Simple Machine Learning Models to Neural Networks

https://bit.ly/2S8CQd2

Git clone https://github.com/LeChangAlex/TechXPlore_workshop.git

Outline

- Image Recognition
- Machine Learning and Applications
  - Supervised Learning vs. Unsupervised Learning
- Supervised learning
  - Classification vs Regression
  - Overfitting vs Underfitting
- Neural Networks
  - Activation Functions?
  - Optimization?
  - Loss Functions?
- Convolutional Neural Networks
- Hands-on Demo with Fashion MNIST

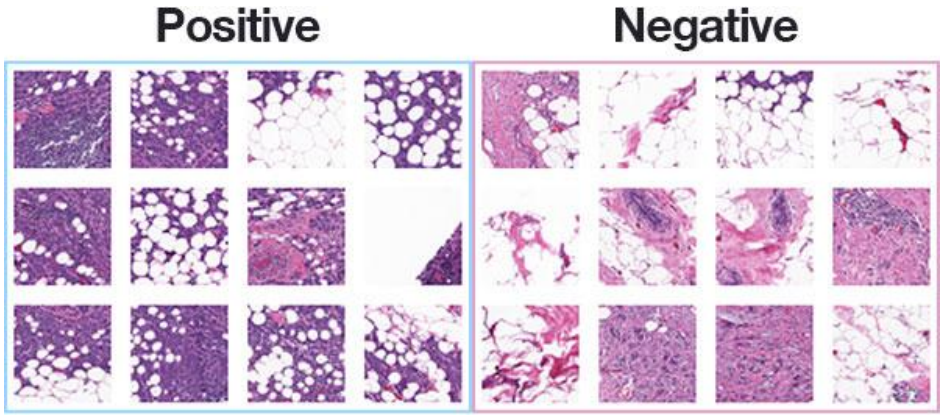
Computer Vision (Image Recognition)

To let the computer see what the humans see, and understand what the humans understand!

- The computer and human's brain sees and analyses *pixels* and *patterns*.
- Thus, image recognition is to let the computer to imitate brain pattern recognition system.
- What tasks does image recognition cover?
  - Image classification
  - Object detection
  - Image segmentation
  - Even generation of synthetic images!

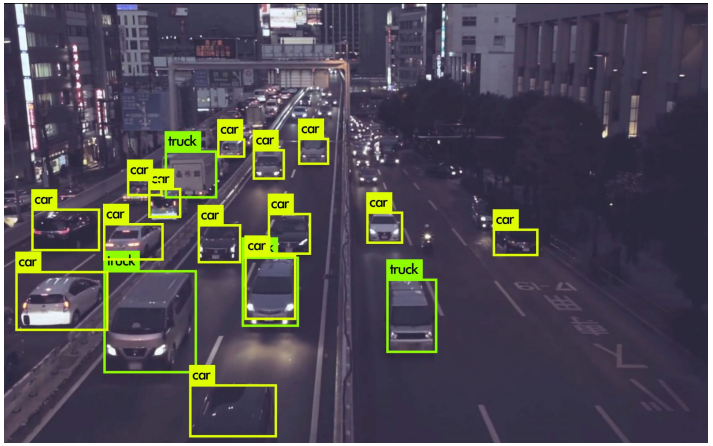
Computer Vision (Image Recognition)

Image classification



pyimagesearch.com/2019/02/18/breast-cancer-classification-with-keras-and-deep-learning/

Object Detection



https://www.move-lab.com/blog/tracking-things-in-object-detection-videos

Classical Computer Science vs Machine Learning

Task of interest: Classification

- Classical CS:
  - Define the underlying structure of the data and an algorithm to map input to prediction.

- ML:
  - Define a model/algorithm with trainable parameters to learn the data's underlying structure and mapping to predictions.

Types of Machine Learning

Supervised Learning

- Learn mapping from **input** space to **label** space

Unsupervised Learning

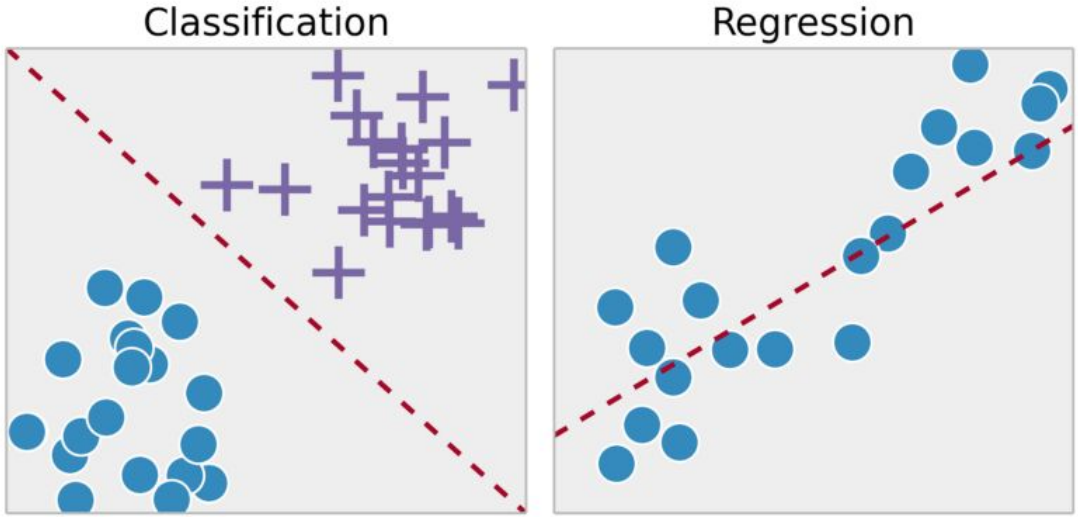
- Learn mapping from **input** space to **lower-dimensional** space

Reinforcement Learning

- Learn mapping from **input** space to **action** space

Supervised Learning

We have labels.



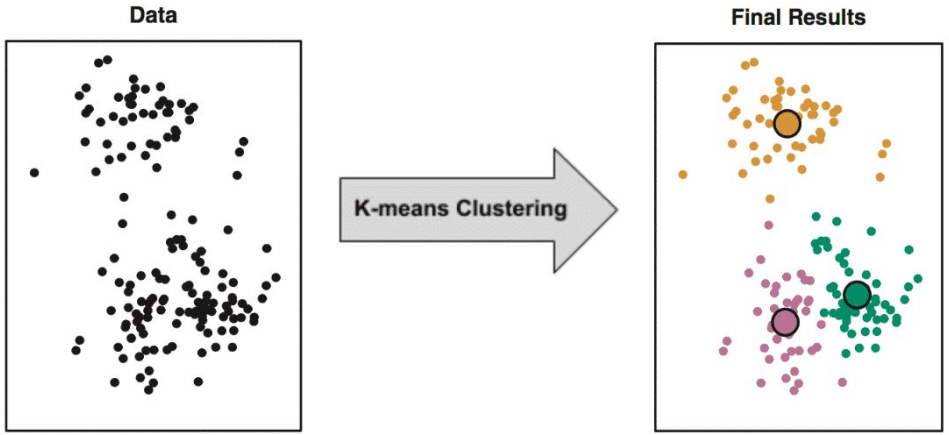
E.g.

- Cancer prediction from X-ray, using past doctor annotations
- Sentiment analysis from Reddit threads/comments, using subreddit as label
- Loan payment prediction, using historical data.

Unsupervised Learning

No labels, clustering, learning structure of the data

- PCA, t-SNE
- Representation Learning
  - Voice Recognition
  - Facial Recognition
- Generative modeling
  - Generative Adversarial Networks
  - Variational AutoEncoders

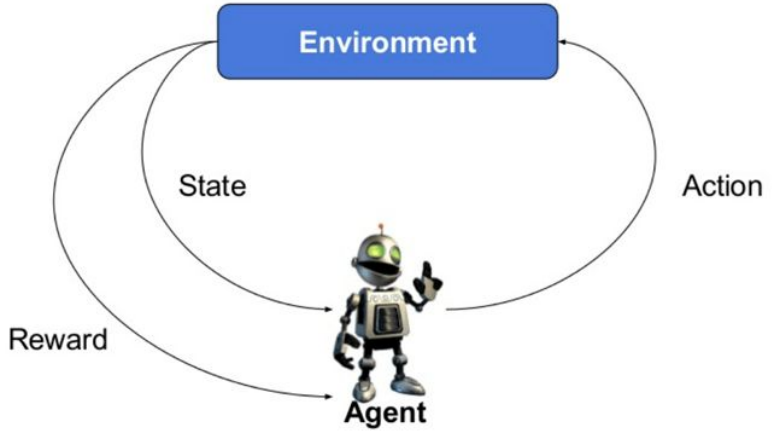


Reinforcement Learning

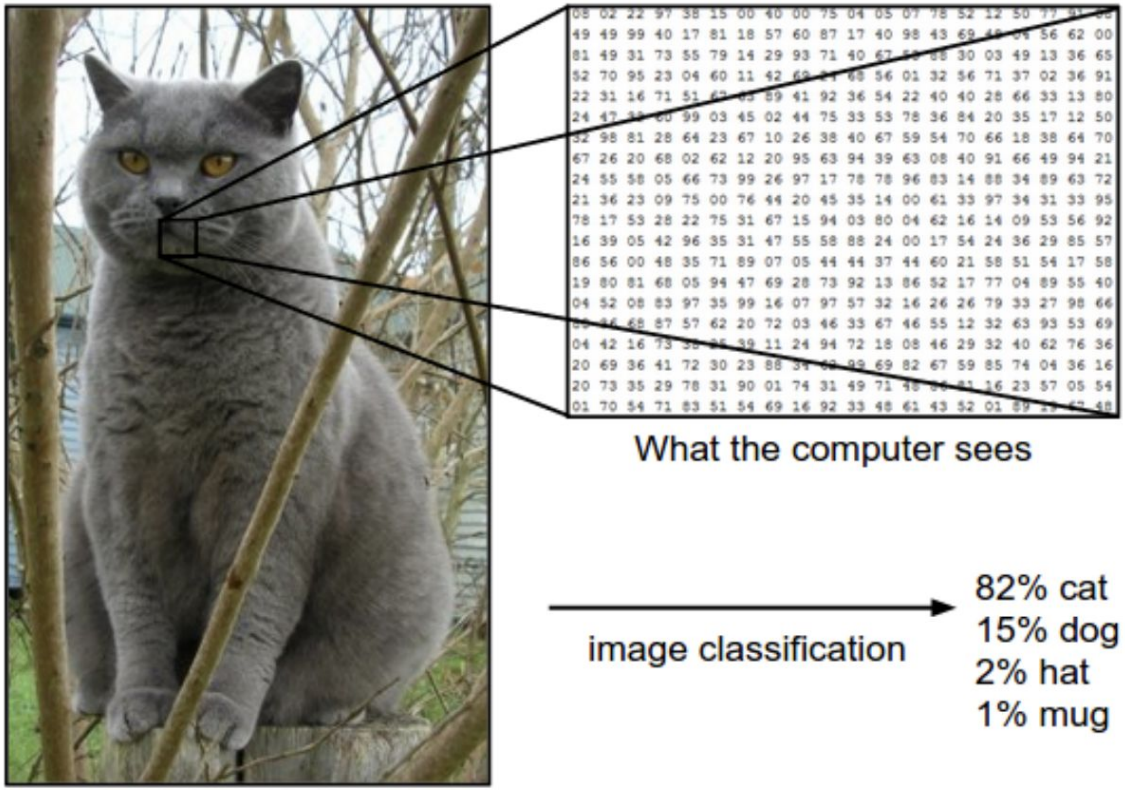
Goal-oriented maximize expected reward

e.g.

