Review: Overfitting

Model succeeds in fitting (finite) training data by exploiting spurious input-label correlations that do not generalize.

Guard against overfitting by always setting aside a validation set. Regularize by early stopping, weight penalty, and other methods.
**Gradient descent**: Start from some $\theta_0 \in \mathbb{R}^d$, repeatedly minimize local approx. of $f$ around $\theta_t$ by

$$\theta_{t+1} = \theta_t - \eta_t \nabla f(\theta_t)$$

“learning rate” gradient of $f$ at $\theta_t$

**Stochastic**: If $f$ is an average of “component” functions, can quickly estimate $\nabla f$ from a mini-batch
Gradient Descent Convergence

- Generally, for convex functions, gradient descent will converge
  - Stop by (a combination of): max number of iterations, plateau in validation error, and other criteria
- The learning rate $\eta$ may be very important to ensure rapid convergence (or convergence at all)

(LeCun et al, 1996)
Review: Feature Learning

Feature learning (aka. deep learning, neural networks)

1. Learn an input encoder $\text{enc}_\theta : \mathbb{R}^d \rightarrow \mathbb{R}^H$ alongside linear classifier!

2. Use SGD to minimize a loss function differentiable in $\theta$
Deep Learning: Definition

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

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

- We will refer to \( F_k \) as layer \( k \)

- E.g., deep learning for classification:

\[ f_c(x; w, b, \theta_1, \theta_2, \ldots, \theta_L) = w_c \cdot f(x; \theta_1, \theta_2, \ldots, \theta_L) + b_c \]

- All parameters \((w, b, \theta_1, \theta_2, \ldots, \theta_L)\) are learned jointly

- We can think of \( f(x; \theta_1, \theta_2, \ldots, \theta_L) \) as learned features for \( x \)
  or a learned representation of \( x \) (doesn’t depend on the class being scored)

- Learning methods that are not deep: SVMs, nearest neighbor classifiers, decision trees, perceptron
Example: Feedforward Classifier

Encoder

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

Linear classifier (\( L \) classes)

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

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

\[
\hat{J}_N(\theta) = -\frac{1}{N} \sum_{i=1}^{N} p_\theta(y_i|x_i)
\]

Central question: What is the gradient of \( \hat{J}_N \) with respect to \( \theta = (W, b, U, a) \)?
Linear Classifier Gradients

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

$$\hat{J}_N(\theta) = \frac{1}{N} \sum_{i=1}^{N} \log \left( \sum_{y=1}^{L} \exp(w_y^\top h_i + b_y) \right) - w_y^\top h_i - b_y$$

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

$$\nabla_{w_y} \hat{J}_N(W, b) = \frac{1}{N} \sum_{i=1}^{N} \left( p_\theta(y|x_i) - \left[ [y = y_i] \right] \right) h_i$$

$$\nabla_{b_y} \hat{J}_N(W, b) = \frac{1}{N} \sum_{i=1}^{N} p_\theta(y|x_i) - \left[ [y = y_i] \right]$$
Feedforward Encoder Gradients

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

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

- $\frac{\partial \hat{J}_N(\theta)}{\partial h_i}$: Gradient of $\hat{J}_N(\theta) \in \mathbb{R}$ 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 \( \delta_i := \sum_{y=1}^{L} p_{\theta}(y|x_i)w_y - w_{y_i} \in \mathbb{R}^H \)

\[
\frac{\partial \hat{J}_N(\theta)}{\partial h_i} = \frac{1}{N} \delta_i
\]

\[
\frac{\partial h_i}{\partial U_{j,k}} = e_k \odot g'(Ux_i + a)[x_i]_j
\]

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

\[
\nabla_U \hat{J}_N(\theta) = \frac{1}{N} \sum_{i=1}^{N} x_i \left( \delta_i \odot g'(U^T x_i + a) \right)^\top \in \mathbb{R}^{d \times H}
\]

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

\[ z_i = U^\top x_i + a \quad \mathbb{R}^H \]
\[ h_i = g(z_i) \quad \mathbb{R}^H \]
\[ p_i = \text{softmax}(W^\top h_i + b) \quad [0, 1]^L \]
\[ J = \frac{1}{N} \sum_{i=1}^{N} \log[p_i]_{y_i} \quad \mathbb{R} \]

