CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine

Discussion 1

This discussion worksheet/note contains information from future lectures.

Welcome to CS 182/282A - we're excited to have you here and have some *deep* conversations with you! This discussion will cover some statistics review.

1 Class Logistics

Welcome to the first discussion!

- The goal of the sections/discussions is to provide useful supplemental information to the main lecture
- There will be a mix of practical skills discussions and theoretical discussion

List of Discussion Schedules is available on Piazza. If you have requests on topics we should include in any future discussions, please let us know.

More importantly, please familiarize yourselves with class logistics available on our class website.

Problem 0: Class Logistics

Read through the syllabus on the class website, and answer the following questions:

1. What times will lectures happen?
2. When are the midterms?
3. How much slip days will you be given?
4. Can you use slip days for the final project?

2 Machine Learning Overview

2.1 Formulating Learning Problems

In this course, we will discuss 3 main types of learning problems:

- Supervised Learning
- Unsupervised Learning
- Reinforcement Learning

In supervised learning, you are given a dataset $\mathcal{D} = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ containing input vectors and labels, and attempt to learn $f_\theta(\cdot)$ such that $f_\theta(x)$ approximates the true label y.

In unsupervised learning, your dataset is unlabeled, and $\mathcal{D} = \{x_1, \ldots, x_n\}$, and you attempt to learn properties of the underlying distribution of $\mathcal{D}$.

In reinforcement learning, you do not have a fixed dataset, but instead interact with an environment to gather data. From these interactions, our goal is to learn a policy $\pi(a|s)$ that maps states s_t, to actions a_t, in order to maximize the sum of rewards $\sum_t r(s_t, a_t)$ over a time horizon.

2.2 Solving Machine Learning Problems

To solve a machine learning problem, you must first define three "parameters".

1. Pick a model class (for example, do you want to use logistic regression or do you want to use a deep neural network?)
2. Pick a loss function (how do you want to determine the "badness" of your model performance?)
3. Pick your optimizer (how are you going to optimize your model parameters θ to minimize the loss?)

Then, you typically run this on a big CPU or GPU.

2.3 Dataset Splits During Training

In the case when hyper-parameter tuning is possible (e.g., learning rate of deep nets), in addition to training and test sets, you should hold out a validation set. The following policies should be taken when using training/validation/test sets:

- Only train your model on the training set, but not the validation set and test set.
- You should never tune your hyper-parameters on your test set or choose the best model based on the performance on the test set.
- The test set should only be run once after you have finalized your model, regardless of whether you use cross-validation or a single training-validation split. You should hold out your test set until you have finalized your model.
- You should use a new test set when you train a new model.

Problem 1: Validation Potpourri

1. Why should you never tune your hyperparameters on your test set?
2. What should your validation set be used for?
3. Describe a general ML workflow with datasets

3 Statistics Review

3.1 Probability Review

Definition 1 (Dataset)**.** *A dataset $\mathcal{D}$ of size n is composed of n individual examples $\{(x_1, y_1), \ldots, (x_n, y_n)\}$, where $x_i \in \mathbb{R}^d$ represents the ith input feature and y_i represents the ith label. Datasets without the label y_i are called unlabeled datasets, and datasets with these labels are called labeled datasets.*

In general, each example could represent any data type: scalar values, images, text, audio waves and more.

Definition 2 (Joint Distribution)**.** *The joint distribution of two random variables A and B is the probability of both events co-occurring, and is written as $\mathbb{P}(A, B)$.*

Suppose we would like to model the probability distribution of our data. This will be a model of the *joint distribution* of our data, which is given by

$$\mathbb{P}(x_1, \ldots, x_n) \tag{1}$$

Definition 3 (Conditional Probability)**.** *The conditional probability of two random variables A and B is the probability of one occurring* given that *the other has occurred. The probability that A has occurred given that B has occurred is denoted $\mathbb{P}(A|B)$*

Definition 4 (Independence)**.** *If A and B are independent random variables, and their probabilities are $\mathbb{P}(A)$ and $\mathbb{P}(B)$, then their joint probability is $\mathbb{P}(A, B) = \mathbb{P}(A) \times \mathbb{P}(B)$. In other words, A and B are independent iff $\mathbb{P}(A) = \mathbb{P}(A|B)$.*

We often assume that datasets consist of *independent, identically distributed (i.i.d.)* samples. Notice what this does to the joint distribution of our data from Eq 1.

$$\mathbb{P}(x_1, \ldots, x_n) = \prod_{i=1}^{n} \mathbb{P}(x_i) \tag{2}$$

Finally, we have the identity

$$\mathbb{P}(A, B) = \mathbb{P}(A|B)\mathbb{P}(B) = \mathbb{P}(B|A)\mathbb{P}(A) \tag{3}$$

Dividing by $\mathbb{P}(B)$ then gives us Bayes' Theorem.

$$\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)} \tag{4}$$

Problem 2: Do I Have a Flu?

Let $\mathbb{P}(H)$ be the probability you have a headache, and $\mathbb{P}(F)$ be the probability you have a flu. Calculate $\mathbb{P}(F)$, $\mathbb{P}(H)$, $\mathbb{P}(H|F)$. Then, calculate $\mathbb{P}(F|H)$ using Bayes' Theorem, given the following data:

Headache	Flu
N	N
Y	N
N	N
Y	Y
Y	Y
N	Y

3.2 Estimators

In statistics, we often observe $X \sim P_\theta$ where P_θ is a class of probability distribution parameterized by θ. Here, X is the data and observed, and θ is a parameter and unobserved. Then, the goal of estimation is the following:

$$\text{We observe } X \sim P_\theta \text{ and estimate the value of some estimand } g(\theta)$$

Definition 5 (Statistic)**.** *A statistic is any function $T(X)$ of the observed data X.*

Definition 6 (Estimator)**.** *Estimator $f_\theta(X)$ are rules to calculate an estimate of some function of observed data. In other words, an estimator is any statistic meant to guess an estimand $g(\theta)$. We also often use the "hat" notation, $\hat{Y}$ to denote an estimator.*

For example, a common estimator of the population mean is the sample mean defined by: $\bar{X} = \frac{1}{N}\sum_{i=1}^{n} X_i$.

Definition 7 (Bias and Variance of Estimator)**.** *Bias of an estimator is a measure of how much does the expected value of the estimator differ from the true distribution, and for a particular test input x, the bias can be formulated as $Bias(f_\theta(x)) = \mathbb{E}_{y\sim p(y|x)}[f_\theta(x) - y]$. Variance of an estimator is a measure of how much the estimator differs from the expected value of the estimator on average, and can be formulated as $Var(f_\theta(x)) = \mathbb{E}_{y\sim p(y|x)}[(f_\theta(x) - \mathbb{E}[f_\theta(x)])^2]$.*

Specifically, a **unbiased** estimator is one where $\mathbb{E}_{y\sim p(y|x)}[f_\theta(x)] = y$ (using the "hat" notation, we can equivalently write $\mathbb{E}[\hat{y}] = y$). The best estimator, thus, has low bias, and low variance. So why don't we always use an unbiased estimator? Sometimes, we might want to introduce a little bit of bias if it significantly decreases the variance. We will see more of this and ask you to derive the **Bias-Variance Tradeoff** in the future.

4 Function Approximation & Risk Functions

There is a lot of hype surrounding deep neural networks, but at their core they are just ways of learning functions. For example, in the case of classification, we try to learn $\mathbb{P}(y|x)$, that is the probability our true label is some class y given input features x. In the case of regression, it's a similar continuous response variable. In the case of generative models, we are trying to learn to approximate a whole distribution. In all of the cases, we are trying to find an estimator $f_\theta(x)$ of a true distribution y.

To find $f_\theta(x)$, we must adjust the weights and biases in the network, often called the **parameters** θ of the network, in order to minimize the *distance* between the estimated distribution $f_\theta(x)$ and the true distribution y.

But how do we define these distance metrics? It turns out we can use **Risk function** to evaluate how well an estimator performs.

4.1 Loss Functions & Risk Functions

Definition 8 (Loss Function)**.** *Loss function $\mathcal{L}(x, y, \theta)$ measures the "badness" of an estimator, and is often measured in terms of some distance between the estimate and true estimator.*

For example, the zero-one loss is $\sum_{i=1}^{n} \delta(f_\theta(x_i) \neq y_i)$ where we add one if the estimate is off, and add zero is the estimate is correct.

Problem 3: Derivative of Sigmoid

Sigmoid function is a popular activation function in neural networks (we will learn more about what this means in due course). Let us denote the sigmoid function as

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

Calculate the partial derivative of the sigmoid function with respect to x in terms of $\sigma(x)$.

Problem 4: Derivative of Softmax (Challenge)

Recall the softmax function, defined by

$$p_i = \frac{e^{f_i(x)}}{\sum_{j=1}^{n} e^{f_j(x)}}$$

Softmax can be thought of as a multi-class extension to sigmoid function, and its derivative is often used for optimization. Calculate the partial derivative of the softmax function with respect to $f_k(x)$ for each k.

Definition 9 (Risk Function)**.** *The risk function is the expected loss (known as the risk), measured as a function of the parameter θ, so*

$$R(\theta; f(\cdot)) = \mathbb{E}_{x \sim p(x), y \sim p(y|x)}[\mathcal{L}(x, y, \theta)]$$

For example, if $\mathcal{L}(x, y, \theta) = (y - f_\theta(x))^2$ (squared error loss), then $R(\theta; f(\cdot)) = \mathbb{E}_{x \sim p(x), y \sim p(y|x)}[(y - f_\theta(x))^2]$, also known as the **mean squared error** (or **MSE**). This is the expected squared deviation of the estimator from the true distribution (over the true distribution).

That said, we cannot directly optimize this objective (i.e., minimize the risk), since we do not have access to the true distribution, so we cannot sample $x \sim p(x)$ and we only have the dataset $\mathcal{D}$. Instead, we use **empirical risk minimization** where we replace the true distribution by the **empirical distribution** from $\mathcal{D}$.

Definition 10 (Empirical Risk)**.** *The empirical risk is the risk evaluated on* ***samples*** *from the true distribution, and approximates the true risk. It is given by:*

$$\frac{1}{n}\sum_{i=1}^{n}\mathcal{L}(x_i, y_i, \theta)$$

Supervised learning is (usually) empirical risk minimization, and we must ask: is this the same as true risk minimization? To answer this question, we will analyze the bias-variance tradeoff in next week's discussion section.

5 Summary

- We discuss three main types of ML problems: supervised, unsupervised and reinforcement learning.
- Solving ML problems requires us to pick a model class, loss function and an optimizer.
- Recall the Bayes Theorem,
$$\mathbb{P}(A|B) = \frac{\mathbb{P}(B|A)\mathbb{P}(A)}{\mathbb{P}(B)}$$
- Recall that an estimator are rules to calculate an estimate of some function of the observed data, and will often be denoted by $f_\theta(X)$ where X is the data and θ are parameters
- Loss functions measure the "badness" of an estimator, and the risk is the expected loss.
- Divide your data into training, validation and test sets. Use training set to train your model, validation to tune your hyperparameters and test set to calculate the final accuracy.

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine **Discussion 2**

This discussion will first talk about the bias-variance tradeoff and go through an example to illustrate how regularization can affect the bias and variance. We will then go over a review of notation in vector/matrix calculus that we will need to understand backpropagation, and then finally review several optimization algorithms covered in lecture.

1 Bias-Variance Tradeoff

1.1 Intuitive Understanding of Bias-Variance Tradeoff

First, recall the definitions of bias and variance from last discussion,

Definition 1 (Bias of an Estimator)**.** *The bias of an estimator is a measure of how much does the expected value of the estimator differ from the true target. Suppose we have a randomly sampled training set $\mathcal{D}$, and we select an estimator denoted $\theta = \hat{\theta}(\mathcal{D})$. Then, for a particular test input x, the bias of our estimator's prediction on x is given as* $\text{Bias}(f_\theta(x)) = \mathbb{E}_{y\sim p(y|x),\mathcal{D}}[f_\theta(x) - y]$*. The variance of an estimate is a measure of how much the estimate differs from the expected value of the estimate, and is given by* $\text{Var}(f_\theta(x)) = \mathbb{E}_{\mathcal{D}}[(f_\theta(x) - \mathbb{E}_{\mathcal{D}}[f_\theta(x)])^2]$.

In supervised learning, our goal is to learn a function that does well in terms of the true risk. However, we generally do not know the true distribution and only have access to a dataset of samples from the distribution, and instead learn by minimizing the *empirical risk* (often with an additional regularization term) instead.

Even though we cannot directly optimize the risk, we can still attempt to better understand sources of error in our estimation. In particular, when our loss is the squared error, we can derive the **bias-variance** decomposition of the MSE. Specifically, we will find that the MSE estimator is exactly to the variance of the estimator plus the square of its bias (and an irreducible error).

Intuitively, the bias and variance can be summarized by the following graphic:

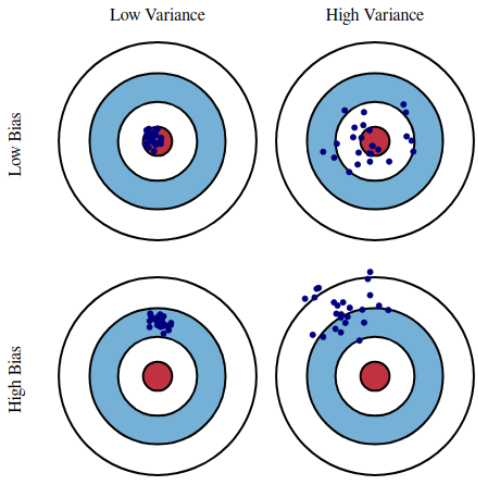


Figure 1: A visual explanation of the bias and variance. Figure from [**?**].

Here, notice that when there is a high variance, the estimates are more spread out, but when there is a high bias, we see a general deviation away from our target. The best estimates are those with low variance and low bias since they mostly hit the target.

1.2 Bias-Variance Tradeoff Mechanics

Problem 1: Deriving Bias-Variance Tradeoff

Suppose we have a randomly sampled training set $\mathcal{D}$ (drawn independently from our test data), and we select an estimator denoted $\theta = \hat{\theta}(\mathcal{D})$ (for example, via empirical risk minimization).
Show that we can decompose our expected mean squared error for a particular test input x into a bias, a variance and an irreducible error term as below:

$$\mathbb{E}_{Y\sim p(y|x),\mathcal{D}}[(Y - f_{\hat{\theta}(\mathcal{D})}(x))^2] = \text{Var}(f_{\hat{\theta}(\mathcal{D})}(x)) + \text{Bias}(f_{\hat{\theta}(\mathcal{D})}(x))^2 + \sigma^2$$

You may find it helpful to recall the formulaic definitions of Variance and Bias, reproduced for you below:

$$\text{Var}(f_{\hat{\theta}(\mathcal{D})})(x) = \mathbb{E}_{\mathcal{D}}\left[(f_{\hat{\theta}(\mathcal{D})}(x) - \mathbb{E}[f_{\hat{\theta}(\mathcal{D})}(x)])^2\right]$$
$$\text{Bias}(f_{\hat{\theta}(\mathcal{D})}(x)) = \mathbb{E}_{Y\sim p(Y|x),\mathcal{D}}[f_{\hat{\theta}(\mathcal{D})}(x) - Y]$$

We have now decomposed our test risk into a bias, variance and irreducible error term. As there is nothing we can do about the irreducible error, this tells us that we need to choose the learning algorithm and/or hyperparameters $\hat{\theta}(\cdot)$ in order to simultaneously achieve low bias and low variance. The next two questions will show how the choice of estimator $\hat{\theta}$ can influence bias and variance. In particular, we will see that ℓ_2 regularization in linear regression can provide a *tradeoff* between bias and variance.

Problem 2: Deriving Bias and Variance of Linear Regression Models

Our dataset consists of $\mathcal{D} = \{(x_i, y_i)\}_{i=1}^n$. We let the label vector $Y = \mathbf{X}\theta + \boldsymbol{\varepsilon}$ where θ is the true linear predictor and each noise variable $\boldsymbol{\varepsilon}_i$ is i.i.d. with mean 0 and variance 1. We use the ordinary least squares model. Calculate the bias and covariance of the $\hat{\theta}$ estimate and use that to compute the bias and variance of the prediction at particular test inputs x. Recall that the OLS solution is given by

$$\hat{\theta} = (X^\top X)^{-1} X^\top Y,$$

where $X \in \mathbb{R}^{n\times d}$ is our (nonrandom) data matrix, $Y \in \mathbb{R}^d$ is the (random) vector of training targets. For simplicity, assume that $X^\top X$ is diagonal (we could have applied an orthogonal transformation to make this the case), or for an even simpler problem that doesn't require linear algebra, assume $X \in \mathbb{R}^{n\times 1}$, making $X^\top X$ simply a scalar value.

Problem 3: Deriving Bias and Variance of Linear Regression Models (Challenge)

What happens to the bias and variance if we instead use an ℓ_2 regularized estimator

$$\tilde{\theta} = (X^\top X + \lambda I)^{-1} X^\top Y?$$

2 Vector and Matrix Calculus Review

In this section, we review vector and matrix calculus, and formalize the notation we will use. These notations will be required to understand backpropagation in the next lectures. Henceforth, we will denote scalars with lowercase letters (e.g., x), vectors with bolded lowercase letters (e.g., $\mathbf{x}$) and matrices with upper case letters (e.g., X). We will use similar conventions for functions depending on the shape of its output (e.g., $\mathbf{g}(\cdot)$ denotes a function with a vector valued output).

Gradients with respect to vectors We first define the gradient of a scalar function with respect to a vector input. Suppose we have a function $f : \mathbb{R}^d \to \mathbb{R}$, which maps a d-dimensional vector to a scalar. Then we define the gradient of f at a particular input x to be a column vector (the same shape as the input) consisting of partial derivatives at x:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f}{\partial x_1}(\mathbf{x}) \\ \vdots \\ \frac{\partial f}{\partial x_d}(\mathbf{x}) \end{bmatrix}.$$

We note that this choice of notation (laying out gradients to be the same shape as input) is not universally used; you will often find sources using the opposite convention (especially in mathematics) with gradients as row vectors (and Jacobians will be the transpose of what we describe next). However, we will use this convention for deep learning because it is intuitive (for example, in gradient descent, we often write $\theta \leftarrow \theta - \alpha \nabla L(\theta)$, which only makes sense when θ and $\nabla L(\theta)$ are the same shape) and because it is easily extended to gradients for matrices and higher dimensional arrays.

Problem 4: Gradient of squared ℓ_2 norm

Suppose we have a vector $\mathbf{x} \in \mathbb{R}^d$, and let $f(\mathbf{x}) = \|\mathbf{x}\|_2^2 = \mathbf{x}^\top \mathbf{x}$. Compute the gradient $\nabla f(\mathbf{x})$.

Jacobians We now consider the case where $\mathbf{f} : \mathbb{R}^n \to \mathbb{R}^m$ has vector valued inputs and outputs. Let $f_i : \mathbb{R}^n \to \mathbb{R}$ be the function that outputs the ith component of $\mathbf{f}$. Then, we can view our Jacobian (which we shall denote as $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$) as stacking together the gradients of f_i for $i \in \{1, \ldots, m\}$. That is, the Jacobian $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ will be an $n \times m$ matrix with entries given by

$$\left(\frac{\partial \mathbf{f}}{\partial \mathbf{x}}\right)_{ij} = \frac{\partial f_j}{\partial x_i}.$$

Problem 5: Jacobian of a linear map

Suppose we have a vector $\mathbf{x} \in \mathbb{R}^d$ and a matrix $A \in \mathbb{R}^{d \times n}$. Let $\mathbf{f}(\mathbf{x}) = A^\top \mathbf{x} \in \mathbb{R}^n$. Compute the Jacobian of $\mathbf{f}$ with respect to $\mathbf{x}$.

Multivariate Chain Rule We first recall the basic chain rule when everything is scalar valued. Suppose we have an input x, compute $y = g(x)$, then compute $z = f(y)$. Then the chain rule says

$$\frac{\partial z}{\partial x} = \frac{\partial y}{\partial x} \frac{\partial z}{\partial y}.$$

Now let's consider the case where $\mathbf{y}$ is vector valued in $\mathbb{R}^n$ (x and z remain scalars). Summing over the

contributions of each entry of $\mathbf{y}$, we see $\frac{\partial z}{\partial x}$ is now a scalar given by

$$\sum_{i=1}^{n} \frac{\partial y_i}{\partial x} \frac{\partial z}{\partial y_i}.$$

Finally, let's consider the case when $\mathbf{x}$ is also a vector in $\mathbb{R}^m$. From our calculation with scalar x and vector $\mathbf{y}$, we know the jth entry of $\frac{\partial z}{\partial \mathbf{x}}$ is given by the partial derivative

$$\frac{\partial z}{\partial x_j} = \sum_{i=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial x_j}.$$

Stacking together the entries $\frac{\partial z}{\partial x_j}$ into a vector, we see that the gradient of the output z with respect to x is given by the product of the Jacobian matrix of $\mathbf{y}$ with respect to $\mathbf{x}$ and the gradient of z with respect to $\mathbf{y}$:

$$\overbrace{\frac{\partial z}{\partial \mathbf{x}}}^{\mathbb{R}^m} = \overbrace{\frac{\partial \mathbf{y}}{\partial \mathbf{x}}}^{\mathbb{R}^{m \times n}} \overbrace{\frac{\partial z}{\partial \mathbf{y}}}^{\mathbb{R}^n}.$$

Problem 6: Combining the two previous calculations with the chain rule

Suppose we have a vector $\mathbf{x} \in \mathbb{R}^d$ and a matrix $A \in \mathbb{R}^{d \times n}$. Let $\mathbf{g}(\mathbf{x}) = A^\top \mathbf{x} \in \mathbb{R}^n$, and let $f(\mathbf{y}) = \|\mathbf{y}\|_2^2$. Compute the gradient of $f(\mathbf{g}(\mathbf{x}))$ with respect to $\mathbf{x}$.

