Machine learning lecture slides

Optimization II: Neural networks

- Architecture of (layered) feedforward neural networks
- Universal approximation
- Backpropagation
- Practical issues

Parametric featurizations

- So far: data features (x or $\varphi(x)$) are fixed during training
  - Consider a (small) collection of feature transformations φ
  - Select φ via cross-validation – outside of normal training
- "Deep learning" approach:
  - Use φ with many tunable parameters
- $\mathbb{R}$
  - Varying parameters of φ allows
  - Major challenge: optimization

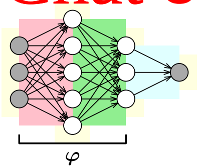


Figure 1: Neural networks as feature maps

- *Architecture* of a *feedforward neural network*
  - Directed acyclic graph $G = (V, E)$
  - One *source* node (vertex) per input to the function $(x_1, \dots, x_d)$
  - One *sink* node per output of the function
  - Internal nodes are called *hidden units*

$\mathbb{R}$

$$h_v := \sigma_v(z_v), \quad z_v :$$

  - $\sigma_v : \mathbb{R} \to \mathbb{R}$ is the *activation func*
  - E.g., sigmoid function $\sigma_v(z) = 1/(1 + e^{-z})$
- Inspired by neurons in the brain

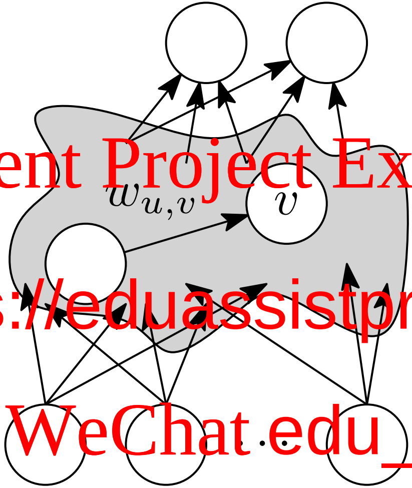


Figure 2: Computation DAG of a feedforward neural network

Standard layered architectures

- Standard feedforward architecture arranges nodes into *layers*
  - Initial layer (layer zero): source nodes (input)
  - Final layer (layer L): sink nodes (output)
  - (Layer counting is confusing, usually don't count input)
- Edges only go from one layer to the next

k

-

$$f(x) = \sigma_L(W_L \sigma_{L-1}(\cdots$$

  - Layer ℓ has d_ℓ nodes
  - $W_\ell \in \mathbb{R}^{d_\ell \times d_{\ell-1}}$ are the weight para
  - Scalar-valued activation function applied coordinate-wise to input

- Often also include "bias" parameters $b_\ell \in \mathbb{R}^{d_\ell}$

$$f(x) = \sigma_L(b_L + W_L \sigma_{L-1}(\cdots \sigma_1(b_1 + W_1 x) \cdots))$$

- The tunable parameters: $\theta = (W_1, b_1, \ldots, W_L, b_L)$

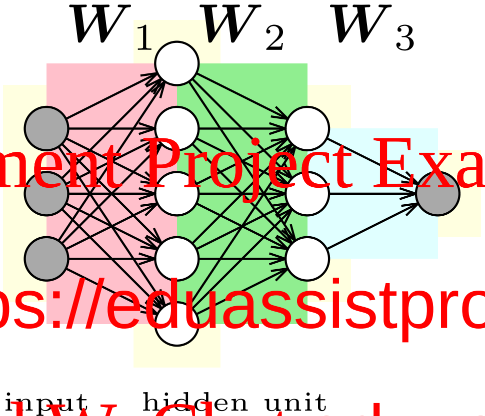


Figure 3: Layered feedforward neu

Well-known activation functions

- *Heaviside*: $\sigma(z) = \mathbf{1}_{\{z \geq 0\}}$
  - Popular in the 1940s; also called *step function*
- *Sigmoid* (from logistic regression): $\sigma(z) = 1/(1 + e^{-z})$
  - Popular since 1970s
-

  $.1)$
-
  - Popular since 2012
- *Identity*: $\sigma(z) = z$
  - Popular with luddites

