Deep Learning for Robotic Vision

An Introduction

Niko Suenderhauf

Queensland University of Technology
Australian Centre for Robotic Vision
What is Deep Learning?
What is Deep Learning?

Artificial Intelligence
What is Deep Learning?

- Intelligence demonstrated by machines.
  - The study of "intelligent agents": any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals.
  - Machines that mimic "cognitive" functions that humans associate with the human mind, such as "learning" and "problem solving".
What is Deep Learning?

Machine learning is the scientific study of algorithms and statistical models that computer systems use to perform a specific task without using explicit instructions, relying on patterns and inference instead.

Machine Learning algorithms build a mathematical model based on sample data, known as "training data", in order to make predictions or decisions without being explicitly programmed to perform the task.
**What is Deep Learning?**

Deep learning allows computational models that are composed of multiple processing layers to learn representations of data with multiple levels of abstraction.

Deep learning discovers intricate structure in large data sets by using the backpropagation algorithm to indicate how a machine should change its internal parameters that are used to compute the representation in each layer from the representation in the previous layer.

What is Robotic Vision?
What is Robotic Vision?

Input
- Images
- Data

Output
- Images
- Data
What is Robotic Vision?

**Input**
- Images
- Data

**Output**
- Images
- Data

Diagram:
- Images → Image Processing
- Data → ?
- ? → ?
- ? → ?
What is Robotic Vision?

Input

Images

Data

Output

Images

Image Processing

Data

Computer Graphics
What is Robotic Vision?

Input

Images

Data

Output

Images

Data

Images Processing

Computer Vision

Computer Graphics

?
What is Robotic Vision?

Input

Images

Images

Data Science

Images

Image Processing

Computer Vision

Data

Computer Graphics
What is Robotic Vision?

“Computer Vision on a robot?”

Input

Images

Data

Output

Images

Data

Images

Computer Graphics

Computer Vision

Data Science
What is Robotic Vision?

Output

“Computer Vision on a robot?”
What is Robotic Vision?
What is Robotic Vision?

This is where robotic vision differs from computer vision. For robotic vision, perception is only one part of a more complex, embodied, active, and goal-driven system.

Robotic vision therefore has to take into account that its immediate outputs (object detection, segmentation, depth estimates, 3D reconstruction, a description of the scene, and so on), will ultimately result in actions in the real world.

In a simplified view, whereas computer vision takes images and translates them into information, robotic vision translates images into actions.

Supervised (Deep) Learning
Supervised Learning

Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-output pairs.

It infers a function from labeled training data consisting of a set of training examples.
Supervised Learning

Supervised learning is the machine learning task of learning a function that maps an input to an output based on example input-output pairs.

It infers a function from labeled training data consisting of a set of training examples.

Training examples: (image, label) $X = \{(\text{image1}, \text{label1}), (\text{image2}, \text{label2}), (\text{image3}, \text{label3}), \ldots \}$
Supervised Learning

Training examples: (image, label)

$X = \{(\text{dog}, \text{dog}), (\text{cat}, \text{cat}), (\text{car}, \text{car}), \ldots \}$

Goal: Learn function $f: \text{Image} \rightarrow \text{Label}$

$f(\text{cat}) = \text{cat}$ (if all goes well)
Nearest Neighbor Classifiers
Every Image can be rearranged into a vector.
3072-Dimensional Space
3072-Dimensional Space
Linear Classifiers
The diagram illustrates a hyperplane defined by the weight vector $w$, which separates two sets of data points. The set $A$ is on the right side of the hyperplane, and the set $B$ is on the left side. The conditions $w^T x < 0$ and $w^T x > 0$ indicate which side of the hyperplane a point lies on. The blue dots represent set $A$, and the red dots represent set $B$. The green line represents the decision boundary determined by the weight vector $w$.
\[ w = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \]

\[ w^T x < 0 \]

\[ w^T x > 0 \]
\[
\begin{align*}
\mathbf{w} &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} \\
\mathbf{w}^T \mathbf{x} &> 0 \\
\mathbf{w}^T \mathbf{x} &< 0 \\
\mathbf{A} &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} 5 \\ 5 \end{pmatrix} = 10 \\
\mathbf{A} &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} -5 \\ 5 \end{pmatrix} = 0 \\
\mathbf{A} &= \begin{pmatrix} 1 \\ 1 \end{pmatrix} \cdot \begin{pmatrix} -10 \\ 5 \end{pmatrix} = -5
\end{align*}
\]
\[ w^T x + b \geq 0 \]
Interpret values of $y$ as **class-confidences**.

