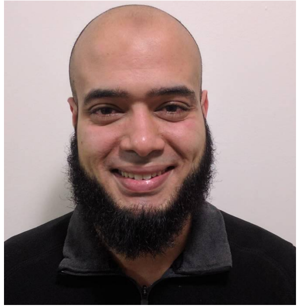
Machine Learning

Convolutional Neural Networks

Mostafa S. Ibrahim
Teaching, Training and Coaching for more than a decade!

Artificial Intelligence & Computer Vision Researcher
PhD from Simon Fraser University - Canada
Bachelor / MSc from Cairo University - Egypt
Ex-(Software Engineer / ICPC World Finalist)

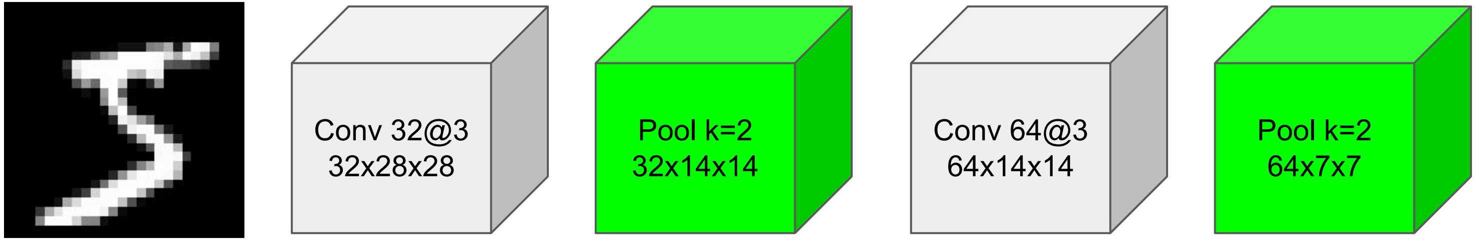


Convolutional Neural Networks (CNNs)

- CNN deep networks process structured grid data (images, videos, etc)
- A CNN stacks series of convolutional and 2D-pool layers that keep **transforming** and **reducing** the input feature map to a **target** map
  - Intermediate layers can be added. Popular choices
    - Activations, mostly RELU-based / Dropout and Batchnorm layers
- Again our goal, we can **reduce** a huge image input 3x256x256 into 128x7x7
  - Then: we utilize this reduced output of only 7x7 resolution (with rich 128 channels)
    - Exact final values up to the design
  - We typically resize big images or crop them into (e.g. into 256x256 or 224x224)
  - Otherwise, we will have a huge number of convolutional layers to reduce to the goal map

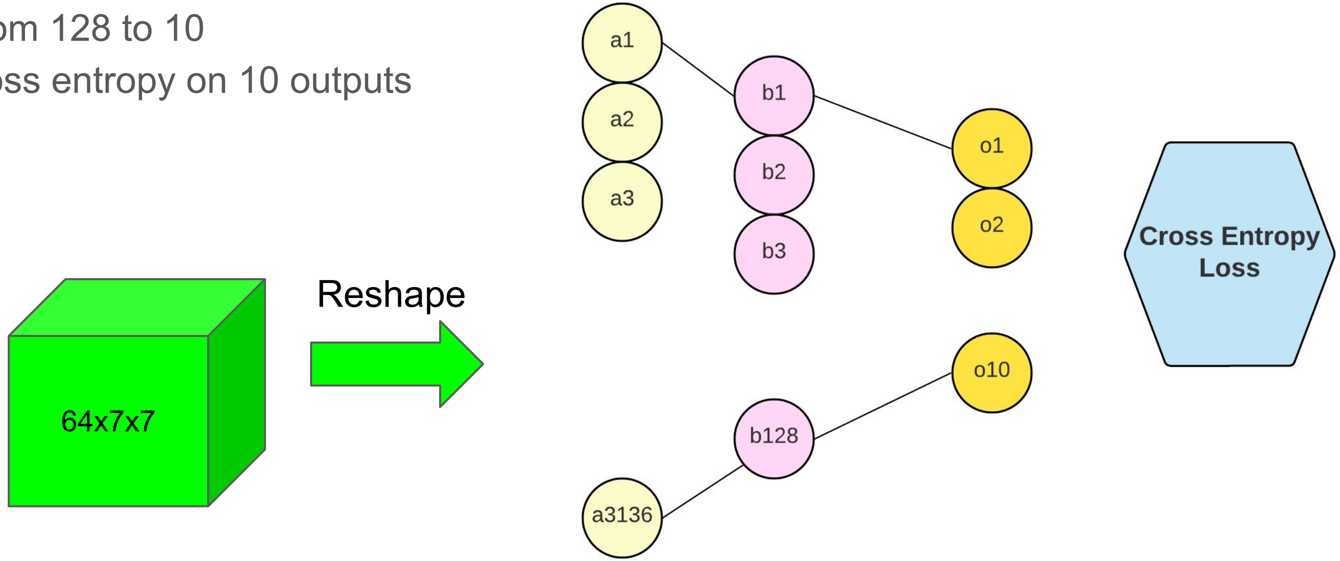
MNIST multiclass classifications

- Assume we would like to classify MNIST digit image (1x28x28)
- If we applied
  - 2D conv layer (1⇒32) with kernel(3x3) and padding=same, we get output map: 32x28x28
  - Then pool it with 2x2 non-overlapping kernel to get 32x14x14 ⇒ then apply RELU
  - Repeat. 2D conv (32⇒64) to get output map: 64x14x14
    - Then 2D pool followed by RELU to get 64x7x7
- Now, these 64*7*7 = 3136 are a strong transformed representation for input
  - Now apply a **normal 2-layers classical NN** on this transformed input



