Kernel machines and neural networks

COMS 4721 Spring 2022

Daniel Hsu

Kernel machines

Motivation

- **Feature expansions** $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$
  - In modeling some problems, may not want to assume that target function (e.g., $f(\vec{x}) = \mathbb{E}[Y \mid \vec{X} = \vec{x}]$) is linear function of $\vec{x}$
  - Feature expansions provide a way to "upgrade" linear regression / classification methods to produce non-linear functions / non-hyperplane decision boundaries
- What if you have very little prior knowledge about the target function?

Why polynomials?

Taylor's theorem. For any continuous function $f\colon \mathbb{R} \to \mathbb{R}$, any $k \in \mathbb{N}$, and any $x_0 \in \mathbb{R}$, if f is "nice" enough, there exists a degree-k polynomial $P\colon \mathbb{R} \to \mathbb{R}$ such that

$$\lim_{x \to x_0} \frac{|f(x) - P(x)|}{|x - x_0|^k} \;=\; 0$$

Why polynomials?

Taylor's theorem. For any continuous function $f\colon \mathbb{R} \to \mathbb{R}$, any $k \in \mathbb{N}$, and any $x_0 \in \mathbb{R}$, if f is "nice" enough, there exists a degree-k polynomial $P\colon \mathbb{R} \to \mathbb{R}$ such that

$$\lim_{x \to x_0} \frac{|f(x) - P(x)|}{|x - x_0|^k} = 0$$

- Polynomials can give good **local** approximations

Why polynomials?

Taylor's theorem. For any continuous function $f\colon \mathbb{R} \to \mathbb{R}$, any $k \in \mathbb{N}$, and any $x_0 \in \mathbb{R}$, if f is "nice" enough, there exists a degree-k polynomial $P\colon \mathbb{R} \to \mathbb{R}$ such that

$$\lim_{x \to x_0} \frac{|f(x) - P(x)|}{|x - x_0|^k} \;=\; 0$$

- Polynomials can give good **local** approximations
- Higher **degree** $k \to$ better approximation (assuming f is "nice" enough)

Why polynomials?

Taylor's theorem. For any continuous function $f\colon \mathbb{R} \to \mathbb{R}$, any $k \in \mathbb{N}$, and any $x_0 \in \mathbb{R}$, if f is "nice" enough, there exists a degree-k polynomial $P\colon \mathbb{R} \to \mathbb{R}$ such that

$$\lim_{x \to x_0} \frac{|f(x) - P(x)|}{|x - x_0|^k} \; = \; 0$$

- Polynomials can give good **local** approximations
- Higher **degree** $k \to$ better approximation (assuming f is "nice" enough)
- There's also version for "nice" multivariate functions $f\colon \mathbb{R}^d \to \mathbb{R}$

Why polynomials?

Taylor's theorem. For any continuous function $f\colon \mathbb{R} \to \mathbb{R}$, any $k \in \mathbb{N}$, and any $x_0 \in \mathbb{R}$, if f is "nice" enough, there exists a degree-k polynomial $P\colon \mathbb{R} \to \mathbb{R}$ such that

$$\lim_{x \to x_0} \frac{|f(x) - P(x)|}{|x - x_0|^k} = 0$$

- Polynomials can give good **local** approximations
- Higher **degree** $k \to$ better approximation (assuming f is "nice" enough)
- There's also version for "nice" multivariate functions $f\colon \mathbb{R}^d \to \mathbb{R}$

Maybe not so impressive . . .

Really, why polynomials?

Weierstrass approximation theorem. For any continuous function $f\colon \mathbb{R} \to \mathbb{R}$, any bounded interval $[a, b] \subset \mathbb{R}$, and any $\varepsilon > 0$, there exists a polynomial $P\colon \mathbb{R} \to \mathbb{R}$ such that

$$\max_{x \in [a,b]} |f(x) - P(x)| \leq \varepsilon$$

Really, why polynomials?

Weierstrass approximation theorem. For any continuous function $f\colon \mathbb{R} \to \mathbb{R}$, any bounded interval $[a, b] \subset \mathbb{R}$, and any $\varepsilon > 0$, there exists a polynomial $P\colon \mathbb{R} \to \mathbb{R}$ such that

$$\max_{x \in [a,b]} |f(x) - P(x)| \leq \varepsilon$$

- Polynomials give good approximations **uniformly over a bounded interval** (!)

Really, why polynomials?

Weierstrass approximation theorem. For any continuous function $f\colon \mathbb{R} \to \mathbb{R}$, any bounded interval $[a, b] \subset \mathbb{R}$, and any $\varepsilon > 0$, there exists a polynomial $P\colon \mathbb{R} \to \mathbb{R}$ such that

$$\max_{x \in [a,b]} |f(x) - P(x)| \leq \varepsilon$$

- Polynomials give good approximations **uniformly over a bounded interval** (!)
- Degree of P may need to grow with $1/\varepsilon$

Really, why polynomials?

Weierstrass approximation theorem. For any continuous function $f\colon \mathbb{R} \to \mathbb{R}$, any bounded interval $[a, b] \subset \mathbb{R}$, and any $\varepsilon > 0$, there exists a polynomial $P\colon \mathbb{R} \to \mathbb{R}$ such that

$$\max_{x \in [a,b]} |f(x) - P(x)| \leq \varepsilon$$

- Polynomials give good approximations **uniformly over a bounded interval** (!)
- Degree of P may need to grow with $1/\varepsilon$
- There's also version for multivariate functions $f\colon \mathbb{R}^d \to \mathbb{R}$

Really, why polynomials?

Weierstrass approximation theorem. For any continuous function $f\colon \mathbb{R} \to \mathbb{R}$, any bounded interval $[a,b] \subset \mathbb{R}$, and any $\varepsilon > 0$, there exists a polynomial $P\colon \mathbb{R} \to \mathbb{R}$ such that

$$\max_{x \in [a,b]} |f(x) - P(x)| \leq \varepsilon$$

- Polynomials give good approximations **uniformly over a bounded interval** (!)
- Degree of P may need to grow with $1/\varepsilon$
- There's also version for multivariate functions $f\colon \mathbb{R}^d \to \mathbb{R}$

Upshot: Even with little information about target function, can still hope to get good approximations using polynomials of high-enough degree

Caveat

Caveat:

- In standard IID model, OLS with feature expansion may have bad MSE unless

$$\text{sample size} \gg \text{dimension of feature expansion}$$

Caveat

Caveat:

- In standard IID model, OLS with feature expansion may have bad MSE unless

$$\text{sample size} \gg \text{dimension of feature expansion}$$

- Dimension of degree-k polynomial expansion (in $\mathbb{R}^d$): $\Theta(d^k)$

Caveat

Caveat:

- In standard IID model, OLS with feature expansion may have bad MSE unless

  $$\text{sample size} \gg \text{dimension of feature expansion}$$

- Dimension of degree-k polynomial expansion (in $\mathbb{R}^d$): $\Theta(d^k)$
- Saving grace: **inductive bias** (e.g., **data augmentation**, **margins**)

Caveat

Caveat:

- In standard IID model, OLS with feature expansion may have bad MSE unless

  $$\text{sample size} \gg \text{dimension of feature expansion}$$

- Dimension of degree-k polynomial expansion (in $\mathbb{R}^d$): $\Theta(d^k)$
- Saving grace: **inductive bias** (e.g., **data augmentation**, **margins**)

Issue that still remains: **Computation**

Trick to compute dot products quickly (simple case)

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^2 \to \mathbb{R}^6$: (Don't mind the $\sqrt{2}$'s)

$$\vec{\varphi}(\vec{x}) \ := \ (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2)$$

Trick to compute dot products quickly (simple case)

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^2 \to \mathbb{R}^6$: (Don't mind the $\sqrt{2}$'s)

$$\vec{\varphi}(\vec{x}) \;:=\; (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2)$$

Basic operation needed in feature space: **Dot products** $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$

$$\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) \;=\; 1 + 2x_1 z_1 + 2x_2 z_2 + x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 z_1 x_2 z_2 \quad \text{(6 terms to add)}$$

Trick to compute dot products quickly (simple case)

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^2 \to \mathbb{R}^6$: (Don't mind the $\sqrt{2}$'s)

$$\vec{\varphi}(\vec{x}) \;:=\; (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1x_2)$$

Basic operation needed in feature space: **Dot products** $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$

$$\begin{aligned}\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) &= 1 + 2x_1z_1 + 2x_2z_2 + x_1^2z_1^2 + x_2^2z_2^2 + 2x_1z_1x_2z_2 \quad \text{(6 terms to add)} \\ &= 1 + 2(x_1z_1 + x_2z_2) + (x_1z_1 + x_2z_2)^2\end{aligned}$$