Gradients with respect to matrices and higher dimensional arrays Now suppose we have a function $f : \mathbb{R}^{d_1 \times d_2} \to \mathbb{R}$, which maps a d_1 by d_2 matrix to a scalar. We will again define the gradient at a particular input matrix X to be a matrix of the same shape as X, consisting of the partial derivatives with respect to each entry of the matrix.

$$\nabla_X f(X) = \begin{bmatrix} \frac{\partial f}{\partial X_{11}} & \cdots & \frac{\partial f}{\partial X_{1,d_2}} \\ \vdots & \ddots & \vdots \\ \frac{\partial f}{\partial X_{d_1,1}} & \cdots & \frac{\partial f}{\partial X_{d_1,d_2}} \end{bmatrix}.$$

Similarly, we can generalize this convention of having the gradient match the shape of the input when our input were higher dimensional arrays (e.g. in the weights of a convolutional layer).

We can define a version of a Jacobian for vector-valued functions with matrix inputs that preserves the matrix dimensions (similarly for higher dimensional arrays as well). Suppose $\mathbf{f} : \mathbb{R}^{d_1,d_2} \to \mathbb{R}^n$, then we can define the Jacobian to be a rank-3 tensor (an array with 3 indices) in $\mathbb{R}^{d_1 \times d_2 \times n}$ with each entry given by

$$\left(\frac{\partial \mathbf{f}}{\partial X}\right)_{ijk} = \frac{\partial f_k}{\partial X_{ij}}.$$

We will now go through the chain rule calculation again, this time with a matrix input. Suppose $X \in \mathbb{R}^{d_1,d_2}$, $\mathbf{y} = \mathbf{g}(X) \in \mathbb{R}^n$ and $z = f(\mathbf{y}) \in \mathbb{R}$. Again, we have that the partial derivative with respect to each entry of the matrix X_{ij} is given by

$$\frac{\partial z}{\partial X_{ij}} = \sum_{k=1}^{n} \frac{\partial z}{\partial y_i} \frac{\partial y_i}{\partial X_{ij}}.$$

Similarly to the vector input case, we can again succinctly write out the full gradient with respect to the matrix X as

$$\overbrace{\frac{\partial z}{\partial X}}^{\mathbb{R}^{d_1 \times d_2}} = \overbrace{\frac{\partial y}{\partial X}}^{\mathbb{R}^{d_1 \times d_2 \times n}} \overbrace{\frac{\partial z}{\partial y}}^{\mathbb{R}^{n}}.$$

Note that the product of the rank-3 tensor (or 3-dimensional array) and vector can be a seen as a generalization of a matrix vector multiplication. Multiplying a matrix $X \in \mathbb{R}^{m \times n}$ by a vector $y \in \mathbb{R}^n$ results in a vector in $\mathbb{R}^m$ where each entry is the inner product of a row of X with y. The product of a rank-3 tensor $A \in \mathbb{R}^{d_1 \times d_2 \times n}$ with a vector $\mathbf{b} \in \mathbb{R}^n$ then forms a $d_1 \times d_2$ matrix, where each entry is the inner product of a "row" of A and $\mathbf{b}$.

We also note that this calculation of the gradient with respect to a matrix X is equivalent to first flattening X to a vector, computing the gradient with respect to the flattened X using the previous multivariate chain rule for vectors, and then reshaping the gradients back to match the original matrix shape of X.

Problem 7: Revisiting with a matrix derivative instead

In problem 5, we computed the gradient of $z = \left\|A^\top \mathbf{x}\right\|_2^2$ with respect to $\mathbf{x}$. We will now repeat this exercise, but instead compute the gradient with respect to A.
Suppose we have a vector $\mathbf{x} \in \mathbb{R}^d$ and a matrix $A \in \mathbb{R}^{d \times n}$. Let $\mathbf{g}(A) = A^\top \mathbf{x} \in \mathbb{R}^n$, and let $f(\mathbf{y}) = \|\mathbf{y}\|_2^2$. Compute the gradient of $f(\mathbf{g}(A))$ with respect to A.

Finally, as a matter of notation, note that the way we order derivatives in our chain rule (with the final output on the rightmost side) is again a result of our chosen convention for gradients and Jacobians. You may notice in other texts that the chain rule is written in the reversed order using a different convention for Jacobians.

3 Optimization methods

To perform empirical risk minimization, we need to choose an algorithm to compute the optimal parameters for the empirical risk. In deep learning, we almost always use methods based off stochastic gradient descent due its scalability (both in terms of dataset size and model size), and we'll go through and review several optimization methods as introduced in lecture.

3.1 Gradient Descent

For all our algorithms, we assume we can compute the gradients of our loss function for each data point $\nabla_\theta L(\boldsymbol{x}_i, y_i, \boldsymbol{\theta})$. The negative gradient of a function gives the steepest descent direction; that is, the direction we should move in order to decrease the loss most quickly if we moved an infinitesimally small amount.

Given this, the most natural method for minimizing our training loss is to iteratively compute the gradient for the entire dataset $\mathcal{D}$, and update our parameter some small amount in that direction. This leads to the (batch) **gradient descent** algorithm which computes iterates as

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \frac{\alpha}{|\mathcal{D}|} \sum_{\boldsymbol{x}_i, y_i \in \mathcal{D}} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}_i, y_i, \boldsymbol{\theta}^t),$$

where α is a fixed scalar known as the step size. However, this naive algorithm requires us to take the average gradient over the entire training dataset for each iteration, which can be too slow for larger datasets.

3.2 Stochastic Gradient Descent (SGD)

In order to speed up gradient descent, we can instead sample a random subset of the dataset for each iteration, leading to the (minibatch) **stochastic gradient descent** algorithm. The gradient estimate at each iteration is now noisy (with the amount of noise depending on the minibatch size), but is an unbiased estimator for the true gradient and can still provide useful directions to update our parameters. Compared to batch gradient descent, we are trading off variance in the gradients for much faster gradient computation that does not need to scale with the dataset size. While extra noise can mean it requires more iterations to converge, the faster individual iterate time can more than offset the cost.

The SGD algorithm proceeds exactly as the gradient descent algorithm, only replacing the full training set $\mathcal{D}$ with a new random minibatch B^t for each iteration

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \frac{\alpha}{|B^t|} \sum_{\boldsymbol{x}_i, y_i \in B^t} \nabla_{\boldsymbol{\theta}} L(\boldsymbol{x}_i, y_i, \boldsymbol{\theta}^t) \qquad\qquad B^t \subseteq \mathcal{D}.$$

All algorithms below are all compatible with stochastic gradients, so for notational convenience, we will now denote the (possibly stochastic) gradient estimate at iterate t to simply be $\nabla L(\theta^t)$. The new SGD/GD update in this notation would simply be

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t - \alpha \nabla L(\boldsymbol{\theta}^t).$$

3.3 When does gradient descent work poorly?

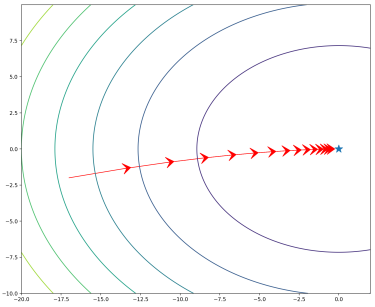
We'll first understand when gradient descent fails for a very simple convex problem, which will motivate different extensions. We consider a simple quadratic loss function $f(x_1, x_2) = a_1 x_1^2 + a_2 x_2^2$ and we'll look at the behavior of GD for different settings of the loss parameters a_1, a_2.

Gradient descent can work well when f is "nice" in some sense (which we'll refer to as being well-conditioned), which will be the case if a_1 and a_2 are similar in magnitude. We see this in Figure 2a, where the direction of gradient always aligns closely with the direction of the optimum, and GD can work with a fairly large learning rate to make quick steady progress to the optimum.

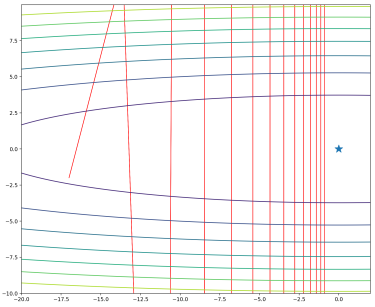
In Figures 2b and 2c, we consider an "ill-conditioned" problem where $a_2 \gg a_1$. In Figure 2b, with the same learning rate as before, the gradients tend to have a much larger vertical component than horizontal, and

the iterates now oscillate greatly along the vertical axis and diverge. If we use a much smaller learning rate in Figure 2c, we are able to stabilize the parameter updates in the vertical axis, but we end up making much slower progress to the optimum along the horizontal axis.

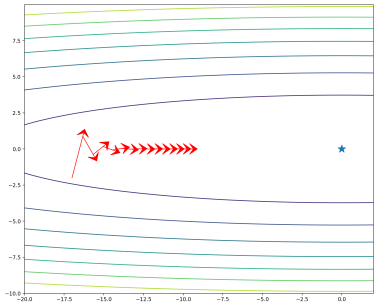
Thus, when our iterates are forced to always move in the direction of the gradient, we can have situations where either the iterates are very unstable, or our learning rate is so small that we make very slow progress. To address this pathology, we shall now examine different methods which do not update the parameters exactly in the direction of the gradient, but instead modifies the direction based on past gradients seen.



(a) With a large learning rate, GD on this well-conditioned problem makes fast steady progress towards the optimum.



(b) With a large learning rate, GD on this ill-conditioned problem oscillates and diverges in the vertical direction.



(c) With a small learning rate, GD on this ill-conditioned problem remains stable, but makes very slow progress towards the optimum.

Figure 2: Different behaviors of gradient descent on different problems and with different learning rates.

3.4 SGD with momentum

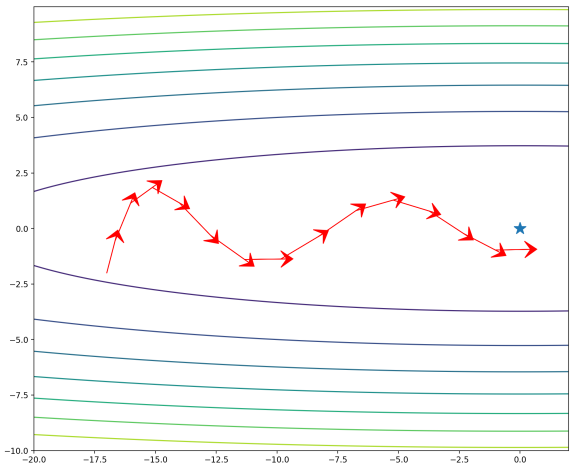


Figure 3: Effect of using momentum on the ill-conditioned quadratic problem. SGD with momentum quickly realizes that all the gradients consistently point towards the right, so it "gains momentum" and moves faster along that direction over time. On the other hand, the alternating vertical gradients tend to cancel out over the past iterations, damping the amount of vertical oscillation. This way, the iterates make quick progress toward the optimum along the horizontal direction, while not diverging along the vertical axis.

One way to help alleviate this problem is to accumulate gradient information across previous iterates in order to damp the oscillations and focus on the directions where we have consistently been moving in. In the example in the left of Figure 2b, we notice that each gradient consistently moves the iterate towards the right (towards the optimum), while alternating iterations have gradients point in different directions along

the vertical axis. If we average the past gradients using momentum (Figure 4), we see that the vertical components of the gradient will tend to cancel out and stabilize, allowing the horizontal movement towards the right to dominate and lead us to the optimum more quickly. In the stochastic gradient setting, momentum can also reduce the impact of noise, again by smoothing out the oscillations int he gradient direction incurred by the random data sampling.

Concretely, the momentum method (the particular variant we use is also called the heavy-ball method) keeps a moving average of our gradients, weighting more recent gradients more heavily, and updates our iterate in the direction of the weighted average. The iterates proceed as

$$\begin{aligned} \mathbf{v}^t &= m\mathbf{v}^{t-1} + \nabla L(\boldsymbol{\theta}^t) \\ \boldsymbol{\theta}^{t+1} &= \boldsymbol{\theta}^t - \alpha \mathbf{v}^t. \end{aligned}$$

Here $\mathbf{v}^t$ is our accumulated gradient vector, m controls how much we remember the past gradients, and α is our step size as before.

3.5 RMSProp and Adagrad

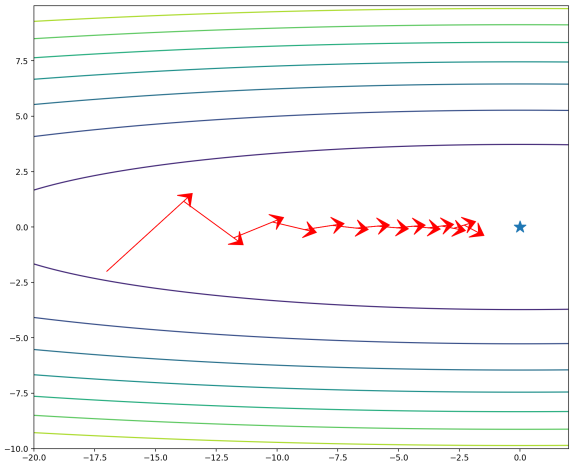


Figure 4: Effect of using adaptive learning rates on the ill-conditioned quadratic problem. In this case, we use RMSProp, which is able to rescale the different parameter updates to make fast progress to the optimum instead of oscillating as regular gradient descent did.

Adaptive learning rate methods, which include RMSprop and Adagrad, are an alternative approach towards selecting a better direction based on rescaling the different components of the gradient to form a new direction. One issue we notice with the ill-conditioned loss in the previous figures is that the loss is much more sensitive along the vertical axis than the horizontal one, and that the extreme sensitivity along the vertical axis was what forced us to use a small learning rate for gradient descent. Intuitively, if we could decouple learning rates for each coordinate and rescale our updates to place more importance on the horizontal direction, we would move much faster towards the optimum.

Concretely, both RMSProp and Adagrad do this individual rescaling by estimating a vector $\mathbf{s}^t$ that tracks the "size" of the past gradients in each dimension. They both update the kth coordinate of the parameters according to

$$\theta_k^{t+1} = \theta_k^t - \frac{\alpha}{\sqrt{s_k^t + \epsilon}} \nabla_{\theta_k} L(\theta^t),$$

where ϵ is a small constant for numeric stability (to avoid dividing by zero).

RMSProp and Adagrad differ in how they update s^t:

$$s_k^t = \beta s_k^{t-1} + (1-\beta)(\boldsymbol{\nabla}_{\theta_k} L(\theta^t)^2 \qquad \text{RMSProp}$$
$$s_k^t = s_k^{t-1} + \beta(\boldsymbol{\nabla}_{\theta_k} L(\theta^t))^2 \qquad \text{Adagrad}$$

RMSProp keeps a *running average* of per dimension gradient magnitudes, while Adagrad keeps a *sum*. Thus, for Adagrad, the vector $\mathbf{s}^t$ monotonically increases over time, causing the effective learning for each coordinate to decrease monotonically.

3.6 Adam and other algorithms

Adam is a popular optimizer in deep learning that essentially combines the adaptive learning rates of RMSProp with momentum.

For additional information about these optimization methods (as well as numerous others), here's a blog post with a list of popular optimization methods for deep-learning: http://ruder.io/optimizing-gradient-descent/

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine

Discussion 3

This discussion will cover some practice applying backpropagation, and introduce the convolution operator.

1 Mechanical Backpropagation

In this section, we will work through some calculations used during backpropagation.

Recall the softmax function $\mathbf{p} : \mathbb{R}^k \to \mathbb{R}^k$, with entries given by

$$p_i(\mathbf{z}) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}.$$

Each entry p_i corresponds to the probability assigned to the label i. We derived in Discussion 1 that the partial derivatives of $p_i(\mathbf{z})$ for each entry of $\mathbf{z}$ is given by,

$$\begin{aligned}\frac{\partial p_i(\mathbf{z})}{\partial z_j} &= \begin{cases} p_i(\mathbf{z})(1 - p_j(\mathbf{z})) & \text{if } i = j \\ -p_i(\mathbf{z})p_j(\mathbf{z}) & \text{if } i \neq j \end{cases} \\ &= p_i(\mathbf{z})(\delta_{ij} - p_j(\mathbf{z})).\end{aligned}$$

We can then concicsely write the full gradient with respect to $\mathbf{z}$ as

$$\nabla p_i(\mathbf{z}) = p_i(\mathbf{z})(\mathbf{e}_i - \mathbf{p}(\mathbf{z})),$$

where $\mathbf{e}_i$ is the unit vector with 1 at index i and 0 elsewhere.

In this example, we will maximize the log-likelihood of the given labels in our dataset, which motivates the following loss for a multiclass logistic regression model.

$$L(\mathbf{x}, y, W, \mathbf{b}) = -\log p_y(W\mathbf{x} + \mathbf{b}).$$

Problem 1: Gradient with respect to linear layer parameters

Utilize the chain rule to compute the gradient of $L(\mathbf{x}, y, W, \mathbf{b})$ with respect to W and $\mathbf{b}$.

Suppose now that we had a multilayer neural network and $W, \mathbf{b}$ were the the parameters of the last layer of the network. To compute gradients of the earlier parameters of the network with backpropagation, we also need to compute the gradient of the loss with respect to $\mathbf{x}$ and pass it backwards.

Problem 2: Gradient with respect to input

Utilize the chain rule to compute the gradient of $L(\mathbf{x}, y, W, \mathbf{b})$ with respect to $\mathbf{x}$.

Having computed these, one then simply needs to also compute the backwards pass through the chosen activation function to able backpropagate through fully-connected feedforward networks!

We'll now move on to a slightly more complicated example of backpropagation involving a *skip connection*, which you'll see again when we cover ResNets.

Problem 3: Gradient in a nonlinear computation graph

Suppose we have $\mathbf{y} = W_2\sigma(W_1\mathbf{x}) + \mathbf{x}$, where σ is the ReLU activation. Letting $\delta_{\mathbf{y}}$ denote the gradient of the loss with respect to $\mathbf{y}$, compute the gradient of the loss with respect to $\mathbf{x}$.

2 Convolutional Neural Networks

Convolutional neural networks[1] (CNN) are a type of neural network architecture that have become the key ingredient for state of the art modern computer vision performance.

They perform operations similar to feed-forward neural networks that we have discussed, but explicitly account for spatial structure in the data, and so are very common for computer vision tasks where inputs are images. That said, CNNs can also be applied to non-image data with similar structure in the input, such as time series or text data (in which case they're takign advantage of temporal structure).

2.1 Convolution (Cross-Correlation) Operator

At the heart of CNNs is the convolution operator. In this discussion, what we refer to as a convolution is actually the **cross-correlation** operator here instead, which is the exact same but with the indexing of the weights in $\mathbf{w}$ inverted. For example, "convolutional" layers in the deep learning library Pytorch are also actually cross-correlations instead, and homework 1 will also similarly have you implement cross-correlation instead of the actual convolution.

To motivate the use of convolutions, we will work through an example of a 1-D convolution calculation to illustrate how convolutions work over a single spatial dimension. Suppose we have an input $\mathbf{x} \in \mathbb{R}^n$, and filter $\mathbf{w} \in \mathbb{R}^k$. We can compute the convolution of $\mathbf{x} \star \mathbf{w}$ as follows:

1. Take your convolutional filter $\mathbf{w}$ and align it with the beginning of $\mathbf{x}$. Take the dot product of $\mathbf{w}$ and the $\mathbf{x}[0 : k-1]$ (using Python-style zero-indexing here) and assign that as the first entry of the output.
2. Suppose we have stride s. Shift the filter down by s indices, and now take the dot product of $\mathbf{w}$ and $\mathbf{x}[s : k-1+s]$ and assign to the next entry of your output.
3. Repeat until we run out of entries in $\mathbf{x}$.

