Convolutional Neural Networks

Ruwen Qin
Stony Brook University

November 5, 2025

Data used in this module:

- NNData

Python notebook used in this module

- Convolutional_Neural_Networks.ipynb

1 Introduction

Many types of data, such as images, are naturally represented in structured grids where nearby elements are related. A common but limited approach is to flatten such a structure into a one-dimensional vector for use in fully connected multilayer perceptrons (MLPs), which ignore the relationships within the data. Convolutional neural networks (CNNs) address this limitation by preserving and leveraging local patterns, making them well-suited for learning from data with inherent spatial or sequential structure. CNNs are computationally efficient because they use fewer parameters than fully connected networks and can be easily parallelized on GPUs. Their efficiency and flexibility have led to widespread adoption across various domains, including images, audio, text, time series, and even graph-structured data. In this learning module, we will walk through the basic operations that comprise the foundation of all convolutional networks. This lecture note is mainly adopted from the references [1, 2].

2 From Fully-Connected Layers to Convolution

Imagine that we want to detect an object in an image. It seems reasonable that whatever method we use to recognize objects should not be overly concerned with the precise location of the object in the image, as well as the object's position. We should recognize a car doesn't matter where it appears in the image, and what direction is it going, in Figure 1. CNNs systematize this idea of spatial invariance, exploiting it to learn useful representations with fewer parameters.

Figure 1: Cars in traffic (source: from Internet)

Meanwhile, the edges, textures, and shapes that define each car in Figure 1 are represented by groups of nearby pixels working together, rather than relationships across the entire image. Therefore, to recognize a car or the traffic pattern, a learning system can focus on local regions first, capturing patterns that are spatially close, before considering broader relationships across the image.

In summary: to design neural networks for computer vision, we follow two key principles:

- *Translation invariance* implies that all patches of an image will be treated in the same manner.
- *Locality* means early layers focus on small, local regions before combining information across the image. Deeper layers should be able to capture longer-range features of the image, in a way similar to higher level vision in nature.

These principles motivate the design of CNNs to be introduced in this learning module.

Images are not two-dimensional objects but rather three dimensional tensors, characterized by a height (H), width (W), and channel (C). RGB images have three channels: Red, Green, and Blue. Many satellite images, in particular for agriculture and meteorology, have tens to hundreds of channels, generating hyper-spectral images instead. They report data on many different wavelengths. While the first two of these axes concern spatial relationships, the third can be regarded as assigning a multidimensional representation to each pixel location. We thus index the input $\mathbf{X}$ as $[\mathbf{X}]_{i,j,k}$ and, accordingly, the hidden layer $\mathbf{H}$ is also in three dimensions. The convolution kernel (also named filter) $\mathbf{V}$ and the bias $\mathbf{U}$ for aggregating inputs can be in three dimensions as well.

2.1 Translation Invariance

Translation invariance implies that a shift in the input $\mathbf{X}$ should simply lead to a shift in the hidden representation $\mathbf{H}$. This is only possible if the kernel $\mathbf{V}$ and bias $\mathbf{U}$ do not depend on the location (i, j, k). As such, we can simplify the definition for $\mathbf{H}$:

$$[\mathbf{H}]_{i,j,k} = u + \sum_a \sum_b \sum_c [\mathbf{V}]_{a,b,c}[\mathbf{X}]_{i+a,j+b,k+c}. \tag{1}$$

This is a convolution! We are effectively weighting pixels at $(i + a, j + b, k + c)$ in the vicinity of location (i, j, k) with coefficients $[\mathbf{V}]_{a,b,c}$ to obtain the value $[\mathbf{H}]_{i,j,k}$. Note that the kernel $[\mathbf{V}]_{a,b,c}$ and the bias u are independent of location, thus needing many fewer coefficients than otherwise.

2.2 Locality

Locality means we should not have to look very far away from location (i, j, k) in order to glean relevant information for assessing what is going on at $[\mathbf{H}]_{i,j}$. This means that outside some range $|a| > \Delta$ or $|b| > \Delta$, we should set $[\mathbf{V}]_{a,b,c} = 0$. Equivalently, we can rewrite $[\mathbf{H}]_{i,j,k}$ as

$$[\mathbf{H}]_{i,j,k} = u + \sum_{a=-\Delta}^{\Delta} \sum_{b=-\Delta}^{\Delta} \sum_{c=-\Delta}^{\Delta} [\mathbf{V}]_{a,b,c}[\mathbf{X}]_{i+a,j+b,k+c}. \tag{2}$$

That is, $\mathbf{V}$ is a kernel with a fixed size of window.

3 Convolutions

3.1 Convolutional Layer

A convolutional layer picks convolution kernels with windows of a given size and weighs intensities accordingly. It cross-correlates the input and a kernel and adds a scalar bias to produce an output. The two parameters of a convolutional layer are the kernel and the scalar bias. When training a CNN model, we typically initialize the kernels randomly, just as we would with a fully connected layer.

3.2 Cross-Correlation Operation

Let's ignore channels for now and see how this works with two-dimensional data and hidden representations. Figure 2 shows a two-dimensional input tensor with a height of 3 and width of 3. The shape of the kernel window is given by the height and width of the kernel, and here it is 2×2. The shaded portions are the first output element (19) as well as the input and kernel tensor elements used for the output computation:

$$(0)(0) + (1)(1) + (3)(2) + (4)(3) = 19. \tag{3}$$

When shifting the kernel to the right by one pixel, we can calculate the output element, 25:

$$(1)(0) + (2)(1) + (4)(2) + (5)(3) = 25. \tag{4}$$

Moving the kernel to the left and down by a pixel, we can calculate the output element, 37:

$$(3)(0) + (4)(1) + (6)(2) + (7)(3) = 37. \tag{5}$$

Further shifting the kernel to the right by one pixel, we calculate the last output element, 43:

$$(4)(0) + (5)(1) + (7)(2) + (8)(3) = 43. \tag{6}$$

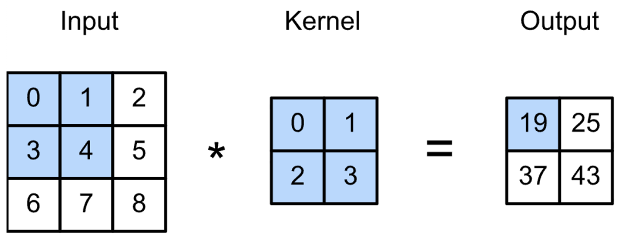


Figure 2: Cross-correlation Operation (source: from [2])

