## 10 Enhancing Optimization

### Gradient Clipping

$\frac{\partial E}{\partial \theta} \leftarrow \frac{\tau}{\left|\frac{\partial E}{\partial \theta}\right|}\frac{\partial E}{\partial \theta}$ if $\left|\frac{\partial E}{\partial \theta}\right| > \tau$

## 11 Visual Systems and CNN

- if a layer has $k$ filters, each kernel has a bias

### Multi Channel Inputs
learnable parameter for one channel kernel size

$n \times n \times \text{channel number} + 1\text{bias}$

- CNN avoid parameter explosion
- locality: nearby pixels matter more
- translational invariance: patterns matter anywhere

## 12 Hopfield Networks

### Hopfield Networks

- neuron update rule:
  $x_i = \begin{cases} -1 & \text{if } \vec{x}W + b_i < 0 \\ 1 & \text{if } \vec{x}W + b_i \geq 0 \end{cases}$
- graph has cycles, so backprop won't work

### Hopfield Energy (symmetric $W$)

$E = -\frac{1}{2}\sum_i \sum_{j \neq i} x_i W_{ij} x_j - \sum_i b_i x_i = -\frac{1}{2}\vec{x}W\vec{x}^\top - \vec{b}\vec{x}^\top$

- where $W_{ii} = 0$

### Minimizing Energy (Gradient Descent)

$\frac{\partial E}{\partial x_j} = -\sum_{i \neq j} x_i W_{ij} - b_j$

- or: $\nabla_{\vec{x}} E = -\vec{x}W - \vec{b} \Rightarrow \tau_x \frac{d\vec{x}}{dt} = \vec{x}W + \vec{b}$

### Weight Gradients

- if $i \neq j$: $\frac{\partial E}{\partial W_{ij}} = -x_i x_j$
- if $i = j$: $\frac{\partial E}{\partial W_{ii}} = -x_i^2 = -1$
- gradient vector: $\nabla_W E = -\vec{x}^\top \vec{x} + I_{N \times N}$
- add identity matrix to keep $W_{ii} = 0$ during gradient descent

### Learning Rule (Over All $M$ Targets)

$\nabla_W E = -\frac{1}{M}\sum_{s=1}^{M} (\vec{x}^{(s)})^\top \vec{x}^{(s)} + I = -\frac{1}{M}X^\top X + I$

- weight update: $W \leftarrow W + \kappa\left(\frac{1}{M}X^\top X - I\right)$
- $X^\top X$ computes coactivation states between all neuron pairs
- since input patterns $X$ are fixed, gradient direction is constant across iterations
- steady-state solution: $W^* = \frac{1}{M}X^\top X - I$

## Probability

### Probability Rules

- Bayes' rule: $P(A|B) = \frac{P(B|A)P(A)}{P(B)}$
- Jensen's inequality: for convex $f$: $f(\mathbb{E}[X]) \leq \mathbb{E}[f(X)]$, for concave $f$: $f(\mathbb{E}[X]) \geq \mathbb{E}[f(X)]$

## 13 RBM

### Restricted Boltzmann Machines (RBMs)

- connections between layers are symmetric, weight matrix $W$

### RBM Energy

$E(v, h) = -\sum_{i=1}^{m}\sum_{j=1}^{n} v_i W_{ij} h_j - \sum_{i=1}^{m} b_i v_i - \sum_{j=1}^{n} c_j h_j$

- matrix form: $E(v, h) = -vWh^\top - bv^\top - ch^\top$ where $W \in \mathbb{R}^{m \times n}$
- discordance cost: $-vWh^\top$
- operating cost: $-bv^\top - ch^\top$

### Boltzmann Probability

$q(v, h) = \frac{1}{Z}e^{-E(v,h)}$ where $Z = \sum_{v,h} e^{-E(v,h)}$

- lower-energy states visited more frequently:
- $E(v^{(1)}, h^{(1)}) < E(v^{(2)}, h^{(2)}) \Rightarrow q(v^{(1)}, h^{(1)}) > q(v^{(2)}, h^{(2)})$

### Training RBM as Generative Model

For inputs $v \sim p(v)$, want RBM $q_\theta$ such that:

$\max_\theta \mathbb{E}_{v \sim p}[\ln q_\theta(v)]$ or equivalently

$\min_\theta \mathbb{E}_{v \sim p}[-\ln q_\theta(v)]$

- loss function: $L = -\ln q_\theta(V)$ for given $V$
- expanding: $L = -\ln\left(\frac{1}{Z}\sum_h e^{-E_\theta(V,h)}\right)$
- rewriting: $L = -\ln\left(\sum_h e^{-E_\theta(V,h)}\right) + \ln\left(\sum_v \sum_h e^{-E_\theta(v,h)}\right)$