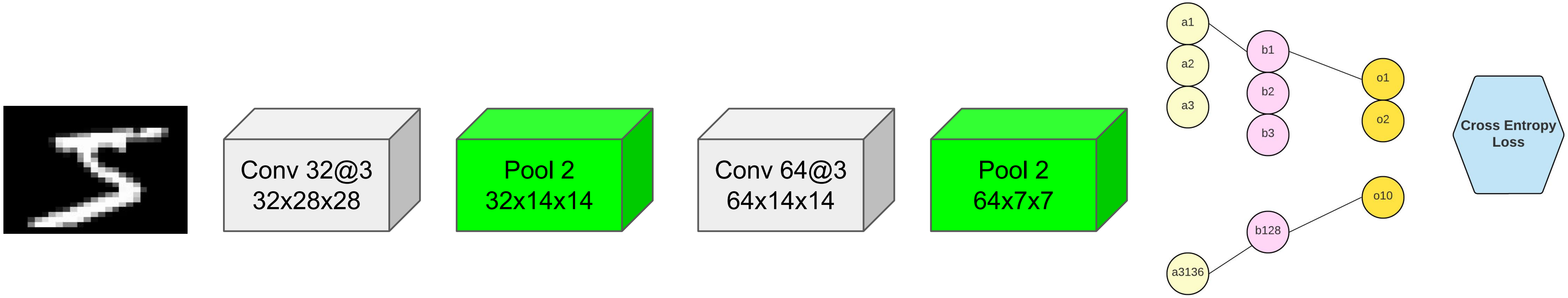
Fully Connected Layer

- The CNN blocks are convolutional ends with feature maps Cxwxh
- To move the output toward our goal, we can flatten this as Cxwxh nodes
- Then we feed this input on 2 dense layers that maps to our goal
    - One layer to map 3136 to 128
    - Another layer from 128 to 10
    - Softmax and cross entropy on 10 outputs

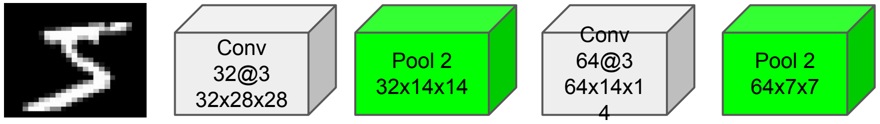


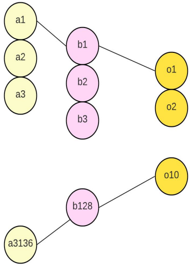
Big Picture

- So overall, start with an image
- CNN blocks
    - Use convolution layer to extract multiple features from the input
    - Use pooling to reduce the spatial size (more local context + less future computations)
    - Apply non-linear activations such as RELU
- After the last CNN block: apply 2 FC layers followed by the loss



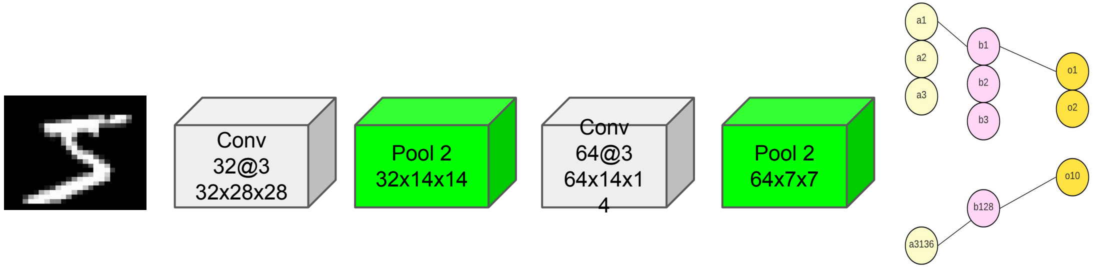
Network Initialization






```python
class MnistCNN(nn.Module):
    def __init__(self):
        super(MnistCNN, self).__init__()
        # input is 1x28x28, so input channels = 1
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding='same')
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding='same')

        # we end with 64 channels. We pool twice so 28 => 14 => 7 for dimensions
        self.last_conv_length = 64 * 7 * 7
        self.fc1 = nn.Linear(self.last_conv_length, 128)
        self.fc2 = nn.Linear(128, 10)
        self.activation = nn.functional.relu
```