Trick to compute dot products quickly (simple case)

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^2 \to \mathbb{R}^6$: (Don't mind the $\sqrt{2}$'s)

$$\vec{\varphi}(\vec{x}) \;:=\; (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2)$$

Basic operation needed in feature space: **Dot products** $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$

$$\begin{aligned}
\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) &= 1 + 2x_1 z_1 + 2x_2 z_2 + x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 z_1 x_2 z_2 && \text{(6 terms to add)} \\
&= 1 + 2(x_1 z_1 + x_2 z_2) + (x_1 z_1 + x_2 z_2)^2 \\
&= (1 + x_1 z_1 + x_2 z_2)^2 && \text{(3 terms to add)}
\end{aligned}$$

Trick to compute dot products quickly (simple case)

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^2 \to \mathbb{R}^6$: (Don't mind the $\sqrt{2}$'s)

$$\vec{\varphi}(\vec{x}) \;:=\; (1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, x_2^2, \sqrt{2}x_1 x_2)$$

Basic operation needed in feature space: **Dot products** $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$

$$\begin{aligned}
\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) &= 1 + 2x_1 z_1 + 2x_2 z_2 + x_1^2 z_1^2 + x_2^2 z_2^2 + 2x_1 z_1 x_2 z_2 && \text{(6 terms to add)} \\
&= 1 + 2(x_1 z_1 + x_2 z_2) + (x_1 z_1 + x_2 z_2)^2 \\
&= (1 + x_1 z_1 + x_2 z_2)^2 && \text{(3 terms to add)} \\
&= (1 + \vec{x} \cdot \vec{z})^2
\end{aligned}$$

Some (small) savings! But what about in $\mathbb{R}^d$?

Trick to compute dot products quickly, again

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = 1 + 2d + \binom{d}{2} = \Theta(d^2)$:

$$\vec{\varphi}(\vec{x}) \;:=\; (1, \underbrace{\sqrt{2}x_1, \ldots, \sqrt{2}x_d}_{d \text{ linear terms}}, \underbrace{x_1^2, \ldots, x_d^2}_{d \text{ square terms}}, \underbrace{\sqrt{2}x_1x_2, \ldots, \sqrt{2}x_{d-1}x_d}_{\binom{d}{2} \text{ cross terms}})$$

Trick to compute dot products quickly, again

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = 1 + 2d + \binom{d}{2} = \Theta(d^2)$:

$$\vec{\varphi}(\vec{x}) \ := \ (1, \underbrace{\sqrt{2}x_1, \dots, \sqrt{2}x_d}_{d \text{ linear terms}}, \underbrace{x_1^2, \dots, x_d^2}_{d \text{ square terms}}, \underbrace{\sqrt{2}x_1x_2, \dots, \sqrt{2}x_{d-1}x_d}_{\binom{d}{2} \text{ cross terms}})$$

Dot products $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$:

$$\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) \ = \ (1 + \vec{x} \cdot \vec{z})^2 \qquad \text{(only } d+1 \text{ terms to add)}$$

Trick to compute dot products quickly, again

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = 1 + 2d + \binom{d}{2} = \Theta(d^2)$:

$$\vec{\varphi}(\vec{x}) \;:=\; (1, \underbrace{\sqrt{2}x_1, \dots, \sqrt{2}x_d}_{d \text{ linear terms}}, \underbrace{x_1^2, \dots, x_d^2}_{d \text{ square terms}}, \underbrace{\sqrt{2}x_1x_2, \dots, \sqrt{2}x_{d-1}x_d}_{\binom{d}{2} \text{ cross terms}})$$

Dot products $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$:

$$\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) \;=\; (1 + \vec{x} \cdot \vec{z})^2 \qquad \text{(only } d+1 \text{ terms to add)}$$

Naïve implementation to compute dot product: $\Theta(d^2)$ time; Using the trick: $\Theta(d)$

Trick to compute dot products quickly, again

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = 1 + 2d + \binom{d}{2} = \Theta(d^2)$:

$$\vec{\varphi}(\vec{x}) \;:=\; (1, \underbrace{\sqrt{2}x_1, \dots, \sqrt{2}x_d}_{d \text{ linear terms}}, \underbrace{x_1^2, \dots, x_d^2}_{d \text{ square terms}}, \underbrace{\sqrt{2}x_1x_2, \dots, \sqrt{2}x_{d-1}x_d}_{\binom{d}{2} \text{ cross terms}})$$

Dot products $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$:

$$\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) \;=\; (1 + \vec{x} \cdot \vec{z})^2 \qquad \text{(only } d+1 \text{ terms to add)}$$

Naïve implementation to compute dot product: $\Theta(d^2)$ time; Using the trick: $\Theta(d)$

Degree-k polynomial expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = \Theta(d^k)$:

$$\vec{\varphi}(\vec{x}) \;:=\; (\text{all possible monomials over } x_1, \dots, x_d \text{ of degree} \le k)$$

Trick to compute dot products quickly, again

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = 1 + 2d + \binom{d}{2} = \Theta(d^2)$:

$$\vec{\varphi}(\vec{x}) \;:=\; (1, \underbrace{\sqrt{2}x_1, \dots, \sqrt{2}x_d}_{d \text{ linear terms}}, \underbrace{x_1^2, \dots, x_d^2}_{d \text{ square terms}}, \underbrace{\sqrt{2}x_1x_2, \dots, \sqrt{2}x_{d-1}x_d}_{\binom{d}{2} \text{ cross terms}})$$

Dot products $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$:

$$\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) \;=\; (1 + \vec{x} \cdot \vec{z})^2 \qquad \text{(only } d+1 \text{ terms to add)}$$

Naïve implementation to compute dot product: $\Theta(d^2)$ time; Using the trick: $\Theta(d)$

Degree-k polynomial expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = \Theta(d^k)$:

$$\vec{\varphi}(\vec{x}) \;:=\; (\text{all possible monomials over } x_1, \dots, x_d \text{ of degree} \leq k)$$

Dot products $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$:

$$\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) \;=\; (1 + \vec{x} \cdot \vec{z})^k \qquad \text{(only } d+1 \text{ terms to add)}$$

Trick to compute dot products quickly, again

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = 1 + 2d + \binom{d}{2} = \Theta(d^2)$:

$$\vec{\varphi}(\vec{x}) \;:=\; (1, \underbrace{\sqrt{2}x_1, \dots, \sqrt{2}x_d}_{d \text{ linear terms}}, \underbrace{x_1^2, \dots, x_d^2}_{d \text{ square terms}}, \underbrace{\sqrt{2}x_1x_2, \dots, \sqrt{2}x_{d-1}x_d}_{\binom{d}{2} \text{ cross terms}})$$

Dot products $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$:

$$\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) \;=\; (1 + \vec{x} \cdot \vec{z})^2 \qquad \text{(only } d+1 \text{ terms to add)}$$

Naïve implementation to compute dot product: $\Theta(d^2)$ time; Using the trick: $\Theta(d)$

Degree-k polynomial expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = \Theta(d^k)$:

$$\vec{\varphi}(\vec{x}) \;:=\; (\text{all possible monomials over } x_1, \dots, x_d \text{ of degree} \leq k)$$

Dot products $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$:

$$\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) \;=\; (1 + \vec{x} \cdot \vec{z})^k \qquad \text{(only } d+1 \text{ terms to add)}$$

Naïve implementation to compute dot product: $\Theta(d^k)$ time; Using the trick: $\Theta(d)$

Trick to compute dot products quickly, again

Quadratic expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = 1 + 2d + \binom{d}{2} = \Theta(d^2)$:

$$\vec{\varphi}(\vec{x}) \;:=\; (1, \underbrace{\sqrt{2}x_1, \dots, \sqrt{2}x_d}_{d \text{ linear terms}}, \underbrace{x_1^2, \dots, x_d^2}_{d \text{ square terms}}, \underbrace{\sqrt{2}x_1x_2, \dots, \sqrt{2}x_{d-1}x_d}_{\binom{d}{2} \text{ cross terms}})$$

Dot products $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$:

$$\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) \;=\; (1 + \vec{x} \cdot \vec{z})^2 \qquad \text{(only } d+1 \text{ terms to add)}$$

Naïve implementation to compute dot product: $\Theta(d^2)$ time; Using the trick: $\Theta(d)$

Degree-k polynomial expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ where $D = \Theta(d^k)$:

$$\vec{\varphi}(\vec{x}) \;:=\; (\text{all possible monomials over } x_1, \dots, x_d \text{ of degree} \leq k)$$

Dot products $\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z})$:

$$\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{z}) \;=\; (1 + \vec{x} \cdot \vec{z})^k \qquad \text{(only } d+1 \text{ terms to add)}$$

Naïve implementation to compute dot product: $\Theta(d^k)$ time; Using the trick: $\Theta(d)$

But how about dot products between $\vec{\varphi}(\vec{x})$ and weight vector $\vec{w} \in \mathbb{R}^D$?