### Combined Gradient

$\nabla_\theta L = \nabla_\theta L_1 + \nabla_\theta L_2 = \mathbb{E}_{q(h|V)}[\nabla_\theta E_\theta] - \mathbb{E}_{q(v,h)}[\nabla_\theta E_\theta]$

### Computing Gradient for $W_{ij}$

For parameter $\theta = W_{ij}$:

$\nabla_{W_{ij}} E(V, h) = \nabla_{W_{ij}}[-\sum_{i=1}^{m}\sum_{j=1}^{n} V_i W_{ij} h_j - \sum_{i=1}^{m} b_i V_i - \sum_{j=1}^{n} c_j h_j] = -V_i h_j$

- similarly: $\nabla_{W_{ij}} E(v, h) = -v_i h_j$
- gradient of loss: $\nabla_{W_{ij}} L = -\mathbb{E}_{q(h|V)}[V_i h_j] + \mathbb{E}_{q(v,h)}[v_i h_j]$
- first term: expected value under posterior distribution
- second term: expected value under joint distribution

### Contrastive Divergence for Training RBMs
makes differentiation possible

**Step 1:** Clamp visible states to $V$, calculate hidden probabilities:

$q(h_j|V) = \sigma(VW_{\cdot j} + c_j)$

- then: $\nabla_W L_1 = -V^\top \sigma(VW + c)$
- results in rank-1 outer product in $\mathbb{R}^{m \times n}$

**Step 2:** Compute expectation using Gibbs Sampling:

$\langle v_i h_j \rangle_{q(v,h)} \equiv \mathbb{E}_{q(v,h)}[v_i h_j] = \sum_v \sum_h q(v, h) v_i h_j$

- Gibbs sampling computes average $v_i h_j$: $\nabla_W L_2 = v^\top \sigma(vW + c)$
- also an outer product

### Weight Update Rule

$W \rightarrow W - \eta(\nabla_W L_1 + \nabla_W L_2)$

$W \rightarrow W + \eta V^\top \sigma(VW + c) - \eta v^\top \sigma(vW + c)$

- $\eta$: learning rate
- first term: **positive phase** (clamped visible state)
- second term: **negative phase** (after one Gibbs sampling step)

### Sampling an RBM

After training, generate new data points $V^{(1)}, V^{(2)}, \dots, V^{(M)}$ via Gibbs sampling from $P(h|v), P(v|h)$:

```
initialize v = V^(0)
(random data point)
for t = 1 to M:
    sample h^(t) ~ P(h|v^(t-1))
    = sigmoid(v^(t-1) W + c)
    sample v^(t) ~ P(v|h^(t))
    = sigmoid(W h^(t) + b)
return v^(1), v^(2), ...,
v^(M)
```

## 14 Autoenc & Vector Embed.

### Loss Function

$L(x', x)$

- minimizes reconstruction error between output $x'$ and original input $x$
- i.e., how well can we rebuild the input after compressing it?

input and output layers have same size and state

### Tied Weights
decoder uses transpose of encoder weights (reduces parameters, adds regularization)

### Vector Embeddings / Word Representations

- problem: one-hot vectors don't capture similarity ("happy" and "elated" are equally distant from "cat")

### Predicting Word Co-occurrences (Neural Network Approach)

- use 3-layer neural network to predict co-occurrences
- input: one-hot word vector
- output: probability of each word's co-occurrence

$y = f(v, \theta)$ where $v \in \mathcal{W}$

$y \in \mathcal{P}^{N_v} = \{p \in \mathbb{R}^{N_v} \mid p \text{ is a probability vector}\}$

- i.e., $\sum p_i = 1, \quad p_i \geq 0 \quad \forall i$
- $y_j$ = probability that word$_j$ appears nearby

### Neural Network Architecture

- output layer uses **softmax**
- hidden layer is smaller than input/output (bottleneck)
- this squeezing forces similar words to have similar representations
- the hidden layer activations *are* the word embeddings

### word2vec

- (1) **treats common phrases as new words**: e.g., "New York" → one token
- (2) **randomly ignores very common words**: e.g., "the" dominates word pairs
- (3) **negative sampling**: only backprop on some negative cases (not all 70k words)

### Embedding Space

- low-dimensional space where similar inputs map to similar locations
- **why it works**: similar words co-occur with same set of words → similar outputs → similar hidden activations

## 15 Variational Autoencoders

### Variational Autoencoders

- goal: not just reconstruct samples, but generate ANY valid sample
- want to sample from $p(x)$, the distribution of inputs
- idea: sample from lower-dimensional latent space $z \sim p(z)$, then generate $x$ from $z$
- e.g., for digits, $z$ could represent digit class, thickness, slant, etc.