The bigger $y_i$, the more confident we are that $x$ is of class $i$. 

$$y = W^T x + b$$
\[ W^T = \begin{pmatrix} 0.15 & 0.36 \\ -1.63 & 0.28 \\ 0.25 & -1.05 \end{pmatrix} \quad b = \begin{pmatrix} -0.84 \\ 1.57 \\ -0.02 \end{pmatrix} \]

\[ y = W^T x + b \]
\[ \mathbf{w}^T = \begin{pmatrix} 0.15 & 0.36 \\ -1.63 & 0.28 \\ 0.25 & -1.05 \end{pmatrix} \quad \mathbf{b} = \begin{pmatrix} -0.84 \\ 1.57 \\ -0.02 \end{pmatrix} \]

\[ y = \mathbf{w}^T \mathbf{x} + \mathbf{b} \]

\[ \mathbf{w}^T \cdot \begin{pmatrix} 10 \\ 10 \end{pmatrix} + \mathbf{b} = \]

\[ \mathbf{w}^T \cdot \begin{pmatrix} 10 \\ -5 \end{pmatrix} + \mathbf{b} = \]

\[ \mathbf{w}^T \cdot \begin{pmatrix} -10 \\ 0 \end{pmatrix} + \mathbf{b} = \]
\[ W^T = \begin{pmatrix} 0.15 & 0.36 \\ -1.63 & 0.28 \\ 0.25 & -1.05 \end{pmatrix}, \quad b = \begin{pmatrix} -0.84 \\ 1.57 \\ -0.02 \end{pmatrix} \]

\[ y = W^T x + b \]
\[ W^T = \begin{pmatrix} 0.15 & 0.36 \\ -1.63 & 0.28 \\ 0.25 & -1.05 \end{pmatrix} \quad b = \begin{pmatrix} -0.84 \\ 1.57 \\ -0.02 \end{pmatrix} \]

\[ y = W^T x + b \]
We are actually projecting from 2D into 3D!

\[
\begin{align*}
W^T \cdot \begin{pmatrix} 10 \\ 10 \end{pmatrix} + b &= \begin{pmatrix} 3.99 \\ -11.94 \\ -8.04 \end{pmatrix} \\
W^T \cdot \begin{pmatrix} 10 \\ -5 \end{pmatrix} + b &= \begin{pmatrix} -1.03 \\ -16.12 \\ 7.75 \end{pmatrix} \\
W^T \cdot \begin{pmatrix} -10 \\ 0 \end{pmatrix} + b &= \begin{pmatrix} -2.32 \\ 17.87 \\ -2.53 \end{pmatrix}
\end{align*}
\]

\[y = W^T x + b\]
\[ y = W^T x + b \]
Towards a Neural Network
\[ y = W^T x + b \]

\[ W = \begin{pmatrix}
  w_{0,0} & w_{0,1} & w_{0,2} \\
  w_{1,0} & w_{1,1} & w_{1,2}
\end{pmatrix} \]

\[ b + \sum_i w_{i,0} \cdot x_i \]

\[ b + \sum_i w_{i,1} \cdot x_i \]

\[ b + \sum_i w_{i,2} \cdot x_i \]
\[ y = W^T x + b \]

```python
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 3)

    def forward(self, x):
        x = self.fc1(x)
        return x
```
\[ y = W^T x + b \]

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(2, 3)

    def forward(self, x):
        x = self.fc1(x)
        return x

W = net.state_dict()['fc1.weight'].numpy()
b = net.state_dict()['fc1.bias'].numpy()
print('weights W:
', W)
print('bias b:
', b)

weights W:
[[ 0.32842654  0.49274433]
 [-1.6420735   0.38735208]
 [ 0.42878065 -1.0042973 ]]  
bias b:
[-0.98728037  2.0173173  0.08274412]
Every Image can be rearranged into a vector.

Shape: (32,32,3)

Shape: (1024,1,3)

Shape: (3072,1)
$y = W^T x + b$
\[ y = W^T x + b \]

class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(3072, 10)

    def forward(self, x):
        x = self.fc1(x)
        return x
Loss Functions
(How Good is the Model?)
Loss Function

How good or bad are the current parameters?

\[ y = W^T x + b \]
Loss Function

How good or bad are the current parameters?