Living in the span

Observation: Many "linear" learning methods yield $\vec{w}$ in the **span of the training feature vectors**

- OLS, ridge regression, PCR, SVM, PCA
- Gradient descent, SGD (on SSE & SLL objectives) if initialized at zero
- Perceptron, Online Perceptron

Living in the span

Observation: Many "linear" learning methods yield $\vec{w}$ in the **span of the training feature vectors**

- OLS, ridge regression, PCR, SVM, PCA
- Gradient descent, SGD (on SSE & SLL objectives) if initialized at zero
- Perceptron, Online Perceptron

When used with feature expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$, this means there exist $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$ such that

$$\vec{w} \;=\; \sum_{i=1}^{n} \alpha_i \, \vec{\varphi}(\vec{x}_i)$$

where $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$ are the training examples

I.e., there's an **implicit representation** of $\vec{w}$ in terms of $\vec{\alpha} := (\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$ & training examples

Living in the span

Observation: Many "linear" learning methods yield $\vec{w}$ in the **span of the training feature vectors**

- OLS, ridge regression, PCR, SVM, PCA
- Gradient descent, SGD (on SSE & SLL objectives) if initialized at zero
- Perceptron, Online Perceptron

When used with feature expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$, this means there exist $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$ such that

$$\vec{w} \;=\; \sum_{i=1}^{n} \alpha_i \, \vec{\varphi}(\vec{x}_i)$$

where $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$ are the training examples

I.e., there's an **implicit representation** of $\vec{w}$ in terms of $\vec{\alpha} := (\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$ & training examples

To compute $\vec{\varphi}(\vec{x}) \cdot \vec{w}$, can use

$$\vec{\varphi}(\vec{x}) \cdot \vec{w} \;=\; \sum_{i=1}^{n} \alpha_i \left(\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{x}_i) \right)$$

Living in the span

Observation: Many "linear" learning methods yield $\vec{w}$ in the **span of the training feature vectors**

- OLS, ridge regression, PCR, SVM, PCA
- Gradient descent, SGD (on SSE & SLL objectives) if initialized at zero
- Perceptron, Online Perceptron

When used with feature expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$, this means there exist $\alpha_1, \ldots, \alpha_n \in \mathbb{R}$ such that

$$\vec{w} \;=\; \sum_{i=1}^{n} \alpha_i \, \vec{\varphi}(\vec{x}_i)$$

where $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n)$ are the training examples

I.e., there's an **implicit representation** of $\vec{w}$ in terms of $\vec{\alpha} := (\alpha_1, \ldots, \alpha_n) \in \mathbb{R}^n$ & training examples

To compute $\vec{\varphi}(\vec{x}) \cdot \vec{w}$, can use

$$\vec{\varphi}(\vec{x}) \cdot \vec{w} \;=\; \sum_{i=1}^{n} \alpha_i \left(\vec{\varphi}(\vec{x}) \cdot \vec{\varphi}(\vec{x}_i) \right)$$

But how do you get the α_i's?

Perceptron with feature expansion

Given: Training data $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n) \in \mathbb{R}^d \times \{0, 1\}$

Perceptron with feature map $\vec{\varphi} \colon \mathbb{R}^d \to \mathbb{R}^D$

- Initialize $\vec{w} := \vec{0} \in \mathbb{R}^D$
- Loop:
  - Pick any example $(\vec{x}_i, y_i)$ that is misclassified by $\vec{w}$
  - (If there is no such example, halt and return $\vec{w}$)
  - Update $\vec{w}$:

$$\vec{w} := \begin{cases} \vec{w} + \vec{\varphi}(\vec{x}_i) & \text{if } y_i = 1 \\ \vec{w} - \vec{\varphi}(\vec{x}_i) & \text{if } y_i = 0 \end{cases}$$

Perceptron with feature expansion

Given: Training data $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n) \in \mathbb{R}^d \times \{0, 1\}$

Perceptron with feature map $\vec{\varphi} \colon \mathbb{R}^d \to \mathbb{R}^D$

- Initialize $\vec{w} := \vec{0} \in \mathbb{R}^D$
- Loop:
  - Pick any example $(\vec{x}_i, y_i)$ that is misclassified by $\vec{w}$
  - (If there is no such example, halt and return $\vec{w}$)
  - Update $\vec{w}$:

$$\vec{w} := \begin{cases} \vec{w} + \vec{\varphi}(\vec{x}_i) & \text{if } y_i = 1 \\ \vec{w} - \vec{\varphi}(\vec{x}_i) & \text{if } y_i = 0 \end{cases}$$

- Initialize $\alpha_i := 0$ for all $i = 1, \ldots, n$

Perceptron with feature expansion

Given: Training data $(\vec{x}_1, y_1), \dots, (\vec{x}_n, y_n) \in \mathbb{R}^d \times \{0,1\}$

Perceptron with feature map $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$

- Initialize $\vec{w} := \vec{0} \in \mathbb{R}^D$
- Loop:
  - Pick any example $(\vec{x}_i, y_i)$ that is misclassified by $\vec{w}$
  - (If there is no such example, halt and return $\vec{w}$)
  - Update $\vec{w}$:

$$\vec{w} := \begin{cases} \vec{w} + \vec{\varphi}(\vec{x}_i) & \text{if } y_i = 1 \\ \vec{w} - \vec{\varphi}(\vec{x}_i) & \text{if } y_i = 0 \end{cases}$$

- Initialize $\alpha_i := 0$ for all $i = 1, \dots, n$
- Update with example $(\vec{x}_i, y_i)$:

$$\alpha_i := \begin{cases} \alpha_i + 1 & \text{if } y_i = 1 \\ \alpha_i - 1 & \text{if } y_i = 0 \end{cases}$$

Ridge regression with feature expansion

Given: Training data $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$

$$A := \begin{bmatrix} \longleftarrow & \vec{\varphi}(\vec{x}_1)^\mathsf{T} & \longrightarrow \\ & \vdots & \\ \longleftarrow & \vec{\varphi}(\vec{x}_n)^\mathsf{T} & \longrightarrow \end{bmatrix} \in \mathbb{R}^{n \times D}, \qquad \vec{b} := \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^n$$

Ridge regression with feature expansion

Given: Training data $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$

$$A := \begin{bmatrix} \longleftarrow & \vec{\varphi}(\vec{x}_1)^\mathsf{T} & \longrightarrow \\ & \vdots & \\ \longleftarrow & \vec{\varphi}(\vec{x}_n)^\mathsf{T} & \longrightarrow \end{bmatrix} \in \mathbb{R}^{n \times D}, \qquad \vec{b} := \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^n$$

Fact: $(A^\mathsf{T} A + \lambda I)^{-1} A^\mathsf{T} = A^\mathsf{T}(AA^\mathsf{T} + \lambda I)^{-1}$ for any $\lambda > 0$

Ridge regression with feature expansion

Given: Training data $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$

$$A := \begin{bmatrix} \longleftarrow & \vec{\varphi}(\vec{x}_1)^\mathsf{T} & \longrightarrow \\ & \vdots & \\ \longleftarrow & \vec{\varphi}(\vec{x}_n)^\mathsf{T} & \longrightarrow \end{bmatrix} \in \mathbb{R}^{n \times D}, \qquad \vec{b} := \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^n$$

Fact: $(A^\mathsf{T} A + \lambda I)^{-1} A^\mathsf{T} = A^\mathsf{T} (A A^\mathsf{T} + \lambda I)^{-1}$ for any $\lambda > 0$

Therefore, ridge regression solution $\vec{w}$ can be written as

$$\vec{w} \; = \; A^\mathsf{T} \underbrace{(A A^\mathsf{T} + \lambda I)^{-1} \vec{b}}_{\vec{\alpha}} \; = \; \sum_{i=1}^{n} \alpha_i \, \vec{\varphi}(\vec{x}_i)$$

Ridge regression with feature expansion

Given: Training data $(\vec{x}_1, y_1), \ldots, (\vec{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$

$$A := \begin{bmatrix} \longleftarrow & \vec{\varphi}(\vec{x}_1)^\mathsf{T} & \longrightarrow \\ & \vdots & \\ \longleftarrow & \vec{\varphi}(\vec{x}_n)^\mathsf{T} & \longrightarrow \end{bmatrix} \in \mathbb{R}^{n \times D}, \qquad \vec{b} := \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^n$$