### Generative Model Formulation

$p(x) = \int p_\theta(x|z)p(z)\,dz$

- have dataset $X$, want to find $\theta$ to maximize likelihood of observing $X$
- $p(x|z)$: mapping from latent $z$ to data $x$ (decoder)

### Gaussian Decoder Assumption

Assume $p_\theta(x|z)$ is Gaussian with mean $d(z, \theta)$ and std $\Sigma$:

$-\ln p_\theta(x|z) = \frac{1}{2\Sigma^2}\|X - d(z, \theta)\|^2 + C$

- given samples $z$, we can learn decoder $d(z, \theta)$
- objective: $\max_\theta \mathbb{E}_{z \sim p(z)}[p_\theta(x|z)]$ or $\min_\theta \mathbb{E}_{z \sim p(z)}[\|X - d(z, \theta)\|^2]$
- can use Monte Carlo to approximate: $\mathbb{E}_{p(z)}[p_\theta(x|z)] = \int p_\theta(x|z)p(z)dz$

### Sampling from Latent Space

- if we sample randomly, we choose improbable $z$'s where $p(z_i) \approx 0$

### Choose the Latent Distribution

Let $q(z)$ be our chosen distribution over $z$:

$p(x) = \mathbb{E}_{z \sim p}[p(x|z)] = \int dz\, p(x|z)p(z) = \int dz\, p(x|z)\frac{p(z)}{q(z)}q(z) = \mathbb{E}_{z \sim q}\left[p(x|z)\frac{p(z)}{q(z)}\right]$

### Evidence Lower Bound (ELBO)

Expected negative log likelihood (NLL):

$-\ln p(x) \leq -\mathbb{E}_{q(z)}\left[\ln p(x|z) + \ln\frac{p(z)}{q(z)}\right]$

- rewrite RHS: $-\ln p(x) \leq \mathrm{KL}(q(z)\|p(z)) - \mathbb{E}_{q(z)}[\ln p(x|z)]$
- where KL divergence: $\mathrm{KL}(q(z)\|p(z)) = -\mathbb{E}_{z \sim q}\left[\ln\left(\frac{p(z)}{q(z)}\right)\right]$
- (1) + (2) is upper bound on NLL; minimizing it maximizes likelihood $p(x)$

### VAE Strategy

- (1) choose convenient latent distribution: $p(z) \sim \mathcal{N}(0, I)$
- design $q(z)$ to be close to $\mathcal{N}(0, I)$: $\min_q \mathrm{KL}(q(z)\|\mathcal{N}(0, I))$
- how? encoder outputs $\mathcal{N}(\mu, \sigma^2)$, then pressure encoder to give $\mu = 0, \sigma^2 = I$

### KL Divergence for Gaussians (Closed Form)

$\mathrm{KL}(\mathcal{N}(\mu, \sigma^2)\|\mathcal{N}(0, I)) = \frac{1}{2}(\sigma^2 + \mu^2 - \ln\sigma^2 - 1)$

- want to minimize this (push encoder toward standard normal)
- but reconstruction loss pushes back...

### Reconstruction Loss (Term 2)

$\mathbb{E}_q[\ln p(x|z)]$ is reconstruction loss

- can write as $\mathbb{E}_q[\ln p(x|\hat{x})]$ where $\hat{x} = d(z, \theta)$ (deterministic decoder)

### Reparameterization Trick

$z = \mu(x, \theta) + \epsilon \odot \sigma(x, \theta), \quad \epsilon \sim \mathcal{N}(0, I)$

- makes distribution differentiable for backprop
- stochasticity comes from separate variable $\epsilon$ (vector of random values)

### VAE Training Process

- encode $x$: compute $\mu(x, \theta)$ and $\sigma(x, \theta)$ using neural network
- sample: $z = \mu + \epsilon\sigma, \quad \epsilon \sim \mathcal{N}(0, I)$
- compute KL loss: $\frac{1}{2}(\sigma^2 + \mu^2 - \ln\sigma^2 - 1)$
- decode: $\hat{x} = f(x, \theta) = d(z)$
- compute reconstruction loss:
- Gaussian $p(x|\hat{x})$: $\frac{1}{2}\|\hat{x} - x\|^2$
- Bernoulli $p(x|\hat{x})$: $\sum_x x \ln\hat{x}$

### Full VAE Objective

$E = \mathbb{E}_x[L(x, \hat{x}) + \beta(\sigma^2 + \mu^2 - \ln\sigma^2 - 1)]$

- first term is reconstruction, second is KL, both terms differentiable w.r.t. $\theta$, so can use gradient descent
- $\beta$ balances reconstruction vs. KL divergence loss
- VAE latent space has fewer holes (closer to $\mathcal{N}(0, I)$)

