ECE 046211 - Technion - Deep Learning

Tal Daniel

Tutorial 06 - Convolutional Neural Networks & Visual Tasks

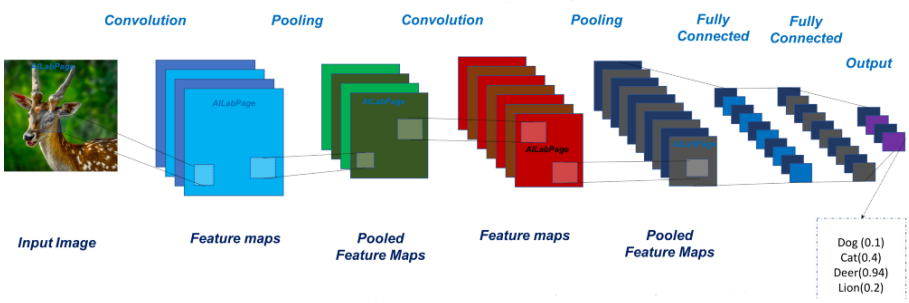


Image Source

Agenda

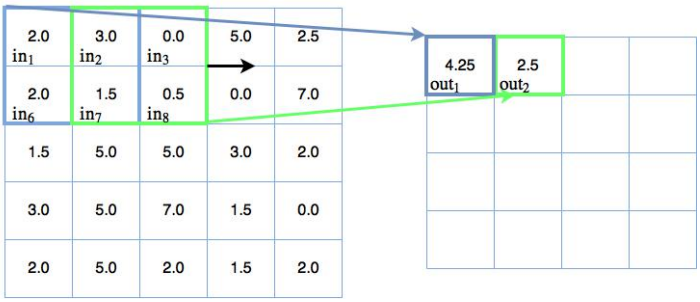
In [1]:
```
# imports for the tutorial
import numpy as np
import matplotlib.pyplot as plt
import time
import os

# pytorch
import torch
import torch.nn as nn
import torchvision
import torchvision.transforms as transforms
```

2D Convolution

Mathematically, 2D convolution is defined as:

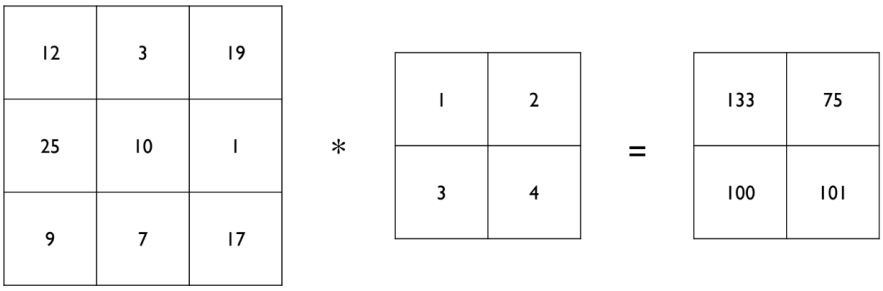
$$f[n,m] * h[n,m] = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f[k,l] \cdot h[n-k, m-l]$$

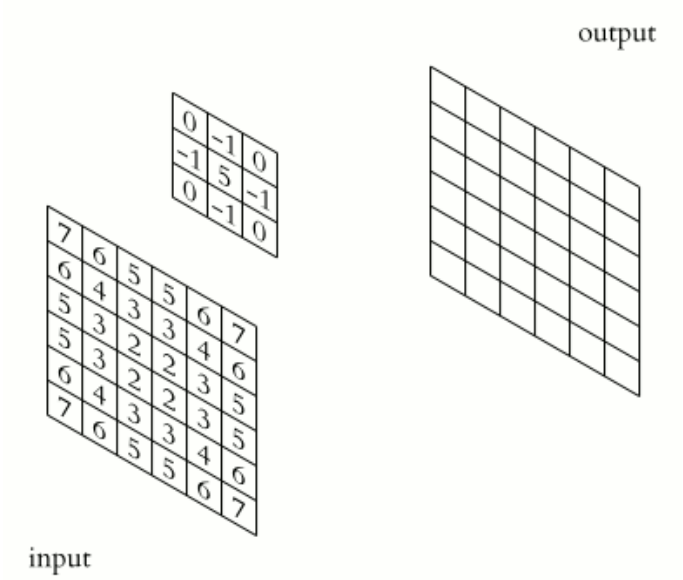
Convolution is moving a window or filter across the image being studied. This moving window applies to a certain neighborhood of nodes as shown below – here, the filter applied is (0.5 × the node value):

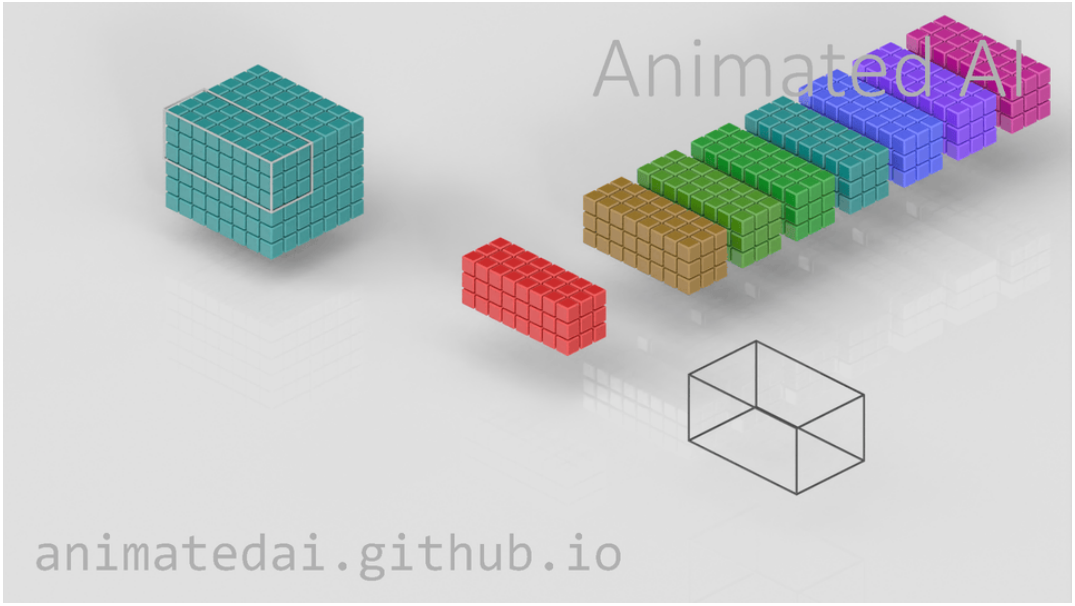


- In our course, we will treat 2D convolution as *cross-correlation*.

Numerical Example

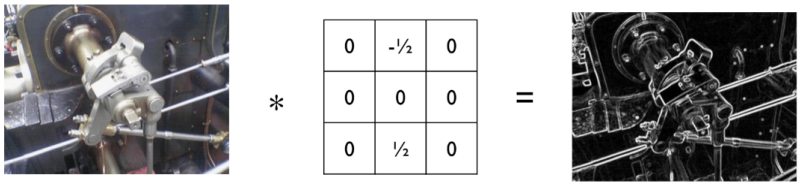




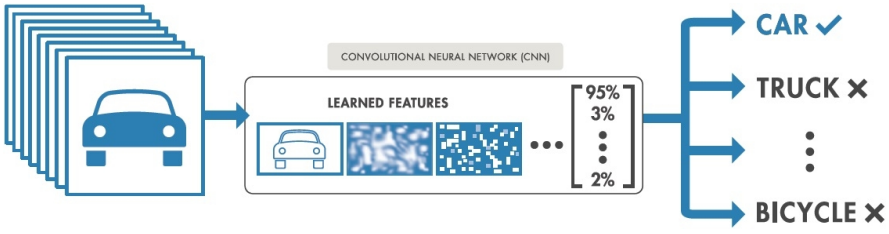


Convolution as Feature Extractors for Classification

- Convolution is useful since it helps us find intersting insights/features from images.
- For example, the gradient/derivative filter helps us detect **edges**.



- Recall that in classification tasks we need good features for better classification performance.
- In *image classification*, we usually want to classify images into categories.
- Imagine that we have a filter for each class, and that by applying this filter, we get a **probability** for the input image to be from this filter's class.



  - Image Source

- What are features? Consider the following illustrative example - classifying *cats* and *dogs*.
  - How do we tell the difference between cats and dogs? One can look at the length of the tail, shape of the paws, pattern of the fur and etc...
  - Humans can usually tell these just by looking at the sample. But what do computers see?
- In classification tasks, we need *good features* to learn a function that maps from samples to labels.
- **Raw pixels** are usually not expressive enough features! That is because raw pixels do not capture the *spatial relationship* in the image.
- Extracting features from raw pixels can be done using a deep learning network (which is a complex, non-linear function of the input).
  - Using just linear layers (multi-layer Perceptron) might work for simple images (e.g. MNIST), but they have a lot of parameters!
  - Using convolution, we can capture **spatial structures** (e.g., pixels the shape a tail).

Training Multi-class Classification Models

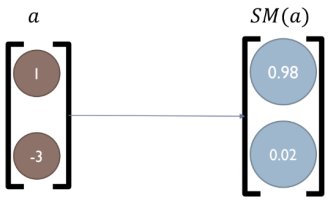
- Assume we designed an architecture for classification, but how do we train it to output the correct class of the input (for example, an image)? Just as we did previously!
- The output of the final fully-connected layer in a classification model is a vector of length num-classes, which is exactly the number of classes we have (for example, in MNIST or CIFAR-10, we have 10 classes, thus the output dimension is 10)!
- In the output vector, we want entry i to be the probability of the input to be from class i.
- But how do we force this vector to output probability and not just some numbers?
- We consider the final output vector to represent *scores*, which we will normalize to be probabilities using the **Softmax** function.

The Softmax Function

- The Softmax function is defined as:

$$Softmax(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{M} e^{x_j}}, i \in [1, \dots, M], x \in \mathcal{R}^M$$

- This forces the output vector to sum to 1, just like probabilities.



