CSC413 Notes

Jenci Wei

Winter 2023

Contents

1 Introduction **3**

2 Linear Models **4**

3 Multilayer Perceptrons **8**

4 Backpropagation **9**

5 Automatic Differentiation **12**

6 Distributed Reprsentations **14**

7 Optimization **18**

8 Generalization **28**

9 Convolutional Neural Networks **32**

10 Interpretability **37**

11 Large-Scale Generative Models **40**

12 Recurrent Neural Networks **42**

13 Attention **46**

14 Transformers **50**

15 CNN For Speech Analysis **54**

16 Large Language Models **56**

17 Generative Adversarial Networks **58**

18 Variational Autoencoders **60**

19 Graph Neural Networks **65**

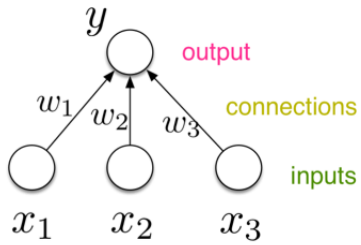
20 Q-Learning **68**

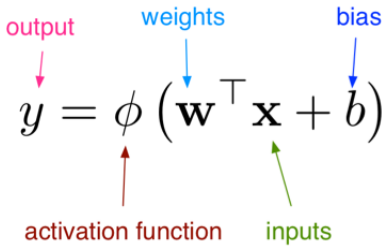
1 Introduction

Machine Learning

- Machine learning approach: program an algorithm to automatically learn from data/experience
- Types
  - **Supervised learning**: have labelled examples of the correct behaviour, i.e. ground truth input/output response
  - **Reinforcement learning**: learning system receives a reward signal, tries to learn to maximize the reward signal
  - **Unsupervised learning**: no labelled examples, instead, looks for patterns in the data

Neural Networks





- NNs are collections of thousands/millions of these simple processing units that together perform useful computations
- Inspiration from the brain
- Very effective across a range of application, and widely used

Deep Learning

- Emphasizes that the algorithms often involve hierarchies with *many stages of processing*
- Higher layers in the network often learn higher-level, more interpretable representations

2 Linear Models

Problem Setup

- Want to predict a scalar t as a function of a vector x
- We have a dataset of pairs $\left\{\left(x^{(i)}, t^{(i)}\right)\right\}_{i=1}^{N}$
- The $x^{(i)}$ are called **input vectors**, and the $t^{(i)}$ are called **targets**
- Model: y is a linear function of x

$$y = w^\top x + b$$

  - y is the **prediction**
  - w is the **weight vector**
  - b is the **bias**
  - w and b together are the **parameters**
  - Setting of the parameters are called **hypotheses**
- **Loss function**: squared error

$$\mathcal{L}(y, t) = \frac{1}{2}(y - t)^2$$

  - $y - t$ is the **residual**, and we want to make this small in magnitude
  - The 1/2 factor makes the calculations convenient
- **Cost function**: loss function averaged over all training examples

$$\mathcal{J}(w, b) = \frac{1}{2N}\sum_{i=1}^{N}\left(y^{(i)} - t^{(i)}\right)^2 = \frac{1}{2N}\sum_{i=1}^{N}\left(w^\top x^{(i)} + b - t^{(i)}\right)^2$$

Vectorization

- Can organize all the training examples into a matrix X with 1 row per training example, and all the targets into a vector t

$$X = \begin{bmatrix} {x^{(1)}}^\top \\ \vdots \\ {x^{(N)}}^\top \end{bmatrix}$$

- Computing the predictions for the whole dataset:

$$Xw + b1 = \begin{bmatrix} w^\top x^{(1)} \\ \vdots \\ w^\top x^{(N)} \end{bmatrix} = \begin{bmatrix} y^{(1)} \\ \vdots \\ y^{(N)} \end{bmatrix} = y$$

```
y = np.dot(X, w) + b
```

- Computing the squared error cost across the whole dataset:

$$\mathcal{J} = \frac{1}{2N}\|y - t\|^2$$

```
cost = np.sum((y - t) ** 2) / (2. * N)
```

Optimization

- Want to minimize the cost function
- The minimum of a smooth function (if exists) occurs at a *critical point*, i.e. point where the partial derivatives are all 0
- Two strategies
  1. **Direct solution**: derive a formula that sets the partial derivatives to 0 (may not work for all cases)
  2. **Iterative methods**: repeatedly apply an update rule which slightly improves the current solution (e.g. gradient descent)

Gradient Descent

- **Gradient descent** is an *iterative algorithm*
- We *initialize* the weights to something reasonable (e.g. all zeros), and repeatedly adjust them in the *direction of steepest descent*
- The gradient descent update decreases the cost function for small enough α

$$w_j \leftarrow w_j - \alpha \frac{\partial \mathcal{J}}{\partial w_j} = w_j - \frac{\alpha}{N} \sum_{i=1}^{N} \left(y^{(i)} - t^{(i)} \right) x_j^{(i)}$$

- α is the **learning rate**, i.e. the larger it is, the faster w changes
- Update rule in vector form:

$$w \leftarrow w - \alpha \nabla \mathcal{J}(w) = w - \frac{\alpha}{N} \sum_{i=1}^{N} \left(y^{(i)} - t^{(i)} \right) x^{(i)}$$

- GD is more efficient than direct solution for regression in high-dimensional spaces

Feature Maps

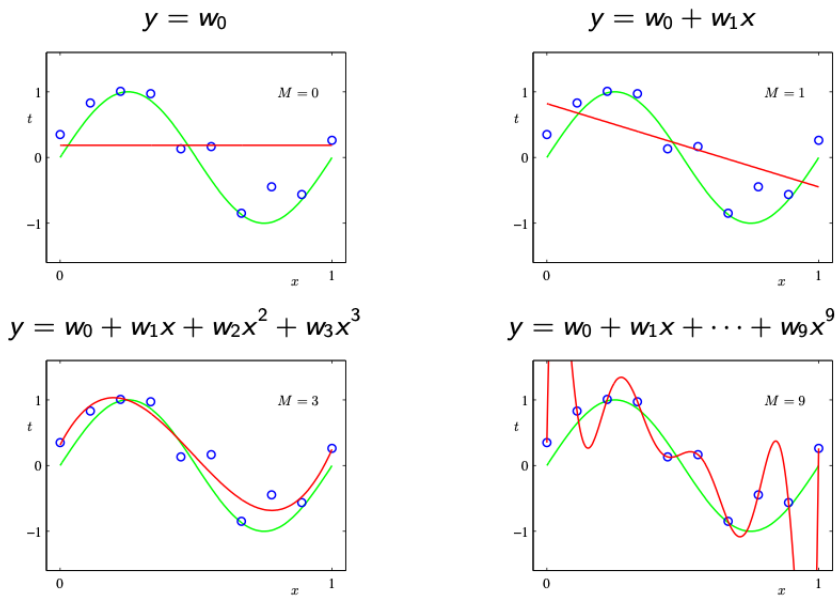
- Can convert linear models into nonlinear models using feature maps

$$y = w^\top \psi(x)$$

- **Polynomial regression**: y is a polynomial in x, i.e. $\psi(x) = \left(1, x, \ldots, x^D\right)^\top$

$$y = w_0 + w_1 x + \cdots + w_D x^D$$

- Does not require changing the algorithm, just use $\psi(x)$ as the input vector
- Does not require a bias term, which could be absorbed into ψ
- Hard to choose good features



Generalization

- **Underfitting**: the model is too simple so that it does not fit the data
- **Overfitting**: the model is too complex so that it fits perfectly but does not generalize
- We want our models to **generalize** to data they haven't seen before
- **Hyperparameter**: something we cannot include in the training procedure (e.g. degree of polynomial in polynomial regression)
- Can tune hyperparameters using a **validation set**

Binary Linear Classification

- **Classification**: predict a discrete-valued target
- **Binary**: predict a binary target $t \in \{0, 1\}$
  - **Positive examples**: training examples with $t = 1$
  - **Negative examples**: training examples with $t = 0$
- **Linear**: model is a linear function of x, thresholded at 0

$$z = w^\top x + b$$

$$\text{output} = \begin{cases} 1, & \text{if } z \geq 0 \\ 0, & \text{if } z < 0 \end{cases}$$

Logistic Regression

- We cannot optimize classification accuracy directly with gradient descent because it is discontinuous
- We can define a **surrogate loss function** that is easier to optimize
- The logistic regression model outputs a continuous value $y \in [0, 1]$ that can be interpreted as the probability of the example being positive

- The **logistic function** is **sigmoidal** (S-shaped):

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

- A linear model with a logistic nonlinearity is known as **log-linear**

$$z = w^\top x + b$$
$$y = \sigma(z)$$

- σ is called an **activation function**, and z is called the **logit**
- Use **cross-entropy loss** to penalize incorrectness based on confidence

$$\mathcal{L}_{\text{CE}} = -t \log y - (1 - t) \log(1 - y) = \begin{cases} -\log y, & \text{if } t = 1 \\ -\log(1 - y), & \text{if } t = 0 \end{cases}$$

- **Logistic regression** combines the logistic activation function with cross-entropy loss

Multiclass Classfication

- Targets form a discrete set $\{1, \ldots, K\}$
- Can be represented as **one-hot vectors**, or **one-of-K encoding**:

$$t = \underbrace{(0, \ldots, 0, 1, 0, \ldots, 0)}_{\text{entry } k \text{ is 1}}$$

- There are D input dimensions and K output dimensions, requiring $K \times D$ weights, which is arranged as a **weight matrix** W
- We also have a K-dimensional vector b of biases
- Linear predictions:

$$z_k = \sum_j w_{kj} x_j + b_k$$

- Vectorized form:

$$z = Wx + b$$

- Activation function: **softmax function** (a multivariable generalization of the logistic function)

$$y_k = \text{softmax}(z_1, \ldots, z_K)_k = \frac{e^{z_k}}{\sum_{k'} e^{z_{k'}}}$$

  - The inputs z_k are called the **logits**
  - Properties
    - Outputs are positive and sum to 1, which can be interpreted as probabilities
    - If one of the z_ks is much larger than the others, softmax(z) is approximately the argmax

- Loss function: cross-entropy

$$\mathcal{L}_{\text{CE}}(y, t) = -\sum_{k=1}^{K} t_k \log y_k = -t^\top \log y$$

  where the log is applied elementwise

- Can combine the softmax and cross-entrop into a **softmax-cross-entropy** function

3 Multilayer Perceptrons

Neural Networks

- Can connect lots of units together in to a *directed acyclic graph*, which gives a *feed-forward neural network*
- Typically units are grouped together into *layers*
- Each layer connects N input units to M output units
- **Fully connected layer**: all input units are connected to all output units
  - In which case we need an $M \times N$ weight matrix
  - The output units are a function of the input units, i.e.

$$\mathbf{y} = f(\mathbf{x}) = \phi(\mathbf{W}\mathbf{x} + \mathbf{b})$$

- **Multilayer perceptron**: a multilayer network consisting of fully connected layers
  - Each layer computes a function, so the network computes a composition of functions, i.e.

$$\begin{aligned}\mathbf{h}^{(1)} &= f^{(1)}(\mathbf{x}) \\ \mathbf{h}^{(2)} &= f^{(2)}(\mathbf{h}^{(1)}) \\ &\vdots \\ \mathbf{y} &= f^{(L)}(\mathbf{h}^{(L-1)})\end{aligned}$$

  - Equivalently

$$\mathbf{y} = f^{(L)} \circ \cdots \circ f^{(1)}(\mathbf{x})$$

Expressive Power

- Any sequence of *linear* layers can be equivalently represented with a single linear layer
  - Deep linear networks are *no more expressive* than linear regression
- Multilayer feed-forward neural net with *nonlinear* activation functions are **universal approximators**
  - They can approximate any function arbitrarily well
- Limits of universality
  - We may need to represent an exponentially large network
  - Learning a function could mean overfitting to it
  - We want a *compact* representation

4 Backpropagation

Gradient Descent

- Gradient descent moves opposite to the gradient (i.e. direction of steepest descent)
- Want to compute the cost gradient $d\mathcal{J}/d\mathbf{w}$, which is a vector of partial derivatives
  - This is the average $d\mathcal{L}/d\mathbf{w}$ over all training examples

Univariate Chain Rule

- If $f(x)$ and $x(t)$ are univariate functions, then

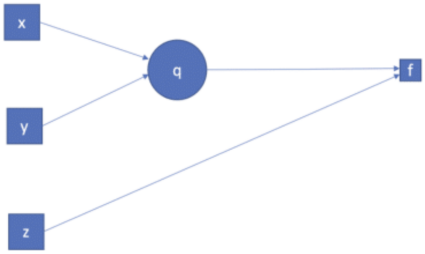
$$\frac{d}{dt}f(x(t)) = \frac{df}{dx}\frac{dx}{dt}$$

Computation Graph

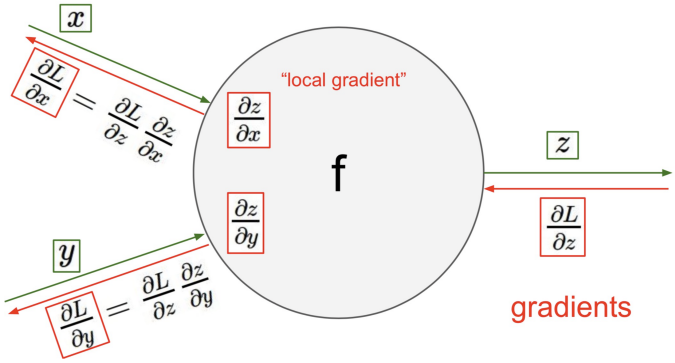
- A directed graph where the *nodes* correspond to *operations* or *variables*
- Variables can feed their value into operations, and operations can feed their output into other operations
- Every node in the graph defines a function of the variables
- E.g.

$$f(x, y, z) = (x + y) * z$$

$$q = x + y; f = q * z$$



- Compute the gradients via a *backward pass*



- Notation for "local gradient":

$$\overline{y} \triangleq \frac{d\mathcal{L}}{dy}$$

  called the **error signal**

Multivariate Chain Rule

- Computation graph may have a *fan-out*

- Suppose we have a function $f(x, y)$ and functions $x(t), y(t)$.where the variables are scalar-valued. Then

$$\frac{d}{dt} f(x(t), y(t)) = \frac{\partial f}{\partial x}\frac{dx}{dt} + \frac{\partial f}{\partial y}\frac{dy}{dt}$$

  in error signal notation:

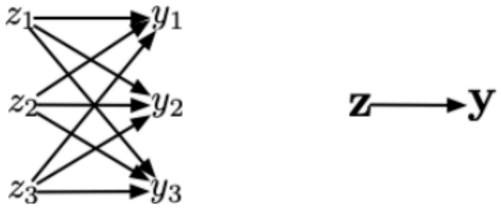
$$\bar{t} = \bar{x}\frac{dx}{dt} + \bar{y}\frac{dy}{dt}$$

Backpropagation Algorithm

- Let $v_1, \ldots, v_N$ be a *topological ordering* of the computation graph (i.e. parents come before children)
- Let v_N be the variable we are trying to compute derivatives of (e.g. loss)
- Forward pass:
  1. For $i = 1, \ldots, N$ do compute v_i as a function of Parent(v_i)
- Backward pass:
  1. $\bar{v}_N = 1$
  2. For $i = N-1, \ldots, 1$ do compute $\bar{v}_i = \sum_{j \in \text{Child}(v_i)} \bar{v}_j \frac{\partial v_j}{\partial v_i}$

Vector Form

- Consider the following computation graph:



  where $\mathbf{z} \in \mathbb{R}^N$ and $\mathbf{y} \in \mathbb{R}^M$

- Backprop rules:

$$\bar{z}_j = \sum_k \bar{y}_k \frac{\partial y_k}{\partial z_j} \quad \Longrightarrow \quad \bar{\mathbf{z}} = \frac{\partial \mathbf{y}}{\partial \mathbf{z}}^\top \bar{\mathbf{y}}$$

  where $\partial \mathbf{y} / \partial \mathbf{z}$ is the *Jacobian matrix*

$$\left(\frac{\partial \mathbf{y}}{\partial \mathbf{z}}\right)_{M \times N} = \begin{bmatrix} \frac{\partial y_1}{\partial z_1} & \cdots & \frac{\partial y_1}{\partial z_n} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial z_1} & \cdots & \frac{\partial y_m}{\partial z_n} \end{bmatrix}$$

- We never explicitly construct the Jacobian, it is simpler and more efficient to compute the Vector Jacobian Product (VJP) directly

Hessian

$$\mathbf{H} = \begin{bmatrix} \frac{\partial^2 \mathcal{L}}{\partial x_1^2} & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_1 \partial x_n} \\ \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_2^2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_2 \partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{L}}{\partial x_n \partial x_1} & \frac{\partial^2 \mathcal{L}}{\partial x_n \partial x_2} & \cdots & \frac{\partial^2 \mathcal{L}}{\partial x_n^2} \end{bmatrix}$$

- We never explicitly construct the Hessian, it is simpler and more efficient to compute the Vector Hessian Product (VHP) directly

Backward Pass For Vector Form

1. $\overline{\mathbf{v}_n} = 1$
2. For $i = N-1, \ldots, 1$ do compute $\overline{\mathbf{v}_i} = \sum_{j \in \text{Child}(\mathbf{v}_i)} \frac{\partial \mathbf{v}_j}{\partial \mathbf{v}_i}^\top \overline{\mathbf{v}_j}$

Computational Cost

- Forward pass: 1 *add-multiply operation* per weight

$$z_i = \sum_j w_{ij}^{(1)} x_j + b_i^{(1)}$$

- Backward pass: 2 *add-multiply operations* per weight

$$\overline{w_{ki}^{(2)}} = \overline{y}_k h_i$$
$$\overline{h}_i = \sum_k \overline{y}_k w_{ki}^{(2)}$$

- Backward pass is 2x more expensive than forward pass
- For a MLP, the cost is *linear* in the number of layers, and *quadratic* in the number of units per layer

5 Automatic Differentiation

Terminology

- **Automatic differentiation (autodiff)**: a general way of taking a program which computes a value, and automatically constructing a procedure for computing derivatives of that value
- **Backpropagation**: a special case of autodiff applied to neural nets
  - In machine learning, we use backprop synonymously with autodiff
- **Autograd**: a particular autodiff library

Autodiff

- An autodiff system converts the program into a sequence of *primative operations (ops)* which have specified routines for computing derivatives
- In this representation, backprop can be done in a complete mechanical way

Building the Computation Graph

- Most autodiff systems, including Autograd, explicitly construct the computation graph
  - Autograd builds them by *tracing* the forward pass computation, allowing for an interface similar to NumPy
- A node of the computation graph is represented by the `Node` class, with attributes
  - `value`, the actual value computed on a particular set of inputs
  - `fun`, the primitive operation defining the node
  - `args` and `kwargs`, the arguments the op was called with
  - `parents`, the parent `Node`s
- Autograd's fake NumPy module (`import autograd.numpy as np`) provides primitive ops which look like NumPy functions, but also builds the computation graph

