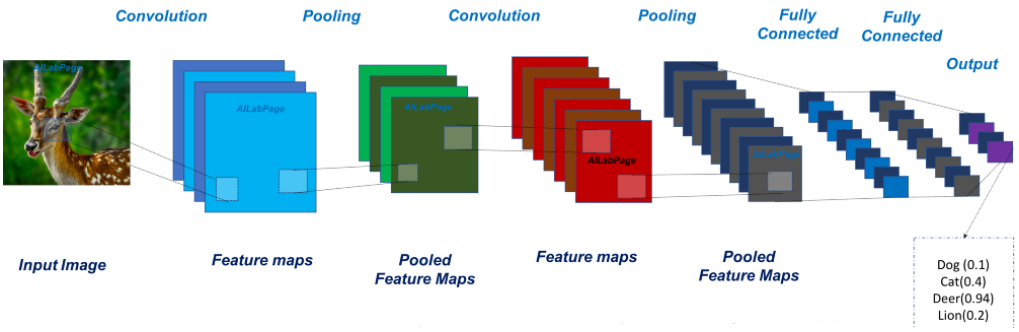
EE 046746 - Technion - Computer Vision

Tutorial 04-05 - Convolution & Deep Learning



- Image Source

Agenda

- Multi-Layer Perceptrons)
- 2D Convolution
- Convolution-based Classification
- Convolutional Neural Networks (CNNs))
- Regularization
- Data Augmentation
- CIFAR-10 Classification with PyTorch
- The CNN Story
- Other Applications of CNNs
- The Problem with CNNs
- Recommended Videos
- Credits

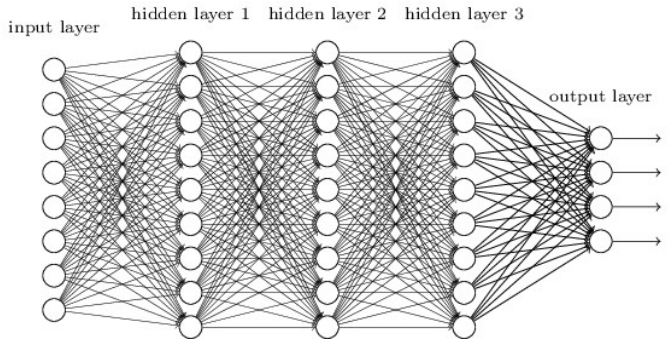
In [1]:

```
# 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
```

Multi-Layer Perceptron (MLP)

- Other names: Fully-Connected (FC) Network (FCN), Dense Network
- An MLP is composed of one input layer, one or more hidden layers and a final output layer.

- Every layer, except the output layer includes a bias neuron which is fully connected to the next layer.
- When the number of hidden layers is larger than 2, the network is usually called a deep neural network (DNN).



Multi-Layer Perceptron (MLP) Cont.

- The algorithm is composed of two main parts: forward pass and backward pass.
- In the forward pass, for each training instance, the algorithm feeds it to the network and computes the output of every neuron in each consecutive layer (using the network for prediction is just doing a forward pass).
- Then, the output error (the difference between the desired output and the actual output from the network) is computed.
- After the output error calculation, the network calculates how much each neuron in the last hidden layer contributed to the output error (using the chain rule).
- It then proceeds to measure how much of these error contributions came from each neuron in the previous layers until reaching the input layer.
- This is the backward pass: measuring the error gradient across all the connection weights in the network by propagating the error gradient backward in the network (this is the backpropagation process).

Multi-Layer Perceptron (MLP) Cont.

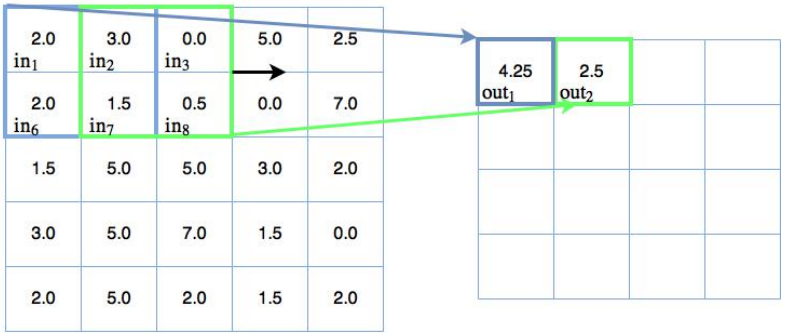
- In short: for each training instance the backpropagation algorithm first makes a prediction (forward pass), measures the error, then goes in reverse to measure the error contribution from each connection (backward pass) and finally, using Gradient Descent, updates the weights in the direction that reduces the error.

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

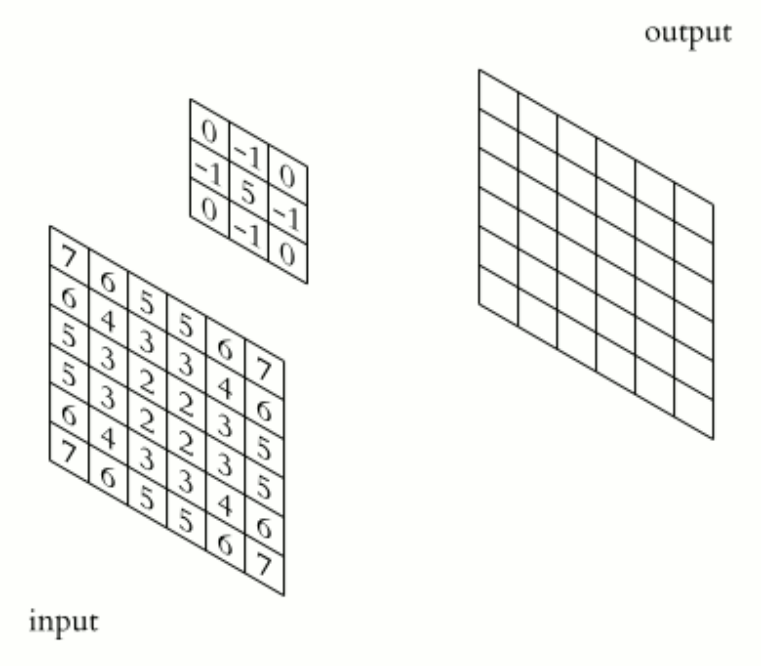
12	3	19
25	10	1
9	7	17

*

1	2
3	4

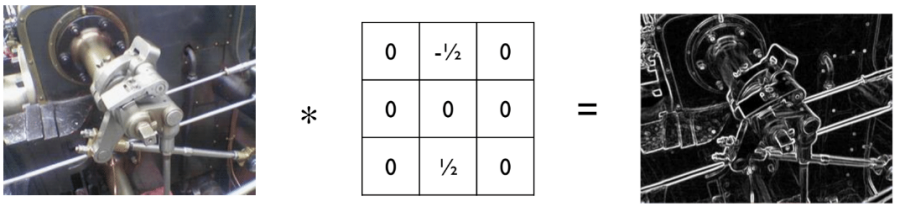
=

133	75
100	101

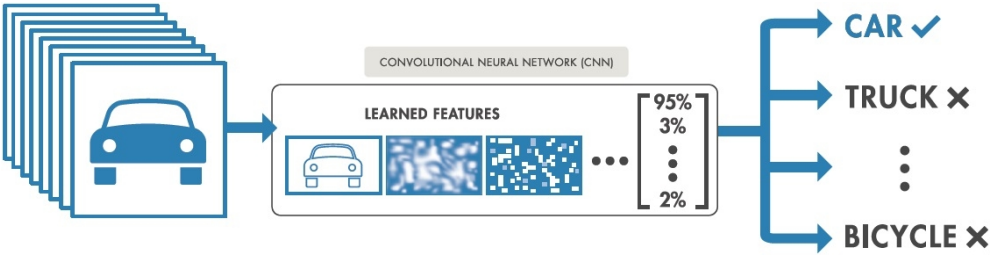


Convolutuion as Feature Extractors for Classification

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



- 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.

