Overview

- Structure and power of neural networks
- Backpropagation
- Practical issues
- Convolutions

Parametric featurizations I

- So far: features \((x \text{ or } \phi(x))\) are fixed during training
  - Consider a (small) collection of feature transformations \(\phi\)
  - Select \(\phi\) via cross-validation – outside of normal training

- “Deep learning” approach:
  - Use \(\phi\) with many tunable parameters
  - Optimize parameters of \(\phi\) during normal training process

Parametric featurizations II

- Neural network: parameterization for function \(f: \mathbb{R}^d \rightarrow \mathbb{R}\)
  - \(f(x) = \phi(x)^Tw\)
  - Parameters include both \(w\) and parameters of \(\phi\)
  - Varying parameters of \(\phi\) allows \(f\) to be essentially any function!
  - Major challenge: optimization (a lot of tricks to make it work)

Figure 1: Neural network
Feedforward neural network

- **Architecture** of a feedforward neural network
  - Directed acyclic graph $G = (V, E)$
  - One *source* node (vertex) per input, one *sink* node per output
  - Other nodes are *hidden units*
  - Each edge $(u, v) \in E$ has a *weight parameter* $w_{u,v} \in \mathbb{R}$
  - *Value* $h_v$ of node $v$ given values of parents is
    \[
    h_v := \sigma_v(z_v) = \sigma_v(\sum_{u \in V: (u,v) \in E} w_{u,v} \cdot h_u).
    \]
  - $\sigma_v : \mathbb{R} \to \mathbb{R}$ is the *activation function* (e.g., sigmoid)
  
- **Figure 2: Feedforward neural network architecture**

Standard layered architectures

- Standard architecture arranges nodes into sequence of $L$ *layers*
  - Edges only go from one layer to the next
  - Can write function using matrices of weight parameters
    \[
    f(x) = \sigma_L(W_L \sigma_{L-1}(\cdots \sigma_1(W_1 x) \cdots))
    \]
  - $d_\ell$ nodes in layer $\ell$; $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ are weight parameters
  - Activation function $\sigma_\ell : \mathbb{R} \to \mathbb{R}$ is applied coordinate-wise to input
  - Often also include "bias" parameters $b_\ell \in \mathbb{R}^{d_\ell}$
    \[
    f(x) = \sigma_L(b_L + W_L \sigma_{L-1}(\cdots \sigma_1(b_1 + W_1 x) \cdots))
    \]
  - Tunable parameters: $\theta = (W_1, b_1, \ldots, W_L, b_L)$

- **Figure 3: Standard feedforward architecture**

Well-known activation functions

- **Heaviside**: $\sigma(z) = 1_{\{z \geq 0\}}$
  - Popular in the 1940s; also called *step function*
- **Sigmoid** (from logistic regression): $\sigma(z) = 1/(1 + e^{-z})$
  - Popular since 1970s
- **Hyperbolic tangent**: $\sigma(z) = \tanh(z)$
  - Similar to sigmoid, but range is $(-1, 1)$ rather than $(0, 1)$
- **Rectified Linear Unit (ReLU)**: $\sigma(z) = \max\{0, z\}$
  - Popular since 2012
- **Identity**: $\sigma(z) = z$
  - Popular with luddites
- **Softmax**: $\sigma(v)_i = \exp(v_i)/\sum_j \exp(v_j)$
  - Special vector-valued activation function
  - Popular for final layer in multi-class classification

- **Power of non-linear activations**
  - What happens if every activation function is linear/affine?
Necessity of multiple layers

- Suppose only have input and output layers, so function $f$ is
  \[ f(x) = \sigma(b + w^T x) \]
  where $b \in \mathbb{R}$ and $w \in \mathbb{R}^d$ (so $w^T \in \mathbb{R}^{1 \times d}$)
  - If $\sigma$ is monotone (e.g., Heaviside, sigmoid, hyperbolic tangent, ReLU, identity), then $f$ has same limitations as a linear/affine classifier

Figure 4: XOR problem

Neural network approximation theorems

- **Theorem** (Cybenko, 1989; Hornik, Stinchcombe, & White, 1989): Let $\sigma_1$ be any continuous non-linear activation function from above. For any continuous function $f: \mathbb{R}^d \to \mathbb{R}$ and any $\varepsilon > 0$, there is a two-layer neural network (with parameters $\theta = (W_1, b_1, w_2)$) s.t.
  \[ \max_{x \in [0,1]^d} |f(x) - w_2^T \sigma_1(b_1 + W_1 x)| < \varepsilon. \]