3.3 Feature Maps

The convolutional layer output in Figure 2 is sometimes called a *feature map*, as it can be regarded as the learned representations (features) in the spatial dimensions (e.g., width and height) to the subsequent layer. In CNNs, for any element of some layer, its *receptive field* refers to all the elements (from ALL the previous layers) that may affect the calculation of during the forward propagation. Note that the receptive field may be larger than the actual size of the input.

3.4 Padding

One tricky issue when applying convolutional layers is that we tend to under utilize pixels on the perimeter of our image, especially those in the corners. Figure 3 depicts the count of pixel utilization as a function of the convolution kernel size and the position within the image.

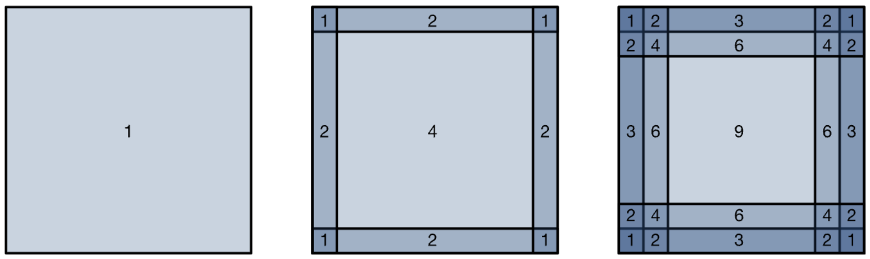


Figure 3: Padding (source: from [2])

Since we typically use small kernels, for any given convolution we might only lose a few pixels, but this can add up as we apply many successive convolutional layers. One straightforward solution to this problem is *padding*, which adds extra pixels of filler around the boundary of our input image, thus increasing the effective size of the image. Figure 4 pads a (3,3) image to become (5,5).

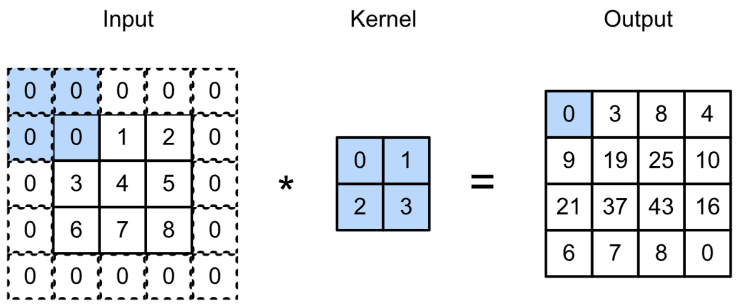


Figure 4: Padding (source: from [2])

3.5 Stride

Sometimes, either for computational efficiency or because we wish to downsample, we move the kernel more than one element at a time, skipping the intermediate locations. This is particularly useful if the convolution kernel is large since it captures a large area of the underlying image. We refer to the number of rows and columns traversed per slide as *stride*. Figure 5 illustrates an example where the stride is two along the axis of width and three along the axis of height.

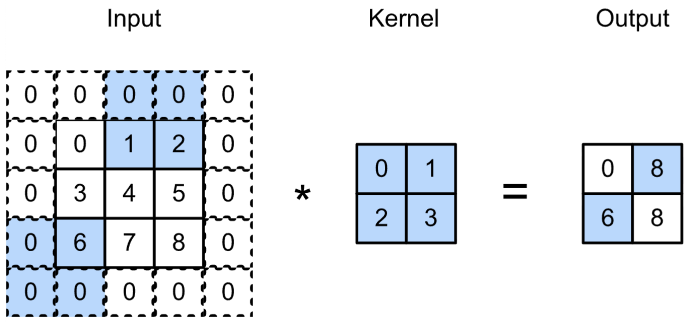


Figure 5: Stride (source: from [2])

In general, the output shape of a convolution is:

$$(\lfloor(n_{\mathrm{H}} - k_{\mathrm{H}} + 2p_{\mathrm{H}})/s_{\mathrm{H}}\rfloor + 1) \times (\lfloor(n_{\mathrm{W}} - k_{\mathrm{W}} + 2p_{\mathrm{W}})/s_{\mathrm{W}}\rfloor + 1). \tag{7}$$

where

- n_{H} and n_{W}: height and width of the input
- k_{H} and k_{W}: height and width of the kernel
- p_{H} and p_{W}: height and width of the padding
- s_{H} and s_{W}: height and width of the stride.

3.6 Multiple Channels

3.6.1 Multiple Input Channels

When we add channels into the mix, our inputs and hidden representations both become three-dimensional tensors. For example, each RGB input image has shape $C \times H \times W$. We refer to this axis, with a size of C, as the channel dimension.

