DEPARTMENT OF COMPUTING

IMPERIAL COLLEGE OF SCIENCE, TECHNOLOGY AND MEDICINE

Popular Network Architectures (& BatchNorm)

LeNet-5 [4], MNIST, AlexNet [3], ImageNet, VGG [5], Inception (GoogleLeNet), BatchNorm [2], ResNet [1], DenseNet, Squeeze-Excite Net U-Net, Data Augmentation

Author: Anton Zhitomirsky

Contents

1 LeNet-5

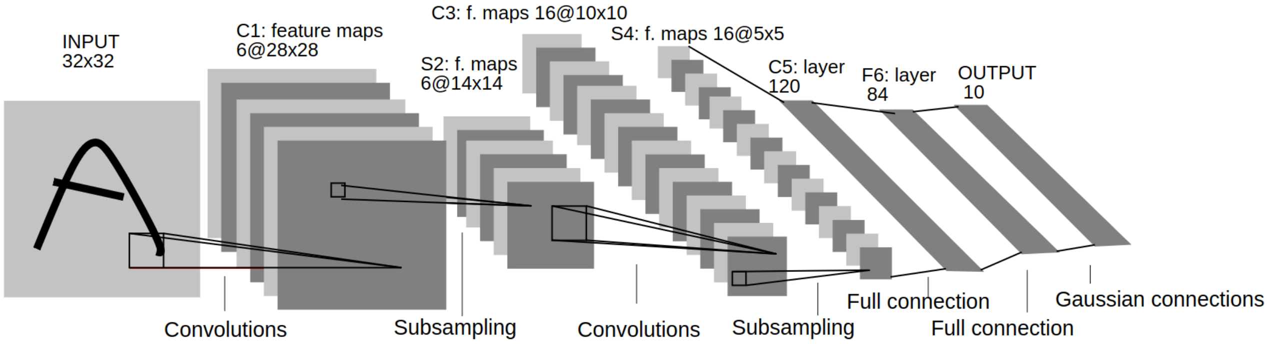

Figure 1: Architecture of LeNet-5 a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical [4]

```python
# from https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html
# https://www.analyticsvidhya.com/blog/2023/11/lenet-architectural-insights-and-practical-implementation/
import torch
import torch.nn as nn
import torch.nn.functional as F


class Net(nn.Module):

    def __init__(self):
        super(Net, self).__init__()
        # 1 input image channel, 6 output channels, 5x5 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 5)
        self.pool1 = nn.AvgPool2d(2, stride=2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.pool2 = nn.AvgPool2d(2, stride=2)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 5 * 5, 120)  # 5*5 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.conv1(x)
        x = F.relu(x)
        x = self.pool1(x)
        x = self.conv2(x)
        x = F.relu(x)
        x = self.pool2(x)

        x.torch.flatten(x,1) # flatten all dimensions except the batch dimension

        x = self.fc1(x)
        x = F.relu(x)
        x = self.fc2(x)
        x = F.relu(x)
        x = self.fc3(x)
```

code/LetNet.py

LeNet [4] was initially designed for low-resolution, black and white image recognition, specifically for digits. It demonstrated that CNNs could reliably perform both tasks of object localization and recognition (on low-resolution black and white images).

The last steps of the LeNet-5 architecture employ fully connected layers to convert features into final predictions. The scenario of MNIST data didn't matter, because of its small number of output classes. This *complicates the architecture when scaling up, influencing both computational resources and architecture*.

The Convolutional Neural Network Architecture ensures some degree of shift, scale and distortion invariance: *local receptive fields, shared weights* (or weight replication), and spatial or temporal *sub-sampling*.

Once a feature has been detected, its exact location becomes less important. Only its approximate position relative to other features is relevant.

