technische universität dortmund

fakultät für informatik

Proseminar „Convolutional Neural Networks - Methoden und Anwendungen“

Bilderkennung mit Vgg

Justin Schartner

18. Mai 2022

Struktur

1 Einleitung

2 Allgemeines

3 Architektur

4 Training

5 Beispiel

6 Bewertung

7 Ausblick

Thema

Problemstellung

Gibt es CNN-Architekturen, welche Bilder noch besser, als schon bekannte Architekturen klassifizieren können?

Lösung

Durch die Erhöhung der Tiefe eines CNN verspricht man sich genauere Aussagen über Bilder machen zu können.

Ergebnis

Die VGG-Architektur hat bewiesen, dass die Tiefe eines CNN, eine ausschlaggebende Komponente hinsichtlich der Bilder-Klassifizierung ist.

Gliederung

- Was ist VGG? **Allgemeines**
- Wie funktioniert VGG, was macht es besonders? **Architektur**
- Wie trainiert man ein VGG-net, was ist wichtig? **Training**
- Wie implementiert man ein VGG-net? **Beispiel**
- Wie gut ist VGG? **Bewertung**

Struktur

1 Einleitung

2 Allgemeines

3 Architektur

4 Training

5 Beispiel

6 Bewertung

7 Ausblick

Eckdaten

- **V**isual **G**eometry **G**roup
- Department of Engineering Sciecne, University of Oxford
- Karen Simonyan und Andrew Zisserman
- **Veröffentlicht**: 4 Sep 2014
- **Letzte Änderung**: 10 Apr 2015

Idee

Steigerung der Genauigkeit durch:

- Steigerung der Tiefe, des CNNs
- Schachtelung von Convolutional-Layer-Blöcken
- Einsatz von kleinen 3x3-Filtern und einer Stride von 1

Aufgaben/ Einsatz

Bilderkennung

- Imagenet
- Pneumoina Image
- Deep Facial Emotion Recognition
- Plankton Classification
- Plant Image Classification
- ...

Struktur

1 Einleitung

2 Allgemeines

3 Architektur

4 Training

5 Beispiel

6 Bewertung

7 Ausblick

Architekturarten

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 x 224 RGB-image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Architekur

Convolutional Layers

- receiptive field: 3x3, 1x1
- activation: RelU, stride: 1, padding: 1, channels: 64, 128, 512, 512

Pooling Layer

- Max-Pool
- field: 2x2, stride: 2

Fully-Connected Layers

- activation: RelU
- channels: 4096, 4096, 1000

Softmax Layer

Convolutional Layers

- 2D - Convolution
- Aktivierungsfunktion: ReLU
- Stride: 1x1
- Padding: 1, 0

  $\Rightarrow$ Die Breite und Höhe des Inputs wird beibehalten
- Kernel: 3x3 oder 1x1

  $\Rightarrow$ Minimaler Kernel für den Vergleich von Links/Rechts Oben/Unten
- Filter Anzahl: 64, 128, 256, 512

