Mathematical foundations of Machine Learning 2025 – lesson 6

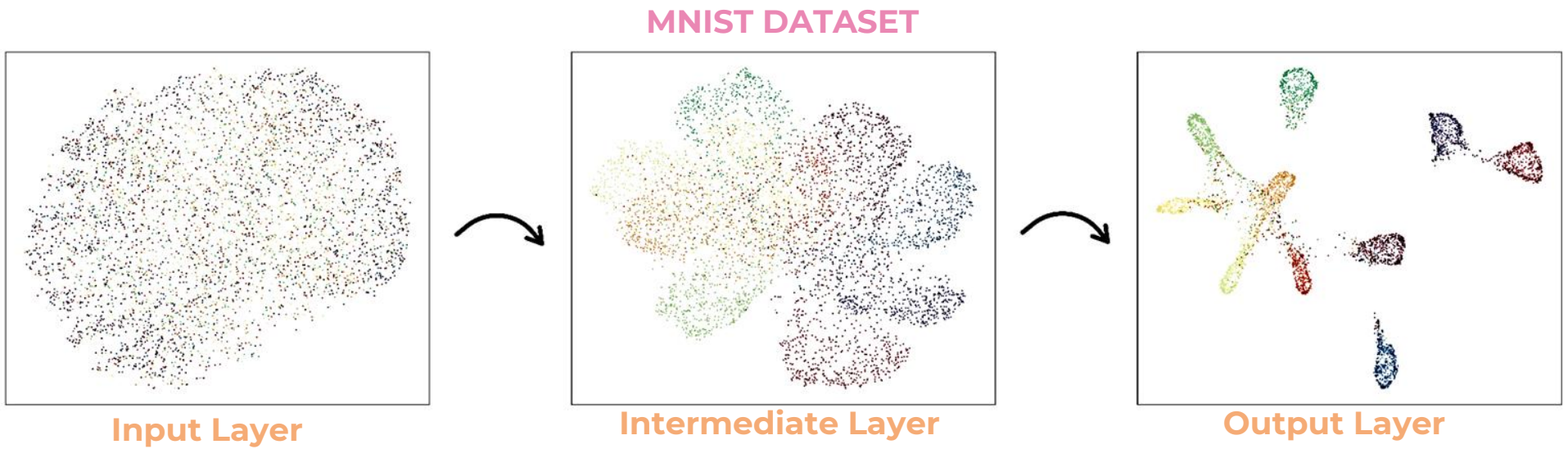
Shai Dekel

TEL AVIV UNIVERSITY

The role of deep learning

- Good separability in input feature space → classical ML models are sufficient
    - Logistic Regression, Support Vector Machines, Random Forest, Gradient Boosting, etc.

- **If not, can we transform to a better feature space through feature engineering/deep learning ?**



Deep Learning building blocks

Convolutions

Assignment: The convolution of $f, g \in L_1(\mathbb{R}^n)$ is defined by $f * g(x) := \int_{\mathbb{R}^n} f(x-y) g(y) dy$.

(i) Prove that $f * g \in L_1(\mathbb{R}^n)$,

(ii) Prove that $f * g = g * f$,

(iii) With the Fourier Transform defined by $\hat{f}(w) = \int_{\mathbb{R}^n} f(x) e^{-i\langle w,x \rangle} dx$, for $w \in \mathbb{R}^n$, show that

$$(f * g)\hat{}\,(w) = \hat{f}(w)\hat{g}(w), \quad \forall w \in \mathbb{R}^n.$$

(iv) Let $f \in L_1(\mathbb{R}^2)$ be a piecewise constant function. Design a 'filter' $g \in L_\infty(\mathbb{R}^2)$, with support in $[-\varepsilon/2, \varepsilon/2]^2$, for some $\varepsilon > 0$, such that $f * g$ is 'significant' only in ε neighborhoods of points where f has 'almost' vertical edges.

Discrete convolutions

1D discrete filter $g := \{g_k\}_{k=-M}^{M}$. The filter size is $2M+1$. Let $f = \{f_j\}_{j=-\infty}^{\infty}$. The discrete convolution is

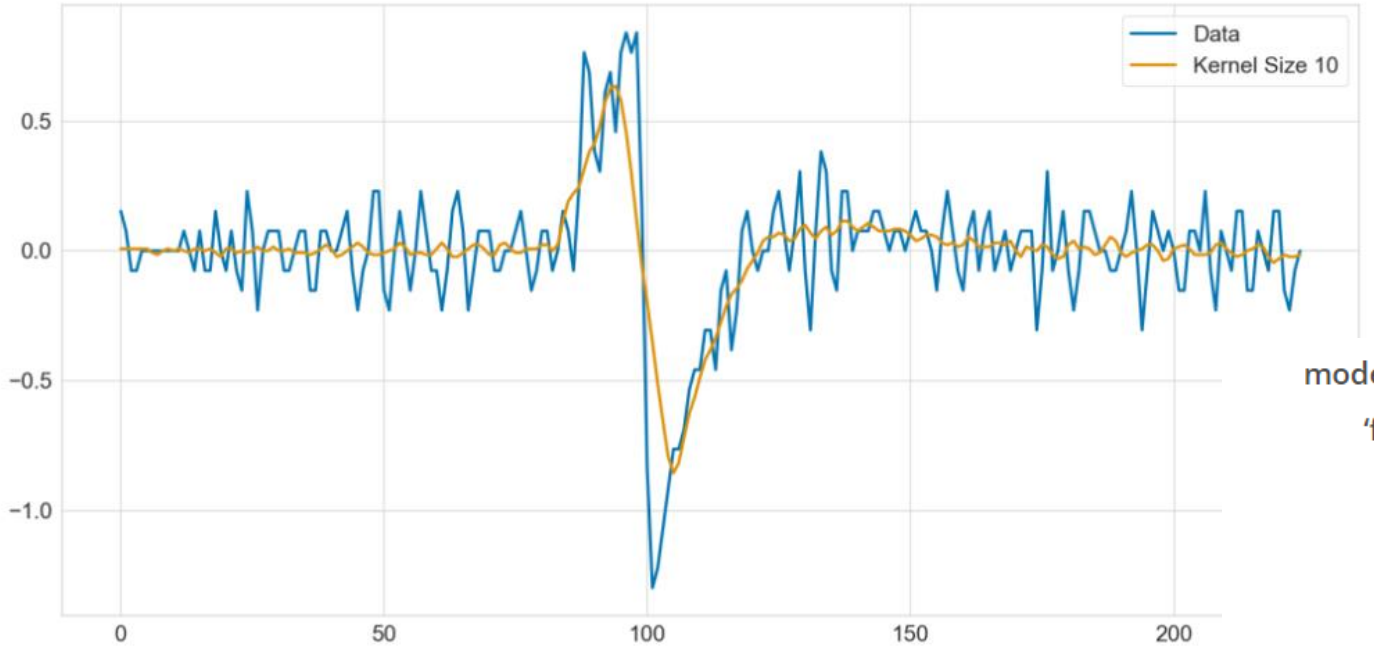
$$f * g(k) := \sum_{j=-\infty}^{\infty} f_j g_{k-j} .$$

Examples:

1. Smooth convolution, low-pass filter $g = (g_{-1}, g_0, g_1) = \left[\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\right]$.
2. High-pass filter $g = (g_{-1}, g_0, g_1) = \left[-\frac{1}{4}, \frac{1}{2}, -\frac{1}{4}\right]$. It has two vanishing moments:
   a. If $f_j = c,\ \forall j$, $f * g(k) = -\frac{1}{4}c + \frac{1}{2}c - \frac{1}{4}c = 0$,
   b. If $f_j = aj + b$, $f * g(k) = -\frac{1}{4}\big(a(k-1)+b\big) + \frac{1}{2}(ak+b) - \frac{1}{4}\big(a(k+1)+b\big) = 0$.

   It is designed to detect 'jumps'

Discrete convolutions




```python
import numpy as np

data = np.load("example_data.npy")
kernel_size = 10
kernel = np.ones(kernel_size) / kernel_size
data_convolved = np.convolve(data, kernel, mode='same')
```

mode : *{'full', 'valid', 'same'}, optional*

'full':

By default, mode is 'full'. This returns the convolution at each point of overlap, with an output shape of (N+M-1,). At the end-points of the convolution, the signals do not overlap completely, and boundary effects may be seen.

'same':

Mode 'same' returns output of length `max(M, N)`. Boundary effects are still visible.

'valid':

Mode 'valid' returns output of length `max(M, N) - min(M, N) + 1`. The convolution product is only given for points where the signals overlap completely. Values outside the signal boundary have no effect.

Discrete convolutions

2D discrete filter $g := \left\{ g_{k_1,k_2} \right\}_{k_1,k_2=-M}^{M}$. Filter size is $(2M+1)^2$. Let $f = \left\{ f_j \right\}_{j \in \mathbb{Z}^2}$. The discrete convolution is

$$f * g(k) = f * g(k_1, k_2) := \sum_{j \in \mathbb{Z}^2} f_j g_{k-j} .$$

Example Smooth convolution, low-pass filter

$$g = \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix}$$

Discrete convolutions

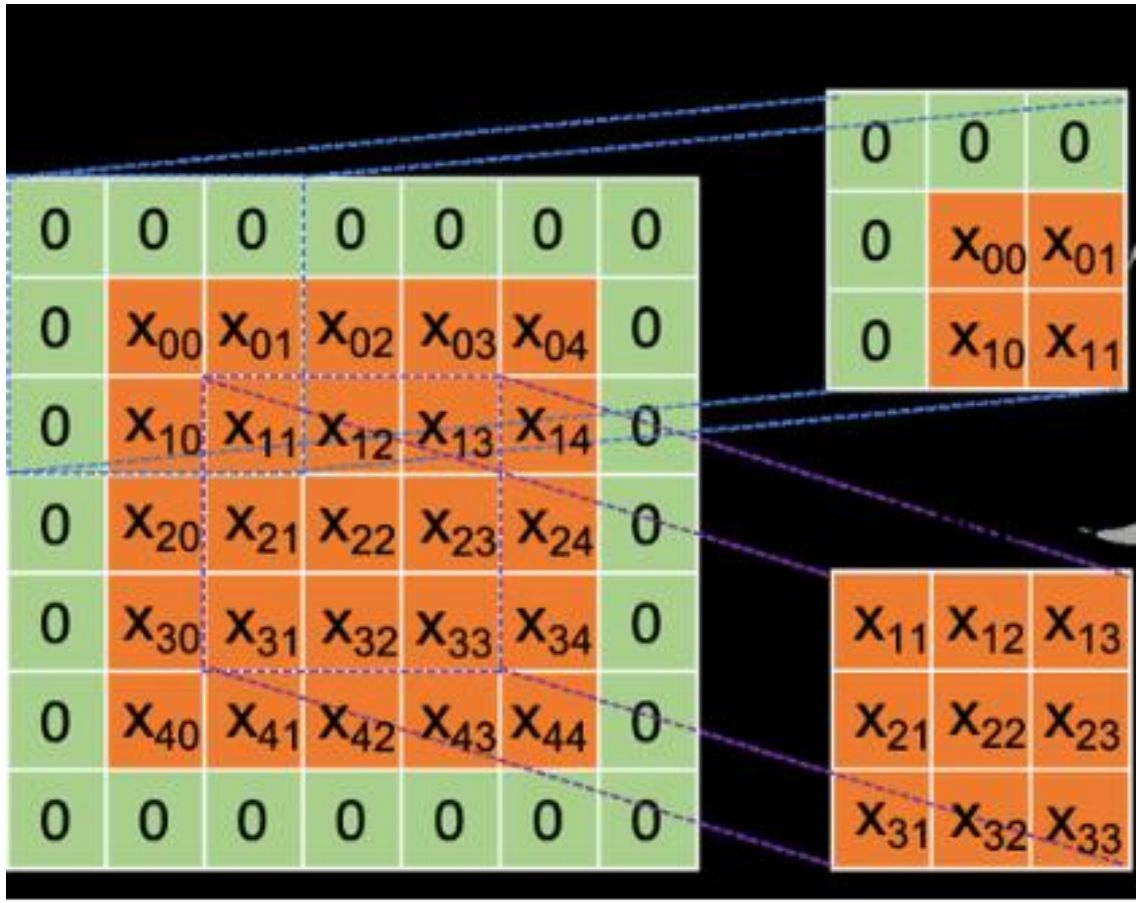
One can decompose a symmetric 2D filter through a tensor decomposition using two 1D filters $\tilde{g}_1, \tilde{g}_2$

$$f * g(k_1, k_2) = \left[\left[f(\cdot, k_2) * \tilde{g}_1\right](k_1, \cdot) * \tilde{g}_2\right](k_1, k_2)$$

Example $\tilde{g}_1 = \tilde{g}_2 = \tilde{g} = (\tilde{g}_{-1}, \tilde{g}_0, \tilde{g}_1) = \left[\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\right]$

$$\begin{aligned} f * g(k_1, k_2) = & \frac{1}{4}\left(\frac{1}{4} f_{k_1-1,k_2-1} + \frac{1}{2} f_{k_1-1,k_2} + \frac{1}{4} f_{k_1-1,k_2+1}\right) \\ & + \frac{1}{2}\left(\frac{1}{4} f_{k_1,k_2-1} + \frac{1}{2} f_{k_1,k_2} + \frac{1}{4} f_{k_1,k_2+1}\right) \\ & + \frac{1}{4}\left(\frac{1}{4} f_{k_1+1,k_2-1} + \frac{1}{2} f_{k_1+1,k_2} + \frac{1}{4} f_{k_1+1,k_2+1}\right) \end{aligned}$$