Below, we illustrate a 1D convolution with stride 1.

$$\overbrace{\begin{bmatrix} x_1 \\ \vdots \\ x_k \\ \vdots \\ x_n \end{bmatrix}}^{\text{Input vector } \mathbf{x} \in \mathbb{R}^n} \star \overbrace{\begin{bmatrix} w_1 \\ \vdots \\ w_k \end{bmatrix}}^{\text{Convolutional filter } \mathbf{w} \in \mathbb{R}^k} = \overbrace{\begin{bmatrix} \sum_{i=1}^k w_i x_i \\ \sum_{i=1}^k w_i x_{i+1} \\ \vdots \\ \sum_{i=1}^k w_i x_{i+n-k} \end{bmatrix}}^{\text{Output vector } \mathbf{y} \in \mathbb{R}^{n-k+1}}$$

We see that the output vector is smaller than the input vector ($\mathbb{R}^{n-k-1}$ compared to $\mathbb{R}^n$). A common way to address this is **zero-padding**, in which we append zeros on both ends of the input vector before applying the convolution (note that there are other conventions for zero-padding as well).

Often, we'll be dealing with multiple spatial dimensions (2 spatial dimensions in the case of images). In this case, we would need to slide our filter along all spatial dimensions to construct the output.

[1] Recommended reading: http://cs231n.github.io/convolutional-networks/

Problem 4: Test your know knowledge of convolution dimensions

In this problem, we will run a series of convolution-related operations to better understand how dimensions are affected by convolutions.

1. Suppose you have a $32 \times 32 \times 3$ image (a 32×32 image with 3 input channels). What are the resulting dimensions when you convolve with a $5 \times 5 \times 3$ filter with stride 1 and 0 padding?
2. What if we zero-pad the input by 2?
3. Suppose we now stack 10 of these $5 \times 5 \times 3$ filters and continue to zero pad the input by 2. What is the new shape of the output, and how many parameters are in our filters (not including any bias parameters)?
4. What would be the spatial dimensions after applying a 1×1 convolution? Think about what this does.

2.2 Convolutions as Matrix Multiplication

We note that convolutions are a linear operation. Recalling linear algebra, any linear map (between finite-dimensional spaces) can be expressed as a matrix, so we will see in this section how to write a convolution as a matrix multiplication.

Problem 5: Expressing convolutions as matrix multiplication

We shall again consider a 1D convolution. Consider an input $\mathbf{x} \in \mathbb{R}^4$ and filter $\mathbf{w} \in \mathbb{R}^3$. Letting $\bar{\mathbf{x}}$ denote the result of zero-padding the input by 1 on each end, what is the matrix W such that

$$\overbrace{W}^{\mathbb{R}^{4\times 6}} \overbrace{\begin{bmatrix} 0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \\ 0 \end{bmatrix}}^{\text{Zero padded input } \bar{\mathbf{x}} \in \mathbb{R}^6} = \bar{\mathbf{x}} * \mathbf{w}?$$

We can observe now that the resulting matrix will be very sparse (most entries are 0) if the filter size is much smaller than the input size, corresponding to the fact that such convolutions exploit spatial locality. We also observe that there is a lot of parameter reuse, as the convolutional filter weights are repeated many times throughout the explicit matrix.

This has several implications. First of all, this implies that convolutional layers are less expressive than fully-connected layers (as fully connected layers are represented by arbitrary matrices).

Another important implication stems from the fact that we have very optimized tools for computing matrix multiplications. While a naive implementation of a convolution will require looping over all the spatial dimensions, it will turn out that reformulating the convolution as a matrix multiplication will often be much faster due to these optimizations (for example, the Cythonized im2col function in part 4 of homework 1 essentially does this).

2.3 Backwards Pass for a Convolution

We'll consider the same 1D convolution as before, but without zero-padding for simplicity.

Problem 6: Backwards pass for convolutions

Let $\mathbf{y} = \mathbf{x} * \mathbf{w} \in \mathbb{R}^2$, where $\mathbf{w} \in \mathbb{R}^3$, $\mathbf{x} \in \mathbb{R}^4$. Let $\boldsymbol{\nabla}_{\mathbf{y}} L$ denote the gradient of the loss with respect to the output of the convolution. Compute the gradients of L with respect to $\mathbf{x}$ and $\mathbf{y}$. Can you express the gradients as convolutions themselves?

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine **Discussion 4**

This discussion will cover CNN architectures, batch normalization, weight initializations, ensembles and dropout.

1 Convolutional Neural Networks Architectures

We will survey the some most famous convolutional neural net architectures.

LeNet. Among the earlier CNN architectures, LeNet is the most widely known. LeNet was used mostly for handwritten digit recognition on the MNIST dataset. Importantly, LeNet used a series of convolutional layers, then pooling layers, followed by several fully connected (FC) layers.

AlexNet. The AlexNet architecture popularized CNNs in computer vision, when it won the ImageNet ILSVRC Challenge in 2012 by a large margin. AlexNet has a similar architectural design as LeNet, except that it is bigger (more neurons) and deeper (more layers). In addition, AlexNet demonstrated the benefits of using the ReLU activation and dropout for vision tasks, as well as the use of GPUs for accelerated training.

VGGNet This network was the runner-up in ILSVRC 2014 to GoogLeNet, and showed the benefit of (a) increasing the number of layers, and (b) using only convolutional operators stacked on each other. A downside is that this network has roughly 138 million parameters, so in general, consider using Residual Nets (see next item).

ResNet These networks use skip connections to allow inputs and gradients to propagate faster throughout the network (either forward or backwards). Residual networks were state of the art for image recognition results in mid-2016, and the general backbone is commonly used as of today. They have substantially fewer parameters than VGG. The exact number depends on what type of "ResNet-X" is used, where "X" represents the number of layers; PyTorch offers pretrained models for 18, 34, 50, 101, and 152. For reference, ResNet-152 should have about 60 million parameters.)

Problem: Vanishing Gradients in ResNet

How does skip connection in ResNet help solve the vanishing gradient problem?

2 Batch Normalization

The main idea behind Batch Normalization is to transform every sampled batch of data so that they have $\mu = 0$, $\sigma^2 = 1$. Using Batch Normalization typically makes networks significantly more robust to poor initialization. It based on the intuition that it is better to have unit Gaussian inputs to layers at initialization. However, the reason for why batch normalization works is not entirely understood, and there are conflicting views between whether Batch Normalization reduces covariate shift, improves smoothness over the optimization landscape, or other reasons.

In practice, when using batch normalization, we add a BatchNorm layer immediately after ecah FC or convolutional layer, either before or after the non-linearity. The key observation is that normalization is a relatively simple differentiable operation, so we do not add too much additional complexity in the network.

Noticeably, Batch Normalization proceeds by first computing the empirical mean and variance of some mini-batch B of size m from the training set.

$$\mu_B = \frac{1}{m}\sum_{i=1}^{m} a_i$$
$$\sigma_B^2 = \frac{1}{m}\sum_{i=1}^{m}(a_i - \mu_B)^2$$

Then, for a layer of network, each dimension, $a^{(i)}$ is normalized appropriately,

$$\bar{a}_i^{(k)} = \frac{a_i^{(k)} - \mu_B^{(k)}}{\sqrt{\left(\sigma_B^{(k)}\right)^2 + \epsilon}}$$

where ϵ is added for numerical stability.

In practice, after normalizing the input, we squash the result through a linear function with learnable scale γ and bias β, so, we have,

$$\bar{a}_i^{(k)} = \frac{a_i^{(k)} - \mu_B^{(k)}}{\sqrt{\left(\sigma_B^{(k)}\right)^2 + \epsilon}}\gamma + \beta$$

Intuitively, γ and β allows us to restore the original activation if we would like, and during training, they can learn other distribution that would be better initialization than standard Gaussian.

Problem: Examining the BatchNorm Layer

1. Draw out the computational graph of the BatchNorm layer
2. Given some *dout*, the derivative of the output of the BatchNorm layer, compute the derivatives with respect to input x and parameters γ, β

3 Ensembles

Definition 1 (Ensemble)**.** *Ensemble (bagging or boosting) group several models trained on the same task into a single model to aggregate predictions.*

Intuition The intuition for ensembles come from the recognition that neural networks have many parameters, often with high variance. Then, if we have multiple learners, we can average out the variance.

Ensemble Methods There are two ways we typically proceed with ensemble methods:

1. **Prediction Averaging**. Train N neural networks independently. Then, average their predictions (either probabilistically or by majority vote)
2. **Parameter Averaging**. Parameter averaging does not work in the same way as prediction averaging. Instead, we would only average over parameters from the context of snapshot ensembles, and average parameters over one trajectory, not over independent runs.

In practice, we do not need to reshuffle our dataset (and resample with replacement), since there is already a lot of randomness in neural network training form weight initialization, minibatch shuffling and SGD.

Making Ensemble Methods Faster Unfortunately, a downside to ensemble methods is that they can be very slow.

1. Only make classification layers (e.g., FC layers) ensembles.
2. Snapshot ensemble. Save out parameter snapshots over the course of SGD optimization and use each snapshot as a model.

4 Dropout

Definition 2 (Dropout)**.** *Dropout is a popular technique for regularizing neural networks by randomly removing nodes with probability* $1 - p_{keep}$ *in the forward pass. However, the model is unchanged at test time.*

Intuition Dropout can be thought of as representing an ensemble of neural networks, since each forward pass is effectively a different neural network, since random nodes are removed.

Activation Scaling A caveat about dropout is that we must divide the activation by p, since we do not change the model at test time, but we notice that none of our dimensions will then be forced to 0. Below is sample code to demonstrate how Dropout works in practice for a 3-layer network.

```
def dropout_train(X, p):
    """
    Forward pass for a 3-layer network.
    NOTE: For simplicity, we do not include backwards pass or parameter update

    X: Input
    p: Probability of keeping a unit active (e.g., higher p leads to less dropout)
    """
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    U1 = (np.random.rand(*H1.shape) < p) / p # first dropout mask. Notice /p
    H1 *= U1 # Drop the activations
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    U2 = (np.random.rand(*H2.shape) < p) / p # second dropout mask. Notice /p
    H2 *= U2 # Drop the activations
    out = np.dot(W3, H2) + b3
    return out

def predict(X):
    """ Forward pass at test time """
    H1 = np.maximum(0, np.dot(W1, X) + b1)
    H2 = np.maximum(0, np.dot(W2, H1) + b2)
    out = np.dot(W3, H2) + b3
    return out
```

Problem: Dropout Review

Explain why Dropout could improve performance and when we should use it

5 Weight Initialization

One of the reasons for poor model performance can be attributed to poor weight initialization. In class, we discussed two types of weight initialization,

1. Basic initialization: Ensure activations are reasonable and they do not grow or shrink in later layers (for example, Gaussian random weights or Xavier initialization)
2. Advanced initialization: Work with the eigenvalues of Jacobians

Problem: Deriving Xavier Initialization

Let our activation be the `tanh` activation, which is approximately linear with small inputs (i.e., $\mathsf{Var}(a) = \mathsf{Var}(z)$, where z is the output of the activation followed by some linear layer). We furthermore assume that weights and inputs are i.i.d. and centered at zero, and biases are initialized as zero.
We would like the magnitude of the variance to remain constant with each layer. Derive the Xavier Initialization, which initializes each weight as,

$$W_{ij} = \mathcal{N}\left(0, \frac{1}{D_a}\right)$$

where D_a is the dimensionality of a

6 Aside: ReLU Activations and its Relatives

Definition 3 (ReLU Activation)**.** *ReLU Activation is defined as,* $ReLU(x) = \max(0, x)$*, and is a popular activation function.*

On top of ReLU Activation, there exists its close relatives, like:

- **Leaky ReLU**. Instead of defining the ReLU as 0 for all $x < 0$, Leaky ReLU defines it as a small linear component of x.
- **ELU**. Instead of defining the ReLU as 0 for all $x < 0$, ELU defines it as $\alpha(e^x - 1)$ for some α

Problem: (Review) Forward and Backward Pass for ReLU

Compute the output of forward pass of a ReLU layer with input x as given below:

$$y = \text{ReLU}(x)$$

$$x = \begin{bmatrix} 1.5 & 2.2 & 1.3 & 6.7 \\ 4.3 & -0.3 & -0.2 & 4.9 \\ -4.5 & 1.4 & 5.5 & 1.8 \\ 0.1 & -0.5 & -0.1 & 2.2 \end{bmatrix}$$

With the gradients with respect to the outputs $\frac{dL}{dy}$ given below, compute the gradient of the loss with respect to the input x using the backward pass for a ReLU layer:

$$\frac{dL}{dy} = \begin{bmatrix} 4.5 & 1.2 & 2.3 & 1.3 \\ -1.3 & -6.3 & 4.1 & -2.9 \\ -0.5 & 1.2 & 3.5 & 1.2 \\ -6.1 & 0.5 & -4.1 & -3.2 \end{bmatrix}$$

Problem: ReLU Potpourri

1. What advantages does using ReLU activations have over sigmoid activations?
2. ReLU layers have non-negative outputs. What is a negative consequence of this problem? What layer types were developed to address this issue?

7 Summary

- Recall the main ConvNet architectures (LeNet, AlexNet, GoogLeNet, VGGNet, ResNet). In particular, recall why bottleneck layers is ResNet are important.
- Batch Normalization proceeds by first computing empirical mean and variance, then rescaling each activation and squashing through γ and β
- Ensembles group several models into a single model. To make this quicker, we can either only make classification layers ensembles or use snapshot ensemble.
- Dropouts are methods for randomly removing nodes, and intuitively represent an ensemble of networks, since each forward pass is effectively a different network

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine **Discussion 5**

This discussion covers vision applications and introduction to visualizing networks (style transfer, Dream Machine)

1 Review of Vision Problems

For most of the class thus far, when we discuss applying neural networks in practice to vision applications, we have largely assumed an image classification task. That is, given an image, we let the network output the probabilities of the true label belonging to a variety of classes.

However, there are more types of standard computer vision problems, namely, *object localization, object detection, semantic segmentation.* Below, we outline the four main types of computer vision problems.

Image Classification Given an image, we would like the network output the probabilities of the true label belonging to a variety of classes. This type of problem was the main focus of the course so far.

Object Localization Determine a *bounding box* for the object in the image that determines the class. In this type of problem, only one object is involved, and indeed, we know ahead of time that there is only one object class of interest in the image. Often, the bounding box objective may be simultaneously trained with the classification objective, resulting in a loss objective that is the sum of the two loss terms, the L_2 and the cross-entropy loss, respectively.

Object Detection Determine multiple objects in an image and their bounding boxes, with performance measured by *mean average precision* (mAP). There may be many objects, and several instances of the same object class (for e.g., several dogs) in the same picture. This means that, in contrast to image classification where the network only has to identify one object, the network has to predict a varying number of bounding boxes. In literature, object detection can be solved using Faster R-CNN.

Semantic Segmentation Label every pixel in the image. Here, we can naively run a CNN classifier for each pixel. However, better solutions, like `UNet`, exists in literature. *Semantic* segmentation means we do not worry about distinguishing between different instances of a class, in contrast to the aptly-named *instance* segmentation problem.

Problem 1: Calculate mAP

A common metric for object detection is *mean average precision* (mAP), where we compute average precision (AP) separately for each class, then average over classes. We say that a detection is a true positive if it has IOU (Intersection over Union) with a ground truth box greater than some threshold, and we can calculate the AP as the area under the precision / recall curve for each class.
You run an object detector, and get the following results:

Example	Predicted/Ground Truth IOU
A	0.29
B	0.11
C	0.701
D	0.001
E	0.92
F	0.45

If all candidates are true positives and we threshold the IOU at 0.5, what is the Average Precision of our object detector? What is the Mean Average Precision (mAP) when using the thresholds $(0, 1)$? Please note that we did not explicitly cover mAP in lecture.

2 Object Detection: R-CNN

Faster R-CNN is a popular technique for object detection problems, and stands for Faster *Region*-CNN. Faster R-CNN uses these regions as areas in the image that are likely to contain objects. More precisely, a *Region Proposal Network* predicts proposals from CNN features. The CNN features were obtained from passing the original input image through several convolutional layers.

The network is trained jointly using four losses, which normally means adding up the objectives (possibly with different weights).

3 Segmentation: Transposed Convolution & U-Net

We briefly comment on an operator called the *transpose convolution*, because it's often used for *upsampling* a convolutional neural network during segmentation tasks. This operator increases the resolution of the intermediate tensors, which we often want if we want the output of our network to be an image (e.g., of the same size as the input images). Note that early convolutional and pooling layers tend to *downsample* or reduce the size of tensors. Please note that it is sometimes referred to as a *deconvolution* operator, but it is not the preferred wording because it is an overloaded term with other definitions commonly used.

The transpose convolution can be thought of as flipping the forward and backward passes of the convolution step. In addition, the naming comes from how it can be implemented in a similar manner as in convolution but with the weight matrix transposed (along with different padding).

Convolutional layers typically downsample images spatially but sometimes we want to upsample. For example in semantic segmentation or in DCGAN where we generate images from random noise of a lower dimension.

Problem 2: 2D Transpose Convolution Mechanics

Let our input be

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

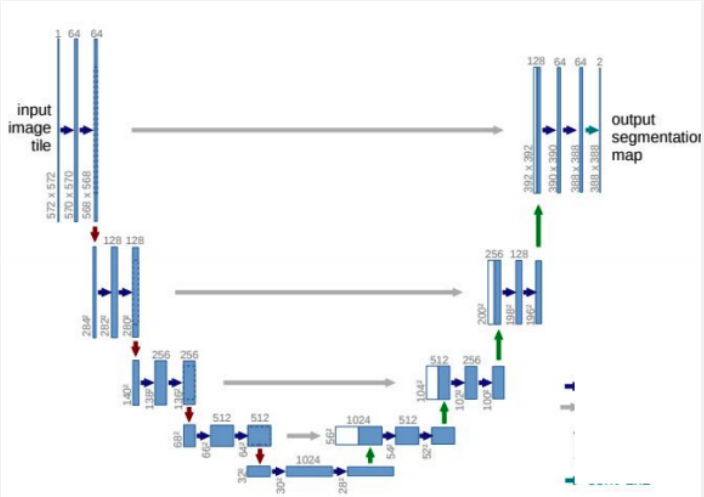
and kernel be

$$\begin{bmatrix} 0 & 1 \\ 2 & 3 \end{bmatrix}$$

Assume input, output channels of 1, padding of 0 and stride of 1, what is the output of transposed convolution layer?

Problem 3: U-Net Potpourri

The figure below shows a U-Net architecture.



Answer the following questions,

(a) What operations are represented by upward arrows in the figure?

(b) What is the role of rightward arrows?

(c) Which of the transformations in the network have learned parameters?

4 DeepDream

The main idea behind DeepDream is to exaggerate details in an image that look like recognizable objects.

Simply put, the procedure for DeepDream is as follows,

1. Pick a layer
2. Forward propagate to the layer
3. Set the gradient at that layer to the activation at that layer
4. Backpropagate to update the image

5 Style Transfer

Style transfer is a class of algorithms to manipulate digital images, or videos, in order to adopt the appearance or visual style of another image.

We assume an input image p and a style image a. Then, the image p is fed through a CNN (in the original paper, through a VGG-19 architecture), and network layer activations are sampled at the early to middle layers of the CNN. The style image a is also fed through the same CNN, and network activations are sampled at early to middle layers of the CNN, and are encoded into a Gram matrix, $S(a)$. The Gram matrix encodes the correlation between different features.

Then, the goal of style transfer is to synthesize some output x, such that $C(x)$ approximates $C(p)$, and $S(x)$ approximates $S(a)$. Then, our loss function is,

$$\mathcal{L}(x) = \|C(x) - C(p)\|_2 + k\|S(x) - S(a)\|_2$$

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine

Discussion 6

This discussion covers recurrent neural networks and LSTMs. Your TA will also walk through some midterm preparation for the midterm.

1 Recurrent Neural Network

The world is full of sequential information, from video to language modelling to time series data. In particular, we would like to model these sequences using neural networks, and solve some major types of tasks that we would like to solve with sequence models.