1.1 shared weights

This algorithm is particularly useful for shared-weight networks because the weight sharing creates ill-conditioning of the error surface. Because of the sharing, one single parameter in the first few layers can have an enormous influence on the output. Consequently, the second derivative of the error with resp ect to this parameter may b e very large, while it can be quite small for other parameters elsewhere in the network

1.2 sub-sampling

A simple way to reduce the precision with which the position of distinctive features are encoded in a feature map is to reduce the spatial resolution of the feature map. This can be achieved with a so-called sub-sampling layers which performs a local averaging and a sub-sampling, reducing the resolution of the feature map, and reducing the sensitivity of the output to shifts and distortions.

1.3 Loss

Maximum Likelihood Estimation Criterion (MLE), which is equivalent to the Mean Squared Error (MSE)

$$E(W) = \frac{1}{P}\sum_{p=1}^{P} y_{D^P}(Z^P, W)$$

Where y_{D^P} is the output of the D_p-th RBF unit, i.e. the one that corresponds to the correct class of the input pattern Z^p.

It lacks three important properties:

1. **Trivial Solution with Adaptation of RBF Parameters**: If the parameters of the Radial Basis Function (RBF) are allowed to adapt, the MLE criterion has a trivial and unacceptable solution. In this solution, all RBF parameter vectors become equal, leading to a constant and unchanging state of the network. The network effectively ignores the input, and all RBF outputs become equal to zero.

2. **Lack of Competition Between Classes**: The MLE criterion lacks competition between different classes in the training process. Introducing a more discriminative training criterion, referred to as the Maximum A Posteriori (MAP) criterion, could address this issue. The MAP criterion aims to maximize the posterior probability of the correct class or minimize the logarithm of the probability of the correct class given the input. Unlike MLE, MAP introduces a competitive element by pushing up the penalties of incorrect classes in addition to pushing down the penalty of the correct class.

3. **Potential Collapsing Phenomenon**: When RBF weights are allowed to adapt with the MLE criterion, there is a risk of a collapsing phenomenon where all RBF centers become equal. The discriminative criterion of MAP prevents this collapsing effect by keeping the RBF centers apart from each other. The discriminative criterion ensures that the posterior probabilities of different classes remain distinct, preventing the network from ignoring the input.

2 AlexNet

The AlexNet [3] was introduced in 2012 and introduced more layers for larger inputs and larger filters. Originally, the architecture was split into two columns/streams, but this was a workaround for the hardware limitations at the time.

It used the ImageNet dataset which contained color images with nature objects of larger size compared to MNIST (469×387 vs 28×28) and 1.2 million images vs 60k images.

Its key improvements on the LeNet were

- Add a dropout layer after two hidden dense layers (better robustness /regularization)

  Dropout allowed for much deeper networks by introducing regularization not just at the input layer but throughout multiple layers of the network. This made it possible to control the complexity of the model more effectively.

- Change activation function from sigmoid to ReLu (no more vanishing gradient)

  enabling training of deeper networks more efficiently.

- MaxPooling instead of Average Pooling

  This made the learned features more shift-invariant, which is important for object recognition. Max pooling generally retains the most salient features and discards less useful information, making the model more robust.

- Heavy data Augmentation like cropping, shifting, and rotation.

- Model ensembling

  Ensemble methods is a machine learning technique that combines several base models in order to produce one optimal predictive model

- Using softmax function for classification

- Increase in kernel size and stride

  design choice to accommodate the higher resolution of images in the ImageNet dataset compared to MNIST.

AlexNet is substantially more complex, being about 250 times more computationally expensive. However, it is only ten times larger parametrically than LeNet-5. This way AlexNet is notorious for its high memory usage.

