# Deep Learning and Its Applications

## Convolutional Neural Network and Its Application in Image Recognition

Xin Sui

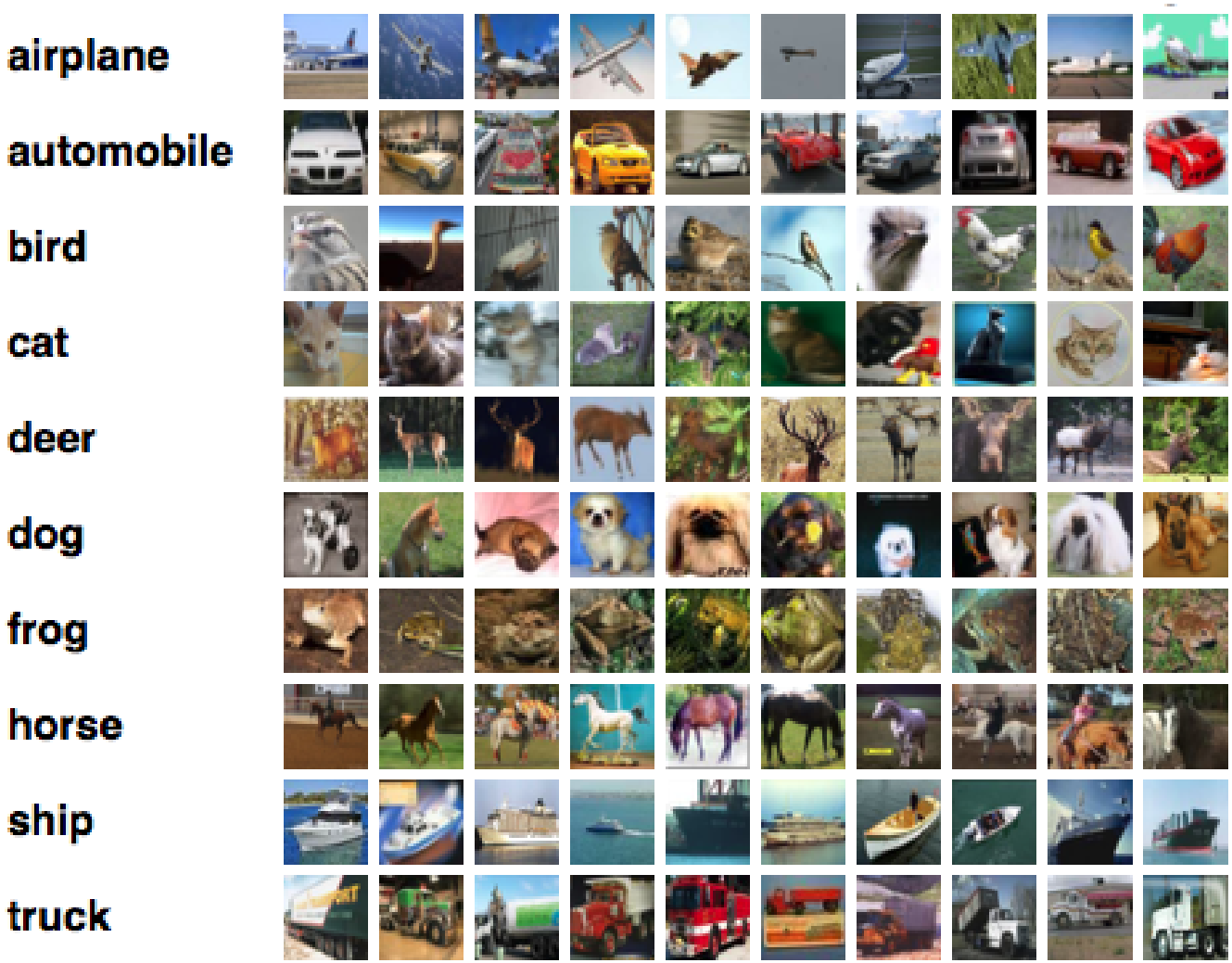
Oct 28, 2016

# Outline


1. A Motivating Example
2. The Convolutional Neural Network (CNN) Model
3. Training the CNN Model
4. Issues and Recent Advances
5. Code Demonstration