Making Predictions

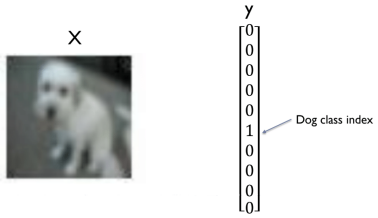
- OK great, we have an output vector of probabilities, so how do we predict the label of the input image?
- Simple! Just take the $argmax$:

$$\hat{y} = Softmax(NN(x))$$

$$c_{pred} = argmax_i(\hat{y})$$

Loss Function - Cross Entropy

- In order to train the model in an end-to-end fashion, we need to define a loss function which we can minimize using optimization techniques.
- Let us assume that our model output (after softmax) is $\hat{y}$ and the real label (the real class, given to us) is y.



- **Insight**:

$$argmax(\hat{y}) = argmax(y)$$

- Ideally, this is what we would want from our model, so what loss function would drive $\hat{y}$ to be as close as possible to y?

- As y is a *one-hot vector* and $\hat{y}$ represents probabilities, the **Cross Entropy** loss function fits right in!

- Let W denote the weights of the CNN, and W^* the optimal weights.
- In this case, the optimal weights are:

$$W^* \leftarrow argmin_W \left(-\sum_{x,y} 1 \cdot \log(p_c) \right)$$

$$p_c = \hat{y}_c$$

- $c \in [1, \ldots, M]$ is the correct class, $\hat{y}_c$ is the c^{th} entry in the output vector $\hat{y}$.
- So the **Cross Entropy** loss function is:

$$\mathcal{L} = -\log(p_c)$$

Let's analyze this loss fucntion, which represents how bad we are currently doing:

$$p_c = 0 \rightarrow \mathcal{L} = -\log(0) = \infty$$

$$p_c = 0.1 \rightarrow \mathcal{L} = -\log(0.1) = 2.3$$

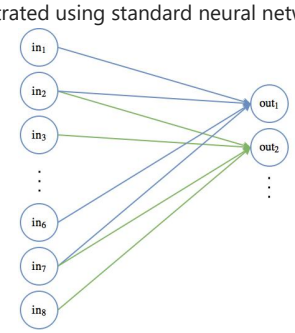
$$p_c = 0.9 \rightarrow \mathcal{L} = -\log(0.9) = 0.1$$

$$p_c = 1 \rightarrow \mathcal{L} = -\log(1) = 0$$

The larger the loss, the worse the prediction. We want to minimize it!

Convolutional Neural Networks (CNNs)

- Convolutional Neural Networks (CNNs) are deep neural networks that contain layers of stacked convolution layers or filters.
- They are mainly used for image datasets, but are also useful for other areas (natural lanugage processing for example).
- In the convolutional part of the CNN, we can imagine a moving filter sliding across all the available nodes / pixels in the input image. This operation can also be illustrated using standard neural network node diagrams:



- The first position of the moving filter connections is illustrated by the blue connections, and the second is shown with the green lines. The weights of each of these connections are usually learned.

The Basic Principles of CNNs

We will now cover the basic building blocks of Convolutional Neural Networks (CNNs) and the ideas the enable efficient learning on image data.

Feature Mapping and Multiple Channels

- Since the weights of individual filters are held constant as they are applied over the input nodes, they can be trained to select certain features from the input data.
- In the case of images, it may learn to recognize common geometrical objects such as lines, edges and other shapes which make up objects.
- This is where the name *feature mapping* comes from. Because of this, **any convolution layer needs multiple filters which are trained to detect different features**.

- To learn different filters to detect various features, we need them to be **initialized differently**, as we learned in the previous tutorial -- intialization matters!

Pooling

- It is a sliding window type technique, but instead of applying weights, which can be trained, it applies a statistical function of some type over the contents of its window.
- The most common type of pooling is called **max pooling**, and it applies the $max()$ function over the contents of the window.

There are two main benefits to pooling in CNN's:

1. It reduces the number of parameters in your model by a process called *down-sampling* (if the following layer is an FC layer).
2. It makes feature detection more robust to object orientation and scale changes.

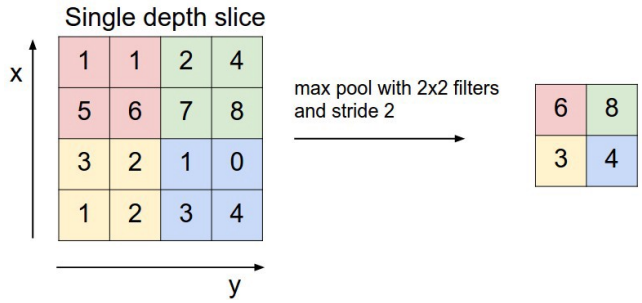
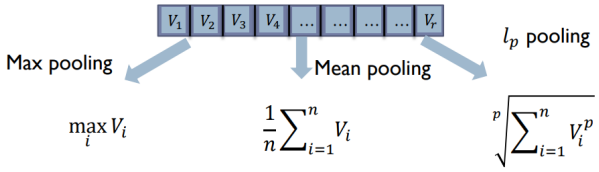


Image Source

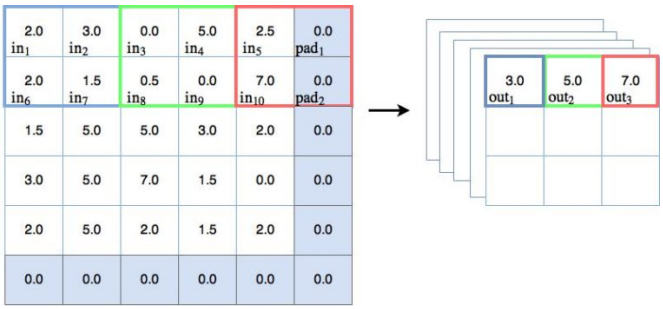
- Other pooling operators:



- Pooling generalizes over lower level, more complex information.
- Let's imagine the case where we have convolutional filters that, during training, learn to detect the digit "9" in various orientations within the input images.
- In order for the Convolutional Neural Network to learn to classify the appearance of "9" in the image correctly, it needs to "activate" whenever a "9" is found anywhere in the image, no matter what the size or orientation the digit is (except for when it looks like "6", that is).
- Pooling can assist with this higher level, generalized feature selection.

Strides and Down-sampling

- In the pooling diagram below, you will notice that the pooling window shifts to the right each time by 2 places.
- This is called a **stride of 2**, which should be considered both in the x and y direction.
  - In other words, the stride is actually specified as $[2, 2]$.
- One important thing to notice is that, if during pooling the stride is greater than 1, then the output size will be reduced.
- As can be observed below, the 5 x 5 input is reduced to a 3 x 3 output.
- This is a good thing – it is called down-sampling, and it can potentially reduces the number of trainable parameters in the model in case the number of parameters in following layers is dependent in the input dimension (e.g., a linear layer).



- Images from adventuresinmachinelearning.com

Padding

- In the pooling diagram above there is an extra column and row added to the 5 x 5 input – this makes the effective size of the pooling space equal to 6 x 6.
- This is to ensure that the 2 x 2 pooling window can operate correctly with a stride of [2, 2] -- *padding*.
- These nodes are basically dummy nodes – because the values of these dummy nodes is 0, they are basically invisible to the max pooling operation.
- Padding will need to be considered when constructing our Convolutional Neural Network in PyTorch.
- Padding is also used to **retain** or **enlarge** the dimensions of the input. For example, a "same" type padding means the output 2D tensor will have the same 2D dimensions as the input tensor.
- While we usually pad with zeros, it is possible to pad with other values (e.g., reflection or cyclic padding).

The FC Layer

- The fully connected layer can be thought of as attaching a standard classifier onto the information-rich output of the network, to "interpret" the results and finally produce a classification result.
  - That is, the output of the convolutional layers is the new "input features" for the classifier.
- In order to attach this fully connected layer to the network, the dimensions of the output of the Convolutional Neural Network needs to be *flattened*.
- **Note**: some networks (e.g., Fully-Convolutional Network, FCN) do not use FC layers at all!
  - For example, you can design the convolutional kernels such that the final output is a tensor of shape `[num_classes, 1, 1]` (`num_channels` channels, height of 1 and width of 1) and use that for classification. It also useful for networks that perform image segmenation or generation, where the output is pixels.

Calculating the Convolutional Layer Output Shape

We define the following parameters of a *convolutional layer*:

- W_{in} - the width of the input
- F - filter size
- P - padding
- S -stride

The output width:

$$W_{out} = \frac{W_{in} - F + 2P}{S} + 1$$

Consider the input images of size 28×28, filter size of 5×5, padding of 2 and a stride of 1, the output of the convolutional layers, just before the FC:

$$W_{1,out} = \frac{28 - 5 + 2 * 2}{1} + 1 = 28 \rightarrow MaxPooling(2x2) \rightarrow 28/2 = 14$$

$$W_{2,out} = \frac{14 - 5 + 2 * 2}{1} + 1 = 14 \rightarrow MaxPooling(2x2) \rightarrow 14/2 = 7$$

So the input to the FC layer is $7x7 = 49$ (because we have 7 for the width and 7 for the height).