  $\Rightarrow$ Filter lernen Muster des Inputs zu erkennen

Convolutional Layers

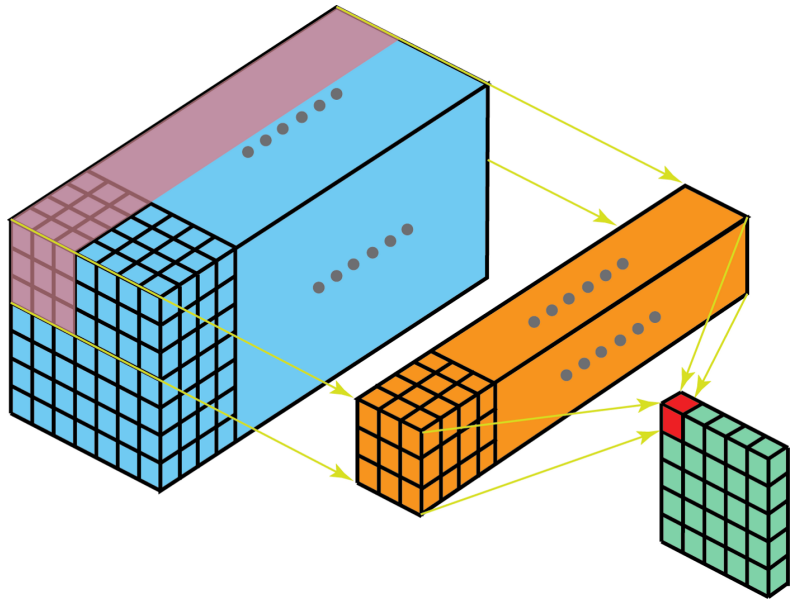

Abbildung: Quelle ?

$$\text{(Breite x Höhe x Tiefe)} \xrightarrow[\text{Conv2d(Filter: } \mathbf{3x3xTiefe}\text{, Filter Anzahl: } \mathbf{n})]{} \text{(Breite x Höhe x n)}$$

ReLU

Rectified Linerar Unit

$$f(x) = \begin{cases} x & \text{x>0} \\ 0 & \text{sonst} \end{cases}$$

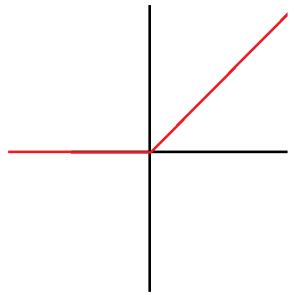

Abbildung: $f(x)$

$$f'(x) = \begin{cases} 1 & \text{x>0} \\ 0 & \text{x<0} \end{cases}$$

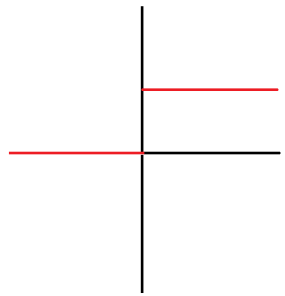

Abbildung: $f'(x)$

Max-Pooling

- Stride: 2x2
- Kernel: 2x2

  $\Rightarrow$ Die Breite und Höhe wird halbiert

  $\Rightarrow$ Daten werden auf die auschlaggebenden Informationen reduziert

Fully Connected Layers

- Input: (7x7x512)

  $\Rightarrow$ (7x7x512) Input-Neuronen

- Aktivierungsfunktion: ReLU

- Zwei versteckte Layer mit jeweils 4096 Neuronen

- ImageNet-Klassifiezunrg von 1000 Klassen

  $\Rightarrow$ 1000 Output-Neuronen

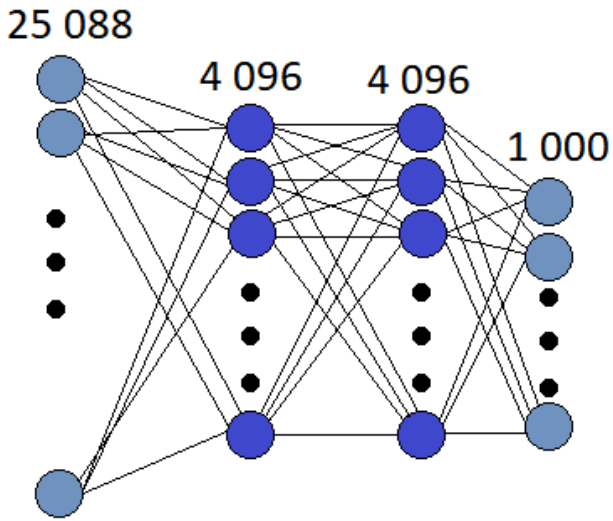

Abbildung: full-yconnected layer

Softmax

$$\sigma(\vec{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}}$$

- normalisierte Exponentialfunktion
- kategoriale Verteilung
- Transformation in den Wertebereich [0,1]