# A Motivating Example

- Image Classification: A classic application of CNN

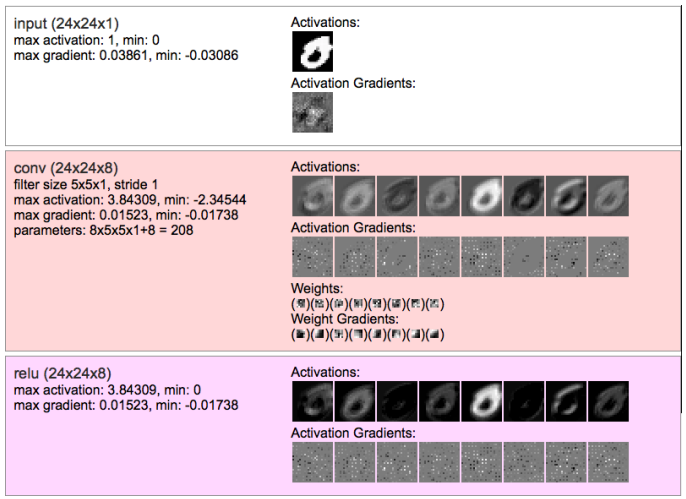


# The CNN Model

- Visualization of a typical CNN Model
- Layers in the model
  1. Convolutional Layer: 1 feature map $\rightarrow$ 8 feature maps
  2. Pooling Layer: shrinks the resolution: $24 \times 24 \rightarrow 12 \times 12$
  3. Convolutional Layer: 8 feature maps $\rightarrow$ 16 feature maps
  4. Pooling Layer: shrinks the resolution: $12 \times 12 \rightarrow 4 \times 4$
  5. Fully Connected Layer: each output fully connects all $4 * 4 * 16$ pixels
  6. Softmax Layer: applies the softmax function

# Key Elements in Convolutional Layer

- This convolutional layer takes 1 feature map of size $24 \times 24$ to 8 feature maps of size $24 \times 24$.
- In: $p_{in}$ feature maps (images) of resolution $m_{in} \times n_{in}$: (denote each pixel by) $I_i[x, y]$, $i \in [p_{in}], x \in [m_{in}], y \in [n_{in}]$, where $[a] = \{0, 1, 2, \cdots, a-1\}$.
- Out: $p_{out}$ feature maps of resolution $m_{out} \times n_{out}$: $O_j[x, y]$, $j \in [p_{out}], x \in [m_{out}], y \in [n_{out}]$.
- Filter size $u \times v \times p_{in}$.
- Activation function $h(\cdot)$. E.g., relu function: $h(x) = \max\{0, x\}$.



# 2-D Discrete Convolution

- For simplicity, assume $p_{in} = p_{out} = 1$, and $h(x) = x$ for now. Let $F[x, y]$ be the filter of size $u \times v \times 1$, that is, $x \in [u], y \in [v]$.
- Recall $I[x, y]$ is the input image of size $m_{in} \times n_{in}$, while $O[x, y]$ is the output image of size $m_{out} \times n_{out}$.
- Then,

$$O[x, y] = \sum_{a \in [u]} \sum_{b \in [v]} I[x - a, y - b] F[a, b] := (I * F)[x, y]$$

for all $x \in [m_{out}], y \in [n_{out}]$.

  - The operator $*$ denotes 2-D discrete convolution. For simplicity, denote this by $O = I * F$
  - Let $I[x - a, y - b] = 0$ when $x - a \notin [u]$ or $y - b \notin [v]$.
  - Visualization of 2-D convolution
  - Convolution detects features in an image

