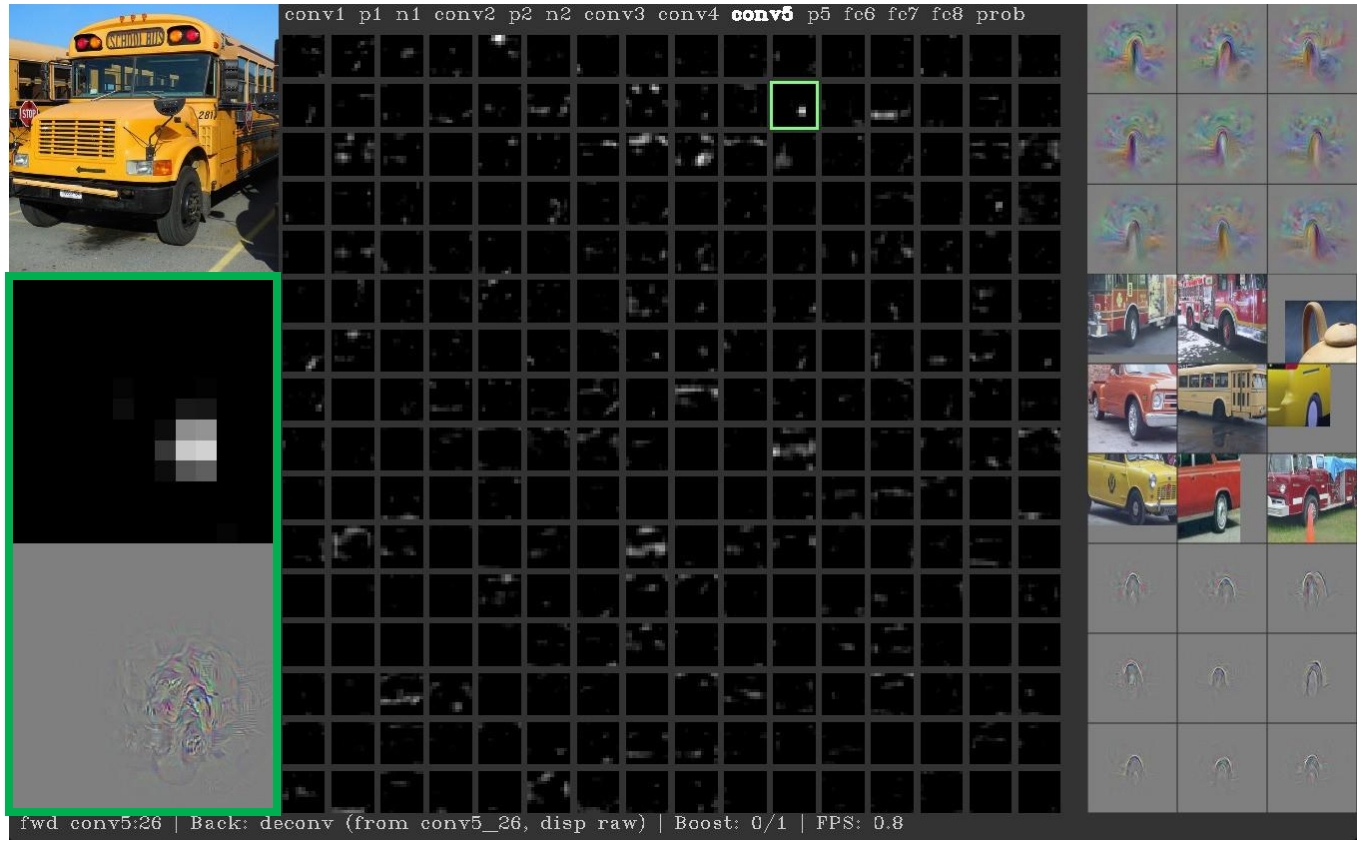
Visualizing Filters in Intermediate Layers

Intermediate layers:

