Department of Computing

Imperial College of Science, Technology and Medicine

Popular Network Architectures (& BatchNorm)

LeNet-5 [5], MNIST, AlexNet [4], ImageNet, VGG [6], Inception (GoogleLeNet), BatchNorm [3], ResNet [2], DenseNet, Squeeze-Excite Net U-Net, Data Augmentation

Author: Anton Zhitomirsky

Contents

1 LeNet-5

LeNet [5] 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 *subsampling*.

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 maybe 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.

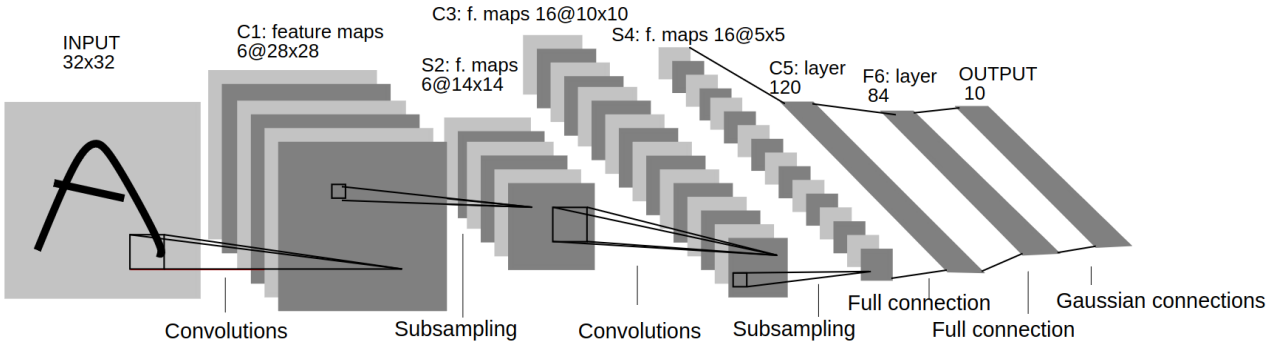


Figure 1: Architecture of LeNet-5 a Convolutional Neural Network, here for digits recognition. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical [5]

```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

2 AlexNet

The AlexNet [4] 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.

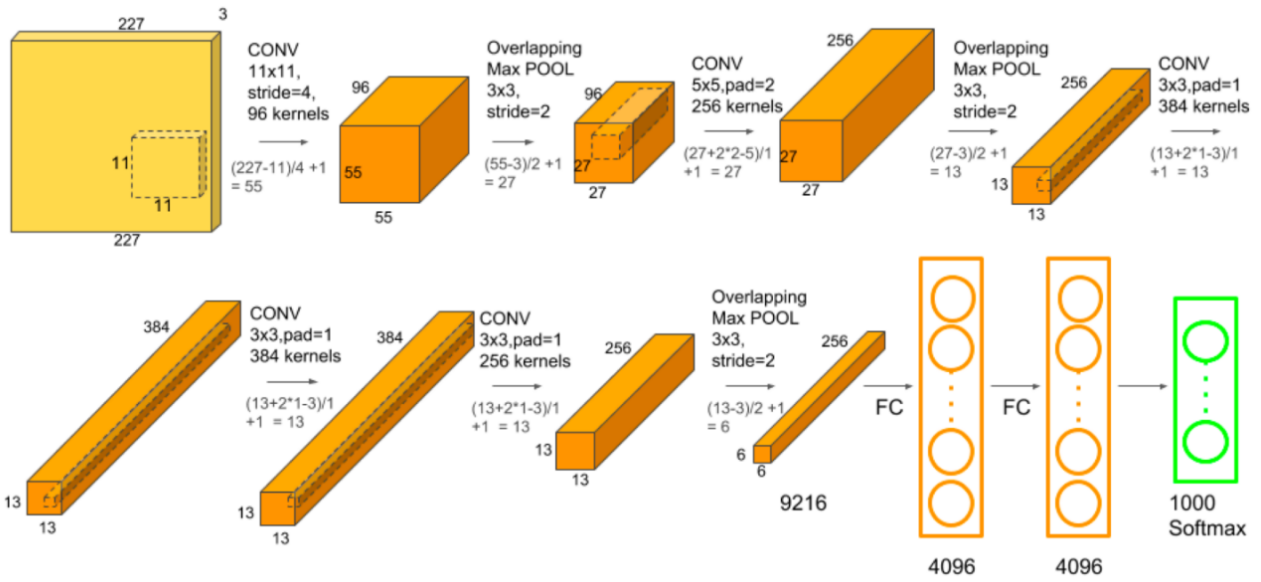


Figure 2: Architecture of AlexNet