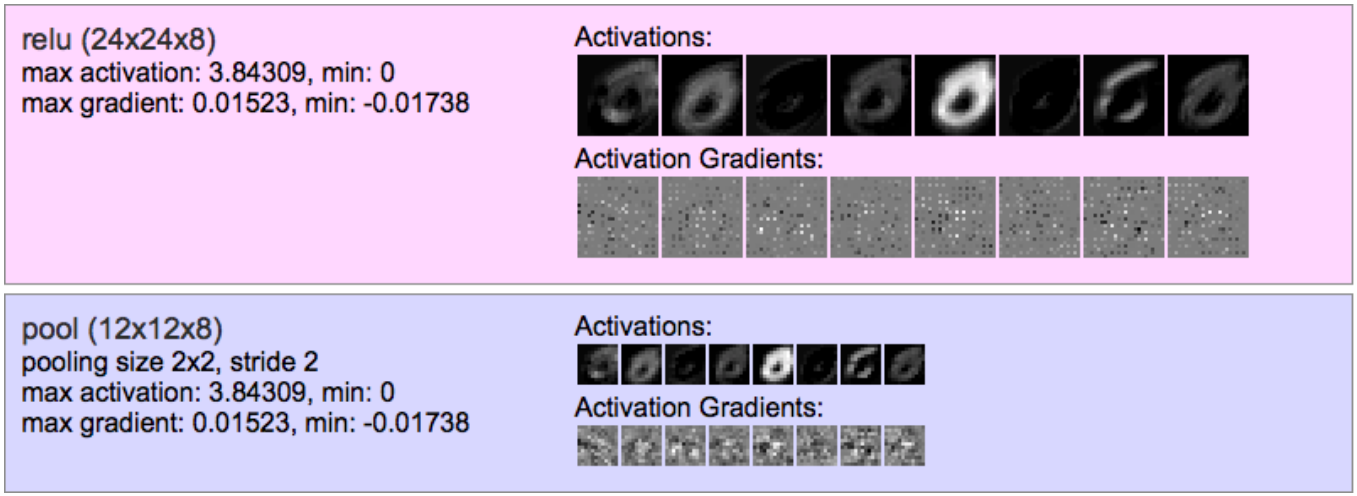
- For simplicity, it is commonly used that $(m_{out}, n_{out}) = (m_{in}, n_{in})$. It is named 'same padding'.

# Convolutional Layer

- Let us remove the assumptions $p_{in} = p_{out} = 1$, and $h(x) = x$ one-by-one to obtain the final form of a convolutional layer.
- When $p_{in} \neq 1$,
$$O = \sum_{i \in [p_{in}]} I_i * F_i.$$
- Additionally, when $p_{out} \neq 1$,
$$O_j = \sum_{i \in [p_{in}]} I_i * F_i^{(j)} \text{ for all } j.$$
- In the most general case, where $h(x) \neq x$ and there is an additional bias term,
$$O_j = h(\sum_{i \in [p_{in}]} I_i * F_i^{(j)} + b_j) \text{ for all } j.$$
- Note that in a convolutional layer, there are $p_{out}$ filters, each of size $u \times v \times p_{in}$.

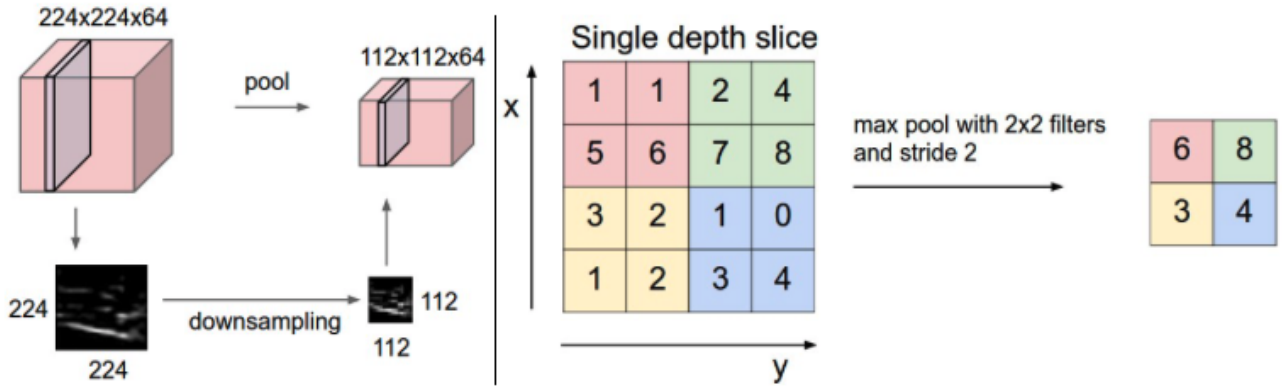
# Key Elements in Pooling Layer

- This pooling layer takes 8 feature maps of size $24 \times 24$ to 8 feature maps of size $12 \times 12$.
- In: $p$ feature maps of resolution $m_{in} \times n_{in}$: $I_i[x, y]$, $i \in [p], x \in [m_{in}], y \in [n_{in}]$.
- Out: $p$ feature maps of resolution $m_{out} \times n_{out}$: $O_i[x, y]$, $i \in [p], x \in [m_{out}], y \in [n_{out}]$.
- Pooling size: $u \times v$.
- Pooling method: max pooling, average pooling, etc.



