CS231n

Convolutional Neural Networks for Visual Recognition

Sungchan Yi

2019 Fall

Contents

1 Introduction

1.1 History of Computer Vision

- ...
- ImageNet: Image classification challenge
- Huge improvement on 2012 - Convolutional Neural Networks (**CNN**)

1.2 CS231n Overview

- Focus on **image classification**
- Object detection, action classification, image captioning ...
- **CNN** !

2 Image Classification

2.1 Image Classification

- Map: Input Image $\rightarrow$ Predetermined Category
- Semantic Gap - computers only see the pixel values of the image
- Many problems that cause the pixel values to change
  - Viewpoints
  - Illumination
  - Deformation (poses)
  - Occlusion (part of the object)
  - Background Clutter (object looks similar to the background)
  - Interclass Variation (objects come in different sizes and shapes)
- Image classification algorithms must be able to handle all these problems

2.2 Attempts to Classify Images

- Finding corners or edges to determine the object
  - Brittle, doesn't work very well
  - **Not scalable** to work with other objects
- **Data-Driven Approach**

    1. Collect a dataset of images and labels
    2. Use ML algorithms to train a classifier
    3. Evaluate the classifier on new images

- Data-driven approach is much more general

2.3 Nearest Neighbor

1. Memorize all data and labels
2. Predict the label of the **most similar** training image

- How to compare images? **Distance Metric** !
    - L_1 distance (Manhattan Distance)[1]

$$d_1(I_1, I_2) = \sum_{\text{pixel } p} |I_1^p - I_2^p|$$

- With N examples, training takes $\mathcal{O}(1)$, prediction takes $\mathcal{O}(N)$.
- Generally, we want prediction to be fast, but slow training time is OK
- Only looking at a **single** neighbor may cause problems (outliers, noisy data, etc.)
- Motivation for **kNN**s

2.4 k-Nearest Neighbors

- Take the **majority vote** from k closest points
- L_2 distance (Euclidean Distance)

$$d_2(I_1, I_2) = \sqrt{\sum_{\text{pixel } p} (I_1^p - I_2^p)^2}$$

- L_1 distance depends on the coordinate system L_2 doesn't matter
- Generalizing the distance metric can lead to classifying other objects such as texts

2.5 Hyperparameters

- **Hyperparameters** are choices about the algorithm that we set, rather than by learning the parameter from data
- In kNNs, the value of k and the distance metric are hyperparameters
- **Problem dependent**. Must go through trial and error to choose the best hyperparameter

[1]Dumb way to compare, but does reasonable things *sometimes* ...

- Setting Hyperparameters
  - Split the data into 3 sets. Training set, validation set, and test set
  - Train the algorithm with many different hyperparameters on the training set
  - Evaluate with validation set, choose the best hyperparameter from the results
  - Run it once on the test set, and this result goes on the paper
- Never used on image data ...
  - Slow on prediction
  - Distance metrics on pixels are not informative (Distance metric may not capture the difference between images)
  - Curse of dimensionality - data points increase exponentially as dimension increases, but there may not be enough data to cover the area densely (Need to densely cover the regions of the n-dimensional space, for the kNNs to work well)

2.6 Linear Classification

- Lego blocks - different components, as building blocks of neural networks
- **Parametric Approach**
  - Image x, weight parameters W for each pixel in the image
  - A function $f : (x, W) \rightarrow$ numbers giving class scores for each class
  - If input x is an $n \times 1$ vector and there were c classes, W must be $c \times n$ matrix
- Classifier summarizes our knowledge on the data and store it inside the parameter W
- At test time, we use the parameter W to predict
- How to come up with the right structure for f ?
- **Linear** Classifier : $f(x, W) = Wx \ (+b)$
- There may be a bias term b to show preference for some class label
- (Idea) Each row of W will work as a template for matching the input image, and the dot product of each row and the input image vector will give the **similarity** of the input data to the class
- Problems with linear classifier
  - Learning single template for each class
  - If the class has variations on how the class might appear, the classifier averages out all the variations and tries to recognize the object by a single template
  - Using deeper models to remove this restriction will lead to better accuracy

- Linear classifiers draws hyperplanes on the n-dimensional space to classify images
- If hyperplanes cannot be drawn on the space, the linear classifier may struggle (parity problem, multi-modal data)

3 Loss Functions and Optimization

3.1 Motivation

- We saw that W can act as a template for each class
- How do we choose such W ?
- To choose the best W, we should be able to **quantify** the goodness of prediction across the training data

3.2 Loss Functions

- Dataset of examples: $\{(x_i, y_i)\}_{i=1}^{N}$, where x_i is an input data, and $y_i \in \mathbb{Z}$ is a correct label for the data
- Suppose we have a prediction function f, then the **loss over the dataset** is a sum of loss over examples
$$L = \frac{1}{N}\sum_i L_i\big(f(x_i, W), y_i\big)$$
- Now we want to choose a W that **minimizes the loss function**
- **Multiclass SVM loss**
  - Scores vector $s = f(x_i, W)$
  - SVM loss for each data (Hinge loss)
$$L_i = \sum_{j \neq y_i} \max\left(0, s_j - s_{y_i} + 1\right)$$
  - As the score for the true category (s_{y_i}) increases, the loss goes down linearly, until it gets above a safety margin because now the example is correctly classified
  - Simply put - We are happy[1] if *the score for the correct label is much higher than all the other scores* by some margin[2]
  - Minimum loss is 0, maximum loss is ∞
  - Quadratic hinge loss - May be used to put more loss on scores that are totally off (Depends on how we want to weigh off different mistakes that the classifier makes)

[1]We are happy when the loss is small

[2]The constant 1 in the equation can actually be generalized

- Suppose we found a W that makes $L = 0$. But this W is not unique[3], so how do we choose such W ?
- We have only written down loss **in terms of the data**. We only told the classifier to find the W that fits the data
- But in practice, we only care about the performance on the test data

3.3 Regularization

- We add an additional **regularization** term to the loss function, which **encourages** the model to somehow pick a simpler W
- We use a regularization penalty(loss) $R(W)$, then

$$L = \frac{1}{N}\sum_i L_i\big(f(x_i, W), y_i\big) + \lambda R(W)$$

  and the λ is a hyperparameter that trades off between data loss and regularization loss
- Common regularization functions
  - L_2 regularization

$$R(W) = \sum_i \sum_j W_{ij}^2$$

  - L_1 regularization

$$R(W) = \sum_i \sum_j |W_{ij}|$$

- Anything that you do to the model that *penalizes the complexity of the model*
- L_1 regularization thinks less non-zero entries are less complex, L_2 thinks spreading numbers all across entries of W is less complex

3.4 Softmax Loss

- Multinomial Logistic Regression
- Multiclass SVM - no interpretation for each score
- Consider the scores as *unnormalized* $\log$ *probabilities of the classes*
- $s = f(x_i, W)$

$$\mathrm{P}\left(Y = k \,|\, X = x_i\right) = \frac{e^{s_k}}{\sum_j e^{s_j}} \qquad \text{(softmax function)}$$

- Now we have a probability distribution, and we want this to match the distribution that put all it's weight on the correct label

[3] $2W$ will also give 0 loss

- Loss is the $-\log$ of the probability of the true class[4] (**Cross-Entropy**)

$$L_i = -\log\left(\frac{e^{s_{y_i}}}{\sum_j e^{s_j}}\right)$$

- Minimum loss is 0, maximum loss is ∞

3.5 Optimization

- Minimize the loss function!
- Strategy #1 : Random Search (...)
- Strategy #2 : **Follow the slope** - Use the geometry of the space to find *which direction from here will decrease the loss function*
- **Gradients** ! - Directional derivatives to find out which direction we should take a step, to minimize the loss function
- Use calculus to compute analytic gradients and use them in code[5]

3.6 Gradient Descent

- The direction of the negative gradient is the fastest decrease
- Key Idea: Starting from the initial guess x_0, iteratively compute x_n by the following

$$x_{n+1} = x_n - \gamma \cdot \nabla F(x_n)$$

  and hope to converge to some x
- Constant γ is the **step size**, sometimes called the **learning rate**, which is an important hyperparameter
- This is **slow** ... Computing gradients for each iteration is too costly
- **Stochastic Gradient Descent** uses a **minibatch** (subset of training data) to estimate the true gradient