Zero padding for a 3x3 filter



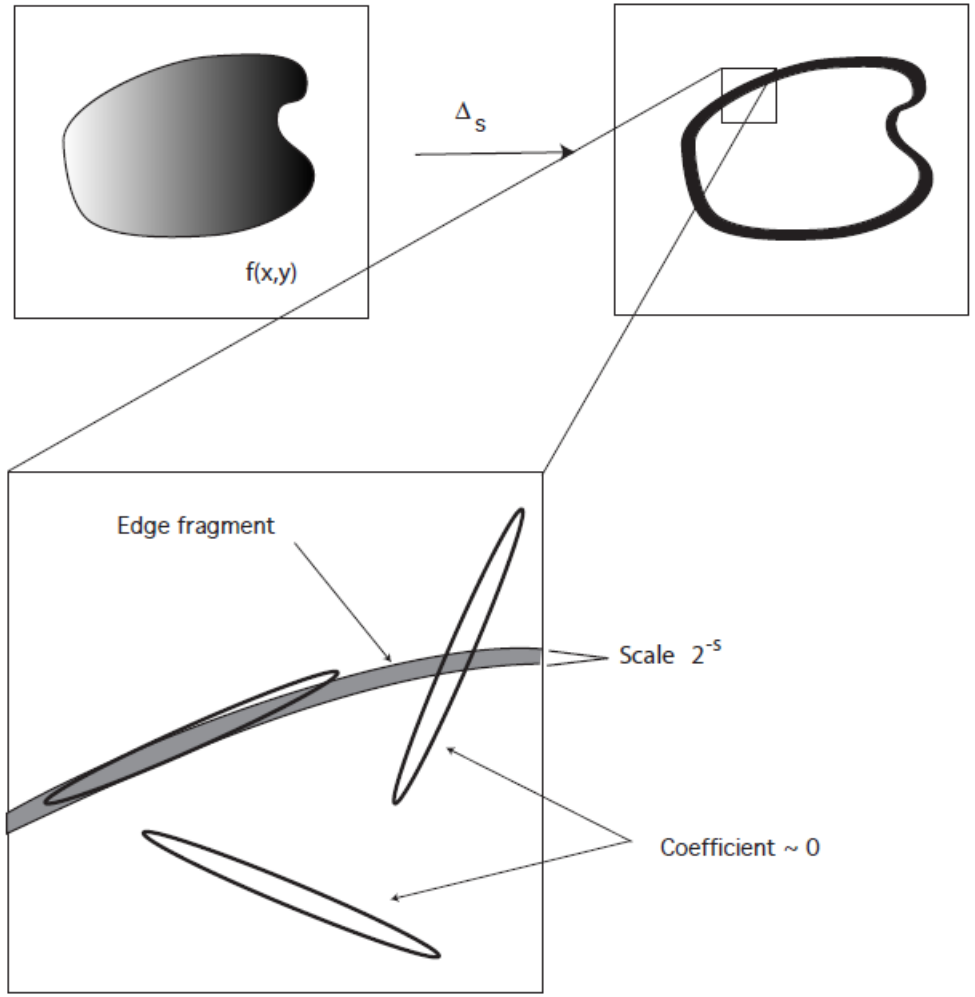
Mirror/reflection padding for a 3x3 filter

3	5	1
3	6	1
4	7	9

1	6	3	6	1	6	3
1	5	3	5	1	5	3
1	6	3	6	1	6	3
9	7	4	7	9	7	4
1	6	3	6	1	6	3

Classic convolutions – Curvelets

- Curvelets: a 2d “stable basis” designed to capture edge singularities
- Each element is associated with: location, frequency, directionality
- Coefficients of curvelets not located “on” or not aligned with edge singularity are “insignificant”



Curvelets – construction via tiling in Fourier domain

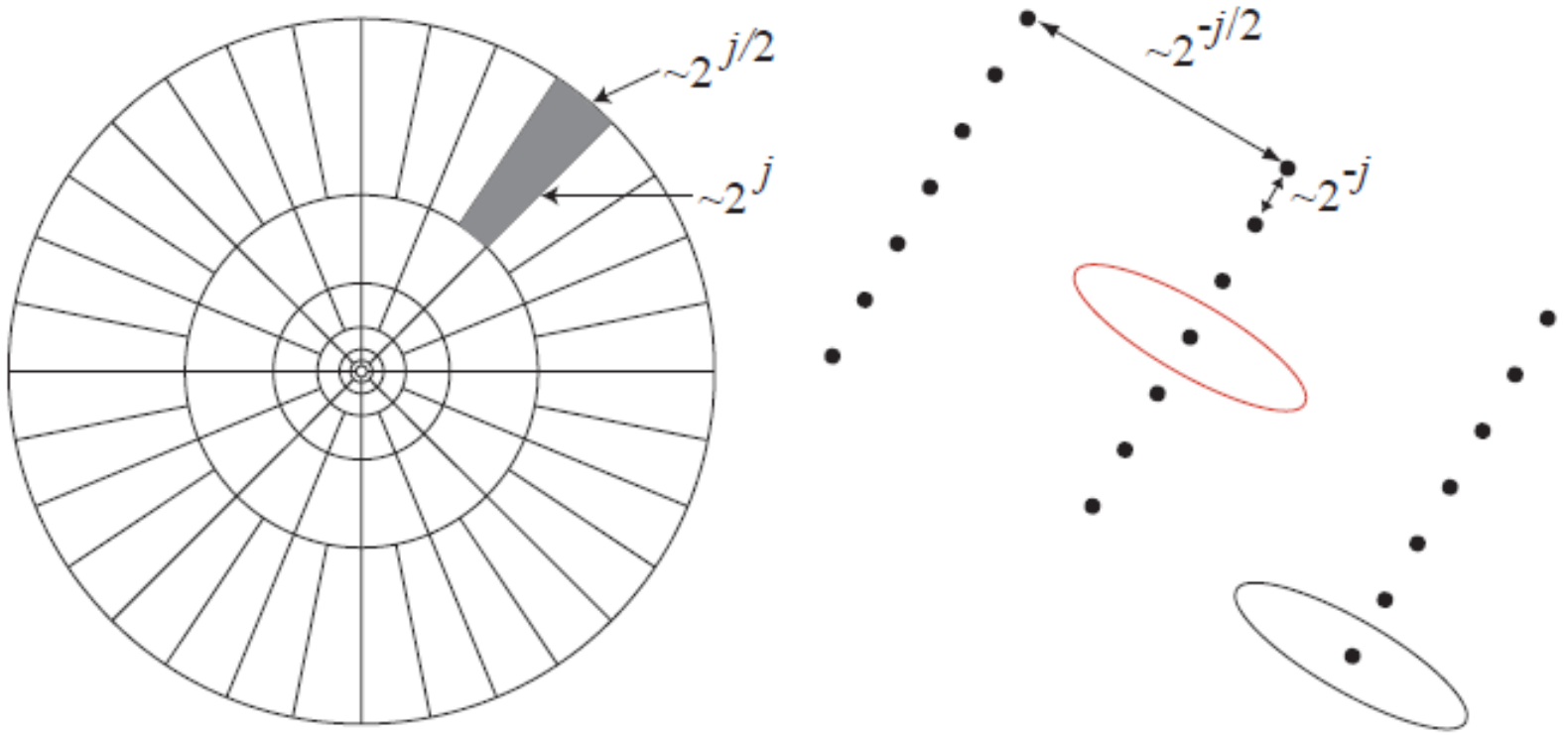


Figure 1: Curvelet tiling of space and frequency. The figure on the left represents the induced tiling of the frequency plane. In Fourier space, curvelets are supported near a "parabolic" wedge, and the shaded area represents such a generic wedge. The figure on the right schematically represents the spatial Cartesian grid associated with a given scale and orientation.

Classic convolutions – Curvelets

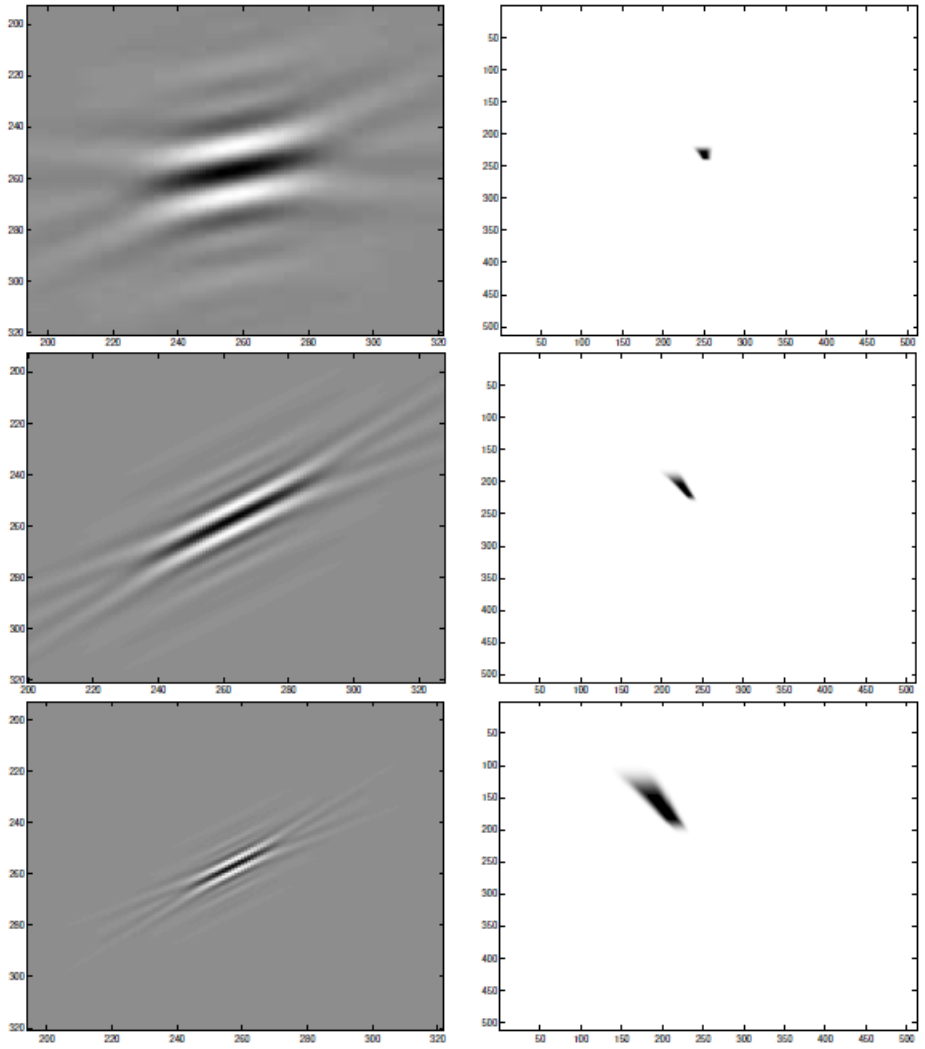
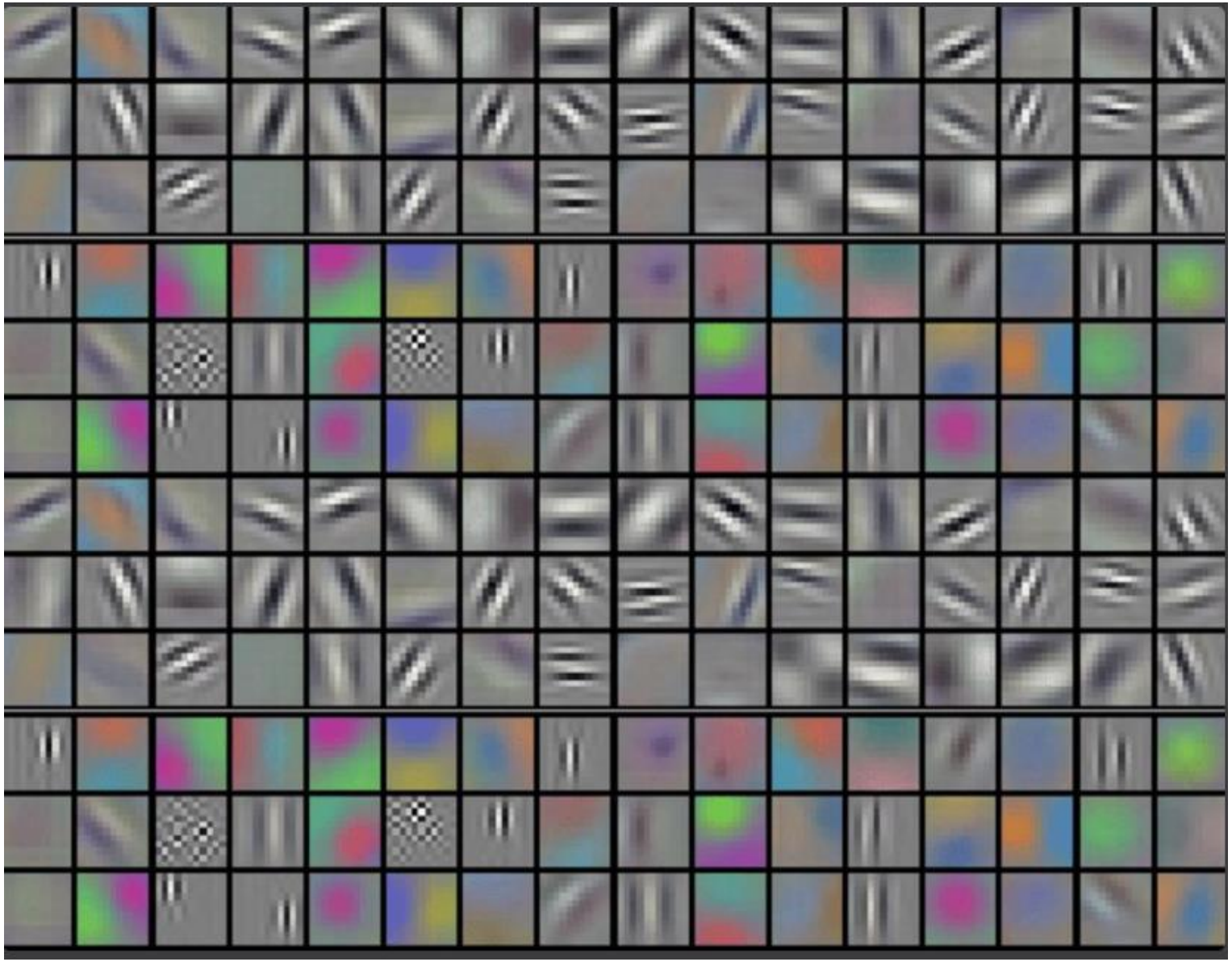


Figure 10: Curvelets at increasingly fine scales. The left panels represent curvelets (real part) in the spatial domain (as functions of the spatial variable x). The right panels show the modulus of the Fourier transform (as functions of the frequency variable ω). The color map is the same as in Figure 9.

Convolutions – through learning

- Convolution filters of first layer in a computer vision deep learning network: learning of edge & color detectors
- The coefficients of the filters are part of the “weights” of the network.



