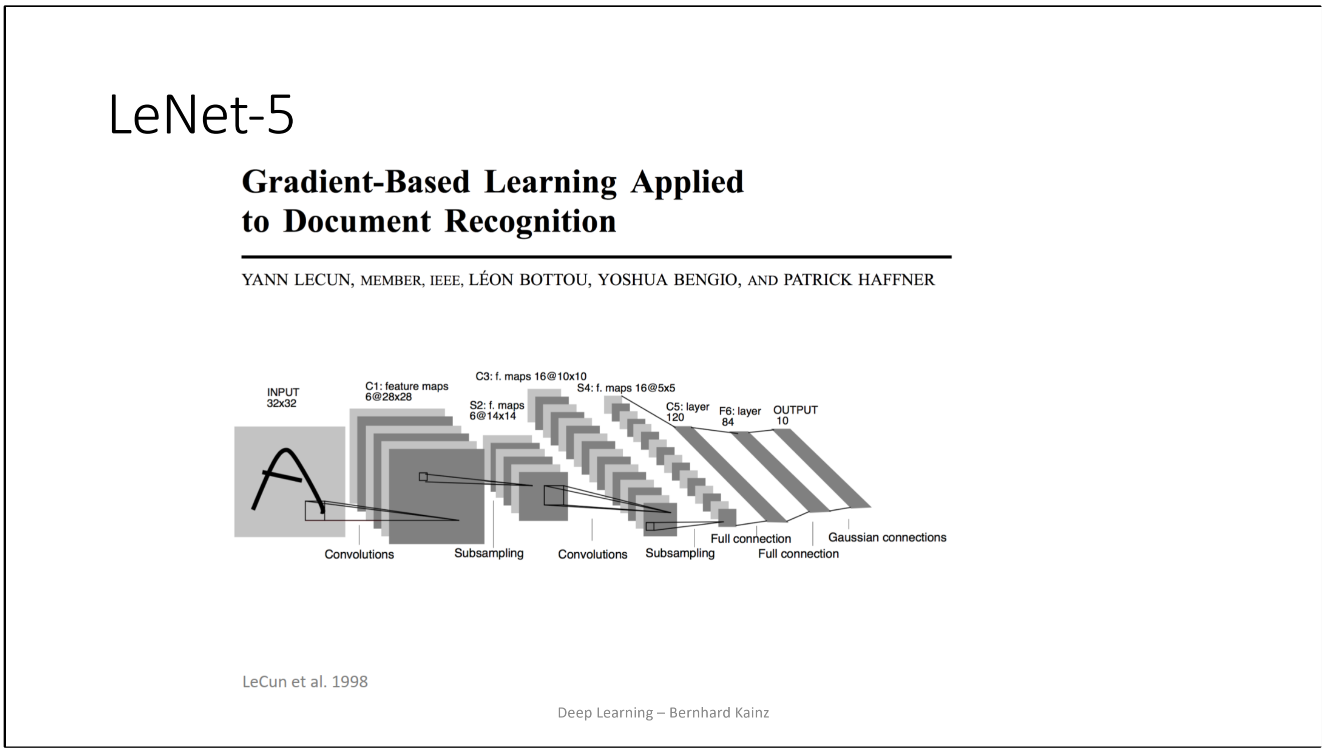
Deep Learning – some popular architectures and history

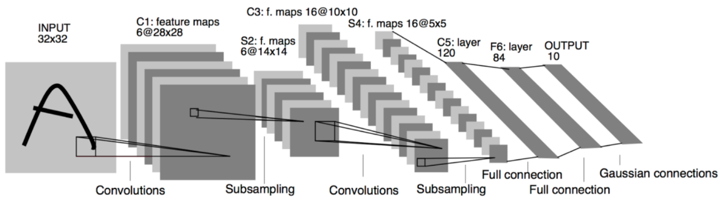
Bernhard Kainz



LeNet-5

Gradient-Based Learning Applied to Document Recognition

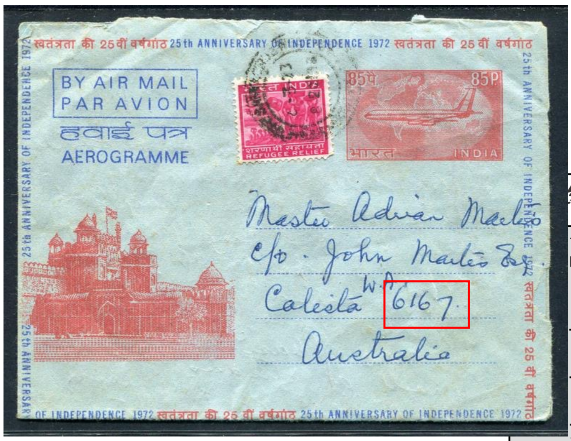
YANN LECUN, MEMBER, IEEE, LÉON BOTTOU, YOSHUA BENGIO, AND PATRICK HAFFNER

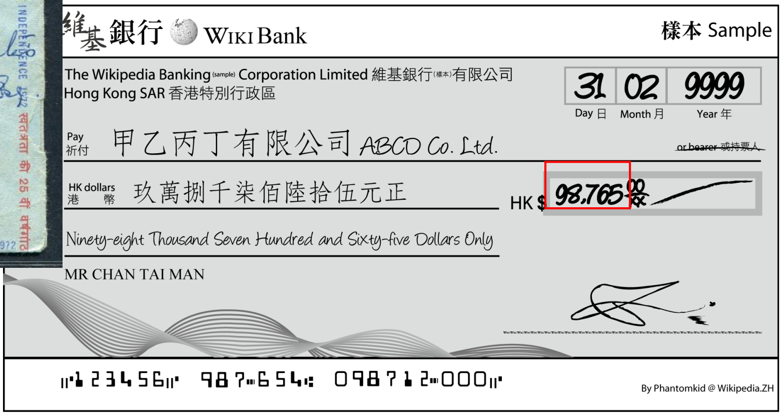


LeCun et al. 1998

This is LeNet-5, developed by Yann LeCun and his team in 1995.
Firstly, it's important to appreciate the historical context. Back in the '90s, the field of deep learning was nowhere near as advanced as it is today, and creating a neural network was a significant engineering challenge. Building LeNet-5 was a monumental effort that likely took between three to six months to implement, including all the tooling.
Let's look at the architecture. LeNet was initially designed for low-resolution, black and white image recognition, specifically for digits. The input to the network is a 32x32 pixel grayscale image.
The first step is a convolutional layer that transforms the input into six feature maps, each of size 28x28. Following that, an average pooling layer reduces the dimensionality to 14x14, but still retains the six channels. Note that pooling is applied individually to each channel and not across them, so the number of channels remains unchanged.
A second convolutional layer further reduces the feature maps to a size of 10x10 but increases the number of channels to 16. Another round of average pooling halves the dimensions again, leaving us with a 5x5 feature map with 16 channels.
The architecture employs fully connected layers. The first has 120 units, followed by another one with 84 units. The last layer is quite interesting; it used a Gaussian Radial Basis Function (RBF) to map these features into 10 classes.
Given the technology and resources available at the time, LeNet-5 was a significant step forward in the field and laid the groundwork for future CNN architectures. It's crucial to recognize its contribution to show how far we've come, and how these early networks influenced today's far more complex architectures.

Handwritten digit recognition





維基銀行 WIKI Bank 樣本 Sample

The Wikipedia Banking (sample) Corporation Limited 維基銀行(樣本)有限公司
Hong Kong SAR 香港特別行政區

31 02 9999
Day 日 Month 月 Year 年

Pay 祈付 甲乙丙丁有限公司 ABCD Co. Ltd. ~~or bearer 或持票人~~

HK dollars 港幣 玖萬捌千柒佰陸拾伍元正 HK $ 98,765.00

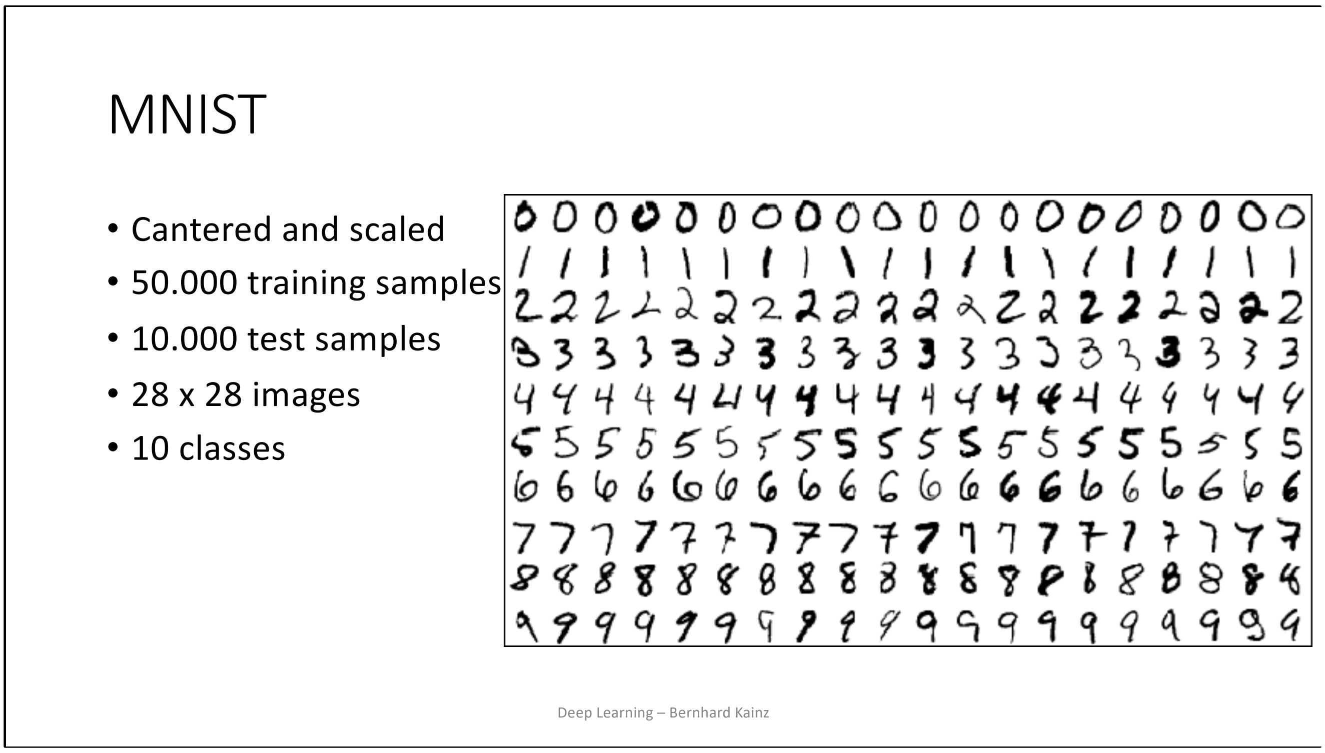
Ninety-eight Thousand Seven Hundred and Sixty-five Dollars Only

MR CHAN TAI MAN

⑆123456⑆ 987⑉654⑈ 098712⑉000⑆

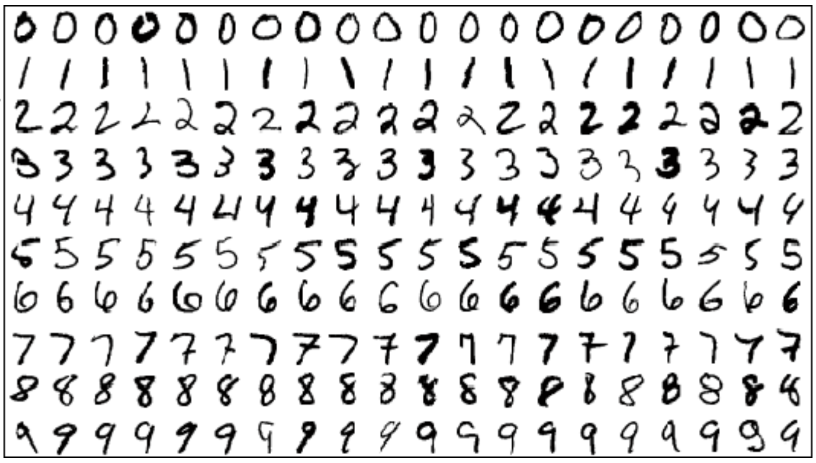
By Phantomkid @ Wikipedia.ZH

Why was LeNet such a big deal in 1995?
The answer lies in the immediate application potential of the technology. At the time, AT&T had an ongoing project focused on handwritten digit recognition.
This was not just an academic exercise; there was real commercial interest and immediate need. Specifically, both banks and postal services were seeking ways to automate their operations.
Imagine the postal office.
Every day, they would receive thousands of letters with handwritten postal codes that needed to be sorted and processed manually. Similarly, banks had to deal with handwritten checks, including handwritten amounts that needed to be read and processed.
Automating these operations would save time and reduce human error, but it required a level of technology that could reliably perform handwriting recognition.
However, the challenge was twofold.
Not only did they need to recognize the characters, but they also had to locate them within a larger document--a form of object recognition.
You can't predict what a digit is until you know where it's located on the page or check.
That's where LeNet came in. Its pioneering architecture demonstrated that Convolutional Neural Networks could reliably perform both tasks--object localization and recognition--on low-resolution black and white images. It was the solution to a problem that had immediate practical applications, making it a significant milestone not just in the field of machine learning, but also for commercial technologies.
So, in a nutshell, LeNet wasn't just a breakthrough in machine learning theory; it was a technology that met a pressing real-world need, and that's why it was such a big deal.



MNIST

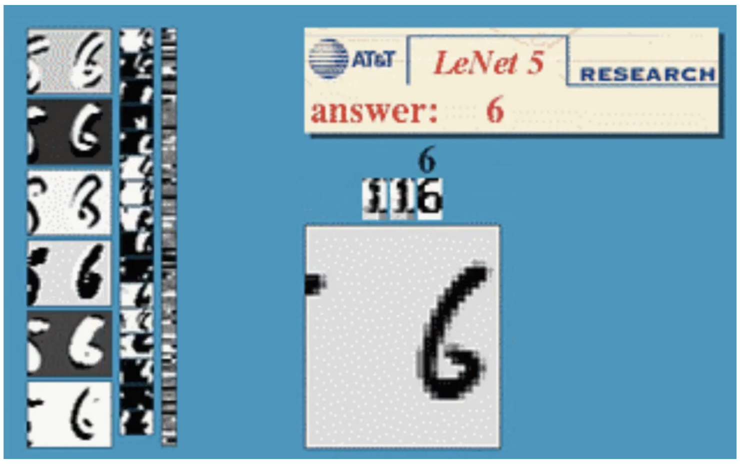
- Cantered and scaled
- 50.000 training samples
- 10.000 test samples
- 28 x 28 images
- 10 classes



MNIST was created specifically for handwritten digit recognition.
It's a curated collection of centered and scaled images, making it easier for models to learn the necessary features.
The dataset is divided into 50,000 images for training and an additional 10,000 images for testing.
Each image is at a resolution of 28 by 28 pixels, which keeps the computational requirements reasonable without sacrificing too much detail.
Now, you might wonder why there are exactly 10 classes in MNIST.
Well, the answer is straightforward: there are 10 digits—0 through 9—that we commonly use.
And it's worth noting that these aren't just any digits. They're realistic digits obtained from real-world examples like letters.
These digits have been segmented properly to ensure that they are good representations for the task of handwritten digit recognition.
So, the MNIST dataset served as both a benchmark and a real-world applicable resource when it was first released, and it continues to be a staple in the machine learning community.

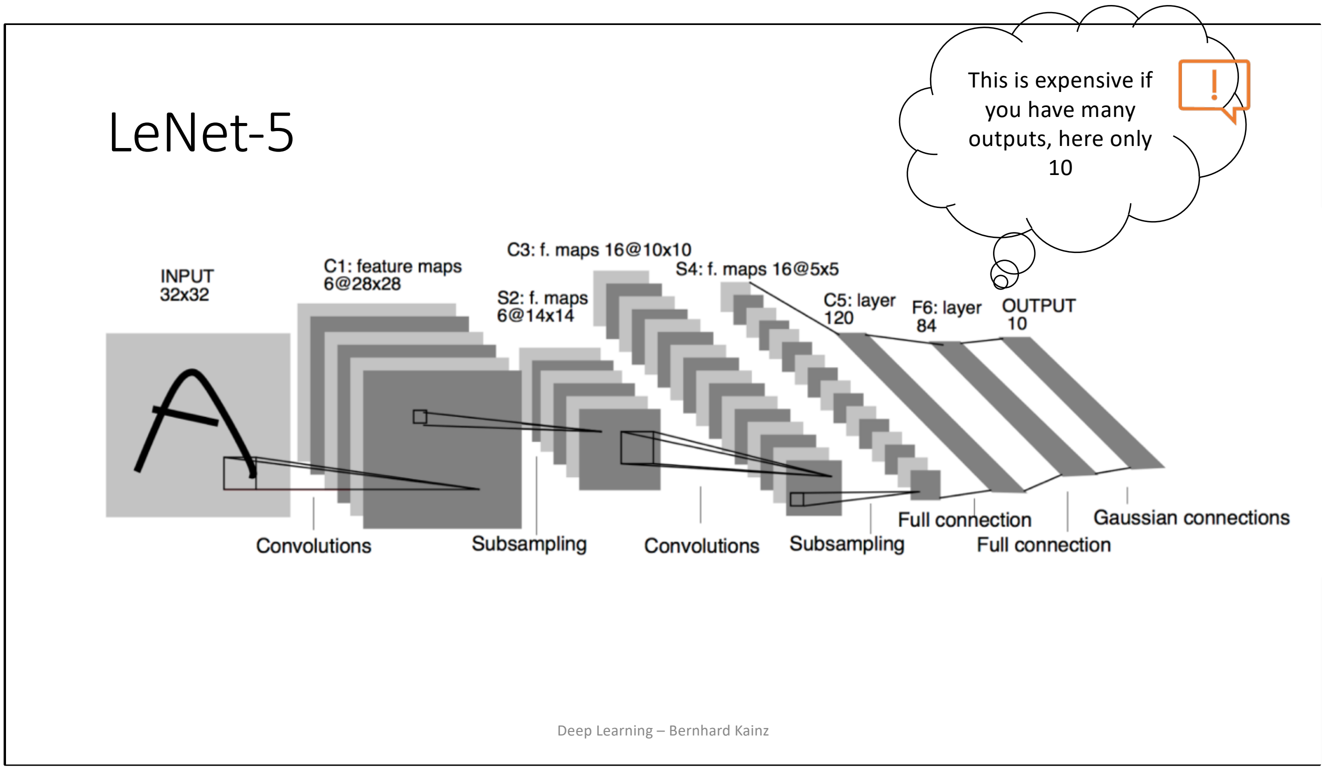
Demo from 1995

https://www.youtube.com/watch?v=yxuRnBEczUU



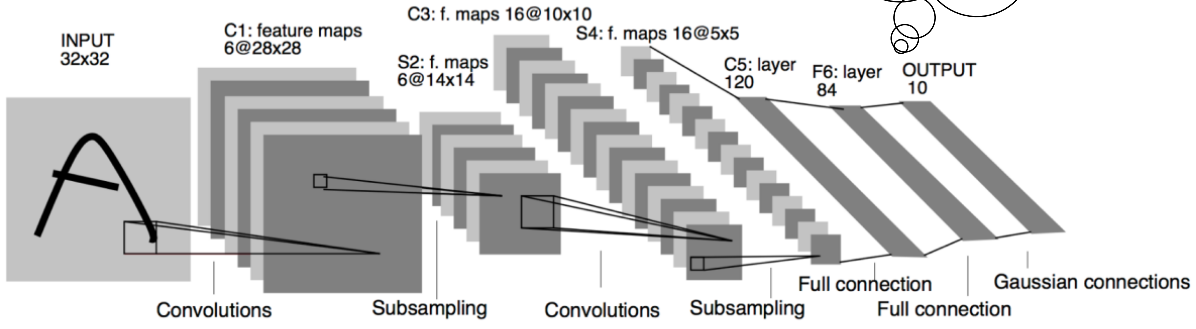
This is the LeNet-5 architecture in action from 1995.
If back to the future would play now, Martie McFly would travel back to this time!

In the demo, you'll notice digits being scanned and processed through the network.
The network then outputs its estimate of what digit it thinks is represented by the input.
Interestingly, even if the digit is slightly shifted, the network still accurately recognizes it. For example, a five remains a five regardless of its position.
On the left-hand side of the demo, you'll see the activations of the various layers in the network.
In the first layer, right after the convolutions, the activations act like edge detectors. They identify vertical and horizontal edges and enhance contrast.
Moving to the next layer, these basic features are aggregated into higher-level, but still spatially related, features.
As we go further into the network, you'll see the activations for the fully connected layers, which are much more diverse in what they represent.
Ultimately, all these activations contribute to an output that is a vector of probabilities, each representing the likelihood of a particular digit being the input.
The 1998 paper that presented this LeNet-5 demo is a landmark paper in the field of computer vision.
I strongly recommend reading it. We've also included it in the notes for this lecture.
The paper also goes into detail about graph transducers, a topic that, even today, isn't as fully appreciated as it should be.

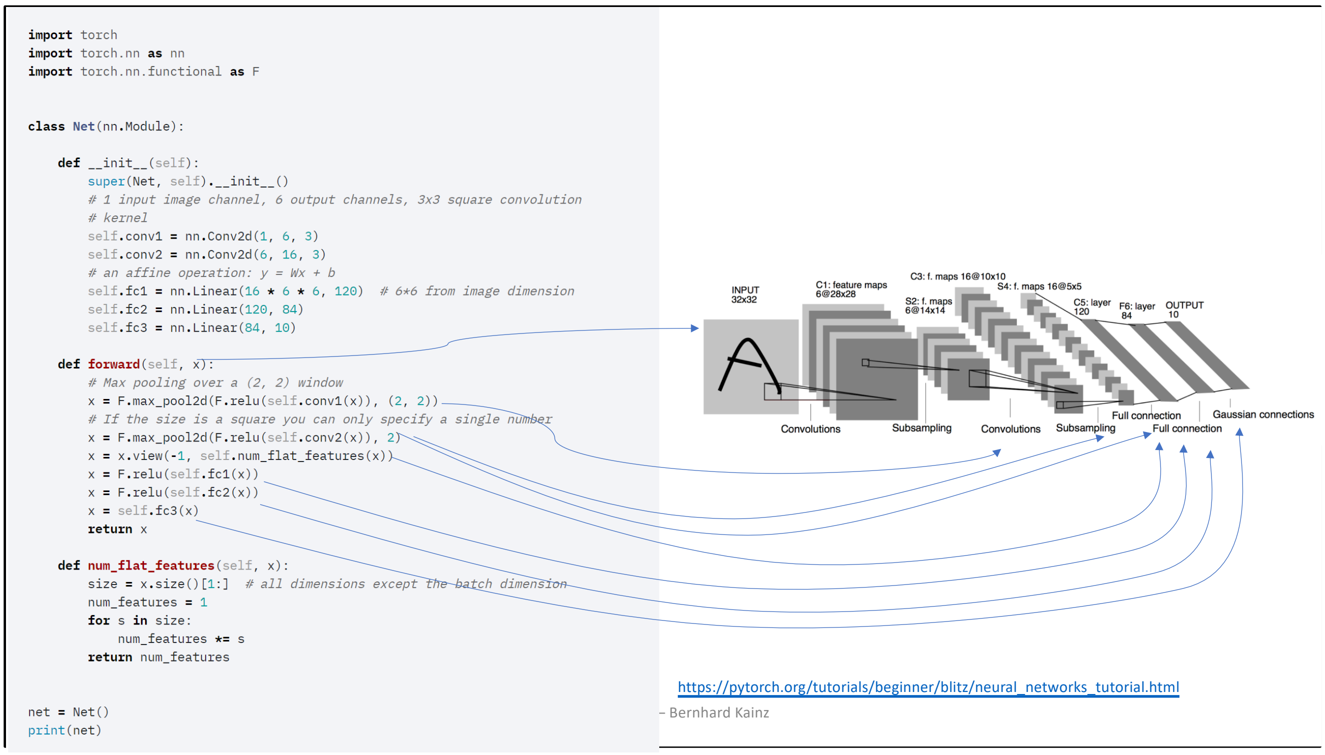


LeNet-5

This is expensive if you have many outputs, here only 10



Let’s dig deeper into the role of fully connected layers in the LeNet-5 architecture.
The final stages of LeNet-5 employ fully connected layers to convert features into final predictions.
In scenarios with a small number of output classes, such as the ten digits in the MNIST dataset, the computational burden is manageable.
However, as we scale to problems like ImageNet, with perhaps a thousand different classes, the fully connected layers become a computational bottleneck.
So while they are straightforward and effective for smaller problems, they can complicate the architecture when scaling up.
These layers then become a dominant factor in network design, influencing both computational resources and performance.
That's why alternative strategies might be needed to handle the computational expenses for large-scale problems.
So, while fully connected layers have their merits, they also bring along some challenges we need to consider when designing more complex networks.



```python
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, 3x3 square convolution
        # kernel
        self.conv1 = nn.Conv2d(1, 6, 3)
        self.conv2 = nn.Conv2d(6, 16, 3)
        # an affine operation: y = Wx + b
        self.fc1 = nn.Linear(16 * 6 * 6, 120)  # 6*6 from image dimension
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        # Max pooling over a (2, 2) window
        x = F.max_pool2d(F.relu(self.conv1(x)), (2, 2))
        # If the size is a square you can only specify a single number
        x = F.max_pool2d(F.relu(self.conv2(x)), 2)
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features


net = Net()
print(net)
```

https://pytorch.org/tutorials/beginner/blitz/neural_networks_tutorial.html

– Bernhard Kainz

Today, implementing LeNet-5 is quite straightforward, thanks to frameworks like PyTorch.
Contrast this to 1995, when implementing this network would have been a significant engineering feat.
For our implementation, we use the torch.nn.functional library to directly access various network functions.
While PyTorch offers multiple ways to define these functions, we opt for a straightforward approach here.
Our network class will inherit from nn.Module, a requirement for compatibility with PyTorch's optimizer.
Within this class, we define all the necessary layers and functionalities.
PyTorch's nn.Module also requires us to define a forward function, through which the data will pass.
The backward propagation is automatically handled by PyTorch, as long as you're using differentiable functions.
For the forward pass, we aim for a sequential composition of layers.
Here, we have two sets of convolution and average pooling layers, followed by two fully connected layers.
And just like that, you've got a functional LeNet-5 model built with modern tools.

AlexNet

AlexNet

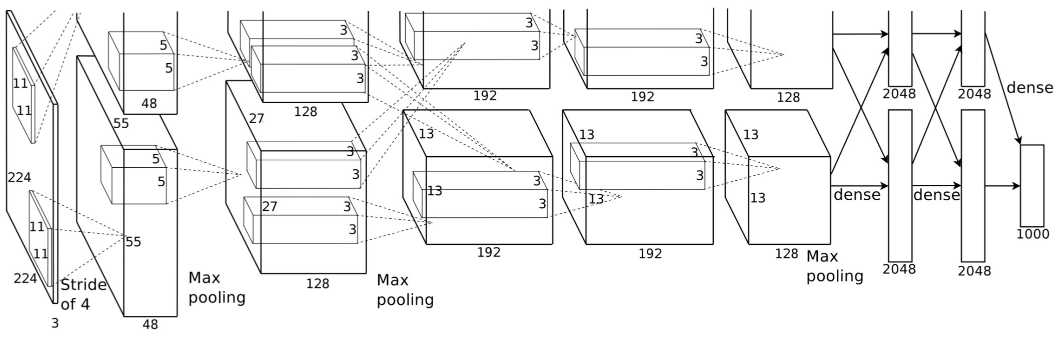
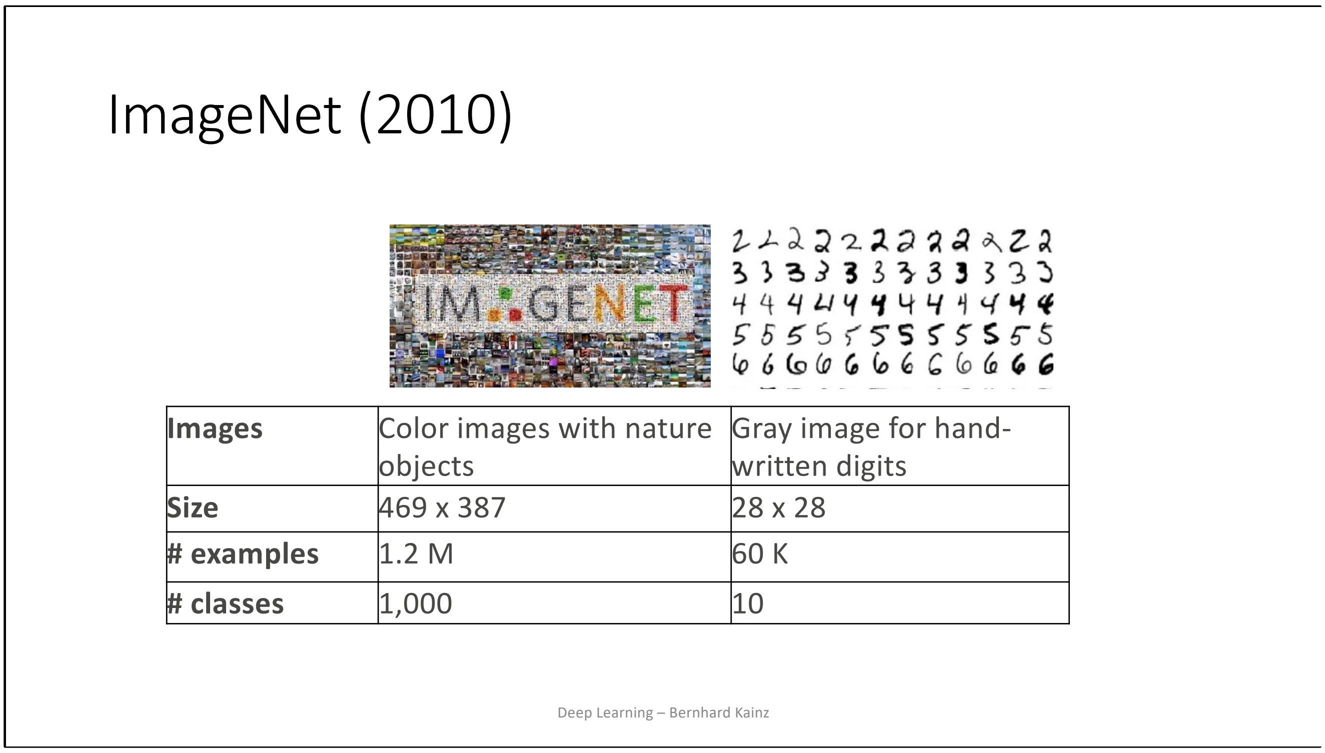


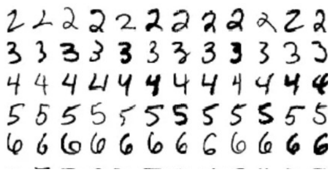
Figure 2: An illustration of the architecture of our CNN, explicitly showing the delineation of responsibilities between the two GPUs. One GPU runs the layer-parts at the top of the figure while the other runs the layer-parts at the bottom. The GPUs communicate only at certain layers. The network's input is 150,528-dimensional, and the number of neurons in the network's remaining layers is given by 253,440–186,624–64,896–64,896–43,264–4096–4096–1000.