3.7 Image Features

- Feeding raw pixel values don't work well
- Compute various **feature representations** of the image, concatenate them and feed it to the classifier

[4]You can view this as the KL divergence between the target and computed distribution, maximum likelihood estimate

[5]Analytic gradients are fast, exact and error-prone, while numerical one is slow and approximate

- What is the best feature transform? [6]
- Example: Color Histogram, Histogram of Oriented Gradients (Local orientation of edges)

4 Introduction to Neural Networks

4.1 Computational Graphs

- Graph to represent any function, where each node is a function (some operation)
- Advantage - **Backpropagation**: Recursively using the chain rule in order to compute the gradient with respect to every variable in the computational graph

4.2 Backpropagation

- Suppose we have a computational graph for loss function L, and we try to calculate the gradient at some node corresponding to the operation f
- Suppose f takes x, y as inputs and outputs z, i.e. $z = f(x, y)$
- We are given the gradient for z, namely $\frac{\partial L}{\partial z}$. With this we want to calculate $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$
- Use the **chain rule** to get

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial x} \quad \text{and} \quad \frac{\partial L}{\partial y} = \frac{\partial L}{\partial z} \cdot \frac{\partial z}{\partial y}$$

- Usually operation f is simple, so we can find the local gradients, $\frac{\partial z}{\partial x}$ and $\frac{\partial z}{\partial y}$ [1] [2]
- Now $\frac{\partial L}{\partial x}$ and $\frac{\partial L}{\partial y}$ will be used **recursively** for the nodes that produced x, y
- **Sigmoid function**

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

  Useful fact is that

$$\frac{d\sigma(x)}{dx} = (1 - \sigma(x)) \cdot \sigma(x)$$

- Patterns in backward flow
  - Addition gate *distributes* gradients
  - Max gate *routes* gradients

[6]For instance, conversion from polar to Cartesian coordinates

[1]With this method, we won't have to derive an expression for the gradient for cases where the loss function is very complicated. We can just work on the level where gradients are easy to compute

[2]The expression for $\partial L / \partial z$ may be complex, but in practice, this value will be a single number received from upstream, which we multiply to the local gradient. The expressions for local gradients will be all we need, which will be very simple

  - Multiplication gate *switches* gradients

- Note that multiple gradients coming in from upstream are added together
- When x, y, z become vectors, the gradients will be **Jacobian matrices**
- But computational graphs will be very large, so it is impractical to write down each gradient formula by hand for all parameters

4.3 Neural Networks

- We talked about linear score function $f = Wx$
- 2-layer neural network $f = W_2 \cdot \max(0, W_1 x)$ ...
- Stack multiple layers with non-linear function in between[3]
- **Activation functions**

5 Convolutional Neural Networks

5.1 History

- Mark I Perceptron (1957)
- Adaline / Madaline (1960)
- Backpropagation (1986)
- Restricted Boltzmann Machines (2006)
- Convolutional Neural Networks (2012)

5.2 Structure

- Fully Connected Layer
  - $32 \times 32 \times 3$ image $\to$ 3072×1 vector x
  - 10×3072 weight matrix W
  - Calculate Wx (Each score is the dot product of row of W and x)
- **Convolutional Layer**
  - Keep the structure of the image as $32 \times 32 \times 3$
  - Take a filter, ex. $5 \times 5 \times 3$ filter w

[3]Composition of linear functions will simplify to a linear function

  - Slide over the image spatially, and compute dot products (**convolution**) $w^T x + b$ [1]
  - Filters *always* extend the full depth of the input volume
  - Result: $28 \times 28 \times 1$ **activation map**
  - Take **multiple filters** and create a set of activation maps, ex. 6 filters will give a new image of activation maps with size $28 \times 28 \times 6$
  - ConvNet is a sequence of convolution layers, interspersed with activation functions
  - **Hierarching of filters** - filters at the earlier layers represent low level features (ex. edges)
  - At the mid-level, the filters look for more complex features like corners and blobs
  - When many filters are stacked, they learn types of simple to more complex features
  - Whole structure: Conv, Non-linear (ReLU ...), Conv, Non-linear, Pool layer (once in a while to downsample the size), Conv ... and a fully connected layer at the end to compute the scores for each class

- Spatial Dimensions
  - $N \times N$ image, $F \times F$ filter
  - With stride 1, results in $(N - F + 1) \times (N - F + 1)$ activation map
  - With stride S, size will be $(N - F)/S + 1$
  - If S doesn't divide $N - F$, pad the input image with zeros
  - Common to see Conv layers with stride 1, filters of size $F \times F$ and zero-padding with $\frac{F-1}{2}$ (This will preserve size spatially)
  - Images convolved repeatedly with filters will shrink the image sizes, but shrinking too fast isn't good (won't work well - you lose information)

- Dimensions Summary - The Conv Layer
  - Accepts a volume of size $W_1 \times H_1 \times D_1$
  - With 4 hyperparameters
    * K: Number of filters
    * F: size of filter
    * S: stride
    * P: amount of zero padding
  - Produces a volume of size $W_2 \times H_2 \times D_2$, where

$$W_2 = \frac{W_1 - F + 2P}{S} + 1 \qquad H_2 = \frac{H_1 - F + 2P}{S} + 1 \qquad D_2 = K$$

  - With parameter sharing it produces $F^2 D_1$ weights per filter, for a total of $F^2 D_1 K$ weights and K biases

[1]Consider the filter weights as stretched into a long vector

- 1 number for a single convolution (dot product)
- Activation map is a sheet of neuron outputs, each connected to a small region (receptive field) in the input, all of the sharing parameters (in the filter w)
- With k filters, Conv layer consists of neurons arranged in a 3D grid, and there will be k different neurons all looking at the same region in the input volume
- Note that the fully connected layer will look at the full input volume

5.3 Pooling Layer

- Makes the representations smaller and more manageable (downsampling)
- Accepts a volume of size $W_1 \times H_1 \times D_1$
- With 2 hyperparameters
  - F: spatial extent
  - S: stride
- Produces a volume of size $W_2 \times H_2 \times D_2$ where

$$W_2 = \frac{W_1 - F}{S} + 1 \qquad H_2 = \frac{H_1 - F}{S} + 1 \qquad D_2 = D_1$$

- Doesn't do anything to the depth
- Introduces zero parameters since it computes a **fixed function** of the input, ex. Max, Average
- Uncommon to use zero-padding for pooling layers

6 Training Neural Networks I

6.1 Activation Functions

- **Sigmoid function**
  - Form:
$$\sigma(x) = \frac{1}{1 + e^{-x}}$$
  - Squashes numbers to range $[0, 1]$
  - Nice interpretation as a saturating firing rate of a neuron
  - Saturated neurons *kill* the gradients - gradient is smaller as x gets further from 0
  - Sigmoid outputs are not zero-centered - if the input to a neuron is always positive, the gradients of the weights will be either all positive or all negative, resulting in increasing or decreasing all the weights. This is inefficient

  - $\exp(x)$ is computationally expensive

- **Hyperbolic Tangent** - $\tanh(x)$
  - Squashes numbers to range $[-1, 1]$
  - Zero centered
  - But kills gradients when saturated

- **Rectified Linear Unit - ReLU**
  - Form: $f(x) = \max\{0, x\}$
  - Does not saturated in the positive region
  - Computationally efficient
  - Converges much faster than $\sigma(x)$, $\tanh(x)$ in practice
  - More biologically plausible than $\sigma(x)$
  - Not zero-centered
  - Gradient is 0 when $x < 0$ - 0 gradient will never update the weights

- **Leaky ReLU**
  - Form: $f(x) = \max\{0.01x, x\}$
  - All benefits of ReLU
  - Gradient will not die!
  - Generalization - **Parametric Rectifier (PReLU)** where $f(x) = \max\{\alpha x, x\}$ [1]

