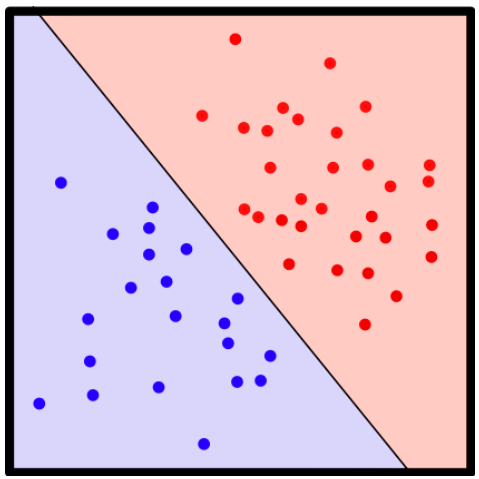
Lectures on Natural Language Processing

4. Introduction to Deep Learning

Karl Stratos

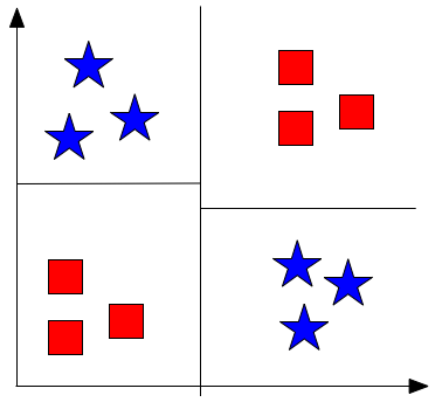
Review: Feature Learning

Linearly separable



Accuracy 100% ✓

Not linearly separable (e.g., XOR)



Accuracy $\leq 50\%$ ✗

Feature learning (aka. deep learning, neural networks)

1. Learn an input encoder $\mathbf{enc}_\theta : \mathbb{R}^d \to \mathbb{R}^H$ alongside linear classifier!
2. Use SGD to minimize a loss function differentiable in θ

Deep Learning: Definition

- A system that employs a hierarchy of features of the input, learned end-to-end jointly with the predictor.

$$\phi(x; \theta_1, \theta_2, \ldots, \theta_L) = F_1(x; \theta_1)$$

- We will refer to F_l as **layer** l

Deep Learning: Definition

- A system that employs a hierarchy of features of the input, learned end-to-end jointly with the predictor.

$$\phi(x; \theta_1, \theta_2, \ldots, \theta_L) = F_2(F_1(x; \theta_1); \theta_2)$$

- We will refer to F_l as **layer** l

Deep Learning: Definition

- A system that employs a hierarchy of features of the input, learned end-to-end jointly with the predictor.

$$\phi(x; \theta_1, \theta_2, \ldots, \theta_L) = F_L(F_{L-1}(\cdots F_2(F_1(x; \theta_1); \theta_2) \cdots); \theta_L)$$

  This is a vector in $\mathbb{R}^{D_L}$ (D_L: final feature dimension)
- We will refer to F_l as **layer** l

Deep Learning: Definition

- A system that employs a hierarchy of features of the input, learned end-to-end jointly with the predictor.

$$\phi(x; \theta_1, \theta_2, \ldots, \theta_L) = F_L(F_{L-1}(\cdots F_2(F_1(x; \theta_1); \theta_2) \cdots); \theta_L)$$

  This is a vector in $\mathbb{R}^{D_L}$ (D_L: final feature dimension)
- We will refer to F_l as **layer** l
- E.g., deep learning for classification:

$$f_y(x; W, b, \theta_1, \theta_2, \ldots, \theta_L) = w_y \cdot \phi(x; \theta_1, \theta_2, \ldots, \theta_L) + b_y$$

  - All parameters $(W, b, \theta_1, \theta_2, \ldots, \theta_L)$ are learned jointly
  - $\phi(x; \theta_1, \theta_2, \ldots, \theta_L) \in \mathbb{R}^{D_L}$: learned representation of x

Deep Learning: Definition

- A system that employs a hierarchy of features of the input, learned end-to-end jointly with the predictor.

$$\phi(x; \theta_1, \theta_2, \ldots, \theta_L) = F_L(F_{L-1}(\cdots F_2(F_1(x; \theta_1); \theta_2) \cdots); \theta_L)$$

  This is a vector in $\mathbb{R}^{D_L}$ (D_L: final feature dimension)
- We will refer to F_l as **layer** l
- E.g., deep learning for classification:

$$f_y(x; W, b, \theta_1, \theta_2, \ldots, \theta_L) = w_y \cdot \phi(x; \theta_1, \theta_2, \ldots, \theta_L) + b_y$$

  - All parameters $(W, b, \theta_1, \theta_2, \ldots, \theta_L)$ are learned jointly
  - $\phi(x; \theta_1, \theta_2, \ldots, \theta_L) \in \mathbb{R}^{D_L}$: learned representation of x
- Learning methods that are not deep:
  SVMs, nearest neighbor classifiers, decision trees, perceptron

Example: Feedforward Classifier

Encoder

- $\mathbf{enc}_{U,a} : \mathbb{R}^d \to \mathbb{R}^H$ defined by $\mathbf{enc}_{U,a}(x) = g(U^\top x + a)$
- Parameters: $U = [u_1 \dots u_H] \in \mathbb{R}^{d \times H}$ and $a \in \mathbb{R}^H$
- Nonlinear and (sub-)differentiable **activation** function $g : \mathbb{R} \to \mathbb{R}$, applied elementwise (i.e., $[g(z)]_i = g(z_i)$)

Linear classifier (K classes)

- Parameters: $W = [w_1 \dots w_K] \in \mathbb{R}^{H \times K}$ and $b \in \mathbb{R}^K$
- Model: $p_\theta(y|x) \propto \exp(w_y^\top \mathbf{enc}_{U,a}(x) + b)$

Training: Given $(x_1, y_1) \dots (x_N, y_N) \in \mathbb{R}^d \times \{1 \dots K\}$, minimize

$$\widehat{J}_N(\theta) = -\frac{1}{N} \sum_{i=1}^{N} \log p_\theta(y_i|x_i)$$

What is the gradient of $\widehat{J}_N$ with respect to $\theta = (W, b, U, a)$?

Linear Classifier Gradients

Define $h_i := \mathbf{enc}_{U,a}(x_i)$. Then

$$\widehat{J}_N(\theta) = \frac{1}{N}\sum_{i=1}^{N} \log\left(\sum_{y=1}^{K} \exp(w_y^\top h_i + b_y)\right) - w_{y_i}^\top h_i - b_{y_i}$$

h_i is not a function of (W, b), so we already know the gradients from before: for each $y \in \{1 \dots K\}$

$$\nabla_{w_y} \widehat{J}_N(\theta) = \frac{1}{N}\sum_{i=1}^{N} \left(p_\theta(y|x_i) - \mathbb{1}(y = y_i)\right) h_i$$

$$\nabla_{b_y} \widehat{J}_N(\theta) = \frac{1}{N}\sum_{i=1}^{N} p_\theta(y|x_i) - \mathbb{1}(y = y_i)$$

Feedforward Encoder Gradients

- $\widehat{J}_N(\theta)$ is a function of $U_{j,k} \in \mathbb{R}$ through $h_1 \dots h_N \in \mathbb{R}^H$.
- By the **chain rule**:

$$\frac{\partial \widehat{J}_N(\theta)}{\partial U_{j,k}} = \sum_{i=1}^{N} \underbrace{\left(\frac{\partial \widehat{J}_N(\theta)}{\partial h_i} \right)^\top}_{1 \times H} \underbrace{\frac{\partial h_i}{\partial U_{j,k}}}_{H \times 1}$$