Backward (Gradients for \( W, b \) omitted)

\[ \delta_i = Wp_i - w_{y_i} \quad \mathbb{R}^H \]
\[ \nabla_U \hat{J}_N(\theta) = \frac{1}{N} \sum_{i=1}^{N} x_i (\delta_i \odot g'(z_i))^\top \quad \mathbb{R}^{d \times H} \]
\[ \nabla_a \hat{J}_N(\theta) = \frac{1}{N} \sum_{i=1}^{N} \delta_i \odot g'(z_i) \quad \mathbb{R}^H \]
Nonlinear Activation Function

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

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

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

- Popular activation functions

\[
\begin{align*}
\text{ReLU}(z) &= \max\{0, z\} & \text{ReLU}'(z) &= \begin{cases} 
1 & \text{if } z \geq 0 \\
0 & \text{otherwise}
\end{cases} \\
\text{tanh}(z) &= \frac{\exp(2z) - 1}{\exp(2z) + 1} & \text{tanh}'(z) &= 1 - \text{tanh}(z)^2 \\
\sigma(z) &= \frac{1}{1 + \exp(-z)} & \sigma'(z) &= \sigma(z)(1 - \sigma(z))
\end{align*}
\]
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

- $\hat{J}_N$ 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 $\hat{J}_N$ (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 turns out to be “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
Universality of Feedforward

Claim. Given any \((x_1, y_1) \ldots (x_N, y_N) \in \mathbb{R}^d \times \mathbb{R}\) (assume \(x_i\) distinct), there exists a feedforward network \(f : \mathbb{R}^d \to \mathbb{R}\) with \(2N + d\) parameters such that \(f(x_i) = y_i\) for all \(i = 1 \ldots N\).
Universality of Feedforward

**Claim.** Given any \((x_1, y_1) \ldots (x_N, y_N) \in \mathbb{R}^d \times \mathbb{R}\) (assume \(x_i\) distinct), there exists a feedforward network \(f : \mathbb{R}^d \to \mathbb{R}\) with \(2N + d\) parameters such that \(f(x_i) = y_i\) for all \(i = 1 \ldots N\).

**Proof.** (Zhang et al., 2016)

1. Find \(a \in \mathbb{R}^N\) so that \(z_i = a^\top x_i\) are distinct.
Universality of Feedforward

**Claim.** Given any \((x_1, y_1), \ldots, (x_N, y_N) \in \mathbb{R}^d \times \mathbb{R}\) (assume \(x_i\) distinct), there exists a feedforward network \(f : \mathbb{R}^d \to \mathbb{R}\) with \(2N + d\) parameters such that \(f(x_i) = y_i\) for all \(i = 1 \ldots N\).

**Proof.** (Zhang et al., 2016)

1. Find \(a \in \mathbb{R}^N\) so that \(z_i = a^\top x_i\) are distinct.
2. WLOG assume \(z_1 < z_2 < \cdots < z_N\).
3. Find \(b \in \mathbb{R}^N\) so that \(b_1 < z_1 < b_2 < z_2 < \cdots < b_N < z_N\).
Universality of Feedforward

Claim. Given any \((x_1, y_1) \ldots (x_N, y_N) \in \mathbb{R}^d \times \mathbb{R}\) (assume \(x_i\) distinct), there exists a feedforward network \(f : \mathbb{R}^d \rightarrow \mathbb{R}\) with \(2N + d\) parameters such that \(f(x_i) = y_i\) for all \(i = 1 \ldots N\).

Proof. (Zhang et al., 2016)

1. Find \(a \in \mathbb{R}^N\) so that \(z_i = a^\top x_i\) are distinct.
2. WLOG assume \(z_1 < z_2 < \cdots < z_N\).
3. Find \(b \in \mathbb{R}^N\) so that \(b_1 < z_1 < b_2 < z_2 < \cdots < b_N < z_N\).
4. Define \(A \in \mathbb{R}^{N \times N}\) by \([A]_{i,j} = \max\{0, z_i - b_j\}\).
5. Note \([A]_{i,j} > 0\) iff \(z_i > b_j\) iff \(i \geq j\), so \(A\) is (lower) triangular.
Universality of Feedforward

**Claim.** Given any \((x_1, y_1) \ldots (x_N, y_N) \in \mathbb{R}^d \times \mathbb{R}\) (assume \(x_i\) distinct), there exists a feedforward network \(f : \mathbb{R}^d \to \mathbb{R}\) with \(2N + d\) parameters such that \(f(x_i) = y_i\) for all \(i = 1 \ldots N\).