- Deepmind’s AlphaGo
- Robotics
- Search Problems in general
- Chat bots
- Deciding when to take medical tests in the Emergency Room.



What an image looks like to the computer:



[Image credit: Andrej Karpathy]

Outline

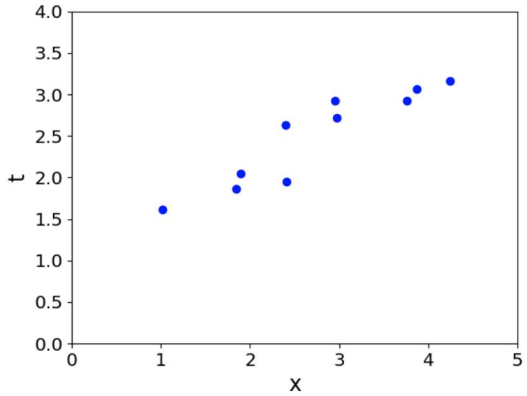
- Image Recognition
- Machine Learning and Applications
  - Supervised Learning vs. Unsupervised Learning
- Supervised learning
  - Classification vs Regression
  - Overfitting vs Underfitting
- Neural Networks
  - Activation Functions?
  - Optimization?
  - Loss Functions?
- Convolutional Neural Networks
- Hands-on Demo with Fashion MNIST

Input Vectors

- Mathematically, our training set consists of a collection of pairs of an input vector $\mathbf{x} \in \mathbb{R}^d$ and its corresponding **target**, or **label**, t
  - **Regression**: t is a real number (e.g. stock price)
  - **Classification**: t is an element of a discrete set $\{1, \ldots, C\}$
  - These days, t is often a highly structured object (e.g. image)
- Denote the training set $\{(\mathbf{x}^{(1)}, t^{(1)}), \ldots, (\mathbf{x}^{(N)}, t^{(N)})\}$
  - Note: these superscripts have nothing to do with exponentiation!

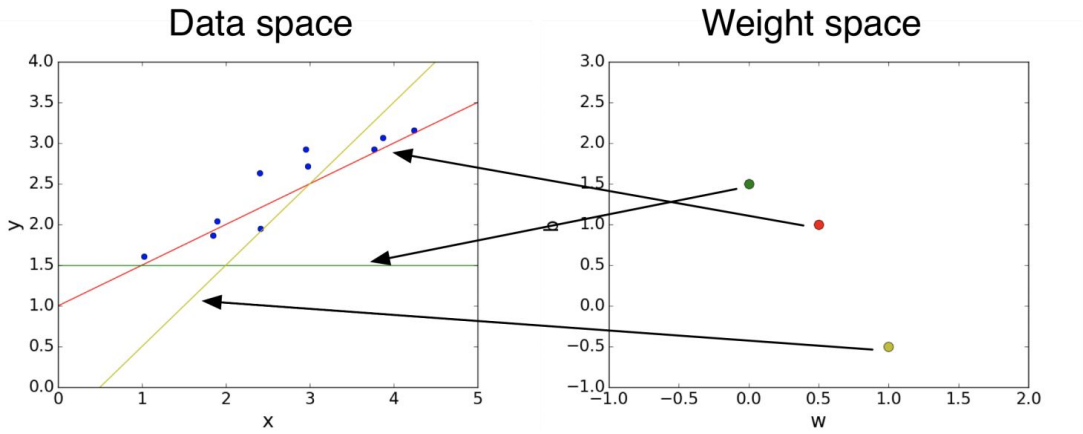
Assume a linear regression problem

Problem Setup



- Want to predict a scalar t as a function of a scalar x
- Given a dataset of pairs $\{(\mathbf{x}^{(i)}, t^{(i)})\}_{i=1}^{N}$
- The $\mathbf{x}^{(i)}$ are called inputs, and the $t^{(i)}$ are called targets.

Problem Setup

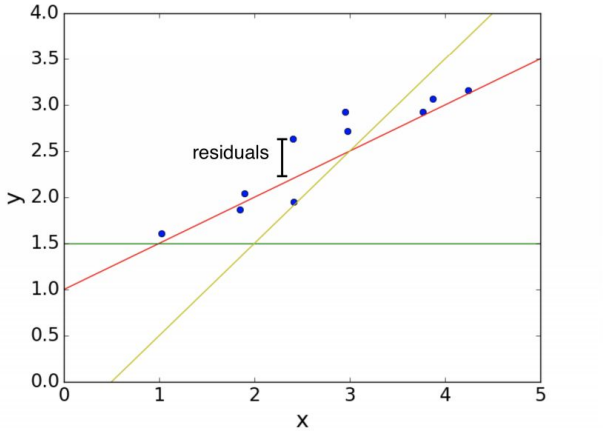


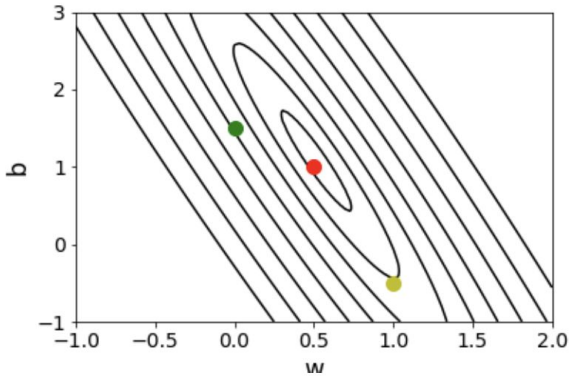
- Model: y is a linear function of x:

$$y = wx + b$$

- y is the prediction
- w is the weight
- b is the bias
- w and b together are the parameters
- Settings of the parameters are called hypotheses

Problem Setup





Problem Setup

- Loss function: squared error (says how bad the fit is)

$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

- $y - t$ is the residual, and we want to make this small in magnitude
- The $\frac{1}{2}$ factor is just to make the calculations convenient.
- Cost function: loss function averaged over all training examples

$$\begin{aligned}\mathcal{J}(w, b) &= \frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)} - t^{(i)}\right)^2 \\ &= \frac{1}{2N}\sum_{i=1}^{N}\left(wx^{(i)} + b - t^{(i)}\right)^2\end{aligned}$$

Vectorization

- We can take this a step further. Organize all the training examples into the design matrix $\mathbf{X}$ with one row per training example, and all the targets into the target vector $\mathbf{t}$.

one feature across all training examples

$$\mathbf{X} = \begin{pmatrix} \mathbf{x}^{(1)\top} \\ \mathbf{x}^{(2)\top} \\ \mathbf{x}^{(3)\top} \end{pmatrix} = \begin{pmatrix} 8 & 0 & 3 & 0 \\ 6 & -1 & 5 & 3 \\ 2 & 5 & -2 & 8 \end{pmatrix}$$

one training example (vector)

- Computing the predictions for the whole dataset:

$$\mathbf{X}\mathbf{w} + b\mathbf{1} = \begin{pmatrix} \mathbf{w}^\top \mathbf{x}^{(1)} + b \\ \vdots \\ \mathbf{w}^\top \mathbf{x}^{(N)} + b \end{pmatrix} = \begin{pmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{pmatrix} = \mathbf{y}$$

Vectorization

- Computing the squared error cost across the whole dataset:

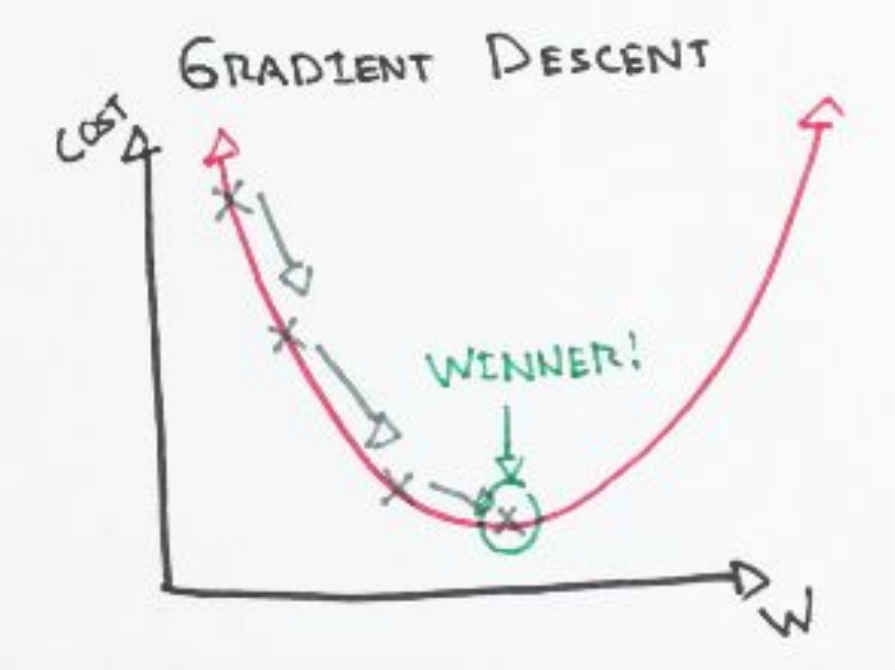
$$\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}$$

$$\mathcal{J} = \frac{1}{2N}\|\mathbf{y} - \mathbf{t}\|^2$$

- In Python:

```
y = np.dot(X, w) + b
cost = np.sum((y - t) ** 2) / (2. * N)
```

Approach to optimize our loss function



https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

Gradient descent

- Observe:
  - if $\partial \mathcal{J} / \partial w_j > 0$, then increasing w_j increases $\mathcal{J}$.
  - if $\partial \mathcal{J} / \partial w_j < 0$, then increasing w_j decreases $\mathcal{J}$.
- The following update decreases the cost function:

$$
\begin{aligned}
w_j &\leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} \\
&= w_j - \frac{\alpha}{N} \sum_{i=1}^{N} (y^{(i)} - t^{(i)})\, x_j^{(i)}
\end{aligned}
$$

- α is a learning rate. The larger it is, the faster $\mathbf{w}$ changes.
  - We'll see later how to tune the learning rate, but values are typically small, e.g. 0.01 or 0.0001

Linear Regression Optimization

For i=1:train_iterations:

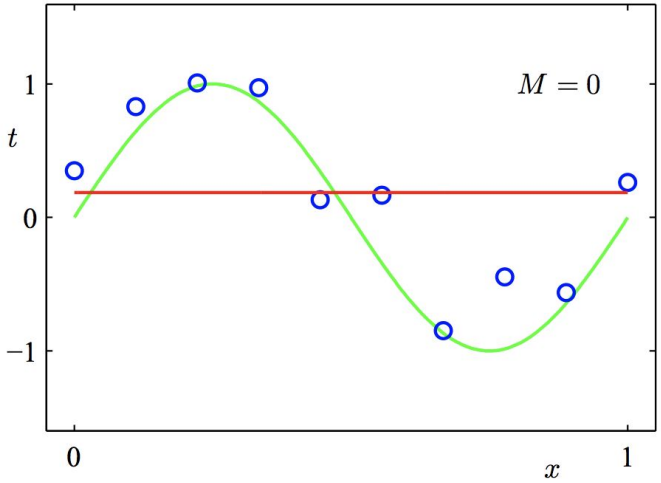
1. Compute current model predictions $\mathbf{y} = \mathbf{X}\mathbf{w} + b\mathbf{1}$
2. Compute the Loss $\mathcal{J} = \frac{1}{2N}\|\mathbf{y} - \mathbf{t}\|^2$
3. Compute Gradients $\frac{\partial \mathcal{J}}{\partial w_j}$
4. Update parameters $w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}$