Power of non-linear activations

- What happens if every activation function is linear/affine?
  - Overall function is affine
  - An unusual way to parameterize an affine function
- Therefore, use non-linear/non-affine activation functions

- Suppose only have input and output layers, so function f is

$$f(x) = \sigma(b + w^{\mathsf{T}} x)$$

where $b \in \mathbb{R}$ and $w \in \mathbb{R}^d$ (so $w^{\mathsf{T}} \in \mathbb{R}^{1 \times d}$)

- If σ is monotone (e.g., Heaviside, sigmoid, hyperbolic tangent,

Figure 4: XOR data set

Necessity of multiple layers (2)

- XOR problem
  - Let $x^{(1)} = (+1, +1)$, $x^{(2)} = (+1, -1)$, $x^{(3)} = (-1, +1)$, $x^{(4)} = (-1, -1)$.
  - $y^{(i)} = +1$ iff coordinates of $x^{(i)}$ are the same. (XNOR)
  - Suppose (w, b) $\mathbb{R}^2$ $\mathbb{R}$ satisfies

$$b + w^\top \quad {}^{(3)}$$

  - Add up the equations:

$$b + w^\top(x^{(2)} + x$$

  - But $x^{(2)} + x^{(3)} - x^{(1)} = x^{(4)}$, so

$$b + w^\top x^{(4)} < 0.$$

In other words, cannot correctly label $x^{(4)}$.

Neural network approximation theorems

- **Theorem** (Cybenko, 1989; Hornik, Stinchcombe, & White, 1989): Let σ_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

- This property of such families of neural netw *universal approximation*.
- Many caveats
  - "Width" (number of hidden units) may n
  - Does not tell us how to find the network
  - Does not justify deeper networks

Stone-Weierstrass theorem (polynomial version)

Theorem (Weierstrass, 1885): For any continuous function $f\colon [a,b] \to \mathbb{R}$, and any $\epsilon > 0$, there exists a polynomial p such that

$$\sup_{x \in [a,b]} |f(x) - p(x)| < \epsilon.$$

Theorem (Stone, 1937): Let $K \subset \mathbb{R}^d$ be any bounded set. Let A be a set of continuous functions on K such that the following hold.

(1) A is an *algebra* (i.e., A is closed under addition, multiplication, and scalar multiplication).

(2)

(3) , there $h \in A$ $h(x) \neq h(y)$

For any continuous function $f: K \to$ s $h \in A$ such that

$$\sup_{x \in K} |f(x) - h(x)| < \epsilon.$$

Let $K = [0,1]^d$, and let

$$A = \left\{ x \mapsto \sum_{k=1}^{m} a_k \cos(x^\top w_k + b_k) : \right\}.$$

(Che

Two-layer neural networks with exp activation functions

Let $K = [0,1]^d$, and let

$$A = \left\{ x \mapsto \sum_{k=1}^{m} a_k \exp(x^\top w_k + b_k) : \right\}.$$

(Che

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

  - Regression $y_i \quad \mathbb{R}$

$$y_i \in \{- \quad ,$$

$$\hat{\mathcal{R}}(\theta) = \frac{1}{n} \sum_{i=1}^{n} \ln(1$$

(Could use other surrogate loss functions . . .)

  - Can also add regularization terms, but also common to use *algorithmic regularization*
- Typically objective is not convex in parameters θ
- Nevertheless, local search (e.g., gradient descent, 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.
-

- Goal: compute $\frac{\partial J}{\partial w_{u,v}}$ for every edge
- Initial step of backprop: *forward pr*
  - Compute z_v's and h_v's for every no
  - Running time: linear in size of network
- We'll see that 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,

$$\frac{\partial J}{w_{u,v}} = \frac{\partial J}{\partial \hat{y}_i} \cdot \frac{\partial \hat{y}_i}{w_{u,v}}.$$

-
- $\frac{}{w_{u,v}}$
- Assume for simplicity there is just a single out

- Chain rule, again:

$$\frac{\partial \hat{y}}{\partial w_{u,v}} = \frac{\partial \hat{y}}{\partial h_v} \cdot \frac{\partial h_v}{\partial w_{u,v}}$$

-
-

$z_v \quad w_{u,v} \cdot h_u \qquad w_{u,v}$

  - Therefore

$$\frac{\partial h_v}{\partial w_{u,v}} = \frac{\partial h_v}{\partial z_v} \frac{\partial z}{\partial w}$$