AlexNet, introduced in 2012, marked a turning point in the field of computer vision.
It competed in the ImageNet classification task and drastically outperformed all existing non-deep learning models.
This was the architecture that truly kickstarted the widespread adoption and research into convolutional neural networks.
Structurally, AlexNet resembles LeNet, but with a few key differences.
First, it has more layers: Five convolutional layers and three fully connected layers.
The input images are 224 by 224 pixels with 3 color channels, unlike LeNet which handled much smaller grayscale images.
AlexNet uses 11 by 11 filters with a stride of 4, though later research has shown that smaller filters can also be effective.
The fully connected layers are of size 4096, and the final layer connects to the 1000 classes in ImageNet through a softmax activation.
Now, if you've ever looked at an AlexNet diagram, you might notice it's split into two columns or streams.
This was a workaround for the hardware limitations at the time.
The network was trained on GTX580 GPUs with only 3GB of memory.
To fit the model, the architecture was divided between two GPUs, each handling half of the neurons or feature maps.
In layers like conv1, conv2, conv4, and conv5, feature maps from the same GPU were used for connections.

However, in layers like conv3 and FC6, FC7, and FC8, both GPUs communicated to give neurons a full view of the preceding layer.
This kind of parallel processing is far easier today, thanks to features in modern frameworks like PyTorch's DataParallel.

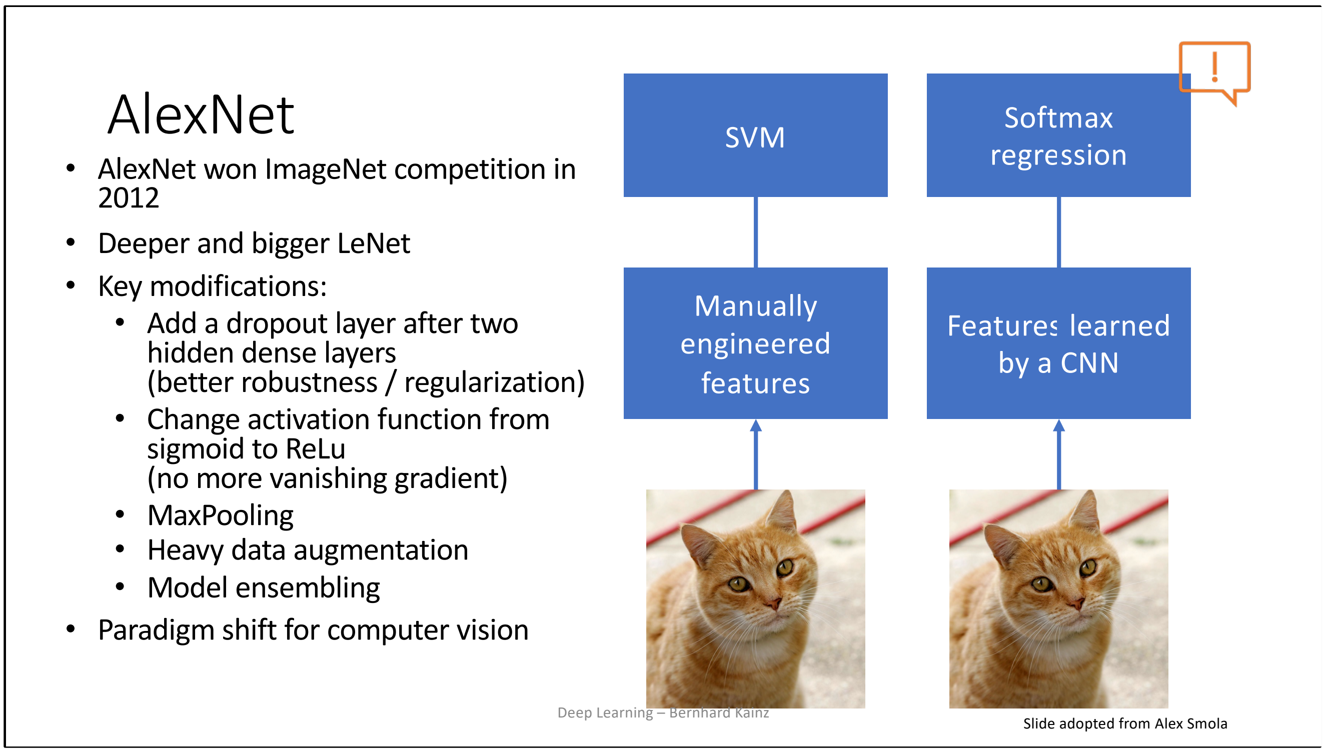


ImageNet (2010)



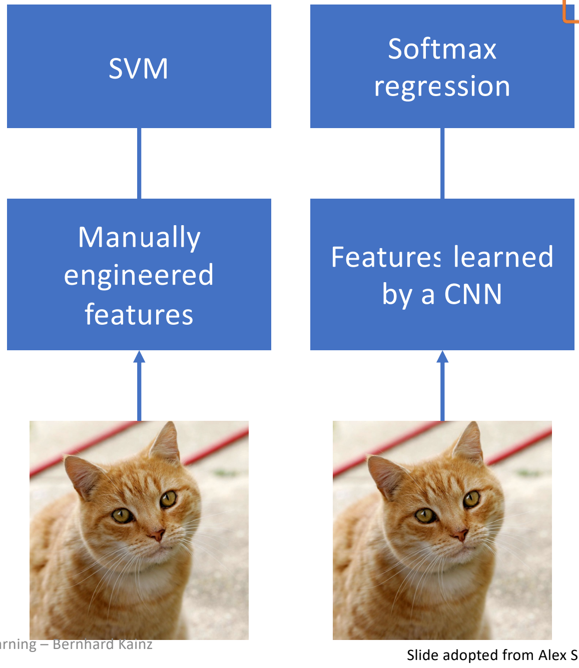
Images	Color images with nature objects	Gray image for hand-written digits
Size	469 x 387	28 x 28
# examples	1.2 M	60 K
# classes	1,000	10

Let's delve into the data that made AlexNet's breakthrough possible.
The ImageNet dataset was released in 2010 and was considered a massive dataset at that time.
It contained a whopping 1.2 million examples distributed across a thousand different classes.
To give you some context, compare this to MNIST, which only had 60,000 samples and 10 classes.
Not only did the quantity of data change but also the complexity.
ImageNet featured images with dimensions ranging from 469 by 384 pixels, a significant jump from MNIST's 28 by 28 pixel images.
Additionally, ImageNet images were in full color, represented by three channels: red, green, and blue.
This was a departure from the grayscale images that networks like LeNet were initially designed to handle.
The increase in both the volume and complexity of the data made ImageNet a challenging task, but also one that showcased the capabilities of models like AlexNet.
This dataset and its challenges pushed the boundary of what was possible in image classification at that time.



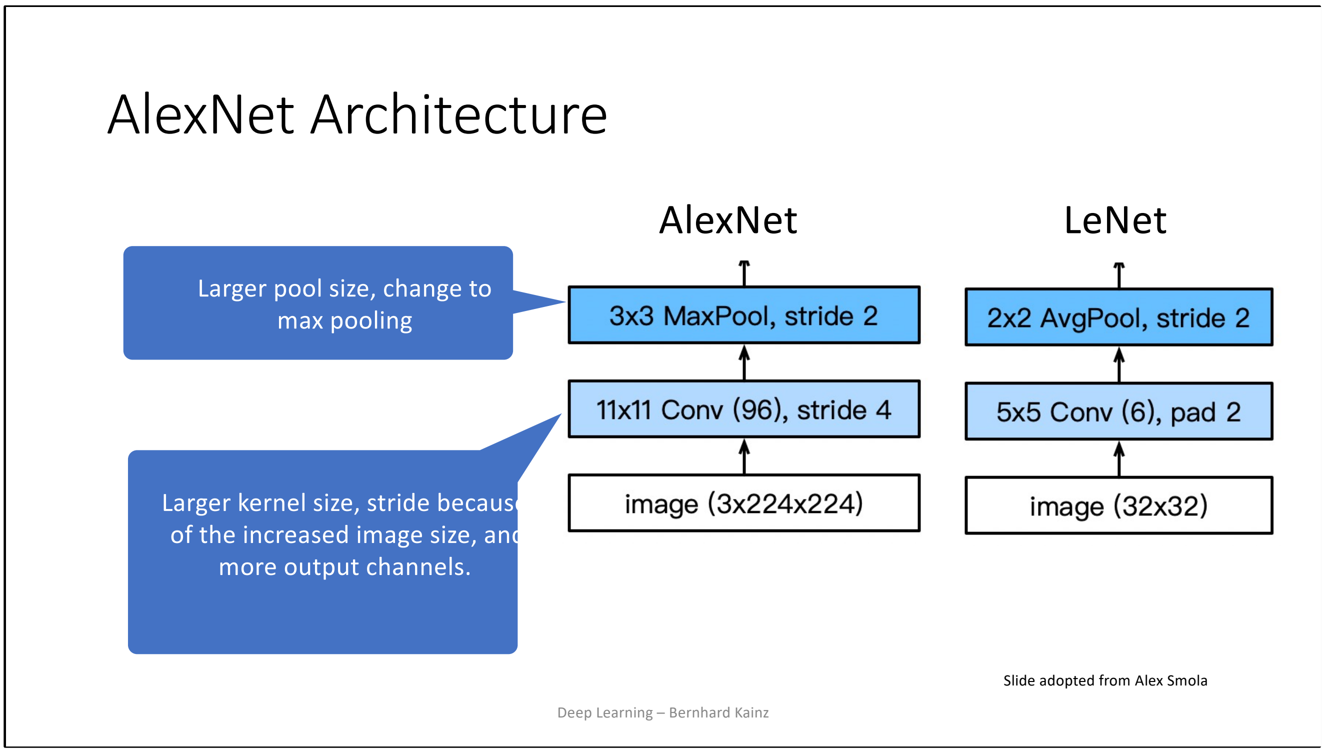
AlexNet

- AlexNet won ImageNet competition in 2012
- Deeper and bigger LeNet
- Key modifications:
  - Add a dropout layer after two hidden dense layers (better robustness / regularization)
  - Change activation function from sigmoid to ReLu (no more vanishing gradient)
  - MaxPooling
  - Heavy data augmentation
  - Model ensembling
- Paradigm shift for computer vision



Slide adopted from Alex Smola

Let's talk about the pivotal moment in 2012 when AlexNet won the ImageNet competition.
Before this milestone, the common practice in computer vision was to manually engineer features and use a Support Vector Machine (SVM) for classification.
AlexNet revolutionized this approach by learning features automatically and using a softmax function for classification.
However, AlexNet was more than just a scaled-up version of LeNet.
One significant innovation was the introduction of dropout 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.
Another important change was the use of Rectified Linear Units (ReLU) as activation functions.
ReLU helped mitigate the vanishing gradient problem, enabling training of deeper networks more efficiently.
Max pooling was another change that replaced average pooling in AlexNet.
This made the learned features more shift-invariant, which is important for object recognition.
To secure their win, the AlexNet team employed heavy data augmentation techniques like cropping, shifting, and rotation, along with model ensembling.
The result? A paradigm shift not just in computer vision but it set the stage for advancements in other domains like speech recognition and natural language processing.



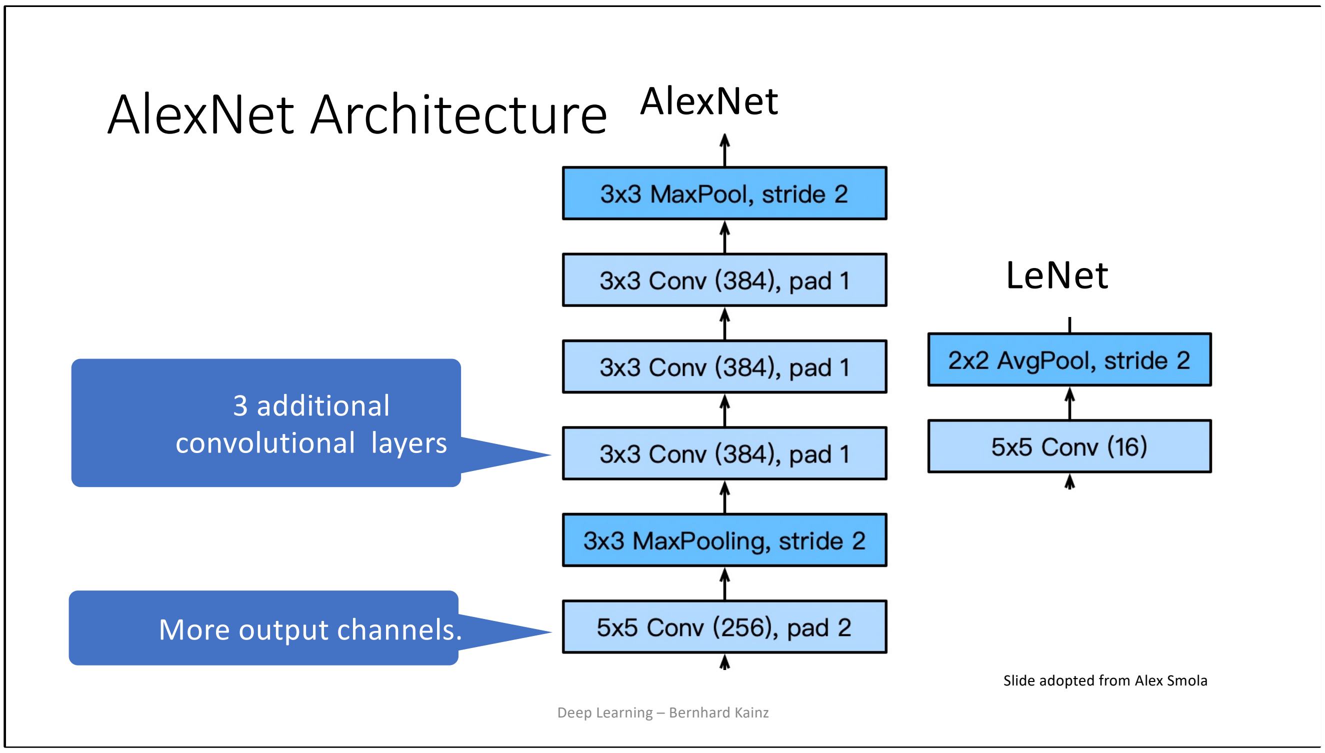
AlexNet Architecture

Larger pool size, change to max pooling

Larger kernel size, stride because of the increased image size, and more output channels.

Slide adopted from Alex Smola

One of the key advancements in AlexNet was the use of larger pool sizes.
This was coupled with a transition from average pooling to max pooling.
Max pooling generally retains the most salient features and discards less useful information, making the model more robust.
Another important modification was in the kernel size and the stride of the convolutional layers.
The increase in kernel size and stride was a design choice to accommodate the higher resolution of images in the ImageNet dataset compared to MNIST.
These changes weren't just arbitrary; they were essential for handling the complexity and the dimensions of the incoming images.
Lastly, we also see an increase in the number of output channels, as mentioned in the previous slide.
More output channels mean the network can learn a greater variety of features, thus boosting its performance.

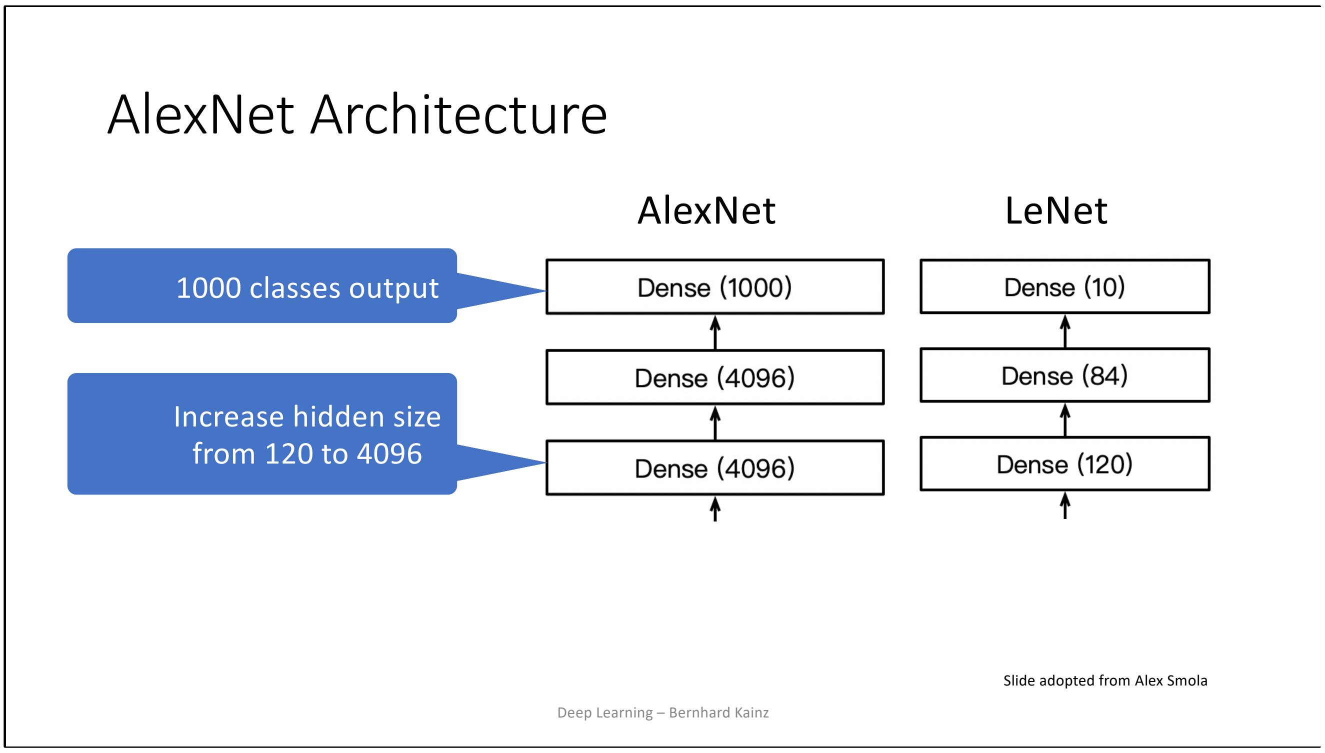


AlexNet Architecture

Slide adopted from Alex Smola

Deep Learning – Bernhard Kainz

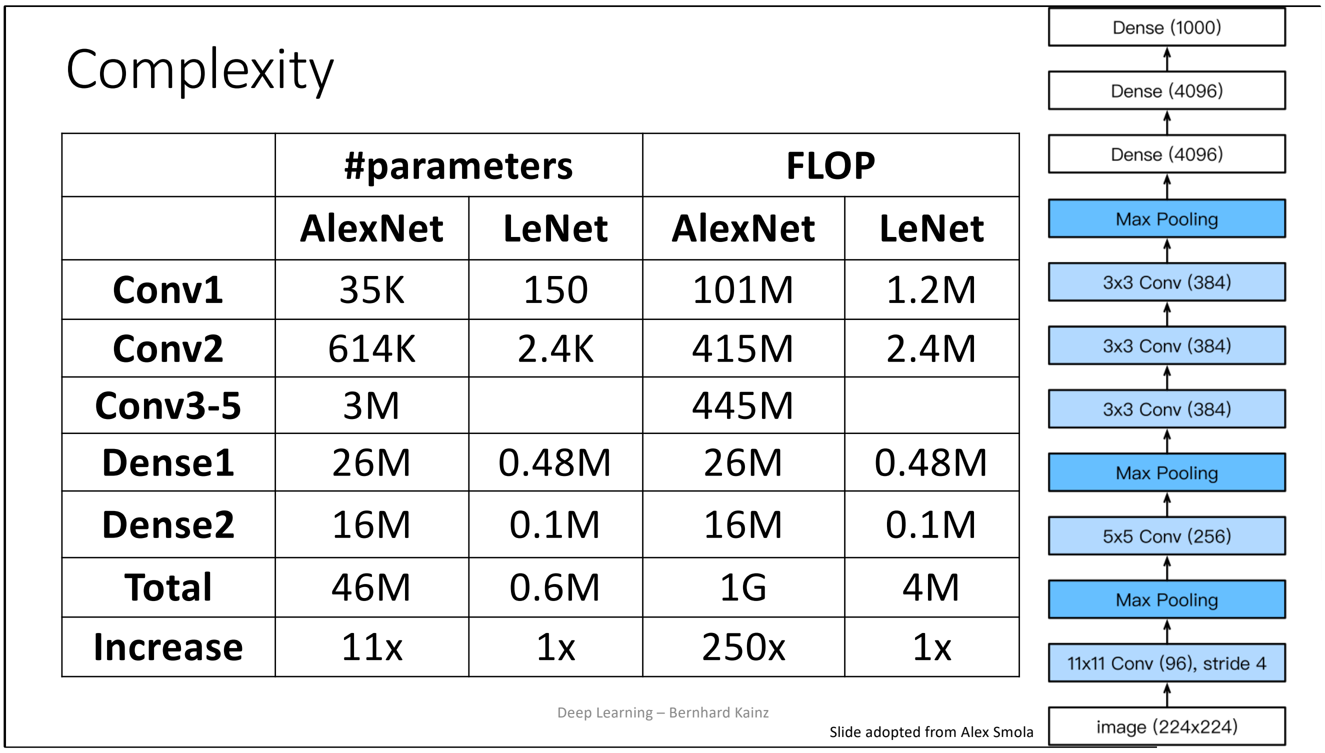
Here are some of the key technical changes that set AlexNet apart from its predecessors.
Firstly, the pool size was increased, and max pooling was adopted.
Max pooling helps to make the network more robust and less sensitive to variations in the input.
Secondly, the kernel size and stride were also adjusted, primarily because of the larger image sizes in ImageNet compared to datasets like MNIST.
This allows the network to capture more complex and varied features from the images.
Lastly, the number of output channels was increased significantly.
More output channels mean that the network can learn more complex and high-level features.
These changes were crucial for handling the increased complexity and size of the ImageNet dataset.



AlexNet Architecture

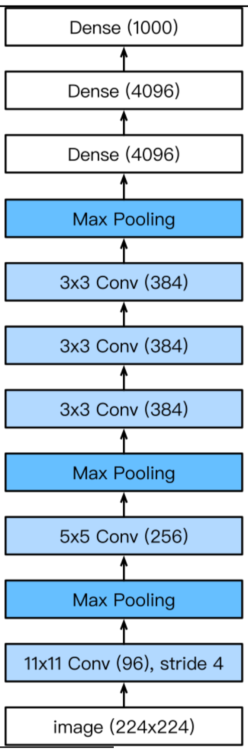
Slide adopted from Alex Smola

The output layer in AlexNet is designed for 1,000 classes.
This is a significant increase compared to LeNet, which was initially designed for a much simpler 10-class problem like MNIST.
The increase in the number of classes necessitates a more complex model capable of distinguishing between a greater variety of features.
Additionally, the hidden layer size was drastically increased from 120 units in LeNet to a whopping 4,096 in AlexNet.
This allows AlexNet to learn a much richer set of features from the input data.
The increase in hidden size adds more capacity to the model, making it more capable of handling the complexity of the ImageNet dataset.
The increased output classes and hidden size make AlexNet more suitable for large-scale, complex tasks, setting a new standard in the field.



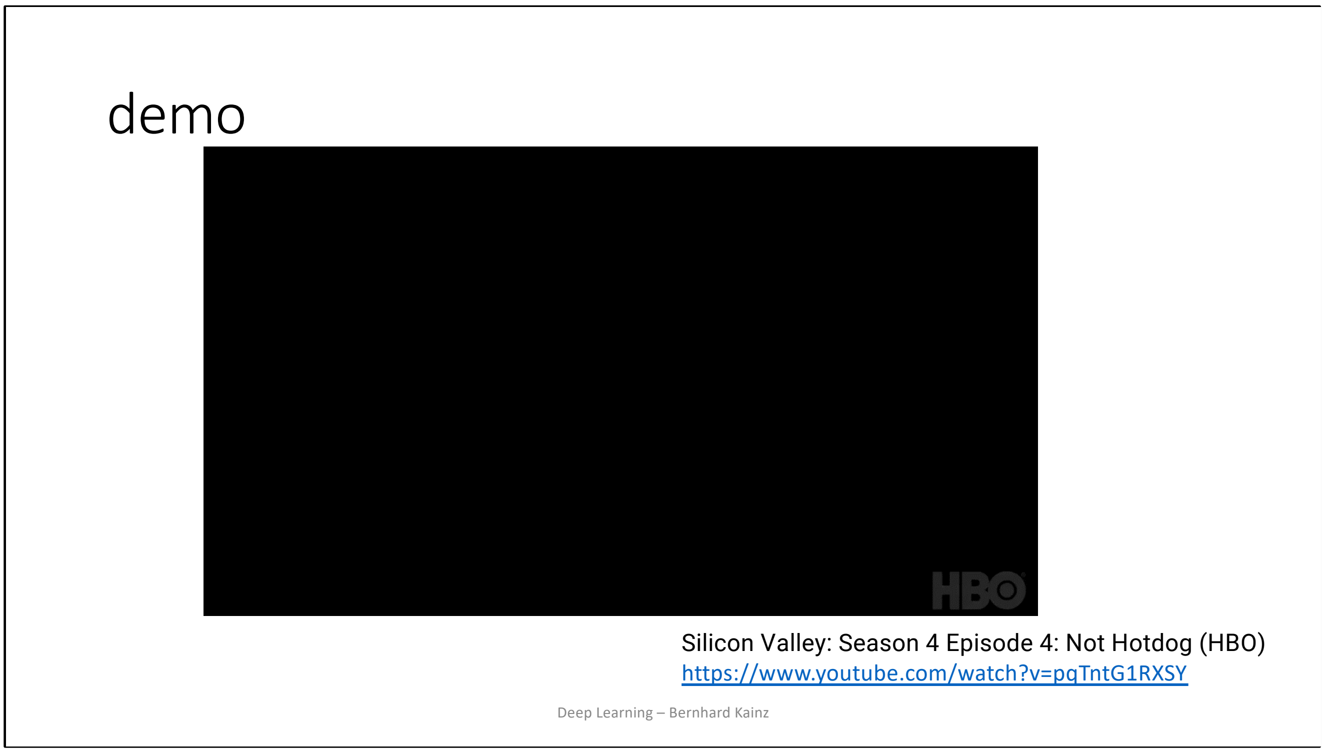
Complexity

	#parameters		FLOP	
	AlexNet	**LeNet**	**AlexNet**	**LeNet**
Conv1	35K	150	101M	1.2M
Conv2	614K	2.4K	415M	2.4M
Conv3-5	3M		445M	
Dense1	26M	0.48M	26M	0.48M
Dense2	16M	0.1M	16M	0.1M
Total	46M	0.6M	1G	4M
Increase	11x	1x	250x	1x



Slide adopted from Alex Smola

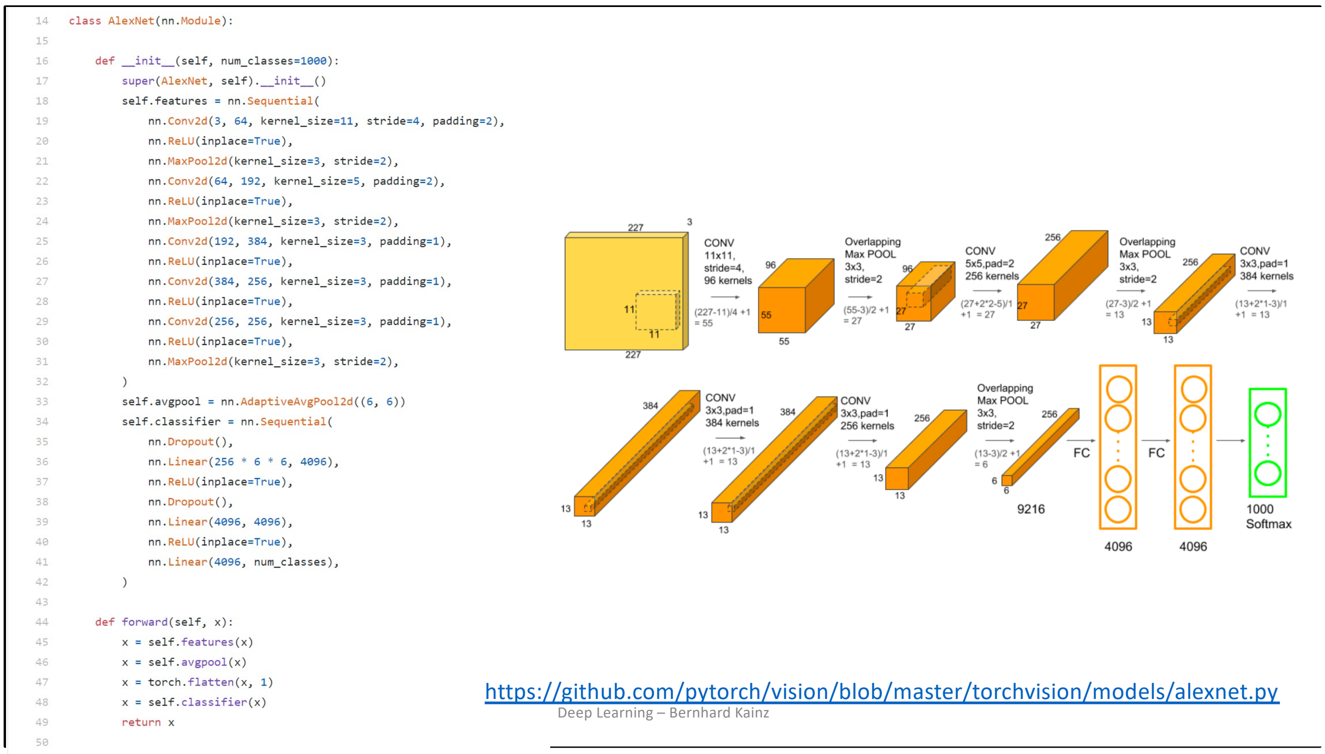
Let's talk about the complexity of AlexNet compared to LeNet-5.
AlexNet is substantially more complex, being about 250 times more computationally expensive.
However, in terms of the number of parameters, it's only about ten times larger than LeNet-5.
This brings us to an interesting point about the trade-off between computation and memory.
AlexNet is notorious for its high memory usage, particularly when it was first introduced.
It’s interesting to note how this trade-off has evolved over the years.
Today, with advancements in hardware, the ratio would probably skew even more towards computation rather than memory.
Modern GPUs can handle complex calculations much more efficiently, allowing for even more computationally intensive models.