```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 [6] 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

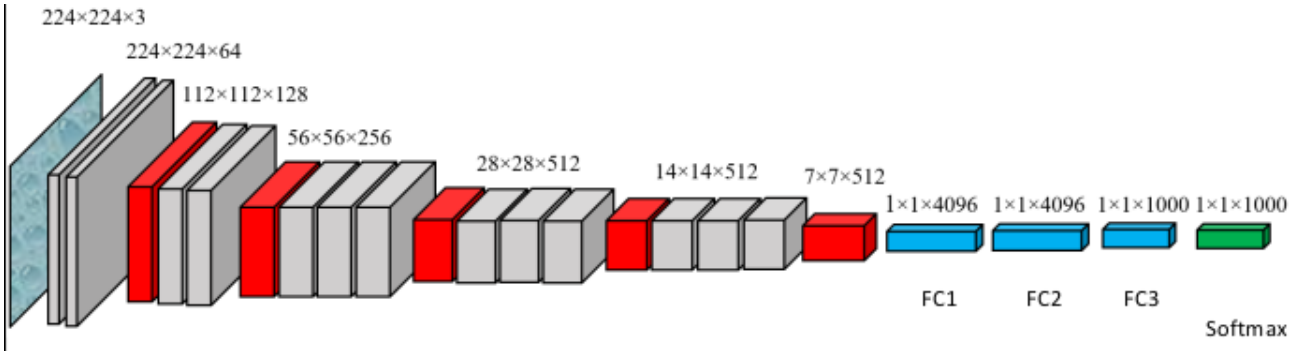


Figure 3: Architecture of VGG

```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():
                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(progress: bool = True, **kwargs: Any) -> VGG:
    return _vgg('vgg_16', "D", False, 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 2×2 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.

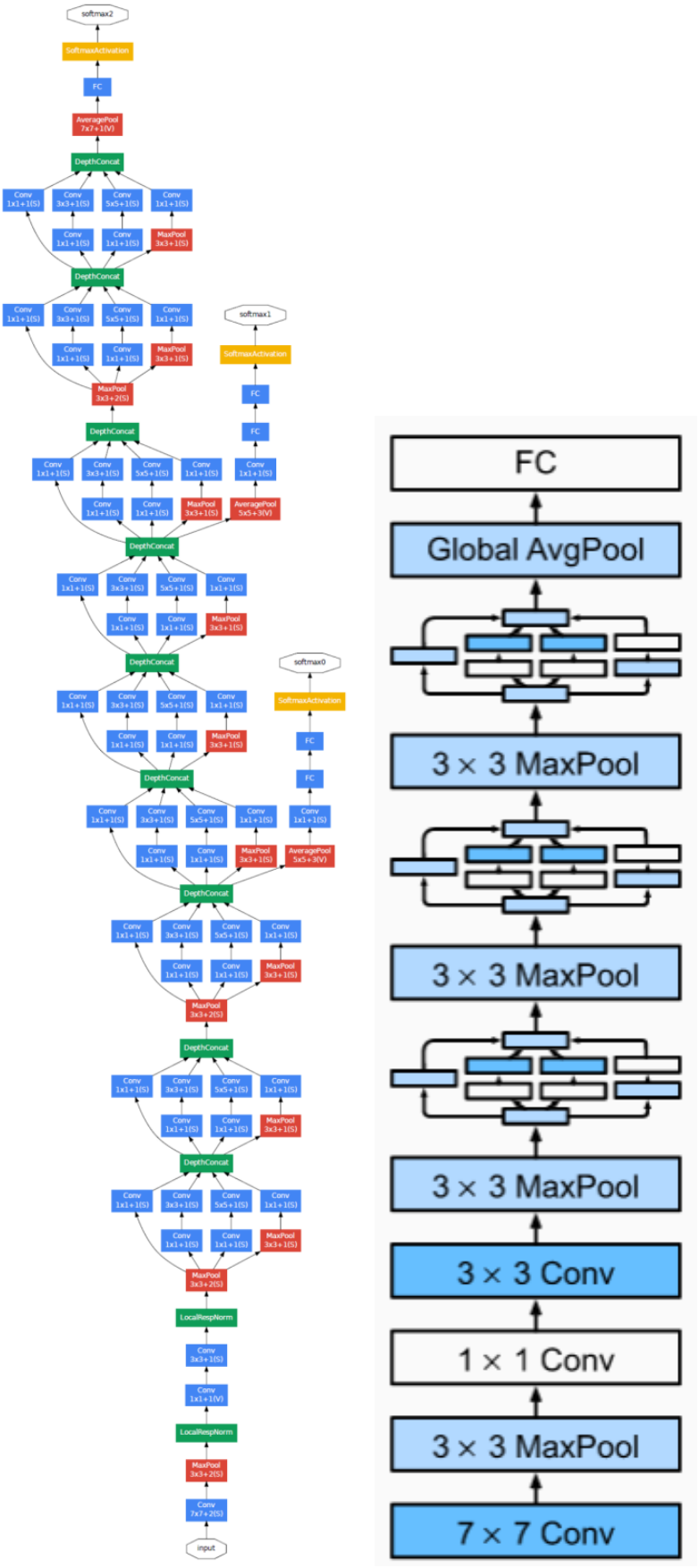
3.3 Performance

VGG tends to be a lot slower when compared to the throughput of AlexNet (Figure 6), however, it makes up for in terms of accuracy. While VGG might require more computational resources, it generally provides superior performance.

4 Inception

The Inception architecture [7] is both deep and introduces the concept of parallel paths within the network which offers multiple pathways for data to flow. The architecture combines the best of different types of convolutions and pooling layers to enhance its performance.

The decision was:



(a) Inception Architecture

(b) Simplified Inception architecture

Figure 4: Inception code can be found at `http://d2l.ai` chapter named 8.4. Multi-Branch Networks (GoogLeNet)

if you opt for 5×5 convolutions, you will end up with many parameters, leading to a lot of computational cost and possibly overfitting, even though it might be more expressive. On the other hand, if you go with 1×1 convolutions, you'll have a more controlled, memory-efficient architecture, but it may not perform as well.

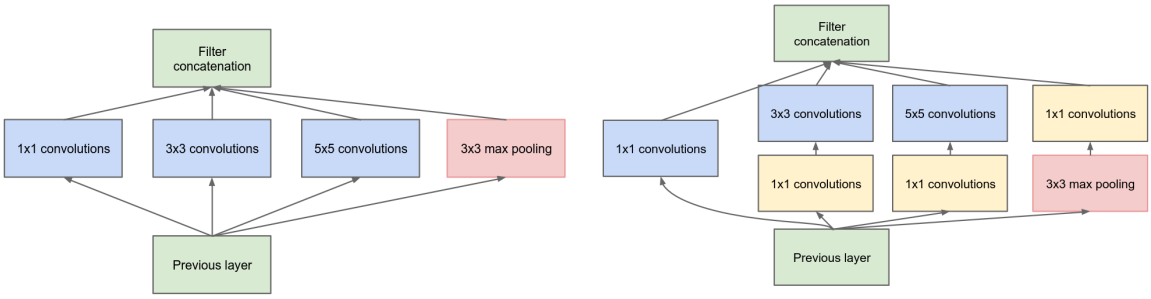
"The most straightforward way of improving the performance of deep neural networks is by increasing their size. This includes both increasing the depth - the number of levels - of the network and its width: the number of units at each level. This is as an easy and safe way of training higher quality models, especially given the availability of a large amount of labeled training data" [7]

This comes with two major drawbacks

- Bigger size typically means a larger number of parameters, which makes the enlarged network more prone to overfitting, especially if the number of labeled examples in the training set is limited.

  This can become a major bottleneck, since the creation of high quality training sets can be tricky and expensive, especially if expert human raters are necessary to distinguish between fine-grained visual categories like those in ImageNet

