Lecture: Deep Learning

Juan Carlos Niebles and Ranjay Krishna
Stanford Vision and Learning Lab
## CS 131 Roadmap

<table>
<thead>
<tr>
<th>Pixels</th>
<th>Segments</th>
<th>Images</th>
<th>Videos</th>
<th>Web</th>
</tr>
</thead>
<tbody>
<tr>
<td>Convolutions</td>
<td>Resizing</td>
<td>Recognition</td>
<td>Motion</td>
<td>Neural networks</td>
</tr>
<tr>
<td>Edges</td>
<td>Segmentation</td>
<td>Detection</td>
<td>Tracking</td>
<td>Convolutional neural networks</td>
</tr>
<tr>
<td>Descriptors</td>
<td>Clustering</td>
<td>Ontology</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Today's agenda

• Perceptron
• Linear classifier
• Loss function
• Gradient descent and backpropagation
• Neural networks
Today's agenda

• Perceptron
  • Linear classifier
  • Loss function
  • Gradient descent and backpropagation
  • Neural networks
1950s Age of the Perceptron
1957 The Perceptron (Rosenblatt)
1969 Perceptrons (Minsky, Papert)

1980s Age of the Neural Network
1986 Back propagation (Hinton)
1990s Age of the Graphical Model
2000s Age of the Support Vector Machine

2010s Age of the Deep Network
deep learning = known algorithms + computing power + big data
Learning representations by back-propagating errors

David E. Rumelhart*, Geoffrey E. Hinton†
& Ronald J. Williams*

* Institute for Cognitive Science, C-015, University of California, San Diego, La Jolla, California 92093, USA
† Department of Computer Science, Carnegie-Mellon University, Pittsburgh, Philadelphia 15213, USA

We describe a new learning procedure, back-propagation, for networks of neurone-like units. The procedure repeatedly adjusts the weights of the connections in the network so as to minimize a measure of the difference between the actual output vector of the net and the desired output vector. As a result of the weight adjustments, internal 'hidden' units which are not part of the input or output come to represent important features of the task domain, and the regularities in the task are captured by the interactions of these units. The ability to create useful new features distinguishes back-propagation from earlier, simpler methods such as the perceptron-convergence procedure.

There have been many attempts to design self-organizing neural networks. The aim is to find a powerful synaptic modification rule that will allow an arbitrarily connected neural network to develop an internal structure that is appropriate for a particular task domain. The task is specified by giving the desired state vector of the output units for each state vector of the input units. If the input units are directly connected to the output units it is relatively easy to find learning rules that iteratively adjust the relative strengths of the connections so as to progressively reduce the difference between the actual and desired states. This is not so easy when we introduce hidden units whose actual or desired states are not specified by the task. (In perceptrons, there are 'feature analysers' between the input and output that are not true hidden units because their input connections are fixed by hand, so their states are completely determined by the input vector: they do not learn representations.) The learning procedure must decide under what circumstances the hidden units should be active in order to help achieve the desired input-output behaviour. This amounts to deciding what these units should represent. We demonstrate that a general purpose and relatively simple procedure is powerful enough to construct appropriate internal representations.

The simplest form of the learning procedure is for layered networks which have a layer of input units at the bottom; any number of intermediate layers; and a layer of output units at the top. Connections within a layer or from higher to lower layers are forbidden, but connections can skip intermediate layers. An input vector is presented to the network by setting the states of the input units. Then the states of the units in each layer are determined by applying equations (1) and (2) to the connections coming from lower layers. All units within a layer have their states set in parallel, but different layers have their states set sequentially, starting at the bottom and working upwards until the states of the output units are determined.

The total input, \( x_j \), to unit \( j \) is a linear function of the outputs, \( y_i \), of the units that are connected to \( j \) and of the weights, \( w_{ji} \), on these connections

\[
x_j = \sum_i y_i w_{ji}
\]

Units can be given biases by introducing an extra input to each unit which always has a value of 1. The weight on this extra input is called the bias and is equivalent to a threshold of the opposite sign. It can be treated just like the other weights.