Vector-Jacobian Products

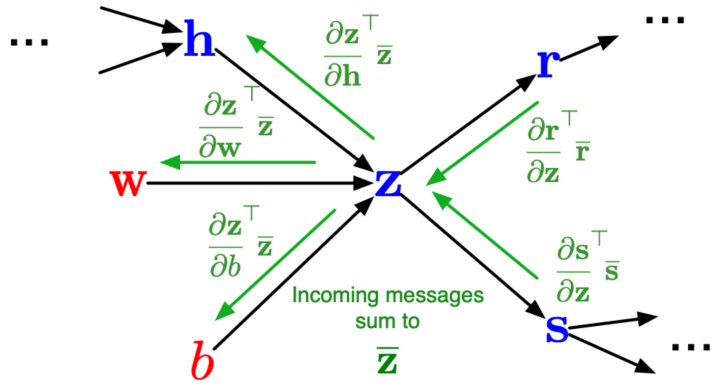
- For each primitive operation, we must specify VJPs for *each* of its arguments
- `defvjp` can be used to register VJPs, which are added to a dict
  - E.g.

```
defvjp(negative, lambda g, ans, x: -g)
defvjp(exp, lambda g, ans, x: ans * g)
defvjp(log, lambda g, ans, x: g / x)
```

Backprop as Message Passing

- Each node receives a bunch of messages from its children, which it aggregates to get its error signal
- It then passes messages to its parents
- Each of these messages is a VJP
- Each node needs to know how to compute its outgoing messages, i.e. the VJPs corresponding to each of its parents

- The implementation of z doesn't need to know where $\overline{z}$ comes from



Grad

- `grad` is a wrapper around `make_vjp`, which uilds the computation graph and feeds it to `backward_pass`
- Can be viewed as a VJP, if we treat $\overline{\mathcal{L}}$ as a 1×1 matrix

6 Distributed Reprsentations

Probability and Bayes' Rule

- Goal: build a speech recognition system, want to be able to infer a likely sentence s fiven the observed speech signal a
- The *generative* approach is to build 2 components
  1. An **observation model**, represented as $\Pr(a \mid s)$, which tells us how likely the sentence s leads to the acoustic signal a
  2. A **prior**, represented as $\Pr(s)$, which tells us how likely a given sentence s is
- Having those components, we can use **Bayes' Rule** to infer a **posterior distribution** over sentences given the speech signal:

$$\Pr(s \mid a) = \frac{\Pr(s)\Pr(a \mid s)}{\sum_{s'} \Pr(s')\Pr(a \mid s')}$$

Language Modelling

- **Language modelling**: learning a good distribution $\Pr(s)$ of sentences
- Assume we have a corpus of sentences $s^{(1)}, \ldots, s^{(N)}$, the **maximum likelihood** criterion says we want our model to maximize the probability our model assigns to the observed sentences
- Under the assumption that the sentences are independent, we maximize:

$$\prod_{i=1}^{N} \Pr\left(s^{(i)}\right)$$

- Can maximize the **log probability**, which decomposes as a sum

$$\log \prod_{i=1}^{N} \Pr\left(s^{(i)}\right) = \sum_{i=1}^{N} \log \Pr\left(s^{(i)}\right)$$

- Let the sentence s be a sequence of words $w_1, \ldots, w_T$, then using the *chain rule of conditional probability*:

$$\Pr(s) = \Pr(w_1, \ldots, w_T) = \Pr(w_1)\Pr(w_2 \mid w_1) \cdots \Pr(w_T \mid w_1, \ldots, w_{T-1})$$

  - The language modelling problem is equivalent to being able to predict the next word
- We typically make a **Markov assumption**, i.e. the distribution over the next word only depends on the preceding few words
  - E.g. if we use a context of length 3:

$$\Pr(w_t \mid w_1, \ldots, w_{t-1}) = \Pr(w_t \mid w_{t-3}, w_{t-2}, w_{t-1})$$

  - Such a model is called **memoryless**
  - This becomes a supervised prediction problem
  - The model is **autoregressive**

N-Gram Language Models

- We can learn a Markov model using a *conditional probability table* (i.e. that specify the probability of each word transitioning to each other word)

- Can estimate the probabilities from the *empirical distribution*, e.g.

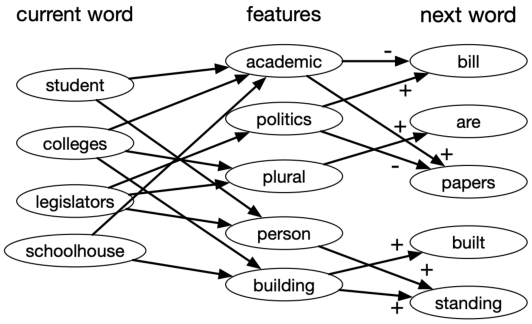
$$\Pr(w_3 = \text{cat} \mid w_1 = \text{the}, w_2 = \text{fat}) = \frac{\Pr(w_1 = \text{the}, w_2 = \text{fat}, w_3 = \text{cat})}{\Pr(w_1 = \text{the}, w_2 = \text{fat})} \approx \frac{\text{count(the fat cat)}}{\text{count(the fat)}}$$

- The phrases we are counting are called **n-grams**, where n is the length, so this is an **n-gram language model**
- Problems with this approach
  - The number of entries in the conditional probability table is exponential in the context length
  - *Data sparsity*: most n-grams never appear in the corpus, even if they are possible
- Traditional ways to deal with data sparsity
  - Use a short context (tradeoff being model is less powerful)
  - Smooth the probabilities (i.e. adding imaginary counts)
  - Make predictions using an ensemble of n-gram models with different n

Distributed Representations

- Conditional probability tables are a kind of **localist representation**, i.e. all the information about a particular word is stored in one place (a column of the table)
- Since different words are related, we want to be able to *share information* between them
- **Distributed representation**: the information of a given word is distributed throughout the representation

	bill	is	are	papers	built	standing
academic	−			+		
politics	+			−		
plural		−	+			
person						+
building					+	+



  - Representation of how each attribute influences the next word

Neural Language Model

- Predicting the distribution of the next word given the previous K is a multiway classification problem
- **Inputs**: previous K words
- **Target**: next word

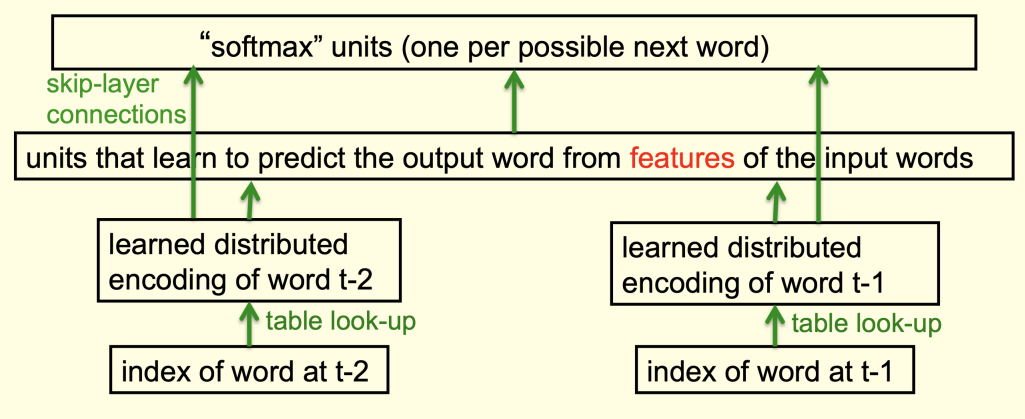
- **Loss**: cross-entropy (equivalent to maximum likelihood)

$$-\log \Pr(s) = -\sum_{t=1}^{T} \log \Pr(w_t \mid w_1, \ldots, w_{t-1}) = -\sum_{t=1}^{T} \sum_{v=1}^{V} t_{tv} \log y_{tv}$$

  where

  - t_{iv} is the one-hot encoding for the ith word
  - y_{iv} is the predicted probability for the ith word being index v

Neural Language Model



- If we use a 1-of-K encoding for the words, the first layer can be tought of as a linear layer with *tied weights*
- The weight matrix acts like a lookup table
  - Each column is the *representation* of the word, also called an **embedding**, **feature vector**, or **encoding**
- High-dimensional embeddings
  - Most vectors are nearly orthogonal (i.e. dot product close to 0)
  - Most points are far away from each other

GloVe: Neural Language Model as Matrix Factorization

- Global Vector (GloVe) embeddings are a simpler and faster approach based on matrix factorization similar to principal component analysis (PCA)
- **Distributional hypothesis**: words with similar distributions have similar meanings
- Consider a *co-occurrence matrix* X, which counts the number of times two words appear nearby (i.e. less than p positions apart)
  - This is a $V \times V$ matrix, where V is the vocabulary size (very large)
  - Suppose we fit a rank-K approximation $X \approx R\tilde{R}^\top$, where R and $\tilde{R}$ are $V \times K$ matrices
  - Each row r_i of R is the K-dimensional representation of a word
  - Each entry is approximated as $x_{ij} = r_i^\top \tilde{r}_j$
  - This means that more similar words are more likely to co-occur

- Minimizing the squared Frobenius norm

$$\left\| X - R\tilde{R}^\top \right\|_F^2 = \sum_{i,j} \left(x_{ij} - r_i^\top \tilde{r}_j \right)^2$$

  is equivalent to PCA

- Since X is large, we can reweight the entries so that only nonero counts matter
- Since word counts are a heavy-tailed distribution, the most common words would dominate the cost function, we approximate $\log x_{ij}$ instead of x_{ij}
- GloVe embedding cost function:

$$\mathcal{J}(R) = \sum_{i,j} f(x_{ij}) \left(r_i^\top \tilde{r}_j + b_i + \tilde{b}_j - \log x_{ij} \right)^2$$

$$f(x_{ij}) = \begin{cases} \left(\frac{x_{ij}}{100} \right)^{3/4}, & \text{if } x_{ij} < 100 \\ 1, & \text{if } x_{ij} \geq 100 \end{cases}$$

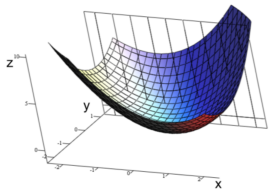
  - b_i and $\tilde{b}_j$ are bias parameters
  - Can avoid computing $\log 0$ since $f(0) = 0$
  - Since we only need to consider the nonzero entries of X, we get a big compuutational savings since X is very sparse

Word Analogies

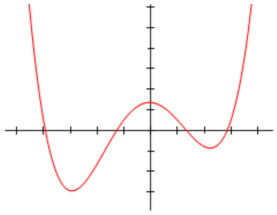
- The mappings (e.g. city $\to$ capital) corresponds roughly to a single direction in the vector space of word representations
- Can perform arithmetic on word vectors
  - E.g. for the word analogy "A is to B as C is to ?", we can find the word that is closest to $B - A + C$

7 Optimization

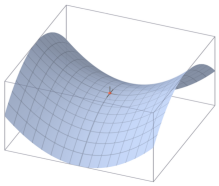
Optimization Landscapes



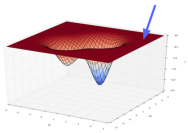
convex functions



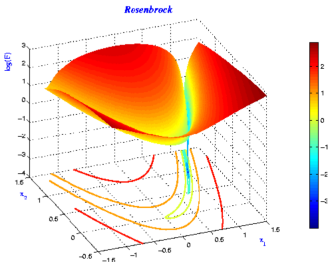
local minima



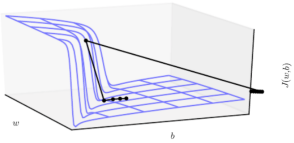
saddle points



plateaux



narrow ravines



cliffs (covered in a later lecture)

Hessian Matrix

- The **Hessian matrix**, denoted H, or $\nabla^2 \mathcal{J}$, a is a matrix os second derivatives

$$H = \nabla^2 \mathcal{J} = \begin{bmatrix} \frac{\partial^2 \mathcal{J}}{\partial \theta_1^2} & \frac{\partial^2 \mathcal{J}}{\partial \theta_1 \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_1 \partial \theta_D} \\ \frac{\partial^2 \mathcal{J}}{\partial \theta_2 \partial \theta_1} & \frac{\partial^2 \mathcal{J}}{\partial \theta_2^2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_2 \partial \theta_D} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 \mathcal{J}}{\partial \theta_D \partial \theta_1} & \frac{\partial^2 \mathcal{J}}{\partial \theta_D \partial \theta_2} & \cdots & \frac{\partial^2 \mathcal{J}}{\partial \theta_D^2} \end{bmatrix}$$

- Symmetric matrix because $\frac{\partial^2 \mathcal{J}}{\partial \theta_i \partial \theta_j} = \frac{\partial^2 \mathcal{J}}{\partial \theta_j \partial \theta_i}$
- Locally, a function can be approximated by its *second-order Taylor approximation* around a point $\boldsymbol{\theta}_0$

$$\mathcal{J}(\boldsymbol{\theta}) \approx \mathcal{J}(\boldsymbol{\theta}_0) + \nabla \mathcal{J}(\boldsymbol{\theta}_0)^\top (\boldsymbol{\theta} - \boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top H(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

- A *critical point* is a point where the gradient is 0, in which case

$$\mathcal{J}(\boldsymbol{\theta}) \approx \mathcal{J}(\boldsymbol{\theta}_0) + \frac{1}{2} (\boldsymbol{\theta} - \boldsymbol{\theta}_0)^\top H(\boldsymbol{\theta}_0)(\boldsymbol{\theta} - \boldsymbol{\theta}_0)$$

- Since the Hessian is symmetric, it has only real eigenvalues, and there is an orthogonal basis of eigenvectors

  $$H = Q \Lambda Q^\top \qquad \text{(Spectral Decomposition)}$$

  where

  - Q is an orthogonal matrix (whose columns are eigenvectors)
  - Λ is a diagonal matrix (whose diagonal entries are the eigenvalues)
- H can be thought of as the *curvature* of a function
  - Suppose we move along a line defined by $\boldsymbol{\theta} + tv$ for some vector v, then the second-order Taylor approximation is
$$\mathcal{J}(\boldsymbol{\theta} + tv) \approx \mathcal{J}(\boldsymbol{\theta}) + t\nabla\mathcal{J}(\boldsymbol{\theta})^\top v + \frac{t^2}{2} v^\top H(\boldsymbol{\theta}) v$$
  - In a direction where $v^\top H v > 0$, the cost function curves upwards, i.e. has *positive curvature*
  - In a direction where $v^\top H v < 0$, the cost function curves downwards, i.e. has *negative curvature*
- A matrix A is **positive definite** if $v^\top A v > 0$ for all $v \neq 0$ (i.e. it curves upwards in all directions)
  - A matrix is positive definite iff all its eigenvalues are positive
- A matrix A is **positive semidefinite (PSD)** if $v^\top A v \geq 0$ for all $v \neq 0$
  - A matrix is PSD iff all its eigenvalues are nonnegative
- For any critical point $\boldsymbol{\theta}_*$, if $H(\boldsymbol{\theta}_*)$ exists and is positive definite, then $\boldsymbol{\theta}_*$ is a local minimum (since all directions curve upwards)

Convex Functions

- A set S is **convex** if for any $x_0, x_1 \in S$,
$$(1 - \lambda)x_0 + \lambda x_1 \in S$$
for $0 \leq \lambda \leq 1$
- A function f is **convex** if for any x_0, x_1,
$$f((1 - \lambda)x_0 + \lambda x_1) \leq (1 - \lambda)f(x_0) + \lambda f(x_1)$$
  - Equivalently, the set of points lying above the graph of f is convex
  - f is bowl-shaped
- If $\mathcal{J}$ is **smooth** (twice differentiable), then it is convex iff its Hessian is PSD everywhere
  - Squared error, logistic-cross-entropy, and softmax-cross-entropy are all convex
- For a linear model, $z = w^\top x + b$ is a linear function of w and b. If the loss function is convex as a function of z, then it is convex as a function of w and b
  - Linear regression, logistic regression, and softmax regression are all convex

Local Minima

- If a function is convex, it has no **spurious local minima**, i.e. any local minimum is also a global minimum
- Training a network with hidden units cannot be convex because of *permutation symmetries* (i.e. we can re-order the hidden units in a way that preseerves the function computed by the network)
- Special case: a univariate function is convex iff its second derivative is nonnegative everywhere

Saddle Points

- **Saddle point**: a point where

1. $\nabla \mathcal{J}(\boldsymbol{\theta}) = 0$
2. $H(\boldsymbol{\theta})$ has some positive and some negative eigenvalues (i.e. some directions with positive curvature and some with negative curvature)

- If we are exactly on the saddle point, then we are stuck
- Can get unstuck by a slight perturbation
- Avoid initializing all weights to zero

Plateaux

- **Plateau**: a flat region
- Examples: 0-1 loss, hard threshold, logistic activation with least squares
- **Saturated unit**: being in the flat region of the activation function (e.g. end of the sigmoid curve)
- If there is a ReLU unit whose input is always negative, then the weight derivatives will be 0
  - Called a **dead unit**

Ill-Conditioned Curvature

- Long, narrow ravines
  - Suppose H has some large positive eigenvalues (i.e. high-curvature directions) and some eigenvalues close to 0 (i.e. low-curvature directions)
  - Gradient descent bounces back and forht in the high-curvature directions and makes slow progress in low-curvature directions
  - Known as **ill-conditioned curvature**, common in neural net training
  - Consider a convex quadratic objective

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{2}\boldsymbol{\theta}^\top A \boldsymbol{\theta}$$

    where A is PSD
  - Can be shown that for learning rate α and an eigenvalue λ_i:
    - If $0 < \alpha\lambda_i \leq 1$, then the loss decays to 0
    - If $1 < \alpha\lambda_i \leq 2$, then the loss oscillates
    - If $\alpha\lambda_i > 2$, then the loss diverges
- Ill-conditioned curvature bounds the maximum learning rate choice
  - Need to set the learning rate to $\alpha < 2/\lambda_{\max}$ to prevent instability, where $\lambda_{\max}$ is the largest eigenvalue
  - This bounds the rate of progress in the other directions as

$$\alpha\lambda_i < \frac{2\lambda_i}{\lambda_{\max}}$$

  - The quantity $\lambda_{\max}/\lambda_{\min}$ is the **condition number** of A
- We can center the inputs to have zero mean and unit variance, i.e.

$$\tilde{x}_j = \frac{x_j - \mu_j}{\sigma_j}$$

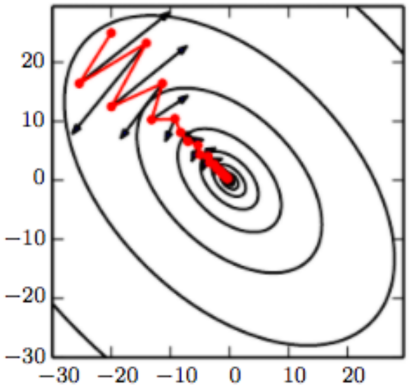
- For hidden units, we could replace logistic units (which ranges from 0 to 1) with tanh units (which ranges from -1 to 1)
- Can use **batch normalization** to explicitly center each hidden activation and speed up training