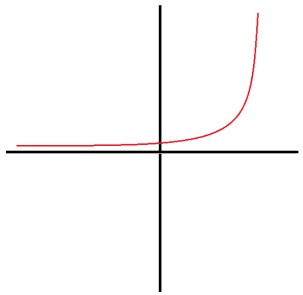

Abbildung: e^x

$$\begin{pmatrix} -0.5 \\ 0.8 \\ 1.3 \end{pmatrix} \xrightarrow{Softmax} \begin{pmatrix} 0.093 \\ 0.342 \\ 0.564 \end{pmatrix}$$

Struktur

1 Einleitung

2 Allgemeines

3 Architektur

4 Training

5 Beispiel

6 Bewertung

7 Ausblick

Training

Optimierung des Traininigs durch:

- **S**tochastic **G**radient **D**escent
- Dropout, p=0.5
- L2-Normalisation
- Momentum
- Batch-Size: 256

Weitere nützliche Faktoren:

- Kleinere Filter-Grösen
- Vor-Initialisierung von Gewichten
- Die Tiefe des Netzwerkes

Training
Details

- Learninig-rate: 10e-2 - 10e-4
- Momentum: 0.9
- Weight-Decay: 5e-4
- Biases wurden mit 0 initialisiert
- VGG16 wurde mit VGG11-Gewichten initialisiert

Training
Bild-Processing

- Bilder wurden zufällig aus anderen Bildern ausgeschnitten
- Bilder wurden zufällig horizontal gedreht
- Bilder wurden zufällig skaliert

$\Rightarrow$ Filter werden trainiert Features auf verschieden Arten zu erkennen

Struktur

1 Einleitung

2 Allgemeines

3 Architektur

4 Training

5 Beispiel

6 Bewertung

7 Ausblick

Vgg initialisieren

1. Variante: Die Komponenten einzeln erstellen und aneinander reihen.
   - Anpassbarkeit an das Problem
   - Das Trainieren des Netzwerkes nimmt mehr Zeit in Anspruch
2. Variante: Benutzen von schon bestehenden (und trainierten) Netzwerken.
   - Sind sofort einsatzbereit
   - Aufwand ist kleiner
   - Die vortrainierten Gewichte können die Trainingszeit minimieren
   - Konfigurierung kann aufwendig sein

Vgg initialisieren

1. Variante

```
30
31      def create_fully_connected_layers(self, num_classes):
32          fully_connected_layers = nn.Sequential(
33              nn.Linear(512*7*7, 4096),
34              nn.ReLU(),
35              nn.Dropout(p=0.5),
36              nn.Linear(4096, 4096),
37              nn.ReLU(),
38              nn.Dropout(p=0.5),
39              nn.Linear(4096, num_classes),
40              nn.Softmax(dim=0)
41          )
42
43          return fully_connected_layers
44
```

```
def create_conv_layers(self, in_channels, architecture):
    layers = []

    #[[64, 64], [128, 128], [256, 256, 256], [512, 512, 512], [512, 512, 512]]
    for block in architecture:
        #[64, 64]
        for layer in block:
            out_channels = layer

            #conv-layer + relu
            layers += [
                nn.Conv2d(in_channels=in_channels, out_channels=out_channels,
                          kernel_size=(3,3), stride=(1,1), padding=(1,1)),
                nn.ReLU()
            ]

            in_channels = out_channels

        #max-pooling
        layers += [nn.MaxPool2d(kernel_size=(2,2), stride=(2,2))]
    return nn.Sequential(*layers)
```