Outline

- Image Recognition
- Machine Learning and Applications
  - Supervised Learning vs. Unsupervised Learning
- Supervised learning
  - Classification vs Regression
  - Overfitting vs Underfitting
- Neural Networks
  - Activation Functions?
  - Optimization?
  - Loss Functions?
- Convolutional Neural Networks
- Hands-on Demo with Fashion MNIST

Fitting polynomials

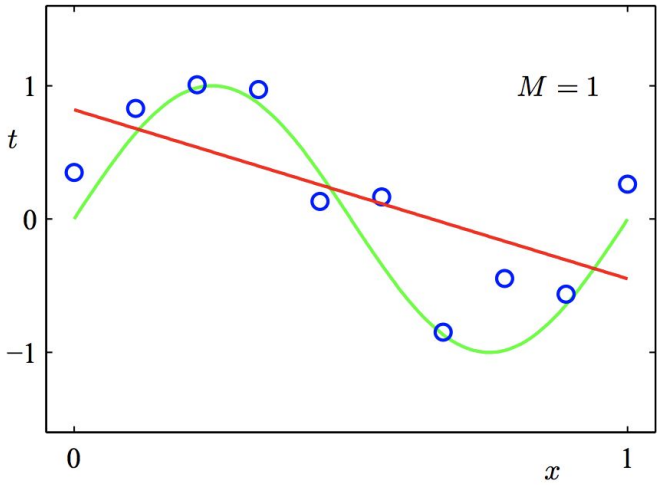
$y = w_0$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

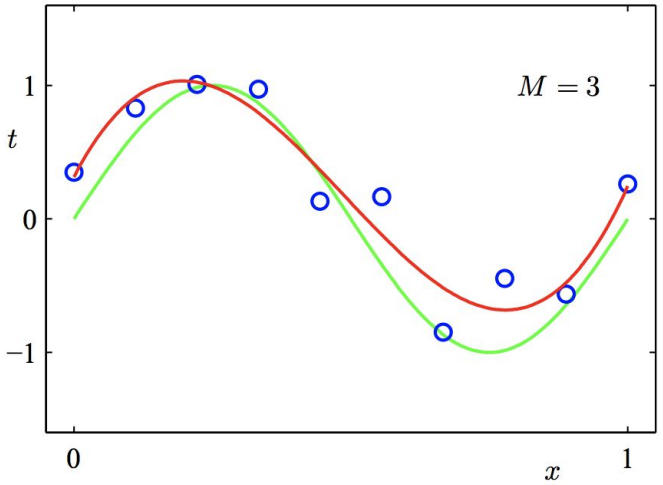
$$y = w_0 + w_1 x$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

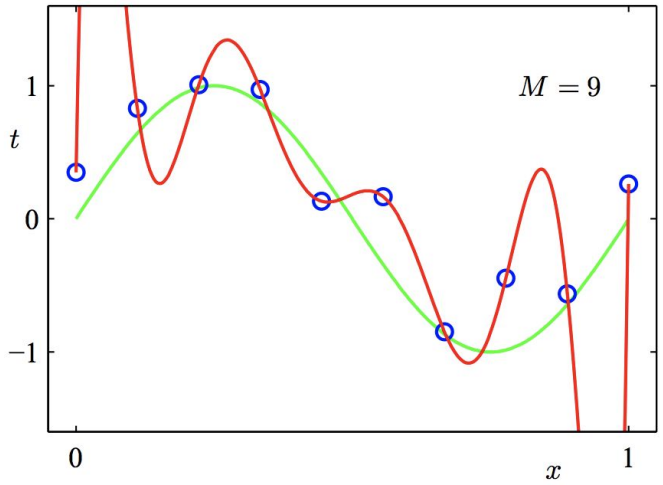
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$



-Pattern Recognition and Machine Learning, Christopher Bishop.

Fitting polynomials

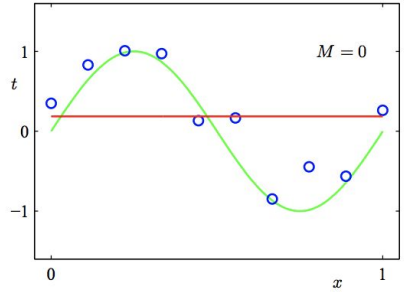
$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \ldots + w_9 x^9$$



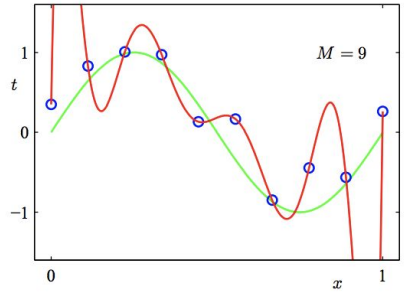
-Pattern Recognition and Machine Learning, Christopher Bishop.

Generalization

Underfitting : model is too simple — does not fit the data.

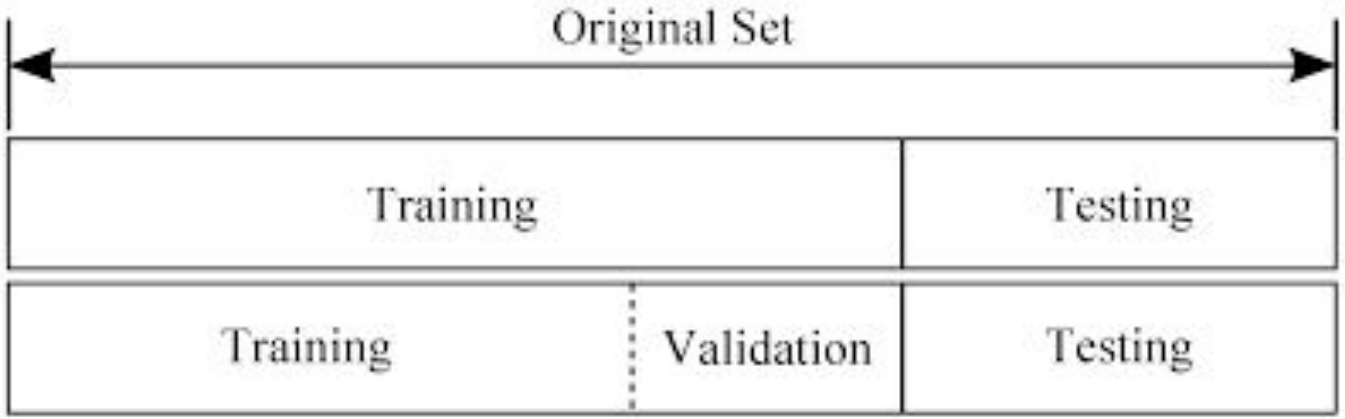


Overfitting : model is too complex — fits perfectly, does not generalize.



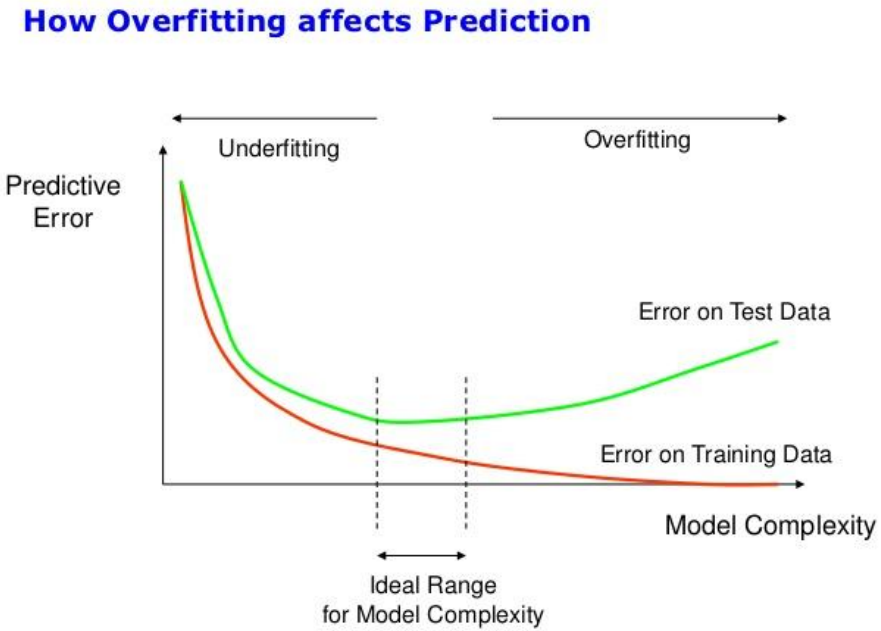
How to inspect and solve overfitting?

Split the training set into two subsets



https://www.programmersought.com/article/32042116778/

How to inspect and solve overfitting?



stats.stackexchange.com/questions/292283/general-question-regarding-over-fitting-vs-complexity-of-models

Outline

- Image Recognition
- Machine Learning and Applications
  - Supervised Learning vs. Unsupervised Learning
- Supervised learning
  - Classification vs Regression
  - Overfitting vs Underfitting
- Neural Networks
  - Activation Functions?
  - Optimization?
  - Loss Functions?
- Convolutional Neural Networks
- Hands-on Demo with Fashion MNIST

Neural Networks

Hypotheses sets with more complex structure, and more parameters.

- **Intuition**: We can model the computational and recognition system of the human's brain.
- **Properties:**
  - Includes more sophisticated hypotheses.
  - Requires much more data, and time to be trained.
  - Vulnerable to overfitting while dealing with simple tasks or small datasets.
  - But, ability to offer great performance in more complicated tasks.