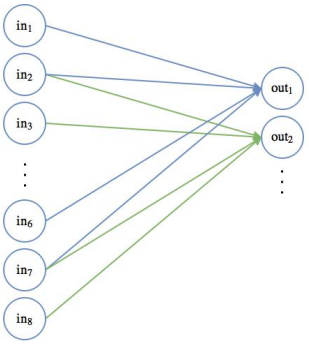


  - Image Source

- 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...
  - So 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! See tutorial 1.5 (Deep Learning and PyTorch basics) for more details.
  - Using convolution, we can capture spatial structures (e.g., pixels the shape a tail).

Convolutional Neural Networks (CNNs)

- Convolutional Neural Networks (CNNs) are deep neural networks that contain layers of stacked convolution layers or filters.
- They are mainy used for image datasets, but are also useful for other areas such as Natural Language Processing (NLP).
- 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, as stated previously, is 0.5 (in this example).

CNNs Properties

Feature mapping and multiple channels

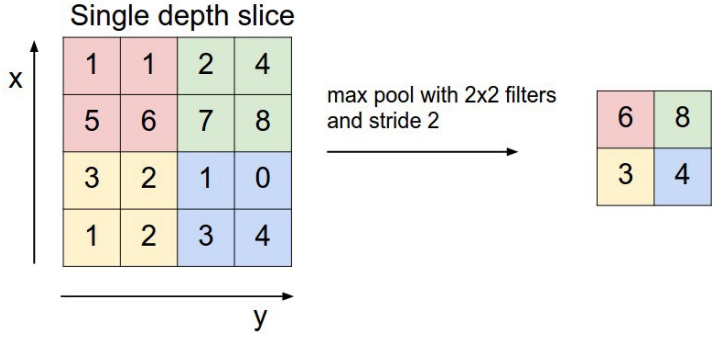
- Since the weights of individual filters are held constant as they are applied over the input nodes, they can be trained to select certain features from the input data.
- In the case of images, it may learn to recognize common geometrical objects such as lines, edges and other shapes which make up objects.
- 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**.

Pooling

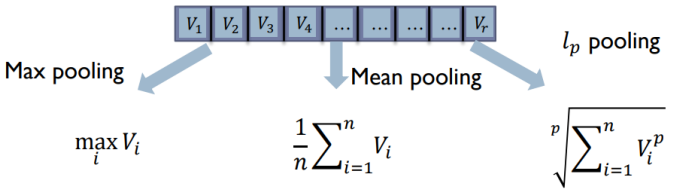
It is a sliding window type technique, but instead of applying weights, which can be trained, it applies a **statistical function** of some type over the contents of its window. The most common type of pooling is called **max pooling**, and it applies the $max()$ function over the contents of the window. 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*
2. It makes feature detection more robust to object orientation and scale changes

- Max-pooling can be seen as "zoom-out", allowing to detect bigger objects with smaller convolutions.



- Image Source

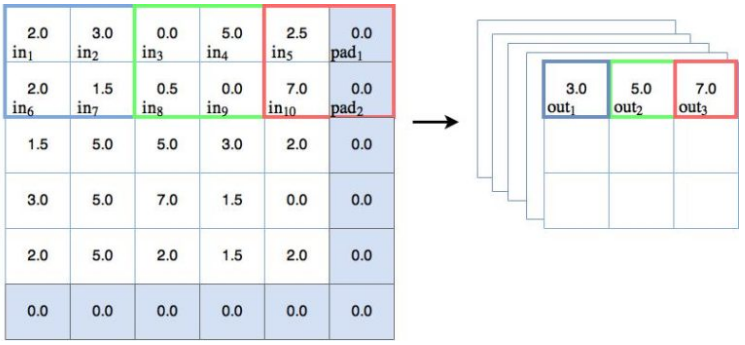


- Other pooling operators:

- 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 in some way "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. An example can be seen here.

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 reduces the number of trainable parameters in the model.



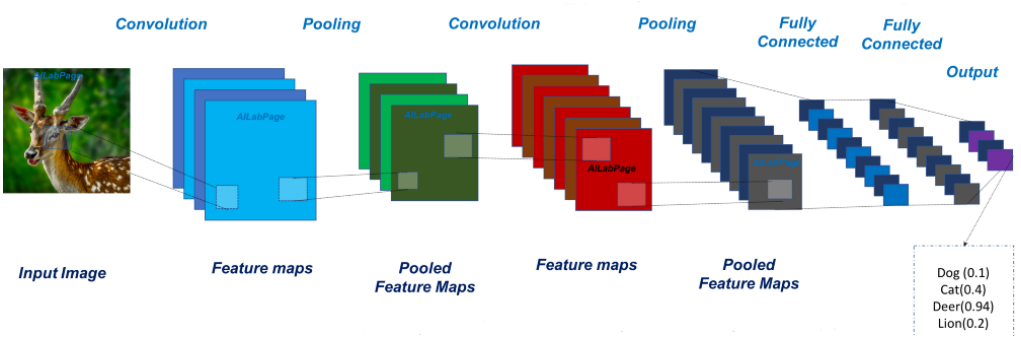
(images from adventuresinmachinelearning.com)

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] and is called *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.

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 convolutuinal 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*.



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 middel level features, usually object parts.
- **High Level** - also called deep features (final layers), which include whole objects (global).