Vgg initialisieren

1. Variante

```python
import torch.nn as nn

vgg16 = [[64, 64], [128, 128], [256, 256, 256], [512, 512, 512], [512, 512, 512]]
vgg19 = [[64, 64], [128, 128], [256, 256, 256, 256], [512, 512, 512, 512], [512, 512, 512, 512]]

class VGG_net(nn.Module):
    def __init__(self, in_channels=3, num_classes=1000, architecture=vgg16):
        super(VGG_net, self).__init__()

        #Convolutional Layers
        self.conv_layers = self.create_conv_layers(in_channels, architecture)

        #Fully Connected Layers
        self.fully_connected_layers = self.create_fully_connected_layers(num_classes)

    def forward(self, x):
        # x = [2, 3, 224, 224]

        x = self.conv_layers(x)
        # x = [2, 512, 7, 7]

        x = x.reshape(x.shape[0], -1)
        # x = [[1 .. (512x7x7)], [1 .. (512x7x7)]]

        x = self.fully_connected_layers(x)
        # x = [[1 .. 1000], [1 .. 1000]]

        return x
```

Vgg initialisieren

1. Variante

```
VGG_net(
  (conv_layers): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU()
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU()
    (4): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU()
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU()
    (9): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU()
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU()
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU()
    (16): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU()
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU()
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU()
    (23): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU()
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU()
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU()
    (30): MaxPool2d(kernel_size=(2, 2), stride=(2, 2), padding=0, dilation=1, ceil_mode=False)
  )
  (fully_connected_layers): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU()
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU()
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
    (7): Softmax(dim=0)
  )
)
```

```python

if __name__ == "__main__":
    net = VGG_net(in_channels=3, num_classes=1000, architecture=vgg16)
    print(net)

```

Vgg initialisieren

2. Variante

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=4096, out_features=1000, bias=True)
  )
)
```

```python

import torch

if __name__ == "__main__":
    net = torch.hub.load('pytorch/vision:v0.10.0', 'vgg16', pretrained=True)
    print(net)


```

Vgg trainieren

```python
def train_network(net, x_train, y_train, epochs):

    #Training-Parameter
    learning_rate = 0.001
    momentum = 0.9

    #Loss function
    criterion = nn.CrossEntropyLoss()
    #Optimizer
    optimizer = optim.SGD(net.parameters(),
                          lr=learning_rate, momentum=momentum)

    for epoch in range(epochs):

        #reset the gradients in net
        optimizer.zero_grad()

        #forward
        outputs = net(x_train)

        #calculate loss
        loss = criterion(outputs, y_train)
        #calculate gradients in net
        loss.backward()

        #backward
        optimizer.step()
```

Vgg benutzen

```
pug 0.9928255081176758
```

```python

#get network
net = net = torch.hub.load('pytorch/vision:v0.10.0', 'vgg16', pretrained=True)

#get image
image = Image.open('dog1.jpg')

#get categories
categories = loadCategories('imagenet_classes.txt')



#preprocess image
input_image = pre_process(image)

#forward
with torch.no_grad():
    output = net(input_image)


#apply softmax
probs = torch.nn.functional.softmax(output[0], dim=0)

#result
guess, guess_prob = findHighestProb(probs, categories)
print(guess, guess_prob)

```

Struktur

1 Einleitung

2 Allgemeines

3 Architektur

4 Training

5 Beispiel

6 Bewertung

7 Ausblick

Ergebnis

- Die Anwendung von mehreren 3x3 Filtern ersetzt die FunktionalitÃ¤t von bsp. 7x7 Filtern und erhÃ¶rt die DiskriminietivitÃ¤t
- Die Tiefe eine CNNs ist auschlaggebend fÃ¼r die Genauigkeit
- VGG-Architekturen haben viele Anwendungsbereiche, erzielen auf verschiedensten Datenbanken erfolgreiche Resultate

Komplexität

- Anzahl an Parametern: 15.1, 15.3, 20.6, 25.9 in Millionen

  Mehr Parameter fÃ¼hren zu einer lÃ¤ngeren Trainigszeit
- Enspricht etwa 528MB

  Hoher Speicher-Verbrauch

Performance

Vergleich

Struktur

1 Einleitung

2 Allgemeines

3 Architektur

4 Training

5 Beispiel

6 Bewertung

7 Ausblick

Ausblick