CS 189/289

Today’s lecture:
1. From logistic to softmax.
2. Convolutional neural networks
3. Residual neural networks (resnets)
CS 189/289

Today’s lecture:
1. From logistic to softmax.
2. Convolutional neural networks
3. Residual neural networks (resnets)
Recall: **logistic loss for binary classification**

Neural networks can be modeled by logistics

Standard Trick: Add a 0th component to the $x_i$ vector, which is fixed to be 1. This is connected with weight $a$

Modeling the probability distribution

We say that the class label $Y$ is a Bernoulli random variable, with its probability parameter $p$ being as above

$$P(Y=1|X) = \frac{1}{1 + e^{-(\beta^T x)}}$$

For compactness, introduce notation $\mu(x) = \frac{1}{1 + e^{-(\beta^T x)}}$ and $\alpha \mu(x) = \frac{1}{1 + e^{-(\beta^T x)}}$

As usual we use $y$ to denote values taken by random variables

$$P(y|X) = \mu(x)^y(1-\mu(x))^{1-y}$$

What if we have more than 2 classes?
From logistic regression to softmax regression

\[
\begin{bmatrix}
    p(Y = 1|X) \\ p(Y = 0|X)
\end{bmatrix} = \begin{bmatrix}
    \mu \\ 1-\mu
\end{bmatrix} = \begin{bmatrix}
    \frac{1}{1 + \exp(-\beta x)} \\ \frac{\exp(-\beta x)}{1 + \exp(-\beta x)}
\end{bmatrix}
\]
From logistic regression to softmax regression

\[ \begin{bmatrix} p(Y = 1|X) \\ p(Y = 0|X) \end{bmatrix} = \begin{bmatrix} \mu \\ 1 - \mu \end{bmatrix} = \begin{bmatrix} \frac{1}{1 + e^{-\beta x}} \\ \frac{e^{-\beta x}}{1 + e^{-\beta x}} \end{bmatrix} \]

Instead we could write this as

\[ \begin{bmatrix} e^{\beta_1 x} \\ e^{\beta_2 x} \end{bmatrix} = \begin{bmatrix} 1 \\ 1 + e^{\beta_2 x - \beta_1 x} \end{bmatrix} \]

Equivalent with \( \beta = \beta_1 - \beta_2 \)
The softmax function for K-class classification

\[
\begin{bmatrix}
p(Y = 1|X) \\
p(Y = 2|X) \\
p(Y = 3|X) \\
\vdots \\
p(Y = K|X)
\end{bmatrix} = \frac{1}{\sum_{i=1}^{K} e^{\beta_i x}} \begin{bmatrix} e^{\beta_1 x} \\
 e^{\beta_2 x} \\
 e^{\beta_3 x} \\
\vdots \\
 e^{\beta_K x} \end{bmatrix}
\]

- Generalization of logistic regression to more than 2 classes.
- “Softmax regression” or “multinomial logistic regression”, parameters \( \beta \).
- Use principle of MLE to set \( \beta \).
- Needs iterative optimization like gradient descent.
- Can also stick at the top of neural network to get a “softmax” loss.
The softmax function for K-class classification

\[
\begin{bmatrix}
-p(Y = 1|X) \\
p(Y = 2|X) \\
p(Y = 3|X) \\
\vdots \\
p(Y = K|X)
\end{bmatrix} = \frac{1}{\sum_{i=1}^{K} e^{\beta_i x}}
\]

For class `i`, the logit (log-odds) is defined as:

\[
\text{logit}_i = \log \left( \frac{P(y=i|x)}{P(y\neq i|x)} \right)
\]

For class `i`, the softmax function is defined as:

\[
P(y = i|x) = \frac{e^{\text{logit}_i}}{\sum_{j=1}^{K} e^{\text{logit}_j}}
\]
CS 189/289

Today’s lecture outline:
1. From logistic to softmax.
2. Convolutional neural networks
3. Residual neural networks (resnets)
Recall: *fully connected* neural networks

\[ O_i = g \left( \sum_j W_{ij} g \left( \sum_k W_{jk} x_k \right) \right) \]