- $\frac{\partial \widehat{J}_N(\theta)}{\partial h_i}$: Gradient of $\widehat{J}_N(\theta) \in \mathbb{R}$ wrt. $h_i \in \mathbb{R}^H$ (easy)
- $\frac{\partial h_i}{\partial U_{j,k}}$: **Jacobian** of $h_i \in \mathbb{R}^H$ wrt. $U_{j,k} \in \mathbb{R}$ (also easy)

$$\left[\frac{\partial h_i}{\partial U_{j,k}} \right]_t = \frac{\partial [h_i]_t}{\partial U_{j,k}}$$

Feedforward Encoder Gradients: Continued

Exercise: Verify that for $\delta_i := \sum_{y=1}^{K} p_\theta(y|x_i) w_y - w_{y_i} \in \mathbb{R}^H$

$$\frac{\partial \widehat{J}_N(\theta)}{\partial h_i} = \frac{1}{N}\delta_i$$
$$\frac{\partial h_i}{\partial U_{j,k}} = e_k \odot g'(Ux_i + a)[x_i]_j$$

where $e_k \in \{0,1\}^H$ is the k-th standard basis vector and $\odot$ is elementwise multiplication. Then

$$\nabla_U \widehat{J}_N(\theta) = \frac{1}{N}\sum_{i=1}^{N} \underbrace{x_i}_{d\times 1} \underbrace{(\delta_i \odot g'(U^\top x_i + a))^\top}_{1\times H} \in \mathbb{R}^{d\times H}$$

Use this to take a gradient step on $U \in \mathbb{R}^{d\times H}$, similarly for $a \in \mathbb{R}^H$

Forward and Backward Pass

Forward

$$
\begin{aligned}
z_i &= U^\top x_i + a && \mathbb{R}^H \\
h_i &= g(z_i) && \mathbb{R}^H \\
p_i &= \mathrm{softmax}(W^\top h_i + b) && [0,1]^K \\
J &= \frac{1}{N}\sum_{i=1}^{N} \log[p_i]_{y_i} && \mathbb{R}
\end{aligned}
$$

Backward (Gradients for W, b omitted)

$$
\begin{aligned}
\delta_i &= W p_i - w_{y_i} && \mathbb{R}^H \\
\nabla_U \widehat{J}_N(\theta) &= \frac{1}{N}\sum_{i=1}^{N} x_i(\delta_i \odot g'(z_i))^\top && \mathbb{R}^{d\times H} \\
\nabla_a \widehat{J}_N(\theta) &= \frac{1}{N}\sum_{i=1}^{N} \delta_i \odot g'(z_i) && \mathbb{R}^H
\end{aligned}
$$

Nonlinear Activation Function

- Nonlinear $g : \mathbb{R} \to \mathbb{R}$ crucial, otherwise we have a linear classifier again (assuming $H \geq \min\{d, K\}$)

$$\textbf{score}_\theta(x, y) = w_y^\top (U^\top x + a) + b_y = v_y^\top x + c_y$$

  where $V = UW \in \mathbb{R}^{d \times K}$ and $c = W^\top a + b$

- Popular activation functions

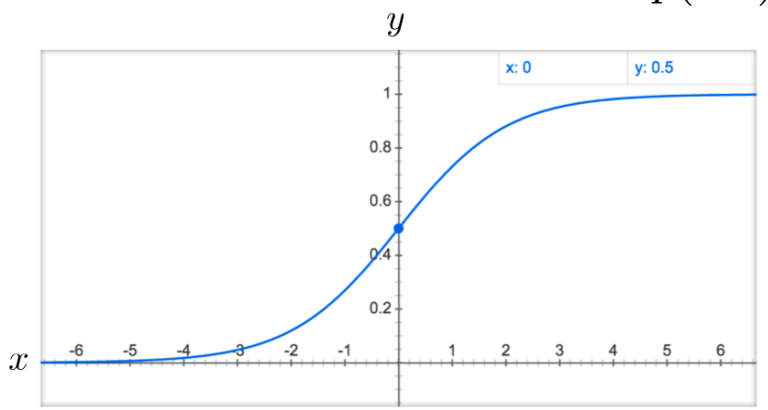
$$\mathrm{ReLU}(z) = \max\{0, z\} \qquad \mathrm{ReLU}'(z) = \begin{cases} 1 & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

$$\tanh(z) = \frac{\exp(2z) - 1}{\exp(2z) + 1} \qquad \tanh'(z) = 1 - \tanh(z)^2$$

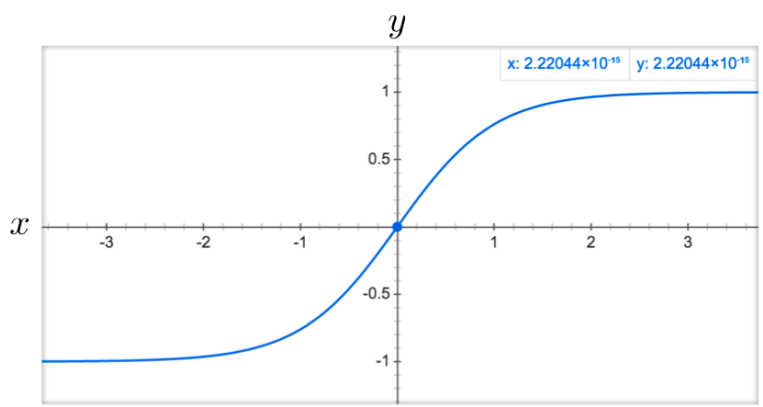
$$\sigma(z) = \frac{1}{1 + \exp(-z)} \qquad \sigma'(z) = \sigma(z)(1 - \sigma(z))$$

Popular Activation Functions

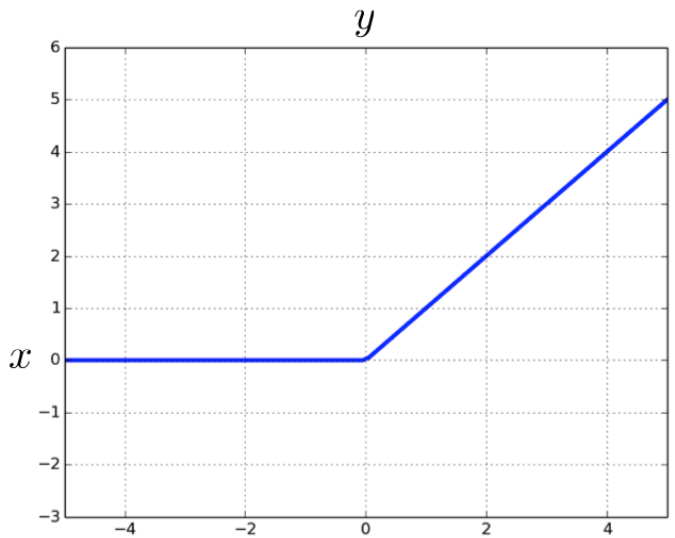
sigmoid, $y = \sigma(x) = \frac{1}{1+\exp(-x)}$