- Many caveats
  - “Width” (number of hidden units) may need to be very large
  - Does not tell us how to find the network
  - Does not justify deeper networks

Fitting neural networks to data

- Training data $(x_1, y_1), \ldots, (x_n, y_n) \in \mathbb{R}^d \times \mathcal{Y}$
- Fix architecture: $G = (V, E)$ and activation functions
- Plug-in principle: find parameters $\theta$ of neural network $f_\theta$ to minimize empirical risk (possibly with a surrogate loss)

  \[ \hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^{n} (f_\theta(x_i) - y_i)^2 \quad \text{regression} \]
  \[ \hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell_{\log}(-y_i f_\theta(x_i)) \quad \text{binary classification} \]
  \[ \hat{R}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ell_{\text{ce}}(\tilde{y}_i, f_\theta(x_i)) \quad \text{multi-class classification} \]

  (Could use other surrogate loss functions . . .)
  - Typically objective is not convex in parameters $\theta$
  - Nevertheless, local search (e.g., SGD) often works well!

Backpropagation

- **Backpropagation** (backprop): Algorithm for computing partial derivatives wrt weights in a feedforward neural network
  - “Clever organization of partial derivative computations with chain rule
  - Use in combination with gradient descent, SGD, etc.

  - Consider loss on a single example $(x, y)$, written $J := \ell(y, f_\theta(x))$
  - Goal: compute $\frac{\partial J}{\partial w_{u,v}}$ for every edge $(u, v) \in E$

  - Initial step of backprop: **forward propagation**
    - Compute $z_v$’s and $h_v$’s for every node $v \in V$
    - Running time: linear in size of network

  - Rest of backprop also just requires time linear in size of network!
Derivative of loss with respect to weights

- Let \( \hat{y}_1, \hat{y}_2, \ldots \) denote the values at the output nodes.
- Then by chain rule,
  \[
  \frac{\partial J}{w_{u,v}} = \sum_i \frac{\partial J}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{w_{u,v}}.
  \]

Derivative of output with respect to weights

- Assume for simplicity there is just a single output, \( \hat{y} \).
- Chain rule, again:
  \[
  \frac{\partial \hat{y}}{\partial w_{u,v}} = \frac{\partial \hat{y}}{\partial h_v} \cdot \frac{\partial h_v}{\partial w_{u,v}}.
  \]

First term: trickier; we’ll handle later

Second term:

Example: chain graph I

- Parameters \( \theta = (w_{0,1}, w_{1,2}, \ldots, w_{L-1,L}) \)
- Fix input value \( x \in \mathbb{R} \); what is \( \frac{\partial h_i}{\partial w_{i-1,i}} \) for \( i = 1, \ldots, L \)?
- Forward propagation:
  \[
  h_0 := x \\
  \text{For } i = 1, 2, \ldots, L: \quad z_i := w_{i-1,i}h_{i-1} \\
  h_i := \sigma(z_i)
  \]

Figure 5: Chain graph; assume same activation \( \sigma \) in every layer
Example: chain graph II

\[ \begin{array}{c}
0 \xrightarrow{w_{0,1}} 1 \xrightarrow{w_{1,2}} \cdots \xrightarrow{w_{L-1,L}} L
\end{array} \]

Figure 6: Chain graph; assume same activation \( \sigma \) in every layer

- **Backprop:**
  - For \( i = L, L-1, \ldots, 1 \):
    \[
    \frac{\partial h_L}{\partial h_i} := \begin{cases} 1 & \text{if } i = L \\ \frac{\partial h_L}{\partial h_{i+1}} \cdot \sigma'(z_{i+1}) w_{i,i+1} & \text{if } i < L \end{cases}
    \]
    \[
    \frac{\partial h_L}{\partial w_{i-1,i}} := \frac{\partial h_L}{\partial h_i} \cdot \sigma'(z_i) h_{i-1}
    \]

**Practical issues I: Initialization**

- Ensure inputs are **standardized**: every feature has zero mean and unit variance (wrt training data)
- Initialize weights randomly for gradient descent / SGD

**Practical issues II: Architecture choice**

- Architecture can be regarded as a “hyperparameter”
- Optimization-inspired architecture choice
  - With wide enough network, can get training error rate zero
  - Use the smallest network that lets you get zero training error rate
  - Then add regularization term to objective (e.g., sum of squares of weights), and optimize the regularized ERM objective
- Entire research communities are trying to figure out good architectures for their problems