Momentum

- Inspired by momentum in physics

$$p \leftarrow \mu p - \alpha \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + p$$

- α is the learning rate
- μ is a damping parameter that is slightly less than 1 (e.g. 0.9 or 0.99)
  - If $\mu = 1$, then p will never settle down (by conservation of energy)
- In high-curvature directions, the gradients cancel each other out, so momentum dampens the oscillations
- In low curvature directions, the gradient points in the same direction, allowing the parameters to pick up speed



- If the gradient is constant, the parameters will reach a terminal velocity of

$$-\frac{\alpha}{1-\mu} \cdot \frac{\partial \mathcal{J}}{\partial \boldsymbol{\theta}}$$

  - This suggests that if we increase μ, then we should lower α to compensate

Ravines

- Even with the aforementioned tricks, narrow ravines are still an obstacle
- The curvature can be many orders of magnitude larger in some directions than others
- Area of research: *second-order optimization*
  - Difficult to scale to large neural nets and large datasets
- Can use the *Adam* optimizer which uses some curvature information

RMSProp and Adam

- **RMSProp**: a variant of GD which rescales each coordinate of the gradient to have norm 1 on average
  - Achieved by keeping an exponential moving average s_j of the squared gradients
- Performs the following update to each coordinate j:

$$s_j \leftarrow (1-\gamma)s_j + \gamma\left(\frac{\partial \mathcal{J}}{\partial \theta_j}\right)^2$$
$$\theta_j \leftarrow \theta_j - \frac{\alpha}{\sqrt{s_j+\epsilon}}\frac{\partial \mathcal{J}}{\partial \theta_j}$$

- If the eigenvectors of the Hessian are axis-aligned (i.e. the Hessian is diagonal, which is a dubious assumption), then RMSProp can corect for the curvature
- RMSProp works slightly better than SGD in practice
- **Adam**: RMSProp + momentum

Gradient Descent

- Goal: minimize a specific objective function

$$\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{J}^{(i)}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\mathcal{L}\left(y\left(\mathbf{x}^{(i)},\boldsymbol{\theta}\right),t^{(i)}\right)$$

- First calculate gradient

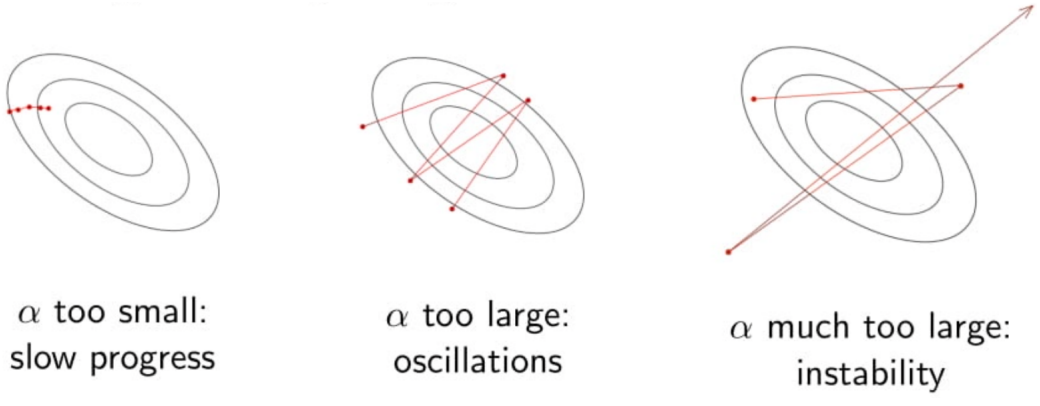
$$\nabla\mathcal{J}(\boldsymbol{\theta}) = \frac{1}{N}\sum_{i=1}^{N}\nabla\mathcal{J}^{(i)}(\boldsymbol{\theta})$$

- Then update parameters

$$\boldsymbol{\theta} = \boldsymbol{\theta} - \alpha\nabla\mathcal{J}(\boldsymbol{\theta})$$

Learning Rate

- The learning rate α is a hyperparameter

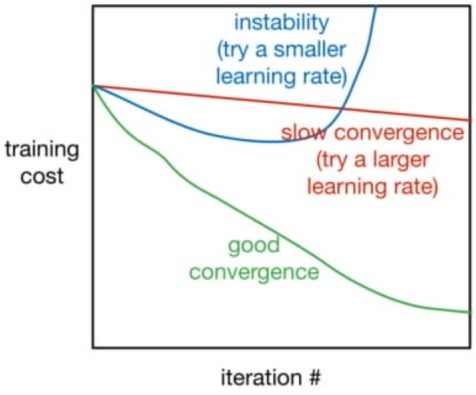


- Good values are typically between 0.001 and 0.1

Training Curves

- **Training curve**: plot of the training cost as a function of iteration
- Use a fixed subset of the training data to monitor the training error

- Only looking at training curves does not guarantee convergence

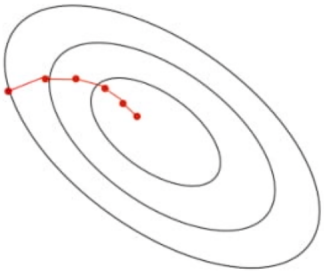


Stochastic Gradient Descent

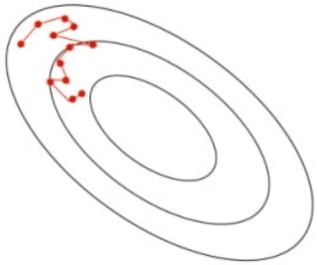
- **Batch training**: compute the gradient by summing over *all* of the training examples
  - Impractical when dataset is large
- **Stochastic gradient descent (SGD)**: update the parameters based on the gradient for a single training example

$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \alpha \nabla \mathcal{J}^{(i)}(\boldsymbol{\theta})$$

- SGD makes progress before looking at all the data
- Stochastic gradient is an *unbiased estimate* of the batch gradient



batch gradient descent



stochastic gradient descent

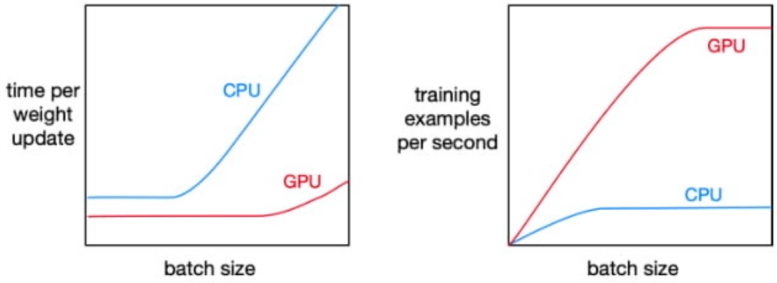
Mini-Batch Gradient Descent

- If we only look at one training example at a time, we can't exploit efficient vectorized operations
- We can compute the gradients on a medium-sized set of training examples, called a **mini-batch**
- Each entire pass over the dataset is called an **epoch**
- Stochastic gradients computed on larger mini-batches have smaller variance

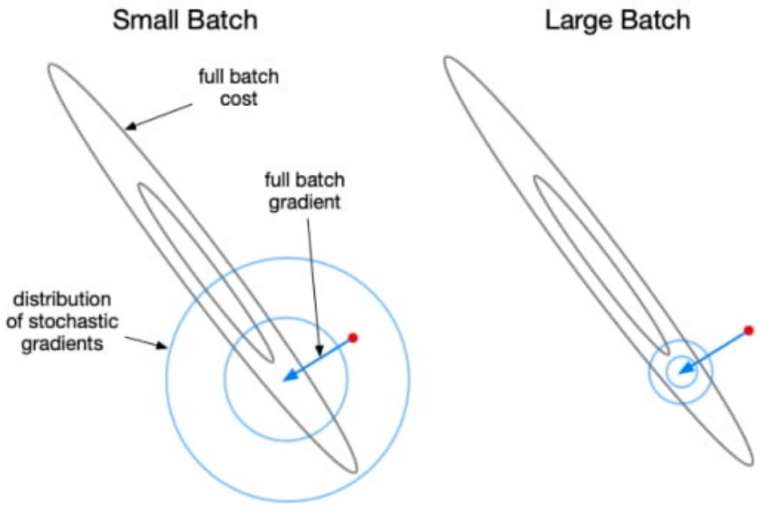
Batch Size

- The mini-batch size S is a hyperparameter
- Large batches: converge in fewer weight updates because each stochastic gradient is less noisy

- Small batches: perform more weight updates per second because each one requires less computation
- If the wall-clock time were proportional to the number of FLOPs, then $S = 1$ would be optimal
  - 100 updates with $S = 1$ requires the same FLOP count as 1 update with $S = 100$
  - With $S = 1$, we compute the gradient at a fresher value of $\boldsymbol{\theta}$ every time, which is better
- We don't use $S = 1$ because larger batches can take advantage of fast matrix operations and parallelism
- An update with $S = 10$ isn't much more expensive than an update with $S = 1$
- Once S is large enough to saturate the hardware efficiencies, the cost becomes linear in S
- GPUs afford more parallelism, so they favour larger batch sizes



- Small batches have large gradient noise, so increasing it could help reducing gradient noise
- If the batch size is already large, then SGD approximates the batch gradient descent updates, so no further benefit from variance reduction



- Could try values 32, 64, 128, 256, etc.
- Tune batch size and learning rate *after* tuning all other hyperparameters

Initialization

- If two hidden units have exactly the same bias and exactly the same incoming/outoging weights, they will always gt exactly the same gradient
  - So they could never learn to be different features
  - We could break symmetry by initializing the weights to have small random values

- If a hidden unit has a big fan-in, small changes on many of its incoming weights can cause the learning to overshoot
  - We want smaller incoming weights when the fan-in is big
  - Typically initialize the weights to be proportional to the square root of fan-in
- When activation function is linear (or close to linear), use Xavier initialization:

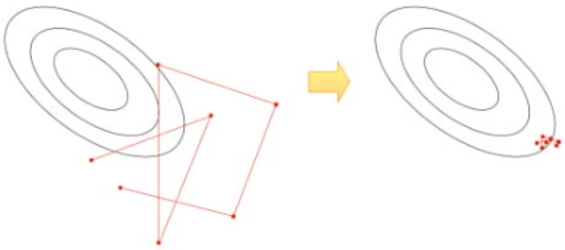
```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in)
```

- When activation function is ReLU, use He initialization:

```
W = np.random.randn(fan_in, fan_out) / np.sqrt(fan_in / 2)
```

Learning Rate

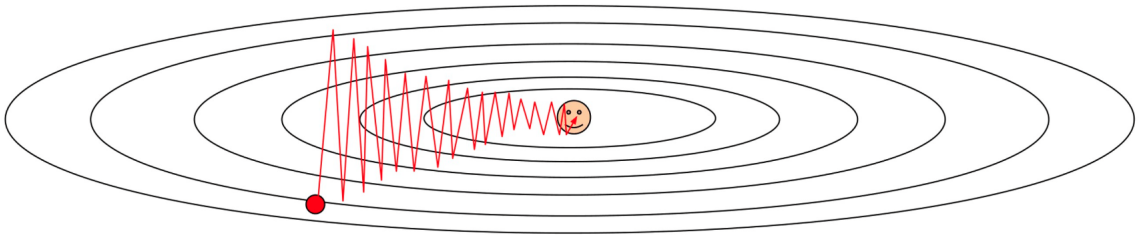
- In stochastic training, the learning rate also influences the *fluctuations* due to the stochasticity of the gradients
- Use a large learning rate early in training, then gradually decay the learning rate to reduce the fluctuations
- We could guess an initial learning rate
  - If the error keeps getting worse or oscillates wildly, then reduce the learning rate
  - If the error is falling fairly consistently but slowly, increase the learning rate
- Turn down the learning rate when the error stops decreasing
- By reducing the learnig rate, we reduce the fluctuations
  - Could appear as loss dropping suddenly
  - But it could come at the expense of long-run performance



- Decaying learning rate over time
  - Step decay: decay learning rate by a factor (e.g. half) every few epochs
  - Exponential decay: $\alpha = \alpha_0 e^{-kt}$
  - $1/t$ decay: $\alpha = \alpha_0/(1 + kt)$
- Can use a grid search to pick values on a logarithmic scale (e.g. 0.1, 0.01, 0.001, etc.)

SGD With Momentum

- Suppose the loss function is steep vertically but shallow horizontally, SGD would perform the following updates



  - Very slow progress alng flat direction, jitter along steep one

- Momentum intuition
  - The weight starts off by following the gradient, but once it has velocity, it no longer does steepest descent
  - Its momentum makes it keep going in the previous direction
  - In directions of high curvature, it damps oscillations by combining gradients with opposite signs
  - It builds up speed in directions with a gentle but consistent gradient

```
vw = 0
beta1 = 0.9

for t in range(1, num_iters):
    batch = get_batch()
    loss = compute_loss(batch, w)

    dw = compute_gradient(loss)
    vw = beta1 * vw + (1 - beta1) * dw

    w -= alpha * vw
```

RMSProp

- RProp: using the gradient but also dividing by the size of the gradient
  - With mini-batch RProp, we divide by a different number for each mini-batch
  - We want to divide by similar numbers for adjacent mini-batches

- RMSProp: keep a moving average of the squared gradient for each weight

$$\text{MeanSquare}(w,t) = 0.9 \cdot \text{MeanSquare}(w,t-1) + 0.1 \left(\frac{\partial E}{\partial w}(t)\right)^2$$

- We could divide the gradient by

$$\sqrt{\text{MeanSquare}(w,t)}$$

```
moment2 = 0
beta2 = 0.9
c = 10e-8

for t in range(1, num_iters):
```

```
        batch = get_batch()
        loss = compute_loss(batch, w)

        dw = compute_gradient(loss)
        moment2 = beta2 * moment2 + (1 - beta2) * dw * dw

        w -= alpha * dw / (sqrt(moment2) + c)
```

Adam

- Adam = Momentum + RMSProp
- May not converge to optimal solution
- Works well in practice

```
    moment1 = 0
    moment2 = 0
    beta1 = 0.9
    beta2 = 0.999
    c = 10e-8

    for t in range(1, num_iters):
        batch = get_batch()
        loss = compute_loss(batch, w)

        dw = compute_gradient(loss)
        moment1 = beta1 * moment1 + (1 - beta1) * dw # Momentum
        moment2 = beta2 * moment2 + (1 - beta2) * dw * dw # RMSProp

        # Bias correction
        unbias1 = moment1 / (1 - beta1 ** t)
        unbias2 = moment2 / (1 - beta2 ** t)

        w -= alpha * unbias1 / (sqrt(unbias2) + c)
```

Learning Process

Problem	**Diagnostics**	**Workarounds**
incorrect gradients	finite differences	fix them, or use autodiff
local optima	(hard)	random restarts
slow progress	slow, linear training curve	increase α, use momentum
instability	cost increases	decrease α
oscillations	fluctuations in training curve	decrease α, use momentum
fluctuations	fluctuations in training curve	decay α, iterate averaging
dead/saturated units	activation histograms	better initialize $\mathbf{W}$, use ReLU
ill-conditioning	(hard)	normalize, momentum/Adam, 2nd-order optimization

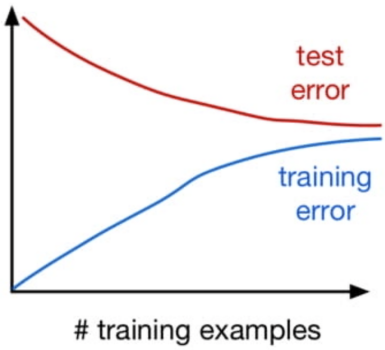
Good Practice in Learning Process

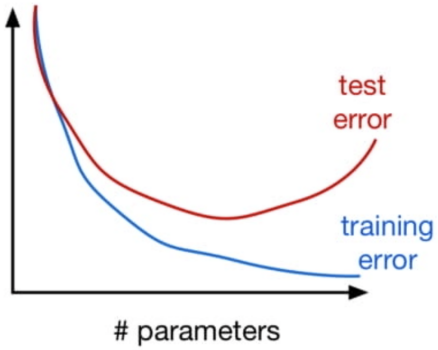
- Tune the batch size and initial learning rate using grid search by monitoring training and validation curves
- Use adaptive learning rate decay
- Choose a good initialization
- Use Adam or SGD with momentum

8 Generalization

Generalization

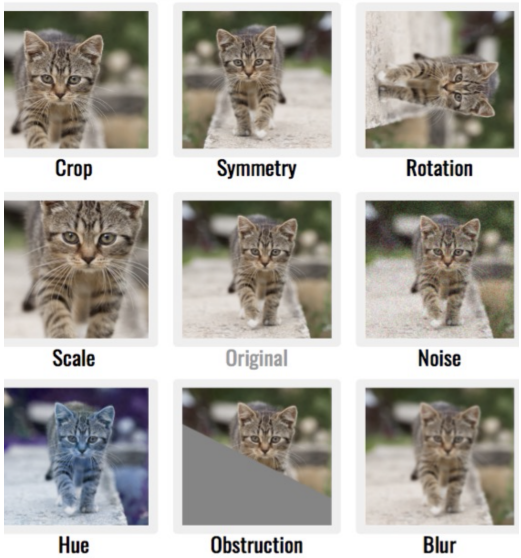
- We want to minimize the generalization error, i.e. error on novel examples



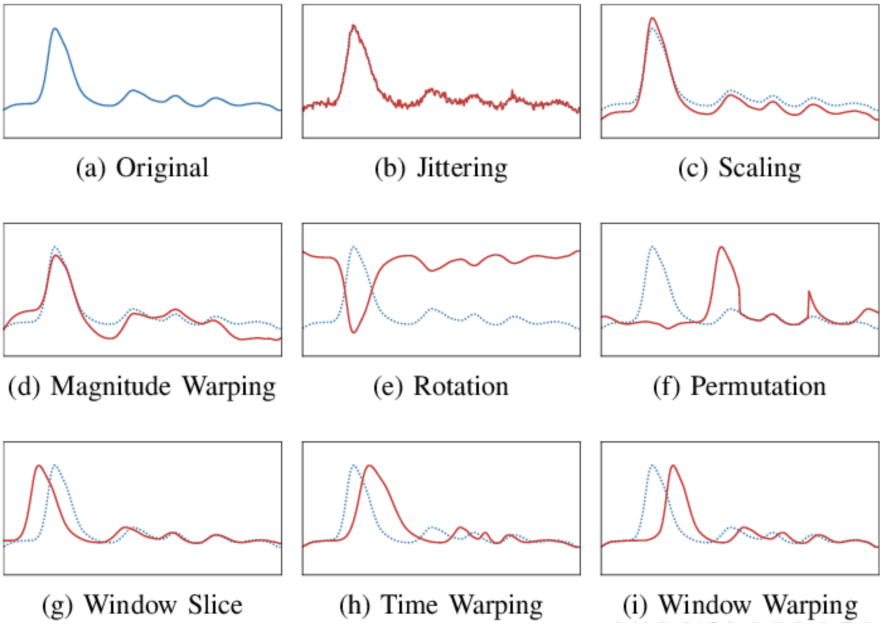


Data Augmentation

- The best way to improve generalization is to collect more data
- **Data augmentation**: augment the training data by transforming the examples
  - E.g. (for visual recognition)
    - Translation
    - Horizontal/vertical flip
    - Rotation
    - Smooth warping
    - Noise (e.g. flipping random samples)
- We only warp the training samples, not the test samples
- Choice of transformations depend on the task (e.g. horizontal flip for object recognition, but not handwritten digit recognition)
- For vision:



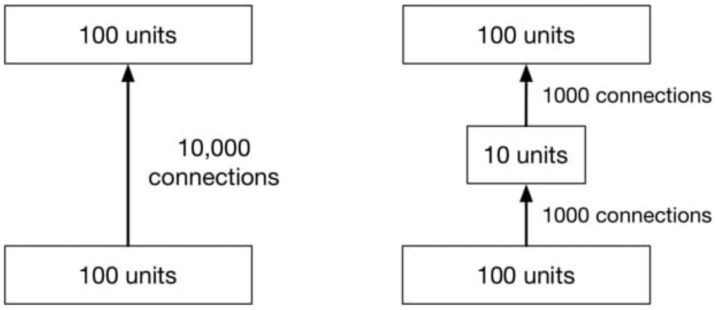
- For time series:



- For natural language:
  - Synonym replacement
  - Random insertion
  - Random swap
  - Back translation (translate to another language and back)

Reducing the Number of Parameters

- Can reduce the number of layers or the number of parameters per layer
- Can add a linear *bottleneck layer*



  - The first network is strictly more expressive than the second (since linear layers don't make a network more expressive)
- E.g. Autoencoder

Weight Decay

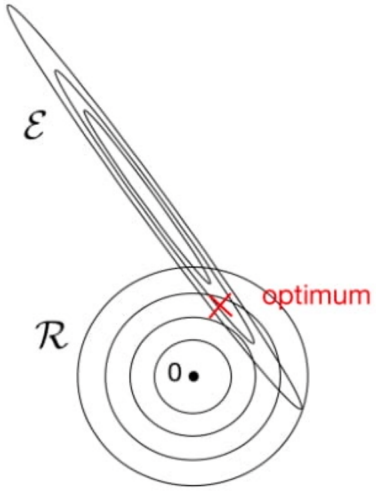
- We can *regularize* a network by penalizing large weight values, thereby encouraging the weights to be small in magnitude

$$\mathcal{J}_{\text{reg}} = \mathcal{J} + \lambda \mathcal{R} = \mathcal{J} + \frac{\lambda}{2} \sum_j w_j^2$$

- In this way, gradient descent update can be interpreted as **weight decay**

$$\mathbf{w} \leftarrow (1 - \alpha\lambda)\mathbf{w} - \alpha\frac{\partial \mathcal{J}}{\partial \mathbf{w}}$$

- Large weights tend to overfit
- Geometric picture:



- Other kinds of regularizers include sum of absolute values
- Regularizers differ by how strongly they prioritize making weights exactly zero vs. not being very large

Early Stopping

- We don't always want to find an optimum for our cost function, it may be advantageous to stop training early
- **Early stopping**: monitor performance on a validation set, stop training when the validation error starts going up
- The validation error could fluctuate because of the stochasticity in the updates
- Since weights start out small, it takes time for them to grow large, therefore, early stopping has a similar effect to weight decay

Ensembles

- If a loss function is convex, we have a lot of predictions, and we don't know which one is best, then we could take their average

$$\mathcal{L}(\lambda_1 y_1 + \cdots + \lambda_N y_N, t) \leq \lambda_1 \mathcal{L}(y_1, t) + \cdots + \lambda_N \mathcal{L}(y_N, t) \quad \text{for } \lambda_i \geq 0, \sum_i \lambda_i = 1$$

  - This is true as long as the loss function is conve (e.g. squared error, cross-entropy, etc.)
- **Ensemble**: having multiple candidate models and averaging their predictions on the test data
- Examples of ensembles
  - Training networks starting from different random initializations (may not give enough diversity to be useful)

  - Training networks on different subsets of the training data (called **bagging**)
  - Training networks with different architectures or hyperparameters, or use other algorithms which aren't neural nets
- Ensembles are expensive, and the predictions are hard to interpret

Stochastic Regularization

- Overfitting networks usually have very precise computations
- **Stochastic regularization**: injecting noise into the computations
- **Dropout**: stochastic regularizer which randomly deactivates a subset of the units (i.e. set their activations to 0)

$$h_j = \begin{cases} \phi(z_j) & \text{with probability } 1-\rho \\ 0 & \text{with probability } \rho \end{cases}$$

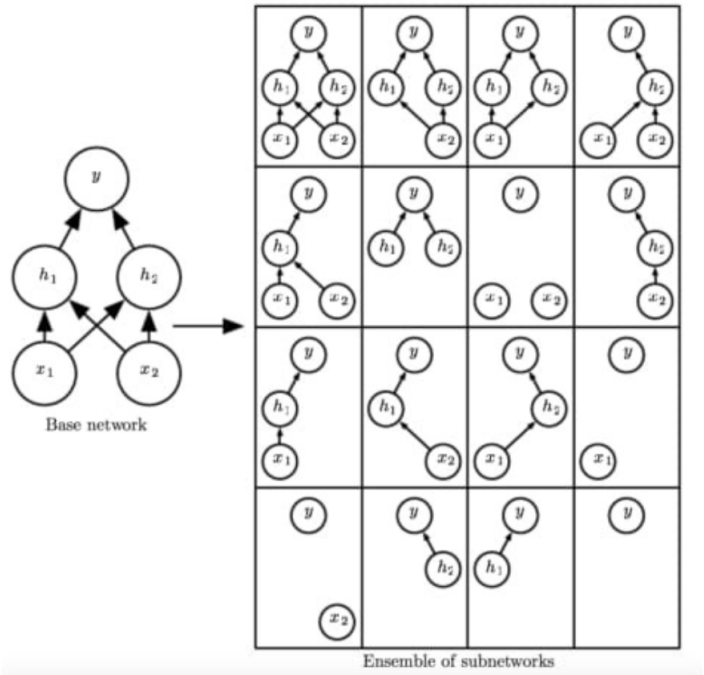
  where ρ is a hyperparameter
- Equivalently,

$$h_j = m_j \phi(z_j)$$

  where m_j is a Bernoulli random variable, independent for each hidden unit
- Backprop rule:

$$\overline{z}_j = \overline{h}_j \cdot m_j \cdot \phi'(z_j)$$

- Dropout can be seen as training an ensemble of 2^D different architectures with shared weights (where D is the number of units)



- At test time, multiply the weights by $1-\rho$ (since the weights are on $1-\rho$ fraction of the time), matching the expectation
- Other stochastic regularizers include batch normalization
- Stochasticity of SGD updates can act as a regularizer (i.e. small mini-batch size)

9 Convolutional Neural Networks

Computer Vision

- Vision needs to be robust to transformations and distortions
  - Change in pose/viewpoint
  - Change in illumination
  - Deformation
  - Occlusion (some objects hidden behind others)
- Many object categories can vary in appearance (e.g. chairs)
- MLP is inefficient in computer vision
  - Too many parameters, since the input size is the number of pixels $\times 3$ (RGB)
  - Each feature (hidden unit) looks at the entire image, but we might not want the first layer to depend on pixels far away
- Want the incoming weights to focus on *local* patterns of the input image
- The same sorts of features that are useful in analyzing one part of the image will probably be useful for analyzing other parts as well (e.g. edges, corners, etc.)
- Want a neural net architecture that enables learning a set of feature detectors *shared* at all image locations

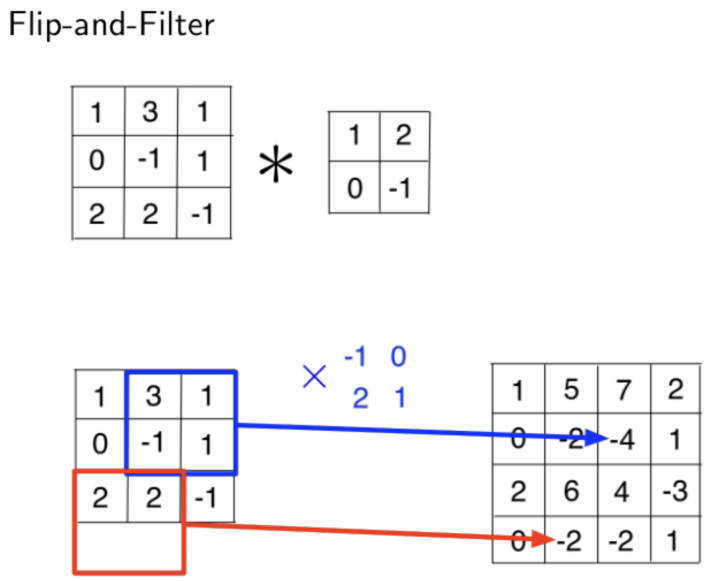
Convolution

- 1-D convolution, assuming a and b are two arrays:

$$(a * b)_t = \sum_\tau a_\tau b_{t-\tau}$$

- 2-D convolution, assuming A and B are 2-D arrays:

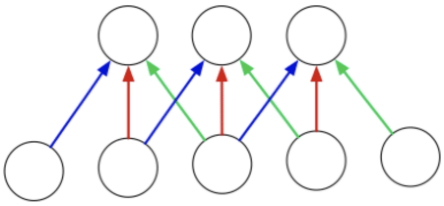
$$(A * B)_{ij} = \sum_s \sum_t A_{st} B_{i-s,j-t}$$

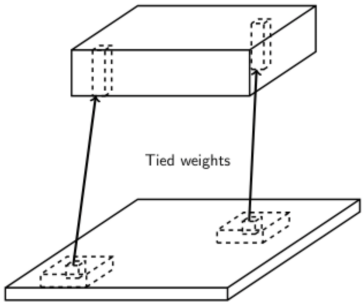


- Properties
  - Commutativity: $a * b = b * a$
  - Linearity: $a * (\lambda_1 b + \lambda_2 c) = \lambda_1 a * b + \lambda_2 a * c$

Convolution Layers

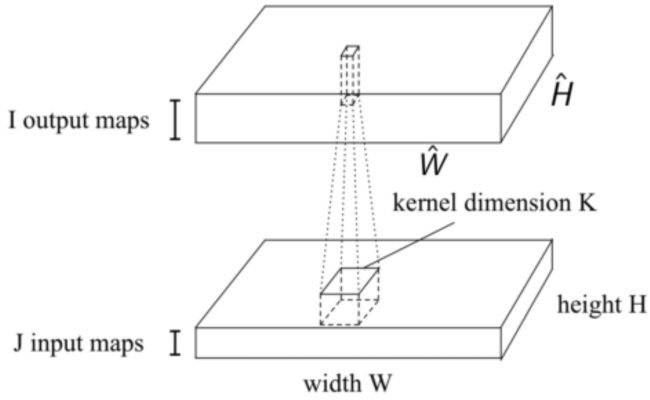
- For fully connected layers, each hidden unit looks at the entire image
- We can have locally connected layers, where each column of hidden units looks at a small region of the image
- Convolution layers is locally connected layers where the weights are shared across the entire image





- Padding: add a border of zeros around the image
- Stride: the number of pixels to shift the filter at each step
- Convolution layers can be stacked

Convolution Layer Parameters



- Input: an array of size $W \times H \times J$
- Hyperparameters:
  - Number of filters M
  - Size of filters K
  - Stride S
  - Number of zero-padding P
- Output: feature maps of size $\hat{W} \times \hat{H} \times I$

- $\hat{W} = (W - K + 2P)/S + 1$
- $\hat{H} = (H - K + 2P)/S + 1$
- $I = M$

Size of a Network

- Measuring the size of a network
  - **Number of units**: important because the activations need to be stored in memory during training (for backprop)
  - **Number of weights**: important because the weights need to be stored in memory, and this number determines the amount of overfitting
  - **Number of connections**: important because there are approximately 3 add-multiply operations per connection (1 in forward pass, 2 in backward pass)
- A fully connected layer with M input units and N output units has MN connections and MN weights

Size of a Conv Net

- Without bias

(Without Bias)	**Fully Connected Layer**	**Convolution Layer**
# Output Units	$\hat{W}\hat{H}I$	$\hat{W}\hat{H}I$
# Weights	$W\hat{W}H\hat{H}IJ$	K^2IJ
# Connections	$W\hat{W}H\hat{H}IJ$	$\hat{W}\hat{H}K^2IJ$

- With bias

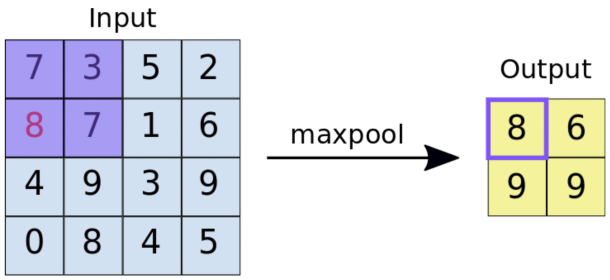
(With Bias)	**Fully Connected Layer**	**Convolution Layer**
# Output Units	$\hat{W}\hat{H}I$	$\hat{W}\hat{H}I$
# Weights	$W\hat{W}H\hat{H}IJ + \hat{W}\hat{H}I$	$K^2IJ + I$
# Connections	$W\hat{W}H\hat{H}IJ + \hat{W}\hat{H}I$	$\hat{W}\hat{H}K^2IJ + \hat{W}\hat{H}I$

- Most of the units and connections are in the convolution layers
- Most of tthe weights are in the fully connected layers (if there are any)

Pooling Layers

- **Pooling layers** reduce the size of the representatino and build in invariance to small transformations
- We most commonly use **max-pooling**, which computes the maximum value of the units in a **pooling group**:

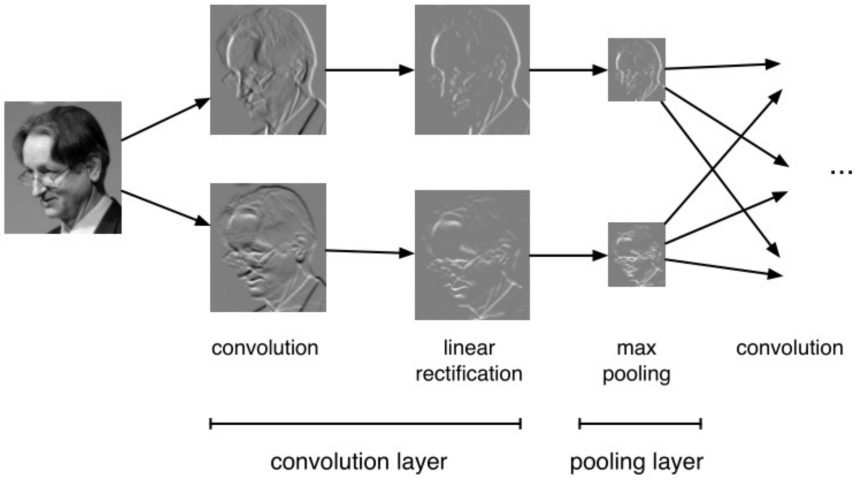
$$y_i = \max_{j \in \text{pooling group}} z_j$$



Pooling Layer Parameters

- Input: an array of size $W \times H \times J$
- Hyperparameters:
  - Size of filters K
  - Stride S
- Output: feature maps of size $\hat{W} \times \hat{H} \times I$
  - $\hat{W} = (W - K)/S + 1$
  - $\hat{H} = (H - K)/S + 1$
  - $I = J$
- Common setting: $K = 2, S = 2$

Convolutional Networks



- Two kinds of layers
  1. **Detections layers** (or **convolution layers**)
  2. **Pooling layers**
- The convolution layer has a set of filters, its output is a set of *feature maps*, each one obtained by convolving the image with a filter
- Common to apply a linear rectification nonlinearity (ReLU)
  - Convolution is linear, therefore we need a nonlinearity
  - Two edges in opposite directions should not cancel
- Because of pooling, higher-layer filters can cover a larger region of the input than equal-sized filters in lower layers

Equivariance and Invariance

- Convolution layers are **equivariant**, i.e. if we translate the inputs, the outputs are translated by the same amount
- Want the network's predictions to be **invariant**, i.e. if we translate the inputs, the prediction should not change

- Pooling layers provide invariance to small translations

Object Recognition

- Object recognition is the task of identifying which object category is present in an image
- Challenging because objects can differ in position/size/shape/appearance, and we have to deal with occlusions, lighting change, etc.
- The best object recognizers are conv nets
- Choice of dataset
  - Which categories to include?
  - Where should the images come from?
  - How many images to collect?
  - How to normalize/preprocess the images?
- Well-known datasets
  - MNIST dataset of handwritten digits
    - LeNet achieved high accuracy on this dataset
  - ImageNet: full-sized images of different categories
    - AlexNet and GoogLeNet achieved high accuracy on this dataset

Semantic Segmentation

- Focuses on making classification of class labels for *every pixel*
- CNN works well for this task, and can use a pre-trained ImageNet classification network

CNN on Health Care

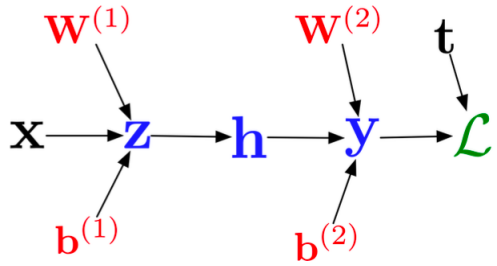
- Can use a pre-trained ImageNet classification network to perform medical image segmentation/classification

Supervised Pre-Training and Transfer Learning

- It is infeasible to train an image classifier from scratch, since we would need a large dataset
- We can fine-tune a pre-trained conv net on ImageNet or OpenImage
- We can fix most of the weights in the pre-trained network, only the weights in the last layer will be randomly initialized and learnt on the current dataset/task
- Fine-tuning
  - Fewer new examples requires more weights from the pre-trained network to be fixed
  - More fine-tuning is needed for dissimilar datasets
  - Set a low learning rate for fine-tuning

10 Interpretability

Input Gradient



- From this graph, we could compute $\partial \mathcal{L}/\partial x$, which contains the model's sensitivity w.r.t. changes of its input
- Can be used to visualize what learned features represent

Feature Visualization

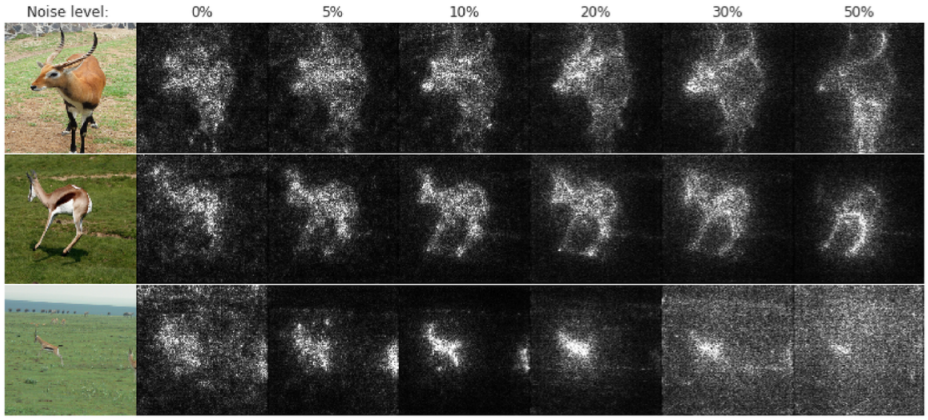
- We could pick the images in the training set which activate a unit most strongly
  - Higher layers seem to pick up more abstract, high-level information
  - However, this method can't tell what the unit is actually responding to in the image
- Input gradients can be noisy and hard to interpret
  - E.g. for a good object recognition conv net (Alex Net), we could compute the gradient of $\log \Pr(y = \text{“deer”} \mid x)$