## 16 Diffusion Models

### Diffusion Models

- goal: generate images from noise by reversing diffusion process
- latent variable model with sequence: $x_0, x_1, \dots, x_T$

### Forward Process (Adding Noise)

- progressively adds noise until we get pure noise at $x_T$
- forward step: $q(x_t \mid x_{t-1})$
- defined as Gaussian: $q(x_t \mid x_{t-1}) = \mathcal{N}(x_t; \sqrt{1-\beta_t}\,x_{t-1}, \beta_t I)$
- for large $T$: $x_T \sim \mathcal{N}(0, I)$
- variance schedule $\beta_1, \dots, \beta_T$ controls noise addition (e.g., linear: $\beta_1 = 10^{-4}, \beta_T = 0.02$)

### Reverse Process (Denoising)

- aims to recover original data from noise
- reverse step: $p_\theta(x_{t-1} \mid x_t)$
- learned by neural network

### Forward Process: From $x_0$ to $x_t$ Directly

$x_t = \sqrt{1-\beta_t}\,x_{t-1} + \sqrt{\beta_t}\,\epsilon_t, \quad \epsilon_t \sim \mathcal{N}(0, I)$

- define $\alpha_t \equiv 1 - \beta_t$, expand recursively:
- $x_t = \sqrt{\alpha_t}(\sqrt{\alpha_{t-1}}\,x_{t-2} + \sqrt{1-\alpha_{t-1}}\,\epsilon_{t-1}) + \sqrt{1-\alpha_t}\,\epsilon_t$
- result: $x_t = \sqrt{\bar{\alpha}_t}\,x_0 + \sum_i c_i\epsilon_i$
- where $\bar{\alpha}_t \equiv \alpha_1 \cdots \alpha_t$ and $\epsilon_i \sim \mathcal{N}(0, I)$
- sum of independent Gaussians is Gaussian: $\sum_i c_i\epsilon_i \sim \mathcal{N}(0, (1-\bar{\alpha}_t)I)$

### Closed-Form Forward Process

$x_t = \sqrt{\bar{\alpha}_t}\,x_0 + \sqrt{1-\bar{\alpha}_t}\,\epsilon, \quad \epsilon \sim \mathcal{N}(0, I)$

- can jump directly from $x_0$ to any $x_t$ without iterating
- solving for $x_0$: $x_0 = \frac{1}{\sqrt{\bar{\alpha}_t}}\left(x_t - \sqrt{1-\bar{\alpha}_t}\,\epsilon\right)$
- to get $x_0$ from $x_t$, need to estimate noise $\epsilon$ using neural network $\epsilon_\theta(x_t, t)$

### Why Not Use Direct Formula?

- $\bar{\alpha}_t = \alpha_1 \cdots \alpha_t$ becomes very small for large $t$
- amplifies noise in our estimate of $\epsilon$
- use iterative sampling algorithm instead (works better in practice)

### Simplified Loss

$\mathbb{E}_{x_0,\epsilon}\left[\lambda_t\|\epsilon - \epsilon_\theta(x_t, t)\|_2^2\right]$

- where $\lambda_t = \frac{\beta_t^2}{2\sigma_t^2\alpha_t(1-\bar{\alpha}_t)}$
- in practice, set $\lambda_t = 1$
- substituting $x_t$: $\mathbb{E}_{x_0,\epsilon}[\|\epsilon - \epsilon_\theta(\sqrt{\bar{\alpha}_t}\,x_0 + \sqrt{1-\bar{\alpha}_t}\,\epsilon, t)\|_2^2]$
- intuition: train network to predict the noise that was added

### Training Algorithm

```
repeat until convergence:
    sample x_0 ~ q(x_0)
    from dataset
    sample t
    ~ Uniform({1, ..., T})
    sample epsilon ~ N(0, I)
    compute x_t
    = sqrt(alpha_bar_t) *
    x_0 + sqrt(1 - alpha_bar_t)
    * epsilon
    gradient descent on L =
    ||epsilon -
    epsilon_theta(x_t, t)||^2
```

### Sampling Algorithm

```
initialize x_T ~ N(0, I)
for t = T, ..., 1:
    sample z ~ N(0, I) if
    t > 1, else z = 0
    compute x_{t-1} =
    (1/sqrt(alpha_t)) *
    (x_t - (beta_t/sqrt(1 -
    alpha_bar_t)) * epsilon_
    theta(x_t, t)) + sigma_t
    * z
return x_0 (generated sample)
```

takes noisy image $x_t$ and timestep $t$ as input outputs predicted noise $\epsilon$