Fact: $(A^\mathsf{T} A + \lambda I)^{-1} A^\mathsf{T} = A^\mathsf{T} (A A^\mathsf{T} + \lambda I)^{-1}$ for any $\lambda > 0$

Therefore, ridge regression solution $\vec{w}$ can be written as

$$\vec{w} \ = \ A^\mathsf{T} \underbrace{(A A^\mathsf{T} + \lambda I)^{-1} \vec{b}}_{\vec{\alpha}} \ = \ \sum_{i=1}^{n} \alpha_i \, \vec{\varphi}(\vec{x}_i)$$

Moreover, matrix $K := A A^\mathsf{T} \in \mathbb{R}^{n \times n}$ is matrix of inner products (a.k.a. **Gram matrix**)

$$K_{i,j} = \vec{\varphi}(\vec{x}_i) \cdot \vec{\varphi}(\vec{x}_j)$$

Ridge regression with feature expansion

Given: Training data $(\vec{x}_1, y_1), \dots, (\vec{x}_n, y_n) \in \mathbb{R}^d \times \mathbb{R}$

$$A := \begin{bmatrix} \longleftarrow & \vec{\varphi}(\vec{x}_1)^\mathsf{T} & \longrightarrow \\ & \vdots & \\ \longleftarrow & \vec{\varphi}(\vec{x}_n)^\mathsf{T} & \longrightarrow \end{bmatrix} \in \mathbb{R}^{n \times D}, \qquad \vec{b} := \begin{bmatrix} y_1 \\ \vdots \\ y_n \end{bmatrix} \in \mathbb{R}^n$$

Fact: $(A^\mathsf{T} A + \lambda I)^{-1} A^\mathsf{T} = A^\mathsf{T} (A A^\mathsf{T} + \lambda I)^{-1}$ for any $\lambda > 0$

Therefore, ridge regression solution $\vec{w}$ can be written as

$$\vec{w} \;=\; A^\mathsf{T} \underbrace{(A A^\mathsf{T} + \lambda I)^{-1} \vec{b}}_{\vec{\alpha}} \;=\; \sum_{i=1}^{n} \alpha_i \, \vec{\varphi}(\vec{x}_i)$$

Moreover, matrix $K := A A^\mathsf{T} \in \mathbb{R}^{n \times n}$ is matrix of inner products (a.k.a. **Gram matrix**)

$$K_{i,j} = \vec{\varphi}(\vec{x}_i) \cdot \vec{\varphi}(\vec{x}_j)$$

So, compute Gram matrix K and solve linear system $(K + \lambda I)\vec{\alpha} = \vec{b}$ for $\vec{\alpha}$

Computation time

Comparing computational costs when using degree-k polynomial expansion (assume $d \ll n \ll d^k$):

Computation time

Comparing computational costs when using degree-k polynomial expansion (assume $d \ll n \ll d^k$):

- Ridge regression with explicit feature expansion
  - Solving for $\vec{w}$: $O(n^2 d^k)$ time
  - Each prediction: $O(d^k)$ time

Computation time

Comparing computational costs when using degree-k polynomial expansion (assume $d \ll n \ll d^k$):

- Ridge regression with explicit feature expansion
  - Solving for $\vec{w}$: $O(n^2 d^k)$ time
  - Each prediction: $O(d^k)$ time
- Ridge regression using implicit representation of $\vec{w}$:
  - Solving for $\vec{\alpha}$: $O(n^3)$ time
  - Each prediction: $O(nd)$ time

Kernels

- Many other feature expansions (besides polynomial expansion) have a similar computational trick to compute dot products

Kernels

- Many other feature expansions (besides polynomial expansion) have a similar computational trick to compute dot products
- Conversely, many easy-to-compute "similarity functions" $\mathsf{k}(\vec{x}, \vec{z})$ are, in fact, the dot product between certain feature expansions $\vec{\varphi}(\vec{x})$ and $\vec{\varphi}(\vec{z})$

Kernels

- Many other feature expansions (besides polynomial expansion) have a similar computational trick to compute dot products
- Conversely, many easy-to-compute "similarity functions" $\mathsf{k}(\vec{x}, \vec{z})$ are, in fact, the dot product between certain feature expansions $\vec{\varphi}(\vec{x})$ and $\vec{\varphi}(\vec{z})$
  - Such similarity functions are called **(positive definite) kernels**

Kernels

- Many other feature expansions (besides polynomial expansion) have a similar computational trick to compute dot products
- Conversely, many easy-to-compute "similarity functions" $\mathsf{k}(\vec{x}, \vec{z})$ are, in fact, the dot product between certain feature expansions $\vec{\varphi}(\vec{x})$ and $\vec{\varphi}(\vec{z})$
  - Such similarity functions are called **(positive definite) kernels**
  - E.g., **Gaussian kernel**

$$\mathsf{k}(\vec{x}, \vec{z}) = \exp\left(-\frac{\|\vec{x} - \vec{z}\|_2^2}{2\sigma^2}\right)$$

    where $\sigma > 0$ is the "bandwidth" of the kernel (a hyperparameter)

Kernels

- Many other feature expansions (besides polynomial expansion) have a similar computational trick to compute dot products
- Conversely, many easy-to-compute "similarity functions" $\mathsf{k}(\vec{x}, \vec{z})$ are, in fact, the dot product between certain feature expansions $\vec{\varphi}(\vec{x})$ and $\vec{\varphi}(\vec{z})$
  - Such similarity functions are called **(positive definite) kernels**
  - E.g., **Gaussian kernel**

$$\mathsf{k}(\vec{x}, \vec{z}) = \exp\left(-\frac{\|\vec{x} - \vec{z}\|_2^2}{2\sigma^2}\right)$$

    where $\sigma > 0$ is the "bandwidth" of the kernel (a hyperparameter)
  - Technically, the feature expansion $\vec{\varphi} \colon \mathbb{R}^d \to \mathbb{R}^D$ may need $D = \infty$

Kernels

- Many other feature expansions (besides polynomial expansion) have a similar computational trick to compute dot products
- Conversely, many easy-to-compute "similarity functions" $\mathsf{k}(\vec{x}, \vec{z})$ are, in fact, the dot product between certain feature expansions $\vec{\varphi}(\vec{x})$ and $\vec{\varphi}(\vec{z})$
  - Such similarity functions are called **(positive definite) kernels**
  - E.g., **Gaussian kernel**

    $$\mathsf{k}(\vec{x}, \vec{z}) = \exp\left(-\frac{\|\vec{x} - \vec{z}\|_2^2}{2\sigma^2}\right)$$

    where $\sigma > 0$ is the "bandwidth" of the kernel (a hyperparameter)
  - Technically, the feature expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ may need $D = \infty$
    Not a problem when using **kernel methods**
    (i.e., versions of ridge regression, Perceptron, etc. that only compute $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$)

Kernels

- Many other feature expansions (besides polynomial expansion) have a similar computational trick to compute dot products
- Conversely, many easy-to-compute "similarity functions" $\mathsf{k}(\vec{x}, \vec{z})$ are, in fact, the dot product between certain feature expansions $\vec{\varphi}(\vec{x})$ and $\vec{\varphi}(\vec{z})$
  - Such similarity functions are called **(positive definite) kernels**
  - E.g., **Gaussian kernel**

$$\mathsf{k}(\vec{x}, \vec{z}) = \exp\left(-\frac{\|\vec{x} - \vec{z}\|_2^2}{2\sigma^2}\right)$$

    where $\sigma > 0$ is the "bandwidth" of the kernel (a hyperparameter)
  - Technically, the feature expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$ may need $D = \infty$
    Not a problem when using **kernel methods**
    (i.e., versions of ridge regression, Perceptron, etc. that only compute $\vec{\alpha} = (\alpha_1, \ldots, \alpha_n)$)
- Resulting predictors with implicit representation

$$g(\vec{x}) = \sum_{i=1}^{n} \alpha_i \, \mathsf{k}(\vec{x}, \vec{x}_i)$$

  are called **kernel machines**

Neural networks

What else is there besides polynomials?

Weierstrass approximation theorem:

- Can approximate any continuous function using polynomials provided degree is high enough

What else is there besides polynomials?