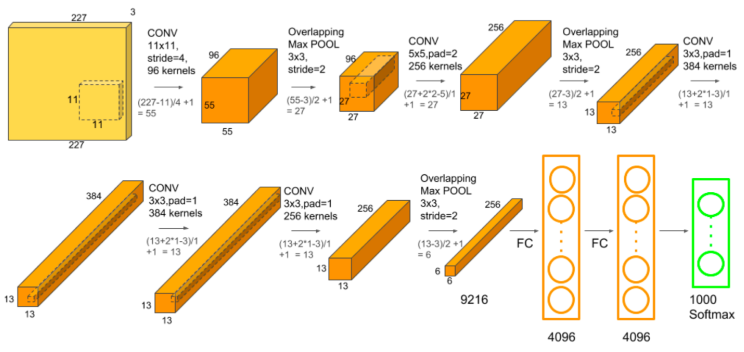
demo

Silicon Valley: Season 4 Episode 4: Not Hotdog (HBO)
https://www.youtube.com/watch?v=pqTntG1RXSY

Let's take a light-hearted turn and talk about how AlexNet made its way into popular culture.
Specifically, I'd like to mention an episode from Season 4 of Silicon Valley, titled "Not Hotdog."
Here is a link to the specific scene on YouTube:
https://www.youtube.com/watch?v=pqTntG1RXSY.
You might be wondering why I'm showing you a sitcom clip instead of an AlexNet demo.
Well, the reason is that an AlexNet demo wouldn't actually be that visually impressive.
What you would see is a slightly worse labeling of the data compared to the ground truth from the test set.
Instead, this sitcom clip gives you an idea of what people thought was suddenly possible with the advent of deep learning and AlexNet.
And yes, the clip is meant to be a joke, but it captures the imagination and excitement around what these neural networks could do.



```python
class AlexNet(nn.Module):

    def __init__(self, num_classes=1000):
        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(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),
            nn.Linear(4096, num_classes),
        )

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

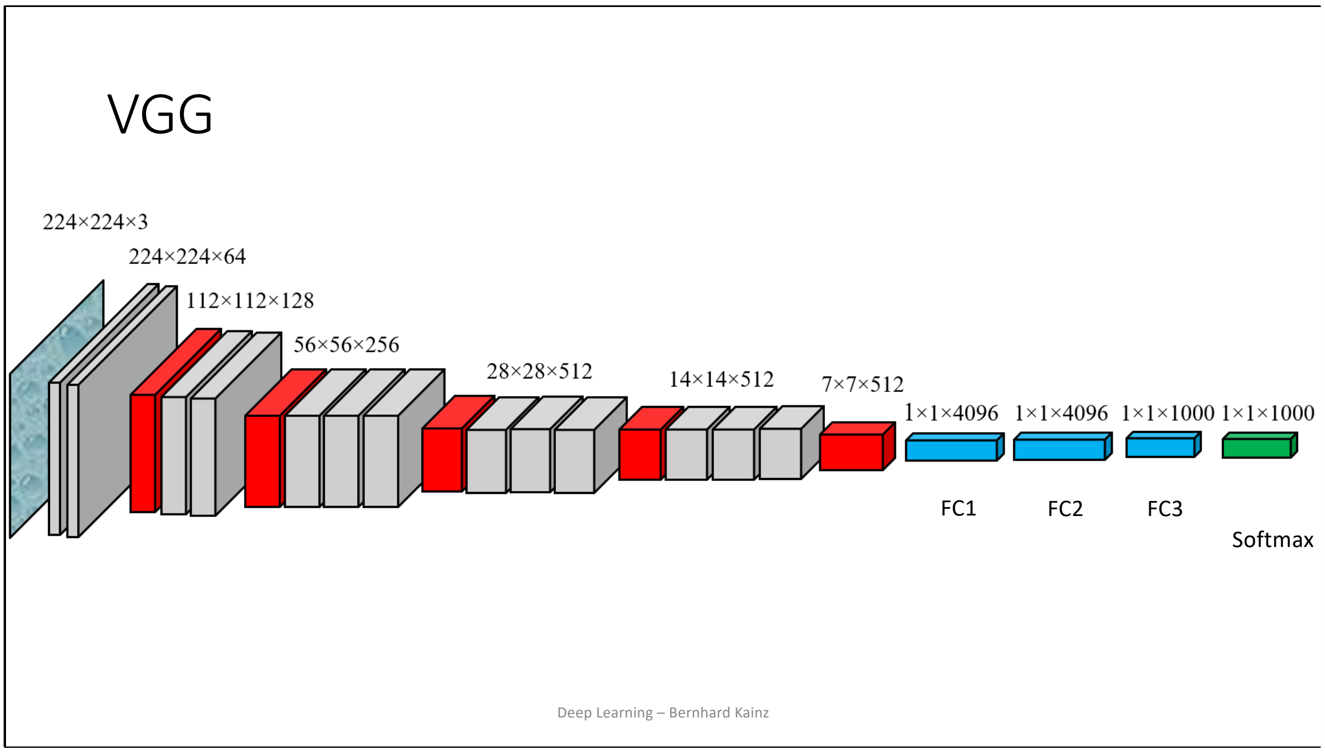


https://github.com/pytorch/vision/blob/master/torchvision/models/alexnet.py

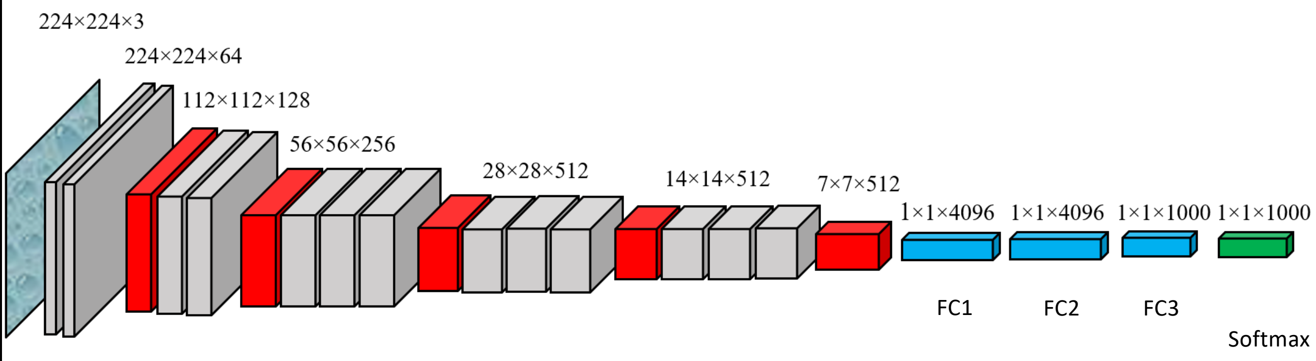
In this slide, we look at how to implement AlexNet using PyTorch.
PyTorch has made it incredibly easy to implement even complex architectures like this.
What you'll see in this slide is a consolidated network diagram.
This is a nod to how far we've come, as there's no longer a need to split the network across two GPUs like in the original AlexNet design.
Let's dive into the code.
You'll see that we are using PyTorch's standard libraries and the code structure should look familiar if you've used PyTorch before.
The architecture follows the original AlexNet blueprint but it's all in one go, thanks to modern hardware capabilities.
Notice how we use ReLU activations and max pooling, staying true to the key features that made AlexNet revolutionary at its time.
We also include the fully connected layers, which are an integral part of the AlexNet design.
That's it for the code walkthrough. It's that straightforward to implement AlexNet in PyTorch nowadays.

VGG

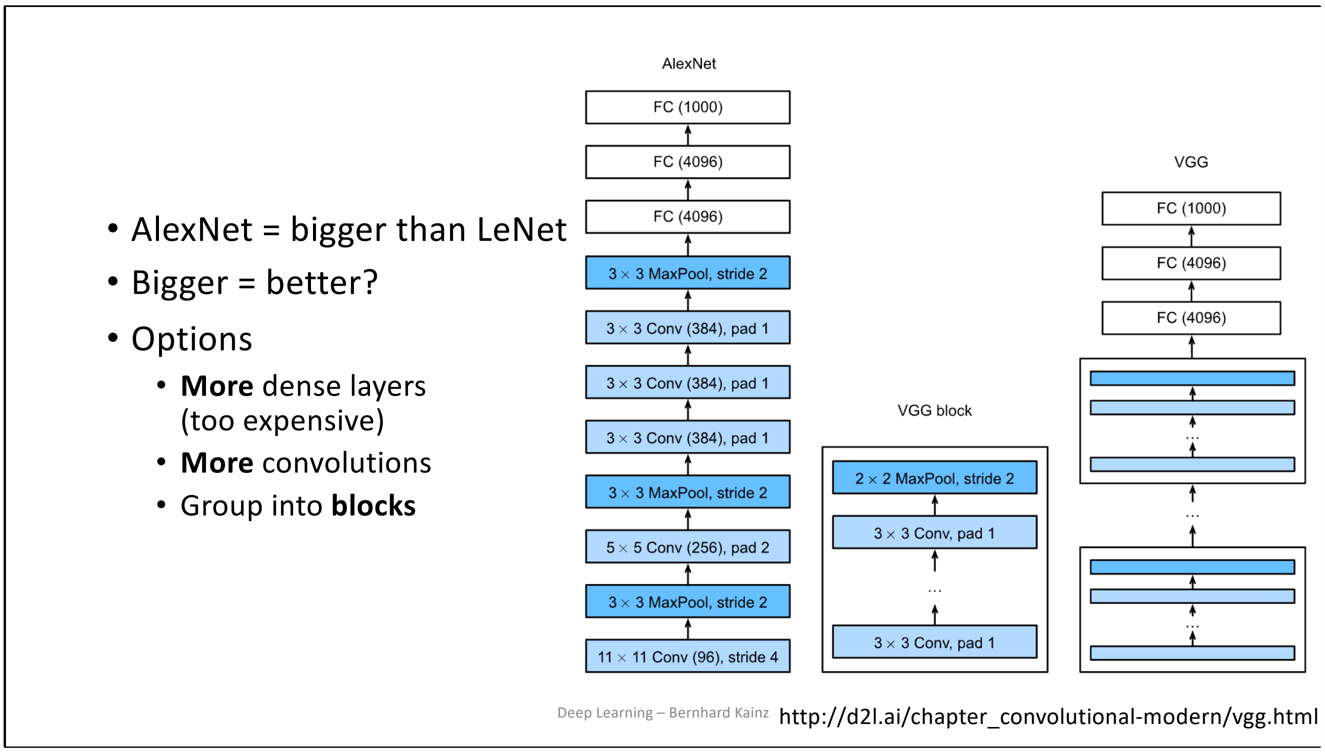
Lecture inspired by Alex Smola with add-ons



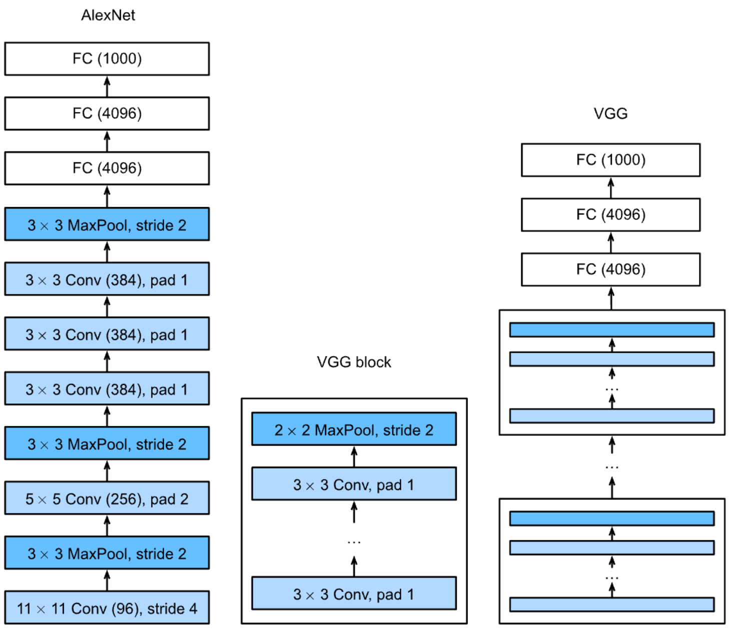
VGG



Now, let's transition to another architecture that you'll likely encounter or use, especially as a backbone for feature extraction.
The next milestone in CNN evolution is VGG, or Visual Geometry Group.
The Visual Geometry Group is actually a research group based at the University of Oxford.
This group took inspiration from AlexNet's "bigger is better" approach and decided to take it a step further.
In doing so, they created an architecture that has been widely used and adopted in various computer vision tasks.
We will dig deeper into what makes VGG unique and why it's often used as a foundational architecture in many projects.



- AlexNet = bigger than LeNet
- Bigger = better?
- Options
  - **More** dense layers (too expensive)
  - **More** convolutions
  - Group into **blocks**

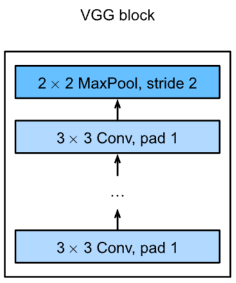


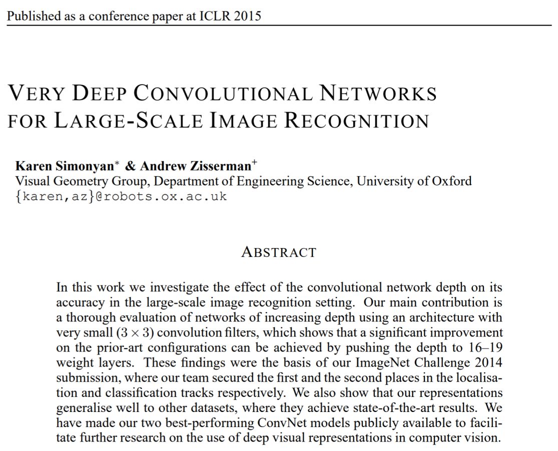
http://d2l.ai/chapter_convolutional-modern/vgg.html

Now let's take a closer look at the VGG architecture and how it compares with AlexNet and LeNet.
One way to make a network larger is to add more dense layers.
However, this approach can quickly become computationally expensive.
Another option is to add more convolutional layers.
You can certainly do that, but as the network grows, specifying each convolution layer individually becomes tedious.
Imagine having to define each layer by hand for a 20, 30, or 40-layer network. It's quite impractical.
This is where VGG's key innovation comes into play—grouping layers into blocks.
These blocks can be easily parameterized, creating a more organized, modular architecture.
This not only simplifies the design process but also makes it easier to fine-tune the model for specific learning tasks.

VGG blocks

- Deeper vs. wider?
  - 13x13?
  - 5x5?
  - 3x3?
  - Deep and narrow = better
- VGG block
  - 3x3 convolutions (pad 1) (n layers, m channels)
  - 2x2 max-pooling (stride 2)




Published as a conference paper at ICLR 2015

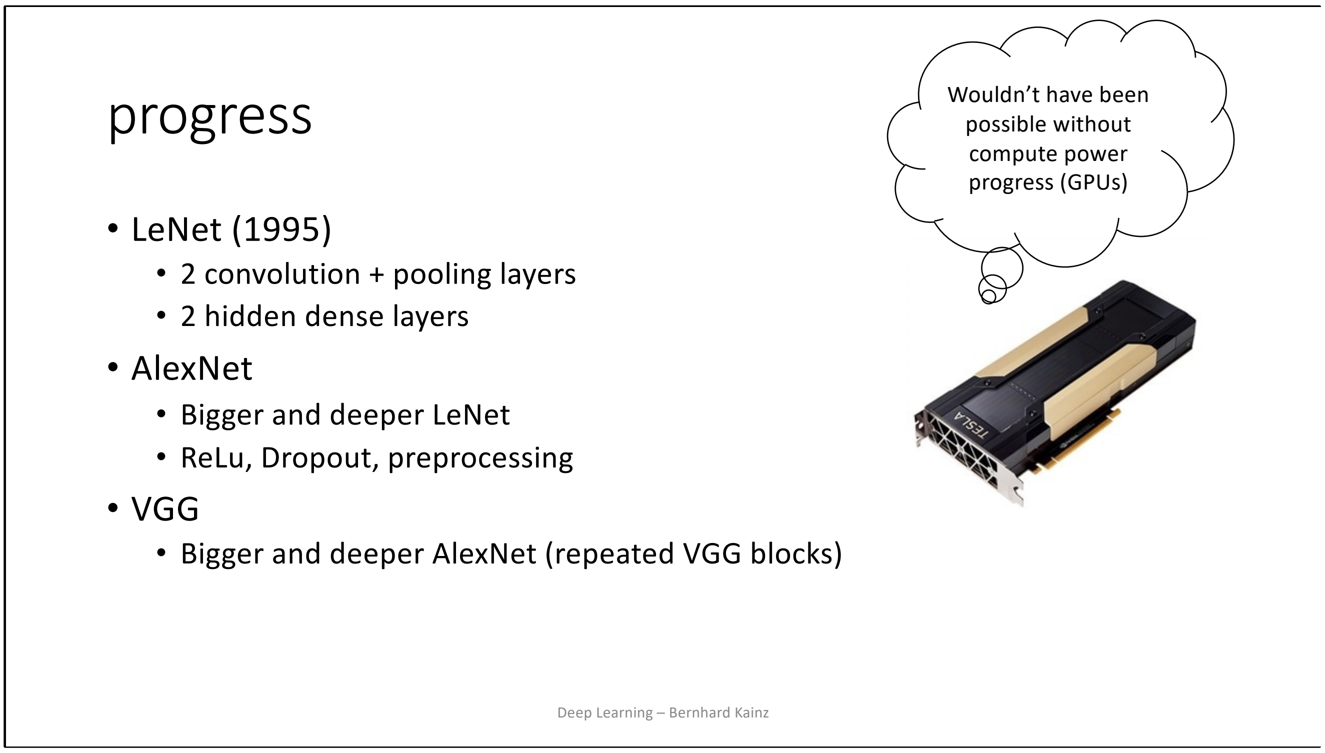
VERY DEEP CONVOLUTIONAL NETWORKS FOR LARGE-SCALE IMAGE RECOGNITION

Karen Simonyan[*] & Andrew Zisserman[+]
Visual Geometry Group, Department of Engineering Science, University of Oxford
{karen,az}@robots.ox.ac.uk

ABSTRACT

In this work we investigate the effect of the convolutional network depth on its accuracy in the large-scale image recognition setting. Our main contribution is a thorough evaluation of networks of increasing depth using an architecture with very small (3 × 3) convolution filters, which shows that a significant improvement on the prior-art configurations can be achieved by pushing the depth to 16–19 weight layers. These findings were the basis of our ImageNet Challenge 2014 submission, where our team secured the first and the second places in the localisation and classification tracks respectively. We also show that our representations generalise well to other datasets, where they achieve state-of-the-art results. We have made our two best-performing ConvNet models publicly available to facilitate further research on the use of deep visual representations in computer vision.

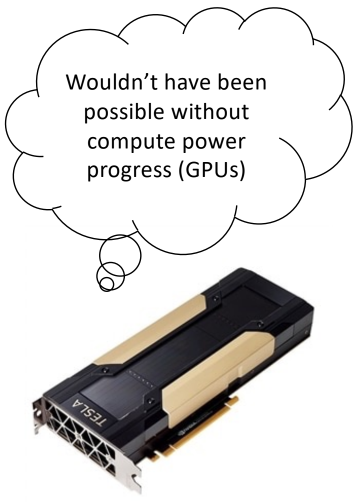
Here is more about the concept of VGG blocks, a foundational element of the VGG architecture.
One key question that had to be answered was whether to use fewer wide convolutions or more narrow ones.
Recent comprehensive analysis from papers has shown that using more layers of narrow convolutions outperforms 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.
So, what does a VGG block consist of?
It typically has several 3x3 convolutions.
If you pad these by one, it maintains the spatial dimensions from the input to the output layer.
At the end of each block, there's a max-pooling layer of 2x2 with a stride of 2, effectively halving the resolution.
Now, imagine stacking several of these blocks together and combining them with dense layers, much like in AlexNet.
Doing this, you can create an entire family of architectures, like VGG-16 or VGG-19, just by varying the number of these blocks.
If we take a step back and look at the overall progress in network architectures, it boils down to making them bigger and deeper.
In the debate between deeper vs. wider networks, it turns out that deep and narrow structures are generally better.



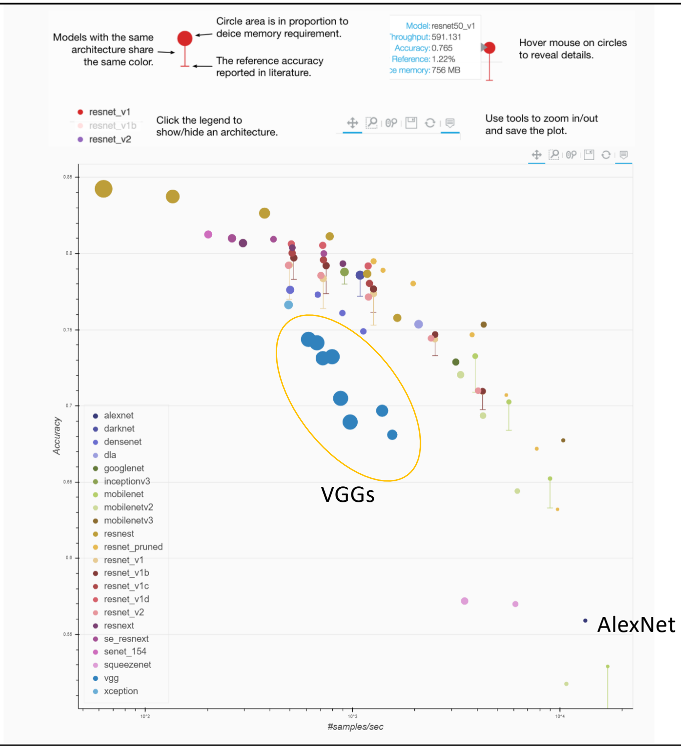
progress

- LeNet (1995)
  - 2 convolution + pooling layers
  - 2 hidden dense layers
- AlexNet
  - Bigger and deeper LeNet
  - ReLu, Dropout, preprocessing
- VGG
  - Bigger and deeper AlexNet (repeated VGG blocks)

Wouldn’t have been possible without compute power progress (GPUs)

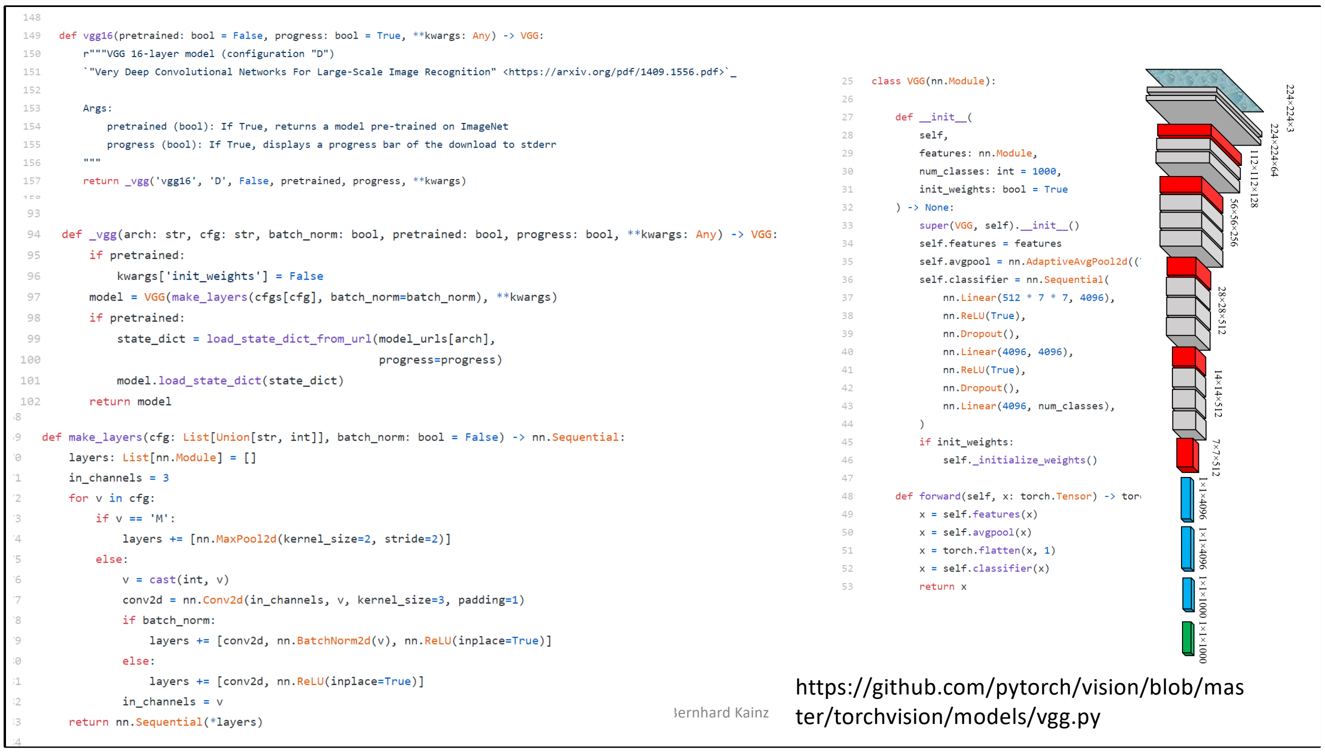


Starting off with LeNet, this was a simpler time with just 2 convolution and pooling layers.
It also featured 2 hidden dense layers.
Fast forward to AlexNet, and you'll notice that it essentially took LeNet and made it bigger and deeper.
But it wasn't just a scaled-up LeNet; it also introduced innovations like ReLU activations, dropout for regularization, and preprocessing techniques.
Then comes VGG, which took the concept of 'bigger and deeper' from AlexNet and ran with it.
VGG introduced the notion of repeated blocks, an architecture style that was both elegant and highly effective.
It's important to note that the leap from LeNet to AlexNet and then to VGG would not have been possible without significant progress in computational power, particularly the use of GPUs.
So the takeaway here is that as computational power has grown, so has the complexity and effectiveness of our network architectures.



https://cv.gluon.ai/model_zoo/classification.html

Let's take a moment to delve into this chart that plots throughput against accuracy for various CNN architectures.
As you can see, VGG tends to be a lot slower when it comes to throughput compared to AlexNet.
However, what it lacks in speed, it more than makes up for in terms of accuracy.
This tells us that while VGG might require more computational resources, it generally provides superior performance.
If you're interested in further comparison, you can check out the link at the bottom which provides a model zoo detailing the performance of various architectures.
Interestingly, after the era of VGG, the trend shifted towards optimizing for both smaller network sizes and higher accuracy.
This indicates that the field is continually evolving to find the best balance between efficiency and effectiveness.

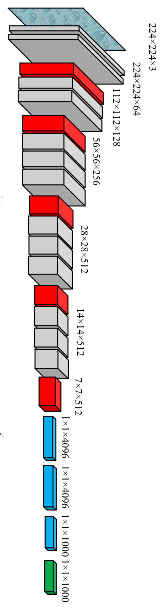


```python
def vgg16(pretrained: bool = False, progress: bool = True, **kwargs: Any) -> VGG:
    r"""VGG 16-layer model (configuration "D")
    `"Very Deep Convolutional Networks For Large-Scale Image Recognition" <https://arxiv.org/pdf/1409.1556.pdf>`_

    Args:
        pretrained (bool): If True, returns a model pre-trained on ImageNet
        progress (bool): If True, displays a progress bar of the download to stderr
    """
    return _vgg('vgg16', 'D', False, pretrained, progress, **kwargs)
```

```python
def _vgg(arch: str, cfg: str, batch_norm: bool, pretrained: bool, progress: bool, **kwargs: Any) -> VGG:
    if pretrained:
        kwargs['init_weights'] = False
    model = VGG(make_layers(cfgs[cfg], batch_norm=batch_norm), **kwargs)
    if pretrained:
        state_dict = load_state_dict_from_url(model_urls[arch],
                                              progress=progress)
        model.load_state_dict(state_dict)
    return model
```

