DS-GA 1003: Machine Learning

Lecture 9: Neural Networks

Logistics

HW4 due on April 14th.

HW5 will be released April 14th, due April 28th. This will be the final homework!

Logistics

HW4 due on April 14th.

HW5 will be released April 14th, due April 28th. This will be the final homework!

Project proposal feedback: 79% done. We aim to release grades and feedback on project proposals by Wednesday! Everyone who submitted a project proposal will get 100%!

Final project due on May 8th. Please pay careful attention to the proposal feedback if you want to receive a high grade, and/or checkin with instructors at office hours!

Tips for the Final Project: Related Work

- You need to cite papers

Tips for the Final Project: Related Work

- You need to cite papers
- You need to cite the most important papers

Tips for the Final Project: Related Work

- You need to cite papers
- You need to cite the most important papers
- But you shouldn't just cite papers

Tips for the Final Project: Related Work

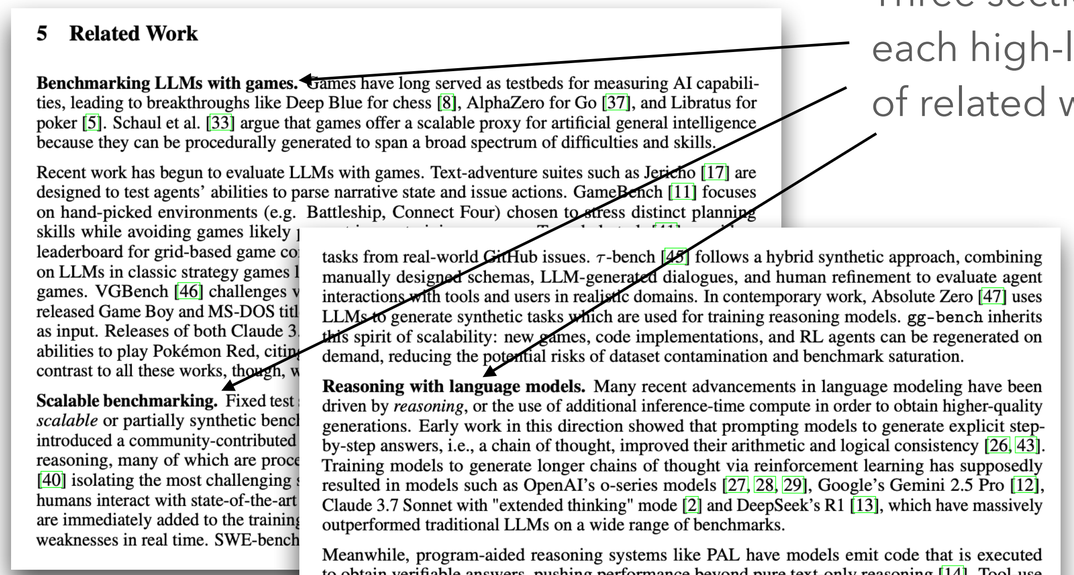
- You need to cite papers
- You need to cite the most important papers
- But you shouldn't just cite papers

- Instead: synthesize prior work, explain high-level takeaways and findings, describe their methods, and then describe how your work contrasts with or goes beyond prior work

Tips for the Final Project: Related Work

Measuring General Intelligence with Generated Games

Three sections: one for each high-level category of related work



5 Related Work

Benchmarking LLMs with games. Games have long served as testbeds for measuring AI capabilities, leading to breakthroughs like Deep Blue for chess [8], AlphaZero for Go [37], and Libratus for poker [5]. Schaul et al. [33] argue that games offer a scalable proxy for artificial general intelligence because they can be procedurally generated to span a broad spectrum of difficulties and skills.

Recent work has begun to evaluate LLMs with games. Text-adventure suites such as Jericho [17] are designed to test agents' abilities to parse narrative state and issue actions. GameBench [11] focuses on hand-picked environments (e.g. Battleship, Connect Four) chosen to stress distinct planning skills while avoiding games likely

leaderboard for grid-based game co

on LLMs in classic strategy games

games. VGBench [46] challenges

released Game Boy and MS-DOS titl

as input. Releases of both Claude 3

abilities to play Pokémon Red, citin

contrast to all these works, though,

Scalable benchmarking. Fixed test

scalable or partially synthetic bench

introduced a community-contributed

reasoning, many of which are proce

[40] isolating the most challenging

humans interact with state-of-the-art

are immediately added to the training

weaknesses in real time. SWE-bench

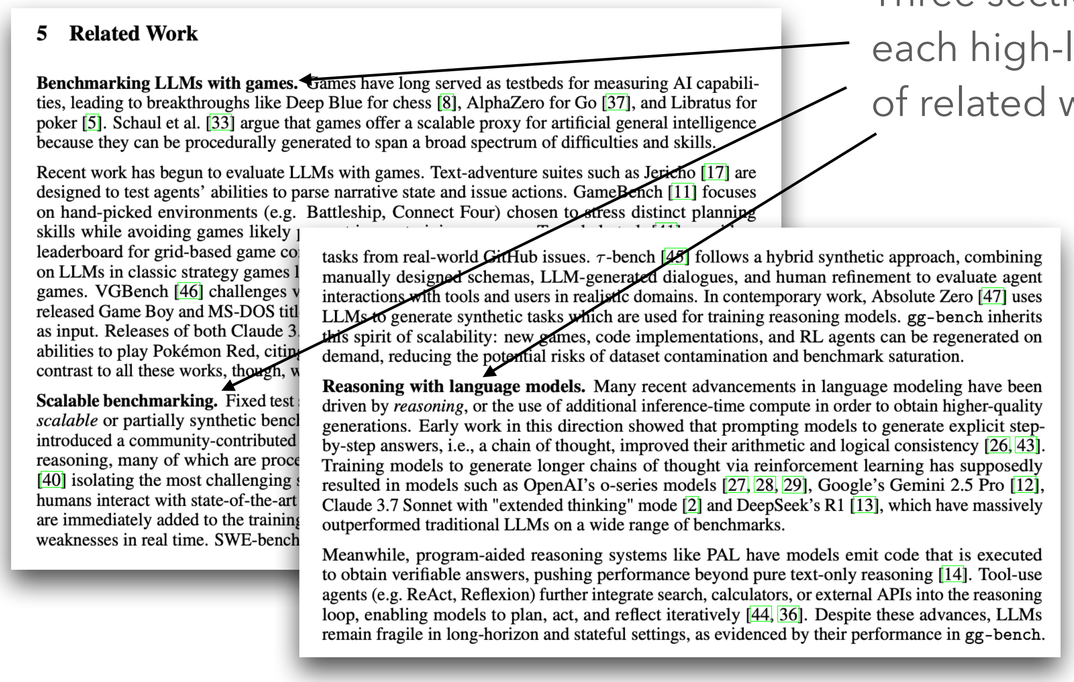
tasks from real-world GitHub issues. τ-bench [45] follows a hybrid synthetic approach, combining manually designed schemas, LLM-generated dialogues, and human refinement to evaluate agent interactions with tools and users in realistic domains. In contemporary work, Absolute Zero [47] uses LLMs to generate synthetic tasks which are used for training reasoning models. `gg-bench` inherits this spirit of scalability: new games, code implementations, and RL agents can be regenerated on demand, reducing the potential risks of dataset contamination and benchmark saturation.

Reasoning with language models. Many recent advancements in language modeling have been driven by *reasoning*, or the use of additional inference-time compute in order to obtain higher-quality generations. Early work in this direction showed that prompting models to generate explicit step-by-step answers, i.e., a chain of thought, improved their arithmetic and logical consistency [26, 43]. Training models to generate longer chains of thought via reinforcement learning has supposedly resulted in models such as OpenAI's o-series models [27, 28, 29], Google's Gemini 2.5 Pro [12], Claude 3.7 Sonnet with "extended thinking" mode [2] and DeepSeek's R1 [13], which have massively outperformed traditional LLMs on a wide range of benchmarks.

Meanwhile, program-aided reasoning systems like PAL have models emit code that is executed to obtain verifiable answers, pushing performance beyond pure text-only reasoning [14]. Tool-use agents (e.g. ReAct, Reflexion) further integrate search, calculators, or external APIs into the reasoning loop, enabling models to plan, act, and reflect iteratively [44, 36]. Despite these advances, LLMs remain fragile in long-horizon and stateful settings, as evidenced by their performance in `gg-bench`.

Tips for the Final Project: Related Work

Measuring General Intelligence with Generated Games



5 Related Work

Benchmarking LLMs with games. Games have long served as testbeds for measuring AI capabilities, leading to breakthroughs like Deep Blue for chess [8], AlphaZero for Go [37], and Libratus for poker [5]. Schaul et al. [33] argue that games offer a scalable proxy for artificial general intelligence because they can be procedurally generated to span a broad spectrum of difficulties and skills.

Recent work has begun to evaluate LLMs with games. Text-adventure suites such as Jericho [17] are designed to test agents' abilities to parse narrative state and issue actions. GameBench [11] focuses on hand-picked environments (e.g. Battleship, Connect Four) chosen to stress distinct planning skills while avoiding games likely

Scalable benchmarking. Fixed test

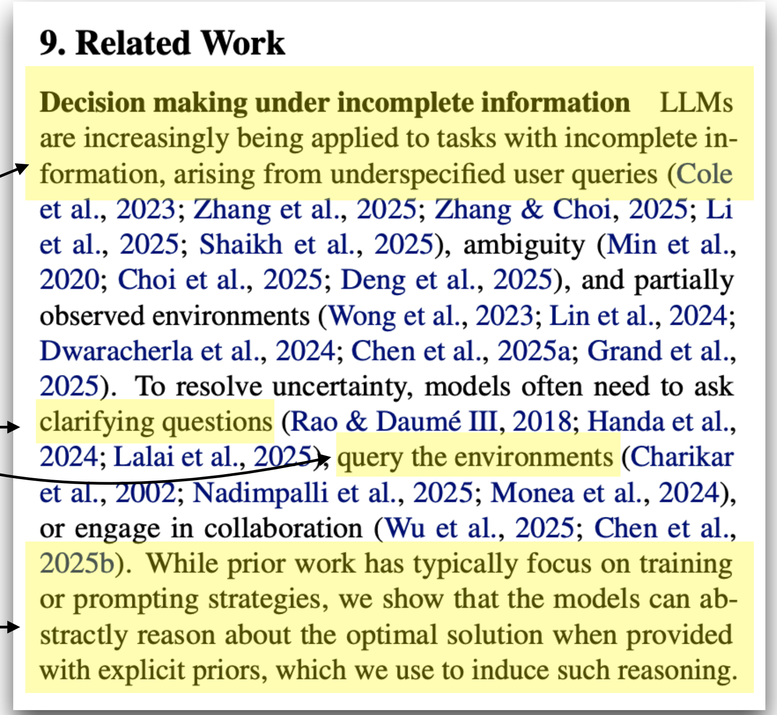
tasks from real-world GitHub issues. τ-bench [45] follows a hybrid synthetic approach, combining manually designed schemas, LLM-generated dialogues, and human refinement to evaluate agent interactions with tools and users in realistic domains. In contemporary work, Absolute Zero [47] uses LLMs to generate synthetic tasks which are used for training reasoning models. `gg-bench` inherits this spirit of scalability: new games, code implementations, and RL agents can be regenerated on demand, reducing the potential risks of dataset contamination and benchmark saturation.

Reasoning with language models. Many recent advancements in language modeling have been driven by *reasoning*, or the use of additional inference-time compute in order to obtain higher-quality generations. Early work in this direction showed that prompting models to generate explicit step-by-step answers, i.e., a chain of thought, improved their arithmetic and logical consistency [26, 43]. Training models to generate longer chains of thought via reinforcement learning has supposedly resulted in models such as OpenAI's o-series models [27, 28, 29], Google's Gemini 2.5 Pro [12], Claude 3.7 Sonnet with "extended thinking" mode [2] and DeepSeek's R1 [13], which have massively outperformed traditional LLMs on a wide range of benchmarks.

Meanwhile, program-aided reasoning systems like PAL have models emit code that is executed to obtain verifiable answers, pushing performance beyond pure text-only reasoning [14]. Tool-use agents (e.g. ReAct, Reflexion) further integrate search, calculators, or external APIs into the reasoning loop, enabling models to plan, act, and reflect iteratively [44, 36]. Despite these advances, LLMs remain fragile in long-horizon and stateful settings, as evidenced by their performance in `gg-bench`.