A unit has a real-valued output, \( y_j \), which is a non-linear function of the weighted sum of inputs. At each step in the iteration, \( y_j \) is updated using a non-linear function of the weighted sum of inputs. The choice of function used for computing \( y_j \) depends on the network application.
Perceptron

inputs $x_1$, $x_2$, $x_3$, ..., $x_N$

weights $w_1$, $w_2$, $w_3$, ..., $w_N$

sum

output $y$

sign function $f$
Aside: Inspiration from Biology

Neural nets/perceptrons are loosely inspired by biology.

But they are NOT how the brain works, or even how neurons work.
Perceptron: for image classification

\[ w_1, \ w_2, \ w_3, \ldots, \ w_N \]

weights

sum

sign function

\[ x_1, \ x_2, \ x_3, \ldots, \ x_N \]

\[ y \]

dog
Recall: image classification pipeline
Recall: we can featurize images into a vector

- Raw pixels
- Raw pixels + (x,y)
- PCA
- LDA
- BoW
- BoW + spatial pyramids
Recall: we can featurize images into a vector

Image Vector

\[ x_1 \]
\[ x_2 \]
\[ x_3 \]
\[ \ldots \]
\[ \ldots \]
\[ \ldots \]
\[ \ldots \]
\[ \ldots \]
\[ \ldots \]
\[ \ldots \]
\[ x_n \]

Raw pixels
Raw pixels + (x,y)
PCA
LDA
BoW
BoW + spatial pyramids
Perceptron: for image classification

\[ y = \text{sign} \left( \sum_{i=1}^{N} w_i x_i \right) \]

where \( w_i \) are the weights, \( x_i \) are the inputs, and \( y \) is the output.
Perceptron: simplified view with one perceptron
Perceptron: simplified view with two perceptron

$x_1$

$x_2$

$x_3$

\vdots

$x_N$

cat

dog
Linear classifier as a set of perceptrons

$$x_1$$

$$x_2$$

$$x_3$$

$$\vdots$$

$$x_N$$

cat
dog

bird
Today's agenda

• Perceptron
• Linear classifier
• Loss function
• Gradient descent and backpropagation
• Neural networks
Linear classifier: input layer
Linear classifier: output layer

'input' layer

'output' layer

\( x_1 \)
\( x_2 \)
\( x_3 \)
\( \vdots \)
\( x_N \)

\( \text{cat} \)
\( \text{dog} \)
\( \text{bird} \)
Linear classifier: mathematical formulation

\[ f(x, W) \]

Weights or Parameters

10 numbers giving class scores
Linear classifier: mathematical formulation

\[ f(x, W) = Wx \]

\[ x = 3072 \times 1 \]

\[ W = ? \]

10 numbers giving class scores

\( W \)

Weights or Parameters

(32x32x3)

3072 dimensional vector
Linear classifier: mathematical formulation

\[ f(x, W) = Wx \]

\[ x = 3072 \times 1 \]

\[ W = 10 \times 3072 \]

Weights or Parameters

10 numbers giving class scores

(32x32x3) 3072 dimensional vector
Linear classifier: function visualized

Image Vector \times \text{dog Weight Vector} = \text{“probability” of the image being an dog}
Linear classifier: function visualized

Image Vector

“probability” of the image being an cat
Linear classifier: function visualized

bird Weight Vector × Image Vector = “probability” of the image being a bird
Linear classifier: function visualized

Truck Weight Vector

Image Vector

“probability” of the image being a truck
Linear classifier: function visualized

\[ X \text{ Weight Matrix} = \text{Image Vector} \]
Linear classifier: bias vector

Weight Matrix

Image Vector

Bias Vector

X =
Linear classifier: size

10x3072
Weight Matrix

3072x1
Image Vector

3072x1
Bias Vector

=
Linear classifier: Making a classification

\[ \text{argmax} \]

234
94
-104
22
1
27.3
-13
-123
99
-82

dog
Interpreting the weights

• Assume our weights are trained on the CIFAR 10 dataset:
Interpreting the weights as templates

Let us look at each row of the weight matrix
Interpreting the weights as templates

We can reshape the vector back into the shape of an image

Diagram:
- Original vector: 1x3072
- Reshaped vector: 32x32x3
Let's visualize what the templates look like

We can reshape the row back to the shape of an image
Interpreting the weights geometrically

- Assume the image vectors are in 2D space to make it easier to visualize.