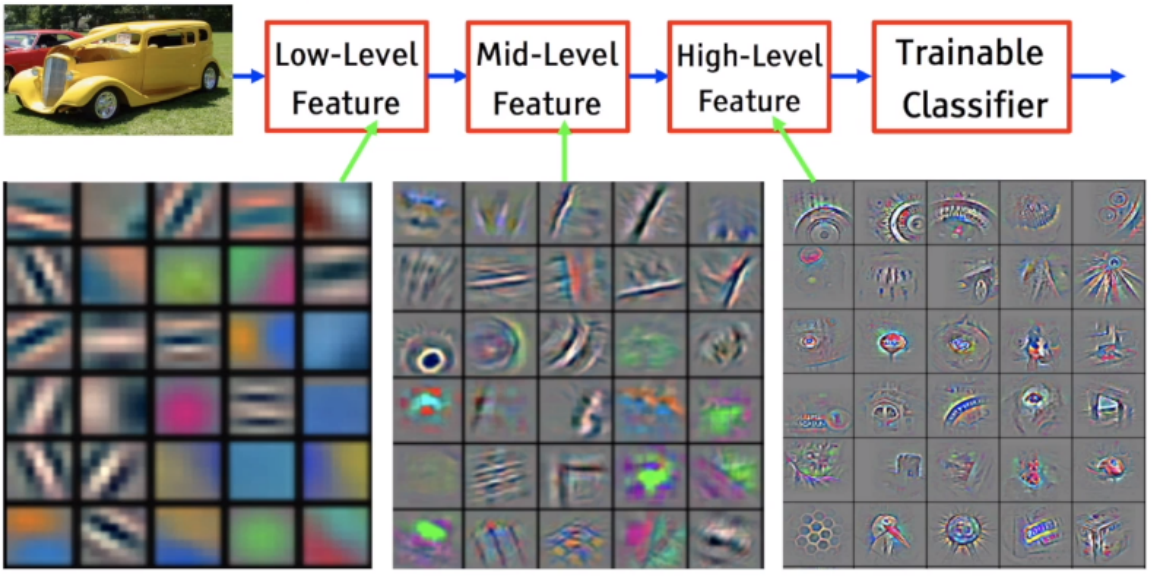


Image Source

Calculating a convolutional layer output size

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:

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

If we also add a parameter called rate/dilation (next tutorial):

- D - rate/dilation

The output width:

$$W_{out} = \left\lfloor \frac{W_{in} - D\,(F-1) + 2P - 1}{S} + 1 \right\rfloor$$

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:

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

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

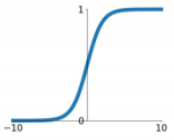
So the input to the FC layer is $7x7 = 49$ (because we have 7 for the width and 7 for the height).

Non-Linear Activations

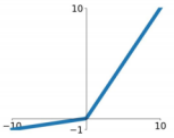
- The key change made to the Perceptron that brought upon the era of deep learning is the addition of **activation function** to the output of each neuron.
- These allow the learning of non-linear functions. A (1-layer) neural network without an activation function is essentially just a linear regression model.

Usually $\psi = g \circ f$
f is the (point-wise) activation function

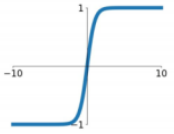
Sigmoid
$\sigma(x) = \frac{1}{1+e^{-x}}$



Leaky ReLU
$\max(0.1x, x)$

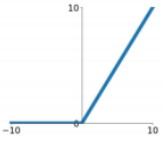


tanh
$\tanh(x)$



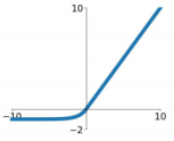
Maxout
$\max(w_1^T x + b_1, w_2^T x + b_2)$

ReLU
$\max(0, x)$



ELU

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



Batch Normalization

- Batch normalization is a technique for improving the speed, performance, and stability of deep neural networks.
  - The reasons behind its effectiveness remain under discussion
- It is used to normalize the input layer by adjusting and scaling the activations.
- Formally:
  - **Input**: $x \in \mathcal{R}^{N \times D}$
  - **Learnable Parameters**: $\gamma, \beta \in \mathcal{R}^{D}$
  - **Intermediates**: $\mu, \sigma \in \mathcal{R}^{D}, \hat{x} \in \mathcal{R}^{N \times D}$
  - **Output**: $y \in \mathcal{R}^{N \times D}$
- In CNNs, we work with inputs of shape $[N, C, H, W]$, where N is the batch size, C is the number of channels and H, W are the height and width of the feature map respectively. BatchNorm in this case is performed **channel-wise**, i.e., on the channel dimension C such that $\gamma, \beta \in \mathcal{R}^{C}$.

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma^2_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma^2_{\mathcal{B}} + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

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 using residual blocks (ResNets).
  - 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.

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...)
- 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.

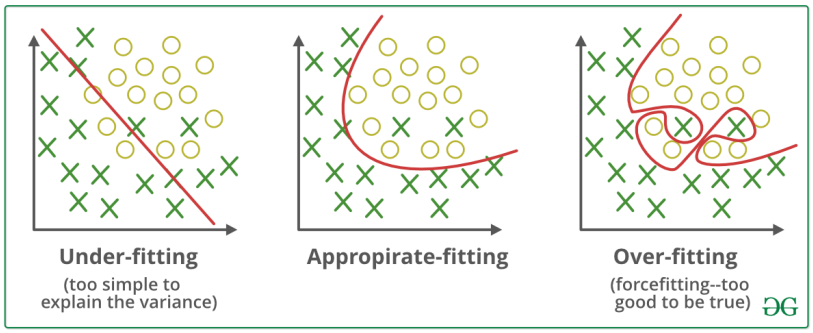


Image Source

- Regularization usually comes in 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 and hence reduce 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()`). In test time, it is turned off (`model.eval()`).

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

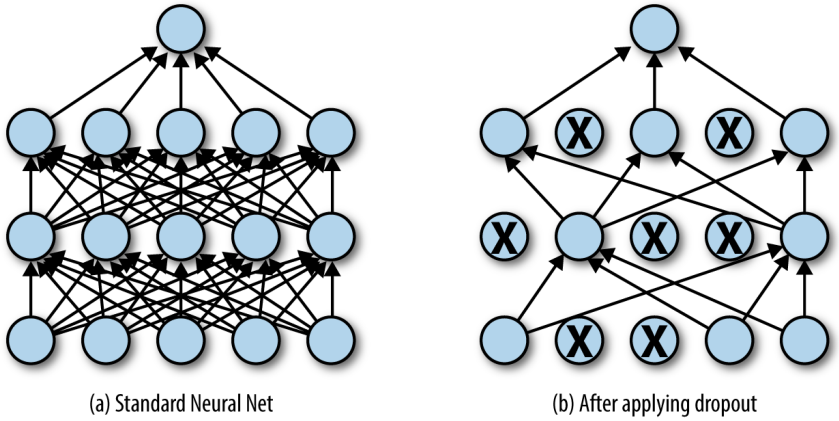


Image Source

Data Augmentation

- Data augmentation is a common technique to improve results and avoid overfitting.
- 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 augementation 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** - Over-fitting 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