Network Feedforward




```python
def forward(self, x):
    # input x: [B, 1, 28, 28]
    x = self.conv1(x)              # [B, 32, 28, 28]
    x = self.activation(x)
    x = self.pool(x)               # [B, 32, 14, 14]   28/2 = 14

    x = self.conv2(x)
    x = self.activation(x)         # [B, 64, 14, 14]
    x = self.pool(x)               # [B, 64, 7, 7]     14/2 = 7

    # Reshape Linearly the last layer
    x = x.view(-1, self.last_conv_length)   # [B, 3136]  where 3136 = 64*7*7
    x = self.fc1(x)                         # [B, 128]
    x = self.activation(x)
    # don't add activation after it for classification (e.g. softmax/sigmoid)
    x = self.fc2(x)                         # [B, 10]
    return x
```

- **AdamW** is one of the best optimizers that works on so many tasks
- This loss receives **logits** and internally applies **softmax**
- model.parameters(): represents **all the weights** of the network

```python
model = MnistCNN()
criterion = nn.CrossEntropyLoss()
optimizer = optim.AdamW(model.parameters(), lr=0.001)

# Model Training
num_epochs = 10
for epoch in range(num_epochs):
    epoch_loss = 0.0
    for inputs, labels in trainloader:
        outputs = model(inputs)

        optimizer.zero_grad()
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        epoch_loss += loss.item()
    print(f"Epoch {epoch + 1}, Loss: {epoch_loss / len(trainloader)}")
```

Network Evaluation

- Evaluation is direct, but we need to be careful and use evaluation settings

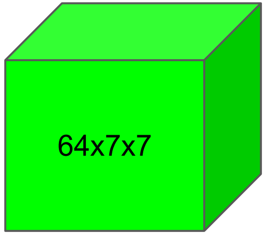
```
correct, total = 0, 0
model.eval()
with torch.no_grad():
    for inputs, labels in testloader:
        outputs = model(inputs)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()

accuracy = 100 * correct / total
print(f"Accuracy on the test set: {accuracy}%")
```

Spatial Structure

- One great advantage about convolutional layers, they keep the spatial information of the input grid (input and output are 2D)
- FCs in the previous example works well
- However, in some cases losing the structure
  - is not acceptable in some tasks, such as in semantic segmentation
  - Is not the best performance, sometimes in classifications
- So the question how can we solve this problem?
  - Keep using conv2d and pool2d layers!
  - Let’s see how we can do that

Replacing first FC

- In FC solution, we ended up with 2 FC layers
  - From 64x7x7 to 128
  - From 128 to 10
- So first, let's add another convolution to count for the first FC layer
  - Let's use 128 filters with k = 3 and valid padding (to reduce width)
  - This generates 128x5x5
- Use pooling to reduce it into 128x1x1
  - Adaptive pooling can help us to pool to target resolution



Replacing second FC

- Now we have 128x1x1
- What if just applied conv2d with kernel = 1 and 10 output filters
- Now this produces 10 logit values
- Please notices, both 128 and 10 values are strongly connected to the 2D **spatial** grid (there is spatial relationship between them)
    - Normal FC by definition just ignore the order of input nodes
        - Any intial order has the same output (order invariant)

New Network

```python
class MnistCNN(nn.Module):
    def __init__(self):
        super(MnistCNN, self).__init__()
        # input is 1x28x28, so input channels = 1
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding='same')
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding='same')
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding='valid')

        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        # Max pool the last 2 dimensions into [B, C, 1, 1]
        self.adaptivepool = nn.AdaptiveMaxPool2d(1)  # or AdaptiveAvgPool2d
        self.conv4 = nn.Conv2d(128, 10, kernel_size=1)

        self.activation = nn.functional.relu
```

New feedforward

```
def forward(self, x):
    x = self.conv1(x)               # [B, 32, 28, 28]
    x = self.activation(x)
    x = self.pool(x)                # [B, 32, 14, 14]   28/2 = 14

    x = self.conv2(x)
    x = self.activation(x)          # [B, 64, 14, 14]
    x = self.pool(x)                # [B, 64, 7, 7]     14/2 = 7

    x = self.conv3(x)               # [B, 128, 5, 5]    7-2 due to padding
    x = self.activation(x)
    x = self.adaptivepool(x)        # [B, 128, 1, 1]    pool 5x5

    x = self.conv4(x)               # [B, 10]           1x1 conv
    x = x.view(x.size(0), -1)
    return x
```