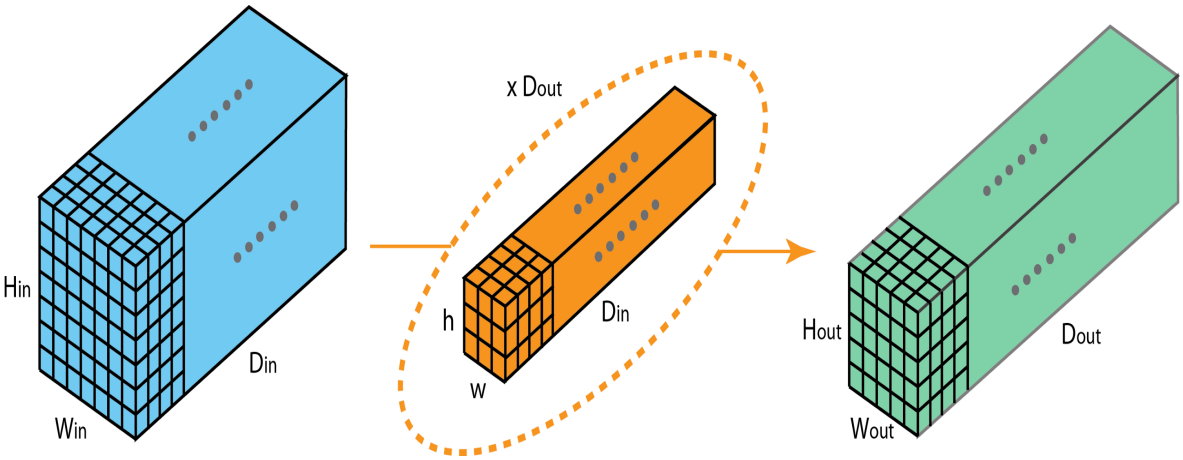
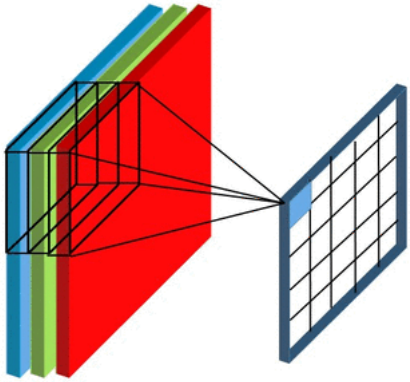


Image Source



Animation by Nadeem Qazi.

Low Level (Shallow) and High Level (Deep) Features

- It is quite common to observe the features (outputs of the convolutional filters) at different levels of the network.
- **Low Level** - also called shallow features (first layers), which include lines, corners and edges.
- **Mid Level** - the middle level features, usually object parts.
- **High Level** - also called deep features (final layers), which include whole objects (global).

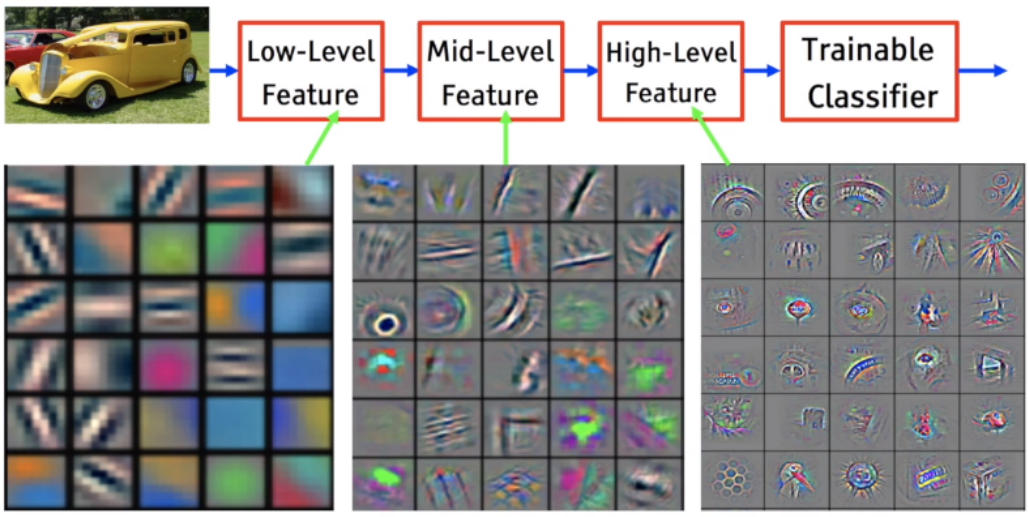
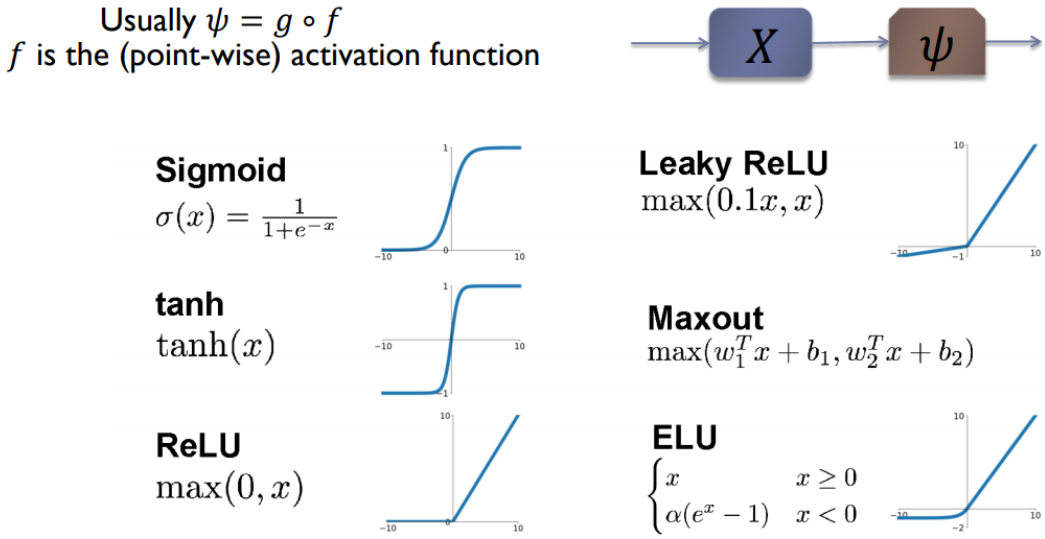


Image Source

Non-Linear Activations

- The key change made to the Perceptron that brought upon the era of deep learning is the addition of **activation function** to the output of each neuron.
- These allow the learning of non-linear functions. A neural network without an activation function is essentially just a linear regression model.



CNN vs. Fully Connected

- Fully connected networks with a few layers can only do so much – to get close to state-of-the-art results in image classification it is necessary to go deeper.
  - In other words, lots more layers are required in the network.
- However, by adding a lot of additional layers, we come across some problems.
  - First, we can run into the vanishing gradient problem. However, this can be solved to an extent by using sensible activation functions, such as the ReLU family of activations or residual networks (skip-connections).
  - Another issue for deep fully connected networks is that the number of trainable parameters in the model (i.e. the weights) can grow rapidly.
    - This means that the training slows down or becomes practically impossible, and also exposes the model to overfitting. CNNs try to solve this second problem by **exploiting correlations between adjacent inputs in images** (*inductive bias* - the locality of pixels).
- Trainable parameters: as Convolutional kernels/filters are usually of smaller sizes such as 7x7, 5x5 and 3x3, it keeps the number of trainable parameters rather low comparing to FC networks.

Regularization - Preventing Overfitting

- A common phenomenon in machine learning is that even though the training error keeps decreasing (training loss keeps going down, training accrucay goes up), the validation/test error goes down but then at some point it starts going up! (which is bad...)
  - In deep learning, we often see **Deep Double Descent**, which deviates from the classical ML regime.
- This is called **overfitting**. Although it's often possible to achieve high accuracy on the training set, what we really want is to develop models that generalize well to a testing set (or data they haven't seen before).
- If you train for too long though, the model will start to overfit and learn patterns from the training data that don't generalize to the test data.
  - We need to find a balance!
- To prevent overfitting, the best solution is to **use more complete training data**. The dataset should cover the full range of inputs that the model is expected to handle. Additional data may only be useful if it covers new and interesting cases.
- A model trained on more complete data will naturally **generalize better**. When that is no longer possible, the next best solution is to use techniques like **regularization**.
- **Regularization** places constraints on the quantity and type of information your model can store. If a network can only afford to memorize a small number of patterns, the optimization process will force it to focus on the most prominent patterns, which have a better chance of generalizing well.

- The opposite of overfitting is **underfitting**. Underfitting occurs when there is still room for improvement on the test data.
  - This can happen for a number of reasons: If the model is not powerful enough, is over-regularized, or has simply not been trained long enough. This means the network has not learned the relevant patterns in the training data.
- Connection to **deep double descent** - adding regularization can alleviate this phenomenon.

- Regularization usually comes in the form of placing constraints on the parameters, or in the case of neural networks, constraints on the weights of the layers.
- It introduces a cost term for bringing in more features with the objective function. Hence, it tries to drive the coefficients of many variables to zero leading to a reduced cost term.
  - Common regularizations are L_2, L_1 regularizations:

$$\text{New Loss}_{L_2} = \text{Original Loss} + \lambda \mid\mid w \mid\mid^2$$

- For deep neural networks (and CNNs) a common regularization technique is **Dropout**.

Dropout Regularization

- First presented in Dropout: A Simple Way to Prevent Neural Networks from Overfitting, 2014.
- Dropout is a regularization method that **approximates training a large number of neural networks with different architectures in parallel**.
- During training, some number of layer outputs (i.e. neurons) are randomly ignored or "dropped out" with some probability p. This has the effect of making the layer look-like and be treated-like a layer with a different number of nodes and connectivity to the prior layer.
- Dropout has the effect of making the training process noisy, forcing nodes within a layer to probabilistically take on more or less responsibility for the inputs.
- Dropout is activated **only during training** (`model.train()`). At test time, it is turned off (`model.eval()`).

Read more - A Gentle Introduction to Dropout for Regularizing Deep Neural Networks

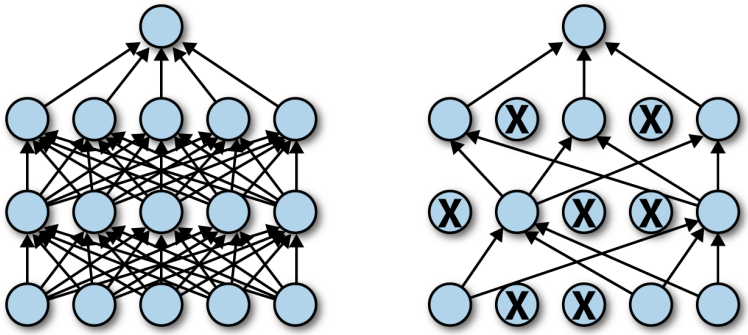


Image Source

Data Augmentation

- Data augmentation is a common technique to improve results and avoid overfitting, helping the network generalize better.
- **How do we get more data when there is limited number of samples?** We can perform data augmentation.
- Data augmenetation enriches the dataset by adding variations of the original samples.
  - And as you know, deep learning flourishes when there is A LOT of data.