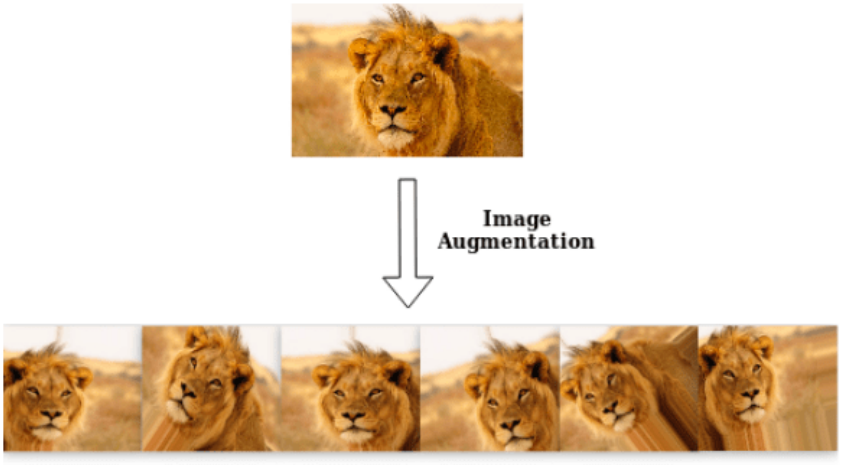
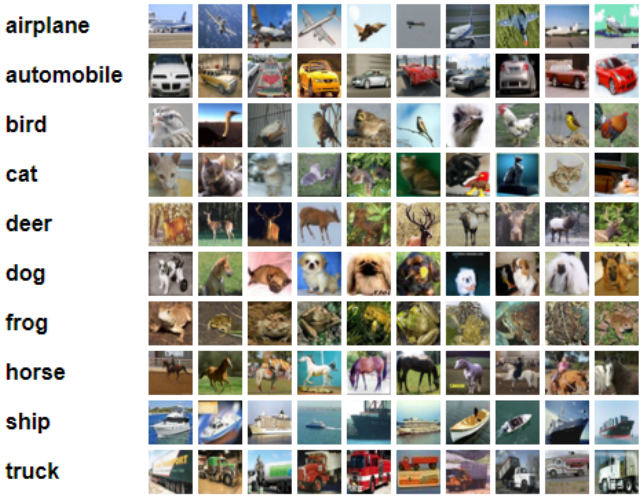


Image Source

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



---------- CODE TIME ----------

In [2]:

```
# define pre-processing steps on the images
# also called "data augementation" (only done for the train set)

transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

# 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)),
])

# 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)
```

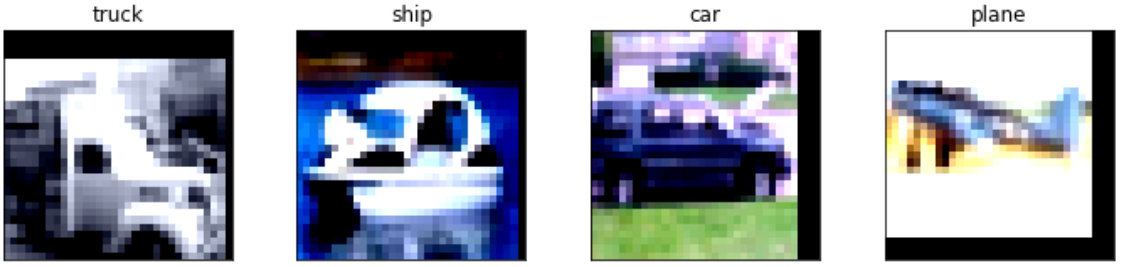
```
Downloading https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz to ./datasets\cifar-10-python.tar.gz

Failed download. Trying https -> http instead. Downloading http://www.cs.toronto.edu/~kriz/cifar-10-python.
tar.gz to ./datasets\cifar-10-python.tar.gz

Files already downloaded and verified
```

In [3]:

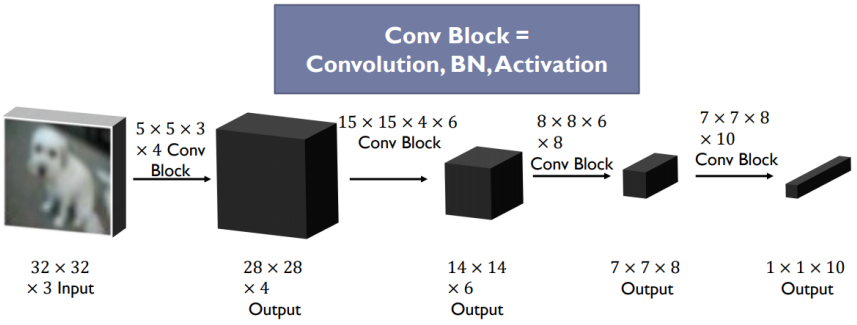
```
# let's see some of the images
def convert_to_imshow_format(image):
    # first convert back to [0,1] range from [-1,1] range - approximately...
    image = image / 2 + 0.5
    image = image.numpy()
    # convert from CHW to HWC
    # from 3x32x32 to 32x32x3
    return image.transpose(1,2,0)

trainloader = torch.utils.data.DataLoader(trainset,
                                          batch_size=4,
                                          shuffle=True)
```

```
dataiter = iter(trainloader)
images, labels = dataiter.next()

fig, axes = plt.subplots(1, len(images), figsize=(12,2.5))
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([])
```

```
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s).
Clipping input data to the valid range for imshow with RGB data ([0..1] for floats or [0..255] for integer
s).
```



Building a CNN-Classifier for CIFAR-10 with PyTorch



In [4]:

```
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),
            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),

            # 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),
            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),
        )


        self.fc_layer = nn.Sequential(
            nn.Dropout(p=0.1),
            nn.Linear(4096, 1024),
            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)

        # flatten
        x = x.view(x.size(0), -1)

        # fc layer
        x = self.fc_layer(x)

        return x
```

In [5]:

```
# calculating the output size of the convolutional layers, before the FC layers
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)
# calculating the number of trainable weights
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
```

Training the CNN Model

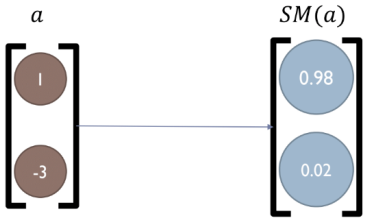
- So we have the model, but how do we train it to output the correct class of the input image?
- As you have probably noticed, the output of the final fully-connected layer is a vector of length 10, which is exactly the number of classes we have!
- We mentioned that we want entry i of the final vector 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

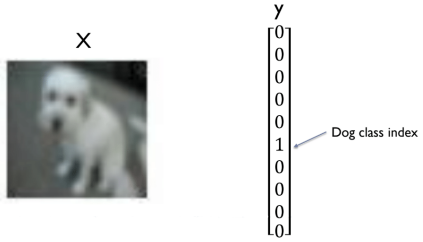
- OK great, we have an output vector of probabilities, so how we predict the label of the input image?
- Simple! Just take the $argmax$:

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

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

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 function, which represents how bad we are currently doing:

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

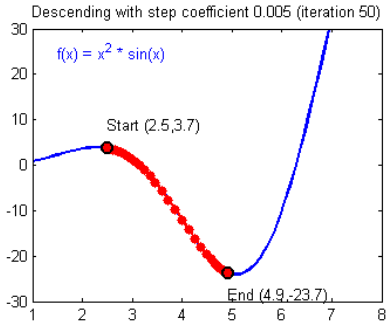
$$p_c = 0.1 \rightarrow \mathcal{L} = -\log(0.1) = 2.3$$

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

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

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

Minimizing the Loss Function with Gradient Descent



- In order to perform Gradient Descent, we need to calculate the dervatives with respect to the network's weights.
- Due to memory reasons, when the dataset is large, we cannot compute the gradients on the whole dataset, and thus we train in **mini-batches** of the data. The `batch_size` is a *hyper-parameter* which needs to be tuned. Usually the sizes are 32, 64, 128...
- In order to propagate the gradients through all of the layers, we need to use the chain rule when calculating the gradients. This is called **backpropagation**.