- Weights: not very interpretable
- Activations: interpretable

The filter in conv 5 layer is activated when it sees a wheel

Conv 5



AlexNet

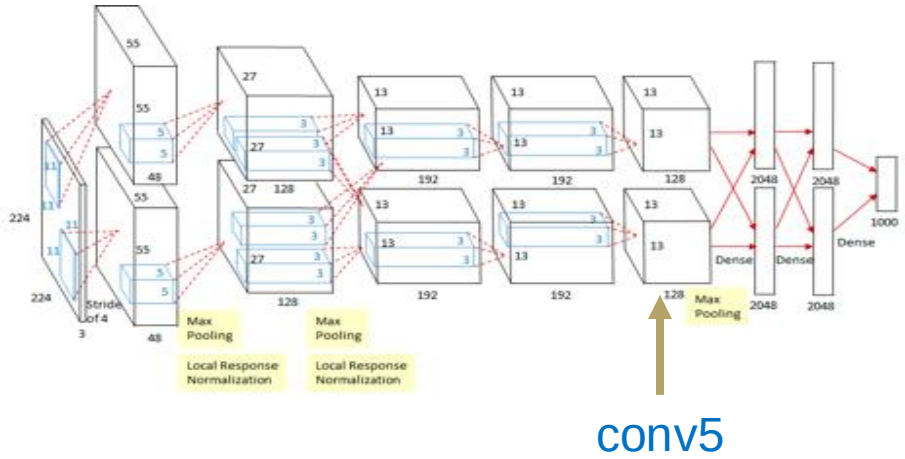
Material adapted from : Fei-Fei Li, Justin Johnson, Serena Yeung, CS 231n

Yosinski et al, "Understanding Neural Networks Through Deep Visualization", ICML DL Workshop 2014.

Visualizing CNNs

Visualizing Filters in Intermediate Layers

- Apart from visualizing the intermediate activations, we can also visualize what **similar visual patterns** in images that cause the maximum activations of certain neurons



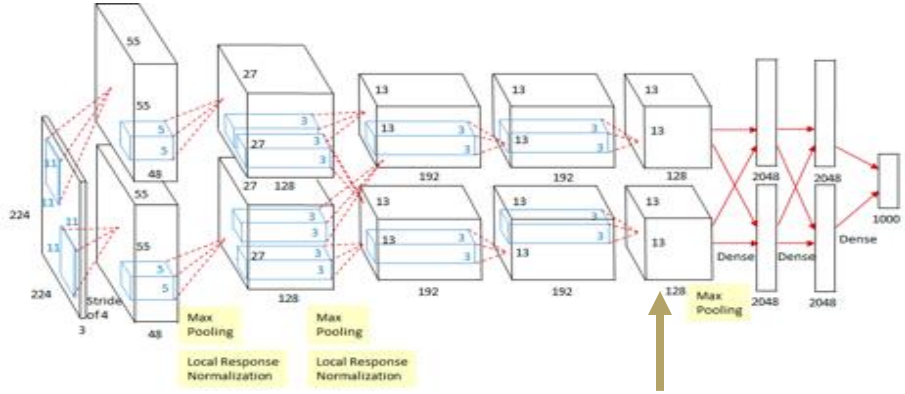
- Maximally Activating Patches:
  - Image patches in the input that cause the maximum activations of certain filters

Springenberg et al, “Striving for Simplicity: The All Convolutional Net”, *ICLR* Workshop 2015

Visualizing CNNs

Visualizing Maximally Activating Patches

- Maximally Activating Patches:
  - Image patches in the input that cause the maximum activations of certain filters
- Obtaining Maximally Activating Patches:
  - Pick activations in a layer, e.g., conv5 (128 x 13 x 13), pick one of the channels 17/128
  - Feed forward many images through the network, record values of the chosen channel
  - Visualize image patches that correspond to **maximal activation**
- Maximally activating patches share **similar visual patterns**



conv5

Maximally activating patches
Each row corresponds to a particular neuron in conv5