- **Exponential Linear Units - ELU**
  - Form:
  $$f(x) = \begin{cases} x & \text{if } x > 0 \\ \alpha(e^x - 1) & \text{if } x \leq 0 \end{cases}$$
  - All benefits of ReLU
  - Closer to zero mean outputs
  - Negative saturation regime compared with Leaky ReLU adds some robustness to noise
  - Computation requires $\exp(x)$

- **Maxout Neuron**
  - Form: $f(x) = \max\{w_1^T x + b_1, w_2^T x + b_2\}$
  - Does not have the basic form of dot product (non-linearity)
  - Generalizes ReLU and Leaky ReLU
  - Linear regime, does not saturate, does not die
  - But doubles the number of parameters per neuron

[1] Backpropagate into α (parameter)

6.2 Data Preprocessing

- Preprocess the data (zero centered - gradient problem, normalized, PCA, Whitening)
- Ex. subtract the mean image from all images, subtract per-channel (RGB) mean

6.3 Weight Initialization

- Initializing as small random numbers sampled from Gaussian distribution $\mathcal{N}(0, 0.01^2)$
- This works okay for small networks ... All activations become 0, weights will get small gradients and will update slowly
- What if sampled from $\mathcal{N}(0, 1^2)$ ... Almost all neurons are completely saturated, gradients will be all zero
- Xavier Initialization - sample from the Gaussian distribution and scale it by the size of input $(1/\sqrt{n})$

6.4 Batch Normalization

- To get Gaussian activations!
- Consider a batch of activations at some layer: To make each dimension unit Gaussian, apply
$$\widehat{x}^{(k)} = \frac{x^{(k)} - \mathbb{E}\left[x^{(k)}\right]}{\sqrt{\operatorname{Var}\left[x^{(k)}\right]}}$$
- Compute the empirical mean and variance independently for each dimension and normalize
- Can use $\beta^{(k)} = \mathbb{E}\left[x^{(k)}\right]$, $\gamma^{(k)} = \sqrt{\operatorname{Var}\left[x^{(k)}\right]}$ to recover the data, and use these as parameters to be learned (scale and shift)
- Usually inserted after fully connected or convolutional layers and before non-linearity
- Improves gradient flow through the network
- Allows higher learning rates
- Reduces the strong dependence on initialization
- Acts as a form of regularization, slightly reduces the need for dropout (maybe)

6.5 Babysitting the Learning Process

- Preprocess, then choose the architecture
- Double check that the loss function is reasonable
- Start off with small training data (make sure that you can overfit a small portion of the training data - sanity check)
- Start with small regularization and find the learning rate that makes the loss go down

6.6 Hyperparameter Optimization

- Cross-validation strategy
- Coarse stage - only a few epochs to get a rough idea of what parameters work
- Finer stage - longer running time for a finer search
- Monitor and visualize the loss curve and the accuracy (overfitting)
- Track the ratio of weight updates / weight magnitudes (want this to be somewhere around 0.001)

7 Training Neural Networks II

7.1 Fancier Optimization

- Problems with Stochastic Gradient Descent
  - High *condition number* - ratio of largest to smallest singular value of the Hessian matrix
  - Loss function may have a local minima or a saddle point[1]
  - The gradients come from minibatches, so they can be noisy
- **Stochastic Gradient Descent**

$$x_{t+1} = x_t - \alpha \nabla f(x_t)$$

- SGD + **Momentum**

$$v_{t+1} = \rho v_t + \nabla f(x_t)$$
$$x_{t+1} = x_t - \alpha v_{t+1}$$

  where v_t represents the 'velocity'[2][3], ρ represents 'friction'

- Gradient may be small near the saddle points / local minima, but the velocity will not be 0, letting us to keep moving and find the optimal parameter
- **AdaGrad**
  - Keep the squares of gradients

$$g_t = g_{t-1} + (\nabla f(x_t))^2$$
$$x_{t+1} = x_t - \alpha \cdot \frac{\nabla f(x_t)}{\sqrt{g_t + \epsilon}}$$

[1]Also, gradient is very small near the saddle point, which means that we will have slow progress

[2]Initial velocity can be set to 0

[3]The velocity can be interpreted as the weighted sum of recent gradients, where the weights are exponentially decaying

  - Add element-wise scaling of the gradient based on the historical sum of squares in each dimension
  - Over long time, the squared sum of gradients get larger, so the step size gets smaller. This is a feature when the search space is convex (convergence)

- **RMSProp**
  - Weights on the squared sum of gradients

$$g_t = \gamma g_{t-1} + (1-\gamma)\left(\nabla f(x_t)\right)^2$$
$$x_{t+1} = x_t - \alpha \cdot \frac{\nabla f(x_t)}{\sqrt{g_t + \epsilon}}$$

  - Momentum on the squared gradients

- **Adam** (Adaptive Moment Estimation)
  - $\approx$ RMSProp + Momentum

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla f(x_t)$$
$$v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla f(x_t)\right)^2$$

  - But since we initialize v_0, m_0 as 0, we consider them as *biased*, so we add a bias correction term to avoid the problem of taking very large steps at the beginning of training

$$\widetilde{m_t} = \frac{m_t}{1-\beta_1^t} \qquad \widetilde{v_t} = \frac{v_t}{1-\beta_2^t}$$

  - Now we have

$$x_{t+1} = x_t - \alpha \cdot \frac{\widetilde{m_t}}{\sqrt{\widetilde{v_t} + \epsilon}}$$

- All these take the **learning rate** (α) as a hyperparameter
- Learning rate may decay over time
- These algorithms are **first-order optimazation** algorithms
- **Second-Order Optimization**
  - Use gradient and Hessian to form quadratic approximation, and step to the minima of the approximation
  - Taylor expansion

$$J(\theta) \approx J(\theta_0) + (\theta - \theta_0)^T \cdot \nabla J(\theta_0) + \frac{1}{2}(\theta - \theta_0)^T \mathbf{H}(\theta - \theta_0)$$

  - Newton parameter update

$$\theta^* = \theta_0 - \mathbf{H}^{-1}\nabla J(\theta_0)$$

  - No learning rate, no hyperparameters

    - Hessian has $\mathcal{O}(n^2)$ elements, inverting takes $\mathcal{O}(n^3)$ ... Too slow for deep learning

- **Model Ensembles**
    - Optimization was about minimizing the objective function and reducing the training loss
    - But we want to increase the accuracy of the model, i.e. reduce the gap between the train and test error
    - Train many independent models, and average their results at test time[4]

7.2 Regularization

- Prevent our model from fitting the training data too well!
- **Dropout**
    - In each forward pass, **randomly** set some neurons' activation to zero
    - Dropout probability is also a hyperparameter
    - (Hand Waving) Prevents co-adaptation of features, forces the network to have a redundant representation
    - Dropout is training a large ensemble of model within a single model (each binary mask) is one model
    - At test time, multiply the output by the dropout probability
- Common pattern
    - **Training**: Add some kind of randomness $y = f_W(x, z)$
    - **Testing**: Average out randomness (sometimes approximate)

$$y = f(x) = \mathrm{E}_z \left[f(x, z) \right] = \int p(z) f(x, z) dz$$

- **Data Augmentation** - filp images, color jitter, random crops and scales, and more random transformation

7.3 Transfer Learning

- Train the model on a huge dataset, and freeze its weights
- Add/remove custom layers that you want to classify and train only for this layer (**fine tuning**)

[4]May reduce overfitting

8 Deep Learning Software

8.1 CPU & GPU

- CPU: Central Processing Unit
- GPU: Graphical Processing Unit
- CPUs have fewer cores, but each core ia much faster and capable, great at **sequential tasks**
- GPUs have more cores, but each core is much slower, great at **parallel tasks**
- GPUs are great for matrix multiplication
- NVIDIA CUDA
- OpenCL

8.2 Deep Learning Frameworks

- Easy to build big computational graphs
- Easy to compute gradients in computational graphs
- Easy to run it all efficiently on GPU

Deep learning softwares change fast thus they are omitted here.