Learning filter decomposition

A trained “convolutional pipeline” for 14 unknowns, instead of 49…

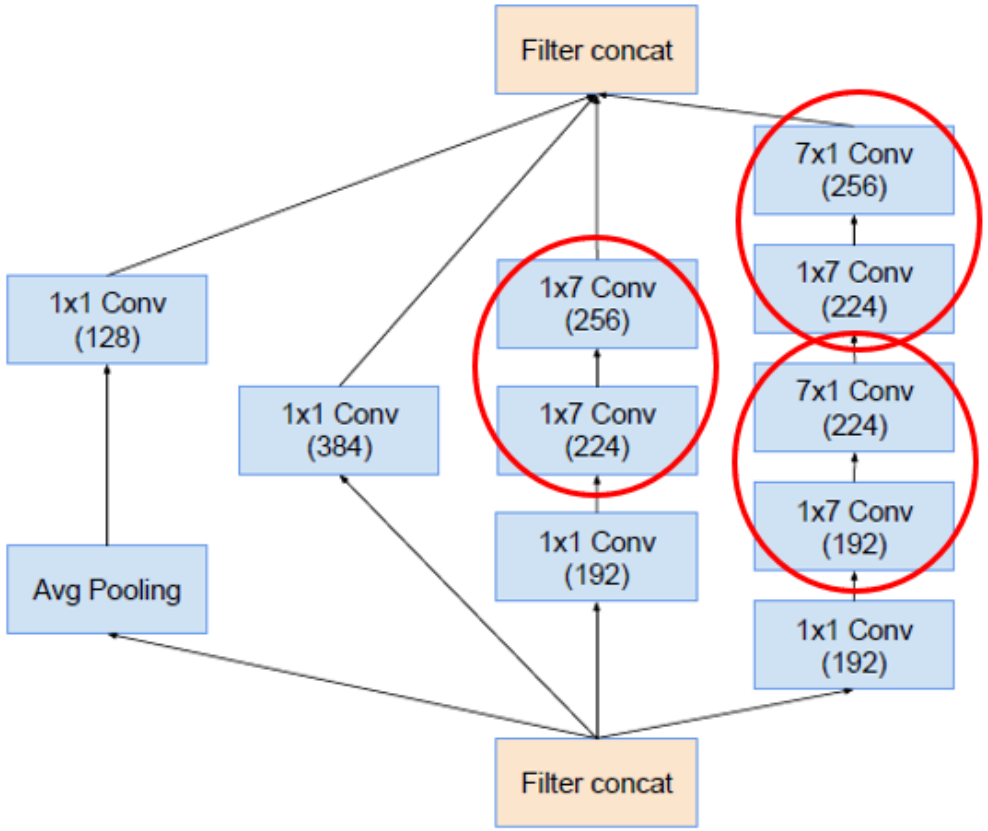


Figure 5. The schema for 17 × 17 grid modules of the pure Inception-v4 network. This is the Inception-B block of Figure 9.

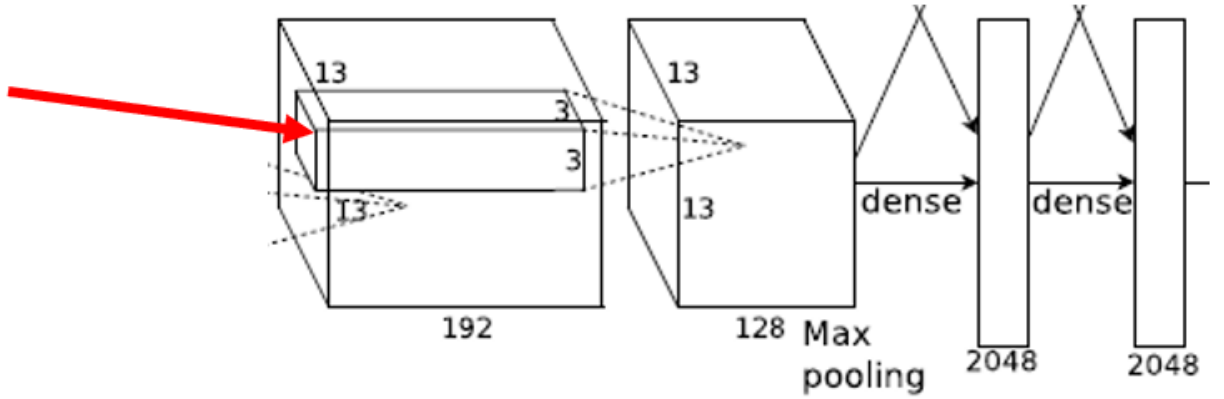
3D discrete convolutions

In computer vision architectures, we typically need 3d convolutions in the inner layers: x,y and z is the feature map (channel) dimensions

$$f * g(k) = f * g(k_1, k_2, k_3) := \sum_{j \in \mathbb{Z}^3} f_j g_{k-j} \cdot$$

The filter is typically localized in the x,y direction, but not in the z direction.

128 filters, each of dimension 3x3x192



Fully connected (dense) layers

Definition Let $M \in M_{n\times m}$ be a nonsingular matrix and $b \in \mathbb{R}^m$ a (bias) vector. The associated affine transform $A : \mathbb{R}^n \to \mathbb{R}^m$, is defined by $Ax = Mx + b$, $\forall x \in \mathbb{R}^n$.

Fully connected layer – every input vector element potentially contributes to any output vector element $x_{out} = Ax_{in} = Mx_{in} + b$.

Remark A Conv layer is a special case of a FC layer! allows to significantly reduce the number of (unknown) weights, based on the assumption of locality, i.e. the output (neurons) have a localization property, they are associated with visual elements in a certain neighborhood (whose support grows with the depth of the layers). So, they should only be affected by inputs in a certain neighborhood.

Non-linearities

A decomposition of affine transforms is an affine transform

$$A_2(A_1x) = M_2(A_1x) + b_2 = M_2(M_1x + b_1) + b_2 = (M_2M_1)x + (M_2b_1 + b_2).$$

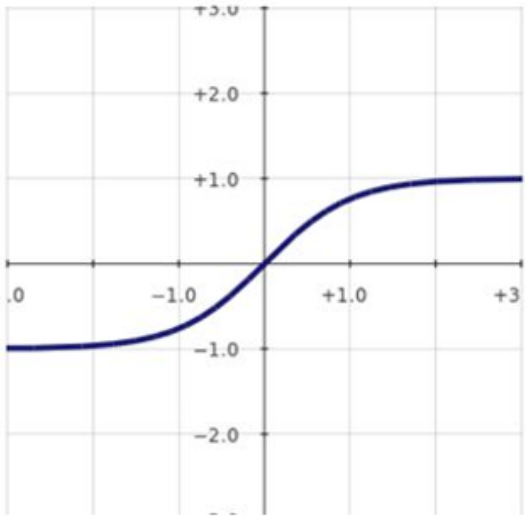
So, without any other functionality, a neural network essentially collapses into one affine transform.

Typical non-linearities

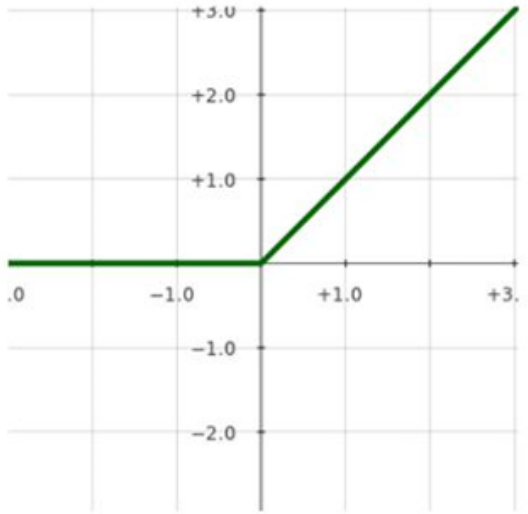
$$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{\frac{e^x - e^{-x}}{2}}{\frac{e^x + e^{-x}}{2}} = \frac{e^x - e^{-x}}{e^x + e^{-x}},$$

ReLU (Rectifier Linear Unit)

f(x) = tanh(x)



f(x) = max(0, x)



Applying non-linearities

Typically, non-linearities are applied pointwise after the affine operations (Conv or FC). So, if $\sigma(y) := (\mathrm{ReLU}(y_1), \ldots, \mathrm{ReLU}(y_m))$. A complete FC layer is then $\sigma(A(x)) = \sigma(Mx + b)$.

Non-linearities can be explained in two ways:

1. Neural Sciences approach - Simulation of neuron activation
2. Approximation-theoretical explanation of ReLu – The entire NN can be viewed as a continuous piecewise linear approximation over the original feature space. Why? Over sub-domains of the original feature subspace, the NN collapses into one local affine transform.

NN as piecewise linear approximation over the input space

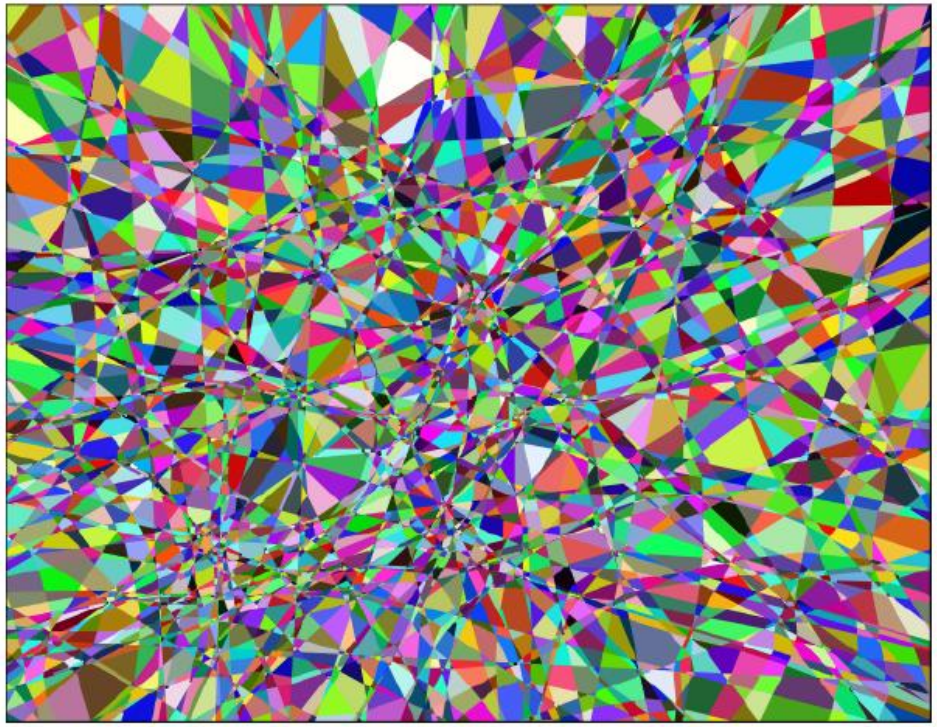
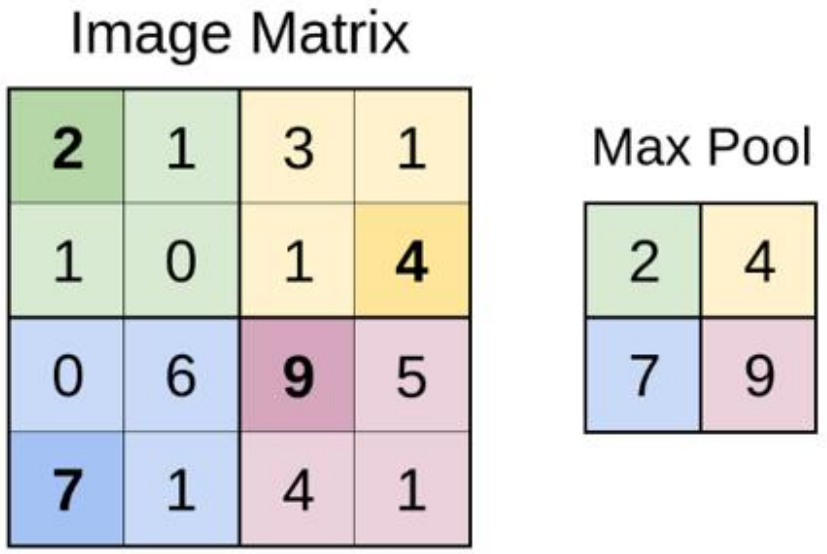


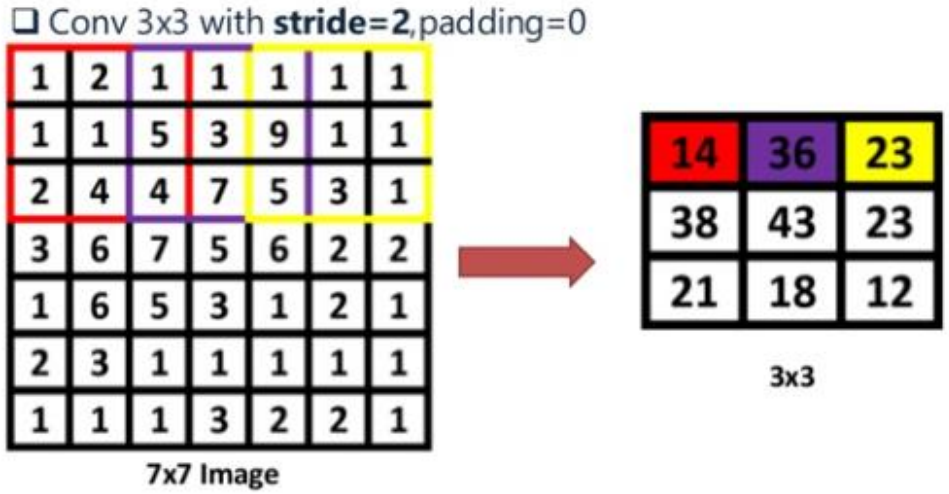
Figure 1. How many linear regions? This figure shows a two-dimensional slice through the 784-dimensional input space of vectorized MNIST, as represented by a fully-connected ReLU network with three hidden layers of width 64 each. Colors denote different linear regions of the piecewise linear network.