# Pooling

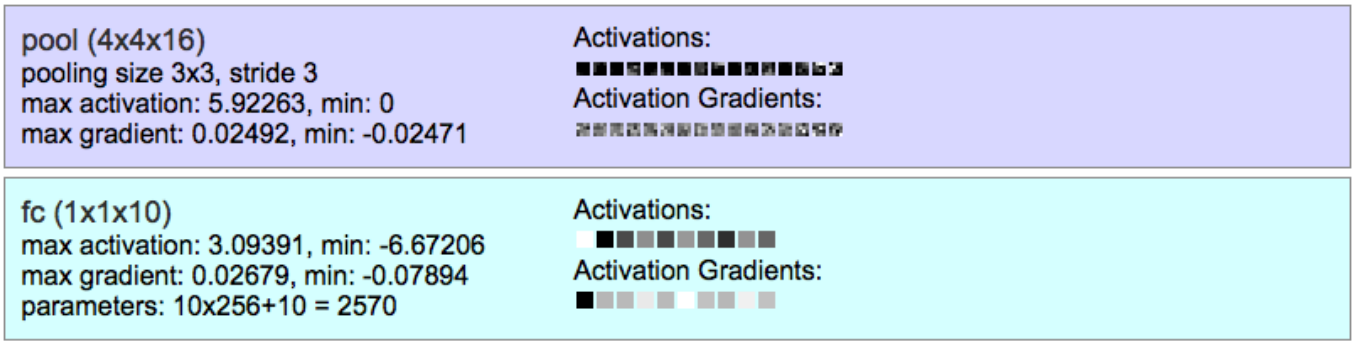
- For all $i \in [p]$, obtain $O_i$ from $I_i$. Intuitively,




- Max-pooling introduces non-linearity in the kernel.

# Fully-connected Layer

- In: $p_{in}$ variables $\boldsymbol{x} \in \mathbb{R}^{p_{in}}$ (every pixel is considered a variable).
- Out: $p_{out}$ variables $\boldsymbol{y} \in \mathbb{R}^{p_{out}}$.
- Activation function $h(\cdot)$.




- The model:

$$\boldsymbol{y} = h(\boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}),$$

where $\boldsymbol{W} \in \mathbb{R}^{p_{out} \times p_{in}}$, $\boldsymbol{b} \in \mathbb{R}^{p_{out}}$, and $h(\cdot)$ is a component-wise function.

# Softmax Function

- For any $\boldsymbol{x} \in \mathbb{R}^K$, the softmax function for every $k \in [K]$ is defined as

$$softmax(k, \boldsymbol{x}) = \frac{e^{x_k}}{\sum_{i \in [K]} e^{x_i}}.$$

- Let $p_k = \mathbb{P}(Y = k)$ be the probability that a given image is in the $k^{\text{th}}$ class, then under the model assumption

$$\mathbb{P}(Y = k) = \frac{1}{Z} e^{\boldsymbol{w}_k^T \boldsymbol{x} + b_k}$$

for all $k \in [K]$ and some explanatory variables $\boldsymbol{x}$, we have

$$p_k = softmax(k, \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}),$$

where $\boldsymbol{W} = [\boldsymbol{w}_1, \boldsymbol{w}_2, \cdots, \boldsymbol{w}_K]^T$ and $\boldsymbol{b} = (b_1, b_2, \cdots, b_K)^T$.

- Combine fully connected layer with $h(x) = x, p_{out} = K$ and the softmax function to obtain class probabilities.
- Cross-entropy loss (essentially negative log-likelihood) can be used to train the model.

# Recap of the CNN Model

- Convolutional Layer: generates features by varying filters.
- Pooling Layer: performs subsampling that shrinks the dimensionality.
- Fully-connected layer + Softmax function: obtain class probabilities.
- Note that Fully-connected layers can also be used alone to introduce non-linearity ($h(x) \neq x$) or shrink dimensionality($p_{out} < p_{in}$).
- A typical CNN model: Conv-Pool-Conv-Pool-FC-FC-Softmax. The more layers there are, the 'deeper' the model is.
- Fully connected layers are usually placed at the end of the model, because it fully connects all pixels and breaks their spatial correlation.