Weierstrass approximation theorem:

- Can approximate any continuous function using polynomials provided degree is high enough

Stone-Weierstrass approximation theorem: (via Hornik, Stinchcombe, & White, 1989)

- Can approximate any continuous function using functions of form

$$g(\vec{x}) = \sum_{i=1}^{D} \alpha_i \exp(\vec{x} \cdot \vec{w}_i)$$

provided D is large enough

What else is there besides polynomials?

Weierstrass approximation theorem:

- Can approximate any continuous function using polynomials provided degree is high enough

Stone-Weierstrass approximation theorem: (via Hornik, Stinchcombe, & White, 1989)

- Can approximate any continuous function using functions of form

$$g(\vec{x}) = \sum_{i=1}^{D} \alpha_i \exp(\vec{x} \cdot \vec{w}_i)$$

provided D is large enough

  - Both the α_i's and $\vec{w}_i$'s may need to depend on the target function

What else is there besides polynomials?

Weierstrass approximation theorem:

- Can approximate any continuous function using polynomials provided degree is high enough

Stone-Weierstrass approximation theorem: (via Hornik, Stinchcombe, & White, 1989)

- Can approximate any continuous function using functions of form

$$g(\vec{x}) = \sum_{i=1}^{D} \alpha_i \exp(\vec{x} \cdot \vec{w}_i)$$

  provided D is large enough

  - Both the α_i's and $\vec{w}_i$'s may need to depend on the target function
  - Can replace $\exp$ with other "activation functions" and approximation property still holds

What else is there besides polynomials?

Weierstrass approximation theorem:

- Can approximate any continuous function using polynomials provided degree is high enough

Stone-Weierstrass approximation theorem: (via Hornik, Stinchcombe, & White, 1989)

- Can approximate any continuous function using functions of form

$$g(\vec{x}) = \sum_{i=1}^{D} \alpha_i \exp(\vec{x} \cdot \vec{w}_i)$$

  provided D is large enough

  - Both the α_i's and $\vec{w}_i$'s may need to depend on the target function
  - Can replace $\exp$ with other "activation functions" and approximation property still holds

- **Another interpretation:** Can approximate any continuous function by linear function with feature expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$

$$\vec{\varphi}(\vec{x}) = (\exp(\vec{x} \cdot \vec{w}_1), \ldots, \exp(\vec{x} \cdot \vec{w}_D))$$

  provided D is large enough and $\vec{\varphi}$'s "parameters" $\vec{w}_1, \ldots, \vec{w}_D$ may depend on target function

What else is there besides polynomials?

Weierstrass approximation theorem:

- Can approximate any continuous function using polynomials provided degree is high enough

Stone-Weierstrass approximation theorem: (via Hornik, Stinchcombe, & White, 1989)

- Can approximate any continuous function using functions of form

$$g(\vec{x}) = \sum_{i=1}^{D} \alpha_i \exp(\vec{x} \cdot \vec{w}_i)$$

  provided D is large enough

  - Both the α_i's and $\vec{w}_i$'s may need to depend on the target function
  - Can replace $\exp$ with other "activation functions" and approximation property still holds

- **Another interpretation:** Can approximate any continuous function by linear function with feature expansion $\vec{\varphi}\colon \mathbb{R}^d \to \mathbb{R}^D$

$$\vec{\varphi}(\vec{x}) = (\exp(\vec{x} \cdot \vec{w}_1), \ldots, \exp(\vec{x} \cdot \vec{w}_D))$$

  provided D is large enough and $\vec{\varphi}$'s "parameters" $\vec{w}_1, \ldots, \vec{w}_D$ may depend on target function

- Called a **neural network**

Kernel machines vs. neural networks

Kernel machine (with kernel k)

$$g(\vec{x}) = \sum_{i=1}^{n} \alpha_i \ \mathsf{k}(\vec{x}, \vec{x}_i)$$

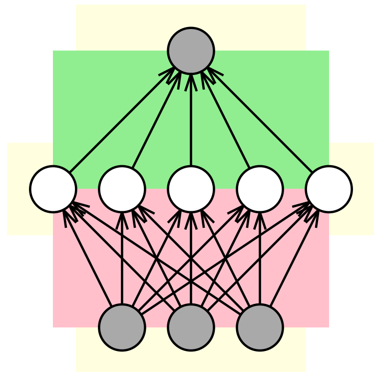
Only α_i's are learned

Neural network (with $\exp$ activation)

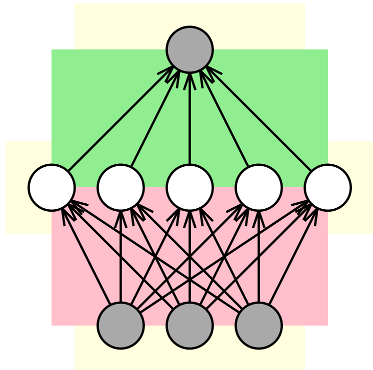
$$g(\vec{x}) = \sum_{i=1}^{D} \alpha_i \ \exp(\vec{x} \cdot \vec{w}_i)$$

Both α_i's and $\vec{w}_i$'s are learned
Can use $D > n$

Anatomy of a neural network

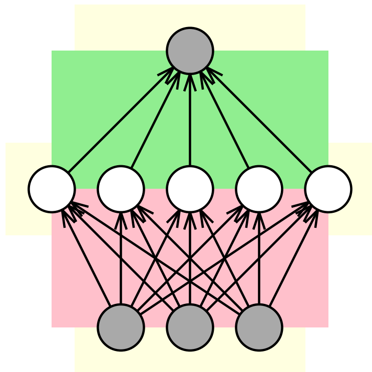


Anatomy of a neural network



- **Bottom layer:** input $\vec{x} = (x_1, \ldots, x_d)$ to function

Anatomy of a neural network

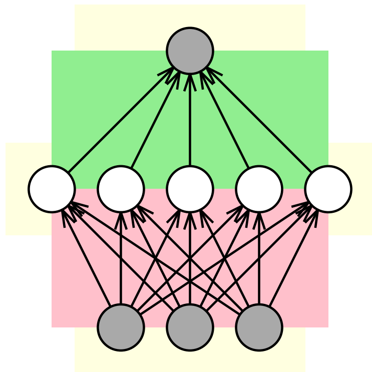


- **Middle layer: hidden units** (a.k.a. **neurons**)
  Each hidden unit computes composition of **activation function** σ_i with affine function of input

$$h_i(\vec{x}) = \sigma_i(\vec{x} \cdot \vec{w}_i + b_i)$$

- **Bottom layer:** input $\vec{x} = (x_1, \ldots, x_d)$ to function

Anatomy of a neural network



- **Top layer:** output of function

  Output is affine combination of hidden units

$$g(\vec{x}) = \sum_{i=1}^{D} \alpha_i \, h_i(\vec{x}) + \alpha_0$$

  (Sometimes also apply an activation function to output)

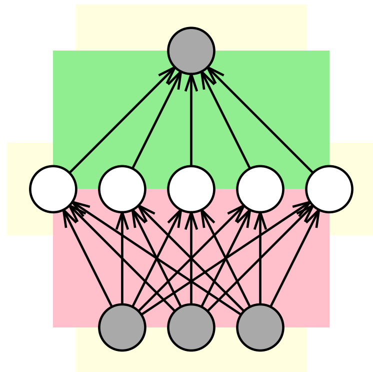
- **Middle layer:** **hidden units** (a.k.a. **neurons**)

  Each hidden unit computes composition of **activation function** σ_i with affine function of input

$$h_i(\vec{x}) = \sigma_i(\vec{x} \cdot \vec{w}_i + b_i)$$

- **Bottom layer:** input $\vec{x} = (x_1, \ldots, x_d)$ to function

Anatomy of a neural network



- **Top layer:** output of function

  Output is affine combination of hidden units

  $$g(\vec{x}) = \sum_{i=1}^{D} \alpha_i \, h_i(\vec{x}) + \alpha_0$$

  (Sometimes also apply an activation function to output)

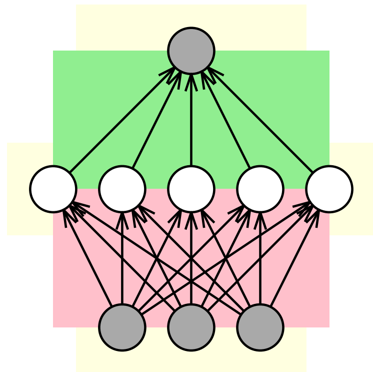
- **Middle layer:** **hidden units** (a.k.a. **neurons**)

  Each hidden unit computes composition of **activation function** σ_i with affine function of input

  $$h_i(\vec{x}) = \sigma_i(\vec{x} \cdot \vec{w}_i + b_i)$$

- **Bottom layer:** input $\vec{x} = (x_1, \ldots, x_d)$ to function
- Arrows in diagram depict functional dependence