## 17 Recurrent NN

### How it works
processes sequence with hidden state that carries information forward from previous steps.

### Mathematical Definition

$\mathbf{h}_i = f(\mathbf{x}_i U + \mathbf{h}_{i-1} W + \mathbf{b})$

### Output $\mathbf{y}$ at time $i$

$\mathbf{y} = \mathrm{Softmax}(\mathbf{h}_i V + \mathbf{c})$

- $\mathbf{y}$ is output at $i$
- $V$ is weight matrix from hidden-to-output transformation
- $\mathbf{c}$ is bias
- maintain memory through hidden state $\mathbf{h}_i$, effective for sequences: language modeling, time-series forecasting, speech recognition
- increasing number of hidden units enhances model's ability to store and process long-term dependencies, higher computational cost & memory usage
- vanilla RNN is prone to vanishing & exploding gradients when the sequence is long, use long short-term memory & gated recurrent units

### Train RNN - Loss Function

need output to match target by minimizing loss function

$\mathcal{L}(\mathbf{y}_1, \dots, \mathbf{y}_N, \mathbf{t}_1, \dots, \mathbf{t}_N) = \sum_{i=1}^{N} \alpha_i \mathcal{L}(\mathbf{y}_i, \mathbf{t}_i)$

- $\mathbf{y}_i$ output at $i$

- $\mathbf{t}_i$ target output at $i$
- $N$ sequence length
- $\mathcal{L}(\mathbf{y}_i, \mathbf{t}_i)$ loss function error between prediction and target, could be cross-entropy for classification or MSE for regression

Train RNN - Find $\theta$
$\theta$ minimizes the expected loss over the entire data set, mathematically
$\theta^* = \arg\min_\theta \mathbb{E}_{(\mathbf{X},\mathbf{T})\in\mathcal{D}}[\mathcal{L}]$
$\theta = \{U, V, W, \vec{b}, \vec{c}\}$ is the set of trainable parameters

Deep RNN Mechanism
input $\vec{x}^n$ is processed by the first RNN layer, generating a hidden state $\vec{h}_1^n$, each subsequent layer $l$ receives hidden state from previous $l-1$ and computes a new hidden, the final layer $L$ produces output $\vec{y}^n$

Deep RNN Layer Math Def
$\vec{h}_L^n = f(\vec{h}_{L-1}^n U_L + \vec{h}_L^{n-1} W_L + b_L)$
- $\vec{h}_i^n$ hidden state at time step $n$ in layer $l$
- $U_l$ is the input-to-hidden weight matrix for layer $l$
- $W_l$ is the recurrent weight matrix within layer $l$
- $f$ is non-linear activation function
- $\vec{b}_i$ are vector biases

the final output is
$\vec{y}_n = \text{Softmax}(\vec{h}_L^n V + \vec{c})$
- $V$ is the weight matrix from the last hidden layer to the output
- $\vec{c}$ is a bias vector
- increase the number of layers while keeping size of hidden state $d_n$ small to maintain reasonable computational cost
- improves representation learning
- deep RNN outperforms shallow ones in speech recognition and language modeling

## 18 Gated Recurrent Units

Why GRU vanilla RNNs this happens due to vanishing gradient when the weight $|w| < 1$, the $n$-th power shrinks exponentially, if $|w| > 1$, the training becomes unstable

New Candidate Hidden State
$\tilde{\vec{h}}^n = \tanh(\vec{h}^{n-1} W + \vec{x}^n U + \vec{b})$
- $W$ is the hidden-to-hidden weight matrix
- $U$ is the input-to-hidden weight matrix
- $\vec{b}$ is the bias vector
- The tanh function ensures that $\tilde{h}^t \in (-1, 1)$

Gate Mechanism
The gate $\vec{g}^n$ determines how much past information is retained
$\vec{g}^n = \sigma(\vec{h}^{n-1} W_g + \vec{x}^n U_g + \vec{b}_g)$
- $W_g$ and $U_g$ are the gate's weight matrices
- $\vec{b}_g$ is the bias vector for the gate
- The $\sigma$ (sigmoid) function ensures that $g^n \in (0, 1)$, meaning it acts as a soft switch

Final Hidden State Update
$\vec{h}^n = \vec{g}^n \odot \tilde{\vec{h}}^n + (1 - \vec{g}^n) \odot \vec{h}^{n-1}$
- $\vec{g}^n$ controls how much of the new candidate state $\tilde{\vec{h}}^n$ is retained
- $(1 - \vec{g}^n)$ controls how much of the previous state $\vec{h}^{n-1}$ is preserved

How It Works
- if $g_i^n \approx 1$, the new state is mostly the candidate state component $\tilde{h}_i^n$, meaning the network updates to new information
- if $g_i^n \approx 0$, the previous hidden state component $h_i^{n-1}$ is mostly preserved, preventing unnecessary updates

for "the city is beautiful", we can set $g$ for city and beautiful to be 1, it retains relevant information