tanh, $y = \tanh(x)$

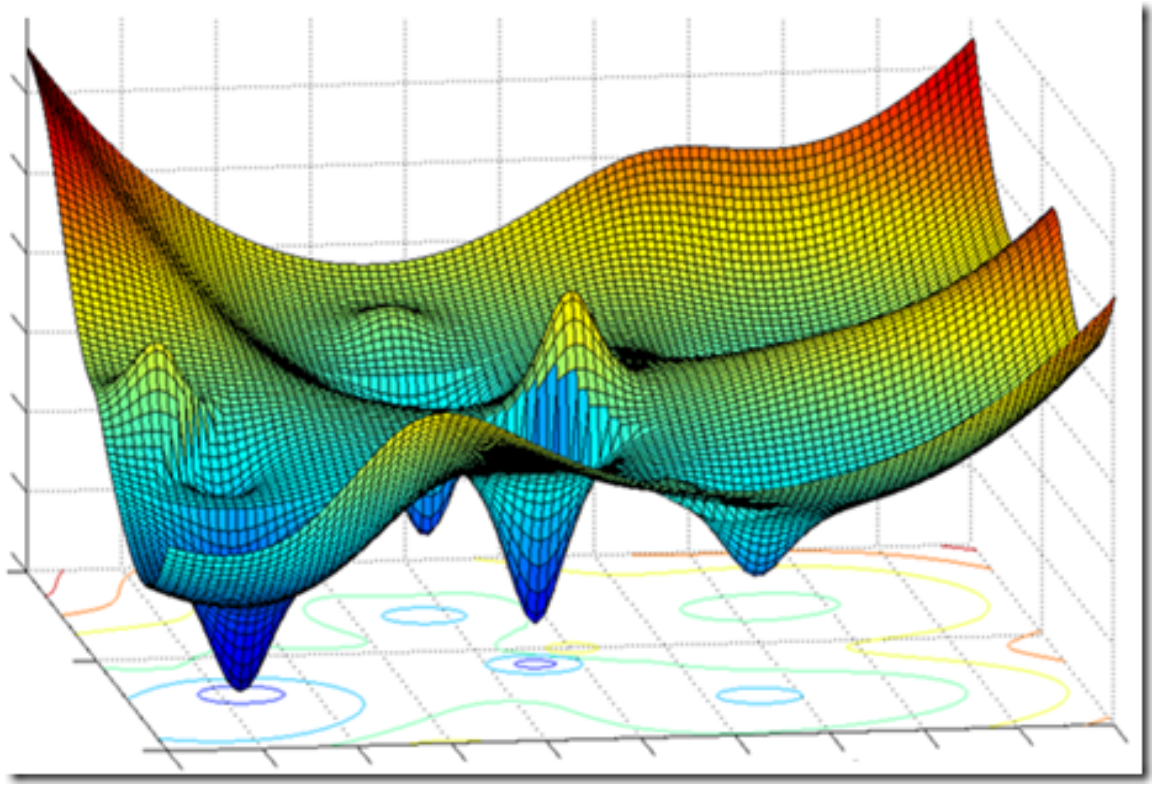


rectified linear unit (ReLU), $y = \max\{0, x\}$:



Nonconvex Objective

- $\widehat{J}_N$ is *not* convex in (U, a).

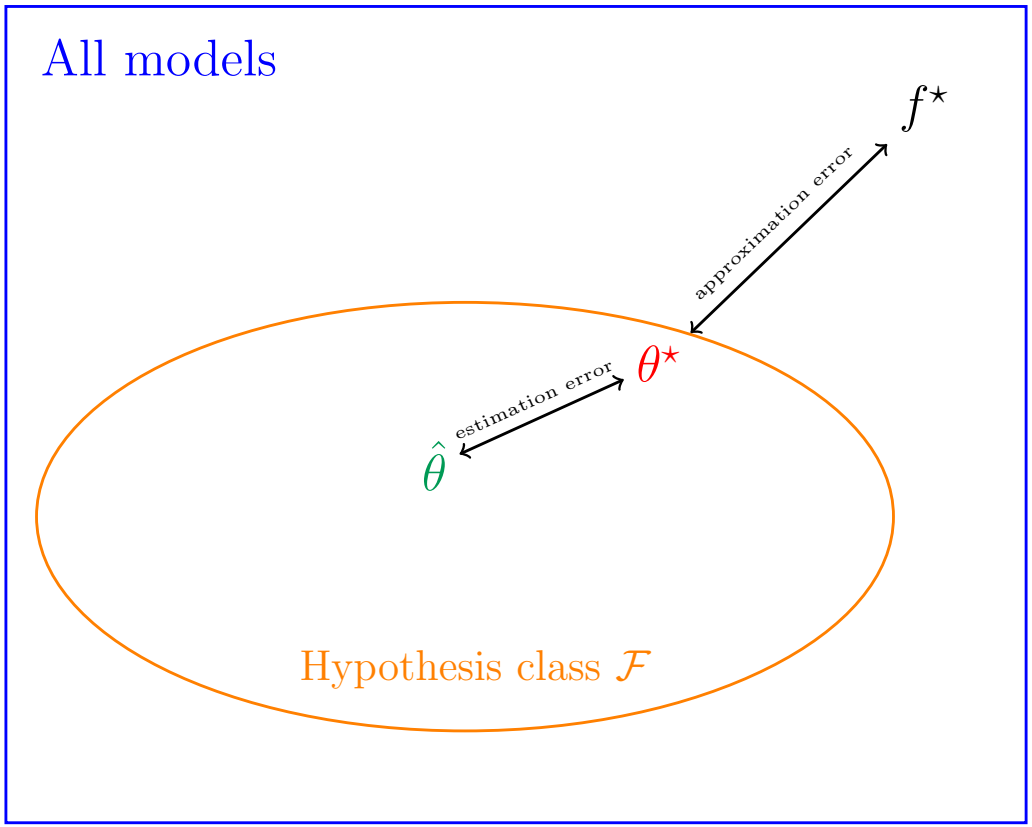


- Gradient descent will still find *some* stationary point.
  - But we don't really care if the stationary point is globally optimal for $\widehat{J}_N$ (in fact that might be bad due to overfitting)
  - What we care: performance on downstream task

Universal Learners

- Feedforward with a nonlinear layer is highly expressive
  - Can separate non-separable examples (see Jupyter Notebook)
- Natural question: What class of functions can it express?
- The answer any function!
  - ... **If** it has enough parameters
  - For this reason, we say neural networks are **universal learners**
- Nothing exciting: This simply says we can memorize all N examples if $H = O(N)$
- Active research on universality with limited number of parameters

Revisiting Approximation vs Estimation Error



- **Approximation error**: due to limited model expressiveness, remains no matter how much data we have
- **Estimation error**: due to limited data, can reduce by getting more data

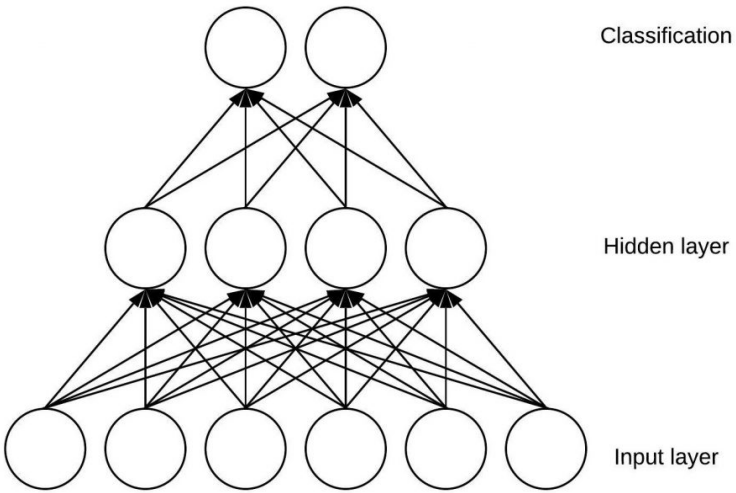
Deep learning: approximation error can always be driven to zero by considering a sufficiently large neural net