- Popular augmentation techniques:
  - **Flip** - Flip images horizontally and/or vertically.
  - **Rotation** - Rotate the images at certain degrees. This may change the size of the image, thus, cropping or padding is a common fix for that.
  - **Scaling** - The image can be scaled outward or inward. This may also change the size of the image, thus, resizing (also stretching) is often followed.
  - **Cropping** - Randomly sample a section from the original image. Then resize this section to the original image size. This is called Random Crop.

- **Translation** - Move the image along the X or Y direction (or both). This forces the neural network to look everywhere.
- **Noise** - Overfitting usually happens when the network tries to learn high frequency features (patterns that occur a lot) that may not be useful. Gaussian noise, which has zero mean, essentially has data points in all frequencies, effectively distorting the high frequency features. This also means that lower frequency components (usually, your intended data) are also distorted, but your neural network can learn to look past that. Adding just the right amount of noise can enhance the learning capability (e.g., add Salt and Pepper).

Read More - Data Augmentation | How to use Deep Learning when you have Limited Data

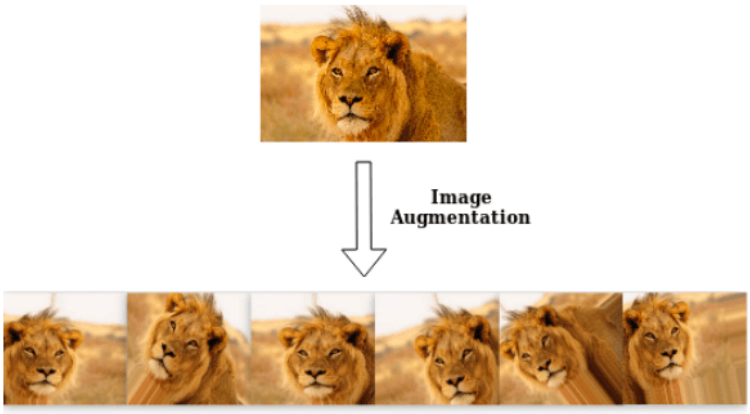


Image Source

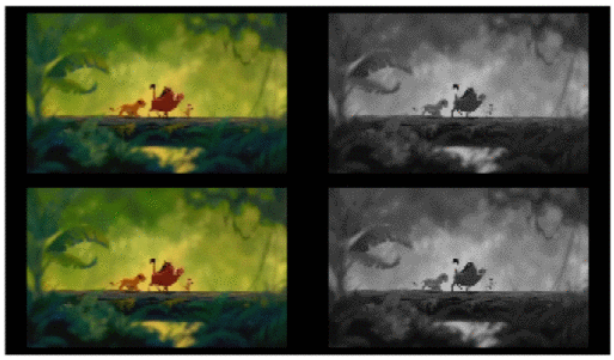
Kornia: Differentiable GPU-Accelerated Augmentations

- Kornia is a differentiable library that allows classical computer vision to be integrated into deep learning models.
- It allows to perform data augmentations and to apply other image and geometry operations/transformations directly on the GPU and making it possible to backpropagate through these operations!
  - This is not possible with the augmentations implemented in `torchvision` and usually requires to send the image tensor back to the CPU.
- See Jupyter Notebook Tutorials using Kornia.

kornia

PyTorch

```
import torch
import kornia

frame: torch.Tensor = load_video_frame(...)

out: torch.Tensor = (
    kornia.rgb_to_grayscale(frame)
)
```



Here is a benchmark performed on Google Colab K80 GPU with different libraries and batch sizes. This benchmark shows strong GPU augmentation speed acceleration brought by Kornia data augmentations. The image size is fixed to 224x224 and the unit is milliseconds (ms).

Libraries	**TorchVision**	**Albumentations**	**Kornia (GPU)**		
Batch Size	**1**	**1**	**1**	**32**	**128**
RandomPerspective	4.88±1.82	4.68±3.60	4.74±2.84	0.37±2.67	0.20±27.00
ColorJiggle	4.40±2.88	3.58±3.66	4.14±3.85	0.90±24.68	0.83±12.96

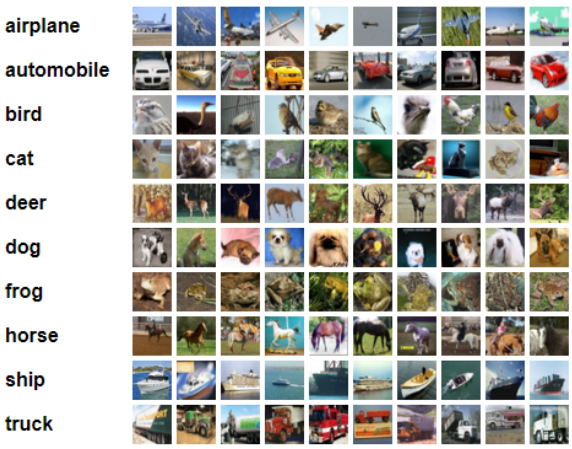
Libraries	TorchVision	Albumentations	Kornia (GPU)		
Batch Size	**1**	**1**	**1**	**32**	**128**
RandomAffine	3.12±5.80	2.43±7.11	3.01±7.80	0.30±4.39	0.18±6.30
RandomVerticalFlip	0.32±0.08	0.34±0.16	0.35±0.82	0.02±0.13	0.01±0.35
RandomHorizontalFlip	0.32±0.08	0.34±0.18	0.31±0.59	0.01±0.26	0.01±0.37
RandomRotate	1.82±4.70	1.59±4.33	1.58±4.44	0.25±2.09	0.17±5.69
RandomCrop	4.09±3.41	4.03±4.94	3.84±3.07	0.16±1.17	0.08±9.42
RandomErasing	2.31±1.47	1.89±1.08	2.32±3.31	0.44±2.82	0.57±9.74
RandomGrayscale	0.41±0.18	0.43±0.60	0.45±1.20	0.03±0.11	0.03±7.10
RandomResizedCrop	4.23±2.86	3.80±3.61	4.07±2.67	0.23±5.27	0.13±8.04
RandomCenterCrop	2.93±1.29	2.81±1.38	2.88±2.34	0.13±2.20	0.07±9.41

Torchvision Datasets

- Torchvision provides many built-in datasets in the `torchvision.datasets` module, as well as utility classes for building your own datasets.
- Torchvision Datasets
  - Examples: `torchvision.datasets.CelebA`, `torchvision.datasets.Flowers102`, `torchvision.datasets.ImageNet`.
- If you want to create a dataset with your own images you can use `torchvision.datasets.ImageFolder` and `torchvision.datasets.VisionDataset`, or you can implement the `Dataset` class yourself.

The CIFAR-10 Dataset

- The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.
  - There is also CIFAR-100, with 100 classes.
- The classes are completely mutually exclusive. There is no overlap between automobiles and trucks. "Automobile" includes sedans, SUVs, things of that sort. "Truck" includes only big trucks. Neither includes pickup trucks.
- Official Site



In [2]:
```
# define pre-processing steps on the images
# also called "data augementation" (only done for the train set)
# to use 'kornia' instead of torchvision, see example after

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4), # input is PIL image
    transforms.RandomHorizontalFlip(), # input is PIL image, can also set the probability parameter 'p'
    transforms.ToTensor(),  # uint8 values in [0, 255] -> float tensor with values [0, 1]
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),  # [mu, sigma] standartization RGB
    # notice the order: first iamges transformation on PIL, then ToTensor, then normalization.
])

# how are the standartization params (mu, sigma) calculated?
# iterate over the entire train set to get the R, G, and B values of the all images, compute their mean and std

# Normalize the test set same as training set without augmentation
transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

# test-time augementations (TTA): for a more robust prediciton, we can apply N augs on each input and get N score

# load dataset
classes = ('plane', 'car', 'bird', 'cat', 'deer',
           'dog', 'frog', 'horse', 'ship', 'truck')

trainset = torchvision.datasets.CIFAR10(
    root='./datasets', train=True, download=True, transform=transform_train)

testset = torchvision.datasets.CIFAR10(
    root='./datasets', train=False, download=True, transform=transform_test)
```

```
Files already downloaded and verified
Files already downloaded and verified
```

In [8]:
```
# let's see some of the images
def convert_to_imshow_format(image):
    # first convert back to [0,1] range
    mean = torch.tensor([0.4914, 0.4822, 0.4465])
    std = torch.tensor([0.2023, 0.1994, 0.2010])
    image = image * std[:, None, None] + mean[:, None, None]  # [:, None, None] changes the shape from [N] to [N,
    image = image.clamp(0, 1).numpy()
    # convert from CHW to HWC
    # from 3x32x32 to 32x32x3
    return image.transpose(1, 2, 0)

trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=4,
                                          shuffle=True)
dataiter = iter(trainloader)
images, labels = next(dataiter)

fig, axes = plt.subplots(1, len(images), figsize=(10, 2.5))
for idx, image in enumerate(images):
    axes[idx].imshow(convert_to_imshow_format(image))
    axes[idx].set_title(classes[labels[idx]])
    axes[idx].set_xticks([])
    axes[idx].set_yticks([])
```

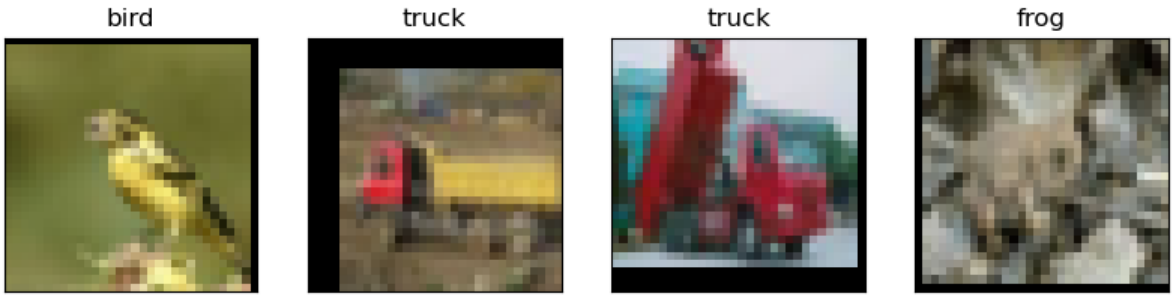


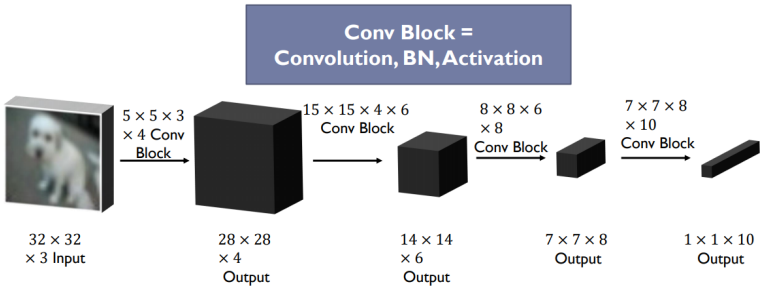
Image Augmentation with Kornia

Here is a code snippet of how we could perform image augmentations batch-wise to speed things up (and also make some of them differentiable).