- Another drawback of uniformly increased network size is the dramatically increased use of computational resources. For example, in a deep vision network, if two convolutional layers are chained, any uniform increase in the number of their filters results in a quadratic increase of computation. If the added capacity is used inefficiently (for example, if most weights end up to be close to zero), then a lot of computation is wasted.



(a) Inception module, naive version (b) Inception module with dimension reductions

Figure 5: Inception Module

4.1 Naiive Inception Module

"In order to avoid patch-alignment issues, current incarnations of the Inception architecture are restricted to filter sizes 1×1, 3×3 and 5×5, however this decision was based more on convenience rather than necessity. It also means that the suggested architecture is a combination of all those layers with their output filter banks concatenated into a single output vector forming the input of the next stage. Additionally, since pooling operations have been essential for the success in current state of the art convolutional networks, it suggests that adding an alternative parallel pooling path in each such stage should have additional beneficial effect, too (Figure 5(a))" [7].

The Problem with the naiive approach is that "even a modest number of 5×5 convolutions can be prohibitively expensive on top of a convolutional layer with a large number of filters. This problem becomes even more pronounced once pooling units are added to the mix: their number of output filters equals to the number of filters in the previous stage. The merging of the output of the pooling layer with the outputs of convolutional layers would lead to an inevitable increase in the number of outputs from stage to stage. Even while this architecture might cover the optimal sparse structure, it would do it very inefficiently, leading to a computational blow up within a few stages" [7].

4.2 Detailed Inception Module

This leads to a second proposed architecture (Figure 5(b)). " judiciously applying dimension reductions and projections wherever the computational requirements would increase too much otherwise. This is based on the success of embeddings: even low dimensional embeddings might contain a lot of information about a relatively large image patch. However, embeddings represent information in a dense, compressed form and compressed information is harder to model. We would like to keep our representation sparse at most places and compress the signals only whenever they have to be aggregated en masse. That is, 1×1 convolutions are used to compute reductions before the expensive 3×3 and 5×5 convolutions. Besides being used as reductions, they also include the use of rectified linear activation which makes them dual-purpose" [7]