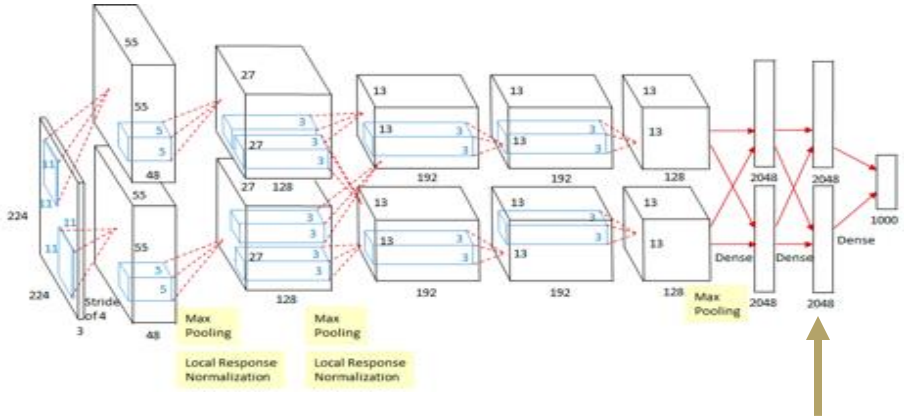
Springenberg et al, "Striving for Simplicity: The All Convolutional Net", *ICLR* Workshop 2015

Visualizing CNNs

Visualizing Last Layer Activations

- We can group the images that have similar class-specific information by exploring last layer activations

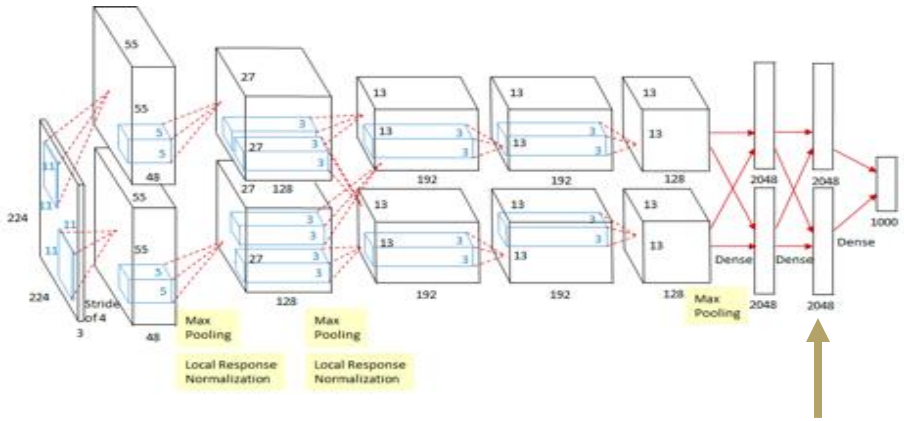
- Last layer activations (embedding):
  - 4096-dimensional feature vector for an image (layer immediately before the classifier)
  - Representations of *entire input images* instead of specific patches
  - Similar embeddings correspond to same classes of input images