Dimension reduction operations

Typically applied after the non-linearities in encoder architectures used for classification



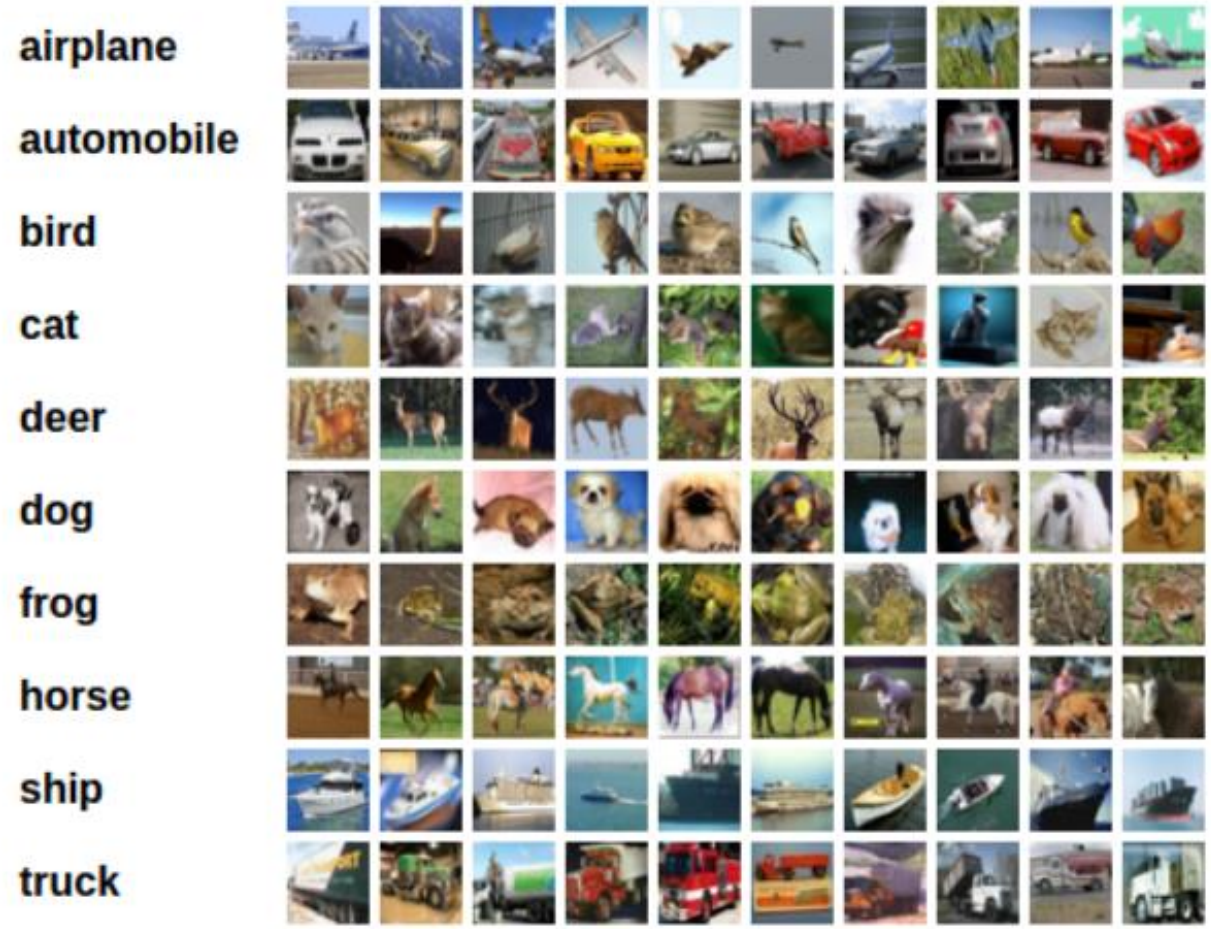
Max-pooling



Strides

Pytorch image CIFAR classification example

For this tutorial, we will use the CIFAR10 dataset. It has the classes: ‘airplane’, ‘automobile’, ‘bird’, ‘cat’, ‘deer’, ‘dog’, ‘frog’, ‘horse’, ‘ship’, ‘truck’. The images in CIFAR-10 are of size 3x32x32, i.e. 3-channel color images of 32x32 pixels in size.



Pytorch image CIFAR classification example

```python
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = torch.flatten(x, 1) # flatten all dimensions except batch
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x


net = Net()
```

CLASS torch.nn.Conv2d(*in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None*) [SOURCE]

Layer Dimensions:

Layer 0 - 3x32x32

Layer 1 - 6x28x28 (pooling) -> 6x14x14

Layer 2 - 16x10x10 (pooling) -> 16x5x5

Layer 3 - 120

Layer 4 - 84

Layer 5 – 10

Layer #weights:

Layer 0→1 - 6x3x5x5+6 = 456

Layer 1→2 - 16x6x5x5+16=2416

Layer 2→3 - 48,000+120 = 48,120

Layer 3→4 - 10,080+84 = 10,164

Layer 4→5 – 840+10=850

Pytorch image CIFAR classification example

$x = (x_1, \ldots, x_{10})$ is the representation at the last layer

$$\Pr(y = Y_k \mid x) := \frac{e^{-x(k)}}{\sum_{j=1}^{10} e^{-x(j)}}$$

Together with previous layer: Soft-Max

$$\text{Loss} = -\frac{1}{\#I} \sum_{i \in I} \log\left(\Pr(Y = y_i \mid x_i)\right)$$

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```

Pytorch image CIFAR classification example

```python
for epoch in range(2):  # loop over the dataset multiple times

    running_loss = 0.0
    for i, data in enumerate(trainloader, 0):
        # get the inputs; data is a list of [inputs, labels]
        inputs, labels = data

        # zero the parameter gradients
        optimizer.zero_grad()

        # forward + backward + optimize
        outputs = net(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        # print statistics
        running_loss += loss.item()
        if i % 2000 == 1999:    # print every 2000 mini-batches
            print(f'[{epoch + 1}, {i + 1:5d}] loss: {running_loss / 2000:.3f}')
            running_loss = 0.0

print('Finished Training')
```

```
[1,  2000] loss: 2.213
[1,  4000] loss: 1.887
[1,  6000] loss: 1.699
[1,  8000] loss: 1.597
[1, 10000] loss: 1.506
[1, 12000] loss: 1.471
[2,  2000] loss: 1.412
[2,  4000] loss: 1.373
[2,  6000] loss: 1.338
[2,  8000] loss: 1.301
[2, 10000] loss: 1.286
[2, 12000] loss: 1.291
Finished Training
```

IMAGENET

14,197,122 images, 21841 synsets indexed

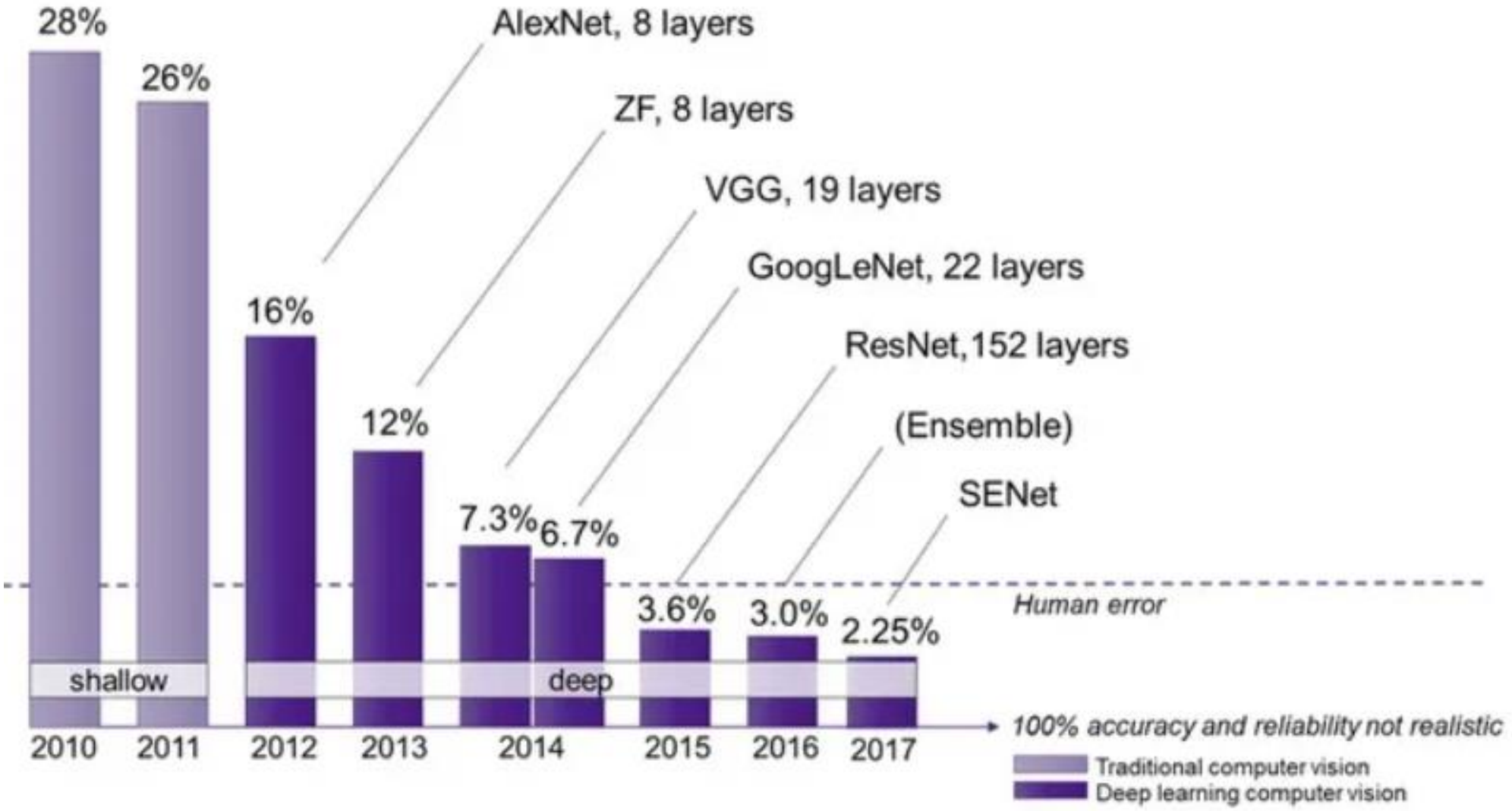
Explore Download Challenges Publications CoolStuff About

Not logged in. Login | Signup

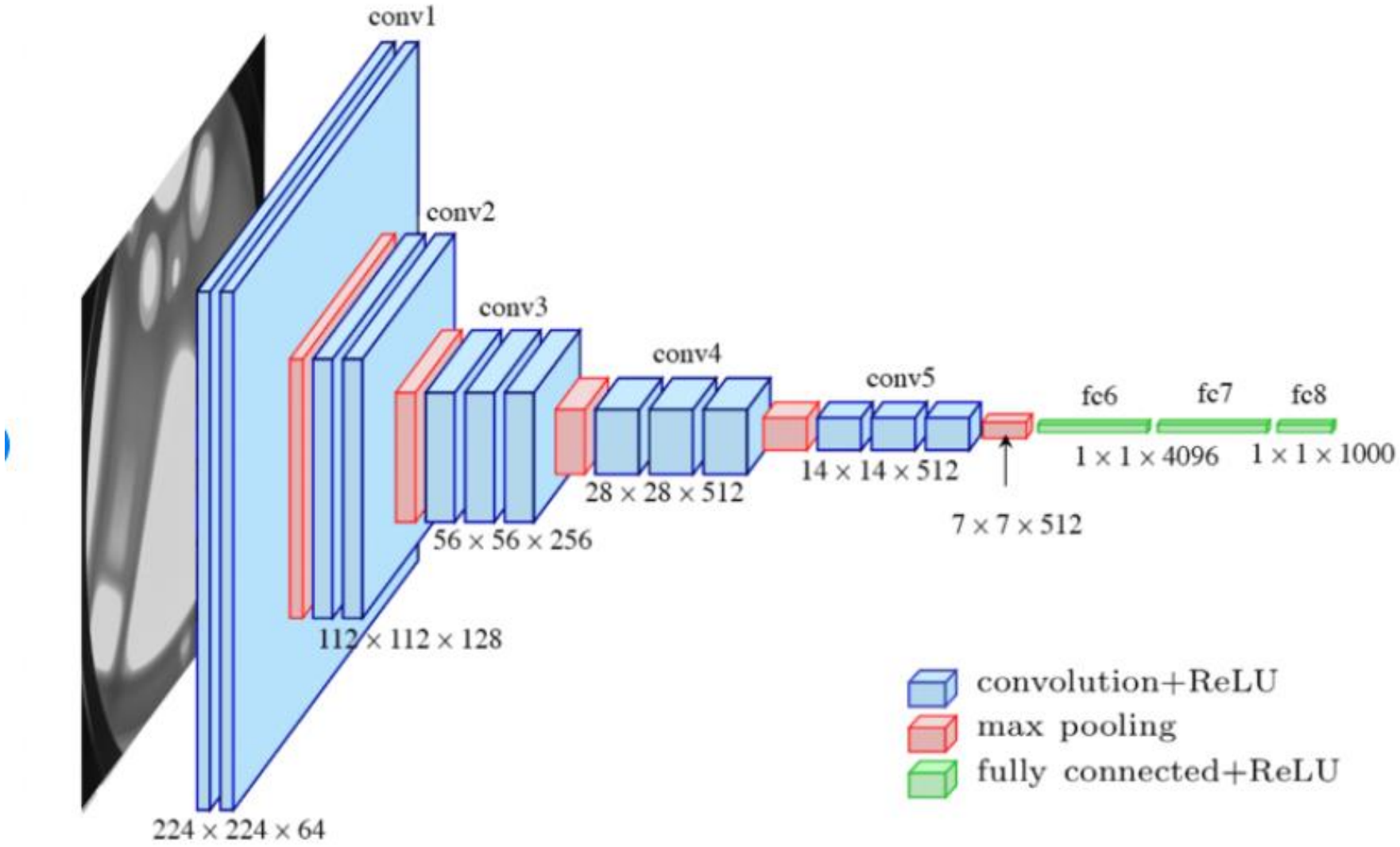
ImageNet is an image database organized according to the WordNet hierarchy (currently only the nouns), in which each node of the hierarchy is depicted by hundreds and thousands of images. Currently we have an average of over five hundred images per node. We hope ImageNet will become a useful resource for researchers, educators, students and all of you who share our passion for pictures.

Click here to learn more about ImageNet, Click here to join the ImageNet mailing list.