1.1 Types of Problems

- **One-to-one** problems take a single input x and produce a single output y. Problems like classification (takes an image as input, and produces a class label as output) and semantic segmentation (image as input, segmentation mask as output) fall under this category.
- **One-to-many** problems take a single input, and produce a sequence of output. Problems like image captioning (takes a single image as input, and produces a caption (a sequence of words) as output) fall under this category.
- **Many-to-many** problems take sequences of inputs and produce sequences of outputs. Problems like language translation (sequence of words in one language to sequence of words in another) fall under this category

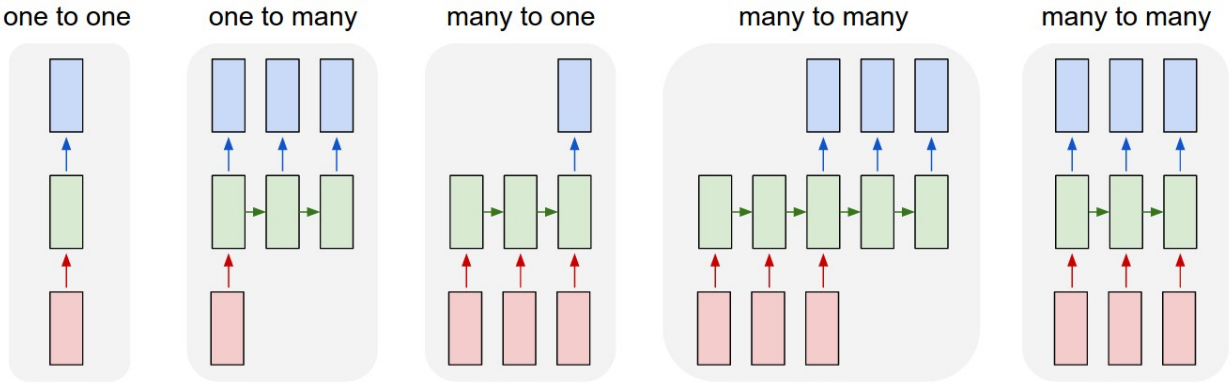
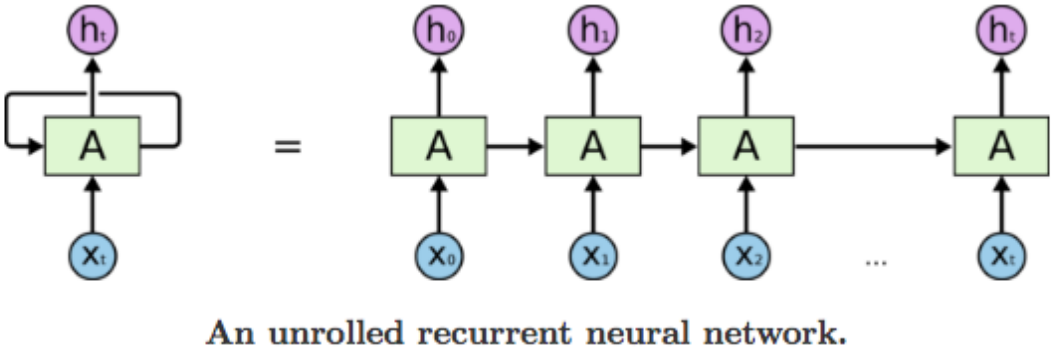


Figure 1: Types of problems we would like to solve using sequential models

1.2 Why the Recurrence?

As you read through this discussion worksheet, you don't process each word entirely on its own, but instead use your understanding from the previous words as well. Traditional neural networks do not have the capability to use its reasoning about previous events to infer later ones. For example, if we would like to classify what is happening at every frame in a movie, this can be framed as an image classification task where the network is provided the current image. However, it is unclear how a traditional neural network should incorporate knowledge from the previous frames in the film to inform later ones.

Recurrent neural networks (RNNs) address this issue, by using the idea of "recurrent connections." RNNs are networks with loops in them that allow information from previous inputs to persist as the network processes the future inputs. These recurrent connections allow information to propagate from "the past" (earlier in the sequence) to the future (later in the sequence).



An unrolled recurrent neural network.

Figure 2: An example of a generic recurrent neural network. This shows how to "unroll" a network through time - instead of thinking about sequence modeling as a single network with shared weights

In Figure 2, we illustrate the RNN computation as it is unrolled through time. Each $i \in \{0, \ldots, t\}$ represents a new timestep in the network. By feeding in a state computed from earlier timesteps as an input together with the current input, information can persist throughout the time as the network "remembers" the past inputs it processed.

1.3 Vanilla RNN

In the following section, we will use the following notation. Denote the input sequence as $x_t \in \mathbb{R}^k$ for $t \in \{1, \ldots, T\}$, and output of the network be $y_t \in \mathbb{R}^m$ for $t \in \{1, \ldots, T\}$. In the following example, we construct a"vanilla" many-to-many RNN, consisting of a node that updates the hidden state h_t and produces an output y_t at each timestep with the following equations:

$$h_t = \tanh(W_{h,h} h_{t-1} + W_{x,h} x_t + B_h)$$
$$y_t = W_{h,y} h_t + B_y$$

where h_t is the time step of a hidden state (one can think of h_{t-1} as the previous hidden state), $W_{\cdot,\cdot}$ be the set of weights (for example, $W_{x,h}$ represents weight matrix that accepts an input vector and produce a new hidden state), y_t be the output at timestep t and B_h and B_y be the bias terms. We can also represent it as the diagram below,

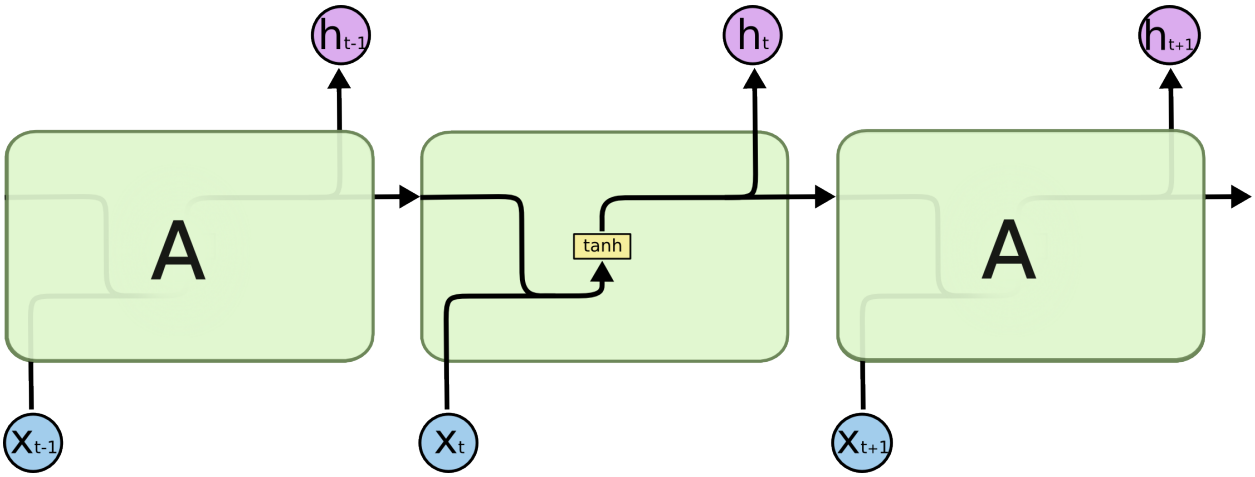


Figure 3: A simple RNN cell. As we can see by the arrows, we only pass a single hidden state from time $t-1$ to time t

In this vanilla RNN, we update to a hidden state "h_t" based on the previous hidden state h_{t-1} and input at the current time x_t, and produce an output which that is a simple affine function of the hidden state. To compute the forward (and backward) passes of the network, we have to "unroll" the network, as shown in Figure 2. This "unrolling" process creates something that resembles a very deep feed forward network (with depth corresponding to the length of the input sequence), with shared affine parameters at each layer. Our gradient is computed by summing the losses from each time-step of the output.

Problem: Gradients in Vanilla RNN

Why are vanishing or exploding gradients an issue for RNNs?

Problem: Coding RNNs Up!

Complete the class definition, started for you below,

```
import numpy as np

class VanillaRNN:
    def __init__(self):
        self.hidden_state = np.zeros((3, 3))
        self.W_hh = np.random.randn(3, 3)
        self.W_xh = np.random.randn(3, 3)
        self.W_hy = np.random.randn(3, 3)
        self.Bh = np.random.randn(3)
        self.By = np.random.randn(3)
        self.hidden_state = np.zeros(3)

    def forward(self, x):
        # Processes the input at a single timestep and updates the hidden state
        self.hidden_state = np.tanh(...)
        self.output = np.dot(...) + ...
        return self.output
```

2 Long Short Term Memory (LSTM)

To address the problem of vanishing and exploding gradients, we can use a different kind of recurrent cell - the LSTM cell (standing for "long short term memory"). The layout of the cell is shown in Figure 4. The LSTM has two states which are passed between timesteps: a "cell memory" C and the hidden state h. The LSTM update is given as follows:

$$
\begin{aligned}
f_t &= \sigma(x_t U^f + h_{t-1} W^f) \\
i_t &= \sigma(x_t U^i + h_{t-1} W^i) \\
o_t &= \sigma(x_t U^o + h_{t-1} W^o) \\
\tilde{C}_t &= \tanh(x_t U^g + h_{t-1} W^g) \\
C_t &= f_t \circ C_{t-1} + i_t \circ \tilde{C}_t \\
h_t &= \tanh(C_t) \circ o_t
\end{aligned}
$$

where $\circ$ represents the Hadamard Product (elementwise multiplication).

The update function is rather complex, but it makes a lot of sense when looking at it in the context of the cell state C as a "memory". First, we compute the value f_t, which we call the "forget" gate, as it controls how we retain information through time. Because of the sigmoid activation function, f_t is bounded between 0 and 1, and the first thing we do is multiply the previous memory by f_t. Intuitively, if f_1 is close to 1, we "remember" the previous state, and if f_t is close to 0, we forget it. Next, we compute i_t, which we consider as the "input/update" gate, which controls how much we update the cell memory at the current timestep. The update gate gets added to the memory cell, so it takes information from the current input x_t and adds it to the memory. Finally, the output gate o_t controls the output of the network, the value that gets passed on to the next cell.

We can compare the LSTM to a vanilla RNN. Since a vanilla RNN had to use the hidden state h_t both to produce outputs as well as store memories, h_t gets updated with an affine map and a tanh activation at every timestep, which can easily lead to vanishing or exploding gradients. On the other hand, the LSTM can use the cell state C_t as its "long-term memory," and backpropagation through the long term memory is much easier since the cell states change fairly slowly in a simple way (simply being a moving average of the compute $\tilde{C}_t$'s.). On the other hand, the hidden state in the LSTM can serve as a "short-term memory" and change quickly through time.

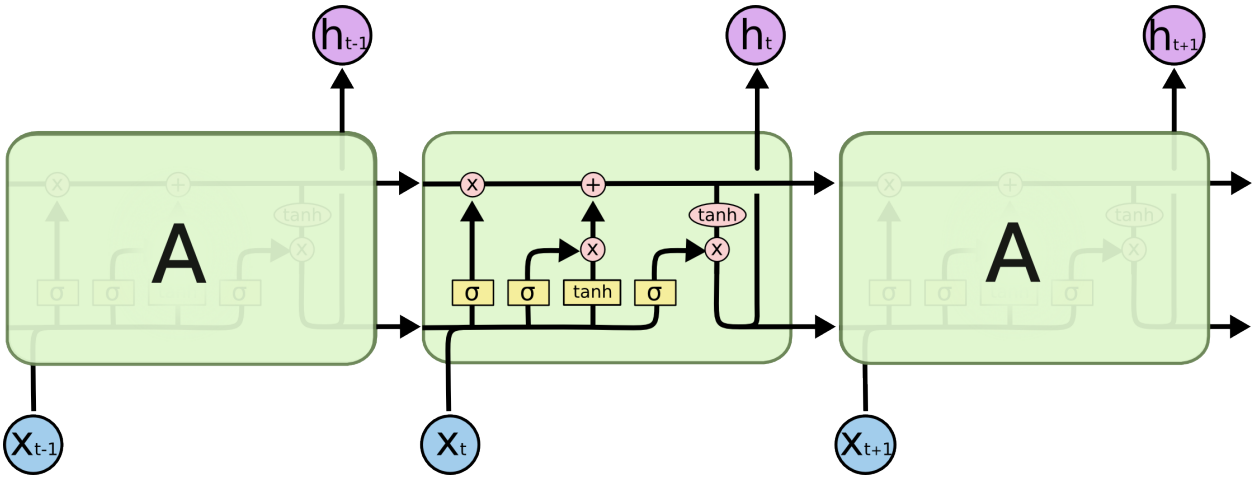


Figure 4: An overview of the LSTM cell

Problem: Backpropagation Through LSTM

Denote the final cost function as J. Compute the gradient $\frac{\partial J}{\partial W^g}$ using a combination of the following gradients,

$$\frac{\partial h_t}{\partial h_{t-1}}, \frac{\partial h_{t-1}}{\partial W^g}, \frac{\partial J}{\partial h_t}, \frac{\partial C_t}{\partial W^g}, \frac{\partial C_{t-1}}{\partial W^g}, \frac{\partial C_t}{\partial C_{t-1}}, \frac{\partial h_t}{\partial C_t}, \frac{\partial h_t}{\partial o_t}$$

Problem: Vanishing Gradient in LSTMs

Using the previously derived gradient, which part of $\frac{\partial J}{\partial W^g}$ allows LSTMs to mitigate the vanishing gradient problem?

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine **Discussion 7**

This discussion covers Attention Mechanisms and Transformers.

1 Attention Mechanisms

For many NLP and visual tasks we train our deep models on, features appear on the input text/visual data often contributes unevenly to the output task. For example, in a translation task, not the entirety of the input sentence will be useful (and may even be confusing) for the model to generate a certain output word, or not the entirety of the image contributes to a certain sentence generated in the caption.

While some RNN architectures we previously covered possess the capability to maintain a memory of the previous inputs/outputs, to compute output and to modify the memory accordingly, these memory states need to encompass information of many previous states, which can be difficult especially when performing tasks with long-term dependencies.

Attention mechanisms were developed to improve the network's capability of orienting perception onto parts of the data, and to allow random access to the memory of processing previous inputs. In the context of RNNs, attention mechanisms allow networks to not only utilize the current hidden state, but also the hidden states of the network computed in previous time steps as shown in Figure 5.

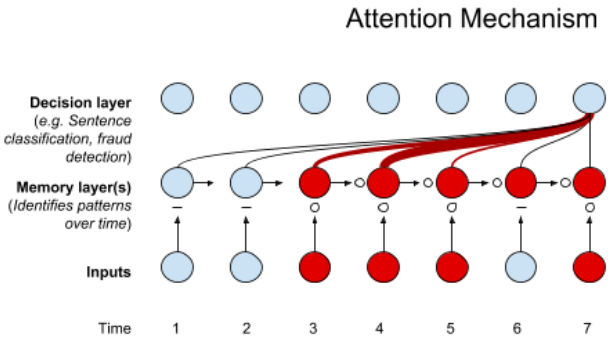


Figure 1: Attention Mechanism Illustrated

1.1 Luong Attention

The Luong attention is one of the commonly used attention mechanism in Neural Machine Translation, which is trained on the task of translating text from source to target language. At each time-step of decoding, the Luong Attention computes a set of alignment (attention) weights a_l based on alignment scores and use this to augment the computation of output probabilities in the translation task.

As shown in Figure 2, we have our encoding states (RNN hidden states when passed a source language into the model first) h_t. At decoding time, we have the original hidden states h_l initially computed with the previous output token and hidden states.

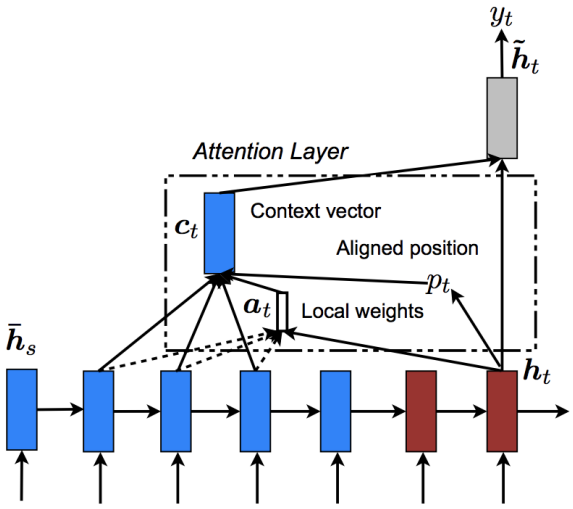


Figure 2: Luong Attention

With h_l and each h_t we compute an alignment score [1], then take a Softmax to obtain attention weights from the alignment scores.

With these attention weights, we compute a context vector c_t, which is a weighted sum of h_s. As we get c_t, we compute a transformation $\tilde{h}_t = \tanh(W_f[h_t; c_t])$ applied on the concatenation of the context vector and the original hidden state. This is then used to compute the final output.

The motivation for this kind of machine translation system was to treat the encoder (the blue part) as like a memory which we can access later during the decoding process (colored red). Most of the early neural machine translation systems ran the encoder with the input sentence to get a hidden state, and then that hidden state was the input to the decoder which needed to generate the resulting sentence. With this model, we are able to use the other hidden state outputs from the encoder, not just the last one.

1.2 Self-Attention in Transformer Networks

Self attention is an attention mechanism introduced in the Transformer architecture which undergoes similar procedures as the Luong attention. The first step of the attention is to compute Q, K, V using different transformations from the original input embedding as shown in Figure 3.

[1] the original paper proposed multiple alignment scores, the general one they propose is $h_l^\top W_a h_t$, another simple 'location' based attention is just computed using the decoding hidden state at time t: $W_a h_l$

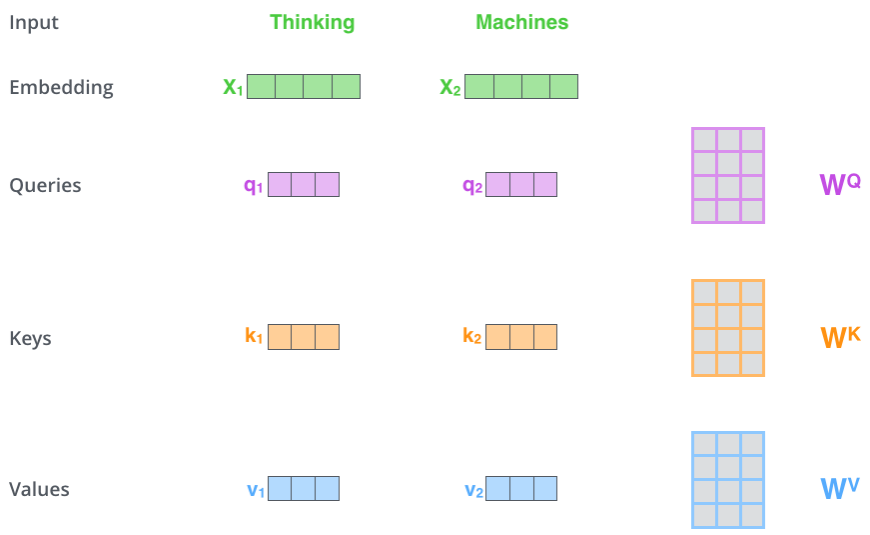


Figure 3: Computing K, Q, V from input embeddings in a Transformer Network

Then, using Q and K, we can compute a dot product as the 'score' of K for Q as shown in Figure 4. Intuitively, Q is the querying term that you would like to find. Its relations for each corresponding K and V pairs (key-value) pairs, can be computed using the key. Note that this dot product is computed across various time steps by matrix multiplication. So we get a score for each K for each Q. We then use a Softmax function to get our attention weights.

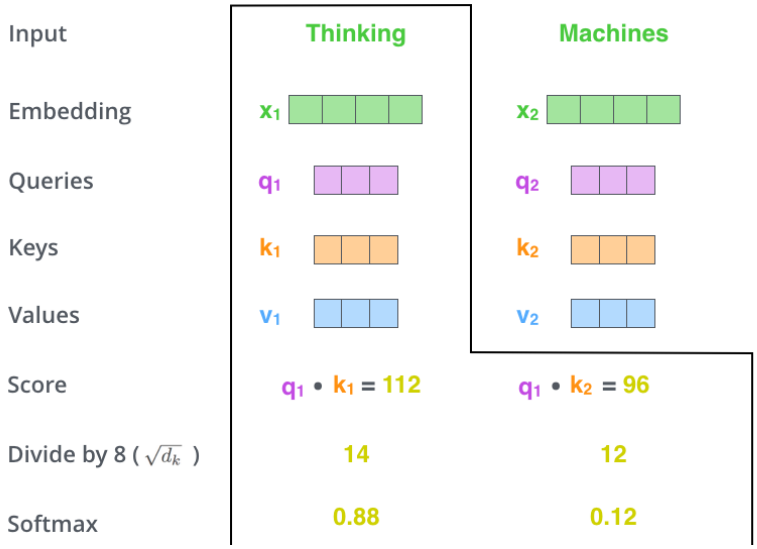


Figure 4: Computing Attention Scores from K, Q, V

Finally, using these weights, we can compute our weighted sum by multiplying the weights with the values. Comparing to the Luong attention, query is analogous to the original h_l, key and query are analogous to the original h_t.

Problem: Attention in RNNs

Explain how we incorporate self-attention into an RNN model at a high-level.

Problem: Generalized Attention in Matrix Form

Consider a form of attention that matches query q to keys $k_1, \ldots, k_t$ in order to attend over associated values $v_1, \ldots, v_t$.
If we have multiple queries $q_1, \ldots, q_l$, how can we write this version of attention in matrix notation?