9 CNN Architectures

9.1 Case Studies

- AlexNet
- ZFNet
- VGGNet
- GoogLeNet
- ResNet

10 Recurrent Neural Networks

10.1 Recurrent Neural Networks

- We might need many-to-many inputs/outputs for our model (variable length processing)

- RNN reads input x and predicts a vector y at some time step
- **Recurrence Formula**
$$h_t = f_W(h_{t-1}, x_t)$$
where h_t is the state at time t, x_t is the input vector at time t, and f_W is some function with parameters W
- Notice that the same function and the same set of parameters are used at every time step, thus the name *recurrent*
- Example
$$h_t = \tanh(W_{hh}h_{t-1} + W_{xh}x_t) \qquad y_t = W_{hy}h_t$$
With the output h_t, calculate y_t using weights W_{hy} to compute the prediction vector y_t
- **Truncated Backpropagation** through time - Run forward and backward through chunks of the sequence instead of the whole sequence
- Image Captioning: CNN+RNN (+ Attention: which part of the image to look at, for every time step)
- Backpropagation on RNN: we need to multiply by some part of the weight matrix, *multiple times*, which may cause gradients to either explode or vanish (gradient clipping - scale gradient if its L_2 norm is too big, or change RNN architecture)

10.2 Long Short Term Memory (LSTM)

- h_t: hidden state, c_t: cell state
$$\begin{pmatrix} i \\ f \\ o \\ g \end{pmatrix} = \begin{pmatrix} \sigma \\ \sigma \\ \sigma \\ \tanh \end{pmatrix} W \begin{pmatrix} h_{t-1} \\ x_t \end{pmatrix}$$
$$c_t = f \odot c_{t-1} + i \odot g \qquad h_t = o \odot \tanh(c_t)$$
  - f: Forget gate, whether to erase cell
  - i: Input gate, whether to write to cell
  - g: Gate gate, how much to write to cell
  - o: Output gate, how much to reveal cell
- Forget a portion of previous cell state c_{t-1}, and determine whether to write g to the cell state, and squash the cell state using $\tanh$, multiply the output to decide how much we want to reveal the cell
- Backpropagation: Element-wise multiplication by the forget gate (uninterrupted gradient flow)

- **Gated Recurrent Units** (GRU)

$$
\begin{aligned}
r_t &= \sigma(W_{xr}x_t + W_{hr}h_{t-1} + b_r) \\
z_t &= \sigma(W_{xz}x_t + W_{hz}h_{t-1} + b_z) \\
\widetilde{h}_t &= \tanh(W_{xh}x_t + W_{hh}(r_t \odot h_{t-1}) + b_h) \\
h_t &= z_t \odot h_{t-1} + (1 - z_t) \odot \widetilde{h}_t
\end{aligned}
$$

11 Detection and Segmentation

11.1 Semantic Segmentation

- For every pixel in the image, assign a category (label the pixel as grass, cat, etc.)
- *Sliding Window* approach: break image down to a small crop and categorize it (computationally expensive)
- **Fully Convolutional** approach: bunch of convolutional layers - after forward pass, scores for categories are calculated for each pixel
- This is also expensive - convolutions at the original image resolution will be very expensive
- Better approach: design the network as a bunch of convolutional layers, with **downsampling** and **upsampling** inside the network
- Downsampling the input image enables us to make the network deep and deal with smaller sizes in the hidden layers. In the last few layers, upsample the image to restore it to the original size
- *Downsampling* can be done by pooling, strided convolution
- *Upsampling* - how to fill an area of pixels with the given weight value
  - Nearest Neighbor - fill the pixels with the same value
  - Bed of Nails - set a single pixel to the value, rest are 0
  - Max unpooling - remember the position of the maximum element and set the value at that position when upsampling (Will need corresponding pairs of downsampling and upsampling layers)
  - Transpose convolution[1] - we have a filter, we multiply the given weight value to the filter and fill the output pixel region with the result. If the regions overlap, just add them

[1]Also called deconvolution, upconvolution, fractionally strided convolution, backward strided convolution

11.2 Classification + Localization

- Want to label the image as some object, and want to determine where the object is (find the boundary of that object)
- Input an image $\rightarrow$ class scores, and box coordinates. Calculate softmax loss with class scores, and calculate L_2 loss with coordinates. Add two losses to get a **multitask loss**
- Human pose estimation

11.3 Object Detection

- Start with some fixed categories
- Given an input image, when an categorized object appears in the image, we want to detect it
- Challenging since we don't know how many number of object we expect to see in one image
- *Sliding Window* approach - similar to semantic segmentation, but also expensive
- **Region Proposals** - Given an input images, gives boxes where objects might be present. Relatively fast to run - Selective Search gives 2000 regions
- **R-CNN**
  - Regions of Interest (RoI) from region proposals
  - Warped image regions are forwarded through ConvNet
  - Classify regions with SVMs and linear regression for bounding box offsets
  - Computationally expensive
  - Training is slow, takes lot of disk space
  - Inference is slow
- **Fast R-CNN**
  - Forward the entire image through the ConvNet
  - Use Conv feature map of image to find RoIs from a proposal method
  - Use RoI pooling layer and use FC layer
  - Use softmax classifier and bounding-box regressors
  - Computing region proposals dominates the computation time
- **Faster R-CNN**
  - Network itself predicts the region proposals
- **Detection without Proposals**: YOLO / SSD

11.4 Instance Segmentation

- Object Detection + Semantic Segmentation
- Detect objects and predict the boundaries
- **Mask R-CNN**

12 Visualizing and Understanding

What's going on inside CNNs? We want to gain intuition for each layers!

- Visualizing the weights learned by the layer - template matching and inner products gives us the idea of what the layer is looking for in the image
- First layer: looking for orientation, opposing colors, angles and edges
- Higher layers: able to visualize, but not very interesting (not very good intuition...)
- Last layer: feature vector for an image (before the classifier)
- Last layer: nearest neighbours in the feature space - the feature space captures the semantic content in the image
- Last layer: **dimensionality reduction** - ex. principle component analysis, **t-SNE**. Can visualize the geometry of the feature space
- Visualizing maximally activating patches - show the image patches that correspond to maximal activations
- Occlusion experiments - mask part of the image and draw a heatmap of probability at each mask location, if the probability changed a lot, that masked region of the image was important for classification
- Saliency maps - compute the gradient of (unnormalized) class score with respect to image pixels
- Intermediate features via guided backpropagation - which part of the image influence the score in a specific neuron
- Gradient ascent - fix the weights and change the pixels in the image to maximize the activation of a neuron (also use regularization term to prevent overfitting)
  - Initialize the image
  - Forward image to compute current scores
  - Backprop to get gradient of neuron value with respect to image pixels
  - Make a small update to the image

  - Multi-modality (???)

- Fooling images - image has no difference to the human eye, but adding slightly different noises to the image may cause the network to classify the image differently
  - Start from an arbitrary image
  - Pick an arbitrary class
  - Modify the image to maximize the class
  - Repeat until the network is fooled
- *Why is this visualization important?* - Response to criticism: To show that these complex models are doing something beneath, to get a nice interpretation for each layer and gain understanding (they aren't random!)
- DeepDream - amplify existing features
- Feature inversion - try to reconstruct image from the feature vector (with gradient ascent and regularization: total variation regularizer encourages spatial smoothness for the image to look more natural)
  - As we go deeper in the layer, the networks throws away the low level details, instead the network tries to keep around the semantic information (a little bit more invariant to small changes in color and texture)
- Texture synthesis - given input patch, generate a larger image with the same texture
- Neural texture synthesis - Gram matrix
- Neural style transfer - feature inversion + Gram reconstruction
- Fast neural style

13 Generative Models

13.1 Unsupervised Learning

- Supervised Learning
  - Data (x, y) : x is data, y is label
  - Goal: Learn a **function** to map $x \to y$
  - Examples: Classification, regression, object detection, semantic segmentation, image captioning, etc.
- Unsupervised Learning
  - Data x: Just data, no labels