Back-propagation algorithm to compute derivatives of the parameters efficiently.
Beyond fully connected, feed-forward architectures:

1. Convolutional
2. Residual
3. Recurrent (not “feed-forward”).
4. Attention and Transformers.
5. Graph

• As long as we have a feed-forward network, and use only differentiable components, we can apply backprop.
• New architectures have led to breakthrough successes.
Pondering fully connected neural networks

- For “fully connected” (FC) layer, $l$, with $n_i(l)$ inputs and $n_o(l)$ outputs, $W_l$ contains $n_i(l) \times n_o(l)$ parameters.
- Adds up quickly to huge #s of parameters.
- Too many parameters can contribute to problems of “overfitting”.

![Diagram of fully connected neural networks]

\[ W_1, W_2, W_3 \]
Pondering fully connected neural networks

- For “fully connected” (FC) layer, \( l \), with \( n_i(l) \) inputs and \( n_o(l) \) outputs, \( W_l \) contains \( n_i(l) \times n_o(l) \) parameters.
- Adds up quickly to huge #s of parameters.
- Too many parameters can contribute to problems of “overfitting”.

➤ Strategy to reduce # of free parameters: “bake” in properties that encode problem symmetries.
Examples of common problem symmetries

**translation invariance**

- Input: $I$
- Output: $f(I)$
- Predict: is a cat vs. not a cat

**translation equivariance**

- Input: $I$
- Output: $f(S(I))$
- Predict: which pixels are cat pixels?

https://www.doc.ic.ac.uk/~bkainz/teaching/DL/notes/equivariance.pdf
Examples of common problem symmetries

**permutation invariance**

\[ f([\text{A}, \text{B}, \text{C}]) = [\text{apple}, \text{banana}] \]

\[ f([\text{A}, \text{B}, \text{C}]) = [\text{apple}, \text{banana}] \]

\[ \vdots \]

\[ f([\text{C}, \text{B}, \text{A}]) = [\text{apple}, \text{banana}] \]

**permutation equivariance**

\[ \text{Perm}(f([\text{A}, \text{B}, \text{C}])) = [\text{apple}, \text{banana}] \]

\[ \text{Perm}(f([\text{A}, \text{B}, \text{C}])) = [\text{apple}, \text{banana}] \]

\[ \vdots \]

\[ \text{Perm}(f([\text{C}, \text{B}, \text{A}])) = [\text{apple}, \text{banana}] \]

\[
\begin{align*}
  f(x) &= f(\text{Perm}(x)) \\
  \text{Perm}(f(x)) &= f(\text{Perm}(x))
\end{align*}
\]

Predict vector output.

https://github.com/yoheikikuta/paper-reading/issues/6
Examples of common problem symmetries

rotation invariance

rotation equivariance

predict phase (is liquid?)
at room temperature

predict forces (vector)

[from David Rothchild]
Examples of common problem symmetries

**translation invariance**

**translation equivariance**

- The *convolution* operation is translation equivariant.
- This operation will form the basis of *convolutional neural networks* (CNNs).
- CNNs also be motivated by the idea of learning re-usable features (next).

Predict: is a cat vs. not a cat

Predict: which pixels are cat pixels?
Features sharing across one input example

“Features” (e.g. is there an eye here?) constructed in fully connected layer cannot be shared across the input (e.g. image), because $w$ is not reused across the image.

One neuron in FC layer:

\[ xw = y \]

$x$ is the cat matrix flattened to a 1D vector

$w$ operates on the entire image
Features sharing across one input example

"Features" (e.g. is there an eye here?) constructed in fully connected layer cannot be shared across the input (e.g. image), because $w$ is not reused across the image.