Problem: Justifying Scaled Self-Attention

In practice, Transformers use a Scaled Self-Attention. Suppose $q, k \in \mathbb{R}^d$ are two random vectors with $q, k \sim \mathcal{N}(\mu, \sigma^2 I)$, where $\mu \in \mathbb{R}^d$ and $\sigma \in \mathbb{R}^+$

1. Define $\mathbb{E}[q^\top k]$ in terms of μ, σ, d
2. Define $Var(q^\top k)$ in terms of μ, σ, d
3. Let s be the scaling factor on the dot product. We would like $\mathbb{E}[q^\top k/s]$ to scale linearly with d. What should s be in terms of μ, σ, d
4. Briefly explain what would happen to the variance of dot product if $s = 1$.

1.3 Tutorials on Attention Networks and Eager Execution on Tensorflow

Here is a tutorial that we recommend you run through in RNNs with `tf.eager` and `keras`: https://colab.research.google.com/github/tensorflow/tensorflow/blob/master/tensorflow/contrib/eager/python/examples/nmt_with_attention/nmt_with_attention.ipynb

2 Transformers

At a high-level, transformers consist of the Transformer Encoder and Transformer Decoders.

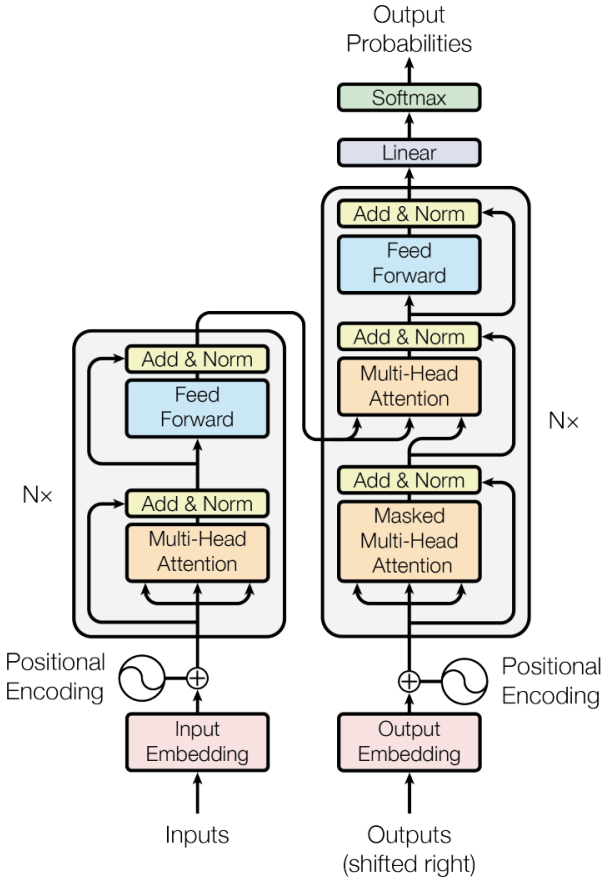


Figure 5: Overview of Transformer architecture

Both operate similarly, except the Transformer Decoder takes x_{target} as input, but Transformer Encoder takes in x_{source} as input. In addition, there are several differences in cross-attention and self-attention operations. In particular, transformers are novel in that they add,

- Positional Encoding: Addresses lack of sequence information
- Multi-headed Attention: Allows querying multiple positions at each layer
- Non-linearities
- Masked Decoding: Prevent attention lookups into the future

2.1 Notations

To ensure a level of clarity, we will let B be the batch size, L_{source} represent the source sequence length, L_{target} be the target sequence length, D represent the model hidden dimension and H represent the number of attention heads.

In particular, transformers receive two sequences as input. The first is $x_{source} \in \mathbb{Z}^{B \times L_{source}}$ and the second is $x_{target} \in \mathbb{Z}^{B \times L_{target}}$. These are integer tensors, and each integer represents a word or token.

2.2 Transformer Encoders

Input & Positional Embedding The source tensor is embedded into the model hidden dimension, and produces a tensor $X_{source} \in \mathbb{R}^{B \times L_{source} \times D}$. We then add a positional encoding that differs for each sequence position in order to enable the model to differentiate the positions in the sequence. In general, we need this information since position of words in a sentence carries information.

Encoder Attention The Encoder Attention is self-attention. Specifically, in Transformer networks, we use the Scaled QKV Attention (not covered explicitly in lecture). In other words, we would like to build a representation of a single sequence such that every position in the sequence has information about every other position in the sequence. In particular, to enable this, we will use the Query-Key-Values (QKV) Attention. Our queries, keys and values will be tensors in $X_{source} \in \mathbb{R}^{B \times L_{source} \times D}$ and weight matrices will be $W_Q, W_K, W_V \in \mathbb{R}^{D \times D}$. Ultimately, we will retrieve,

$$Q = X_{source} W_Q$$
$$K = X_{source} W_K$$
$$V = X_{source} W_V$$

Using Q, K, V, we will compute the attention scores (tensor in $\mathbb{R}^{B \times L_{source} \times L_{source}}$). For each element in the batch, each entry i, j in the matrix would be $\frac{q_i^\top k_j}{\sqrt{D}}$ for scaled dot product attention. Alternatively, we can compute, $\frac{QK^\top}{\sqrt{D}}$. To produce weights over each position in the sequence, we want each score to sum to one over the keys K. To accomplish this, we take a softmax update over the last dimension of the attention scores. Then, to produce the attention update, we multiply these attention weights by our values V,

$$C_{update} = \mathsf{softmax}\left(\frac{QK^\top}{\sqrt{D}}\right) V$$

where $C_{update} \in \mathbb{R}^{B \times L_{source} \times D}$

One of the key changes in Transformers is the *multi-headed attention* mechanism. To turn it into multi-headed attention, we can take any such update matrices and reshape and permute the matrix from shape $B \times L_{source} \times D$ to $B \times H \times L_{source} \times \frac{D}{H}$.

We finally consider padding. In general, we operate on a batch of B sequences, but these sequences may not be the same length. We pad each sequence to L_{source}. To prevent our model from paying attention to padded positions, we add $-\infty$ to attention scores prior to the Softmax of any position that should be ignored.

Feedforward Layer The feedforward layer applies linear transformation to each position, apply a nonlinear activation, then applies a second linear transformation.

2.3 Transformer Decoder

Masked Decoder Self-Attention Masked decoder self-attention is the same as encoder self-attention, but with different masking. In particular, we would like every position to pay attention to all previous positions, but not future positions. To achieve this, we set attention score to $\frac{q_i^\top k_j}{\sqrt{D}}$ if $i \leq j$ and $-\infty$ otherwise.

Encoder-Decoder Attention Encoder-Decoder attention operated similarly as well, except that we have two sequences: (1) generate queries and (2) generate keys-values. Hence, we let $Q = X_{target} W_Q, K = X_{source} W_K, V = X_{source} W_V$, where X_{source} is the output of the transformer encoder on the source sequences.

Problem: Machine Translation

1. What is the reason for positional encoding? How is it typically implemented?
2. What is the advantage of multi-head attention? Give some examples of structures that can be found using multi-head attention
3. For input sequences of length M and output sequences of length N, what are the complexities of (1) Encoder Self-Attention (2) Decoder-Encoder Attention (3) Decoder Self-Attention. Further let k be the hidden dimension of the network
4. Do activation of the encoder depend on decoder activation? How much additional computation is needed to translate a source sequence into a different target language, in terms of M and N?

2.4 Why Transformers

In general, transformers are good for long-range connections, are easy to parallelize and transformers can be made much deeper than RNNs. On the other hand, attention computations are complex to implement and computations take $\mathcal{O}(n^2)$ time.

However, in practice, it turns out the benefits vastly outweigh the downsides, and transformers work better than RNNs and LSTMs in many cases.

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine **Discussion 8**

This discussion covers unsupervised pretraining methods in NLP and imitation learning methods.

1 Unsupervised Pretraining in Natural Language Processing

We will review several techniques for unsupervised pretraining in NLP. The general idea is to use unlabelled text, which is often easily accessible (for example on the internet, in books, other publications, etc...) in order to learn representations of language that can be useful for downstream tasks.

To illustrate why we might expect this to be helpful, we can imagine we want to translate English sentences to French, and are given a labelled dataset of English/French sentence pairs. You can imagine this task would be really difficult if you had no prior knowledge of English, while being much more manageable if you came in with a general understanding of the English language already, which can be learned using unsupervised data (for example, all the English text we see on the internet).

1.1 Word Embeddings

Before in lectures, we didn't worry much about how individual words were represented, often abstracting them away as one-hot vectors for simplicity. Our goal with word embeddings is to map words to real vectors such that distances in the representation are in some sense meaningful, which can make learning much easier for the downstream task. This would imply that "similar" words should be mapped to representations that are close to one another.

1.1.1 Naive Word2Vec

One way to learn embeddings is to optimize the representation to predict nearby words. More precisely, we can consider a *center word* c and try to predict its neighbors in the sentence (*context words* o) via logistic regression. We will associate each two vectors u_w and v_w, and optimize these word embeddings to optimize the likelihood

$$p(o|c) = \frac{\exp\left(u_o^T v_c\right)}{\sum_{w \in V} \exp\left(u_w^T v_c\right)}$$

averaged over all selections of center word c and context word o. Intuitively, this objective means that words that occur together (and so would show up as center/context word pairs o, c) would have higher inner product $u_0^T v_c$. It would also mean that if words a, b were similar in meaning and so were interchangeable, we would expect their embeddings u_a and u_b to be similar (as well as v_a, v_b), since they would appear in similar contexts. The embedding for each word was split into two components u, v to make optimization more tractable, which are simply averaged together to produce the final embedding after training.

1.1.2 Making Word2Vec Tractable

As mentioned in lecture, the normalizing factor $\sum_{w \in V} \exp\left(u_w^T v_c\right)$ involves summing the logits for *every* word in the entire vocabulary, which is very slow given how many words there are in total.

One way to address the issue is to simply train using *binary classification*. We can instead choose to optimize

$$p(o \text{ is the right word}|c) = \frac{1}{1+\exp\left(-u_o^T v_c\right)}.$$

The issue with this approach is that if we only sample positive examples (pairs of words o, c that are actually neighbors of one another), then a trivial solution would be to make u, v to be the same, very large vector for all words, which would cause our binary classifier to always confidently predict that any two words are valid center and context pairs.

To avoid this degenerate solution, we also include *negative samples* in every batch. Instead of just maximizing the likelihood on valid center/context pairs, we'll also reduce the likelihood of words randomly sampled from the dictionary to get the objective

$$\max \sum_{c,o} \left(\log p(o \text{ is the right word}|c) + \sum_{w} \log p(w \text{ is wrong}|c) \right),$$

where the negative examples w are randomly sampled. Now, optimizing this objective will try to push v_c and u_o closer together for valid context/center pairs, while pushing v_c away from u_w for all other words in the vocabulary.

1.2 Pretrained Language Models

One weakness of the previous Word2Vec approach is that we train by simply predicting neighbors from an individual word, and so the representations for a word do not depend on its context after training is complete. This is somewhat limiting, since words can have very different meanings depending in the context. At a high level, one simple way we can embed words in a context-dependent manner is to take a language model (for example an LSTM) trained on some task, and to run a sentence through it, taking the hidden state of the model as the embedding for each word. Since these language models presumably had to use the context in order to solve the task they were trained on, using the hidden state as an embedding should provide context-dependent representations of words. We will briefly discuss two examples of this approach.

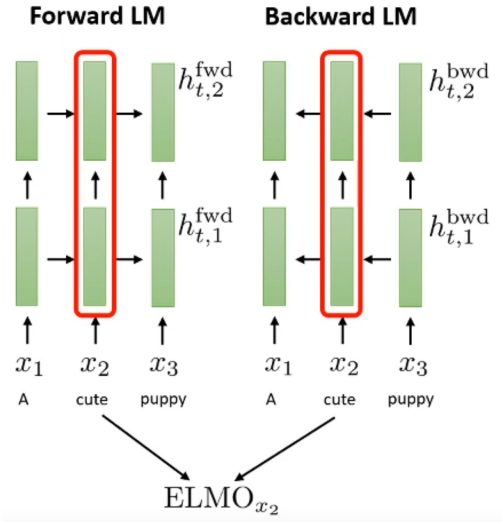


Figure 1: ELMo takes the hidden states in a bi-directional LSTM to generate word embeddings. The LSTMs are both trained via sequence prediction.

ELMo: We note that if we simply ran an LSTM forward through a sentence to generate the embeddings of words, the embedding of each word would only depend on those that came before it, rather than the full

context of the word. ELMo addresses this issue by simply training a bi-directional LSTM (both trained to predict the next/previous word), and concatenating hidden states of both directions together to form an embedding. This approach is very similar to the previous Word2Vec approach in that the embeddings are trained based on predicting nearby words, with the main difference being that the we aren't exactly embedding single words, but rather words in their context.

BERT: Instead of using a bi-directional LSTM like ELMo, BERT relies on a *single* transformer to generate embeddings. While the previous transformers we saw for sequence modeling relied on masked self-attention to avoid peeking into the future, our goal here is to digest the entire context of a word to produce an embedding, which eliminates the need for the mask. However, this presents a complication if we were to try train embeddings to predict the next word like ELMO did. The issue here is that if we did unmasked self-attention, we can already directly see the next word in the input, making prediction completely trivial and preventing useful representations from being learned.

The solution is to simply change the unsupervised task. Instead of predicting the next word in sentence, we instead randomly mask out certain words in the input, and then train the embedding to predict the masked out words. In this way, our model is forced to learn context dependent word-level representations to predict the missing words.

In addition to learning word-level representations by predicting masked out words, BERT also tries to learn *sentence-level* representations. To train this, BERT takes in pairs of sentences, randomly permutes their order, and trains a binary classifier to predict which of the two sentences came first originally. Depending on the downstream task, we can either use the sentence level representation outputted by BERT or the word-level representations in the downstream task. We can use BERT for downstream tasks both by simply finetuning the entire model on the downstream tasks, or taking combinations of the hidden states as fixed representations.

2 Imitation Learning

So far in the course, we have focused on supervised learning for prediction tasks. In imitation learning and reinforcement learning, our goal will now be to solve *control* problems. Control problems will generally involve making sequences of decisions, with each decision affecting the future, in order to accomplish some goal. To formalize this problem, we will introduce **Markov Decision Processes** (MDPs). In imitation learning, we assume we have access to an expert policy that already solves the task we care about, and our goal will be to learn a policy to solve a task by copying these expert demonstrations.

2.1 Markov Decision Processes

A Markov decision process (MDP) is specified by a state space $\mathcal{S}$, an action space $\mathcal{A}$, a transition function $P(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ defining the probability of the next state given the current state and action, and a reward function $r(\mathbf{s}, \mathbf{a})$. A key property in MDPs is the *Markov property*, which similar to Markov chains, states that conditioned on the current state and action, the next state is conditionally independent of anything in the past.

The goal is to learn to take actions in order to maximize the sum of rewards over trajectories in the MDP. Due to the Markov property, all such optimal policies are Markovian (also called stationary processes) and described as conditional distributions $\pi(\mathbf{a}|\mathbf{s})$ over only the current state. This essentially states that the optimal action depends only on the current state, rather than any states or actions further in the past.

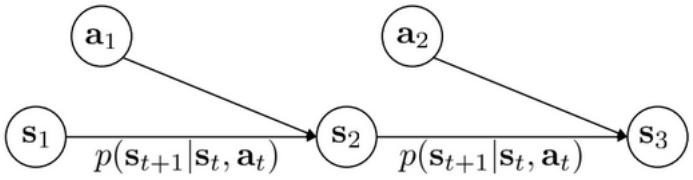


Figure 2: Markov property in MDPs

We can imagine the state $\mathbf{s}$ as capturing the current state of the world, actions $\mathbf{a}$ to be the decisions we make, the transition function $P(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ describing how our actions affect the world around us, and the reward $r(\mathbf{s}, \mathbf{a})$ capturing some notion of success at what we want to accomplish.

While solving imitation learning problems will not explicitly require access to the reward, we should keep in mind that success in imitation learning is not necessarily measured directly in how well we match the expert (as measured in perhaps negative-log-likelihood on the expert dataset like we would consider in supervised learning), but in how well our learned policy actually executes the task we care about. The task we care about is often specified (loosely) as a reward function.

Problem 1: Probability of a trajectory under a Markovian policy

Given a policy $\pi_\theta(\mathbf{a}|\mathbf{s})$, compute the log probability of a trajectory $\tau = ((\mathbf{s}_0, \mathbf{a}_0), (\mathbf{s}_1, \mathbf{a}_1), \ldots, (\mathbf{s}_T, \mathbf{a}_T))$ using the Markov property.

2.1.1 Partially Observed Markov Decision Processes

In practice, we may not be able to access the full state $\mathbf{s}$ that satisfies the Markov property, but instead rely on some limited observations $\mathbf{o}$. For example, when driving a car, we cannot see everything around us, even though things we cannot see may impact how the world behaves around us.

Here, optimal policies may no longer be stationary, but may need to depend on all past observations

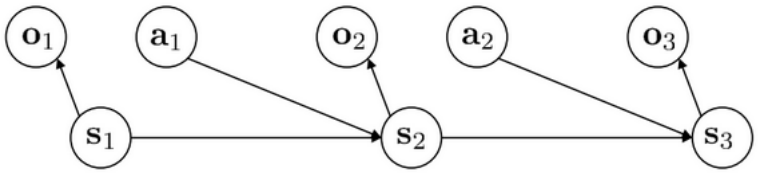


Figure 3: The Markov property still holds for the underlying state $\mathbf{s}$ in POMDPs, but our policies cannot depend on $\mathbf{s}$ directly. Instead, we are forced to take actions based only on the observations $\mathbf{o}$.

$(\pi(\mathbf{a}_t|\mathbf{o}_1 \ldots, \mathbf{o}_t))$. Unless otherwise specified, we're going to assume we're in a fully observed MDP with access to the true state.

2.2 Behavior Cloning

We now describe a very simple approach to imitation learning, which we refer to as **behavior cloning**. We assume that we have a collection of N expert trajectories, where each trajectory τ is a sequence of state-action pairs $((\mathbf{s}_0, \mathbf{a}_0), (\mathbf{s}_1, \mathbf{a}_1), \ldots, (\mathbf{s}_T, \mathbf{a}_T))$. We add every state action pair into our dataset, forming a dataset $\mathcal{D} = \{(\mathbf{s}_0, \mathbf{a}_0), \ldots, (\mathbf{s}_{NT}, \mathbf{a}_{NT})\}$.

Behavior cloning simply trains our policy $\pi(\mathbf{a}|\mathbf{s})$ via supervised learning on this dataset of expert experiences. Each state is considered an input, and the corresponding expert action is considered the label, and if we train with maximum likelihood, our objective becomes

$$\max_\theta \sum_{\mathbf{s},\mathbf{a}\sim\mathcal{D}} \log \pi_\theta(\mathbf{a}|\mathbf{s}).$$

Problem 2: Behavior cloning maximizes likelihood of expert trajectories

Using the decomposition of the probability of a trajectory from the previous problem, show that the behavior cloning objective maximizes the probability of the expert trajectories (assumed to be generated by some expert β that produces a distribution over trajectories) under the learned policy

$$\max_\theta \mathbb{E}_{\tau\sim\beta} \log \pi_\theta(\tau).$$

2.3 DAgger

However, having a policy that assigns high likelihoods to expert trajectories is often not enough to ensure the learned policy attains good performance.

The primary issue is a distribution shift between training and test time. During training, the policy is only being trained on states that were visited by the expert policy. During test time, we actually execute the learned policy, and small mismatches between the learned policy and the expert can potentially take us to new states not seen during training. On these new states that the imitation policy hasn't been trained on, the imitation policy would likely not match the expert's behavior well. Thus, even if the expert policy were capable of recovering from the earlier mistakes and still solve the task, the imitation learning policy would instead continue to make mistakes.

One way to remedy this is to simply do a better job at matching the expert policy to avoid deviating far from the expert trajectories. However, this can require extremely accurate models, and can be tricky to accomplish (even with large amounts of expert data) when the expert policy is non-Markovian (so cannot be matched exactly by a Markovian policy) or if the expert policy is multimodel (and choice our probability distribution for our policy is not expressive enough to match it).

Another approach alter our data distribution we train on to better cover the trajectories we'll encounter during test time. This is the approach taken in the **dataset aggregation (Dagger**) algorithm, which

iteratively collects new trajectories from the current policy, labels those trajectories using the expert policy, and adds the relabeled trajectories to our dataset and retraining. This way, we are constantly updating our state distribution to include our current policy, and we can stop when our imitation policy's distributions stabilize and we obtain good performance in our desired task.

While Dagger can be very effective at mitigating the distribution mismatch issues, it does require the agent to interact with the environment during learning, as well querying the expert to figure out what actions it would take at the new states. Both of these can potentially be costly.

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine

Discussion 9

This discussion focuses on reinforcement learning, covering policy gradient, actor-critic, and Q-learning algorithms.