Regularization for Deep Learning

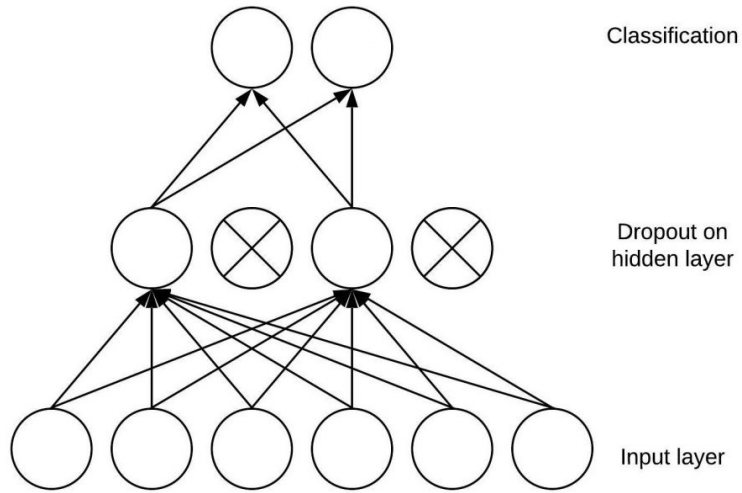
- Large neural networks can easily fit *random* labels.
- Same regularization techniques still useful: early stopping based on validation performance, weight decay
- Additional techniques
  - **Dropout**: Randomly make elements zero.
  - **Label smoothing**: Make one-hot label representation $\{0, 1\}^K$ assign nonzero probabilities to other labels.
  - **Layer normalization**: Standardize elements in a layer.
- Even without explicit regularization, large neural networks can generalize surprisingly well.
  - Some attribute this fact to *implicit* regularization under SGD: "Understanding deep learning requires rethinking generalization" (Zhang et al., 2016)
  - But, in practice, explicit regularization is usually a must

Dropout

"Drop" (i.e., make it zero) each weight value with probability $p \in [0, 1)$. Divide surviving weights by $1 - p$ to restore the overall size of weights.



Without Dropout



With Dropout

Idea: force the hidden layer to learn robust patterns, not memorize

- Only done for training: at test time no dropping or rescaling.
- How does this change the gradients?

Label Smoothing

- Cross-entropy loss $H(\mathbf{pop}(y|x), p_\theta(y|x))$
- Cross-entropy loss with label smoothing: $\alpha \in [0, 1]$

$$H((1-\alpha)\mathbf{pop}(y|x) + \alpha \text{Unif}(\{1 \dots K\}), p_\theta(y|x))$$

- $\alpha > 0$: Assign nonzero probabilities to labels other than gold ("soft targets")

$$\widehat{J}_N(\theta) = -\frac{1}{N}\sum_{i=1}^{N}(1-\alpha)\log p_\theta(y_i|x_i) + \frac{\alpha}{K}\sum_{y=1}^{K}\log p_\theta(y|x_i)$$

- Shown useful for machine translation and other tasks
  - See: "When Does Label Smoothing Help?" (Müller et al., 2019)

Layer Normalization

- Define **LayerNorm** : $\mathbb{R}^H \to \mathbb{R}^H$ by (for some tiny $\epsilon > 0$ to prevent division by zero)

$$\mu(h) := \frac{1}{H}\sum_{i=1}^{H} h_i \qquad \sigma^2(h) := \frac{1}{H}\sum_{i=1}^{H}(h_i - \mu(h))^2$$

$$\textbf{LayerNorm}_i(h) = \frac{h_i - \bar{h}}{\sqrt{\sigma^2(h) + \epsilon}} \quad \forall i = 1 \dots H$$

- This is a differentiable operation, so we will still be able to calculate gradients of the final loss with respect to parameters.
- If we treat vector elements as independent samples, $h' = \textbf{LayerNorm}(h)$ have zero mean and unit variance ("whitened" or "standardized").
  - Model can't overfit by making values wildly different
- Related method: batch normalization (normalization across elements in a batch)

Gradient Calculation

Deep learning is a shockingly flexible paradigm.

$$\textbf{enc}_\theta(x) = \mathrm{ReLU}(U^\top x + a) \qquad U \in \mathbb{R}^{d \times H},\ a \in \mathbb{R}^H$$

$$\textbf{enc}_\theta(x) = \tanh(U^\top \tanh(U^\top \mathrm{ReLU}(U^\top x))) \qquad U \in \mathbb{R}^{d \times d}$$

$$\textbf{enc}_\theta(x) = \textbf{LayerNorm}(\mathrm{ReLU}(V^\top \sigma(U^\top x))) \qquad U \in \mathbb{R}^{d \times H},\ V \in \mathbb{R}^{H \times H'}$$

Any of these encoders can be "plugged" into a linear classifier and trained by SGD on the cross-entropy loss (which remains differentiable).

- Central problem: Must derive gradients for every new loss/model!
- Instead of doing it by hand, can we automate this?

Automatic Differentiation and Backpropagation

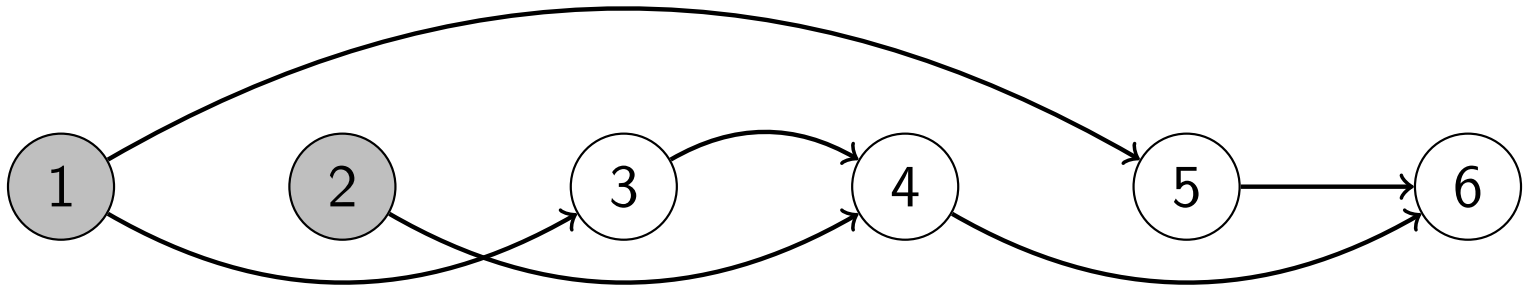
- Automatic differentiation (AD, autodiff) is widely-used in scientific computing
  - Machine learning, optimization, probabilistic programming (given a program, AD can compute its derivative)
- At a high level, AD has two "modes": forward and reverse
- Forward mode AD is best when your function outputs a vector and you have a relatively small number of inputs
- Reverse mode AD is best when your function outputs a scalar but has many inputs
- Which situation better characterizes machine learning?