  - z_v and h_u were computed during forward propagation

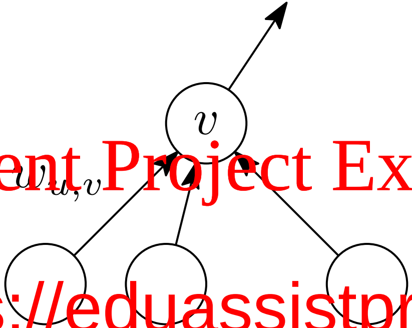


Figure 5: Derivative of a node's output with respect to an in

Derivative of output with respect to hidden units

- Key trick: compute $\frac{\partial \hat{y}}{\partial h_v}$ for all vertices in decreasing order of layer number
- If v is not the output node, then by chain rule (yet again)

$$\frac{\partial \hat{y}}{\partial h_v} = \sum \frac{\partial \hat{y}}{\partial h_{v'}} \cdot \frac{\partial h_{v'}}{\partial h_v}$$

  - $h_{v'} = \sigma_{v'}(z_{v'})$
  - $z_{v'} = w_{v,v'} \cdot h_v +$ (terms not invol
  - Therefore

$$\frac{\partial h_{v'}}{\partial h_v} = \frac{\partial h_{v'}}{\partial z_{v'}} \cdot \frac{\partial z_{v'}}{\partial h_v} = \sigma'(z_{v'}) \cdot w_{v,v'}.$$

  - $z_{v'}$'s were computed during forward propagation
  - $w_{v,v'}$'s are the values of the weight parameters at which we want to compute the gradient

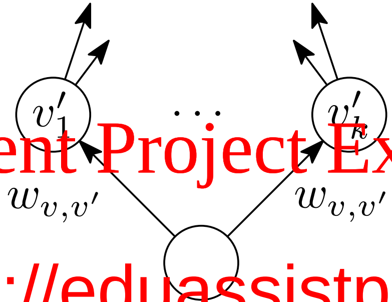


Figure 6: Derivative of the network output with respec
values

- Function $f_\theta \colon \mathbb{R} \to \mathbb{R}$
- Architecture
  - DAG: $0 \longrightarrow 1 \longrightarrow 2 \longrightarrow \cdots \longrightarrow L$
  - Same activation σ in every layer

  - $h_0 := x$
  - For $i = 1, 2, \ldots, L$:

$$z_i := w_{i-1,i} h_{i-1}$$
$$h_i := \sigma(z_i)$$

Example: chain graph (2)

- Backprop:
  - For $i = L, L-1, \ldots, 1$:

$$\frac{\partial h_L}{\partial h_i} := \begin{cases} 1 & \text{if } i = L \\ \frac{\partial h_L}{\;} \quad \sigma'(z_{i+1}) w_{i,i+1} & \text{if } i < L \end{cases}$$

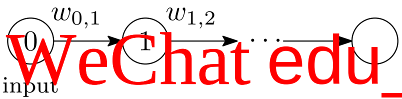


Figure 7: Neural network with a chain computation graph

- Ensure inputs are *standardized*: every feature has zero mean and unit variance (wrt training data)
  - Even better: different features have zero covariance (again, on training data)
  - But this can be expensive

-

  - Heuristic: ensure h_v have similar s
  - E.g., using $\tanh$-activation, if v $1/k$ for all weights $w_{u,v}$
  - Many initialization schemes for other a dealing with bias parameters, . . .

Practical issues II: Architecture choice

- Architecture can be regarded as a "hyperparameter"
  - Could use cross-validation to select, but . . .
  - Many "good" architectures are known for popular problems (e.g., image classification)
  - Unclear what to do for completely new problems
-
  - Then add regularization term to objecti
    of weights), and optimize the regularize
- Entire research communities are trying to
  architectures for their problems

Vector-valued activation: $\sigma\colon \mathbb{R}^{d_\ell} \to \mathbb{R}^{d_\ell}$

- *Softmax* activation: $\sigma(v)_i = \exp(v_i)/\sum_j \exp(v_j)$
- Common to use this in final layer