Inspiration: The Brain

- Our brain has $\sim 10^{11}$ neurons, each of which communicates (is connected) to $\sim 10^4$ other neurons

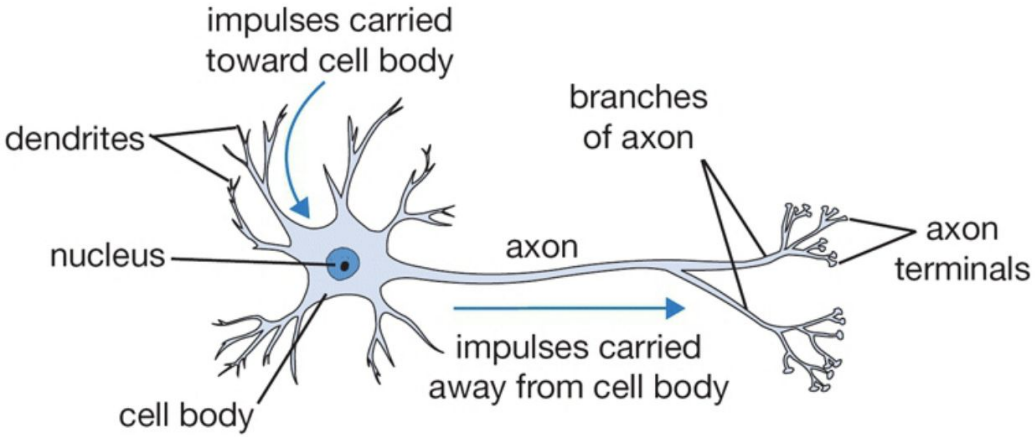
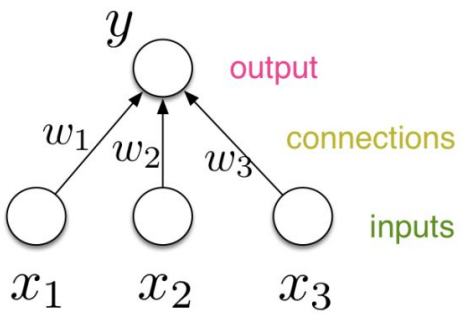


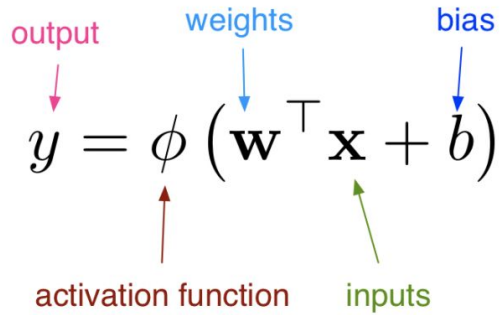
Figure: The basic computational unit of the brain: Neuron

[Pic credit: http://cs231n.github.io/neural-networks-1/]

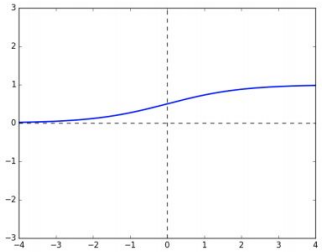
Inspiration: The Brain

- For neural nets, we use a much simpler model neuron, or **unit**:





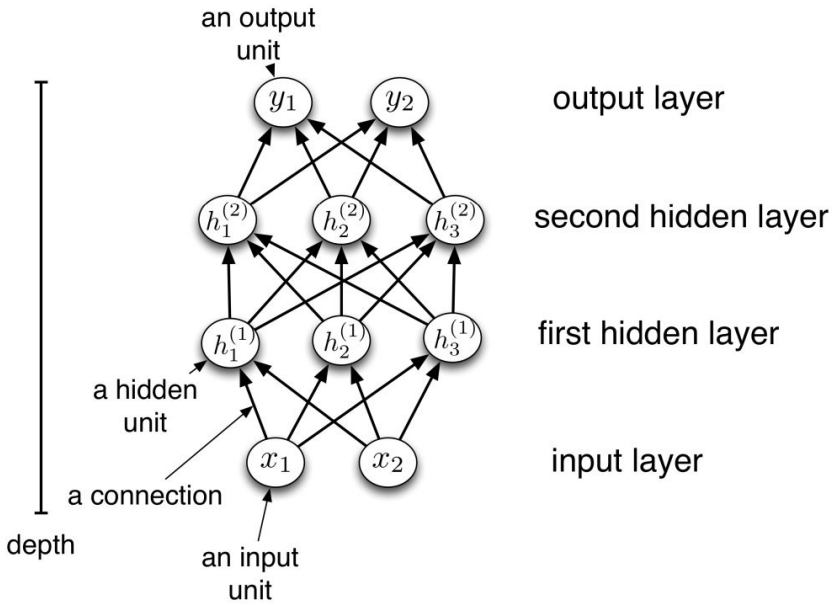
- Compare with logistic regression: $y = \sigma(\mathbf{w}^\top \mathbf{x} + b)$



- By throwing together lots of these incredibly simplistic neuron-like processing units, we can do some powerful computations!

Multilayer Perceptrons

- We can connect lots of units together into a **directed acyclic graph**.
- This gives a **feed-forward neural network**. That's in contrast to **recurrent neural networks**, which can have cycles.
- Typically, units are grouped together into **layers**.



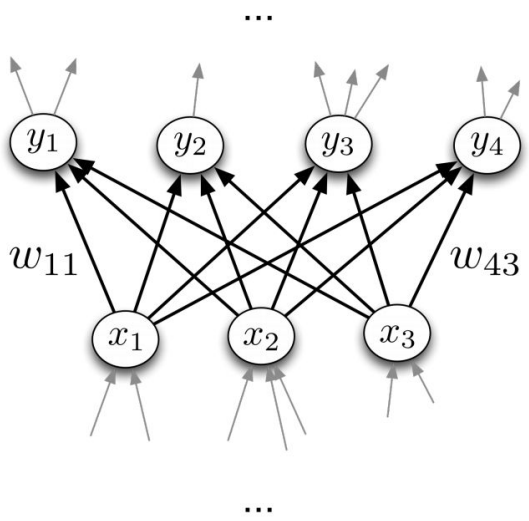
Multilayer Perceptrons

- Each layer connects N input units to M output units.
- In the simplest case, all input units are connected to all output units. We call this a **fully connected layer**. We'll consider other layer types later.
- Note: the inputs and outputs for a layer are distinct from the inputs and outputs to the network.

- Recall from softmax regression: this means we need an $M \times N$ weight matrix.
- The output units are a function of the input units:

$$\mathbf{y} = f(\mathbf{x}) = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$$

- A multilayer network consisting of fully connected layers is called a **multilayer perceptron**. Despite the name, it has nothing to do with perceptrons!



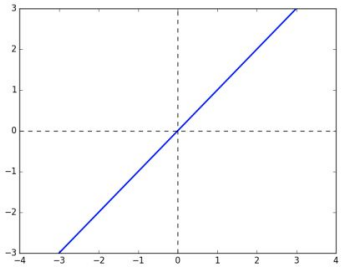
Expressive Power

- We've seen that there are some functions that linear classifiers can't represent. Are deep networks any better?
- Any sequence of *linear* layers can be equivalently represented with a single linear layer.

$$\mathbf{y} = \underbrace{\mathbf{W}^{(3)}\mathbf{W}^{(2)}\mathbf{W}^{(1)}}_{\triangleq \mathbf{W}'}\mathbf{x}$$

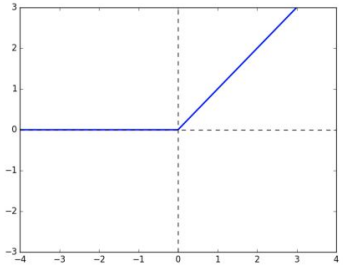
  - Deep linear networks are no more expressive than linear regression.
  - Linear layers do have their uses

Some activation functions:



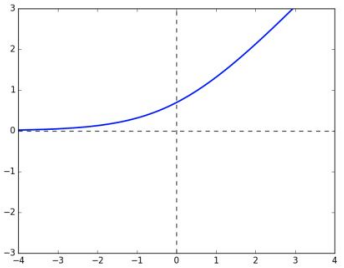
Linear

$y = z$



Rectified Linear Unit (ReLU)

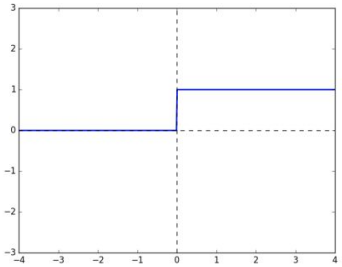
$y = \max(0, z)$



Soft ReLU

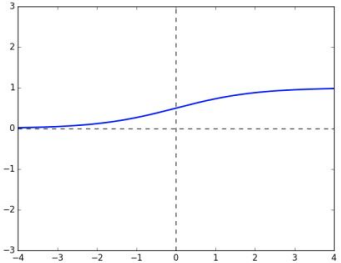
$y = \log 1 + e^z$

Some activation functions:



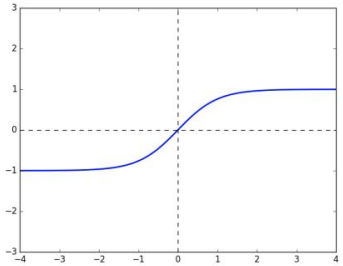
Hard Threshold

$$y = \begin{cases} 1 & \text{if } z > 0 \\ 0 & \text{if } z \leq 0 \end{cases}$$



Logistic

$$y = \frac{1}{1 + e^{-z}}$$

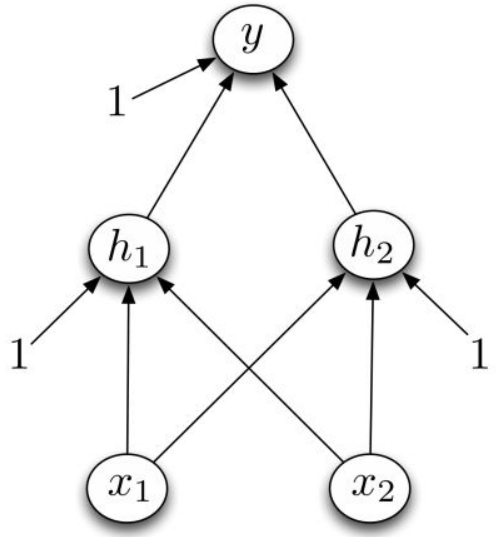


Hyperbolic Tangent (tanh)

$$y = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

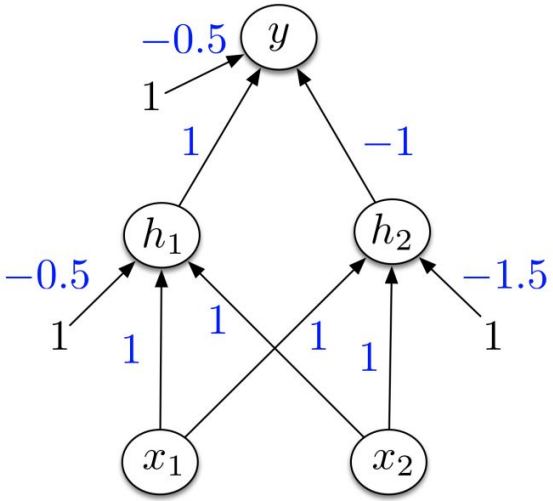
Designing a network to compute XOR:

Assume hard threshold activation function



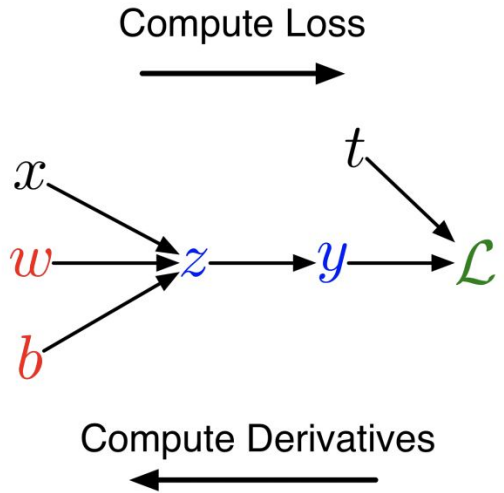
Multilayer Perceptrons

- h_1 computes x_1 OR x_2
- h_2 computes x_1 AND x_2
- y computes h_1 AND NOT x_2



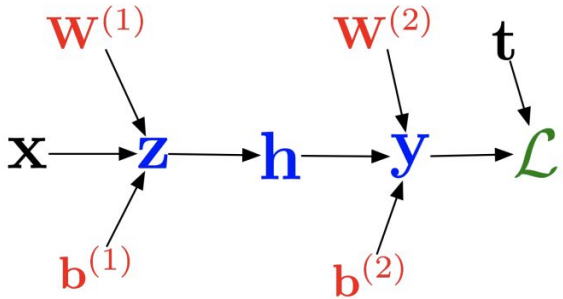
Univariate Chain Rule

- We can diagram out the computations using a **computation graph**.
- The nodes represent all the inputs and computed quantities, and the edges represent which nodes are computed directly as a function of which other nodes.