  - This is partially due to the steepest directions are *local* sensitivity w.r.t. the current input pixels
  - It is difficult to pick out important global features from one instance of the local changes
- **SmoothGrad** is a method to estimate a global “saliency” map
  - Do the backward pass on a few noisy version of the input images, then average their input gradients

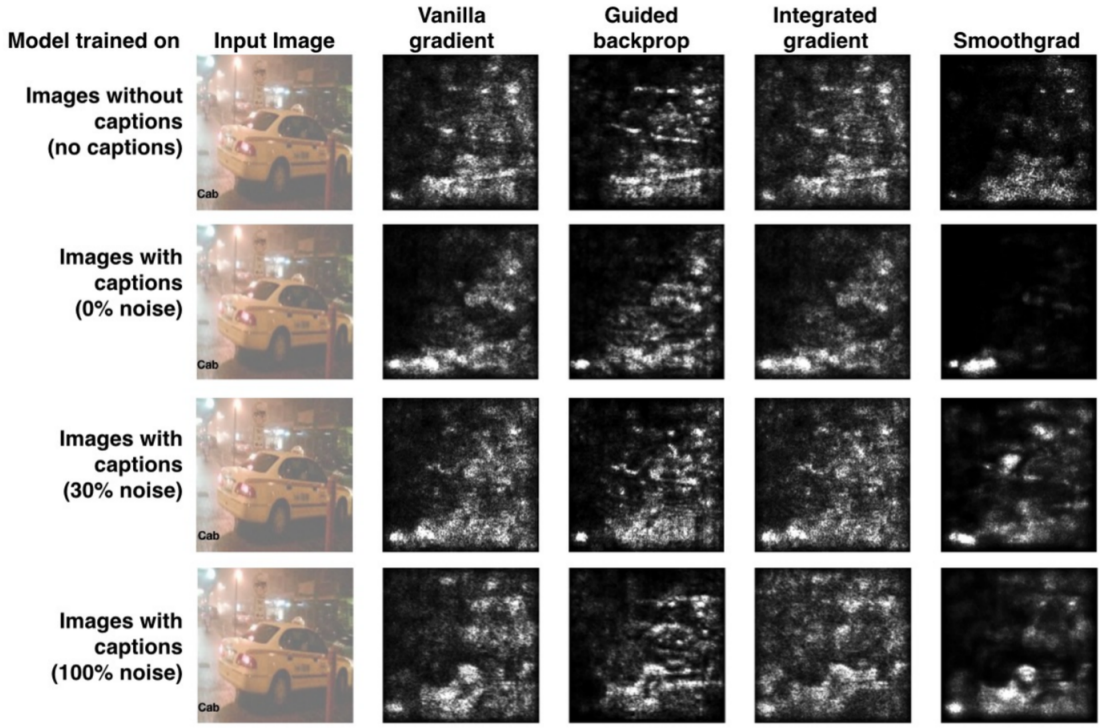
$$S_{\text{deer}} = \frac{1}{N} \sum_{i=1}^{N} \frac{\partial \mathcal{L}_{\text{deer}}}{\partial \mathbf{x}} (\mathbf{x} + \boldsymbol{\epsilon}_i), \quad \boldsymbol{\epsilon} \sim \mathcal{N}(0, \sigma^2 \mathbf{I})$$

  - This averages out the local sensitivity effect from slightly perturbed input images

- Results:



Spurious Correlation



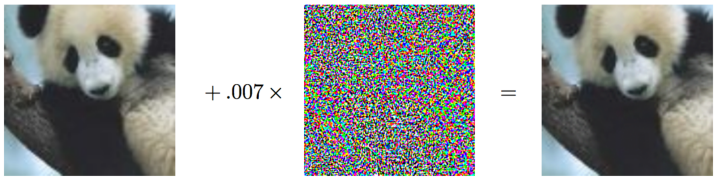
- **Spurious correlation**: unexpected correlation in training data that may not exist in test
  - E.g. the text "Cab" in the example above
- For ConvNets, we could visualize the image gradients to identify spurious correlations

Gradient Ascent on Images

- Can do gradient ascent on an image to maximize the activation of a given neuron
- The resulting image is usually a mixture of different features (one for each category that we want to maximize)

Adversarial Examples

- **Adversarial inputs**: inputs optimized to fool an algorithm
- Given an image for one category (e.g. cat), compute the image gradient to maximize the network's output unit for another category (e.g. dog)
  - Perturn the image very slightly in this direction, then the network would think it's a dog
  - Works better if we take the sign of the entries in the gradient, which is known as the **fast gradient sign method**



$\boldsymbol{x}$
"panda"
57.7% confidence

$\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"nematode"
8.2% confidence

$\boldsymbol{x} + \epsilon\text{sign}(\nabla_{\boldsymbol{x}} J(\boldsymbol{\theta}, \boldsymbol{x}, y))$
"gibbon"
99.3 % confidence

- No reliable method to defend against them (as of 2023)
- We don't need access to the original network, we could train up a new network to match its predictions, and then construct adversarial examples for that
- We could print out an adversarial image and take a picture of it, and it still works

Deep Dream

- Start with an image, and run a conv net on it
- Pick a layer in the network
- Change the image such that units which were already highly activated get activated more strongly (i.e. rich get richer)
  - E.g. set $\overline{\mathbf{h}} = \mathbf{h}$, then do backprop
- Repeat
- This accentuates whatever features of an image that already resembles the object

11 Large-Scale Generative Models

Terminology

- **Large Language Models (LLMs)**: large-scale neural networks (usually $>$ 10B parameters) trained on language modelling objective
- **Pre-trained Language Models (PLMs)**: special case of LLMs
- **Foundation Models**: umbrella name given to a wide range of pre-trained internet-scale models beyond just language models, such as vision, speech, robotics

Examples of Large-Scale Models

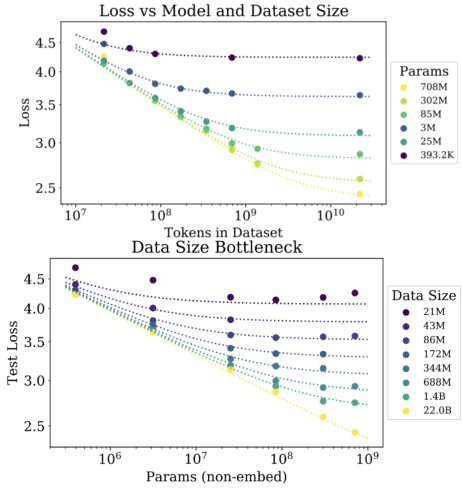
- **Contrastive Language-Image Pre-training (CLIP)**: large-scale image-text model that learns from 400 millions of training examples
- **Stable Diffusion**: large-scale image generator that uses a pre-trained CLIP model
- **Generative Pre-trained Transformer (GPT)**: large-scale language model trained on 100 billions of internet text

Performance

- Pre-trained LLMs can be used directly on new tasks
- *Chain-of-thought* prompting addresses a fundamental limit in LLMs: decouple the compute and the answer length
  - More likely to give the correct answer

Scaling Law Curves

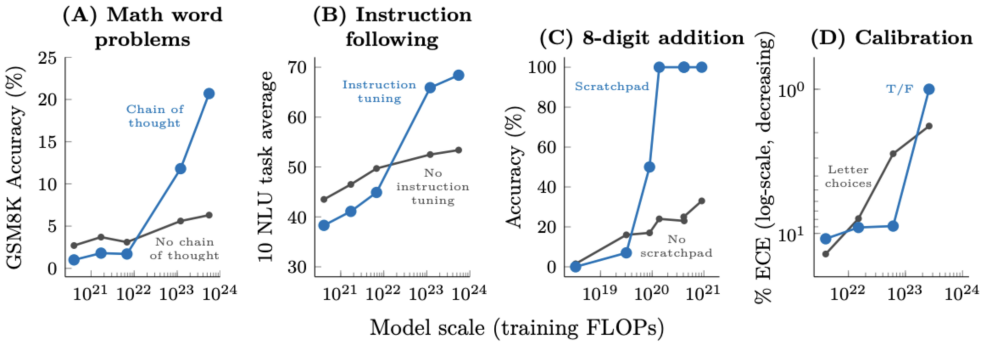
- Since compute resource has continued to scale, these models are expensive to train
- **Power-law scaling**: test loss scales as $1/x$ to both *parameters* and *dataset size* individually until one of them plateaus



- Learning algorithms can also be a bottleneck (sgd vs. adam)

Capabilities

- Some capabilities can be emergent



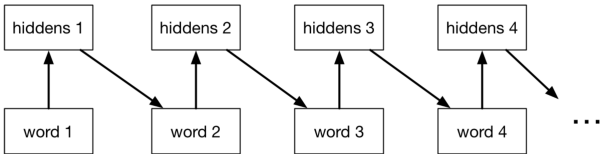
12 Recurrent Neural Networks

Overview

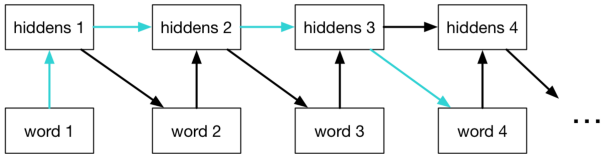
- Instead of making predictions from a fixed-size input to a fixed-size output, we may be interested in predicting sequences
  - E.g. speech-to-text, text-to-speech, caption generation, machine translation
- **Sequence-to-sequence prediction**: input and output are both sequences
- **Markov assumption**:

$$\Pr(w_i \mid w_1, \ldots, w_{i-1}) = \Pr(w_i \mid w_{i-k} \ldots, w_{i-1})$$

  which means that the model is *memoryless*
- Autoregressive models (such as the neural language model) are memoryless, so they can only use information from their immediate context (e.g. context length = 1)



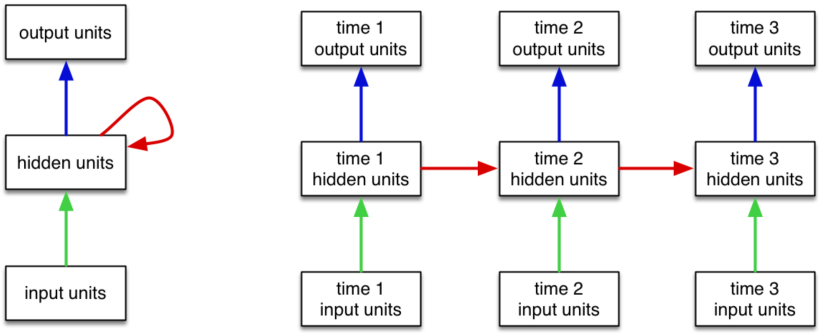
- We could add connections between the hidden units to make it a **recurrent neural network (RNN)**



  - Having a memory lets an RNN use long-term dependencies

Recurrent Neural Nets

- Can view RNN as a dynamical system with one set of hidden units which feed into themselves
- The network's graph would have self-loops
- Can *unroll* the RNN's graph by explicitly representing the units at all time steps
- Weights and biases are shared between all time steps (except possibly the first/initial time step)

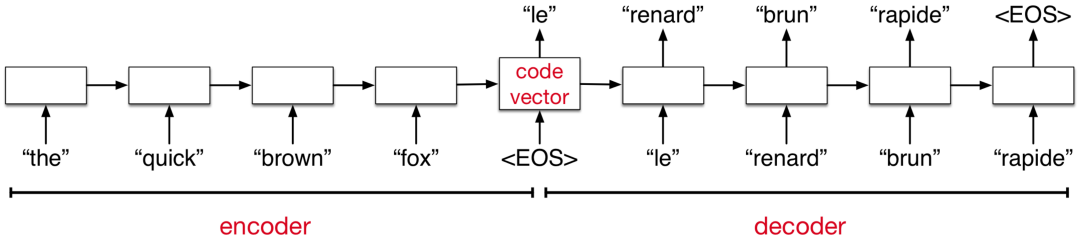


Language Modelling

- Can represent each word as an indicator vector, and let the model predict a distribution
- When we generate from the model, the outputs feed back into the network as inputs
- **Teacher forcing**: the inputs are the tokens from the training set (rather than the network's outputs) at training time
- Challenges
  - Vocabularies can be very large, predicting distributions over them could be computationally infeasible
  - We need to deal with words never seen before
  - In some languages (e.g. German), it is hard to define what should be considered a word
- We could also model text one *character* at a time
  - Can address previously unseen words
  - This makes long-term memory essential

Neural Machine Translation

- Sentences might not be the same length, words might not align perfectly
- Might need to resolve ambiguities using information from later in the sentence
- **Sequence-to-sequence architecture**: the network first reads and memorizes the sentence; when it sees the **end token**, it starts outputting the translation



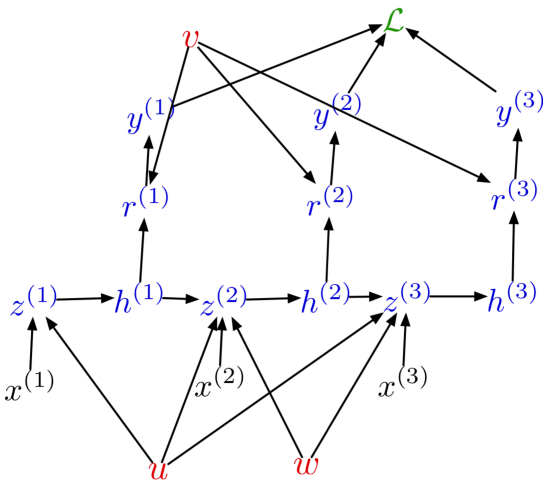
  - The encoder and decoder are two different networks with different weights

Backprop Through Time

- **Backprop through time**: learning RNN weights using backprop on the unrolled network
- The derivatives could explode or vanish
  - This is because of the Jacobians multiply

$$\frac{\partial h^{(T)}}{\partial h^{(1)}} = \prod_{t=2}^{T} \frac{\partial h^{(t)}}{\partial h^{(t-1)}}$$

  - Matrices can explode or vanish just like scalar values
  - The forward pass has nonlinear activation functions which squash the activations, preventing them from blowing up
  - The backward pass is linear, so it's hard to keep things stable



Activations:

$$\overline{\mathcal{L}} = 1$$

$$\overline{y^{(t)}} = \overline{\mathcal{L}} \frac{\partial \mathcal{L}}{\partial y^{(t)}}$$

$$\overline{r^{(t)}} = \overline{y^{(t)}}\, \phi'(r^{(t)})$$

$$\overline{h^{(t)}} = \overline{r^{(t)}}\, v + \overline{z^{(t+1)}}\, w$$

$$\overline{z^{(t)}} = \overline{h^{(t)}}\, \phi'(z^{(t)})$$

Parameters:

$$\overline{u} = \sum_t \overline{z^{(t)}}\, x^{(t)}$$

$$\overline{v} = \sum_t \overline{r^{(t)}}\, h^{(t)}$$

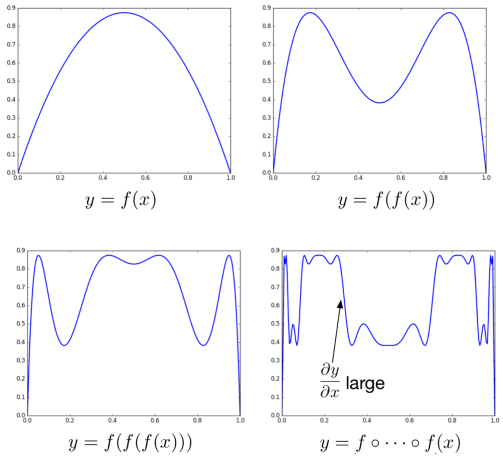
$$\overline{w} = \sum_t \overline{z^{(t+1)}}\, h^{(t)}$$

Iterated Function

- Each hidden layer computes some function of the previous hiddens and the current input, so the function gets iterated

$$h^{(4)} = f\left(f\left(f\left(h^{(1)}, x^{(2)}\right), x^{(3)}\right), x^{(4)}\right)$$

- Example of iterated function derivative



Gradient Clipping

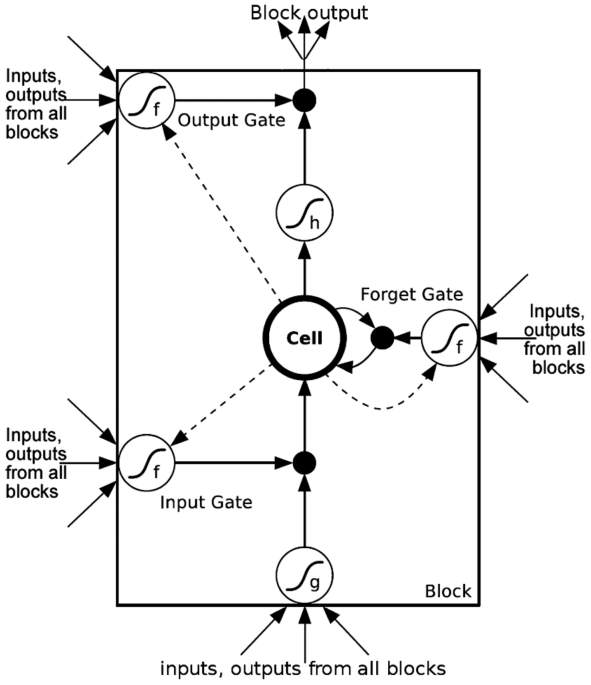
- **Gradient clipping**: clip the gradient f so that it has a norm of at most η

$$g \leftarrow \begin{cases} \frac{\eta g}{\|g\|}, & \text{if } \|g\| > \eta \\ g, & \text{elsewise} \end{cases}$$

- This prevents the gradients from blowing up

Long Short-Term Memory

- **Long Short-Term Memory**: an architecture that makes it easy to remember information over long time periods
- The network's activations are its short-term memory and its weights are its long-term memory
- Wants the short-term memory to last for a long time period
- Composed of memory cells which have controllers saying when to store or forget information
- Replaces each single unit in an RNN by a memory block