ImageNet classification of 1000 classes – top 5 results



VGG-16 Net (2015)



Residual Blocks

Deep Residual Learning for Image Recognition ← Cited 267,160 times from 2015

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

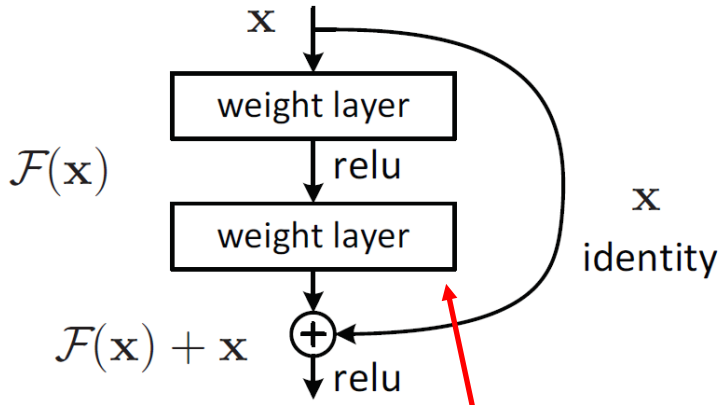


Figure 2. Residual learning: a building block.

Modeling residuals

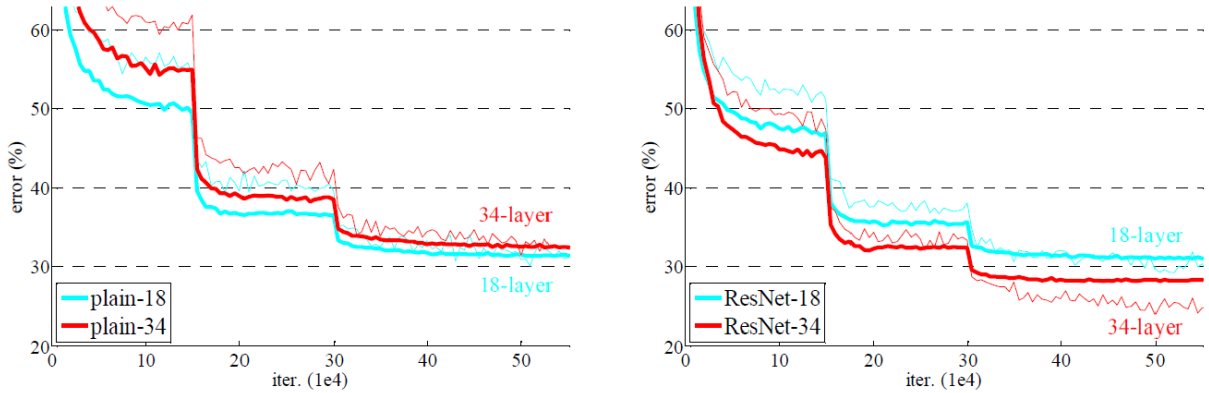


Figure 4. Training on **ImageNet**. Thin curves denote training error, and bold curves denote validation error of the center crops. Left: plain networks of 18 and 34 layers. Right: ResNets of 18 and 34 layers. In this plot, the residual networks have no extra parameter compared to their plain counterparts.

Inception Blocks (2016)

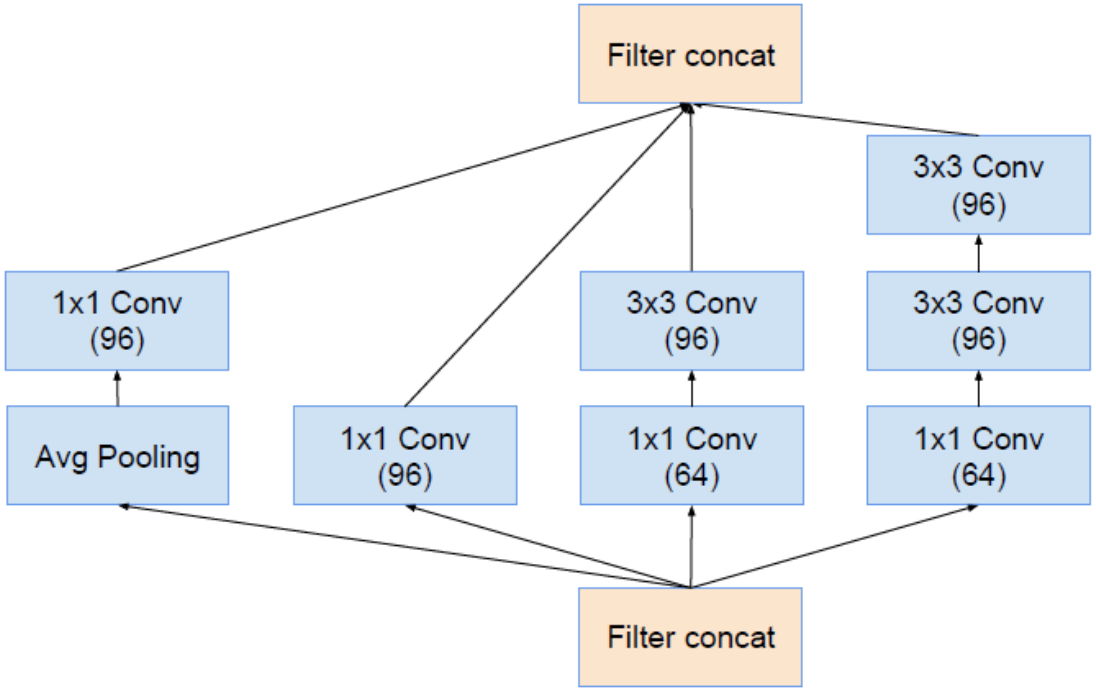


Figure 4. The schema for 35 × 35 grid modules of the pure Inception-v4 network. This is the Inception-A block of Figure 9.

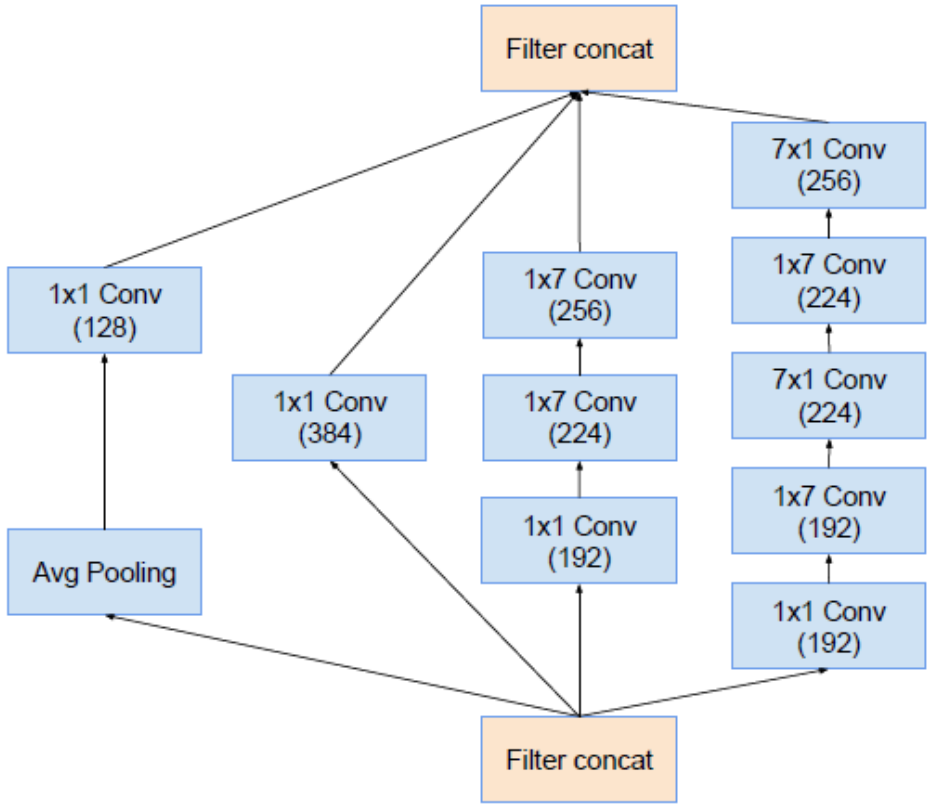
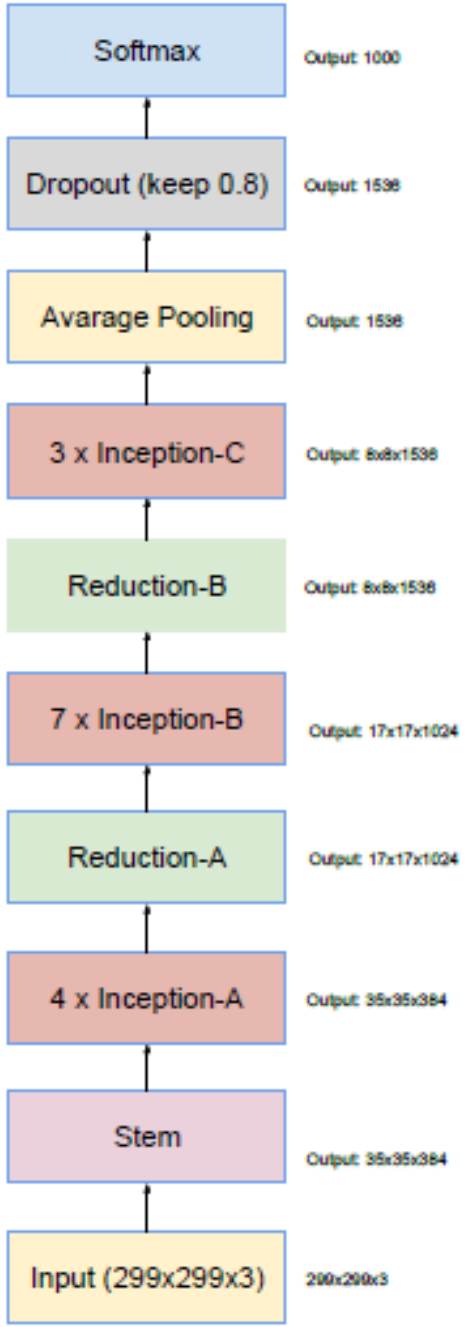


Figure 5. The schema for 17 × 17 grid modules of the pure Inception-v4 network. This is the Inception-B block of Figure 9.



Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning

Christian Szegedy
Google Inc.
1600 Amphitheatre Pkwy, Mountain View, CA
szegedy@google.com

Sergey Ioffe
sioffe@google.com

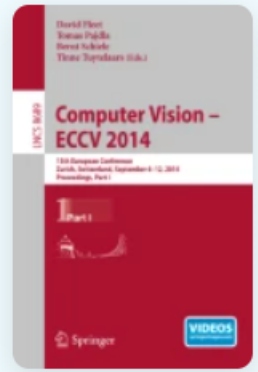
Vincent Vanhoucke
vanhoucke@google.com

Alex Alemi
alemi@google.com

Visualizing and Understanding Convolutional Networks

Conference paper

pp 818–833 | Cite this conference paper



Computer Vision – ECCV 2014

(ECCV 2014)

Matthew D. Zeiler & Rob Fergus

Sections | References

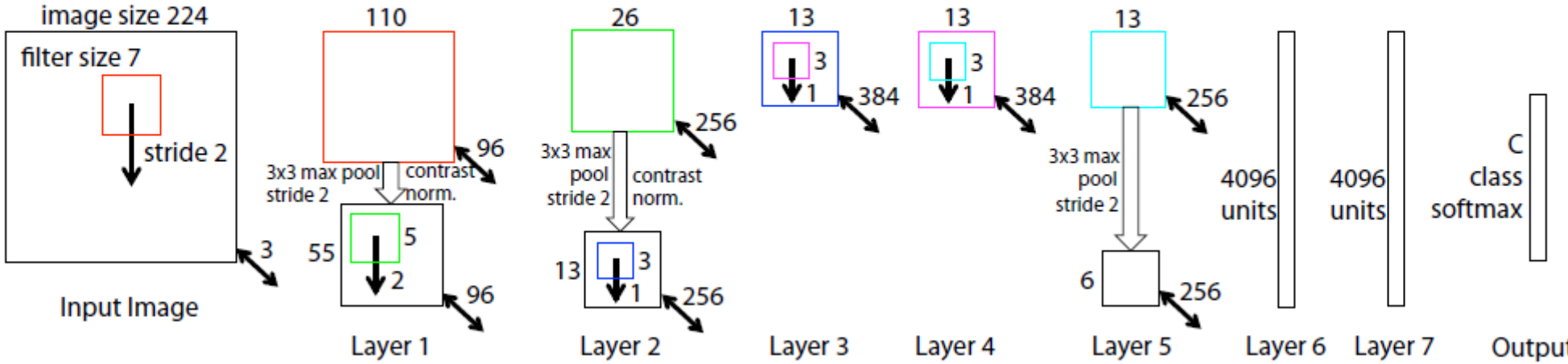


Fig. 3. Architecture of our 8 layer convnet model. A 224 by 224 crop of an image (with 3 color planes) is presented as the input. This is convolved with 96 different 1st layer filters (red), each of size 7 by 7, using a stride of 2 in both x and y. The resulting feature maps are then: (i) passed through a rectified linear function (not shown), (ii) pooled (max within 3x3 regions, using stride 2) and (iii) contrast normalized across feature maps to give 96 different 55 by 55 element feature maps. Similar operations are repeated in layers 2,3,4,5. The last two layers are fully connected, taking features from the top convolutional layer as input in vector form ($6 \cdot 6 \cdot 256 = 9216$ dimensions). The final layer is a C-way softmax function, C being the number of classes. All filters and feature maps are square in shape.

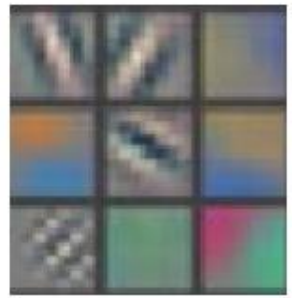
3 Training Details

We now describe the large convnet model that will be visualized in Section 4. The architecture, shown in Fig. 3, is similar to that used by Krizhevsky *et al.* [18] for ImageNet classification. One difference is that the sparse connections used in Krizhevsky's layers 3,4,5 (due to the model being split across 2 GPUs) are replaced with dense connections in our model. Other important differences relating to layers 1 and 2 were made following inspection of the visualizations in Fig. 5, as described in Section 4.1.