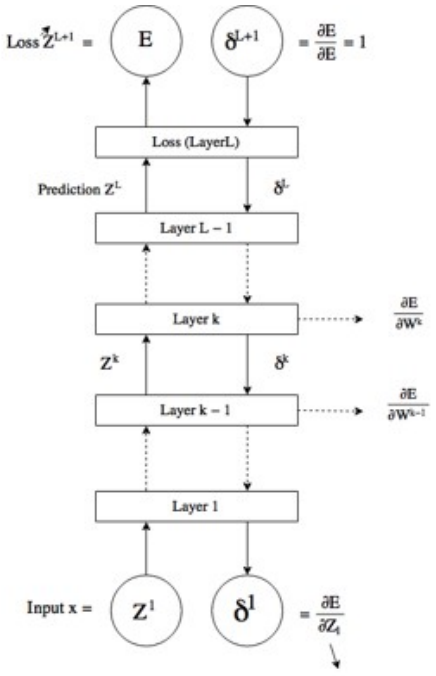
Backpropagation

- Denote the output of the k^{th} layer as $f(Z^{(k)})$ and the input to the next layer $Z^{(k+1)}$.
- **Forward Pass**: $Z^{(k+1)} = f(Z^{(k)})$
- **Backward Pass**: $\delta^{(k+1)} = \frac{\partial E}{\partial Z^{(k+1)}}$
- Applying the **chain rule** for a single layer:

$$\frac{\partial E}{\partial Z^{(k)}} = \frac{\partial E}{\partial Z^{(k+1)}} \frac{\partial Z^{(k+1)}}{\partial Z^{(k)}} = \delta^{(k+1)} \frac{\partial Z^{(k+1)}}{\partial Z^{(k)}} = \delta^{(k+1)} \frac{\partial f(Z^{(k)})}{\partial Z^{(k)}}$$

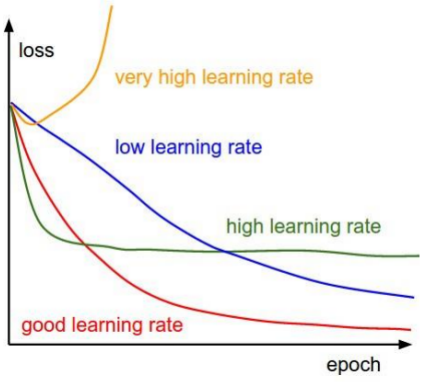
- The **gradient with respect to layer parameters** (if it has any):

$$\frac{\partial E}{\partial W^{(k)}} = \frac{\partial E}{\partial Z^{(k+1)}} \frac{\partial Z^{(k+1)}}{\partial W^{(k)}} = \delta^{(k+1)} \frac{\partial Z^{(k+1)}}{\partial W^{(k)}}$$



Learning Rate

- As we use Gradient Descent, we also have the **learning-rate** *hyper-parameter* which needs to be tuned.



Learning Rate

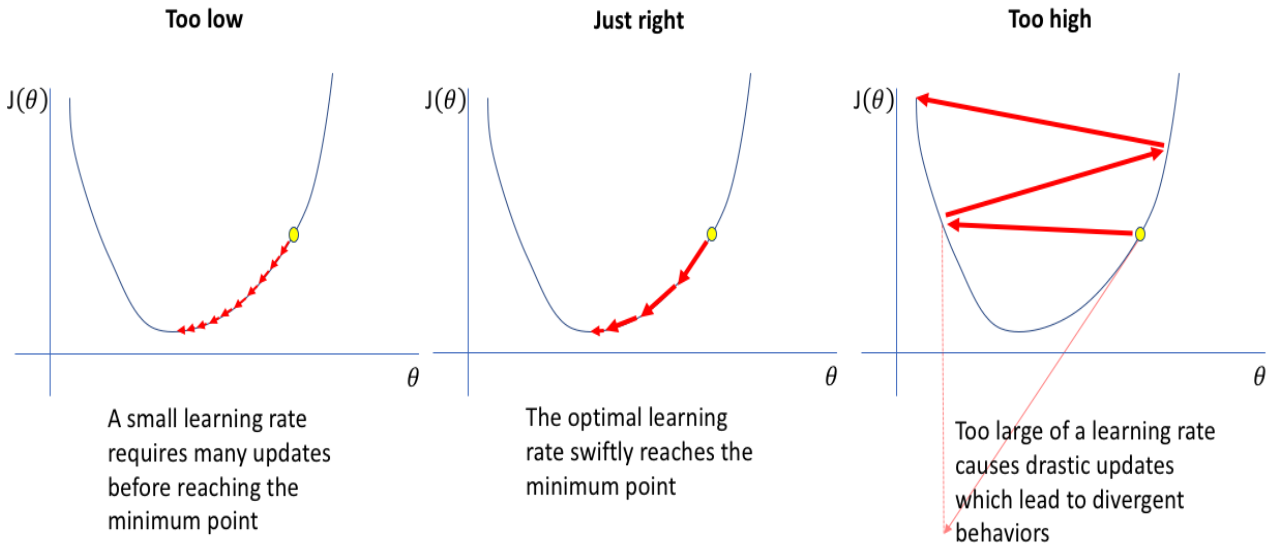


Image Source

---------- CODE TIME ----------

In [6]:
```
# 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()

# 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 [7]:
```
# function to calcualte accuracy of the model
def calculate_accuracy(model, dataloader, device):
    model.eval() # put in evaluation mode
    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 [8]:

```
# training loop
for epoch in range(1, epochs + 1):
    model.train()  # put in training mode
    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)

        # 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,
    epoch_time = time.time() - epoch_time
    log += "Epoch Time: {:.2f} secs".format(epoch_time)
    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 ...')
```