Automatic Differentiation and Backpropagation

- Automatic differentiation (AD, autodiff) is widely-used in scientific computing
  - Machine learning, optimization, probabilistic programming (given a program, AD can compute its derivative)
- At a high level, AD has two "modes": forward and reverse
- Forward mode AD is best when your function outputs a vector and you have a relatively small number of inputs
- Reverse mode AD is best when your function outputs a scalar but has many inputs
- Which situation better characterizes machine learning?
- **Backpropagation = reverse mode AD**
  - DAG + chain rule

DAG

A **directed acylic graph (DAG)** is a directed graph $G = (V, A)$ with a **topological ordering**



$$V = \{1, 2, 3, 4, 5, 6\}\,,\ V_I = \{1, 2\}\,,\ V_N = \{3, 4, 5, 6\}$$

$$A = \{(1, 3), (1, 5), (2, 4), (3, 4), (4, 6), (5, 6)\}$$

$$\mathbf{pa}(4) = \{2, 3\}$$

$$\mathbf{ch}(1) = \{3, 5\}$$

$$\Pi_G = \{(1, 2, 3, 4, 5, 6),\ (2, 1, 3, 4, 5, 6)\}$$ (possible topological orderings)

For backpropagation: usually assume have many roots and 1 leaf

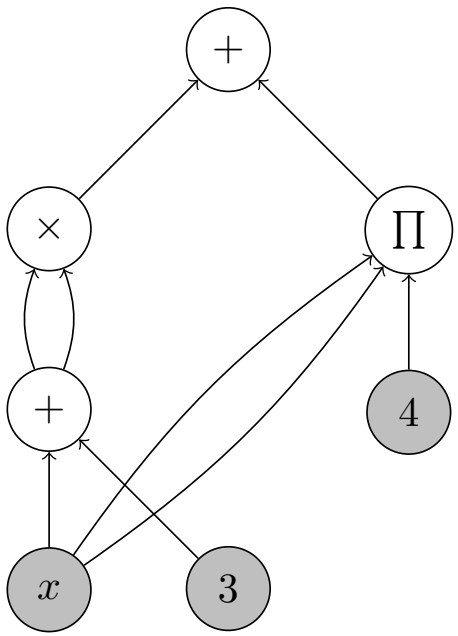
Computation Graph

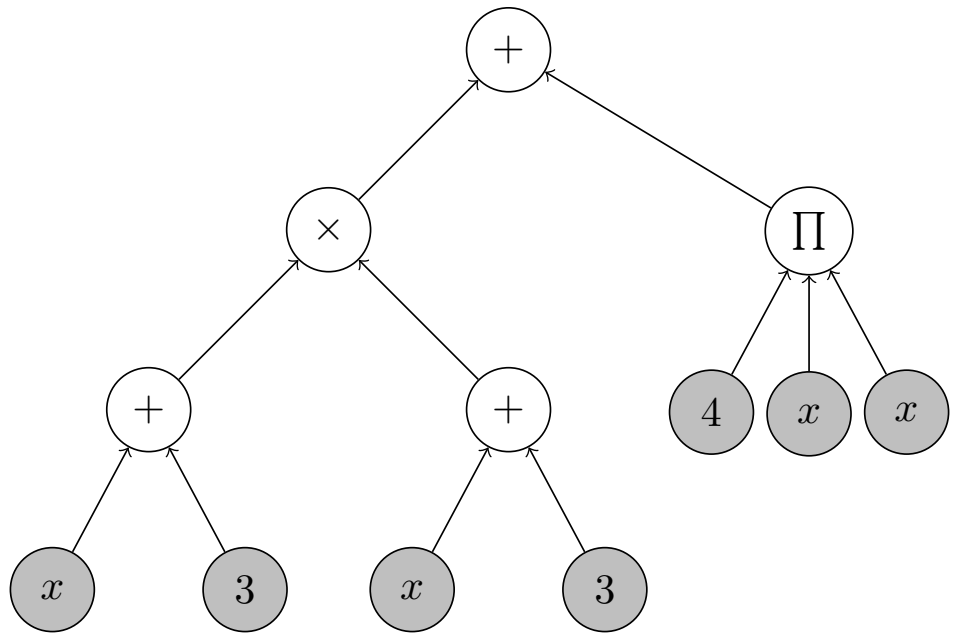
- DAG $G = (V, A)$ with a single output node $\omega \in V$.
- Every node $i \in V$ is equipped with a **value** $x^i \in \mathbb{R}$:
  1. For input node $i \in V_I$, we assume $x^i = a^i$ is given.
  2. For non-input node $i \in V_N$, we assume a differentiable **function** $f^i : \mathbb{R}^{|\mathbf{pa}(i)|} \to \mathbb{R}$ and compute

$$x^i = f^i((x^j)_{j \in \mathbf{pa}(i)})$$

- Thus G represents a *function* $\left\{a^i\right\}_{i \in V_I} \mapsto x^\omega$
- **Forward pass**
  1. Pick some topological ordering $\pi \in \Pi_G$
  2. For i in order of π, if $i \in V_N$ is a non-input node, set $x^i \leftarrow a^i := f^i((a^j)_{j \in \mathbf{pa}(i)})$
- Forward pass populates $x^i = a^i$ for every $i \in V$.

Multiple Possible Computation Graphs





These two computation graphs represent the same expression $(x+3)^2+4x^2$ but first has fewer nodes/edges.

Forward Pass: Populate Value Slots

Construct the computation graph associated with the function

$$f(x, y) := (x + y)xy^2$$

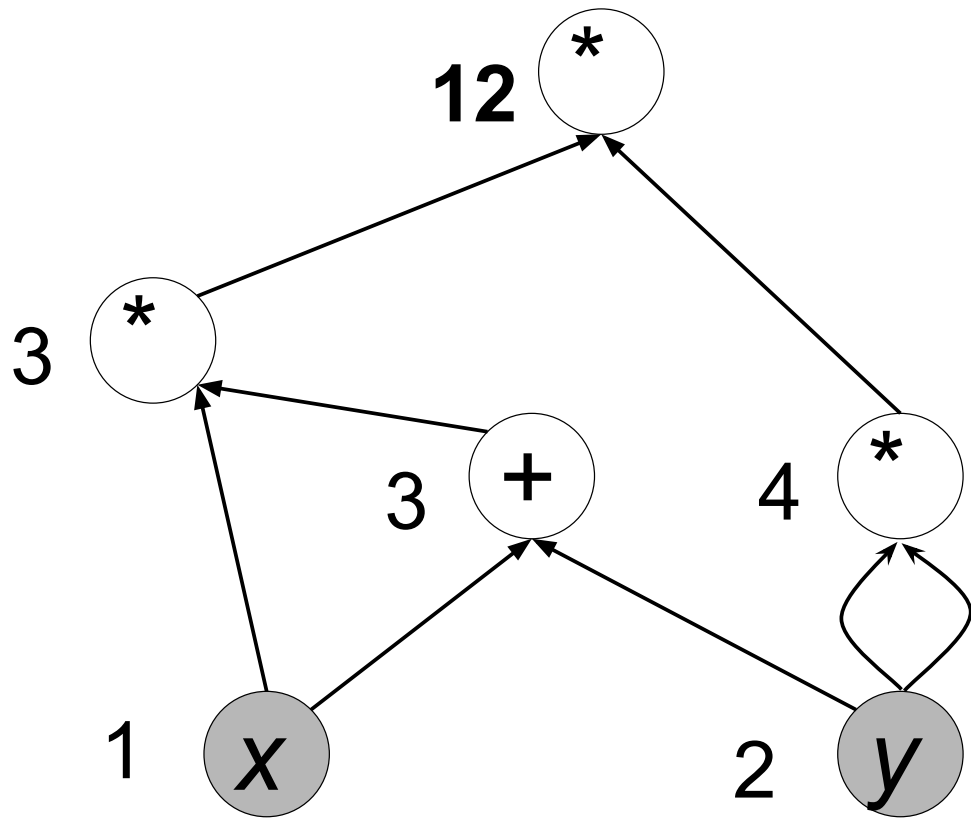
Compute its output value at $x = 1$ and $y = 2$ by performing a forward pass.