```python
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)
```

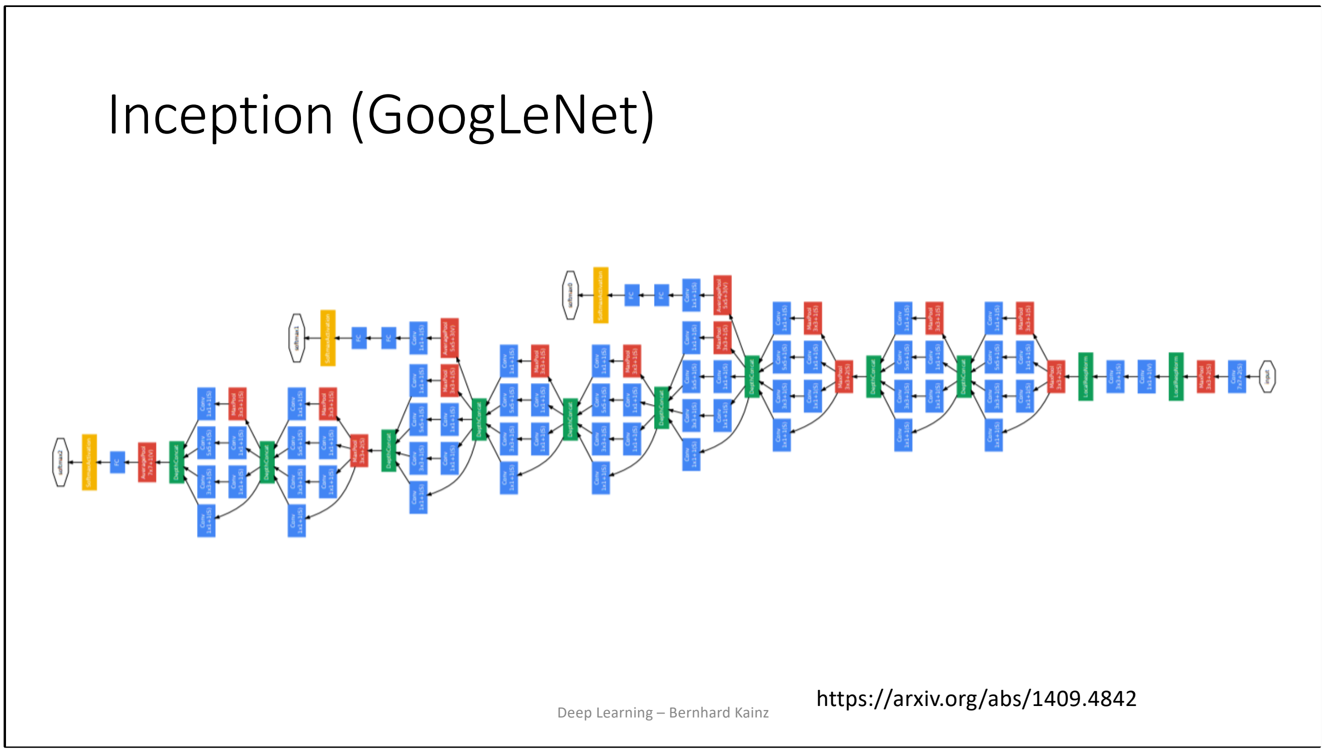
```python
class VGG(nn.Module):

    def __init__(
        self,
        features: nn.Module,
        num_classes: int = 1000,
        init_weights: bool = True
    ) -> None:
        super(VGG, self).__init__()
        self.features = features
        self.avgpool = nn.AdaptiveAvgPool2d((
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )
        if init_weights:
            self._initialize_weights()

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



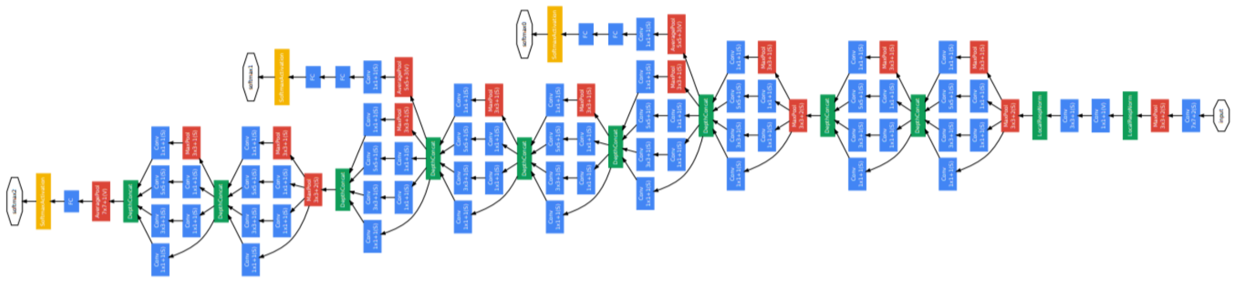
https://github.com/pytorch/vision/blob/master/torchvision/models/vgg.py

Bernhard Kainz

This code provides an implementation of the VGG architecture using PyTorch.
Starting off, the code imports the necessary modules like PyTorch, as well as some utility functions and classes.
The VGG class inherits from nn.Module, which is the base class for all neural network modules in PyTorch.
In the __init__ method, the class defines its layers: features for convolutional layers, avgpool for pooling, and classifier for fully connected layers.
The code also includes an option to initialize weights using Kaiming initialization, which is beneficial for ReLU activation functions.
The forward method outlines how data flows through this network.
The make_layers function generates a sequence of layers based on a given configuration, adding either max-pooling or convolutional layers.
The cfgs dictionary provides pre-defined configurations for various VGG models, specifying the number and types of layers.
The _vgg function constructs a VGG model by taking a configuration string and other optional parameters like batch normalization.
Meta information about the model's weights is also included, specifying where they can be downloaded and their corresponding accuracies on ImageNet.
The script includes several variants of VGG like VGG16, VGG19, both with and without batch normalization, as indicated by the suffix _BN.
At the end of the code, it seems like there's some snippet for registering weights for a VGG model with Batch Normalization.

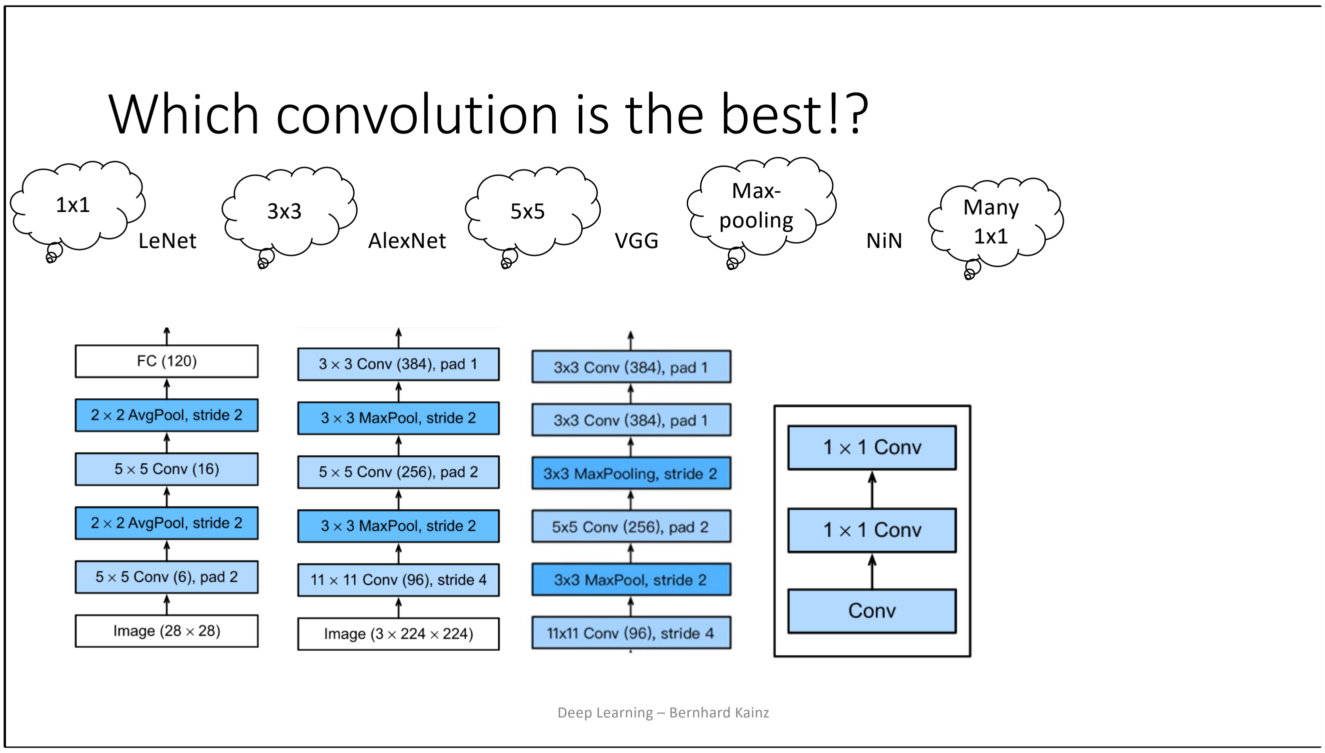


Inception (GoogLeNet)



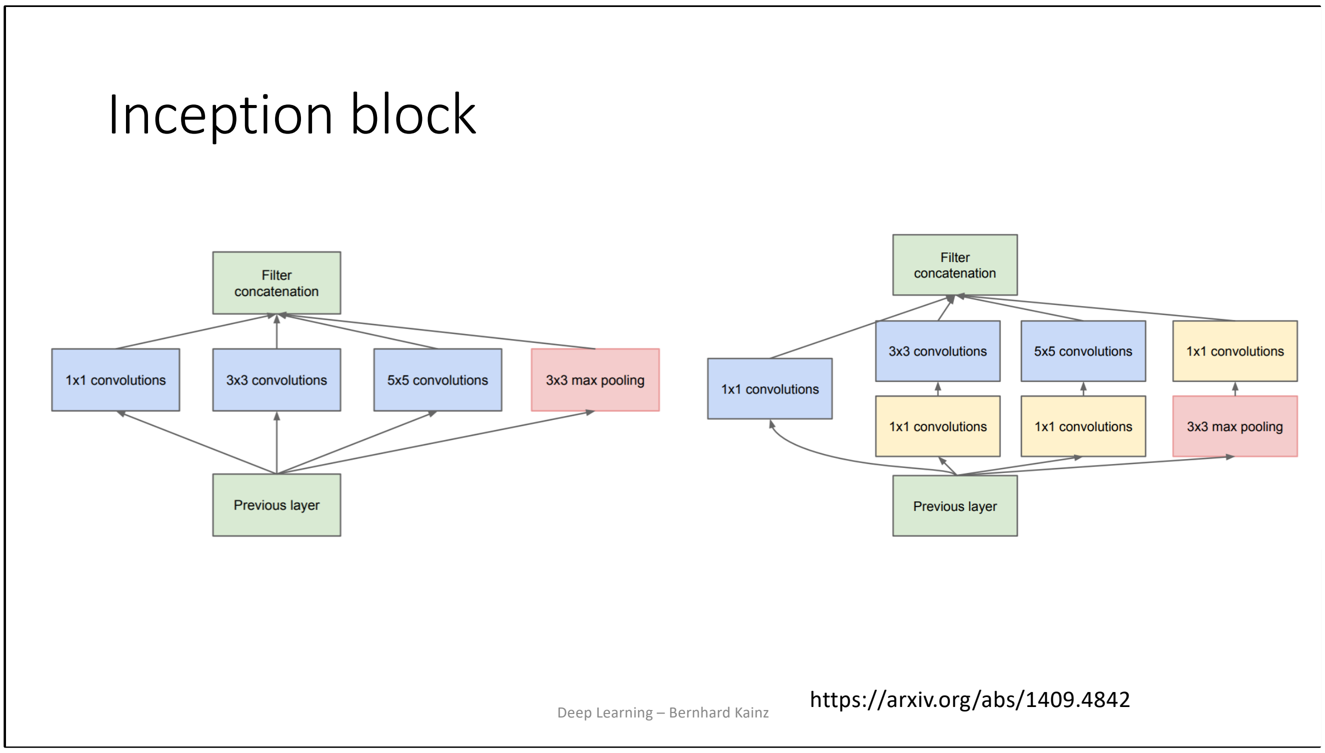
https://arxiv.org/abs/1409.4842

Another stepping stone was the Inception architecture, another landmark in the history of CNNs.
This revolutionary architecture was designed by researchers at Google.
What makes Inception unique is that it's both deep and introduces the concept of parallel paths within the network for the first time.
Unlike previous architectures that had a single path for data to flow through layers, Inception offers multiple pathways.
The advantage of these parallel paths is to capture different types of features more effectively.
The architecture combines the best of different types of convolutions and pooling layers to enhance its performance.
It's this unique parallel structure that has made Inception a go-to architecture for many complex tasks.
As we delve deeper into how these parallel paths work and why they make Inception so effective.
In summary, Inception marks a paradigm shift by introducing parallelism inside a deep network, further improving performance and feature representation.
Thank you for listening, and let's explore the Inception architecture in more detail in the next slides.

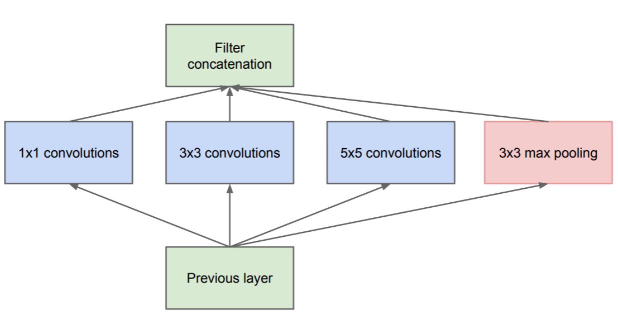


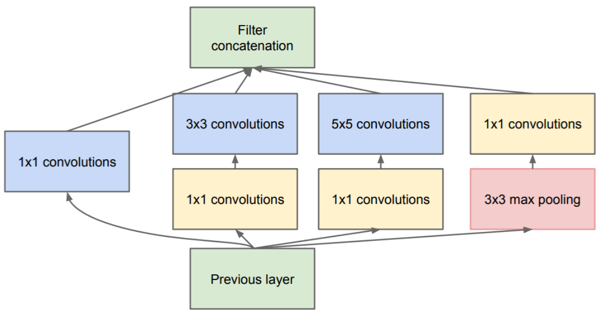
Which convolution is the best!?

Before we dive into the Inception architecture, let's take a moment to recap what we've covered so far.
On the left, we've explored LeNet, which utilizes 5x5 convolutions.
Then we moved onto AlexNet, which employs a mixture of 11x11, 3x3, and 5x5 convolutions.
Following that, we discussed VGG, which primarily focuses on using 1x1 convolutions.
You might be thinking, "This is quite a mess, isn't it?"
Exactly, it becomes confusing to decide which type of convolution should be used for optimal performance.
Should we use 1x1, 3x3, 5x5 convolutions? Or maybe even max pooling? It's hard to make a definitive choice.
This was the dilemma faced by researchers in what we might call the "deep learning stone age" as they tried to construct effective convolutional blocks.
For instance, if you opt for 5x5 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 1x1 convolutions, you'll have a more controlled, memory-efficient architecture, but it may not perform as well.
So, what's the solution to this convolution conundrum?
Well, that's where the Inception architecture comes in, aiming to bring the best of all worlds.



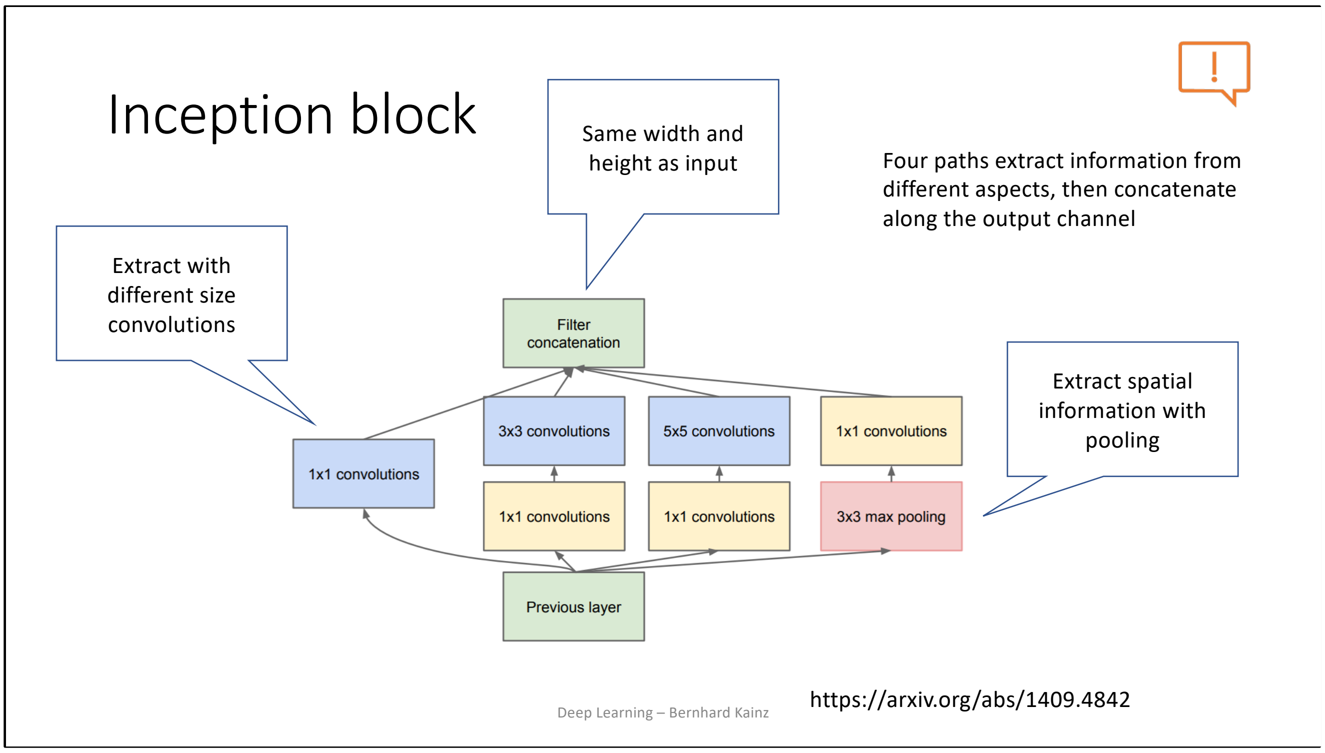
Inception block





https://arxiv.org/abs/1409.4842

So, what was the groundbreaking solution to the convolution dilemma we discussed earlier?
Simply put, the idea was not to decide on one type of convolution over another.
This novel approach gave birth to what we now call the Inception network, named after the famous movie with the concept of "we need to go deeper."
That phrase literally inspired the naming and the architecture of this network.
Let's examine the Inception block, which is essentially a "let's try everything" approach to the problem.
It includes 1x1 convolutions—because why not?
It also has a sequence of 1x1 convolutions followed by 3x3 convolutions.
And for good measure, it adds 1x1 followed by 5x5 convolutions.
Even max pooling followed by 1x1 convolutions find their way into the Inception block.
The idea here is to throw everything into the mix, and hopefully, something will provide the optimal solution.
On the left side of the slide, you'll see a simpler version of the Inception block, and on the right is a more complex version that also incorporates 1x1 convolutions.
The Inception architecture is a clever way to integrate various types of convolutions to cover all bases.



Inception block

Same width and height as input

Four paths extract information from different aspects, then concatenate along the output channel

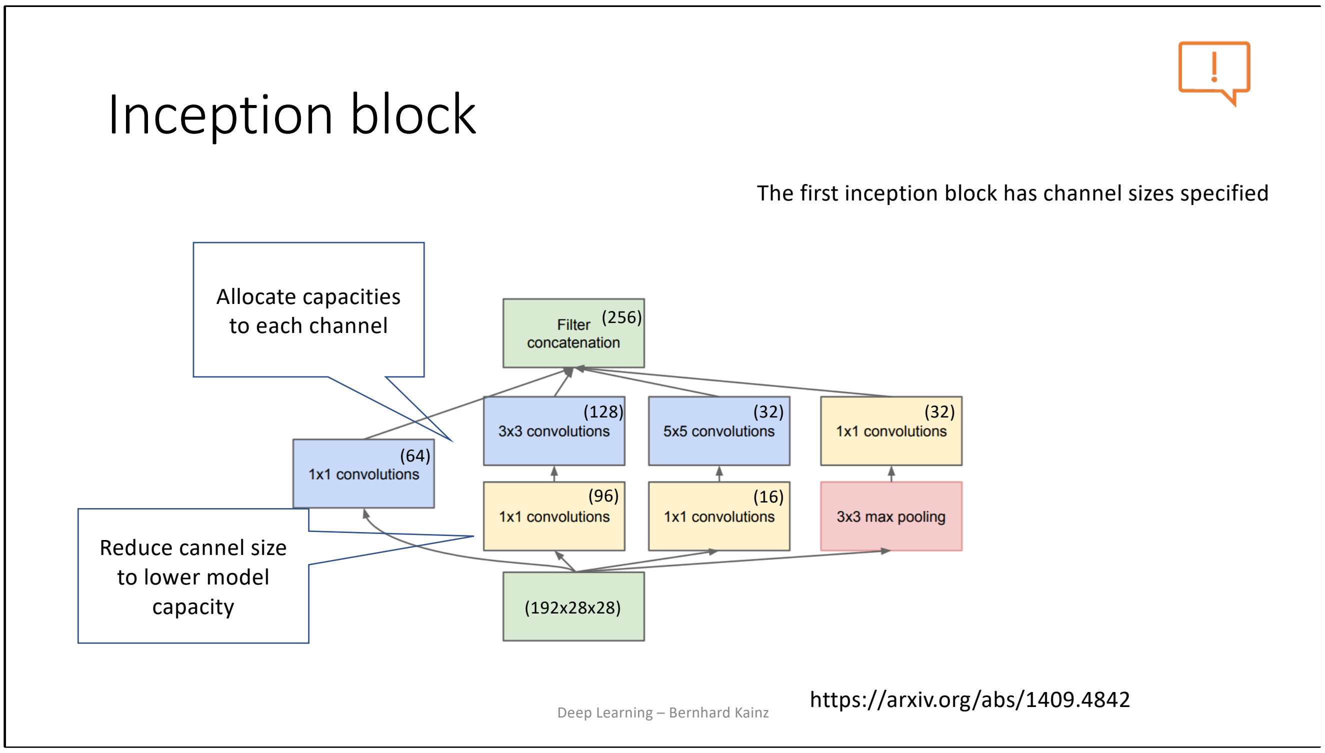
Extract with different size convolutions

Extract spatial information with pooling

Deep Learning – Bernhard Kainz

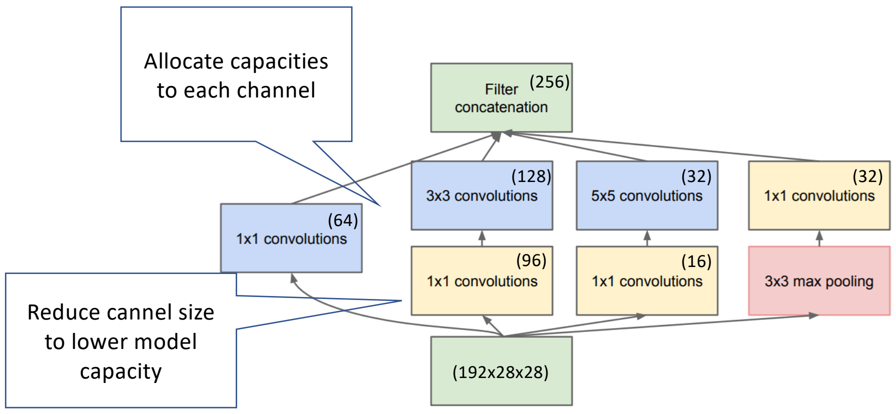
https://arxiv.org/abs/1409.4842

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 3x3 convolutions use half-padding by one, and the 5x5 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.
So now, you essentially have different channels within the architecture focused on different tasks.
One channel might excel at recognizing cats, while another might be better suited for identifying birds.
This diverse approach is what makes the Inception block so interesting and effective.
You might still be asking, "Why is this specific combination of layers the right one?"
The answer is, the researchers probably experimented with many variations, and this configuration yielded the best results.
For our purposes today, let's just say it's the result of a lot of trial and error.
And that's the essence of the Inception block—combining different types of convolutions in a single, cohesive unit.



Inception block

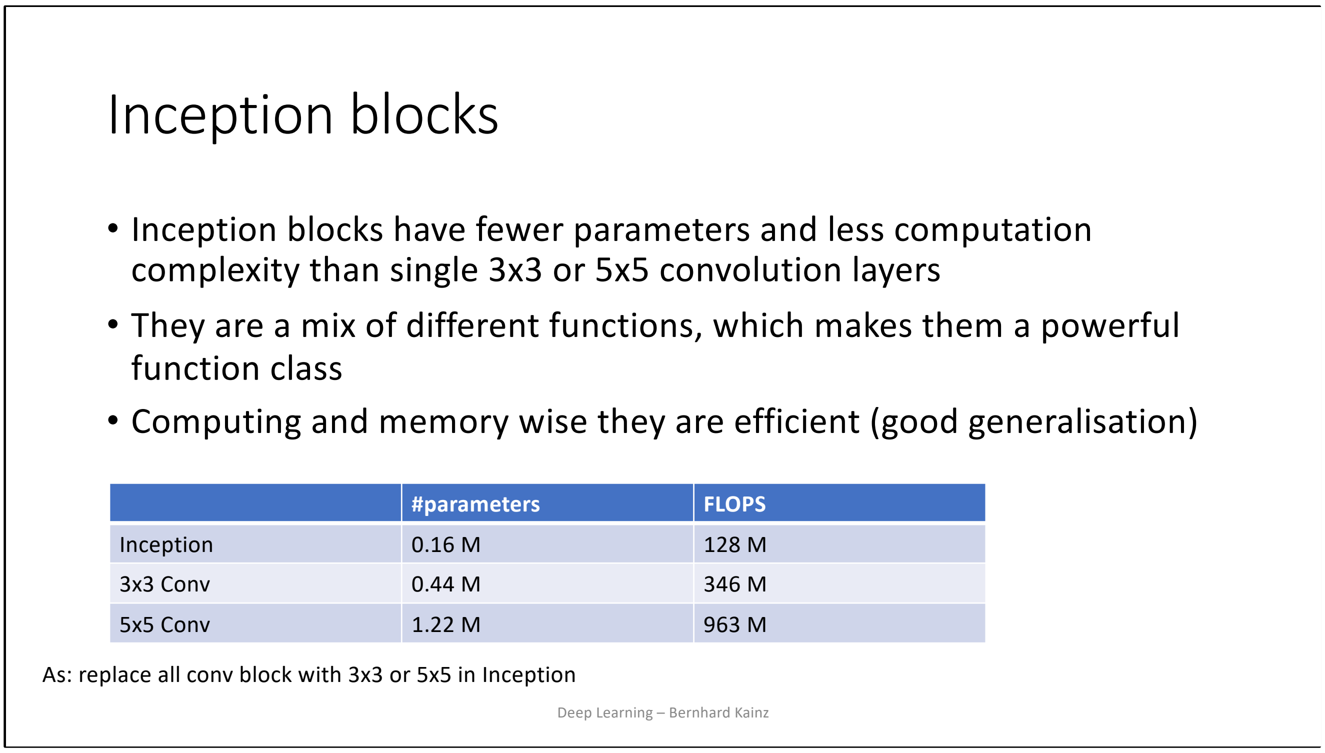
The first inception block has channel sizes specified



Deep Learning – Bernhard Kainz

https://arxiv.org/abs/1409.4842

Let's dive into some of the specifics of the Inception block, such as channel numbers.
The first Inception block uses 64 channels for the 1x1 convolutions.
For the 3x3 convolutions, it uses 128 channels.
And for the 5x5 convolutions, 32 channels are used, primarily because 5x5 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.
So, while it may seem complex, the underlying principle isn't terribly deep or mysterious.
The input number of channels doesn't particularly matter in the grand scheme of the architecture.
What's essential is that different features are fed through different channels, and you get an appropriate number of output channels.
This flexibility is one of the strengths of the Inception architecture, allowing it to adapt to various types of data and tasks.
And that's a quick look at the nuts and bolts of an Inception block.