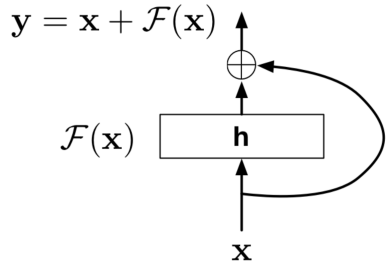
  - $c_{t+1} = c_t \times \text{forget gate} + \text{new input} \times \text{input gate}$
  - $i = 0 \wedge f = 1 \implies$ remember previous value
  - $i = 1 \wedge f = 1 \implies$ add to the previous value
  - $i = 0 \wedge f = 0 \implies$ erase the value
  - $i = 1 \wedge f = 0 \implies$ overwrite the value
- In each step, we have a vector of memory cells c, a vector of hidden units h, and vectors of input, output, and forget gates i, o, f

$$\begin{bmatrix} i_t \\ f_t \\ o_t \\ g_t \end{bmatrix} = \begin{bmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{bmatrix} W \begin{bmatrix} y_t \\ h_{t-1} \end{bmatrix}$$
$$c_t = f_t \odot c_{t-1} + i_t \odot g_t$$
$$h_t = o_t \odot \tanh(c_t)$$

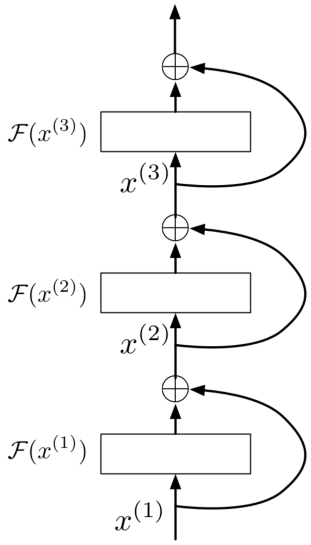
13 Attention

Deep Residual Networks and Skip Connections

- We can use linear skip connections to pass information through a network
- The exploding/vanishing gradient problems also apply to MLPs and conv nets
  - This becomes a problem if conv net is deep
- **Residual block**: each layer adds something (i.e. a residual) to the previous value, rather than producing an entirely new value



- Can string together residual blocks



- If we set the parameters such that $\mathcal{F}\left(\mathbf{x}^{(l)}\right) = 0$ in every layer, then it passes $\mathbf{x}^{(1)}$ through the network unmodified
  - It is easy for the network to represent the identity function
- Backprop:

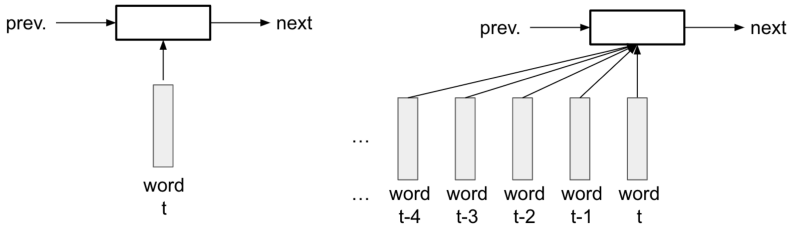
$$\overline{\mathbf{x}^{(l)}} = \overline{\mathbf{x}^{(l+1)}} + \overline{\mathbf{x}^{(l+1)}} \frac{\partial \mathcal{F}}{\partial \mathbf{x}} = \overline{\mathbf{x}^{(l+1)}} \left(\mathbf{I} + \frac{\partial \mathcal{F}}{\partial \mathbf{x}} \right)$$

  - As long as $\partial \mathcal{F} / \partial \mathbf{x}$ is small, the derivatives are stable

Attention

- Drastically improves the performance on the long sequences

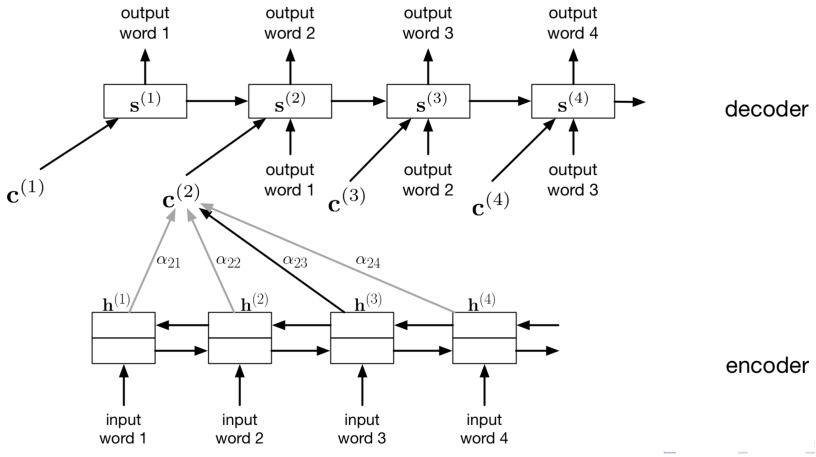
- Attention allows the model to take a "sneak peak" of the past history



- Each output word comes from one word, or a handful of words, from the input; we could learn to attend to only the relevant ones as we produce the output

Attention-Based Machine Translation

- The model has both an encoder and a decoder
- The encoder computes an *annotation* of each word in the input
  - It takes the form of a *bidirectional RNN*
  - Idea: the information earlier or later in the sentence can help disambiguate a word
  - The RNN uses an LSTM-like architecture called a *gated recurrent unit (GRU)*
- The decoder network is also an RNN, which makes predictions one word at a time, and its predictions are fed back in as inputs
  - It also receives a *context vector* $\mathbf{c}^{(t)}$ at each time step, which is computed by attending to the inputs



- The context vector is computed as a weighted average of the encoder's annotations

$$\mathbf{c}^{(i)} = \sum_j \alpha_{ij} \mathbf{h}^{(j)}$$

- The attention weights are computed as a softmax, where the inputs depend on the annotation and the decoder's state

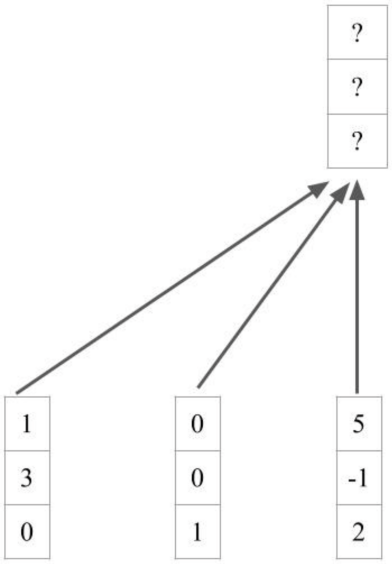
$$\alpha_{ij} = \frac{\exp(\tilde{\alpha}_{ij})}{\sum_{j'} \exp(\tilde{\alpha}_{ij'})}$$

$$\tilde{\alpha}_{ij} = f\left(\mathbf{s}^{(i-1)}, \mathbf{h}^{(j)}\right)$$

- **Content-based addressing**: the attention funcion f depends on tha annotation vector, rather than the position in the sentence
  - E.g. "my language model tells me the next word should be an adjective, find me an adjective in the input"

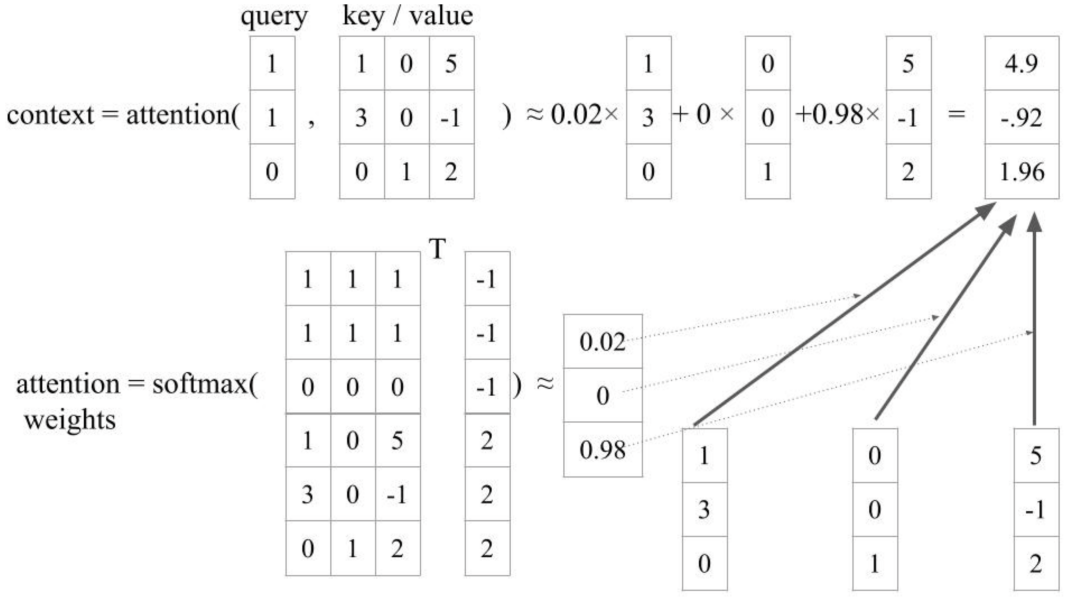
Pooling

- Want to obtain a context vector from a set of annotations



- We could use average pooling, but it is content independent

Bahdanau's Attention



Attention-Based Caption Generation

- Encoder: a classification conv net (VGGNet), which computes a bunch of feature maps over the image

- Decoder: an attention-based RNN, analogous to the decoder in the translation model
  - In each time step, the decoder computes an attention map over the entire image, effectively deciding which regions to focus on
  - It receives a context vector, which is the weighted average of the conv net features
- We could visualize where the network is looking as it generates a sentence

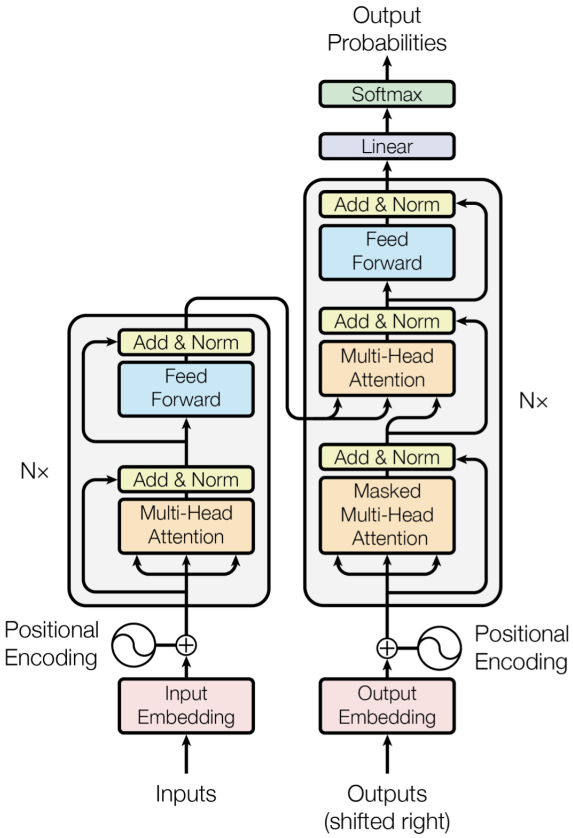
14 Transformers

Limitations of Attention in RNN

- Hard to generalize to long sequences
- Difficult to parallel the computing
- Does not fully exploit the contextual information

Transformers

- We would like our model to have access to the entire history at the hidden layers
- Can use attention to aggregate the context information by attending to one or a few important inputs from the past history
- **Transformer**: encoder-decoder architecture similar to the previous sequence-to-sequence RNN models, except all the recurrent connections are replaced by the attention modules



Self-Attention Layer

- Learn how and where to attend within the whole input sequences
- Can be computed parallelly

1. Calculate **queries**, **keys**, **values**
2. Calculate scaled dot-product attention scores

3. Calculate the output of self-attention layer

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_K}}\right)V$$

Multi-Head Attention

- Humans can attend to many things simultaneously
- We can apply scaled dot-product attention multiple times on the linearly transformed inputs

$$\text{MultiHead}(Q, K, V) = \text{concat}(\mathbf{c}_1, \ldots, \mathbf{c}_h)W^O$$
$$\mathbf{c}_i = \text{attention}\left(QW_i^Q, KW_i^K, VW_i^V\right)$$

1. Calculate multiple keys, queries, and values
2. Calculate multiple attentions
3. Concatenate and compress
   (a) Concatenate all the attention heads
   (b) Multiply with a weight matrix W^O that was trained jointly with the model
   (c) The result would be the matrix that captures information from all the attention heads, which we can send to the next layer

Masked Multi-Head Attention

- Sometimes we don't want to attend to future sequences, we can just mask them

GPS System: Positional Encoding

- The attention encoder outputs do not depend on the order of the inputs
- The order of the sequence is usually important
- We could add positional information of an input token in the sequence into the input embedding vectors

$$\text{PE}_{pos,2i} = \sin\left(\frac{pos}{10000^{2i/d_{\text{emb}}}}\right)$$
$$\text{PE}_{pos,2i+1} = \cos\left(\frac{pos}{10000^{2i/d_{\text{emb}}}}\right)$$

- The final input embeddings are the concatenation of the learnable embedding and the positional encoding

Transformer Machine Translation

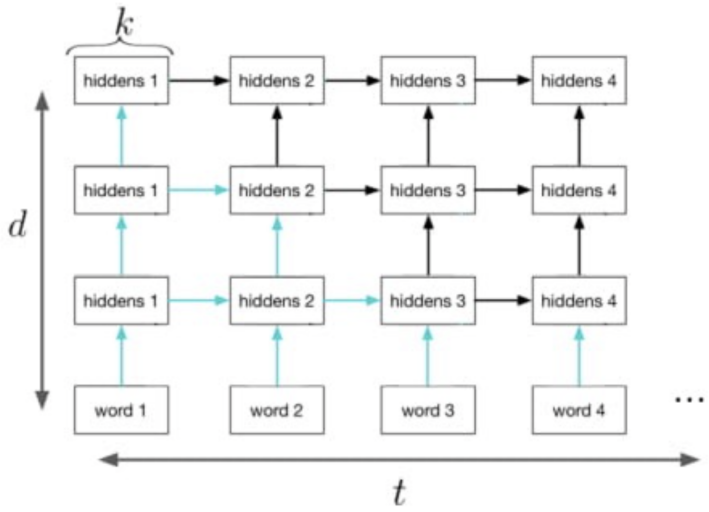
- Self attention layers learn that words like "it" could refer to different entities in different contexts

Computational Cost and Parallelism

- Computational cost
  - **Number of connections**: how many add-multiply operations for the forward and backward pass
  - **Number of time steps**: how many copies of hidden units to store for backpropagation through time

  - **Number of sequential operations**: the computations cannot be parallelized (i.e. the part that requires a for loop)

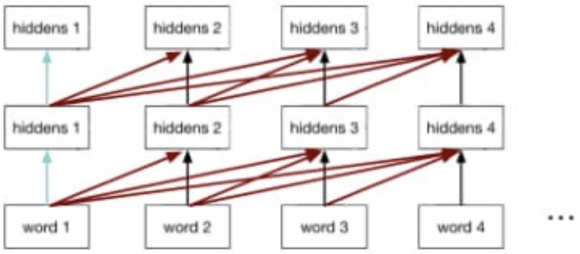
- **Maximum path length across time**: the shortest path length between the first encoder input and the last decoder output
  - Tells us how easy it is for the RNN to remember/retrieve information from the input sequence
- Consider a d layer RNN with k hidden units, training on a sequence of length t



- There are k^2 connections for each hidden-to-hidden connection, for a total of tk^2d connections
- We need to store all tkd hidden units during training
- Only kd hidden units need to be stored at test time
- During backprop, in the standard encoder-decoder RNN, the maximum path length across time is the number of time steps
- Attention-based RNNs have a *constant path length* between the encoder inputs and the decoder hidden states
- During forward pass, attention-based RNNs achieves efficient content-based addressing at the cost of re-computing context vectors at each time step
  - To compute context vector over the entire input sequence of length t using a neural network of k^2 connections, the cost at each time step is tk^2

Improve Parallelism

- RNNs are sequential in the sequence length t due to the number of hidden-to-hidden lateral connections
  - Therefore the parallelism potential for longer sequences is limited
- We can remove the lateral connections, an have a deep autoregressive model where the hidden units depend on all the previous time steps



  - The number of sequential operations is now linear in the depth d, but is independent of the sequence length t (since usually $d \ll t$)

- Self-attention allows the model to learn to access information from the past hidden layer, but decoding is very expensive
- When generating sentences, the computation in the self-attention decoder grows as the sequence gets longer
- Transformers can learn faster than RNNs on parallel processing hardwares for longer sequences

Model	**Training Complexity**	**Training Memory**	**Test Comp.**	**Test Mem.**	**Sequential Operations**	**Maximum Path Length Across Time**
RNN	tk^2d	tkd	tk^2d	kd	t	t
RNN + Attn.	t^2k^2d	t^2kd	t^2k^2d	tkd	t	1
Transformer	t^2kd	tkd	t^2kd	tkd	d	1

- Note that d is usually small

Transformer Language Pre-Training

- We can pre-train a language model for NLP tasks
- The pre-trained model is then fine-tuned on textual entailment, question answering, semantic similarity assessment, document classification, etc.
- Increasing the training data set and the model size has an improvement on the transformer language model

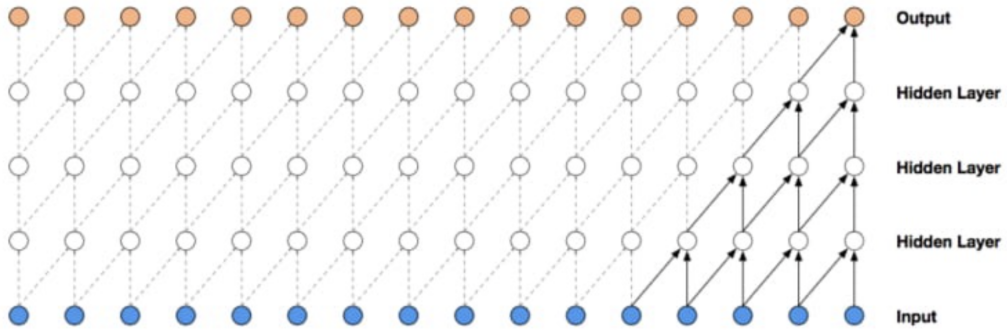
15 CNN For Speech Analysis

Overview

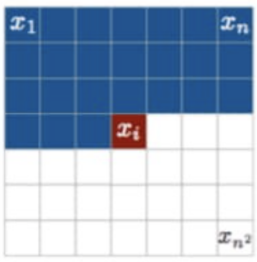
- Goal is to generate very long sequences
  - Can treat an image as a very long sequence using raster scan order
  - A speech signal can be represented as a waveform, with at least 16000 samples per second
- Training an RNN to generate these sequences requires a sequential computation of > 10000 time steps
- Transformers are too expensive to train on 10000 time steps

Causal Convolution

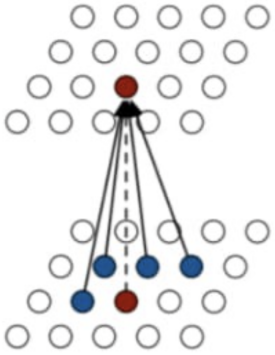
- For RNN language models, we ensured that the model was *causal*, i.e. each prediction depended only oninputs earlier in the sequence
- We can do the same thing using a convolutional architecture



  - Processing each input sequence just requires a series of convolution operations
  - No for loops
- Causal convolution for images:



The image is treated as a very long sequence of pixels using raster scan order.