Cross-Entropy Loss (Softmax Classifier)

- Interpret outputs $y$ as probabilities for each class.
  - (unnormalised log-probabilities)
  - e.g. apply Softmax function to get probabilities

$$L = -y_{\text{true}} + \log \sum_j e^{y_j}$$

score assigned to true class

$$L = - \log \left( \frac{e^{y_{\text{true}}}}{\sum_j e^{y_j}} \right)$$
Loss Function Example 1

True class: “0”

\[ L = -y_{\text{true}} + \log \sum_j e^{y_j} \]

\[ y = (5, -10, -10)^T \]

\[ L = -5 + \log (e^5 + e^{-10} + e^{-10}) \]

\[ = -5 + \log 148.4132 \]

\[ = -5 + 5.000000276 \]

\[ \approx 0 \]
Loss Function Example 2

True class: “1”

\[ L = -y_{\text{true}} + \log \sum_j e^{y_j} \]

\[
\begin{align*}
y &= (5, -10, -10)^T \\
L &= 10 + \log (e^5 + e^{-10} + e^{-10}) \\
    &= 10 + \log 148.4132 \\
    &= 10 + 5.000000276 \\
    \approx 15
\end{align*}
\]
Cross Entropy Loss Intuition

\[ L = -y_{\text{true}} + \log \sum_j e^{y_j} \]

approximates a max function!
Cross Entropy Loss Intuition

$$L = -y_{\text{true}} + \log \sum_{j} e^{y_{j}}$$

approximates a max function!

```python
p = np.random.randn(5)*10
print(p)
np.log(np.sum(np.exp(p)))
```

```
[ -5.7700444  -13.26877559  -6.04029885  2.19204188  6.73428813]
6.744887755211229
```
Cross Entropy Loss Intuition

\[ L = -y_{true} + \log \sum_j e^{y_j} \]

approximates a max function!

→ Minimum Loss when: highest score for correct class!
Cross Entropy Loss Intuition

\[ L = -y_{\text{true}} + \log \sum_j e^{y_j} \]

→ Minimum Loss when: **highest score for correct class**!

→ minimize average loss for all training samples
Training
Finding Good Weights (and Biases)
How do we find the best \((W,b)\)?

Objective: minimize average loss for all training samples. But how? Some ideas:

- Random search
  - randomly choose \((W,b)\), and remember the best
How do we find the best $(W,b)$?

\[
L = -y_{\text{true}} + \log \sum_j e^{y_j}
\]

Objective: minimize average loss for all training samples. But how? Some ideas:

- Random search
  - randomly choose $(W,b)$, and remember the best
- Random local search
  - randomly change $(W,b)$ slowly by adding a small increment, check if that made it better
How do we find the best \((W,b)\)?

Objective: minimize average loss for all training samples. But how? Some ideas:

- Random search
  - randomly choose \((W,b)\), and remember the best
- Random local search
  - randomly change \((W,b)\) slowly by adding a small increment, check if that made it better
- Follow the gradient
  - systematically change \((W,b)\) by computing derivatives

\[
L = -y_{\text{true}} + \log \sum_j e^{y_j}
\]
Gradient Descent

\[ W^+ = W - s \cdot \frac{\partial L}{\partial W} \]

\[ b^+ = b - s \cdot \frac{\partial L}{\partial b} \]

\[ y = W^T x + b \]

\[ L = -y_{true} + \log \sum_j e^{y_j} \]

learning rate step size

derivative of loss with respect to the weights
Gradient Descent

\[ W^+ = W - s \cdot \frac{\partial L}{\partial W} \]

\[ b^+ = b - s \cdot \frac{\partial L}{\partial b} \]

\[ y = W^T x + b \]

\[ L = -y_{\text{true}} + \log \sum_j e^{y_j} \]

Fortunately, automatic differentiation is part of most DL libraries! Same for various optimization methods!
Training a simple linear classifier

\[ y = W^T x + b \]
And Now: Actual Neural Networks
Missing Ingredient

(nonlinear) activation function

- Linear models are often overly simple
- Enables meaningful “stacking” of layers → deep networks

\[
\begin{align*}
  a(b + \sum_i w_{i,0} \cdot x_i) \\
  a(b + \sum_i w_{i,1} \cdot x_i) \\
  a(b + \sum_i w_{i,2} \cdot x_i)
\end{align*}
\]
Missing Ingredient

(nonlinear) activation function

- Linear models are often overly simple
- Enables meaningful “stacking” of layers → deep networks
- Historically: sigmoid function

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]
Missing Ingredient

(nonlinear) activation function

- Linear models are often overly simple
- Enables meaningful “stacking” of layers → deep networks
- Historically: sigmoid function

\[
\sigma(x) = \frac{1}{1 + e^{-x}}
\]

- Many other choices:
  - \( \tanh(x) \)
  - Rectified Linear Unit ReLU = \( \text{max}(0,x) \)
  - ...
Missing Ingredient

(nonlinear) activation function

- Linear models are often overly simple
- Enables meaningful “stacking” of layers → deep networks
- Historically: sigmoid function