- Note: image augmentations with Kornia are performed on the batch and are **not** part of `Dataset` and `DataLoader`.

In [ ]:
```
from kornia import augmentation as K
from kornia.augmentation import AugmentationSequential

# when using 'kornia', no need to do augmentaions with `torch`, just:
transform_train = transforms.ToTensor()

# define a sequence of augmentations
aug_list = AugmentationSequential(
    K.ColorJitter(0.1, 0.1, 0.1, 0.1, p=1.0),
    K.RandomAffine(360, [0.1, 0.1], [0.7, 1.2], [30., 50.], p=1.0),
    K.RandomPerspective(0.5, p=1.0),
    return_transform=False,
    same_on_batch=False,
)

img_aug = aug_list(img_tensor)  # [batch_size, num_ch, h, w]
```

Building a CNN-Classifier for CIFAR-10 with PyTorch



In [2]:
```
class CifarCNN(nn.Module):
    """CNN for the CIFAR-10 Datset"""

    def __init__(self):
        """CNN Builder."""
        super(CifarCNN, self).__init__()

        self.conv_layer = nn.Sequential(

            # Conv Layer block 1
            nn.Conv2d(in_channels=3, out_channels=32, kernel_size=3, padding=1),
            # `paading=1` is the same as `padding='same'` for 3x3 kernels size
            nn.BatchNorm2d(32),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=32, out_channels=64, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),  # input_resolution / 2 = 32 / 2 = 16

            # Conv Layer block 2
            nn.Conv2d(in_channels=64, out_channels=128, kernel_size=3, padding=1),
            nn.BatchNorm2d(128),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=128, out_channels=128, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),  # input_resolution / 4 = 32 / 4 = 8
            nn.Dropout2d(p=0.05),

            # Conv Layer block 3
            nn.Conv2d(in_channels=128, out_channels=256, kernel_size=3, padding=1),
            nn.BatchNorm2d(256),
            nn.ReLU(inplace=True),
            nn.Conv2d(in_channels=256, out_channels=256, kernel_size=3, padding=1),
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=2, stride=2),  # input_resolution / 8 = 32 / 8 = 4
            # the output dimensions: [batch_size, 256, h=input_resolution / 8, w=input_resolution / 8]
        )


        self.fc_layer = nn.Sequential(
            nn.Dropout(p=0.1),
            nn.Linear(4096, 1024),  # 256 * 4 * 4 = 4096
```

```
            nn.ReLU(inplace=True),
            nn.Linear(1024, 512),
            nn.ReLU(inplace=True),
            nn.Dropout(p=0.1),
            nn.Linear(512, 10)
        )


    def forward(self, x):
        """Perform forward."""

        # conv layers
        x = self.conv_layer(x)  # [batch_size, channels=256, h_f=4, w_f=4]

        # flatten - can also use nn.Flatten() in __init__() instead
        x = x.view(x.size(0), -1)  # [batch_size, channels * h_f * w_f=4096]

        # fc layer
        x = self.fc_layer(x)  # [batch_size, n_classes=10]

        return x
```

More Conv2D Properties

- `torch.nn.Conv2D` has more hyperparameters you should know of.
- In detail: `torch.nn.Conv2d(in_channels, out_channels, kernel_size, stride=1, padding=0, dilation=1, groups=1, bias=True, padding_mode='zeros', device=None, dtype=None)`
  - `dilation` controls the spacing between the kernel points, also known as the à trous algorithm ("atrous" convolution). This is usually used to upsample the input maps (e.g., for image segmentation and generation tasks, wher ethe output is an image). Visualization of `dilation`.
  - `groups` controls the connections between inputs and outputs. `in_channels` and `out_channels` must both be divisible by groups. Examples:
    - At `groups=1` (default), all inputs are convolved to all outputs.
    - At `groups=in_channels`, each input channel is convolved with its own set of filters (of size $\frac{\text{out-channels}}{\text{in-channels}}$). When `groups == in_channels` and `out_channels == K * in_channels`, where K is a positive integer, this operation is also known as a **"depthwise convolution"**.
  - `padding_mode` (from `torch` 1.1) controls the values of the padding, can be `'zeros', 'reflect', 'replicate' or 'circular'`. Default: `'zeros'`.
  - From `torch` 1.13, you can use strings to control the padding. `padding='valid'` is the same as no padding. `padding='same'` pads the input so the output has the shape as the input.

`same` convolution:

kernel size	padding
3	1
5	2
7	3

In [3]:
```
# how can we calcualte the output of the convolution automatically?
dummy_input = torch.zeros([1, 3, 32, 32])
dummy_model = CifarCNN()
dummy_output = dummy_model.conv_layer(dummy_input)
print(dummy_output.shape)
dummy_output = dummy_output.view(dummy_output.size(0), -1)
print(dummy_output.shape)
# how many weights (trainable parameters) we have in our model?
num_trainable_params = sum([p.numel() for p in dummy_model.parameters() if p.requires_grad])
print("num trainable weights: ", num_trainable_params)
```

```
torch.Size([1, 256, 4, 4])
torch.Size([1, 4096])
num trainable weights:  5852170
```

In [4]:
```
# calculate the model size on disk
num_trainable_params = sum([p.numel() for p in dummy_model.parameters() if p.requires_grad])
param_size = 0
for param in dummy_model.parameters():
    param_size += param.nelement() * param.element_size()
buffer_size = 0
for buffer in dummy_model.buffers():
```

```
    buffer_size += buffer.nelement() * buffer.element_size()
size_all_mb = (param_size + buffer_size) / 1024 ** 2
print(f"model size: {size_all_mb:.2f} MB")
```

```
model size: 22.33 MB
```

In [5]:
```
# time to train our model
# hyper-parameters
batch_size = 128
learning_rate = 1e-4
epochs = 20

# dataloaders - creating batches and shuffling the data
trainloader = torch.utils.data.DataLoader(
    trainset, batch_size=batch_size, shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(
    testset, batch_size=batch_size, shuffle=False, num_workers=2)

# device - cpu or gpu?
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")

# loss criterion
criterion = nn.CrossEntropyLoss()  # accepts 'logits' - unnormalized scores (no need to apply `softmax` manually)

# build our model and send it to the device
model = CifarCNN().to(device) # no need for parameters as we alredy defined them in the class

# optimizer - SGD, Adam, RMSProp...
optimizer = torch.optim.Adam(model.parameters(), lr=learning_rate)
```

In [6]:
```
# function to calcualte accuracy of the model
def calculate_accuracy(model, dataloader, device):
    model.eval() # put in evaluation mode,  turn off Dropout, BatchNorm uses learned statistics
    total_correct = 0
    total_images = 0
    confusion_matrix = np.zeros([10, 10], int)
    with torch.no_grad():
        for data in dataloader:
            images, labels = data
            images = images.to(device)
            labels = labels.to(device)
            outputs = model(images)
            _, predicted = torch.max(outputs.data, 1)
            total_images += labels.size(0)
            total_correct += (predicted == labels).sum().item()
            for i, l in enumerate(labels):
                confusion_matrix[l.item(), predicted[i].item()] += 1

    model_accuracy = total_correct / total_images * 100
    return model_accuracy, confusion_matrix
```

In [7]:
```
# training loop
for epoch in range(1, epochs + 1):
    model.train()  # put in training mode, turn on Dropout, BatchNorm uses batch's statistics
    running_loss = 0.0
    epoch_time = time.time()
    for i, data in enumerate(trainloader, 0):
        # get the inputs
        inputs, labels = data
        # send them to device
        inputs = inputs.to(device)
        labels = labels.to(device)
        # augmentation with `kornia` happens here inputs = aug_list(inputs)

        # forward + backward + optimize
        outputs = model(inputs)  # forward pass
        loss = criterion(outputs, labels)  # calculate the loss
        # always the same 3 steps
        optimizer.zero_grad()  # zero the parameter gradients
        loss.backward()  # backpropagation
        optimizer.step()  # update parameters

        # print statistics
        running_loss += loss.data.item()

    # Normalizing the loss by the total number of train batches
    running_loss /= len(trainloader)

    # Calculate training/test set accuracy of the existing model
    train_accuracy, _ = calculate_accuracy(model, trainloader, device)
```