  - Goal: Learn some underlying hidden **structure** of the data
  - Examples: Clustering, dimensionality reduction, feature learning, density estimation, etc.
  - Training data is cheap (no label)
  - Understand structure of visual world

13.2 Generative Models

- Given training data, generate new samples from same distribution
- Training data $\sim p_{data}(x)$, Generated samples $\sim p_{model}(x)$
- Want to learn $p_{model}(x)$ similar to $p_{data}(x)$
- Addresses density estimation (core problem in unsupervised learning)
  - Explicit density estimation: explicitly define and solve for $p_{model}(x)$
  - Implicit density estimation: learn model that can sample from $p_{model}(x)$ without explicitly defining it
- Why?
  - Realistic samples for artwork, super-resolution, colorization, etc.
  - Generative models of time-series data can be used for simulation and planning
  - Training generative models can also enable inference of latent representations that can be useful as general features

13.3 PixelRNN and PixelCNN

- Fully visible belief network, explicit density model
- Use chain rule to decompose likelihood of an image x into product of 1D distributions

  $$p(x) = \prod_{i=1}^{n} p(x_i \mid x_1, \dots, x_{i-1})$$

  Then maximize the likelihood of training data
- **PixelRNN**
  - Generate image pixels starting from corner
  - Dependency on previous pixels modeled using and RNN (LSTM)
  - Sequential generation - slow
- **PixelCNN**
  - Still generate image pixels starting from corner

  - Dependency on previous pixels now modeled using a CNN over context region
  - Outputing a distribution over pixel values at each location of image
  - Training: maximize likelihood of training images, softmax loss at each pixel
  - Training time is faster than PixelRNN - parallelize convolutions since context region values are known from training images
  - Generation must still proceed sequentially - slow

- Pros
  - Can explicitly compute likelihood $p(x)$
  - Explicit likelihood of training data gives good evaluation metric
  - Good samples

- Cons
  - Sequential generation is slow

13.4 Autoencoders

- Unsupervised approach for learning a lower-dimensional feature representation from unlabled training data
- Input Data $\rightarrow$ Features $\rightarrow$ Reconstructed input data

$$x \overset{Encoder}{\longrightarrow} z \overset{Decoder}{\longrightarrow} \widehat{x}$$

- Encoder, Decoder: Originally - linear + nonlinearity (sigmoid), later - Deep, fully connected layer, ReLU CNN
- z is usually smaller than x due to dimensionality reduction - we want features to capture meaningful factors of variation in data
- Train so that features can be used to reconstruct original data ("**Autoencoding**" - encoding itself)
- L_2 loss function $\|x - \widehat{x}\|^2$ (Doesn't use labels)
- After training, throw away the decoder
- Encoder can be used to initialize a **supervised** model and add a classifier at the end
- Able to use unlabeled training data to try and learn good general feature representation, use this to initialize supervised learning problem where we only have small data
- Features z capture factors of variation in training data. Can we generate new images from an autoencoder?

13.5 Variational Autoencoders (VAE)

- VAEs define intractable density function with latent z

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

  cannot optimize directly, we have to derive and optimize lower bound on likelihood instead

- Probabilistic spin on autoencoders - will let us *sample* from the model to *generate* data
- Assume training data $\{x^{(i)}\}_{i=1}^{N}$ is generated from underlying unobserved (latent) representation z
- **Intuition**: x is an image, z is latent factors used to generate x - attributes, orientation, etc.
- Sample from true prior $p_{\theta^*}(z)$, then sample from true conditional $p_{\theta^*}(x \mid z^{(i)})$
- We want to estimate the true parameters θ^* of this generative model
- How should we represent this model?
  - Choose prior $p(z)$ to be simple, such as Gaussian (Reasonable for latent attributes)
  - Conditional $p(x \mid z)$ is much more complex (need this to generate image) - represent with neural network
- How to train the model?
  - Remember the strategy for training generative models from FVBN[1] - learn model parameters to maximize likelihood of training data
- Intractability
  - Data likelihood: $p_\theta(x) = \displaystyle\int p_\theta(z) p_\theta(x \mid z)\, dz$
  - $p_\theta(z)$: Simple Gaussian prior
  - $p_\theta(x \mid z)$: Decoder neural network
  - Intractable to compute $p(x \mid z)$ for every z
  - Posterior density also intractable: $p_\theta(z \mid x) = p_\theta(x \mid z) p_\theta(z) / p_\theta(x)$
- Solution
  - In addition to decoder network modeling $p_\theta(x \mid z)$, define an additional encoder network $q_\phi(z \mid x)$ that approximates $p_\theta(z \mid x)$
  - This allows us to derive a lower bound on the data likelihood that is tractable, which we can optimize

[1]Fully Visible Belief Networks

- Since we're modeling probabilistic generation of data, encoder and decoder networks are probabilistic (Mean and diagonal covariance)
  - $x \to$ (Encoder network $q_\phi(z \mid x)$) $\mu_{z|x}$, $\Sigma_{z|x}$ - Sample z from $z \mid x \sim \mathcal{N}(\mu_{z|x}, \Sigma_{z|x})$
  - $z \to$ (Decoder network $p_\theta(x \mid z)$) $\mu_{x|z}$, $\Sigma_{x|z}$ - Sample $x \mid z$ from $x \mid z \sim \mathcal{N}(\mu_{x|z}, \Sigma_{x|z})$
- Encoder and decoder networks are also called recognition/inference and generation networks
- Data likelihood

$$\begin{aligned}
\log p_\theta\left(x^{(i)}\right) &= \mathbb{E}_{z\sim q_\phi(z|x^{(i)})}\left[\log p_\theta\left(x^{(i)}\right)\right] && \left(p_\theta\left(x^{(i)}\right) \text{ does not depend on } z\right)\\
&= \mathbb{E}_z\left[\log \frac{p_\theta(x^{(i)} \mid z)p_\theta(z)}{p_\theta(z \mid x^{(i)})}\right] && (\text{Bayes' Rule???})\\
&= \mathbb{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - \mathbb{E}_z\left[\log \frac{q_\phi\left(z \mid x^{(i)}\right)}{p_\theta(z)}\right] + \mathbb{E}_z\left[\log \frac{q_\phi\left(z \mid x^{(i)}\right)}{p_\theta\left(z \mid x^{(i)}\right)}\right]\\
&= \mathbb{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right]\\
&\qquad - D_{KL}\left(q_\phi\left(z \mid x^{(i)}\right) \parallel p_\theta(z)\right) + D_{KL}\left(q_\phi\left(z \mid x^{(i)}\right) \parallel p_\theta\left(z \mid x^{(i)}\right)\right)
\end{aligned}$$

  - $\mathbb{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right]$: Decoder network gives $p_\theta(x \mid z)$, can compute estimate of this term through sampling
  - $-D_{KL}\left(q_\phi\left(z \mid x^{(i)}\right) \parallel p_\theta(z)\right)$: This KL term (between Gaussians for encoder and z prior) has nice closed-form solution
  - $D_{KL}\left(q_\phi\left(z \mid x^{(i)}\right) \parallel p_\theta\left(z \mid x^{(i)}\right)\right)$: $p_\theta(z \mid x)$ is intractable, can't compute, but we know that KL divergence term is always ≥ 0
  - Since last term is always ≥ 0...
- **Tractable Lower Bound** which we can take gradient of and optimize (differentiable)

$$\mathcal{L}\left(x^{(i)}, \theta, \phi\right) = \mathbb{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right] - D_{KL}\left(q_\phi\left(z \mid x^{(i)}\right) \parallel p_\theta(z)\right)$$

  - $\mathbb{E}_z\left[\log p_\theta(x^{(i)} \mid z)\right]$: Trying to do a good reconstruction of input data
  - $D_{KL}\left(q_\phi\left(z \mid x^{(i)}\right) \parallel p_\theta(z)\right)$: Want this KL term to be small, which means that we want top make approximate posterior distribution close to the prior distribution
- $\therefore \log p_\theta\left(x^{(i)}\right) \geq \mathcal{L}\left(x^{(i)}, \theta, \phi\right)$ (Variational lower bound - ELBO[2])
- Training: **Maximize** lower bound

$$\theta^*, \phi^* = \arg\max_{\theta,\phi} \sum_{i=1}^{N} \mathcal{L}\left(x^{(i)}, \theta, \phi\right)$$