- ConvNet: learn shared features that are applied to every image patch.
- Also gives us *translational equivariance* for each filter ($w$) response.
Fully Connected (FC): no feature sharing

- Uses “global template matching”.
- e.g. one $W$ matrix per class (single layer):

Iteration 1 of training:

From Stella Yu

https://chatbotslife.com/training-mxnet-part-1-mnist-6f0dc4210c62
Fully Connected (FC): no feature sharing

- Uses “global template matching”.
- e.g. one $W$ matrix per class (single layer):

**Iteration 2 of training:**

From Stella Yu

https://chatbotslife.com/training-mxnet-part-1-mnist-6f0dc4210c62
Fully Connected (FC): no feature sharing

- Uses “global template matching”.
- e.g. one $W$ matrix per class (single layer):

Iteration 3 of training:

From Stella Yu

https://chatbotslife.com/training-mxnet-part-1-mnist-6f0dc4210c62
Fully Connected (FC): no feature sharing

- Uses “global template matching”.
- e.g. one $W$ matrix per class (single layer):

Iteration 7 of training:

From Stella Yu

https://chatbotslife.com/training-mxnet-part-1-mnist-6f0dc4210c62
What would re-usable features look like?

- What if we could learn "local feature filters"
- Then on the next layer, learn how to combine them?

https://datascience.stackexchange.com/questions/16463/what-is-are-the-default-filters-used-by-keras-convolution2d
Convolutional NNs (CNNs/"Convnets")

Can view CNNs as a way to construct hierarchical features, each of which get combined at the next level.

[Diagram showing hierarchical feature construction in CNNs]

Lee et al. “Convolutional DBN's ...” ICML 2009

https://www.slideshare.net/milkers/lecture-06-marco-aurelio-ranzato-deep-learning
Convolutional NNs (CNNs/“Convnets”)

Can view CNNs as a way to construct hierarchical features, each of which get combined at the next level.

https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/
Convolutional NNs (CNNs/"Convnets")

With Conv layer:

Now w takes (overlapping) patches

"1D" Conv.

https://ujjwalkarn.me/2016/08/11/intuitive-explanation-convnets/
(2D) Convolution

Convolve one learned "filter", $W$ with the input to get convolution output $\{v_{ij}\}$:

For each position, $i, j$:
1. Element-wise product of $W$ with image patch centered on $i, j$ (e.g. $3 \times 3$).
2. Sum up the results to get one $v_{ij}$.

$W$ called filter/template/kernel

https://github.com/vdumoulin/conv_arithmetic
Convoluotional NNs (CNNs/"Convnets")

- We will actually use multiple feature maps, \( \{W_k\}_{k=1}^K \).
- "Depth" of output "volume" is \( K \):

\[
\begin{align*}
\text{Non-linearity to get hidden node in a hidden layer in CNN}
\end{align*}
\]
Formally: 1D convolution

- For n-dim convolution, we use an n-dim filter.
- So 1D convolution has a 1D filter.

If $a$ and $b$ are two arrays,

$$(a * b)_t = \sum_{\tau \in [0,1,2,\ldots]} a_\tau b_{t-\tau}$$

$t$'th element of the convolution

- $\tau$ is the index of the filter element ('-' means flip filter first)
- Invalid indices, e.g., $t = 1,2,3$ and $\tau = 3$, are boundaries; don't compute those $t^{th}$ entries, or else pad out e.g. with zeros/mirroring input.
- No padding, size of output is $D - K + 1$ for $D$ length input, $K$ length filter.

Cross-correlation: 

$$(a \otimes b)_t = \sum_{\tau} a_\tau b_{t+\tau}$$
1D convolution

\[(a * b)_t = \sum_{\tau} a_{\tau} b_{t-\tau}. \]

Method 1: flip-and-filter

\[
\begin{array}{c}
\text{2} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{-1} \\
\downarrow \\
\text{2} \\
\downarrow \\
\end{array} \quad \star \quad 
\begin{array}{c}
\text{2} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\end{array} \\
= \\
\begin{array}{c}
\text{2} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\end{array} \\
\begin{array}{c}
\text{2} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\end{array} \\
\begin{array}{c}
\text{2} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\end{array} \\
\begin{array}{c}
\text{2} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\end{array} \\
\begin{array}{c}
\text{2} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\text{1} \\
\downarrow \\
\end{array}
\]
1D convolution