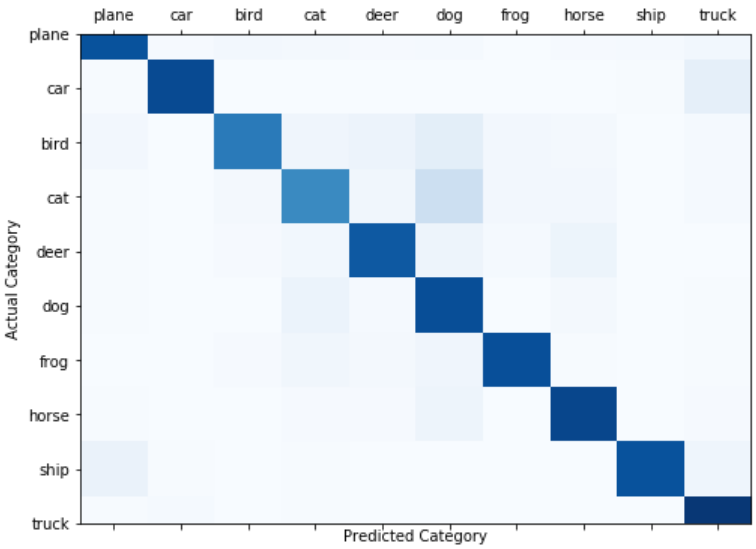
```
    test_accuracy, _ = calculate_accuracy(model, testloader, device)

    log = "Epoch: {} | Loss: {:.4f} | Training accuracy: {:.3f}% | Test accuracy: {:.3f}% | ".format(epoch, runni
    # with f-strings
    # log = f"Epoch: {epoch} | Loss: {running_loss:.4f} | Training accuracy: {train_accuracy:.3f}% | Test accurac
    epoch_time = time.time() - epoch_time
    log += "Epoch Time: {:.2f} secs".format(epoch_time)
    # with f-strings
    # log += f"Epoch Time: {epoch_time:.2f} secs"
    print(log)

    # save model
    if epoch % 20 == 0:
        print('==> Saving model ...')
        state = {
            'net': model.state_dict(),
            'epoch': epoch,
        }
        if not os.path.isdir('checkpoints'):
            os.mkdir('checkpoints')
        torch.save(state, './checkpoints/cifar_cnn_ckpt.pth')

print('==> Finished Training ...')
```

```
Epoch: 1 | Loss: 1.5168 | Training accuracy: 56.276% | Test accuracy: 57.460% | Epoch Time: 61.60 secs
Epoch: 2 | Loss: 1.0972 | Training accuracy: 65.480% | Test accuracy: 66.850% | Epoch Time: 82.20 secs
Epoch: 3 | Loss: 0.9108 | Training accuracy: 70.738% | Test accuracy: 70.790% | Epoch Time: 85.79 secs
Epoch: 4 | Loss: 0.8017 | Training accuracy: 71.934% | Test accuracy: 72.200% | Epoch Time: 82.55 secs
Epoch: 5 | Loss: 0.7141 | Training accuracy: 77.510% | Test accuracy: 77.880% | Epoch Time: 90.47 secs
Epoch: 6 | Loss: 0.6533 | Training accuracy: 78.744% | Test accuracy: 77.530% | Epoch Time: 80.15 secs
Epoch: 7 | Loss: 0.6037 | Training accuracy: 80.628% | Test accuracy: 79.630% | Epoch Time: 83.42 secs
Epoch: 8 | Loss: 0.5586 | Training accuracy: 83.364% | Test accuracy: 81.740% | Epoch Time: 89.09 secs
Epoch: 9 | Loss: 0.5291 | Training accuracy: 83.362% | Test accuracy: 81.300% | Epoch Time: 81.84 secs
Epoch: 10 | Loss: 0.4992 | Training accuracy: 84.848% | Test accuracy: 82.760% | Epoch Time: 83.53 secs
Epoch: 11 | Loss: 0.4725 | Training accuracy: 85.310% | Test accuracy: 83.250% | Epoch Time: 88.62 secs
Epoch: 12 | Loss: 0.4396 | Training accuracy: 86.168% | Test accuracy: 83.710% | Epoch Time: 92.91 secs
Epoch: 13 | Loss: 0.4224 | Training accuracy: 86.700% | Test accuracy: 84.120% | Epoch Time: 79.77 secs
Epoch: 14 | Loss: 0.4053 | Training accuracy: 87.594% | Test accuracy: 84.380% | Epoch Time: 80.96 secs
Epoch: 15 | Loss: 0.3809 | Training accuracy: 88.878% | Test accuracy: 85.430% | Epoch Time: 80.96 secs
Epoch: 16 | Loss: 0.3620 | Training accuracy: 88.470% | Test accuracy: 84.940% | Epoch Time: 82.44 secs
Epoch: 17 | Loss: 0.3496 | Training accuracy: 89.608% | Test accuracy: 85.490% | Epoch Time: 80.95 secs
Epoch: 18 | Loss: 0.3281 | Training accuracy: 89.878% | Test accuracy: 85.800% | Epoch Time: 83.10 secs
Epoch: 19 | Loss: 0.3210 | Training accuracy: 90.690% | Test accuracy: 86.050% | Epoch Time: 80.14 secs
Epoch: 20 | Loss: 0.3026 | Training accuracy: 88.678% | Test accuracy: 84.950% | Epoch Time: 82.72 secs
==> Saving model ...
==> Finished Training ...
```

In [8]:

```
# load model, calculate accuracy and confusion matrix
model = CifarCNN().to(device)
state = torch.load('./checkpoints/cifar_cnn_ckpt.pth', map_location=device)
model.load_state_dict(state['net'])
# note: `map_location` is necessary if you trained on the GPU and want to run inference on the CPU

test_accuracy, confusion_matrix = calculate_accuracy(model, testloader, device)
print("test accuracy: {:.3f}%".format(test_accuracy))

# plot confusion matrix
fig, ax = plt.subplots(1,1,figsize=(8,6))
ax.matshow(confusion_matrix, aspect='auto', vmin=0, vmax=1000, cmap=plt.get_cmap('Blues'))
plt.ylabel('Actual Category')
plt.yticks(range(10), classes)
plt.xlabel('Predicted Category')
plt.xticks(range(10), classes)
plt.show()
```

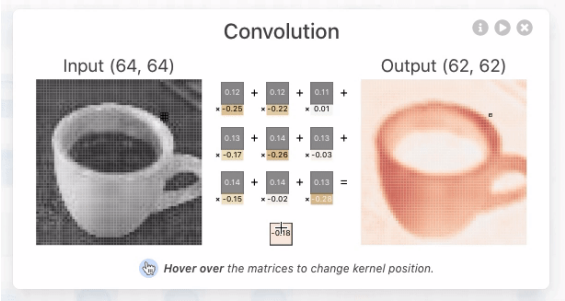
```
test accuracy: 84.950%
```



Visualizing CNN Filters

CNN Explainer Demo

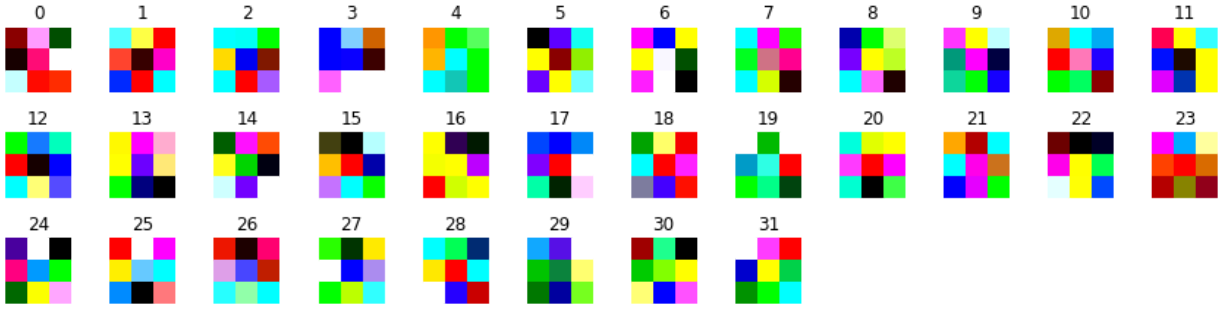
Open Demo



In [9]:
```
# visualize filters
# see available layers, in our case, the stacked layers are called "conv_layer"
print(model.conv_layer)
# extracting the model features at the particular layer number
layer = model.conv_layer[0]
# get the weights
weight_tensor = layer.weight.data.cpu()

# get the number of kernals
num_kernels = weight_tensor.shape[0]

#define number of columns for subplots
num_cols = 12
# rows = num of kernels
num_rows = num_kernels

#set the figure size
fig = plt.figure(figsize=(num_cols, num_rows))

# looping through all the kernels
for i in range(weight_tensor.shape[0]):
    ax1 = fig.add_subplot(num_rows, num_cols, i+1)

    #for each kernel, we convert the tensor to numpy
    npimg = np.array(weight_tensor[i].numpy(), np.float32)
    #standardize the numpy image
    npimg = (npimg - np.mean(npimg)) / np.std(npimg)
    npimg = np.minimum(1, np.maximum(0, (npimg + 0.5)))
    npimg = npimg.transpose((1, 2, 0))
    ax1.imshow(npimg)
    ax1.axis('off')
    ax1.set_title(str(i))
```

```
    ax1.set_xticklabels([])
    ax1.set_yticklabels([])

plt.tight_layout()
```

```
Sequential(
  (0): Conv2d(3, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (2): ReLU(inplace=True)
  (3): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (4): ReLU(inplace=True)
  (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (6): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (7): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (8): ReLU(inplace=True)
  (9): Conv2d(128, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (10): ReLU(inplace=True)
  (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (12): Dropout2d(p=0.05, inplace=False)
  (13): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (14): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
  (15): ReLU(inplace=True)
  (16): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (17): ReLU(inplace=True)
  (18): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
)
```



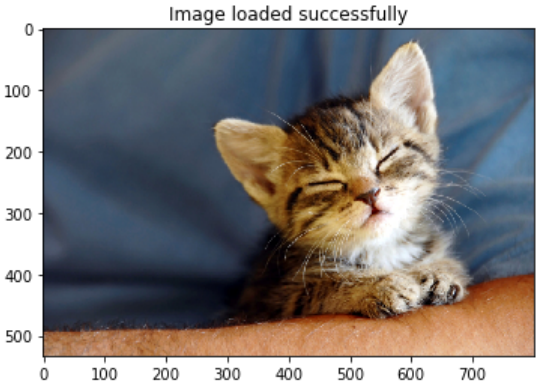
Visualizing Layer Output

- We can see which neurons are active for every input image.
- This way we can a better understanding of what the network sees during forward pass, which probably affects the final prediction.
- Let's see an example by Sarthak Gupta .