Three sections: one for each high-level category of related work

Calibrate-Then-Act: Cost-Aware Exploration in LLM Agents



9. Related Work

Decision making under incomplete information LLMs are increasingly being applied to tasks with incomplete information, arising from underspecified user queries (Cole et al., 2023; Zhang et al., 2025; Zhang & Choi, 2025; Li et al., 2025; Shaikh et al., 2025), ambiguity (Min et al., 2020; Choi et al., 2025; Deng et al., 2025), and partially observed environments (Wong et al., 2023; Lin et al., 2024; Dwaracherla et al., 2024; Chen et al., 2025a; Grand et al., 2025). To resolve uncertainty, models often need to ask clarifying questions (Rao & Daumé III, 2018; Handa et al., 2024; Lalai et al., 2025), query the environments (Charikar et al., 2002; Nadimpalli et al., 2025; Monea et al., 2024), or engage in collaboration (Wu et al., 2025; Chen et al., 2025b). While prior work has typically focus on training or prompting strategies, we show that the models can abstractly reason about the optimal solution when provided with explicit priors, which we use to induce such reasoning.

High-level summary

"Buckets" of prior work

How our work is different

Tips for the Final Project: Analysis

Think about what questions you can answer beyond accuracy/F1/AUROC.

- Which features in your dataset drive performance?
  - Can look at learned weights in linear models
  - For more complex models: remove features and re-train your model
- How much data do your models need to perform well? (Plot sample complexity curves)
- If you trained multiple models, are their errors correlated? What does this tell us about the "easy" and "hard" examples in the dataset? Can we qualitatively tell a difference?

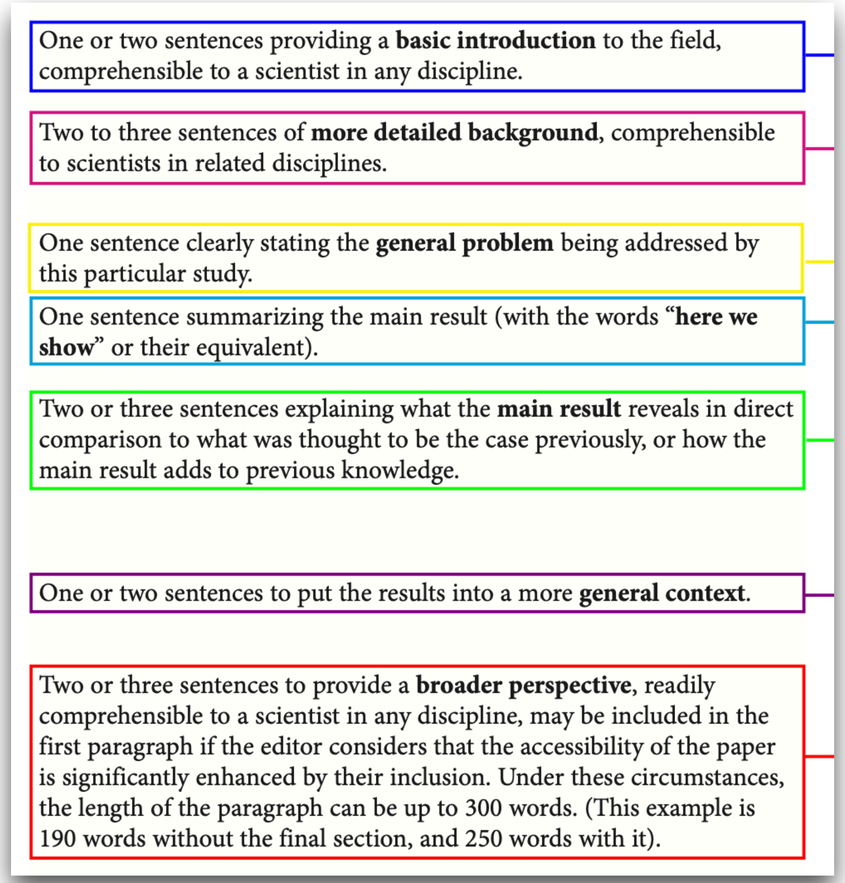
Tips for the Final Project: Accessibility

Machine learning is a big field. We're not experts in every part of it.

Please try to make your final paper as accessible as possible. This means:

- Defining terms you use
- Providing both high-level motivations as well as full technical descriptions of any method you use, including baselines

The goal: anyone who has taken DS-GA 1003 should be able to read and understand your paper.



How to construct a Nature summary paragraph

The Story So Far

- Machine learning methods make different assumptions about the data distribution:

The Story So Far

- Machine learning methods make different assumptions about the data distribution:
  - Logistic regression: assumes linear decision boundary
  - Soft-margin SVMs: assumes data is “close” to linearly separable
  - Naive Bayes: assumes features are conditionally independent given class labels

The Story So Far

- Machine learning methods make different assumptions about the data distribution:
  - Logistic regression: assumes linear decision boundary
  - Soft-margin SVMs: assumes data is “close” to linearly separable
  - Naive Bayes: assumes features are conditionally independent given class labels
- Large $\mathscr{H}$ may lead to overfitting; small $\mathscr{H}$ may lead to underfitting

The Bitter Lesson

The Bitter Lesson

Rich Sutton

March 13, 2019

The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation. Most AI research has been conducted as if the computation available to the agent were constant (in which case leveraging human knowledge would be one of the only ways to improve performance) but, over a slightly longer time than a typical research project, massively more computation inevitably becomes available. Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation. These two need not run counter to each other, but in practice they tend to. Time spent on one is time not spent on the other. There are psychological commitments to investment in one approach or the other. And the human-knowledge approach tends to complicate methods in ways that make them less suited to taking advantage of general methods leveraging computation. There were many examples of AI researchers' belated learning of this bitter lesson, and it is instructive to review some of the most prominent.

In computer chess, the methods that defeated the world champion, Kasparov, in 1997, were based on massive, deep search. At the time, this was looked upon with dismay by the majority of computer-chess researchers who had pursued methods that leveraged human understanding of the special structure of chess. When a simpler, search-based approach with special hardware and software proved vastly more effective, these human-knowledge-based chess researchers were not good losers. They said that ``brute force'' search may have won this time, but it was not a general strategy, and anyway it was not how people played chess. These researchers wanted methods based on human input to win and were disappointed when they did not.

The Bitter Lesson

The Bitter Lesson

Rich Sutton

March 13, 2019

The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation. Most AI research has been conducted as if the computation available to the agent were constant (in which case leveraging human knowledge would be one of the only ways to improve performance) but, over a slightly longer time than a typical research project, massively more computation inevitably becomes available. Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation. These two need not run counter to each other, but in practice they tend to. Time spent on one is time not spent on the other. There are psychological commitments to investment in one approach or the other. And the human-knowledge approach tends to complicate methods in ways that make them less suited to taking advantage of general methods leveraging computation. There were many examples of AI researchers' belated learning of this bitter lesson, and it is instructive to review some of the most prominent.

In computer chess, the methods that defeated the world champion, Kasparov, in 1997, were based on massive, deep search. At the time, this was looked upon with dismay by the majority of computer-chess researchers who had pursued methods that leveraged human understanding of the special structure of chess. When a simpler, search-based approach with special hardware and software proved vastly more effective, these human-knowledge-based chess researchers were not good losers. They said that ``brute force'' search may have won this time, but it was not a general strategy, and anyway it was not how people played chess. These researchers wanted methods based on human input to win and were disappointed when they did not.

The rest of this class:

- Relax assumptions about data distributions
- Bigger hypothesis space $\mathscr{H}$
- Bigger training datasets
- Models that scale effectively

The Bitter Lesson

The Bitter Lesson

Rich Sutton

March 13, 2019

The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin. The ultimate reason for this is Moore's law, or rather its generalization of continued exponentially falling cost per unit of computation. Most AI research has been conducted as if the computation available to the agent were constant (in which case leveraging human knowledge would be one of the only ways to improve performance) but, over a slightly longer time than a typical research project, massively more computation inevitably becomes available. Seeking an improvement that makes a difference in the shorter term, researchers seek to leverage their human knowledge of the domain, but the only thing that matters in the long run is the leveraging of computation. These two need not run counter to each other, but in practice they tend to. Time spent on one is time not spent on the other. There are psychological commitments to investment in one approach or the other. And the human-knowledge approach tends to complicate methods in ways that make them less suited to taking advantage of general methods leveraging computation. There were many examples of AI researchers' belated learning of this bitter lesson, and it is instructive to review some of the most prominent.

In computer chess, the methods that defeated the world champion, Kasparov, in 1997, were based on massive, deep search. At the time, this was looked upon with dismay by the majority of computer-chess researchers who had pursued methods that leveraged human understanding of the special structure of chess. When a simpler, search-based approach with special hardware and software proved vastly more effective, these human-knowledge-based chess researchers were not good losers. They said that ``brute force'' search may have won this time, but it was not a general strategy, and anyway it was not how people played chess. These researchers wanted methods based on human input to win and were disappointed when they did not.

The rest of this class:

- Relax assumptions about data distributions
- Bigger hypothesis space $\mathscr{H}$
- Bigger training datasets
- Models that scale effectively

With fewer assumptions, it's harder to do meaningful theory. This class is about to become much more empirical.

Recap: Logistic Regression

Hypothesis class: $\mathcal{H} = \{h_w(x) = w^\top x : w \in \mathbb{R}^d\}; \mathcal{X} \in \mathbb{R}^d$ and $\mathcal{Y} \in \{0,1\}$

Prediction rule: $\hat{y} = \mathbf{1}[w^\top x \geq 0]$

Probabilistic interpretation: $P(y = 1 \mid x; w) = \sigma(w^\top x) = \dfrac{1}{1 + e^{-w^\top x}}$

Logistic loss: $\ell(w; x, y) = -\, y \log \sigma(w^\top x) - (1 - y)\log(1 - \sigma(w^\top x))$

Training objective: $\min\limits_{w \in \mathbb{R}^d} \dfrac{1}{n} \sum\limits_{i=1}^{n} \left[-y_i \log \sigma(w^\top x_i) - (1 - y_i)\log(1 - \sigma(w^\top x_i))\right] + \dfrac{\lambda}{2}\|w\|_2^2$

Logistic Regression with Feature Function ϕ

Hypothesis class: $\mathcal{H} = \{h_w(x) = w^\top \boldsymbol{\phi}(x) : w \in \mathbb{R}^d\}; \quad \mathcal{X} \in \mathbb{R}^d$ and $\mathcal{Y} \in \{0,1\}$

Prediction rule: $\hat{y} = \mathbf{1}[w^\top \boldsymbol{\phi}(x) \geq 0]$

Probabilistic interpretation: $P(y = 1 \mid x; w) = \sigma(w^\top \boldsymbol{\phi}(x)) = \dfrac{1}{1 + e^{-w^\top \boldsymbol{\phi}(x)}}$

Logistic loss: $\ell(w; x, y) = -\, y \log \sigma(w^\top \boldsymbol{\phi}(x)) - (1 - y)\log(1 - \sigma(w^\top \boldsymbol{\phi}(x)))$

Training objective: $\min_{w \in \mathbb{R}^d} \dfrac{1}{n} \sum_{i=1}^{n} \left[-y_i \log \sigma(w^\top \boldsymbol{\phi}(x_i)) - (1 - y_i)\log(1 - \sigma(w^\top \boldsymbol{\phi}(x_i)))\right] + \dfrac{\lambda}{2}\|w\|_2^2$

Multi-class Logistic Regression

Hypothesis class: $\mathscr{H} = \{h_W(x) = \{w_y \cdot x\}_{y \in \mathscr{Y}} : w_y \in \mathbb{R}^d\}; \quad \mathscr{Y} = \{1, \ldots, K\}$

Prediction rule: $\hat{y} = \arg \max_{k \in \{1,\ldots,K\}} w_k^\top x$

Probabilistic interpretation: $P(y = k \mid x; W) = \dfrac{e^{w_k^\top x}}{\sum_{j=1}^{K} e^{w_j^\top x}}$

Cross-entropy loss: $\ell(W; x, y) = -\log P(y \mid x; W) = -w_y^\top x + \log \sum_{j=1}^{K} e^{w_j^\top x}$

Training objective: $\min_{W \in \mathbb{R}^{K \times d}} \dfrac{1}{n} \sum_{i=1}^{n} \left[-w_{y_i}^\top x_i + \log \sum_{j=1}^{K} e^{w_j^\top x_j} \right] + \dfrac{\lambda}{2} \|W\|_F^2$

Multi-class Logistic Regression

Hypothesis class: $\mathscr{H} = \{h_W(x) = \{w_y \cdot \boldsymbol{\phi}(x)\}_{y \in \mathscr{Y}} : w_y \in \mathbb{R}^d\}; \quad \mathscr{Y} = \{1, \ldots, K\}$

Prediction rule: $\hat{y} = \arg\max_{y \in \mathscr{Y}} w_y \cdot \boldsymbol{\phi}(x)$

Probabilistic interpretation: $p(y \mid x, w) = \frac{e^{(w_y \cdot \boldsymbol{\phi}(x))}}{\sum_{y'} e^{(w_{y'} \cdot \boldsymbol{\phi}(x))}}$

Cross-entropy loss: $\ell(w; x, y) = -\log p(y \mid x, w) = -w_y \cdot \boldsymbol{\phi}(x) + \log \sum_{y'} e^{(w_{y'} \cdot \boldsymbol{\phi}(x))}$

Training objective: $\min_w \frac{1}{n} \sum_{i=1}^{n} \left[-w_{y_i} \cdot \boldsymbol{\phi}(x_i) + \log \sum_{y'} e^{(w_{y'} \cdot \boldsymbol{\phi}(x_i))} \right] + \frac{\lambda}{2} \sum_y \|w_y\|_2^2$

Multi-class Logistic Regression

Probabilistic interpretation: $p(y \mid x, w) = \dfrac{\exp(w_y \cdot \boldsymbol{\phi}(x))}{\sum_{y'} \exp(w_{y'} \cdot \boldsymbol{\phi}(x))}$

$$\begin{aligned} w_1^\top \phi(x) &= -1.1 \\ w_2^\top \phi(x) &= 2.1 \\ w_1^\top \phi(x) &= -0.4 \end{aligned} \xrightarrow{\text{softmax}} \begin{aligned} &0.036 \\ &0.89 \\ &0.07 \end{aligned}$$

Softmax operation: "exponentiate and normalize"

- We'll write this as: softmax$(W\phi(x))$

Feature Engineering

What should ϕ be?