1 Policy Gradients

Recall from the previous discussion, we formulated sequential decision making problems as **Markov Decision Problems** (MDPs). Our goal with reinforcement learning algorithms is to efficiently solve these MDPs.

We recall that each policy $\pi(a|s)$ induces a distribution over trajectories $\tau = ((s_1, a_1), \ldots, (s_T, a_T))$ in the MDP. Our objective can be formalized as finding a policy π_θ (given by some parameters θ) to maximize the expected sum of rewards over the trajectory.

$$\max_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[R(\tau)] = \mathbb{E}_{\pi_\theta}\left[\sum_{t=1}^{T} r(s_t, a_t)\right].$$

Now that we have an optimization problem, the natural thing to do is to solve it using stochastic gradient ascent, which gives us *policy gradient* algorithms.

1.1 Deriving Policy Gradients

We first consider a more general problem of optimizing the expectation of a random variable with respect to some parameters that control it's distribution. Suppose we have some distributions $p_\theta(X)$ over a random variable X, and we wish to compute the gradient of the expectation of a function of X

$$\nabla_\theta \mathbb{E}_{X\sim p_\theta}[f(X)].$$

We first verify the identity

$$\begin{aligned}\nabla_\theta \log g(\theta) &= \frac{\nabla_\theta g(\theta)}{g(\theta)} \\ \nabla_\theta g(\theta) &= g(\theta)\nabla_\theta \log g(\theta).\end{aligned}$$

Recalling that we can rewrite expectations as integrals and switch the order of integration and differentiation (under some regularity assumptions), we have

$$\begin{aligned}\nabla_\theta \mathbb{E}_{X\sim p_\theta(X)}[f(X)] &= \nabla_\theta \int p_\theta(x) f(x)\, dx \\ &= \int \nabla_\theta p_\theta(x) f(x)\, dx \\ &= \int p_\theta(x) \nabla_\theta \log p_\theta(x) f(x)\, dx \qquad \text{applying previous identity} \\ &= \mathbb{E}_{X\sim p_\theta(X)}[\nabla_\theta \log p_\theta(x) f(x)].\end{aligned}$$

Now that our gradient is written as an expectation, we can obtain unbiased gradient estimates using only samples from $p_\theta(X)$.

Applying this with trajectories τ as the random variable X and the total reward $R(\tau)$ as the function f, we obtain the basic policy gradient

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla \log \pi_\theta(\tau) R(\tau)].$$

Expanding out the log probability of a trajectory and ignoring terms that don't depend on θ (see calculations from last discussion), we have

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)\right) R(\tau)\right].$$

The policy gradient algorithm then alternates between sampling trajectories from the current policy, computing stochastic gradient estimates, and updating the policies.

1.2 Reducing Variance

While the policy gradient algorithm is very simple, it requires us to sample new trajectories for every single update, and if we don't sample many trajectories for each update, the gradient estimate can be very noisy. We will now talk about several ways we can try to reduce the variance of the policy gradient estimate.

1.2.1 Reward to Go

The first trick is to note that actions later in the trajectory cannot have any effect on earlier rewards, so instead of using the rewards across the whole trajectory $R(\tau)$, we can use the *reward-to-go* estimator $R_t(\tau) = \sum_{t'=t}^{T} r(s_{t'}, a_{t'})$, which only includes the rewards starting from timestep t.

To show this formally, we can rewrite $J(\theta)$ as the expected sum over timesteps of rewards and simplify:

$$\begin{aligned}
\nabla_\theta J(\theta) &= \sum_{t'=1}^{T} \nabla_\theta \mathbb{E}_{\pi_\theta}[r_{t'}] \\
&= \sum_{t'=1}^{T} \mathbb{E}_{r_{t'}}\left[r_{t'} \sum_{t=1}^{t'} \nabla \log \pi_\theta(a_t|s_t)\right] \qquad \text{probability of } r_{t'} \text{ only depends on actions up to } t'.
\end{aligned}$$

Rearranging the summations, the new policy gradient update would then be

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) \sum_{t'=t}^{T} r(s_{t'}, a_{t'})\right)\right].$$

1.2.2 Baselines

Another commonly used technique to reduce variance is to subtract a baseline from the return estimate. In lecture, you saw how subtracting a constant baseline (replacing $R(\tau)$ with $R(\tau) - b$ for some constant b) did

not bias the gradient estimate, with proof as follows:

$$\begin{aligned}
\mathbb{E}_{\pi_\theta}\left[\nabla \log \pi_\theta(\tau) b\right] &= b\mathbb{E}_{\pi_\theta}\left[\nabla \log \pi_\theta(\tau)\right] && \text{pull constant outside of expectation} \\
&= b \int \pi_\theta(\tau) \nabla \log \pi_\theta(\tau)\, d\tau \\
&= b \int \pi_\theta(\tau) \frac{\nabla \pi_\theta(\tau)}{\pi_\theta(\tau)}\, d\tau \\
&= b \int \nabla \pi_\theta(\tau)\, d\tau \\
&= b \nabla \int \pi_\theta(\tau)\, d\tau \\
&= b\nabla 1 && \pi_\theta(\tau) \text{ integrates to 1 since it is a probability distribution} \\
&= 0. && \text{derivative of a constant function is 0}
\end{aligned}$$

In lecture, we mentioned that setting $b = \mathbb{E}_{\pi_\theta}[R(\tau)]$ to be the average return of the policy was a reasonable choice to reduce variance. We can provide some intuition why here.

Let us compute the variance of

$$\text{Var}(\nabla \log \pi_\theta(\tau)(R(\tau) - b)) = \mathbb{E}[\nabla \log \pi_\theta(\tau)^2 (R(\tau) - b)^2] - \mathbb{E}[\log \pi_\theta(\tau)(R(\tau) - b)]^2.$$

We note the second term is simply the expected policy gradient squared. Since we know constant baselines can't affect that term (since they leave the gradient estimator unbiased). We will then try to pick the baseline b to minimize the first term, under the simplifying assumption (though not true) that $\nabla \log \pi_\theta(\tau)$ and $R(\tau)$ are independent. Using this independence to separate the expectations, we want to find the baseline b that satisfies

$$\begin{aligned}
\arg\min_b \mathbb{E}[\nabla \log \pi_\theta(\tau)^2 (R(\tau) - b)^2] &= \mathbb{E}[\nabla \log \pi_\theta(\tau)^2] \mathbb{E}[(R(\tau) - b)^2] \\
&= \arg\min_b \mathbb{E}[(R(\tau) - b)^2].
\end{aligned}$$

The minimizing constant b is simply the expectation $\mathbb{E}[R(\tau)]$, which motivates why we subtract the average return as a simple constant baseline. In general, due to the untrue independence assumption we made, there are no guarantees that this baseline minimizes variance (or even reduces variance at all), but it tends to work reasonably well in practice.

Problem 1: State-dependent baselines

Show that a subtracting state-dependent baseline $b(s_t)$ also doe not bias the policy gradient estimator. Hint: break up the summation over the timestep t and consider each timestep separately.

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) \left(\sum_{t'=t}^{T} r(s_{t'}, a_{t'}) - b(s_t)\right)\right)\right].$$

Similar in spirit to subtracting out the average return as a baseline, we can thus use the expected future return from state s_t (denoted $V(s_t)$ in the next section) as a baseline to reduce variance.

2 Value functions and Q-learning

For a given policy π, we can define the time-dependent state value function $V_t^\pi(s)$ and the state-action value function $Q_t^\pi(s,a)$ as expectations of future rewards conditioned on being at state s at time t (and taking action a for Q_t^π.).

$$V_t^\pi(s) = \mathbb{E}_\pi\left[\sum_{t'=t}^{T} r_t | s_t = s\right]$$
$$Q_t^\pi(s,a) = \mathbb{E}_\pi\left[\sum_{t'=t}^{T} r_{t'} | s_t = s, a_t = a\right].$$

2.1 Discounting

In the infinite horizon setting ($T = \infty$), future rewards no longer depend on the absolute timestep t, so we can then simply use non-time dependent value functions Q^π and V^π. However, infinite horizons can lead to unbounded future returns, so we introduce a *discount factor* $\gamma \in (0,1)$, which tells to prioritize more immediate rewards. The discounted value functions are then

$$V^\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s\right]$$
$$Q^\pi(s,a) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a\right].$$

We note that these two functions are also related by $V^\pi(s) = \mathbb{E}_{a\sim\pi(a|s)}[Q^\pi(s,a)]$ by the tower property of conditional expectations. In practice, we'll often use discounted returns even if there is a finite horizon T in order to reduce variance and make things easier to estimate, both in value function learning and directly in policy gradient methods.

2.2 Learning Value Functions via the Bellman Equations

To learn value functions, one straightforward approach would be to simply to take Monte Carlo estimates and regress onto the observed returns from sampled trajectories. Similar to our previous policy gradient algorithms, it will suffer due to the returns in trajectories having high variance. In practice, we will often learn value functions with a *dynamic programming* approach based on a set of consistency equations the value functions satisfy, known as the **Bellman equations**.

Separating out the reward at the first timestep from the future rewards, we have

$$\begin{aligned} V^\pi(s) &= \mathbb{E}_\pi[r_0] + \mathbb{E}_\pi\left[\sum_{t=1}^{\infty} \gamma^t r_t | s_0 = s\right] \\ &= \mathbb{E}_\pi[r_0] + \mathbb{E}_{s_1}\left[\mathbb{E}_\pi\left[\sum_{t=1}^{\infty} \gamma^t r_t | s_1 = s_1\right] | s_0 = s\right] \\ &= \mathbb{E}_\pi[r_0] + \gamma \mathbb{E}_{s_1}\left[V(s_1) | s_0 = s\right]. \end{aligned}$$

This tells us the value at the current state is simply the expected immediate reward plus the discount times the expected value of the next state.

Similarly for Q-functions, we can derive

$$Q^\pi(s,a) = r(s,a) + \gamma \mathbb{E}_{s_1}\left[\mathbb{E}_{a_1\sim\pi(\cdot|s_1)}[Q^\pi(s_1,a_1)]\right].$$

If we consider a tabular setting (where we compute exact values for every state), we can find the value function of the policy by satisfying the Bellman equations. Concretely, we can take transitions (s, a, r, s') from the policy's trajectories and update $V(s)$ towards the target value $r + \gamma V(s')$ (with a step size α:

$$V(s) \leftarrow V(s) - \alpha(r(s, a) + \gamma V(s') - V(s)).$$

Note that learning the state value function requires *on-policy* samples from the policy being evaluated, due to the expectation over the immediate action that leads to the immediate reward.

Alternatively, we can learn the Q-values instead with

$$V(s) \leftarrow Q(s, a) - \alpha(r(s, a) + \gamma \mathbb{E}_{a' \sim \pi(a'|s')} Q(s', a') - Q(s, a)),$$

and note that this can be learned with *off-policy* data, as only the future values $Q(s', a')$ depend on the policy being evaluated.

2.2.1 Learning with Function Approximation

We see that the tabular updates above resemble taking a gradient step on the squared error between the current value $V(s)$ and a regression target $r(s, a) + \gamma V(s'))^2$ (similarly for the Q-values). We can straighforwardly extend these updates to a setting where our value functions are parameterized by some parameters ϕ (for example a deep neural network), and take a gradient step on the objective (with some loss function L such as squared error)

$$\mathbb{E}_{s,a,s' \sim \pi} \left[L(V^\pi_\phi(s), r(s, a) + \gamma V_{\bar{\phi}}(s')) \right].$$

Here, $\bar{\phi}$ is an identical copy of the current parameters ϕ, to indicate that we do not pass gradients through the target value $V_{\bar{\phi}}(s')$. Due to this moving target, doing these updates is **not** doing gradient descent on any fixed objective, since the regression targets are constantly changing as we optimize.

In practice, these updates are often unstable due to these moving target values, so we often slow down how the target parameters $\bar{\phi}$ change. One way is to simply keep $\bar{\phi}$ fixed for multiple iterations of updates to ϕ, before copying the current parameters to the target. Another way to slow down the changing target values is to update $\bar{\phi}$ as an exponential moving average of ϕ.

We again emphasize that state-action values, unlike the state values, can be learned directly from off-policy data, leading to taking updates in the direction given by

$$\mathbb{E}_{s,a,s' \sim D} \left[\nabla_\phi L(Q^\pi_\phi(s, a), r(s, a) + \gamma \mathbb{E}_{a' \sim \pi(a'|s} [Q_{\bar{\phi}}(s', a')]) \right].$$

Here, the dataset D can include transitions from other policies, for example the trajectories from previous policies (known as *experience replay*). By reusing experiences from past policies to learn the Q-function of the current policy, learned Q-values can potentially make reinforcement learning much more sample-efficient.

2.3 Using Value Functions in Actor-Critic Algorithms

Assuming we had the exact state values $V(s)$ or state-action values $Q(s, a)$, we can go back to our policy gradient algorithm and verify that the following gradient estimators are all unbiased.

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t) Q_t^\pi(s_t, a_t))\right)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)(r(s_t, a_t) + \gamma V_{t+1}^\pi(s_{t+1}))\right)\right] \\
&= \mathbb{E}_{\pi_\theta}\left[\left(\sum_{t=1}^{T} \nabla \log \pi_\theta(a_t|s_t)(r(s_t, a_t) + \gamma V_{t+1}^\pi(s_{t+1}) - \underbrace{V_t^\pi(s_t)}_{\text{state dependent baseline}})\right)\right].
\end{aligned}
$$

Note that if we did have the exact value functions, these would lead to lower variance estimates for our policy gradient, since we have replaced noisy estimates of future returns with their exact expectations. Using learned value functions V_ϕ^π and Q_ϕ^π in the above updates leads to the broad class of *actor-critic* algorithms, where we update our policy (the actor) using learned value functions (the critic). Since we use learned values instead of the true conditional expectations, this will typically bias our policy gradient estimates, but often improves policy learning due to the much decreased variance in the updates.

3 Q-Learning

In the previous section, we discussed value functions and how they could be combined with policy gradient updates to solve RL problems. In Q-learning, we will explore how to discuss how we can learn to solve RL problems without explicitly keeping track of a policy, but instead only keeping track of a value function.

While we previously learned value functions for the current policy π, we can instead try to directly learn the values associated with the *optimal policy* denoted π^*. Let Q^* denote the state-action value of the optimal policy. Then, from our previous section, we know it satisfies a Bellman equation

$$Q^*(s,a) = r(s,a) + \gamma \mathbb{E}_{s'}\left[\mathbb{E}_{a' \sim \pi^*(\cdot|s')}[Q^*(s',a')]\right].$$

Now we note that the optimal policy π^* must also take actions that maximize future values. For simplicity, if π^* is a determnistic policy mapping states to actions, then we must have $\pi^*(s) = \arg\max_a Q^*(s,a)$. Substituting this in, we have

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

which will form the basis of Q-learning algorithms. Q-learning can loosely be seen as always trying to learn the value of a deterministic policy that always take the action that maximizes the current Q^*-values.

Note that Q-learning requires us to be able to compute the optimal action over the Q-values, which typically limits us discrete action spaces (though we can get around this with either approximations to the max or assuming additional structure over the Q-functions).

The classic Q-learning algorithm will then alternate between sampling a new transitions using a policy derived from the Q-function and updating the Q-function on that new transition. In deep Q-learning algorithms, we'll often use experience replay, where we sample batches of previously seen experiences to update along

$$\mathbb{E}_{s,a,s' \sim D}\left[\nabla_\phi L(Q^*_\phi(s,a), r(s,a) + \gamma \max_{a'}[Q^*_{\bar{\phi}}(s',a')])\right].$$

As mentioned previously, this can help make Q-learning much more sample efficient as well as helping stabilize the updates.

3.1 Exploration in Q-Learning

As Q-learning implicitly gives us a deterministic policy, exploration can be very limited if we directly use that policy for exploration. Intuitively, we can fall into a situation where we think we know the optimal behavior, and if we always act optimally according to our beliefs, we never experience anything new that might lead us to better returns.

One simple approach to get around this is the ϵ-greedy strategy. Instead of always taking the optimal action, we will instead take a completely random action with probability ϵ and act optimally otherwise. This way, the random actions can eventually lead us to discover better behaviors.

Another approach is known as Boltzmann exploration, where we explore according to

$$\pi(a|s) \propto \exp\Big(Q^*_\phi(s,a)\Big),$$

which is a random policy that weights each action according to how well we think the action will do.

We again note that unlike policy gradient algorithms, we can trivially explore using other policies since our goal is to learn Q-values directly, which can be done with off-policy data.

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine

Discussion 10

This discussion focuses on deep unsupervised methods.

1 Generative Models

Generative models represent the full joint distribution $p(x, y)$ where x is an input sample and y is an output. In deep learning, we are interested in generating new samples that generalize a given dataset. In particular, this process may not require labeled data, as the goal is to understand some representation of the dataset distribution.

Broadly speaking, deep generative models include autoregressive models, autoencoder models and generative adversarial networks. In this discussion, we discuss deep generative models from the context of autoregressive models and autoencoder models.

2 Autoregressive Generative Models

An autoregressive model is a generative model that generates one sample at a time conditioned on its prior predictions.

In general, training autoregressive generative models divide up x into dimensions $x_1, \ldots, x_n$, and discretize them into k values. We finally model $p(x)$ via the chain rule. For example, when we try to generate one pixel at a time *conditioned* on pixel values from prior predictions, consider a $n \times n$ image with pixels in some order $(x_1, \ldots, x_{n^2}$. Then, we can define the model as,

$$p_\theta(x_1, \ldots, x_{n^2}) = \prod_{i=1}^{n^2} p_\theta(x_i | x_1, \ldots, x_{i-1})$$

Since the distribution $p(x_i|x_1, \ldots, x_{i-1})$ is complex, we model the distribution using a neural network, f_θ. Using the same example of generating pixels, we can begin at one corner, and proceed diagonally throughout the 2D spatial map, and use f_θ to sample pixels one at a time by conditioning on previously generated pixels.

Unfortunately, this work of generating and sampling each pixels conditioned on prior generated pixels is expensive. Instead, in practice, we use either **PixelRNN**, **PixelCNN** or **Pixel Transformer**.

PixelCNN Uses a CNN to model the probability of a pixel, given *previous* pixels. PixelCNN is slow at generating images, because there is a pass through the entire network for each pixel. But it is fast to train because there is no recurrence (only a single pass for the image) since the spatial maps are known in advance. During training the convolutions must be masked to ignore pixels at the same or later position in the pixel generation order.

PixelRNN PixelRNN uses RNN (or LSTM) to generate images. PixelRNN remembers the state from more distant pixels using the recurrent states. In particular, the PixelRNN uses recurrence instead of the 3×3 convolutions to allow long-range dependencies, and can generate a full row of pixels in one pass.

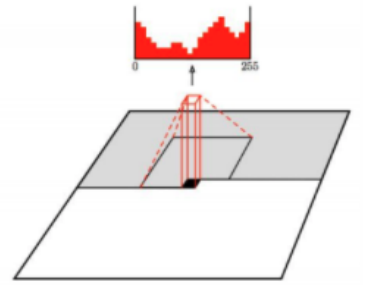


Figure 1: PixelCNN. Generate images from the corner, and dependency on previous pixels are modeled using a CNN over the context region

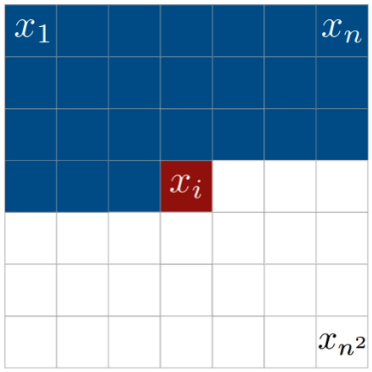


Figure 2: PixelRNN. This figure shows conditional probability. x_i is related to the previous points (blue part) and is unrelated to the point after it (white part)

Training is generally longer, though, due to the recurrence involved, since each row has its own hidden state in the LSTM layers.

PixelTransformer Pixel Transformers are similar to PixelCNN and PixelRNN, but they use a multi-headed attention network.

Problem 1: PixelCNN vs. Pixel RNN

What is the main difference between PixelCNN and PixelRNN? In particular, comment on:

- Run-time of PixelCNN and PixelRNN at training time
- Run-time of PixelCNN and PixelRNN at test time
- Generation of Pixels

3 Autoencoders

Autoencoders are methods to train a network to encode an image into some hidden state and then decode that image as accurately as possible from the hidden state. During this process, we force the autoencoder to learn a structured representation. Generally, autoencoders comprise of an *encoder* and a *decoder*, with a hidden state z. They are typically implemented as neural networks to compress the input into a smaller hidden state, and then decompressed through the decoder. Once the training is done, we can discard the second part of the network, and use z as the useful features for the original data.

These learned latent representations can be used on downstram tasks, like classification. For example, the VAE paper shows that VAE achieves adversarial robustness in downstream tasks on colorMNIST and CelebA datasets.