4.3 How the Modules fit together

"As these "Inception modules" are stacked on top of each other, their output correlation statistics are bound to vary: as features of higher abstraction are captured by higher layers, their spatial concentration is expected to decrease suggesting that the ratio of 3×3 and 5×5 convolutions should increase as we move to higher layers" [7].

Now, you might be wondering, how do all these different types of convolutions fit together in the Inception block? The key to making this work is using appropriate padding. For example, the 3×3 convolutions use half-padding by one, and the 5×5 convolutions use half-padding by two. This ensures that the dimensions of the inputs and outputs align and remain the same. Once that's taken care of, you simply stack all these layers together.

4.4 Channels

The first inception block has channels sizes specified.

The first Inception block uses 64 channels for the 1×1 convolutions. For the 3×3 convolutions, it uses 128 channels. And for the 5×5 convolutions, 32 channels are used, primarily because 5×5 convolutions already come with a large number of parameters—25 times 32 in this case. When it comes to max pooling, a few other dimensions are included. The goal is to have all these different parts sum up to 256 channels.

4.5 Advantages

The Inception block is designed to use a relatively low number of parameters (better than just striaght up sticking with one dimension of convolution) and floating-point operations (FLOPs), without sacrificing performance.

It has less computational operations:

$$k^2 \times \overbrace{\boxed{c_{in}}}^{fixed} \times c_{out} \times \overbrace{\boxed{m_h \times m_w}}^{fixed}$$

Consider a convolutional kernel of size $k \times k$, c relating to the channels and dimension of the image m.

We have the flexibility to adjust $c_{in} \wedge c_{out}$.

$$\overbrace{\boxed{c_{in} \times m_h \times m_w}}^{fixed} \times \sum_{j \in paths} \left(k_j^2 \times c_{out,j}\right)$$

By carefully allocating resources - varying the number of channels and kernel sizes - we can optimize the network's performance.

4.6 Channels

The Inception blocks are designed in such a way that they need fewer parameters and less computational complexity than a single 3x3 or 5x5 convolutional layer, as shown in Table 1. If we were to have 256 channels in the output layer, Inception needs only 16,000 parameters and costs only 128 Mega FLOPS, whereas a 3x3 convolutional layer will need 44,000 parameters and cost 346 Mega FLOPS, and a 5x5 convolutional layer will need 1,22,000 parameters and cost 963 Mega FLOPS. So Inception blocks, essentially get the same job done as single convolutional layers, with much better memory and compute efficiency. from here