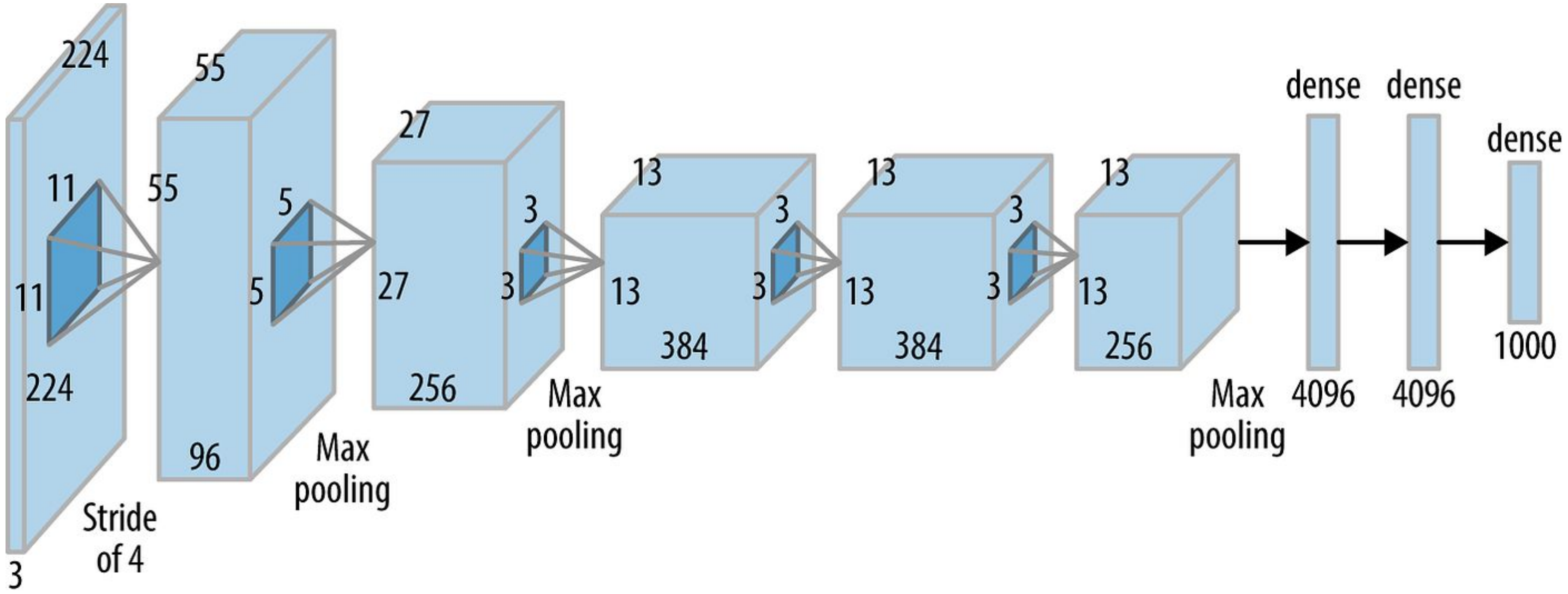
1x1 Conv

- The last 1x1 convolution looks the only possible way due to size Cx1x1
- However 1x1 conv plays a **big role** even on general maps CxWxH
- One common use of 1x1 convolutions is to **reduce the number** of channels in the feature maps.
    - For example input is 512x32x32. With 1x1 conv and 64 filters you get 64x32x32
        - Observe this is less memory than 3x3 kernel and faster
    - Observe: 1x1 focus on a single pixel and doesn’t consider neighbours
- Use case 1: after concatenation 2 big maps from different branches reduce their size with 1x1 conv
- Use case 2: create several 2D conv with different kernels then concatenate and reduce as mentioned (Google's InceptionNet) to see different scales

AlexNet

- “AlexNet is a convolutional neural network that was designed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton.
- The network won the 2012 **ImageNet** Large Scale Visual Recognition Challenge, a competition that aimed at encouraging the development of algorithms for object detection and image classification.
- The success of AlexNet is often credited with revitalizing interest in deep learning and convolutional neural networks (CNNs) in the field of machine learning.”

AlexNet


224
55
27
13
13
13
dense
dense
dense
11
55
5
3
3
3
11
5
27
3
13
3
13
3
13
224
96
256
384
384
256
1000
Stride of 4
3
Max pooling
Max pooling
Max pooling
4096
4096

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

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

AlexNet Key Achievements

- Trained on **large scale data**
  - trained on two NVIDIA GTX 580 GPUs making effective GPUs utilizations
- **ReLU Activation**: ReLU activation function is computationally more efficient than tanh and sigmoid
  - Also minimizes the risk of vanishing gradient problem
- **Dropout**: layers for **regularization**, reducing the risk of overfitting.

After AlexNet

- The success of AlexNet sparked significant interest in deep learning and led to the development of a variety of architectures for image classification tasks
- Many of of these new achievements managed to have deeper and more complex network that performs better
    - We learned several tricks from these networks
- 2014 **VGG** (Utilizes small 3x3 convolutional filters)
- 2014 **GoogLeNet** / Inception: deeper but had fewer parameters
- 2015 ResNet (Residual Networks) - Microsoft - 168k citations
    - Important and still relevant breakthrough. Introduced **residual** connections to allow **gradients** to flow through many layers, making it possible to train **very deep networks.**
    - Variants include ResNet-50, ResNet-101, and ResNet-152.