Plot created using Wolfram Cloud
Today's agenda

• Perceptron
• Linear classifier
• Loss function
• Gradient descent and backpropagation
• Neural networks
Training linear classifiers

We need to learn how to pick the weights in the first place.

Formally, we need to find $W$ such that

$$\min_W \text{Loss}(x, y, \hat{y})$$

where $x$ is the image, $y$ is the true label, $\hat{y}$ is the model's predicted label.

All we have to do is define a loss function!
Given training data:

\[ y = wx \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10.1</td>
</tr>
<tr>
<td>2</td>
<td>1.9</td>
</tr>
<tr>
<td>3.5</td>
<td>3.4</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
</tr>
</tbody>
</table>

What do you think is a good approximation weight parameter for this data point?
Given training data:

\[ y = wx \]

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>10.1</td>
<td>4</td>
<td>20.1</td>
</tr>
<tr>
<td>2</td>
<td>1.9</td>
<td>2</td>
<td>13.1</td>
</tr>
<tr>
<td>3.5</td>
<td>3.4</td>
<td>3.5</td>
<td>2.4</td>
</tr>
<tr>
<td>1</td>
<td>1.1</td>
<td>1</td>
<td>0.1</td>
</tr>
</tbody>
</table>

What about this one?
Properties of a loss function

Given several examples

\[
\{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\}
\]

and a perceptron

\[
\hat{y} = wx
\]

where \(x_i\) is image and \(y_i\) is (integer) label
(0 for dog, 1 for cat, etc.)

A loss function \(L_i(x_i, y_i, \hat{y}_i)\) tells how good our current classifier
- When the classifier predicts correctly \((y_i = \hat{y})\), the loss should be low
- When the classifier makes mistakes \((y_i \neq \hat{y})\), the loss should be high
Properties of a loss function

Given several examples

\[ \{(x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N)\} \]

and a perceptron

\[ \hat{y} = wx \]

where \( x_i \) is image and \( y_i \) is (integer) label
(0 for dog, 1 for cat, etc.)

Loss over the entire dataset is an average of loss over examples:

\[ L = \frac{1}{N} \sum_{i=1}^{N} L_i(y_i, \hat{y}_i) \]
How do we choose $L_i$?

YOU get to chose the loss function!
(some are better than others depending on what you want to do)
Squared Error (L2)
(a popular loss function) ((why?))

\[ L_i(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2 \]

Not robust to outliers
$L_1$ loss

$L_i(y_i, \hat{y}_i) = |y_i - \hat{y}_i|$
L1 Loss
\[ L_i(y_i, \hat{y}_i) = |y_i - \hat{y}_i| \]

L2 Loss
\[ L_i(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2 \]

Zero-One Loss
\[ L_i(y_i, \hat{y}_i) = 1|y_i = \hat{y}_i| \]

Hinge Loss
\[ L_i(y_i, \hat{y}_i) = \max(0, 1 - y_i \hat{y}_i) \]
Softmax Classifier (Multinomial Logistic Regression)

• It allows us to treat the outputs of a model as probabilities for each class.
• Common way of measuring distance between probability distributions is KL divergence.
\[ D_{KL} = \sum_y P(y) \log \frac{P(y)}{Q(y)} \]

• where \( P \) is the ground truth distribution
• and \( Q \) is the model's output score distribution
Softmax Classifier (Multinomial Logistic Regression)

• KL divergence.

\[ D_{KL} = \sum_y P(y) \log \frac{P(y)}{Q(y)} \]

• In our case, \( P \) is only non-zero for the correct class.