**Proof.** (Zhang et al., 2016)

1. Find \(a \in \mathbb{R}^N\) so that \(z_i = a^\top x_i\) are distinct.
2. WLOG assume \(z_1 < z_2 < \cdots < z_N\).
3. Find \(b \in \mathbb{R}^N\) so that \(b_1 < z_1 < b_2 < z_2 < \cdots < b_N < z_N\).
4. Define \(A \in \mathbb{R}^{N\times N}\) by \([A]_{i,j} = \max\{0, z_i - b_j\}\).
5. Note \([A]_{i,j} > 0\) iff \(z_i > b_j\) iff \(i \geq j\), so \(A\) is (lower) triangular.
6. Define \(f(x) := w^\top \text{ReLU}((a^\top x \ldots a^\top x) + b)\). We can find \(w \in \mathbb{R}^N\) such that \(y_i = f(x_i)\) for all \(i\) since this is equivalent to solving for \(w\) in \((y_1 \ldots y_N) = Aw\) and \(A\) is invertible.

■
Regularization for Deep Learning

- Large neural networks can easily fit *random* labels.
- Same regularization techniques still useful: early stopping based on validation performance, $l_2$ weight penalty
- Additional techniques
  - **Dropout**: Randomly make elements zero.
  - **Label smoothing**: Make one-hot label representation $\{0, 1\}^L$ 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 definitely helps
“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.

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 $H(\text{pop}(y|x), p_\theta(y|x))$

- Cross-entropy loss with label smoothing: $\alpha \in [0, 1]$

$$H((1 - \alpha)\text{pop}(y|x) + \alpha \text{Unif}([1 \ldots L]), p_\theta(y|x))$$

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

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

- Shown useful for machine translation and other tasks

  - See: “When Does Label Smoothing Help?” (Müller et al., 2019)
Layer Normalization

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

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

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

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

- 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, $h' = \text{LayerNorm}(h)$ 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)
Deep learning is a flexible paradigm.

\[ \text{enc}_\theta(x) = \text{ReLU}(U^\top x + a) \quad U \in \mathbb{R}^{d \times H}, \ a \in \mathbb{R}^H \]

\[ \text{enc}_\theta(x) = \tanh(U^\top \tanh(U^\top \text{ReLU}(U^\top x))) \quad U \in \mathbb{R}^{d \times d} \]

\[ \text{enc}_\theta(x) = \text{LayerNorm}(\text{ReLU}(V^\top \sigma(U^\top x))) \quad U \in \mathbb{R}^{d \times H}, \ V \in \mathbb{R}^{H \times H'} \]

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

▶ Bottleneck: Have to derive gradients for every new loss/model
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
A directed acyclic graph (DAG) is a directed graph $G = (V, A)$ with a topological ordering.

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

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

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

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

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

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 $f^i : \mathbb{R}^{|\text{pa}(i)|} \to \mathbb{R}$ and compute
     \[
     x^i = f^i((x^j)_{j \in \text{pa}(i)})
     \]
- Thus $G$ represents a function $\{a^i\}_{i \in V_I} \mapsto x^\omega$

- **Forward pass**
  1. Pick some topological ordering $\pi \in \Pi_G$
  2. For $i$ in order of $\pi$, if $i \in V_N$ is a non-input node, set
     \[
     x^i \leftarrow a^i := f^i((a^j)_{j \in \text{pa}(i)})
     \]
- 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.
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 \( x^\omega \) 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 \( x^i_I = (x^j)_{j \in \text{pa}(i)} \), their values 
\( a^i_I = (a^j)_{j \in \text{pa}(i)} \)

▶ Chain rule on the DAG structure

\[
\begin{align*}
  z^i & := \left. \frac{\partial x^\omega}{\partial x^i} \right|_{x_I = a_I} = \sum_{j \in \text{ch}(i)} \left. \frac{\partial x^\omega}{\partial x^j} \right|_{x_I = a_I} \times \left. \frac{\partial x^j}{\partial x^i} \right|_{x_I = a_I} \\
\end{align*}
\]
Key Ideas of Backpropagation