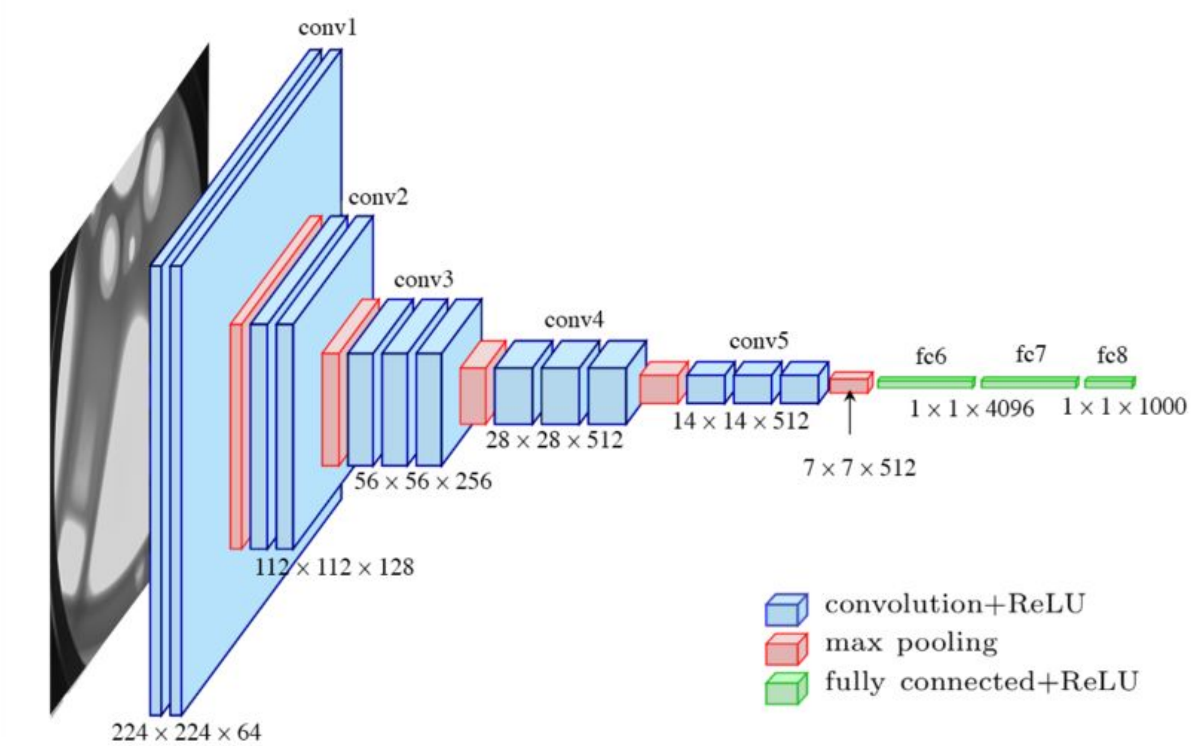
After AlexNet

- 2017 DenseNet: Connects each layer to every other layer
- 2017 MobileNets: Designed for mobile and embedded vision applications
  - Utilizes depthwise separable convolutions to reduce the number of parameters
  - Still widely used - has many variants
- 2019 EfficientNet: Scales the network width, depth, and resolution efficiently
- 2020 Vision Transformer (ViT):
  - Game changer. Applies transformer architectures instead of classical CNNs styles
- 2021 Swin Transformer: local and global attention

Transformers in vision

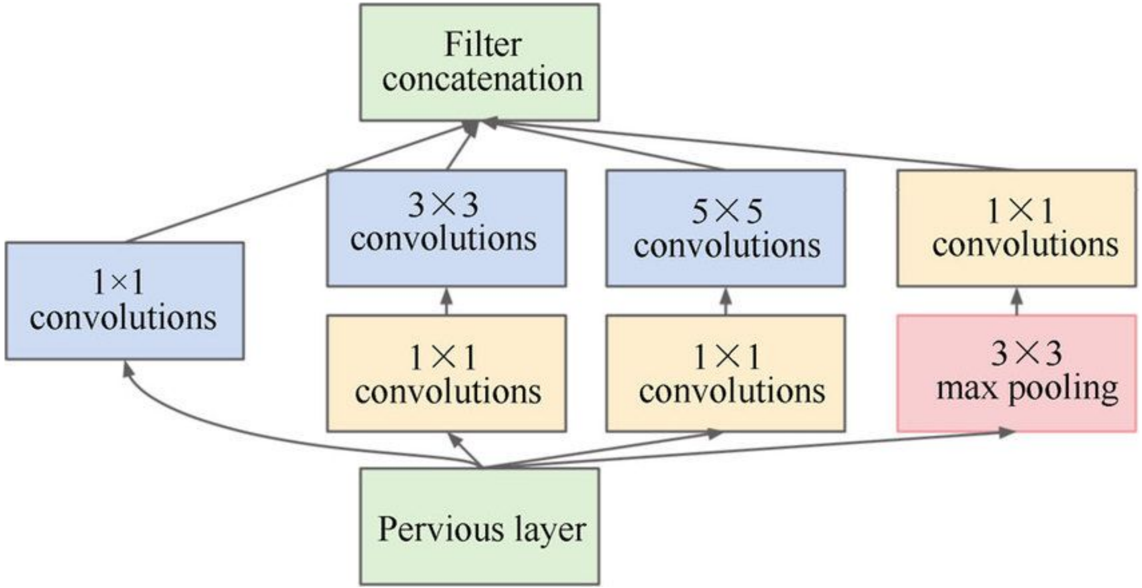
- There are many transformers in computer vision field
  - Vision Transformer (ViT):
  - Swin Transformer
  - DETR (DEtection TRansformer)
  - MLP-Mixer
  - ConViT (Convolutional Vision Transformer)
  - CrossViT
  - T2T-ViT (Tokens-to-Tokens Vision Transformer)
  - LeViT (Lightweight Vision Transformer)