**Convolutional nets**

- Neural networks with **convolutional layers**
  - Useful when inputs have locality structure
  - Sequential structure (e.g., speech waveform)
  - 2D grid structure (e.g., image)
  - ... 
- Weight matrix \( W_\ell \) is highly-structured
  - Determined by a small **filter**
  - Time to compute \( W_\ell h_{\ell-1} \) is typically \( \ll d_\ell \times d_{\ell-1} \) (e.g., closer to \( \max\{d_\ell, d_{\ell-1}\} \))
Convolutions I

- Convolution of two continuous functions: \( h := f \ast g \)
  \[
  h(x) = \int_{-\infty}^{+\infty} f(y)g(x-y) \, dy
  \]
- If \( f(x) = 0 \) for \( x \notin [-w, +w] \), then
  \[
  h(x) = \int_{-w}^{+w} f(y)g(x-y) \, dy
  \]
- Replaces \( g(x) \) with weighted combination of \( g \) at nearby points
- For functions on discrete domain, replace integral with sum
  \[
  h(i) = \sum_{j=-\infty}^{\infty} f(j)g(i-j)
  \]

Convolutions II

- E.g., suppose only \( f(0), f(1), f(2) \) are non-zero, and \( g \) is zero-padded (in this case, \( g(i) = 0 \) for \( i < 1 \) or \( i > 5 \)). Then:
  \[
  \begin{bmatrix}
  h(1) \\
  h(2) \\
  h(3) \\
  h(4) \\
  h(5) \\
  h(6) \\
  h(7)
  \end{bmatrix}
  =
  \begin{bmatrix}
  f(0) & 0 & 0 & 0 & 0 \\
  f(1) & f(0) & 0 & 0 & 0 \\
  f(2) & f(1) & f(0) & 0 & 0 \\
  0 & f(2) & f(1) & f(0) & 0 \\
  0 & 0 & f(2) & f(1) & f(0) \\
  0 & 0 & 0 & f(2) & f(1) \\
  0 & 0 & 0 & 0 & f(2)
  \end{bmatrix}
  \begin{bmatrix}
  g(1) \\
  g(2) \\
  g(3) \\
  g(4) \\
  g(5)
  \end{bmatrix}
  \]

2D convolutions I

- Similar for 2D inputs (e.g., images), except now sum over two indices
  - \( g \) is the input image
  - \( f \) is the filter
  - Lots of variations (e.g., padding, strides, multiple “channels”)

2D convolutions II

- Similar for 2D inputs (e.g., images), except now sum over two indices
  - \( g \) is the input image
  - \( f \) is the filter
  - Lots of variations (e.g., padding, strides, multiple “channels”)

![Figure 7: Convolutional layer](image)

![Figure 8: 2D convolution, with padding, no stride](image)

![Figure 9: 2D convolution, with padding, no stride](image)
2D convolutions III

- Similar for 2D inputs (e.g., images), except now sum over two indices
  - \( g \) is the input image
  - \( f \) is the filter
  - Lots of variations (e.g., padding, strides, multiple “channels”)

> Figure 10: 2D convolution, with padding, no stride

2D convolutions IV

- Similar for 2D inputs (e.g., images), except now sum over two indices
  - \( g \) is the input image
  - \( f \) is the filter
  - Lots of variations (e.g., padding, strides, multiple “channels”)

> Figure 11: 2D convolution, with padding, no stride

24 / 26

Postscript: Tangent model

- Let \( f_\theta : \mathbb{R}^d \rightarrow \mathbb{R} \) be a neural network function with parameters \( \theta \in \mathbb{R}^p \) (\( p \) is total number of parameters)
- Fix \( x \), and consider first-order approximation of \( f_\theta(x) \) around \( \theta = \theta^{(0)} \):

\[
 f_\theta(x) \approx f_{\theta^{(0)}}(x) + \nabla f_{\theta^{(0)}}(x)^T (\theta - \theta^{(0)})
\]

Here, \( \nabla \) is gradient wrt parameters \( \theta \), not wrt input \( x \)
- Consider feature transformation \( \varphi(x) := \nabla f_{\theta^{(0)}}(x) \), determined entirely by initial parameters \( \theta^{(0)} \)
- If the first-order approximation is accurate (i.e., we never consider \( \theta \) too far from \( \theta^{(0)} \)), then back to a linear model (over \( p \) features \( \varphi(x) \in \mathbb{R}^p \))
  - Called the tangent model
- Upshot: To really exploit power of “deep learning”, we must be changing parameters a lot during training!

25 / 26

26 / 26