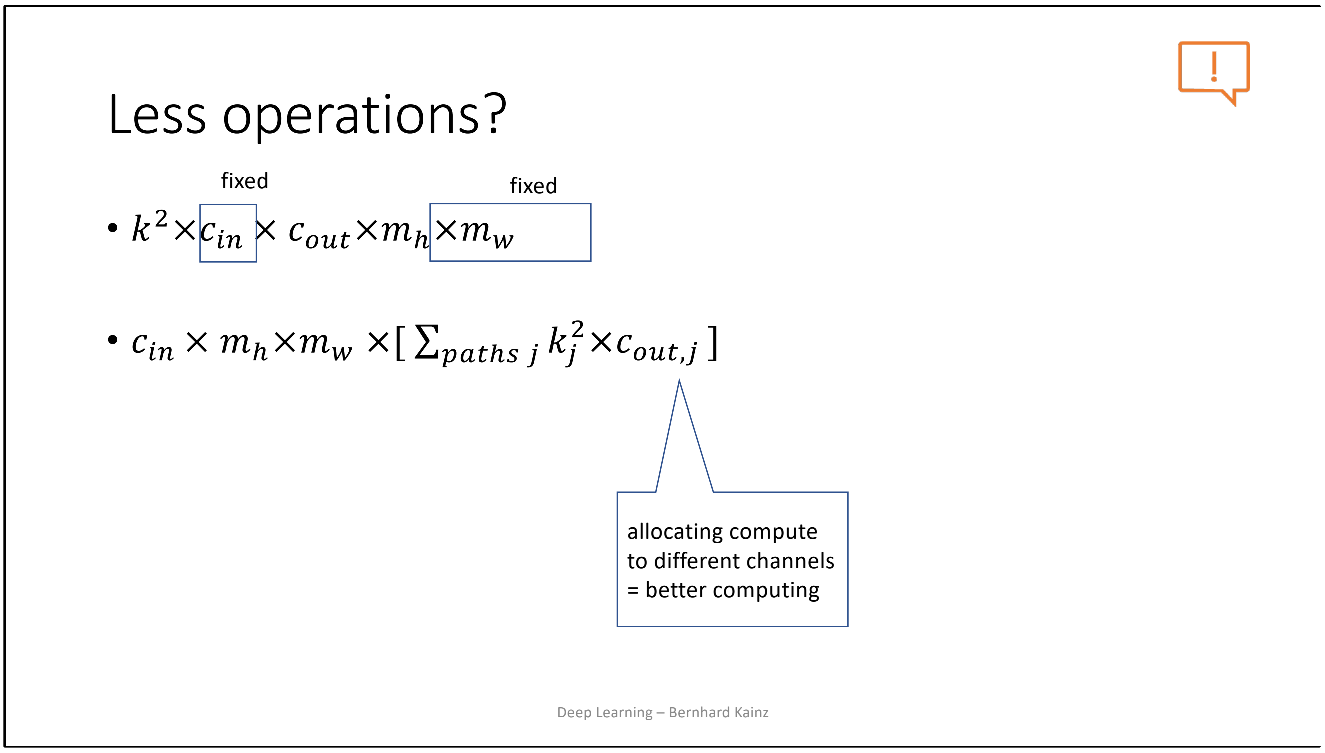
Inception blocks

- Inception blocks have fewer parameters and less computation complexity than single 3x3 or 5x5 convolution layers
- They are a mix of different functions, which makes them a powerful function class
- Computing and memory wise they are efficient (good generalisation)

	#parameters	FLOPS
Inception	0.16 M	128 M
3x3 Conv	0.44 M	346 M
5x5 Conv	1.22 M	963 M

As: replace all conv block with 3x3 or 5x5 in Inception

Let's talk about one of the most critical advantages of the Inception architecture: efficiency.
The Inception block is designed to use a relatively low number of parameters and floating-point operations (FLOPs), without sacrificing performance.
This is a key benefit.
For example, if you need 256 output channels, using an Inception block would require only 160,000 parameters and 128 Mega FLOPs.
Contrast this with using only 3x3 convolutions, which would cost about three times as many parameters and FLOPs.
And if you were to use only 5x5 convolutions, it would cost around eight times more, making the Inception block far more efficient.
The underlying idea is that if you can accomplish the same task with fewer parameters and operations, the network should perform better.
This efficiency is what primarily motivated the design of the Inception block.
So, why is the Inception block simpler in terms of computational requirements?
To answer that, we'd have to delve into the algebra behind it, which elegantly balances complexity and efficiency.
So, the Inception architecture isn't just about throwing everything at the wall to see what sticks; it's also about doing so in a computationally efficient manner.



Less operations?

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

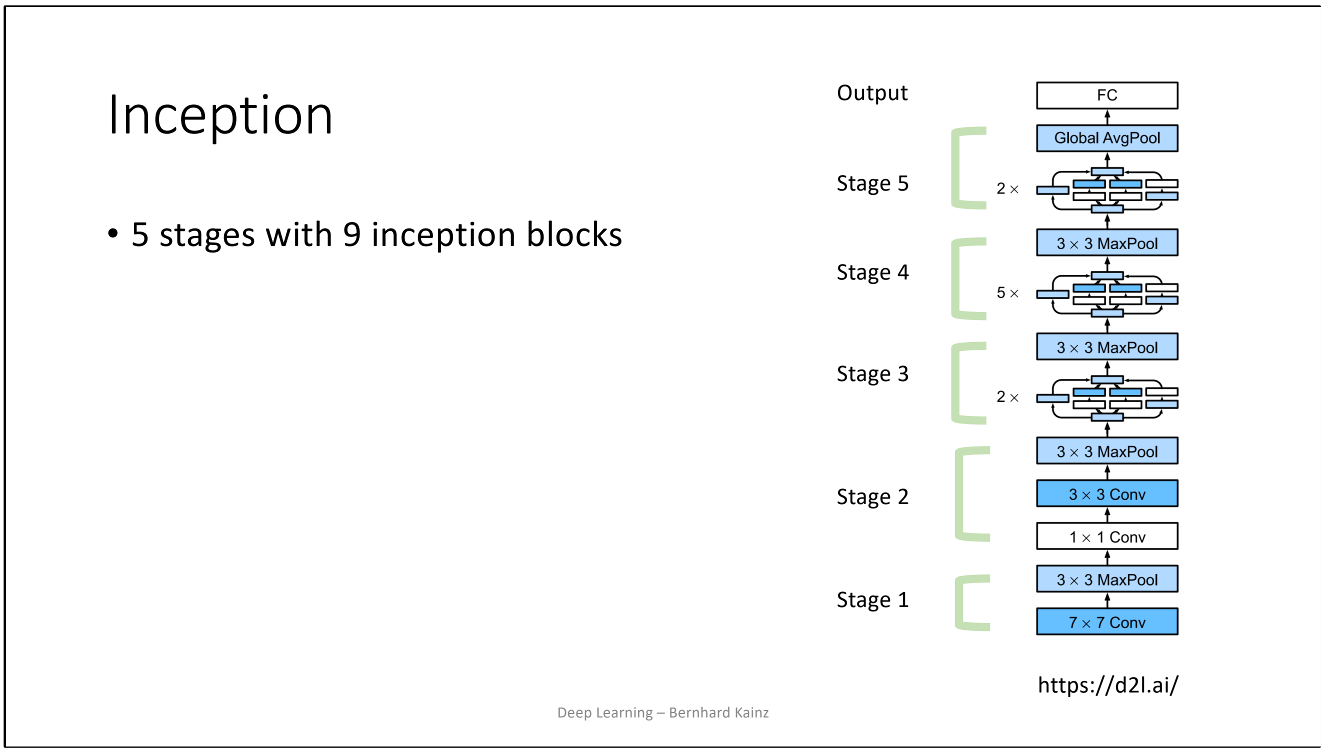
fixed fixed

- $c_{in} \times m_h \times m_w \times [\sum_{paths\ j} k_j^2 \times c_{out,j}]$

allocating compute to different channels = better computing

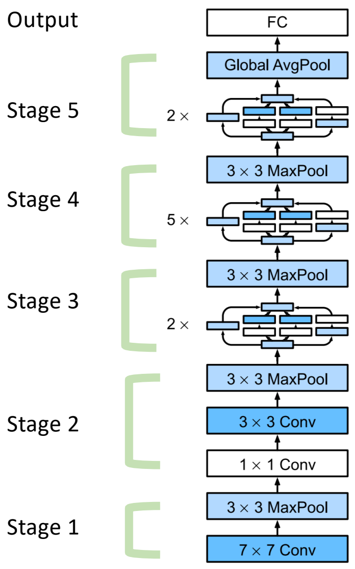
Deep Learning – Bernhard Kainz

Now let's delve into the mathematics behind the Inception architecture, which explains its efficiency and effectiveness.
We start by looking at a specific layer with a convolutional kernel of size *K×K, for example, 3x3 or 5x5.*
You can express this layer's computational cost as K2×Cin×Cout, where Cin and Cout are the number of input and output channels, respectively.
We also have to consider the dimensions of the input image, represented as M for height and W for width.
Notice that M and W are fixed, but we have the flexibility to adjust Cin and outCout.
The total computational cost then becomes Cin×M×W multiplied by a sum over the different paths j, each with its own Kj2×Cout,j.
By carefully allocating resources -- varying the number of channels and kernel sizes -- we can optimize the network's performance.
In other words, the Inception block allows for a judicious allocation of computational resources across different paths, offering a more efficient solution than using a single type of convolution throughout the architecture.
This is where the Inception architecture shines: it combines the benefits of various convolutions and pooling operations while optimizing computational cost.



Inception

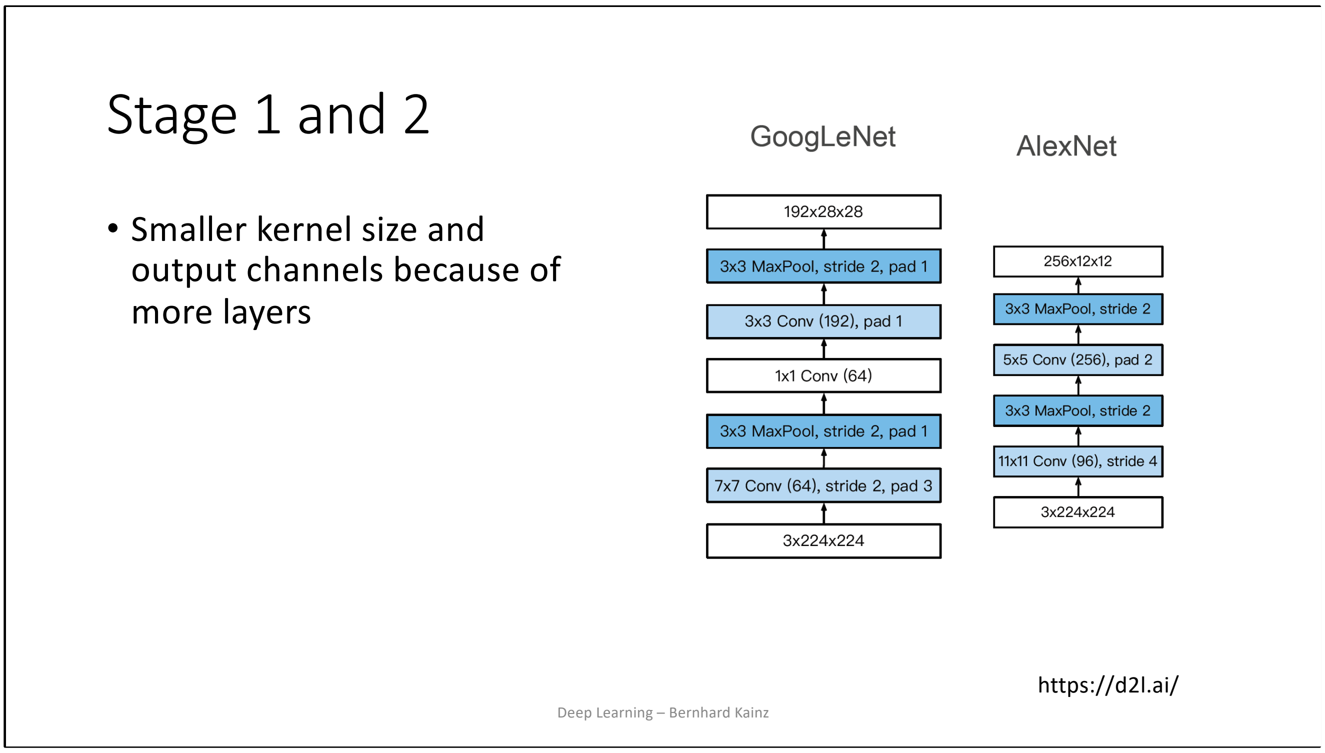
- 5 stages with 9 inception blocks



https://d2l.ai/

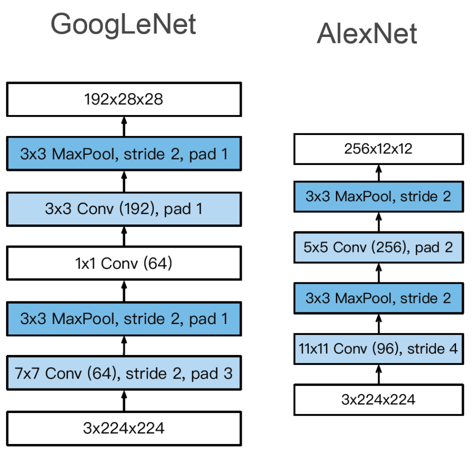
Let's now look at the full structure of the Inception network, often represented in stages for easier understanding.
The diagram you see is from the Deep Learning for Text and Sequences (d2l) book, which provides a more intuitive visualization compared to the original paper.
The network is divided into five main stages.
Starting with stage one, it's fairly standard and similar to other convolutional neural networks we've seen.
It begins with broad convolutions and pooling operations to ensure basic translation invariance and quickly reduce dimensionality.
The use of max pooling in this stage effectively halves the resolution of the input.
Stage two also focuses on capturing spatial correlations and ends with a pooling operation—again, pretty standard stuff.
Now, things get interesting in stage three, where we introduce the Inception blocks.
Here, you'll find two Inception blocks followed by max pooling, which again halves the resolution.
What's critical to note here is that as we reduce the resolution, we increase the number of channels, capturing higher-order features from the input.
Following this, we have a sequence of five more Inception blocks, where most of the interesting nonlinear computations take place.
So why this specific arrangement of Inception blocks?
The answer is empirical; the authors probably experimented with various configurations

and found this one to perform the best.
In subsequent versions and follow-ups like AmoebaNet, more advanced methods like genetic algorithms have been used to automatically search for the most effective architecture, provided that computational resources are not a constraint.



Stage 1 and 2

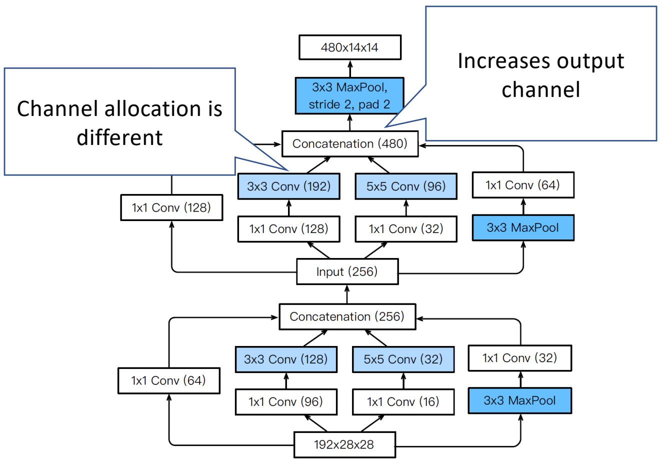
- Smaller kernel size and output channels because of more layers



https://d2l.ai/

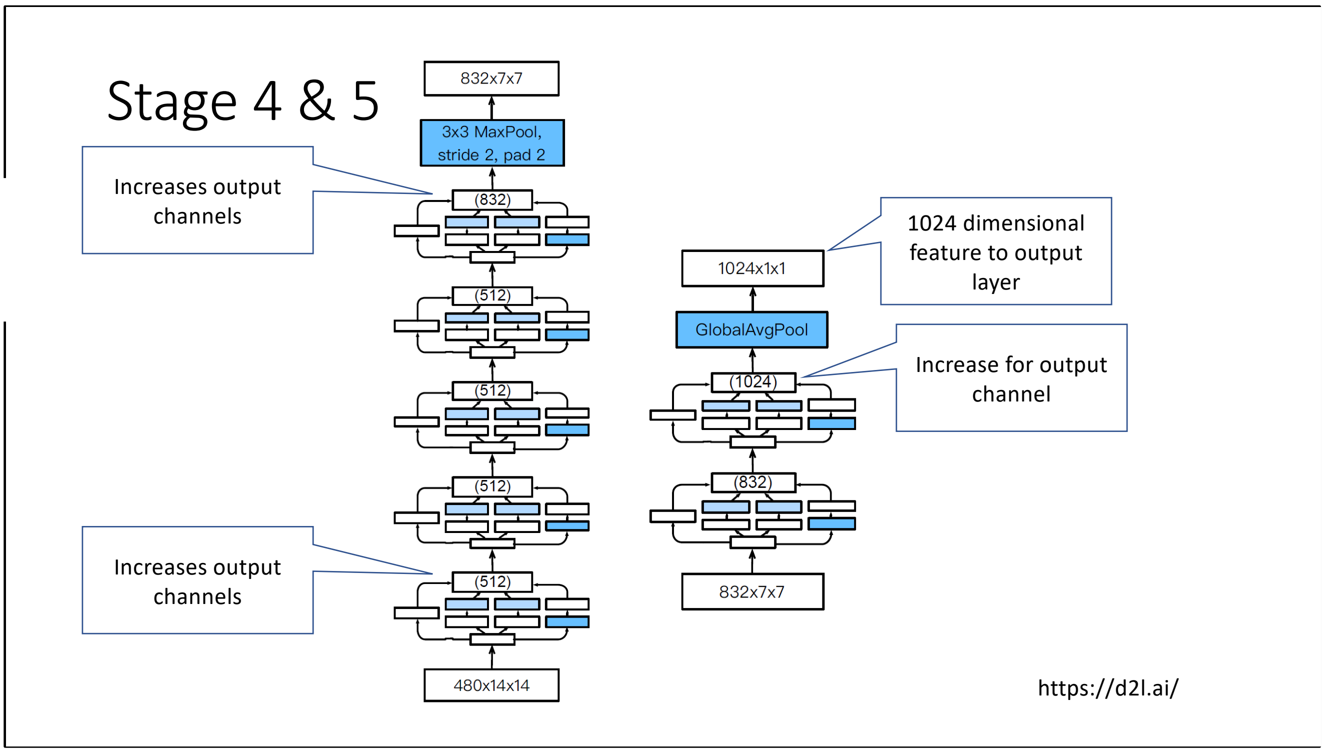
Now let's focus on stages one and two of the Inception architecture and compare them to AlexNet.
In AlexNet, the first convolution is quite wide, at 11x11.
In contrast, Inception starts with a 7x7 convolution, followed by 4 instances of 1x1 convolutions and then a 3x3 convolution, along with pooling.
So, as you can see, it's not vastly different from what AlexNet does in its initial stages.
The key difference here lies in the number of channels.
While AlexNet results in a 12x12 feature map, Inception retains a 28x28 feature map, allowing it to keep more of the original information.
In essence, the first two stages of Inception offer a more compact yet information-rich representation compared to AlexNet.

Stage 3



https://d2l.ai/

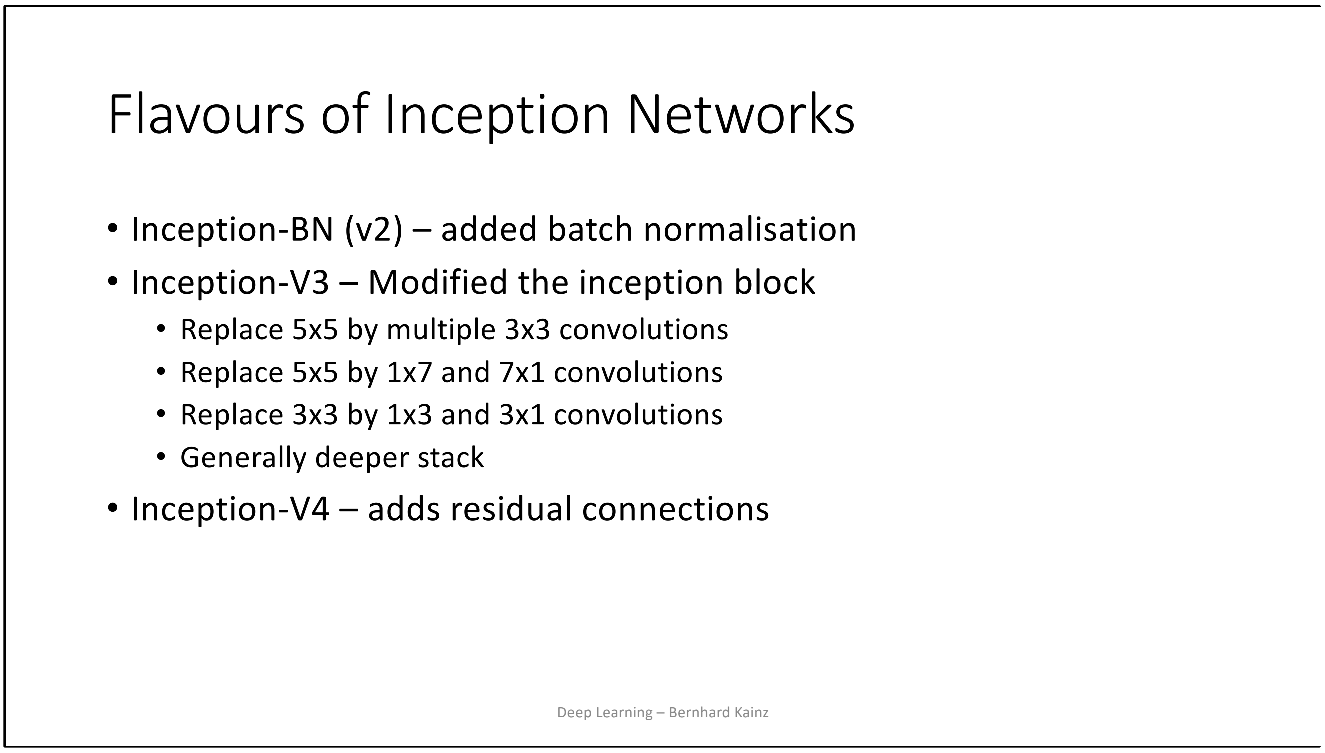
Moving on to stage 3 of the Inception architecture.
Here, we see a lot of activity and variation, even within the first and second blocks of this stage.
For example, the first block utilizes 256 channels, while the second one jumps to 480 channels.
This stage also reduces the resolution of the feature maps, taking you from an initial 28x28 down to a 14x14 grid.
This kind of dynamic allocation between channels and blocks is one of the reasons why Inception is so effective.
Another point to note is that there are different versions of the Inception network.
These versions often differ in terms of the number of stages or blocks they include, usually as a trade-off for computational complexity and performance.
The variation in the number of stages or blocks is usually due to specific needs or constraints, such as computational resources or the complexity of the data being processed.



Stage 4 & 5

832x7x7
3x3 MaxPool, stride 2, pad 2
(832)
(512)
(512)
(512)
(512)
480x14x14
Increases output channels
Increases output channels
1024x1x1
GlobalAvgPool
(1024)
(832)
832x7x7
1024 dimensional feature to output layer
Increase for output channel
https://d2l.ai/

Now let's dive into stages 4 and 5 of the Inception architecture.
Structurally, these stages are quite similar to each other.
The number of channels continues to increase, going from 512 in stage 4 up to 832 in stage 5.
You might wonder, why increase the channels before max pooling?
The reason is that max pooling will reduce the resolution, so it's advantageous to store as much information as possible per channel before that happens.
By the end of stage 5, the network boasts a whopping 1,024 channels.
Interestingly, this number aligns with the number of classes you might be trying to predict, depending on the dataset.
Then, the architecture employs a global average pooling operation.
Conducting global average pooling over a 7x7 grid is a clever design choice, as it enables the network to make more informed decisions based on the most essential features across the spatial dimensions.



Flavours of Inception Networks

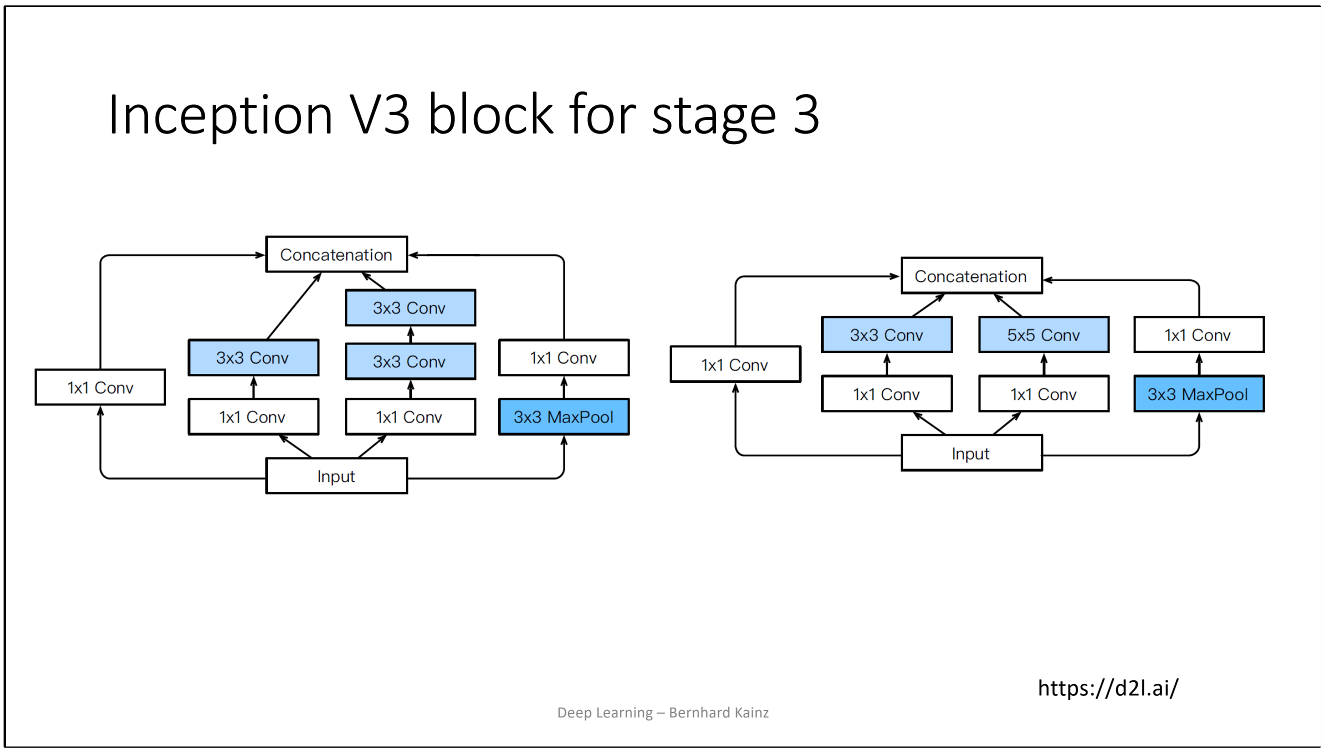
- Inception-BN (v2) – added batch normalisation
- Inception-V3 – Modified the inception block
  - Replace 5x5 by multiple 3x3 convolutions
  - Replace 5x5 by 1x7 and 7x1 convolutions
  - Replace 3x3 by 1x3 and 3x1 convolutions
  - Generally deeper stack
- Inception-V4 – adds residual connections

Deep Learning – Bernhard Kainz

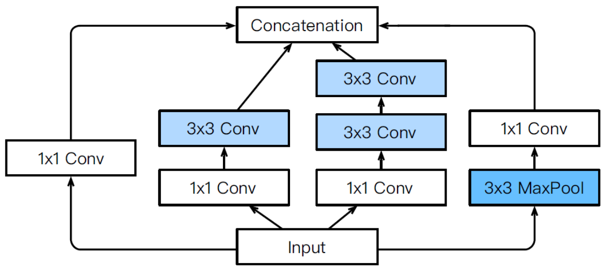
Let's talk about the evolution of GoogleNet, as it didn't just stop at the initial inception.
The architecture has seen several versions: GoogleNet v2, v3, and v4.
Each version introduced various improvements over its predecessor.
GoogleNet v2 incorporated a feature known as batch normalization, which we'll cover in more detail later on.
Then comes GoogleNet v3, which goes a step further in convolutional complexity.
Not satisfied with just the usual 1x1, 3x3, and 5x5 convolutions, this version experiments with more shapes like 1x5 or 5x1 or even 1x7.
This was an attempt to make each inception block even more versatile and deep.
If you're curious about what happens when you try to push this approach to its limits, look at AmoebaNet.
AmoebaNet is an interesting chapter in the world of convolutional neural networks.
It was developed using a form of AutoML, specifically neural architecture search via evolutionary algorithms.
The goal was to automatically discover a neural network architecture that performed well for a given task.
AmoebaNet doesn't adhere strictly to a single design principle, like the inception modules in GoogleNet.
Instead, its architecture is discovered through a process of evolutionary search, where many architectures are tested and the best-performing ones are "bred" to create new architectures.

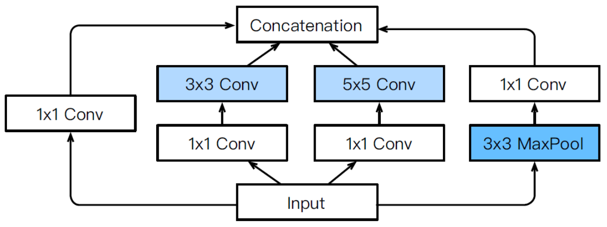
This architecture uses a plethora of different convolution shapes and styles.
This is like survival of the fittest, but for neural network structures.

Finally, GoogleNet v4 takes inspiration from another giant in the field, ResNet.
It imports some of the ResNet ideas into the Inception framework.
However, it's worth mentioning that, despite these innovations, GoogleNet v4 still doesn't outperform ResNet directly.