- If we're dealing with raw numerical data: the identity function
- If we also need to capture nonlinear structure: a kernel
- If we're dealing with natural language text: ????

Feature Engineering

“One-hot" vectors

Gives us unique representations for each word, but doesn’t capture similarities between similar words

$$\boldsymbol{\phi}(\text{a}) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \boldsymbol{\phi}(\text{the}) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \boldsymbol{\phi}(\text{dog}) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

$$\boldsymbol{\phi}(\text{an}) = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \boldsymbol{\phi}(\text{cat}) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \boldsymbol{\phi}(\text{cats}) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

Feature Engineering

“One-hot" vectors

Gives us unique representations for each word, but doesn’t capture similarities between similar words

Instead, we can try to manually engineer vectors that capture similarities between words, but this gets complicated fast…

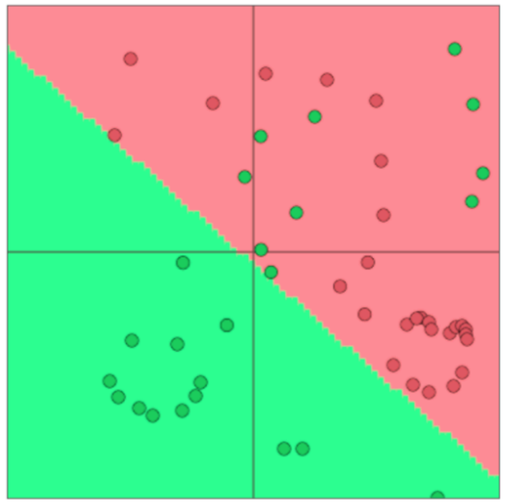
$$\boldsymbol{\phi}(\text{a}) = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad \boldsymbol{\phi}(\text{the}) = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix} \quad \boldsymbol{\phi}(\text{dog}) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 1 \\ 0 \\ 1 \\ \vdots \\ 0 \end{bmatrix}$$

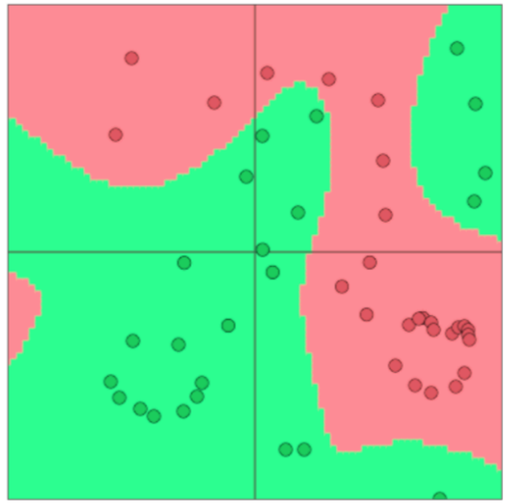
$$\boldsymbol{\phi}(\text{an}) = \begin{bmatrix} 0 \\ 1 \\ \vdots \\ 0 \\ 1 \\ 1 \\ \vdots \\ 0 \end{bmatrix} \quad \boldsymbol{\phi}(\text{cat}) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 1 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \quad \boldsymbol{\phi}(\text{cats}) = \begin{bmatrix} 1 \\ 0 \\ \vdots \\ 1 \\ 1 \\ 0 \\ \vdots \\ 1 \end{bmatrix}$$

Neural Networks: Automated Feature Engineering

The key idea: learn ϕ from data

- Classical approaches work with hand-designed feature spaces
- Representation learning: automatically learn good features and representations
- Deep learning: attempt to learn multiple levels of representation to capture increasing degrees of complexity and abstraction





Neural Networks: Automated Feature Engineering

The key idea: learn ϕ from data

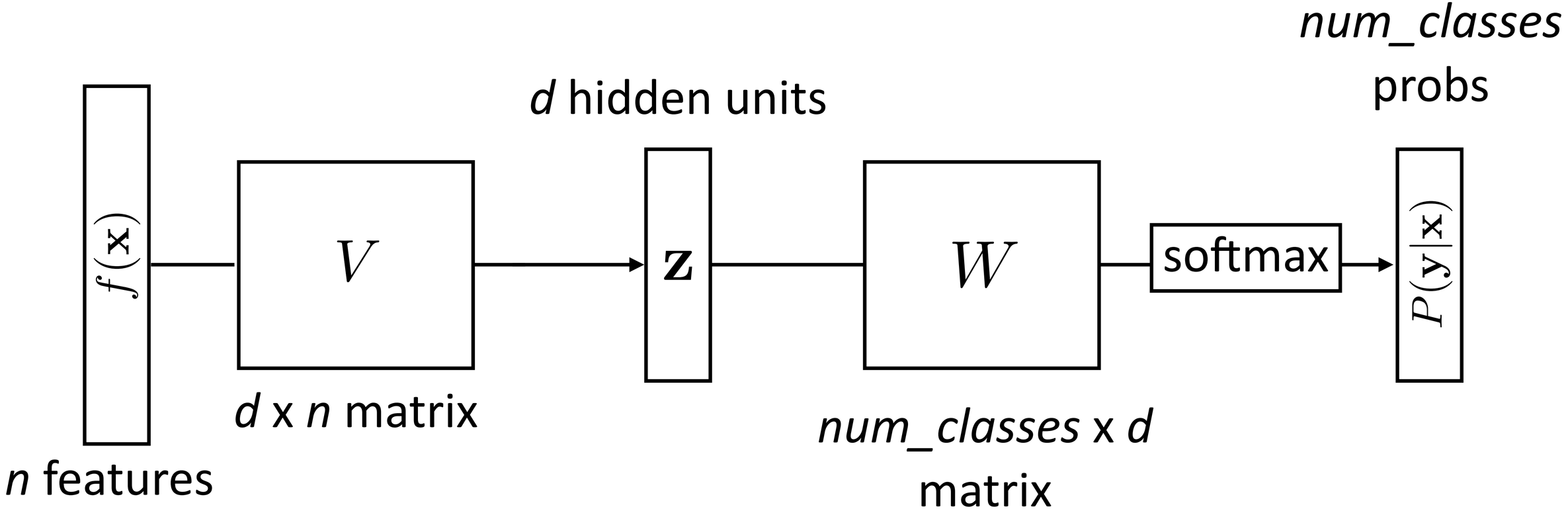
Logistic regression: $p(y \mid x) = \text{softmax}(W\phi(x))$

Neural networks: $p(y \mid x) = \text{softmax}(Wg(Vf(x)))$

$p(y \mid x) = \text{softmax}(Wg(V^2(g(V^1f(x)))))$

Neural Networks: Automated Feature Engineering

$$p(y \mid x) = \text{softmax}(W(Vf(x)))$$



Activation Functions: Expanding the Hypothesis Space

$$p(y \mid x) = \text{softmax}(W(Vf(x)))$$

What's the issue with this?

Activation Functions: Expanding the Hypothesis Space

$$p(y \mid x) = \text{softmax}(W(Vf(x)))$$

What's the issue with this? Composition of linear maps is linear:

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}}_{C=AB}$$

Activation Functions: Expanding the Hypothesis Space

$$p(y \mid x) = \text{softmax}(W(Vf(x)))$$

What's the issue with this? Composition of linear maps is linear:

$$\underbrace{\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix}}_{A} \underbrace{\begin{bmatrix} b_{11} & b_{12} & b_{13} & b_{14} \\ b_{21} & b_{22} & b_{23} & b_{24} \\ b_{31} & b_{32} & b_{33} & b_{34} \\ b_{41} & b_{42} & b_{43} & b_{44} \end{bmatrix}}_{B} = \underbrace{\begin{bmatrix} c_{11} & c_{12} & c_{13} & c_{14} \\ c_{21} & c_{22} & c_{23} & c_{24} \\ c_{31} & c_{32} & c_{33} & c_{34} \\ c_{41} & c_{42} & c_{43} & c_{44} \end{bmatrix}}_{C=AB}$$

This means our hypothesis class is the same size as in traditional logistic regression!

Activation Functions: Expanding the Hypothesis Space

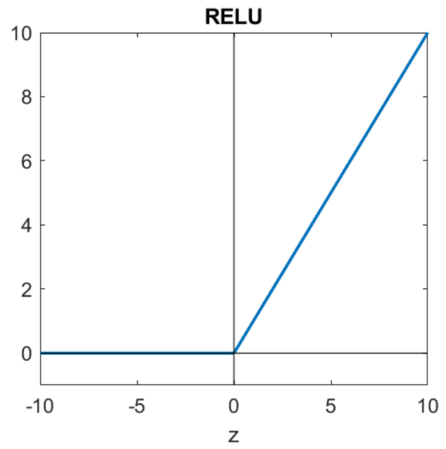
$$p(y \mid x) = \text{softmax}(W\mathbf{g}(Vf(x)))$$

Adding a nonlinearity allows us to express a more complex hypothesis space:

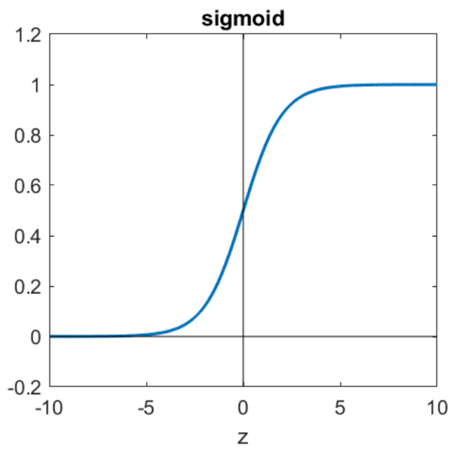
Activation Functions: Expanding the Hypothesis Space

$$p(y \mid x) = \text{softmax}(W\mathbf{g}(Vf(x)))$$

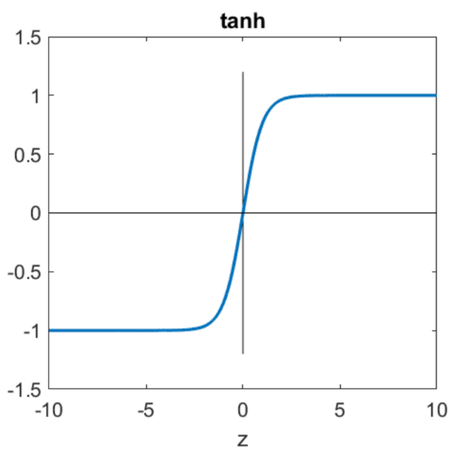
Adding a nonlinearity allows us to express a more complex hypothesis space:



$$\mathbf{g}(z) = \max(0,z)$$



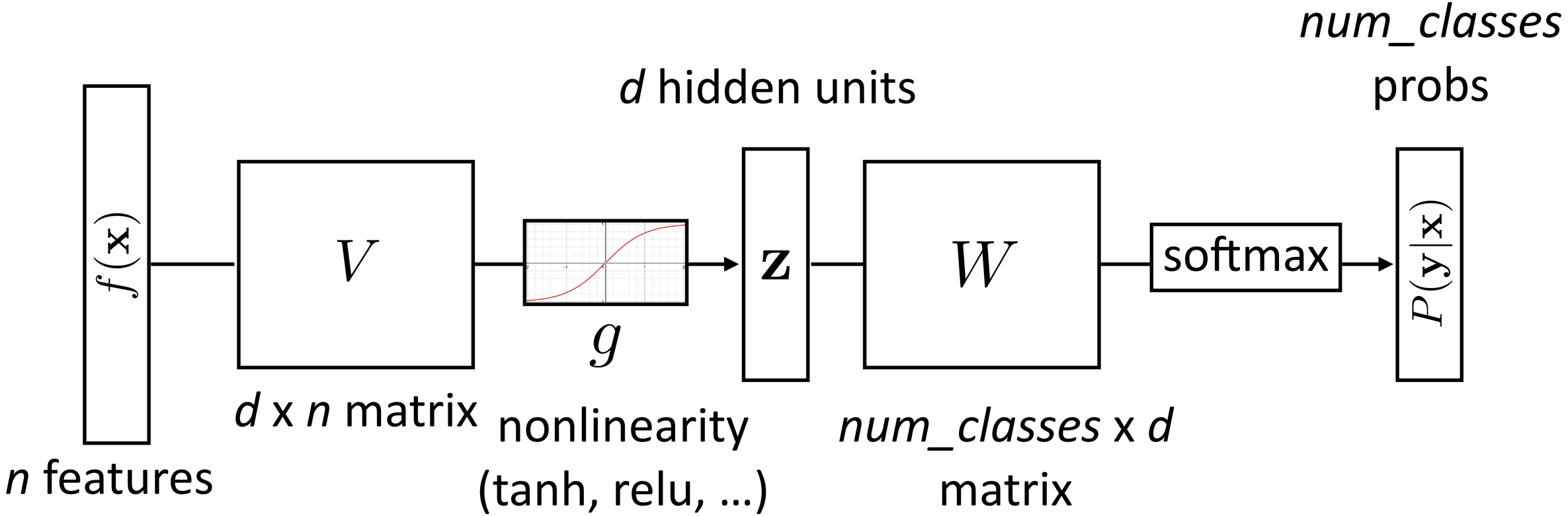
$$\mathbf{g}(z) = \frac{1}{1+e^{-z}}$$



$$\mathbf{g}(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}}$$

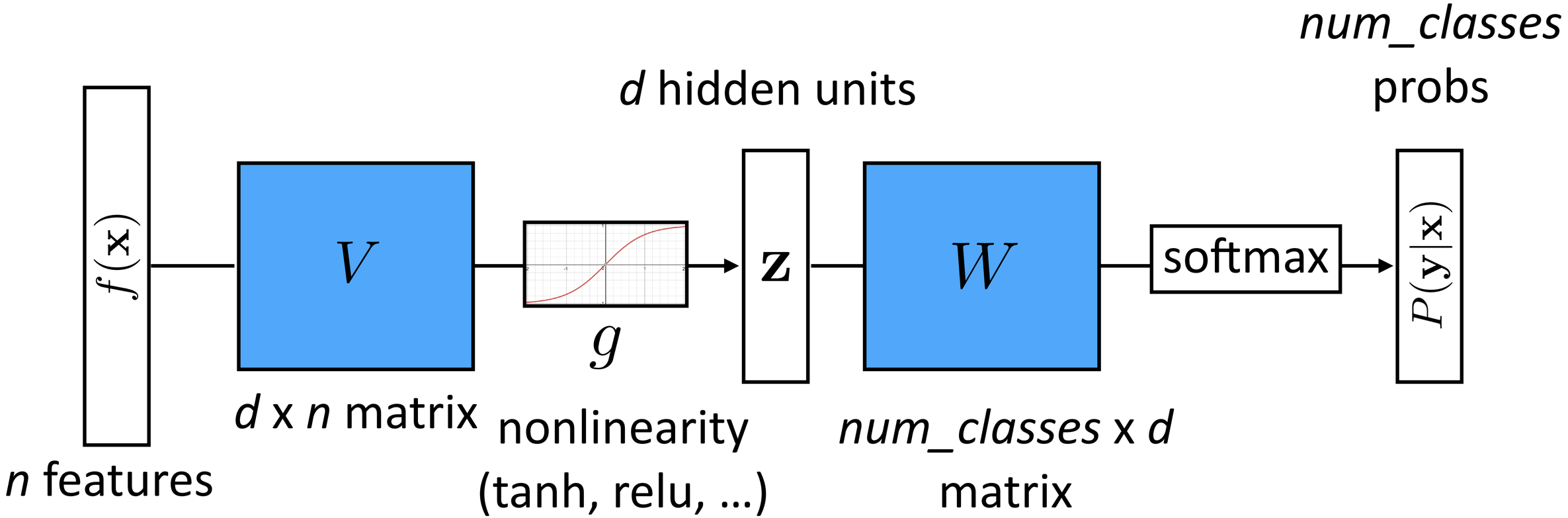
Neural Networks: Automated Feature Engineering

$$p(y \mid x) = \text{softmax}(Wg(Vf(x)))$$



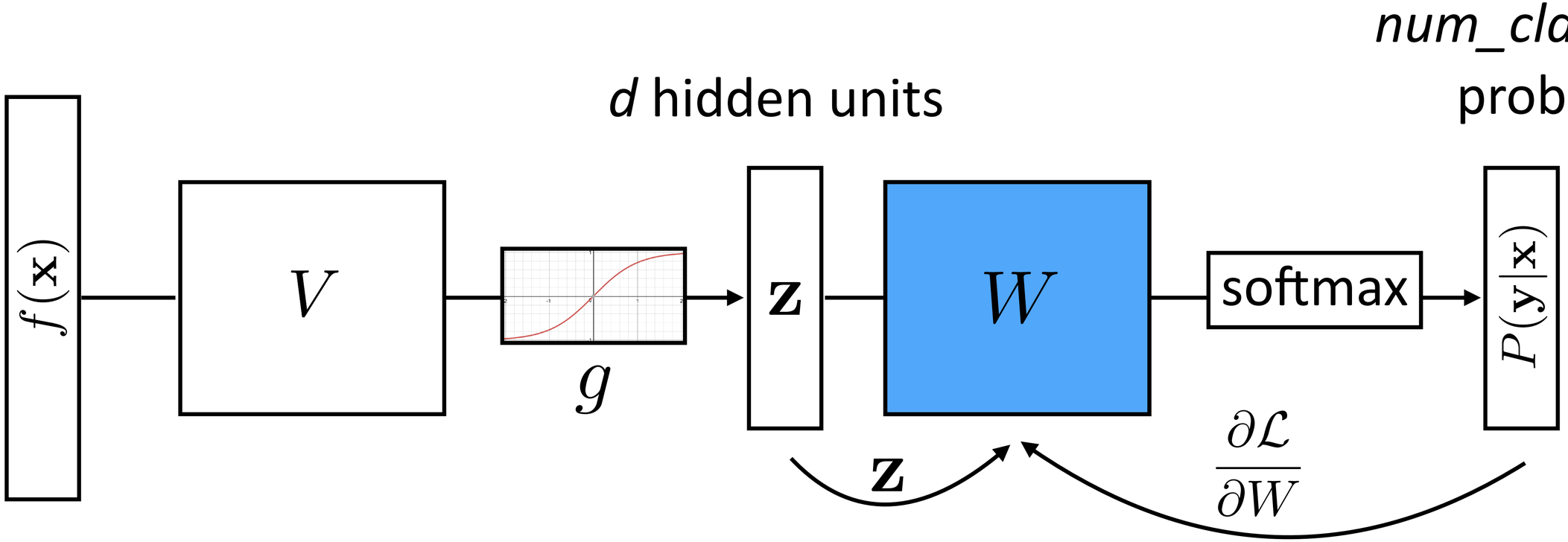
Backprop: How Do We Learn the Weights?

$$p(y \mid x) = \text{softmax}(Wg(Vf(x)))$$



Backprop: How Do We Learn the Weights?

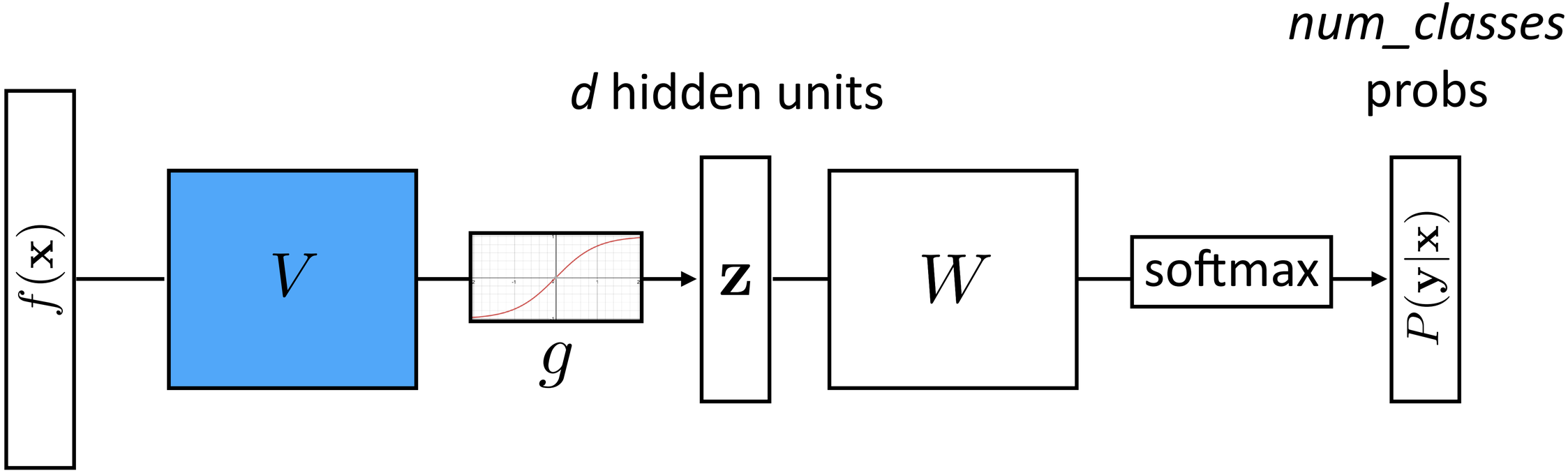
$$p(y \mid x) = \text{softmax}(Wg(Vf(x)))$$



Gradient w.r.t. *W*: looks like logistic regression, can be computed treating **z** as the input features

Backprop: How Do We Learn the Weights?

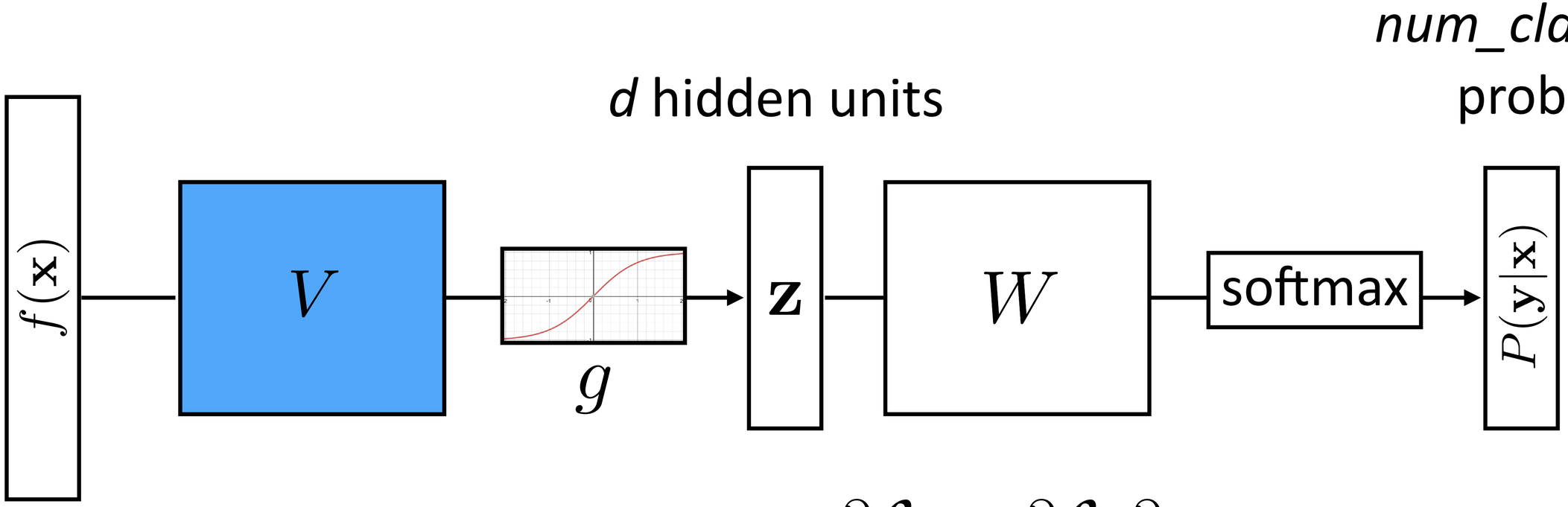
$$p(y \mid x) = \text{softmax}(Wg(Vf(x)))$$



But how do we go about computing gradients for intermediate representations?