VGG Net



Img src

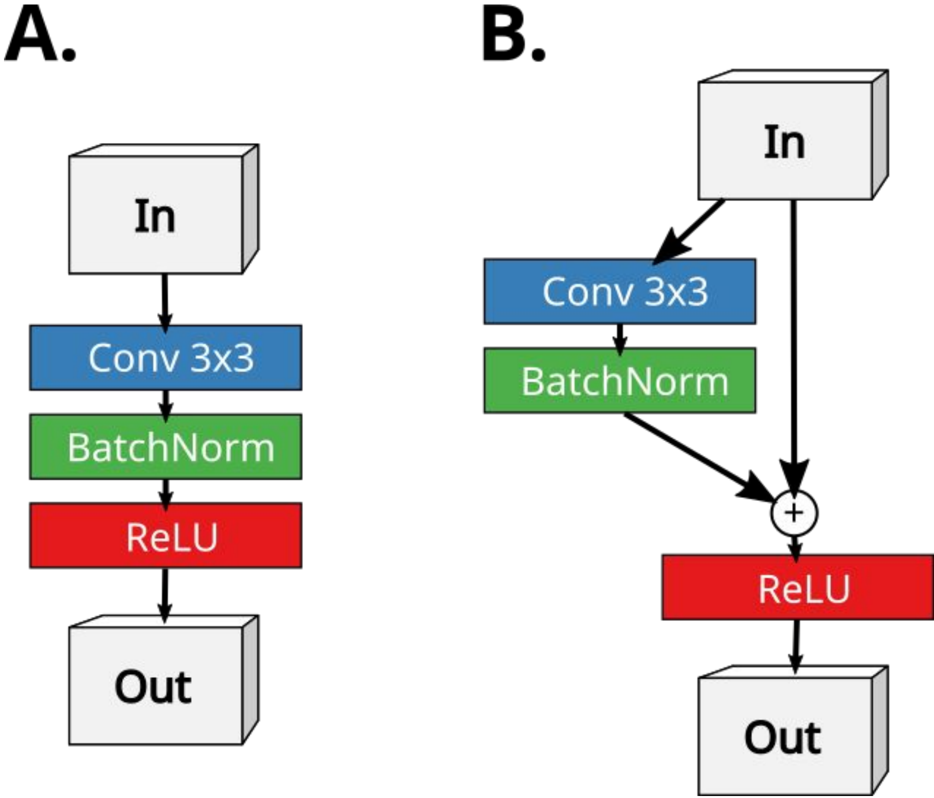
GoogleNet inception Module



Img src

ResNet Block

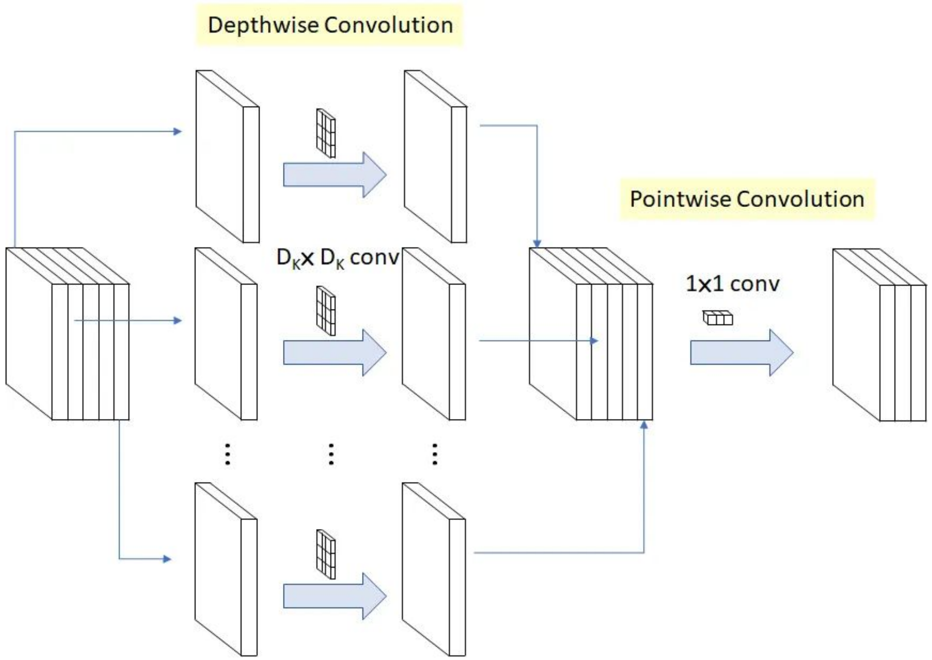
- Typical
  - H(x) = relu(conv(x))
- Resnet
  - F(x) = relu(conv(x) + x)
- This added x (element-wise) is a great change for gradients