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]

- Many other choices:
  - $\tanh(x)$
  - **Rectified Linear Unit ReLU** = $\max(0,x)$
  - ReLU is most commonly used
Missing Ingredient

(nonlinear) activation function

- Linear models are often overly simple
- Enables meaningful “stacking” of layers → deep networks
- Historically: sigmoid function

\[ \sigma(x) = \frac{1}{1 + e^{-x}} \]
Deep Networks

\[ z = a(W_0^T x + b_0) \]

\[ y = W_1^T z + b_1 \]

Shape: (32,32,3)

Shape: (3072,1)
Convolutional Networks
Kernel

Image

Patch

Dot product
3 channels (RGB)
shape (3, 244, 244)
Convolution:
Slide filter over all locations, perform dot product.

3 x 244 x 244 Image

3 x 11 x 11 filter

1 (scalar) result
Convolution:
Slide filter over all locations, perform dot product.

3 x 11 x 11 filter

3 x 244 x 244 Image
Convolution:
Slide filter over all locations, perform dot product.
**Convolution:**
Slide filter over all locations, perform dot product.

3 x 244 x 244 Image

3 x 11 x 11 filter

---

**Image:**
A grayscale image of a cat with a spiral pattern drawn over it.
1st Convolutional Layer

Alexnet

ResNeXt
3 channels (RGB)
shape (3, 244, 244)

Alexnet Conv1: 64 filters, size (3, 11, 11)
Alexnet Conv1: 64 filters, size (3, 11, 11)
3 channels (RGB)
shape (3, 244, 244)

Result: (64, 55, 55)
3 channels (RGB)
shape (3, 244, 244)

conv1 (64, 55, 55)
3 channels (RGB) shape (3, 244, 244)

conv1 (64, 55, 55)

conv2 (192, 27, 27)
3 channels (RGB)
shape (3, 244, 244)

conv1 (64, 55, 55)

conv2 (192, 27, 27)
3 channels (RGB)
shape (3, 244, 244)

conv1 (64, 55, 55)

conv2 (192, 27, 27)
3 channels (RGB)
shape (3, 244, 244)

conv1 (64, 55, 55)

conv2 (192, 27, 27)
3 channels (RGB)
shape (3, 244, 244)

conv1 (64, 55, 55)

conv2 (192, 27, 27)
3 channels (RGB)
shape (3, 244, 244)

conv1 (64, 55, 55)

conv2 (192, 27, 27)
3 channels (RGB)
shape (3, 244, 244)

conv1 (64, 55, 55)

conv2 (192, 27, 27)
```python
alexnet = torchvision.models.alexnet(pretrained=True)
print(alexnet)

AlexNet(
    (features): Sequential(
        (0): Conv2d(3, 64, kernel_size=(11, 11), stride=(4, 4), padding=(2, 2))
        (1): ReLU(inplace)
        (2): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
        (3): Conv2d(64, 192, kernel_size=(5, 5), stride=(1, 1), padding=(2, 2))
        (4): ReLU(inplace)
        (5): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
        (6): Conv2d(192, 384, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (7): ReLU(inplace)
        (8): Conv2d(384, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (9): ReLU(inplace)
        (10): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
        (11): ReLU(inplace)
        (12): MaxPool2d(kernel_size=3, stride=2, padding=0, dilation=1, ceil_mode=False)
    )
    (avgpool): AdaptiveAvgPool2d(output_size=(6, 6))
    (classifier): Sequential(
        (0): Dropout(p=0.5)
        (1): Linear(in_features=9216, out_features=4096, bias=True)
        (2): ReLU(inplace)
        (3): Dropout(p=0.5)
        (4): Linear(in_features=4096, out_features=4096, bias=True)
        (5): ReLU(inplace)
        (6): Linear(in_features=4096, out_features=1000, bias=True)
    )
)
```
```python
resnext = torchvision.models.resnext101_32x8d(pretrained=True)
print(resnext)
```
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
3 channels (RGB)
shape (3, 244, 244)
1000 classes

Shape: (9216,1)  Shape: (4096,1)  Shape: (1000,1)
Nonlinear projections from one space into another. Until classes are linearly separable.
Backpropagation
3 channels (RGB)
shape (3, 244, 244)

(64, 55, 55)
(192, 27, 27)

predictions (1, 10)
3 channels (RGB) shape (3, 244, 244)

O_0 (64, 55, 55)

O_1 (192, 27, 27)

predictions (1, 10)

conv1 parameters

W_0

conv2 parameters

W_1

fc1 parameters

W_2

loss
3 channels (RGB)
shape (3, 244, 244)

predictions (1, 10)
3 channels (RGB)
shape (3, 244, 244)