Anatomy of a neural network



- **Top layer:** output of function
  Output is affine combination of hidden units

$$g(\vec{x}) = \sum_{i=1}^{D} \alpha_i \, h_i(\vec{x}) + \alpha_0$$

  (Sometimes also apply an activation function to output)
- **Middle layer:** **hidden units** (a.k.a. **neurons**)
  Each hidden unit computes composition of **activation function** σ_i with affine function of input

$$h_i(\vec{x}) = \sigma_i(\vec{x} \cdot \vec{w}_i + b_i)$$

- **Bottom layer:** input $\vec{x} = (x_1, \ldots, x_d)$ to function
- Arrows in diagram depict functional dependence
- **Parameters:** $\vec{w}_i$'s, b_i's, and α_i's

Neural network as a straight-line program

(Generalized) straight-line program that implements the neural network function:

$$
\begin{aligned}
u_1 &:= \vec{x} \cdot \vec{w}_1 + b_1 \\
v_1 &:= \sigma_1(u_1) \\
u_2 &:= \vec{x} \cdot \vec{w}_2 + b_2 \\
v_2 &:= \sigma_2(u_2) \\
&\vdots \\
u_D &:= \vec{x} \cdot \vec{w}_D + b_D \\
v_D &:= \sigma_D(u_D) \\
\mathrm{out} &:= \alpha_1 \times v_1 + \cdots + \alpha_D \times v_D + \alpha_0
\end{aligned}
$$

Neural network as a straight-line program

(Generalized) straight-line program that implements the neural network function:

$$
\begin{aligned}
u_1 &:= \vec{x} \cdot \vec{w}_1 + b_1 \\
v_1 &:= \sigma_1(u_1) \\
u_2 &:= \vec{x} \cdot \vec{w}_2 + b_2 \\
v_2 &:= \sigma_2(u_2) \\
&\vdots \\
u_D &:= \vec{x} \cdot \vec{w}_D + b_D \\
v_D &:= \sigma_D(u_D) \\
\text{out} &:= \alpha_1 \times v_1 + \cdots + \alpha_D \times v_D + \alpha_0
\end{aligned}
$$

(This is useful for the “forward pass” in autodiff!)

Neural network as a straight-line program, again

(Generalized) straight-line program that implements the neural network function:

$$\begin{aligned} \vec{u} &:= W\vec{x} + \vec{b} \\ \vec{v} &:= \sigma(\vec{u}) \\ \mathrm{out} &:= \vec{\alpha} \cdot \vec{v} + \alpha_0 \end{aligned}$$

Parameters: $W = [\vec{w}_1 | \cdots | \vec{w}_D]^\mathsf{T} \in \mathbb{R}^{D \times d}$, $\vec{b} = (b_1, \ldots, b_D) \in \mathbb{R}^D$, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_D) \in \mathbb{R}^D$, $\alpha_0 \in \mathbb{R}$
In code above, vector-valued activation function $\sigma \colon \mathbb{R}^D \to \mathbb{R}^D$ applies σ_i to i-th coordinate of input

Neural network as a straight-line program, again

(Generalized) straight-line program that implements the neural network function:

$$\begin{aligned} \vec{u} &:= W\vec{x} + \vec{b} \\ \vec{v} &:= \sigma(\vec{u}) \\ \text{out} &:= \vec{\alpha} \cdot \vec{v} + \alpha_0 \end{aligned}$$

Parameters: $W = [\vec{w}_1 | \cdots | \vec{w}_D]^\mathsf{T} \in \mathbb{R}^{D \times d}$, $\vec{b} = (b_1, \ldots, b_D) \in \mathbb{R}^D$, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_D) \in \mathbb{R}^D$, $\alpha_0 \in \mathbb{R}$
In code above, vector-valued activation function $\sigma \colon \mathbb{R}^D \to \mathbb{R}^D$ applies σ_i to i-th coordinate of input

Some other common activation functions:

Neural network as a straight-line program, again

(Generalized) straight-line program that implements the neural network function:

$$\begin{aligned}\vec{u} &:= W\vec{x} + \vec{b} \\ \vec{v} &:= \sigma(\vec{u}) \\ \text{out} &:= \vec{\alpha} \cdot \vec{v} + \alpha_0\end{aligned}$$

Parameters: $W = [\vec{w}_1 | \cdots | \vec{w}_D]^\mathsf{T} \in \mathbb{R}^{D \times d}$, $\vec{b} = (b_1, \ldots, b_D) \in \mathbb{R}^D$, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_D) \in \mathbb{R}^D$, $\alpha_0 \in \mathbb{R}$
In code above, vector-valued activation function $\sigma \colon \mathbb{R}^D \to \mathbb{R}^D$ applies σ_i to i-th coordinate of input

Some other common activation functions:

- Heaviside (a.k.a. step function): $\sigma_i(z) = \mathbb{1}\{z > 0\}$ (popular in 1940s)

Neural network as a straight-line program, again

(Generalized) straight-line program that implements the neural network function:

$$\vec{u} := W\vec{x} + \vec{b}$$
$$\vec{v} := \sigma(\vec{u})$$
$$\text{out} := \vec{\alpha} \cdot \vec{v} + \alpha_0$$

Parameters: $W = [\vec{w}_1 | \cdots | \vec{w}_D]^\mathsf{T} \in \mathbb{R}^{D \times d}$, $\vec{b} = (b_1, \ldots, b_D) \in \mathbb{R}^D$, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_D) \in \mathbb{R}^D$, $\alpha_0 \in \mathbb{R}$
In code above, vector-valued activation function $\sigma \colon \mathbb{R}^D \to \mathbb{R}^D$ applies σ_i to i-th coordinate of input

Some other common activation functions:

- Heaviside (a.k.a. step function): $\sigma_i(z) = \mathbb{1}\{z > 0\}$ (popular in 1940s)
- Sigmoid (a.k.a. logistic): $\sigma_i(z) = 1/(1 + e^{-z})$ (popular since 1970s)

Neural network as a straight-line program, again

(Generalized) straight-line program that implements the neural network function:

$$\begin{aligned} \vec{u} &:= W\vec{x} + \vec{b} \\ \vec{v} &:= \sigma(\vec{u}) \\ \text{out} &:= \vec{\alpha} \cdot \vec{v} + \alpha_0 \end{aligned}$$

Parameters: $W = [\vec{w}_1 | \cdots | \vec{w}_D]^\intercal \in \mathbb{R}^{D \times d}$, $\vec{b} = (b_1, \ldots, b_D) \in \mathbb{R}^D$, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_D) \in \mathbb{R}^D$, $\alpha_0 \in \mathbb{R}$
In code above, vector-valued activation function $\sigma \colon \mathbb{R}^D \to \mathbb{R}^D$ applies σ_i to i-th coordinate of input

Some other common activation functions:

- Heaviside (a.k.a. step function): $\sigma_i(z) = \mathbb{1}\{z > 0\}$ (popular in 1940s)
- Sigmoid (a.k.a. logistic): $\sigma_i(z) = 1/(1 + e^{-z})$ (popular since 1970s)
- Rectified Linear Unit (ReLU): $\sigma_i(z) = \max\{0, z\}$ (popular since 2012)

Neural network as a straight-line program, again

(Generalized) straight-line program that implements the neural network function:

$$\vec{u} := W\vec{x} + \vec{b}$$
$$\vec{v} := \sigma(\vec{u})$$
$$\text{out} := \vec{\alpha} \cdot \vec{v} + \alpha_0$$

Parameters: $W = [\vec{w}_1 | \cdots | \vec{w}_D]^\mathsf{T} \in \mathbb{R}^{D \times d}$, $\vec{b} = (b_1, \ldots, b_D) \in \mathbb{R}^D$, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_D) \in \mathbb{R}^D$, $\alpha_0 \in \mathbb{R}$
In code above, vector-valued activation function $\sigma \colon \mathbb{R}^D \to \mathbb{R}^D$ applies σ_i to i-th coordinate of input

Some other common activation functions:

- Heaviside (a.k.a. step function): $\sigma_i(z) = \mathbb{1}\{z > 0\}$ (popular in 1940s)
- Sigmoid (a.k.a. logistic): $\sigma_i(z) = 1/(1 + e^{-z})$ (popular since 1970s)
- Rectified Linear Unit (ReLU): $\sigma_i(z) = \max\{0, z\}$ (popular since 2012)
- "Softmax": $\sigma(\vec{z}) = (e^{z_1}, \ldots, e^{z_D}) / \sum_{i=1}^{D} e^{z_i}$ ($\mathbb{R}^D \to \mathbb{R}^D$; terrible naming choice)