• For example, consider the case where we only have 3 classes:

\[
\begin{bmatrix}
1 \\
0 \\
0
\end{bmatrix}
\]
Softmax Classifier (Multinomial Logistic Regression)

- KL divergence.

\[
D_{KL} = \sum_y P(y) \log \frac{P(y)}{Q(y)}
\]

\[
= - \log Q(y)
\]

\[
= - \log \text{Prob}[f(x_i, W) == y_i]
\]

Correct outputs:

\[
\begin{bmatrix}
1 \\
0 \\
0
\end{bmatrix}
\]

- dog
- cat
- bird
Softmax Classifier (Multinomial Logistic Regression)

\[ L_i = - \log Prob[f(x_i, W) == y_i] \]

Remember our linear classifier:

\[ \hat{y} = wx \]

there are no limits on the output space. Meaning that the model can generate outputs > 1 or < 0.

model outputs

\[
\begin{bmatrix}
3.2 \\
5.1 \\
-1.7
\end{bmatrix}
\]

correct outputs

\[
\begin{bmatrix}
1 \\
0 \\
0
\end{bmatrix}
\]

dog

cat

bird
**Softmax Classifier** (Multinomial Logistic Regression)

\[ L_i = - \log \text{Prob} [f(x_i, W) == y_i] \]

We need a mechanism to convert or normalize the outputs into probability ranges [0,1].

Solution: SOFTMAX: \( \text{Prob}[f(x_i, W) == k] = \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}} \)

**Example:**

- Model outputs: \( \begin{bmatrix} 3.2 \\ 5.1 \\ -1.7 \end{bmatrix} \)
- Correct outputs: \( \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \)

**Classes:** dog, cat, bird
**Softmax Classifier** (Multinomial Logistic Regression)

\[ L_i = -\log \text{Prob}[f(x_i, W) == y_i] \]

We need a mechanism to convert or normalize the outputs into probability ranges [0,1].

Solution: SOFTMAX: \[ \text{Prob}[f(x_i, W) == k] = \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}} \]
Softmax Classifier (Multinomial Logistic Regression)

\[ L_i = - \log \text{Prob}[f(x_i, W) == y_i] \]

In this case, what is the loss:

\[ L_i = ??? \]
Softmax Classifier (Multinomial Logistic Regression)

\[ L_i = -\log \text{Prob}[f(x_i, W) == y_i] \]

In this case, what is the loss:

\[ L_i = -\log(0.13) = 2.04 \]
Softmax Classifier (Multinomial Logistic Regression)

\[ L_i = - \log Prob[f(x_i, W) == y_i] \]

What is the minimum and maximum values that the loss can be?
Softmax Classifier (Multinomial Logistic Regression)

\[ L_i = -\log \text{Prob}[f(x_i, W) == y_i] \]

At initialization, all the weights will be random. In this case, we can assume that the outputs will have the same probability. In this case, what will the initial loss be?
Softmax Classifier (Multinomial Logistic Regression)

\[ L_i = -\log \text{Prob}[f(x_i, W) == y_i] \]

At initialization, all the weights will be random. In this case, we can assume that the outputs will have the same probability. In this case, what will the initial loss be?

\[ L_i = -\log \left( \frac{1}{C} \right) = \log(C) = \log(10) = 2.03 \]
Today's agenda

- Perceptron
- Linear classifier
- Loss function
- Gradient descent and backpropagation
- Neural networks
Gradient descent visualized: Minimizing loss
Gradient descent visualized: Minimizing loss
Gradient descent visualized: Minimizing loss
Gradient descent visualized: Minimizing loss
Gradient descent visualized: Minimizing loss
Gradient descent visualized: Minimizing loss
Gradient descent visualized: Minimizing loss
Gradient descent visualized: Minimizing loss
Gradient Descent Pseudocode

for i in {0,...,num_epochs}:
  for x, y in data:
    \(\hat{y} = f(x, W)\)
    \(L = \sum_i L_i (y_i, \hat{y}_i)\)
    \(\frac{dL}{dW} = ???\)
    \(W := W - \alpha \frac{dL}{dW}\)
Partial derivative of loss to update weights