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)
  - . . .
- .g., closer to $\max\{d_\ell, d_{\ell-1}\}$)

- Convolution of two continuous functions: $h := f * g$

$$h(x) = \int_{-\infty}^{+\infty} f(y)\, g(x-y)\, dy$$

-

$$-w$$

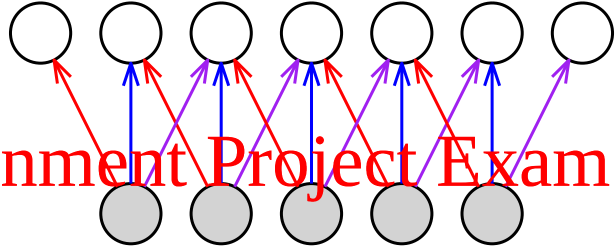
  - Replaces $g(x)$ with weighted comb s

- For functions on discrete domain, replace integral with sum

$$h(i) = \sum_{j=-\infty}^{\infty} f(j)\, g(i-j)$$

$$\begin{bmatrix} h(2) \\ h(3) \\ h(4) \\ h(5) \\ h(6) \\ h(7) \end{bmatrix} = \begin{bmatrix} f(1) & f(0) & 0 & 0 & 0 \\ f(2) & f(1) & f(0) & 0 & 0 \\ 0 & f(2) & f(1) & f(0) & 0 \\ 0 & 0 & f(2) & f(1) & f(0) \\ 0 & 0 & 0 & f(2) & f(1) \\ 0 & 0 & 0 & 0 & f(2) \end{bmatrix} \begin{bmatrix} g(1) \\ g(2) \\ g(3) \\ g(4) \\ g(5) \end{bmatrix}$$

(Here, we pretend $g(i) = 0$ for $i < 1$ and $i > 5$.)



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")
-

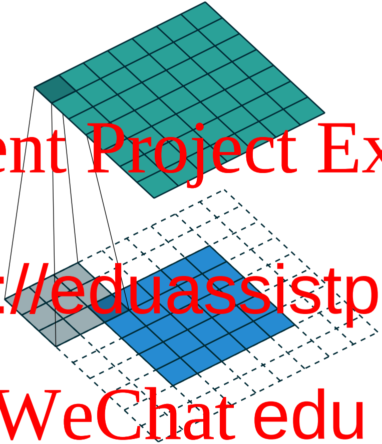


Figure 9: 2D convolution

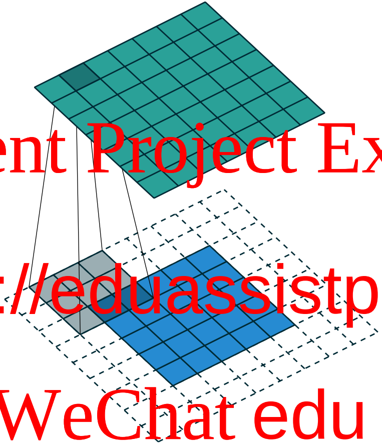


Figure 10: 2D convolution

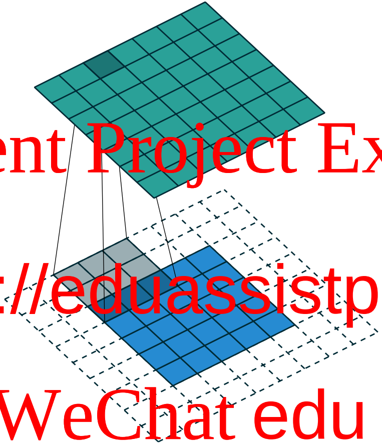


Figure 11: 2D convolution

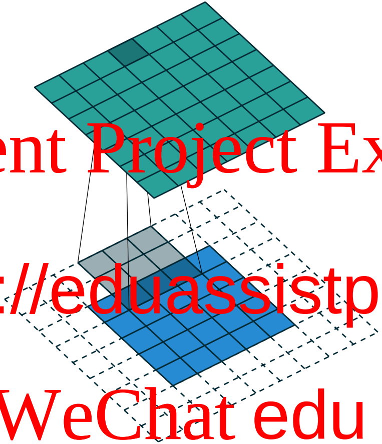


Figure 12: 2D convolution