Neural network as a straight-line program, again

(Generalized) straight-line program that implements the neural network function:

$$\begin{aligned} \vec{u} &:= W\vec{x} + \vec{b} \\ \vec{v} &:= \sigma(\vec{u}) \\ \text{out} &:= \vec{\alpha} \cdot \vec{v} + \alpha_0 \end{aligned}$$

Parameters: $W = [\vec{w}_1 | \cdots | \vec{w}_D]^\mathsf{T} \in \mathbb{R}^{D \times d}$, $\vec{b} = (b_1, \ldots, b_D) \in \mathbb{R}^D$, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_D) \in \mathbb{R}^D$, $\alpha_0 \in \mathbb{R}$
In code above, vector-valued activation function $\sigma \colon \mathbb{R}^D \to \mathbb{R}^D$ applies σ_i to i-th coordinate of input

Some other common activation functions:

- Heaviside (a.k.a. step function): $\sigma_i(z) = \mathbb{1}\{z > 0\}$ (popular in 1940s)
- Sigmoid (a.k.a. logistic): $\sigma_i(z) = 1/(1 + e^{-z})$ (popular since 1970s)
- Rectified Linear Unit (ReLU): $\sigma_i(z) = \max\{0, z\}$ (popular since 2012)
- "Softmax": $\sigma(\vec{z}) = (e^{z_1}, \ldots, e^{z_D}) / \sum_{i=1}^{D} e^{z_i}$ ($\mathbb{R}^D \to \mathbb{R}^D$; terrible naming choice)
- Max pooling: $\sigma(\vec{z}) = \max\{z_1, \ldots, z_D\}$ ($\mathbb{R}^D \to \mathbb{R}^1$; popular in computer vision)

Neural network as a straight-line program, again

(Generalized) straight-line program that implements the neural network function:

$$\vec{u} := W\vec{x} + \vec{b}$$
$$\vec{v} := \sigma(\vec{u})$$
$$\text{out} := \vec{\alpha} \cdot \vec{v} + \alpha_0$$

Parameters: $W = [\vec{w}_1 | \cdots | \vec{w}_D]^\mathsf{T} \in \mathbb{R}^{D \times d}$, $\vec{b} = (b_1, \ldots, b_D) \in \mathbb{R}^D$, $\vec{\alpha} = (\alpha_1, \ldots, \alpha_D) \in \mathbb{R}^D$, $\alpha_0 \in \mathbb{R}$
In code above, vector-valued activation function $\sigma \colon \mathbb{R}^D \to \mathbb{R}^D$ applies σ_i to i-th coordinate of input

Some other common activation functions:

- Heaviside (a.k.a. step function): $\sigma_i(z) = \mathbb{1}\{z > 0\}$ (popular in 1940s)
- Sigmoid (a.k.a. logistic): $\sigma_i(z) = 1/(1 + e^{-z})$ (popular since 1970s)
- Rectified Linear Unit (ReLU): $\sigma_i(z) = \max\{0, z\}$ (popular since 2012)
- "Softmax": $\sigma(\vec{z}) = (e^{z_1}, \ldots, e^{z_D}) / \sum_{i=1}^{D} e^{z_i}$ ($\mathbb{R}^D \to \mathbb{R}^D$; terrible naming choice)
- Max pooling: $\sigma(\vec{z}) = \max\{z_1, \ldots, z_D\}$ ($\mathbb{R}^D \to \mathbb{R}^1$; popular in computer vision)
- Identity: $\sigma_i(z) = z$ (you might be surprised . . .)

More neural networks

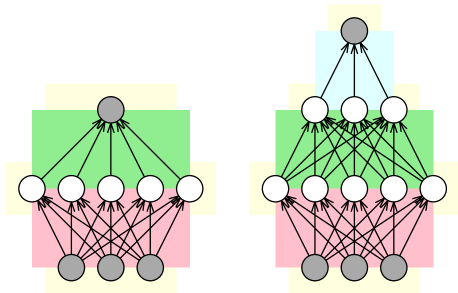
Modern lingo:

- Parameterized function = “neural network”
- Function template = “**architecture**”

More neural networks

Modern lingo:

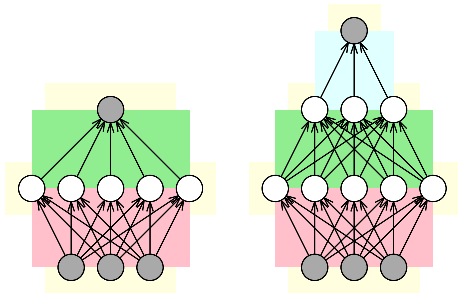
- Parameterized function = "neural network"
- Function template = "**architecture**"



More neural networks

Modern lingo:

- Parameterized function = "neural network"
- Function template = "**architecture**"

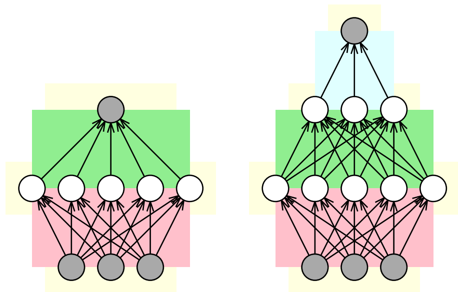


Define architecture directly with code!

More neural networks

Modern lingo:

- Parameterized function = "neural network"
- Function template = "**architecture**"



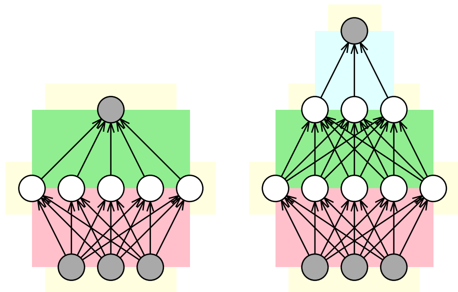
Define architecture directly with code!

Second one just needs a few extra lines . . .

More neural networks

Modern lingo:

- Parameterized function = "neural network"
- Function template = "**architecture**"



Define architecture directly with code!

Second one just needs a few extra lines ...

To approximate certain functions, may be more size-economical to use multiple layers of hidden units

Fitting neural networks to data

Generic strategy:

- Goal: For $(\vec{X}, Y) \sim P$,

$$\min_{f \colon \mathbb{R}^d \to \mathbb{R}} \mathbb{E}[\ell(Y, f(\vec{X}))]$$

  where $\ell(y, p)$ is loss function (e.g., square loss, zero-one loss, logarithmic loss)

Fitting neural networks to data

Generic strategy:

- Goal: For $(\vec{X}, Y) \sim P$,

$$\min_{f \colon \mathbb{R}^d \to \mathbb{R}} \mathbb{E}[\ell(Y, f(\vec{X}))]$$

  where $\ell(y, p)$ is loss function (e.g., square loss, zero-one loss, logarithmic loss)
- Assume IID model for training data

Fitting neural networks to data

Generic strategy:

- Goal: For $(\vec{X}, Y) \sim P$,

$$\min_{f \colon \mathbb{R}^d \to \mathbb{R}} \mathbb{E}[\ell(Y, f(\vec{X}))]$$

  where $\ell(y, p)$ is loss function (e.g., square loss, zero-one loss, logarithmic loss)
- Assume IID model for training data
- Use empirical distribution P_n as plug-in estimate of P to get training objective J
  (If original objective uses zero-one loss, replace with a *differentiable* surrogate loss)

Fitting neural networks to data

Generic strategy:

- Goal: For $(\vec{X}, Y) \sim P$,

$$\min_{f \colon \mathbb{R}^d \to \mathbb{R}} \mathbb{E}[\ell(Y, f(\vec{X}))]$$

  where $\ell(y, p)$ is loss function (e.g., square loss, zero-one loss, logarithmic loss)
- Assume IID model for training data
- Use empirical distribution P_n as plug-in estimate of P to get training objective J
  (If original objective uses zero-one loss, replace with a *differentiable* surrogate loss)
- Change "$\min$ over *all functions* $f \colon \mathbb{R}^d \to \mathbb{R}$" to "$\min$ over neural network functions"

Fitting neural networks to data

Generic strategy:

- Goal: For $(\vec{X}, Y) \sim P$,

  $$\min_{f\colon \mathbb{R}^d \to \mathbb{R}} \mathbb{E}[\ell(Y, f(\vec{X}))]$$

  where $\ell(y, p)$ is loss function (e.g., square loss, zero-one loss, logarithmic loss)
- Assume IID model for training data
- Use empirical distribution P_n as plug-in estimate of P to get training objective J
  (If original objective uses zero-one loss, replace with a *differentiable* surrogate loss)
- Change "$\min$ over *all functions* $f\colon \mathbb{R}^d \to \mathbb{R}$" to "$\min$ over neural network functions"
- Possibly incorporate regularization into J
  (E.g., data augmentation)

Fitting neural networks to data

Generic strategy:

- Goal: For $(\vec{X}, Y) \sim P$,

$$\min_{f \colon \mathbb{R}^d \to \mathbb{R}} \mathbb{E}[\ell(Y, f(\vec{X}))]$$

  where $\ell(y, p)$ is loss function (e.g., square loss, zero-one loss, logarithmic loss)
- Assume IID model for training data
- Use empirical distribution P_n as plug-in estimate of P to get training objective J
  (If original objective uses zero-one loss, replace with a *differentiable* surrogate loss)
- Change "$\min$ over *all functions* $f \colon \mathbb{R}^d \to \mathbb{R}$" to "$\min$ over neural network functions"
- Possibly incorporate regularization into J
  (E.g., data augmentation)
- Attempt to minimize J with respect to neural network parameters (e.g., via SGD)
  (Autodiff is very helpful here!)