In particular, there are several key mechanisms to force the autoencoder to learn a structured representation of the data,

1. Dimensionality: Force the hidden state to be smaller than the input/output, so the network must compress information
2. Sparsity: Force the hidden state to be sparse, so the network must be compressed
3. Denoising: Corrupt the input with noise, and force the autoencoder to learn to distinguish noise from the signal
4. Probabilistic Modeling: Forces the hidden state with a prior distribution

In practice, Autoencoders are far less used today, since there exists better alternatives for both representation learning (VAEs, contrastive learning) and generation (GANs, VAEs, autoregressive models).

Bottleneck Autoencoder Bottleneck Autoencoder can be viewed as non-linear dimensionality reduction, and can be used as since dimensionality is lower and there are various algorithms tractable in low-dimensional spaces. This design is antiquated and rarely used. The idea is simple to implement, but reducing dimensionality often fails to provide the structure we want. When the number of hidden dimensions is larger than input/output, we call it *overcomplete*, and this may learn the identity function.

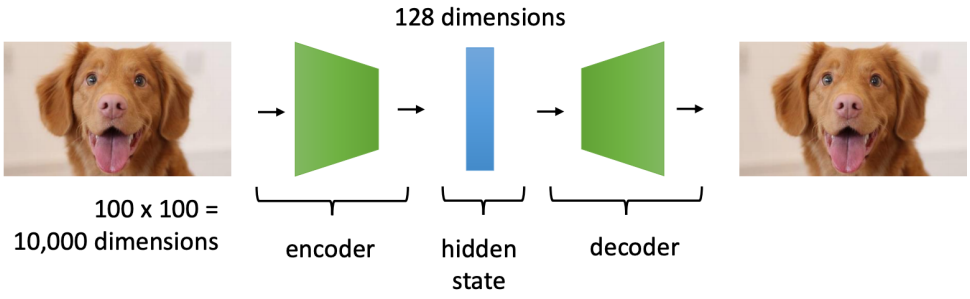


Figure 3: Classical bottleneck architecture reducing 10000 dimensions to 128 dimensions

Denoising Autoencoder Denoising Autoencoders corrupt the input with noise and runs a Bottleneck Autoencoder. However, there are many variants on this idea. In practice, it is unclear which layer to choose for the bottleneck, and there are some ad-hoc choices (e.g., how much noise to add).

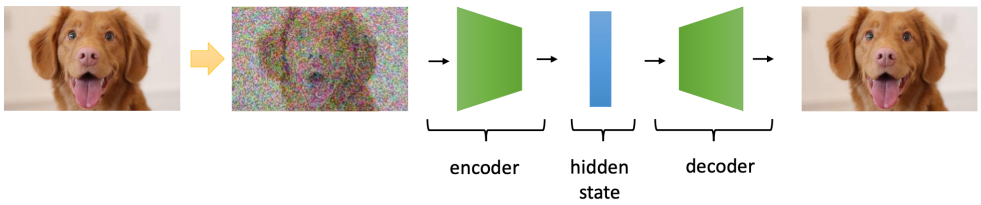


Figure 4: Denoising architecture with a Bottleneck

Sparse Autoencoder Sparse Autoencoder originates from sparse coding theory in the brain, and is an attempt to describe the input with a sparse representation, by letting most values to zero. In this autoencoder, the dimensionality may be very large, and uses a sparsity loss, $\sum_{j=1}^{D} |h_j|$. In practice, choosing the regularizer and adjusting hyperparameters can be very hard.

4 Latent Variable Models

Formally, a latent variable model p is a probability distribution over observed variables x and latent variables z (variables that are not directly observed but inferred), $p_\theta(x, z)$. Because we know z is unobserved, using learning methods learned in class (like supervised learning methods) are unsuitable.

Indeed, our learning problem of maximizing the log-likelihood of the data turns from,

$$\theta \leftarrow \arg\max_\theta \frac{1}{N} \sum_{i=1}^{N} \log p_\theta(x_i)$$

to the following,

$$\theta \leftarrow \arg\max_\theta \frac{1}{N} \sum_{i=1}^{N} \log \int p_\theta(x_i|z)p(z)dz$$

where we recognize $p(x) = \int p(x|z)p(z)dz$. Unfortunately, the integral is intractable, but we will discuss ways to find a tractable lower bound.

4.1 Varational Autoencoders (VAE)

The VAE uses the autoencoder framework to generate new images. For the following description of encoder and decoder of the VAE, let us assume our input x is a 28×28 photo of a handwritten digit in black-and-white, and we wish to encode this information into a latent representation of space z.

Encoder Encoder maps a high-dimensional input x (like the pixels of an image) and then (most often) outputs the parameters of a Gaussian distribution that specify the hidden variable z. In other words, they output $\mu_{z|x}$ and $\Sigma_{z|x}$. We implement this as a deep neural network, parameterized by ϕ, which computes the probability $q_\phi(z|x)$. We can sample from this distribution to get noisy values of the representation z.

Decoder Decoder maps the latent representation back to a high dimensional reconstruction, denoted as $\hat{x}$, and outputs the parameters to the probability distribution of the data. We implement this as another neural network, parametrized by θ, which computes the probability $p_\theta(x|z)$. Following the digit example, if we represent each pixel as a 0 (black) or 1 (white), probability distribution of a single pixel can be then represented using a Bernoulli distribution. Indeed, the decoder gets as input the latent representation of a digit z and outputs 784 Bernoulli parameters, one for each of the 784 pixels in the image.

Training VAEs To train VAEs, we find parameters that maximize the likelihood of the data,

$$\theta \leftarrow \arg\max_\theta \frac{1}{N} \sum_{i=1}^{N} \log \int p_\theta(x_i|z)p(z)dz$$

This integral is intractable. However, we can show that it is possible to optimize a tractable lower bound on the data likelihood, called the Evidence Lower Bound (ELBO),

$$\mathcal{L}_i = \mathbb{E}_{z\sim q_\phi(z|x_i)}[\log p_\theta(x_i|z)] - D_{KL}(q_\phi(z|x_i)||p(z))$$

Problem 2: Blurry Images

Why do VAEs typically produce blurry images?

4.2 Variational Inference

In this subsection, we will derive the variational approximation in discrete form, and discuss the re-parametrization trick.

Problem 3: Latent Variable Model

Write out the log-likelihood objective of a discrete latent variable model.

Problem 4: Variational Approximation

Show that

$$\sum_{i=1}^{N} \log p_\theta(x_i) \geq \sum_{i=1}^{n} \mathbb{E}_{q(z|x_i)}[\log p_Z(z) - \log q(z|x_i) + \log p_\theta(x_i|z)]$$

Hint: Use Jensen's Inequality, which states, $\log \mathbb{E}[X] \geq \mathbb{E}[\log X]$

Problem 5: Variational Approximation Optimization

To optimize the Variational Lower Bound derived in the previous problem, which distribution do we sample z from?

Combining it with Entropy Recall the entropy function,

$$H(p) = -\mathbb{E}[\log p(x)] = -\int_X p(x) \log p(x) dx$$

and also recall KL-Divergence,

$$D_{KL}(q||p) = \mathbb{E}\left[\log \frac{q(x)}{p(x)}\right] = -E[\log p(x)] - H(q)$$

We can show that, our derived approximation can be reformulated as the Evidence Lower Bound (ELBO),

$$\mathcal{L}_i = \mathbb{E}_{z \sim q_\phi(z|x_i)}[\log p_\theta(x_i|z)] - D_{KL}(q_\phi(z|x_i)||p(z))$$

Re-Parametrization Trick The re-parametrization trick allows us to break $q_\phi(z|x)$ into a deterministic and stochastic portion, and re-parametrize from $q_\phi(z|x)$ to $g_\phi(x, \epsilon)$. In fact, we can let, $z = g_\phi(x, \epsilon) = g_0(x) + \epsilon \cdot g_1(x)$ where $\epsilon \sim p(\epsilon)$. This reparametrization trick is simple to implement, and has low variance [1]

Problem 6: Reparametrization Example

Let us get an intuition for how we might use re-parametrization in practice. Assume we have a normal distribution q, parametrized by θ, such that, $q_\theta(x) \sim \mathcal{N}(\theta, 1)$, and we would like to solve,

$$\min_\theta \mathbb{E}_q[x^2]$$

Use the re-parametrization trick on x to derive the gradient.

4.3 Normalizing Flows

In Flows, we wish to map simple distributions (easy to sample and evaluate densities) to complex ones (learned via data), and they describe the transformation of a probability density through a sequence of

[1] See Appendix of this paper for more information

invertible mappings. Let us consider a directed, latent-variable model over observed variables X and latent variables Z.

In practice, we learn an invertible mapping $f_\theta : \mathbb{R}^n \to \mathbb{R}^n$ from z to x.

$$x = f_\theta(z)$$
$$z = f_\theta^{-1}(x)$$

We then, maximize the likelihood of $p(x)$.

Hence, we can interpret *Normalizing Flow*, as (1) we would like the change of variables give a normalized density at $\mathcal{N}(0, 1)$ after applying an invertible transformation, and (2) that the invertible transformations can be composed with each other to create more complex invertible transformations.

Problem 7: Flow Objective

In Flows, our training objective is to let maximize the log-likelihood of x, where we have,

$$x = f_\theta(z)$$
$$z = f_\theta^{-1}(x)$$

Write out the training objective explicitly, then use change of variables to derive the Normalizing Flow objective. Also derive the properties that f_θ must satisfy for practical flows.

There are two main flow models we discuss in class,

1. Nonlinear Independent Components Estimation (NICE)
2. Real Non-Volume Preserving Transformation (Real-NVP)

NICE NICE model composes two invertible transformations: additive coupling layers and rescaling layers. The coupling layer in NICE partitions a variable z into two disjoints subsets, $z_{1:d}$ and $z_{d+1:n}$. Then it applies the following forward mapping,

1. $x_{1:d} = z_{1:d}$ (identity mapping)
2. $x_{d+1:n} = z_{d+1:n} + g_\theta(z_{1:d})$ where g_θ is the neural net

and the following inverse mapping,

1. $z_{1:d} = x_{1:d}$ (identity mapping)
2. $z_{d+1:n} = x_{d+1:n} - h_\theta(z_{1:d})$ where h_θ is the neural net

Here, notice that the Jacobian of the forward map is lower triangular, whose determinant is simply the product of the elements on the diagonal, which is 1. Also, note that, then we have that the mapping is *volume preserving*, meaning that the transformed distribution p_x will have the same "volume" compared to the original one p_z.

Real-NVP Real-NVP adds scaling factors to the transformation,

$$x_{d+1:n} = \exp\big(h_\theta(z_{1:d})\big) \odot z_{d+1:n} + h_\theta(z_{1:d})$$

where $\odot$ represents element-wise product. This results in a non-volume preserving transformation.

Problem 8: Real-NVP Determinant

Determine the determinant of the Jacobian of the forward map of the Real-NVP. In other words, find,

$$|\det\left(\frac{df(z)}{dz}\right)|$$

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine **Discussion 11**

This discussion focuses on generative adversarial networks and adversarial attacks.

1 Generative Adversarial Networks

In the previous discussion, we focused on training generative models by directly maximizing the likelihood of the inputs we observe in our training data. This week, we'll consider an alternative method for training generative models for high dimensional inputs that does not explicitly compute likelihoods, but will train the generative model to fool a learned a learned *discriminator* network, which is being trained to distinguish between real and generated inputs.

1.1 The GAN Game

For GANs, we will typically generate samples by first sampling vectors from some fixed noise distribution $Z \sim p(Z)$, and then passing them through a (deterministic) generator function G_θ (parameterized by θ) to obtain samples $\tilde{X} = G_\theta(Z) \sim p_G(\tilde{X})$.

Our discriminator will be a binary classifier D_ϕ (parameterized by ϕ), which will be trained to discriminate between the generated samples $\tilde{X}$ and the true data distribution $X \sim p(X)$.

We can write view the training of GANs as solving a two-player game given by

$$\begin{aligned}\min_{G_\theta} \max_{D_\phi} &= E_{X\sim p(X)}[\log D_\phi(X)] + E_{\tilde{X}\sim p_G(\tilde{X})}[\log\Big(1 - D_\phi(\tilde{X})\Big)] \\ &= E_{X\sim p(X)}[\log D_\phi(X)] + E_{Z\sim p(Z)}[\log\big(1 - D_\phi(G_\theta(Z))\big)].\end{aligned}$$

If we hold the generator G_θ fixed, then training the discriminator D_ϕ would be exactly the same as training a normal binary classifer. If we hold the discriminator D_ϕ fixed, then training the generator is simply optimizing the generator to generate samples that the discriminator thinks are valid inputs.

Note that this objective does not require us to compute the likelihoods $p_\theta(\tilde{x})$ for any generated image (or any other image) under the generator's distribution, which gives us more flexibility unlike our previously covered latent variable models which required tractably computable log-likelihoods to train.

In practice, we optimize GANs by alternating taking gradient steps on the discriminator and generator, rather than fully optimizing the discriminator before updating the generator as this minimax game suggests.

1.2 GANS with the "perfect" discriminator

To gain intuition for what training a GAN *should* do, we consider an idealized setting where our discriminator is infinitely expressive and is fully optimized to convergence for every generator update.

In this case, for any input x and fixed generator G, the optimal discriminator D^* assigns probability

$$D^*(x) = \frac{p(x)}{p(x) + p_G(x)}.$$

Problem 1: Optimal Discriminator

Show that the optimal discriminator probability $D^*(x)$ is given by the expression above.

We can substitute this optimal discriminator into our two player game and reduce to a single optimization over the generator G_θ as

$$\min_{G_\theta} E_{X\sim p(X)}\left[\log\left(\frac{p(X)}{p(X)+p_G(X)}\right)\right] + E_{\tilde{X}\sim p_G(\tilde{X})}\left[\log\left(\frac{p_G(\tilde{X})}{p(\tilde{X})+p_G(\tilde{X})}\right)\right].$$

Defining $q(x) = \frac{p(x)+p_G(x)}{2}$, then we can rewrite the objective as

$$\min_{G_\theta} \underbrace{E_{X\sim p(X)}\left[\log p(X) - \log q(X)\right]}_{D_{KL}(p(x)\|q(x))} + \underbrace{E_{\tilde{X}\sim p_G(\tilde{X})}\left[\log p_G(\tilde{X}) - \log q(\tilde{X})\right]}_{D_{KL}(p_G(x)\|q(x))} + \text{constant}.$$

We recognize this objective as being (up to the additive constant that doens't matter for optimization) to precisely be the Jensen-Shannon divergence between the true data distribution $p(X)$ and the generator distribution $p_G(\tilde{X})$. This shows that in the ideal setting with a perfect discriminator, training a GAN does in fact optimize the generator distribution to be close to the data distribution (as measured by the Jensen-Shannon divergence).

1.3 Training GANs in Practice

Of course, we will generally not find the optimal discriminator (due to computational limitations and representational limitations on the discriminator architecture), so we will generally not precisely be minimizing the Jensen-Shannon divergence when training GANs. It also turns out that having a perfect discriminator and directly trying minimize the Jensen-Shannon can be very undesirable for training GANs. In Figure 1, we see that the ideal discriminator values (red) are essentially constant on all the generated data, so the gradient for the generator would be extremely small, which can make optimizing the generator extremely slow.

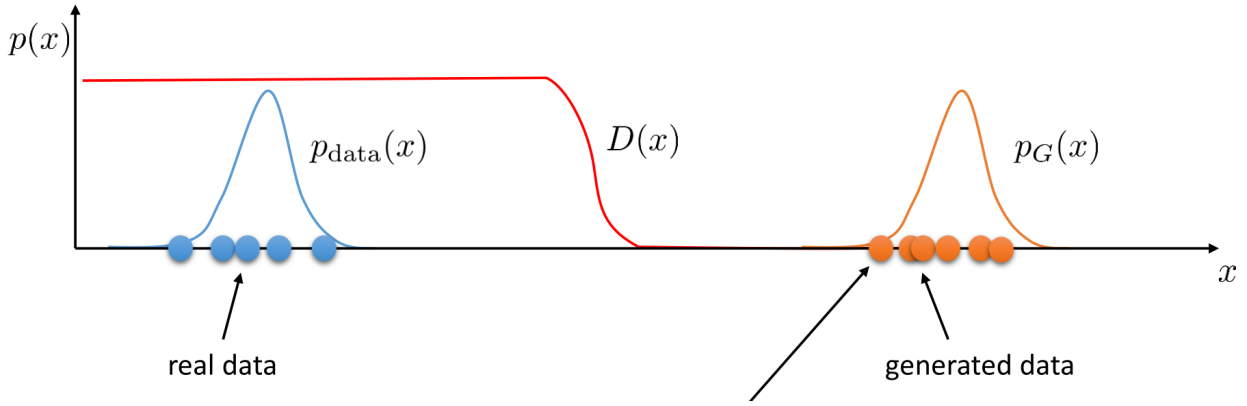


Figure 1: Perfect discriminator values (red line) with real (blue) and generated (orange) inputs in a 1D example. The generator will have very little gradient signal on how to improve, making this discriminator undesirable.

The key idea to improve GAN training is that we don't actually care how well the discriminator does by itself, but rather we only care that it provides a useful signal to improve the generator. A common way we can accomplish this is to restrict the expressivity of discriminator, often by enforcing some additional smoothness condition. As we see in Figure 2, the smoother discriminator values in green can provide a more useful learning signal for the generator.

Wasserstein GANs (which motivate imposing a Lipschitz constraint on the discriminator as minimizing the Wasserstein distance between the generator and real data distributions), gradient penalty GANs, and

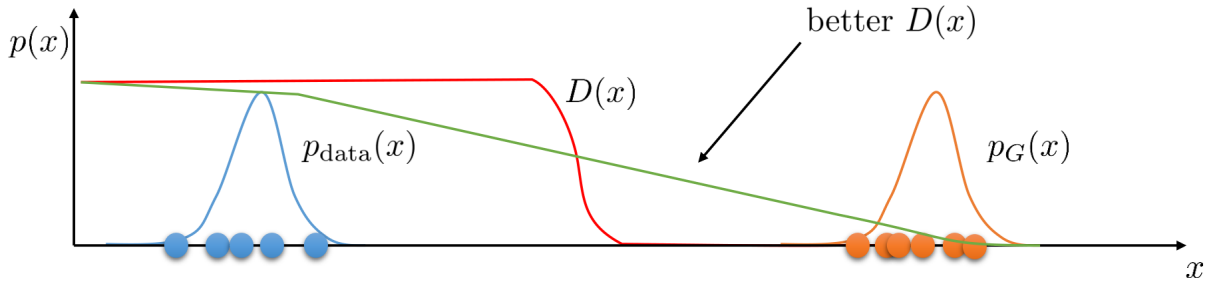


Figure 2: A discriminator constrained to be Lipschitz continuous (green line) with real (blue) and generated (orange) inputs in a 1D example. The generator can now follow the slope of the green line to move generated samples towards the real data.

spectral normalized GANs all enforce different notions of smoothness on the discriminator in order to make learning the generator easier.

Other tricks to avoid vanishing gradients include using real-valued discriminators like in Least-squares GAN (which turns out to be equivalent to minimizing the Pearson χ^2 divergence under ideal settings), and instance noise, which adds noise to the inputs to smooth out the densities of both the real and generated data distributions, improving the learning signal.

1.3.1 Techniques for Lipschitz Continuity

Recall from lecture that we can the Wasserstein distance $W(p, p_G)$ as

$$W(p, p_G) = \sup_{\|f\|_L \leq 1} E_{X \sim p(X)}[f(x)] - E_{\tilde{X} \sim p_G(\tilde{X})}[f(x)],$$

where f is restricted to be a 1-Lipschitz function. The original WGAN paper suggested approximating f (analagous to the discriminator) with a neural network parameterized by ϕ and enforcing a Lipschitz constraint by clipping the each entry of the weights ϕ to have magnitude less than ϵ. While this **weight clipping** will ensure that the discriminator f_ϕ is K-Lipschitz for some K, but the exact constant K would depend on the architecture and can be a bit complicated to compute.

Another more elegant way to enforce Lipschitz continuity via **gradient penalties**. Here, we add an additional term $\lambda(\left\|\nabla_x f(x)\right\|_2 - 1)^2$ to our loss for the discriminator, directly encouraging the norm of the gradients to have norm close to 1. While this doesn't strictly enforce 1-Lipschitzness due to it only being a soft penalty on the gradient norm, it is effective and commonly used.

Finally, **spectral normalization** can be used to strictly enforce a 1-Lipschitz constraint on the discriminator f_ϕ. We first note that if two functions f, g are L_1 and L_2 Lipschitz respectively, then their composition $f \circ g$ is $L_1 L_2$-Lipschitz, and we can extend this to any finite composition of Lipschitz functions.

Problem 2: Composition of Lipschitz Functions

If f, g are L_1, L_2 Lipschitz functions, then prove their composition $f \circ g$ is $L_1 L_2$-Lipschitz.