3 channels (RGB)
shape (3, 244, 244)

$O_0$ (64, 55, 55)

$O_1$ (192, 27, 27)

predictions (1, 10)

$L = f(o_2)$

$o_2 = g(o_1, W_2)$

$L = f(g(o_1, W_2))$
3 channels (RGB)
shape (3, 244, 244)

\[ \mathbf{O}_0 \quad (64, 55, 55) \]
\[ \mathbf{O}_1 \quad (192, 27, 27) \]
predictions (1, 10)

\[ \mathbf{L} = f(\mathbf{o}_2) \]
\[ \mathbf{o}_2 = g(\mathbf{o}_1, \mathbf{W}_2) \]
\[ \mathbf{L} = f(g(\mathbf{o}_1, \mathbf{W}_2)) \]

\[ \frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial o_2} \cdot \frac{\partial o_2}{\partial W_2} \]
3 channels (RGB)
shape (3, 244, 244)

\[ O_0 = (64, 55, 55) \]

\[ O_1 = (192, 27, 27) \]

predictions (1, 10)

\[ L = f(o_2) \]

\[ o_2 = g(o_1, W_2) \]

\[ L = f(g(o_1, W_2)) \]

\[ o_1 = h(o_0, W_1) \]

\[ L = f(g(h(o_0, W_1), W_2)) \]

\[ \frac{\partial L}{\partial W_1} \]

\[ \frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial o_2} \cdot \frac{\partial o_2}{\partial W_2} \]

\( W_1 \)

\( W_2 \)
3 channels (RGB)
shape (3, 244, 244)

\[ \mathbf{O}_0 \quad (64, 55, 55) \]

\[ \mathbf{O}_1 \quad (192, 27, 27) \]

predictions (1, 10)

\[
\begin{align*}
\mathbf{L} &= f(\mathbf{o}_2) \\
\mathbf{o}_2 &= g(\mathbf{o}_1, \mathbf{W}_2) \\
\mathbf{L} &= f(g(\mathbf{o}_1, \mathbf{W}_2)) \\
\mathbf{o}_1 &= h(\mathbf{o}_0, \mathbf{W}_1) \\
\mathbf{L} &= f(g(h(\mathbf{o}_0, \mathbf{W}_1), \mathbf{W}_2))
\end{align*}
\]

\[
\begin{align*}
\frac{\partial \mathbf{L}}{\partial \mathbf{W}_1} &= \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \cdot \frac{\partial \mathbf{o}_2}{\partial \mathbf{o}_1} \cdot \frac{\partial \mathbf{o}_1}{\partial \mathbf{W}_1} \\
\frac{\partial \mathbf{L}}{\partial \mathbf{W}_2} &= \frac{\partial \mathbf{L}}{\partial \mathbf{o}_2} \cdot \frac{\partial \mathbf{o}_2}{\partial \mathbf{W}_2}
\end{align*}
\]
3 channels (RGB)
shape (3, 244, 244)

\[ O_0 \] (64, 55, 55)
\[ O_1 \] (192, 27, 27)
predictions (1, 10)

\[ \frac{\partial L}{\partial W_0} = \frac{\partial L}{\partial o_2} \cdot \frac{\partial o_2}{\partial o_1} \cdot \frac{\partial o_1}{\partial W_1} \cdot \frac{\partial o_0}{\partial W_0} \]

\[ \frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial o_2} \cdot \frac{\partial o_1}{\partial W_1} \]

\[ \frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial o_2} \cdot \frac{\partial o_2}{\partial W_2} \]

loss
In the context of training and validation, it is crucial to stop training at the point where training loss begins to rise on the validation loss curve. This point is referred to as overfitting, where the model performs well on the training data but poorly on unseen data. It is important to monitor both training and validation losses to ensure the model generalizes well to new data.
Applications
Image Classification

Image -> ConvNet -> Representation -> Linear Classifier -> Class Labels
Semantic Segmentation

Image -> ConvNet -> Representation -> Per-Pixel Class Probabilities
Object Detection

Image → ConvNet → Representation

- [x, y, width, height]
- confidence
- class label
Reinforcement Learning

Image → ConvNet → Representation → Distribution over actions
What is your task?
Fine Tuning

Freeze early layer in ConvNet (use as fixed feature extractor). Re-initialise last layer(s) and only train them.
Tips and Tricks

http://karpathy.github.io/2019/04/25/recipe/
Deep Learning for Robotic Vision

An Introduction

Niko Suenderhauf

Queensland University of Technology
Australian Centre for Robotic Vision