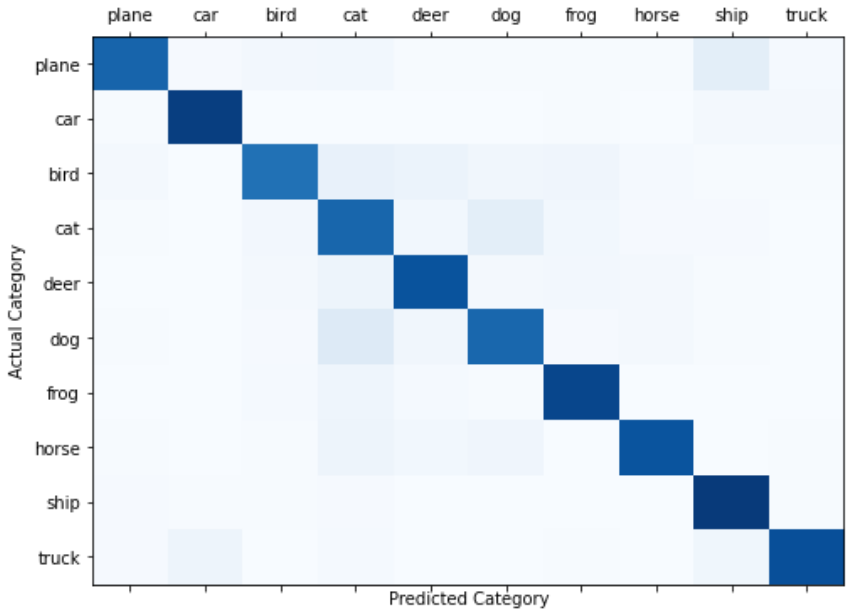
```
Epoch: 1 | Loss: 1.5194 | Training accuracy: 55.432% | Test accuracy: 57.620% | Epoch Time: 44.60 secs
Epoch: 2 | Loss: 1.1147 | Training accuracy: 64.698% | Test accuracy: 65.520% | Epoch Time: 40.83 secs
Epoch: 3 | Loss: 0.9336 | Training accuracy: 69.920% | Test accuracy: 70.540% | Epoch Time: 46.16 secs
Epoch: 4 | Loss: 0.8183 | Training accuracy: 73.812% | Test accuracy: 73.040% | Epoch Time: 42.90 secs
Epoch: 5 | Loss: 0.7264 | Training accuracy: 76.746% | Test accuracy: 76.370% | Epoch Time: 45.65 secs
Epoch: 6 | Loss: 0.6666 | Training accuracy: 77.440% | Test accuracy: 76.000% | Epoch Time: 43.90 secs
Epoch: 7 | Loss: 0.6169 | Training accuracy: 80.026% | Test accuracy: 78.550% | Epoch Time: 43.05 secs
Epoch: 8 | Loss: 0.5738 | Training accuracy: 81.196% | Test accuracy: 79.650% | Epoch Time: 45.32 secs
Epoch: 9 | Loss: 0.5361 | Training accuracy: 82.692% | Test accuracy: 80.280% | Epoch Time: 43.26 secs
Epoch: 10 | Loss: 0.5051 | Training accuracy: 84.484% | Test accuracy: 81.760% | Epoch Time: 48.97 secs
Epoch: 11 | Loss: 0.4742 | Training accuracy: 85.320% | Test accuracy: 82.570% | Epoch Time: 47.13 secs
Epoch: 12 | Loss: 0.4508 | Training accuracy: 87.004% | Test accuracy: 83.550% | Epoch Time: 45.83 secs
Epoch: 13 | Loss: 0.4242 | Training accuracy: 85.536% | Test accuracy: 82.390% | Epoch Time: 45.55 secs
Epoch: 14 | Loss: 0.4044 | Training accuracy: 87.498% | Test accuracy: 83.630% | Epoch Time: 45.26 secs
Epoch: 15 | Loss: 0.3867 | Training accuracy: 88.742% | Test accuracy: 85.210% | Epoch Time: 44.72 secs
Epoch: 16 | Loss: 0.3730 | Training accuracy: 89.514% | Test accuracy: 85.080% | Epoch Time: 48.15 secs
Epoch: 17 | Loss: 0.3508 | Training accuracy: 89.760% | Test accuracy: 85.290% | Epoch Time: 46.50 secs
Epoch: 18 | Loss: 0.3380 | Training accuracy: 89.258% | Test accuracy: 85.120% | Epoch Time: 47.71 secs
```

```
Epoch: 19 | Loss: 0.3241 | Training accuracy: 90.722% | Test accuracy: 85.650% | Epoch Time: 48.09 secs
Epoch: 20 | Loss: 0.3062 | Training accuracy: 90.536% | Test accuracy: 85.470% | Epoch Time: 46.13 secs
==> Saving model ...
==> Finished Training ...
```

In [9]:

```
# 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'])

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: 85.470%
```



In [10]:

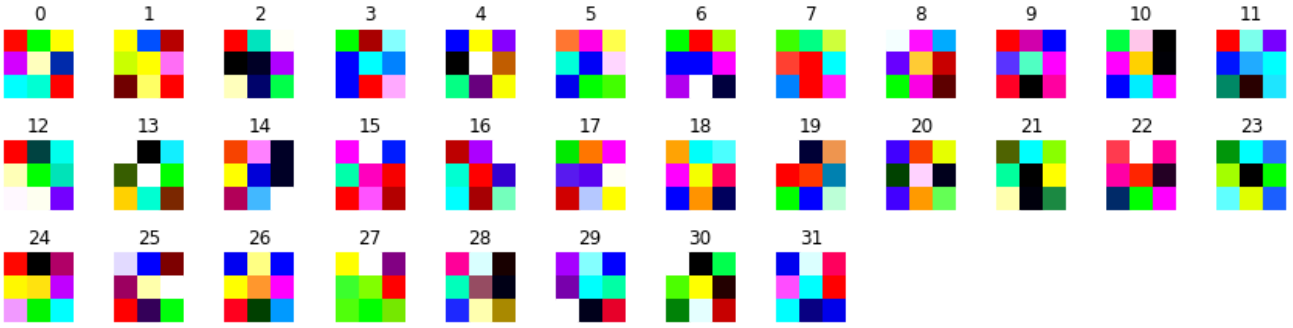
```
# visualize filters - more methods in the appendix to this tutorial
# observe available layers, in our case, the stacked layers are called "conv_layer"
print(model.conv_layer)
# extracting the model features at the particular layer number
layer = model.conv_layer[0]  # to plot other layers, see the appendix tutorial
# 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
```

```
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)
)
```

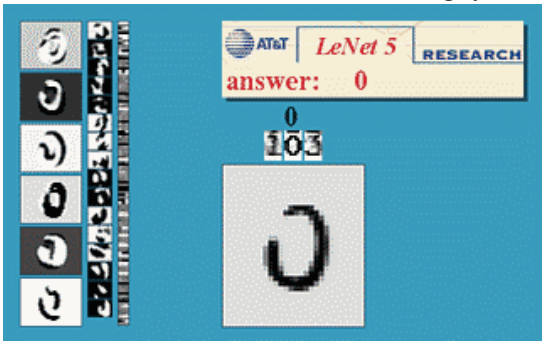
In [11]:

```
#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()
```