5 Batch Norm

The problem deep networks face is with convergence, making training difficult and time consuming. There is a strategy to perform "deep supervision" which involves backpropogating from intermediate stages to help the network learn - however, this has its limitations [TODO].

5.1 Motivation

Very deep models involve the composition of several functions or layers. The gradient tells how to update each parameter, under the assumption that the other layers do not change. In practice, we update all of the layers simultaneously; gradients flow from the top layer down to the bottom layers of the network. As a result, the last layers start to adapt first, followed by the layers below them, creating a cascade of adaptations throughout the network.

However, this leads to a problem: As the bottom layers adapt, they change the features that are fed back up to the top layers. This means that the top layers, which had already started to adapt, have to readjust to these new inputs. Training Deep Neural Networks is complicated by the fact that the distribution of each layer's inputs changes during training, as the parameters of the previous layers change [1]. **Each layer's learning destabilizes the next, causing a slow convergence process that takes a long time for all layers to adapt properly**.

5.2 Solution

Batch normalization can be applied to any input or hidden layer in a network [1].

Batch Norm mitigates this issue by normalizing the features within each mini-batch, thereby stabilizing the training process and speeding up convergence. The idea is to make minor corrections to the layers by adjusting their mean and variance during training.

$$\mu_B = \frac{1}{|B|}\sum_{i\in B} x_i \quad \sigma_B^2 = \frac{1}{|B|}\sum_{i\in B}(x_i - \mu_B)^2 + \epsilon$$

$$x_{i+1} = \gamma\frac{x_i - \mu_B}{\sigma_B} + \beta \tag{1}$$

Where γ is variance and β is the mean. $|B|$ represents the size of the mini-batch, and x_i are the individual data points in that mini-batch. ϵ is a small positive value such as 10^{-8} introduced to avoid the potential problem of undefined gradient.

We compute the mean and variance of a given mini-batch during training. Then, re-normalize each input feature by subtracting its mean and dividing it by its standard deviation. The model also learns two parameters to scale (γ) and shift (β) the normalized features.

We then adjust these first and second moments separately from the rest of the network learning, leading to Equation 1.

5.3 Advantages