In [10]:
```
# helper functions
def to_grayscale(image):
    """
    input is (d,w,h)
    converts 3D image tensor to grayscale images corresponding to each channel
    """
    image = torch.sum(image, dim=0)
    image = torch.div(image, image.shape[0])
    return image

def normalize(image, device=torch.device("cpu")):
    normalize = transforms.Normalize(
    mean=[0.485, 0.456, 0.406],
    std=[0.229, 0.224, 0.225]
    )
    preprocess = transforms.Compose([
    transforms.Resize((224,224)),
    transforms.ToTensor(),
    normalize
    ])
    image = preprocess(image).unsqueeze(0).to(device)
    return image


def predict(image, model, labels=None):
    _, index = model(image).data[0].max(0)
    if labels is not None:
        return str(index.item()), labels[str(index.item())][1]
```

```
    else:
        return str(index.item())

def deprocess(image, device=torch.device("cpu")):
    return image * torch.tensor([0.229, 0.224, 0.225]).to(device) + torch.tensor([0.485, 0.456, 0.406]).to(device

def load_image(path):
    image = Image.open(path)
    plt.imshow(image)
    plt.title("Image loaded successfully")
    return image
```

In [3]:
```
# load sample image
kitten_img = load_image("./assets/kitten.jpg")
```



Torchvision Pre-Trained Models

- The `torchvision.models` subpackage contains definitions of models for addressing different tasks, including: image classification, pixelwise semantic segmentation, object detection, instance segmentation, person keypoint detection, video classification, and optical flow.
- You can see all the available here - Models and pre-trained weights.
  - In code, you can use `torchvision.models.list_models` to see a list of all available models.
- Examples:

In [ ]:
```
# examples of laoding pre-trained models - not meant to run, check out the link in previous block.
from torchvision.models import resnet50, ResNet50_Weights

# Example 1
# Using pretrained weights:
resnet50(weights=ResNet50_Weights.IMAGENET1K_V1)
resnet50(weights="IMAGENET1K_V1")
resnet50(pretrained=True)  # old approach, deprecated

# Using no weights:
resnet50(weights=None)
resnet50()
resnet50(pretrained=False)  # old approach, deprecated

# Initialize the Weight Transforms
weights = ResNet50_Weights.DEFAULT
preprocess = weights.transforms()

# Apply it to the input image
img_transformed = preprocess(img)
```

In [ ]:
```
# Example 2
from torchvision.io import read_image
from torchvision.models import resnet50, ResNet50_Weights

img = read_image("test/assets/encode_jpeg/grace_hopper_517x606.jpg")

# Step 1: Initialize model with the best available weights
weights = ResNet50_Weights.DEFAULT
model = resnet50(weights=weights)
model.eval()

# Step 2: Initialize the inference transforms
preprocess = weights.transforms()
```

```
# Step 3: Apply inference preprocessing transforms
batch = preprocess(img).unsqueeze(0)

# Step 4: Use the model and print the predicted category
prediction = model(batch).squeeze(0).softmax(0)
class_id = prediction.argmax().item()
score = prediction[class_id].item()
category_name = weights.meta["categories"][class_id]
print(f"{category_name}: {100 * score:.1f}%")
```

In [4]:
```
# load pre-trained model
device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
model = models.vgg16(pretrained=True).to(device)
# we are not interested in training this network, so no need for gradient calculation
model.requires_grad_(False)
# put in evaluation mode to turn-off DropOut and use BatchNorm's running mean and std (not the batch mean and sta
model.eval();
```

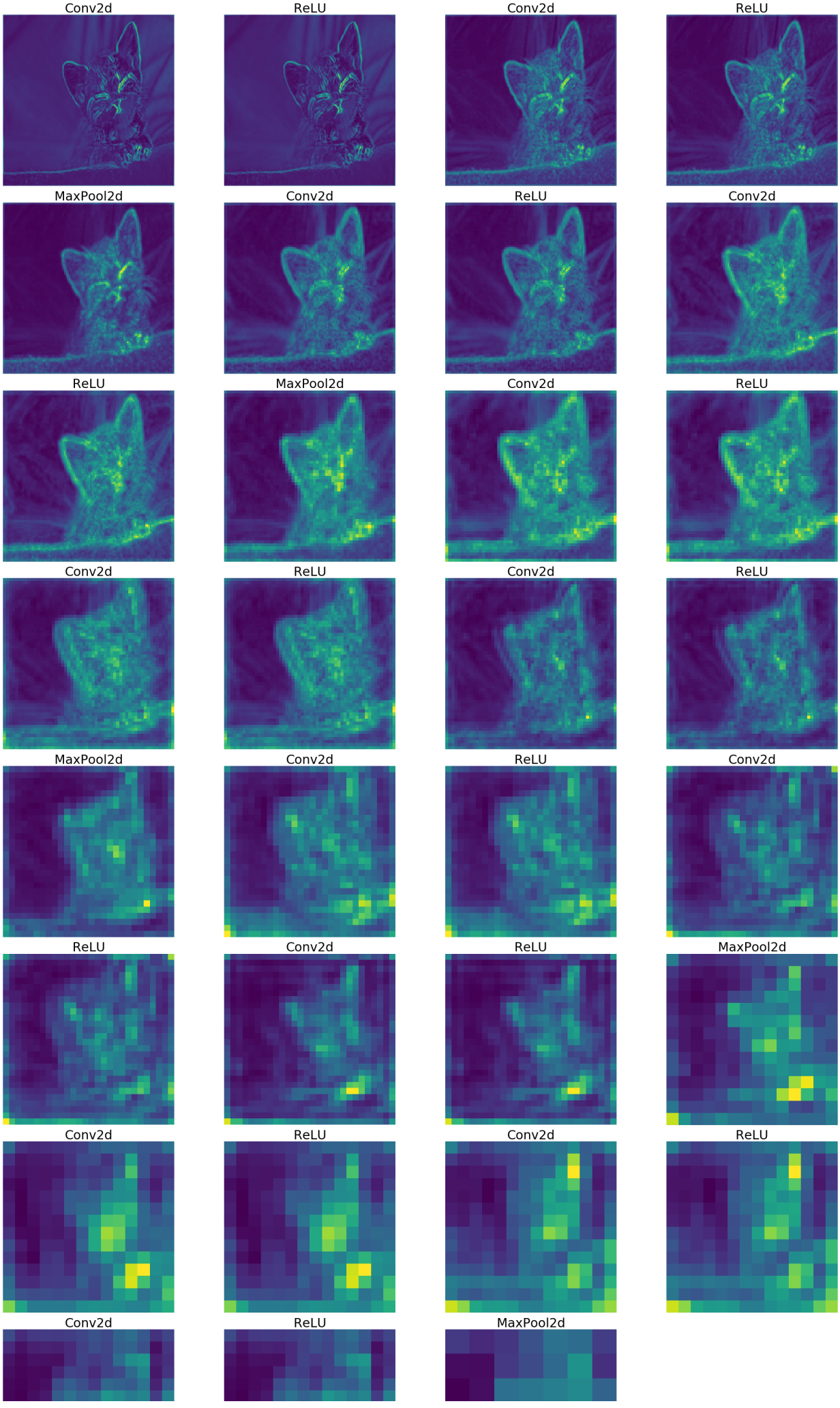
In [12]:
```
# pre-process image and predict label
prep_img = normalize(kitten_img, device)
print("predicted class:", predict(prep_img, model))
```

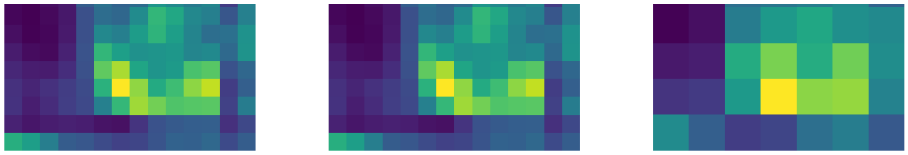
```
predicted class: 281
```

Output of Each Layer

In [17]:
```
def layer_outputs(image, model):
    modulelist = list(model.features.modules())
    outputs = []
    names = []
    for layer in modulelist[1:]:
        image = layer(image)
        outputs.append(image)
        names.append(str(layer))

    output_im = []
    for i in outputs:
        i = i.squeeze(0)
        temp = to_grayscale(i)  # take the mean
        output_im.append(temp.data.cpu().numpy())

    fig = plt.figure(figsize=(30, 50))

    for i in range(len(output_im)):
        a = fig.add_subplot(8, 4, i+1)
        imgplot = plt.imshow(output_im[i])
        a.set_axis_off()
        a.set_title(names[i].partition('(')[0], fontsize=30)
    plt.tight_layout()
#     plt.savefig('layer_outputs.jpg', bbox_inches='tight')
```

In [18]:
```
layer_outputs(prep_img, model)
```


Conv2d
ReLU
Conv2d
ReLU
MaxPool2d
Conv2d
ReLU
Conv2d
ReLU
MaxPool2d
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
MaxPool2d
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
MaxPool2d
Conv2d
ReLU
Conv2d
ReLU
Conv2d
ReLU
MaxPool2d



Output of Each Filter for a Certain Layer

In [22]:
```
def filter_outputs(image, model, layer_to_visualize):
    modulelist = list(model.features.modules())
    if layer_to_visualize < 0:
        layer_to_visualize += 31
    output = None
    name = None
    for count, layer in enumerate(modulelist[1:]):
        image = layer(image)
        if count == layer_to_visualize:
            output = image
            name = str(layer)

    filters = []
    output = output.data.squeeze().cpu().numpy()
    for i in range(output.shape[0]):
        filters.append(output[i,:,:])

    fig = plt.figure(figsize=(10, 10))

    for i in range(int(np.sqrt(len(filters))) * int(np.sqrt(len(filters)))):
        ax = fig.add_subplot(np.sqrt(len(filters)), np.sqrt(len(filters)), i+1)
        imgplot = ax.imshow(filters[i])
        ax.set_axis_off()
    plt.tight_layout()
```

In [23]:
```
filter_outputs(prep_img, model, 0)
```

 Are CNNs the Holy Grail? The Problem with CNNs

Deep NNs are sensitive to adversarial attacks.