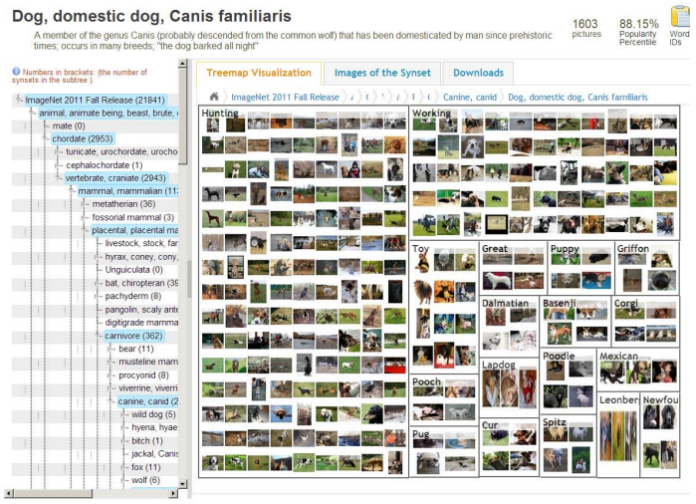


The CNN Story

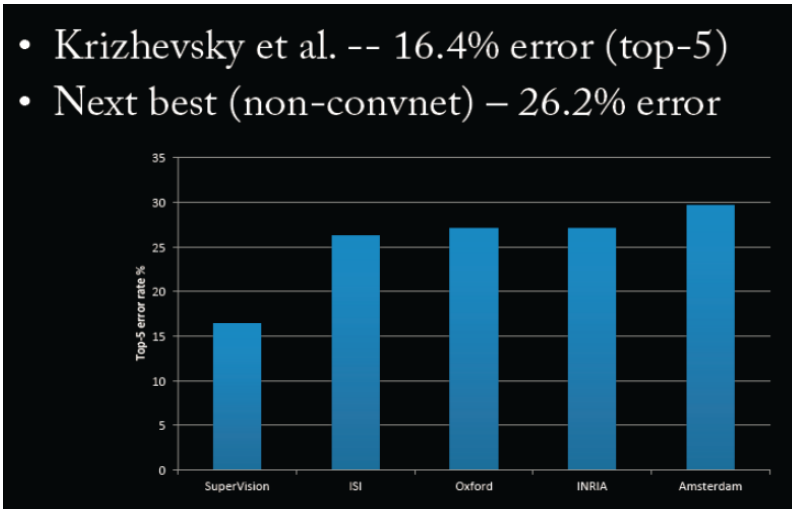
- **1996** - Lenet-5 - core of CNR check reading system, used by US banks.



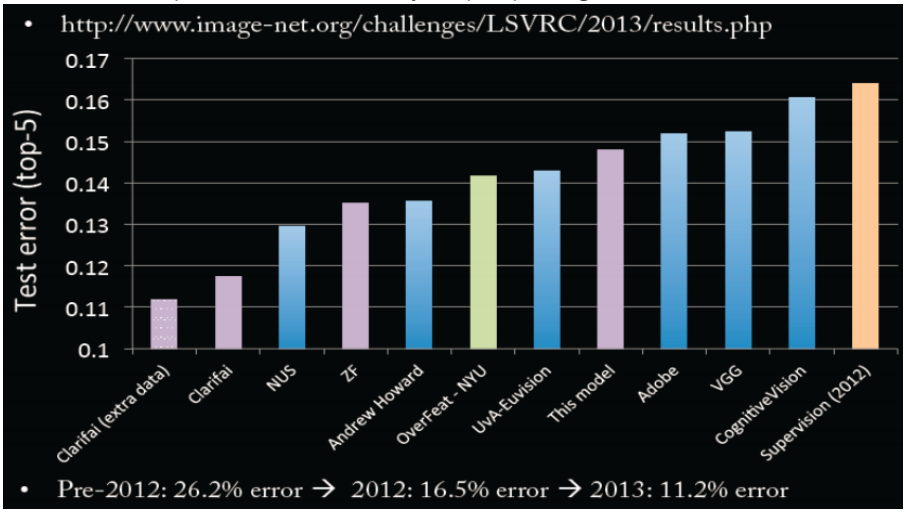
- **2012** - ILSVRC - Imagenet Large Scale Visual Recognition Challenge
  - Imagenet data base: 14M labeled images, 20K categories.



- **2012** - AlexNet wins the challenge by a significant margin!



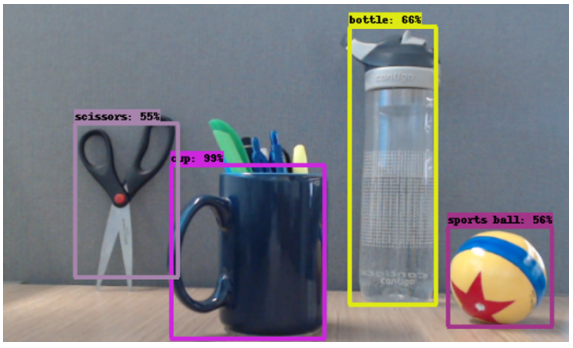
- **2013** - thanks to deep CNNs, the results only keep improving.



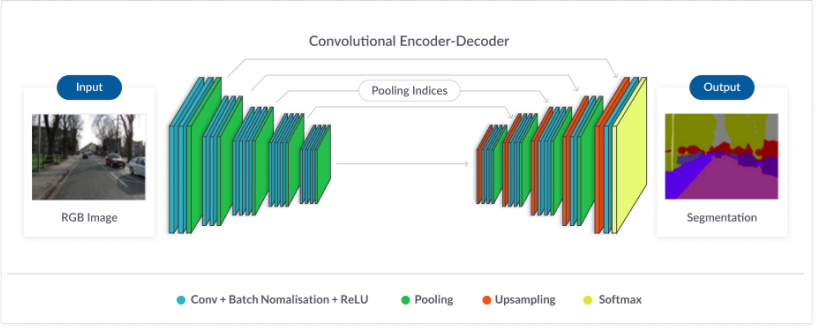
- **Today** - Link

RANK	METHOD	TOP 1 ACCURACY	TOP 5 ACCURACY	NUMBER OF PARAMS	EXTRA TRAINING DATA	PAPER TITLE	YEAR	PAPER	CODE
1	NoisyStudent (EfficientNet-L2)	88.4%	98.7%	480M	✓	Self-training with Noisy Student improves ImageNet classification	2020		
2	BiT-L (ResNet)	87.8%			✓	Large Scale Learning of General Visual Representations for Transfer	2019		
3	NoisyStudent (EfficientNet-L2)	87.4%	98.2%	480M	✓	Self-training with Noisy Student improves ImageNet classification	2019		
4	NoisyStudent (EfficientNet-B7)	86.9%	98.1%	66M	✓	Self-training with Noisy Student improves ImageNet classification	2020		
5	FixResNeXt-101 32x48d	86.4%	98.0%	829M	✓	Fixing the train-test resolution discrepancy	2019		
6	NoisyStudent (EfficientNet-B6)	86.4%	97.9%	43M	✓	Self-training with Noisy Student improves ImageNet classification	2020		