We can restrict the connectivity pattern in each layer to make it causal. This can be implemented by clamping some weights to zero.

- We can turn a causal CNN into an RNN by adding recurrent connections

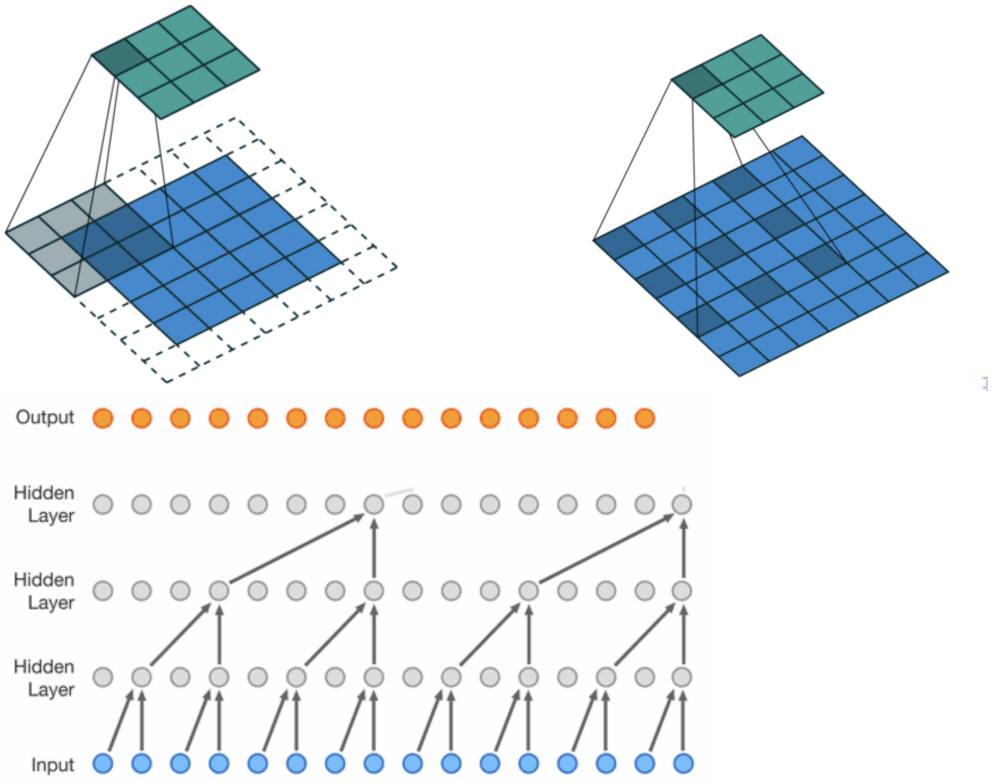
RNN	CNN
Can use information from all past time steps	Limited context
Training is expensive (for loop over time steps)	Training is cheap (convolutions)
Generating is expensive (for loop)	Generating is expensive (for loop)

PixelCNN and PixelRNN

- Two autoregressive models of images
- Output is a softmax over 256 possible pixel intensities
- Can be used to complete an image with occlusions
- Generated images are good and of high-resolution
- PixelCNN has a lower training time than PixelRNN
- The performance of PixelRNN is better than PixelCNN

Dilated Convolution

- Used to increase a CNN's receptive field



WaveNet

- Autoregressive model for raw audio based on causal dilated convolutions
- Audio needs to be sampled at at least 16000 frames per second for good quality, so the sequences are very long
- WaveNet uses dilations of $1, 2, \ldots, 512$, so each unit at the end of the block has a receptive field of length 1024, or 64ms
- It stacks several of these blocks, so the total context length is about 300ms

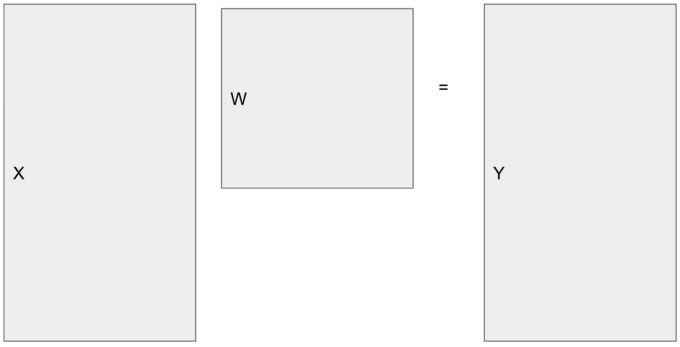
16 Large Language Models

Model Architectures

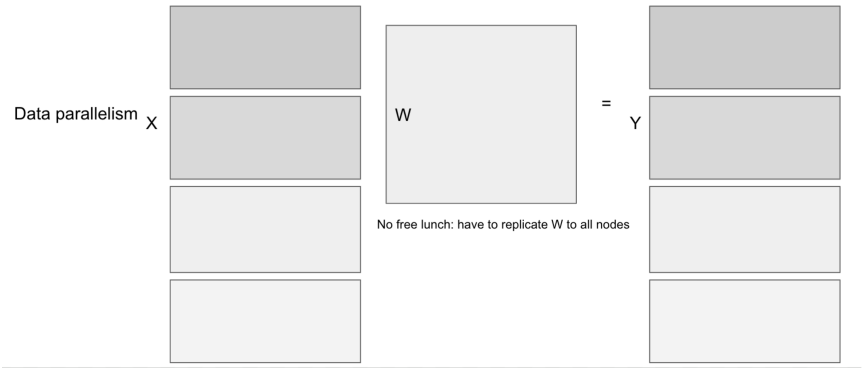
- Want to scale up transformers to billions of parameters in models like ChatGPT
- Can compute the number of parameters
- Want to know whether a model is undertrained or overtrained in term of training data size vs. model size

Data and Model Parallelism

- Matrix multiplication:

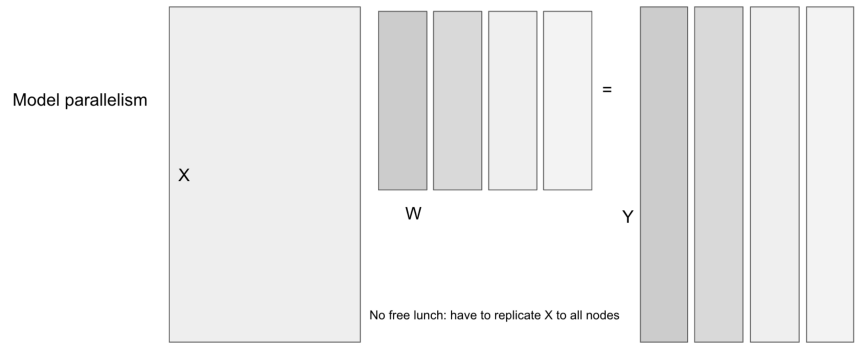


- Naive Data parallelism:



  - Have to replicate W to all nodes
  - Communication cost is $\approx 2|W|$ (one for forward pass and one for backward pass)

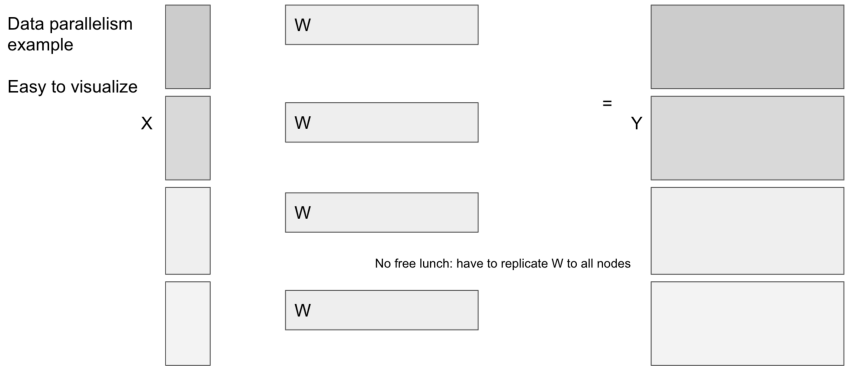
- Model parallelism:



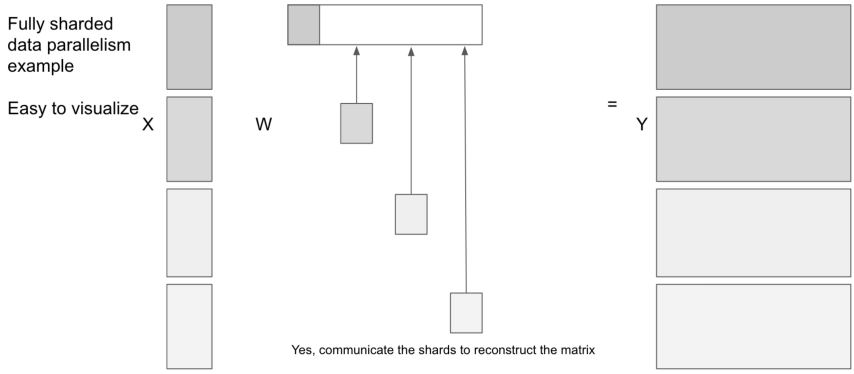
Fully Sharded Data Parallelism

- Take one column of X and one row of W, we could compute the outer product

- We could use data parallelism



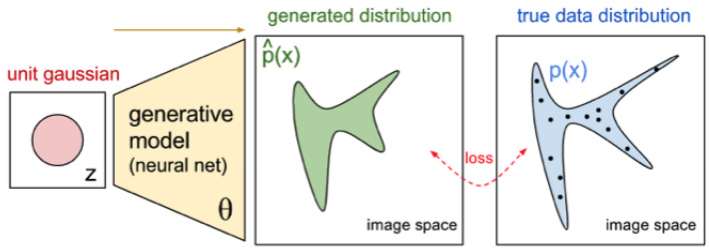
- We want to further reduce the redundancy



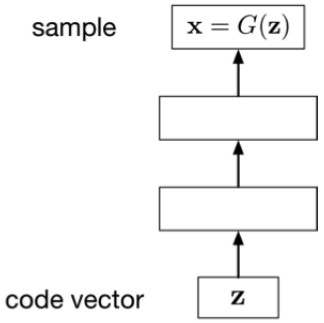
- Communication cost $\approx 3|W|$ (once forward and twice backward)
- Peak memory usage is lower

17 Generative Adversarial Networks

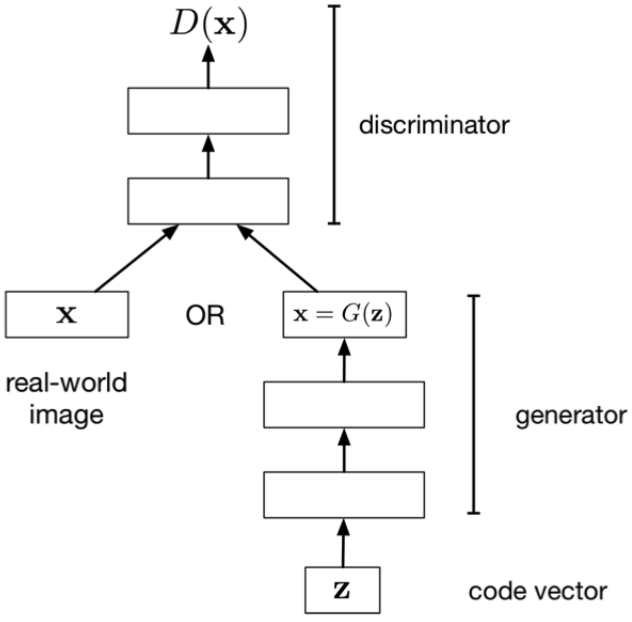
Generator Networks



- Autoregressive models explicitly predict a distribution at each step
- Can also train a NN to produce approximate samples from this distribution
- Start by sampling the *code vector* z from a fixed, simple distribution (e.g. spherical Gaussian)
- The **generator** network computes a differentiable function G mapping z to an x in data space



Generative Adversarial Networks



- Train 2 different networks

  - The **generator network** tries to produce realistic-looking samples
  - The **discriminator network** tries to figure out whether an image came from the training set or the generator network

- The generator network tries to "fool" the discriminator network
- Let D denote the discriminator's predicted probability of being data
- Discriminator's cost function: cross-entropy loss for classification

$$\mathcal{J}_D = \mathbb{E}_{x\sim\mathcal{D}}[-\log D(x)] + \mathbb{E}_z[-\log(1 - D(G(z)))]$$

- Can formulate the cost function for the generator as the opposite of the discriminator

$$\mathcal{J}_G = -\mathcal{J}_D = \text{const} + \mathbb{E}_z[\log(1 - D(G(z)))]$$

  - Called the *minimax formulation*, since the generator and discriminator are playing a *zero-sum game* against each other

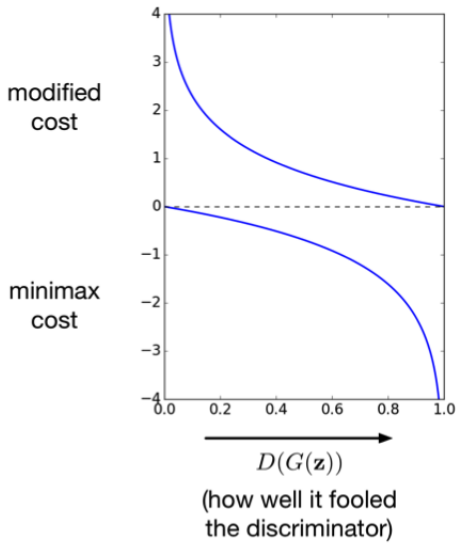
$$\max_G \min_D \mathcal{J}_D$$

- To update the discriminator, use backprop on the classification objective
- To update the generator:
  1. Backprop the derivatives of the discriminator, but don't modify the discriminator weights
  2. Flip the sign of the derivatives when we are at the generator
  3. Update the generator weights using backprop

Improving Cost Function

- The minimax cost function suffers from **saturation**
  - If the generated sample is very bad, the discriminator's prediction is close to 0, and the generator's cost would be flat
- Can modify the generator cost to

$$\mathcal{J}_G = \mathbb{E}_z[-\log D(G(z))]$$

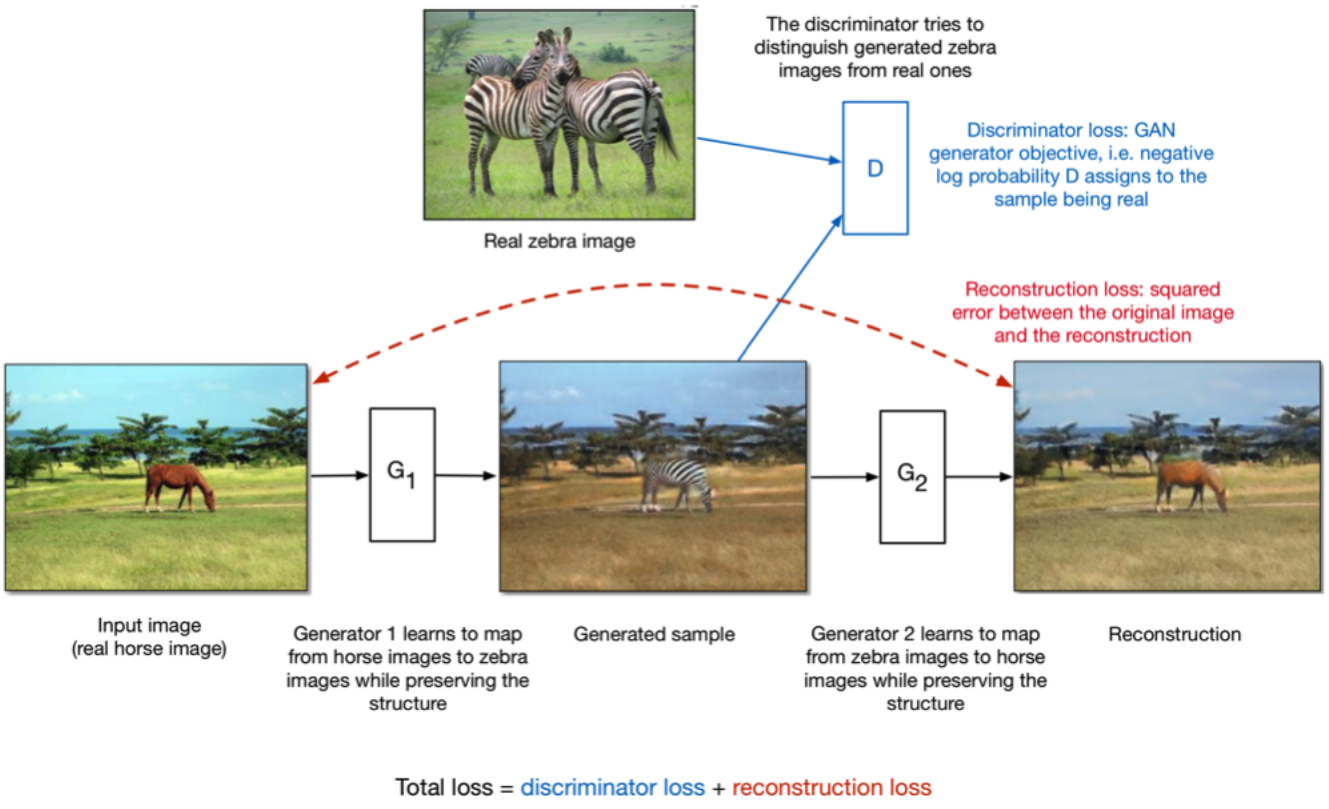


Applications

- Can produce high-resolution images
- However we don't know how well they're modelling the distribution
- No way to tell if they are dropping important modes from the distribution

CycleGAN

- Style transfer problem: change the style of an image while preserving the content
- Hard to find paired data (same content in both styles) for supervised learning
- CycleGAN trains 2 different generator nets to go from style 1 to 2, and vice versa
- Make sure the generated samples of style 2 are indistinguishable from real images by a discriminator net
- Make sure the generators are *cycle-consistent*, i.e. mapping from style 1 to 2 and back again should give almost the original image

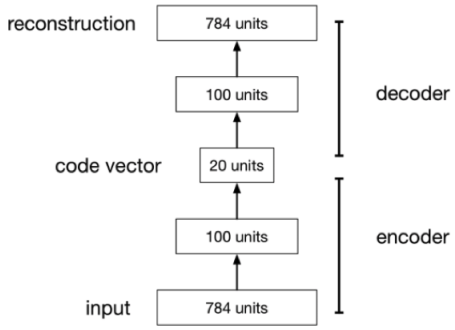


18 Variational Autoencoders

Autoencoders

- Feed-forward neural net that takes an input $\mathbf{x}$ and predict $\mathbf{x}$
- Add a *bottleneck layer* to make this task nontrivial