# Optimization: Stochastic Gradient Descent

- When 1) data size is too large, or 2) data is obtained in a sequential way, that some future data is not accessible currently, gradient descent may not be practical to train a CNN model.
- Stochastic Gradient Descent (SGD) can be used.
  - Split the whole dataset $(\boldsymbol{X}, \boldsymbol{y})$ into $B$ mini-batches $(\boldsymbol{X}^{(i)}, \boldsymbol{y}^{(i)})$, $i = 1, 2, \cdots, B$.
  - At iteration $t$, find a mini-batch $(\boldsymbol{X}^t, \boldsymbol{y}^t)$ (usually iteratively) and then perform update $\boldsymbol{\theta}^{(t)} \leftarrow \boldsymbol{\theta}^{(t-1)} - \eta \nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}^{(t-1)} | \boldsymbol{X}^t, \boldsymbol{y}^t)$, where $\boldsymbol{\theta}$ is the set of all parameters in the model, and $f(\cdot)$ is the loss function.
- Challenge: vanishing gradient problem

# Vanishing Gradient Problem

- Let $\sigma(x) = 1/(1 + \exp\{-x\})$ be the commonly-used sigmoid function.
- Assume a neural network $x_{k+1} = \sigma(w_k x_k)$, where $k = 1, 2, \cdots, K$, where $x_1$ is the input. Let $f(x_{K+1})$ be the loss function.

$$\frac{\partial f}{\partial w_1} = \frac{\partial f}{\partial x_{K+1}} \frac{\partial x_{K+1}}{\partial x_K} \cdots \frac{\partial x_3}{\partial x_2} \frac{\partial x_2}{\partial w_1} = f'(x_{K+1}) \prod_{k=2}^{K} \sigma'(w_k x_k) w_k \sigma'(w_1 x_1) x_1$$

- Since $\sigma'(x) \leq 1/4$, $\frac{\partial f}{\partial w_1} \leq (1/4)^K f'(x_{K+1}) \prod_{k=2}^{K} w_k x_1$.
- Intuitively, when $w_k$'s are not too large, the gradient vanishes as $K$ increases.
- Practically, the first few layers are hard to train when the network is deep.

# To Deal With This Issue

- There are several ways to deal with this issue in deep learning.

1. Use other activation functions, such as ReLU: $h(x) = max\{0, x\}$. Since $\max h'(x) = 1$, it helps to pass down the gradient while keeps non-linearity.
2. Use other ways to connect layers, such as short-cut connections.
3. Modify the way to update parameters, for example, use momentum in the updates.

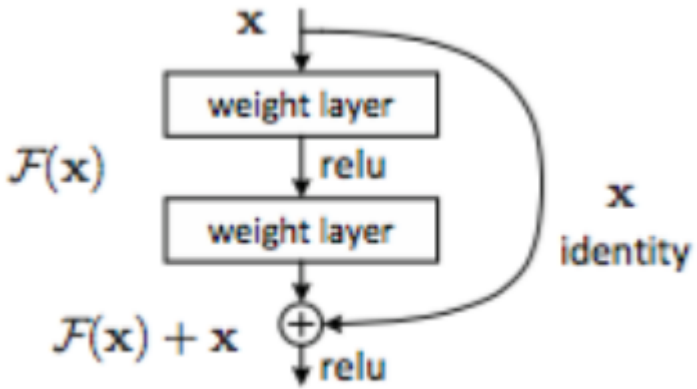



Figure 2. Residual learning: a building block.

# Deep Learning

- Why deep models?
  - Theoretically, the intuition is that, the deeper the model, the more complex the feature space is.
  - Computationally, deep learning models, such as CNN, leads to easy massive parallelization.
  - But hard to interpret.
- Some frontier applications of deep learning

# Packages: Theano/TensorFlow

- These packages are quite similar. In fact, some developers of Theano went to develop TensorFlow for Google.
- Low-level, highly customizable.
- Automatic computation of derivatives.
- Same code can be run on either CPU or GPU.
- Fast (cuda) C implementations.
- High-level libraries, such as pylearn2, are available.
- Actively developed and maintained. (In the past year in Theano, they added support for multiple GPUs, implemented average pooling, and made faster implementations available as far as I know)
- Hard to debug.
- (My code is available online)

# Graph

$$c = a^2 + 2b - 2a$$