Then the input data contains multiple channels, we need to construct a convolution kernel with the same number of input channels as the input data, so that it can perform cross-correlation with the input data. Figure 6 illustrates an example where the input tensor has two channels. Accordingly, a kernel with two channels is used. In calculating the element, 56, in the output tensor, the cross-correlation operation is:

$$[(1)(1) + (2)(2) + (4)(3) + (5)(4)] + [(0)(0) + (1)(1) + (3)(2) + (4)(3)] = 37 + 19 = 56. \tag{8}$$

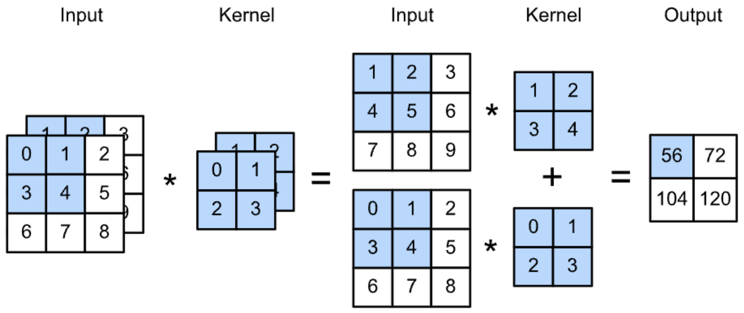


Figure 6: (source: from [2])

3.6.2 Multiple Output Channels

In many neural networks, we increase the number of channels as the network goes deeper, while reducing the spatial resolution. We can think of each channel as capturing different features of the input. However, channels do not work independently. Instead, they are learned together. So instead of one channel representing an edge detector, it's more accurate to say that edges are detected along certain combinations or directions in the space formed by all channels.

Denote by c_I and c_O the number of input and output channels, respectively, and by k_H and k_W the height and width of the kernel. To get an output with multiple channels, we can create a kernel tensor of shape $c_I \times k_H \times k_W$ for every output channel. We concatenate them on the output channel dimension, so that the shape of the convolution kernel is $c_O \times c_I \times k_H \times k_W$. In cross-correlation operations, the result on each output channel is calculated from the convolution kernel corresponding to that output channel and takes input from all channels in the input tensor. For example, if we would like to increase the number of output channels in Figure Figure 6 to there, shape of the kernel becomes $3 \times 2 \times 2 \times 2$.

3.6.3 1×1 Convolutional Layer

The only computation of the 1×1 convolution occurs on the channel dimension. Figure 7 shows the cross-correlation computation using the 1×1 convolution kernel with 3 input channels and 2 output channels. We could think of the convolutional layer as constituting a fully connected layer applied at every single pixel of spatial location of the input tensor to transform the corresponding input values into output values. That is, a 1×1 convolution layer learns pixel-wise combinations of features across channels.

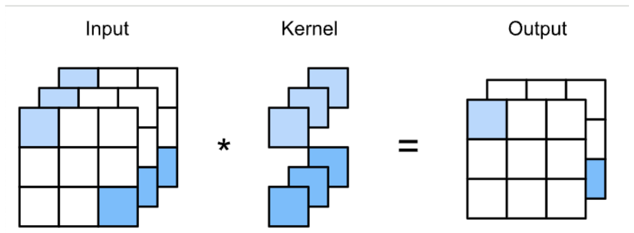


Figure 7: 1×1 convolution (source: [2])

3.7 Pooling

In many vision tasks, we need a global understanding of the image, for example, determining if it contains a car. CNNs achieve this by gradually aggregating spatial information through deeper layers, which have larger receptive fields and coarser feature maps. This process allows the network to capture global patterns while preserving local feature extraction at earlier stages. Pooling layers further help by reducing spatial resolution and making the representations less sensitive to objects' location and small shifts in the input, thus improving translation invariance.

Pooling is a simple operation. It aggregates results over a window of values. All convolution semantics, such as strides and padding apply in the same way as they did previously. However, pooling applies to each channel separately, leaving the number of channels unchanged.

3.7.1 Max- and Average pooling

Pooling operators consist of a fixed-shape window that is slid over all regions in the input according to its stride, computing a single output for each location traversed by the the pooling window. Unlike the cross-correlation computation, the pooling layer contains no parameters. That is, pooling operators are deterministic, typically calculating either the maximum or the average value of the elements in the pooling window. These operations are called max-pooling and average pooling, respectively.

Specifically, the average pooling operator averages over adjacent pixels to obtain an image with better signal-to-noise ratio. Max-pooling, as illustrated in Figure 8, selects the largest value within each pooling window. Using the maximum value helps preserve the strongly activated features, which are likely to indicate the presence of an important pattern. Therefore, in many cases, max-pooling, is preferable to average pooling. A popular choice of max-pooling window is (2,2) to quarter the spatial resolution of output.

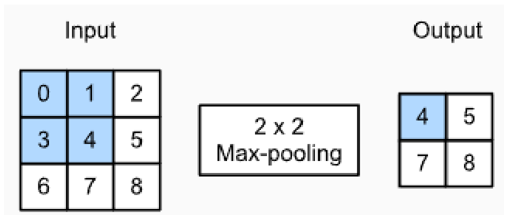


Figure 8: Max pooling (source: [2])

3.7.2 Stride and Padding for Pooling Layers

We can adjust the pooling operation to achieve a desired output shape by padding the input and adjusting the stride.

Since pooling aggregates information from an area, deep learning frameworks default to matching pooling window sizes and stride. For example, if we use a pooling window of shape (3, 3), we get a stride shape of (3, 3) by default. Let's consider an example, where the shape of an input tensor is (4, 4), the shape of the max-pooling layer is (2, 2), and the stride shape is chosen to be the same as the pooling window. According to Equation (7), the padding is zero.

Of course, the stride and padding can be manually specified to override framework defaults if required. For the same example discussed above, if the pooling window is chosen to be (3,3) and the stride shape is chosen to be (2,2), the padding size is one along each side. What would be the padding sizes if both the pooling window and the stride shape are (2,3) and the desired output shape is still (2,2)?

3.7.3 Multiple Channels

When processing multi-channel input data, the pooling layer pools each input channel separately, rather than summing the inputs up over channels as in a convolutional layer. This means that the number of output channels for the pooling layer is the same as the number of input channels.

4 LeNet: An Example of CNN

We now have all the ingredients required to assemble a fully-functional CNN. In this section, we will introduce LeNet introduced by Yann LeCun[3]. Figure 9 illustrates the LeNet's network architecture.