Given training data point \((x, y)\), the linear classifier formula is: \(\hat{y} = Wx\)

Let's assume that the correct label is class \(k\), implying \(y = k\)

\[
\text{Loss} = L(\hat{y}, y) = -\log \frac{e^{\hat{y}_k}}{\sum_j e^{\hat{y}_j}}
\]

\[
= -\hat{y}_k + \log \sum_j e^{\hat{y}_j}
\]

Calculating the loss \(\frac{dL}{dW}\) is hard mathematically. But we can use the chain rule to make it simpler:

\[
\frac{dL}{dW} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dW}
\]
Partial derivative of loss to update weights

Given training data point \((x, y)\), the linear classifier formula is: \(\hat{y} = Wx\)

Let's assume that the correct label is class \(k\), implying \(y = k\)

\[
\text{Loss} = -\hat{y}_k + \log \sum_j e^{\hat{y}_j}
\]

Now, we want to update the weights \(W\) by calculating the direction in which to change the weights to reduce the loss:

\[
\frac{dL}{dW} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dW}
\]

We know that: \(\frac{d\hat{y}}{dW} = x\) but what about \(\frac{dL}{d\hat{y}}\)?
Partial derivative of loss to update weights

\[ L = -\hat{y}_k + \log \sum_j e^{\hat{y}_j} \]

To calculate \( \frac{dL}{d\hat{y}} \), we need to consider two cases:

Case 1: \( \hat{y} = k \)

\[ \frac{dL}{d\hat{y}} = -1 + \frac{e^{\hat{y}_k}}{\log \sum_j e^{\hat{y}_j}} \]

Case 2: \( \hat{y} = l \neq k \)

\[ \frac{dL}{d\hat{y}} = \frac{e^{\hat{y}_l}}{\log \sum_j e^{\hat{y}_j}} \]
Partial derivative of loss to update weights

Putting it all together:

\[
\frac{dL}{dW} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dW}
\]

\[
\frac{dL}{dW} = \begin{bmatrix}
\frac{e^{\hat{y}_0}}{\log \sum_j e^{\hat{y}_j}} \\
\frac{e^{\hat{y}_1}}{\log \sum_j e^{\hat{y}_j}} \\
\vdots \\
\frac{e^{\hat{y}_k}}{\log \sum_j e^{\hat{y}_j}} \\
\frac{e^{\hat{y}_{3071}}}{\log \sum_j e^{\hat{y}_j}} \\
\end{bmatrix} x
\]
Gradient Descent Pseudocode

for i in {0,...,num_epochs}:
    for x, y in data:
        \( \hat{y} = f(x, W) \)
        \( L = \sum_i L_i (y_i, \hat{y}_i) \)
        \( \frac{dL}{dW} = \text{We know how to calculate this now!} \)
        \( W := W - \alpha \frac{dL}{dW} \)
Backprop – another way of computing gradients

\[ \hat{y} = Wx \]

\[ \frac{dL}{dW} = \frac{dL}{dz} \frac{dz}{dW} \]
Backprop - another way of computing gradients

\[
\hat{y} = Wx \\
L = \text{Loss}(\hat{y}, y)
\]

\[
\frac{dL}{dW} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dW}
\]

**Key Insight:**
- visualize the computation as a graph
- Compute the forward pass to calculate the loss.
- Compute all gradients for each computation backwards
Backprop example in 1D:

\[ W = 1 \]

\[ \frac{d\hat{y}}{dW} = ? \]

\[ x = 2 \]

\[ \frac{dL}{d\hat{y}} = 1.2 \]

\[ \hat{y} = 2 \]

\[ \text{Loss} \]

\[ y = 0 \]

We know that:

\[ \frac{dL}{dW} = \frac{dL}{d\hat{y}} \frac{d\hat{y}}{dW} \]

\[ = \frac{dL}{d\hat{y}} x \]

\[ = 1.2 x \]

\[ = 1.2 \times 2 \]

\[ = 2.4 \]
Interpreting the weights geometrically

• Assume the image vectors are in 2D space to make it easier to visualize.