Backprop: How Do We Learn the Weights?

$$p(y \mid x) = \text{softmax}(Wg(Vf(x)))$$



Key idea: use the chain rule!

$$\frac{\partial \mathcal{L}}{\partial V} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial V}$$

Recall: Derivatives and the Chain Rule

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = ?$$

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = 12x^3 - 4x + 5$$

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = 12x^3 - 4x + 5$$

$$\frac{d}{dx}\left[\log(x)\right] = ?$$

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = 12x^3 - 4x + 5$$

$$\frac{d}{dx}\left[\log(x)\right] = \frac{1}{x}$$

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = 12x^3 - 4x + 5$$

$$\frac{d}{dx}\left[\log(x)\right] = \frac{1}{x}$$

Chain rule:

$$\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}}\frac{\partial \mathbf{z}}{\partial x}$$

$$\frac{\partial}{\partial x}\log(x^2 + 1) = ?$$

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = 12x^3 - 4x + 5$$

$$\frac{d}{dx}\left[\log(x)\right] = \frac{1}{x}$$

Chain rule:

$$\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial x}$$

$$\frac{\partial}{\partial x} \log(x^2 + 1) = ?$$

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = 12x^3 - 4x + 5$$

$$\frac{d}{dx}\left[\log(x)\right] = \frac{1}{x}$$

Chain rule:

Step 1: Let $z = x^2 + 1$ and $\mathscr{L} = \log(z)$

$$\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial x}$$

$$\frac{\partial}{\partial x} \log(x^2 + 1) = ?$$

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = 12x^3 - 4x + 5$$

$$\frac{d}{dx}\left[\log(x)\right] = \frac{1}{x}$$

Chain rule:

$$\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial x}$$

$$\frac{\partial}{\partial x} \log(x^2 + 1) = ?$$

Step 1: Let $z = x^2 + 1$ and $\mathscr{L} = \log(z)$

Step 2:

$$\frac{\partial \mathscr{L}}{\partial z} = \frac{1}{z} = \frac{1}{x^2 + 1}$$

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = 12x^3 - 4x + 5$$

$$\frac{d}{dx}\left[\log(x)\right] = \frac{1}{x}$$

Chain rule:

$$\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial x}$$

$$\frac{\partial}{\partial x} \log(x^2 + 1) = ?$$

Step 1: Let $z = x^2 + 1$ and $\mathscr{L} = \log(z)$

Step 2:

$$\frac{\partial \mathscr{L}}{\partial z} = \frac{1}{z} = \frac{1}{x^2 + 1}$$

$$\frac{\partial z}{\partial x} = \frac{\partial}{\partial x}(x^2 + 1) = 2x$$

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = 12x^3 - 4x + 5$$

$$\frac{d}{dx}\left[\log(x)\right] = \frac{1}{x}$$

Chain rule:

$$\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial x}$$

$$\frac{\partial}{\partial x} \log(x^2 + 1) = ?$$

Step 1: Let $z = x^2 + 1$ and $\mathscr{L} = \log(z)$

Step 2: $\frac{\partial \mathscr{L}}{\partial z} = \frac{1}{x^2 + 1}$ $\frac{\partial z}{\partial x} = 2x$

Recall: Derivatives and the Chain Rule

Basic derivatives:

$$\frac{d}{dx}\left[3x^4 - 2x^2 + 5x - 7\right] = 12x^3 - 4x + 5$$

$$\frac{d}{dx}\left[\log(x)\right] = \frac{1}{x}$$

Chain rule:

$$\frac{\partial \mathscr{L}}{\partial x} = \frac{\partial \mathscr{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial x}$$

$$\frac{\partial}{\partial x} \log(x^2 + 1) = ?$$

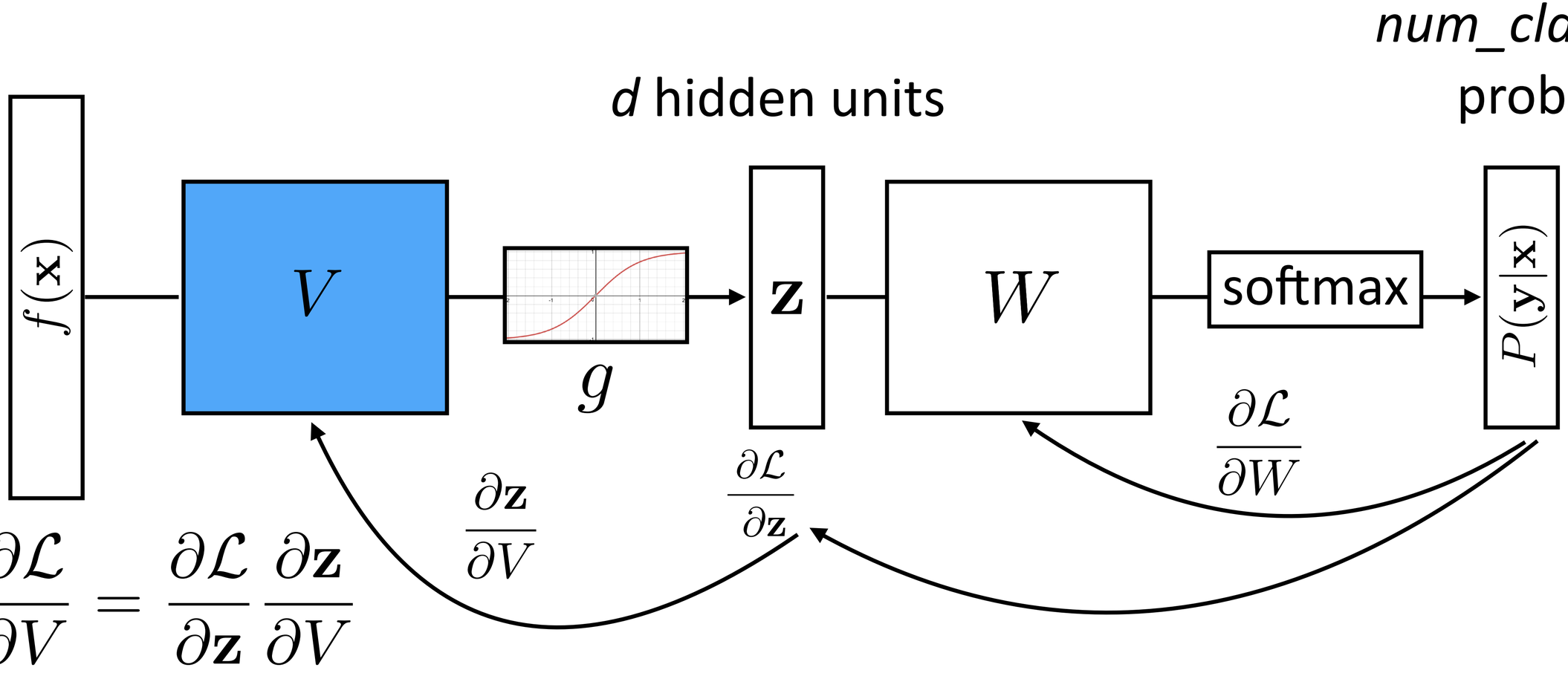
Step 1: Let $z = x^2 + 1$ and $\mathscr{L} = \log(z)$

Step 2: $\frac{\partial \mathscr{L}}{\partial z} = \frac{1}{x^2 + 1}$ $\frac{\partial z}{\partial x} = 2x$

Step 3: $\frac{\partial}{\partial x} \log(x^2 + 1) = \frac{2x}{x^2 + 1}$

Backprop: How Do We Learn the Weights?

$p(y \mid x) = \text{softmax}(Wg(Vf(x)))$



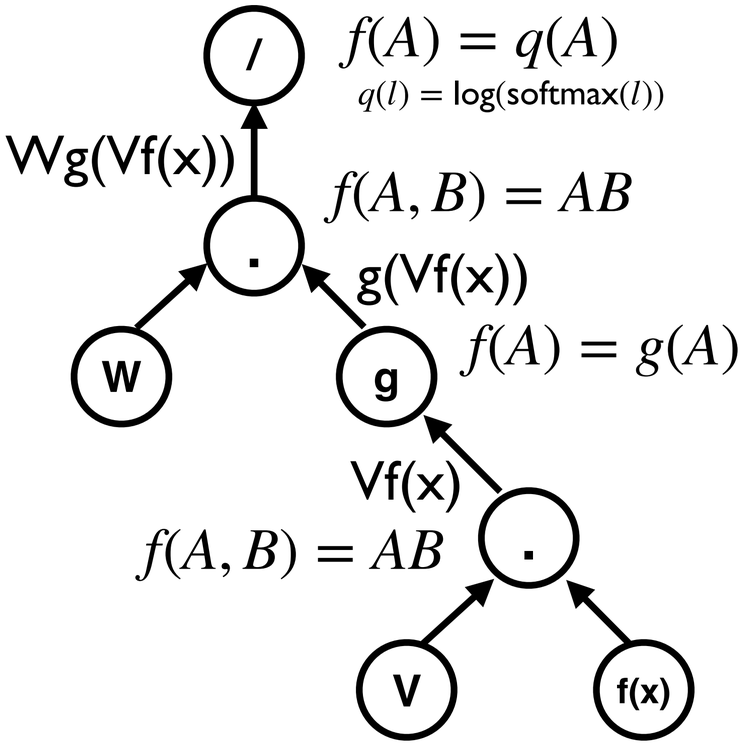
Computation Graphs

$$p(y \mid x) = \text{softmax}(Wg(Vf(x)))$$

A functional description of the required computation:

- **Nodes** represent {tensor, matrix, vector, scalar} values
- An **edge** represents a function argument (and also a data dependency). Edges are just pointers to nodes.
- A node with an incoming edge is a function of that that edge's tail node.

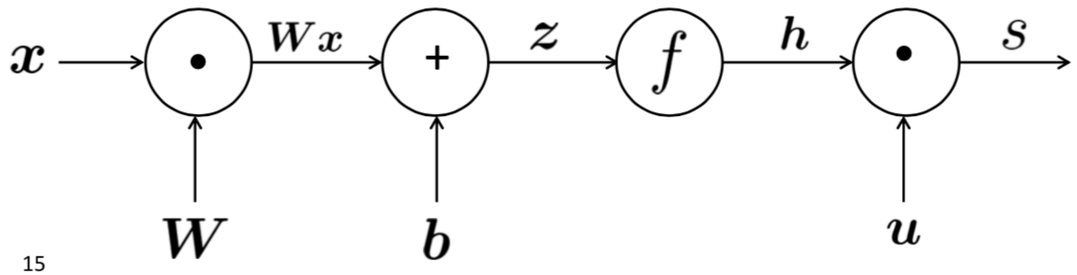
The order in which we traverse the graph is important!