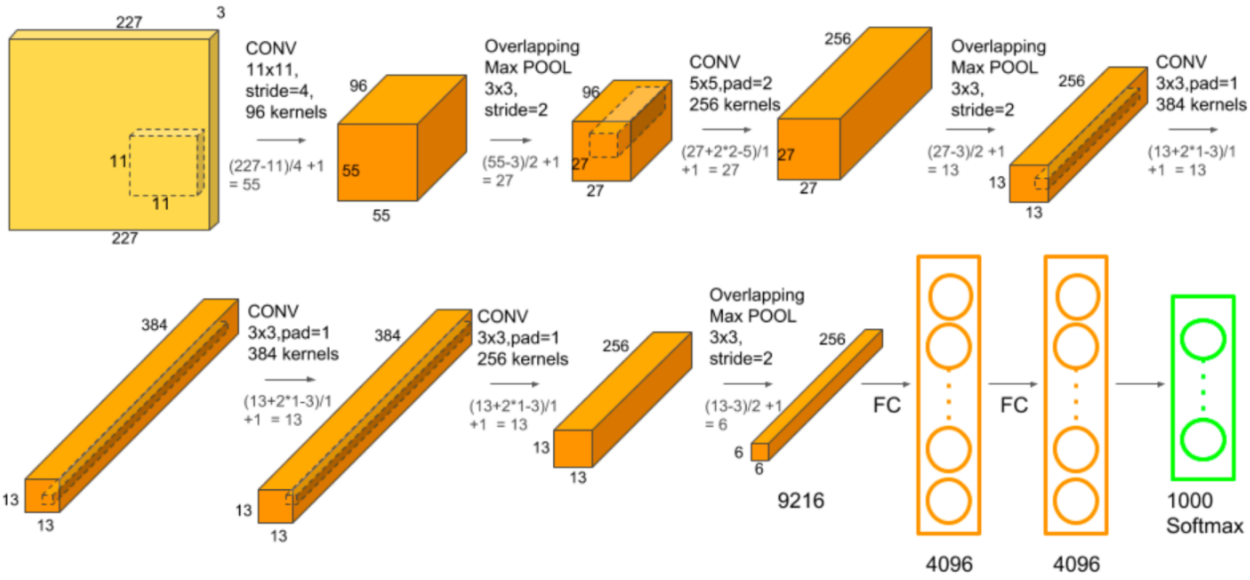

Figure 2: Architecture of AlexNet

```python
import torch
import torch.nn as nn

class AlexNet(nn.Module):
    def __init__(self, num_classes: int = 1000, dropout: float = 0.5) -> None:
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(
            nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(64, 192, kernel_size=5, padding=2),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(192, 384, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.Conv2d(256, 256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
        )
        self.avgpool = nn.AdaptiveAvgPool2d((6, 6))
        self.classifier = nn.Sequential(
            nn.Dropout(p=dropout),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

code/AlexNet.py

3 VGG

The Visual Geometry Group [5] uses the 'bigger means better' philosophy. VGG introduced the notion of repeated blocks.

- **NOT Add more dense layers** - computationally expensive
- **NOT Add more convolutional layers** - as the network grows, specifying each convolutional layer individually becomes tedious
- **Group Layers into blocks** - these blocks can easily be parameterized, creating a more organized, modular architecture

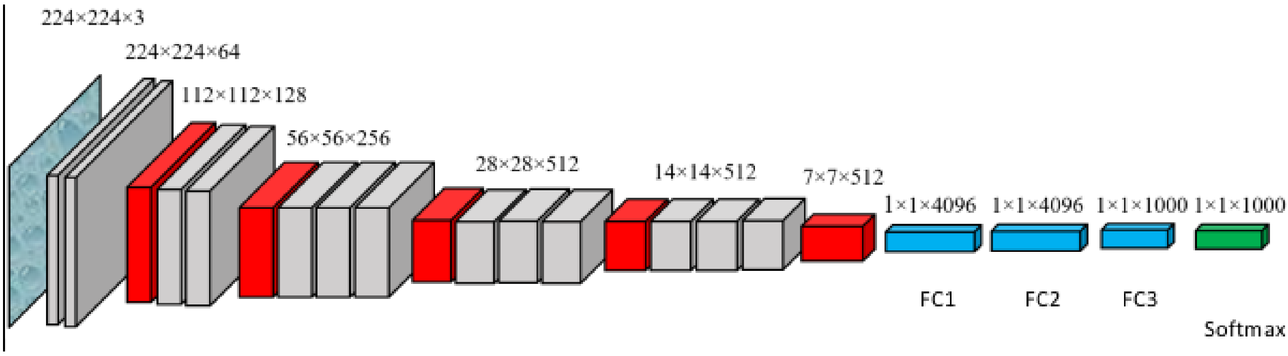

Figure 3: Architecture of VGG