Method 2: translate-and-scale

\[(a \ast b)_t = \sum_{\tau} a_{\tau} b_{t-\tau} = \]

\[
\begin{array}{c}
\begin{array}{c}
2 \\
\downarrow \ \\
-1
\end{array}
\end{array}
\ast
\begin{array}{c}
\begin{array}{ccc}
1 & 1 & 2 \\
\uparrow & \uparrow & \uparrow
\end{array}
\end{array}
= 2 \times
\begin{array}{c}
\begin{array}{c}
1 \\
\uparrow
\end{array}
\end{array}
\]

\[
+ \begin{array}{c}
\begin{array}{c}
1 \\
\uparrow
\end{array}
\end{array}
\]

\[
= \begin{array}{c}
\begin{array}{c}
4 \\
\uparrow
\end{array}
\end{array}
+ \begin{array}{c}
\begin{array}{c}
2 \\
\uparrow
\end{array}
\end{array}\]
1D convolution

Method 3

Convolution can also be viewed as matrix multiplication:

\[(a * b)_t = \sum_{\tau} a_{\tau} b_{t-\tau} = (2, -1, 1) \ast (1, 1, 2) = \begin{pmatrix} W_k \\ 1 \\ 1 \\ 1 \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ -1 \\ 1 \\ 1 \\ 2 \end{pmatrix} = \]

\[= \]

\[W_k \] has size \(5 \times 3\), which means it has 15 entries, yet there are only 3 parameters. Why Convnets to have relatively few parameters!
From 1D to 2D convolution

Method 1: Flip-and-Filter

\[
(A \ast B)_{ij} = \sum_s \sum_t A_{st} B_{i-s,j-t}
\]
From 1D to 2D convolution

\[(A \ast B)_{ij} = \sum_s \sum_t A_{st} B_{i-s,j-t}\]

**Method 2: Translate-and-Scale**

\[
\begin{array}{ccc}
1 & 3 & 1 \\
0 & -1 & 1 \\
2 & 2 & -1 \\
\end{array}
\times
\begin{array}{ccc}
1 & 2 \\
0 & -1 \\
\end{array}
= + \begin{array}{ccc}
1 & 3 & 1 \\
0 & -1 & 1 \\
2 & 2 & -1 \\
\end{array}
\times
\begin{array}{ccc}
1 & 5 & 7 & 2 \\
0 & -2 & -4 & 1 \\
2 & 6 & 4 & -3 \\
0 & -2 & -2 & 1 \\
\end{array}
= + \begin{array}{ccc}
1 & 3 & 1 \\
0 & -1 & 1 \\
2 & 2 & -1 \\
\end{array}
\times
\begin{array}{ccc}
1 & 2 \\
0 & -1 \\
\end{array}
2D convolution

- Image is $D \times D$.
- $N$ filters each of size $K \times K$.
- No zero-padding.

Then output from one filter has size:

$$(D - K + 1) \times (D - K + 1)$$

For all $N$ filters,

$$N \times (D - K + 1) \times (D - K + 1)$$

https://github.com/vdumoulin/conv_arithmetic
Fully-connected layer (no shared features)

Example: 200x200 image
40K hidden units

- Spatial correlation is local
- Waste of resources + we have not enough training samples anyway..

~2B parameters!!!
Convolutional layer

 learns shared features via learned convolution kernels

Example: 200x200 image
40K hidden units
Filter size: 10x10

4M parameters
Convolution Layer

One “neuron”/kernel that “looks at” 5x5 region and outputs a sheet of activation map

\[
 h_{ij} = \sigma(v_{ij} + w_0)
\]
Convolution Layer

Add a second neuron/kernel.