Forward Pass: Populate Value Slots

Construct the computation graph associated with the function

$$f(x, y) := (x + y)xy^2$$

Compute its output value at $x = 1$ and $y = 2$ by performing a forward pass.



Gradient Slots

- Notation: Input slots $x_I = (x^i)_{i \in V_I}$, their values $a_I = (a^i)_{i \in V_I}$
- For every node $i \in V$, we introduce an additional slot $z^i \in \mathbb{R}$ storing the gradient of x^ω wrt. x^i at $x_I = a_I$:

$$z^i := \left. \frac{\partial x^\omega}{\partial x^i} \right|_{x_I = a_I}$$

- **Goal of backpropagation**: Calculate z^i for every $i \in V$.

Key Ideas of Backpropagation

- Notation: Parental slots $x_I^i = (x^j)_{j \in \mathbf{pa}(i)}$, their values $a_I^i = (a^j)_{j \in \mathbf{pa}(i)}$
- Chain rule on the DAG structure

$$z^i := \left.\frac{\partial x^\omega}{\partial x^i}\right|_{x_I = a_I} = \sum_{j \in \mathbf{ch}(i)} \left.\frac{\partial x^\omega}{\partial x^j}\right|_{x_I = a_I} \times \left.\frac{\partial x^j}{\partial x^i}\right|_{x_I^j = a_I^j}$$

Key Ideas of Backpropagation

- Notation: Parental slots $x_I^i = (x^j)_{j \in \mathbf{pa}(i)}$, their values $a_I^i = (a^j)_{j \in \mathbf{pa}(i)}$
- Chain rule on the DAG structure

$$z^i := \left.\frac{\partial x^\omega}{\partial x^i}\right|_{x_I = a_I} = \sum_{j \in \mathbf{ch}(i)} \left.\frac{\partial x^\omega}{\partial x^j}\right|_{x_I = a_I} \times \left.\frac{\partial x^j}{\partial x^i}\right|_{x_I^j = a_I^j}$$

$$= \sum_{j \in \mathbf{ch}(i)} z^j \times \underbrace{\frac{\partial f^j(x_I^j)}{\partial x^i}}_{\text{Jacobian of } f^j \text{ wrt } x^i} \Bigg|_{x_I^j = a_I^j}$$

Key Ideas of Backpropagation

- Notation: Parental slots $x_I^i = (x^j)_{j \in \mathbf{pa}(i)}$, their values $a_I^i = (a^j)_{j \in \mathbf{pa}(i)}$
- Chain rule on the DAG structure

$$z^i := \left.\frac{\partial x^\omega}{\partial x^i}\right|_{x_I = a_I} = \sum_{j \in \mathbf{ch}(i)} \left.\frac{\partial x^\omega}{\partial x^j}\right|_{x_I = a_I} \times \left.\frac{\partial x^j}{\partial x^i}\right|_{x_I^j = a_I^j}$$

$$= \sum_{j \in \mathbf{ch}(i)} z^j \times \underbrace{\frac{\partial f^j(x_I^j)}{\partial x^i}}_{\text{Jacobian of } f^j \text{ wrt } x^i} \Bigg|_{x_I^j = a_I^j}$$

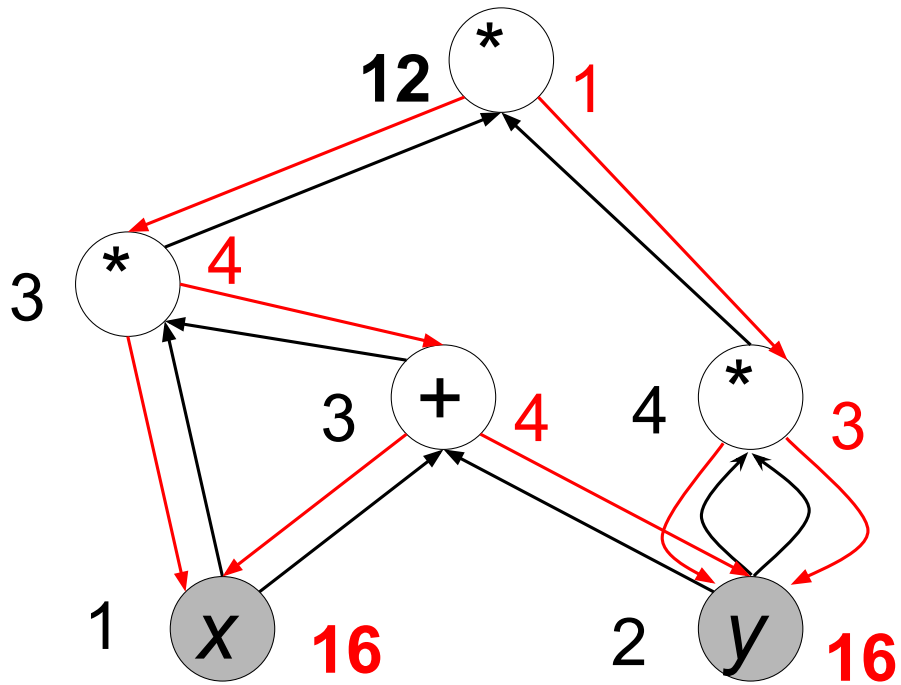
- **Backward pass**
  1. Base case: $z^\omega = 1$
  2. For i in reverse order of π: $z^i \leftarrow \sum_{j \in \mathbf{ch}(i)} z^j \times \left.\frac{\partial f^j(x_I^j)}{\partial x^i}\right|_{x_I^j = a_I^j}$

Backward Pass: Populate Gradient Slots

Calculate the gradient of $f(x, y) := (x + y)xy^2$ with respect to x at $x = 1$ and $y = 2$ by performing backpropagation.

Backward Pass: Populate Gradient Slots

Calculate the gradient of $f(x, y) := (x + y)xy^2$ with respect to x at $x = 1$ and $y = 2$ by performing backpropagation.



$$\left.\frac{\partial f(x,y)}{\partial x}\right|_{(x,y)=(1,2)} = 16$$

Implementation

- Each type of function f creates a child node from parent nodes and initializes its gradient to zero.
  - "Add" function creates a child node c with two parents (a, b) and sets $c.z \leftarrow 0$.
- Each node has an associated **forward** function.
  - Calling forward at c populates $c.x = a.x + b.x$ (assumes parents have their values).
- Each node also has an associated **backward** function.
  - Calling backward at c "broadcasts" its gradient $c.z$ (assumes it's already calculated) to its parents

$$a.z \leftarrow a.z + c.z$$
$$b.z \leftarrow b.z + c.z$$

- In deep learning, input nodes are model parameters, output node is scalar loss.
  - Once we run the forward and backward pass, gradient of the loss wrt. model parameters stored in the input nodes.