Img src

Mobilenet

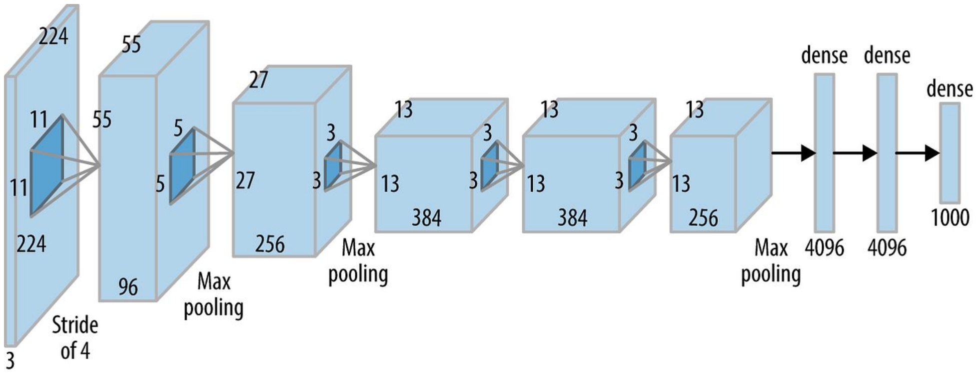
- Depthwise convolution + 1x1 conv can reduce computations



Img src

CNN Learned Features

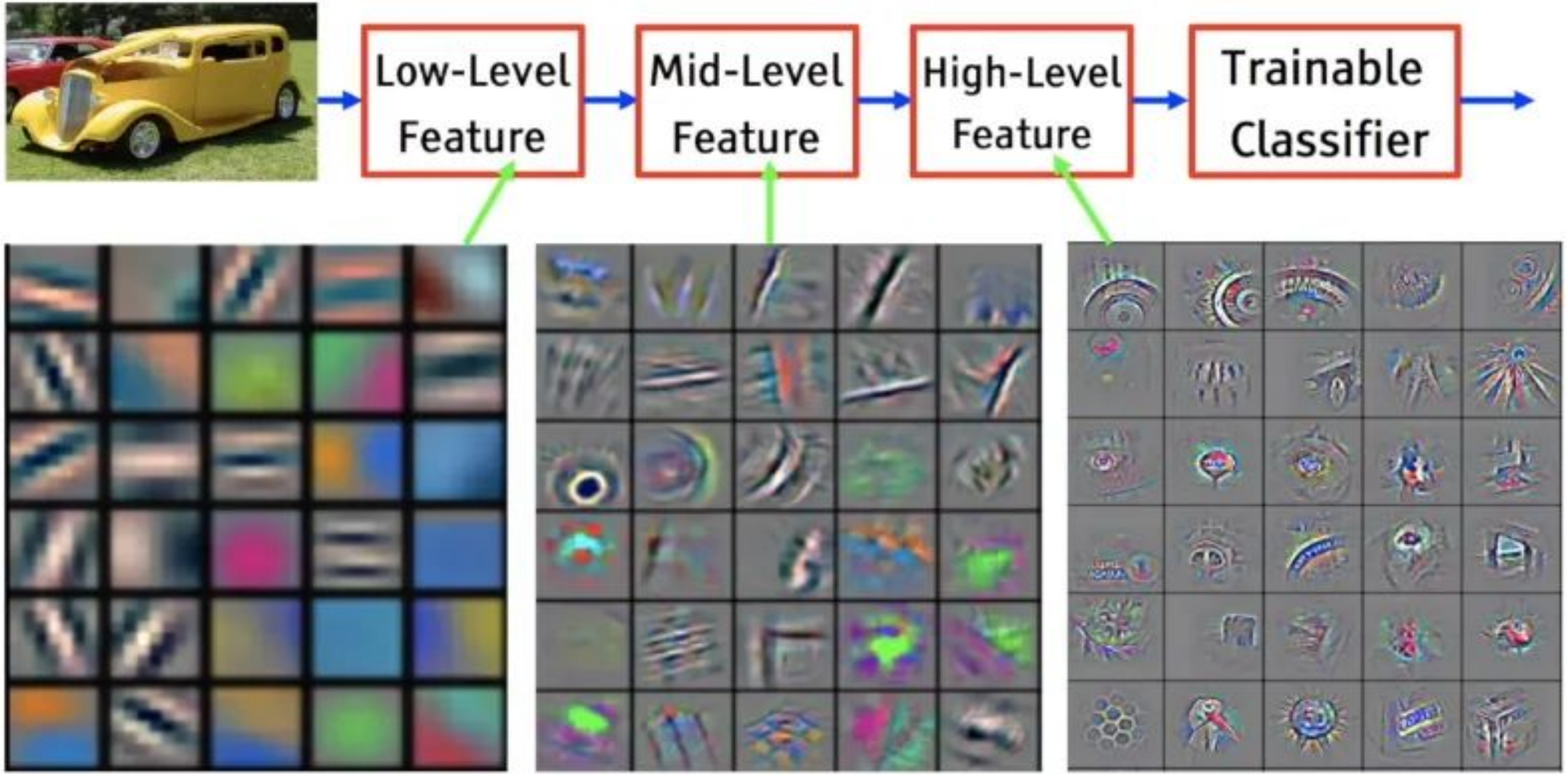
- What kind of features are learned by our layers in a deep network?
- To keep it simple, we may think in the layers as of 3 levels
- **Early layers**
    - Learn low-level features
- **Intermediate layers**
    - Learn mid-level features
- **Late layers**
    - Learn high-level features