3x32x32 image 3x5x5 filter

Add a second (green) filter:

Convolve (slide) over all spatial locations

two 1x28x28 activation map
Convolution Layer

- **3x32x32 image**
- **Can keep on adding, e.g. 6 filters, each 3x5x5**
- **6x3x5x5 filters**
- **Stack activations to get a 6x28x28 output image!**

6 activation maps, each 1x28x28
Convolution Layer

3x32x32 image

Also 6-dim bias vector:

Convolution Layer

28x28 grid, at each point a 6-dim vector

Stack activations to get a 6x28x28 output image!
Intuition of 2D convolution kernels

Intuition of 2D convolution kernels

"oriented edges"
Intuition of 2D convolution kernels

\[
\begin{array}{ccc}
0 & -1 & 0 \\
-1 & 4 & -1 \\
0 & -1 & 0 \\
\end{array}
\]

"sharpen"
Intuition of 2D convolution kernels

\[
\begin{array}{ccc}
  w_{11} & w_{12} & w_{13} \\
  w_{21} & w_{22} & w_{23} \\
  w_{31} & w_{32} & w_{33}
\end{array}
\]

\[ \ast \]

Gradient descent on loss will decide.
Receptive Fields

For convolution with kernel size K, each element in the output depends on a K x K **receptive field** in the input.
Receptive Fields

Each successive convolution adds $K - 1$ to the receptive field size.
With $L$ layers the receptive field size is $1 + L \times (K - 1)$

Careful – “receptive field wrt to the input”
vs “receptive field wrt the previous layer”
Receptive Fields

Each successive convolution adds $K - 1$ to the receptive field size.
With $L$ layers the receptive field size is $1 + L \times (K - 1)$.
Receptive Fields

Each successive convolution adds $K - 1$ to the receptive field size.
With $L$ layers the receptive field size is $1 + L \times (K - 1)$

**Problem:** For large images we need many layers for each output to “see” the whole image.

**Solution:** downsample inside the network
1. “Strided” convolution
2. Pooling
1. **Strided Convolution**

Input: 7x7
Filter: 3x3
Stride: 2
1. Strided Convolution

Input: 7x7
Filter: 3x3
Stride: 2
1. Strided Convolution

Input: 7x7
Filter: 3x3
Stride: 2
Output: 3x3
1. Strided Convolution

Input: 7x7
Filter: 3x3
Stride: 2

Output: 3x3

In general:
Input: W
Filter: K
Padding: P
Stride: S

Output dimension: \( \frac{(W - K + 2P)}{S} + 1 \)
(one dimension of the output square)
2. Pooling layers downsample its inputs
Also adds some local translational *invariance* (by summing/averaging):

By “pooling” (e.g., taking max) filter responses at different locations we gain robustness to the exact spatial location of features.

2. Max pooling

Max pooling with 2x2 kernel size and stride 2

No learnable parameters!
2. Average pooling

- Single depth slice

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>3</th>
<th>2</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>6</td>
<td>6</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

- Avg pooling with 2x2 kernel size and stride 2

- No learnable parameters!
Side note: sigmoid vs ReLU non-linearity in NNs

ReLU:
1. Gradient doesn’t die in one direction.
2. More efficient to compute.
3. Easier to get exactly zero activations: sparsity.
Putting it altogether! ConvNets: conv + ReLU + pooling

\[ y_i = \text{ReLU}(z_i) \]

\{z_i\} from convolutions

max pooling

convolution

convolution layer

pooling layer
Putting it altogether! ConvNets: conv + ReLU + pooling
Receptive field increases

$$y_i = \text{ReLU}(z_i)$$
Example CNN architecture

https://github.com/gwding/draw_convnet
Training CNNs

Gradient descent with back-propagation algorithm.

1. Goal is still MLE/ maximize cross-entropy.
2. Shared weights (via one convolution filter) → sum over gradient for each use of one filter.
3. Max-pooling → gradient only gets back-propagated through the neuron that “won” the max pool—technically this is a “sub-gradient”.