- Bottleneck layer has dimension much smaller than input



Autoencoder Usages

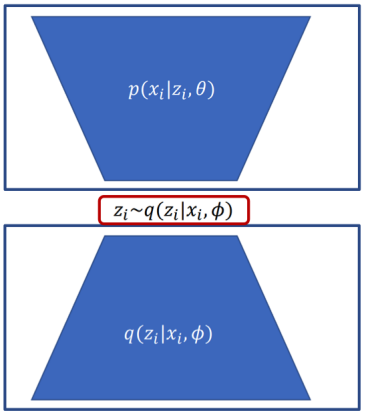
- Can map high-dimensional data into 2 dimensions for visualization
- Compression of file size (requires a VAE)
- Learn abstract fefatures in an unsupervised way
- Learn a semantically meaningful representation (e.g. interpolate between images)

Deep (Nonlinear) Autoencoders

- Learn to project the data onto a low-dimensional nonlinear maniold
- This manifold is the image of the encoder
- Not generative, so they don't define a distribution

Variational Autoencoder (VAE)

- Encoder learns the distribution of latent space given the observations
- Sampling process in the middle
- Decoder learns the generative process given the sampled latent vectors



Observation Model

- Consider training a generator network with maximum likelihood

$$p(\mathbf{x}) = \int p(\mathbf{z})p(\mathbf{x} \mid \mathbf{z})d\mathbf{z}$$

- If $\mathbf{z}$ is low-dimensional and the decoder is deterministic, then $p(\mathbf{x}) = 0$ almost everywhere
  - Because the model only generates samples over a low-dimensional sub-manifold of $\mathcal{X}$
- Can define a noisy observation model, e.g.

$$p(\mathbf{x} \mid \mathbf{z}) = \mathcal{N}(\mathbf{x}; G_{\boldsymbol{\theta}}(\mathbf{z}), \eta \mathbf{I})$$

  where $G_{\boldsymbol{\theta}}$ is the function computed by the decoder with parameters $\boldsymbol{\theta}$
- Since $G_{\boldsymbol{\theta}}(\mathbf{z})$ is very complicated, we can't find a close form to the integral
- Can maximize a lower bound on $\log p(\mathbf{x})$

Variational Inference

- Can obtain the lower bound using Jensen's inequality: for a convex function h of a random variable X,

$$\mathbb{E}[h(X)] \geq h(\mathbb{E}[X])$$

- If h is concave (i.e. $-h$ is convex),

$$\mathbb{E}[h(X)] \leq h(\mathbb{E}[X])$$

- $\log z$ is concave, so

$$\mathbb{E}[\log X] \leq \log \mathbb{E}[X]$$

- Suppose we have some distribution $q(\mathbf{z})$, can use Jensen's inequality to obtain the lower bound

$$\begin{aligned}
\log p(\mathbf{x}) &= \log \int p(\mathbf{z}) p(\mathbf{x} \mid \mathbf{z}) d\mathbf{z} \\
&= \log \int q(\mathbf{z}) \frac{p(\mathbf{z})}{q(\mathbf{z})} p(\mathbf{x} \mid \mathbf{z}) d\mathbf{z} \\
&\geq \int q(\mathbf{z}) \log \left[\frac{p(\mathbf{z})}{q(\mathbf{z})} p(\mathbf{x} \mid \mathbf{z}) \right] d\mathbf{z} \\
&= \mathbb{E}_q \left[\log \frac{p(\mathbf{z})}{q(\mathbf{z})} \right] + \mathbb{E}_q [\log p(\mathbf{x} \mid \mathbf{z})]
\end{aligned}$$

- For the term $\mathbb{E}_q[\log p(\mathbf{x} \mid \mathbf{z})]$, since we assumed a Gaussian observation model,

$$\begin{aligned}
\log p(\mathbf{x} \mid \mathbf{z}) &= \log \mathcal{N}(\mathbf{x}; G_{\boldsymbol{\theta}}(\mathbf{z}), \eta \mathbf{I}) \\
&= \log \left[\frac{1}{(2\pi\eta)^{d/2}} \exp \left(-\frac{1}{2\eta} \left\| \mathbf{x} - G_{\boldsymbol{\theta}}(\mathbf{z}) \right\|^2 \right) \right] \\
&= -\frac{1}{2\eta} \left\| \mathbf{x} - G_{\boldsymbol{\theta}}(\mathbf{z}) \right\|^2 + \text{const}
\end{aligned}$$

  which is the expected square error in reconstructing $\mathbf{x}$ from $\mathbf{z}$, called **reconstruction term**
- For the term $\mathbb{E}_q \left[\log \frac{p(\mathbf{z})}{q(\mathbf{z})} \right]$, this is the **Kullback-Leibler (KL) divergence**

$$\mathbb{E}_q \left[\log \frac{p(\mathbf{z})}{q(\mathbf{z})} \right] = -\mathrm{D}_{\mathrm{KL}}(q(\mathbf{z}) \mid\mid p(\mathbf{z}))$$

  - KL divergence measures the distance between probability distributions
  - However it does not satisffy the axioms to be a distance metric
  - Typically $p(\mathbf{z}) = \mathcal{N}(\mathbf{0}, \mathbf{I})$, hence the KL term encourages q to be close to $\mathcal{N}(\mathbf{0}, \mathbf{I})$

- We are trying to maximize the **variational lower bound**, or **variational free energy**:

$$\log p(\mathbf{x}) \geq \mathcal{F}(\boldsymbol{\theta}, q) = \mathbb{E}_q[\log p(\mathbf{x} \mid \mathbf{z})] - \mathrm{D}_{\mathrm{KL}}(q \mid\mid p)$$

- Want to choose q to make the bound as tight as possible
- The gap

$$\log p(\mathbf{x}) - \mathcal{F}(\boldsymbol{\theta}, q) = \mathrm{D}_{\mathrm{KL}}(q(\mathbf{z}) \mid\mid p(\mathbf{z} \mid \mathbf{x}))$$

  which means that we want q to be close to the posterior distribution $p(\mathbf{z} \mid \mathbf{x})$
- The reconstruction term is minimized when q is a point mass on

$$\mathbf{z}_* = \arg\min_{\mathbf{z}} \|\mathbf{x} - G_{\boldsymbol{\theta}}(\mathbf{z})\|^2$$

  but a point mass would have infinite KL divergence, so the KL term forces q to be more spread out

Reparametrization Trick

- Assign q a parametric form:

$$q(\mathbf{z}) = \mathcal{N}(\mathbf{z}; \boldsymbol{\mu}, \boldsymbol{\Sigma})$$

  where $\boldsymbol{\mu} = (\mu_1, \ldots, \mu_K)$ and $\boldsymbol{\Sigma} = \mathrm{diag}(\sigma_1^2, \ldots, \sigma_K^2)$
- Since it's hard to differentiate through an expectation, we apply the **reparametrization trick** to q:

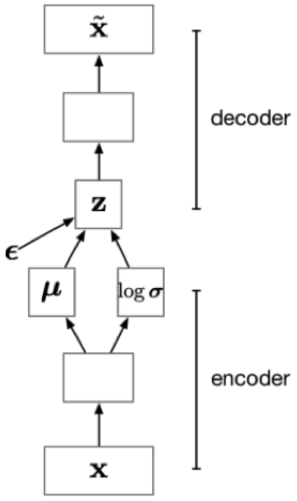
$$z_i = \mu_i + \sigma_i \epsilon_i$$

  where $\epsilon_i \sim \mathcal{N}(0, 1)$
- Hence

$$\overline{\mu_i} = \overline{z_i} \qquad \overline{\sigma_i} = \overline{z_i} \epsilon_i$$

Amortization

- For each training example:
  1. Fit q to approximate the posterior for the current $\mathbf{x}$ by doing gradient ascent on $\mathcal{F}$
  2. Update the decoder parameters $\boldsymbol{\theta}$ with gradient ascent on $\mathcal{F}$
- However this requires an expensive iterative procedure for every training example
- Idea: *amortize* the cost of inference by learning an **inference network** which predicts $(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ as a function of $\mathbf{x}$
- The outputs of the inference net are $\boldsymbol{\mu}$ and $\log \boldsymbol{\Sigma}$, where the log representation ensures $\boldsymbol{\Sigma} > 0$
  - If $\boldsymbol{\Sigma} \approx \mathbf{0}$, then the network computes $\mathbf{z}$ deterministically
  - Since the KL term encourages $\boldsymbol{\Sigma} > 0$, so $\mathbf{z}$ is noisy in general
- The notation $q(\mathbf{z} \mid \mathbf{x})$ emphasizes that q depends on $\mathbf{x}$, and it's not actually a conditional distribution
- Combine with the decoder network, we get the **variational autoencoder** since the inference net is like an encoder
- The parameters of both the encoder and decoder networks are updated using a single pass of backprop



VAE and Other Generative Models

- VAE is like an autoencoder, except it's generative (defines a distribution $p(\mathbf{x})$)
- Generation only requires 1 forward pass (unlike autoregressive models)
- Can interpolate between two vectors in the latent space

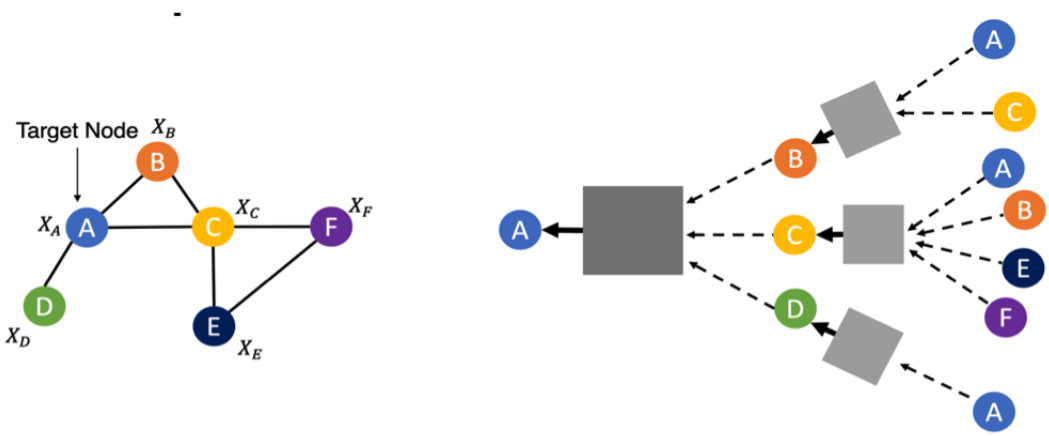
19 Graph Neural Networks

Graph

- A graph is composed of
  - Nodes (aka vertices)
  - Edges connecting a pair of nodes
- Nodes can have feature vectors

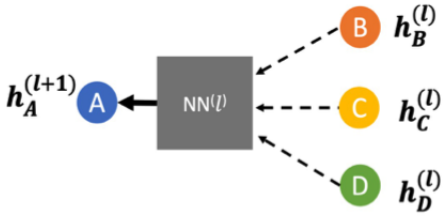
Problem Setup

- Given
  - A graph
  - Node attributes
  - Part of nodes are labelled
- Find
  - Node embeddings
- Predict
  - Labels for the remaining nodes
- Connected nodes are related/informative/similar



Forward Propagation

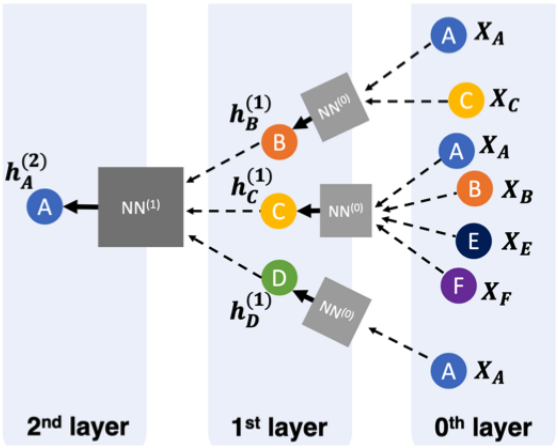
1. Aggregate messages from neighbours
   - $h_v^{(l)}$: node embedding of v at lth layer
   - $\mathcal{N}(v)$: neighbouring nodes of v
   - $\boldsymbol{f}^{(l)}$: aggregation function at lth layer (core part of GNNs)
   - $m_v^{(l)}$: message vector of v at lth layer



Neighbors of node A
$\mathcal{N}(A) = \{B, C, D\}$

$$m_A^{(l)} = \boldsymbol{f}^{(l)}\left(h_A^{(l)}, \left\{h_u^{(l)} : u \in \mathcal{N}(A)\right\}\right) = \boldsymbol{f}^{(l)}\left(h_A^{(l)}, h_B^{(l)} h_C^{(l)} h_D^{(l)}\right)$$

2. Transform messages
   - $\boldsymbol{g}^{(l)}$: transformation function at lth layer (commonly 1-layer MLP)
   - $h_A^{(l+1)} = \boldsymbol{g}^{(l)}\left(m_A^{(l)}\right)$



Training GNNs

- Semi-supervised learning: only a subset of nodes have labels

$$\min_{\Theta} \mathcal{L}(\mathbf{y}, f(\mathbf{z}_v))$$

  - $\mathbf{y}$: node label
  - $\mathcal{L}$ could be L_2 if $\mathbf{y}$ is real number, or cross-entropy if $\mathbf{y}$ is categorical
  -
  - Node embedding $\mathbf{z}_v$ is a function of input graph

- Unsupervised setting: no node label available
  - Use the graph structure as the supervision
  - "Similar" nodes have similar embeddings

$$\mathcal{L} = \sum_{z_u, z_v} \text{CE}(y_{u,v}, \text{DEC}(z_u, z_v))$$

- $y_{u,v} = 1$ when nodes u, v are similar
- CE is the cross entropy
- DEC is the decoder, such as inner product

Questions About GNN

1. Width: how many neighbours to consider?
   - If we aggregate over all neighbours, GNNs have scalability issues
   - There might be hub nodes who are connected to a huge number of nodes
   - Can only sample a fixed number of neighbours
     - Random sampling or importance sampling
2. Depth: informative neighbours could be indirectly connected with a target node
   - 2-layer or 3-layer GNNs are commonly used
   - Oversmoothing: when GNNs become deep, nodes share many neighbours, so node embeddings become indistinguishable

20 Q-Learning

Reinforcement Learning

- Agent interacts with an environment, which we treat as a black box
- Not allowed to inspect the transition probabilities, reward distributions, etc.

Markov Decision Process (MDP)

- Markov assumption: all relevant information is encapsulated in the current state
- Components of an MDP:
  - Initial state distribution $p(s_0)$
  - Transition distribution $p(s_{t+1} \mid s_t, a_t)$
  - Reward function $r(s_t, a_t)$
- Policy $\pi_{\boldsymbol{\theta}}$ is parameterized by $\boldsymbol{\theta}$
- Assume a *fully observable* environment, i.e. s_t can be observed directly

Finite and Infinite Horizon

- Finite horizon MDPs
  - Fixed number of steps T per episode
  - Maximize expected return $R = \mathbb{E}_{p(\tau)}[r(\tau)]$
- Infinite horizon MDPs
  - Cannot sum infinitely many rewards, so we need to discount them
  - **Discounted return**
$$G_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \cdots$$
  - Want to choose an action to maximize expected discounted return
  - $\gamma < 1$ is the **discount factor**
    - Small γ: myopic
    - Large γ: farsighted

Value Function

- **Value function** $V^\pi(s)$ of a state s under policy π: the expected discounted return if we start in s and follow π
$$V^\pi(s) = \mathbb{E}[G_t \mid s_t = s] = \mathbb{E}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \mid s_t = s\right]$$
- Impractical to compute the value function, but we can try to approximate/learn it
- Has a recursive formula
$$\begin{aligned} V^\pi(s) &= \mathbb{E}_{a_t, a_{t+i}, s_{t+i}}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i} \mid s_t = s\right] \\ &= \mathbb{E}_{a_t}[r_t \mid s_t = s] + \gamma \mathbb{E}_{a_t, a_{t+i}, s_{t+i}}\left[\sum_{i=0}^{\infty} \gamma^i r_{t+i+1} \mid s_t = s\right] \\ &= \mathbb{E}_{a_t}[r_t \mid s_t = s] + \gamma \mathbb{E}_{s_{t+1}}\left[V^\pi(s_{t+1}) \mid s_t = s\right] \\ &= \sum_{a,r} P^\pi(a \mid s_t) p(r \mid a, s_t) \cdot r + \gamma \sum_{a,s'} P^\pi(a \mid s_t) p(s' \mid a, s_t) \cdot V^\pi(s') \end{aligned}$$

Action-Value Function

- Can use a value function to choose action

$$\arg\max_a r(s_t, a) + \gamma \mathbb{E}_{p(s_{t+1}|s_t,a_t)}[V^\pi(s_{t+1})]$$

- However this requires taking the expectation w.r.t. the environment's dynamics, which we don't have access
- Can instead learn an **action-value function**, or **Q-function**, which is the expected return if we take action a and then follow the policy

$$Q^\pi(s, a) = \mathbb{E}[G_t \mid s_t = s, a_t = a]$$

- Relationship with value function:

$$V^\pi(s) = \sum_a \pi(a \mid s) Q^\pi(s, a)$$

- Optimal action:

$$\arg\max_a Q^\pi(s, a)$$

Bellman Equation

$$Q^\pi(s, a) = r(s, a) + \gamma \mathbb{E}_{p(s'|s,a)\pi(a'|s')}[Q^\pi(s', a')]$$

Optimal Bellman Equation

- The *optimal policy* π^* maximizes the expected discount return
- The *optimal action-value function* Q^* is the action-value function for π^*
- The *optimal Bellman equation* gives a recursive formula for Q^*:

$$Q^*(s, a) = r(s, a) + \gamma \mathbb{E}_{p(s'|s,a)} \left[\max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a \right]$$

- Can approximate Q^* by trying to solve the optimal Bellman equation

Q-Learning

- Let Q be an action-value function which hopefully approximates Q^*
- The **Bellman error** is the update to our expected return when we observe the next state s'

$$\underbrace{r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a)}_{\text{inside } \mathbb{E} \text{ in RHS of Bellman equation}} - Q(s_t, a_t)$$

- Bellman error is 0 when Q satisfies the Bellman equation
- **Q-learning** is an algorithm that repeatedly adjusts Q to minimize the Bellman error
- Each time we sample consecutive states and actions (s_t, a_t, s_{t+1}):

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

  - Inside the square brackets is Bellman error

Exploration-Exploitation Tradeoff

- Q-learning only learns about the states and actions it visits
- **Exploration-exploitation tradeoff**: the agent should sometimes pick suboptimal actions in order to visit new states and actions
- Can use **ϵ-greedy policy**
  - With probability $1 - \epsilon$, choose the optimal action according to Q
  - With probability ϵ, choose a random action

Function Approximation

- So far, we've been assuming a tabular representation of Q: 1 entry for every state/action pair
  - Impractical to store
  - Does not share structure between related states
- Solution: approximate Q using a parameterized function, e.g.
  - Linear function approximation: $Q(s, a) = w^\top \psi(s, a)$
  - Compute Q with a neural net
- Update Q using backprop

$$t \leftarrow r(s_t, a_t) + \gamma \max_a Q(s_{t+1}, a)$$
$$\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha(t - Q(s, a)) \frac{\partial Q}{\partial \boldsymbol{\theta}}$$

- Can store experience into a *replay buffer*, and perform Q-learning using stored experience
  - This does not need new experience for every SGD update