• Let’s start with one class: dog.

• Initialize the weights randomly

• What is the bias vector here?
Interpreting the weights geometrically

- Assume the image vectors are in 2D space to make it easier to visualize.
- Let's start with one class: dog.
- Initialize the weights randomly
- Now let's add two data points
Interpreting the weights geometrically

- Assume the image vectors are in 2D space to make it easier to visualize.
- Let's start with one class: dog.
- Initialize the weights randomly
- Now let's add two data points
- Update the weights
Interpreting the weights geometrically

- Assume the image vectors are in 2D space to make it easier to visualize.
- Let's start with one class: dog.
- Initialize the weights randomly
- Now let's add two more data points
- Update the weights
Interpreting the weights **geometrically**

- Assume the image vectors are in 2D space to make it easier to visualize.
- Let's start with one class: *dog*.
- Initialize the weights randomly.
- Now let's add two more data points.
- Update the weights.
Interpreting the weights geometrically

Plot created using Wolfram Cloud
Recall: image classification pipeline

Training Images

- airplane
- automobile
- bird
- cat
- deer
- dog
- frog
- horse
- ship
- truck

Test Image

- Image Features
- Training
- Learned Classifier
- Prediction
Recall: we can **featurize** images into a vector

- Raw pixels
- Raw pixels + (x,y)
- PCA
- LDA
- BoW
- BoW + spatial pyramids
Features sometimes might not be linearly separable

\[ f(x, y) = (r(x, y), \theta(x, y)) \]
Remember our linear classifier
Let's change the features by adding another layer
2-layer network: mathematical formula

- linear classifier:
  \[ y = W x \]

- 2-layer network:
  \[ y = W_2 \max(0, W_1 x) \]

- 3-layer network:
  \[ y = W_3 \max(0, W_2 \max(0, W_1 x)) \]

Choosing the number of layers is a new hyperparameter!
Today's agenda

- Perceptron
- Linear classifier
- Loss function
- Gradient descent and backpropagation
- Neural networks
2-layer network: mathematical formula

• linear classifier:
  \[ y = Wx \]

• 2-layer network:
  \[ y = W_2 \max(0, W_1 x) \]

We know the size of \( x = 1 \times 3072 \) and \( y = 10 \times 1 \)

So what are \( W_1 \) and \( W_2 \)

We know that they must be: \( W_1 = h \times 3072 \) and \( W_2 = 10 \times h \)

\( h \) is a new hyperparameter!
2-layer network: mathematical formula

- linear classifier:
  \[ y = Wx \]

- 2-layer network:
  \[ y = W_2 \max(0, W_1 x) \]

Why is the \( \max(0, _) \) necessary? Let's see what happens when we remove it:

\[ y = W_2 W_1 x = Wx \]

where

\[ W = W_2 W_1 \]
Activation function

The non-linear max function allows models to learn more complex transformations for features.

Choosing the right activation function is another new hyperparameter!

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
\[ \begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases} \]
Recall: image classification pipeline

Training Images

- airplane
- automobile
- bird
- cat
- deer
- dog
- frog
- horse
- ship
- truck

Test Image

Image Features

- Learned Classifier

Prediction

Training Images

Image Features

Training

Labels
2-layer neural network performance

• ~40% accuracy on CIFAR-10 test
  – Best class: Truck (~60%)
  – Worst class: Horse (~16%)

• Check out the model at: https://tinyurl.com/cifar10
Next Time…

Backpropagation with deep neural networks

Convolutional neural networks

Designing architectures