CNN Learned Features

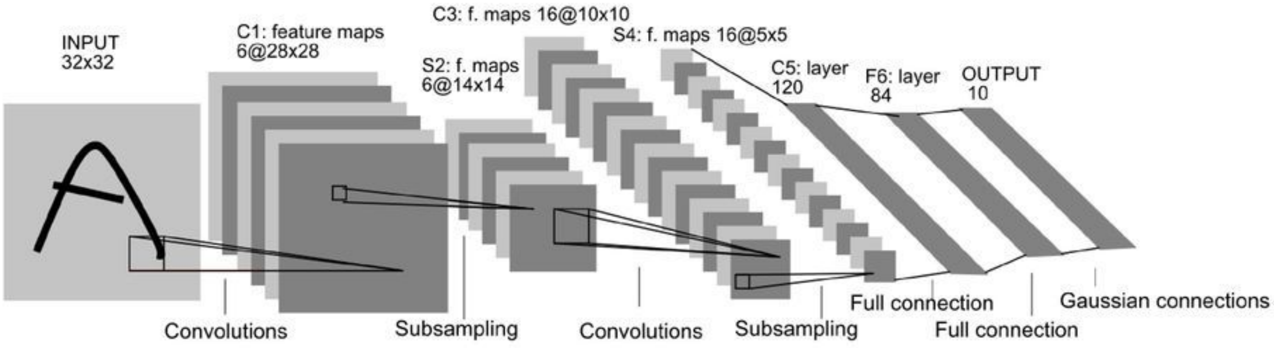
- Early layers
  - A 3x3 kernel observe a very small region (local scope) for input like 224x224
  - Maybe we learn **low-level features** such as edges, corners, angles, textures, colors, etc
- Intermediate Layers
  - With more pooling layers, we get a smaller resolution, e.g. 56x56   (224/4 = 56)
    - Each pixel is actually corresponding to 4x4 region
  - Now a 3x3 kernel sees 12x12 region
  - Maybe we learn **mid-level features:** shapes, complex textures, part of the object (ear/eye)
- Late layers
  - We may end up with a resolution like 7x7   (a pixel represents 32x32)
  - Now a 3x3 kernel sees 63x63 region
  - We learn **high-level features** and object parts
- CNNs learn **abstract features** from pixels

CNN Feature Visualization



History: LeNet 5 - Yann LeCun *et al*

- The paper "Gradient-Based Learning Applied to Document Recognition" is a seminal work by Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner that was published in **1998**. This influential paper is often credited with advancing the development of convolutional neural networks (CNNs) and their application to document recognition



Model Latency

- Sometimes you would like to measure how much time a single example/epoch takes
  - Please notice in the first iteration there could be things that are initialized / cached, so this run is bigger and MUST be excluded
  - Feed e.g. 100 batches and compute the average
- For proper **benchmarking**, you may use torch.cuda.**synchronize**()
  - When you call torch.cuda.synchronize(), it forces the CPU to wait until all the CUDA kernels have finished processing on the GPU before the CPU proceeds with the execution of subsequent code. By default, most CUDA operations are asynchronous with respect to the host CPU. This means that the CPU can queue up a series of CUDA operations and then go on to other tasks while the GPU is still working.
- See attached code sample

Reporting a model

- Your manager expects from you several insights so be ready
- What is the used model? **What** are the other alternatives?
  - Pros and cons? **Why** did you select the current one?
- What are the total number of parameters?
- How much time do you need for training? How many GPUs?
- What is the model latency?
- What is the performance on ML/Business metrics?
  - Comparison with available SOTA models?
- What is the used dataset? How many examples for train/val?

Relevant Materials

- CNN: Link, Link
- Deep Learning Memory Usage and Pytorch Optimization Tricks

"Acquire knowledge and impart it to the people."

"Seek knowledge from the Cradle to the Grave."