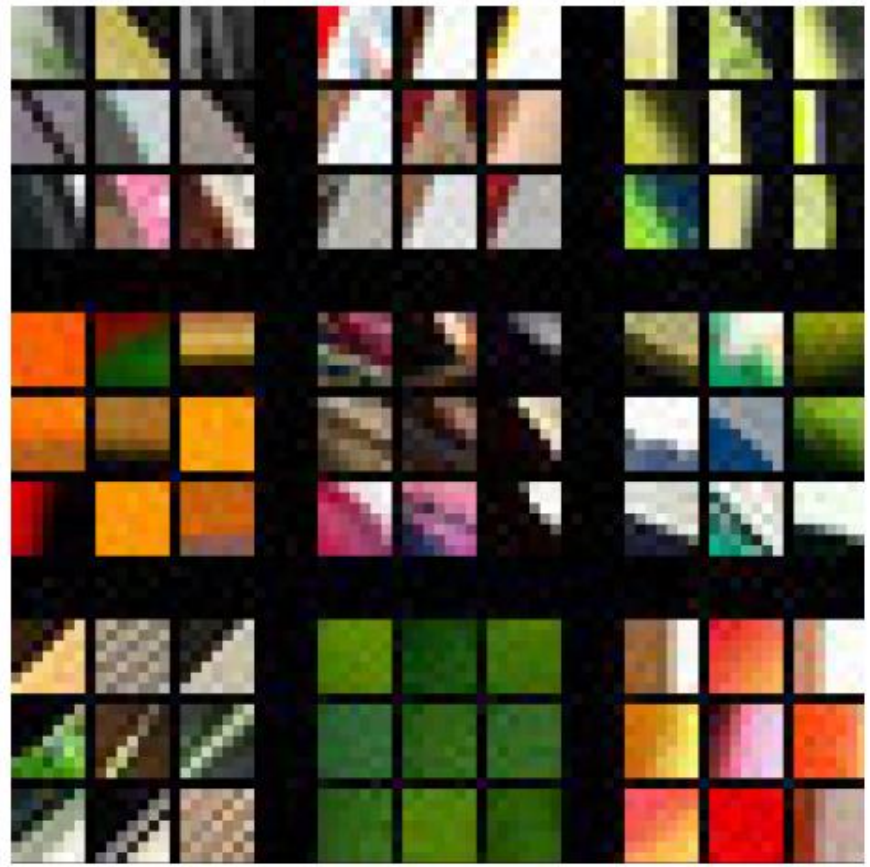
The model was trained on the ImageNet 2012 training set (1.3 million images, spread over 1000 different classes) [6]. Each RGB image was preprocessed by resizing the smallest dimension to 256, cropping the center 256x256 region, subtracting the per-pixel mean (across all images) and then using 10 different sub-crops of size 224x224 (corners + center with(out) horizontal flips). Stochastic gradient descent with a mini-batch size of 128 was used to update the parameters, starting with a learning rate of 10^{-2}, in conjunction with a momentum term of 0.9. We

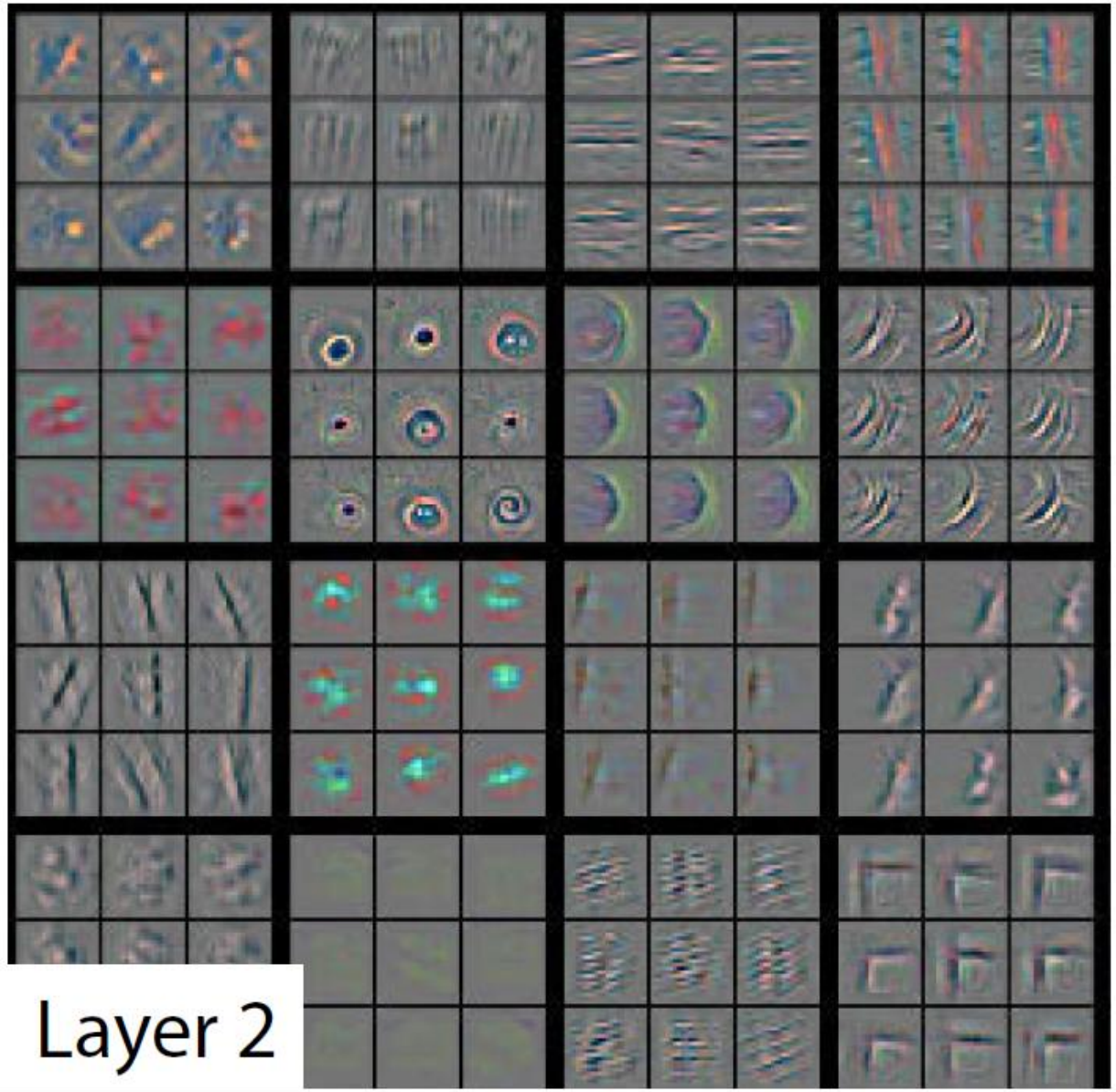
Feature Visualization: Fig. 2 shows feature visualizations from our model once training is complete. For a given feature map, we show the top 9 activations, each projected separately down to pixel space, revealing the different

structures that excite that map and showing its invariance to input deformations. Alongside these visualizations we show the corresponding image patches. These have greater variation than visualizations which solely focus on the discriminant structure within each patch. For example, in layer 5, row 1, col 2, the patches