Backpropagation

In vectorized form:



Forward pass:

$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\mathbf{h} = \sigma(\mathbf{z})$$
$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$
$$\mathcal{L} = \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2$$

Backward pass:

$$\overline{\mathcal{L}} = 1$$
$$\overline{\mathbf{y}} = \overline{\mathcal{L}}\,(\mathbf{y} - \mathbf{t})$$
$$\overline{\mathbf{W}^{(2)}} = \overline{\mathbf{y}}\mathbf{h}^\top$$
$$\overline{\mathbf{b}^{(2)}} = \overline{\mathbf{y}}$$
$$\overline{\mathbf{h}} = \mathbf{W}^{(2)\top}\overline{\mathbf{y}}$$
$$\overline{\mathbf{z}} = \overline{\mathbf{h}} \circ \sigma'(\mathbf{z})$$
$$\overline{\mathbf{W}^{(1)}} = \overline{\mathbf{z}}\mathbf{x}^\top$$
$$\overline{\mathbf{b}^{(1)}} = \overline{\mathbf{z}}$$

Optimization

For i=1:num_epochs:

1. Compute current model predictions***
2. Compute the Loss $\mathcal{L} = \frac{1}{2}\|\mathbf{t} - \mathbf{y}\|^2$
3. Compute Gradients $\frac{\partial \mathcal{J}}{\partial w_j}$
4. Update parameters $w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j}$

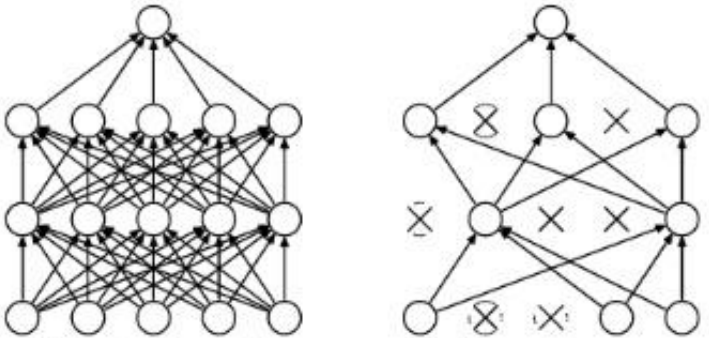
$$\mathbf{z} = \mathbf{W}^{(1)}\mathbf{x} + \mathbf{b}^{(1)}$$
$$\mathbf{h} = \sigma(\mathbf{z})$$
$$\mathbf{y} = \mathbf{W}^{(2)}\mathbf{h} + \mathbf{b}^{(2)}$$

***Predictions are done on a subset of the data (mini-batch) each iteration to reduce runtime

Dropout

Another approach to prevent overfitting in training phase.

- Similar to bagging (approximation of bagging)
- Act like **regularizer** (reduce overfit)
- Instead of using all neurons, “dropout” some neurons randomly (usually 0.5 probability)



(a) Standard Neural Net (b) After applying dropout.

Outline

- Image Recognition
- Machine Learning and Applications
  - Supervised Learning vs. Unsupervised Learning
- Supervised learning
  - Classification vs Regression
  - Overfitting vs Underfitting
- Neural Networks
  - Activation Functions?
  - Optimization?
  - Loss Functions?
- Convolutional Neural Networks
- Hands-on Demo with Fashion MNIST

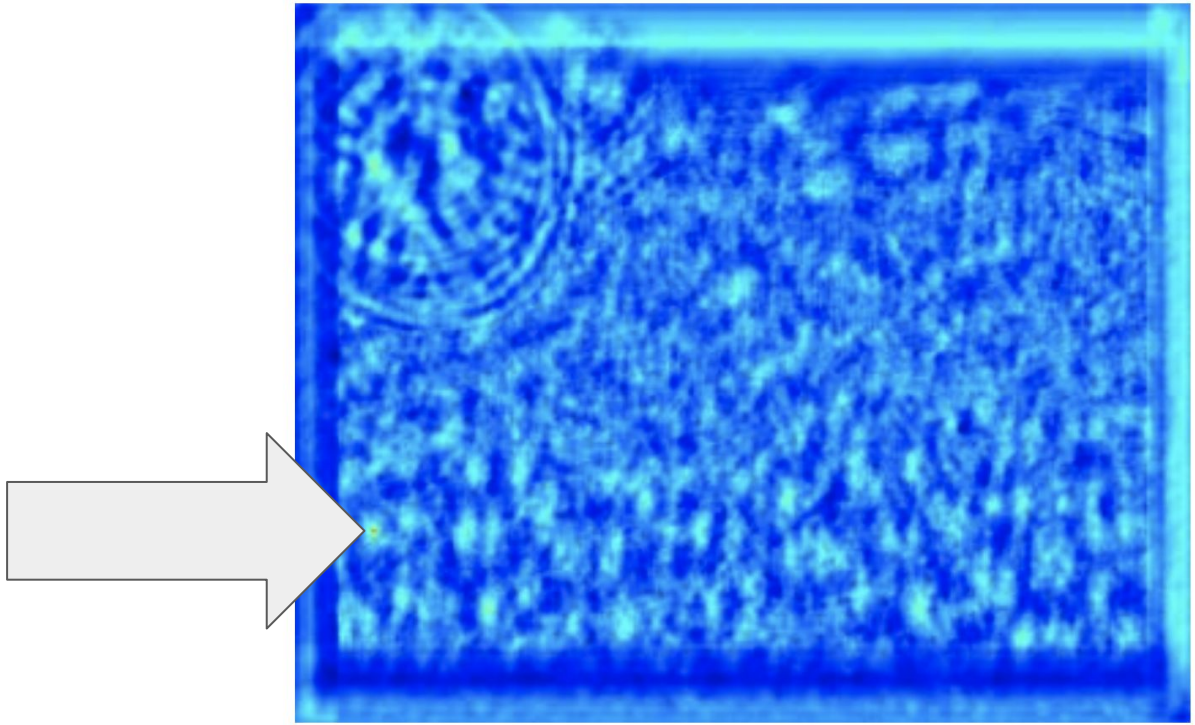
Let’s have a flashback to the slides in Lecture 2: “Linear Filters”.

Back to Waldo

image I

filter F

Back to Waldo



- Result of normalized cross-correlation

Back to Waldo

image I

filter F

Examples: Partial Derivatives of an Image

Figure: Using correlation filters

[Source: K. Grauman]

2-D Convolution

What does this convolution kernel do?

$*$

1	0	-1
2	0	-2
1	0	-1

2-D Convolution

The thing we convolve by is called a kernel, or filter.

What does this convolution kernel do?

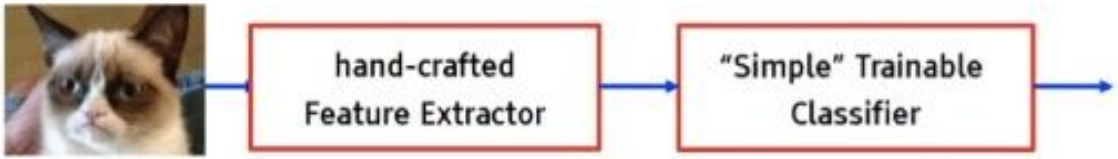
$*$

0	1	0
1	4	1
0	1	0

Idea: Training Linear Filters.

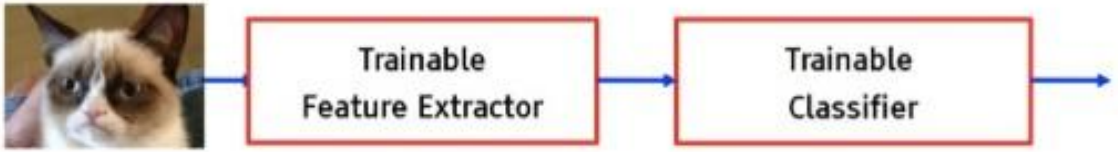
Traditional Approach

The traditional approach uses fixed feature extractors.



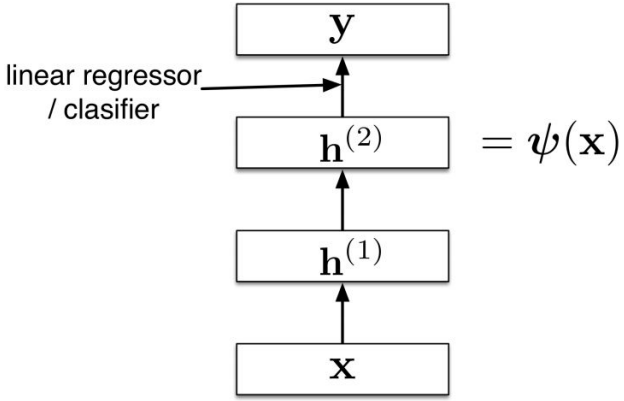
Deep Learning Approach

Deep Learning approach uses trainable feature extractors.

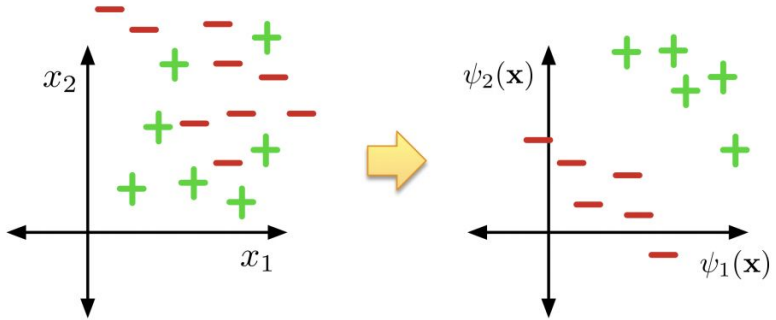


Feature Learning

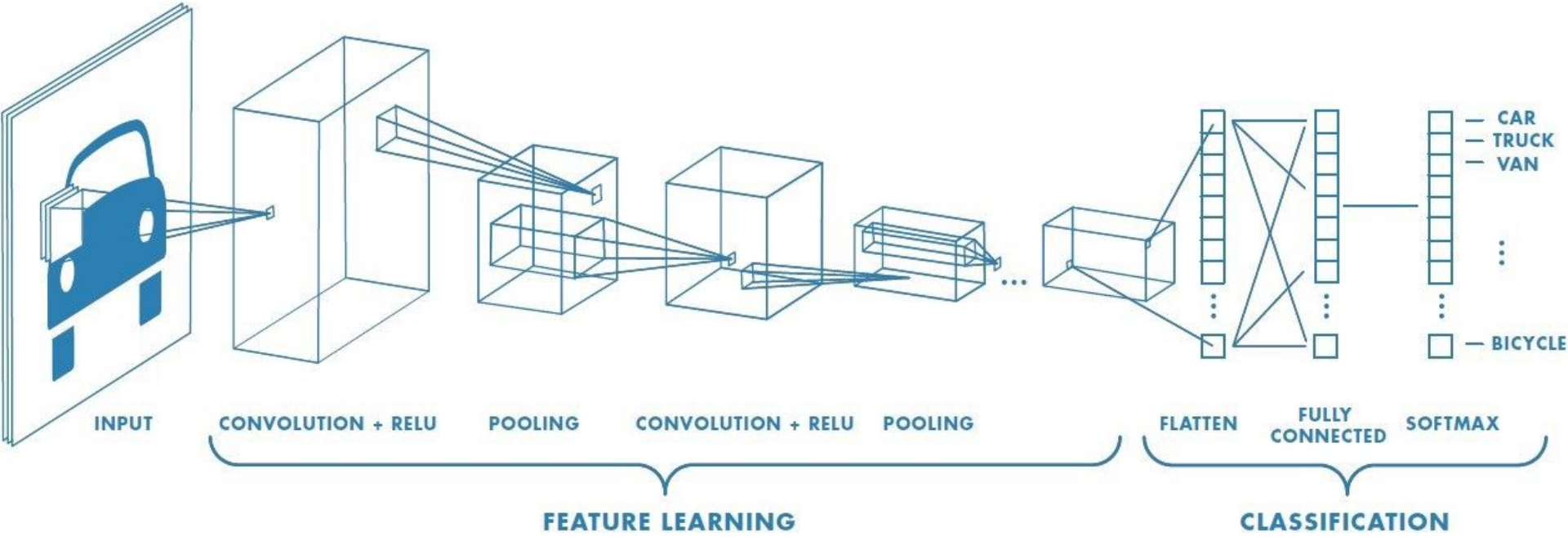
- Neural nets can be viewed as a way of learning features:

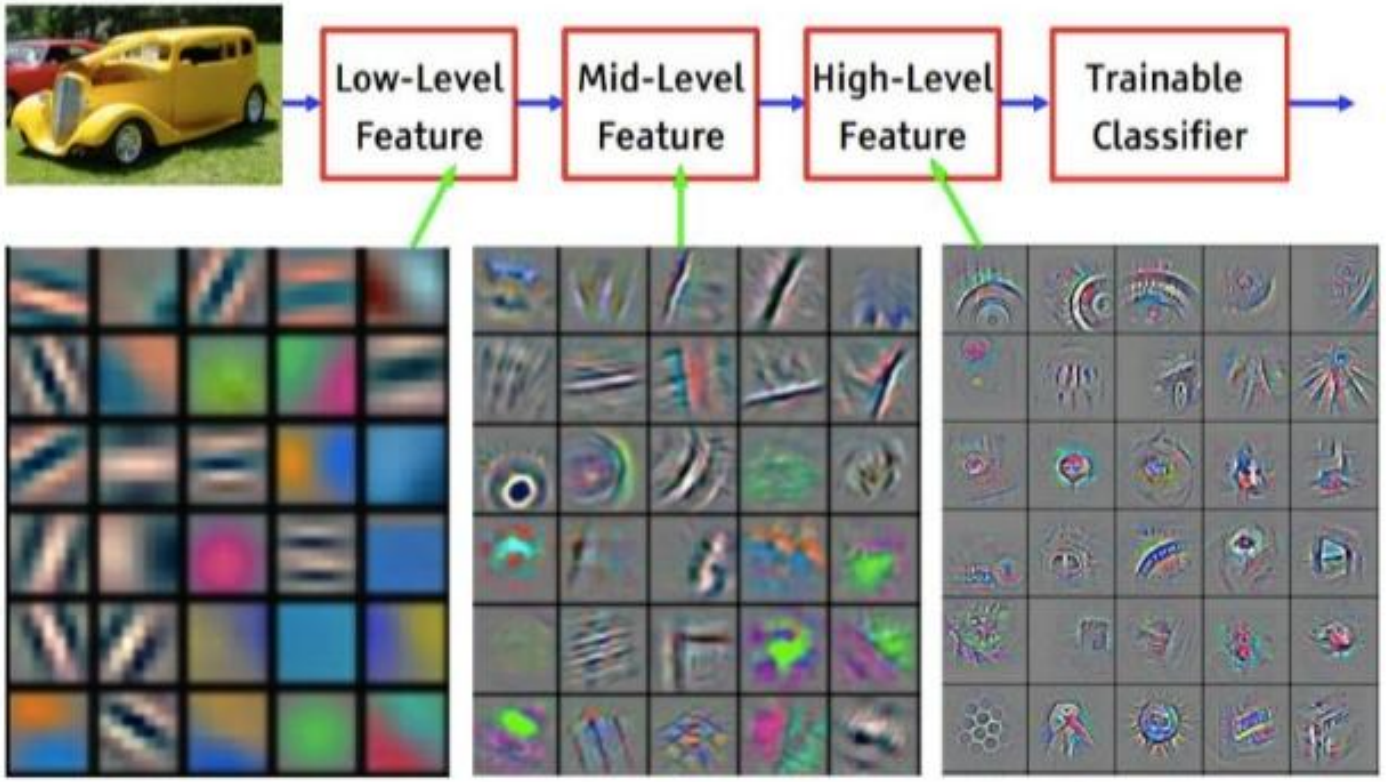


- The goal:

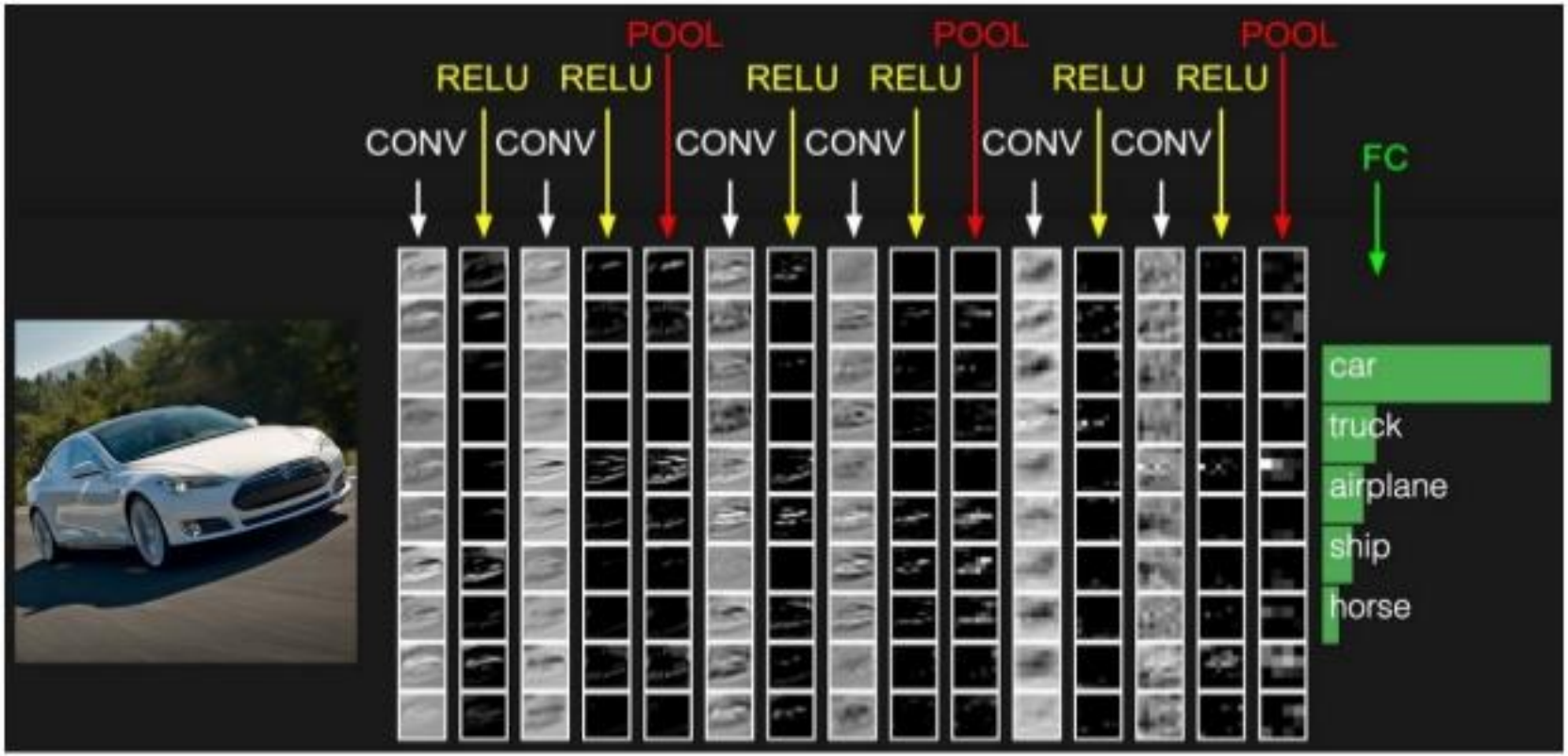


Convolutional Neural Network (CNN)



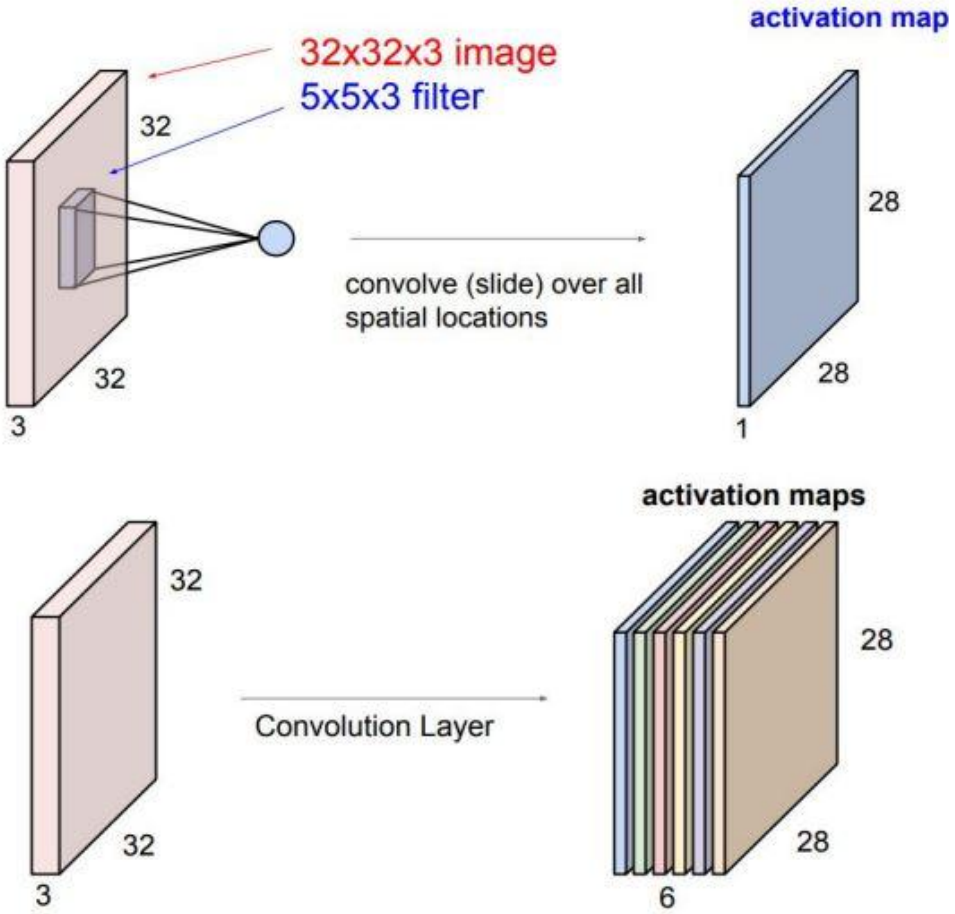


Feature visualization of convolutional net trained on ImageNet from [Zeiler & Fergus 2013]