This way, we are able to "normalize" each mini-batch separately, making the training of deep networks more stable and faster (dramatically reducing the number of training epochs required to train deep networks). The normalization can be undone by the network by learning appropriate γ and β parameters if it sees a benefit to it.

5.4 Updated Aim

The original motivation of batch normalization is reducing covariate shift. This is not correct. BatchNorm actually worsens covariate shift, yet it still aids in model convergence.

BatchNorm is effectively acting as a form of regularization by introducing noise into the model.

Here's how it works: You calculate the mean and variance of a mini-batch, let's say, of 64 observations. Since you're working with a small sample, both the mean and the variance are subject to noise. You then normalize the features using these noisy statistics, introducing a random scale and shift into your model at each batch. This random noise acts as a regularizing factor, which is why you often don't need additional regularization techniques like dropout when you're using BatchNorm. However, this property makes BatchNorm sensitive to the size of the mini-batch.

If your mini-batch is too large, you're not introducing enough noise for effective regularization. If it's too small, the noise level becomes counterproductive, affecting convergence. This mini-batch size sensitivity becomes especially significant in multi-GPU settings, where batch sizes are often adjusted.

5.5 Application

If you're working with a dense layer, a single normalization is applied to all the activations in that layer.

In the case of a convolutional layer, a separate normalization is performed for each channel.

5.6 Randomization during use

You don't want this randomness when you're using the model for predictions. So, you fix the gamma and beta parameters that the model has learned during training. Instead of using batch statistics, you use the running average for the mean and variance to normalize the features.

6 Performance

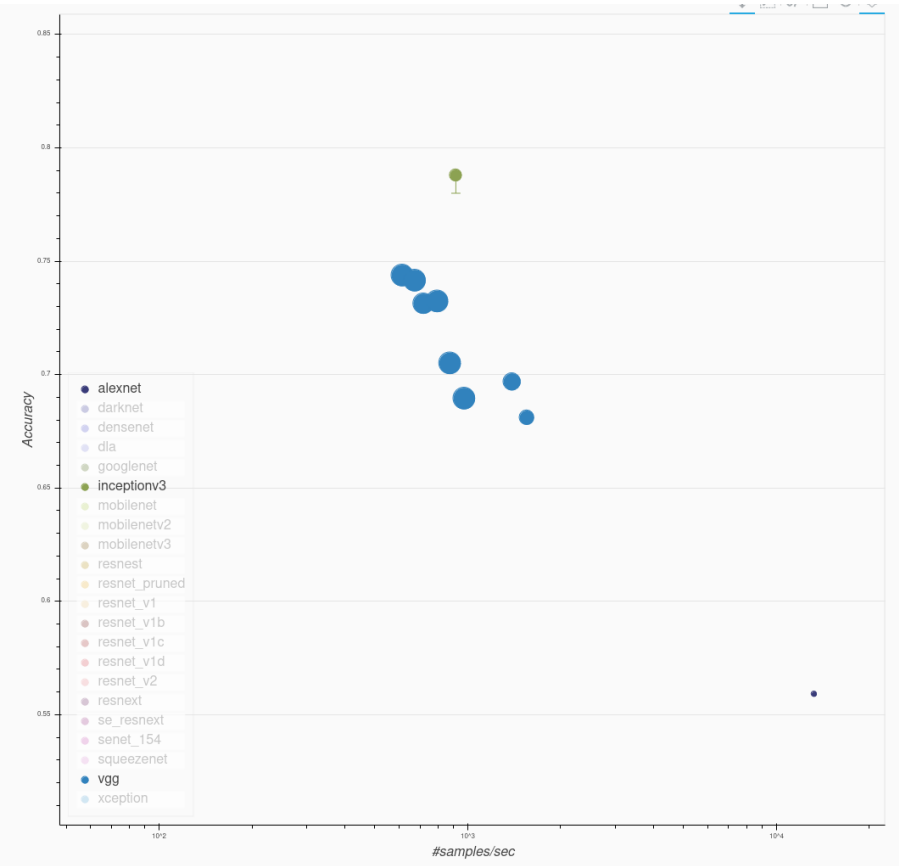


Figure 6: The performance of the architectures

References

[1] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

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

[3] 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.

[4] 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/c399862d3b9d6b76c8436e924a6Paper.pdf.

[5] 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.

[6] 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.

[7] Christian Szegedy et al. *Going Deeper with Convolutions*. 2014. arXiv: 1409.4842 [cs.CV]. URL: https://arxiv.org/abs/1409.4842.