  - Calculate the KL term from the $z \mid x$ that came from the encoder network
  - And calculate the expectation from the $x \mid z$ which came from the output of decoder

[2] Evidence Lower Bound

- For every minibatch of input data, compute this forward pass, and backprop

- Generating Data - Use decoder network, sample z from prior
  - Diagonal prior on z: independent latent variables
  - Different dimensions of z encode interpretable factors of variation
  - Also good feature representation that can be computed using $q_\phi(z \mid x)$

- **Summary**
  - Probabilistic spin to traditional autoencoders allows generating data
  - For training, defines an intractable density, derives and optimizes a (variational) lower bound

- Pros
  - Principled approach to generative models
  - Allows inference of $q(z \mid x)$, can be useful feature representation for other tasks

- Cons
  - Maximizes lower bound of likelihood is ok, but not as good evaluations as PixelRNN/PixelCNN
  - Samples blurrier and lower quality compared to SOTA GANs

- Areas of Research
  - More flexible approximations (richer approximate posterior instead of diagonal Gaussian)
  - Incorporating structure in latent variables

13.6 Generative Adversarial Networks (GAN)

- What if we give on explicitly modeling density, and just want ability to sample?
- Take a game-theoretic approach - learn to generate from training distribution through 2 player game
- Problem: Want to sample from complex, high-dimensional training distribution
- Solution: Sample from a simple distribution and learn transformation to training distribution
- Training GANs
  - **Generator network**: Try to fool the discriminator by generating real-looking images
  - **Discriminator network**: Try to distinguish between real and fake images
  - Train jointly in **minimax game**

- Objective function:

$$\min_{\theta_g} \max_{\theta_d} \left(\mathbb{E}_{x \sim p_{data}} [\log D_{\theta_d}(x)] + \mathbb{E}_{z \sim p(z)} \left[\log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right] \right)$$

  - Discriminator (θ_d) wants to **maximize objective** such that $D(x)$ is close to 1 (real image) and $D(G(z))$ is close to 0 (fake image)
  - Generator (θ_g) wants to **minimize objective** such that $D(G(z))$ is close to 1 (discriminator is fooled into thinking generated $G(z)$ is real)

- Alternate between:

  - **Gradient ascent** on discriminator

$$\max_{\theta_d} \left(\mathbb{E}_{x \sim p_{data}} [\log D_{\theta_d}(x)] + \mathbb{E}_{z \sim p(z)} \left[\log(1 - D_{\theta_d}(G_{\theta_g}(z))) \right] \right)$$

  - **Gradient ascent** on generator[3]

$$\max_{\theta_g} \left(\mathbb{E}_{z \sim p(z)} \left[\log(D_{\theta_d}(G_{\theta_g}(z))) \right] \right)$$

    (Instead of minimizing likelihood of discriminator being correct, now maximize likelihood of discriminator being wrong)

- GAN training algorithm
- After training, use generator network to generate new images
- GANs: Convolutional Architectures (Radford et al, ICLR 2016)
- GANs: Interpretable Vector Math
- Pros
  - Beautiful, SOTA samples
- Cons
  - Trickier, more unstable to train
  - Can't solve inference queries such as $p(x)$, $p(z \mid x)$
- Areas of Research
  - Better loss functions, more stable training (Wasserstein GAN, LSGAN)
  - Conditional GANs, GANs or all kinds of applications

14 Deep Reinforcement Learning

Problems involving and **agent** interacting with an **environment**, which provides **numeric award** signals. Goal is to learn how to take actions in order to maximize reward

[3] $\log(1-x)$ doesn't work well with gradient descent - low gradient on bad samples, high gradient on good samples

14.1 Reinforcement Learning

- Setup
  - Environment gives Agent some state s_t
  - Agent takes action a_t
  - Environment gives back reward r_t, and next state s_{t+1}

14.2 Markov Decision Process

- Mathematical formulation of the RL problem
- **Markov property**: Current state completely characterizes the state of the world
- Defined by $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$
  - $\mathcal{S}$: set of possible states
  - $\mathcal{A}$: set of possible actions
  - $\mathcal{R}$: distribution of reward given (state, action) pair
  - $\mathbb{P}$: transition probability given (state, action) pair
  - γ: discount factor (how much we value rewards coming up soon or later on)
- Process
  - At $t = 0$, environment samples initial state $s_0 \sim p(s_0)$
  - Repeat until Done:
    * Agent selects action a_t
    * Environment samples reward $r_t \sim R(\cdot \mid s_t, a_t)$
    * Environment samples next state $s_{t+1} \sim P(\cdot \mid s_t, a_t)$
    * Agent receives reward r_t and next state s_{t+1}
  - A policy π is a function from $\mathcal{S}$ to $\mathcal{A}$ that specifies what action to take in each state
- **Objective**: Find policy π^* that maximizes cumulative discounted reward - $\sum\limits_{t \geq 0} \gamma^t r_t$
  - How do we handle the randomness (initial state, transition probability ...)
  - Maximize the **expected sum of rewards**

$$\pi^* = \arg\max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \mid \pi\right] \text{ w.r.t } s_0 \sim p(s_0), a_t \sim \pi(\cdot \mid s_t), s_{t+1} \sim p(\cdot \mid s_t, a_t)$$

14.3 Value Function and Q-Value Function

- Following a policy produces sample trajectories $s_0, a_0, r_0, s_1, a_1, r_1, \ldots$
- The **value function** at state s is the expected cumulative reward from following the policy from state s (Measures how good a state is)

$$V^{\pi}(s) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, \pi\right]$$

- The Q-**value function** at state s and action a is the expected cumulative reward from taking action a in state s and then following the policy (Measures how good a state-action pair is)

$$Q^{\pi}(s, a) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi\right]$$

- The optimal Q-value function Q^* is the maximum expected cumulative reward achievable from a given (state, action) pair

$$Q^*(s, a) = \max_{\pi} \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \mid s_0 = s, a_0 = a, \pi\right]$$

- Q^* satisfies the following **Bellman Equation**

$$Q^*(s, a) = \mathbb{E}_{s' \sim \mathcal{E}}\left[r + \gamma \max_{a'} Q^*(s', a') \mid s, a\right]$$

  Intuition: If the optimal state-action values for the next time-step $Q^*(s', a')$ are known, then the optimal strategy is to take the action that maximizes the expected value of $r + \gamma Q^*(s', a')$

- The optimal policy π^* corresponds to taking the best action in any state as specified by Q^*

14.4 Solving for the Optimal Policy: Q-Learning

- **Value iteration** algorithm: Use the Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s', a') \mid s, a\right]$$

  Q_i will converge to Q^* as $i \to \infty$

- Problem: not scalable. Must compute $Q(s, a)$ for every state-action pair. If state is pixels, it is computationally infeasible to compute this for the entire state space
- Solution: Use a function approximator to estimate $Q(s, a)$
- Q-learning: Use a function approximator to estimate the action-value function[1]

$$Q(s, a; \theta) \approx Q^*(s, a)$$

  Deep Q-learning: If the function approximator is a deep neural network

[1] θ is the function parameters (weights)

- Want to find a Q-function that satisfies the Bellman equation. Thus loss function will measure the error from the Bellman equation
- **Forward Pass** - Loss function $L_i(\theta_i)$

$$L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot)}\left[(y_i - Q(s,a;\theta_i))^2\right] \quad \text{where } y_i = \mathbb{E}_{s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) \mid s,a\right]$$

- **Backward Pass** - Gradient update (w.r.t Q-function parameters θ)

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s,a\sim\rho(\cdot);s'\sim\mathcal{E}}\left[r + \gamma \max_{a'} Q(s',a';\theta_{i-1}) - Q(s,a;\theta_i)\nabla_{\theta_i} Q(s,a;\theta_i)\right]$$