Multi-Variable Case

- Computation graph in which input values that are vectors

$$x^i \in \mathbb{R}^{d^i} \qquad \forall i \in V$$

But the output value $x^\omega \in \mathbb{R}$ is always a scalar

- Gradients: vectors of the same size!

$$z^i \in \mathbb{R}^{d^i} \qquad \forall i \in V$$

- Backpropagation: same form using the generalized chain rule

$$z^i = \sum_{j \in \mathbf{ch}(i)} \underbrace{\left.\frac{\partial x^\omega}{\partial x^j}\right|_{x_I = a_I}}_{1 \times d^j} \times \underbrace{\left.\frac{\partial x^j}{\partial x^i}\right|_{x_I^j = a_I^j}}_{d^j \times d^i}$$

$$= \sum_{j \in \mathbf{ch}(i)} z^j \times \underbrace{\frac{\partial f^j(x_I^j)}{\partial x^i}}_{\text{Jacobian of } f^j \text{ wrt. } x^i} \Bigg|_{x_I^j = a_I^j}$$

Standard Layers

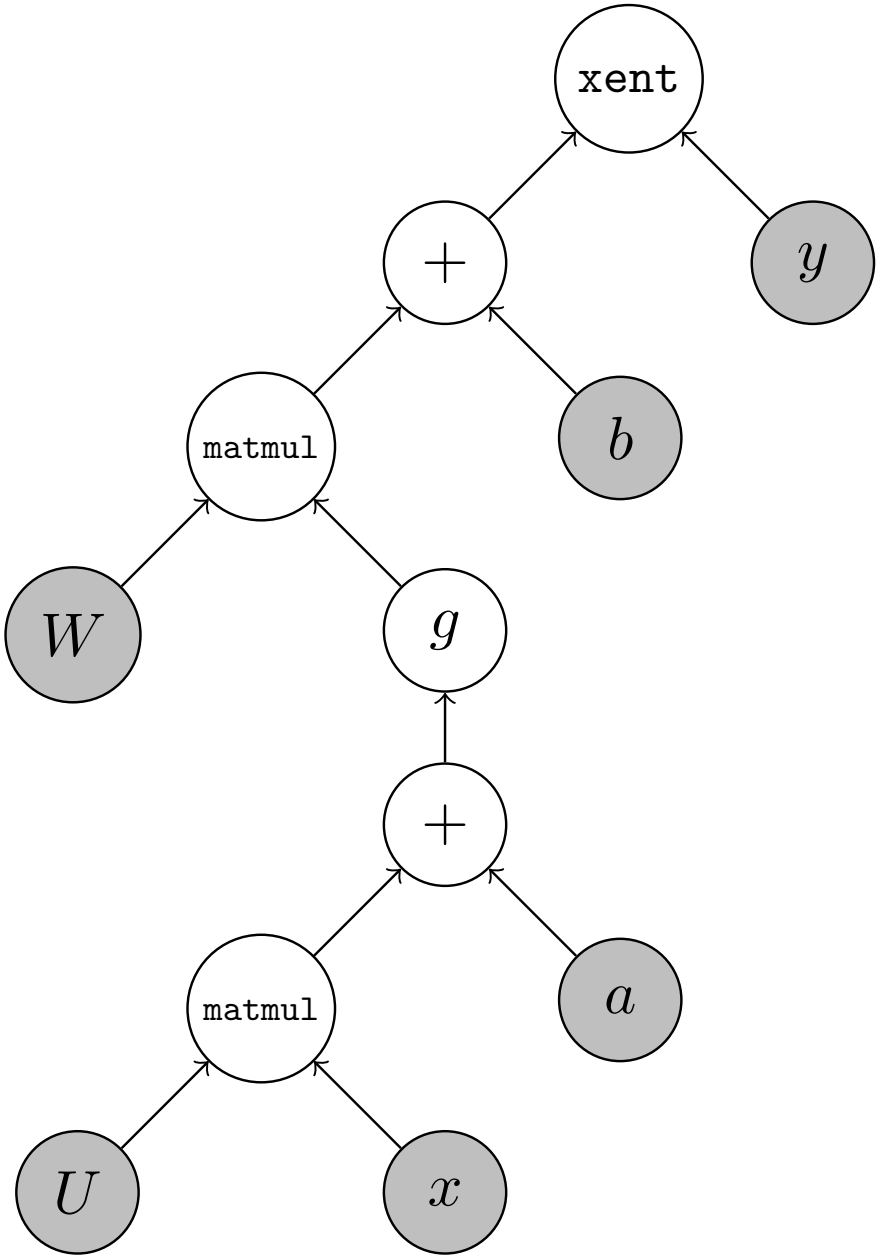
Deep learning libraries provide many pre-defined nodes (aka. layers)

- Element-wise addition $f(x, y) = x + y$, product $f(x, y) = x \odot y$
- Element-wise log $f(x) = \log(x)$, exponentiation $f(x) = \exp(x)$
- Scalar mult. $f(x, \alpha) = \alpha x$, matrix-vector product $f(A, x) = Ax$
- Softmax: $f(u) = \mathrm{softmax}(u)$
- Cross-entropy loss:
  $f([l_1 \dots l_N], (y_1 \dots y_N)) = -(1/N) \sum_i \log \mathrm{softmax}_{y_i}(l_i)$
- Dropout with probability p: $f(u) = \mathbf{Drop}_p(u)$

Each has its own forward and backward function, can plug and play

- Still have to be careful with numerical stability (e.g., always use an explicit cross-entropy loss layer, rather than using softmax which has unstable gradient)
- Syntactic sugar: "$z = x + y$" creates a computation graph under the hood

Loss of Feedforward Classifier



- Single-example loss

$$z = Ux + a$$
$$h = g(z)$$
$$l = Wh + b$$
$$J = -\log \text{softmax}_y(l)$$

- In practice, batch many examples into one computation graph
- (No transpose needed, shape weights appropriately)

Aside: Dropout Implementation

- Forward: Stochastically define a masking vector scaled by $(1-p)$, and save it for backward

$$\texttt{mask} = \left(\frac{\cdot}{0.7}, 0, \frac{\cdot}{0.7}\right)$$

- Backward: Use saved mask to threshold/scale child gradient

$$\textbf{Drop}_{0.3}^{\texttt{mask}}((u_1, u_2, u_3)) = \left(\frac{u_1}{0.7}, 0, \frac{u_3}{0.7}\right)$$

$$\frac{\partial \textbf{Drop}_{0.3}^{\texttt{mask}}((u_1, u_2, u_3))}{\partial (u_1, u_2, u_3)} = \begin{bmatrix} \frac{1}{0.7} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \frac{1}{0.7} \end{bmatrix}$$

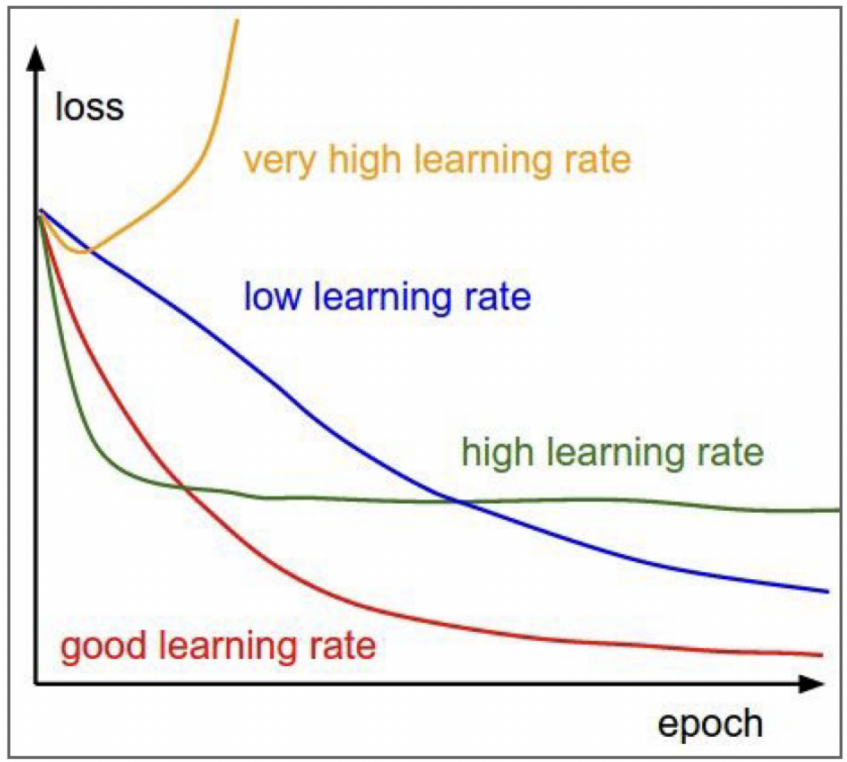
$$(z_1, z_2, z_3)\frac{\partial \textbf{Drop}_{0.3}^{\texttt{mask}}((u_1, u_2, u_3))}{\partial (u_1, u_2, u_3)} = \underbrace{\textbf{Drop}_{0.3}^{\texttt{mask}}((z_1, z_2, z_3))}_{\text{equiv. to applying same dropout to grad}}$$