Visualizing CNNs

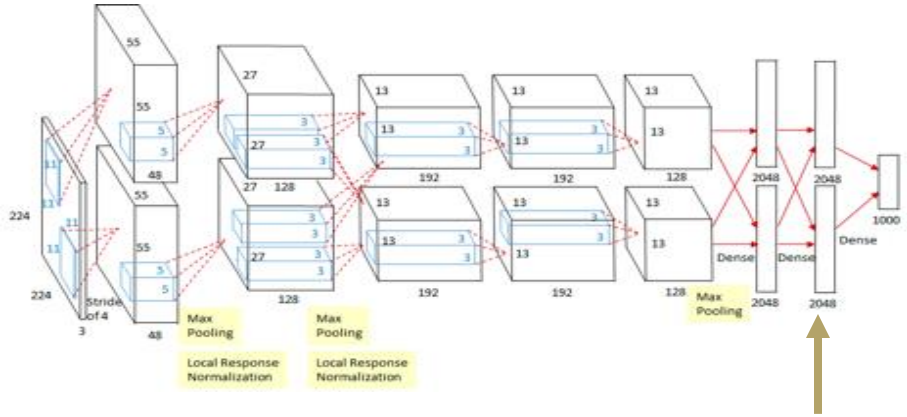
Visualizing Last Layer Activations

- Last layer activations (embedding):
  - 4096-dimensional feature vector for an image (layer immediately before the classifier)
  - Representations of *entire input images* instead of specific patches
  - Similar embeddings correspond to same classes of input images
- Feed forward many images through the network, collect the final layer feature vectors
- Visualize input images that have similar last layer embeddings