CNNs Applications in Computer Vision

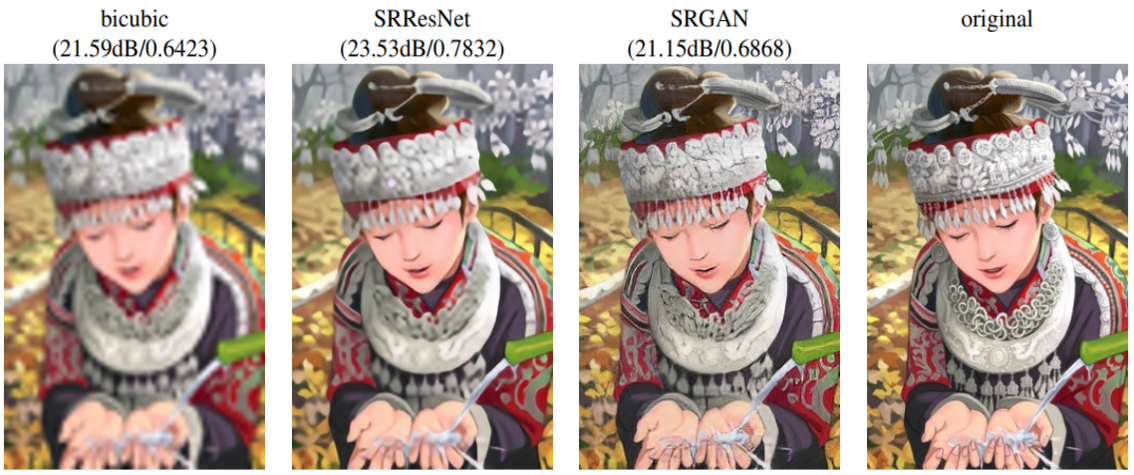
- **Object Detection**
  - 

  - Source

- **Semantic Segmentation**
  - 

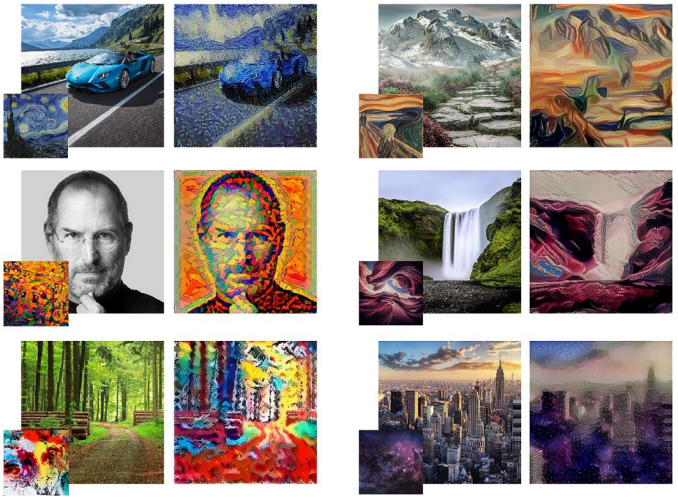
  - Source

- **Super Resolution**



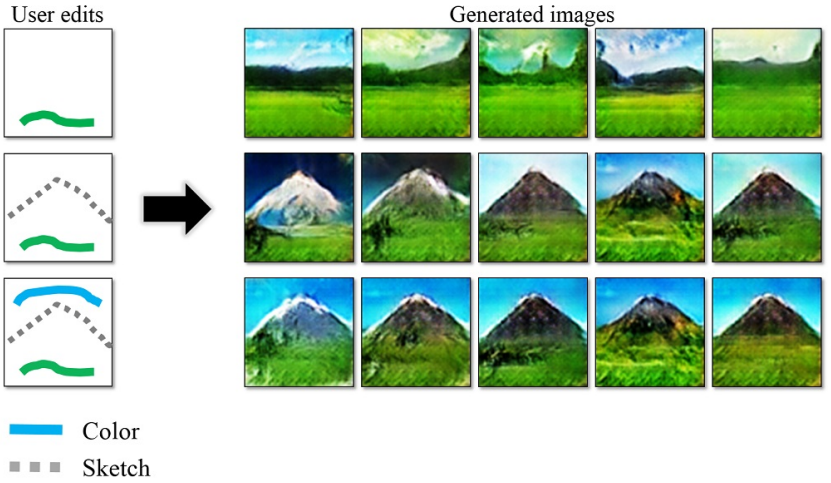
- Source

- **Style Transfer**



- Source

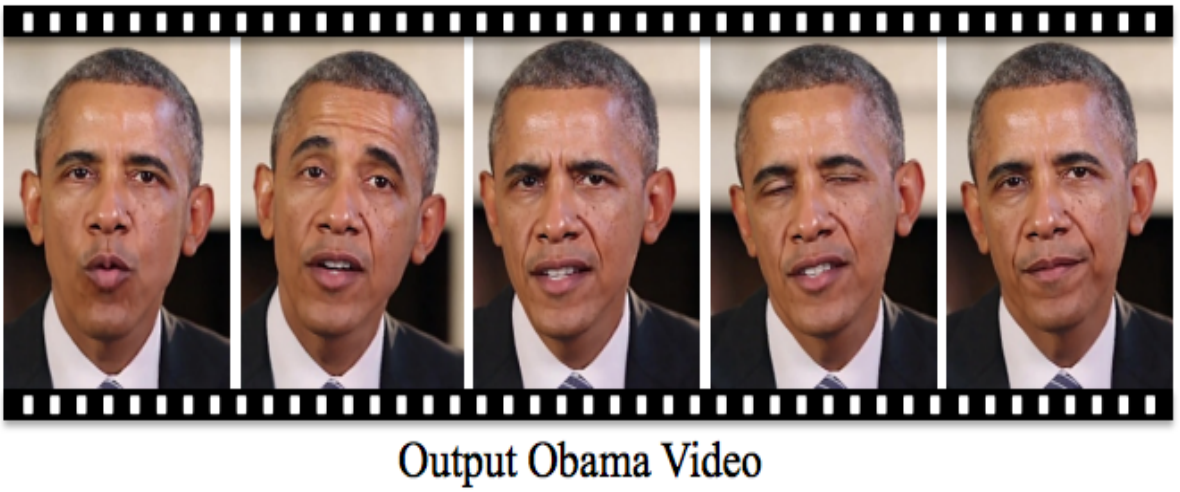
- **Image Editing**



- Source

- **Image Generation**
  - StyleGAN V2 - thispersondoesnotexist.com

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



Output Obama Video

  -
  - Source

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.

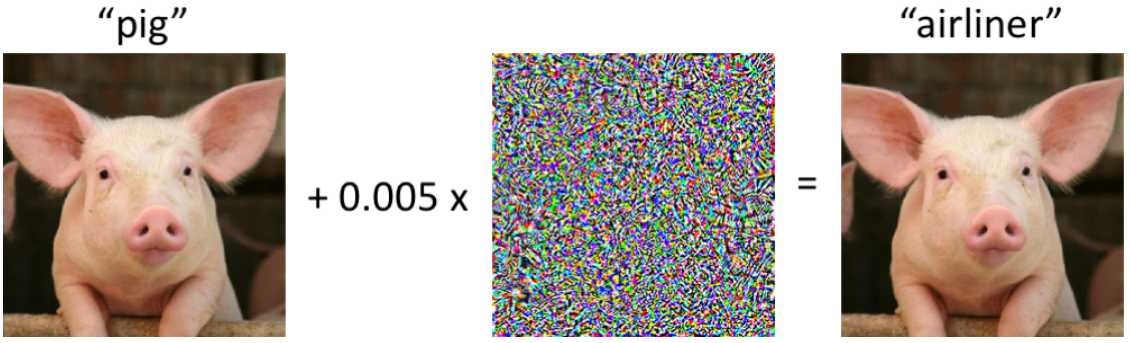


Image Source

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)

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

- EE 046746 Spring 21 - Tal Daniel
- 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
- Icons from Icon8.com - https://icons8.com