- Iteratively try to make the Q-value close to the target value y_i it should have, if Q-function corresponds to optimal Q^* (and optimal policy π^*)
- Learning from batches of consecutive samples is problematic
  - Samples are correlated
  - Current Q-network parameters determine next training samples, which can lead to bad feedback loops
- **Experience Replay**
  - Continually update a **replay memory** table of transitions (s_t, a_t, r_t, s_{t+1}) as game (experience) episodes are played
  - Train Q-network on random minibatches of transitions from the replay memory, instead of consecutive samples
  - Each transition can also contribute to multiple weight updates (greater data efficiency)
- Deep Q-Learning with Experience Replay
  - Initialize replay memory, Q-network
  - Play M episodes (full games)
  - Initialize state (starting game screen pixels) at the beginning of each episode
  - For each timestep t of the game
    - ∗ With small probability, select a random action (explore), otherwise select greedy action from current policy
    - ∗ Take the action a_t and observe the reward r_t and next state s_{t+1}
    - ∗ Store transition in replay memory
    - ∗ Experience Replay: Sample a random minibatch of transitions from replay memory and perform a gradient descent step

14.5 Policy Gradients

- Problem with Q-learning: Q-function can be very complicated
- Can we learn a policy directly?
- Define a class of parametrized policies $\Pi = \{\pi_\theta, \theta \in \mathbb{R}^m\}$
- For each policy, define its value $J(\theta) = \mathbb{E}\left[\sum_{t \geq 0} \gamma^t r_t \mid \pi_\theta\right]$
- We want to find the optimal policy $\theta^* = \arg\max_\theta J(\theta)$
- Apply gradient ascent on policy parameters

14.6 Reinforce Algorithm

- Expected reward: $J(\theta) = \int_\tau r(\tau)p(\tau;\theta)\,d\tau$, where $r(\tau)$ is the reward of a trajectory τ
- $\nabla_\theta J(\theta) = \int_\tau r(\tau)\nabla_\theta p(\tau;\theta)\,d\tau$ (Intractable; Gradient of an expectation is problematic when p depends on θ)
- Trick: $\nabla_\theta p(\tau;\theta) = p(\tau;\theta)\dfrac{\nabla_\theta p(\tau;\theta)}{p(\tau;\theta)} = p(\tau;\theta)\nabla_\theta \log p(\tau;\theta)$
- Then $\nabla_\theta J(\theta) = \int_\tau (r(\tau)\nabla_\theta \log p(\tau;\theta))p(\tau;\theta)\,d\tau = \mathbb{E}_{\tau\sim p(\tau;\theta)}\left[r(\tau)\nabla_\theta \log p(\tau;\theta)\right]$
- This can be estimated with Monte Carlo sampling
- Can we compute those quantities without knowing the transition probabilities?
- $p(\tau;\theta) = \prod_{t\geq 0} p(s_{t+1} \mid s_t, a_t)\pi_\theta(a_t \mid s_t)$
- $\log p(\tau;\theta) = \sum_{t\geq 0} (\log p(s_{t+1} \mid s_t, a_t) + \log \pi_\theta(a_t \mid s_t))$
- Differentiation gives $\nabla_\theta \log p(\tau;\theta) = \sum_{t\geq 0} \nabla_\theta \log \pi_\theta(a_t \mid s_t)$ (Does not depend on transition probabilities)
- Therefore when sampling a trajectory τ, we estimate $J(\theta)$ with

$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0} r(\tau)\nabla_\theta \log \pi_\theta(a_t \mid s_t)$$

- Interpretation
  - If $r(\tau)$ is high, push up the probabilities of the actions seen
  - If $r(\tau)$ is low, push down the probabilities of the actions seen

- This gradient estimator $\nabla_\theta J(\theta)$ sufferes from high variance because credit assignment is really hard
- Variance Reduction
  - Push up the probabilities of an action seen, only by the cumulative future reward from that state
$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0}\left(\sum_{t'\geq t} r_{t'}\right)\nabla_\theta \log \pi_\theta(a_t \mid s_t)$$
  - Use discount factor γ to ignore delayed effects
$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0}\left(\sum_{t'\geq t} \gamma^{t'-t} r_{t'}\right)\nabla_\theta \log \pi_\theta(a_t \mid s_t)$$
- Variance Reduction: *Baseline*
  - Important: Whether a reward is better or worse than what you expect to get
  - Idea: Introduce a baseline function dependent on the state s_t
$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0}\left(\sum_{t'\geq t} \gamma^{t'-t} r_{t'} - b(s_t)\right)\nabla_\theta \log \pi_\theta(a_t \mid s_t)$$
- Baseline Function
  - Simple: Constant moving average of rewards experienced so far from all trajectories
  - Better: Push up the probability of an action from a state, if this action was better than the **expected value** of what we should get from that state
  - We are happy with an action a_t in a state s_t if $Q^\pi(s_t, a_t) - V^\pi(s_t)$ is large
  - Now we have
$$\nabla_\theta J(\theta) \approx \sum_{t\geq 0}\left(Q^{\pi_\theta}(s_t, a_t) - V^{\pi_\theta}(s_t)\right)\nabla_\theta \log \pi_\theta(a_t \mid s_t)$$
- **Actor-Critic** Algorithm
  - But we don't know Q, V, can we learn them?
  - Combine policy gradients and Q-learning by training both an actor (the policy) and a critic (the Q-function)
  - The actor decides with action to take, and the critic tells the actor how good its action was and how it should adjust
  - Also alleviates the task of the critic as it only has to learn the values of (state, action) pairs generated by the policy
  - Can also incorporate Q-learning tricks
  - **Advantage function**: how much an action was better than expected
$$A^\pi(s,a) = Q^\pi(s,a) - V^\pi(s)$$

15 Efficient Methods and Hardware for Deep Learning

Deep learning is changing our lives, the models are getting bigger. But there are challenges,

- *Model size*: It is hard to distribute large models through over-the-air update
- *Speed*: Long training time limits ML researcher's productivity
- *Energy efficiency*: Takes a lot of electricity and hardware (larger model $\rightarrow$ more memory reference $\rightarrow$ more energy)

We want to improve the efficiency of deep learning by **algorithm-hardware co-design**

15.1 Algorithms for Efficient Inference

- **Pruning**
  - Remove some of the neurons, and try to reach the same accuracy
  - Prune away 80% of the parameters, but with retraining the leftover parameters, accuracy is recovered to 100%
  - Iterative pruning and retraining will let you prune away up to 90% without loss of accuracy
- **Weight Sharing**
  - Not all weights have to be exact - too accurate numbers lead to overfitting
  - Trained quantization - use centroids
  - Discrete weights
  - Use Huffman coding to optimize the number of bits used to represent the weights
- **Quantization**
  - Train with normal floats, and gather the statistics for weight and activation such as mininum, maximum
  - Choose the proper radix point position to decrease the number of bits used
- **Low Rank Approximation**
  - For a convolution layer, it can be broken down to 2 convolution layers
  - Also works for fully-connected layers by using SVD
- **Binary / Ternary Net**
  - Use only $\pm 1, 0$ to represent numbers
  - For training, use the full precision weight, but for inference we only need 3 weights

- **Winograd Transformation**
  - Transform the input feature map to another feature map (easily transformable)
  - Transform the filter map to a tensor and calculate the product as a pointwise multiplication
  - Transform the output product back to a convolution (inverse transform)

15.2 Hardware for Efficient Inference

Common goal: *Minimize the memory access*

- Google TPU: 8 bit integers
- EIE: the first DNN accelerator for sparse, compressed model (save memory bandwidth)
  - Take advantage of sparse weight and activation - ignore multiplication of 0
  - Share weights
  - Encoded weight (relative index) used in matrix multiplication

15.3 Algorithms for Efficient Training

- **Parallelization**
  - Frequency and single threaded performance is getting plateaued, but the number of cores is increasing
  - Take advantage of increasing cores $\rightarrow$ parallelization
  - *Data Parallelism*: Feed multiple data into the network (but limited by batch size)[1]
  - *Model Parallelism*: Split the model over multiple processors - by layer, conv layers by map region, FC layers by output activation
  - *Hyperparameter Parallelism*: Try many alternative networks in parallel
- **Mixed Precision with FP16 and FP32**
  - FP16 storage/input, full precision product
  - Summation with FP32 accumulator, convert to FP32 result for weight updates
- **Model Distillation**
  - Multiple large senior neural networks teach a student model neural network (smaller model size)
  - Training on softened result label results in much faster convergence on smaller data
- **Dense-Sparse-Dense Training** (DSD)

[1]If the batch is too large, SGD becomes gradient descent, which is impractical.