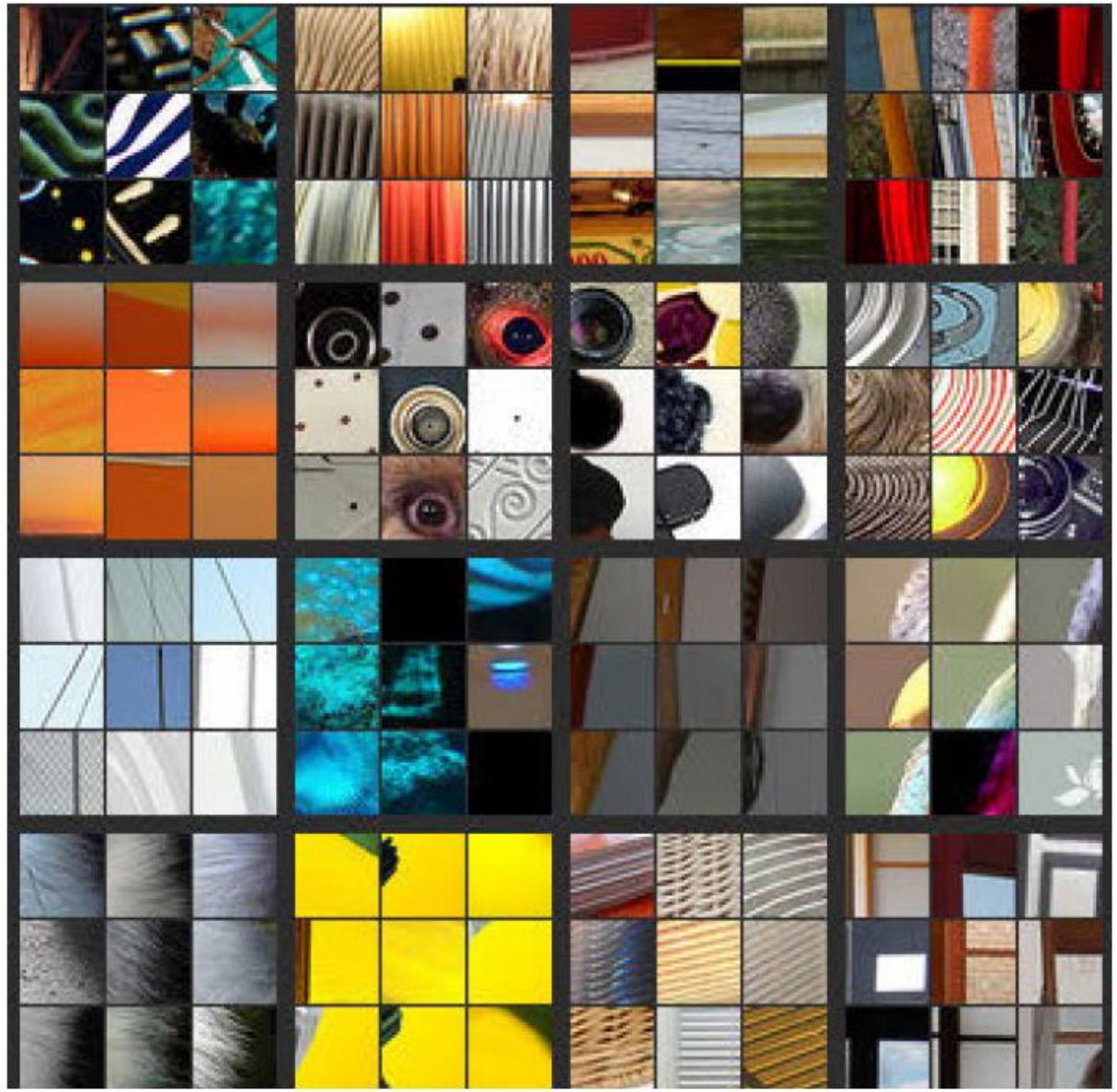


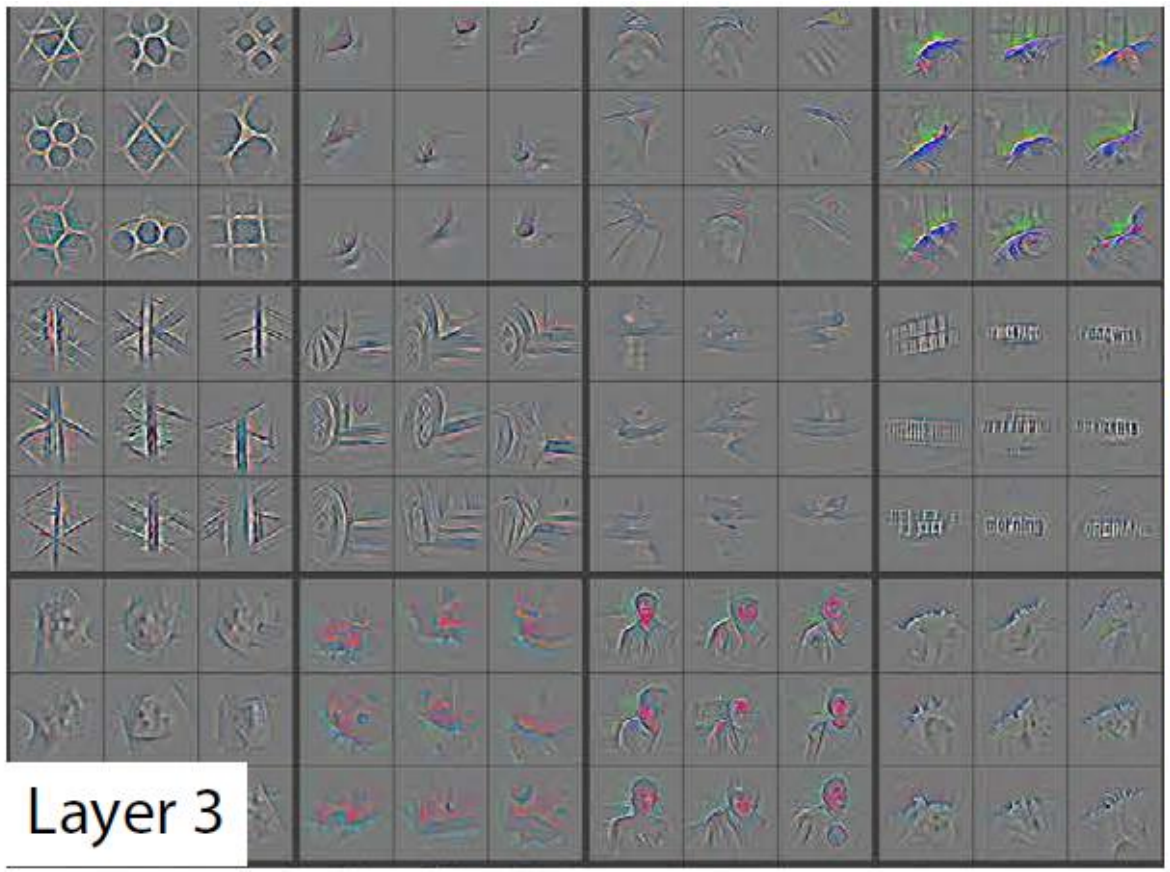
Layer 1

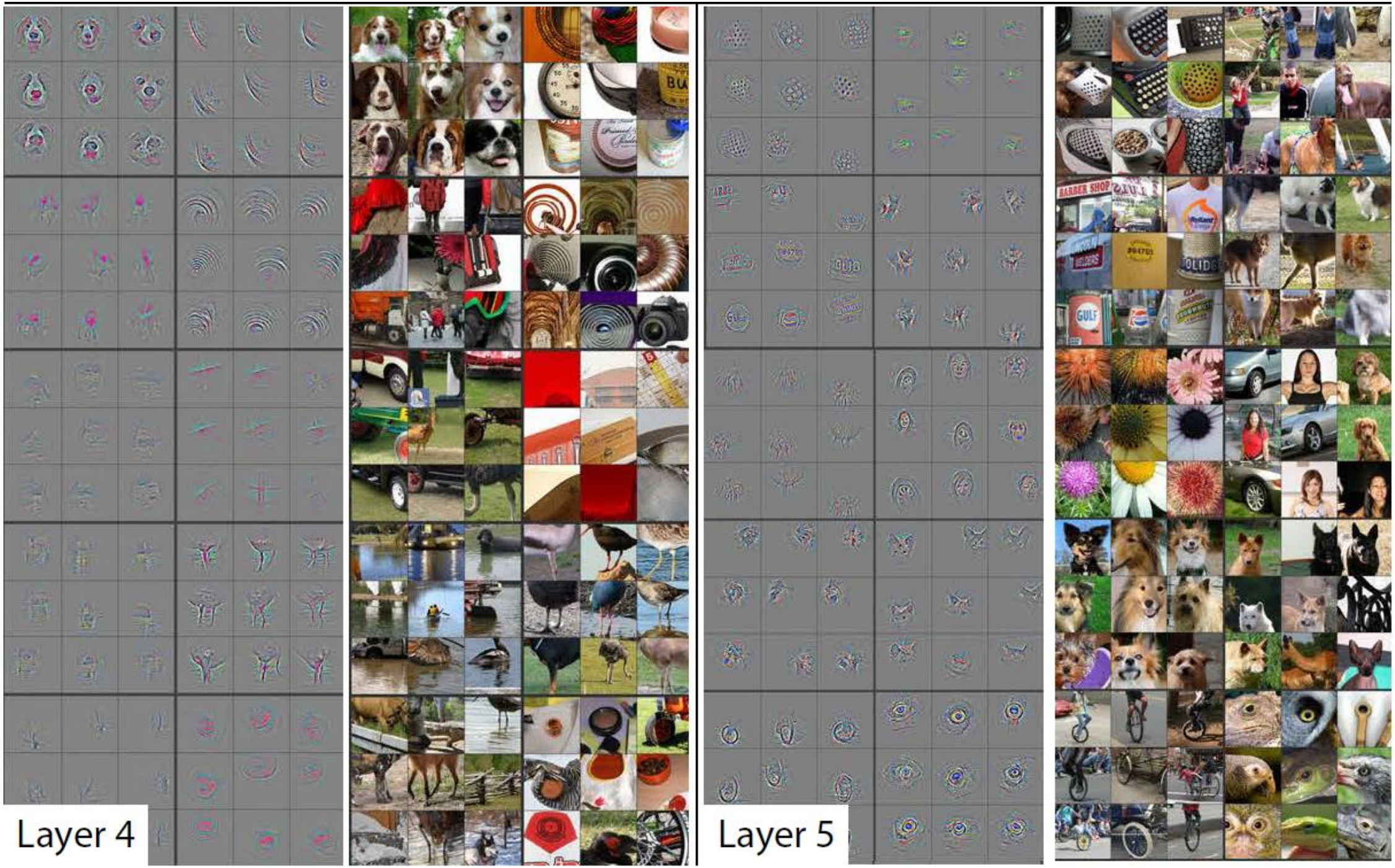




Layer 2




Layer 3


Layer 4
Layer 5

Simple loss functions and backpropagation

Regression loss functions

$\tilde{f}(x)$ is the output of the network. It needs to be piecewise smooth. So the building blocks of the network need to be piecewise smooth.

For example, the max-pooling operation $\max\left(f_1(x),\ldots,f_M(x)\right)$ of smooth functions $\{f_1,\ldots,f_M\}$, is piecewise smooth.

Mean Squared Error (MSE) $$\frac{1}{\#I_{\text{train}}}\sum_{i\in I_{\text{train}}}\left(\tilde{f}(x_i)-y_i\right)^2$$

Mean Averaged Error (MAE) $$\frac{1}{\#I_{\text{train}}}\sum_{i\in I_{\text{train}}}\left|\tilde{f}(x_i)-y_i\right|$$

Piecewise smooth

Backpropagation

Assume a model with one weight $f(x) = f(x, w)$. Assume loss is $L(w) = \sum_i \left(f(x_i, w) - y_i \right)^2$

Minimization is performed using gradient descent. Start with an initial (random) guess $w^{(0)}$. At each step k compute for a certain subset $\Lambda^{(k)}$ of the training dataset

$$\left. \frac{d}{dw} L \right|_{w^{(k)}} = 2 \sum_{i \in \Lambda^{(k)}} \left(f\left(x_i, w^{(k)}\right) - y_i \right) \frac{\partial f\left(x_i, w^{(k)}\right)}{\partial w}$$

$$= 2 \left\langle f\left(x_i, w^{(k)}\right) - y_i, \frac{\partial f\left(x_i, w^{(k)}\right)}{\partial w} \right\rangle_{\Lambda^{(k)}}$$

Backpropagation

Now suppose we are using a 2-layer model with 2 weights $f(x) = f_2(f_1(x, w_1), w_2)$, $w_1, w_2 \in \mathbb{R}$, $w = (w_1, w_2)$,

$$\frac{d}{dw_2} L\Big|_{w^{(k)}} = 2 \sum_{i \in \Lambda^{(k)}} \left(f_2\left(f_1\left(x_i, w_1^{(k)}\right), w_2^{(k)}\right) - y_i \right) \frac{\partial f_2\left(f_1\left(x_i, w_1^{(k)}\right), w_2^{(k)}\right)}{\partial w_2}$$

$$\underset{\text{tensor form}}{=} 2 \left\langle f_2\left(f_1\left(x_i, w_1^{(k)}\right), w_2^{(k)}\right) - y_i, \frac{\partial f_2\left(f_1\left(x_i, w_1^{(k)}\right), w_2^{(k)}\right)}{\partial w_2} \right\rangle_{\Lambda^{(k)}}$$

$$\frac{d}{dw_1} L\Big|_{w^{(k)}} = 2 \sum_{i \in \Lambda^{(k)}} \left(f_2\left(f_1\left(x_i, w_1^{(k)}\right), w_2^{(k)}\right) - y_i \right) \frac{\partial f_2\left(f_1\left(x_i, w_1^{(k)}\right), w_2^{(k)}\right)}{\partial f_1} \frac{\partial f_1\left(x_i, w_1^{(k)}\right)}{\partial w_1}$$

So, the first layer passes to the second layer the information $\left\{ f_1\left(x_i, w_1^{(k)}\right) \right\}_{\Lambda^{(k)}}, \left\{ \frac{\partial f_1\left(x_i, w_1^{(k)}\right)}{\partial w_1} \right\}_{\Lambda^{(k)}}$

Journal of Machine Learning Research 18 (2018) 1-43 Submitted 8/17; Published 4/18

Automatic Differentiation in Machine Learning: a Survey

Atılım Güneş Baydin GUNES@ROBOTS.OX.AC.UK
Department of Engineering Science
University of Oxford
Oxford OX1 3PJ, United Kingdom

Barak A. Pearlmutter BARAK@PEARLMUTTER.NET
Department of Computer Science
National University of Ireland Maynooth
Maynooth, Co. Kildare, Ireland

Alexey Andreyevich Radul AXCH@MIT.EDU
Department of Brain and Cognitive Sciences
Massachusetts Institute of Technology
Cambridge, MA 02139, United States

Jeffrey Mark Siskind QOBI@PURDUE.EDU
School of Electrical and Computer Engineering
Purdue University
West Lafayette, IN 47907, United States

Under the hood of automatic differentiation

```
tuple<float,float> evaluateAndDerive(Expression Z, Variable V) {
   if isVariable(Z)
      if (Z = V) return {valueOf(Z), 1};
      else return {valueOf(Z), 0};
   else if (Z = A + B)
      {a, a'} = evaluateAndDerive(A, V);
      {b, b'} = evaluateAndDerive(B, V);
      return {a + b, a' + b'};
   else if (Z = A - B)
      {a, a'} = evaluateAndDerive(A, V);
      {b, b'} = evaluateAndDerive(B, V);
      return {a - b, a' - b'};
   else if (Z = A * B)
      {a, a'} = evaluateAndDerive(A, V);
      {b, b'} = evaluateAndDerive(B, V);
      return {a * b, b * a' + a * b'};
}
```

$$
\begin{aligned}
\langle u,u'\rangle + \langle v,v'\rangle &= \langle u+v, u'+v'\rangle \\
\langle u,u'\rangle - \langle v,v'\rangle &= \langle u-v, u'-v'\rangle \\
\langle u,u'\rangle * \langle v,v'\rangle &= \langle uv, u'v+uv'\rangle \\
\langle u,u'\rangle \,/\, \langle v,v'\rangle &= \left\langle \frac{u}{v}, \frac{u'v-uv'}{v^2} \right\rangle \quad (v \neq 0) \\
\sin\langle u,u'\rangle &= \langle \sin(u), u'\cos(u)\rangle \\
\cos\langle u,u'\rangle &= \langle \cos(u), -u'\sin(u)\rangle \\
\exp\langle u,u'\rangle &= \langle \exp u, u'\exp u\rangle \\
\log\langle u,u'\rangle &= \langle \log(u), u'/u\rangle \quad (u>0) \\
\langle u,u'\rangle^k &= \langle u^k, u'ku^{k-1}\rangle \quad (u \neq 0) \\
|\langle u,u'\rangle| &= \langle |u|, u' \operatorname{sign} u\rangle \quad (u \neq 0)
\end{aligned}
$$

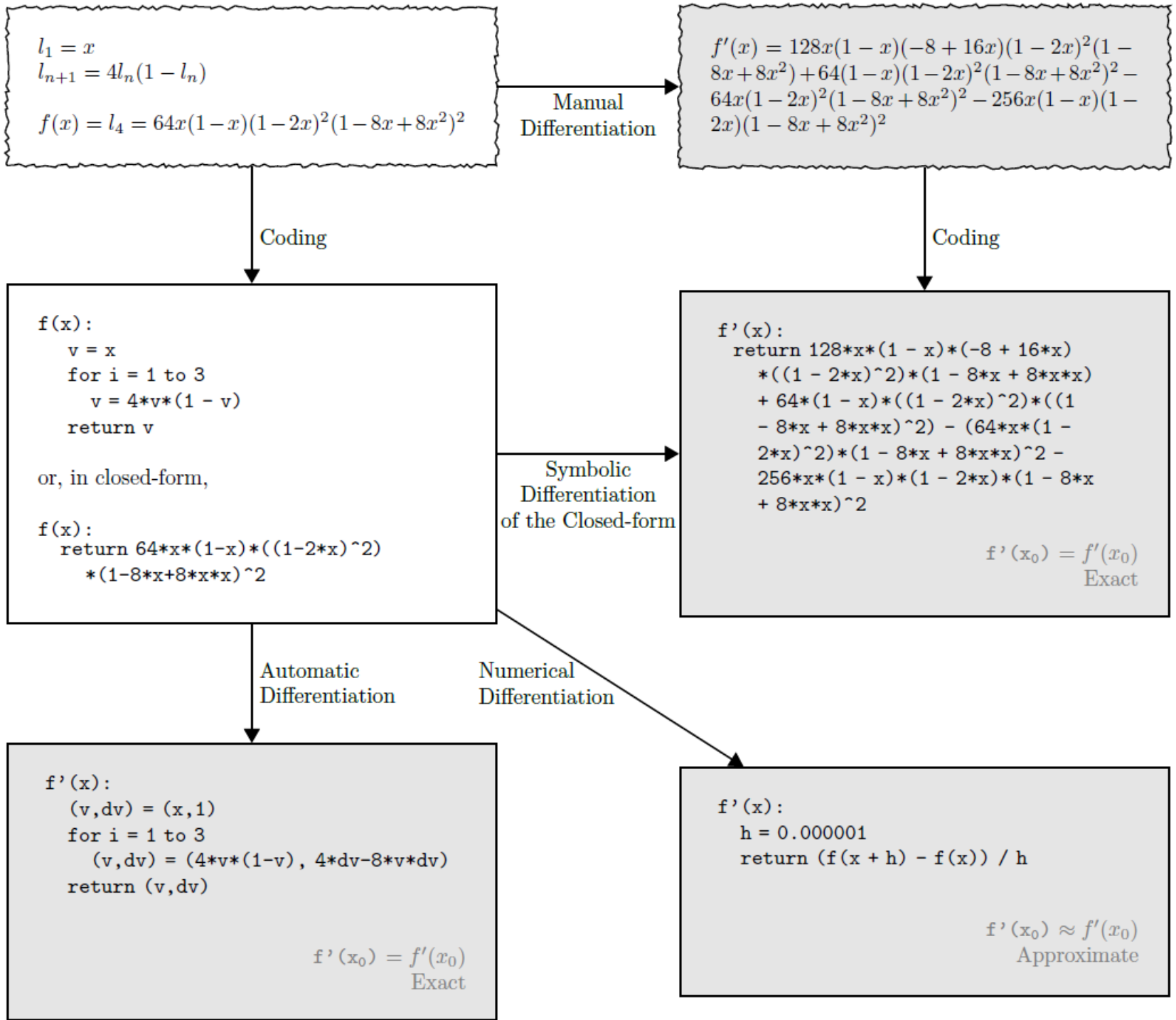

$l_1 = x$
$l_{n+1} = 4l_n(1 - l_n)$
$f(x) = l_4 = 64x(1-x)(1-2x)^2(1-8x+8x^2)^2$
Manual Differentiation
$f'(x) = 128x(1-x)(-8+16x)(1-2x)^2(1-8x+8x^2)+64(1-x)(1-2x)^2(1-8x+8x^2)^2-64x(1-2x)^2(1-8x+8x^2)^2-256x(1-x)(1-2x)(1-8x+8x^2)^2$
Coding
Coding
f(x):
v = x
for i = 1 to 3
v = 4*v*(1 - v)
return v
or, in closed-form,
f(x):
return 64*x*(1-x)*((1-2*x)^2)
*(1-8*x+8*x*x)^2
Symbolic Differentiation of the Closed-form
f'(x):
return 128*x*(1 - x)*(-8 + 16*x)
*((1 - 2*x)^2)*(1 - 8*x + 8*x*x)
+ 64*(1 - x)*((1 - 2*x)^2)*((1
- 8*x + 8*x*x)^2) - (64*x*(1 -
2*x)^2)*(1 - 8*x + 8*x*x)^2 -
256*x*(1 - x)*(1 - 2*x)*(1 - 8*x
+ 8*x*x)^2
f'(x0) = f'(x0)
Exact
Automatic Differentiation
Numerical Differentiation
f'(x):
(v,dv) = (x,1)
for i = 1 to 3
(v,dv) = (4*v*(1-v), 4*dv-8*v*dv)
return (v,dv)
f'(x0) = f'(x0)
Exact
f'(x):
h = 0.000001
return (f(x + h) - f(x)) / h
f'(x0) ≈ f'(x0)
Approximate

Table 2: Forward mode AD example, with $y = f(x_1, x_2) = \ln(x_1) + x_1 x_2 - \sin(x_2)$ evaluated at $(x_1, x_2) = (2, 5)$ and setting $\dot{x}_1 = 1$ to compute $\frac{\partial y}{\partial x_1}$. The original forward evaluation of the primals on the left is augmented by the tangent operations on the right, where each line complements the original directly to its left.