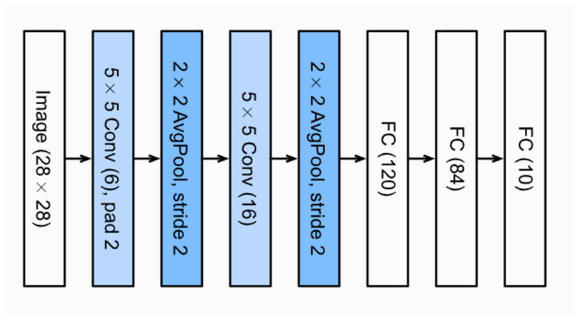


Figure 9: Network Architecture of LeNet-5 (source: [2])

We implement this LeNet for classifying the inspection images in NNdata. Our modification includes the number of channels for the input images (3 for RGB images) and the number of output (1 for binary classification).

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

    # define components of the network
    def __init__(self):
        super().__init__()
        # Convolutional layers
        self.conv1 = nn.Conv2d(in_channels=3, out_channels=6, kernel_size=5, stride=1,
    padding=2)
        self.conv2 = nn.Conv2d(in_channels=6, out_channels=16, kernel_size=5, stride=1)

        # pooling layer
        self.pool = nn.AvgPool2d(kernel_size=2, stride=2)

        # Fully connected layers
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 1)

        # Activation functions
        self.relu = nn.ReLU()
        self.sigmoid = nn.Sigmoid()

    # define the forward propagation process, which put network components in the desired
    sequence from input to output
    def forward(self, x):
        x = self.relu(self.conv1(x))
        x = self.pool(x)
        x = self.relu(self.conv2(x))
        x = self.pool(x)
        x = torch.flatten(x, 1)  # flatten all dimensions except batch
        x = self.relu(self.fc1(x))
        x = self.relu(self.fc2(x))
        x = self.sigmoid(self.fc3(x))
        return x
```

5 A Self-Designed CNN

We designed a CNN with two convolutional blocks followed by two fully-connected layers. Each convolutional block contains a convolutional layer followed by the ReLU activation function and then a maximum pooling layer. A dropout layer with 50% dropout probability is applied between the two fully-connected layers.

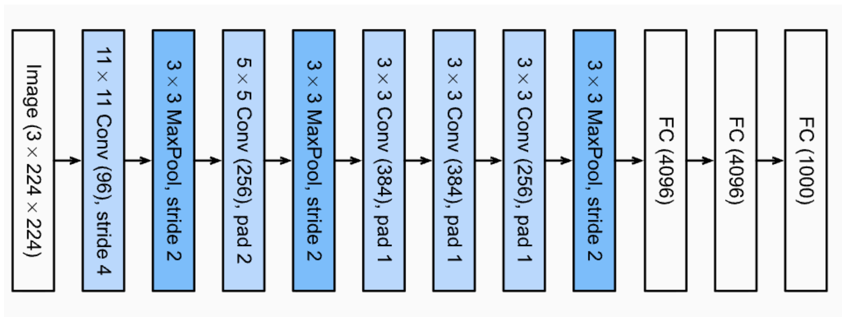


Figure 10: AlexNet (source: [2])

```
class SimpleCNN(nn.Module):
"""
A simplified CNN with two convolutional blocks and one fully connected layer.
Uses ReLU activations, max pooling, and dropout regularization.
"""
def __init__(self):
    super().__init__()
    # Convolutional layers
    self.conv1 = nn.Conv2d(in_channels=3, out_channels=64, kernel_size=3)  # -> 26x26x64
    self.conv2 = nn.Conv2d(in_channels=64, out_channels=32, kernel_size=3) # -> 11x11x32

    # Pooling layers
    self.pool1 = nn.MaxPool2d(kernel_size=2, stride=2)                     # -> 13x13x64
    self.pool2 = nn.MaxPool2d(kernel_size=2, stride=2)                     # -> 5x5x32

    # Fully connected layers
    self.fc1 = nn.Linear(32 * 5 * 5, 64)
    self.fc2 = nn.Linear(64, 1)

    # Dropout
    self.dropout = nn.Dropout(0.5)

    # Activation functions
    self.relu = nn.ReLU()
    self.sigmoid = nn.Sigmoid()

def forward(self, x):
    x = self.relu(self.conv1(x))
    x = self.pool1(x)
    x = self.relu(self.conv2(x))
    x = self.pool2(x)
    x = torch.flatten(x, 1)  # flatten all dimensions except batch
    x = self.relu(self.fc1(x))
    x = self.dropout(x)
    x = self.sigmoid(self.fc2(x))
    return x
```

6 Modern CNNs

In 2009, the ImageNet dataset was released [4], challenging researchers to learn models from 1 million examples, 1000 each from 1000 distinct categories of objects. The ImageNet team used Google Image Search to prefilter large candidate sets for each category and employed the Amazon Mechanical Turk crowdsourcing pipeline to confirm for each image whether it belonged to the associated category. ImageNet pushes computer vision and machine learning research forward. Below we overview a few deep CNNs developed.

6.1 AlexNet

AlexNet employs an 8-layer CNN, as Figure 10 shows. It won the ImageNet Large Scale Visual Recognition Challenge 2012. AlexNet showed that the features obtained by learning can transcend manually-designed features, breaking the previous paradigm in computer vision.

Below is the our implementation of AlexNet to the inspection image classification problem with modification of its input and output dimension.