Computation Graphs

Forward pass:

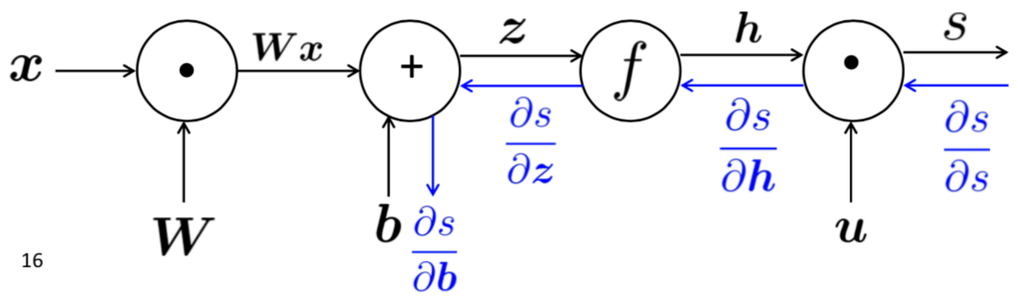
- Given parameters and an input, make a decision
- Visit nodes in topological order



Computation Graphs

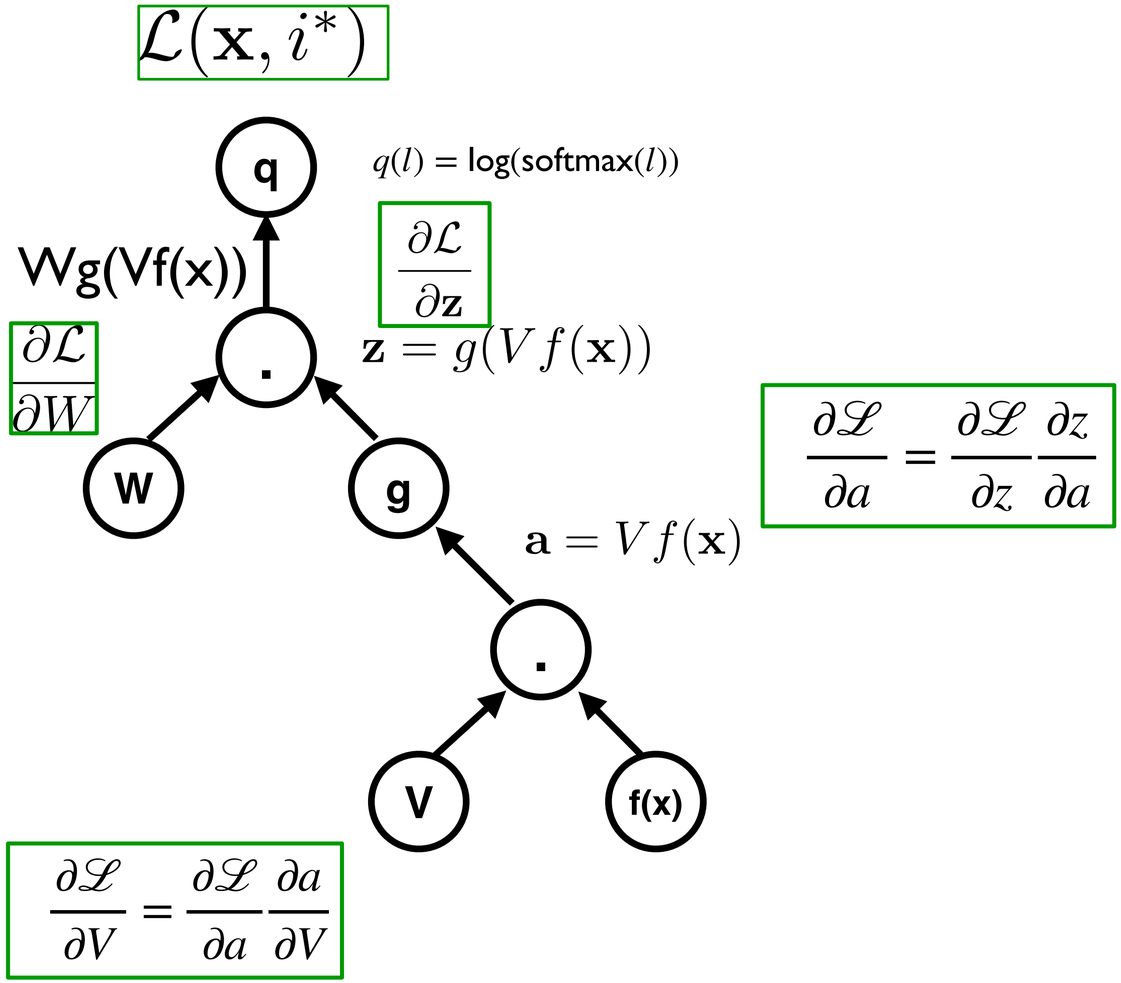
Backward pass:

- Loop over the nodes in reverse topological order, starting from the final node
- This answers: how does the output change if we make changes to the input?



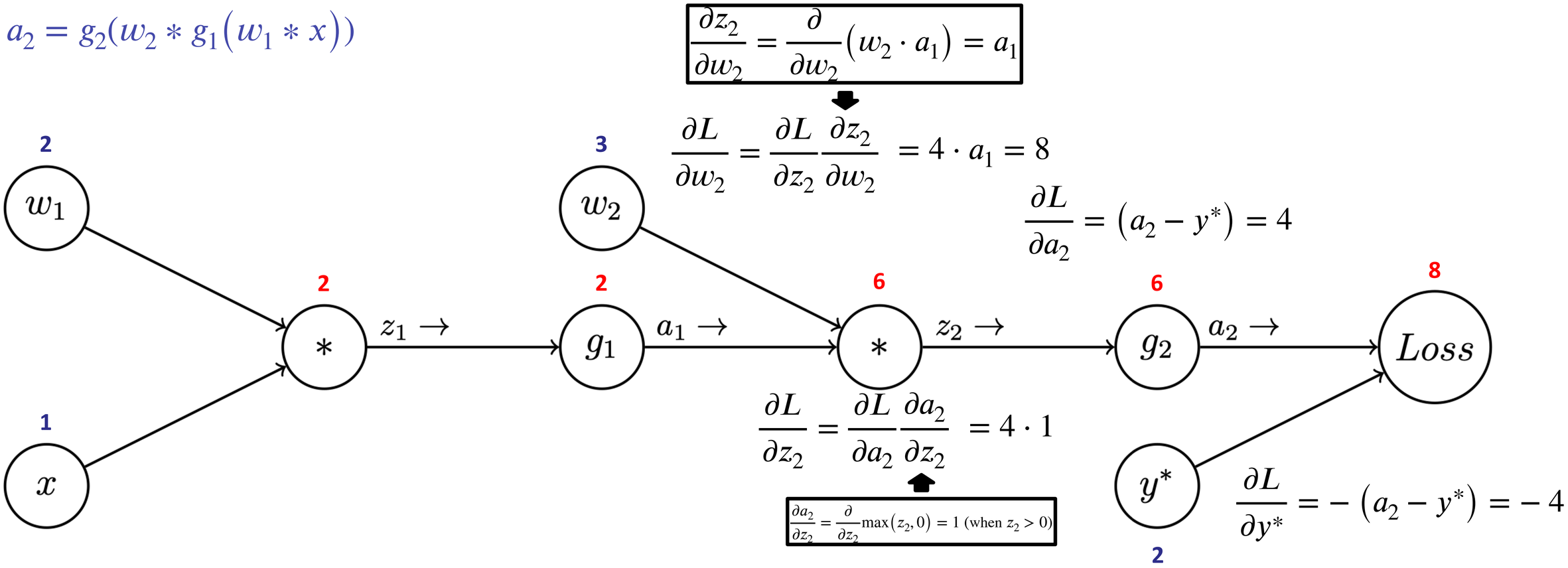
Computation Graphs

Backward pass:



Worked Example of Backpropagation

- Build a computation graph and apply chain rule: $f(x) = g\big(h(x)\big)$ $\quad f'(x) = h'(x) \cdot g'(h(x))$
- Example: neural network with quadratic loss: $L\big(a_2,\ y^*\big) = \frac{1}{2}\big(a_2 - y^*\big)^2$ and ReLU activations $g(z) = \max(0,z)$
- $a_2 = g_2(w_2 * g_1\big(w_1 * x\big))$



Recap: Backpropagation

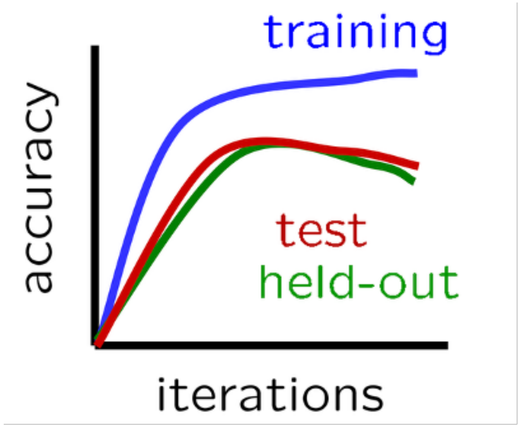
- Key idea #1: Use the chain rule!

$$\frac{\partial \mathcal{L}}{\partial V} = \frac{\partial \mathcal{L}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial V}$$

- Key idea #2: Re-use derivatives computed from later layers in computing derivatives for earlier layers, allowing for more efficient computation of gradients.

Preventing Overfitting: Dropout

- Early stopping:



- Weight regularization: $\min_{\{W^{(\ell)}\}} \frac{1}{n} \sum_{i=1}^{n} \mathscr{L}(y_i, f(x_i)) + \frac{\lambda}{2} \sum_{\ell} \|W^{(\ell)}\|_F^2$

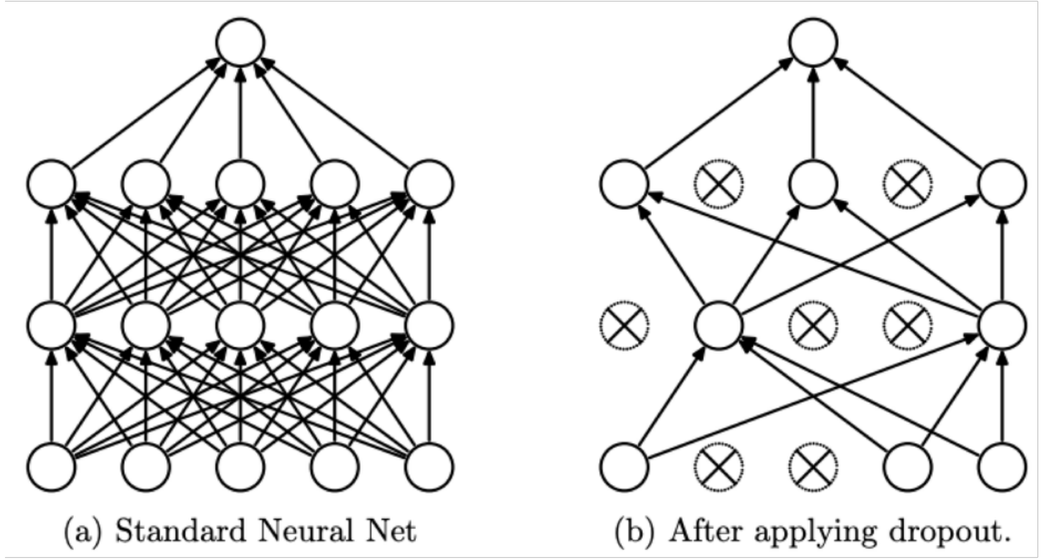
- Dropout

Preventing Overfitting: Dropout

- Key idea: “damage” the network during training to increase redundancy

- At each training step, with probability (1-p) set an activation to zero (i.e., drop it)

- When making predictions, don’t apply dropout, but multiply weights by p (rescaling)

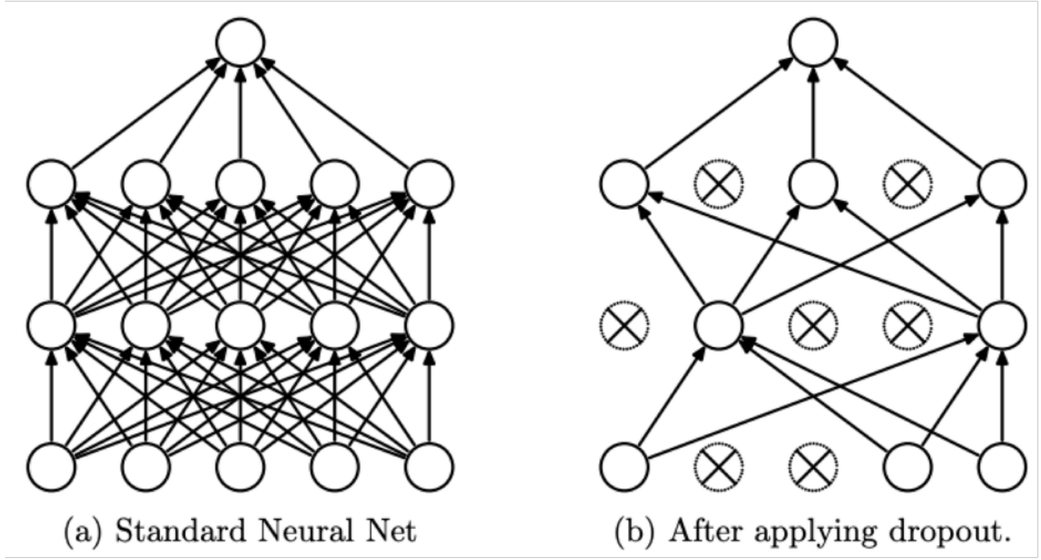


Preventing Overfitting: Dropout

- Key idea: “damage” the network during training to increase redundancy

- At each training step, with probability (1-p) set an activation to zero (i.e., drop it)
- When making predictions, don’t apply dropout, but multiply weights by p (rescaling)

- In recent practice, LLMs often don’t use dropout and instead avoid overfitting by training on increasingly large datasets (the bitter lesson!)



(a) Standard Neural Net (b) After applying dropout.

How do we actually update the weights?