Inception V3 block for stage 3

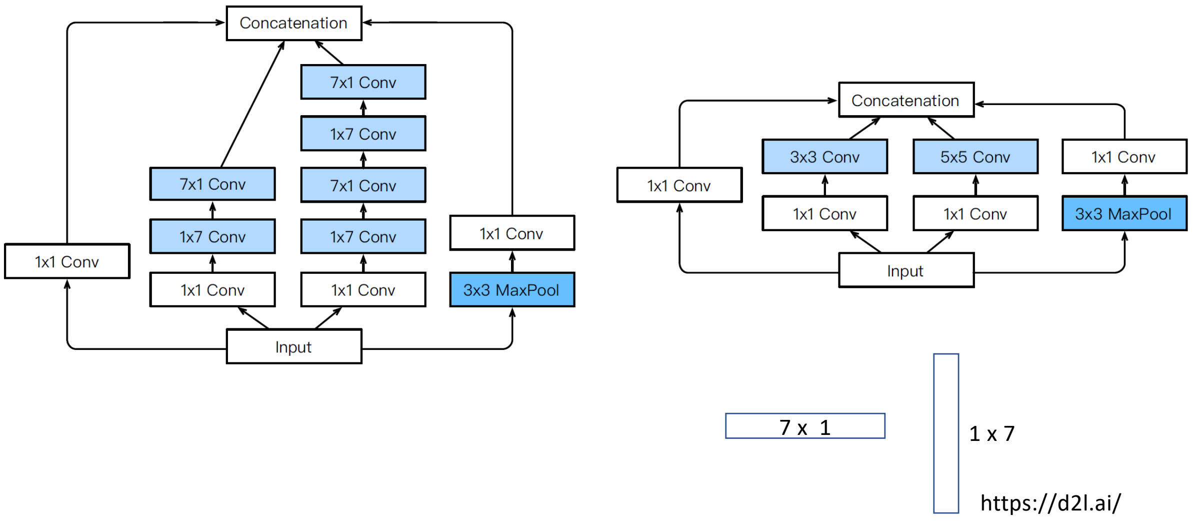




https://d2l.ai/

Let's dissect the changes between different versions of the Inception architecture, focusing on stage 3.
In version 3, you'll notice a key modification compared to its predecessor.
On the right-hand side, the 5x5 convolution has been replaced by two 3x3 convolutions.
Why is this significant?
This change is actually in line with findings from the Simonian and Zisserman paper.
They examined whether wide and shallow networks are better than deep and narrow ones.
It turns out that deep and narrow networks perform better, providing the rationale for this architectural tweak.
So, replacing a single 5x5 with two 3x3 convolutions is not just an arbitrary change; it's backed by research.
It is, in the grand scheme of Inception architecture changes, a fairly benign modification, but one that builds upon empirical evidence.

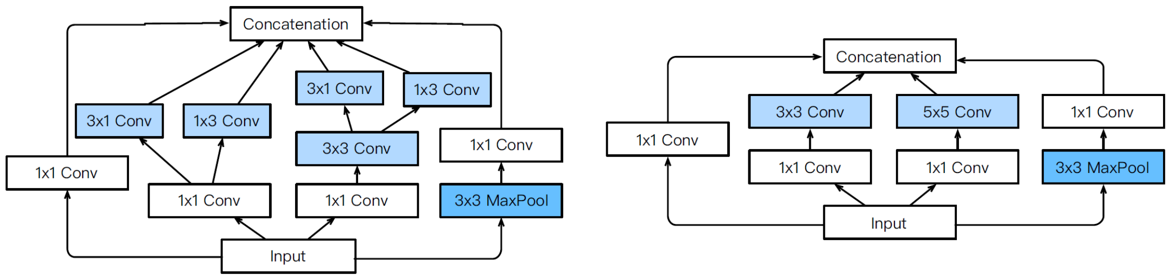
Inception V3 block for stage 4



https://d2l.ai/

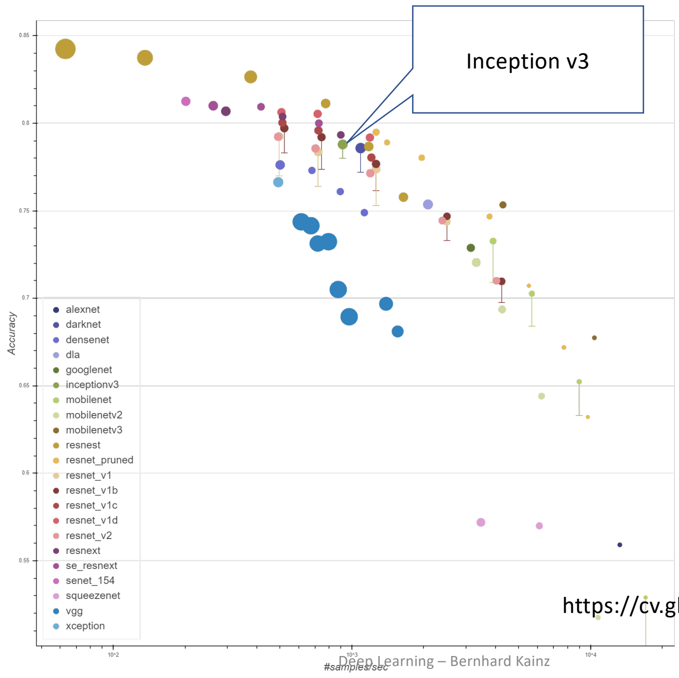
Let's now focus our attention on Stage 4 of the Inception architecture, where some of the most impactful changes were made over time.
One of the most significant shifts was the replacement of 5x5 convolutions with asymmetric 1x7 and 7x1 convolutions.
This was a groundbreaking moment in CNN design -- employing asymmetric convolutions for the first time.
What makes 7x1 unique?
Well, it's quite narrow, and it requires just 7 parameters per channel, which is computationally more efficient than a 3x3 convolution.
So how does this compare to the traditional 5x5 convolution?
A 5x5 convolution has 25 parameters, while alternating four layers of 7x1 and 1x7 convolutions only costs 28 parameters.
The computational cost is roughly the same, but the network becomes more expressive.
So, why do we even need to alternate between 1x7 and 7x1 convolutions?
Why not just stick to one of them?
The reason is that using only 1x7 or 7x1 would focus the network's learning on either vertically or horizontally contiguous features.
If you stick to one, you could end up with a network that's excellent at recognizing one type of feature but terrible at the other.
Generally, that's not what you'd want, unless you have a very specific goal in network design.

Inception V3 block for stage 5



https://d2l.ai/

Now let's delve into Stage 5 of the Inception architecture, where things get even more interesting.
In this stage, the architecture replaces the traditional 3x3 convolutions with 1x3 and 3x1 convolutions.
Yes, you're right if you've noticed a pattern here; the architecture is progressively employing more asymmetric shapes for its convolutions.
With these changes, the Inception architecture inches closer to achieving state-of-the-art performance.
This shows how iterative refinements and innovative ideas can substantially impact the effectiveness of a neural network.

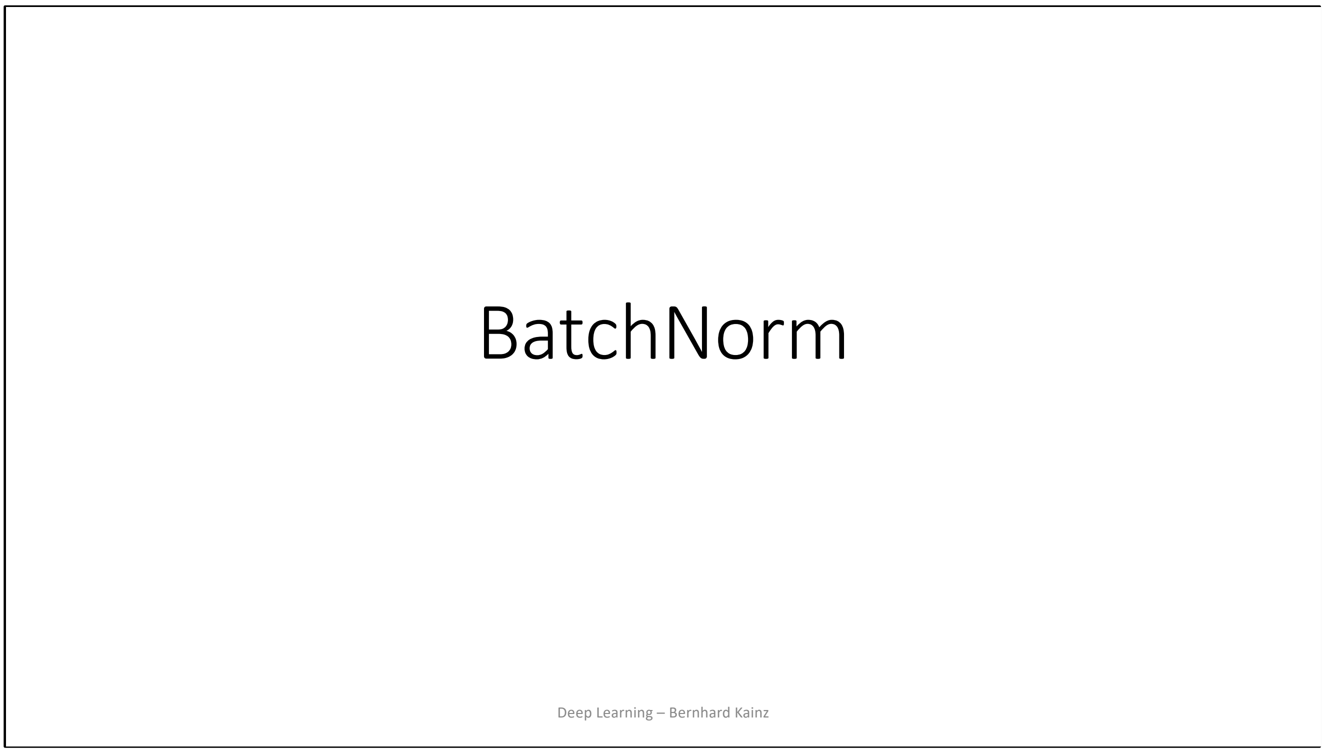


https://cv.gluon.ai/model_zoo/classification.html

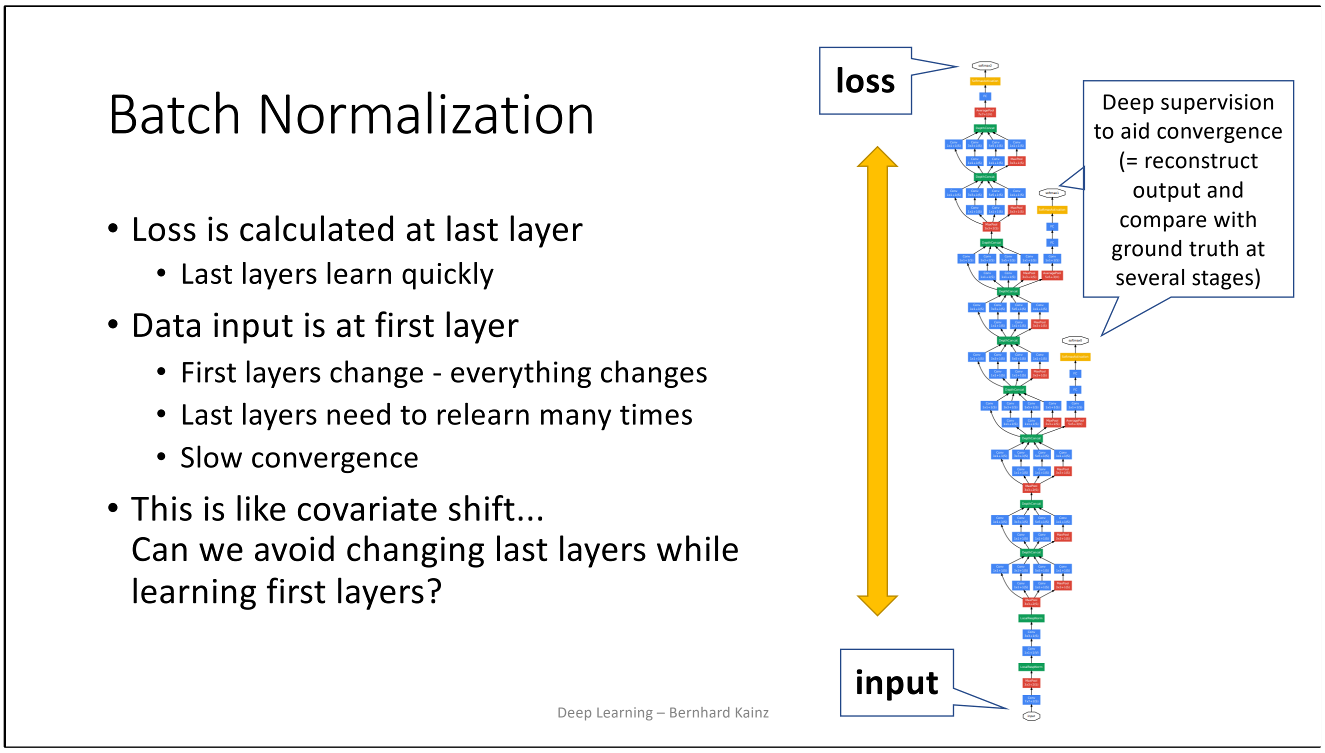
Let's now look at a plot that shows accuracy versus throughput for a variety of models.
This data comes from the Gluon Model Zoo, where multiple architectures are implemented and trained in a consistent manner.
The size of each dot on the plot corresponds to the memory footprint required to run that specific model.
Smaller dots indicate models that are more computationally efficient, whereas dots that are higher up the y-axis represent models with better accuracy.
The lines you see on the plot indicate attempts at reproduction by independent researchers.
These lines help highlight the gap between published results and what is actually achievable, often revealing that a lot of the performance gains are due to better training methods, not just the architecture.
Having so much variety lends traction to the idea of just ensembling a lot of these architectures to obtain even better results.

What do we learn from that?

- Dense layers are computationally and memory intensive. Real-world problems with big input tensors and many classes will prohibit their use.
- Again: 1x1 convolutions act like a multi-layer perceptron per pixel.
- Scientists are humans and need a while to understand the power of new approaches. Eventually they do but a lot of vanity is involved in the process.
- If not sure, just take all options and let the optimization decide or even learn this through trial and error (genetic algorithm, AmoebaNet)

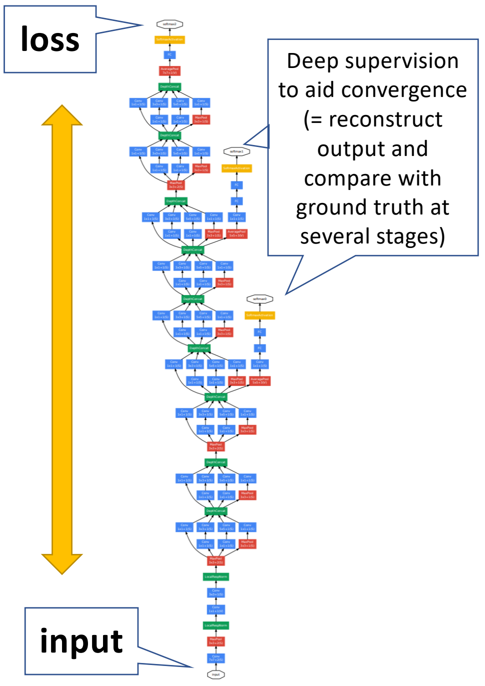


BatchNorm



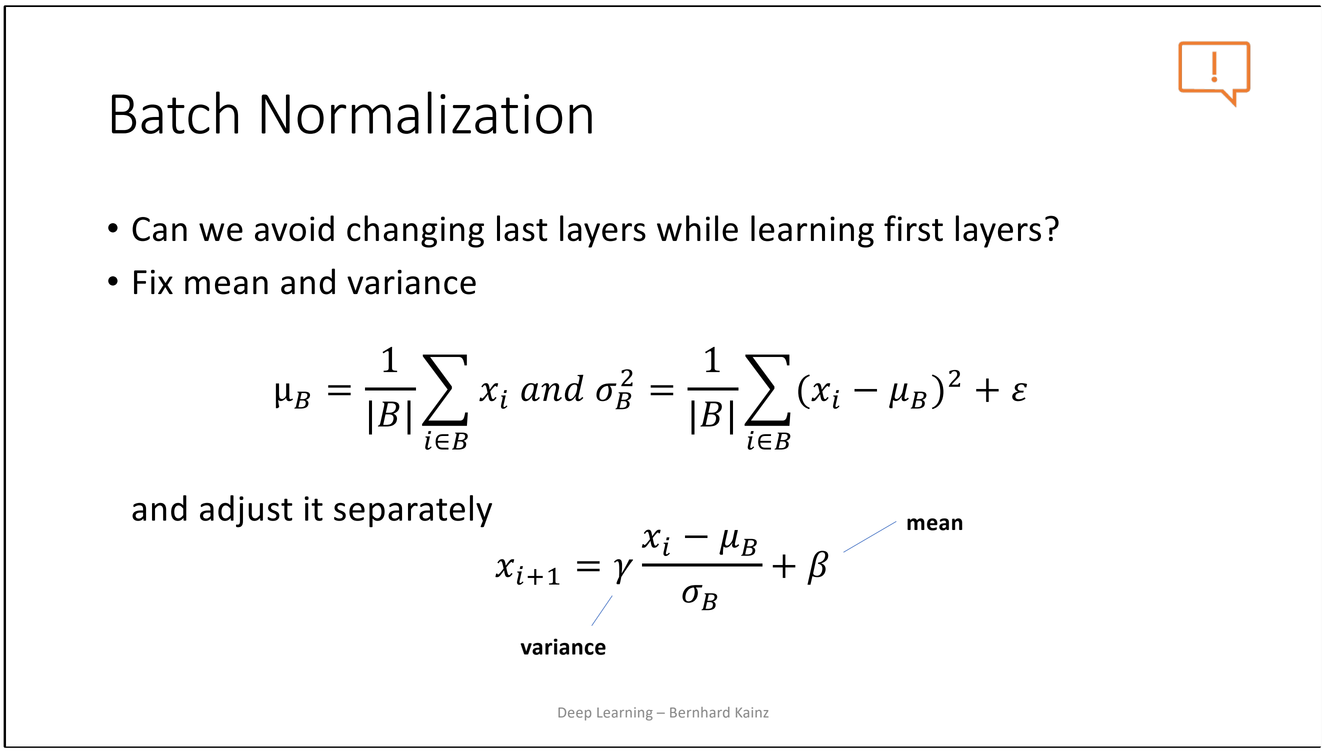
Batch Normalization

- Loss is calculated at last layer
  - Last layers learn quickly
- Data input is at first layer
  - First layers change - everything changes
  - Last layers need to relearn many times
  - Slow convergence
- This is like covariate shift...
  Can we avoid changing last layers while learning first layers?


loss
Deep supervision to aid convergence (= reconstruct output and compare with ground truth at several stages)
input

Deep Learning – Bernhard Kainz

Let's now shift our focus briefly to the concept of Batch Normalization, a key innovation aimed at addressing the challenges posed by very deep networks.
Deep networks often struggle with convergence, making training both difficult and time-consuming.
Some solutions like "deep supervision" exist, which involve backpropagating loss from intermediate stages to help the network learn.
However, this approach had limitations, prompting researchers to seek more effective methods.
Enter Batch Normalization or 'Batch Norm,' which was developed to solve this exact issue.
The intuition behind Batch Norm is that during training, 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.
Essentially, each layer's learning destabilizes the next, causing a slow convergence process that takes a long time for all layers to adapt properly.
Batch Norm mitigates this issue by normalizing the features within each mini-batch, thereby stabilizing the training process and speeding up convergence.



Batch Normalization

- Can we avoid changing last layers while learning first layers?
- Fix mean and variance

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

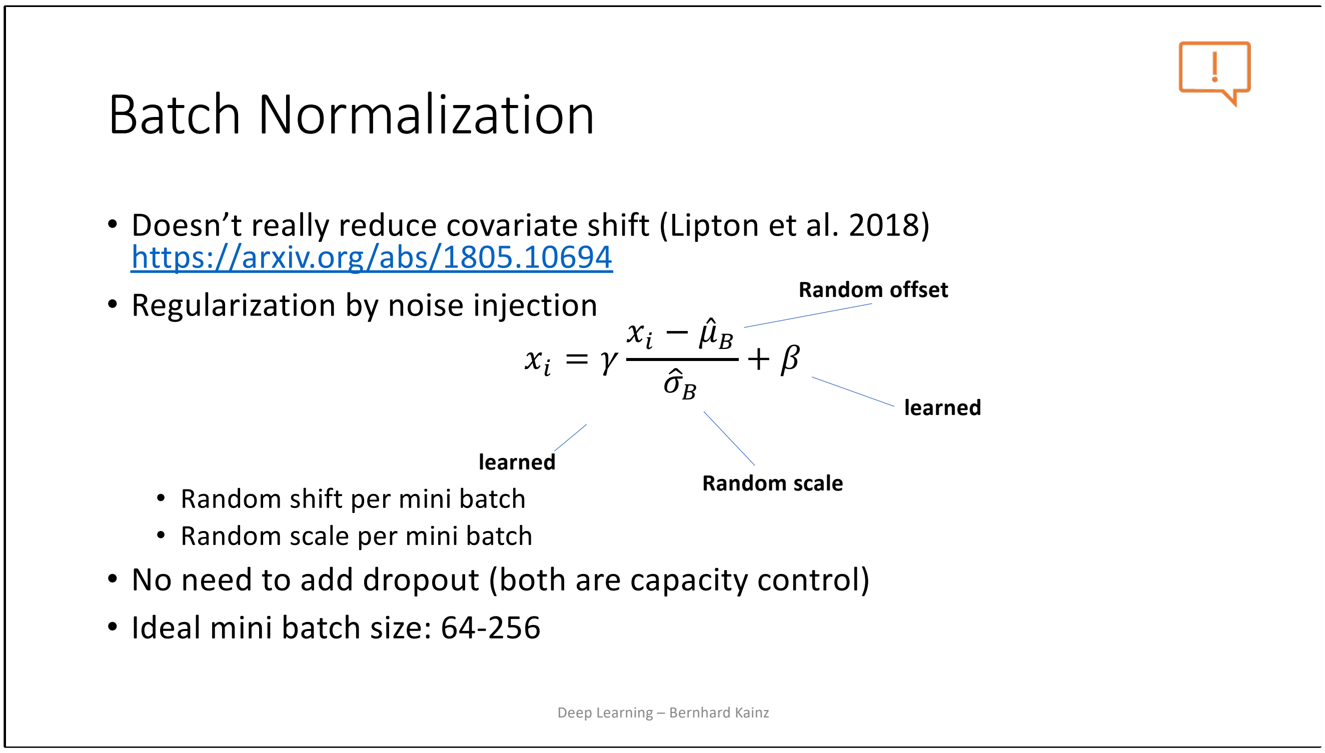
and adjust it separately

$$x_{i+1} = \underbrace{\gamma}_{\text{variance}} \frac{x_i - \mu_B}{\sigma_B} + \overbrace{\beta}^{\text{mean}}$$

Deep Learning – Bernhard Kainz

Let's dive into how Batch Normalization or 'Batch Norm' actually functions in solving the problem of slow convergence in deep networks.
The idea is to make minor corrections to the layers by adjusting their mean and variance during training.
We don't aim to fully fix these layers, but rather fine-tune them using an affine transform that the model learns during training.
An affine transform essentially means multiplication and addition operations.
So, what we do is compute the mean and variance -- represented by mu and Sigma squared—of a given mini-batch during training.
The next step is to re-normalize each input feature by subtracting its mean and dividing it by its standard deviation.
But we don't stop there; we introduce two new parameters, gamma and beta, which the model learns.
These parameters are used to scale (gamma) and shift (beta) the normalized features.
This process is what we refer to as Batch Normalization.
You might be asking why exactly this works or if the initial intuition about correcting the internal covariate shift was really why Batch Norm was created.
Well, the efficacy of Batch Norm has been proven across numerous experiments, although the complete understanding of why it works so effectively is still a topic of ongoing research.
The first equation μB=|B|1∑i∈Bxi calculates the mean (μB) of the mini-batch B. Here, |B|

is the size of the mini-batch, and xi are the individual data points in that mini-batch.
The second equation $\sigma B2=|B|1\sum i\in B(xi-\mu B)2+\epsilon$ calculates the variance ($\sigma B2$) of the same mini-batch. The term ϵ is a small constant added for numerical stability.
Now, the idea is to adjust these first and second moments separately from the rest of the network learning. That leads us to the third equation $xi+1=\gamma\sigma B(xi-\mu B)+\beta$.
In this equation, xi+1 is the normalized and adjusted version of the original data point xi. The terms γ and β are scale and shift parameters that are learned during training.
By using these equations, we're able to "normalize" each mini-batch separately, making the training of deep networks more stable and faster. However, remember that γ and β are learned, allowing the network some flexibility to undo this normalization if it finds it beneficial for the learning task at hand.



Batch Normalization

- Doesn't really reduce covariate shift (Lipton et al. 2018) https://arxiv.org/abs/1805.10694
- Regularization by noise injection

Random offset

$$x_i = \gamma \frac{x_i - \hat{\mu}_B}{\hat{\sigma}_B} + \beta$$

learned

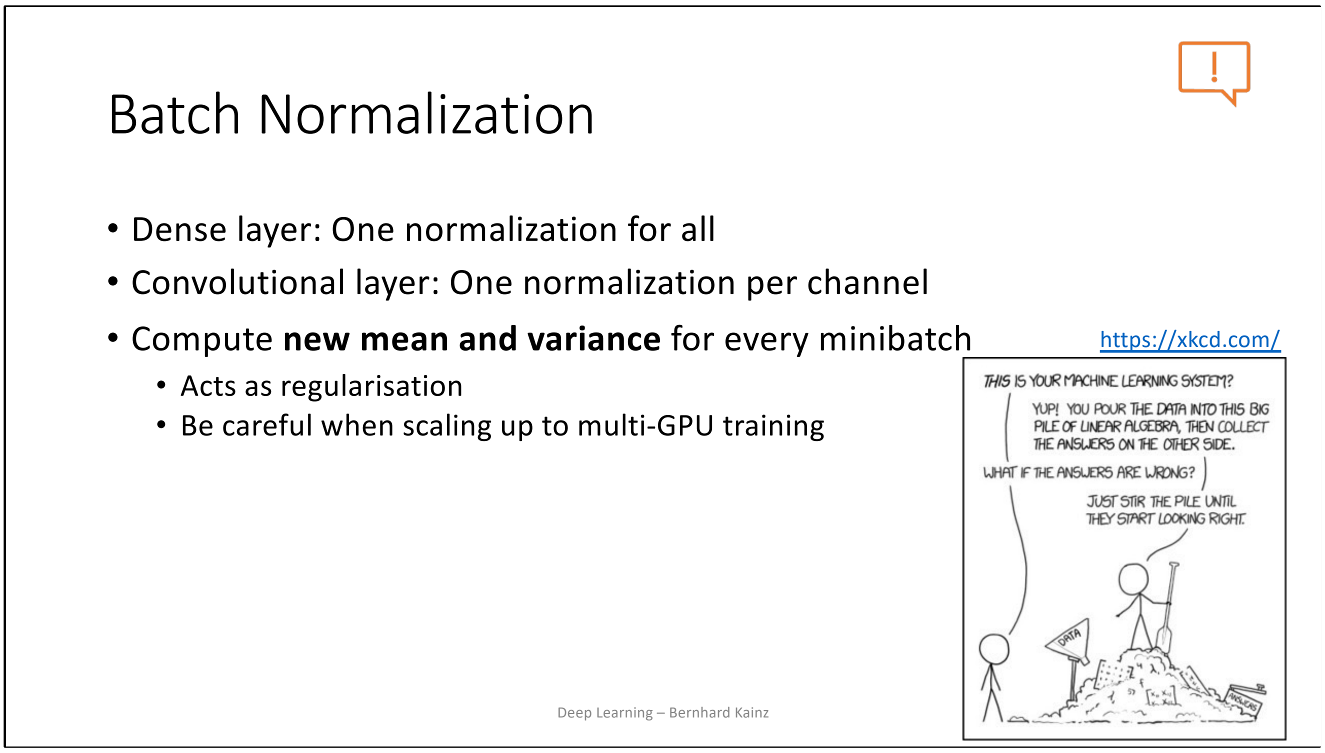
learned

Random scale

  - Random shift per mini batch
  - Random scale per mini batch
- No need to add dropout (both are capacity control)
- Ideal mini batch size: 64-256

Let's address a crucial point about Batch Normalization: its original motivation of reducing covariate shift is actually not correct.
A study by Lipton and colleagues revealed that BatchNorm may even worsen covariate shift, yet it still aids in model convergence.
So, what's going on here?
It turns out that 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.

So, what happens during inference or test time?
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.



Batch Normalization

- Dense layer: One normalization for all
- Convolutional layer: One normalization per channel
- Compute **new mean and variance** for every minibatch
  - Acts as regularisation
  - Be careful when scaling up to multi-GPU training

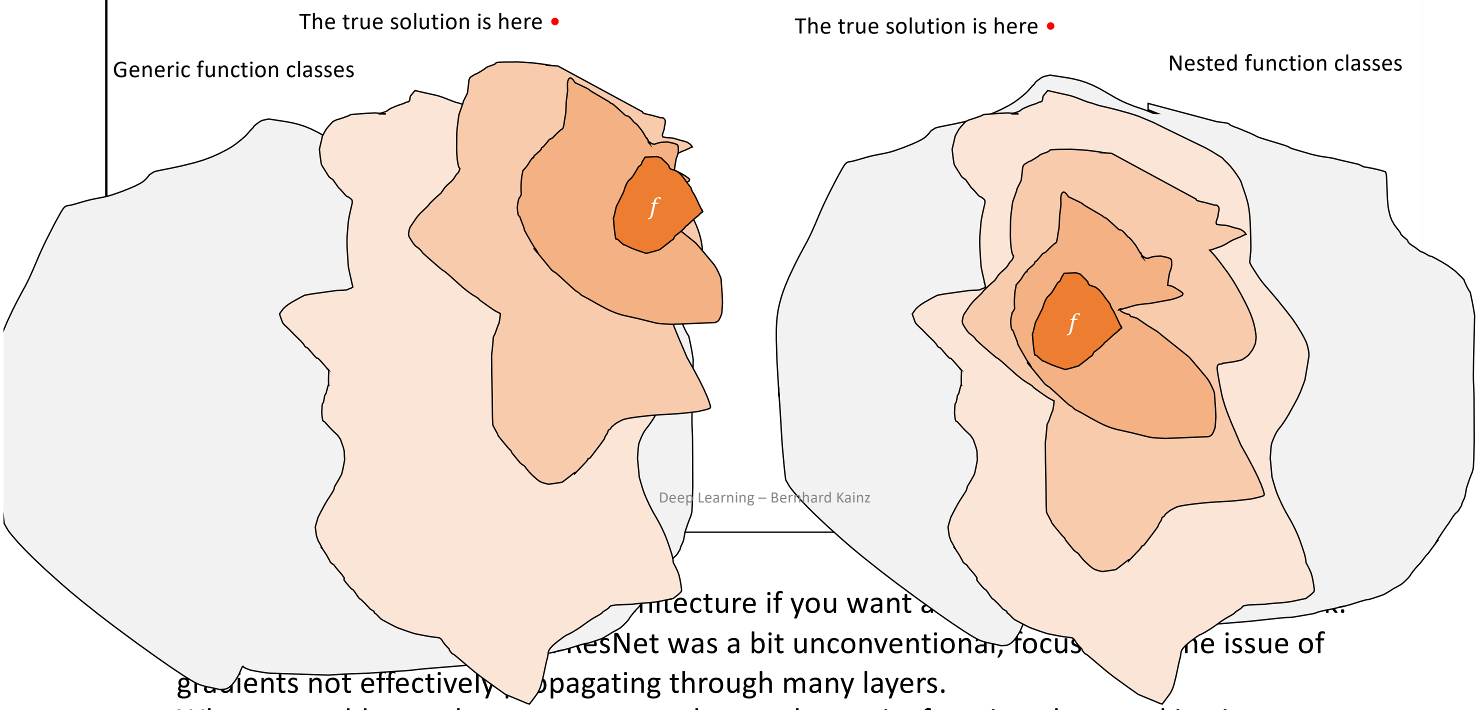
https://xkcd.com/

THIS IS YOUR MACHINE LEARNING SYSTEM?
YUP! YOU POUR THE DATA INTO THIS BIG PILE OF LINEAR ALGEBRA, THEN COLLECT THE ANSWERS ON THE OTHER SIDE.
WHAT IF THE ANSWERS ARE WRONG?
JUST STIR THE PILE UNTIL THEY START LOOKING RIGHT.
DATA
ANSWERS

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.
Batch normalization calculates a new mean and variance for every mini-batch during the training phase.
It's essential to set the 'train' flag correctly in your network to ensure appropriate behavior.
This is because batch normalization acts differently during the training and testing phases.