```
AlexNet(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (9): ReLU(inplace=True)
    (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
  (classifier): Sequential(
    (0): Dropout(p=0.5, inplace=False)
    (1): Linear(in_features=9216, out_features=4096, bias=True)
    (2): ReLU(inplace=True)
    (3): Dropout(p=0.5, inplace=False)
    (4): Linear(in_features=4096, out_features=4096, bias=True)
    (5): ReLU(inplace=True)
    (6): Sequential(
      (0): Linear(in_features=4096, out_features=1, bias=True)
      (1): Sigmoid()
    )
  )
)
```

6.2 VGG

While AlexNet offered empirical evidence that deep CNNs can achieve good results, it did not provide a general template to guide subsequent researchers in designing new networks. The design of neural network architectures has grown progressively more abstract, with researchers moving from thinking in terms of individual neurons to whole layers, and now to blocks, repeating patterns of layers.

The idea of using blocks first emerged from the Visual Geometry Group (VGG) at Oxford University, in their VGG network ([5]). It is easy to implement these repeated structures in code with any modern deep learning framework by using loops and subroutines.

The key idea of VGG was to use multiple convolutions in between down-sampling via max-pooling in the form of a block, which was designed to address the limitations of rapid decrease of input images' spatial resolution in the traditional CNN architecture (i.e., a convolutional layer-nonlinearity- a pooling layer). Their research showed that deep and narrow networks significantly outperform their shallow counterparts. Stacking 3×3 convolutions has become a gold standard in later deep learning networks. Figure 11 illustrates the design of VGG. A VGG block consists of a sequence of convolutions with kernels with padding of 1 (keeping height and width) and nonlinearity followed by a max-pooling layer with stride of 2 (halving height and width after each block). Therefore, the number of convolutional layers and the number of output channels are precisely the arguments required to call a VGG block. The convolutional part of the network connects several VGG blocks in succession, as shown in Figure 11. As such, VGG defines a family of networks rather than just a specific manifestation. To build a specific network we simply iterate over arch to compose the blocks.

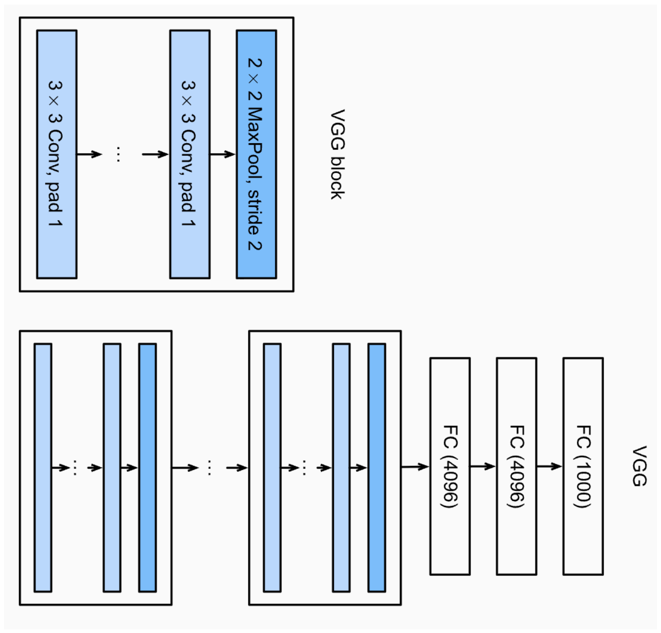


Figure 11: VGG (source: [2])

BBelow is our implementation of VGG-16 for the inspection image classification example. The VGG-16 architecture consists of five convolutional (VGG) blocks:

- **Blocks 1-2:** Each of these two blocks contains two convolutional layers with 3×3 kernels, padding of 1×1, and ReLU activation, followed by a max-pooling layer. The input and output channel sizes are $(3 \to 64)$ for the first block and $(64 \to 128)$ for the second block.
- **Blocks 3-5:** Each of the last three blocks contains three convolutional layers, also using 3×3 kernels, 1×1 padding, and ReLU activation, followed by a max-pooling layer. The output channel sizes of these blocks are 256, 512, and 512, respectively.

In total, the network includes 13 convolutional layers $(2 + 2 + 3 + 3 + 3 = 13)$, followed by three fully connected layers. This gives 16 layers with learnable parameters in total, hence the name **VGG-16**. There are other versions of VGG models, such as VGG-11, VGG-13, and VGG-19.