Recall: Gradient descent

$$\min_w \mathscr{L}(w) = \frac{1}{n} \sum_{i=1}^{n} -\log P(y^{(i)} \mid x^{(i)}; w)$$

Compute the gradient on all data at once and then make a global update

This approach will usually be inefficient for very large datasets

init w

for iter = 1,2,…

$$w \leftarrow w - \alpha \nabla \mathscr{L}(w)$$

How do we actually update the weights?

Stochastic gradient descent

$$\min_w \mathscr{L}(w) = \frac{1}{n} \sum_{i=1}^{n} -\log P(y^{(i)} \mid x^{(i)}; w)$$

Observation: once gradient on one training example has been computed, might as well incorporate it before computing the next one

This is more efficient, but doesn't take advantage of modern hardware

init w

for iter = 1,2,…

pick random j

$w \leftarrow w - \alpha \nabla [-\log P(y^{(j)} \mid x^{(j)}; w)]$

How do we actually update the weights?

Minibatch gradient descent

$$\min_w \mathscr{L}(w) = \frac{1}{n} \sum_{i=1}^{n} -\log P(y^{(i)} \mid x^{(i)}; w)$$

Observation: gradient over small set of training examples (=mini-batch) can be computed in parallel, might as well do that instead of a single one

This is the most efficient of the three approaches we've seen so far!

init w

for iter = 1,2,…

pick random mini-batch J

$$w \leftarrow w - \alpha \sum_{j \in J} \nabla[-\log P(y^{(j)} \mid x^{(j)}; w)]$$

Three ways to go beyond vanilla SGD

1. Computing second-order derivatives

2. Using momentum

3. Setting adaptive learning rates

Concept #1: Computing Second-Order Derivatives

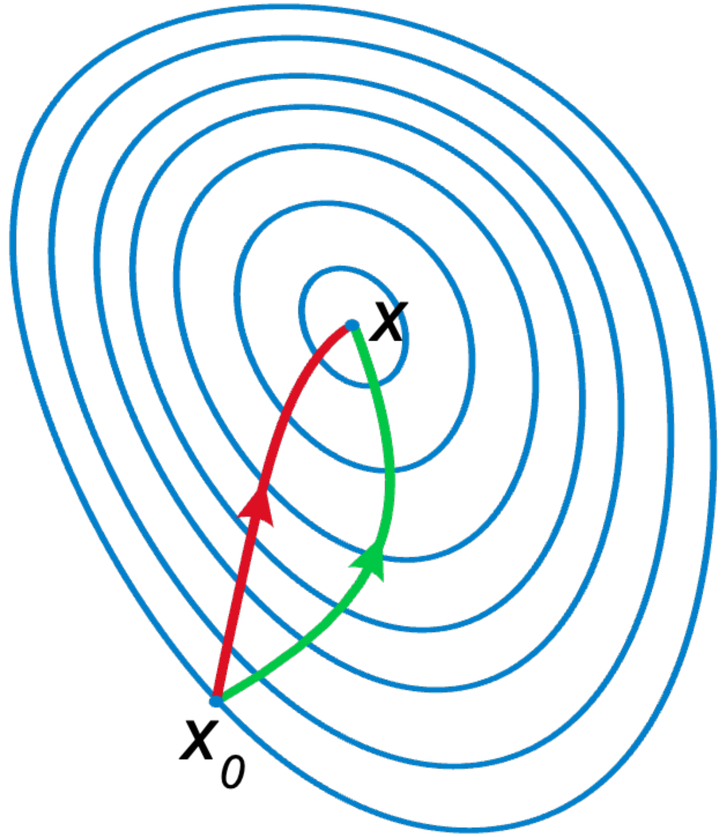
Newton's Method (in 1D):

- Don't just use the slope, use the curvature
- Want to optimize: $\max_{\theta} f(\theta)$
- Apply the Taylor expansion: $f(\theta + h) = f(\theta) + f'(\theta)h + \frac{1}{2}f''(\theta)h^2$
- Find the value of h that minimizes:

$$0 = \frac{\partial}{\partial h}\left[f(\theta) + f'(\theta)h + \frac{1}{2}f''(\theta)h^2\right] = f'(\theta) + f''(\theta)h$$

- Rearrange to get update:

$$h = -\frac{f'(\theta)}{f''(\theta)} \qquad \theta_{t+1} = \theta_t + h = \theta_t - \frac{f'(\theta)}{f''(\theta)}$$



Concept #2: Momentum

Potential issues with vanilla SGD:

- Can take a long time to converge if the learning rate is too low
- Can bounce around in “ravines” without making much progress toward a local optimum

Image 2: SGD without momentum

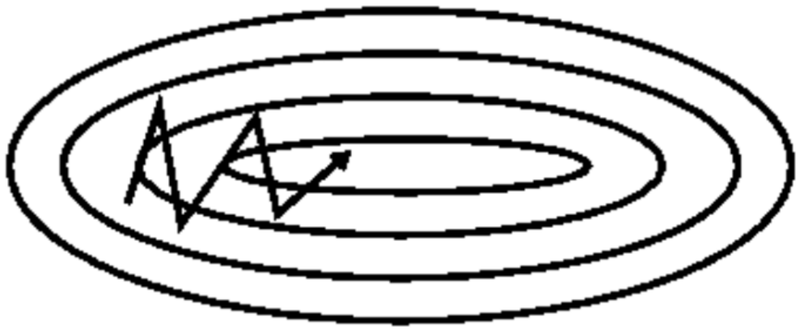


Image 3: SGD with momentum

Concept #2: Momentum

Stochastic gradient descent with momentum:

- Keep a running sum of old updates: $v_t = \gamma v_{t-1} + \eta \nabla_\theta f(\theta_t)$
- Perform a standard gradient descent step: $\theta_{t+1} = \theta_t - v_t$

Image 2: SGD without momentum

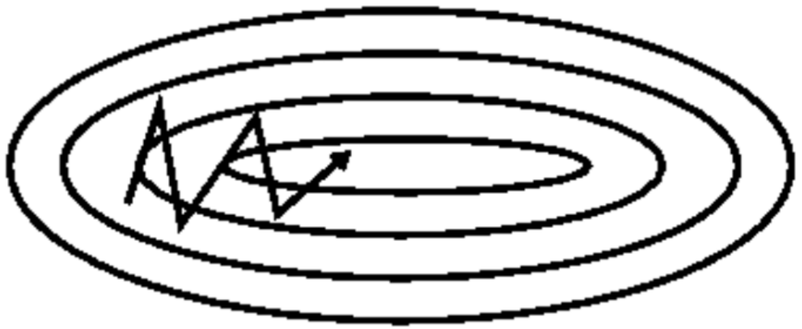


Image 3: SGD with momentum

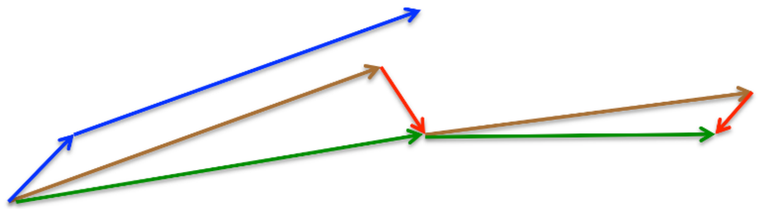
Concept #2: Momentum

Nesterov accelerated gradient (NAG):

- Key idea: “anticipate” where momentum will take you and compute the gradient at that point instead. Can think of this as momentum with planning

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta f(\theta_t - \gamma v_{t-1})$$

$$\theta_{t+1} = \theta_t - v_t$$



brown vector = jump, red vector = correction, green vector = accumulated gradient

blue vectors = standard momentum

Concept #3: Adaptive Learning Rates

Recall: learning rates

- Determines how much we update weights in the direction of the gradient
- Often: want to set this in terms of how much it updates the weights
- Often: want to lower learning rate over time (learning rate scheduling)

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta f(\theta_t)$$

Concept #3: Adaptive Learning Rates

Recall: learning rates

- Determines how much we update weights in the direction of the gradient
- Often: want to set this in terms of how much it updates the weights
- Often: want to lower learning rate over time (learning rate scheduling)

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta f(\theta_t)$$

- Key idea: different learning rates for each parameter
    - We can make larger or smaller updates depending on how important a feature is (small updates for frequent features; big updates for rare features). This idea underlies: Adagrad, RMSProp, Adam, etc.

Concept #3: Adaptive Learning Rates

Adagrad:

- Compute the gradient at the i-th parameter:

$$g_{t,i} = \nabla_\theta(\theta_{t,i})$$

- Make an update based on the adaptive learning rate:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{G_{t,ii} - \epsilon}} \cdot g_{t,i}$$

- Where G_t is a diagonal matrix where entry i, i is the sum of squares of the gradients up to timestep t. (However, this causes an issue where the sum of squares grows over time.)

Concept #3: Adaptive Learning Rates

RMSProp:

- Based on decaying running average of gradients:

$$E[g^2]_t = 0.9 \cdot E[g^2]_{t-1} + 0.1 \cdot g_t^2$$

- Adaptive learning rate based on running average instead of accumulated sum:

$$\theta_{t+1,i} = \theta_{t,i} - \frac{\eta}{\sqrt{E[g^2]_{t,i} + \epsilon}} \cdot g_{t,i}$$

Concept #3: Adaptive Learning Rates

Adaptive Moment Estimation (Adam):

- Combining momentum with adaptive learning rates:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

- Bias correction to prevent running averages from tending toward zero:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

- Then update parameters:

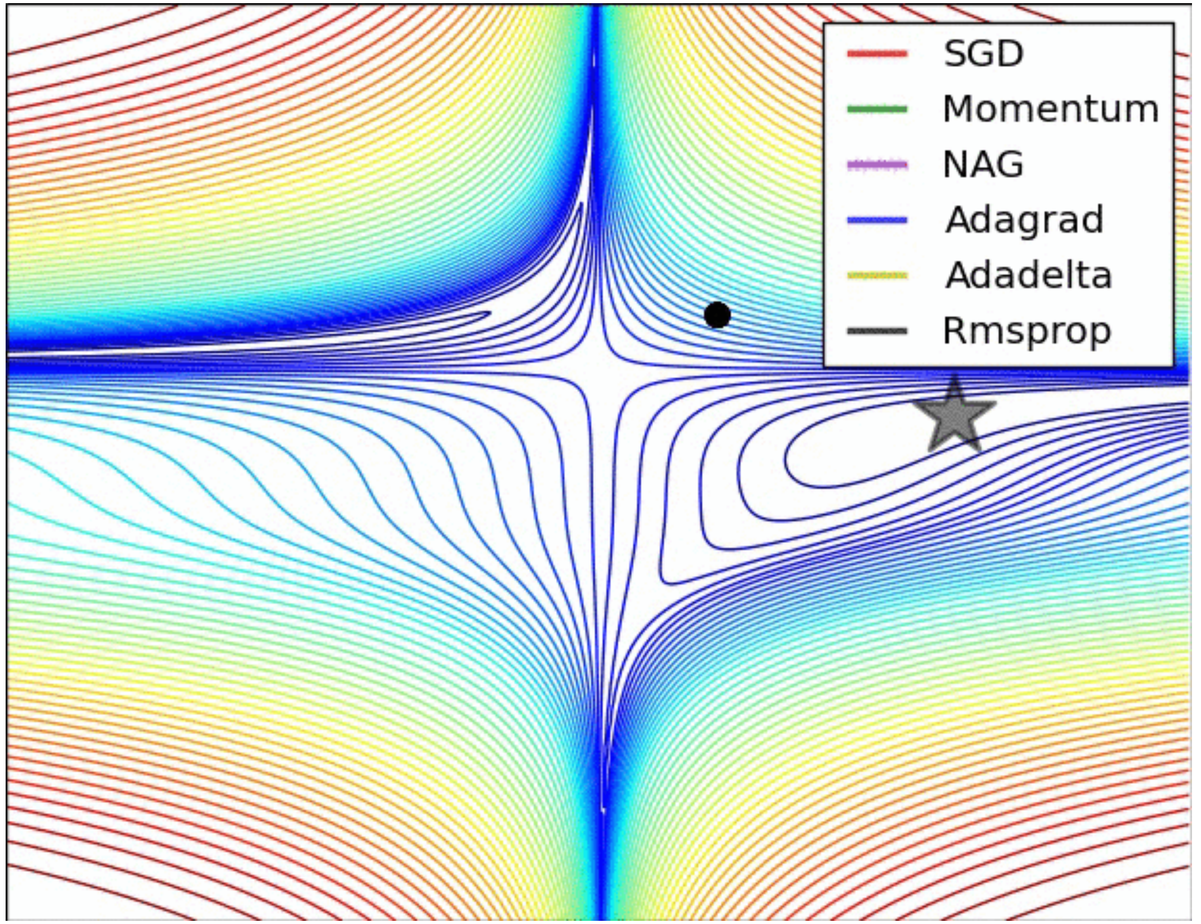
$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} \hat{m}_t$$

Key concepts in optimization

1. Computing second-order derivatives: better informed updates, but updates are more computationally expensive

2. Using momentum: can prevent models from getting stuck/being slow on certain loss landscapes

3. Setting adaptive learning rates: we can make smaller or larger updates depending on how important a feature is



Street-fighting math

Things we might want to know about a neural network:

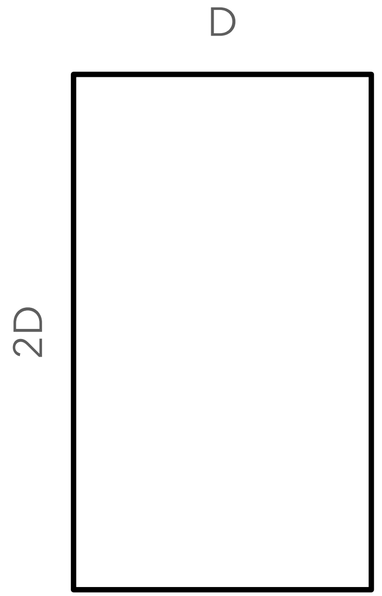
- How many parameters does it have?
- How much compute does it use?
- How much memory will it require?