- For example: consider the following image, where on the left, we have an image of a pig that is correctly classified by a state-of-the-art convolutional neural network.
- After perturbing the image slightly (every pixel is in the range [0, 1] and changed by at most 0.005), the network now returns class “airliner” with high confidence.

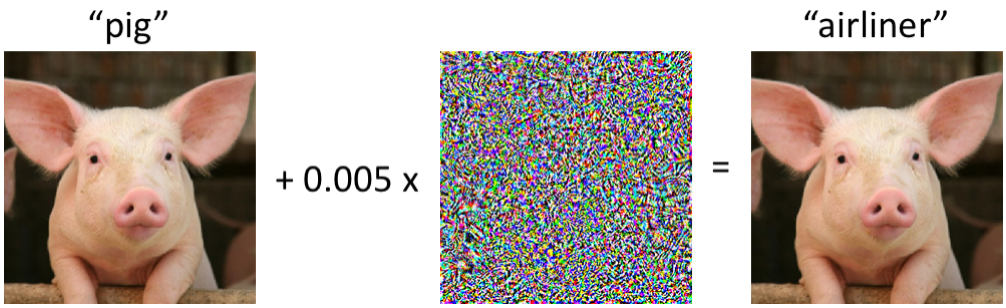


Image Source

Recognition algorithms generalize poorly to new environments

(A) **Cow: 0.99**, Pasture: 0.99, Grass: 0.99, No Person: 0.98, Mammal: 0.98

(B) No Person: 0.99, Water: 0.98, Beach: 0.97, Outdoors: 0.97, Seashore: 0.97

(C) No Person: 0.97, **Mammal: 0.96**, Water: 0.94, Beach: 0.94, Two: 0.94

Recognition in Terra Incognita (Beery et al., 2018)

Neural Networks tend to exhibit undesirable biases

Fig. 8: Pairs of pictures (columns) sampled over the Internet along with their prediction by a ResNet-101.

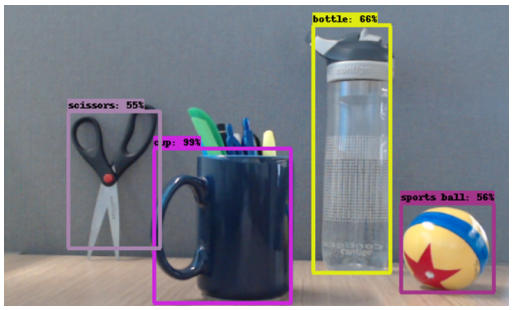
- The reasons why the model learns these biases are unclear.
  - One hypothesis is that despite the balanced distribution of races in pictures labeled basketball, black persons are more represented in this class in comparison to the other classes.

ConvNets and ImageNet Beyond Accuracy: Understanding Mistakes and Uncovering Biases (Stock and Cisse, 2018)

CNNs Applications in Computer Vision

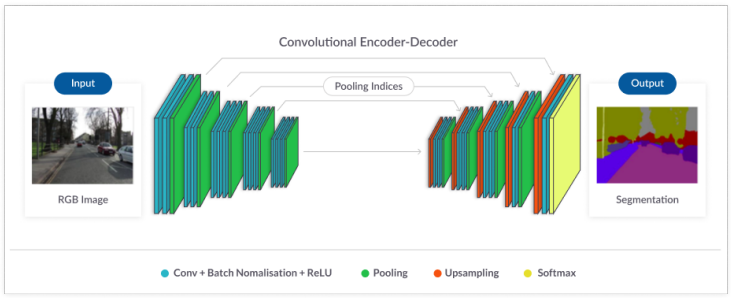
See 1000+ computer vision tasks with benchmarks and papers on PapersWithCode.com.

- **Object Detection**



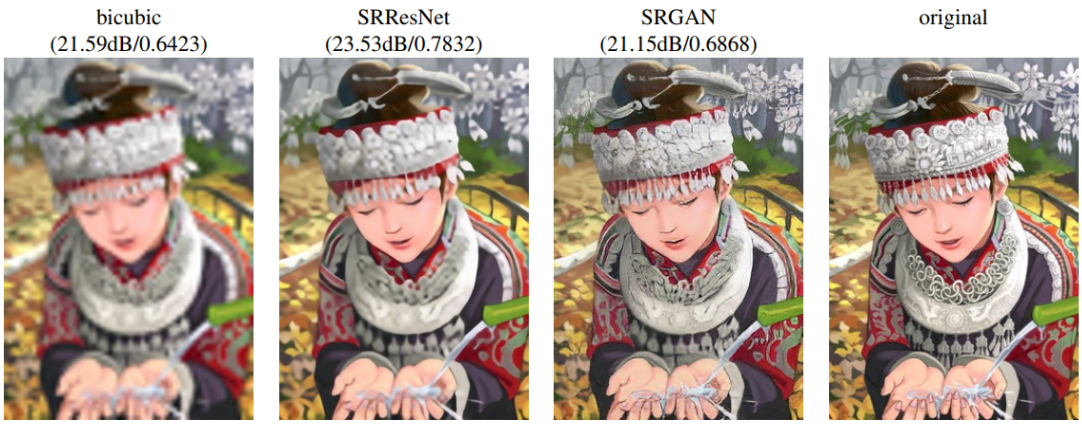
Source

- **Semantic Segmentation**



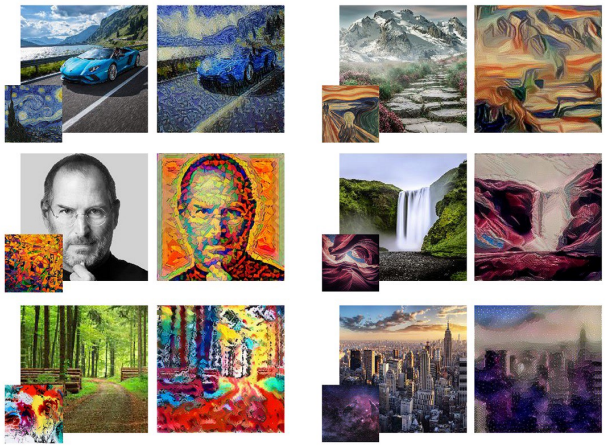
Source

- **Super Resolution**



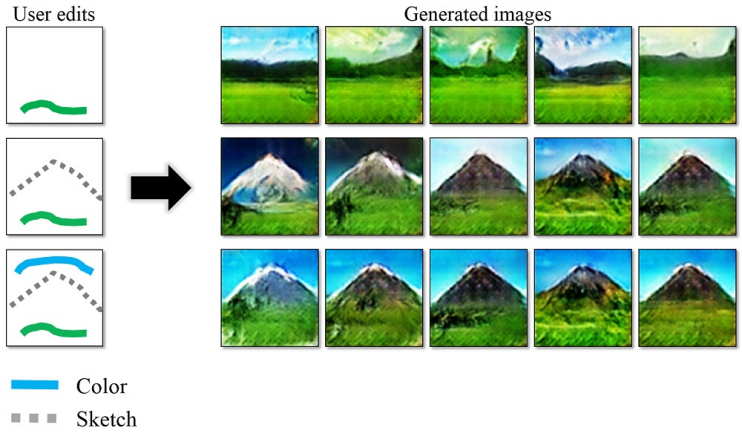
Source

- **Style Transfer**



Source

- **Image Editing**

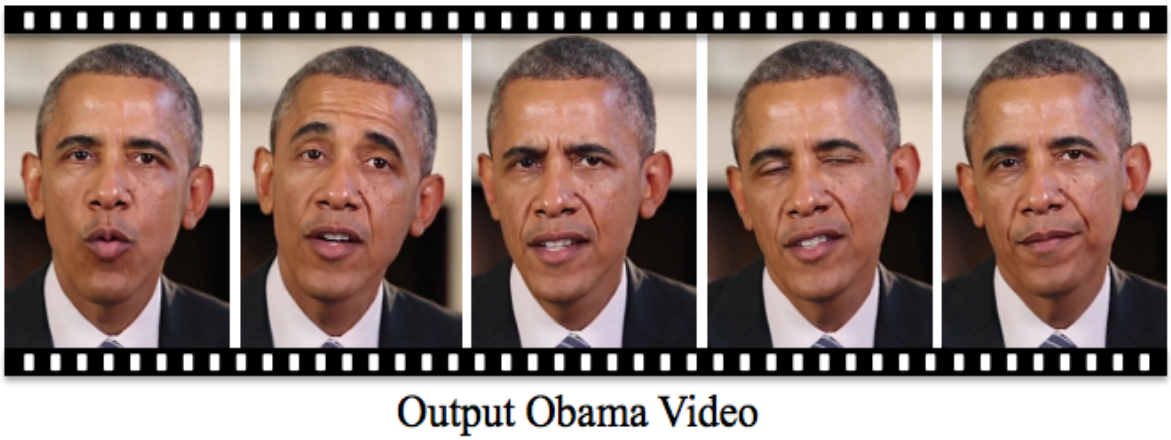


Source

- **Image Generation**

StyleGAN (V1-3) - thispersondoesnotexist.com

- **Multi-Signals** Synthesizing Obama: Learning Lip Sync from Audio



Source

Recommended Videos

Warning!

- These videos do not replace the lectures and tutorials.
- Please use these to get a better understanding of the material, and not as an alternative to the written material.

Video By Subject

- Convolutional Neural Networks - Convolutional Neural Networks | MIT 6.S191
  - A previous version of this lecture - Convolutional Neural Networks | MIT 6.S191
- Deep Neural Networks with PyTorch - Stefan Otte: Deep Neural Networks with PyTorch | PyData Berlin 2018

Credits

- Icons made by Becris from www.flaticon.com
- Icons from Icons8.com - https://icons8.com
- Some slides from CS131 and CS231n (Stanford)
- Deep Learning with Pytorch on CIFAR10 Dataset - Zhenye's Blog
- CIFAR-10 Classifier Using CNN in PyTorch - Stefan Fiott