```
VGG(
  (features): Sequential(
    (0): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): Conv2d(64, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (3): ReLU(inplace=True)
    (4): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (5): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (6): ReLU(inplace=True)
    (7): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (8): ReLU(inplace=True)
    (9): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (10): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (11): ReLU(inplace=True)
    (12): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (15): ReLU(inplace=True)
    (16): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (17): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (18): ReLU(inplace=True)
    (19): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (20): ReLU(inplace=True)
    (21): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (22): ReLU(inplace=True)
    (23): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (24): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (25): ReLU(inplace=True)
    (26): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (27): ReLU(inplace=True)
    (28): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (29): ReLU(inplace=True)
    (30): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (avgpool): AdaptiveAvgPool2d(output_size=(7, 7))
  (classifier): Sequential(
    (0): Linear(in_features=25088, out_features=4096, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=4096, out_features=4096, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Sequential(
      (0): Linear(in_features=4096, out_features=256, bias=True)
      (1): ReLU()
      (2): Dropout(p=0.4, inplace=False)
      (3): Linear(in_features=256, out_features=1, bias=True)
      (4): Sigmoid()
    )
  )
)
```

6.3 GoogLeNet

GoogLeNet [6], won the ImageNet Challenge in 2014, is seen as the first network that exhibited a clear distinction among the stem (data ingest), body (data processing), and head (prediction) in a CNN. This design pattern has persisted ever since in the design of deep networks:

- The stem is given by the first two or three convolutions that operate on the image. They extract low-level features from the underlying images.
- The body comprises convolutional blocks.
- Finally, the head maps the features obtained so far to the required classification, segmentation, detection, or tracking problem.

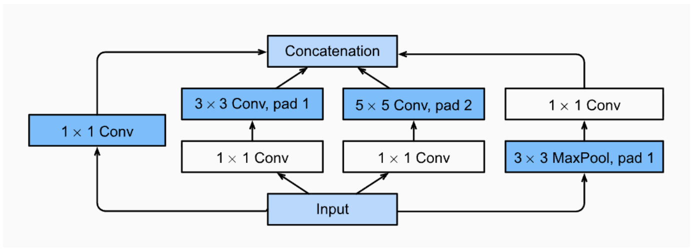
GoogLeNet is a multi-branch network, as shown in Figure 12. The key contribution in GoogLeNet was the design of the network body. It solved the problem of selecting convolution kernels in an ingenious way. The basic convolutional block in GoogLeNet is called an Inception block, illustrated in Figure 12a, stemming from the meme "we need to go deeper" from the movie Inception. GoogLeNet explores the image in a variety of filter sizes so that details at different extents can be recognized efficiently. As Figure 12b shows, GoogLeNet uses a stack of a total of 9 inception blocks, arranged into three groups with max-pooling in between, and global average pooling in its head to generate its estimates. Max-pooling between inception blocks reduces the dimensionality.

6.4 ResNet

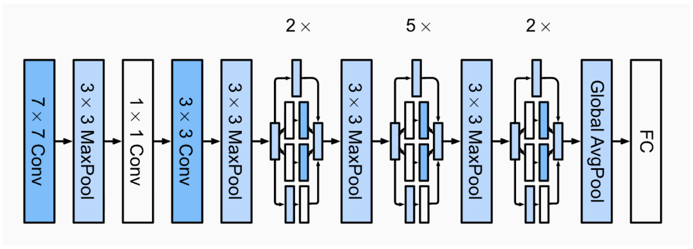
At the heart of residual network (ResNet) [7] is the idea that every additional layer should more easily contain the identity function, $f(\mathbf{x}) = \mathbf{x}$, as one of its elements. This means that if adding more layers does not improve the representation, the network can at least maintain the performance of the previous layers instead of degrading it. This consideration is rather profound but they led to a surprisingly simple solution, a *residual block*. ResNet won the ImageNet Large Scale Visual Recognition Challenge in 2015.

Figure 13a illustrates a residual block. Let the input be $\mathbf{x}$. Suppose the desired mapping to learn is $f(\mathbf{x})$, which serves as the input to the activation function at the top. In a standard block (left), the network must directly learn $f(\mathbf{x})$. In contrast, a residual block (right) learns the *residual mapping*:

$$g(\mathbf{x}) = f(\mathbf{x}) - \mathbf{x}, \tag{9}$$

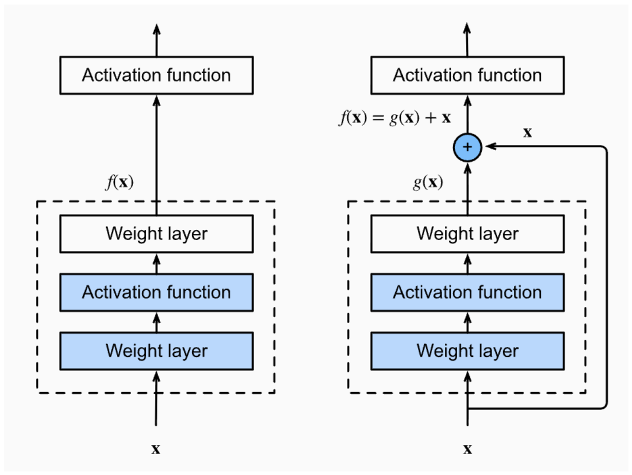


(a) Inception block in GoogLeNet

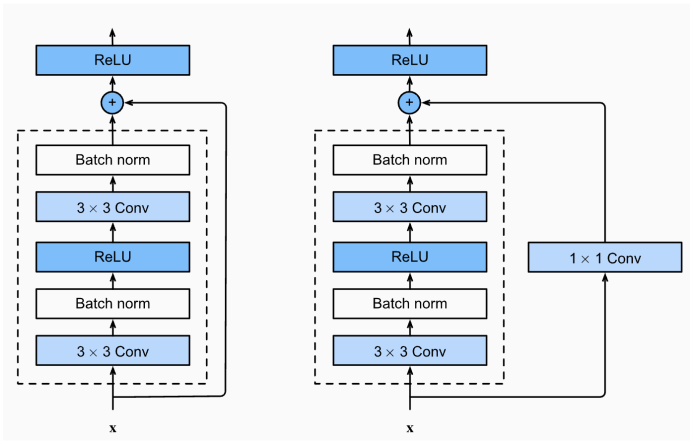


(b) GoogLet Architecture (source: [2])

Figure 12: GoogLeNet



(a) Left: a regular block; right: a residual block in ResNet



(b) ResNet architecture: without (left) and with (right) 1×1 convolution for the skip connection

Figure 13: ResNet (source: [2])

hence the name residual block. If the ideal mapping is the identity $f(\mathbf{x}) = \mathbf{x}$, then $g(\mathbf{x}) = 0$, making it much easier to learn since the weights and biases can simply approach zero.

This design provides two major benefits. First, it simplifies optimization by letting the network focus on learning small adjustments $g(\mathbf{x})$ rather than the entire mapping $f(\mathbf{x})$ from scratch. Second, the solid line that bypasses the main layers and carries $\mathbf{x}$ directly to the addition operator, which is called a *residual connection*, creates a shortcut path for gradients during backpropagation. This improves gradient flow, mitigating the vanishing gradient problem and enabling much deeper networks to train effectively. As a result, residual blocks help inputs and gradients propagate more easily across layers, leading to faster and more stable training.