How does this differ at inference time vs. training time? Let’s estimate it.

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D

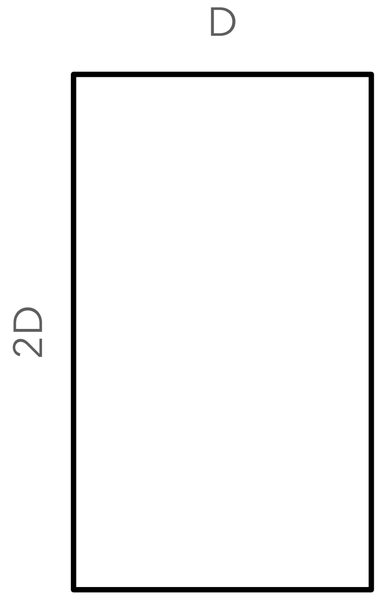


Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D

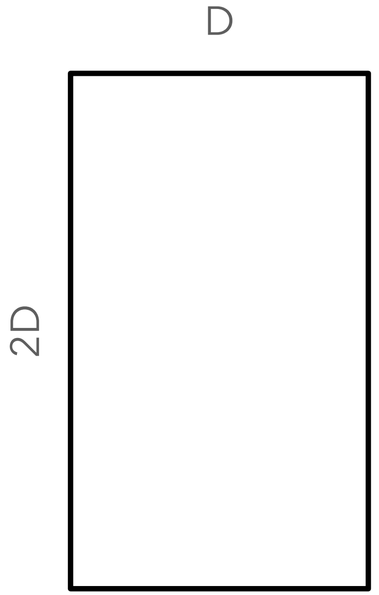


Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



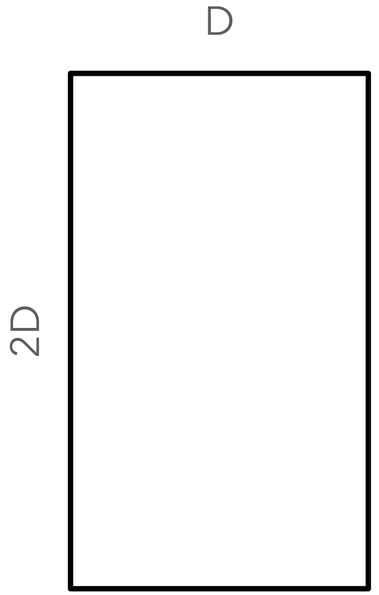
- Parameters: Dx2D

Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



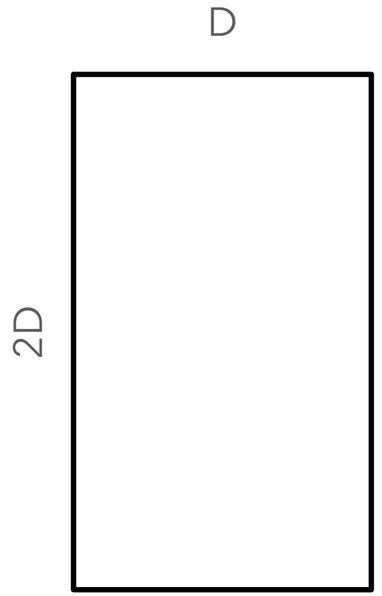
- Memory (inference):

Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



- Memory (inference):

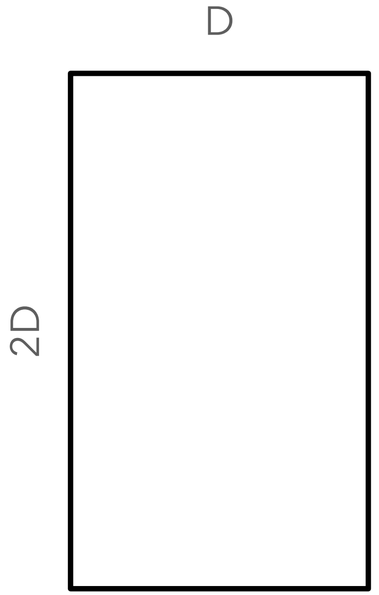
  Dx2D (parameters) +

Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



- Memory (inference):

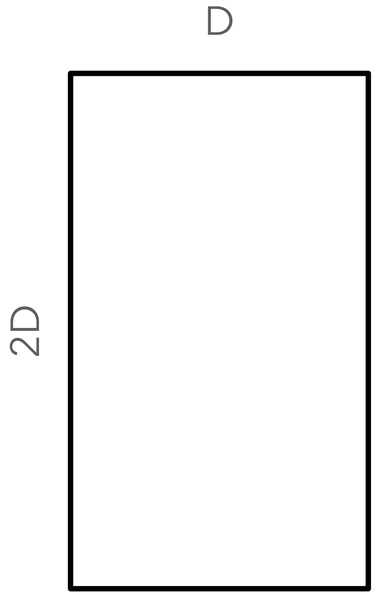
  Dx2D (parameters) + Bx2D (biggest activation)

Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



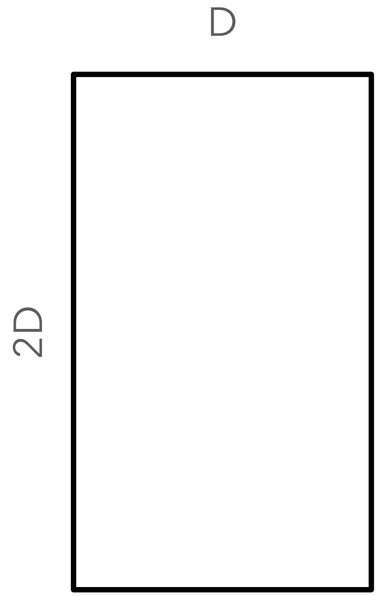
- Memory (training):

Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



- Memory (training):

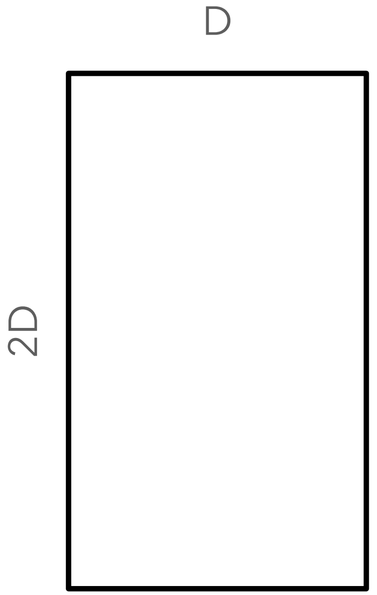
  Dx2D (parameters) +

Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



- Memory (training):

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t} \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

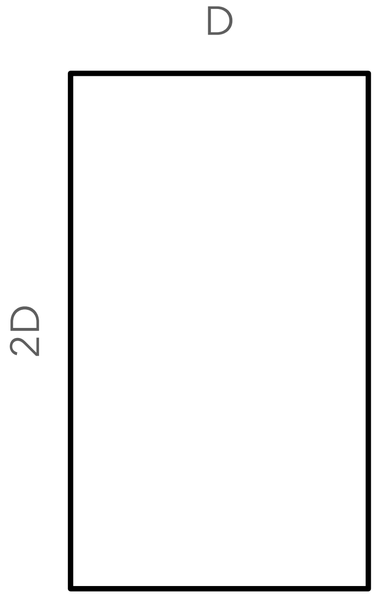
  Dx2D (parameters) +
  2xDx2D (optimizer state) +

Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



- Memory (training):

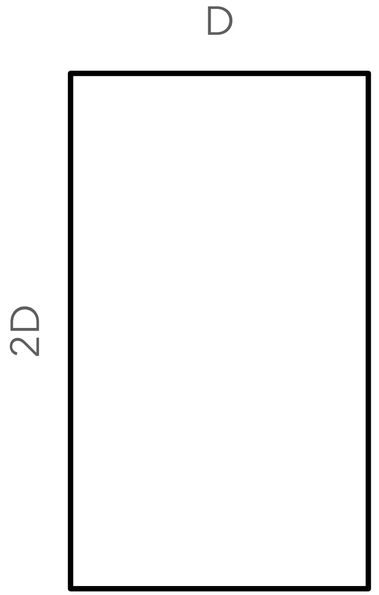
  Dx2D (parameters) +
  2xDx2D (optimizer state) +
  BxD + Bx2D (all activations)

Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



- Compute (inference):

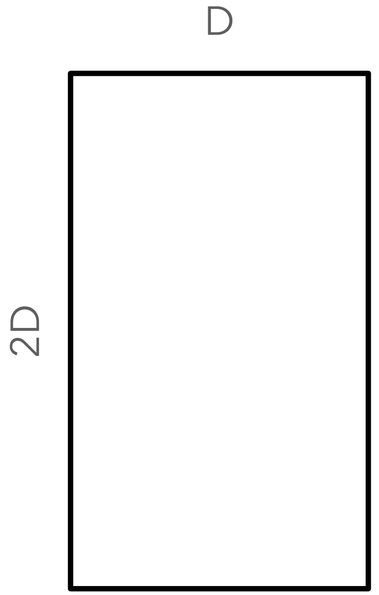
  BxDx2D (forward pass)

Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



- Compute (training):

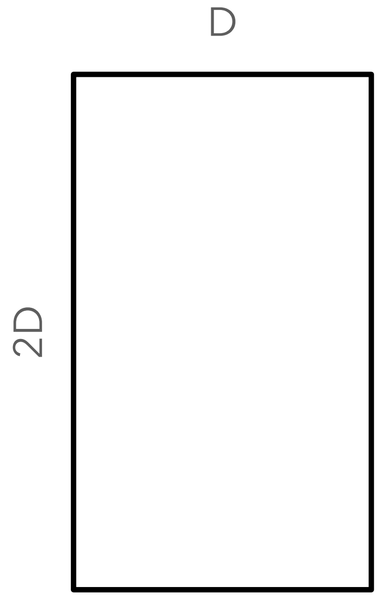
  BxDx2D (forward pass) +

Material adapted from Sasha Rush: Street-Fighting Transformers

Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D



- Compute (training):

  BxDx2D (forward pass) +
  2xBxDx2D (backward pass)

Material adapted from Sasha Rush: Street-Fighting Transformers

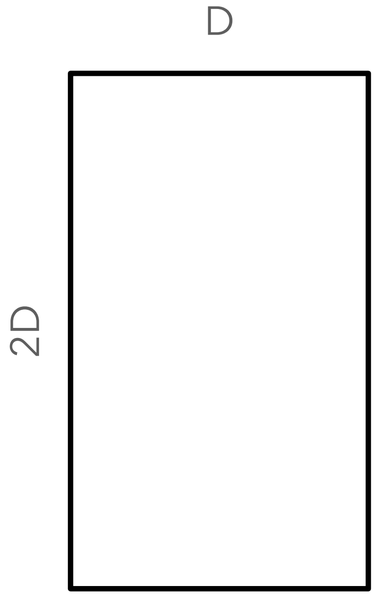
Street-fighting math

Let’s consider a simple neural network of size Dx2D:

- Batch size - B
- Hidden size - D
- Compute (training):

  BxDx2D (forward pass) +
  2xBxDx2D (backward pass)

Why 2x for the backward pass? Roughly: because we’re computing derivatives with respect to both the weights and the input



Material adapted from Sasha Rush: Street-Fighting Transformers