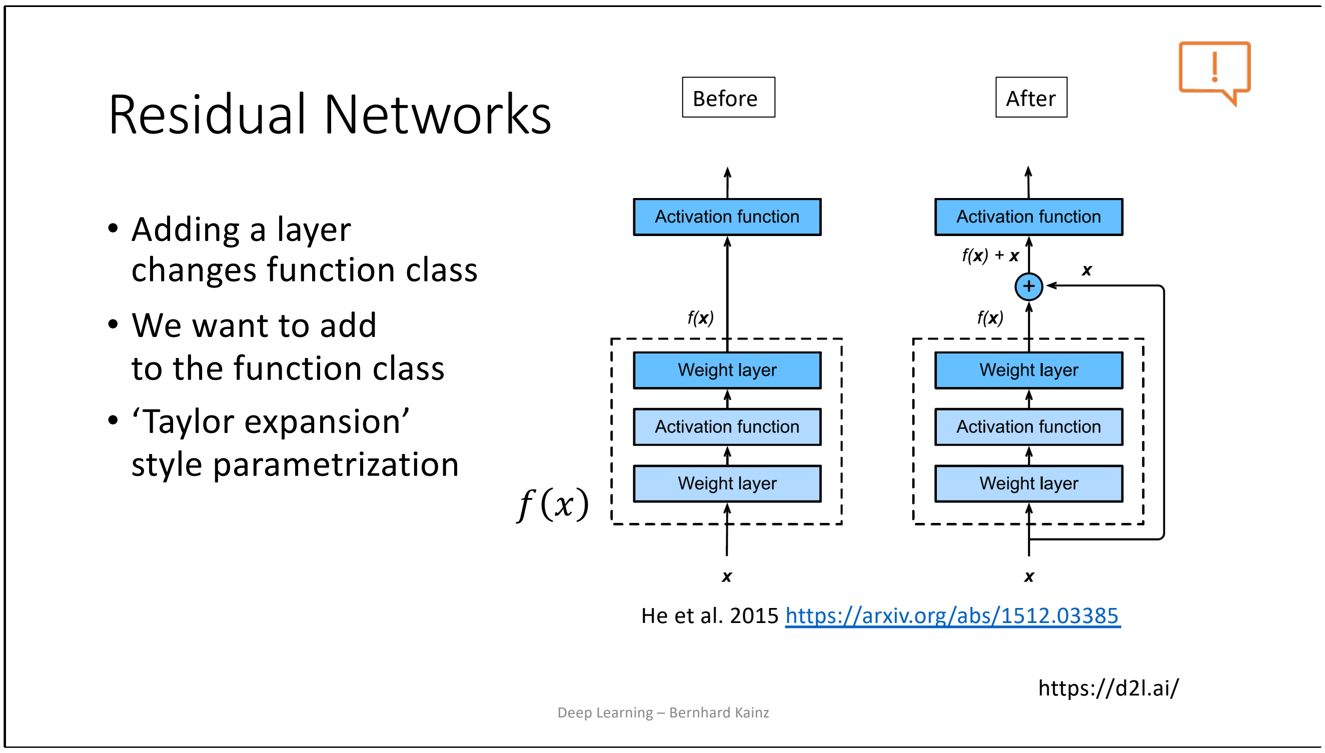
ResNet

Does adding layers improve accuracy?



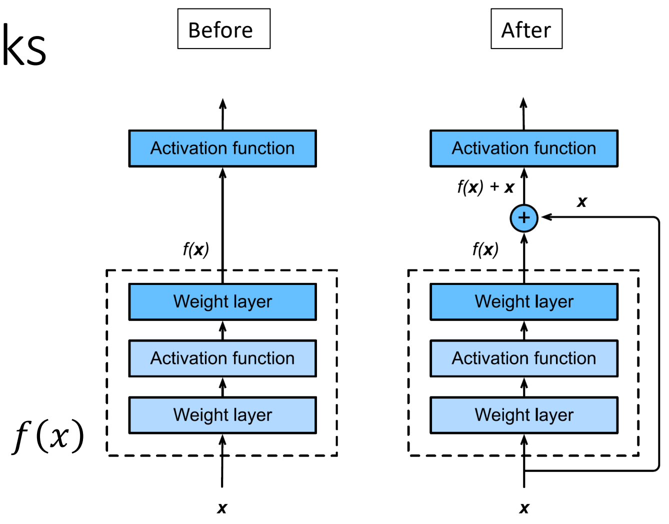
itecture if you want a
esNet was a bit unconventional, focus he issue of
gr ients not effectively opagating through many layers.
When you add more layers to a network, you change its function class, making it more versatile but also different.
More layers means more parameters, granting us the power to model more complex functions.
However, as we add layers, our function classes do become more powerful but also diverge from each other.
Imagine trying to approximate a 'true function,' represented here by a red dot.
Adding more layers might get you closer to this truth initially, but after a certain point, you could drift away.
This situation makes it difficult to make engineering decisions about how many layers should be in your network.
In an ideal world, like the theories presented in statistical learning papers, your function classes would be neatly nested as you add layers, each one more powerful but similar to the last.
Unfortunately, deep networks don't always work this way, so how can we make it more like our ideal scenario?
The goal is to have function classes that are both increasingly powerful and nested as we add layers.

This way, as we continue to add layers, we keep moving closer to approximating the 'true function' represented by the red dot.
While these function classes may not be convex, the aim is to make them as nested as possible, which is what ResNet achieves.



Residual Networks

- Adding a layer changes function class
- We want to add to the function class
- 'Taylor expansion' style parametrization

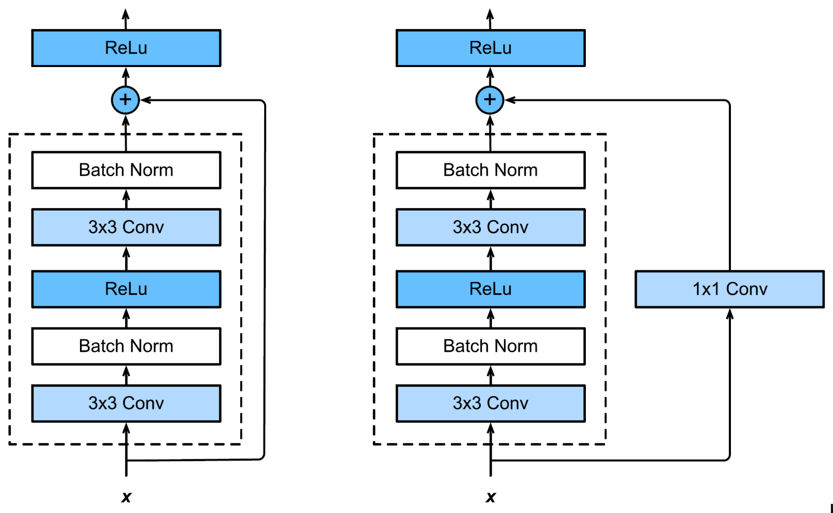


He et al. 2015 https://arxiv.org/abs/1512.03385

https://d2l.ai/

The ingenious innovation of ResNet came from He et al. in 2015.
Instead of parameterizing around a zero function f(x)=0, ResNet parameterizes around the identity function f(x)=x.
This shift is incredibly insightful because when all parameters are zero, adding the input to the output results in the simplest function, which is the identity function.
You can think of this approach as akin to a Taylor expansion but for neural networks.
Now, as you tune your parameters, you start to diverge from this identity function.
The beauty is that you don't have to learn the identity function from scratch, reducing unnecessary computational overhead.
It implies a different inductive bias and allows for the addition of new layers without disrupting the outputs of previous layers.
Although ResNet doesn't precisely implement nested function classes, it comes remarkably close.
This approach is encapsulated in what we call a ResNet block, where the input is added to the output residual.
This simple yet effective change allows for deeper networks without the degradation problems commonly associated with them.

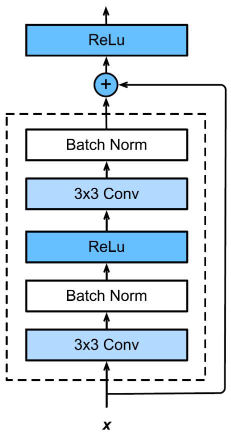
ResNet Block



https://d2l.ai/

ResNet architecture consists of multiple building blocks, each with specific layers.
Each block generally contains two or more convolutions, each followed by a batch normalization and a Rectified Linear Unit (ReLU).
The innovation lies in adding the original input back to the output of this sequence.
In some cases, we might run a 1x1 convolution on the input before adding it to the output, especially if we need to change the dimensionality of the input to match the output.
This setup makes it easier for the network to learn an identity mapping, essentially enabling the network to decide whether or not a change is beneficial.
What we're discussing here is primarily the forward function, which defines how the network transforms the input to the output.
In a typical block, you'd have something like y=BatchNorm1(Conv1(x)), followed by ReLU, and then another batch norm.
If the block includes a shortcut connection, you'd add the original input x to this sequence, after potentially applying another convolution to x.
To make this work seamlessly, you need to carefully choose the dimensions of your convolutions, which is a concept we already covered when discussing the Inception architecture.
Finally, this arrangement of batch norm and ReLU blocks makes for a powerful, yet computationally efficient, network structure.

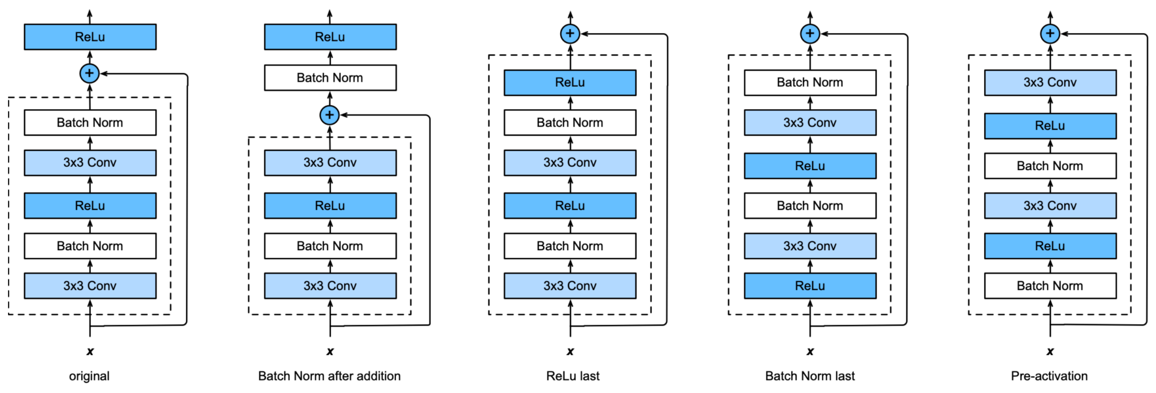
ResNet Block



```
def forward(self, X):
    Y = self.bn1(self.conv1(X))
    Y = nd.relu(Y)
     Y = self.bn2(self.conv2(Y))
     if self.conv3:
         X = self.conv3(X)
     return nd.relu(Y + X)
```

Let's take a closer look at how ResNet's architecture is implemented in code.
The **forward** method defines the sequence of operations that each ResNet block performs.
The input, denoted as *X*, first passes through a convolutional layer, **conv1**, followed by batch normalization, **bn1**.
We then apply the ReLU activation function to add non-linearity to the output.
The next sequence is another convolution, **conv2**, followed by another batch normalization, **bn2**.
Now, if we have an additional convolution layer, **conv3**, it will be applied to the original input *X*.
Finally, we add this potentially transformed input *X* to *Y* and apply ReLU.
This is the essential magic of ResNet: the addition of the original input to the output, which helps the network learn more efficiently and effectively.
Remember, the conditional **if self.conv3:** allows us to optionally adjust the dimensions of *X* to ensure that it can be added to *Y*.
This is a simplified representation; in a real-world application, you might have additional complexities.

ResNet block flavours



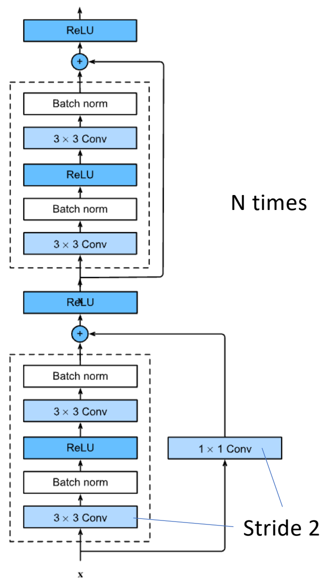
Trial and error of every permutation

https://d2l.ai/

ResNet has seen various adaptations and experiments over the years.
Researchers have tried different configurations, especially in terms of the placement of Batch Normalization and the addition operation.
Some studies have placed the batch normalization after the addition, challenging conventional sequence.
Others have experimented with the order of ReLU, convolutions, and batch normalization, testing all three permutations.
The block sequence might remain the same, but the internal ordering can differ.
Interestingly, there's no one-size-fits-all answer to which configuration is best; it often depends on the specific problem and dataset at hand.
This may sound frustrating, but it's the reality of deep learning.
The field progresses by trial and error: researchers experiment, find what works well in a given context, and then those architectures often become standard practice.

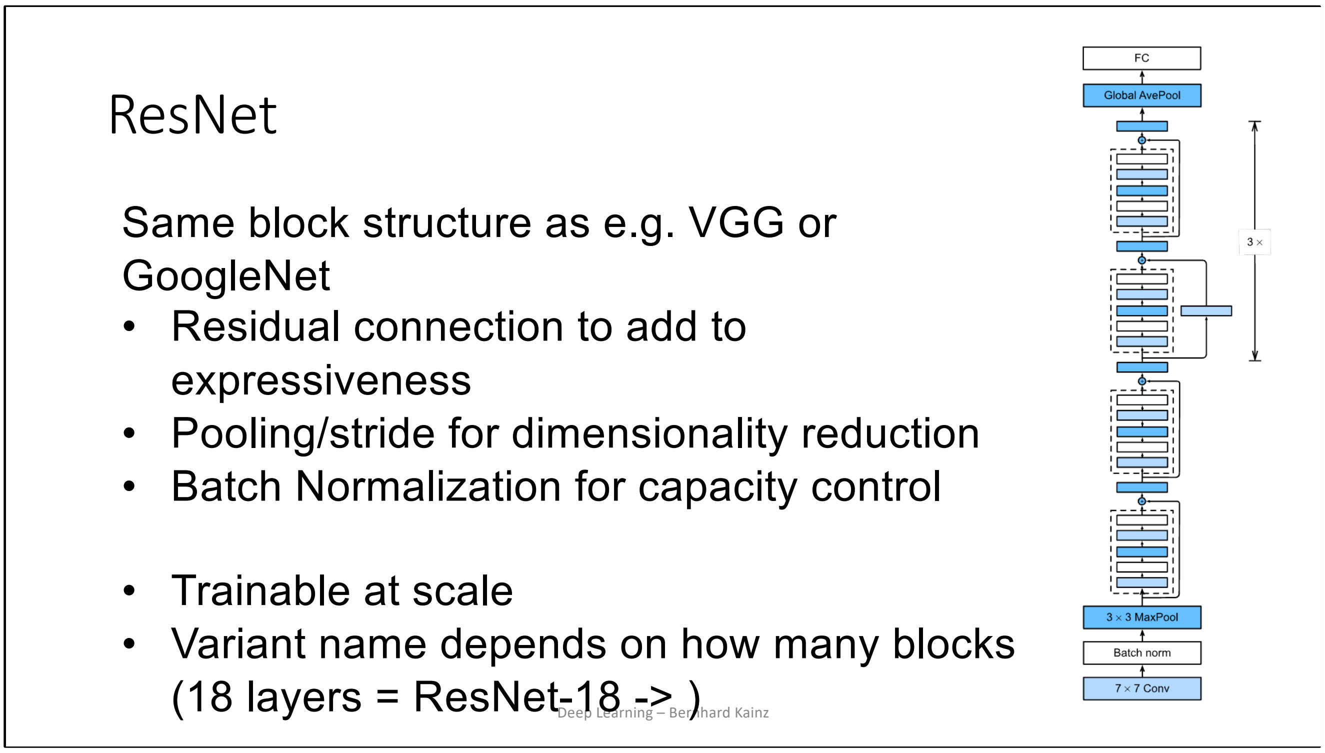
ResNet Module

- Downsample per module (stride=2)
- Enforce some nontrivial nonlinearity per module (via 1x1 convolution)
- Stack up in blocks



https://d2l.ai/

Let's delve into the ResNet module.
It closely resembles what we've previously seen, but the primary building block here is the ResNet block, not the Inception block.
A typical ResNet module may include a block with downsampling; this usually employs a stride of 2.
Following this, several other standard ResNet blocks are stacked.
Downsampling in each module is commonly performed using a stride of 2, serving to reduce the spatial dimensions of the feature maps.
To introduce complexity and non-linearity into the network, 1x1 convolutions are often employed within the ResNet blocks.
Finally, these blocks are stacked together to form a deep and complex network architecture.
This structure has proven to be highly effective for a wide range of machine learning tasks.



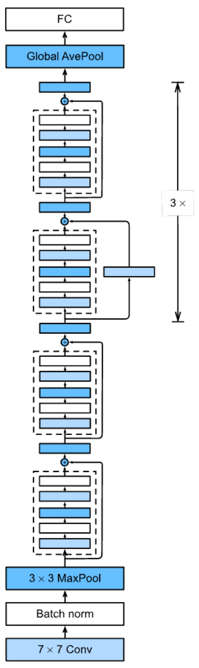
ResNet

Same block structure as e.g. VGG or GoogleNet

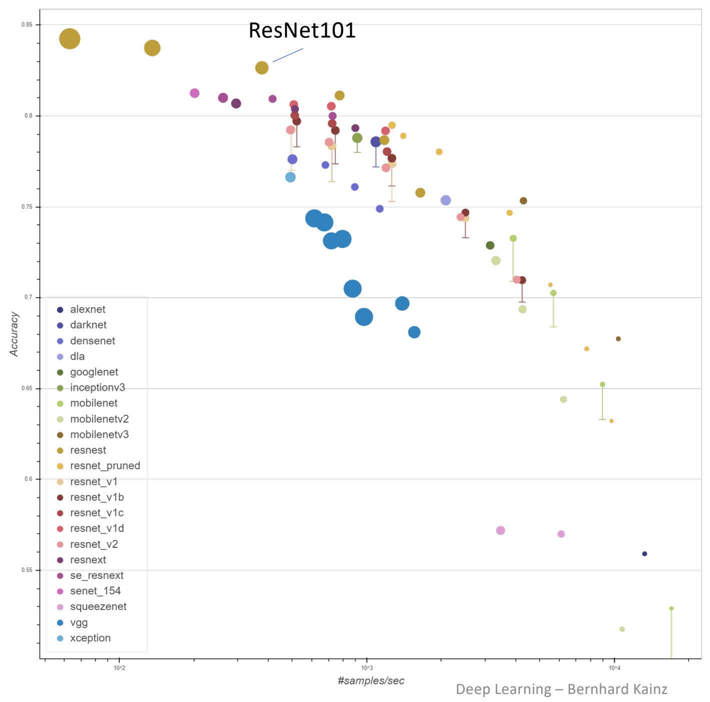
- Residual connection to add to expressiveness
- Pooling/stride for dimensionality reduction
- Batch Normalization for capacity control

- Trainable at scale
- Variant name depends on how many blocks (18 layers = ResNet-18 ->)

Deep Learning – Bernhard Kainz



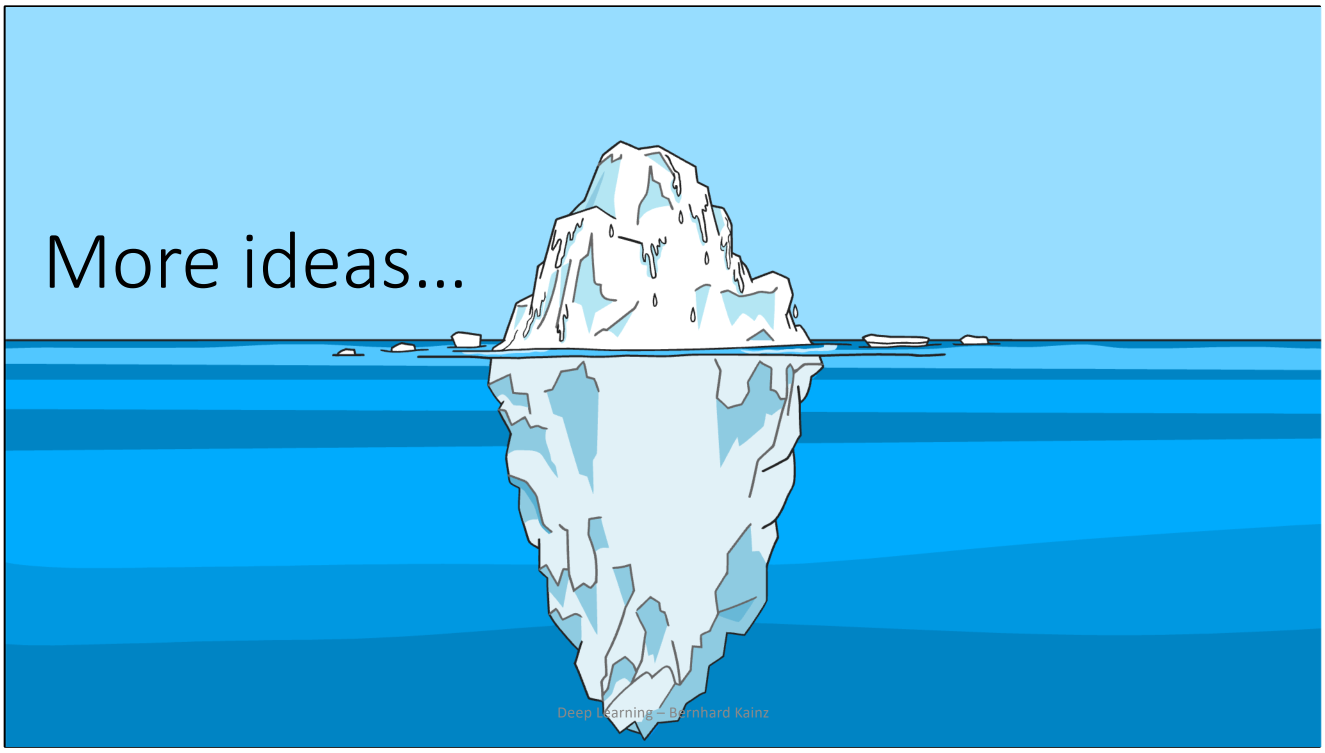
Let's take a moment to appreciate the full architecture of ResNet, which scales impressively from 18 layers all the way up to 200+ layers.
The lower layers of this architecture closely resemble what we've observed in previous architectures like VGG and GoogleNet.
They usually start off with a typical sequence of convolutions, batch normalization, and pooling layers.
Residual connections are a defining feature that contribute to the network's expressiveness and allow for training deeper models without facing vanishing or exploding gradients.
To manage dimensionality, ResNet uses pooling and strides just like earlier architectures.
Batch normalization layers are interspersed throughout to control the capacity and stabilize the training process.
What makes ResNet particularly remarkable is that it's designed to be trainable at scale.
The naming convention for ResNet variants is straightforward; the number of layers corresponds to the variant name, such as ResNet-18 for an 18-layer architecture.



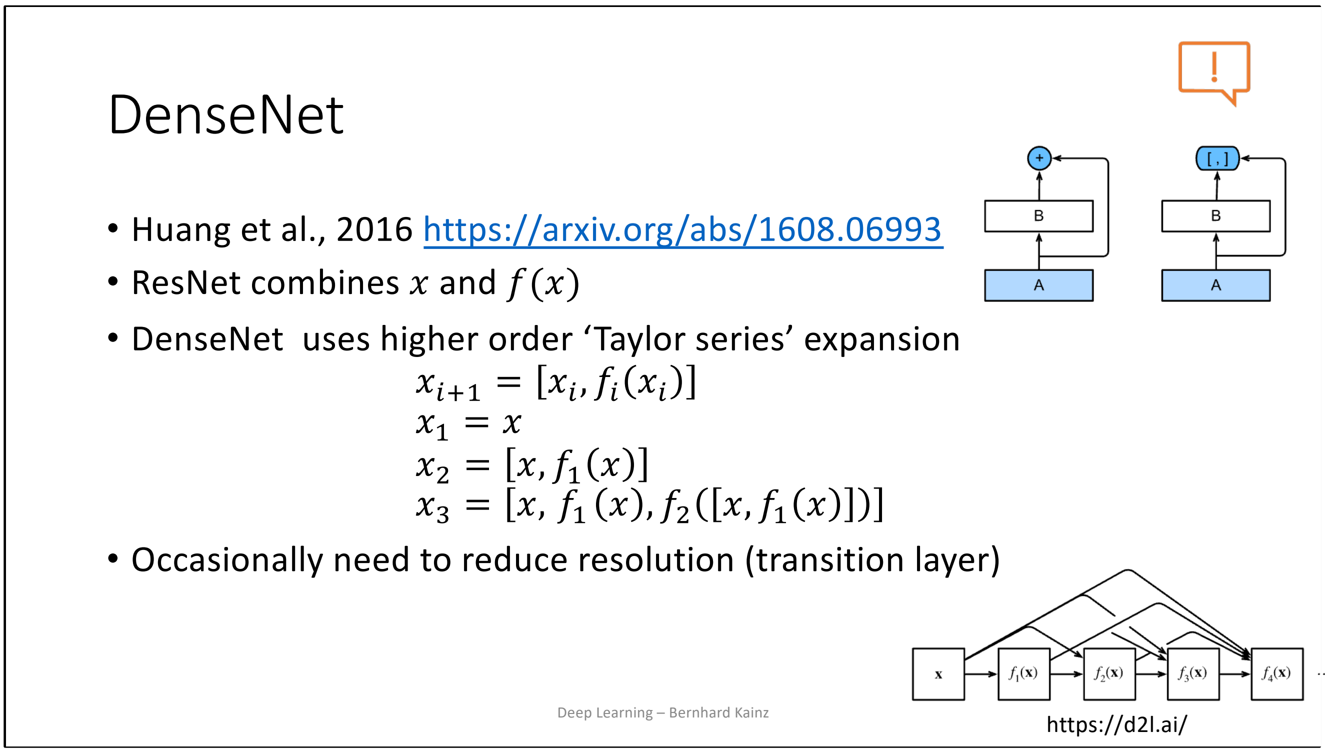
https://cv.gluon.ai/model_zoo/classification.html

On this slide, you'll see a chart that plots accuracy against throughput.
ResNet-101, represented by the dot at the top, is virtually state-of-the-art.
Yes, it may be slower in terms of processing speed, but what it lacks in speed, it makes up for in accuracy.
In terms of its size or footprint, ResNet-101 is actually more compact compared to the Inception network we discussed earlier.
If you look at the orange circle on the chart, that represents the performance of the Inception network, which you'll notice sits lower on the accuracy axis.
ResNet-101 offers a balanced trade-off, providing you with both high accuracy and a smaller model size.

More ideas…



So, you might be wondering, what comes after ResNet?
It's a valid question, given how pivotal ResNet has been in the field of computer vision.
Well, ResNet opened the floodgates to a whole zoo of architectures that were inspired by, or built upon, its foundational ideas.
What I'll present next is just the tip of the iceberg, but it's enough to give you a sense of the innovation that has occurred post-ResNet.