Figure 13b shows that ResNet's residual block has two convolutional layers with the same number of output channels. Each convolutional layer is followed by a batch normalization layer and a ReLU activation function. Then, we skip these two convolution operations and add the input directly before the final ReLU activation function. This kind of design requires that the output of the two convolutional layers has to be of the same shape as the input, so that they can be added together. If we want to change the number of channels, we need to introduce an additional convolutional layer to transform the input into the desired shape for the addition operation.

Batch normalization in ResNet is a popular and effective technique that consistently accelerates the convergence of deep networks. It conveys three key benefits:

- Preprocessing: Normalizes activations in intermediate layers, similar to standardizing input features, putting parameters on comparable scales for faster and more stable optimization.
- Numerical stability: Reduces the effect of widely varying activation magnitudes, mitigating internal covariate shift and enabling higher learning rates.
- Regularization: Acts as implicit noise injection, helping deeper networks resist overfitting, analogous to dropout or input noise.

Denote by $\mathcal{B}$ a minibatch and let $\mathbf{x} \in \mathcal{B}$ be an input to batch normalization (BN). In this case the batch normalization is defined as follows:

$$\mathrm{BN}(\mathbf{x}) = \boldsymbol{\gamma} \odot \frac{\mathbf{x} - \hat{\boldsymbol{\mu}}_{\mathcal{B}}}{\hat{\boldsymbol{\sigma}}_{\mathcal{B}}} + \boldsymbol{\beta}, \tag{10}$$

where $\hat{\boldsymbol{\mu}}_{\mathcal{B}}$ is the sample mean and $\hat{\boldsymbol{\sigma}}_{\mathcal{B}}$ is the sample standard deviation of the minibatch $\mathcal{B}$. After applying standardization, the resulting minibatch has zero mean and unit variance.

6.5 DenseNet

Recall the Taylor expansion for functions. At the point $x = 0$ it can be written as

$$f(x) = f(0) + x \cdot \left[f'(0) + x \cdot \left[\frac{f''(0)}{2!} + x \cdot \left[\frac{f'''(0)}{3!} + \cdots \right] \right] \right]. \tag{11}$$

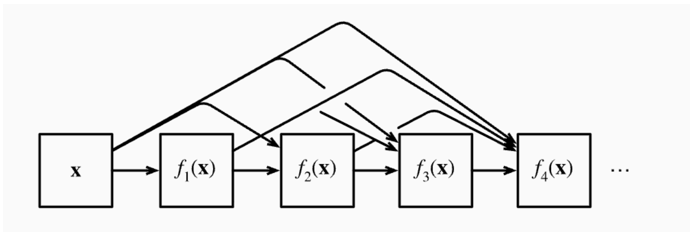


Figure 14: DenseNet (source: [2])

ResNet decomposes functions into a simple linear term and a more complex nonlinear one. DenseNet [8] is designed to capture information beyond that. The key difference between ResNet and DenseNet is that in the latter case outputs are concatenated (denoted by $[,]$) rather than added. As a result, we perform a mapping from $\mathbf{x}$ to its values after applying an increasingly complex sequence of functions:

$$\mathbf{x} \to [\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})]), f_3([\mathbf{x}, f_1(\mathbf{x}), f_2([\mathbf{x}, f_1(\mathbf{x})])]), \ldots]. \tag{12}$$

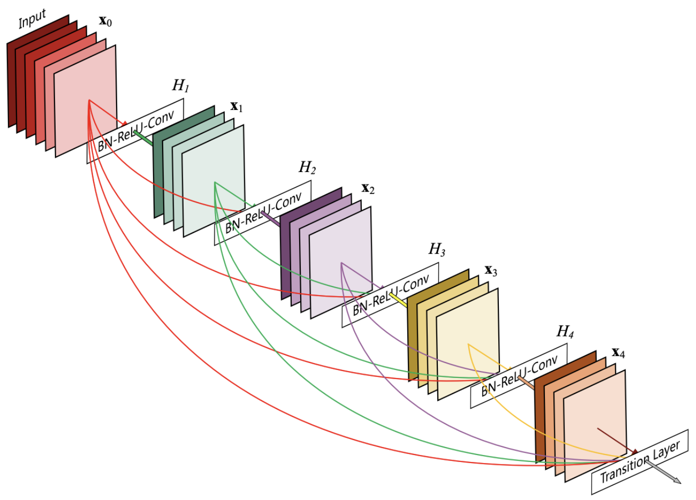


Figure 15: A 5-layer dense block with a growth rate of $k = 4$. Each layer takes all preceding feature-maps as input. (source: Figure 1 in [8])

The main components that comprise a DenseNet are *dense blocks* and *transition layers*. The former define how the inputs and outputs are concatenated, while the latter control the number of channels since the expansion in Equation (12) can be quite high-dimensional. Figure 15 illustrates a 5-layer dense block with a growth rate of $k = 4$. Each layers performs BN-ReLU-Conv(1,1)-BN-ReLU-Conv(3,3) and produce 4 feature maps. With the dense connection, the number of output channels of this block is $4 \times (5 - 1) + k_0$. A transition layer reduces the number of feature-maps produced by each dense block. Each transition layer is a conv (1,1) followed by an average-pooling (2,2).

7 From CNN Architectures to Representation Learning

So far, we have reviewed several influential CNN architectures: AlexNet, VGG, GoogLeNet, ResNet, and DenseNet. Each of these CNN architectures demonstrates increasing depth, sophisticated connectivity patterns, and techniques to ease optimization (e.g., residual and dense connections). While these architectures differ in their design principles, a unifying theme is that they all aim to extract *useful features* or *representations* from raw input data.

Representation learning refers to the process by which a network automatically discovers the features required for a task directly from data, instead of relying on hand-engineered features. In CNNs, early layers typically learn low-level patterns (edges, textures), while deeper layers capture higher-level semantic features (parts of objects, global structures). By encoding data in increasingly abstract representations, deep networks can achieve superior performance on complex tasks such as image classification, object detection, and segmentation.

8 Transfer Learning

Training a model from scratch requires requires a large amount of labeled data and computational resources. The learned representations are often general enough to be useful for related tasks, which leads naturally to the concept of *transfer learning*.

Transfer learning leverages representations learned on a large source dataset to improve performance on a smaller, potentially related target dataset. We can reuse the convolutional feature extractor (i.e., the learned representations) and fine-tune it for a new task. We can freeze the entire convolutional base (called backbone) and only train head for the downstream task to reuse already learned representations. We can also fine-tune part or all of the pretrained network on the target dataset.