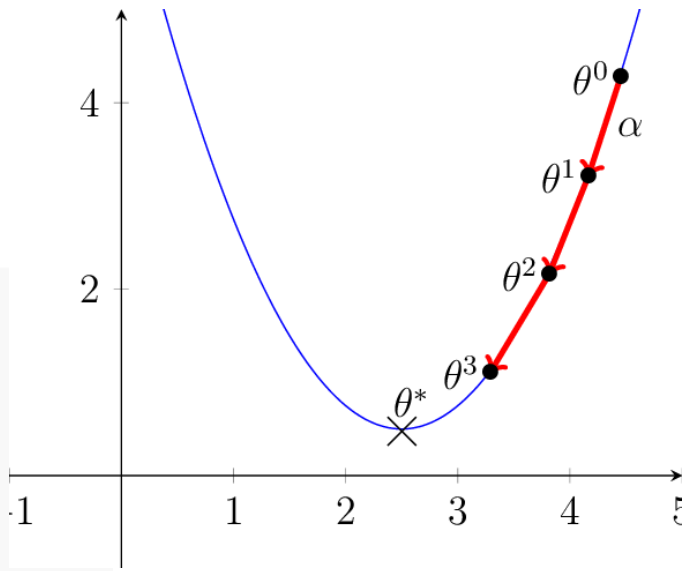
Forward Primal Trace			Forward Tangent (Derivative) Trace		
v_{-1}	$= x_1$	$= 2$	$\dot{v}_{-1}$	$= \dot{x}_1$	$= 1$
v_0	$= x_2$	$= 5$	$\dot{v}_0$	$= \dot{x}_2$	$= 0$
v_1	$= \ln v_{-1}$	$= \ln 2$	$\dot{v}_1$	$= \dot{v}_{-1}/v_{-1}$	$= 1/2$
v_2	$= v_{-1} \times v_0$	$= 2 \times 5$	$\dot{v}_2$	$= \dot{v}_{-1} \times v_0 + \dot{v}_0 \times v_{-1}$	$= 1 \times 5 + 0 \times 2$
v_3	$= \sin v_0$	$= \sin 5$	$\dot{v}_3$	$= \dot{v}_0 \times \cos v_0$	$= 0 \times \cos 5$
v_4	$= v_1 + v_2$	$= 0.693 + 10$	$\dot{v}_4$	$= \dot{v}_1 + \dot{v}_2$	$= 0.5 + 5$
v_5	$= v_4 - v_3$	$= 10.693 + 0.959$	$\dot{v}_5$	$= \dot{v}_4 - \dot{v}_3$	$= 5.5 - 0$
y	$= v_5$	$= 11.652$	$\dot{y}$	$= \dot{v}_5$	$= \mathbf{5.5}$

An overview of gradient descent optimization algorithms*

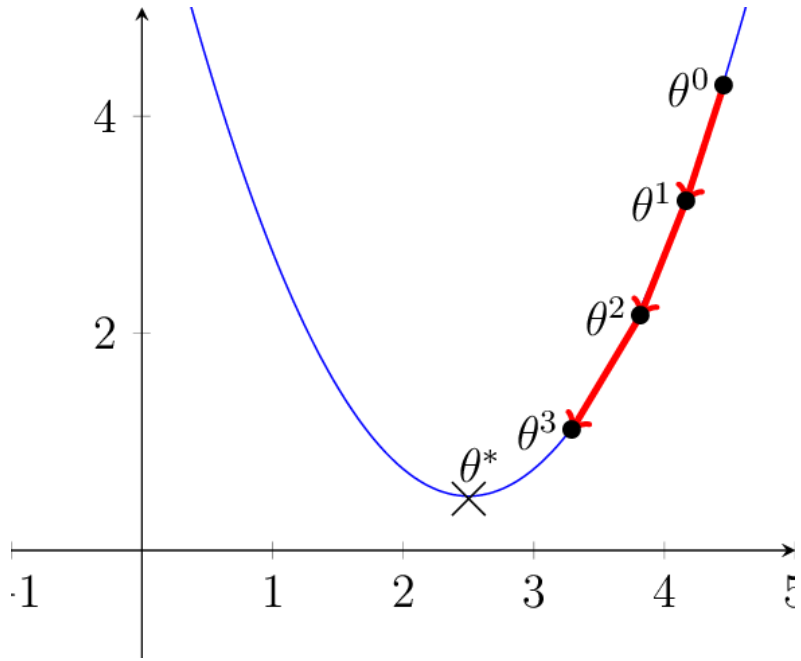
Sebastian Ruder
Insight Centre for Data Analytics, NUI Galway
Aylien Ltd., Dublin
ruder.sebastian@gmail.com

```
import torch.optim as optim

criterion = nn.CrossEntropyLoss()
optimizer = optim.SGD(net.parameters(), lr=0.001, momentum=0.9)
```



Gradient descent is a way to minimize an objective function $J(\theta)$ parameterized by a model's parameters $\theta \in \mathbb{R}^d$ by updating the parameters in the opposite direction of the gradient of the objective function $\nabla_\theta J(\theta)$ w.r.t. to the parameters. The learning rate η determines the size of the steps we take to reach a (local) minimum. In other words, we follow the direction of the slope of the surface created by the objective function downhill until we reach a valley.[5]



Vanilla gradient descent, ~~aka batch gradient descent~~, computes the gradient of the cost function w.r.t. to the parameters θ for the entire training dataset:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta) \tag{1}$$

As we need to calculate the gradients for the whole dataset to perform just *one* update, ~~batch~~ gradient descent can be very slow and is intractable for datasets that do not fit in memory. ~~Batch~~ gradient descent also does not allow us to update our model *online*, i.e. with new examples on-the-fly.

In code, batch gradient descent looks something like this:

```
for i in range(nb_epochs):
  params_grad = evaluate_gradient(loss_function, data, params)
  params = params - learning_rate * params_grad
```

Mini-batch gradient descent finally takes the best of both worlds and performs an update for every mini-batch of n training examples:

$$\theta = \theta - \eta \cdot \nabla_\theta J(\theta; x^{(i:i+n)}; y^{(i:i+n)}) \tag{3}$$

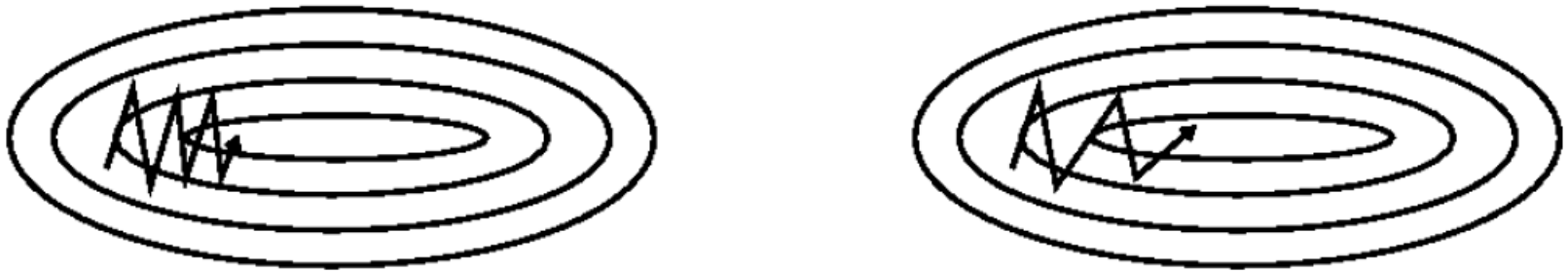
This way, it a) reduces the variance of the parameter updates, which can lead to more stable convergence; and b) can make use of highly optimized matrix optimizations common to state-of-the-art deep learning libraries that make computing the gradient w.r.t. a mini-batch very efficient. Common mini-batch sizes range between 50 and 256, but can vary for different applications. Mini-batch gradient descent is typically the algorithm of choice when training a neural network and the term SGD usually is employed also when mini-batches are used. Note: In modifications of SGD in the rest of this post, we leave out the parameters $x^{(i:i+n)}; y^{(i:i+n)}$ for simplicity.

In code, instead of iterating over examples, we now iterate over mini-batches of size 50:

```
for i in range(nb_epochs):
  np.random.shuffle(data)
  for batch in get_batches(data, batch_size=50):
    params_grad = evaluate_gradient(loss_function, batch, params)
    params = params - learning_rate * params_grad
```

4.1 Momentum

SGD has trouble navigating ravines, i.e. areas where the surface curves much more steeply in one dimension than in another [20], which are common around local optima. In these scenarios, SGD oscillates across the slopes of the ravine while only making hesitant progress along the bottom towards the local optimum as in Figure 2a.



(a) SGD without momentum

(b) SGD with momentum

Figure 2: Source: Genevieve B. Orr

Momentum [17] is a method that helps accelerate SGD in the relevant direction and dampens oscillations as can be seen in Figure 2b. It does this by adding a fraction γ of the update vector of the past time step to the current update vector[8]

$$\begin{aligned} v_t &= \gamma v_{t-1} + \eta \nabla_\theta J(\theta) \\ \theta &= \theta - v_t \end{aligned} \tag{4}$$

The momentum term γ is usually set to 0.9 or a similar value.

Momentum - exponential averaging of past gradients

$$
\begin{aligned}
v_t &= \gamma v_{t-1} + \eta \nabla_\theta J\left(\theta^{(t)}\right) \\
&= \gamma\left(\gamma v_{t-2} + \eta \nabla_\theta J\left(\theta^{(t-1)}\right)\right) + \eta \nabla_\theta J\left(\theta^{(t)}\right) \\
&= \gamma^2 v_{t-2} + \gamma \eta \nabla_\theta J\left(\theta^{(t-1)}\right) + \eta \nabla_\theta J\left(\theta^{(t)}\right) \\
&\cdots \\
&= \gamma^n v_{t-n} + \eta\left(\gamma^{n-1} \nabla_\theta J\left(\theta^{(t-n+1)}\right) + \cdots + \gamma \nabla_\theta J\left(\theta^{(t-1)}\right) + \nabla_\theta J\left(\theta^{(t)}\right)\right)
\end{aligned}
$$

4.3 Adagrad

Adagrad [8] is an algorithm for gradient-based optimization that does just this: It adapts the learning rate to the parameters, performing larger updates for infrequent and smaller updates for frequent parameters. For this reason, it is well-suited for dealing with sparse data. Dean et al. [6] have found that Adagrad greatly improved the robustness of SGD and used it for training large-scale neural nets at Google, which – among other things – learned to recognize cats in Youtube videos[10]. Moreover, Pennington et al. [16] used Adagrad to train GloVe word embeddings, as infrequent words require much larger updates than frequent ones.

Previously, we performed an update for all parameters θ at once as every parameter θ_i used the same learning rate η. As Adagrad uses a different learning rate for every parameter θ_i at every time step t, we first show Adagrad's per-parameter update, which we then vectorize. For brevity, we set $g_{t,i}$ to be the gradient of the objective function w.r.t. to the parameter θ_i at time step t:

$$g_{t,i} = \nabla_{\theta_t} J(\theta_{t,i}) \tag{6}$$

The SGD update for every parameter θ_i at each time step t then becomes:

$$\theta_{t+1,i} = \theta_{t,i} - \eta \cdot g_{t,i} \tag{7}$$

In its update rule, Adagrad modifies the general learning rate η at each time step t for every parameter θ_i based on the past gradients that have been computed for θ_i:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} + \epsilon}} \cdot g_{t,i} \tag{8}$$

$G_t \in \mathbb{R}^{d \times d}$ here is a diagonal matrix where each diagonal element i, i is the sum of the squares of the gradients w.r.t. θ_i up to time step t[11], while ϵ is a smoothing term that avoids division by zero (usually on the order of $1e-8$). Interestingly, without the square root operation, the algorithm performs much worse.

As G_t contains the sum of the squares of the past gradients w.r.t. to all parameters θ along its diagonal, we can now vectorize our implementation by performing an element-wise matrix-vector multiplication $\odot$ between G_t and g_t:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{G_t + \epsilon}} \odot g_t. \tag{9}$$

One of Adagrad's main benefits is that it eliminates the need to manually tune the learning rate. Most implementations use a default value of 0.01 and leave it at that.

Adagrad's main weakness is its accumulation of the squared gradients in the denominator: Since every added term is positive, the accumulated sum keeps growing during training. This in turn causes the learning rate to shrink and eventually become infinitesimally small, at which point the algorithm is no longer able to acquire additional knowledge. The following algorithms aim to resolve this flaw.

4.6 Adam

Adaptive Moment Estimation (Adam) [10] is another method that computes adaptive learning rates for each parameter. In addition to storing an exponentially decaying average of past squared gradients v_t like Adadelta and RMSprop, Adam also keeps an exponentially decaying average of past gradients m_t, similar to momentum:

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1 - \beta_1) g_t \\ v_t &= \beta_2 v_{t-1} + (1 - \beta_2) g_t^2 \end{aligned} \tag{19}$$

m_t and v_t are estimates of the first moment (the mean) and the second moment (the uncentered variance) of the gradients respectively, hence the name of the method. As m_t and v_t are initialized as vectors of 0's, the authors of Adam observe that they are biased towards zero, especially during the initial time steps, and especially when the decay rates are small (i.e. β_1 and β_2 are close to 1).

They counteract these biases by computing bias-corrected first and second moment estimates:

$$\begin{aligned} \hat{m}_t &= \frac{m_t}{1 - \beta_1^t} \\ \hat{v}_t &= \frac{v_t}{1 - \beta_2^t} \end{aligned} \tag{20}$$

They then use these to update the parameters just as we have seen in Adadelta and RMSprop, which yields the Adam update rule:

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon}\hat{m}_t \tag{21}$$

The authors propose default values of 0.9 for β_1, 0.999 for β_2, and 10^{-8} for ϵ. They show empirically that Adam works well in practice and compares favorably to other adaptive learning-method algorithms.