Fitting neural networks to data

Generic strategy:

- Goal: For $(\vec{X}, Y) \sim P$,

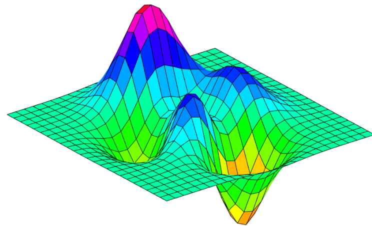
$$\min_{f \colon \mathbb{R}^d \to \mathbb{R}} \mathbb{E}[\ell(Y, f(\vec{X}))]$$

  where $\ell(y, p)$ is loss function (e.g., square loss, zero-one loss, logarithmic loss)
- Assume IID model for training data
- Use empirical distribution P_n as plug-in estimate of P to get training objective J
  (If original objective uses zero-one loss, replace with a *differentiable* surrogate loss)
- Change "$\text{min over } \textit{all functions } f \colon \mathbb{R}^d \to \mathbb{R}$" to "$\text{min over neural network functions}$"
- Possibly incorporate regularization into J
  (E.g., data augmentation)
- Attempt to minimize J with respect to neural network parameters (e.g., via SGD)
  (Autodiff is very helpful here!)

Note: J is typically not convex function of neural network parameters ¯_(ツ)_/¯

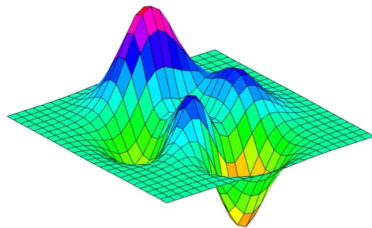
Practical issues: Optimization

- Objective is often not convex function of parameters, and often has saddle point at zero



Practical issues: Optimization

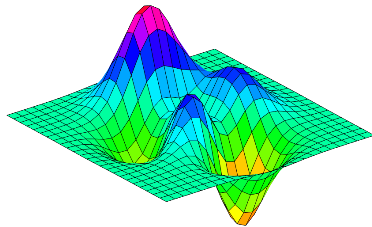
- Objective is often not convex function of parameters, and often has saddle point at zero



  - Run SGD starting from randomly chosen parameter values

Practical issues: Optimization

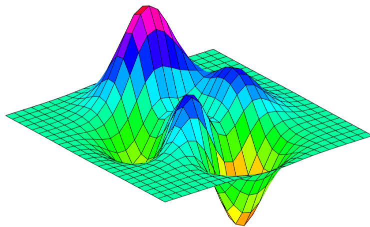
- Objective is often not convex function of parameters, and often has saddle point at zero



  - Run SGD starting from randomly chosen parameter values
  - Many heuristics available for initial parameter distribution

Practical issues: Optimization

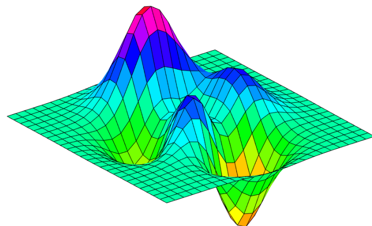
- Objective is often not convex function of parameters, and often has saddle point at zero



  - Run SGD starting from randomly chosen parameter values
  - Many heuristics available for initial parameter distribution
  - These heuristics often rely on inputs $\vec{x}$ being standardized and/or uncorrelated (PCA can help!)

Practical issues: Optimization

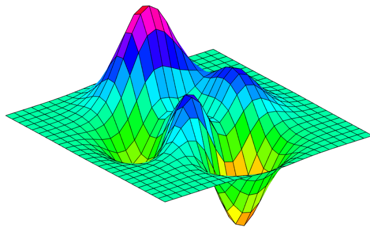
- Objective is often not convex function of parameters, and often has saddle point at zero



  - Run SGD starting from randomly chosen parameter values
  - Many heuristics available for initial parameter distribution
  - These heuristics often rely on inputs $\vec{x}$ being standardized and/or uncorrelated (PCA can help!)
- No single step size for SGD will work well for all problems
  - Many heuristics available for choosing "schedule" of step sizes

Practical issues: Optimization

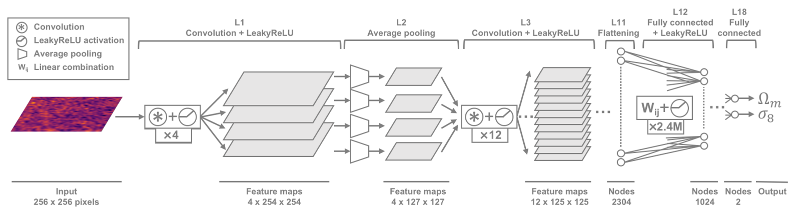
- Objective is often not convex function of parameters, and often has saddle point at zero



  - Run SGD starting from randomly chosen parameter values
  - Many heuristics available for initial parameter distribution
  - These heuristics often rely on inputs $\vec{x}$ being standardized and/or uncorrelated (PCA can help!)
- No single step size for SGD will work well for all problems
  - Many heuristics available for choosing "schedule" of step sizes
- See "Efficient BackProp" paper (LeCun, Bottou, Orr, Müller, 1998)

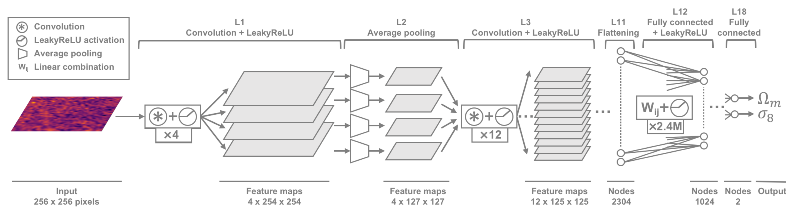
Practical issues: Architecture choice

- Easy to design and use new architectures (thanks to high-quality autodiff software)



Practical issues: Architecture choice

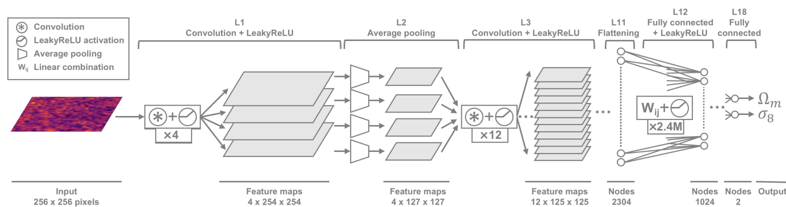
- Easy to design and use new architectures (thanks to high-quality autodiff software)



- What architecture should you use? ($\equiv$ What is the "right" inductive bias for your problem?)
  - Entire research communities (e.g., natural language processing, computer vision, compbio) devote considerable effort to finding good architectures for their problems

Practical issues: Architecture choice

- Easy to design and use new architectures (thanks to high-quality autodiff software)



- What architecture should you use? ($\equiv$ What is the "right" inductive bias for your problem?)
  - Entire research communities (e.g., natural language processing, computer vision, compbio) devote considerable effort to finding good architectures for their problems
- Some choices are driven by goal of making optimization easier
  - E.g., differentiable activation functions, very wide layers of hidden units

But no panacea

Summary

- Kernel machines and neural networks are powerful approaches to go beyond linear predictors
- In practice, more flexibility with neural networks (for better and/or for worse)
  Autodiff and autodiff software has made a big difference
- No single solution works well for all problems