Transfer learning is particularly effective when the target dataset is small, as it allows the model to exploit prior knowledge captured in the pretrained weights, accelerating convergence and improving generalization. For example, using a ResNet, DenseNet, or VGG pretrained on ImageNet, we can adapt the network to classify inspection images or detect defects by replacing the final fully connected layer and optionally fine-tuning deeper layers.

9 Foundation Models

The success of transfer learning paved the way for a new paradigm in machine learning: the development of *foundation models* [9]. Figure 16

A foundation model is a large-scale model trained on massive, diverse datasets using self-supervised or weakly supervised learning objectives. These models, such as (Contrastive Language–Image Pretraining) CLIP, Segment Anything Model (SAM), and GPT-family models, can be adapted to a wide range of downstream tasks with minimal task-specific fine-tuning.

Unlike traditional transfer learning, where pretraining occurs on a specific domain (e.g., ImageNet for vision tasks), foundation models are trained to learn broadly generalizable representations across domains and modalities. For example, CLIP jointly learns visual and textual representations, enabling zero-shot image classification, retrieval, and multimodal reasoning without additional labeled data.

Foundation models usually have all or part of the following important characteristics:

- Scale: They are trained with billions of parameters and massive datasets spanning multiple domains.
- Generalization: Their representations transfer effectively across a wide variety of downstream tasks, often with minimal or no fine-tuning.
- Multimodality: Many are designed to process multiple types of input (e.g., text, image, or audio) simultaneously.
- Adaptability: They enable new workflows such as prompting, few-shot learning, and zero-shot inference.

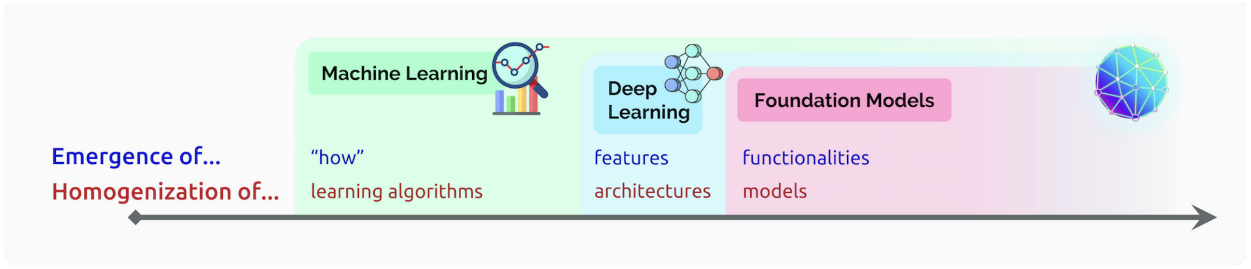


Figure 16: The story of AI has been one of increasing emergence and homogenization. With the introduction of machine learning, how a task is performed emerges (is inferred automatically) from examples; with deep learning, the high-level features used for prediction emerge; and with foundation models, even advanced functionalities such as in-context learning emerge. At the same time, machine learning homogenizes learning algorithms (e.g., logistic regression), deep learning homogenizes model architectures (e.g., Convolutional Neural Networks), and foundation models homogenizes the model itself (e.g., GPT-3). (source: Figure 1 in [9])

References

[1] Alexander Amini and Ava Amini. Mit introduction to deep learning. `https://introtodeeplearning.com/`, 2025. © Alexander Amini and Ava Amini, Massachusetts Institute of Technology.

[2] Aston Zhang, Zachary C Lipton, Mu Li, and Alexander J Smola. *Dive into deep learning*. Cambridge University Press, 2023.

[3] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[4] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. Imagenet: A large-scale hierarchical image database. In *2009 IEEE conference on computer vision and pattern recognition*, pages 248–255. Ieee, 2009.

[5] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[6] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[8] Gao Huang, Zhuang Liu, Laurens Van Der Maaten, and Kilian Q Weinberger. Densely connected convolutional networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4700–4708, 2017.

[9] Rishi Bommasani, Drew A. Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S. Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, Erik Brynjolfsson, Shyamal Buch, Dallas Card, Rodrigo Castellon, Niladri Chatterji, Annie Chen, Kathleen Creel, Jared Quincy Davis, Dorottya Demszky, Chris Donahue, Moussa Doumbouya, Esin Durmus, Stefano Ermon, John Etchemendy, Kawin Ethayarajh, Li Fei-Fei, Chelsea Finn, Trevor Gale, Lauren Gillespie, Karan Goel, Noah Goodman, Shelby Grossman, Neel Guha, Tatsunori Hashimoto, Peter Henderson, John Hewitt, Daniel E. Ho, Jenny Hong, Kyle Hsu, Jing Huang, Thomas Icard, Saahil Jain, Dan Jurafsky, Pratyusha Kalluri, Siddharth Karamcheti, Geoff Keeling, Fereshte Khani, Omar Khattab, Pang Wei Koh, Mark Krass, Ranjay Krishna, Rohith Kuditipudi, Ananya Kumar, Faisal Ladhak, Mina Lee, Tony Lee, Jure Leskovec, Isabelle Levent, Xiang Lisa Li, Xuechen Li, Tengyu Ma, Ali Malik, Christopher D. Manning, Suvir Mirchandani, Eric Mitchell, Zanele Munyikwa, Suraj Nair, Avanika Narayan, Deepak Narayanan, Ben Newman, Allen Nie, Juan Carlos Niebles, Hamed Nilforoshan, Julian Nyarko, Giray Ogut, Laurel Orr, Isabel Papadimitriou, Joon Sung Park, Chris Piech, Eva Portelance, Christopher Potts, Aditi Raghunathan, Rob Reich, Hongyu Ren, Frieda Rong, Yusuf Roohani, Camilo Ruiz, Jack Ryan, Christopher Ré, Dorsa Sadigh, Shiori Sagawa, Keshav Santhanam, Andy Shih, Krishnan Srinivasan, Alex Tamkin, Rohan Taori, Armin W. Thomas, Florian Tramèr, Rose E. Wang, William Wang, Bohan Wu, Jiajun Wu, Yuhuai Wu, Sang Michael Xie, Michihiro Yasunaga, Jiaxuan You, Matei Zhang, Michael Zhang, Tianyi Zhang, Xikun Zhang, Yuhui Zhang, Lucia Zheng, Kaitlyn Zhou, and Percy Liang. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.