POOL
POOL
POOL
RELU
RELU
RELU
RELU
RELU
RELU
CONV
CONV
CONV
CONV
CONV
CONV
FC
car
truck
airplane
ship
horse

Convolutional Layer

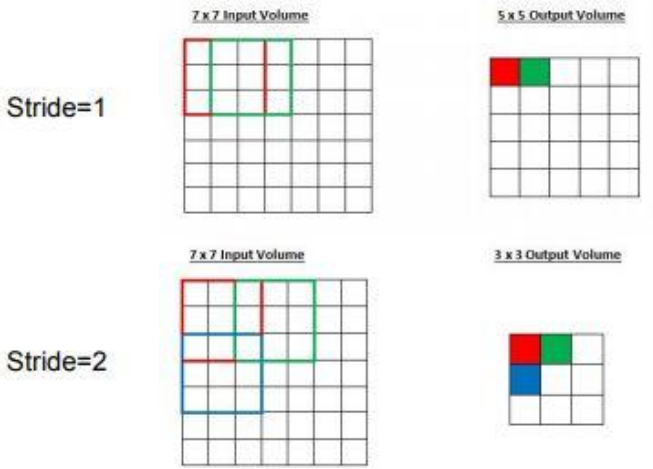
- We can view this as a convolution operation between a filter and an input image, but the filters are trainable.
- Each filter produce a 2d output that we call it activation map
- We generally use more than one filter in each layer
- In the example below, 6 filter together create a convolutional layer
- We stack activation maps depth-wise and feed it to the next layer
- Each neural network that contains convolutional layer is called convolutional neural network

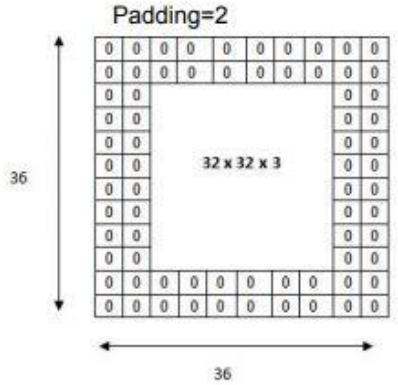


Pic Credit: http://cs231n.github.io/

Convolutional Layers Terminology

- To define a conv layer you need to specify few parameters:
  - Filter Size (The spatial location that each neuron looks at. Normally 3*3 or 5*5)
  - Number of Filters (The number of channels in the output activation maps)
  - Stride (Number of pixels that the filter moves in each iteration)
  - Padding (You might want to keep the spatial size of the image the same)

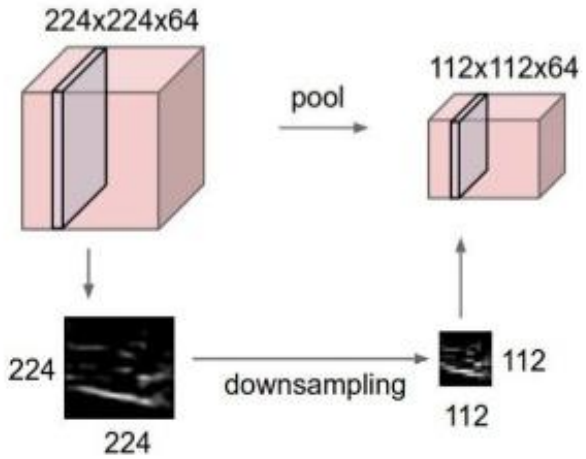




https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/

Pooling Layer

- **Downsample** image to reduce parameter
- Usually use **max pooling** (take maximum value in region)



These layers reduce the unneccessary complexity of the network; but they also reduce the dependency of the extracted features to the spatial location.

Batch Normalization

- **Make output to be gaussian distribution**, but normalization cost a lot
  - Calc mean, variance in each dimension (**assume each dims are uncorrelated**)
  - Calc mean, variance **in mini-batch** (not entire set)
- Normalize constrain non-linearlity and constrain network by assume each dims are uncorrelated
  - Linear transform output (factors are parameter)

Batch Normalization

- When test, calc mean, variance using entire set (use **moving average**)

- BN act like **regularizer** (don't need Dropout)

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

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

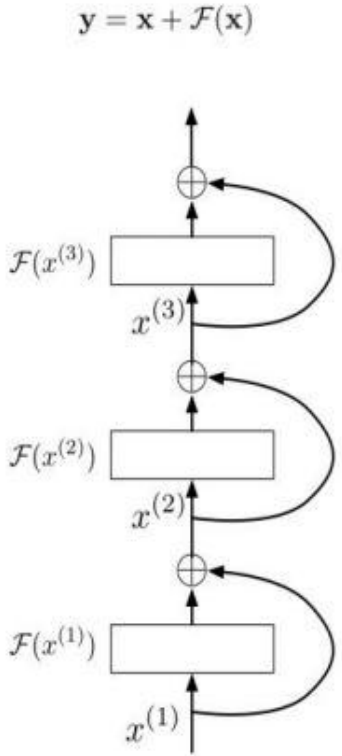
$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

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

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

Residual Connection

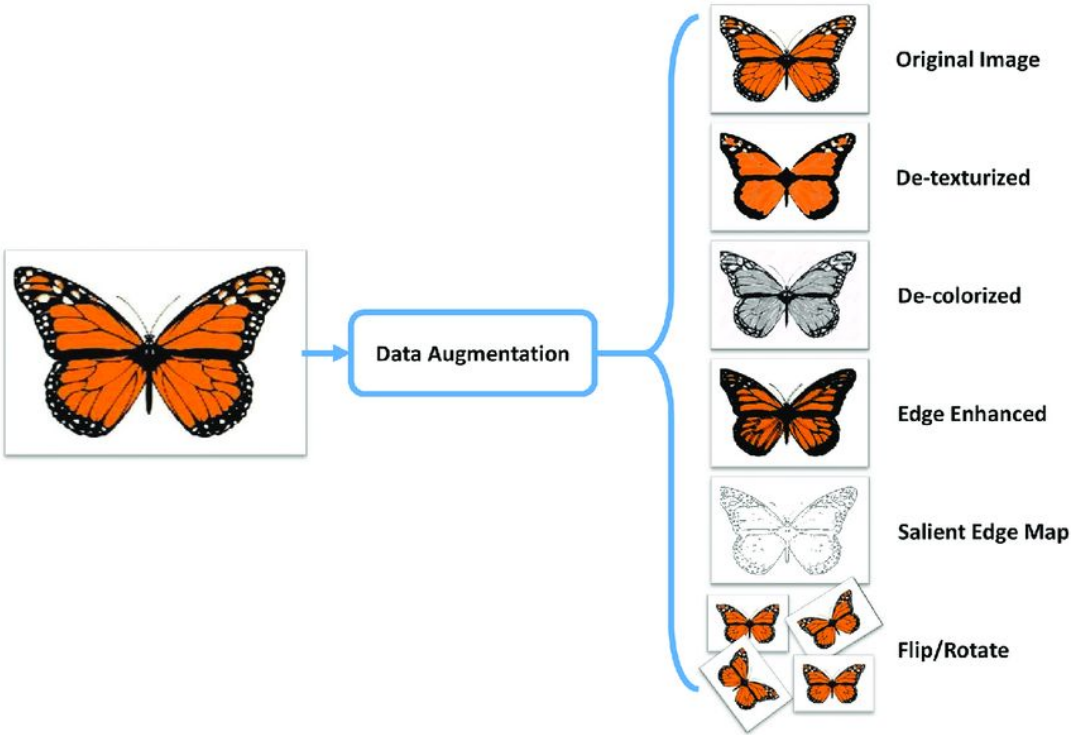
- We expect increasing the number of layers to improve the results (or at least gives the results as good as shallower networks)
- But increasing the number of layers made the optimization much harder
- Residual Neural Networks add a simple connection between layers and showed promising results with huge number of layers



Credit to: Sajad Norouzi

Data Augmentation

A technique to enlarge your training set.



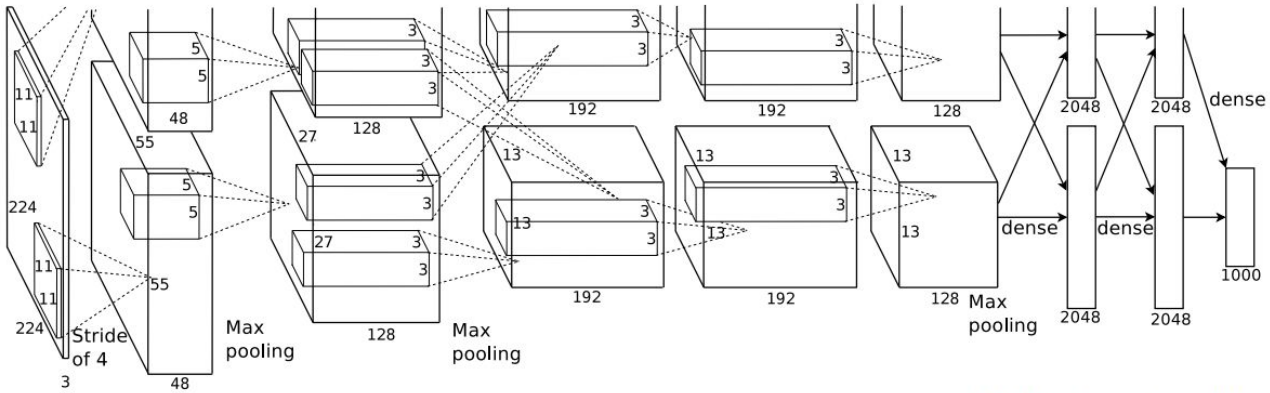
https://medium.com/secure-and-private-ai-writing-challenge/data-augmentation-increases-accuracy-of-your-model-but-how-aa1913468722

ImageNet

- Imagenet, biggest dataset for object classification: `http://image-net.org/`
- 1000 classes, 1.2M training images, 150K for test

AlexNet

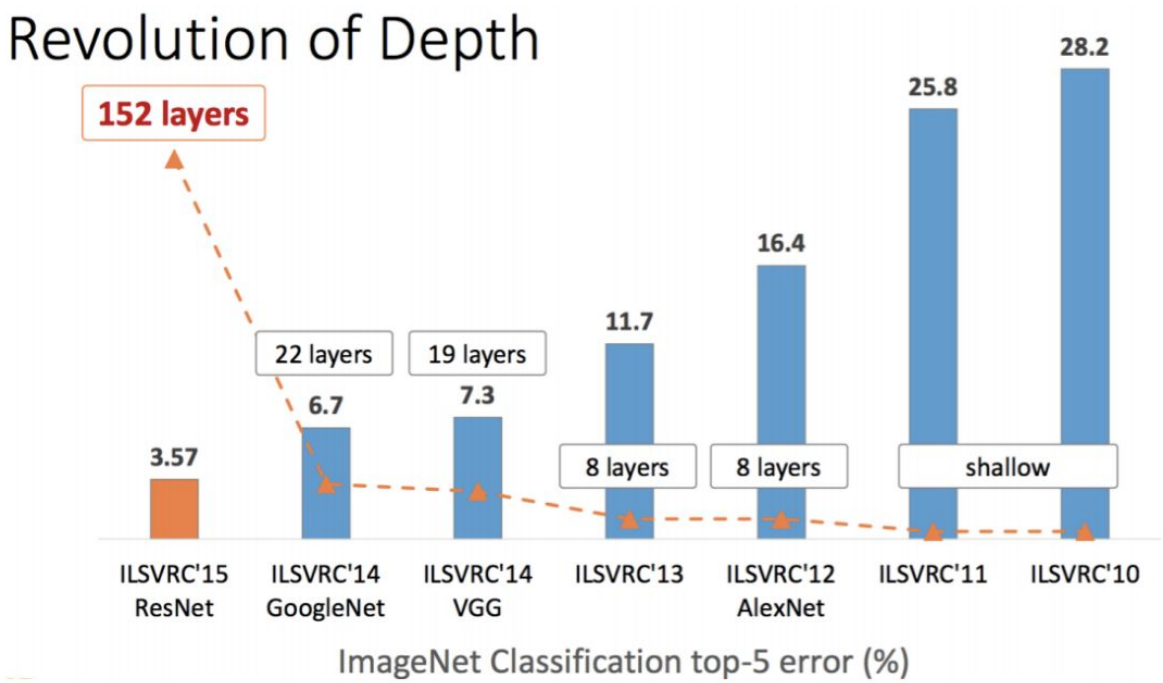
- AlexNet, 2012. 8 weight layers. 16.4% top-5 error (i.e. the network gets 5 tries to guess the right category).
- Closest competitor: 26.1%



(Krizhevsky et al., 2012)

- The two processing pathways correspond to 2 GPUs. (At the time, the network couldn't fit on one GPU.)
- AlexNet's stunning performance on the ILSVRC is what set off the deep learning boom of the last 6 years.