Full GRU
update gate
$\vec{g}^n = \sigma(\vec{h}^{n-1} W_g + \vec{x}^n U_g + \vec{b}_g)$
reset gate
$\vec{r}^n = \sigma(\vec{h}^{n-1} W_r + \vec{x}^n U_r + \vec{b}_r)$
candidate state
$\tilde{\vec{h}}^n = \tanh((\vec{h}^{n-1} \odot \vec{r}^n) W + \vec{x}^n U + \vec{b})$
final state
$\vec{h}^n = \vec{g}^n \odot \tilde{\vec{h}}^n + (1 - \vec{g}^n) \odot \vec{h}^{n-1}$
- reset gate $\vec{r}^n$ determines how much of the previous hidden state $\vec{h}^{n-1}$ should be forgotten before new candidate hidden state
- when $\vec{r}^n$ is close to 0, the model is more reliant on new input
- when when $\vec{r}^n$ is close to 1, more past information is retained

## 19 Attention Mechanism

transformers is good for natural language processing
Tokenization breaks down a piece of text (like a sentence) into smaller units called tokens
Embedding each word is a vector of size $d$ and represents a row, the embedding $X$ of a sentence with three words is $\in \mathbb{R}^{3\times d}$
Self Attention for each word $x_i$
- Queries $\vec{q}_i = \vec{x}_i W^{(Q)}$
  $Q = XW^{(Q)} \quad n \times d \cdot d \times \ell$
  what the word is looking for
- Keys $\vec{k}_i = \vec{x}_i W^{(K)}$
  $K = XW^{(K)} \quad n \times d \cdot d \times \ell$
  what the word has, used to decide if the word is relevant
- Values $\vec{v}_i = \vec{x}_i W^{(V)}$
  $V = XW^{(V)} \quad n \times d \cdot d \times \ell$
  provides information once the word is chosen as relevant

for the matrices
- all matrices are in $\mathbb{R}^{d\times\ell}$
- $\ell$ is a hyperparameter
- $k$ and $v$ belong to the words that might be attended to
- $q$ belongs to the word that is doing the attending

Computing Attention Scores
the attention of $\vec{q}_i$ on $\vec{k}_j$ is
$S_{ij} = \vec{q}_i \cdot \vec{k}_j, \quad j = 1, \ldots, n$
$S_{ij}$ is the vector $i$'s score for vector $j$'s, how important $k_j$ is to $q_i$, so $S_{12}$ is how important 3 is to 1

Full Attention Matrix
$S = QK^T \quad n \times \ell \cdot \ell \times n$
Self-Attention Output each row sums up to 1
$A = \text{Softmax}(S/\sqrt{d}),\ A \in \mathbb{R}^{n\times n}$
Attention Head
$H = A \cdot V \quad H \in \mathbb{R}^{n\times\ell}$
- original word embedding w contextual information important words, importance still matrix $A$
- it has the same size as the original embedding matrix $X$

for each token output
$\vec{H}_i = \sum_{j=1}^n A_{ij} \vec{v}_j$
- $A_{ij}$ attention score of query $\vec{q}_i$ on key $\vec{k}_j$
- $\vec{v}_j$ is the value associated with input $j$

Positional Encoding Problem
sentences with same words but different order have the same attention head, word embeddings do not contain positional information, self-attention is permutation equivalent
Positional Encoding
impose new order
$\vec{x}_i \Rightarrow \vec{x}_i' = \vec{x}_i + \text{PE}(i)$
PE is defined as
$\text{PE}(i)_{2j} = \sin\left(\frac{i}{10000^{2j/d}}\right)$
$\text{PE}(i)_{2j+1} = \cos\left(\frac{i}{10000^{2j/d}}\right)$
- $\text{PE}(i)$ is the positional encoding vector for position $i$
- frequency changes with dimension $j$
- 10000 is a scaling constant to allow converge of a large max sequence length

Multi-Head Attention
allows the model to jointly attend to information, each head can learn distinct aspects or features, it runs $n$ weight matrices in parallel
$\text{Multi-Head Attention} = \text{Concat}(\text{H}^{(1)}, \ldots, \text{H}^{(\text{h})})\, W^O \in \mathbb{R}^{n\times d_{\text{model}}}$
- each $H^{(\mu)}$ is computed independently in each attention head $\mu$
- concat is $\in \mathbb{R}^{n\times(h\ell)}$
- $W^O \in \mathbb{R}^{(h\ell)\times d_{\text{model}}}$
- the number of heads is denoted by $h$