We then note typical neural nets can be written as compositions of functions $f_n \circ \sigma \circ \ldots, \sigma, f_1$, where σ represents the nonlinear activation functions and f_i are some affine layer parameterized by (W_i, b_i). Therefore, one way to ensure f_ϕ is 1-Lipschitz is by ensuring each f_i and σ are all 1-Lipschitz.

We can easily verify that the ReLU activation is 1-Lipschitz (as it is either constant with slope zero or linear with slope 1), so all that remains is to enforce that each linear layer f_i is 1-Lipschitz. Clearly, the Lipschitzness of f_i does not depend on the bias parameter b_i, so we only need to consider the Lipschitzness of

the function $g(x) = W_i x$. The Lipschitz constant K of the linear function g can be written as the supremum

$$\sup_{\|x\|_2=1} \|W_i x\|_2,$$

which we recognize to be the spectral norm $\sigma(W_i)$ (the largest singular value of W_i). Thus, a simple way to enforce 1-Lipschitzness for each linear layer (and thus the whole network) is to simply renormalize $W_i \leftarrow \frac{W_i}{\sigma(W_i)}$ after each gradient update, as the spectral norm $\sigma(W_i)$ is fairly straightforward and cheap to compute.

2 Adversarial Examples

Adversarial examples are inputs that are specially chosen or constructed to fool a model. One reason we study adversarial examples is that they directly offer ways to exploit our learned models. As a real world example, we can imagine someone modifying a stop sign in order to fool a self-driving car system into not stopping, which can potentially lead to crashes.

In this section, we will discuss some common formulations for adversarial attacks, strategies for constructing these adversarial examples, and techniques for mitigating adversarial attacks.

2.1 Formulation

One common way we formulate adversarial attacks is by allowing an additive perturbation δ to a real image x, while restricting the size of the perturbation by enforcing $\|\delta\| < \epsilon$ for some choice of size $\|\cdot\|$ and some budget ϵ. Intuitively, the idea here is that we would like the adversarial examples to be imperceptible to humans and to not change the "true" label, hence the restriction on how much we are allowed to perturb the real input.

To find an adversarial example for some particular loss function and model specified by θ, we want to fool the model (by forcing it to have high loss) by solving the constrained problem

$$\delta^* * = \arg\max_{\delta} L_\theta(\underbrace{x+\delta}_{\text{perturbed input}} , \underbrace{y}_{\text{original label}}) \quad \text{s.t} \|\delta\| \leq \epsilon,$$

and take $x' = x + \delta^*$ as our adversarial example. For images, common choices of the norm $\|\cdot\|$ include the ∞-norm $\|\delta\|_\infty = \max_i |\delta_i|$ or the usual 2-norm $\|\delta\|_2 = \sqrt{\sum_i \delta_i^2}$.

Finally, we note that this definition, while providing a formal definition of adversarial attacks that is reasonable to analyze, does not necessarily align well with adversarial examples in the real world. Real world adversaries are not necessarily limited in how much they perturb the input (at least, often not limited in precisely the way we assume in this formulation).

2.2 Adversarial Attack Strategies

We first consider **white-box attacks**, where we assume the adversary has full access to our model θ. In particular, we assume the adversary can compute gradients $\nabla_x L_\theta(x, y)$, The attacker can then solve the constrained problem with standard first-order constrained optimization techniques like dual gradient ascent to find the optimal perturbation δ^*.

We can also construct much simpler attacks with the **fast gradient sign method** (FGSM). Instead of solving for the optimal δ^* exactly, which can be somewhat computationally expensive, we instead make a first-order approximation to the loss as

$$L'_\theta(x + \delta, y) \approx L_\theta(x, y) + \nabla_x L_\theta(x, y)^T \delta.$$

Using our linearized loss L' instead of the true loss L, we can then easily construct the optimal δ^* using only one gradient evaluation and often a closed form solution for the optimal δ^*.

For example, the optimal solution for a 2-norm constraint would result in the perturbation

$$\delta^* = \epsilon \frac{\boldsymbol{\nabla}_x L_\theta(x, y)}{\left\|\boldsymbol{\nabla}_x L_\theta(x, y)\right\|_2}.$$

For an ∞-norm constraint of ϵ, the optimal perturbation is given by

$$\delta^* = \epsilon \operatorname{sign}(\boldsymbol{\nabla}_x L_\theta(x, y)),$$

hence why this simple attack is known as the fast gradient sign method.

Problem 3: FGSM optimal perturbation

Show that the optimal perturbation δ given a linear loss and ∞-norm constraint is given by the expression above.

In practice, this simple method is very fast and convenient, works well against standard neural networks, but can often be defeated by simple defenses.

We can also consider **black-box attacks**, which do not assume access to the internal workings of the model θ, but instead only observes the model's predictions. Without access to gradients, we can no longer directly run first-order optimization algorithms for the FGSM, as those required taking the gradient of the loss with respect to the input. However, in black box settings, we can still *estimate* the gradient with a finite differences approach.

Recalling that the (single variable) derivative $f'(x)$ is defined as the limit $\lim_{\epsilon \to 0} \frac{f(x+\epsilon)-f(x)}{\epsilon}$, a simple finite differences method estimates gradients by taking small perturbations in each dimension to the input and seeing how the loss changes to approximate that coordinate of the derivative. In practice, this allows us to approximate the gradient with only a moderate number of queries to the model, which we can then use to run our adversarial attacks.

Another way we can construct black box attacks is to learn our own model, perform a white box attack on our own model, and reuse the generated adversarial image to attack the target model. It turns out this strategy often works quite well, and relies on the two models having similar gradients with respect to the input.

2.3 Adversarial Defense

There are many techniques for detecting adversarial examples or making models robust to them. One common way to robustfy networks to adversarial attacks is to incorporate adversarial attacks into the training procedure via a robust loss function, which we refer to as **adversarial training**.

Instead of finding the optimal parameter $\theta^* = \arg\min_\theta \sum_{x,y \in D} L_\theta(x, y)$ we instead minimize the robust loss

$$\theta^* = \arg\min_\theta \sum_{x,y \in D} \max_{\|\delta\| \leq \epsilon} L_\theta(x + \delta, y).$$

Thus, we are explicitly training our network to be robust to the type of attack we expect at test time.

However, adversrial training does come with its drawbacks. Computing the adversarial attack for each input at every training step can slow down training, and often times decreases the accuracy on clean data. Additionally, adversarial training with respect to a certain perturbation constraint does not necessarily generalize to other perturbations.

Problem 4: Adversarial Robustness to Other Pertubations

Suppose we are training a model to classify whether or not there is a stop sign in an image, and our training set consists only of images taken during the day. We use adversarial training to be robust to perturbations δ satisfying $\|\delta\|_\infty \leq 0.5$, and verify that our model robustly generalizes well with this perturbation. Which of the following different perturbation sets should we expect our model to also be robust to?

1. $\{\delta : \|\delta\|_\infty \leq 0.1\}$
2. $\{\delta : \|\delta\|_\infty \leq 1.0\}$
3. $\{\delta : \|\delta\|_2 \leq 0.5\}$
4. $\{\delta : \|\delta\|_2 \leq 1.0\}$
5. Images of stop signs at night, with much dimmer lighting.

CS 182/282A Designing, Visualizing and Understanding Deep Neural Networks
Spring 2021 Sergey Levine **Discussion 12**

This discussion focuses on Meta Learning.

1 Overview of Meta-Learning

Machine learning models often require training with a large number of samples. However, in practice, we may have very little number of samples. Meta learning aims to solve the problem of designing a machine learning model that can quickly adapt to new tasks. In essence, instead of training a model to solve one particular task, we instead train it *learn to learn*. We consider a few concrete meta learning tasks,

- A game bot that can quickly master a new game
- Classifier that can tell whether a given image belongs to a class (possibly unseen during training time) after being provided a few examples of that class.

2 Meta Learning for Supervised Learning

For supervised meta-learning, models are typically trained over a variety of learning tasks. Each task is associated with a labeled dataset $\mathcal{D}$ that contains both feature vectors and true labels. We split each dataset into two parts, $\mathcal{D}^{tr}$ for adapting and a prediction set $\mathcal{D}^{ts}$ for evaluating. For example, in the few-shot classification framework, we consider the case where $\mathcal{D}^{tr}$ contains a few labelled examples for each of classes.

To make predictions on D^{ts} for each task, we first *adapt* our model using the training sets $\mathcal{D}^{\sqcup\nabla}$ before using the updated model to make predictions.

In regular supervised learning, we typically computed $\theta^* = \arg\min_\theta \mathcal{L}_\theta(\mathcal{D}^{tr}) = f_{\text{learn}}(\mathcal{D}^{tr})$ where $\mathcal{L}_\theta(\cdot)$ is our loss function and f_{learn} is our learning algorithm.

However, in the case of meta-learning, we attempt to learn,

$$\theta^* = \arg\min_\theta \sum_{i=1}^{n} \mathcal{L}(\phi_i, \mathcal{D}_i^{ts})$$

where $\phi_i = f_{\text{adapt}}(\theta, \mathcal{D}_i^{tr})$ is the adapted per-task parameters, obtained by running an adaptation procedure on the training set with some "hyperparameters" θ.

Notice that during meta-learning, the model is trained to learn tasks in the meta-training set, and there are two optimization loops – the inner-loop learner, which tries to solve each individual task using the training sets, and the meta-learner, which controls how the inner-loop learner learns in order to maximize how well it generalizes to the test sets.

2.1 Black-Box Meta-Learning

One way of doing black-box meta-learning is to train a recurrent model, like a RNN or an LSTM. The main idea is to read the training set sequentially, then process new inputs from the task. For example, in an image classification setting, this may involve passing in a set of (image, label) pairs of a dataset sequentially, followed by new examples that must be classified.

Here, the meta-learner uses gradient descent and the learner rolls out the recurrent model.

2.2 Non-parametric Meta Learning

Matching Networks Task of Matching Networks is to learn a classifier for any given small training set. This classifier defines some probability distribution over output labels y given a test sample x, and the classifier output is defined as the sum of labels of support weighted by an attention kernel.

$$p_\theta(y_j^{ts}|x_j^{ts}, \mathcal{D}_i^{tr}) = \sum_{k:y_k^{tr}=y_j^{ts}} p_{nearest}(x_k^{tr}|x_j^{ts})$$

The attention kernel depends on two embedding functions f and g. f is the embedding function for encoding the test sample and g is the embedding function for encoding the training set $\mathcal{D}^{tr}$.

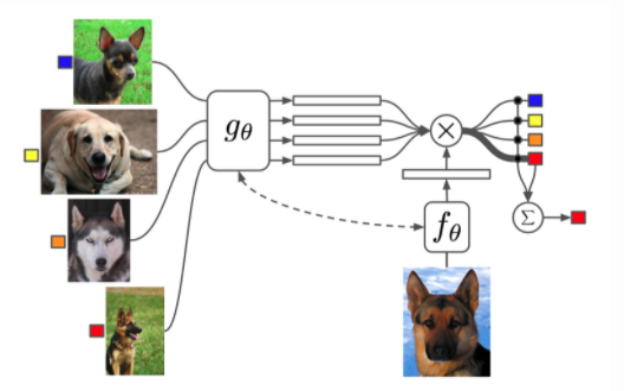


Figure 1: Architecture of Matching Networks, where f is the embedding function for encoding the test sample and g is the embedding function for encoding the training set $\mathcal{D}^{ts}$

Attention weight is defined as,

$$p_{nearest}(x_k^{tr}|x_j^{ts}) \propto \exp\Big(g(x_k^{tr}, \mathcal{D}_i^{tr})^\top f(x_j^{ts}, \mathcal{D}_i^{tr})\Big)$$

Prototypical Networks Prototypical Networks two embedding functions f and g. f is the embedding function for encoding the test sample and g is the embedding function for encoding the training set $\mathcal{D}^{tr}$. We define a *prototype* feature for every class $c \in C$, as the mean vector of the embedded training samples in the class,

$$c_y = \frac{1}{N_y} \sum_{k:y_k^{tr}=y} g(x_k^{tr})$$

Here, the distribution over classes for a given test input is given by,

$$p_\theta(y|x_j^{ts}, \mathcal{D}_i^{tr}) \propto \exp\Big(c_y^\top f(x_j^{ts})\Big)$$

2.3 Model-Agnostic Meta-Learning (MAML)

One of the biggest success stories of transfer learning has been initializing vision networks using pre-training. When approaching any new vision task, a well-known paradigm is to first collect labeled data for the task, acquire a network pre-trained on ImageNet classification, and then fine-tune the network on the collected data. This way, the neural network can learn new image-based tasks more efficiently by drawing on the features learned by the network pretrained on ImageNet.

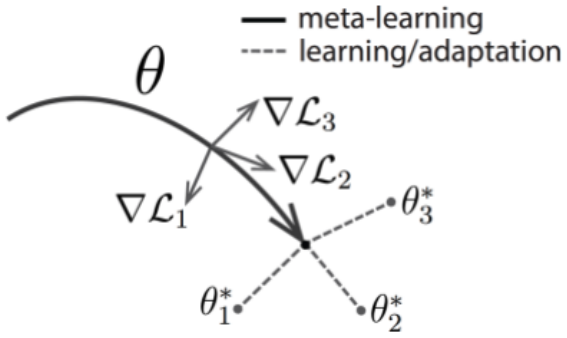


Figure 2: During the course of meta-learning (bold line), MAML optimizes for a set of parameters such that when a gradient step is taken with respect to a task $\mathcal{T}_i$ (gray lines), parameters are close to the optimal parameters θ_i^* for task $\mathcal{T}_i$

Model-Agnostic Meta-Learning (MAML) extends this idea by explicitly optimizing the "pretrained" network to be able to adapt in a few gradient steps to a variety of tasks. MAML directly optimizes over the finetuning gradient descent procedures for each task in order to compute an optimal initialization.

We let our model be f_θ with model parameters θ, and we assume we are given tasks $\mathcal{T}_i$ and associated datasets. Then, when we are given a new task, we can update the model parameters by a few gradient steps,

$$\theta_i^{'} = f_\theta(\mathcal{D}_i^{tr}) = \theta - \alpha\nabla_\theta\mathcal{L}_i(\theta, \mathcal{D}_i^{tr})$$

Notice this only optimizes for one task. To find a good generalization across a variety of tasks, we update θ across all tasks,

$$\theta \leftarrow \theta - \beta\sum_i \mathcal{L}(\theta - \alpha\nabla_\theta\mathcal{L}(\theta, \mathcal{D}_i^{tr}), \mathcal{D}_i^{ts})$$

A summary of the algorithm can be seen below:

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta\mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha\nabla_\theta\mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta\nabla_\theta \sum_{\mathcal{T}_i\sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

Figure 3: This algorithm uses slightly different notations than ours, with $\nabla_\theta\mathcal{L}_{\mathcal{T}_i}(f_\theta) = \nabla_\theta\mathcal{L}_i(\theta, \mathcal{D}_i^{tr})$

Problem 1: Gradient Update of MAML

Derive the gradient update for MAML outer loop. Let $\mathcal{L}^{(0)}$ and $\mathcal{L}^{(1)}$ represent the loss at the first and second mini-batches. Assume in the inner loop, we have already performed k inner gradient steps,

$$\begin{aligned}\theta_0 &= \theta_{meta} \\ \theta_1 &= \theta_0 - \alpha \nabla_\theta \mathcal{L}^{(0)}(\theta_0) \\ &\dots \\ \theta_k &= \theta_{k-1} - \alpha \nabla_\theta \mathcal{L}^{(0)}(\theta_{k-1})\end{aligned}$$

You are given your meta-objective in the outer loop after sampling a new data batch is,

$$\theta_{meta} \leftarrow \theta_{meta} - \beta g_{MAML}$$

Derive g_{MAML}.
Hint: Use the chain rule.

MAML has a number of advantages. MAML does not make any restrictions or assumptions on the form of model beyond being learned through gradient descent. It is also does not need to introduce any additional parameters into the network, and only uses gradient descent in both the inner and outer loops. Lastly, the authors showed it can be easily applied to a number of domains, and the method substantially outperformed a number of existing approaches on popular few-shot image classification benchmarks.

3 Meta-Reinforcement Learning

The meta reinforcement learning setup is analogous to the meta learning set up, except we maximize expected rewards, rather than minimizing loss.

$$\theta^* = \arg\max_\theta \sum_{i=1}^{n} \mathbb{E}_{\pi_{\phi_i}(\tau)}[R(\tau)]$$

where $\phi_i = f_\theta(\mathcal{M}_i)$ and $\{\mathcal{M}_i\}_{i=1}^n$ is the set of training MDPs.

The overall configuration of meta RL algorithms similar to an ordinary RL algorithm, with the key difference between that the policy now has to, in some way, incorporate past experiences in the current MDP to adapt and perform well in the current MDP. This can be viewed as solving a partially observed MDP, where we do not directly observe what MDP we are currently solving and have to reason about the MDP using the current trajectories.

Keeping Track of History in Meta-RL One natural approach to meta-RL is to feed a history of experiences in the current MDP into the model, so the policy can adapt to the current MDP. An intuitive model that arises from this intention is using recurrent neural networks or its variants (like an LSTM).

Given a new test MDP, $\mathcal{M}$, our policy would simply take trajectories in the MDP, updating the RNN hidden states as it processes these new experiences.

During training, our algorithm would proceed as follows:

1. Sample new MDP $\mathcal{M}_i$
2. Reset the hidden state of the model
3. Collect multiple trajectories, updating hidden states as we go, and finally update the model weights using the policy gradient. Do not reset the hidden states between episodes.
4. Repeat from Step 1

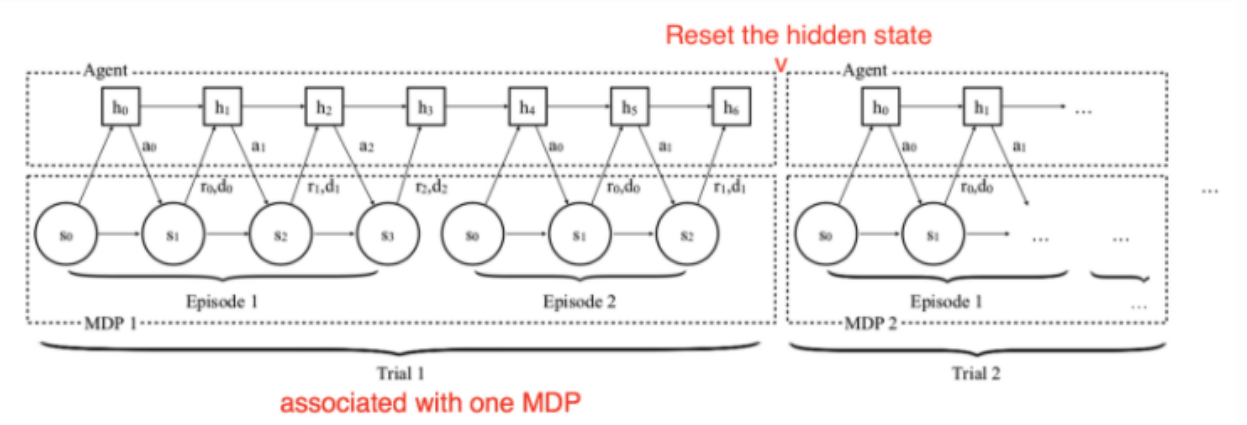


Figure 4: Model as described in the RL^2 paper. This is an illustration of the procedure of the model interacting with a series of MDPs in training time. Between episodes in each MDP, model weights are not reset, but across MDPs, they are reset.

Numerous architectures for summarizing past experiences have been developed for meta RL, including usage of standard RNN (LSTM) architecture, attention and temporal convolutions and parallel permutation-invariant context encoders.

Problem 2: Resetting the Hidden State

In the algorithm, we reset the hidden states between tasks, but not between episodes.

MAML for RL In using MAML for reinforcement learning, we specify a new loss for task $\mathcal{T}_i$ and model f_ϕ. We let $q(x_{t+1}|x_t, a_t)$ be the transition distribution, and loss $\mathcal{L}_{\mathcal{T}_i}$ correspond to the (negative) reward function r. Then the entire task $\mathcal{T}_i$ is a MDP with horizon H, and learner is allowed to query a limited number of sample trajectories for few-shot learning.

Then, our loss takes the form,

$$\mathcal{L}_{\mathcal{T}_i}(f_\phi) = -\mathbb{E}_{s_t, a_t \sim f_\phi, q_{\mathcal{T}_i}} \left[\sum_{i=1}^{H} r_i(x_t, a_t) \right]$$

Since the expected reward is not directly differentiable like in our usual supervised learning settings, we use policy gradient methods to estimate the gradient for model gradient updates and meta-optimization.

Below is the general algorithm for MAML for reinforcement learning,

Algorithm 3 MAML for Reinforcement Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample K trajectories $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using f_θ in $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 4
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample trajectories $\mathcal{D}_i' = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using $f_{\theta_i'}$ in $\mathcal{T}_i$
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 4
11: **end while**

Figure 5: General MAML for RL algorithm. Equation 4 is the loss function as stated above