```python
from functools import partial
from typing import Any, cast, Dict, List, Optional, Union

import torch
import torch.nn as nn

cfgs: Dict[str, List[Union[str, int]]] = {
    "A": [64, "M", 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],
    "B": [64, 64, "M", 128, 128, "M", 256, 256, "M", 512, 512, "M", 512, 512, "M"],
    "D": [64, 64, "M", 128, 128, "M", 256, 256, 256, "M", 512, 512, 512, "M", 512, 512, 512, "M"],
    "E": [64, 64, "M", 128, 128, "M", 256, 256, 256, 256, "M", 512, 512, 512, 512, "M", 512, 512, 512, 512, "M"],
}

class VGG(nn.Module):
    def __init__(
        self, features: nn.Module, num_classes: int = 1000, init_weights: bool = True, dropout: float = 0.5
    ) -> None:
        super(VGG, self).__init__()
        self.features = features
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(p=dropout),
            nn.Linear(4096, num_classes),
        )
        if init_weights:
            for m in self.modules():
```

```python
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode="fan_out", nonlinearity="relu")
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)

    def forward(self, x: torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x


def make_layers(cfg: List[Union[str, int]], batch_norm: bool = False) -> nn.Sequential:
    layers: List[nn.Module] = []
    in_channels = 3
    for v in cfg:
        if v == "M":
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            v = cast(int, v)
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)


def _vgg(arch: str, cfg: str, batch_norm: bool, progress: bool, **kwargs: Any ) -> VGG:
    model = VGG(make_layers(cfgs[cfg], batch_norm=batch_norm, **kwargs))
    return model

def vgg16(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> VGG:
    return _vgg('vgg_16', "D", False, pretrained, progress, **kwargs)
```

code/VGG.py

3.1 fewer wide convolutions or more narrow convolutions?

Recent comprehensive analysis from papers has shown that using more layers of narrow convolutions outperforms using fewer wide ones. This has been a general trend in network design: having more layers of simpler functions is generally more powerful than fewer layers of more complex functions.

3.2 The VGG block

Several 3×3 convolutions, padded by one maintains the spatial dimensions from the input to the output layer, and at the end, max-pooling layer of 2x2 and stride of 2 halves the resolution.

Combining these blocks with dense layers, creates an entire family of architectures just by varying the number of blocks.

3.3 Performance

VGG tends to be a lot slower when compared to the throughput of AlexNet, however, it makes up for in terms of accuracy. While VGG might require more computational resources, it generally provides superior performance.

References

[1] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV]. URL: https://arxiv.org/abs/1512.03385.

[2] Sergey Ioffe and Christian Szegedy. *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*. 2015. arXiv: 1502.03167 [cs.LG]. URL: https://arxiv.org/abs/1502.03167.

[3] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet classification with deep convolutional neural networks". In: *Commun. ACM* 60.6 (May 2017), pp. 84–90. ISSN: 0001-0782. DOI: 10.1145/3065386. URL: https://papers.nips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a6 Paper.pdf.

[4] Y. Lecun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2324. DOI: 10.1109/5.726791. URL: http://vision.stanford.edu/cs598_spring07/papers/Lecun98.pdf.

[5] Karen Simonyan and Andrew Zisserman. *Very Deep Convolutional Networks for Large-Scale Image Recognition*. 2015. arXiv: 1409.1556 [cs.CV]. URL: https://arxiv.org/pdf/1409.1556.pdf.