  - Prune the network, train it, then recover the weights and train again
  - Learn the trunk first, then learn the leaves
  - DSD produces same model architecture but can find better optimization solution, arrives at better local minima, and achieves higher prediction accuracy across a wide range of deep neural networks on CNNs/RNNs/LSTMs

15.4 Hardware for Efficient Training

- GPUs can be used for training
- Computation, memory, communication have to be balanced in order to achieve good performance
- Volta GPU, Tensor Core, Google Cloud TPU

16 Adversarial Examples and Adversarial Training

16.1 Adversarial Examples

- **Adversarial example** is an example that has been carefully computed to be misclassified
- In a lot of cases, we can add noise to the original image that is indistinguishable from the human eye, but the classification result will change
- This noise that we add can be carefully constructed so that we can attack the CNN into misclassification
- These adversarial examples came up when people tried to fool spam filters in 2004
- In 2013, researchers found that neural networks could also be fooled
- Many models can be fooled, such as linear models (logistic regression, softmax regression, SVMs, decision trees, nearest neighbors)

16.2 Reasons Behind Adversarial Examples

- Why do these adversarial examples occur?
- First thoughts - Are they from *overfitting*?
  - A complicated deep neural network will learn to fit the training set, but its behavior on the test set is undefined. Then it can make *random* mistakes that attackers can exploit
  - The model usually has too much parameters[1], it can assign some probability mass on some regions on the input space randomly. In these random regions, adversarial examples may occur

[1]more parameters than it actually needs to represent the training task

  - If overfitting is the actual cause, then each adversarial example is more or less the result of bad luck. If we fit the model again or use a slightly different model, we would expect the model to make different random mistakes (on the inputs that are off the training set)
  - But it turned out that this was not random
  - Many different models would classify the same adversarial examples and they would assign the same class to them
  - We took the difference between an original example and an adversarial one, which gave us a direction in the input space, and we could add that same direction vector to any clean example to almost always get an adversarial example

- Could this be *underfitting*?
  - Linear models[2] may not be able to capture the features of the input space completely (ex. the results of XOR cannot be captured with a hyperplane)
  - Linear models also have high confidence on inputs that are very far from the decision boundary, even though inputs from that region were never given in training time
  - But are deep neural nets actually linear?
  - It turns out that modern neural nets are *piecewise linear* or (ReLU, Maxout, Sigmoid[3], LSTM[4])
  - Linearity on *the mapping from the input of the model to the output of the model.*[5]
  - Very small changes to many pixels can result in very far travels in the input vector space
  - You can create changes that are almost imperceptible but actually move the input data really far and get a large dot product with the coefficients of the linear function that the model represents
  - When generating adversarial examples, we want to create perturbation, but not change the class. We use the maxnorm to constrain the perturbation

    $$J(\widetilde{x}, \theta) \approx J(x, \theta) + (\widetilde{x} - x)^T \nabla_x J(x)$$

    We want to maximize $J(x, \theta) + (\widetilde{x} - x)^T \nabla_x J(x)$ subject to $\|\widetilde{x} - x\|_\infty \leq \epsilon$.
  - No pixel can change by more than ϵ, but the L_2 norm can get really big, but you can't concentrate all the changes for that L_2 norm to erase parts of the original input

[2]Or models that are too simple

[3]The problem of sigmoid is that input data has to be carefully tuned so that the input will be near 0 - or we get small gradients - and the sigmoid function is approximately linear near 0

[4]Uses *addition* between time steps to accumulate and remember information over time. Then interaction between a very distant time step in the past and the present will be highly linear

[5]The mapping from the parameters to the output is obviously non-linear, since weight matrices are multiplied together. This is what makes neural network training so difficult

  - *Fast Gradient Sign Method*: Take the gradient of the cost that we used to train the network (w.r.t the input) and take the sign of that gradient. The sign will enforce the maxnorm constraint
$$\widetilde{x} = x + \epsilon \cdot \text{sign}(\nabla_x J(x))$$
  - Base on this linearity hypothesis, the researchers found out that there were *adversarial subspaces* in the input space
  - Originally, they thought that adversarial examples exist nearly everywhere[6], since they could generate an adversarial example corresponding to any clean example
  - But further analysis revealed that every real example is actually near on of those boundaries, that if you cross it, you enter the adversarial subspace
  - Once in the adversarial subspace, all the points nearby will also be adversarial examples
  - This poses security threats, because it suggests that you only need to get the *direction* right, not the exact coordinate in the input space

- *Adversarial examples are not noise*
  - Noise has very little effect on the classification decision compared to adversarial examples
  - In high dimensional spaces, when you choose some reference vector, and choose a random vector, these two vectors will have zero dot product, on average
  - Check Talyor series approximation - random vectors have almost no effect on the cost, but adversarial examples are chose to maximize the effect

- Estimating the dimension of the adversarial subspace
  - The dimensions tell you how likely you are to find an adversarial example by generating random noise
  - If most of the directions are adversarial, then random directions would end up being adversarial most of the time
  - *Different models will often misclassify the same adversarial examples*
  - The larger the dimension is, it is more likely that the two adversarial subspaces intersect - you can transfer the same adversarial examples from one model to another

- Modern machine learning algorithms are wrong almost everywhere, they work well only on naturally occurring inputs

- Quadratic Models
  - Shallow RBF networks can resist adversarial perturbations very well
  - Adding perturbations to the input image (to try and fool the network) using the gradients of this classifier, the image will actually change and it will be perceptible to the human eye

[6]Just like rational numbers existing inside real numbers

  - But the problem is that this shallow model has low accuracy, so if you try to make it deeper, it gets extremely difficult to train, because of the gradients are near zero throughout RN

- Adversarial examples can generalize from one dataset to another and one model to another
  - When we train networks, we want them to generalize, independent of the training data we choose
  - So in training, the network can learn similar weights, even though the training data is different
  - This makes them vulnerable to similar adversarial examples
  - If you want to attack some target model, you can gather your own data, or look at the input/outputs of that target model, and train your own network to generate adversarial examples and attack the target model (You don't even have to know the parameters, algorithms, architecture) (Highly transferable)
  - An adversarial example that can fool all the models in an ensemble is highly likely to fool another model

- Studying adversarial examples could tell us how to significantly improve current machine learning algorithms (deal with ambiguity or unexpected inputs)

16.3 Adversarial Training

- How do we defend against these attacks?
- Using a generative model is not sufficient to solve the problem
- Training on adversarial examples might work, for example, training on adversarial example and testing on clean examples gave higher accuracy, meaning that the adversarial examples are a good regularizer
- If the model is overfitting, it can make the model overfit less, but if the model is underfitting, it will make the model underfit worse
- Other models like SVM, linear regression, kNNs will not benefit much from adversarial training, because these are always going to be *linear*
- Deep neural nets, on the other hand, they have a clear path to resisting adversarial examples, because they can be trained to be non-linear
- *Virtual Adversarial Training*: You can train without labels!
- Suppose during training the model classified input x as class A or B. We add adversarial perturbation intended to change the guess of $\widetilde{x}$, but constrain the network so that the output of $\widetilde{x}$ should match the old guess (class A or B)

- Enables semi-supervised learning where you can learn from both unlabeled and labeled examples
- Applying model-based optimization (after solving the adversarial example problem) will give us new inventions by finding input that maximizes model's predicted performance and build something we wish we had

16.4 Summary on Adversarial Examples

- Attacking is extremely easy, but defending is difficult
- Adversarial training provides regularization and semi-supervised learning
- The out-of-domain input problem is a bottleneck for model-based optimization generally