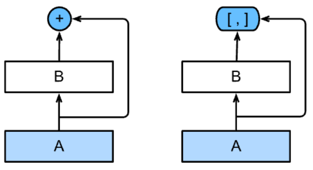


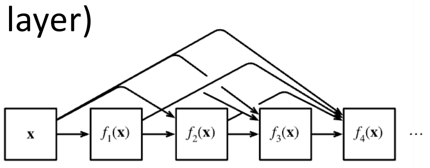
DenseNet

- Huang et al., 2016 https://arxiv.org/abs/1608.06993
- ResNet combines x and $f(x)$
- DenseNet uses higher order 'Taylor series' expansion

$$\begin{aligned} x_{i+1} &= [x_i, f_i(x_i)] \\ x_1 &= x \\ x_2 &= [x, f_1(x)] \\ x_3 &= [x, f_1(x), f_2([x, f_1(x)])] \end{aligned}$$

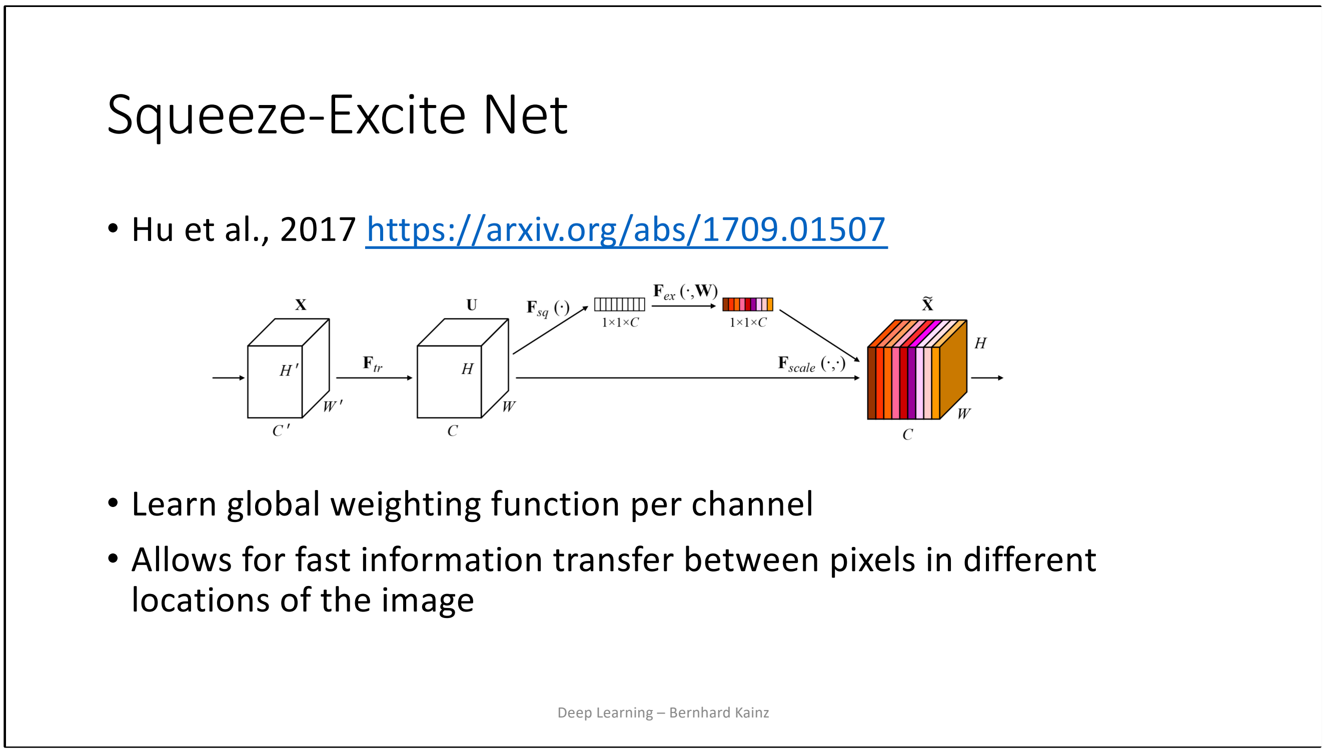
- Occasionally need to reduce resolution (transition layer)





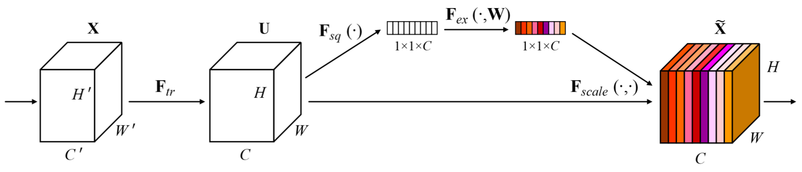
https://d2l.ai/

DenseNet stands for Densely Connected Convolutional Networks, and as the name suggests, its defining feature is dense connectivity.
Each layer in a DenseNet is connected to every other layer in a feed-forward manner, quite unlike traditional CNNs where each layer is only connected to its immediate successor.
The key advantage of this dense connectivity is feature reuse: each layer gets the feature maps from all the preceding layers, not just the last one.
This feature reuse not only makes the network highly efficient but also requires fewer parameters for high performance.
So, you get a network that's both computationally and memory-efficient.
Dense connections also help alleviate the vanishing gradient problem, which is often a challenge in very deep networks.
This improves the gradient flow through the network, making optimization easier.
Additionally, the architecture itself acts as a form of regularization, reducing the likelihood of overfitting, even when the model is large.
One of the big wins of DenseNet is its scalability: you can easily adjust the number of layers and other hyperparameters to make it suitable for a variety of tasks and data sizes.



Squeeze-Excite Net

- Hu et al., 2017 https://arxiv.org/abs/1709.01507



- Learn global weighting function per channel
- Allows for fast information transfer between pixels in different locations of the image

Let's dive into a network architecture that's a bit more 'exciting'—the Squeeze-Excite Network, or SE-NET for short. What sets this apart is its use of attention mechanisms."

What is Attention?
"In machine learning, attention allows a model to focus on significant features within the data, rather than indiscriminately treating all features equally. Think of it like spotlighting the essential parts in an otherwise noisy scene."

Analogy: Channels for Cat, Dog, Dinosaur
"Now, imagine you have various 'channels' within an image you're analyzing. Let's call them 'cat,' 'dog,' and 'dinosaur' channels. These channels represent the features that the model is focusing on at any given time."

The Paradox of Recognition
"Here's the tricky part: To effectively recognize a cat, for instance, you'd think you'd need to focus on the 'cat channel.' But how can you do that before you know it's a cat? That's a paradox SE-NET aims to solve."

Information Transfer in Conventional CNNs
"In traditional Convolutional Neural Networks, or CNNs, information moves relatively slowly from layer to layer. This can mean that it takes several layers for information from

one corner of an image to influence the other corner."

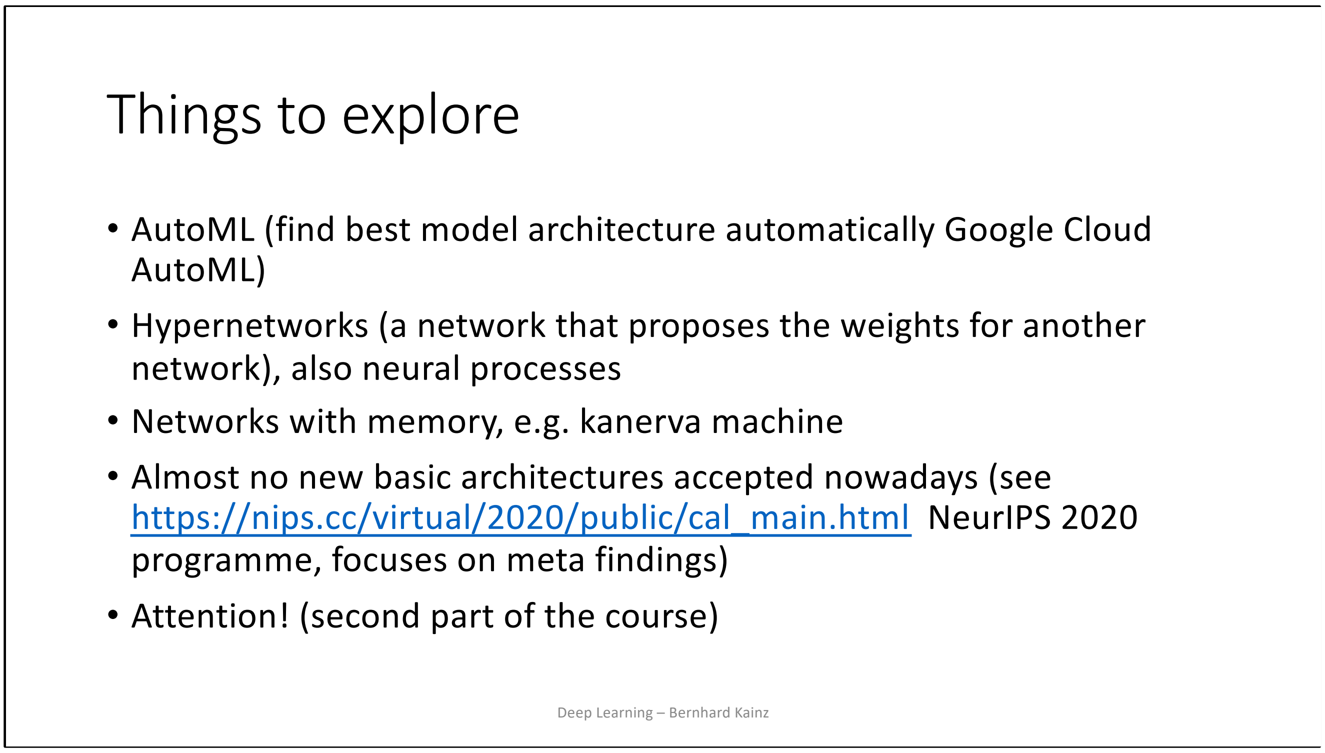
Context Matters
"SE-NETs are smarter. Say you have a bowl of milk in one part of the image. The presence of this object might indicate there's likely to be a cat somewhere else. SE-NET uses this kind of contextual information to influence its decisions."

Technical Details
"So, how does SE-NET do it? Here's a simplified explanation:

The network calculates a global descriptor for each channel.
These descriptors go through a small neural network that outputs new descriptors.
These new descriptors are then used to scale or 'excite' the original channels.
This is computationally efficient and offers a more refined way to focus on what's crucial in the image."
Model Zoo
"In terms of performance, SE-NETs are a big deal. They are among the top-performing architectures in various model repositories, commonly known as 'model zoos.'"

The paradox of needing to know what you are recognizing before you actually recognize it is addressed in the Squeeze-and-Excite Network (SE-Net) through the use of attention mechanisms. The attention mechanism works by "squeezing" the information from all channels into a global descriptor and then "exciting" or re-weighting each channel based on this descriptor. Essentially, it uses global context to guide the focus on specific channels, without needing to first identify what the object is.
Here's a simplified explanation of how the paradox is resolved:
1.**Squeeze**: First, SE-Net takes a global average pooling of each channel to get a single descriptor that captures the global information of that channel.
2.**Descriptor**: These descriptors are then fed into a small neural network (usually a fully-connected layer followed by a ReLU and another fully-connected layer) to create a new set of descriptors. These new descriptors are designed to capture which channels are more important given the current global context of the image.
3.**Excite**: The output descriptors are then used to scale the original channels. This scaling effectively "excites" or enhances the channels that are likely to be more important for a given task, based on the global context of the image.
4.**Re-weighting Channels**: So if the global context suggests that the image likely contains a cat, the "cat channel" would be scaled up, emphasizing features relevant to cats and making it easier for subsequent layers to detect the cat.
This procedure is computationally efficient and allows the network to focus on relevant features without needing to explicitly know in advance what those features represent (like a "cat" or a "dog"). It's a way of allowing the network to dynamically allocate its attention based on the overall features in the image, resolving the paradox.

SE-NETs combine the power of attention mechanisms with the foundational strengths of CNNs. They allow for a selective, context-aware focus on important features within images, making them both efficient and effective."
s



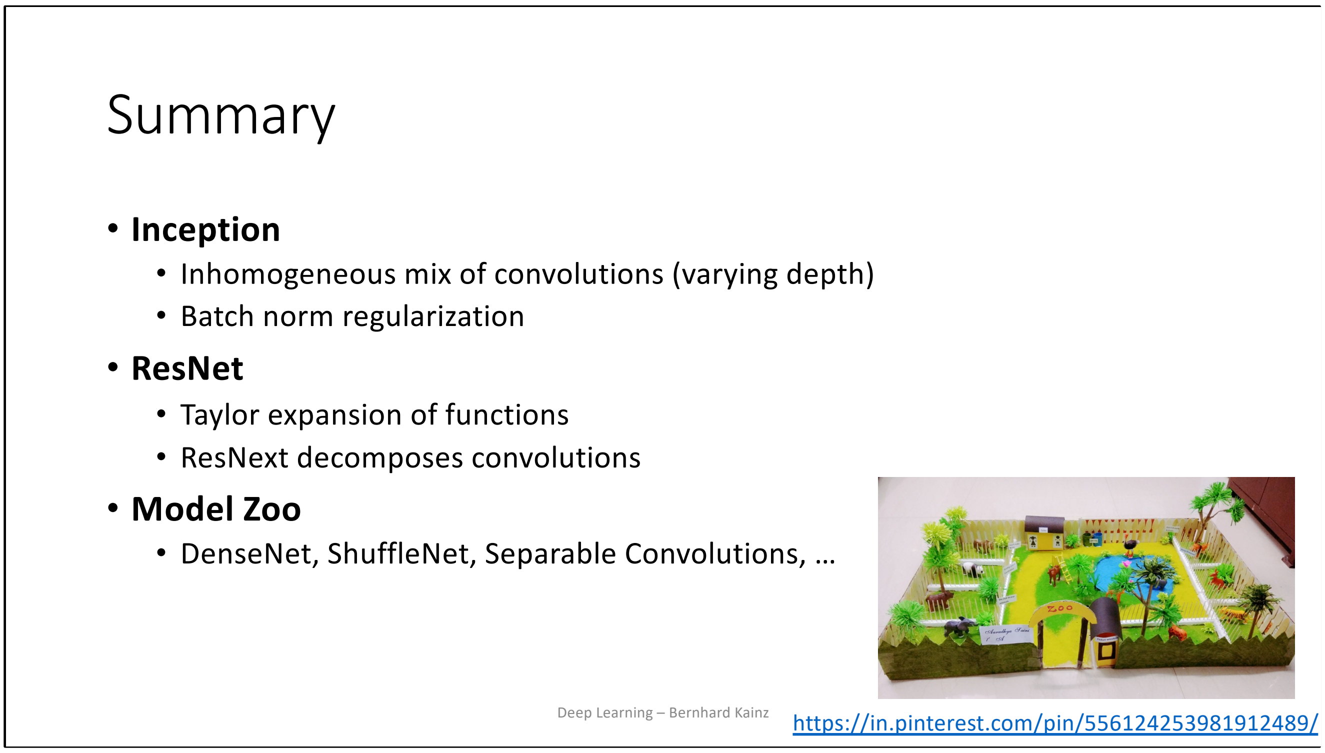
Things to explore

- AutoML (find best model architecture automatically Google Cloud AutoML)
- Hypernetworks (a network that proposes the weights for another network), also neural processes
- Networks with memory, e.g. kanerva machine
- Almost no new basic architectures accepted nowadays (see https://nips.cc/virtual/2020/public/cal_main.html NeurIPS 2020 programme, focuses on meta findings)
- Attention! (second part of the course)

Deep Learning – Bernhard Kainz

Let's now look at some intriguing directions and trends in the field of neural network architectures.

First on the list is AutoML, specifically Google Cloud AutoML, which automates the process of finding the best model architecture.
It takes the guesswork out of model selection, aiming to automatically find the most effective network for your specific problem.
Next, we have Hypernetworks, which are essentially networks that generate weights for another network.
These networks offer a higher level of abstraction and can be particularly useful in complex tasks.
We also have networks with memory features, like the Kanerva machine, which enables more dynamic and contextual decision-making.
Interestingly, it seems that the introduction of fundamentally new architectures has slowed down.
For instance, a glance at the NeurIPS 2020 program shows that the focus has shifted more towards meta-findings rather than novel basic architectures.
Finally, keep an eye out for the second part of this course where we'll delve into the concept of Attention, a mechanism that's been revolutionary in various tasks, especially in NLP but increasingly in vision tasks as well.

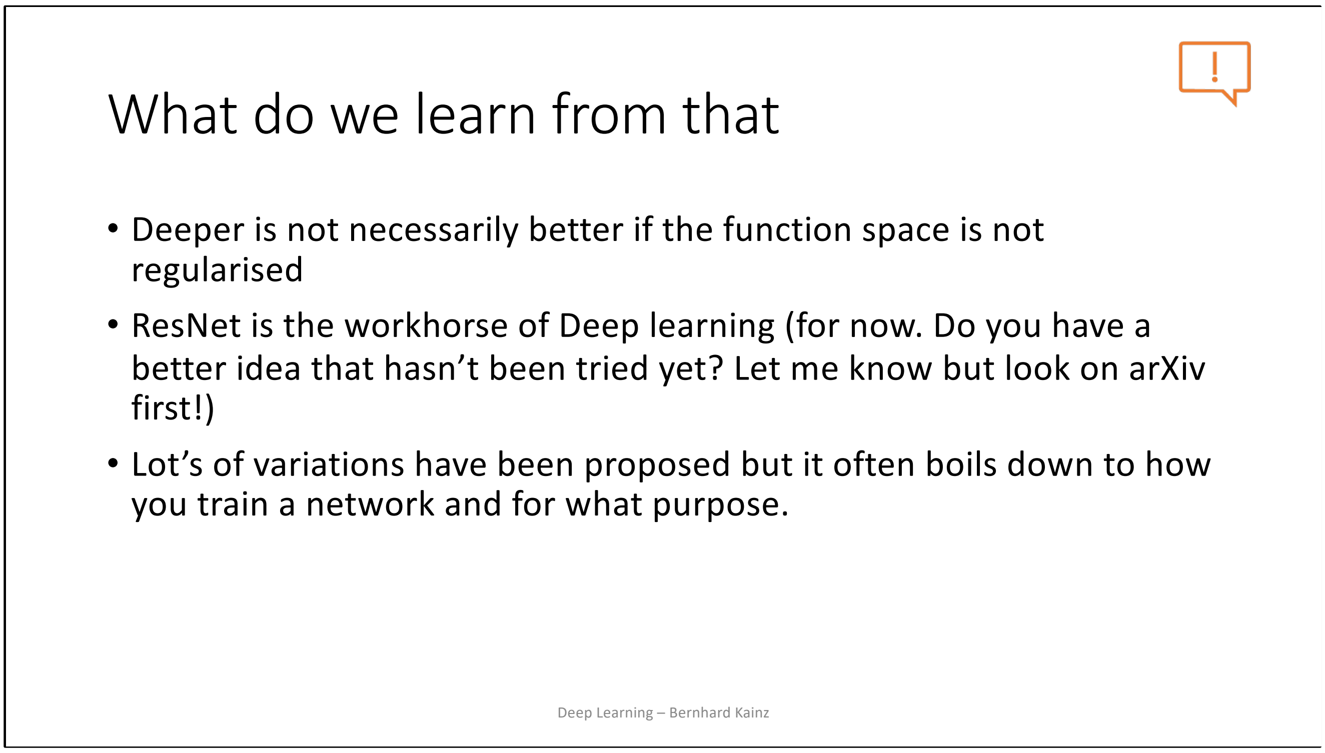


Summary

- **Inception**
  - Inhomogeneous mix of convolutions (varying depth)
  - Batch norm regularization
- **ResNet**
  - Taylor expansion of functions
  - ResNext decomposes convolutions
- **Model Zoo**
  - DenseNet, ShuffleNet, Separable Convolutions, …

https://in.pinterest.com/pin/556124253981912489/

Inception is notable for its inhomogeneous mix of convolutions, allowing the network to adapt to various scales and aspects of the input data.
Additionally, Inception employs batch normalization as a form of regularization, which aids in faster and more stable training.
On the other hand, ResNet introduces the idea of residual learning, which can be thought of as a form of Taylor expansion for neural networks.
It enables us to train very deep networks by ensuring more efficient gradient flow during backpropagation.
The architecture known as ResNeXt further extends ResNet by decomposing convolutions, offering even greater efficiency and performance.
Beyond these, there are various other noteworthy architectures like DenseNet, ShuffleNet, and Separable Convolutions, which bring their own unique innovations to the table.
In particular, SE-Net and ShuffleNet stand out for their innovative contributions to the field.
So, in essence, the landscape of CNN architectures is vast and ever-evolving, each with its own set of advantages tailored for different types of tasks and data.

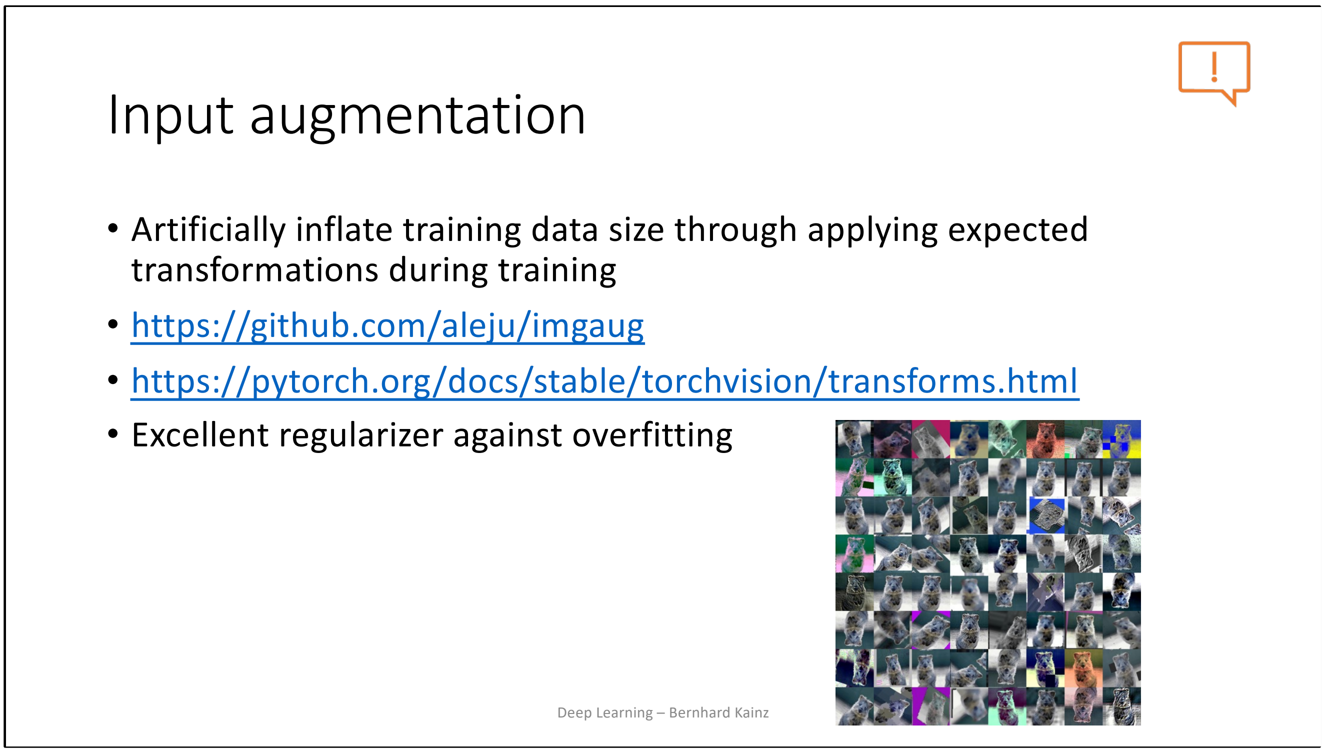


What do we learn from that

- Deeper is not necessarily better if the function space is not regularised
- ResNet is the workhorse of Deep learning (for now. Do you have a better idea that hasn’t been tried yet? Let me know but look on arXiv first!)
- Lot’s of variations have been proposed but it often boils down to how you train a network and for what purpose.

One takeaway is that "deeper" doesn't automatically equate to "better," especially if the function space isn't properly regularized.
Regularization techniques help in constraining the function space, thereby making the network generalize better to unseen data.
ResNet has undeniably become the workhorse of deep learning, at least for the time being.
If you think you've got a groundbreaking idea that could surpass ResNet, it might be worthwhile to check if similar concepts have already been published on arXiv.
It's important to note that while various modifications and improvements on architectures like ResNet have been proposed, the effectiveness often comes down to how well you train the network and for what specific purpose.
So, the architecture is just one piece of the puzzle; the training regime and the problem you're tackling are equally crucial.

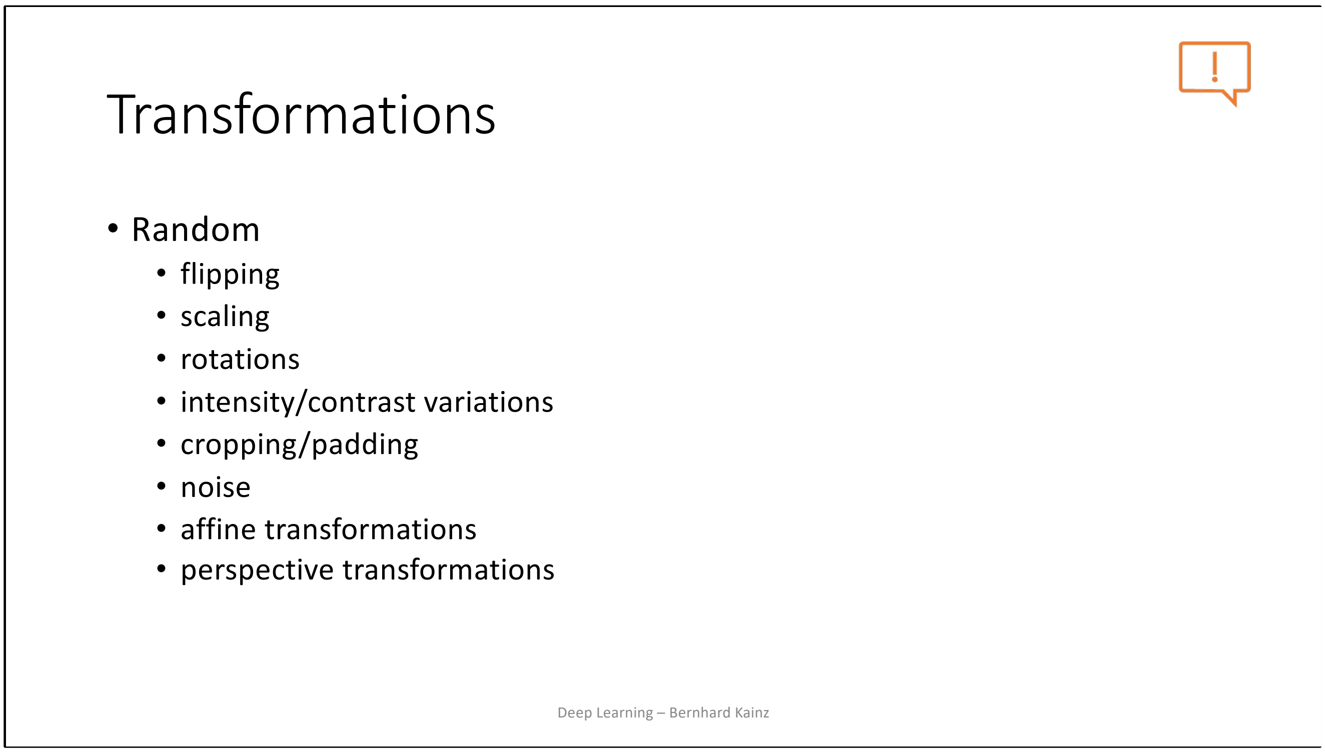
Data Augmentation



Input augmentation

- Artificially inflate training data size through applying expected transformations during training
- https://github.com/aleju/imgaug
- https://pytorch.org/docs/stable/torchvision/transforms.html
- Excellent regularizer against overfitting

Input data augmentation is a technique used to increase the size and diversity of your training set.
It does this by applying a series of random but realistic transformations to each data point during training.
While it might seem tangential to the topic of loss functions, data augmentation is actually a vital tool in deep learning.
By increasing the size of your training data artificially, you're effectively adding more "experiences" for your model, which can improve generalization.
Data augmentation can be done through various libraries, including imgaug for image data and PyTorch's torchvision.transforms for both image and non-image data.
You can find code samples and tutorials on their respective websites.
Here are some useful resources: imgaug GitHub repository and PyTorch Transforms Documentation.
One of the main benefits of data augmentation is its role as a regularizer.
It helps prevent overfitting by ensuring that the model encounters a variety of different, yet plausible, examples during training.
Do not underestimate the power of data augmentation.
It complements the choice of your loss function and serves to make your deep learning model more robust and adaptable.



Transformations

- Random
  - flipping
  - scaling
  - rotations
  - intensity/contrast variations
  - cropping/padding
  - noise
  - affine transformations
  - perspective transformations

Deep Learning – Bernhard Kainz

Input data augmentation involves applying a variety of transformations to the original data.
These transformations are usually random and aim to mimic plausible variations that the model may encounter in the real world.

Random flipping can simulate the natural orientations of objects or scenes in images.
For example, a cat can be oriented left or right in a photo, and flipping allows your model to recognize both.

Scaling is another useful transformation.
It helps the model generalize across different sizes of the same object or feature.

Rotations add another layer of complexity by altering the angle of the data points.
This is especially useful in tasks like object recognition, where orientation can vary widely.

Intensity and contrast variations help the model adapt to different lighting conditions and image qualities.
These are crucial for image processing tasks, such as medical imaging or outdoor scene recognition.

Cropping and padding alter the focus and frame of the data points.
This can be helpful for tasks where the subject can be off-center or partially visible.

Adding noise to the data serves as an effective way to improve the model's robustness.
It mimics real-world scenarios where data may not always be clean or noise-free.

Affine transformations include operations like translation, scaling, and shearing.
These offer another way to introduce variability into your data, helping your model generalize better.

Perspective transformations alter the viewpoint of the object or scene.
This helps in applications like augmented reality, where the perspective can dramatically change the appearance of objects.

These diverse transformations enable your model to generalize better and reduce the likelihood of overfitting.
Data augmentation, when used correctly, complements your chosen loss function to create a more robust learning process.