## 20 Transformers

Batch Normalization (One Feature for All Samp.)
Batch N on a Single Neuron
let $i$ a particular neuron (or feature) indexed by $i$ and a mini-batch of size $D$, let $h_i^{(d)}$ be the activations of neuron $i$ on example $d$, for $d = 1, \ldots, D$
- the mini-batch mean of neuron $i$ is $\mu_i = \frac{1}{D}\sum_{d=1}^D h_i^{(d)}$
- the corresponding mini-match variance is $\sigma_i^2 = \frac{1}{D}\sum_{d=1}^D (h_i^{(d)} - \mu_i)^2$
- normalize each activation $\hat{h}_i^{(d)} = \frac{h_i^{(d)} - \mu_i}{\sigma_i}$ or more stably $\hat{h}_i^{(d)} = \frac{h_i^{(d)} - \mu_i}{\sqrt{\sigma_i^2 + \varepsilon}}$

Layer Normalization
$y = \text{LN}(x)$ normalizes all features within one layer for a single example
for hidden vector $\vec{h} \in \mathbb{R}^H$, compute the mean and variance of its coordinates then normalized activation is
$\hat{\vec{h}} = \frac{\vec{h} - \mu}{\sqrt{\sigma^2 + \varepsilon}}$
Layer Norm Definition
$\text{LN}(\vec{h}) = \boldsymbol{\alpha} \odot \hat{\vec{h}} + \boldsymbol{\beta}$ where $\boldsymbol{\alpha}, \boldsymbol{\beta} \in \mathbb{R}^H$ are learned scale and shift parameters
Why use LN
let $h_1 = 5, h_2 = 5000$, to train $z = w_1 h_1 + w_2 h_2 + b$, we need to keep $w_2$ very small, inefficient learning. LN scales $h_1$ and $h_2$ so they're comparable
Add and Norm Module after a multi-head attentionblock in a transformer, we use
$y = \text{LN}(x + \text{MHA}(x))$
BN vs. LN
- BN normalizes over the batch dimension, LN normalizes over the feature dimension, BN solves it across examples, LN solves it within each example
- often prefer LN over BN because batch size is often variable, LN does not depend on batch statistics

FF Layer in En/Decoder
$x \to$ attention $\to$ norm $\to$ FF layer $\to$ norm $\to$ output
inside each encoder and decoder there is a position-wise FFN that applies weights and biases independently to each position of the sequence
$f(x) = W_2 \varphi(W_1 x + b_1) + b_2$
Autoregressive
the decoder is autoregressive, each prediction depends only on previous predictions, not future ones
Next-Token NLL
loss that's minimized
$\mathcal{L} = \mathbb{E}_D[\sum_{i=1}^N - \log p_\theta(x_i | x_{<i})]$
Benefit of Transformer
- long range dependency
- every position can directly attend to every other position in a single step, $O(1)$ path length
- direct gradient flow

## 21 Adversarial Attacks

given the data set $\mathcal{D} = \{(x, t) \mid x \in X,\ t \in \{1, \ldots, K\}\}$
- $X$ is output space
- $t$ is true target

Real World Issue
- might cause autonomous vehicles to not recognize stop signs
- misclassifications of numbers that might be undetected by human eyes

Classification Error
$R(f) \triangleq \mathbb{E}_{(x,t)\sim\mathcal{D}}[\text{card}\{\arg\max_i y_i \neq t \mid y = f(x)\}]$
- $\arg\max_i y_i$ the index of the largest element of $y$
- card is the cardinality (number of elements)

$\varepsilon$-Ball
$\mathcal{B}(x, \varepsilon) = \{x' \in X \mid \|x - x'\| \leq \varepsilon\}$
Adversarial Attacks
is there $x' \in \mathcal{B}(x, \varepsilon)$ such that $\arg\max_i(y_i) \neq t$ for $y = f(x')$, is there $x'$ such that it's output probability vector is classified incorrectly?
Gradient-Based Whitebox Attack
going up the gradient to maximize loss
Untargeted Attack
$x' = x + k\nabla_x E(f(x; \theta, t(x)))$ or $\max_{x'\in\mathcal{B}(x,\varepsilon)}[L(f(x'), t]$ update input based on gradient wrt input, only need the model to classify incorrectly, gradient ascent to increase loss
Targeted Attack
$x' = x - k\nabla_x E(f(x; \theta), l)$ where $l \neq t(x)$ or
$\max_{x'\in\mathcal{B}_{(x,\varepsilon)}}[Lf(x'), l],\ l \neq t$
gradient descent in the direction to decrease loss for the wrong class
Fast Gradient Sign Method
adjust by each pixel by $\varepsilon$
$\Delta x = \varepsilon\, \text{sign}(\nabla_x E)$
can also find the smallest $\|\Delta x\|$ that causes misclassification
$\min_{\|\Delta x\|}[\arg\max_i(y_i(x)) \neq t(x)]$
Why Fooled So Easily?
output dimension is flattened to a high dimension, move in one direction by a small step causes movement in all directions
$w^T \Delta x = \sum_i^n w_i x_i = \sum_i^n w_i\, \text{sign}(\Delta_x E)\varepsilon$, when $n$ is large, small $\varepsilon$ contributes to a huge loss