Initialization Strategies

- Non-convex objective; initialization is important
- All zeros? Bad idea: all units learn the same thing!
- Random: small values (e.g., $\mathcal{N}(0, .01), \text{Unif}(-0.01, 0.01)$)
  - Problem: variance of *activation* grows with number of inputs
- The "Xavier" scheme (Glorot et al.): normalize the scale to provide roughly equal variance throughout the network
  - If n inputs, draw from $\mathcal{N}(\mu = 0, \sigma^2 = 1/n)$
  - Problem: implicitly assumes linear activations, breaks with ReLUs
- The "Kaiming" scheme (He et al): designed for ReLUs
  - Draw from $\mathcal{N}(0, 2/n)$, where n is the number of inputs
- Note: still OK to init biases with zeros

Learning Rate for Neural Networks

- For deep networks, setting the right learning rate is crucial.
- Typical behaviors, monitoring *training loss*:



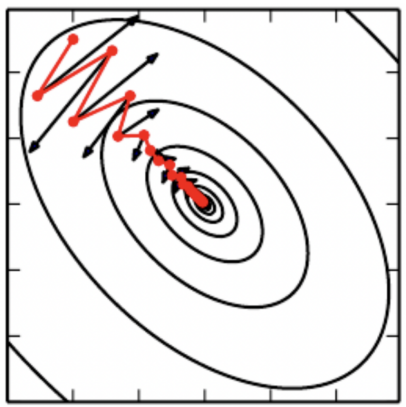
(A. Karpathy)

- High LR $\rightarrow$ NaN crash, usually fixable by making LR smaller

Gradient Descent with Momentum

- SGD has trouble navigating "ravines" where surface curves much more steeply in one dimension than in another,
- SGD oscillates across the slopes of the ravine, making hesitant progress towards the (local) optimum.
- Momentum helps accelerate SGD in the relevant direction and dampens oscillations.

$$\Delta\theta_t = \gamma\Delta\theta_{t-1} + \eta_t \nabla J(\theta_t)$$
$$\theta_{t+1} = \theta_t - \Delta\theta_t$$



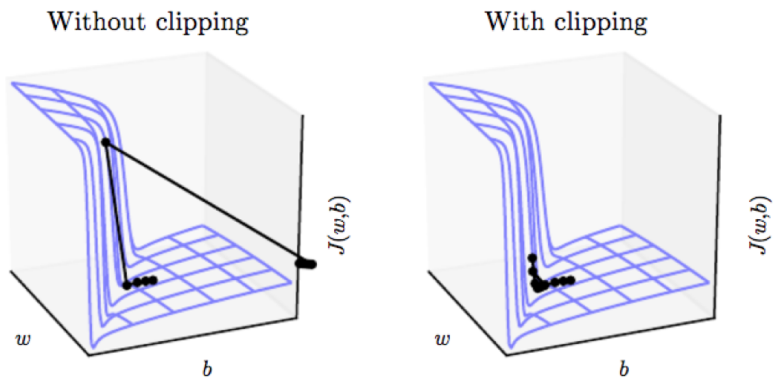
(Goodfellow et al.)

Gradient Clipping

- Because of nonlinearity gradient vectors can "explode"
  - Particularly problematic if the network has many layers (e.g., recurrent). Why? Result: NaN loss
- Helpful trick: clip gradient update to have norm at most C

$$\Delta\theta \mapsto C\frac{\Delta\theta}{||\Delta\theta||}$$

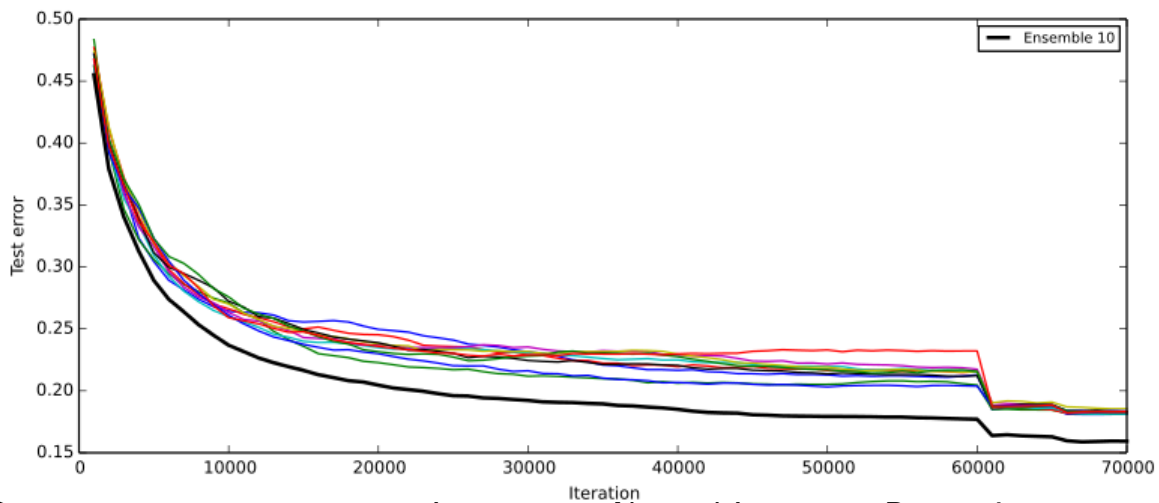
- Intuition: navigate steep local areas more conservatively



- "Never hurts": can set C to be large to turn it off

Ensembles of networks

- We may want to train multiple networks and somehow combine them
- Reduces variance (we have stochastic training of non-convex objective)
- Directly average the network weights? Terrible idea
- Averaging unit activations: equally bad
- Better idea: average the predictions
- Multi-class settings: output of network t is $p_t = (p_t(y = 1), \ldots, p_t(y = K))$ then use $\frac{1}{T} \sum_t p_t$



Need for Specialized Neural Architectures

- Feedforward implicitly assumes the input is a single vector.
- NLP: Input is a *sequence* (of symbols)!
- Option 1: BOW representation
  - Loses lots of information (e.g., ordering), high-dimensional
- Option 2: Giant feedforward with input dimension = max sequence length
  - Computationally intractable, too many parameters to learn
- Solution: Develop specialized architectures that can handle *variable* input lengths.
  - Example: Convolutional, recurrent, transformer
- Important to keep in mind: These specialized architectures are still "feedforward" (with weight sharing)
  - Feedforward is the building block of deep learning.