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

\[
  z^i := \frac{\partial x^\omega}{\partial x^i} \bigg|_{x_I = a_I} = \sum_{j \in \text{ch}(i)} \frac{\partial x^\omega}{\partial x^j} \bigg|_{x_I = a_I} \times \frac{\partial x^j}{\partial x^i} \bigg|_{x^j_I = a^j_I}
\]

\[
  = \sum_{j \in \text{ch}(i)} z^j \times \left( \frac{\partial f^j(x^j_I)}{\partial x^i} \bigg|_{x^j_I = a^j_I} \right)
\]

Jacobian of $f^j$ wrt $x^i$
Key Ideas of Backpropagation

- Notation: Parental slots \( x_I^i = (x_j^j)_{j \in \text{pa}(i)} \), their values \( a_I^i = (a_j^j)_{j \in \text{pa}(i)} \)

- Chain rule on the DAG structure

\[
\begin{align*}
    z^i &:= \left. \frac{\partial x^\omega}{\partial x^i} \right|_{x_I = a_I} = \sum_{j \in \text{ch}(i)} \left. \frac{\partial x^\omega}{\partial x^j} \right|_{x_I = a_I} \times \left. \frac{\partial x^j}{\partial x^i} \right|_{x_I = a_I} \\
    &= \sum_{j \in \text{ch}(i)} z^j \times \left. \frac{\partial f^j(x_I^j)}{\partial x^i} \right|_{x_I^j = a_I} \\
    &\text{Jacobian of } f^j \text{ wrt } x^i
\end{align*}
\]

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

\[
\frac{\partial f(x, y)}{\partial x} \bigg|_{(x, y) = (1, 2)} = 16
\]
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

\[ x^i \in \mathbb{R}^{d^i} \quad \forall i \in V \]

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

▶ Gradients: vectors of the same size!

\[ z^i \in \mathbb{R}^{d^i} \quad \forall i \in V \]

▶ Backpropagation: same form using the generalized chain rule

\[
\begin{align*}
  z^i &= \sum_{j \in \text{ch}(i)} \left( \frac{\partial x^\omega}{\partial x^j} \bigg|_{x^I = a^I} \right)_{1 \times d^j} \times \left( \frac{\partial x^j}{\partial x^i} \bigg|_{x^j^I = a^j_I} \right)_{d^j \times d^i} \\
  &= \sum_{j \in \text{ch}(i)} z^j \times \left( \frac{\partial f^j(x^j_I)}{\partial x^i} \bigg|_{x^j^I = a^j_I} \right)
\end{align*}
\]

Jacobian of \( f^j \) wrt. \( x^i \)
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: $f(u) = \text{softmax}(u)$
- Cross-entropy loss:
  
  $$
  f([l_1 \ldots l_N], (y_1 \ldots y_N)) = -(1/N) \sum_i \log \text{softmax}_{y_i}(l_i)
  $$
- Dropout with probability $p$: $f(u) = \text{Drop}_p(u)$

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

\[
\text{Drop}_{0.3}((u_1, u_2, u_3)) = \left( \frac{u_1}{0.7}, 0, \frac{u_3}{0.7} \right)
\]

\[
\frac{\partial \text{Drop}_{0.3}((u_1, u_2, u_3))}{\partial (u_1, u_2, u_3)} = \begin{bmatrix}
\frac{1}{0.7} & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & \frac{1}{0.7}
\end{bmatrix}
\]

\[
(z_1, z_2, z_3) \frac{\partial \text{Drop}_{0.3}((u_1, u_2, u_3))}{\partial (u_1, u_2, u_3)} = \text{Drop}_{0.3}((z_1, z_2, z_3))
\]
Initialization Strategies

- Non-convex objective; initialization is important
- All zeros? Bad idea: all units learn the same thing!
- Random: small values (e.g., $\mathcal{N}(0, 0.01), \text{Unif}(-0.01, 0.01))$
  - 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*:

  ![Graph showing learning rate behavior](image)

  - 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

- Doesn’t change objective (only for updating weights). “Never hurts”, set $C$ to be very 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 $p_t = (p_t(y = 1), \ldots, p_t(y = L))$ then use $\frac{1}{T} \sum_t p_t$

![Graph showing test error over iterations for different ensembles](image)
Need for Specialized Neural Architectures

- Feedforward implicitly assumes the input is a single vector.
- NLP: Input is a sequence!
- 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: building blocks of deep learning