## 22 Adversarial Defense

suppose we have a model $f : X \to \mathbb{R}$ and the dataset $(X, T)$, where $X \subset \mathbb{X}$, and $T \in -1, 1$, we use
- $\text{sign}(f(X))$ indicates the class of $x$
- classification is correct if $f(X)T > 0$

Classification "Natural" Loss
$\mathcal{R}_{\text{nat}}(f) = \mathbb{E}_{(X,T)}[\text{card}\{f(X)T \leq 0\}]$
counts how many points are misclassified, this loss doesn't care about adversarial robustness at all, a point could be barely correctly classified (margin close to 0), and this loss counts it the same as a point that's very confidently correct
Robust Loss
$\mathcal{R}_{\text{rob}}(f) = \mathbb{E}_{(X,T)}[\text{card}\{X' \in \mathcal{B}(X, \varepsilon) \mid f(X')T \leq 0\}]$
- ball represents all possible adversarial perturbations an attacker could make
- it says if any point in the $\varepsilon$-region can be misclassified then it's bad

then we use a smooth function g that approximates the step function since card is a step function
Full TRADES
$\min_f \mathbb{E}_{(X,T)}[g(f(X)T) + \max_{X'\in\mathcal{B}(X,\varepsilon)} g(f(X) \cdot f(X'))]$
- first term ensures $X$ is properly classified
- second term adds a penalty for models $f$ that place the decision boundary within $\varepsilon$ of $X$, where$f(X)$ and $f(X')$ will have opposite signs

Implementation
for each gradient update
- run several steps of gradient ascent to find $X'$
- evaluate joint loss $\text{loss} = g(f(X)T) + \beta g(f(X)f(X'))$ where $\beta$ s a hyperparameter
- use gradient of the loss to update weights

## 23 Generative Adversarial Network

Cost Function of GANs
$C(\theta_D, \theta_G) = -\frac{1}{2}\mathbb{E}_{x_{\text{real}}\sim p_{\text{data}}}[\log D_{\theta_D}(x_{\text{real}})] - \frac{1}{2}\mathbb{E}_{z\sim p_z}[\log(1 - D_{\theta_D}(G_{\theta_G}(z)))]$
- $\theta_D$ parameters (weights) of the discriminator network
- $\theta_G$ parameters (weights) of the generator network
- $p_{\text{data}}$ the distribution of real data
- $p_z$: the distribution of noise (usually standard Gaussian)
- $D_{\theta_D}(x)$ discriminator's output (probability) for input $x$
- $G_{\theta_G}(z)$ generator's output (fake sample) for noise $z$

GAN Training (Min-Max Game)
$\max_{\theta_G} \min_{\theta_D} C(\theta_D, \theta_G)$
- implemented by alternating gradient *descent* on discriminator and gradient *ascent* on generator
  $\theta_D \leftarrow \theta_D - \eta_D \nabla_{\theta_D} C(\theta_D, \theta_G)$
  $\theta_G \leftarrow \theta_G + \eta_G \nabla_{\theta_G} C(\theta_D, \theta_G)$
- discriminator $D$ aims to minimize cost $C$, ideally $D(x_{\text{real}}) = 1$ and $D(x_{\text{fake}}) = 0$
- $D(x_{\text{real}}) \approx D(x_{\text{fake}}) \approx 0.5$

Relation to Untargeted A A
classifier $f_\theta(x)$ with loss $\ell(f_\theta(x), y)$, untargeted solves
$x_{\text{adv}} = \arg\max_{x'\in B(x,\varepsilon)} \ell(f_\theta(x'), y)$
- gradient ascent on input: $x \leftarrow x + \alpha\nabla_x \ell(f_\theta(x), y)$
- adversarial training (TRADES) min-max objective $\min_\theta \max_{x'\in B(x,\varepsilon)} \ell(f_\theta(x'), y)$
- same structure as GAN $\min_{\theta_D} \max_{\theta_G} C(\theta_D, \theta_G)$
- discriminator $D_{\theta_D}$ is classifier $f_\theta$
- generator $G_{\theta_G}(z)$ is learned adversary searching for $x_{\text{fake}} = G_{\theta_G}(z)$ that maximally confuses discriminator

Intuition and Training Phases
- discriminator $D$ distinguishes fake from real
- generator $G$ improves, discriminator's task hard
- ideally, generator produces data indistinguishable from real, discriminator assigns probability 0.5 to both
- GAN trained until discriminator can no longer reliably distinguish real vs generated