Visualizing CNNs

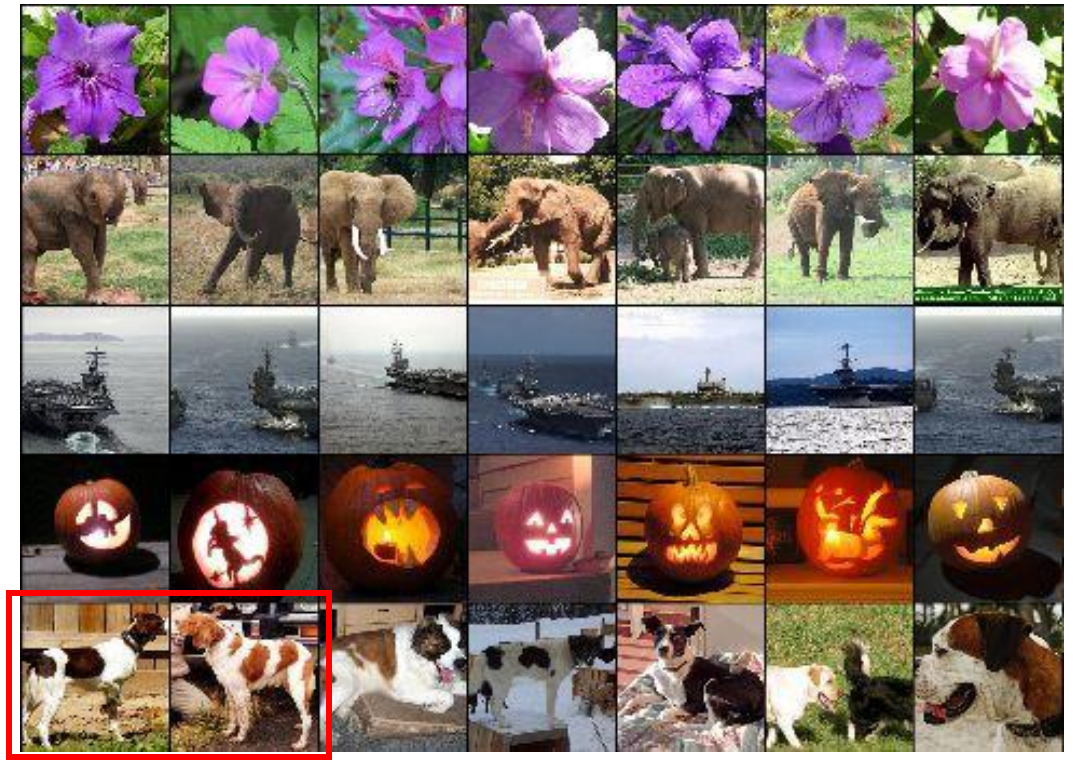
Visualizing Last Layer Activations



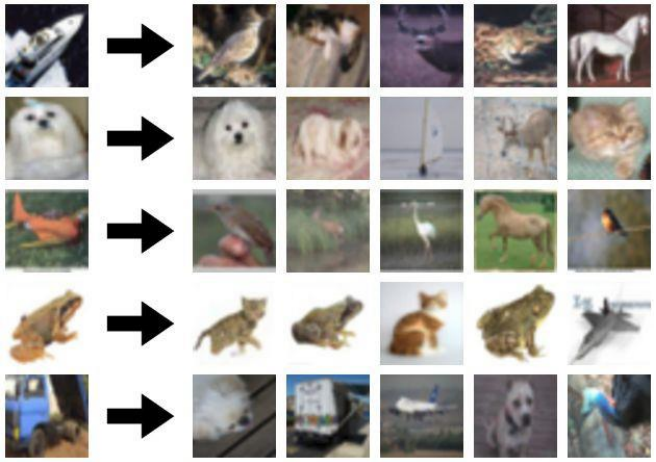
fully-connected 7

- Last layer embedding:
  - 4096-dimensional feature vector for an image

Test image L2 Nearest neighbors in **feature** space



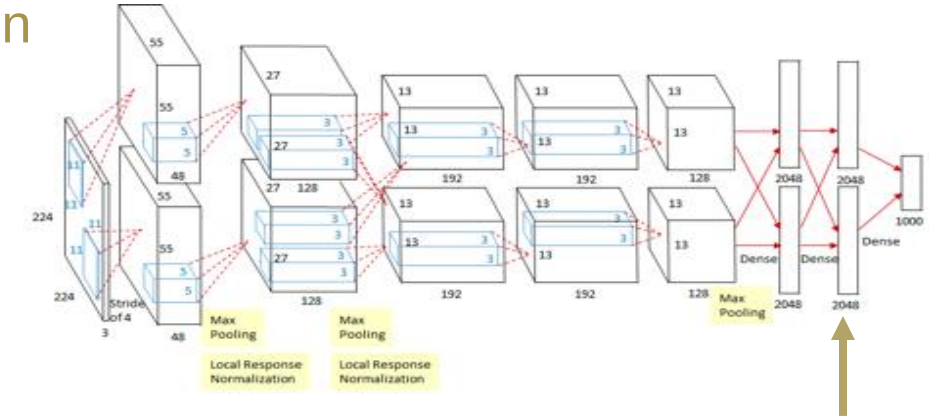
Recall: Nearest neighbors in **pixel** space



- The **features** of the two dogs **share L2 similarity**
- The images do not share L2 similarity due to horizontally flipped poses

Visualizing CNNs

Visualizing Last Layer Activations via Dimensionality Reduction



fully-connected 7

- Last layer embedding:
  - 4096-dimensional feature vector for an image
- Visualize the “feature space” by reducing dimensionality of feature vectors from 4096 to 2 dimensions
  - Each 2-dim feature correspond to an input image

Van der Maaten and Hinton, “Visualizing Data using t-SNE”, JMLR 2008

Terminology

- $Distribution$: (sample space) the set of all possible samples
- $Dataset$: a set of samples drawn from a distribution
- $Batch$: a subset of samples drawn from the dataset
- $Sample$: a single data object represented as a set of features
- $Feature$: value of a single attribute, property, in a sample. Could be numeric or categorical.

Appendix A: Notations

- x_i: a single feature
- $\boldsymbol{x}_i$: feature vector (a data sample)
- $\boldsymbol{x}_{:,i}$: feature vector of all data samples
- $\boldsymbol{X}$: matrix of feature vectors (dataset)
- N: number of data samples
- $\boldsymbol{W}$: weight matrix
- $\boldsymbol{b}$: bias vector
- $\boldsymbol{v}(t)$: first moment at time t
- $\boldsymbol{G}(t)$: second moment at time t
- $\boldsymbol{H}(\boldsymbol{\theta})$: Hessian matrix
- P: number of features in a feature vector
- α: learning rate
- Bold letter/symbol: vector
- Bold capital letters/symbol: matrix