Results: Object Classification



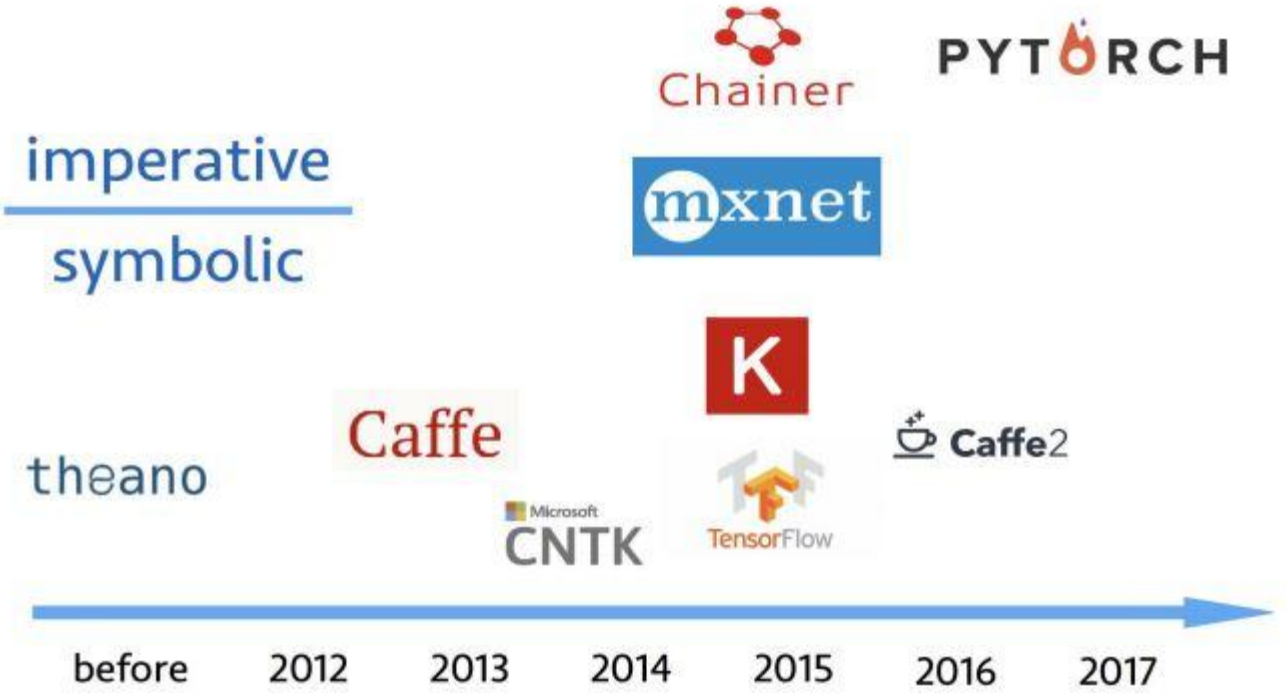
Slide: R. Liao, Paper: [He, K., Zhang, X., Ren, S. and Sun, J., 2015. Deep Residual Learning for Image Recognition. arXiv:1512.03385, 2016]

Outline

- Image Recognition
- Machine Learning and Applications
  - Supervised Learning vs. Unsupervised Learning
- Supervised learning
  - Classification vs Regression
  - Overfitting vs Underfitting
- Neural Networks
  - Activation Functions?
  - Optimization?
  - Loss Functions?
- Convolutional Neural Networks
- Hands-on Demo with Fashion MNIST

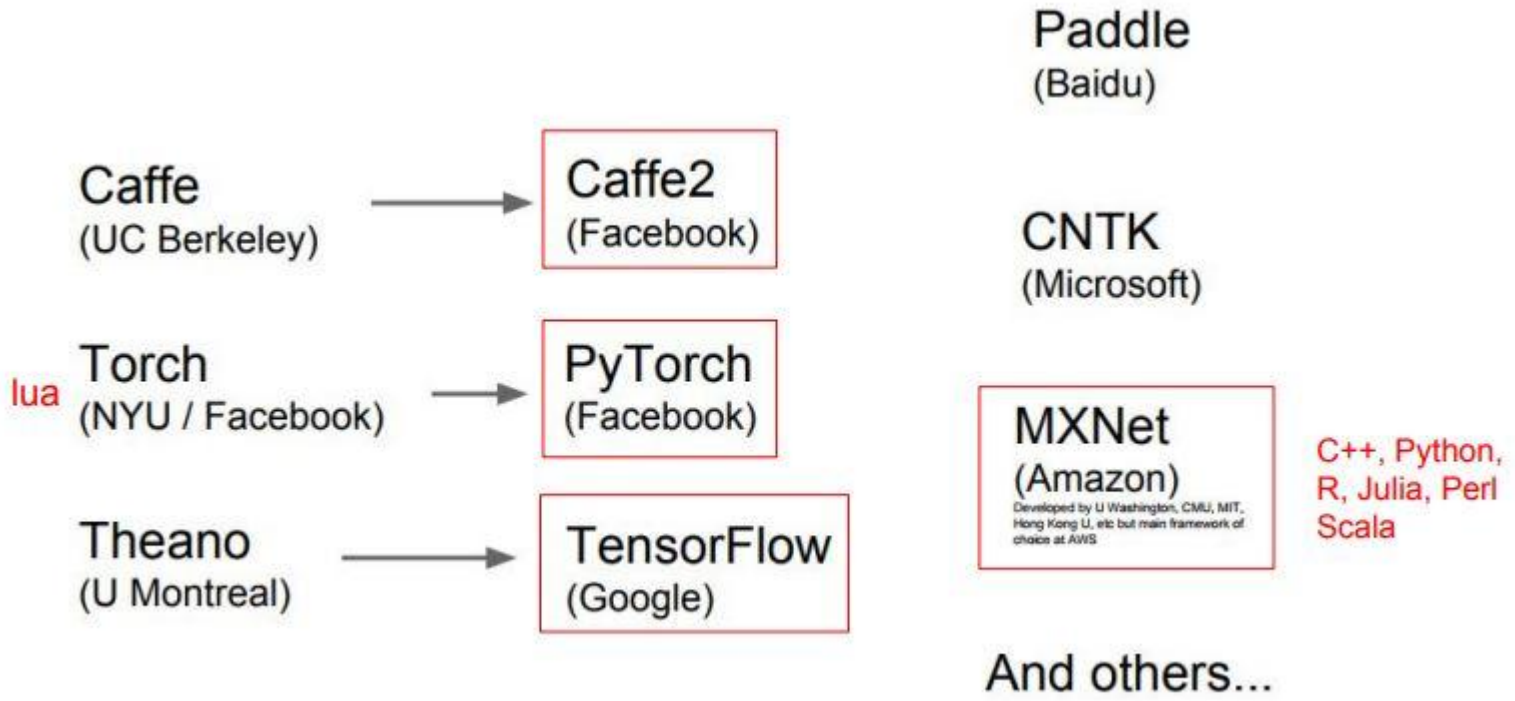
Popular Machine Learning Frameworks

- **Imperative**: perform computation as you run them.
- **Symbolic**: define the function first, then compile them.



https://web.cs.ucdavis.edu/~yjlee/teaching/ecs289g-winter2018

Popular Machine Learning Frameworks



https://web.cs.ucdavis.edu/~yjlee/teaching/ecs289g-winter2018

PyTorch Framework

Flexible, and easy for implementing sophisticated algorithms.

Three levels of abtraction:

1. **Tensors:** Imperative ndarrays that run on GPU.
2. **Variables:** Nodes in a computational graph that store data and their gradients.
3. **Module:** A neural network layer that may store and learn weight and parameters.

How to implement a training procedure on torch?

1. Loading data.
2. Building model (architecture).
3. Defining loss function (criterion) and optimizer.
4. Merging the previous steps in your “training loop”.
5. Evaluating the model on the test set.

We will see an implementation example in the shared hands-on notebook.

Model Definition

Define Layers

- The output layer (fc2) has 10 units (10 digits)

Define the Forward Pass

- Call each layer with the input tensor as argument

Exercise:

Print the dimension of the tensor at each layer

```
class Net(nn.Module):
    #This defines the structure of the NN.
    def __init__(self):
        super(Net, self).__init__()
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        self.conv2 = nn.Conv2d(10, 20, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()  #Dropout
        self.fc1 = nn.Linear(320, 50)
        self.fc2 = nn.Linear(50, 10)

    def forward(self, x):
        #Convolutional Layer/Pooling Layer/Activation
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        #Convolutional Layer/Dropout/Pooling Layer/Activation
        x = F.relu(F.max_pool2d(self.conv2_drop(self.conv2(x)), 2))
        x = x.view(-1, 320)
        #Fully Connected Layer/Activation
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        #Fully Connected Layer/Activation
        x = self.fc2(x)
        #Softmax gets probabilities.
        return F.log_softmax(x, dim=1)
```

Define Optimization Steps

```
def train(epoch):
    model.train()
    for batch_idx, (data, target) in enumerate(train_loader):
        if args['cuda']:
            data, target = data.cuda(), target.cuda()
        #Variables in Pytorch are differenciable.
        data, target = Variable(data), Variable(target)
        #This will zero out the gradients for this batch.
        optimizer.zero_grad()
1       output = model(data)
        # Calculate the loss The negative log likelihood loss. It is useful
2       loss = F.nll_loss(output, target)
        #dloss/dx for every Variable
3       loss.backward()
        #to do a one-step update on our parameter.
4       optimizer.step()
        #Print out the loss periodically.
        if batch_idx % args['log_interval'] == 0:
            print('Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
                epoch, batch_idx * len(data), len(train_loader.dataset),
                100. * batch_idx / len(train_loader), loss.data[0]))
```

Any questions?

Then, let’s go to our hands-on notebook!